MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 2003

Final Exam – Example solutions

Closed Book – three sheets of notes

Separately, we have distributed an answer sheet. You may use the space on the exam booklet for whatever temporary work you find useful, but you **MUST** enter your answers into the answer sheet. **Only this will be graded.** Each problem that requires an answer has been numbered. Place your answer at the corresponding number in the answer sheet.

Your code will be judged on clarity and good programming practice, as well as correct function.

The answer sheet asks for your section number and tutor, use the chart below.

Section	Time	Location	Rec. Instructor	Tutors
1	9:00	26-314	Jovan Popovic	Toh Ne Win
2	10:00	34-301	Konrad Tollmar	Matt Peters
3	10:00	26-314	Jovan Popovic	Limor Fried
4	10:00	34-304	Robert Miller	Adnan Dosani
5	11:00	34-301	Konrad Tollmar	Matt Peters Limor Fried Alex Andersson
6	2:00	26-204	Robert Miller	Todd Atkins Aaron Strauss
7	12:00	34-301	Michael Collins	Alex Andersson
8	1:00	34-303	Tony Eng	Aaron Strauss
9	1:00	34-30	Michael Collins	Samitha Samaranayake
10	2:00	34-303	Tony Eng	Todd Atkins
11	11:00	26-314	Fredo Durand	Dave Feinberg
12	12:00	26-314	Fredo Durand	Aaron Strauss Dave Feinberg Adnan Dosani Samitha Samaranayake

No grades for 6.001 will be available until after 10 AM on Tuesday, May 27. After that time, you may (1) wait until grades are mailed by the Registrar in June; (2) access your grades via WEBSIS (http://student.mit.edu/), or (3) contact the course secretary, Jill Fekete, in NE46-606. No grades will be given out by phone; you must appear in person with ID if selecting the third option.

Completed final exams will not be handed back to students, but will be available starting on May 27 for students to look at (but not remove from) Jill Fekete's office, Room NE46-606 (400 Technology Square).

Part 1 (15 points):

Each of the parts below should be treated as independent. For each part, a sequence of expressions is given, which you may assume is typed into 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. If the sequence results in a error, write **error** and indicate the kind of error. If the final expression returns a procedure, write `procedure` and indicate the type of the procedure.

Question 1:

```
((lambda (x + y) (+ y x)) 2 / 4)
```

2

Question 2:

```
(define f
   (lambda (f)
     (lambda (n)
       (n f))))

((f 10) (lambda (x) (* x 2)))
```

20

Question 3:

```
(define s
  (cons-stream 1
     (stream-map (lambda (x) (* 2 x)) s)))

(define (add-streams s1 s2)
  (cons-stream (+ (stream-car s1) (stream-car s2))
               (add-streams (stream-cdr s1) (stream-cdr s2))))

(stream-car
  (stream-cdr
    (stream-cdr
      (add-streams s (stream-cdr (stream-cdr s))))))
```

20

Question 4:

```
(define a (list 'a 'b))

(define b (list 'a a 'b))

(set-cdr! (cdr a) (cddr b))

b
```

(a (a b b) b)

Question 5:

