MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001-- Structure and Interpretation of Computer Programs
Fall Semester, 1998, Final Exam

Be sure to write your name on all pages of this exam.

Print Your Name:

Your Recitation Instructor:

Your Tutor:

Please write clear and concise answers to the questions in the spaces provided in this booklet. You may use scratch paper if you need, but the spaces we provide are the only places we will look at when grading. Your solutions, particularly to programming problems, may be judged not only on whether they work or not, but also on clarity and ease of understanding.

Note that not all questions are of equivalent difficulty, so you may wish to skim all questions in this exam booklet before beginning.

Any comments you would like to make on this exam:

Comments from graders:

Problem	Value	Grade	Grader
1	30		
2	35		
3	25		
4	40		
5	25		
6	30		
7	15		
Total	200		

Problem 1

For each of the statements below, circle T if the statement is true, and F if the statement is false.

T F The halting theorem states that it is impossible to examine any piece of code and determine if it will terminate or not.

T (F) In the United States, software can be copyrighted but not patented.

(T) F A Java compiler converts class definitions written in the Java programming language into bytecodes that are hardware-independent.

T (F) Java is an object-oriented programming language that supports class variables, class methods, instance variables, instance methods, and multiple inheritance.

T (F) Garbage collection cannot be used in Java because Java programs must react to the user or to the network in real time.

(T) F Mark/sweep garbage collection may require a stack of depth *n* in order to mark a memory with *n* pairs.

T (F) If the Scheme evaluator supports the `delay` and `force` special forms, it is possible for the Scheme user to implement `cond` as a simple procedure without additional extensions to the evaluator.

T (F) Deadlock can occur between two processes that are running in parallel if a mutual exclusion approach is used (such as the synchronization approach discussed in class) in which both processes try to gain access to a single shared resource.

(T) F The possible values for `z` at the completion of the parallel-execution below

```
(define z 5)
(define (P1) (set! z (+ z 10)))
(define (P2) (set! z (* z 2)))
(parallel-execute P1 P2)
```

are 10, 15, 20, and 30.

(T) F The value of the following expression is equal to the expression itself:

```
( (lambda (x) (list x (list 'quote x)))
 '(lambda (x) (list x (list 'quote x))) )
```

T F **Extra credit (0 points):** In everything other than legal documents, it is not discouraged to speak without resorting to double negatives, except when writing comments in your code.

Problem 2

You are interested in creating a `computer-account` object to help with the administration of a computer system that may have many users. The behavior you desire is that each account (instance of `computer-account`) can have its own unique `username` which is specified when the account is created. The `username` can later be retrieved with the `USERNAME` method.

Part A.

Your team of 6.001 graduates supplied the following different implementations for the `computer-account` class. For each implementation, indicate whether the code will work correctly or not; if it will not work correctly, explain why.

A1:

```
(define (make-computer-account uname)
  (let ((username '()))
    (set! username uname)
    (lambda (message)
      (case message
        ((USERNAME) (lambda (self) username))
        (else (no-method))))))
```

Circle one: (CORRECT) or INCORRECT BECAUSE:

A2:

```
(define make-computer-account
  (let ((username '()))
    (lambda (uname)
      (set! username uname)
      (lambda (message)
        (case message
          ((USERNAME) (lambda (self) username))
          (else (no-method)))))))
```

Circle one: CORRECT or (INCORRECT) BECAUSE:

the same variable `username` would be shared by all instances.

A3:

```
(define make-computer-account
  (lambda (username)
    (lambda (message)
      (case message
        ((USERNAME) (lambda (self) username))
        (else (no-method))))))
```

Circle one: (CORRECT) or INCORRECT BECAUSE:

Part B.

Next, we wish to create a new class `password-account` that inherits the behavior of `computer-account` and extends it in the following way. Each `password-account` instance will also have a password, and the account will support a `LOGIN` method that checks to see if the correct password has been given before allowing the user to `begin-session` (the `begin-session` procedure is implemented elsewhere). The code below is an attempt to accomplish this, but it is missing several critical parts. You are to provide any additional needed code or other modifications below. You may mark directly on the code below to (clearly) indicate where your additions belong, and to X-out where code is being deleted or changed.

```
(define make-password-account
  (lambda (uname passwd)
    (let ((password passwd))                    (account (make-account uname))
      (lambda (message)
        (case message
           ((LOGIN) (lambda (self pass)
                      (if (eq? pass password)
                          (begin-session)
                          (error "Incorrect password"))))
           (else (no-method)))))))

                                  (account message)
```

Part C.

