Programming III COMP2209

Coursework 2

Individual Coursework *(cf. collaboration policy)*

- *Aims of this coursework:*
  1. *Programming with a Scheme environment.*
  2. *Programming recursive functions on S-expressions, deep lists and graphs.*
  3. *Write purely functional solutions.*
- *Deadline: Week 10, Thursday December 8, 4pm, 4.00pm.*
- *Electronic submission: Four files:*
  1. `tree-equal.scm`*: answer to question 1.*
  2. `tree-map.scm`*: answer to question 2.*
  3. `tree-merge.scm`*: answer to question 3.*
  4. `tree-to-graph.scm`*: answer to question 4.*
  5. `tree-filter-cps.scm`*: answer to question 5.*
- *Learning outcomes:*
  1. *program in a functional style;*
  2. *practice the key mechanisms underpinning the functional programming model;*
  3. *understand the concept of functional programming and be able to write programs in this style in the context of Scheme.*
- *Scheme Language: Standard (R5RS)*
- *Weighting: 20% for each question.*

Question	*Overall Weight*	*Manual Marking*	*Automated Marking*
Question 1	*20%*	*0%*	*20%*
Question 2	*20%*	*10%*	*10%*
Question 3	*20%*	*0%*	*20%*
Question 4	*20%*	*0%*	*20%*
Question 5	*20%*	*10%*	*10%*
Total	*100%*	*20%*	*80%*

- *Automated marking: Each question will be marked* ***automatically*** *by running a test sequence based on the illustrations given in the appendices. Note that tests in the appendices are purely illustrative: a correct solution is expected to satisfy these tests; however, satisfying a test sequence is not a guarantee of a correct solution. The automated marks will be determined by the tests in the appendices and further tests to ensure that solutions are not hard-coded for a specific test sequence. Syntax of the notation used in the test sequence and details of how to run tests yourself are available from* *`https://secure.ecs.soton.ac.uk/notes/comp2209/15-16/testing.html`*
- *Marking scheme (manual marking): The manual mark componennt will be based on style (e.g. code structure and layout, choice of recursion schema, efficiency, memory usage). The breakdown is as follows.*

criterion	*Description*	*Outcomes*	*Marks*
base case	*writing the base case of a recursion on flat lists; correctness, style and efficiency*	*1,2,3*	*30%*
inductive case	*writing the inductive case of a recursion on flat lists; correctness, style and efficiency*	*1,2,3*	*70%*

In this coursework, we consider the following notion of binary tree, in which both nodes and leaves are labelled by sets of symbols.

$$\begin{array}{rcl} \langle btree \rangle & ::= & \langle leaf \rangle \mid \langle node \rangle \\ \langle leaf \rangle & ::= & \langle labels \rangle \\ \langle node \rangle & ::= & \langle labels \rangle \, \langle btree \rangle \, \langle btree \rangle \\ \langle labels \rangle & ::= & \textit{a set of symbols represented as a list} \end{array}$$

We adopt the abstract data type approach, with:

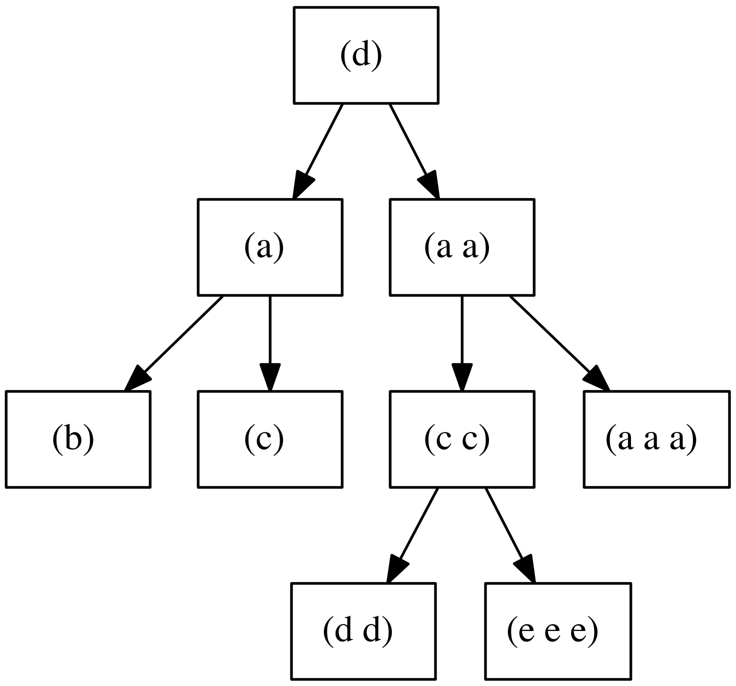
- constructors: `make-node`, `make-leaf`
- accessors: `tree-labels`, `node-left`, and `node-right`
- predicates: `leaf?`, `node?`.

