CS 450 Homework 8
Computing with Register Machines
Bill Campbell (verbatim from Carl Offner)
Spring 2016

We will work through Sections 1 and 2 of Chapter 5 of the text in a number of short assignments:

1. Constructing some simple register machines: **Due in class, Wednesday April 20**. I won't collect these, but we will discuss them in class, and I will expect to see your work in class. These problems are very important; they form the basis for the rest of our work in this course:
    1. Exercise 5.1 (page 494)
    2. Exercise 5.2 (page 498)
    3. Exercise 5.4a (page 510)
2. Playing with the register machine simulator: The code is in `~wrc/public_html/cs450/hw8/regsim.scm`.
    1. Exercise 5.7 (page 515). More generally, make up some new machines and try them out as well. I won't collect this. (So if you don't do it, I guess I won't know. Well, that's not quite true. A more correct statement would be this: If you don't do this, you will most likely not have the slightest idea how to do anything else in this chapter. Unfortunately, my experience has been that many students figure that if the work is not going to be collected and graded, they can skip it. This always leads to massive difficulties later.)
3. Modifying the register machine simulator: **Due Sunday, May 1 at 11:59 PM (collected electronically)**.
    1. (Extra credit) Exercise 5.8 (page 523).
    2. (Extra credit) Exercise 5.9 (page 529).
    3. Exercise 5.19 (page 532).

    The two extra credit problems are a good chance to make up some points you may have lost on the last homework, which was admittedly quite difficult.

    Do these problems as follows: create the following files in your `hw8` directory:

    - `regsim.scm`
    - `regsim_5.9.scm`
    - `notes.txt`

    The first two of these files should both start out as the original `regsim.scm`. Do the work for Exercises 5.8 and 5.19 in `regsim.scm`, and do the work for Exercise 5.9 in `regsim_5.9.scm`. I will collect all three files.

    The reason for this is that while Exercise 5.9 is instructive, it actually implements something that we would have to undo later when we get to compilation. So we do it in a separate file, and then we will never use that file again. Your new file `regsim.scm`, however, will be used later, so please code carefully, and put in comments as appropriate.

    I don't think any of these problems are really hard. Problem 5.19 probably will take more time than the others, though. Also, the modifications to `regsim.scm` that you make for problem 5.19 will be useful to you in later assignments. I would say that, as usual, all these problems require more thinking than coding. One word of advice: don't worry about making your solution to Problem 5.19 particularly efficient. Anything that works is fine. Be sure you document your design in `notes.txt`. And as usual, I expect to see some interesting ideas and thoughts of yours in

`notes.txt`. What did you find difficult? What did you get wrong at first? What did you learn from this? What would you do differently? And so on.

Also, please read Exercise 5.19 carefully, so that you make sure that your breakpoints get placed properly. It's easy to be "off by one". Please be careful about this. It's a very common mistake.

Some remarks about labels

Here is some more explanation of how labels and the instruction list work. This may be helpful to you; in the past, I have found that students often get confused about this.

An instruction list is just that -- a list of instructions. So if the program is

```
start
  goto (label here1))
here1
  (assign a (const 4))
  (goto (label there))
here2
  (assign a (const 4))
  (goto (label there))
there
```

Then the instruction list (after assembly) looks like this:

```
(((goto (label here1)) . (lambda () ...))
 ((assign a (const 4)) . (lambda () ...))
 ((goto (label there)) . (lambda () ...))
 ((assign a (const 4)) . (lambda () ...))
 ((goto (label there)) . (lambda () ...)))
```

Now some students have thought that the label list in `regsim.scm` matches the label `here1` with the instruction that follows it -- `(assign a (const 4))` and then matches the label `here2` with the instruction that follows it -- `(assign a (const 4))`. The trouble is, of course, that those two instructions are really the same, even though they occur in two different places in the program.

So that can't be right. Instead, what happens is that the label `here1` is matched with the instruction list

```
(((assign a (const 4)) . (lambda () ...))
 ((goto (label there)) . (lambda () ...))
 ((assign a (const 4)) . (lambda () ...))
 ((goto (label there)) . (lambda () ...)))
```

and the label here2 is matched with the instruction list

```
((assign a (const 4)) . (lambda () ...))
 ((goto (label there)) . (lambda () ...)))
```

So the two labels really do point to different locations in the program.

