CS 450 Homework 10

Compilation!

Carl Offner

Spring 2022

Due Tuesday, May 17, 5:00 PM

In this assignment you will make some modifications to the compiler presented in Chapter 5 of the text to make it suitable for compiling a larger and more respectable subset of Scheme. In particular, you will be able to compile the metacircular evaluator of Chapter 4.

Start by copying from your `hw9` directory into your new `hw10` directory the following files:

```
regsim.scm
syntax.scm
eceval-support.scm
```

(You don't need `eceval.scm`.) Then copy from `~offner/cs450/hw10` into your `hw10` directory the following files:

```
compiler.scm
compiler-shell.scm
machine-shell.scm
```

You may make modifications to any or all of these files. (Well actually, you won't need to make any changes at all to `compiler-shell.scm`, and I would be astonished if you needed to make any changes to `machine-shell.scm`. So you can pretty much forget about modifying those two files.) I will collect all of them, and in addition I will collect the usual file

```
notes.txt
```

In doing these problems, you will want to add more Scheme primitives to the ones already supported by the compiler. You add them simply in `eceval-support.scm` to `primitive-procedures`; each new primitive is a one-line addition to that list.

Just to be clear about this: the primitive procedures you add in this way are names of procedures that the compiler (which is, after all, compiling Scheme code) will recognize in the Scheme code it is compiling as being Scheme primitive procedures. So for instance, if you are compiling a Scheme program that includes the Scheme primitive `not`, then you would want to add `not` to the list of `primitive-procedures` in `eceval-support.scm`. And **don't** add them to `primitive-procedures` in `s450.scm`! `s450.scm` is just the program you will be compiling. Dont make **any** changes to it. The reason for this is that the `primitive-procedures` in `s450.scm` are what that Scheme interpreter will recognise as Scheme primitive procedures in what **it** (i.e., the Scheme interpreter `s450.scm`) is interpreting. This is a bit confusing at first, I know.

Also please note that there are some changes in the compiler that I have given you from the compiler in the book, principally in the lists of modified registers that are passed around.

---

1. Add support in the compiler for `let` and `or`. You can implement these simply as syntax transformations: just add procedures `let->appl` and `or->if` to `syntax.scm` and then use these procedures in the dispatch procedure of the compiler. Do it as follows:

   - `let->appl` turns a `let` expression into an application of a `lambda` expression. (That is, it turns a `let` expression into a procedure call. We covered this very early in the term, so it's in your notes. Make sure you look it up, and make sure you understand it.

   - `or->if` turns an `or` expression into an `if` expression. Remember that an `or` can have more than two "arguments".

   This is entirely analogous to the way the compiler handles `cond` by transforming it into a set of nested `if` clauses using the syntax procedure `cond->if`. The main dispatch procedure of the compiler recognizes each `cond` expression and invokes `cond->if`, passing the result to `compile`. Of course you may also need some supporting functions for `let->appl` and `or->if` in `syntax.scm`. Add whatever you need, and put them in `syntax.scm`.

   And **please** look up the definitions of `let` and `or` in R5RS to make sure you have really implemented the Scheme definitions of these procedures. You will undoubtedly be surprised at what you find. If you don't do this, I can almost guarantee that your compiler will not pass my tests.

2. Extend the compiler so you can compile the metacircular evaluator from Chapter 4. I have placed a very slightly modified version of that evaluator in `~offner/cs450/hw10/s450.scm` which you should link to. (Or copy it into your own directory. But I'm not going to collect it; I'll use my own copy. So don't even think of editing it.) This is the version I want you to be able to compile. The main difference between this version of s450.scm and the original one is that after the metacircular evaluator is loaded it starts executing without waiting to be invoked. This is necessary because otherwise `machine-shell.scm` exits immediately to the underlying Scheme.

   This is essentially Exercise 5.50 on page 610 of the textbook. It's fairly difficult, so let me give you some advice:

   - As I mentioned above, there are some primitive operations that you will need to add to the initial environment of the program being compiled. (The primitive procedures currently in that environment are in the list `primitive-procedures` in `eceval-support.scm`. You can add to this list. Take a look through s450.scm and see what you need. For example, you can put `cadr`, `eq?`, `set-car!` in as primitives.)

