1. What will Scheme print in response to the following expressions? If an expression produces an error message, you may just write “error”; you don’t have to provide the exact text of the message. Also, draw a box and pointer diagram for the value produced by each expression.

> (append (cons (list 1 2) (list 2 3)) '(5 6))

> (let ((y (list '(1) 2 3)))
   (cons '(7 . 8) (cdr y)))

2. Draw a box and pointer diagram for the following list.

(3 (9 (2 7) 8) ()))

3. We’re going to make a new ADT called a hider. A hider provides procedures for encoding and decoding a value, along with a description.

   (define (make-hider description encoder decoder)
     (list description (cons encoder decoder)))

   (a) Write selectors hider-description, encoder, and decoder. Given a hider, they should return the appropriate value.

   (define (hider-description hider)
     ______________________________________________________)

   (define (encoder hider)
     ______________________________________________________)

   (define (decoder hider)
     ______________________________________________________)

   (b) Now, we want to test if our hiders work properly. Given a hider and a value, we say that the hider works properly if encoding and then decoding gives us back what we had originally. Write works? That takes a hider and a value and tests the hider on the value.

   (define (works? hider val)

4. Louis Reasoner has been looking over the Scheme-1 code and decides that apply-1 is doing unnecessary work. He argues that we can type (apply (lambda (x) (* x x)) '(3)) in STk, so why not take advantage of that in Scheme-1.

For reference, here is apply-1 before Louis makes his proposed change:
(define (apply-1 proc args)
  (cond ((procedure? proc)
            (apply proc args))
        ((lambda-exp? proc)
            (eval-1 (substitute (caddr proc)
                                (cadr proc)
                                args
                                '()))))

He says we can change apply-1’s body to:

(define (apply-1 proc args)
  (cond ((or (procedure? proc) (lambda-exp? proc))
            (apply proc args))
        (else (error "bad proc: " proc))))

Using Louis’ new apply-1, show what Scheme-1 would print given the following inputs. If an expression produces an error message, you may just write “error”; you don’t have to provide the exact text of the message.

Scheme-1> car
=>
Scheme-1> (* 3 7)
=>
Scheme-1> (lambda (x) (+ x 1))
=>
Scheme-1> ((lambda (x) (+ x 1)) 3)
=>

5. This question concerns the Tree abstract data type (with datum and children) discussed in lecture.

We’re going to use Trees to store words. Each datum in the tree is a letter, and each path from the root node to a leaf represents a word. For example, the tree

```
              c
             /\  
            a  d
           / \  /  
          r  p b  d
         /   /  /  
        t   r  p  b
```

represents the words cart, cap, cob, and cod. Note that this tree does not contain the word car or the word art, because a word must extend
from the root to a leaf.

Write a procedure `contains-word?` that takes such a Tree and a word, and returns `#t` if the tree contains the word, or `#f` if not.

6. Write a procedure `list-split` that takes in a list and a length, and breaks up the original list into sublists of that length. For example,

```
STk> (list-split '(a b c d e f g h) 2)
((a b) (c d) (e f) (g h))

STk> (list-split '(a b c d e f) 4)
((a b c d) (d e))

STk> (list-split '() 5)
()
```

Note that the last element of the returned value (but only the last one) may be shorter than the specified length.

**Hint:** This will be much, much, much easier if you do not try to write it iteratively! Think about meaningful helper procedures, e.g. `nth-cdr`. 