1 Introduction

You will write some simple recursive functions on lists, prove a simple algebraic law, explore an alternative representation for binary trees, and write some simple recursive functions on binary trees.

2 Description

- Write a function `list?` that takes one argument (an arbitrary S-expression) and returns one result (a boolean). The function `list?` should return `#t` if the argument is a proper list (either the empty list or a pair whose second element is a proper list) and should return `#f` otherwise. Examples:

  ```scheme
  -> (list? 1)
  #f
  -> (list? '())
  #t
  -> (list? '(1 2))
  #t
  -> (list? (cons 1 (cons 2 '())))
  #t
  -> (list? (cons 1 2))
  #f
  ```

  *Hint:* This function mostly follows the general template of recursive functions on lists. However, the examples of recursive functions on lists have assumed that their arguments are proper lists, whereas the `list?` function can only assume that its argument is an S-expression. The `list?` function may require one or more additional base cases.

- Write a function `list-prefix?` that takes two arguments (both lists) and returns one result (a boolean). The function `list-prefix?` should return `#t` if the first argument list is a prefix of the second argument list and returns `#f` otherwise. Use `equal?` to compare elements. You may assume that the two arguments are proper lists. Examples:

  ```scheme
  -> (list-prefix? '() '(a b c))
  #t
  -> (list-prefix? '(a) '(a b c))
  #t
  -> (list-prefix? '(a b c) '(a b c))
  #t
  -> (list-prefix? '(b c) '(a b c))
  #f
  -> (list-prefix? '(a b c d) '(a b c))
  #f
  ```
• Prove the following algebraic law:

\[ \text{append-assoc law: } (\text{append } x s (\text{append } y s z s)) = (\text{append } (\text{append } x s y s) z s) \]

The proof is by structural induction on \( x s \).

- Base case: Prove \((\text{append } () (\text{append } y s z s)) = (\text{append } (\text{append } () y s) z s)\).

- Inductive step: Prove \((\text{append } (\text{cons } a s) (\text{append } y s z s)) = (\text{append } (\text{append } (\text{cons } a s) y s) z s)\)
  assuming the inductive hypothesis \((\text{append } a s (\text{append } y s z s)) = (\text{append } (\text{append } a s y s) z s)\).

Be sure to justify each step in the calculational proof; make use of the algebraic laws from Appendix C. Hint: the proof is greatly simplified by using the \text{append-empty} and \text{append-cons} laws.

• The previous problem proves that \text{append} is associative, in the sense that the list produced by \((\text{append } x s (\text{append } y s z s))\) is equivalent to the list produced by \((\text{append } (\text{append } x s y s) z s)\). However, these two expressions are not equivalent in the number of \text{cons} cells allocated. How many \text{cons} cells are allocated by \((\text{append } x s (\text{append } y s z s))\)? How many \text{cons} cells are allocated by \((\text{append } (\text{append } x s y s) z s)\)?

• Recall Prof. Ramsey’s “library” for binary trees:

```scheme
;; producer
(define make-internal-node (label left right) (list3 label left right))
;; observers
(define internal-node? (x)
  (if (pair? x)
    (if (pair? (cdr x))
      (if (pair? (cddr x))
        (null? (cdddr x))
        #f)
    #f))

(define internal-node-left (node) (cadr node))
(define internal-node-right (node) (caddr node))
(define internal-node-label (node) (car node))
(define leaf? (t) (not (internal-node? t)))
(define label (node) (if (leaf? t) t (internal-node-label t)))
```

Prof. Fluet proposes the following “library” for binary trees:

```scheme
;; creator
(val leaf (list1 'leaf))
;; producer
(define node (l x r) (list4 'node l x r))
;; observers
(define leaf? (x) (equal? x leaf))
(define node? (x) (if (list? x)
    (if (equal? (length x) 4)
      (equal? (car x) 'node)
      #f))

(define left (node) (cadr node))
(define label (node) (caddr node))
(define right (node) (caddr (cadr node))) ;; no cadddr in basis
(define btree? (x) (if (leaf? x)
    #t
    (if (node? x)
      (and (btree? (left x))
           (btree? (right x)))
      #f)))
```

2
Using Prof. Ramsey’s library, here is an example tree, built either using the creator and producer functions:

```
(val treeEx (make-internal-node 'A
  (make-internal-node 'B 'C' D)
  (make-internal-node 'E
    (make-internal-node 'F 'G 'H)
    'I)))))
```

or built with a literal S-expression:

```
(val treeEx '(A (B C D) (E (F G H) I)))
```