To test your code, the following possible implementation of this abstract data type may be used.

```
(define the-node-tag    (list 'node))
(define the-leaf-tag    (list 'leaf))

(define node?
  (lambda (tree)
    (and (pair? tree) (eq? (tree-tag tree) the-node-tag))))

(define leaf?
  (lambda (tree)
    (and (pair? tree) (eq? (tree-tag tree) the-leaf-tag))))

(define tree-tag    car)
(define tree-labels cadr)
(define node-left   caddr)
(define node-right  cadddr)

(define make-node
  (lambda (labels left right)
    (list the-node-tag labels left right)))

(define make-leaf
  (lambda (labels)
    (list the-leaf-tag labels)))
```

The following code snippet constructs a tree that can be represented graphically as follows.

```
(make-node '(d)
           (make-node '(a) (make-leaf '(b)) (make-leaf '(c)))
           (make-node '(a a)
                      (make-node '(c c) (make-leaf '(d d)) (make-leaf '(e e e)))
                      (make-leaf '(a a a))))
```



Note: You can assume the existence of functions `set-equal?`, `set-union`, `set-minus` to process sets represented as flat lists. The definitions of these functions as well as an implementation of the abstract data type binary tree can be found in Appendix F, and will be loaded in each test suite.

1. Define a function `tree-equal?` that expects two binary trees and returns true if they are the same, and false otherwise.

   Appendix A contains illustrations of the function `tree-equal?`.

   The test suite checks several features of the solution:

   (a) Usage of imperative constructs is not permitted (tests `$check-no-set-cxr`, `$check-no-set`).

   (b) Code should work with alternative implementations of the abstract data type.

   (c) Solution should not allocate pairs (test `$calculate-cons-usage`).

2. Define a function `tree-map` that takes a function `f` and a binary tree `tree`, and returns the binary tree resulting from the application of the function `f` to all sets of labels in `tree`.

   Then, define a function `remove-labels` that takes a set of labels (expressed as a list) and a tree as argument, and removes all occurrences of these labels from all nodes and leaves of the tree.

   Appendix B contains illustrations of the functions `tree-map` and `remove-labels`.

   The test suite checks several features of the solution:

   (a) Usage of imperative constructs is not permitted (tests `$check-no-set-cxr`, `$check-no-set`).

   (b) Code should work with alternative implementations of the abstract data type.

   (c) Tests for `remove-labels`.

   (d) Sharing is not preserved by `tree-map` (see test `$check-no-sharing2`).

   (e) Memory allocation is caused by the potential application of the function and the construction of the result (e.g., tests `$calculate-cons-usage`, `$calculate-cons-usage2`).

3. First, let us define some terminology. A node n_1 in tree t_1 and a node n_2 in tree t_2 are in *corresponding positions*, if they can be reached with the same sequence of accessors (`node-left` and `node-right`). A node n_1 in tree t_1 is said to be *without correspondant* in tree t_2, if the sequence of accessors to reach n_1 in t_1 is not applicable to t_2. For example, in the tree illustrated above, the sequence `node-left`, `node-left`, `node-left` is not applicable.

   Define a function `tree-merge` that takes two binary trees and returns a new binary tree, resulting from the merge of nodes and leaves in corresponding positions in the two trees; nodes and leaves in one of the trees without correspondant in the other tree belong to the result at the corresponding position.

   Appendix C contains illustrations of the function `tree-merge`.

   The test suite checks several features of the solution:

   (a) Usage of imperative constructs is not permitted (tests `$check-no-set-cxr`, `$check-no-set`).

   (b) Code should work with alternative implementations of the abstract data type.

