CS61A Midterm 2, Spring 2011

1. Object-Oriented Programming (6 points)

A Child class (using the parent clause)
B Class variable
C Default method
D Initialize method/clause
E Instance variable
F Instantiation variable
G Method

COLOR: Lets us know what color the automobile is.
    The correct answer is F (instantiation variable). The idea is that every automobile must have a color - this implies that COLOR is either an instance variable, or an instantiation variable.
    After some thought, we realize that COLOR must be an instantiation variable, because all automobiles are created /with/ a (possibly different) color. It doesn't quite make sense to create an automobile without specifying its color.

POLLUTION: Tells us how much pollution has been produced by all the automobiles of Berkeley, measured in smogallons. (One smogallon of pollution is produced whenever an automobile uses one gallon of gas.)
    The correct answer is B (Class variable). The hint here is "how much pollution has been produced by /all/ the automobiles." The total amount of pollution is "known" by every automobile object (also, each automobile contributes to the total amount of pollution) - thus, it must be a class variable.

STARTING-FUEL: A new automobile contains a random amount of fuel (between 0 and its capacity).
    The correct answer is E (Instance variable). The idea is that every car always knows how much fuel it started out with - so, the two "possible" answers are instance variable or instantiation variable. However, it wouldn't be an instantiation variable, because this would imply that, at creation time, the /user/ gets to specify the starting fuel. This is not a "random amount" of fuel!
    Instead, the starting value of STARTING-FUEL will be initialized to a random amount, something like:

```
(define-class (automobile fuel-efficiency fuel-capacity color)
    (class-vars (pollution 0))
    (instance-vars (starting-fuel (random fuel-capacity)))
    ...)
```

    Really this variable ought to be called FUEL, so that it can vary as the automobile drives!

MILES-REMAINING: Tells us how far this automobile can go on its current fuel.
    The correct answer is G (Method). In order to compute this quantity, we need to compute the number of miles we can go, which depends on the current fuel level and the FUEL-EFFICIENCY. This computation must be done within a method body, because the current fuel level can change during the course of the simulation.

OLD-AUTOMOBILE: Like a regular automobile, but produces two smogallons of pollution per gallon of fuel rather than one.
    The correct answer is A (Child class). The key hint was, "/Like/ a regular automobile, /but/ ...". In other words, an OLD-AUTOMOBILE inherits everything from the AUTOMOBILE class, but overrides the pollution behavior.

Starting MILES-REMAINING: When instantiated, a new AUTOMOBILE prints its MILES-REMAINING (based on its starting fuel).
    The correct answer is D (Initialize method/clause). Here, we want some /action/ to occur when the automobile is being /created/ - this is what the INITIALIZE method/clause is designed for. So, it could look something like:

```
(define-class (automobile fuel-efficiency fuel-capacity color)
    (intiialize
        (print (se '(miles remaining is) miles-remaining)))
    ...)
```

Scoring:
1 point each, all or nothing.

2. ADT usage (4 points)

The corrected program is below:

```
(define (make-square a)
  (attach-tag 'square a) )

(define (make-circle a)
  (attach-tag 'circle a) )

(define (area b)
  (cond ((equal? (type-tag b) 'square) (* (contents b) (contents b)))
        ((equal? (type-tag b) 'circle) (* 3.14 (contents b) (contents b)))
        (else (error "Unknown shape")) ))

(define (area-shapes list-of-shapes)
  (if (null? list-of-shapes)
      '()
      (cons (area (car list-of-shapes))
            (area-shapes (cdr list-of-shapes))) ))
```

Common errors included
- Defining your own constructors and selectors whose name did not represent the functionality of attach-tag, type-tag, and contents.
  Example:

```
(define (get-radius x) (cdr x))
(define (get-side x) (cdr x))
(define (area b)
(cond ((equal? (type-tag b) 'square) (* (get-side b) (get-side b)))
      ((equal? (type-tag b) 'circle) (* 3.14 (get-radius b) (get-radius b)))
      (else (error "Unknown shape")) ))
```

  This is wrong, because the idea of type-tagging, is to create a generalized way of attaching tags to data, getting those tags, and getting the data. get-side and get-radius are not generalized selectors for getting the data values back.
- creating new constructors and selectors for the CONS, CAR, and CDR calls in the AREA-SHAPES procedure. CONS, CAR, and CDR were correct, because LIST-OF-SHAPES is a list.
- changing the procedure area-shapes to use map. The implementation of AREA-SHAPES is not a data abstraction error. Granted, using map would be a cleaner implementation, but Louis asked you to fix the DAVs, and not to make his code nicer.

