

---

# CS 421 – Spring 2007

## Lecture Notes Set 4:

### Introduction to OCaml - Continued

Elsa L. Gunter<sup>1</sup>  
Transcribed by: Pooja Mathur

---

**Corresponding to Slides:** 01-intro (slides 35-56)

Made available: January 24, 2007

Revision 1.0

---

## 1 Change Log

1.0 Initial Release.

## 2 More on Functions

### 2.1 Closures (slides 35-40)

It was mentioned before that functions are technically values in OCaml. But the question is, what value is recorded when a function is declared. Well, the answer to that is a closure. A closure is almost just the function, but that doesn't entirely make sense. When you write a function, there are a few hanging ends. One is that you can use various, previously defined identifiers. And the way OCaml works, you want to be using the bindings that were present at the time the function was written and not some rewrite that happened later. So, when you try to remember the function to use it, OCaml has to remember the variables and their values at the time.

The next loose end is that your input argument is supposed to be taken as different than the variables used in the body of the program. This is because you don't want OCaml to look up the value of the argument from the old environment, the value is supposed to be the new input. So now, OCaml has to remember the body of the function, the identifiers in the function and keep separate which are supposed to be previously defined and which are the input arguments. Basically, this means a closure is comparable to a triple of sorts, where  $v_1$ - $v_2$  are the identifiers that are used in the expression  $exp$  and  $\rho_f$  refers to the environment that was around when  $f$  was written.

$$f \rightarrow \langle (v_1, \dots, v_n) \rightarrow exp, \rho_f \rangle$$

This is the value stored by OCaml when a function is written.

So, for example, recall the function `plus_x`.

```
#let plus_x y = y + x;;
```

Also, recall the environment at the time this function was written is as follows:

$$\rho_{plus\_x} = \{x \rightarrow 12, \dots, y \rightarrow 24, \dots\}$$

This will give the closure of:

$$\langle y \rightarrow y + x, \rho_{plus\_x} \rangle$$

So, when we use this, we want to use the inputted value for  $y$  and not the value in the environment, which is what the  $y \rightarrow y + x$  is referring to.

Now that we know how functions are stored, how do we use this information to make an evaluation of the function. This varies from language to language. First, one looks at the whole term and evaluates the outermost term. This is the side that starts with "fun". Then the term that is on the left is evaluated until one gets a closure. Once you get to that point, you evaluate your argument until it becomes a value of some kind. Then you create a new environment which is

---

<sup>1</sup>© 2007, Share and Enjoy

the environment that is inside the closure updated with the input identifier being mapped to the value that we just got for the argument. Then you proceed to evaluate the body in that new environment.

What this is doing is basically evaluating a function from the left most argument onwards. Recall `add_three x y z`. This evaluation is what makes it apply the function to `x` first and then to `y` and then to `z`. However, OCaml does allow some degree of freedom in that sometimes the order of evaluation can be left undecided. That is, whether you evaluate the function or the argument first, means that you have a level of non-determination in your function, which in turn can lead to different values or side effects in terms of printing.

So let's look at an example with `plus_x`:

First we have the environment at the time `plus_x` was defined.

$$\rho = \{plus\_x \rightarrow \langle y \rightarrow y + x, \rho_{plus\_x} \rangle, \dots, y \rightarrow 3, \}$$
 where  $\rho_{plus\_x} = \{x \rightarrow 12, y \rightarrow 24,$ 

Now we look at what identifiers are defined as arguments

Eval (`plus_x y`)

Next we evaluate the body:

Eval (`app < y → y + x, ρplus_x > 3`)

Then the input is placed into the environment:

Eval (`y + x`) in (`y → 3`) +  $\rho_{plus\_x}$

Lastly, the final evaluation is made.

Eval (`3 + 12`) = 15

We have been discussing to some degree the idea of environments of functions leading to the scope of variables.

So let's look at an example:

```
let x = 27;;
let f x =
  let x = 5 in
    (fun x → print_int x) 10;;
f 12;;
```

We start off by binding 27 to `x`. Then we define a function that takes the argument `x` and for the body of the function binds 5 to `x`. The expression continues by declaring another function that is then applied to 10. And we end by calling function `f` on 12.

So what value of `x` is printed?

