CSSE 304 Assignment #13 (interpreter milestone #1)

Deliverables: Your code (submit to PLC server). When you submit, include teammates' usernames.
A13 participation survey (on Moodle, due the day after the A13 late day date).

This is a **team assignment** (possibly with a few solo or pair groups). My intention (for this and other interpreter assignments) is not that you will divide-and-conquer, but that you will instead get together to do the work (i.e., pair/trio programming) so that everyone is involved in the solutions of all parts, and everyone understands all of it. I expect everyone to understand all of the code that your team submits, and that you will do your part to make sure that all team members understand it.
For some of the interpreter assignments, including this one, there will be a brief survey on Moodle in which you will be asked to verify that this is the case.
If you have taken this course before, you are not allowed to use code that you wrote with a different partner in a previous term. And of course you are not allowed to submit code that was written by someone who is not in your group.

When you submit your code to the PLC server

- Only one group member should submit your final version; it will show up as a submission from everyone if you include all usernames on the PLC server's submission page (only necessary for the final submission that is to be graded).
- If your interpreter code is distributed over several files (I have provided both single-file and multi-file starting code):
  a. Your main file should be `main.ss`. It should load the other files. The argument to each `load` statement should be just a filename, not a full pathname. There is an example of this in the multi-file starting code.
  b. Create a `.zip` file containing your source files, including `main.ss`. You do not have to include `chez-init.ss` in your `.zip` file, but it is okay if you do. **Do not include any *folders* in your `.zip` file!**
  c. Submit that `.zip` file to the A13 assignment on the server.

Late day usage for pair assignment: Perhaps (on this or a future interpreter assignment) one partner (A) will run out of late days while the other partner (B) still has late days left. If this happens, partner A's score will be based on the best submission before the due date, while Partner B's score will be based on the best submission before the late day deadline. If you are a potential Partner B in this situation, the best thing you can do for both of you is to push your partner to work with you early so you can submit this interpreter milestone early; then partner A will have a late day to use if you both need it for a later assignment.

Collaborative test case development: If you think you have good test case(s) that cover things (or combinations of things) that the current test cases don't cover, please post them in the *interpreter project* folder on Piazza. I may add some of your test cases to the ones that the grading program uses. If I use your test case(s), I will give you some bonus homework points. Even if I don't use your new tests, it will be good for everyone to have a richer test bed.

(120 points) programming problem
This assignment is not conceptually difficult, but there are many details to understand and to implement. The simple parser and interpreter presented in class can be obtained from

- http://www.rose-hulman.edu/class/csse/csse304/202110/Homework/Assignment_13/A13-multiple-files.zip (a separate file for each major part of the code, perhaps easiest for the development process), or
- http://www.rose-hulman.edu/class/csse/csse304/202110/Homework/Assignment_13/all.ss (a single file containing all of the starting code, easiest to submit, because you do not have to make a ZIP file each time)

Your first job is to understand the given code, and then you will add features to the given parser and interpreter. You should substitute part or all of your own A11 parser for the simple parser that I provided. Some of the code that you need to add to your interpreter will be done in class or is in the EoPL book. Don't just blindly add it to your code; instead be sure that you understand everything as you are adding it.

The interpreted language: The textbook describes an alternate, Pascal-like syntax for the interpreted language. The authors' rationale is that you are less likely to get confused between the interpreted language and the implementation language (Scheme) in which you implement the interpreter. However, I think you can handle interpreting Scheme-like syntax without much confusion, and the benefits of writing a Scheme-like interpreter outweigh the disadvantages. For one thing, you can be sure that your interpreter is getting the correct answer for most expressions, because you can simply input the same expressions

directly into Petite *Chez* Scheme and see what Scheme returns. Thus, whenever an exercise in the book describes a different syntax, you should use standard Scheme syntax instead.

