CS450: Structure of Higher Level Languages

Assignment 9

Due: Sunday, December 20, 2020, by Midnight

Taken from assignments by Profs. Carl Offner and Ethan Bolker

Making Changes to the Explicit-Control Evaluator

In this assignment you will make some small modifications to the explicit-control evaluator presented in Chapter 5 of the text, and experiment with enabling and disabling tail-recursion.

The assignment is not easy and will take you a fair amount of time. But please note that of the 5 problems, the last three are very straightforward and probably won't take very much time at all. You can do these problems in any order.

Start by copying your `regsim.scm` file into the same location as the other files handed out on the course webpage, since it is loaded by `eceval-support.scm`

You may end up making edits to any of these files (with the exception of `load-eceval.scm`; you won't need to make any changes to that file). I will collect them all. I will also collect a file `notes.txt`, as usual.

Remember that you invoke the explicit-control evaluator by loading `load-eceval.scm`. For instance, from the Racket window you can just type

```
load-eceval.scm
```

If `eceval` exits to the underlying Scheme because of an error, you can restart `eceval` from within Scheme by simply typing

```
(start eceval)
```

You should do either problem 1 or problem 2. You don't have to do both. Pick either one. If you do them both, I will count the second one as extra credit.

I personally found question 2 somewhat easier, but your experience may be different.

You must do problems 3, 4, and 5.

1. The footnote on page 549 points out that the dispatch could be written in data-directed style, as we did in `s450.scm`. Implement this.

   Some hints: conceptually this is really very similar to what we did in `s450.scm`. You create a table to manage the special forms, just as you did for `s450.scm`. The difference is this: In `s450.scm` the `cdr` of each pair was the lambda expression that implemented the special form. But now the `cdr` of each pair is the label in `eceval.scm` at which the code implementing that special form is found. So `install-special-form` wants to be called like this, in `eceval-support.scm`:

   ```
   (install-special-form 'quote 'ev-quoted)
   (install-special-form 'set! 'ev-assignment)
   ```

   and so on. `eceval.scm` will use a built-in operation `type-of`, which is just as trivially simple as it was in `s450.scm`. You will also need to check to see if a "bare symbol" is a special form name, so you will

probably need a built-in operation in `eceval.scm` for this purpose. Don't be surprised if you end up having two jumps to the same location; at least, that's what happened when Prof. Offner implemented this.

When he implemented this, he found it necessary to allow instructions in `eceval.scm` of the form

```
(assign val (label (reg val)))
```

That is, in the expression (label xxx), xxx must be able to be a register reference rather than a literal. To do this you have to add some code to `regsim.scm`. This happens in `make-elementary-exp`. (**Note:** if you don't need to do this, that's perfectly fine. I'm curious to see how different people do this. I just wanted to warn you of something that might come up and how to deal with it).

2. Exercise 5.24 (page 560). This exercise asks you to add support for the special form `cond`. In the previous Exercise 5.23 (which I am not asking you to do), the book suggests you "cheat" and use the syntax transformer `cond->if` in `syntax.scm`. Please don't do this, and don't use `expand-clauses`, either. Make `cond` a special form with its own machine code in `eceval.scm` and its own label `ev-cond`. You will of course need some of the selectors such as `cond-clauses` from `syntax.scm` as built-in operators; just remember to add them to the list of built-in operators in the machine. You may want to add some more selectors of your own; feel free to do this.

   You will want to be careful about considering which registers need to be saved and restored when implementing this. Present your reasoning in `notes.txt`.

   My comment (Nurit Haspel): Regarding this, to save yourself a lot of headache, remember that the machine has only **one** stack, but many registers need to be saved and restored. This means they are all saved to and restored from the same stack. The monitored stack can come in handy. You will need to carefully consider the order in which you save and restore registers so that it will be compatible with the LIFO order of the stack. Look at other examples for reference, and remember that in the end of it all, you need to `(goto (reg continue))`, which indicates where the program was before you started. Save it at some point in the beginning.

3. Exercise 5.26 (page 564). For clarity, call this function (which the book calls factorial) `fact-i` (the "i" for "iterative"). Be sure to answer the questions in parts a and b.

   Note that this exercise does not involve writing any code. You just have to experiment with the machine.

4. Exercise 5.27 (page 564). For clarity, call this function (which the book also calls factorial) `fact-r` (the "r" for "recursive").

   Like the last exercise, this exercise does not involve writing any code. In both exercises, you should collect data at least for values of n from 1 through 5.

5. Exercise 5.28 (page 565). This requires changing some of the code in `eceval.scm`, but the book shows you exactly what to do. Add the code anywhere you want, just so long as it won't be "fallen into", and just leave the label `ev-sequence` for the new code commented out. Then to try the new code for this exercise, comment out the old label `ev-sequence` and uncomment the new one.

   Please note that this problem is similar to the two previous ones combined, so you will need to report just as much data and just as many formulas as in those two problems combined.

A word about problems 3, 4 and 5: You are asked to write some simple formulas. You have all taken algebra (and calculus, for that matter) and you should all know how to write a simple formula cleanly. For instance, suppose you come up with some expression such as

$2(n-3)+11$

This is not what you want to write. You want to simplify this, so you write

$2n+5$

And you do not write

$2*n+5$ **!!! WRONG !!!**

That's not mathematical notation. That is what you might write if you were writing code in some computer language. But you are not writing code in some computer language. You are writing an algebraic expression.

Please be careful about this.