Chapter 5

Lists

Lists represent an important data structure, mainly because of their success in the Lisp language. Lists in ML are *homogeneous*: a list cannot contain elements of different types. This may be annoying to new ML users, yet lists are not as fundamental as in Lisp, since ML provides a facility for introducing new types allowing the user to define more precisely the data structures needed by the program (cf. chapter 6).

5.1 Building lists

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

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

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

We may build lists of numbers:

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

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

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

Lists of functions:

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

and indeed, lists of anything desired.

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

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

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

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

5.2 Extracting elements from lists: pattern-matching

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

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

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

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

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

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

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

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

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

5.3 Functions over lists

We will see in this section the definition of some useful functions over lists. These functions are of general interest, but are not exhaustive. Some of them are predefined in the Caml Light system. See also [9] or [37] for other examples of functions over lists.

Computation of the length of a list:

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

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

Concatenating two lists:

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

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

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

Reversing a list:

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

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

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

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

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

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

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

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

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

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

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

i.e. to `15`.

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

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

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

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

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

Exercises

Exercise 5.1 *Define the* `combine` *function which, when given a pair of lists, returns a list of pairs such that:*

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

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

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