   (c) Memory allocation is caused by the taking unions of sets of labels and the construction of the result (tests `$calculate-cons-usage`, `$calculate-cons-usage2`, `$calculate-cons-usage3`).

4. Define a function `tree-to-graph` that takes a binary tree and returns a graph, represented as a set of vertices and edges; this set is expressed as a list.

   The definition of vertex and edge are as follows:

   $$\begin{array}{rcl} \langle \mathit{vertex} \rangle & ::= & \texttt{(vertex } \langle \mathit{id} \rangle\ \langle \mathit{labels} \rangle \texttt{)} \\ \langle \mathit{edge} \rangle & ::= & \texttt{(edge } \langle \mathit{id}_1 \rangle\ \langle \mathit{id}_2 \rangle \texttt{)} \\ \langle \mathit{id} \rangle & ::= & \text{a symbol, starting with letter } \texttt{t} \text{ followed by a succession of } \texttt{l} \text{ and } \texttt{r} \end{array}$$

   Each vertex is identified by an identifier $\langle id \rangle$ and has a set of labels. Each edge contains the identifiers of the vertices it is connecting. The first is the source of the edge, whereas the second is its destination.

Each node of a tree will be labelled by `t` (representing the top node) followed by a succession of `l` and `r`, denoting left and right accessors. Thus, the root is denoted `t`; its left child is labelled `tl`, and the right child of its right child is labelled `trr`. You may find it useful to use the following auxiliary functions to create a symbol by concatenating symbols contained in a list of symbols.

```
(define create-label
  (lambda (l)
    (string->symbol (symbol-append-reverse l))))

(define symbol-append-reverse
  (lambda (los)
    (if (null? los)
        ""
        (string-append (symbol-append-reverse (cdr los))
                       (symbol->string (car los))))))
```

Appendix D contains illustrations of the function `tree-to-graph`.

The test suite checks several features of the solution:

(a) Usage of imperative constructs is not permitted (tests `$check-no-set-cxr`, `$check-no-set`).

(b) Code should work with alternative implementations of the abstract data type.

(c) Visual test: making use of the `to-output` function (see Appendix D), the graphs should be correctly displayed, by relying on the `graphviz` program[1], which can be invoked with

```
dot -Tpdf tree1.dot > tree1.pdf
```

(d) Allocation should be no more than required for constructing the result.

5. The aim of this exercise is to practice continuation-passing style. Define a function `tree-filter-cps` that takes a predicate `pred`, a tree `tree` and a continuation `cont` and passes to the continuation two results: a tree of the same structure, containing all sets of labels satisfying the predicate and the empty list otherwise, and a tree of the same structure, containing all sets of labels not satisfying the predicate and false otherwise. The predicate is also expected to be written in continuation-passing style and expects a set of labels and a continuation.

   Appendix E contains illustrations of the function `tree-filter-cps`.

   The test suite checks several features of the solution:

   (a) Usage of imperative constructs is not permitted (tests `$check-no-set-cxr`, `$check-no-set`).

   (b) Code should work with alternative implementations of the abstract data type.

   (c) Memory allocation with `cons` is caused by constructing the results (tests `$calculate-cons-usage`, `$calculate-cons-usage-discard-init-cont`).

   (d) The initial continuation should "normally" be invoked, however, test `$calculate-cons-usage-discard-init-cont` shows that if the predicate discard its continuation, the initial continuation will be ignored.

[1]URL: http://www.graphviz.org/Download.php

A Illustrations for `tree-equal`

Test file available from https://secure.ecs.soton.ac.uk/notes/comp2209/16-17/cwk/cwk2/tree-equal-public.tst

```
(load "tree-equal.scm")
:marks 0
+++

(load "tree-implementation-list.scm")
:marks 0
+++

(instrument 'reset '* 0)
:marks 0
---


(tree-equal? (make-leaf '(a)) (make-leaf '(a)))
#t

(tree-equal? (make-leaf '(a b)) (make-leaf '(a b)))
#t

(tree-equal? (make-leaf '(a b)) (make-leaf '(b a)))
#t



(tree-equal? (make-node '(c1) (make-leaf '(a1)) (make-leaf '(b1)))
             (make-node '(c1) (make-leaf '(a1)) (make-leaf '(b1))))
#t

(tree-equal? (make-node '(c1 c) (make-leaf '(a a1)) (make-leaf '(b1 b)))
             (make-node '(c1 c) (make-leaf '(a1 a)) (make-leaf '(b b1))))
#t




(instrument 'get 'set-cxr! '_)   ;;; check that there was no mutation
$check-no-set-cxr
:marks 2
0

(instrument 'get 'set! '_)       ;;; check that there was no assignment
$check-no-set
:marks 2
0

(let ((tree (make-node '(c1) (make-leaf '(a1)) (make-leaf '(b1)))))
  (instrument-count-pairs (lambda ()
                            (tree-equal? tree tree))))

$calculate-cons-usage
:marks 4
0
```