Some hints for Exercise 5.19

I'm going to give you some hints for the required problem (Exercise 5.19) because we have so little time

for it and I really want you to be successful in this. This is the problem that involves creating a debugging facility in regsim.scm by putting in the ability to set breakpoints.

1. As I already indicated, the important thing is to keep this simple. In the past I have had a number of students who tried to do this using call/cc. This is not wrong at all, and could be quite elegant, but it is also pretty tricky. I think only one person actually succeeded in doing this. When I did the problem, that's not what I did. I suggest that you *not* try this.
2. I also have had many students who tried to implement a breakpoint by modifying the instruction sequence at that point. This is not a bad idea at all, and in fact, I think many debuggers insert breakpoints this way. The only problem—from our point of view in this course—is that if you do that, you have to be able to recover the original instruction after you continue. This is pretty tricky to do. I don't recommend it.
3. What I did was pretty naive and pretty simple: I just kept a separate list of breakpoints. Each breakpoint is a data structure that tells you where to break. You could include additional information, such as the label and offset that were specified by the user. That could help in reporting when you reach a breakpoint. You shouldn't need to modify the instruction list in any way to do this.
4. To handle breakpoints, you need to know where the labels correspond to in the instruction sequence. There is a problem here: by the time the assembler has returned (and so by the time make-machine has returned), the label list doesn't exist any more—it was just a local variable ("labels") in the assemble procedure. So you will need to save this in a local machine variable (alongside pc, flag, stack, and the-instruction-sequence in make-new-machine). You can perform this save just before the assemble procedure returns—to be precise, at the end of the lambda expression that is the second argument to the assemble procedure. (Well, not *quite* at the end. You still want this lambda expression to return what it formerly did; namely the list ("insts") of assembled instructions. So that has to be the very last expression in the body of that expression. But just before that would be a good place to perform the save.)
5. You can also make the breakpoint list be a local variable of the machine.

6. One more point that I have seen students get confused by: Breakpoints really have nothing to do with labels. A breakpoint is defined by a position in the program. What is "a position in the program"? It's exactly the kind of thing that you store in the pc register. The only thing that we use labels for in this connection is to allow the user to specify a breakpoint by saying something like "it's 3 instructions after label xyz". But breakpoints are *not* stored internally in terms of labels. And the way you tell if the instruction you are about to execute is at a breakpoint has nothing to do with looking at labels.

   Here's something else you should think about in this connection: Suppose you have the following code for the machine *my-machine*:

```
start
  goto (label here1))
here1
  (assign a (const 3))
  (goto (label there))
here2
  (assign b (const 4))
  (goto (label there))
there
```

   (This code is modified from some rather meaningless code in Exercise 5.8, and is equally meaningless.) It is perfectly legal to do this:

```
(set-breakpoint my-machine 'here1 3)
```

and it sets a breakpoint just before the assignment to register `b` just after the `here2` label.

Further, after doing that, you should be able to remove that breakpoint by *either* of the following Scheme expressions:

```
(cancel-breakpoint my-machine 'here1 3)
```

or

```
(cancel-breakpoint my-machine 'here2 1)
```

So again, we see that breakpoints can't be stored internally in terms of labels.

You have to understand this. If you don't, you need to ask me about it, and please do this immediately.

7. And finally, please be careful that you read the problem carefully. In particular, take careful note of the example that the book refers to (regarding a breakpoint in the gcd-machine). You don't want to be "off by 1".

I only make these suggestions because I think they will make your job easier. If you have already thought through a different design, that's fine with me. In fact, I have seen quite a large number of different ways of doing this problem over the years. There's no one best way. I just don't want you to get stuck.

Also, as I hinted at above, I found the breakpoint facility to be very useful when I did the last assignment. You may also find it useful. So really try to make work well; you'll be thankful later!

I hope that helps. And *please* don't hesitate to send email!