

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
 Department of Electrical Engineering and Computer Science
 6.001—Structure and Interpretation of Computer Programs
 Fall Semester, 2006

Quiz II – Example Solutions

Part 1. (20 points)

Question 1: We are going to create a circlist out of a traditional list, by mutating the list.

Here is a template for what we need:

```
(define (last-cons lst)
  (cond ((null? lst) (error "not a list"))
        ((null? (cdr lst)) lst)
        (else (last-cons (cdr lst)))))

(define (create-circlist lst)
  (cond ((null? lst) (error "not a list"))
        (else ANSWER1)))
```

What expression(s) should be used for ANSWER1?

```
(set-cdr! (last-cons lst) lst)
lst
```

Question 2: Suppose we want to find the “length” of a circlist, that is, how many elements are in the circular list. We can do this by walking around the circlist, counting as we go, until we reach our initial starting point. Write the procedure `length-circlist`:

```
(define (length-circlist circlist)
  (define (countit where end n)
    (if (eq? where end)
        n
        (countit (cdr where) end (+ 1 n))))
  (countit (cdr circlist) circlist 1))
```

Question 3: To examine an element in the circlist or to change our location in the circlist, we have the following:

```
(define head car)
(define go-right cdr)
```

Note that `go-right` takes a circlist and returns a circlist with its head one element to the right of the original circlist.

Write a procedure called `go-left`, which returns a circlist with its head one element to the left of the original circlist.

Briefly describe your `go-left` algorithm in English.

start at current point, keep moving along cdrs until reach point where next cdr is start point, then return

Provide your definition.

```
(define (go-left circlist)
  (define (clickit where end)
    (if (eq? (cdr where) end)
        where
        (clickit (cdr where) end)))
  (clickit circlist circlist))
```

Part 2. (25 points)

```
(define (create-node value)
  (let ((right #f)
        (left #f))
    (lambda (msg)
      (cond ((eq? msg 'value) value)
            ((eq? msg 'right) right)
            ((eq? msg 'left) left)
            ((eq? msg 'set-right!)
             (lambda (new)
               (set! right new)))
            ((eq? msg 'set-left!)
             (lambda (new)
               (set! left new)))
            (else (error "don't know message" msg))))))
```

We now want to take a traditional list as input, and create a double-linked list, composed of these message-passing nodes.

```
(define (create-double-list lst)
  (let ((set-of-nodes (map create-node lst)))
    (connect set-of-nodes 'right)
    (connect set-of-nodes 'left)
    (car set-of-nodes)))
```

To do this, we need to write the procedure `connect`.

```
(define (connect set dir)
  (cond ((null? (cdr set)) 'done)
        ((eq? dir 'right)
         ANSWER4)
        ((eq? dir 'left)
         ANSWER5)
        (else (error "unknown direction"))))
```

Question 4: What expression(s) should be used for ANSWER4?

Describe your answer **briefly** in English

send message to first element to set right pointer to next element, then continue to connect down cdr of list

Provide your code

```
((car set) 'set-right!) (cadr set))
(connect (cdr set) dir)
```

Question 5: What expression(s) should be used for ANSWER5?

Describe your answer **briefly** in English

same idea as above

Provide your code

```
((cadr set) 'set-left!) (car set)
(connect (cdr set) dir)
```

Question 6: Suppose we have evaluated the following:

Write definitions for `dcar`, `dcdr`, `dcur` and `dcadr`.

```
(define dcar (lambda (dlist) (dlist 'value)))
(define dcdr (lambda (dlist) (dlist 'right)))
(define dcur (lambda (dlist) (dlist 'left)))
(define dcadr (lambda (dlist) ((dlist 'right) 'value)))
```

Question 7:

Given the ability to traverse a list in either direction, write a `mapdouble` function that works like `map`, but on double-linked lists, and takes three arguments: (`mapdouble fn lst dir`), where the last argument indicates the direction in which to traverse `lst`, applying `fn` to each element of `lst` and producing a double-linked list of the results. Note that if the initial value for `lst` points to an element midway along a double list, the result of `mapdouble` should include only instances of elements from that point to the end of the list in the specified direction, and should point to the corresponding list element in the new list. For example, if we do

```
(define (get-last lst dir)
  (if (not (lst dir))
      lst
      (get-last (lst dir) dir)))

(define (get-values lst dir)
  (if (not lst)
      '()
      (cons (dcar lst)
            (get-values (lst dir) dir))))
```

Write the procedure `mapdouble`

```
(define (mapdouble fun lst dir)
  (if (eq? dir 'right)
      (create-double-list (map fun (get-values lst dir)))
      (get-last
       (create-double-list (reverse (map fun (get-values lst dir))))
       'right)))
```

Part 3. (16 points)

Consider the following sequence of expressions:

```
(define (monitor proc)
  (let ((args '())
        (calls 0))
    (lambda (n)
      (set! args (cons n args))
      (set! calls (+ 1 calls))
      (cond ((number? n) (proc n))
            ((eq? n 'calls) calls)
            ((eq? n 'args) args))))))

(define m-sq (monitor (lambda (x) (* x x))))

(m-sq 4)

(m-sq 6)
```

Question 8: For each of the following frames, indicate the lowest frame of the enclosing environment, choosing one of **GE**, **E1**, **E2**, **E3**, **E4** or **none** or **not shown**.

