L8: Environment Model

CS1101S: Programming Methodology

Low Kok Lim

October 6, 2021

Outline

- State (3.1)
- Environment Model (3.2)

Outline

- State (3.1)
- Environment Model (3.2)

Recap: Variable Declaration Statement

```
let name = expression;
```

- Declares a ***variable*** *name* in the current scope and initializes its value to the value of *expression*
- From now on, *name* will evaluate to the value of *expression*
- Note that from Source §3 onwards, **function parameters** are **variables**

Recap: Assignment Statement

```
name = expression;
```

- *name* is a **variable**; not evaluated
- *expression* is evaluated, then its value is ***assigned*** to the variable *name*
- From now on, *name* will evaluate to the value of *expression*

Assignment: Pros

- Assignment allows us to create objects with ***state***
- State allows objects to behave differently over time

Assignment: Cons

- Harder to reason about programs
  - Harder to debug
  - Harder to verify correctness

- **Substitution model of evaluation breaks down!**
  - Not powerful enough to explain state
  - Need a more sophisticated model — ***Environment Model***

Substitution Model Breaks Down

- Consider

```
function make_simplified_withdraw(balance) {
    return amount => {
        balance = balance - amount;
        return balance;
    }
}
```

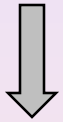
- Use **substitution model** to evaluate

```
(make_simplified_withdraw(25))(20);
```

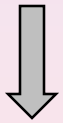
Substitution Model Breaks Down

- Use substitution model to evaluate

```
(make_simplified_withdraw(25))(20);
```



```
(amount => { balance = 25 - amount; return 25; })(20);
```



```
balance = 25 - 20; return 25; // WRONG!
```

- It returns `25`, which is wrong!

Why Substitution Model Breaks Down?

- Substitution model considers a constant/variable as **just a name for a value**
  - Its value will not change
  - Therefore, one can be substituted for the other

- But **assignment** considers a variable as a **"container" holding a value**
  - The contents of the container may be **changed over time**
  - The container is maintained in a structure called an ***environment***

Outline

- State (3.1)
- Environment Model (3.2)

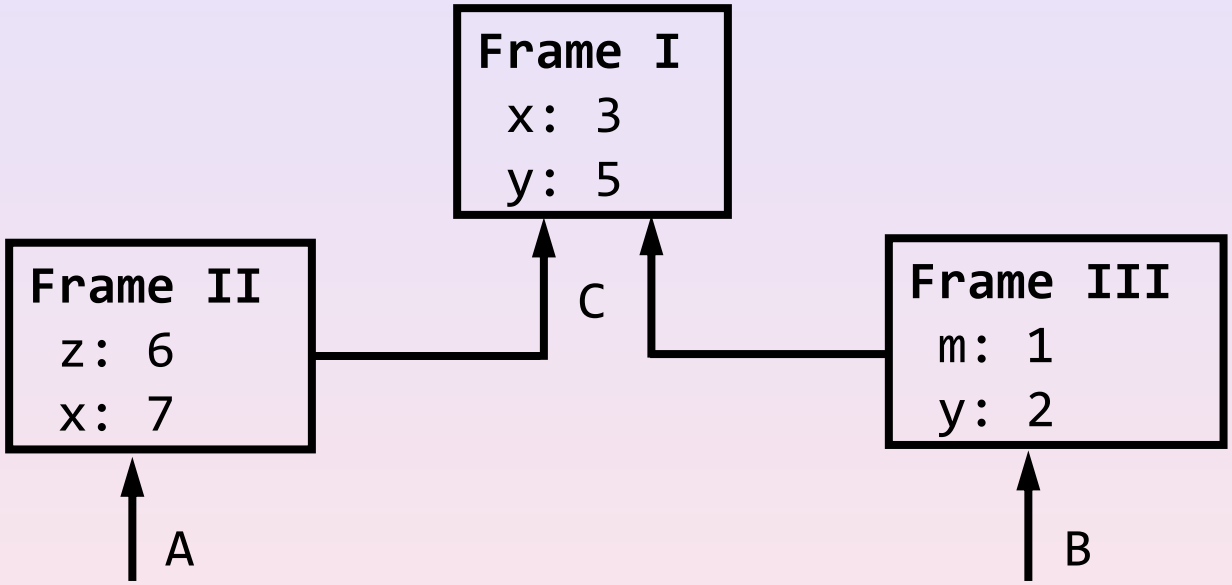
Environment

- An ***environment*** determines the context in which an expression should be evaluated
- Every **expression** is evaluated with respect to the **current** environment

Environments

- An ***environment*** is a ***sequence of frames***
- Each frame contains ***bindings*** of values to names
- A frame points to its ***enclosing environment***, the next one in the sequence
  - Unless the frame is the ***global frame***
- ***Extending*** an environment means ***adding*** a new frame “inside” the old one

Example Environments



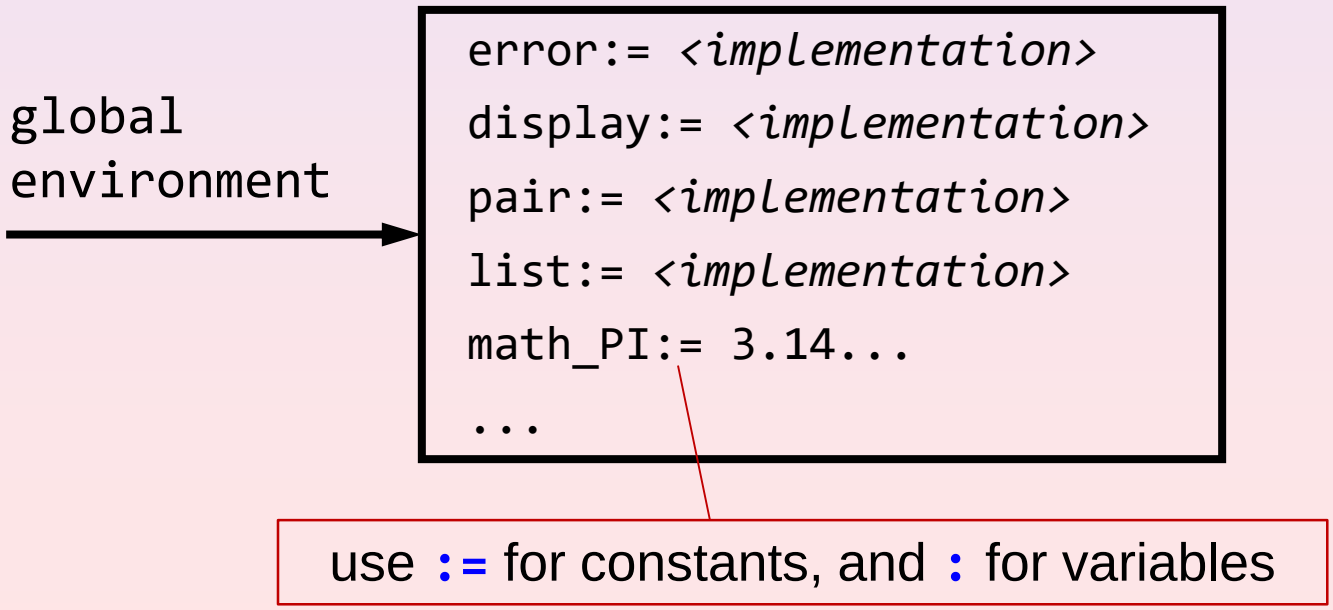
- **A**, **B**, **C** are environments
- Value of x in environment **A** is **7**
- Value of x in environment **B** is **3**
- x in `Frame II` ***shadows*** x in `Frame I`
- Value of `m` in environment **A**? ***unbound***

Accessing an Environment

- To evaluate a **name**, look up its value in the **current frame**
- If **not found in current frame**, then look in **enclosing environment**, and so on
- If **still not found**, then the name is said to be ***unbound*** in the environment

Global Environment

- The ***global environment*** consists of a single **frame** with bindings of **primitive** and **pre-declared** functions and constants
  - Functions' names are bound to their implementations

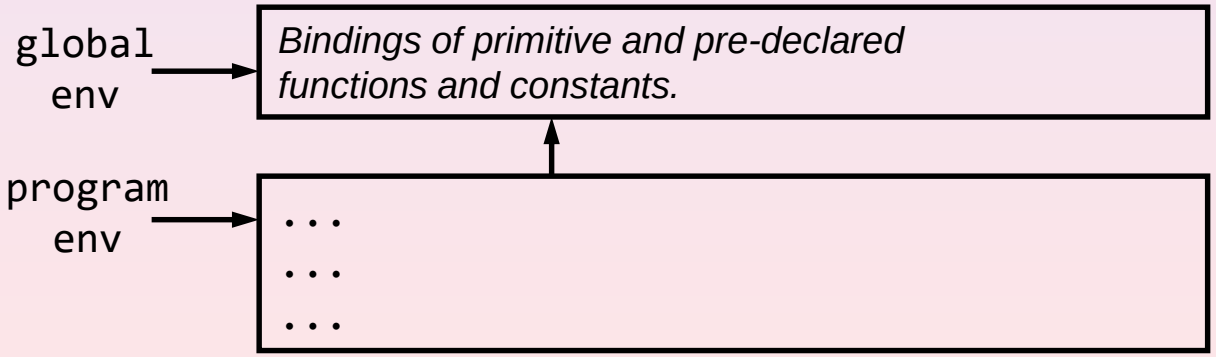


Evaluation of **Blocks**

- Evaluating a **block** `{ statements }` extends the environment by adding a **new frame**
  - New frame contains bindings of the **constants** and **variables** declared in the block
  - Initially, these **constants** and **variables** don't have associated values (**unassigned**)
  - When the **declarations** are evaluated, the **names** are **assigned** their values
- New **frame is not created** if the block has **no constant & variable declaration**
  - **There are no empty frames**

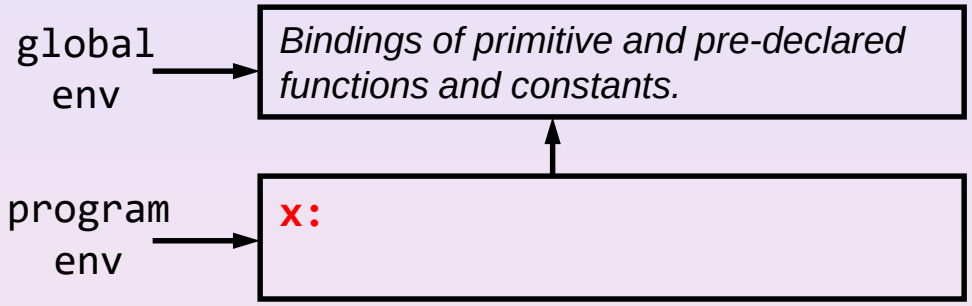
Evaluation of **Blocks**

- **User program** is considered in an implicit **program block**
  - It directly extends the **global environment**
  - It is called the **program environment**



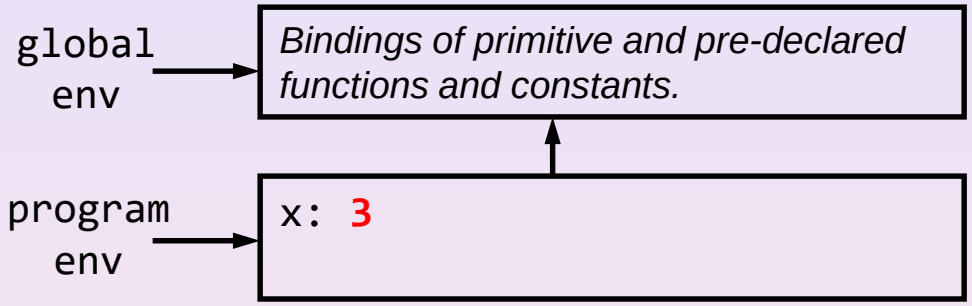
Example for Evaluating Blocks

```
let x = 3;

{
    const x = 5;
    let y = 8;
}
```



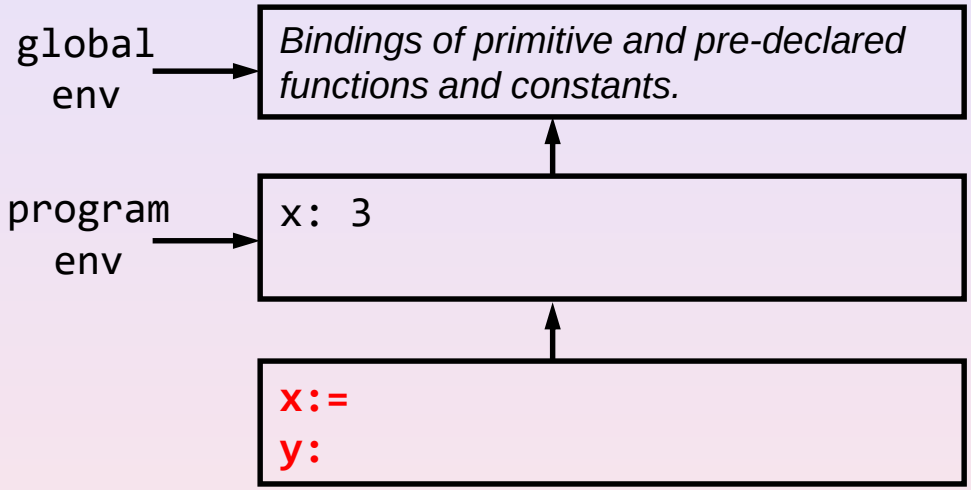
Example for Evaluating Blocks

```
let x = 3;

{
    const x = 5;
    let y = 8;
}
```



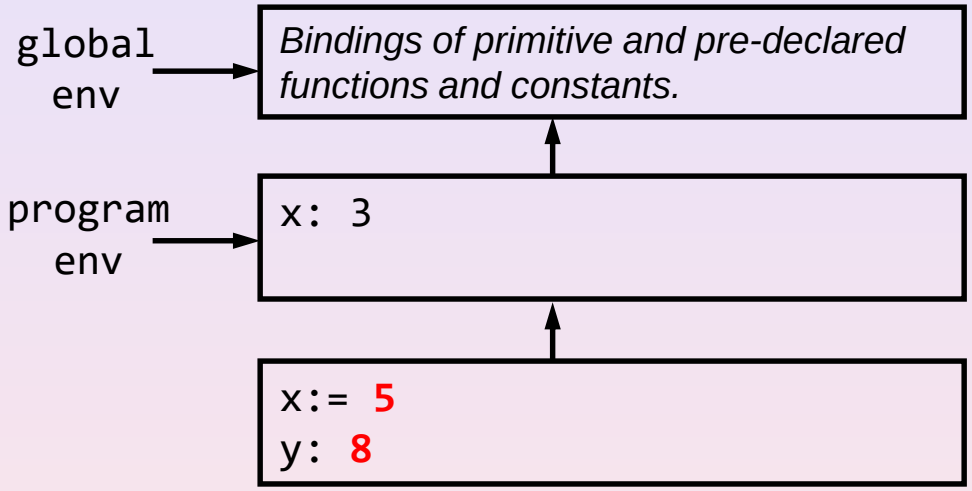
Example for Evaluating Blocks

```
let x = 3;

{
    const x = 5;
    let y = 8;
}
```



Example for Evaluating Blocks

```
let x = 3;

{
    const x = 5;
    let y = 8;
}
```



Evaluation of **Declarations** and **Assignments**

- **Constant** / **variable** declaration statements:

```
const name = value;
let name = value;
```

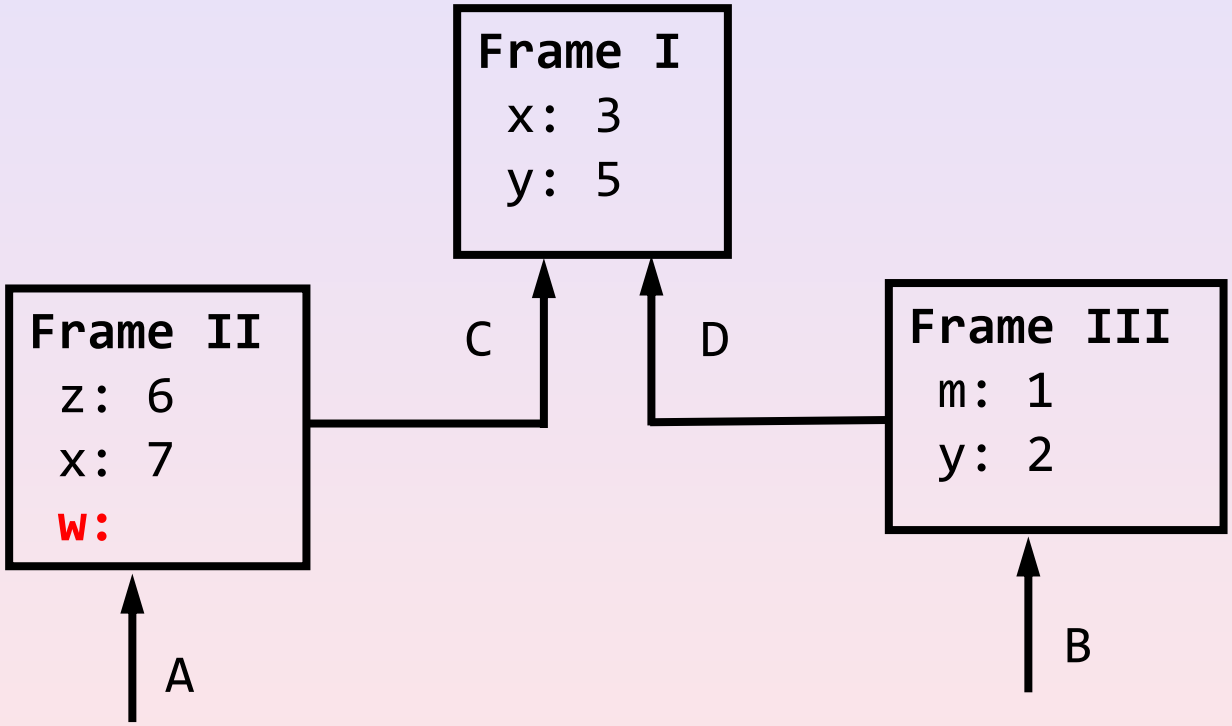
  - Changes binding of *name* to *value* in the **current frame**

