CS220 Programming Principles

Final Exam (June 20, 2018)

I certify that my answers to this exam are all my own work.

(Name) ______________________________ (ID No.)______________________

(Signature) ____

Question	Out of	Marks
1	**20**	
2	**25**	
3	**25**	
4	**25**	
5	**20**	
6	**35**	
7	**20**	
8	**40**	
9	**20**	
10	**20**	
Total	**250**	

When writing procedures, write a straightforward code. Do not try to make your program slightly more efficient at the cost of making it impossible to read and understand. **<u>If I cannot read your code, it means a wrong answer.</u>** Assume that all input are correct, that is, no program needs to expect wrong input. Use the following functions when you need:

```
(define (show-stream s n)
  (cond ((= n 0) 'done)
        (else (display "  ")
              (display (stream-first s))
              (show-stream (stream-rest s) (- n 1)))))

(define (stream-ref s n)
    (if (= n 0)
        (stream-first s)
        (stream-ref (stream-rest s) (- n 1))))

(define (stream-map proc s)
   (if (stream-empty? s)
       empty-stream
       (stream-cons (proc (stream-first s))
                    (stream-map proc (stream-rest s)))))
(define (stream-map2 proc s1 s2)
    (if (stream-empty? s1)
        empty-stream
        (stream-cons (proc (stream-first s1) (stream-first s2))
                (stream-map2 proc (stream-rest s1) (stream-rest s2)))))

(define (stream-filter pred s)
  (cond ((stream-empty? s) empty-stream)
        ((pred (stream-first s))
         (stream-cons (stream-first s)
              (stream-filter pred (stream-rest s))))
        (else (stream-filter pred (stream-rest s)))))

(define (scale-stream factor stream)
         (stream-map (lambda (x) (* x factor)) stream))
```

```
(define (interleave s1 s2)
  (if (stream-empty? s1)
      s2
      (stream-cons (stream-first s1)
                   (interleave s2 (stream-rest s1)))))

(define (add-streams s1 s2) (stream-map2 + s1 s2))
(define (div-streams s1 s2) (stream-map2 / s1 s2))
(define (mul-streams s1 s2) (stream-map2 * s1 s2))

; s1 is a stream that may or may not be infinite,
; and s2 is a promise (delayed object) that will generate a stream.
(define (stream-append-delayed s1 delayed-s2)
   (if (stream-empty? s1)
       (force delayed-s2)
       (stream-cons (stream-first s1)
                    (stream-append-delayed (stream-rest s1) delayed-s2))))

(define zeros (stream-cons 0 zeros))
(define ones (stream-cons 1 ones))
(define integers (stream-cons 1 (add-streams ones integers)))
(define odds (stream-cons 1 (stream-map (lambda (x) (+ x 2)) odds)))
(define evens (scale-stream  2 integers))
```

1. What will Scheme print in response to the following expressions? If an expression produces an error message, you may just write “**`error`**”; you don’t have to provide the exact text of the message. If the value of an expression is a procedure, just write “**`procedure`**”.

(a)
```
(define (f lst)
    (define (g lst)
             (set! lst (cons 4 lst))
             lst)
    (let ((y (g lst)))
        y))

(define a (list 1 2 3))

(f a)
```

____

```
a
```

____

(b)
```
(define y 5)
(define (agent x)
   (let ((y 0))
      (lambda () (x)) ))
(define mission
    (agent (lambda () (set! y (+ y 1)))))

(mission)
(mission)
 y
```

____

(c)
```
(define alt (stream-cons 0 (interleave integers alt)))
(show-stream alt 10)
```

(d)
```
(define cube (lambda (x) (* x x x)))
(define ff
   (lambda (g)
     (lambda (ff)
     (ff g))))
 ((ff 3) cube)
```

2. In a new scheme session, you enter the following forms in the order given:

```
> (define (f)
      (let ((a 3)) (lambda (x) (set! a (+ x a)) a)))
> (define my-f (f))
> (define my-f2 (f))
> (define g (let ((a 3)) (lambda (x) (set! a (+ a x)) a)))
> (define my-g1 g)
> (define my-g2 g)
```

Next to each of the forms below *which are entered in this order*, write the return value.

> **(my-f 4)**

> **(my-f 5)**

> **(my-f2 5)**

```
> (my-f2 4)
```

```
> (my-g1 4)
```

```
> (my-g1 5)
```

```
> (my-g2 5)
```

```
> (my-g2 4)
```

3. Write a scheme function **`(take s n) : stream x integer -> list`**, which accepts an infinite stream ***s*** and a nonnegative integer ***n***, and returns a list that consists of the first ***n*** elements of the stream. When *n* is zero, it returns an empty list. DO NOT define any help function.

```
(define (take s n)
    (if ( =  < A >  < B > )  < C >
        (< D >  ( < E >   < F > )
                (take ( < G >  < H > ) (-  n 1))) ))
```

<A> : ______________________ **<B> :** ______________________

<C> : ______________________ **<D> :** ______________________

<E> : ______________________ **<F> :** ______________________

<G> : ______________________ **<H> :** ______________________

5. Write a brief scheme expression that generates the following stream **g**:

   { (1) (2 1) (3 2 1) (4 3 2 1) (5 4 3 2 1) (6 5 4 3 2 1) …}

   In the above, the notation { . . . } represents a stream, while (. . .) represents a list. Note that { . . . } is not a Scheme notation, so you cannot use it in your program. Notice that **g** is an infinite stream of lists. Make your program as simple as possible. DO NOT DEFINE helper functions.

```
(define g
   (stream-cons < A >
               (stream-map2 < B > (< C >  < D >)  g)))
```

