

---

# CS 421 – Spring 2007

## Lecture Notes Set 2:

### Introduction to OCaml

Elsa L. Gunter<sup>1</sup>

Transcribed by: Pooja Mathur

---

**Corresponding to Slides:** 01-intro (slides 15-27)

Made available: January 19, 2007

Revision 1.0

---

## 1 Change Log

1.0 Initial Release.

## 2 To Use OCaml

### 2.1 Logging in

The following are details on how you can remotely get to the machines that run OCaml and from which you will need to submit your homework. Also note that it is advantageous, but not necessary, to run an X terminal. Any other terminal will allow you to follow the steps, but if you wanted to use start using emacs or something, you will want to be using an X terminal.

- Start off with typing: `ssh -X netID@dcllnx[1-41].ews.uiuc.edu`
  - If you do not type in `-X` then X windows packets will not be allowed through
  - NetID is your netID
  - `dcllnx` can be replaced by `eelnx` (those are the other EWS Linux machines)
  - Pick a number between 1-41 for the `dcllnx` computers, 1-47 for the `eelnx` ones
- If you haven't logged into that machine before, it may ask you if you want to accept a security key from the machine, you should say yes.
- Type in you AD (Active Directory password).
- You should now be connected to that EWS machine.

### 2.2 Using OCaml

Type `/usr/local/bin/ocaml` and press enter. You will now find yourself running OCaml.

- Now that you are able to run OCaml, you can use this in a way, for example, like a calculator. If you type:  
`# 3 + 4;;`  
And you will get for an answer:  
`- : int = 7`

---

<sup>1</sup>© 2007, Share and Enjoy

- The double semi-colon is the command to tell OCaml it is ok to compile. It is telling it that you have finished entering as many declarations that you intend to enter at this time and it will go ahead and evaluate them.
- So you can see that you you can use this as an interpreter.
- To exit OCaml, ctrl+d. Then you get back to the UNIX prompt. Then if you want to leave the whole system, you should logout.
  - A couple notes on the OCaml that you are running: The one that is being run on these machines are a few years old. If you were to go and download a version from the INRIA website, you will get a more recent version, 3.09. This new version is not compatible with the compiler version that currently runs on the EWS machines. It has been asked of EWS to upgrade their version and this should happen within a couple weeks. So if you were to run 3.09 locally it should soon be compatible with the version on the EWS machines.

### 3 Programming in OCaml

Comments on the slides

- The fuschia color is all printed by the computer
- The orange are indicative of keywords or special symbols/characters

#### 3.1 Some of the basics...

- The open parenthesis followed by an asterisk, (\*, starts a comment. It is closed by the asterisk and then the closing parenthesis, \*)
  - Using this notation, comments can be multiple lines long.
  - You may put comments within comments, unlike C/C++'s `/**/` notation, OCaml's `(**)` come in balanced pairs. Use your comments wisely, this can make writing code easier, but reading code harder.
- In OCaml there is a distinction between expressions and declarations. An expression is a term that will evaluate to a value. Using the `let` keyword, you can declare a variable. A declaration will contain an expression, but it will also cause a binding to occur. The variable is bound to the expression. You do not have to tell it type information. OCaml figures it out for you.
- OCaml is different from most languages, variables are more like identifiers. This is because you cannot reassign a variable once it has been evaluated. The name of the variable can be reused and be associated with a new value, but a particular variable cannot change.

#### 3.2 Environments (slide 16)

When making a declaration, you are adding to a table. That table has a mathematical abstraction called an environment. There is a recording of association with names to data objects (a value). So when you make a declartion, you are changing the state of the current environment. It adds to it, possibly hiding something that was already there, a new binding, a new association.

When we make that declaration, you might be obscuring something that was in the list with that name. So, in essence, you are doing a functional update. You can think of an environment as being a partial function, but in terms of implementation, it is generally stored as a stack. It simply associates a name with a value. Since it is a stack you will be associating the most recent versions of identifiers with the most recent values, and not the later ones. For the moment, you should get you mind around the idea of this stack, and in terms of implementation, you can think of it as an association list. But you also want to think of it as a mathematical function, a partial function, something that is not defined for all names, just for the names that have been introduced so far.

### 3.3 Types in OCaml (slide 17)

- Integers
  - You have already seen integers in OCaml, they are 31 bits long. There is a 32-bit integer, but that type is `int32`. The reason for the difference with the types is a difference in internals for storing information.
- Strings
  - To use strings, you start with double quotes, `"`, you have your string, and then you end with double quotes, `"`. When we want to use a character not available to us, for example, a new line, you have to use the new line escape sequence, which is, slash then the letter n. Or for a tab, slash and then the letter t. It obeys unix conventions.
  - You can call a procedure on the string, like `print_string`. This interprets the escape sequence and for something like slash n, you will get a new line.
  - Also, you see there is something called unit there. Unit is the value that is associated with the result of print string. The unit is a little different than void from C++, you can't make a stack of void. In OCaml, you can make a list of units. Units are treated as entities in OCaml, but in most instances they are equivalent to void.
- Sequences
  - Expressions - A semi-colon in the middle of two expressions causes the sequences of the evaluation of those expressions. It will evaluate the first expression, and then the next. The result of a sequence of semi-colons is the last value in the sequence. Even though that last value is your result, it does evaluate each expression in the sequence in order. If you forget the semi-colon in between expressions, it turns out to be a type error. So OCaml would try to evaluate it, but it would not be what you were looking for. It is a function application.
  - Declarations - Now, using the double semi-colons, you can list numerous declarations of variables, and have it not try to compile, until after your sequence has completed. Then, OCaml will evaluate each declaration in the order at which it occurs, and only leave this as one increment to the environment. There could be numerous new declarations, but it will still only move the environment stack by one. The binding occurs, however, as the declarations are evaluated. So this does not lead to a simultaneous declaration, but, in fact, sequential declarations. No semi-colons are needed to separate declarations in the sequence. Unlike sequencing expressions, you will get a syntax error.

