

Announcements

- **Get started on project 1**
 - ask for help when you need it
- **Look at the scheme chapter in the Sethi book in SERC**
- **Homework 3 due Wednesday at start of class**

1

Scheme

- **let and let***
- **closures**
- **currying**
- **tail-recursion**
- **foldl**

2

Binding Local Variables

In scheme, how to achieve something like this:

```
x = (f a)
y = (g b)
<expression involving x and y>
```

A solution using lambda expressions:

```
((lambda (x y)
  <expression involving x and y>)
 (f a)
 (g b))
```

3

Let and Let*

```
(let ((x (f a))
      (y (g b)))
  <expression involving x and y > )
```

- (f a) and (g b) evaluated and bound to variables *in parallel*

```
(let* ((x (f a))
       (y (g x)))
  <expression involving x and y > )
```

- (f a) and (g x) evaluated and bound to variables *in order*, as if the lets were nested inside each other

4

Let and Let*

```
(let ((f (lambda (x) (+ x x)))
      (y 3))
      (f y))
→ 6
```

```
(let* ((f (lambda (x) (+ x x)))
       (g (lambda (x) (* 2 (f x))))
      (y 3))
      (g y))
→ 12
```

5

Closures

- A result of having lexical scoping...
- A closure is a function value plus the environment in which it is to be evaluated
 - Sometimes need to include variables not local to the function so it can be evaluated later
- A closure can be used as an ordinary function
 - Applied to arguments
 - Passed as an argument
 - Returned as a value

6

Closure example

- Consider the following:
 - > (define (make-adder k)
 (lambda (x) (+ k x)))
 - > (define add3 (make-adder 3))
 - > (add3 5) → 8
- k is a “free” variable in (lambda (x) (+ k x))
- Scheme defines add3 using a closure, and keeps track of the value of k:
 - (lambda (x) (+ k x)) & { k → 3 }

7

Evaluation of Closures

```
(define (gg z)
  (let* ((x 2)
        (f (lambda(y) (+ x y))))
    (map f z)))
```

- gg is a closure which is (lambda (z) (map f z)) where
 - the defining environment is { x → 2; f → (lambda (y) (+ x y)) }
 - we need this environment to evaluate gg
- > (gg '(1 2 3))
1. value of gg is its closure
 2. closure is expanded by argument association with parameter z
 { x → 2; f → (lambda (y) (+ x y)); z → '(1 2 3) }
 3. evaluation occurs and (3 4 5) is returned

8

Currying

- **A function that adds two numbers x and y :**
 - `(define add (lambda (x y) (+ x y)))`
- **... applied to two values:**
 - `(add 3 5) → 8`
- **A function that, when given x , returns a function which adds x to another number y :**
 - `(define curried-add (lambda (x) (lambda (y) (+ x y))))`
- **... applied to two values, in two applications!**
 - `((curried-add 3) 5)`

9

Currying and closures

- **It all happens with closures: scheme keeps track of the curried variables' bindings**
- **No need to stop at two arguments...**

```
> (define curried-add3
  (lambda (x)
    (lambda (y)
      (lambda (z) (+ x y z)))))
> (((curried-add3 2) 3) 7) → 12
```
- **Also can “stop” evaluation part way through:**

```
> (define addstuff ((curried-add3 2) 3))
> (addstuff 7) → 12
```

10

Currying

- **What's going on?**
 - We are reducing n-ary functions to n applications of unary functions
 - Can always do this, so n-ary functions don't add more power to your language

```
+ : R x R → R      curried+ : R → (R → R)
> (define (curried-add x) (lambda (y) (+ x y)))
> ((curried-add 2) 3) → 5
> (let ((f (curried-add 1)))
      (f 4)) → 5
```

11

Tail Recursion

```
(define (number-list? l)
  (cond ((null? l) #t)
        ((not (number? (car l))) #f)
        (else (number-list? (cdr l)))))
```

Notice how the recursive call is the return value without any further computation

This is good, because:

- The compiler can optimize the call by removing the explicit recursion (so it's as efficient as a loop would be)
- The complexity of the algorithm can sometimes be improved by writing the function in a tail-recursive form.

12

Accumulators

Sometimes, a function can be written in tail-recursive form by using accumulators

- an extra argument holds intermediate results
- main function calls a “helper function” with initial value for this argument (the base case)

Examples:

- Two versions of length
- Two versions of reverse: “naive” reverse is $O(n^2)$, but reverse with an accumulator is $O(n)$

13

Length, revisited

- The old version...

```
(define (len l)
  (if (null? l)
      0
      (+ 1 (len (cdr l)))))
```

- Tail recursive version:

```
(define (len-tr l so-far)
  (if (null? l)
      so-far
      (len-tr (cdr l) (+ 1 so-far))))

(define (len2 l) (len-tr l 0))
```

14

Reverse

- **Built-in:** `(reverse '(1 2 3))` → `(3 2 1)`

- **Naive version: $O(n^2)$** —since `append` is $O(n)$

```
(define (reverse1 l)
  (if (null? l)
      ()
      (append (reverse1 (cdr l)) (list (car l)))))
```

- **Tail recursive version: $O(n)$** —since `cons` is $O(1)$

```
(define (reverse-tl l revl)
  (if (null? l)
      ()
      (reverse-tl (cdr l) (cons (car l) revl))))
(define (reverse2 l) (reverse-tl l ()))
```

15

foldl

- `(foldl <operation> <list> <accum>)`

- same argument structure as `foldr`

- binary operation is left associative

- **Compare:**

- `(foldr + '(1 2 3) 0)` → `(+ 1 (+ 2 (+ 3 0)))` → 6

- `(foldl + '(1 2 3) 0)` → `(+ (+ (+ 0 1) 2) 3)` → 6

- **Results can differ if operation not associative**

- `(foldr - '(1 2 3) 0)` → `(- 1 (- 2 (- 3 0)))` → 2

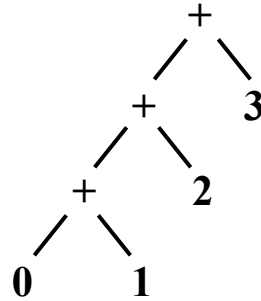
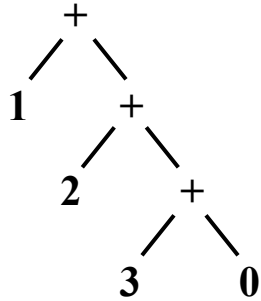
- `(foldl - '(1 2 3) 0)` → `(- (- (- 0 1) 2) 3)` → -6

16

foldr vs. foldl

`(foldr + '(1 2 3) 0) →`
`(+ 1 (+ 2 (+ 3 0))) → 6`

`(foldl + '(1 2 3) 0) →`
`(+ (+ (+ 0 1) 2) 3) → 6`



17

foldr vs. foldl

- **Definitions:**

```
(define (foldr f l base)
  (if (null? l)
      base
      (f (car l) (foldr f (cdr l) base))))
```

```
(define (foldl f l accum)
  (if (null? l)
      accum
      (foldl f (cdr l) (f accum (car l)))))
```

tail recursive,
so more efficient than foldr

18

Reverse using foldl and foldr

- **Code for reverse2 and foldl are similar**

```
(define (reverse2 l)
  (foldl (lambda (x y) (cons y x))
        l ()))
```

- **Can be defined using foldr, but is inefficient**

```
(define (reverse1 l)
  (foldr (lambda (x y) (append y (list x)))
        l ()))
```