The CAML reference manual

Pierre Weis, Maria Virginia Aponte, Alain Laville, Michel Mauny, Ascander Suarez

The CAML Reference Manual

Pierre Weis

María-Virginia Aponte

Alain Laville

Michel Mauny

Ascánder Suárez

Projet Formel

INRIA-ENS

Version 2.6.1

The CAML Reference Manual

Projet Formel

INRIA-ENS

Version 2.6.1

Contents

Part I

Demonstration

Chapter 1

CAML in action

We start this manual by a short demonstration of the capacity of the language to help the user to construct a prototype of an application. Our example will be an interactive desk calculator. We will introduce the basic notions of the language from scratch. But, this is not intended to be a tutorial, rather a brief recall of the main features (for a tutorial see the CAML PRIMER).

First, some advertising!

1.1 Interfaces and facilities

CAML is a rich system:
Facilities include

- Secure and easy parser generation with the "grammar" construct of the language.
- Separate compilation with a simple modules feature.
- Completely integrated and typechecked macro facilities.
- Compiler pragmas with automatic environment checking.
- Debugging tools (a sophisticated trace package).
- Automatic documentation of source files.
- Scanner generation.
- Easy development of application systems with high level tools.

Numerous interfaces are provided:

- Algebraic computation can be done by a specialised process via an interface with the "maple" system.

- Graphics use an interface with Postscript: CAML outputs code for the postcript interpreter.
- An Xdr interface allows communication between CAML and external C functions.
- CAML interacts nicely with Latex, since the language is able to expand examples written in CAML inside a latex file. Moreover a CAML program can be turned into a latex file via another existing utility (in the spirit of WEB).

1.2 Starting

CAML is an interactive system:

- The user types in an expression and
- CAML responds by printing the value of that expression, together with its type:

```
#1;;
1 : num
```

Predefined values are:

- integers which belong to the type num (in fact num is {integers} ∪ {rational numbers} ∪ {floats})
- boolean values (type "bool")
- strings (type "string")

```
#1+2;;
3 : num

#1+2=2+1;;
true : bool

#"The CAML"^" language";;
"The CAML language" : string
```

1.3 Pairs and Functions

Some type constructors are given to build new types from basic types. The only predefined type constructors are * (cartesian product) and -> (infix notation) (building functional types). (In fact, the constructor vect is also predefined).

```
#1+2,3+4;;
(3,7) : (num * num)

#1+2,3=4;;
(3,false) : (num * bool)

#fst (1,true);;
1 : num

#snd (1,true);;
true : bool

#1,2,3;;
(1,2,3) : (num * num * num)

#fst (1,2,3);;
1 : num

#snd (1,2,3);;
(2,3) : (num * num)

#fst (snd (1,2,3));;
2 : num
```

1.4 Declarations

1.4.1 Global declarations

Declarations are bindings of identifiers to values:

```
#let x=1+2;;
Value x = 3 : num

#let y=x+4;;
Value y = 7 : num

#x*y;;
21 : num
```

Variables that are declared using the `let` construct are usable in expressions. Their binding is not modifiable (unless they are of the type of modifiable values). They may only be redefined by a new `let`. But this defines a new variable, putting the previous one "out of scope".

1.4.2 Local declarations

These disappear at the end of the evaluation of the phrase they appeared in:

```
#let x="The CAML " and y="language" in x^y;;
"The CAML language" : string

#y*4 where y=20;;
80 : num

#x;;
3 : num
```

In this case, the global value of x is still 3, and has not been modified by the local declaration of x.

1.5 Functions as values

1.5.1 Defining a function

To declare a function is simple and natural:

```
#let f (x) = 2*x+1;;
Value f = <fun> : (num -> num)

#let g(x,y) = 2*x+y;;
Value g = <fun> : (num * num -> num)
```

The system doesn't know how to print functional values because their internal representation is machine code. Application of a function to its argument is easy as well:

```
#f (2);;
5 : num

#g (2,3);;
7 : num
```

A relaxed syntax exists to use and define functions. Bound arguments and applications are denoted by simple juxtaposition:

```
#let f x = 2*x+1;;
Value f = <fun> : (num -> num)

#f 2;;
5 : num
```

1.5.2 Anonymous functions

A function is a first class citizen:

```
#function x -> 2*x+1;;
<fun> : (num -> num)
```

function may be abbreviated by fun:

```
#fun x -> 2*x+1;;
<fun> : (num -> num)

#function (x,y) -> 2*x+y;;
<fun> : (num * num -> num)
```

These are alternative ways to define f and g, as identifiers bound to values which are functions:

```
#let f = function x -> 2*x+1;;
Value f = <fun> : (num -> num)

#let g = function (x,y) -> 2*x+y;;
Value g = <fun> : (num * num -> num)
```

1.5.3 Higher-order functions

The -> type constructor may take any type arguments ty1 and ty2 to build a legal type ty1 -> ty2. ty1 and ty2 may be functional themselves, and this leads to functions having functions as arguments and (or) results.

```
#let h = function x -> (function y -> 2*x+y);;
Value h = <fun> : (num -> num -> num)

#let k = h 1;;
Value k = <fun> : (num -> num)

#k 3;;
5 : num

#h 2 3;;
7 : num
```

The above function named h is the *curried* version of the function we named g before. Expression h 2 3 has to be read as (h 2) 3 following the usual λ−calculus convention.

1.5.4 Recursive functions

The `let rec` and `where rec` constructs allows the definition of "recursive" functions, that is, functions defined in terms of themselves:

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

#fact 5;;
120 : num
```

1.5.5 Definition by "case"

One may use case analysis on the value of the argument to define the same function as in:

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

There are alternative syntaxes for defining recursive functions:

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

#fact 20;;
2432902008176640000 : num
```

The special symbol _ may be used to stand for "any other case":

```
#let is_one = function 1 -> true | _ -> false;;
Value is_one = <fun> : (num -> bool)

#is_one 1;;
true : bool

#is_one 0;;
false : bool
```

1.5.6 Infix identifiers

It is possible to decide that an identifier should be parsed as infix (that is between two tokens as + in 1+2). There are already predefined such identifiers: +, *, ... but the user may declare his own infixes.

After an infix declaration, one has to either write the infix identifier as an infix, or to use the `prefix` keyword, to use it as prefix identifier.

```
##infix power;;
Ident power is now parsed as an infix
Directive () : unit

#let rec x power y =
#    if x=0 then 0
#    if y=0 then 1
#    else x*(x power (y-1));;
Value prefix power = <fun> : (num * num -> num)

#2 power 8;;
256 : num
```

1.6 Typing and polymorphism

The CAML toplevel tries to give a type to each expression typed in by the user. The typing phase does not need any type annotation in the program to give it a type. Knowing types of basic values and primitive operations and using typing rules for application and functions, the type-checker produces the most general type for each expression (Milner, 1978).

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

'a, 'b, ... are *type variables*. A type variable stands for *any* legal type. As an example we can *instantiate* 'a with 'b -> 'b, and thus apply the function identity to itself:

```
#let id = identity identity;;
Value id = <fun> : ('a -> 'a)

#id 1;;
1 : num

#(* Parsed as (identity (identity)) (1) *)
#identity identity 1;;
1 : num

##infix o;;
Warning: o is already parsed as an infix
Ident o is now parsed as an infix
Directive () : unit

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

The type of the o function (mathematical function composition) means that the domain of the f argument has to be the same as the codomain of the g argument, and that the last argument (x) has to belong to the domain of the g argument. If these constraints are not satisfied in an expression, then the type-checker will reject that expression.

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

#g;;
<fun> : (num * num -> num)

#f o g;;
<fun> : (num * num -> num)

#g o f;;

line 1: ill-typed phrase, the variable f of type
(num -> num) cannot be used with type instance
(num -> num * num) in g o f
1 error in typechecking

Typecheck Failed
```

The type of the function o is said to be "polymorphic": type variables occur in it.

A general functional to define the curried version of a function having a pair of arguments is easily defined

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

The converse function is not more complex:

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

Notice that these functions interact nicely to give instances of the identity function:

```
#curry o uncurry;;
<fun> : (('a -> 'b -> 'c) -> 'a -> 'b -> 'c)

#uncurry o curry;;
<fun> : (('a * 'b -> 'c) -> 'a * 'b -> 'c)
```

1.7 Defining new types

It is possible to define new types which may be mutually recursive, by defining their constructors and giving the types of their arguments.

Assume that in the language we are modelling, an expression can be either

- a number (thus a constant expression).
- a variable name
- a binding introduced by a `let` keyword
- an unary operation
- a binary operation

Then the data type for these expressions may be:

```
#type expression =
#     Constexp of num
#   | Varexp of string
#   | Letexp of string * expression * expression
#   | Unopexp of string * expression
#   | Binopexp of string * expression * expression
#;;
Type expression defined
     Constexp : (num -> expression)
   | Varexp : (string -> expression)
   | Letexp :
     (string * expression * expression -> expression)
   | Unopexp : (string * expression -> expression)
   | Binopexp :
     (string * expression * expression -> expression)
```

These types may be polymorphic, for example, the type `list` is defined in the CAML system by:

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

Thus the empty list is a polymorphic object:

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

With our new constructors we may define new values:

```
#Constexp 1;;
(Constexp 1) : expression

#Constexp 2.0e+10;;
(Constexp (2.0e+10)) : expression

#Binopexp("+", Constexp 1, Constexp 2);;
(Binopexp ("+",(Constexp 1),(Constexp 2))) : expression
```

1.7.1 Function call "by-pattern"

We define now a function which formally simplifies the expressions of our tiny language. This function does a case analysis on its "expression" argument, and according to the shape of this expression returns a simplified expression: this defines a set of simplification rules for expressions.

For instance we may translate the mathematical rule $-0 == 0$ with the CAML pattern matching case:

```
Unopexp ("-",Constexp 0) -> Constexp 0
```

Here is the whole definition of the simplificator

```
#let rec simpl = function
#(* UNARY MINUS *)
#   Unopexp ("-",Constexp 0) -> Constexp 0
# | Unopexp ("-",Unopexp ("-",x)) -> simpl x
# | Unopexp ("-",x) as arg -> check (arg,Unopexp ("-",simpl x))
#(* ADDITION *)
# | Binopexp ("+",Constexp m,Constexp n) -> Constexp (m+n)
# | Binopexp ("+",Constexp 0,x) -> simpl x
# | Binopexp ("+",x,Constexp 0) -> simpl x
# | Binopexp ("+",x,y) as arg ->
#    check (arg,Binopexp ("+",simpl x,simpl y))
#(* SUBTRACTION *)
# | Binopexp ("-",Constexp m,Constexp n) -> Constexp (m-n)
# | Binopexp ("-",Constexp 0,x) -> Unopexp ("-",simpl x)
# | Binopexp ("-",x,Constexp 0) -> simpl x
# | Binopexp ("-",x,y) as arg ->
#    check (arg,Binopexp ("-",simpl x,simpl y))
#(* MULTIPLICATION *)
# | Binopexp ("*",Constexp m,Constexp n) -> Constexp (m*n)
# | Binopexp ("*",Constexp 0,x) -> Constexp 0
# | Binopexp ("*",x,Constexp 0) -> Constexp 0
# | Binopexp ("*",Constexp 1,x) -> simpl x
# | Binopexp ("*",x,Constexp 1) -> simpl x
# | Binopexp ("*",x,y) as arg ->
```

```
#    check (arg,Binopexp ("*",simpl x,simpl y))
#(* DIVISION *)
# | Binopexp ("/",Constexp m,Constexp n) -> Constexp (m/n)
# | Binopexp ("/",Constexp 0,x) -> Constexp 0
# | Binopexp ("/",x,Constexp 0) -> Binopexp
#                                    ("/",Constexp 1,Constexp 0)
# | Binopexp ("/",Constexp 1,x) -> Binopexp
#                                    ("/",Constexp 1,simpl x)
# | Binopexp ("/",x,Constexp 1) -> simpl x
# | Binopexp ("/",x,y) as arg ->
#    check (arg,Binopexp ("/",simpl x,simpl y))
#(* LOCAL DECLARATIONS *)
# | Letexp (var,exp1,exp2) ->
#            let e1 = simpl exp1 and e2 = simpl exp2
#            in Letexp (var,e1,e2)
# | x -> x
#
#and check (e1,e2) = if e1 = e2 then e1 else simpl e2
#;;
Value simpl = <fun> : (expression -> expression)
Value check = <fun> :
      (expression * expression -> expression)
```

There is also a `case ... of ...` construct. One may have written the previous function with:

```
let rec simpl expr =
case expr of Unopexp ("-", Constexp 0) -> Constexp 0
   | ...
     ...;;
```

`case ... of ...` is syntactically equivalent to `match ... with ...`:

```
#match simpl (Binopexp("+", Constexp 1, Constexp 1))
#with Constexp 2 -> true
# | _ -> false
#;;
true : bool
```

It is also possible to define product types and, indeed, cartesian product is just a special case of product type and may be defined by:

```
type 'a * 'b = {Fst:'a; Snd:'b};;
```

and projections are:

```
let fst p = p.Fst and snd p = p.Snd;;
```

```
      .Head : ('a Stream -> 'a)
    ; .Tail : ('a Stream -> 'a Stream)
Type Frozen defined
      Freeze : ('a -> 'a Frozen)

#let rec (Freeze Nat) = (Freeze {Head = 0; Tail = map succ Nat}
# where rec map f = function
#  {Head=h; Tail=t} -> {Head=f h; Tail=map f t});;
Value Nat = {Head=0; Tail=*} : num Stream

#Nat.Tail.Head;;
1 : num

#(* Updating is automatic *)
#Nat;;
{Head=0; Tail={Head=1; Tail=*}} : num Stream
```

- I/O (channels): the basic notion of I/Os in CAML is the stream of characters.

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

- Autoload functions whose code is loaded from a file at the first call to the function.

```
#autoload toto: num -> num and tutu: void -> bool
#from "my_file"
#;;
Warning: (autoload) my_file.lo not found
Forward tutu : (unit -> bool)
Forward toto : (num -> num)
Directive () : unit
```

1.10 Concrete syntax manipulation

Using a concrete syntax for "expressions":

```
#let MLconst_to_MLvar =
#function MLconst (mlstring s) -> MLvar s
# | _ -> failwith "wrong \"IDENT\" lexeme"
#;;
Value MLconst_to_MLvar = <fun> : (ML -> ML)

#grammar Expr =
```

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

Then naming the grammar and an entry of it, calls the corresponding parser:

```
#<:Expr:Expr<1+2>>;;
(Binopexp ("+",(Constexp 1),(Constexp 2))) : expression

#let x=<<1>>;;
Value x = (Constexp 1) : expression

#let y=<<x+2*3+y>>;;
Value y =
      (Binopexp
       ("+",(Varexp "x"),
        (Binopexp
         ("*",(Constexp 2),
          (Binopexp ("+",(Constexp 3),(Varexp "y")))))))) :
      expression
```

If an escape has been defined in the grammar, interaction with the metalanguage is allowed:

```
#<<^x + ^y>>;;
(Binopexp
 ("+",(Constexp 1),
  (Binopexp
   ("+",(Varexp "x"),
    (Binopexp
     ("*",(Constexp 2),
      (Binopexp ("+",(Constexp 3),(Varexp "y"))))))))) :
 expression
```

One can even use its concrete syntax within patterns:

```
#let is_zero = function
#      <<0>> -> true
#    | <<^x+^y>> -> message "this is an addition!"; false
#    | _ -> false;;
Value is_zero = <fun> : (expression -> bool)
```

A small pretty-printer for elements of type **expression**:

```
#let print_expression =
#
```

```
#let rec print_exp = function
#     <<{^Constexp n^}>> -> print_num n
#   | <<{^Varexp v^}>> -> print_string v
#   | <<let ^v = ^expr1 in ^expr2>> ->
#       print_string "let"; print_break(1,0);
#       print_string v; print_break(1,0);
#       print_string "="; print_break(1,0);
#       print_exp expr1; print_break(1,0);
#       print_string "in"; print_break (1,0);
#       print_exp expr2
#   | <<^exp1 ^op ^exp2>> -> print_string"(";
#       print_exp exp1; print_break(1,0);
#       print_string op; print_break(1,0);
#       print_exp exp2; print_string ")"
#   | <<- ^exp>> -> print_string "-"; print_exp exp
#   | _ -> failwith "Incorrect expression"
#    (* When given another unary operator *)
#
# in function exp -> open_hovbox 0; print_exp exp; close_box()
#;;
Value print_expression = <fun> : (expression -> unit)

#print_expression (simpl <<2*2-(2*3)+x*y+(-1)/(2-3)>>);
#print_newline();;
(2 * (2 - (6 + (x * (y + (-1 / -1))))))
() : unit
```

Make our expressions pretty-printer available for the CAML system itself:

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

#simpl <<let x=2*2-(2*3)+x*y+(-1)/(2-3) in x*x-2>>;;
let x = (2 * (2 - (6 + (x * (y + (-1 / -1)))))) in (x * (x
- 2)) : expression
```

We now rewrite `simpl` in a more convenient way, using concrete syntax in each sides of match rules:

```
#let rec simpl = function
#(* UNARY MINUS *)
#     <<-0>> -> <<0>>
#   | <<-(- ^x)>> -> simpl x
#   | <<-^x>> as arg -> check(arg,<<-{^simpl x^}>>)
#(* ADDITION *)
```

```
#     | <<{^Constexp m^}+{^Constexp n^}>> -> Constexp (m+n)
#     | <<0+^x>> -> simpl x
#     | <<^x+0>> -> simpl x
#     | <<^x+^y>> as arg -> check(arg,<<{^simpl x^}+{^simpl y^}>>)
#(* SUBTRACTION *)
#     | <<{^Constexp m^}-{^Constexp n^}>> -> Constexp (m-n)
#     | <<0-^x>> -> <<-{^simpl x^}>>
#     | <<^x-0>> -> simpl x
#     | <<^x-^y>> as arg -> check(arg,<<{^simpl x^}-{^simpl y^}>>)
#(* MULTIPLICATION *)
#     | <<{^Constexp m^}*{^Constexp n^}>> -> Constexp (m*n)
#     | <<0*^x>> -> <<0>>
#     | <<^x*0>> -> <<0>>
#     | <<1*^x>> -> simpl x
#     | <<^x*1>> -> simpl x
#     | <<^x*^y>> as arg -> check(arg,<<{^simpl x^}*{^simpl y^}>>)
#(* DIVISION *)
#     | <<{^Constexp m^}/{^Constexp n^}>> -> Constexp (m/n)
#     | <<0/^x>> -> <<0>>
#     | <<^x/0>> -> <<1/0>>
#     | <<1/^x>> -> <<1/{^simpl x^}>>
#     | <<^x/1>> -> simpl x
#     | <<^x/^y>> as arg -> check(arg,<<{^simpl x^}/{^simpl y^}>>)
#(* LOCAL DECLARATIONS *)
#     | <<let ^var = ^exp1 in ^exp2>> ->
#             let e1 = simpl exp1 and e2 = simpl exp2
#             in <<let ^var = ^e1 in ^e2>>
#     | x -> x
#
#and check(e1,e2) = if e1=e2 then e1 else simpl e2;;
Value simpl = <fun> : (expression -> expression)
Value check = <fun> :
      (expression * expression -> expression)
```

1.11 Debugging tools

Trace facilities are provided

```
#trace "fact";;
() : unit

#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
```

Numerous options are available, through a predefined concrete syntax for trace orders:

```
#<:Trace<
# fact x : trace x from fact with predicate (fun x -> x > 3);
#          print x with
#           (fun x -> print_string ("Number "^(string_of_num x)));
#          before x do
#           (fun x -> message "Fact is called from fact !");
#          after x do
#           (fun x result ->
#            message ("Result was: "^(string_of_num result)))
#>>;;
Warning: fact already traced
() : unit

#fact 5;;
Fact is called from fact !
Fact is called from fact !
<0>fact (*from [0]fact*) (Number 4) -->
Fact is called from fact !
Fact is called from fact !
Fact is called from fact !
Fact is called from fact !
Result was: 1
Result was: 1
Result was: 2
Result was: 6
<0>fact (*from [0]fact*) (.)   = 24
Result was: 24
Result was: 120
120 : num
```

Information about identifiers can be given by the system:

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

and identifiers can be looked for in the global symbol table:

```
#search_variable"act";;
fact fact_iter do_act   extract_string subtract subtractq
hash_subtract
() : unit
```

1.12 A simple calculator

- We use abstract syntax trees of arithmetic expressions.
- We define abstract syntax trees for toplevel expressions.
- Then we define their denotational semantics.
- At last, a toplevel loop is defined.

Let us recall definition of abstract syntax trees:

```
type expression =
     Constexp of num
   | Varexp of string
   | Letexp of string * expression * expression
   | Unopexp of string * expression
   | Binopexp of string * expression * expression
;;
```

We add now the abstract syntax of toplevel expressions, which may be either expressions to be evaluated or declarations of globals:

```
#(* Type of toplevel expressions *)
#type toplevel_expression =
# Exp of expression
# | Global_let of string * expression
# | End
#;;
Type toplevel_expression defined
     Exp : (expression -> toplevel_expression)
   | Global_let :
     (string * expression -> toplevel_expression)
   | End : toplevel_expression
```

And the complete grammar associated with the desk calculator is:

```
#grammar for values Dc =
#
# delimitor
#string is "\""
#comment is "%"
# ;
# precedences
#left "+";
#left "*"
# ;
#
#rule
#
#entry Top_exp = parse Top_exp1 te; Literal "?" -> te
#
#and Top_exp1 = parse
#    Literal "let"; IDENT s; Literal "="; Expr e -> Global_let(s,e)
#  | Expr e -> Exp e
#  | Literal "." -> End
#
#and Expr = parse
#    NUM n -> Constexp n
#  | IDENT s -> Varexp s
#  | Literal "let"; IDENT s; Literal "=";
#    Expr e1; Literal "in"; Expr e2 -> Letexp (s,e1,e2)
#  | Unopexp e -> e
#  | Binopexp e -> e
#  | Literal "("; Expr e; Literal ")" -> e
#
#and Unopexp = parse
#    Literal ("-" as minus); Expr e -> Unopexp (minus,e)
#
#and Binopexp = parse
#    Expr e1; Binop bop; Expr e2 -> Binopexp (bop,e1,e2)
#
#and Binop = parse
#    Literal ("+" as bop) -> bop
#  | Literal ("*" as bop) -> bop
#  | Literal ("-" as bop) -> bop
#  | Literal ("/" as bop) -> bop
#;;
Calling Yacc ... ......................................................
Value Dc = <fun> : (string -> Parsers)
```

```
Grammar Dc for values defined
  entry Top_exp : toplevel_expression
```

From this grammar we define a parser:

```
#let parse_dc_top_exp () =
# match eval_syntax (((Dc "Top_exp").Parse) ()) with
#  dynamic (d : toplevel_expression) -> d;;
Warning: 1 partial match in this phrase
Value parse_dc_top_exp = <fun> :
      (unit -> toplevel_expression)
```

1.13 Denotational semantics of a small desk calculator

We define the global environment of the calculator and its associated manipulation tools:

```
#let global_env = ref ([]: (string * num) list);;
Value global_env = (ref []) : (string * num) list ref

#let init_env () = global_env:=[]; ()
#and store_in_global_env (var,val as binding) =
#    global_env:=binding::!global_env;
#    print_string (var^" gets bound to ");
#    val
#;;
Value init_env = <fun> : (unit -> unit)
Value store_in_global_env = <fun> : (string * num -> num)
```

Now we define the semantics of expressions:

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

#let rec semexp env = function
#    Constexp n -> n
#  | Varexp var -> assoc var env ? raise UNBOUND var
#  | Letexp (var,exp,exp') ->
#          let n = semexp env exp in
#          semexp ((var,n)::env) exp'
#  | Unopexp (name,exp) ->
#          semunop name (semexp env exp)
#  | Binopexp (name,exp,exp') ->
#          sembinop name (semexp env exp,semexp env exp')
```

```
#
#and semunop = function
#    "-" -> minus | s -> failwith ("unknown unop"^s)
#
#and sembinop = function
#    "+" -> prefix +
#  | "*" -> prefix *
#  | "-" -> prefix -
#  | "/" -> prefix /
#  | s -> failwith ("unknown unop"^s)
#;;
Value semexp = <fun> :
      ((string * num) list -> expression -> num)
Value semunop = <fun> : (string -> num -> num)
Value sembinop = <fun> : (string -> num * num -> num)
```

and the semantics of toplevel expressions

```
#exception END_DC;;
Exception END_DC defined

#let global_sem env = function
#    Exp exp -> semexp env exp
#  | Global_let (var,exp) -> let n = semexp env exp in
#                            store_in_global_env (var,n)
#  | End -> raise END_DC
#;;
Value global_sem = <fun> :
      ((string * num) list -> toplevel_expression -> num)

#(* To compute the value of a toplevel expression *)
#let compute exp = global_sem !global_env exp;;
Value compute = <fun> : (toplevel_expression -> num)
```

The calculator itself and its toplevel loop:

```
#let dc_step e =
# try
# print_num (compute e);
# print_newline();print_newline()
# with UNBOUND var -> message ("*** "^var^" is unbound! ***");
#                     print_newline()
#| END_DC -> message "Bye" reraise
#| failure s -> message s; print_newline()
#| break -> message "Bye"; raise END_DC
#| _ -> message "ERROR"; print_newline();;
```

```
Value dc_step = <fun> : (toplevel_expression -> unit)

#let dc_top e =
# try dc_step e with _ -> ();;
Value dc_top = <fun> : (toplevel_expression -> unit)

#let dc_eval = dc_step o parse_dc_top_exp;;
Value dc_eval = <fun> : (unit -> unit)

#let dc_loop () =
# try while true do dc_eval () done
# with
#  END_DC -> set_prompt "#";;
Value dc_loop = <fun> : (unit -> unit)

#let dc () =
# print_newline(); set_prompt "-> "; init_env();
# dc_loop()
#;;
Value dc = <fun> : (unit -> unit)

#(* Create a load function for dc, using the function
#   load_with_loop, available from the CAML system *)
#let load_dc_in_channel = load_with_loop dc_loop;;
Value load_dc_in_channel = <fun> : (in_channel -> unit)

#let load_dc s =
#    let is = open_in s in
#    load_dc_in_channel is;
#    close_in is;
#    message ("dc file "^s^" loaded");;
Value load_dc = <fun> : (string -> unit)
```

Let's give a "dc" session as an example:

```
(* Entering calculator's toplevel loop *)
dc();;

-> 3 ?
3

-> x ?
*** x is unbound! ***

-> let x=3 ?
```

```
x gets bound to 3

-> let x=10 in x*x ?
100

-> x-x ?
0

-> let x=x+1 ?
x gets bound to 4

-> let y = 11111111111111111111111111111111111111111+1 ?
y gets bound to 11111111111111111111111111111111111111112

-> y*1.5 ?
ERROR

-> (* Exiting (notice that comments are still available) *)
-> .?
Bye

#1;;
1 : num
```

When then load the following simple "dc" file:

```
1+2 ?
. ?
```

using:

```
#load_dc "./aux/dc";;
3

Bye
dc file ./aux/dc loaded
() : unit
```

Part II

System Manual

Chapter 2

Lexical conventions of CAML

Beware:
-- All Caml lexical units are sequences of *up to 256 characters.*

2.1 Idents

Identifiers are names used to denote values, types and exceptions. Identifiers belong to the lexical class <*Ident*>.

Identifiers are sequences of letters and digits, and the characters "_" and "'", starting with a letter. Lower and upper case letters are distinct. Like all lexical units, an identifier can not be longer than 256 characters.

Thus `x'2`, `don't_be_silly` and `substitute_double_'_by_single_'` are perfectly legal identifiers.

Some identifiers may be legal as well, but not available to the user since they are reserved keywords, as for example "`let`" or "`match`".

Beware:
-- The empty list "`[]`" symbol is also an identifier.
-- Type variables are identifiers preceded by a quote character "'", e.g.: `'a`.

2.2 Infixes

Infixes are special identifiers which may appear only at some predefined place in the syntax of CAML phrases, namely *between* two syntactic elements (like "+" in `1 + 2`).

<*Infix*> is a dynamic lexical unit, the user can add or delete infixes, using pragmas or directives "`infix`" and "`uninfix`" which set on or off the infix status of an identifier.

```
#infix UNION;;
#pragma infix "UNION";;
```

The action of these commands is taken into account in the phrase following their execution. Each "`Infix`" becomes an "`Ident`" when preceded by the keyword "`prefix`", as for example in "`prefix o`".

Finally, the following keywords can also be used as identifiers using the keyword "`prefix`": "`*`", "`/`", "`+`", "`-`", "`::`", "`@`", "`^`" "`not`", "`&`", "`or`", "`:=`", "`->`" "`<=`", "`=`", "`>=`", "`<`", "`<>`", "`==`", "`>`", "`!`".

The list of the identifiers (or symbol) currently having an infix status is given by the function:

♣ `caml_infixes : (unit -> string list)`

```
#caml_infixes();;
["Co"; "o"; "mod"; "quo"; "\\"; "~"; "'"] : string list
```

2.3 Booleans

The lexical class <*Bool*> has two elements: the two sequences of characters `true` and `false`, which are by the way keywords of the language.

2.4 Numbers

Beware:
-- Spaces *are not* allowed inside any numerical lexical units.

2.4.1 Integers

The lexical class <*Int*> is the set of decimal representation of "small" integers which must belong to the range -32767, +32767, as opposed to "big nums" which can be almost arbitrary large.

Integers start with a sharp character "#" optionally followed by "+" or "-", followed by a sequence of digits:

<*Int*> ::= "#" [<*sign*>] <*digits*>
<*sign*> ::= "-" | "+"
<*digits*> ::= <*d*>$^+$
<*d*> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

```
##-1;;
#-1 : int

#(* #32768 is out of range *)
##32768;;

line 2 Syntax error:
```

```
Skipping: 3 2 7 6 8 ;;
Parse Failed
```

2.4.2 Floating numbers

The lexical class <*Float*> is the set of decimal representation of "floating point" numbers.

They start with a sharp character "**#**" optionally followed by "**+**" or "**-**", followed by a sequence of digits, followed by a dot "**.**" and a non empty sequence of digits, optionally followed by an exponant.

An exponant is an "**e**" character optionally followed by "**+**" or "**-**" and followed by a non empty sequence of digits.

<*Float*> ::= "**#**" [<*sign*>] <*digits*> "**.**" <*digits*> [<*exponant*>]
<*exponant*> ::= "**e**" [<*sign*>] <*digits*>
<*sign*> ::= "**-**" | "**+**"
<*digits*> ::= <*d*>$^+$

For example:

```
##+1.1;;
#1.1 : float

##-1.2e-1;;
#-0.12 : float
```

2.4.3 Nums

The lexical class <*Num*>: these are the deciaml representation of numbers which are arbitrarily long sequences of digits (up to 256 characters).

<*Num*> ::= <*digits*> ["**.**" <*digits*> [<*exponant*>]]
<*exponant*> ::= "**e**" [<*sign*>] <*digits*>
<*sign*> ::= "**-**" | "**+**"
<*digits*> ::= <*d*>$^+$

```
#1;;
1 : num

#11111111111111111111111111111;;
11111111111111111111111111111 : num

#1.2e-3;;
0.0012 : num

#(* In the following + is parsed as the binary
#   operation over numbers, hence the syntax error *)
```

```
#+1.2e-3;;

line 3 Syntax error:
Skipping: + 0.0012 ;;
Parse Failed
```

Notice that the rational numbers do not belong to a special lexical class: they are just parsed as a division of two numbers and printed alike:

```
#1/2;;
1/2 : num
```

2.5 Strings

The lexical class <*String*>: strings are sequences of ASCII characters enclosed between double quote symbols "", for example `"Voici une chaine"`.

Inside strings, the character "\", named the *escape* character, has a special meaning: the sequence

- "\\" is interpreted as the back-slash character.
- "\"" is interpreted as the doublequote character.
- "\n" is interpreted as the line-feed character.
- "\r" is interpreted as an end of line.
- "\t" is interpreted as the tab character.
- "\f" is interpreted as the form-feed character.
- "\r" is interpreted as a carriage return.
- "\b" is interpreted as a back space.
- "\c" where c is a line-feed character is ignored: this is used to continue a string which may then spread onto several lines. In addition all spaces after the line-feed are ignored.
- "\c" where c is another character is finally interpreted as c.

Notice that the CAML system expands some of these special escapes when printing:

```
#"\"";;
"\"" : string

#"This chain : - has no line-feed in it \
#              - nor extra spaces!";;
```

```
"This chain : - has no line-feed in it - nor extra spaces!" :
 string

#"\a";;
"a" : string
```

2.6 Comments

2.6.1 Nested comments

Roughly speaking, comments are arbitrary sequences of characters enclosed by the two specialised parentheses (* and *).

As a matter of fact comments may be nested, and you may then put comments inside comments. This ensure the possibility to ignore as a comment an already commented CAML program.

Beware:

-- inside comments lexical analysis is performed: thus strings are properly skipped, even in:

```
#(****************** Obsolete Code
#
#(* This is a comment inside a comment *)
#let caml_ends_comments =
#    "a string may contain *), comments are continued";;
#
#(* although the preceding string contained "*)",
#   this string was correctly skipped *)
#
#(* (* The sequence of character *) is also skipped
#   inside concrete syntax quotations *)
#let caml_mult_operator = <:CAML:Expr<(prefix *)>>;;
#
#End of obsolete Code ****************************)
#
#1;;
1 : num
```

-- Nevertheless, there exists some very particular CAML programs which are not treated as a comment when enclosed between comments. For instance:

```
#let multiply = (prefix *);;
Value multiply = <fun> : (num * num -> num)
```

is a perfectly correct CAML program which belongs to this class, since *) would fool the lexical analyser which would consider it as an end of comments.

```
#(**** Obsolete
# let multiply = (prefix *);;****)
#1;;

line 2 Syntax error:
Skipping: ;; * * * * ) 1 ;;
Parse Failed
```

Another class of programs of this nasty family can be written using user's defined grammars which could enclose between their particular strings the fatal sequences of characters (* or *) or between their particular comments the sequence >>.

-- Another point to notice is that comments *must always end*: they are never implicitely closed, even at the end of a file:

```
#(*
#1;;

line 4 End of comment not found

Parse Failed
```

2.6.2 Simple comments

These are arbitrary sequence of characters starting and ending with the character %:

```
#% In CAML the string delimitor is " %
#1;;
1 : num
```

Inside this kind of comments no analysis is done at all, hence the " is skipped properly. The major drawback of these comments is the impossibility to add (or remove) comments around an entire program, without some rewriting of the program. Hence these comments should be limited to very particular uses, such as the example above ...

2.7 Lexical conventions in grammars

Grammars can inherit automatically any of the lexical classes of CAML. It suffices to cite the name of a class in the left-hand part of a grammar rule to get it available in the parser.

In addition, one may change the delimitors of the class <*String*>, and define particular comments which will be an alias of the CAML "simple" comments, by precising a single character to start and end comments.

Chapter 3

The Syntax of CAML

To express the syntax of CAML, we use a simplified notation, based on the BNF syntax, in fact the grammar interface with YACC has been used to produce the parser of the language.

3.1 Syntax definition conventions

The metalanguage used to specify the syntax is based on the BNF. The meta-symbols have the following meaning:

::=	:	is defined as,
|	:	or,
[x]	:	0 or 1 occurrence of x
(x) *	:	0 or more occurrences of x,
(x) +	:	1 or many occurrences of x,
(x | y)	:	x or y,
<*Name*>	:	the non-terminal symbol Name
"xyz"	:	the keyword xyz

The syntaxes may use any of the CAML lexical units (`Ident`, `String`, `Bool`, `Infix`, `Num`, `Int` and `Float`).

<*Caml expression0*> designates a CAML expression (entry point: Expr0 in the CAML grammar) which is a basic lexical unit or a parenthesised expression.

3.2 Toplevel grammar

CAML possesses a toplevel grammar with its specific keywords. This grammar has to be seen as implementing a parser whose role consists in chosing what parser has to be called: directives or pragmas, autoload declarations, expressions or declarations, grammar definitions, printer definitions.

<*Toplevel phrase*>

::= "#" <*Directive or Pragma*>
| "module" <*Module import statement*>
| "end" "module" <*Module export statement*>
| "autoload" <*Autoload declaration*>
| "grammar" <*Grammar declaration*>
| "printer" <*Printer declaration*>
| <*Expression or Declaration*>

This toplevel syntax does not recognize macros since the "#" sign calls the parser for directives and pragmas. Thus, one has to enclose toplevel macro calls between parentheses. For example:

```
##pragma let X=<:Caml:Expr<1+2>>;;
Pragma Value X = <:Caml:Expr<1+2>> : ML

##X+1;;

line 1 Syntax error:
Skipping: X + 1 ;;
Parse Failed

#(#X)+1;;
4 : num
```

3.3 Constant Expressions

Constant expressions are built by using directly lexical items. See section 2 for more information about these items.

<*Constant*> ::= <*NUM*>
| <*INT*>
| <*FLOAT*>
| <*STRING*>
| <*BOOL*>
| "(" ")"
| "{" "}"

3.4 Expressions

Expressions belong to the syntactical category named *Expr* which is described by using categories $Expr_i$ for $i = 0, \ldots, 12$ representing expressions with tighter precedence. The category *Expr0* is important since its parser `parse_caml_expr0` is useful for defining escapes from a grammar to CAML. Auxiliary categories *Olprog*, *Excn*, *Ifexpr*, *Fnexpr*, *Match*, *Trycase*, *Umatch* and *Trymatch* are defined later in the section and we will devote special sections to *Pat*, *Decl*, *Type*, *Constructor*, *ValBdg* ...

```
<Expr>     ::= "protect" <Expr>
             | <Expr12> "?" <Expr>
             | <Expr12> "where" <ValBdg>
             | <Expr12> "where" "rec" <ValBdg>
             | "vector" <Expr> "of" <Expr3>
             | "vector" "of" <Expr3>
             | "segment" <Expr> "of" <Expr3>
             | "segment" "of" <Expr3>
             | <Fnexpr> (* Function bodies, case expressions, exception *)
                          handlers and local declarations (let ...in
                          ...construct)
             | <Expr12>
<Expr12>   ::= <Expr11> (";" <Fnexpr> |<Expr11>)*
<Expr11>   ::= <Ifexpr> (* Conditional expressions *)
             | <Expr10>
<Expr10>   ::= "raise" <Excn>
             | "failwith" <Expr10>
             | <Expr9>
<Expr9>    ::= <Expr8> ":=" (<Fnexpr> |<Expr11>)
             | "at" <IDENT> "<-" (<Fnexpr> |<Expr11>)
             | <Expr8>
<Expr8>    ::= <Expr7> "," (<Fnexpr> |<Ifexpr> |<Expr8>)
             | <Expr1> "." <IDENT> "<-" (<Fnexpr> |<Ifexpr> |<Expr8>)
             | <Expr1> "." <NUM> "<-" (<Fnexpr> |<Ifexpr> |<Expr8>)
             | <Expr1> "." "(" <Expr> ")" "<-" (<Fnexpr> |<Ifexpr> |<Expr8>)
             | <Expr7>
```

```
<Expr7>    ::= <Expr7> "or" <Expr7>
             | <Expr7> "&" <Expr7>
             | <Expr6>
<Expr6>    ::= "not" <Expr5>
             | <Expr5>
<Expr5>    ::= <Expr4> "=" <Expr4>
             | <Expr4> "<" <Expr4>
             | <Expr4> ">" <Expr4>
             | <Expr4> "<=" <Expr4>
             | <Expr4> ">=" <Expr4>
             | <Expr4> "<>" <Expr4>
             | <Expr4> "==" <Expr4>
             | <Expr4>
```

```
<Expr4>  ::=  <Expr4> <INFIX> <Expr4>
           |  <Expr4> "@" <Expr4>
           |  <Expr4> "^" <Expr4>
           |  <Expr4> "::" <Expr4>
           |  <Expr4> "+" <Expr4>
           |  <Expr4> "-" <Expr4>
           |  <Expr4> "*" <Expr4>
           |  <Expr4> "/" <Expr4>
           |  "-" <Expr4>
           |  <Expr3>
<Expr3>  ::=  <Expr2> (<Expr2>)*
           |  <Constructor> (* cf. section 3.5 *)
           |  <Constructor> <Constructor>
           |  <Constructor> <Expr0>
<Expr2>  ::=  "!" <Expr2>
           |  <Expr1>
<Expr1>  ::=  <Expr1> "." <IDENT>
           |  <Expr1> "." <NUM>
           |  <Expr1> "." "(" <Expr> ")"
           |  <Expr0>
```

```
<Expr0>  ::=  <Constant> (* cf. section 3.3 *)
           |  <IDENT>
           |  "prefix" <Infixes>
           |  <Olprog>
           |  "#" <Expr0>
           |  "fail"
           |  "[" "]"
           |  "[" (<Expr11> (";" <Expr11>)* ) "]"
           |  "[|" "|]"
           |  "[|" (<Expr11> (";" <Expr11>)* ) "|]"
           |  "[<" ">]"
           |  "[<" (<Expr11> (";" <Expr11>)* ) ">]"
           |  "(" <Expr> ")"
           |  "(" <Expr> ":" <Type> ")"
           |  "(" <Expr> "#:" <Expr> ")"
           |  "while" <Expr> "do" <Expr> "done"
           |  "begin" "do" <Expr12> "end" "do"
           |  "begin" "while" <Expr> "do" <Expr> "end" "while"
           |  "begin" "fun" <Match> "end" "fun"
           |  "begin" "function" <Umatch> "end" "function"
           |  "begin" "match" <Expr> "with" <Umatch> "end" "match"
           |  "begin" "case" <Expr> "of" <Umatch> "end" "case"
           |  "begin" "try" <Expr> "with" <Trymatch> "end" "try"
           |  "begin" "if" <Expr> "then" <Expr> "else" <Expr> "end" "if"
           |  "begin" "if" <Expr> "then" <Expr> "end" "if"
           |  "{" <IDENT> "=" <Expr11> (";" <IDENT> "=" <Expr11>)* "}"
```

Object languages expressions:

```
<Olprog>  ::=  "<<" <Expression from default grammar> ">>"
            |  "<:" <IDENT> "<";
               <Expression from specified grammar at default entry point> ">>"
            |  "<:" <IDENT> ":" <IDENT> "<";
               <Expression from specified grammar at specified entry point> ">>"
```

Exceptions:

```
<Excn>  ::=  "(" <Excn> ")"
          |  <IDENT> (<Fnexpr> |<Expr10>)
          |  <IDENT>
```

Conditional expressions:

```
<Ifexpr>  ::=  ("if" <Expr> "then" (<Fnexpr> |<Ifexpr> |<Expr10>))+
               ["else" (<Fnexpr> |<Ifexpr> |<Expr10>)]
```

Function bodies, case expressions, exception handlers and local declarations:

```
<Fnexpr>  ::=  "fun" <Match>
            |  "function" <Umatch>
            |  "match" <Expr> "with" <Umatch>
            |  "case" <Expr> "of" <Umatch>
            |  "try" <Expr> "with" <Trymatch>
            |  <Decl> (* cf. section 3.7 *) "in" <Expr> *)
```

Matches (with only one pattern):

```
<Umatch>  ::=  <Pat> "->" <Expr>
            |  <Umatch> "|" <Pat> "->" <Expr>
            |  <Umatch> "#|" <Expr>
```

Matches with possibly several patterns:

```
<Match>  ::=  (<Pat0> (* cf. section 3.6 *))+ "->" <Expr>
            |  <Match> "|" (<Pat0>)+ "->" <Expr>
            |  <Match> "#|" <Expr>
```

Matches for exceptions:

```
<Trymatch>  ::=  <Trycase>
              |  <Trymatch> "|" <Trycase>
              |  <Trymatch> "#|" <Expr>
```

Match case for exceptions:

```
<Trycase>  ::=  <Pat> "->" "reraise"
             |  <Pat> "->" <Expr> "reraise"
             |  <Pat> "->" <Expr>
```

3.5 Identifiers and constructors

```
<MLIdent2>  ::=  "prefix"; <MLInfix>
              |  <MLIdent0>

<MLIdent0>  ::=  <IDENT>
              |  "["; "]"

<MLIdent>  ::=  <MLIdent0>
             |  <MLInfix>

<MLInfix>  ::=  <Infixes>
             |  <Prefixes>

<Infixes>  ::=  <Typeinfixes>
             |  "*"
             |  "=="
             |  "="

<Typeinfixes>  ::=  <Inf>
                 |  "-"

<Prefixes>  ::=  "not"
              |  "!"
```

```
<Inf>  ::=  <INFIX>
         |  "+"
         |  "/"
         |  "<="
         |  ">="
         |  "<>"
         |  "<"
         |  ">"
         |  "::"
         |  "@"
         |  "&"
         |  "or"
         |  "^"
         |  ":="

<Constructor>  ::=  "'"; <MLIdent2>
```

3.6 Patterns

```
<Pat>  ::=  <Pat5> "as" <MLIdent2>
         |  <Pat5>

<Pat5>  ::=  <Pat4> ("|" <Pat4>)+
           |  <Pat4>

<Pat4>  ::=  <Pat4> "," <Pat4>
           |  <Pat4> "::" <Pat4>
           |  <Pat4> <INFIX> <Pat4>
           |  <Pat1>

<Pat1>  ::=  (<Constructor> |<MLIdent2>) <Pat0>
           |  "dynamic" <Pat0>
           |  <Pat0>

<Pat0>  ::=  <Constant>
           |  "_"
           |  "-" <NUM>
           |  <Constructor>
           |  <MLIdent2>
           |  "#" <Expr0>
           |  "[" <Pat> (";" <Pat>)* "]"
           |  "{" <Labpat> (";" <Labpat> )* "}"
           |  "(" <Pat> ")"
           |  "(" <Pat> ":" <Type> ")"
           |  <Pat0> "at" <MLIdent>
           |  <Olprogram>
```

```
<Labpat>  ::=  <IDENT> "=" <Pat>
            |  <IDENT>
            |  "_"
```

3.7 Declarations

```
<Decl>  ::=  ("let" |"value") ["rec"] <ValBdg> ("and" <Valbdg>)*
          |  "type" <TyBdg> ("and" <Tybdg>)*
          |  "exception" <ExcBdg> ("and" <ExcBdg>)*
          |  "(" <Decl> ")"
```

Value bindings:

```
<ValBdg>  ::=  <Pat0> ("," <Pat0>)* "=" <Expr>
            |  <Pat0> (<Pat0>)+ "=" <Expr>
            |  <Pat0> <Inf> <Pat0> "=" <Expr>
            |  <ValBdg> "|" <Pat0> (<Pat0>)+ "=" <Expr>
```

Type bindings:

```
<TyBdg>  ::=  <Ty> "=" <Constructors>
            |  <Ty> "=" <Labels>
            |  <Ty> "==" <Type>
<Ty>  ::=  [<Tyvars>] <MLIdent2>
         |  <Varty> <Infixes> <Varty>
<Tyvars>  ::=  <Varty>
             |  "(" <Varty> ("," <Varty>)* ")"
<Constructors>  ::=  <Constrs>
                  |  "[" <Constrs> "]"
<Constrs>  ::=  <Constr1>
             |  <Constrs> "|" <Constr1>
             |  <Constrs> "#|" <Expr0>
<Constr1>  ::=  (<Constructor> |<MLIdent2>) "of" <Type>
             |  (<Constructor> |<MLIdent2>)
             |  ("mutable" |"!") (<Constructor> |<MLIdent2>) "of" <Type>
             |  ("lazy" |"*") (<Constructor> |<MLIdent2>) "of" <Type>
<Labels>  ::=  <Labs>
            |  "{" <Labs> "}"
<Labs>  ::=  <Lab1>
           |  <Labs> "|" <Lab1>
           |  <Labs> "#;" <Expr0>
<Lab1>  ::=  <MLIdent> ":" <Type>
           |  ("mutable" |"!") <MLIdent> ":" <Type>
           |  ("lazy" |"*") <MLIdent> ":" <Type>
```

Exception bindings:

```
<ExcBdg>  ::=  <MLIdent> "of" <Type>
            |  <MLIdent>
```

3.8 Type expressions

```
<Type>     ::= <Type> "->" <Type>
             | <Type2> "*" <Type>
             | <Type2>

<Type2>    ::= <Type1> <Type_infixes> <Type2>
             | <Type1>

<Type1>    ::= <Var_ty>
             | <MLIdent2>
             | "(" <Type> ")"
             | <Type1> <MLIdent2>
             | "(" <Typel> ")" <MLIdent2>

<Typel>    ::= <Type> ("," <Type>)+

<Var_ty>   ::= "'" <MLIdent>
```

3.9 Syntax of Pragmas and Directives

Here is the syntax of pragmas and directives. Pragmas and directives may take two forms: one is the general form allowing to use any CAML expression and declaration only at compile time (pragmas) or both at compile time and load time (directives). The second form consists in abbreviations for the first form. It permits to have a nicer syntax (one hopes so!) for some basic form.

All (toplevel) syntactic constructs described in this section must be preceded by a "#" recognized by the toplevel CAML parser; they must also be followed by ";;".

3.9.1 Basic forms

```
<Basic form of Pragmas>
  ::= "pragma" <Declaration> ";;"
   | "pragma" <Expression> ";;"
<Basic form of Directives>
  ::= "directive" <Declaration>
   | "directive" <Expression>
```

3.9.2 Other Pragmas and Directives

We give here the syntax of degenerated forms of pragmas and directives. For each of them, we indicate between comment delimitors (at the right hand side of the page) whether the current construct is a pragma or a directive.

<Degenerated forms of Pragmas or Directives>

```
::=  "use" <CAML Expression>                                          (* directive *)
  |  "load" <CAML Expression>                                         (* directive *)
  |  "compile" <CAML Expression>                                      (* directive *)
  |  "infix" <IDENT>                                                  (* directive *)
  |  "uninfix" <IDENT>                                                (* directive *)
  |  "fast" "arith" <CAML Expression>                                 (* directive *)
  |  "open" "compilation" <CAML Expression>                           (* directive *)
  |  "open" "optimization" <CAML Expression>                          (* directive *)
  |  "open" "printing" <CAML Expression>                              (* directive *)
  |  "open" "overloading" <CAML Expression>                           (* directive *)
  |  "open" "module" <CAML Expression>                                (* directive *)
  |  "set" "default" "grammar"
     (<STRING> |<IDENT>) [":" <STRING> |<IDENT> ]                    (* pragma *)
  |  "default" "grammar"                                              (* directive *)
  |  "printer" [<STRING>] (<STRING> | <IDENT>) [<STRING>]             (* Caml Expression
  |  "default" "printer" "for" "type" <Caml Type>                     (* Caml Expression
  |  "eval" "when" "print" <Caml Expression>                          (* directive *)
  |  "quit"                                                           (* pragma *)
```

The syntactic constructs which are indicated as "(* Caml Expression *)" are treated as CAML toplevel expressions (i.e. unevaluated at compile time, evaluated at load time).

Chapter 4

Basic values and basic data types

Convention:
In this manual, the symbol ♣ in the left margin of the page, will indicate a function (or set of functions) provided by the CAML system. These functions are listed in the index, and separately with their type in the glossary.

4.1 Types and values

Since CAML is strongly typed (i.e. every legal phrase has a type), the description of the values of the language is very close to the one of the types.
In CAML values are built using very few concepts:

- **Built-in** values, such as strings or integers.
- **Definable** data types : these are means to define new values which are elements of new types, introduced by a *type declaration.* Definable data types fall into two distinct categories:
    1. **Sums**, from the mathematical notion of "disjoint union": the disjoint union of two sets A and B is the set C (classically named A + B) of all the elements belonging either to A or to B, marked with a tag indicating whether they come from A or B.
    2. **Products**, from the mathematical notion of "cartesian product": the cartesian product of two sets A and B is the set C (classically named A × B) of all the pairs whose first component is an element of A and second component an element of B.

As for values, types may be either

1. **Built-in** types (such as `string` or `int`).

2. **Defined** types (such as `list`).

In short:

- Values are built using predefined constants or **constructors**
- Types are similarly built using predefined *type* constants or *type* constructors.
- New type constants and constructors are declared and these declarations introduce new *value* constructors associated with the new type.

4.1.1 Value Constructors

Sum type declarations introduce new values, bound to identifiers of a special kind: the *constructors* (the word "constructor" being intended to be a shorthand for "constructor of values").

These constructors may be **constants** or **injections** from a given type to the type being defined, and in the latter case they possess a functional type (one calls them "functional" constructors).

Let us give some examples of types predefined in CAML:

- The classical type of boolean values contains only two constants `true` and `false`: thus it may be defined as

```
type bool = true | false;;
```

  This phrase declares the two new constant values `true` and `false`, which are the two constant constructors of the new defined type `bool`. The meaning of the above type declaration is exactly that every value belonging to the type `bool` must be either `true` or `false`. Thus the symbol "`|`" may be read as `or`.

  In the mathematical sense, we have `bool = {true} + {false}`: the set `bool` is the disjoint union of the two singleton sets `{true}` and `{false}`.

- An enumerated set may be defined with a type declaration, listing all the elements of the set, and thus introducing these elements in the language as new constant constructors.

- The predefined type `unit` which possesses only one element denoted by "`()`" could be defined as:

```
type unit = ();;
```

  which defines its single element as a new constant value of the language.

- Integer values may be seen as an infinite enumerated set: it is the infinite sum of the singleton sets `{0}`, `{1}`, `{2}`, `{3}`, ..., as suggested by

```
type int = 0 | 1 | 2 | 3 | .... | -1 | -2 | ....;;
```

This declares an infinite number of constant constructors, i.e. exactly all integer values (but, since it is impossible to declare infinitely many constants in practice, the type `int` must be predefined).

- In the same vein, string values may also be seen as an infinite sum of singleton sets (all the sets `{s}` where `s` ranges on all the legal strings of the language (every finite sequence of characters enclosed by doublequotes `""`)).

- Let us write a "real" type definition:

```
type variable = Variable of string;;
```

In this case we define a new type "`variable`", together with a new functional constructor "`Variable`", which injects string values into the new type `variable`: `Variable "x"` belongs to the type `variable`.

This is used to distinguish between data types which are semantically identical (i.e. contain the same set of values, as the type `variable` and the predefined type `string`) but must not be identified for the sake of clarity (and safety) of the programs manipulating values of such isomorphic types.

- Basic cartesian product is defined in CAML: elements of type `t1 * t2` are pairs of values of type `t1` and `t2`:

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

4.1.2 Type constructors

Types are built using constructors as well, which may be either constant (as predefined types `string` or `num`) or functional. For example `list` is such a functional type constructor, since from type argument `string` it builds the type `string list` and from type argument `bool` the type `bool list`. Do not be confused by the fact that the constructor `list` is postfixed. It constructs types in the same way as the constructor `Variable` from our previous type definition builds values: `bool list` is strictly analogous to `Variable "foo"`, except that a type is built instead of a value.

The predefined type constructor "`*`" has two type variables as arguments, since it builds (the type of) the cartesian product of two types.

Type constructors may have as many arguments as desired.

4.2 Sum Types

4.2.1 Simple examples

First we define the type of string lists: a list may be either empty (then we denote it with the constant constructor `Empty`) or a pair of its first element in front of another list (then we denote it with the functional constructor `Add`):

```
#type string_list = Empty | Add of (string * string_list);;
Type string_list defined
     Empty : string_list
   | Add : (string * string_list -> string_list)

#let l = Empty;;
Value l = Empty : string_list

#Add ("toto",Add ("tata",l));;
(Add ("toto",(Add ("tata",Empty)))) : string_list
```

The drawback of the type `string_list` is that we cannot reuse it for another kind of list: we ought to define another type to get list of integers, or list of whatever. In fact, it is possible to define a *polymorphic* type list which will provide us with completely general list constructors.

When a functional *type* constructor is defined it takes as argument a *type* variable. Such a type variable is denoted by an identifier (element of the lexical class <*Ident*>) preceded by a "'" symbol to distinguish it from ordinary constant type names. For instance the empty list will have type `t list` for every type `t`, which will be written `'t list`.

```
#type 't List = Empty | Add of 't * 't List;;
Type List defined
     Empty : 'a List
   | Add : ('a * 'a List -> 'a List)
```

Now we have:

```
#Empty;;
Empty : 'a List

#Add (1,Empty);;
(Add (1,Empty)) : num List

#Add ("oui",Empty);;
(Add ("oui",Empty)) : string List
```

4.2.2 Pattern matching

Constructors have the important property that they allow *pattern matching* on values of their types. Pattern matching is a kind of switch among different cases according to an argument value. For instance using the (implicit) constructor 1, you may write the following function which will decide whether its argument is in fact the constructor 1:

```
#let is_one = function 1 -> true | x -> false;;
Value is_one = <fun> : (num -> bool)
```

Cases of matches are clauses <*pattern*> "->" <*expression*>.

- The value of <*expression*> is returned if the corresponding clause is selected.
- If <*pattern*> is reduced to a variable (such as x in the preceding example) then every value of the type matches the corresponding pattern.
- Patterns of clauses are tried in the order of presentation. Thus permutation of clauses generally affects the semantics: `is_one` is equivalent to

  ```
  #let is_one x = if x = 1 then true else false;;
  Value is_one = <fun> : (num -> bool)
  ```

  but if we change the order of the clauses:

  ```
  #let is_one = function x -> false | 1 -> true;;
  Warning: 1 unused match case in this phrase
  Value is_one = <fun> : (num -> bool)
  ```

  the function will always return `false` since every value of type `num` matches the first pattern x. Thus the second clause will never be tried, hence the system warning.
- Moreover pattern matching is *specific* to constructors and variables. You *cannot match with computed values as patterns.*

Consider:

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

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

This function will allways return **true**, since **int_one** is not a declared constructor (either explicitly in a type definition, or implicitly as one of the basic constant values of the language). It is then understood by the system as a variable, distinct from the global one with the same name which is bound to the value 1, and again all values of type **num** match the first clause.

Beware:

-- To prevent such mistakes it is a good habit to use capitalized identifiers for constructors and labels, and lower case identifiers for variables.

In addition to constructors and variables, the special symbol "_" (**wildcard**) may appear in patterns: it stands for "whatever". The function **is_one** may then be rewritten with such a wildcard:

```
#let is_one = function 1 -> true | _ -> false;;
Value is_one = <fun> : (num -> bool)
```

The last purpose of patterns is to effect bindings of values or part of values, needed in the right hand-side of the clause. For instance to compute the sum of all the elements of a list:

```
#let rec sum = function
#  Empty -> 0
#| Add (x,l) -> x + (sum l);;
Value sum = <fun> : (num List -> num)
```

There we have bound simultaneously the head and the tail of the list argument to x and l in the left hand-side of the clause. Notice that these bindings are done recursively with the proper values:

```
#sum (Add (1,Add (2, Add (3,Empty))));;
6 : num
```

This binding effect of pattern matching together with proper use of "wildcard" clauses is very powerful and elegant.

```
#let is_not_empty = function
#    Add (_,_) -> true
#  | _ -> false;;
Value is_not_empty = <fun> : ('a List -> bool)
```

Moreover patterns may overlap (a pattern may be an *instance* of another one) and in such a case ambiguities are solved using order of presentation: if a given value matches 2 clauses the first one is chosen. For instance:

```
#let rec sum = function
#  Empty -> 0
#| Add (x,Empty) -> x
#| Add (0,l) -> sum l
```

```
#| Add (x,(Add (0,1))) -> sum (Add (x,1))
#| Add (x,1) -> x + (sum 1);;
Value sum = <fun> : (num List -> num)
```

the value `Add(0,Add (0,Empty))` matches the 3 last clauses.

These implicit rules can simplify the number of necessary clauses. Consider the following function which appends lists. We can write it as follows:

```
#let rec append_lists = function
#  Empty,Empty -> Empty
#| Empty,L -> L
#| L,Empty -> L
#| Add (x,1),L1 -> Add (x, append_lists (1,L1));;
Value append_lists = <fun> : ('a List * 'a List -> 'a List)
```

We can discard the first clause and still get the same function:

```
#let rec append_lists = function
#  Empty,L -> L
#| L,Empty -> L
#| Add (x,1),L1 -> Add (x, append_lists (1,L1));;
Value append_lists = <fun> : ('a List * 'a List -> 'a List)
```

the value `Empty,Empty` matches the 2 first clauses, and the first one will be applied. In our example this is not very important, but you must be careful about the semantics of pattern matching, when clauses overlap.

Partial matches

So far the set of clauses we used were always *exhaustive*: every value of the source type matched at least one clause.

In fact pattern matching can be not completely exhaustive as in:

```
#let is_agreement = function "yes" -> true | "no" -> false;;
Warning: 1 partial match in this phrase
Value is_agreement = <fun> : (string -> bool)
```

Since there are more than 2 constant constructors in the type `string` the match may fail:

```
#is_agreement "ok";;

Pattern matching Failed
```

When encountering this kind of pattern matching, the compiler adds a safeguard to prevent runtime failures, namely an extra clause

```
| _ -> raise match_failure
```

This exception is then trapped by the toplevel of the language which reports the above message (about exceptions see the section 4.5).

Beware:

-- Remember that it is a nasty programming habit to allow partial matches into your programs. Do your best to avoid them since they often lead to unexpected errors.

Complex patterns

A pattern may be as complex as desired including as many constant constructors, wildcards and variables as necessary. Bindings may occur in any sub-pattern of a pattern using the syntactic construction **as**, which names a part of a pattern for further use in the right-hand part of the clause.

```
#let rec remove_but_one = function
#  Add(x,(Add (y,1) as L)) -> remove_but_one L
#| L -> L;;
Value remove_but_one = <fun> : ('a List -> 'a List)
```

Beware:

-- All patterns in CAML must be **linear**, that is to say *no variable can appear more than once in a pattern*:

```
#let egal = function (x,x) -> true | _ -> false;;

line 1: multiple definition of variable x in x,x
1 error in typechecking

Typecheck Failed
```

4.2.3 Disjoint union

The general type of disjoint union of two types can be defined in CAML as a polymorphic type definition:

```
type 't1 + 't2  = Inl of 't1 | Inr of 't2
;;
```

You may notice that +, as a type constructor, gets two parameters (or more exactly a pair of parameters).

With Inl you inject a value into the left summand of the type and similarly for Inr into the right summand.

```
#let x = Inl 1;;
Value x = (Inl 1) : (num + 'a)
```

There is no typechecking constraint on the right summand, thus it remains polymorphic. If it is not desired, it is possible to add an explicit type constraint to get an instance of the above type for x:

```
#let (x':num + bool) = Inl 1;;
Value x' = (Inl 1) : (num + bool)

#let (y:num + bool) = Inr false;;
Value y = (Inr false) : (num + bool)

#let boolean_value_of = function
#    Inl 1 -> true | Inl _ -> false | Inr b -> b;;
Value boolean_value_of = <fun> : ((num + bool) -> bool)

#boolean_value_of x';;
true : bool

#boolean_value_of y;;
false : bool

#boolean_value_of x;;
true : bool
```

4.3 Products

Product type declarations are similar to sum type declarations, but one defines the format of a record of values with labelled fields. Instead of constructors one declares new labels[1] to name the fields of the record:

```
#type car = {Weight:num; Speed:num; Brand:string};;
Type car defined
     .Weight : (car -> num)
   ; .Speed : (car -> num)
   ; .Brand : (car -> string)
```

This defines a new type of records with 3 fields, which must contain respectively a num for the labels Weight and Speed, and a string for the label Brand. Then we can build values of type car by just filling in the fields of the record with any value compatible with the declared type of the field:

```
#let R25 = {Weight=1200; Speed=220; Brand="Renault"};;
Value R25 = {Weight=1200; Speed=220; Brand="Renault"} : car

#let BX16 = {Weight=1050; Speed=200; Brand="Citroen"};;
```

[1] which have not to be different from all other record's labels.

```
Value BX16 = {Weight=1050; Speed=200; Brand="Citroen"} :
      car
```

As for sum types, accessing is done via pattern matching:

```
#let weight_of_car = function
#    {Weight = w; Speed = _; Brand = _} -> w;;
Value weight_of_car = <fun> : (car -> num)

#weight_of_car R25;;
1200 : num
```

The semantics of variables and wildcards in pattern matching is identical to those for sum types, but for products there is in addition a special ellipsis for non "interesting" fields: an underscore just following a field delimitor ";", that is "; _" means "whatever may be the other fields and their contents" (more precisely this _ stands for *0 or more* fields). The preceding example may be rewritten as:

```
#let weight_of_car = function {Weight=w; _} -> w;;
Value weight_of_car = <fun> : (car -> num)

#weight_of_car R25;;
1200 : num
```

Another way to access the content of a field is provided with the "." notation: `r.field_name` returns the content of the field `field_name` into the record `r`.

```
#R25.Speed > BX16.Speed;;
true : bool
```

Finally notice that pattern matching with sum and product types may be mixed arbitrarily.

```
#type loading = {Weight : num; Goods : string};;
Type loading defined
     .Weight : (loading -> num)
   ; .Goods : (loading -> string)

#type vehicle = Car of car | Lorry of car*loading;;
Type vehicle defined
     Car : (car -> vehicle)
   | Lorry : (car * loading -> vehicle)

#let weight_of_vehicle = function
#  Car {Weight=w;_} -> w
#| Lorry ({Weight=w;_},{Weight=w';_}) -> w+w';;
Value weight_of_vehicle = <fun> : (vehicle -> num)
```

Cartesian product

This is the common binary product type: the cartesian product t1 × t2 of two types t1 and t2. That is, the type of all the pairs which may be built with elements of t1 and t2. It is a polymorphic product type definition with two type variables:

```
#type ('t1,'t2) pair = {Fst:'t1; Snd:'t2};;
Type pair defined
     .Fst : (('a,'b) pair -> 'a)
   ; .Snd : (('b,'a) pair -> 'a)

#{Fst = 1; Snd = "oui"};;
{Fst=1; Snd="oui"} : (num,string) pair
```

4.4 Type abbreviations

These are mainly used to make type information easier to read and enter. They are introduced with an "==" symbol between a type and its abbreviated version:

```
#type single == unit;;
Type single abbreviates unit
```

Then you get:

```
#();;
() : single
```

An abbreviation may be "active" or "inactive":

♣

```
echo_abbrev : (string -> bool -> single)
echo_abbrevs : (bool -> single)
```

With `echo_abbrev "single" false` you disable the reduction of the type abbreviation named `single`, and you revert to normal reduction mode with `echo_abbrev "single" true`.

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

With `echo_abbrevs` you can switch globally (i.e. for all abbreviations) the reduction process.

4.5 Exception handling

Apart from values and types, the basic notions of *exceptions* and exceptional values are used to control the way computation runs. One can think of exceptional values as signals which are sent, and which may be received by functions. Sending is performed by the "`raise`" construct and receiving by the "`try`" ... "`with`" construct.

Exceptions are typechecked, and thus have to be declared:

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

Now the exception `Failure` can be raised:

```
#raise (Failure "uncaught");;
Global exception Failure was raised with value
  "uncaught" : string

User's exception never trapped
```

and trapped:

```
#try raise (Failure "uncaught")
#with Failure "uncaught" -> 1;;
1 : num
```

The "`with`" part of a "`try`" ... "`with`" is just an ordinary match on exceptional values.

Now two functions may interact through exception handling:

```
#let divide_primitive (x,y) =
#    if y = 0 then raise (Failure "overflow")
#    else x/y;;
Value divide_primitive = <fun> : (num * num -> num)

#let divide_without_overflow (x,y) =
#    try divide_primitive (x,y) with
#     (Failure "overflow") -> 0
#;;
Value divide_without_overflow = <fun> : (num * num -> num)

#divide_without_overflow (1,0);;
0 : num
```

This exception mechanism[2] is as fast ordinary procedure call, it even can be faster. When returning from a recursive function with an exceptional value, all the recursive calls are discarded at once.

[2]This mechanism is (almost completely) independant from the underlying operating system.

4.6 Mutable and lazy values

When declaring a type, some constructors or labels can be qualified as "`mutable`" or "`lazy`":

- "`mutable`" means that the argument of the constructor (or the content of the field of the label) may be dynamically (and *physically*) updated: the corresponding value is still a constant (i.e. bound to the same address in the computer memory) but its "value" may change (i.e. the content of the address may be updated).

- "`lazy`" means that the argument of the constructor (or the content of the field of the label) will not be computed *at once*, but if and only if this argument is absolutely necessary for further computation. In this case it will be effectively computed, but *only once*, and the address bound to the argument will be physically updated with the computed value.

Mutable values are used mainly to maintain a global state which changes dynamically when the program runs. Global variables bound to mutable values allow as well information passing between as many function as desired: every function may read (and write) the global variables to get (and give) information about the current state of the computation. This may be simulated with extra argument parameters and results, but at the price of useless effort to maintain and read the programs.

Lazy values mainly enable potentially "infinite" data structures. Values of that kind cannot be completely evaluated and hence they are progressively computed ("expanded" in a "lazy" way) as long as the computation needs it.

Another advantage of lazy values is the automatic avoidance of useless computations (not needed for the final result). This implies that so-called "side-effects" must be avoided in lazy values, because they may never be evaluated at all or at least in a surprising order.

4.7 Mutable values in sums

4.7.1 References

We first define the basic type of mutable values:

```
type 't ref = mutable ref of 't
;;
```

Now we may define a counter using a reference:

```
#let counter = ref 1;;
Value counter = (ref 1) : num ref
```

4.7.2 Occurrences

We present the basic mechanism to update mutable values. Other ways to update values will be presented afterwards.

To update mutable values one has to name the precise place where the value has to be "changed". This is done by special annotations of patterns using the keyword "at", followed by an identifier naming the occurrence. For instance

```
match ... with ref (y at occ) -> ...
```

designates y as being at occurrence named occ, and then it is possible to update y in the right-hand side of the match.

The physical replacement of a value at a designated occurrence occ has the following syntax:

```
at occ <- new_value
```

and the semantics is that new_value is inserted at the place designated by the occurrence occ.

The construct `at occ <- new_value`, may be read "at occ put new_value", or "occ receives new_value".

The value of the whole construct is the updated value (i.e. `new_value`).

```
#match counter with ref (_ at occ) ->
#at occ <- 2;;
2 : num

#counter;;
(ref 2) : num ref
```

4.7.3 Primitives for references

References are a very commmon instance of mutable values. Thus, we provide specialised functions to update references and to find their contents:

♣
```
prefix ! : ('a ref -> 'a)
prefix := : ('a ref * 'a -> 'a)
```

"prefix" allows the distinction between an "infix" identifier and its "prefix" form. This is explained in section 5.1.

- `prefix !` (read as "prefix" "deref") returns the content of a reference. It is defined as:

  ```
  let prefix ! (ref x) = x;;
  ```

- `prefix :=` is used to assign a value to a reference. It may be defined as

```
let x_ref := x =
    match x_ref with ref _ at occ -> at occ <- x;;
```

Or more concisely by

```
let (ref _ at occ) := x =
 at occ <- x;;
```

4.7.4 Using references

Symbol generation

We define a generator of symbols in the spirit of the well-known Lisp "gensym"[3]. We define simultaneously the symbol generator and the function which resets it: `reset_gen_sym`.

