CSE 341, Winter 2008, Lecture 7 Summary

Standard Disclaimer: These comments may prove useful, but certainly are not a complete summary of all the important stuff we did in class. They may make little sense if you missed class, but hopefully will help you organize and process what you have learned.

We first briefly complete our course-motivation discussion, and then proceed to considering functions that take and return other functions.

Our course-motivation discussion last time focused on, (1) why study programming languages and (2) why learn to use functional programming languages. To focus a bit more on the actual organization and topics in this course, we can ask why we are learning ML, Scheme, and Ruby. Each has several features that help us learn essential language concepts:

- ML has parametric polymorphism, which is complementary to OO-style subtyping, a rich module system for abstract types, and rich pattern-matching
- Scheme has dynamic type, hygienic macros, fascinating control operators (though we may skip this), and a minimalist design
- Ruby has classes but not types and more complete commitment to OO than Java

Were the course longer we would also investigate Haskell (a pure, lazy functional language), Prolog (a logic language with unification and backtracking), and Smalltalk (much like Ruby, but even more OO and a more elegant language design).

We will also compare and contrast functional and object-oriented programming styles (two major high-level language paradigms), as well as dynamic versus static typing. These are orthogonal issues and after this course you will have seen one language with each combination:

	dynamically typed	statically typed
functional	Scheme	SML
object-oriented	Ruby	Java

The next three lectures will focus on first-class functions and function closures. By "first-class" we mean that functions can be computed and passed wherever other values can in ML. As examples, we can pass them to functions, return them from functions, put them in pairs, have them be part of the data a datatype constructor carries, etc. Function closures refer to functions that use variables defined outside of them, which we will start seeing next lecture. The term *higher-order function* just refers to a function that takes or returns other functions.

Here is a first example of a function that takes another function:

```
fun n_times (f,n,x) =
    if n=0
    then x
    else f (n_times(f,n-1,x))
```

We can tell the argument `f` is a function because the last line calls `f` with an argument. What `n_times` does is compute `f(f(...(f(x))))` where the number of calls to `f` is `n`. That is a geuninely useful helper function to have around. For example, here are 3 different uses of it:

```
fun double x = x+x
val x1 = n_times(double,4,7) (* answer: 112 *)

fun increment x = x+1
val x2 = n_times(increment,4,7) (* answer: 11 *)

val x3 = n_times(tl,2,[4,8,12,16]) (* answer: [12,16] *)
```

Like any helper function `n_times` lets us *abstract* the common parts of multiple computations so we can *reuse* some code in different ways by passing in different arguments. The main novelty is making one of those arguments a function, which is a powerful and flexible programming idiom. It also makes perfect sense — we are not introducing any new language constructs here, just using ones we already know in ways you may not have thought of.

Let us now consider the type of `n_times`, which is `('a -> 'a) * int * 'a -> 'a`. It might be simpler at first to consider the type `(int -> int) * int * int -> int`, which is how `n_times` is used for `x1` and `x2` above: It takes 3 arguemnts, the first of which is itself a function that takes and return an `int`. Similarly, for `x3` we use `n_times` as though it has type `(int list -> int list) * int * int list -> int list`. But choosing either one of these types for `n_times` would make it less useful (some of our examples would not type-check). The type `('a -> 'a) * int * 'a -> 'a` says the third argument and result can be any type, but they have to be the *same* type, as does the argument and return type for the first argument. When types can be any type and do not have to be the same as other types, we use different letters (`'b`, `'c`, etc.)

This is called *paramteric polymorphism*, the ability of functions to take arguments of any type. It is technically a separate issue from first-class functions since there are functions that take functions and do not have polymorphic types and there are functions with polymorphic types that do not take functions. However, many of our examples with first-class functions will have polymorphic types. That is a good thing because it makes our code more reusable. It is quite essential to ML. For example, without paramteric polymorphism, we would have to redefine lists for every type of element that a list might have. Instead, we can have functions that work for any kind of list, like `length`, which has type `'a list -> int`.

