

## **Assignment # 3 Part B (Monday Nov. 2nd 2009, 6:00pm sharp)**

---

**Details:**

- advanced programming in Scheme
- A3 weights 8%, this is Part B of assignment 3 and Part A has been posted

### **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. The files you should submit are:

coverpage.txt -- the cover page  
a3a.ss -- Scheme source code for questions in A3 part A  
a3b.ss -- Scheme source code for tree questions in A3 part B  
testing.txt -- testing for both A3 part A and A3 part B  
a3fp.txt or a3fp.pdf -- the formal proof question in A3 part B

*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 names specified.
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).

### **A3 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 # 3 Part B

---

The total points for this part is 35 points. A3 total points are 75 + 5 (bonus).

(Question 1 and Question 2 are in Part A)

### Question 3 [25 points]:

In the previous assignment, you saw one way to represent trees as lists of lists. Another one way of representing a tree in Scheme is as list of lists, where each of these lists is of the form

(`'node <t1> <t2>`) (where `<t1>` and `<t2>` are trees), or  
(`'leaf <v>`) (where `<v>` is any value)

Note that in this representation, only non-empty binary trees are allowed, and the empty tree cannot be represented.

We can make tree construction a little easier by defining two constructors and we can manipulate them by defining some predicates.

(a) [2 points]:

Write a recursive function to compute the height of a tree. Name this function **tree-height1**.

Write a recursive function to compute the yield of a tree (a list of the elements occurring at the leaves of the tree, from left to right). Name this function **tree-yield1**.

(b) [7 points]:

(b.1) [5 points]

We can use a macro to abstract this common pattern of comparing a tree to either a node or leaf, and then extracting children of the node or value of the leaf. Write a macro **tree-case** that will do this. For example,

```
(tree-case (node 3 4)
  ((node x y) (list x y))
  ((leaf x) (+ x 1))) => '(3 4)
(tree-case (leaf 6)
  ((node x y) (list x y))
  ((leaf x) (+ x 1))) => 7
```

(b.2) [2 points]

Write **tree-yield** and **tree-height** (recursively) using **tree-case**. Name these functions **tree-yield2** and **tree-height2**.

(c) [7 points]:

We can define a fold function on trees in a way that is similar to **foldr/foldl** for lists. The fold function can be used to encapsulate a common type of recursion for trees; namely, where the recursion can be broken up into a "thing to do for the nodes" and a "thing to do for the leaves". Thus, fold for trees should

take three parameters: a unary function that is applied to each leaf, and a binary function that is applied to the subresults at each node. For example,

```
(tree-fold + id (node (node (leaf 3) (leaf 4)) (leaf 5))) => 12
(tree-fold node leaf (node (leaf 0) (leaf 1)))
=> '(node (leaf 0) (leaf 1))
```

(c.1) [5 points] Write **tree-fold** (using recursion).

(c.2) [2 points] Write **tree-yield** and **tree-height** using **tree-fold**, name them **tree-yield3** and **tree-height3** respectively.

(d) [9 points]:

We define an expression tree to be lists of the form

```
('add <e1> <e2>) (where <e1> and <e2> are expression trees),
or
('sub <e1> <e2>) (where <e1> and <e2> are expression trees),
or
('mul <e1> <e2>) (where <e1> and <e2> are expression trees),
or
('div <e1> <e2>) (where <e1> and <e2> are expression trees),
or
('num <v>), where <v> is a number.
```

Once again, some helper functions are provided.

```
(define (add e1 e2) (list 'add e1 e2))
(define (sub e1 e2) (list 'sub e1 e2))
(define (mul e1 e2) (list 'mul e1 e2))
(define (div e1 e2) (list 'div e1 e2))
(define (num x) (list 'num x))
(define (add? t) (and (eq? (car t) 'add) (= (length t) 3)))
(define (sub? t) (and (eq? (car t) 'sub) (= (length t) 3)))
(define (mul? t) (and (eq? (car t) 'mul) (= (length t) 3)))
(define (div? t) (and (eq? (car t) 'div) (= (length t) 3)))
(define (num? t) (and (eq? (car t) 'num) (= (length t) 2)))
```

We can evaluate an expression tree by evaluating each subtree, and then performing the operation corresponding to the node. For example,

```
(eval (add (num 2) (mul (num 3) (num 4)))) => 14
(eval (sub (mul (num 2) (num 2)) (num 1))) => 3
```

(d.1) [1 point] Write the evaluation function recursively, and name it **eval1**.

We may wish to write an **exp-tree-case** macro skin to our **tree-case** macro to make this code a little cleaner. But we can do better, and write a macro that works for any kind of tree. For example,

```
(match (node 3 4)
  ((node x y) (list x y))
  ((leaf x) (+ x 1))) => (3 4)
(match (leaf 6)
  ((node x y) (list x y)))
```

```

((leaf x) (+ x 1))) => 7
(match (add (num 3) (div (num 2) (num 5)))
  ((add x y) x)
  ((div x y) y)
  ((num x) x)) => (num 3)
(match (div (num 2) (num 5))
  ((add x y) x)
  ((div x y) (list x y))
  ((num x) x)) => ((num 2) (num 5))

```

(d.2) [7 points] Write a macro **match**. You may have to use a helper macro.

(d.3) [1 point] Write `eval` recursively using `match`. Name this function **eval2**.

#### Question 4 (Formal Proofs) [10 points]:

In large-scale programming projects, it is important to document properties that are believed to hold of their sub-components. These properties not only document the intended behavior of a component, but can establish conditions on the intended context of its invocation to ensure its portability. Formally proving these properties is thus a way of verifying the correctness of a software component in all of its potential invocations. Test suites, while convenient, at best improve our confidence in a component correctness, by evaluating it on a finite collection of possible inputs.

In the case of functional programming, functional procedures can naturally be thought of as the components, and the properties that we need to prove correct often consist of algebraic equations that establish certain invariants over their possible invocations. Below, we will be looking at invariants that pertain to lists and their lengths.

Consider the following two procedures:

```

(define sum
  (lambda (lst)
    (if (null? lst) 0
        (+ (car lst) (sum (cdr lst))))))

(define mult
  (lambda (c lst)
    (if (null? lst) '()
        (cons (* c (car lst)) (mult c (cdr lst))))))

```

Your task is to prove that, if `lst` is a list of integers and `k` is an integer, then:

$$(\text{sum} (\text{mult } k \text{ lst})) = (* k (\text{sum } \text{lst}))$$

(a) [4 points]:

Examine the procedures `sum` and `mult`. From inspection of the code, write four facts (similar to the ones in the above example) that you will use in your proof.

(b) [6 points]:

Prove that if `lst` is a list of integers and `k` is an integer, then  
 $(\text{sum } (\text{mult } k \text{ lst})) = (* k (\text{sum } \text{lst}))$   
using induction. Each step of your proof must be justified with  
case, arith, IH, or one of the four facts from Part a.

**Summary for submission of a3 part B:**

For Question 4 (Formal Proofs) in part B, please save it either  
in a plain txt file named **a3fp.txt** or in a pdf format (named  
**a3fp.pdf**).

Here is a summary of the names of **functions or macros** you should  
submit in **a3b.ss**:

- tree-height1
- tree-yield1
- tree-case
- tree-height2
- tree-yield2
- tree-fold
- tree-height3
- tree-yield3
- eval1
- match
- eval2