

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1995

Final Exam
Closed Book

Please write clear, concise answers to the questions in the spaces provided in this booklet. You may use scratch paper if you need, but the provided spaces are the **only** places we will look at when grading.

Write your name here:

Write your recitation instructor's name here:

Write your TA's name here:

Write the name of Marcy's pet chicken here:

Final grades will be available beginning December 21, 1995, in NE43-711. You must come in person, and show proof of identity. We will not give out grades over the phone. **Please DO NOT CALL the Course 6 undergraduate office for grades.** Graded final examinations will not be returned to the examinees, but may be reviewed in NE43-711.

This exam has 12 Problems.

Please do not write below this line:

| | Value | Grade | Grader | | Value | Grade | Grader | | Value | Grade | Grader |
|---|-------|-------|--------|---|-------|-------|--------|-----|-------|-------|--------|
| 1 | 12 | | | 5 | 11 | | | 9 | 25 | | |
| 2 | 17 | | | 6 | 25 | | | 10 | 13 | | |
| 3 | 24 | | | 7 | 8 | | | 11 | 25 | | |
| 4 | 8 | | | 8 | 16 | | | 12 | 16 | | |
| | | | | | | | | Ttl | 200 | | |

Question 1 (12 points): There are four independent parts to this problem. For each part, a sequence of expressions is given, which you may assume is typed to a Scheme interpreter and evaluated in the order shown. Write the value that will be printed in response to the **last expression** in each sequence. Determine your answer carefully: no partial credit will be given.

Part 1a:

```
(define x '(y z))
```

```
(define y (append '(x) (list x 'x)))
```

y

Answer:

Part 1b:

```
(define a '(1 2 3))
```

```
(set-car! (cdr a) '(4 5))
```

a

Answer:

Part 1c:

```
(define s1 (cons-stream 1 (stream-map (lambda (x) (+ x 3)) s1)))
```

```
(define s2 (stream-filter (lambda (x) (even? x)) s1))
```

```
(stream-car (stream-cdr (stream-cdr s2)))
```

Answer:

Part 1d:

```
(define g (lambda (*) (lambda (a b) (a * b))))
```

```
((g 3) - 5)
```

Answer:

Question 2 (17 points):

Write a procedure `reverse` to reverse a list, with the following behavior:

```
(reverse '(1 (2 3) 4 (5 (6 7))))  
;Value: ((5 (6 7)) 4 (2 3) 1)
```

Your procedure must give rise to a recursive process, not an iterative one.

What is the order of growth in time of your procedure, for a list of n elements?

Briefly explain why it has this order of growth.

Write a procedure `deep-reverse` to deep reverse a list, i.e.

```
(deep-reverse '(1 (2 3) 4 (5 (6 7))))  
;Value: (((7 6) 5) 4 (3 2) 1)
```

Again be sure your procedure generates a recursive process.

Question 3 (24 points):

Part a: Fill in the blanks in the following description: In the eval/apply structure of the metacircular evaluator, EVAL takes as inputs and and calls APPLY with and .

Part b: Each of the following statements is either true (T) or false (F). For each one, circle the best answer:

In Scheme, procedures are not first class objects, because they can take arbitrary numbers of arguments. T F

Noncomputability of the Halting Problem implies that an optimal Scheme compiler is a logical impossibility. T F

In the compiler described in the notes, the “modified” registers of a compiled instruction sequence are necessarily a subset of the “needed” registers of that sequence. T F

Self parameters in objects allow better control of inheritance. T F

The original motivation behind the World Wide Web, according to its creator, Tim Berners-Lee, was to create a net wide browser. T F

The basic reason that the ANALYZE evaluator is faster than the EVAL/APPLY evaluator is that a procedure body can be analyzed once, but executed many times. T F

The main advantage of the AMB evaluator over the EVAL/APPLY evaluator is that it incorporates tail recursion. T F

In the US a patent has a lifetime of 75 years. T F

Independent reinvention is a valid defense against claims of patent infringement. T F

Independent reinvention is a valid defense against claims of copyright infringement. T F

Question 4 (8 points):

Suppose that `parallel-execute` is a special Scheme construct that creates a separate process for each argument (each of which is a procedure of no arguments) and which applies each argument, running each concurrently in its own process. Now consider the following example.

```
(define x 10)

(parallel-execute (lambda () (set! x (* x x)))
                  (lambda () (set! x (+ x 1))))
```

What are all the possible values of `x` after the completion of this concurrent execution?

Now suppose that we evaluate the following, where `make-serializer` behaves as described in lecture and in the notes:

```
(define x 10)

(define s (make-serializer))

(parallel-execute (lambda () (set! x ((s (lambda () (* x x))))))
                  (s (lambda () (set! x (+ x 1)))))
```

What are the possible values for `x` after this evaluation?

Question 5 (11 points):

Consider the following set of expressions:

```
(define f
  (lambda (x)
    (let ((y 2))
      (lambda () (+ x y))))))
```

```
(define g (f 5))
```

Draw an environment diagram that results from the evaluation of these expressions.



What is the value of `g`?