B Illustrations for `tree-map`

Test file available from https://secure.ecs.soton.ac.uk/notes/comp2209/16-17/cwk/cwk2/tree-map-public.tst

```
(load "tree-map.scm")
:marks 0
+++

(load "tree-implementation-list.scm")
:marks 0
+++

(instrument 'reset '* 0)
:marks 0
---

;;; following test are dependent on the concrete representation


(tree-map (lambda (x) x) (make-leaf '(a)))
((leaf) (a))


(tree-map (lambda (x) x)
          (make-node '(c1) (make-leaf '(a1)) (make-leaf '(b1))))
((node) (c1) ((leaf) (a1)) ((leaf) (b1)))

(remove-labels '(a b) (make-node '(c1) (make-leaf '(a1 a)) (make-leaf '(b1 b))))
((node) (c1) ((leaf) (a1)) ((leaf) (b1)))

(define tree-equal? equal?)
:marks 0
---

;;;;; following test are independent of representation


(tree-equal? (tree-map (lambda (x) x) (make-leaf '(a)))
             (make-leaf '(a)))
#t

(tree-equal? (tree-map (lambda (x) x)
                       (make-node '(c1) (make-leaf '(a1)) (make-leaf '(b1))))
             (make-node '(c1) (make-leaf '(a1)) (make-leaf '(b1))))
#t

(tree-equal? (remove-labels '(a b)
                (make-node '(c1) (make-leaf '(a1 a)) (make-leaf '(b1 b))))
             (make-node '(c1) (make-leaf '(a1)) (make-leaf '(b1))))
#t

(instrument 'get 'set-cxr! '_)   ;;; check that there was no mutation
$check-no-set-cxr
:marks 2
0

(instrument 'get 'set! '_)       ;;; check that there was no assignment
$check-no-set
:marks 2
0

(let* ((tree1 (make-leaf '(a1)))
       (tree2 (make-node '(c1) tree1 tree1)))
  (let ((tree3 (tree-map (lambda (x) x) tree2)))
    (and (eq? (node-left tree2) (node-right tree2))
         (not (eq? (node-left tree3) (node-right tree3))))))
```

```
$check-no-sharing
:marks 2
#t


(let ((tree (make-node '(c1) (make-leaf '(a1)) (make-leaf '(b1)))))
  (instrument-count-pairs (lambda ()
                            (tree-map (lambda (x) x)
                                      tree))))

$calculate-cons-usage
:marks 4
8

(let ((tree (make-node '(c1) (make-leaf '(a1)) (make-leaf '(b1)))))
  (instrument-count-pairs (lambda ()
                            (tree-map list
                                      tree))))

$calculate-cons-usage2
:marks 4
11


(let ((tree (make-node '(c1 c2) (make-node '(a1 a2) (make-leaf '(y1 y2)) (make-leaf '(z1 z2))) (make-leaf '(b
  (- (instrument-count-pairs (lambda ()
                               (tree-map (lambda (l) (append l '())) tree)))
     (instrument-count-pairs (lambda ()
                               (tree-map (lambda (x) x) tree)))))

$calculate-cons-usage3
:marks 4
10
```

