

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 2003

Quiz I – Example Solutions

Closed Book – one sheet of notes

Below are example solutions for Quiz 1. In many cases, other answers were acceptable. These are simply intended to show possible solutions.

Part 1: (17 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 its type.

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

Question 1.

```
(3 * 5)
```

error, not a procedure

Question 2.

```
(define * /)
(define / *)
(/ 12 (* 6 3))
```

6, number

Question 3.

```
(lambda (a)
  (lambda (b)
    (/ (+ a b) (- a b))))
```

compound, number -> (number -> number)

Question 4.

```
((lambda (n)
  (lambda (x) (expt x n)))
  5)
```

compound, number->number

Question 5.

```
((lambda (arg) ARG) (lambda () 5))
```

compound, -> number

Question 6.

```
((lambda (arg) (ARG)) (lambda () 5))
```

5, number

Question 7.

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

((merge (lambda (x) (+ x 5))
        (lambda (x) (* x 5)))
 2)
```

35, number

Part 2: (30 points)

We have seen an example of a tree as a data structure in lecture. A variation on this data structure is a binary tree, given by the following constructors:

```
(define (make-node elt left right)
  (list elt (make-branches left right)))

(define (make-branches left right)
  (list left right))
```

Question 8: Draw a box-and-pointer diagram for

```
(define test (make-node 3 1 5))
```

Question 9: Complete the data abstraction for a node, by defining get-elt, get-branches, get-left, get-right, so that, for example:

```
(get-elt (make-node 1 'foo 'bar))
;Value: 1

(get-left (get-branches (make-node 1 'foo 'bar)))
;Value: foo

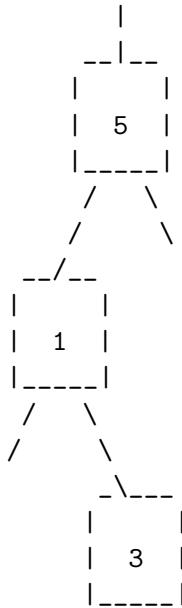
(get-right (get-branches (make-node 1 'foo 'bar)))
;Value: bar
```

Of course, in practice, our branches will be other trees, not symbols. The examples above are simply intended to demonstrate the contract of the data abstraction.

```
(define get-elt car)
(define get-branches cadr)
(define get-left car)
(define get-right cadr)
```

Thus, a tree starts with a root node, which would contain some value (called `elt`) and pointers to the left and right branches of that node. Each branch also contains a value, and left and right branches, which are possibly empty. The idea is that we can impose an order on our tree, so that all the elements lying in the left branch of a node are “less than” the element at that node, which in turn is “less than” all the elements lying in the right branch of the node.

Here is an example of a tree, in which first 5 was inserted, then 1 and then 3. Nodes are simply represented as boxes (not to be confused with box-and-pointer diagrams in this example).



Now we would like to create code to insert elements into a tree, such as:

```
(define empty-tree nil)
(define tree (insert 5 empty-tree))

(define tree (insert 10 tree))
```

and so on. Under this approach, each new element is inserted with a new call to `insert` and we have to redefine the name for our tree to the result of that insertion. Note that if we try to insert a value already present in the tree, the procedure should simply return the tree, without any additional insertion. Here is the completed code for questions 10-12:

```
(define (insert elt tree)
  (cond ((null? tree) (MAKE-NODE ELT EMPTY-TREE EMPTY-TREE))
        ((= elt (get-elt tree)) TREE)
        ((< elt (get-elt tree))
         (MAKE-NODE (GET-ELT TREE)
                    (INSERT ELT (GET-LEFT (GET-BRANCHES TREE)))
                    (GET-RIGHT (GET-BRANCHES TREE))))
        (else
         (MAKE-NODE (GET-ELT TREE)
                    (GET-LEFT (GET-BRANCHES TREE))
                    (INSERT ELT (GET-RIGHT (GET-BRANCHES TREE)))))))
```

If we have a set of things to insert, it would be nice to have a single procedure:

```
(define tree (insert-multi '(1 5 9 3 16) nil))
```

Here is the completed code for Question 13

