

# **Programming Abstractions**

**Week 9-1: Dynamic Bindings and Parameter Passing**

**Stephen Checkoway**

```
(define f
  (let ([x 1]
        [y 2])
    (λ (z) (if z x y)))))
```

What is the value of `(f 10)`?

- A. This is a run-time error because 10 isn't a boolean
- B. This is some other sort of error
- C. 1
- D. 2
- E. None of the above

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

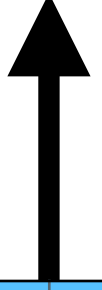
Variable	Value
y	3



# 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))))
```

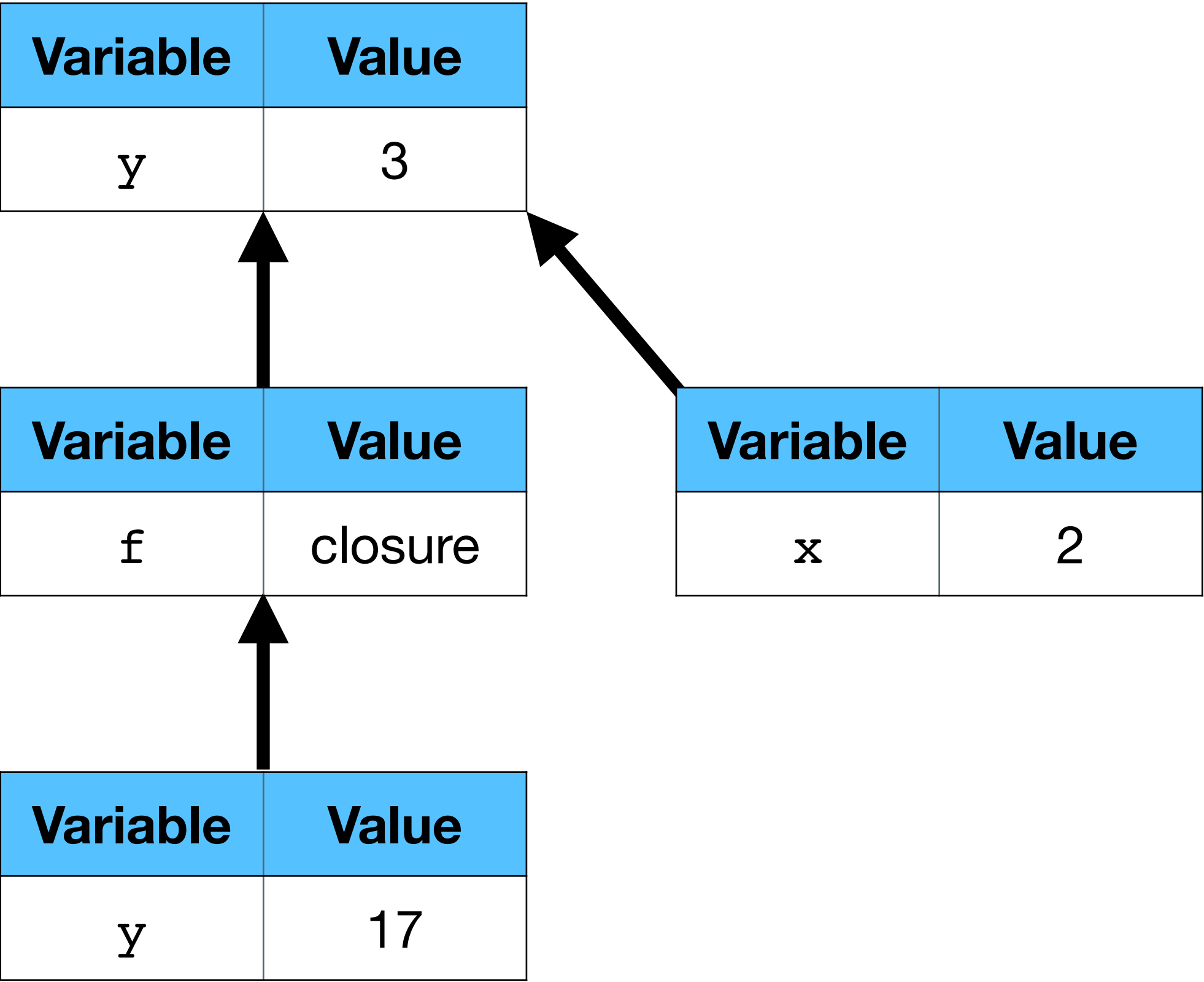
Variable	Value
y	3

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 new environment

# Dynamic binding example

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

# Dynamic binding example

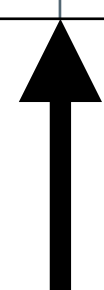
```
(let ([y 3])  
  (let ([f (λ (x) (+ x y))])  
    (let ([y 17])  
      (f 2))))
```

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))))
```

