CS 450 Homework 4
Bill Campbell (verbatim from Carl Offner)
Spring 2016

Due Sunday, March 6, 11:59 PM

The problems are collected electronically, as usual, with the exception of Problems 2 and 5. Problems 2 and 5 should be hand-written (unless you really want to use some drawing tool, but I don't think that's worth the effort). Those two problems can be handed in at the **beginning** of class on Tuesday, October 8. That means at 5:30 PM sharp -- not 5:35 PM. I will not accept either of these problems after the class has started. If for some reason you cannot be present when the class starts, make sure that these problems are in my mailbox in the Math/CS office no later than 5:00 PM.

Please note that problems 2 and 5 are exercises that show that you understand the environment model. To do this, you have to go back and do *exactly* what we discussed in class. If you just think you sort of understand things and write down what seems to be correct, you will almost certainly get these problems wrong. Many students in the past have done just that, and then they are astonished to find out that what they did makes no sense at all.

There are two big parts to this assignment, and you will hand in three files (electronically):

- **ASanswers.scm**—this will contain the answers to Part 1 (except for the two drawings already mentioned above).
- **path.scm**—this will contain a Scheme program that solves the problem of Part 2.
- **notes.txt**—This is the usual notes file. You should have something interesting and useful to say about these problems. I will take this very seriously.

Part 1: Assignment, Local State, and the Environment Model

There is a lot of new material here. It will take getting used to. You will find this material in Sections 3.1 and 3.2 of the text.

Problems 2 and 5 should be written out on paper and handed in as described above. Handwriting is fine, but *please* make them neat, clear, and easy to read.

Put your answers to the rest of the problems in file `ASanswers.scm` in your new project directory `.../cs450/hw4`, with discussion when required as Scheme comments. I will load and test your file, so be sure it's bug free. Anything that isn't yet working should be a stub or a comment. Any tests that you ran that are in this file should also be commented out.

1. Rewrite the `make-account` procedure on page 223 so that it uses `lambda` more explicitly. Create several versions, as follows. (**Important: please do exactly what each of the three following versions specifies; no more and no less.**)

    1. First version: First, replace

    ```
    (define (make-account balance)
     ...)
    ```

    by

    ```
    (define make-account-lambda
      (lambda ...
    ```

```
      ...
    )
  )
```

Then, for the first two internal procedure definitions, replace `(define (proc args) ...)` with `(define proc (lambda (args) ...))`.

Finally, replace the

```
(define (dispatch ...)
...
...)
dispatch)
```

construction with a lambda expression which is evaluated and returned (but of course not called). As indicated above, call this procedure `make-account-lambda`.

2. Second version: Start with a copy of the first version. Then inline the internal procedures `deposit` and `withdraw`. That is, replace references to them by the bodies of the procedures. Then you can eliminate the definitions of those procedures. Call this procedure `make-account-inline`.
3. Third version (A little extra credit): Start with a copy of the second version. I don't know how to say this without doing it for you, but you might then notice that a `lambda` can be factored out of the `cond` in your last version. (If you don't know what I'm talking about here, just ignore this part of the problem.) If you do this, call this new version `make-account-inline-factored`.

Note that none of these three versions of `make-account` contains a `dispatch` procedure.

2. In Exercise 3.11 (Page 251), Abelson and Sussman ask you to show how the environment model explains the behavior of make-account. Do that problem twice, once for the function `make-account` as A&S wrote it, with internal defines, and then again for the function `make-account-lambda` of problem 1.

   Notes:

   - Please only draw two pictures -- one for `make-account` and one for `make-account-lambda`. In spite of what the problem in the book says, please don't show in the picture how the deposit and withdrawal get made. I just want you to show the picture after

     ```
     (define acc (make-account 50))
     ```

     has been evaluated.
   - Please be careful: The two pictures you will draw are different. Make sure you understand how they are different, and why. This is a key point of the problem.
   - I am looking for clarity here. It's not my job to try to figure out what you have in mind. You have to tell me. Remember the main principle: explain things to me as if I had no idea what was going on.
   - The rest of the assignment is independent of the understanding called for here, so you can continue with the questions below before you have finished understanding the environment model.
3. Exercise 3.2 (page 224). Please read the statement of the problem very carefully. It tells you precisely what you should do. Please do exactly what it says.