- **Assignment** statement:

```
name = new_value;
```

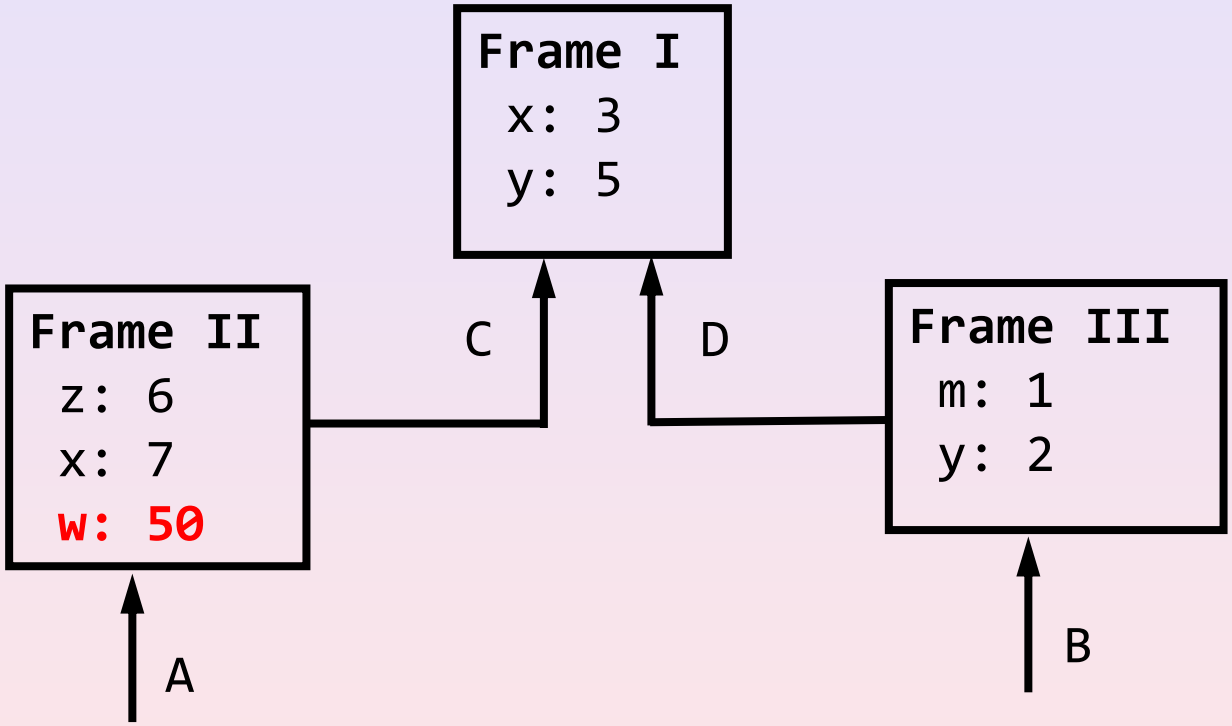
  - Searches for *name* beginning from **current frame**, then changes its binding to *new_value*
    - If *name* is a **constant**, signal an error
    - If *name* is **unbound** or **unassigned**, signal an error

Example Environments



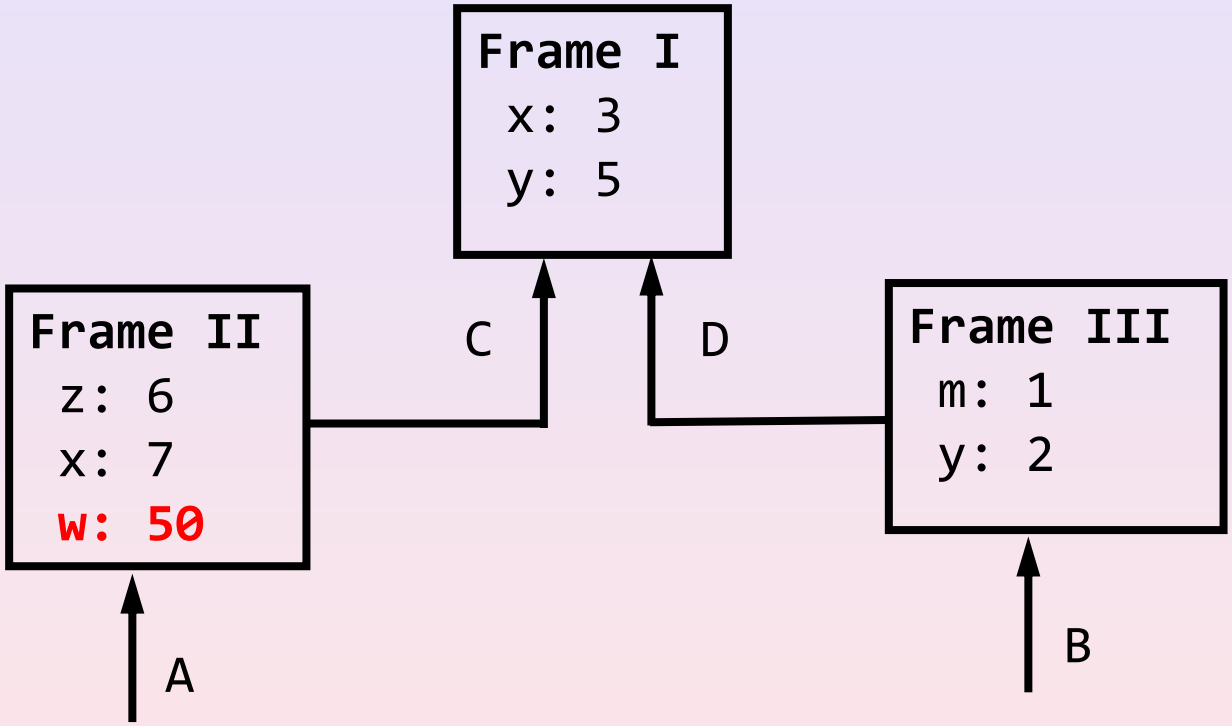
- Evaluate in **A**: `let w = 50;`

Example Environments



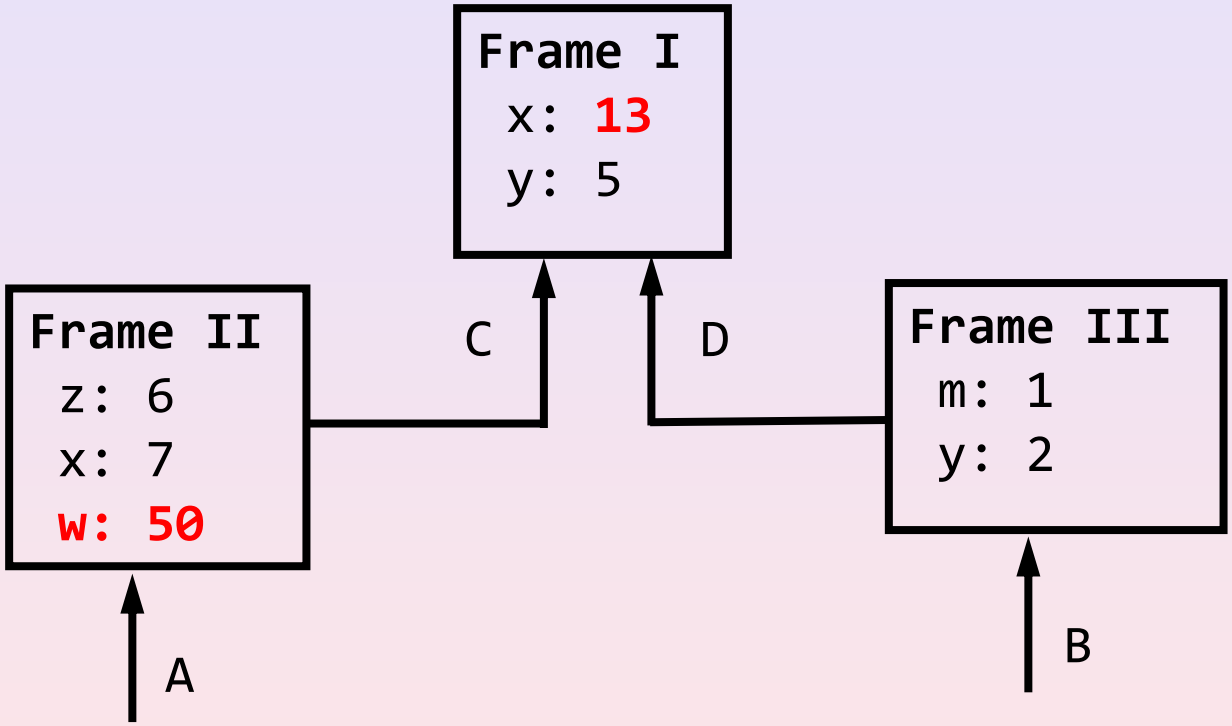
- Evaluate in **A**: `let w = 50;`
  - `w` in `Frame II` is assigned with the value **50**

Example Environments



- Evaluate in **B**: `x = x + 10;`

Example Environments

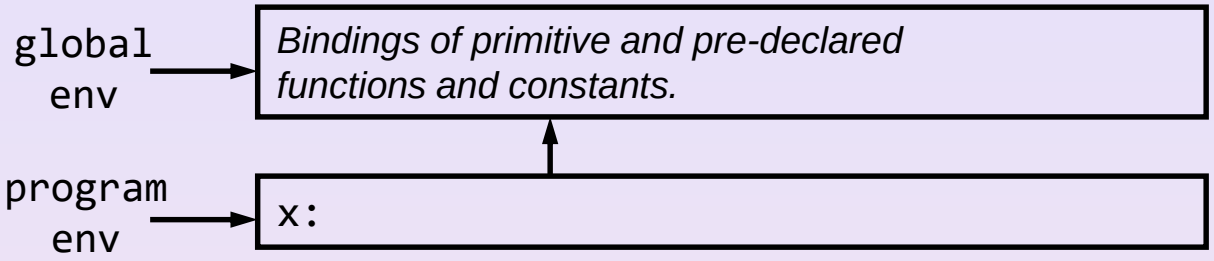


- Evaluate in **B**: `x = x + 10;`
  - The value of `x` in `Frame I` is updated from **3** to **13**

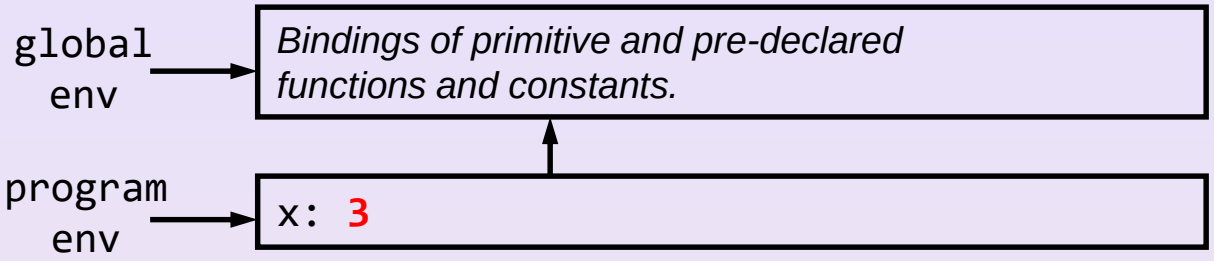
Block Example

```
let x = 3;

if (x <= 1) {
    let y = x;
    x = y + 1;
} else {
    let z = x;
    x = z * z;
}
{
    x = x + 1;
}
{
    let x = 5;
    let y = 8;
    {
        let x = 20 + y;
    }
    x = x + 1;
}
x;
```

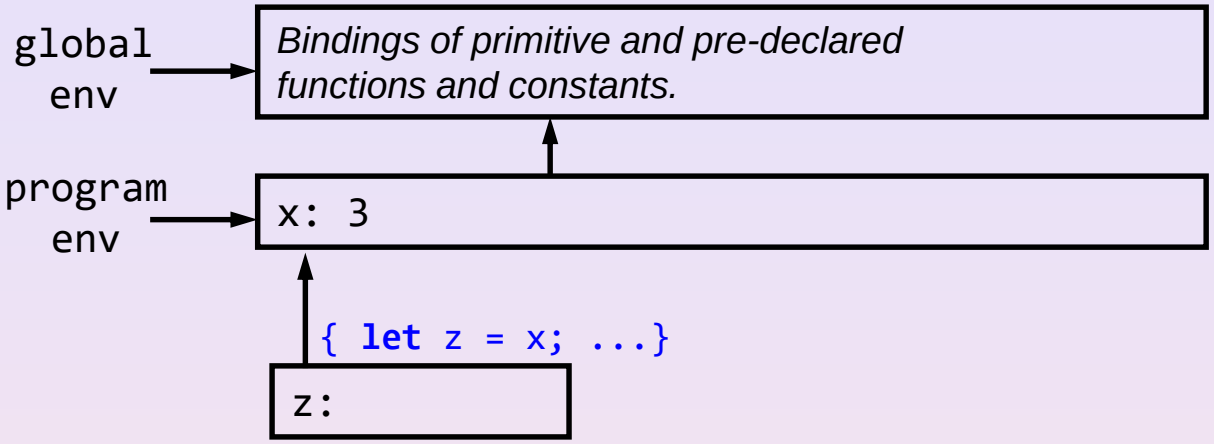
Show in Playground



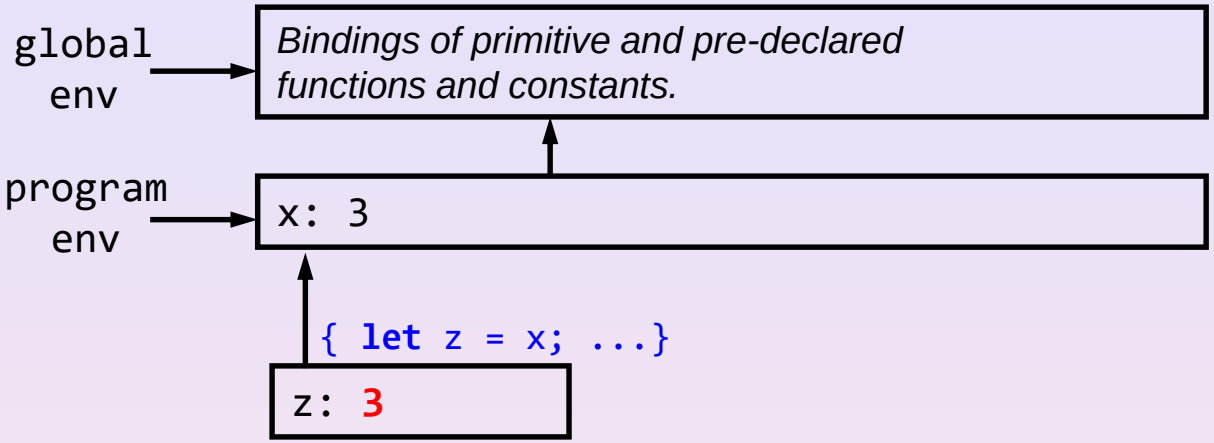

```
let x = 3;

if (x <= 1) {
    let y = x;
    x = y + 1;
} else {
    let z = x;
    x = z * z;
}
{
    x = x + 1;
}
{
    let x = 5;
    let y = 8;
    {
      let x = 20 + y;
    }
    x = x + 1;
}
x;
```



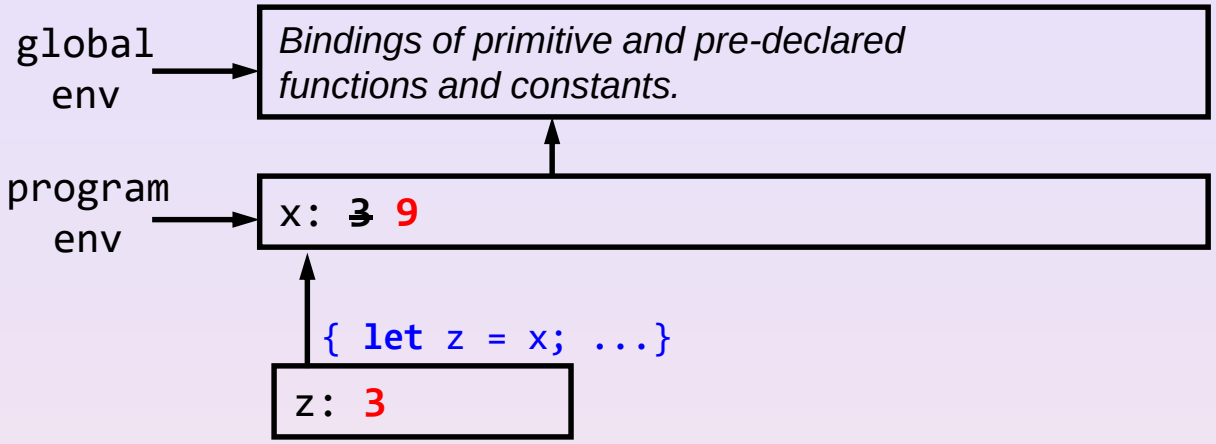

```
let x = 3;

if (x <= 1) {
    let y = x;
    x = y + 1;
} else {
    let z = x;
    x = z * z;
}
{
    x = x + 1;
}
{
    let x = 5;
    let y = 8;
    {
      let x = 20 + y;
    }
    x = x + 1;
}
x;
```



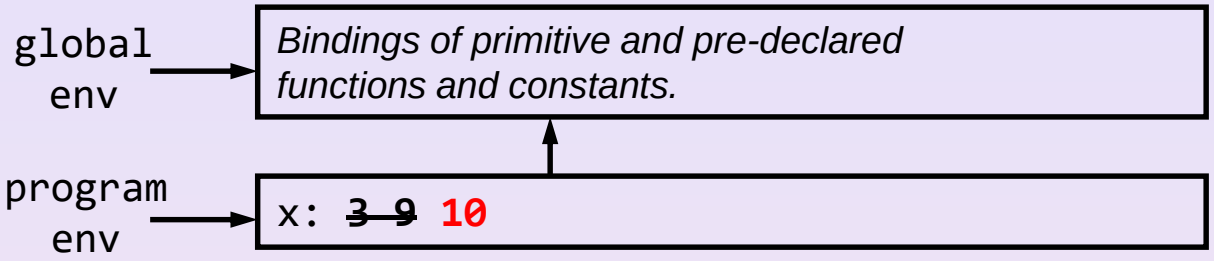

```
let x = 3;

if (x <= 1) {
    let y = x;
    x = y + 1;
} else {
    let z = x;
    x = z * z;
}
{
    x = x + 1;
}
{
    let x = 5;
    let y = 8;
    {
      let x = 20 + y;
    }
    x = x + 1;
}
x;
```



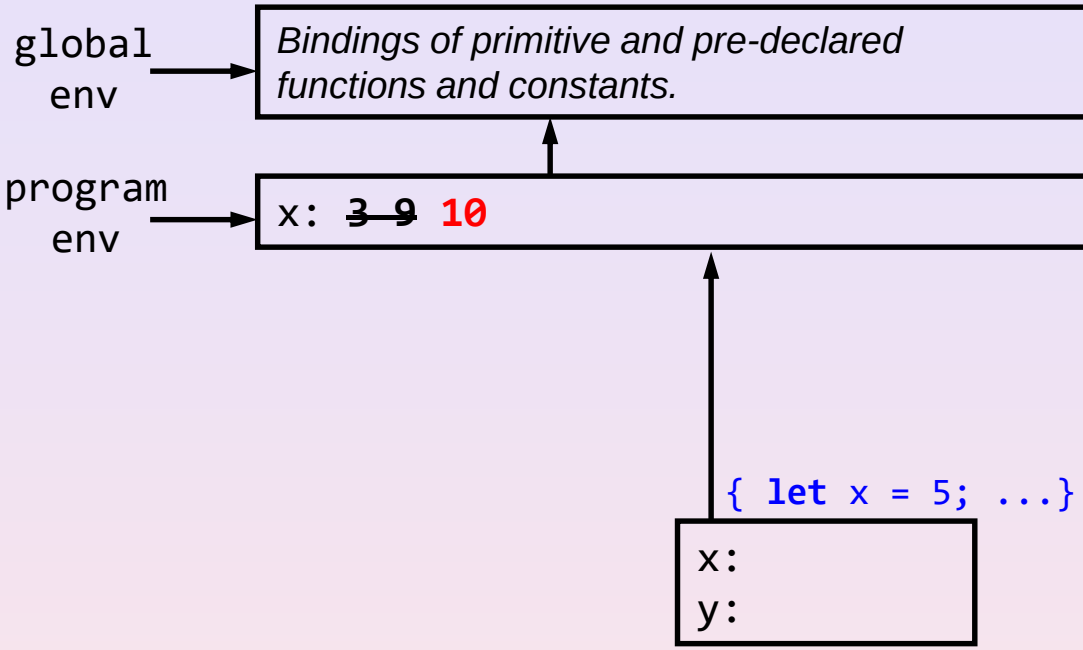

```
let x = 3;

if (x <= 1) {
    let y = x;
    x = y + 1;
} else {
    let z = x;
    x = z * z;
}
{
    x = x + 1;
}
{
    let x = 5;
    let y = 8;
    {
      let x = 20 + y;
    }
    x = x + 1;
}
x;
```



