MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 2002

Quiz I – Sample solutions

Below are example solutions for each of the questions. These are not the only possible answers, but they are the most common ones.

Part 1: (16 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`. If the evaluation returns a user-created procedure, write `compound procedure`.

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

Question 1.

```
(+ (* 2 3) (- 5 2))
```

9, number

Question 2.

```
>
```

primitive procedure; number, number $\mapsto$ boolean

Question 3.

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

3, number

Question 4.

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

8, number

Question 5.

```
((lambda (a)
   (lambda (b)
     (exp a b)))
 5)
```

compound; number $\mapsto$ number

Question 6.

```
(define x 1)
(define y 5)
(define (proc x)
  (let ((y 2))
    (* x y)))
(proc 3)
```

6, number

Question 7.

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

((compose (lambda (x) (+ x 2))
          (lambda (x) (* x 2)))
  3)
```

8, number

Question 8.

```
(define (echo f n)
  (if (= n 0)
      (lambda (x) x)
      (lambda (x)
        (f ((echo f (- n 1)) x)))))

((echo (lambda (x) (* x x)) 3) 2)
```

256, number

Suppose we are given a small database of personnel information. This database is represented as a **list** of entries, each entry is made from the following constructor:

```
(define (make-entry names age)
  (list names age))
```

The `names` part of an entry is created using the constructor:

```
(define make-names list)
```

Here is an example database:

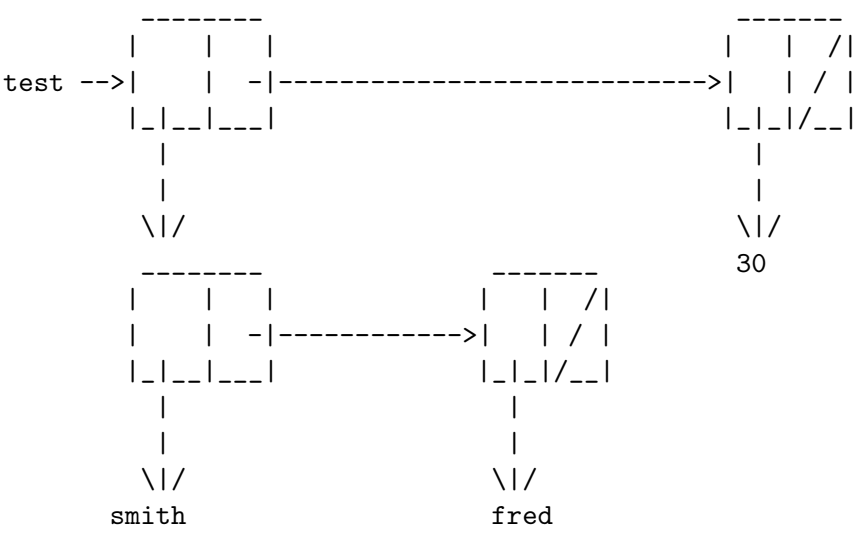
```
(define ourdata
  (list (make-entry (make-names 'smith 'john 'henry) 30)
        (make-entry (make-names 'jones 'anne 'marie 'heather) 29)
        (make-entry (make-names 'smith 'fred) 50)
        (make-entry (make-names 'doe 'jane 'elizabeth) 38)
        (make-entry (make-names 'roe 'marie 'jane) 15)
))
```

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

Part 2: (14 points)

Question 9: Draw a box-and-pointer diagram for the structure corresponding to `test`, where

```
(define test (make-entry (make-names 'smith 'fred) 30))
```



Question 10: Complete the `entry` abstraction by providing selectors for `names` and `age`

```
(define names car)
(define age cadr)
```

Question 11: Complete the `names` abstraction by providing selectors for `family-name` and `other-names`, e.g.

```
(other-names (make-names 'jones 'anne 'marie 'heather))
;Value: (anne marie heather)

(family-name (make-names 'jones 'anne 'marie 'heather))
;Value: jones

(define family-name car)
(define other-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 ages between 25 and 30, or we might want to get the entries of people whose first name is Jane. Remember our procedure:

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

Question 12: 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 "other" or "given" names, not the family name). You may assume that every entry in the database has at least one given name.

Here is the completed code so that, for example,

```
(filter (called-by 'jane) ourdata)
;Value: (((doe jane elizabeth) 38))

(define (called-by name)
    (LAMBDA (EQ? NAME (CAR (OTHER-NAMES (NAMES X)))))
```

Question 13: Suppose we want to find all the people whose ages are within a specified range.

```
(filter (ages-between 25 30) ourdata)
;Value: (((smith john henry) 30)
         ((jones anne marie heather) 29))
```

Here is the completed code.

```
(define (ages-between low high)
  (LAMBDA (X)
    (LET ((VAL (AGE X)))
      (AND (>= VAL LOW) (<= VAL HIGH))))
  )
```

Question 14: 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 `eq?`, until it either reaches the end of the list (in which case it returns the empty list) or until it finds an element of the list that matches (in which case it returns that element).

