KOÇ UNIVERSITY
Department of Computer Engineering
Comp200 Structure and Interpretation of Computer Programs
Fall 2013

Project 3: Graphs

Due date: 29th November, 2015

- How to Obtain Dependencies and Main Script
    - Download your template project3.scm from: *DriveF@VOL/COURSES/UGRADS/COMP200/SHARE/[username]/project3/.*

- How to Submit Your Work
    - Please do the necessary editing on your project3.scm and make sure your file loads and runs without errors by using RUN (DrRacket) command. **If your code generates syntactic error, it will not be graded.**
    - Please rename your project3.scm file to project3_solution.scm and upload it to the folder *DriveF@VOL/COURSES/UGRADS/COMP200/HOMEWORK/[username]/project3/.*
    - You should submit only a single .scm file. No .docx, partial files or else. If you want to put a note, use comments in .scm.
    - No late submissions will be accepted

- Language
    - While using DrRacket please select the <u>pretty big</u> option from the language toolbar.
- Help
    - Please first try to understand the provided codes in the Project 3 and search **Google for your simple questions**. If you have further questions you can email to comp200help@ku.edu.tr.
- Grading
  Your work will be graded according to the following points:
    - **Run**: Your code should run without any syntactic errors.
    - **Correctness**: Your code should work correctly (No semantic errors).
    - **Clarity**: Your explanations, transcripts, and code should be simple, well-structured, well-presented, and clear.
    - **Completeness**: our code should be complete and should not require any other dependencies apart from given ones.
    - **Readability**: Your code should be readable

      "Software readability is a property that influences how easily a given piece of code can be read and understood. Since readability can affect maintainability, quality etc., programmers are very concerned about the readability of code."
      -D. Posnett, A. Hindle, P. Devanbu

1 Directed Graphs

The essence of the Web, for the purpose of understanding the process of searching, is captured by a formal abstraction called a directed graph. A graph (like the one in Figure 1), consists of nodes and edges. In this figure, the nodes are labelled U through Z. Nodes are connected to other nodes via edges. In a directed graph, each edge has a direction so that the existence of an outgoing edge from one node to another node (e.g. node X to node Y) does not imply that there is an edge in the reverse direction (e.g. from node Y to node X). Notice that there can be multiple outgoing edges from a node as well as multiple incoming edges to a node, e.g. there are edges from both Y and Z to W. The set of nodes reachable via a single outgoing edge from a given node is referred to as the node's children. For example, the children of node W are nodes U and X. Lastly, a graph is said to contain a cycle if you start from some node and manage to return to that same node after traversing one or more edges. So for example, the nodes W, X and Y form a cycle, as does the node V by itself.

A second example of a directed graph is shown in Figure 2. This particular directed graph happens to be a tree: each node is pointed to by only one other node and thus there is no sharing of nodes, and there are no cycles (or loops).

In order to traverse a directed graph, let's assume that we have two selectors for getting information from the graph:

- (find-node-children graph node) returns a list of the nodes in graph that can be reached in one step by outbound edges from node. For example, in Figure 2 the children of node B are C, D, E, and H – things that can be reached in one hop by an outgoing edge.
- (find-node-contents graph node) returns the contents of the node. For example, when we represent the web as a graph, we will want the node contents to be an alphabetized list of all of the words occurring in the document at node.

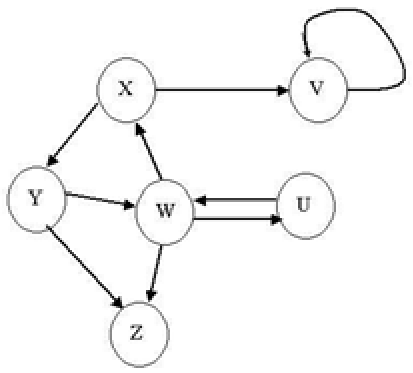

Figure 1: An example of a general graph.

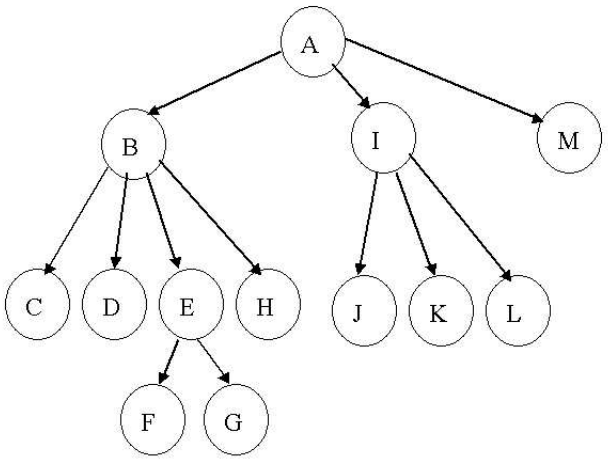

Figure 2: An example of a tree, viewed as a directed graph.

