Functional Programming - 2

- Higher Order Functions
  - Map on a list
  - Apply
  - Reductions: foldr, foldl
  - Lexical scoping with *let*'s

Higher Order Functions

- Functions as 1st class values
- Functions as arguments

```
(define (f  g  x) (g  x))
  (f   number?   0) yields #t
  (f   len  ‘(1 (2 3)) ) yields 2
  (f   (lambda (x) (* 2  x))  3) yields 6
```

- Functions as return values

```
(define incr (lambda (n) (+ 1 n)) )
  (incr  1) returns 2,
  incr returns #procedure:incr
```

Built-in function *map*

- Higher order function used to apply another function to every element of a list
- Takes 2 arguments: a function f and a list ys and builds a new list by applying the function to every element of the (argument) list

```
(define (map f  ys)
               (if (null? ys)  ‘( )
               (cons (f (car ys)) (map f (cdr ys))))))
```

Built-in function map

```
(define (map f  ys) (if (null? ys)  ‘( )
               (cons (f (car ys)) (map f (cdr ys))))))
```

(map incr ‘(1 2 3 4)) returns (2 3 4 5)

(map incr ‘(-1 0 1)) returns (0 1 2)

(map (lambda (x) (* 2 x)) ‘(1 2 3)) returns (2 4 6)

Possible to define a new map function map2 that takes n-ary functions and applies them to n lists, creating a new list

(map2 + ‘(1 2 3) ‘(4 5 6)) returns (5 7 9)

How map works?

```
(define (map f  ys) (if (null? ys) ‘( )
                        (cons (f (car ys)) (map f (cdr ys))))))
```

TRACE of execution:

```
(map abs ‘( -1  2  -3)
   (cons (abs  -1) (map  abs (2 -3)))
               (cons (abs  2) (map abs (-3)))
                              (cons (abs -3) (map abs ‘())
                                              ‘()
                              (3)
               (2 3)
   (1 2 3)
(list 1 2 3)
```

Try stepping through the mapp definition in DrRacket.

Using map

Define atomcnt3 which uses map to calculate the number of atoms in a list. atomcnt3 creates a list of the count of atoms in every sublist and apply of + calculates the sublist sum.

```
(define (atomcnt3 s) (cond ((atom? s) 1)
                           (else (apply   + (map atomcnt3 s)))))
```

```
(atomcnt3  ‘(1 2 3)) returns 3
(atomcnt3 ‘((a b) d)) returns 3
(atomcnt3 ‘(1 ((2) 3) (((3) (2) 1)))) returns 6
```

How does this function work?

apply

apply is a built-in function whose first argument f is a function and whose second argument ys is a list of arguments for that function

evaluation of apply applies f to ys

(apply + ‘(1 2 3)) returns 6

(apply zero? ‘(2)) returns #false

(apply zero? ‘(0)) returns #true

(apply (lambda (n) (+ 1 n)) ‘(3)) returns 4

The power of apply is that it lets your program build an S-expression to evaluate during execution, and then lets it be evaluated.

---

foldr

- Higher order function that takes a binary, associative operation and uses it to “roll-up” a list

```
(define (foldr op ys id)
     (if (null? ys)  id
             (op (car ys) (foldr op (cdr ys) id))  ))
```

(foldr + ‘(10 20 30) 0) yields

(+ 10 (foldr + (20 30) 0))

(+ 10 (+ 20 (foldr + (30) 0)))

(+ 10 (+ 20 (+ 30 (foldr + () 0))))

(+ 10 (+ 20 (+ 30 0))) yields 60

Think of inserting the op where the cons constructor is placed to build the list.

The Power of Higher Order Functions

- Can compose higher order functions to form compact powerful functions

```
(define (sum   f   ys) (foldr  + (map  f ys) 0))
```

- sum takes a function f and a list ys
- sum applies f to each element of the list and then sums the results

```
(sum (lambda (x) (* 2 x))  ‘(1 2 3)) yields 12
(sum square  ‘(2 3)) yields 13
```

Using foldr

```
(foldr append  ‘((1 2) (3 4))  ‘( ) ) yields
   (app (list 1 2) (foldr append  ‘((3 4))  ‘( ) ) )
                  (app (list 3 4) (foldr append  ‘( )  ‘( ) ))
                         ‘( )
                   (list 3 4)
   (list 1 2 3 4)
```

