**CSSE 304 Assignment 16 (interpreter assignment #2) Updated for Spring, 2014**

This is a group assignment. When you submit it

* Include the usernames of other group member(s) on the PLC server submission page.
* If your interpreter code is distributed over several files  
  a. Your main file should be main.ss. It should probably just load the other files.

b. Create a .zip file containing all of your source files. You do not have to include chez-init.ss or any file that it loads in your .zip file.

c. Submit this .zip file for the A16 assignment.

For each interpreter assignment, there will be a brief participation survey on Moodle.

**(100 points) programming problem**

Note that some of the test cases will be the same as in the previous assignment, in order to give you a chance to get some of the cases that you missed in that assignment, as well as to do regression testing.

**Summary:** The major features you are to add to the interpreted language (in addition to those you added in the previous assignment) are:

1. All of the kinds of arguments (proper list, single variable, improper list) to lambda expressions.
2. One-armed if: (if <exp> <exp>)
3. Whatever additional primitive procedures are needed to handle the test cases I give you. This applies to subsequent interpreter assignments also.

syntax-expand (allows you (but not the user of your interpreter) to introduce new syntactic constructs without changing the interpreter) and additional syntactic extensions (at least add begin, let\*, and, or, cond).

I suggest that you thoroughly test each addition before adding the next one. Augmenting unparse whenever you augment parse is a good idea, to help you with debugging. But after you have syntax-expanded an expression, unparse-exp will no longer give you back the original, unexpanded expression.

You should be able to do the first two items from the above bulleted list in a very short time; most of the credit for this assignment will be for the remaining items, but they will be useless if the first two don't work.

**A more detailed description of what you are to do:**

**Add** the variations of lambda that allow the arguments to be a single symbol or an improper list of variables.

**Add** one-armed if. You may find *Chez* Scheme'svoid procedure useful in this implementation:  
> **(void)**

> (**list (void))**

(#<void>)

> **(list (if #f 1))**

(#<void>)

**Add** the primitive procedures that are necessary to handle the assignment's test cases. The map and apply procedures may be slightly more interesting than most of the primitive procedures from the previous assignment.

**Add syntax-expand.** It isn't really necessary for the interpreter to treat let as a core expression, since it can be implemented in terms of lambda. After an expression has been parsed, pass it to a new procedure (you must write it) called syntax-expand that replaces each parsed let expression in the abstract syntax tree by the equivalent application of a lambda expression (you did something similar in a previous assignment). You should write syntax-expand in such a way that you can add other expansions later (hint: use cases). One benefit of using syntax-expand rather than interpreting let directly is that this and future versions of the interpreter (eval-exp) need not have a case for let, let\*, and, etc., so writing those interpreters should be simpler. Don't forget that syntax-expand may need to expand subexpressions of an expression as well, if those subexpressions also contain syntactic extensions such as let.

You will need to modify eval-one-exp (and perhaps rep as well) to include syntax-expand in the interpretation process. Something like:

(define eval-one-exp

(lambda (exp) (eval-exp (syntax-expand (parse exp)))))))

Think about what the code for syntax-expand would be like if it were written as a pre-processor to the parser, rather than a post-processor. This will increase your appreciation for having the interpreter use abstract syntax trees instead of unparsed expressions!

Note that it is acceptable, and perhaps more efficient, to "mix up" parsing and syntax expansion (having parse-exp call syntax-expand and vice-versa), but it could make debugging harder, so I do not recommend it.

**Modify** your parser and syntax-expand to allow some other special forms, including begin, cond, and, or, let\*, and case (see section 5.3 of TSPL). The basic goal here is that your interpreter now should handle any non-recursive code (that doesn't require define) from the early part of the course. Your version of cond does not need to handle the clauses with the => or the clauses with only a test described in Section 5.3 of TSPL); you only have to do clauses of the forms   
(test exp1 exp2 …) or (else exp1 exp2 …) as described in the same section.

**Add** (to the parser and interpreter, possibly *via* syntax-expand) a looping mechanism that is not found in Scheme: (while test-exp body1 body2 ...) first evaluates test-exp. If the result is not #f, the bodies are evaluated in order, and the test is repeated, just like a *while* loop in other languages. I do not care whether a value is returned, since while should never be used in a context where a return value is expected. You must not use Scheme's define-syntax in your implementation.