CS251 Programming Languages **Handout # 10**
Prof. Lyn Turbak **January 31, 2007**
Wellesley College

Using OCAML

ML is a function-oriented programming language developed by Robin Milner and others at Edinburgh University in the late 1970s. The name ML, which stands for **m**eta-**l**anguage, was chosen because the primary initial use of the language was as a meta-language for specifying theorem provers. In this course, we useML as a meta-language for specifying interpreters and translators. The salient features of ML include first-class functions, static typing via automatically inferred types, parametric polymorphism, algebraic (i.e. sum-of-products) data types, pattern matching, garbage collection, and a powerful module system. As we shall see, all of these features facilitate writing interpreters and translators. The version of ML used in CS251 is OBJECTIVE CAML, also known as OCAML. This handout describes how to run OCAML on the Linux workstations here at Wellesley.

You will use Emacs to create OCAML program files, and you will load these files into an interactive OCAML top-level interpreter. Type reconstruction is performed on all OCAML programs and you cannot test your programs until they pass the type checker. Understanding error messages from the type checker and using them to pinpoint type errors in your program are important skills that you will need to hone.

There are three ways to run OCAML on the Linux workstations:

1. in a regular shell window;
2. in a shell window in Emacs; and
3. in a special OCAML buffer within Emacs.

These three approaches are described below in Secs. 1–3. I strongly recommended the last approach — a special OCAML buffer within Emacs — since it simplifies many interactions. Nevertheless, you should still read all sections since many of the notes in the early sections are still relevant for the later ones.

This handout only scratches the surface of OCAML. For more detailed documentation, browse the following web pages:

`http://caml.inria.fr/`
`http://www.ocaml.org/`

1 Running OCAML in a Shell

1.1 Launching the OCAML interpreter

The simplest way to run OCAML on the Linux workstations is in a Linux shell window by executing the command `ocaml`. This will print a herald on the screen and eventually the '`#`' prompt of OCAML. For example:

```
[gdome@jaguar gdome] ocaml
        Objective Caml version 3.07+2
#
```

You can now evaluate expressions by typing them in followed by two semi-colons and ENTER. The two semi-colons tell the interpreter that you are done with the expression. This allows you to have expressions with multiple lines. For example, here is the transcript of a session in which two single-line expressions and one multiple-line expression are evaluated:

```
# 1 + 2;;
- : int = 3
# let xs = List.map (fun x -> x * 2) [4; 3; 7];;
val xs : int list = [8; 6; 14]
# let a = 1 + 2 in
    let b = 4 * a in
      (a,b);;
- : int * int = (3, 12)
```

If you forget the semi-colons at the end of an expression, OCAML will think you are continuing the expression onto the next line. In this case, you can just type the two semi-colons followed by ENTER to indicate that you are done. For example:

```
# 1 + 2
  ;;
- : int = 3
```

The OCAML interpreter expects that the unit of evaluation will be a top-level declaration, typically one of the form

```
let name = exp;;
```

It is common to declare functions using the form

```
let function-name formal1 ... formaln = exp;;
```

This is syntactic sugar for

```
let function-name = fun formal1 -> ... fun formaln -> exp;;
```

For example, the declaration

```
let avg a b = (a+b)/2;;
```

is equivalent to

```
let avg = fun a -> fun b -> (a+b)/2;;
```

Recursive functions are declared via the form

```
let rec function-name formal1 ... formaln = exp;;
```

For example,

```
# let rec sumBetween m n =
    if m > n then
      0
    else
      m + (sumBetween (m+1) n);;
val sumBetween : int -> int -> int = <fun>
# sumBetween 1 10;;
- : int = 55
```

If you just enter an expression E, the OCAML interpreter treats it as a declaration of the form `let - =` E, where `-` in this context is a special variable that is not actually bound to the result.

1.2 Loading Files

It is tedious to type all declarations directly at the OCAML interpreter. It is especially frustrating to type in a long declaration only to notice that you made an error near the beginning and you have to type it in all over again. In order to reduce your frustration level, it is wise to use a text editor (e.g., Emacs) to type in all but the simplest OCAML declarations. This way, it is easy to correct bugs and to save your declarations between different sessions with the OCAML interpreter. (Note: the file extension that you should use for OCAML files is `.ml`. Using this extension will enable various OCAML features in Emacs. See Sec. 4 for details.)

If *filename* is the name of a file containing OCAML declarations and expressions, evaluating

```
#use "filename"
```

will evaluate all of the expressions in the file, one by one, as if you had typed them in by hand. You must type the `#` character. `#use` is an example of a **directive** – a function-like entity that can be invoked in the top-level interpreter but is *not* an OCAML function. OCAML has several directives, all of which begin with the symbol `#`. This symbol is different from the `#` that serves as a prompt! For example:

```
# #use "ps2.ml";;
val sum_multiples_of_3_or_5 : 'a * 'b -> int = <fun>
val contains_multiple : 'a * 'b -> bool = <fun>
val all_contain_multiple : 'a * 'b -> bool = <fun>
val merge : 'a * 'b -> 'c list = <fun>
val alts : 'a -> 'b list = <fun>
val cartesian_product : 'a * 'b -> 'c list = <fun>
val bits : 'a -> 'b list = <fun>
val inserts : 'a * 'b -> 'c list = <fun>
val permutations : 'a -> 'b list = <fun>
```

Note how OCAML gives the type of each declaration in the file.

