CS450: Structure of Higher Level Languages

Fall 2020 Assignment 6

Due: Nov. 25 by midnight, on Gradescopes

Taken from assignments by Profs. Carl Offner and Ethan Bolker

The Metacircular Evaluator

This assignment consists of reading, understanding, and modifying the metacircular evaluator in Chapter 4 of Abelson and Sussman. The assignment has lots of pieces. Be sure you start early. Moreover, some of the things you need to do are more subtle than you may first think.

Guidelines

Start with the handout file `s450.scm`. That file contains all the relevant code from A&S, with small changes so that it runs on Dr. Racket. The name of the evaluator has been changed to `s450`. The invoking function is now `(s450)`, not `(driver-loop)`. This is the file we have been discussing in class.

You will create two files (which I will collect):

- `s450.scm` . This will be the (commented) scheme code that you write. It will be an edited version of the original `s450.scm`.
- `a notes file:` This is for more discursive comments and discussion about the decisions you made in implementing the code for these problems, how you tested the code, and anything else that might be relevant. It will be graded manually.

IMPORTANT: Testing your code will be crucial. Test everything as you build it. Do not wait till you have finished coding the assignment to start testing. Keep careful notes. You will undoubtedly be surprised by some things, or have difficulty understanding what is going on in some cases. You should report all of this in `notes.txt`.

1. (You should have already done this.) Play with the s450 evaluator:

   ```
   ==> (s450)
   s450==> ...
   ```

   or

   ```
   ==> (load "s450.scm")
   ==> (s450)
   s450==> ...
   ```

   See how much of real Scheme is there. Write in your note file about what you found.

2. Rewrite `xeval` so that the handling of special forms is data directed. (This is Exercise 4.3 on page 374.) For each special form (`lambda`, `set!`, `cond`, `define`, `quote`, ...) there is an appropriate semantic action – a scheme procedure A&S have already written. If you put those actions in a lookup table keyed by the symbol representing the special form then the `cond` in `xeval` will be much shorter, with a single table lookup for almost all the cases now listed separately. And you will be able to add special forms to `s450` without editing `xeval`, which will now look something like this (in pseudocode):

```
let action = (lookup-action (type-of expression))
    if lookup succeeded
     invoke action, passing it the expression and the environment
     else cond ... ;; check for a few other types of expressions
```

You should write scheme functions like `lookup-action` and `type-of` to hide the lookup table implementation. (There are various ways to build a lookup table. You might want to review Section 3.3.3 (pages 266–271) for some ideas. We also used it in the second part of HW4, for example). For clarity below, we will sometimes refer to this lookup table as the "special form table", since that really is what it is.

There's nothing fancy going on here. And `type-of` is an extremely simple procedure that just takes the expression and tells you what special form it represents. It's that simple.

Please note that this lookup table for special forms is not part of the environment. It's completely separate. The only thing that is managed by the environment is variables, and a special form name is not a variable.

To insert a new special form in the special form table, write the procedure `install-special-form` and call it this way (for example):

```
==> (install-special-form 'set! (lambda (exp env) ... ) )
```

Since install-special-form is not itself a special form in scheme, its first argument will be evaluated, and therefore generally needs to be quoted, as in the example above, to prevent this. (Indeed, it is possible to call `install-special-form` with the first argument being an unquoted expression that evaluates to the symbol for the form you are installing.) The second parameter, action, should be a lambda expression (or something that evaluates to a lambda expression) that takes two parameters, `exp` and `env`. `xeval` in `s450` will arrange to pass those parameters when action is called. Since `install-special-form` is not a special form, that lambda expression will be evaluated in the environment in which `install-special-form` is called before being passed to the body of `install-special-form`.

Since `install-special-form` is called only for its side effect, it returns no useful information. However, to avoid generating garbage, have it return the name of the special form being installed.

Your implementation should ensure that a new special form cannot be installed using the name of a variable that is already defined.

Similarly, a special form (that is already in the special-form table) cannot be "reinstalled" using `install-special-form`.

Once you have written `install-special-form` you can save yourself lots of time typing at the `s450==>` prompt to test your code by installing `load` as a special form. The code you need is in the handout `load.s450`.

If you do this, **please** include the text of `load.s450` in your file `s450.scm`. Don't `(load "load.s450")`.

3. Clean up the way primitive procedures are installed in `s450`: write a procedure `install-primitive-procedure` that can be called like this:

```
(install-primitive-procedure <name>  <action>)
```

For instance,

```
(install-primitive-procedure 'car car)
```

This procedure should insert the definition of the primitive procedure directly in the global environment. Make sure that it is impossible to reuse the name of a special form for the name of a primitive procedure.

