CSCI 275: Programming Abstractions

Lecture 27: Scoping Methods

Fall 2024

Stephen Checkoway
Slides from Molly Q Feldman

Functional Language of the Week: F#

- Is based *not* on the JVM, but on the .NET Framework that underlies C# and other Microsoft-based languages
- Borrows ideas from the ML family of languages (OCaml, for instance)
- F# versus C#? The founder make a strong argument for F#’s support of concurrent/parallel programming.
  - Interesting interview here! https://www.red-gate.com/simple-talk/opinion/geek-of-the-week/don-syme-geek-of-the-week/

Functional Language of the Week: F#

```
/// Square the odd values of the input and add
one, using F# pipe operators.
let squareAndAddOdd values =
    values
    |> List.filter (fun x -> x % 2 <> 0)
    |> List.map (fun x -> x * x + 1)

let numbers = [ 1; 2; 3; 4; 5 ]

let result = squareAndAddOdd numbers
```

Pipeline Operators (like in R)

```
let result3 = apply1 (fun x -> x + 1) 100

let result4 = apply2 (fun x y -> x * y ) 10 20
```

Also lambdas of course!

https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/functions/

Today (& Friday)'s Goal

Talk about design of a language and how it impacts implementation

- In MiniScheme, you are implementing a certain language that has certain rules
- Many times, we have choices for these rules
- Today & next time: what we *could* and *can* do for rules about how to understand variables

Lexical Binding

High level: Variable Usage

There are two ways a variable can be used in a program:

- As a declaration
- As a "reference" or use of the variable

Scheme/Racket has two kinds of variable declarations

- the **bindings** of a `let`-expression and
- the **parameters** of a `lambda`-expression

Note: **Back to no mutation world!!**
No set! or begin here

Scope of a declaration

The *scope* of a declaration is the portion of the expression or program to which that declaration applies

Lexical binding

- Scope of a variable is determined by textual layout of the program
- C, Java, Scheme/Racket use lexical binding

Dynamic binding

- Scope of a variable is determined by most recent **runtime** declaration
- Bash and classic Lisp use dynamic binding

Scope in Scheme

Scope of variables bound (declared) in a `let` is the body of the `let`

Scope of parameters in a `lambda` is the body of the `lambda`

```
(let ([x 5]
      [y 10])
  (* ((lambda (z) (+ z y)) 7)
     x
     y))
```

We mentioned scope when we discussed how to implement MiniScheme environments

Shadowing bindings

Shadowing: Declaring a new variable with the same name as an existing variable in an enclosing scope

```
(let ([x 5]
      [y 10])
  (* ((lambda (x) (+ x y)) 7)
     x
     y))
```

We say that the inner binding for x *shadows* the outer binding for x

How to determine the appropriate binding?