C1. Complete the expression below to create a `password-account` for Ben Bitdiddle (Ben selects the symbol `benbit` for his username and `2tothe2is4` for his password).

```
(define ben-account
      (make-password-account 'benbit '2tothe2is4)
  )
```

C2. Write an expression to show how Ben can successfully log in to the system.

```
((ben-account 'LOGIN) ben-account '2tothe2is4)
```

Part D.

It is sometimes useful to have an account with two passwords. Rather than further subclass password-account, we wish to change the implementation of password account by adding a method `ALLOW-JOINT` which takes a second password as an argument. By default, the account is created as a single password account, but using the `ALLOW-JOINT` method one can add a second password. You need to supply the missing line of code `<EXP1>` below.

(Note that for the moment we are not concerned with the changes you made in part c, rather, we are only focusing on the new `ALLOW-JOINT` method in this part.)

```
(define make-password-account
  (lambda (uname passwd)
    (let ((password1 passwd)
          (password2 '()))
      (lambda (message)
        (case message
           ((LOGIN) (lambda (self pass)
                      (if (eq? pass password)
                          (begin-session)
                          (error "Incorrect password"))))
           ((ALLOW-JOINT) (lambda (self second-passwd)
                            <EXP1>
                            ))
           (else (no-method)))))))
```

`<EXP1>`:

```
(set! password2 second-passwd)
```

Part E.

In order to deal with the situation when a second password has indeed been specified, we need to change our `LOGIN` method as well. Replace the (`if ...`) expression from the `LOGIN` method of part D with a new expression so that if the account is a single user account, then only when the correct `password1` is supplied should the user be able to login. If the account has been enabled as a joint account, then only when `password1` or `password2` is supplied should login be successful.

```
(if (or (eq? password1 pass)
        (and (not (null? password2))
             (e1? password2 pass)))
    (begin-session)
    (error "Incorrect password"))
```

Problem 3

In each part of this problem you are given a sequence of register machine instructions with the entry point or entry label `start` that implements some register machine subroutine. Our goal is to understand how these instructions perform as a subroutine. After each set of instructions several statements will be made: circle the letter for **all** statements that are true (there will usually be more than one true statement) and fill in any blanks if the statement is true and has a blank.

Note that each of the subroutines to follow may well be intended to compute different things. Before each part, you should assume that the following instructions are issued (e.g. to call the subroutine from a "fresh" register machine) initialized as follows:

```
  (assign x (const '(10 11 (12 13))))
  (assign continue (label done))
  (goto (label start))  ;call the subroutine
done
  ... <here we stop the machine and look at the ans register>
```

Part A.

```
start
  (assign ans (const 0))
loop
  (test (op null?) (reg x))
  (branch (reg continue))
  (assign ans (op +) (reg ans) (const 1))
  (goto (label loop))
```

(A1.) The resulting register machine is iterative (results in a stack of constant depth).

A2. The resulting register machine is recursive (results in a stack with maximum depth that depends on the size of the input `x`).

A3. Returns (to the `done` label) with the following value in the `ans` register: _____

A4. Returns (to the `done` label) with the stack in a different state than it was at the `start` entry.

(A5.) Results in a process which continues forever.

A6. Results in a process which terminates with the following error: _____

Part B.

```
start
  (assign x (op cdr) (reg x))
  (test (op null?) (reg x))
  (branch (label end))
  (save continue)
  (assign continue (label add-to-count))
  (goto (label start))
end
  (assign ans (const 1))
  (goto (reg continue))
add-to-count
  (assign ans (op +) (reg ans) (const 1))
  (restore continue)
  (goto (reg continue))
```

B1. The resulting register machine is iterative (results in a stack of constant depth).

(B2.) The resulting register machine is recursive (results in a stack with maximum depth that depends on the size of the input `x`).

(B3.) Returns (to the `done` label) with the following value in the `ans` register: **3**

B4. Returns (to the `done` label) with the stack in a different state than it was at the `start` entry.

B5. Results in a process which continues forever.

