

MASSACHUSETTS INSTITUTE OF TECHNOLOGY  
 Department of Electrical Engineering and Computer Science  
 6.001—Structure and Interpretation of Computer Programs  
 Spring Semester, 2007

**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 space 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	20		
2	22		
3	18		
4	22		
5	18		
Total	100		

For your reference, the TAs are:

- Matt Brown
- Austin Clement
- Michael Craig
- Stephen McCamant
- Brennan Sherry
- James Wnorowski
- Sarah Wu

**Part 1: (20 points)**

For each of the following 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 each question, 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 (a b)
  (- (/ a b)
    (+ a b)))
 8 4)
```

Value:  Type:

**Question 2:**

```
((lambda (a + b)
  (+ a b))
 3 * 4)
```

Value:  Type:

**Question 3:**

```
((lambda (x)
  (lambda (y)
    (expt x y)))
 3)
```

Value:  Type:

**Question 4:**

```
(lambda (a b) (* 2 a))
```

Value:  Type:

**Question 5:**

```
(define x 3)  
(define y 4)  
((lambda (x) (* x y)) 2)
```

Value:  Type:

In the rest of this quiz, we are going to help Ben Bitdiddle plan his entry for his dorm’s NCAA “March Madness” betting pool. If you are not a basketball fan, don’t worry – you don’t need to know about basketball to answer the questions.

While the questions all follow a common theme and build on one another, they are all independent: Even if you get a question wrong, or leave it blank, you can still go on to the rest of the questions and answer them. In those later questions you can use the names of the functions you were asked to write in earlier questions, whether or not you got that previous question correct.

Some quick background. Every March, the NCAA hosts a tournament to select the national collegiate basketball champion. Seedings for this tournament (that is, where a selected team is placed) are very important, since they determine what competition each team will face. Tournaments are run as a single elimination event, that is, once you lose you are done. In the NCAA tournament, for example, there are four brackets, each with 16 teams, and the tournament is set up to favor teams with better (lower) seeds. In a bracket with 16 teams, the team seeded 1 would first play the team seeded 16, 2 would play 15, and so on. The winner of the 1 versus 16 game would then face the winner of the 8 versus 9 game, and so on. An example tournament seeding is shown below, with the winner of each game moving on to the next round.



To select the seedings, the selection committee looks at many factors, one of which is a “power rating” that measures how well a team performed during the regular season. Listed below is an example bracket for the tournament, where each “entry” includes the team name (as a string), their seeding, and their power ranking. Notice that the ordering of this bracket is set up so that each pair of teams defines a first round game, and the winners of each adjacent pair defines the next round game, and so on.

```
(define test-bracket
  (list (make-entry "UCLA" 1 2)
        (make-entry "Kansas" 16 5)
        (make-entry "Tennessee" 8 21)
        (make-entry "Kentucky" 9 17)
        (make-entry "Southern Illinois" 5 13)
        (make-entry "UNLV" 12 25)
        (make-entry "Pittsburgh" 4 10)
        (make-entry "Duke" 13 16)
        (make-entry "Arizona" 14 18)
        (make-entry "UNC" 3 3)
        (make-entry "Maryland" 11 11)
        (make-entry "Wisconsin" 6 4)
        (make-entry "Florida" 10 6)
        (make-entry "Memphis" 7 7)
        (make-entry "Texas AM" 15 8)
        (make-entry "Ohio State" 2 1)))
```

Ben is interested in trying to predict the winner of the tournament (and as a consequence the winner of each game), and we are going to help him out.

**Part 2: (22 points)**

First, the elements of a bracket are a list of entries, created using:

```
(define (make-entry team seed ranking)
  (list team seed ranking))
```

**Question 6:** Write an implementation for the accessors to complete this data abstraction, specifically

entry-team

entry-seed

entry-ranking

Ben's goal is to predict winners of each game. For now, he assumes that there will be some procedure to predict the winner for any pair of teams. With that assumption, the following procedure will take a bracket, and select the winner for each game, returning a new bracket with half as many teams (just the winners). The assumption is that the procedure to which `probable-winner` is bound will return one of its two arguments, based on some logic that we will get to. Note that a bracket is represented as a list.