We use a local reference counter which is shared between `gen_sym` and `reset_gen_sym`, incremented each time `gen_sym` is called and reset to 0 when necessary by `reset_gen_sym`.

```
#let gen_sym,reset_gen_sym =
# let sym_counter = ref 0 in
# (function () -> "symbol"^
#                 (string_of_num (sym_counter:=!sym_counter+1))),
# (function () -> sym_counter:=0);;
Value gen_sym = <fun> : (unit -> string)
Value reset_gen_sym = <fun> : (unit -> num)

#gen_sym();;
"symbol1" : string

#gen_sym();;
"symbol2" : string

#reset_gen_sym();;
0 : num

#gen_sym();;
"symbol1" : string
```

For loop

Combined with exception handling, references allow us to define a "for" loop. We define first a specific exception to be raised when a next step is required:

```
#exception next of num ref;;
Exception next of num ref defined
```

[3] Except that in Lisp, gensym returns new, unbound *identifiers*, not strings.

Then the iterator itself:

```
#let for (index,start,stop,step) =
# index:=start;
# let rec loop body =
#  if !index > stop then ()
#  else
#  (try body index
#   with next j ->
#    (* Test physical identity of the two references index and j *)
#    if j==index
#    (* This is our index: increment it and continue *)
#     then (index:=!index+step;loop body)
#    (* This is not our index: exception is propagated *)
#     else raise next j)
# in loop;;
Value for = <fun> :
      (num ref * num * num * num -> (num ref -> unit) ->
       unit)
```

Then we can use two interleaved loops, with two indexes `i` `j`:

```
#for (ref 0,1,2,1)
# (fun i ->
#   (for (ref 0,1,4,2)
#     (fun j ->
#       print_string"i = ";print_num !i;print_space();
#       print_string"j = ";print_num !j;print_newline();
#       raise next j));
#   raise next i);;
i = 1 j = 1
i = 1 j = 3
i = 2 j = 1
i = 2 j = 3
() : unit
```

Recursivity and physical modifications

Using mutables you may easily create cyclic structures, even functional ones:

```
#(* Creation of a dummy closure for fib *)
#let fib_dummy = ref (I : num -> num);;
Value fib_dummy = (ref <fun>) : (num -> num) ref

#(* The body of fib refers to fib_dummy *)
#let fib_body =
```

```
# function 1 -> 1 | 2 -> 1 | n -> !fib_dummy(n-1)+!fib_dummy(n-2);;
Value fib_body = <fun> : (num -> num)

#(* Which is updated to fib_body to create a cyclic closure *)
#fib_dummy:=fib_body;;
<fun> : (num -> num)

#(* Now fib is defined to be the cyclic closure *)
#let fib = !fib_dummy;;
Value fib = <fun> : (num -> num)

#fib 20;;
6765 : num
```

Furthermore, we can avoid all these global declarations and simply write a recursive delcaration with no `let rec`:

```
#let fib = let fib=ref I in fib:=
# function 1 -> 1 | 2 -> 1 | n -> !fib(n-1)+!fib(n-2);;
Value fib = <fun> : (num -> num)

#fib 20;;
6765 : num
```

Forward and autoload

This kind of dummy declarations is extended to 2 toplevel constructs:

- "`forward`" <*Ident*> ":" <*Type*> declares a *function* of the given type, which has no associated value. The given type must be unifiable to a functional one. For instance:

  ```
  #(* Declaring a forward function *)
  #forward foo : num -> num;;
  Forward foo : (num -> num)

  #(* Binding this forward to a value, whose type must be
  #   more general than the declared type *)
  #let foo x = x;;
  Value foo = <fun> : (num -> num)

  #foo 1;;
  1 : num
  ```

  This construct may be used to built cyclic closures as well:

```
#forward fib : num -> num;;
Forward fib : (num -> num)

#let fib = function
#          1 -> 1 | 2 -> 1 | n -> fib(n-1) + fib(n-2);;
Value fib = <fun> : (num -> num)

#fib 20;;
6765 : num
```

There is no loss of efficiency with this kind of declarations, compared with a classical "`let rec`"[4].

- "`autoload`" <*Ident*> ":" <*Type*> "`from`" <*file_name*> declares a *function* of the given type, which has no associated value. This value will be found in the <*file_name*> compiled code file: at the first call to the function, the file is loaded and the corresponding value bound to the function.

These two constructs are extended to simultaneous declarations, using the keyword "and" between successive function names.

4.8 Mutable values in products

We start with a basic example: mutable pairs, and then a full example with queues.

4.8.1 Mutable pairs

Mutable pairs are records of two mutable fields, let us name them `Car` and `Cdr`[5]:

```
#type ('a,'b) mpair = {mutable Car : 'a; mutable Cdr : 'b};;
Type mpair defined
     .Car : (('a,'b) mpair -> 'a)
   ; .Cdr : (('b,'a) mpair -> 'a)
```

For mutable records, construction and accessing are as usual:

```
#let a_pair = {Car = 1; Cdr = "oui"};;
Value a_pair = {Car=1; Cdr="oui"} : (num,string) mpair

#a_pair.Car;;
1 : num
```

[4] At least in the present implementation.

[5] Which are the acronyms of Content of Address part of the Register and Content of Decrement part of the Register. Pourquoi pas ?

4.8.2 Updating mutable values in products

Update is also done by pattern matching:

```
#match a_pair with {Car = _ at occ; _} -> at occ <- 2;;
2 : num

#a_pair;;
{Car=2; Cdr="oui"} : (num,string) mpair
```

We have an extension of the "dot" notation to assign fields of records: `r.label <- expression` will update the field named label with the value of expression.

```
#a_pair.Cdr <- "non";;
"non" : string

#a_pair;;
{Car=2; Cdr="non"} : (num,string) mpair
```

4.8.3 Queues

As an example we will define an efficient implementation of "fifo" (first-in first out) stacks or queues. Queues are similar to lists, except that elements are not added in front but at the end of the list. For this purpose we will use mutable list cells and modify physically the end of the list when adding an element.

Hence we define the type of mutable list cells. Let us first define recursively the type of mutable cells and mutable queue bodies (list of cells) which may be empty or a "cons" of some cell:

```
#type 'a qbody = qnil | qcons of 'a cell
#and 'a cell = {mutable Head : 'a ; mutable Tail : 'a qbody};;
Warning: type qbody redefined
Type qbody defined
     qnil : 'a qbody
   | qcons : ('a cell -> 'a qbody)
Type cell defined
     .Head : ('a cell -> 'a)
   ; .Tail : ('a cell -> 'a qbody)
```

Now we can define queues which possess an insert pointer, which is the cell where a new element can be added and a "body" which contains the items enqueued:

```
#type 'a queue =
#      {mutable Insert_point : 'a qbody;
#       mutable Queue_body : 'a qbody};;
Warning: type queue redefined
```

```
Type queue defined
     .Insert_point : ('a queue -> 'a qbody)
   ; .Queue_body : ('a queue -> 'a qbody)
```

Now we present the function which given a cell and a queue will enqueue the cell: if the queue is empty then its body becomes the cell itself, and we store the cell in the insert point as well, since the insert point must be a pointer to the end of the queue. Otherwise we update the insert point with the new cell. This will automatically update the body of the queue, due to the physical sharing between the last cell of the queue and the insert point:

```
#let enqueue cell = function
#   {Queue_body=qnil;_} as queue ->
#     queue.Insert_point <- queue.Queue_body <- cell
#
# | {Queue_body=qcons _; Insert_point=qcons insert} as queue ->
#     insert.Tail <- cell;
#     queue.Insert_point <- cell
#
# | _ -> failwith "ill built queue";;
Value enqueue = <fun> : ('a qbody -> 'a queue -> 'a qbody)
```

The converse function successively gets the enqueued elements. It fails if the queue is empty, otherwise it returns the head of the queue and updates the queue body with the tail of the queue.

```
#let dequeue = function
#   {Queue_body=qnil;_} -> failwith "empty queue"
# | {Queue_body=qcons {Head=x;Tail=tl}; _} as queue ->
#      queue.Queue_body <- tl;x;;
Value dequeue = <fun> : ('a queue -> 'a)
```

Now we define the allocator of cells and queues for numbers:

```
#let new_num_cell (x:num) = qcons {Head=x;Tail=qnil};;
Value new_num_cell = <fun> : (num -> num qbody)

#let enqueue_num x q = enqueue (new_num_cell x) q;;
Value enqueue_num = <fun> : (num -> num queue -> num qbody)

#let new_num_queue () =
#    {Insert_point=qnil;Queue_body=(qnil:num qbody)};;
Value new_num_queue = <fun> : (unit -> num queue)

#let q = new_num_queue ();;
Value q = {Insert_point=qnil; Queue_body=qnil} : num queue
```

```
#enqueue_num 1 q;enqueue_num 2 q;enqueue_num 3 q;
#enqueue_num 4 q;enqueue_num 5 q;;
(qcons {Head=5; Tail=qnil}) : num qbody

#(* For advanced users: may be skipped at first reading *)
#show q;;
(&1 1 2 3 4 . &1) where  &1 = (5 . qnil)() : unit

#print_num (dequeue q);print_num (dequeue q);
#print_num (dequeue q);print_num (dequeue q);
#print_num (dequeue q);print_num (dequeue q);;
12345
Evaluation Failed: empty queue
```

4.9 Restrictions on mutable values and exceptions

There is a restriction on the type of mutable values *built* in a program: namely they can be of any monomorphic type, or must possess a polymorphic type with no free variables.

For instance, the empty list is polymorphic:

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

Thus we can use it as a list of any type:

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

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

#true::l;;
[true] : bool list
```

Hence we cannot define a mutable value including it:

```
#let x = ref [];;

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

Typecheck Failed
```

This is done for sake of safety since, if we could define a polymorphic reference containing the empty list we may then add to the list any element of any type, and this is not compatible with the definition of the type list which states that lists are indeed homogeneous:

Imagine that the types of polymorphic reference were generalized:

```
let l = ref [] in
l:=1::!l;;
l:=true::!l;;
```

At this step l would contain a number and a boolean!

This should be improved in the future using the so-called notion of "weak" type variables.

Notice that it is still possible to use polymorphic mutable values if there is no need to generalize their types:

```
#let l = ref [] in
#l:=1::!l;;
[1] : num list
```

and that the restriction is only for the *construction* of mutable values.

One may freely define a function with polymorphic mutable arguments, as long as no polymorphic mutable values are *built* inside the body of the function and returned as part of the result:

```
#(* This function does not built mutable values *)
#fun (ref x) -> x;;
<fun> : ('a ref -> 'a)

#(* This function attempts to build polymorphic mutable values *)
#fun x -> ref x;;

line 2: cannot generalize type 'a
for argument of mutable sum constructor ref
1 error in typechecking

Typecheck Failed
```

Finally, remember that the generalization of types is implicit at the end of every toplevel phrase, and is performed before each "in part" of a "let" construct in the CAML programs.

Beware:

-- The same restriction exists for exceptions.

4.10 Overloading of labels

Labels of records may be shared by several product types: this is a kind of *overloading* since a label may have several declared types. Thus the CAML typechecker uses a strategy to find out to which type a label belongs in a particular use of the label.

The main point to keep in mind is that the system allows any overloading, as soon as the proposed CAML phrases are not *completely* ambiguous.

Consider the types `foo` and `bar` sharing the same labels:

```
#type foo = {L1 : string ; L2 : string};;
Type foo defined
     .L1 : (foo -> string)
   ; .L2 : (foo -> string)

#type bar = {L1 : bool; L2 : string};;
Type bar defined
     .L1 : (bar -> bool)
   ; .L2 : (bar -> string)
```

Then the following are clearly not ambiguous:

```
#let x = {L1="poi"; L2="sdcs"};;
Value x = {L1="poi"; L2="sdcs"} : foo

#let y = {L1 = true; L2="poi"};;
Value y = {L1=true; L2="poi"} : bar
```

Nor is the following function (although it is a bit harder to prove this fact):

```
#fun {L1=x;L2=y} -> x=y;;
<fun> : (foo -> bool)
```

Nevertheless, the labels of the types `bar` and `foo` can create unresolvable ambiguities as in:

```
#function {L1=x; _} -> x;;

line 1: ill-typed phrase,
cannot resolve overloading ambiguity in {L1=x; _}
1 error in typechecking

Typecheck Failed
```

Open overloading

If you heavily use overloading over labels, you may ask the system to "open" overloading. That is to say that in case of complete ambiguity it chooses the last defined type and continues the typechecking (emitting a warning).

```
##open overloading true;;
Directive () : unit

#function{L1=x; _} -> x;;
Warning: Overloading solved by default ...
<fun> : (bar -> bool)
```

Beware:
-- This option should not be exercised in common programming situations, since this may lead to very complex typechecking problems, where more than one type may be acceptable for the same phrase!

Consider the following legal type definition:

```
#type rec0 = {lab:num;label:rec0}
#and  rec1 = {lab:num;label:rec2}
#and  rec2 = {lab:num;label:rec1};;
Type rec0 defined
     .lab : (rec0 -> num)
   ; .label : (rec0 -> rec0)
Type rec1 defined
     .lab : (rec1 -> num)
   ; .label : (rec1 -> rec2)
Type rec2 defined
     .lab : (rec2 -> num)
   ; .label : (rec2 -> rec1)
```

The types `rec0`, `rec1` and `rec2` share exactly the same labels, and in each type, the label `lab` contains values of type `num`.

Now consider the following legal phrase:

```
#let rec x = {lab=1;label={lab=2;label=x}};;
Warning: Overloading solved by default ...
Warning: unsafe recursive declaration
Value x =
      {lab=1;
       label=
         ......................... ...

limit_print_depth exceeded
 : rec1
```

It is completely ambiguous, and the default strategy of the system for resolving ambiguities has found `rec1` for the expression `{lab=1;label={lab=2;label=x}}`. Thus x has been declared with type `rec1`.

But now, just add a type constraint on x, constraining x to be of type `rec0`:

```
#let rec (x:rec0) = {lab=1;label={lab=2;label=x}};;
Warning: unsafe recursive declaration
Value x =
      {lab=1;
       label=
        ..... ...

limit_print_depth exceeded
 : rec0
```

rec0 is not an erroneous type for x!

Moreover constraining x to be of type rec1 or rec2 is still correct!

```
#let rec (x:rec1) = {lab=1;label={lab=2;label=x}};;
Warning: unsafe recursive declaration
Value x =
      {lab=1;
       label=
        ..... ...

limit_print_depth exceeded
 : rec1

#let rec (x:rec2) = {lab=1;label={lab=2;label=x}};;
Warning: unsafe recursive declaration
Value x =
      {lab=1;
       label=
        ..... ...

limit_print_depth exceeded
 : rec2
```

4.11 Lazy values

4.11.1 Freezing values

We define the type of "frozen" values, by using the lazy constructor Freeze:

```
#type 'a frozen = lazy Freeze of 'a;;
Type frozen defined
     Freeze : ('a -> 'a frozen)
```

Now we can evaluate:

```
#let frozen_two = Freeze (succ 1);;
Value frozen_two = (Freeze *) : num frozen
```

which does not compute the application succ 1. Instead, the CAML system has created a "thaw" (an unevaluated application), which is denoted by the toplevel pretty-printer with a * symbol. Notice that this is useless in the case of immediate constants involving no computation. In practice this is optimized:

```
#let one = Freeze 1;;
Value one = (Freeze 1) : num frozen
```

Evaluation of frozen values is done when *accessing* them (via pattern matching in right-hand side of clauses). Thus a general forcing function for elements of type frozen is:

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

since it accesses x:

```
#Unfreeze frozen_two;;
2 : num
```

As soon as forced, a frozen value is updated:

```
#frozen_two;;
(Freeze 2) : num frozen
```

Notice that the pattern matching process does not force the evaluation of thaws, if there is no need to access them:

```
#let Id (Freeze x as thaw) = thaw;;
Value Id = <fun> : ('a frozen -> 'a frozen)

#Id (Freeze (succ 1));;
(Freeze *) : num frozen
```

This is not always the case, compare with:

```
#let is_frozen_two = function
#   Freeze 2 -> true
# | _ -> false;;
Value is_frozen_two = <fun> : (num frozen -> bool)

#let y = (Freeze (succ 2));;
Value y = (Freeze *) : num frozen

#is_frozen_two y;;
false : bool

#y;;
(Freeze 3) : num frozen
```

4.11.2 Lazy pairs

The type of lazy pairs is a bit more complex:

```
#type ('a,'b) lpair = {lazy Lfst: 'a; lazy Lsnd:'b};;
Type lpair defined
     .Lfst : (('a,'b) lpair -> 'a)
   ; .Lsnd : (('b,'a) lpair -> 'a)
```

Let us define a lazy pair:

```
#let lazy_pair = {Lfst=succ 1;Lsnd=succ 2};;
Value lazy_pair = {Lfst=*; Lsnd=*} : (num,num) lpair
```

and destructors for lazy pairs:

```
#let lfst {Lfst=x;Lsnd=_} = x
#and lsnd {Lfst=_;Lsnd=x} = x;;
Value lfst = <fun> : (('a,'b) lpair -> 'a)
Value lsnd = <fun> : (('a,'b) lpair -> 'b)
```

Now notice the *minimal* forcing done by `lfst` and `lsnd`:

```
#lfst lazy_pair;;
2 : num

#lazy_pair;;
{Lfst=2; Lsnd=*} : (num,num) lpair

#lsnd lazy_pair;;
3 : num

#lazy_pair;;
{Lfst=2; Lsnd=3} : (num,num) lpair
```

Using the laziness of this type you may then recursively define a function and a constant value using this function. We recursively define a lazy pair whose components are the function and the constant value which uses it, as the lazy pair of 2 expressions. This builds frozen values which are forced by the toplevel when defining the two global identifiers:

```
#let rec {Lfst=fact5 ; Lsnd=fact} =
#    {Lfst=fact 5;
#     Lsnd=function 1 -> 1 | n -> n * fact (n-1)};;
Value fact5 = 120 : num
Value fact = <fun> : (num -> num)
```

4.11.3 Lazy lists or streams

Now we continue with lazy lists, which will allow us to compute with potentially infinite objects:

```
#type 'a stream = {lazy Hd : 'a ; lazy Tl : 'a stream};;
Type stream defined
     .Hd : ('a stream -> 'a)
   ; .Tl : ('a stream -> 'a stream)
```

We need a map functional to apply a function on each element of a stream:

```
#let rec map_stream f = function
# {Hd=x;Tl=t} -> {Hd=f x;Tl = map_stream f t};;
Value map_stream = <fun> :
      (('a -> 'b) -> 'a stream -> 'b stream)
```

4.11.4 Potentially infinite lists

The stream of natural numbers begins with 0 and its tail is just the mapping of the successor function on itself!

```
#let rec (Freeze Nat) =
#    Freeze {Hd=0;Tl = map_stream succ Nat};;
Value Nat = {Hd=0; Tl=*} : num stream
```

Notice the use of `Freeze` to be sure that lazy values are built, which are automatically forced by the toplevel when accessing `Nat` (while defining it).

Notice that this is not magic:

```
#let rec (Freeze x) = (Freeze x);;

Undefined
```

The system reports "undefined" when a thaw needs its own value to be completely evaluated.

The nth element of a stream is computed by:

```
#let rec nth_stream n {Hd=h;Tl=t} =
# if n<=1 then h
#  else nth_stream (n-1) t;;
Value nth_stream = <fun> : (num -> 'a stream -> 'a)

#nth_stream 5 Nat;;
4 : num
```

A simple recursive function will display the beginning of a stream up to a given integer rank:

```
#let rec display_num_stream n =
# if n = 0 then (fun _ -> ())
#  else
#  (function {Hd=x;Tl=t} ->
#    display_num x;display_string" ";display_num_stream (n-1) t);;
Value display_num_stream = <fun> :
      (num -> num stream -> unit)

#display_num_stream 10 Nat;;
0 1 2 3 4 5 6 7 8 9 () : unit
```

Notice the automatic update of Nat:

```
#Nat;;
{Hd=0;
 Tl=
  {Hd=1;
   Tl=
    {Hd=2;
     Tl=
      {Hd=3;
       Tl=
        {Hd=4;
         Tl=
          {Hd=5;
           Tl=
            {Hd=6;
             Tl=
              {Hd=7; Tl={Hd=8; Tl={Hd=9; Tl={Hd=*; Tl=*}}}}}}}}}}
 } : num stream
```

To compute the stream of even numbers, we just have to add Nat to Nat, i.e. to apply the function add to each pair of elements of Nat and of another instance of Nat in turn.

Thus we need a functional to apply a function with two arguments on two streams:

```
#let rec map_stream2 f = function
#  {Hd=x1;Tl=t1} ->
# function {Hd=x2;Tl=t2} ->
#  {Hd=f x1 x2;Tl = map_stream2 f t1 t2};;
Value map_stream2 = <fun> :
      (('a -> 'b -> 'c) -> 'a stream -> 'b stream ->
       'c stream)

#nth_stream 20 (map_stream2 add Nat Nat);;
38 : num
```

Now we define the stream of Fibonacci numbers: it starts with the value 1 then 1 again, and afterwards the nth number is obtained by addition of the nth-1 and nth-2 numbers of the stream:

```
#let rec (Freeze Fib) =
#    Freeze {Hd=1; Tl={Hd=1; Tl=map_stream2 add Fib Fib.Tl}};;
Value Fib = {Hd=1; Tl=*} : num stream

#display_num_stream 10 Fib;;
1 1 2 3 5 8 13 21 34 55 () : unit
```

In the same spirit it is possible to define the stream of prime numbers.
We use the well known sieve of Erathostenes:

```
#let rec sieve p = sieve_rec where rec sieve_rec = function
# {Hd=x;Tl=t} -> if x mod p = 0 then sieve_rec t
#                 else {Hd=x;Tl=sieve_rec t};;
Value sieve = <fun> : (num -> num stream -> num stream)
```

Now the function `primes` which will recursively sift the tail of the stream of natural numbers:

```
#let rec primes = function
# {Hd=x;Tl = t} -> {Hd=x;Tl= primes (sieve x t)};;
Value primes = <fun> : (num stream -> num stream)
```

Now we just need to sift natural numbers starting from 2:

```
#let rec (Freeze Nat_from_2) =
#    Freeze {Hd=2;Tl = map_stream succ Nat_from_2};;
Value Nat_from_2 = {Hd=2; Tl=*} : num stream

#let Primes = primes Nat_from_2;;
Value Primes = {Hd=*; Tl=*} : num stream
```

We have "obtained" the whole set of prime numbers! (it just remains to be expanded as needed!)

```
#display_num_stream 10 Primes;;
2 3 5 7 11 13 17 19 23 29 () : unit
```

4.11.5 Formal series

Our aim is to define the function `sinus` and `cosinus` as the streams of the coefficients of their serial developments.

First we define the negation of a stream:

```
#let sf_uminus = map_stream (fun x -> (-x));;
Value sf_uminus = <fun> : (num stream -> num stream)
```

and an integrating function for series:

```
#(* Formal series integration fonction *)
#let integrate l =
#    (intrec 0 l)
#    where rec intrec n {Hd=x;Tl=l} =
#        {Hd=(x/(n+1));Tl=(intrec (n+1) l)}
#;;
Value integrate = <fun> : (num stream -> num stream)
```

We define the stream $(-1)^n$:

```
#let rec (Freeze l) = Freeze {Hd=1;Tl=(sf_uminus l)};;
Value l = {Hd=1; Tl=*} : num stream

#display_num_stream 10 l;;
1 -1 1 -1 1 -1 1 -1 1 -1 () : unit
```

and then we integrate it to get:

```
#let log1 = {Hd=0;Tl=integrate l};;
Value log1 = {Hd=0; Tl=*} : num stream

#display_num_stream 10 log1;;
0 1 -1/2 1/3 -1/4 1/5 -1/6 1/7 -1/8 1/9 () : unit
```

and the recursive definition of sinus and cosinus as the integrals of each other, just giving the first element of the series:

```
#let rec (Freeze {Lfst=cosinus;Lsnd=sinus}) =
#        Freeze {Lfst={Hd=1;Tl=integrate (sf_uminus sinus)};
#                Lsnd={Hd=0;Tl=integrate (cosinus)}};;
Value cosinus = {Hd=1; Tl=*} : num stream
Value sinus = {Hd=0; Tl=*} : num stream
```

To get the first elements of a stream as a list:

```
#let rec stream_to_list S =
#    function 0 -> []
#           | n -> S.Hd::stream_to_list S.Tl (n-1);;
Value stream_to_list = <fun> :
      ('a stream -> num -> 'a list)
```

and finally the first elements of sinus and cosinus:

```
#stream_to_list sinus 5;;
[0; 1; 0; -1/6; 0] : num list
```

```
#stream_to_list cosinus 5;;
[1; 0; -1/2; 0; 1/24] : num list

#sinus;;
{Hd=0;
 Tl=
  {Hd=1;
   Tl={Hd=0; Tl={Hd=-1/6; Tl={Hd=0; Tl={Hd=*; Tl=*}}}}} :
 num stream
```

4.11.6 Forcing lazy values

A primitive construct is provided to (recursively) force the evaluation of a lazy value:

```
#let x = Freeze (succ 1);;
Value x = (Freeze *) : num frozen

#force x;;
(Freeze 2) : num frozen

#x;;
(Freeze 2) : num frozen
```

A toplevel directive is also available to force automatically the evaluation when printing lazy values. If you want to get this feature type in `#eval when print true;;`. Be careful with potentially infinite data structures!

4.12 Redefining types

When you redefine a type (i.e. when you define a type with the same name as an existing one), all its constructors or labels are removed from the current global symbol table: they become "unbound". In addition, the system emits a warning. Identifiers bound to values of the old type (which is now "out of scope") do not disappear. Their type is now written suffixed by a question mark:

```
#type barfoo = Barfoo of num;;
Type barfoo defined
     Barfoo : (num -> barfoo)

#let x = Barfoo 1;;
Value x = (Barfoo 1) : barfoo

#type barfoo = Foobar of string;;
Warning: type barfoo redefined
```

```
Type barfoo defined
     Foobar : (string -> barfoo)

#Barfoo;;

line 1: unbound variable Barfoo
1 error in typechecking

Typecheck Failed

#x;;
(Barfoo 1) : barfoo?
```

4.13 Implementation issues

All values in CAML are implemented using values of the predefined type `obj` (more on objects in section 20):

```
type obj = obj_int of int | obj_float of float |
           obj_string of string | obj_atom of atom |
           obj_vect of obj vect | obj_cons of obj * obj
;;
```

There is a function which maps a value on its representation

♣ `Repr : ('a -> obj)`

The representation algorithm for concrete types is fairly sophisticated: it tends to use as litle memory allocation as possible. In particular it employs the concept of "superfluous" constructors, which are removed from the internal representation of data. For instance

```
#Repr (Foobar "oui");;
<:obj<"oui">> : obj
```

the constructor `Foobar` is superfluous.

One can inspect the representation of values with the function `info` which gives information about identifiers:

♣ `info : (string -> unit)`

```
#info"barfoo";;
barfoo is :
 -- a concrete type with :
      superfluous constructor Foobar : (string -> barfoo)
() : unit
```

Chapter 5

Arithmetic and boolean operations

We now start to describe the primitives operations provided on basic types: in this chapter we examine primitives for numbers and booleans.

Since the CAML system is written in CAML (it is *bootstrapped*) the set of primitives available to the user is the one needed in the implementation of the language.

5.1 Infix identifiers

You may write in CAML `3+2`: in this expression "+" is written in "*infix form*" that is to say that "+" appears *between* the arguments. But the addition is clearly a binary operation: thus its type is `(num * num) -> num`. In other words we should apply the addition on the *pair* `(3,2)`, writing `+ (3,2)`: in this expression the addition appears in "*prefix form*". In fact CAML knows the function "`prefix +`", whose type is `(num * num) -> num`. But since it is much more convenient and natural to write `2 + 3`, CAML provides the user with the infix form as a syntactical commodity: "`x + y`" is expanded as "`prefix + (x,y)`".

```
#+;;

line 1 Syntax error:
Skipping: + ;;
Parse Failed

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

#prefix + (3,2) = 3+2;;
true : bool
```

In this chapter many primitives will have an infix form.

5.2 Booleans

Definition

The type `bool` has two constant constructors, `true` and `false`. For efficiency reasons the type `bool` is a primitive type, but `bool` could have been declared in CAML in the following way (but note that this definition would produce a syntax error, since `true` and `false` are reserved keywords):

```
type bool = true | false;;
```

Conditional

The *conditional* language construct `if b then e1 else e2`, evaluates the expression `b` and if the result is `true` (resp. `false`), the expression `e1` (resp. `e2`) is evaluated and its result is the result of the whole expression. Remark that the conditional is a lazy construct in the sense that only one of the expressions (`e1`,`e2`) is evaluated.

The two principal logical connectors (`&` and `or`) are defined in terms of the conditional.

- "`e1 & e2`" is equivalent to "`if e1 then e2 else false`".
- "`e1 or e2`" is equivalent to "`if e1 then true else e2`".

♣ `prefix not : (bool -> bool)`

- `not e1` is equivalent to `if e1 then false else true`.

♣ `prefix & : (bool * bool -> bool)`
`prefix or : (bool * bool -> bool)`

Not all the infix operators have a corresponding prefix version: even though the symbol `&` is used in infix position, as in `(x=1)&(y=2)`, `prefix &` is *not equivalent* to the symbol `&`, since it is impossible to define a call-by-value function corresponding to `&` (because `e & e'` evaluates to `false` as soon as `e` does, even if the evaluation of `e'` would fail or loop).

Since `e & e'` is expanded on the fly as the CAML phrase: `if e then e' else false`, clearly if `e` evaluates to false `e'` is *not evaluated.*

Note that it is no help to define:

```
#let boolean_and e e' = e & e';;
Value boolean_and = <fun> : (bool -> bool -> bool)
```

because when the body of boolean_and is executed e and e' will have been completely evaluated:

```
#boolean_and false fail;;

Evaluation Failed: fail

#false & fail;;
false : bool
```

Thus prefix operator "prefix &" is a real *function*, equivalent to the previous boolean_and.

A similar phenomenon exists with or.

```
#prefix or (true,true);;
true : bool

#prefix or (true,fail);;

Evaluation Failed: fail

#true or fail;;
true : bool
```

It is possible to define functions to implement other logical connectives, as nand or nor. Even the conditional can be expressed in terms of a function, but notice that the laziness of the conditional is not inherited by these functions:

```
#let nand x y = not (x & y)
#and nor x y = not (x or y)
#and cond test t f = if test then t else f;;
Value nand = <fun> : (bool -> bool -> bool)
Value nor = <fun> : (bool -> bool -> bool)
Value cond = <fun> : (bool -> 'a -> 'a -> 'a)

#let careful_divide x y =
# if y = 0 then x else x quo y
#and careful_divide2 x y =
# cond (y = 0) x (x quo y);;
Value careful_divide = <fun> : (num -> num -> num)
Value careful_divide2 = <fun> : (num -> num -> num)

#careful_divide 1 0;;
1 : num
```

```
#careful_divide2 1 0;;

Evaluation Failed: quo
```

♣ `neg : (('a -> bool) -> 'a -> bool)`

neg is the functional version of not. It can be used to negate a predicate:

```
let neg predicate x = not predicate x
;;
```

Beware:
-- not has a particular precedence distinct from the one of application. Thus the above definition is equivalent to:

```
let neg predicate x = not (predicate x);;
```

You can not use the η-rule of λ-calculus to simplify it!

5.3 Equality

In mathematics, equality is usually defined as identity. In a computer the story is not so simple; since different "computed objects" can be created which are mathematically identical but physically completely different (i.e. stored in different memory locations). Another problem arises when objects share common sub-structures: we may have cycles in the representation of objects and the equality test should take care of this.

So we can distinguish two different kinds of equality: identity of computer locations (which evidently ensures equality of objects), and equality which tests if objects are isomorphic.

5.3.1 Identity

♣ `eq : ('a * 'a -> bool)`
`prefix == : ('a * 'a -> bool)`

eq means equality of adresses where objects are stored (i.e. physical identity of the locations holding the representations in the computer memory). Thus `eq (e1,e2)` (or equivalently `e1 == e2`) tests the *identity* (internally the same object) of `e1` and `e2`. This primitive is very efficient since it just performs *one* pointer comparison.

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

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

#eq(!x,!y);;
true : bool
```

Be careful with identity of numbers:

- semantical equality of small integers is identity.
- semantical equality of floating numbers is identity (up to precision problems).
- semantical quality of other kinds of numbers is not identity.

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

#1/2 == 3/2 - 1;;
false : bool

#1.0 == 2.0 - 1.0;;
true : bool
```

5.3.2 Equality

Primitives for equality

♣
```
prefix = : ('a * 'a -> bool)
prefix <> : ('a * 'a -> bool)
equal : ('a * 'a -> bool)
```

- `e1 = e2` compares the two values of the same type and produces **true** when they are *equal*.
- `e1 <> e2` is equivalent to `not e1=e2`.
- `equal (e1,e2)` tests the *equality* of `e1` and `e2`, if they are *not functional objects* (in which case `equal` fails with the string `"equal"`). `equal` is more sophisticated than = since it is able to handle cyclic data (for which = may loop). On the other hand, `equal` is less efficient than =, since it performs much more computation to prove that data are isomorphic.

All this operators are primitives in CAML.

Beware:

-- All these functions are *undefined* when applied to frozen values. Before the use of one of them with such lazy values you have to force the values to ensure their complete evaluation. -- `prefix =` is in **no way** the same function as **equal** (see above).

5.3.3 Semantics of equality

The primitive =

The use of the well-known symbol = suggests that `prefix =` is bound to the corresponding expected predicate. It is indeed true for values of non functional types. But note the following points:

Beware:

-- `prefix =` **may loop** on cyclic values, because it examines recursively all the components of its arguments. This is particularly the case for recursive functional values. See the primitive **equal** in the following to test equality in such a case.

-- `prefix =` is in no way the mathematical extensional equality when used with functions. In this case `prefix =` means identity (i.e. identity of representations in the computer memory). So, necessarily, semantically distinct functions would not be equal, since their representation are not physically the same. But different occurrences of the same textual expression may build functions which are extensionally equal but not recognized as such by `prefix =`:

```
#let f x = x+1;;
Value f = <fun> : (num -> num)

#let g x = x+1;;
Value g = <fun> : (num -> num)

#f=g;;
false : bool
```

f and g are the same mathematical function, but `prefix =` does not detect this because their internal representations are really different objects. Compare with:

```
#let h = f;;
Value h = <fun> : (num -> num)

#h=f;;
true : bool
```

Different evaluations of the same textual expression may build equal objects:

```
#let f = add 1
#and g = add 1 ;;
Value f = <fun> : (num -> num)
Value g = <fun> : (num -> num)

#eq (f,g);;
false : bool

#f=g;;
true : bool
```

the two distinct calls to the same curried function with the same argument leads to distinct objects, but `prefix =` is able to prove them isomorphic.

For references, equality means equality of contents:

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

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

#x=y;;
true : bool

#!x=!y;;
true : bool

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

x and y are different locations even though they hold the same value.

In short, non functional values are equal iff they are semantically equal, whereas function equality is *unreliable.*

```
#let f = I;;
Value f = <fun> : ('a -> 'a)

#f = I;;
false : bool

#let g = f;;
Value g = <fun> : ('a -> 'a)

#g = f;;
true : bool
```

The primitive equal

♣ `equal : ('a * 'a -> bool)`

This is "cautious" equality since **equal** tests equality without looping on values having cycles. It fails with `"equal"` if you try to compare functions which are not identical ("eq") and is "eq" for references.

```
#let f = I;;
Value f = <fun> : ('a -> 'a)

#equal(f,I);;

Evaluation Failed: equal

#let g = f;;
Value g = <fun> : ('a -> 'a)

#equal(g,f);;
true : bool
```

5.4 Numbers

Numbers consist of:

- small integers: type `int`.
- floating point representations of reals: type `float`.
- symbolic representations of large integers and rationals, internally represented as structures of type `obj` (see chapter 20).

These three sorts of numbers are grouped in one predefined type of numbers: num. Most of the time the casual user will manipulate num values without worrying about their representations.

More sophisticated users will take advantage of the definition of the num type:

```
type num = Int of int | Float of float |
           Big_num of obj vect
;;
```

The elements of the type num are completed by the special values, *infinity* (`1/0` or `-1/0`) and *undefined* (`0/0`). This is a predefined type in CAML as well as most of its operators.

5.4.1 Reading numbers

There are special conventions to input the different kinds of numbers.

Reading num elements

One may type in such numbers with the following syntax:

```
NUM ::=  digit+ {'.' digit+} {'e' {'+' | '-'} digit+}
```

which stands for:

- a number is a non null sequence of digits,
- optionally followed by a dot and a non null sequence of digits,
- optionally followed by e which is followed optionally by + or - and a non null sequence of digits.

Rule of thumb: a number always begins and ends by a digit.

Reading Rationals and Naturals

The lexical convention above applies to naturals, for which it is just:

```
digit+
```

The notation for rationals is the common mathematical one, i.e. two integers separated with a / symbol:

```
#1/2;;
1/2 : num

#1/-2;;
-1/2 : num

#(* In this case + is parsed as the binary operator *)
#+1/2;;

line 2 Syntax error:
Skipping: + 1 / 2 ;;
Parse Failed
```

Beware:
-- This notation is *not* a lexical convention: rationals do not belong to a special lexical class.

Reading floating point numbers

Some elements of the type **num** are floating point numbers, in a range depending on the machine which runs the CAML system but generally $-1\ 10^{38}$; $+1\ 10^{38}$.

```
#1.0;;
1.0 : num

#1e2;;
100.0 : num

#1e-1;;
0.09999999 : num

#1.2e+2;;
120.0 : num

#1.;;

line 1 Syntax error:
Skipping: ;;
Parse Failed

#.1;;

line 1 Syntax error:
Skipping: . 1 ;;
Parse Failed
```

Reading "small" Integers

Elements of type **int** are "small integers" in the range -32767, +32767 (more exactly the ring, since there is no notion of overflow). Integer values of type **int** are mainly manipulated for the sake of efficiency, since their representation is only a 16 bit value (moreover they are not allocated, hence there is no need to garbage collect them).

The maximum value of a "small" integer is:

♣ `max_int : num`

```
let max_int = 32767
;;
```

CAML is able to read small integers entered directly, when you type in a number preceded by a sharp character "#", following the lexical convention:

```
INT ::= '#' {'-' | '+'} digit+

##32767;;
#32767 : int

##-1;;
#-1 : int

##+1;;
#1 : int

##32768;;

line 1 Syntax error:
Skipping: 3 2 7 6 8 ;;
Parse Failed
```

Reading numbers does some magic

When you type in a number (of type num) the constructors of the type num are automatically added by the lexical analyser : typing-in 1 is equivalent to type in `Int #1` and analogously for `1.1` and `Float #1.1`. For big numbers (or rationals) the corresponding vector of elements of the type `obj` is built and applied to the constructor `Big_num`.

5.4.2 Primitive operators for numbers

Most of the operations on numbers are "generic", which means that they are able to handle every kind of number, with implicit coercions if necessary (e.g. basic arithmetic operations +, *, - and /). Some others are specialized for one precise kind of numbers as is the primitive operation `succ_int : int -> int`.

Roughly speaking, the user needs not to worry about all these subtleties about numbers and uses as numbers the elements of the predefined type num which contains implicitly all the other kinds of numbers, along with all its operations which "know" how to handle the necessary coercions.

Addition

♣ `prefix + : (num * num -> num)`

n1+n2 gives the result of the sum of n1 and n2.
Example:

```
#1/3 + 2/3;;
1 : num
```

Beware:
-- The normalisation of rational numbers is *not* done by the addition: while computing with rationals, the results are *not normalised.*

The normalisation process only takes place *when printing*: thus the runtimes reported by the CAML system, which does not include the time necessary to print the results, may be much less than the real evaluation time.

Subtraction

♣
```
prefix - : (num * num -> num)
minus : (num -> num)
```

`prefix -` is the binary subtraction function. **`minus`** is the unary function (unary minus) which returns the negation of its single numeric argument: **`minus n`** or **`-n`** are equivalent to `0-n`.

```
#prefix - (7,3);;
4 : num

#minus 7;;
-7 : num

let minus n = 0-n
;;
```

Example:

```
#52131--5212+-123;;
57220 : num

#let x = 5 in -x;;
-5 : num
```

Multiplication and Division

♣
```
prefix * : (num * num -> num)
prefix / : (num * num -> num)
```

`n1*n2` gives `n1` times `n2` as result. **`n1/n2`** gives **`n1`** divided by **`n2`**, these are total functions and both are primitives.
Example:

```
#255452*222*4545;;
257748513480 : num

#(7/3) / 2;;
7/6 : num

#5/0;;
1/0 : num
```

Absolute value

♣ `abs : (num -> num)`

`abs x` returns the absolute value of the number `x`.

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

5.4.3 Incrementing and decrementing

Incrementing and decrementing numbers

♣ `pred : (num -> num)`
`succ : (num -> num)`

These are bound to the mathematical successor and predecessor functions when used with integers: `pred n` (resp `succ n`) is a primitive equivalent to `n-1` (resp `n+1`).

```
let pred x = x-1
and succ x = x+1;;

#succ 1;;
2 : num

#succ 1.2e3;;
1201.0 : num

#succ (1/2);;
3/2 : num

#(* Beware of precedence *)
#succ 1/2;;
1 : num
```

Incrementing and decrementing counters

♣
```
incr : (num ref -> num)
decr : (num ref -> num)
```

When one has created a reference containing a number (whose type is **num**), it is convenient to increment and decrement its value, using these two (primitive) functions semantically equivalent to:

```
let incr i = i:= succ !i and decr i = i:= pred !i;;
```

5.4.4 Curried operators

♣
```
add : (num -> num -> num)
sub : (num -> num -> num)
mult : (num -> num -> num)
div : (num -> num -> num)
```

add, sub, mult and div are the curried version of `prefix +`, `prefix -`, `prefix *` and `prefix /`:

```
let add x y = x+y and sub x y = x-y and mult x y = x*y
and div x y = x/y
;;

#let add5 x = add 5 x in
# add5 20,add5 10;;
(25,15) : (num * num)
```

5.4.5 Integer results

Some of the operations on numbers are made to take or produce only integers.

♣
```
prefix quo : (num * num -> num)
prefix mod : (num * num -> num)
integer : (num -> num)
floor : (num -> num)
round : (num -> num)
numerator : (num -> num)
denominator : (num -> num)
```

- `n1 quo n2` is the integer (part of a) division.
- `n1 mod n2` gives the remainder of an integer division and for integers `n1` and `n2` is equivalent to `n1 - (n1 quo n2)*n2`. Both `quo` and `mod`, fail to produce a result when `n2` is either `0` or `0/0`.
- The function `integer` produces the integer part of a number whereas `floor` returns the mathematical integer part (`floor x` is the greatest integer lower or equal to x). These two functions differ with no-positive numbers arguments. Both are useful, since if you want to compute each digit of a rational you need `integer`:

```
#integer (-1/2);;
0 : num

#floor (-1/2);;
-1 : num
```

- The function `round` returns the nearest integer of a number.
- Finally `numerator` and `denominator` give access to the two components of a rational.

Example:

```
#(7/3) quo 2;;
1 : num

#(7/3) mod 2;;
1/3 : num

#0 quo 0;;

Evaluation Failed: quo

#1.2 quo 1;;
1 : num

#0 mod 0;;

Evaluation Failed: mod

#integer (1/3);;
0 : num

#integer (-0.5);;
```

```
0 : num

#integer (-1);;
-1 : num

#numerator (30/14);;
15 : num

#denominator(30/14);;
7 : num

#denominator 5;;
1 : num

#denominator 3.14;;
1 : num

let floor x = x quo 1
;;

let round x = (x quo 1)+(if x mod 1 < 1/2 then 0 else 1)
;;

#floor (-0.5);;
-1 : num

#floor 1.2;;
1 : num

#round 1.4;;
1 : num

#round 1.5;;
2 : num
```

5.4.6 Logical operations

♣
```
land : (num * num -> num)
lor : (num * num -> num)
lxor : (num * num -> num)
lshift : (num * num -> num)
lnot : (num -> num)
```

These are "logical" functions which operate on numbers *bitwise*. These numbers must be "small" integers, that is 16 bits words. All these functions are expanded on the fly into one or a few lines of assembly code. Their meanings are:

- `land (n1,n2)`: logical "and" between `n1` and `n2`. For instance to find if a number is a power of 2, you may define

```
#let is_power2 n = n=land(n,-n);;
Value is_power2 = <fun> : (num -> bool)

#is_power2 4;;
true : bool

#is_power2 3;;
false : bool
```

- `lor`: logical "or".
- `lxor`: exclusive "or".
- `lshift (n1,n2)`: shifts `n1` of `n2` positions. If `n2` is positive the shift is done to the left (towards the most significant bits, hence it is a multiplication) If `n2` is negative the shift is done to the right (towards the least significant bits, then it is a division). Using lshift you may extract a bit field from an integer representation, as in:

```
#let extract n i = lshift (land (n, lshift (1,i)), -i);;
Value extract = <fun> : (num -> num -> num)

#map (extract 24) (interval 0 15);;
[0; 0; 0; 1; 1; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0] : num list

#map (extract (-24)) (interval 0 15);;
[0; 0; 0; 1; 0; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1] : num list
```

- `lnot n`: returns the logical complement of `n`.

5.4.7 Floating point operations

Some of the operations on numbers are made to take or produce only numbers which are representations of floats. These are the classical:

♣
```
sin : (num -> num)
cos : (num -> num)
asin : (num -> num)
```

```
acos : (num -> num)
atan : (num -> num)
exp : (num -> num)
log : (num -> num)
log10 : (num -> num)
power : (num * num -> num)
sqrt : (num -> num)
float : (num -> num)
```

- The function `float` produces the floating point representation of a number.
- Other primitives may be applied to numbers which are automatically coerced to a floating point representation if necessary.

As an example one can create a function `tan` using the mathematical definition:

```
#let tan x = (sin x) / (cos x);;
Value tan = <fun> : (num -> num)

#tan 1;;
1.557408 : num

#atan it;;
0.9999999 : num

#let pi_over_2 = 3.14159265/2;;
Value pi_over_2 = 1.570796 : num

#(* Computation with floating point numbers approximates *)
#tan pi_over_2;;
13245400.0 : num

#(* But sometimes consistently ! *)
#atan it;;
1.570796 : num
```

5.4.8 Small integers operations

The normal use of arithmetic must be done with elements of the type `num`, and thus very few operations are provided for the type `int`.

One can create integers from element of type `num` with explicit coercions:

♣
```
int_of_num : (num -> int)
num_of_int : (int -> num)

let int_of_num = function
    Int n -> n
  | B ->
      if is_integer B
       then match integer B with
              Int n -> n | _ -> failwith "int_of_num"
       else failwith "int_of_num"
and num_of_int = Int
;;
```

Example:

```
#int_of_num 32767;;
#32767 : int

#int_of_num 32768;;

Evaluation Failed: int_of_num
```

In addition there is a coercion between strings and integers:

♣
```
string_of_int : (int -> string)
```

Basic operations for "small" integers

Most often, small integers are used to create counters, generally initialized to zero. So the two specialized primitive operations `succ_int` and `pred_int` are provided. In addition, specialized versions of arithmetic operations are provided.

♣
```
succ_int : (int -> int)
pred_int : (int -> int)
add_int : (int * int -> int)
mult_int : (int * int -> int)
sub_int : (int * int -> int)
quo_int : (int * int -> int)
le : (int * int -> bool)
ge : (int * int -> bool)
lt : (int * int -> bool)
gt : (int * int -> bool)
```

```
#succ_int #0;;
#1 : int

#it - 1;;

ill-typed phrase, the variable it of type int
cannot be used with type instance num in it
1 error in typechecking

Typecheck Failed

#add_int(#1,#2);;
#3 : int

#add_int(#32767,#10);;
#-32759 : int

#mult_int(#32767,#2);;
#-2 : int

#ge (#1,#2);;
false : bool
```

- `add_int`, `mult_int`, `sub_int` and `quo_int`, are the four arithmetic operations on integers.

- `le`, `ge`, `lt` and `gt`, are arithmetic comparisons corresponding respectively to <=, >=, < and >.

Notice that the specialized versions of the arithmetic operations are primitives. In a sense they are not "cautious", since they do not check for overflow.

A careful (but less efficient) version may be defined as in:

```
#let mult_int_care (x,y) =
# match (Int x) * (Int y)
# with Int result -> result
#    |      _      -> failwith "overflow"
#;;
Value mult_int_care = <fun> : (int * int -> int)

#mult_int_care(#32767,#2);;

Evaluation Failed: overflow
```

"int" Counters

If a reference containing an integer is built, one can define a general function to increment such a reference with:

```
#let incr_int x = x:=succ_int !x
#and decr_int x = x:=pred_int !x;;
Value incr_int = <fun> : (int ref -> int)
Value decr_int = <fun> : (int ref -> int)
```

Recall that equality between small integers may be tested with =, or equivalently with eq (since small integers are uniquely represented).

5.4.9 Numbers of type float

One can create floats (numbers of type `float`) from elements of type `num` with explicit coercions:

♣
```
float_of_num : (num -> float)
num_of_float : (float -> num)

let float_of_num n =
 (let float_of_num_float = function
      Float f -> f | _ -> failwith "float_of_num" in
   float_of_num_float (float n))

and num_of_float = Float
;;
```

In addition there is a coercion between strings and floats:

♣
```
string_of_float : (float -> string)
```

Basic operations for floats

♣
```
add_float : (float * float -> float)
sub_float : (float * float -> float)
mult_float : (float * float -> float)
div_float : (float * float -> float)
```

These are basic operations specialized to elements of type "`float`".

5.4.10 Comparisons between numbers

♣
```
prefix > : (num * num -> bool)
prefix < : (num * num -> bool)
prefix <= : (num * num -> bool)
prefix >= : (num * num -> bool)
```

These are the usual predicates for numbers of type num.

```
#1.1 < 3/2;;
true : bool
```

♣
```
min : (num -> num -> num)
max : (num -> num -> num)
```

min and max are bound to the expected curried functions.

```
let max x y = if x <= y then y else x
and min x y = if x <= y then x else y
;;
```

5.4.11 Miscellaneous operations for numbers

Discriminators

♣
```
is_float : (num -> bool)
is_int : (num -> bool)
is_integer : (num -> bool)
```

- the function is_float allows you to test if a value of type num is in fact a floating number.
- one can test if a given number is a small integer using is_int, and if a number is an integer value with is_integer.

```
let is_float = function Float _ -> true | _ -> false
;;

let is_int = function Int _ -> true | _ -> false
;;
```

```
#is_int 12;;
true : bool

#is_integer 12;;
true : bool

#is_int 1234567890;;
false : bool

#is_integer 1234567890;;
true : bool

#is_int 1.0;;
false : bool

#is_integer 1.0;;
false : bool

#is_int (1 / 2);;
false : bool

#is_integer (1 / 2);;
false : bool
```

Coercion between objects and numbers

♣

```
num_of_obj : (obj -> num)
int_of_obj : (obj -> int)
float_of_obj : (obj -> float)
```

These are the usual coercions between representations of numbers and numbers.

```
let int_of_obj = function
    obj_int i -> i | _ -> failwith "int_of_obj"
and float_of_obj = function
    obj_float f -> f | _ -> failwith "float_of_obj"
;;
```

Coercion between strings and numbers

♣
```
string_of_num : (num -> string)
num_of_string : (string -> num)
```

- `string_of_num` and `num_of_string` maps element of type `num` onto corresponding elements of type `string`, such that the following relation holds for each `n` from type `num`:

```
num_of_string (string_of_num n) = n
```

♣
```
string_of_integer : (int -> string)
string_of_floating : (float -> string)
string_of_big_num : (obj vect -> string)
```

- `string_of_integer`, `string_of_floating` and `string_of_big_num` map onto strings the elements of the type `num` issued respectively from the set of "small" integers, floating point numbers and big integers or rational numbers. They are used to define `num_of_string`.

```
#string_of_float #1.1;;
"#1.1" : string

#string_of_floating #1.1;;
"1.1" : string
```

The function `string_of_num` is defined in CAML as a simple pattern matching on the values of type `num`:

```
let string_of_num = function
    Int i -> string_of_integer i
  | Float f -> string_of_floating f
  | Big_num b -> string_of_big_num b
;;
```

The two functions `int_of_string` and `float_of_string` are not provided but are easy to write:

```
#let int_of_string = function
#     "" -> failwith"int_of_string"
#   | s ->
#      if ascii_code s = ascii_code "#"
```

```
#       then
#        (match num_of_string (sub_string s 1 (length_string s))
#         with Int i -> i | _ -> failwith"int_of_string")
#       else failwith"int_of_string"
#and float_of_string = function
#     "" -> failwith"float_of_string"
#    | s ->
#      if ascii_code s = ascii_code "#"
#       then
#        (match num_of_string (sub_string s 1 (length_string s))
#         with Float f -> f | _ -> failwith"float_of_string")
#       else failwith"float_of_string"
#;;
Value int_of_string = <fun> : (string -> int)
Value float_of_string = <fun> : (string -> float)
```

5.4.12 Random functions

♣ `random : (int -> int)`
`init_random : (int -> int)`

There is a pseudo random number generator in the CAML system. It is based on the linear congruence method given by D.E. Knuth ("The Art of Computer Programming"). It provides two functions : **init_random** and **random**. The first one resets the generator to the given value, the second one returns a random value in the range between 0 and its argument.

Both use argument of type **int** which must be greater than or equal to 0 for **init_random** and strictly greater than 0 for **random**.

```
#print_int(random #3);print_int(random #3);print_int(random #3);;
#0#0#2() : unit
```

Chapter 6

General purpose functions

We describe some commonly used functions applicable to pairs, lists, and other CAML values. All are definable in CAML. For each function we give, in addition to its type, a CAML declaration equivalent to that defining it in the current system. In fact, this part of the documentation is automatically produced from the prelude of the system, in order to always be up-to-date.

6.1 Operations on pairs

6.1.1 Infix identifiers not bound to functions

These are essentially syntactical matters.

Pair construction

`prefix ,` is obviously useless since it should be a special case of identity:

```
let prefix , (x,y) = (x,y);;
```

A curried (see next section) function to build pairs is provided:

♣

```
pair : ('a -> 'b -> 'a * 'b)
```

This is just the curried form of ",":

```
let pair x y = x,y
;;
```

6.1.2 Destructors for pairs

♣
```
fst : ('a * 'b -> 'a)
snd : ('a * 'b -> 'b)
```

`fst` and `snd` are polymorphic predefined operators to access the components of a pair. They may be defined in CAML with

```
let fst (x,_) = x
and snd (_,y) = y;;
```

6.2 General combinators

6.2.1 Curryfication

A very important and useful feature of CAML is the possibility to "curry" a function. The idea is mainly to allow the definition of functions with several arguments, which can receive one argument at a time. For example consider the function `add` defined by:

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

This function is not exactly `prefix +`, since `prefix +` has only one argument (which is actually a pair) whereas `add` has two arguments. Thus the type of `add` is `num -> num -> num`, i.e. `add` has to receive a number, and then another one before it returns their sum. We say that add is the "curried" form of `prefix +`, since we have broken the pair, which is the single argument of `prefix +`, into two components which are the two arguments of `add`. It is a first way to understand the "currying" mechanism.

Another way is to get a deeper understanding of `add`'s type: `num -> num -> num` is nothing but `num -> (num -> num)`. That is to say that `add` is a function of high order: given a number, add leads to another function (the type of which is `num -> num`). It could seem very hard to understand at first glance, but in fact it is something that we commonly do while reasoning: Nobody should be surprised if you say " add 2 to your last result ", though this really means: Use addition with the first argument fixed to 2, and apply this function to your last result, i.e. use the curried version of addition with first argument 2 and apply it to your last result. In CAML it would be:

```
(add 2) (your_last_result);;
```

There are many ways to understand currying, and in the case of `add`, one can think of `add` as a "generic" function, to get the family of functions (with type `num -> num`) which add a constant value. For example we can define the function `add_2`, which increases its argument by 2, with the following:

```
let add_2 = add 2;;
```

and if you need the function which increases its argument twofold:

```
let double = mult 2
 where mult x y = x * y;;
```

This process is so useful that a functional "curry" is provided, which maps a function to its curried form. Uncurry reverts this process.

♣ `curry : (('a * 'b -> 'c) -> 'a -> 'b -> 'c)`
`uncurry : (('a -> 'b -> 'c) -> 'a * 'b -> 'c)`

```
let curry f x y = f (x,y) and uncurry f (x,y) = f x y
;;

#curry prefix + 2 3;;
5 : num
```

In the same way we could have defined the function `double` as:

```
let double = mult 2
 where mult = curry ( prefix * );;
```

6.2.2 Composition

♣ `prefix o : (('a -> 'b) * ('c -> 'a) -> 'c -> 'b)`

`prefix o (f,g)`, or equivalently `f o g` , returns the functional composition of `f` and g.

```
let f o g = fun x -> f (g x)
;;
```

Example:

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

#let double x = 2*x;;
Value double = <fun> : (num -> num)

#let quadr_square = square o double
#and double_square = double o square;;
```

```
Value quadr_square = <fun> : (num -> num)
Value double_square = <fun> : (num -> num)

#double_square 3;;
18 : num

#quadr_square 3;;
36 : num
```

♣
```
pairing : (('a -> 'b) * ('c -> 'd) -> 'a * 'c -> 'b * 'd)
distr_pair : (('a -> 'b) -> 'a * 'a -> 'b * 'b)
tee : (('a -> 'b) * ('a -> 'c) -> 'a -> 'b * 'c)
```

- `pairing (f,g)` combines `f` and `g` to yield the function on pairs determined by applying `f` and `g` to the corresponding components of the pairs.
- `distr_pair f` applies the function `f` to each component of a pair and returns the pair of results. Notice that the typechecking of CAML forces the pair argument of `distr_pair` to be homogeneous (due to the application of `f` to its components).
- `tee (f,g)` applies the functions `f` and `g` to a common argument and returns the pair of results.

```
let pairing (f,g) (x,y) = f x,g y
and distr_pair f (x,y) = f x,f y and tee (f,g) x = f x,g x
;;
```

6.2.3 Combinators, as in Curry and Feys

These are commonly used functional to build new functions from already existing functional values.

♣
```
I : ('a -> 'a)
K : ('a -> 'b -> 'a)
CK : ('a -> 'b -> 'b)
C : (('a -> 'b -> 'c) -> 'b -> 'a -> 'c)
W : (('a -> 'a -> 'b) -> 'a -> 'b)
B : (('a -> 'b) -> ('c -> 'a) -> 'c -> 'b)
S : (('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c)
```

The predefined combinators are:

- the polymorphic identity function `I`.

- the "cancellators" K and CK which forget their second (resp. first) argument.
- the "permutator" C which exchanges the arguments of a curried function.
- the "duplicator": W f x provides the function f with two copies of x.
- B is just the curried form of the usual functional composition o.
- S is the classical combinator from combinatory logic, which distributes its third argument to each of its functional arguments.

I is a primitive operation whose definition is equivalent to:

```
let I x = x;;

let K x y = x
and CK x y = y
and C f x y = f y x
and W f x = f x x
and B f g x = f (g x)
and S f g x = f x (g x)
;;
```

♣ `prefix Co : (('a -> 'b -> 'c) * ('d -> 'a) -> 'b -> 'd -> 'c)`

Permutation-composition, named so because `(f Co g) = C (f o g)`.

```
let f Co g = fun x y -> f (g y) x
;;
```

Note that a so-called "apply" operator with the following semantics:

```
#let apply f x = f (x);;
Value apply = <fun> : (('a -> 'b) -> 'a -> 'b)
```

is useless, since this apply is a special case of I.

6.2.4 Tacticals

Tacticals are functions that may be applied to other functions to sequence them or try one after the other, ... This terminolgy is issued from automatic theorem proving, where tacticals are used to build strategies to find the proofs.

♣ `can : (('a -> 'b) -> 'a -> bool)`

Test for failure: `can f x` applies `f` to `x`, returning true if evaluation succeeds and false otherwise:

```
let can f x = f x;true ? false
;;
```

♣
```
iterate : (('a -> 'a) -> num -> 'a -> 'a)
repeat : (num -> ('a -> 'b) -> 'a -> unit)
prim_rec : (('a -> num -> 'a) -> 'a -> num -> 'a)
```

- `iterate f` performs self composition of `f` a given number of times. It satisfies: `iterate f n x = (f (f (f ... (f x) ... )))` which is just the expanded form of the relation `iterate f n = (f o f o f ... o f)` with `n` occurrences of `f`.
- `repeat n f x` evaluates sequentially `n` times the application of the function `f` to the argument `x` and then returns `()`: it is mainly used for side-effects when `f` is a "command".
- `prim_rec f init n` evaluates the nth application of `f`, starting from `init`.

```
#(* The factorial function *)
#let fact = prim_rec mult 1;;
Value fact = <fun> : (num -> num)

#(* Another way to define the primitive range *)
#let range = prim_rec (C cons) [];;
Value range = <fun> : (num -> num list)

#let letters = prim_rec (fun s n -> (ascii (n+64))^s) "" 26;;
Value letters = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" : string
```

```
let iterate f = iterate_f
 where rec iterate_f n x =
  if n <= 0 then x else iterate_f (pred n) (f x)
;;

let repeat n action arg = repeat_action n
 where rec repeat_action n =
  if n <= 0 then () else (action arg;repeat_action (pred n))
;;

let prim_rec f = loop
 where rec loop init = function
     0 -> init | n -> loop (f init n) (n-1)
;;
```

Example:

```
#let hello () = message "hello world!";;
Value hello = <fun> : (unit -> unit)

#repeat 3 hello ();;
hello world!
hello world!
hello world!
() : unit
```

If we wanted to define the LCF tacticals THEN and ORELSE in CAML, we could proceed as follows. We recall that:

- `(f THEN g) x`, applies `f` to `x` and then returns `g(x)`.
- `(f ORELSE g) x`, applies `f` to `x` and if it fails (raising exception `failure`) then returns `g(x)`.

```
##infix THEN;;
Ident THEN is now parsed as an infix
Directive () : unit

#let a THEN b = function x ->  (a x; () ? ()); b x;;
Value prefix THEN = <fun> :
      (('a -> 'b) * ('a -> 'c) -> 'a -> 'c)

##infix ORELSE;;
Ident ORELSE is now parsed as an infix
Directive () : unit

#let a ORELSE b = function x -> a x ? b x;;
Value prefix ORELSE = <fun> :
      (('a -> 'b) * ('a -> 'b) -> 'a -> 'b)
```

6.2.5 Fixed Point Combinator

♣
```
Y : ((('a -> 'b) -> 'a -> 'b) -> 'a -> 'b)
fix_point : (('a -> 'a) -> 'a -> 'a)
```

- `Y` is a classical fixed point combinator which returns the function which is the fixed point of a functional.
- `fix_point f x` iterates the function `f` starting from the value `x`, to find a fix point for `f` (that is a value `f_p` such that `f f_p = f_p`).

```
let Y (f:('a -> 'b) -> 'a -> 'b) = g where rec g x = f g x
;;

let rec fix_point f x = if y = x then x else fix_point f y
 where y = f x
;;
```

6.2.6 Sharing transducers

♣ `share : (('a -> 'a) -> 'a -> 'a)`

Assuming that a function fails for identity (raising the special exception `Identity`), `share f` permits maximum sharing between the result of `f` and its argument.

```
exception Identity of unit
;;

let share f x = try f x with Identity -> x
;;

#let rec share_append (l1,l2) = share append_rec l1
# where rec append_rec = function
#  [] -> if l2<>[] then l2 else raise Identity
#| x1::l1 -> x1 :: append_rec l1;;
Value share_append = <fun> : ('a list * 'a list -> 'a list)

#let l = [1;2;3] in
#eq (l, share_append (l,[]));;
true : bool

#let l = [1;2;3] in
#eq (l, share_append ([],l));;
true : bool
```

♣ `pair_share : (('a -> 'a) -> 'a * 'a -> 'a * 'a)`
`share_pair_share : (('a -> 'a) -> 'a * 'a -> 'a * 'a)`

- If `f` is a function designed for sharing (i.e. it raises exception `Identity` in case of identity between its argument and its result), `pair_share f` extends this property to pairs. `pair_share` raises `Identity` when the argument pair can be shared.

- share_pair_share f is equivalent to share (pair_share f), thus it shares its argument pair as well.

```
let pair_share f (x,y) =
 try f x,share f y with Identity -> x,f y
;;


let share_pair_share f (x,y as pair) =
 try f x,share f y
 with Identity -> try x,f y with Identity -> pair
;;

#(* Share Identity (che're but not expensive):
#   instead of returning its argument its raises Identity *)
#let Sharing_Identity x = raise Identity;;
Value Sharing_Identity = <fun> : ('a -> 'b)

#(* Let us define some pair *)
#let a_pair = ("foo","bar");;
Value a_pair = ("foo","bar") : (string * string)

#(* If we distribute Sharing_Identity on the elements of a pair,
#   the pair itself is shared *)
#eq(a_pair, share_pair_share Sharing_Identity a_pair);;
true : bool

#(* But if we distribute the usual identity function I
#   on the elements of a pair, the pair itself is not shared *)
#eq(a_pair, distr_pair I a_pair);;
false : bool

#(* Share_pair_share is really useful and can not be
#   simulated using share and distr_pair *)
#eq(a_pair, distr_pair (share Sharing_Identity) a_pair);;
false : bool

#(* If a sharing function is not encapsulated in a ''share''
#   call, then Identity may escape to the CAML toplevel
#   which reports a message *)
#eq(a_pair, pair_share Sharing_Identity a_pair);;

Exception Identity not trapped
```

6.3 List and set processing functions

6.3.1 List definition

♣
```
[] : 'a list
prefix :: : ('a * 'a list -> 'a list)
```

The type `list` of homogeneous sequences of elements, is a type with one type argument, and is defined *quite normally* as

```
type 't list = [] | prefix :: of 't * 't list
;;
```

It is not advisable to redefine it, since the parser of the CAML system "knows" the abbreviation `[e1;e2; ... ;en]` for `e1 :: e2 :: ... :: []`. Thus strange things may happen if you redefine this "almost built in" type.

6.3.2 List primitives

♣
```
cons : ('a -> 'a list -> 'a list)
uncons : ('a list -> 'a * 'a list)
```

`cons` is the curried form of `prefix ::` and `uncons` is the converse operation: it maps a non-empty list (which a pair of an element and a list) onto the pair of its head and its tail.

```
let uncons = function
    prefix :: p -> p | _ -> failwith "uncons"
;;

let cons x y = x::y
;;
```

♣
```
hd : ('a list -> 'a)
tl : ('a list -> 'a list)
null : ('a list -> bool)
singleton : ('a -> 'a list)
append : ('a list -> 'a list -> 'a list)
prefix @ : ('a list * 'a list -> 'a list)
```

- hd returns the first element of a list and tl the rest of the list. These two functions fail (with respective failure string "tl" and "hd") when called with the empty list.
- null tests whether a list is empty or not.
- singleton builds a list whose unique element is the argument of singleton.
- l1 @ l2 catenates the two lists l1 and l2, and append is its curried version.

```
let null = function [] -> true | _ -> false
and hd = function [] -> failwith "hd" | x::l -> x
and tl = function [] -> failwith "tl" | x::l -> l
and singleton x = [x]
;;
```

♣ `length : ('a list -> num)`

length l returns the length of the list l.

```
let length l = length_l (0,l)
 where rec length_l = function
     n,[] -> n | n,_::l -> length_l (succ n,l)
;;
```

♣ `mem : ('a -> 'a list -> bool)`
`mem_list_list : ('a -> 'a list list -> bool)`

- mem x l returns true if some element of l is equal to x, otherwise false.
- mem_list_list x ll tests if x is a member of one of the lists of the list of lists ll.

```
let mem x = mem_x
 where rec mem_x = function
     [] -> false | y::l -> x = y or mem_x l
;;
```

mem_list_list is equivalent to:

```
(* Membership of a list of lists *)
(*|
Value mem_list_list : ('a -> 'a list list -> bool)
 CAML_system{mem}
|*)
let rec mem_list_list s = mem_rec where rec mem_rec =
 function [] -> false
        | l1::L -> (mem s l1) or (mem_rec L)
;;
```

6.3.3 Indexing

♣

```
nth : ('a list -> num -> 'a)
item : ('a list -> num -> 'a)
index : ('a -> 'a list -> num)
```

- The first function returns the *n*th element of a list.
- `item l n` returns the element of the list `l` having the position `n` (starting from 0).
- Finally `index` gives the position of the first occurrence of a given value in a list.

```
let index x l = index_x (1,l)
 where rec index_x = function
     n,y::l -> if x = y then n else index_x (succ n,l)
   | _ -> failwith "index"
;;

let nth l n =
 if n <= 0 then failwith "nth" else (nth_op (l,n)
  where rec nth_op = function
      hd::_,1 -> hd
    | _::tl,n -> nth_op (tl,pred n)
    | [],_ -> failwith "nth")
;;

let item l n = nth l (succ n)
;;
```

6.3.4 Reversing lists

♣ `rev_append : ('a list -> 'a list -> 'a list)`
`rev : ('a list -> 'a list)`

- `rev l` reverses the list `l`.
- `rev_append l1 l2` appends `l2` to the reverse of the list `l1`.
  Thus `rev_append l1 l2 == rev l1 @ l2`.

```
#rev_append [5;6;7] [4;3;2;1];;
[7; 6; 5; 4; 3; 2; 1] : num list

let rec rev_append =
 fun [] -> I | (x::l') l -> rev_append l' (x::l)
;;

let rev l = rev_append l []
;;
```

6.3.5 Finding items in lists

The following functions search lists for elements with various properties; those returning elements fail if no such element is found, those returning boolean results never fail.

♣ `find : (('a -> bool) -> 'a list -> 'a)`
`try_find : (('a -> 'b) -> 'a list -> 'b)`

- `find p l` returns the first element of `l` which satisfies the predicate `p`.
- `try_find f l` returns the result of applying `f` to the first member of `l` for which the application of `f` succeeds.

```
let find p = find_p
 where rec find_p = function
     x::l -> if p x then x else find_p l
   | _ -> failwith "find"
;;

let try_find f = try_find_f
 where rec try_find_f = function
     [] -> failwith "try_find" | h::t -> f h ? try_find_f t
;;
```

6.3.6 List functionals

♣
```
map : (('a -> 'b) -> 'a list -> 'b list)
end_map : (('a -> 'b) -> 'b list -> 'a list -> 'b list)
map_i : ((num -> 'a -> 'b) -> num -> 'a list -> 'b list)
end_map_i :
((num -> 'a -> 'b) -> 'b list -> num -> 'a list -> 'b list)
map_share : (('a -> 'a) -> 'a list -> 'a list)
share_map_share : (('a -> 'a) -> 'a list -> 'a list)
map2 : (('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list)
```

- `map f l` returns the list obtained by applying `f` to the elements of `l` in turn:

  ```
  map f [a1;...;an] = [f a1;...;f an]
  ```

- `end_map f default l` returns the list obtained by applying `f` to the elements of `l` in turn, but in front of the list `default` (i.e. `map f` = `end_map f []` and `end_map f default l = (map f l) @ default`).