The diagram illustrates the dynamic binding process using three frames. Each frame is a table with two columns: 'Variable' and 'Value'. The frames are arranged vertically, with arrows pointing upwards from the bottom frame to the middle frame, and from the middle frame to the top frame, indicating the lookup path.

Variable	Value
y	3

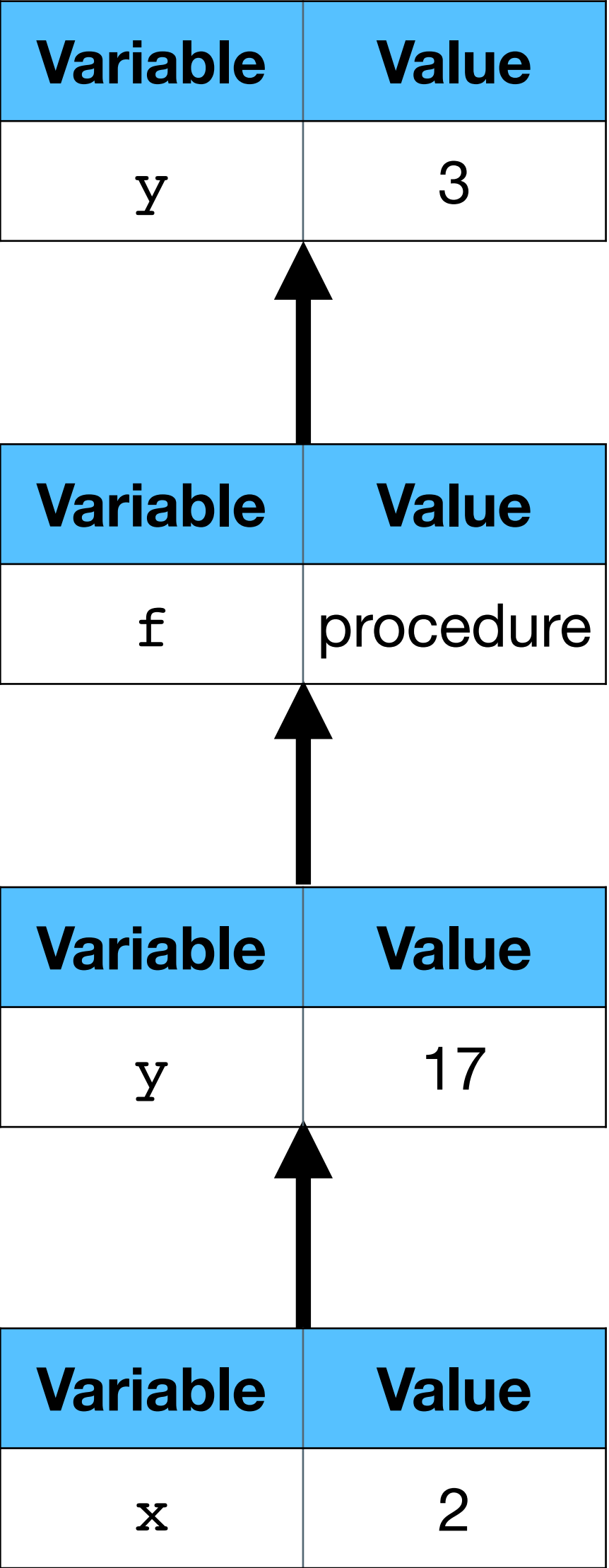
Variable	Value
f	procedure

Variable	Value
y	17



# Dynamic binding example

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



```
(let* ([x 10]
       [f (λ (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 (λ (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])
    (λ (x) (+ x z))))
```

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

- A. Lexical: 12  
Dynamic: 12
- B. Lexical: 12  
Dynamic: 102
- C. Lexical: 102  
Dynamic: 12

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

- D. Lexical: 102  
Dynamic: 102
- E. None of the above

# Why was dynamic binding ever used?

It's easy to implement

- Dynamic binding was understood several years before static binding

It made sense to some people that  $(\lambda (x) (+ x y))$  should use whatever the latest 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

```
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

```
1  #!/bin/bash
2
3  x=0
4
5  setx() {
6      x=$1
7  }
8
9  printx() {
10     echo "${x}"
11 }
12
```

```
13 main() {
14     printx # prints 0
15     setx 10
16     printx # prints 10
17     local x=25
18     printx # prints 25!
19     setx 100
20     printx # prints 100!
21 }
22
23 main
24 printx # prints 10
```



# Parameter-passing mechanisms

# Three mechanisms

## Pass by value

- Arguments are evaluated in the caller's environment
- Argument values are bound to parameters

## Pass by reference

- Arguments must be variables
- Addresses of arguments are bound to the parameters

## Pass by name

- Arguments are not evaluated