```
(define (run-a-round bracket probable-winner)
  ;; okay to assume that there are an even number of elements in bracket
  (if (null? bracket)
      bracket
      (cons (probable-winner (car bracket) (cadr bracket))
            (run-a-round (cddr bracket) probable-winner))))
```

For example, applying `run-a-round` to `test-bracket` together with an appropriate argument for `probable-winner` might return

```
(("UCLA" 1 2)
 ("Tennessee" 8 21)
 ("Southern Illinois" 5 13)
 ("Pittsburgh" 4 10)
 ("Arizona" 14 18)
 ("Maryland" 11 11)
 ("Florida" 10 6)
 ("Ohio State" 2 1))
```

### Question 7:

Rewrite `run-a-round` in iterative form. Be sure that the result preserves the order that you saw with the recursive version (you may want to use `reverse`, which is a builtin Scheme procedure that reverses the elements of a list).

**Question 8:**

To complete the evaluation of a bracket, we need to keep running rounds until there is only one team left. Complete the following procedure to do this, using `run-a-round`. For example, applying `run-all-rounds` to `test-bracket` together with an appropriate argument for `probable-winner` might return `("UCLA" 1 2)`. **Note that we want this procedure to return a single entry, not a list of one entry.** Complete this procedure below.

```
(define (run-all-rounds bracket probable-winner)
```

**Part 3: (18 points)****Question 9:**

Now we need a way of predicting each winner. Ben's first idea is to take the ranking of each team, call them  $r_1$  and  $r_2$ . He predicts that the probability that the first team will win is given by

$$\frac{r_2}{r_1 + r_2}.$$

The Scheme procedure `random`, when called with no arguments will return a random number between 0 and 1. Using this, create a procedure that will return one of the two teams, depending on whether the random number selected is greater than the ratio above:

```
(define winner-ranking (lambda (team1 team2)
```

**Question 10:** An alternative idea is to use the same logic, but with the team's seeding rather than the team's power ranking; if the seedings of the two teams are  $s_1$  and  $s_2$ , then the probability that the first team will win is given by

$$\frac{s_2}{s_1 + s_2}.$$

Ben could just write a procedure to capture this idea, but instead he decides to generalize the previous procedure in the following way. He wants a procedure `generate-winner` so that:

```
(define winner-ranking (generate-winner entry-ranking))
```

is equivalent to the code written for the previous question, but

```
(define winner-seed (generate-winner entry-seed))
```

would provide a procedure that bases winners on seedings.

Write the procedure `generate-winner`:

```
(define (generate-winner accessor)
```

**Part 4: (22 points)**

Now suppose that Ben runs his code for the tournament many times. Because of the uncertainty in deciding each game winner, it is possible that he will get different results, although some results will be replicated. He uses the following code to create a list of results:

```
(define (gather-results bracket probable-winner n)
  (if (= n 0)
      '()
      (cons (run-all-rounds bracket probable-winner)
            (gather-results bracket probable-winner (- n 1)))))
```

For example:

```
(define test-data (gather-results test-bracket winner-ranking 10))
```

might result in

```
(("UCLA" 1 2)
 ("UCLA" 1 2)
 ("Tennessee" 8 21)
 ("Pittsburgh" 4 10)
 ("Florida" 10 6)
 ("Ohio State" 2 1)
 ("Ohio State" 2 1)
 ("UCLA" 1 2)
 ("Ohio State" 2 1)
 ("UCLA" 1 2)
 )
```

He now wants to analyze those results. To do this, he is going to build a **histogram** – this is a data structure that counts the number of times an element appears in a list. Because each result has several pieces of information, Ben wants to be able to decide on the information to be histogrammed, for example, he might want to know how often a given team (e.g., **UCLA**) wins, or how often a team with a given seed (e.g. the first seed) wins, or how often a team with a given ranking wins.

So we need a procedure such that, for example, evaluating `(get-elts entry-team test-data)` would return

```
("UCLA" "UCLA" "Tennessee" "Pittsburgh" "Florida" "Ohio State" "Ohio State"
 "UCLA" "Ohio State" "UCLA")
```

and evaluating `(get-elts entry-seed test-data)` would return

