
CS 421 – Spring 2007

Lecture Notes Set 3:

Introduction to OCaml - Continued

Elsa L. Gunter¹
Transcribed by: Pooja Mathur

Corresponding to Slides: 01-intro (slides 26-47)

Made available: January 22, 2007

Revision 1.0

1 Change Log

1.0 Initial Release.

2 Basic Procedures

2.1 Basic Syntax (slides 26-27)

- Use the keyword `let` to specify that you are introducing the binding of a name
- Follow with the name of the procedure
- Next is a sequence of arguments (can be a variable, function, can be multiple variables, functions)
- Then an equal sign
- Finally comes the body of the function
 - Behind the scenes this there is mapping from the `fun` keyword to the body of the procedure with the part `"fun n → n + 2"`. Because of this, you do not need to bind a function to a name.

So you can use the syntax `"fun [args] →"` to write nameless functions. Another example is making a pair out of two nameless functions. By doing this, you can pull out either of the two functions at any time and use them. So, you have a data structure of functions.

Functions in OCaml are first class data, they can be applied, put into data structures, they don't need names, they are expressions in their own right, and they can be passed as arguments into another function or even returned from function as you will see later.

Remember, when a procedure uses a variable name, the scope of that variable is only for the life of the function while it is running. The environment returns to its former state after the procedure has completed.

2.2 An Example (slides 28-31)

Here is an example to consider:

```
# let x = 12;;  
val x : int = 12  
# let plus_x y = y + x;;  
val plus_x : int → int = <fun>
```

¹© 2007, Share and Enjoy

The function `plus_x` has one parameter of `y`. The body of the function is `y + x`. Before that `x` had been declared to be 12. That goes into the environment. So, when running the procedure on 3, that is, `plus_x 3`, you get 15 as your answer because `x` is 12 in the environment at the time.

```
# plus_x 3;;  
- : int = 15
```

But what if we redeclare `x` to be 7.

```
# let x = 7;;  
val x : int = 7
```

Now, this environment has a new declaration of `x`, and for any new uses of `x`, 7 will be used. The old value is still in the old environment, that environment is just hidden. So note, that does not delete the old value of `x`, there are two declarations of `x` in the overall environment stack. However, for old uses of the name `x`, maybe something from the previous environment, like using `plus_x` again, which `x` is going to be used? Well, the old environment that contains the declaration of `plus_x` had 12 as the value. So even though you have hidden 12 from any future use as `x` (unless you declare it as such again) that value is bound to the `plus_x` procedure. So running `plus_x 3` will still give you the value of 15.

```
# plus_x 3;;  
- : int = 15
```

In something like C/C++, which uses assignment, `plus_x 3` would return 10. If you were to do something called "call by name", you would also get 10. However, OCaml works differently, it is called "call by value" and you get 15.

So overall, when learning any new programming language, you need to figure out, what method of evaluation does this language use? What is the scope of the variable?

2.3 Procedures with more than one argument (slides 32-33)

To use more than one argument, all you do is add a space and the name of that argument. So in the example, we have `add_three x y z` where `x`, `y`, and `z` are all arguments of the function. The type of `add_three` works like this. You have `add_three` is a function that takes an integer and returns a function that takes a integer that returns another function that takes an integer and finally returns an integer.

```
# let add_three x y z = x + y + z;;  
val add_three : int → int → int → int = <fun>
```

When the procedure is run on the example, `add_three 6 3 2`, what is happening is `add_three` is run on 6 which returns a function. That is then run on 3 which will return another function, and then is finally run on 2 which returns an integer, the result of 11.

You can write something that says `add_three 6` which actually makes sense in OCaml. If you wrote that you would get a function that needed two arguments and had 6 already in there as the original argument. Note that there is no default values for any of the arguments, so if you typed `add_three 6`, nothing would happen with the procedure until it received its last two arguments.

