

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
 Department of Electrical Engineering and Computer Science
 6.001—Structure and Interpretation of Computer Programs
 Fall Semester, 2004

Quiz I – Example Solutions

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.

NAME:

Section Number:

Tutor's Name:

PART	Value	Grade	Grader
1	20		
2	36		
3	28		
4	16		
Total	100		

For your reference:

Section	Time	Location	Rec. Instructor	Tutors
10	10:00	26-328	Seth Teller	Ben Vandiver
11	11:00	26-314	Srini Devadas	David Ziegler
12	12:00	26-3314	Srini Devadas	David Pritchard
1	1:00	26-372	Trevor Darrell	Marshall Tappen
2	2:00	26-372	Trevor Darrell	All

Part 1: (20 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 expression does not result in an error, also state the “type” of the returned value, using the notation from lecture. 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**, and its **type**. If the evaluation returns a user-created procedure, write **compound procedure**, and also indicate its **type** using the notation we introduced in lecture. You may assume that evaluation of each sequence takes place in a newly initialized Scheme system.

Question 1.

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

Value (or error):**11**Type (or kind of error if appropriate):**Number****Question 2.**

```
(lambda (a b)  
  (if (> a 0)  
      b  
      (lambda (x) (b (- x)))))
```

Value (or error):**Compound procedure**Type (or kind of error if appropriate):**Number, (Number \mapsto A) \mapsto (Number \mapsto A)**

Question 3.

```
(lambda (p)
  (lambda (y)
    (if (p y) y (abs y))))
```

Value (or error):**Compound procedure**

Type (or kind of error if appropriate):**(number \rightarrow boolean) \mapsto (number \mapsto number)**

Question 4.

```
(define (square x) (* x x))
(define (test f n)
  (if (= n 0)
      (lambda (x) x)
      (lambda (x)
        (f ((test f (- n 1)) x)))))
```

```
((test square 3) 2)
```

Value (or error):**256**

Type (or kind of error if appropriate):**Number**

Part 2: (36 points)

If you are new to Boston, you may have noticed that long-time Bostonians are very serious about their Red Sox. Once again, the Red Sox have made it to the playoffs and are hoping to end their 86 year World Series' drought. Unfortunately, Red Sox fans are accustomed to seeing their team find inventive ways to lose in the playoffs. Hence, we are going to help the team, by building a small database of baseball batting statistics, so that the manager can select the best lineup. Don't worry if you don't know much about baseball; the questions should be self-explanatory.

Here is the definition of a data structure for storing information about an individual player: it includes information about the player, his number of appearances at the plate, and the number of hits and walks in those appearances. First, we have a data structure for hits:

```
(define (make-hit-stats singles doubles triples homers)
  (list singles doubles triples homers))
```

We will denote the **type** of the data structure created by this constructor as type: **Hits**.

For a player's statistics, we have

```
(define (make-player uniform-number plate-appearances hits walks)
  (list uniform-number plate-appearances hits walks 0))
```

We will denote the **type** of the data structure created by this constructor as type: **Stats**. Note that the last element of this data structure is the value 0. We will use this part of the data structure for the batter's average, which we will need to compute (see below).

Question 5. For example

```
(define manny-hits (make-hit-stats 87 44 0 43))
(define manny-stats (make-player 6 644 manny-hits 82))
```

Draw a box-and-pointer diagram of the value associated with `manny-stats`.



Question 6. Write accessors (or selectors) to extract the appropriate information from the data structure, with the behavior indicated by the type specification. You may assume that the underlying data structure is a list, and use some appropriate choice of `car`, `cdr`, `cadr` and so on; or of `first` and `rest`; or using `list-ref`:

```
plate-appearances: Stats --> number
hits: Stats --> Hits
doubles: Stats --> number
```

Be sure to write all three accessors in the space below:

```
(define (plate-appearances stats) (cadr stats))

(define (hits stats) (caddr stats))

(define (doubles stats)
  (cadr (hits stats)))
```

Question 7.

We want to modify each player's statistics to include their batting average, which is defined as the total number of hits (singles + doubles + triples + homers) all divided by the number of plate appearances less the number of walks, i.e. $\frac{s+d+t+h}{pa-w}$. Initially, we set the average to 0; we now want to compute the correct value.

Complete the definition of a procedure `add-average` so that the player's `Stats` data structure is updated to include the player's actual average at the end of the list. Do this using some combination of `map`, `filter`, `fold-right` and appropriate data abstractions. Note that we are using `cons`, `car` and `cdr` to manipulate lists.

```
(define new-manny-stats (add-average manny-stats))

new-manny-stats
;Value: (6 644 (87 44 0 43) 82 .310)

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

(define (but-last lst)
  (if (null? (cdr lst))
      nil
      (cons (car lst) (but-last (cdr lst)))))

(define (add-average player)
  (append (but-last player) ADD-YOUR-CODE))
```

You may find the following definitions useful:

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

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

(define (fold-right op init lst)
  (if (null? lst)
      init
      (op (car lst) (fold-right op init (cdr lst)))))
```

What should be used in place of `ADD-YOUR-CODE` in `add-average`.

```
(list (/ (fold-right + 0 (hits player))
        (- (plate-appearances player) (walks player))))
```

Question 8.

Now we want to apply this change to all the members of the team. Assume that a team is represented as a list of player `Stats` data structures. Write a **linear recursive** procedure, `update-team`, that creates a new team list with averages included, so that if `bosox` were a list of team players created using the initial data structure for players, then

```
(define new-bosox (update-team bosox))
```

would create a new version of the team statistics.

