File I/O

In OCaml, there are a number of ways in which you can directly interact with files or the computer in real time. We shall discuss some of the language's built-in functionality.

1 Writing to the screen

OCaml has a value `()` which represents *nothing* and has type **unit**. There are a number of functions that allow a user to print data to their screen. For example, `print_int`, `print_string`, `print_newline` print integers, strings or newlines (moves to the beginning of the next line) respectively. They have respective types **int** $\rightarrow$ **unit**, **string** $\rightarrow$ **unit** and **unit** $\rightarrow$ **unit**. This printing is a side-effect and `()` (that is, nothing) is returned by these functions.

It is possible to produce multiple side-effects sequentially by using the following syntax: x `;` y. This evaluates the expression x and ignores its result (best practices dictate that this should be **unit** anyway), then evaluates y. For example:

```
print_dict_entry : int * string -> unit

let print_dict_entry (k, v) =
  print_int k;
  print_newline ();
  print_string v;
  print_newline ()
```

We can "extract" a side-effect from each element of a list in an analagous way to the map function:

```
iter : ('a -> 'b) -> 'a list -> unit

let rec iter f l =
  match l with
    [] -> ()
  | h::t -> f h; iter f t
```

2 Reading from the keyboard

Along with outputting to the computer, it is also possible to take input from the computer. The functions `read_line` and `read_int` take a string or int from the user respectively (where pressing the enter key indicates that the input is completed). These have types **unit** $\rightarrow$ **string** and **unit** $\rightarrow$ **int** respectively. If the user input a non-integer to `read_int`, (`Failure "int_of_string"`) is raised. `int_of_string` makes an integer from a string and `string_of_int` makes a string representation of an integer. Below is an example of these functions in use.

```
read_dict_number : int -> (int * string) list
read_dict : unit -> (int * string) list

exception BadNumber

let rec read_dict_number n =
  if n = 0 then [] else
```

```
    try
      let i = read_int () in
        let name = read_line () in
          (i, name) :: read_dict_number (n - 1)
    with
      Failure _ ->
        print_string "This is not a valid integer.";
        print_newline ();
        print_string "Please enter integer and name again.";
        print_newline ();
        read_dict_number n

let rec read_dict () =
  print_string "How many dictionary entries to input?";
  print_newline ();
  try
    let n = read_int () in
      if n < 0 then BadNumber else read_dict_number n
  with
    Failure _ ->
      print_string "Not a number. Try again";
      print_newline ();
      read_dict ()
  | BadNumber ->
      print_string "Number is negative. Try again";
      print_newline ();
      read_dict ()
```

3 Using files

There are also various built-in functions for working with files in OCaml. A place we can read from has type **in_channel** and a place we can write to has type **out_channel**. We can create an input or output channel for some file using the functions `open_in` and `open_out` respectively, where this opens a channel associated with the given filename. They thus have types **string** → **in_channel** and **string** → **out_channel** respectively. If a file can't be opened, a `Sys_error` is raised. It is important that these channels are closed once they have been finished with. This can be done with the functions `close_in` and `close_out`, each with types **in_channel** → **unit** and **out_channel** → **unit** respectively.

Once we have opened these channels, it is possible to read and write to them, respectively. Reading is done using `input_line`, which has type **in_channel** → **string**. Writing is done using `output_string` and `output_char`, which have types **out_channel** → **string** → **unit** and **out_channel** → **char** → **unit** respectively. Note that if we want to read or write integers, we must convert them to and from strings using `int_of_string` and `string_of_int`.

We now write a program to copy a file `a.txt` to a file `b.txt`.

```
copy_file_ch : in_channel -> out_channel -> unit
copy_file : string -> string -> unit

exception CopyFailed
```

```
let rec copy_file_ch from_ch to_ch =
  try
    output_string to_ch (input_string from_ch);
    output_string "\n";
    copy_file_ch from_ch to_ch
  with
    End_of_file -> ()

let copy_file file1 file2 =
  try
    let from_ch = open_in file1 in
      let to_ch = open_out file2 in
        copy_file_ch from_ch to_ch;
        close_in from_ch;
        close_out to_ch
  with
    _ -> raise CopyFailed
```

We call this function as `copy_file "a.txt" "b.txt"`. Note the use of the character '\n' to represent a newline. See also the use of the exception `End_of_file`, which is raised when there is nothing more to read from a file.