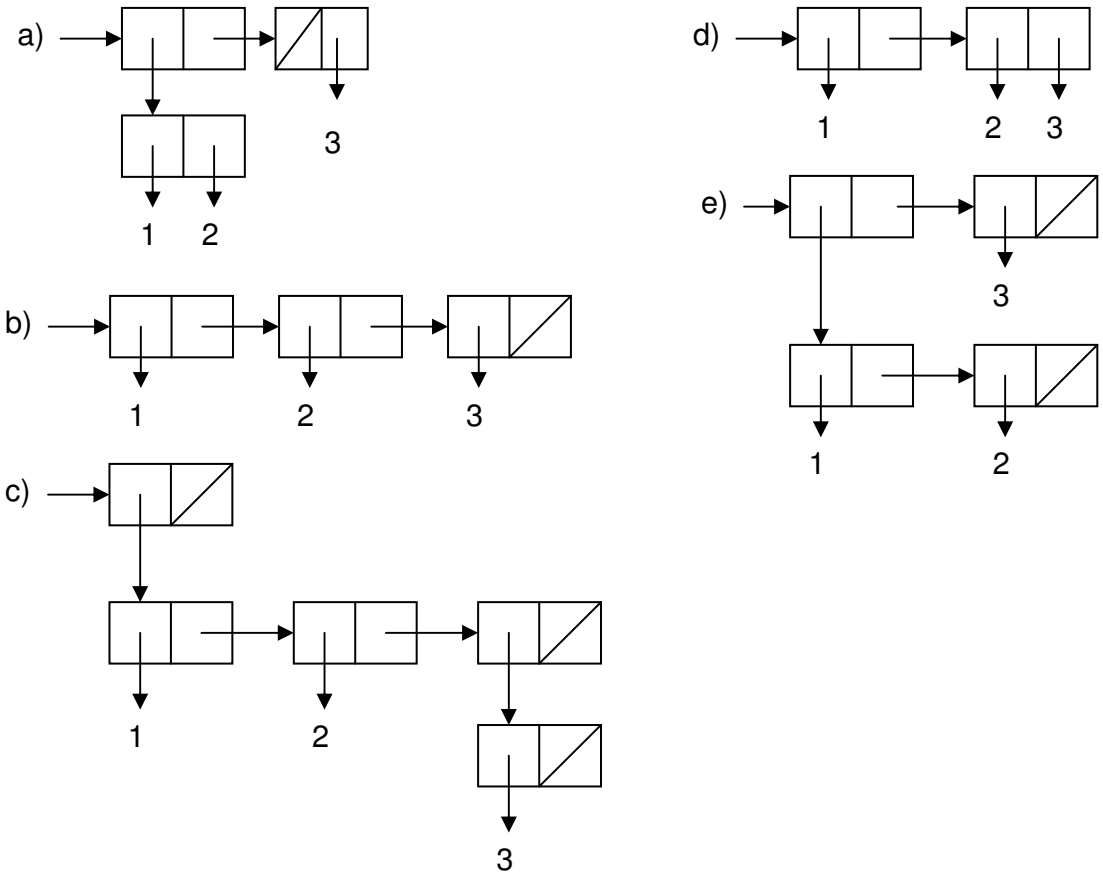
**CS 61A, Fall, 2002, Midterm #2, L. Rowe**

1. (10 points, 1 point each part) Consider the following five box-and-arrow diagrams.




For each of the following Scheme expressions, indicate which diagram is produced when the expression is evaluated. It may be that some diagrams above are not used in any answer below, and the diagrams for some answers may appear above. Enter the answer "none" if the result of evaluating the expression does not match a diagram above.

(i) (list (cons 1 2) (cons '() 3))

(ii) (list (append 1 2 '(3)))

(iii) '((1 2 (3)))

(iv) (accumulate cons nil
(filter number? '(a 1 b 3 c d 3)))

(v) (list 1 2 '(3))

(vi) (let ((x '(1))) (set-cdr! x (cons 2 3)))

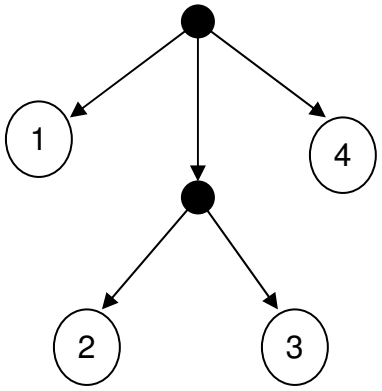
(vii) (append (list '(1 2)) '(3))

(viii)
```
(let ((x (list 1 2 3)))
        (set-car! (cddr x) (cons (caddr x) nil))
        (list x))
```

(ix) '((1 2) 3)

(x) '(1 (2) 3)

2. (10 points) The textbook suggests that we can view lists that contain lists as trees. For example, the list '(1 (2 3) 4) can be thought of as a tree that looks like.