<A> : ______________________ **<B> :** ______________________

<C> : ______________________ **<D> :** ______________________

6. <u>Draw the environment diagram</u> resulting from evaluating the following expressions, and <u>show the result printed</u> by the expressions where indicated. <u>Do not erase</u> any box that you have drawn.

```
> (define x 10)
> (define (bar y) (* x y))
> (define (foo x)
     (let ((x 5) (y x))
        (lambda (x) (+ y (bar x)))))
> ((foo 3) 4)
```

____

GE →

7. Define a one-parameter procedure **previous?** that returns **#t** or **#f** as follows:

The first time **previous?** is called, it should return **#f**. After that, it returns **#t** if the argument is the same as the argument that was passed to it the previous time it was called. It returns **#f** otherwise. The following examples show the behavior of **previous?**. Your procedure must be self-contained, that is, you cannot use any global variable. Note that the argument of **previous?** can be any simple data type such as number, symbol, or character.

```
(define previous?  <your code to answer>)
(previous? 3)
#f
(previous? 4)
#f
(previous? 4)
#t
(previous? 3)
#f
```

```
(define previous?  ;; your code to complete the program follows
   (let ((called-before #f )
         (previous-param 0))  ; can be any value
     (lambda (input)
       (let ((result (eq? < A >  input)))
         (set!  < B >   < C > )
         (if < D >
             < E >
             (begin (set! called-before #t) < F > ))))))
```

<A> : ____________________ **<B>** : ____________________

<C> : ____________________ **<D>** : ____________________

<E> : ____________________ **<F>** : ____________________

8. Many useful mathematical functions can be expanded into what is known as a power series, a sum of terms of increasingly high-order powers of the function's argument. For example, we have

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \quad \cdots$$

We are going to approximate exponentiation by adding up terms in this series. Thus our first approximation to e^x would be

0

and our subsequent approximations would be

$$1 \qquad 1+x \qquad 1+x+\frac{x^2}{2!} \qquad 1+x+\frac{x^2}{2!}+\frac{x^3}{3!} \qquad 1+x+\frac{x^2}{2!}+\frac{x^3}{3!}+\frac{x^4}{4!}+ \quad \cdots$$

and so on.

We are going to do this using stream. Assume that the functions `add-streams`, `div-streams`, `mul-streams` and `scale-stream` are provided as above.

(a) Define **factorials** to be an infinite stream of factorials. Assume that the first few elements of this stream should be **1,1,2,6,24**, …..

```
(define factorials
    ( ___< A >___   ___< B >___
                ( ___< C >___   factorials   ___< D >___ )))
```

<A> : ______________________ **<B> :** ______________________

<C> : ______________________ **<D> :** ______________________

(b) We need to generate a stream of powers of *x*, for any *x*. To do this, we will create a procedure **powers** which should behave as follows:

```
> (show-stream (powers 2) 10)
;; will show 1 2 4 8 16 32 64 128 256 512
```

Here is a definition of **powers**:

```
(define (powers x)
   (define pwrs
       ( ___< A >___   ___< B >___
                   ( ___< C >___  ___< D >___  ___< E >___)))
   pwrs )
```

<A> : ______________________ **<B> :** ______________________

<C> : ______________________ **<D> :** ______________________

<E> : ______________________

(c) We can use the pieces defined above to create a stream of terms for a power series. The procedure **`(exp-terms 2)`**, for example, should return the sequence of values

$$\frac{1}{1} \qquad \frac{2}{1!} \qquad \frac{2^2}{2!} \qquad \frac{2^3}{3!} \qquad \frac{2^4}{4!}$$

and so on.

```
(define (exp-terms x)
     ( __< A >__ ( __< B >__  __< C >__ ) __< D >__ ))
```

<A> : ______________________ **<B> :** ______________________

<C> : ______________________ **<D> :** ______________________

(d) Using this, complete the definition below so that **`exp-approx`** will return a stream of successively better approximations of e^x, as listed at the beginning of this problem.

```
(define (exp-approx x)
   (define __< A >__
      (stream-cons  __< B >__
                    (__< C >__  __< D >__
                           ( __< E >__  __< F >__ ))))
    approx)
```

<A> : ______________________ **<B> :** ______________________

<C> : ______________________ **<D> :** ______________________

<E> : ______________________ **<F> :** ______________________

9. We will discuss how to use message-passing with dyadic functions:

((num1 ’add) num2)

will result in the complex number with value **(num1 $+_c$ num2).**

Shown below is the message-passing implementation of complex number from the text. Fill in the blank to handle the **add** message correctly, supposing that both operands are complex numbers.

```
(define (make-from-real-imag x y)
  (define (dispatch op)
    (cond ((eq? op 'real-part) x)
          ((eq? op 'imag-part) y)
          ((eq? op 'magnitude)
           (sqrt (+ (* x x) (* y y))))
          ((eq? op 'angle) (atan y x))
          ((eq? op  ( < A > ))
           (  < B >   ( num )
                 (  < C >
                     ( +  < D >  (  < E >  'real-part))
                     ( +  < F >  (  < E >  'imag-part)))))
          (else (error ”Unknown op” op))))
  dispatch)
```

<A> : ____________________ **<B> :** ____________________

<C> : ____________________ **<D> :** ____________________

<E> : ____________________ **<F> :** ____________________