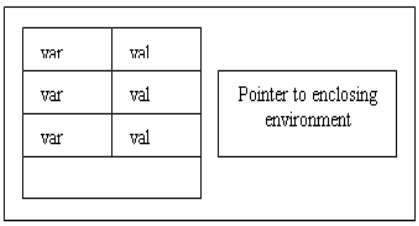
CSSE 304 Day 17-18 Summary We have done some syntax analysis; now we move on toward interpretation.

1. **Main question we'll answer:** How do variable bindings and lexical scoping work in an actual interpreter? What data structures are needed?
2. Logically, an *environment* is a table of variable names (symbols) and their associated values. There is a dynamic ***global environment***, `define` and `set!` alter it.

3. **Local (lexical) environments**: Application of a `lambda`-created procedure, or execution of a `let`, `let*`, or `letrec` creates a *local environment* that holds the local variables and associated values.



4. Evaluate a `let` expression:
    a. Evaluate (in the current environment) the expressions to get the values to be assigned to the let variables
    b. Create a new local environment with bindings for the let variables. The "enclosing environment" pointer points to the current environment.
    c. Evaluate the body of the let in this new environment.
5. Example:

```
(define a 5)
(let ([z (+ a 3)]
      [t 7])
   (let ([y (+ z a)])
      (list a t z y)))
```

6. How do procedures work?
    d. When a lambda-expression is evaluated, what is returned?
    e. What kind of info needs to be stored in a procedure?
    f. What happens when a procedure is applied?
7. A procedure (also known as a **closure**) is created when a lambda-expression is evaluated.
8. A closure consists of three parts. See the diagram above.
9. Note that a lambda expression is *not* a procedure. What is it?

list of formal argument names	code (body of the procedure)	local environment that existed when the procedure was created

10. Procedure application:
    g. The expressions for the procedure and its arguments are evaluated.
    h. A new local environment is created.
        i. Each variable from the procedure's formal parameter list is bound to the corresponding value from the actual argument list.
        ii. The new environment's "pointer to an enclosing environment" is set to point to the local environment that is the third part of the closure.
    i. The body of the procedure is evaluated, using this new local environment. If a variable is not found in local environment or something it points to, look in the global environment.

11. **Simple Example:** Draw diagrams to the right of the code.

```
> (define add2  (lambda (car) (+ car 2)))
 > (add2 17)
```

12. Nested Lambda example:

```
((lambda (x)
   ((lambda (y)
      (+ x y))
    15))
  20)
```

13. Another example:

```
>(define fact
        (lambda (n)
          (fact2 n 1))
>(define fact2
        (lambda (n acc)
      (if (zero? n)
          acc
          (fact2 (- n 1) (* n acc)))))
>(fact 3)
```

14. Evaluate let* expressions

Expand the let* to nested lets and then evaluate.

Evaluate letrec expressions

Create a new local environment, similar to a let environment, except that:

- The "saved environment" pointers of all closures that are bound to the letrec variables point to the new environment, not the enclosing environment.
- Evaluate the body of the letrec in this new environment.

15. letrec example

```
(define odd?
  (letrec ([odd? (lambda (n)
                   (if (zero? n)
                       #f
                       (even? (- n 1))))]
           [even? (lambda (m)
                    (if (zero? m)
                        #t
                        (odd? (- m 1))))])
    (lambda (x)     ; should use eta-reduction here, but
      (odd? x))))   ; diagram is more interesting if we don't!
>(odd? 2)
```

16. Final example

```
    >(define f
       (lambda (x)
         (let ([a (lambda (y z) (+ x y z))])
           (lambda (b)
             (a (+ 5 b) x))))
     >((f 3) 4)
```