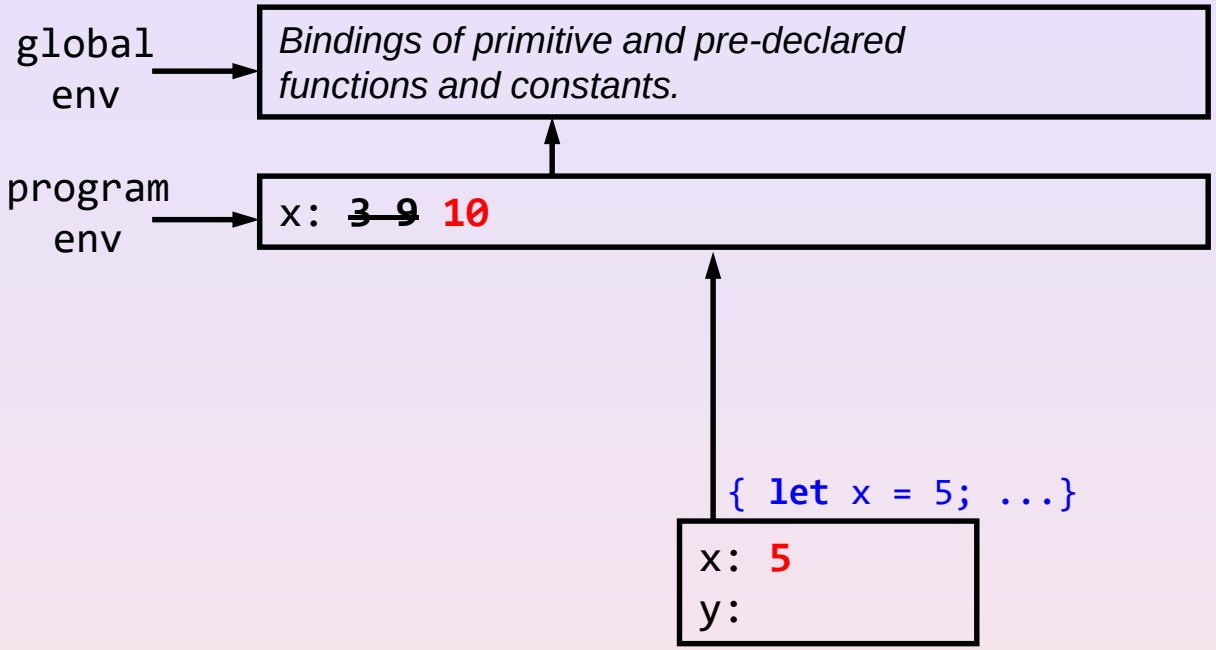

```
let x = 3;

if (x <= 1) {
    let y = x;
    x = y + 1;
} else {
    let z = x;
    x = z * z;
}
{
    x = x + 1;
}
{
    let x = 5;
    let y = 8;
    {
      let x = 20 + y;
    }
    x = x + 1;
}
x;
```



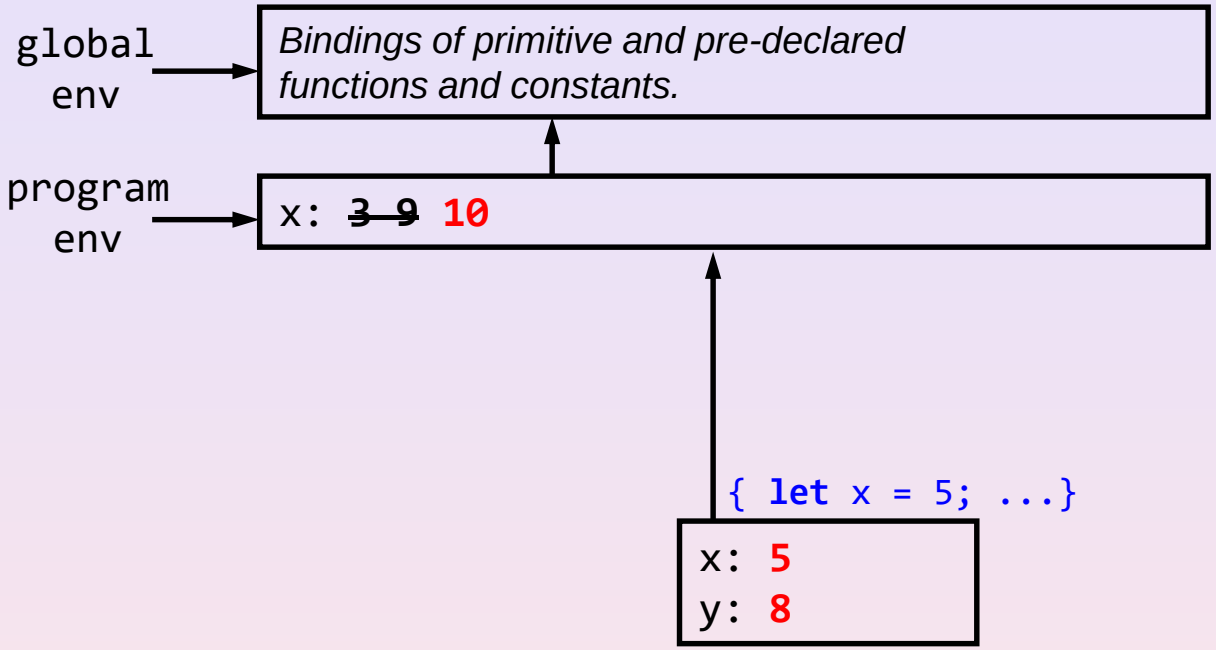

```
let x = 3;

if (x <= 1) {
    let y = x;
    x = y + 1;
} else {
    let z = x;
    x = z * z;
}
{
    x = x + 1;
}
{
    let x = 5;
    let y = 8;
    {
      let x = 20 + y;
    }
    x = x + 1;
}
x;
```



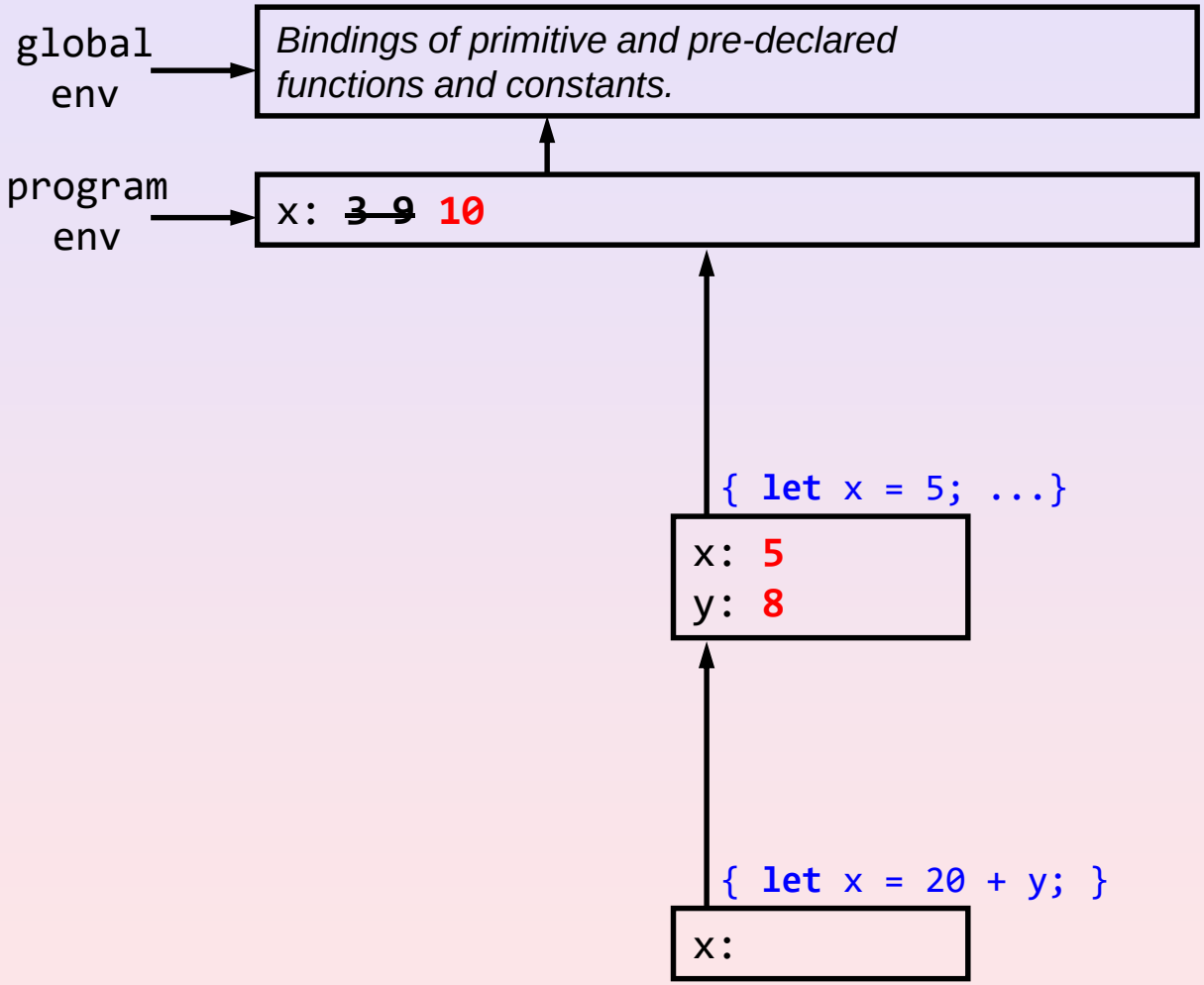

```
let x = 3;

if (x <= 1) {
    let y = x;
    x = y + 1;
} else {
    let z = x;
    x = z * z;
}
{
    x = x + 1;
}
{
    let x = 5;
    let y = 8;
    {
      let x = 20 + y;
    }
    x = x + 1;
}
x;
```



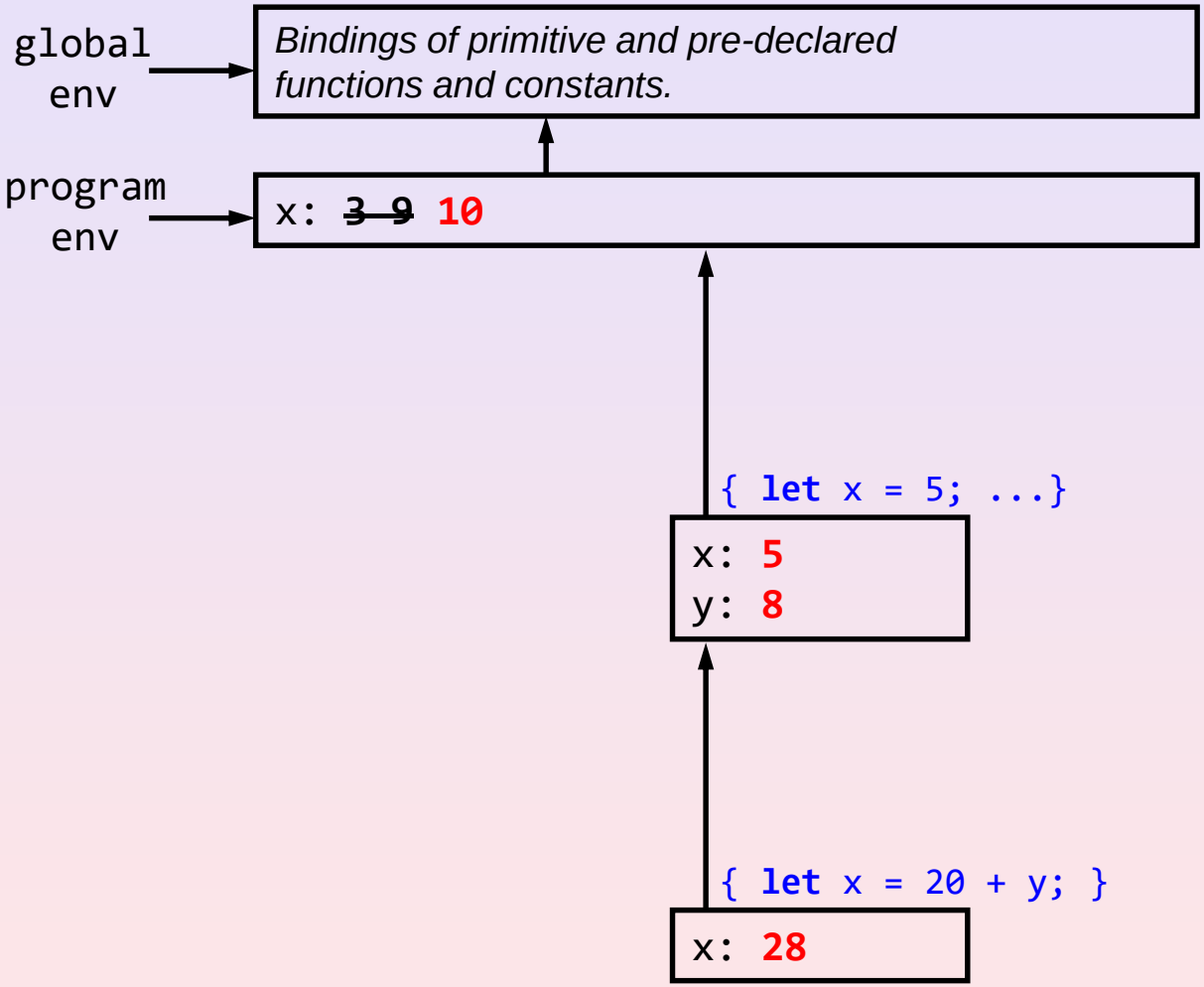

```
let x = 3;

if (x <= 1) {
    let y = x;
    x = y + 1;
} else {
    let z = x;
    x = z * z;
}
{
    x = x + 1;
}
{
    let x = 5;
    let y = 8;
    {
      let x = 20 + y;
    }
    x = x + 1;
}
x;
```



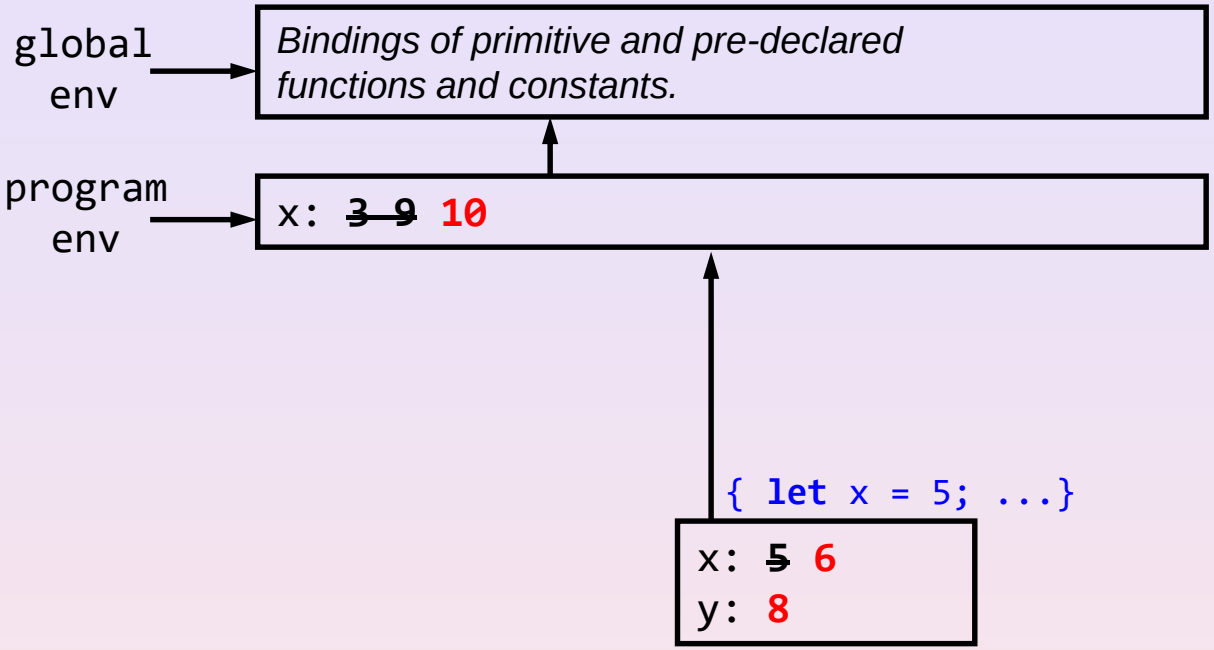

```
let x = 3;

if (x <= 1) {
    let y = x;
    x = y + 1;
} else {
    let z = x;
    x = z * z;
}
{
    x = x + 1;
}
{
    let x = 5;
    let y = 8;
    {
      let x = 20 + y;
    }
    x = x + 1;
}
x;
```



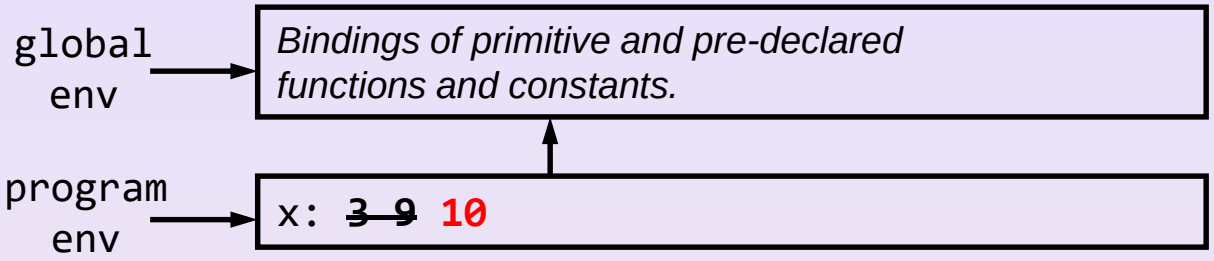

```
let x = 3;

if (x <= 1) {
    let y = x;
    x = y + 1;
} else {
    let z = x;
    x = z * z;
}
{
    x = x + 1;
}
{
    let x = 5;
    let y = 8;
    {
      let x = 20 + y;
    }
    x = x + 1;
}
x;
```



