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

Recitation – Wednesday, February 25

1. Student Presentation

Using box and pointer notation, explain how map works. In particular, use a simple example of applying map to a list, e.g.

```
(map (lambda (x) (* x 2)) (list 1 2 3))
```

and show at each stage what box and pointer structures are present and which ones are arguments to a call to map .

2. List Warm-Up

`Map` takes a procedure and a list as arguments, and returns a list of the results of applying the procedure to each item in the list. Using `map`, write a procedure `apply-all` that takes a list of procedures and an item as arguments and returns a list of the results of applying each procedure to the item.

```
==> (apply-all (list sqrt square cube factorial fib) 4)
(2 16 64 24 3)
(define (apply-all ops x)
    (map (lambda (op) (op x)) ops))
```

3. Recursive Append

Let's take a closer look at the procedure `append` that was introduced in lecture:

```
(define (append a b)
  (if (null? a)
      b
      (cons (car a) (append (cdr a) b))))
```

Draw the box and pointer diagrams for `(append (list 1 2) (list 3 4 5))`. Notice that the second list is never looked at!

4. Copy

Consider the procedure `copy` which takes a list and returns a copy of the list. How do each of the following differ?

```
(define (copy-ident x) x)

(define (copy-map x) (map (lambda (y) y) x))

(define (copy-recurse x)
  (if (null? x)
      nil
      (cons (car x) (copy-recurse (cdr x)))))
```

Notice that `copy-map` and `copy-recurse` are both recursive processes. Let's write an iterative copy:

Warning: the below is not copy!

```
(define (*copy-iter* x)
  (define (aux x ans)
    (if (null? x)
        ans
        (aux (cdr x) (cons (car x) ans))))
  (aux x nil))
```

The above is not copy. Actually, it's reverse!

```
(define (copy-iter x) (reverse (reverse x)))
```

5. Iterative Append

Given what we learned about iterative vs. recursive processes operating on lists, write an iterative version of append.

```
(define (append a b)
  (define (aux x ans)
    (if (null? x)
        ans
        (aux (cdr x) (cons (car x) ans))))
  (aux (reverse a) b))
```

6. Lists Vs. Trees

So far, we've been working on lists, while we've ignored the elements of the list. What does the following return?
`(reverse (list 1 (list 2 3) (list 4 5 6)))`

Write a function `deep-reverse` that when called on the above tree will reverse all the elements.
`(deep-reverse (list 1 (list 2 3) (list 4 5 6))) ==> ((6 5 4) (3 2) 1)`

```
(define (deep-reverse x)
  (define (aux x ans)
    (cond ((null? x) ans)
   ((not (pair? x)) x)
          (else
    (aux (cdr x) (cons (deep-reverse (car x)) ans)))))
  (aux x nil))
```

7. More Trees

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))))))
```