MASSACHVSETTS INSTITVTE OF TECHNOLOGY

Department of Electrical Engineering and Computer Science

6.001 -- Structure and Interpretation of Computer Programs

Spring Term 1995

Final Exam

Open Book

Put your name here:____

Put your tutor's name here:______________________________________

Recitation instructor:____

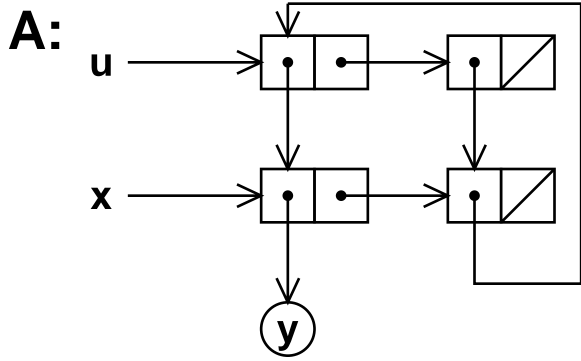
ALSO, PUT YOUR NAME IN THE SPACE PROVIDED AT THE TOP OF EACH PAGE.

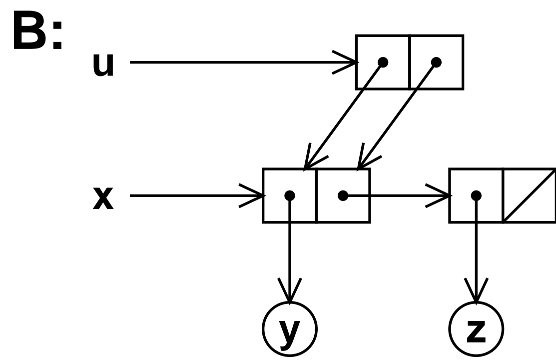
If you have any comments about this exam please write them here:

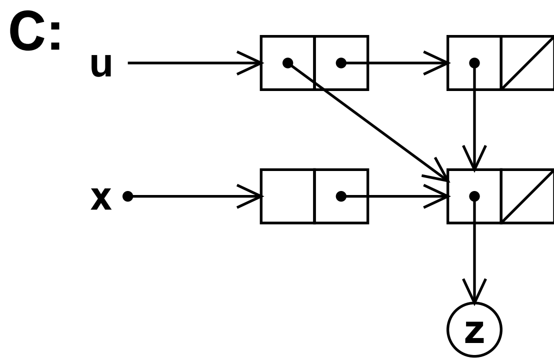
Please do not write below this line -- reserved for administrivia.

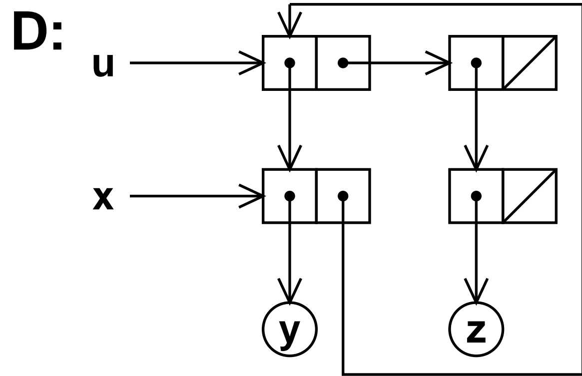
Grader's Comments:

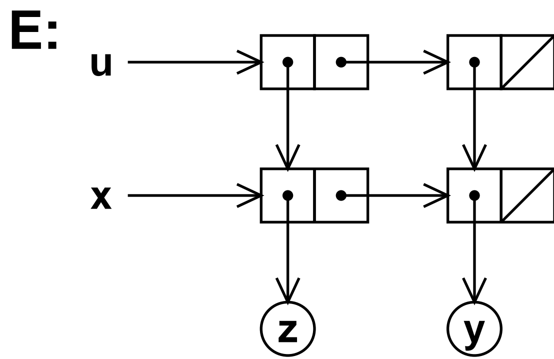
Problem	Grade	Grader
1		
2		
3		
4		
5		
6		
Total		