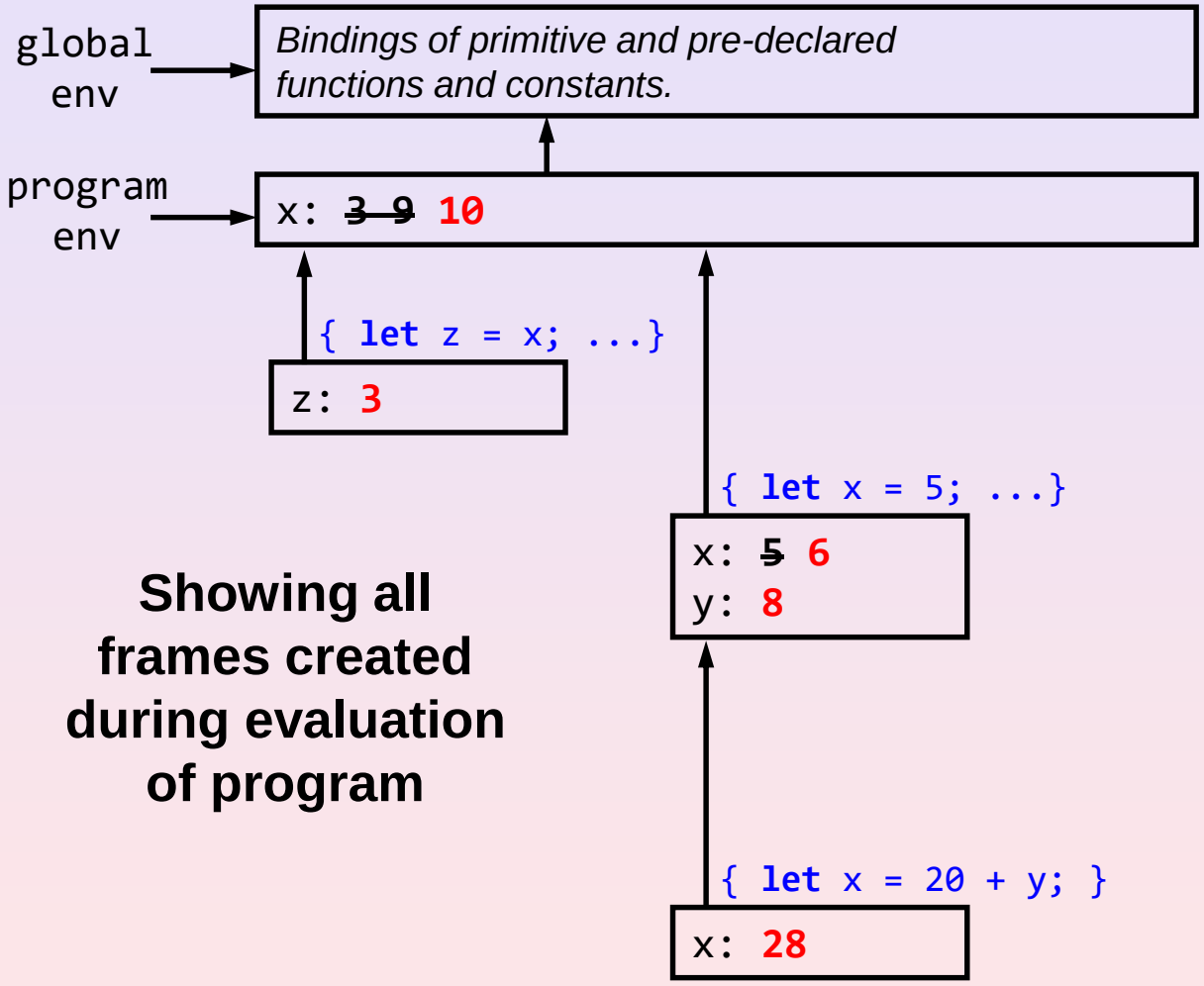

```
let x = 3;

if (x <= 1) {
    let y = x;
    x = y + 1;
} else {
    let z = x;
    x = z * z;
}
{
    x = x + 1;
}
{
    let x = 5;
    let y = 8;
    {
      let x = 20 + y;
    }
    x = x + 1;
}
x;
```




```
let x = 3;

if (x <= 1) {
    let y = x;
    x = y + 1;
} else {
    let z = x;
    x = z * z;
}
{
    x = x + 1;
}
{
    let x = 5;
    let y = 8;
    {
      let x = 20 + y;
    }
    x = x + 1;
}
x;
```




```
let x = 3;

if (x <= 1) {
    let y = x;
    x = y + 1;
} else {
    let z = x;
    x = z * z;
}
{
    x = x + 1;
}
{
    let x = 5;
    let y = 8;
    {
      let x = 20 + y;
    }
    x = x + 1;
}
x;
```



Showing all frames created during evaluation of program

```
let x = 3;

if (x <= 1) {
    let y = x;
    x = y + 1;
} else {
    let z = x;
    x = z * z;
}
{
    x = x + 1;
}
{
    let x = 5;
    let y = 8;
    {
      let x = 20 + y;
    }
    x = x + 1;
}
x;
```

Evaluation of **Function Applications**

- Every **function application** extends the environment in which the ***function was created***
- The **new frame** contains bindings of **parameter variables** to actual arguments
  - New **frame is not created** if the function has **no parameter**
- The **function body block** is evaluated in this new environment

Evaluation of **Lambda Expressions**

- Lambda expression:

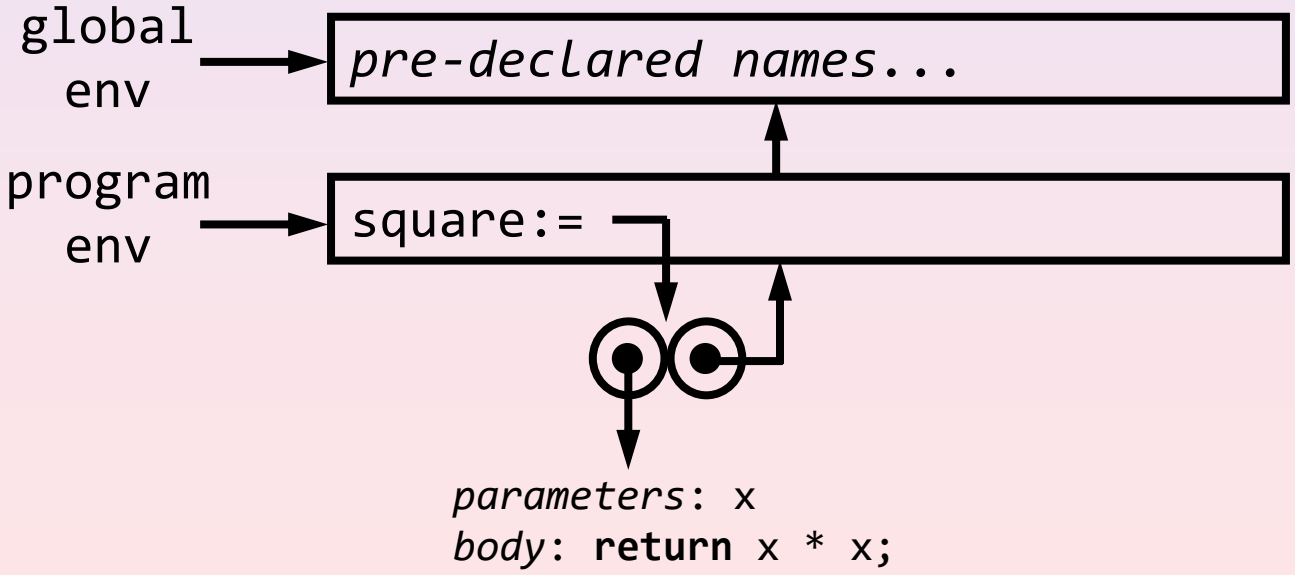
```
(var1, var2, ...) => { body }
```

  - Creates a ***function object***, represented as **two circles**
  - One points to *body* (text)
  - The other points to the environment **in which the function expression was evaluated**
  - A function therefore remembers the **environment in which it was created**

Example for Evaluating Lambda Expressions

```
const square = x => x * x;
square(5);
```

- After evaluating the constant declaration in **program env**:



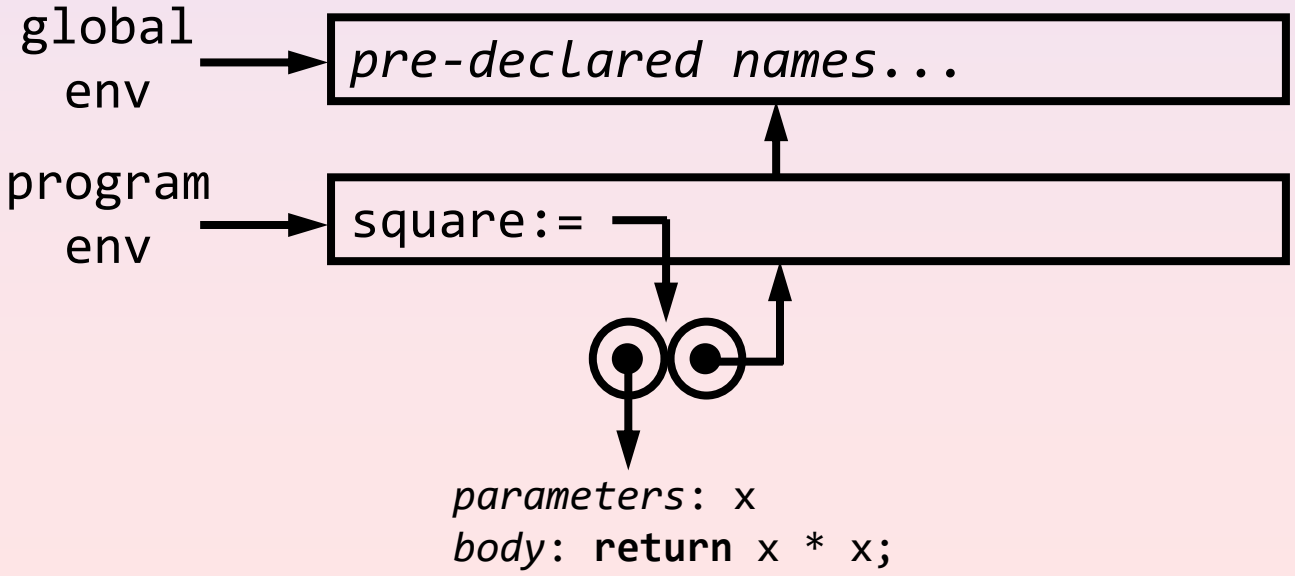
Example for Evaluating Function Declarations

```
function square(x) { return x * x; }
```

which is equivalent to*

```
const square = x => x * x;
```

- After evaluating the function declaration in **program env**:



Evaluation of **Function Applications**

- To evaluate a **function application**:

```
fun(arg1, arg2, ..., argn);
```

- Evaluate the **subexpressions** in the **current environment**

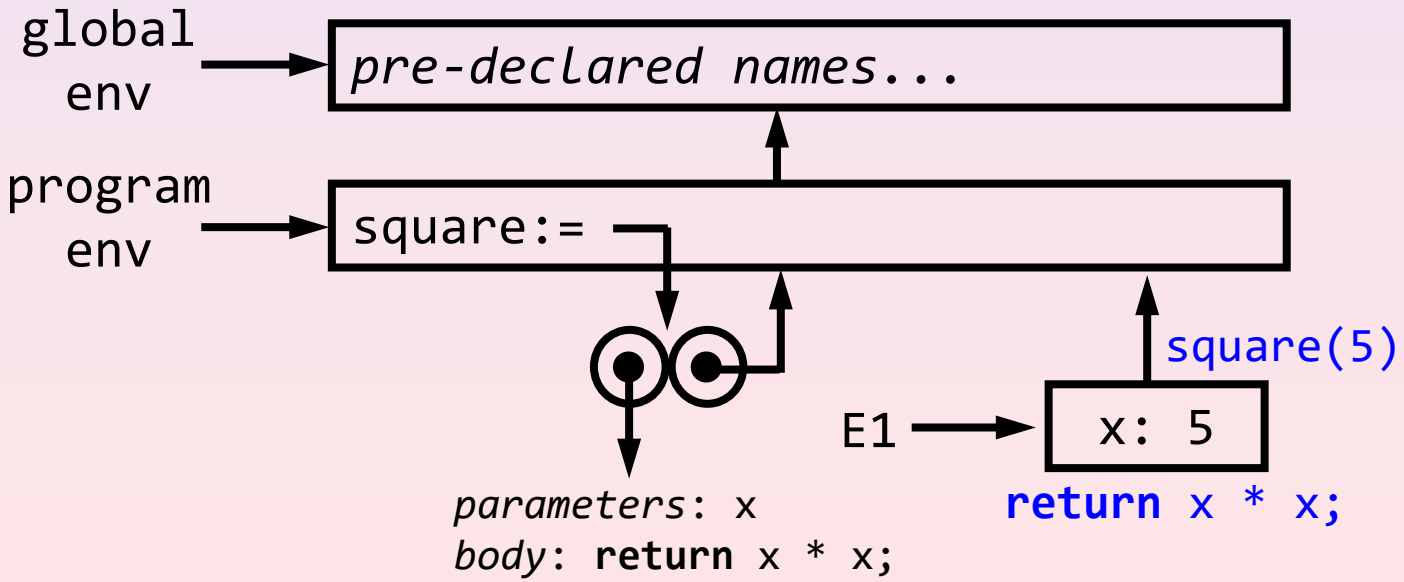
- Apply the value of the **function subexpression** (*fun*) to the values of the **argument subexpressions** (*arg1*, *arg2*, ..., *argn*)

Applying a Function Value to Argument Values

- Create a new frame that **points to the environment of the function**
  - **The environment of the function** is the environment in which the function was created
- In this new frame, **bind the formal parameters to the actual arguments**
- Evaluate the **body block** of the function in the new environment

Example: `square(5);`

- Evaluate subexpressions `square` and `5` in **program env**
- Create new frame that points to environment of `square`
- In this frame, bind value 5 to parameter `x`
- Evaluate body `return x * x;`



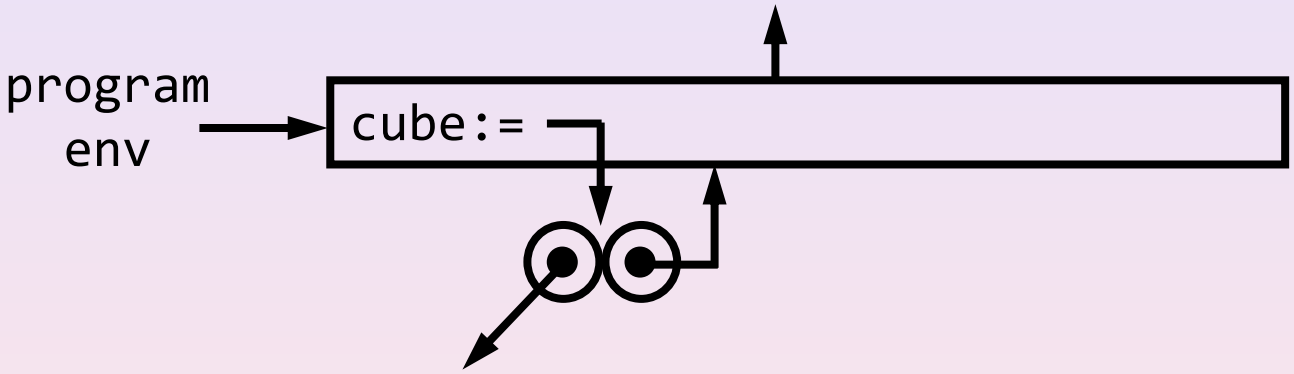
Example with Local Declaration

- The `cube` function has a **local constant** `y`

```
function cube(x) {
    const y = x * x * x;
    return y;
}
cube(3); ➔ 27
```

cube Example

- After evaluating the declaration of function cube

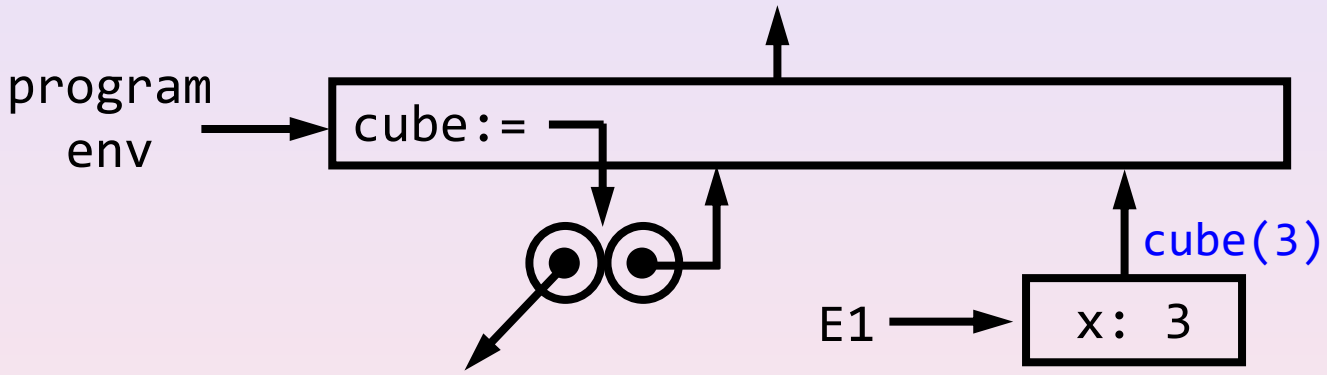


parameters: x
body:
```
const y = x*x*x;
return y;
```

```
function cube(x) {
    const y = x * x * x;
    return y;
}
cube(3);
```

cube Example

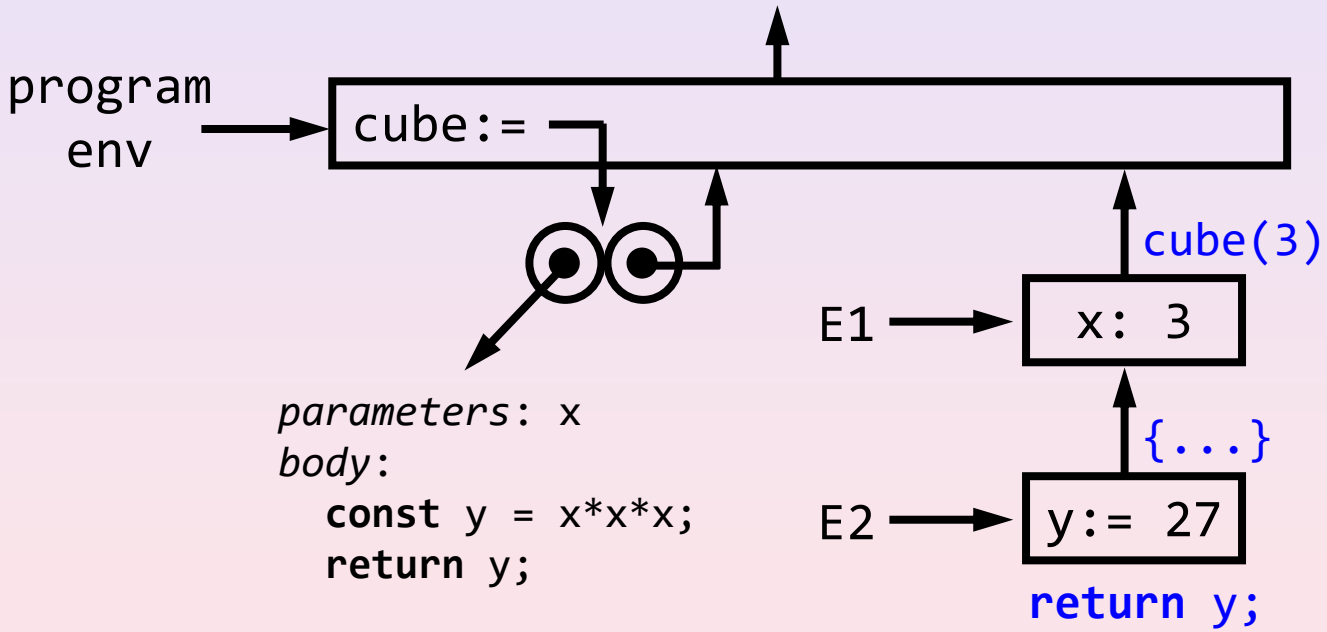
- Evaluate `cube(3)` in **program env**




```
function cube(x) {
    const y = x * x * x;
    return y;
}
cube(3);
```