Try this out using the stepper in DrRacket and watch how foldr works

➢ (list 1 2 3 4)

Defining len (list length function) from foldr.

```
(define (len z) (foldr (lambda (x  y) (+ 1 y))  z   0))
```

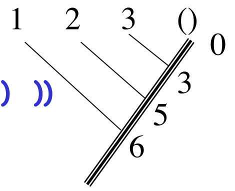
Informal Trace of len

```
(len '(5 6 7)) is
(foldr (lambda (x  y) (+ 1 y))  '(5 6 7)   0))
   ( (lambda (x y) (+ 1 y)) 5 (foldr (lambda (x y) (+ 1 y)) '(6 7) 0) )
                         ( (lambda...) 6 (foldr (lamb...) '(7) 0) )
                                  ( (lamb.. 7 (foldr (lamb...) '( ) 0) )
                                                   0
                                  ( (lambda (x y) (+ 1 y)) 7 0) yields 1
                         ((lambda (x y) (+ 1 y)) 6 1) yields 2
   ( (lambda (x y) (+ 1 y)) 5 2) yields 3
3
```

Fold operations

- Operations that combine elements of an S-expr in an ordered manner
- **foldr** - right associative
  - (foldr + '(1 2 3) 0) can see computation tree in which partial sums are calculated in order down the right branch

```
(define (foldr op ys id)
        (if (null? ys)  id
                (op (car ys) (foldr op (cdr ys) id))  ))
```

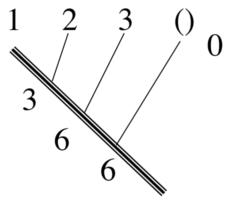


Fold operations

- foldl - left associative, more efficient than foldr
  - (foldl + ‘(1 2 3) 0) can see computation tree in which partial sums are calculated in order down the left branch
  - Note foldl uses less storage than foldr, because doesn’t need to keep values in the recursive copies;

  Instead it accumulates sum as it recurses downward

```
(define (foldl g ys u)
          (if (null? ys) u  (foldl g (cdr ys) (g u (car ys)))))
```

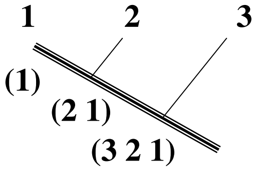


Using foldl

```
(define (rev xs)
       (foldl (lambda (x y) (cons y x)) xs ‘()))
```

then (rev ‘(1 2 3)) will result in the following:

order of execution:



```
(cons 1 ‘())
(cons 2 ‘(1))
(cons 3 ‘(2 1))
```

```
(define (foldl g ys u) (if (null? ys)
                            u
                            (foldl g (cdr ys) (g u (car ys)))))
```

Comparison of Fold Functions

```
(define (foldr op ys id)
       (if (null? ys)  id
              (op (car ys) (foldr op (cdr ys) id))  ))
(define (foldl g ys u)
            (if (null? ys) u  (foldl g (cdr ys) (g u (car ys)))))
```

- Compare underlined portions of these 2 functions
  - Can see that foldl returns the value obtained from a recursive call to itself!
  - Foldr contains a recursive call, but it is not the entire return value of the function

---

Let expressions

Let-expr ::= (let (Binding-list) S-expr1)
Let*-expr ::= (let* (Binding-list) S-expr)
Binding-list ::= (Var S-expr) { (Var S-expr) }

- Let and Let* expressions define a binding between each Var and the S-expr value, which holds during execution of S-expr1
- Let evaluates the S-exprs in parallel (no order specified); Let* evaluates them from left to right.
- Both used to associate temporary values with variables for a local computation
- Variables declared in let's follow lexical scoping rules

Let Examples

(let ((x 2)) (* x x)) yields 4

(let ((x 2)) (let ((y 1)) (+ x y))) yields 3

(let ((x 10) (y (* 2 x))) (* x y)) is an error because all exprs evaluated in parallel and simultaneously bound to the vars

(let* ((x 10) (y (* 2 x))) (* x y)) yields 200

Let Examples

(let ((x 10)) ; causes x to be bound to 10

(let ((f (lambda (a) (+ a x)))) ;causes f to bound to the lambda expr

(let ((x 2)) (f 5))))

Evaluation yields (+ 5 10) = 15, NOT (+ 5 2) = 7

In dynamic scoping the answer would be 7!

(define (f z) (let* ((x 5) (f (lambda (z) (* x z)))) (map f z)))

What does this function do?