Chapter 11

Standalone programs and separate compilation

So far, we have used Caml Light in an interactive way. It is also possible to program in Caml Light in a batch-oriented way: writing source code in a file, having it compiled into an executable program, and executing the program outside of the Caml Light environment. Interactive use is great for learning the language and quickly testing new functions. Batch use is more convenient to develop larger programs, that should be usable without knowledge of Caml Light.

Note for Macintosh users: batch compilation is not available in the standalone Caml Light application. It requires the MPW environment (see the Caml Light manual).

11.1 Standalone programs

Standalone programs are composed of a sequence of phrases, contained in one or several text files. Phrases are the same as at toplevel: expressions, value declarations, type declarations, exception declarations, and directives. When executing the stand-alone program produced by the compiler, all phrases are executed in order. The values of expressions and declared global variables are not printed, however. A stand-alone program has to perform input and output explicitly.

Here is a sample program, that prints the number of characters and the number of lines of its standard input, like the `wc` Unix utility.

```
let chars = ref 0;;
let lines = ref 0;;
try
  while true do
    let c = input_char std_in in
      chars := !chars + 1;
      if c = `\n` then lines := !lines + 1 else ()
  done
with End_of_file ->
  print_int !chars; print_string " characters, ";
  print_int !lines; print_string " lines.\n";
  exit 0
```

```
;;
```

The `input_char` function reads the next character from an input channel (here, `std_in`, the channel connected to standard input). It raises exception `End_of_file` when reaching the end of the file. The `exit` function aborts the process. Its argument is the exit status of the process. Calling `exit` is absolutely necessary to ensure proper flushing of the output channels.

Assume this program is in file `count.ml`. To compile it, simply run the `camlc` command from the command interpreter:

```
camlc -o count count.ml
```

The compiler produces an executable file `count`. You can now run `count` with the help of the "camlrun" command:

```
camlrun count < count.ml
```

This should display something like:

```
306 characters, 13 lines.
```

Under Unix, the `count` file can actually be executed directly, just like any other Unix command, as in:

```
./count < count.ml
```

This also works under MS-DOS, provided the executable file is given extension `.exe`. That is, if we compile `count.ml` as follows:

```
camlc -o count.exe count.ml
```

we can run `count.exe` directly, as in:

```
count.exe < count.ml
```

See the reference manual for more information on `camlc`.

11.2 Programs in several files

It is possible to split one program into several source files, separately compiled. This way, local changes do not imply a full recompilation of the program. Let us illustrate that on the previous example. We split it in two modules: one that implements integer counters; another that performs the actual counting. Here is the first one, `counter.ml`:

```
(* counter.ml *)
type counter = { mutable val: int };;
let new init = { val = init };;
let incr c = c.val <- c.val + 1;;
let read c = c.val;;
```

Here is the source for the main program, in file `main.ml`.

```
(* main.ml *)
let chars = counter__new 0;;
let lines = counter__new 0;;
try
  while true do
    let c = input_char std_in in
      counter__incr chars;
      if c = `\n` then counter__incr lines else ()
  done
with End_of_file ->
  print_int (counter__read chars); print_string " characters, ";
  print_int (counter__read lines); print_string " lines.\n";
  exit 0
;;
```

Notice that references to identifiers defined in module `counter.ml` are prefixed with the name of the module, `counter`, and by `__` (the "long dash" symbol: two underscore characters). If we had simply entered `new 0`, for instance, the compiler would have assumed `new` is an identifier declared in the current module, and issued an "undefined identifier" error.

Compiling this program requires two compilation steps, plus one final linking step.

```
camlc -c counter.ml
camlc -c main.ml
camlc -o main counter.zo main.zo
```

Running the program is done as before:

```
camlrun main < counter.ml
```

The `-c` option to `camlc` means "compile only"; that is, the compiler should not attempt to produce a stand-alone executable program from the given file, but simply an object code file (files `counter.zo`, `main.zo`). The final linking phases takes the two `.zo` files and produces the executable `main`. Object files must be linked in the right order: for each global identifier, the module defining it must come before the modules that use it.

Prefixing all external identifiers by the name of their defining module is sometimes tedious. Therefore, the Caml Light compiler provides a mechanism to omit the `module__` part in external identifiers. The system maintains a list of "default" modules, called the currently opened modules, and whenever it encounters an identifier without the `module__` part, it searches through the opened modules, to find one that defines this identifier. Searched modules always include the module being compiled (searched first), and some library modules of general use. In addition, two directives are provided to add and to remove modules from the list of opened modules:

- `#open "module";;` to add `module` in front of the list;
- `#close "module";;` to remove `module` from the list.

For instance, we can rewrite the `main.ml` file above as:

```
#open "counter";;
let chars = new 0;;
let lines = new 0;;
try
  while true do
    let c = input_char std_in in
      incr chars;
      if c = `\n` then incr lines
  done
with End_of_file ->
  print_int (read chars);
  print_string " characters, ";
  print_int (read lines);
  print_string " lines.\n";
  exit 0
;;
```

After the `#open "counter"` directive, the identifier `new` automatically resolves to `counters__new`.

If two modules, say `mod1` and `mod2`, define both a global value `f`, then `f` in a client module `client` resolves to `mod1__f` if `mod1` is opened but not `mod2`, or if `mod1` has been opened more recently than `mod2`. Otherwise, it resolves to `mod2__f`. For instance, the two system modules `int` and `float` both define the infix identifier `+`. Both modules `int` and `float` are opened by default, but `int` comes first. Hence, `x + y` is understood as the integer addition, since `+` is resolved to `int__+`. But after the directive `#open "float";;`, module `float` comes first, and the identifier `+` is resolved to `float__+`.

11.3 Abstraction

Some globals defined in a module are not intended to be used outside of this module. Then, it is good programming style not to export them outside of the module, so that the compiler can check they are not used in another module. Also, one may wish to export a data type abstractly, that is, without publicizing the structure of the type. This ensures that other modules cannot build or inspect objects of that type without going through one of the functions on that type exported in the defining module. This helps in writing clean, well-structured programs.

The way to do that in Caml Light is to write an explicit interface, or output signature, specifying those identifiers that are visible from the outside. All other identifiers will remain local to the module. For global values, their types must be given by hand. The interface is contained in a file whose name is the module name, with extension `.mli`.

Here is for instance an interface for the `counter` module, that abstracts the type `counter`:

```
(* counter.mli *)
type counter;;        (* an abstract type *)
value new : int -> counter
  and incr : counter -> unit
  and read : counter -> int;;
```

Interfaces must be compiled separately. However, once the interface for module A has been compiled, any module B that uses A can be immediately compiled, even if the implementation of A is not yet compiled or even not yet written. Consider:

```
camlc -c counter.mli
camlc -c main.ml
camlc -c counter.ml
camlc -o main counter.zo main.zo
```

The implementation `main.ml` could be compiled before `counter.ml`. The only requirement for compiling `main.ml` is the existence of `counter.zi`, the compiled interface of the `counter` module.

Exercises

Exercise 11.1 *Complete the* `count` *command: it should be able to operate on several files, given on the command line. Hint:* `sys__command_line` *is an array of strings, containing the command-line arguments to the process.*

Part III

A complete example