cube Example

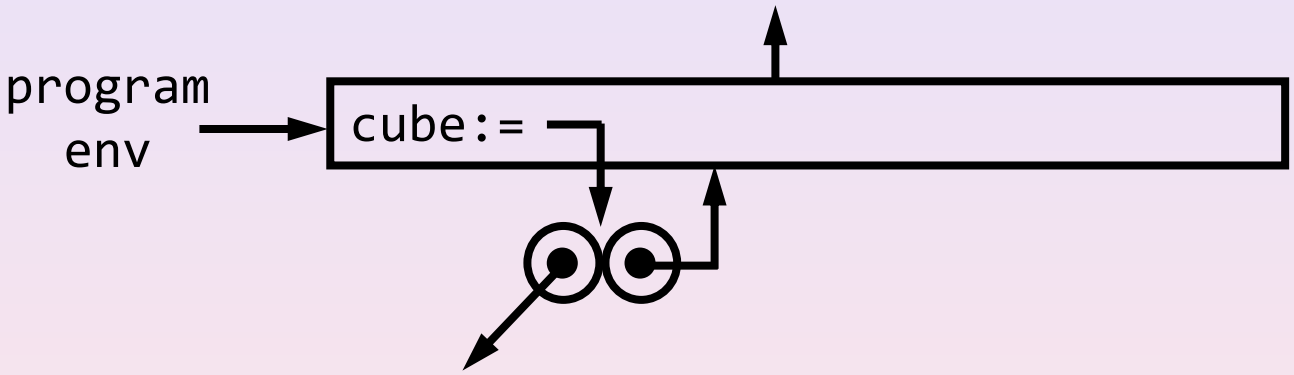
- Evaluate function **body block**




```
function cube(x) {
    const y = x * x * x;
    return y;
}
cube(3);
```

cube Example

- Environment after evaluation of `cube(3)`



parameters: x
body:

```
const y = x*x*x;
return y;
```

```
function cube(x) {
    const y = x * x * x;
    return y;
}
cube(3);
```

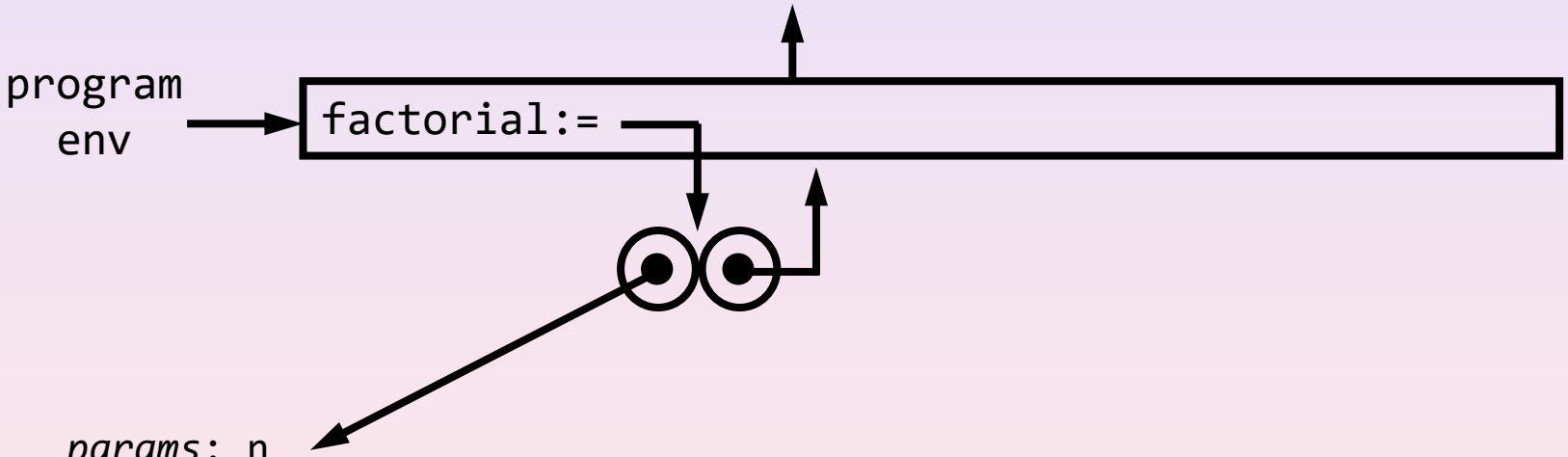
Example with Recursion

- Recursive `factorial` function

```
function factorial(n) {
    return n <= 1
           ? 1
           : n * factorial(n - 1);
}
factorial(2); ➔ 2
```

factorial Example

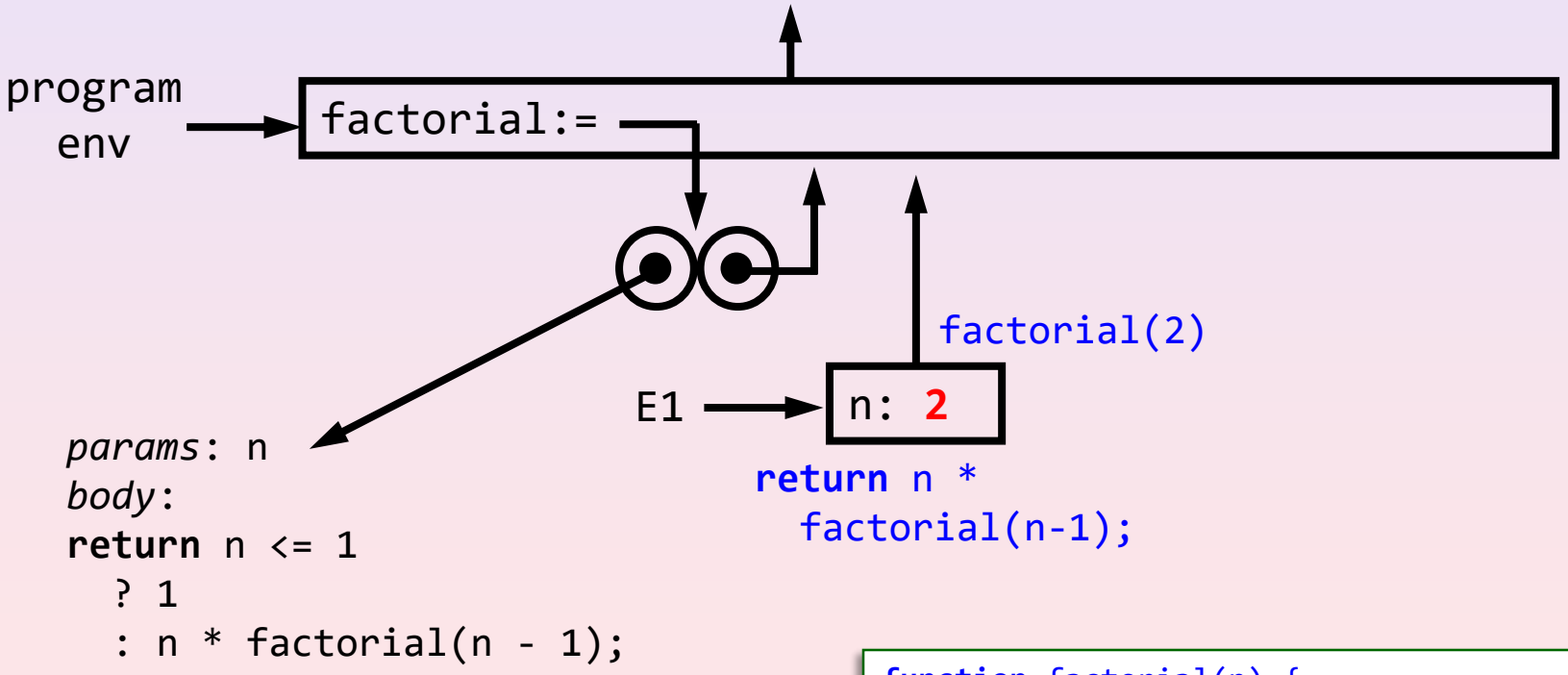
- After evaluating the declaration of function `factorial`




```
function factorial(n) {
    return n <= 1 ? 1 : n * factorial(n - 1);
}
factorial(2);
```

factorial Example

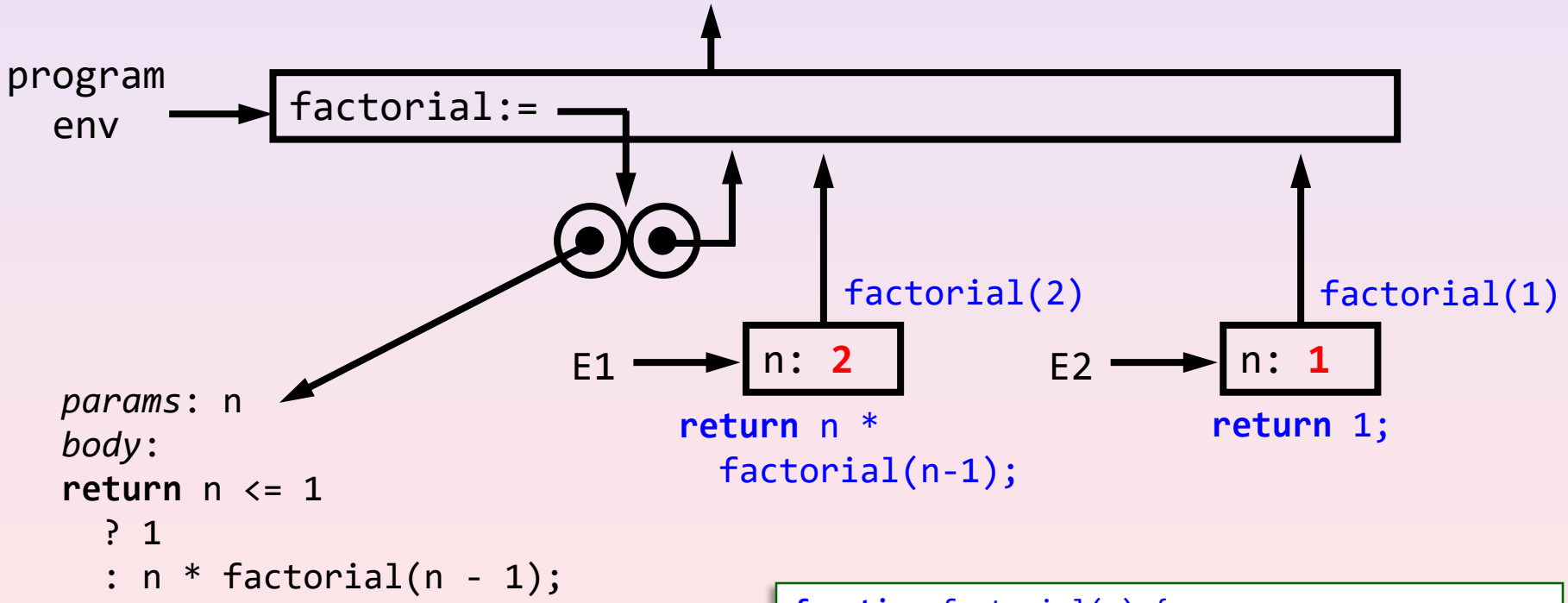
- Evaluate `factorial(2)` in **program env**




```
function factorial(n) {
    return n <= 1 ? 1 : n * factorial(n - 1);
}
factorial(2);
```

factorial Example

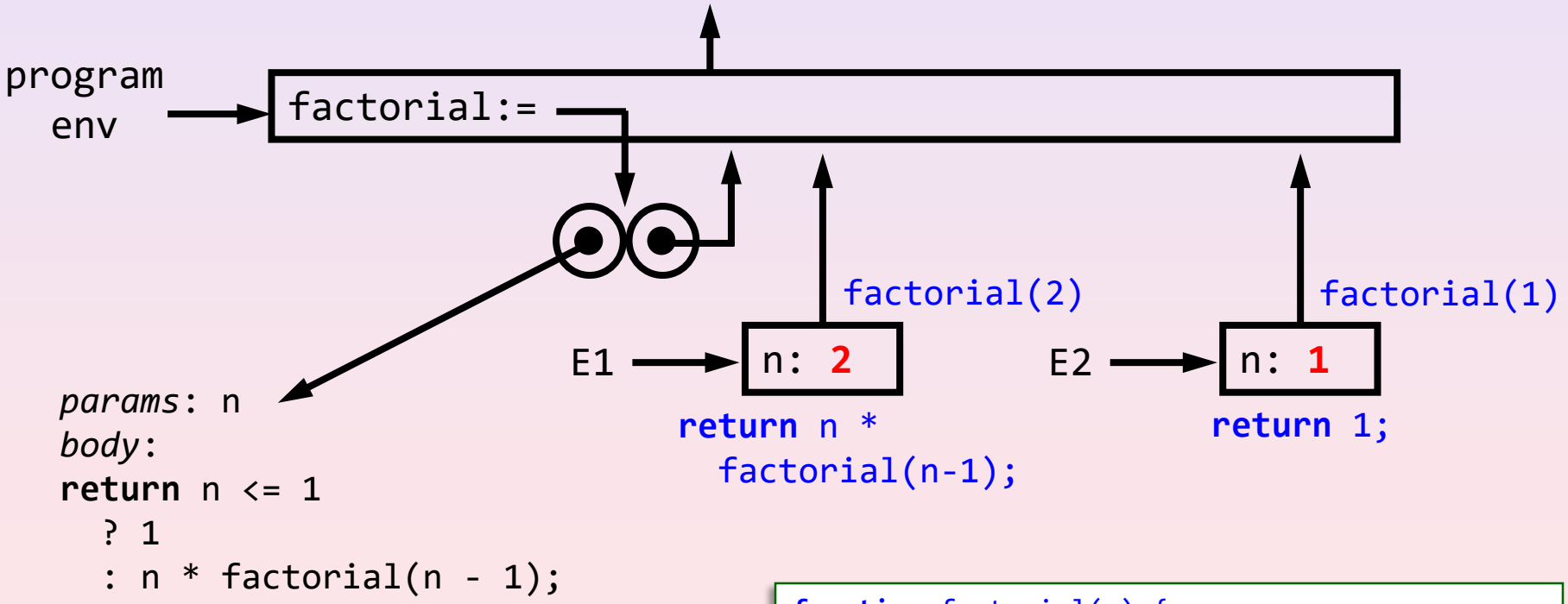
- Evaluate `factorial(n-1)` in **Environment E1**




```
function factorial(n) {
    return n <= 1 ? 1 : n * factorial(n - 1);
}
factorial(2);
```

factorial Example

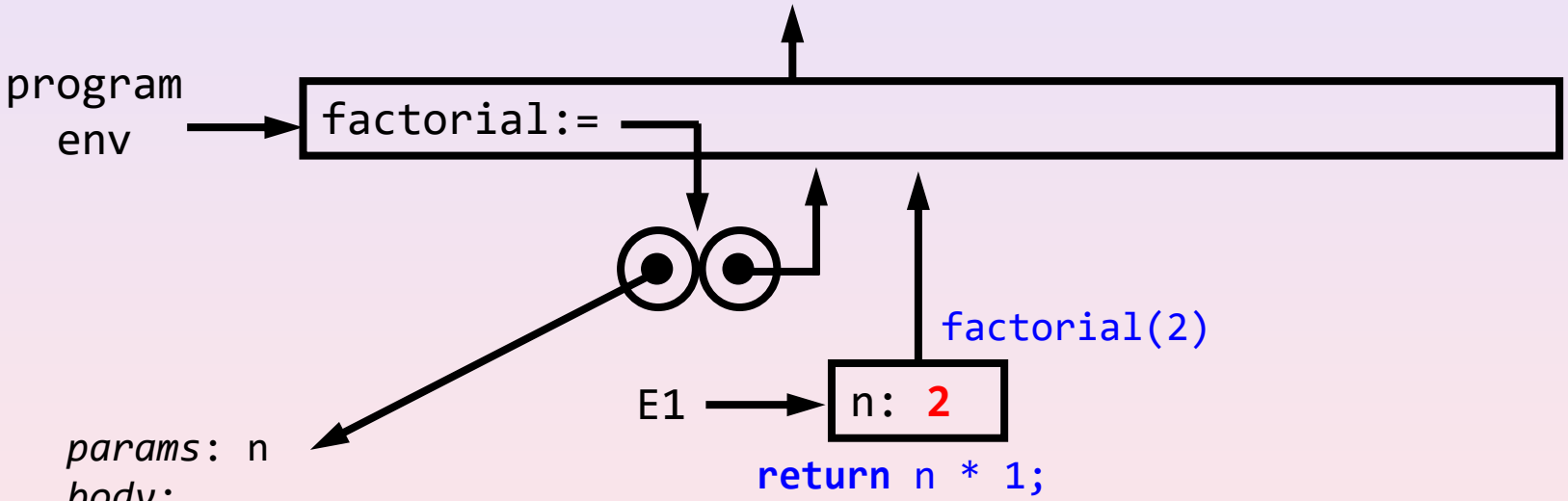
- Evaluate `return 1;` in **Environment E2**




```
function factorial(n) {
    return n <= 1 ? 1 : n * factorial(n - 1);
}
factorial(2);
```

