Chapter 9

Basic input/output

We describe in this chapter the Caml Light input/output model and some of its primitive operations. More complete information about IO can be found in the Caml Light manual [20].

Caml Light has an imperative input/output model: an IO operation should be considered as a side-effect, and is thus dependent on the order of evaluation. IOs are performed onto *channels* with types `in_channel` and `out_channel`. These types are *abstract*, i.e. their representation is not accessible.

Three channels are predefined:

```
#std_in;;
- : in_channel = <abstr>

#std_out;;
- : out_channel = <abstr>

#std_err;;
- : out_channel = <abstr>
```

They are the “standard” IO channels: `std_in` is usually connected to the keyboard, and printing onto `std_out` and `std_err` usually appears on the screen.

9.1 Printable types

It is not possible to print and read every value. Functions, for example, are typically not readable, unless a suitable string representation is designed and reading such a representation is followed by an interpretation computing the desired function.

We call *printable type* a type for which there are input/output primitives implemented in Caml Light. The main printable types are:

- characters: type `char`;
- strings: type `string`;
- integers: type `int`;
- floating point numbers: type `float`.

We know all these types from the previous chapters. Strings and characters support a notation for escaping to ASCII codes or to denote special characters such as newline:

```
#‘A‘;;
- : char = ‘A‘
```

```
#‘\065‘;;
- : char = ‘A‘
```

```
#‘\\‘;;
- : char = ‘\\‘
```

```
#‘\n‘;;
- : char = ‘\n‘
```

```
#"string with\na newline inside";;
- : string = "string with\na newline inside"
```

The “\” character is used as an escape and is useful for non-printable or special characters.

Of course, character constants can be used as constant patterns:

```
#function ‘a‘ -> 0 | _ -> 1;;
- : char -> int = <fun>
```

On types such as `char` that have a finite number of constant elements, it may be useful to use *or-patterns*, gathering constants in the same matching rule:

```
#let is_vowel = function
#  ‘a‘ | ‘e‘ | ‘i‘ | ‘o‘ | ‘u‘ | ‘y‘ -> true
#| _ -> false;;
is_vowel : char -> bool = <fun>
```

The first rule is chosen if the argument matches one of the cases. Since there is a total ordering on characters, the syntax of character patterns is enriched with a “`..`” notation:

```
#let is_lower_case_letter = function
#  ‘a‘..‘z‘ -> true
#| _ -> false;;
is_lower_case_letter : char -> bool = <fun>
```

Of course, or-patterns and this notation can be mixed, as in:

```
#let is_letter = function
#  ‘a‘..‘z‘ | ‘A‘..‘Z‘ -> true
#| _ -> false;;
is_letter : char -> bool = <fun>
```

In the next sections, we give the most commonly used IO primitives on these printable types. A complete listing of predefined IO operations is given in [20].

9.2 Output

Printing on standard output is performed by the following functions:

```
#print_char;;
- : char -> unit = <fun>
```

```
#print_string;;
- : string -> unit = <fun>
```

```
#print_int;;
- : int -> unit = <fun>
```

```
#print_float;;
- : float -> unit = <fun>
```

Printing is *buffered*, i.e. the effect of a call to a printing function may not be seen immediately: *flushing* explicitly the output buffer is sometimes required, unless a printing function flushes it implicitly. Flushing is done with the `flush` function:

```
#flush;;
- : out_channel -> unit = <fun>
```

```
#print_string "Hello!"; flush std_out;;
Hello!- : unit = ()
```

The `print_newline` function prints a newline character and flushes the standard output:

```
#print_newline;;
- : unit -> unit = <fun>
```

Flushing is required when writing standalone applications, in which the application may terminate without all printing being done. Standalone applications should terminate by a call to the `exit` function (from the `io` module), which flushes all pending output on `std_out` and `std_err`.

Printing on the standard error channel `std_err` is done with the following functions:

```
#prerr_char;;
- : char -> unit = <fun>
```

```
#prerr_string;;
- : string -> unit = <fun>
```

```
#prerr_int;;
- : int -> unit = <fun>
```

```
#prerr_float;;
- : float -> unit = <fun>
```

The following function prints its string argument followed by a newline character to `std_err` and then flushes `std_err`.

```
#prerr_endline;;
- : string -> unit = <fun>
```

9.3 Input

These input primitives flush the standard output and read from the standard input:

```
#read_line;;
- : unit -> string = <fun>
```

```
#read_int;;
- : unit -> int = <fun>
```

```
#read_float;;
- : unit -> float = <fun>
```

Because of their names and types, these functions do not need further explanation.

9.4 Channels on files

When programs have to read from or print to files, it is necessary to open and close channels on these files.

9.4.1 Opening and closing channels

Opening and closing is performed with the following functions:

```
#open_in;;
- : string -> in_channel = <fun>
```

```
#open_out;;
- : string -> out_channel = <fun>
```

```
#close_in;;
- : in_channel -> unit = <fun>
```

```
#close_out;;
- : out_channel -> unit = <fun>
```

The `open_in` function checks the existence of its filename argument, and returns a new input channel on that file; `open_out` creates a new file (or truncates it to zero length if it exists) and returns an output channel on that file. Both functions fail if permissions are not sufficient for reading or writing.
Warning:

- Closing functions close their channel argument. Since their behavior is unspecified on already closed channels, anything can happen in this case!

- Closing one of the standard IO channels (`std_in`, `std_out`, `std_err`) have unpredictable effects!

9.4.2 Reading or writing from/to specified channels

Some of the functions on standard input/output have corresponding functions working on channels:

```
#output_char;;
- : out_channel -> char -> unit = <fun>
```

```
#output_string;;
- : out_channel -> string -> unit = <fun>
```

```
#input_char;;
- : in_channel -> char = <fun>
```

```
#input_line;;
- : in_channel -> string = <fun>
```

9.4.3 Failures

The exception `End_of_file` is raised when an input operation cannot complete because the end of the file has been reached.

```
#End_of_file;;
- : exn = End_of_file
```

The exception `sys__Sys_error` (`Sys_error` from the module `sys`) is raised when some manipulation of files is forbidden by the operating system:

```
#open_in "abracadabra";;
Uncaught exception: sys__Sys_error "abracadabra: No such file or directory"
```

The functions that we have seen in this chapter are sufficient for our needs. Many more exist (useful mainly when working with files) and are described in [20].

Exercises

Exercise 9.1 *Define a function* `copy_file` *taking two filenames (of type* `string`*) as arguments, and copying the contents of the first file on the second one. Error messages must be printed on* `std_err`*.*

Exercise 9.2 *Define a function* `wc` *taking a filename as argument and printing on the standard output the number of characters and lines appearing in the file. Error messages must be printed on* `std_err`*.*

Note: it is good practice to develop a program in defining small functions. A single function doing the whole work is usually harder to debug and to read. With small functions, one can trace them and see the arguments they are called on and the result they produce.