Note, we have not said anything yet about the actual representation of a graph, a node, or an edge. We are simply stating an abstract definition of a data structure.

1.1 Directed Graph Abstraction

We will build a graph abstraction to capture the relationships as shown in Figures 1 and 2, as well as enable us to have some contents at each node. You should study the code in search.scm provided with the project very closely; parts of it are described in the following discussion.

We will assume that our graph is represented as a collection of graph-elements. Each graph-element will itself consist of a node (represented as a symbol – the name of the node), a list of children nodes, and some contents stored at the node (which in general can be of any type). The constructors, type predicate, and accessors for the graph-element abstraction are shown below:

```
;;; Graph Abstraction
;;;
;;;    Graph                          a collection of Graph-Elements
;;;    Graph-Element                    a node, outgoing children from the
;;;                                     node, and contents of the node
;;;     Node = symbol                       a symbol label or name for the
node
;;;    Contents = anytype             the contents of the node

;;---------------
;; Graph-Element

; make-graph-element: Node,list<Node>,Contents -> Element
(define (make-graph-element node children contents)
  (list 'graph-element node children contents))

(define (graph-element? element)              ; anytype -> boolean
  (and (pair? element) (eq? 'graph-element (car element))))

; Get the node (the name) from the Graph-Element
(define (graph-element->node element)        ; Graph-Element -> Node
  (if (not (graph-element? element))
      (error "object not element: " element)
      (first (cdr element))))

; Get the children (a list of outgoing node names) from the Graph-Element
(define (graph-element->children element)   ; Graph-Element -> list<Node>
  (if (not (graph-element? element))
      (error "object not element: " element)
      (second (cdr element))))

; Get the contents from the Graph-Element
(define (graph-element->contents element)    ; Graph-Element -> Contents
  (if (not (graph-element? element))
      (error "object not element: " element)
      (third (cdr element))))
```

Given this representation for a graph-element, we can build the graph out of these elements as follows:

```
(define (make-graph elements)               ; list<Element> -> Graph
  (cons 'graph elements))

(define (graph? graph)                      ; anytype -> boolean
  (and (pair? graph) (eq? 'graph (car graph))))

(define (graph-elements graph)              ; Graph -> list<Graph-Element>
  (if (not (graph? graph))
      (error "object not a graph: " graph)
      (cdr graph)))

(define (graph-root graph)                  ; Graph -> Node|null
  (let ((elements (graph-elements graph)))
    (if (null? elements)
        #f
        (graph-element->node (car elements)))))
```

In the above implementation, we will arbitrarily consider the first graph-element to hold the "root" for the graph. The procedure graph-root returns the root node.

Given these abstractions, we can construct the graph in Figure 2 (with node a as the root) using:

```
(define test-graph
  (make-graph (list
    (make-graph-element 'a '(b i m) '(some words))
    (make-graph-element 'b '(c d e h) '(more words))
    (make-graph-element 'c '() '(at c node some words))
    (make-graph-element 'd '() '())
    (make-graph-element 'e '(f g) '(and even more words))
    (make-graph-element 'f '() '())
    (make-graph-element 'g '() '())
    (make-graph-element 'h '() '())
    (make-graph-element 'i '(j k l) '(more words yet))
    (make-graph-element 'j '() '())
    (make-graph-element 'k '() '())
    (make-graph-element 'l '() '()))))
```

Note that several of the nodes have no children, and that several have no contents.

