Homework 9: Mutation

Due:

All the directions from homework 2 apply to this homework. In particular, remember to hand in a printout for procedures, and either a calculation or a transcript showing your testing.

We highly recommend that you put each problem in a separate file, and when making a transcript use the Scheme procedure `load` to read that file in to the Scheme interpreter.

Note that no makeups will be accepted after May 5; so you may not be able to take or makeup some incompletes in this chapter.

1 Section 11.2, Assignment and State

1. (0 points; suggested practice) What are the results of the following expressions? Why does Scheme give those results?

```
(let ((x 2))       ;; (a)
  (writeln x)
  (let ((x 'hmm))
    (writeln x) (set! x 'inside) (writeln x))
  (writeln x))

(letrec ((du (lambda (x)  ;; (b)
               (writeln x)
               (if (not (zero? x))
                   (begin
                     (set! x (sub1 x))
                     (du x)
                     (writeln x))))))
  (du 4))
```

2. (0 points; suggested practice) What happens when you give Scheme the following?

```
(begin
  (define foo 3)
  (writeln foo)
  (let ((bar 4))
    (define foo 5)
    (writeln foo))
  (writeln foo))
```

   What does this tell you about the difference between `define` and `set!`?

1.1 Imperative-Style Programming with While Loops and Set!

In this section you will write while loop versions of procedures. A "while loop version" should avoid the use of `vector-generator`, and instead use `make-vector`, `vector-set!`, `set!`, and `while-proc` (program 11.8). For example, the following procedure is a while loop version of program 9.5, on page 272.

```
(load "/home/cs227/lib/ch11.ss")
(define while while-proc)  ; rename program 11.8 on page 353

(define vector-stretch-while
  ; TYPE: vector, integer -> vector
  (lambda (vec new-size)
    ; ENSURES: result is a new copy of vec, but having size new-size,
    ; result filled with () if bigger
    (let ((size (vector-length vec))
          (res (make-vector new-size))
          (i 0))
      (while (lambda () (< i new-size))
             ; INVARIANT: i <= new-size and for all 0 <= j < i,
             ; the jth element of res is correctly filled in
             (lambda ()
               (if (< i size)
                   (vector-set! res i (vector-ref vec i))
                   (vector-set! res i '()))
               (set! i (add1 i))))
      res)))
```

3. (5 points; basic) Write a while loop version of the `successive-powers` procedure, in exercise 9.1.

4. (10 points; basic) Write a while loop version of `multiply-by-scalar`. (Hint, you can do this by writing a while loop version of `vector-map`.) For example,

```
> (define my-vec (vector 1 2 3))
> my-vec
#(1 2 3)
> (multiply-by-scalar 7 my-vec)
#(7 14 21)
> my-vec
#(1 2 3)
```

5. (5 points) Write a while loop version of `vector-generator` (see program 9.21).

6. (10 points; extra credit only) Write a while loop version of `matrix-product` (see program 9.38).

7. (10 points) Write a while loop version of program 5.4 on page 139. That is, instead of using the letrec and tail recursion, use a call to `while-proc` and use `set!`. Test your program to see that it correctly computes factorials.

8. (10 points) Write a while loop version of program 4.24 on page 124. (Hint, you will need two variables.)

9. (10 points) You may have noticed that many of the while loop procedures, especially those that work with vectors share a similar structure. This looping structure is called a "for loop", because in many programming languages there is a statement called `for` that does this. Write a procedure `from-to` that takes two numbers `from` and `to` and returns a procedure that takes a procedure `one-step` (of type `number -> void`), and which returns nothing. The call `((from-to` n m`)` p`)` should execute, in order, `(`p n`)`, `(`p $n+1$`)`, ..., `(`p m`)`. For example,

```
> ((from-to 0 4) (lambda (x) (display x) (display ", ")))
0, 1, 2, 3, 4, > ((from-to 4 4) (lambda (x) (writeln (* x x))))
```

```
16
> ((from-to 4 0) (lambda (x) (writeln "you goofed")))
>
```

10. (0 points; suggested practice) Rewrite your while loop programs using `from-to`.

11. (10 points) Write `vector-generator` using `from-to`. Be sure to test it and check that it is giving the exact number of elements.

12. (10 points; extra credit only) Write a procedure `from-to-by` that takes three numbers `from`, `to`, and `by`, and returns a procedure that takes a procedure `one-step` (of type `number -> void`), and which returns nothing. The call `((from-to-by` n m b`)` p`)` should execute, in order, `(`p n`)`, `(`p $n+b$`)`, ..., `(`p m`)`, if $m = n + k \times b$ for some natural number k. If $m \neq n + k \times b$ for some natural number k, then the last call of p should be `(`p $n + k' \times b$`)`, where k' is the smallest natural number such that if b is positive, then $n + (k'+1) \times b$ is larger than m, or if b is negative, then $n + (k'+1) \times b$ is smaller than m. Note that `(from-to-by` n m `1)` is equivalent to `(from-to` n m`)`. For example,

```
> ((from-to-by 4 0 -1) (lambda (x) (display x) (display ", ")))
4, 3, 2, 1, 0, > ((from-to-by 4 4 3) (lambda (x) (writeln (* x x))))
16
> ((from-to-by 4 12 3) (lambda (x) (display x) (display ", ")))
4, 7, 10, >
```

13. (7 points; extra credit only) Do exercise 11.9.

14. (0 points; suggested practice only) Do exercise 11.10. Check your answer on the computer.

15. (20 points; extra credit only) Rewrite the `display-tower-of-hanoi` program, using `set!`, `while-proc`, and 3 variables: `left`, `center`, and `right`, to hold the number of disks on the corresponding post. After each move print the state in a form such as:

```
left: 3 center: 0 right: 2
```

So that the output is a series of lines of this form. For example, `(display-tower-of-hanoi 2)` would print the following.

```
left: 2 center: 0 right: 0
left: 1 center: 1 right: 0
left: 0 center: 1 right: 1
left: 0 center: 0 right: 2
```

1.2 Memoization

In this section you can use either a functional or an imperative style as you see fit.

16. (0 points; suggested practice) Do exercise 11.1 on page 349. This is to show how memoizing works.

17. (6 points) Do exercise 11.2.

18. (4 points) Do exercise 11.3. The procedure timer (program 10.19) is found in `/home/cs227/lib/timer.ss`. If you use some other Scheme instead of Chez Scheme, see the comments in the file.

19. (12 points; extra credit only) Do exercise 11.4.

20. (10 points) Do exercise 11.5.

21. (10 points) Do exercise 11.6.

22. (12 points) Do exercise 11.7.