B6. Results in a process which terminates with the following error: _____

Part C.

```
start
  (save x)
  (assign x (op cdr) (reg x))
  (test (op null?) (reg x))
  (branch (label found_last))
  (goto (label start))
found_last
  (restore ans)
  (assign ans (op car) (reg ans))
  (goto (reg continue))
```

C1. The resulting register machine is iterative (results in a stack of constant depth).

(C2.) The resulting register machine is recursive (results in a stack with maximum depth that depends on the size of the input `x`).

(C3.) Returns (to the `done` label) with the following value in the `ans` register: **(12 13)**

(C4.) Returns (to the `done` label) with the stack in a different state than it was at the `start` entry.

C5. Results in a process which continues forever.

C6. Results in a process which terminates with the following error: _____

Part D.

```
start
  (save x)
  (assign x (op cdr) (reg x))
  (test (op null?) (reg x))
  (branch (label found_last))
  (restore ans)                ;this is the only change from part C
  (goto (label start))
found_last
  (restore ans)
  (assign ans (op car) (reg ans))
  (goto (reg continue))
```

(D1.) The resulting register machine is iterative (results in a stack of constant depth).

D2. The resulting register machine is recursive (results in a stack with maximum depth that depends on the size of the input `x`).

(D3.) Returns (to the `done` label) with the following value in the `ans` register: **(12 13)**

D4. Returns (to the `done` label) with the stack in a different state than it was at the `start` entry.

D5. Results in a process which continues forever.

D6. Results in a process which terminates with the following error: _____

Problem 4

The object oriented system in Scheme that we used this term had mechanisms for creating instances with their own local state, methods that could be invoked by message passing to an instance, and a mechanism for inheritance through delegation. In this problem, we explore a mechanism to support *class variables* and *class methods* in addition to *instance variables* and *instance methods*.

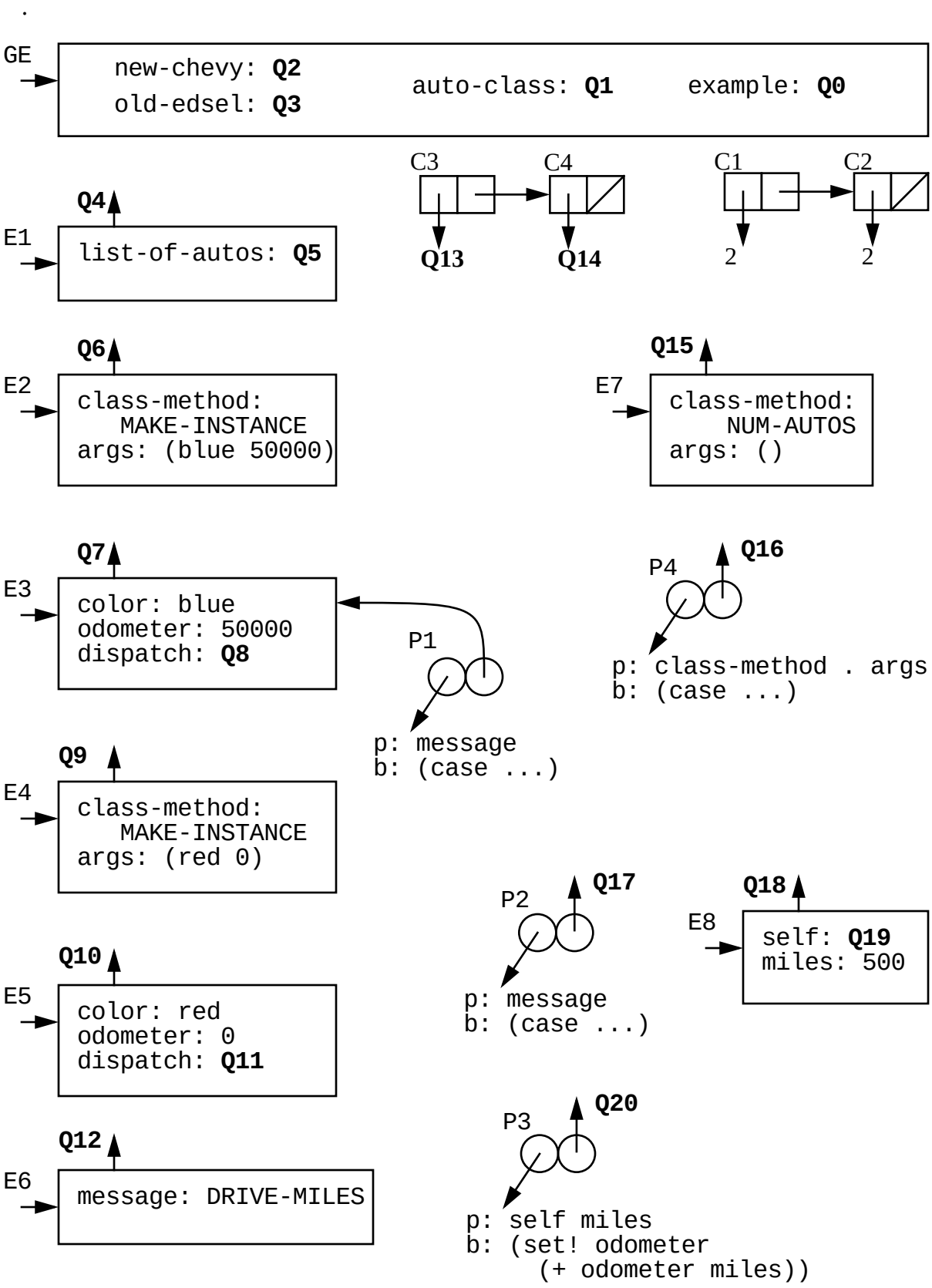
Consider the following expression which is evaluated in the global environment (and which creates part of the environment diagram shown on the next page):

```
(define auto-class
  (let ((list-of-autos '()))
    (lambda (class-method . args)
      (case class-method
        ((MAKE-INSTANCE)
         (let ((color (car args))
               (odometer (cadr args)))  ;miles car has been driven
           (define (dispatch message)
             (case message
               ((GET-COLOR) (lambda (self) color))
               ((SET-COLOR) (lambda (self newcolor)
                              (set! color newcolor)))
               ((DRIVE-MILES) (lambda (self miles)
                                (set! odometer (+ odometer miles))
                                odometer))))
           (set! list-of-autos (cons dispatch list-of-autos))
           dispatch))
        ((NUM-AUTOS) (length list-of-autos))))))
```