   - Now here is the hardest part of this assignment: You will need to implement support for `map` and `apply` in the compiler. These are not primitives, and they cannot be handled by syntax transformations (like `cond->if`). You have to generate code for them directly. You dispatch on each of them in the main dispatch procedure of the compiler. But in this case, there is no syntax transformation. You handle it like `compile-sequence` or `compile-assignment`, generating code directly.

   - We only need the simple versions of each. That is, they each take a procedure and *one* list of arguments.

   - Here is one idea that might help you in compiling `apply`:

     An apply expression looks like this:

```
(apply proc_exp param_list)
```

So generate code as follows:

1. Generate code to compile the `proc_exp` into the `proc` register. Store this code in a local variable named `proc-code`.
2. Generate code which when executed will evaluate the `param_list` and place the resulting list of evaluated expressions in the `argl` register. Store this code in a local variable named `operand-codes`.

   Note that what is passed in as `param_list` might not look like a list. It might be a function call that evaluates to a list. It might even be a variable that evaluates to a list. So you just have to generate code to evaluate `param_list`. That code, when executed, will produce a list of evaluated arguments, and that's what you need for apply.
3. When these codes have been executed, the procedure is set up properly, so all you have to do is execute the procedure code itself. The code for this is generated by `compile-procedure-call`. Hence you put together the following three pieces of code:

   ```
   proc-code
   operand-codes
   (compile-procedure-call ...)
   ```
4. Of course, when I say "put together", I mean you have to use something like `(preserving ...)`.

Something like this should work just fine, and it's very short.

- Handling `compile-map` is probably a bit harder. You will want to give some careful thought to exactly what you want your code to do. See if you can write out a design, and then use that design when you are writing the code. If you get stuck, by all means send me email, and give me all the information you have, starting with your design. Please give me a simple test case showing something that goes wrong, and also send me all the files you have modified, so I can see what's going on and try things out myself.
- You may need to add additional registers to the register machine. I needed two extra registers when I did this. Those are the registers `temp1` and `temp2`. If you need to add extra registers, add them to the machine description in `machine-shell.scm`, and also add them to the list `all-regs` in `compiler.scm`. You probably won't need to add any more, however, and in fact, you quite likely won't need the ones I added either. (Very few people need them, and in retrospect, I shouldn't have needed them either.)
- I'm guessing that the most difficult part of this will be managing the saving and restoring of registers, using `preserving` and so on. Give yourself plenty of time for debugging.

  The debugging facility that you added to `regsim.scm` can help you a lot. Adding "print" statements (i.e., `display` expressions) in the compiler or the generated code can also be useful.

  One thing to keep in mind: If you put `display` expressions in your `compiler.scm` to aid in debugging, remember that you will not see their output on the monitor. All the output from the compiler goes to the file `*.cmp`. So that's where you should look for the output from those expressions. Of course that output will be intermixed with the rest of the compiler output, so be sure to identify the output in some easily recognized way.

In debugging a problem, spend whatever time it takes to cut the problem down to an absolutely minimal size. Usually once you do this, it is easy to see where the problem is coming from. Many times students throw up their hands and ask me for help debugging without having done this. So either I refuse, or I end up spending an hour or more cutting their problem down until it is quite obvious what is going wrong. You should learn to do this, if you haven't already; it's an important technique and can save you a lot of time. (And it will save me a lot of time also, so I would really appreciate it.)

Another piece of advice about debugging, assuming your compiler is actually generating code: once you have cut the problem down to an absolutely minimal size, look at the generated code (i.e., the ".cmp" file). You can often see what is going wrong pretty quickly when you see something strange in that file.

When you finish this, you should be able to compile `s450.scm` into `s450.cmp` and execute that file using `machine-shell.scm` to get a running Scheme interpreter. Of course, as the book points out, because of the multiple levels of interpretation going on, this interpreter will run rather slowly, but it should run well, and you should experiment with it to see that it does.

3. In `notes.txt` please describe clearly how you approached this work, any particular difficulties or decisions you had to make and how you resolved them. In addition, if you have any thoughts on additional things you would like to have done (if you had time, I know...) please put them in also.

   Here's something to think about: I wrote above that `apply` and `map` cannot be handled by syntax transformations. Can you explain why that is true? If you can, put your reasoning in `notes.txt`.