factorial Example

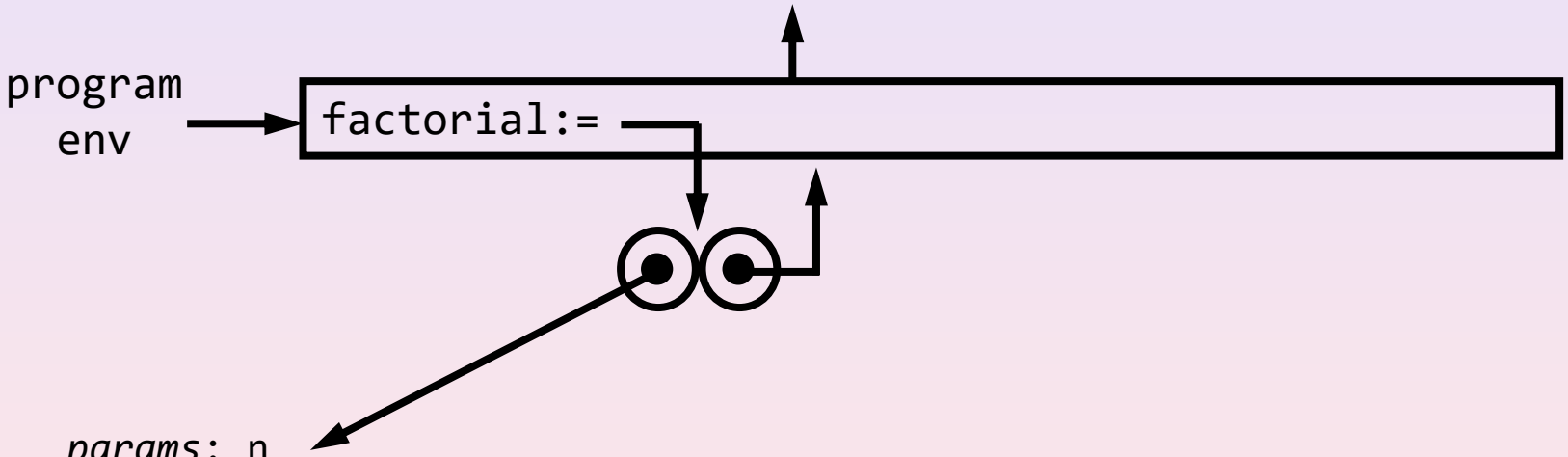
- Evaluate `return n * 1;` in **Environment E1**




```
function factorial(n) {
    return n <= 1 ? 1 : n * factorial(n - 1);
}
factorial(2);
```

factorial Example

- Environment after evaluation of `factorial(2)`



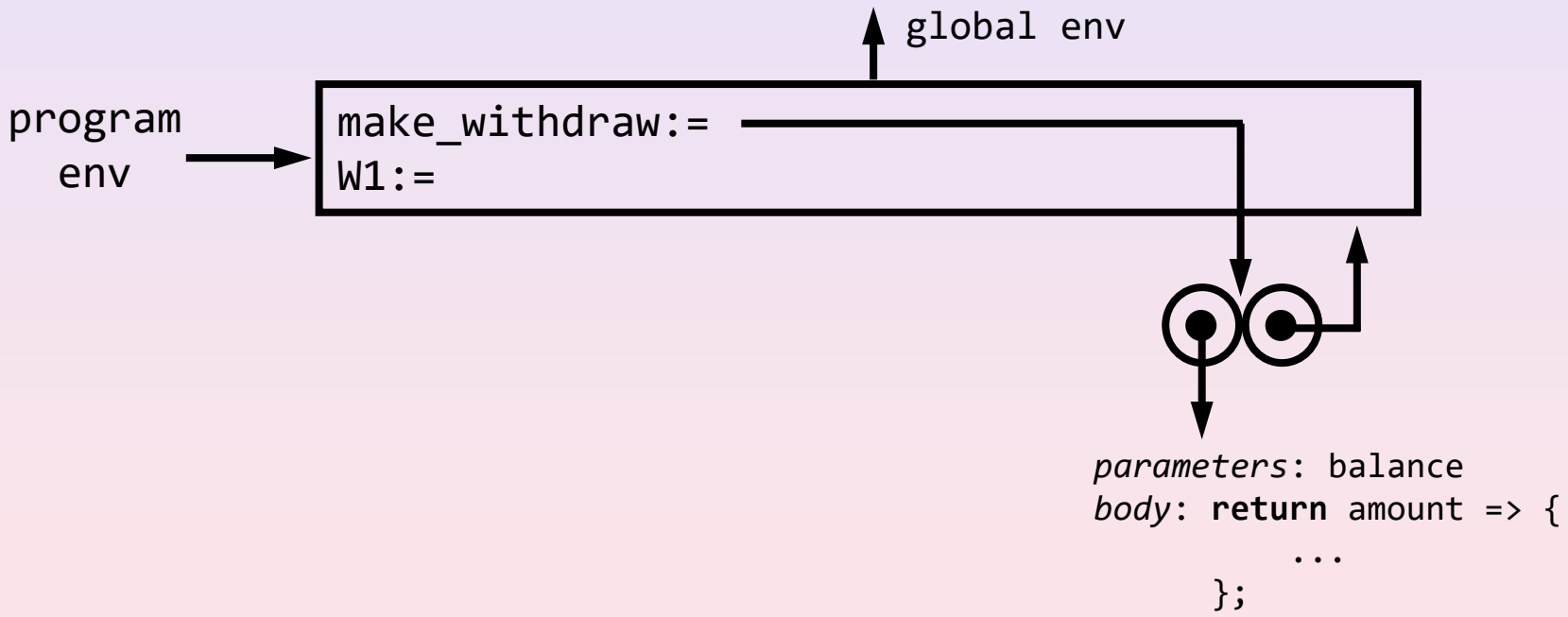

```
function factorial(n) {
    return n <= 1 ? 1 : n * factorial(n - 1);
}
factorial(2);
```

make_withdraw Example

```
function make_withdraw(balance) {
    return amount => {
        if (balance >= amount) {
            balance = balance - amount;
            return balance;
        } else {
            return "Insufficient funds";
        }
    };
}

const W1 = make_withdraw(100);
W1(40); ➔ 60
```

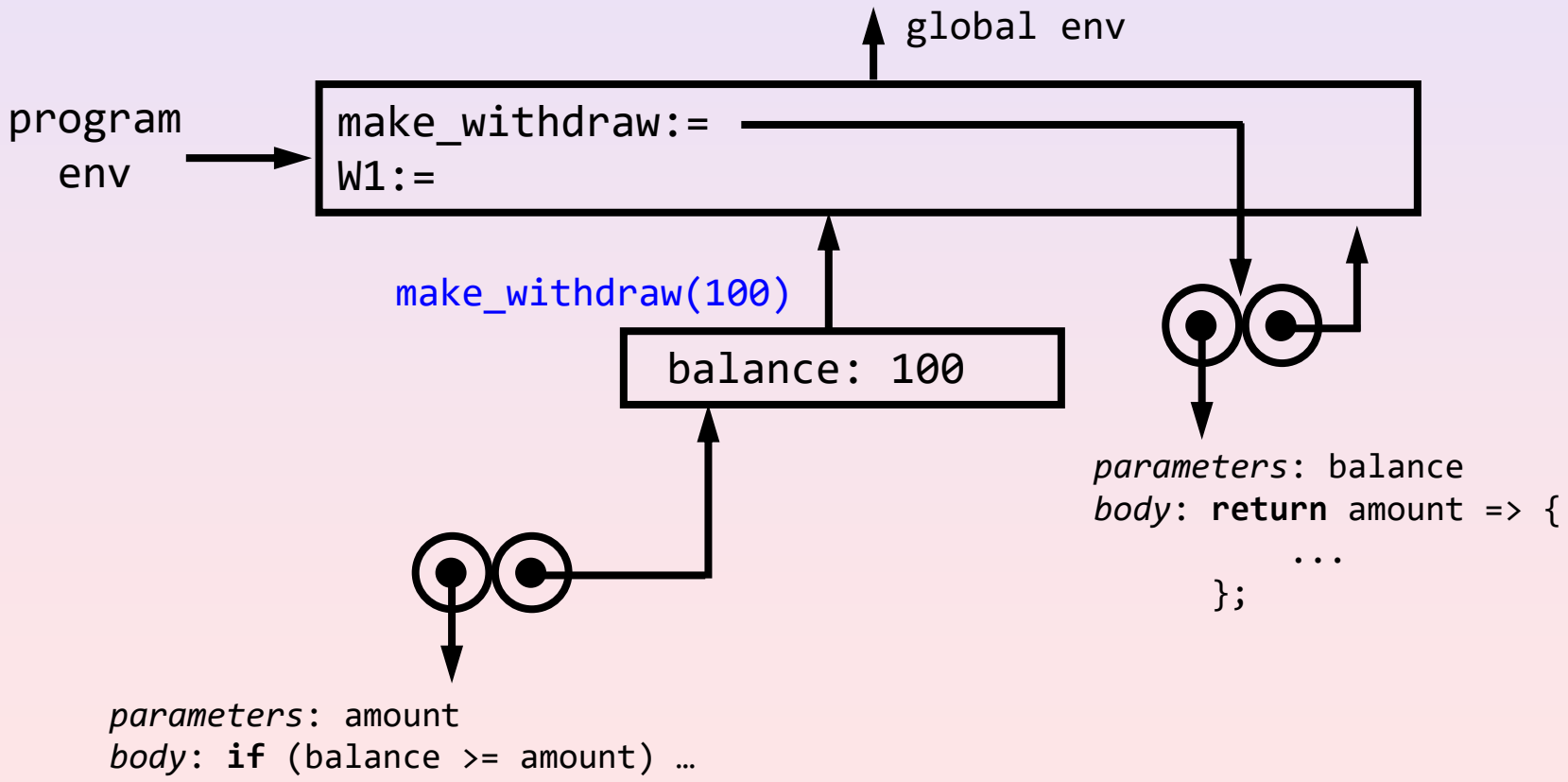
make_withdraw Example

- After declaring function `make_withdraw`



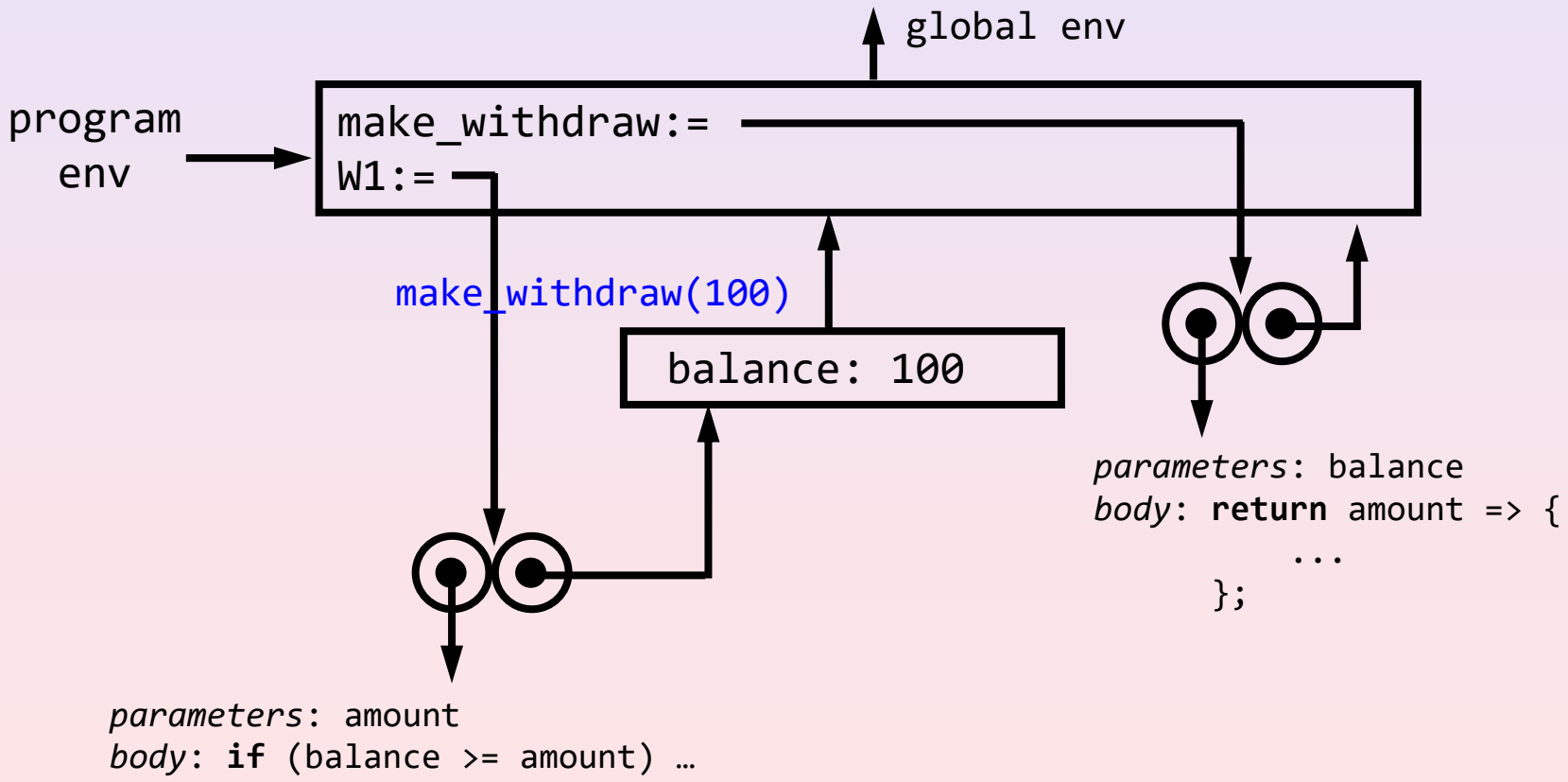
make_withdraw Example

- Evaluate `make_withdraw(100)`



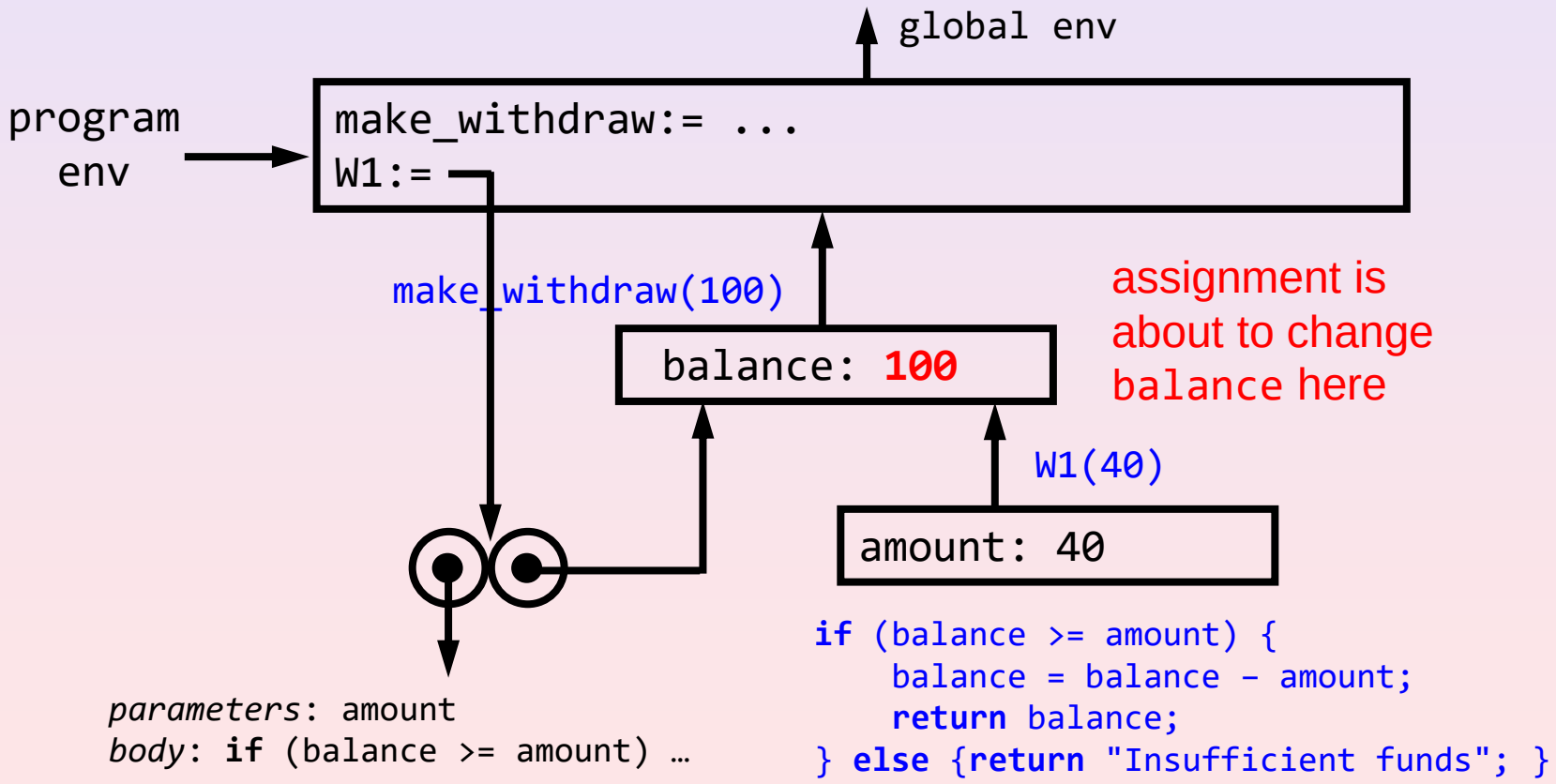
make_withdraw Example

- Evaluate **const** W1 = make_withdraw(100);



