Notes for MP0 - Basic OCaml
CS 421, Spring, 2010

We will make much use of the OCaml programming language in this class. It has some advantages over the languages that you already know — which I assume are Java and C++ — when it comes to writing compilers, which is the first thing we'll do. It also has some interesting advanced features, which we will cover in the second half of the course, that are increasingly finding their way into the most popular languages (e.g. Java, C#, Python) and into recent experimental languages that you may have heard of (F#, Clojure, Scala, and others).

We will take two classes at the beginning of the semester (lectures 2 and 3) to cover the parts of OCaml we'll be using in the first half of the course. These notes cover the simpest aspects of OCaml — how to declare variables and define functions, run the interactive system, write expressions, etc.

This handout should be enough for you to do MP0. There are numerous online resources for OCaml; the class web page points to the main ones.

Please read these notes, and do MP0, before the second class.

Using OCaml

OCaml can be used as an interactive system, and that is what you will usually do. MP0 explains where to find it and how to start it. The OCaml prompt is just a pound sign (#). At the prompt, you can enter any expression and OCaml will evaluate it and print its value.

Even in interactive mode, you will usually want to type your program into a file (with the extension "ml") using a text editor. To load it, type

```
# #use "filename";;
```

Note that the command is `#use`, not just `use`.

Declaring variables

Declare variables using the `let` keyword:

```
# let i = 4;;
# let x = 1.049;;
# let b = true;;
# let s = "a string";;
# let c = '!';;
```

Arithmetic and logical expressions

The usual arithmetic operators — `+`, `-` (unary and binary), `*`, `/` — are used for integer operations. For floats, you need to follow the operator by a period — `+.`, `-.` (unary and binary), `*.`, `/.`; for floats, you also have `**` for exponentiation.

Comparison operators are `=`, `<`, `>`, `<=`, `>=`, and `<>`. Unlike arithmetic operators, these are overloaded; they can be used either to compare integers or floats (although you can't mix the two in a single comparision).

The logical connectives are `&` (and), `or`, and `not`. String concatentation is denoted with the caret (`^`).

You can't do much with characters except by converting them to integers and back. Use functions `int_of_char` and `char_of_int` for these two conversions.

Function calls and function definition

Function call and definition are the most obvious places where OCaml syntax differs from C or Java. Function calls are indicated by juxtaposition: "f x" means apply f to x (i.e. f(x) in most languages). "f x y" means apply f to x and y (i.e. f(x,y)).

Define functions using the `let` keyword:

```
let double x = x + x;;
let hyp x y = (x*.x +. y*.y) ** 0.5;;
let perim x y = x +. y +. (hyp x y);;
```

The big ticket item here is the lack of a return statement. In OCaml, there are no statement, only expressions. So the body of a function is always an expression. The value of that expression is always returned as the result of a call to that function; hence, there is no need to say "return".

Note that parentheses are still used for grouping, as usual — just not for surrounding function arguments. Note also that the usual precedence rules apply. There is one new precedence rule: function application takes precedence over everything else. This means that "f x+1" is equivalent to "(f x)+1"; If you want "f (x+1)", you have to write it that way.

Lexical issues

As in C and Java, OCaml programs are generally free-form — you can insert spaces or newlines pretty much at will.

OCaml comments use the syntax "`(*` ... `*)`". These are like C's traditional multiline comments (`/*` ... `*/`). There is no single-line comment (like C++'s `//`). Also, unlike in C or Java, OCaml's comments *nest*: If you write "`(* ... (* ... *) ... *)`", OCaml will treat the entire thing as a comments; if you write "`/*` ... `/*` ... `*/` ... `*/`" in C, the first `*/` terminates the comment. OCaml's version is a lot handier for commenting out entire chunks of code, if the chunks contain comments.

Conditional expressions

C and Java have a conditional expression called `?:`, which is used as follows: e1 ? e2 : e3. If e1 is true (or non-zero, in C), then this has the value of e2, otherwise it has the value of e3. This type of expression is not that well known or frequently used.

OCaml has a conditional expression which is very heavily used — since OCaml has no statements, it is used as much as the conditional statement (if) in C or Java. Its syntax is: `if` e1 `then` e2 `else` e3, and it works just the same as the `?:` operator.

```
let max x y = if x<y then y else x;;
```

Parentheses can be used to resolve ambiguities when using conditional expressions. The first two of these expressions are equivalent; the third is different:

```
if x<y then y else x + z;;
if x<y then y else (x + z);;
(if x<y then y else x) + z;;
```

Sequencing and output

Sometimes you will want to print some values for debugging, and then continue with the computation as before. To do this, you need a way to print, and a way to sequence expressions.

For sequencing, use semi-colon: "e1; e2" evaluates e1 and then e2, and returns the value of e2. This doesn't normally make sense — why evaluate e1 if you're just going to discard its result? — but it does if e1 has a side effect — like printing.

For printing, use `print_int`, `print_float`, etc. For example, to print x and y before returning their max:

```
let max x y = print_int x; print_int y; if x<y then y else x;;
```

Recursive definitions

Most of your functions will use recursion. To define a recursive function, you cannot use the "let" notation as is; you need to add the word "rec":

```
let rec fac x = if x=0 then 1 else x * fac (x-1);;
```

If you want to define mutually-recursive functions, there is a problem: whichever one you define first refers to the other one which isn't defined yet; that's not allowed in OCaml. To solve this, define them in the same "let rec" using the keyword "and":

```
let rec even x = if x=0 then true else odd (x-1)
    and odd x = if x=0 then false else even (x-1);;
```

What's missing

Obviously, we've covered only a small part of OCaml. You'll learn new features and syntax throughout the semester. Here are some obvious missing things we'll eventually get to:

Assignment statements. Assignment statements exist in OCaml, but they are used very sparingly, and we will have little use for them. You will have to learn to program without them, which brings us to...

While loops. Without assignment statements, while loops have essentially no uses (think about it). Instead, we use recursion. So, again, you need to learn to program using recursion instead of loops.

Aggregates. We have not said how to use arrays or any other kind of aggregate data structure. The most important aggregate in OCaml is *lists*; we will discuss those in lecture 2.