As you can see, a branch of a tree connects two nodes. A node with no branches is called a *leaf node*. This tree has six nodes and five branches. Four nodes are leaves, one node is the root, and the remaining node has two leaves (i.e., 2 and 3) below it.

(i) (4 points) Draw a box-and-arrow diagram for the example list '(1 (2 3) 4). Remember to add the arrow that points at the list.

(ii) (2 points) Eva Lu Ator makes an interesting observation. She tells Louis Reasoner: "Hey Louis, observe that the number of branches in any tree is always one less than the number of nodes in the tree." Louis is not convinced that Eva is correct, so he writes the following procedures, each of which takes a tree as an argument (i.e., a list), to verify her claim:

```
(define (is-eva-right? t)
        (= (- (count-nodes t) 1) (count-branches t)))

(define (count-nodes t)
        (cond ((null? t) 1)
        ((not (pair? t)) 1)
```

```
        (else (+ (count-nodes (car t))
                 (count-nodes (cdr t))))))

(define (count-branches t)
        (cond ((null? t) 0)
        ((not (pair? t)) 0)
        (else (+ (count-branches (car t))
                 (count-branches (cdr t))))))
```

Eva remarks, "Louis, your count-nodes procedure always returns the number of leaves, not the number of nodes." Is she right? Circle your answer.

**YES** **NO**

(iii) (2 points) Eva further says, "Your count-branches procedure always computes the wrong answer if the tree is not empty." Is she right? Circle your answer.

**YES** **NO**

(iv) (2 points) If the answer to part (iii) is YES, suggest a very small change in one of the cond clauses that will rectify the error in count-branches. You can mark the change in the code above. If the answer to part (iii) is NO, draw a **non-empty** tree for which count-branches does compute the right answer.

3. (10 points) Any number greater than 1 can be expressed as a product of prime numbers. For example, the following prime decompositions illustrate this observation:

6 = 2 * 3
12 = 2 * 2 * 3
13 = 13
30 = 2 * 3 * 5
65 = 5 * 13
90 = 2 * 3 * 3 * 5
600 = 2 * 2 * 2 * 3 * 5 * 5

We are looking for "simple numbers" that are numbers for which every factor in the prime decomposition appears only once. Therefore, 6, 13, 30, and 65 in the list above are simple numbers. However, 12 is not a simple number because 2 appears twice in its prime decomposition. Similarly, 90 and 600 are not simple numbers.

We want to write a procedure all-simple-numbers that will return all simple numbers less than or equal o some number N. The algorithm for the procedure will work as follows: First write down all numbers between 2 and N. From that list of numbers, erase all numbers which have 2 factors of 2 (i.e. are divisible by 4). Next, erase factors of 5 (i.e. are divisible by 25). We skipped 4 because any number that was divisible by 4*4 is divisible by 4 and they were eliminated in the first step. Note that we are removing numbers from the list as we go along and using the next number in the list to decide

which numbers to remove. Continue this process until the end of that list is reached. At that point, the list contains only simple numbers. For example,

(all-simple-numbers 5) => ( 2 3 5)

We have given you a code skeleton for this procedure. You are to fill in the blanks so that the procedure (all-simple-numbers n) returns a list of simple numbers les that or equal to n. Remember, 1 is not simple, so the list begins with 2.

```
(define (enumerate-interval m n)
        (if (> m n)
            nil
            (cons m (enumerate0interval (+ m 1) n))))
(define (all-simple-numbers n)
        (define (helper remaining)
                (if (null? remaining)
                    nil
                    (cons _____________________
                          (helper (filter
        _____________________________________________

        _____________________________________________)))))

        (helper (enumerate-interval 2 n)))
```

Hint: you might find the remainder procedure useful.

4. (20 points) This question uses the following procedures defined for binary tress:

(make-tree e t1 t2) returns a tree node with entry e and t1 as the left child and t2 as the right child

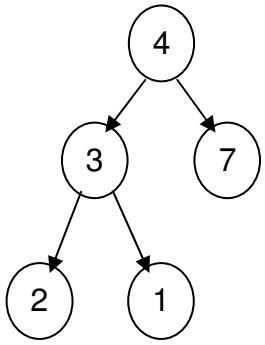
(entry t) returns the entry at node t

(left-branch t) returns the tree node at the left branch of t

(right-branch t) returns the tree node at the right branch of t

The empty tree is represented by nil (i.e., the empty list). For example, the following code builds the tree shown on the right:

```
(define two (make-tree 2 '() '()))
(define one (make-tree 1 '() '()))
(define three (make-tree 3 two one))
(define seven (make-tree 7 '() '()))
(define four (make-tree 4 three seven))
```




A *catamorphism* is a function that abstracts recursion over a structure. For example, the function accumulate described in the book is a catamorphism that operates on lists. You

are going to complete the procedure acc-tree that will work on binary trees similar to the way accumulate works on lists. acc-tree will take three arguments:

1. op – a function that takes the entry for a node, the accumulated version of the left sub-tree and the accumulated version of the right sub-tree
2. init – the base case for the accumulation
3. tree – the tree over which to perform the accumulation

For example, if we have a tree with numbers at each node entry, the call

(acc-tree + 0 four) => 17

For the tree constructed above.

