MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall 1997

Problem Set 6

Issued: Tuesday, October 21, 1997

Due: Wednesday, October 29, in recitation.

Tutorial preparation for the week of October 27:
SICP Exercises 3.10, 3.12, 3.16, 3.23.

Reading for this Problem Set: *SICP* Sections 3–3.3.3
Reading for Lectures: *SICP* Sections 3–3.3.3, 3.5

Searching Cyclic Graphs

1 Searching Graphs

Searching through trees, list-structures, links on the world-wide web, cities on a map, are all commonplace activities. The procedures for carrying out such searches are a case where programming with side-effects (from both `set!` and `set-car!`, `set-cdr!`) is the natural approach.

The structures we want to search through can be modelled by an abstract data type known as a *directed graph.* A directed graph consists of a set of *nodes*; each node N has a set of directed *edges* from it to some other nodes. These other nodes are said to be *adjacent* to N and are also called the *sons* of N; conversely, N is the *father* of it sons. In this problem set, each node of a graph will have a also *name* which is a Scheme symbol, and some other stuff we'll get to shortly.

For example, we might abstract the 6.001 staff reporting structure as having nodes Rod, Albert, Daniel, Martin, Saman, Steve, Tara, Robin, Christiana, Ien, Anna, Barb, Ben, Jeremy, Jill, David. There will be an edge from each staff member all other staff members to whom weekly reports or directions msut be given. So there are edges from both Rod and Albert to Daniel, Martin, and Saman, to whom Rod and Albert have to report the weekly plan for recitation. There is an edge from each TA to their recitation instructor, because they should report how their tutees have done on the weekly problem set. There is an edge from Rod to Albert, another one back from Albert to Rod because they have to coordinate their weekly lectures, one from Rod to Jill and one back, etc.

The standard representation for a directed graph is an *adjacency list*. This is simply a list of information about each node. The node information consists of its name and a list of the names of its sons. For example, the adjacency list for the 6.001 weekly reporting structure (as far as we have decribed it) would be:

```
(define 6001-reporting
  '((Rod (Albert Daniel Martin Saman Jill))
    (Albert (Rod Daniel Martin Saman Jill))
    (Ien (Saman))
    (Tara (Saman))
    (Anna (Albert))
    (Steve (Daniel))
     ...
    (Jill (Albert Rod))
    ...))
```

You can see the full adjacency list in the attached code file `6001rprt.scm`.

2 Tree Search

Suppose we wanted to determine if a binary tree with labelled nodes contained a node with a given label. Here (in contrast to *SICP*, Section 2.3.3), we are not assuming that the labels appear in any particular arrangement. Then we would perform the obvious "tree search": see if the given label is the label of the root node of the tree, or, recursively, see if the given label appears in the left or right subtrees. This corresponds to the procedure:

```
(define (depth-first-tree-search label tree)
  (and (not (empty? tree))
       (or (same-label? label (label-of tree))
           (depth-first-tree-search label (left-branch tree))
           (depth-first-tree-search label (right-branch tree)))))
```

Note that the order in which we inspected the root label or the subtrees does not effect whether we find the label. For example, another perfectly good tree search is:

```
(define (other-tree-search label tree)
  (and (not (empty? tree))
       (or (other-tree-search label (left-branch tree))
           (other-tree-search label (right-branch tree))
           (same-label? label (label-of tree)))))
```

Pre-Lab Exercise 1: In the code file `trees.scm` you will find the definition of a tree:

```
;                                    1
;                                /       \
;                               2         3
;                             /   \       / \
;                            4     5     6   7
;                                 / \
;                                8   9
```

List the order in which the nodes of tree are visited in the evaluation of

```
(depth-first-tree-search 10 tree)
```

and

```
(other-tree-search 10 tree)
```

There is another important tree search strategy known as *breadth-first* search which works a little differently. Instead of recursively searching whole subtrees, we search the tree level by level: check if the root has the given label, then check *all* the sons of the root, then all the grandsons, etc. This is easily managed by keeping a list of nodes to be checked. Starting with the list consisting solely of the root, check for the label at the first node on the list, and if it is not there, delete that first node but add its sons *at the end of the list.* This idea translates directly into a tail-recursive procedure:

```
(define (breadth-first-tree-search label tree)
  (define (breadth-first-trees subtrees)
    (and (not (null? subtrees))
         (let ((subtree (car subtrees)))
           (cond ((empty? subtree)
                  (breadth-first-trees (cdr subtrees)))
                 ((same-label? label (label-of subtree)) #t)
                 (else (breadth-first-trees
                        (append (cdr subtrees)
                                (list (left-branch subtree)
                                      (right-branch subtree))))))))))
  (breadth-first-trees (list tree)))
```

Pre-Lab Exercise 2: List the order in which the nodes of tree are visited in the evaluation of

```
(breadth-first-tree-search 10 tree)
```

Lab-exercise 3: To verify your answers to Pre-Lab Exercises 1 and 2, you can modify the `same-label?` procedure to "remember" the sequence of tree labels supplied to it during a search. To read and reset the remembered labels, procedures `show-visited-labels` and `clear-visited-labels!` will also be needed.

