Bryan Kline

CS326

Homework 2

LaTex (TeXstudio)

09/29/2016

1. Consider an implementation of sets with Scheme lists. A set is an unordered collection of elements, without duplicates.

a)

Write a recursive function (is-set? L), which determines whether the list L is a set.

```
(define (is-set? L)
    (cond
        ((null? L) #t)
        ((member? (car L) (cdr L)) #f)
        (else (is-set? (cdr L)))))
```

NOTE: This function and following functions assume the member? function:

```
(define (member? x L)
    (cond
        ((null? L) #f)
        ((equal? x (car L) ) #t)
        (else (member? x (cdr L)))))
```

b)

Write a recursive function (make-set L), which returns a set built from list L by removing duplicates, if any. Remember that the order of set elements does not matter.

```
(define (make-set L)
    (cond
        ((is-set? L) L)
        (else
        (if (member? (car L) (cdr L))
        (make-set (cdr L))
        (append (list (car L)) (make-set (cdr L)))))))
```

c)

Write a recursive function (subset? A S), which determines whether the set A is a subset of the set S.

```
(define (subset? A S)
    (cond
        ((null? A) #t)
        ((member? (car A) S) (subset? (cdr A) S))
        (else #f)))
```

d)

Write a recursive function (union A B), which returns the union of sets A and B.

```
(define (union A B)
    (cond
        ((null? A) B)
        ((member? (car A) B) (union (cdr A) B))
        (else (union (cdr A) (append (list (car A)) B)))))
```

e)

Write a recursive function (intersection A B), which returns the intersection of sets A and B.

```
(define (intersection A B)
    (cond
        ((null? A) A)
        ((member? (car A) B) (cons (car A) (intersection (cdr A) B)))
        (else (intersection (cdr A) B))))
```

2. Consider an implementation of binary trees, it may be useful to define three auxiliary functions (val T), (left T) and (right T), which return the value in the root of tree T, its left subtree and its right subtree, respectively:

```
(define (val T)
    (car T))

(define (left T)
    (car (cdr T)))

(define (right T)
    (car (cdr (cdr T))))
```

a)

Write a recursive function (tree-member? V T), which determines whether V appears as an element in the tree T. The following example illustrates the use of this function:

```
(define (tree-member? v T)
    (cond
        ((and (not (null? T)) (equal? v (val T))) #t)
        ((and (not (null? T)) (< v (val T))) (tree-member? v (left T)))
        ((and (not (null? T)) (> v (val T))) (tree-member? v (right T)))
        (else (null? T) #f)))
```

b)

Write a recursive function (preorder T), which returns the list of all elements in the tree T corresponding to a preorder traversal of the tree.

```
(define (preorder T)
    (cond
        ((null? T) T)
        (else (cons (val T) (cons (preorder (left T)) (preorder (right T))))))))
```

c)

Write a recursive function (inorder T), which returns the list of all elements in the tree T corresponding to an inorder traversal of the tree.

```
(define (inorder T)
    (cond
        ((null? T) T)
        (else (append (inorder (left T)) (cons (val T) (inorder (right T)))))))
```

3. Write a recursive function (deep-delete V L), which takes as arguments a value V and a list L, and returns a list identical to L except that all occurrences of V in L or in any sublist of L have been deleted.

```
(define (deep-delete v L)
    (cond
        ((null? L) L)
        ((equal? v (car L)) (deep-delete v (cdr L)))
        ((list? (car L)) (cons (deep-delete v (car L)) (deep-delete v (cdr L))))
        (else (cons (car L) (deep-delete v (cdr L))))))
```

Extra Credit A binary search tree is a binary tree for which the value in each node is greater than or equal to all values in its left subtree, and less than all values in its right subtree. The binary tree given as example in problem 2 also qualifies as a binary search tree. Using the same list representation, write a recursive function (insert-bst V T), which returns the binary search tree that results by inserting value V into binary search tree T.

```
(define (insert-bst v T)
    (cond
        ((and (not (null? T)) (equal? v (val T))) L)
        ((and (not (null? T)) (< v (val T))) (append T (insert-bst v (left T)))
        ((and (not (null? T)) (> v (val T))) (append T (insert-bst v (right T)))
        (else (null? T) ( cons v T ))))
```

NOTE: This doesn't quite work yet, the logic is that, as in C++, inserting into a BST is the same as searching for where it should be and then just putting it there. In functional programming however there must be some way to join the parts of the tree that are coming back from deep recursive calls onto the current level of recursion so that at the top level the whole tree with the item added is returned, but I don't have time to finish it.