# Homework 5

Carl Eastlund

Due **Mon., Apr. 8** at **9:00pm**.

**Collaboration Policy:** Your work on this assignment must be your own. You *may not* copy files from other students in this class, from people outside of the class, from the internet, or from any other source. You *may not* share files with other students in this class.

You *may* discuss the problems, concepts, and general techniques used in this assignment with other students, so long as you do not share actual solutions.

If you are in doubt about what you *may* and *may not* do, ask the course instructor before proceeding. If you violate the collaboration policy, you will receive a zero as your grade for this entire assignment and you will be reported to OSCCR (`northeastern.edu/osccr`).

## 1 Instructions

As usual, provide the code for your solution in `solution.rkt`. There are no explicit mathematical arguments needed for this assignment; you do not need to supply any LaTeX or PDF files.

For this assignment, you can use any built-in datatypes and libraries that come with Racket—hash tables, queues, for loops, etc.—so long as you are able to reason about their time and space complexity. You may not use any external, third-party libraries.

## 2 Data Definition: Data as Graph Elements

The data in any computer program forms a graph. Every allocated value in memory is a vertex; every reference from one value to another is an edge. Technically this is a *multigraph*, as there can be duplicate edges. For instance, in `(vector '() '())`, there are two edges from the vector to the empty list. In this assignment we make the data-as-graph representation explicit. Here is our data definition:

```
;; Memory is a (Vectorof Value)
;; A Value is one of:
;; - a String
;; - a (Vectorof Pointer)
;; A Pointer is (box Natural-Number)
;;   where the value in the box is an index into the Memory.
```

For instance, the following program constructs several values:

```
(define new-york (vector "new" "york"))
(define nyc (vector new-york "comma" new-york))
(define up-state (vector "buffalo" new-york))
```

In our representation of memory, we might represent these values as:

```
(define memory1
  (vector
    ;; index 0:
    "new"
    ;; index 1:
    "york"
    ;; index 2, the definition of new-york:
    (vector (box 0) (box 1))
    ;; index 3:
    "comma"
    ;; index 4, the definition of nyc:
    (vector (box 2) (box 3) (box 2))
    ;; index 5:
    "buffalo"
    ;; index 6, the definition of up-state:
    (vector (box 5) (box 2))))
```

# 3 Problem 1: Garbage Collection

```
(collect Memory (Listof Natural-Number)) : (list Memory (Listof Natural-Number))
```

Write the function `collect` whose type is described above. Its inputs are a `Memory` and a list of *root* indices: the locations of values that must be kept. The function `collect` must create a new `Memory` containing these locations and any others that are reachable from them, but no others. The result is a two-element list containing the new `Memory` and updated locations for the root indices.

For instance, we might run `collect` on our original data example:

```
(collect memory1 (list 2 6))
```

The program needs the values of `new-york` and `up-state`, but not `nyc`. One possible, correct result is:

```
(list
  (vector
    ;; index 0:
    "'new"
    ;; index 1:
    "york"
    ;; index 2, the definition of new-york:
    (vector (box 0) (box 1))
    ;; index 3:
    "buffalo"
    ;; index 4, the definition of up-state:
    (vector (box 3) (box 2)))
  (list 2 4))
```

Note that some of the values move to new positions; every `Pointer` to them changes accordingly, and the root index for `up-state` changes from 6 to 4. It does not matter which values move or what order they appear in; your solution only needs to make sure the right values are in the `Memory` and all the `Pointers` and root indices "follow" the moving indices properly.

Your solution will be graded on its asymptotic running time; the faster, the better.

# 4 Problem 2: Rendering Data

`(snapshot Memory Natural-Number) : (list (Listof Defn) Expr)`

Write the function `snapshot` whose type is described above. Its output type uses additional data definitions:

```
;; A Defn is either:
;; - (list 'define Symbol Expr)
;; - (list 'vector-set! Expr Expr Expr)
;; An Expr is one of:
;; - String
;; - Number
;; - (list 'quote '())
;; - Symbol
;; - (list 'cons Expr Expr)
;; - (cons 'vector (Listof Expr))
```

The function `snapshot` creates a program that reconstructs a value in memory, even if that value contains cyclic self-references. Every location reachable from the given index gets its own `define` in the generated program. These definitions should be topologically sorted so that every reference to a defined variable comes after the definition. Wherever cyclic references make this impossible, fill in a dummy value for the undefined pointers initially, then use `vector-set!` to fix them when the cycle is complete.

You may want to use `gensym` to create unique symbol names, although you are free to create your own using `string->symbol` or `format-symbol` as well.

For example, the following `Memory` represents a cyclic pair of vectors:

```
(define cyclic
  (vector
    (vector (box 1) (box 2))
    "Jacob's"
    (vector (box 3) (box 0))
    "ladder"))
```

Here is one possible, correct program that recreates the value at index 0:

```
(define v0 "Jacob's")
(define v1 "ladder")
(define v2 (vector v0 ""))
(define v3 (vector "ladder" v2))
(vector-set! v2 1 v3)
v2
```

The corresponding result for `(snapshot cyclic 0)` would be:

```
(list
  (list
    (list 'define 'v0 "Jacob's")
    (list 'define 'v1 "ladder")
    (list 'define 'v2 (list 'vector 'v0 ""))
    (list 'define 'v3 (list 'vector "ladder" 'v2))
    (list 'vector-set! 'v2 1 'v3))
  'v2)
```

Your solution will be graded on its asymptotic running time; the faster, the better.