```
(define (update-team team)
  (if (null? team)
      nil
      (cons (add-average (car team))
            (update-team (cdr team)))))
```

Question 9. Write an iterative version of `update-team`. Don't worry if the order of the players is not maintained.

```
(define (update-team team)
  (define (aux todo done)
    (if (null? todo)
        done
        (aux (cdr todo)
              (cons (add-average (car todo)) done))))
  (aux team nil))
```


Part 3: (28 points)

Now, we want to sort our statistics, to find the player with the best average. This means we need a way of sorting a list. For simplicity, we will assume that we are given a list of numbers (i.e., we have extracted the averages from the team list of players). Consider the following code:

```
(define (bubble-up lst)
  (define (aux best seen rest)
    (if (null? rest)      ; if no more elements
        (cons best seen) ; add largest to front of other elements
        (if (> best (car rest))
            (aux best (cons (car rest) seen) (cdr rest))
            ;; keep best, add next element to seen, move on to rest
            (aux (car rest) (cons best seen) (cdr rest))
            ;; have new best, add old best to seen, move on to rest
            )))
  (aux (car lst) nil (cdr lst)))

(define (bubble-sort lst)
  (if (null? lst)
      nil
      (let ((trial (bubble-up lst))) ;; get best element and others
        (cons (car trial)           ; put best at front
              (bubble-sort (cdr trial)))))) ;; sort rest of list
```

Look at this code carefully. Assume that we want to measure the order of growth of these procedures in space, **measured by the maximum number of deferred operations at any point in the computation**, and the order of growth in time, **measured by the total number of primitive operations** (which here we restrict to `cons`, `car`, `cdr`, `null?` and `>`).

For each order of growth question, you should choose from:

- $\Theta(1)$
- $\Theta(\log n)$
- $\Theta(n)$
- $\Theta(n \log n)$
- $\Theta(n^2)$
- $\Theta(2^n)$
- other

Question 11.

What is the order of growth in space for `bubble-up`, where n is the length of the input argument?

Your answer: $\Theta(1)$

Question 12.

What is the order of growth in time for **bubble-up**, where n is the length of the input argument?

Your answer: $\Theta(n)$

Question 13.

What is the order of growth in space for **bubble-sort**, where n is the length of the input argument?

Your answer: $\Theta(n)$

Question 14.

What is the order of growth in time for **bubble-sort**, where n is the length of the input argument?

Your answer: $\Theta(n^2)$

Now we want to consider a different method for sorting a list. For simplicity we will assume that all the elements of the list are distinct (i.e., the same value doesn't appear twice in the list). Here is the procedure:

```
(define (sort lst)
  (let ((best (max lst))) ; get biggest element
    (cons best ; add biggest element to front
          (sort ; sort rest of list, having removed best
                (filter (lambda (x) (not (= x best)))
                        lst))))))
```

Question 15. Assume you are given

```
(define (max2 a b)
  (if (> a b) a b))
```

Using this, write **max**. You may assume that it is always called with a non-empty list of numbers as argument and should return the largest element in the list.

```
(define (max lst)
  (if (null? (cdr lst))
      (car lst)
      (max2 (car lst) (max (cdr lst)))))
```

Question 16.

What is the order of growth in space for **max**, where n is the length of the input argument?

Your answer: $\Theta(n)$

Question 17.

What is the order of growth in time for `max`, where n is the length of the input argument?

Your answer: $\Theta(n)$

Question 18.

What is the order of growth in space for `sort`, where n is the length of the input argument?

Your answer: $\Theta(n)$

Question 19.

What is the order of growth in time for `sort`, where n is the length of the input argument?

Your answer: $\Theta(n^2)$

Question 20. Here is another way of sorting a list. The idea is to split the list in half, sort each half (recursively) and then merge the resulting sorted lists.

The following procedures are useful in accomplishing this

```
(define (split lst)
  (define (help one other rest)
    (if (null? rest)
        (list one other) ;; create a list of two lists
        (help (cons (car rest) other) one (cdr rest))))
    ;; add next element to one sublist and continue
  (help nil nil lst))

(define (merge l1 l2)
  (if (null? l1)
      l2 ; just add 2nd list
      (if (null? l2)
          l1 ; just add 1st list
          (if (> (car l1) (car l2))
              ; put biggest at front, merge rest
              (cons (car l1) (merge (cdr l1) l2))
              (cons (car l2) (merge l1 (cdr l2)))))))
```

The following code fragment will accomplish a merge sort for us.

```
(define (split-sort lst)
  (cond ((null? lst) YOUR-CODE-1)
        ((null? (cdr lst)) YOUR-CODE-2)
        (else (let ((splitup (split lst)))
                  YOUR-CODE-3))))
```

What code should you use for YOUR-CODE-1?

nil

What code should you use for YOUR-CODE-2?

lst

What code should you use for YOUR-CODE-3?

(merge (split-sort (car lst)) (split-sort (cadr lst)))

Part 4: (16 points)

Consider the following procedures:

```
(define (create-multiplier f)
  (lambda (x) (* x (f x))))

(define (echo f n)
  (if (= n 1)
      f
      (lambda (g) ((echo f (- n 1)) (f g)))))

(define identity (lambda (x) x))

(define more (lambda (x) (+ x 1)))
```

Question 21. What value is returned by the following expression

```
((create-multiplier identity) 7)
```

49

Question 22. What value is returned by the following expression

```
((create-multiplier more) 3)
```

15

Question 23. What value is returned by the following expression

```
((echo create-multiplier 3) identity) 2)
```

16

Question 24. What value is returned by the following expression

```
((echo create-multiplier 3) more) 2)
```

32