

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
 Department of Electrical Engineering and Computer Science
 6.001—Structure and Interpretation of Computer Programs
 Fall Semester, 2004

Quiz II – Example 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	30		
2	26		
3	24		
4	20		
Total	100		

For your reference:

Section	Time	Location	Rec. Instructor	Tutors
10	10:00	26-328	Seth Teller	Ben Vandiver
11	11:00	26-314	Srini Devadas	David Ziegler
12	12:00	26-3314	Srini Devadas	David Pritchard
1	1:00	26-372	Trevor Darrell	Marshall Tappen
2	2:00	26-372	Trevor Darrell	All

Part 1: (30 points)

In lecture, we introduced the idea of a binary tree. This is a data structure consisting of nodes. Each node contains a value, and a left and right branch, each of which is a (possibly empty) binary tree.

Here is a template for a data abstraction of a binary tree:

```
(define (node value left right)
  (list value left right))
(define value car)
(define left cadr)
(define right caddr)
(define empty-tree? null?)
(define the-empty-tree nil)
```

We want to support two operations on a binary tree, determining if a value is contained at some node in the tree, and inserting elements into the tree. Binary trees have the property that all the values to the left of a node are less than the value at the node, and all the values to the right of a node are greater than the value at a node.

Here is an outline of code to determine if a value is present in a binary tree.

```
(define (present? val tree)
  (cond ((empty-tree? tree) #f)
        ((= val (value tree)) #t)
        ((< val (value tree))
         ANSWER-1)
        (else JUST-LIKE-ANSWER-1)))
```

Question 1. What code should be used in place of ANSWER-1?

(present? val (left tree))

Note that by symmetry, the code to be used in place of JUST-LIKE-ANSWER-1 will be very similar to that for ANSWER-1, so we will just assume that it exists.

Here is an outline of code to insert a new value into a binary tree. This procedure will return a new tree, with the value insert in the right part of the tree.

```
(define (insert val tree)
  (cond ((empty-tree? tree)
        ANSWER-2)
        ((= val (value tree))
         ANSWER-3)
        ((< val (value tree))
         ANSWER-4)
        (else JUST-LIKE-ANSWER-4)))
```

Question 2. What code should be used in place of ANSWER-2?

(node val the-empty-tree the-empty-tree)

Question 3. What code should be used in place of ANSWER-3?

tree

Question 4. What code should be used in place of ANSWER-4?

(node (value tree) (insert val (left tree)) (right tree))

By symmetry, the code to be used in place of JUST-LIKE-ANSWER-4 will be very similar to that used for ANSWER-4 so we will just assume that the code exists.

Suppose that we want to measure the order of growth of these procedures in space, **measured by the maximum number of deferred operations at any point in the computation**, and the order of growth in time, **measured by the total number of primitive operations** (which here we restrict to `cons`, `car`, `cdr`, `null?`, `=` and `<`).

For each order of growth question, you should choose from:

- $\Theta(1)$
- $\Theta(\log n)$
- $\Theta(n)$
- $\Theta(n \log n)$
- $\Theta(n^2)$
- $\Theta(2^n)$
- other

Question 5.

What is the order of growth **in space and in time** for `present?`, where n is the number of nodes in the tree? Assume that the tree is balanced, that is on average there are as many elements in the left and right branch of each node.

Space: $\Theta(1)$

Time: $\Theta(\log n)$

Question 6.

What is the order of growth **in space and in time** for `insert`, where n is the number of nodes in the tree? Assume that the tree is balanced, that is on average there are as many elements in the left and right branch of each node.

Space: $\Theta(\log n)$

Time: $\Theta(\log n)$

Now suppose we want to insert new values into a binary tree, using mutation to modify the existing tree. For example:

```
(define test (node 10 the-empty-tree the-empty-tree))

(insert! 2 test)

(insert! 20 test)

test
;Value: (10 (2 #f #f) (20 #f #f))
```

To do this, we will need to mutate the structure of a node in a tree.

Question 7.