The following expressions are then evaluated in sequence in the global environment:

```
(define example (list 2 2))
(define new-chevy (auto-class 'MAKE-INSTANCE 'red 0))
(define old-edsel (auto-class 'MAKE-INSTANCE 'blue 50000))
(auto-class 'NUM-AUTOS) ==> 2
((new-chevy 'DRIVE-MILES) new-chevy 500) ==> 500
```

creating the environment diagram as shown on the next page. In this environment diagram, the different environments are labeled `E1` through `E8` (and `GE` for the global environment); the different procedure objects are labeled `P1` through `P4`; and cons cells are labeled `C1` through `C4`. Your job is to complete the "wiring" of the environment diagram by completing the table on page 10 to indicate what each of the pointers **`Q0`** through **`Q20`** should point to (`E1-E8, GE, P1-P4, C1-C4`, or none of the above). While you are free to draw on the environment diagram to help work through the evaluation of the above expressions (beginning with the `(define auto-class ...)` expression), we will ONLY look at the table on page 10 where you must put your final answers.


GE
new-chevy: Q2
old-edsel: Q3
auto-class: Q1
example: Q0
C3
C4
C1
C2
Q13
Q14
2
2
Q4
E1
list-of-autos: Q5
Q6
E2
class-method:
MAKE-INSTANCE
args: (blue 50000)
Q15
E7
class-method:
NUM-AUTOS
args: ()
Q7
E3
color: blue
odometer: 50000
dispatch: Q8
P1
p: message
b: (case ...)
Q16
P4
p: class-method . args
b: (case ...)
Q9
E4
class-method:
MAKE-INSTANCE
args: (red 0)
Q17
P2
p: message
b: (case ...)
Q18
E8
self: Q19
miles: 500
Q10
E5
color: red
odometer: 0
dispatch: Q11
Q12
E6
message: DRIVE-MILES
Q20
P3
p: self miles
b: (set! odometer
(+ odometer miles))

Part A.

In the following table, indicate what each of the "question pointers" **`Q1`** through **`Q20`** should be in the environment diagram on the preceding page. For example, **`Q0`** should be a pointer to the cons cell C1 as indicated in the table below. You should indicate the objects being pointed to as one of the environments labeled `E1` through `E8` (or `GE` for the global environment); the different procedure objects labeled `P1` through `P4`; or a cons cells labeled `C1` through `C4`. If the pointer goes to none of these, write "none" in the table for that cell.

