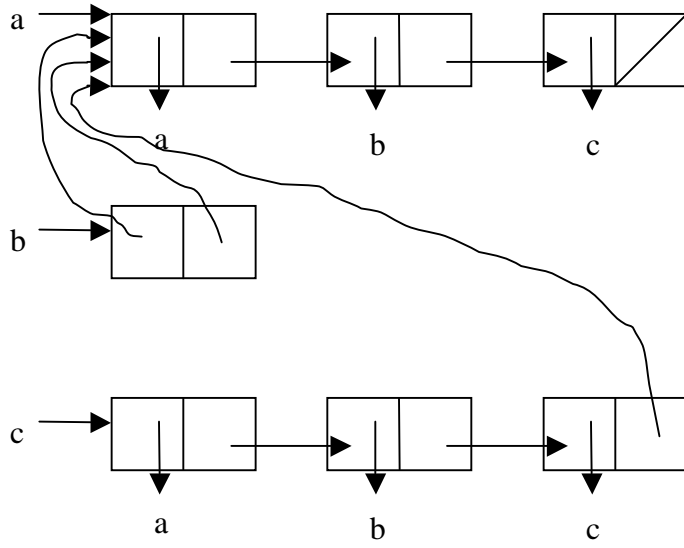
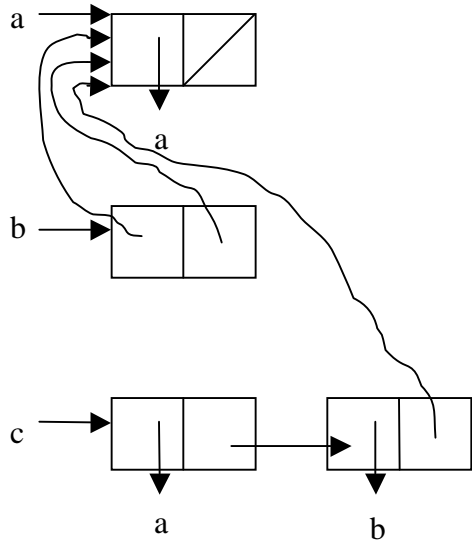


**Solutions to Sample Final Exam**

**Problem 1**



((a b c) a b c)



((a) a)

## ***Problem 2***

```
(define (map-two-streams f s1 s2)
  (cons-stream (f (stream-car s1)
                  (stream-car s2))
               (map-two-streams f
                                 (stream-cdr s1)
                                 (stream-cdr s2))))

(define (min-two-streams s1 s2)
  (map-two-streams min s1 s2))
```

There is one multiplication done to compute each element of the stream, so the  $n$ th element will need  $n$  multiplications. However, if the answer was already computed, no multiplications would be necessary. Or if the  $(n-1)$ th element had been computed, just one more multiplication is necessary.

This is a bit of a trick question. The  $n$ th element still needs  $n$  multiplications to be computed. However, if we ask for the  $(n+1)$ th element after computing  $n$ , we'll need to redo all of the prior  $n$  multiplications since we're not using memoization.

## ***Problem 3***

```
(define (make-opl-lecturer name birthplace threshold)
  (let ((person (make-person name birthplace threshold)))
    (lambda (message)
      (cond ((eq? message 'bring-cookies-to-exam)
             (lambda (self)
               (let ((location (ask self 'place)))
                 (ask self 'move-to cookie-store)
                 (ask self 'take cookies)
                 (ask self 'move-to location))))
            ((eq? message 'say)
             (lambda (self stuff)
               (ask person 'say stuff)
               (ask person 'say '(Scheme is fun!))))
            (else (get-method person message))))))
```

#### ***Problem 4***

Insert before application? in mc-eval:

```
((infix? exp)
 (mc-apply (mc-eval (infix-operation exp) env)
            (list-of-values (infix-operands exp) env)))
```

Rest of necessary code:

```
(define (infix? exp)
  (or (eq? (cadr exp) 'uml:+)
      (eq? (cadr exp) 'uml:-)
      (eq? (cadr exp) 'uml:*)
      (eq? (cadr exp) 'uml:/)))

(define (infix-operator exp) (cadr exp))

(define (infix-operands exp)
  (cons (car exp) (caddr exp)))
```

#### ***Problem 5***

```
(define (make-frame variables values)
  (if (null? variables)
      nil
      (cons (cons (car variables) (car values))
            (make-frame (cdr variables) (cdr values)))))
```

#### ***Problem 6***

Mark and sweep:

0	1	2	3	4	5	6	7	8	9
X	X	P5	X	X	N1	X	X	X	X
E0	P0	N3	P1	P3	N4	P4	P6	P7	E8
0	0	10	0	0	10	0	0	0	0

Root: P2

Free: P9

Stop and copy:

0	1	2	3	4	5	6	7	8	9
P2	P7	♥	P6	N2	♥	N2	P2	N6	P5
P7	N7	P10	P1	P1	P11	P7	P3	N7	P4

10	11	12	13	14	15	16	17	18	19
P11	N1								
N3	N4								

Root: P10

Free: P12

Stop and copy is better when there is more garbage than good cells. Only the good cells are copied. With mark and sweep, all cells must be touched, regardless of whether they are good or garbage.

Mark and sweep uses (almost) all of the memory, while stop and copy can only use half of the memory for its active space.

### ***Problem 7***

The syntactic analysis is only done once. For a recursive function, this will save time.

Yes, we may do extra work. For example, if an “if” statement is only used once, we will have analyzed both the consequent and alternative, although only one will be evaluated.

### *Extra Credit*

Here's the complete function. The underlined portions show what was changed from the original.

```
(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan frame)
      (cond ((null? frame)
             (env-loop (enclosing-environment env)))
            ((eq? var (caar frame)) (cdar frame))
            (else (scan (cdr frame)))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable" var)
        (scan (first-frame env))))
  (env-loop env))
```

The following procedures would need to be changed:

```
set-variable-value!
define-variable!
add-binding-to-frame!
frame-variables
frame-values
```