Well, the `x = 27` will never be used because `f` takes in `x` as an argument. And by the previous discussion on how closures work, we know that arguments take precedence over bindings when used in a function. The 12 used when calling the function `f` is not used because once the function starts to run, the 12 is hidden by the 5. Then, the expression in `f`, the `fun x → print_int x` hides the 5 because the `fun x` part is saying that we have another function that is taking in `x` as an argument. Like the 27, the previously bound 5 is hidden. We are then left with the 10, which is then printed to the screen after the function is finished.

## 2.2 Curried and Uncurried Functions (slides 45-46)

We know that OCaml lets you write a function that takes an argument that returns a function that takes an argument that returns a function, etc.

What can happen now is instead of sending in multiple arguments, you can send in a more complex data structure that can allow the function to perform its same evaluation.

Recall the function `add_three` which had a type of:

$$\text{val add\_three : int } \rightarrow \text{ int } \rightarrow \text{ int } \rightarrow \text{ int} = \langle \text{fun} \rangle$$

The function takes an integer and returns a function that takes an integer, etc. But, how does that compare to the function `add_triple`

```
# let add_triple (u,v,w) = u + v + w;;
val add_triple : int * int * int → int = <fun>
```

This function takes a triple and does the same thing that `add_three` does, except, this function has "one" argument where `add_three` has three. `add_triple` takes all its input at once, there doesn't seem to be any partial application of the function. So, when writing functions, you have to think, which application of arguments is better for the function.

There is some terminology associated with these terms. Functions that take arguments one at a time are called *curried*. This means `add_three` is *curried*. Functions that take all their input at once are called *uncurried*. There is a function that you can write that can convert the a function from *curried* to *uncurried* and vice versa. This is referred to as *uncurrying* and *currying*.

You can make a function out of *uncurried* functions that make it seem like you are doing a *partial application*. You can define a function that takes an argument, for example, and applies it to a "partially" evaluated *uncurried* function. So, in the example below, a function was written that takes an argument `x`. With that it is used to finish evaluating `add_triple` which already has the input of 5 and 4. This is similar to the `add_three` partial function of `add_three 5 4` which was used the other day. Doing this is basically the idea of *currying*. We have just *curried* the function `add_triple`.

```
# fun x → add_triple (5,4,x);;
- : int → int = <fun>
```

The fully *curried* function would be `fun x y z → add_triple (x,y,z);;`

### 3 Lists (slides 48,51)

Lists are the first example you will see of something that is called *recursive data types*. A *recursive data type* is a data type that is allowed to have components in it that are all of the same type as total data structure itself. A pair is not a *recursive data type*. A pair can have pairs as it's components, however the type of the inner pairs are a different type than the total pair. A list will have a head element and then a tail list will have exactly the same type as the whole list. This is also known as *algebraic datatype*.

Tuples can have varying data types for its elements. Lists on the other hand have to be of the same type. Lists are called *homogeneous* because of this. So the following list won't compile in OCaml.

```
#let bad_list = [1;3.2;7];;
```

#### 3.1 Making Lists (slides 49-50, 52-53)

Making lists There are two forms of lists.

- The empty list which is written as `[]`
- Non-empty list is written as `x :: xs`
  - `x` is the head element of the list
  - `xs` is the tail list of the whole list, which means `xs` is not just one element, it can be, but it can also be numerous elements
    - \* The `::` is called "cons"
    - \* The `x`, and `xs` are just variable names, you don't have to use `x` and `xs` when referring to lists. You can call them what ever you like.

Examples of the list structure:

```
[x] is equivalent to x :: []
```

```
[x1 ;x2 ;x3 ; ... ; xn] == x1 :: x2 :: x3 :: ... :: xn :: []
```

Note: You must use semi-colons inside the square brackets. If you use commas, you won't get an error. However, OCaml will treat this as a tuple of size `n`.

Examples of lists

The first is the a list of the first five fibonacci numbers.

```
# let fib5 = [8;5;3;2;1];;
```

```
val fib5 : int list = [8; 5; 3; 2; 1]
```

This next example takes the 6th fibonacci number and adds it to the beginning of the `fib5` and binds that to the name `fib6`. This works because you have a element to start the list followed by a list of the same type.

```
# let fib6 = 13 :: fib5;;
```

```
val fib6 : int list = [13; 8; 5; 3; 2; 1]
```

You can check your values in a list by writing out the list and seeing it's equivalence to the name.

```
# (8::5::3::2::1::1::[ ]) = fib5;;  
- : bool = true
```

Also, you can append an element to a list or a list to a list with the @ symbol.

```
# fib5 @ fib6;;  
- : int list = [8; 5; 3; 2; 1; 1; 13; 8; 5; 3; 2; 1; 1]
```

Here is a little exercise

Which one of these lists is invalid?

