Functional Programming - 2

- **Higher Order Functions**
  - **E.g., map, reduce**
  - **apply and eval**
- **Lexical scoping with *let*'s**
- **Closures**
- **Currying**

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 #<closure () (lambda (n) (+ 1  n )>
```

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))))))
```

(map abs '(-1 2 -3 -4)) returns (1 2 3 4)

(map (lambda (x) (+ 1 x)) '(1 2 3)) returns (2 3 4)

(map + '(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)
```

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 ll is a list of arguments for that function

evaluation of apply applies f to ll

(apply + '(1 2 3)) returns 6
(apply zero? 2) returns #f
(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.

eval

eval takes an S-expression and evaluates it (as though it was a program)

```
(define (atomcnt2 s)
    (cond ((null? s) 0)
          ((atom? s) 1)
          (else (eval (cons '+ (map atomcnt2 s))))))
```

Note similarity in usage of apply and eval

reduce

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

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

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

The Power of Higher Order

- **Can compose higher order functions to form compact powerful functions**

(define (sum f ys) (reduce + (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 reduce

(reduce app '((1 2) (3 4)) '( )) yields

(app '(1 2) (reduce app '((3 4)) '( )))

(app '(3 4) (reduce app '( ) '( )))

'( )

(3 4)

(1 2 3 4)

> (reduce append '((1 2) (3 4)) '()) *trace on remus of this evaluation*

"CALLED" reduce #[proc] ((1 ...) (3 4)) ()

"CALLED" reduce #[proc] ((3 4)) ()

"CALLED" reduce #[proc] () ()

"RETURNED" reduce ()

"RETURNED" reduce (3 4)

"RETURNED" reduce (1 2 3 4)

;Evaluation took 0 mSec (0 in gc) 1513 cells work, 103 bytes other

(1 2 3 4)

Using reduce

Defining len (list length function) from reduce.

(define (len z) (reduce (lambda (x y) (+ 1 y)) z 0))

```
> (trace len)
> (trace reduce)
>(len '(1 2 3))
"CALLED" len (1 2 3)
 "CALLED" reduce #[proc] (1 2 3) 0
  "CALLED" reduce #[proc] (2 3) 0
   "CALLED" reduce #[proc] (3) 0
    "CALLED" reduce #[proc] () 0
    "RETURNED" reduce 0
   "RETURNED" reduce 1
  "RETURNED" reduce 2
 "RETURNED" reduce 3
"RETURNED" len 3
;Evaluation took 10 mSec (0 in gc) 2002 cells work, 137 bytes other
3
```


Trace of len

(len '(1 2 3)) is

(reduce (lambda (x y) (+ 1 y)) '(1 2 3) 0))

((lambda (x y) (+ 1 y)) 1 (reduce (lambda (x y) (+ 1 y)) '(2 3) 0))

((lambda…) 2 (reduce (lamb…) '(3) 0))

((lamb.. 3) (reduce (lamb…) '( ) 0))

0

"((lambda (x y) (+ 1 y)) 3 0)" yields 1

((lambda (x y) (+ 1 y)) 2 1) yields 2

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

3

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-expr's in parallel; Let* evaluates them from left to right.**
- **Both used to associate temporary values with variables for a local computation**
- **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  ((x  10)) ;causes x to be bound to 10
        (let   ((f (lambda (a) (+ a  x)))  ;causes f to bound to 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?

Closures

- 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 closure can eventually be evaluated
- A closure can be used as a function
  - Applied to arguments
  - Passed as an argument
  - Returned as a value

Evaluation of Closures

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

gg is actually 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.

```
>(square 2)
4 ; is assumed to be evaluated in the context of the empty environment {}
>(gg  '(1 2 3 ))
```

1. value of gg is its closure
2. closure environment is expanded by argument association with parameter
   { x → 2; f → (lambda (y) (+ x y)); z → '(1 2 3) }
3. evaluation occurs and (3 4 5) is returned

More on Closures

```
>(define ff (lambda (x) (* 2  x)))  ;binds ff to a closure
>ff  ; can' t see the closure ff is bound to
#<CLOSURE (x) (* 2 x)>
>(ff  3)  ;evaluation in empty environment {}
6
>( ff )  ; error, can't evaluate a closure without its arguments
```

similarly, if you define a function with 2 arguments, you need to evaluate it on both arguments! can we do better? yes

Currying allows us to build functions from partial evaluation of other functions!

Currying

```
>(define (mm x  y) (* x  y))
>mm ; returns a closure
>(mm 2)  ; returns error because mm expects 2 arguments, not 1!
>(mm 2 3) ; returns 6
(define hh (lambda (x) (lambda (y) (* x y) )))
> hh ; closure is value returned
#<CLOSURE (x) (lambda (y) (* x y))> ; closure returned
> (hh 2)
#<CLOSURE (y) (* x y)>
> ((hh 5 ) 3) ; note how have to give arguments to a curried function
15              ;one by one
> ((hh 2) 3)  ; with first argument 5, (hh 5) is the 5 times function
6               ;with first argument of 2, (hh 2) is the 2 times function
```

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+ x) (lambda (y) (+ x y)))

    ((curried+ 2) 3) yields 5

    (let ((f (curried+ 1))) (f 4)) yields 5