9 Mutation

Exercise 9.1. Devise a Scheme expression that could have four possible values, depending on the order in which application subexpressions are evaluated.

Solution.

Exercise 9.2. Draw the environment that results after evaluating:

```
> (define alpha 0)
> (define beta 1)
> (define update-beta! (lambda () (set! beta (+ alpha 1))))
> (set! alpha 3)
> (update-beta!)
> (set! alpha 4)
```

Solution.

Exercise 9.3. Draw the environment that results after evaluating the following expressions, and explain what the value of the final expression is. (Hint: first, rewrite the let expression as an application.)

```
> (define (make-applier proc) (lambda (x) (proc x))
> (define p (make-applier (lambda (x) (let ((x 2)) x))))
> (p 4)
```

Solution.

Exercise 9.4. What is the value of (*mlist-length pair*) for the pair shown in Figure 9.5?

Solution.

Exercise 9.5. [⋆] Define a *mpair-circular?* procedure that takes a MutablePair as its input and outputs true when the input contains a cycle and false otherwise.

Solution.

Exercise 9.6. Define an imperative-style procedure, *mlist-inc!* that takes as input a MutableList of Numbers and modifies the list by adding one to the value of each element in the list.

Solution.

Exercise 9.7. [⋆] Define a procedure *mlist-truncate!* that takes as input a MutableList and modifies the list by removing the last element in the list. Specify carefully the requirements for the input list to your procedure.

Solution.

Exercise 9.8. [⋆] Define a procedure *mlist-make-circular!* that takes as input a MutableList and modifies the list to be a circular list containing all the elements in the original list. For example, (*mlist-make-circular!* (*mlist* 3)) should produce the same structure as the circular pair shown in Figure 9.5.

Solution.

Exercise 9.9. [⋆] Define an imperative-style procedure, *mlist-reverse!*, that reverses the elements of a list. Is it possible to implement a *mlist-reverse!* procedure that is asymptotically faster than the *list-reverse* procedure from Example 5.4?

Solution.

Exercise 9.10. [⋆⋆] Define a procedure *mlist-aliases?* that takes as input two mutable lists and outputs true if and only if there are any mcons cells shared between the two lists.

Solution.

Exercise 9.11. Define the *mlist-map!* example from the previous section using *while*.

Solution.

Exercise 9.12. Another common imperative programming structure is a repeat-until—textbf*repeat-*
repeat-until *until* loop. Define a *repeat-until* procedure that takes two inputs, a body procedure and a test procedure. The procedure should evaluate the body procedure repeatedly, until the test procedure evaluates to a true value. For example, using *repeat-until* we could define *factorial* as:

```
(define (factorial n)
  (let ((fact 1))
    (repeat-until
     (lambda () (set! fact (* fact n)) (set! n (- n 1)))
     (lambda () (< n 1)))
    fact))
```

Solution.

Exercise 9.13. [⋆⋆] Improve the efficiency of the indexing procedures from Section 8.2.3 by using mutation. Start by defining a mutable binary tree abstraction, and then use this and the MutableList data type to implement an imperative-style *insert-into-index!* procedure that mutates the input index by adding a new word-position pair to it. Then, define an efficient *merge-index!* procedure that takes two mutable indexes as its inputs and modifies the first index to incorporate all word occurrences in the second index. Analyze the impact of your changes on the running time of indexing a collection of documents.

Solution.