Question 6 (25 points):

You are to add a new special form to the metacircular evaluator in Chapter 4 of the notes. The form is a `while` expression, and has the following syntax:

```
(while <pred> <body>)
```

for example:

```
(while (< i 10)
      (if (> (f i) 0)
          (g i)
          (g (- i))))
      (set! i (+ i 1)))
```

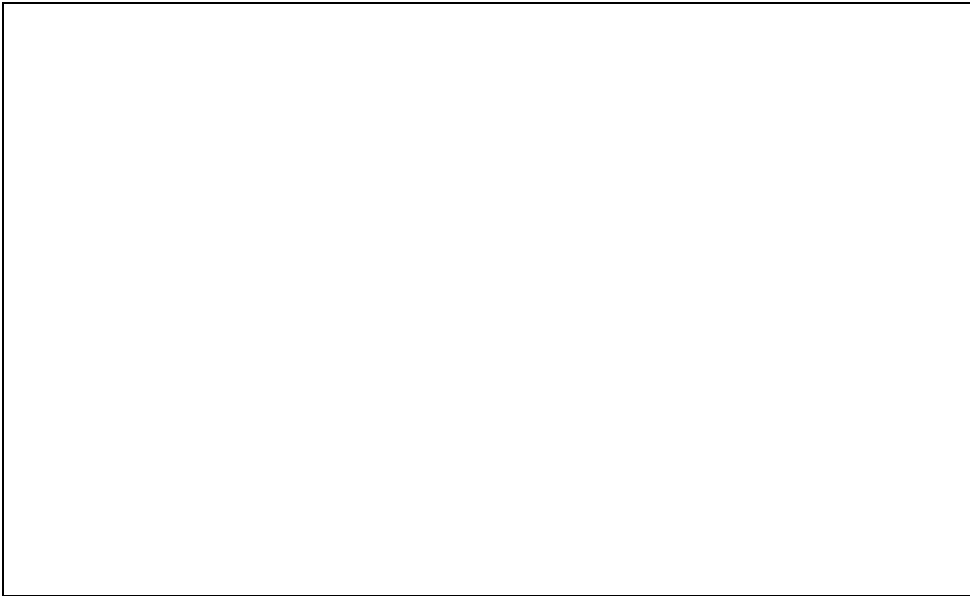
The idea behind a `while` loop is to test the first, or predicate, expression. If it is true, then the body of the while (which may consist of several expressions) is evaluated. This continues until the predicate expression evaluates to false. At this point, the `while` expression is exited, returning the symbol `done`.

In this problem, we ask you to complete the code necessary to install this new form into the language.

Part a:

Complete the definition of a predicate `while?`, which returns true if the argument is a while expression:

```
(define (while? exp)
```



```
)
```

Part b: Assume that the procedure `eval-while` will be used to control the actual evaluation of the `while` loop. Indicate below what expression(s) must be added to `eval` and where they should be added, in order to install our means for evaluating this type of expression into our evaluator.

```
(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))))
```

What expression must be added?

Where should it be added?

Part c:

Assume that the following selectors are provided for you:

```
(define (while-test exp) (cadr exp))
```

```
(define (while-body exp) (caddr exp))
```

Using these data abstractions, provide a definition for `eval-while`.

Part d:

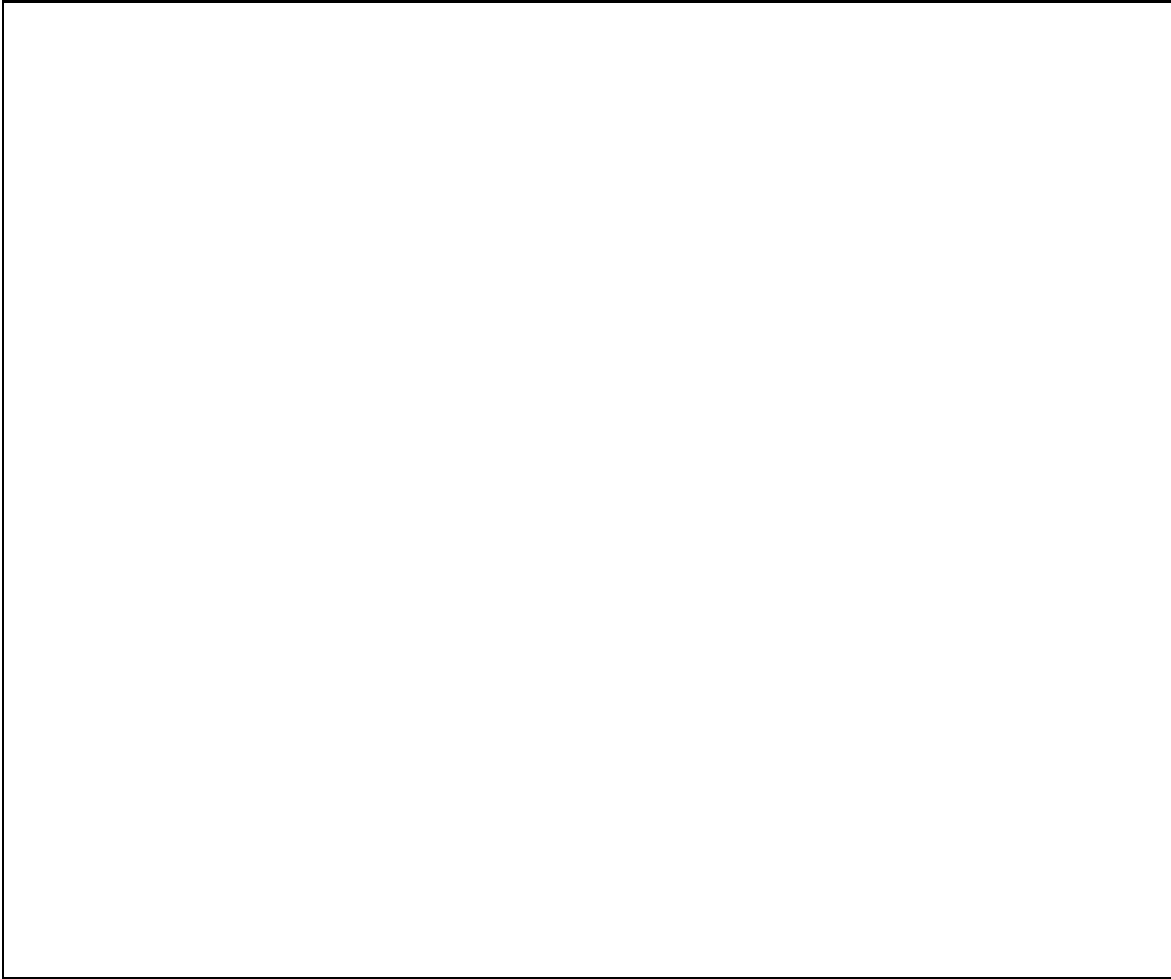
Now suppose that we want to add our `while` expressions to the `analyze` evaluator of chapter 4 of the notes.