We would like to have some accessors to get connectivity and contents information out of the graph. We first define a procedure to find a graph-element in a graph, given the node (i.e. the symbol or name that identifies the element):

```
; Find the specified node in the
graph
(define (find-graph-element graph node)    ; Graph,Node -> Graph-Element|null
  (define (find elements)
    (cond ((null? elements) '())
          ((eq? (graph-element->node (car elements)) node)
           (car elements))
          (else (find (cdr elements)))))
  (find (graph-elements graph)))
```

We are often more interested in the node children or node contents, rather than the graph-element. The find-node-children and find-node-contents accessor procedures can be implemented as follows:

```
; Find the children of the specified node in the graph
(define (find-node-children graph node)         ; Graph,Node -> list<Node>|null
  (let ((element (find-graph-element graph node)))
    (if (not (null? element))
        (graph-element->children element)
        '())))

; Find the contents of the specified node in the
graph
(define (find-node-contents graph node)         ; Graph,Node -> contents|null
  (let ((element (find-graph-element graph node)))
    (if (not (null? element))
        (graph-element->contents element)
        '())))
```

In our representation above, we use node names (Node = symbol) to reference a graph-element in a graph; the children of a node are represented as a list of other node names. An alternative to this approach would be to make the node itself a full abstract data type, so that a node object would have identity, and the children of a node could be, for example, a list of the actual children node objects. The tradeoff would be more work in building the graph (e.g. to link together actual node objects as nodes and edges are added to a graph), but substantial savings when nodes are requested from the graph (i.e. by avoiding a linear search of the graph-elements for the matching node name). With such an alternative abstraction, when requesting a child node one can achieve constant time access (in the size of the graph), as opposed to linear time access as in the current implementation.

2 Searching a Graph

How can we search a graph? The basic idea is that we need to start at some node and traverse the graph in some fashion looking for some goal. The search might succeed (meaning that some goal is found), or it might fail (meaning that some goal was not found). This very basic and abstract search behavior can be captured in the following procedure:

```
;; search: Node, (Node->Boolean), (Graph, Node -> List<Node>),
;;           (List<Node>, List<Node> -> List<Node>), Graph
;;             --> Boolean

(define (search initial-state goal? successors merge graph)
  ;; initial-state is the start state of the search
  ;;
  ;; goal? is the predicate that determines whether we have
  ;; reached the goal
  ;;
  ;; successors computes from the current state all successor states
  ;;
  ;; merge combines new states with the set of states still to explore
  (define (search-inner still-to-do)
    (if (null? still-to-do)
        #f
        (let ((current (car still-to-do)))
          (if *search-debug*
              (write-line (list 'now-at current)))
          (if (goal? current)
              #t
              (search-inner
               (merge (successors graph current) (cdr still-to-do)))))))
  (search-inner (list initial-state)))
```

Note the use of the *search-debug* flag. If we set this global variable to #t, we will see the order in which the procedure is traversing the graph.

2.1 Looking at search

What does this search procedure do? Well, let's look at it a bit more closely. Search takes several arguments. The first is the initial state of the search. For our purposes, this will be a Node or in other words, the name of some node in our graph. The second is a goal? procedure, which is applied to a node to determine if we have reached our goal. This procedure will presumably examine some aspect of the node (for example, maybe it wants to see if a particular word is contained in the contents of that node) to decide if the search has reached its termination point. The third is a procedure for finding the successors of the node, which in this case basically means finding the children of a node in the graph on which we are searching. The fourth is a procedure for combining the children of a node with any other nodes that we still have to search. And the final argument is the graph over which we are searching.

Looking at the code, you can see that we start with a list of nodes to search. If the first one meets our goal? criterion, we stop. If not, we get the children of the current node, and combine them in some fashion with the other nodes in our collection to search. This then becomes our new list of nodes to consider, and we continue.

2.2 Search Strategies

There are two common approaches for searching directed graphs, called depth-first search and breadth-first search. In a depth-first search we start at a node, pick one of the outgoing links from it, explore that link (and all of that link's outgoing links, and so on) before returning to explore the next link out of our original node. For the graph in Figure 2, that would mean we would examine the nodes (if we go left-to-right as well as depth-first) in the order: a, b, c, d, e, f, g, h, i, j, k, l, and finally m (unless we found our goal earlier, of course). The name “depth-first” comes from the fact that we go down the graph (in the above drawing) before we go across.

In a breadth-first search, we visit a node and then all of its “siblings” first, before exploring any “children.” For Figure 2, we'd visit the nodes in the order a, b, i, m, c, d, e, h, j, k, l, f, g.

We can abstract the notions of depth-first, breadth-first, and other kinds of searches using the idea of a search strategy. A search strategy will basically come down to what choice we make for how to order the nodes to be explored.

