

Programming Abstractions

Lecture 22: Variable Bindings

Stephen Checkoway

Announcements

HW 6—MiniScheme A–E—due Friday

Office Hours: Friday 13:30–14:30

Lexical Binding

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 has two kinds of variable declarations

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

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

Java example

What is the scope of y in this Java program?

Could we print y instead of x in the last line?

```
public static void main(String[] args) {  
    int x = 1;  
    while (x < 10) {  
        int y = x;  
        System.out.println(y);  
        x += 1;  
    }  
    System.out.println(x);  
}
```

Scope in Scheme

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

Scope of parameters in a λ is the body of the λ

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

Shadowing bindings

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

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

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

Determining the appropriate binding

Start at the use of a variable

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

The first binding you find is the appropriate binding

(If there are no such bindings, we say the variable is *free*; Racket requires all variables be bound)

```

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

```

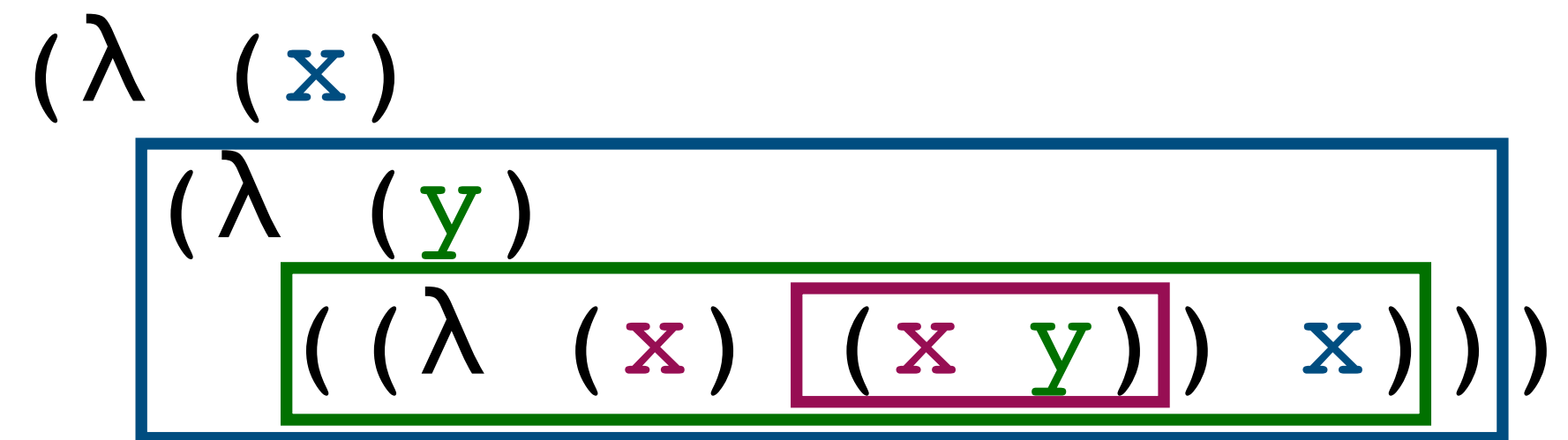
Which row of the table corresponds to line numbers where the variable indicated in the column was bound?

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.

	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

Contour diagrams

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



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

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

```
(λ (x y z)
  (if x
      (let ([y 10]
             [z 20])
        (+ x y z))
      (- y z)))
```

Which is the correct contour for the variable x?

- A. Blue dotted rectangle
- B. Green dashed rectangle
- C. Purple solid rectangle
- D. Orange fuzzy rectangle?

```

(λ (x y z)
  (if x
    (let ([y 10]
          [z 20])
      (+ x y z))
    (- y z)))

```

Which is the correct contour for the inner variable *y*?

- A. Blue dotted rectangle
- B. Green dashed rectangle
- C. Purple solid rectangle
- D. Orange fuzzy rectangle?

