University of Toronto
CSC324 – Principles of Programming Languages, Fall 2009

Assignment # 2 (Extended Due: Thursday, Oct. 10th 2009, 6:00pm sharp)

Details:
- basic programming in Scheme
- weight 8%

Silent Policy

A silent policy will take effect 24 hours before this assignment is due. This means that no question about this assignment will be answered, whether it is asked on the newsgroup, by email, or in person.

Handing in this Assignment

You should submit your work electronically. To submit files electronically, use the CDF secure website:

https://www.cdf.toronto.edu/students

or use the CDF **submit** command. Type **man submit** for more information. You should submit the following files:

a2.ss The Scheme code for your solutions.
testing.txt Your testing document (see below).
coverpage.txt An electronic cover page for your assignment.

Warning: You must follow the guidelines for electronic submission, available on your course website for assignments. Marks will be deducted for incorrect submission.

Testing

For each function that you write, you must describe the testing strategy that you used. This should include the test cases that you designed for your function, what output your function returned, and an explanation of why the test case and output are significant in verifying the correctness of the function. Read the following for a discussion of good testing practices:

http://www.cs.toronto.edu/~yilan/324f09/testing.pdf

The test file must be named as **testing.txt**. Only plain text submissions will be marked.

IMPORTANT NOTE

Since we will test your code electronically, **you must**:

1. Make certain that your code runs in Dr. Scheme on CDF, using the "Module" language.
2. Comment out any partial solutions, or code that contains errors. If your submission contains errors that prevent it from running on CDF, it will receive no marks for correctness.
3. Use the exact function names specified. See the last page of the assignment for a summary of functions that must be submitted.
4. Use the exact file name specified (a2.ss).
5. Do not display anything but the function output (no text messages to the user, fancy formatting, etc. -- just what is in the assignment handout).

A2 Bulletin Board

Important corrections (hopefully few or none) and clarifications to the assignment will be posted on the Assignment webpage, linked from your CSC324 home page. You are also responsible for monitoring the CSC324 bulletin board.

How you will be marked

Code will be marked with respect to correctness, style and documentation. For style and documentation guidelines, please see the assignment website.

Assignment # 2

[1 mark]

0. Fill out the cover sheet correctly.

Note: All the solutions for this assignment are very short. You will use recursion to solve them. Use unrestricted lambda if necessary. Make sure use choose the language in DrScheme as "Module" and click run first to make sure the correct language is loaded.

[4 marks]

1. Consider a flat list including natural numbers. Write recursive function **(minimum List)**, which takes a list and returns '() if the set is empty and the smallest number in the list otherwise. Assume that the input format of list is always correct. For example, (note that ">" is the prompt in the running interface)

```
>(minimum '())
()

>(minimum '(45 3 47 6 2 8 2))
2

>(minimum '(23 45 65 5 34 6 6))
5
```

[5 marks]

2. Write a function **(list-tail List N)** which takes a list List and an integer N and returns the remaining sublist after the first N elements of the list have been removed. When N is no more than 0, return the whole list, and when N is great than the length of the list, return empty list. Assume that the input List and integer N are of the right type and format. For example,

```
>(list-tail '(45 3 4 7 6 28) 3)
(7 6 28)

>(list-tail '((2 3) 45 65 53 (4 6 6)) 1)
(45 65 53 (4 6 6))

>(list-tail '(45 3 4 7 6 28) 0)
(45 3 4 7 6 28)

>(list-tail '(45 3 4 7 6 28) 9)
()
```

[30 marks]

3. We represent a matrix with m rows of n columns by a list of m lists each containing n elements. We assume m and n are positive integers.

So the matrix

```
1  2   3  4
5  6   7  8
9  10  11 12
```

is represented by the list of lists:

```
((1 2 3 4) (5 6 7 8) (9 10 11 12))
```

a) First, write 2 procedures:

- **(make-matrix m n lst)**
- **(unmake-matrix mat)**

(make-matrix m n lst) takes a list lst of elements and makes an m row, n column matrix out of it. You may assume lst has length m times n. Assume the elements in lst are in row-order, that is, row 1 followed by row 2, etc.

(unmake-matrix mat) takes a matrix (as specified above) and returns the list of its elements in row order, that is, the elements of row 1, followed by the elements in row 2, etc.