- `map_i f start l` returns as well the list obtained by applying `f` to the elements of `l` in turn, but it provides to `f` the rank of the item processed by `f`:

  ```
  map_i f start [e1; e2; ... ; ei; ... ; en] =
                [f start e1; f (start+1) e2; ... ;
                 f (start+i) ei; ... ;
                 f (start+n) en]
  ```

  Example:

  ```
  #map_i pair 0 ["a";"b";"c"];;
  [(0,"a"); (1,"b"); (2,"c")] : (num * string) list

  #map_i
  # (fun i e ->
  #   print_string"The ";print_num i;
  #   print_string"th element is: ";message e)
  # 0
  # ["a";"b";"c";"d"];;
  The 0th element is: a
  The 1th element is: b
  The 2th element is: c
  The 3th element is: d
  [(); (); (); ()] : unit list
  ```

- `end_map_i` is the analog of `end_map` for `map_i`.
- `map_share` extends the sharing property to lists. `map_share` raises `Identity` when the argument list can be shared. `share_map_share f` is equivalent to `share (map_share f)`, thus :
- `share_map_share` tries to share its result with its list argument.
- `map2 f l1 l2` maps a curried function with two arguments on two lists. Arguments of `f` are picked up in `l1` and `l2` in turn (thus `l1` and `l2` must have the same length).

```
let map f = map_f
 where rec map_f = fun [] -> [] | (a::l)  ->  (f a::map_f l);;

let end_map f default = map_f
 where rec map_f = function
     [] -> default | a::l -> f a::map_f l
;;

let fun_pair f =
 let rec map_share_f = function
     [] -> raise Identity
   | x::r ->
       try f x::share_map_share_f r
       with Identity -> x::map_share_f r
 and share_map_share_f = share map_share_f in
  map_share_f,share_map_share_f in
let map_share = fst o fun_pair
and share_map_share = snd o fun_pair
;;

let map2 f l1 l2 = map2_f (l1,l2)
 where rec map2_f = function
     [],[] -> []
   | h1::t1,h2::t2 -> f h1 h2::map2_f (t1,t2)
   | _ -> failwith "map2"
;;

let map_i f = map_i_rec
 where rec map_i_rec i = function
     [] -> [] | x::l -> f i x::map_i_rec (i+1) l
;;

let end_map_i f default = map_i_rec
 where rec map_i_rec i = function
     [] -> default | x::l -> f i x::map_i_rec (i+1) l
;;
```

♣ `map_uncons : (('a -> 'b) -> 'a * 'a list -> 'b * 'b list)`
`do_list_uncons : (('a -> 'b) -> 'a * 'a list -> unit)`

These functions are designed to operate on non-empty lists (that is a pair of an element and a list), and for such arguments they correspond respectively to `map` and `do_list`.

```
let map_uncons f (x,L) = f x,map f L
and do_list_uncons f (x,L) = f x;do_list f L
;;
```

♣ `it_list : (('a -> 'b -> 'a) -> 'a -> 'b list -> 'a)`

For `f`, `b`, and `l` arguments, this function performs iterated compositions of the curried function `f`, using elements of `l` as second arguments of `f`; `b` is used for the first argument of the first call of `f`, and thereafter the result of each call is passed as the first argument of the next call. `it_list` is characterised by:

```
it_list f a [b1;...;bn] = (f (...(f a b1)...) bn)
```

```
let it_list f = it_list_f
 where rec it_list_f a = function
     [] -> a | b::l -> it_list_f (f a b) l
;;
```

♣ `sigma : (num list -> num)`
`pi : (num list -> num)`

In an example using `it_list`, these two functions return the sum (resp. the product) of all the elements of a list:

```
let sigma = it_list add 0 and pi = it_list mult 1
;;
```

♣ `list_it : (('a -> 'b -> 'b) -> 'a list -> 'b -> 'b)`

For `f`, `l`, and `b` arguments, this function performs iterated compositions of the curried function `f`, using elements of `l` as first arguments of `f`; `b` is used for the second argument of the first call of `f`, thereafter the result of each call is passed as the second argument of the next call. `list_it` is characterised by:

```
list_it f [l1;l2;...;ln] b = f l1 (f l2 (...(f ln b)...))
                           = ((f l1) o (f l2) o...o (f ln)) b
```

Notice that `it_list` is tail recursive and thus more efficient than `list_it` ...

As an example we can define the "old" `rev_itlist` function, from ML V6.2, using:

```
#let rev_itlist f = C (list_it (C f));;
Value rev_itlist = <fun> :
      (('a -> 'b -> 'a) -> 'a -> 'b list -> 'a)

let list_it f l b = list_it_f l
 where rec list_it_f = function
     [] -> b | a::l -> f a (list_it_f l)
;;
```

♣ `end_it_list : (('a -> 'a -> 'a) -> 'a -> 'a list -> 'a)`

`end_it_list` is almost the same as `it_list` but for non empty lists it does not use an extra argument for the first call of `f`: the first element of `l` will be used instead. If the list is empty then `default` is returned. `end_it_list` is characterised by:

```
end_it_list f default [x1; ...; xn] =
 (f ... (f x1 x2) ... xn) (for n>1)
 x1                       (for n = 1)
 default                  (for n = 0)

let end_it_list f default = function
   [] -> default
 | b::l -> it_list f b l
;;
```

♣ `fold :`
`(('a -> 'b -> 'a * 'c) -> 'a -> 'b list -> 'a * 'c list)`
`fold_share :`
`(('a * 'b -> 'a * 'b) -> 'a * 'b list -> 'a * 'b list)`
`share_fold_share :`
`(('a * 'b -> 'a * 'b) -> 'a * 'b list -> 'a * 'b list)`

The functionnal `fold` is used to map a curried function on a list in a special way, since the function updates a "memory" (presumably an association list viewed as an environment) which is passed to each call in turn. Thus the result of `fold f init_memory l` is the pair `m,l'` where `m` is the new memory modified by all the successive calls to `f` and `l'` is the list of results.

```
 fold f a1 [b1; ... ;bn] =
        let a2,c1 = f a1 b1 in
          ...
        let a,cn = f an bn in
          a,[c1; ... ;cn]

let fold f = fold_f
 where rec fold_f a1 = function
     [] -> a1,[]
   | b1::bl ->
       let a2,c2 = f a1 b1 in
       let a,cl = fold_f a2 bl in a,c2::cl
;;
```

As an example if you want to map the **successor** function on a list of integers, while counting the number of elements of the list, you may use as environment the current index of the item processed by **successor** :

```
#fold (fun i n -> i+1,succ n) 0 [1;2;3];;
(3,[2; 3; 4]) : (num * num list)
```

In the same vein, you get a definition of the functional **map_i**, using **fold** :

```
#let map_i f start l =
#    snd (fold (fun i x -> (i+1,f i x)) start l);;
Value map_i = <fun> :
      ((num -> 'a -> 'b) -> num -> 'a list -> 'b list)

#map_i (fun i s -> s^(string_of_num i)) 1 ["a";"b";"c"];;
["a1"; "b2"; "c3"] : string list
```

An equivalent but more functional way to define **map_i** is:

```
#let map_i f start = snd o fold consf start
#    where consf i x = (i+1,f i x);;
Value map_i = <fun> :
      ((num -> 'a -> 'b) -> num -> 'a list -> 'b list)
```

Sharing versions of **fold** are also available:

```
let fold_share f =
 let rec fold_share_f = (function
     _,[] -> raise Identity
   | a1,b1::bl ->
       try let a2,c2 = f (a1,b1) in
           let a,cl = share_fold_share_f (a2,bl) in a,c2::cl
       with Identity ->
```

```
              let a,c1 = fold_share_f (a1,b1) in a,b1::c1)

 and share_fold_share_f = share fold_share_f in
  fold_share_f
;;

let share_fold_share f =
 let rec fold_share_f = (function
     _,[] -> raise Identity
   | a1,b1::bl ->
       try let a2,c2 = f (a1,b1) in
           let a,c1 = share_fold_share_f (a2,bl) in a,c2::c1
       with Identity ->
              let a,c1 = fold_share_f (a1,bl) in a,b1::c1)

 and share_fold_share_f = share fold_share_f in
  share_fold_share_f
;;
```

♣

```
do_list : (('a -> 'b) -> 'a list -> unit)
do_list_i : ((num -> 'a -> 'b) -> num -> 'a list -> unit)
do_list2 : (('a -> 'b -> 'c) -> 'a list -> 'b list -> unit)
```

- `do_list f l` applies `f` to every element of `l` in turn and then returns `()`: no list of results is built.
- `do_list_i f start l` is a specialised version of `do_list` which provides to its functional argument `f` the rank of the element to which `f` is applied (starting from `start`).
- `do_list2 f l1 l2` works in the same way for a curried function with two arguments which are taken from `l1` and `l2` in parallel.

```
let do_list f = do_list_f
 where rec do_list_f = function
     [] -> () | x::l -> f x;do_list_f l
;;

let do_list_i f = do_list_f
 where rec do_list_f i = function
     [] -> () | x::l -> f i x;do_list_f (succ i) l
;;
```

```
let do_list2 f l1 l2 = do_list2_f (l1,l2)
 where rec do_list2_f = function
     [],[] -> ()
   | h1::t1,h2::t2 -> f h1 h2;do_list2_f (t1,t2)
   | _ -> failwith "do_list2"
;;
```

6.3.7 Various list operators

List concatenation, either infix or prelix form:

♣
```
append : ('a list -> 'a list -> 'a list)
prefix @ : ('a list * 'a list -> 'a list)

let append l1 l2 = if l2 = [] then l1 else list_it cons l1 l2
;;

let l1 @ l2 = if l2 = [] then l1 else append l1 l2
;;
```

Notice that append may be defined as:

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

but this should copy the first argument of append even if the second is the empty list.

♣
```
partition : (('a -> bool) -> 'a list -> 'a list * 'a list)
replicate : (num -> 'a -> 'a list)
flat : ('a list list -> 'a list)
flat_map : (('a -> 'b list) -> 'a list -> 'b list)
```

- `partition p l` returns a pair of lists; the first element of this pair is the list of all the elements of l which satisfy the predicate p, and the second list of the pair is the rest of l.
- `replicate x n` builds a list of length n, all elements of which are equal to x.
- `flat ll` flattens a list of lists.
- `flat_map f l` maps the function `f` on the list `l`, and catenates the results which must therefore be lists.

```
let partition p l = list_it fork l ([],[])
 where fork a (pos,neg) =
  if p a then a::pos,neg else pos,a::neg
;;

let replicate n =
 if n < 0 then failwith "replicate"
  else fun x -> replicate_aux ([],n)
       where rec replicate_aux = function
           l,0 -> l | l,n -> replicate_aux (x::l,pred n)
;;

let flat ll = list_it append ll []
;;

let flat_map f = flat_map_f
 where rec flat_map_f = function
     [] -> [] | x::l -> f x@flat_map_f l
;;
```

♣ `first_n : (num -> 'a list -> 'a list)`
`last_n : (num -> 'a list -> 'a list)`

- `first_n n l` returns the first `n` elements of the list `l`.
- `last_n n l` returns the last `n` elements of the list `l`.

```
let rec first_n =
 fun 0 -> K []
   | n -> function
          [] -> failwith "first_n"
        | x::l -> x::first_n (pred n) l
and last_n n l =
 iterate tl (length l-n) l ? failwith "last_n"
;;
```

6.3.8 Set operations

Set operators

♣ `exists : (('a -> bool) -> 'a list -> bool)`
`for_all : (('a -> bool) -> 'a list -> bool)`

- `exists p l` applies `p` to the elements of `l` in order until one is found which satisfies `p`, or until the list is exhausted, returning true or false accordingly. More exactly `exists p [x1; ... ;xn]` is false if `p(xi)` is false for every `i`. Otherwise, it returns `p(xi)` for `i` minimum such that `p(xi)` is not false.

- `for_all p l` applies `p` to every element of `l` to see if they all verify `p`. More exactly `for_all p [x1; ... ;xn]` is true if `p(xi)` is true for every `i`. Otherwise, it returns `p(xi)` for `i` minimum such that `p(xi)` is not true.

```
let for_all p = for_all_p
 where rec for_all_p = function
     [] -> true | a::l -> p a & for_all_p l
;;

let exists p = exists_p
 where rec exists_p = function
     [] -> false | a::l -> p a or exists_p l
;;
```

As an example, one can define `mem_list_list` using:

```
let mem_list_list x = exists (mem x)
;;
```

♣
```
for_all2 : (('a -> 'b -> bool) -> 'a list -> 'b list -> bool)
exists2 : (('a -> 'b -> bool) -> 'a list -> 'b list -> bool)
for_all_pair : (('a -> 'a -> bool) -> 'a list -> bool)
exists_pair : (('a -> 'a -> bool) -> 'a list -> bool)
```

- `for_all2` and `exists2` are the analogues of `exists` and `for_all` for a curried predicate applied in turn on the corresponding elements of two lists examined in parallel.

- `exists_pair` sees if there exists a pair of elements in a list which verifies a given predicate, and `for_all_pair` checks if all the pairs built up from elements of the list verify the predicate.

The following characterization may help:

```
exists_pair p [x1; ... ; xn] == true
iff there exists a pair xi,xj st. i<j and p xi xj

for_all_pair p [x1; ... ; xn] == true
iff for all pairs xi,xj st. i<j :  p xi xj
```

As an example we can define the function `is_set` which tests if a given finite list is in fact a finite set (that is, that there are no two equal elements in the list):

```
#let is_set = for_all_pair (curry (prefix <>));;
Value is_set = <fun> : ('a list -> bool)

let for_all2 rel = for_all2_rel
 where rec for_all2_rel =
  fun [] -> (function
            [] -> true | _ -> failwith "for_all2")
    | (x1::l1) -> function
            x2::l2 -> rel x1 x2 & for_all2_rel l1 l2
          | _ -> failwith "for_all2"
;;

let exists2 rel = exist2_rel
 where rec exist2_rel =
  fun [] -> (function
            [] -> false | _ -> failwith "exists2")
    | (x1::l1) -> function
            x2::l2 -> rel x1 x2 or exist2_rel l1 l2
          | _ -> failwith "exists2"
;;

let exists_pair p = exists_pair_p
 where rec exists_pair_p = function
     [] -> false | x::l -> exists (p x) l or exists_pair_p l
;;

let for_all_pair p = for_all_pair_p
 where rec for_all_pair_p = function
     [] -> true | x::l -> for_all (p x) l & for_all_pair_p l
;;
```

Filtering lists

♣
```
filter : (('a -> bool) -> 'a list -> 'a list)
filter_pos : (('a -> bool) -> 'a list -> 'a list)
filter_neg : (('a -> bool) -> 'a list -> 'a list)
filter_succeed : (('a -> 'b) -> 'a list -> 'a list)
```

- `filter p l` applies p to every element of `l`, returning the list of those which satisfy p.

- `filter_pos` is an alias of `filter`.
- `filter_neg` is the converse function which returns the list of all the elements of `l` which do *not* satisfy `p`.
- `filter_succeed f l` which returns the list of all the elements of `l` for which the call of `f` does not fail (i.e. does not raise the exception `failure`).

```
#filter_neg (fun x -> x = 1) [1;2;3;4];;
[2; 3; 4] : num list

#filter_pos (fun x -> x = 1) [1;2;3;4];;
[1] : num list

let filter_pos p = (share filter_aux
 where rec filter_aux = function
     [] -> raise Identity
   | x::l ->
       if p x then x::filter_aux l else share filter_aux l)

and filter_neg p = (share filter_aux
 where rec filter_aux = function
     [] -> raise Identity
   | x::l ->
       if p x then share filter_aux l else x::filter_aux l)

and filter_succeed p = (share filter_aux
 where rec filter_aux = function
     [] -> raise Identity
   | x::l -> p x;x::filter_aux l ? share filter_aux l)
;;

let filter = filter_pos
;;
```

♣ `map_succeed : (('a -> 'b) -> 'a list -> 'b list)`
`do_list_succeed : (('a -> 'b) -> 'a list -> unit)`

- `map_succeed f l` applies f to every element of `l`, returning the list of results for those elements of `l` for which the application of `f` succeeds.
- `do_list_succeed` is the corresponding functional for functions used only for side-effects.

The function `map_succeed` could be defined simply as:

```
let map_succeed f = map f (filter_succeed f);;
```

For obvious efficiency reasons it is defined instead with:

```
let map_succeed f = map_f
 where rec map_f = function
     [] -> [] | h::t -> f h::map_f t ? map_f t
;;

let do_list_succeed f = do_list_f
 where rec do_list_f = function
     [] -> () | x::l -> f x;do_list_f l ? do_list_f l
;;
```

Notice that the local functions `map_f` or `do_list_f` never raise the exception `failure`.

6.3.9 Mapping "in accumulators"

These functions are used to map functions on lists, accumulating the results in a list reference.

♣
```
do_list_replace :
(('a -> 'b) -> 'b list ref -> 'a list -> unit)
do_list_i_replace :
((num -> 'a -> 'b) -> 'b list ref -> num -> 'a list -> unit)
do_list_succeed_replace :
(('a -> 'b) -> 'b list ref -> 'a list -> unit)
```

- `do_list_replace f accu l` applies f to every element of `l`, adding the results in the list reference `accu`. Then it returns `()`, in the tradition of the `do_list` family.
- `do_list_succeed_replace` and `do_list_i_replace` are the functionals corresponding to `do_list_i` and `do_list_succeed`.

```
let do_list_replace f accu = do_list_f
 where rec do_list_f = function
     [] -> () | x::l -> accu := f x::!accu;do_list_f l
;;

let do_list_i_replace f accu = do_list_f
 where rec do_list_f i = function
     [] -> ()
   | x::l -> accu := f i x::!accu;do_list_f (succ i) l
;;
```

```
let do_list_succeed_replace f accu = do_list_f
 where rec do_list_f = function
     [] -> ()
   | x::l -> accu := f x::!accu;do_list_f l ? do_list_f l
;;
```

Sets

♣
```
add_set : ('a -> 'a list -> 'a list)
make_set : ('a list -> 'a list)
intersect : ('a list -> 'a list -> 'a list)
subtract : ('a list -> 'a list -> 'a list)
union : ('a list -> 'a list -> 'a list)
distinct : ('a list -> bool)
```

The first function is the set constructor. The second function maps a list into a set, deleting duplicate elements, and the next three provide the obvious functions on sets represented as lists without repetitions (this may be checked using `distinct`).

```
let add_set x s = if mem x s then s else x::s
;;

let make_set l = share make_aux l
 where rec make_aux = function
     [] -> raise Identity
   | x::l ->
       if mem x l then share make_aux l else x::make_aux l
;;

let intersect l1 l2 = filter_pos (C mem l2) l1
and subtract l1 l2 = filter_neg (C mem l2) l1
;;

let union l1 l2 = urec l1
 where rec urec = function
     [] -> l2
   | a::l -> if mem a l2 then urec l else a::urec l
;;

let rec distinct = function
    h::t -> not mem h t & distinct t | _ -> true
;;
```

6.3.10 Association lists

An "association list" is just a list of pairs. It is used to record some values with some selectors to access the values. To some extent it is just a way to represent the graph of a finite function.

♣
```
equal_fst : ('a -> 'a * 'b -> bool)
equal_snd : ('a -> 'b * 'a -> bool)
assoc : ('a -> ('a * 'b) list -> 'b)
rev_assoc : ('a -> ('b * 'a) list -> 'b)
mem_assoc : ('a -> ('a * 'b) list -> bool)
except_assoc : ('a -> ('a * 'b) list -> ('a * 'b) list)
except_rev_assoc : ('a -> ('b * 'a) list -> ('b * 'a) list)
pair_assoc : ('a -> ('a * 'b) list -> 'a * 'b)
pair_rev_assoc : ('a -> ('b * 'a) list -> 'b * 'a)
```

- `equal_fst x p` (resp. `equal_snd`) returns true if the first (resp. second) element of the pair `p` is equal to `x`.
- `assoc x l` searches the list `l` of pairs for one whose first component is equal to `x`, returning the second component of the first pair found.
- `rev_assoc y l` similarly searches for a pair whose second component is equal to `y`, and returns its first component.
- `mem_assoc x l` returns true or false according to the existence in the list of pairs `l` of a pair whose first component is equal to `x`.
- `except_assoc x l` returns the list `l` without the first pair whose first component is `x` (resp. second component for `except_rev_assoc`).
- `pair_assoc x l` searches the list `l` of pairs for one whose first component is equal to `x`, returning the pair found as result.
- `pair_rev_assoc x l` searches the list `l` of pairs for one whose second component is equal to `x`, returning the pair found as result.

```
let equal_fst u (x,y) = u = x and equal_snd u (x,y) = u = y
;;

let pair_assoc x = find (equal_fst x)
and pair_rev_assoc x = find (equal_snd x)
;;

let assoc x = snd o (pair_assoc x)
and rev_assoc x = fst o (pair_rev_assoc x);;
```

```
let except_assoc e = except_e where rec except_e = function
    [] -> []
 | (x,_ as y)::l -> if x=e then l else y::except_e l
;;
let except_rev_assoc e = except_e where rec except_e = function
    [] -> []
 | (_,x as y)::l -> if x=e then l else y::except_e l
;;

let mem_assoc = exists o equal_fst
;;
```

6.3.11 Adding and removing items

♣
```
except : ('a -> 'a list -> 'a list)
chop_list : (num -> 'a list -> 'a list * 'a list)
change : ('a list -> num -> ('a -> 'a) -> 'a list)
update : ('a list -> num -> 'a -> 'a list)
```

- except x l returns the list l without *the first* occurrence of x.
- chop_list n l divides l into a pair of lists, the first element of the pair is built from the first n elements of l and the second with the remainder of l. chop-list is characterised by:

```
chop_list n [e1; ...; en; e[n+1]; ... ; e[n+m]] =
            [e1; ...; en] , [e[n+1]; ...; e[n+m]]
```

- change l n f returns the list l' built with the elements of l except for the nth, a, which is replaced by its image by f, f a.
- update l n x returns l' such that l' is l except the nth element of l which becomes x.

```
let except e = except_e
 where rec except_e = function
    [] -> []
   | elem::l -> if e = elem then l else elem::except_e l
;;

let change l n f = change_f l n
 where rec change_f =
  fun [] -> failwith "change"
    | (h::t) n ->
        if n = 1 then f h::t else h::change_f t (pred n)
;;
```

```
let update l n x = change l n (K x)
;;

let chop_list = (fun n l -> chop_aux n ([],l))
 where rec chop_aux =
  fun 0 (l1,l2) -> rev l1,l2
    | n -> function
           _,[] -> failwith "chop_list"
         | l1,h::t -> chop_aux (pred n) (h::l1,t)
;;

#except 2 [1;2;3;1;2;3];;
[1; 3; 1; 2; 3] : num list

#chop_list 2 it;;
([1; 3],[1; 2; 3]) : (num list * num list)

#update [1;2;3;4] 2 3;;
[1; 3; 3; 4] : num list

#change it 2 pred;;
[1; 2; 3; 4] : num list
```

Handling the last element of lists

♣
```
last : ('a list -> 'a)
sep_last : ('a list -> 'a * 'a list)
add_last : ('a -> 'a list -> 'a list)
except_last : ('a list -> 'a list)
```

These are obvious functions to deal with the last element of a list.

```
let last = function
    [] -> failwith "last"
  | prefix :: pair -> last_aux pair
      where rec last_aux = function
          x,[] -> x | _,prefix :: pair -> last_aux pair
;;

let sep_last = function
    [] -> failwith "sep_last"
  | prefix :: pair -> sep_last_aux pair
      where rec sep_last_aux = function
```

```
          h,[] as pair -> pair
        | h,prefix :: pair ->
            let x,l = sep_last_aux pair in x,h::l
;;

let add_last x l = l@[x]
;;

let except_last = function
    [] -> failwith "except_last"
  | prefix :: pair -> except_last_aux pair
      where rec except_last_aux = function
          _,[] -> []
        | x,prefix :: pair -> x::except_last_aux pair
;;
```

Rotating lists

♣ `rotate_left : ('a list -> 'a list)`
`rotate_right : ('a list -> 'a list)`

These two functions work so that the first (resp. the last) element of a list becomes the last one (resp. the first one).

```
let rotate_left = function
    [] -> failwith "rotate_left" | a::l -> l@[a]
and rotate_right = function
    [] -> failwith "rotate_right"
  | l -> prefix :: (sep_last l)
;;
```

6.3.12 Sorting lists (quick sort)

♣ `select : (('a -> bool) -> 'a list -> 'a * 'a list)`
`sort_append :`
`(('a -> bool) -> ('a * 'a -> bool) -> 'a list -> 'a list ->`
`'a list)`
`sort : (('a * 'a -> bool) -> 'a list -> 'a list)`

- `select p l` returns a pair `(elem,l')`, where `elem` is the first element of `l` which satisfies the predicate `p`, and `l'` is the rest of the list `l`.

- `sort_append p order l l'` is designed mainly to implement the function `sort`. It appends the list `l'` to the list `l`, but `l` has been sorted by the binary predicate `order` (using the predicate `p` to select the way the list `l` is cut while sorting).

- `sort order l` sorts the list `l` according to the binary predicate `sort`. The algorithm implemented is "quick sort".

```
let select p = select_p
 where rec select_p = function
     [] -> failwith "select"
   | prefix :: (x,l as pair) ->
       if p x then pair
        else (let u,ll = select_p l in u,x::ll)
;;

let sort_append p le = sort_aux
 where rec sort_aux = function
     [] -> I
   | l ->
       let y,rest = select p l in
       let left,right = partition (fun x -> le (x,y)) rest in
        sort_aux left o cons y o sort_aux right
;;

let sort order list =
 sort_append (fun _ -> true) order list []
;;
```

6.3.13 Lists of pairs and pairs of lists

♣

```
split : (('a * 'b) list -> 'a list * 'b list)
combine : ('a list * 'b list -> ('a * 'b) list)
```

These two map a list of pairs into the corresponding pair of lists, and conversely (with `combine` failing if its argument lists are not of the same length).

```
let rec split = function
    (x1,x2)::l -> let l1,l2 = split l in x1::l1,x2::l2
  | [] -> [],[]
;;
```

```
let rec combine = function
    h1::t1,h2::t2 -> (h1,h2)::combine (t1,t2)
  | [],[] -> []
  | _ -> failwith "combine"
;;
```

♣

```
it_list2 :
(('a -> 'b * 'c -> 'a) -> 'a -> 'b list -> 'c list -> 'a)
list_it2 :
(('a * 'b -> 'c -> 'c) -> 'a list -> 'b list -> 'c -> 'c)
```

These are iterators corresponding to `it_list` and `list_it` for a curried function and two lists.

```
let it_list2 f init list1 list2 =
 it_list f init (combine (list1,list2) ? failwith "it_list2")
;;

let list_it2 f list1 list2 =
 list_it f (combine (list1,list2) ? failwith "list_it2")
;;
```

6.3.14 Lists of consecutive integers

♣

```
interval : (num -> num -> num list)
range : (num -> num list)
```

- `interval n m` returns the list of all integers in increasing order, from `n` to `m`.
- `range n` gives the first `n` integers.

```
let interval n m = interval_n ([],m)
 where rec interval_n (l,m) =
  if n > m then l else interval_n (m::l,pred m)
;;

let range = interval 1
;;
```

6.3.15 Testing identity of values in lists

♣
```
eq_fst : ('a -> 'a * 'b -> bool)
eq_snd : ('a -> 'b * 'a -> bool)
assq : ('a -> ('a * 'b) list -> 'b)
rev_assq : ('a -> ('b * 'a) list -> 'b)
except_assq : ('a -> ('a * 'b) list -> ('a * 'b) list)
pair_assq : ('a -> ('a * 'b) list -> 'a * 'b)
pair_rev_assq : ('a -> ('b * 'a) list -> 'b * 'a)
mem_assq : ('a -> ('a * 'b) list -> bool)
memq : ('a -> 'a list -> bool)
exceptq : ('a -> 'a list -> 'a list)
add_setq : ('a -> 'a list -> 'a list)
make_setq : ('a list -> 'a list)
intersectq : ('a list -> 'a list -> 'a list)
subtractq : ('a list -> 'a list -> 'a list)
unionq : ('a list -> 'a list -> 'a list)
```

These are functions corresponding to the "assoc" and "mem" families, using the predicate eq instead of = to test equality.

```
let eq_fst x (t,_) = eq (t,x) and eq_snd x (_,t) = eq (t,x)
;;

let pair_assq x = find (fun (t,_) -> eq (t,x))
and pair_rev_assq x = find (fun (_,t) -> eq (t,x))
;;

let mem_assq = exists o eq_fst
;;

let assq x = snd o pair_assq x

and rev_assq x = fst o pair_rev_assq x

and memq x = (memq_x
 where rec memq_x = function
     [] -> false | y::l -> eq (x,y) or memq_x l)
;;

let except_assq e = except_e
 where rec except_e = function
     [] -> []
   | (x,_ as y)::l -> if x == e then l else y::except_e l
;;
```

```
let add_setq x s = if memq x s then s else x::s
;;
```

It would be very easy to define the analogue of **make_set** for the **eq** function:

```
#let rec make_setq = function
#  [] -> []
#| (x::L) -> if memq x L then make_setq L else x::make_setq L;;
Value make_setq = <fun> : ('a list -> 'a list)
```

We choose instead a sharing version, for sake of efficiency:

```
let make_setq l = share make_aux l
 where rec make_aux = function
     [] -> raise Identity
   | x::l ->
       if memq x l then share make_aux l else x::make_aux l
;;

let exceptq e = exceptq_e
 where rec exceptq_e = function
     [] -> []
   | elem::l -> if eq (e,elem) then l else elem::exceptq_e l
;;

let intersectq l1 l2 = filter (C memq l2) l1
and subtractq l1 l2 = filter (neg (C memq l2)) l1
;;

let unionq l1 l2 = urec l1
 where rec urec = function
     [] -> l2
   | a::l -> if memq a l2 then urec l else a::urec l
;;
```

6.4 Segments

Segments are homogeneous sequences of CAML values, stored in consecutive memory cells for the sake of efficiency. The positions of items in a segment are counted starting from 0.

6.4.1 Creating segments

As for lists, elements of segments cannot be modified. You may either create functional segments (using the keyword **segment** preceding a list) or segments of data:

- `segment of l` creates a segment whose elements are those of the list `l`.
- `segment n of e` creates a segment of length n, whose elements are all equal to `e`.

An alternative way to create segments is to enumerate their elements using the concrete notation for segments:

```
#[<1;2;3>];;
[<1; 2; 3>] : num seg
```

6.4.2 Primitive operations on segments

♣ `seg_item : ('a seg * num -> 'a)`

- `seg_item (seg,n)` returns the item number `n` of the segment `seg`.

♣ `seg_length : ('a seg -> num)`
`is_null_seg : ('a seg -> bool)`

- `seg_length seg` returns the length of the segment `seg`.
- `is_null_seg` is bound to the expected predicate and allows to test if a segment has no items at all.

♣ `do_seg : (('a -> 'b) -> 'a seg -> unit)`
`map_seg : (('a -> 'b) -> 'a seg -> 'b list)`
`do_seg_i : ((num -> 'a -> 'b) -> 'a seg -> unit)`
`map_seg_i : ((num -> 'a -> 'b) -> 'a seg -> 'b list)`

- `do_seg f seg` is the analog of `do_list` for segments: it executes `f` on each element of the segment `seg` and returns `()`.
- `map_seg f seg` is the analog of `map` for segments: it returns the list of results of applying `f` on each element of the segment.
- `map_seg_i` and `do_seg_i` are slightly different versions of the primitive without `_i`: they provide the item rank in the segment to which the function is applied.

```
let do_seg f v = paired_do_seg (f,v)
;;
```

```
let do_seg_i f v = paired_do_seg_i (f,v)
;;

let map_seg f v =
 let l = pred (seg_length v) in
 maprec 0
 where rec maprec n =
  f (seg_item (v,n))::(if n = l then [] else maprec (succ n))
;;

let map_seg_i f v =
 let l = pred (seg_length v) in
 maprec 0
 where rec maprec n =
  f n (seg_item (v,n))::
  (if n = l then [] else maprec (succ n))
;;
```

♣ `seg_of_obj : (obj -> obj seg)`
`is_segment : (obj -> bool)`

- `is_segment obj` tests if its argument `obj` is the representation of a CAML segment.
- `seg_of_obj obj` given an `obj` which is the representation of a segment returns this segment as a segment of `objs`.

6.5 Vectors

Vectors are homogeneous sequences of CAML values, stored in consecutive memory cells. The positions of items in a vector are counted starting from 0.

6.5.1 Creating vectors

As for references, vectors cannot be polymorphic. You may create vectors of data using the following syntax, analogous to the syntax for segments:

- `vector of l` creates a vector whose elements are those of the list `l`.
- `vector n of e` creates a vector of length n, whose elements are all equal to `e`.

An alternative way to create vectors is to enumerate their elements using the concrete notation for vectors:

```
#[|1;2;3|];;
[|1; 2; 3|] : num vect
```

6.5.2 Primitive operations on vectors

Access and physical modification of cells in a vector have concrete syntax constructs similar to access and replacement in mutable structures:

- `vector "." "(" <expression> ")"`
  where `vector` and `expression` are expressions evaluating respectively to a vector `v` and a number `n`, returns the content of the cell of index `n` in the vector `v`.

- `vector "." "(" <expression1> ")" "<-" <expression2>`
  where `vector`, `expression1` and `expression2` are expressions evaluating respectively to a vector `v`, a number `n` and an expression `e`, physically modifies the content of the cell of index `n` in the vector `v`, which becomes the value `e`. The value returned by the whole construct is `e`.

```
#let v = [|1;2;3|];;
Value v = [|1; 2; 3|] : num vect

#let w = [| v; [|5|] |];;
Value w = [|[|1; 2; 3|]; [|5|]|] : num vect vect

#(* Access: vect . ( expr ) *)
#w.(0);;
[|1; 2; 3|] : num vect

#w.(0).(0);;
1 : num

#(* Replacement:  vect . ( expr ) <- expr *)
#v.(0) <- 2; v;;
[|2; 2; 3|] : num vect

#w.(0);;
[|2; 2; 3|] : num vect
```

Primitive functions to access and assign vector's cells are available:

♣
```
vect_assign : ('a vect * num * 'a -> 'a)
vect_item : ('a vect * num -> 'a)
```

- `vect_item (vect,n)` returns the item number `n` of the vector `vect`.
- `vect_assign (vect,n,value)` changes the element of the vector `vect` whose index is `n` by the CAML value `value`.

♣
```
vect_length : ('a vect -> num)
is_null_vect : ('a vect -> bool)
```

- `vect_length vect` returns the length of the vector `vect`.
- `is_null_vect` is bound to the expected predicate and allows one to test if a vector has no items at all.

♣
```
do_vect : (('a -> 'b) -> 'a vect -> unit)
modify_vect : (('a -> 'a) * 'a vect -> unit)
map_vect : (('a -> 'b) -> 'a vect -> 'b list)
do_vect_i : ((num -> 'a -> 'b) -> 'a vect -> unit)
modify_vect_i : ((num -> 'a -> 'a) * 'a vect -> unit)
map_vect_i : ((num -> 'a -> 'b) -> 'a vect -> 'b list)
list_of_vect : ('a vect -> 'a list)
fold_vect :
(('a -> 'b -> 'a * 'c) -> 'a -> 'b vect -> 'a * 'c list)
it_vect : (('a -> 'b -> 'a) -> 'a -> 'b vect -> 'a)
vect_it : (('a -> 'b -> 'b) -> 'a vect -> 'b -> 'b)
```

- `do_vect f vect` is the analog of `do_list` for vectors: it executes `f` on each element of the vector `vect` and returns `()`.
- `modify_vect f vect` maps the function `f` on the element of the vector `vect` and modifies each item with the result of the application of `f` to this item.
- `map_vect f vect` is the analog of `map` for vectors: it returns the list of results of applying `f` on each element of the vector.
- `map_vect_i`, `do_vect_i` and `modify_vect_i` are slightly different versions of the primitive without `_i`: they provide the item rank in the vector to which the function is applied.
- `list_of_vect v` returns the list of the elements of the vector `v`.
- `fold_vect`, `it_vect`, `vect_it` are the functionals for vectors corresponding to `fold`, `it_list` and `list_it`.

```
let do_vect f v = paired_do_vect (f,v)
;;

let do_vect_i f v = paired_do_vect_i (f,v)
;;
```

```
let map_vect f v = maprec 0
 where rec maprec n =
  if n >= vect_length v then [] else f v.(n)::maprec (succ n)
;;

let map_vect_i f v = maprec 0
 where rec maprec n =
  if n >= vect_length v then []
   else f n v.(n)::maprec (succ n)
;;

let list_of_vect = map_vect I
;;

let fold_vect f a1 v =
 let l = vect_length v in
 fold_f a1 0
 where rec fold_f a1 n =
  if n >= l then a1,[]
   else (let a2,c2 = f a1 v.(n) in
         let a,cl = fold_f a2 (succ n) in a,c2::cl)
;;

let it_vect f a v = it_vect_f a 0
 where rec it_vect_f a n =
  if n >= vect_length v then a
   else it_vect_f (f a v.(n)) (succ n)
;;

let vect_it f v a = vect_it_f 0
 where rec vect_it_f n =
  if n >= vect_length v then a
   else f v.(n) (vect_it_f (succ n))
;;
```

♣ `vect_of_obj : (obj -> obj vect)`
`is_vector : (obj -> bool)`

- `is_vector obj` tests if its `obj` argument is the representation of a CAML vector.
- `vect_of_obj obj` given an obj which is the representation of a vector returns this vector as a vector of objs.

Chapter 7

String Manipulation Functions

Important Note:

The positions of the characters in a string are always counted beginning at 0.

Thus the first character of a string has position 0, and the last character of a string of length 3 has position 2.

7.1 Definition of strings

Strings are sequences of ISO 8859/1 8 bits characters[1], enclosed between double quote symbols "", for example `"This is a string"`.

7.1.1 Escape character

Inside strings, there are some special conventions: the character "\" named the *escape* character, has a particular meaning:

- "\\" is interpreted as the back-slash character.
- "\"" is interpreted as the doublequote character.
- "\n" is interpreted as the line-feed character.
- "\t" is interpreted as the tab character.
- "\f" is interpreted as the form-feed character.
- "\r" is interpreted as a carriage return.
- "\b" is interpreted as a back space.
- "\c" where c is a line-feed character is ignored: this is used to continue a string which may then spread onto several lines. In addition all spaces after the line-feed are ignored.

[1] In this documentation these characters are improperly called ASCII characters.

- "\n" ($0 < n \leq 9$) is interpreted as n spaces.
- "\0" is interpreted as 10 spaces.
- "\S" is interpreted as 1 space.
- "\c" where c is any other character, is finally interpreted as the character c.

```
#message"Un doublequote \" !";
#message"Une barre oblique \\ !";
#message"Une tabulation\t!";
#message"Premie're ligne\nDeuxie'me ligne !";
#message"Une fausse tabulation\8!";
#message"\a";;

Un doublequote " !
Une barre oblique \ !
Une tabulation  !
Premie're ligne
Deuxie'me ligne !
Une fausse tabulation        !
a
() : unit
```

The character \ at the end of a line indicates that the string is continued on the next line:

```
#message"This chain has : - no line-feed in it \
#                         - nor extra spaces!";;
This chain has : - no line-feed in it - nor extra spaces!
() : unit
```

Notice that following the common toplevel printing habit, *escapes* characters (\'s) are reintroduced into the strings before the characters \ and " when printing these strings at toplevel in order to be able to input to CAML exactly what it displays to the user:

```
#"\"";;
"\"" : string

#"\\";;
"\\" : string
```

Beware:
-- The other escaped characters are not reintroduced in the strings.
-- Non printable characters have no defined printing conventions, thus the CAML system is unable to read them.

There are primitives to print or turn a string into a readable string:

```
♣ string_for_read : (string -> string)
  print_string_for_read : (string -> unit)

  #print_string_for_read "\\\"";;
  \\\"() : unit
```

7.1.2 Predefined common characters

The following characters are bound to CAML variables:

```
♣ return_char : string
  line_feed_char : string
  form_feed_char : string
  tab_char : string
  space_char : string
  back_space_char : string
  space_chars : string
```

The definition of these identifiers is:

```
let return_char = "\r" (* ascii 13 *)
and line_feed_char = "\n" (* ascii 10 *)
and form_feed_char = "\f" (* ascii 12 *)
and tab_char = "\t" (* ascii 9 *)
and space_char = " " (* ascii 32 *)
and back_space_char = "\b" (* ascii 8 *);;

(* Common words separators *)
let space_chars =
    space_char^tab_char^line_feed_char^
    return_char^form_feed_char;;
```

7.2 Comparisons of strings

```
♣ eq_string : (string * string -> bool)
  le_string : (string * string -> bool)
  ge_string : (string * string -> bool)
  lt_string : (string * string -> bool)
  gt_string : (string * string -> bool)
```

These functions use the lexicographic ordering induced by the ASCII code in order to compare their arguments. `eq_string` is a specialised version of = (a little more efficient). The others decide respectively if the first argument is less than or equal, greater than or equal, less than and greater than the second one.

7.3 Length of strings

♣ `length_string : (string -> num)`

`length_string s` returns the number of characters in `s`.

```
#length_string("This chain is 32 characters long");;
32 : num
```

7.4 String Constructors

♣ `concat : (string -> string -> string)`
`prefix ^ : (string * string -> string)`
`implode : (string list -> string)`

The first one returns the concatenation of its arguments, it is also provided as the infix operator ^. The second one returns the concatenation of all the elements of its string list argument.

```
#concat "Comment passer" "\na' la ligne suivante";;
"Comment passer
a' la ligne suivante" : string

#"Voici la " ^ "version infixe " ^ "de concat.";;
"Voici la version infixe de concat." : string

#implode
# ["Comment indenter";"\n\tun texte
#  "\navec une tabulation"];;
"Comment indenter
        un texte
avec une tabulation" :
 string
```

Notice that `implode` is definable in CAML, as an alias of the obsolete `concat_list` function:

```
#let concat_list l = list_it concat l "";;
Value concat_list = <fun> : (string list -> string)

#let implode = concat_list;;
Value implode = <fun> : (string list -> string)
```

♣ `sub_string : (string -> num -> num -> string)`

`sub_string s pos len` returns the string of length `len` beginning at `pos` in `s`. It only fails when `pos` is less than 0. Asking for too many characters results in getting the end of the string. Asking for 0 or less than 0 character results in the empty string. Asking for a substring beginning after the end of the string results in the empty string.

```
#sub_string "Mes 5 premiers caracteres" 0 5;;
"Mes 5" : string

#sub_string "mais si on en demande trop" 4 300;;
" si on en demande trop" : string

#sub_string "ou trop peu" 1 (-4);;
"" : string

#sub_string "ou ailleurs" (-5) 4;;

Evaluation Failed: sub_string
```

♣ `explode : (string -> string list)`

`explode s` returns the list of all the characters in `s`, each of them is a string of length 1.

```
#explode "abcdef";;
["a"; "b"; "c"; "d"; "e"; "f"] : string list
```

Thus, `implode o explode` is the (roughly speaking) identity function.

Now you can easily define a function which reverses a string as in:

```
#let rev_string = implode o rev o explode;;
Value rev_string = <fun> : (string -> string)

#rev_string "TulastropecraseCesarceportsalut";;
"tulastropecraseCesarceportsaluT" : string
```

♣ `ascii_code : (string -> num)`

Returns the internal ASCII code of the first character of its argument, which must be a non-empty string.

```
#ascii_code "L";;
76 : num

#ascii_code "Lmnop";;
76 : num

#ascii_code "";;

Evaluation Failed: nth_ascii
```

♣ `ascii : (num -> string)`

`ascii n` returns a string of length 1, whose unique character has `c` as internal ASCII code, where `c` is defined as `(integer n) mod 256`.

```
#ascii 68;;
"D" : string

#ascii (-184.76);;
"H" : string
```

Beware:

-- `ascii` returns chars in the whole range `[0-255]`: this may be confusing for those programs which do not test these cases and assume that a character has code less than 128 (for example when indexing an array with the numeric value of the character this may lead to fatal errors): in the following example, the obtained character is not printable. Thus the CAML toplevel forces it in the range `[0-127]` in order to print it. However it retains its real code in the internal representation.

```
#ascii 196;;
"D" : string
#it = "D";;
false : bool
```

♣ `implode_ascii : (num list -> string)`

`implode_ascii l` returns the string which is the concatenation of all the characters whose internal ASCII codes are the elements of `l`.

```
#implode_ascii [67;65;77;76];;
"CAML" : string

#implode_ascii [(68+256);68];;
"DD" : string
```

Beware:
-- `implode_ascii` creates strings whose characters are in the whole range `[0-255]`, with the same conversion as for the function `ascii` (and the same printing confusion):

```
#implode_ascii [127;(-1);0;(1/2)];;
"^?^?^@^@" : string
#implode_ascii [(-1)] = implode_ascii [127];;
false : bool
```

♣ `explode_ascii : (string -> num list)`

`explode_ascii s` returns the list of the ASCII codes of the characters of `s`.

```
#explode_ascii "Dodo";;
[68; 111; 100; 111] : num list

#explode_ascii (implode_ascii [(68+256);68;127;(-1)]);;
[68; 68; 127; 255] : num list
```

♣ `make_string : (num -> string -> string)`

`make_string n s` returns a new string of `n` characters which are all identical to `c`, where `c` is the first character of `s`. `make_string` is equivalent to the following CAML definition:

```
let make_string n s =
 implode (replicate n (ascii (ascii_code s)));;

#make_string 5 "a";;
"aaaaa" : string

#make_string 5 "abcd";;
```

```
"aaaaa" : string

#make_string 0 "a";;
"" : string

#make_string (-1) "a";;

Evaluation Failed: make_string
```

♣ `replace_string : (string -> string -> num -> string)`

`replace_string s1 s2 pos` *physically* replaces in `s1` the string beginning in `pos` by `s2`, and then returns the (modified) string `s1`. It does not replace more characters than the length of `s2` and never changes the length of `s1`.

```
#replace_string "monsieur" "es" 1;;
"messieur" : string

#replace_string "monsieur" "eigneur" 4;;
"monseign" : string

#replace_string "monsieur" " le pre'sident" 8;;
"monsieur" : string
```

Beware:

-- since `replace_string` physically modifies `s1`. In case of sharing this may be very confusing. One has to keep track of copies to find which string will be modified:

```
#let s = "papa";;
Value s = "papa" : string

#(* Second occurrence of s is a copy ! *)
#let t = (s,"Mon " ^ s);;
Value t = ("papa","Mon papa") : (string * string)

#replace_string s "i" 3;;
"papi" : string

#t;;
("papi","Mon papa") : (string * string)
```

7.5 Access functions in strings

♣ `index_string : (string -> string -> num -> num)`

`index_string s1 s2 pos` returns the position of the beginning of the first occurrence of `s2` in `s1` after position `pos`. Returns `-1` if `s2` does not occur in `s1` after position `pos` and returns `pos` if `s2` is `""`.

```
#let str = "Ah le beau langage que voila!";;
Value str = "Ah le beau langage que voila!" : string

#index_string str "la" 1;;
11 : num

#index_string str "la" 8;;
11 : num

#index_string str "la" 100;;
-1 : num

#index_string str "la" (-12);;

Evaluation Failed: index_string

#index_string str "gg" 10;;
-1 : num
```

♣ `pos_string : (string -> string -> num)`

`pos_string s1 s2` returns the position of the beginning of the first occurrence of `s2` in `s1`. It may be defined by

```
let pos_string s1 s2 = index_string s1 s2 0;;
```

♣ `scan_string : (string -> string -> num -> num)`

`scan_string s1 s2 pos` returns the position of the first occurrence in `s1` after `pos` of *any* character belonging to `s2`.

```
#scan_string
# "N'e'crivez que quand l'inspiration vous posse'de
#  et vous presse. (G. Sand)" "Non" 0;;
0 : num

#scan_string
# "La Terre n'est pas une plane'te quelconque.
#  (A. De Saint-Exupe'ry)" "pas" 500;;
-1 : num

#scan_string
# "Si je vais de rocher en rocher, le me^me torrent
#  devient autre a' chaque pas. (Alain)" "oui" (1/2);;
1 : num

#scan_string
# "Il e'tait, quoique riche, a' la justice enclin (V. Hugo)"
# "oui" (-1);;

Evaluation Failed: scan_string
```

♣ `span_string : (string -> string -> num -> num)`

`span_string s1 s2 pos` returns the position of the first occurrence in `s1` after pos of *any* character NOT belonging to `s2`.

```
#span_string "Bonne anne'e""Bonne sante'" 0;;
-1 : num

#span_string "Bonne anne'e""annie" 6;;
10 : num

#span_string "Bonne anne'e""anne" 80;;
-1 : num

#span_string "Bonne anne'e""anne'e 1988" 6;;
-1 : num

#span_string "Bonne anne'e""annette" (-2);;

Evaluation Failed: span_string
```

♣ `nth_char : (num -> string -> string)`
`nth_ascii : (num * string -> num)`

- `nth_char n s` returns the character in position `n` in `s`, as a single character string.
- `nth_ascii (n,s)` returns the ascii code of the character in position `n` in `s`.

```
#nth_char 0 "oups!";;
"o" : string

#nth_char 0 "";;
"" : string

#nth_char (-1) "oups!";;

Evaluation Failed: sub_string

#nth_char (17/16) "oups!";;
"u" : string
```

♣ `present : (string -> string -> bool)`

`present s1 s2` returns true if some character of `s2` is present in `s1` (this ensures that `s1` and `s2` have at least one common character).

```
#present
# "Des voiles s'enfuyant comme l'espoir qui passe. (V. Hugo)"
# "()";;
true : bool

#present "oui" "";;
false : bool
```

♣ `first_n_string : (num -> string -> string)`
`last_n_string : (num -> string -> string)`

`first_n_string n s` (resp `last_n_string n s`) returns the first (resp last) `n` characters of `s`. There must be at least `n` characters in `s`.

```
#first_n_string 5
# "Ils continue'rent leur route, allant toujours sans savoir
#  ou' ils allaient ... trompant l'ennui et la fatigue par le
#  silence et le bavardage. (Diderot)";;
"Ils c" : string

#first_n_string 100 "Passe' le danger, on se moque du saint.";;

Evaluation Failed: first_n_string

#first_n_string 0
# "Ils croyaient s'affranchir suivant leur passions
#  ils e'taient esclaves d'eux-me^mes. (La Fontaine)";;
"" : string

#first_n_string (-1)
# "Le bonheur s'obtient en n'y pensant pas.
#  (H de Montherlant)";;
"" : string

#first_n_string (1/2)
# "Sans doute tout n'est pas e'gal dans ce petit livre
#  encore que je n'en voulusse rien retrancher. (A. Gide)";;
"" : string
```

♣ `extract_string : (string -> num -> num -> string)`

`extract_string s from to` returns the part of the string `s` beginning in position `from` and ending in position `to`. The value of `from` is not allowed to be less than 0.

```
#extract_string "poipoipoi" 1 2;;
"oi" : string

#extract_string "poipoipoi" 2 1;;
"" : string

#extract_string "poipoipoi" 2 (-1);;
"" : string

#extract_string "poipoipoi" 20 21;;
"" : string
```

```
#extract_string "poipoipoi" (-1) 2;;

Evaluation Failed: sub_string

#extract_string "poipoipoi" 1 100;;
"oipoipoi" : string
```

♣ `skip_string : (string -> string -> string)`

`skip_string skip s` returns the substring of `s` beginning with the first character not belonging to `skip` and ending at the end of `s` .

```
#% To skip spaces and tabs at the beginning of a string %
#let skip_spaces_and_tabs =
#    skip_string (implode_ascii [9;32]);;
Value skip_spaces_and_tabs = <fun> : (string -> string)

#skip_spaces_and_tabs
# ("  "^(ascii 9)^"ab cd srt"^(ascii 9)^"A");;
"ab cd srt A" : string
```

In this family, two functions are predefined in the CAML system:

♣ `skip_space : (string -> string)`
`skip_space_return : (string -> string)`

The former one is equivalent to our preceding example `skip_spaces_and_tabs`, and the last one to `skip_string space_chars`.

♣ `chop_string : (num -> string -> string * string)`

`chop_string n s` splits `s` into two strings at position `n`.

```
#chop_string 3 "abcdefgh";;
("abc","defgh") : (string * string)

#chop_string (-3) "abcdefgh";;

Evaluation Failed: chop_string

#chop_string 100 "abcdefgh";;
("abcdefgh","") : (string * string)

#chop_string (1/2) "abcdefgh";;
("","abcdefgh") : (string * string)
```

7.6 Word operations

♣ `first_word : (string -> string)`

`first_word s` extracts the first word of the string `s` (it returns the string beginning with the first character not in `space_chars` and ending as soon as one of these characters is encountered).

```
#first_word " The CAML language !";;
"The" : string

#first_word "The CAML language !";;
"The" : string

#first_word "\nThe CAML language !";;
"The" : string
```

♣ `rev_words : (string -> string -> string list)`
`words2 : (string -> string -> string list)`
`words : (string -> string list)`
`break_string : (string -> string -> string list)`

- `rev_words sep s` splits the string `s` into words separated by one (or several) of the characters belonging to `sep` (the result is given in reverse order which may be sometime useful).

  ```
  #rev_words " " " The CAML language !";;
  ["!"; "language"; "CAML"; "The"] : string list

  #rev_words "-., " " The ,caml lan---guage !";;
  ["!"; "guage"; "lan"; "caml"; "The"] : string list
  ```

- `words2 sep s` splits the string `s` into words separated by one or several of the characters of the string `sep`. `words2` is equivalent to:

  ```
  let words2 sep = let revwords = rev_words sep in
      function s -> rev (revwords s);;
  ```

- `words` is equivalent to:

  ```
  let words = words2 space_chars;;
  ```

- `break_string sep s` splits the string `s` into words separated by `sep`, without skipping multiple occurrences of `sep`:

```
#break_string "_" "ab_cd__e_";;
["ab"; "cd"; ""; "e"; ""] : string list
```

7.7 Coercion and Conversion Functions

There exist coercion and conversion functions between the type `string` and other CAML types.

♣
```
string_of_bool : (bool -> string)
string_of_obj : (obj -> string)
string_of_num : (num -> string)
bool_of_string : (string -> bool)
obj_of_string : (string -> obj)
num_of_string : (string -> num)

#(* Conversions bool <--> string *)
#string_of_bool true;;
"true" : string

#bool_of_string "true";;
true : bool

#bool_of_string "yes";;

Evaluation Failed: bool_of_string

#(* Conversion string towards obj *)
#obj_of_string "oui";;
<:obj<oui>> : obj

#obj_of_string "1";;
<:obj<1>> : obj

#obj_of_string "";;

System error: implode <:obj<1>>

#obj_of_string "\"\"";;
<:obj<"">> : obj
```

```
#(* Conversion string towards num *)
#num_of_string "1";;
1 : num

#num_of_string " 1";;

Evaluation Failed: num_of_string

#num_of_string "-1";;
-1 : num

#num_of_string "- 1";;

Evaluation Failed: num_of_string

#num_of_string "1/2";;
1/2 : num

#num_of_string "-1/2";;
-1/2 : num

#num_of_string "zero";;

Evaluation Failed: num_of_string
```

Chapter 8

Dynamic values

In statically typed languages admitting user-defined types, some useful functions cannot be typed. Although the CAML type system is quite powerful, there exist functions which cannot be typed in the "core type system".

As an example, consider a function `extern` which writes its argument into a file. Since that function must be able to write a boolean value and also an integer, it must be universal (i.e. must work on any type of value). Consider now the inverse function `intern` which reads an object from a file and returns its value. The type of the `intern` function must be (`string -> `τ). Since the value returned may be of any type, the type τ cannot be an usual CAML type because it can be:

- neither a particular type without excluding all other types,
- nor the polymorphic type `'a`, because this would mean that a value returned by the function possesses *all the types* which is not the case: we only know that the value will have *one type*[1].

We do not need polymorphism here (the value returned does not possess *all types*) but a type being the *disjoint union of all types* (meaning that the value returned possesses *one type* amongst all the possible types).

Thus the idea to maintain the type of the value close to the value itself, so that `intern` will return a value **with its corresponding type** (as previously written by the function extern").

So, roughly speaking, we need a type for values composed of a value *and* its type. We call such values *dynamic values*. A type for dynamic values may also be useful to a universal printing function. A printing function must be applicable to values of any type but needs to know the type of its argument to be able to print it: some values may have identical internal representations while being of completely different types, and thus have different external representations.

For the next example, dynamic values are essential. We consider a CAML function which, when given a piece of CAML abstract syntax (i.e. a CAML expression), evaluates it and returns the value of that expression. Of course, the

[1] Note that such a value may of course possess several type instances if its type is polymorphic.

domain of such a function must be the type of abstract syntax trees of the language, but its codomain is the type of dynamic values since the result of a CAML evaluation is a value together with its type[2].

We can see the type of dynamic values as being an infinite disjoint union indexed by all possible CAML types. The type of dynamic values is implemented as the dyn CAML type.
The representation of a value of type dyn is composed of:

- the representation of a value v and
- the representation of a type τ such that $v : \tau$.

Obviously, only the type-checker is able to construct a dynamic value (i.e. values of type dyn) since it is the only part of the system which is able to know whether it is safe or not to decide if v is of type τ.

Once dynamic values are created, we want to be able to extract their internal value typed with their internal type. This process is called *coercion*, and is integrated into pattern-matching. A coercion of a dynamic value d with internal value v and internal type τ into a value of type τ' succeeds if the type τ' is an *instance* of the type τ. Moreover, it is possible to accept the coercion only if the value v is matched by a specific pattern.

Dynamic values all have the same type, it is thus possible to build heterogeneous lists for example simply by gathering each element with its type into a dynamic value. We often know the different types we want to build lists with, and defining a type as being the finite disjoint union of these types is usually sufficient.

8.1 Creating dynamic values

There is only one way to build dynamic values: the keyword "dynamic" is used in order to transform the value of an expression into a dynamic value. Syntactically, this is written as a function application:

```
#dynamic 1;;
(dynamic (1 : num)) : dyn

#dynamic (hd [true;false]);;
(dynamic (true : bool)) : dyn

#dynamic succ;;
(dynamic (<fun> : (num -> num))) : dyn

#[dynamic 1; dynamic true];;
[(dynamic (1 : num)); (dynamic (true : bool))] : dyn list
```

[2] We see here an analogy between top-level values and dynamic values.

Building a dynamic value from an expression succeeds if that expression is typable, and if that type is completely *generalizable*. This is because the compiler must know statically what type representation it has to compile. For example, this is not the case of the following function:

```
#let mk_dyn = function x -> dynamic x;;

line 1: cannot generalize type 'a
for dynamic expression: dynamic x
1 error in typechecking

Typecheck Failed
```

Type checking failed because the code for the `mk_dyn` function should dynamically depend of the type of the `x` parameter (and we don't want that code to be dynamically updated). Here, the type of `x` was *not* generic.
It is still possible to build dynamic values with generic internal types such as:

```
#dynamic (function x -> x,x);;
(dynamic (<fun> : ('a -> 'a * 'a))) : dyn
```

Here, the type of the function `(function x -> x,x)` is generic (there are no external type constraint on the `x` formal parameter).

The constraint about type variables occurring in the internal type of a dynamic value is similar to the one concerning top-level values: since values defined at top-level must be either monomorphic (i.e. their type contains no type variable) or generic, the type-checker refuses top-level type variables representing unknown types coming for example from the type of a modifiable value.

As top-level values can be used with different type instances in their scope[3], dynamic values with internal polymorphic types are still polymorphic when coerced (see the following section).

Summary

- A dynamic value is composed of the representation of a value v together with the representation of a type τ such that $v : \tau$.
- Dynamic values are created by using the "`dynamic`" <*Expression*> syntactic construct (from a syntactical point of view, analoguous to a function or constructor application).
- Creation succeeds if the expression is typable and its type is completely generalizable.

[3] This is also true for local values introduced by the `let (rec)` construct.

8.2 Coercion

As mentioned above, the fact of retrieving a value typed in the usual way from a dynamic value is called *coercion*. This facility is integrated to usual pattern-matching. The class of patterns is extended by the following rule:

<*Pattern*> ::= ...
| "`dynamic`" "(" <*Pattern*> ")"

Syntactically, the keyword "`dynamic`" is used as a constructor and parentheses may be unnecessary in some cases (depending of the pattern following "`dynamic`"). Such patterns will be called *dynamic patterns* in the following.

A dynamic pattern may appear everywhere a pattern is legal, particularly inside of another pattern.

A coercion means *coercion to a particular type*. With the syntax above, since type constraints are legal inside patterns, it is possible to specify a type into which the dynamic values have to be coerced using these constraints. However, sometimes the type constraint is redundant because the pattern which is supposed to match the (value part of the) dynamic value can be sufficiently explicit. For example, in `(dynamic 1)`, there is no need of `(1 : num)`. So, the type-checker infers by itself the "internal" type of a dynamic pattern, but needs sometimes a type constraint in order to produce a coercion into a more specific type. We will see below that the more specific are internal types of dynamic patterns, the more values they match. The dynamic pattern `(dynamic x)`, which is equivalent to `(dynamic (x : 'a))` *will never match any value*[4]! The point is that type variables and value variables in patterns play totally different roles.

From now on, we will note $(p : \tau)$ a dynamic pattern `dynamic` (p) where p has type τ.

8.2.1 Matching with dynamic patterns

During the process of matching a value d against a dynamic pattern $p : \tau$, d must be of type `dyn`. Let v_d be the internal value of d and τ_d be the type denoted by the type part of d. If the type τ is not an instance of τ_d then the next rule is tried or the matching fails if the current rule was the last one. If τ is an instance of τ_d, then v_d is matched against the pattern p. If that matching succeeds, then the matching of d against the dynamic pattern succeeds, otherwise it fails and execution proceeds as usual.

In the following example, the second rule is chosen:

```
#match dynamic (2+3) with dynamic true -> 1
#                       | dynamic 5    -> 2
#                       | dynamic 6    -> 3;;
Warning: 1 partial match in this phrase
2 : num
```

[4]This happens also with the pattern `dynamic (x : empty)` where the type `empty` is defined by: `type empty = E of empty`.

Notice that, in this example, matching rules are not exhaustive and that the only way to write a total match is to add a rule with a pattern restricted to a variable or a "catch all" rule as in:

```
#match dynamic (2+3) with dynamic (_ : bool) -> 1
#                       | dynamic (n : num) -> n
#                       | _ -> 0;;
5 : num
```

Even if the patterns are exhaustive for each internal type considered, the only way of being exhaustive on the internal types is to add a rule matching all possible internal types. It is easy to see that only a variable fulfills this role.

8.2.2 Polymorphism

A dynamic pattern of type τ will match only dynamic values whose type part is more general than τ. Thus the variables bound by a dynamic pattern are polymorphic. For example, we may write:

```
#let f = function dynamic (I : 'a -> 'a) -> (I true, I I)
#                 | _ -> failwith "was waiting for identity";;
Value f = <fun> : (dyn -> bool * ('a -> 'a))
```

where the bound variable I has different type instances in its scope. We may now write:

```
#f (dynamic I);;
(true,<fun>) : (bool * ('a -> 'a))

#f (dynamic succ);;

Evaluation Failed: was waiting for identity
```

8.2.3 Order of match rules

As in usual pattern-matching, the textual order of match rules is important, but may be counter-intuitive in presence of dynamic patterns. Some match rules may be ignored by the compiler if it detects that the considered rule will never be used. Rules containing too specific dynamic patterns (corresponding to more general types) may be ignored if more general patterns (using types instances) precede them. For example, the second case of the following function will always be ignored:

```
#function dynamic (L : num list) -> L
#         | dynamic (L : 'a  list) -> L ;;
Warning: 1 partial match in this phrase
<fun> : (dyn -> num list)
```

That function first tries to match a list of numbers, and, in case of failure, should try to find a value with type `'a list`. It is easy to see that if the argument was such a value, then it would have been matched by the first rule, hence the second rule may be ignored.

The value part of dynamic patterns has also to be considered. The following function does not have any unused match case:

```
#function ((_::_ as L) : num list) -> L
#       | (L : 'a list) -> L;;
<fun> : (num list -> num list)
```

The second rule will be chosen when the argument is the dynamic empty list of type `'a list`.

Summary

- Coercion of dynamic values into usual values is integrated into pattern-matching. A new kind of patterns is introduced. They are written:

  "`dynamic`" "(" <*Pattern*> ")"

- Matching a dynamic value d against a dynamic patterns $p : \tau$ consists in checking that τ is an instance of the internal type of d *and* that p matches the internal value of d. If one of these fails, then the pattern-matching fails.

- All variables bound by a dynamic pattern are polymorphic.

8.3 Functions manipulating dynamic values

Several functions need to use the type dyn in order to be typable. For each function below, the reader will find more information about it in the section following its name.

♣
```
print : (dyn -> unit)
print_dyn : (dyn -> unit)
```

`print` prints its (dynamic) argument in the standard output.

```
#print_dyn (dynamic 1);;
(dynamic (1 : num))() : unit

#print (dynamic 1);;
1() : unit
```

print can be used as a generic printing routine (with typing restrictions due to building of dynamic values) (see 11.2.1).

♣ `eval_syntax : (ML -> dyn)`

eval_syntax evaluates a CAML abstract syntax tree (of type ML) into a dynamic value (see 17.1).

```
#eval_syntax <:Caml:Expr<1+2>>;;
(dynamic (3 : num)) : dyn
```

♣ `eval_string : (string -> dyn)`

eval_string evaluates a string which is a CAML expression in concrete syntax (of type string) into a dynamic value (see 17.2).

```
#eval_string"1+2";;
(dynamic (3 : num)) : dyn
```

♣ `extern : (string -> dyn -> unit)`

extern writes a (non functional) dynamic value onto persistent storage (in a file) (see 9.1).

♣ `intern : (string -> dyn)`

intern restores a dynamic value in the current session from persistent storage (see 9.2).

♣ `MLquote : (dyn -> ML)`

MLquote is the constructor which builds the CAML abstract syntax tree for a constant value (see 16.3.2).

Chapter 9

Persistent objects

It is possible to save CAML values into external memory. The values have to be of type dyn (cf. chapter 8). They may also be read from external memory.

This is an experimental package: the functions described below may be quite slow in presence of very large data structures.

9.1 Saving persistent objects

The fonction `extern` takes as argument a file name and a dynamic value and writes the dynamic value into the specified file with "`.obj`" suffix appended.

♣ `extern : (string -> dyn -> unit)`

As an example:

```
#extern "/tmp/mylist" (dynamic (interval 0 5));;
/tmp/mylist.obj written
() : unit

#probe_file "/tmp/mylist.obj";;
true : bool
```

The `extern` function fails when:

- the file cannot be written
- one attempts to save either functional values or non complety evaluated (lazy) data structures.

The `extern` function saves a value and preserves its sharing: shared substructures are still shared after restoring them in a core image. It saves also type information about the `extern`-ed value and the consistence of that information is checked when the persistent object in restored. This type information is similar to the import statement of a module and the verification acts as checking the legality of that statement.

9.2 Restoring persistent objects

The function `intern` takes as argument a file name (without `.obj` extension) and returns a dynamic value.

♣ `intern : (string -> dyn)`

As an example:

```
#intern "/tmp/mylist";;
(dynamic ([0; 1; 2; 3; 4; 5] : num list)) : dyn
```

The `intern` function fails when:

- the specified file does not exist or is unreadable;
- the specified file has not been created by the `extern` function;
- the internal types of the objects do not exist in the current session or do not have the same definition.

Chapter 10

The CAML Channel System

We now describe the CAML character channel system. This system is essentially intended to provide Input/Output operations, although it includes a "pipe" feature which provides communication between (generally mutually recursive) functions of a CAML process. This I/O system is partially an implementation of the paper "Standard ML Input/Output" by Robert W. Harper (Polymorphism, Vol II, Number 2, October 85).

Introduction

Beginners may find an example of a simple session using channels in the CAML Primer. This very part of the documentation is rather technical and not intented to be a tutorial ...

All the functions constituting the channel system handle characters (as internal codes of type num), or strings, or character streams.

There are two types of channels: input channels (CAML type `in_channel`) and output channels (CAML type `out_channel`).

Each has several variants according to which kind of communication (file, terminal, pipe) is provided by the channel. A channel may be either open (able to work) or closed (and may no longer be used).

Input channels may be *interactive* or not: interactive input channels may wait until they receive *all* the characters they are asking for; non interactive do not (see functions `read` and `read_line`): they return immediately what is available in the input (even if this is nothing at all).

The terminal is always an interactive input channel, pipes are never, files may or may not be interactive according to the specification given when creating them.

Output channels may be simple (associated with one output) and are named `out_channel` or multiple (associated with as many outputs as one wants) and are named `broadcast_channel`.

Each channel is connected to a physical communication channel, named a "port". Up to 8 simultaneously active ports are available (let alone the ones associated to the predefined channels `std_in` and `std_out`).

Several active channels may share the same port, for example when they all write to the same file (and similar for read).

When failing during the I/O operations, all the functions raise the predefined exception `io_failure` with a string describing (as far as possible) the cause of the failure.

10.1 Predefined Channels

There are two predefined channels at the beginning of a CAML session:

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

They are connected with the standard input and output of the session (the terminal for an interactive session).

These two channels are rather special. One cannot close them and one cannot open other channels connected to the standard I/O ports. As `std_in` and `std_out` are ordinary CAML identifiers (and thus may be bound to other values), one has to take care not to lose the standard I/O channels if they are further needed.

10.2 Channel Opening Functions

♣
```
open_in : (string -> in_channel)
inter_open_in : (string -> in_channel)
```

These two functions create input channels from files. The second creates an interactive one and the first a non-interactive one.

The `<string>` argument is the name of the file as it must be given to the operating system. The file must exist when executing one of these functions.

If no input channel was connected to the file when executing these functions, the newly created one will begin reading at the beginning of the file. If one (or more) input channels were already connected to the file the new one will be placed at the same place as (all) the previously defined ones. This fact (together with the input functions operating mode) implies that

Beware:

-- *All* the input channels connected to a given file always read together, that is at the same place in the file (it is impossible to get one channel reading from the beginning of the file and another one reading from the middle).