C Illustrations for `tree-merge`

Test file available from https://secure.ecs.soton.ac.uk/notes/comp2209/16-17/cwk/cwk2/tree-merge-public.tst

```
(load "tree-merge.scm")
:marks 0
+++

(load "tree-implementation-list.scm")
:marks 0
+++

(instrument 'reset '* 0)
:marks 0
---

;;; following test are dependent of the concrete representation
(tree-merge (make-leaf '(a)) (make-leaf '(b)))
((leaf) (a b))


(tree-merge (make-node '(c1) (make-leaf '(a1)) (make-leaf '(b1)))
            (make-node '(c2) (make-leaf '(a2)) (make-leaf '(b2))))
((node) (c1 c2) ((leaf) (a1 a2)) ((leaf) (b1 b2)))

(tree-merge (make-leaf '(c1))
            (make-node '(c2) (make-leaf '(a2)) (make-leaf '(b2))))
((node) (c1 c2) ((leaf) (a2)) ((leaf) (b2)))


(tree-merge (make-node '(c2) (make-leaf '(a2)) (make-leaf '(b2)))
            (make-leaf '(c1)))
((node) (c2 c1) ((leaf) (a2)) ((leaf) (b2)))


(define tree-equal? equal?)
:marks 0
---

;;;;; following test are independent of representation


(tree-equal? (tree-merge (make-leaf '(a)) (make-leaf '(b)))
             (make-leaf '(a b)))
#t


(tree-equal? (tree-merge (make-node '(c1) (make-leaf '(a1)) (make-leaf '(b1)))
                         (make-node '(c2) (make-leaf '(a2)) (make-leaf '(b2))))
             (make-node '(c1 c2) (make-leaf '(a1 a2)) (make-leaf '(b1 b2))))
#t

;;; (c1) and (c2) are in corresponding positions
;;; (a2) and (b2) are without correspondant
(tree-equal? (tree-merge (make-leaf '(c1))
                         (make-node '(c2) (make-leaf '(a2)) (make-leaf '(b2))))
             (make-node '(c1 c2) (make-leaf '(a2)) (make-leaf '(b2))))
#t


(tree-equal? (tree-merge (make-node '(c2) (make-leaf '(a2)) (make-leaf '(b2)))
                         (make-leaf '(c1)))
             (make-node '(c2 c1) (make-leaf '(a2)) (make-leaf '(b2))))
#t
```

```
(instrument 'get 'set-cxr! '_)    ;;; check that there was no mutation
$check-no-set-cxr
:marks 2
0

(instrument 'get 'set! '_)        ;;; check that there was no assignment
$check-no-set
:marks 2
0

(instrument-count-pairs (lambda ()
                          (tree-merge (make-node '(c1) (make-leaf '(a1)) (make-leaf '(b1)))
                                      (make-node '(c2) (make-leaf '(a2)) (make-leaf '(b2))))))

$calculate-cons-usage
:marks 4
27

(instrument-count-pairs (lambda ()
                          (tree-merge (make-node '(c1) (make-leaf '(a1)) (make-leaf '(b1)))
                                      (make-node '(c2 c3) (make-leaf '(a2 a3)) (make-leaf '(b2 b3))))))

$calculate-cons-usage2
:marks 4
27

(instrument-count-pairs (lambda ()
                          (tree-merge (make-node '(c1 c3) (make-leaf '(a1 a3)) (make-leaf '(b1 b3)))
                                      (make-node '(c2) (make-leaf '(a2)) (make-leaf '(b2))))))
$calculate-cons-usage3
:marks 4
30
```

