Programming Paradigms
CISC-280 Fall 2000

Final Exam

NAME:

There are 100 points on 9 pages.

6 points

For each procedure, say if it is linear recursive, tail recursive/iterative, or tree recursive.

```
(define (square-list items)
  (if (null? items)
      nil
      (cons (square (car items)) (square-list (cdr items)))))

(define (eval-sequence exps env)
  (cond ((last-exp? exps) (eval (first-exp exps) env))
        (else (eval (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))

(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1)) (fib (- n 2))))))
```

6 points

Write expressions to get the symbol “hello” from each of the following list variables x, y, and z:

```
(define x '((foo bar baz) (hello fritz marble)))
(define y (cons  '((name fred) (state pa) (total 1556))
                 '(((name barney) (state nj) (total 876))
                   ((name hello) (state ny) (total 1038)))))
(define z (append (list 'baby 'mother)
                  (list 'doctor 'hello)
                  (cons 'creature (list 'judgement 'butcher 'engineer)))
```

8 points

Procedures as parameters and returned values: Define a procedure (compose f g). The procedure `compose` takes two, one-argument procedures `f` and `g`, and it RETURNS a one-argument procedure that for any argument x returns f of g of x (i.e. f(g(x)) in mathematical notation). For example, compose could be used like this (assume `factorial` takes one arg and returns a number, and `square` takes one arg and returns a number):

```
(define biggie (compose factorial square))
(biggie 2) --> 24 ;this is (2x2)!
(biggie 10) --> 933262154439441526816992388562667004
    907159682643816214685929638952175999932299156089
    414639761565182862536979208272237582511852109000
    16864000000000000000000000000 ;This is (10x10)!



(define (compose f g)
```

10 points

Consider the following definitions:

```
(define (five) (+ 2 3))
(define foo five)
(define bar (five))
(define (baz) five)
(define (fizz) (five))
```

What do the following do and/or return? If it's an error, say so. If the result is just a procedure (i.e. DrScheme prints `#<procedure:baz>`), then say so. Hint: you might want to rewrite any procedures using syntactic sugar above into their base "lambda" form. Hint: take your time and work it out using the substitution model.

```
(foo)

bar

(bar)

((baz))

(fizz)
```

5 points

Define a procedure `(trace proc)`. It returns a *procedure* which does the same thing as `proc` except that it prints "`entering #<proc:proc>`" before executing proc and "`exiting #<proc:proc> with result`" afterwards. Make sure that it returns the original value. You may assume that traced functions only take 1 argument, or use the (. args) notation for arbitrary arguments. Assume `(display square)` prints `#<procedure:square>`.

For example, after you have defined `trace`, you could have the following:

```
(define (square x) (* x x))
(define (add a b) (+ a b))
(define (test x y) (add (square x) (square y)))
(define square (trace square))
(define add (trace add))
>(test 2 3)
entering #<procedure:square>
exiting #<procedure:square> with 4
entering #<procedure:square>
exiting #<procedure:square> with 9
entering #<procedure:add>
exiting #<procedure:add> with 13
13
```

15 points

Draw the environment diagram (all frames and user-defined procedure definitions) that results from executing the following lines of Scheme code.

```
(define (make-withdraw balance)
   (lambda (amount)
      (if (> = balance amount)
           (begin (set! balance (- balance amount))
                        balance)
            "Insufficient Funds")))
(define W1 (make-withdraw 100))
(W1 50)
```

10 points

Object Oriented Programming: In class we defined a simple OOP system for Scheme. We did an example involving the speaker, lecturer, and arrogant-lecturer classes (each inherits from the previous one):

```
(define (ask object message . args)
  (let ((method (get-method object message)))
    (if (method? method)
        (apply method (cons object args))
        (error "No method" message (cadr method)))))
(define (make-speaker)
  (define (self message)
    (cond ((eq? message 'say)
           (lambda (self stuff) (display stuff)))
          (else (no-method message))))
  self)
(define (make-lecturer)
  (let ((speaker (make-speaker)))
    (define (self message)
      (cond ((eq? message 'lecture)
             (lambda (self stuff)
               (ask self 'say stuff)
               (ask self 'say '(you should be taking notes))))
            (else (get-method speaker message))))
    self))
(define (make-singer)
  (lambda (message)
    (cond ((eq? message 'say)
           (lambda (self stuff)
             (display (append '(tra-la-la --) stuff))))
          ((eq? message 'sing)
           (lambda (self)
             (display '(tra-la-la))))
          (else (no-method "SINGER")))))
```

We also defined Ben to be both a singer and a lecturer.

```
(define ben
  (let ((singer (make-singer))
        (lecturer (make-lecturer)))
    (lambda (message)
      (let ((sing (get-method singer message))
            (lect (get-method lecturer message)))
        (if (method? sing)
            sing
            lect)))))
```

Questions are on the next page....