4. Exercise 3.3 (page 225).

   In doing this problem, build on your solution to Problem 1. In fact, see if you can use one of your solutions to Problem 1 as a "black box" -- that is, make the solution to this problem a "wrapper" procedure that just invokes one of the versions of make-account from Problem 1, after handling password checks. In this way, you don't have to copy any of the body of the original make-account procedure. (It isn't necessary that you do it this way -- this is just a suggestion.)

   Call your new function `make-pw-account`. And yes, I really meant this—note that this is different from what the book says.

5. Exercise 3.9 (page 243). Let's make it simpler, however: show how to compute `(factorial 3)` (rather than `(factorial 6)`).

   Be sure to read the footnote. And follow the construction that I gave in class *exactly*. You may think the pictures should look different. And if something bothers you about how the pictures look, you should write about it in your `notes.txt` file. But the construction that I specified is actually what happens internally. You will have to understand it for later assignments.

Part 2: Dynamic Programming and the Remarkable Effectiveness of Memoization

Statement of the problem:

Here is a problem: we are given a weighted DAG—that is, a directed acyclic graph with a weight on each edge. You can assume that all the weights are non-negative. One of the nodes is called `start` and another is called `end`. There are paths through the graph from `start` to `end`. Each path has a `cost`—this cost is the sum of the weights on edges that make up the path. The problem is to find a path of minimal cost from `start` to `end`.

Note that there might be more than one path of minimal cost. We only have to find one of them. And we are guaranteed that there is at least one path from `start` to `end`.

You can think of this as a simplified form of the kind of problem that is solved all the time by Google maps or by a GPS device in your car: What's the best way to get from one point to another?

This problem is computationally expensive!

One way to approach this would be to write a recursive routine which walked the graph starting at `start` and explored all paths, stopping each path when `end` was reached, and keeping track of the cost of each path.

This would certainly work, because there is only a finite number of paths from `start` to `end`.

On the other hand, the number of such paths could easily be so great that the program would take an inordinate amount of time to complete.

Think, for example, of a hypercube of dimension `n`. How many vertices does it have? Can you figure out how many paths are there from one vertex to the "opposite" one? Can you draw any conclusion from this? (This question may be more sophisticated than it seems. Don't spend a huge amount of time on it.)

An idea that can be used to simplify the computation

On the other hand, even though there can be an absolutely huge number of paths from `start` to `end`, many of them quite likely overlap. For instance, suppose we have two paths like this:

```
start → a → b → … → p → … → t → end
start → x → y → … → p → … → t → end
```

which start out differently but from point `p` on are the same. Then if we know the cost of the sub-path

```
p → … → t → end
```

then we can compute the cost of each of the original paths with less effort, because we only needed to compute the cost of that sub-path once. We'll come back to this later.

How the data is input

Before we go on, let me explain how the data will be input to the program you are going to write. There will be a file containing the specification of the graph. The name of the file will be `dist.dat`. That's it. You can't use any other file name. This is just to make things simple (even at the cost of being a bit user-unfriendly). Please don't try to change this.

A typical (but very small) `dist.dat` file might look like this:

```
(start p1 3)
(start p2 7)
(p1 p2 1)
(p2 end 11)
(p1 end 22)
```

Each line in the file looks like a Scheme list. Each line represents an edge in the graph. The first two elements in the list are the source and the target of the edge, in that order. The last element of the list—which is always a non-negative integer—is the weight of that edge.

Before going any farther, you should draw a picture of this graph for yourself. (Don't hand it in.) You should be able to see that the shortest path from `start` to `end` has cost 15.