♣ `open_out : (string -> out_channel)`

This function creates an output channel towards a file. The `<string>` argument is the name of the file as it can be given to the operating system.

The effect of this function depends on the situation when it is executed:

If one (or more) output channel was already connected to the file, the new one will be placed at the end of the file, where (all) the previously defined ones already is (are).

If no output channel was connected to the file, the file is truncated to size 0, if it already existed, a new (empty) file, with name the `<string>` parameter of `open_out`, is created and the new channel is ready to write to it.

♣ `open_append : (string -> out_channel)`

The only difference with `open_out` is the following:

If a file of name `<string>` already exists and no output channel is connected with it, it is not truncated and the new channel will write, appending text to the end of the file.

♣ `broadcast : (out_channel list -> out_channel)`

This function creates a new output channel, which will write in *all* the channels of the list which is given as parameter. Such channels are intended for allowing echoing of output. Channels in the list must of course be output channels, but are allowed to be broadcast channels themselves. The list is not allowed to be empty: in this case `broadcast` raises exception `io_failure` with value `empty broadcast`.

♣ `new_pipe : (unit -> in_channel * out_channel)`

This function creates a pair of channels (one input and one output), the input one reading what is written by the output one. Such a pair is intended to let processes communicate (for example when using a pre-processor while processing an input). They are only associated with a memory buffer, which is a list of strings (not with a file). The size of the strings in the buffer is defined when creating the pipe. One may know the value that will be used and modify it, if needed, using the following functions (initially, it is set to 256):

♣ `pipe_buffer_string_length : (unit -> num)`
`set_pipe_buffer_string_length : (num -> num)`

Failure strings for opening functions are:

- `missing file name`: the empty string was given as name for a file[1].
- `no channel available`: user has to close some channel before opening one more.
- `cannot open file`: non existent file, access rights or such error.
- `system`: all other system errors.

Example:

```
#let foo = open_in "bar";;
Value foo = (in_channel
      (file "/usr/local/caml/V2-6.1/doc/manual/bar",opened,
       non interactive,port 6)) :
      in_channel

#let foo = open_out "bar";;
Value foo = (out_channel
      (file "/usr/local/caml/V2-6.1/doc/manual/bar",opened,
       port 6)) :
      out_channel

#let foo = broadcast [std_out; foo];;
Value foo =
      broadcast
       [(out_channel (terminal,opened,port (-1)));
        (out_channel
        (file "/usr/local/caml/V2-6.1/doc/manual/bar",
         opened,port 6))] :
      out_channel

#let (i_s, o_s) = new_pipe ();;
Value i_s = (in_channel
      (pipe 1,opened,non interactive,no port)) : in_channel
Value o_s = (out_channel (pipe 1,opened,no port)) :
      out_channel
```

10.3 Channel Closing Functions

It is not necessary to close the channels, but closing them allows the system to recover the resources they used (buffers and channels). Moreover, closing an output

[1] even if it is not forbidden by the underlying operating system.

channel connected to a file ensures that the whole buffer is written in the file (see the functions `output` and `flush`).

♣ `close_in : (in_channel -> unit)`
`close_out : (out_channel -> unit)`

These functions close a channel, returning () as value. Their meaning is that the channel will no more be used and that the system resources it was using may be reallocated.

Closing an input (resp. output) channel breaks the physical communication with a file only if it was the last input (resp. output) channel associated with this file.

Closing an output channel towards a file writes to the file the whole content of the buffer associated with the file even if other channels are writing to this file and remain open.

Closing standard input and output is not possible.

Closing the (unique) input channel from a pipe deletes the pipe and will cause an error if writing is attempted to this pipe, even if the output channel is not closed.

♣ `close_in_file : (string -> unit)`
`close_out_file : (string -> unit)`

Close *all* the channels associated with a file. They physically remove the connexion with the file. It is *not* an error to use these functions when there exists no open channel connected with the file.

♣ `all_close_in_files : (unit -> unit)`
`all_close_out_files : (unit -> unit)`

They close all the channels associated with files.

Failure strings for closing functions are:

- `channel already closed`
- `cannot close standard`: attempt to close `std_in` or `std_out`.
- `unable to close`: failure in the closing process.
- `system`: all other system errors.

10.4 Read Functions

The basic assumption is that an input channel may be empty without causing a failure or stopping the process when a function attempts to input characters from this channel. Particular cases are the functions `read` and `read_line`.

When taking characters from a file using an input channel, if other input channels are open from the same file, the characters which have been taken by the input function are no longer available for the other channels. Next attempt to input from this file, using any of these channels, will begin with the first character that has been left by this input. This ensures that all input channels from a file are always at the same place in this file. In some sense all the channels reading from the same file compete to receive the characters available from that file.

♣ `input : (in_channel -> num -> string)`

Attempts to take `<num>` characters from the input channel `<in_channel>` and returns them as a string. If less than `<num>` characters are available from the channel, the function returns, without failing, all the available characters. In particular, if the channel is empty (when a channel is at the end of a file, for example) the empty string is returned.

No conversion is made on the characters, except that the end of line is always returned as a single character (line-feed, ASCII decimal code 10).

♣ `input_line : (in_channel -> string)`

Attempts to return all characters from the input channel `<in_channel>` up to an end-of-line mark. If no such mark is found in the channel, all the available characters are returned without causing a failure. If the mark is found, it is returned at the end of the string, in order to distinguish this case from the previous one.

♣ `lookahead : (in_channel -> string)`

This function returns the first character available from the input channel `<in_channel>` if there is any, or the empty string if the channel is empty. It differs from the input function, since the character is not removed from the channel and remains available for further input. Consequently, successive calls of the `lookahead` function with the same parameter return the same result.

♣ `end_of_channel : (in_channel -> bool)`

`end_of_channel x` is an equivalent of `lookahead x = ""`.

The four preceding functions `input`, `input_line`, `lookahead` and `end_of_channel` handle every kind of input channel. The following two (`read` and `read_line`) are devoted to `interactive` input channels, i.e. standard input or file input channels created with the `inter_open_in` function.

Interactive channels are assumed to evolve during the session (terminal for example, or files providing communication with external process) so that one may have to wait until sufficiently many characters become available when attempting to read them.

♣ `read : (in_channel -> num -> string)`

Returns `<num>` characters from the input channel `<in_channel>` which must be interactive.

If less than `<num>` characters are available from the channel when calling this function, it waits until they become available. Waiting is done using no (terminal input) or few (disk input) CPU time.

♣ `read_line : (in_channel -> string)`

Same as read except it waits until reading an end-of-line mark rather than a given number of characters. Since it only returns when the mark has been reached, this end-of-line mark is *never* part of the returned string.

Beware:

-- These two functions are highly system dependent: users have to test them before using them confidently.

♣ `ml_pipe_in : (io_buffer -> num -> string)`

Takes `<num>` char in the buffer of a pipe. Its essential job is the handling of the buffer. It could be a local function in input.

Some functions are furthermore provided to allow more efficiency in I/O processes, but their use needs more care than the preceding ones. They cannot be used with pipes.

♣ `buf_input : (in_channel -> string -> num -> unit)`

This function takes `<num>` characters in the `<in_channel>` and physically places them in the `<num>` first positions of the `<string>`. It raises exception `ml_eof` if

it cannot give as many characters as asked. This exception is of type num and is raised with the number of characters which have been placed in the <string>.

The <in_channel> must not be the "in_channel" part of a pipe.

♣ `get_ascii : (in_channel -> num)`

It reads one character from the channel and gives it as an internal (ASCII) code.

♣ `get_last_inpos : (in_channel -> num)`

Returns the position, in the input buffer, where the last character was read.

♣ `reread_but_last : (in_channel -> num -> string)`

Intended to be used with the preceding function. Allows one to get the string between the position <num> in the input buffer and the position before the last character read. One can thus scan the buffer with `get_ascii`, creating a new string only when one knows exactly which it must be: one avoids creating many intermediate strings that would only be garbage-collected.

Failure strings for reading functions are:

- `closed`: attempt to use a closed channel.
- `negative number of char`: in `input`, `read` or `buf_input`.
- `not interactive`: in `read` or `read_line`.
- `pipe`: in `buf_input`, `get_ascii`, `get_last_inpos` or `reread_but_last`.
- `too many chars`: `buf_input` asked for more characters than buffer may contain.
- `eof`: `get_ascii` was invoked on an empty channel.
- `unable to read`: other errors.

10.5 Write Functions

When sending characters to a file to which more than one output channel is open, the outputs of the different channels are interleaved in the file in the same order that they has been put into the buffer by the calls to the output functions.

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

Appends the `<string>` argument to the output channel `<out_channel>`.

If the channel is associated to the standard output, the string is sent to the terminal as soon as the function is executed.

If the channel is associated to a pipe, the string is available to the input channel associated with the pipe as soon as the function is executed.

If the channel is associated with a file, the string is placed (totally or partially) in a buffer[2], which is only sent to the file when full. Using the `flush` function, one can ensure that the file has received all the characters that have been sent to it by the `output` function.

♣ `output_line : (out_channel -> string -> unit)`

Adds an end-of-line mark at the end of the `<string>` argument and places this new string in the output channel `<out_channel>` using the `output` function.

♣ `output_ascii : (out_channel -> num -> unit)`

It places in the `<out_channel>` the character with internal (ASCII) code `<num>`. It does not build any string.

♣ `flush : (out_channel -> unit)`

Writes into a file the content of the associated buffer.

Two more functions are available, but are mostly intended for system use.

♣ `ml_file_out :`
`(channel_state ref -> io_buffer * io_port -> string -> unit)`

[2] one buffer for each file (and not for each channel).

It essentially does the buffer handling when writing to a file.

♣ `ml_file_out_ascii :`
`(channel_state ref -> io_buffer * io_port -> num -> unit)`

Same as `ml_file_out` for ASCII writing.

Failure strings for writing functions are:

- `closed`: attempt to use a closed channel.
- `pipe broken`: attempt to write into a pipe the `<in_channel>` part of which is closed.
- `unable to write`: other errors.

10.6 Informations on I/O operations

You may get the list of all the files currently used by the I/O system with:

♣ `in_files : (unit -> unit)`
`out_files : (unit -> unit)`

You may be aware that the two predefined types `in_channel` and `out_channel` are abstract ones: their constructors are non accessible when entering the core image and you can create values of these types only by an explicite call to one of the primitives which builds channels. Moreover there is a special printer used to display the channel values. In case of difficult debugging situations, you may ask the system to display the internal representation of channels using

♣ `show_in_channel_buffer : (in_channel -> unit)`
`show_out_channel_buffer : (out_channel -> unit)`

10.7 Redirecting the toplevel in channel

The user may redirect the CAML toplevel in channel with:

♣ `open_read_from : (in_channel -> unit)`
`close_read_from : (unit -> unit)`

Connected with the grammar facilities, this may be used to load a file which is written in some user's defined grammar.

The toplevel current in channel is reported by:

♣ `current_in_channel : (unit -> in_channel)`

10.8 Efficiency of I/O operations

I/O operations efficiency is very sensitive to the way you design your programs, hence a careful programmer may speed up his I/O routines more than one hundred times. So we shall examine the same algorithm, with different versions in order to compare computational behaviour of the different I/O primitives, and finally insist on how you must attach importance to the choice of the right primitive to use.

Roughly speaking one has to consider that files may be treated line per line rather than character per character: since characters operations need much more function calls and create many strings (namely a string for each character read or written).

Our example is a simple filter which removes all text surrounded by % in a file (i.e. the CAML convention for simple comments). The source file:

```
% Initial declaration %
let x = 1;;
```

will be output as

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

First we accomodate CAML to our aim:

```
#let v = ref [|1|];;
Val v = (ref [|1|]) : num vect ref

#(v := vect_of_list (replicate  20000 1)); ();;
() : void

#let v2 = ref [|1|];;
Val v2 = (ref [|1|]) : num vect ref

#(v2 := vect_of_list (replicate  10000 1)); ();;
() : void
```

This is to reduce the size of free space in the HEAP to about 40 Kbytes. This will ensure that in most of the cases the Garbage Collector will be called due to a lack of space for new strings values.

In order to know the run times of the different evaluations, we switch on the CAML timer:

```
#timers true;;
() : unit
Evaluation has needed: Runtime: 0.00s GC: 0.00s Conses: 0
```

Now CAML is ready. Notice that for each experiment the output to the target file is displayed on the screen, but we have omitted it here, since it is not our purpose to examine it ! Notice too that input file **comments** is 21706 characters long and the output file contains 16820 characters.

10.8.1 First version

Naive version: input is done one character at a time and no partial binding of the I/O functions with the channels. Moving one character needs one I/O call. This is of course the most inefficient version. It uses the following code:

```
let remove_comment_0 infile outfile =
  let inp = open_in infile
  and outp = open_out outfile
  and (from_filter, to_writer) = new_pipe ()  in
  let rec filter () =
   while (not (end_of_channel inp)) & ((lookahead inp) <> "%")
   do output to_writer (input inp 1) done;
   input inp 1;
   while (not (end_of_channel inp)) & ((input inp 1) <> "%")
   do () done
  and writer () =
   while (not (end_of_channel from_filter))
   do output (broadcast [std_out ; outp])
             (input from_filter 1) done;
   if (not (end_of_channel inp)) then (filter() ; writer ())
  in
  filter ();writer ();
  close_in inp;close_out outp;close_in from_filter;;

gc true;;
<<(gc 27 0 0 0 0 0 0 3 cons 24320 symbol 2348 string 7889 vector 2386
   float 1012 fix 1024 heap (39) code (217))>>

#remove_comment_0 "./aux/comments" "./aux/foo0";;
.................................................
Runtime: 485.47s

#gc_info ();;
<<(gc 67 0 0 0 0 0 40 3 cons 24320 symbol 2348 string 7889 vector 2386
   float 1012 fix 1024 heap (39) code (217))>>
```

We can see that 80 calls to GC were done (for a total time of about 280 seconds): 40 of them due to a lack of space in the CONS area, 40 due to a lack of space in the string value area. We can't evaluate the total string value space used since each GC works in all the areas.

10.8.2 Second version

The only difference with the preceding one is a curried binding of the I/O functions with the channels as soon as it is possible.

This example used only 70 calls to the GC, most of them (52 vs 18) due to a lack of space in the HEAP. About 35 seconds were saved on the GC time, 30 more on the function call and binding time.

10.8.3 Third version

Here we use partial evaluation as well, but rather than inputing and outputing character by character, we shall use the functions on lines. The code is slightly more complicated since one has to carefully distinguish the areas between "%". This code is as follows:

```
let remove_comment_2 infile outfile =
 let inp = open_in infile
 and outp = open_out outfile
 and (from_filter, to_writer) = new_pipe () in
 let write = output (broadcast [outp ; std_out])
 and get () = input_line inp
 and give = output to_writer
 and take () = input_line from_filter
 and flag = ref true in
 let rec filter fstring =
  (let pos_pc = pos_string fstring "%" in
   if (pos_pc = (-1)) then (if !flag then give fstring)
   else (if !flag then give (sub_string fstring 0 pos_pc);
         flag := not !flag;
         filter (sub_string fstring (pos_pc + 1)
                    ((length_string fstring)
                            - (succ pos_pc)))))
and writer () =
 let line = get () in
  (write (take ());
   if not (line = "") then (filter line; writer ()))
in
 filter (get ());writer ();
 close_in inp;close_out outp;close_in from_filter;;
```

```
#gc true;;
<<(gc 85 0 0 0 0 0 92 5 cons 24320 symbol 2348 string 7886 vector 2383
   float 1012 fix 1024 heap (39) code (217))>>
Runtime: 3.45s

#remove_comment_2 "./aux/comments" "./aux/foo2";;
.................................................
Runtime: 26.33s

#gc_info ();;
<<(gc 85 0 0 0 0 0 97 5 cons 24192 symbol 2348 string 7878 vector 2380
   float 1012 fix 1024 heap (7) code (217))>>
```

The runtime amount needed is drastically reduced, since this version runs more than 18 times faster than the first one !

One may estimate the total size of the strings built since all the GC calls are due to a lack of space in the HEAP: It is about 230 Kbytes (5 complete GC and it remains only 7 Kbytes).

10.8.4 Fourth version

In order to built less strings, we use bufferised input (function `buf_input`). Moreover this allows us to input 256 characters at each read operation, rather than a line of at most 80 characters. Then the new code is:

```
let remove_comment_3 infile outfile =
 let inp = open_in infile
 and outp = open_out outfile
 and (from_filter, to_writer) = new_pipe ()
 and buf = make_string 256 " "  in
 let write = output (broadcast [outp ; std_out])
 and get () = try buf_input inp buf 256 ; buf
              with ml_eof n -> sub_string buf 0 n
 and give = output to_writer
 and take () = input from_filter 256
 and flag = ref true in
 let rec filter fstring =
  (let pos_pc = pos_string fstring "%" in
    if (pos_pc = (-1)) then (if !flag then give fstring)
    else (if !flag then give (sub_string fstring 0 pos_pc);
          flag := not !flag;
          filter (sub_string fstring (pos_pc + 1)
                             ((length_string fstring)
                              - (succ pos_pc)))))
and writer () =
```

```
  write (take ());
  if not (end_of_channel inp) then (filter (get ()); writer ())
 in
 filter (get ());writer ();
 close_in inp;close_out outp;close_in from_filter;;

#gc true;;
<<(gc 85 0 0 0 0 0 97 6 cons 24320 symbol 2348 string 7883 vector 2381
   float 1012 fix 1024 heap (39) code (217))>>
Runtime: 3.44s

#remove_comment_3 "./aux/comments" "./aux/foo3";;
................................................
Runtime: 10.44s

#gc_info ();;
<<(gc 85 0 0 0 0 0 99 6 cons 24192 symbol 2348 string 7875 vector 2378
   float 1012 fix 1024 heap (36) code (217))>>
```

This version needs only about 80 Kbytes of HEAP space.

10.8.5 Fifth version

Still trying to minimize the strings allocation, we use more sophisticated functions on strings to handle the CAML buffer. This avoids many calls to the function sub_string which builds a new string each time it is called.

```
let remove_comment_4 infile outfile =
 let inp = open_in infile
 and outp = open_out outfile
 and (from_filter, to_writer) = new_pipe ()
 and buf = make_string 256 " "  in
 let write = output (broadcast [outp ; std_out])
 and get () = try buf_input inp buf 256 ; buf
              with ml_eof n -> sub_string buf 0 n
 and give = output to_writer
 and take () = input from_filter 256
 and flag = ref true in
 let rec filter fstring = (filtrec
  where rec filtrec n =
   (let pos_pc = index_string fstring "%" n in
     if (pos_pc = (-1)) then
      (if !flag then give (sub_string fstring n 256))
       else (if !flag then
              give (extract_string fstring n (pred pos_pc));
```

```
              flag := not !flag;
              filtrec (succ pos_pc))))
 and writer () =
  write (take ());
  if not(end_of_channel inp) then (filter (get());writer())
 in
 filter (get ()) 0;writer ();
 close_in inp;close_out outp;close_in from_filter;;

#gc true;;
<<(gc 85 0 0 0 0 0 99 7 cons 24320 symbol 2348 string 7880 vector 2379
   float 1012 fix 1024 heap (39) code (217))>>
Runtime: 3.44s

#remove_comment_4 "./aux/comments" "./aux/foo4";;
................................................
Runtime: 7.05s

#gc_info ();;
<<(gc 85 0 0 0 0 0 100 7 cons 24192 symbol 2348 string 7870 vector 2376
   float 1012 fix 1024 heap (1) code (217))>>
```

This saved 4 Kbytes but it needs only one GC (we had some luck !!)

10.8.6 Sixth version

The saving of the HEAP space in this example is rather surprising. It is due to a feature of the buffers for pipes. These are lists of strings of a given size (default size is 256 characters). As soon as one of these strings is full, another is added to the list, in order to receive the next input. In the preceding version we used an input buffer of 256 characters. Each time it contained no % it was totally given to the pipe: hence a second string was added in the buffer of the pipe. After having taken the content of the buffer the list was reset to a single string, and this was repeated many times. Hence in this last version we use a CAML input buffer of only 255 characters which never makes full the first string of the buffer !! This is the only difference with the preceding version.

Now we exactly saved 23 Kbytes of HEAP space !

10.8.7 Seventh version

If one only wants to copy the file without the comments, one may use a simpler and more efficient function (without pipes), for example:

```
let remove_comment_6 infile outfile =
 let inp = open_in infile
 and outp = open_out outfile
```

```
 and buf = make_string 256 " "  in
 let write = output outp
 and get () = try buf_input inp buf 256 ; buf
              with ml_eof n -> sub_string buf 0 n
 and flag = ref true in
 let rec filter fstring = (filtrec
  where rec filtrec n =
   (let pos_pc = index_string fstring "%" n in
     if (pos_pc = (-1)) then
      (if !flag then
        write (if n=0 then fstring
               else (sub_string fstring n 256));
       if not (end_of_channel inp) then filter (get ()) 0)
       else (if !flag then
              write (extract_string fstring n (pred pos_pc));
              flag := not !flag;
              filtrec (succ pos_pc))))
 in
 filter (get ()) 0;
 close_in inp;close_out outp;;
```

The results are as follows:

```
#gc true;;
<<(gc 85 0 0 0 0 0 101 9 cons 24192 symbol 2347 string 7870 vector 2376
   float 1012 fix 1024 heap (39) code (215))>>
Runtime: 3.84s

#remove_comments_6 "./aux/comments" "./aux/foo";;
.................................................
Runtime: 3.18s

#gc_info ();;
<<(gc 85 0 0 0 0 0 101 9 cons 24320 symbol 2347 string 7870 vector 2376
   float 1012 fix 1024 heap (20) code (215))>>
```

One can see that only 19 Kbytes of HEAP were used.

10.8.8 Eighth version

Here we use the CAML library package automat. The program is then simpler:

```
let treat_comments = make_automaton
 [do_until_char "%" copy; do_on_char ignore;
  do_thru_char "%" ignore];;
```

```
let remove_comments infile outfile = treat_comments
 (open_in infile,(broadcast [open_out outfile;std_out]));;
```

Now the results become:

```
#gc true;;
<<(gc 85 0 0 0 0 0 101 10 cons 23424 symbol 2348 string 7771 vector 2279
   float 1012 fix 1024 heap (34) code (207))>>
 : obj
Runtime: 3.50s

#remove_comments "./aux/comments" "./aux/foo7";;
................................................
() : void
Runtime: 3.11s

#gc_info ();;
<<(gc 85 0 0 0 0 0 101 10 cons 23424 symbol 2348 string 7771 vector 2279
   float 1012 fix 1024 heap (8) code (207))>>
 : obj
```

For this version, no GC is needed during computation, but it uses a little heap space more than preceding one ... but code is considerably simplified and clearer, while performances are maintained at a high level of efficiency ...

10.8.9 Ninth version

This one is your own ! If it is more efficient than the preceding one please report it, everybody will benefit from your programming effort !

10.8.10 Conclusion

With a careful glance at our I/O routines (or using the automat package), we have speeded up the computation more than one hundred times. So please don't report on the "extremely slow" performances of CAML I/Os without trying to rewrite your program with efficient primitives ...

Chapter 11

Formatting Functions

The CAML system provides general facilities to write pretty-printing programs. In this chapter we describe them and all other questions related to printing and formatting of output.

Be aware that there are 3 kinds of printing routines, which are listed below ordered by computational expenses they require:

- "display" functions are basic primitives to print CAML objects. They do not format their arguments.
- "echo" functions are a bit more sophisticated, since they break automatically lines when the input spreads over a limit which can be set by the user.
- "print" functions are the formatting functions of the system: they are very powerful but, since they involve a complex formatting algorithm, they are rather slow.

Thus, if you need to print very large output in a trivial way, use "display" versions of the printing primitives.

11.1 Common printing functions

11.1.1 Displaying text

♣
```
display_bool : (bool -> unit)
display_num : (num -> unit)
display_string : (string -> unit)
```

These are all functions which return `()`, with the side effect of printing their argument exactly as they would be printed at toplevel, except that `display_string` does not print the surrounding string quotes ".

♣
```
display_newline : (unit -> unit)
display_flush : (unit -> unit)
display_message : (string -> unit)
```

- `display_newline` outputs a newline and `display_flush` is used to flush the output (for instance when asking for some interactive answer, to ensure that the question is effectively displayed on the terminal).
- `display_message` is used to print a message i.e. a string followed by a newline. As one might think it is equivalent to:

```
let display_message s =
    display_string s;display_newline();;
```

11.1.2 Elementary formatting primitives

When the preceding functions are not powerful enough (namely when too long lines are not advisable), you may use the following primitives which allow very simple formatting, limited to line-breaking as soon as `echo_margin` characters have been printed with one of the above printing functions. Very often such a formatting behaviour is sufficient and you need not to use the more complex (and much more computational time consuming) primitives of the CAML pretty-printer.

The "echo" versions of the above printing functions have a similar behaviour to their "display" correspondant when "echoing" is not set up by the primitive `open_echo`: for instance

```
close_echo();echo_num 1;;
```

is equivalent to

```
display_num 1;;
```

♣
```
open_echo : (unit -> unit)
close_echo : (unit -> unit)
echo_break : (num * num -> unit)
set_echo_margin : (num -> num)
```

- `open_echo` and `close_echo` enable and prevent the "echo" line breaking mechanism. They should be nested and balanced like parentheses.
- `echo_break (n,m)` has the following effect: if the current line is not broken `n` spaces are printed, otherwise `m` spaces are printed at the beginning of the line (`m` may be `0`).

- `set_echo_margin` allows you to fix the maximum number of characters printed on a line (default 77, maximum value 255).

♣
```
echo_bool : (bool -> unit)
echo_num : (num -> unit)
echo_string : (string -> unit)
```

These are "echo" versions of the former "display" primitives to print booleans numbers and strings.

♣
```
echo_newline : (unit -> unit)
echo_flush : (unit -> unit)
echo_message : (string -> unit)
pn_echo_message : (string -> unit)
```

- `echo_newline`, `echo_message` and `echo_flush` are analoguous to the "display" versions.
- `pn_echo_message` echoes a message preceded by a newline.

```
#open_echo();;
() : unit

#set_echo_margin 55;;
55 : num

#let echo_word w = echo_string w;echo_break(1,0);;
Value echo_word = <fun> : (string -> unit)

#let echo_phrase = do_list echo_word;;
Value echo_phrase = <fun> : (string list -> unit)

#echo_phrase
#   ["A fundamental article";"on the";
#    "Categorical";"Abstract";"Machine";"is";":"];
#echo_newline();
#echo_phrase
#   ["Guy Cousineau,";"Pierre-Louis Curien,";"Michel Mauny,"];
#echo_newline();
#echo_phrase
#   ["The Categorical";"Abstract";"Machine";
#    "Functionnal Programming";"Languages";"and Computer";
```

```
#    "Architecture,";"Ed. J.P. Jouannaud,";
#    "LNCS 201";"Springer-Verlag";"1985."];
#echo_newline();
#close_echo();;
A fundamental article on the Categorical Abstract
Machine is :
Guy Cousineau, Pierre-Louis Curien, Michel Mauny,
The Categorical Abstract Machine
Functionnal Programming Languages and Computer
Architecture, Ed. J.P. Jouannaud, LNCS 201
Springer-Verlag 1985.
() : unit
```

11.2 Formatting primitives

11.2.1 Enqueuing text

All the text printed by the CAML system goes through a pretty-printer, controllable by the user. The pretty-printer queues the text until it determines where line breaks will be needed; at any point, the queue may hold up to three lines of text. The text consists of nested blocks; a block that cannot fit on one line is broken in designated places. All the following functions are used to queue text in the pretty-printer queue. They must be used with the formatting primitives, in order to obtain their real power: in all other cases they are equivalent to their echo version.

♣
```
print_bool : (bool -> unit)
print_num : (num -> unit)
print_string : (string -> unit)
print_int : (int -> unit)
print_float : (float -> unit)
print : (dyn -> unit)
print_dyn : (dyn -> unit)
print_string_for_read : (string -> unit)
print_quoted_string : (string -> string -> unit)
print_caml_syntax : (MLsyntax -> unit)
```

These are primitives to print values of basic data types. The function `print_string_for_read` prints strings with reexpansion of escape characters.

Beware:

-- The `print` and `echo` versions must not be confused: both print their argument but, for example, `echo_num` is the ordinary printing function for numbers while `print_num` does much more work, since its number argument is *not* displayed *at once*, because it is enqueued. Thus, in case of channel operations, the user may be

confused because the pretty-printer queue may be cleared later than he thought it would be.

```
#print_num 1;output std_out "2";;
21() : void
```

In this example, pretty-printer queue was cleared by the CAML toplevel since it is systematically cleared when the evaluation of a CAML phrase ends, but it was cleared *after* having written 2 into `std_out` !
You may try as well:

```
quit (print_string "The CAML pretty-printer is very lazy !");;
```

and compare with:

```
quit (echo_string "The CAML pretty-printer is very lazy !");;
```

11.2.2 Formatting boxes

♣
```
open_hbox : (num -> unit)
open_vbox : (num -> unit)
open_hvbox : (num -> unit)
open_hovbox : (num -> unit)
close_box : (unit -> unit)
print_break : (num * num -> unit)
print_cut : (unit -> unit)
print_space : (unit -> unit)
```

The pretty-printing uses the concept of "boxes", inside which text is printed. Pretty-printer breaks lines automatically, where the user indicates that lines may be broken. A box is one of the following:

- vertical box (`vbox`): every break signalled by the user systematically leads to a new line.
- horizontal box (`hbox`): pretty-printer prints this box on a single line.
- horizontal-vertical box (`hvbox`): the box is displayed on a single line if it is possible, else it is broken at every break-point signalled by the user.
- horizontal or vertical box (`hovbox`): the box is displayed on a single line if it is possible, else it is still broken on break-points indicated by the user, but pretty-printer tries to display as much material as it is possible on the same line (i.e. number of new-lines is minimized).

The primitives used with formatting boxes are:

- `print_break(width,offset)` tells the pretty-printer that the line may be broken here, **if necessary**. If the line is broken, then `offset` is added to the current indentation; otherwise, `width` determines the amount of blank space inserted horizontally.

- `close_box ()` signals the end of a box.

- `open_hbox offset` signals the beginning of a box that should be printed on a single line. If this is not possible the end of the box is rejected at left margin with `offset` as indentation.

- `open_vbox offset` signals the beginning of a box that should be broken at every break-point (with `offset` added to the current indentation).

- `open_hvbox offset` signals the beginning of a box that should be broken consistently. That is to say that if the box cannot fit on a single line, it will be broken anywhere a break has been set. The argument of **open_hvbox** is the number of spaces (`offset`) added to the indentation of any lines broken inside the box.

- `open_hovbox offset` signals the beginning of a box that should be broken inconsistently. That is to say that if the box cannot fit on a single line, it will be broken into as few lines as possible. The argument of **open_hovbox** is the number of spaces (`offset`) added to the indentation of any lines broken inside the box.

- `print_cut` is quivalent to `print_break(0,0)` and `print_space` to `print_break(1,0)`.

```
#(* Box with null offset *)
#open_hovbox 0;print_string "CAM";
#print_space();print_string "is an acronym of";
#print_space();print_string "Categorical";
#print_space();print_string "Abstract";
#print_space();print_string "Machine,";
#print_space();print_string "a model for the evaluation";
#print_space();print_string "of functional languages,";
#print_space();print_string "based upon the";
#print_space();print_string "Categorical Combinatory Logic";
#print_space();print_string "(from P.L. Curien).";
#close_box();;
CAM is an acronym of Categorical Abstract Machine,
a model for the evaluation of functional languages,
based upon the Categorical Combinatory Logic
(from P.L. Curien).() : unit

#(* Non null offset *)
```

```
#open_hovbox 2;print_string "CAML";
#print_space();print_string "is a functional language,";
#print_space();print_string "whose compilation is based";
#print_space();print_string "upon the";
#print_space();print_string "Categorical";
#print_space();print_string "Abstract";
#print_space();print_string "Machine,";
#print_space();print_string "implemented first by";
#print_space();print_string "G. Cousineau.";
#close_box();;
CAML is a functional language, whose compilation is based
  upon the Categorical Abstract Machine,
  implemented first by G. Cousineau.() : unit
```

Notice that calls to `open_*box` (where * stands for any kind of boxes) and `close_box` should nest like brackets. When writing a recursive printing routine, you may include `open_hovbox 0` as its first command and `close_box ()` as its last.

11.2.3 Flushing the pretty-printer queue

♣
```
print_flush : (unit -> unit)
print_newline : (unit -> unit)
force_newline : (unit -> unit)
```

- `print_flush ()` prints all the queued text.
- `print_newline ()` prints all the queued text but followed by a carriage return.
- `force_newline ()` tells the pretty-printer to add a carriage return inside a formatting box, where it normally has no reason to introduce one: in this case the box remains open and the pretty-printer records that some more room is available into the new line added.

Notice that `print_flush` and `print_newline` close *all* boxes currently active. So it may be the last command of a printing routine, to be sure that you close all the boxes you have open. Nevertheless the CAML toplevel ends with a `print_newline` command, in order to print all queued text remaining in the queue: this generally ensures that your text will be completely displayed if you are printing on the terminal (in case of pretty-printer output redirection this is no more ensured).

Be aware that `print_newline` is not "recursively secure", since it closes all boxes:

```
#let game_over () = print_string"Game over!";print_newline();;
Value game_over = <fun> : (unit -> unit)

#let bye () =
# open_vbox 0;
#  print_string"Now boy";print_cut();
#  open_hovbox 0;
#   print_string"It's the end:";print_space();
#   game_over();
#   print_space();
#   print_string"See you soon";
#  close_box();
# close_box()
#;;
Value bye = <fun> : (unit -> unit)

#bye();;
Now boy
It's the end: Game over!
 See you soon
Formatting Failed: Attempt to close the last box
```

The problem is that all the routines which use a function which performs a `print_newline` has to be careful about it! This is almost untractable in practice if every printing routine may execute a `print_newline`. In particular this is forbidden in user's defined printers used by the CAML system.

11.2.4 Fooling printing primitives

♣
```
print_as_null : (string -> unit)
print_as : (num -> string -> unit)
echo_as : (num -> string -> unit)
echo_as_null : (string -> unit)
```

These are used to "fool" the pretty-printer: one may precise the length which have to be taken into account by the pretty-printer when printing a given string.

```
#let print_with_decor s =
#    print_as_null "{\em ";print_string s;
#    print_as_null"}";;
Value print_with_decor = <fun> : (string -> unit)
```

This an easy way to send control characters to a terminal to get a fancy output while keeping proper indentation.

11.2.5 Controlling output extension

Number of active boxes

♣
```
max_depth : (unit -> num)
max_print_depth : (num -> num)
limit_depth : (unit -> num)
limit_print_depth : (num -> num)
full_print : (('a -> 'b) -> 'a -> 'b)
print_with_max_depth : (num -> ('a -> 'b) -> 'a -> 'b)
```

There are two limits to box nesting extension: `max_print_depth` is the first limit the ("normal" one), beside which boxes may be open but all the text contained in them is omitted. The second limit (`limit_print_depth`) is crucial: the pretty-printer presumes that your printing routine is looping and thus it aborts.

The maximum depth of box nesting, set to a default value of 35, is reported by `max_depth` and may be reset by the command `max_print_depth`. When the current depth exceeds the maximum depth, the content of the box is ignored and is replaced by a single character: the ellipsis character (default ".").

When `max_print_depth` limit is exceeded, the default number of boxes you can still open is 20. Beside this limit the pretty-printer prints " ...", clears the queue and fails with exception `format` and value `"Limit print depth exceeded"`. This limit is reported by `limit_depth`, and may be reset too, by the command `limit_print_depth`. Notice that it is impossible to set `limit_print_depth` to be less than `max_print_depth`.

```
#max_print_depth 5;limit_print_depth 10;;
10 : num

#let rec looping () =
#open_hovbox 2;
#print_string"Is \"Camel\"";print_space();
#open_hovbox 2;print_string"the name of an animal";
#print_space();
#open_hovbox 2;print_string"or a programming language ?";
#looping();;
Value looping = <fun> : (unit -> 'a)

#looping();;
Is "Camel"
  the name of an animal
    or a programming language ?Is "Camel"
                                ..... ...

Formatting Failed: limit_print_depth exceeded
```

`print_with_max_depth` and `full_print` are used to call a printing routine with a temporarily fixed (resp. unlimited) number of boxes allowed.

Ellipsis

♣
```
ellipsis : (unit -> string)
set_ellipsis : (string -> string)
print_ellipsis : (unit -> bool)
set_print_ellipsis : (bool -> bool)
```

The current ellipsis is returned by `ellipsis ()`, it is set by `set_ellipsis` and may be completely omitted (except once, for the level equal to `max_depth`) if `print_ellipsis false` is evaluated. This is useful when you want to provide different views of an object displayed by some formatting routine: setting the ellipsis to, say "?", and evaluating `print_ellipsis false`, you may fit the amount of material displayed using `max_print_depth`.

```
#let pretty_message () =
# let print_word s = print_string s; print_space() in
# open_hovbox 0;print_word"The CAML system";
#  print_word"is due to";
#  open_hovbox 0;
#   open_hovbox 0;
#   print_word"the programming efforts of";
#    open_hovbox 0;print_word"the Formel project";
#     open_hovbox 0;print_word"at INRIA,";close_box();
#    print_word"and mainly";
#    open_hovbox 0;print_word"to";close_box();
#    close_box();
#   close_box();
#  open_hovbox 0;print_word"A.";close_box();
# close_box();
# close_box();
# open_hovbox 0;print_string"Suarez";print_break(0,0);
# print_string ".";
# close_box();
#print_newline();;
Value pretty_message = <fun> : (unit -> unit)

#set_ellipsis"";set_print_ellipsis false;;
false : bool

#max_print_depth 3;pretty_message();;
```

```
The CAML system is due to Suarez.
() : unit

#max_print_depth 4;pretty_message();;
The CAML system is due to the programming efforts of A.
Suarez.
() : unit

#max_print_depth 5;pretty_message();;
The CAML system is due to
the programming efforts of the Formel project and mainly
A. Suarez.
() : unit

#max_print_depth 6;pretty_message();;
The CAML system is due to
the programming efforts of
the Formel project at INRIA, and mainly to A. Suarez.
() : unit

#(* resetting limits to default values *)
#set_ellipsis".";set_print_ellipsis true;;
true : bool

#limit_print_depth 55;max_print_depth 35;;
35 : num
```

The preceeding example is evidently (only) a case study, since boxes are mainly used in recursive routines which automatically open more or less boxes depending on the complexity of their argument. For instance the pretty-printer of a programming language may be able with these features to hide more or less the complexity of a program.

11.2.6 Margin and indentation

♣

```
set_margin : (num -> unit)
set_max_indent : (num -> unit)
```

The margin and the maximum indentation limit have default value 72 and 62. These functions set them to whatever (between 21 and 255).

11.3 Printing auxiliaries

11.3.1 Messages

♣ `message : (string -> unit)`
`pn_message : (string -> unit)`
`pn_print_string : (string -> unit)`

Used to print a message these functions are equivalent to:

```
let message s = print_string s; print_newline();;
let pn_message s = print_newline();message s;;
let pn_print_string s = print_newline();print_string s;;
```

11.4 Formatters

The set of primitives described above define the behaviour of the "generic formatter" or "generic pretty-printer".

An instance of the generic pretty-printer devoted to a particular channel is called a formatter. There may be several active formatter at the same time.

Each formatter consists into a channel associated with a state of the generic pretty-printer, and a "kind" of the formatter. A kind is just the registrement of "who" is printing with the formatter: it may be either the "system" or the "user" or the "error" routines, or a combination of these. For instance, the default situation when entering the core image is to have only one formatter writing to `std_out`, and this formatter possesses the three attributes as its "kind", since everything printed by the system goes into `std_out`.

When a formatter associated with a given mode is (ordered by the user) to be closed, then the mode automatically reverts to another formatter devoted to the same mode if it exists, else it reverts to `std_out` and a warning is emitted.

When closing an `out_channel`, a check up is performed to flush and suppress the associated formatter if any, and to revert the mode to another formatter (if there is none the mode is set up to a formatter onto `std_out` as well).

All the primitives described so far where intended to print on any formatter.

11.4.1 Printing on error formatter

♣ `error_print_string : (string -> unit)`
`error_message : (string -> unit)`
`error_pn_message : (string -> unit)`

These are analogues to the above primitives on current user's formatter, but they directly use the error formatter.

11.4.2 Opening and closing formatters

One opens or closes a formatter with:

```
♣ open_user_print : (out_channel -> unit)
  close_user_print : (out_channel -> unit)
  open_error_print : (out_channel -> unit)
  close_error_print : (out_channel -> unit)
  open_system_print : (out_channel -> unit)
  close_system_print : (out_channel -> unit)
  all_open_print : (out_channel -> unit)
  all_close_print : (out_channel -> unit)
```

The `all_*` versions opens or closes formatters for the three modes.

One redirects the output of printing routines (respectively for user, system and error) using:

```
♣ user_print_on : (out_channel -> unit)
  error_print_on : (out_channel -> unit)
  system_print_on : (out_channel -> unit)
  all_print_on : (out_channel -> unit)
```

thus you may simultaneously use severall formatters on severall streams and switch easily from one to another, dispatching the output of *the same printing routines* you wrote (may be interactively in debugging mode).

11.5 Redirecting the output of the pretty-printer

When you write a printing routine, it is very convenient to test it using output on the terminal, but as soon as it works fine, you may then need to redirect its output to a file. In CAML this is simply done by assigning a different stream to the pretty-printer output using

```
♣ user_print_to : (out_channel -> unit)
  error_print_to : (out_channel -> unit)
  system_print_to : (out_channel -> unit)
  all_print_to : (out_channel -> unit)
```

The three first functions redirect the output from the user, from error and from the system, the last redirects all the output.

For example to print into the file "toto":

```
let toto = open_out "toto";;
all_print_to toto;;
```

To revert to the standard state, evaluate:

```
all_print_to std_out;;
close_out toto;;
```

These "echo" versions will redirect the output as well but will also echo to the terminal:

♣
```
user_echo_to : (out_channel -> unit)
error_echo_to : (out_channel -> unit)
system_echo_to : (out_channel -> unit)
all_echo_to : (out_channel -> unit)
end_all_echo_to : (unit -> unit)
```

Chapter 12

Pretty-printing Grammar

The pretty-printing package provides a grammar "Pretty" predefined as "autoload" in the CAML system[1] to write with great facility printing orders and pretty-printers.

12.1 Syntax for pretty-printing definition

12.1.1 Syntax definition conventions

The metalanguage used to specify the syntax is based on the BNF. The meta-symbols have the following meaning:

::=	:	is defined as,
|	:	or,
[x]	:	0 or 1 occurrence of x
(x) *	:	0 or more occurrences of x,
(x) +	:	1 or many occurrences of x,
(x | y)	:	x or y,
<*Name*>	:	the non-terminal symbol Name
"xyz"	:	the keyword xyz

The syntaxes may use any of the CAML lexical units (`Ident`, `String`, `Bool`, `Infix`, `Num`, `Int` and `Float`).

<*Caml expression0*> designates a CAML expression (entry point: Expr0 in the CAML grammar) which is a basic lexical unit or a parenthesised expression.

12.1.2 "Pretty" syntax

<*Syntax*> ::=

[1] Thus there is a normal delay to wait after the first call to this grammar.

```
    <Printer>
  | <Sequence>
  | "let rec" <Pattern> "=" <Printer> "in" <Syntax>
  | "with" <Printer> "do" <Syntax> "done"

<Sequence> ::=
    <Printing order> ( ";" <Printing order> ) *

<Printing order> ::=
    "\\"
  | [ <NUM> ] "-"
  | "space" [ <IDENT> |"(" <Caml expression> ")" ]
  | "\-"
  | "\"
  | "\(" <Caml expression> ")"
  | [ "Literal" |"LITERAL" ] <STRING>
  | <Object>
  | <Local Printer> <Object>
  | "{^" <Caml expression> "^}"
  | "with" <Printer> "do" <Sequence> "done"
  | "(" "let" "rec" <Pattern> "=" <Printer> "in" <Sequence> ")"
  | "if" <Caml expression>  "then" <Printing order>  [ "else" <Printing order> ]
  | "[" <Box Name> <Sequence> "]"

<Printer> ::=
    "print" <Pattern> "->" <Sequence> ( "|" <Pattern> "->" <Sequence> ) *
  | Local printer

<Local Printer> ::=
    "(" <Printer> ")"
  | <IDENT>
  | "NUM" |"Num"
  | "STRING" |"String"
  | "IDENT" |"Ident"
  | "FLOAT" |"Float"
  | "{" <Caml expression> "}"
  | "*" <Local Printer>
  | "+" <Local Printer>
  | "\*" <Local Printer>
  | "\+" <Local Printer>

<Object> ::=
    <IDENT>
  | "{" <Caml expression> "}"

<Box Name> ::=
```

```
    "<" "hov"  [ "(" <Caml expression> ")" |<Num> ] ">"
  | "<" "hv"   [ "(" <Caml expression> ")" |<Num> ] ">"
  | "<" "v"    [ "(" <Caml expression> ")" |<Num> ] ">"
  | "<" "h"    [ "(" <Caml expression> ")" |<Num> ] ">"
```

12.2 Entries of grammar Pretty

This grammar is a macro: it builds up a CAML expression which is evaluated in the current environment. There are two entries:

- "Prog": this one returns an object of type ML. This entry is useful to look at the CAML sentences built by the grammar.
- "Exec": this one returns the same object, which is evaluated in the current environment where it is included. It's the default entry.

The first entry can be simulated by parsing the second one:

```
#<:Pretty:Prog< Num n ;- ; "boys"; \\ >>;;
<:Caml:Expr<print_num n;
            print_string " ";
            print_string "boys";force_newline ()>> :
 ML
```

Thus using the CAML programs pretty-printer (pretty function):

```
#pretty();;
#<:Pretty< Num n; -; "boys"; \\>>;;
print_num n;
print_string " ";print_string "boys";force_newline ()
() : unit

#fun n -> <:Pretty< Num n ;- ; "boys"; \\ >>;;
<fun> : (num -> unit)

#it 5;;
5 boys
() : unit
```

12.3 Pretty grammar structures

The first basic structure is the printing orders sequence, where orders are separated by a semicolon. Each one provides a printing effect and they are sequentially executed. The orders are:

- printing orders
- control orders
- printing boxes

The other structure is the printer's definition, which builds a printing function.

12.4 Printing orders

12.4.1 Simple orders

- "-": prints a space .
- <*NUM*> "-": print n spaces.
- "space" followed by an argument (Identifier or Caml expression between parenthesis) prints a number of spaces equal to this argument.
- "\\": forces a newline.
- <*STRING*> (optionnaly preceded by the keyword "`Literal`" (or "`LITERAL`")): prints this string with *print_string*.
- "`{^`" <*Caml expression*> "`^}`": executes this expression (it's an escape from pretty-printing grammar to CAML).

```
#<:Pretty:Prog<"hello"; - 4;Literal "I'm an happy";
#           - (2 + 2);"taxpayer";\\;{^4^}>>;;
<:Caml:Expr<print_string "hello";
            print_string (make_string 4 " ");
            print_string "I'm an happy";
            print_string (make_string (2+2) " ");
            print_string "taxpayer";force_newline ();4>> :
 ML

#<:Pretty<"hello"; - 4;"I'm an happy";
#           - (2 + 2);"taxpayer";\\;{^4^}>>;;
hello    I'm an happy    taxpayer
4 : num
```

12.4.2 Binary orders

When you want to print *the value* of a variable or a CAML expression, the proper order is a <*Printer*> followed by:

- <*IDENT*>: prints with the previous printer the value bound to this identifier in the current environment,

- "{^" *<Caml expression>* "^}": prints with the previous printer the result of the evaluation of this expression.

The printer is optional. If there is no printer, the value is printed with the default printer (*print_string*).

```
#<:Pretty:Prog<Num x; print_bool { x=1 }; y; \\>>;;
<:Caml:Expr<print_num x;
            print_bool (x = 1);
            print_string y;force_newline ()>> :
 ML

#fun x y -> <:Pretty<Num x;-; print_bool { x=1 }; -; y; \\>>;;
<fun> : (num -> string -> unit)

#it 2 "hello";;
2 false hello
() : unit
```

12.5 Printer

12.5.1 Syntax of printers

A printer may be designated by:

- an *<IDENT>*: the printer is the function bound to this Identifier.
- a "{" *<Caml expression>* "}": the printer is the function returned by the evaluation of this expression.
- keyword "Num" or "NUM": the printer is the function *print_num*.
- keyword "Float" or "FLOAT": the printer is the function *print_float*.
- keyword "String" or "STRING": the printer is the function *print_string*.
- keyword "Ident" or "IDENT": the printer is the function *print_ident*.
- a local definition.
- an iterator applied to another printer.

The printer local definitions look like CAML functions definitions except that:

- the keyword "print" (or "printer") is used instead of the keyword "function",
- printing order sequences replace Caml expressions in right-hand sides of matching rules.

```
#fun x ->
# <:Pretty<"the representation is";-;
#       (print true -> Num {1}
#            | false -> Num {0}) x>>;;
<fun> : (bool -> unit)

#it true;;
the representation is 1() : unit
```

12.5.2 Iterators

A printer can be an iterator applied to another printer. Iterators are:

- "*": applies the printer to each element of a list (= *do_list*).
- "*: applies the printer to each element of a list in reverse order.
- "+": applies the printer to the first element of a pair and to each element of the list, which is the second component of the pair.
- "\+": same meaning but in reverse order.

```
#<:Pretty:Prog< * Num l; \+ Float l'>>;;
<:Caml:Expr<do_list print_num l;
            (fun f (x,l) -> do_list f (rev l);f x)
             print_float l'>> :
 ML

#fun l -> <:Pretty<"["; + (print x -> Num x; -) l; "]"; \\>>;;
<fun> : (num * num list -> unit)

#it (1,[2;3;4]);;
[1 2 3 4 ]
() : unit
```

12.6 Control Orders

A sequence element can be one of control orders.

12.6.1 Default printer

The default printer is *print_string*, but we can define another default printer with the order "with" <*Printer*> "do" <*Sequence*> "done".

```
#<:Pretty:Prog< with Num do x; y; "is" ; z done>>;;
<:Caml:Expr<print_num x;
```

```
            print_num y;print_string "is";print_num z>> :
 ML

#fun x y z ->
# <:Pretty<with (print x -> Float x; -) do x; y done; z>>;;
<fun> : (float -> float -> string -> unit)
```

12.6.2 Local affectation and recursive definition

If a complex printer is often needed or if we want to define many recursive printers, we can use: "`let`" "`rec`" <*Pattern*> "`::=`" <*Printer defined with keyword print*> "`in`" <*Sequence*>.

```
#type Calcul =
#       Num of num
#     | Plus of Calcul & Calcul
#;;
Type Calcul defined
     Num : (num -> Calcul)
   | Plus : (Calcul * Calcul -> Calcul)

#let addition e1 e2 =
#<:Pretty<
# let rec print_expr = print
#     Num n -> Num n
#   | Plus (x,y) -> print_expr x; " + "; print_expr y in
#
# "("; print_expr e1; ") + ("; print_expr e2; ") = ";\\;
# print_expr {Plus (e1,e2)}; \\
#>>;;
Value addition = <fun> : (Calcul -> Calcul -> unit)

#addition
# (Plus(Plus (Num 1,Plus(Num 3,Num 0)),Plus(Num 1,Num 5)))
# (Plus(Num 6, Plus (Num 7, Num 8)));;
(1 + 3 + 0 + 1 + 5) + (6 + 7 + 8) =
1 + 3 + 0 + 1 + 5 + 6 + 7 + 8
() : unit
```

12.6.3 Conditional test

The sentence "`if`" "`then`" "`else`" is available for conditionally printing orders.

```
#let plural b s =
# <:Pretty< String s;
```

```
#           if b=0 or b=1 then () else "s";\\>>;;
Value plural = <fun> : (num -> string -> unit)

#plural 3 "show";plural 1 "show";;
shows
show
() : unit
```

The sentence "begin" "if" ... "then" ... ["else" ...] "end" "if" is used to have a sequence instead of a single order in the actions.

12.7 Printing Boxes

The concept of formatting boxes (see preceding chapter) is used to describe the concrete layout of patterns: a box may contains many objects which are orders or sub-boxes sequences separated by break-points; the box wraps around them an imaginary rectangle.

12.7.1 Box types

The type of boxes specifies the way the components of the box will be displayed and may be one of the following kinds:

- "h": to catenate objects horizontally.
- "v": to catenate objects vertically.
- "hv": to catenate objects as with an "h box" but an automatic vertical folding is applied when the horizontal composition does not fit into the width of the associated output device.
- "hov": to catenate objects horizontally but if the horizontal composition does not fit, a vertical composition will be applied, trying catenate horizontally as many objects as possible.

This kind can be followed by an offset value, which is the offset added to the current indentation when breaking lines inside the box.

12.7.2 Boxes syntax

A box is described by a sequence surrounded by "[...]". The first element of the sequence (optional) is the box kind: this kind surrounded by the symbols "<>" is one of the keywords "h", "hv", "v", "hov" followed by the optional offset (a *<Num>* or *<CAML expression>* between parentheses).

The default offset is 0 and the default kind is hov. Thus the default box is <hov 0>.

12.7.3 Break-points

In order to specify where the pretty-printer is allowed to break, one of the following break-points may be used:

- "\" (`print_cut()`): simple break-point if the line is not broken here, no space is included.
- "\-" (`print_space()`): if the line is broken the composition is the same as with "\", otherwise a blank space (" ") is included here.
- "\(i,j)" (`print_break(i,j)`): if the line is broken, the value i is added to the current indentation for the following one; otherwise j blank spaces are inserted.

Break-points are inserted in the sentence as other printing orders (between semicolons). "\" and "\-" are identical to "\(0,0)" and "(1,0)". The optional parenthesised CAML expression expected after the break-point "\", should return a $(num * num)$ typed value.

Break points are useless in a "h box" and break point "" is identical to "\" in a "v box".

```
#let x = "******" in
#<:Pretty< [<v> x ; \(0,2); x; \(0,4); x;
#           \(0,2); x; \; x] ; \\>>;;
******
  ******
    ******
  ******
******
() : unit

#<:Pretty< - 10;
#[<hov (-5)> "this"; \-; "text"; \-; "is"; \-; "printed"; \-;
# "in"; \-; "a"; \-; "paragraph"; \-; "as"; \-; "well"; \-;
# "as"; \-; "possible"; \-; "the"; \-; "first"; \-; "line"; \-;
# "has"; \-; "an"; \-; "indentation"; \-; "equal"; \-;
# "to"; \-; "10"; \-; "and"; \-; "the"; \-; "next"; \-;
# "ones"; \-; "an"; \-; "indentation"; \-; "equal"; \-;
# "to"; \-; "5."; \-; "the"; \-; "last"; \-; "word"; \-;
# "is"; \-; "printed"; \-; "with"; \-; "a"; \-; "global"; \-;
# "indentation"; \-; "equal"; \-; "to"; \-;
# "40."; \(100,40); "End"; \-];\\>>;;
         this text is printed in a paragraph as well as
    possible the first line has an indentation equal to
    10 and the next ones an indentation equal to 5. the
```

```
     last word is printed with a global indentation equal
     to
     40.
                                                  End

() : unit

#type expr =
#      Cond of (expr*expr)*expr*expr
#    | Var of string
#    | Op of expr*expr ;;
Type expr defined
     Cond : ((expr * expr) * expr * expr -> expr)
   | Var : (string -> expr)
   | Op : (expr * expr -> expr)

#let rec print_expr = <:Pretty<
# print Var v -> v
#     | Op (x,y) -> [<hv 2> print_expr x;-;"+";\-;print_expr y]
#     | Cond ((c1,c2),t,e) ->
#       [<hv 1>
#        [<hov 2>
#         "if";\-;
#          print_expr c1; -; "="; \(1,2);
#          print_expr c2]; \-;
#        [<hov 2> "then"; \-; print_expr e]; \-;
#        [<hov 2> "else"; \-; print_expr e]
#       ]
#>>;;
Value print_expr = <fun> : (expr -> unit)

#print_expr
# (Op (Var "variable_1",
#      Cond
#      ((Cond
#        ((Var "variable_2",
#          Op(Var "variable_3",Var "variable_4")),
#          Op (Var "variable_5",Var "variable_6"),
#              Var "variable_7"),
#              Var "variable_8"),
#          Op (Var "variable_9",Var "variable_9"),
#            Cond
#            ((Var "variable_10",Var "variable_11"),
#              Op(Var "variable_12",Op(Var "variable_13",
```

```
#                                        Var "variable_14")),
#              Op(Var "variable_16",Var "variable_17")))));;
variable_1 +
  if
    if variable_2 = variable_3 + variable_4
     then variable_7
     else variable_7 = variable_8
   then
     if variable_10 = variable_11
      then variable_16 + variable_17
      else variable_16 + variable_17
   else
     if variable_10 = variable_11
      then variable_16 + variable_17
      else variable_16 + variable_17() : unit
```

12.8 Pretty and Grammar

The Pretty syntax is very similar to grammar syntax. In fact the analogy is enhanced by the possibility to define a "toplevel" printer: syntactically a printer looks like a grammar (with rules and entries) but is introduced by "`printer`" instead of "`grammar`".

Thus when we have to define a pretty-printer for an object language, we can work from the grammar with the toplevel syntax.

12.8.1 Pretty-printers toplevel syntax

<*Printer syntax*>	::=	"`printer`" <*Ident*> "=" [<*Delimitor*>] <*Rules list*>
<*Delimitor*>	::=	"`delimitor`" <*String*>
<*Rules list*>	::=	"`rule`" <*Rule*> ("`and`" <*Rule*>) *
<*Rule*>	::=	["`entry`"] <*Ident*> "=" <*Printer*>

12.8.2 Use of Pretty-printer toplevel syntax

This syntax is a macro for recursive functions declaration; each rule is a printing function definition, but only rules designated bu the keyword "`entry`" will be global functions.

Another keyword for printer is provided : "`quoted_ string`"; it prints a string between delimitors defined after the keyword "`delimitor`". The default delimitor is ".

12.8.3 Example

As an example we define a pretty-printer for λ-calculus:

```
#(* The abstract syntax *)
#type lambda =
#    Var of string
#  | App of (lambda & lambda)
#  | Let of (string & lambda & lambda)
#  | Abs of (string & lambda)
#;;
Type lambda defined
     Var : (string -> lambda)
   | App : (lambda * lambda -> lambda)
   | Let : (string * lambda * lambda -> lambda)
   | Abs : (string * lambda -> lambda)

#(* The concrete syntax *)
#grammar for values Lambda =
#rule entry Lam =
#    parse Literal "let"; IDENT x; "="; Lam M;
#          Literal "in"; Lam N -> Let(x,M,N)
#        | Literal "\\"; IDENT x; Literal "."; Lam M
#           -> Abs(x,M)
#        | Lam1 M -> M
#
#and Lam0 =
#    parse IDENT x -> Var x
#        | Literal "("; Lam M; Literal ")" -> M
#
#and Lam1 =
#    parse Lam1 M; Lam0 N -> App(M,N)
#        | Lam0 M -> M
#;;
Calling Yacc ... ..........................................
Value Lambda = <fun> : (string -> Parsers)
Grammar Lambda for values defined
  entry Lam : lambda

#(* The pretty printer *)
#printer Lambda =
#rule entry Lam =
#    print
#      Let (x,M,N)
#         -> [<hov>Literal "let"; \-; IDENT x; \-;
#                  Literal "="; \-; Lam M; \-;
#                  Literal "in"; \-; Lam N]
#    | Abs (x,M)
```

```
#         -> [<hov>Literal "\\"; \-; IDENT x; \-;
#                 Literal "."; \-; Lam M]
#     | M  -> [<hov>Lam1 M]
#
#and Lam0=
#     print Var x -> [<hov>IDENT x]
#         | M -> [<hov>Literal "("; \-; Lam M; \-; Literal ")"]
#
#and Lam1=
#     print App (M,N) -> [<hov>Lam1 M; \-; Lam0 N]
#         | M -> [<hov>Lam0 M]
#;;
Value Lam = <fun> : (lambda -> unit)
```

Now we can use our pretty-printer to print abstract syntax trees:

```
#Lam <<let x = y in \x.x>>;;
let x = y in \ x . x() : unit

#(* Notice the ''automatic'' management
#   of useless parentheses *)
#Lam << (((f g) h) x) y>>;;
f g h x y() : unit
```

If we define Lam as a user's defined printing function usable by the CAML toplevel printing routines we get:

```
#(* We define Lam as a toplevel printer, usable
#   by the CAML system with the directive printer *)
##printer Lam;;
New printer defined for type: lambda
() : unit

#<<let x = y in \x.((t z) (x y z))>>;;
let x = y in \ x . t z ( x y z ) : lambda
```

Chapter 13

Directives and Pragmas

Directives and Pragmas are commands given to the toplevel environment. For example, they can be used to set up compilation modes or to define new infix identifiers.

We first describe the different kinds of CAML toplevel loops, and then give the difference between directives, pragmas and usual toplevel expressions.

13.1 Toplevel modes

The CAML toplevel can be in one of the following modes:

- interactive mode
- code loading mode
- compiling mode
- pseudo-loading mode

The interactive mode is the toplevel loop you get when beginning a session or loading a source file.

The code loading mode is entered when loading a code file.

In both interactive and code loading modes, expressions are evaluated and the symbol table is updated with the values of symbols. The first one inputs CAML syntax, the second one inputs code.

The compiling mode is a mode where toplevel expressions are compiled but not evaluated: the code is not executed. Only types and exceptions declarations are evaluated, and the symbol table is temporarily updated, but no value is assigned to symbols. Once one gets out of the compilation mode, no trace of the compilation is left in the symbol table. Such a loop is entered when compiling a source file: code is written in the compiled file.

The pseudo-loading mode is a mode where only types and exceptions declarations are executed. Code of expressions or other declarations is neither loaded nor

executed. As in compilation mode, the symbol table is temporarily updated, but no value is assigned to symbols. This mode is entered by the system, for example when compiling a file which uses another file: the file used is not really loaded (but pseudo-loaded): just what is useful to be able to compile the file which contains the use directive is loaded.

The compiling mode is to the interactive mode what the pseudo-loading mode is to the code loading mode.

A directive is a toplevel command to be executed whatever the current mode is. It is a kind of *eval-when load compile* facility. A pragma is a command to be executed only in compilation mode (*eval-when compile*).

Directives and Pragmas are syntactically distinguished from usual toplevel expressions by starting with #. Following a #, directive and pragma expressions follow essentially the same syntax as CAML phrases enriched by some new keywords (standing as abbreviations for the most usual commands).

Directives and Pragmas have their own symbol tables. Values declared as directives (resp. pragmas) are stored in the directive (resp. pragma) table; evaluation of directive expressions (resp. pragma) are (roughly speaking) done in that table.

Since the `directive` and `pragma` keywords may be followed by any legal CAML phrase, there is a notion of environment attached to directives and pragmas. Pragmas and directives together with their corresponding environments make sense in compilation and loading modes, they are only extended to interactive mode for consistency's sake.

First of all, one should keep in mind that when a toplevel loop asks for compiling a file, the CAML system pushes another toplevel loop which will takes its input from the file and will be popped when the compilation will terminate.

The directive and pragma environments should be seen as extensions of the *calling* environment, extensions being valid only during the compilating (and loading, in case of directives) phase.

The directive environment is *linked* to the system environment (the environment you get when you call the caml command) in the following way: when typing an expression, free variables are searched first in the directive environment, and then in the system environment. So, the directive environment is on top of the system environment.

The pragma environment is on top of the directive environment, in the same way.

These environments exist in interactive mode, and some new ones are allocated as soon as you compile a file. In this case, a new directive environment is pushed on top of directive environments, a new pragma environment is pushed on top of the old one which is itself pushed on top of the new directive environment.

13.2 Syntax of directives and pragmas

The syntax of directives uses the keyword `directive`, followed by any expression:

```
#directive use "another_file";;
```

is an example of directive meaning that what will follow uses a file named "another_file", expressing their dependency.

In any mode, such dependencies have to be taken in account.

There are predefined directives and pragmas, with the following syntax:

<Directives_or_Pragmas> ::= "#" *<Command>* ";;"

```
 <Command>   ::= "directive" <declaration or expression>
               | "pragma" <declaration or expression>
               | "use" <string>
               | "load" <string>
               | "compile" <string>
               | "infix" <identifier>
               | "uninfix" <identifier>
               | "open" "compilation" <boolean>
               | "open" "overloading" <boolean>
               | "open" "printing" <boolean>
               | "open" "module" <boolean>
               | "open" "optimization" <number>
               | "fast" "arith" <boolean>
               | "default" "grammar"
               | "set" "default" "grammar" <syntax>
               | "eval" "when" "print" <boolean>
               | "printer" <identifier>
               | "sharing" "printer" "for" "type" <type>
               | "default" "printer" "for" "type" <type>
               | "quit"
;
<constraint list>  ::= <identifier> ":" <type>
                     | <constraint list> "and" <constraint list>
;
<syntax>           ::= <identifier>
                     | <identifier> ":" <identifier>
;
```

13.3 Semantics of directives and pragmas

The `#directive ...;;` and `#pragma ...;;` constructs cover all the other cases not listed above.

Other constructs are directives (except the ones concerning printers, which have the same behaviour as usual toplevel commands). They usually correspond to a function call, and calling that function as a pragma may be useful (particularly functions concerning compilation modes and infixes). Many functions modifying

the compiling environment (`load`, `use` or `autoload`) are secure when used as directives but are not as pragmas!

The `#use ...;;` construct calls the `use` facility described in the section about "*Using files*". Note that `use` is available as a function in the directive environment.

```
##directive use;;
Directive <fun> : (string -> unit)
```

The `#load ...;;` (resp. `#compile ...;;`) constructs call the `load` (resp. `compile`) function on their argument which must be a file name.

The `#infix ...;;` construct calls the `infix` function. One should use `#pragma infix ...;;` if the identifier has to be treated as an infix only locally.

The `#uninfix ...;;` construct undoes what has been done by an `infix` directive.

The `#fast arith ...;;` directive tells the compiler that from now on, it has to generate tests as least as possible. For instance in this mode, arithmetic is supposed to be done with "small" integers and addition is translated into a single machine instruction. When accessing vectors or strings tests for range are ellipsed as well.

The `fast_arith` flag is global, so it can be sometimes hard to guess the real state of the system. The `compile_fast` function should be easier to use.

The `#open optimization ...;;` directive tells the compiler to optimize the compiled code:

- Optimization level 0 is no optimizations at all (and is a safe mode).
- Optimization level 1 is as `fast_arith true` (and thus may be unsafe when accessing outside vectors or strings).
- Optimization level 2 (or equivalently more than 2) optimizes as much as possible. It is *experimental and not secure*: in addition with arithmetic and accesses, recursive function calls are optimized.
- Negative optimization level is equivalent to level 0.

The `#open compilation ...;;` directive ellipses tests concerning the building of recursive objects (which now does not lead to compilation errors). This mode is thus unsafe, since the generated code may be uncorrect.

The `#open overloading ...;;` directive tells the typechecker not to reject CAML phrases which present completely ambigous parts. It uses instead a default strategy to give a type to these ambigous parts which is generally the last declared type compatible with the typechecking constraints (more exactly: among those types which allow the resolution of the first ambigous type constraint, the last declared type is chosen. Then typecheking goes on, and the same process is applied for every overloading ambiguity that may appear).

The `#set default grammar ...;;` construct specifies what will be the default object syntax used in the sequel. This is again a global value, and may lead to the same kind of problems as the `fast_arith` facility.

The `#default grammar;;` construct returns the name of the default object syntax and its default entry point.

The `#open printing ...;;` directive allows the system to print values of abstract types as much as possible, even if it involves to print labels or constructors out of scope.

The `#open module ...;;` directive tells the system to export all the values and types of a module regardless of its export part. This is only used for debugging.

The `#eval when print ...;;` directive is used to automatically force lazy values when printing them.

The `#printer ... for ...;;` construct defines a function as a new *printer* for objects of its domain with a syntax name (which may be unspecified, giving the appearance of a default syntax).

The `#default printer for type ...;;` removes a (maybe) existing printer on the specified type, or tells the system to print objects of the type as it usually does.

The `#sharing printer for type ...;;` tells the CAML system to show the sharing of structures of a given type.

```
#type 'a shared = Nil_shared | Shared of 'a * 'a shared;;
Type shared defined
     Nil_shared : 'a shared
   | Shared : ('a * 'a shared -> 'a shared)

##open compilation true;;
Directive () : unit

#let rec x = Shared (1,x);;
Warning: unsafe recursive declaration
Value x =
      (Shared
       (1,
        (Shared
         (1,
          (Shared
           (1,
            (Shared
             (1,
              (Shared
               (1,
                (Shared
                 (1,
                  (Shared
                   (1,
                    (Shared
```

```
                    (1,
                     (Shared
                      (1,
                       (Shared
                        (1,
                         (Shared
                          (1,
                           (Shared
                            (1,
                             (Shared
                              (1,
                               (Shared
                                (1,
                                 (Shared
                                  (1,
                                   (Shared
                                    (1,
                                     (Shared
                                      (1,
                                       (Shared
                                        (..................... ...

limit_print_depth exceeded
 : num shared

##sharing printer for type num shared;;
() : unit

#x;;
shared0
 where shared0 = (Shared (1,shared0)) : num shared

#(* The sharing is specific to a type instance *)
#let rec y = Shared ("foo",y);;
Warning: unsafe recursive declaration
Value y =
      (Shared
       ("foo",
        (Shared
         ("foo",
          (Shared
           ("foo",
            (Shared
             ("foo",
```

```
(Shared
 ("foo",
  (Shared
   ("foo",
    (Shared
     ("foo",
      (Shared
       ("foo",
        (Shared
         ("foo",
          (Shared
           ("foo",
            (Shared
             ("foo",
              (Shared
               ("foo",
                (Shared
                 ("foo",
                  (Shared
                   ("foo",
                    (Shared
                     (..................... ...

limit_print_depth exceeded
 : string shared
```

13.4 Autoload

13.4.1 Autoload functions

It is very convenient to "pseudo" define a function needing a large piece of code, as an "autoload" identifier. The code will be loaded from a (compiled) file, automatically and silently, at the first invocation to the function. This evidently spares code zone of the memory, if the function is never used.

Numerous functions of the CAML system are defined "autoload" (including the `trace` function), that is why you may think that some of them run very slowly: in fact you wait the time needed to load their corresponding files.