Summary: The major features that you are to add to the interpreted language in this assignment are:

- additional primitive procedures (the ones that are needed in order to evaluate the A13 test cases)
- literals
  - numbers
  - Boolean constants `#t`, and `#f`
  - `quote`d values, such as `'( )`, `'(a b c)`, `'(a b . (c))`, `'#(2 5 4)`
  - string literals, such as `"abc"`
- `if` two-armed only: i.e., `(if test-exp then-exp else-exp)`. You'll add one-armed `if` later.
- `let` (not `let*` or named `let` yet; those will appear in later assignments; don't remove them from your parser, though)
- `lambda` (just the normal `lambda` with a proper list of arguments); variable-argument `lambda` gets added later.
- `rep` and `eval-one-exp` (two alternative interfaces to your interpreter, described below)
- I suggest that you thoroughly test each feature before adding the next one. Augmenting `unparse-exp` whenever you augment `parse-exp` is a good idea, to help you with debugging. However, this is not a requirement.

A more detailed description of what you are to do:

For each feature in this and subsequent interpreter assignments, the syntax and semantics should be the same as Scheme's unless the assignment documents specify otherwise.

Add the Boolean constants `#t`, and `#f` to the interpreted language, along with quoted data, such as lists and vectors. Also add string literals such as `"abcd"`. You do not need to add any string-manipulation procedures yet, but it will be nice to at least be able to use strings for output messages. Add vector literals (as with strings, Scheme will do the "real" parsing; your `parse-exp` simply needs to call `vector?` to see if an expression is a vector).

Add `if` expressions to the interpreted language. Most of the code is in the textbook, but you will have to adapt it to recognize and use Boolean and other literal values— in addition to the numeric values. In the book's interpreted language, the authors represent *true* and *false* by numbers. In Scheme (and thus in your interpreter), any non-*false* value should be treated as *true*. The number 0 is a *true* value in Scheme (and in your interpreted language), but 0 is the false value in the textbook's language).
Note: Recall that `if` has two forms, with and without an "else" expression. Your interpreter must eventually support both, but for this assignment only the "with" version (i.e., “two-armed” `if`) is required for this assignment.

Add several primitive procedures including `+`, `-`, `*`, `/`, `add1`, `sub1`, `zero?`, `not`, = and < (and the other numeric comparison operators), and also `cons`, `car`, `cdr`, `list`, `null?`, `assq`, `eq?`, `equal?`, `atom?`, `length`, `list->vector`, `list?`, `pair?`, `procedure?`, `vector->list`, `vector`, `make-vector`, `vector-ref`, `vector?`, `number?`, `symbol?`, `set-car!`, `set-cdr!`, `vector-set!`, `display`, `newline` to your interpreter. Add the `c**r` and `c***r` procedures (where each `"*"` stands for an `"a"` or `"d"`).
Note: You may use built-in Scheme procedures in your implementation of most of the primitive procedures, as I do in the starting code that is provided for you. At least one of the above procedures has a much simpler implementation, and at least one cannot be implemented by using the corresponding Scheme procedure. You may want to enhance the `prim-proc` mechanism from the provided code so that a `prim-proc` knows how many arguments it is allowed to take, and it reports an error if the interpreted code attempts to apply it with an incorrect number of arguments. Otherwise, this incorrect code will default to a Scheme error message such as "Exception in `cadr`", which will make no sense to the author of the interpreted code. However, the official test cases should only test code that has no errors.

Add `let` (not "named-`let`" yet) expressions to the interpreted language, with the standard Scheme syntax: `(let ([var exp]…) body1 body2 …)`. The `...` in this description is basically the same thing as the Kleene * (like the notation we used in `define-syntax`); it means "zero or more occurrences of the previous item". Every `let` expression must have at least one body. (The parser work should have been done in a previous assignment). It is okay for you to interpret named-`let` as well, but this is not required until a later assignment, and will be easier after we discuss `letrec` implementation in class.
Add code that is similar to that in EoPL 3.2, allowing `let` expressions to be directly interpreted. Recall that after the extended environment is created, the bodies are executed in order, and the value of the last body is returned. We suggest that you test this code and understand it thoroughly before you go on to implement `lambda`.

Add `lambda` expressions of the form `(lambda (var ...) body1 body2 ...)`. See the description of `rep` (short for "read-eval-print") below for a description of what to print if a closure or primitive procedure is part of the final value of an expression entered interactively in the interpreter. Section 3.3 of EoPL may be helpful.

