CS61A, Spring 1995
Midterm #2

Question 1 (2 points):

What will Scheme print in response to the following expressions? Also, draw a ``box and pointer" diagram for the result of each expression:

```
(cddaar '(((a b c d e) (f g h i j))
          ((l m n o p) (q r s t u))))

(cons '(a b) (append '(c d) '(e f)))
```

Question 2 (2 points):

Each of the following expressions, when evaluated, returns a list. Identify the *type* (e.g., procedure, non-numeric word, list, number) of the *last* element in each of the lists.

```
'(list (count 'hello))

'(list count)

(list 'count count)

(list 'hello (count 'hello))
```

Question 3 (5 points):

This question concerns the *tree* abstract data type, as defined by these selectors and constructor:

```
(define datum car)
(define children cdr)
(define make-node cons)
```

Write the higher-order procedure `tree-accumulate`. Its two arguments are a function `fn` and a tree. The function `fn` will take two arguments and will be associative. (That is, you don't have to worry about the order in which you find the nodes in the tree.) `Tree-accumulate` will combine all of the `datum` elements in the tree by applying `fn` to them two at a time, analogous to ordinary `accumulate`. For example:

```
> (define my-tree
    (make-node 3 (list (make-node 4 '())
                       (make-node 7 '())
```

```
                        (make-node 2 (list (make-node 3 '())
                                           (make-node 8 '())))))))
> (tree-accumulate + my-tree)
27
> (tree-accumulate max my-tree)
8
```

Respect the tree data abstraction!

Hint: The book's version of `accumulate` has an extra argument to provide a return value for the case in which no numbers are in the range of values provided. We don't need that here, because a tree always has at least one node, so there is always at least one value available. If you only have one value, just return that value; the function `fn` would require two arguments. Be sure to get the base cases right!

Hint: Write a helper procedure called `forest-accumulate`.

Question 4 (5 points):

Given below is code from Abelson and Sussman for representing a set of numbers as an *unordered* list and as an *ordered* list. The names have been changed to reflect the underlying representations.

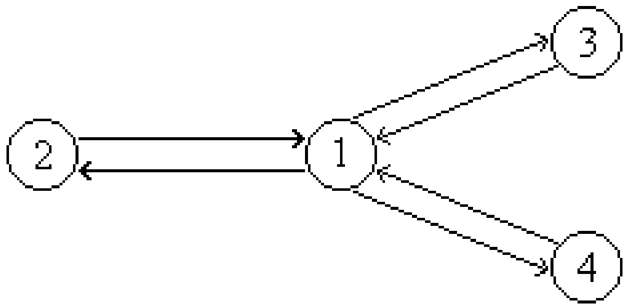
```
(define (element-of-unordered-set? x set)
  (cond ((null? set) #f)
        ((equal? x (car set)) #t)
        (else (element-of-unordered-set? x (cdr set)))))

(define (adjoin-unordered-set x set)
  (if (element-of-unordered-set? x set)
      set
      (cons x set)))

(define (element-of-ordered-set? x set)
  (cond ((null? set) #f)
        ((= x (car set)) #t)
        ((< x (car set)) #f)
        (else (element-of-ordered-set? x (cdr set)))))

(define (adjoin-ordered-set? x set)
  (cond ((null? set) (cons x set))
        ((= x (car set)) set)
        ((< x (car set)) (cons x set))
        (else (cons (car set) (adjoin-ordered-set? x (cdr set)))) ) )
```

(a) Define constructors `make-empty-unordered-set` and `make-empty-ordered-set` that use manifest types to allow distinguishing between the two set representations.

(b) Define a procedure `element-of-set?` that uses **data-directed programming** to choose the correct `element-of-...` procedure to determine if a given word is an element of a given set. That is, your procedure should still work, unchanged, if we later invent another representation for sets.

Question 5 (5 points):

This two-part question is about data-directed programming. We are going to use a list to represent a *finite state machine* (FSM) like this:



The numbered circles represent the *states* of the machine. At any moment the machine is in a particular state. The machine reads words, one letter at a time. If the machine is in some state, and sees a certain letter, it follows the arrow from its state with that letter as its label, and ends up in a new state. For example, if the machine above is in state 1 and it sees the letter B, it goes into state 3.

We'll represent a FSM by a list of all its transition arrows. The machine above has six such arrows:

```
((1 A 2) (1 B 3) (1 C 4) (2 A 1) (3 B 1) (4 C 1))
```

If the machine reads a letter for which there is no applicable arrow, it should enter state number 0. In effect, there are ``invisible'' arrows like (2 C 0) that needn't be represented explicitly in the list.

(a) Write a function `transition` that takes three arguments: a FSM list, a current state number, and a letter. The function should return the new state. For example, if `fsm` represents the list above, we should be able to do this:

```
> (transition fsm 1 'C)
4
> (transition fsm 2 'C)
0
```

(b) We want an FSM to process words, not just single letters. The machine ``reads'' the word one letter at a time. Our sample FSM, starting in state 1, should process the word AACCAAB by going through states 2, 1, 4, 1, 2, 1, and 3. The final state, 3, is the result of processing the word.

Write a function `process` that takes as its arguments a FSM, a starting state number, and a word. It should return the ending state:

```
> (process fsm 1 'AACCAAB)
3
> (process fsm 1 'AAAC)
0
```

Take a peek at the solutions