1. Start at the use of a variable

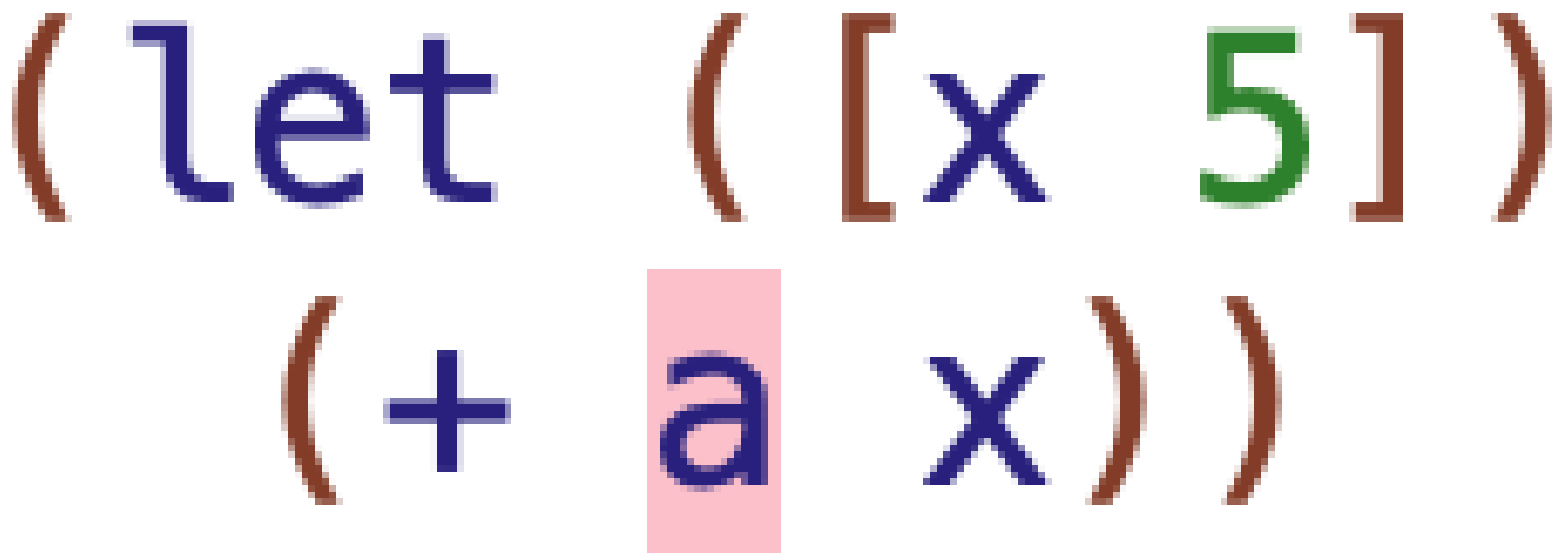
2. Search the enclosing regions starting with the innermost and working outward looking for a binding (declaration) of the variable

3. The first binding you find is the appropriate binding

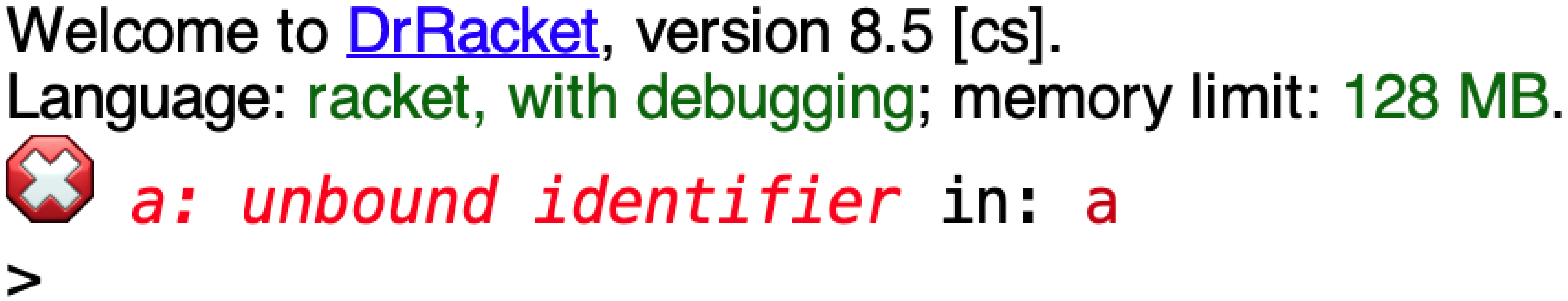
If there are no such bindings, we say the variable is *free*

Free Bindings? Problem!

If there are no such bindings found, we say the variable is *free*. Racket requires all variables to be bound.



```
(let ([x 5])
  (+ a x))
```



```
Welcome to DrRacket, version 8.5 [cs].
Language: racket, with debugging; memory limit: 128 MB.
a: unbound identifier in: a
>
```

```
1. (lambda (x y z)
2.   (if x
3.       (let ([y 10]
4.             [z 20])
5.         (+ x y z))
6.       (- y z)))
```

