# CSE341, Fall 2011, Lecture 8 Summary

*Standard Disclaimer: This lecture summary is not necessarily a complete substitute for attending class, reading the associated code, etc. It is designed to be a useful resource for students who attended class and are later reviewing the material.*

So far, the functions we have passed to or returned from other functions have been *closed*: the function bodies used only the function's argument(s) and any locally defined variables. But we know that functions can do more than that: they can use any bindings that are in scope. Doing so in combination with higher-order functions is very powerful, so it is crucial to learn effective idioms using this technique. But first it is even more crucial to get the semantics right. This is probably the most subtle and important concept in the entire course, so go slowly and read carefully.

## Lexical Scope

*The body of a function is evaluated in the environment where the function is* ***defined****, not the environment where the function is* ***called****.* Here is a very simple example to demonstrate the difference:

```
val x = 1
fun f y = x + y
val x = 2
val y = 3
val z = f (x+y)
```

In this example, `f` is bound to a function that takes an argument `y`. Its body also looks up `x` in the environment where `f` was defined. Hence this function *always* increments its argument since the environment at the definition maps `x` to 1. Later we have a different environment where `f` maps to this function, `x` maps to 2, `y` maps to 3, and we make the call `f x`. Here is how evaluation proceeds:

- Look up `f` to get the previously described function.
- Evaluate the argument `x+y` in the *current* environment by looking up `x` and `y`, producing 5.
- Call the function with the argument 5, which means evaluating the body `x+y` in the *"old"* environment where `x` maps to 1 extended with `y` mapping to 5. So the result is 6.

Notice the argument was evaluated in the current environment (producing 5), but the function body was evaluated in the "old" environment. We discuss below why this semantics is desirable, but first we define this semantics more precisely and understand the semantics with additional silly examples that use higher-order functions.

This semantics is called *lexical scope*. The alternate, inferior semantics where you use the current environment (which would produce 7 in the above example) is called *dynamic scope*.

## Environments and Closures

We have said that functions are values, but we have not been precise about what that value exactly is. We now explain that a function value has *two parts*, the *code* for the function (obviously) and the *environment that was current when we created the function.* These two parts really do form a "pair" but we put "pair" in quotation marks because it is not an ML pair, just something with two parts. You *cannot* access the parts of the "pair" separately; all you can do is call the function. This call uses both parts because it evaluates the code part using the environment part.

This "pair" is called a *function closure* or just *closure*. The reason is that while the code itself can have *free variables* (variables that are not bound inside the code so they need to be bound by some outer environment), the closure carries with it an environment that provides all these bindings. So the closure overall is "closed" — it has everything it needs to produce a function result given a function argument.

In the example above, the binding `fun f y = x + y` bound `f` to a closure. The code part is the function `fn y => x + y` and the environment part maps `x` to 1. Therefore, any call to this closure will return `y+1`.

**(Silly) Examples Including Higher-Order Functions**

Lexical scope and closures get more interesting when we have higher-order functions, but the semantics already described will lead us to the right answers.

Example 1:

```
val x = 1
fun f y =
    let
        val x = y+1
    in
        fn z => x + y  + z
    end
val x = 3
val g = f 4
val y = 5
val z = g 6
```

As in the first example, `f` is bound to a closure where the environment part maps `x` to 1. So when we later evaluate `f 4`, we evaluate `let val x = y + 1 in fn z => x + y + z end` in the environment `x` maps to 1 extended to map `y` to 4. But then due to the let-binding we shadow `x` so we evaluate `fn z => x + y + z` in an environment where `x` maps to `5` and `y` maps to 4. How do we evaluate a function like `fn z => x + y + z`? We create a closure with the current environment. So `f 4` returns a closure that, when called, will always add 9 to its argument, no matter what the environment is at any call-site. Hence in the last line of the example `z` will be bound to 15.

Example 2:

```
fun f g =
    let
        val x = 3
    in
        g 2
    end
val x = 4
fun h y = x + y
val z = f h
```

In this example `f` is bound to a closure that takes another function `g` as an argument and returns the result of `g 2`. The closure bound to `h` *always* adds 4 to its argument because the argument is `y`, the body is `x+y`, and the function is defined in an environment where `x` maps to 4. So in the last line, `z` will be bound to 6. The binding `val x = 3` is totally irrelevant: the call `g 2` is evaluated by looking up `g` to get the closure that was passed in and then using that closure with *its environment* (in which `x` maps to 4) with 2 for an argument.

**Why Lexical Scope**

While lexical scope and higher-order functions take some getting used to, decades of experience is clear that this semantics is what we want. In the next section of this lecture and all of the next lecture we will see various widespread idioms that are powerful and rely on lexical scope.

But first we can also motivate lexical scope by showing how dynamic scope (where you just have one current environment and use it to evaluate function bodies) leads to some fundamental problems.

