Dynamic Binding
vs
Lexical Binding

What is the value of the following expression?

```
(let ([y 3])
        (let ([f (lambda (x) (+ x y) ) ])
                (let ([y 17])
                        (f 2) ) ) )
```

The real question, of course, is What is the value of y in the body of f when we call (f 2)?

```
(let ([y 3])
        (let ([f (lambda (x) (+ x y) ) ])
                (let ([y 17])
                        (f 2) ) ) )
```

a) Scheme, Java and C use *static binding,* also called *lexical binding.* They connect the reference of y to the nearest surrounding declaration of y, which in this case is [y 3].

```
(let ([y 3])
      (let ([f (lambda (x) (+ x y) ) ])
            (let ([y 17])
                  (f 2) ) ) )
```

b) Early Lisp used *dynamic binding*, which connects a reference of y to the most recent declaration of y, which in this case is [y 17].

In Scheme, which is lexically scoped -- a lambda expression evaluates to a closure, which is a triple containing the environment at the time the lambda is evaluated (the surrounding environment) and the parameters and body of the lambda expression.

When we apply this closure to argument expressions we evaluate the arguments in the current environment, make a new environment that extends the closure's environment with the new bindings, and evaluate the closure's body within this new environment.

Think of how this applies to our example:

```
(let ([y 3])
        (let ([f (lambda (x) (+ x y) ) ])
                (let ([y 17])
                        (f 2) ) ) )
```

The outer let creates an environment in which y is bound to 3 and the inner let ([f .....]) is evaluated in this environment. To do this we evaluate the lambda expression in our environment where y is bound to 3. It evaluates to a closure in which which has y bound to 3 in the closure environment. When we call that closure with argument x, y evaluates to 3.

In Java and C, which are also lexically scoped but without lambda expressions, the environments are much more static. At the time a program is compiled the compiler can keep track of the environments and link each variable reference to its declaration and its location on the runtime stack.

Think back to this example:

```
(let ([y 3])
        (let ([f (lambda (x) (+ x y) ) ])
                (let ([y 17])
                        (f 2) ) ) )
```

With lexical scoping we said this returns 5. In dynamic scoping the binding of y that applies in the application (f 2) is [y 17], and the whole expression returns 19.

How would you evaluate lambdas and applications in a dynamically scoped language?

How would you evaluate lambdas and applications in a dynamically scoped language?

a) There is no need for closures; they maintain the lexical environment, which dynamic binding does not use.
b) The value of a lambda expression is just its parameters and body.
c) To apply a procedure to a list of arguments, we extend the *current* environment with the bindings of the parameters to their argument values and evaluate the body in this environment.

Why do (or did) people use dynamic binding?

- It was easy to implement. Indeed, dynamic binding was understood several years before static binding.
- It made sense to some people; the function
  (lambda (x) (+ x y) should use whatever the latest
  version of y.

Why do we now use lexical binding?

- Most languages we use today are derived from Algol-60, which used lexical binding.
- Compilers can use lexical addresses, known at compile time, for all variable references.
- Code from lexically-bound languages is easier to verify.
- It makes more sense to most people.

Here is a simple example of a Python program that illustrates static scoping:

```
def f(a):
    def g(x):
        return x+a
    return g

def main():
    h = f(5)
    a = 45
    print( h(6)) # prints 11, not 51.

main()
```