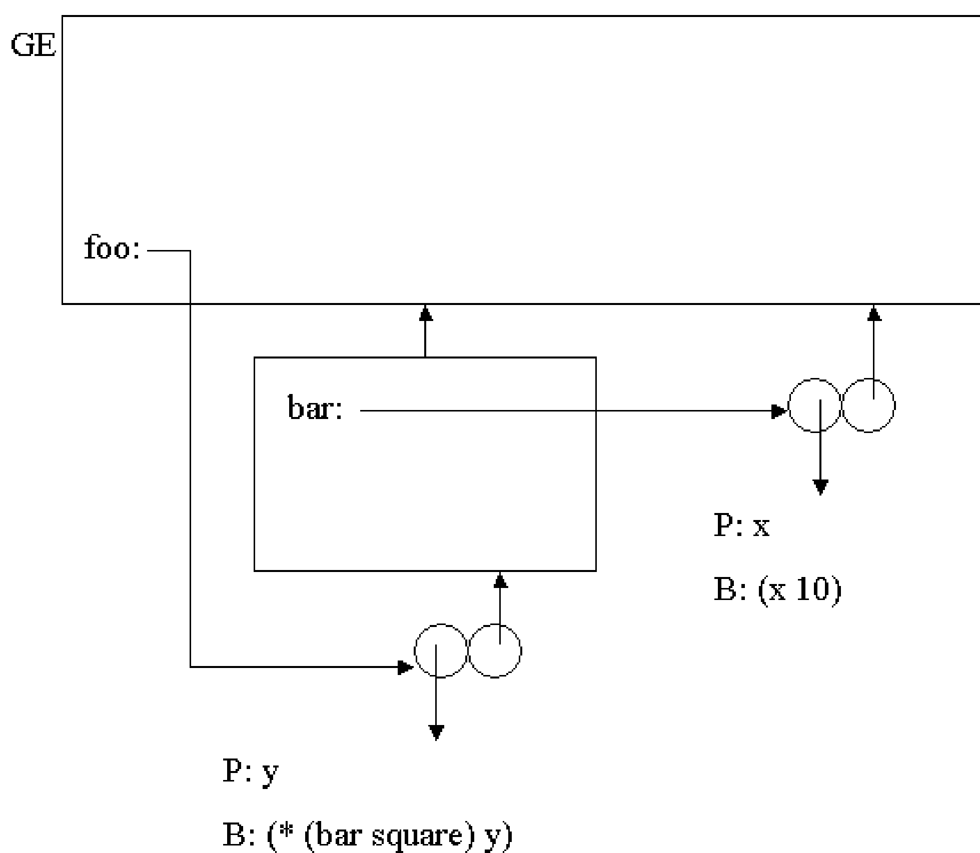
```
(let ((temp expt))
  (lambda (n)
    (lambda (x) (temp x n))))
```

procedure: number $\mapsto$ (number $\mapsto$ number)

Part 2: (15 points)

Below is an environment diagram.

Question 6: Write a single Scheme expression, which does not use mutation, whose evaluation would give rise to this diagram. Assume that garbage collection has removed all unreferenced "double bubbles" and frames.



Part 3 (33 points):

We want to add a new special form to our Meta-circular evaluator (a copy of which is attached at the end of the exam). This special form is a `do` loop, an example of which is shown below:

```
(do ((i 0 inc 10)
     (j x (lambda (x) (+ x 2)) (check i)))
  (do-something i)
  (do-something-else j))
```

Here is how a `do` evaluates. The first subexpression in a `do` is a set of one or more clauses. Each clause consists of a *loop variable* (i and j in this case); an initial value for each variable (0 and `x` in this case); a procedure to apply to increment the loop variable (`inc` and `(lambda (x) (+ x 2))` in this case); and an end value. For simplicity, we will restrict ourselves to loop variables that only take on integer values. The format is that each loop variable assumes its initial value (evaluated in the current environment). Then each variable is tested to see if it is greater than or equal to its end value (the end value is determined by evaluating the end test expression relative to the current values of the variables). If **any** variable passes this test, the `do` terminates and returns `#t`. Otherwise, the body of the `do` is evaluated – this is the set of expressions after the variable clauses; note that there can be more than one expression. Also note that evaluation is done relative to the current values of the variables. Then each variable is incremented by using its procedure to compute a new value, and the process repeats.

Assume that we have added the following expressions to our syntax:

```
(define (do? exp) (tagged-list exp 'do))

(define do-clauses cadr)
(define do-body cddr)
```

and that the following clause has been added to `m-eval`:

```
((do? exp)
 (ev-do exp env))
```

Finally, we have

```
(define (ev-do exp env)
  (ev-do-helper (do-clauses exp)
                (do-body exp)
                env))
```

We want you to complete this addition of a special form by completing the definition for `ev-do-helper`

```
(define (ev-do-helper clauses body env)
  (let ((new-env (extend-environment
                   (map QUESTION-7 clauses)
                   (map QUESTION-8 clauses)
                   QUESTION-9)))
    (if (m-eval (or-clauses clauses) new-env)
        #t
        (begin (eval-sequence body new-env)
               (ev-do-helper
                   (map QUESTION-10 clauses)
                   QUESTION-11
                   QUESTION-12)))))
```

The basic idea is that we will create a new environment with the loop variables bound to their initial values. Relative to that environment we will determine if any of the end tests are true, in which case we will return the symbol `done`. If not, we evaluate the body of the `do` and then repeat the process by creating a new set of clauses and calling the helper procedure again.

Complete the following questions.

Question 7: What expression should be used for `QUESTION-7`?

(map car clauses)

Question 8: What expression should be used for `QUESTION-8`?