1. [2; 3; 4; 6]
2. [2,3; 4,5; 6,7]
3. [(2.3,4); (3.2,5); (6,7.2)]
4. [[hi; there]; [wahcha]; []; [doin]]

Well, to check, look at the types. The first is homogeneous, each element is of type int. The second is homogeneous, each element is a pair of ints. The third, the first two elements are a pair of first a float and then an int, however the third element is an int and then a float. Therefore, making the list heterogeneous and therefore, invalid. Finally, the last one is homogeneous, each element is a list of strings. Even the 3rd element in that list is ok because it is the empty list of strings.

### 3.2 Functions over Lists (slide 54-55)

When writing functions over list, pattern matching is a little different. Lists will either be the empty list or the cons.

So, say you want to write the function, `double_up`. The function will take in a list and return a list where all the elements in the original will have been doubled. (Not double the value, but rather instead of one element, there will be two elements of the same value.

```
# let rec double_up list =  
  match list  
  with [ ] → [ ] (* pattern before →, expression after *)  
       | (x :: xs) → (x :: x :: double_up xs);  
val double_up : 'a list → 'a list = <fun>
```

So, to start with you have to decide what your list looks like. This is where pattern matching comes in. Either you will be starting with the empty list, in which case you return the empty list, "or", which is the bar in the code, you have at least one element. In that case, you write the head twice and you perform the operation on the rest of the list, aka the tail list. So, this pattern works well even if your list only has one element. That is because the `x` would stand for the head and the `xs` would stand for the empty list, or `[]`. So if you wrote `x :: y :: z →`, this means you are looking for a list with at least two elements. `x` is the head, `y` is the next element, and then `z` could be the rest of the list or it could mean the empty list. However, this is unordinary. It is generally, unnecessary to write complex patterns.

Note that when you are writing a function over a recursive data type, you can be certain, with 99% accuracy, that you will be writing a recursive function. Moreover, with 90% accuracy, your function will be followed with the structure of the recursive data type you are passing in (in this example, the empty list).

Now, if you ran this function on `fib5`, the following is what you would get returned to you.

```
# let fib5_2 = double_up fib5;;  
val fib5_2 : int list = [8; 8; 5; 5; 3; 3; 2; 2; 1; 1; 1; 1]
```

Another example would be the following. You have the list defined below and you apply `double_up` to it.

```
# let silly = double_up ["hi"; "there"];;  
val silly : string list = ["hi"; "hi"; "there"; "there"]
```

Another function one can implement is reversing a list

This is one way to reverse a list in OCaml. You take a list in, do your pattern matching. Then you append the head at the moment, to the end of the tail list.

```
# let rec poor_rev list =  
  match list  
  with [ ] → [ ]  
       | (x::xs) → poor_rev xs @ [x];;
```

```

val poor_rev : 'a list → 'a list = <fun>
# poor_rev silly;;
- : string list = ["there"; "there"; "hi"; "hi"]

```

However, this is called `poor_rev` for a reason. This is a poor way of reversing the list because to append something to the end of the list, you have to iterate through each element in the list to get to the last element to append the next element. So this is a rather expensive function.

### 3.3 Functions applying functions on lists (slide 56)

This next example exemplifies how to run through a list and alter the elements of the by applying a function to it.

You start by declaring the recursive function and send it a function and a list.

```
# let rec map f list =
```

Then the pattern matching occurs. What the pattern matching is saying is, if you have the empty list, match it to the empty list, otherwise, match the head of the list to the new value you would get after applying the function to it. Then as recursive functions go, apply the overall function to the tail list.

```

match list
with [] → []
| (h::t) → (f h) :: (map f t);;
map : ('a → 'b) → 'a list → 'b list = <fun>

```

So, if you ran the `map` function on the function `plus_two` and the list `fib5`, the `map` function would add two to every element in the `fib5` list. In the second example, you are subtracting one from each element in `fib6`.

```

# map plus_two fib5;;
- : int list = [10; 7; 5; 4; 3; 3]
# map (fun x → x - 1) fib6;;
: int list = [12; 7; 4; 2; 1; 0; 0]

```