Programming III COMP2209

Coursework 2 (final)

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 03, 4.00pm.*
- *Electronic submission: Electronic submission only. Four files:*
  1. `convert-to-memory.scm`*: answer to question 1.*
  2. `convert-from-memory.scm`*: answer to question 2.*
  3. `tracer.scm`*: answer to question 3.*
  4. `copy.scm`*: answer to question 4.*
- *Learning outcomes:*
  1. *to program in a functional style*
  2. *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: 25% for each question.*
- *Automated marking: Sixty percent (60%) of the marks allocated to each question will be computed* ***automatically*** *by running a test sequence based on the illustrations given in appendix. Note that tests in appendix 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 appendix 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 rest of the marks 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 S-expression, deep list, and graph; correctness, style and efficiency*	*1,2,3*	*30%*
inductive case	*writing the inductive case of a recursion on S-expression, deep list, and graph;correctness, style and efficiency*	*1,2,3*	*70%*

Introduction The motivation of this coursework item is twofold. On the one hand, it is designed to develop programming skills related to recursion over trees and graphs. On the other hand, it also aims to introduce some concepts related to garbage collection, which is a technique for automated memory management; specifically, the coursework will be concerned with issues related to memory representation of data, reachability analysis, and copying collectors. This is a fairly complex domain with lots of algorithms. For an outstanding reference of the field, see [1].

A Simple Tree Data Structure For the purpose of this coursework, we consider a simplified version of S-expressions consisting of pairs and strings.

$$\begin{array}{rcl} \langle \textit{mini-Sexp} \rangle & ::= & \texttt{"string of characters"} \\ & & \mid (\ \langle \textit{mini-Sexp} \rangle\ .\ \langle \textit{mini-Sexp} \rangle\ ) \end{array}$$

Functional View of Memory Given that the coursework aims to develop functional programming skills, we will **not allow the use of assignment and mutation**. However, strictly speaking, data structure constructors do perform side-effects: indeed, invocation of a constructor such as `cons` returns a new pair, but to do so, it allocates a new section of memory. The side-effect is that after allocation, there is less memory available than before allocation. So, for the purpose of the coursework, we will represent a memory as a linked list. Allocating an object into memory will be done by `cons`ing one or more elements to the linked list (to create a representation of that object in memory).

The grammar of a memory is as follows. A memory is a (possibly empty) list of locations, each location can contain a string or a pair. A pair consists of the address of its car and cdr values in memory.

$$\begin{array}{rcl} \langle \textit{memory} \rangle & ::= & () \quad \textit{// empty memory} \\ & & \mid (\ \langle \textit{contents} \rangle\ .\ \langle \textit{memory} \rangle\ ) \\ \langle \textit{contents} \rangle & ::= & \texttt{"string of characters"} \\ & & \mid (\ \langle \textit{address} \rangle\ .\ \langle \textit{address} \rangle\ ) \\ \langle \textit{address} \rangle & ::= & \textit{a number} \end{array}$$

Our simplifying assumption is that any string (whatever its length) or any pair requires one position in memory. Note that, as we explain below, addresses are counted from the end of the list. In the grammar above, there is no restriction on addresses: when a memory location contains a pair, the addresses contained in that pair may refer to location that precede or follow the current location.

Example of Memory We illustrate a memory at three different times t_1, t_2, and t_3.
Memory at some point t_1. Addresses 0, 1, 2 respectively contain strings `"a"`, `"b"`, and `"c"`. (Note how addresses are counted from the end of the list.)

```
("c" "b" "a")
```

Memory at time t_2 after allocating string `"d"` *at position 3.* (Note again how addresses are counted from the end of the list. This technique allows us to allocate new objects at the front of the list, without the address of previously allocated objects having to change.)

```
("d" "c" "b" "a")
```

Memory at time t_3 after allocating a pair at location 4. The car position contains address 0, at which we find `"a"` and the cdr position contains address 3, at which we find `"d"`. So the external representation of the pair found at address 4 is `("a" . "d")`.

```
((cons 0 3) "d" "c" "b" "a")
```

While the representation as a list allows us to model extensible memories, it is also useful in some of our functions to have a vector representation. For this, we use the following function, which constructs a vector, after reversing the list.

```
(define make-memory
  (lambda (l)
    (apply vector (reverse l))))
```

At that point, the addresses represent the position in the vector, and they can be dereferenced in constant time.