The `#use` directive can be used within a file to load other files. For example, here is the contents of a file `load-ps2.ml` that loads two other files:

```
#use "ps2.ml"
#use "ps2-test.ml"
```

The filename given to `#use` may be either an absolute pathname (such as the absolute pathname `/students/gdome/cs251/ps2/ps2.ml`) or a pathname relative to the current working directory. By default, the current working directory for the OCAML interpreter is the current working directory of the shell from which it was invoked, and by default this is your `cs` home directory (e.g. `/students/gdome`). So rather than evaluating

```
#use "/students/gdome/cs251/ps2/ps2.ml"
```

you could instead evaluate

```
#use "cs251/ps2/ps2.ml"
```

It is typical to load many files (or the same file many times) from the same directory. Moreover, files that themselves contain `#use` (like `load-ps1.ml` considered above) often have built into them an assumption that you load them from a particular directory. For these reasons, you need to be able to change the current working directory within the OCAML interpreter. To do this, evaluate

```
#cd dirname
```

where *dirname* is the name of the directory you want to be the new current working directory. This name can either be an absolute pathname, or a pathname relative to the current working directory. For example,

```
#cd "/students/gdome/cs251/ps2"
```

sets the OCAML directory to `/students/gdome/cs251/ps2`. If this is followed by

```
#cd "../ps1"
```

the OCAML directory is now to `/students/gdome/cs251/ps1`.

OCAML does *not* understand the usual Linux abbreviation of ~ for the user's home directory (such as `/students/gdome`), so you have to type the long form for your home directory.

1.3 Exiting OCAML

To terminate your session with the OCAML interpreter, either use the `#quit` directive, or type `C-d` (i.e., control-D)[1].

2 Running OCAML in an Emacs Shell

You could do all of your OCAML programming in CS251 using just the techniques outlined in Sec. 1 above. However, you will find yourself constantly swapping attention between the Emacs editor (where you write your code) and the Linux shell running the OCAML interpreter (where you evaluate your code). Moreover, when logged in remotely, you often do not have the luxury of multiple windows.

You can do all OCAML editing and execution within a single Emacs window. The most straightforward way to do this is to run the OCAML interpreter in a shell inside of Emacs. You can start a Linux shell inside Emacs via `M-x shell`[2] this creates a special shell buffer named `*shell*`. You can then run OCAML inside this shell as described above. By using Emacs window-splitting techniques[3], you can see both the OCAML interpreter and the file you're editing on a single screen.

Another advantage to running OCAML under an Emacs shell is that the Emacs commands `M-p` and `M-n` cycle back and forth through the shell input history. So when testing a program in the OCAML interpreter, you needn't retype a test expression typed earlier; instead, type `M-p` a few times.

Yet another advantage is that the Emacs shell is a buffer that can easily be saved as a file. So it is easy to save a transcript of your interactions with OCAML.

3 Running OCAML in an Special Emacs Buffer

An even better way to run OCAML inside of Emacs is to create a special OCAML evaluation buffer. This can be done via the Emacs command `M-x run-caml`.[4] After executing this command, you will be prompted in the Emacs mini-buffer (at the bottom of the screen) for the version of OCAML to run. The default is `ocaml`; just type ENTER. This creates an OCAML interaction buffer whose name is `*inferior-caml*`.

There are two main advantages to using `*inferior-caml*` over the usual Emacs `*shell*` buffer for your OCAML interactions:

1. `*inferior-caml*` understands OCAML formatting conventions (e.g. how to indent subexpressions when the TAB key is pressed), and uses different colors to highlight keywords, strings, comments, etc.

2. When using `*inferior-caml*` you are not tying up your `*shell*` buffer, which you might want for another purpose.

[1]It is common in Unix systems for `C-d` to represent the "end of input".

[2]Notational convention: `M-x`, pronounced "Meta x", is entered by typing the "`Meta`" key and "`x`" key at the same time. On PC keyboards, the key labeled "`Alt`" is usually treated as the "`Meta`" key. On keyboards without a true "`Meta`" key, you can instead press the `ESC`ape key followed by pressing the "`x`" key.

[3]E.g., `C-x 2` splits a window in two, and `C-x o` moves the cursor between windows. See Emacs documentation for more details.

[4]This command is available only if you include the expression `(load "/usr/network/dot-emacs")` in your `~/.emacs` file.

For these reasons, I recommend that you always use `M-x run-caml` in Emacs to execute OCAML programs.

4 OCAML Emacs Mode

You can get the same OCAML formatting conventions used in the `*inferior-caml*` buffer in any Emacs editing buffer by putting it in "CAML mode". To do this manually, go to the buffer in Emacs and execute the Emacs command `M-x caml-mode`. If your OCAML file ends in a `.ml` extension, Emacs will automatically put your buffer into CAML mode when it loads the file into the buffer.[5]

OCAML mode helps you write OCAML code because it understands the formatting conventions for OCAML code. Like many Emacs modes, it helps you match parentheses by flashing the matching open parenthesis whenever you type a close parenthesis. Additionally, OCAML mode helps you put your code in pretty-printed format. Typing the `TAB` key will indent the code on that line according to the OCAML indentation conventions. You should get into the habit of hitting `TAB` after every return so that you start typing the next line at the appropriate indentation level. You can format an entire expression by typing `ESC C-q` when the cursor is at the first character of the expression. Keeping your OCAML expressions indented properly is important for reading and debugging code. Indenting the code will often highlight that the structure of the expression has a bug.

OCAML mode also understands what OCAML keywords are, and will color keywords, strings, and comments different from the rest of the text to highlight them.

[5]The automatic use of CAML mode in buffers editing `.ml` files is available only if you include the expression `(load "/usr/network/dot-emacs")` in your `~/.emacs` file.