CSE 341 October 27, 1998

Assignment 1 Solution

1. Write and test a Scheme function pythagoras that takes the lengths of the two legs of a right triangle and returns the length of the hypotenuse. For example:

   ```
   (pythagoras 3.0 4.0)  => 5.0
   ```

   Solution:

   ```
   ;;;;;;;;;;;;;;;;;;;;
   ;;; homework 1.1 ;;;
   ;;;;;;;;;;;;;;;;;;;;


   ;; this function returns the length
   ;; of the hypotenuse of a right triangle
   ;; with sides of length b and h
   (define (pythagoras b h)
      (sqrt (+ (* b b) (* h h)))
   )
   ```

   Output:

   ```
   1 ]=> (pythagoras 1 1)

   ;Value: 1.4142135623730951

   1 ]=> (pythagoras 3.0 4.0)

   ;Value: 5.
   ```

2. Write and test a Scheme function no-duplicates that takes a list of numbers as an argument, and returns a new list which is like the argument but with duplicate elements dropped. The order of the elements in the returned list doesn't matter. For example:

   ```
   (no-duplicates '(1 3 3 8 5 3 5))  => (1 3 8 5)
   (no-duplicates '(1 3 5))  => (1 3 5)
   (no-duplicates ())  => ()
   ```

   Solution:

   ```
   ;;;;;;;;;;;;;;;;;;;;
   ;;; homework 1.2 ;;;
   ;;;;;;;;;;;;;;;;;;;;
   ```

```
;; This function takes a list and returns a new list with
;; duplicate elements removed.
(define (no-duplicates l)

  ;; base case: an empty list has no duplicated.
  ;; just return the empty list.
  (if (null? l)
      '()

      ;; get the head of the list into a local variable.
      (let ((head (car l))
            (tail (cdr l))
           )

           ;; if the first element is in the rest of the list
           ;; then don't add it now.
           (if (member head tail)
               (no-duplicates tail)

               ;; otherwise, add the first element because it
               ;; is the only time this element occures in the
               ;; list.
               (cons head (no-duplicates tail))
           )
      )
  )
)
```

Output:

```
1 ]=> (no-duplicates '(1 3 3 8 5 3 5))

;Value 3: (1 8 3 5)

1 ]=> (no-duplicates '(1 3 5))

;Value 4: (1 3 5)

1 ]=> (no-duplicates ())

;Value: ()
```

3. Write and test Scheme functions contains, add, and delete to manipulate binary search trees. contains should take a tree and an element as arguments, and return true or

false depending on whether the tree contains the element. add should take a tree and an element as arguments, and return a new tree with the element added. delete should take a tree and an element as arguments, and return a new tree with the element deleted. For both add and delete, the existing tree that is the argument should not be altered.

If you try to add an element to a tree that already contains that element, just return the tree (don't make a new tree). If you try to delete an element from a tree that doesn't contain it, signal an error using

```
(error "element not found")
```

You should choose an appropriate representation for binary trees (hint: represent a node in the tree as a list with 3 elements).

Solution:

```
;;;;;;;;;;;;;;;;;;;;
;;; homework 1.3 ;;;
;;;;;;;;;;;;;;;;;;;;


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; these are helper functions to get at a tree's data. ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; is the tree empty?
(define (tree-null? tree) (null? tree))

;; create a new tree with root data, as 'data', with left subtree 'left'
;; and right subtree 'right'
(define (tree-new data left right) (list data left right))

;; get the right subtree.
(define (tree-right node) (caddr node))

;; get the left subtree.
(define (tree-left node) (cadr node))

;; get the data from the root element of the tree.
(define (tree-data node) (car node))

;; this returns a new tree with all of the elements
;; of 'l' added to 'tree'.  (this was not required
;; it just makes constructing a tree easier)
```

```scheme
(define (tree-add-list tree l)
   (if (null? l)
       tree
       (tree-add-list (add tree (car l)) (cdr l))
   )
)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; These functions were required by the homework. ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; this returns the subtree rooted at 'val' in 'tree'
;; if 'val' is not in 'tree' this returns ()

(define (tree-subtree tree val)
  (cond
     ((tree-null? tree) '())
     ((< val (tree-data tree)) (tree-subtree (tree-left tree) val))
     ((> val (tree-data tree)) (tree-subtree (tree-right tree) val))
     (else tree)
  )
)


;; this returns #t if 'val' is in 'tree'

(define (contains tree val)
  (cond
     ((tree-null? tree) #f)
     ((< val (tree-data tree)) (contains (tree-left tree) val))
     ((> val (tree-data tree)) (contains (tree-right tree) val))
     (else #t)
  )
)




;; this adds 'val' to 'tree'
;;    * if 'val' is alread in 'tree', it returns the same tree.

(define (add tree val)
  (cond
```

```
      ;; add val here.
      ((tree-null? tree) (tree-new val '() '()))

      ;; add val to left subtree.
      ((< val (tree-data tree))
         (tree-new
            (tree-data tree)
            (add (tree-left tree) val)
            (tree-right tree)
          )
      )

      ;; add val to right subtree.
      ((> val (tree-data tree))
         (tree-new
            (tree-data tree)
            (tree-left tree)
            (add (tree-right tree) val)
          )
      )


      ; tree already has val.
      (else tree)
  )
)




;; this deletes 'val' from the tree 'tree'
;;    * it is an error if 'val' is not in 'tree'
;;    * calls helper function tree-delete-root

(define (delete tree val)
  (cond

      ;; no such element.
      ((tree-null? tree) (error "element not found"))

      ;; delete val from left subtree.
      ((< val (tree-data tree))
          (tree-new
```

```
          (tree-data tree)
          (delete (tree-left tree) val)
          (tree-right tree)
        )
     )

     ;; delete val from right subtree.
     ((> val (tree-data tree))
        (tree-new
          (tree-data tree)
          (tree-left tree)
          (delete (tree-right tree) val)
        )
     )

     ;; val found.
     (else (tree-delete-root tree))
  )
)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; these are helper functions for 'delete' ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;; this deletes the root from 'tree'
;;    * 'tree' must not be null.
;;    * this calls tree-find-min

(define (tree-delete-root tree)
  (if (tree-null? tree)

     ;; it is an error to call this function
     ;; with a null tree.
     (error "tree-delete-root called on null tree")

     ;; lets get the data, left, and right subtrees
     ;; into local variables.
     (let
       ((data (tree-data tree))
        (left (tree-left tree))
        (right (tree-right tree))
       )
```

```
       (cond

         ;; tree has no children, return
         ;; empty tree.  This line
         ;; is not necessary.  Why?
         ((and (tree-null? left) (tree-null? right)) '())


         ;; tree has no left subtree,
         ;; root deletion leaves just
         ;; right subtree.
         ((tree-null? left) right)

         ;; tree has no right subtree,
         ;; root deletion leaves just
         ;; left subtree.
         ((tree-null? right) left)

         ;; both subtrees are not null.
         ;; delete min from right subtree
         ;; and put min as root of this tree.
         (else (let* ((min (tree-find-min right))
                      (new-right (delete right min))
                     )
                     (tree-new min left new-right)
               )
         )
       )
    )
  )
)



;; this finds the minimum element in a tree.
;;    * 'tree' must not be null.

(define (tree-find-min tree)

   ;; it is an error to call this function
   ;; on an empty tree
   (if (tree-null? tree)
      (error "tree-find-min called on null tree")
```

```
        ;; get the left subtree and the data
        ;; into local variables.
        (let ((left (tree-left tree))
              (data (tree-data tree))
             )

             ;; if the left subtree is empty
             ;; then this is the minimum element.
             (if (tree-null? left)
                 data

                 ;; there are smaller elements in the
                 ;; left subtree.  recurse.
                 (tree-find-min left)
             )
        )
     )
  )
```