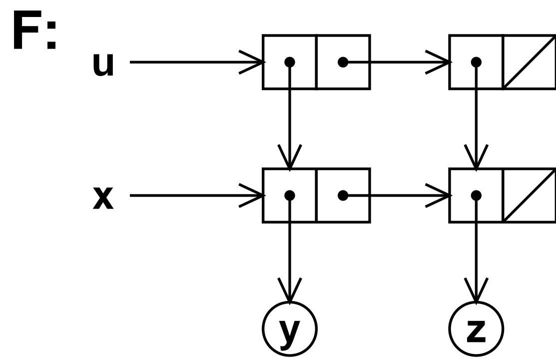

A:
u
x
y

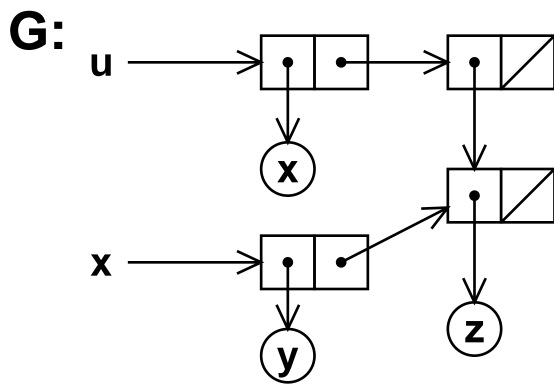

B:
u
x
y
z

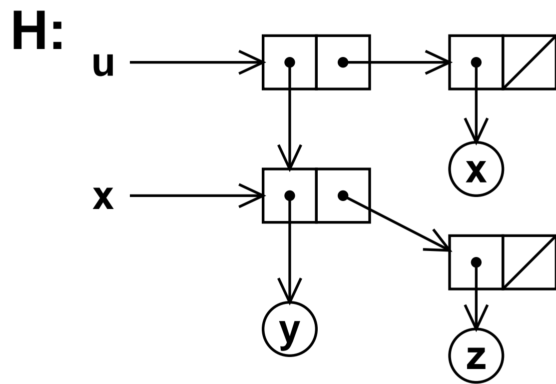

C:
u
x
z


D:
u
x
y
z


E:
u
x
z
y


F:
u
x
y
z


G:
u
x
x
z
y


H:
u
x
x
y
z

 Name:____

Problem 1

A. Match the final values of U and X in each of the cases below with one of the box-and-pointer diagrams (A through H) on the facing page. Note that a diagram may match more than one of these cases or fewer than one case. (Note that the first two lines of each case are the same.)

```
1. (define x (list 'y 'z))
   (define u (list x (cdr x)))
   (set-car! u 'x)

2. (define x (list 'y 'z))
   (define u (list x (cdr x)))
   (set-car! u (cadr u))

3. (define x (list 'y 'z))
   (define u (list x (cdr x)))
   (set-cdr! (car u) u)

4. (define x (list 'y 'z))
   (define u (list x (cdr x)))
   (set-cdr! u x)

5. (define x (list 'y 'z))
   (define u (list x (cdr x)))
   (set-car! (cdr u) 'x)

6. (define x (list 'y 'z))
   (define u (list x (cdr x)))
   (set-car! (cadr u) 'y)
   (set-car! (car u) 'z)
```

Answer:

1.____ 2.____ 3.____

4.____ 5.____ 6.____

B. Begin with the list structures shown in figures A through H, and evaluate (SET! U 9). In each case, how many of the CONS cells in the diagram can be reclaimed by the garbage collector?

Answer:

A.____ B.____ C.____ D.____

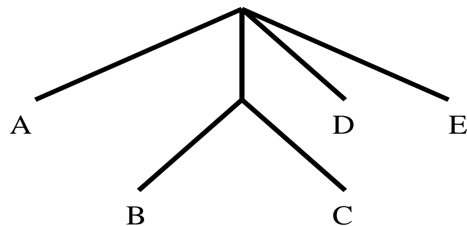
E.____ F.____ G.____ H.____

Problem 2

A. In the space provided below draw a box-and-pointer diagram showing the structure of the list that prints as (A (B C) D E).

