MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 1998

Recitation – Friday, February 27

0. Announcements

- Quiz 1 coming up! Wednesday, March 11th, 1998. 5pm - 7pm XOR 7pm - 9pm.

1. Student Presentation

The two programs below were found in a trash can, perhaps because they didn't work quite right. From their names, they appear to be attempts to create a list whose elements are the same but in reverse order from some starting list. With the help of box-and-pointer diagrams, explain what each actually does.

```
(define (reverse L1)
  (cond ((null? L1) nil)
        (else (reverse (cdr L1))
              (cons (car L1) (cdr L1)))))
(define (reverse L1)
  (if (null? L1)
      nil
      (cons (reverse (cdr L1)) (car L1))))
```

2. Symbols, Quote, and Eq?

Draw the box and pointer diagrams for each of the following expressions, and write what is printed out by the interpreter:

```
(cons 'a '(b))      ==>
(cons '(a) '(b))    ==>
(list 'a 'b)        ==>
(list '(a) '(b))    ==>
(append '(a) '(b)) ==>
```

Given the following definitions, what are the values of the eq? expressions below?

```
(define a 2)                  (eq? a b)  ==>
(define b 'two)               (eq? d 'b) ==>
(define c b)                  (eq? b d)  ==>
(define d 'b)                 (eq? b c)  ==>
```

Given the a-list data structure below, evaluate the following assq expressions:

```
(define desc
  '((where Room-34-304)
    (what (course 6.001)
          (section 4))
    (when (day Friday)
          (month February)
          (date 27)
          (year 1998)
          (hour 10)
          (minute 00))
    (why (scheme rules!))))
```

```
(define (assq name a-list)
  (cond ((null? a-list) #f)
        ((eq? name (caar a-list))
         (car a-list))
        (else (assq name (cdr a-list)))))


(assq 'why desc)  ==> (why (scheme rules!))
(assq 'what desc) ==> (what (course 6.001) (section 4))
(assq 'date (assq 'when desc)) ==> <ERROR>
```

3. Comparing Trees

We can use `eq?` to compare symbols. What about trees? Write the function `equal?` that takes two trees of symbols and returns true if the same symbols are arranged in the same structure.

```
(equal? '(this is a list) '(this is a list))
;Value: #t

(equal? '(this (is a) list) '(this (is a) list))
;Value: #t

(equal? '(this is a list) '(this (is a) list))
;Value: #f
```

```
(define (equal? a b)
  (or (and (null? a) (null? b))
      (and (symbol? a) (symbol? b) (eq? a b))
      (and (pair? a) (pair? b)
           (equal? (car a) (car b))
           (equal? (cdr a) (cdr b)))))
```

4. Occurrences

Write the function `occurrences` that takes a symbol and a tree and counts the number of times that symbol appears in the tree. For example, `(occurrences 'a '((a b) ((a a) c b (((a))))) ==> 4`

```
(define (occurrences symbol tree)
  (cond ((null? tree) 0)
        ((eq? symbol tree) 1)
        ((not (pair? tree)) 0)
        (else
         (+ (occurrences symbol (car tree))
            (occurrences symbol (cdr tree))))))
```

5. All Pairs

Write the function `all-pairs` that takes a list and returns a list of all pairs of the elements in the list. For example, `(all-pairs '(1 2 3 4)) ==> ((1 2) (1 3) (1 4) (2 3) (2 4) (3 4))`

```
(define (all-pairs s)
  (if (null? s)
      nil
      (append (map (lambda (x) (list (car s) x)) (cdr s))
              (all-pairs (cdr s)))))
```

6. Fringe

Write the function `fringe` that takes a tree structure and returns a flat list of the leaves of the tree. For example `(fringe (list 1 (list 2) 3)) ==> (1 2 3)`

```
(define (fringe x)
  (cond ((null? x) nil)
        ((not (pair? x)) (list x))
        (else
          (append (fringe (car x))
                  (fringe (cdr x))))))
```