MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 1998

Problem Set 6

- Issued: Tuesday, March 10
- Tutorial preparation for: Week of March 16
- Written solutions due: Friday, March 20 in recitation
- Reading: Read sections 3.1 and 3.3.1 before lecture on March 12. Read section 3.2 before lecture on March 17.

Searching the World Wide Web

This problem set explores some issues that arise in constructing a "spider" or an "autonomous agent" that crawls over the documents in the World Wide Web. For our purposes, the Web is a very large (and growing!) collection of documents. Each document contains some text and also links to other documents, in the form of URLs.

In this problem set we'll be working with programs that can start with an initial document and follow the references to other documents to do useful things. For example, we could construct an index of all the words occurring in documents, and make this available to people looking for information on the web (as in Digital Equipment Corporation's AltaVista system at http://AltaVista.digital.com).

Just in case you aren't fluent with the details of HTTP, URLs, URIs, HTML, XML, XSL, HTTP-NG, DOM, and the rest of the alphabet soup that makes up the technical details of the Web, here's a simplified version of what goes on behind the scenes:

1. The Web consists of a very large number of things called documents, identified by names called URLs. For example, the 6.001 home page has the URL `http://mit.edu/6.001`. The URL contains the name of a protocol (HTTP in this case) that can be used to fetch the document, as well as the information needed by the protocol to specify which document is intended (`mit.edu/6.001` in this case).
2. By using the HTTP protocol, a program (most commonly a browser but any program can do this—and "autonomous agents" and spiders are examples of such programs that aren't browsers) can retrieve a document whose URL starts with `HTTP:`. The document is returned to the program, along with information about how it is encoded, for example, ASCII or Unicode text, HTML, images in GIF or JPG or MPEG or PNG or some other format, an Excel or Lotus spreadsheet, etc.
3. Documents encoded in HTML form can contain a mixture of text, images, formatting information, and links to other documents. Thus, when a browser (or other program) gets an HTML document it can extract the links from it, yielding URLs for other documents in the Web. If these are in HTML format, then they can be retrieved and will yield yet more links, and so on.

4. A *spider* is a program that starts with an initial set of URLs, retrieves the corresponding documents, adds the links from these documents to the set of URLs and keeps on going. Every time it retrieves a document, it does some (hopefully useful) work in addition to just finding the embedded links.

5. One particularly interesting kind of spider constructs an *index* of the documents it has seen. This index is similar to the index at the end of a book: it has certain key words and phrases, and for each entry it lists all of the URLs that contain that word or phrase. There are many kinds of indexes, and the art/science of deciding what words or phrases to index and how to extract them is at the cutting edge of research (it's part of the discipline called *information retrieval*). We'll talk about *full text indexing*, which means that every word in the document (except, perhaps, the very common words like "and," "the," "a," and "an") is indexed.

1. Searching a directed graph

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 consists of *nodes*. One node may be connected to other nodes through *edges*. In a directed graph, such as the Web, the edges point in a given direction – that we can go from a node through an *outgoing edge* to another node does not imply that there is an edge in the reverse direction. Thus, an HTML document might be a node in a graph (represented by its URL) and from each such node there would be edges corresponding to the links in that document to other documents.

For example, a collection of web documents might be organized as shown in figure 1 (where each lettered node indicates a URL). This particular directed graph happens to be a tree (each node is pointed to by only one other node) but this need not be true in general.

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

- `((node->neighbors` *graph*`)` *node*`)` returns a list of the nodes that can be reached by outbound edges from *node*.
- `((node->text` *graph*`)` *node*`)` returns an alphabetized list of all of the words occurring in the document at *node*.

Notice that `node->neighbors` and `node->text` are procedures that take a graph as argument and return the appropriate selector to apply to a node in the graph. On the Web, `(node->neighbors` *graph*`)` would produce a procedure that takes as input a URL (the node) and produces as output a list of URLs that are the links inside the document. This would involve retrieving the document using its URL, parsing the HTML and extracting the information from `<a HREF=...>`, `<image src=...>` and similar tags. Similarly, `(node->text` *graph*`)` would produce a procedure that retrieves the document, discards all of the mark-up commands, alphabetizes (and removes duplicates) from the text, and returns the resulting list of words.