(map (lambda (x) (m-eval (cadr x) env)) clauses)

Question 9: What expression should be used for `QUESTION-9`?

env

Question 10: What expression should be used for `QUESTION-10`?

```
(map (lambda (c) (list (car c)
                       (list (caddr c) (cadr c))
                       (caddr c)
                       (cadddr c)))
     clauses)
```

Question 11: What expression should be used for `QUESTION-11`?

body

Question 12: What expression should be used for `QUESTION-12`?

new-env

Finally, we need to convert the collection of end tests into a single `or` expression (in the example above, this would be `(or (>= i 10) (>= j (check i)))`. Here is a template for `or-clauses`:

```
(define (or-clauses cls)
  (define (loop cls)
     (if (null? cls)
         QUESTION-13
         QUESTION-14))
 (cons 'or (loop cls)))
```

Question 13: What expression should be substituted for `QUESTION-13`?

'()

Question 14: What expression should be substituted for `QUESTION-14`?

```
 (cons (list '>= (car (car cls)) (caddr (car cls)))
       (loop (cdr cls)))
```

Part 4 (21 points)

You are going to write register machine code to compute the mean or average of a list of numbers.

Question 15: First, write an **iterative** procedure to compute the length of a list. Assume that the register `arg` holds the list to be measured, that the result is to be returned in the register `val`, that the continuation is in the `continue` register, and that the label `length` marks the beginning of the code. Use the register `temp` to hold any temporary values. You may assume that `car, cdr, cons, null?, +, -, *, /, =, >, <` are built in primitive procedures. You may assume that `length` can clobber the contents of `arg` and that any other procedure that uses `length` as a subprocedure will first save the contents of `arg`.

```
length
  (assign val 0)
loop
  (test (op null?) (reg arg))
  (branch (reg continue))
  (assign val (op +) (reg val) (const 1))
  (assign arg (op cdr) (reg arg))
  (goto (label loop))
```

Next, we want to write a piece of register machine code for computing the sum of a list of numbers. Assume that `arg` holds the list of numbers (you may assume that the sum of an empty list of numbers is 0), that `continue` holds the continuation, that `temp` is available for holding temporary values, and that the result is to be stored in `val`. Remember that the convention of register machines is that a value may only be restored into the **same** register from which it was saved.

We want to write a **recursive** procedure using register machine code. Here is a template for the code:

```
sum
  (test (op null?) (reg arg))
  (branch (label end-list))
  QUESTION-16
  (assign continue (label do-sum))
  (assign temp (op car) (reg arg))
  (save temp)
  (assign arg QUESTION-17)
  (goto QUESTION-18)
end-list
  (assign val QUESTION-19)
  (goto QUESTION-20)
do-sum
  (restore temp)
  QUESTION-21
  (assign val (op +) (reg temp) (reg val))
  (goto QUESTION-22)
```

Question 16: What piece of register machine code should be used in place of `QUESTION-16`?

(save continue)

Question 17: What piece of register machine code should be used in place of `QUESTION-17`?

(op cdr) (reg arg))

Question 18: What piece of register machine code should be used in place of `QUESTION-18`?

(label sum)

Question 19: What piece of register machine code should be used in place of `QUESTION-19`?

(const 0)

Question 20: What piece of register machine code should be used in place of `QUESTION-20`?

(reg continue)

Question 21: What piece of register machine code should be used in place of `QUESTION-21`?

(restore continue)

Question 22: What piece of register machine code should be used in place of `QUESTION-22`?

(reg continue)

Finally, we want to use the code we have written above to compute the mean or average of a list.

Assume that `arg` holds the list of numbers (you may assume that the list is not empty), that `continue` holds the continuation, that `temp` is available for holding temporary values, and that the result is to be stored in `val`. Remember that the convention of register machines is that a value may only be restored into the **same** register from which it was saved.

Here is a template:

```
mean
  (save continue)
  (assign continue (label after-sum))
  (save arg)
  (goto (label sum))
after-sum
  QUESTION-23
  (save val)
  (assign continue (label after-length))
  (goto (label length))
after-length
  QUESTION-24
  QUESTION-25
  (assign val QUESTION-26)
  (restore continue)
  (goto (reg continue))
```

Question 23: What piece of register machine code should be used in place of `QUESTION-23`?

(restore arg)

Question 24: What piece of register machine code should be used in place of `QUESTION-24`?

(assign temp (reg val))

Question 25: What piece of register machine code should be used in place of `QUESTION-25`?

