Assignment 1: OCaml notes

`.mli` files

For this and all subsequent assignments, we will be supplying you with one or more OCaml *interface files* to be used with your code. For this assignment, the file is called `lab1.mli` (note the `.mli` extension; all OCaml interface files have this filename extension), and you should download it into the same directory that you are using to write and test your `lab1.ml` code.

The interface file consists mostly of type signatures of functions, though occasionally it will have other things as well. The `lab1.mli` interface file looks like this:

```
(* Interface file for lab1.ml *)

val sum_squares_to : int -> int
val sum_of_squares_of_two_largest : int -> int -> int -> int
val factorial : int -> int
val e_term : int -> float
val e_approximation : int -> float
val is_even : int -> bool
val is_odd : int -> bool
val f_rec : int -> int
val f_iter : int -> int
val pascal_coefficient : int -> int -> int
```

The `val` declarations indicate that the type signature of a particular value is being described. Here, all such values are functions (functions are values in OCaml). For instance, the `factorial` function has the type signature:

```
val factorial : int -> int
```

which indicates that it takes one argument (an `int`) and returns an `int`, as you would expect. Functions which take more arguments (like `pascal_coefficient`, which takes two integer arguments) have somewhat less intuitive type signatures:

```
val pascal_coefficient : int -> int -> int
```

You might have expected something like this instead:

```
val pascal_coefficient : int int -> int  (* WRONG *)
```

The reason why this is wrong is that arguments to OCaml functions are automatically *curried*, which means that they can be partially applied. (The name "curried" is a tribute to Haskell Curry, a logician who provided much of the theoretical foundation for modern functional programming languages.) In this case, it means that if we call `pascal_coefficient` with only one argument (an integer), it will return a function that takes the other integer argument and returns the integer result. Currying can occasionally give rise to confusing error messages, but it's also extremely handy in practice, as we will see.

OK, let's assume you've written all of your code, and you want to check that it conforms to the type declarations in the `.mli` file. How do you do that? The simplest way is to compile your code along with the interface file from the command line:

```
$ ocamlc -c lab1.mli lab1.ml
```

If no error messages are printed, your code is at least type-correct! Also, if you list the files in your directory, you will see two new ones: `lab1.cmi` and `lab1.cmo`. These are the (byte-code) compiled versions of the `lab1.mli` and `lab1.ml` files, respectively. Note that you have to put the `.mli` file before the `.ml` file on the command line; this won't work:

```
$ ocamlc -c lab1.ml lab1.mli
```

unless the `.cmi` file has already been compiled, in which case you don't have to have `lab1.mli` on the command line anyway. This is a bit annoying, but we live with it.

You can now load the `.cmo` file into an interactive OCaml session as follows:

```
# #load "lab1.cmo";;
```

In this case, nothing will be printed if there are no errors. A different way to load your code is to use the `#use` command:

```
# #use "lab1.ml";;
```

If you do this, then OCaml will compile your code and print out the signature of every value in `lab1.ml`. When using `#use`, you don't need to compile your code beforehand. When using `#load`, you do. As a result, we tend to use `#use` more than `#load` when interactively developing code. You should know both forms.

Let's go back to what would happen if you typed:

```
# #load "lab1.cmo";;
```

You might expect that you could then use all the functions in `lab1.ml` (as you could if you'd used `#use`). Actually, that isn't the case (yet). If you try, this will happen:

```
# pascal_coefficient;;
Error: Unbound value pascal_coefficient
```

Huh? We just loaded `lab1.ml`, and `lab1.ml` defines `pascal_coefficient`, so why do we get an error message? It turns out that `#load` loads the code as a separate *module* called `Lab1` (the name of the file, without the extension, and with the first letter capitalized). This is like saying `import lab1` in Python. We can get the function by using the module name as a prefix:

```
# Lab1.pascal_coefficient;;
- : int -> int -> int = <fun>
```

Notice that we have to capitalize the first letter of `Lab1`. Module names are always capitalized in OCaml.

If this is too tedious, you can dump all the names in the module into the local namespace by using an `open` declaration:

```
# open Lab1;;
# pascal_coefficient;;
- : int -> int -> int = <fun>
```

This is like saying `from lab1 import *` in Python. Alternatively, if you used `#use` to load and compile the code, all the functions in the file are available immediately. This is another reason why `#use` is used more often for interactive development.

It's important to realize that `#use` and `#load` are special commands of the OCaml interactive interpreter (also called the "toplevel"); they are *not* part of the OCaml language itself. (There are other special interpreter commands as well, which we will get to when we need them.)

We will have much, much more to say about the OCaml module system in future assignments. OCaml actually has the most powerful module system of any computer language in wide use.

Once you have finished writing your code, you should always compile it against the `.mli` file we provide to check that your code not only compiles, but also has the type signature we want. (Using `#use` will check that your code compiles, but it may not have the type signature we want.)

In later assignments, we will introduce the `dune` compilation manager, which will automate all the tedious parts of compilation.