Answer:

The fringe of a tree is the sequence of terminal leaf elements of the tree. For example, the list (A (B C) D E) can be thought of as rerpresenting a tree whose terminal leaves are the symbols. The fringe of this tree is the sequence A, B, C, D, E.



A simple way to find a list representing the fringe of a tree that is represented by lists is

```
(define (fringe1 tree)
  (cond ((null? tree) '())
        ((pair? tree)
         (append (fringe1 (car tree))
                 (fringe1 (cdr tree))))
        (else (list tree))))
```

where APPEND is defined as

```
(define (append a b)
  (if (null? a)
      b
      (cons (car a)
            (append (cdr a) b))))
```

For example,

```
(fringe1 '(a (b (c d) e) f (g h)))
;Value: (a b c d e f g h)
```

Note that two distinct trees may have the same fringe:

```
(fringe1 '(a (b c) d e))
;Value: (a b c d e)

(fringe1 '(a b (c d e)))
;Value: (a b c d e)
```

Problem 2 continued

B. Louis left out the NULL? clause in his version of FRINGE1. How would the value of (FRINGE1 '(A (B C) D E)) be printed in Louis's program? Write your answer in the space provided below:

Answer:

It is alleged that FRINGE1 is a rather bad way of computing the fringe of a tree. Consider an alternate strategy:

```
(define (fringe2 tree)
  (define (walk subtree ans)
    (cond ((null? subtree)
           ans)
          ((pair? subtree)
           (walk (car subtree)
                 (walk (cdr subtree) ans)))
          (else
           (cons subtree ans))))
  (walk tree '()))
```

C. In the computation of (FRINGE1 '(A (B C) D E)) how many new CONS cells are made?

Answer:

D. In the computation of (FRINGE2 '(A (B C) D E)) how many new CONS cells are made?

Answer:

Problem 2 continued

It is instructive to consider how to determine if two trees have the same fringe. The simplest way to do this is to test the equality of the lists representing the fringes of the trees. However Ben observes that if we do this we must construct the entire fringes of the two trees in order to compare them for equality, even if the two fringes differ in the first component. Various ideas are considered...

Eventually Ben comes up with the following incremental procedure for getting successive leaves of the tree.

```
(define (fringe-generator tree)
  (define (next)
    (walk tree (lambda () 'done)))
  (define (walk subtree continue)
    (cond ((null? subtree)
           (continue))
          ((pair? subtree)
           (walk (car subtree)
                 (lambda ()
                   (walk (cdr subtree)
                         continue))))
          (else
           (set! next continue)
           subtree)))
  (lambda () (next))                    ;**3**
  )
```

This program is alleged to have the following behavior:

```
(define w (fringe-generator '(a (c b) d e)))
;Value: w

(w)
;Value: a

(w)
;Value: c

(w)
;Value: b

(w)
;Value: d

(w)
;Value: e

(w)
;Value: done
```

Problem 2 continued

E. Louis says that the line **3** is clearly too complicated. He says that since NEXT is a procedure that takes zero arguments then (LAMBDA () (NEXT)) = NEXT, so we should replace that line with the equivalent simpler expression. Circle all of the statements that are true of Louis's suggestion.

1. The whole FRINGE-GENERATOR program is garbage and could not work with or without Louis's suggested modification.

2. The identity that Louis used to justify his suggestion may be wrong in the presence of assignments.

3. Louis's patch accidentally fixes an otherwise fatal bug in the program.

4. Louis's patch breaks the program so that the modified program only works for lists with no more than two levels of nesting.