```
(1 1 8 4 10 2 2 1 2 1)
```

**Question 11:** Using some combination of `map` and/or `filter`, write an implementation of `get-elts`. For your convenience, we have

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

(define (filter pred lst)
  (cond ((null? lst) '())
        ((pred (car lst)) (cons (car lst) (filter pred (cdr lst))))
        (else (filter pred (cdr lst)))))
```



Ben's idea is to use `map`, `filter` and `length` to convert the list returned by `get-elts` into a histogram. We need a data structure for histograms:

```
(define empty-histogram '())
(define empty-histogram? null?)
(define first-histogram car) ; returns first element
(define rest-histogram cdr) ; returns rest of histogram
(define adjoin-histogram cons)

(define (histogram-element key num) (list key num)) ; unit of histogram
(define histogram-key car) ; key is thing being counted
(define histogram-number cadr) ; number is times it appears
```

Ben now wants a procedure `build-hist` so that

```
(build-hist (get-elts entry-team test-data) string=?)
```

would return a histogram, in which, for example, the key `UCLA` would be paired with the number 4 since `UCLA` appears four times in the example.

**Question 12:** We want you to complete the code for `build-hist`, which will complete the data abstraction. Note that the parameter `same?` in `build-hist` will be used to compare keys (in the example above we used `string=?` which compares two strings to decide if they are the same), and `elts` is a list of keys.

```
(define (build-hist elts same?)
  (if (null? elts)
      empty-histogram
      (let ((next (car elts)))
        (adjoin-histogram
          (histogram-element next ANSWER-4)
          (build-hist ANSWER-5 same?))))))
```

Using some combination of `map`, `filter` and/or `length`, write an expression for `ANSWER-4`. The basic idea is to process the list `elts` to determine the number of times that the element referenced by `next` appears.

What expression should be used for `ANSWER-5`, so that a histogram for the remaining elements is constructed? Use some combination of `map`, `filter` and/or `length`.

**Part 5: (18 points)**

Ben now can create histograms of results, but to make things clearer, he wants to sort them with the most common element first. Here is some code to do this:

```
(define (sort-hist hist)
  ; returns histogram sorted in descending order of frequency
  ; e.g. if hist=(("Florida" 1) ("UCLA" 5) ("Ohio State" 3))
  ; then the result is (("UCLA" 5) ("Ohio State" 3) ("Florida" 1))
  (if (empty-histogram? hist)
      empty-histogram
      (let ((best (get-max hist)))
        (adjoin-histogram best (sort-hist (remove best hist))))))

(define (get-max hist)
  ; returns the maximum element of the histogram,
  ; e.g. if hist=(("Florida" 1) ("UCLA" 5) ("Ohio State" 3)),
  ; then the result is ("UCLA" 5)
  (if (empty-histogram? hist)
      #f
      (get-max-h (first-histogram hist) (rest-histogram hist))))

(define (get-max-h best rest)
  ; returns the maximum of best and the rest of the histogram
  (if (empty-histogram? rest)
      best
      (if (> (histogram-number best)
              (histogram-number (first-histogram rest)))
          (get-max-h best (rest-histogram rest))
          (get-max-h (first-histogram rest) (rest-histogram rest)))))

(define (remove elt hist)
  ; removes an element from the histogram, e.g.
  ; e.g. if elt is ("UCLA" 5) and hist is (("Florida" 1) ("UCLA" 5)),
  ; then the result is (("Florida" 1)).
  (if (empty-histogram? hist)
      empty-histogram
      (if (equal? elt (first-histogram hist))
          (rest-histogram hist)
          (adjoin-histogram (first-histogram hist)
                            (remove elt (rest-histogram hist))))))
```

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

For the questions below, choose your answer from:

**Question 13:** What is the order of growth in time of the procedure `get-max` measured in terms of the length of the histogram?

What is the order of growth in space of the procedure `get-max`, measured as the maximum number of deferred operations, as a function of the length of the histogram?

**Question 14:** What is the order of growth in time of the procedure `remove` measured in terms of the length of the histogram?

What is the order of growth in space of the procedure `remove`, measured as the maximum number of deferred operations, as a function of the length of the histogram?

**Question 15:** What is the order of growth in time of the procedure `sort-hist` measured in terms of the length of the histogram?

What is the order of growth in space of the procedure `sort-hist`, measured as the maximum number of deferred operations, as a function of the length of the histogram?