COMPUTER SCIENCE 51

Spring 2009

cs51.seas.harvard.edu

Prof. Greg Morrisett
Prof. Ramin Zabih

Today

- Higher-order functions
  - lambda expressions
  - passing lambdas to functions
  - returning new lambda from functions
- Encoding features with lambda
  - let declarations
  - contracts
  - pretty-printers

Lambda Expressions

When you write a function defn:

```
(define (inc x) (+ 1 x))
```

This is short-hand for:

```
(define inc (lambda (x) (+ 1 x))
```

To save space, I will write:

```
(define inc (λ (x) (+ 1 x))
```

Lambda's

- Lambda expressions are anonymous functions: `(λ (x) (* x x))`
  - The square function.
  - It doesn't need a name any more than we need to define 3 to be `three` or `z`.
- Lambda expressions are values, just like numbers, strings, etc.
  - we can pass a lambda to a function.
  - we can return a lambda from a function.
  - we can place a lambda in a list.
- Let's see why this is useful...

Factoring

Consider the following two functions:

```
(define (sum x)
  (if (empty? x) 0
      (+ (car x) (sum (cdr x)))))

(define (prod x)
  (if (empty? x) 1
      (* (car x) (prod (cdr x)))))
```

Factoring

```
(define (foldr f u x)
  (if (empty? x) u
      (f (car x)
         (foldr f u (cdr x)))))

(define (sum x) (foldr + 0 x))
(define (prod x) (foldr * 1 x))
(define (flatten x) (foldr append empty x))
(define (mystery x) (foldr cons empty x))
```

How to think of foldr

- A list value looks like this:
```
(cons v1
  (cons v2 (...
    (cons vn empty)...)))
```
- A call to foldr with function f and value u replaces all of the cons's with f and empty with u:
```
(f v1
  (f v2 (...
   (f vn u)...)))
```

For Example:

```
(foldr + 0 (list 1 2 3 4)) =
(foldr + 0 (cons 1
              (cons 2
               (cons 3
                (cons 4 empty)…)=
(+ 1
  (+ 2
    (+ 3
      (+ 4 0)))) = 10
```

Another Example

```
(foldr * 1 (list 1 2 3 4)) =
(foldr * 1 (cons 1
              (cons 2
               (cons 3
                (cons 4 empty)…)=
(* 1
  (* 2
    (* 3
      (* 4 1)))) = 24
```

A Slightly Different fold

```
(define (foldl f u x)
  (if (empty? x) u
      (foldl f (f (car x) u) (cdr x))))

(define (sum2 x) (foldl + 0 x))
(define (prod2 x) (foldl * 1 x))
(define (append2 x) (foldl append empty x))
```

How to think of foldl

- Given a list `x`:
```
(cons v1
  (cons v2 (...
    (cons vn empty)...)))
```
- A call to `foldl` with `f` and `u` on `x` yields:
```
(f vn ... (f v2 (f v1 u))...)
```

Computing Lengths

We can compute the length of a list using foldr (or foldl):

```
(define (inc h x) (+ 1 x))
(define (length x) (foldr inc 0 x))
```

But it's a pain to have to define inc just so we can call foldr. This is where **lambda** comes in!

```
(define (length x)
  (foldr (λ (h x) (+ 1 x)) 0 x))
```

Scope

- There are two distinct `x`'s in this defn:

```
(define (length x)
  (foldr (λ (h x) (+ 1 x)) 0 x))
```

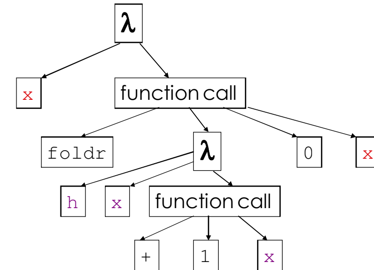
- Recall that this is short-hand for:

```
(define length
  (λ (x)
    (foldr (λ (h x) (+ 1 x)) 0 x))
```

- General rule for *static* scope:
  - an occurrence of a variable corresponds to the nearest enclosing lambda for that variable.