Note that as before, since `install-primitive-procedure` is not a special form, the name generally needs to be quoted.

As in the case of `install-special-form`, this procedure is called purely for its side effect, so it does not return a useful value. However, to avoid generating a lot of garbage, have it return the name of the primitive procedure being installed.

This is important! After you have done this, change the startup code so that all the primitive procedures are created by calls to `install-primitive-procedure`. Then be sure to delete the original code that defined the primitive procedures. So in particular, you should not have the variables

```
primitive-procedures
primitive-procedure-names
primitive-procedure-objects
```

in your code. Please make sure they are no longer in your code; I'm going to check for that. If you don't understand what I mean here, please get in touch with me. This is a small thing you need to do, but if you don't do it, my tests just won't work.

Then install + and some other scheme procedures of your choice as primitive procedures in s450. Note that a primitive procedure in s450 does not have to be primitive in the underlying scheme.

4. In s450 now, entering the name of a special form at the prompt causes a crash. Fix things so that instead a message is generated, like this:

```
s450==> if
Special form:  if
```

5. Fix things so that the name of a special form cannot be redefined or assigned to. That is, we should not be able to get away with typing

```
s450==> (define if 3)
```

or

```
s450==> (set! if 3)
```

6. Write and install the following new special forms:

`(defined? <symbol>)`	returns #t iff the `<symbol>` is defined in the current environment.
`(locally-defined? <symbol>)`	returns #t iff the `<symbol>` is defined in the first frame of the current environment.
`(make-unbound! <symbol>)`	removes the `<symbol>` binding(s) from every frame in the current environment.
`(locally-make-unbound! <symbol>)`	remove the `<symbol>` binding from the first frame of the current environment.

(These last two special forms constitute Exercise 4.13 on p. 380 with the issue raised there resolved.)

`locally-make-unbound!` should not report an error if the `<symbol>` is not locally bound to begin with. There is just nothing to do in that case.

Design a way to test these four new special forms. **Hint:** in order to create a situation in which the local and global versions of your new special forms should behave differently, you will have to build some environments with more than one frame. The only way to do that from the s450 prompt is by applying a procedure. (And see the note above about loading s450 test files.) For instance, you might try defining something like this within s450:

```
(define f
   (lambda (a b)
     (display (locally-defined? a))
     (display (locally-defined? b))
     (locally-make-unbound! a)
     (locally-make-unbound! b)
     (display (locally-defined? a))
     (display (locally-defined? b))
     )
   )
```

and then executing (f 3 4). What should happen? And there are more complex expressions you could create. You should do that as well. And you should write about this in your notes file, of course. **Notice:** `display` is a primitive procedure. You will need to install it for the above example to work. By now I hope you know how to do that!

Some Friendly Warnings

- My testing script relies heavily on `install-special-form` and `install-primitive-procedure`. So you have to make sure that you have implemented them. If you haven't, I won't be able to test your code. **This is important.** I don't care how good the rest of your code is. **If these two procedures don't work, nothing else will make sense**. If you need help, ask me for it, and please ask early. You will be making extensive changes to the file `s450.scm`; please make frequent backups, and make sure your code is clean and understandable. This is by no means a trivial matter of last-minute cleanups: you will probably end up moving large amounts of code from one part of the file to another, and introducing whole new sections of code. Your code (and your notes file) has to be written so that:

  - You can understand it now.
  - I can understand it now.
  - You will be able to understand it in six months. (And don't kid yourself: this is a stringent test, reminiscent of real life situations.)

  You will be using your code as the starting code for the next assignment (Yes! I really meant that!), so it has to be absolutely clearly written.

  And to say again something I've said before: when you write this code (and these notes), don't imagine that you are the student and I am the teacher. Instead, imagine that you are a senior programmer in some company who has developed this code, and that you are about to move on to a different project. Imagine that I am a junior programmer who has just been hired to take over your code. I don't understand very much about it (or anything else) at all, and the only thing I have to go by is the notes and comments that you have written, as well as the code itself, which therefore also has to be clear and well-formatted.

- Finally, one thing Prof. Offner have noticed in the past: Some people seem to think that because we are storing the actions for special forms in a special forms table, then we must also be storing the actions for primitive procedures in a primitive procedures table. This is absolutely false! Where are primitive procedures stored? Make sure you know the answer to that question and put it in your notes file (I'm serious. I'll take points off if it's not there).