So, if you had a function that took a function and an integer, how do you differentiate between passing in a function "add_three 6", since that is a function and then an integer, as opposed to `add_three` and the number 6 as the integer. The answer is that you use parentheses strategically. Place parentheses around the `(add_three 6)` if you want that to be the function with another integer as the other argument. Otherwise, it would interpret it as first `add_three` and then 6 as another argument.

```
# let h = add_three 5 4;;  
val h : int → int = <fun>  
# h 3;;  
- : int = 12  
# h 7;;  
- : int = 16
```

Now, this example, has `add_three 5 4` and that returns a function, as we have learned. It then binds `h` to that new function. So when `h 3` is run, that actually means something first of all, and then it turns out to be adding 9 to 3, which is expected, since `h` is a function that adds `5 + 4 + z`. In the first case `z` is 3. The second time, `z` is 7 because `h 7` was run. The result from that is 16.

2.4 Passing Functions as Arguments (slide 34)

```
# let thrice f x = f (f (f x));;
val thrice : ('a → 'a) → 'a → 'a = <fun>
```

In this example we have a function, `thrice` which takes a function and an integer and what it does is applies the function to the integer and that returns an integer, and then the function is applied to that value and returns something, and then the function is applied on that value and finally returns it in the end as the final result. Note that the parentheses are important. Without them, the function won't be doing what you want it to.

When using a function as an argument, you have to make sure you are using it type correctly. That is, it is a function, so you should treat it like one. It doesn't make sense to add a function to a function or something along those lines. And for functions, one way of using it that is type correct is applying it.

Note the type of a function passed as an argument. It isn't a float or an integer or a boolean, OCaml says 'a. This means that you can send in what ever type you want into the thrice will be able to work with it. However the function that you send in has to be compatible with the variable to give it. It won't make sense to pass in a function that does addition and then give it a string as the variable. Because the thrice function works this way, you also can only send in a function that takes in a type and returns that same type. If you were to use thrice with a function that does not return the same type as it takes in, you would get a type error. The reason is because the second time the argument function tries to run, it is taking in the wrong type. Like `print_string`, takes a string and returns a unit. You can't send `print_string` into thrice, because it would eventually try to take in a unit as an argument and that doesn't make sense to `print_string`. However it is possible to write a function that takes a function that returns something different than its input, but it's much more complicated.

```
# let g = thrice plus_two;;
val g : int → int = <fun>
# g 4;;
- : int = 10
# thrice (fun s → "Hi! " ^ s) "Good-bye!";;
- : string = "Hi! Hi! Hi! Good-bye!"
```

When you apply thrice to `plus_two` and send in 4 as the argument. You get 10. It does that by taking 4 adding 2, you get 6, then add two to that, you get 8, and you add 2 and get 10 as your final answer. Then with the strings example, the function concatenates the strings in the function and then ends with the last string "Good-bye!". Note that the thrice used with the strings was the same thrice that was used with the integers. It did not get re-compiled. This is known as structural polymorphism. The thrice function does not care about the type of x.

2.5 Overview so far (slide 35)

So far we have seen that we can use functions as first class citizens. We can use them in data structures, we can pass them as argument, return them as results, do partial applications of them.

Now, the question is, what is it that we got back after only using a partial application of a function. Like the `h = add_three 5 4` example. That is a "value". It is the result of a compilation and evaluation of a function. The thing we get back is called a closure. The closure basically the description on how to use the function. It holds three bits of information, the arguments needed, the body of the procedure, and the environment when the procedure was written.

2.6 Recursive Functions (slide 41)

This is function that in the process of evaluation needs to be able to call itself on simpler input.

```
# let rec factorial n = if n = 0 then 1 else n * factorial (n - 1);;
val factorial : int → int = <fun>
```

To declare a recursive function, you need to use the `rec` keyword after the `let` keyword. If you were to forget this, the supposed recursive call would actually be calling a previously defined function of the same name. For example, `factorial`, the `factorial (n - 1)` call would not be calling this function, it would call the `factorial` that was last defined. And since this function is only currently being compiled, it wouldn't make sense, without the `rec` keyword for `factorial` to call this function.

For recursive functions to work you need to understand two things. First, you need to understand your base case. The next thing is the recursive case. So basically this is similar to induction, where you have your base case and your inductive step.

