Lists

1 Pattern-matching

Pattern matching allows you to select conditions neatly, without the need for many if statements. The syntax is most clearly seen through an example:

```
factorial : int -> int

let rec factorial a =
  match a with
    1 -> 1
  | _ -> a * factorial (a - 1)
```

The pattern _ is used as a "wild card" as it can match anything. OCaml will warn you if your list of pattens is not exhaustive. Note that all of the expressions that the patterns are matched to must have the same type and thus, this is the type of the whole expression.

You can also pattern match multiple things at once with the syntax:
`match a, b with 0, 0 ->` *`expression1`* `| x, y ->` *`expression2`* `| ...`

2 Making Lists

A *list* is an ordered collection of zero or more elements of the same type. They are denoted by enclosing the elements in square brackets, separated by semicolons, for example: `[1; 2; 3; 4; 5]`. If the list isn't empty (that is, it doesn't contain no elements), it can be decomposed into a head (the first element) and a tail (a list containing the rest of the elements). If the list only has one element, the tail is the empty list. The *cons* operator "`::`" adds an element to the front of a list; it can be used to separate a list `l` into its head and tail `h::t`. This is especially useful when pattern matching as it allows us to separate a list into its contents, for example:

```
sum : int list -> int

let rec sum l =
  match l with
    [] -> 0
  | h::t -> h + sum t
```

We can separate a list into even more elements as `a::(b::t)` is the same as `a::b::t`. We can access any element of the list in this way (although practically we wouldn't want to decompose it much more in this way). Thus, cons can distinguish lists of specific lengths or with particular contents (for example, a list that can be written as `a::b::t` must contain at least two elements).

The similar *append* operator `@` can combine two lists together. This is time proportional to the length of the left list, while cons is constant time.

If a function works regardless of the type of its arguments, it is called *polymorphic*. For example, the following length function will find the length of the list input, regardless of the type of elements this list contains. This list has type α `list` as a result.

```
length : 'a list -> list

let rec length l =
```

```
  match l with
    [] -> 0
  | h::t -> 1 + length t
```

Conversely, the sum function defined above is not polymorphic as it only operates over lists of type int list. A list can have other lists as its elements, in which case its type will be α `list list`.

We shall now define some useful functions for manipulating lists:

```
take : int -> 'a list -> 'a list
drop : int -> 'a list -> 'a list
member : 'a -> 'a list -> bool

let rec take n l =
  match l with
    [] ->
      if n = 0
        then []
        else raise (Invalid_argument "take")
  | h::t ->
      if n < 0 then raise (Invalid_argument "take") else
        if n = 0 then [] else h :: take (n - 1) t

let rec drop n l =
  match l with
    [] ->
      then []
      else raise (Invalid_argument "drop")
  | h::t ->
      if n < 0 then raise (Invalid_argument "drop") else
        if n = 0 then l else drop (n - 1) t

let rec member n l =
  match l with
    [] -> false
  | h::t -> n = h || member n t
```

Note that if pattern matches are not exhaustive, the interpreter will throw a warning and there is the possiblility that you experience a run-time error. This has been handled above using exceptions (discussed in the Datatypes and Trees chapter).

We shall now discuss a very powerful form of recursion called *tail recursion*. Often, recursion results in large expressions being built up (that is, many function calls on the stack), which can be inefficient as it is related to the size of the argument. Tail recursion bypasses this by using an accumulator. By passing an extra argument, we can store extra information at each function call. This allows us to perform calculations that are independent of the size of the argument. Here is an example of such a recursive function which reverses a list (note that the obvious way using `@` would be inefficient - quadratic in time). When using `h::t` as an argument in a function, remember to enclose it in brackets.

```
rev : 'a list -> 'a list -> 'a list

let rec rev_inner a l =
  match l with
```

```
    [] -> a
  | h::t -> rev_inner (h :: a) t

let rev l = rev_inner [] l
```

As an aside, while tail recursion is certainly more efficient than normal recursion, it is not necessarily worth aiming for efficiency at the expense of simplicity. In more involved situations, tail recursion can lead to bulky function calls with many arguments. For the most part, write correct programs that avoid any especially obvious and gross inefficiencies, while being as straightforward as possible. If the program is too slow, then it is permissible to refactor it in a more subtly efficient way.