While the context of this coursework is different than what we have studied so far in this module, techniques used in several functions seen in lectures and tutorials are relevant here: linearize, mirror with sharing, flatten, collect-nodes.

The first two questions are dealing with pure trees, providing no support for sharing and cycles. The remaining two questions have to support sharing and cycles.

1. Define a function `convert-to-memory-representation` that takes three inputs:

   - a mini-Sexpression compatible with ⟨*mini-Sexp*⟩;
   - a memory compatible with ⟨*memory*⟩;
   - the length of the memory.

   The function `convert-to-memory-representation` returns three values, packaged up in a list of three elements:

   - the address of the allocated mini-Sexpression;
   - the new memory after allocation;
   - the length of the new memory.

   The reason why we pass the length of the memory and return the length of the new memory is to avoid having to compute the length of the memory at each recursive step. (Indeed, a call to `length` is expensive since the cost of the `length` function is proportional to the length of its argument.)

   It is required that `convert-to-memory-representation` allocates data in memory following a depth-first traversal order of the mini-Sexpression.

   Appendix A contains illustrations of the function `convert-to-memory-representation`.

   The solution is not expected to deal with sharing: test `$test-sharing-is-ignored` (Appendix A) shows that the allocated data structure does not preserve the sharing of the original mini-expression.

   Likewise, the solution is not expected to deal with cycles: test `$test-cycle-does-not-terminate` (Appendix A) shows that a cycle in input results in a non-terminating function.

2. Define a function `convert-from-memory` that takes two inputs:

   - a valid address;
   - a memory represented as a vector (as returned by `make-memory`).

   The function `convert-from-memory` returns a mini-Sexpression satisfying ⟨*mini-Sexp*⟩.

   Appendix B contains illustrations of the function `convert-from-memory`.

   The solution is not expected to deal with sharing: test `$test-memory-sharing-is-ignored` (Appendix B) shows that the data structure allocated in memory exhibits some sharing, but not the resulting mini-expression.

   Likewise, the solution is not expected to deal with cycles: test `$test-memory-cycle-does-not-terminate` (Appendix B) shows that a cycle in memory results in a non-terminating program.

3. **Context:** *Given a memory and a set of roots, automatic memory management defines a notion of reachability, where roots are valid addresses for the memory. By definition, a root address is reachable. If a pair is located at a reachable address, then the address of its car and the address of its cdr are reachable. The purpose of this question is to implement this reachability functionality.*

   Define a function `tracer` that takes a list of root addresses and a memory represented as a vector (as returned by `make-memory`), and returns the list of reachable addresses.

   Appendix C contains illustrations of the function `tracer`.

   The order of addresses returned by `tracer` is not specified. For the purpose of testing, we construct a "reachability table", which is a vector, indexed by addresses, containing values 1 or 0 to indicate if an address is reachable or not, respectively.

   The function `tracer` must handle cycles: if the memory contains representations of cycles, `tracer` is not allowed to loop infinitely. It must return a solution. See test `$test-cycle1` (Appendix C).

   The function `tracer` must avoid traversing a given memory cell multiple times. Thus, the addresses contained in the list returned by `tracer` should all be different. See test `$test-sharing1` (Appendix C).