**You may not use set! procedures to answer this problem.**

(i) (8 points) Fill-in the blanks to complete the definition of acc-tree.

```
(define (acc-tree op init tree)
        (if (null? tree)
            _____________________________________

            (op  ( _____________________________ )

                 ( _____________________________ )

                 ( _____________________________ ))))
```

(ii) (6 points) Use acc-tree to write the procedure map-tree that will take as arguments a procedure and a tree and return a new tree in which the procedure has been applied to every entry in the argument tree. Fill-in the blanks in the following code:

```
(define (map-tree op tree)
        (define (map-op entry left right)
                _____________________________ )

        (acc-tree _____________________________ ))
```

(iii) (6 points) Use your acc-tree procedure to convert a tree to a list of entries in infix, prefix, and postfix order. Complete the procedures infix-op, prefix-op, and postfix-op to complete the following uses of acc-tree.

```
(acc-tree infix-op nil tree)
(acc-tree prefix-op nil tree)
(acc-tree postfix-op nil tree)

(define (infix-op entry left right)
        ( _____________________________ ))

(define (prefix-op entry left right)
        ( _____________________________ ))
```

```
(define (postfix-op entry left right)
        ( _____ ))
```

5. (10 points) (Hint: read the entire problem including code before starting on this question.) This problem will explore an implementation of matrices that uses the message-passing style presented in the textbook. Specifically, assume we have defined a procedure make-matrix that takes arguments giving the number of rows and columns and a list for each row in the matrix. Matrix entries must be numbers. For example, to create the matrix

```
| 1  4  3  0 |
| 5  0  1  0 |
| 7  0  0  1 |
```

You can make the following procedure call

```
(define m (make-matrix 3 4
                       '(1 4 3 0)
                       '(5 0 1 0)
                       '(7 0 0 1)))
```

Matrices created in this way can respond to three messages:

- num-rows – returns the number of rows
- num-cols – returns the number of columns
- entry – takes a row and column number and returns the matrix entry at that position

For example, using m defined above, the following results are produced:

```
(m 'num-rows) => 3
(m 'num-cols) => 4
(m 'entry 2 0) => 7  ; numbering starts at 0
(m 'entry 3 0) => "index out of bounds"
```

The following code implements make-matrix:

```
(define (make-matrix nrows ncols . matrix)
        (lambda (op . args)
           (cond ((eq? op 'num-rows) nrows)
                 ((eq? op 'num-cols) ncols)
                 ((eq? op 'entry)
                       (let ((i (car args)) (j (cadr args)))
                         (if (and (>= i 0) (>= j 0)
                                  (< i nrows) (< j ncols))
                             (list-ref (list-ref matrix i) j)
                             (error "index out of bounds"))))
                       (else (error "message not defined"))))))
```

We want to add a new procedure to matrices that compares two of them and returns true if they are the same (i.e., they have the same number of rows and columns and the corresponding entries are equal), and false otherwise. We can do this by creating a new

constructor make-matrix-with-same? that returns a matrix with the additional procedure same?. For example,

```
(define m1 (make-matrix-with-same? 3 4
                '(1 4 3 0) '(5 0 1 0) '(7 0 0 1)))
(define m2 (make-matrix-with-same? 3 4
                '(1 4 3 0) '(5 0 1 0) '(7 0 0 1)))
(define m3 (make-matrix with same? 2 4
                '(1 4 3 0) '(5 0 1 0)))
(m1 'num-rows) => 3
(m1 'num-cols) => 4
(m1 'entry 2 0) => 7
(m1 'same? m2) => #t
(m1 'same? m3) => #f
```

Fill-in the blanks for the following definitions of make-matrix-with-same?:

```
(define (make-matrix-with-same? nrows ncols . rows)
    (let ((orig-matrix
        (apply make-matrix
            (append (list nrows ncols) rows))))
    (define (row-same? m rownum) ; m is matrix
        (define (helper column) ; check each col of row
            (if (< colnum 0)
                #t
                _____________________________________________

                _____________________________________________

                _____________________________________________ ))
            (helper (- ncols 1)))

    (define (matrix-same? m) ; m is second arg to 'same?
        (define (helper rownum) ; check each row
            (if (< rownum 0)
                #t
                (and (row-same? m rownum)
                    (helper (- rownum 1)))))
        (helper (- nrows 1)))
    (lambda (op . args)
        (if (eq? op 'same?)
            (let ((m (car args)))
            (if (and (= (m 'num-rows)
                        (orig-matrix 'num-rows))
                        (= (m 'num-cols)
                        (orig-matrix 'num-cols)))
                (matrix-same? m)
                #f))
```

( _____ )))))