### 3.4 Variables and Binding (slides 18-20)

Expressions do not change the environment. However, evaluating declarations will. Keep in mind that OCaml's environment, when starting to run it is not empty, it comes with the basics, numbers (1,2,3,4,...), operators (+,-,=,...), etc, but will take that as given. So, the test example, maps the name test to the value false, it does not map test to the expression `3 ; 2`, but the evaluation of that expression. This mapping creates a new environment. The next sequential declaration example creates an even newer environment. Generally, when you have new values, multiples of the same names does not make sense. But in this case, with an association table, the lookup will move from left to right and will use the first value it sees, ignoring the rest.

Before, it was stated that when you evaluate an expression, it does not alter the environment. However, it may be useful to have a local binding, one that is evaluated for some other expression or function, but does not last outside of that scope. This works with the `let...in` construct. One uses this inside of an expression. The example shows that b, for the scope of the expression is evaluated to `a+a`, which from the previous code, a is equal to 3. So while the expression `b*b` is being evaluated, we see b as 6 giving us the final result as 36 for c. Now that the expression was completed, b is returned to its original value of 5. So this value of b is just hidden during the scope of the expression. But is returned once it has been evaluated. Keep in mind that you are not reassigning to b, you are creating a "new variable" that only has the life span of that expression.

You will get a temporary environment while inside the expression. The old environment is hidden. At the completion of the expression. We are returned to the old environment. Technically, the value of `b` being equal to 5 is still in the list during the evaluation of the expression, but it is further to the right. So during any look up, the 5 will never be seen. Because of this, only the currently functional variables in the environment are being displayed, but don't let this throw you, the `b` mapped to 5 is still there.

- Terminology note - Output can either refer to the result returned from a function application and also text that is printed to the screen. Usually, the term output will be used to refer to the result of some function. However, it may once in a while be referring to something printed to the screen. Remember that it is an overloaded term in English, but ask if you are confused.

### 3.5 Overloading (slide 22)

OCaml likes everything to be differentiated, unlike many other languages. OCaml does not allow overloading for anything. So, `+` and `*`, etc, only operate on integers. They do not work on floats. To use such operations on floats you have the `+.`  and the `*.` , etc. operators. If you try using `+` on floats you will run into type errors. OCaml tries every hard to tell you what it thinks is wrong. The error messages may not be right on the mark, but they do get very close. At first, they may seem a bit cryptic, but it well worth your while to try to figure them out.

The example in the code is telling something is wrong with the characters starting at the 8th characters (the first character of the line is in the zeroeth spot). Then it goes until the 12th character, exclusively. In other words, the 11th character is the last character you care about. Don't let the notation fool you. When it says characters 8-12, it means characters [8-12) or inclusive-exclusive. The error message is telling you that you are using a float where it expects an integer. But really you want to be using a different operator, the `+.`  operator to be exact in this example.

### 3.6 Implicit Coercion (slide 23)

OCaml cannot figure out for you that when you are using a float, like 1.0, that you want it to use the integer 1 instead, like in the example. Or if you had the expression `1.0 +. 2;` it will not be able to convert the 2 into 2.0, the float. Both cases will result in a type error. Type procedures must be used to change types, for example, there is the procedure `IntToFloat()`.

### 3.7 Booleans (slides 24-25)

OCaml has the normal booleans, `true` and `false`.

OCaml has a few special functions specifically for the use of booleans. The first is `if...then...else`. It is not exactly a procedure, because it only evaluates one part of the expression. When evaluating the `if/then/else` expression, the `then` and `else` parts must evaluate to the same type. You can't have an integer in one and then a float in the other.

- `&`: evaluates and - When using `and`, OCaml evaluates the first expression and immediately stops if it is false. OCaml only looks at the second expression if there is a reason to. In the example, it does check the second expression.
- `or`: evaluates disjunctions - Using `or` is similar in that it only checks the second expression if it has to, so if the first expression is true, it stops and returns true. That is what happens in the example.
- `not`: evaluates the expression then flips it

For the examples involving the print statements, the print only happens if the expression associate with with results in true. Even though both evaluate to true overall, the associated expression matters. The problem is, what if the associated expression with the "bye" statement was true. It would still not print out, and that is because OCaml would have seen that 3 was greater than 1 and stopped there. Because of this they are not proper procedures because they do not evaluate all of the arguments.

### 3.8 Functions (slides 26-27)

Functions are first-class objects in their own right. They have a type and therefore can be put into data structures.

The easiest way to make a function is using the `let` keyword. The name your function, your parameters, in the example, it is `n`, and then equals, and what your function does. What is really happening is that you are creating a function that takes a variable `n` and maps it to the expression of `n+2`, as seen in the example.

This generally works the same as you have seen so far. The declaration creates a new binding and the binding creates an alteration in the environment. You get a name mapped to a value, the difference now is that the value is more complicated.

However, you do not need to give names to your functions. You can do all sorts of things with this. You can apply it to a value, stick it in a list, put it in a pair, you can even pass it as an argument to another function.

Also note that these functions use variables that had already been declared. What this does is cause the previous variables to be hidden and you are now, in the scope of these functions, using different, newly assigned variables. So before, `x` was equal to 24. That gets hidden and when the first function is applied to the 5, you get then answer 15. At that point, `x` goes back to 24.

Note: the asterisk in the example pair is simply declaring that it is a pair, it is not multiplying the functions together.