

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 2005

Quiz II

Closed Book – two sheets 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 time: Tutor's Name:

| PART | Value | Grade | Grader |
|-------|-------|-------|--------|
| 1 | 16 | | |
| 2 | 30 | | |
| 3 | 26 | | |
| 4 | 15 | | |
| 5 | 13 | | |
| Total | 100 | | |

Part 1. (16 points)

Here is a “cute” way of writing a procedure of one argument, so that each time it is evaluated with an argument, it returns the result of its previous evaluation (it returns false the first time it is evaluated).

```
(define (previous proc)
  (let ((last #f))
    (lambda (arg)
      (let ((temp last))
        (set! last (proc arg))
        temp))))
```

For example, if we evaluate:

```
(define funny-square (previous (lambda (x) (* x x))))
```

then we get

```
(funny-square 5) ==> #f
```

```
(funny-square 10) ==> 25
```

Suppose we trace out with an environment model the evaluation of the two definitions, and the first call to funny-square. Attached at the end of the quiz is a partially completed environment diagram. You are to complete this diagram and then use the labels on the parts of the environment diagram to answer the following questions.

Question 1: For each of the following frames, indicate the enclosing environment, choosing one of GE, E1, E2, E3, E4, E5 or none or not shown.

| Frame: | Enclosing Environment |
|--------|-----------------------|
| GE | none |
| E1 | GE |
| E2 | E1 |
| E3 | E2 |
| E4 | E3 |
| E5 | GE |

1 point each

Question 2: For each of the following procedure objects, indicate the enclosing environment, choosing one of GE, E1, E2, E3, E4, E5 or none or not shown.

| Procedure: | Enclosing Environment |
|------------|-----------------------|
| P1 | GE |
| P2 | GE |
| P3 | E2 |

Question 3: For each of the following variable names, indicate the value to which it is bound at the end of the evaluation of the expressions, choosing one of GE, E1, E2, E3, E4, E5 or P1, P2, P3 or a symbol, a number or a boolean value.

| Variable: | Environment | Value |
|--------------|-------------|----------------|
| funny-square | GE | P3 |
| previous | GE | P1 |
| proc | E1 | P2 |
| last | E2 | 25 |
| arg | E3 | 5 |
| temp | E4 | # f |
| x | E5 | 5 |

Part 2: (30 points)

We are going to explore a new data structure, called a **binary tree**, and implement it as an abstract data type. This is a tree structure that has the property that each node has at most two branches, a left and right branch. Each node also has a value associated with it. For the purposes of this problem, we will assume those values are integers. Finally, a binary tree has the property that all the values in the left branch of a node are less than the value at the node, and all the values in the right branch of a node are greater than the value at the node. We also assume that no value appears more than once in a binary tree.

We implement a node in the tree using a list:

```
;; data abstraction

(define (make-node left right node-contents)
  (list left right node-contents))

(define (empty-tree? tree)
  (null? tree))

(define empty-tree '())
```

Here is an example of a simple tree:

```
(define subtree1 (make-node empty-tree empty-tree 6))
(define subtree2 (make-node empty-tree empty-tree 15))
(define tree1 (make-node subtree1 subtree2 10))
```

A conceptual view of this tree, as well as the corresponding box-and-pointer diagram for this structure is shown in the figure attached at the end of the quiz. Note that a binary tree need not be balanced (i.e. nodes could have only a left or a right branch).

Question 4:

Write the corresponding selectors for this constructor, called `left`, `right` and `node-contents`

3

```
(define left car)
(define right cadr)
(define node-contents caddr)
```

Now to add a new element into a binary tree, we will use the following code, which does not use mutation, and returns a new version of a tree:

```
(define (insert v tree)
  (cond ((empty-tree? tree) (make-node empty-tree empty-tree v))
        ((= v (node-contents tree)) tree)
        (< v (node-contents tree))
          ANSWER-5
        )
        (else
          ANSWER-6)))
```

Question 5: What code should be used in place of ANSWER-5?

(5) `(make-node (insert v (left tree))
(right tree)
(node-contents tree))`

Note that ANSWER-6 will have a great deal of similarity with ANSWER-5, so we will assume that the appropriate code has been provided for this part.

