CSSE 304 Assignment #17 (4th interpreter assignment) updated for Spring, 2015

This group assignment is to be done by your interpreter team.

- Same turnin instructions as the previous assignments.
- Many of the test-case points will be for "regression tests" over language features from the previous assignments. Another chance to get credit for those things, and to make sure that adding new features did not break old ones.
- For each interpreter assignment, there will be a brief participation survey on Moodle.

1. (200 points) Add additional syntax to your interpreted language.

Summary: The major new features to be added are:

- `set!` for local variables
- `define` (top-level only), including definitions of recursive procedures. You do not need to support the `(define (a b c) e)` form or any other forms that create procedures without using an explicit `lambda`. You do not need to support local defines.
- any additional primitive procedures needed for the test cases
- `set!` for global variables.
- `reset-global-env`
- reference parameters

If `top-level-eval` or something that it calls creates a global variable or changes the value of a global variable, that change stays in effect through subsequent calls to `eval-one-exp`, unless `reset-global-env` is called (or unless evaluation of a later expression changes it again).

The purpose of `reset-global-env` is to handle the case where your interpreter does something wrong and messes up the global environment when evaluating one of your/my test cases. If I call `(reset-global-env)`, I should then be able to continue with the rest of the test cases, without your score being adversely affected by the evaluation of the bad (for you) previous test case. Some of my test cases may call this procedure before calling `eval-one-exp`. **Do not fail to implement it**.

I suggest that you thoroughly test each additional language feature before adding the next one. Augmenting `unparse` whenever you augment `parse` may help you with debugging.

Details of some of the above bulleted items:

Write the zero-argument procedure **`reset-global-env`**, which is a regular Scheme procedure (not something interpreted by your interpreter). The code I have given you below is intended to clarify its function, not to make you rewrite your interpreter. You will need to adapt it to your particular code. You may not already have the `make-init-env` thunk, but it should be simple to modify your `(define init-env ...)` code to create it if you want to follow my approach.

```
(define reset-global-env
   (lambda () (set! global-env (make-init-env))))
```

Note that, as described in EoPL, `set!` is used only to change the values stored in existing bindings, not to create new bindings (*Chez* Scheme and some other implementations allow the use of `set!` to create new bindings, but your interpreter does not have to allow this).

When you add `define` to your interpreted language, you are only required to add top-level **`define`**, as in http://scheme.com/tspl4/further.html#./further:h1 . Here is the relevant part of the grammar:

```
<form> ::=  <definition> | <expression>
<definition> ::=  <variable definition> | (begin <definition>*)
<variable definition> ::= (define <variable> <expression>)
```

Note that Scheme's `define` has a very different meaning (and restrictions on where it can be used) when used inside a `letrec`, `let`, or `lambda`; your interpreter **is not required** to implement this. To handle top-level `define`, you probably will want to modify the `top-level-eval` procedure so it uses `cases` to determine whether the form is a `define` or not; this procedure processes definitions and expressions (the latter by sending them to `eval-exp`, the former by evaluating the expression part *via* `eval-exp`, then modifying the global environment). Definitions (and only definitions) change the global environment `(set!`

may change the *value* of something in the global environment, but not the actual binding of the variable, which is a reference). Note that the global environment is "special," in the sense that all other environments are lexical, but the global environment is dynamic.

Reference parameters. Scheme always passes parameters to procedure calls **by value**. An alternative, calling **by reference**, is described in EoPL. What you are to implement is a new syntax that gives the creator of a procedure the ability to specify whether each formal parameter is a "by-value" parameter or a "by-reference" parameter. If a parameter is a single symbol (as usual), it is by-value. If it is `(ref sym)` where `sym` is a symbol, then it is by-reference. This will require modifying your parser (and most likely modifying some of your datatypes)

The following code samples might illuminate the meaning of (ref x):

```
--> (let ([a 3]
          [b 4]
          [swap (lambda ((ref x) (ref y)) ; both x and y passed by reference
                  (let ([temp x])
                  (set! x y)
                  (set! y temp)))])
     (swap a b)
     (list a b))
(4 3)
--> (let ([a 3]
         [b 4]
         [swap (lambda (x y)  ; both x and y passed by value
                 (let ([temp x])
                   (set! x y)
                   (set! y temp)))])
     (swap a b)
     (list a b))
(3 4)
--> (let ([a 3]
          [b 4]
          [swap (lambda ((ref x) y) ; only x passed by reference
                  (let ([temp x])
                    (set! x y)
                    (set! y temp)))])
      (swap a b)
      (list a b))
(4 4)
```

I do not plan to spend class time on this aspect of the interpreter. I want your group to read about it and be creative in figuring out how to implement it. Some of the ideas in the book may be useful, but there are other possible approaches that are likely to work for this.

Extra-credit problem: 40 points.

You can only earn these points if you get at least 140 points on the regular part of the assignment.

Modify your parser so it generates lexical-address information for local variable uses and references. Modify `apply-env` for local environments so that it uses this lexical address info to go more directly to the location of a variable without having to actually compare the variable to symbols in the environment. This should make the lookup time for a local variable be Θ (lexical depth), since once we get to the correct local environment, the lookup of the value in the vector will be constant time when we already know the position. (The original `apply-env` implementation is Θ(position + lexical depth) Incorporate these things into your interpreter. **If you do this, please send me an email saying that you have done it.** It must work for both variable uses and for variable references (i.e. in `set!`).