D Illustrations for `tree-to-graph`

Test file available from https://secure.ecs.soton.ac.uk/notes/comp2209/16-17/cwk/cwk2/tree-to-graph-public.tst

```
(load "tree-to-graph.scm")
:marks 0
+++

(load "tree-implementation-list.scm")
:marks 0
+++


(instrument 'reset '* 0)
---


(equal? (tree-to-graph (make-leaf '(a)))
        (list '(vertex t (a))))
#t


(set-equal? (tree-to-graph (make-node '(c1)
                                      (make-leaf '(a1))
                                      (make-leaf '(b1))))
            '((vertex t (c1)) (edge t tl) (edge t tr) (vertex tl (a1)) (vertex tr (b1))))
#t


(set-equal? (tree-to-graph (make-node '(c1)
                                      (make-leaf '(a1))
                                      (make-node '(b1)
                                                 (make-leaf '(d1))
                                                 (make-leaf '(e1)))))
            '((vertex t (c1)) (edge t tl) (edge t tr) (edge tr trl) (edge tr trr)
              (vertex tl (a1)) (vertex tr (b1)) (vertex trl (d1)) (vertex trr (e1))))
#t

(instrument 'get 'set-cxr! '_)   ;;; check that there was no mutation
$check-no-set-cxr
:marks 2
0

(instrument 'get 'set! '_)       ;;; check that there was no assignment
$check-no-set
:marks 2
0

(let ((tree (make-node '(c1) (make-leaf '(a1)) (make-leaf '(b1)))))
  (instrument-count-pairs (lambda ()
                            (tree-to-graph tree))))
$calculate-cons-usage
:marks 4
22  ;;; 3 vertices * 3  (9 pairs) + 2 edges * 3 (6 pairs)
    ;;; + 5 pairs for list + 2 pairs to cons 'l and 'r to (t)

(begin
  (define to-output-port
    (lambda (port ll)
      (display "digraph \"comp2209\" { size=\"16,12\"; rankdir=\"TT\";" port)
      (newline port)
      (for-each (lambda (element)
                  (if (eq? (car element) 'vertex)
                      (begin
                        (display (cadr element) port)
                        (display " [ shape=\"None\", " port)
                        (display " label=\"" port)
                        (display (caddr element) port)
                        (display " \" ]" port))
                      (begin
                        (display (cadr element) port)
                        (display " -> " port)
                        (display (caddr element) port)
                        (display " [ label=\"" port)
                        ;;(display (cadddr element) port) ;; no label
                        (display "\" ]" port)))
                  (newline port))
                ll)
      (display "}" port)
      (newline port)))

  (define to-output
    (lambda (filename tree)
      (let ((port (open-output-file filename)))
        (to-output-port port (tree-to-graph tree))
        (close-output-port port)))))
:marks 0
---


(begin
  (to-output "tree1.dot"
             (make-node '(d)
                        (make-node '(a) (make-leaf '(b)) (make-leaf '(c)))
                        (make-node '(a a)
                                   (make-node '(c c) (make-leaf '(d d)) (make-leaf '(e e e)))
                                   (make-leaf '(a a a)))))
  (file-exists? "tree1.dot"))
:marks 2
#t

(begin
  (to-output "tree2.dot"
             (make-node '(c1) (make-leaf '(a1)) (make-leaf '(b1))))
  (file-exists? "tree2.dot"))
:marks 2
#t
```

