CSSE 304 Assignment 14 (interpreter milestone #2)

This is a team assignment. Same submission rules/process as for A13 (including late days policy). Don't forget to include teammate's usernames on submission page.

(100 points) programming problem

Note that some of the test cases will be the same as for A13, to give you a chance to get some of the cases that you may have 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 A13) are:

- All of the kinds of arguments (proper list, single variable, improper list) to `lambda` expressions.
- One-armed `if`: `(if <exp> <exp>)`
- Whatever additional primitive procedures are needed to handle the test cases I give you. This requirement applies to subsequent interpreter assignments also.
- `syntax-expand` allows you (but not necessarily the user of your interpreter) to introduce new syntactic constructs without changing `eval-exp`. Write `syntax-expand` and use it to add some syntactic extensions (at least add `begin`, `let*`, `and`, `or`, `cond`).

I suggest that you thoroughly test each added language feature before adding the next one. Augmenting `unparse-exp` whenever you augment `parse-exp` is a good idea, to help you with debugging. But be aware that after you have syntax-expanded an expression, `unparse-exp` will no longer give you back the original, unexpanded expression. However, `unparse-exp` can still be very useful for debugging.

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. I suggest that you have a separate `expression` variant for the "non-fixed-arguments" lambdas.

Add one-armed `if`. You may find *Chez* Scheme's `void` procedure useful in this implementation:

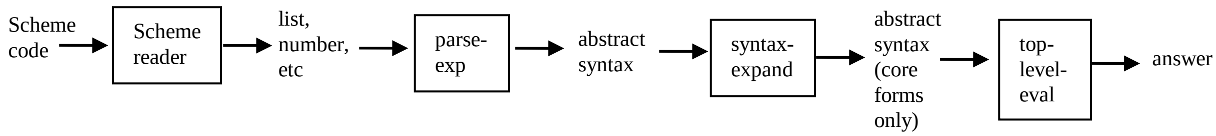
```
> (void)
> (list (void))
(#<void>)
> (if #f 1)
> (list (if #f 1))
(#<void>)
```

Add the primitive procedures that are necessary to handle the assignment's test cases. Implementation of 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 as an application of `lambda`. After an expression has been parsed, pass the parsed expression 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 (the idea is similar to `let->application` from a previous assignment, but this time it is recursive). 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 `eval-exp` need not have cases for `let`, `let*`, `and`, etc., so writing `eval-exp` should be simpler. Don't forget that `syntax-expand` will need to expand subexpressions of an expression as well, if those subexpressions also contain syntactic extensions such as `let`. You do not have to implement "named `let`" until assignment 16.

You will need to modify `eval-one-exp` (and perhaps `rep` as well) to include `syntax-expand` in the interpretation process. For example, you may have something like:

```
(define eval-one-exp
   (lambda (exp)
      (top-level-eval (syntax-expand (parse-exp exp)))))
```



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 this can make debugging harder, so I do not recommend it. First parse the code, then syntax expand, then evaluate. This is a suggestion, not a requirement.

Aside: 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 as in the above code. This will increase your appreciation for having the interpreter use abstract syntax trees instead of raw, unparsed expressions!

Modify your parser and `syntax-expand` to allow some other syntactic forms, including `begin`, `cond`, `and`, `or`, `let*`. The basic goal here is that your interpreter now should be able to interpret almost any non-recursive Scheme code (if it doesn't require `define` or `set!`) 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 interpret clauses of the forms `(test exp1 exp2 …)` or `(else exp1 exp2 …)` as described in the same section. These are the only kinds of `cond` clauses that I have used in class examples.

Add (to the parser and interpreter) 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. We do not care whether `while` returns a value, since `while` should never be used in a context where a return value is expected. For now, you will most likely need to add `while` as a core form in your `eval-exp` procedure.