If we want to insert several elements into a tree, we can use:

```
(define (insert-all lst tree)
  (if (null? lst)
      tree
      (insert-all (cdr lst) (insert (car lst) tree))))
```

To see if an element is in the tree, we can use

```
(define (find v tree)
  (cond ((empty-tree? tree) #f)
        ((= v (node-contents tree)) #t)
        (< v (node-contents tree))
          (find v (left tree))
        (else (find v (right tree)))))
```

A useful thing to do with a tree is flatten it, that is, return a list of the node-contents of the tree.

Question 6: Complete the definition of flatten

```
(define (flatten tree)
  (if (empty-tree? tree)
      '()
      ANSWER-6))
```

7

```
(append (flatten (left tree))
        (list (node-contents tree)
              (flatten (right tree))))
      or
      (cons (node-contents tree)
            (flatten (right tree)))

or
(cons (node-contents tree) (append (flatten (left tree))
                                  (flatten (right tree)))))
```

Now suppose we want to remove an element from a tree. An example is shown in the attached figure, where an element is removed, and the resulting tree still preserves the ordering required. Here is a template we can use:

```
(define (remove v tree)
  (cond ((empty-tree? tree) empty-tree)
        ((= v (node-contents tree))
         (cond ((empty-tree? (left tree)) (right tree))
               ((empty-tree? (right tree)) (left tree))
               (else
                (let ((new-node
                      (make-node (left tree)
                                 empty-tree
                                 (node-contents (right tree)))))
                  ANSWER-7))))
        ((< v (node-contents tree))
         (make-node (remove v (left tree))
                    (right tree)
                    (node-contents tree)))
        (else (make-node (left tree)
                          (remove v (right tree))
                          (node-contents tree))))))
```

To complete this code, we need to write ANSWER-8. The idea we want you to use is to make a new subtree, in which we keep the same left branch of the current subtree, but we “pull up” the value at the top of the right subtree to be the new value of this branch. We then need to insert the remaining elements of the right subtree into the tree.

We can do this by using insert-all and flatten

Question 7: Using this, complete the code for remove by supplying code for ANSWER-7.

(insert-all (append (flatten (left right tree))
 (flatten (right right tree)))
 new-node)

Finally, we could instead do insertion “in-place” meaning by mutating the existing tree structure.

Question 8:

Complete the following code:

```
(define (insert! v tree)
  (cond ((empty-tree? tree) (make-node empty-tree empty-tree v))
        ((= v (node-contents tree)) tree)
        ((< v (node-contents tree))
         (if (empty-tree? (left tree))
             (mutate-left! tree ANSWER-8-A)
             ANSWER-8-B))
        (else
         (if (empty-tree? (right tree))
             (mutate-right! tree ANSWER-8-C)
             ANSWER-8-D))))
```

Note that the code for ANSWER-8-C will be very similar to ANSWER-8-A, and ANSWER-8-D will be very similar to ANSWER-8-B, so we need only complete one set.

First, the procedure mutate-left! should take as argument two trees, and change the left branch of the first tree to point to the second tree. Provide this procedure

(define (mutate-left! tree new) or (define mutate-left! proc!)
 (set-cdr! tree new))

Next, the procedure `mutate-right!` should take as argument two trees, and change the right branch of the first tree to point to the second tree. Provide this procedure

2

```
(define (mutate-right! tree new)
  (set-car! (cadr tree) new))
```

Complete ANSWER-8-A below.

2

```
(make-node empty-tree empty-tree v)
```

Complete ANSWER-8-B below.

2

```
(insert! v (left tree))
```

Part 3. (26 points)

We are going to count the number of times each word appears in a list of words, using a data structure called a **histrogram**. The basic element of our histogram is a count

```
(define (make-count word)
  (list word 1))

(define get-word car)
(define how-many cadr)

(define (increment-count count)
  (list (get-word count)
        (+ 1 (how-many count))))
```

We will represent a histogram as a list of counts.