(restore val)

Question 26: What piece of register machine code should be used in place of `QUESTION-26`?

(op /) (reg val) (reg temp))

Part 5 (16 points)

We have seen that one way to add new forms to our evaluator is to use syntactic transformations. For example, rather than adding a procedure to handle evaluation of `cond` expressions, we can first convert a `cond` into a set of `if`s, and then rely on the evaluator's built in capability to handle `if` expressions to do the work. Thus to `m-eval` we might add

```
((cond? exp)
 (m-eval (cond->if exp) env))
```

Suppose we have the following data abstraction for handling `cond` expressions and their components:

```
(define clauses cdr)
(define first-clause car)
(define rest-clauses cdr)
(define no-clauses? null?)
(define predicate car)
(define consequents cdr)
```

Assume that there is at least one consequent in each clause. If evaluation reaches the end of clauses with no true predicate, the entire expression should return `#f`. Be sure that your code handles the `else` clause properly. Also be sure to use appropriate data abstractions from the Meta-circular evaluator (attached at the end of the exam).

Here is a template for the syntactic transformation:

```
(define (cond->if exp)
  (aux (clauses exp)))

(define (aux cls)
  (if (no-clauses? cls)
      QUESTION-27
      (if (eq? (predicate (first-clause cls)) 'else)
          QUESTION-28
          QUESTION-29)))
```

Question 27: What expression should be used in place of `QUESTION-27`?

'()

Question 28: What expression should be used in place of `QUESTION-28`?

(cons 'begin (consequents (first-clause cls)))

Question 29: What expression should be used in place of `QUESTION-29`?

```
(list 'if
      (predicate (first-clause cls))
      (cons 'begin (consequents (first-clause cls)))
      (aux (rest-clauses cls)))
```

Part 6 (30 points)

Consider the following general procedure for processing trees:

```
(define (tree-manip car-op leaf-op merge init tree)
  (if (null? tree)
      init
      (if (leaf? tree)
          (leaf-op tree)
          (merge (car-op (car tree))
                 (tree-manip car-op leaf-op merge init (cdr tree))))))
```

You may assume that `leaf?` is a predicate that returns true when applied to a leaf of a tree.

Suppose that we define

```
(define test-tree '(1 (2 (3 (4) 5) 6) 7))
```

Question 30:

Complete the code to create a procedure that will compute the breadth of a tree, defined as the number of branches at the top-most level of the tree, e.g.

```
(breadth test-tree)
;Value: 3
```

```
(define (breadth tr)
  (tree-manip QUESTION-30 0 tr))
```

```
(define (breadth tree) (tree-manip (lambda (x) 1) (lambda (x) 1) + 0 tree))
(define (breadth tree) (tree-manip (lambda (x) 1) (lambda (x) x) + 0 tree))
```

Question 31:

Complete the code to create a procedure that will compute the depth of a tree, defined as the maximum number of elements between the top level of the tree, and a leaf of the tree, e.g.

```
(depth test-tree)
;Value: 4
```

```
(define (depth tr)
  (tree-manip QUESTION-31 0 tr))
```

```
(define (depth tree) (tree-manip
   (lambda (x) (lambda (x) (+ 1 (depth x))) (lambda (x) 0) max 0 tree))
```

Question 32:

Complete the code to create a procedure that will reverse the top level of a tree, e.g.

```
(reverse test-tree)
;Value: (7 (2 (3 (4) 5) 6) 1)
```

```
(define (reverse tr)
  (tree-manip QUESTION-32 nil tr))
```

```
(define (reverse tree) (tree-manip (lambda (x) (list x)) (lambda (x) (list x))
  (lambda (x y) (append y x)) '() tree)
```

Question 33:

Complete the code to create a procedure that will deep reverse a tree, e.g.

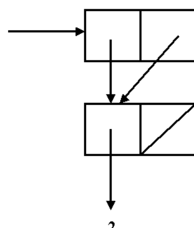
```
(deep-reverse test-tree)
;Value: (7 (6 (5 (4) 3) 2) 1)
```

```
(define (deep-reverse tr)
  (tree-manip QUESTION-33 nil tr))
```

```
(define (deep-reverse tree)
  (tree-manip (lambda (x) (list (deep-reverse x)) (lambda (x) x)
              (lambda (x y) (append y x))
 '() tree))
```

Part 7 (16 points)

Consider the following box and pointer diagram:



Question 34. What does this print as?