Write a procedure, **change-left!**, that takes a node and a tree as arguments, and mutates the node so that the left branch now points to the new tree.

```
(define (change-left! node tree) (set-car! (cdr node) tree))
```

You may assume that a very similar procedure, **change-right!**, exists that takes a node and a tree as arguments, and mutates the node so that the right branch now points to the new tree.

Using these two procedures, complete the following code:

```
(define (insert! val tree)
  (cond ((empty-tree? tree)
        ANSWER-8)
        ((= val (value tree))
         nil) ;; value already present
        (< val (value tree))
        ANSWER-9)
        (else
         JUST-LIKE-ANSWER-9)))
```

insert! should have the following behavior. If we insert an element into an empty tree, it should return a tree with one leaf. Otherwise, the return value of **insert!** is irrelevant, as we use it to perform a side effect on an existing tree.

Question 8: What code should be used in place of **ANSWER-8**?

(node val the-empty-tree the-empty-tree)

Question 9: What code should be used in place of **ANSWER-9**?

```
(if (empty-tree? (left tree))
    (change-left! tree
      (node val the-empty-tree the-empty-tree))
    (insert! val (left tree)))
```

By symmetry, the code to be used in place of **JUST-LIKE-ANSWER-9** will be ver similar to that for **ANSWER-9** so we will just assume that the code exists.

Question 10.

What is the order of growth **in space and in time** for **insert!**, where n is the number of nodes in the tree? Assume that the tree is balanced, that is on average there are as many elements in the left and right branch of each node.

Space: $\Theta(1)$

Time: $\Theta(\log n)$

Part 2: (26 points)

A convenient tool for tracing procedure usage is called a **metered** procedure. For example:

```
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))))

(define my-fact (metered fact))

(my-fact 3)
;Value: 6

(my-fact 5)
;Value: 120

(my-fact 'count)
;Value: 2

(my-fact 'calls)
;Value: (5 3)
```

Thus, in our metered version of the procedure, we can ask how many times the procedure has been called, and what the arguments were on which it was called.

Suppose we evaluate the following, in order:

```
(define (metered proc)
  (let ((count 0)
        (calls '()))
    (lambda (arg)
      (cond ((and (symbol? arg) (eq? arg 'reset))
             (set! count 0)
             (set! calls '()))
            ((and (symbol? arg) (eq? arg 'count))
             count)
            ((and (symbol? arg) (eq? arg 'calls))
             calls)
            (else
             (set! count (+ count 1))
             (set! calls (cons arg calls))
             (proc arg))))))

(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))))

(define my-fact (metered fact))

(my-fact 1)
```

Attached is a partially completed environment diagram that shows the state of the environment after evaluation of these three expressions. Using the frame labels **GE**, **E1**, **E2**, **E3**, **E4** or **E5**, or the procedure object labels **P1**, **P2**, **P3** or **P4**, answer the following questions:

Question 11: For each of the following environments, indicate the value of its enclosing environment, using one of **GE**, **E1**, **E2**, **E3**, **E4**, **E5** or **none** or **not shown**.

	Enclosing environment
GE	none
E1	GE
E2	GE
E3	GE
E4	E5
E5	E3

Question 12: For each of the following variables, indicate the value to which it is bound, using a number, a list, or one of **GE**, **E1**, **E2**, **E3**, **E4**, **E5**, **P1**, **P2**, **P3**, **P4**

Variable	Environment	Value to which bound
metered	GE	P1
my-fact	GE	P4
fact	GE	P2
proc	E3	P2
count	E5	1
calls	E5	(1)
arg	E4	1

Part 3: (24 points)

In lecture, we saw some examples of **graphs**, which consist of a set of **nodes**, each of which contains a value and a set of neighbors (other nodes):

```
(define (make-node value neighbors)
  (list value neighbors))

(define node-value car)
(define node-neighbors cadr)

(define (add-neighbor! node new)
  (set-car! (cdr node) (cons new (cadr node))))

(define n4 (make-node '(delta epsilon) '()))
(define n3 (make-node 'gamma (list n4)))
(define n2 (make-node 'beta (list n3)))
(define n1 (make-node 'alpha (list n2)))
(add-neighbor! n4 n1)
```

