CS 450 Homework 2

Carl Offner

Spring 2022

Due Monday, February 7, 5:00 PM

As will be the case for all future assignments, I will collect all work electronically at 5:00 PM on the day it is due. So be sure it is in the proper directory at that time.

Put your answers in file `ASanswers.scm` in your new `hw2` directory `.../cs450/hw2`, with discussion when required as Scheme comments. (A Scheme comment starts with a semicolon and continues to the end of the line.) I will load and test your file, so be sure that

- It's bug free.
- Anything (such as an explanation that you might write) that is not Scheme code is commented, so the Scheme interpreter does not attempt to evaluate it.

In particular, anything that isn't yet working should be a stub or a comment.

And just to be clear on this: if your file does not load into UMB Scheme without errors, I will not even look at it—and that will be true for every assignment. I don't have time to fix that kind of bug. If you are having difficulty, you need to send me email **before** you pass in the homework, and preferably, the earlier the better.

Also please make sure that

- None of your lines is more than 80 characters long. I plan to enforce this—you have been warned.
- You have spelled things correctly. You can use a spell-checker to help with this.
- There are no spurious control characters in your code or any other text you write. If you use Emacs, this will not be a problem. If you use some Microsoft software, it almost certainly will be a problem.
- You have Unix line endings, not Microsoft line endings. Use `dos2unix` if you're not sure.

Finally, a word of warning. Some people have posted answers to some of the problems from the book on the web. I strongly suggest that you **not** look for them and **not** use them. The reason is simple: whether those answers are right or wrong, you won't learn a thing. I know this. I've seen it happen. And if you have't learned the material in this assignment, and learned it well, you won't pass this course.

I'm happy to help you myself. I'm happy to have you get help from other students and other places, provided you acknowledge that help. But

"getting help" does not mean looking at any else's code or having them tell you what to write. I'm not at all happy to have you copy answers from anyone else or anyplace else, under any circumstance.

The purpose of the following problems is to become familiar with the `quote` special form, and to learn to think recursively about lists and to use the list primitives of Scheme: `cons`, `car` and `cdr`.

1. Write a Scheme procedure `is-list?` that tests to see if an object is or is not a list, according to the definition of a list in the Scheme language definition. (Where is this definition? OK, I'll tell you: it's on page 25 of the Revised Report, in Section 6.3.2, which is titled "Pairs and lists". It's the second paragraph in that section.)

   Note: UMB Scheme already contains a built-in function `list?` that does this, and is written so cleverly that it works on cyclic data structures. Please write your own procedure, however; you may assume that the argument passed to it is not cyclic. (And if you don't know what "cyclic" means in this context, don't worry about it.)

2. Exercise 2.18 (page 103). Call your procedure `my-reverse`. Now that you know about `quote` you can test with more than lists of integers.

3. Exercise 2.20 (page 104). (And remember what you learn in this exercise! You'll use it later in this course.)

4. Exercise 2.21 (page 106).

5. Exercise 2.23 (page 107). Call your procedure `my-for-each`.

6. Do exercises 2.24, 2.25, and 2.26 (page 110), and also 2.53 (page 144), but don't turn in the answers.

   Note: Many students (and I know you are all pressed for time) figure this means they can just skip these exercises. Of course I won't know if you don't do them. But I guarantee that you will need to understand these four exercises for a lot of the code that you will be writing in future assignments. So please be sure to do these and take them seriously. And if you don't understand something, please ask me.

7. Exercise 2.54 (page 145), with the following change: in the statement of the problem, substitute `eqv?` for `eq?`. This makes the definition closer to the Scheme language definition. (In fact, it would be even better to substitute "pair" for "list" in the recursive definition given in the problem. You can do this.) Call your procedure `my-equal?`.

8. a. Define a procedure `every?` which takes two formal parameters:

   - `pred` is a predicate, and
   - `seq` (i.e., "sequence") is a list.

   `(every? pred seq)` evaluates to #t when every element of `seq` satisfies `pred`, and evaluates to #f otherwise.

b. What should happen if the list is empty? Justify your answer in a (clearly written) comment.

Here's a hint, which I expect you to use in your answer: if `seq1` and `seq2` are lists, then it should be the case that

```
(every? pred (append seq1 seq2))
```

should be the same as

```
(and (every? pred seq1) (every? pred seq2))
```

This should make sense to you. Make sure it does.

Please note: I am asking you for what **should** happen. So in particular, if you write something like "I tried it out and this is what happened.", your answer is automatically incorrect. (On the other hand, you definitely should try some of this out. I have occasionally seen students hand in an explanation of what they thought should happen, but it was wrong, and if they had only tried it out, they would have seen that it was wrong. So you should definitely try things out. But just reporting what happened when you tried something is not what I am looking for.)

I'm looking for clearly expressed reasoning here. You have all had a discrete mathematics course, so you should know what I mean. (Of course, as I explained above, if what **does** happen is not what **should** happen, something is wrong, either with your reasoning or with your code, right?)

Here's some more on that hint: This is somewhat similar to how we prove that the product of a negative number and a positive number is negative, and also that the product of two negative numbers is positive. I have written up a proof in this style in Why does a negative times a negative make a positive?, which you should definitely look at. But be careful: you can't just copy what is there and make a few little substitutions. The details are quite different.

9. Exercise 2.59 (page 153). Call your procedure `unordered-union-set`.

10. Exercise 2.62 (page 155). Call your procedure `ordered-union-set`.

11. Define a procedure `remove-val` which takes two parameters: a value and a list. It returns a list which is the same as the original list except that every element having the given value is deleted. If there are no such elements, the original list is returned. So for instance,

```
(remove-val 3 '(4 5)) ==> (4 5)
(remove-val 3 '(2 3 4 3)) ==> (2 4)
```

This exercise will also be useful to you in a later assignment. (And I really mean this. In the past, I have been amazed by the number of students who really did badly in a later assignment because they had completely forgotten this.)