For this problem set we'll begin with something simpler. We assume that our graph is represented as a list structure. The graph will be a list of entries, where each entry signifies a node. Each entry

```
                            A
                            |
        -----------------------------
       /                    |        \
      v                     v         v
      B                     I         M
      |                     |
   -------                 ---
  /  / \  \               / | \
 v  v  v   v             v  v  v
C   D  E   H            J   K   L
       |
       --
      /  \
      v   v
      F   G
```

Figure 1: A collection of linked web documents, viewed as a graph.

will itself be list, consisting of a node (represented as a symbol—the name of the node), a list of neighboring nodes, and some text stored at the node (represented as a list of symbols).

```
(define (make-graph-entry node neighbors text)
  (list node neighbors text))

(define (graph-entry->node entry) (first entry))
(define (graph-entry->neighbors entry) (second entry))
(define (graph-entry->text entry) (third entry))
```

For example, we could construct the graph in figure 1 as

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

Note that several of the nodes have no neighbors, and that several have no text.

Given this representation, we can generate the procedures to use for obtaining the node neighbors and node text:

```
(define (node->neighbors graph)
  (lambda (node)
    (let ((entry (assq node graph)))
      (if entry
          (graph-entry->neighbors entry)
          '()))))

(define (node->text graph)
  (lambda (node)
    (let ((entry (assq node graph)))
      (if entry
          (graph-entry->text entry)
          '()))))
```

How can we search this graph? Let's assume that we have some way to recognize when to stop searching. We'll represent this by a predicate `goal?` that takes a node as argument and tests whether the node is the goal of the search. The `goal?` procedure might examine the text at the node, or the name of the node, it might just say whether the searcher has run out of time, or it might just never return true (so the searcher will visit the entire graph), or whatever.

There are two common strategies for searching, 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 1, 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 1, 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 is a procedure that determines the order in which to visit the nodes of the graph. The procedure takes a node as argument and returns one of three values:

- returns true if the node is a goal
- returns false if there are no more nodes to visit
- otherwise returns the next node to visit

Rather than writing strategies directly, we'll write some procedures that create strategies. A *strategy-maker* is a procedure that returns a strategy procedure. The strategy-maker procedure takes two arguments:

- a `goal?` procedure
- a `neighbors` procedure that, when applied to a node, produces the neighboring nodes (i.e., the nodes one can reach by following the outbound edges from that node)

For example, we'll have a procedure `make-df-strategy` that makes the depth-first (DF) strategy, that will behave as follows:

`((make-df-strategy` *goal?-procedure neighbors-procedure*`)` *node*`)`

will return true if *node* is a goal, false if there are no more nodes to visit, and otherwise return the next node to examine after *node*.

Given that we can implement strategy makers, the following procedure is an effective mechanism for searching any finite graph. Note the use of the `*debugging*` flag, that we can set to true if we want to record the order in which the procedure is traversing the graph:

```
(define (search graph strategy-maker start-node goal?)
  (if *debugging*
      (write-line (list 'start start-node)))
  (let ((searcher
         (strategy-maker goal? (node->neighbors graph))))
    (define (loop node)
      (let ((next-node (searcher node)))
        (cond ((eq? next-node #T) 'FOUND)
              ((eq? next-node #F) 'NOT-FOUND)
              (else
               (if *debugging*
                   (write-line (list 'from node 'to next-node)))
               (loop next-node)))))
    (loop start-node)))
```

The hard part, of course, is writing a strategy-maker. The strategy procedure that it creates should take a node as input and return the next node to be examined. Notice that the searcher is not a mathematical function: if a node has several neighbors, then the strategy must return a different `next-node` each time it is called with that input node. This is a tip-off that there must be side-effects in the definition of the strategy procedure. In fact, we'll see that there are two different places where side-effects occur.

