

---

# CS 421 – Spring 2007

## Lecture Notes Set 5:

### Lists and Recursion

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

Transcribed by: Pooja Mathur

---

**Corresponding to Slides:** 02-ocaml (slides 23-31)

Made available: January 26, 2007

Revision 1.0

---

## 1 Change Log

1.0 Initial Release.

## 2 More on Lists

Lists should be viewed in two ways, first as a very important complex data structure in functional programming. Lists and pairs together can allow you to do anything you may want to do with algebraic data types, but it helps to have the other structures around anyway. Lists give us the ability to have a recursive data structure. However, lists are also seen as the first of many examples we will see of recursive data types.

In a list, its elements are of the same type that the list is as a whole and so its definition involves itself and therefore is considered recursive. With this idea in mind, we will see that almost all the functions you write with lists will be recursive. We will have to write a function that is written in terms of itself. You have already seen from previous lectures, some examples of functions on lists.

### 2.1 Fold\_right (slide 23)

Lists are also a case where we will be wanting to write very generic functions for. We generally do not care what is in the list, only that the function maps correctly and traverses correctly. For example, when writing reverse for a list. You should not care if there are integers in the list or strings. All that should matter is that the order of the entries of the list are reversed. Basically, what this is saying, is that we only truly care about the back bone of the function. What you need to remember is that you need to separate in your head the function that you want to perform on the whole list from the function that you want to perform on each element of the list.

An example of this is the map function.

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

This function takes in a list and intends to perform some function  $f$  on each element of the head of the list and then recurses on the tail list. This works with the basic pattern matching that we have been using. The empty list maps to the empty list. That is our base case. We don't want to recurse on this, if we do, we will have an infinite loop. Then for the non-empty list, we match the head,  $h$ , to the value that is returned from performing function  $f$  on  $h$ . The tail list,  $t$ , is matched to the recursive call of  $\text{map } f \ t$ . So we are calling  $\text{map}$  again, but on a smaller portion of the whole list and we still want to perform function  $f$  on the rest of the list. This is an example of forward recursion. We perform

---

<sup>1</sup>© 2007, Share and Enjoy

the function and then move on to the next element. Then we perform the function and move on, etc, etc, until you get to the base case. Once you're there, all that is left to do is to return from all the function calls you made, no work is needed to be done, everything was done on your way down the list. This is probably the most common type of recursion you will see.

Note the type of the map function, it takes in a function that takes in an argument that returns something as well as taking in a list. Then it returns a list. Notice that the function takes in a argument of 'a and map takes in a list of type 'a list. Then the function f returns something of type 'b and then the whole map function overall, returns a list of type 'b list. In other words, you can see that what the function does is directly related to the list.

As for the function f, We have no idea what function f does, all we know is that we want to apply it to the elements of the list. So, using plus\_two as an example, when we run map on plus\_two and the list fib5 we get:

```
# map plus_two fib5;;  
- : int list = [10; 7; 5; 4; 3; 3]
```

But remember, we don't have to declare the function before hand. Here we simply write a function that takes an integer and subtracts 1 from that integer. We run that with fib6 and get:

```
# map (fun x → x - 1) fib6;;  
: int list = [12; 7; 4; 2; 1; 0; 0]
```

There are a few minor advantages to declaring the function directly in the map function call. First, if you are only planning on using the function for something like this, there is no point of declaring the function f elsewhere and taking up space in the environment for no reason. Another reason is for readability, if one does not intelligently name your functions, then you might have a hard time figuring out what exactly should be happening in the overall function, where as with the code written directly in, one can see, in this case, obviously, 1 will be subtracted from the each element in fib6. This does not make a huge difference. If you are writing a long complex function, then by all means, write it somewhere else, give it a name.

Another example of something you can do which a this is bind certain functions to the map function. That is, let's say you have multiple lists that you want to perform a certain function f1 on. You can bind the function map f1 to a name and then all of a sudden you have a new recursive function that performs f1 on the list you give it. You can run it multiply times and it works because you have made this binding. This was another example of partial application of functions.

## 2.2 Fold left (slide 24)

Another generic form of recursion requires you to recurse to the base case and then begin your accumulation of information as you return back from all the recursive calls. The opposite of this is when you pass down the work you have done so far on the list, as you recurse through. Then you get to the base case, you have been accumulating information the whole time and you are hopefully done. For example, if you wanted to add all the elements in a list, you would could start with the head element, start with that value and as you recurse through the rest of the list, sum up each element you pass by and then send that new value forward when making the recursive call. When you get to the base case of the empty list, you would be done because you have been summing up the elements as you have moved along.

There is an operator that will let you do this. It is called fold\_left. Here is an example.

```
# let rec fold_left f a list =  
  match list  
  with [] → a  
       | (x :: xs) → fold_left f (f a x) xs;;  
val fold_left : ('a → 'b → 'a) → 'a → 'b list → 'a = <fun>
```

The function f being passed in can be any function again. The argument a is the place holder of what is being accumulated and passed along. Then, list is the list being passed in.

What this is function is basically saying is that you have a function f that does some work on the head of the list and places that result in a. Then the recursion takes place, moving down the list, everytime adding more to the place holder, a. When you get to the very end of the list, you want to return the accumulated value, so the empty list is matched to a. The matching for the non-empty list, where it performs the function f on the head of the list, the (f a x) is what becomes the new accumulated value.