First, suppose in Example 1 the body of `f` was changed to `let val q = y+1 in fn z => q + y + z`. Under lexical scope this is fine: we can always change the name of a local variable and its uses without it affecting anything. Under dynamic scope, now the call to `g 6` will make no sense: we will try to look up `q`, but there is no `q` in the environment at the call-site.

Second, consider again the original version of Example 1 but now change the line `val x = 3` to `val x = "hi"`. Under lexical scope, this is again fine: that binding is never actually used. Under dynamic scope, the call to `g 6` will look-up `x`, get a string, and try to add it, which should not happen in a program that type-checks.

Similar issues arise with Example 2: The body of `f` in this example is awful: we have a local binding we never use. Under lexical scope we can remove it, changing the body to `g 2` and know that this has no effect on the rest of the program. Under dynamic scope it would have an effect. Also, under lexical scope we *know* that any use of the closure bound to `h` will add 4 to its argument regardless of how other functions like `g` are implemented and what variable names they use. This is a key separation-of-concerns that only lexical scope provides.

For “regular” variables in programs, lexical scope is the way to go. There are some compelling uses for dynamic scoping for certain idioms, but few languages have special support for these (Racket does) and very few if any modern languages have dynamic scoping as the default. But you have seen one feature that is more like dynamic scope than lexical scope: exception handling. When an exception is raised, evaluation has to “look up” which handle expression should be evaluated. This “look up” is done using the dynamic call stack, with no regard for the lexical structure of the program.

**Passing Closures to Iterators Like Filter**

The examples above are silly, so we need to show useful programs that rely on lexical scope. The first idiom we will show is, like last lecture, passing functions to iterators like *map* and *filter*. The functions we passed last time did not use their environment (only their arguments and maybe local variables), but being able to pass in closures makes the higher-order functions much more widely useful. Consider:

```
fun filter (f,xs) =
    case xs of
        [] => []
      | x::xs’ => if f x then x::(filter(f,xs’)) else filter(f,xs’)

fun allGreaterThanSeven xs = filter (fn x => x > 7, xs)

fun allGreaterThan (xs,n) = filter (fn x => x > n, xs)
```

Here, `allGreaterThanSeven` is “old news” — we pass in a function that removes from the result any numbers 7 or less in a list. But it is much more likely that you want a function like `allGreatherThan` that takes the “limit” as a parameter `n` and uses the function `fn x => x > n`. Notice this requires a closure and lexical scope! When the implementation of `filter` calls this function, we need to look up `n` in the environment where `fn x => x > n` was defined.

Here are two additional examples:

```
fun allShorterThan1 (xs,s) = filter (fn x => String.size x < String.size s, xs)
fun allShorterThan2 (xs,s) =
    let
        val i = String.size s
    in
        filter(fn x => String.size x < i, xs)
    end
```

Both these functions take a list of strings `xs` and a string `s` and return a list containing only the strings in `xs` that are shorter than `s`. And they both use closures, to look up `s` or `i` when the anonymous functions get called. The second one is more complicated but a bit more efficient: The first one recomputes `String.size s` once per element in `xs` (because `filter` calls its function argument this many times and the body evaluates `String.size s` each time). The second one "precomputes" `String.size s` and binds it to a variable `i` available to the function `fn x => String.size x < i`.

**Fold and More Closure Examples**

Beyond map and filter, a third incredibly useful higher-order function is *fold*, which can have several slightly different definitions and is also known by names such as *reduce* and *inject*. Here is one common definition:

```
fun fold (f,acc,l) =
  case l of
    []     => acc
  | hd::tl => fold (f, f(acc,hd), tl)
```

`fold` takes an "initial answer" `acc` and uses `f` to "combine" `acc` and the first element of the list, using this as the new "initial answer" for "folding" over the rest of the list. We can use `fold` to take care of iterating over a list while we provide some function that expresses how to combine elements. For example, to sum the elements in a list `lst`, we can do:

```
fold ((fn (x,y) => x+y), 0, lst)
```

As with `map`, much of `fold`'s power comes from clients passing closures that can have "private fields" (in the form of variables bindings) for keeping data they want to consult. Here are two examples. The first counts how many elements are in some integer range. The second checks if all elements are strings shorter than some other string's length.

```
fun numberInRange (xs,lo,hi) =
  fold ((fn (x,y) =>
           x + (if y >= lo andalso y <= hi then 1 else 0)),
        0, xs)

fun areAllShorter (xs,s) =
    let
        val i = String.size s
    in
        fold((fn (x,y) => x andalso String.size y < i), true, xs)
    end
```

This pattern of splitting the recursive traversal (`fold` or `map`) from the data-processing done on the elements (the closures passed in) is fundamental. In our examples, both parts are so easy we could just do the whole thing together in a few simple lines. More generally, we may have a very complicated set of data structures to traverse or we may have very involved data processing to do. It is good to *separate these concerns* so that the programming problems can be solved separately.