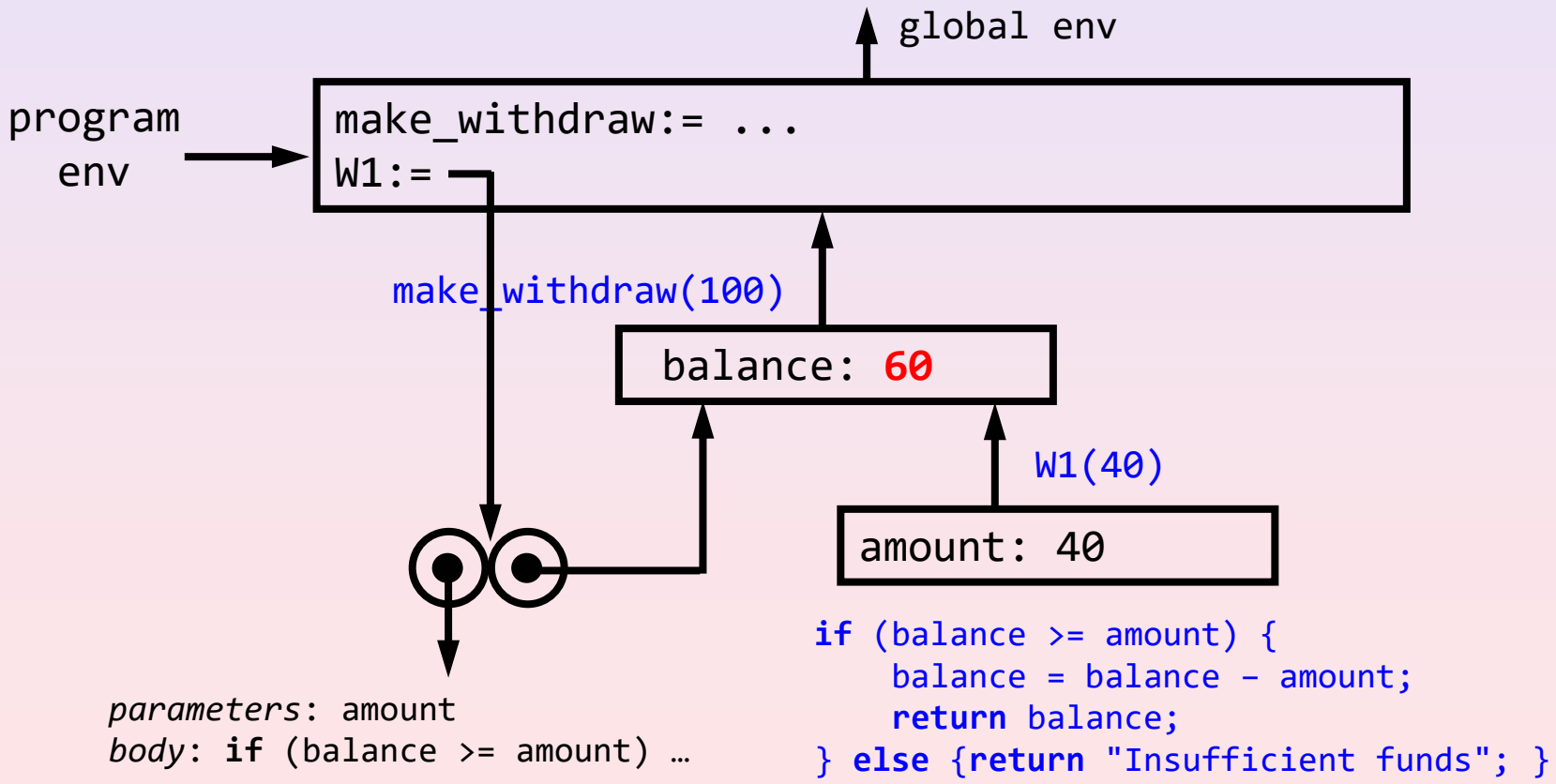
make_withdraw Example

- Evaluate `W1(40)`



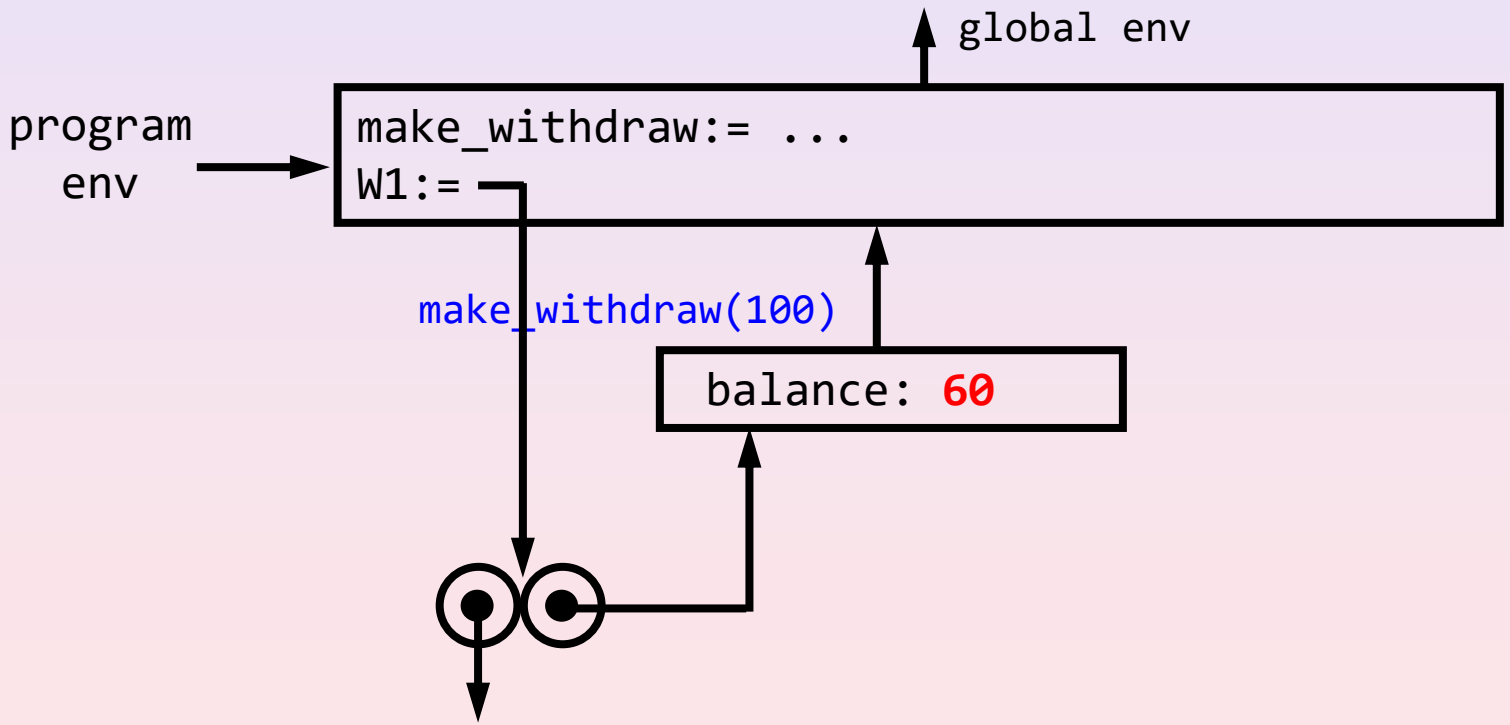
make_withdraw Example

- Evaluate `W1(40)`



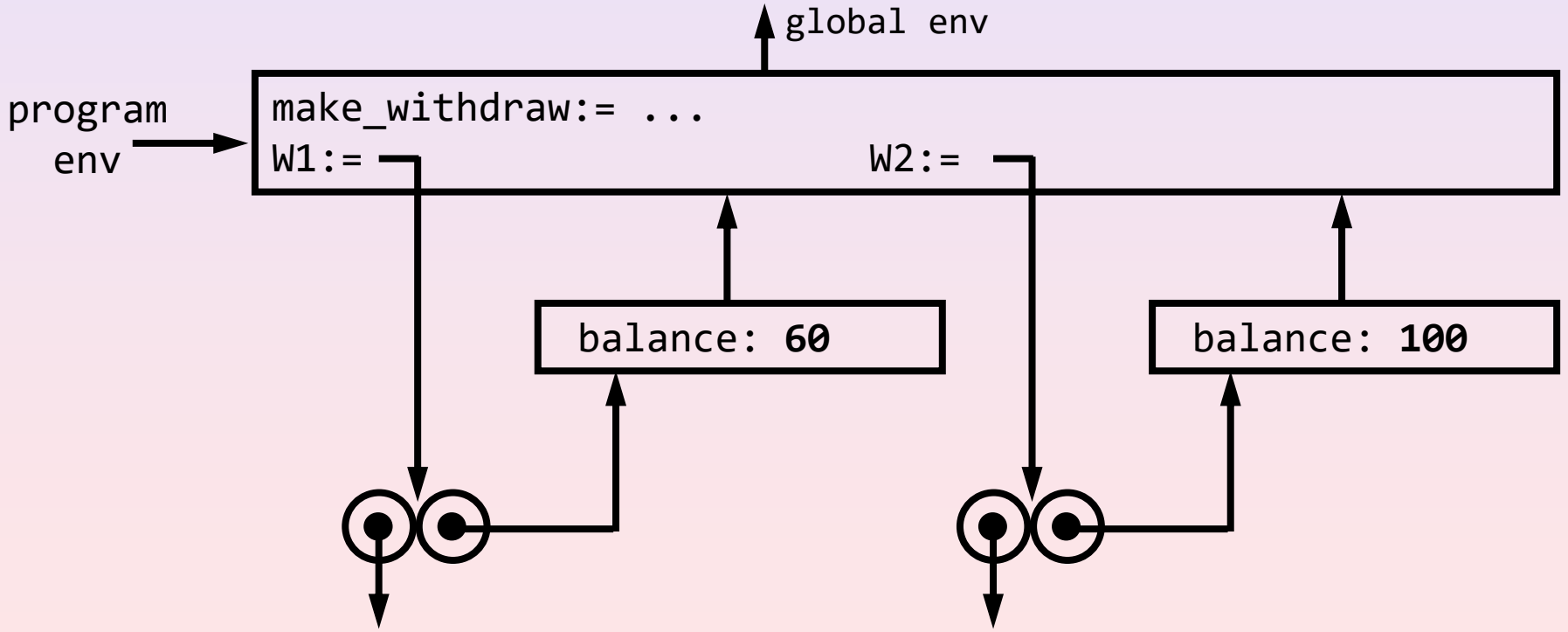
make_withdraw Example

- Environment after evaluating `W1(40);`



make_withdraw Example

- Environment after evaluating `const W2 = make_withdraw(100);`
- `W2` has its own state variable



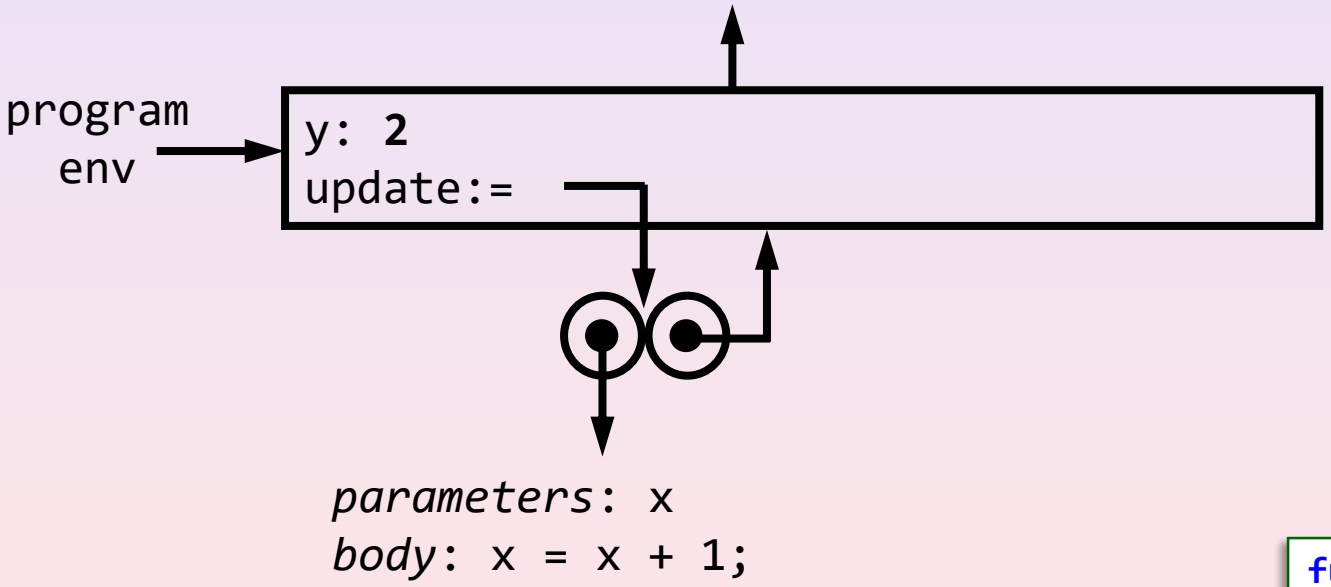
update Example

- What is the result of this program?

```
function update(x) {
    x = x + 1;
}
let y = 2;
update(y);
y; ➔ 2
```

update Example

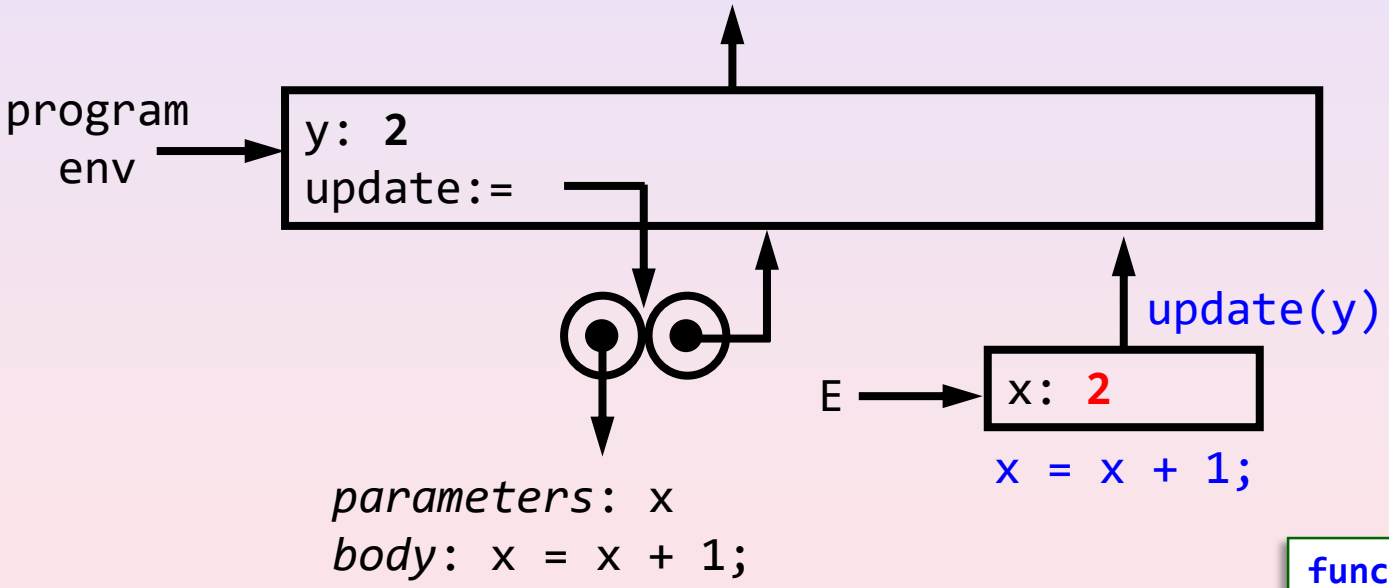
- After evaluating declarations of function update and variable y




```
function update(x) {
    x = x + 1;
}
let y = 2;
update(y);
y;
```

update Example

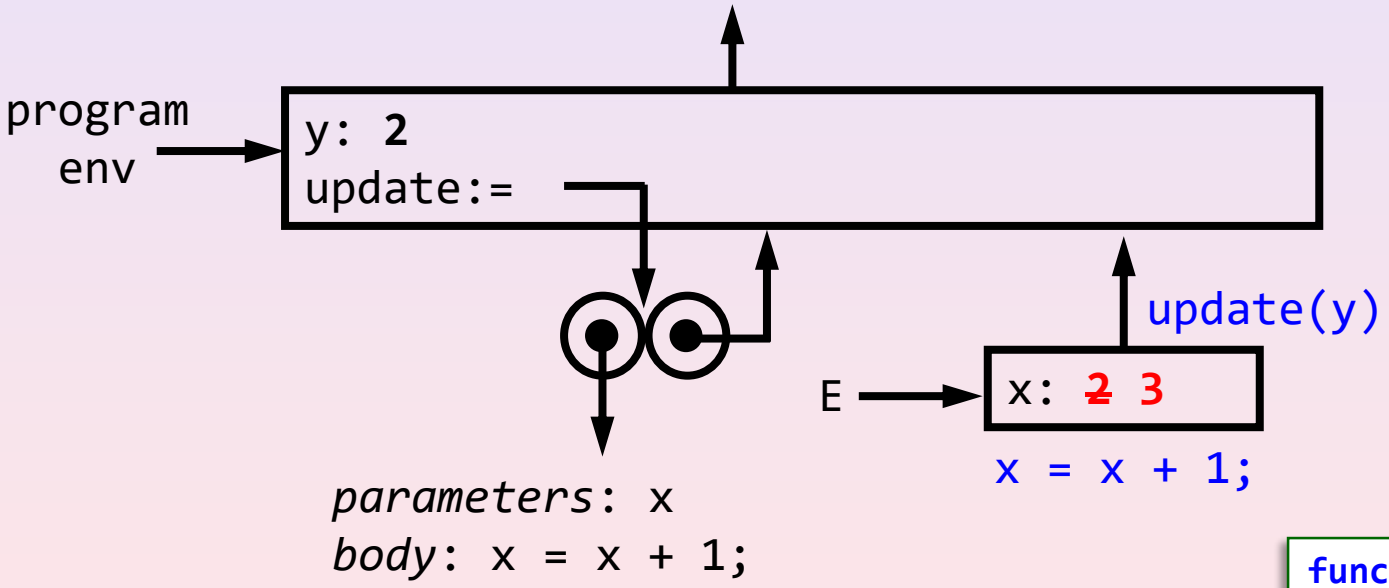
- Evaluate update(y) in **program env**




```
function update(x) {
    x = x + 1;
}
let y = 2;
update(y);
y;
```

update Example

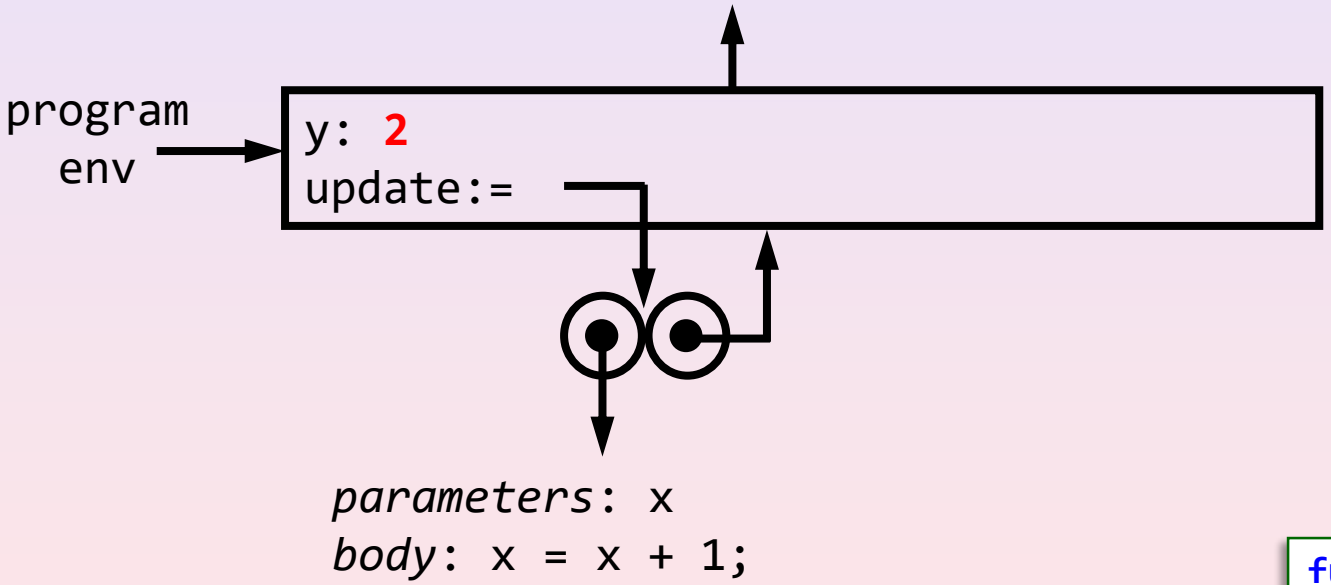
- Evaluate `x = x + 1;` in **Environment E**




```
function update(x) {
    x = x + 1;
}
let y = 2;
update(y);
y;
```

update Example

- Evaluate y in **program env**




```
function update(x) {
    x = x + 1;
}
let y = 2;
update(y);
y;
```

