List Processing

Recursive Functions on Lists

;;* len *requires a list as an argument

```
(define (len x)  (cond ( (null?  x)  0)
                       (else        (+ 1 (len (cdr  x))))))
```

Trace: (len '(1 2)) --top level call

```
Trace: (len '(1  2))  --top level call
       x = (1  2)
              (len '(2)) --recursive call 1
              x = (2)
                     (len '( ) ) -- recursive call 2
                     x = ( )
                     returns 0 --return for call 2
              returns (+  1  0) =1 --return for call 1
       returns (+ 1  1) = 2 --return for top level call
```

List Append (vs. cons)

```
(define (app  x  y)  ;;takes 2 lists as arguments
   (cond  ((null?  x)  y)
           (else  (cons (car x)  (app (cdr x)  y)))))

(app ‘( )  ‘( ) )   ==>  ( )
(app ‘( )  ‘( 1 4 5))  ==>  (1 4  5)
(app ‘(5  9) ‘(a  (4)  6))  ==>  (5  9  a  (4)  6)
```


Atomcount Function

```
;; atom is a non-pair -- something that has no car and cdr
(define  (atomcount  x)
   (cond  ((null?  x)         0)
           (not (pair? x))  1)
           (else  (+ (atomcount  (car x))  (atomcount  (cdr x)))) ))
```

```
(atomcount ‘(a))   --> 1
Trace: (atomcount ‘((a b) ((d)) )
   (+   (atomcount ‘(a b)
              (+        (atomcount a)
                               1
                        (atomcount ‘(b)
                               (+ (atomcount b)  --> 1
                                  (atomcount ()) --> 0
      (atomcount ‘((d)) )
              (+ (atomcount ‘(d)) ---> ... ---> 1
                 (atomcount ())    --> 0
```

Equality Testing

eq?

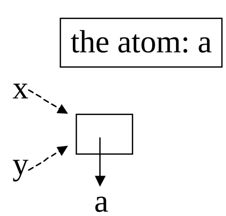
- **predicate that can check symbols for equal values**
  - **by storing unique internal value**
- **may also check other atoms for equality**
  (but not reliably on numbers and chars! for that use eqv?)
- **doesn't do what you might expect on lists because**
  (cons 'a '()) and (cons 'a '()) return different objects

equal?

- **(recursive) comparison function, ok for lists too**

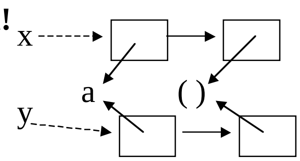
How eq? works

(define (f x y) (list x y))
so (f 'a 'a) yields (a a).
How does Scheme implement this?
It binds both x and y to the same atom a.
eq? checks that x and y both point to the same place



Say we called (f '(a) '(a)). then x and y don't point to the same list at all!

the arguments: (a)



Higher Order Functions

- **Functions as 1st class values**
- **Functions as arguments**

(define (f g x) (g (car x)))

(f number? ‘(0 a)) yields #t

(f len ‘((2 3) (4))) 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**

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

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

- **Generalized map:** f can have n arguments, and n lists are passed in

(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)
```

314F’01 AB/LS/BGR

9

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?

314F’01 AB/LS/BGR

10

apply

apply is a built-in function whose first argument f is a function and whose second argument ls is a *list* of arguments for that function; (apply f ls) evaluates f with the parameters in list ls.

Why needed? (+ '(1 2 3)) --> type error; instead:

(apply + '(1 2 3)) --> (+ 1 2 3) --> 6

(apply (lambda (n) (+ 1 n)) '(3)) --> 4

The power of *built-in* apply is that it lets a function like + take a non-fixed number arguments.

If f takes one argument only, then

(define (apply1 f x) (f x)) does the job

Beware:

(apply null? '(1 2)) --> (null? 1 2) --> type error; instead:

(apply null? (list (list 1 2))) --> (null? (list 1 2)) --> #f

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:

(apply f ls) == (eval (cons f ls))

reduce

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

(reduce f (a b c d) unit) == ***f(a,f(b,f(c,f(d,unit))))***

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

***Conceptual trace*:**

```
(reduce + '(10 20 30) 0)  -->
(+  10 (reduce + (20 30) 0) )
(+ 10  (+ 20 (reduce + (30) 0) ))
(+ 10  (+ 20  (+ 30 (reduce + ( ) 0) )))
(+ 10 (+ 20 (+ 30  0))))  yields 60
```


Using reduce

Defining len (list length function) via reduce.

```
(define (len z) (reduce (lambda (x  y) (+ 1 y))  ls   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 '(b c d)) -->
(reduce (lambda (x y) (+ 1 y)) ‘(b c d) 0))
((lambda (x y) (+ 1 y)) b (reduce (lambda (x y) (+ 1 y)) ‘(c d) 0))
((lambda…) c (reduce (lamb…) ‘(d) 0))
((lamb..)d (reduce (lamb…) ‘( ) 0))
0
“((lambda (x y) (+ 1 y)) d 0)” yields 1
((lambda (x y) (+ 1 y)) c 1) yields 2
((lambda (x y) (+ 1 y)) 1 2) yields 3
3

Using reduce to define other functions

- **Generalize applying binary function to a list of values:**
  `v f v f v f id` **vs** `(reduce f (v v v) id)`
  (reduce append ‘((1 2) (3 4) (5 6 7)) ‘( )) ---> (1 2 3 4 5 6 7)
  (reduce * ‘(3 1 4) 1) ---> 12
  (reduce * ‘(3 1 4) 0) ---> ??
  (reduce max ‘(3 2 4) ??)
- **Define old/new functions on list using primitives**
  What is (reduce cons lst ‘()) ?
  (append first second) is (reduce cons first second) !!!