Implementing Dynamic Binding

What is the value of the following expression?

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

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

a) Scheme, Java, C and MiniScheme 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], so with lexical binding this expression evaluates to 5.

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

b) *Dynamic binding* connects a reference of y to the most recent declaration of y, which in this case is [y 17]. Under dynamic binding this expression evaluates to 19.

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.

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.

To modify MiniScheme to use dynamic binding instead
binding, it is easiest to leave the closures in place and j
the closure-environment. This means we will have to g
proc an argument for the env parameter of eval-exp.
apply-proc now says

```
(define apply-proc (lambda (p args)
                     (cond
                       .....
                      [(closure? p) (let ([params (closure-ids p)]
                                          [bod (closure-body p)]
                                          [cenv (closure-env p)])
                                      (eval-exp bod (new-extended-env params args cenv)))]
```

To use dynamic binding we would change this to

```
(define apply-proc (lambda (p args  env)
                     (cond
                       .....
                      [(closure? p) (let ([params (closure-ids p)]
                                          [bod (closure-body p)]  )
                                          [cenv (closure-env p)])
                                    (eval-exp bod (new-extended-env params args env)))]
```