

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
 Department of Electrical Engineering and Computer Science
 6.001—Structure and Interpretation of Computer Programs
 Spring Semester, 2002

Quiz I – Solutions

Listed below are example answers for each of the questions. There are alternative answers in some cases, these are simply intended to demonstrate the most common answer.

Part 1: (16 points)

For each of the following expressions or sequences of expressions, state what value is returned as the result of evaluating the final expression in each set. For each question, state what value would be returned if that expression or sequence of expressions were the only expressions evaluated in a newly initialized Scheme system.

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 all non-error values, also write the type of the returned value, using the notation from lecture.

(* 2 4)

8, number

(2 - 4)

Error – not a procedure

+

primitive; number, number \mapsto number

(lambda (a b) (if a b (- b)))

compound; (boolean, number \mapsto number)

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

compound, number \mapsto number

```
((lambda (x) (if x + *))
 #t)
 2 4)
```

6, number

```
(define sq (lambda (x) (* x x)))
(define (comp f g)
  (lambda (x) (f (g x))))
((comp sq (lambda (x) (* x 2))) 3)
```

36, number

```
(define a 3)
(define b 2)
(define (foo b)
  (let ((a 4))
    (* a b)))
(foo 5)
```

20, number**Part 2: (20 points)**

In many mathematical and physical computations, it is common to use “natural logarithms” and their inverses, “natural exponents”, which are often written as e^x . A common way to find a good approximation to e^x is to use the series

$$e^x \approx \sum_{k=0}^n \frac{x^k}{k!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^n}{n!}.$$

Clearly, as we let n get larger, the approximation will get better.

Assume that the Scheme procedure `fact` computes factorial, and that the Scheme procedure `expt` computes exponentials, that is, `(fact n)` computes $n!$, and `(expt x k)` computes x^k .

Here is the completed code (for questions 9 and 10).

```
(define (exp x n)
  (if (= n 0)
      1
      (+ (/ (EXPT X N) (FACT N))
         (EXP X (- N 1)))))
```

Other common mathematical expressions can be approximated by similar summations:

$$\cos x \approx \sum_{k=0}^n (-1)^k \frac{x^{2k}}{(2k)!} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots + (-1)^n \frac{x^{2n}}{(2n)!}$$

$$\sin x \approx \sum_{k=0}^n (-1)^k \frac{x^{2k+1}}{(2k+1)!} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots + (-1)^n \frac{x^{2n+1}}{(2n+1)!}$$

One way to do this is to note that each expression is a sum of terms that depend only on k and x ,

$$\cos x \approx \sum_{k=0}^n t_k(x)$$

where $t_k(x) = 0$ if k is odd, and otherwise

$$t_k(x) = (-1)^{\frac{k}{2}} \frac{x^k}{k!}$$

A similar version holds for \sin , with a different set of terms, $t_k(x)$. Thus we could write a more general procedure for computing sums of this form:

```
(define (trig n term)
  (define (summit sum x k)
    (if (= k n)
        sum
        (summit (+ sum (term x (+ k 1)))
                 x
                 (+ k 1))))
  (lambda (x)
    (summit (term x 0) x 0)))
```

Question 11: Thus we can implement a good approximation to cosine by completing the following definition. You may assume that the predicate `odd?` exists, and returns true if its argument is an odd integer:

```
(define my-cos
  (trig 10
    (lambda (x k)
      (if (odd? k)
          0
          (* (expt -1 (/ k 2)) (/ (expt x k) (fact k)))))))
```

We could use a similar method for sine, but we can also take advantage of the fact that $\frac{d \cos(x)}{dx} = -\sin(x)$. Thus

```
(define (sin x)
  (* -1 ((derivative cos) x)))
```

Recall that we can approximate a derivative by:

$$\frac{df(x)}{dx} \approx \frac{f(x + dx) - f(x)}{dx}$$

Question 12: Assume that `(define dx 0.001)`.

Write a definition for `derivative`.

```
(define (derivative f)
  (lambda (x) (/ (- (f (+ x dx)) (f x)) dx)))
```

Part 3: (20 points)

Suppose you are given a set of integers, represented as a list. Here is a procedure, `member?`, of type: `number, List<number> ↦ boolean`, which returns `true` if the first argument is in the set, and `false` otherwise.

```
(define (member? x s)
  (if (null? s)
      #f
      (if (= x (car s))
          #t
          (member? x (cdr s)))))
```

Now suppose you are given two different sets, each of which contains unique members (i.e. no number is repeated within a single set).