((2) 2)

For each of the following expressions, indicate by selecting **true** or **false**, whether the expression gives rise to the box-and-pointer structure shown above.

Question 35.

```
'((2) 2)
```

F

Question 36.

```
(list (list 2) 2)
```

F

Question 37.

```
(let ((temp (list 2)))
  (cons temp temp))
```

T

Question 38.

```
(let ((temp (list 2)))
  (let ((foo (cons (list 2) temp)))
    foo))
```

F

Question 39.

```
(let ((temp (list 2)))
  (let ((foo (cons (list 2) temp)))
    (set-car! foo temp)
    foo))
```

T

Question 40.

```
(let ((temp (list 2)))
  (let ((foo (cons (list 2) temp)))
    (set! (car foo) temp)
    foo))
```

F

Part 8 (16 points):

The following code fragment was generated using the Scheme compiler.

Question 41: What Scheme expression was compiled to generate line 11?

a

Question 42: What Scheme expression was compiled to generate lines 8–21?

(= a 0)

Question 43: What Scheme expression was compiled to generate lines 36–47?

(test a)

Question 44: What Scheme expression was compiled to generate lines 30–60?

(other (test a) b)

Question 45: What Scheme expression was compiled to generate lines 8–61?

(if (= a 0) b (other (test a) b))

Question 46: What Scheme expression was compiled to generate lines 1–62?

(lambda (a b) (if (= a 0) b (other (test a) b)))

Question 47: What Scheme expression was compiled to generate the entire set of instructions?

(define (other a b) (if (= a 0) b (other (test a) b)))

```
1.    (assign val (op make-compiled-procedure) (label entry2) (reg env))
2.    (goto (label after-lambda1))
3. entry2
4.    (assign env (op compiled-procedure-env) (reg proc))
5.    (assign env (op extend-environment) (const (a b)) (reg argl) (reg env))
6.    (save continue)
7.    (save env)
8.    (assign proc (op lookup-variable-value) (const =) (reg env))
9.    (assign val (const 0))
10.   (assign argl (op list) (reg val))
11.   (assign val (op lookup-variable-value) (const a) (reg env))
12.   (assign argl (op cons) (reg val) (reg argl))
13.   (test (op primitive-procedure?) (reg proc))
14.   (branch (label primitive-branch14))
15. compiled-branch13
16.   (assign continue (label after-call12))
17.   (assign val (op compiled-procedure-entry) (reg proc))
18.   (goto (reg val))
19. primitive-branch14
20.   (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
21. after-call12
22.   (restore env)
23.   (restore continue)
24.   (test (op false?) (reg val))
25.   (branch (label false-branch4))
26. true-branch5
```

```
27.   (assign val (op lookup-variable-value) (const b) (reg env))
28.   (goto (reg continue))
29. false-branch4
30.   (assign proc (op lookup-variable-value) (const other) (reg env))
31.   (save continue)
32.   (save proc)
33.   (assign val (op lookup-variable-value) (const b) (reg env))
34.   (assign argl (op list) (reg val))
35.   (save argl)
36.   (assign proc (op lookup-variable-value) (const test) (reg env))
37.   (assign val (op lookup-variable-value) (const a) (reg env))
38.   (assign argl (op list) (reg val))
39.   (test (op primitive-procedure?) (reg proc))
40.   (branch (label primitive-branch8))
41. compiled-branch7
42.   (assign continue (label after-call6))
43.   (assign val (op compiled-procedure-entry) (reg proc))
44.   (goto (reg val))
45. primitive-branch8
46.   (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
47. after-call6
48.   (restore argl)
49.   (assign argl (op cons) (reg val) (reg argl))
50.   (restore proc)
51.   (restore continue)
52.   (test (op primitive-procedure?) (reg proc))
53.   (branch (label primitive-branch11))
54. compiled-branch10
55.   (assign val (op compiled-procedure-entry) (reg proc))
56.   (goto (reg val))
57. primitive-branch11
58.   (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
59.   (goto (reg continue))
60. after-call9
61. after-if3
62. after-lambda1
63.   (perform (op define-variable!) (const other) (reg val) (reg env))
64.   (assign val (const ok))
65.   (goto (reg continue))
```

Part 9 (15 points)