In order to guarantee that compilation and loading of files containing "autoloading" declarations will behave properly, `autoload` needs to declare symbols at compile time, and then it must be executed as a directive. Actually, autoload

is a syntatic construction belonging to the CAML directive's meta-language. Its informal syntax is:
"autoload" <*function_constraints_list*> "from" <*filename*>
where <*filename*> is a string denoting a filename:
<*filename*> ::= <*Caml expression*>
and <*function_constraint_list*> is given by:
<*function_constraint_list*> ::= <*function_name*> ":" <*type*> | <*function_constraint_list*> "and" <*function_constraint_list*>

autoload takes as last argument a file name, and complains if it does not exist or has no corresponding code file. The first argument is a list of type constraints of functions declared autoload.

```
#autoload toto : num -> num
#     and tata : num -> num -> void
#from "foo";;
Warning: (autoload) foo.lo not found
Forward tata : (num -> num -> unit)
Forward toto : (num -> num)
Directive () : unit

#toto;;
<fun> : (num -> num)

#tata;;
<fun> : (num -> num -> unit)
```

declares the toto and tata functions to be autoloaded from the file "foo.lo".

Beware:

-- Autoload call fails if:

- if one of the function constraints does not correspond to a functional types.

-- If a file is loaded, an (autoloaded) function call fails if

- the function called is not defined in file.
- the type of the function loaded from the file is different from the function definition type.
- the file has already been loaded by the user.

13.4.2 Autoload grammars

The autoload principle is extended to grammars. The syntax is as follows:
"autoload" "grammar" Ident "with" <*Entries*> from <*filename*>
<*Entries*> ::= <*Entry*> | <*Entry*> "and" <*Entries*>
<*Entry*> ::= String "at" "entry" String

<filename> ::= *<Caml expression>*
For instance:

```
#autoload grammar foo
#with parse_expr at entry Expr
# and parse_decl at entry Decl
#from "foobar";;
Warning: (autoload) foobar.lo not found
Forward parse_decl : (unit -> ML)
Forward parse_expr : (unit -> ML)
Grammar foo (autoload) defined
   with entries Expr Decl
Directive () : unit
```

Chapter 14

Loading, Compiling and Saving

CAML code can be put in a file exactly in the same way as you would type it at the system toplevel. Such a *source file* can be either loaded at the current toplevel, or compiled to get a *code file* which can be loaded afterwards.

At toplevel or while loading source code files, CAML phrases are compiled, loaded and finally executed. Loading compiled code files allows to skip the first phase, which is about ten times faster than loading source file versions.

In this chapter we start by examining how the CAML system knows where to look for files. Next we describe the different kinds of toplevels in CAML and how to exploit them, as well as primitives for loading and compiling files, for saving sessions and doing separate compilation. Finally we describe the *very useful* command **use**: do not skip this section !

14.1 Where to look for files

Some of the commands in this chapter take as arguments strings which are interpreted as filenames or directory names following some of the UNIX standard conventions. In particular the symbols "." and ".." are interpreted as in UNIX when appearing in strings denoting filenames or directory names (but "~" is not, it is taken as a directory with name "~").

14.1.1 Search paths

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

The CAML system looks up for files according to the value of a search path variable. This variable contains a string list representing the list of directory names for the host operating system. According to its initial value, files are first searched

in the user's current working directory (the directory from which the CAML system was called), and then successively in each directory in the path. The default search path also contains the system directory where the CAML system resides. The function `search_path` returns the current value of the search path. The function `add_path` adds a directory in front of the search path, and the function `delete_path` deletes a directory name from the search path. Both functions return the resulting path.

14.1.2 Basic directory manipulation

♣
```
cd : (string -> string)
pwd : (unit -> string)
home_dir : (unit -> string)
```

It is possible to ask for the home directory with the command `home_dir` and for the current working directory with `pwd`. The last one can be changed with `cd`.

```
#search_path();;
["/usr/local/caml/V2-6.1/doc/manual";
 "/usr/local/caml/V2-6.1/lib"] : string list

#add_path "..";;
["/usr/local/caml/V2-6.1/doc/manual";
 "/usr/local/caml/V2-6.1/doc"; "/usr/local/caml/V2-6.1/lib"] :
 string list

#pwd();;
"/usr/local/caml/V2-6.1/doc/manual" : string

#home_dir();;
"/home/beaune/formel/weis" : string
```

14.1.3 System directories

♣
```
system_directory : string
lib_directory : string
```

These global identifiers are bound respectively to the names of the directories where the CAML system files and CAML library files reside.

14.1.4 Finding files

Finding names of files

♣ `abs_path : (string -> string)`
`base_name : (string -> string)`
`dir_of : (string -> string)`

- `abs_path` normalizes a file name in the UNIX way.
- `base_name`, given a file name finds the base name in it without the access path.
- `dir_of` given a filename, normalizes it and gives as result the access path to the `base_name`.

```
#abs_path"../../doc/manual/aux/rm_comments0.ml";;
"/usr/local/caml/V2-6.1/doc/manual/aux/rm_comments0.ml" :
 string

#base_name it;;
"rm_comments0.ml" : string

#dir_of it;;
"/usr/local/caml/V2-6.1/doc/manual/" : string
```

Testing existence of files

♣ `find_file : (string -> string)`
`find_ml_file : (string -> string)`

`find_file f` returns the absolute path of the first file matching the string `f` in the search path (or fails with `"file not found"`).

`find_ml_file` performs the same operation as the previous command for CAML files.

Testing anteriority of files

♣ `newest : (string -> string -> string)`

`newest f1 f2` returns the last modified file when both exist, one of them if only that one exist, otherwise fails (with `"files not found"`).

Operating on files

♣ `mv : (string -> string -> unit)`
`rm : (string -> unit)`

- `rm f` unlinks the file `f`.
- `mv f1 f2` renames (moves) the file named `f1` into the file `f2`.

14.2 Loading files

It is possible to load CAML code from a file, rather than to type it interactively at toplevel. A source CAML file has extension `.ml` whereas a compiled file has extension `.lo`.

14.2.1 The primitive "load"

This function is the most commonly used for loading code.

♣ `load : (string -> unit)`

`load <string>` reads and evaluates CAML phrases from the input file (if it is a source file) exactly as if they had been typed by the user at toplevel. Its argument can be any filename to which the extension `.ml` is appended (with failure for unacceptable filenames and for non-existent files).

Beware:

-- In order to always behave as one would expect, `load` does some magic:

Suppose there is a file named `"toto.ml"` and its compiled version `"toto.lo"` in the current directory. Then the command `load "toto"` will load the compiled file `"toto.lo"`, if it is *newer* than `"toto.ml"`, otherwise it will load the source version `"toto.ml"`.

Thus as soon as a file has been compiled (and if the source code is not modified afterwards) `load` loads its compiled version.

Nevertheless it is possible to force 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 load the code version (if it exists !).

Notice that `load"toto"` searches for the file `"toto"` in the whole `search_path`, and that as soon as a file `"toto.ml"` has been found `load` searches, *in the same directory*, for a file `"toto.lo"`: if it is newer it is loaded, otherwise the source file `"toto.ml"` is loaded. This is designed to ensure that a file `"toto.lo"`, staying far away in the search path will not be loaded instead of a source file closer, even if the source file is older ...

If any failure occurs during loading, the call of `load` fails without the rest of the file being read. Calls of `load` can be nested, i.e. a file being input by `load` can itself contain calls of `load`.

14.2.2 Other loading primitives

♣ `loadt : (string -> unit)`
`loadf : (string -> unit)`

Commands `loadt` and `loadf` behave exactly as `load` except that the printing of information during loading is not affected by the current print mode setting of the CAML toplevel. With `loadt` results are always printed at toplevel even if echo information for types and values have been set to false. With `loadf` a single period is printed for every toplevel phrase successfully read and evaluated. This save computation time (about 10 to 20 percent). These two functions look up for files along the search path.

♣ `lib_load : (string -> unit)`

This function allows loading packages from the CAML library.

14.2.3 System loading primitives

♣ `loadc : (string -> unit)`
`loads : (string -> unit)`

- `loadc` loads a compiled file whose absolute name is provided (no search is done) and to which extension `.lo` is appended. `loadc"toto"` performs the same action as `load"toto.lo"`.
- `loads` loads always source files.

14.3 Compiling files

Toplevel CAML phrases are always compiled on the fly. A CAML file can be compiled and the result put in an object code file.

Separate compilation of interdependent files can be achieved with the module mechanism described below. Nevertheless if you chose to compile related files without modules, a minimal set of preaution must be taken. First of all, and because of the static binding, files must be compiled in turn, and of course, loaded

in the same order. More generally the environment of compilation and loading of each file must be the same, to ensure a correct behaviour. Finally, since no type checking is done during execution, when a file B depends from a file A that has been recompiled, is advisible to recompile B to ensure that it is correctly typed with respect to the new version of A.

The command `use` described below can be very useful in dealing with sets of files.

14.3.1 The primitive "compile"

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

The conventions are the same as for the `load` command, as far as search paths and file names are concerned.

- `compile`: a CAML source file is compiled, saved, and then loaded and executed. The compiled code (placed in the same directory as the source file) may be loaded afterwards, by one of the loading commands (e.g. `load` or directive `use`).

- `compil`: a CAML source file is compiled (without further loading nor execution) and the compiled code is placed in the same directory as the source file. This command is useful to compile modules in files when the interface environment is not ready to their loading. Aditionally, it prevents from overflow in the code zone when compiling big systems.

14.3.2 Other compilation primitives

♣
```
compilet : (string -> unit)
compilef : (string -> unit)
compile_fast : (string -> unit)
```

`compilet` and `compilef` print toplevel results in the same way as `loadt` and `loadf` respectively do.

`compile_fast` compiles files in `fast_arith` mode `true`: i.e. the compiler is allowed to optimize arithmetic operations (and some other operations involving numbers, strings and vectors).

14.3.3 System compilation primitives

These primitives are rough and very basic. They are not so smoothly protected from user's errors and are not as user friendly as the previous primitives. In fact, they are not intended for normal use of the CAML system, but are provided to sophisticated users to built large sub-systems.

♣ `compil_fast : (string -> unit)`

`compil_fast` : compile files as `compile_fast` above but without further loading. It do not perform any search according to the current search path (its file name argument is supposed to exist)

14.4 Using files

`use` is a very useful directive for maintaining programs built up from many files. It provides the minimal capacities to build systems with both automatic updating of compiled code files when necessary, and recursive expression of file dependencies.

Since the same behaviour must be guaranteed at compilation and at loading time, `use` is a directive. At loading time `#use` has the same effect as `load`, except that the file is loaded only if it has not been already loaded during the session (so several `#use` commands for the same file are strictly equivalent to just one).

If the file is not compiled, or if it is older than its source version, then `#use` *automatically recompiles* it, and then loads its compiled version. In this case `#use` displays a warning and compilation is performed as usual, in all others cases `#use` works silently.

At compile time a `#use "toto"` in the file `titi` ensures that the file `"toto"`, if not already loaded, will be "pseudo" loaded: i.e. all the declarations in the file `"toto"` will be taken into account before the beginning of the compilation of the file `"titi"` (cf. directive description). Pseudo loading is faster than loading and is just what the compiler needs to generate the compiled code.

It is good practice to begin every CAML file by use directives, expressing the files that are needed for execution of the present one: this allows automatic maintenance of compatibility between source and compiled versions of your files and ensures that proper environment exists for the present file (remember that CAML uses static binding), both at compile and at loading time.

Beware:

-- Be careful to describe files dependency "as flat" as possible: i.e. put all the `use` directives for all the files of your system in the main module. Otherwise the CAML system may fail to recompile your system since it needs to open a channel for each compilation currently active, and because there are about 8 channels available to compile, the dependency tree of a system must have a depth less than 8.

Example:

```
(* Main : file main.ml *)
#use "titi";;
#use "tata";;
#use "toto";;

(* There is the rest of file main.ml *)

(* File titi.ml does not use anything *)

(* File tata.ml *)
#use "titi";;
(*rest of the file tata.ml *)

(* File toto.ml *)
#use "tata";;
(* rest of toto.ml *)
```

In this case the CAML system will open at most 2 channels to compile the entire system, since files "titi.ml", "tata.ml" and "toto.ml" are compiled *in turn*, while compiling "main.ml".

If we suppose that the two directives #use "titi" and #use "tata" are removed from the head of "main.ml", "main.ml" becomes:

```
(* Main : file main.ml *)
#use "toto";;

(* There rest of file main.ml *)
```

For the user point of view this is perfectly equivalent to the previous version and this set of files will be correctly handled by the CAML system.

But the compiler must now open 4 channels *simultaneously*: one to compile "main.ml", then a second channel for the file "toto.ml", which leads it to open a third, to start the compilation of "tata.ml", in which the use directive obliges the compiler now to open a fourth channel more for the file "titi.ml".

In case of really large systems, with many files, a bad dependency design may lead the number of files *simultaneously* compiled to exceed the CAML system capabilities.

14.5 Saving and reentering core images

♣ `reset : (unit -> unit)`

The primitive `reset` quits the currently running CAML image and calls a fresh one. This is useful when the environment is corrupted ...

♣ `save_image : (unit -> unit)`
`save_caml_image : (string * string -> unit)`
`restore_image : (string -> unit)`

A core image of the current session is dumped with the command `save_image`. This allows saving the complete state of the CAML session as a core image, which may be reloaded later with `restore_image` command. When evaluated `save_image ()` asks for 2 strings: the first string value 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.

```
save_image();;
Core name ? toto
Banner ? Mon corps
() : void
```

will store core image `"toto.core"` and then you get:

```
restore_image "toto";;

Mon corps

#
```

You can then ask CAML with the option `-r` to directly restore your core image. For instance a typical unix command to call the application whose core image is `toto.core` would be:

```
caml -r toto.core
```

The function `save_caml_image` is a non interactive version of `save_image`: it may be used in "make" files, when CAML reads from standard input.

Beware:
-- core images tend to be wasteful of storage. They should be limited to subsystems.
-- It is not advisable to put a `save_image` command in a file, since this disturbs the CAML toplevel streams: when the core image is saved a channel is open to the

file, and when it is restored the CAML toplevel will try to close it ! Nevertheless the toplevel will restore more or less the situation, but it will make a violent protest against this poor programming behaviour ... So, please, use this command *interactively* at toplevel.

Chapter 15

Separate Compilation

CAML provides a very simple system of modules, whose principal aim is to ensure separate compilation and to facilitate the organization of large programs into separate units.

Each module is composed of three parts: import interface, body and export interface. The import interface serves to make the module into an independent unit that can be compiled separately: it specifies the type information necessary to compile the module body. The body is composed of a sequence of CAML phrases. The export interface specifies the body components which will be available at top-level after linking the module.

Through its interfaces, we can regard a module as an isolated environment which communicates with its enclosing one: importing information about objects it needs, and exporting some of its components to make them available in the environment where they will be used. To deal with a large system, we split it into modules that can be compiled in isolation. Afterwards we link these modules and incrementally build the environment corresponding to the whole program. In the following sections, we will show how to define and use modules in CAML.

15.1 Syntax of Modules

An informal syntax for modules is described by the following grammar. The syntactic categories are written in italics.

```
<module>  ::=  "module" <ident> [ <import> ] ";;"
                 <body>
               "end" "module" [ <export> ] ";;"
```

```
<import>      ::=  "using" <imp_specs> [ "from" <string_list> ] ";;"
<imp_specs>   ::=  "type" <ty_spec> [ ";" <imp_specs> ]
                 | "value" <val_spec> [ ";" <imp_specs> ]
                 | "exception" <exc_spec> [ ";" <imp_specs> ]
<val_spec>    ::=  <ident> ":" <ty> [ "and" <val_spec> ]
<ty_spec>     ::=  <ident> [ <args> ]
                 | <ty_decl>
<exc_spec>    ::=  <exc_decl>

<export>      ::=  "with" <exp_specs>
<exp_specs>   ::=  "type" <ident> ( "and" <ident> ) *
                 | "exception" <ident> ( "and" <ident> ) *
                 | "value" <ident> ( "and" <ident> ) *

body          ::=  sequence of CAML phrases
```

Note that this syntax does not allow modules to be nested. In the following example we define a module which imports a type and a function defining an order relation over it. It implements functions for ordered insertion and for sorting lists of the imported type. As result, the sorting function is exported.

```
module Order
 using  type elem;
        value  leq : elem & elem -> bool;;
  let rec insert x =
    fun []     -> [x]
     | (y::l) -> if leq(x,y) then x::y::l
                 else y::(insert x l);;
  let rec sort =
    fun []     -> []
     | (x::l) -> insert x (sort l);;
end module
 with value sort;;
```

15.2 Using Modules

We use modules in CAML to compile separately programs and to build systems out of these parts. In their utilisation two phases can be distinguished: compiling and linking. Compilation of a module defines a new typecheking environment composed of a standard prelude augmented with the module's import specification. Linking forces the match of the module's import specification with the current environment. In case of success the body and the current environment are linked and the top-level is increased with the declarations listed in the export. We shall examine below these two phases in detail.

15.3 Compiling Modules

A module in CAML is always compiled separetely. As first step the import specification is typechecked in a standard prelude environment composed only of predefined primitives and types. The body typechecking and compilation proceed in this prelude augmented with the import specification, and once this is done the legality of the export specification (when it exists) is verified. Intuitively, an export specification is legal if it specifies only symbols local to the body and depending only on types in the import or in the export parts. Consider a module M with import specification I, body B and export E:

```
module M using I ;;
   B
end module with E ;;
```

We denote by P the standard prelude environment and by $+$ the operation of increasing an environment with another one. We say that a type τ can be exported from M if τ depends only on types in the prelude or on imported types or on types already exported, i.e., on types in $P+I+E$. We say that the export specification E is well-typed in M if for each symbol specified in E as:

- `value` v, then v is defined in B as a value with type τ and τ can be exported
- `exception` x, then x is defined in B as an exception with type τ and τ can be exported
- type t, then t has been defined in B as a type and each type involved in its definition can be exported

Note that this definition prevents from exporting symbols appearing in the import specification, which guarantees not redefining imported types when loading.

The following module defines a factorial function which needs an external function `decrm` that we forget to specify in the import interface. As result we shall not be able to typecheck the module's body.

```
#let decrm = fun x -> x-1;;
Value decrm = <fun> : (num -> num)

#module Fact;;

# let rec fact = fun 0 -> 1
#                | n -> n *(fact (decrm n));;

line 2: unbound variable decrm in fact (decrm n)
1 error in typechecking

Typecheck Failed

#end module
```

```
# with value  fact;;
Invalid export query :
unbound value fact in current module
1 error in typechecking

Typecheck Failed
```

The following is an example of illegal export specification. The value `f` depends on the local type `t` which is not exported, and the value `x` is not local to the module.

```
module Ill_export
 using value x:string;;
  type t = C of string;;
  let  f s = C s;;
end module
 with value x and f;;
```

The systems reports the error:

```
#compil"aux/ill_export";;


Type t defined
     C : (string -> t)

Value f : (string -> t)

Invalid export query :
unbound value x in current module
value  f cannot be exported :
 f of type (string -> t)
 requires type t to be exported
2 errors in typechecking

Typecheck Failed

Evaluation Failed: compile_source /usr/local/caml/V2-6.1/doc/manual/aux/ill_export.ml
```

A module can be written interactively or put in a file to be compiled. In the latter case the command `compil` can be used to produce a code file which can be loaded afterwards. This is the right command to use when we really want to do separate compilation, if for instance, we want to link only when building the whole system or if we simply do not want to matter about the current environment at each compilation step. Note the utilisation of `compil` in this last example.

15.4 Linking Modules

To link a module, we can load a compiled code file (generated by `compil`), we can write it interactively, or we can load a source file containing it. In the

former case the whole compilation takes place independently of the linking. In the two latter, compilation and linking are both done, as it happens with toplevel phrases. In both cases the matching between the current loading environment and the module's import interface is verified. An environment e matches an import specification I if for each symbol specified in I as:

- `value` x : τ, then there is a value x in e with type τ' and τ' is at least as polymorphic as τ
- `exception` x `of` τ, then there is an exception x in e with type τ' and τ' and τ are equal after expansion of all type abbreviations
- `type [` *args* `]` t = D, then there is a type t in e and if n is the number of variables in *args* and n' is the number of arguments of t in e then n and n' are equal. Additionally, if t was defined in e by D' then D and D' are syntactically identical modulo renaming of type variables and once all type abbreviation have been expanded
- `type [` *args* `]` t, then there is a type t in e and if n is the number of variables in *args* and n' is the number of arguments of t in e then n and n' are equal

The following is an example of successful loading of a file containing compiled code corresponding to the module `Order` seen previously, where the type `elem` is instantiated by integers.

```
#type elem == num;;
Type elem abbreviates num

#let leq = prefix <=;;
Value leq = <fun> : (elem * elem -> bool)

#load"aux/order";;
/usr/local/caml/V2-6.1/doc/manual/aux/order.lo loaded
() : unit

#sort;;
<fun> : (elem list -> elem list)
```

Given the module:

```
module A
 using type t = C1 of bool | C2 of string
       and  q = {L1 : bool};
       value f : 'a -> 'a
       and   g : t -> bool;;
end module;;
```

An example of a mismatching environment of loading is:

```
#type t = C1 of bool and q == bool & string;;
Type t defined
     C1 : (bool -> t)
Type q abbreviates (bool * string)

#let f x = not x;;
Value f = <fun> : (bool -> bool)

#load"aux/a";;



signature mismatch, unbound value g

signature mismatch, unbound constructor C2  in type t

signature mismatch,
current q binding must be a record type binding

signature mismatch,
f has an instance of type:
(bool -> bool)
which should match :
'a -> 'a

/usr/local/caml/V2-6.1/doc/manual/aux/a.ml loaded
() : unit
```

Anomalies:

These rules seem quite natural, but unfortunately they are not sufficient to describe the specification of the proper environment in which a module will be loaded. Because of the last one, there are cases in which an environment matches an import specification and nevertheless they cannot be linked. To understand how this can happen, we must remark that in separate compilation not only type information is extracted from an import specification but also value representations are chosen for compiling construction or destruction of values on user defined types. When matching a type declaration with an import type specification the value representation for both types must be the same.

We refer to type specifications introduced by type x as type parameters, since they can be matched by any type declaration. In the current implementation, incompatibilities of representations can occasionally arise when matching with an import containing type parameters and type specifications depending on them, as in the example:

```
module Problem
  using type q;
```

```
        type t = C1 of q | C2 of int;;
end module;;
```

Because of optimisations done in value representation, the representation for values of type t chosen with the knowledge of the representation for values of type q can be different of the one chosen without any information about q. When linking the module above with the environment containing the following declaration, the system reports a representation mismatch.

```
#type q = C of string
#and  t = C1 of q | C2 of int;;
Warning: type q redefined
Warning: type t redefined
Type q defined
     C : (string -> q)
Type t defined
     C1 : (q -> t)
   | C2 : (int -> t)

#load"aux/problem";;


signature mismatch,
representation mismatch for constructor C1 in type t
1 error in typechecking

Typecheck Failed

Evaluation Failed: load_code /usr/local/caml/V2-6.1/doc/manual/aux/problem.lo
```

We do not want to change the import match rules to take account of these problems because it would make them implementation dependent, but we do not want either to completely avoid the possibility of parametrizing a module by any type implemention. As in CAML we have found that in practice the representation mismatch errors arise rarely, we have decided to keep parameters types even if our typing rules are not exhaustive. As long as we do not have problems in linking that gives us more generic modules. A straightforward solution when a representation mismatch arises is to specify the entire description of those types appearing in types where a mismatch is reported, which means that in some cases we will have to abandon the generality to have good typechecking. In the example above, we would have to specify the implementation of type q which causes the representation mismatch:

```
module Problem
  using type q;
        type t = C1 of q | C2 of int;;
end module;;
```

We finish our description of modules with a facility for creating linking environments when loading modules. It is possible to specify in a module the environment in which it will be loaded. A list of files to be loaded before the module loading can be specified in the import part.

```
module Compiler
 using type  expr = ..
       value parser: ..
       ....       from "parser";"typecheck";;

  :
  :
end module
 with ...;;
```

The files in the implementation list must be compiled before loading and they will be loaded in the same order as they have been specified. The specified files will be loaded only if they have not been loaded before.

Chapter 16

Object Language Parsing

16.1 Introduction

16.1.1 Natural Language parsing

The general problem we face here, is to provide to the computer a way to "understand" the user's input: since input is always a string of characters which has no *a priori* meaning. Roughly speaking, the way CAML understands your input tends to mimic the natural understanding. Let us examine how we can put some meaning on the character string `The little cat is white`, supposed to be an English sentence . First we isolate the words : `The` , `little` , `cat` , `is` , `white` . This phase is called in computer litterature *lexical analysis* . Then we recognize the grammatical kind of each word in order to arrange them in groups : `is` is a verb, `little` is an adjective, `cat` a noun and `The` an article. So we find that the subject of the phrase is the group `The little cat`, its verb `is` , and that `white` qualifies `The little cat`. So, independently of sense, we can affirm now that the phrase is grammatically correct. For a computer this step is the *parsing*, which finds out that a given string is legal and does not have to be rejected. Moreover our English sentence is no more an unstructured character string, since we have got some information about each word. If we record this knowledge, it becomes something like:

```
phrase (
 verb (is),
 has_subject (group [article (The) ;
                     adjective (little);
                     noun (cat)]),
 has_object (adjective (white)))
```

This explicit tree-like form is called the *abstract syntax* of the phrase, versus the *concrete syntax*, which is the original string.

Finally we put a meaning on the abstract syntax and understand that the entire phrase means that the color of the little cat is white. For a computer this former

step is just *computing*, i.e. to give some semantics to the input.

16.1.2 CAML grammar

The same three phases (lexical analysis, parsing, computing) take place into the CAML system for each input phrase. If we consider `let x = 1;;`, first step is to recognize that `let` is a CAML keyword, `x` an identifier, `=` a symbol, `1` a number and `;;` the CAML convention to end the input.

During the second step, the parser (the program that performs parsing) discovers that entire phrase is a *declaration*, involving the *pattern* `x` and the *expression* `1`.

The abstract syntax may be given by the CAML system:

```
#parse_string "let x = 1;;\L";;
(MLdecl (MLlet ((MLvarpat "x"),(MLconst (mlnum 1))))) :
 MLsyntax
```

The final step, *understanding* or *computing*, will be to modify the toplevel state, such that the identifier `x` becomes bound to the number `1` .

16.1.3 Abstract versus Concrete syntax

The attentive reader may have noticed that strictly speaking, concrete syntax is useless to command the computer, since the computing phase starts when the abstract syntax is obtained. This is theoretically perfectly true, but practically obviously false ! Nobody has considered speaking English in abstract syntax, making explicit all his grammatical constructions, the way we did ! The point is that a great amount of implicit knowledge is hidden in our brain, which allows us to automatically translate from the concrete to the abstract English syntax. Then the input character string is an *extremely compact* coded form of the abstract syntax. This encoding is done according to these numerous implicit grammatical rules shared by all English people. The point is that without this encoding it would be virtually impossible to speak !

The same phenomena arise in CAML where the abstract syntax of a single function may have up to several thousand parentheses, which must imperatively be well balanced. No human being could write as many parentheses nor check their balance.

So, the CAML "philosophy" claims that it is absolutely essential to be able to provide the concrete syntax to human users inputting to the computer, and to let the computer synthesize by itself the abstract syntax according to some rules specified in some other computer program. As soon as complex data must be input, a concrete syntax must be defined, and the CAML system allows then the design of an (almost) arbitrary *user defined* concrete syntax.

The CAML system provides facilities to describe an abstract syntax as well (namely with concrete (!) data types). Now the aim of this chapter is to explain

how the CAML system provides a general framework to describe a concrete syntax, and to deal with automatic translation from concrete to abstract syntax.

16.2 Usage of Concrete Syntaxes

We present here the way to use concrete syntaxes in CAML, before describing how to define them. So far, we used the word "syntax" in order to emphasize on the possibility of considering either concrete or abstract form of a language's syntax. From now on, we will in general use the word "grammar" except for particular cases where a language is involved, using then "syntax". The concept of grammar is more general since the result of parsing might not be the production of abstract syntax trees but rather of arbitrary data structures.

Since CAML has this possibility of manipulating concrete syntaxes, in certain circumstances, we call it the *metalanguage* and we call *object languages* the languages corresponding to concrete syntaxes. Usage of object languages can take two forms:

- in order extend the CAML toplevel syntax: suppose you are implementing an evaluator of a prototype language. Then you surely prefer write pieces of abstract syntax of your language in "concrete" form (i.e. "`1+2`") rather than in abstract form (i.e. "`'Plus ('Const 1, 'Const 2)`").
- and the possibility of having a parsing function for your language, i.e. a function which, when called on some form of input containing concrete syntax of your prototype language (standard input, character string, a file...), returns the abstract syntax tree corresponding to that input.

Usage of parsing functions is a kind of *advanced feature* since it usually needs a certain knowledge about evaluation functions, dynamic values, toplevel loops and syntax error handling (cf section 16.6.3)..
The extension of the CAML toplevel syntax is fairly easy to use and provides readibility of programs. For example, if you have a type for arithmetic expressions, it is simpler and more readable to write "`<<1+2>>`" than "`'Plus('Const 1, 'Const 2)`". The "`<<`" and "`>>`" symbols denote an escape of the CAML syntax towards a (in fact the default one) user-defined concrete syntax.

Grammars have names and sub-grammars. The general notation is:

"`<:G:E< ... >>`"

in order to denote a phrase of the user-defined grammar "`G`" at entry point"`E`". An entry point is a name for a sub-grammar. There is a notion of default entry point, and we write:

"`<:G<... >>`"

to denote a phrase of the syntax `G` at its default entry point.

It is thus possible to have several grammars in the same CAML session, and the default grammar (called between "<<" and ">>") is usually the last defined. However, it is possible to change the default grammar by using the "**set default grammar**" directive. As an example:

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

makes "Caml" the default grammar. It is also possible to specify the entry point which has to be considered as the default one by writing something like:

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

which makes "Type" the default entry point of the "CAML" The CAML toplevel parser, when it encounters some special character sequence (such as "<<" or "<:G<") stops working and calls one of the parsers it knows in order to scan the input. When this parser stops, returning its resulting CAML abstract syntax tree, the CAML toplevel parser scans the remaining input, which must start with ">>".

Grammars may be used in CAML expressions as well as in CAML patterns. However, there are some care to take about grammar names, and grammars returning objects such as functions, cyclic or lazy data structures. See section 16.10.

16.3 Parser definitions

A parser definition has a twofold goal:

- to provide a definition of valid inputs to the parser,
- to specify the semantic values to be returned by the parser.

We call *grammar specification* the definition of valid inputs to a parser.

16.3.1 Grammar specifications

Having a language in mind, a grammar specification, consists in a specification of a context-free grammar *generating* that language. Usually, these specifications are given with a set of *production rules* containing *terminals* or *literals* (i.e. lexical entities) and *non terminals*, one of them being particularized and called the *start symbol*. An example of such a specification is:

$$
\begin{array}{lcl}
expr & \rightarrow & NUM \\
expr & \rightarrow & expr \text{ “+” } expr \\
expr & \rightarrow & \text{“(” } expr \text{ “)”}
\end{array}
$$

This grammar generates a simple language of arithmetic expressions. *NUM* is a terminal representing the lexical class of numbers and "+", "(" and ")" are literals representing the plus and parentheses symbols. The symbol *expr* is a non terminal representing the syntactic class of these arithmetic expressions. It is also the start symbol of the specification although this is not explicit.

It is easy to understand what is the language generated by this grammar specification by symbolically generating the *parse trees* of the grammar. This may be done by starting from the start symbol *expr*, and rewriting it with the rules given above oriented from left to right.

Notice that there may be several parse trees associated to a valid input. For example, in the grammar defined above, `1+2+3` could be read as `(1+2)+3` or `1+(2+3)`. Such situations are called *ambiguities* and do not arise if the parser is made *deterministic*. This is the case of all parsers produced by CAML.

16.3.2 Semantic values

One may associate *a semantic action* to each production rule of a grammar specification. A semantic action may refer to values of semantic actions associated to terminals and non terminals appearing in the left part of the production. Parsing produces then parse trees *annotated* by semantic actions, and the value returned by parser call is the value of the semantic action annotating the root node of the produced parse tree.

CAML parsers are functions, and thus must possess a functional type. Their source type is:

- either the `unit` type when they analyse input from the keyboard,
- or `string` when they take their input from strings,
- or `in_channel` when reading in an input stream.

The target type of a parser should be the type of the semantic values it produces. But, since parsers can be made "system parsers" (i.e. called by the CAML top-level parser), their target type is always the `ML` type which is the type of abstract syntax for CAML expressions.

A parser specification is a top-level construct using the keyword "`grammar`". In the following, we call such specifications *grammar definitions*.

There are two kinds of grammar definitions in CAML. The most general is called a *definition of a grammar for values*. In this case, semantic actions are executed during parsing, and their result is transformed into a trivial program (i.e. a value of type `ML`) whose evaluation returns the expected value. Such programs are built using the following constructor:

```
#MLquote;;
<fun> : (dyn -> ML)
```

where dyn is the type of *dynamic values* (cf. Chapter 8). As an example, we may evaluate[1] such a program and coerce the result into a regular CAML value:

```
#MLquote (dynamic 1);;
<:Caml:Expr<1>> : ML

#eval_syntax it;;
(dynamic (1 : num)) : dyn

#match it with dynamic (n : num) -> n
#            | _ -> failwith "not a number";;
1 : num
```

Obviously, typechecking and evaluation of such programs is a trivial task and thus is done efficiently. Definitions of grammars for values are statically typechecked.

The other kind of grammars called *grammars for programs* produce parsers returning CAML (non trivial) programs. Programs produced by such a parser need complete typechecking and evaluation. They provide a general antiquotation mechanism.

16.4 Syntax of grammar definitions

We give in this section the syntax of grammar definitions. We do not give here the description of the usage of macros inside grammar definitions: this is described at Chapter 18.

The full syntax is in fact a bit less rigid than the one given in this section. See appendix 33 for a complete description.

<Grammar declaration>
  ::= "grammar" "for" "values" *<IDENT>* "="
      <header>
      <Non terminal definitions> ";;"
    | "grammar"["for" "programs"] *<IDENT>* "="
      <header>
      <Non terminal definitions> ";;"

<header>
  ::=
    | ["delimitors" "string" "is" *<STRING>* ";"
                    "comment" "is" *<STRING>* ";"]
      ["precedences" (*<Precedence>* ";")$^+$]

<Precedence>
  ::= "right" [*<Terminal>* | *<Literal>*] $^+$
    | "left" [*<Terminal>* | *<Literal>*] $^+$
    | "nonassoc" [*<Terminal>* | *<Literal>*] $^+$
    | "precedence" *<IDENT>*

[1] See chapter 17 for an explanation about eval_syntax.

```
<Terminal>
  ::=  ("NUM" | "Num")
     | ("BOOL" | "Bool")
     | ("IDENT" | "Ident")
     | ("STRING" | "String")
     | ("INT" | "Int")
     | ("INFIX" | "Infix")
     | ("FLOAT" | "Float")
<Literal>
  ::=  ["Literal"] <STRING>
<Non terminal definitions>
  ::=  "rule" <Production rule>
       ("and" <Production rule>)*
<Production rule>
  ::=  ["entry"] (<IDENT> | "(" <IDENT> ":" <Type> ")") "=" <Rule body>
<Rule body>
  ::=  "parse" <Stream pattern> "->" <Caml expression>
       ("|" <Stream pattern> "->" <Caml expression>)*
<Stream pattern>
  ::=
     | <Binding> (";" <Binding>)*
       ["with" "precedence" [<Terminal> | <Literal> | <IDENT>]
<Binding>
  ::=  [<IDENT> | <Terminal>] <Caml Pattern0>
     | "Literal" <STRING>
     | "Literal" "(" <STRING> "as" <IDENT> ")"
     | "{" <Caml expression> "}" <Caml Pattern0>
     | "(" <Rule body> ")"
     | "(" "*" "(" <Rule body> ")" ")" <Caml Pattern0>
     | "(" "+" "(" <Rule body> ")" ")" <Caml Pattern0>
```

Here is an example of a simple grammar definition for list of numbers:

```
#grammar for values list_of_nums =
#(* This is a comment.                              *
# * There is no declaration part in this grammar. *)
#
#rule entry Main =
#    parse Literal "["; Literal "]" -> []
#        | Literal "["; Num_sequence L; Literal "]" -> L
#
#and Num_sequence =
#    parse NUM n; ( * (parse Literal ";"; NUM n -> n)) L -> n::L
#;;
Calling Yacc ... ..........................................................
Value list_of_nums = <fun> : (string -> Parsers)
Grammar list_of_nums for values defined
  entry Main : num list
```

The grammar has two non terminals: `Main` which is the entry point and `Num_sequence` which recognizes sequences of numbers seperated by semicolons. The first rule of `Main` says that the empty list must be returned when parsing "[" immediately followed by "]", and the list returned by the non terminal `Num_sequence` when these two literals enclose input recognized by `Num_sequence`.

The definition of the non terminal `Num_sequence` says that such a sequence starts with a lexical entity `NUM` (whose semantic value is of type `num`, let us call it `n`), and then any (possibly empty) sequence of semicolon followed by a "NUM" returning a list of numbers `L`. The semantic value returned in this case is `n::L`. The "*" symbol when "applied" to a non terminal definition body acts as a macro and generates a rule for a (possibly empty) sequence of this non terminal.

Beware:

-- A common syntax error in grammar declarations consists in writing "`(* (parse ...`" which indicates the beginning of a comment instead of "`( * (parse ...`" indicating an iteration.

A grammar definition returns a functional value of type `string -> Parsers` which, when applied to a start symbol name, returns a record containing parsers for the corresponding grammar.

16.5 Typing grammar definitions

Typing a grammar definition is similar to typing a function definition. Left members of production rules have to be read as stream patterns, and terminals and non terminals are considered as constructors for streams: their source type is thus the type of semantic values that they return and is also the type of variables following terminals and non terminals.

For example, the terminal `NUM` returns values of type `num`, hence, occurrences of `n` are of type `num` in the following production:

```
parse NUM n -> n+1
```

Here are the types of semantic values returned by terminals:

`Literal`	returns values with type	`string`,
`INFIX`	"	`string`,
`NUM`	"	`num`,
`INT`	"	`int`,
`FLOAT`	"	`float`,
`BOOL`	"	`bool`,
{ <*expr*> }	"	the type of <*expr*>.

The last case denotes actions to be executed during parsing. They are useful inside of definitions of grammars for programs (cf. section 16.6.3).

The special forms "*" and "+", when applied to an anonymous non terminal producing semantic values of type τ return semantic values of respective types τ `list` and τ * τ `list` (standing for *non empty lists of type* τ).

Nevertheless, these typing rules are valid only for grammars producing values (i.e. grammar defined with a phrase such as: `grammar for values...`). Actions of grammars for programs have to be read as *"the syntax of the following CAML expression"* instead of its value. In this case, terminals produce values of type `ML`, and actions in the left part of production rules must return values of type `ML`. In both cases, one may obtain values with their expected type (embedded into dynamic values) by evaluating the result of parsing. With a grammar producing values, this evaluation step is trivial while with a grammar producing programs, it involves a full step of typechecking, compilation and execution.

16.6 Semantics of grammar definitions

We do not give here a complete and formal description of the semantics of a grammar definition: we only outline it and examples will illustrate the most important points.

Any grammar definition makes that grammar available for the system, i.e. phrases belonging to the corresponding object language may be typed by enclosing then between "<<" and ">>". We will often use this facility in the examples below.

In fact (and maybe surprisingly), grammars producing values are the most easy to understand. We consider them as the primitive construct and we will see that grammars for programs (i.e. the only interface available in previous versions of CAML) may be seen as a macro facility for definitions of grammars producing values (of type `ML`).

First of all, we have to mention that grammar definitions are translated into auxiliary files given as input to the Yacc parser generator. Yacc produces an output file containing a parser definition together with its parsing tables. CAML extracts these tables, translates them into CAML data structures which will be passed as argument to the CAML parser. These facts explain why some features of grammar definitions look like Yacc features (precedences treatment, LALR(1) parsers, ...).

Yacc generates parsing tables for LALR(1) grammars. This implies that only one token is sufficient in order to know what state the parser has to go to. This is particularly important when a grammar definition contains actions calling other parsers: a parser stops when the input scanned is not recognized; then the parser returns the last semantic value reduced only if one token is sufficient to stop. In particular, if the parser had to scan two or more tokens, then it enters in error states and produces a syntax error. More information and examples will be found in section 16.6.3.

16.6.1 Grammars producing values

We saw above that a grammar definition was composed of two parts:

- a *declaration part* in which some lexical information is given to the CAML lexical analyser, and precedences are specified in order to be used for disambiguating the grammar;

- and a part containing the grammar definition itself, composed of definitions of auxiliary non terminals and entry points.

Declaration part

The declaration part allows to specify what will be the characters enclosing strings and the characters enclosing comments (knowing that (* and *) are available in any syntax: system and user-defined).

As an example, we give a new syntax for strings by defining:

```
#grammar for values new_string =
#delimitors
#    string is "'";
#rule entry Main =
#    parse STRING s -> s
#;;
Calling Yacc ... ..............................................
Value new_string = <fun> : (string -> Parsers)
Grammar new_string for values defined
  entry Main : string

#<<'foo'>>;;
"foo" : string
```

Precedences are similar to Yacc precedences: only their syntax differs. The keywords `right`, `left` and `nonassoc` assign associativity information to the terminals and literals following them. For example, `right "+" "-"` means that the "+" and "-" operators have the same precedence and associate to the right. The `nonassoc` keyword says that no associavity rule may be used on the specified terminals or literals.

The `precedence` keyword binds an identifier to a precedence, knowing that the order of precedence specification is relevant. As an example, here is the header of the CAML syntax:

```
grammar for values Caml =

delimitors      (* string and comment delimitors *)
    string is "\""
    comment is "%"
;
precedences             (* in increasing order                    *)
    right "->";
    right "or";
    right "&";
    right "<-" ":=";    (* two terminals or literals in the same *
                         * associativity specification receive   *
                         * the same precedence                   *)
    right ",";
    right INFIX;
    right "@" "^";
```

```
    right "::";
    left "+" "-";
    left "*" "/";
    precedence uminus; (* uminus gets bound to a precedence higher *
                        * than the one of "*" and lower than the   *
                        * the of "."                               *)
    left "."           (* Note that an associativity specification *
                        * has the effect of declaring a new keyword*
                        * This fact may be used in order to
                        * introduce new keywords even if they are  *
                        * not effectively used in the grammar.     *
                        * This is useful for calling other parsers *
                        * inside of grammar actions                *)
;
```

Associativity information and precedences are only used by Yacc in order to disambiguate grammars. For more information, see the Yacc documentation.

Grammar specification

The part dedicated to the grammar specification contains syntactic and semantic information. They are organized in a multiple declaration of non terminals, one of them at least being an entry point (*start symbol* in the Yacc terminology). The syntactic information consists of sequences of literals, terminals and non terminals permitting to specify what should be read for the current non terminal to be recognized. The semantic information consists in giving names to semantic values returned by terminals, literals and non terminals and are used in semantic actions which will be evaluated when the production rule will be reduced.

As an example, consider the following grammar definition, and the side effects realized during the parsing phase:

```
#grammar for values sum =
# precedences left "+";
#rule entry Main =
#    parse {display_string "Starting parsing..."} _; Expr e;
#          {display_string "End of parsing. "} _ -> e
#and Expr =
#    parse NUM n -> display_num n; display_string " "; n
#        | Expr n1; Literal ("+" as s); {display_string (s^" ")} _ ;
#          Expr n2 -> n1+n2
#        | Literal "("; Expr n; Literal ")" -> n
#;;
Calling Yacc ... ....................................................
Value sum = <fun> : (string -> Parsers)
Grammar sum for values defined
  entry Main : num

#<<1+2+3>>;;
Starting parsing...1 + 2 + 3 End of parsing. 6 : num
```

Side effects are executed at each reduction of a non terminal production.

16.6.2 Grammars producing programs

This was an example of a grammar producing values. A grammar producing programs works the same way, but semantic actions appearing at the right hand side of the "->" are actions producing CAML abstract syntax trees. The last example would not even typecheck because in `display_string s`, `s` is no longer of type `string` but of type `ML`: it is the *syntax of* a character string. However, actions are still well typed because an action such as `display_num n` stands for `<:CAML:Expr<display_num #n>>`. Let us rewrite the previous grammar definition into a grammar for programs:

```
#grammar for programs sum =
# precedences left "+";
#rule entry Main =
#    parse {display_string "Starting parsing..."} _; Expr e;
#          {display_string "End of parsing. "} _ -> e
#and Expr =
#    parse NUM n -> display_num n; display_string " "; n
#        | Expr n1; Literal ("+" as s); {display_string (s^" ")} _ ;
#          Expr n2 -> n1+n2
#        | Literal "("; Expr n; Literal ")" -> n
#;;

line 8: ill-typed phrase, the variable s of type ML
cannot be used with type instance string in s^" "
1 error in typechecking

Typecheck Failed
```

Now, we eliminate the type error and see that CAML warns the user that some identifiers appearing in the action are in fact pieces of syntax for identifiers and thus have a dynamic semantics (i.e. context-dependent).

```
#grammar for programs sum =
# precedences left "+";
#rule entry Main =
#    parse {display_string "Starting parsing..."} _; Expr e;
#          {display_string "End of parsing. "} _ -> e
#and Expr =
#    parse NUM n -> display_num n; display_string " "; n
#        | Expr n1; Literal "+"; {display_string ("+ ")} _ ;
#          Expr n2 -> n1+n2
#        | Literal "("; Expr n; Literal ")" -> n
#;;
Warning: variable(s) prefix +, display_string, display_num
will be dynamically bound
Calling Yacc ... ..................................................
Value sum = <fun> : (string -> Parsers)
Grammar sum for programs defined
```

```
  entry Main

#<<1+2+3>>;;
Starting parsing...+ + End of parsing. 1 2 3 6 : num
```

We can see now that actions used as terminals are executed at parse time while evaluation of (syntax of) semantic actions takes place in the normal toplevel loop and is independent of parsing.

Moreover, some actions will seem to be never executed. Real actions of building CAML abstract syntax trees will be executed, but some of these trees will never be evaluated. Consider the following example:

```
#grammar for programs sum =
# precedences left "+";
#rule entry Main =
#    parse {display_string "Starting parsing..."} _; Expr e;
#          {display_string "End of parsing. "} _ -> e
#and Expr =
#    parse NUM n -> display_num n; display_string " "; n
#        | Expr n1; Literal "+"; {display_string ("+ ")} _ ;
#          Expr n2 -> n1+n2
#        | Literal "("; Expr n; Literal ")" -> n
#        | Expr n; Ignored x -> n
#and Ignored =
#    parse Literal "ignore" -> display_string "Not ignored!"
#;;
Warning: variable(s) prefix +, display_string, display_num
will be dynamically bound
Calling Yacc ... ..........................................................
Value sum = <fun> : (string -> Parsers)
Grammar sum for programs defined
  entry Main

#<<1+2+3 ignore>>;;
Starting parsing...+ + End of parsing. 1 2 3 6 : num
```

The abstract syntax tree of `display_string "Not ignored!"` has been built but has never been evaluated.

16.6.3 Mixing different grammars

Grammar definition have naturally a recursive structure. Non terminals are defined in a mutually recursive way. It is thus possible to mix two grammars by giving two entry points to a grammar. This is useful when one is interested in having access to different sublanguages of the same language, with sharing of the parsing tables.

Here is an example of such a mixing: we define a data structure for lambda calculus with type annotations of lambda-bound or `let`-bound variables. We then give a non ambiguous concrete syntax to that calculus (notice that we do not use associativity information for operators).

```
#type lambda =
#    'Bool of bool
#  | 'Num of num
#  | 'Var of string
#  | 'App of (lambda & lambda)
#  | 'Let of (typed_ident  & lambda & lambda)
#  | 'Abs of (typed_ident & lambda)
#
#and typed_ident =
#    'Typed_var of string & lambda_type
#  | 'Untyped_var of string
#
#and lambda_type =
#    'Num_type
#  | 'Bool_type
#  | 'Arrow_type of lambda_type & lambda_type
#;;
Type lambda defined
     'Bool : (bool -> lambda)
   | 'Num : (num -> lambda)
   | 'Var : (string -> lambda)
   | 'App : (lambda * lambda -> lambda)
   | 'Let : (typed_ident * lambda * lambda -> lambda)
   | 'Abs : (typed_ident * lambda -> lambda)
Type typed_ident defined
     'Typed_var : (string * lambda_type -> typed_ident)
   | 'Untyped_var : (string -> typed_ident)
Type lambda_type defined
     'Num_type : lambda_type
   | 'Bool_type : lambda_type
   | 'Arrow_type :
     (lambda_type * lambda_type -> lambda_type)

#grammar for values Lambda =
#rule entry Term =
#    parse Literal "let"; Typed_ident x ; "=" ; Term M ;
#          Literal "in"; Term N -> 'Let(x,M,N)
#        | Literal "\\"; Typed_ident x; Literal "."; Term M -> 'Abs(x,M)
#        | Term1 M -> M
#
#and Term0 =
#    parse BOOL b -> 'Bool b
#        | NUM n -> 'Num n
#        | IDENT x -> 'Var x
#        | Literal "("; Term M; Literal ")" -> M
#
#and Term1 =
#    parse Term1 M; Term0 N -> 'App(M,N)
#        | Term0 M -> M
#
```

```
#and Typed_ident =
#    parse IDENT v -> 'Untyped_var v
#        | IDENT v; Literal ":"; Type1 t -> 'Typed_var(v,t)
#
#and Type1 =
#    parse Type t1; Literal "->"; Type1 t2 -> 'Arrow_type(t1,t2)
#        | Type t -> t
#
#and entry Type =
#    parse Literal "N" -> 'Num_type
#        | Literal "B" -> 'Bool_type
#        | Literal "("; Type1 t; Literal ")" -> t
#;;
Calling Yacc ... ..............................................................
Value Lambda = <fun> : (string -> Parsers)
Grammar Lambda for values defined
  entry Term : lambda
  entry Type : lambda_type
```

We now have access independently to the syntax of terms and types as shown by the `info` function:

```
#info "Lambda";;
Lambda is :
 -- a variable bound to the value
      <fun> : (string -> Parsers)
 -- a grammar for values with entries
              Term: lambda
              Type: lambda_type
() : unit
```

We may thus type in:

```
#<:Lambda:Term<let x:N = 3 in succ x>>;;
('Let
 (('Typed_var ("x",'Num_type)),('Num 3),
  ('App (('Var "succ"),('Var "x")))))  :
 lambda
```

as well as:

```
#<:Lambda:Type<((N -> N) -> (B -> B))>>;;
('Arrow_type
 (('Arrow_type ('Num_type,'Num_type)),
  ('Arrow_type ('Bool_type,'Bool_type))))  :
 lambda_type
```

There is also another way of mixing completely independent grammars by calling a parser inside of a parse-time action (i.e. any action of a grammar for values

or actions used as terminal in grammars for programs). Before going into more details, we must take a look at parsing functions produced by a grammar declaration.

16.6.4 Parsing functions and toplevel parsers

So far, we have used the fact that a grammar declaration defines a parser which is made available to the CAML toplevel parser by writing pieces of concrete syntax between "<<" and ">>" quotation marks (with optional name and entry point). This is done automatically. But what about parsing functions available to the user? We saw that a grammar declaration gives a value whose name is the grammar name and whose type is always `string -> Parsers`. This function, when applied to an entry point name returns a data structure containing parsing functions:

```
#info "Parsers";;
Parsers is :
 -- a record type with labels :
      Parse holding values of type : (unit -> ML)
      Parse_string holding values of type : (string -> ML)
      Parse_channel holding values of type
      : (in_channel -> ML)
      Parse_raw holding values of type : (unit -> ML)
() : unit

#Lambda "Type";;
{Parse=<fun>; Parse_string=<fun>; Parse_channel=<fun>;
 Parse_raw=<fun>} : Parsers
```

This data structure is a records with four components which are all parsing functions, differing only in the way they take their input. The field:

- `Parse` holds a parser which can be called at toplevel.

  ```
  #(Lambda "Type").Parse ();;
  #(N -> N)
  #
  <:Caml:Expr<('Arrow_type ('Num_type,'Num_type))>> : ML
  ```

- `Parse_string` holds a parser which can be called on a piece of concrete syntax held in a string.

  ```
  #((Lambda "Type").Parse_string) "(N -> N)";;
  <:Caml:Expr<('Arrow_type ('Num_type,'Num_type))>> : ML
  ```

- `Parse_channel` holds a parser working in an input stream.

- `Parse_raw` holds a parser which must be used in order to be called by other parsers.

The last label above holds a parsing function doing no initialization of the lexical or syntactic analysers. Here is an example of its usage; we define a grammar for lists of lambda-calculus terms above in the following grammar:

```
#let parse_lambda_raw =
#  let parse_lambda = (Lambda "Term").Parse_raw in
#  function () -> (match parse_lambda()
#                  with MLquote(dynamic(M : lambda)) -> M
#                     | _ -> failwith "parse_lambda")
#;;
Value parse_lambda_raw = <fun> : (unit -> lambda)

#grammar for values Lambda_list =
#rule entry List =
#    parse Literal "["; Literal "]" -> []
#        | Literal "["; Lambda_seq L; Literal "]" -> L
#
#and Lambda_seq =
#    parse Lambda M; ( * (parse Literal ";"; Lambda N -> N)) L
#         -> M::L
#
#and Lambda = parse {parse_lambda_raw ()} M -> M
#;;
Calling Yacc ... ..........................................................
Value Lambda_list = <fun> : (string -> Parsers)
Grammar Lambda_list for values defined
  entry List : lambda list

#<<[]>>;;
[] : lambda list

#<<[\x.x;\y.y]>>;;
[('Abs (('Untyped_var "x"),('Var "x")));
 ('Abs (('Untyped_var "y"),('Var "y")))] : lambda list
```

We first defined the `parse_lambda_raw` parsing function which returns values of type `lambda`. Then, we used that function in a terminal action. Such an usage of parsing function is very useful, but the user has to be particularly careful about the neat separation between grammars. More precisely, in order for a called parser to return early enough, we have to make sure that the context in which it will be called is not ambiguous with the calling grammar. Since these grammars are separately analyzed, Yacc cannot detect such ambiguities.

16.7 Using quotation and antiquotation mechanisms

We saw in the previous section that a grammar definition had two side effects:

- to produce parsing functions,
- to make a parser available to the CAML toplevel.

Parsing functions are usable as any CAML functional value, and we will not emphasize on that point. To add a new parser to the CAML toplevel parser is very useful since it permits to talk about some CAML values by means of a user defined concrete syntax.

For example, using the concrete syntax for terms of lambda-calculus given before, it is possible to write:

```
##set default grammar Lambda:Term;;
Default grammar is now Lambda:Term
Pragma () : unit

#let S = <<\x.\y.\z.(x z) (y z)>>
#and K = <<\x.\y.x>> in 'App('App(S,K), K) = <<\x.x>>;;
false : bool
```

Since Lambda is a grammar for values, this is similar to the Lisp `quote`. Moreover, it is also possible to use a concrete syntax in a pattern as in:

```
#let is_lambda_x_dot_x =
#    function <<\x.x>> -> true
#           | _ -> false
#in map is_lambda_x_dot_x [ <<\x.x>>; <<\y.y>> ];;
[true; false] : bool list
```

The concrete syntax for lambda-terms provides a syntax for constant values of type `lambda`. But what happens if we want to write the expression `'App('App(S,K), K)` above in a syntax closer to the concrete syntax of lambda-terms application?

In order to do this, we must use a grammar for programs, and add a kind of *escapes* to CAML. We need something similar to the lisp `antiquote` together with its comma denoting escapes. This is possible by defining the following grammar for programs:

```
#grammar for programs Lambda =
#rule entry Term =
#    parse Literal "let"; Typed_ident x ; "=" ; Term M ;
#          Literal "in"; Term N -> 'Let(x,M,N)
#        | Literal "\\"; Typed_ident x; Literal "."; Term M -> 'Abs(x,M)
#        | Term1 M -> M
#
#and Term0 =
#    parse BOOL b -> 'Bool b
#        | NUM n -> 'Num n
#        | IDENT x -> 'Var x
#        | Escape e -> e
#        | Literal "("; Term M; Literal ")" -> M
```

```
#
#and Term1 =
#    parse Term1 M; Term0 N -> 'App(M,N)
#        | Term0 M -> M
#
#and Typed_ident =
#    parse IDENT v -> 'Untyped_var v
#        | IDENT v; Literal ":"; Type1 t -> 'Typed_var(v,t)
#        | Escape e -> e
#
#and Type1 =
#    parse Type t1; Literal "->"; Type1 t2 -> 'Arrow_type(t1,t2)
#        | Type t -> t
#
#and entry Type =
#    parse Literal "N" -> 'Num_type
#        | Literal "B" -> 'Bool_type
#        | Literal "("; Type1 t; Literal ")" -> t
#
#and Escape =
#    parse Literal "^"; {parse_caml_expr0 ()} e -> e
#;;
Calling Yacc ... .....................................................................
Value Lambda = <fun> : (string -> Parsers)
Grammar Lambda for programs defined
  entry Term
  entry Type
```

From the grammar for values we wrote a grammar for programs, and we added escapes to the metalanguage (CAML): the non terminal `Escape` calls during the parsing phase a parser of CAML expressions, and these expressions will be inserted in the program built by the parser for lambda-terms. We may now write:

```
#let S = <<\x.\y.\z.(x z) (y z)>>
#and K = <<\x.\y.x>> in <<^S ^K ^K>> = <<\x.x>>;;
false : bool
```

It is also possible to use antiquotation in patterns as in:

```
#let body_of_abstraction =
#    function <<\^x.^M>> -> M
#           | _ -> failwith "Not an abstraction"
#in body_of_abstraction <<\x.x>>;;
('Var "x") : lambda
```

16.8 Translating CAML V2-5 syntax files

It is possible to automatically translate CAML V2-5 syntax files into the V2-6 grammars format. There are still some restrictions: Yacc comments (between

"/*" and "*/" are not recognized; you have to remove them before. Moreover, the translator does not invent clever variable names in order to replace the "$i" Yacc variables. It calls them "dollar_i".

The function is:

♣ `translate_mly_file : (string -> string -> unit)`

and waits for a CAML V2-5 syntax file name (without .mly extension) and a target file name.

16.9 Building one's own toplevel

There are general facilities to trap and report syntax errors:

♣
```
print_syntax_error : (num * string * string list -> unit)
parsing_handler :
((num * string * string list -> 'a) -> ('b -> 'a) ->
 'b -> 'a)
```

If you have defined a toplevel loop, you can write a "loader" function for your files:

♣ `load_with_loop : ((unit -> 'a) -> in_channel -> unit)`

16.10 Pitfalls

Care must be taken about grammar names. They are used by Yacc and by the operating system. The major problem comes with the "'" character which may appear in CAML identifiers. Unfortunately, it has a special meaning for Yacc and for Unix. CAML tries to avoid these problems but sometimes does not succeed. This will be fixed in future versions, but in CAML V2-6, avoid non-alphanumeric characters in grammar names.

Moreover, a grammar name cannot be a constructor name; this is not surprising, but this fact is checked during grammar compilation and not during typechecking.

There are also a number of identifiers which must not be redefined for a grammar definition to work properly.

Other problems may arise while using grammars for values. These problems concern using concrete syntax inside of patterns. Grammar for values may return values that cannot be used as patterns such as functions. CAML detect some cases, but some are not detected. Amongst these cases, one may find:

- compiling files using concrete syntaxes producing functional values or lazy data structures,
- using concrete syntaxes producing functional values (in that case, one get a syntax error)
- actions doing input without calling a parser.

16.11 Summary

- Grammars definitions use the "`grammar`" toplevel construct.
- It is possible to use precedence and associativity informations in order to disambiguate grammars.
- A grammar may contain several entry points.
- Pattern matching may occur during the binding of semantic values returned by non terminals appearing in left members of production rules: this pattern matching occurs at parse time, partial matches are reported during the compilation of auxiliary files and pattern matching failures (as any action failure) cause parsing failures. *Thus it is wise to make sure that these matchings and more generally actions are safe.*
- There are two kinds of grammar definitions:
  - grammars for values: they are the basic form of grammar definitions. They are statically typechecked and admit arbitrary patterns in left members of production rules;
  - grammars for programs: they act as macros for actions producing CAML programs. They admit only variables and wildcards as patterns in left members of production rules. However, a special kind of actions is admitted as left member of a production rule: they are called *terminal actions* and are executed at parse time. Typechecking of grammars for programs makes sure that grammars will produce values of type `ML` (i.e. CAML programs).
- A grammar definition declares:
  - a value with name the name of the grammar. This value is a fonction which, when applied to an entry point name returns a data structure containing parsers. This data structure is of type Parsers described by:

```
#info "Parsers";;
Parsers is :
 -- a record type with labels :
      Parse holding values of type : (unit -> ML)
```

```
        Parse_string holding values of type : (string -> ML)
        Parse_channel holding values of type
        : (in_channel -> ML)
        Parse_raw holding values of type : (unit -> ML)
() : unit
```

  - a new "system parser" giving access to pieces of user-defined grammars input in the CAML toplevel.

- Actions may contain calls to other parsers (coming from a `Parse_raw` field). This may be used in order to "mix" different grammars.

Chapter 17

Evaluation functions

Some evaluation functions are available in CAML. They mainly take a piece of CAML abstract syntax (possibly under some concrete form) and return a value imbedded inside a dynamic value (cf. chapter 8). Coercions are then necessary in order to retrieve the static type of the result.

17.1 Abstract syntax evaluation

The basic evaluation function is:

```
eval_syntax : (ML -> dyn)
```

It takes a value of type ML and evaluates it:

```
#eval_syntax  (MLapply
#              (MLvar "+",
#               MLpair (MLconst (mlnum 1),
#                       MLconst (mlnum 2)),[]));;
(dynamic (3 : num)) : dyn
```

This example could also be written by using the Caml grammar:

```
#eval_syntax <:Caml:Expr<1+2>>;;
(dynamic (3 : num)) : dyn
```

Free variables of the program to be evaluated take their value in the global environment:

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

#eval_syntax <:Caml:Expr<x+2>>;;
(dynamic (3 : num)) : dyn

#let x=true in eval_syntax <:Caml:Expr<x-1>>;;
(dynamic (0 : num)) : dyn
```

Execution exceptions may be trapped by the adequate handler:

```
#try eval_syntax <:Caml:Expr<raise (failure "evaluation failed")>>
#with failure s -> dynamic s;;
(dynamic ("evaluation failed" : string)) : dyn
```

Type errors may also be handled by using a *catch all* handler, but error messages cannot be inhibited:

```
#try eval_syntax <:Caml:Expr<1+true>> with _ -> dynamic 0;;

line 1: ill-typed phrase, the constant true of type bool
cannot be used with type instance num in 1+true
1 error in typechecking
(dynamic (0 : num)) : dyn
```

The function `eval_syntax` may of course be composed with a user-defined parser such as:

```
let my_evaluation_function = eval_syntax o my_parser;;
```

17.2 Concrete syntax evaluation

The function `eval_string` takes a string as argument and returns a dynamic value. The string is supposed to contain a valid CAML expression under concrete form.

```
#eval_string "1+2";;
(dynamic (3 : num)) : dyn

#eval_string "let x=2 in x+1";;
(dynamic (3 : num)) : dyn
```

Note that in this case, the parsing of the string occurs at run time, and syntax errors may then occur at this time.

The function `eval_string` is nothing but the composition of the evaluation function describe above and the CAML parser for toplevel expressions in strings.

Chapter 18

Macros

18.1 The tools of macro-evaluation

Macros are the natural way of extending the syntax of a language without changing its parser's definition. Macros are programs dealing with abstract syntax, and the parser is in charge of evaluate macro-programs.

CAML provides the tools necessary to implement naturally macros and macro-evaluation. These tools are:

- grammar definitions: the syntax of CAML itself is available to the user,
- and evaluation functions: they permit to evaluate pieces of abstract syntax into values.

CAML possesses two syntaxes: `Caml` and `CAML`. `Caml` is a syntax for values (it is indeed the one into which you write your programs). It possesses escapes through the `#` sign. These escapes wait for CAML concrete expressions (resp. patterns, etc.) and attempt to macro-evaluate them into CAML values of type `ML` (resp. `MLpat` etc.) which are types of CAML abstract syntax trees. These escapes implement macro-calls.

The `CAML` syntax is the version "for programs" of `Caml`. It has the same syntax for escapes, but these escapes are not evaluated.

Examples:

```
#<:Caml:Expr<let x=1 in x+y>>;;
<:Caml:Expr<x+y where x = 1>> : ML

#<:CAML:Expr<let x=1 in x+y>>;;
<:Caml:Expr<x+y where x = 1>> : ML

#let Z = <:Caml:Expr<x'+y'>> in <:Caml:Expr<x+y+#Z>>;;

line 1: unbound variable Z
1 error in typechecking
(while parsing grammar Caml:Expr)
```

```
Typecheck Failed

#let Z = <:Caml:Expr<x'+y'>> in <:CAML:Expr<x+y+#Z>>;;
<:Caml:Expr<x+y+(x'+y')>> : ML
```

In the two first phrases, although the grammars `Caml` and `CAML` are different grammars, they return the same result in this case. In the two last phrases, we notice the different escape mechanisms in `CAML` and `Caml`: `Z` has been evaluated in two different environments.

The grammar `Caml` evaluates its escapes, and does it in a special environment called *environment of pragmas*. Escapes from `CAML` are of the same nature than escapes from any grammar for programs to the CAML toplevel parser.

18.2 Environment of Pragmas

The environment of pragmas is the environment where macros have to be defined in order to be used in the CAML toplevel grammar (i.e. the `Caml` grammar). This environment is accessible through the `#pragma` or `#` keywords.

```
##pragma let SYS_V = false and BERKELEY = true;;
Pragma Value SYS_V = false : bool
Pragma Value BERKELEY = true : bool

##pragma let BANNER =
#            if SYS_V then <:Caml:Expr<"(System V)">>
#            else <:Caml:Expr<"(Berkeley)">>;;
Pragma Value BANNER = <:Caml:Expr<"(Berkeley)">> : ML

#let max_file_name_length =
#    #(if SYS_V then <:Caml:Expr<14>>
#      if BERKELEY then <:Caml:Expr<1/0>>
#      else failwith "Unknown UNIX system");;
Value max_file_name_length = 1/0 : num
```

Identifiers `SYS_V` and `BERKELEY` are bound in the pragmas environment to boolean values and looked up in that environment during the evaluation of the conditional while the CAML parser is working. The conditional tests their values and returns pieces of CAML abstract syntax which is inserted in the output of the CAML parser. The result of the parser may be consulted by using the pretty-printer:

```
#pretty();;
#let max_file_name_length =
#    #(if SYS_V then <:Caml:Expr<14>>
#      if BERKELEY then <:Caml:Expr<1/0>>
#      else failwith "Unknown UNIX system");;
#
let max_file_name_length = 1/0
() : unit
```

Macro definitions and macro expressions may be arbitrary complex and the whole expressive power of the language may be used. As an example, the following MAP function macro-expands calls of a function over a list of arguments:

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

##pragma let APP = fun f x -> <<#f #x>>;;
Pragma Value APP = <fun> : (ML -> ML -> ML)

##pragma
#    let rec MAP f =
#        function (<<[]>> as S) -> S
#               | <<#x::#L>> -> <<(#f #x)::#(MAP f L)>>
#               | MLlist(x,L) -> MLlist(APP f x,map (APP f) L)
#               | S -> <<map #f #S>>;;
Pragma Value MAP = <fun> : (ML -> ML -> ML)
```

We are now ready to see macro calls to MAP:

```
#pretty();;
#let from_2_to_10 =
#    let l1 = #(MAP <<succ>> <<[1;2;3]>>) in
#    let l2 = #(MAP <<succ>> <<4::5::6::(map (add 5) l1)>>)
#    in l1@l2;;
#
let from_2_to_10 =
 let l1 = [succ 1; succ 2; succ 3] in
 l1@l2
 where l2 =
  succ 4::succ 5::succ 6::map succ (map (add 5) l1)
() : unit
```

and to evaluate the declaration:

```
#let from_2_to_10 =
#    let l1 = #(MAP <<succ>> <<[1;2;3]>>) in
#    let l2 = #(MAP <<succ>> <<4::5::6::(map (add 5) l1)>>)
#    in l1@l2;;
Value from_2_to_10 = [2; 3; 4; 5; 6; 7; 8; 9; 10] :
      num list
```

18.3 Macro Calls

Macro calls may be written everywhere an expression is legal. In this case, the expression which is macro evaluated must return values of type ML. It is also possible to write macro calls in patterns (the type must then be MLpat) in type definitions, functions defined by cases, and grammar definitions.

All macro calls have the following syntax:

"#" *<Caml Expression0>*

such as `#SYS_V` or `#(check_system SYS_V)`.

Macro calls may occur at the same place as:

- *<Caml Expression0>* and must be of type `ML`.
  For example:

```
#pretty();;
#print_string #BANNER; print_newline();;
#
print_string "(Berkeley)";print_newline ()
() : unit

#pretty();;
#1+ #(if SYS_V then <<0>> else <<1>>);;
#
1+1
() : unit
```

- *<Caml Pattern0>* and must be of type `MLpat`.
  For example:

```
#pretty();;
#let is_banner =
#    function #(if SYS_V then <:Caml:Pat<"(System V)">>
#                else <:Caml:Pat<"(Berkeley)">>) -> true
#           | _ -> false;;
#
let is_banner = function
    "(Berkeley)" -> true | _ -> false
() : unit
```

- an item of a type definition (except the first one) and must return values of type `MLconstruct list` (resp. `MLlabel list`) in case of a sum type (resp. record type).
  For example:

```
#pretty();;
#type directory_elem =
#    Regular_file of string * ancesters
#  | Directory of string * (directory_elem list) * ancesters
# #| (if not SYS_V
#     then <:Caml:Constr<
#            Symbolic_link of string *
#                             directory_elem * ancesters>>
#     else [])
#
#and ancesters =
```

```
#  { Father_dirs: (string * (directory_elem list)) list
# #;(if not SYS_V
#    then <:Caml:Labl<Father_symbolinks:
#                        (string * directory elem) list>>
#    else [])}
#;;
#
type directory_elem = Regular_file of string * ancesters |
                      Directory of string *
                                   directory_elem list *
                                   ancesters |
                      Symbolic_link of string *
                                       directory_elem *
                                       ancesters
and ancesters = { Father_dirs : (string *
                                 directory_elem list) list ;
                  Father_symbolinks : (string *
                                       directory elem) list }
() : unit
```

- match rule (except the first one) and must return values of type (**MLpat * ML**) **list**.
  For example:

```
#pretty();;
#let truncate_file_names =
#  function Regular_file (name,ancs) ->
#            Regular_file (sub_string name 0 max_file_name_length,
#                          map truncate_file_names ancs)
#         | Directory (name,elems) ->
#            Directory (sub_string name 0 max_file_name_length,
#                       map truncate_file_names elems)
#        #|(if SYS_V then []
#           else [ <:Caml:Pat<Symbolic_link f>>,
#                  <:Caml:Expr<Symbolic_link
#                              (truncate_file_names f)>>])
#;;
#
let truncate_file_names = function
    Regular_file (name,ancs) ->
      Regular_file
      (sub_string name 0 max_file_name_length,
       map truncate_file_names ancs)
  | Directory (name,elems) ->
      Directory
      (sub_string name 0 max_file_name_length,
       map truncate_file_names elems)
  | Symbolic_link f ->
      Symbolic_link (truncate_file_names f)
() : unit
```

- type constraint and must return values of type `MLtype`.
  For example:

```
#pretty();;
#let length_of_file_name s =
#    (length_string s #:(if SYS_V then <:Caml:Type<int>>
#                        else <:Caml:Type<num>>))
#;;
#
let length_of_file_name s = (length_string s : num)
() : unit
```

- non terminal definition (except the first one) and must return values of type `Grammar_rule list`.