Given a word (which is represented by a symbol, not a string), and a histogram, we want to create a new histogram with the number of times that word has been seen increased by one. Note that there will be exactly one count for each word in the histogram.

```
(define (update-histogram-once word hist)
  (cond ((null? hist)
        ANSWER-9)
        ((eq? word (get-word (car hist)))
        ANSWER-10)
        (else
        ANSWER-11)))
```

You are to complete this code. Note that this should be done without mutation. Also note that a word may not yet be in the histogram, so your code will have to create a new count in that case.

Question 9

What code is needed for ANSWER-9?

③

```
(list (make-count word))
```

Question 10

What code is needed for ANSWER-10?

④

```
(cons (make-count (car hist))
      (cdr hist))
```

Question 11

What code is needed for ANSWER-11?

④

```
(cons (car hist)
      (update-histogram-once word (cdr hist)))
```

Now, given a whole list of words, we want to update the entire histogram, using the above procedure:

```
(define (update-histogram words hist)
  (if (null? words)
      hist
      ANSWER-12))
```

Question 12

Write the code for ANSWER-12.

⑤

```
(update-histogram (cdr words)
                  (update-histogram-once (car words) hist))
```

Now suppose we want to sort the histogram into alphabetical (also known as lexicographic) order. Here is a procedure that will do this:

```
(define (sort hist)
  (if (null? hist)
      '()
      (let ((next (smallest hist)))
          (cons next (sort (remove next hist))))))

(define (smallest hist)
  (define (help current rest-hist)
    (cond ((null? rest-hist) current)
          ((symbol? get word current) get word (car rest-hist))
          (else ANSWER-13)))
  (help (car hist) (cdr hist)))
```

```
(define (remove elt hist)
  ANSWER-15)
```

Question 13

Assuming that `symbol<?` is a predicate that returns true if the first argument would appear before the second one in a dictionary, complete the code for ANSWER-13.

②

```
(help (current (cdr rest-hist)))
```

Question 14

Assuming that `symbol<?` is a predicate that returns true if the first argument would appear before the second one in a dictionary, complete the code for ANSWER-14.

②

```
(help (car rest-hist) (cdr rest-hist)))
```

Question 15

Complete the code for ANSWER-15.

⑥

```
(if (eq? elt (car hist))
    (cdr hist)
    (cons (car hist) (remove elt (cdr hist))))))
```

Part 4: (15 points)

It would be nice if we could “undo” definitions, if we change our mind about a particular variable value.

For example, consider the following behavior (where the value returned is shown at the right):

```
(define test (start 5)) ==> 5
(value test) ==> 5
(undefine test 10) ==> 10
(value test) ==> 10
(undefine test '()) ==> 5
(undefine test '()) ==> error
```

In other words, we initially define a variable value, using `start`. If we then `redefine` the variable to a new value, it gets that value. If we `redefine` the variable with an empty list as the “new value”, it reverts to the previous value, and will continue reverting until it runs out of previous defined values.

To implement this, we are going to use a procedure with internal state:

```
(define (start val)
  (let ((history '())
        (current val))
    (lambda (action newval)
      (cond ((eq? action 'value) 'no-change)
            ((eq? action 'change)
             ANSWER-16)
            ((eq? action 'unchange)
             ANSWER-17)
            (else (error "I don't know how to do this")))
      current)))

(define (value arg)
  (arg 'value ()))

(define (redefine arg newval)
  (if (null? newval)
      (arg 'unchange '())
      (arg 'change newval)))
```

Question 16:

Complete the code for ANSWER-16.

(6)

```
(set! history (cons current history))  
(set! current new val)
```

Question 17:

Complete the code for ANSWER-17.

(9)

```
(if (null? history)  
    (print "error")  
    (begin (set! current (car history))  
           (set! history (cdr history))))
```

Part 5: (13 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 `done?`. All we need to know about the tree abstraction, in addition to `done?`, is that it includes a selector `children`, which when applied to a node in the tree returns a list of that node's children.

Here is a template for searching in a tree:

```
(define (search start done? merge-fn)
  (define (search1 queue)
    (if (null? queue)
        #f
        (let ((current (car queue)))
          (if (done? current)
              current
              (search1 (merge-fn (children current)
                                (cdr queue)))))))
  (search1 (list start)))
```

This procedure uses a queue, represented by a list, to keep track of nodes yet to be searched. It applies a procedure (`done?`) to determine when it has found the node for which it is looking. And it uses a procedure (`merge-fn`) to combine children of the current node with nodes still in the queue to be checked, to create a new queue.

