MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 2006

Quiz I

Closed Book – one sheet of notes

Throughout this quiz, we have set aside space in which you should write your answers. Please try to put all of your answers in the designated spaces, as we will look only in this spaces when grading.

Note that any procedures or code fragments that you write will be judged not only on correct function, but also on clarity and good programming practice. Also note that while there may be a lot of reading to do on a problem, there is relatively little code to write, so please take the time to read each problem carefully.

NAME:

Section Time: Tutor's Name:

PART	Value	Grade	Grader
1	23		
2	18		
3	24		
4	15		
5	20		
Total	100		

For your reference, the TAs are:

- Arnab Bhattacharyya
- Austin Clements
- Estanislao Fidelholtz
- Harold Fox
- Sumudu Watugala
- Tom Wilson
- Tom Yeh

Part 1: (23 points)

For each of the following expressions or sequences of expressions, state the value returned as the result of evaluating the final expression in each set, after evaluating any previous expressions in the set; or indicate that the evaluation results in an error. If the result is an error, state in general terms what kind of error (e.g. you might write "error: wrong type of argument to procedure"). If the evaluation returns a built-in procedure, write `primitive procedure`. If the evaluation returns a user-created procedure, write `compound procedure`.

For some of the questions, we also ask you to identify the "type" of the returned expression, using the notation introduced in lecture (assuming that the expression does not result in an error).

You may assume that evaluation of each sequence takes place in a newly initialized Scheme system.

Question 1.

```
(lambda (x y) (* 2 x))
```

Value: Type:

Question 2.

```
(2 * 3)
```

Value: Type:

Question 3.

```
((lambda (x y)
    (* (+ x y)
       (/ x y)))
 6
 3)
```

Value: Type:

Question 4.

```
(define x 1)
(define y 2)
(define (doit x)
  (let ((y 3))
    (list x y)))
(doit 4)
```

Value:

Question 5.

```
(((lambda (x)
     (lambda (y)
        (- (* x x) (* y y))))
  5)
 3)
```

Value: Type:

Question 6.

```
((lambda (x)
     (lambda (y)
        (- (* x x) (* y y))))
  5)
```

Value: Type:

Question 7.

```
(list + 2 3)
```

Value:

Part 2: (18 points)

You have been hired by the EECS department to help them keep track of student enrollment numbers – how many students are taking Course VI classes each term. To do this, you have built a small database with the following characteristics:

- For each class, we have a data abstraction consisting of a class-tag and a collection of class-statistics. The tag is a class number (e.g. 6.001), and the collection of statistics is a list of term information. We use the constructor `make-class` to create such abstractions.
- Given a data structure for a class, the selector `class-tag` will return the tag, and the selector `class-stats` will return the statistics. For example

```
(define testclass1
  (make-class 6.001 (list (make-term 2006.1 200)
                          (make-term 2006.2 350)
                          (make-term 2005.1 245)
                          (make-term 2005.2 342))))

(class-tag testclass1)
6.001

(class-stats testclass1)
((2006.1 200) (2006.2 350) (2005.1 245) (2005.2 342))
```

- For an individual term from a list of class-statistics, there is a data abstraction that includes information about the term (e.g. 2006.1 for spring of 2006, 2006.2 for fall of 2006), and information about the number of students registered that term. We use the constructor `make-term` to create such abstractions.
- The selector `term` will retrieve the term information from a specific entry in the list of statistics, the selector `registered` will retrieve the number of registered students. For example

```
(term (car (class-stats testclass1)))
2006.1
```

- The entire database is a collection of class data abstractions. To determine if there are any classes in the database, we use the predicate `no-classes?`. To extract the next class in the database, we use `next-class` and to extract everything but the next class, we use `rest-classes`. For example

```
(define testclass2
  (make-class 6.002 (list (make-term 2006.1 234)
                          (make-term 2006.2 198)
                          (make-term 2005.1 245)
                          (make-term 2005.2 183))))
```

```
    (define testclass3
      (make-class 6.003 (list (make-term 2006.1 192))))

    (define testdata (list testclass1 testclass2 testclass3))

    (rest-classes testdata)
    ((6.002 (2006.1 234) (2006.2 198) (2005.1 245) (2005.2 183))
     (6.003 (2006.1 192)))
```

To make this clearer, here is the code for these abstractions:

```
(define (make-class tag terms)
   (cons tag terms))
(define class-tag car)
(define class-stats cdr)

(define no-classes? null?)
(define next-class car)
(define rest-classes cdr)

(define make-term list)
(define term car)
(define registered cadr)
```

Question 8. We want to compute the total number of students that have taken a particular class:

```
(total-students 6.001 testdata)
1137
```

where

```
(define (total-students class data)
  (cond ((no-classes? data) (error "class not found"))
        ((= class (class-tag (next-class data)))
         (add-em-up (class-stats (next-class data))))
        (else (total-students class (rest-classes data)))))
```

Complete the procedure by providing code for `add-em-up`, using a procedure that is linear recursive.

Complete the procedure by providing code for `add-em-up`, using a procedure that is iterative.