If we want to find an node with a particular value, we can search through the graph, starting at some designated start point. For purposes of this question, you may assume that values stored at nodes are symbols or lists of symbols.

```
(define (find value where)
  (define (helper value todo done)
    (cond ((null? todo) 'not-found)
          ((memq (car todo) done) ;; have already examined next node
           ANSWER-13)
          (ANSWER-14
           (car todo)) ; return the actual node
          (else ANSWER-15-16)))
  (helper value (list where) '()))
```

The procedure `find` is used to retrieve the node containing the value being sought. The idea is to use `todo` to keep a list of nodes to be examined, and to use `done` to keep track of those nodes which it has already examined (so we don't loop forever inside the graph). The procedure `memq` returns a true value if the first argument is contained in the second argument (which is a list).

Question 13. What code should be used to handle the search when the next node in the `todo` list has already been examined? (ANSWER13)

```
(helper val (cdr todo) done)
```

Question 14. What code should be used to test if the next node in the `todo` list has an value that matches the value being sought? (ANSWER14)

```
(equal? val (node-value (car todo))) – 4
```

Question 15. What code should be used to continue the search, with the property that the procedure should execute a **breadth first** search? (ANSWER-15-16)

```
(helper val  
  (append (cdr todo) (node-neighbors (car todo)))  
  (cons (car todo) done))
```

Question 16. What code should be used to continue the search, with the property that the procedure should execute a **depth first** search? (ANSWER-15-16)

```
(helper val  
  (append (node-neighbors (car todo)) (cdr todo))  
  (cons (car todo) done))
```

Part 4: (20 points)

A **ring** is a new kind of data structure, in which the elements are conceptually linked like a circular list. Each element has a next element and a previous element, as well as a value associated with it. We can implement a ring using message passing:

```
(define (elt val prev next)
  (lambda (msg)
    (case msg
      ((value) val)
      ((previous) prev)
      ((next) next)
      ((set-previous) (lambda (who) (set! prev who)))
      ((set-next) (lambda (who) (set! next who))))))

(define e1 (elt 1 nil nil))
(define e2 (elt 2 nil nil))
(define e3 (elt 3 nil nil))
```

To construct a ring from a list of elements, we must set up the **next** and **previous** links in each element. First, we can link all the **next** pointers, using the following procedure. The underlying idea is to create a **next** link from each element in the list to the subsequent element in the list, and finally to create a **next** link from the last element in the list to the first, for example by

```
(create-ring-from-list (list e1 e2 e3))

(define (create-ring-from-list lst)
  ;; assumes lst is not empty and is in the order desired
  (define (linkup current rest)
    (cond ((null? rest)
           ANSWER-17)
          (else ANSWER-18
                 (linkup (car rest) (cdr rest)))))
  (linkup (car lst) (cdr lst)))
```

Question 17: What code should be used in place of ANSWER-17?

```
((current 'set-next) (car lst))
```

Question 18: What code should be used in place of ANSWER-18?

```
((current 'set-next) (car rest))
```

This creates our “next” links, but we still need to set up our “previous” links, that is, each element should also have a link to the previous element in the ring. To do this, we use:

```
(define (invert-all start)
  (if (null? (start 'previous))
      (begin (invert-element start)
              (invert-all (start 'next)))
      'done))
```

which relies on the procedure `invert-element`. This procedure should create a “previous” link for the current element:

```
(define (invert-element where)
  (define (find start)
    (if ANSWER-19
        ANSWER-20
        (find (start 'next))))
  (find where))
```

Question 19: What code should be used in place of ANSWER-19, in order to tell when the value associated with the variable `start` is the element immediately previous to the value associated with the variable `where`?

`(eq? (start 'next) where)`

Question 20: What code should be used in place of ANSWER-20, in order to create a “previous” link for the value associated with the variable `where`?

`((where 'set-previous) start)`