Don't make any assumptions about the order in which these edges are placed in the file. For instance, this would describe exactly the same graph:

```
(p2 end 11)
(start p2 7)
(p1 p2 1)
(start p1 3)
(p1 end 22)
```

And for that matter, the nodes (with the exception of `start` and `end`) could have any names, in any order. So this would also really be the same graph, with the internal nodes renamed:

```
(p1 end 11)
(start p1 7)
(p2 p1 1)
(start p2 3)
(p2 end 22)
```

It also might be that there are other "initial" or "final" nodes in the graph. For instance, you might have something like this:

```
(start p1 3)
(p0 p1 2)
(start p2 7)
(p1 p2 1)
(p2 end 11)
(p2 p3 1)
(p5 p6 3)
(p1 end 22)
```

But in any case, there will always be one or more paths from `start` to `end`, and it is *only* those paths that we care about.

Also note that between any two nodes in the graph, there is either no edge, or there is 1 edge. There is never more than 1 edge.

Reading in the graph

As in the previous assignment, you can use the following code to read in the graph:

```
;; read-file produces a list whose elements are the expressions in the file.

(define (read-file)
  (let ((expr (read)))
    (if (eof-object? expr)
        '()
        (cons expr (read-file)))))

;; Here we go:  read in the file that defines the graph

(define data (with-input-from-file "dist.dat" read-file))
```

This will give you a variable named `data` that holds a list the elements of which are just the lines in the file `dist.dat`.

Building the main lookup table

The next thing you will need to do is to build a lookup table. This table should enable you to implement a lookup function (in fact, let's call it `lookup`) that takes the names of two nodes in the graph (like `start` and `p2`, for instance). If these two nodes are not the source and the target (in that order) of an edge, this function should evaluate to `#f`. Otherwise, it should evaluate to the cost of that edge.

We talked in class about how to build lookup tables (and the same thing is in the textbook, in section 3.3.3). In this case, we are talking about a two-dimensional table, right? You should use the method we talked about in class.

A naive way to solve our problem

A straightforward but naive way to solve our problem would just be via a depth-first walk of the graph, starting at the `start` node. Pseudo-code for this might look something like this. (Bear in mind that this pseudo-code is not really "pseudo-Scheme", but it could be easily translated into it.):

```
procedure cost(node) // returns an integer
  if the node has no children, return "infinity".
  for each child of node
    if child is "end"
      just compute (lookup node child)
```

```
    else
      compute (lookup node child) + cost(child)
  return the minimum of those computed values
```

Here "infinity" just means some number that is so big that it couldn't possibly be the value of any path. You can use the number 1000000 (i.e., 1 million) if you want—I guarantee that I won't try your code out on any graphs with weights that could possibly add up to a million. And of course any graphs that you make up should satisfy the same constraint. That shouldn't be hard.

I suggest implementing this. It's quite easy, and it will give you some practice in specifying graphs and convincing yourself that you understand what is going on. Of course you want to make sure that what you get back is what you think you *should* get back.

Why does this method work?

This method, which is quite simple, works for a reason which is actually pretty sophisticated: Suppose that we have a path `P` from `start` to `end`, and suppose `P` passes at some point through a node `p`. And suppose the cost of this path is minimal—that is, there is no other path from `start` to `end` with lower cost. *Then the part of this path (call it `Pp-end`) starting from `p` and going all the way to `end` must also be minimal*, in the sense that there is no other path `S` from `p` to `end` whose cost is smaller than the cost of `Pp-end`.

Why is this? The reason is this: suppose there *was* a path `S` from `p` to `end` whose cost was less than the cost of `Pp-end`. Then we could make a new path as follows:

Follow the original path `P`from `start` to `p`. Then, instead of continuing on `Pp-end`, follow `S` from `p` to `end` This new path from `start` to `end` will have a cost strictly less than the cost of `P`, which is of course a contradiction, since we started out by assuming that the cost of `P` was minimal.

Thus we have proved that if `P` is a minimal-cost path from `start` to `end`, then the sub-path of `P` from any node `p` to `end` is also a minimal-cost path from `p` to `end`.

And that's why the algorithm above works. At each stage we take the minimal cost of any path from the children of `node` and put it together with the cost to each child. We are thus guaranteed to get the minimal cost from the `node` itself.