In our examples above, we defined `double` and `increment` at the top-level, but we already know we can define functions anywhere. So if we were only going to use `double` as the argument to `n_times`, we could instead do this:

```
n_times(let fun f x = x+x in f end, 4, 7)
```

As always, the let-expression returns the result of its body, which in this case is the function bound to `f`, which takes its argument and doubles it. However, this is poor style because ML has a much more concise way to define functions right where you use them:

```
n_times(fn x => x+x, 4, 7)
```

This code is exactly like our previous version: It defines a function that takes an argument `x` and returns `x+x`. It is called an *anonymous function* because we never gave it a name (like `f`). It is common to use anonymous functions as arguments to other functions.

The only thing you cannot do with an anonymous function is recursion, exactly because you have no name to use for the recursive call. In such cases, you need to use a `fun` binding as before. For non-recursive functions, you could use anonymous functions with `val` bindings instead of a `fun` binding. For example, these two bindings are exactly the same thing:

```
fun increment x = x + 1
val increment = fn x => x+1
```

They both bind `increment` to a value that is a function that returns its argument plus 1.

We now consider a very useful higher-order function over lists:

```
fun map (f,lst) =
    case lst of
        [] => []
      | fst::rest => (f fst)::(map(f,rest))
```

The `map` function (provided by the ML standard library as `List.map`, but the implementation really is the 4 lines above) takes a list and a function `f` and produces a new list by applying `f` to each element of the list. Here are two example uses:

```
val x4 = map (increment, [4,8,12,16]) (* answer: [5,9,13,17] *)
val x5 = map (hd, [[1,2],[3,4],[5,6,7]]) (* answer: [1,3,5] *)
```

The type of `map` is illuminating: `('a -> 'b) * 'a list -> 'b list`. You can pass `map` any kind of list you want, but the argument type of `f` must be the element type of the list (they are both `'a`). But the return type of `f` can be a different type `'b`. The resulting list is a `'b list`. For `x4`, both `'a` and `'b` are *instantiated* with `int`. For `x5`, `'a` is `int list` and `'b` is `int`.

The definition and use of map is an incredibly important idiom even though our particular example is simple. We could have easily written a recursive function over lists of integers that incremented all the elements, but instead we divided the work into two parts: The map implementer knew how to traverse a recursive data structure, in this case a list. The map client knew what to do with the data there, in this case increment each number. You could imagine either of these tasks — traversing a complicated piece of data or doing some calculation for each of the pieces — being vastly more complicated and best done by different developers without making assumptions about the other task. That's exactly what writing `map` as a helper function that takes a function lets us do. Next lecture has more to say about how `map` and one other function are revolutionizing internet-scale computing using exactly this idea.

Here is a second very useful higher-order function for lists. It takes a function of type `'a -> bool` and an `'a list` and returns the `'a list` containing only the elements of the input list for which the function returns true:

```
fun filter (f,lst) =
    case lst of
        [] => []
      | fst::rest => if f fst
                     then fst::(filter (f,rest))
                     else filter (f,rest)
```

Here is an example use that assumes the list elements are pairs with second component of type `int`; it returns the lsit elements where the second component is even:

```
fun get_all_even2 lst = filter((fn (_,v) => v mod 2 = 0), lst)
```

(Notice how we are using a pattern for the argument to our anonymous function.)

Finally, functions can also return functions. Here is an example:

```
fun double_or_triple f =
    if f 7
    then fn x => 2*x
    else fn x => 3*x
```

The type of `double_or_triple` is `(int -> bool) -> (int -> int)`: The if-test makes the type of `f` clear and as usual the two branches of the if must have the same type, in this case `int->int`. However, ML will print the type as `(int -> bool) -> int -> int`, which is the same thing. The parentheses are unnecessary because the `->` "associates to the right", i.e., `t1 -> t2 -> t3 -> t4` is `t1 -> (t2 -> (t3 -> t4))`.