Here's an initial attempt at a depth-first strategy-maker. It doesn't quite work, but it's a good place to start. The idea is that it maintains (via side-effects to the variable `*to-be-visited*`) a list of nodes that need to be visited, in the order in which they should be visited. Whenever a new node is visited, all of its neighbors are added to `*to-be-visited*`. When `*to-be-visited*` is empty, we've visited all the nodes we can get to, so we give up:

```
(define (make-df-strategy-1 goal? neighbors)
  (let ((*to-be-visited* '()))
    (define (where-next? here)
      (set! *to-be-visited*
```

```
          (append (neighbors here) *to-be-visited*))
    (cond ((goal? here) #T)
          ((null? *to-be-visited*) #F)
          (else
           (let ((next (car *to-be-visited*)))
             (set! *to-be-visited* (cdr *to-be-visited*))
             next))))
  where-next?))
```

This simple algorithm does not work in general (see tutorial exercise 3), but it does work for the graph in figure 1.

2. Tutorial Exercises

Tutorial exercise 1: Do exercise 3.2 of the textbook.

Tutorial exercise 2: The following variation of `make-df-strategy-1` has a bug that causes it to not work, even on the `test-data`. Explain what is wrong. In particular, why does it matter where `*to-be-visited*` is declared?

```
(define (make-df-strategy-2 starting-point goal? neighbors)
  (define (where-next? here)
    (let ((*to-be-visited* '()))         ; Nodes to look at
      (set! *to-be-visited*              ; Add on new nodes
            (append (neighbors here) *to-be-visited*))
      (cond ((goal? here) #T)            ;  We win!
            ((null? *to-be-visited*) #F) ; Nowhere left to look
            (else                        ; Visit next node
             (let ((next (car *to-be-visited*)))
               (set! *to-be-visited* (cdr *to-be-visited*))
               next))))
    where-next?)))
```

Tutorial exercise 3: The following partial definition mimics the graph of web pages at the 6.001 web site. Each node here is the URL of a web page and the neighbor nodes are the URLs referenced in the links on the page.

```
(define web
  (list
   (make-graph-entry
    'http://mit.edu/6.001
    '(http://mit.edu/6.001
      http://mit.edu/6.001/SchemeImplementations
      http://mit.edu/6.001/PSets)
    '(... words extracted from http://mit.edu/6.001 ...))
   (make-graph-entry
    'http://mit.edu/6.001/SchemeImplementations
```

```
   (http://mit.edu/6.001/getting-help
    http://mit.edu/6.001/lab-use
    *the-goal*)
   '(... words extracted from http://mit.edu/6.001/SchemeImplementations ...))
  (make-graph-entry
   'http://mit.edu/6.001/getting-help
   '(http://mit.edu/6.001
     http://mit.edu/6.001/SchemeImplementations)
   '(... words extracted from http://mit.edu/6.001/getting-help))
  ...))
```

Demonstrate that `make-df-strategy-1` fails on this graph. What is the essential difference between the `test-data` and `web` examples that causes `make-df-strategy-1` to fail here?

3. Programming assignment: A web spider

Begin by loading the code for problem set 6. This will define the search and data structure procedures listed above. Just to make sure everything is working, evaluate

```
(search test-data make-df-strategy-1 'a
        (lambda (node) (eq? node 'l)))
```

This should search the `test-data` graph until the searcher finds node `l`, and you should see the nodes visited in depth-first order.

Computer exercise 1: `Make-df-strategy-1` creates a depth-first search strategy procedure. A breadth-first search strategy can be obtained by modifying *only one line* of the procedure `make-df-strategy-1`, leaving the total number of characters in the procedure unchanged! Do this (name your procedure `make-bf-strategy-1`), demonstrate that it works on `test-data`, and write a short (but clear) explanation of why it works.

