1. Environmental Diagrams

Exercise 1. Draw the environmental diagrams and show the return values:

\[
\text{(define } x 3)\]

\[
\text{(define baz}
\begin{array}{l}
\text{ (let } ((x 100))
\text{ (f (lambda (y) (+ x y)))}
\text{ (lambda (x) (f (* x x)))))}
\end{array} \]

\[
\text{(baz 7)}
\]

Answer: 52

Note: for environment diagrams, it’s best to step through the process in EnvDraw.

\[
\text{(define x 'old-x)}
\]
\[
\text{(define y 'old-y)}
\]

\[
\text{((lambda () (define y 6)}
\begin{array}{l}
\text{ (set! x 2))}
\end{array} \]

\[
\text{x}
\]
\[
\text{y}
\]

Answer: x is 2, y is 'old-y
Exercise 2. Fill in the blanks with the response to the indicated expressions. The answers are 11, 121, 1001, 1111 but not necessarily in that order!

(HINT: You shouldn’t have to draw environmental diagrams to figure this out. In each case, ask yourself: is a a class variable or an instance variable? Is b a class variable or an instance variable?)

(define make-foo1
  (let ((a 1))
    (lambda ()
      (let ((b 1))
        (lambda ()
          (let ((a (* a 10))
                (b (+ b a)))
            b)))))

(define foo1-1 (make-foo1))
(define foo1-2 (make-foo1))

(foo1-1)
(foo1-1)
(foo1-2) ==> ?

Answer: 1001

(define make-foo2
  (let ((a 1)
        (b 1))
    (lambda ()
      (let ((a (* a 10))
            (b (+ b a)))
        b)))))

(define foo2-1 (make-foo2))
(define foo2-2 (make-foo2))

(foo2-1)
(foo2-1)
(foo2-2) ==> ?

Answer: 1111
(define make-foo3
  (let ((b 1)
        (lambda ()
          (let ((a 1))
            (lambda ()
              (set! a (* a 10))
              (set! b (+ b a))
              b))))))
(define foo3-1 (make-foo3))
(define foo3-2 (make-foo3))
(foo3-1)
(foo3-1)
(foo3-2) ==> ?

Answer: 121

(define make-foo4
  (lambda ()
    (let ((a 1)
           (b 1))
      (lambda ()
        (set! a (* a 10))
        (set! b (+ b a))
        b))))
(define foo4-1 (make-foo4))
(define foo4-2 (make-foo4))
(foo4-1)
(foo4-1)
(foo4-2) ==> ?

Answer: 11
Exercise 3.
(define f (cons 1 2))
(define g (list f f))
(set! f 3)
(set-car! (car g) 4)
g
Answer: ((4 . 2) (4 . 2))

(define f (list 2 3))
(define g (append f f))
(set-car! g f)
g
Answer: ((2 3) 3 2 3)

(let ((x (list 1 2 3)))
  (set-car! x (list 'a 'b 'c))
  (set-car! (cdar x) 'd)
  x)
Answer: ((a d c) 2 3)
Exercise 4. Write a procedure \texttt{link-first!} that takes a non-empty list of numbers and destructively links together all its elements with the same value as the car of the list. For example:

\begin{verbatim}
STk> (define ls '(1 2 3 2 1 2 1))
STk> (link-first! ls)
STk> ls
\end{verbatim}

\begin{verbatim}
(1 1 1)
DO NOT create any additional pairs!
\end{verbatim}

\begin{verbatim}
(define (link-first! ls)
  (cond ((null? (cdr ls)) '())
        ((= (car ls) (cadr ls)) (link-first! (cdr ls)))
        (else (set-cdr! ls (cddr ls))
       (link-first! ls)))))
\end{verbatim}

4. Vectors

Exercise 5. Write a procedure \texttt{subvector} that takes three arguments: a vector \texttt{vec} and two non-negative integers \texttt{start} and \texttt{end} that are less than the length of \texttt{vec}. It should return a new vector containing only the elements of \texttt{vec} at positions between \texttt{start} and \texttt{end} inclusive. For example:

\begin{verbatim}
STk> (subvector #(we all live in a yellow submarine) 2 5)
\end{verbatim}

\begin{verbatim}
#(live in a yellow)
\end{verbatim}

If \texttt{end} is less than \texttt{start}, return an empty vector.

\begin{verbatim}
(define (subvector vec start end)
  (let ((new-vec (make-vector (+ (- end start) 1))))
    (define (loop index)
      (if (= index (vector-length new-vec))
          new-vec
        (begin (vector-set! new-vec index (vector-ref vec (+ index start)))
               (loop (+ index 1))))))
  (loop 0)))
\end{verbatim}
5. Streams

**Exercise 6.** What are the first 10 elements of the following stream?

```scheme
(define s
  (cons-stream 1
               (interleave (stream-cdr integers) s)))
```

Answer: (1 2 1 3 2 4 1 5 3 6 ...)

**Exercise 7.** Implicitly define the following stream without defining any procedures. You may use the stream `ones` and `integers`, and common stream operators.

```
( () (()) (((()))) ((((())()) ((((()())))))) (((((())()))) ... )
```

```scheme
(define s1 (cons-stream '()
                      (stream-map list s1)))
```

```
( 0 3 8 15 24 35 48 63 80 99 ... )
```

```scheme
(define s2 (stream-map - (stream-map * integers integers) ones))
```

6. Concurrency

**Exercise 8.** Analyze the following code and write down all possible answers:

```scheme
(define x 3)
(parallel-execute (lambda () (set! x (+ (* x x) 1)))
                   (lambda () (let ((y x))
                                 (set! x (+ y y))))
```

Answer: 20, 37, 10, 6, 19