Lexical depth

The lexical depth of a variable reference is 1 less than the number of contours crossed between the reference and the declaration it refers to

The diagram shows three nested lambda expressions. The outermost expression is $(\lambda (x) \dots)$ with a blue box around the parameter x . Inside it is $(\lambda (y) \dots)$ with a green box around the parameter y . Inside that is $((\lambda (x) (x\ y))\ x)$. In this innermost expression, the parameter x is enclosed in a pink box, and the variable y is enclosed in a green box. The variable x in the innermost expression refers to the x in the outermost expression, crossing the green box of the middle lambda. The variable y in the innermost expression refers to the y in the middle lambda, crossing the blue box of the outer lambda.

$(\lambda (x)$
 $(\lambda (y)$
 $((\lambda (x) (x\ y))\ x))$

In $(x\ y)$

- x has lexical depth 0
- y has lexical depth 1

The other x has lexical depth 1

What is the lexical depth of `m` in the expression `(* m x)` in this procedure?

```
(define fun
  (λ (m lst)
    (foldl (λ (x acc) (+ (* m x) acc))
            0
            lst)))
```

A. 0

B. 1

C. 2

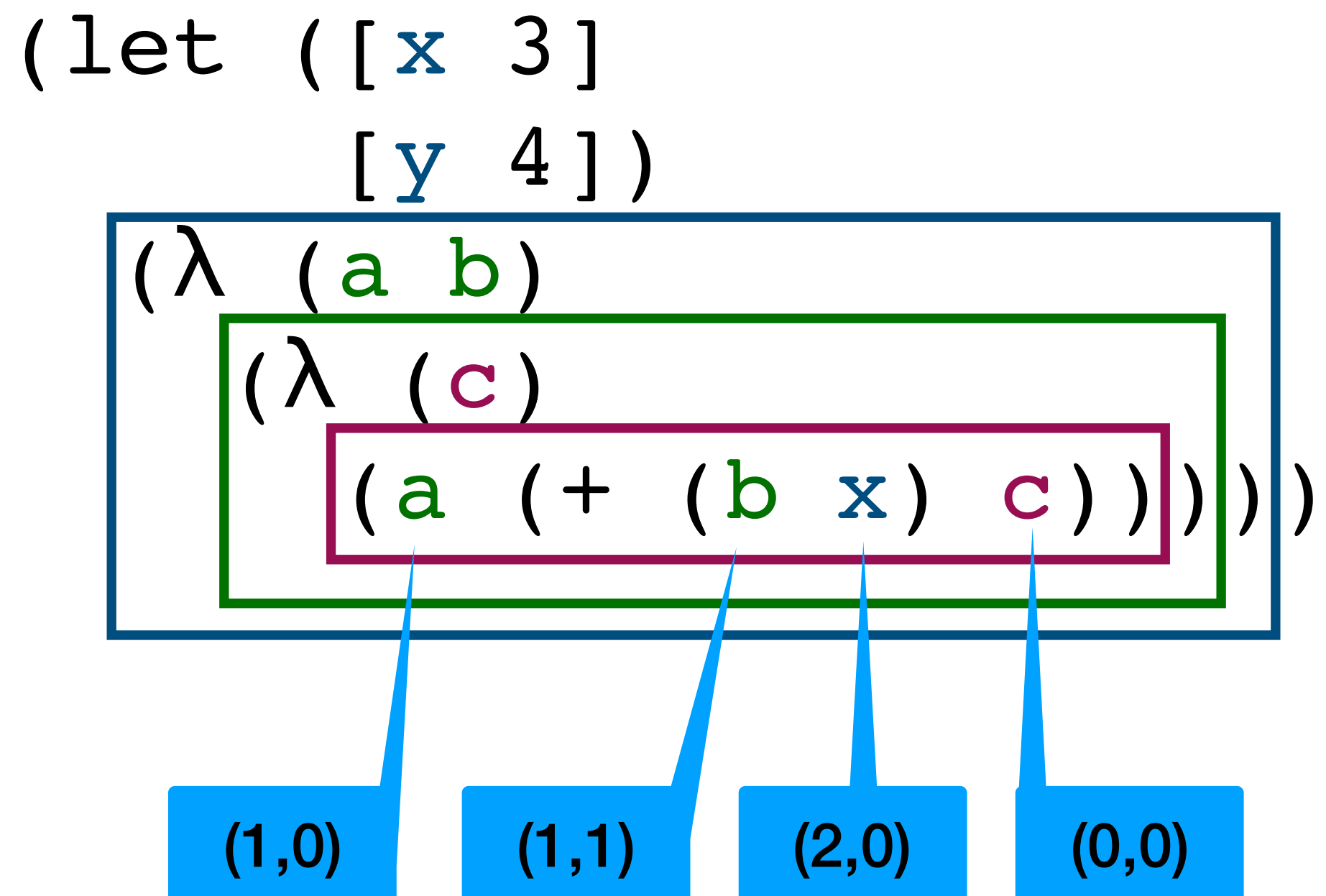
D. 3

E. 4

Lexical addresses

(depth, position)

We can use the lexical depth of a variable along with the 0-based position of the variable in its declaration to come up with a *lexical address* of the variable



Lexical addresses are essentially pointers to where the variable can be found in the environment; can eliminate names

Dynamic binding vs. lexical binding

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 (λ (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

Lexical binding example

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

Lexical binding example

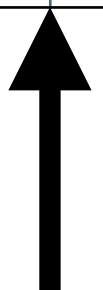
```
(let ([y 3])  
  (let ([f (λ (x) (+ x y))])  
    (let ([y 17])  
      (f 2))))
```

Variable	Value
y	3

Lexical binding example

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

Variable	Value
y	3



Variable	Value
f	closure

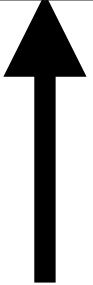
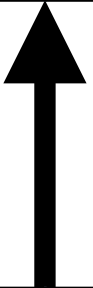
Lexical binding example