   > **Discussion** *As far as garbage collection is concerned, the reachability table we have constructed here tells us the memory addresses that contain garbage objects. A garbage collector would then "reclaim" these addresses by making them part of the "free list", which lists all the addresses available for data structure allocation. The term usually used for this type of garbage collection is called* **mark and sweep***, where the mark phase is implemented by* `tracer`*, identifying all garbage, and the sweep phase reclaim garbage objects into the free list.*
   >
   > *If we were to support the free list in our modelisation of a memory, we would have to revisit the functional memory model, since allocating a cell can no longer be done simply by consing some contents at the front of the memory list. Instead, we would have to allocate at an address contained in the free list.*
   >
   > *A recursive solution requires a stack to traverse the data structure in memory. However, memory data structures may be substantially bigger than stack space. So it is important that the tracer does not run out of stack, especially at a time when memory is scarce (this is why the garbage collector was activated). There exist techniques to write iterative functions to trace the memory, but which rely on encoding "the remainder of memory still to be traversed" in the memory itself. Likewise, there are techniques to encode the flag indicating whether an object is garbage or not; for instance, in the object representation itself. These optimisations are not in scope for the coursework!*

4. **Context:** *There is an automated memory management technique called* **copying collector** *[1]. In here, the overall system's memory is split in two. One half is used as the heap, and all allocations are taking place in this section of memory. Once this half is full, the copying collector makes a copy of the reachable data structures to the other half. The roles of the two halves is then swapped, and computation can resume. The purpose of this question is to implement this copying functionality.*

   Define a function `copy` that takes

   - a list of root addresses and
   - a memory representation as a vector (as returned by `make-memory`),

   and returns three values packaged up as a list of three elements:

   - The corresponding list of roots in the new memory space,
   - A new memory into which all reachable data has been copied,
   - The size of the new memory.

Appendix D contains illustrations of the function `copy`.

The function `copy` must handle cycles: if the memory contains representations of cycles, `copy` is not allowed to loop infinitely. It must return a solution. See tests $test-copy-cycle and $test-copy-cycle2 in Appendix D.

The function `copy` must not copy a given memory cell multiple times. See test `$test-copy-sharing` in Appendix D.

The organisation of the memory returned by `copy` is not specified. A way of handling cycles is to "copy a pair in a temporary location" of the to-space, as soon as it is encountered, before traversing its car and cdr. This way, if there is a cycle leading back to the pair, we can detect that it has already been processed.

The challenge is that at that point we do not know the addresses for the car and cdr of the temporalily allocated pair. Once the addresses of the car and cdr in the to-space have been identified, the pair in the temporary location need to be "updated". It is a requirement of the coursework that side-effects are not used. So, this update will have to be implemented functionaly over the memory representation we manipulate (see function `subst` defined in the tutorial as a suggestion to implement this functionality).

Note, similarly to the previous exercise, for the purpose of testing, we also construct a "reachability table".

> **Discussion** *A downside of a copying collector is that it requires the overall system memory to be split in two zones. So effectively, half of the memory only is effectively usable at any point in time. A benefit of a copying collector is that compacts memory into the to-space, allowing locality to be improved.*

A Illustrations for `convert-to-memory-representation`

Latest version of test file available from https://secure.ecs.soton.ac.uk/notes/comp2209/15-16/cwk/cwk2/to-memory-public.tst

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

convert-to-memory-representation*
:marks 0
***  ;;; error is expected to make sure that
     ;;; this function is not predefined in submissions

;;; to make sure I pass the right memory length, I use
;;; this auxilliary function.

(define convert-to-memory-representation*
  (lambda (mini-Sexp memory)
    (convert-to-memory-representation mini-Sexp memory (length memory))))
:marks 0
---

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

(convert-to-memory-representation* "a" '())
(0 ("a") 1)

(convert-to-memory-representation* "a" '("x" "y"))
(2 ("a" "x" "y") 3)

(convert-to-memory-representation* "x" '("x" "y"))
(2 ("x" "x" "y") 3)

(convert-to-memory-representation* (cons "a" "b") '())
(2 ((0 . 1) "b" "a") 3)

(convert-to-memory-representation* (cons (cons "x" "y") (cons "a" "b")) '())
(6 ((2 . 5) (3 . 4) "b" "a" (0 . 1) "y" "x") 7)

(let* ((pair (cons "x" "y"))
       (shared-sexp (cons pair pair)))
  (convert-to-memory-representation* shared-sexp '()))
$test-sharing-is-ignored
(6 ((2 . 5) (3 . 4) "y" "x" (0 . 1) "y" "x") 7)
  ;;; the pair ("x" . "y") has been copied at addresses 2 and 5
  ;;; Likewise for string "y" and "x"