```
(define (insert-multi lst tree)
  (if (null? lst)
      tree
      (INSERT-MULTI (CDR LST) (INSERT (CAR LST) TREE))))
```

Question 14: Assuming that elements are inserted into the tree in random fashion, so that the number of elements in the left and right branch of each node is roughly the same, what is the order of growth in space and time for insert? Measure this in terms of n , the number of nodes in the tree.

space – $O(\log n)$

time – $O(\log n)$

Question 15: Assuming that elements are inserted into the tree in strictly increasing order, what is the order of growth in space and time for insert? Measure this in terms of n , the number of nodes in the tree.

space – $O(n)$

time – $O(n)$

Part 3: (22 points)

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

```
(define test '(1 (2 (3 (4) 5) 6) 7))
```

then we have

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

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

Question 16. Here is a recursive version of reverse:

```
(define (reverse lst)
  (if (null? lst)
      nil
      (append (reverse (cdr lst))
              (list (car lst)))))
```

Write the procedure `ireverse` so that it also reverses a list, but gives rise to an iterative process.

```
(define (ireverse lst)
  (define (aux sofar todo)
    (if (null? todo)
        sofar
        (aux (cons (car todo) sofar) (cdr todo))))
  (aux nil lst))
```

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

Assume that `leaf?` returns true if its argument is not a list or pair, and that

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

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

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

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

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

D: (7 6 5 4 3 2 1)

E: infinite loop

F: some other error

G: some other value

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

Question 17.

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

A

Question 18.

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

F

Question 19.

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

B

Question 20.

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

E

Question 21.

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

B**Part 4: (10 points)**

The procedure `repeated` takes two arguments, a procedure `p` of type `number` \mapsto `number`, and a number `n`. `Repeated` returns a procedure of type `number` \mapsto `number` that will apply the procedure `P` `n` times in succession. Here is a definition:

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

Question 22: Using `repeated`, implement multiplication, using only the operations of addition, by supplying an expression for Q22-ANSWER.

```
(define (mul a b)
  ((REPEATED (LAMBDA (X) (+ A X)) B) 0))
```

Question 23: Using `repeated`, implement exponentiation, using only the operations of multiplication, by supplying an expression for Q23-ANSWER.

```
(define (expt a b)
  ((REPEATED (LAMBDA (Y) (MUL Y C)) N) 1))
```

Part 5: (21 points)

We can represent DNA with a list of symbols (A, C, G, or T) representing the bases from which DNA is assembled. Suppose you are given a list of these symbols, and you want to find the longest stretch of consecutive symbols (e.g. the longest stretch of A's in a row). Here is the completed code for a procedure that would return a list of the lengths of consecutive stretches of a symbol (answering Questions 24 and 25):

```
(define (find-consecutive-sequences input)
  (define (counter runsum match todo)
    (if (null? todo)
        (list (list match runsum))
        (if (eq? match (car todo))
            (COUNTER (+ RUNSUM 1) MATCH (CDR TODO))
            (CONS (LIST MATCH RUNSUM) (COUNTER 0 (CAR TODO) TODO))))))
  (counter 0 (car input) input))

(define test (find-consecutive-sequences '(g a a g c t a a t a a a a a t a)))
;Value: ((g 1) (a 2) (g 1) (c 1) (t 1) (a 2) (t 1) (a 6) (t 1) (a 1))
```

The idea is that this procedure keeps counting (using `runsum`) the number of consecutive instances of a symbol, until it reaches a new symbol. In that case, it adds a list of the symbol and number of instances to its output, and starts a new count with the next symbol.

Here are some useful procedures for manipulating lists

```
(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)))))

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

Using these, provide answers for the following questions.

Question 26: Suppose that `test` is the output of an application of `find-consecutive-sequences` such as shown above). Write an expression using `map`, `filter` and/or `accumulate` that will return a list of only those sequences involving the symbol `a`, e.g.

```
((a 2) (a 2) (a 6) (a 1))
```

```
(filter (lambda (x) (eq? (car x) 'a)) test)
```

Question 27: Suppose we define `runs-of-a` to be the result of evaluating the expression in Question 26, i.e. `runs-of-a` has the value `((a 2) (a 2) (a 6) (a 1))` for the example shown. Write an expression using `map`, `filter` and/or `accumulate` that finds the largest consecutive sequence of `a`'s. You may assume that `max2` is a procedure that returns the larger of its two arguments.

```
(accum max2 0 (map cadr runs-of-a))
```