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

Variable	Value
y	3

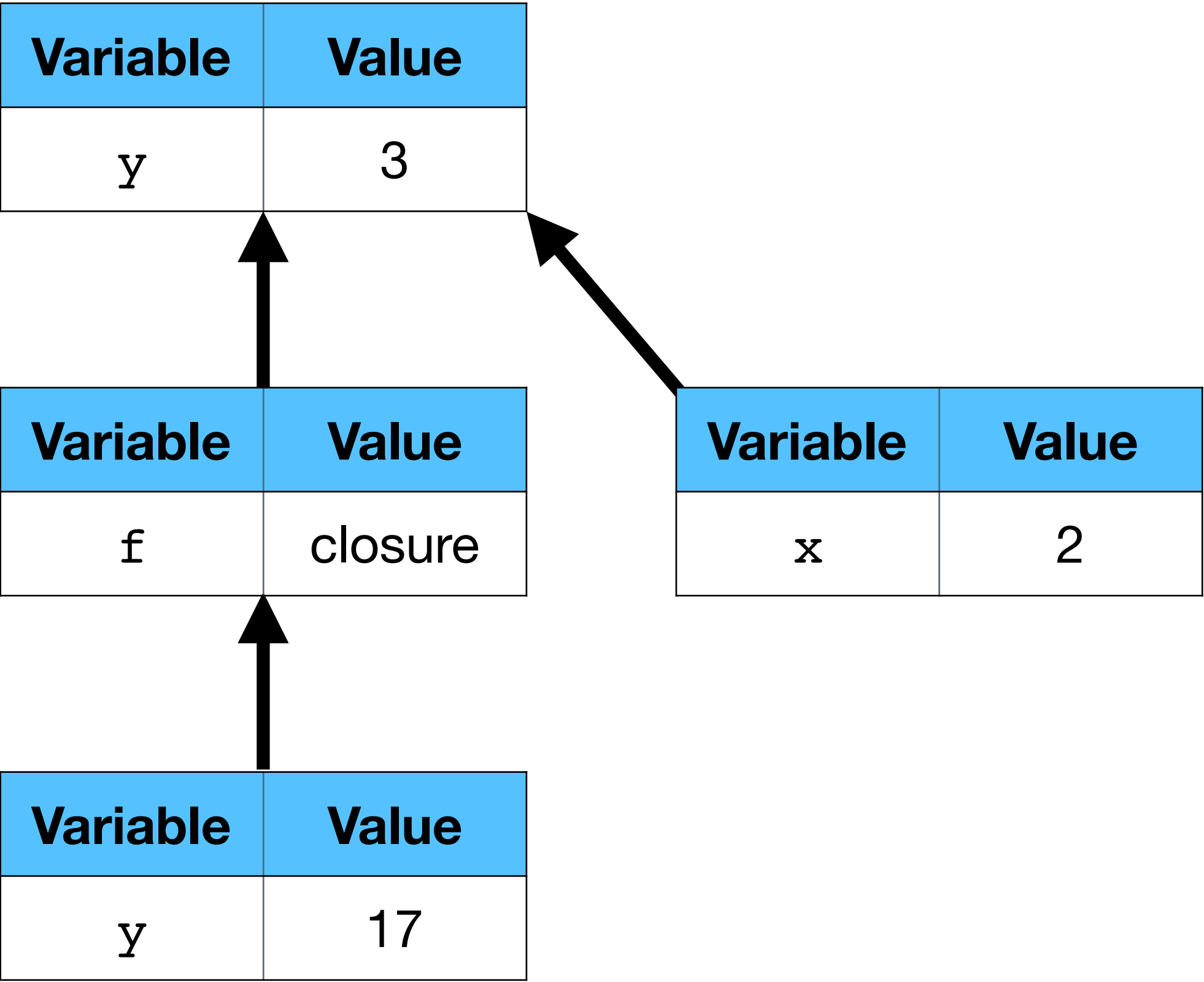
Variable	Value
f	closure

Variable	Value
y	17



Lexical binding example

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



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

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

Dynamic binding example

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

Dynamic binding example

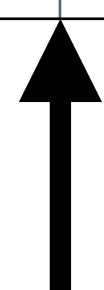
```
(let ([y 3])  
  (let ([f (λ (x) (+ x y))])  
    (let ([y 17])  
      (f 2))))
```