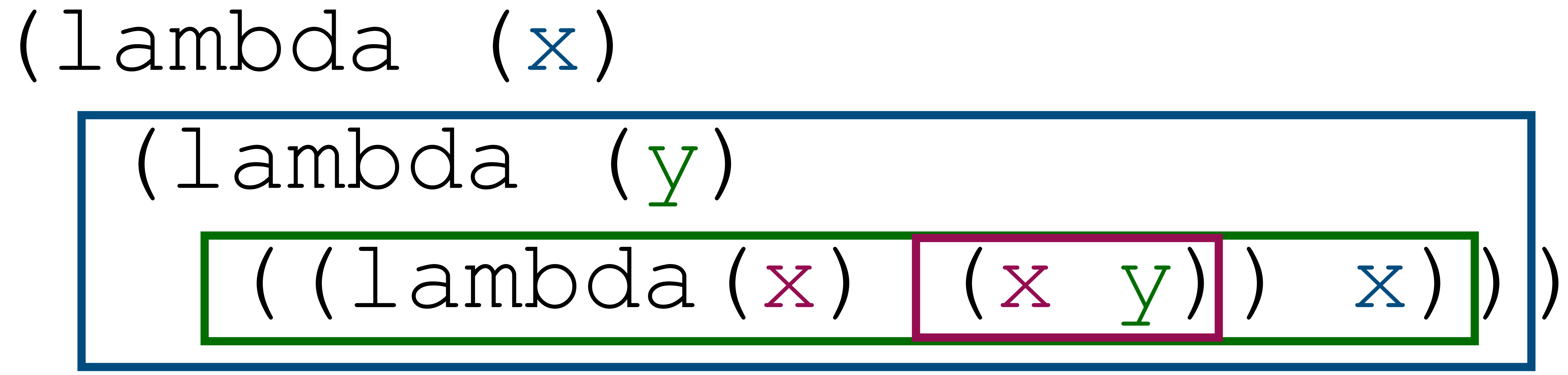
	Line 5 `x`	Line 5 `y`	Line 5 `z`	Line 6 `y`	Line 6 `z`
A	1	1	1	1	1
B	2	3	4	3	4
C	2	3	4	1	1
D	1	3	4	1	1
E	1	3	4	3	4

Which row of the table corresponds to line numbers where the variable indicated in the column was bound?

e.g., E indicates that the variables used in line 5 are bound in lines 1, 3, and 4 and the variables used in line 6 are bound in lines 3 and 4.

Visualizing Scope with Contour Diagrams

Draw the boundaries of the regions in which variable bindings are in effect



```
(lambda (x)
  (lambda (y)
    ((lambda(x) (x  y)) x)))
```

The body of a `let` or a `lambda` expression determines a contour

Each variable refers to the innermost declaration *outside* its contour

Lexical binding vs. Dynamic Binding

Recall: Scope of a declaration

The *scope* of a declaration is the portion of the expression or program to which that declaration applies

Lexical binding

- Scope of a variable is determined by textual layout of the program
- C, Java, Scheme/Racket use lexical binding

Dynamic binding

- Scope of a variable is determined by most recent **runtime** declaration
- Bash and classic Lisp use dynamic binding

What is the value of y in the body of `(f 2)`

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

With lexical (also called static) binding: `y` is 3

- The value of `y` comes from the closest lexical binding of `y`, namely `[y 3]`

With dynamic binding: `y` is 17

- The value of y comes from the most-recent *run-time* binding of `y`, namely `[y 17]`

Lambdas in a *lexically-scoped* language

A lambda expression evaluates to a closure which is a triple containing

- the environment at the time the lambda is evaluated
- the parameters
- the body of the lambda

When we apply the closure to argument expressions

- we evaluate the arguments in the current environment
- extend the **closure's** environment with bindings of parameters to argument values
- evaluate the closure's body in the extended environment