- The text of the arguments is passed to the function and replace the parameters in the function's body

# Aside: Mutation and sequencing

To see the difference between pass by value and pass by reference, we need to be able to mutate (modify) variables

In Scheme, `(set! var value)`

```
(let ([v 10])
```

```
  (begin (displayln v)      ; prints 10
```

```
         (set! v 20)
```

```
         (displayln v))) ; prints 20
```

```
(begin exp1 ... expn)
```

- Evaluates each expression and returns the value of the final one
- The other  $n-1$  expressions are only useful for their *side effects* like printing or modifying variables
- `begin` isn't actually needed here, `let` allows multi-expression bodies

# Pass by value vs. by reference

```
(let ([v 0]
      [f (λ (x) (set! x 34))])
  (f v)
  v)
```

## Pass by value

- ▶ When evaluating `(f v)`, `x` is initially bound to 0
- ▶ The `(set! x 34)` sets the value of `x` to 34; `v` remains bound to 0
- ▶ The final `v` evaluates 0 and thus the whole expression evaluates to 0

## Pass by reference

- ▶ When evaluating `(f v)`, `x` and `v` refer to the same variable with value 0
- ▶ The `(set! x 34)` sets the value of that variable to 34
- ▶ The final `v` (and the whole expression) evaluates to 34

```
(define (f x y)
  (let ([z x])
    (set! x (* y 2))
    (set! y (* z 3))))
(let ([a 1] [b 2])
  (f a b)
  (list a b))
```

What is the value of the let expression assuming pass by value? What about pass by reference?

A. Value: ' ( 1 2 )

Reference: ' ( 1 2 )

B. Value: ' ( 1 2 )

Reference: ' ( 4 3 )

C. Value: ' ( 4 3 )

Reference ' ( 1 2 )

D. Value: ' ( 1 2 )

Reference: ' ( 4 12 )

# Pass by reference in Scheme (sort of)

We create a box which holds a value

The value of the box itself is the address of the variable and can be passed to functions

The value inside the box can be mutated

```
(let ([v (box 0)]  
      [f (λ (x) (set-box! x 34))])  
  (f v)  
  (unbox v)) ; Returns 34
```

# Pass by value vs name

## Pass by value