You may define helper/sub-functions if you like. Here are some examples:

```
>(make-matrix 2 3 '(a b c d e f))
((a b c) (d e f))

>(make-matrix 3 2 '(a b c d e f))
((a b) (c d) (e f))

>(make-matrix 0 0 '())
()

>(unmake-matrix '((a b c) (d e f)))
(a b c d e f)

>(unmake-matrix '((a) (b) (c) (d)))
(a b c d)
```

b) Now, write a procedure **(sumCol mat)** that takes an integer matrix mat and returns a list so that each element is the sum of the corresponding column. When the matrix is empty, return an empty list. Assume that the format of the

input matrix is correct and each element is an integer. Here are some examples.

```
>(sumCol '((1 2 3) (2 4 5)))
(3 6 8)

>(sumCol '((1 1)(2 3)(5 8)))
(8 12)

>(SumCol '((1) (2) (-3) (4) (5) (1)))
(10)
```

[10 marks]

4. Write a function **(appendNew LstArgs)**, which appends the elements of an undefined number of lists specified in LstArgs. For example,

```
>(appendNew '(1 2 3) '(3 5)'(8 9) '(7))
(1 2 3 3 5 8 9 7)
```

[30 marks]

5. Manipulating Binary Trees

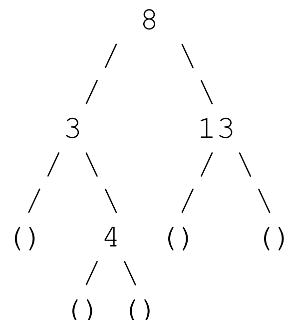
We may think of a list graphically as a tree structure. A binary tree -- as you know -- is a data structure consisting of nodes and each node has exactly one value and at most two branches, *left* and *right:*

(value left right)

In this question, we consider a binary tree of integers, whose leaves are empty lists. For example, the list

(8 (3 () (4 () ())) (13 () ()))

represents the following tree.



You should think of the list (value left right) as describing a tree rather than just a node. Thus value will be the root of the entire (sub)tree rather than just the value of the node. The value of tree (3 () (4 () ())) is the tree whose root is 3, its left branch is (), and its right branch is (4 () ()). Write some functions to manipulate binary trees, assuming that trees contain integers:

- **(binary-tree? structure)** checks whether a given structure is a binary tree defined in our question by checking whether or not each node in the tree has two branches except for leaves and each leaf in the tree is an empty list. In particular, an empty list is always a binary tree.
- **(root tree)** takes a tree and returns empty list if the tree is empty and the root otherwise.
- **(left tree)** takes a binary tree and returns its left subtree. Return empty list if the tree is empty.
- **(right tree)** takes a binary tree and returns its right subtree. Return empty list if the tree is empty.
- **(build-tree val left right)** takes a value, a left and a right subtrees and returns a tree with two branches.
- **(flip-tree tree)** swaps the left and right branches for each node of a binary tree.
- **(insert val tree)** takes an integer number val and a binary tree, with the integer value added to the bottom of the tree according to the algorithm: for each non-leaf node starting from the top, if val < node then insert into the left branch, otherwise into the right branch. See examples below.
- **(search val tree)** takes an integer value val and a tree, returns #t if val is found in the tree and #f otherwise.

Calls are, for example:

```
>(binary-tree? '3)
#f

>(binary-tree? '())
#t

>(binary-tree? '(4 5 6))
#f

>(root '(5 (3 () (7 () ())) ()))
5

>(left '(2 (3 () ()) (4 () ())))
(3 () ())

>(right (right '(2 (3 () ()) (4 (5 () ()) (7 () ())))))
(7 () ())

>(build-tree '6 '(4 () ()) '(8 (7 () ()) (9 () ())))
```

```
(6 (4 () ()) (8 (7 () ()) (9 () ())))

>(flip-tree '(6 (5 (4 () ()) ()) (7 () (9 () ()))))
(6 (7 (9 () ()) ()) (5 () (4 () ())))

>(insert '5 '())
(5 () ())

>(insert '5 '(8 (3 (2 () ()) (4 () ())) (13 (12 () ()) (14
() ()))))
(8 (3 (2 () ()) (4 () (5 () ()))) (13 (12 () ()) (14 ()
())))

>(search '12 '(8 (3 (2 () ()) (4 ()())) (13 (12 () ()) (14
() ())))))
#t
```

Summary for Submission:

Here is a summary of functions you should submit in **a2.ss**, together with parameters that each one must accept (you don't have to use the given parameter names, but you **must** use the same function names, number of parameters, and order of parameters):

- (minimum List)
- (list-tail List N)
- (make-matrix m n list)
- (unmake-matrix mat)
- (sumCol mat)
- (appendNew …) with unlimited number of arguments
- (binary-tree? structure)
- (root tree)
- (left tree)
- (right tree)
- (build-tree val left right)
- (flip-tree tree)
- (insert val tree)
- (search val tree)