Correctness Checking Code:

```
;; make sure t 'contains' everything in l.
;; this returns a list of the elements #t if the
;; element is in the tree and the element number
;; if the element is missing.
(define (check-contents t l)
   (map (lambda (val)
        (if (contains t val)
            #t
            val
        )
     )
     l
   )
)

;; this prints out a message followed by an object value.
(define (print message value)
   (display ";  ")
   (display message)
   (write value)
   (newline)
)
```

```
;; initial elements to add to tree.
(define el '(11 5 2 7 15 13 19))

;; create complete tree:
;;                   11
;;               5       15
;;             2   7   13  19
(define t (tree-add-list '() el))

(newline)
(newline)

;; make sure t contains everything.
(display ";; make sure t contains everything ")
(display "-- this checks add.\n")
(print "el => " el)
(print "t => " t)
(print "(check-contents t el) => " (check-contents t el))
(newline)

;; delete 13.
(define t-13 (delete t 13))
(check-contents t-13 el)
(display ";; remove 13 from t ")
(display "-- this checks delete on a leaf\n")
(print "t-13 => " t-13)
(print "(check-contents t-13 el) => " (check-contents t-13 el))
(newline)

;; delete 15 (in addition to 13)
(define t-13-15 (delete t-13 15))
(display ";; remove 13 from t-15 ")
(display "-- this checks delete on a node with only one child\n")
(print "t-13-15 => " t-13-15)
(print "(check-contents t-13-15 el) => " (check-contents t-13-15 el))
(newline)

;; delete 5 (from original tree)
(define t-5 (delete t 5))
(display ";; remove 5 from t ")
(display "-- this checks delete on a node with two children\n")
(print "t-5 => " t-5)
```

```
(print "(check-contents t-5 el) => " (check-contents t-5 el))
(newline)

;; delete 11 (root) from original tree
(define t-11 (delete t 11))
(display ";; remove 11 from t ")
(display "-- this checks delete of root\n")
(print "t-11 => " t-11)
(print "(check-contents t-11 el) => " (check-contents t-11 el))
(newline)

;; see if t contains any old element that is not in t.
(display ";; check to see if t contains 99\n")
(print "(contains t 99) => " (contains t 99))

;; by now, we should be pretty convinced
;; that all of the tree functions work.
(newline)
(newline)
```

Output:

```
;; make sure t contains everything  -- this checks add.
;  el => (11 5 2 7 15 13 19)
;  t => (11 (5 (2 () ()) (7 () ())) (15 (13 () ()) (19 () ())))
;  (check-contents t el) => (#t #t #t #t #t #t #t)

;; remove 13 from t -- this checks delete on a leaf
;  t-13 => (11 (5 (2 () ()) (7 () ())) (15 () (19 () ())))
;  (check-contents t-13 el) => (#t #t #t #t #t 13 #t)

;; remove 13 from t-15 -- this checks delete on a node with only one child
;  t-13-15 => (11 (5 (2 () ()) (7 () ())) (19 () ()))
;  (check-contents t-13-15 el) => (#t #t #t #t 15 13 #t)

;; remove 5 from t -- this checks delete on a node with two children
;  t-5 => (11 (7 (2 () ()) ()) (15 (13 () ()) (19 () ())))
;  (check-contents t-5 el) => (#t 5 #t #t #t #t #t)

;; remove 11 from t -- this checks delete of root
;  t-11 => (13 (5 (2 () ()) (7 () ())) (15 () (19 () ())))
;  (check-contents t-11 el) => (11 #t #t #t #t #t #t)

;; check to see if t contains 99
;  (contains t 99) => ()
```