In project 4, we saw the idea of a continuation as a means of controlling evaluation. For example, here is factorial written in continuation style:

```
(define (fact n cont)
  (if (= n 1)
      (cont 1)
      (fact (- n 1) (lambda (x) (cont (* n x)) ))))
```

so that

```
(define id (lambda (x) x))

(fact 3 id)
;Value: 6
```

Here is fibonacci, written the traditional way (remember that fibonacci of 1 is 1, of 2 is 1 and otherwise is the sum of the previous two fibonacci numbers):

```
(define (fib n)
  (if (or (= n 1) (= n 2))
      1
      (+ (fib (- n 1)) (fib (- n 2)))))
```

Question 48: Complete the following definition so that `fib` is written in continuation style.

```
(define (fib n cont)
  (if (or (= n 1) (= n 2))
      (cont 1)
      QUESTION-48))
```

(fib (- n 1) (lambda (f1) (fib (- n 2) (lambda (f2) (cont (+ f1 f2))))))

Part 10 (18 points)

In a mark-sweep garbage collector, the MARK phase recursively traces all structure accessible from the root. It marks each new cell it reaches, so that it can avoid retracing substructures. The SWEEP phase scans memory, finding all unmarked cells, which it strings together into a list. The pointer to that list is kept in the FREE register. When memory is needed (e.g., in CONS) the cell pointed at by FREE is allocated and FREE is set to the CDR of that cell. When the free list becomes empty, the garbage collector is called again. The MARK phase 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. Here 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	P6	E0	P7	N2	E0
the-marks	0	0	0	0	0	0	0	0

Question 49:

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 in the answer sheet.

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

Question 50: Show the state of the memory when control reaches GC-DONE. Please use the diagram in the answer sheet.

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

Question 51: What is the contents of the FREE register when control reaches GC-DONE?

Question 52: What is the maximum number of items that are on the garbage collector stack at any time in running this example? Explain your answer.

Question 53: 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).

The Core Evaluator

```
(define (m-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) (m-eval (cond->if exp) env))
        ((application? exp)
         (m-apply (m-eval (operator exp) env) (list-of-values (operands exp) env)))
        (else (error "Unknown expression type -- EVAL" exp))))

(define (m-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))))

(define (list-of-values exps env)
  (cond ((no-operands? exps) '())
        (else (cons (m-eval (first-operand exps) env)
                    (list-of-values (rest-operands exps) env)))))

(define (eval-if exp env)
  (if (m-eval (if-predicate exp) env)
      (m-eval (if-consequent exp) env)
      (m-eval (if-alternative exp) env)))

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

(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)
                       (m-eval (assignment-value exp) env)
                       env))

(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
                    (m-eval (definition-value exp) env)
                    env))
```

Representing Expressions

```
(define (tagged-list? exp tag)
  (and (pair? exp) (eq? (car exp) tag)))

(define (self-evaluating? exp)
  (or (number? exp) (string? exp) (boolean? exp)))

(define (quoted? exp) (tagged-list? exp 'quote))
(define (text-of-quotation exp) (cadr exp))

(define (variable? exp) (symbol? exp))
(define (assignment? exp) (tagged-list? exp 'set!))
(define (assignment-variable exp) (cadr exp))
(define (assignment-value exp) (caddr exp))

(define (definition? exp) (tagged-list? exp 'define))
(define (definition-variable exp)
  (if (symbol? (cadr exp))   (cadr exp)   (caadr exp)))
(define (definition-value exp)
  (if (symbol? (cadr exp))
      (caddr exp)
      (make-lambda (cdadr exp) (cddr exp))))  ; formal params, body

(define (lambda? exp) (tagged-list? exp 'lambda))
(define (lambda-parameters lambda-exp) (cadr lambda-exp))
(define (lambda-body lambda-exp) (cddr lambda-exp))
(define (make-lambda parms body) (cons 'lambda (cons parms body)))

(define (if? exp) (tagged-list? exp 'if))
(define (if-predicate exp) (cadr exp))
(define (if-consequent exp) (caddr exp))
(define (if-alternative exp)
  (if (not (null? (cdddr exp))) (cadddr exp) 'false))
(define (make-if pred conseq alt) (list 'if pred conseq alt))

(define (cond? exp) (tagged-list? exp 'cond))
(define (cond-clauses exp) (cdr exp))

(define (begin? exp) (tagged-list? exp 'begin))
(define (begin-actions begin-exp) (cdr begin-exp))
(define (last-exp? seq) (null? (cdr seq)))
(define (first-exp seq) (car seq))
(define (rest-exps seq) (cdr seq))
(define (sequence->exp seq)
  (cond ((null? seq) seq)
        ((last-exp? seq) (first-exp seq))
        (else (make-begin seq))))
(define (make-begin exp) (cons 'begin exp))

(define (application? exp) (pair? exp))
```

```
(define (operator app) (car app))
(define (operands app) (cdr app))
(define (no-operands? args) (null? args))
(define (first-operand args) (car args))
(define (rest-operands args) (cdr args))
```

