Quiz 9a

1. (3 points) What will the Scheme interpreter print in response to each of the following expressions? Also, draw a “box and pointer'' diagram for the result of each printed expression. If any expression results in an error, just write “ERROR''; you don't have to give the precise message. Hint: It'll be a lot easier if you draw the box and pointer diagram *first*!

(let ((x (list 1 2 3))

(y (list 4 (list 5) 6)))

(set-cdr! (cadr y) x)

(set-car! x (caadr y))

y)

**(4 (5 5 2 3) 6)**

**Note: (4 (5 . (5 2 3)) 6) is wrong! Scheme never prints dot followed by**

**open parenthesis.**

**1 point for what Scheme prints.**

**1 point for box and pointer diagram. (I didn’t draw it because I was too lazy. If you have questions email Pedro)**

(let ((x (list 'cs 'is 'cool))

(y (list 'x)))

(set-cdr! (cdr y) x)

y)

**ERROR**

**1 point**

2. (4 points) Implement a procedure lists->assoc! that takes a list of keys and a list of values as its arguments and, using mutation (set-car! and set-cdr!), creates a single association list.

> (define L1 (list 1 2 3))

L1

> (define L2 (list 'a 'b 'c))

L2

> (lists->assoc! L1 L2)

okay ;; return value is ignored

> L1

((1 . a) (2 . b) (3 . c))

The pairs of the first argument should become the spine pairs of the result; the second argument need not be preserved. Your implementation may **not** create any new pairs! You may assume the two arguments are always lists of equal length.

**(define (lists->assoc! l1 l2)**

**(if (null? l1)**

**'okay**

**(let ((cdr-l2 (cdr l2)))**

**(set-cdr! l2 (car l2))**

**(set-car! l2 (car l1))**

**(set-car! l1 l2)**

**(lists->assoc! (cdr l1) cdr-l2))))**

**Keep in mind that there are several ways to do this and still get the right answer. Also there are several ways of doing almost this and getting a wrong answer. For example order matters since we are doing mutation. So if the set!’s are switched around then the answer is probably wrong.**

**Also the return value (in the solution “okay”) is irrelevant.**

**-1 for switching the order of the set!ing**

**-1 for not saving the cdr of l2 (or something similar)**

**-3 for no recursion**

**-1 for no base case**

**-1 for having assoc being in l2 and not in l1**

**-3 for not using mutation**

**Take off other points for any strange syntax or logical errors as you see fit.**

3. (3 points) You are given a binary tree, in which the nodes are represented in the form

indicated by these selectors:

(define datum car)

(define left-branch cadr)

(define right-branch cddr)

The empty tree is represented by the empty list.

Write traverse!, a procedure that takes a binary tree as its argument, and rearranges the pairs to form an inorder traversal --- a linear sequence of the data from the tree, in left-to-right order. (If the tree is a binary search tree, for example, then the result will be a sorted sequence.)

A binary tree with *N* nodes contains *2N* pairs. You will string together the pairs containing the data, and discard the pairs containing the branch pointers (after you've collected the data from those branches).

Note: **Do not allocate any new pairs** in your solution. Modify the existing pairs.

Note: In this problem you are changing a structure's abstract data type, from tree to sequence. In such situations, data abstraction doesn't make much sense; just use car, set-cdr!, etc.

**The easiest solution is based on remembering that Scheme has an APPEND!**

**procedure that does an append-by-mutation:**

**(define (traverse! tree)**

**(if (null? tree)**

**'()**

**(let ((lh (traverse! (left-branch tree))))**

**(set-cdr! tree (traverse! (right-branch tree)))**

**(if (null? lh)**

**tree**

**(append! lh tree)))))**

**I've used the binary tree selectors for clarity, even though, as the exam**

**noted, this entire problem violates the idea of data abstraction by mutating**

**one data type into another.**

**If you didn't think of APPEND! you'd basically have to invent it yourself:**

**(define (traverse! tree)**

**(if (null? tree)**

**'()**

**(let ((lh (traverse! (left-branch tree))))**

**(set-cdr! tree (traverse! (right-branch tree)))**

**(if (null? lh)**

**tree**

**(begin (set-cdr! (last-pair lh) tree)**

**lh)))))**

**Scoring:**

**3 correct**

**2 has the idea**

**1 has an idea**

**0 other**

**Solutions using CONS or APPEND got zero unless they were really nice**

**in some other way, in which case one.**