

CSE 341: Programming Languages

Dan Grossman

Spring 2004

Lecture 8+ — Function Closures

Today

- Continue examples of functions taking and returning other functions
- Discuss *free variables* in function bodies
- In general, discuss environments and lexical scope
- See key idioms using first-class functions

If you remember one thing...

We evaluate expressions in an environment.

To evaluate function application $(e1\ e2)$, we:

1. evaluate $e1$ to a value that is a function $(fn\ x\ =>\ e3)$
2. evaluate $e2$ to a value v
3. evaluate $e3$ in the environment in which $fn\ x\ =>\ e3$ was *defined*, extended to map x to v

There are *lots* of good reasons for this semantics.

To do homework 3, pass the midterm, and be a competent programmer, you must understand this.

Example 1

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

Example 2

```
val x = 1
```

```
fun f y = let val x = 2 in fn z => x + y + z end
```

```
val x = 3
```

```
val g = f 4
```

```
val y = 5
```

```
g 6
```

Example 3

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

```
val x = 4
```

```
fun h y = x + y
```

```
f h
```

Scope

A key language concept: how are user-defined things *resolved*?

We have seen that ML has *lexically scoped* variables?

Another (more-antiquated-for-variables, sometimes-useful) approach is *dynamic scope*?

Example of dynamic scope: Exception handlers (where does `raise` transfer control?)

The more restrictive “no free variables” makes important idioms impossible.

Why lexical scope?

1. Functions can be reasoned about (defined, type-checked, etc.) where defined
2. Function meaning not related to choice of variable names
3. “Closing over” local variables creates private data; function definer *knows* function users do not depend on it

Key idioms with closures

- Create similar functions
- Pass functions with private data to iterators (*map*, *fold*, ...)
- Combine functions
- Provide an ADT (see section)
- As a *callback* without the “wrong side” specifying the environment.
- Partially apply functions (“currying”)

Create similar functions

```
val addn = fn n => fn m => n+m
val increment = addn 1
val add_two = addn 2
fun f n =
  if n=0
  then []
  else (addn n)::(f (n-1))
```

Private data, for map/fold

On Monday we saw map, this fold function is even more useful:

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

Example uses (without using private data):

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

Example use (with private data):

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

More on fold and private data

Another more general example:

```
fun f4 (l,g) = fold ((fn (l2,y) => (g y)::l2), [], l)
```

A fold function over a data structure is much like a *visitor pattern* in OOP.

We define fold once and do not restrict the type of the function passed to fold or the environment in which it is defined.

In general, libraries should not unnecessarily restrict clients.

Combine functions

```
fun f1 (g,h) = fn x => g (h x)
```

```
fun f2 (g,h) = fn x =>
```

```
    case g x of NONE => h x | SOME y => y
```