Let (in Scheme)

Last updated Spring 2019.

This guide serves as a review and extension of the Let special form covered in class. While this goes fairly in depth, please refer to lecture slides and discussions first.

Definitions and Background

The `let` Special Form

Let allows you to create local bindings and then evaluate some expression with the bindings defined earlier. All `let` expressions take the form:

```
(let (<binding> <binding> ...) <expr>)
```

Each binding is a list of 2 elements. The first is the symbol that the name will be bound to. The second is an expression (either atomic or combination). Once the expression is evaluated, it is then "bound" to the symbol

Some Examples

Let's take a look at a `let` special form in action!

```
scm> (let ((x 1) (y 2)) (+ x y))
3
scm> (define a 2)
a
scm> (define b 5)
```

```
b
scm> (let ((n (*a b)) (m b)) (* n m))
50
```

How does `let` really work?

Each `let` expression can actually be rewritten as a `lambda` expression call!

- The symbols in the bindings become the parameters of the `lambda` function.
- Thus, the expressions the symbols are bound to become the arguments passed into the `lambda` function call.
- The expression in the `let` becomes the body of the `lambda` function.

The `lambda` equivalent to the expression `(let ((x 1)) (+ x 1))` is

```
((lambda (x) (+ x 1)) 1)
```

A few more examples...

- `(let ((x 1) (y 2)) (+ x y))` is equivalent to `((lambda (x y) (+ x y)) 1 2)`
- `(let ((n (* a b)) (m b)) (* n m))` is equivalent to `((lambda (n m) (* n m)) (* a b) b)`

What can't we do with `let`?

Looking at how every `let` expression has a `lambda` equivalent, we realize that `let` actually has a limitation! From the `lambda` equivalent, when can see that the arguments passed into the lambda function **are not bound to their respective symbols until the `lambda` frame**. In other words, the bindings in a

`let` expression only exist inside the local frame and not in the global frame.

For example, take the follow `let` expression,

```
(let ((foo 3) (bar (+ foo 2))) (+ foo bar))
```

Why won't the previous expression work?

Recall the previous paragraph, which said, "bindings in the `let` expression only exist inside the local frame and not in the global frame". **When making bindings, `let` will lookup symbols in the global frame.**

Thus, the first binding `(foo 3)`, where the value 3 is bound to `foo`, is valid. In the local frame, we now have a symbol `foo`. However, the second binding `(bar (+ foo 2))` will error. When Scheme looks for the symbol `foo`, `foo` doesn't exist in the global frame! If this is confusing, let's take a look at the `lambda` equivalent.

```
((lambda (foo bar) (+ foo bar)) 3 (+ foo 2))
```

The `lambda` expression is defined without error; however, when we try to evaluate the arguments passed into the `lambda` expression, `(+ foo 2)` will error **because `foo` does not exist in the global frame**.

To summarize, we can only reference symbols in the global frame when making bindings in `let`. Bindings cannot refer to each other, because bindings only exist in the local frame.