Scoring:
+1 both ATTACH-TAG changes
+1 both TYPE-TAG changes
+1 both CONTENTS changes
+1 nothing else is changed

3. Scheme-1 (7 points)

In this problem, we asked you to list all calls to eval-1 and apply-1 with their arguments. To answer this, you just needed to carefully trace the code. The answers were as follows:

Scheme-1: (+ 2 3)

```
E (+ 2 3)
E +
E 2
E 3
A [+] (2 3)
```

Scheme-1: ((lambda (x) x) 7)

```
E ((lambda (x) x) 7)
E (lambda (x) x)
E 7
A (lambda (x) x) (7)
E 7
```

In this problem, many people forgot to "E 7" after "A (lambda (x) x) (7))." Remember that if Scheme-1 is applying a lambda procedure, it will EVAL the return value of the SUBSTITUTE call by APPLY.

Scheme-1: (filter vowel? '(t a r d i s))

```
E (filter vowel? '(t a r d i s))
E filter
E vowel
E (quote (t a r d i s))
A [filter] ([vowel?] (t a r d i s))
```

In this problem, some people forgot that after VOWEL? is evaluated, it becomes a primitive. So they wrote

```
A [filter] (vowel? (t a r d i s))
```

But this is incorrect, because MAP is not being applied to the word vowel?, but the primitive VOWEL? procedure.

Scoring:
+3 The three APPLY calls are completely correct. If you left out parentheses around the arguments all three times, we didn't take off any points (+1 each)
+1 You tried to EVAL the return value of SUBSTITUTE after APPLY in the second part
+1 you tried to EVAL all numbers and procedure names
+1 you eval'ed all of '(cat dog rat) together, instead of separately eval'ing each item
+1 Everything is completely correct (a catch-all point)
-1 You added too many extraneous EVAL or APPLY calls

4. The GET/PUT Table (9 points)

(Part A)
We would like to add connections between nodes into our table as a way to define a maze. The examples in the question have defined a standard way to insert a connection:

(put <Starting point> <Direction> <Ending point>)

Given two nodes, a starting node FROM and an ending node TO, and a direction DIR, we can define a connection in our table using a PUT call:

(put FROM DIR TO)

However, we also need to define the connection in the reverse direction:

(put TO (OPPOSITE-DIR DIR) FROM)

Therefore, the code looks like:

```
(define (connect dir from to)
  (put from dir to)
  (put to (opposite-dir dir) from))
```

Note that CONNECT returns the word "okay" and the function PUT also returns "okay".

(We apologize for the sample call to PUT getting the arguments mixed up!)

Scoring:
[0] for not using put at all
+1 for having the correct argument order for PUT
+1 for having two calls to PUT
+1 for using OPPOSITE-DIR on DIR

(Part B)
For the function TRAVERSE, we are given a starting room and a list of directions. We would like to return the ending room after following the list of directions starting at the starting room.

The standard solution looks like:

```
(define (traverse room dirs)
   (if (null? dirs)
       room
       (let ((new-room (get room (car dirs))))
         (if new-room
             (traverse new-room (cdr dirs))
             #f))))
```

However, we can see that if we query (get #f <some-dirs>), we will get #f back, so the following shorter solution also works:

```
(define (traverse room dirs)
  (if (null? dirs)
      room
      (traverse (get room (car dirs)) (cdr dirs))))
```

A common mistake is to check if the GET call returns an empty list instead of #f.

```
(define (traverse room dirs)
  (cond (...)
        ((NULL? (GET ROOM (CAR DIRS))) ...)
        ...))
```

Another mistake was to not have a null check for DIRS, i.e. forgetting the base case.

We also saw couple solutions with SET!, which received points if the got the SET! exactly right. General SET! solution looks like:

```
(define (traverse room dirs)
  (if (null? dirs)
      room
      (begin
        (set! room (get room (car dirs)))
        (traverse room (cdr dirs)))))
```

Notice that there is no actual reason to use the SET!, since it's immediately used as the argument to TRAVERSE.

Some students hardcoded the list of rooms:

```
(define lists-of-rooms (list 'start 'dead-end 'twisty-passage 'exit))
```

then used the list of rooms to traverse the maze. This solution got 4 points off because it doesn't make use of the table or any other maze, which is the point of the question.

