

MASSACHUSETTS INSTITUTE OF TECHNOLOGY  
 Department of Electrical Engineering and Computer Science  
 6.001—Structure and Interpretation of Computer Programs  
 Spring Semester, 2005

**Quiz I – Sample 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	17		
3	20		
4	12		
5	15		
6	16		
Total	100		

For your reference:

Section	Time	Location	Rec. Instructor	Tutors
10H	10:00	34-301	Horn	Dalley
10C	10:00	36-144	Cutler	Yu
10G	10:00	34-304	Koile	various
11G	11:00	26-210	Koile	various
11H	11:00	34-301	Horn	Gleyzer
11C	11:00	26-314	Collins	Danaher
12C	12:00	34-301	Collins	Pritchard
12B	12:00	26-314	Barzilay	Wilson
1B	1:00	34-303	Barzilay	various
1D	1:00	34-301	Durand	Vlasic
2C	2:00	26-322	Cutler	Zhou
2D	2:00	34-303	Durand	various

**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 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.**

>

**primitive procedure: number, number  $\mapsto$  boolean**

**Question 2.**

```
(define * /)
(define + -)
(* 12 (+ 6 2))
```

**3: number**

**Question 3.**

```
((lambda (a + b) (+ b a))
 2 - 4)
```

**2: number**

**Question 4.**

```
((lambda (a)
  (lambda (b)
    (+ (sqrt a) (sqrt b))))
  5)
```

**compound procedure: number  $\mapsto$  number**

**Question 5.**

```
(define arg 5)
(define local-arg 3)
(define (proc arg)
  (let ((local-arg 2))
    (list arg local-arg)))
(proc 1)
```

**(1 2): List;number;**

Consider the following simple database of personnel information. The entire database is represented as a **list** of entries. Each entry is made using the following constructor:

```
(define (make-entry person salary position)
  (list person salary position))
```

The **person** part of an entry is created using the constructor:

```
(define make-person list)
```

Note that values for names and positions are represented using strings, while salaries are represented using numbers. Here is an example database:

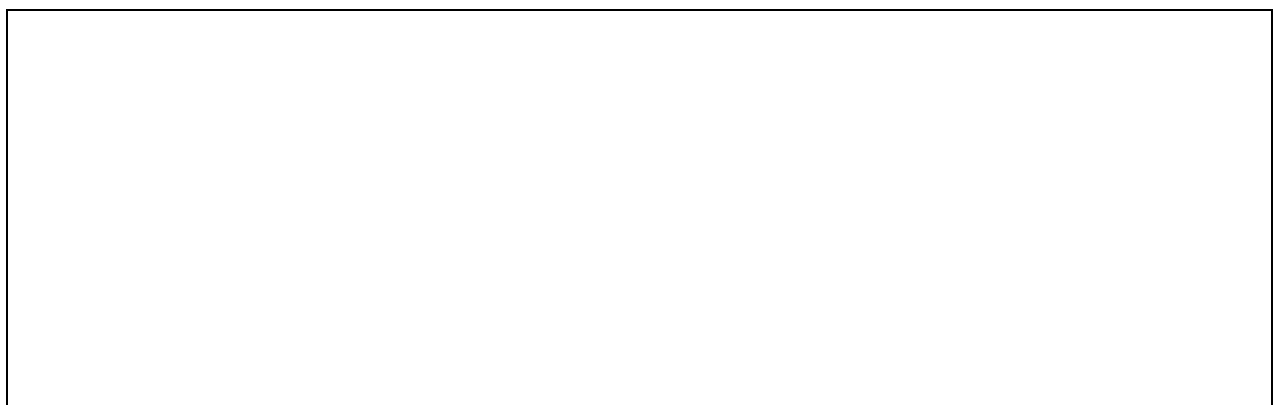
```
(define sampledata
  (list (make-entry (make-person "smith" "john" "henry") 30000 "president")
        (make-entry (make-person "jones" "anne" "marie" "heather") 60000 "hacker")
        (make-entry (make-person "smith" "fred") 55000 "hacker")
        (make-entry (make-person "doe" "jane" "elizabeth") 38000 "assistant")
        (make-entry (make-person "roe" "marie" "jane") 29000 "vice-president")))
```

Note that the "family" name is always the first element of the person abstraction, but there can be arbitrarily many "given" names.

## Part 2: (17 points)

**Question 6:** Draw a box-and-pointer diagram for the structure corresponding to **test**, where

```
(define test (make-entry (make-person "smith" "john" "henry") 30000 "president"))
```



**Question 7:** Complete the `entry` abstraction by providing selectors for `person`, for `salary` and for `position`. For example:

```
(person test)
;Value: ("smith" "john" "henry")
```

**solution:**

```
(define person car)
(define salary cadr)
(define position caddr)
```

**Question 8:** Complete the `person` abstraction by providing selectors for `family-name` and `given-names`, e.g.

```
(given-names (make-person "jones" "anne" "marie" "heather"))
;Value: ("anne" "marie" "heather")

(family-name (make-person "jones" "anne" "marie" "heather"))
;Value: "jones"
```

Thus the family name is always the first name in the entry, the given names are any names after this.

**solution:**

```
(define family-name car)
(define given-names cdr)
```

**Part 3: (20 points)**

Now suppose that we want to retrieve entries from the database that satisfy certain constraints. For example, we might want to get all the entries of people with the position of "hacker", or we might want to get the entries of people whose first name is "Jane", or all the entries of people with salaries between 25000 and 50000. Remember our procedure:

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

**Question 9:** We want a way of getting entries from the database with a particular first name (where by first name, we mean the first of the "given" names, not the family name). You may assume that every entry in the database has at least one given name. You may also find the procedure `string=?`, which compares two strings, to be useful.