3 Tuples and Pattern Matching

3.1 Tuples (slide 42)

- Pairs - get two pieces of data
- Triples - gets three pieces
- 4-tuples - get 4 pieces, etc

– Note: The types of data for each component of a tuple do not have to be the same.

```
# let s = (5,"hi",3.2);;
val s : int * string * float = (5, "hi", 3.2)
```

The commas make the data structure. When you see an asterick in a type, it is part a tuple. The examples shows an `int * string * float`.

```
# let x = 2, 9.3;; (* tuples don't require parens*)
val x : int * float = (2, 9.3)
```

Also, note that the parentheses are not necessary. They are only there to help with disambiguation and parsing. But technically you can write that `x = 2, 9.3` and that turns out to be equivalent to `x = (2, 9.3)`. Remember, above it was stated that the commas make this structure.

3.2 Pattern Matching (slides 42-44)

```
# let (a,b,c) = s;;
val a : int = 5
val b : string = "hi"
val c : float = 3.2
```

To get to the components of the tuple you can use pattern matching. So, `let (a,b,c) = s` is telling OCaml, that you know `s` has is a triple, basically. And each component of `s` is mapped to the pattern that you listed. So `5` is bound to `a`, `"hi"` is bound to `b`, and then `3.2` is bound to `c`. They also then become part of the environment.

Note that you can't you the same variable while pattern matching. For example `let (a,b,a) = s` won't work. OCaml won't let you bind the same variable to differnt components, even if they turn out to have the same value.

You can also nest patten matching.

```
# let d = ((1,4,62),("bye",15),73.95);;
val d : (int * int * int) * (string * int) * float = ((1, 4, 62), ("bye", 15), 73.95)
```

The example is a triple. But it's components are a triple, a pair, and a float. Now, to pattern match, it's slightly more complicated.

```
# let (p,(st,_) = d;;
val p : int * int * int = (1, 4, 62)
val st : string = "bye"
```

To get to the components, you can use `let (p,(st,_) = d` and this gets the triple and binds it to `p`. The next part of the pattern matching takes you into the pair itself. The `st` is the first component of the pair and that binds to the string `"bye"`. The underscore in matched to what ever place it holds. But the important part is that there is no binding with the underscore. So the float at the end is matched to the underscore as well, but again, is not bound. Think of the underscore as a wild card. It matches to any possible type, but is only used for structural pattern matching.

```
# let fst_of_3 (x,_,_) = x;;
val fst_of_3 : 'a * 'b * 'c → 'a = <fun>
# s;; (*remember, s was declared earlier*)
```

```
- : int * string * float = (5, "hi", 3.2)
# fst_of_3 s;;
- : int = 5
```

So, to extract the first component of a triple, for example, you write the function `fst_of_3 (x,_,_) = x;;`. This means that the 2nd and 3rd components can have any other type at all, it doesn't matter, you just want to get the first one. So remember that `s` was `(5, "hi", 3.2)`. When you run `fst_of_3 s`, you get `5`.

3.3 Matching Expressions (slide 47)

So what if a pattern can match in different ways. So, what if you wanted to turn a triple into a pair. You write a match statement. Basically, what is happening is that you match triple (the argument you passed in) with the pattern `(0, x, y)` and if that is the case, you map it to the pair `(x, y)`, or if the pattern was `(x, 0, y)` you map it to `(x, y)` or if the pattern was `(x,y,0)` or you have the case `(x, y, _)`. Basically you want to remove the 0 from the triple, or get the first to components alone. However if you did not include the last line, OCaml would give you a warning, saying that your match was incomplete. There are possible triples that would not be handled by the other cases. So if you ran the program with a triple that didn't fit the profiles listed, you would get an exception error.

```
# let triple_to_pair triple =
  match triple
  with (0, x, y) → (x, y)
       | (x, 0, y) → (x, y)
       | (x, y, 0) → (x, y)
       | (x, y, _) → (x, y);;
val triple_to_pair : int * int * int → int * int = <fun>
```