Complete the following definition of a visited-label mechanism and use it to check your answers to Pre-Lab Exercises 1 and 2. Watch out!: now that you're programming with side-effects, you'll have to pay careful attention to the state of the remembered labels. Be sure to `clear-visited-labels!` between each tree search.

```
(define (show-visited-labels) 'uninitialized)
(define (clear-visited-labels!) 'uninitialized)

(define (install-visited-labels!)
  (let ((nodes '()))
    (set! same-label?
      (lambda (given-label visited-label)
;;     ...))
    (set! clear-visited-labels!
;;   ...)
    (set! show-visited-labels
;;   ...)))

(install-visited-labels!)
```

Lab-exercise 4: The tree-search procedures above generalize naturally to arbitrary, not just binary, trees. Modify the tree-search procedures above to work on the adjacency-list representation of arbitrary trees. Show some tests (as always) of your procedures on several adjacency lists.

3 Cyclic graph search

Unfortunately, the `6001-reporting` graph above is not a tree. For example, Rod has Albert as a son, but Albert also has Rod as a son. A sequence of nodes, each of which is the son of the preceding node, that ends with the same node at which it began, is called a *cycle* in a directed graph. So Rod, Albert, Rod is a cycle of two nodes in the graph; also Jill, Albert, Rod, Jill is a cycle of three nodes.

Pre-Lab Exercise 4 Describe what happens when the following expressions are evaluated in order:

```
(define tree '(1 (2 () ()) (3 () ())))
(set-car! (cdr tree) tree)
(caddr tree)
(car (cddadr tree))
(breadth-first-tree-search 4 tree)
```

Tree search terminates because searching a tree is done by recursively searching the sons of the root of the tree, and the sons are *smaller* than the tree. In a graph with cycles, in which a son could have its own father as a descendant, there is no longer any obvious sense in which a son is smaller than its father.

While the tree search procedures have a problem on graphs with cycles, the basic idea behind them extends easily to such graphs. All that is needed is the ability to "mark" a node when it is visited. When the search encounters a marked node, it can ignore it since the mark indicates that the node has already been searched from — or is in the process, recursively, of being searched from.

Here is the version of depth-first search which will search an arbitrary directed graph, starting from a node with a given `name`, until it finds a satisfactory node. The value of parameter `found-it?` is intended to be a predicate which determines whether a visited node is satisfactory.

```
(define (depth-first-search name graph found-it?)
  (define (search-from father)
    (if (found-it? father)
        father
        (search-sons (adjacent-nodes father))))
  (define (search-sons sons)
    (if (null? sons)
        (fail)
        (let ((first-son (car sons)))
          (if (marked? first-son)
              (search-sons (cdr sons))
              (begin (mark! first-son)
                     (let ((first-son-result
                            (search-from first-son)))
                       (if (fail? first-son-result)
                           (search-sons (cdr sons))
```

```
                         first-son-result)))))))
    (begin (install-marking-package! graph)  ;clears any old marks
           (let ((node (node-with-name name)))
             (begin (mark! node)
                    (node-name (search-from node))))))
```

Here we have treated `graph` as an abstract data type with a procedure `adjacent-nodes` which returns the sons of a node. Nodes are assumed to have distinct names, and procedures `node-with-name` and `node-name` go back and forth between nodes and their names. (The reason to distinguish nodes from their names will be explained very soon.)

There is also an abstract marking mechanism with a procedure `mark!` to mark a node and a procedure `marked?` to determine whether a node has been marked. Now the recursive graph searching process terminates because *the number of unmarked nodes* decreases at each recursive call.

In the same way, we can define

```
(define (breadth-first-search name graph found-it?)
  (define (bfs nodes)
    (if (null? nodes)
        (fail)
        (let ((node (car nodes)))
          (cond ((marked? node) (bfs (cdr nodes)))
                ((found-it? node) node)
                (else
                 (begin
                   (mark! node)
                   (bfs (append (cdr nodes) (adjacent-nodes node)))))))))
  (begin (install-marking-package! graph)
         (node-name (bfs (list (node-with-name name))))))
```

Lab-Exercise 5: Use the graph searching procedures to determine some "reporting" paths between several different pairs of 6001 staff members pages according to the graph given in `6001rprt.scm`. To do this, you will have to apply the `linkup` procedure in `search.scm` to convert the `6001-reporting` adjacency-list representation into an abstract graph representation

```
(define 6001-graph (linkup 6001-reporting))
```

as well as to define suitable `found-it?` procedures to use.

Lab-Exercise 6: The idea we used in Lab-Exercise 3 to collect information about visited tree nodes will work for graph search too. Define a `found-it?` procedure and appropriate auxiliary procedures (for example, like `show-visited-labels!`)

so that the graph search procedures will stop after finding ten nodes and allow the user to see the set of visited node names.

4 Finding Paths

When a desired node is found through a graph search, the search has implicitly discovered a path from the starting node to the desired one. It is often useful to know this path explicitly, and it is easy to modify the search procedures to do this. Since the father of a node is always available when the node is first reached in a search, all that is needed is to record the father inside a node whenever it is first reached.

