CS 450 Homework 3

Carl Offner

Spring 2022

Due Monday, February 14, 5:00 PM

This assignment uses the functional aspects of Scheme to construct a relatively simple but non-trivial program. Once you have finished this assignment you will have a good command of the functional style.

I will collect two files from your `hw3` directory:

- `convert.scm`
- `notes.txt`

whose contents I describe below.

In contrast to the last two assignments, this assignment is much larger and will take much more time, spread over many days. It is not an assignment that you can just jump into and do, and it is not something you can leave for the weekend. There is some serious thinking involved. Please read everything carefully before you do anything else. In fact, read it more than once. It will save you a lot of time, and you will understand this whole area much better.

Transforming Units

You may know that Google has a wonderful facility for converting units. For instance, suppose you are cooking and you forget how many teaspoons are in a cup. You can simple type "1 cup in teaspoons" into the Google search window, and the answer will come up. And you can even perform more complicated (or silly, depending on how you look at it) queries. For instance, if you haven't already done this—it's a pretty well-known way to amaze your friends—try typing

```
furlongs per fortnight in miles per hour
```

into the Google search box. And you can also type things like

```
27.5 furlongs per fortnight in miles per hour
```

In this assignment you are going to write a program that does more or less the same thing.

Before we get going, I should say that there are a number of such programs available on the web. I don't believe that any of them are written in Scheme. And further, I don't believe that looking at them is going to help you. (I know this because when I started to do this, that's exactly what I

did, and I finally realized that it was getting me nowhere.) On the other hand, I am going to give you some fairly significant suggestions, which I hope will help you a lot in doing this.

And as usual, the work you hand in must be entirely your own in any case.

We are not going to write a complete converter program. That would be too much for an assignment like this. But if we did, it would have three parts:

1. **Input:** take a phrase, and turn it into some internal representation of what is being asked for. This is a very complex undertaking. For instance, the program has to be able to recognize plurals ("meters" vs. "meter", for instance) and common prefixes ("kilo-", "centi-", and so on). This takes up the bulk of most of these programs from what I can see. It's an interesting problem, but it's not the problem we really care about here.

2. **Processing:** Take the "from" quantity (e.g., "27.5 furlongs per fortnight", only written in our internal format) and our "to" quantity (e.g., "miles per hour", again written in our internal format) and perform the translation. This is what we are going to do. In our case, the format of the "from" quantity will be a list, like this:

   ```
   (27.5 (furlong 1)(fortnight -1))
   ```

   and the format of the "to" quantity will also be a list, like this:

   ```
   ((mi 1)(hr -1))
   ```

   and the actual query will be written as a Scheme function taking these two arguments, like this:

   ```
   (convert '(27.5 (furlong 1)(fortnight -1)) '((mi 1)(hr -1)))
   ```

   Question: why are the quotes necessary? (Put your answer in notes.txt.)

3. **Output:** The value of the expression we just wrote will be

   ```
   (0.01023065476190476 (mi 1)(hr -1))
   ```

   A user-friendly procedure would put this back into everyday English. But we're not going to worry about that. We'll just leave it as is.

But we are not going to do all this:

We are going to cut out the Input and Output parts of the program completely and only implement the Processing part. We do this by insisting that the user pass in the query in a very formalized manner—essentially, in the internal representation that we will be using. This makes the program

we will write much less user-friendly, but also makes it possible for us to do this as a homework assignment. And in any case, what we produce here could be used as the central component of a fancier program. (That "fancier program" would consist of a "front end" that takes in ordinary English queries as above and translates them into the form we will use here; then just passes those translated forms into the program of this assignment.)

The Important Question

How does the program know how to perform the transformations? Equivalently, how does it know how different units are related? The answer is that it gets this information from a file that it reads in. That file is named `units.dat`. I have created this file for you, and you can find it in ~offner/cs450/hw3/units.dat. It is a plain text file. You should look at it and see how it is constructed. Copy it into your hw3 directory. And **don't change it in any way**.

One thing you will notice is that it is very simple indeed. All units are defined in terms of certain "elementary" units. (And an elementary unit, for this purpose, is a unit that is not defined in terms of another unit in the `units.dat` file.) All length units, for instance are represented in terms of meters. Now it would be actually more convenient in constructing this file to allow it to contain entries in which some units were defined in terms of other derived units, like this:

```
    (yd (3 (ft 1)))
```

But this would complicate our processing, and so we won't do it.