We can add a clause to `analyze`:

```
(define (analyze exp)
  (cond ((self-evaluating? exp)
        (analyze-self-evaluating exp))
        ...
        ((while? exp) (analyze-while exp))
        ...
        (else
         (error "Unknown expression type -- ANALYZE" exp))))
```

Complete the installation of `while` into the `analyze` evaluator, by providing the procedure `analyze-while`:

```
(define (analyze-while exp)
```



```
)
```

Question 7 (8 points):

Here is the code for `apply` from the metacircular evaluator of chapter 4 of the notes:

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
        (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           arguments
           (procedure-environment procedure)))) ; ****
        (else
         (error
          "Unknown procedure type -- APPLY" procedure))))
```

Suppose we modify the line marked `****` in `apply` to replace the expression `(procedure-environment procedure)` by `the-global-environment`.

Provide a simple example of a procedure which will result in a different value being returned when run with the standard `apply` versus the modified `apply`.

Provide a brief, but clear, explanation of why your example returns different values with the two evaluators.

A large, empty rectangular box with a thin black border, intended for the student to provide a brief but clear explanation of why their example returns different values with the two evaluators.

Question 8 (16 points):

We want to build a new kind of data structure, called a queue. A queue is a linear structure that supports two operations: `insert` places an element at the end of the queue, and `delete` removes an element from the front of the queue, and returns the value of that element. We can define a data abstraction for the queue, with a constructor `make-queue`, and two selectors `head` and `tail`. We also have two mutators, `mutate-head!` and `mutate-tail!`. One possible implementation of these is:

```
(define make-queue cons)
(define head car)
(define tail cdr)
(define mutate-head! set-car!)
(define mutate-tail! set-cdr!)
```

The internal structure of the queue will be represented as a list, with the head and tail of the queue pointing to the beginning and end of the list, respectively.

A simple way to initialize a queue is to build it out of a list, as shown below:

```
(define (construct-queue lst)
  (let ((ptrs (make-queue '() '())))
    (mutate-head! ptrs lst)
    (mutate-tail! ptrs (last lst))
    ptrs))

(define (last lst)
  (cond ((null? lst) lst)
        ((null? (cdr lst)) lst)
        (else (last (cdr lst)))))

(define foo (construct-queue '(1 2)))

(delete foo)
;Value: 1

(insert 3 foo)
;Value: done

(delete foo)
;Value: 2

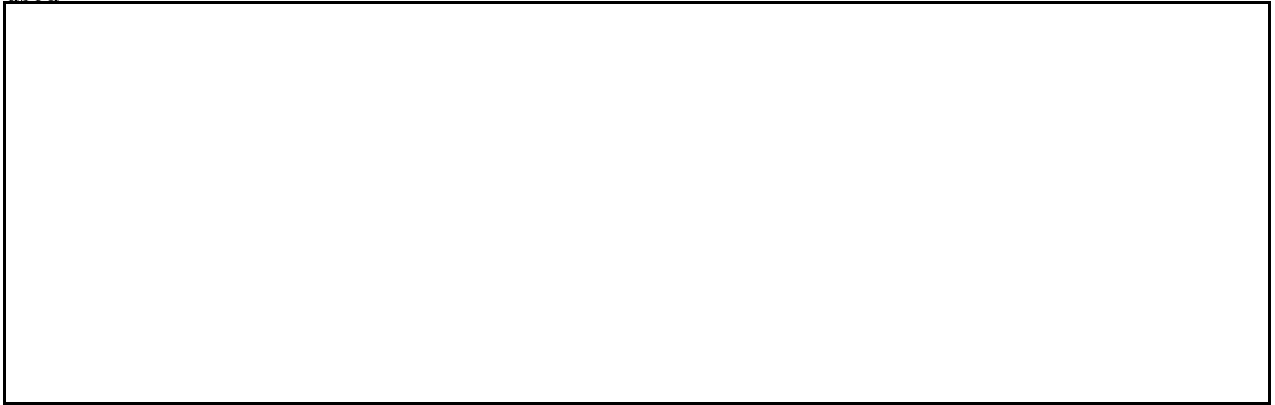
(delete foo)
;Value: 3

(delete foo)
;Nothing in Queue
;Type D to debug error, Q to quit back to REP loop: q
```


Part a: Draw a box and pointer diagram for `foo` after the evaluation of

