CS 450 Homework 4

Carl Offner

Spring 2022

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

Due Wednesday, February 23, 5:00 PM

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, as well as in the recent lectures.

For this first part of the assignment, you will hand in problems 2 and 5 on paper (in class), and the rest of the problems electronically:

- Problems 2 and 5 should be written out on paper and passed in at the beginning of class on Wednesday, February 23. Handwriting is fine, but the work *must* be neat, clear, and easy to read.

  I will not accept papers after the class has started. That means 5:30 PM sharp—not 5:35 PM. If for some reason you cannot be present when the class starts, make sure you email me copies of these papers 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.

- 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.

- You should not need to use `set-car!` or `set-cdr!` anywhere in Part 1 of this assignment. If you do, I won't even look at it. (And I'm not doing this to be mean. Neither of them would help you in any case.)

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. Consider the procedure `new-withdraw` which we talked about in Lecture 6, and which we discussed again in Lecture 7. The implementation of that procedure is sketched in Figure 10 on page 8 of Lecture 7.

    1. I want you to draw what that picture looks like after

        ```
        (new-withdraw 25)
        ```

```
is evaluated.
```

2. Then draw a second picture that shows the situation after

   ```
   (new-withdraw 30)
   ```

   is **subsequently** evaluated.

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. And do not use any global variables in doing this problem.

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 mean 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 than what you get. And if something bothers you about how the pictures look, you should write about it *as a comment* in your **ASanswers.scm** 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

Due Monday, February 28, 5:00 PM

You will hand in three files electronically:

- **naive-path.scm**—this will contain a Scheme program that solves the "naive" version of the problem of Part 2. (That is it contains the code for `naive-cost` and any helper functions and data structures.)
- **path.scm**—this will contain a Scheme program that solves the complete version of 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.

Statement of the problem

Here is a problem: we are given a finite weighted DAG—that is, a **directed acyclic graph** with finitely many nodes and 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?

Discussion and some preliminary ideas

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`.

Question: How do we know that there are only finitely many paths from `start` to `end`? This is actually a more sophisticated question than you may at first think. Please be careful about your reasoning.

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.

Consider, for example, this dag.