Elements of Procedural Programming

In OCaml, one would typically use the Functional Programming paradigm. This entails having pure functions map arguments to return values without side-effects, and hence without changing the value of a "name" (this is captured by the idea of not changing state). However, practically it is often valuable to be able to modify the value assigned to a given name. This is more in the style of *imperative programming*, where commands (that describe which variables to alter) are given and executed in order. OCaml allows for this approach also.

In order to be able to modify the value assigned to a name, one must assign this name to a *reference*, which is really just a pointer. This has type $\alpha \to \alpha$ **ref**. For example, `let x = ref 0` builds a reference of type **int ref**. You extract the value (or dereference) using `!x`, which has type α **ref** $\to \alpha$. You update the value of a reference using `x := 50`, which has type α **ref** $\to \alpha \to$ **unit**.

```
swap : 'a ref -> 'a ref -> unit

let swap a b =
  let t = !a in
    a := !b; b := t
```

OCaml allows you to omit the else statement in an **if ... then ... else** statement. Doing so assumes that the else statement returns (). As a result the **then** statement must also have type **unit**. OCaml also allows you to collect statements together using **begin ... end**, so that you can maintain code clarity without having to use too many parentheses:

```
if x = y then
  begin
    a := !a + 1
    b := !b - 1
  end
```

1 Iteration

While recursion is the preferred method of looping in OCaml, it is also possible to use both while and for loops, which both have type **unit** and have the following syntax, respectively: **while** *boolean expression* **do** *expression* **done** and **for** *name = start* **to** *end* **do** *expression* **done**. **While loops** perform an action subject to a boolean condition, while **for loops** perform an action subject to a varying parameter. For example, `for x = 1 to 5 do print_int x; print_newline () done`. The following function finds the smallest power of two greater than or equal to some number.

```
smallest_pow2 : int -> int

let smallest_pow2 x =
  let t = ref 1 in
    while !t < x do
      t := !t * 2
    done;
    !t
```

2 Arrays

An array is a data type which stores a fixed number of elements of the same type (contiguously in memory). They are called with the syntax: `let a = [|1; 2; 3; 4; 5|]` and have type α **array**. Arrays have constant time lookup by giving the index or *subscript* of a value: `a.(0)` returns the int 1. Note that arrays are zero indexed. The syntax to update a value in the array (insertion is also constant time) is `a.(`*subscript*`) <-` *expression*, which has type **unit**, but has the side effect of changing a value in `a`. Be careful not to access elements outside of the range of the array or an Exception will be raised.

Arrays and lists are similar but have some key differences: the length of an array is constant, whereas an array can be resized; arrays can be mutated (that is, a value within it can be changed), which is not the case for a list; it is possible to lookup a value from anywhere in the array, whereas it is only possible to lookup the head of a list.

The length of an array can be easily obtained with the built-in function `Array.length a`, which has type α **array** $\rightarrow$ **int**.

An array can be built using `Array.make`, which has type **int** $\rightarrow \alpha \rightarrow \alpha$ **array**. This takes an argument for the length of the array and an initial value to assign every element to. You can nest an array in an array, for example `Array.make 3 (Array.make 3 5)`.

Before we present an example using arrays, we introduce a few more built-in functions. `String.iter` of type (**char** $\rightarrow$ **unit**) $\rightarrow$ **string** $\rightarrow$ **unit** calls a function on every char in a string. `int_of_char` and `char_of_int` convert charaters to and from integers using their ASCII codes (for example, `int_of_char 'C'` returns the int 67).

Note that `[ref 5; ref 5]` and `let x = ref 5 in [x; x]` are not the same, as changing one of the values in the former has no impact on the other variable, whereas changing one of the variables in the latter will change both correspondingly. This is because x is a pointer so both values in the list are pointing to the same thing. It is incredibly important to be aware of this to prevent unexpected errors.

```
table : int -> int array array

let table n =
  let a = Array.make n [||] in
    for x = 0 to n - 1 do
      a.(x) <- Array.make n 0
    done;
    for y = 0 to n - 1 do
      for x = 0 to n - 1 do
        a.(x).(y) <- (x + 1) * (y + 1)
      done
    done;
    a
```

This returns the n times table. Note that if we created the initial array by writing `let a = Array.make n (Array.make n 0)`, the program wouldn't work, due to the fact that every inner array would be refering to the same thing, so changing a value in one array would change the corresponding value in every array.

Now, we write a larger program which finds the number of characters, words, lines and sentences in a text file, as well as tallying up the frequency of each character. The implementation is fairly crude (counting space charaters to count words, and punctuation '.', '!', '?' for sentences).

```
print_histogram : int array -> unit
channel_statistics : in_channel -> unit
```

```
file_statistics : string -> unit

let print_histogram arr =
  print_string "Character frequencies:";
  print_newline ();
  for x = 0 to 255 do
    if arr.(x) > 0 then
      begin
        print_string "For character '";
        print_char (char_of_int x);
        print_string "' (charater number ";
        print_int x;
        print_string ") the count is ";
        print_int arr.(x);
        print_string ".";
        print_newline ()
      end
  done

let channel_statistics in_channel =
  let lines = ref 0 in
  let characters = ref 0 in
  let words = ref 0 in
  let sentences = ref 0 in
  let histogram = Array.make 256 0 in
    try
      while true do
        let line = input_line in_channel in
          lines := !lines + 1;
          characters := !characters + String.length line + 1;
          String.iter
            (fun c ->
               match c with
                 '.' | '?' | '!' -> sentences := !sentences + 1
               | ' ' -> words := !words + 1
               | _ -> ())
            line;
          String.iter
            (fun c ->
              let i = int_of_char c in
                histogram.(i) <- histogram.(i) + 1)
            line
      done
    with
      End_of_file ->
        characters := !characters - 1;
        print_string "There were ";
        print_int !lines;
        print_string " lines, making up ";
        print_int !characters;
        print_string " characters with ";
```

```
        print_int !words;
        print_string " words in ";
        print_int !sentences;
        print_string " sentences.";
        print_newline();
        print_histogram histogram

let file_statistics name =
  let channel = open_in name in
    try
      channel_statistics channel;
      close_in channel
    with
      _ -> close_in channel
```