```
(define foo (construct-queue '(1 2 3)))
```

For this part, you may assume that the implementation of the abstraction given above is being used.



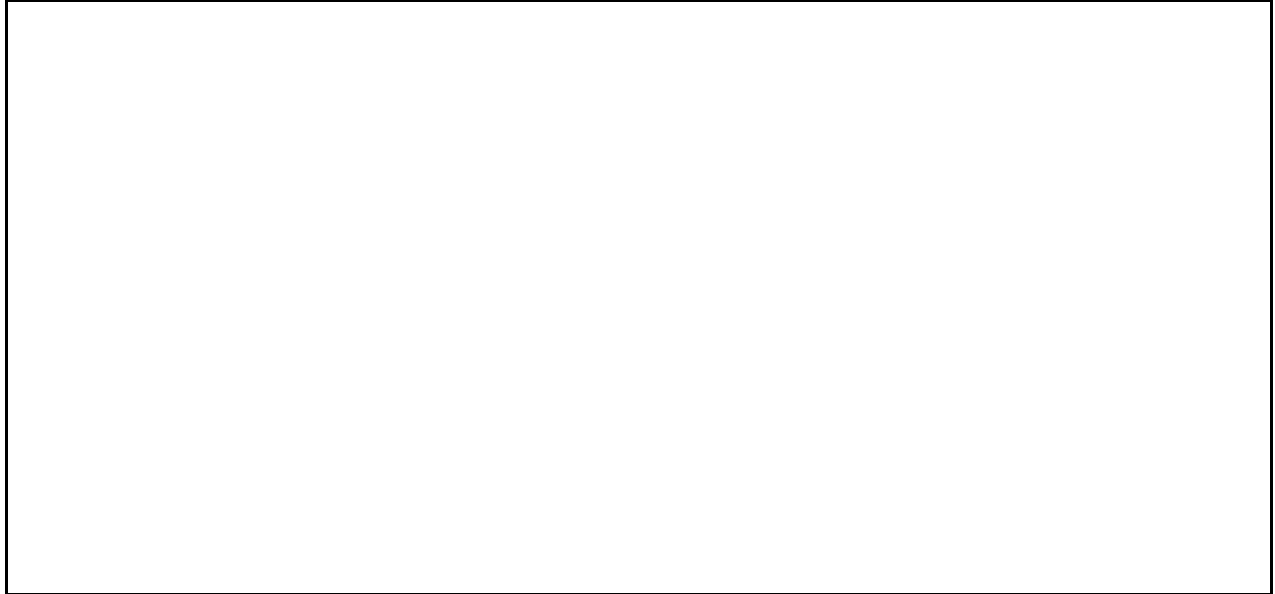
Part b: Using the data abstractions defined above, provide definitions for `insert` and `delete`.



Part c: Suppose we make the following change to our implementation

```
(define make-queue list)
```

Make whatever additional changes that are needed to keep the implementation consistent:



Question 9 (25 points):

In what follows, the `INSTRUMENT` procedure takes as argument a procedure `P` of one argument. `INSTRUMENT` returns an “instrumented procedure” that, when given an argument, calls `P` on that argument, but also keeps a count of how many times `P` has been called. Calling the instrumented procedure with the special argument `'COUNT` returns the count.

```
(define i-square (instrument square))
```

```
(i-square 3)
```

```
9
```

```
(i-square 25)
```

```
625
```

```
(i-square 'count)
```

```
2
```

```
(i-square 10)
```

```
100
```

```
(i-square 'count)
```

```
3
```

The following attempted definition of `INSTRUMENT` has a bug:

```
(define (instrument proc)
  (lambda (arg)
    (let ((counter 0))
      (if (eq? arg 'count)
          counter
          (begin (set! counter (+ 1 counter))
                 (proc arg))))))
```

Part a: How does this version of `INSTRUMENT` fail? Below, we’ve reprinted the interaction from above that shows the intended behavior of `INSTRUMENT`. In the blanks following each of the five calls, show what value is actually returned in each case.