Question Pointer	Object Pointed To
`Q0`	C1
`Q1`	**P4**
`Q2`	**P2**
`Q3`	**P1**
`Q4`	**GE**
`Q5`	**C3**
`Q6`	**E1**
`Q7`	**E2**
`Q8`	**P1**
`Q9`	**E1**
`Q10`	**E4**

Question Pointer	Object Pointed To
`Q11`	**P2**
`Q12`	**E5**
`Q13`	**P1**
`Q14`	**P2**
`Q15`	**E1**
`Q16`	**E1**
`Q17`	**E5**
`Q18`	**E6**
`Q19`	**P2**
`Q20`	**E6**

Part B.

In part a, we assigned `old-edsel` to be an instance of the `auto` class. If we evaluate the following expression in the global environment:

```
(set! old-edsel new-chevy)
```

is it now safe to garbage collect the object that `old-edsel` was originally pointing to? Explain why or why not.

No, the `list-of-autos` still holds a valid pointer to the auto instance that was `old-edsel`.

Part C.

It is sometimes necessary for an auto maker to issue a `RECALL` when some defect or safety problem is discovered. Modify the `auto-class` below to complete the class method `RECALL` that will look at each auto ever created and perform some user-specified recall test and modification operation (named `user-recall-procedure` below) on each auto. To make this change, provide the code for `<EXP>` below.

```
(define auto-class
  (let ((list-of-autos '()))
    (lambda (class-method . args)
      (case class-method
        ((MAKE-INSTANCE)
         (let ((color (car args))
               (odometer (cadr args)))  ;miles car has been driven
           (define (dispatch message)
             (case message
               ((GET-COLOR) (lambda (self) color))
               ((SET-COLOR) (lambda (self newcolor)
                              (set! color newcolor)))
               ((DRIVE-MILES) ...)))
           (set! list-of-autos (cons dispatch list-of-autos))
           dispatch))
        ((RECALL)
         (let ((user-recall-procedure (car args)))
            <EXP>))
        ((NUM-AUTOS) (length list-of-autos))))))
```

`<EXP>`:

```
(map user-recall-procedure list-of-autos)
```

Part D.

It has just been determined that all red autos are dangerous. Complete the following expression to issue a recall in which you change the color of all red cars to green.

```
(auto-class 'RECALL

     (lambda (auto)
       (if (eq? ((auto 'GET-COLOR) auto) 'red)
           ((auto 'SET-COLOR) auto 'green)
           'ok))


)
```

Problem 5

The term's over, and you are busy packing because you need to move out of your room. In your desk, you find a piece of paper with two compiled code fragments. Intrigued by your discovery, you sit down on your bed and occupy yourself with the pleasant diversion of trying to figure out what scheme expression was compiled to produce each compiled code fragment.

Here's the first compiled excerpt:

```
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 (f))
            (reg argl) (reg env))
6.  (assign val (op lookup-variable-value) (const f) (reg env))
7.  (assign val (op lookup-variable-value) (const f) (reg env))
8.  (goto (reg continue))
9. after-lambda1
10. (perform (op define-variable!) (const x) (reg val) (reg env))
11. (assign val (const ok))
```

From lines 1 and 4, you deduce that the scheme expression is some sort of procedure definition. Examining lines 1-11, answer each part below:

Part A.
How many arguments does the procedure take? **`1`**

Part B.
What is the procedure body? **`(begin f f)`**

Part C.
What is the name of the procedure being defined? **`x`**

Part D.
What is the scheme expression that was compiled? **`(define (x f) f f)`**

Part E.
Which line in the compiled code is unnecessary (other than line 11) and can be eliminated without changing the result?

`Line 6 or 7`

Part F.
What scheme expression produces the same compiled code as above, without the unnecessary line of part E?

`(define (x f) f)`

Further down on the very same page **immediately following** the code shown in parts A through E, you find this second excerpt:

```
1.  (assign proc (op lookup-variable-value) (const x) (reg env))
2.  (assign val (const 2))
3.  (assign argl (op list) (reg val))
4.  (test (op primitive-procedure?) (reg proc))
5.  (branch (label primitive-branch8))
6. compiled-branch7
7.  (assign val (op compiled-procedure-entry) (reg proc))
8.  (goto (reg val))
9. primitive-branch8
10. (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
11. (goto (reg continue))
```

Part G.