Expressions as Trees

```
(λ (x) (foldr (λ (x) (+ 1 x)) 0 x)
```



Principle of Re-Naming

We should always be able to re-name a lambda-bound variable as long as we re-name all of the corresponding free occurrences.

```
(define length
  (λ (x)
    (foldr (λ (h x) (+ 1 x)) 0 x))

(define length
  (λ (z)
   (foldr (λ (h q) (+ 1 q)) 0 z))
```

Why Important?

```
(define length
  (λ (x)
    (foldr (λ (h x) (+ 1 x)) 0 x))
```

- The behavior of a function shouldn't depend upon what variable names we pick.
  - information hiding, just like in last lecture!
  - basic principle of composition
- Suppose it did:
  - suppose we picked another rule for resolving which lambda a variable corresponds to.
  - then my code might conflict with your code.
  - your code might change the behavior of mine.

Summary So Far

- In Scheme, functions are values.
  - specifically, lambda-expressions
  - a.k.a. anonymous functions.
  - they are 1st class citizens, just like numbers.
- `(define (f x y z) e)` is short-hand for `(define f (λ (x y z) e)).`
  - so we bind a lambda to a name, just as we might bind 3.1415 to the name pi.
- Lambdas are *statically* scoped.
  - a variable corresponds to the nearest enclosing lambda.
  - supports principle of re-naming.

More Factoring

```
(define (square-all x)
  (if (empty? x) empty
      (cons (square (car x))
            (square-all (cdr x)))))

(define (inc-all x)
  (if (empty? x) empty
      (cons (inc (car x))
            (inc-all (cdr x)))))
```

The Map function

```
(define (map g x)
  (if (empty? x) empty
        (cons (g (car x))
              (map g (cdr x)))))

(define (square-all x)
  (map (λ (y) (* y y)) x))

(define (inc-all x)
  (map (λ (y) (+ 1 y)) x))
```

Map in terms of Foldr

```
(define (foldr f u x)
  (if (empty? x) u
      (f (car x) (foldr f u (cdr x)))))

(define (map g x)
  (foldr (λ (h t) (cons (g h) t)) empty x))

(define (map g x)
  (if (empty? x) empty
      ((λ (h t) (cons (g h) t)) (car x)
        (foldr (λ (h t) (cons (g h) t))
          empty (cdr x)))))

(define (map g x)
  (if (empty? x) empty
      (cons (g (car x) (map g (cdr x)))))
```

Filter

```
(define (filter p? x)
  (cond [(empty? x) x]
        [(p? (car x))
          (cons (car x)
             (filter p? (cdr x)))]
        [else (filter p? (cdr x))]))

(filter even? (list 1 2 3 4)) ⇓
  (list 2 4)
```

Filter in terms of Foldr?

```
(define (foldr f u x)
  (if (empty? x) u
      (f (car x) (foldr f u (cdr x)))))

(define (filter p? x)
  (foldr (λ (h t) (if (p? h) (cons h t) t))
   empty x))
```

Filter-and-Square?

- Suppose I want to filter out the negative numbers and square them:

```
(define (pos-square x)
  (map square (filter positive? x)))
```

- Now suppose I want to do it in one pass:

```
(define (pos-square x)
  (foldr (λ (h t)
        (if (positive? h) (cons (square h) t) t)
    empty x))
```

Fusion

We can always reduce a foldr-filter or foldr-map into a single pass as long as we don't have *side-effects*.

```
(foldr f u (map g x)) =
  (foldr (λ (h t) (f (g h) t)) u)

(foldr f u (filter p? x)) =
  (foldr (λ (h t)
           (if (p? h) (f h t) t) u)
```

Who Cares?

- Process lots of web-pages to extract information.
  - programmer writes code using a combination of maps, filters, and reduces
    - e.g., build a dictionary that calculates the total number of links to each web-page.
  - compiler optimizes into as few passes as possible.
    - these data sets are HUGE.
    - additionally map, filter, and certain foldr's can be parallelized to run on many different machines.
- http://labs.google.com/papers/mapreduce.html

So Far

- We've only considered passing functions to other functions.
- What about returning functions?

```
(define (compose f g)
  (λ (x) (f (g x))))
```

Consider...

```
(define (compose f g)
  (λ (x) (f (g x))))

(define inc-n-square
  (compose inc square))

(define inc-n-square
  (λ (x) (inc (square x)))

(define (inc-n-square x)
  (inc (square x)))
```

What Does This Do?

```
(define (id x) x)
(define (mystery x) (foldr compose id x)
(mystery (list inc square dec)) =

(foldr compose id
  (cons inc
    (cons square
      (cons dec empty)))) =

(compose inc
  (compose square
    (compose dec id))) =

(λ (x) (inc (square (dec x)))
```

Remember Contracts?