Lambdas in a *dynamically-scoped* language

A lambda expression evaluates to a procedure which is just a pair containing

- the parameters
- the body of the lambda

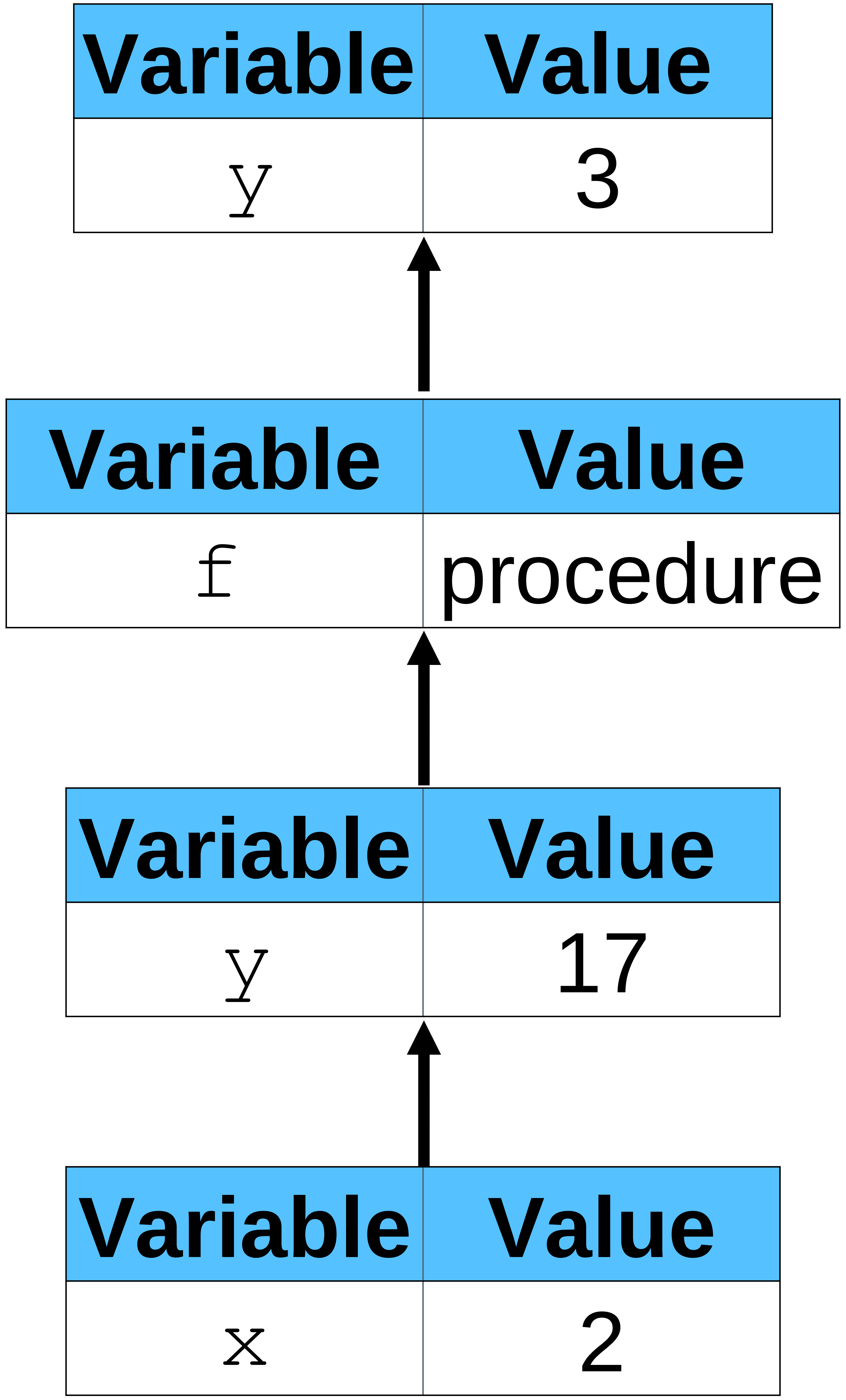
No environment!

When we apply the procedure to argument expressions

- we evaluate the arguments in the current environment
- extend the **current** environment with bindings of parameters to argument values
- evaluate the lambda's body in the extended environment