Marking nodes

In tutorial exercise 3, you discussed a problem with `make-df-strategy-1`: it doesn't prevent visiting a node that has already been examined, and can therefore go into an infinite loop. As you saw in lecture on Thursday, March 12, we we can fix the problem by using a pair of procedures:

- `(note-visited!` *node*`)`: remembers that we've visited *node*
- `(deja-vu?` *node*`)`: test whether or not we've visited *node*

The implementations of the search strategy-maker procedures shown in lecture all use a global variable to store their state. If we want to be able to run several searches concurrently (for example searching for different goals or using different strategies to see which finishes first) then it's important that we produce a new pair of procedures for each strategy procedure. The following version of a depth-first strategy-maker does this:

```
(define (make-df-strategy goal? neighbors)
  (let ((mark-procedures (make-mark-procedures)))
    (let ((deja-vu? (car mark-procedures))
          (note-visited! (cadr mark-procedures))
          (*to-be-visited* '()))
      (define (try-node candidates)
        (cond ((null? candidates) #F)
              ((deja-vu? (car candidates))
               (try-node (cdr candidates)))
              (else
               (set! *to-be-visited* (cdr candidates))
               (car candidates))))
      (define (where-next? here)
        (note-visited! here)
        (set! *to-be-visited*
              (append (neighbors here) *to-be-visited*))
        (if (goal? here)
            #T
            (try-node *to-be-visited*)))
      where-next?)))
```

Note the procedure used here: `(make-mark-procedures)` returns a list whose first element is a procedure that works like `deja-vu?` and whose second element is a procedure that works like `note-visited!`.

Computer exercise 2: Implement `make-mark-procedures`. The procedure should be very short (less than a dozen lines of code), and you can make use of the built-in Scheme procedure `memq`. Do not worry about efficiency. We're concerned here only with the correctness of the program and with making sure that you understand how to introduce the local state necessary to make the procedures work correctly.

To show that your implementation of `make-mark-procedures` works, use `search` and `make-df-strategy` to walk sample graph `web`. Also modify your breadth-first search strategy maker from computer exercise 1 to create procedure `make-bf-strategy`, that makes sure that it visits nodes only once. Demonstrate that this procedure works by showing that it can traverse `web`. Give the order in which the nodes are visited for depth-first search and for breadth-first search.

Indexing the web

Now let's turn to the problem of creating a full-text index of documents on the Web, like the one created by AltaVista. We'll assume that we have a graph that represents the World Wide Web, using node names that correspond to URLs (as in the sample `web` given earlier). Remember we're assuming that we have a procedure `node->text`, that gets us the alphabetized text at the node. For example, `((node->text web) 'http://mit.edu/6.001)` yields the list (at least it did for the real 6.001 web page on February 9, 1998):

```
(11:08 1997 1998 339-0052 34-101 6.001 6.001-WEBMASTER@AI.MIT.EDU
```

```
 7 7-9PM 9 947-2394 ABOUT ALL AM AND ARE ASSIGNMENT ASSIGNMENTS
 BY CALENDAR CAN CHANGE COLLABORATIVE COMMENTS COMPUTER COPYRIGHT
 DEBUGGING DISCUSSION DO DOCUMENTATION FEB FEBRUARY FOR FORUM
 GENERAL GET GETTING GUIDELINES HELP HOMEWORK HOW I IN INDIVIDUAL
 INFORMATION INSTITUTE INTERPRETATION IS LAST LECTURE LINE
 MASSACHUSETTS ME MODIFIED MONDAY MY OF ON ON-LINE ORAL OWN PM
 POLICY PRESENTATIONS PREVIOUS PROBLEM PROGRAMS RECITATION
 RECITATIONS RECORDS REMINDER RESERVED RIGHTS SCHEME SECTION
 SECTIONS SEND SET SITE SOFTWARE SPECIAL SPRING STAFF STRUCTURE
 SUBJECT TECHNOLOGY TELL TERMS THE THIS TO UP USE WEEKS WHAT
 WHERE WHICH WORK WRITING)
```

Let's create a data abstraction for the index. It consists of three procedures:

- `(initialize-index!)`: Clears any previous information from the index.
- `(add-to-index! url list-of-words)`: Add all of the words to the index, together with some information to indicate that they occur in this URL.
- `(find-in-index word)`: Returns a list of all the URLs that have been entered into the index as associated with that word.