- grammar sort and must return one of the following values:
  - `"values"` in order to specify a grammar for values,
  - `"programs"` for a grammar for programs.

- grammar name in a grammar definition and must return a value of type `string` (beware: the value must represent a valid CAML identifier).

18.4 Macros and Grammars

Macros and grammars fit well together to extend the CAML syntax in order to have new constructs. The realization of such extensions may be easy in case of very simple extensions and become quickly very hard. Here is an example of definition of such an extension.

This extension intends to replace the following CAML syntactic construct:

```
if test1 then result1
if test2 then result2
        .
        .
        .
else resultN
```

where the last line is optional. The new construct looks like:

```
  test1 -> result1
| test2 -> result2
        .
        .
        .
| _ -> resultN
```

where the last line is also optional.

The extension consists in a grammar cond which will be used as a macro. The following module contains only the grammar definition and exports nothing: its loading has the side-effect of adding a new grammar in the CAML toplevel.

```
#module Cond;;

#grammar for values cond =
#rule entry top =
#    parse Expr c; Literal "->"; Expr t; tope e
#          -> <:CAML:Expr< if #c then #t else #e >>
#and tope =
#    parse Literal "|"; Expr c; Literal "->"; Expr t; tope e
#          -> <:CAML:Expr< if #c then #t else #e >>
#        | Literal "|"; Literal "_"; Literal "->"; Expr e -> e
#        | -> <:Caml:Expr< () >>
#and Expr =
#    parse {parse_caml_expr()} e -> e
#;;
Calling Yacc ... ........................................
Value cond = <fun> : (string -> Parsers)
Grammar cond for values defined
  entry top : ML

#end module;;
```

We are now ready to use this grammar as shown in the following phrases:

```
#pretty();;
#let test x =
#   #<:cond< x=n   -> 0
#          | x=n+1 -> 1
#          | x=n+2 -> 2
#          | x=n+3 -> 3
#          | _     -> -1
#          >>;;
#
let test x = if x = n then 0 if x = n+1 then 1
 if x = n+2 then 2 if x = n+3 then 3 else -1
() : unit

#pretty();;
#let test x =
#   #<:cond< x=n   -> message "0"
#          | x=n+1 -> message "1"
#          | x=n+2 -> message "2"
#          | x=n+3 -> message "3"
#          >>;;
```

```
#
let test x = if x = n then message "0"
 if x = n+1 then message "1" if x = n+2 then message "2"
 if x = n+3 then message "3"
() : unit
```

Chapter 19

Automatic Documentation

CAML provides some facilities for the automatic documentation of programs. The information generated is the same as the CAML compiler produces for the declarations of values, types and exceptions. Additionally the list of all of the global values used in the declaration and the files in which they are defined (when possible) are also generated. This information is generated as decorated CAML comments (`(*| ... |*)`) in the file.

19.1 Documenting a file

♣ `cmn_compile : (string -> unit)`

This is the main function of this utility, it takes a file name (Say "`myprog`") and produces a code file ("`myprog.lo`") and a new commented version of the source file (the original source file is saved in "`myprog.sv.ml`"). There are two variants of this function which are also available:

♣ `comment_file : (string -> unit)`
`cmn_compil : (string -> unit)`

`comment_file` produces only the commented file, and `cmn_compil` is a variant of `cmn_compile` in which the compiled code is not loaded, giving an inconsistent environment only good for compiling other files.

19.2 Global values

As files are documented, `comments` stores for each value, the file in which it is defined, this information is used to classify global values used in each phrase.

Global values not defined in any of the files previously commented in the same session, are considered as belonging to the "CAML_system". The following function looks in a code file, to find out the information corresponding to the values of the corresponding source file:

♣ `cmn_ps_loadc : (string -> unit)`

The functions `cmn_compile`, `cmn_compil` and `cmn_ps_loadc` outside their use for documentation, work in the same way as their corresponding functions explained in the chapter "Loading and Compiling".

19.3 Bugs

This program erases all the comments of the form `(*| ... |*)` even if they where not generated by this utility.

Sometimes global references are not registered in the list of globals of the declaration.

If there are macro expansions in the file, only the type information and the globals of the expansion produced in the environment in which the file was commented, are given.

Chapter 20

Internal Representations

CAML values are represented in the machine memory as numbers, strings and cells from abstract machine LLM3 of Le-Lisp. This chapter deals with the representation algorithm, and the primitives provided to examine internal representations.

20.1 Lisp objects

20.1.1 Definition

It is possible to manipulate the Lisp objects as a CAML primitive type, named obj. The type obj is similar to a sum of string, atom, num, vectors and pairs of obj's, except that the pair is a pair of assignable pointers. This permits to manipulate various graph and destructive data structures. Notice that a predefined concrete syntax resides in the CAML system to parse values of type obj.

First are the injections:

♣
```
obj_string : (string -> obj)
obj_atom : (atom -> obj)
obj_int : (int -> obj)
obj_float : (float -> obj)
obj_vect : (obj vect -> obj)
obj_cons : (obj * obj -> obj)
```

This last constructor is the LISP cell constructor.

The LISP atom nil is defined in the CAML system as obj_nil.

```
#obj_nil;;
<:obj<()>> : obj

#obj_cons (obj_nil,obj_nil);;
<:obj<(())>> : obj
```

```
#<<()>>;;
<:obj<()>> : obj

#(* Numbers *)
#<<123>>;;
<:obj<123>> : obj

#(* Pairs *)
#<<(1 . 2)>>;;
<:obj<(1 . 2)>> : obj

#(* Lists *)
#<<(1 2 3)>>;;
<:obj<(1 2 3)>> : obj

#(* Strings *)
#<<"toto">>;;
<:obj<"toto">> : obj
```

20.1.2 Coercions from objects

Coercions for atomic objs:

♣
```
string_of_obj : (obj -> string)
num_of_obj : (obj -> num)
float_of_obj : (obj -> float)
int_of_obj : (obj -> int)
```

♣
```
string_from_obj : (obj -> string)
list_of_obj : (obj -> obj list)
vect_of_obj : (obj -> obj vect)
```

- `string_from_obj` coerces *any* obj into a string.
- `list_of_obj` coerces objs which are indeed a list into list of objs (and respectively for `vect_of_obj` with vectors).

20.1.3 Primitives for objs

The equality test may be done with the ordinary function:

♣ `eq : ('a * 'a -> bool)`

♣ `obj_left : (obj -> obj)`
`obj_right : (obj -> obj)`

Using pattern matching one gets the two projections (car and cdr):

```
#let obj_left = function
#    <<(^x . ^y)>> -> x | _ -> failwith "obj_left"
#and obj_right = function
#    <<(^x . ^y)>> -> y | _ -> failwith "obj_right";;
Value obj_left = <fun> : (obj -> obj)
Value obj_right = <fun> : (obj -> obj)
```

One can print values of type `obj` with:

♣ `print_obj : (obj -> unit)`

Finally, the LISP evaluator. To be used with great care!

♣ `lisp_eval : (obj -> obj)`

20.2 Watching internal representations

♣ `Repr : ('a -> obj)`

This function maps a CAML value to its `obj` internal representation:

```
#Repr [1;2;3];;
<:obj<(1 2 3)>> : obj

#Repr incr;;
<:obj<(() . 4928)>> : obj

#Repr Repr;;
<:obj<(() . 4948)>> : obj
```

Note that `[$XXX]` where `XXX` is an hexadecimal number, denotes an address (generally in the code zone and corresponding to the beginning of the code part of a closure).

20.3 Showing Representations

The function **show** provides a way of looking at internal representations of CAML objects, showing sharing and loops.

♣ `show : ('a -> unit)`

prints the representation of a CAML value under the form:

```
<< object with pointers where
   pointer = sub-object
   ...
   pointer = sub-object >>
```

Here is an example (we need to "open" the compilation to allow the recursive declaration of the non functional values `loop` and `vect_funct_pair_loop`):

```
##open compilation true;;
Directive () : unit

#let vect = [|1;2;3|]
#and funct = fun () -> ()
#and pair = (true,"true") in
#let rec loop = 1 :: loop
#and vect_funct_pair_loop = (vect,funct,pair,loop) in
#show [vect_funct_pair_loop;vect_funct_pair_loop];;
Warning: unsafe recursive declaration
(&1 &1) where

&1 = (#[ 1 2 3 ] . ((() . 10496) (t . "true") . &2))
&2 = (1 . &2)() : unit

##open compilation false;;
Directive () : unit
```

The output format used by **show** is as follows:

- `&i` is a pointer to a shared sub-part of the object.
- `#[ ...]` represents a vector.
- `[$ ...]` represents a code adress (part of the function's closure representation).
- `" ... "` is a string.
- `(...)` and `1`, `t`... can be seen as usual Lisp objects.

Part III

System Utilities

Chapter 21

System Functions

21.1 Following CAML work

♣
```
verb_system : (bool -> unit)
system_is_verb : (unit -> bool)
switch_verbose : (unit -> unit)
```

When the CAML system is verbose, you may follow the different phases of the toplevel work (including the loading of the files):

```
#verb_system true;;
entering system -> 0
() : unit
quitting system -> 1
entering system -> 0

#1;;
Typing...
Compiling...
Assembling...
Loading...
Running...
quitting system -> 1
entering system -> 0
1 : num
quitting system -> 1
entering system -> 0

#verb_system false;;
Typing...
```

```
Compiling...
Assembling...
Loading...
Running...
quitting system -> 1
() : unit
```

When a phrase is entered at toplevel it passes through several processing functions. It is

1. parsed
2. typechecked
3. compiled into abstract machine code
4. this code is assembled into Lisp Assembly Code (from Le-Lisp LLM3 virtual machine).
5. the LAP code is loaded and run.

The first phases may be followed using the above primitives:

♣
```
PARSE : (unit -> MLsyntax)
TYPE : (unit -> unit)
CAM : (unit -> code)
LAP : (unit -> unit)
```

21.2 Changing the toplevel behaviour

21.2.1 General setting

♣
```
verbose : (unit -> unit)
echo_verbose : (num -> unit)
```

verbose is an interactive function which asks you for an option (among 11) to toggle:

```
#verbose();;

Choose an option to switch:

<1> : Run times
<2> : Parsing times
```

```
<3> : Typechecking times
<4> : Compilation times
<5> : Assembling times
<6> : Loading times
<7> : System times
<8> : Types are not abbreviated
<9> : Types are not printed
<10> : Values are not printed
<11> : System verbose
Choose a number ? 8
() : void
```

`echo_verbose` does the same job but it may be used in programs. It expects as argument the number you would give as answer to `verbose`.

21.2.2 Type abbreviations

♣
```
echo_abbrev : (string -> bool -> unit)
echo_abbrevs : (bool -> unit)
```

With `echo_abbrev "toto" false` you disable the reduction of the type abbreviation named `toto`, and you revert to normal reduction mode with `echo_abbrev "toto" true`. With `echo_abbrevs` you may switch globally (i.e. for every abbreviations) the reduction process.

21.2.3 Toplevel printing

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

When programming a sub-system it is convenient to completely inhibit printing from the CAML toplevel (to let the sub-system display whatever it has to display).

```
#echo_types false;;
()

#echo_values false;;
#1;;#echo_types true;; : unit

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

21.2.4 Toplevel directives

With `#eval when print true` evaluation of lazy values is forced when printing.

With `#open printing true` values of abstract types are printed as much as possible, even if constructors or labels are currently unknown.

With `#open overloading true`, overloading of records labels may be ambiguous (a default strategy is used: that is, the first type compatible with the whole set of type constraints is chosen).

With `#open compilation true` recursive definition of values (with non functional types) is allowed (but this may leads to execution errors).

With `#fast arith true`, arithmetics is supposed to operate on "small integers", numerous primitives are expansed in-line, particularly vector and strings bounds are not verified when accessing them (this may leads to execution errors as well).

21.3 Date

♣
```
hour : (string -> string)
date : (string -> string)
```

These functions returns respectively the system hour and date, and use their string argument to format their result, separating fields with this string.

```
#hour "/";;
"12/34/21" : string

#date "/";;
"23/08/90" : string
```

21.4 Timing

21.4.1 Timing all user's computations

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

The first function sets the recording of runtimes on or off. The second records too the times spent during garbage collection.

Beware:

-- for the time being, the number of cells used during the computation as reported by the function `timers` remains 0 on the Sun version of the language.

21.4.2 Toplevel timings

♣

```
switch_run_times : (unit -> unit)
switch_assemble_times : (unit -> unit)
switch_parse_times : (unit -> unit)
switch_loader_times : (unit -> unit)
switch_compile_times : (unit -> unit)
switch_typing_times : (unit -> unit)
switch_system_times : (unit -> unit)
```

These functions are used when a precise part of the toplevel work has to be spied. They all switch on or off the feature they are bound to.

```
#switch_system_times();;
() : unit

#(* To get a not null parsing time ! *)
#let x = 1 in let y = 1 in let z = 1 in let t = 1 in let u = 1 in
#let (x::l) = map succ (replicate 1000 1) in ();;
Warning: 1 partial match in this phrase
() : unit
Typechecking of expressions: 0.02s Parsing: 0.05s
Assembling: 0.02s Runtime: 0.04s

#switch_system_times ();;
() : unit
```

21.5 To stop

♣

```
quit : (unit -> unit)
```

This function terminates the CAML session.

Note: the execution of a CAML expression may be interrupted, in UNIX, with the signal INT, usually bound to key rubout or CTRL C. This is useful for interruption of looping programs, for instance.

If this does not suffice you may attempt to kill your CAML session with the QUIT signal usually bound to the key CTRL \.

21.6 Help functions

Information about identifiers

There is an elementary info system provided by the single function

♣ `info : (string -> unit)`

This function helps you to know if a given ident is a type, or a constructor, or whether it is bound or not, or if it is the name of a primitive.

```
#info"num";;
num is :
 -- a concrete type with :
      superfluous constructor Big_num : (obj vect -> num)
      superfluous constructor Float : (float -> num)
      superfluous constructor Int : (int -> num)
() : unit

#info"::";;
:: is :
 -- a system superfluous constructor with type
      : ('a * 'a list -> 'a list)
 -- a keyword for grammar Caml

 -- an infix ident.
() : unit

#info"failure";;
failure is :
 -- an exception with type : string
() : unit

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

Searching defined identifiers

♣ `search_symbol : (string -> unit)`
`search_symbol_start : (string -> unit)`
`search_constructor : (string -> unit)`
`search_type : (string -> unit)`

```
search_exception : (string -> unit)
search_variable : (string -> unit)
```

With these functions you may find out if an identifier is stored in the global symbol table of the CAML system or find all the stored identifiers whose name contains some character string. The `search_symbol_start` function looks for an identifier starting by a given string. All the other ones find the identifiers containing their string argument. As the functions names may suggest it is possible to search for either exceptions or types or constructors or values (`search_variable`) or anything (`search_symbol`).

```
#search_exception "bot";;

() : unit

#search_constructor ":";;
::
() : unit

#search_variable "string_of";;
string_of_integer string_of_float string_of_bool string_of_obj
string_of_big_num string_of_atom string_of_num string_of_floating
string_of_int
() : unit
```

21.7 Terminal echoing

♣ `echo : (bool -> unit)`

With `echo false` you invalidate the echoing of CR-LF at the end of each line. This is particularly useful when running CAML under emacs.

21.8 Garbage Collection

♣
```
gc : (bool -> obj)
gc_info : (unit -> unit)
gc_alarm : (bool -> unit)
```

- `gc true` forces a garbage collection.

- `gc_alarm true` turns the garbage collector in verbose mode: for each garbage collection a message is displayed on terminal.
- `gc_info ()` gives statistics about garbage collection.

The information reported by the garbage collector is as follows:

```
   GC's due to the cell zone
    |             GC's ordered by the user
    |               |  Number of cells (or K cells)
    |               |       |   Number of symbols
    |               |       |            |        Number of strings
    |               |       |            |           |
    V               V       V            V           V
(gc 51 0 0 0 0 0 0 13 cons 31929 symbol 1819 string 17601 vector
   18355 float 0 fix 0 heap (471) code (266))
```

the heap and code zone are given in kilo bytes.

21.9 Communication with the operating system

♣ `comline : (string -> unit)`

Sends the `<string>` argument to the operating system.

```
#comline"pwd";;
/usr/local/caml/doc/manual
() : void
```

21.10 CAML initialisation

♣ `init_fun : (unit -> unit) ref`

This is a global reference which may be bound to a function by the user. This function is executed when entering the CAML core image. Thus you can initialize the CAML system to fit it to your application. A typical end of file to build a sub-system can be:

```
let init () = load"init_file";;
init_fun:=init;;
save_caml_image("application_name","My banner");;
```

Chapter 22

Trace

Many semantic programming errors result in type errors and are trapped and reported by the typechecker. But a program, although type errors free, may still be wrong and return unexpected values. In this case, you may spy the "input-output" behavior of this program by writing down the arguments given and the results obtained for each of its calls.

The trace package provides numerous functions to trace user's programs in a some (more or less) standard way. In addition there exists a special syntax, named `Trace`, to command the trace package.

Beware:

-- All the "tracing" functions and the grammar "Trace" are predefined as "autoload" into the CAML system, and thus one should expect a delay after the first call to one of them.

The first part of this description is devoted to the standard use of the trace mechanism, without using the grammar "Trace". The second part gives the general settings available. Finally the third part shows how to get a fine tuning over the trace mechanism using the trace grammar and how to deal with particular problems of trace.

22.1 Standard Tracing

22.1.1 Basic tracing functions

The main problem to face when tracing is the tremendous amount of material displayed on the terminal. Interesting information is very often lost in all this output. Thus (and even for the purpose of this explanation) it is necessary to adjust the way the trace will run.

In this section maximum information will be provided when tracing. The trace is in "verbose mode".

Three basic functions are provided to trace the user's defined functions:

♣ `trace : (string -> unit)`

This one is of general purpose (in particular, it traces only *the last level calls* to curried functions, and thus is not too "verbose"). Its argument is the name of the function to be traced[1].

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

#trace"add";;
() : unit

#add (1,2);;
<0>add ((1,2) : (num * num)) -->
<0>add ((1,2) : (num * num))   = 3 : num
3 : num
```

The number which is displayed, enclosed into < and >, is the depth of surrounding calls. The parenthesised expression is the current argument of the call.

```
#let add_2_fold (x,y) = 2*(add(x,y));;
Value add_2_fold = <fun> : (num * num -> num)

#trace "add_2_fold";;
() : unit

#add_2_fold (3,4);;
<0>add_2_fold ((3,4) : (num * num)) -->
 <1>add ((3,4) : (num * num)) -->
 <1>add ((3,4) : (num * num))   = 7 : num
<0>add_2_fold ((3,4) : (num * num))   = 14 : num
14 : num
```

When add is called, `add_2_fold` is still running, hence the number of surrounding calls is 1.

It is possible to trace recursive functions:

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

#trace"fact";;
```

[1]This argument cannot be a functional value: a string is needed to display the name of the traced function.

```
() : unit

#fact 4;;
<0>fact (4 : num) -->
 <1>fact (3 : num) -->
  <2>fact (2 : num) -->
   <3>fact (1 : num) -->
   <3>fact (1 : num)   = 1 : num
  <2>fact (2 : num)   = 2 : num
 <1>fact (3 : num)   = 6 : num
<0>fact (4 : num)   = 24 : num
24 : num
```

Trace of curried functions is available as well but this needs a little thought about trace mechanism:

```
#let add3 x y z = x+y+z;;
Value add3 = <fun> : (num -> num -> num -> num)

#trace"add3";;
() : unit

#add3 1 2 3;;
<0>add3 1 : num 2 : num (3 : num) -->
<0>add3 1 : num 2 : num (3 : num)   = 6 : num
6 : num

#add3 1;;
<fun> : (num -> num -> num)
```

No trace for `add3 1`!

In fact `add3` is a curried function. `trace` acts only when the traced function has received its complete set of arguments. If you want to trace any call of a curried function, regardless of the number of given arguments, use:

♣ `trace_all_args : (string -> unit)`

This function is designed to trace every argument of a curried function (although it works with non curried functions). Remember that, in CAML, a curried function is really a function of one argument yielding to another function: so the trace may be rather surprising at first glance.

```
#trace_all_args "add3";;
Warning: add3 already traced
() : unit
```

```
#add3 1 2 3;;
<0>add3 (1 : num) -->
<0>add3 (1 : num)   = <fun> : (num -> num -> num)
<0>add3 1 : num (2 : num) -->
<0>add3 1 : num (2 : num)   = <fun> : (num -> num)
<0>add3 1 : num 2 : num (3 : num) -->
<0>add3 1 : num 2 : num (3 : num)   = 6 : num
6 : num
```

If you partially evaluate a function (more exactly if you do not provide the complete set of arguments) intermediate functions are traced by `trace_all_args`:

```
#let add3_x = add3 1;;
<0>add3 (1 : num) -->
<0>add3 (1 : num)   = <fun> : (num -> num -> num)
Value add3_x = <fun> : (num -> num -> num)
```

then add3_x is automatically bound to a traced function:

```
#let add3_x_y = add3_x 2;;
<0>add3 1 : num (2 : num) -->
<0>add3 1 : num (2 : num)   = <fun> : (num -> num)
Value add3_x_y = <fun> : (num -> num)
```

Beware: -- The trace package automatically generates names for anonymous closures: what is displayed between the > symbol and the left parenthesis which indicates the argument of the call (`add3 1 : num` in the example above) is the name of the function or more precisely the name of the *closure* traced. For example when tracing `add3_x_y`, the trace package will generate the name "`add3 1 : num 2 : num`" for the corresponding closure:

```
#add3_x_y 4;;
<0>add3 1 : num 2 : num (4 : num) -->
<0>add3 1 : num 2 : num (4 : num)   = 7 : num
7 : num
```

Now, if only some arguments have to be traced, you may specify them by giving their rank:

♣ `trace_args : (string -> num list -> unit)`

This one traces only the arguments the ranks of which are given in the num list.

```
#trace_args "add3" [1;3];;
Warning: add3 already traced
() : unit
```

This command will trace all the calls to the function add3 which are such as (add3 x) and such as (add3 x y z) (i.e. calls to the first and third argument but *not* calls to the second argument).

```
#add3 4 5 6;;
<0>add3 (4 : num) -->
<0>add3 (4 : num)   = <fun> : (num -> num -> num)
<0>add3 4 : num 5 : num (6 : num) -->
<0>add3 4 : num 5 : num (6 : num)   = 15 : num
15 : num
```

22.1.2 Tracing functions with a predicate

When a fine tuning is required, tracing with a predicate may help: then a function call is reported if and only if the argument verifies a given condition. Primitives provided are:

♣
```
trace_if : (string -> string -> unit)
trace_args_if : (string -> (num * string) list -> unit)
trace_all_args_if : (string -> string -> unit)
```

trace_if is analogous to trace but needs a second argument: the name of the predicate used to trace the function (and similarly for trace_all_args_if). trace_args_if needs a list of pairs (rank of the traced argument, name of the corresponding predicate).

```
#let rec fib 1 = 1 | fib 2 = 1 | fib n = fib(pred n)+fib(n-2);;
Value fib = <fun> : (num -> num)

#let pred_fib = fun 5 -> true | _ -> false;;
Value pred_fib = <fun> : (num -> bool)

#trace_if "fib" "pred_fib";;
() : unit

#fib 6;;
<0>fib (5 : num) -->
<0>fib (5 : num)   = 5 : num
8 : num
```

22.1.3 Tracing functions from functions

When a function is very often used (for example map), it may be traced *only when called by another named function.* Primitives provided are:

♣
```
trace_from : (string -> string -> unit)
trace_args_from : (string -> (num * string) list -> unit)
trace_all_args_from : (string -> string -> unit)
```

- `trace_from` is analogous to `trace` but needs a second argument: the name of the calling function (and respectively for `trace_all_args_from`).
- `trace_args_from` needs a list of pairs (rank of the traced argument, name of the corresponding calling function).

In this mode the trace package will insert a comment `(* from [n] caller *)`. "`caller`" is the name of the function which is calling the traced function and "`n`" is the number of instances of "`caller`" simultaneously active.

```
#let g x = add3 x;;
Value g = <fun> : (num -> num -> num -> num)

#trace_all_args_from "add3" "g";;
Warning: add3 already traced
() : unit

#add3 1 2 3;;
6 : num

#g 2;;
<0>add3 (*from [0]g*) (2 : num) -->
<0>add3 (*from [0]g*) (2 : num)   = <fun>
: (num -> num -> num)
<fun> : (num -> num -> num)
```

It is possible to trace a function from itself:

```
#trace_from "fact" "fact";;
Warning: fact already traced
() : unit

#fact 3;;
<0>fact (*from [0]fact*) (2 : num) -->
 <1>fact (*from [1]fact*) (1 : num) -->
 <1>fact (*from [1]fact*) (1 : num)   = 1 : num
<0>fact (*from [0]fact*) (2 : num)   = 2 : num
6 : num
```

22.1.4 The trace mechanism

Closures versus functions

One has to be aware of the difference between functions (i.e. *identifiers* bound to a functional value) and closures (which are such *values* with a functional type). In CAML only closures can be traced. So when the trace mechanism is called in order to trace a function given by an identifier, it only uses[2] the closure which is the value of this identifier and, in fact, physically modifies this closure. Thus if two different functions (i.e. two different identifiers) are bound to the same closure, both are traced simultaneously.

```
#let f x = x + 1;;
Value f = <fun> : (num -> num)

#let g = f;;
Value g = <fun> : (num -> num)

#trace "f";;
() : unit

#g 2;;
<0>f (2 : num) -->
<0>f (2 : num)   = 3 : num
3 : num
```

The function g is traced (with f as function identifier !), even though it has not been explicitly requested by the user.

22.1.5 Untracing

One may untrace temporarily (using `set_trace`) or definitively (using `untrace_fun` for a single function, or `untrace` for the whole set of currently traced functions).

Two functions are provided to untrace functions:

♣
```
untrace : (unit -> unit)
untrace_fun : (string -> unit)
```

Their meanings are clear, knowing that `untrace()` untraces all traced functions. Untracing a function is just retrieving the initial closure of the function.

```
#untrace_fun"fact";;
() : unit
```

[2] apart from the use of the function name to display information!

Beware:
-- If a closure was dynamically created during the session by an already traced function, there is no function name associated to it in the trace package. Then it is impossible to untrace such a closure directly but the command **untrace** will disable completely the trace output from this closure (which still remains changed).

-- If a traced function is redefined, its name will be printed suffixed with the symbol "?"; it may be possible to untrace this function, writing its name with this symbol although it is not bound to any value in the user's environment.

`untrace` removes completely the trace mechanism. But it is often useful to temporarily compute without the long display provided by the traced functions without loosing the trace environment ... For this use:

♣ `set_trace : (bool -> unit)`

`set_trace false` disables the display. **`set_trace true`** will revert to the current tracing environment.

Notice that if a traced function behaves suddenly as its original untraced version, you must (before reporting any bug of the trace package!) try **`set_trace true`**: this may revert the situation to a better one.

22.1.6 Exceptions when tracing

When an exception is encountered during the evaluation of a traced function the trace reports it and the exception is propagated:

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

#trace"k";;
() : unit

#k 1;;
<0>k (1 : num) -->
<0>k (1 : num)   = 1 : num
1 : num

#k 0;;
<0>k (0 : num) -->
<0>k (0 : num) <-- Raising failure with value "fail"
: string

Evaluation Failed: fail
```

```
#(k 0) ? 0;;
<0>k (0 : num) -->
<0>k (0 : num) <-- Raising failure with value "fail"
: string
0 : num
```

22.1.7 Tracing the primitive operations

Some very basic functions (called primitives) cannot be properly traced since they are not bound to any closure. Their compilation often leads to a single machine instruction or to on-line code expansion (e.g. succ).

Nevertheless if you really want to trace a primitive, you can do it. A closure will be created and trace information put into it. Future uses of the primitive will call the closure and therefore will be traced. Previous uses of the primitive do not call the closure and therefore, will not be traced.

Thus, you have to trace the primitive *before* any occurrence of its name which has to be traced. Then the primitive function will be properly traced during the rest of the session (calls already encapsulated in some compiled code being excepted, e.g. it is no use to trace succ, if one wants to trace calls of succ from within functions loaded from an already compiled file).

```
#(* succ is not already traced. Hence this occurrence of succ
#   will be impossible to trace afterwards *)
#let f  x = succ x;;
Value f = <fun> : (num -> num)

#(* from now, the future occurrences of succ will be traced *)
#trace "succ";;
Warning: succ is a primitive operation
() : unit

#(* The ocurrence of succ embedded in f is not traced *)
#f 2;;
3 : num

#(* The occurrence of succ in g is regularly traced *)
#let g x = succ x;;
Value g = <fun> : (num -> num)

#g  2;;
<0>succ (2 : num) -->
<0>succ (2 : num)   = 3 : num
3 : num
```

The primitive status of a function is reported by the info function.

22.1.8 Tracing polymorphic functions

Tracing polymorphic functions is possible but can lead to surprising effects since polymorphic arguments cannot be properly printed because their type is unknown when the trace is turned on the polymorphic function.

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

#trace "map";;
Warning: map is a polymorphic function
() : unit

#map pred [1;2;3];;
<0>map <fun> : ('a -> 'b) ([-; -; -] : 'a list) -->
 <1>map <fun> : ('a -> 'b) ([-; -] : 'a list) -->
  <2>map <fun> : ('a -> 'b) ([-] : 'a list) -->
   <3>map <fun> : ('a -> 'b) ([] : 'a list) -->
   <3>map <fun> : ('a -> 'b) ([] : 'a list)   = []
   : 'b list
  <2>map <fun> : ('a -> 'b) ([-] : 'a list)   = [-]
  : 'b list
 <1>map <fun> : ('a -> 'b) ([-; -] : 'a list)   = [
 -; -] : 'b list
<0>map <fun> : ('a -> 'b) ([-; -; -] : 'a list)   =
   [-; -; -]
: 'b list
[0; 1; 2] : num list
```

The first argument to map (actually pred) is always printed as a polymorphic function, and its second argument as a polymorphic list. A better way to trace this kind of functions is described in the following section. See also the section 13.3.

22.1.9 Trace with representation of values

♣ `set_trace_with_show : (bool -> bool)`

In case of sharing or infinite lists the standard printing function is no longer useful to trace: so a new mechanism is necessary and one can set up another trace mode where only internal representations (see 20) of values are shown. This is also very convenient when one has to trace the polymorphic functions.

```
#trace_standard true;;
```

```
() : unit

#set_trace_with_show true;;
true : bool

#trace "fib";;
Warning: fib already traced
() : unit

#fib 3;;
<0>fib (3) -->
 <1>fib (2) -->
 <1>fib (.)   = 1
 <1>fib (1) -->
 <1>fib (.)   = 1
<0>fib (.)   = 2
2 : num

#trace"map";;
Warning: map already traced
Warning: map is a polymorphic function
() : unit

#map pred [1;2;3];;
<0>map (() . 26552) ((1 2 3)) -->
 <1>map (() . 26552) ((2 3)) -->
  <2>map (() . 26552) ((3)) -->
   <3>map (() . 26552) (()) -->
   <3>map . (.)   = ()
  <2>map . (.)   = (2)
 <1>map . (.)   = (1 2)
<0>map . (.)   = (0 1 2)
[0; 1; 2] : num list
```

Beware:

-- The first argument to `map` (`pred`) is printed as a closure (a pair of an empty environment and a code address), the second one as the representation of a list of `num`.

-- This kind of format may be rather confusing since the compiler uses a representation algorithm which may associate the same internal object to very different user's data.

In the following example the used data structure is a list of elements from a new concrete type, `New_num`, different of the predefined type `num`. Notice that the trace output is the same as in the preceding paragraph when tracing `map pred [1;2;3]`. In fact the values of the concrete type `New_num` are represented by the

same internal objects as the values of the type num !

```
#type New_num = New_num of num;;
Type New_num defined
     New_num : (num -> New_num)

#let New_pred = fun (New_num n) -> New_num (pred n);;
Value New_pred = <fun> : (New_num -> New_num)

#map New_pred [New_num 1;New_num 2;New_num 3];;
<0>map (() . 26558) ((1 2 3)) -->
 <1>map (() . 26558) ((2 3)) -->
  <2>map (() . 26558) ((3)) -->
   <3>map (() . 26558) (()) -->
   <3>map . (.)   = ()
  <2>map . (.)   = (2)
 <1>map . (.)   = (1 2)
<0>map . (.)   = (0 1 2)
[(New_num 0); (New_num 1); (New_num 2)] : New_num list
```

22.1.10 Local closure phenomenon

Some complex functions are traced in a rather surprising way when computation leads to the building of local closures which cannot be traced.

```
#let map_int (f:num -> num) = fun [] -> [] | l -> map_rec l
# where rec map_rec = fun [] -> [] | (h::l) -> (f h)::(map_rec l);;
Value map_int = <fun> :
      ((num -> num) -> num list -> num list)

#trace"map_int";;
() : unit

#map_int succ [1;2;3],;
<0>map_int <fun> ([1; 2; 3]) -->
<0>map_int . (.)   = [2; 3; 4]
[2; 3; 4] : num list
```

The local closure `map_rec` is not traced, since it is built when computing the result of `map_int succ [1;2;3]`.

22.2 Trace output format

22.2.1 Use of the directive "printer"

The "printer" directive allows the CAML system to call a user defined function to print the values of some given type. The trace package is suited to take account of this directive and it is the easiest way to control the trace output format.

For example, let us define a function which prints only the first three elements of a num list and another one for the printing of 'a list.

```
#let print_num_list =
#let print_three = function
# x1::x2::x3::_ ->
#  print_num x1; print_string"; ";
#  print_num x2 ;print_string"; ";
#  print_num x3 ; print_string"; ..."
#| [x1;x2] ->
#  print_num x1; print_string"; "; print_num x2
#| [x1] -> print_num x1
#| [] -> ()
#in function l -> print_string "[";print_three l;print_string "]";;
Value print_num_list = <fun> : (num list -> unit)

#let print_alpha_list =
#let print_head = function
# x1::x2::x3::_ ->
#  print_string"-1st-"; print_string"; ";
#  print_string"-2nd-" ;print_string"; ";
#  print_string"-3rd-" ; print_string"; ..."
#| [x1;x2] ->
#  print_string"-1st-"; print_string"; "; print_string"-2nd-"
#| [x1] -> print_string"-1st-"
#| [] -> ()
#in function l -> print_string "[";print_head l;print_string "]";;
Value print_alpha_list = <fun> : ('a list -> unit)
```

Now we define these functions as "printers" and we use them to define an infinite list and to trace hd. Notice that, since hd is a polymorphic function, it cannot call print_num_list but print_alpha_list instead. Thus when trace is set up on hd, the printer statically chosen to print its arguments must be polymorphic as well.

```
##printer print_num_list;;
New printer defined for type: num list
() : unit
```

```
##printer print_alpha_list;;
New printer defined for type: 'a list
() : unit

#let rec x = 1 :: x;;
Warning: unsafe recursive declaration
Value x = [1; 1; 1; ...] : num list

#trace "hd";;
Warning: hd is a system function
Warning: hd is a polymorphic function
() : unit

#hd x;;
<0>hd ([-1st-; -2nd-; -3rd-; ...]) -->
<0>hd (.)   = -
1 : num
```

22.2.2 Flags for trace

Some flags are provided in the trace package to control the output format: more precisely one may use the following functions:

♣
```
trace_arguments : (bool -> bool)
trace_types : (bool -> bool)
trace_closures : (bool -> bool)
trace_result_type : (bool -> bool)
trace_result : (bool -> bool)
trace_on_result : (bool -> bool)
```

The feature bound to the function is evidently enabled or disabled depending on the value of the provided boolean argument.

- `trace_arguments true`: the values of arguments are printed while tracing.
- `trace_types true`: the types of arguments are printed while tracing.
- `trace_closures true`: the results which are closures are printed while tracing.
- `trace_result_type true`: the types of results are printed while tracing.
- `trace_result true`: the results of the traced functions are printed (it is useful to inhibit this feature to follow computation when results are very large and it needs a long time to print them).

- `trace_on_result true`: the arguments are displayed when the results of the function calls are printed.

Default option:
When the trace file is loaded, functions are traced using (`trace_standard true`), see next paragraph.

22.2.3 General settings for trace

Designed to set all the flags at the same time, the global settings are:

♣
```
trace_standard : (bool -> unit)
trace_simple : (bool -> unit)
trace_verbose : (bool -> unit)
```

As far as preceding flags are concerned, and knowing that all these standard settings work with `trace_result true`, which is omitted, they are equivalent to:

```
let trace_standard = fun
    true ->
     trace_arguments true;trace_types false;trace_closures false;
     trace_result_type false;trace_on_result false
   |false ->
     trace_arguments true;trace_types true;trace_closures true;
     trace_result_type true;trace_on_result false;;

let trace_simple = fun
    true ->
     trace_arguments true;trace_types false;trace_closures false;
     trace_result_type false;trace_on_result true
   |false ->
     trace_arguments true;trace_types true;trace_closures false;
     trace_result_type true;trace_on_result true;;

let trace_verbose b =
     trace_arguments b;trace_types b;trace_closures b;
     trace_result_type b;trace_on_result b;;
```

Default option:
When the trace file is loaded, functions are traced using (`trace_standard true`).

22.2.4 Trace infos and current state of tracing mode

The current status of trace is displayed on the standard output using:

♣ `trace_status : (unit -> unit)`

which recalls the names of the setting functions as well.

```
#trace_status();;
Functions :
 map_int : ((num -> num) -> num list -> num list)
 map : (('a -> 'b) -> 'a list -> 'b list)
 fib : (num -> num)
 k : (num -> num)
f? : forgotten type
 add3 : (num -> num -> num -> num)
 add_2_fold : (num * num -> num)
 add : (num * num -> num)
Flags :
 Types (trace_types) : not traced
 Arguments (trace_arguments) : traced
 Type of results (trace_result_type) : not traced
 Arguments on return (trace_on_result) : not traced
 Results when closures (trace_closures) : not traced
 Results of traced functions (trace_result) : traced
First call traced (set_trace_bottom) : 0
Last call traced (set_trace_depth) : 20
Number of function calls allowed is not limited (set_trace_limit)
Trace prints representations of values (set_trace_with_show) :
false
() : unit
```

22.2.5 Limits to trace expansion

It is sometimes very useful to trace only selected calls of a given function and to limit the number of calls traced. One deals with this kind of features with:

♣ `set_trace_limit : (num -> unit)`
`set_trace_depth : (num -> unit)`
`set_trace_bottom : (num -> unit)`

- `set_trace_limit` determines the maximum number of calls allowed.
- `set_trace_depth` determines the last call traced (after this one each call is printed as ".").
- `set_trace_bottom` determines the first call traced (before this one each call is printed as "<").

```
#set_trace_depth 3;set_trace_bottom 2;;
() : unit

#trace"fib";fib 6;;
Warning: fib already traced
<
  <2>fib (4) -->
   <3>fib (3) -->


   <3>fib (.)   = 2
   <3>fib (2) -->
   <3>fib (.)   = 1
  <2>fib (.)   = 3
  <2>fib (3) -->
   <3>fib (2) -->
   <3>fib (.)   = 1
   <3>fib (1) -->
   <3>fib (.)   = 1
  <2>fib (.)   = 2
<
  <2>fib (3) -->
   <3>fib (2) -->
   <3>fib (.)   = 1
   <3>fib (1) -->
   <3>fib (.)   = 1
  <2>fib (.)   = 2
  <2>fib (2) -->
  <2>fib (.)   = 1
8 : num

#set_trace_limit 10;fib 13;;
<
  <2>fib (11) -->
   <3>fib (10) -->

Trace limit exceeded ...
```

Let us reset these limits to more convenient values for the remainder of this explanation:

```
#set_trace_bottom 0;set_trace_depth 20;set_trace_limit 100;;
() : unit
```

22.3 Grammar for trace orders

This grammar provides an easy way to refine the trace mechanism. With it, the user may syntactically specify the arguments which have to be traced, with which predicates, from which functions ... and may even precise what has to be done before (or after) the trace calls.

22.3.1 Specifying functions and arguments

Functions and arguments are denoted by identifiers. The grammar `Trace` can be used with just one identifier: the name of the function to be traced. Then `<:Trace<f>>` is equivalent to `trace "f"`.

The specification of the arguments of the function is done with a list of identifiers. We call it *parameters*. For instance in:

- "`f x y`": `x` and `y` designate the two first arguments of `f`

This list may contain one occurrence of the special ellipsis symbol "`...`", which stands for any number of identifiers. For example:

- "`f ...`" designate the function "`f`" and all its arguments.
- "`f ... x y`": `x` and `y` designate the last two arguments of `f`,
- "`f x ...y z`": `x` is the first argument, `y` and `z` are the last two arguments of `f`.

Furthermore, the parameter list may be reduced to "`all`" (thus equivalent to "`...`") and then it designates all the arguments.

22.3.2 Specifying orders

The user may define a specific set of trace orders for each argument separately. An order may be precised by options. The options follow the symbol "`:`" and are separated by the symbol "`;`". There are four orders:

1. "`trace`" which enables or disables the trace according to the chosen options
2. "`print`" which precises the way the arguments are (or are not) printed
3. "`before`" and "`after`" which indicates what has to be done before and after a call to a traced function.

Default option: if there is no order, only the last element of the parameters is traced.
Thus:

- `<:Trace< f ...>>` is equivalent to `trace_all_args"f"`
- `<:Trace< f ... x>>` is equivalent to `trace"f"`.

Trace order

A tracing order begins with the keyword "`trace`" followed by the parameters. For example, `<: Trace < f x y z : trace y>>` traces only the second argument of the function.

The options are:

- "`false`": the arguments indicated by the parameter are not traced
- "`from`" followed by a function's names list: arguments are traced only if the function is called by one of the functions named after the keyword "`from`"
- "`with`" "`predicate`" followed by a predicate which is a functional value. Each argument designated by the parameter is traced only if the predicate applied to it returns the value `true`.

The options "`from`" and "`predicate`" can be used together in any order.

- `<:Trace<f ... t : trace t from g>>` is equivalent to `trace_from "f" "g"`
- `<:Trace<f ... t : trace t with predicate p>>` is equivalent to `trace_if "f" "p"`.

```
#let g z = add3 1 1 z;;
Value g = <fun> : (num -> num)

#<:Trace< add3 x ... z:
#  trace z with predicate (fun z -> z=2) from g;
#  trace x >>;;
Warning: add3 already traced
() : unit

#add3 1 2 0;;
<0>add3 (1) -->
3 : num

#g 3;;
<0>add3 (1) -->
5 : num

#g 2;;
<0>add3 (1) -->
<0>add3 (*from [0]g*) 1 1 (2) -->
<0>add3 (*from [0]g*) . . (.)   = 4
4 : num
```

Beware:
-- If there is more than one tracing order on the same argument of a function, only the last one is significant.

Print order

A printing order begins with the keyword "`print`" followed by parameters and some options.
Default option: if there is no printing order in the sentence, all arguments are printed with the usual printer of the CAML system. If there is no option, only arguments designated by the parameter will be printed (with the usual printer).
The options are:

- "`never`": as expected the arguments designated by the parameters are never printed.
- "`with`" followed by a functional value which is a printing function. The tracing system uses this printer for the arguments designated by the parameters. In particular you may use `show` and the special option "`raw`" to print *without user's defined printers.*

If there is more than one printing order on the same argument, only the last one is significant.

```
#let my_if x y z = if x then y else z;;
Value my_if = <fun> : (bool -> 'a -> 'a -> 'a)

#let print_french_bool = function true -> print_string "vrai"
#                               | false -> print_string "faux";;
Value print_french_bool = <fun> : (bool -> unit)

#<:Trace<my_if x y z :
#   print x with print_french_bool;
#   trace z>>;;
Warning: my_if is a polymorphic function
() : unit

#my_if false 1 2;;
<0>my_if faux _ (_) -->
<0>my_if . . (.)   = -
2 : num
```

Before and after orders

This is used to evaluate a "prelude" and a "postlude" function before and after the call to the traced function, for instance to report information that cannot be reported by the standard options.

A prelude order has the following syntax:
"before" <*Parameters*> "do" followed by a functional value (the prelude one).
A postlude order has the following syntax:
"after" <*Parameters*> "do" followed by a functional value (the postlude one).

Each time the traced function receives an argument belonging to the parameters following before, the prelude function is called with this argument. The postlude function is called when the traced function receives an argument belonging to the parameters following after. The postlude function is called with two values: the argument and the result if the traced call succeeds, and only with the argument if the call fails.

```
#let prelude x =
#    echo_string "here is x: ";echo_num x;echo_newline();;
Value prelude = <fun> : (num -> unit)

#let postlude y _ =
#    echo_string"add3 has received it second argument: ";
#    echo_num y;echo_newline();;
Value postlude = <fun> : (num -> 'a -> unit)

#<:Trace<add3 x y z :
#        before x do prelude; after y do postlude>>;;
Warning: add3 already traced
() : unit

#add3 4 5 6;;
here is x: 4
add3 has received it second argument: 5
<0>add3 4 5 (6) -->
<0>add3 . . (.)   = 15
15 : num
```

The computation proceeds as follows:

- when add3 receives x the expression prelude x is evaluated and then computation of add3 x starts.
- when exiting from add3 x y with result r, the expression postlude y r is evaluated, and then the normal result r is returned.

This process takes place for each argument of the function traced, even if this argument is not traced, and even if an exception is raised during the evaluation of the call to the function.

Let us describe the semantics with a CAML program: if f is the traced function and f_n this function called with its $(n-1)$ first arguments. If a prelude and a postlude are defined on the nth argument, the function f_n becomes:

```
function xn ->
 prelude xn;
 try let result = fn xn in
     postlude xn result;
     result
 with _ -> postlude xn reraise
```

A complete example

```
#let f x y = if y = 1 then failwith "one" else x + y;;
Value f = <fun> : (num -> num -> num)

#<:Trace<f x y:
#   before x do
#    (fun x1 ->
#      message("the first argument is "^(string_of_num x1)));
#   before y do
#    (fun x2 ->
#      message("the second argument is "^(string_of_num x2)));
#   after y do
#    (fun y ->
#      print_string "after the call with the second argument (";
#      print_num y;print_string"), ";
#      fun r ->
#      message ("the evaluation leads to "^(string_of_num r)));
#   trace all;
#   print all with show>>;;
Warning: f already traced
() : unit

#f 1;;
the first argument is 1
<0>f (1) -->
<fun> : (num -> num)

#f 1 2;;
the first argument is 1
<0>f (1) -->
the second argument is 2
<0>f 1 (2) -->
<0>f . (.)   = 3
after the call with the second argument (2),
the evaluation leads to 3
3 : num
```

```
#f 1 1;;
the first argument is 1
<0>f (1) -->
the second argument is 1
<0>f 1 (1) -->
<0>f . (.) <-- Raising failure with value "one"
after the call with the second argument (1),
Evaluation Failed: one
```

Beware:
-- Remember that the `print` versions of the formatting primitives do not display at once their argument but will enqueue it for further formatting. You must use the `echo` and `display` versions to interleave correctly your output with the trace one. Notice the effect of `print_*` in the preceding example: the phrase

```
after the call with the second argument (1),
```

was displayed *after* the trace normal message

```
Raising failure with value "one"
```

even if it has been evaluated *before* !

22.3.3 Advanced Features

So far, for sake of simplicity, function names were given by identifiers. In fact, function names are defined as follows:

- *<Ident>*: the traced function is the functional value bound to this global identifier,
- *#<Caml expression0>*: macrocall (cf chapter 18). The traced function is the functional value bound to the global identifier, whose name is the value of this expression (which hence must be of type `string`).

Using macros, you may redefine the basic tracing functions as follows:

```
#let trace s = <:Trace<#s>>;;
Value trace = <fun> : (string -> unit)

#let trace_all_args s =
#    <:Trace<#s ... :trace ...>>;;
Value trace_all_args = <fun> : (string -> unit)

#let trace_if s p =
#    <:Trace<#s ... y : trace y with predicate #p>>;;
Value trace_if = <fun> : (string -> string -> unit)
```

```
#let trace_from s caller =
#    <:Trace<#s ... y : trace y from #caller>>;;
Value trace_from = <fun> : (string -> string -> unit)

#let my_flag = ref true;;
Value my_flag = (ref true) : bool ref

#let flag = my_flag in
#    <:Trace<f x :
#     trace x with predicate (fun x -> !flag)>>;;
Warning: f already traced
() : unit

#f 2;;
<0>f (2) -->
<fun> : (num -> num)

#my_flag:=false;;
false : bool

#f 3;;
<fun> : (num -> num)

#let my_trace_if (p:num -> bool) =
#     <:Trace<f y x : trace x with predicate p>>;;
Value my_trace_if = <fun> : ((num -> bool) -> unit)

#let my_predicate x = x=2;;
Value my_predicate = <fun> : (num -> bool)

#my_trace_if my_predicate;;
Warning: f already traced
() : unit

#f 0 0;;
0 : num

#f 0 2;;
<0>f 0 (2) -->
<0>f . (.)   = 2
2 : num

#let true_pred x = x>1 and false_pred x = x=0;;
Value true_pred = <fun> : (num -> bool)
```

```
Value false_pred = <fun> : (num -> bool)

#let my_trace' b =
#  <:Trace<f x : trace x with predicate
#            (if b then true_pred else false_pred)>>;;
Value my_trace' = <fun> : (bool -> unit)

#my_trace' false;;
Warning: f already traced
() : unit

#f 0;;
<0>f (0) -->
<fun> : (num -> num)

#f 2;;
<fun> : (num -> num)

#my_trace' true;;
Warning: f already traced
() : unit

#f 0;;
<fun> : (num -> num)

#f 2;;
<0>f (2) -->
<fun> : (num -> num)
```

22.4 The trace syntax

22.4.1 Syntax definition conventions

The metalanguage used to specify the syntax is based on the BNF. The meta-symbols have the following meaning:

::=	:	is defined as,
|	:	or,
[x]	:	0 or 1 occurrence of x
(x)*	:	0 or more occurrences of x,
(x)+	:	1 or many occurrences of x,
(x | y)	:	x or y,
<*Name*>	:	the non-terminal symbol Name
"xyz"	:	the keyword xyz

The syntaxes may use any of the CAML lexical units (`Ident`, `String`, `Bool`,

`Infix`, `Num`, `Int` and `Float`).

<Caml expression0> designates a CAML expression (entry point: Expr0 in the CAML grammar) which is a basic lexical unit or a parenthesised expression.

22.4.2 Trace syntax

<Trace> ::= *<Command>* ("`and`" *<Command>*) *

<Command> ::= *<Function name>* [*<Parameters>*] [":" *<Orders list>*]

<Order list> ::= *<Order>* (";" *<Order>*) *

<Order> ::=

```
        "trace"    <Parameters> [ <Trace options> ]
                   ( "," <Parameters> [ <Trace options> ] ) *
      | "print"    <Parameters> [ <Print options> ]
                   ( "," <Parameters> [ <Print options> ] ) *
      | "before"   <Parameters> "do" <Trace expression>
                   ( "," <Parameters> "do" <Trace expression> ) *
      | "after"    <Parameters> "do" <Trace expression>
                   ( "," <Parameters> "do" <Trace expression> ) *
```

<Function name> ::= `Ident` | "`#`" *<Caml expression0>*

<Parameters> ::= (`Ident`) * ["..."] (`Ident`) * |"`all`"

<Trace expression> ::=

```
    "#" Ident
  | "#" <Caml expression0>
  | <Caml expression0>
```

<Trace options> ::=

```
    "false"
  | "from" ( <Function name> ) + [ "with" "predicate" <Trace expression> ]
  | "with" "predicate" <Trace expression> [ <from> ( <Function name> ) + ]
```

<Print options> ::=

```
  | "never"
  | "with" <Trace expression>
  | "with" "raw"
```

22.4.3 The syntax semantics

The grammar "Trace" returns a CAML expression, which is evaluated at toplevel. The trace mechanism needs *the name* of the traced function. So if

<*function name*> is an Ident, the returned value is the corresponding string. On the other hand, if <*Trace expression*> is an Ident then the returned value is the corresponding CAML variable.

```
#pretty();;
#<:Trace<f  ... x : trace x with predicate p>>;;
trace_execute
 [{Trace_function_name="f"; Trace_function_first_args=[];
   Trace_function_ellipsis_args=[{Trace_arg_name="...";
                                  Trace_arg_predicate=
                                  Trace_bool false;
                                  Trace_arg_printer=
                                  Trace_printer_usuel;
                                  Trace_arg_from_functions=[];
                                  Trace_arg_prelude=Trace_none;
                                  Trace_arg_postlude=Trace_none}]
   ;
   Trace_function_last_args=[{Trace_arg_name="x";
                              Trace_arg_predicate=
                              Trace_predicate (dynamic p);
                              Trace_arg_printer=
                              Trace_printer_usuel;
                              Trace_arg_from_functions=[];
                              Trace_arg_prelude=Trace_none;
                              Trace_arg_postlude=Trace_none}]}]
() : unit
```

All the CAML expressions and identifiers following the symbol # are evaluated, just after the parsing. Typechecking errors concerning them are reported by the toplevel during the evaluation of the trace call.

```
#<:Trace<#f x : trace x with predicate (fun x -> y)>>;;

line 1: ill-typed phrase, the variable f of type
(num -> num -> num) cannot be used with type instance
string in (f : string)

line 1: unbound variable y in fun x -> y
2 errors in typechecking

Typecheck Failed
```

Chapter 23

Memo mechanism

This package allows automatic transformation of a function into a so-called "memo-function". This kind of function generally spares some computation when it is called twice with the same argument since it does not forget their results. That is to say that a table is built by the function and is updated each time it is called, remembering that certain arguments lead to certain results, so that a further call to the function with the same argument directly returns the desired result without more computation than a search in the table of previously obtained results.

23.1 Standard memo primitives

♣
```
memo : (string -> unit)
no_memo_fun : (string -> unit)
no_memo : (unit -> unit)
```

As in the trace function one may provide the name of the function to be "memoized", say f, to the function `memo`, then f becomes a memo-function.

- `no_memo_fun` reverts this mechanism: the closure whose name is argument becomes a "standard" closure.
- `no_memo` suppresses the effect of all previous memo commands.

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

#let rec fib (1 | 2) = 1
#      | fib n       = fib (pred n) + fib (n-2);;
Value fib = <fun> : (num -> num)
```

```
#fib 20;;
6765 : num
Runtime: 0.54s

#memo "fib";;
() : unit
Runtime: 0.02s

#fib 20;;
6765 : num

#fib 100;;
354224848179261915075 : num
Runtime: 0.10s

#fib 100;;
354224848179261915075 : num
```

Remark that in this case, without "memos", it should be impossible to compute `fib 100` in a recursive way, since the result is exactly the number of recursive calls needed in the non "memoized" version, which is evidently much too large ! In this case the memo mechanism converts an exponential algorithm into an (almost) linear one.

23.2 Other memo primitives

♣
```
memoq : (string -> unit)
memo_args_if : (string -> (num * string) list -> unit)
```

With `memoq`, you get "memoization" with the `eq` (physicall equality) primitive, instead of `=`. This is convenient for closures or cyclic data structures, or references. In the same vein you can use `memo_args_if` when you want to provide your own equality to the memo mechanism (for instance the first argument may be searched with `eq`, the second with "=" and the third by a user's defined predicate) in a similar way used for `trace_args_if`.

Beware:

-- Unconsidered use of this feature may completely overflow the memory since very large lists may be built and maintened (i.e. garbage is never collected). To prevent this overflow, you may evaluate `no_memo_fun "f";memo "f"` to reset the association list enclosed in the memoized function `f`, as soon as you are sure that you no longer need the previously computed results.

-- Moreover a memo function may run more slowly than its original version, typically when the overhead to retrieve the result is much bigger than to recompute it. Consider for example the memo version associated with:

```
let is_one = fun 1 -> true | _ -> false;;
```

If memoized, the function `is_one` will spend much more time to record its results than to compute directly the result. The story is even worse when a recursive function never *reuses* a previously computed result (the so-called Ackerman function leads to this category of problems).

Nevertheless the memo mechanism may be still very useful in appropriate cases.

Chapter 24

Statistics

In order to improve efficiency it is often the case that the programmer needs to detect which parts of his source text is the more often used. Then he may center his efforts on these parts to produce efficient code. This package offers the possibility of recording the number of calls of functions, and the computational time spent inside them.

24.1 Statistics on number of calls

24.1.1 Primitives for call statistics

There are three functions provided:

♣
```
stat : (string -> unit)
no_stat : (unit -> unit)
no_stat_fun : (string -> unit)
```

- `stat` is an "autoload" function predefined in the CAML system, which, given the name of the function `foo`, will record all the calls to `foo` and report these statistics (if any) for each phrase evaluated at top-level.
- `no_stat_fun` reverts this process for a single function.
- `no_stat` cancells all statistics currently active and reverts the old behaviour of "statted" functions.

Beware:
-- if no calls to any functions you have put statistics on are performed during the evaluation of a CAML phrase, the CAML toplevel will nevertheless display `Statistics:`. This is designed because the statistics tend to slow down the functions and you have to remember to "unstat", in order to return to the normal speed of computation.

24.1.2 Examples of statistics

```
#let rec map f = map_rec where rec
# map_rec = fun [] -> [] | (h::t) -> f h :: map_rec t;;
Value map = <fun> : (('a -> 'b) -> 'a list -> 'b list)

#stat "map";;
() : unit
Statistics :

#(* Note that calls at different levels are reported separately *)
#let g = map succ in g (replicate 100 1);();;
() : unit
Statistics :
map : 1 1

#(* Don't forget that statistics are in action somewhere *)
#print_string "CAML";;
CAML() : unit
Statistics :

#no_stat_fun "map";;
No more statistics on map
() : unit

#(* Statistics slow down computation *)
#timer true;;
() : unit

#let g = map succ in g (replicate 1000 1);();;
() : unit
Runtime: 0.06s

#stat "map";;
() : unit
Statistics :

#let g = map succ in g (replicate 1000 1);();;
() : unit
Statistics :
map : 1 1
Runtime: 0.09s

#(* Back to standard performance *)
#no_stat();;
```

```
No more statistics on map
() : unit

#let g = map succ in g (replicate 1000 1);();;
() : unit
Runtime: 0.05s
```

24.2 Recording Runtimes

To complete the preceding statistics on the dynamic behaviour of functions, we describe now how to record the amount of computational time spent in a given function.

Primitives for timing

♣
```
timing : (string -> unit)
no_timing_fun : (string -> unit)
no_timing : (unit -> unit)
```

These are strictly analogous to statistics on calls.

Notice that for efficiency purposes a recursive function is not "timed" as soon as it is currently running (i.e.: it has been already called for some argument). This prevents the overhead of recording times becoming prohibitive. Without this restriction, the functions which do a very small amount of work for each recursive call (e.g. `fib`), would become completely inefficient. For the same reasons only the functions of a single argument can be "timed".

Printing messages when reporting times

♣
```
set_timing_message : (string -> string -> unit)
timing_with_message : (string -> string -> unit)
```

- `timing_with_message` allow to put time recording on a function and simultaneously to fix a message which will be display when reporting runtimes of the function. For example CAML uses this possibility to print `Runtime:` when timer is on.
- `set_timing_message` fixes (or changes) the reporting message of a function with runtime statistics.

Timing examples

```
#let rec fib = fun (1|2) -> 1 | n -> fib (pred n) + fib (n-2);;
Value fib = <fun> : (num -> num)

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

#timer true;;
() : unit

#timing"fact";;
() : unit

#timing"fib";;
() : unit
Runtime: 0.02s

#fib 20;fact (fib 10);fact (fib 9);;
2952327990396041408476186096435200000000 : num
fib: 1.00s fact: 0.02s Runtime: 1.02s

#no_timing();;
No more timings on fib
No more timings on fact
No more timings on Run
() : unit

#timer true;;
() : unit

#fib 20;fact (fib 10);fact (fib 9);;
2952327990396041408476186096435200000000 : num
Runtime: 0.59s
```

Beware:
-- no_timing switches off the CAML timer since this timer is just a particular case using the timing package !

Part IV

CAML Library

Chapter 25

User's prelude

This file contains a set of general purpose functions, which has not been included in the very prelude of the language since these functions are not so often used.

25.1 Iterators

♣
```
it_map :
(('a -> 'b -> 'a) -> ('c -> 'b) -> 'a -> 'c list -> 'a)
set_extension : (('a -> 'b list) -> 'a list -> 'b list)
it_pair_list :
(('a -> 'b * 'c -> 'a) -> 'a -> 'b list * 'c list -> 'a)
```

- `it_map f g b l` "it_lists" `f` on the result of the mapping of `g` on the list `l`:

  `it_map f g b l = it_list f b (map g l)`

- `set_extension f l` builds a set from the results obtained by the application of `f` to the elements of `l` (`f` is supposed to produce lists).

- `it_pair_list f (l1,l2)` `it_lists` `f` on the list of pairs obtained from elements of `l1` and `l2`.

```
let it_map f g =
 let rec it_map_f_g a = function
     [] -> a | b::l -> it_map_f_g (f a (g b)) l in
  it_map_f_g
;;

let set_extension f = it_map union f []
;;
```

```
let it_pair_list f init pair_list =
 it_list f init (combine pair_list)
;;
```

Example:

```
#set_extension (make_set o explode) ["language";"machine"];;
["l"; "u"; "g"; "m"; "a"; "c"; "h"; "i"; "n"; "e"] :
 string list
```

♣ `num_map : ((num -> 'a -> 'b) -> 'a list -> 'b list)`

`num_map f l` maps `f` on `l` providing to each call the position in the list of the argument of the call:
`num_map f [e1; e2; ... ; ei; ... ; en] =`
`[f 1 e1; f 2 e2; ... ; f i ei; ... ; f n en]`
Example:

```
#num_map pair ["a";"b";"c"];;
[(1,"a"); (2,"b"); (3,"c")] : (num * string) list

#num_map
# (fun i e ->
#   print_string"The ";print_num i;print_string"th element is: ";
#   message e)
# ["a";"b";"c";"d"];;
The 1th element is: a
The 2th element is: b
The 3th element is: c
The 4th element is: d
[(); (); (); ()] : unit list
```

`num_map` may be defined with

```
let num_map f = snd o fold consf 1
    where consf i x = (i+1,f i x);;
```

But in fact `num_map` is just a closure of `map_i`, since we have:

```
#let num_map = (C map_i) 1;;
Value num_map = <fun> :
      ((num -> 'a -> 'b) -> 'a list -> 'b list)

let num_map f = map_i f 1
;;
```

25.2 Merging and sorting lists

♣
```
merge :
(('a * 'a -> bool) -> 'a list -> 'a list -> 'a list)
merge_num : (num list -> num list -> num list)
sort_num : (num list -> num list)
```

- `merge` is used to get the union of two sets represented as sorted lists.
- `merge_num` is specially devoted to lists of numbers.
- `sort_num` sorts lists of numbers in increasing order.

```
let merge ord l1 l2 =
 let rec merge_ord = function
     h1::t1,h2::t2 ->
       if h1 = h2 then h1::merge_ord (t1,t2)
       if ord (h1,h2) then h1::merge_ord (t1,h2::t2)
        else h2::merge_ord (h1::t1,t2)
   | [],l -> l
   | l,[] -> l in
  merge_ord (l1,l2) ? failwith "merge"
;;

let merge_num = merge prefix <
;;

let sort_num = sort prefix <
;;
```

25.3 Association lists

♣ `inverse_assoc : (('a * 'b) list -> ('b * 'a) list)`

- `inverse_assoc` built the inverse a table of association.

```
let inverse_assoc = map (fun (x,y) -> y,x)
;;
```

Chapter 26

Hashing

26.1 The universal hashing function

♣ `hash_univ : ('a -> num)`
`hash_univ_param : (num * num * 'a -> num)`

These functions associate with every CAML value a positive number (a small integer, indeed), trying to map them as evenly as possible. The average behaviour is quite good, but `hash_univ` is only supposed to verify the two following properties:

- two equal values are guaranteed to have the same hashing values, i.e. `x = y` implies `hash_univ x = hash_univ y`
- `hash_univ` always terminates, even on circular values (though the hashing may be somehow poor on such data)

Hashing is achieved by recursive descent on the representation of the value, and combining hashing values for the leaves of this representation, i.e. strings, integers, floating point numbers, and atoms, as follows:

```
##fast arith true;;
Directive () : unit

#let rec hash_obj obj =
#  let cpt_meaningful = ref 10 and cpt_depth = ref 1000
#  and hash_address x =
#    match address_of_obj x with
#        Addr_short n -> (num_of_int n)
#      | Addr_long(n,n') -> (num_of_int n')
#  in hash_rec 0 obj
```

```
#  where rec hash_rec accu obj =
#    if !cpt_depth<0 or !cpt_meaningful<0 then accu else
#      match obj with
#          <:obj< () >> ->
#                accu
#        | obj_int i ->
#                decr cpt_meaningful; accu * 263 + Int i
#        | obj_float f ->
#                decr cpt_meaningful; accu * 263 + Float f
#        | obj_string s ->
#                decr cpt_meaningful;
#                (hash_string accu (pred(length_string s))
#                 where rec hash_string accu i =
#                  if i<0 then accu else
#                   hash_string
#                    (accu * 263 + nth_ascii(i,s)) (pred i))
#        | obj_cons(obj1,obj2) ->
#                decr cpt_depth;
#                hash_rec (hash_rec accu obj1) obj2
#        | obj_vect V ->
#                decr cpt_depth;
#                (hash_vector accu (pred (vect_length V))
#                 where rec hash_vector accu i =
#                  if i<0 then accu else
#                   hash_vector
#                    (hash_rec accu (vect_item(V,i))) (pred i))
#        | x ->  decr cpt_meaningful;
#                accu * 263 + hash_address x
#;;
Value hash_obj = <fun> : (obj -> num)

#let hash_univ = hash_obj o Repr
#;;
Value hash_univ = <fun> : ('a -> num)
```

The actual function `hash_univ` is written in machine language, for the sake of efficiency.

`hash_univ_param` is similar to `hash_univ`, but it uses two extra `num` arguments to tailor `hash_univ` to a particular use. Namely: `hash_univ_param(n1,n2,O)` finds an hash code for the object `O`, exploring `O` until at least `n2` significant leaves has been seen, under the constraint that at most `n1` recursive calls to the hashing function can be made.

`hash_univ O` is equivalent to `hash_univ_param (1000,20,O)`. In case of structures sharing a big tree from their root, `hash_univ` may have bad results, finding the same hash code for a lot of different objects. An example is hash coding all the

sub-lists of a list which has many instances of the same element (or even worse, only one repeated element). In this case it may help to tune the hashing function using `hash_univ_param` instead of `hash_univ`.

26.2 Using dynamic hashing instead of lists

A dynamic hashing table is a vector of lists, called *buckets*, holding a collection of elements, with the convention that all the elements of the bucket number n hash to the value n (modulo the length of the vector). Hence, adding, removing an item, and searching for a given item may be restricted to the bucket whose number is the hash key of the item. Given a hashing table of size m, supposing a perfectly regular hashing, this allows storing n elements with maximum retrieval time proportional to n/m, instead of n in case of a linear list.

The choice of a size for the hash table is left to the user; if the amount of data is not too important, a n/m ratio of 1 to 5 is sensible. Also, it is well known that hashing usually works better when the table size is a prime number.

♣ `suggested_hash_table_size : (num -> num)`

The following function is intended to help you choose the right size: giving it an estimated size, it will compute the first prime greater than it.

```
let suggested_hash_table_size n =
 let is_prime n =
  let dmax = integer (sqrt n) in
  let rec is_prime_rec d =
   d >= dmax or
   n mod d <> 0 & (n mod d+2 <> 0 & is_prime_rec (d+6)) in
   n mod 2 <> 0 & (n mod 3 <> 0 & is_prime_rec 5) in
 let rec hash_size_rec m = if is_prime m then m
  if is_prime (m+2) then m+2 else hash_size_rec (m+6) in
  hash_size_rec (6*integer (n/6)+5)
;;
```

Once chosen the size, creating an empty hash table should be achieved by the following primitive:

```
let hash_init n = vector n of [];;
```

but the current typechecker cannot handle this function, because `[]` is polymorphic and polymorphic vectors are always prohibited. Hence the user have to allocate its table "by hand", constraining the empty list if necessary:

```
let my_hash_table = vector 1013 of ([] : (string & num) list);;
```

♣ `hash_clear : ('a list vect -> unit)`
`hash_copy : ('a vect -> 'a vect -> unit)`

The function **hash_clear** empties (physically) the given hash table; **hash_copy** copies the content of the first given hash table into the second one, throwing away its content.

```
let hash_clear v = modify_vect ((fun _ -> []),v)
and hash_copy v v' =
 do_vect_i (fun i b -> vect_assign (v',i,b)) v
;;
```

26.3 Hashed sets

We give here some primitives to deal with sets represented by hashed tables. Most operations — especially, membership — are much faster than with the usual list representation. But adding and removing elements is done via side effects on the table, making these primitives unsuitable for some applications.

♣ `hash_add_elem : ('a -> 'a list vect -> 'a)`
`hash_remove_elem : ('a -> 'a list vect -> unit)`
`hash_mem : ('a -> 'a list vect -> bool)`

The functions **hash_add_elem** and **hash_remove_elem** respectively add one element to the given hashed set (except if it was already here, i.e. we implement here true sets), and remove one element. They work by side effects on the table, as mentioned above. The function **mem_hash** is the exact counterpart of **mem**, i.e. it tests membership.

