Lists Revisited

Recall the Cons Cell Representation:

The *pair* or *cons cell* is the most fundamental of Scheme's structured object types.

A **list** is a sequence of **pairs**; each pair's cdr is the next pair in the sequence.

The cdr of the last pair in a **proper list** is the empty list. Otherwise the sequence of pairs forms an **improper list**. I.e., an empty list is a proper list, and and any pair whose cdr is a proper list is a proper list.

An improper list is printed in **dotted-pair notation** with a period (dot) preceding the final element of the list. A pair whose cdr is not a list is often called a **dotted pair**

Creating lists

`quote`: `'(1 (2 3) ()) => (1 (2 3) ())`
  or `(quote (1 (2 3) ())) => (1 (2 3) ())`

`list`: `(list 1 '(2 3) ()) => (1 (2 3) ())`

`cons`: Build it, piece by piece.

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

`append`: Appending lists

```
(append lst '(4 5)) => ((1 (2 3) () 4 5))
```

cons vs. list: The procedure `cons` actually builds *pairs*, and there is no reason that the cdr of a pair must be a list.

The procedure `list` is similar to `cons`, except that it takes an arbitrary number of arguments and always builds a proper list.

E.g., `(list 'a 'b 'c)` → `(a b c)`

Testing for Equality

- `(eq? a b)`: Returns `#t` iff `a` and `b` are the same Scheme object. (Don't use `eq?` with numbers!)
- `(= a b)`: Returns `#t` iff `a` and `b` are numerically equal. Pre: `a` and `b` must evaluate to numbers.
- `(eqv? a b)`: Similar to `eq?`, but works for numbers and characters. More expensive than `eq?`, however.
- `(equal? a b)`: Returns `#t` iff `a` and `b` have the same structure and contents. Thus, `equal?` recursively tests for equality. The most expensive equality predicate.

Recommended Reading:
 Dybvig §6.1, 2nd ed. (available online), or
 Dybvig §6.2, 3rd ed.

Testing for Equality (cont.)

The `eq?` predicate doesn't work for lists.

Why not?

1. `(cons 'a '())` makes a new list
2. `(cons 'a '())` makes a(nother) new list
3. `eq?` checks if its two args are *the same*
4. `(eq? (cons 'a '()) (cons 'a '()))` evaluates to () (ie, #f)

Lists are stored as pointers to the first element (car) and the rest of the list (cdr).

Symbols are stored uniquely, so `eq?` works on them.

Equality Checking for Lists

For lists, need a comparison procedure to check for the same **structure** in two lists. How might you write such a procedure?

```
(define (myequal? x y)
  (or (and (atom? x) (atom? y) (eq? x y))
      (and (not (atom? x)) (not (atom? y))
           (myequal? (car x) (car y))
           (myequal? (cdr x) (cdr y)))))
```

- `(equal? 'a 'a)` evaluates to `#t`
- `(equal? 'a 'b)` evaluates to `()`
- `(equal? '(a) '(a))` evaluates to `#t`
- `(equal? '((a)) '(a))` evaluates to `()`

Does this really work? Hint: atoms are numbers, does this work for numbers? Play around with it and with the built-in predicate procedure `equal?`.

Other Useful Predicates

- `(null? a)`: Returns `#t` iff `a` is the empty list (or `#f`, depending on the implementation).
- `(pair? a)`: Returns `#t` iff `a` is a pair, *i.e.*, a cons cell.
- `(number? a)`: Returns `#t` iff `a` is a number.
- `(min list)`: Returns the minimum of a list of numbers.
- `(max list)`: Returns the maximum of a list of numbers.
- `(even? a)`: Returns `#t` iff `a` is even.

Lots more in Dybvig §6.

Recursive Procedures: Counting

```
(define (atomcount x)
  (cond ((null? x) 0)
        ((atom? x) 1)
        (else (+ (atomcount (car x))
                 (atomcount (cdr x))))))
```

- `(atomcount '(1 2))` ⇒ 2
- `(atomcount '(1 (2 (3)) (5)))` ⇒ 4:

```
(at '(1 (2 (3)) (5)))
(+ (at 1) (at ((2 (3)) (5))))
(+ 1 (+ (at (2 (3))) (at ((5)))))
(+ 1 (+ (+ (at 2) (at ((3)))) (+ (at (5)) (at ()))))
(+ 1 (+ (+ 1 (+ (at (3)) (at ()))) (+ (+ (at 5) (at ())) 0)))
(+ 1 (+ (+ 1 (+ (+ (at 3) (at ())) 0)) (+ (+ 1 0) 0)))
(+ 1 (+ (+ 1 (+ (+ 1 0) 0)) (+ 1  0)))
(+ 1 (+ (+ 1 (+ 1 0)) 1))
(+ 1 (+ (+ 1 1) 1))
(+ 1 (+ 2 1))
(+ 1 3)
4
```

This is called "car-cdr-recursion.'

Efficiency Issues

Problem: Evaluating the same expression twice.