For this purpose, we extend our graph abstraction to include a mechanism for setting, `set-value!`, and retrieving, `node-value`, the value in a node. In this problem set, the only values we make use of are node fathers.

When we're interested in finding paths, we would expect breadth-first search to be the preferred method, since it finds the shortest path between an initial node and a desired node.

```
(define (breadth-first-search-path-back name graph found-it?)
  (define (bfsb sons)
    (if (null? sons)
        (fail)
        (let ((first-son (car sons)))
          (cond ((marked? first-son) (bfsb (cdr sons)))
                ((found-it? first-son)
                 first-son)
                (else
                 (begin
                   (mark! first-son)
                   (let ((grandsons (adjacent-nodes first-son)))
                     (begin
                       (for-each
                        (lambda (grandson)
                          (if (fail? (node-value grandson))
                              (set-value! grandson first-son)))
                        grandsons)
                       (bfsb (append (cdr sons) grandsons)))))))))
  (begin
    (install-value-package! graph) ;clears any old marks and values
    (let ((node (node-with-name name)))
      (begin
        (set-value! node node)
        (node-name (bfsb (list node)))))))
```

```
(define (path-back-from name graph)
  (define (path-back node)
    (let ((val (node-value node)))
      (cond ((eq? val node)
             (list (node-name node)))
            ((fail? val) (fail))
            (else
             (cons (node-name node) (path-back val))))))
  (path-back (node-with-name name)))
```

Lab-Exercise 7: Use the `-path-back` search procedures on your examples from Lab-Exercise 5 to find the paths between the nodes you found were connected.

5 Linked graphs

In the adjacency list representation of a graph, a node *is* its name. But suppose we want to find information about the first son of a node — say whether that first son has a son — from the information about the node. Finding the first son, S, from the information about a node, N, is easy: it is the `caadr` of the node information for N (the `car` is the name N and the `cadr` is the list of sons). But to find out if S has a son, it is necessary to search through the adjacency list for the node information about S; at that point the sons of S are the `cadr` of the node information for S, but the search for the node information about S could take time proportional to the number of nodes in the graph.

The procedure `linkup` converts an adjacency list representation into a more efficient one. In a linked-graph representation a node *is* its node information. So from the information about N, to find out whether the first son S of N has a son, we simply take `caadr` of N to get S, and then `cadr` of S to get its sons: no search for the node information about S because it is right there.

Prelab-Exercise 8a: Draw the box and pointer diagram for

```
(linkup '((1 (2 3)) (2 ()) (3 (1))))
```

Lab-Exercise 8b: Verify your answer to 8a (as best you can) by evaluating `(linkup '((1 (2 3)) (2 ()) (3 (1))))`.

Lab-Exercise 9: Define an `unlinked` procedure which undoes `linkup`: `(unlinked (linkup adj-list))` will `equal?` `adj-list` whenever the value of `adj-list` is an adjacency list.

Optional Lab Exercise 10: The linked-graph representation is tricky to figure out and to remember. And because the graph search procedures are written in terms of a graph abstraction, they will work without change if the standard adjacency-list representation is used throughout for graphs. Instead of storing marks and values *in* the graph representation, they can be "remembered" by the marking and value setting procedures themselves, as in Lab-exercise 3. Complete the definitions of

```
(define (install-naive-marking-package! adjacency-list)
  (let ((marks '()))
    (begin
      (set! node-name ...)      ;export NODE-NAME
      (set! node-with-name ...) ;export NODE-WITH-NAME
      (set! marked? ...)        ;export MARKED?
      (set! mark! ...)          ;export MARK!
      (set! adjacent-nodes
            (lambda (node)
              (let ((node-info ...)
                (if node-info
                    (cadr node-info)
               ;ADJACENCY-LIST may omit node not adjacent to anything:
                    '())))))))
```

Optional Lab Exercise 11: We dealt with a lot of circular list structures in this problem set and you surely noticed the problem the usual MITScheme printer has with them. To avoid such printing problems, define a procedure `make-acyclic` which takes a possibly circular list structure and returns a copy of the structure omitting just enough edges to break all the cycles, so the usual printer can handle it. (The marking/searching techniques in this problem set directly adapt to this problem.)

Turn in this sheet along with your answers to the questions in the problem set:

How much time did you spend on this homework assignment?

Indicate all of the following facilities you in doing this problem set:

- The 6.001 lab in building 36.
- MIT Scheme on Project Athena.
- MIT Scheme on a Windows PC.
- EdScheme on a PC.
- Some other Scheme system (which one?).

We encourage you to work with others on problem sets as long as you acknowledge it (see the 6.001 General Information handout).

If you cooperated with other students, LA's, or others, or found portions of your answers for this problem set in references other than the course text (such as some of the course archives), please indicate your consultants' names and your references in the space below. Also, explicitly label all text and code you are submitting which is the same as that being submitted by one of your collaborators.

Otherwise, write "I worked alone using only the course reference materials." and sign your statement.