MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 2005

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	28		
2	28		
3	24		
4	20		
Total	100		

For your reference the list of TA's is:

- Tom Wilson
- I-Ting Angelina Lee
- Chang She
- J.D. Zamfirescu

Part 1: (28 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, 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`.

If the expression does not result in an error, also state the "type" of the returned expression, using the notation introduced in lecture.

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

Question 1.

```
+
```

Value: [ ] Type: [ ]

Question 2.

```
(1 + 3)
```

Value: [ ] Type: [ ]

Question 3.

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

Value: [ ] Type: [ ]

Question 4.

```
((lambda (x y)
   (+ (* x x)
      (/ y y)))
 5
 3)
```

Value: [ ] Type: [ ]

Question 5.

```
(define x 5)
(define y 2)
(define (doit x)
  (let ((y 1))
    (list x y)))
(doit 3)
```

Value: [ ] Type: [ ]

Question 6.

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

Value: [ ] Type: [ ]

Question 7.

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

Value: [ ] Type: [ ]

Part 2: (28 points)

We are going to work with a simple database that represents a student's transcript. A transcript will consist of a sequence of terms, and each term will consist of a set of subjects and grades. We will, for simplicity, represent subjects by their numerical value (e.g. 6.001), and we will represent grades by their numerical values (e.g. an A is a 5, an A- is a 4.7, a B+ is a 4.3, a B is a 4, etc.).

We are going to assume some data abstractions for use in our system. Our most basic unit will be a subject/grade pair, which we will call a subject record (or just a `subjectrec` for short). Our constructor for an individual subjectrec is `make-subjectrec` with selectors `subject` and `grade`:

```
(subject (make-subjectrec 6.001 5))
6.001

(grade (make-subjectrec 6.001 5))
5
```

To represent a term, we will have:

- a procedure that creates an empty term, `make-empty-term`,
- `add-subjectrec` will add a subjectrec to an existing term, and return the new term,
- to get the first subjectrec in a term, we will use `first-subjectrec`
- to get everything but the first subjectrec, we will use `rest-term`, and
- we will test if a term is empty using `end-term?`.

```
(define ft04
  (add-subjectrec
     (make-subjectrecrec 6.001 5)
     (add-subjectrec (make-subjectrec 18.01 4.3)
                     (add-subjectrec (make-subjectrec 8.01 4)
                                     (add-subjectrec (make-subjectrec 14.001 3)
                                                     (make-empty-term))))))
```

Similarly, to represent a transcript, we will have:

- a procedure that creates an empty transcript, `make-empty-transcript`,
- `add-term` will add a term to an existing transcript, and return the new transcript,
- to get the first term in a transcript, we will use `first-term`
- to get everything but the first term, we will use `rest-transcript`, and
- we will test if a transcript is empty using `end-transcript?`.

Suppose we want to compute a term's GPA (grade point average), which is simply the average numerical grade for the courses during the term (don't worry about the fact that courses might have different units).

We are going to do this in stages.

Question 8.

Write a procedure, `term-sum`, that takes as input a term, and returns the sum of the grades for the subjectrecs in that term. For example:

```
(term-sum ft04)
16.3
```

Here is a template for the procedure, you need to insert the missing parts:

```
(define (term-sum term)
  (if INSERT-1
      0
      (+ INSERT-2
         INSERT-3)))
```

INSERT-1:

INSERT-2:

INSERT-3:

Question 9.

Similarly, write a procedure, `term-subjects`, that takes as input a term, and returns the total number of subjectrecs in that term. For example:

```
(term-subjects ft04)
4
```

Using this, we can then create

```
(define (term-gpa term)
  (/ (term-sum term) (term-subjects term)))
```

so that

```
(term-gpa ft04)
4.075
```

Here is a template for the procedure, you need to insert the missing parts:

```
(define (term-subjects term)
  (if INSERT-4
      0
      (+ INSERT-5
         INSERT-6)))
```

INSERT-4: will be the same as INSERT-1 in Question 8 above.

INSERT-5:

INSERT-6:

Question 10. Now suppose we want to get the GPA for a full transcript.

You are to write a procedure, `overall-sum`, that takes as input a transcript, and returns the sum of the grades for all the subjectrecs in the transcript. You should use `term-sum` as part of your solution. For example:

```
(overall-sum test-transcript)
31.0
```

Once we have `overall-sum`, clearly we can write a very similar procedure, `overall-subjects`, that takes as input a transcript, and returns the total number of subjectrecs in the transcript. For example:

```
(overall-subjects test-transcript)
8
```

If we assume that `overall-subjects` has been completed, we can then create:

```
(define (overall-gpa term)
  (/ (overall-sum term) (overall-subjects term)))
```

so that

```
(overall-gpa test-transcript)
3.875
```

Write `overall-sum`:

Question 11.