```
(define i-square (instrument square))
```

```
(i-square 3)          intended value: 9    actual value:
```

```
(i-square 25)        intended value: 625  actual value:
```

```
(i-square 'count)    intended value: 2    actual value:
```

```
(i-square 10)        intended value: 100  actual value:
```

```
(i-square 'count)    intended value: 3    actual value:
```

| |
|--|
| |
| |
| |
| |
| |

Part b: Correct the bug in the above definition of `INSTRUMENT`. (Hint: Only a small change is required.)

Part c: Suppose we now want to make an Object Oriented version of an instrumented procedure, using the system described and used in Problem Set 6. Below is the framework for such an idea, (assume that `proc` is a procedure, not an object).

You are to complete the definition by filling in the three methods:

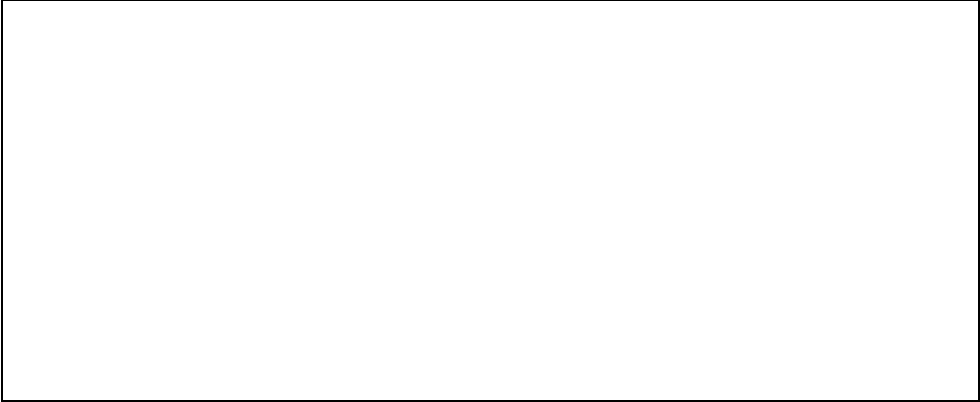
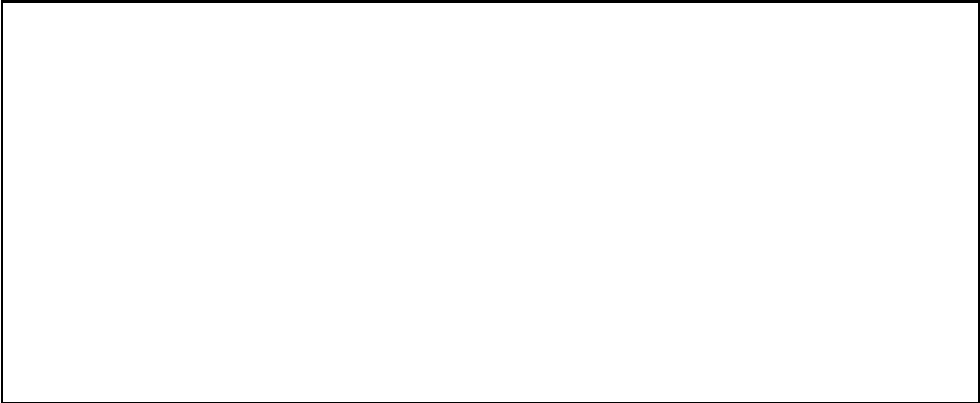
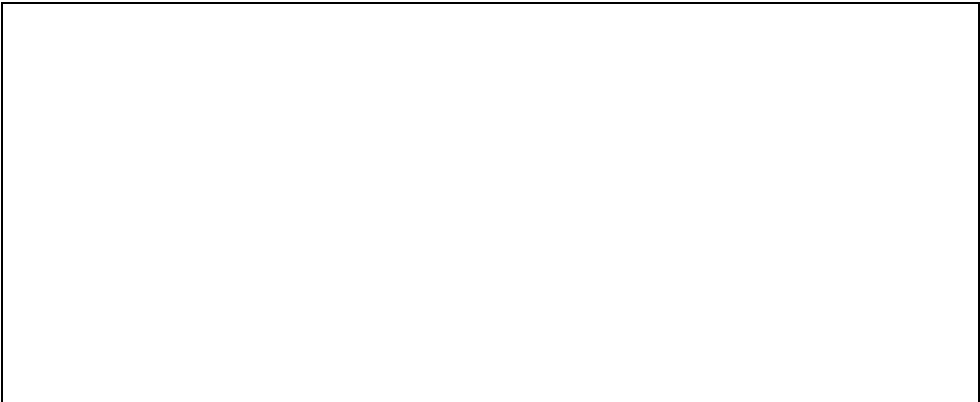
- the `reset` method should cause the state variable `counter` to return to 0.
- the `counts` method should return the value of the state variable `counter`
- the `run` method should increment the state variable `counter`, and return the value of running the procedure `proc` on its argument. For convenience, you may assume that we are only considering procedures of one argument.

Thus, an example of using this implementation of an instrumented procedure would be:

```
(define i-square (oops-inst square))
```

```
(ask i-square 'reset)
;Value: done
```

```
(ask i-square 'run 3)
;Value: 9
```

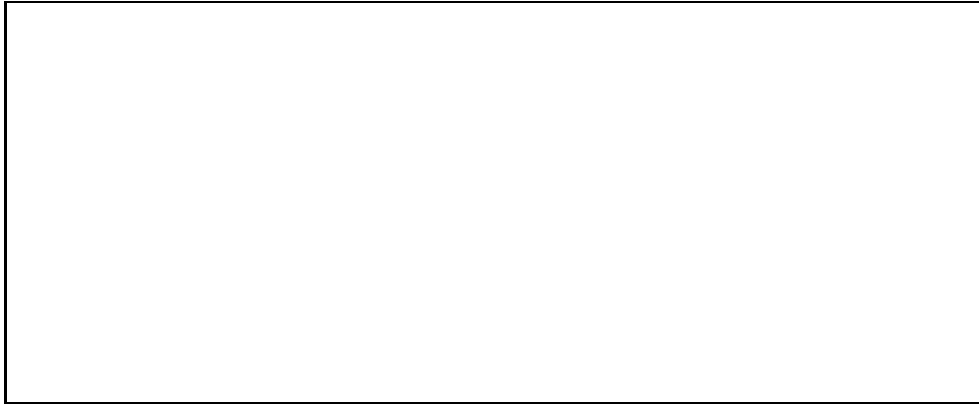
```
(define (oops-inst proc)
  (let ((counter 0))
    (lambda (message)
      (cond ((eq? message 'reset)
             
             )
            ((eq? message 'counts)
             
             )
            ((eq? message 'run)
             
