Environment Diagrams: A Reference

So long as you retain your spirit of exploration, surely you shall find your way out. This I believe.

Definitions
expression - a question we ask scheme
value - an answer to an expression that we type into scheme; can be stored in an environment
environment - where we store all our answers and also the context in which we evaluate all our functions; made up of a bunch of frames that point to each other; changes
frame - group of bindings, may point to an "enclosing" frame (where to look next if you can't find something in that frame)
bindings - variable names pointing to values
variable name - way of labeling a value
pointer - arrow that associates a name to a value OR a frame to an enclosing frame OR a function with the frame which it was defined in
function - code that can be run with some input and some environment to get an interesting answer
formal parameters - the dummy variables used in functions that get replaced with argument values when the function is called with input
argument expression - an expression that we give to a function as input
argument value - the value of an argument expression
global environment - the environment that we start off with; has the global frame
global frame - the frame that we start off with; bunch of default bindings that let you do primitive stuff
environment diagram - way of keeping track of a changing environment; bane of CS 61AS students everywhere

Rules of Evaluation
These are the rules Scheme follows when you type stuff in to return an answer. **The trick here is to be familiar enough with the rules so that you will follow them blindly.** It might be worthwhile to go through a few examples problems (like the discussion problems) by following the rules step-by-step, and also to bring a copy of the rules with you to the final. If you're not sure what a rule means, see the example to follow and post a Piazza question if you're confused still.

Assignment (define *var expr*)

1. Evaluate *expr* fully.
2. In the current frame, create *var* (if it doesn't already exist) and bind it to the **value** of *expr.*

Looking up a variable *var*

1. Start a the current frame and look for the binding with *var* as the name. We care about what *var* is pointing to.
2. If it isn't there, go to the frame that the current frame points to and look there.
3. If you reach the global frame and you still couldn't find *var*, yell at the programmer.

Mutation (set! *var expr*)

1. Evaluate *expr* fully

2. Follow the rules for *Looking up a variable* for *var*. Instead of returning what *var* points to, we change it to be the **value** of expr.

Creating Functions (define (func …) …) OR (lambda …)

1. Make a function (double-bubble thingie).
2. Point the left bubble to the arguments and body of the procedure
3. Point the right bubble to the current frame (this is where it is defined).
4. ONLY if this is a define statement, make a binding in the current frame of func to the function (draw a pointer from "func" in the current frame to the bubble.)

Evaluating User Defined Functions (func a b … z)

1. Evaluate func. This should lead you to an existing function bubble. This is an important step. This is an important step. **THIS IS AN IMPORTANT STEP.**
2. Evaluate the argument expressions. Remember what the argument values are in your head.
3. Create a new frame and point it to where the function points to (where it was defined).
4. Bind the formal parameters to the argument values you found two steps back.
5. Evaluate the body of the function in the context of this new frame. Whatever the function returns is the answer to the function call.

Example

Now what do these mean? Let's run through some examples.

```
; Example 1
STk> (define (square x)
       (* x x))
STk> (define x 2)
STk> (define x (square x))
STk> x
```

```
; Example 2
STk> (define (make-honorific prefix honorific)
      (lambda (name)
        (if prefix
          (se honorific name)
          (se name honorific) ))
STk> (define jp (make-honorific #f ‘san))
STk> (jp ‘Rohin)
```

```
STK> (define us (make-honorific #t ‘mr))
(us ‘Pedro)
```

Draw the environments for the examples here. Draw a separate diagram for each example.

Here are the complete rules for the environment model:

- Every expression is either an atom or a list
- At any time there is a *current frame*, initially the global frame.

I. Atomic Expressions.

A. Numbers, strings, `#t`, and `#f` are self-evaluating.

B. If the expression is a symbol, find the *first available* binding. (That is, look in the current frame; if not found there, look in the frame "behind" the current frame; and so on until the global frame is reached.)

II. Compound expressions (lists): If the car of the expression is a symbol that names a special form, then follow its rules (see II.B). Otherwise, the expression is a procedure invocation.

A. Procedure invocation

1. Evaluate all the subexpressions (using these same rules).
2. Apply the procedure (the value of the first subexpression) to the arguments (the values of the other subexpressions).
    a. If the procedure is compound (user-defined):
        i. Create a frame with the formal parameters of the procedure bound to the actual argument values.
        ii. Extend the procedure's defining environment with this new frame.
        iii. Evaluate the procedure body, using the new frame as the current frame.

***** ONLY COMPOUND PROCEDURE INVOCATION CREATES A FRAME *****

    b. If the procedure is primitive:
        i. Apply it by magic.

B. Special forms

1. `Lambda` creates a procedure. The left circle points to the text of the `lambda` expression; the right circle points to the defining environment, i.e., to the current environment at the time the `lambda` is seen.

***** ONLY `LAMBDA` CREATES A PROCEDURE *****

2. `Define` adds a *new* binding to the *current frame*.
3. `Set!` changes the *first available* binding (see I.B. for the definition of "first available")
4. `Let` == `lambda` (II.B.1) + invocation (II.A)
5. `(define (...) …)` = `lambda` (II.B.1) + define (II.B.2)
6. Other special forms follow their own rules (`cond`, `if`).