5. The program works as advertised without Louis's patch, but Louis's patch makes the program fail.

6. With Louis's patch the call to the generator (the procedure W above) always returns the same result.

F. Assuming that Ben's FRINGE-GENERATOR is correct, we can now write an efficient SAMEFRINGE? procedure which returns #T when the fringes of its arguments are the same.

```
(samefringe? '(a (b c) d e) '(a b (c d e)))
;Value: #t

(samefringe? '(a (c b) d e) '(a b (c d e)))
;Value: #f

(define (samefringe? tree1 tree2)
  (let ((walk1 (fringe-generator tree1))
        (walk2 (fringe-generator tree2)))
    (define (compare leaf1 leaf2)
      (cond ((eq? leaf1 'done) (eq? leaf2 'done))
            ((eq? leaf2 'done)
             <**missing-part1**>)
            ((eq? leaf1 leaf2)
             <**missing-part2**>)
            (else #f)))
    (compare (walk1) (walk2))))
```

In the spaces provided below write the expressions that belong in the places marked as missing.

<**missing-part1**>:

<**missing-part2**>:

Problem 3

Programmers often need a way to measure the resources used in a procedure. For example, it is often useful to collect statistics on the total amount of time that a particular procedure takes in all of its uses in some big system. The following procedure takes a procedure of one argument and returns a new, instrumented procedure of one argument that records the cumulative running time used in the original procedure. Assume that applying the PROCESS-RUNTIME procedure returns the current value, in seconds, of the approximate time that the calling process has run.

```
(define timed
  (lambda (f)                              ;**1**
    (let ((accum 0))                       ;**2**
      (lambda (arg)                        ;**3**
        (cond ((eq? arg 'get-time)
               ;;                           **4**
               accum)
              ((eq? arg 'clear-time)
               (set! accum 0))
              (else
               (let ((start (process-runtime)))
                 (let ((value (f arg)))
                   (let ((end (process-runtime)))
                     ;;                     **5**
                     (set! accum
                           (+ (- end start)
                              accum)))
                   value))))))))
```

Problem 3 continued

So, for example, we could make a timed version of the familiar extremely inefficient tree-recursive procedure for Fibonacci numbers:

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1))
         (fib (- n 2)))))

(define ifib (timed fib))

(ifib 20)
;Value: 6765

(ifib 'get-time)                   ;IFIB is the timed version of FIB
;Value: 7.53

(ifib 20)
;Value: 6765

(ifib 'get-time)
;Value: 14.88

(ifib 'clear-time)