Representing procedures

```
(define (make-procedure parameters body env)
  (list 'procedure parameters body env))
(define (compound-procedure? exp)
  (tagged-list? exp 'procedure))
(define (procedure-parameters p) (list-ref p 1))
(define (procedure-body p) (list-ref p 2))
(define (procedure-environment p) (list-ref p 3))
```

Representing environments

```
;; Implement environments as a list of frames; parent environment is
;; the cdr of the list.  Each frame will be implemented as a list
;; of variables and a list of corresponding values.
(define (enclosing-environment env) (cdr env))
(define (first-frame env) (car env))
(define the-empty-environment '())

(define (make-frame variables values) (cons variables values))
(define (frame-variables frame) (car frame))
(define (frame-values frame) (cdr frame))
(define (add-binding-to-frame! var val frame)
  (set-car! frame (cons var (car frame)))
  (set-cdr! frame (cons val (cdr frame))))

(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Too many args supplied" vars vals)
          (error "Too few args supplied" vars vals))))

(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars) (env-loop (enclosing-environment env)))
            ((eq? var (car vars)) (car vals))
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable -- LOOKUP" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame) (frame-values frame)))))
  (env-loop env))
```

```
(define (set-variable-value! var val env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars) (env-loop (enclosing-environment env)))
            ((eq? var (car vars))
             (set-car! vals val))        ; Same as lookup except for this
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable -- SET!" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame) (frame-values frame)))))
  (env-loop env))

(define (define-variable! var val env)
  (let ((frame (first-frame env)))
    (define (scan vars vals)
      (cond ((null? vars) (add-binding-to-frame! var val frame))
            ((eq? var (car vars)) (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))
    (scan (frame-variables frame)
          (frame-values frame))))
```

Primitive Procedures and the Initial Environment

```
(define (primitive-procedure? proc) (tagged-list? proc 'primitive))
(define (primitive-implementation proc) (cadr proc))
(define primitive-procedures
  (list (list 'car car)
        (list 'cdr cdr)
        (list 'cons cons)
        (list 'null? null?)
        (list '+ +)
        (list '> >)
        (list '= =)
        (list '* *)
        (list 'display display)
        (list 'not not)
        ; ... more primitives
        ))

(define (primitive-procedure-names) (map car primitive-procedures))
(define (primitive-procedure-objects)
  (map (lambda (proc) (list 'primitive (cadr proc))) primitive-procedures))
(define (apply-primitive-procedure proc args)
  (apply (primitive-implementation proc) args))

(define (setup-environment)
  (let ((initial-env (extend-environment (primitive-procedure-names)
                                         (primitive-procedure-objects)
                                         the-empty-environment)))
```

```
    (define-variable! 'true #t initial-env)
    (define-variable! 'false #f initial-env)
    initial-env))
(define the-global-environment (setup-environment))
```

The Read-Eval-Print Loop

```
(define input-prompt ";;; M-Eval input:")
(define output-prompt ";;; M-Eval value:")
(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (m-eval input the-global-environment)))
      (announce-output output-prompt)
      (display output)))
  (driver-loop))
(define (prompt-for-input string)
  (newline) (newline) (display string) (newline))
(define (announce-output string)
  (newline) (display string) (newline))
```

Suppose we want to add some simple type-checking to our language. In particular, we would like to be able to specify conditions or constraints on the types of arguments that a procedure may take, with the idea that before we apply the procedure, we ensure that the supplied arguments meet those constraints. For example, we could extend our syntax to allow:

```
(define (foo (number? x) (list? y) z)
   some-body)
```

The idea is that when we are about to apply `foo` to some arguments, we will ensure that the first argument satisfies `number?` and the second argument satisfies `list?` before proceeding. Since there are no constraints specified on the last argument, anything is acceptable.