```
(let* ([v 0]
       [f (λ (x) ; Don't need begin in λ body
             (set! v (+ v 1))
             x)])
  (f (+ v 5)))
```

## Pass by value

- `f` is called with value 5 so `x` is bound to 5
- `v` is set to 1
- `x` is returned

# Pass by value vs name

## Pass by name

```
(let* ([v 0]
      [f (λ (x) ; Don't need begin in λ body
            (set! v (+ v 1))
            x)])
  (f (+ v 5)))
```

## Pass by name

- The text of `f`'s body becomes the two expressions (by replacing `x` with the text of the argument)  
`(set! v (+ v 1))`  
`(+ v 5)`
- `v` is set to 1 and then 6 is returned



# Pass by name in Scheme: macros

```
(define-syntax-rule (name param1 ... paramn) body)
```

We can create macros where the arguments are substituted textually for the parameters (we'll probably discuss this more later in the semester)

```
(let ([v 0])  
  (define-syntax-rule (f x)  
    (begin  
      (set! v (+ v 1))  
      x))  
  (f (+ v 5)))
```

This isn't quite the same as pass by name because Scheme macros don't allow free variables (here, `v` always refers to the `v` in the `let` expression)

# Pass by x

## Pass by value

- Easiest to understand and most common
- Used by Scheme, Java, C, Python, Bash, and most other languages

## Pass by reference

- Allows modifying passed in variables which can be useful in languages that don't support returning multiple values
- Supported by C++, C#, Rust, and others

## Pass by name

- Least common mechanism and by far the most difficult to reason about
- Used by macro languages like TeX, m4, and C's preprocessor
- Macro constructs in languages like Scheme and Rust

# Pass by name in TeX

TeX is a macro language for writing documents

```
1 \def\work#1#2{%  
2   All work and no play makes #1 a dull #2.\par  
3 }  
4 \def\sad#1#2dull{%  
5   #1 a sad%  
6 }  
7 \work{Jack}{boy}  
8 \work{\sad{Steve}}{professor}  
9 \bye
```

All work and no play makes Jack a dull boy.

All work and no play makes Steve a sad professor.

# Pass by name in the C preprocessor

```
1 #include <stdio.h>
2
3 #define swap(x, y) \
4     int tmp = x; \
5     x = y; \
6     y = tmp
7
8 int main() {
9     int arr[5] = {4, 4, 4, 4, 4};
10    int idx = 2;
11    swap(idx, arr[idx]);
12    printf("%d\n", idx);
13    printf("{%d, %d, %d, %d, %d}\n",
14           arr[0], arr[1], arr[2], arr[3], arr[4]);
15    return 0;
16 }
```

swap sets `idx` to 4 and then `arr[4]` to 2, `arr[2]` is unchanged!

# Rust

```
1 fn by_value(mut x: u32) {  
2     x += 1;  
3 }  
4  
5 fn by_ref(x: &mut u32) {  
6     *x += 1;  
7 }  
8
```

Prints

0

1

7

```
9 fn main() {  
10     let mut v = 0;  
11  
12     macro_rules! by_name {  
13         ($x:stmt) => {  
14             v += 1;  
15             $x  
16         }  
17     }  
18  
19     by_value(v);  
20     println!("{}", v);  
21     by_ref(&mut v);  
22     println!("{}", v);  
23     by_name!(v += 5);  
24     println!("{}", v);  
25 }
```

# Implementing pass by reference

MiniScheme implements pass-by-value (or will, once you implement lambdas in the next homework)

We can make it pass-by reference by

- storing each value in a box;
- when calling functions, do not unbox the values, but pass the boxes as normal;
- unbox when performing primitive procedures

# Implementing pass by name

We can make MiniScheme pass by name via function re-writing

- Don't evaluate arguments at all
- In `(apply-proc p args)`, rewrite the procedure's body (which is a parse tree) replacing each use of a parameter with the parse tree for the corresponding argument