```
let hash_add_elem e v =
 (if not mem e (vect_item (v,i))
   then vect_assign (v,i,e::vect_item (v,i));
  e
 where i = hash_univ e mod vect_length v)

and hash_remove_elem e v =
 (vect_assign (v,i,except e (vect_item (v,i)));()
 where i = hash_univ e mod vect_length v)
```

```
and hash_mem e v =
 match vect_item (v,hash_univ e mod vect_length v) with
    [] -> false
  | [e1] -> e = e1
  | [e1; e2] -> e = e1 or e = e2
  | L -> mem e L
;;
```

Beware:
-- Physical modification of values stored in hashed sets is not advisable, since the hash key for a modified element will change also, but the element will stay in the same bucket, hence retrieval of that element will usually fail. For instance:

```
#let v = vector 10 of ([] : string list) and s = "foo";;
Value v = [|[]; []; []; []; []; []; []; []; []; []|] :
      string list vect
Value s = "foo" : string

#hash_add_elem s v;;
"foo" : string

#hash_mem s v;;
true : bool

#replace_string s "a" 1;;
"fao" : string

#hash_mem s v;;
false : bool
```

♣
```
hash_union : ('a list vect -> 'a list vect -> unit)
hash_intersect : ('a list vect -> 'a list vect -> unit)
hash_subtract : ('a list vect -> 'a list vect -> unit)
```

These are the usual operations between two sets, where the result is physically put into the secod one. These primitives operate bucket by bucket, hence they don't work on tables of different sizes.

```
let hash_union,hash_intersect,hash_subtract =
 let hash_oper oper v v' =
  if vect_length v = vect_length v'
   then do_vect_i
```

```
        (fun i b' ->
         vect_assign (v',i,oper (vect_item (v,i)) b'))
        v'
   else failwith "set operation : different lengths" in
  hash_oper union,hash_oper intersect,hash_oper subtract
;;
```

♣
```
hash_merge : ('a list vect -> 'a list vect -> unit)
hash_resize : ('a list vect -> 'a list vect -> unit)
```

The function `hash_merge` also perform union of two hashed sets, but in a different way, so as to handles as well tables of different sizes. This gives a way to copy the content of a hashed set into another one of different size, as showed by `hash_resize`. However, these functions are slower than `hash_copy` and `hash_union`, because all the elements have to be hashed again.

```
let hash_merge v v' =
 do_vect (map (fun e -> hash_add_elem e v')) v
;;

let hash_resize v v' = hash_clear v';hash_merge v v'
;;
```

26.4 Hashed association lists

Hashing allows efficient implementation of association lists, i.e. lists of pairs associating a value to a key.

♣
```
hash_add_point :
('a * 'b -> ('a * 'b) list vect -> 'a * 'b)
hash_remove_point : ('a -> ('a * 'b) list vect -> unit)
hash_assoc : ('a -> ('a * 'b) list vect -> 'b)
hash_mem_assoc : ('a -> ('a * 'b) list vect -> bool)
```

Adding a pair of a key and a value to the hashed A-list, i.e. binding a key to a value, is achieved through `hash_add_point`; removing of such a binding, given its key, is done by `hash_remove_point`. The function `hash_assoc` retrieves the value associated with the given key; `hash_mem_assoc` indicates whether the given key is bound or not.

Binding a key to different values, i.e. adding several pairs with the same key and different values, is allowed, though these primitives always use the most recent binding, i.e. `hash_assoc` will always return the value associated last to the key, and `hash_remove_point` will always throw away the last binding of a key, therefore "popping" the previous one.

```
let hash_add_point (key,val as pair) v =
 (vect_assign (v,i,pair::vect_item (v,i));pair
 where i = hash_univ key mod vect_length v)

and hash_remove_point key v =
 (vect_assign (v,i,except_assoc key (vect_item (v,i)));()
 where i = hash_univ key mod vect_length v)

and hash_mem_assoc key v =
 match vect_item (v,hash_univ key mod vect_length v) with
    [] -> false
  | [key1,_] -> key = key1
  | [key1,_; key2,_] -> key = key1 or key = key2
  | L -> mem_assoc key L

and hash_assoc key v =
 match vect_item (v,hash_univ key mod vect_length v) with
    [] -> failwith "find"
  | [key1,val1] ->
      if key = key1 then val1 else failwith "find"
  | [key1,val1; key2,val2] -> if key = key1 then val1
      if key = key2 then val2 else failwith "find"
  | L -> assoc key L
;;
```

26.5 Using hashing to build shared structures

When using very large data structures in which the same datas occur several times, i.e. where many parts of the structure are equal, trying to share these datas, i.e. having but one physical representation of the data in memory and many pointers to it, might prove dramatically space-saving.

Such a sharing is conveniently achieved by keeping a list of all values we have used so far, and after building a new value, searching the list for a previous value which is equal to the new value, and later always using the "old" value thus found; that way, the value we built won't be used anymore and will be garbage collected soon.

Such a search is usually time-consuming; efficient hashing allows keeping this time affordable. Don't forget that cluttered memory means frequent time-wasting garbage collection, hence sharing may, in the end, speed up some programs.

♣
```
hash_repr : ('a -> 'a list vect -> 'a)
hash_find_repr : ('a -> 'a list vect -> 'a)
```

Given an (initially empty) table and a value, the function **hash_find** returns the *representative* for it, i.e. it gives a value equal to the starting value (**hash_repr** v table = v is always true), and when called several times with the same value, it returns the same physical object (i.e. v = v' implies that **eq(hash_repr** v table, hash_repr v' table) is true).

Using hash_repr is straightforward: after initializing a hash table, each time you create, say, a string, and you are about to put it into some relatively permanent data structure (one with a long lifespan, such as a symbol table, for instance), call hash_repr on it and use the result instead.

The function hash_find_repr is similar, but fails when the given value is not already in the table. It is sometimes useful, as the following example shows.

```
let hash_find_repr e v =
 match vect_item (v,hash_univ e mod vect_length v) with
    [] -> failwith "find"
  | [e1] -> if e = e1 then e1 else failwith "find"
  | [e1; e2] -> if e = e1 then e1
      if e = e2 then e2 else failwith "find"
  | L ->
      let rec find_repr = function
          [] -> failwith "find"
        | e'::L -> if e = e' then e' else find_repr L in
       find_repr L
;;

let hash_repr e v = hash_find_repr e v ? hash_add_elem e v
;;
```

Advanced example: sharing substructures

Suppose you are using a term-like data structure, such as, for instance:

```
#type term = Leaf of string  |  Node of string & term list;;
Type term defined
     Leaf : (string -> term)
   | Node : (string * term list -> term)
```

Maximal sharing, i.e. sharing of common sub-terms, may then be achieved by the following share_term function:

```
#let rec share_term =
# let sharing_table = vector 263 of [] in
#  function Leaf s as term -> hash_repr term sharing_table
#         | Node(s,L) as term ->
#           try
#           (* The whole term is already in the table *)
#           hash_find_repr term sharing_table
#           with failure "find" ->
#           (* Otherwise: we find recursively representatives
#              for the elements of L and we build a new Node
#              with these sons; this new Node is the one we
#              put in the hash table and return *)
#           hash_add_elem (Node(s,map share_term L)) sharing_table
#;;
Value share_term = <fun> : (term -> term)
```

This is a tried-and-true technique: in the CAML system itself, the types of global values are represented by a term-like structure; sharing them as shown led to a space saving of no less than 120 Kbytes ...

Chapter 27

String Utilities

This package, named "strings.ml", from the CAML library provides some non standard functions useful for string operations.

27.1 Character operations

Common punctuation marks are recorded in a variable:

```
#punctuation;;
";,.:?!" : string
```

♣ `is_letter : (string -> bool)`

Tests if a character string is a letter.

♣ `substitute_char :`
`(string -> string -> string -> num -> string)`

`substitute_char c s' s n` substitutes all the occurrences (starting from position `n`) of the character `c` in the string `s` by the string `s'`.

```
#substitute_char "i" "la" "oui monsieur" 0;;
"oula monslaeur" : string
```

27.2 Word operations

♣ `in_string : (string -> string -> string -> num -> string)`

`in_string char1 char2 s n` extracts the first occurrence of a substring of `s`, matching `char1*char2` (where * stands for any number of any character), starting from position `n`.

Example:

```
#let word = in_string space_char space_char;;
Value word = <fun> : (string -> num -> string)

#word "to to ti" 1;;
"to" : string
```

♣

```
following_word : (string -> num -> string)
rev_split_string :
(string -> string -> string -> string list)
```

`rev_split_string sep single_word s` splits the string `s` into words some are separated by any character belonging to `sep` some are single characters (belonging to `single_word`). Result is in reverse order (due to the algorithm used) and no `rev` is done since it may be convenient (or indifferent) to get this order.

Example:

`rev_split_string (space_chars^punctuation) "" s` splits s into words regardless of punctuation chars:

```
# rev_split_string (space_chars^ punctuation) """oui,monsieur";;
["monsieur"; "oui"] : string list
```

♣

```
rev_split_words : (string -> string list)
split_words : (string -> string list)
```

`rev_split_words` is equivalent to:

```
let rev_split_words = rev_split_string space_chars punctuation;;
```

and returns punctuation chars as single words of the result.

```
#split_words "oui,monsieur";;
["oui"; ","; "monsieur"] : string list
```

27.3 Scanning strings

These functions extend the capabilities of some standard CAML functions to lists of strings. For instance the two following primitives extend respectively `scan_string` and `index_string`:

♣
```
lscan_string : (string -> string list -> num * string)
lindex_string :
(string -> string list -> num -> num * string)
```

- `lscan_string s sl` finds in `s` the first occurrence of one of the strings of the list of strings `sl` (if two strings of `sl` match the same position the longer is choosen).
- `lindex_string s sl n` finds in `s` the first occurrence of one of the strings of the list of strings `sl`, starting from position `n` (if two strings of `sl` match the same position the longer is choosen).

```
#lscan_string
#("Un matin il s'e'veille avec cette pense'e\L"^
# "Jeunesse aux jours dore's, je t'ai donc de'pense'e !\L"^
# "V. Hugo")
#["je";"tu";"il"];;
(9,"il") : (num * string)

#lindex_string
#("La transitivite' se de'finit par la proprie'te' \L"^
# "qu'a' un proce's, e'voque' par un verbe, d'e^tre de'crit \L"^
# "alternativement du point de vue du sujet (agent) et du point \L"^
# "de vue de l'objet (patient). (Wagner et Pinchon)")
#["le";"la";"\L";"un";"de";"du"] 0;;
(20,"de") : (num * string)
```

♣
```
prefix_index_string :
(string -> string list -> num -> num * string)
prefix_string : (string -> string list -> num * string)
find_which_string :
(string list -> string * num -> num * string)
```

- `prefix_index_string s sl n` returns the pair of a string and its length. This string is the longest prefix of `s` equal to a string of `sl`, starting from `n`. If there is no such prefix it returns the pair `(0,"")`.

- `prefix_string s sl n` returns the pair of a string and its length. This string is the longest prefix of `s` equal to a string of `sl`.
- `find_which_string sl (s,n)` finds which string of the list of strings `sl` matches `s` starting from character `n` .

```
#prefix_index_string
# "Mais une exception n'infirme pas une re'gle elle la confirme."
# ["Wagner et Pinchon";"Grammaire du francais";"P 284";
#  "con";"infirme";"confirme";"exc";"excep";"exception"]
# 9;;
(9,"exception") : (num * string)

#find_which_string
# ["vous";"obligera";"o";"obli";"obliger"]
# ("Je vous obligerais bien a' lire cette ligne !",8);;
(8,"obligera") : (num * string)
```

Chapter 28

Interactivity

This package gives some primitives to deal with interactivity in CAML.

28.1 Asking questions

♣ `ask : (string -> string)`
`repeatedly_ask : ((unit -> 'a) -> (unit -> 'b) -> 'b)`

- `ask s` will display `s` on the terminal and then wait for an answer which is returned as an uninterpreted string.
- `repeatedly_ask display_fun ask_fun` repeats a question until `ask_fun` succeeds (screen is redisplayed after 3 consecutives errors).

28.2 Reading answers

♣ `get_ans : (string -> (string -> 'a) -> 'a)`
`get_positive_num : (unit -> num)`
`get_bounded_num : (num -> num -> num)`
`get_num : (unit -> num)`
`get_num_list : (unit -> num list)`
`get_num_if : ((num -> 'a) -> 'a)`

- `get_ans prompt acceptor` will display `prompt` and then passe the answer to `acceptor` which is supposed to verify that the answer is correct.
- `get_num ()` will fail if the answer is not a number.
- `get_positive_num ()` will fail if the answer is not a positive number.

- `get_bounded_num from to` will fail if the answer is not into the range `from,to`.
- `get_num_if cond` will get a number if some condition is verified by the answer.
- `get_num_list ()` will ask for a list of numbers separated by commas.

28.3 Menus

♣
```
menu : (string -> string list -> 'a list -> 'a)
display_menu : (string -> string list -> num)
```

- `display_menu title optionl` displays a menu using the string `title` as title and the list `optionl` as list of options.

```
#display_menu "Mon menu" ["toto";"tata";"titi"];;

Mon menu

<1> : toto
<2> : tata
<3> : titi
3 : num
```

- `menu title optionl commandl` displays a list of options and chooses a command to be executed (or returns an element of `commandl` whatever it will be : thanks to polymorphism !).

Chapter 29

Automata

This package provides a kit to build file scanners. It is still under design but may be very useful as soon as you have to operate on file from inside the CAML system. It is a very convenient alternative to use "automat" instead of a new complete program using the stream features of CAML, since you have not to "rediscover the powder" (see example at the end of the chapter describing the stream system). Be aware of the size of this package (it is rather large), and thus it needs some time to load it.

We start by a brief description of the fonctionalities, and then we give detailed explanations and examples.

29.1 Scanning automata

In this package an "automaton" is just a file scanner, that is a list of commands which are repeatedly executed until the end of the file is reached.

While an "automaton" is running through the file, it maintains some global variables:

- the "pattern found": the last search string found by a command.
- the "character found": the last character found by a command which searches characters.
- the "accu": a string buffer, where the user may accumulate strings found during the scanning.
- an internal mark on the file, which is moved when searching patterns.

A command is either:

1. a scanning command (an action which is performed while looking for some string patterns)
2. a simple command (which performs no scanning: the mark does not move).

29.1.1 Scanning commands

A scanning command has two parts: a scanning order which precises what has to be searched, and a scanning action which is performed while executing the scanning order. For example:

```
do_until_word "bar" copy
```

The scanning order is `do_until_word "bar"`, and the scanning action is `copy`. The semantics is that the input of the automaton will be copied into the output, until the word `bar` is encountered.

Scanning orders are prefixed by `do_`, scanning actions are not.

Main scanning actions are:

- `copy`
- `ignore`
- `accumulate`

There are several families of scanning orders, acting respectively on characters (e.g. `do_until_char`), simple words (e.g. `do_until_word`), or list of words (e.g. `do_until`).

29.1.2 Simple commands

Simple commands may act either:

- On the accumulator (e.g. `copy_accu`).
- On the pattern found (e.g. `copy_pattern`).
- On an immediate string argument (e.g. `copy_string`).

Simple commands may be more complex, and take functions or scanning actions as argument. For instance `do_thru_pattern ignore`, will advance the mark after the pattern found.

An important example is the command `do_when_found`: it applies the last string pattern found by the automaton to a function which will perform other commands depending on the pattern found. For instance

```
do_when_found
 (fun "a" -> copy_string "aa"
   |  "b" -> do_thru_pattern ignore
   |   _  -> copy_pattern)
```

will copy the string `aa` into the output if the pattern found is `a` (and the mark is not moved), will ignore the pattern if it is `b`, and otherwise will copy it.

All these commands, scanning orders and action are detailed below.

29.2 Primitives provided to built automata

An automaton is build from a list of functions to be applied repeatedly in turn, while scanning a file. The functions are scanning actions involving patterns to be recognized in the file and actions (which are still functions) to be performed when (or until or before) a pattern is found.

♣
```
make_automaton :
((unit -> 'a) list -> in_channel * out_channel -> unit)
make_automaton_with :
((unit -> unit) list -> (unit -> 'a) list ->
 (unit -> unit) list -> in_channel * out_channel -> unit)
```

`make_automaton` is the standard function to build automata: it takes a list of actions to execute in turn (and *repeatedly*) while scanning and returns an automaton. This automaton will repeatedly perform the list of actions until it reaches the end of the file. For user's convenience `make_automaton_with` takes as arguments a list of actions to be performed at the beginning of the file (i.e. before starting to scan the file), a list of rules, and a list of actions to perform at the end of the file. The scanning rules may involved characters (strings of length 1), words (i.e. a single string) or patterns (i.e. list of strings matched in parallel).

Beware :

-- Due to the way automata are implemented **only one** automaton can be in use at any time during a session.

-- At the beginning you may be rather confused by the messages given by the CAML typechecker while writing automata: a good way to get better informations is to evaluate at the beginning of your session a declaration like the following one (evidently, suited to your own taste):

```
type command == void -> void
and 'a action == 'a -> void
and 'a rule == void -> 'a
and 'a rules == ('a rule) list
and automaton == (instream & outstream) -> void;;
```

Now for instance:

```
#make_automaton;;
- : ('a rules -> automaton)

#make_automaton_with;;
- : (void rules -> 'a rules -> void rules -> automaton)

#let rec find_string =
# [do_thru_char "\"" ignore; copy_string "\"";
```

```
#  do_copy_string; find_string];;

ill-typed phrase :
[(do_thru_char "\"" ignore); (copy_string "\"");
 do_copy_string; find_string]
has an instance of type : void rules
which should match type : command in
let rec find_string =
    [(do_thru_char "\"" ignore); (copy_string "\"");
      do_copy_string; find_string]
1 error in typechecking

Typecheck Failed
```

Instead of the message:

```
ill-typed phrase :
[do_thru_char "\"" ignore; copy_string "\"";
 do_copy_string; find_string]
has an instance of type : (void -> void) list
which should match type : (void -> void) in
let rec find_string =
 [do_thru_char "\"" ignore; copy_string "\""; do_copy_string;
  find_string]
1 error in typechecking

Typecheck Failed
```

Beware:
-- All well typed phrases are not always suitable for automata (as for all other CAML programs) : you should be careful of the evaluation order. For instance:

```
#let auto = make_automaton_with
#           [K (message"Copying ...")]
#           [do_to_end_of_file copy]
#           [K (message"Done !")];;
Copying ...
Done !
Value auto = <fun> : (in_channel * out_channel -> unit)
```

The application of the combinator K ensures a good typechecking, since

```
#K ();;
<fun> : ('a -> unit)
```

But the side effects are done at the building phase of the automaton, when evaluating the arguments of K, which (presumably) is not what the user intends to get,

since the message will be displayed not when the automaton is used but when it is built.

To correct this you may use:

```
#let auto = make_automaton_with
#           [do_act message "Copying ..."]
#           [do_to_end_of_file copy]
#           [do_act message "Done !"];;
Value auto = <fun> : (in_channel * out_channel -> unit)
```

29.3 Scanning chars

We may scan the input to find a character belonging (or not belonging) to a given string.

♣
```
do_until_char : (string -> ('a -> 'b) -> 'a -> unit)
do_while_char_until : (string -> ('a -> 'b) -> 'a -> unit)
do_thru_char : (string -> ('a -> 'b) -> 'a -> unit)
do_while_char_thru : (string -> ('a -> 'b) -> 'a -> unit)
```

The general idea is that (for instance) `do_until_char str act` performs `act` on the input until one of the characters belonging to the string `str` is found. Then this character is the "character found". In this package "thru" stands for: do `act` to just *after* the character found. "until" is the converse: `act` is executed until just *before* the character found.

"while" stands for the negation of until (something like until *not* the following pattern). So, knowing that `ignore` is the action that does nothing, `do_while_char_thru "ab" ignore` will do nothing while the characters are either an "a" or a "b" and the following character will be ignored too.

♣
```
do_on_char : (('a -> 'b) -> 'a -> unit)
do_skip_chars : (num -> ('a -> 'b) -> 'a -> unit)
```

- `do_on_char` skips the following char and performs `action`.
- `do_skip_chars n action` will do `action` after skiping the following `n` chars.

29.3.1 Scanning words

With these functions the input is matched against a single string (we call it a word). To satisfy this kind of pattern all the characters of the word must be present consecutively in the input in the same order.

♣ `do_until_word : (string -> ('a -> 'b) -> 'a -> unit)`
`do_thru_word : (string -> ('a -> 'b) -> 'a -> unit)`

`do_until word act`,`do_thru word act` stop to process the input with `act` before `word` (until) or after `word` (thru) is encountered. These functions update the "pattern found" with the word they have found. Algorithm used is:

1. Find the first character of `word`.
2. Ensure that at least enough chars remain in the buffer (`length_string word` characters)
3. Find if `word` is present at the current character position.

29.3.2 Operating on string patterns

♣ `do_until : (string list -> ('a -> 'b) -> 'a -> unit)`
`do_thru : (string list -> ('a -> 'b) -> 'a -> unit)`

These are analogous to the functions on words and characters, but they may be used for a list of strings. They find the longest possible string belonging to their string list argument, as soon as a prefix of one of these is detected in the input. When they have found a string matching one of the strings of the list, "pattern found" is updated with this string. Example `do_thru ["oui";"\Lnon";"oui\L"]` will find the first occurrence of one of the following: either the string `non` at the beginning of a line, or the string `oui` anywhere in the file, or the string `oui` ending a line. If the first occurrence of one of these strings is `oui` and if this string ends the line then the end-of-line will be in the "pattern found".

♣ `do_until_match :`
`((string * (string -> 'a)) list -> ('b -> 'c) -> 'b -> 'a)`
`do_thru_match :`
`((string * (string -> 'a)) list -> ('b -> 'c) -> 'b -> 'a)`

These are analogous to `do_until` and `do_thru`, but a function is associated with each pattern. When a pattern has been found, mark is set (before or after the pattern according to the meaning of `until` and `thru`), action is performed, and the function associated with the pattern found is applied to the pattern. In short

```
do_until_match
 ["a", (fun x -> action1);
  "b", (fun x -> action2)] scanning_act
```

is a short hand for the sequence of commands:

```
do_until ["a";"b"] scanning_act;
do_when_found
 (fun "a" -> action1
    | "b" -> action2)
```

♣ `do_find_pat_until :`
`(string list -> ('a -> 'b) -> 'a -> unit)`
`do_find_pat_thru :`
`(string list -> ('a -> 'b) -> 'a -> unit)`

Find which pattern of their string list argument matches the current position in the character input stream, and set the "pattern found" to the appropriate value (which may be "", if none of the given patterns matches).

29.4 Scanning Actions

When you have provided a pattern to the scanning functions, you must provide a "scanning action' i.e. what has to be done while looking for the pattern.

29.4.1 Common actions

♣ `ignore : ('a -> unit)`
`copy : (unit -> unit)`
`copy_char : (unit -> unit)`

The meaning of `ignore` is obvious: it is a "do nothing". `do_until_char "a" copy` will copy the input into the output stream of the automaton until just before the first a present in the input.

Function `copy_char` takes the next char from the input, puts the mark after it, and copies it into the output.

29.4.2 Accumulating

You may accumulate strings in the buffer provided by the package.

♣ `accumulate : (unit -> unit)`
`copy_accumulate : (unit -> unit)`

The first function stores the input in the buffer, the second one stores the input and copies it into the output stream.

♣
```
accu_flush : (unit -> string)
accu_dump : (unit -> string)
accu_clear : (unit -> unit)
```

`accu_dump` returns a string which is a copy of the content of the accumulator, `accu_clear` clears the accumulator, and `accu_flush` does the both.

♣
```
copy_accu : (unit -> unit)
copy_flush_accu : (unit -> unit)
```

`copy_accu` will copy the accumulator in the current output. `copy_flush_accu` does the same and flushes the accumulator as well.

Advanced users may notice that:

```
copy_flush_accu = copy_string o accu_flush
```

or (see the description of `do_seq` below)

```
copy_flush_accu = do_seq [copy_accu; accu_clear]
```

29.5 String Actions

You must provide string arguments (not found in the input of the automaton) to the following functions.

♣
```
copy_string : (string -> unit -> unit)
message_string : (string -> unit -> unit)
accumulate_string : (string -> unit -> unit)
copy_accumulate_string : (string -> unit -> unit)
```

29.6 Acting on pattern found

As soon as a scanning action has succeeded you may use the pattern found with the following functions.

♣
```
copy_pattern : ('a -> unit)
copy_char_found : ('a -> unit)
```

Of course they copy into the output stream the pattern or the character that has just been found.

♣
```
do_on_pattern : ((string -> 'a -> 'b) -> 'a -> 'b)
do_on_char_found : ((string -> 'a -> 'b) -> 'a -> 'b)
```

They perform an action that uses as first argument the pattern or character found.

♣
```
do_thru_pattern : (('a -> 'b) -> 'a -> unit)
do_thru_char_found : (('a -> 'b) -> 'a -> unit)
```

They skip the pattern or character found before performing an action.

Advanced users may notice the following:

```
do_on_pattern copy = copy_pattern
```

♣
```
do_when_found : ((string -> 'a -> 'b) -> 'a -> 'b)
do_when_char_found : ((string -> 'a -> 'b) -> 'a -> 'b)
do_when_ascii_found : ((num -> 'a -> 'b) -> 'a -> 'b)
```

They apply a CAML function, defined by cases, to the found pattern. For instance the following sequence:

```
do_until ["oui";"non"] accumulate;
do_when_found (fun "oui" -> accumulate_string " monsieur"
                | _     -> accumulate_string " madame")
```

will accumulate the string " monsieur" if the pattern "oui" has been found, and will accumulate the string " madame" otherwise.

29.7 To stop

These are functions used to stop processing (or to continue to the end with the same action).

♣
```
do_to_end_of_file : (('a -> 'b) -> 'a -> unit)
ignore_to_end_of_file : ('a -> unit)
copy_to_end_of_file : ('a -> unit)
```

These continue to do the corresponding actions until end of file is encountered.

```
let ignore_to_end_of_file = do_to_end_of_file ignore
and copy_to_end_of_file = do_to_end_of_file copy_rest;;
```

For instance to copy a file you just define:

```
let copy_file = make_automaton [copy_to_end_of_file];;
```

♣

```
do_end : ('a -> 'b)
```

`do_end` aborts processing.

29.8 Combining actions

♣

```
do_act : (('a -> 'b) -> 'a -> unit -> unit)
do_seq : (('a -> 'b) list -> 'a -> unit)
do_repeat : (('a -> 'b) -> 'a -> unit)
```

- you may apply some CAML function `f` with argument `arg` using `do_act f arg`.
- `do_seq action_list` is used to execute a sequence of actions.
- `do_repeat action` will repeat the action until the end-of-file is encountered.

One may be surprised that we use `do_act f arg` and not `f arg` to evaluate `f arg`. This is because we need some delay when building the automaton:

```
make_automaton
 [message"debut";
  ...
 ];;
```

The intention is certainly not that the evaluation of `message"debut"` occurs when the automaton is built, but when it runs. Hence the `do_act message "debut"`, which is evaluated when building but displays the message when running.

29.9 Using an automaton

Automata produced using `automat.ml` must be provided with a pair of streams (one input stream and one output stream). It is very important to define the automaton whithout these arguments, since the automaton is "compiled" using partial evaluation. Thus:

```
let echo_file s =
 make_automaton [copy_to_end_of_file] (open_in s,std_out);;
```

is not equivalent to

```
let echo_gen = make_automaton [copy_to_end_of_file];;
let echo_file s = echo_gen (open_in s,std_out);;
```

For so simple an example there is not a great difference, but in case of complex pattern matching the second version will be very much faster than the first, since all optimisations and analysis on patterns are done once for all.

29.10 Examples

29.10.1 Simple Examples

Here is a simple program to pick up CAML comments in a file: one skips the input *until* "word" (* is found, then copy *thru* *):

```
#(* find comments in a file (naive algorithm) *)
#let find_comment =
# make_automaton
#  [do_until_word "(*" ignore;
#   do_thru_word "*)" copy];;
Value find_comment = <fun> :
      (in_channel * out_channel -> unit)

#let comments_of s = find_comment (open_in s,std_out);;
Value comments_of = <fun> : (string -> unit)
```

This must be a little more complicated if one consider the second kind of comments (any sequence of characters surrounded by %):

- one must ignore the input *until* either (* or %.
- if the *pattern found* is (* we just copy *thru* the closing *), else we copy *twice* thru % (since dot is "*until*" the first % starting the comment).
- finally we copy a newline to get the comments on separated lines.

```
#(* find comments in a file (simple algorithm) *)
#let find_comment =
# make_automaton
#  [do_until ["(*";"%"] ignore;
#   do_when_found
#    (fun "(*" -> do_thru_word "*)" copy
#       | "%"  -> do_seq [do_thru_char "%" copy;
#                         do_thru_char "%" copy]);
#   copy_string "\L"];;
Warning: 1 partial match in this phrase
Value find_comment = <fun> :
      (in_channel * out_channel -> unit)

#let comments_of s = find_comment (open_in s,std_out);;
Value comments_of = <fun> : (string -> unit)
```

Beware:
-- It is very easy to build a looping automaton (i.e. an automaton which seems to do nothing): you must be careful to always advance the mark when the process goes on ... For instance this function also intended to *skip* all the old CAML comments (which are arbitrary sequences of characters surrounded by %) will loop on the first character % encountered in the file:

```
let skip_old_comments =
 make_automaton [do_until_char "%" copy;
                 do_until_char "%" ignore]
```

in effect, the two actions `do_until_char` do not pass through the pattern % they have found, since the first stops *before* the first % encountered in the input stream, and set the mark here. Then the second action is performed and it does nothing at all, since it finds at once a %, it resets the mark at the same place (and `ignore` will ignore an empty sequence of characters). Then the first action is redone and so on ...

One may be tempted to correct this with:

```
let skip_old_comments =
 make_automaton [do_until_char "%" copy;
                 do_thru_char "%" ignore]
```

but a little thought shows that this automaton just removes all the characters % found in the file scanned ...

To definitively correct this, you may use:

```
let skip_old_comments =
 make_automaton [do_until_char "%" copy;
                 (* To skip the "%" found by
```

```
                the preceding action *)
              do_thru_char "%" ignore;
              (* To skip all the stuff, including
                 the next character "%" *)
              do_thru_char "%" ignore];;
```

Or equivalently

```
#let skip_old_comments =
# make_automaton [do_until_char "%" copy;
#                 (* skip the "%" just found *)
#                 do_thru_pattern ignore;
#                 do_thru_char "%" ignore];;
Value skip_old_comments = <fun> :
      (in_channel * out_channel -> unit)
```

-- When writting an automaton you may notice that you still benefit from the CAML full functionality: a scanning action may have a functional parameter to achieve a greater degree of generality.

Thus `skip_old_comments` may be simply defined with:

```
#let do_on_comments action =
# [do_until_char "%" action;do_thru_pattern ignore];;
Value do_on_comments = <fun> :
      (('a -> 'b) -> ('a -> unit) list)

#let skip_old_comments =
# make_automaton (do_on_comments copy @ do_on_comments ignore);;
Value skip_old_comments = <fun> :
      (in_channel * out_channel -> unit)
```

29.10.2 Advanced Examples

Using this kind of automata this program finds the CAML strings in a file:

```
#(* Mark is supposed to be after the double quote beginning
#   the string.
#   Action is performed on all the following string including
#   the double quote ending the string *)
#let rec do_on_string action = do_seq
#    [do_thru ["\\\\";"\\\"";"\""] action;
#     do_when_found
#      (fun "\\\\" -> do_on_string action
#       |   "\\\"" -> do_on_string action
#       |   _ -> action)];;
Value do_on_string = <fun> : (('a -> unit) -> 'a -> unit)
```

```
#(* Now do_on_string may be used to copy or ignore a string *)
#let do_copy_string = do_on_string copy;;
Value do_copy_string = <fun> : (unit -> unit)

#let do_ignore_string = do_on_string ignore;;
Value do_ignore_string = <fun> : ('a -> unit)

#let rec find_string =
# do_seq ([do_thru_char "\"" ignore; copy_string "\"";
#          do_copy_string] @ [find_string]);;
Warning: unsafe recursive declaration
Value find_string = <fun> : (unit -> unit)

#let find_strings = make_automaton [find_string];;
Value find_strings = <fun> :
      (in_channel * out_channel -> unit)
```

In the same vein, here is a program which scans a CAML file, finding all the CAML phrases in turn, while copying comments between the phrases. Then an auxiliary function (not provided here) `pretty_string` is used to pretty-print the CAML phrase found:

```
let rec do_on_comment action = do_seq
   [do_thru ["(*";"*)";"%";"\""] action ;
    do_when_found
     (fun "(*" -> do_seq[do_on_comment action;do_on_comment action]
      |   "*)" -> action
      |   "%" -> do_seq[do_on_old_comment action;
                        do_on_comment action]
      |   _   -> do_seq[do_on_string action;do_on_comment action])]

 and do_on_old_comment = do_thru_char "%"

 and do_on_string action = do_seq
    [do_thru ["\\\\";"\\\"";"\""] action;
     do_when_found
      (fun "\\\\" -> do_on_string action
       |   "\\\"" -> do_on_string action
       |   _ -> action)]

and do_on_spaces = do_while_char_until " \T\L";;


let rec find_caml_phrase =
```

```
do_seq
[do_until ["(*";"%";"\"";";;"] accumulate;
 do_when_found
      (fun "(*" -> do_seq [do_thru_pattern ignore;
   do_on_comment ignore;find_caml_phrase]
       |   "%" -> do_seq [do_thru_pattern ignore;
  do_on_old_comment ignore;find_caml_phrase]
       |   "\"" -> do_seq [do_thru_pattern accumulate;
   do_on_string accumulate;find_caml_phrase]
       |   _    -> do_seq [do_thru_pattern accumulate;
                           accumulate_string "\L"])];;

(* To copy comments between phrases *)
let rec copy_comments = do_seq [
 do_on_spaces copy;
 do_find_pat_thru ["(*";"%"] copy;
 do_when_found
  (fun "(*" -> do_seq [do_on_comment copy;copy_comments]
    |  "%" -> do_seq [do_on_old_comment copy;copy_comments]
    |  _   -> ignore)];;

let pretty_phrase =
 [copy_comments;
  find_caml_phrase;
  pretty_string o accu_flush];;

let pp_file = make_automaton pretty_phrase;;

(* Auxiliary function to open properly the streams *)
let pretty_file_gen (s1,s2) =
 let is = open_in s1 in
 let os = try open_out s2 with _ -> close_in is reraise in
 message ("Processing "^s1);
 print_to os;
 pp_file (is,os);
 print_to std_out;
 message (s1 ^ " processed");;

(* The modified file gets extension ".pp" *)
let pretty_file s =
 pretty_file_gen ((s^".ml"),(s^".pp"));;
```

29.10.3 A part of the CAML "latex" automaton

Here we give another example which is taken from the CAML "latex" utility. All these functions are the scanning ones: their purpose is only to get a string which is then passed to a function which performs the appropriate action. For instance `treat_caml_include` will find in turn all the files which have to be included into the latex file and pass their names in turn to the function:
`caml_include : string -> void`
which copies the text of this file into the latex file.

Useless to say that this example is recursively defined with the chapter concerning the Latex Interface.

```
let rec treat_caml_include =
do_seq
[do_until ["\L";"\\end{caml_include}\L"] accumulate;
 do_when_found
  (fun "\L" -> do_seq [do_thru_pattern ignore;
                       caml_include o accu_flush;
                       treat_caml_include]
   | _ -> do_thru_pattern ignore)];;
```

Notice here the use of `do_seq`: `treat_caml_include` is a single action and may thus be recursively inserted in the list of actions to be performed.

```
(* To find a CAML phrase to be evaluated (quietly) *)
let rec treat_caml_eval =
do_seq
[do_until [";;\L";"\\end{caml_eval}\L"] accumulate;
 do_when_found
  (fun ";;\L" -> do_seq [do_thru_pattern accumulate;
                         caml_eval o accu_flush;
                         treat_caml_eval]
   | _       -> do_thru_pattern ignore)];;

(* Find a CAML phrase to be evaluated with its output reported *)
let rec treat_caml_example =
do_seq
[do_until [";;\L";"\\end{caml_example}\L"] accumulate;
 do_when_found
  (fun ";;\L" -> do_seq [do_thru_pattern ignore;
                         accumulate_string ";;";
                         caml_example o accu_flush;
                         treat_caml_example]
   | _       -> do_thru_pattern ignore)];;

let rec treat_caml_verify =
```

```
do_seq
[do_until [";;\L";"\\end{caml_verify}\L"] accumulate;
 do_when_found
  (fun ";;\L" -> do_seq [do_thru_pattern ignore;
 accumulate_string ";;";
         caml_verify o accu_flush;
         treat_caml_verify]
   |  _      -> do_thru_pattern ignore)];;

let rec treat_caml_type_of =
do_seq
[do_until ["\L";"\\end{caml_type_of}\L"] accumulate;
 do_when_found
  (fun "\L" -> do_seq [do_thru_pattern ignore;
       caml_type_of o accu_flush;
       treat_caml_type_of]
   |  _    -> do_thru_pattern ignore)];;


(* just copy ! (in the future may be more sophisticated) *)
let treat_caml =
 do_seq [do_until ["\\end{caml}\L"] copy;do_thru_pattern ignore]
;;

let treat_caml_primitive =
 do_seq [do_until_word "\\end{caml_primitive}\L" accumulate;
         do_thru_pattern ignore;
         caml_primitive o accu_flush]
;;

(* Mark is supposed to be after "\begin{caml" *)
let treat_caml_parts =
 [do_find_pat_thru
          ["_example}\L"; "_type_of}\L"; "_primitive}\L";
           "_eval}\L"; "_verify}\L";
           "_include}\L"; "}\L"]
  ignore;
  do_when_found
   (fun "_example}\L" ->
        do_seq [copy_string begin_verbatim;
                treat_caml_example;
                copy_string end_verb_newline]
    |  "_type_of}\L" ->
       do_seq [copy_string begin_verb_newline;
```

```
                treat_caml_type_of;
                copy_string end_verb_newline]
      |  "_primitive}\L" -> treat_caml_primitive
      |  "_eval}\L" -> treat_caml_eval
      |  "_verify}\L" ->
         do_seq [copy_string begin_verb_newline;
                treat_caml_verify;
                copy_string end_verb_newline]
      |  "_include}\L" ->
         do_seq [copy_string begin_verb_newline;
                treat_caml_include;
                copy_string end_verb_newline]
      |  "}\L" ->
         do_seq [copy_string begin_verb_newline;
                treat_caml;
                copy_string end_verb_newline]
      |   _   -> copy_string begin_caml);
  accu_clear];;

(* Copy the file until "\begin{caml", then treat this
   "CAML part" and loop *)
let latex_gen =
 let treat_file =
  [do_until [begin_caml] copy;
   do_thru_pattern ignore] @
  treat_caml_parts in
 make_automaton treat_file;;
```

Notice that in this example `begin_caml` is bound to the string `"\begin{caml"`, and the same for `begin_verbatim` (`\begin{verbatim}`) and `begin_verb_newline` (`\begin{verbatim}\L`). The careful reader may have noticed too the particular use of "let rec" in these examples ...

Chapter 30

Latex Interface

A simple interface between CAML and latex files has been written using the package `automat`. It is useful for people who need to write documentation based on CAML programs. This interface is a filter which produces a latex file, reading from another latex file (or in fact any file) and which allows automatic treatment of CAML parts in the later file.

30.1 CAML examples

30.1.1 Regular environment

Examples written in CAML, surrounded by `\begin{caml_example}` and `\end{caml_example}` are executed[1] and their output added to the file produced. Example:

```
\begin{caml_example}
let x =1;;
x;;
\end{caml_example}
```

becomes in the ouput file:

```
\begin{verbatim}

#let x =1;;
Value x = 1 : num

#x;;
1 : num

\end{verbatim}
```

Beware: -- Caml phrases must imperatively end with the exact sequence of characters `;;\n`. Otherwise the phrase is not recognised by the filter and is ignored.

[1] Evaluation is done in the global environment of the latex filter.

30.1.2 CAML examples "star"

There is an option to `caml_example` which is `caml_example*` with which the evaluated phrase is not copied into the file, but its result is.

30.1.3 CAML examples "double star"

This is the basic option, `caml_example**` which only evaluates a phrase for its side-effects (which are written into the file). Neither the phrase nor the result are copied into the file.

30.2 Single CAML phrase

A single phrase may be processed if enclosed between `begin{caml_phrase}` and `\end{caml_phrase}`. This phrase is treated as a "`caml_example`". This is useful when one wants to send a phrase to a parser:

```
\begin{caml_phrase}

parse_caml_syntax();;
1;;

\end{caml_phrase}
```

is then handled properly.

In addition "star" versions are provided (`caml_phrase*` and `caml_phrase**` with the same meaning as for `caml_example`).

30.3 CAML print it

The environment `caml_print_it` is used to print the value of the variable "`it`".

`caml_print_it*` is also available.

30.4 CAML eval

One may specify that a given expression must be evaluated quietly, i.e. without printing the results. One uses `\begin{caml_eval}` and `\end{caml_eval}`.

Example:

```
\begin{caml_eval}
load"mon_fichier";;
init();;
(* Don't forget that now 1 must be equal to 0, *)
(* in order to compile correctly \-expressions *)
hack();;
\end{caml_eval}
```

all this stuff will desappear, and will not be printed on the latex file produced, but CAML file `mon_fichier.ml` will have been loaded and phrase `init()` and `hack()` executed.

Environment `caml_eval` is different from the environment `caml_example**` since printing side-effects are done in the CAML toplevel environment (thus generally to the channel `std_out`).

30.5 CAML type of idents

The CAML system may find and record for you the type of a function or identifier using `\begin{caml_type_of}` and `\end{caml_type_of}`.

Example:

```
\begin{caml_type_of}
map
prefix +
\end{caml_type_of}
```

will produce

```
\begin{verbatim}

map : (('a -> 'b) -> 'a list -> 'b list)
prefix + : (num * num -> num)

\end{verbatim}
```

30.6 CAML verify

If you want to verify the syntax and the typechecking of your examples (but without execution), you use `\begin{caml_verify}` ... `\end{caml_verify}`

Example:

```
\begin{caml_verify}
let x = 1;;
\end{caml_verify}
```

becomes in the output file

```
\begin{verbatim}

let x = 1;;

\end{verbatim}
```

Notice that if an error occurred it will be printed in the file produced (except for "skippings" of parser which are displayed on terminal).

30.7 CAML

An alternative for verbatim could be `\begin{caml}` ... `\end{caml}`: for the time being `latex_file` will substitute `caml` by `verbatim`, but in the near future, your programs will be automatically pretty-printed, comments will be emphasized, CAML keywords will be printed with boldface, with user's keywords in blue, and so on ...

30.8 CAML include

If you need to include a CAML source file in your latex file, use `caml_include`
Example:

```
\begin{caml_include}
toto.ml
toto.mly
\end{caml_include}
```

becomes
`\begin{verbatim}`
There you find the text of the files `toto.ml` and `toto.mly`
`\end{verbatim}`

30.9 CAML syntax examples

In the environment `caml_user_syntax_example`:

- each phrase is echoed with a "`user_syntax_prompt`"
- both a prefix string and a postfix string are catenated to the phrase and sent to the CAML toplevel.

It can be used to treat user's examples running on a specialised toplevel with a user's defined syntax, setting:

```
user_syntax_terminator := "TERMINATOR";;
user_syntax_prefix := "user_toplevel(<:user_value_syntax<";;
user_syntax_postfix := ">>);;";;
user_syntax_prompt := "PROMPT: ";;
```

in a "`caml_eval`" latex environment.
For instance with the above evaluation, a phrase like

```
\x.xTERMINATOR
```

is sent to CAML as the phrase:

```
user_toplevel(<:user_value_syntax<\x.xTERMINATOR>>);;
```

The prompt used to print the input phrase in the latex file will be `PROMPT:` . That is to say the input phrase is displayed in the latex file as:

```
PROMPT: \x.xTERMINATOR
```

Here is an example from the CAML demo which begins this manual:

```
\begin{caml_eval}
user_syntax_prefix := "dc_top (<:Dc:Top_exp<";;
user_syntax_postfix := ">>);;";;
user_syntax_prompt := "-> ";;
user_syntax_terminator := "?";;
\end{caml_eval}

\begin{caml_user_syntax_example}
3 ?
x ?
let x=3 ?
let x=10 in x*x ?
x-x ?
let x=x+1 ?
let y = 11111111111111111111111111111111111111111+1 ?
y*1.5 ?
(* Exiting (notice that comments are still available) *)
.?
\end{caml_user_syntax_example}
```

30.10 Using the CAML latex filter

There are two primitives provided:

♣
```
latex_file : (string -> unit)
use_latex_file : (string -> unit)
```

- `latex_file "toto"` produces modified file `toto.ml.tex` reading from file `toto.tex`.
- `use_latex_file "toto"` will produced a new output file iff source file is more recent than object file (this is design to spare computation). Notice that if several files may share toplevel environment, `use_latex_file` is useless, since all files have to be processed in turn.

Bugs:

Everywhere a CAML phrase has to be read, the exact sequence of characters (";;"^ascii 10) is the supposed to end the phrase. So if this sequence is present in a string or in a comment the scanner may be confused ...

In order to turn these bugs into features, if you want to include this very sequence of characters in a string, write in your programs ;;\L and *not*

```
 ";;
"
```

The similar problem inside comments may be solved ending your comments just after the end of phrase mark ";;".

For example use:

```
\begin{caml_example}
(* now useless !
let x = 1;;*)
 etc ...
\end{caml_example}
```

and *not*:

```
\begin{caml_example}
(* now useless !
let x = 1;;
*)
 etc ...
\end{caml_example}
```

Keep in mind too, that `latex_file` needs the *exact* sequences of characters documented here: do not hope that `\begin{caml_example }` will be understood nor `\begin{caml_example}␣` ... This is partly designed to allow use of the filter on this very part of the documentation !

30.11 Mixing Latex and CAML in a file

It is possible to interweave CAML phrases and their documentation, in the spirit of WEB (D. Knuth). The same file may then be compiled as usual by the CAML system and processed by the following primitive to extract documenting parts from it. This allows you to maintain the documentation as closed as possible to the source code.

♣ `latex_caml_file : (string -> unit)`

A file may be processed by the function `latex_caml_file` provided that it is written according to the following format:

- The documenting parts have to be enclosed between the special *lines*:

  `(*\`

  and

  `\*)`

  (where the character "\" is supposed to stand for the l of "latex"!)

- In addition, you get the full capabilities of the `latex_file` filter inside the documenting parts.

- Caml phrases *outside of the documenting parts* are included into the output file, except if enclosed between special lines:

  `(*\begin{caml_ignore}*)`

  and

  `(*\end{caml_ignore}*)`

- Comments inside the CAML programs are ignored, unless written as documenting parts.

```
(*\
This will be included in the output file as a latex text.
\*)

(* This CAML comment will be ignored.
   The following CAML phrase is also included in the ouput *)
let x = 1;;
```

Here we give an example, which is the very file `user_prelude.ml` of the CAML library, and whose processed version is included in this manual.

```
(***********************************************************************)
(*                                                                     *)
(*                      Projet       Formel                           *)
(*                                                                     *)
(*                              CAML                                   *)
(*                                                                     *)
(***********************************************************************)
(*                                                                     *)
(*                              Inria                                  *)
(*                       Domaine de Voluceau                           *)
(*                       78150  Rocquencourt                           *)
```

```
(*                              France                                  *)
(*                                                                      *)
(************************************************************************)

(* user_prelude.ml General purpose functions                            *)
(*                 These functions are defined autoload                 *)
(*                 in the CAML system                                   *)

(*\
\chapter{User's prelude}

This file contains a set of general purpose functions, which has not been
 included in the very prelude of the language since these functions are
 not so often used.

\section{Iterators}
\index{Iterators}
\begin{caml_primitive}
it_map
set_extension
it_pair_list
\end{caml_primitive}

\begin{itemize}
\item \verb"it_map f g b l" ``it\_lists'' \verb"f" on the result of the
mapping of \verb"g" on the list \verb"l":
\par\noindent
\verb"it_map f g b l = it_list f b (map g l)"
\par\noindent
\item \verb"set_extension f l" builds a set from the results obtained
by the application of \verb"f" to the elements of \verb"l"
 (\verb"f" is supposed to produce lists).
\item \verb"it_pair_list f (l1,l2)" \verb"it_list"s \verb"f" on the list
of pairs  obtained from elements of \verb"l1" and \verb"l2".
\end{itemize}

\*)
let it_map f g = it_map_f_g
 where rec it_map_f_g a = fun [] -> a | (b::l) -> (it_map_f_g (f a (g b)) l);;

(* set_extension = - : (('a -> 'b list) -> 'a list -> 'b list) *)
(*let set_extension f = it_list (fun l x -> union l (f x)) [];;*)
let set_extension f = it_map union f [];;

let it_pair_list f init pair_list = it_list f init (combine pair_list)
;;

(*\
\par\noindent
Example:
```

```
\begin{caml_example}
set_extension (make_set o explode) ["language";"machine"];;
\end{caml_example}

\begin{caml_primitive}
num_map
\end{caml_primitive}
\verb"num_map f l" maps \verb"f" on \verb"l" providing to each call the
position in the list of the argument of the call:
\par\noindent
\verb"num_map f [e1; e2; ... ; ei; ... ; en] ="\\
\verb"          [f 1 e1; f 2 e2; ... ; f i ei; ... ; f n en]"
\par\noindent
Example:
\begin{caml_example}
num_map pair ["a";"b";"c"];;
num_map
 (fun i e ->
   print_string"The ";print_num i;print_string"th element is: ";
   message e)
 ["a";"b";"c";"d"];;
\end{caml_example}

\verb"num_map" may be defined with
\begin{caml}
let num_map f = snd o fold consf 1
    where consf i x = (i+1,f i x);;
\end{caml}

But in fact \verb"num_map" is just a closure of \verb"map_i", since we have:
\begin{caml_example}
let num_map = (C map_i) 1;;
\end{caml_example}
\*)

(*|
Value num_map : ((num -> 'a -> 'b) -> 'a list -> 'b list)
{fold,o}
|*)
let num_map f = map_i f 1;;

(*\
\section{Merging and sorting lists}
\index{Merging and sorting lists}
\begin{caml_primitive}
merge
merge_num
sort_num
\end{caml_primitive}
```

```
\begin{itemize}
\item \verb"merge" is used to get the union of two sets represented as
 sorted lists.
\item \verb"merge_num" is specially devoted to lists of numbers.
\item \verb"sort_num" sorts lists of numbers in increasing order.
\end{itemize}
\*)
(*|
Value merge : (('a & 'a -> bool) -> 'a list -> 'a list -> 'a list)
|*)
let merge ord l1 l2 = (merge_ord (l1,l2) ? failwith "merge")
  where rec merge_ord = function
(h1::t1),(h2::t2) -> if h1 = h2 then h1::merge_ord (t1,t2)
             if ord(h1,h2) then h1::merge_ord(t1,(h2::t2))
                           else h2::merge_ord((h1::t1),t2)
| [],l -> l
| l,[] -> l;;

(*|
Value merge_num : (num list -> num list -> num list)
{merge}
|*)
let merge_num = merge (prefix <);;

let sort_num = sort (prefix <);;

(*\
\section{Association lists}
\index{Association lists}
\begin{caml_primitive}
inverse_assoc
\end{caml_primitive}

\begin{itemize}
\item \verb"inverse_assoc" built the inverse a table of association.
\end{itemize}
\*)

let inverse_assoc = map (function (x,y) -> (y,x));;

(*\begin{caml_ignore}*)
(*\
\section{Removing the last elements of a list}
\begin{caml_primitive}
except_last_n
\end{caml_primitive}

\begin{itemize}
\item \verb"except_last_n n l" returns the elements of the list \verb"l" the
\verb"n" last being excepted.
```

```
\end{itemize}

\*)
(*
let except_last_n n l =

 type 'a num_or_list = Int of int | List of 'a list in

  let rec except_aux n = function
    [] -> Int #0
  | (x::l) -> match except_aux n l with
                Int p -> if p == n then List [x]
                         else Int (succ_int p)
              | List l -> List (x::l) in
 (match
   except_aux (int_of_num n) l
  with Int _ -> failwith "except_last_n"
     | List l -> l)
;;
*)
(* latex_caml_file has to ignore this definition since end_it_list is
   documented in the prelude *)
(*|
Value end_it_list : (('a -> 'a -> 'a) -> 'a -> 'a list -> 'a)
{it_list}
|*)
let end_it_list f default = fun [] -> default | (b::l) -> it_list f b l;;
(*\end{caml_ignore}*)
```

Part V

Miscellaneous

Chapter 31

Pitfalls and known bugs

31.1 Syntactic problems

31.1.1 Comments

Remember that comments are processed by the lexical analyser in order to be able to suppress (almost) arbitrary pieces of code. Thus inside comments strings and object languages parts are recognized and skipped properly.

For beginners this could lead to errors as in:

```
(* In CAML strings begin and end with " *)
let s = "yes";;
```

or

```
(* Inside strings, " is written \" *)
let s' = "A doublequote: \"";;
```

where the comments never end.

Known bug: when object language pieces of syntax are inside comments and contain themself special comments (as set by a comment delimitor order in the header of a grammar definition — see Object Language Parsing chapter) the special comments may not contain the string >>.

31.1.2 Keywords

It is not possible to use as an identifier one of the keywords of CAML: e.g.

```
#let lazy = false;;

line 1 Syntax error:
Skipping: lazy = false ;;
Parse Failed
```

The exhaustive list of the keywords of the language is given at the end of this chapter.

By the way the two values `true` and `false` belong to a special lexical class. Thus it is not possible to use them as ordinary identifiers.

31.1.3 Toplevel keywords

Toplevel keywords are keywords of the toplevel grammar. The toplevel grammar is the grammar which parses a toplevel CAML phrase. It just decides what kind of phrase it is given looking at the first token and then calls a specific parser. The list of toplevel keywords is given at the end of this chapter.

A toplevel keyword has a special meaning when it begins a toplevel phrase but has no special meaning inside an expression or a declaration except if it is also an ordinary keyword.

For instance, we can write:

```
#let module = ();;
Value module = () : unit
```

but

```
#module;;

line 1 Syntax error:
Skipping: ;;
Parse Failed
```

is supposed to enter the syntax of a module import. Though "end" is a toplevel keyword:

```
#let end = quit;;

line 1 Syntax error:
Skipping: end = quit ;;
Parse Failed
```

since "end" is also an ordinary keyword.

31.1.4 Infixes

Infix identifiers are treated in a special way by the parser. When you need to use them in prefix position remember to prefix them with the keyword `prefix`. You may use them as ordinary identifiers, even in type declarations:

```
##infix plus;;
Ident plus is now parsed as an infix
Directive () : unit
```

```
#type number = prefix plus of num & num;;
Type number defined
     prefix plus : (num * num -> number)

#1 plus 2;;
(1 plus 2) : number

##infix arrow;;
Ident arrow is now parsed as an infix
Directive () : unit

#type 'a arrow 'b == 'a -> 'b;;
Type ('a arrow 'b) abbreviates ('a -> 'b)

#I;;
<fun> : ('a arrow 'a)
```

Thus a predefined infix identifier may be used as a type constructor as well:

```
#type ('a,'b) prefix + = Inl of 'a | Inr of 'b;;
Warning: type + redefined
Type + defined
     Inl : ('a -> ('a + 'b))
   | Inr : ('a -> ('b + 'a))
```

31.1.5 Closing syntactic constructions

Some CAML syntactic constructions are open and then closed with special keywords. For instance, the `while` ... `do` ... `done` construction is open with `while` and closed with `done`. In contrast, the `if` ... `then` ... `else` ... construction is not closed.

There are implicite rules, used by the CAML parser to close open constructions. Nevertheless you can explicitly close a construction using parentheses:

```
(if true then 1 else 2);3;;
```

CAML syntactic constructions which are not explicitly closed generally tend to spread *as long as possible.* This is not true for the previous example, since the sequence has stronger precedence than the conditional (thus parentheses may be removed in the above example).

Common case is for instance:

```
if true then (if false then e1 else e2);;
```

which is equivalent to:

```
if true then if false then e1 else e2;;
```

Put it another way: an "else part" refers to the closest "if" without "else" part. This rule is followed by "match" and "try" constructions:

```
match E with
 p1 -> e1
|p2 -> match E' with
         p'1 -> e'1
       | p'2 -> e'2;;
```

the last "match case" `| p'2 -> e'2` refers to `match E' with`.

But be careful with the indentation, which is not understood by the parser, and may erroneously suggest that an open construction is closed:

```
match E with
 p1 -> e1
|p2 -> match E' with
         p'1 -> e'1
       | p'2 -> e'2
|p3 -> e3;;
```

is not equivalent to:

```
match E with
 p1 -> e1
|p2 -> (match E' with
          p'1 -> e'1
        | p'2 -> e'2)
|p3 -> e3;;
```

The rule is thus simply to parenthesize `match` and `try` constructs when they appear inside another `match` and `try` construct.

31.1.6 Reraising an exception

You can reraise an exception by the *syntactic* construct:

```
try f x with E -> e reraise;;
```

Note that you must write `e reraise` but not `e; reraise`.

31.1.7 "and" inside "where"

The `where` construction is hard to use inside large expressions, especially when it introduces many declarations.

Beginners should use it only in very simple cases, and to close it with parentheses, as in:

```
(E where x = e)
```

Pretty printing

As a matter of fact the system's pretty printer avoids `where` as much as possible. It is a good habit for beginners to use the pretty printer to get clearer programs, with significant indentation:

♣ `pretty : (unit -> unit)`

```
#pretty();;
#match E with
# p1 -> e1
#|p2 -> match E' with
#         p'1 -> e'1
#       | p'2 -> e'2
#|p3 -> e3;;

match E with
   p1 -> e1
 | p2 -> match E' with p'1 -> e'1 | p'2 -> e'2 | p3 -> e3

;;
```

31.2 Static binding

The "static binding" discipline of CAML may be surprising for the beginner. Consider:

```
#let rec fib = function
#  1 -> 0 | 2 -> 1 | n -> fib(pred n) + fib(pred(pred n));;
Value fib = <fun> : (num -> num)

#let double_fib x = (fib x) + (fib x);;
Value double_fib = <fun> : (num -> num)

#fib 5;;
3 : num

#double_fib 5;;
6 : num
```

Then the user understands that the definition of `fib` is erroneous. He corrects it with:

```
#let rec fib = function
#  1 -> 1 | 2 -> 1 | n -> fib(pred n) + fib(pred(pred n));;
```

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

#fib 5;;
5 : num
```

Now you can imagine that `double_fib` is corrected:

```
#double_fib 5;;
6 : num
```

it is not the case! The static binding discipline has definitively bound `double_fib` to the erroneous version of `fib`.

Nevertheless, for debugging purposes, you can get a flavour of dynamic binding using the primitive:

♣ `relet : (string -> string -> unit)`

For instance:

```
#let rec fib = function
#  1 -> 0 | 2 -> 1 | n -> fib(pred n) + fib(pred(pred n));;
Value fib = <fun> : (num -> num)

#let double_fib x = (fib x) + (fib x);;
Value double_fib = <fun> : (num -> num)

#let rec new_fib = function
#  1 -> 1 | 2 -> 1 | n -> new_fib(pred n) + new_fib(pred(pred n));;
Value new_fib = <fun> : (num -> num)

#relet "fib" "new_fib";;
() : unit

#fib 5;;
5 : num

#double_fib 5;;
10 : num
```

Beware:
-- This function is safe but *may fail* to change all the bindings done after the definition of the rebound identifier. This is clearly the case when rebinding a curried function which has already been partially applied to bind an identifier. Ror instance if g was defined by `let g = map succ;;` the evaluation of `relet "map" "new_map"` will not change the semantic of g (this is already true for the

`fib` example above, since if `x` was defined by `let x = fib 5` the "relet" of `fib` to `new_fib` does not change the value of `x` !). The `relet` primitive is still very useful during the debugging phase of a program.

31.3 Equal

♣ `equal : ('a * 'a -> bool)`

The primitive = may loop on cyclic values. In this case you should use `equal`. This function is autoloaded from the CAML library. It knows how to treat such complex data, but it is less efficient than = since it may perform much more computation to test if its two arguments are indeed isomorphic.

31.4 Typechecker messages

31.4.1 Strange messages

The abstract syntax of CAML does not reflect exactly the user's input. Thus the system pretty printer which works with the abstract syntax of a program, tries to figure out what the original program was. It is usually quite good for this job, but sometimes it is completely fooled by the syntax transformations done by the parser, such as for the `&` or `or` constructions, or the conditional with dangling `else`. The better example known of this fact is due to Alain Laville:

```
#true & (true,true);;

ill-typed phrase, the constant false of type bool
cannot be used with type instance (bool * bool) in
true & (true,true)
1 error in typechecking

Typecheck Failed
```

Although "`false`" does not appear in the concrete syntax, it is generated in the abstract syntax:

```
#parse_string"true & (true,true);;\n";;
(ML
 (MLcond
  ((MLconst (mlbool true)),
   (MLpair
    ((MLconst (mlbool true)),(MLconst (mlbool true)))),
   (MLconst (mlbool false))))) :
MLsyntax
```

hence, the strange message of the typechecker.

31.4.2 Strange errors

Sometimes errors reported by the typechecker are rather surprising since they are inferred from previous errors :

```
#let l = ref ([] : foo list);;

line 1: unbound type foo in ([] : foo list)

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

Typecheck Failed
```

In this case the typechecker has tried to recover from the first error to continue the typechecking in order to report other errors if any. Thus, having no informations at all about type `foo`, it uses `'a` for the type `foo`. But now the list `[]` becomes polymorphic and the identifier `l` refers to a polymorphic reference, which leads to an extra typechecking error.

This is not very common and in most cases all the reported errors are relevant to real bugs of the source text.

31.5 Interruptions

Interrupts seem to be ignored during garbage collection: in fact they are simply bufferized and taken into account *after* the garbage collection process is completely achieved.

However, it is not always advisable to interrupt the system, since when recovering from errors, it may be running some critical pieces of code when it has to undo some actions.

31.6 Formatting failures

The main reason why the formatting could fail: you ask for closing too many boxes. Namely it is 2 more than the number you have open: you have already closed the system's box (1 more) and try to close another one. This is easily recoverable, since it is a real bug of your printing routine.

31.7 System failures

The so-called "system" errors are are due either to the underlying operating system (which occur for instance when trying invalid I/O operations) or to the basic

primitives of the CAML system.

31.8 Compiler Warnings

31.8.1 Partial matches

In fun cases

When you write a definition by cases whose set of patterns is not exhaustive the compiler sends a warning and automatically adds an extra match case which catches all the forgotten cases to prevent from run time failures. Namely, it adds:

```
| _ -> raise match_failure
```

For instance :

```
let f = fun 1 -> true;;
```

is automatically completed and the compiled definition becomes

```
let f = fun 1 -> true | _ -> raise match_failure;;
```

In let

A very common source of partial matches is the "let" construction. For instance when you bind some variables to sub-patterns of a pattern, as in:

```
let (h::t) = l in ...
```

It is more surprising when `l` is a constant :

```
#fun x -> let l = [x] in let (h::t) = l in h;;
Warning: 1 partial match in this phrase
<fun> : ('a -> 'a)
```

Remember that this kind of let is automatically replaced by a match :

```
let pat = e in e'    is rewritten as    match e with pat -> e'
```

This explains the former partial match, since

```
match l with
 (h::t) -> h
```

does not care for the constructor `[]` of the type `list`.

By the way, notice that the type of non null lists may be

```
#type 'a non_null_list == 'a & 'a list;;
Type 'a non_null_list abbreviates ('a * 'a list)
```

since the second element of the pair may now perfectly be empty.

Now we write :

```
#fun x -> let l = x,[] in let (h,t) = l in h;;
<fun> : ('a -> 'a)
```

Beware:
-- It is a very bad programming habit to let partial matches in your programs, since the message you get in case of failure is not very informative. Moreover a partial match often reveals a really forgotten case.

31.8.2 Erroneous partial matches

Sometimes the compiler fails to detect that a match is exhaustive, and erroneously reports a partial match. This is a real (minor) bug of the CAML compiler, since the extra code added is in fact useless. Nevertheless the compiled code is correct, and the program still gets the proper semanctics:

```
#type complex = Real of num | Complex of num & num;;
Type complex defined
     Real : (num -> complex)
   | Complex : (num * num -> complex)

#let add_complexes =
#function Real(x),y -> "add a real to a complex or a real"
#       | x,Real(y) -> "add a complex to a real"
#       | Complex(x,y),Complex(u,v) -> "add a complex to a complex";;
Warning: 1 partial match in this phrase
Value add_complexes = <fun> : (complex * complex -> string)

#add_complexes (Real 1,Real 2);;
"add a real to a complex or a real" : string

#add_complexes (Real 1,Complex (1,2));;
"add a real to a complex or a real" : string

#add_complexes (Complex (1,2),Real 2);;
"add a complex to a real" : string

#add_complexes (Complex (1,2),Complex (1,2));;
"add a complex to a complex" : string
```

31.8.3 Unused match cases

When the compiler detects that the set of patterns of a match is exhaustive, it throws away all the extra cases and warns you:

```
#function [] -> 1
#      | x::l -> 2
#      |   _  -> failwith"Unknown case";;
Warning: 1 unused match case in this phrase
<fun> : ('a list -> num)
```

Very often this kind of warnings occurs for very large definitions when one adds an extra match case, as a life guard, to catch all possible forgotten cases, in order to report a message (as in: `| _ -> failwith"Forgotten case"`).

In this situation all is fine: the compiler just informs you that the match is in fact complete. But when this warning occurs with many unused cases, it is likely to be a real error in your program: the classical source of such a bug is to use an ill-spelled constructor in a pattern, since this constructor (or rather what you think to be a constructor!) is understood by the compiler as a variable name. Here is an example:

```
#type foo = Foo of num | Foo_Bar | Bar;;
Type foo defined
     Foo : (num -> foo)
   | Foo_Bar : foo
   | Bar : foo

#let check_foo = function
#   Foo _ -> true
# | Foo_bar -> false
# | Bar -> failwith"Bar is not appropriate in this case";;
Warning: 1 unused match case in this phrase
Value check_foo = <fun> : (foo -> bool)
```

In this definition, the second case catches all the other possible cases, since the constructor is named `Foo_Bar` and not `Foo_bar`. Thus `Foo_bar -> false` is equivalent to `x -> false`:

```
#check_foo (Foo 0);;
true : bool

#check_foo Bar;;
false : bool
```

31.8.4 Recursive definitions of non abstraction

When the compiler cannot *ensure* that it can safely generate code for a recursive declaration, it fails in normal mode with the following message:

```
#let rec x = x;;

Ill construction of a recursive value
```

However, you can *force* the compiler to accept such a declaration by saying:

```
##open compilation true;;
Directive () : unit
```

The compiler complains with a warning, but it will do its best. This warning indicates that you possibly have made a declaration which *cannot be compiled safely* by the CAML system. For instance:

```
#let rec x = x;;
Warning: unsafe recursive declaration
Value x = (* MONSTRE *) : 'a
```

Such a value of type 'a is obviously erroneous, so the pretty printer will insert the comment (* MONSTRE *) when it detects a monster. MONSTRE is an acronym for Mysterious Object with Non Significant Type Revealing an Error. Note that such monsters can only be created in open mode which is always unsafe. It is easy to produce a real error with a monster.

When you force the compilation of a recursive declaration, one cannot really predict what is going to happen, as shown by the following example:

```
#type bug = Bug of string | bug of string;;
Type bug defined
     Bug : (string -> bug)
   | bug : (string -> bug)

#let rec (Bug x) = Bug x;;
Warning: unsafe recursive declaration
Warning: 1 partial match in this phrase

Pattern matching Failed

#let rec (bug x) = bug x;;
Warning: unsafe recursive declaration

Ill construction of a recursive value
```

In the former case the pattern was tested before being built. In the latter, the loader, but not the compiler, detected that something was going to be wrong, and reported an error.

You may even create completely erroneous values:

```
#let rec x = [1] @ x @ [2];;
Warning: unsafe recursive declaration
Value x =
      [1;

Evaluation Failed: num_of_obj
```

But compare with:

```
#let rec x = [1] @ x in show x;;
Warning: unsafe recursive declaration
&1 where  &1 = (1 . &1)() : unit
```

Roughly speaking a recursive declaration is safe if you declare only functional values which are *syntactic* abstractions.

For instance you cannot recursively declare basic constants:

```
#let rec x = "o"^x;;
Warning: unsafe recursive declaration

System error: catenate <:obj<(())>>
```

Nor can you apply a function while defining it:

```
#let rec fact = function 1 -> 1 | n -> n * fact(n-1)
#and result = fact 10;;
Warning: unsafe recursive declaration
Caml : I quit on signal 7
```

Nevertheless in many cases, the code generated by the compiler will be correct, in particular when you declare some lists:

```
#let rec x = [1]
#and y = 1::x;;
Warning: unsafe recursive declaration
Value x = [1] : num list
Value y = [1; 1] : num list
```

One could think that the compiler should simply fail, but "recursive declarations of non abstraction" are still useful, and it seems better to allow them provided the user explicitly ask for unsafe compilation. We then hope that he will check that his declaration is in fact correct (for instance asking for the system to print the bindings of the recursively defined identifiers).

31.9 Termination

The CAML typechecker *does not* guarranty the termination of programs. Thus it is possible to compile a file *which cannot be loaded.*

We recall that there are 4 main sources of infinite loops in the language:

- "`let rec`" and "`while`" constructs:

  ```
  #let rec loop () = loop ();;
  Value loop = <fun> : (unit -> 'a)
  ```

This example may be much more complex. In particular:

```
let f =
 let rec loop x = let z = loop x in (fun y -> z) in
 (* Optimising using partial evaluation *)
 let y = loop 1 in
 fun () -> y;;
```

Identifier `f` is defined with a pure "`let`" construct. It has a functional type and is easily compiled. Nevertheless it cannot be evaluated.

- Mutable values:

```
#let loop =
# let f = ref (I:unit -> unit) in f := (fun x -> !f x);;
Value loop = <fun> : (unit -> unit)
```

`loop ()` fails to terminate.

- Values of *non positive* types. Non positive types are types τ which posseses constructors (or labels) with argument of type τ. This is more difficult to find out:

```
#type negative = Looping of negative -> negative;;
Type negative defined
     Looping : ((negative -> negative) -> negative)

#let mk_loop (Looping f) = f (Looping f);;
Value mk_loop = <fun> : (negative -> negative)

#mk_loop (Looping I);;
(Looping <fun>) : negative
```

The evaluation of `mk_loop (Looping mk_loop)` never terminates.

- Dynamic values fall into the preceding category. For instance:

```
#let f = function
# (dynamic (x:dyn -> dyn) as d) -> x d;;
Warning: 1 partial match in this phrase
Value f = <fun> : (dyn -> dyn)
```

The evaluation of `f (dynamic f)` loops!