Study the following sections: lines 1-3, lines 4-5, lines 6-8, lines 9-11. What was the scheme expression that was compiled?

(x 2)

Part H.

Looking at the code, we know what scheme expressions produced both the first and the second compiled excerpts. Using this knowledge, how can we optimize the compiled code in the second excerpt? In other words, which lines in the compiled code in the second excerpt are unnecessary and can be safely removed?

Line 4, 5, 9, 10, 11 are not needed, since we know x is a compound procedure (and will stay a compound procedure since x is not rebound before excerpt 2, and there is no entry point to jump into excerpt 2).

Lines 2 and 3 can be combined to (assign argl (op list) (const 2))

Ultimately, the whole thing could be replace with line 2 since we know what procedure X does.

Problem 6

In our discussion of the Java programming language, we saw that Java is *strongly typed*: that is, one must *declare* the type of a variable to indicate that only values consistent with that type may be assigned to the variable, e.g.

```
int i = 5;
String s = "Hello";
...
i = 27;     // This is okay
i = false;  // This is NOT okay
```

This is an interesting design decision which enables the Java compiler or interpreter to check that assignments during compile-time or run-time are type-consistent, and issue an error if not.

In this problem we consider changes to our Scheme evaluator to add some degree of type declarations to the language. In particular, we wish to extend the syntax of variable definitions to explicitly require that a type for the variable be given, e.g.

```
(define number x 27)
(define string y "Hello")
(define procedure abs-value
  (lambda (x) (if (> x 0) x (- x))))
```

or generally

```
(define <type> <var> <val>)
```

In this problem we will first extend our frame representation to store the type associated with variables in the environment, then we will extend the evaluation procedures to ensure that we interpret the extended `define` syntax, and finally we will also extend the assignment semantics so that an error is generated if one attempts to later assign a value with an inconsistent type to a variable.

Part A.

In the representation of frames and environments discussed in the book and used on the problem set, we represented a frame as a list of variable names (symbols), together with a list of variable values in one to one correspondence:

```
(define (make-frame variables values)
  (cons variables values))
```

We will change this so that we instead keep a list of variable types, a list of variable names, and a list of variable values, all in one-to-one correspondence:

```
(define (make-frame types variables values)
  (list types variables values))
```

Complete the definitions of the accessor functions consistent with this new frame representation:

```
(define (frame-types frame)
     (car frame)
 )

(define (frame-variables frame)
      (cadr frame)
 )

(define (frame-values frame)
      (caddr frame)
 )
```

Part B.

We also have to change the `add-binding-to-frame!` procedure to work with our new frame representation. Here is the old procedure:

```
(define (add-binding-to-frame! var val frame)
  (set-car! frame (cons var (car frame)))
  (set-cdr! frame (cons val (cdr frame))))
```

Complete the revised version of `add-binding-to-frame!` to work with our new representation:

```
(define (add-binding-to-frame! type var val frame)

      (set-car! frame (cons type (car frame)))
      (set-car! (cdr frame) (cons var (cadr frame)))
      (set-car! (cddr frame) (cons val (caddr frame)))

 )
```

Part C.

The `lookup-variable-value` procedure from the evaluator is shown below.

```
(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" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
```

We see that this procedure still works with our modified frame representation to retrieve a variable value from the environment.

We will also need a `lookup-variable-type` procedure so that later on we can retrieve the declared type for a variable from the frame (e.g. for later use in type checking). Using the `lookup-variable-value` procedure above as an example, we have created a start toward the new `lookup-variable-type` procedure below. Complete the modifications necessary in the code below to complete the definition of `lookup-variable-type`. Be careful to clearly indicate where any additional expressions you write are to be inserted, and to indicate or X-out any code to be deleted or changed.