Variable	Value
y	3

Dynamic binding example

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

Variable	Value
y	3



Variable	Value
f	procedure

Dynamic binding example

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

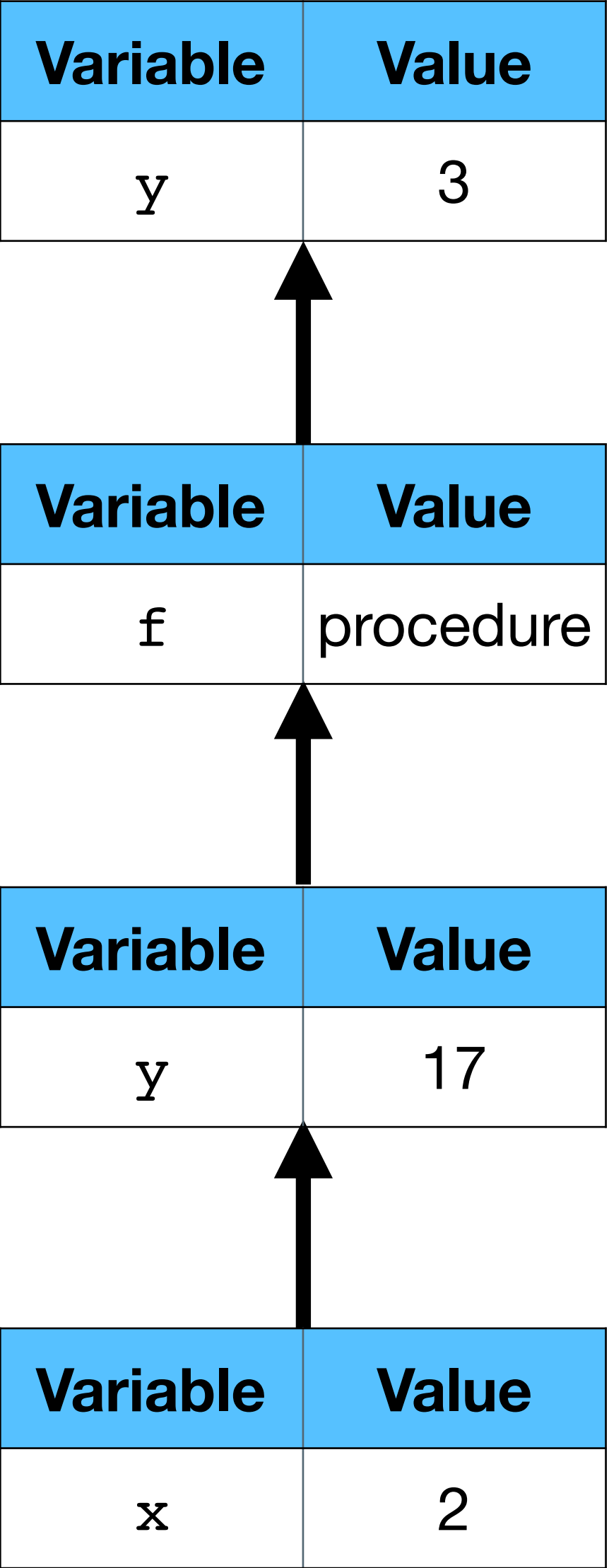
Variable	Value