(let* ((pair (cons "x" "y"))
       (shared-sexp (cons pair pair)))
  (set-car! shared-sexp shared-sexp) ;;; make a loop
  (convert-to-memory-representation* shared-sexp '()))
$test-cycle-does-not-terminate
oo ;;; means non terminating programme


;;; inputs not compatible with mini-Sexp are not accepted
```

```
(convert-to-memory-representation* 1234 '())
***

(convert-to-memory-representation* 'a '())
***

(convert-to-memory-representation* '() '())
***

(convert-to-memory-representation* #(1 2) '())
***

(instrument 'get 'set-cxr! '_)   ;;; check that there was no mutation in solution (the only one w
:marks 2
1

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

B Illustrations for `convert-from-memory`

Latest version of test file available from https://secure.ecs.soton.ac.uk/notes/comp2209/15-16/cwk/cwk2/from-memory-public.tst

```
(load "from-memory.scm")
:marks 0
+++

(define make-memory
  (lambda (l)
    (apply vector (reverse l))))
:marks 0
---

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

(convert-from-memory 0 (make-memory '("a")))
"a"

(convert-from-memory 0 (make-memory '(a)))
***

(convert-from-memory 0 (make-memory '(1234)))
***

(convert-from-memory 0 (make-memory '(())))
***

(convert-from-memory 2 (make-memory '("a" "x" "y")))
"a"

(convert-from-memory 1 (make-memory '("a" "x" "y")))
"x"

(convert-from-memory 2 (make-memory '((0 . 1) "b" "a")))
("a" . "b")

(convert-from-memory 6 (make-memory '((2 . 5) (3 . 4) "b" "a" (0 . 1) "y" "x")))
(("x" . "y") . ("a" . "b"))

(convert-from-memory 6 (make-memory '((2 . 2) (3 . 4) "b" "a" (0 . 1) "y" "x")))
(("x" . "y") . ("x" . "y"))


(convert-from-memory 7 (make-memory '("m" "n" (9 . 8) (0 . 1) (3 . 4) "b" "a" (6 . 7) "y" "x")))
;;; in this example, Sexp children  may be in higher addresses
("m" . "n")

(convert-from-memory 2 (make-memory '("m" "n" (9 . 8) (0 . 1) (3 . 4) "b" "a" (6 . 7) "y" "x")))
;;; in this example, Sexp children  may be in lower addresses or higher addresses
(("x" . "y") . ("m" . "n"))


(let ((Sexp (convert-from-memory 6 (make-memory '((2 . 2) (3 . 4) "b" "a" (0 . 1) "y" "x")))))
  (eq? (car Sexp) (cdr Sexp)))
    ;;; the pair at location 6 has its car and cdr fields pointing to the same address 2
```

```
$test-memory-sharing-is-ignored
:marks 3
#f

(convert-from-memory 6 (make-memory '((6 . 6) (3 . 4) "b" "a" (0 . 1) "y" "x")))
    ;;; the pair at address 6 points to itself
$test-memory-cycle-does-not-terminate
:marks 3
oo


;;; out of range, and incorrect addresses

(convert-from-memory 10 (make-memory '("a")))
***

(convert-from-memory -10 (make-memory '("a")))
***

(convert-from-memory 'a (make-memory '("a")))
***

;;; memory content not compatible with <memory> grammar are not accepted

(convert-from-memory 0 (make-memory '(1234)))
***

(convert-from-memory 0 (make-memory '(cons a b)))
***

(map (lambda (x)
       (equal? x '(("x" . "y") . ( "x" . "y"))))
     (list   (convert-from-memory 0 (make-memory '("y" "x" (2 . 3) (1 . 1))))
             (convert-from-memory 0 (make-memory '("x" "y" (3 . 2) (1 . 1))))
             (convert-from-memory 0 (make-memory '("x" (3 . 1) "y" (2 . 2))))
             (convert-from-memory 0 (make-memory '("y" (1 . 3) "x" (2 . 2))))
             (convert-from-memory 0 (make-memory '((1 . 2) "y" "x" (3 . 3))))
             (convert-from-memory 0 (make-memory '((2 . 1) "x" "y" (3 . 3))))

             (convert-from-memory 1 (make-memory '("y" "x" (0 . 0) (2 . 3))))
             (convert-from-memory 1 (make-memory '("x" "y" (0 . 0) (3 . 2))))
             (convert-from-memory 1 (make-memory '("x" (3 . 0) (2 . 2) "y")))
             (convert-from-memory 1 (make-memory '("y" (0 . 3) (2 . 2) "x")))
             (convert-from-memory 1 (make-memory '((0 . 2) "y" (3 . 3) "x")))
             (convert-from-memory 1 (make-memory '((2 . 0) "x" (3 . 3) "y")))

             (convert-from-memory 2 (make-memory '("y" (0 . 0) "x" (1 . 3))))
             (convert-from-memory 2 (make-memory '("x" (0 . 0) "y" (3 . 1))))
             (convert-from-memory 2 (make-memory '("x" (1 . 1) (3 . 0) "y")))
             (convert-from-memory 2 (make-memory '("y" (1 . 1) (0 . 3) "x")))
             (convert-from-memory 2 (make-memory '((0 . 1) (3 . 3) "y" "x")))
             (convert-from-memory 2 (make-memory '((1 . 0) (3 . 3) "x" "y")))

             (convert-from-memory 3 (make-memory '((0 . 0) "y" "x" (1 . 2))))
             (convert-from-memory 3 (make-memory '((0 . 0) "x" "y" (2 . 1))))
             (convert-from-memory 3 (make-memory '((1 . 1) "x" (2 . 0) "y")))
             (convert-from-memory 3 (make-memory '((1 . 1) "y" (0 . 2) "x")))
             (convert-from-memory 3 (make-memory '((2 . 2) (0 . 1) "y" "x")))
```

```
             (convert-from-memory 3 (make-memory '((2 . 2) (1 . 0) "x" "y")))))
(#t #t #t #t #t #t #t #t #t #t #t #t #t #t #t #t #t #t #t #t #t #t #t #t #t #t #t)


(instrument 'get 'set-cxr! '_)   ;;; check that there was no mutation
:marks 3
0

(instrument 'get 'set! '_)       ;;; check that there was no assignment
:marks 3
0
```