             )
            (else (no-method proc))))))
```

Part d:

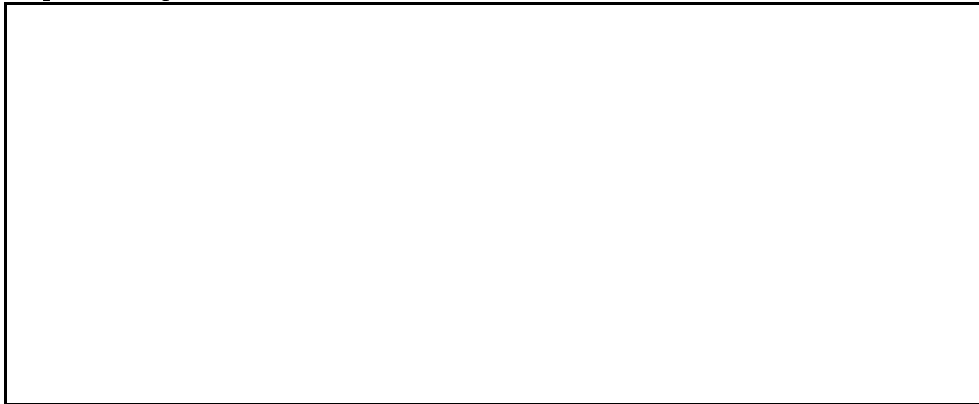
Now suppose that we want to create a new kind of OOPS instrumented procedure, which keeps track of the number of times the procedure is called with a negative argument. Below is the outline for such an idea. You are to complete the definition by filling in the needed methods:

- the `reset` method should cause the state variables `counter` and `negs` to return to 0.
- the `counts` method should return a list of the values of the state variables `negs` and `counter`
- the `run` method should increment the state variable `counter`, and if the argument to which the procedure is applied is negative, it should also increment `negs`. Finally, it should return the value of running the procedure `proc` on its argument. For convenience, you may assume that we are only considering procedures of one argument.
- in the final slot, you should provide a method for inheriting any other methods from the underlying instrumented procedure.

```
(define (new-oops-inst proc)
  (let ((negs 0)
        (basic (oops-inst proc)))
    (lambda (message)
      (cond ((eq? message 'reset)
```



```
)
  ((eq? message 'counts)
```



```
)
  ((eq? message 'run)
```



```
)
  (else
```



```
))))))
```

Question 10 (13 points)

Below is a simple recursive procedure:

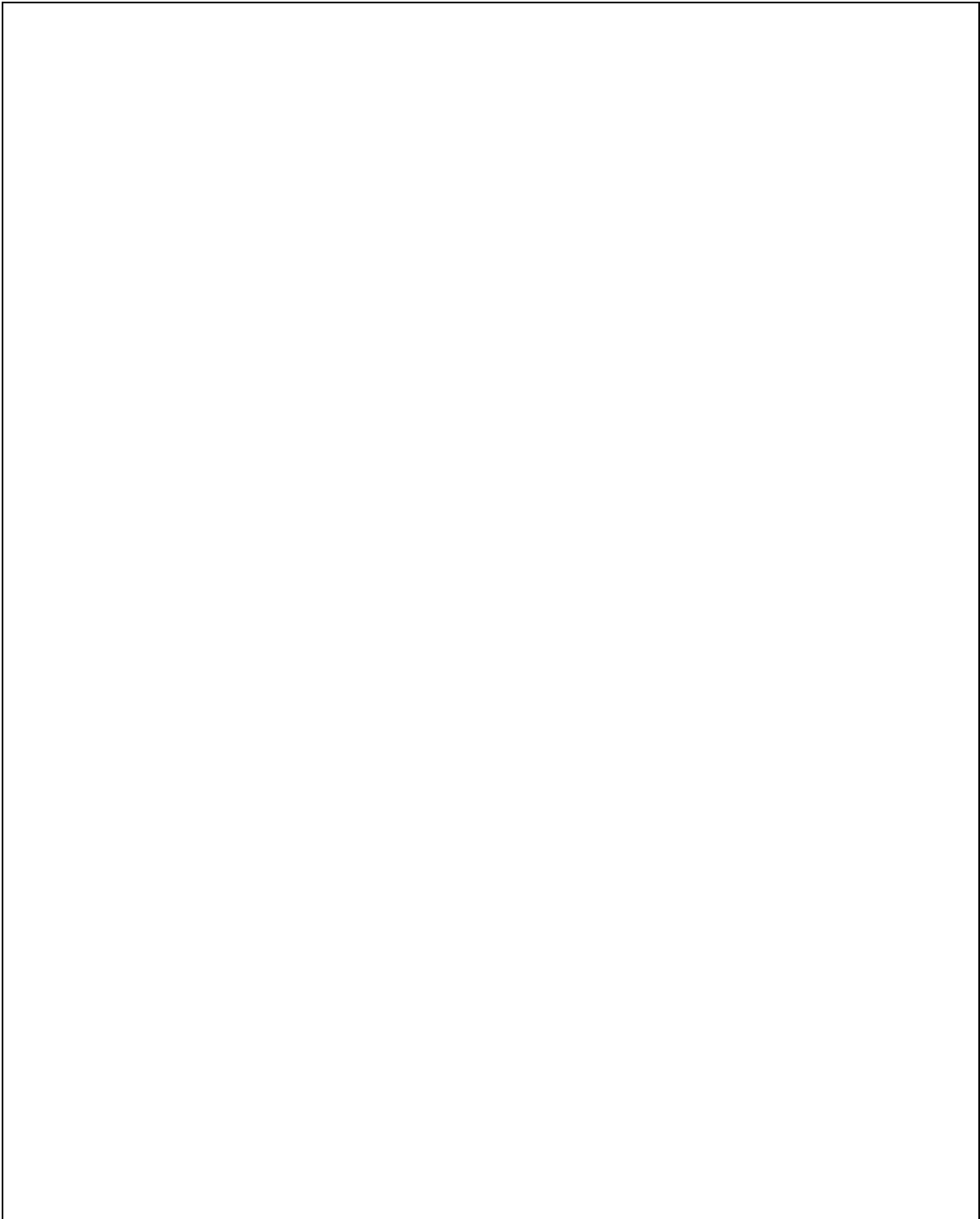
```
(define (f a b)
  (if (= a 0)
      0
      (+ b (f (- a 1) b))))
```

In the space below, write a register machine version of this procedure, which exhibits a linear recursive behavior. The machine may assume that the registers **a** and **b** are initially loaded with arguments, and it should leave its result in the **val** register when done.

(registers a b continue val)

(operations + - =)

(controller



)

Question 11 (25 points)**Part a:**

Assume that `ran` is a primitive Scheme procedure that generates random numbers in the range 0 to 1, e. g.