Complete the following code so that, for example,

```
(filter (called-by "jane") sampledata)
;Value: (((("doe" "jane" "elizabeth") 38000 "assistant")))
```

```
(define (called-by name)
```

**solution:**

```
(lambda (x) (string=? name (car (given-names (names x)))))
```

**Question 10:** Suppose we want to find all the people who have salaries at least as large as a specified amount, and who hold a particular position.

```
(filter (salary-and-position 60000 "hacker") sampledata)
;Value: (((("jones" "anne" "marie" "heather") 60000 "hacker")))
```

Complete the following code fragment:

```
(define (salary-and-position minimum posn)
```

**solution:**

```
(lambda (entry)
  (and (>= (salary entry) minimum)
       (string=? (position entry) position)))
```

**Question 11:** Assume that the procedure `one-of` has the following behavior. It takes two arguments, an element and a list. It successively tests for equality of the element to entries in the list, using `string=?`, until it either reaches the end of the list (in which case it returns `false`) or until it finds an element of the list that matches (in which case it returns `true`). Using this, supply an expression for `INSERT1` in the expression below.

```
(define (has-name name)
  INSERT1)

(filter (has-name "marie") sampledata)
;Value: ((("jones" "anne" "marie" "heather") 60000 "hacker")
         ("roe" "marie" "jane") 29000 "vice-president"))
```

**solution:**

```
(lambda (x) (one-of name (given-names (names x))))
```

**Question 12:** Recall the procedure:

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

Provide an expression for `INSERT2` so that evaluating `(map INSERT2 sampledata)` would return a list of the number of names (both family and given names) for each entry, e.g.

```
(map INSERT2 sampledata)
;Value: (3 4 2 3 3)
```

You may assume that `length` is a procedure that returns the number of elements (or length) of a list.

**solution:**

```
(lambda (x) (length (names x)))
```

**Part 4 (12 points)** Given our little data base of personnel files, we might want to be able to sort the entries, for example by increasing salary. Here is a procedure for sorting:

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

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

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

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

```
(sort sampledata < salary =)
```

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`, `extractor` 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 13:** What is the order of growth in time of the procedure `find-best`?

**B**

**Question 14:** What is the order of growth in space of the procedure `find-best`?

**A**

**Question 15:** What is the order of growth in time of the procedure `remove`?

**B**

**Question 16:** What is the order of growth in space of the procedure `remove`?

**B**

**Question 17:** What is the order of growth in time of the procedure `sort`? Remember to include the effect of `find-best` and `remove`.

**D**

**Question 18:** What is the order of growth in space of the procedure `sort`? Remember to include the effect of `find-best` and `remove`.

**B**

**Part 5 (15 points)**

Here is a procedure for composing two procedures:

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

and here is a procedure for applying a given procedure some number of times

```
(define (repeated f n)
  (if (= n 1)
      f
      (CODE-ADDITION-HERE)))
```

You will consider some possible pieces of code to complete `repeated`. The expected behavior is:

```
(define (square x) (* x x))
;Value: "square --> #[compound-procedure 7 square]"
```

```
(repeated square 4)
;Value: #[compound-procedure 8]
```

```
((repeated square 3) 2)
;Value: 256
```

```
((repeated square 4) 2)
;Value: 65536
```

```
((repeated square 2) 3)
;Value: 81
```

For each of the following pieces of code, indicate whether it correctly completes the implementation. If it does not, indicate whether it results in an error, or what value it returns for the last example above:

**Question 19** `(lambda (x) ((compose f (repeated f (- n 1))) x))`

**Correct**

**Question 20** `(lambda (x) ((compose (repeated f (- n 1)) f) x))`

**correct**

**Question 21** `(lambda (x) (compose f (repeated f (- n 1))))`

**incorrect – returns procedure**

**Question 22** `((lambda (x) (compose f (repeated f (- n 1)))) x)`

**incorrect – error**

**Question 23** `(compose f (repeated f (- n 1)))`

**correct**

**Part 6 (16 points)**

We want to add a simple type check to our language, specifically the ability to state requirements on arguments to procedures, so that if an argument supplied to a procedure does not meet the specified conditions, we will exit with an error. For example, here is a procedure that computes the greatest common divisor of two numbers:

```
(define (gcd x y)
  (assert x number?)
  (assert y (list number? (lambda (a) (>= a 0))))
  (if (= y 0)
      x
      (gcd y (remainder x y))))
```

The idea behind `assert` is that it will apply one or more tests to the specified argument. If all the tests are true, it will simply return some value (e.g. the value `#T`) and continue. If not, it will exit with some error message. Thus, in the example above, the first `assert` expression would check that the parameter `x` has a value that is a number, while the second expression would check that the parameter `y` has a value that is both a number and is no less than 0.

**Question 24:**

Complete the following definition. `procedure?` will return true if the argument is a procedure. Remember that `error` is a Scheme procedure that will cease evaluation and return the user to top level with an error message. If you use `error` don't worry about details of the message returned, just use something you think is reasonable.

```
(define (assert test-val procs)
  (cond ((procedure? procs)
        INSERT3)
        ((null? procs)
         #T)
        (else INSERT4)))
```

What code should be used for `INSERT3`?

**solution**

```
(if (procs test-val)
    #T
    (error ...))
```

What code should be used for `INSERT4`?

**solution**

```
(if ((car procs) test-val)
    (assert test-val (cdr procs))
    (error ...))
```