Suppose we want to find the subjectrec for a particular subject out of a full transcript. Your job is to write a procedure `find-subjectrec-in-term` that looks for the subject record in a particular term. Your procedure will be used by:

```
(define (find-subjectrec subj transcript)
  (if (end-transcript? transcript)
      #f
      (let ((temp (find-subjectrec-in-term subj (first-term transcript))))
        (if temp
            temp
            (find-subjectrec subj (rest-transcript transcript))))))

>(find-subjectrec 8.01 ft04)
(8.01 4)
```

You may assume that a subject appears at most once in a transcript.

Write the procedure `find-subjectrec-in-term`:

Part 3: (24 points)

In this part we are going to explore a different way to compute GPA's. While we will rely on the same data abstractions, you can otherwise treat this part as independent of Part 2.

Question 12.

A different way of computing a GPA is to first convert the information from a term into a list of values, and then apply list operations. For example, to compute the GPA of a term, we could first convert the term into a list of grades, then add them up, and divide by the number of subjectrecs.

The following procedure is a general purpose way of converting a data structure into a list, including doing some processing on each element as it is added to the list. `Collection` is some data structure to be processed, which has associated selectors `front` to get the first element, `rest` to get the remaining elements, and `end-test?` to tell if we are at the end of the data structure. `Convert` is used to do something to each element as it is extracted from the data structure.

```
(define (convert-to-list convert front rest end-test? collection)
  (if (end-test? collection)
      '()
      (cons (convert (front collection))
            (convert-to-list convert front rest end-test? (rest collection)))))
```

Using this procedure, plus the procedure `foldr`:

```
(define (foldr proc init lst)
  (if (null? lst)
      init
      (proc (car lst) (foldr proc init (cdr lst)))))
```

write a new procedure to compute a term's GPA:

Question 13.

Suppose we want to get a list of all of the subjects in a transcript. We can again use the concepts of `convert-to-list` and `foldr`. You may also find the following procedure useful:

```
(define (append lst1 lst2)
  (if (null? lst1)
      lst2
      (cons (car lst1) (append (cdr lst1) lst2))))
```

Write the procedure `get-all-subjectrecs` that takes a transcript as input and returns a list of all the subjectrecs, e.g.

```
>(define my-test (get-all-subjectrecs test-transcript))

>my-test
((6.001 5) (18.01 4.3) (8.01 4) (14.001 3) (6.002 3) (18.02 4.7) (8.02 3.7) (14.002 3.3))
```

Question 14.

Once we have a transcript as a list of subjectrecs, we can use list operations to compute a GPA. Using `foldr` and `map` defined as

```
(define (map proc lst)
  (if (null? lst)
      '()
      (cons (proc (car lst))
            (map proc (cdr lst)))))
```

complete the following procedure to compute a GPA:

```
(define (new-gpa transcript)
  (let ((lst (get-all-subjectrecs transcript)))
    INSERT-HERE))
```

Part 4: (20 points)

Suppose that we want to sort a list of elements (e.g. a list of subjectrecs from the previous part – although you can treat this part as independent of the previous part). Here is a procedure for sorting:

```
(define (find-best best todo compare)
  (if (null? todo)
      best
      (if (compare (car todo) best)
          (find-best (car todo) (cdr todo) compare)
          (find-best best (cdr todo) compare))))

(define (remove elt todo same)
   (if (null? todo)
       nil
       (if (same elt (car todo))
           (cdr todo)
           (cons (car todo) (remove elt (cdr todo) same)))))

(define (sort data compare same)
  (let ((trial (find-best (car data) (cdr data) compare)))
     (let ((todo (remove trial data same)))
       (if (null? todo)
           (list trial)
           (cons trial (sort todo compare same))))))
```

For example, to sort our data by increasing subject number, we would evaluate:

```
(sort (get-all-subjectrecs transcript)
      (lambda (x y) (< (subject x) (subject y)))
      eq?)
```

We are going 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), measured as a function of the size of data, denoted by n. Assume that the procedures used for `compare` and `same` use constant time and space.

For each of the following questions, choose the description from these options that best describes the order of growth of the process. If you select "something else", please state why.

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

Question 15. What is the order of growth in time of the procedure `find-best`?

Question 16. What is the order of growth in space of the procedure `find-best`?

Question 17. What is the order of growth in time of the procedure `remove`?

Question 18. What is the order of growth in space of the procedure `remove`?

Question 19. What is the order of growth in time of the procedure `sort`? Remember to include the effect of `find-best` and `remove`.

Question 20. What is the order of growth in space of the procedure `sort`? Remember to include the effect of `find-best` and `remove`.