(ifib 21)
;Value: 10946

(ifib 'get-time)
;Value: 12.31
```

A. Using the measurements recorded above, circle the best estimate of the time it would take to compute (ifib 22).

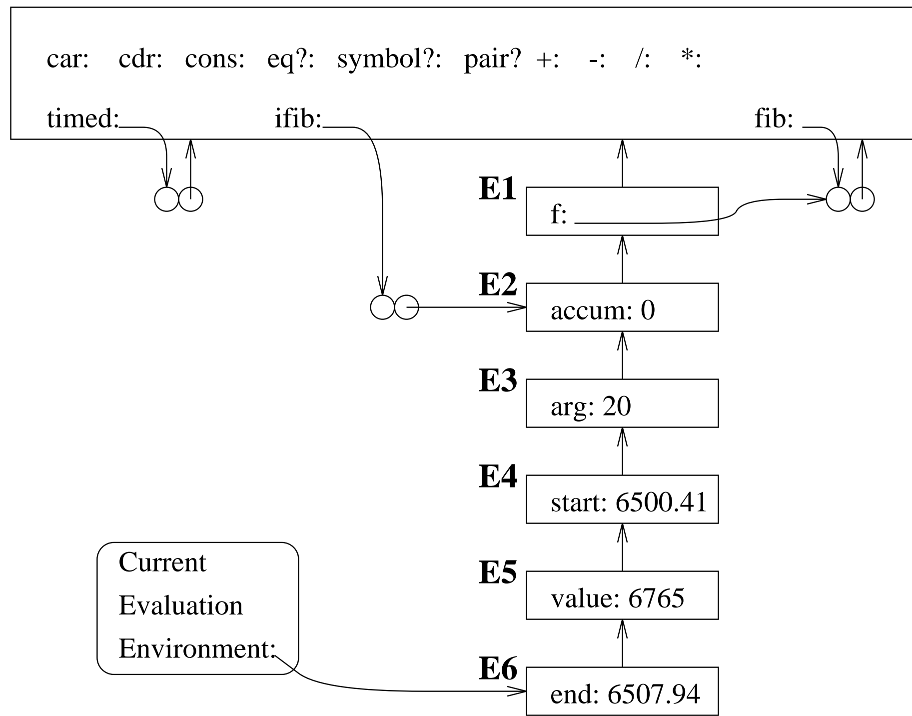
Answer: 9.1 15.2 19.7 23.7 52.6

B. The environment diagram on the facing page shows the state of the evaluator at the place marked **5** in the first evaluation of the expression (IFIB 20) in the sequence above.

1. Which of the frames in the diagram may be reclaimed after the evaluation of the (IFIB 20) expression returns because they will never again be needed? Circle the useless frames in the following list.

Answer: E0 E1 E2 E3 E4 E5 E6

2. After the (IFIB 20) evaluation is completed the sequence goes on to ask for the result of (IFIB 'GET-TIME). Draw, on the given diagram, the additions to the environment that must be made to represent the state of the evaluator when it gets to the place marked **4** in this evaluation. Make sure you redirect the current evaluation environment arrow to point at the correct frame for this situation.


E0
car: cdr: cons: eq?: symbol?: pair? +: -: /: *:
timed:
ifib:
fib:
E1
f:
E2
accum: 0
E3
arg: 20
E4
start: 6500.41
E5
value: 6765
E6
end: 6507.94
Current
Evaluation
Environment:

Problem 3 continued

C. Circle the statements that correctly describe the changed behavior of the TIMED program if we interchange the lines labeled **1** and **2**. Assume no CLEAR-TIME requests are made.

1. The time reported by any timed procedure would always be zero.

2. The time reported by any timed procedure would be the time accumulated in the execution of that procedure.

3. The time reported by any timed procedure would be the time accumulated in the execution of all timed procedures.

4. The time reported by any timed procedure would only be accurate if the underlying procedure were not recursive.

5. The time reported by any timed procedure would be the same as the time reported by any other timed procedure if it were queried at that instant instead.

D. Circle the statements that correctly describe the changed behavior of the TIMED program if we interchange the lines labeled **2** and **3**. Assume no CLEAR-TIME requests are made.

1. The time reported by any timed procedure would always be zero.

2. The time reported by any timed procedure would be the time accumulated in the execution of that procedure.

3. The time reported by any timed procedure would be the time accumulated in the execution of all timed procedures.

4. The time reported by any timed procedure would only be accurate if the underlying procedure were not recursive.

5. The time reported by any timed procedure would be the same as the time reported by any other timed procedure if it were queried at that instant instead.

Problem 3 continued

E. Suppose we had concurrent processes running on our system, and we wanted to accumulate the time used in IFIB by all processes. That is, if one process uses IFIB for approximately 100 seconds and another uses IFIB for approximately 72, seconds we want IFIB to report that it has been used for approximately 172 seconds. Explain, in the space provided below, why this might be a problem and what you might do to fix it. You may illustrate your answer with a timing diagram and patches to the TIMED program.

Answer:

Problem 4

A. As long as we are working with evaluators that support lazy parameters, as in Problem Set 8, it might seem that there is a very simple implementation of a lazy CONS, namely

```
(define (lazy-cons (x lazy) (y lazy))
  (cons x y))
```

This idea doesn't quite work because (circle ALL the correct explanations):

1. Applying the primitive CONS will force the delayed arguments prematurely.

2. CONS is by nature lazy and thus does not need to be made lazy.

3. It is necessary to memoize at least the CDR.

4. The forcing procedure will not be applicable in this situation.

5. Compiled procedures cannot be made to handle the delayed arguments.

An alternative idea is to make a special form that is a lazy version of CONS. To do this, we first specify the syntax

```
(define (lazy-cons-form? exp)
  (tagged-list? exp 'lazy-cons-form))
```

Then, in the metacircular evaluator with lazy parameter declarations (from Problem Set 8) we add an extra case to the MEVAL dispatch on expression type:

```
(define (meval exp env)
  (cond ((self-evaluating? exp) exp)
         ...
        ((lazy-cons-form? exp) (eval-lazy-cons-form exp env))   ;***
         ...
        (else (error "Unknown exp MEVAL" exp))))
```

and include in the evaluator the procedure EVAL-LAZY-CONS-FORM:

```
(define (eval-lazy-cons-form exp env)
  (cons (cons *lazythunk*
              (lambda () (meval (car (operands exp)) env)))
        (cons *lazythunk*
              (lambda () (meval (cadr (operands exp)) env)))))
```

Problem 4 continued

With this modification to the lazy evaluator we can make, for example, an infinite stream of 1's:

```
MEVAL=> (define ones (lazy-cons-form 1 ones))
```

and then access any of the 1's in this list:

```
MEVAL=> (car (cdr (cdr (cdr ones))))
;;M-value: 1
```

B. What will be the printed for <???> below (circle the ONE correct description):

```
MEVAL=> ones
;;M-value: <???>
```

1. output of the form "(1 . <something>)".
2. no output until an "out-of-memory" error.
3. "(" followed by an unending sequence of 1's.
4. output of the form "(1 (<something>))".
5. a printed representation of a finite list structure whose leaves are either the symbol *LAZYTHUNK* or the number 1.
6. a printed representation of a finite list structure whose leaves either the symbol *LAZYTHUNK* or procedures.
7. a printed representation of a thunk.
8. a printed representation of a finite list structure whose leaves are either the number 1 or procedures.
9. a printed representation of a finite list structure whose leaves are either the number 1, the symbol *LAZYTHUNK*, or procedures.

Problem 4 continued

But the job is not quite done. For example, we can define in the lazy language the ACCUMULATE procedure of Chapter 2:

```
 MEVAL=> (define (accumulate op init seq)
           (if (null? seq)
               init
               (op (car seq)
                   (accumulate op init (cdr seq)))))
```

so that, for example

```
 (accumulate + 0 '(1 2 3))       ==>   6
 (accumulate cons '() '(1 2 3)) ==>  (1 2 3)
```

C. Evaluating the following expression in the lazy evaluator causes trouble:

```
                MEVAL=> (accumulate lazy-cons-form '() '(1 2 3))
```

The problem is (circle ALL the correct explanations):

1. The empty lazy list is not necessarily EQ? to the empty list.

2. Special forms are not first-class objects.

3. LAZY-CONS-FORM is not bound to a procedure.

4. LAZY-CONS-FORM can only be applied to lazy arguments.

5. The accumulation using LAZY-CONS-FORM is no different from the one with CONS.

6. Quoted lists are not compatible with the lazy evaluator.

Problem 4 continued

D. We can achieve the desired lazy cons behavior by defining a LAZY-CONS procedure and using it instead of LAZY-CONS-FORM thereafter, as in

```
MEVAL=> (car (cdr (accumulate lazy-cons '() '(1 2 3))))
;;M-value: 2
```

Circle the ONE definition of LAZY-CONS below which is appropriate:

```
1.  (define (lazy-cons x y) (lazy-cons-form x y))

2.  (define (lazy-cons x (y lazy)) (lazy-cons-form x y))

3.  (define (lazy-cons (x lazy) (y lazy)) (lazy-cons x y))

4.  (define (lazy-cons (x lazy) (y lazy)) (cons x y))

5.  (define (lazy-cons (x lazy-memo) (y lazy-memo)) (cons x y))

6.  (define (lazy-cons (x lazy) (y lazy)) (delay (lazy-cons x y)))

7.  (define (lazy-cons x (y lazy)) (delay (cons x y)))

8.  (define (lazy-cons (x lazy) (y lazy)) (delay (cons x y)))

9.  (define (lazy-cons (x lazy) (y lazy)) (cons (delay x) (delay y)))
```

Problem 4 continued

Now that we have installed a lazy cons procedure in the metacircular evaluator we can follow the same strategy to install it in the analyzing evaluator as well. First, we add to the ANALYZE dispatch:

```
 (define (analyze exp)
   (cond ((self-evaluating? exp) (analyze-self-evaluating exp))
          ...
         ((lazy-cons-form? exp) (analyze-lazy-cons-form exp))  ;***
          ...
         (else (error "Unknown expression ANALYZE" exp))))
```

and then we include in the evaluator the procedure ANALYZE-LAZY-CONS-FORM.

E. Complete the definition of the procedure ANALYZE-LAZY-CONS-FORM in the space provided below:

Answer:

```
 (define (analyze-lazy-cons-form exp)
```

```
1      (save continue)
2      (assign argl (op empty-arglist))
3      (assign continue (label after-rand7))
4      (save env)
5      (save argl)
6      (assign val (const assignment))
7      (goto (reg continue))
   after-rand7
8      (restore argl)
9      (assign argl (op adjoin-arg) (reg val) (reg argl))
10     (restore env)
11     (assign continue (label after-rator2))
12     (save argl)
13     (assign val (op make-compiled-procedure) (label proc-entry3) (reg env))
14     (goto (reg continue))
   proc-entry3
15     (assign env (op extend-environment) (const (instruction)) (reg argl) (reg
16     (save continue)
17     (restore continue)
18     (save continue)
19     (assign argl (op empty-arglist))
20     (assign continue (label after-rand6))
21     (save env)
22     (save argl)
23     (assign val (op lookup-variable-value) (const instruction) (reg env))
24     (goto (reg continue))
   after-rand6
25     (restore argl)
26     (assign argl (op adjoin-arg) (reg val) (reg argl))
27     (restore env)
28     (assign continue (label after-rator5))
29     (save argl)
30     (assign val (op lookup-variable-value) (const list) (reg env))
31     (goto (reg continue))
   after-rator5
32     (restore argl)
33     (restore continue)
34     (test (op primitive-procedure?) (reg val))
35     (branch (label primitive-apply4))
36     (assign env (op compiled-procedure-env) (reg val))
37     (assign val (op compiled-procedure-entry) (reg val))
38     (goto (reg val))
   primitive-apply4
39     (assign val (op apply-primitive-procedure) (reg val) (reg argl))
40     (goto (reg continue))
   after-rator2
41     (restore argl)
42     (restore continue)
43     (test (op primitive-procedure?) (reg val))
44     (branch (label primitive-apply1))
45     (assign env (op compiled-procedure-env) (reg val))
46     (assign val (op compiled-procedure-entry) (reg val))
47     (goto (reg val))
   primitive-apply1
48     (assign val (op apply-primitive-procedure) (reg val) (reg argl))
49     (goto (reg continue))
```

Problem 5

The following question refers to code generated by the NAIVE-COMPILE procedure of Problem Set 10. The compiled code is on the facing page, with line numbers added for reference.

A. Decompiling.

1. Lines 23-24 are the result of naively compiling what subexpression?
Answer:

2. Lines 30-31 are the result of naively compiling what subexpression?
Answer:

3. Lines 18-40 are the result of naively compiling what subexpression?
Answer:

4. Lines 13-40 are the result of naively compiling what subexpression?
Answer:

5. Lines 6-7 are the result of naively compiling what subexpression?
Answer:

6. Lines 1-49 are the result of naively compiling what expression?
Answer:

B. Assume that the standard Scheme primitives (+ - ... LIST CONS ...) have not been redefined. Circle the numbers of five lines from the compiled code will be skipped when the code is executed.

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18

19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36

37 38 39 40 41 42 43 43 44 45 46 47 48 49

C. Naive compiler code is very inefficient. In fact, over forty of the lines from the compiled code could safely be deleted. Circle the numbers of three lines other than (GOTO (REG CONTINUE)) which canNOT be deleted.

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18

19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36

37 38 39 40 41 42 43 43 44 45 46 47 48 49

D. Explain in one sentence why a (GOTO (REG CONTINUE)) instruction would have to remain undeleted.