The type of the function works like this, the function takes something of type 'a and turns it to something of type 'b and then returns something of type 'a. That returned value is placed in a place holder of type 'a. Then the over all function does this on a list of type 'b list. After the recursion is finished, the final result of type 'a is returned.

Take the following example. The type sent in is of type unit which is converted to a string for print\_string and then results in something of type unit, since that is how print\_string works. Then resulting unit is placed in a place holder for units. The function is run on each element in the list below and the final return value is a unit, the unit of "hithere".

```
# fold_left
  (fun () → print_string)
  ()
  ["hi"; "there"];;
hithere- : unit = ()
```

Fold\_left is what is helping to accumulate our final value. It is making it so we don't have to worry so much about. Again, it is making it so you don't have to do any work after you get to the empty list.

## 2.3 Fold\_right (slide 25)

Now, let's look at fold\_right. This is very similar to fold\_left, the only difference is that here, we will recurse to the end of the list, and then do the work as the functions return. This is another example of forward recursion. We are going to keep looking forward until we get to the end of the list.

Now the example for fold\_right has three arguments. The f, like normal, the the function that causes the accumulation of values. list is the list you are sending in to be recursed over. And then b is your base case, it is what you are going to start working with once you get to the tail of the list.

```
# let rec fold_right f list b =
  match list
  with [] → b
  | (x :: xs) → f x (fold_right f xs b);;
val fold_right : ('a → 'b → 'b) → 'a list → 'b → 'b = <fun>
```

So, here, the function f takes two arguments. It takes the head of the list and the value that is returned from the performance of the recursive calls. The types work similarly. Function f takes in an 'a and a 'b and returns a 'b is performed on all the elements in a list of type 'a list. Then the next 'b refers to the accumulator being of type 'b and the last 'b in the type is the final return value for the whole function.

```
# fold_right
  (fun s → fun () → print_string s)
  ["hi"; "there"]
  ();;
therehi- : unit = ()
```

This example shows fold\_right in action. Basically, what happens is, the list is traversed to the end, then the actions of function f, which in this case ends up referring to a print\_string, happen. So, now that we traversed the list to its end, we can start performing the functions on the elements starting with the tail. So, we print "there" and then "hi"

To use a function like this you need to be prepared with two idea. First, what will be your base value? Then, what will be the function that will do the accumulation?

Now, fold\_left corresponds to iteration which corresponds to a for loop. FOlr\_right is a more general form of recursion but is not as powerful as other recursive functions over lists. This means there are functions that you can write that cannot be factored through fold\_right. But fold\_right also has another unique property, and that is that they are guaranteed to terminate. So, as long as the function f passed in as an argument terminates, then the whole function will be able to terminate.

Note: Map, fold\_left and fold\_right are the most important of the higher order operators for lists.

## 3 Recursion

### 3.1 General Idea (slides 26-27)

Now that we have seen recursion over lists, what about recursion in general? Well, general recursion in OCaml follows the same idea. You declare a recursive function that you want to use to figure out something for a certain value. In the example in the slides, we want to find out the square of the value,  $n$ . The function itself, still does matching. It matches the base case up with the base value, where in the example, ends up matching 0 to 0. Then, for the recursive case, it matches the  $n$ th value with the expression where the recursive call is made.

When talking about correctness, in a case like this, that is referring to the questions, did you handle the base case correctly and does the function eventually terminate. These two things are intertwined. If you handled your base case correctly then your function should terminate. Same opposite also will hold. If your function terminates, than most likely, sans some weird error, you handled the base case correctly.

The overall idea of recursion is similar to induction.

### 3.2 Structural Recursion (slides 28-29)

Structural recursion is the idea of taking something apart, performing some operation on those elements and then putting everything back together again. This doesn't work with infinite data structures though, because the recursion will never terminate. In other words, the correctness of this recursion is dependant on your data structure being finite.

Here is an example that computes the length of a list. The case that we have the empty list, then there is no length and we return 0. Otherwise, for the head you add 1 to the length of the tail list. That is, you get to the very tail of the list and then add 1 as you return from the function calls.

```
# let rec length list = match list
  with [] rightarrow 0 (* Nil case *)
       | x :: xs rightarrow 1 + length xs;; (* Cons case *)
val length : 'a list rightarrow int = <fun>
```

### 3.3 Structural Recursion vs. Forward Recursion (slide 30)

What is the difference between these two types of recursion? Well, as stated before, with structural recursion, the input is broken into smaller pieces and then put back together again. With forward recursion, first you run through and make all the recursive calls that you will be making and then you make your final result from the evaluation of the partial results of the when the recursive calls were made. So to remember, when you have forward recursion, the very first thing you do is, you recurse.

In both cases, for everytime you make a recursive call, you are adding a partial function to your call stack. This means you are using up memory space. Now, while this may not matter for functions that are linear, this idea makes a huge difference on exponential functions. So not only will the function take exponential time to run, it will also be taking up exponential space.

### 3.4 Examples of Forward Recursion (slide 31)

We've seen two examples of forward recursion already. We've seen it in `double_up` and `poor_rev`.

```
# let rec double_up list = ...
  | (x :: xs) rightarrow (x :: x :: double_up xs);;
# let rec poor_rev list = ...
  | (x::xs) rightarrow poor_rev xs @ [x];;
```

In both cases, as you can see, the first thing either function has to do is recurse down the list.