- Suppose PLT didn't provide us with contracts as a built-in mechanism.
- Remember how much boiler-plate code we had to write to test arguments and results for function?
- Perhaps we can use lambdas to factor this boiler-plate out...

Recall Naive Sets:

```
(define (insert x s)
  (if (and (number? x) (set? s))
      (make-set (cons x (set-elements s)))
      (error 'insert
        "expecting a number and a set")))

(define (union s1 s2)
  (if (and (set? s1) (set? s2))
      (make-set (append (set-elements s1)
                        (set-elements s2))
      (error 'union "expecting two sets")))
```

Simple Contract Factoring

```
(define (check p1? p2? f)
  (λ (x1 x2) (if (and (p1? x) (p2? x)) (f x1 x2)
                 (error 'check "bad arguments"))))

(define insert
  (check number? set?
    (λ (x s) (make-set (cons x
                        (set-elements s))))))

(define union
  (check set? set?
    (λ (s1 s2) (make-set (append (set-elements s1)
                               (set-elements s2))))))
```

Another Trick

We can define `let` in terms of `lambda`.

```
(let ([x1 e1]
      [x2 e2]
       ...
      [xn en])
 e)
=
((λ (x1 x2 ... xn) e) e1 e2 .. en)
```

Example

```
(let ([x (square 3)]
      [y (inc 4)])
 (+ x y))
=
((λ (x y) (+ x y)) (square 3) (inc 4))
=
((λ (x y) (+ x y)) 9 5)
=
(+ 9 5)
=
14
```

Multi-Argument Functions

We can also encode multi-argument functions using 1-argument functions and lambdas:

```
(λ (x1 x2 ... xn) e) ≈
 (λ (x1) (λ (x2) ... (λ (xn) e) ...))

(f e1 e2 ... en) ≈
  ((...((f e1) e2)...) en)
```

Example

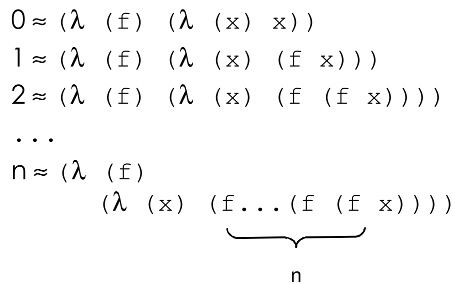
```
(λ (x y) (+ x y)) ≈
  (λ (x) (λ (y) (+ x y))

((λ (x y) (+ x y)) 9 5) ≈
(((λ (x) (λ (y) (+ x y)) 9) 5) =
((λ (y) (+ 9 y) 5) =
(+ 9 5) =
14
```

In fact...

- You can encode a Turing-complete language with just this little subset of Scheme:
  - one-argument lambdas
  - variables
  - function calls

Numbers as Lambdas



Addition

```
inc := (λ (n)
         (λ (f) (λ (x)
           (f ((n f) x)))))

add := (λ (n) (λ (m)
               ((m inc) n)))
```

Booleans, If, Ifzero

true ≈ `(λ (t) (λ (f) t))`
false ≈ `(λ (t) (λ (f) f))`

(if e1 e2 e3) ≈ `((e1 e2) e3)`

```
ifzero :=
  (λ (n)
   ((n (λ (x) false)) true))
```

Lists

empty ≈ `(λ (f) (λ (x) x))`
cons ≈ `(λ (h)`
```
        (λ (t)
         (λ (f)
          (λ (x) ((f h)
                   ((t f) x)))))))
```
(foldr f u x) ≈ `((x f) u)`

Summary

- Functions are 1st-class, statically-scoped.
  - can pass to other functions, return them, place them in data structures, etc.
  - crucial for factoring out patterns in code.
- Some key higher-order list operations:
  - foldr, foldl, map, filter
  - can be defined in terms of foldr
  - fusion lets us reduce the number of passes
- Many linguistic features can be coded using **λ**.
  - contracts, let, multi-argument functions
  - natural numbers, booleans, lists, ...