A. [10 points] How can we modify Ben so that if a message is handled by both Ben's "singer" AND Ben's "lecturer", that BOTH methods are executed?

15 points

Concurrency. Examine the following Monkeys-and-rope simulation:

```
;;; Rope is 0 if empty, positive n for n monkeys going east, negative n for
n monkeys going west.
(define *ROPE* 0)
(define (make-monkey name)
  (let ((side (if (= (random 2) 0) 1 -1)))
    (define (try-cross)
      (sleep (random 2)) ;wait a while
      (if (test-and-grab-rope side)
          (begin (cross)
                 (release-rope side)
                 (print-success side name))
          (try-cross)))
    try-cross))
(define (same-sign? x y)
  (or (= x y)
      (and (> x 0) (> y 0))
      (and (< x 0) (< y 0))))
(define (test-and-grab-rope side)
    (cond ((= *ROPE* 0)
           (set! *ROPE* (+ *ROPE* side))
           #t)
          ((same-sign? *ROPE* side)
           (set! *ROPE* (+ *ROPE* side))
           #t)
          (else #f)))
(define (release-rope side)
  (set! *ROPE* (- *ROPE* side)))
(define (print-success side name)
  (display "Monkey ")
  (display name)
  (display " went from ")
  (if (> side 0)
      (display "west to east.")
      (display "east to west."))
  (newline))
(define (cross) (sleep (random 2)))
(define (test)
  (parallel-execute (make-monkey 1) (make-monkey 2) (make-monkey
3) (make-monkey 4) (make-monkey 5) (make-monkey 6) (make-monkey 7)
(make-monkey 8) (make-monkey 9) (make-monkey 10)))
```

A. [5 points] How many serializers (created by `(make-serializer)`) are needed to make this code safe? Remember that a serializer takes a procedure (or lambda expression) as an argument and returns a new, "protected" lambda such that no two procedures protected by the same serializer can have their execution interleaved in parallel. The classic example is to protect access to a shared variable that is being mutated. It's best if the smallest amount of code is placed in the serializer,

since the code inside the serializer CANNOT execute in parallel with any other serialized code.

B. [10 pts] Modify the above code with the appropriate serializer(s) Please feel free to just mark up the code above.

15 points

Recall that the code for our implementation of the Scheme Evaluator was:

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

The code as it stands does NOT handle all of Scheme's Special Forms. Modify the code above to handle the AND special form. Recall that AND evaluates its expressions from left to right. If any expression evaluates to false, false is returned and any remaining expressions are NOT evaluated. If all the expressions evaluate to true values, then the value of the last expression is returned. If there are no expressions then true is returned.

A. [5 pts] Assume we already have a procedure `(eval-and exp env)`, and a predicate `(and? exp)`. Indicate how we would modify the above code to add in the AND special form (just write it in above and put an arrow to the right spot).

B. [10 pts] Write the procedure `(eval-and exp env)`. Just use the plain old regular list operations we've used before, don't use the complex redefinitions from the book chapter 4.

4 points

Complete the following alternative definition of the infinite stream of integers:

```
(define integers (cons-stream 1 (stream-map <??> integers)))
```

6 points

The procedure `reorder-stream` rearranges items in one stream (the data-stream) into the order specified by another stream (the order stream), which consistes of item numbers specifying the desired order. For example, if the order stream starts with 4, 13, 2, 8 and the order stream starts with 3, 1, 4, 2 then the result stream will start with 2, 4, 8, 13. (The first item of the result is the third item of the data, the second item of the result is the first item of the data, etc.) [application: some fancy memory-based sorting, or even edit decision lists for video streams]

A. Complete the following definition:

```
(define (reorder order-stream data-stream)
   (cond ((stream-null? order-stream) the-empty-stream)
               ((stream-null? data-stream) the-empty-stream)
               (else (cons-stream <??>  <??>))))
```

The following stream definitions are used to define the stream of factorials.

```
(define ones
  (cons-stream 1 ones))
(define integers
  (cons-stream 1 (add-streams ones integers)))
(define fact
  (cons-stream 1 (mul-streams fact integers)))
```

B. How many multiplications are performed when computing the nth factorial value in the fact stream? Explain your answer.

C. If there were no memoization in Scheme, how many multiplications would be performed when computing the nth factorial value in the fact stream? Explain your answer.