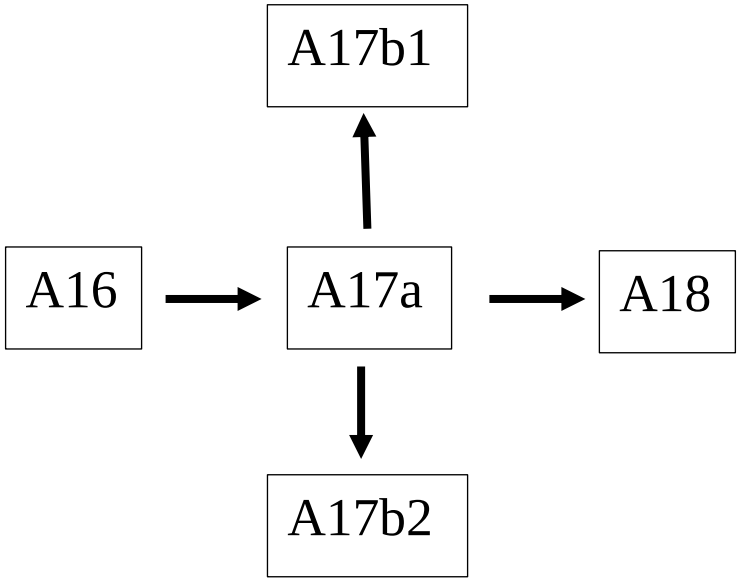
**CSSE 304 Assignment #17 (4th interpreter milestone)**

There are **three** parts to this milestone: A17a, A17b1, and A17b2. They all have their own separate requirements and testcases.
Part b is the first **very** challenging team assignment.

- Same turn-in instructions as the previous assignments.
- Some of the test-case points are "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.

# Code branches:

The end is in sight! First, you will need to complete A17a. Once you're done with A17a, **save a copy** (through Git branches, a .zip file, or anything else where you can revert back to the original A17a). A17b1, A17b2, and A18 will all build on the requirements of A17a, but you aren't required to build them on top of each other. Most students find it easier to build three separate versions, each starting from A17a.




**Part a**. Add additional syntax to your interpreted language. Make sure the old stuff still works.

**Summary:** The major new features to be added are:

- `set!` for local (bound) 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 of `define` that create procedures without using an explicit `lambda`. You do not need to support local `define`s (i.e. `define` inside `let`, `lambda` or `letrec`). Your interpreter must support `define` inside a top-level `begin`.
- any additional primitive procedures that are needed for our test cases. More details below.
- `set!` for global (free) variables. As described in class, you only need to have this work for variables that have already been defined. More details below.
- `reset-global-env`
- If `top-level-eval` or something that it calls creates a global variable or changes the value of a global variable, then the value of that global variable stays in effect through subsequent calls to `eval-one-exp`, unless `reset-global-env` is called (or unless evaluation of a later `set!` 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 we call `(reset-global-env)`, we 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 the test cases may call `reset-global-env` before calling `eval-one-exp`. **Do not fail to implement it**.

I suggest that you thoroughly test each additional interpreter feature before adding the next one. It is not required, but updating `unparse` whenever you update `parse` may help you with debugging.

**Details of some of the above bulleted items:**

1. 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!` in our interpreted language is used only to change the values stored in existing bindings, not to create new bindings (*Chez* Scheme and some other implementations also allow the use of `set!` to create new bindings, but your interpreter is not required to allow this).

2. When you add `define` to your interpreted language, you are only required to add top-level **`define`**, a slight variation on http://scheme.com/tspl4/further.html#./further:h1 . Here is the relevant part of the grammar:
   ```
   <form>            ::=  <definition> | <expression>
   <definition>      ::=  <variable definition> | (begin <definition>+ <expression>*)
   <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 in A17. 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; then `top-level-eval` 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 static (no new variables can be added after it is created) , but the global environment is dynamic.

**When you have finished everything up to this point, you should save a copy of your code, to use for Assignment 18 and the final exam; they will not require reference parameters or lexical address. You should probably also make two new "branches" of your code; one that implements pass-by-reference parameters and one that implements lexical address. You are not required to implement both of those in the same interpreter (but you are allowed to do so if you wish).**

# Part b1: reference parameters

**Reference parameters.** Scheme always passes arguments to procedure applications **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 in a `lambda` expression is a "by-value" parameter or a "by-reference" parameter. If a formal parameter is a single symbol (as usual), it is by-value (i.e. pass to the procedure the value of the argument). If it is `(ref sym)` where `sym` is a symbol, then it is by-reference (pass to the procedure a reference to the argument). This will require modifying your parser (and most likely modifying some of your datatypes). **Something to think about:** What if the actual argument passed in to a by-reference formal parameter is an expression that is not a variable lookup? You'll have to think about what to do in this case.

Note that the use of `ref` here is somewhat like the use of `*` in the type of a parameter in C. However, unlike C, the caller of the procedure with a ref parameter does not have to provide the equivalent of C's `&` when calling a procedure that was created with a `ref` parameter. The interpreter has to do the right thing when the procedure call happens.

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.

## Part b2 Lexical address

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 efficiently go to the location of a local 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(number of variables in all local envs).

Suggested order of procedure calls: `(eval-exp (lexical-address (syntax-expand (parse-exp source-exp)))`.

**Submission** There are three A17 assignments on Gradescope. The first has tests for A17a's new syntax. The second has tests for A17b1's reference parameters. The third has tests similar to A17a's, but you are required to implement lexical addressing.. In order to actually earn the points the server gives you for A17b2, you must correctly use lexical address in your interpreter. We will determine this by checking your code by hand.

# Piazza Q&A from past terms

# (A17 Part a)

### A17: Some ideas and code for implementing set! are in the session 26 slides

One of the TAs told me that some students who came to the F-217 lab yesterday were not aware that this code is in the slides.

As a result, `set!` implementation is something that you should be able to implement and debug in a couple of hours, then move on to the more interesting parts that you have to figure out, such as `define` and pass-by-reference parameters.