We have not discussed how to read in a file to a Scheme program. We *will* deal with this later in the term. But in the meantime, here's how you do it: At the end of your file `convert.scm` put the following code:

```
------------------------
;; 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 database.

(define source (with-input-from-file "units.dat" read-file))
------------------------
```

This will read in the file `units.dat` and put the contents as a Scheme list into the variable `source`. Then you can just use that list as a lookup table. (How do you look up things in that table? Think about `assoc`.)

To make things as clear as possible, let us make some definitions:

- A *base-unit* is something like `m` (i.e., "meter"), or `sec` (i.e., "second").
- A *unit* is a list consisting of a `base-unit` together with an exponent. For instance, the `unit`

  ```
  (m 2)
  ```

  represents "meters squared", or "square meters"—these mean the same thing.
- A *unit-list* is a list of `unit`s. For instance,

  ```
  ((mi 1)(hr -1))
  ```

  represents "miles per hour". And

  ```
  ((kg 1)(m 1)(sec -2))
  ```

  represents "kilogram meters per second squared".
- A *quantity* is a list whose first element is a number and the rest of whose elements are units. That is, the `cdr` of a *quantity* is a *unit-list*. For example,

  ```
  (1 (kg 1)(m 1)(sec -2))
  ```

  represents "1 kilogram meter per second squared", which is also known as "1 newton", and is the unit of force in the MKS ("meter-kilogram-second") system that I hope you are familiar with from whatever physics course you took.
- Finally, the procedure we are going to write is named "convert", and it has the following signature:

  ```
  (convert <quantity> <unit-list>)
  ```

  I've already shown you what the output looks like (i.e., the kind of Scheme expression this expression evaluates to) above.

Now here are some ideas that you may find useful. Of course you can really do this any way you want, and I certainly don't insist that you take any of these ideas exactly as I am writing them. (In the past I have found that students often have very original and interesting ideas about how to solve particular problems, and so I encourage you to experiment with different ways of doing things.)

First of all, let's look at the general idea of what you want to do. Conceptually, it's really pretty simple. You are given a *quantity*—say it is of the form

```
(a U)
```

where `a` is a number and `U` is the sequence of *units* in a *unit-list*. For instance, `U` might be `(mi 1)(hr -2)`. And suppose you want to convert this original quantity to a quantity whose sequence of units is `V`; i.e., a quantity whose *unit-list* is `(V)`.

The first thing you have to check is that `U` is compatible with `V`. What this really means is that if you express both `U` and `V` in terms of the same elementary units, then `U` turns out to be a constant multiple of `V`. So the first thing you need to do is to transform `U` and `V` so they are expressed in terms of the same elementary units. You will need a procedure that does this. As already specified above, you may assume that if a base-unit is not found by your lookup procedure, then it is already an elementary unit (like m, or sec, for instance).

Applying the procedure, you will then get

```
U = u U-normalized

V = v V-normalized
```

where `U-normalized` and `V-normalized` are written in elementary units, and where `u` and `v` are numbers.

Now `U` and `V` are compatible if and only iff `U-normalized` is the same as `V-normalized`. So you will need a procedure to check this. And somewhere along the way, you will need to take account of the fact that the order of units in a unit-list doesn't matter. For instance, `(m 1)(sec -2)` is entirely equivalent to `(sec -2)(m 1)`, or even to `(sec -1)(m 2)(sec -1)(m -1)`.

Assuming then that `U` and `V` are compatible, you should be able to see that

```
U = (u/v) V
```

In fact, I want to you show this in your `notes.txt` discussion.

Finally then, we get

```
a U = (au/v) V
```

So that's what you have to do.

You will need a number of helper functions to do this. Give them useful names that will help you (and me and anyone else who needs to read your code—you might want to come back and look at this code a year from now, for example) understand what you are doing. And comment them so it is clear what sorts of things they take as input and what they produce when evaluated. Usually giving a good example of how the function is called and what it returns is the way to go when doing this. Further, comments inside the function body can be very helpful in understanding tricky code. Code is not easy to read. Make it easy for the reader; and remember that *you* might be the reader.