2.3 A Depth-First Strategy

Here's an initial attempt at a depth-first search strategy. It doesn't quite work on all cases, but it's a good place to start.

```
(define (DFS-simple start goal? graph)
  (search start
          goal?
          find-node-children
          (lambda (new old) (append new old))
          graph))
```

And here is an example of using it

```
(DFS-simple 'a
        (lambda (node) (eq? node '1))
        test-graph)
```

In this case, we are searching a particular graph test-graph, starting from a node with name a. We are looking for a node named 1 (hence our second argument). We use our graph abstraction to extract the children of a node (i.e. find-node-children). The key element is how we choose to order the set of nodes to be explored. Note the fourth argument. We can see that this will basically take the list of nodes to be explored, and add the new children to the end of the list. This should give us a depth first search (you should think carefully about why).

This simple method does not work in general, but it does work for the graph in Figure 2.

3 An Index Abstraction

We will also be interested in constructing an index. To do this, we construct a general purpose index abstraction.

An Index enables us to associate values with keys, and to retrieve those values later on given the key. Here we will assume that a key is a Scheme symbol (i.e. Key = symbol), and that a value is also a symbol (i.e. Val = symbol). Our index will be a mutable data structure.

A concrete implementation for an index is as follows. An Index will be a tagged data object that holds a list of Index-Entries. Each Index-Entry associates a Key with a list of values for that Key, i.e.

```
;;    Index = Pair<Index-Tag, list<Index-Entry>>
;;    Index-Entry = Pair<Key, Pair<list<Val>, null>>
```

Thus our index implementation is shown (partially) below. You will be asked in the exercises to complete the implementation. The index implementation makes use of the Scheme procedure assv; you will find it helpful to consult the Scheme manual as to what this procedure does.

```
(define (make-index)              ; void -> Index
  (list 'index))

(define (index? index)            ; anytype -> boolean
  (and (pair? index) (eq? 'index (car index))))
```

```
; This is an internal helper procedure not to be used externally.
(define (find-entry-in-index index key)
  (if (not (index? index))
      (error "object not an index: " index)
      (assv key (cdr index))))

; returns a list of values associated with key
(define (find-in-index index key)        ; Index,Key -> list<Val>
  (let ((index-entry (find-entry-in-index index key)))
    (if (not (null? index-entry))
        (cadr index-entry)
        '())))

(define (add-to-index! index key value) ; Index,Key,Val -> Index
  (let ((index-entry (find-entry-in-index index key)))
    (if (null? index-entry)
        ;; no entry -- create and insert a new one...
        ... TO BE IMPLEMENTED
        ;; entry exists -- insert value if not already there...
        ... TO BE IMPLEMENTED
        ))
  index)
```

4 Programming assignment

In the project, you will be dealing with a graph data structure. Open up your project-3.scm file and read the instructions and start answering it.

Exercise 0 & 1: You should write down your answers as comment lines inside of the file.

You have the implementation of DFS. Just to make sure everything is working, evaluate

```
(DFS-simple 'a
        (lambda (node) (eq? node 'l))
        test-graph)
```

This should traverse the test-graph graph until the search finds node l (lowercase L), and you should see the nodes being visited in depth-first order.

Exercise 2: A breadth-first search.

Our previous example used a depth-first strategy. A breadth-first search strategy can be obtained by modifying only one line of the DFS-simple, leaving the total number of characters in the expression unchanged! Do this to create a new procedure (call it BFS-simple), demonstrate that it works on test-graph, and write a short (but clear) explanation of why it works.

Exercise 3: The Index Abstraction.

Your task is to complete the implementation of the index abstraction. Complete the definition of add-to-index! so that we have available the following procedures:

- (add-to-index! index key value): Add the value under the given key in the index. Make sure that you prevent the addition of null values.

Verify that your add-to-index! works with the test cases examples.

Exercise 4: Marking visited nodes.

In Exercise 1, you discussed a problem with DFS-simple. One way to fix this problem is to keep track of what nodes you have visited. The basic idea is that when we move to a "new" node, we can check to see if we have already examined that node. If we have, we can simply remove it from our list of nodes to explore, ignore any children (since they will have also already been visited), and move on to the next node in the list.

You should be able to use the definition of search as a starting point to create a new procedure (call it search-with-cycles) that keeps track of already visited nodes, and implements the idea described above.

To show that your implementation works, use it with a depth first strategy to create a new procedure (call it DFS_cycled) that implements full depth first search. Use it to walk the sample graph test-cycle which is defined for you in search.scm. Show that it visits nodes at most once. Also create a breadth first search (call it BFS_cycled), and also show that it only visits nodes once (albeit in a different order).

Once you are sure these procedures are working, give the order in which the nodes are visited for depth-first search and for breadth-first search.