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 '(1) '(4 5)) => (1 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, to create *local* variables and to bind them to expression results. The scope of these variables is limited to the scope of the let statement.

```
(let ((var1 expr1)
      ...
      (varn exprn))
      body
)
```

The variables can only be used within the body of the let.
Evaluation: expr1, ... exprn are evaluated in some **undefined order**, saved, and then assigned to var1,..varn. In our interpreter, they have the appearance of being evaluated **in parallel**.

```
(let* ((var1 expr1)
       ...
       (varn exprn))
       body
)
```

Again, the variables can only be used within the body of let*.

Evaluation: evaluation and binding is **sequential**, i.e., the evaluation of `expr1` is bound to `var1`, the evaluation of `expr2` is then bound to `var2`, etc.

Let and let* Example

```
(define a 100) (define b 200) (define c 300)

(let ((a 5)
      (b (+ a a))
      (c (+ a b)))
     (list a b c)
)
```

What does this return? What are a, b, c bound to now? (Answer: still 100, 200, 300)

```
(let* ((a 5)
      (b (+ a a))
      (c (+ a b)))
     (list a b c)
)
```

What does this return?

Note that `let*` can be simulated by nested `let`s.

```
(let ((a 5))
   (let ((b (+ a a)))
      (let ((c (+ a b)))
         (list a b c)
      )
    )
)
```


Structured Data - Binary Search Trees

Nested lists can be used to define a variety of data structures.

E.g., A complete binary tree can be represented as a list with 3 elements: `(root left-subtree right-subtree)`

```
1 ]=> (define mytree
         '(dog (bird (aardvark () ()) (cat () ()))
                     (possum (frog () ()) (wolf () ()))))
;Value: mytree


1 ]=> mytree
;Value 1: (dog (bird (aardvark () ()) (cat () ()))
                     (possum (frog () ()) (wolf () ())))

1 ]=> (car mytree)
;Value: dog

1 ]=> (car (cdr mytree))
;Value 2: (bird (aardvark () ()) (cat () ()))
```

Binary Search Trees (cont.)

```
1 ]=> (define empty-tree?
              (lambda (tree) (null? tree)))
;Value: empty-tree?

1 ]=> (define left-tree
              (lambda (tree)
                 (if (empty-tree? tree) 'Error
                                        (cadr tree))))
;Value: left-tree

1 ]=> (left-tree mytree)
;Value 2: (bird (aardvark () ()) (cat () ()))

1 ]=> (define right-tree
              (lambda (tree)
                 (if (empty-tree? tree) 'Error
                                        (caddr tree))))
;Value: right-tree

1 ]=> (right-tree mytree)
;Value 3: (possum (frog () ()) (wolf () ()))
```


Binary Search Tree (cont.)

```
1 ]=> (define root-tree
              (lambda (tree)
                 (if (empty-tree? tree) 'Error
                                        (car tree))))
;Value: root-tree

1 ]=> (root-tree mytree)
;Value: dog

1 ]=> (define contains?
         (lambda (tree sym)
           (cond ((empty-tree? tree) ())
                 ((equal? (root-tree tree) sym) #t)
                 (else (or (contains? (left-tree tree) sym)
                           (contains? (right-tree tree) sym)
      )))))
;Value: contains?

1 ]=> (contains? mytree 'aardvark)
;Value: #t


1 ]=> (contains? mytree 'elephant)
;Value: ()
```

Binary Search Tree (cont.)

```
1 ]=> (define pre-order
         (lambda (tree)
            (if (null? tree) '()
                (cons (root-tree tree)
                      (append (pre-order (left-tree tree))
                              (pre-order (right-tree tree))
       )))))
;Value: pre-order

1 ]=> (pre-order mytree)
;Value 4: (dog bird aardvark cat possum frog wolf)

1 ]=> (define in-order
         (lambda (tree)
            (if (null? tree) '()
                (append (in-order (left-tree tree))
                        (cons (root-tree tree)
                              (in-order (right-tree tree))
      ))))
;Value: in-order

1 ]=> (in-order mytree)
;Value 5: (aardvark bird cat dog frog possum wolf)
```


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))` ⇒
  `(1 2 3 4)`

  `(mymap (lambda (x) (+ 1 x)) '(-1 2 -3))` ⇒
  `(0 3 -2)`

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

  `(map cons '(a b c) '((1) (2) (3)))` ⇒
  `((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
```


Higher-order Procedures: `my-reduce`

```
(define (my-reduce op l id)
 (if (null? l)
     id
     (op (car l)
         (my-reduce op (cdr l) id))
))
```

A binary $\mapsto$ n-ary procedure.

The `my-reduce` procedure takes a binary operation and applies it right-associatively to a list of an arbitrary number of arguments.

NOTE: `my-reduce` is not equivalent to `apply`.

Higher-order Procedures: `my-reduce`

`(my-reduce + '(1 2 3) 0)` ⇒ 6:

```
(my-reduce + '(1 2 3) 0)
(+ 1 (my-reduce + '(2 3) 0))
(+ 1 (+ 2 (my-reduce + '(3) 0)))
(+ 1 (+ 2 (+ 3 (my-reduce + '() 0))))
(+ 1 (+ 2 (+ 3 0)))
6
```

Note: `(+ 1 2 3)` ⇒ 6

`(my-reduce / '(24 6 2) 1)` ⇒ 8:

```
(my-reduce / '(24 6 2) 1)
(/ 24 (my-reduce / '(6 2) 1))
(/ 24 (/ 6 (my-reduce / '(2) 1)))
(/ 24 (/ 6 (/ 2 (my-reduce / '() 1))))
(/ 24 (/ 6 (/ 2 1)))
8
```

Note: `(/ 24 6 2)` ⇒ 2

Higher-order Procedures: `my-reduce`

Given `union`, which takes two lists representing sets and returns their union:

```
1 ]=> (apply union '((1 3)(2 3 4)))
;Value 21: (1 2 3 4)

1 ]=> (apply union '((1 3)(2 3)(4 5)))
;The procedure #[compound-procedure union]
;has been called with 3 arguments;
;it requires exactly 2 arguments.

1 ]=> (reduce union '((1 3)(2 3)(4 5)) '())
;Value 22: (1 2 3 4 5)
```

Question: How would you have to change `my-reduce` to be able to take `intersection` as its function argument?

Important

Note that Scheme has a built-in higher-order procedure `reduce` that is different from `my-reduce`. You may use `my-reduce` in assignments and tests. In assignments, you would of course have to define it by copying the code provided here. In tests, you may use it without defining it.

Example Practice Procedures

- cdrLists: given a list of lists, form new list giving all elements of the cdr's of the sub-lists.
  ((1 2) (3 4 5) (6)) $\Rightarrow$ (2 4 5)

- swapFirstTwo: given a list, swap the first two elements of the list.
  (1 2 3 4) $\Rightarrow$ (2 1 3 4)

- swapTwoInLists: given a list of lists, form new list of all elements in all lists, with first two of each swapped.
  ((1 2 3)(4)(5 6)) $\Rightarrow$ (2 1 3 4 6 5)

- addSums: given a list of numbers, sum the total of all sums from 0 to each number.
  (1 3 5) $\Rightarrow$ 22