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

Question 18: For a depth first search, what argument should be used for `merge-fn`?

②

`append`

Question 19: For a breadth first search, what argument should be used for `merge-fn`?

④

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

Question 20: For a best first search, what argument should be used for `merge-fn`?

⑤

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

G
E

funny-square
previous

E
1

proc

E
2

last

E
3

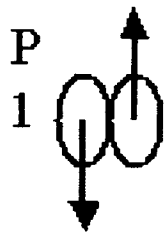
arg

E
4

temp

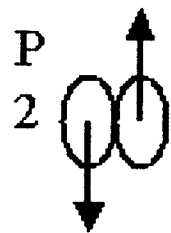
E
5

x



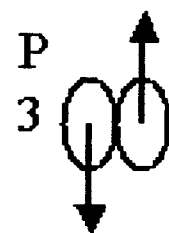
par: proc

body: (let
((last ...



par: x

body: (* x x)

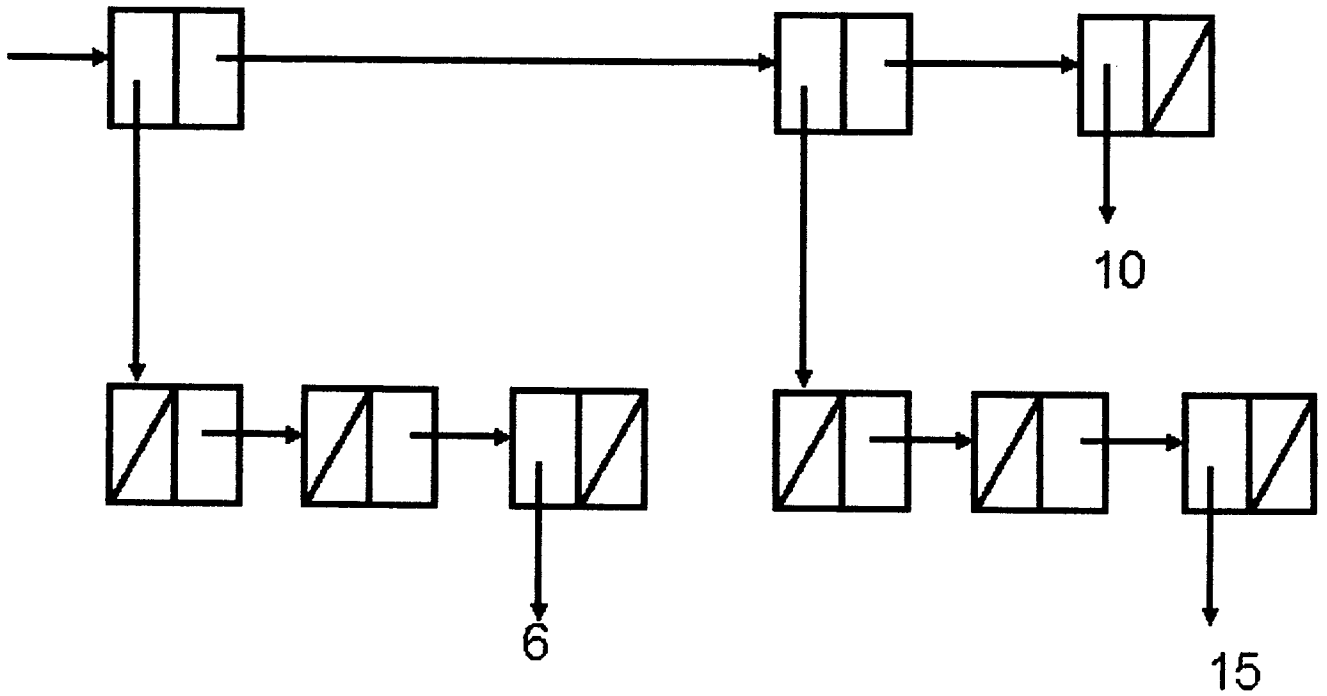


par: arg

body: (let
((temp ...

Figure for Part 2

Box-and-pointer view
of binary tree



Abstract view of
binary tree