Using Prof. Fluet’s library, here is the same example tree, built either using the creator and producer functions:

```
(val treeEx (node (node (node leaf 'C leaf) 'B (node leaf 'D leaf))
  'A
  (node (node (node leaf 'G leaf)
    'F
    (node leaf 'H leaf))
  'E
  (node leaf 'I leaf))))
```

or built with a literal S-expression:

```
(val treeEx '(node (node (node leaf C leaf) B (node leaf D leaf))
  A
  (node (node (node leaf G leaf) F (node leaf H leaf))
  E
  (node leaf I leaf))))
```

Discuss the pros and cons of the two “libraries” for binary trees.

- Write (using Prof. Fluet’s “library” for binary trees) a function `btree->list` takes one argument (a binary tree) and returns one result (a list). The function `btree->list` should return a list containing the elements of the binary tree according to an inorder traversal. You may assume that the argument is a proper binary tree (satisfying `btree?`). Example:

  ```scheme
  -> (btree->list treeEx)
  (C B D A G F H E I)
  ```

- Write (using Prof. Fluet’s “library” for binary trees) a function `btree-height` that takes one argument (a binary tree) and returns one result (an integer). The function `btree-height` should return the height of the argument binary tree. (The height of a binary tree is the length of the longest path in the binary tree). You may assume that the argument is a proper binary tree (satisfying `btree?`). Example:

  ```scheme
  -> (btree-height treeEx)
  4
  ```

3 Requirements and Submission

You may use the reference interpreter (see Appendix A), but there may only be one active laptop in each group.

At the end of class, submit the group’s solutions either as hard-copy or by e-mail to mtf@cs.rit.edu; be sure to include the names of all group members in the submission.

A Interpreter

A reference μScheme interpreter is available on the CS Department Linux systems (e.g., glados.cs.rit.edu and queeg.cs.rit.edu and ICLs 1 and 2) at:

```
/usr/local/pub/mtf/plc/bin/uscheme
```

Use the reference interpreter to check your code.
B  μScheme Initial Basis Functions

(define and (b c) (if b c b))
(define or (b c) (if b b c))
(define not (b) (if b #f #t))

(define max (x y) (if (> x y) x y))
(define min (x y) (if (< x y) x y))

(define length (xs)
 (if (null? xs)
  0
  (+ 1 (length (cdr xs)))))

(define atom? (x) (or (number? x) (or (symbol? x) (or (boolean? x) (null? x)))))

(define equal? (s1 s2)
 (if (or (atom? s1) (atom? s2))
  (= s1 s2)
  (and (equal? (car s1) (car s2)) (equal? (cdr s1) (cdr s2)))))

(define append (xs ys)
 (if (null? xs)
  ys
  (cons (car xs) (append (cdr xs) ys))))

(define revapp (xs ys)
 (if (null? xs)
  ys
  (revapp (cdr xs) (cons (car xs) ys))))

(define reverse (xs) (revapp xs '()))

C  Algebraic Laws

* null?-empty law:  (null? '()) = #t
* null?-cons law:  (null? (cons x y)) = #f
* pair?-empty law:  (pair? '()) = #f
* pair?-cons law:  (pair? (cons x y)) = #t
* car-cons law:  (car (cons x y)) = x
* cdr-cons law:  (cdr (cons x y)) = y
* if-#t law:  (if #t x y) = x
* if-#f law:  (if #f x y) = y
* if-not law:  (if (not p) x y) = (if p y x)
* equal?-reflexive law:  (equal? x x) = #t
* equal?-symmetric law:  (equal? x y) = (equal? y x)
* length-empty law:  (length '()) = 0
* length-cons law:  (length (cons x xs)) = (+ 1 (length xs))
* length-append law:  (length (append xs ys)) = (+ (length xs) (length ys))
* length-revapp law:  (length (revapp xs ys)) = (+ (length xs) (length ys))
* append-empty law:  (append '() ys) = ys
* append-cons law:  (append (cons x xs) ys) = (cons x (append xs ys))
* revapp-empty law:  (revapp '() ys) = ys
* revapp-cons law:  (revapp (cons x xs)) = (revapp xs (cons x ys))