Here is the current version of `m-apply`

```
(define (m-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))))
```

We need to change this to handle the type checking as follows:

```
(define (m-apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (let ((params (procedure-parameters procedure)))
           (if (safe? params arguments)
               (eval-sequence
                 (procedure-body procedure)
                 (extend-environment
                     (map (lambda (x) (if (symbol? x) x (cadr x)))
                          params)
                     arguments
                     (procedure-environment procedure)))
               (error ``incorrect argument type'' procedure))))
        (else (error "Unknown procedure type -- APPLY" procedure))))
```

To complete this change, we need to implement `safe?` which should check that each argument meets the specified type constraint:

```
(define (safe? params args)
  (cond ((null? params)
         QUESTION_1)
        ((null? args) #f)
        ((symbol? (car params))
         QUESTION_2)
        (QUESTION_3
         (safe? (cdr params) (cdr args)))
        (else #f)))
```

Question 31: What expression should be used for `QUESTION-1`?

(null? args)

Question 32: What expression should be used for `QUESTION-2`?

(safe? (cdr params) (cdr args))

Question 33: What expression should be used for `QUESTION-3`?

(m-apply (car (car params)) (list (car args)))

BACKGROUND INFORMATION—THIS PAGE CONTAINS NO QUESTIONS

```
; Our familiar OOP system

(define (ask object message . args)
  (apply-method object object message args))

(define (delegate to from message . args)
  (apply-method to from message args))

(define (apply-method in-object for-object message args)
  (let ((method (get-method message in-object)))
    (cond ((method? method)
           (apply method for-object args))
          ((eq? in-object for-object)
           (error "No method for" message 'in (safe-ask 'unnamed-object in-object 'NAME)))
          (else (error "Can't delegate" message
                       "from" (safe-ask 'unnamed-object for-object 'NAME)
                       "to" (safe-ask 'unnamed-object in-object    'NAME))))))

(define (safe-ask default-value obj msg . args)
  (let ((method (get-method msg obj)))
   (if (method? method)
       (apply-method obj obj msg args)
       default-value)))

(define (get-method message object)     ; single-inheritance
  (object message))

(define (find-method message . objects) ; multiple-inheritance
  (define (try objects)
    (if (null? objects)
        (no-method)
        (let ((method (get-method message (car objects))))
          (if (not (eq? method (no-method)))
              method
              (try (cdr objects))))))
  (try objects))

(define (method? x)
  (cond ((procedure? x) #T)
        ((eq? x (no-method)) #F)
        (else (error "Object returned this non-message:" x))))

(define no-method
  (let ((tag (list 'NO-METHOD)))
    (lambda () tag)))
```

BACKGROUND INFORMATION—THIS PAGE CONTAINS NO QUESTIONS

Part 8 (points)**

Suppose we want to insert an element into a possibly ordered collection of elements. For each of the following `insert` procedures, indicate the order of growth of the procedure in using three different resources: $T(n)$, $S(n)$, and $H(n)$. Here, $T(n)$ is the time required, measured by the number of primitive operations performed. $S(n)$ is the space required, measured by the maximum depth of the stack. And finally, $H(n)$ is the space required, but now measured by the number of new `cons` cells explicitly allocated from the heap (ignoring `cons` cells allocated by the system to handle frames and environments). In each case, n is the number of elements already in the database `db`. Select your answer from the following:

1. $O(1)$
2. $O(n)$
3. $O(n^2)$
4. $O(2^n)$
5. $O(\log n)$
6. none of the above

Question 44:

```
(define (insert elt db) (cons elt db))
```

1 1 1

Question 45:

```
(define (insert elt db)
  (if (null? db)
      (list elt)
      (if (< elt (car db))
          (cons elt db)
          (cons (car db)
               (insert elt (cdr db))))))
```

n n n

Question 46:

```
(define (insert elt db)
  (if (= (length db) 0)
      (list elt)
      (if (< elt (car db))
          (cons elt db)
          (cons (car db)
```

```
                  (insert elt (cdr db))))))

  (define (length lst)
    (if (null? lst)
        0
        (+ 1 (length (cdr lst)))))
```

n^2 **n n**

Question 47:

```
  (define (insert elt db)
    (define (help cur elt next)
       (if (null? next)
           (set-cdr! cur (list elt))
           (if (< elt (car next))
               (let ((new (list elt)))
                 (set-cdr! cur new)
                 (set-cdr! new next))
               (help next elt (cdr next)))))
    (if (null? db)
        (list elt)
        (if (< elt (car db))
            (cons elt db)
            (help db elt (cdr db)))))
```

n 1 n