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

Next we need to make changes to the evaluator to handle our extended syntax. Repeated below is `meval` which calls `eval-definition` to handle the `(define ...)` syntax:

```
(define (meval exp env)
  (cond ((self-evaluating? exp) exp)
        ...
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ...
        ((application? exp)
         (mapply (meval (operator exp) env)
                 (list-of-values (operands exp) env)))
        (else (error "Unknown expression type -- EVAL" exp))))
```

To handle our new type declaration syntax, we revise `eval-definition` as follows:

```
(define (eval-definition exp env)
  (define-variable! (definition-type exp)  ;added for types
                    (definition-variable exp)
                    (meval (definition-value exp) env)
                    env)
  'ok)
```

The `eval-definition` procedure uses the syntax procedures `definition-type`, `definition-variable`, and `definition-value`. The implementation for `definition-type` must be updated in order for it to handle both the fully declared definition of variables, e.g. `(define number x 5)` and the syntactic sugared form for procedure definition:

```
(define (definition-type exp)
  (if (symbol? (cadr exp))
      (cadr exp)      ;fully declared case
      'procedure))    ;syntactic sugared procedure definition case
```

The old implementations for `definition-variable` and `definition-value` are shown below; these were necessary to deal with the syntactic sugared form of procedure definition.

```
(define (definition-variable exp)
  (if (symbol? (cadr exp))
      (cadr exp)      ;regular definition case
      (caadr exp)     ;procedure definition case
      ))

(define (definition-value exp)
  (if (symbol? (cadr exp))
      (caddr exp)      ;regular definition case
      (make-lambda     ;procedure definition case
       (cdadr exp)          ;formal parameters
       (cddr exp))))        ;body
```

With our type declaration extension, we would like the "shorthand" (syntactic sugared) way of writing a procedure:

```
(define (abs-value x)
  (if (> x 0) x (- x)))
```

to be interpreted equivalently to a fully declared definition for the variable `abs-value`:

```
(define procedure abs-value
  (lambda (x) (if (> x 0) x (- x))))
```

Part D.
Revise the `definition-variable` procedure (shown on the previous page) to handle both the simple `define` syntax such as `(define number x 5)` and the sugared form of a procedure definition as shown above:

```
(define (definition-variable exp)
  (if (symbol? (cadr exp))
      (caddr exp)      ;regular (typed) definition case
      (caadr exp)      ;procedure definition case
      ))
```

Part E.
Revise the `definition-value` procedure (shown on the previous page) to handle both the simple `define` syntax such as `(define number x 5)` and the sugared form of a procedure definition as shown above:

```
(define (definition-value exp)
  (if (symbol? (cadr exp))
      (cadddr exp)      ;regular definition case
      (make-lambda     ;procedure definition case
       (cdadr exp)          ;formal parameters
       (cddr exp))))        ;body
```

Part F.

Finally, we also want to update the evaluation of assignments so that errors are generated if a type-inconsistent assignment is attempted, e.g.

```
(define number x 5)
(set! x 'foo)
==> ERROR: attempt to assign to variable X a value of wrong type
```

To implement this, you should assume you already have available a procedure `(type-consistent? type value)` that returns `#t` if the type of the object `value` is consistent with the declared `type`:

```
(type-consistent? 'number 5) ==> #t
(type-consistent? 'string 'foo) ==> #f
```

Shown below is the `set-variable-value!` procedure from the evaluator which you will need to modify:

```
(define (set-variable-value! var val env)
  (define (env-loop env)
    (define (scan types vars vals)
      (cond ((null? vars)
             (env-loop (enclosing-environment env)))
            ((eq? var (car vars))
             (if (type-consistent? (car types) val)
               (set-car! vals val)
               (error "wrong type"))
            (else (scan (cdr types) (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable -- SET!" var)
        (let ((frame (first-frame env)))
          (scan (frame-types frame)
                (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
```

Modify the procedure above in order to check for type consistency in variable assignments. Be sure to clearly indicate where any additional code you write is inserted, and to mark code that is to be deleted.

Problem 7

This problem considers a small list structured memory. As discussed in class, the contents of memory cells are represented in a shorthand notation, where "N<#>" indicates the storage of a number, "P<index>" indicates the storage of a pointer to the cell at a given index, and "E0" indicates the null pointer. We extend this notation with "S<symbol>" to indicate a symbol, e.g. Sfoo indicates that the cell contains the symbol "foo".

The current contents of memory, as well as the `free-list` pointer, are shown below.