[6] total
-4 for hardcoding rooms as a list
-4 no base case but correct GET call
-2 use null? to check result of a call to GET
-1 DAV
[0] for no recursive call
[0] for no GET call

5. Binary Trees (6 points)

This question was a simple recursive function to build a binary tree
from a list. You were told that you can assume that the list was always
a valid expression, so you didn't have to do any checking for whether
there was enough elements or if the list was empty. Below is a simple solution.

```
(define (make-arith-tree expr)
  (if (list? expr)
      (make-binary-tree (cadr expr)
                        (make-arith-tree (car expr))
                        (make-arith-tree (caddr expr)))
      (make-binary-tree expr '() '())))
      ; ^ This case (where it's not a list) handles numbers at the leaves
```

The most common errors were forgetting that leaves had to be binary
trees or forgetting that binary trees have a separate abstraction from
capital T trees. So many people did one of these two INCORRECT solutions
and only got 4 points:

```
; Incorrect because the leaves are not binary trees, which is failing to
; follow the abstraction properly. Majority of students did a solution
; like this.
(define (make-arith-tree expr)
  (if (list? expr)
      (make-binary-tree (cadr expr)
                        (make-arith-tree (car expr))
                        (make-arith-tree (caddr expr)))
      expr))

; Incorrect because it is not using the specific binary tree abstraction
; as we asked you to do in the problem statement and instead uses the
; Tree abstraction. This was wrong even if you called the constructor
; MAKE-BINARY-TREE!
(define (make-arith-tree expr)
  (if (list? expr)
      (make-tree (cadr expr)
                 (list (make-arith-tree (car expr))
                       (make-arith-tree (caddr expr))))
      (make-tree expr '())))
```

Even more serious errors included forgeting to include a base case, and
forgetting to recursively call the procedure to handle lists like

```
  ((1 + 7) + (4 + 3))
```

which were shown in the examples. Most students got 4 points on this
question because they forgot to handle leaves correctly.

Grading Rubric
6 Perfect
5 Trivial Mistakes
4 "The Idea"
- Forgetting to make a binary tree for the numbers at the leaves of the tree.
- Using the capital-T Tree data abstraction instead of the binary tree data
  abstraction.
- Using only CDRs to attempt to access part of the math expression.
2 "An Idea"
- No base case
- Makes a tree
- Recurses on both the left and right sides of each expression
- Calling MAKE-ARITH-TREE on the operation
0 Other

Postscript: Over half the class got this question wrong, with most answers being
  some variation of the first incorrect solution listed above. Why would this
  be, we wondered. The branches of a binary tree have always been binary trees,
  just like the children of capital-T Trees are always a list of Trees. Didn't
  we have a problem working with binary trees?

  Turns out we didn't, really. We covered them in section, but lab that week
  just used binary trees to make sets, which was given to you. The homework that
  week worked with mobiles. And mobiles DID have plain numbers as leaf nodes.

  Because of this confusion, we decided to give everyone 1pt back on the exam.
  This is a blanket point because the people who got the correct answer might
  have had to spend more time on the problem to get it right. And it's not two
  points because the wrong answer is still wrong!

  This point will not appear in glookup but will be factored into the final
  letter grades at the end of the semester.

6. Message Passing (7 points)

```
  (define (make-array lst)
    (lambda (message)
      (cond
        ((equal? message 'length) (length lst))
        ((number? message) (list-ref lst message))
        (else (position message lst)))))
```

The essentials of a correct solution to this problem are the following:
  1. Return a message-dispatch procedure
  2. Correctly detect all three types of messages: LENGTH, numbers, and words.
  3. Use LIST-REF and POSITION correctly
  4. Correctly ensure that the message "length" does not get interpreted as
      searching for the word "length" in the array.

Common errors:
  Most students answered this question correctly (sometimes with a few trivial errors), but many made the mistake of forgetting that a message is just that -- a message, and not a procedure, or procedure application. This showed up with errors like:

```
   ((equal? message 'length) (message lst))
```

which would produce a bad function error.

  Another source of error was forgetting that list-ref and position exist, trying to redefine them, and then getting them wrong. We were not harsh on this.

  The order of COND clauses and the choice of predicate really matters. Some students chose WORD? as a predicate and had it come first -- which would be true for both words and numbers, and cause those two messages to collide.

Scoring:
+1 for returning a procedure
+1 for returning a message passing procedure
+1 for correctly detecting all messages
+1 for having the length case handled separately from other words
+1 for using POSITION (or similar function) properly
+1 for using LIST-REF (or similar function) properly
+1 for whether or not the written code works (a catch-all point)