C Illustrations for `tracer`

Latest version of test file available from https://secure.ecs.soton.ac.uk/notes/comp2209/15-16/cwk/cwk2/tracer-public.tst

```
(load "tracer.scm")
:marks 0
+++

(define reachability-table
  (lambda (l size)
    (let ((table (make-vector size 0)))
      (for-each (lambda (address)
                  (if (and (number? address)
                           (< address size)
                           (>= address 0))
                      (vector-set! table address 1)
                      (error 'reachability-table
                             "not valid address"
                             address)))
                l)
      table)))
:marks 0
---

(define make-memory
  (lambda (l)
    (apply vector (reverse l))))
:marks 0
---



(begin
  (define size 0)
  (define mem '())
  (define mini-null "null")  ;;; reserved symbol for end of list
  (define mini-cons (lambda (x y)
                      (let ((address size))
                        (set! mem (cons (cons  x y) mem))
                        (set! size (+ size 1))
                        address)))
  (define mini-quote (lambda (x)
                       (let ((address size))
                         (set! mem (cons x mem))
                         (set! size (+ size 1))
                         address)))
  (define deref (lambda (x)
                  (list-ref mem (- (- size 1) x))))
  (define mini-null? (lambda (x)
                       (eq? (deref x) mini-null)))
  (define mini-car (lambda (x)
                     (car (deref x))))
  (define mini-cdr (lambda (x)
                     (cdr (deref x))))
  (define mini-append (lambda (l1 l2)
                        (if (mini-null? l1)
                            l2
                            (mini-cons (mini-car l1) (mini-append (mini-cdr l1) l2))))))
```

```
:marks 0
---

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

(tracer '(0) (make-memory '("a")))
(0)

(tracer '(2) (make-memory '("a" "x" "y")))
(2)

(tracer '(1) (make-memory '("a" "x" "y")))
(1)

(tracer '(1 1) (make-memory '("a" "x" "y")))
(1)

(reachability-table (tracer '(1 2) (make-memory '("a" "x" "y"))) 3)
#(0 1 1)  ;;; this solution is dependent on the order of traversal

(tracer '(1 2) (make-memory '("a" "x" "y")))
  ;;; this solution is dependent on the order of traversal, so give all possible answers
(or (1 2)
    (2 1))


(tracer '(3) (make-memory '("w" "x" (1 . 5) "z" "y" "a")))
  ;;; this solution is dependent on the order of traversal, so give all possible answers
(or (1 3 5)
    (3 1 5)
    (3 5 1)
    (1 5 3)
    (5 3 1)
    (5 1 3))

(let ((memory-list '((2 . 5) (3 . 4) "b" "a" (0 . 1) "y" "x")))
  (reachability-table (tracer '(6) (make-memory memory-list)) (length memory-list)))
#(1 1 1 1 1 1 1)

(let ((memory-list '("hi" "foo" (2 . 5) (3 . 4) "b" "a" (0 . 1) "y" "x")))
  (reachability-table (tracer '(6) (make-memory memory-list)) (length memory-list)))
#(1 1 1 1 1 1 1 0 0)

(let ((memory-list '("z" (6 . 6) (3 . 4) "b" "a" (0 . 1) "y" "x")))
       ;;; at address 6, we find a pair with car and cdr pointing to address 6
  (reachability-table (tracer '(6) (make-memory memory-list)) (length memory-list)))