y	3

Variable	Value
f	procedure
y	17

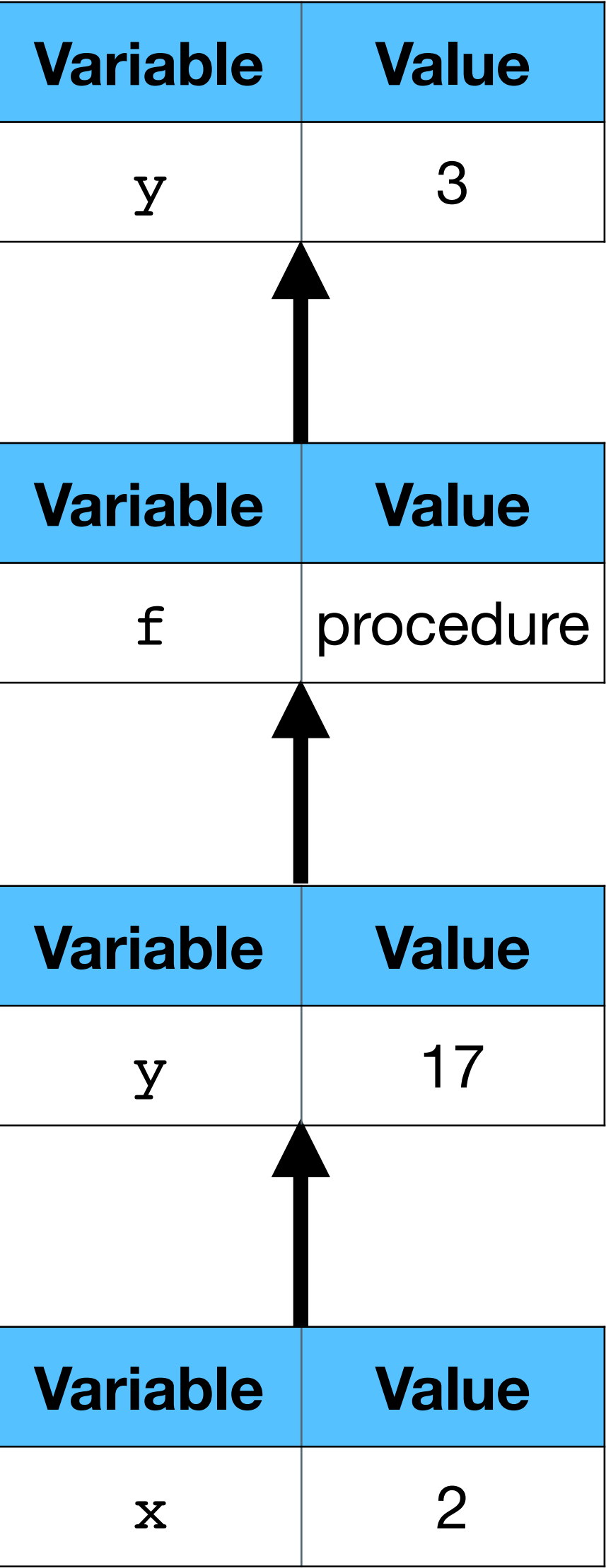
Variable	Value
y	17

Dynamic binding example

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



Dynamic binding



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

Lexical binding