Frame:	Enclosing Environment
GE	none
E1	E2
E2	E4
E3	E2
E4	GE

Question 9: For each of the following procedure objects, indicate the lowest frame of the enclosing environment, choosing one of **GE**, **E1**, **E2**, **E3**, **E4** or **none** or **not shown**.

Procedure:	Enclosing Environment
P1	GE
P2	E2
P3	GE
P4	E4

Question 10: For each of the following variable names, indicate the value to which it is bound in the specified environment at the end of the evaluation of the expressions. Indicate the value by choosing one of **GE**, **E1**, **E2**, **E3**, **E4**, **E5** or **P1**, **P2**, **P3** or a symbol, a number, a list of numbers or a boolean value.

Variable:	Environment	Value
monitor	GE	P3
m-sq	GE	P2
proc	E4	P1
args	E2	(6 4)
calls	E2	2

Part 4: (15 points)

We have seen in lecture that searching a tree structure is an important task in many applications. For this part of the quiz, we are going to search in a tree for the first node that satisfies a predicate `reach-goal?`. All we need to know about the tree abstraction, in addition to `reach-goal?`, is that it includes a selector `children`, which when applied to a node in the tree returns a **list** of that node's children.

Our queue abstraction has the following properties:

- the element at the head of the queue is selected using `front-queue`
- a queue containing all elements, except the head, is selected using `rest-queue`
- the procedure `queue-to-list` converts the queue into a list of its elements, preserving order
- `make-queue` takes as argument a list of elements and returns a queue with the element's order preserved
- `empty-queue?` tests if the argument is an empty queue

Here is an example tree search procedure:

```
(define (search start reach-goal? combine)
  (define (search1 queue)
    (if (empty-queue? queue)
        #f
        (let ((current (front-queue queue)))
          (if (reach-goal? current)
              current
              (search1 (make-queue
                        (combine
                         (children current)
                         (queue-to-list (rest-queue queue))))))))))
  (search1 (make-queue (list start))))
```

This search procedure uses a queue to keep track of nodes yet to be searched. It applies a procedure, `reach-goal?`, to determine when it has found the node for which it is looking, i.e. the goal node. If it has not found the goal node, it creates a new queue using the procedure, given the argument `combine`, to combine children of the current node with nodes still in the queue to be checked. Note that `combine` should take as input two lists, and return a list with its elements in appropriate order for constructing a queue.

For purposes of this part, you may assume that `(sort L)` sorts a list of nodes based on an ordering criterion relevant to the search.

Question 11: For a depth first search, what expression should be supplied for `combine` in the procedure `search`?

`append`

Question 12: For a breadth first search, what expression should be supplied for `combine` in the procedure `search`?

`(lambda (x y) (append y x))`

Question 13: For a best first search, what expression should be supplied for `combine` in the procedure `search`?

`(lambda (x y) (sort (append x y)))`

Part 5: (24 points)

Here is a definition of a higher order procedure for performing computations on trees. You may assume the following abstraction for a tree:

- `leaf?` is a predicate that returns true for any object that is not a tree
- `first-branch` returns the first branch of a tree
- `rest-branches` returns a tree with the first branch removed
- `empty-tree?` is a predicate that returns true if the argument is an empty tree
- `empty-tree` is a primitive data structure representing an empty tree
- `add-branch` takes as arguments two trees, and returns a tree with the first argument as the first branch of the new tree, followed by all the branches of the second argument
- `reverse` takes as argument a tree, and returns a new tree with the branches in reverse order

```
(define (tree-message operation tree leaf-operation base)
  (cond ((null? tree) base)
        ((leaf? tree) (leaf-operation tree))
        (else
         (operation (tree-message operation (car tree) leaf-operation base)
                    (tree-message operation (cdr tree) leaf-operation base))))))
```

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

Below are some possible outcomes for operations on `test-tree`:

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

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

C: 28

D: (1 2 3 4 5 6 7)

E: (7 6 5 4 3 2 1)

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

G: 7

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

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

J: error

K: none of the above

For each expression below, identify the item in the above list that matches its behavior.

Question 14: G

```
(tree-message + test-tree (lambda (x) 1) 0)
```

Question 15: C

```
(tree-message + test-tree (lambda (x) x) 0)
```

Question 16: A

```
(tree-message add-branch test-tree (lambda (x) x) empty-tree)
```

Question 17: E

```
(tree-message (lambda (x y) (append y x)) test-tree list '())
```

Question 18: I

```
(tree-message (lambda (x y) (reverse (add-branch x y))) test-tree (lambda (x) x) empty-tree)
```

Question 19: H

```
(tree-message (lambda (x y) (add-branch x (reverse y))) test-tree (lambda (x) x) empty-tree)
```