E Illustrations for `tree-filter-cps`

Test file available from https://secure.ecs.soton.ac.uk/notes/comp2209/16-17/cwk/cwk2/tree-filter-cps-public.tst

```
(load "tree-filter-cps.scm")
:marks 0
+++

(load "tree-implementation-list.scm")
:marks 0
+++

(instrument 'reset '* 0)
:marks 0
---

;;; following test are dependent of the concrete representation

;;;;
(define member-a
  (lambda (l cont)
    (cond ((null? l) (cont #f))
          ((member (car l) '(a aa aaa aaaa aaaaa)) (cont #t))
          (else (member-a (cdr l) cont)))))
:marks 0
---


(tree-filter-cps member-a (make-leaf '(a))  (lambda (yes no) yes))
((leaf) (a))

(tree-filter-cps member-a (make-leaf '(a))  (lambda (yes no) no))
((leaf) ())


(tree-filter-cps member-a (make-node '(aa) (make-leaf '()) (make-leaf '()))  (lambda (yes no) yes))
((node) (aa) ((leaf) ()) ((leaf) ()))

(tree-filter-cps member-a (make-node '(aa) (make-leaf '()) (make-leaf '()))  (lambda (yes no) no))
((node) () ((leaf) ()) ((leaf) ()))


(tree-filter-cps member-a (make-node '(bb) (make-leaf '(a)) (make-leaf '(x)))  (lambda (yes no) yes))
((node) () ((leaf) (a)) ((leaf) ()))

(tree-filter-cps member-a (make-node '(bb) (make-leaf '(a)) (make-leaf '(x)))  (lambda (yes no) no))
((node) (bb) ((leaf) ()) ((leaf) (x)))


(tree-filter-cps member-a (make-node '(aa) (make-leaf '(a)) (make-leaf '(b)))  (lambda (yes no) yes))
((node) (aa) ((leaf) (a)) ((leaf) ()))

(tree-filter-cps member-a (make-node '(aa) (make-leaf '(a)) (make-leaf '(b)))  (lambda (yes no) no))
((node) ()   ((leaf) ())  ((leaf) (b)))


(define tree-equal? equal?)
:marks 0
---

;;;;; following test are independent of representration


(tree-equal? (tree-filter-cps member-a (make-leaf '(a))  (lambda (yes no) yes))
             (make-leaf '(a)))
#t
```

```
(tree-equal? (tree-filter-cps member-a (make-leaf '(a))  (lambda (yes no) no))
             (make-leaf '()))
#t


(tree-equal? (tree-filter-cps member-a
                              (make-node '(aa) (make-leaf '(a)) (make-leaf '(b)))
                              (lambda (yes no) yes))
             (make-node '(aa) (make-leaf '(a)) (make-leaf '())))
#t


(tree-equal? (tree-filter-cps member-a
                              (make-node '(aa) (make-leaf '(a)) (make-leaf '(b)))
                              (lambda (yes no) no))
             (make-node '() (make-leaf '()) (make-leaf '(b))))
#t


(instrument 'get 'set-cxr! '_)   ;;; check that there was no mutation
$check-no-set-cxr
:marks 2
0

(instrument 'get 'set! '_)       ;;; check that there was no assignment
$check-no-set
:marks 2
0

(begin
  (define the-node-tag    (list 'node))
  (define the-leaf-tag    (list 'leaf))

  (define make-node
    (lambda (labels left right)
      (vector the-node-tag labels left right)))
  (define make-leaf
    (lambda (labels)
      (vector the-leaf-tag labels)))
  (define node?
    (lambda (tree)
      (and (vector? tree) (eq? (tree-tag tree) the-node-tag))))

  (define leaf?
    (lambda (tree)
      (and (vector? tree) (eq? (tree-tag tree) the-leaf-tag))))

  (define tree-tag
    (lambda (vec)
      (vector-ref vec 0)))

  (define tree-labels
    (lambda (vec)
      (vector-ref vec 1)))

  (define node-left
    (lambda (vec)
      (vector-ref vec 2)))

  (define node-right
    (lambda (vec)
      (vector-ref vec 3))))

:marks 0
---
```

```
(let ((tree (make-node '(aaa) (make-leaf '(b)) (make-leaf '(a)))))
  (instrument-count-pairs (lambda ()
                            (tree-filter-cps member-a tree  cons) )))

$calculate-cons-usage
:marks 4
1



(let ((tree (make-node '(aaa) (make-leaf '(b)) (make-leaf '(a)))))
  (instrument-count-pairs (lambda ()
                            (tree-filter-cps (lambda (x cont) x) tree  cons) )))

$calculate-cons-usage-discard-init-cont
:marks 4
0

(load "tree-merge.scm")
:marks 0
+++


(let ((tree (make-node '(aa) (make-leaf '(a)) (make-leaf '(b)))))
  (tree-equal? (tree-filter-cps member-a tree  tree-merge)
               tree))
#t





;;; PRIVATE TESTS
```

F Predefined Functions

Test file available from https://secure.ecs.soton.ac.uk/notes/comp2209/16-17/cwk/cwk2/tree-implementation-list.scm

```
(define the-node-tag    (list 'node))
(define the-leaf-tag    (list 'leaf))

(define node?
  (lambda (tree)
    (and (pair? tree) (eq? (tree-tag tree) the-node-tag))))

(define leaf?
  (lambda (tree)
    (and (pair? tree) (eq? (tree-tag tree) the-leaf-tag))))

(define tree-tag    car)
(define tree-labels cadr)
(define node-left   caddr)
(define node-right  cadddr)

(define make-node
  (lambda (labels left right)
    (list the-node-tag labels left right)))

(define make-leaf
  (lambda (labels)
    (list the-leaf-tag labels)))

(define set-union
  (lambda (set1 set2)
    (cond ((null? set1) set2)
  ((memq (car set1) set2) (set-union (cdr set1) set2))
  (else (cons (car set1) (set-union (cdr set1) set2))))))

(define set-minus
  (lambda (set1 set2)
    (cond ((null? set1) '())
          ((member (car set1) set2) (set-minus (cdr set1) set2))
          (else (cons (car set1) (set-minus (cdr set1) set2))))))

(define set-equal?
  (lambda (set1 set2)
    (and (null? (set-minus set1 set2))
         (null? (set-minus set2 set1)))))
```