Harvey Mudd College
Computer Science 60
Fall 2007

Assignment 3
Applications Using Functional Programming
Due. 11:59 p.m., Monday, 24 September 2007

All problems in this assignment are to be done in a purely functional style using the Scheme language. Submit two files: hw03.scm (your code) and hw03.txt (your tests) in a directory called HW3 in your dropbox on Sakai. As always, document your functions clearly. **Be sure to spell each function name exactly as given**, otherwise the automatic grading script might not give you credit for the function.

1. [20 points] In class you saw an efficient O(n) implementation of a function that reverses a list, `my-reverse` using an accumulator argument. It went something like this.

```
(define (my-reverse L)
  (my-reverse-aux L ()))

(define (my-reverse-help L Acc)
  (if (null? L)
      Acc
      (my-reverse-help (rest L) (cons (first L) Acc))))
```

In this case, no computational work occurs after the recursive call to `my-reverse-help`. Such functions are called *tail-recursive*. They can be implemented without adding stack-frames for each function call, which means they can be considerably faster, and more space-efficient than their non-tail-recursive counterparts.

This problem asks you to construct three tail-recursive functions – each will require you to create helper (aka auxiliary), function that uses one (or more!) accumulators.

a. [5 of 20 points] Construct a tail-recursive implementation of the `length` function, called `my-length` which returns the length of its argument, an arbitrary list.

b. [5 of 20 points] Construct a tail-recursive version of factorial, called `my-fac`. The argument is any non-negative integer.

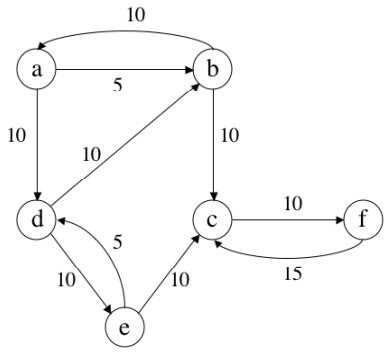
c. [10 of 20 points] Construct a tail-recursive version of a Fibonacci number generator `my-fib` that outputs the Nth element in the Fibonacci sequence: 1, 1, 2, 3, 5, 8, 13, 21, ... in which each term is the sum of the previous two.

Testing requirements: Show tests of each of the above functions that cover a broad range of cases. You do not have to document your tests, just be sure to include all reasonable cases (as we discussed in class). You can just save the contents of the output window as a txt file.

2. [60 points] Implement a function that computes the shortest paths from a specified source node to each node in a directed graph having non-negative distances associated with each arc.

For this problem, a graph is a list of nodes. Each node is a list of two elements: a *name*, which is a symbol, and an a*ssociation list* that specifies the nodes to which this node is directly connected, together with a *direct distance* for that connection. An example graph is shown below:

```
(define graph1 '(
  (a ((b  5) (d 10)))
  (b ((a 10) (c 10)))
  (c ((f 10)))
  (d ((b 10) (e 10)))
  (e ((c 10) (d 5)))
  (f ((c 15)))
  ))
```



In this graph, the nodes are {`a, b, c, d, e, f`}. There is a direct connection from `a` to `b` with a distance of 5, from `a` to `d` with a distance 10, etc. The direct distances are not necessarily symmetric, i.e. the distance from `a` to `b` could be 5, while the direct distance from `b` to `a` could be 10, as in this example. In any case, the direct distance is not necessarily the shortest directed distance from the source to other nodes. There may be paths that go through several nodes. The shortest might even be shorter than the specified direct distance.

The function `shortest-path-length` has two arguments, a source node and a graph, such as the one above. It returns the shortest distance of *each* node in the graph to the source. Moreover, the nodes are listed in increasing order from the source. (If there is a tie, the order within tied nodes is according to `string<?` applied to the names after converting the symbols to strings. Here are some examples for `graph1` above. The shortest path from `a` to `f` has length 25, for example.

```
> (shortest-path-length 'a graph1)
 ((a 0) (b 5) (d 10) (c 15) (e 20) (f 25))

> (shortest-path-length 'd graph1)
 ((d 0) (b 10) (e 10) (a 20) (c 20) (f 30))

> (shortest-path-length 'f graph1)
 ((f 0) (c 15) (a infinity) (b infinity) (d infinity)
  (e infinity))
```

A shortest path length of `infinity` indicates that the corresponding node is *not reachable* from the source.

Testing Requirements: Show the results of your function on three graphs designed to test 3 different cases. Your graphs do not have to be large. Place these tests in the same file you created to test the functions in problem 1. Annotate each test with a short sentence or a few words about why you included it.