Example:

```
(define (longest-nonzero x y)
  (cond ((and (null? x) (null? y)) -1)
        ((> (length x) (length y))
         (length x))
        (else (length y))

  ))
```

What can you do if there is no assignment statement?

Efficiency Issues

Solution 1: Bind values to parameters in a helper procedure.

```
(define (maximum x y)
  (cond ((> x y) x)
        (else y)
  ))

(define (longest-nonzero x y)
  (cond ((and (null? x) (null? y))  -1)
        (else
         (maximum (length x) (length y)))
  ))
```

Note: There is a built-in `max` function.

Note 2: Helper procedures are an important and useful tool!

Efficiency Issues

Solution 2: Use a `let` or `let*` construct, that binds variables to expression results.

```
(let ((var1 expr1)
      ...
      (varn exprn))
 <vars are defined and can be used here>)
(let* ((var1 expr1)
        ...
       (varn exprn))
 <vars are defined and can be used here>)
```

Polymorphic and Monomorphic Functions

- *Polymorphic* functions can be applied to arguments of many forms
- The function `length` is polymorphic: it works on lists of numbers, lists of symbols, lists of lists, lists of anything
- The function `square` is monomorphic: it only works on numbers

Higher-Order Procedures

Procedures as input values:

```
(define (all-num lst)
  (or (null? lst)
      (and (number? (car lst))
           (all-num (cdr lst))))
  )
(define (all-num-f f lst)
  (cond ((all-num lst) (f lst))
        (else 'error))
  )
1 ]=> (all-num-f abs-list '(1 -2 3))
;Value 1: (1 2 3)

1 ]=> (all-num-f abs-list '(1 a))
;Value: error
```

Higher-Order Procedures

Procedures as returned values:

```
(define (plus-list x)
  (cond ((number? x)
         (lambda (y) (+ (sum-n x) y)))
        ((list? x)
         (lambda (y) (+ (sum-list x) y)))
        (else (lambda (x) x))
  ))
1 ]=> ((plus-list 3) 4)
;Value: 10

1 ]=> ((plus-list '(1 3 5)) 5)
;Value: 14
```

Built-In Higher-Order Procedures: `map`

There is a built-in procedure `map`. Let's define our own restricted version first....

```
(define (mymap f l)
  (cond ((null? l) '())
        (else (cons (f (car l))
                    (mymap f (cdr l))))
  ))
```

- `mymap` takes two arguments: a function and a list
- `mymap` builds a new list whose elements are the result of applying the function to each element of the (old) list

Higher-order Procedures: `map`

- Example:

  `(mymap abs '(-1 2 -3 4))` $\Rightarrow$
  `(1 2 3 4)`

  `(mymap (lambda (x) (+ 1 x)) '(-1 2 -3))` $\Rightarrow$
  `(0 3 -2)`

- The built-in `map` will produce the same results, but note that the built-in `map` can take more than two arguments:

  `(map cons '(a b c) '((1) (2) (3)))` $\Rightarrow$
  `((a 1) (b 2) (c 3))`

What's Wrong Here??

```
1 ]=>
(define (atomcount s)
  (cond ((null? s) 0)
        ((atom? s) 1)
        (else (+ (map atomcount s)))
  ))
;Value: atomcount
1 ]=> (atomcount '(a b))

;The object (1 1), passed as an argument
;to +, is not the correct type.
...
2 error>
```

Why doesn't this work?

Using `eval` to Correct the Problem

```
(define (atomcount s)
 (cond ((null? s) 0)
       ((atom? s) 1)
       (else
        (eval
         (cons '+ (map atomcount s)) '()))
))
1 ]=> (atomcount '(a b))

;Value: 2

1 ]=> (atomcount '((1) (2 3 (4)) ((((5))))))

;Value: 5
```

Limitations of Using `eval`

BUT: `eval` only works in the current definition of `atomcount` because numbers evaluate to themselves.

```
1 ]=> (+ 1 2 3)
;Value: 6

1 ]=> (cons '+ '(1 2 3))
;Value 12: (+ 1 2 3)

1 ]=> (eval (cons '+ '(1 2 3)) '())
;Value: 6
```

Using `eval` to Evaluate Expressions

```
1 ]=> (append '(a) '(b))
;Value 13: (a b)
1 ]=> (cons 'append '((a) (b)))
;Value 14: (append (a) (b))

1 ]=> (eval (cons 'append '((a) (b))) '())
;Unbound variable: b
...
1 ]=> (cons 'append '( '(a) '(b) ))
;Value 15: (append (quote (a)) (quote (b)))

1 ]=> (eval
         (cons 'append '( '(a) '(b))) '())
;Value 16: (a b)
```

Too complicated!!

Applying Procedures with `apply`

```
1 ]=> (apply + '(1 2 3))
;Value: 6
1 ]=> (apply append '((a) (b)))
;Value 5: (a b)

1 ]=>
(define (atomcount s)
 (cond ((null? s) 0)
       ((atom? s) 1)
       (else
        (apply + (map atomcount s)))))

;Value: atomcount
1 ]=>  (atomcount '(a (b) c))
;Value: 3
```