Question 13: Write a procedure, `intersection`, which takes as arguments two such sets, and returns a new set, which contains only those elements that appear in both input sets. For example,

```
(define S (list 1 4 3 2 5))
```

```
(define T (list 9 5 1 7 3))
```

```
(intersection S T)
;Value: (1 3 5)
```

Your procedure should use `member?`, and have a recursive behavior. Remember that `car` and `cdr` can be used to extract appropriate components of a list.

```
(define (intersection s t)
  (if (null? s)
      nil
      (if (member? (car s) t)
          (cons (car s) (intersection (cdr s) t))
          (intersection (cdr s) t))))
```

Now suppose that we are going to deal with sets that contain elements other than just numbers. We need to generalize our test for membership to allow for comparisons of such elements. Below is a procedure for creating methods for comparing elements:

```
(define make-member
  (lambda (comp?)
    (define generic-member (lambda (x s)
      (if (null? s)
          #f
          (if (comp? x (car s))
              #t
              (generic-member x (cdr s))))))
    generic-member))
```

Now suppose you are given the following two sets:

```
(define set1
  (list (list #t 1)
        (list #f 2)
        (list #f 3)
        (list #t 4)))

(define set2
  (list (list #t 1)
        (list #t 3)
        (list #f 5)
        (list #f 7)))
```

For each of the following cases, provide the necessary expression to use in place of COMPLETE so that evaluating the intersection expression will give rise to the result shown. You may find it convenient to use the Scheme expressions `and` or `or`.

Question 14: Two elements are the same if the second part of each element is numerically equal.

```
(define member? (make-member?
  (LAMBDA (X Y)
    (= (CADR X) (CADR Y)))))
```

```
(intersection set1 set2)
;Value: ((#t 1) (#f 3))
```

Question 15: Two elements are the same if the first part of each element is equal as a boolean, and the second part is numerically equal.

```
(define member? (make-member?
  (LAMBDA (X Y)
    (AND (CAR X)
```

```
(CAR Y)
(= (CADR X) (CADR Y))))))
```

```
(intersection set1 set2)
;Value: ((#t 1))
```

Part 4: (26 points)

A **string** in Scheme is a sequence of characters, delineated by double quotes, as in ‘‘foo’’ or ‘‘bar’’. For purposes of this question, you may simply treat a string as an abstract data object. Suppose you are given a set of strings, represented as a list, such as

```
(define S (list ‘‘frob’’ ‘‘foo’’ ‘‘bar’’ ‘‘baz’’ ‘‘fiddlesticks’’))
```

We would like to convert this to a list, where each element is itself a list of the string, and the length of the string, e.g.

```
( (‘‘frob’’ 4) (‘‘foo’’ 3) (‘‘bar’’ 3) (‘‘baz’’ 3) (‘‘fiddlesticks’’ 12))
```

Question 16: Using `map`, `filter` and/or `accumulate`, defined as

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

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

```
(define (accumulate op init lst)
  (if (null? lst)
      init
      (op (car lst)
          (accumulate op init (cdr lst)))))
```

write a procedure, `lengthen` that would take as argument a list of strings, and returns a list as described. You may use `string-length` to compute the length of a string.

```
(DEFINE (LENGTHEN LST)
  (MAP (LAMBDA (ELT) (LIST ELT (STRING-LENGTH ELT))) LST)
```

Question 17: Now suppose that you have a list of the new form, for example,

```
(define S-long (lengthen S))

S-long
;Value: (('frob' 4) ('foo' 3) ('bar' 3) ('baz' 3) ('fiddlesticks' 12))
```

We want to recover a list of just the lengths of the strings, from this list. Again, using `map`, `filter` and/or `accumulate`, write a procedure called `just-lengths` that takes as input a list of the form shown for `T`, and returns a list of the lengths of each element

```
(just-lengths S-long)

;Value: (4 3 3 3 12)
```

```
(DEFINE (JUST-LENGTHS LST)
  (MAP CADR LST))
```

Question 18: Suppose we use our previous procedure to get a list of the lengths of the elements. Now we want to get the total length of all the strings in the original list. Write a procedure, `total-length`, using `map`, `filter` and/or `accumulate`, that will return the total length of all the strings in the original list.

```
(total-length (just-lengths (lengthen s)))

;Value: 25
```

```
(DEFINE (TOTAL-LENGTH L)
  (ACCUMULATE + 0 L))
```

Question 19: Finally, suppose we have a list of the form shown for `S-long` above, i.e. a list each of whose elements is a list of a string and a number. We want to process this list to keep the first element of a particular length, that is, the first element with length 3, with length 4, and so on. In the case of `S-long`, this would yield

```
(unique S-long)
;Value: (('foo' 3) ('frob' 4) ('fiddlesticks' 12))
```

Here is an outline of the procedure:

```
(define (unique set)
  (if (null? set)
      nil
      (cons (car set)
            (UNIQUE (FILTER (LAMBDA (X) (NOT (= (CADR X) (CADR (CAR SET))))))
                  (CDR SET))))))
```

Part 5: (18 points)

For each of the following procedures, what is 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), measured as a function of the size of n . You may assume that $+$, $*$, $-$, $=$, $<=$ are primitive operations.

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

Question 20:

```
(define (f x n)
  (define (aux sum k)
    (if (= k n)
        0
        (+ (* x k)
            (aux sum (+ k 1)))))
  (aux 0 0))
```

Space: B, Time: B

Question 21:

```
(define (f x n)
  (define (aux sum k)
    (if (= k n)
        sum
        (aux (+ sum (* x k)) (+ k 1))))
  (aux 0 0))
```

Space: A, Time: B

Question 22:

```
(define (f x n)
  (if (<= n 1)
      1
      (+ (* x n)
          (f x (- n 1))
          (f x (- n 2)))))
```

Space: B, Time: C