3. [20 points] Augment your function `shortest-path-length` to construct a function `shortest-path` that it shows the length of the path, followed by the *predecessor* node on the shortest path. From this information, the path itself can be reconstructed by tracing backward from node to node.

```
> (shortest-path 'a graph1)
 ((a 0 _) (b 5 a) (d 10 a) (c 15 b) (e 20 d) (f 25 c))

> (shortest-path 'f graph1)
 ((f 0 _) (c 15 f) (a infinity _) (b infinity _)
  (d infinity _) (e infinity _))
```

The node before `f` is `c`, the node before `c` is `b`, and the node before `b` is `a`. Thus the shortest path from `a` to `f` is `a-b-c-f` with length 25. The symbol _ should be used to indicate that there is no predecessor in a shortest path, e.g. in the case of the source node or an unreachable node. Please use these definitions so that your answers will agree with ours.

```
(define infinity 'infinity)
(define no-predecessor '_)
```

You may use functions you wrote for problem 2. Try to copy and modify as few functions as possible from the previous problem.

Testing Requirements: None. (but you should still test your function!)

Suggestions for solving problems 2 and 3:

a. Dijkstra's algorithm, which will be described in the lecture and in c-e below, is highly recommended. Also, diagrams of Dijkstra's algorithm in action are included below.

b. Use recursion to "retire" one node at a time, beginning with the source node. When a node is retired its minimum distance from the source is known. (This can be proved; it is not totally obvious. One proof is here: http://www.cs.auckland.ac.nz/software/AlgAnim/dij-proof.html. The Algorithms textbook by Corman, Leiserson, Rivest and Stein provides another, and probably clearer, proof).

c. Keep two lists of wrapped nodes: The list of "retired" nodes and a list of "active" nodes. All nodes start on the active list and reachable nodes move to the retired list, never to return.

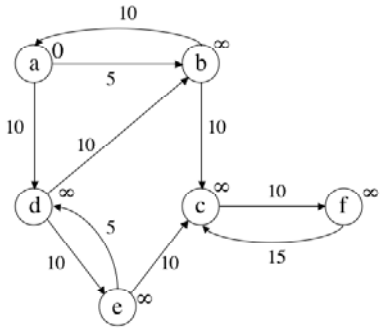
d. On each major step, the active node with the minimum distance is retired and the best estimates of minimum distances to the remaining active nodes are updated for posterity, to reflect the possibility of a path from the newly-retired node to the remaining active nodes. (Updating requires changing the distance component of the active nodes, but not the nodes themselves.) This is the essence of Dijkstra's algorithm. (When a lesser distance is installed, the predecessor is updated too.)

e. One way to find the minimum of a list is to sort the items and choose the first. This is not optimal, but it will be adequate for this problem.

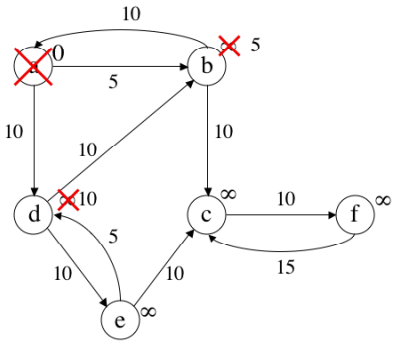
f. You will need to implement your own addition and comparison functions to handle infinity. This could be done using a really large number to represent infinity, but this would be non-robust because it commits that number to this special use. (It would be ok for your initial prototyping, however.)

Dijkstra's algorithm on the given graph example:
X'ed nodes are retired. X'ed numbers are distance updates. Final step not shown.

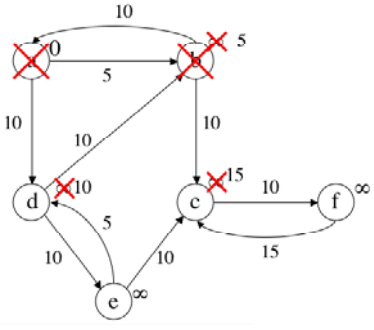
1.
Original node distance assignments (source is node a)



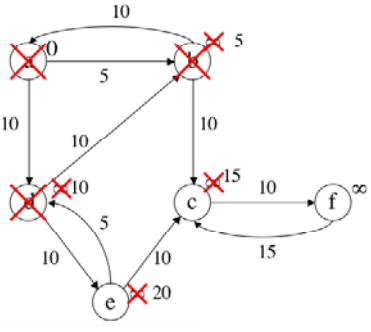
2.
Retiring minimum distance active node: a