```
(ran)
0.486726
```

```
(ran)
0.929204
```

```
(ran)
0.008849
```

```
(ran)
0.283186
```

Assume that successive calls to `RAN` *never* produce the same number.

Louis Reasoner wants to define a stream whose elements consist of different random numbers, as in the sequence above. He attempts to define a stream of random numbers as follows:

```
(define random-stream
  (cons-stream (ran)
              random-stream))
```

Lem E. Tweakit isn't sure that Louis' definition will work, and he suggests the following:

```
(define (make-random-stream)
  (cons-stream (ran)
              (make-random-stream)))
```

```
(define random-stream (make-random-stream))
```

The two friends show their work to Alyssa P. Hacker who suggests that they use `PRINT-STREAM` to examine the first few elements of their streams. Furthermore she suggests that they run their code on two different Scheme interpreters, one that implements the `DELAY` special form using memoization, and one that does not.

For example, in the standard form, `delay` can be a special form such that

```
(delay exp)
```

is syntactic sugar for

```
(lambda () exp)
```

and `force` simply calls the procedure (of no arguments) produced by `delay`, so we can implement `force` as a procedure:

```
(define (force delayed-object)
  (delayed-object))
```

In the memoized form, we build delayed objects so that the first time they are forced, they store the value that is computed. Subsequent forcings will simply return the stored value without repeating the computation. In other words, we implement `delay` as a special-purpose memoized procedure, by using the following procedure, which takes as argument a procedure (of no arguments) and returns a memoized version of the procedure. The first time the memoized procedure is run, it saves the computed result. On subsequent evaluations, it simply returns the result.

```
(define (memo-proc proc)
  (let ((already-run? false) (result false))
    (lambda ()
      (if (not already-run?)
          (begin (set! result (proc))
                 (set! already-run? true)
                 result)
          result))))
```

`Delay` is then defined so that `(delay exp)` is equivalent to

```
(memo-proc (lambda () exp))
```

and `force` is as defined previously.

Louis and Lem take her advice, and, just to be sure, they print their streams twice. Shown below are pairs of printouts, of the sort that either Louis or Lem might have produced.

Possible outcomes:

```
(print-stream random-stream)          ;;; Outcome A
{0.486726 0.929204 0.008849 0.283186 ...
```

```
(print-stream random-stream)
{0.486726 0.929204 0.008849 0.283186 ...
```

```
(print-stream random-stream)          ;;; Outcome B
{0.486726 0.929204 0.008849 0.283186 ...
```

```
(print-stream random-stream)
{0.486726 0.521080 0.297045 0.991644 ...
```

```
(print-stream random-stream)          ;;; Outcome C
{0.486726 0.929204 0.008849 0.283186 ...
```

```
(print-stream random-stream)
{0.365913 0.521080 0.297045 0.991644 ...
```

```
(print-stream random-stream)          ;;; Outcome D
{0.486726 0.486726 0.486726 0.486726 ...
```

```
(print-stream random-stream)
{0.486726 0.486726 0.486726 0.486726 ...
```

```
(print-stream random-stream)          ;;; Outcome E
{0.486726 0.486726 0.486726 0.486726 ...
```

```
(print-stream random-stream)
{0.591003 0.591003 0.591003 0.591003 ...
```

Circle which outcome(s) *could* have been produced by Louis' and Lem's definitions when run on the two types of interpreter. If none of the results printed above are appropriate, circle *None*.

- | | | | | | | |
|---|---|---|---|---|---|------|
| 1) Louis' definition; no memoization of DELAY: | A | B | C | D | E | None |
| 2) Louis' definition; interpreter memoizes DELAY: | A | B | C | D | E | None |
| 3) Lem's definition; no memoization of DELAY: | A | B | C | D | E | None |
| 4) Lem's definition; interpreter memoizes DELAY: | A | B | C | D | E | None |

Part b: In the remaining parts of this problem assume that an infinite stream of random numbers between 0 and 1 can be satisfactorily defined as:

```
(define random-stream (random-stream-generator))
```

Suppose we represent a *point* in the plane as a list of two numbers, the x-coordinate and the y-coordinate. Making use of `RANDOM-STREAM-GENERATOR` and/or `RANDOM-STREAM`, define the stream `RANDOM-POINTS`, which is to be an infinite stream of random points in the unit square (that is, points (x,y) where x and y are in the range [0, 1]).

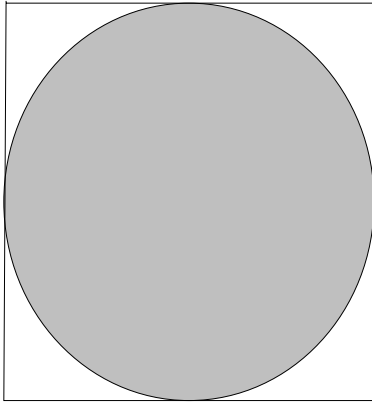
```
(define random-points
  (stream-map-2
```