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

Dynamic binding



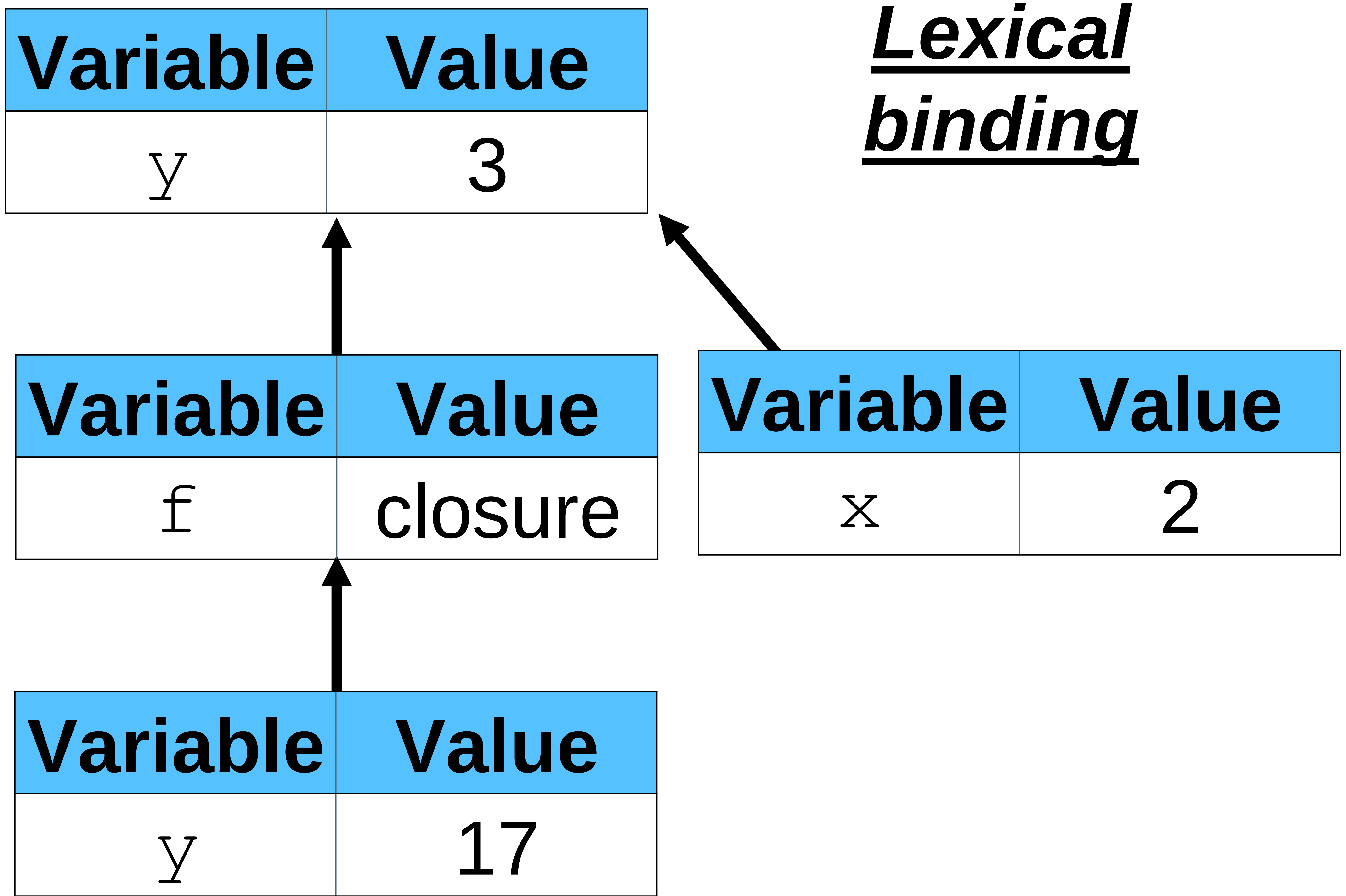
Variable	Value
`y`	3

Variable	Value
`f`	procedure

Variable	Value
`y`	17

Variable	Value
`x`	2

Lexical binding



Variable	Value
`y`	3

Variable	Value
`f`	closure

Variable	Value
`y`	17

Variable	Value
`x`	2

```
(let* ([x 10]
       [f (lambda (x) (+ x x))])
  (f (- x 5)))
```

What is the value of this expression assuming lexical binding? What about dynamic binding?

A. Lexical: 10
   Dynamic: 10

B. Lexical: 10
   Dynamic: 20

C. Lexical: 20
   Dynamic: 10

D. Lexical: 20
   Dynamic: 20

E. None of the above

```
(let* ([x 10]
       [f (lambda (y) (+ x y))])
  (f (- x 5)))
```

What is the value of this expression assuming lexical binding? What about dynamic binding?

A. Lexical: 15
Dynamic: 15

B. Lexical: 15
Dynamic: 10

C. Lexical: 10
Dynamic: 15

D. Lexical: Error
Dynamic: 10

E. None of the above

```
(define f
  (let ([z 100])
    (lambda (x) (+ x z))))

(let ([z 10])
  (f 2))
```

What is the value of this let expression assuming lexical binding? What about dynamic binding?

