CSE341: Programming Languages

Lecture 8
Lexical Scope and Function Closures

Dan Grossman

Fall 2011

Very important concept

- We know function bodies can use any bindings in scope
- But now that functions can be passed around: In scope where?

Where the function was defined

(not where it was called)

- There are lots of good reasons for this semantics
  - Discussed after explaining what the semantics is
- For HW, exams, and competent programming, you must "get this"
- This semantics is called *lexical scope*

Example

Demonstrates lexical scope even without higher-order functions:

```
(* 1 *) val x = 1
(* 2 *) fun f y = x + y
(* 3 *) val x = 3
(* 4 *) val y = 4
(* 5 *) val z = f (x + y)
```

- Line 2 defines a function that, when called, evaluates body **`x+y`** in environment where **`x`** maps to **`1`** and **`y`** maps to the argument
- Call on line 5:
  - Looks up **`f`** to get the function defined on line 2
  - Evaluates **`x+y`** in current environment, producing **`7`**
  - Calls the function, which evaluates the body in the old environment, producing **`8`**

Closures

How can functions be evaluated in old environments that aren't around anymore?

- The language implementation keeps them around as necessary

Can define the semantics of functions as follows:

- A function value has two parts
  - The code (obviously)
  - The environment that was current when the function was defined
- This is a "pair" but unlike ML pairs, you cannot access the pieces
- All you can do is call this "pair"
- This pair is called a *function closure*
- A call evaluates the code part in the environment part (extended with the function argument)

Example

```
(* 1 *) val x = 1
(* 2 *) fun f y = x + y
(* 3 *) val x = 3
(* 4 *) val y = 4
(* 5 *) val z = f (x + y)
```

- Line 2 creates a closure and binds **`f`** to it:
  - Code: "take **`y`** and have body **`x+y`**"
  - Environment: "**`x`** maps to **`1`**"
    - (Plus whatever else is in scope, including **`f`** for recursion)

So what?

Now you know the rule. Next steps:

- (Silly) examples to demonstrate how the rule works for higher-order functions
- Why the other natural rule, *dynamic scope*, is a bad idea
- Powerful idioms with higher-order functions that use this rule
  - This lecture: Passing functions to iterators like **`filter`**
  - Next lecture: Several more idioms

Example: Returning a function

```
(* 1 *) val x = 1
(* 2 *) fun f y =
(* 2a *)     let val x = y+1
(* 2b *)     in fn z => x+y+z end
(* 3 *) val x = 3
(* 4 *) val g = f 4
(* 5 *) val y = 5
(* 6 *) val z = g 6
```

- Trust the rule: Evaluating line 4 binds to `g` to a closure:
  - Code: “take `z` and have body `x+y+z`”
  - Environment: “`y` maps to `4`, `x` maps to `5` (shadowing), …”
  - So this closure will always add `9` to its argument
- So line 6 binds `15` to `z`

Example: Passing a function

```
(* 1 *) fun f g = (* call arg with 2 *)
(* 1a *)     let val x = 3
(* 1b *)     in g 2 end
(* 2 *) val x = 4
(* 3 *) fun h y = x + y
(* 4 *) val z = f h
```

- Trust the rule: Evaluating line 3 binds **h** to a closure:
  - Code: “take **y** and have body **x+y**”
  - Environment: “**x** maps to **4**, **f** maps to a closure, …”
  - So this closure will always add **4** to its argument
- So line 4 binds **6** to **z**
  - Line 1a is as stupid and irrelevant as it should be

Why lexical scope?

1. Function meaning does not depend on variable names used

Example: Can change body to use **q** instead of **x**

- Lexical scope: it can’t matter
- Dynamic scope: Depends how result is used

```
fun f y =
    let val x = y+1
    in fn z => x+y+z end
```

Example: Can remove unused variables

- Dynamic scope: But maybe some **g** uses it (weird)

```
fun f g =
    let val x = 3
    in g 2 end
```

Why lexical scope?

2. Functions can be type-checked & reasoned about where defined

Example: Dynamic scope tries to add a string and an unbound variable to **6**

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

Why lexical scope?

3. Closures can easily store the data they need
   - Many more examples and idioms to come

```
fun greaterThanX x = fn y => y > x

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

fun noNegatives xs = filter(greaterThanX ~1, xs)
```

Does dynamic scope exist?

- Lexical scope for variables is definitely the right default
  - Very common across languages

- Dynamic scope is occasionally convenient in some situations
  - So some languages (e.g., Racket) have special ways to do it
  - But most don't bother

- If you squint some, exception handling is more like dynamic scope:
  - **`raise e`** transfers control to the current innermost handler
  - Does not have to be syntactically inside a handle expression (and usually isn't)

Recomputation

These both work and rely on using variables in the environment

```
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
```

The first one computes **`String.size`** once per element of **`xs`**

The second one computes **`String.size s`** once per list

- Nothing new here: let-bindings are evaluated when encountered and function bodies evaluated when *called*

Iterators made better

- Functions like **`map`** and **`filter`** are *much* more powerful thanks to closures and lexical scope

- Function passed in can use any "private" data in its environment

- Iterator "doesn't even know the data is there" or what type it has

Another famous function: Fold

`fold` (and synonyms / close relatives reduce, inject, etc.) is another very famous iterator over recursive structures

Accumulates an answer by repeatedly applying **`f`** to answer so far

- **`fold(f,acc,[x1,x2,x3,x4])`** computes **`f(f(f(f(acc,x1),x2),x3),x4)`**

```
fun fold (f,acc,xs) =
   case xs of
     []     => acc
   | x::xs => fold(f, f(acc,x), xs)
```

- This version "folds left"; another version "folds right"
- Whether the direction matters depends on **`f`** (often not)

```
val fold = fn : ('a * 'b -> 'a) * 'a * 'b list -> 'a
```

Examples with fold

These are useful and do not use “private data”

```
fun f1 xs = fold((fn (x,y) => x+y), 0, xs)
fun f2 xs = fold((fn (x,y) => x andalso y>=0),
                 true, xs)
```

These are useful and do use “private data”

```
fun f3 (xs,hi,lo) =
    fold(fn (x,y) =>
            x + (if y >= lo andalso y <= hi
                 then 1
                 else 0)),
          0, xs)
fun f4 (g,xs) = fold(fn (x,y) => x andalso g y),
                     true, xs)
```

Why iterators again?

- These “iterator-like” functions are not built into the language
  - Just a programming pattern
  - Though many languages have built-in support, which often allows stopping early without using exceptions

- This pattern separates recursive traversal from data processing
  - Can reuse same traversal for different data processing
  - Can reuse same data processing for different data structures