Problems

1. Mutation

(define x 1)
(set! x (cons x x))
(set! x (cons x x))
(set-cdr! (car x) x)
(set-car! (cddr x) (cadr x))
x

Draw box-and-pointer diagram for x.
2. **Trie implementation** - Used for string searching. Looks like a binary tree, but each node has up to $\Sigma$ children, where $\Sigma$ is size of the alphabet. Each child pointer is labelled with the character.

![Trie Diagram]

Figure 1: Example trie: value of key (a) is X; value of key (b a b) is Y.

In our implementation, we’ll represent a string key as a list of single-character symbols: "hello" = '(h e l l o). In order to look up a key in the trie, start at the root node and follow the appropriately labelled child pointers until you reach the end of the key. To insert a new <key, value> pair, follow key until you reach the end of the trie, then create child nodes until the key is empty, finally store the value at the last node created.

(a) Implement `make-node` which builds a trie node. A node has a value and an initially empty set of children. This should be implemented as a tagged data structure.

```scheme
(define (make-node value)
  ...)
```

(b) Implement `trie-node?` which returns #t if it is passed a trie node as input.

```scheme
(define (trie-node? x)
  ...)
```

(c) Implement `node-value` which takes a node and returns the node’s value.

```scheme
(define (node-value node)
  ...)
```
(d) Implement node-child which takes an item (a one character symbol) and a node, and returns the child of the node labelled with item.

(define (node-child item node)
(define (trie-lookup key node)
  (if (null? key)
      (node-value node)
      (let ((child (node-child (car key) node)))
        (if child
            (trie-lookup (cdr key) child)
            #f))))

(e) Implement trie-insert!, which takes a key (list of items), a value, and the root node of the trie to insert into. Subsequent trie-lookups on key should yield the value. Any intermediate nodes created should have the default value #f.

(define (trie-insert! key value node)