A. Lexical: 12
   Dynamic: 12

B. Lexical: 12
   Dynamic: 102

C. Lexical: 102
   Dynamic: 12

D. Lexical: 102
   Dynamic: 102

E. None of the above

Dynamic MiniScheme

eval-exp `((lambda (x y) (+ x y)) 3 5)`

`apply-proc` will evaluate the closure

```
(closure '(x y)
         (app-exp (var-exp '+)
                  (list (var-exp 'x) (var-exp 'y)))
         e)
```

by calling `eval-exp` on the body in the environment

`e[x ↦ 3, y ↦ 5]`

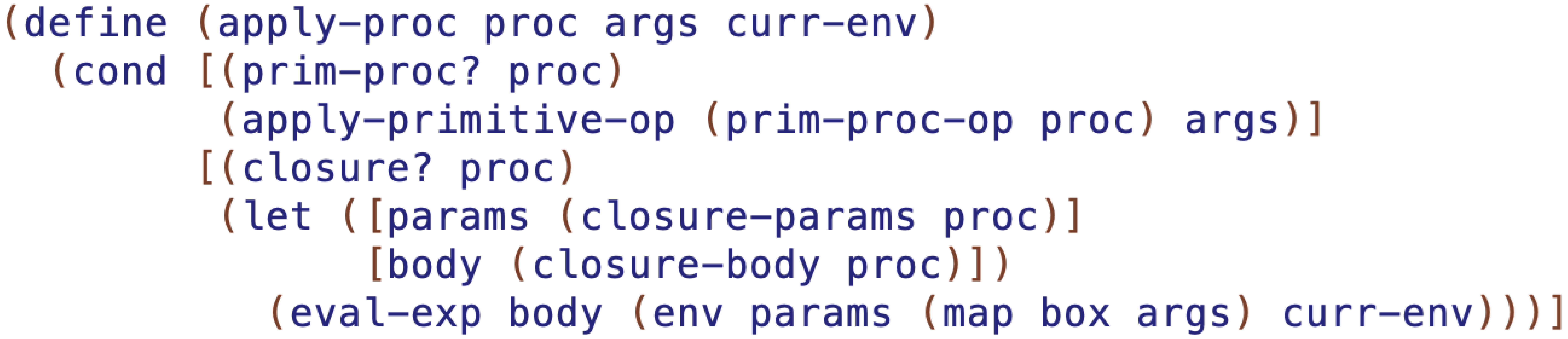
Since the body is an `app-exp`, it'll evaluate `(var-exp '+)` to get `(prim-proc '+)` and the arguments to get `'(3 5)`

How to change to dynamic scope?

1. apply-proc in normal MiniScheme does *not include* the current environment
   - **Change:** make the signature `(apply-proc proc args curr-env)`

2. apply-proc in normal MiniScheme *extends* the **closure's** environment
   - **Change:** ignore the closure's environment! Just extend and evaluate in `curr-env` instead.

How to change to dynamic scope?



```
(define (apply-proc proc args curr-env)
  (cond [(prim-proc? proc)
         (apply-primitive-op (prim-proc-op proc) args)]
        [(closure? proc)
         (let ([params (closure-params proc)]
               [body (closure-body proc)])
           (eval-exp body (env params (map box args) curr-env)))]
```

A Greater Context

Why use dynamic binding?

It's easy to implement! dynamic binding was understood several years before static binding

Without additional context, it makes `(lambda (x) (+ x y))` use whatever the latest, runtime version of `y` is

Why do we now use lexical binding?

Most languages 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

- e.g., in Racket, we can ensure a variable is declared before it is used *before* we run the program
- It makes more sense to most people

Python example

Reminder: this is currying!

```
def fun(x):
    return lambda y: x + y

def main():
    f = fun(10)
    print(f(7))        # Prints 17
    x = 20
    print(f(7))        # Prints 17

main()
```

Bash example

```
#!/bin/bash

x=0

setx() {
  x=$1
}

printx() {
  echo "${x}"
}

main() {
  printx  # prints 0
  setx 10
  printx  # prints 10
  local x=25
  printx  # prints 25!
  setx 100
  printx  # prints 100!
}

main
printx    # prints 10
```