  ;;; the pair at address 6 has a car and cdr pointing to itself
$test-cycle1
:marks 3
#(0 0 0 0 0 0 1 0)


(let ((memory-list '("c" (1 . 6) (3 . 4) "b" "a" (0 . 1) (6 . 1) "x")))
       ;;; at address 6, we find a pair with car and cdr pointing to address 6
  (reachability-table (tracer '(6) (make-memory memory-list)) (length memory-list)))
  ;;; the pair at address 6 has a car and cdr pointing to itself
```

```
$test-cycle2
#(0 1 0 0 0 0 1 0)

(let ((memory-list '("c" (1 . 6) (3 . 4) "b" "a" (0 . 1) (6 . 1) "x")))
       ;;; at address 6, we find a pair with car and cdr pointing to address 6
  (tracer '(6) (make-memory memory-list)))
  ;;; the pair at address 6 has a car and cdr pointing to itself
  ;;; result is order dependent
(or (1 6)
    (6 1))


(let ((memory-list '("hi" (1 . 1) (3 . 4) "b" "a" (0 . 1) "y" "x")))
  (tracer '(6) (make-memory memory-list)))
  ;;; the pair at address 6 has a car and cdr pointing address 1
$test-sharing1
;;; this solution may be dependent on the order of traversal
(or (1 6)
    (6 1))

(let ((memory-list '((5 . 5) (3 . 4) "b" "a" (0 . 1) "y" "x")))
   ;;; at address 6, we find a pair with car and cdr pointing to address 5
  (reachability-table (tracer '(6) (make-memory memory-list)) (length memory-list)))
$test-sharing2
#(0 0 0 1 1 1 1)

(let ((memory-list '((1 . 2) (3 . 4) "b" "a" (0 . 1) "y" "x")))
  (reachability-table (tracer '(6) (make-memory memory-list)) (length memory-list)))
#(1 1 1 0 0 0  1)

;;; wrong memory structure
(tracer '(0) (make-memory '(a)))
***
(tracer '(0) (make-memory '(1)))
***
;;; wrong address
(tracer '(1) (make-memory '(1)))
***


(instrument 'get 'set-cxr! '_)   ;;; check that there was no mutation
:marks 3
0

(instrument 'get 'set! '_)       ;;; check that there was no assignment
:marks 3
0


(let
  ((root  ;;; uses let to force allocation order
    (let* ((l1 (let ((a (mini-quote "a"))
                     (b (mini-quote "b"))
                     (c (mini-quote "c"))
                     (d (mini-quote "d"))
                     (n (mini-quote mini-null)))
                 (mini-cons a
                            (mini-cons b
```

```
                                    (mini-cons c
                                               (mini-cons d
                                                          n))))))
          (l2 (let ((z (mini-quote "z"))
                    (y (mini-quote "y"))
                    (x (mini-quote "x"))
                    (w (mini-quote "w"))
                    (n (mini-quote "n")))
                (mini-cons z
                           (mini-cons y
                                      (mini-cons x
                                                 (mini-cons w
                                                            n)))))))
      (mini-append l1
                   l2))))
;  (display (convert-from-memory root (make-memory mem))) (newline)
  (display (make-memory mem))
  (reachability-table (tracer (list root) (make-memory mem)) size))

#(1 1 1 1 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1)

(instrument 'get 'set! '_)       ;;; check that there was no assignment
:marks 0
44
```