The real way to solve our problem

However, as we have already mentioned, this algorithm is just too costly to run on large graphs. And the reason is that we can end up computing the same information (for common sub-paths) many times. But based on the discussion above, we can get around this difficulty by making sure that we compute the cost of each path just once. We do this by a process called **memoization**.

Please be careful. There is another word, "memorization" that you probably know. That's a different word. The word "memoization" (without the "**r**") is used only in Computer Science. Every computer scientist knows what it means, and almost no one else does. We just have to live with that. The words are clearly related, but they mean different things, so you do have to be careful.

The idea is simple: we keep a table of costs. For each node, when we compute the cost of that node, we enter that node and its cost in the table. Then every time after that, when we want to compute the cost of the node, we first look it up in that table. If it's there, then we know the cost. Otherwise, we compute it and put it and its cost in the table.

In effect we are making a "memorandum" (or "memo") of the cost of each node as we compute it. That's where the term "memoize" comes from.

This use of memoization, which is made possible by the "minimal sub-paths" property we proved above, is called **dynamic programming**. Dynamic programming is one of the most powerful techniques of algorithm construction. It's tremendously useful.

You should implement this. In doing this you will want to be very careful that you really are not redoing computations that can be memoized.

You may find that you need (or want) to have several tables. That's fine. Do what you need to do, and please explain your decisions in `notes.txt`

The final version

Finally, we would like to print out, not only the minimal cost of getting from `start` to `end`, but also a path that has that minimal cost. (Remember that there might me more than one minimal path. That's OK; we only care about one, and it doesn't matter which one.)

To do this, you will also need to memoize a shortest path from each node to `end`. In principle, this is not at all hard to do, but you will need to be careful.

I'm sure you will need some helper functions in doing all this. And will also undoubtedly need some more tables. Be sure to put in comments so that your code is reasonably explanatory. When I did this, I had functions that returned data structures—not just simple numbers or lists. You will want to document that as well.

How should the result be printed out? I would like you to print out the result like this. Suppose you have read in the original graph I gave as an example above. Then your file—which we're calling `path.scm` should be able to be executed by simply typing

```
scheme < path.scm
```

at the Unix prompt. It should print out

```
(8 start p1 p2 end)
```

In other words, you don't call `cost` or any other Scheme function from the Unix command line. You just read in `path.scm`, as I indicated above::

```
scheme < path.scm
```

This will evaluate every Scheme expression in the file and display the result. Of course, that's not what you want. So package up the file as one big `(begin ...)` special form, and make it so that the last thing evaluated in the `(begin ...)` is what you want displayed. That is, the last expressionin the `(begin ...)` expression should be

```
   (cost 'start)
```

Please be sure to do this. This is the way I'm going to test your code. If your interface is different in any way, my tests will fail.

Finally, I expect to see a serious discussion in your `notes.txt` file of the design decisions you made, any

difficulties you ran into, how you resolved them, what you would do differently, and so on. This is important, and I plan to take this very seriously. So don't leave it till the end, and don't blow it off.

A moderate-sized test case

Although in developing your program you will certainly want to work with very small graphs that you make yourself, so that you can debug your code easily, you might like to have a somewhat larger graph to test your code on. I have put a graph `dist.dat` in ~offner/cs450/hw4. You can certainly feel free to use it. You should find that the minimum-length path from `start` to `end` in that graph has length 169.

One final warning. There are other algorithms that might be used to solve this problem. But I want you to use the approach I have given here, which is one based on dynamic programming. So poking around the internet or Wikipedia —or anywhere else, really—isn't going to help at all. You may find another algorithm. But it won't be any easier or more efficient than this one, and in any case, a big point of this problem is to understand this kind of algorithm, which is extremely important in computer science. If you have a question understanding what I've written here, you should ask me.

If you use another approach, I'm not even going to look at it.

Of course, there are design decisions that you will make as you implement this, and different people will do things somewhat differently. That's perfectly OK, and in fact I would expect that. That's not what I'm talking about here. If you're uncertain what I mean in this all, please just send me email.