Question 9. Suppose we want to compute the average load in each class – that is, the average number of students who take each class each term. Here is some code:

```
(define (number-terms class data)
  ;; computes the number of terms for which there is data about registration
  (cond ((no-classes? data) (error "class not found"))
        ((= class (class-tag (next-class data)))
         (length (class-stats (next-class data))))
        (else (number-terms class (rest-classes data)))))

(define (get-class-tags-as-list data)
  (if (no-classes? data)
      '()
      (cons (class-tag (next-class data))
            (get-class-tags-aslist (rest-classes data)))))

(define (average-load data)
  (let ((tags (get-class-tags-as-list data)))
    (compute-average-per-class tags data)))

(average-load testdata)
((6.001 284.25) (6.002 215) (6.003 192))
```

Write the procedure `compute-average-per-class`

Part 3: (24 points)

In this part, we are going to use the selectors from the data abstractions of Part 2, to create a new data abstraction.

Question 10. We would like to convert our database into a new form. Our goal is to create a new kind of database, in which information about a class is stored as a set of nested lists. For example, the information from `testclass1` in this new form would be:

```
(((6.001 2006.1) 200)
 ((6.001 2006.2) 350)
 ((6.001 2005.1) 245)
 ((6.001 2005.2) 342))
```

in other words, we want to combine the class tag and the term information into a single list, which is then combined with the registration data for each term.

Given the following:

```
(define (convert-class class)
  (helper (class-tag class)
          (class-stats class)))
```

write the procedure `helper`

Question 11. Now we want to use this to convert each class structure into our database into this form, and to concatenate all of this information into a single list, e.g.

```
(define new-testdata (convert-all testdata))

new-testdata
(((6.001 2006.1) 200)
 ((6.001 2006.2) 350)
 ((6.001 2005.1) 245)
 ((6.001 2005.2) 342)
 ((6.002 2006.1) 234)
 ((6.002 2006.2) 198)
 ((6.002 2005.1) 245)
 ((6.002 2005.2) 183)
 ((6.003 2006.1) 192))
```

Here is part of the procedure:

```
(define (convert-all data)
  (if (no-classes? data)
      '()
      TO-BE-COMPLETED))
```

Complete this procedure by supplying the missing code.

Once we have converted our database, we can then get information from the database by finding entries in the database that match a specific pattern. For example:

```
(define (extract tester data)
  (cond ((null? data) '())
        ((tester (car data)) (cons (car data) (extract tester (cdr data))))
        (else (extract tester (cdr data)))))
```

can be used to retrieve information from this new formatted database. The key is to create a tester that will look for specific information in each entry in the database.

Question 12.

To get all the information about a specific class, we can use, for example,

```
(extract (make-class-extractor 6.001) new-testdata)

(((6.001 2006.1) 200)
 ((6.001 2006.2) 350)
 ((6.001 2005.1) 245)
 ((6.001 2005.2) 342))
```

What should be the definition of `make-class-extractor`?

Question 13.

To get all the information about a specific class and term, we can use, for example,

```
(extract (make-term-class-extractor 6.001 2005.2) new-testdata)

(((6.001 2005.2) 342))
```

What should be the definition of `make-term-class-extractor`? (You may find it helpful to use `equal?` which compares two lists and returns `true` if the lists are the same.)

Part 4: (15 points)

Here are two procedures to reverse the elements of a list:

```
(define (rev lst)
  (define (help done todo)
     (if (null? todo)
         done
         (help (cons (car todo) done) (cdr todo))))
  (help '() lst))

(define (rev1 lst)
  (if (null? lst)
      '()
      (append (rev1 (cdr lst))
              (list (car lst)))))

(define (append l1 l2)
  (if (null? l1)
      l2
      (cons (car l1) (append (cdr l1) l2))))
```

We would like to measure the order of growth in time (as measured by the number of primitive operations in the computation) and in space (as measured by the maximum number of deferred operations – do not count in space the intermediate data structures constructed by the algorithm).

For each of the following questions, choose the description from these options that best describes the order of growth of the process.

A: constant
B: linear
C: exponential
D: quadratic
E: logarithmic
F: something else

Question 14. What is the order of growth in time of the procedure `rev`, when applied to a list of n elements?

Question 15. What is the order of growth in space of the procedure `rev`, when applied to a list of n elements?

Question 16. What is the order of growth in time of the procedure `rev1`, when applied to a list of `n` elements?

Question 17. What is the order of growth in space of the procedure `rev1`, when applied to a list of `n` elements?

Part 5: (20 points)

Consider the following procedures:

```
(define (comp f g)
  (lambda (x) (f (g x))))

(define (repeat f n)
  (if (= n 1)
      f
      TO-BE-COMPLETED))
```

Question 18. The goal of `repeat` is to return a procedure of one argument that has the effect, when applied to an argument, of recursively apply the argument `f` to the previous result `n` times. Here are two possible completions to this code:

A. `(comp f (repeat f (- n 1)))`

B. `(comp (repeat f (- n 1)) f)`

Which of the following best describes the behavior of this code:

1. Only option A will work as described
2. Only option B will work as described
3. Both option A and B will work as described
4. Neither option A nor B will work as described

Question 19. Assuming that `repeat` is correctly coded, use it to complete the following procedure so that it performs multiplication by successive addition. In other words, `(mul a b)` should return the product of `a` and `b` using only the primitive operation of addition.

```
(define (mul a b)
  TO-BE-COMPLETED)
```

Question 20. Assuming that `repeat` is correctly coded, use it to complete the following procedure so that it performs exponentiation by successive multiplication. In other words, `(my-exp a b)` should return a^b using only the primitive operation of multiplication.

```
(define (my-exp a b)
  TO-BE-COMPLETED)
```