```
))
```

where `STREAM-MAP-2` is defined as

```
(define (stream-map-2 proc s1 s2)
  (cons-stream (proc (stream-first s1) (stream-first s2))
    (stream-map-2 proc (stream-rest s1) (stream-rest s2))))
```

Part c: The unit square has area 1, the area of the inscribed circle is $\pi/4$. So if we select at random a point in the unit square, the point will fall inside the circle with frequency equal to $\pi/4$. As we choose more and more points, the ratio of the number of points inside the circle to the total number of points chosen will more and more likely be a good approximation to $\pi/4$.



Louis Reasoner decides to use this idea, together with the stream `RANDOM-POINTS`, to generate a stream of numbers that approximate π . He first defines the following procedure, which takes a point as argument and returns 1 or 0 depending on whether or not the point is inside the unit circle:

```
(define (count-if-inside-circle p)
  (if (< (+ (square (- (car p) .5))
           (square (- (cadr p) .5)))
      .25)
      1
      0))
```

He defines an infinite stream of ones:

```
(define ones (cons-stream 1 ones))
```

He also defines the following procedure:

```
(define (partial-sums stream)
  (define (rsum initial s)
    (cons-stream initial
                  (rsum (+ (stream-first s) initial)
                        (stream-rest s))))
  (rsum (stream-first stream) (stream-rest stream)))
```

Using procedures and streams defined in this problem, fill in the blanks below to complete Louis's definition of a stream of numbers that approximate π :

```
(define pi-stream  
  (scale-stream  
    4  
    (stream-map-2 /
```

```
      (
```

```
        (stream-map
```

```
          ))
```

```
      (partial-sums
```

```
    ))))
```

Question 12 (16 points)

Suppose we implement a mark-sweep garbage collection method on the memory of a machine. In such a system, the first (or MARK) phase starts at the root and recursively traces out all structure that is accessible from the root. It marks each new cell it gets to, so that if it ever returns to a place by another path, it can avoid tracing out the substructure again. The SWEEP phase scans all of memory, finding all unmarked cells, which it strings together into one large list, called the free-storage list. The pointer to that list is kept in the FREE register. When memory is needed (in CONS, for example) the first cell pointed at by the FREE register is allocated and the FREE register is set to the CDR of that cell. Eventually, the free list becomes empty and the garbage collector is called to make a new free list.

The mark phase of the garbage collector requires a stack to control its recursive operation. This stack cannot be part of the garbage-collected memory, so we allocate a special segment of memory for holding the garbage-collection stack. It is used to save a register with the PUSH instruction. The result is restored with the POP instruction. The following is a simple version of such a garbage collector.

```
garbage-collect
  (assign thing (reg root))
  (assign continue (label sweep))
mark
  (test (op pointer-to-pair?) (reg thing))
  (branch (label mark-pair))
mark-done
  (goto (reg continue))
mark-pair
  (assign mark-flag (op vector-ref) (reg the-marks) (reg thing))
  (test (op =) (reg mark-flag) (const 1))
  (branch (label mark-done))
  (perform (op vector-set!) (reg the-marks) (reg thing) (const 1))
  (push thing)
  (push continue)
  (assign continue (const mark-cdr))
  (assign thing (op vector-ref) (reg the-cars) (reg thing))
  (goto (label mark))
mark-cdr
  (pop continue)
  (pop thing)
  (assign thing (op vector-ref) (reg the-cdrs) (reg thing))
  (goto (label mark))
sweep
  (assign free (const the-empty-list))
  (assign scan (op -) (reg memtop) (const 1))
sweep-loop
  (test (op negative?) (reg scan))
  (branch gc-done)
  (assign mark-flag (op vector-ref) (reg the-marks) (reg scan))
  (test (op =) (reg mark-flag) (const 1))
  (branch (label unmark))
  (perform (op vector-set!) (reg the-cdrs) (reg scan) (reg free))
  (assign free (reg scan))
  (assign scan (op -) (reg scan) (const 1))
  (goto (label sweep-loop))
unmark
  (perform (op vector-set!) (reg the-marks) (reg scan) (const 0))
  (assign scan (op -) (reg scan) (const 1))
  (goto (label sweep-loop))
gc-done
```

Let's examine the behavior of this mark-sweep garbage collector for a small example. Consider, for example, the following initial contents of a tiny memory of 8 pairs (drawn below), where the ROOT register contains the pointer P1.

| Address | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|-----------|----|----|----|----|----|----|----|----|
| the-cars | P3 | P5 | N3 | P0 | P7 | N1 | N4 | N2 |
| the-cdrs | P2 | P2 | P4 | P0 | E0 | P7 | N2 | E0 |
| the-marks | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

ROOT: P8

Part a:

Assume that the MEMTOP register contains a P8. We now start the machine at GARBAGE-COLLECT. Show the state of the list-structured memory when the control reaches the place named SWEEP. Please use the diagram below, but ONLY mark in values that have changed from the diagram above.

| Address | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|-----------|---|---|---|---|---|---|---|---|
| the-cars | | | | | | | | |
| the-cdrs | | | | | | | | |
| the-marks | | | | | | | | |

Part b:

Show the state of the memory and the contents of the FREE register when control reaches GC-DONE. Please ONLY mark those values that have changed from your answer in part a.

| Address | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|-----------|---|---|---|---|---|---|---|---|
| the-cars | | | | | | | | |
| the-cdrs | | | | | | | | |
| the-marks | | | | | | | | |

FREE:

Part c:

What is the maximum number of items that are on the garbage collector stack at any time in running this example? Explain your answer.

Part d:

Assume that our computer has 64,000 pair cells rather than the 8 used in the illustration in the previous parts. Given that PUSH and POP are implemented with a finite auxiliary memory of 100 cells, describe (in a sentence of English) a data structure that will cause the stack to overflow (run out of memory).