Two extra sources of loops are:

- Forward values:

```
#forward f : unit -> unit;;
Forward f : (unit -> unit)

#let f () = f ();;
Value f = <fun> : (unit -> unit)
```

The definition of the body of the forward f introduces a loop.

- Autoload values. Consider the content of the file `essai_autoload.ml`:

```
autoload f : unit -> 'a from "essai_autoload";;
let f () = f ();;
```

After sucessful load of this file, `f ()` loops.

Beware:
-- The runtime system may get corrupted by certain non-terminating programs, or more generally by programs that exceeed the stack capacity. The result may be completely anomalous (illegal instruction, segmentation fault, unexpected return to Unix, impossibility to interrupt or even kill the process with normal kill signal (in this case, use `kill -9` followed by the process identification number), etc ...)

31.10 Undefined values

31.10.1 Undefined lazy values

As far as compilation is concerned, lazy values are similar to abstractions, so the compilation of a recursive lazy value is safe, but its execution may raise the special exception `undefined` or loop.

```
##open compilation false;;
Directive () : unit

#type 'a Freeze = lazy Freeze of 'a;;
Type Freeze defined
     Freeze : ('a -> 'a Freeze)

#let rec (Freeze x) = Freeze x;;

Undefined
```

31.10.2 Undefined forward

When you define a forward, it is bound to a special undefined value by the CAML system. Thus if you use it before having provided a real definition for the forward, a special exception is raised:

```
#forward foo : num -> num;;
Forward foo : (num -> num)

#(* Foo as not yet received a definition *)
#foo 1;;

Undefined forward foo

#(* This is not a real definition !
#   but once for all, it binds foo to ''undefined'' *)
#let foo = foo;;
Value foo = <fun> : (num -> num)

#foo 1;;

Undefined forward foo

#(* We bind g to foo *)
#let g = foo;;
Value g = <fun> : (num -> num)

#(* This does not define the forward value foo:
#   the forward foo is definitively unbound.
#   We just define a new foo global identifier *)
#let foo x = 1;;
Value foo = <fun> : ('a -> num)

#(* Old forward is still unbound *)
#g 1;;

Undefined forward foo
```

31.11 it

The keyword identifier it is always bound to the result of the last toplevel *expression* evaluated, but it is *not* rebound by a declaration:

```
#3;;
3 : num

#it;;
3 : num

#(* After this declaration which leads to evaluate (1+1) *)
```

```
#(* to bound the result to x, the value of it will still be 3 *)
#let x = 1+1;;
Value x = 2 : num

#(* The value of it remains 3 *)
#it;;
3 : num
```

Beware:
-- The keyword identifier "it" cannot be used in compiled files.

31.12 Type abbreviations

If you define polymorphic type abbreviations, be aware that the abbreviation mechanism always uses the most recent one when several of them may be applied to the same type:

```
#type 'a foo == 'a & num;;
Warning: type foo redefined
Type 'a foo abbreviates ('a * num)

#true,2;;
(true,2) : bool foo

#type 'a bar == bool & 'a;;
Type 'a bar abbreviates (bool * 'a)

#true,"CAML";;
(true,"CAML") : string bar

#(* The two rules may be applied, the last is choosen *)
#true,2;;
(true,2) : num bar
```

31.13 Relaxed typechecking

When an expression is included into a sequence of other expressions, the typechecker relaxes the normal type constraints, since the type of the whole expression is only determined by the last expression of the sequence (which is thus typechecked in the usual way).

Typechecking conditionals

Remember that both arms of a conditional must have the same type:

```
#if true then 1 else ();;

line 1: ill-typed phrase, the constant () of type unit
cannot be used with type instance num in if true then 1
1 error in typechecking

Typecheck Failed
```

This constraint is relaxed when a conditional appears inside a sequence, since the result is thrown anyaway:

```
#(if true then 1 else ());1;;
1 : num
```

Typechecking "match" ... "with" and "try" ... "with"

When used inside sequences of expressions the typing rules for `match ... with` and `try ... with` are relaxed:

```
#(match true with true -> 1 | _ -> ());2;;
2 : num

#(try 1 with
#   failure _ -> message "1"
# | io_failure _ -> 4);
#();;
() : unit
```

31.14 Large compilations

In case of compilation of very large CAML phrases, the compilation may suddenly become very slow. For instance, adding a single small line to a phrase as large as 100 lines may increase twofold the compilation time. This is not due to the CAML compiler itself but to the garbage collection: the number of cells required to build the abstract syntax of the phrase has become almost equal to the total amount of cells available in the core image. Then the system remains almost continuously garbage collecting (since each garbage collecting step recovers only a few cells).

In this case it is advisable to use a core image with a larger cell zone (use for example "caml 40").

31.15 Arithmetic

There is a strange "small integer" which is the successor of #32767. This integer is not properly printed and not properly handled by the generic operations (this is a real bug).

```
#succ_int #32767;;
##$8000 : int

#num_of_int it;;
#$8000 : num

#it + 65765;;

System error: abs <:obj<#$8000>>
```

31.16 Directives and pragmas

When the system has to "pseudo load" an already compiled file (when that file has been `use`'d, for example), if that file needs an autoload directive (`use`, `load_syntax` for example) then loading that directive may fail.

To get rid of that problem, make sure that such directives are loaded before compiling (this will be corrected in future versions).

31.17 Fatal errors

A fatal error is an unrecoverable error: the core image is usually definitively corrupted and you must leave the session.

31.17.1 Not enough room

List zone

When there is no more room in the cell zone you get the message:

```
***** Erreur fatale : zone des listes pleine.
```

or

```
***** Fatal error : no room for lists.
```

The core image you are using is then definitively corrupted. Thus you must quit the session.

The first point to check is that none of your routines is looping while building some infinite structures.

If it is not the case, you may try to run your program with a core image containing a larger cell zone, incrementing the numeric argument of the command calling CAML (for instance try "caml 40").

Heap zone

When the heap zone (where strings and vectors are stored) becomes full, you get

```
***** Erreur fatale : zone du tas pleine.
```

or

```
***** Fatal error : heap overflow.
```

If the problem is not implied by a looping routine and if it remains when you use a new core image then call a wizard: it is possible to make a new core image with a larger heap zone.

Code zone

When you have exhausted the code memory capabilities you get

```
***** Erreur fatale : zone du code pleine.
```

or

```
***** Fatal error : no room for code.
```

If this is due to multiple loading of the same files, you just have to quit and call a new core image.

If not, call a wizard again: it is possible to make a new core image with a larger code zone.

Stack

The stack may overflow or underflow with the following messages:

```
***** Erreur fatale : pile pleine.
***** Erreur fatale : pile pleine durant un GC.
***** Erreur fatale : pile vide.
```

or

```
***** Fatal error : stack overflow.
***** Fatal error : stack overflow during GC.
***** Fatal error : stack underflow.
```

This could be due to a looping program.

31.17.2 Memory inconsistencies

This error is fatal in the sense that it reveals that an object is either badly formed or stored in the wrong memory zone.

In this case the garbage collector usually detects the error afterwards and complains with the message:

```
* HEAP-OVNI *
```

where `OVNI` is the french acronym for "UFO".

Then leave the session and try to recompile your programs in a safe session, i.e. without using the `fast arith` nor `open compilation` modes. If you can recompile your programs and if the error is not corrected it is likely to be a CAML system's error, so please report it.

31.18 Reporting bugs

31.18.1 Potential bugs

These are all the other bugs! Be sure that it is not a strange feature described in this manual!

First verify that the bug remains in a safe mode, i.e. that you have not asked for open compilation directives by:

```
#open compilation true;;
```

or:

```
#fast arith true;;
```

or use `compil_fast` or `compile_fast` to compile one of your files. Note that it is not sufficient that the bug appear when the compilation is safe but in a session where compilation has always been safe, since a temporary unsafe compilation may introduce persistant monsters.

If the bug remains, please report it using the "bug report" document.

31.18.2 Real bugs

If you get a message

```
Error in CAML system: ...
```

you really have discovered a "real bug"! So fill in a copy of the "bug report" document you had with the distribution tape and send it to your CAML distributor or mail it at `caml@margaux.inria.fr` with a session where the bug occurs.

31.19 Suggestions

You may have some proposal to improve some features of the language or to add new capabilities. Then mail to
`caml@margaux.inria.fr` or
`...!mcvax!inria!margaux!caml`
with `suggestion` as subject.

You may have written some interesting and general purpose pieces of code worth including in the CAML users library. Then document your file in the "`latex_caml_file`" format and send it to `caml` with `user_lib` as subject.

31.20 List of CAML keywords

These keywords seem to be the source of so many syntactic problems for the beginners, that we recall them thereafter, in alphabetic order.

Notice that some of these keywords are not yet in use in the current system: they are simply already reserved for further extensions of CAML.

```
abstype     and         as          at
begin       case        continue    directive
do          done        dynamic     else
end         exception   fail        failwith
force       freeze      from        fun
function    future      if          in
it          lazy        let         match
mlet        mutable     not         of
or          parallel    prefix      protect
raise       rec         reraise     segment
strict      tags        then        try
type        value       vector      where
while       with
```

We list the symbols currently used in the system:

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

Finally we list the toplevel keywords:

```
#           #+          #-          autoload
end         forward     grammar     module
overload    printer     system
```

Chapter 32

Installing CAML

The installation of CAML supposes that your machine possesses the following directories:

`/usr/local` to put CAML

`/usr/local/bin` to put the `caml` command

The distribution tape contains directories `V2-6.1` (CAML) and `Exemples` (CAML examples).

V2-6.1 is divided in:

- the files used to create the initial core image of the CAML system (`camlisp` directory)
- object files of the CAML system (`lo` directory)
- object files of the CAML library (`lib` directory)
- file prelude.ml in `src`
- Yacc interface (`camlyacc` directory)
- CAML documentation (`doc`) and its `makefile`'s.

`Exemples` is composed of examples files (`Exemples/*.ml`).

Beware :

-- these files are not necessarily up-to-date (from a syntactical point of view)!! The reference should be instead prelude.ml (together with examples in the CAML Primer and the CAML Reference Manual).

To install CAML:

Read the tape on a disk where you have room enough to contain the whole system (10 Megabytes would be enough). If it is possible to do it in /usr/local, do it: installation will be a bit simpler. To do so, execute:

```
% cd /usr/local
% su (to be able to write in /usr/local)
Password:
```

```
# mkdir caml (or make a symbolic link to a directory
  on a partition where there is more room)
# chown <your usual login name> caml
# chgrp <your group> caml
# chmod 755 caml
# exit (or ^D) to quit su mode
% cd caml
% tar xv[OPTION]
% cd V2-6.1
```

The `tar xv` command will work on a VAX. For a SUN tape, execute `tar xvf /dev/rst0`.

Now, supposing that your machine type is `sun` (should be either `vax` or `sun`), just execute `./caml_install`: it will ask you some questions.

If you have read the tape in `/usr/local/caml` installation looks like:

```
% cd <where you read the tape (/usr/local/caml in this case)>/V2-6.1
% ./caml_install
For what machine (vax or sun; default is sun)?
Ok for sun
Put the caml command in (default is /usr/local/bin): /tmp
Moving old caml command into caml.old
System directory already installed.
Done.
In case of problem when trying to run caml, try:
cd /usr/local/caml/V2-6.1; make TYPEMACHINE=sun caml
% /tmp/caml
   CAML (sun) (V 2-6.1) by INRIA Fri Nov 24 1989

#quit();;
A bientot ...
```

If everything seems to work, installation is done. Otherwise, do the following (summarized by the last message of the `./caml_install` above):

- `cd /usr/local/caml/V2-6.1` and then
- either use the "make" command with "TYPEMACHINE=*yourmachine*" as first argument or
- – check the `TYPEMACHINE` variable of `Makefile` is your machine (should be either `vax` or `sun`)
  – then check the `TYPEMACHINE` variable of `camlisp/$TYPEMACHINE/Makefile` (should be the same in both files)
- then try:

```
% make
```

It takes quite a long time to build core images, so be patient.

Once CAML is installed, you might have to "rehash" to let your shell know how to access the "caml" command (will work only if you have the path of the caml command in your `PATH` environment variable). Do it by executing (csh and tcsh only):

```
% rehash
```

Emacs packages are not distributed.

In case of "shell problem" while installing CAML, it might be useful (depending on which version of make you are using) to set the global shell environment variable `SHELL` to `/bin/sh`.

In case of bug concerning CAML installation, fill in a copy of the "bug report" document you had with the distribution tape and send it to the company which distributes the CAML system.

Chapter 33

Appendix

We list here the source files of the grammars used in the CAML system.

33.1 CAML grammars

33.1.1 Toplevel grammar

```
(*************************************************************************)
(*                                                                       *)
(*                      Projet      Formel                               *)
(*                                                                       *)
(*                             CAML                                      *)
(*                                                                       *)
(*************************************************************************)
(*                                                                       *)
(*                             Inria                                     *)
(*                       Domaine de Voluceau                             *)
(*                       78150  Rocquencourt                             *)
(*                             France                                    *)
(*                                                                       *)
(*************************************************************************)

(* top_gram The grammar of the CAML toplevel                             *)
(*          Michel Mauny                                                 *)

(*          (Last edit date : Fri Feb 17 18:45:36 1989)                  *)

let parse_overload_binding _ = raise parse "overloading not yet supported";;

system module Top_gram;;

(*
   Warning: the parser of this grammar must be able to decide which rule to
   reduce by consulting only the first token!
*)
```

```
grammar for values Top =
delimitor
    string is "\""
    comment is "%"
;
precedences
    nonassoc INT FLOAT;

rule entry top =
    parse Literal "#" -> parse_directive()
        | Literal "#+" -> parse_plus_dir()
        | Literal "#-" -> parse_minus_dir()
        | Literal "module" -> parse_import()
        | Literal "end" -> parse_export()
        | Literal "system" -> parse_system_import()
        | Literal "autoload" -> parse_autoload()
        | Literal "grammar" -> parse_grammar_decl()
        | Literal "printer" -> parse_printer_decl()
        | Literal "forward" -> parse_forward_decl()
        | Literal "overload"
            -> MLdecl(MLoverload(prefix :: (parse_overload_binding())))
        | -> lex_reread_last_token();parse_caml()
;;

let parse_gen parse_top =
    fun () ->
        match sys_eval_syntax(parse_top())
        with dynamic (s: MLsyntax) -> s
           | _ -> system_error "Ill built syntax"
;;

let parse_gen_arg parse_top arg =
        match sys_eval_syntax(parse_top arg)
        with dynamic (s: MLsyntax) -> s
           | _ -> system_error "Ill built syntax"
;;

let parse_caml_syntax = parse_gen (Top "top").Parse
and parse_raw_caml_syntax = parse_gen (Top "top").Parse_raw;;


let parse_string = parse_gen_arg (Top "top").Parse_string
and parse_caml_syntax_in_channel = parse_gen_arg (Top "top").Parse_channel
;;

end module
 with value parse_caml_syntax
        and parse_string
        and parse_caml_syntax_in_channel
```

```
        and parse_raw_caml_syntax
;;
```

33.1.2 Internal grammar

```
(*************************************************************************)
(*                                                                       *)
(*                      Projet       Formel                              *)
(*                                                                       *)
(*                           CAML                                        *)
(*                                                                       *)
(*************************************************************************)
(*                                                                       *)
(*                           Inria                                       *)
(*                      Domaine de Voluceau                              *)
(*                      78150  Rocquencourt                              *)
(*                           France                                      *)
(*                                                                       *)
(*************************************************************************)

(* caml_gram The grammar of CAML itself                                  *)
(*           Michel Mauny (translation from old Yacc interface)          *)
(*           Ascander Suarez                                             *)
(*           Pierre Weis                                                 *)

system module caml_gram;;

#+MACROGRAM
let MLpair_to_pair =
    function (MLconst (mlstring s1), MLconst (mlstring s2)) -> (s1,s2)
           | _ -> raise parse "wrong call to an object grammar"
;;
#+MACROGRAM
let MLstring_to_string =
    function MLconst (mlstring s) -> s
           | _ -> raise parse "wrong call to an object grammar"
;;

#pragma let PARSE s=
 if MACROGRAM
 then
 (match s with
  MLpair (s1,MLconst (mlstring "")) ->
   <:CAML:Expr<
    ML_to_MML(do_out_of_system parse_ol_grammar (MLstring_to_string #s1, ""))>>
| MLpair _ ->
   <:CAML:Expr<
    ML_to_MML(do_out_of_system parse_ol_grammar (MLpair_to_pair #s))>>
| _ ->
   <:CAML:Expr<
```

```
    ML_to_MML(do_out_of_system parse_ol_grammar #s)>>)
 else
   <:CAML:Expr<do_out_of_system parse_ol_grammar #s >>
;;


#-MACROGRAM
let mk_type_binding (x,y) z = x,y,z;;

#-MACROGRAM
let MLtypes_of_vars = map (fun s -> MLvartyp s);;

#-MACROGRAM
let eval_to_btagl e =
 match eval_prag_syntax e
 with dynamic ([] : base_tag list) ->
     raise parse "Empty list of tags specified for an anonymous type at import"
    | dynamic (btagl : base_tag list) -> btagl
    | d -> ill_typed_macro e d <:gtype<base_tag list>>
;;

#-MACROGRAM
let eval_macro_expr_ML e =
 match eval_prag_syntax e
 with dynamic (prg : ML) -> prg
    | d -> ill_typed_macro e d <:gtype<ML>>
;;

#-MACROGRAM
let eval_macro_expr_MLdecl e =
 match eval_prag_syntax e
 with dynamic (prg : MLdecl) -> prg
    | d -> ill_typed_macro e d <:gtype<MLdecl>>
;;

#-MACROGRAM
let eval_macro_expr_string_list e =
 match eval_prag_syntax e
 with dynamic (l : string list) -> l
    | d -> ill_typed_macro e d <:gtype<string list>>
;;

#-MACROGRAM
let eval_macro_expr_bool test e_true e_false =
 match eval_prag_syntax test
 with dynamic (true:bool) -> e_true
    | dynamic (false:bool) -> e_false
    | d -> ill_typed_macro test d <:gtype<bool>>
;;
```

```
#-MACROGRAM
let eval_macro_expr_string e =
 match eval_prag_syntax e
 with dynamic (s:string) -> s
    | d -> ill_typed_macro e d <:gtype<string>>
;;

#-MACROGRAM
let eval_macro_expr_MLtype e =
 match eval_prag_syntax e
 with dynamic (mlty: MLtype) -> mlty
    | d -> ill_typed_macro e d <:gtype<MLtype>>
;;

#-MACROGRAM
let eval_macro_expr_MLpat_ML_list e =
 match eval_prag_syntax e
 with dynamic (l: (MLpat & ML) list) -> l
    | d -> ill_typed_macro e d <:gtype<(MLpat & ML) list>>
;;

#-MACROGRAM
let eval_macro_expr_MLconstruct_list e =
 match eval_prag_syntax e
 with dynamic (l: MLconstruct list) -> l
    | d -> ill_typed_macro e d <:gtype<MLconstruct list>>
;;

#-MACROGRAM
let eval_macro_expr_MLlabel_list e =
 match eval_prag_syntax e
 with dynamic (l: MLlabel list) -> l
    | d -> ill_typed_macro e d <:gtype<MLlabel list>>
;;

#-MACROGRAM
let eval_macro_expr_MLpat e =
 match eval_prag_syntax e
 with dynamic (prg: MLpat) -> prg
    | d -> ill_typed_macro e d <:gtype<MLpat>>
;;

#+MACROGRAM
let MLtypes_of_vars = map (fun s -> MLvartyp s);;

#+MACROGRAM
let MLpair_to_syntax =
    function MLpair (MLconst (mlstring s1),MLconst (mlstring s2)) -> s1,s2
           | _ -> system_error "wrong \"Syntax\" lexeme"
;;
```

```
#+MACROGRAM
let ML_to_MML (expr:ML) = MLquote (dynamic expr);;
#+MACROGRAM
let MML_to_MLpat =
    function MLquote(dynamic (expr: ML)) -> MLquote(dynamic(expr_to_pat expr))
           | _ -> system_error "MML_to_MLpat"
;;

grammar for #(if MACROGRAM then "programs" else "values")
            #(if MACROGRAM then "CAML" else "Caml") =

delimitors
    string is "\""
    comment is "%"
;
precedences
    right "->";
    right "or";
    right "&";
    right "<-" ":=";
    right ",";
    right INFIX;
    right "@" "^";
    right "::";
    left "+" "-";
    left "*" "/";
    precedence uminus;
    left ".";
    nonassoc "^}"
;


rule entry Caml =
    parse Caml_top t; Literal ";;" -> t

and Caml_top =
    parse Decl td -> MLdecl td
        | Expr e      -> ML e
       #|(if MACROGRAM then []
          else <:Gram:Rule_body<
          Directive d -> MLdirective d
        | Macro_decl md  -> MLpragma md
            >>)

#and
(if MACROGRAM then <:Gram:Rules< (* For programs case: new entry points *)
    entry MLnum =
        parse NUM n -> MLconst(mlnum n)
            | MLescape e -> MLconst(mlnum e)
```

```
and entry MLstring =
        parse STRING s -> MLconst(mlstring s)
            | MLescape e -> MLconst(mlstring e)

and entry MLbool =
        parse BOOL b -> MLconst(mlbool b)
            | MLescape e -> MLconst(mlbool e)

and entry MLint =
        parse INT i -> MLconst(mlint i)
            | MLescape e -> MLconst(mlint e)

and entry MLfloat =
        parse FLOAT f -> MLconst(mlfloat f)
            | MLescape e -> MLconst(mlfloat e)

and entry MLvar =
    parse MLIdent2 v -> MLvar v
        | MLescape e -> MLvar e
>>

else <:Gram:Rules<
     entry Pragma =
    parse Decl d; Literal ";;" -> Pragmadecl d
        | Expr e; Literal ";;" -> Pragmaexp e

and Directive = parse Literal "directive"; Expr e -> Pragmaexp e

and Macro_decl =
    parse Literal "mlet"; Val_binding vb -> Pragmadecl vb
        | Literal "mlet"; Literal "rec"; Val_binding vb
            -> Pragmadecl (MLrec vb)
>>)

and entry Expr0 =
    parse Ce c -> MLconst c
        | MLIdent2 x -> MLvar x
        | Constructor c -> MLconstr c
        | Literal "it" -> MLit
        | Ol_program ol -> ol
        | MLescape e -> (#(if MACROGRAM then <:Caml:Expr<e>>
                          else <:Caml:Expr<eval_macro_expr_ML e>>))

and Exp1 =
    parse Exp1 e; Literal "."; IDENT id -> MLrecord_access (id,e)
        | Exp1 e; Literal "."; NUM n
            -> MLvect_access (e,MLconst(mlnum n))
        | Exp1 e1; Literal "."; Literal "("; Expr e2; Literal ")"
            -> MLvect_access(e1,e2)
```

```
        | Expr0 e -> e

and Expr13 =
    parse Exp1 e1; Literal "."; IDENT v; Literal "<-"; Fnexpr13 e2
            -> MLupdate (v,e1,e2)
        | Exp1 e1; Literal "."; NUM n; Literal "<-"; Fnexpr13 e2
            -> MLvect_assign (e1,MLconst(mlnum n),e2)
        | Exp1 e1; Literal "."; Literal "("; Expr e2; Literal ")";
          Literal "<-"; Fnexpr13 e3
            -> MLvect_assign (e1,e2,e3)

and Ol_program =
    parse Literal "<<"; {#(PARSE <:Caml:Expr<!default_ol_grammar>>)} e;
          Literal ">>" -> e
        | Literal "<:"; IDENT gname; Literal "<";
          {#(PARSE <:Caml:Expr<(gname,"")>>)} e; Literal ">>" -> e
        | Literal "<:"; IDENT gname; Literal ":"; IDENT ext; Literal "<";
          {#(PARSE <:Caml:Expr<(gname,ext)>>)} e; Literal ">>" -> e

and Expr1 =
    parse Literal ("!" as bang); Expr1 e -> MLapply(MLvar bang,e,[])
        | Literal "dynamic"; Expr1 e -> MLdynamic e
(************
        | Literal "inline"; Expr1 e -> MLinline e
*************)
        | Exp1 e -> e

and Expr2 =
    parse Expr1 e;  (+ (parse Expr1 e -> e)) el -> MLapply(e,el)
        | Expr1 e -> e

and Expr8 =
    parse Expr8 e; INFIX i; Expr8 e_prime
            -> MLapply(MLvar i, MLpair(e,e_prime),[])
        | Expr8 e; Literal ("@" as a); Expr8 e_prime
            -> MLapply(MLvar a, MLpair(e,e_prime),[])
        | Expr8 e; Literal ("^" as c); Expr8 e_prime
            -> MLapply(MLvar c, MLpair(e,e_prime),[])
        | Expr8 e; Literal ("::" as dc); Expr8 e_prime
            -> MLapply(MLvar dc, MLpair(e,e_prime),[])
        | Expr8 e; Literal ("+" as p); Expr8 e_prime
            -> MLapply(MLvar p, MLpair(e,e_prime),[])
        | Expr8 e; Literal ("-" as m); Expr8 e_prime
            -> MLapply(MLvar m, MLpair(e,e_prime),[])
        | Expr8 e; Literal ("*" as st); Expr8 e_prime
            -> MLapply(MLvar st, MLpair(e,e_prime),[])
        | Expr8 e; Literal ("/" as d); Expr8 e_prime
            -> MLapply(MLvar d, MLpair(e,e_prime),[])
        | "-"; Expr8 e with precedence uminus
            -> (#(if MACROGRAM then <:Caml:Expr<MLapply(MLvar "(*-*)",e,[])>>
```

```
                else <:Caml:Expr<mkuminus e>>))
        | Expr2 e -> e

and Expr9 =
    parse Expr8 e; Literal ("=" as eq); Expr8 e_prime
            -> MLapply(MLvar eq, MLpair(e,e_prime),[])
        | Expr8 e; Literal ("<" as lt); Expr8 e_prime
            -> MLapply(MLvar lt, MLpair(e,e_prime),[])
        | Expr8 e; Literal (">" as gt); Expr8 e_prime
            -> MLapply(MLvar gt, MLpair(e,e_prime),[])
        | Expr8 e; Literal ("<=" as le); Expr8 e_prime
            -> MLapply(MLvar le, MLpair(e,e_prime),[])
        | Expr8 e; Literal (">=" as ge); Expr8 e_prime
            -> MLapply(MLvar ge, MLpair(e,e_prime),[])
        | Expr8 e; Literal ("<>" as diff); Expr8 e_prime
            -> MLapply(MLvar diff, MLpair(e,e_prime),[])
        | Expr8 e; Literal ("==" as eq); Expr8 e_prime
            -> MLapply(MLvar eq, MLpair(e,e_prime),[])
        | Expr8 e -> e

and Expr10 =
    parse Literal ("not" as n); Expr9 e -> MLapply(MLvar n, e,[])
        | Expr9 e -> e

and Expr12 =
    parse Expr12 e; Literal "or"; Expr12 e_prime
            -> MLcond(e, MLconst(mlbool true),e_prime)
        | Expr12 e; Literal "&"; Expr12 e_prime
            -> MLcond(e,e_prime, MLconst(mlbool false))
        | Expr10 e -> e

and Expr13 =
    parse Expr12 e; Literal ","; Fnexpr13 e_prime -> MLpair(e,e_prime)
        | Expr12 e -> e

and Expr14 =
    parse Expr13 e; Literal (":=" as ass); Fnexpr16 e_prime
            -> MLapply (MLvar ass,MLpair(e,e_prime),[])
        | "at"; MLIdent id; Literal "<-"; Fnexpr16 e -> MLreplace (id,e)
        | Expr13 e -> e

#and
(if MACROGRAM then <:Gram:Rules<
    Expr15 =
    parse Literal "raise"; Rs_expr e -> MLraise e
        | Literal "failwith"; Expr15 e -> MLraise ("failure",e)
        | Expr14 e -> e
>>
else <:Gram:Rules<
    Expr15 =
```

```
    parse Literal "raise"; Rs_expr e -> mkraise e
        | Literal "failwith"; Expr15 e -> mkraise ("failure",e)
        | Expr14 e -> e
>>)

and Rs_expr =
    parse Literal "("; Rs_expr e; Literal ")" -> e
        | MLIdent id; Fnexpr15 e -> (id,e)
        | MLIdent id -> (id, MLconst mlnull)

and Expr0 =
    parse Literal "fail" -> mkraise("failure", MLconst(mlstring "fail"))
        | Literal "continue" -> MLcontinue

        | Literal "["; Exprs e; Literal "]" -> MLlist e
        | Literal "[|"; Exprs es; Literal "|]"
            -> MLvect_of_list (MLmutable true, MLlist es)
        | Literal "[|"; Literal "|]"
            -> MLvect_of_list (MLmutable true, MLvar "")
        | Literal "[<"; Exprs es; Literal ">]"
            -> MLvect_of_list (MLmutable false, MLlist es)
        | Literal "[<"; Literal ">]"
            -> MLvect_of_list (MLmutable false, MLvar "")

and Exprs =
    parse Fnexpr16 e1; ( * (parse Literal ";"; Fnexpr16 e -> e)) el -> (e1,el)

and If_expr16 =
    parse Literal "if"; Expr test; Literal "then"; Fnexpr16 e_true; Cond c
            -> MLcond(test, e_true, c)

and Cond =
    parse Literal "if"; Expr test; Literal "then"; Fnexpr16 e_true; Cond c
            -> MLcond(test, e_true, c)
        | Literal "else"; Fnexpr16 e -> e
        |   -> MLconst mlnull

and Expr16 =
    parse If_expr16 e -> e
        | Expr15 e -> e

and Seqexpr16 =
    parse Seqexpr16 el; Literal ";"; Fnexpr16 e -> e::el
        | Fnexpr16 e -> [e]

and Expr17 =
    parse Expr16 e; Literal ";"; Seqexpr16 es
            -> (#(if MACROGRAM then <:Caml:Expr<MLseq(e::rev es)>>
                  else <:Caml:Expr<mkseq (e, es)>>))
        | Expr16 e -> e
```

```
and Fnexpr =
    parse Literal "fun"; Match m -> MLmatch m
        | Literal "function"; Umatch um -> MLmatch um
        | Literal "match"; Expr e; Literal "with"; Umatch um
            -> MLapply (MLmatch um, e, [])
        | Literal "case"; Expr e; Literal "of"; Umatch um
            -> MLapply (MLmatch um, e,[])
        | Literal "try"; Expr e; Literal "with"; Trymatch tm
            -> mkhandle (e, tm)
        | Decl d; Literal "in"; Expr e -> MLin (d,e)

and Fnexpr13 = parse Fnexpr e -> e | If_expr16 e -> e | Expr13 e -> e
and Fnexpr15 = parse Fnexpr e -> e | Expr15 e -> e
and Fnexpr16 = parse Fnexpr e -> e | Expr16 e -> e

and entry Expr =
    parse Literal "lazy"; Expr e -> MLeval_mode (Lazy,e)
        | Literal "freeze"; Expr e -> MLeval_mode1 (Lazy,e)
        | Literal "parallel"; Expr e -> MLeval_mode (Parallel,e)
        | Literal "future"; Expr e -> MLeval_mode1 (Parallel,e)
        | Literal "strict"; Expr e -> MLeval_mode (Strict,e)
        | Literal "force"; Expr e -> MLforce e
        | Literal "protect"; Expr e -> MLprotect e
        | Expr17 e1; Literal "?"; Expr e2
            -> MLhandle(e1,[(MLconpat("failure", MLwildpat)),e2])
        | Expr17 e; Literal "where"; Val_binding vb -> MLin(vb, e)
        | Expr17 e; Literal "where"; Literal "rec"; Val_binding vb
            -> MLin(MLrec vb, e)
        | Literal "vector"; Expr e_num; Literal "of"; Expr2 e_init
            -> MLinit_vect (MLmutable true,(e_num,e_init))
        | Literal "vector"; Literal "of"; Expr2 e_list
            -> MLvect_of_list (MLmutable true,e_list)
        | Literal "segment"; Expr e_num; Literal "of"; Expr2 e_init
            -> MLinit_vect (MLmutable false,(e_num,e_init))
        | Literal "segment"; Literal "of"; Expr2 e_init
            -> MLvect_of_list (MLmutable false,e_init)
        | Fnexpr e -> e
        | Expr17 e -> e

and Expr0 =
    parse Literal "("; Expr e; Literal ")" -> e
        | Literal "("; Expr e; Type_constraint str; Literal ")"
            -> MLstraint(e, str)
        | Literal "while"; Expr e_test;
          Literal "do"; Expr e_loop_step; Literal "done"
            -> MLwhile(e_test, e_loop_step)
        | Literal "begin"; Literal "do"; Expr17 e; Literal "end"; Literal "do"
            -> e
        | Literal "begin"; Literal "while"; Expr e_test; Literal "do";
```

```
          Expr e_loop_step; Literal "end"; Literal "while"
            -> MLwhile(e_test, e_loop_step)
        | Literal "begin"; Literal "fun"; Match m; Literal "end"; Literal "fun"
            -> MLmatch m
        | Literal "begin"; Literal "function"; Umatch um;
          Literal "end"; Literal "function"
            -> MLmatch um
        | Literal "begin"; Literal "match"; Expr e; Literal "with"; Umatch um;
          Literal "end"; Literal "match"
            -> MLapply(MLmatch um, e, [])
        | Literal "begin"; Literal "case"; Expr e; Literal "of"; Umatch um;
          Literal "end"; Literal "case"
            -> MLapply(MLmatch um, e, [])
        | Literal "begin"; Literal "try"; Expr e; Literal "with"; Trymatch tm;
          Literal "end"; Literal "try"
            -> mkhandle (e, tm)
        | Literal "begin"; Literal "if"; Expr e_test;
          Literal "then"; Expr e_true;
          Literal "else"; Expr e_false; Literal "end"; Literal "if"
            -> MLcond(e_test, e_true, e_false)
        | Literal "begin"; Literal "if"; Expr e_test;
          Literal "then"; Expr e_true; Literal "end"; Literal "if"
            -> MLcond(e_test, e_true, MLconst mlnull)
        | Literal "{"; Field_list fl; Literal "}" -> MLrecord fl

and Field_list =
    parse Field1 f1; ( * (parse Literal ";"; Field1 f -> f)) fl -> f1,fl

and Field1 =
    parse MLIdent id; Literal "="; Fnexpr16 e -> (id,e)

and Type_constraint =
    parse Literal ":"; Type ty -> ty
       #|(if MACROGRAM then [] else <:Gram:Rule_body<
          Literal "#:"; Expr e -> eval_macro_expr_MLtype e>>)

and entry Match =
    parse Match1 ml -> rev ml

and Match1 =
    parse Bpatat p; Match_rule mr -> [(p, mr)]
        | Match1 ml; Literal "|"; Bpatat p; Match_rule mr -> (p, mr)::ml
       #|(if MACROGRAM then [] else <:Gram:Rule_body<
          Match1 ml; Literal "#|"; Expr e
            -> rev_append (eval_macro_expr_MLpat_ML_list e) ml>>)

and Match_rule =
    parse Bpatat p; Match_rule mr -> MLmatch [p,mr]
        | Literal "->"; Expr e -> e
```

```
and Try_case =
    parse Pat p; Literal "->"; Literal "reraise" -> p,MLreraise
        | Pat p; Literal "->"; Expr e; Literal "reraise" -> p,MLseqreraise e
(*          | Pat p; Literal "->"; Expr17 e; Literal ";"; Literal "reraise" -> p,MLseqrera
        | Umatch_case umc -> umc

and Umatch_case = parse Pat p; Literal "->"; Expr e -> (p,e)

and entry Umatch =
    parse Umatch1 uml -> rev uml

and Umatch1 =
    parse Umatch_case umc -> [umc]
        | Umatch1 uml; Literal "|"; Umatch_case umc -> umc::uml
       #|(if MACROGRAM then [] else <:Gram:Rule_body<
          Umatch1 uml; Literal "#|"; Expr e
            -> rev_append (eval_macro_expr_MLpat_ML_list e) uml>>)

and Trymatch = parse Trymatch1 tml -> rev tml

and Trymatch1 =
    parse Try_case tc -> [tc]
        | Trymatch1 tcl; Literal "|"; Try_case tc -> tc::tcl
       #|(if MACROGRAM then [] else <:Gram:Rule_body<
          Trymatch1 tcl; Literal "#|"; Expr e
            -> rev_append (eval_macro_expr_MLpat_ML_list e) tcl>>)

and entry Decl =
    parse Let _; Val_binding vb -> vb
        | Let _; Literal "rec"; Val_binding vb -> MLrec vb
        | Literal "type"; Type_binding tb -> MLtype tb
        | Literal "exception"; Exc_binding ebl -> MLexception ebl
        | Decl local_decl; Literal "in"; Decl decl -> MLlocal(local_decl,decl)
        | Literal "("; Decl d; Literal ")" -> d

and entry Val_binding =
    parse Val_binding0 vb -> mkdecl vb
        | Val_binding vb1; Literal "and"; Val_binding0 vb2
            -> binarize(vb1, mkdecl vb2)
        | Val_binding vb1; Literal "#and"; Expr0 e
            -> #(if MACROGRAM then <:Caml:Expr<binarize(vb1, mkdecl e)>>
                 else <:Caml:Expr<
                         binarize
                          (vb1, eval_macro_expr_MLdecl e)>>
                )

and Let = parse Literal "let" -> () | Literal "value" -> ()

and Val_binding0 =
    parse Bpat p; Literal "="; Expr e -> [p,e]
```

```
        | Bpatat p; Beurk b
    (* s'appelait "b", mais je sais pce que c'est *)
            -> [p,b]
        | Bpatat p1; Inf i; Bpatat p2; Literal "="; Expr e
            -> [MLvarpat i, MLmatch[MLpairpat(p1, p2),e]]
        | Val_binding0 vb; Literal "|"; Bpatat p; Beurk b -> (p, b)::vb

and Beurk =
    parse Bpatat p; Beurk n -> MLmatch[p,n]
        | Literal "="; Expr e -> e

#and(if MACROGRAM then [] else <:Gram:Rules<
 entry Overload_binding =
    parse Overload_binding0 ov;
          ( * (parse Literal "and"; Overload_binding0 ov -> ov)) ovl;
          Literal ";;"
          -> MLdecl (MLoverload (ov::ovl))

and Overload_binding0 =
    parse MLIdent3 id; Literal "with"; MLIdent3l idl -> id, idl

and entry Forward_decl =
    parse Straint_list strl; Literal ";;" -> MLdecl(MLforward (prefix :: strl))

and MLIdent3_straint =
    parse MLIdent3 id; Type_constraint tc -> (id,tc)
        | Literal "("; MLIdent3 id; Type_constraint tc; Literal ")" -> (id,tc)
>>)

and entry Type_binding =
    parse Type_binding_1 tb;
          ( * (parse Literal "and"; Type_binding_1 tb -> tb)) tbl -> tb::tbl

and Type_binding_1 =
    parse Type_binding_c tb -> MLconcrete_type tb
        | Type_binding_r tb -> MLrecord_type tb
        | Type_binding_abb tb -> MLabbrev_type tb

and Type_binding_args =
    parse Var_tyargs vta; MLIdent2 id -> (id,vta)
        | Var_ty vt1; Infixes i; Var_ty vt2 -> (i,[vt1;vt2])

and Type_binding_c =
    parse Type_binding_args tbargs; Literal "="; Constructors constrs
            -> mk_type_binding tbargs constrs

and Type_binding_r =
    parse Type_binding_args tbargs; Literal "="; Labels lbl
            -> mk_type_binding tbargs lbl
```

```
and Type_binding_abb =
    parse Type_binding_args tbargs; Literal "=="; Type ty
            -> mk_type_binding tbargs ty


and Labels = parse Labs l -> l
                 | Literal "{"; Labs l; Literal "}" -> l

and Constructors = parse Constr cl -> rev cl
                         | Literal "["; Constr cl; Literal "]" -> rev cl

and Var_tyargs =
    parse -> []
        | Var_ty v -> [v]
        | Literal "("; Var_tyl vtl; Var_ty v; Literal ")" -> rev (v::vtl)

and Var_tyl =
    parse -> []
        | Var_tyl vtl; Var_ty v; Literal "," -> v::vtl

and entry Constr =
    parse Constr cl; Literal "|"; Constr1 c -> c::cl
        | Constr1 c -> [c]
       #|(if MACROGRAM then [] else <:Gram:Rule_body<
          Constr c; Literal "#|"; Expr e
            -> rev_append (eval_macro_expr_MLconstruct_list e) c>>)
and Constr1 =
    parse Constructor_or_ident id; Literal "of"; Type ty
            -> MLconstruct (id,ty)
        | Constructor_or_ident id -> MLconstruct0 id
        | Mutable _; Constructor_or_ident id; Literal "of";
          Type ty
            -> MLqconstruct (id,standard_qlabel_qualificator,ty)
        | Lazy _; Constructor_or_ident id; Literal "of";
          Type ty
            -> MLqconstruct (id,standard_llabel_qualificator,ty)

and Lazy =
    parse Literal "lazy" -> () | Literal "*" -> ()
and Mutable =
    parse Literal "mutable" -> () | Literal "!" -> ()

and Labs =
    parse Labl l -> (match rev l with prefix :: labs -> labs
                                    | _ -> system_error "No labels")

and entry Labl =
    parse Labl ll; Literal ";"; Labl l -> l::ll
        | Labl l -> [l]
       #|(if MACROGRAM then [] else <:Gram:Rule_body<
```

```
          Labl ll; Literal "#;"; Expr0 e
            -> rev_append (eval_macro_expr_MLlabel_list e) ll>>)

 and Lab1 =
     parse MLIdent id; Literal ":"; Type ty -> MLlabel (id,ty)
         | Mutable _; MLIdent id; Literal ":"; Type ty
             -> MLqlabel (id,standard_qlabel_qualificator,ty)
         | Lazy _; MLIdent id; Literal ":"; Type ty
             -> MLqlabel (id,standard_llabel_qualificator,ty)

 and entry Exc_binding =
     parse Exc_binding1 eb;
           ( * (parse Literal "and"; Exc_binding1 eb -> eb)) ebl -> eb::ebl

 and Exc_binding1 =
     parse MLIdent id; Literal "of"; Type ty -> (id,ty)
         | MLIdent id -> (id,MLconsttyp("unit",[]))

 and Pat1 =
     parse Constructor_or_ident id; Bpatat p -> MLconpat(id,p)
         | Literal "dynamic"; Bpatat p -> MLdynpat p
         | Bpatat p -> p

 and Pat4 =
     parse Pat4 p1; Literal ","; Pat4 p2 -> MLpairpat(p1, p2)
         | Pat4 p1; INFIX i; Pat4 p2 -> MLconpat(i, MLpairpat(p1, p2))
         | Pat4 p1; Literal ("::" as c); Pat4 p2
             -> MLconpat(c, MLpairpat(p1, p2))
         | Pat1 p -> p

 and Pat5 =
     parse Or_pat orp -> MLorpat orp
         | Pat4 p -> p

 and Or_pat =
     parse Pat4 p1; (+ (parse Literal "|"; Pat4 p -> p)) ops
             -> p1,(prefix :: ops)

 and entry Pat =
     parse Pat5 p; Literal "as"; MLIdent2 id -> MLsynpat(p,id)
         | Pat5 p -> p

 and entry Bpat1 =
     parse Ce c -> MLconstpat c
         | Literal "_" -> MLwildpat
         | Literal "-"; NUM n with precedence uminus -> MLconstpat(mlnum (- n))
         | Constructor s -> MLcon0pat s
         | MLIdent2 id ->  MLvarpat id
         | Literal "strict"; MLIdent2 id -> MLstrictpat (MLvarpat id)
         | MLescape e -> (#(if MACROGRAM then <:Caml:Expr<e>>
```

```
                          else <:Caml:Expr<eval_macro_expr_MLpat e>>))

        | Literal "["; Pats ps; Literal "]" -> ps
(*        | Literal "[|"; Pats ps; Literal "|]" -> mkpatvect ps *)
(*         | Literal "[<"; Pats ps; Literal ">]" -> mkpatseg ps *)
        | Literal "{"; Lab_pats ps; Literal "}" -> MLrecordpat ps
        | Literal "("; Pat p; Literal ")" -> p
        | Literal "(" ; Pat p; Type_constraint str; Literal ")"
            -> MLstraintpat(p,str)
       #|(if MACROGRAM then <:Gram:Rule_body<
         Ol_program ol_exp; {MML_to_MLpat ol_exp} oe -> oe>>
          else <:Gram:Rule_body<
         Ol_program ol_exp -> expr_to_pat ol_exp>>)

and Bpatat =
    parse Bpat1 p; Literal "at"; MLIdent id -> MLoccpat (p,id)
        | Bpat1 p -> p


and Bpat =
    parse Bpatat p1; Literal ","; Bpat p2 -> MLpairpat(p1,p2)
        | Bpatat p -> p

and Pats =
    parse Pat p -> MLconpat("::",MLpairpat(p,MLvarpat ""))
        | Pat p; Literal ";"; Pats pl -> MLconpat("::", MLpairpat(p,pl))

and Lab_pat =
    parse IDENT id; Literal "="; Pat p -> MLlabelpat (id,p)
        | IDENT id -> MLlabelpat (id,MLvarpat id)
        | Literal "_" -> MLlabelwildpat

and Lab_pats =
    parse Lab_pat lp; ( * (parse Literal ";"; Lab_pat lp -> lp)) lpl
            -> lp,lpl

and Ce =
    parse NUM n -> mlnum n
        | INT i -> mlint i
        | FLOAT f -> mlfloat f
        | STRING s -> mlstring s
        | BOOL b -> mlbool b
        | Literal "("; Literal ")" -> mlnull
        | Literal "{"; Literal "}" -> mlnull

and Type1 =
    parse Var_ty vty -> MLvartyp vty
        | MLIdent2 const -> MLconsttyp(const,[])
        | Literal "("; Type ty; Literal ")" -> ty
        | Type1 tyarg; MLIdent2 tyco -> MLconsttyp(tyco,[tyarg])
```

```
        | Literal "("; Typel args; Literal ")"; MLIdent2 ty
            -> MLconsttyp(ty,args)
        #|(if MACROGRAM then <:Gram:Rule_body<MLescape e -> e>>
          else [])

and Typel =
    parse Type ty; (+ (parse Literal ","; Type t -> t)) tyl
            -> ty::(prefix :: tyl)

and Type2 =
    parse Typel ty1; Type_infixes tyi; Type2 ty2
            -> MLconsttyp(tyi,[ty1;ty2])
        | Typel ty -> ty

and entry Type =
    parse Type tysrc; Literal ("->" as arrow); Type tygl
            -> MLconsttyp(arrow,[tysrc;tygl])
        | Type2 ty1; Literal ("*" as b_and); Type ty2
            -> MLconsttyp(b_and,[ty1;ty2])
        | Type2 ty -> ty

and Var_ty = parse Literal "'"; MLIdent var -> var

and MLIdent0 = parse IDENT var -> var
                  | Literal "["; Literal "]" -> ""

and MLIdent = parse MLIdent0 var -> var
                | MLInfix i -> i

and Prefix_Ident = parse Literal "prefix"; MLInfix i -> i

and MLIdent2 =
    parse MLIdent0 var -> var | Prefix_Ident var -> var

and Constructor =
    parse Literal "'"; IDENT id -> ("'"^id)
        | Literal "prefix"; CInfixes id -> id

and Constructor_or_ident =
    parse Constructor c -> c | MLIdent2 id -> id

and Inf =
    parse INFIX i -> i
        | Literal ("+" as p) -> p
        | Literal ("/" as d) -> d  | Literal ("<=" as l) -> l
        | Literal (">=" as g) -> g | Literal ("<>" as d) -> d
        | Literal ("<" as l) -> l  | Literal (">" as g) -> g
        | Literal ("::" as c) -> c | Literal ("@" as a) -> a
        | Literal ("&" as m) -> m  | Literal ("or" as d) -> d
        | Literal ("^" as c) -> c
```

```
        | Literal (":=" as a) -> a

and Type_infixes =
    parse Inf i -> i
        | Literal ("-" as m) -> m

and Infixes = parse
          Type_infixes i -> i
        | Literal ("*" as a) -> a
        | Literal ("==" as a) -> a
        | Literal ("=" as e) -> e

and CInfixes = parse Literal "'"; Infixes i -> ("'"^i)

and Prefixes = parse Literal ("not" as n) -> n | Literal ("!" as b) -> b

and MLInfix = parse Infixes i -> i | Prefixes p -> p

#and
(if MACROGRAM then [] else <:Gram:Rules<
  MLIdent3l =
    parse MLIdent3 id;
          ( * (parse Literal ","; MLIdent3 id -> id)) idl -> id::idl

and MLIdent3 =
    parse IDENT var -> var | Prefix_Ident var -> var

and entry Export_list =
    parse Export1 exp -> [exp]
        | Export_list expl; Literal ";"; Export1 exp -> exp::expl

and Export1 =
    parse Literal "value"; MLIdent1 idl
              -> MLvalue_spec(uncons(map_type_var idl))
        | Literal "type"; MLIdent1 idl
              -> MLtype_spec(uncons(map_unknown_type idl))
        | Literal "abstype"; MLIdent1 idl
              -> MLtype_spec(uncons(map_abstract_type idl))
        | Literal "exception"; MLIdent1 idl
              -> MLexception_spec(uncons(map_type_var idl))

and entry Import_list =
    parse Import2 imp -> imp,[]
        | Import2 imp; Literal "from"; File_list fl -> imp, fl
(* rev fl*)

and File_list =
  parse Expr0 name;
        ( * (parse Literal ","; Expr0 n -> eval_macro_expr_string n)) nl ->
        (eval_macro_expr_string name)::nl
```

```
(*********
 parse Expr0 name -> [eval_macro_expr_string name]
                   | File_list fl; Literal ","; Expr0 name ->
                      (eval_macro_expr_string name)::fl
**********)

and Import2 =
    parse -> []
        | Import1 imp -> [imp]
        | Import2 impl; Literal ";"; Import1 imp -> imp::impl

and Type_name = parse Var_tyargs vty; MLIdent2 id -> (id,vty)

and Import1 =
    parse Literal "value"; Straint_list strl -> MLvalue_spec strl
        | Literal "type"; Type_binding tyl ->  MLtype_spec (uncons tyl)
        | Literal "type"; Type_name tyn -> MLtype_spec
                  (match tyn with (s,sl) ->
                   (MLunknown_type (s,sl,all_the_tags),[]))
        | Literal "type"; Type_name tyn ;
          Literal "with"; Literal "tags"; Expr0 e -> MLtype_spec
                  (match tyn with (s,sl) ->
                    MLunknown_type (s,sl,eval_to_btagl e),[])

        | Literal "exception"; Exc_binding eb -> MLexception_spec (uncons eb)

and MLIdent1 =
    parse MLIdent3 id -> [id]
        | MLIdent1 idl; Literal "and"; MLIdent3 id -> id::idl

and entry Straint_list =
    parse MLIdent3_straint str1;
          ( * (parse Literal "and"; MLIdent3_straint str -> str)) strl -> str1,strl

>>)

and MLescape =
    parse Literal "#";
          #(if MACROGRAM then <:Gram:Token<{parse_caml_expr0()}>>
            else <:Gram:Token<Expr0>>) e -> e
       #|(if MACROGRAM then <:Gram:Rule_body<
          Literal "{^"; {parse_caml_expr ()} e; Literal "^}" -> e>>
          else [])

;;

#-MACROGRAM
let coerce_to_MLsyntax f arg =
    match sys_eval_syntax(f arg)
    with dynamic (s: MLsyntax) -> s
```

```
      | _ -> system_error "Ill built syntax"

and coerce_to_ML f arg =
    match sys_eval_syntax(f arg)
    with dynamic (s: ML) -> s
       | _ -> system_error "Ill built syntax"

and coerce_to_MLpat f arg =
    match sys_eval_syntax(f arg)
    with dynamic (s: MLpat) -> s
       | _ -> system_error "Ill built syntax"

and coerce_to_MLtype f arg =
    match sys_eval_syntax(f arg)
    with dynamic (s: MLtype) -> s
       | _ -> system_error "Ill built syntax"

and coerce_to_MLdecl f arg =
    match sys_eval_syntax(f arg)
    with dynamic (s: MLdecl) -> s
       | _ -> system_error "Ill built syntax"

;;

#-MACROGRAM
let parse_caml_expr = coerce_to_ML (Caml "Expr").Parse_raw
and parse_caml_expr0 = coerce_to_ML (Caml "Expr0").Parse_raw
and parse_caml_pat =  coerce_to_MLpat (Caml "Pat").Parse_raw
and parse_caml_decl =  coerce_to_MLdecl (Caml "Decl").Parse_raw
and parse_caml_val_binding =  coerce_to_MLdecl (Caml "Val_binding").Parse_raw
and parse_caml_pat0 = coerce_to_MLpat (Caml "Bpat1").Parse_raw
and parse_caml_typ = coerce_to_MLtype (Caml "Type").Parse_raw
and parse_caml = coerce_to_MLsyntax (Caml "Caml").Parse_raw
and parse_caml_pragma = (Caml "Pragma").Parse_raw
and parse_caml_strl = (Caml "Straint_list").Parse_raw
and parse_caml_import_list = (Caml "Import_list").Parse_raw
and parse_caml_export_list = (Caml "Export_list").Parse_raw
and parse_forward_decl = coerce_to_MLsyntax (Caml "Forward_decl").Parse_raw
and parse_overload_decl =
    coerce_to_MLsyntax (Caml "Overload_binding").Parse_raw
;;

#-MACROGRAM
let parse_caml_straint_list () =
    match sys_eval_syntax(parse_caml_strl ())
    with dynamic (l:(string * MLtype) * (string * MLtype) list) -> l
       | _ -> system_error "parse_caml_straint_list"
;;

#+MACROGRAM
```

```
let parse_CAML_Expr = (CAML "Expr").Parse_raw
and parse_CAML_Expr0 = (CAML "Expr0").Parse_raw
and parse_CAML_MLnum = (CAML "MLnum").Parse_raw
and parse_CAML_MLstring = (CAML "MLstring").Parse_raw
and parse_CAML_MLbool = (CAML "MLbool").Parse_raw
and parse_CAML_MLint = (CAML "MLint").Parse_raw
and parse_CAML_MLfloat = (CAML "MLfloat").Parse_raw
and parse_CAML_MLvar = (CAML "MLvar").Parse_raw
and parse_CAML_Match = (CAML "Match").Parse_raw
and parse_CAML_Umatch = (CAML "Umatch").Parse_raw
and parse_CAML_Decl = (CAML "Decl").Parse_raw
and parse_CAML_Val_binding = (CAML "Val_binding").Parse_raw
and parse_CAML_Type_binding = (CAML "Type_binding").Parse_raw
and parse_CAML_Constr = (CAML "Constr").Parse_raw
and parse_CAML_Exc_binding = (CAML "Exc_binding").Parse_raw
and parse_CAML_Pat = (CAML "Pat").Parse_raw
and parse_CAML_Pat0 = (CAML "Bpat1").Parse_raw
and parse_CAML_Type = (CAML "Type").Parse_raw
;;

();;

end module
#with(if MACROGRAM then <:Caml:Export_list<
      value parse_CAML_Expr
        and parse_CAML_Expr0
        and parse_CAML_MLnum
        and parse_CAML_MLstring
        and parse_CAML_MLbool
        and parse_CAML_MLint
        and parse_CAML_MLfloat
        and parse_CAML_MLvar
        and parse_CAML_Match
        and parse_CAML_Umatch
        and parse_CAML_Decl
        and parse_CAML_Val_binding
        and parse_CAML_Type_binding
        and parse_CAML_Constr
        and parse_CAML_Exc_binding
        and parse_CAML_Pat
        and parse_CAML_Pat0
        and parse_CAML_Type>>

     else <:Caml:Export_list<
    value parse_caml_expr
      and parse_caml_expr0
      and parse_caml_pat
      and parse_caml_decl
      and parse_caml_val_binding
      and parse_caml_pat0
```

```
        and parse_caml_typ
        and parse_caml
        and parse_caml_pragma
        and parse_caml_straint_list
        and parse_caml_import_list
        and parse_caml_export_list
        and parse_forward_decl
        and parse_overload_decl
        and Caml>>)
;;
```

33.2 Grammar of grammars

```
(*****************************************************************************)
(*                                                                           *)
(*                      Projet      Formel                                   *)
(*                                                                           *)
(*                             CAML                                          *)
(*                                                                           *)
(*****************************************************************************)
(*                                                                           *)
(*                             Inria                                         *)
(*                       Domaine de Voluceau                                 *)
(*                       78150  Rocquencourt                                 *)
(*                             France                                        *)
(*                                                                           *)
(*****************************************************************************)

(* gram          The syntax of grammar definitions                           *)
(* Michel Mauny  *)

(* A faire:

    - Implanter un autre style de "|" ("||" ?) qui joue le role des orpats
    - Mettre des macros et voir la possibilite de partage des re'gles
    - Interpreter les orpats comme des informations "lex" quand c'est possible
      (i.e. quand il s'agit de litteraux inutilises ailleurs)
*)

system module Gram;;

let alpha = <:Caml:Type<'a>>;;

let parse_warning s = warning ("line "^(string_of_num !line_counter)^" "^s)
;;

(* The following are not used in the grammar of programs *)
(* Default grammar header: used when neither delimitors nor precedences  *)
(* are specified.                                                        *)
```

```
#-MACROGRAM
let default_header =
     {String_delimitor="\""; Comment_delimitor="%"; Precedences=[]}
;;

#-MACROGRAM
let default_sc = ("\"","%");;

#-MACROGRAM
(* Checks wether the delimitor proposed has chance of being usable        *)
let check_delimitor kind s =
    match explode s
    with [s'] -> s'
       | _ -> raise parse (kind^" delimitor must be single character")
;;

(* Builds a header from a couple of delimitors and a list of precedences  *)
(* specifications.                                                        *)
#-MACROGRAM
let mk_gheader ((s,c),pl) =
    let (s',c') = (check_delimitor "string" s,
                   check_delimitor "comment" c) in
    if s'=c'
    then raise parse
         "you cannot use same characters as string and comment delimitors"
    else
    {String_delimitor=s'; Comment_delimitor=c'; Precedences=pl}
;;

(* To check legality of precedences declarations *)
#-MACROGRAM
let check_precedences ((tok1,toks) as tokl) =
 (check tok1; do_list check toks; tokl)
 where check =
     function Keyword _ -> ()
            | Non_terminal s ->
                raise parse
            ("non terminal "^s^" is not allowed in precedences declarations")
            | Predef_tok s -> ()
            | _ -> raise parse "Illegal token in precedences declarations"
;;

let check_box = function "hov" -> ()| "h" -> () | "v" -> () | "hv" -> ()
                       | s -> raise parse (s^": illegal box type")
;;

#-MACROGRAM
let eval_macro_expr_Token e =
 match eval_prag_syntax e
 with dynamic (t:Token) -> t
```

```
    | d -> ill_typed_macro e d <:gtype<Token>>
;;

(**********************************************)
(* The grammar of grammar definitions itself *)
(**********************************************)

grammar for #(if MACROGRAM then "programs" else "values")
            #(if MACROGRAM then "GRAM" else "Gram") =
precedences
    nonassoc BOOL "it"
    (* To make sure BOOL and "it" won't be used as identifiers *)
;

(* Main entry point *)
rule entry Token = parser ext_token tok -> tok
and  entry Rule_body = parser cases cl -> prefix :: cl
and  entry Rules = parser grule_list rl -> rev rl

#and(if MACROGRAM then [] else
     <:Gram:Rules<
entry Decl = parser top_decl d; Literal ";;" -> d

and top_decl =
(* The skeleton of a grammar definition *)
    parser Literal "for"; Literal "values"; Gname name; Literal "="; header h;
          grules rl
            -> MLgrammar(Grammar_values{Name=name; Header=h; Rules=rl})
        | Literal "for"; Literal "programs"; Gname name; Literal "="; header h;
          grules rl
            -> MLgrammar(Grammar_programs{Name=name; Header=h; Rules=rl})
        | Gname name; Literal "="; header h; grules rl
            -> MLgrammar(Grammar_programs{Name=name; Header=h; Rules=rl})
        | Literal "for"; Macro e; Gname name; Literal "="; header h;
          grules rl
            -> (let gram =
                match eval_prag_syntax e
                with dynamic ("values" : string) -> Grammar_values
                   | dynamic ("programs" : string) -> Grammar_programs
                   | dynamic (s:string) ->
                     (raise parse (s^
                      ": illegal grammar type (should be values or programs)"))
                   | d -> ill_typed_macro e d <:gtype<string>>
                in MLgrammar(gram{Name=name; Header=h; Rules=rl}))
        | Gname name; Literal "=="; Gname old
            -> MLsyn_grammar (name,old)

(* Declaration part of the grammar *)
and header =
    parser "rule"  -> default_header
```

```
        | -> (parse_warning "forgotten keyword \"rule\", continuing...";
             default_header)
        | defined_header h; "rule" -> h

and defined_header =
    parser delim_kwd _; string_or_comment sc; Literal ";";
          Literal "precedences"; prec_list pl; Literal ";"
            -> mk_gheader (sc, (rev pl))
        | Literal "precedences"; prec_list pl; Literal ";";
          delim_kwd _; string_or_comment sc; Literal ";"
            -> mk_gheader (sc, (rev pl))
        | Literal "precedences"; prec_list pl; Literal ";"
            -> mk_gheader (default_sc, (rev pl))
        | delim_kwd _; string_or_comment sc; Literal ";"
            -> mk_gheader (sc, [])

and delim_kwd =
    parser Literal "delimitors" -> ()
        | Literal "delimitor" -> ()

and string_or_comment =
    parser Literal "string"; is_kwd _; STRING s -> (s,"%")
        | Literal "comment"; is_kwd _; STRING c -> ("\"",c)
        | Literal "string"; is_kwd _; STRING s;
          Literal "comment"; is_kwd _; STRING c -> (s,c)
        | Literal "comment"; is_kwd _; STRING c;
          Literal "string"; is_kwd _; STRING s -> (s,c)

and prec_list =
    parser prec_list hl; Literal ";" ; prec_elem he -> he::hl
        | prec_elem he -> [he]

and prec_elem =
    parser Literal "right"; (+(parser prec_token tok -> tok)) tokl;
           {check_precedences tokl} _
            -> Right_toks tokl
        | Literal "left"; (+(parser prec_token tok -> tok)) tokl;
           {check_precedences tokl} _
            -> Left_toks tokl
        | Literal "nonassoc"; (+(parser prec_token tok -> tok)) tokl;
           {check_precedences tokl} _
            -> Nonassoc_toks tokl
        | Literal "precedence"; IDENT v -> Prec_def v

(* Rules *)
and grules =
    parser grule_list rl -> uncons (rev rl)

and is_kwd =
    parser Literal "is" -> ()
```

```
        |  -> (parse_warning "forgotten keyword \"is\", continuing...")

and Gname =
    parser IDENT name -> name
         | Macro e -> (match eval_prag_syntax e
                       with dynamic (s:string) -> s
                          | d -> ill_typed_macro e d <:gtype<string>>)

>>
    ) (* End of #and *)

and prec_token =
    parser predef_token s -> Predef_tok s
        | literal s -> Keyword s
        | IDENT s -> Non_terminal s

and grule_list =
    parser grule_list rl; Literal "and"; grule r -> r::rl
        | grule r -> [r]
       #| (if MACROGRAM then []
           else <:Gram:Rule_body<
           grule_list rl; Literal "#and"; Caml_Expr0 e
             -> (match eval_prag_syntax e
                 with dynamic (rl1:Grammar_rule list) -> rev_append rl1 rl
                    | d -> ill_typed_macro e d <:gtype<Grammar_rule list>>)>>
         )

and grule =
    parser kind k; typed_ident v; Literal "="; parsing_rule cs
             -> {Kind=k; Rule_name=Rule_name v; Cases=cs}

and kind =
    parser Literal "entry" -> Entry
        |  -> Non_exported

and typed_ident =
    parser IDENT name -> (name,alpha)
        | Literal "("; IDENT name; Literal ":";
          Caml_Type t; Literal ")" -> (name,t)

and parsing_rule =
    parser parse_kwd _; cases cs -> cs

and parse_kwd =
    parser Literal "parse" -> ()
        | Literal "parser" -> ()

and cases =
    parser case_list cl -> uncons(rev cl)
```

```
and case_list =
    parser case_list cl; Literal "|"; case c -> c::cl
        | case_list cl; Macro_case mc -> mc cl
        | case c -> [c]

and Macro_case =
    parser Literal "#|"; Caml_Expr0 e
            ->
            (#(if MACROGRAM then <:Caml:Expr<e>>
               else <:Caml:Expr<
                match eval_prag_syntax e
                with dynamic (cl1: Rule_case list) -> rev_append cl1
                   | d -> ill_typed_macro e d <:gtype<Rule_case list>>
                   >>))

and case =
    parser left_elem_list lml; with_prec wp; rule_action a;
           {#(if MACROGRAM
              then <:CAML:Expr<
                   (<:CAML:Expr<
                       {Left_member= #lml @ #wp; Bindings=[]; Action= #a}>>) >>
              else <:Caml:Expr<
                   let left_mem =
                   match wp with [] -> lml
                                | _::_ -> lml@wp
                   in {Left_member= left_mem; Bindings=[]; Action= a}>>)} c
            -> c
        | rule_action a
            -> {Left_member=[]; Bindings=[]; Action= a}

and left_elem_list =
    parser left_elem l; ( * (parser Literal ";"; left_elem l -> l)) lml
             -> l::lml

and left_elem =
    parser dollar_binding b -> Dollar_binding b
        | wild_binding tok -> Token tok
        | Literal "-" -> Space_annot
        | Literal "\\" -> Break_annot(0,0)
        | Literal "\\\\" -> Newline_annot
        | Literal "\\-" -> Break_annot(1,0)
        | Literal "\\"; Literal "("; NUM n; Literal ","; NUM m; Literal ")"
            -> Break_annot(n,m)
        | Literal "["; pp_offset po;
          {#(if MACROGRAM then <:Caml:Expr< <:Caml:Expr<()>> >>
             else <:Caml:Expr<check_box (fst po)>>)} _;
          left_elem_list ll; Literal "]"
            -> Box_annot(po,ll)
       | Literal "("; left_elem e; Literal ")" -> e
```

```
and dollar_binding =
(* Variable bindings (no pattern-matching) *)
    parser token tok;
          {#(if not MACROGRAM then <:Caml:Expr<parse_caml_pat0 ()>>
              else <:Caml:Expr<parse_CAML_Pat0()>>)} p -> (p,tok)
        | literal_as_binding b -> b

and wild_binding =
    parser literal s -> Keyword s
(*        | token tok; Literal "_" -> tok *)

and ext_token =
    parser literal s -> Keyword s
        | token tok -> tok

and literal_as_binding =
    parser Literal "Literal"; Literal "("; literal_as_binding b; Literal ")"
            -> b
        | STRING lit; Literal "as"; IDENT name -> (MLvarpat name,Keyword lit)
       #|(if MACROGRAM then <:Gram:Rule_body<
          Macro e; Literal "as"; IDENT name -> (MLvarpat name,Keyword e)
          >> else [])

and token =
    parser predef_token s -> Predef_tok s
        | IDENT s -> Non_terminal s
        | Literal "{"; Caml_Expr esc; Literal "}"
            -> Escape_tok {Esc_bindings=[]; Escape_expr=esc}
        | Literal "("; regular r; ")" -> r
        | Macro e -> (#(if MACROGRAM then <:Caml:Expr<e>>
                       else <:Caml:Expr<eval_macro_expr_Token e>>))

and literal =
    parser Literal "Literal"; STRING s -> s
        | STRING s -> s
       #|(if MACROGRAM then <:Gram:Rule_body<
          Literal "Literal"; Macro e -> e>>
          else [])


and regular =
    parser parsing_rule cs -> Regular cs
        | Literal "*"; Literal "("; regular r; Literal ")" -> Star r
        | Literal "+"; Literal "("; regular r; Literal ")"
            -> Concat(r,(Star r))

and rule_action =
    parser Literal "accept"; Caml_Expr e -> Exit_action e
        | Literal "->"; Caml_Expr e -> Regular_action e
```

```
and with_prec =
    parser -> []
        | Literal "with"; Literal "precedence"; prec_token tok
            -> [With_prec tok]

and predef_token =
    parser Literal ("NUM" as s) -> s
        | Literal ("BOOL" as s) -> s
        | Literal ("IDENT" as s) -> s
        | Literal ("STRING" as s) -> s
        | Literal ("INT" as s) -> s
        | Literal ("INFIX" as s) -> s
        | Literal ("FLOAT" as s) -> s

        | Literal "Num" -> "NUM"
        | Literal "Bool" -> "BOOL"
        | Literal "Ident" -> "IDENT"
        | Literal "String" -> "STRING"
        | Literal "Int" -> "INT"
        | Literal "Infix" -> "INFIX"
        | Literal "Float" -> "FLOAT"

and pp_offset =
    parser -> ("hov",0)
        | Literal "<"; name_of_box s; offset_box n; Literal ">" -> (s,n)

and offset_box =
    parser NUM n -> n
        | -> 0

and name_of_box =
    parser IDENT s -> s | INFIX s -> s

and Caml_Type =
    parser {#(if not MACROGRAM then <:Caml:Expr<parse_caml_typ()>>
              else <:Caml:Expr<parse_CAML_Type()>>)} t -> t

and Caml_Expr0 =
    parser {parse_caml_expr0()} e -> e

and Caml_Expr =
    parser {#(if not MACROGRAM then <:Caml:Expr<parse_caml_expr ()>>
              else <:Caml:Expr<parse_CAML_Expr()>>)} e -> e

and Macro =
    parser Literal "#"; Caml_Expr0 e -> e

;;

#-MACROGRAM
```

```
let parse_decl = (Gram "Decl").Parse_raw;;

#+MACROGRAM
let parse_GRAM_Token = (GRAM "Token").Parse_raw
and parse_GRAM_Rules = (GRAM "Rules").Parse_raw
and parse_GRAM_Rule_body = (GRAM "Rule_body").Parse_raw
;;

#-MACROGRAM
let parse_grammar_decl () =
 try
  (match sys_eval_syntax (parse_decl())
   with dynamic (s: MLsyntax) -> s
      | _ -> system_error "parse_grammar_decl")
 with _ -> error_message "(while parsing grammar declaration)" reraise
;;

end module
 #with(if MACROGRAM
       then <:Caml:Export_list<
       value parse_GRAM_Token
         and parse_GRAM_Rules
         and parse_GRAM_Rule_body
         >>
        else <:Caml:Export_list<
        value parse_grammar_decl>>)

;;
```

Index

GLOSSARY

List of the CAML predefined values

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

I S B N - 2 - 7261 - 0769 - 9

(RT 121)