Tutorial 11: Harvesting E-Mail Addresses and Other Evil Things

CS 135 Fall 2007

November 21-23, 2007

Today's tutorial is all about generative recursion. We'll work with strings, graphs, and numbers. The starter code is here (t11-starter.scm). All the material is from lecture module 11.

1 Spam bots

A spam bot is a computer program that crawls the internet looking for email addresses to add to a spam list. One crucial step is extracting (or 'harvesting') email addresses from some text with lots of other stuff in it (usually an html file). Note that most spam bots choose speed over thoroughness, because there are so many email addresses published freely that even a very stupid harvester can be quite effective. This is nice because it means that spam bots are usually easy to fool.

Write a simple email harvester called `harvest-emails`. You Scheme function should consume a single string and produce a list of strings corresponding to the email addresses found. Use these rules to determine what is an email address:

- An email address looks like `name@domain.suffix`, where each of `name`, `domain`, and `suffix` consist only of letters and numbers.
- The `suffix` is never more than 3 characters long.
- The characters surrounding the email address (that is, immediately preceding `name` and following `suffix`), if there are any, are not letters or numbers.

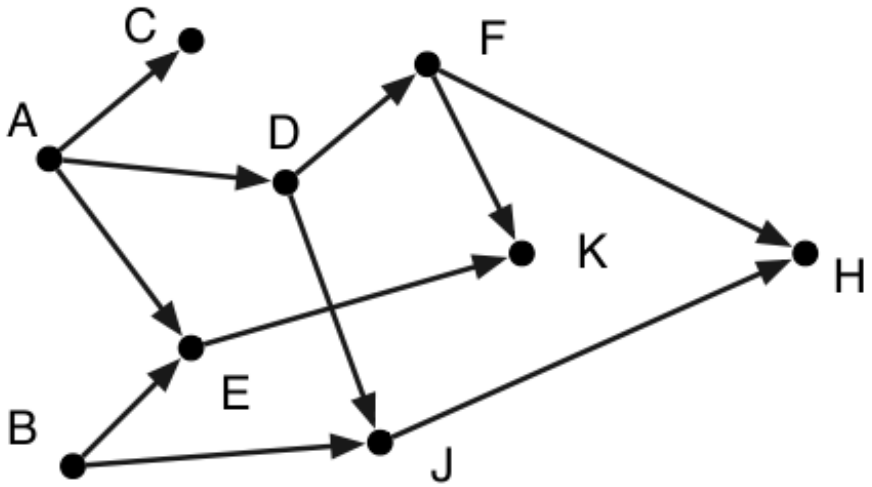
To make this problem a bit shorter, you've been provided with two helper functions in the starter code (t11-starter.scm). `after` consumes two lists such that the first list is a prefix of the second — that is, the second list starts with a copy of the first one. The function produces the sublist of the second list starting after the first list is over. So for example, `(after '(1 2) '(1 2 3 4 5))` returns `'(3 4 5)`.

The function `get-alphanumlist` consumes a list of characters and produces the longest prefix which is all letters or numbers. See the starter code for some examples of use.

2 Viruses in a network

Directed acyclic graphs (DAGs) can be used as a very simplistic model for the flow of information in a network. Each node in the graph represents a single computer in the network, and each line represents a one-way communication connection. The following questions focus on what happens when a single computer in the network is infected with a virus.

The code for the `neighbours` function (from class) is provided in the starter code. For testing, we'll just use the same graph from lecture module 11, since I'm too lazy to make another picture. The code for this is in the starter code as constant `slides-graph`, and a graphical representation is copied below:



2.1 Shortest path to a client

The most insecure computers in a network are almost always the client machines. These can be recognized by having no out neighbors. In graph theory, these kind of nodes are called 'sink nodes'.

So a virus might be more likely to take hold if it is close to some client nodes. Write a function `shortest-to-sink` which consumes the name of a node and a graph and prodces the number of edges on the shortest path from the named node to a client (i.e. sink node).

2.2 Two paths diverged...

A virus will have a greater chance of spreading to a given node if there are multiple (different) paths from the source to the target node. We'll say two paths are distinct as long as they are not completely identical (so they can share some nodes).

Write a function `multiple-routes?` that consumes two labes for the names of the source and destination, and a graph, and produces true iff there are at leats two distinct paths from the source to the destination in the graph.

3 When will it end?

We have seen in class that, when dealing with generative recursion, sometimes it is very difficult to determine if the algorithm always terminates (in fact, this is a famous 'undecidable' problem — Google 'undecidable', 'Alan Turing', or 'halting problem' for more). A program that never terminates can also very easily become a security risk in a computer.

For the following programs, detemine whether they terminate on all valid imput. If so, give a brief justification or proof. If not, give an example input that will make the program run forever, and try to devise conditions on the input that will guarantee termination.

1.
```
;; bar: nat nat -> nat
(define (bar m n)
  (cond [(zero? (* m n)) (+ m n)]
        [else (local [(define q (min m n (sqr (- m n))))]
                (bar (- m q) (- n q)))]))
```

2.
```
;; f: nat -> nat
(define (f n)
  (cond [(zero? n) 0]
        [(zero? (remainder n 2)) (f (quotient n 2))]
        [else (add1 (f (* 8 (sub1 n))))]))
```

3.
```
;; g: nat nat nat -> nat
(define (g m n i)
  (cond [(zero? (remainder i n)) i]
        [else (g m n (+ i m))]))
```

4.
```
;; h: nat nat -> nat
(define (h m n)
  (cond [(< m n) (h n m)]
        [(< (+ m n) 1000)
         (h (- m n) (sqr (add1 n)))]
        [else (* m n)]))
```