	0	1	2	3	4	5	6	7	8	9
the-cars	P1	P5	Sz	P4	Sy	Sx	E0	N20	N30	E0
the-cdrs	E0	P3	N60	E0	P7	N10	P9	P8	P7	E0

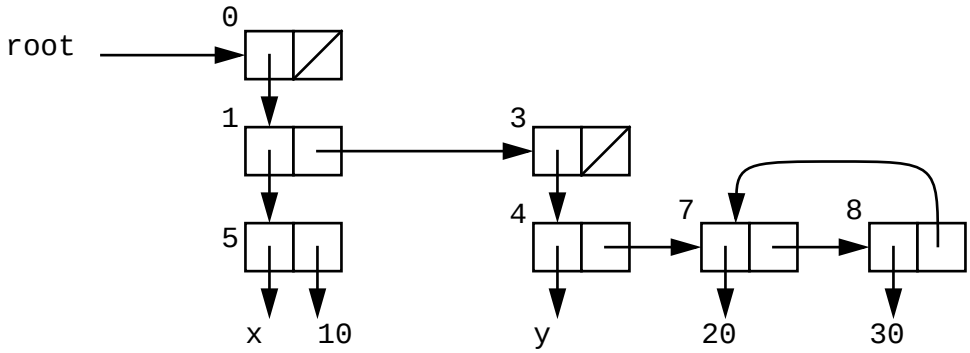
```
root:  P0
free-list: P6
```

The "root" to this data structure corresponds to the global environment, or `GE = P0`.

Free cells in memory are represented as a linked list of unused `cons` cells. That is, the `cdr` part of each free cell points to the "next" free cell in the free list. If there are no more free cells, then the cdr part is the empty list `E0`. New cons cells are allocated by taking the first cell pointed to by the `free-list`, and updating or overwriting the `free-list` register to point to the next free cell.

The contents of this memory correspond to a simplified representation of a Scheme environment such as might be used by the explicit control evaluator (Scheme interpreter). In this case, the environment is represented as a list of frames; each frame contains a list of bindings; each binding is a pair of a variable name (symbol) and its value.

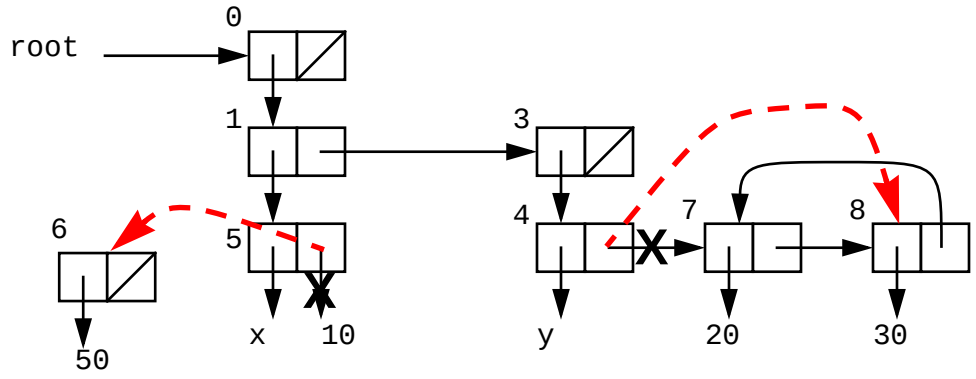
The following box and pointer diagram corresponds to the current global environment (pointed to by the `root` register:



Part A.
The following expression is evaluated in this global environment.

```
(set! y (cdr y))
```

A1. Show what changes on the box and pointer diagram below:



A2. Also show what changes in the memory structure below (repeated here from the previous page) and update the `free-list` or `root` registers if necessary

	0	1	2	3	4	5	6	7	8	9
the-cars	P1	P5	Sz	P4	Sy	Sx	E0	N20	N30	E0
the-cdrs	E0	P3	N60	E0	~~P7~~ **P8**	N10	P9	P8	P7	E0

```
root: P0
free-list: P6
```

Part B.
The following expression is evaluated in our global environment:

```
(set! x '(50))
```

B1. Show any additional changes in the environment box and pointer diagram above.

B2. Show what additional changes occur in the memory structure below, and update the `free-list` or `root` registers if necessary.

	0	1	2	3	4	5	6	7	8	9
the-cars	P1	P5	Sz	P4	Sy	Sx	~~E0~~ **N50**	N20	N30	E0
the-cdrs	E0	P3	N60	E0	P7	~~N10~~ **P6**	~~P9~~ **E0**	P8	P7	E0

```
root: P0
free-list: P6  P9
```

Part C.

The following expression is evaluated with respect to our global environment data structure:

```
(set! y 40)
```

What cells, if any, become available for garbage collection (that is, what cells currently in the environment data structure could be added to the `free-list`) when this expression is evaluated?

Cells 7 and 8