3.
Retiring minimum distance active node: b

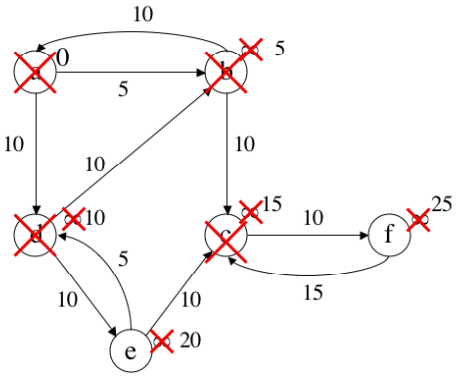


4.
Retiring minimum distance active node: d



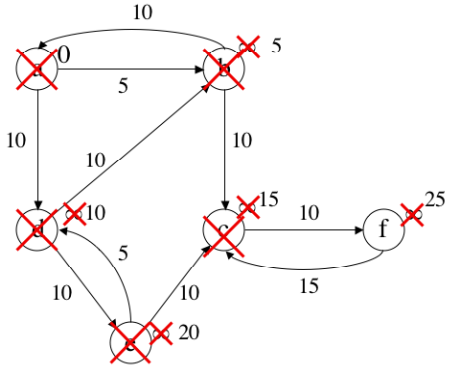
5.

Retiring minimum distance active node: c



6.

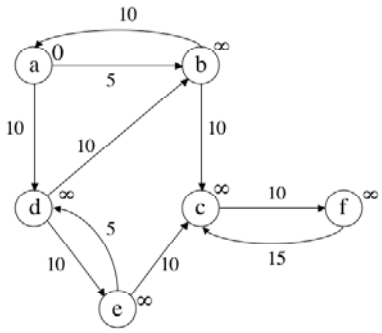
Retiring minimum distance active node: e



Dijkstra's algorithm showing predecessor references (dashed arrows):

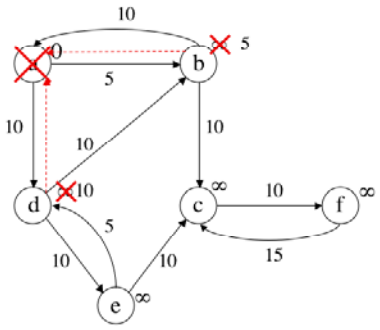
1.

Original node distance assignments (source is node a)



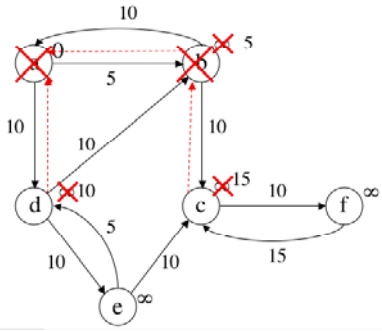
2.

Retiring minimum distance active node: a



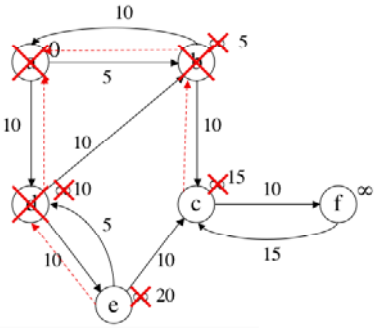
3.

Retiring minimum distance active node: b



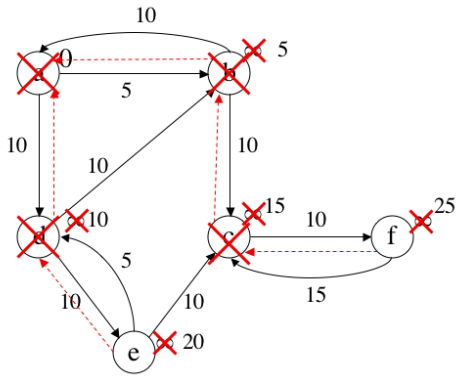
4.

Retiring minimum distance active node: d



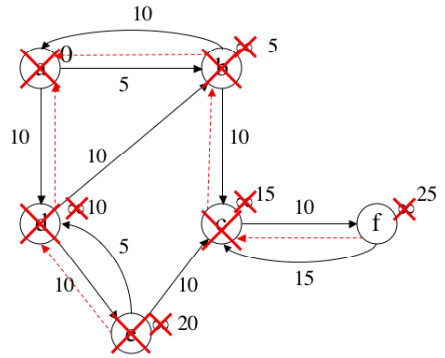
5.

Retiring minimum distance active node: c



6.

Retiring minimum distance active node: e



The final step is not shown, for reasons of brevity.