Here is the completed code.

```
(define (has-name name)
  (LAMBDA (X) (ONE-OF NAME (OTHER-NAMES (NAMES X)))))

(filter (has-name 'marie) ourdata)
;Value: (((jones anne marie heather) 29)
         ((roe marie jane) 15))
```

Question 15: Recall the procedure:

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

Provide an expression for `INSERT4` so that evaluating `(map INSERT4 ourdata)` would return a list of the number of names for each entry, e.g.

```
(map INSERT4 ourdata)
;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.

Here is the expression:

```
 (LAMBDA (X) (LENGTH (NAMES X)))
```

Question 16: Suppose we want to find entries that satisfy several conditions, for example:

```
(define (age&name name low high)
  (double-filter (ages-between low high)
                 (has-name name)))

(filter (age&name 'marie 25 30) ourdata)
;Value: (((jones anne marie heather) 29))
```

Here is the completed code

```
(define (double-filter a b)
  (LAMBDA (X) (AND (A X) (B X))))
end{verbatim}}



\medskip

{\bf Part 4 (16 points)}
Given our little data base of personnel files, we might want to be able to
sort the entries, for example by decreasing age.  Here is a procedure
for sorting:

{\small
\begin{verbatim}
(define (sort data extractor compare)
  (define (find-best&rest best rest seen)
    (if (null? rest)
        (cons best seen)
        (if (compare (extractor (car rest))
                     (extractor best))
            (find-best&rest (car rest)
                            (cdr rest)
                            (cons best seen))
            (find-best&rest best
                            (cdr rest)
                            (cons (car rest) seen)))))
  (let ((trial (find-best&rest (car data) (cdr data) '())))
    (if (null? (cdr trial))
        trial
       (cons (car trial)
             (find-best&rest (car (cdr trial))
                             (cdr (cdr trial))
                             '())))))
```

For example, to sort our data by decreasing age, we would evaluate:

```
(sort ourdata age >)
```

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`.

For each of the following, choose from these options

A: $O(1)$
B: $O(n)$
C: $O(2^n)$
D: $O(n^2)$
E: $O(\log n)$
F: something else

Question 17: What is the order of growth in time of the procedure `find-best&rest`?

B

Question 18: What is the order of growth in space of the procedure `find-best&rest`?

A

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

D

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

B

Part 5 (14 points)

Question 21: We assumed the existence of the procedure `length`, which returned the number of elements in a list. Write an **iterative** version of `length`.

```
(DEFINE (LENGTH LST)
  (DEFINE (AUX L COUNT)
    (IF (NULL? L)
        COUNT
        (AUX (CDR L) (+ 1 COUNT))))
  (AUX LST 0))
```

Question 22: Suppose we are given a tree of integers, represented as a list of lists. Assume that `null?` tests for an empty tree, and that `leaf?` tests for a leaf of a tree. We want to find the maximum depth of a tree, that is the longest path from the root of a tree to a leaf. For example:

```
(maxdepth 3)
;Value: 0

(maxdepth '(1 2 3))
;Value: 1

(maxdepth '(1 (2 (3)) (((4)))))
;Value: 4
```

Write the procedure `maxdepth`.

```
(DEFINE (MAXDEPTH T)
  (COND ((NULL? T) 0)
        ((LEAF? T) 0)
        (ELSE (MAX (+ 1 (MAXDEPTH (CAR T))) (MAXDEPTH (CDR T))))))
```

Part 6: (20 points)

The procedure `reverse` can be used to reverse the elements of a list. For example, suppose `test` has the structure:

```
(1 (2 (3 4) 5) (6 7) 8)
```

then we have

```
(reverse (list 1 2 3 4))
;Value: (4 3 2 1)

(reverse test)
;Value: (8 (6 7) (2 (3 4) 5) 1)
```

As you have noticed, `reverse` only reverses the top level structure of a list, so when given a tree structure such as `test`, it does not recursively reverse the elements of the list. `Deep-reverse` is intended to accomplish a complete reversal of the tree, for example

```
(deep-reverse test)
;Value: (8 (7 6) (5 (4 3) 2) 1)
```

Below are some possible outcomes of applying different implementations of `deep-reverse` to test:

```
A: (8 (7 6) (5 (4 3) 2) 1)

B: (8 (6 7) (2 (3 4) 5) 1)

C: (8 (7 6) (5 (3 4) 2) 1)

D: (8 7 6 5 4 3 2 1)

E: infinite loop

F: some other error

G: some other value
```

For each of the following possible implementations of `deep-reverse` indicate the outcome that matches.

Question 23.

```
(define (deep-reverse tree)
  (if (or (null? tree) (leaf? tree))
      tree
      (map deep-reverse (reverse tree))))
```

A

Question 24.

```
(define (deep-reverse tree)
  (if (or (null? tree) (leaf? tree))
      tree
      (map reverse (deep-reverse tree))))
```

E

Question 25.

```
(define (deep-reverse tree)
  (if (or (null? tree) (leaf? tree))
      tree
      (reverse (map deep-reverse tree))))
```

A

Question 26.

```
(define (deep-reverse tree)
  (if (or (null? tree) (leaf? tree))
      tree
      (cons (reverse (cdr tree))
            (reverse (car tree)))))
```

G

Question 27.

```
(define (deep-reverse tree)
  (if (or (null? tree) (leaf? tree))
      tree
      (append (deep-reverse (cdr tree))
              (list (reverse (car tree))))))
```

C