D Illustrations for `copy`

Latest version of test file available from https://secure.ecs.soton.ac.uk/notes/comp2209/15-16/cwk/cwk2/copy-public.tst

```
(load "copy.scm")
:marks 0
+++

(load "tracer.scm")  ;;; Reference solution to be loaded in final testing
:marks 0
+++

(define reachability-table
  (lambda (l size)
    (let ((table (make-vector size 0)))
      (for-each (lambda (address)
                  (if (and (number? address)
                           (< address size)
                           (>= address 0))
                      (vector-set! table address 1)
                      (error 'reachability-table
                             "not valid address"
                             address)))
                l)
      table)))
:marks 0
---

(define make-memory
  (lambda (l)
    (apply vector (reverse l))))
:marks 0
---


(begin
  (define size 0)
  (define mem '())
  (define mini-null "null")  ;;; reserved symbol for end of list
  (define mini-cons (lambda (x y)
                      (let ((address size))
                        (set! mem (cons (cons  x y) mem))
                        (set! size (+ size 1))
                        address)))
  (define mini-quote (lambda (x)
                       (let ((address size))
                         (set! mem (cons x mem))
                         (set! size (+ size 1))
                         address)))
  (define deref (lambda (x)
                  (list-ref mem (- (- size 1) x))))
  (define mini-null? (lambda (x)
                       (eq? (deref x) mini-null)))
  (define mini-car (lambda (x)
                     (car (deref x))))
  (define mini-cdr (lambda (x)
                     (cdr (deref x))))
  (define mini-append (lambda (l1 l2)
```

```
                        (if (mini-null? l1)
                            l2
                            (mini-cons (mini-car l1) (mini-append (mini-cdr l1) l2))))))
:marks 0
---

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

(copy-memory '(0) (make-memory '("a")))
((0) ("a") 1)

(copy-memory '(2) (make-memory '("a" "x" "y")))
((0) ("a") 1)

(copy-memory '(1) (make-memory '("a" "x" "y")))
((0) ("x") 1)

(copy-memory '(1 1) (make-memory '("a" "x" "y")))
((0 0) ("x") 1)

(copy-memory '(1 2) (make-memory '("a" "x" "y")))
(or ((0 1) ("a" "x") 2)  ;;; this solution is dependent on the order of traversal
    ((1 0) ("x" "a") 2))

(copy-memory '(6) (make-memory '((2 . 5) (3 . 4) "b" "a" (0 . 1) "y" "x")))
:marks 0
((0) ("b" "a" (5 . 6) "y" "x" (2 . 3) (1 . 4)) 7) ;;; this solution is dependent on the order of

(copy-memory '(5) (make-memory '((2 . 5) (5 . 5) "b" "a" (0 . 1) "y" "x")))
$test-copy-cycle
((0) ((0 . 0)) 1)


(copy-memory '(6) (make-memory '((6 . 5) (5 . 6) "b" "a" (0 . 1) "y" "x")))
$test-copy-cycle2
(or ((0) ((1 . 0) (0 . 1)) 2)
    ((1) ((1 . 0) (0 . 1)) 2))

(copy-memory '(6) (make-memory '((2 . 2) (3 . 4) "b" "a" (0 . 1) "y" "x")))
$test-copy-sharing ;;; this solution is dependent on the order of traversal