Here's an example of how it should work:

```
(define (index-document! web)
  (let ((get-text (node->text web)))
    (lambda (url)
     ;; Fetch the text of the URL and insert it into index
     (add-to-index! url (get-text url)))))

(initialize-index!)
((index-document! web)
 'http://mit.edu/6.001)  ; Index one document
(find-in-index 'help)    ; What documents have "help" in them?
   ==> (http://mit.edu/6.001)
(find-in-index '*magic*) ; Which documents have "*magic*" in them?
   ==> #F
```

Computer exercise 3: Write the three procedures defined above. Notice that you will need a variable shared by all three procedures. Make sure that this variable is visible to only these three procedures and not to any other Scheme procedure. (Hint: Your program won't be just three top-level definitions.)

Note: A professionally written version of these procedures would pay careful attention to the efficiency of the algorithm used, and would probably involve alphabetical order and complicated data structures. You should *not* worry about this unless you find yourself with spare time on your hands — and if you do have that time, you might want to look at the Scheme reference manual definition of `sort`, `string<?`, and `symbol->string`. Your tutor will thank you for making your solution easy to understand, even at the expense of performance (and, in later life, many of your colleagues will feel the same way!).

Computer exercise 4: Now let's simulate what AltaVista's spider does: Crawl the entire web (use a goal procedure that always returns false) and produce a full-text index of everything you find. Write a procedure, `make-web-index`, similar to the procedure `search`, that initializes the index, finds all the URLs that can be reached from a given initial URL, indexes them, and returns a procedure that can be used to find all the URLs of documents containing a given word. Your procedure should use `make-bf-strategy` from computer exercise 2 to actually crawl the web.

You can test your program by trying the following example. Which document(s) do you find?

```
(define find-documents
  (make-web-index web 'http://mit.edu/6.001))

(find-documents 'collaborative)
```

Computer exercise 5: Let's put everything together by comparing the performance of crawling the web versus using a full-text index of the web. This won't be a full or fair comparison, but it should give you some ideas about tradeoffs in designing real systems that analyze the contents of the actual Web. To investigate crawling, write two procedures:

1. `(search-any web start-node word)` searches the web (using `make-bf-strategy`) and returns the *first* document that it finds that contains the given word. It should stop searching as soon as it finds such a document.
2. `(search-all web start-node word)` searches the *entire* web (using `make-bf-strategy`) and returns *all* documents that contain the given word.

We'll compare this with the technique of indexing the web using `make-web-index` and `find-documents`.

We've provided a program, `(generate-random-web` *size*`)`, that you can use to create webs of different sizes (total number of nodes) with some randomly generated text. Use this to build several webs. You don't have to make them too big; the procedure will in fact not build anything larger than size 200. For each web, measure the amount of time it takes, starting from the node named `*start*`:

- to use `search-any` to find a document containing the word "help";
- to use `search-any` to find a document containing a word that is not in the web: "Chuckvest";
- to use `search-all` to find all documents containing the word "help";
- to run `make-web-index` to create an index for the web;
- to use `find-documents` to find all documents containing the word "help", not including the time needed to create the index.
- to use `find-documents` to find all documents containing the word "Chuckvest" (there won't be any), not including the time needed to create the index.

To enable you to do timing, we've included the same `timed` procedure that you used in problem set 2.

Write a few short paragraphs explaining the measurements you made and what conclusions you might want to draw about searching and crawling the real Web. If you were building a service to help people find information on the Web, what kinds of factors would you consider in deciding which method to use?

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

1. About how much time did you spend on this homework assignment? (Reading and preparing the assignment plus computer work.)
2. Which scheme system(s) did you use to do this assignment (for example: 6.001 lab, your own NT machine, your own Win95 machine, your own Linux machine)?
3. 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 text (such as some of the archives), please indicate your consultants' names and your references. 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 reference materials," and sign your statement.