You will also probably need a large number of temporary variables. Don't be afraid of the `let` special form. And you may find yourself nesting them, like this:

```
(let ((x ...)
      (y ...))
  (let ((z <something involving x>))
    ...
```

and this nesting might even be rather deep. Don't be afraid of this either. But *do* give your temporary variables descriptive names like `first-unit` or `converted-ratio` (not names like `x` or `y`, as I did above). This is important. I will be looking for this.

Be sure to test your procedure. Make up some miserably hard tests. Remember that a quantity could look like this:

```
(5 (joule 2)(N -3)(hr 1))
```

Although this makes little sense physically, your procedure should handle it just fine. Testing is hard. Try to be as nasty and malicious as possible in the tests you write. That's what software engineeers have to do. Don't ever assume that "no one would ever input something like ...". Believe me, someone will.

To handle complex quantities like this, you will need to find a way of simplifying them. This is part of the process—described above—of representing quantities in terms of elementary units. Be careful about this. It's possible to get confused here. Write down carefully how you are approaching this problem. Make sure you have put in some meaningful comments for this.

What you should not do

- I mentioned above how to recognise elementary units in the `units.dat` file. And I also said you should not alter that file in any way. So you might think that you could just look at the `units.dat` file and make your own list of elementary units and include that list in your code directly.

  Don't do this. Although you should not modify the `units.dat` file yourself, I might use a different one when I am testing your code. And that one of course might have a different set of elementary units. So please be careful about this.

- Please only use material we have covered in class up through Wednesday, February 2. In particular, do not use any non-functional forms such as `set!`. I'm serious about this. If there is even one such form in your code, I won't look at it.

  This is for two reasons:

  1. This point of this assignment is to become fluent in the functional programming style.
  2. In any case, this problem lends itself admirably to a functional solution. I have seen students try to use other techniques. Their code was not well written and generally didn't work. I think you will be amazed at how well the functional programming style works on problems such as this.

The file `notes.txt`

Finally, I am going to take your `notes.txt` file very seriously. First, please make sure you answer in your `notes.txt` file the questions I wrote above. However, if that is all you write about, I will not be very impressed. In addition to that, I want to know what problems you encountered as you did this assignment. What was easy? What took some thinking? What mistakes did you make, and how did you fix them? Would you do things differently a second time? And if you had more time, how would you extend this code to make it more useful and/or user-friendly? (I'm not looking for vague ideas here. If you have an idea, let me know how you would implement it. Don't write actual code, though.)

If you leave writing your `notes.txt` file until you have finished your coding, you most likely won't do a very good job. For one thing, you will have forgotten a lot. For another thing, you will probably be jammed for time. A better idea is to keep some sort of "diary" as you write your code. Then you can turn your diary into the essay in `notes.txt`

However: I don't want your `notes.txt` to be a bunch of raw unedited notes. I want it to be thoughtful and easy to read.

I also don't want to see big long listings of test runs. I do want you to indicate to me how you tested your code. (For instance, what sort of tests did you run? What were you looking for?) But please don't give me all the gory details.

Often I find students who think that they are writing `notes.txt` for me, and they figure that I already know all this. So they write very little, assuming that I can fill what they haven't written. There are two things wrong with this:

1. I'm not a mind-reader. While it may seem strange to you, I usually don't know what is going on in your mind, or how you got to understand something. And I'm not good at guessing that sort of thing. You have to tell me. I mean this. It's true.

2. In any case, you want to write this as if you had spent a considerable amount of time developing this software. And now you are going to move to another project, and someone who is much less experienced than you has just been hired to take this over. You want to write something that person can understand. Remember that the person doesn't really know very much and is just learning about this. Don't assume that person is an expert.

Just as one example of something you might write: It's perfectly OK, and in fact, it's good to write things like this:

```
At first I thought that

      <XXX>

But then I found that it led to the following problem:

      <YYY>

So I changed my design so that

      <ZZZ>
```

This is good, and I always like to read things like this, because it shows me how you came to understand the problem. And furthermore, it will be of great help to someone who is learning about this for the first time, since they will probably have the same misunderstandings, and this will help them get through this process more easily. And that's the point of writing something like this up.

In fact, I frequently put comments like that in my own code, because I know that even if no one else ever looks at it, I myself will probably have to look at it in a few months, or a few years, and the comment will remind me of where the tricky difficulty was.