(or ((0) ("y" "x" (2 . 3) (1 . 1)) 4)
    ((0) ("x" "y" (3 . 2) (1 . 1)) 4)
    ((0) ("x" (3 . 1) "y" (2 . 2)) 4)
    ((0) ("y" (1 . 3) "x" (2 . 2)) 4)
    ((0) ((1 . 2) "y" "x" (3 . 3)) 4)
    ((0) ((2 . 1) "x" "y" (3 . 3)) 4)

    ((1) ("y" "x" (0 . 0) (2 . 3)) 4)
    ((1) ("x" "y" (0 . 0) (3 . 2)) 4)
    ((1) ("x" (3 . 0) (2 . 2) "y") 4)
    ((1) ("y" (0 . 3) (2 . 2) "x") 4)
    ((1) ((0 . 2) "y" (3 . 3) "x") 4)
    ((1) ((2 . 0) "x" (3 . 3) "y") 4)

    ((2) ("y" (0 . 0) "x" (1 . 3)) 4)
    ((2) ("x" (0 . 0) "y" (3 . 1)) 4)
```

```
    ((2) ("x" (1 . 1) (3 . 0) "y") 4)
    ((2) ("y" (1 . 1) (0 . 3) "x") 4)
    ((2) ((0 . 1) (3 . 3) "y" "x") 4)
    ((2) ((1 . 0) (3 . 3) "x" "y") 4)

    ((3) ((0 . 0) "y" "x" (1 . 2)) 4)
    ((3) ((0 . 0) "x" "y" (2 . 1)) 4)
    ((3) ((1 . 1) "x" (2 . 0) "y") 4)
    ((3) ((1 . 1) "y" (0 . 2) "x") 4)
    ((3) ((2 . 2) (0 . 1) "y" "x") 4)
    ((3) ((2 . 2) (1 . 0) "x" "y") 4)

)

(let* ((memory-list '((2 . 5) (3 . 4) "b" "a" (0 . 1) "y" "x"))
       (triple (copy-memory '(6) (make-memory memory-list)))
       (to-roots  (car triple))
       (to-memory (cadr triple))
       (to-size   (caddr triple)))
  (reachability-table (tracer to-roots (make-memory to-memory)) (length to-memory)))
#(1 1 1 1 1 1 1)

(let* ((memory-list '(hi foo (2 . 5) (3 . 4) "b" "a" (0 . 1) "y" "x"))
       (triple (copy-memory '(6) (make-memory memory-list)))
       (to-roots  (car triple))
       (to-memory (cadr triple))
       (to-size   (caddr triple)))
  (reachability-table (tracer to-roots (make-memory to-memory)) (length to-memory)))
#(1 1 1 1 1 1 1)

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

(instrument 'get 'set! '_)       ;;; check that there was no assignment
0

(let ((root (mini-append (mini-cons (mini-quote "a")
                                    (mini-cons (mini-quote "b")
                                               (mini-cons (mini-quote "c")
                                                          (mini-cons (mini-quote "d")
                                                                     (mini-quote mini-null)))))
                         (mini-cons (mini-quote "z")
                                    (mini-cons (mini-quote "y")
                                               (mini-cons (mini-quote "x")
                                                          (mini-cons (mini-quote "w")
                                                                     (mini-quote mini-null))))))))
;  (display (convert-from-memory root (make-memory mem))) (newline)
  (display (make-memory mem))
  (let* ((triple (copy-memory (list root) (make-memory mem)))
         (to-roots  (car triple))
         (to-memory (cadr triple))
         (to-size   (caddr triple)))
    (reachability-table (tracer to-roots (make-memory to-memory)) to-size)))
#(1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1)
```

References

[1] Richard Jones and Rafael Lins. *Garbage Collection. Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.