In later assignments, you'll add many syntactic and semantic forms, including `define`, `let*`, `letrec`, `cond`, and `set!`

YOU ARE TO PROVIDE TWO INTERFACES TO YOUR INTERPRETER

A. A normal interactive interface (read-eval-print loop) `(rep)`

This interface will be used primarily for your interactive testing of your interpreter and to facilitate interaction if you bring the interpreter to me for help. The user should load your code, type

```
(rep)
```

and start entering Scheme expressions. Your read-eval-print loop should display a prompt that is different from the normal Scheme prompt. It should read and evaluate an expression, print the value, and then prompt for the next expression. If the value happens to be (or contain) a closure or prim-proc, your interpreter should not print the internal representation of the procedure, but instead it should print `<interpreter-procedure>`. For debugging purposes, you might want to write a separate procedure called `rep-debug` that behaves like `rep`, except that it displays the internal representation of everything that it prints, including procedures. Using `trace` on your `eval-exp` procedure can be a big help,but be prepared to wade through a lot of output!

If the user types `(exit)` as an input expression to your interpreter, (some later version of) your interpreter should exit and simply return to the Scheme top-level; for now it is okay if your interpreter "crashes". What happens when the user enters illegal Scheme code or encounters a run-time error? Ideally, the interpreter should print an error message and return to the "read" portion of the "read-eval-print" loop. Think about how you might do that. **But we do not expect you to do it for this assignment.** Again it can just crash your interpreter. A sample transcript follows:

```
> (rep)
--> (+ 3 5)
8
--> (list 6 (lambda (x) x) 7)
(6 <interpreter-procedure> 7)
--> (cons 'a '(b c d))
(a b c d)
--> (let ([a 4])
      (+ a 6))
10
--> ((lambda (x)
      ((lambda (y)
        (+ x y))
       5))
     6)
11
--> (let ([t '(3 4)])
      (if (< (car t) 1)
          7
          (let ()
            (set-car! t (+ 1 (cadr t)))
            (cons (cadr t) (car t)))))
(4 . 5)
--> (((lambda (f)
       ((lambda (x) (f (lambda (y) ((x x) y))))
        (lambda (x) (f (lambda (y) ((x x) y))))))
     (lambda (g)
       (lambda (n)
         (if (zero? n)
             1
             (* n (g (- n 1)))))))
   6)
720
-->(exit)
```

B. single-procedure interface: (`eval-one-exp parsed-expression`)

In addition to the `(rep)` interactive interface, you must provide the procedure `eval-one-exp`. The code below is intended to clarify this procedure's function, not to make you rewrite your interpreter. You will of course need to adapt it to your particular program. It may be possible to arrange things so that your `rep` procedure can call `eval-one-exp`.

```
(define eval-one-exp ; you'll add a "local environment" argument.
  (lambda (exp) (eval-exp (parse-exp exp)))))
```

```
>(eval-one-exp '(- (* 2 3) (* 6 3)))
-12
>
```

Note that `eval-one-exp` does not have to be available to the user when running your interpreter interactively. It only has to be available from the normal Scheme prompt. The interpreter test cases will call this procedure. Testing of your `rep` loop will need to be done by hand.

Important notes about local environments and the global environment.

1. Every local environment is static (we cannot add variables to a local environment after its initial creation)
2. The global environment is dynamic (we can use `define` to add variables to it).
3. Thus it does not make logical sense to have any local environments be extensions of the global environment (although the difficulties that arise from doing this will not show up until A17).
4. Here is another, more practical and immediate reason for not having local envs extend the global env: **debugging!**
    a. You are likely to want to trace `eval-exp` and other procedures that take a local env as on argument.
    b. If each step in the trace includes a printout of the entire global environment, it may take a long time for your machine to display the trace, and an even longer time for you to scroll back and find the part of the trace that you are looking for.
    c. If you come to me for debugging help, I will probably ask you to show me a trace. If I see the above phenomenon, it is likely that I will ask you to fix it by separating the global environment form the local environments, then come back to me for help after it is fixed.
5. **Bottom line:** My simple advice on having local environments extend the global environment: **Don't do it!** When you make a top-level call to `eval-exp`, you should pass the empty environment, not the global environment. This may require changing some of the code that I gave you, but it will be worth it in the long run!