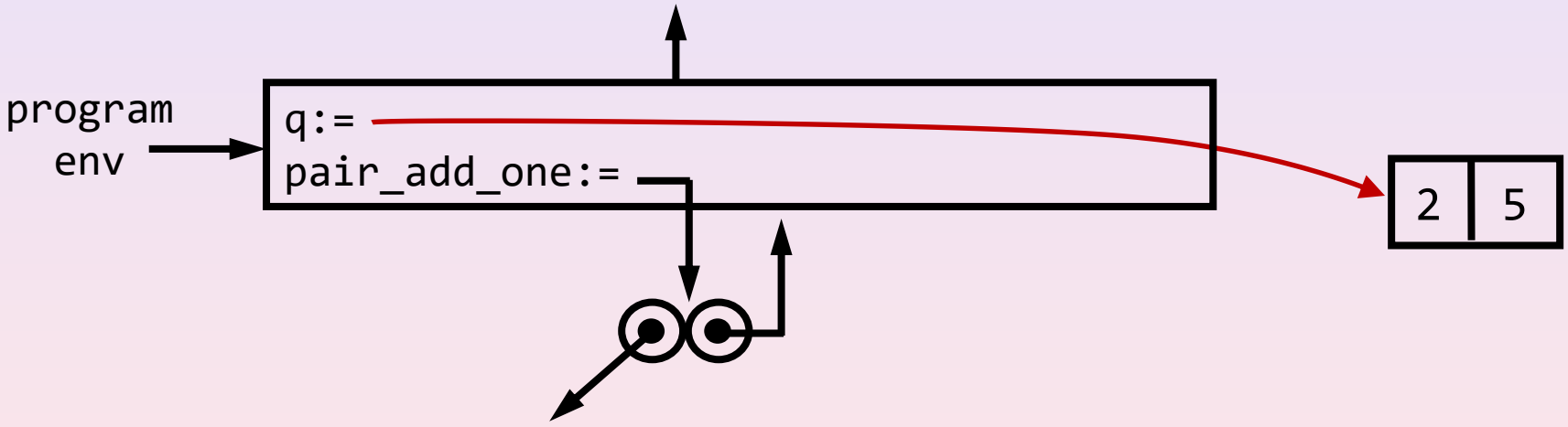
pair_add_one Example

- What is the result of this program?

```
function pair_add_one(p) {
    set_head(p, head(p) + 1);
    set_tail(p, tail(p) + 1);
}
const q = pair(2, 5);
pair_add_one(q);
q; ➔[3, 6]
```

pair_add_one Example

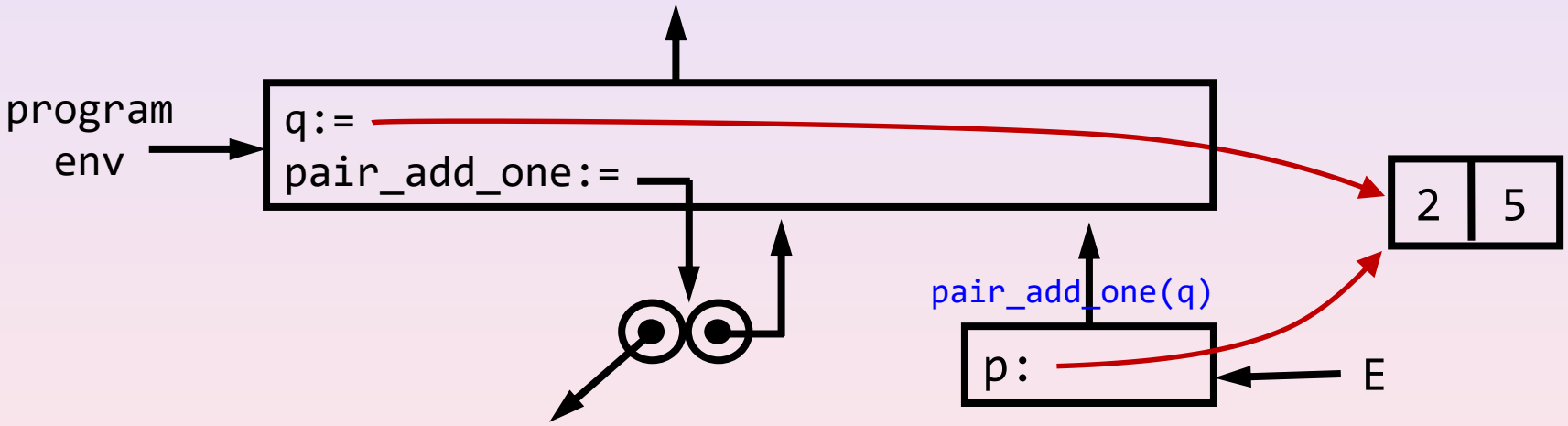
- After evaluating declarations of function `pair_add_one` and constant `q`




```
function pair_add_one(p) {
    set_head(p, head(p) + 1);
    set_tail(p, tail(p) + 1);
}
const q = pair(2, 5);
pair_add_one(q);  q;
```

pair_add_one Example

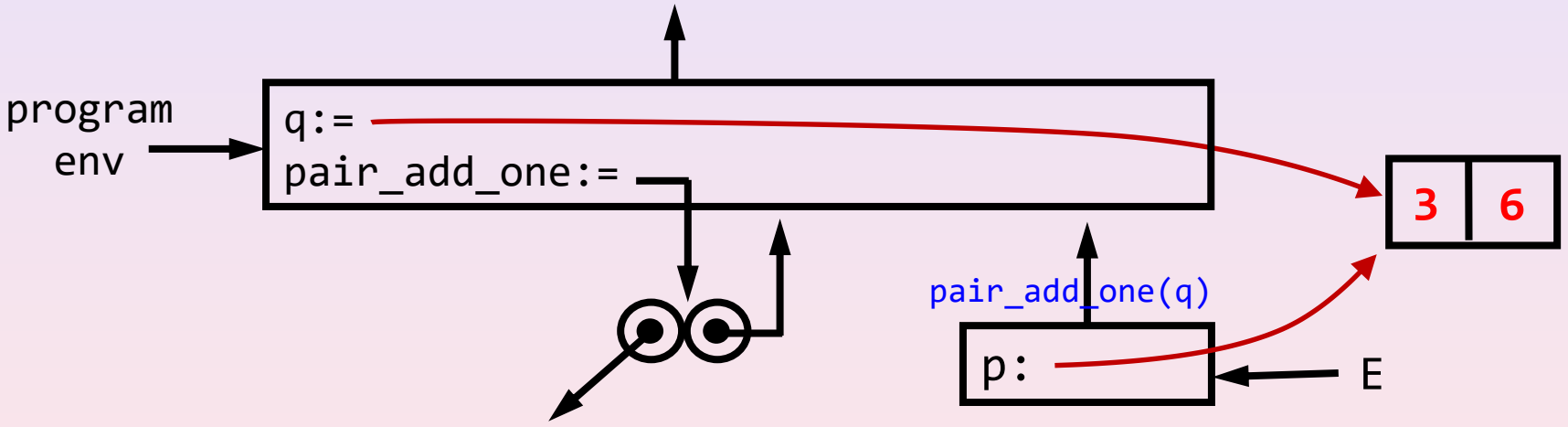
- Evaluate `pair_add_one(q)` in **program env**




```
function pair_add_one(p) {
    set_head(p, head(p) + 1);
    set_tail(p, tail(p) + 1);
}
const q = pair(2, 5);
pair_add_one(q);  q;
```

pair_add_one Example

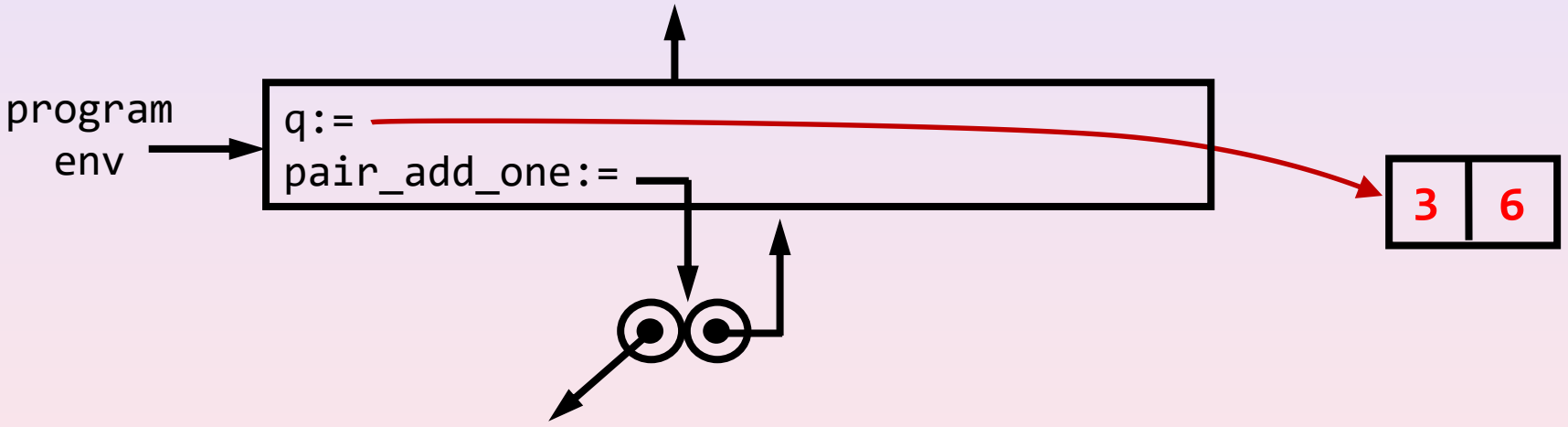
- Evaluate `set_head(p, head(p) + 1)` and `set_tail(p, tail(p) + 1)` in **Environment E**




```
function pair_add_one(p) {
    set_head(p, head(p) + 1);
    set_tail(p, tail(p) + 1);
}
const q = pair(2, 5);
pair_add_one(q);  q;
```

pair_add_one Example

- Evaluate q in **program env**

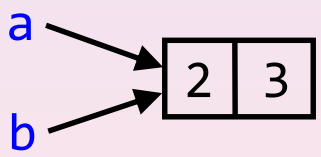



```
function pair_add_one(p) {
    set_head(p, head(p) + 1);
    set_tail(p, tail(p) + 1);
}
const q = pair(2, 5);
pair_add_one(q);  q;
```

Sharing and Identity

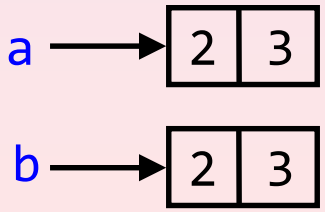
- In `pair_add_one` example, `q` and `p` **share** the same **pair**
- Checking whether **two names** are **sharing** the same **pair**
  - **Example:**

```
const a = pair(2, 3);
const b = a;
a === b; ➔ true
```



  - **Example:**

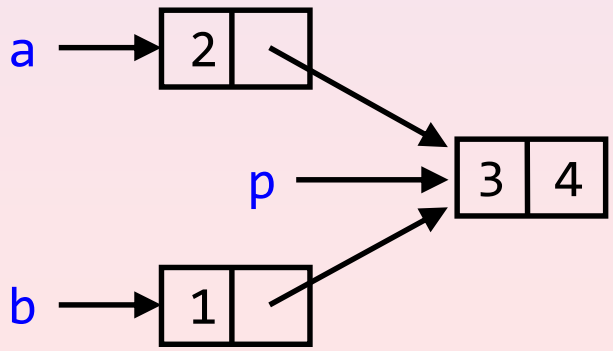
```
const a = pair(2, 3);
const b = pair(2, 3);
a === b; ➔ false
```



Sharing and Identity

- **Structures** sharing pairs
  - **Example:**

```
const p = pair(3, 4);
const a = pair(2, p);
const b = pair(1, p);
tail(a) === tail(b); ➔ true
```

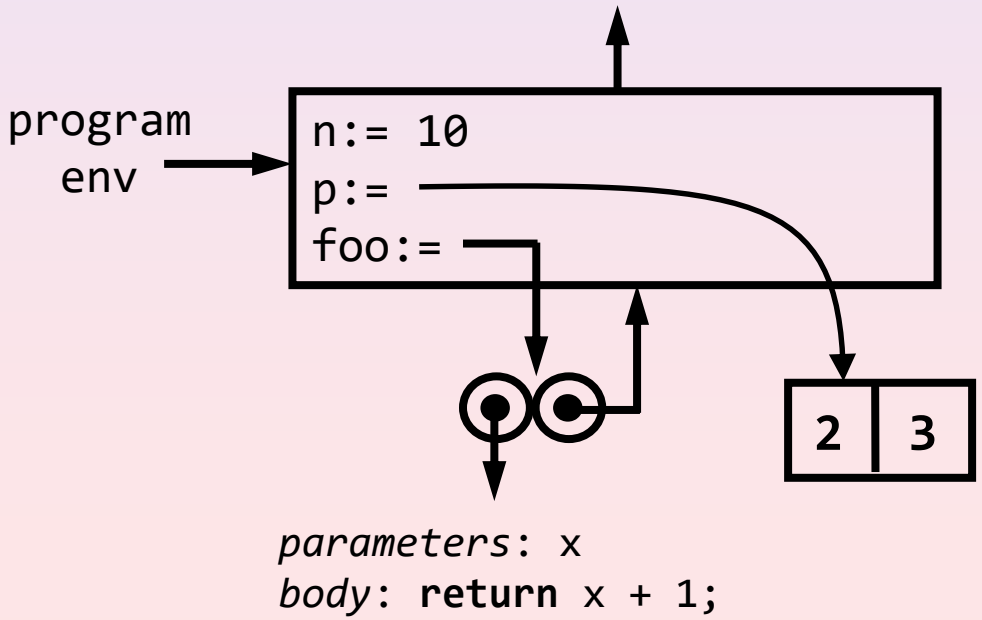


Identity in Source §3: ===

- For `a === b`, we say "is `a` ***identical*** to `b`"
  - `true`, `false`, `null`, `undefined` — each is identical to itself and nothing else
  - **Numbers** — two numbers are identical iff they have the same representation in the double-precision floating-point representation
  - **Strings** — two strings are identical if they have the same characters in the same order
  - **Functions** — functions are made by function expressions, and their creation bestows an identity upon them
  - **Pairs** — pairs are made by the `pair` function, and their creation bestows an identity upon them

Drawing Compound Structures

- **Primitive values** (e.g. numbers, strings, Boolean values, `null`) in bindings are drawn **inside** frames
- **Compound structures** (e.g. pairs, function objects) are drawn **outside** frames

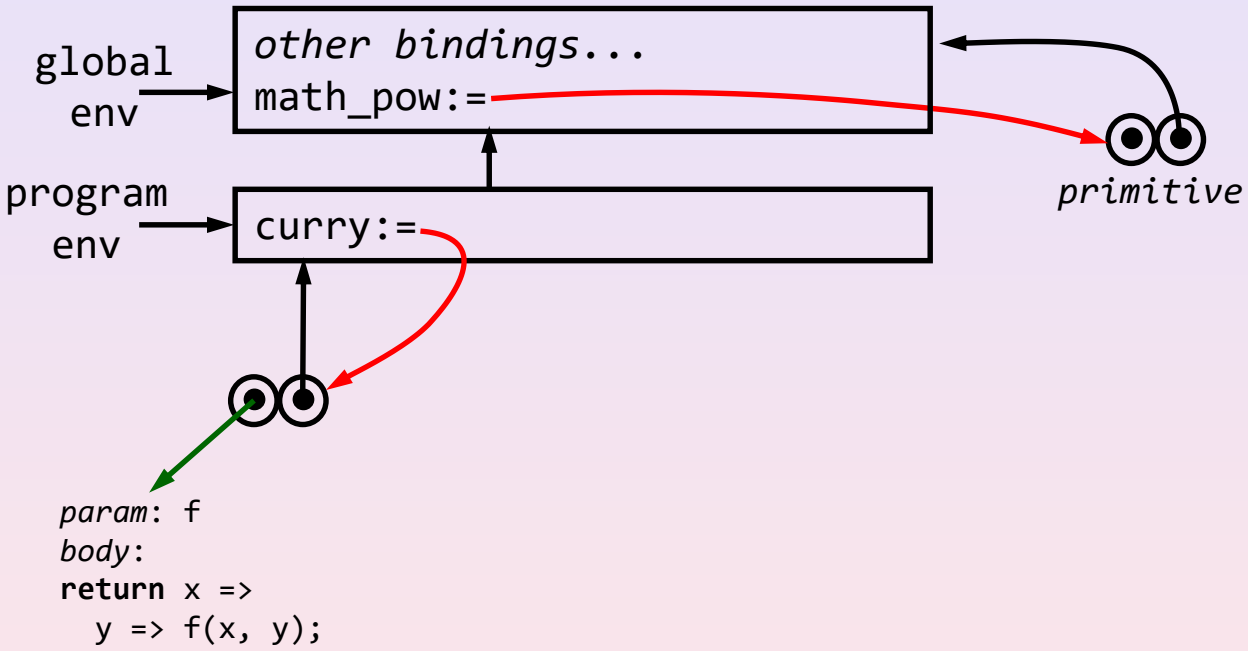


curry Example

- Consider this program

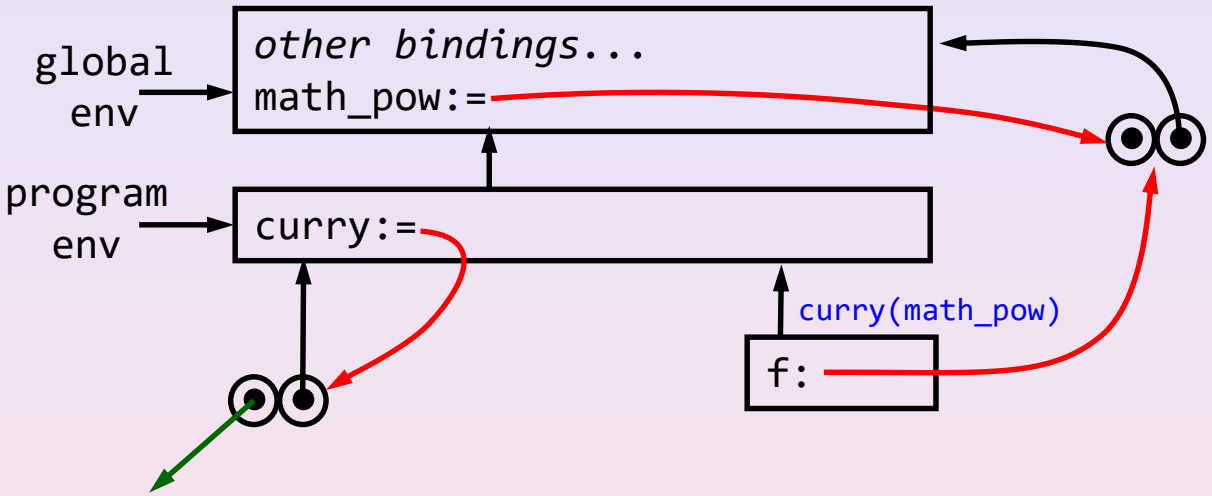
```
function curry(f) {
    return x => y => f(x, y);
}
curry(math_pow)(3)(4);
```

curry Example




```
function curry(f) {
    return x => y => f(x, y);
}
curry(math_pow)(3)(4);
```

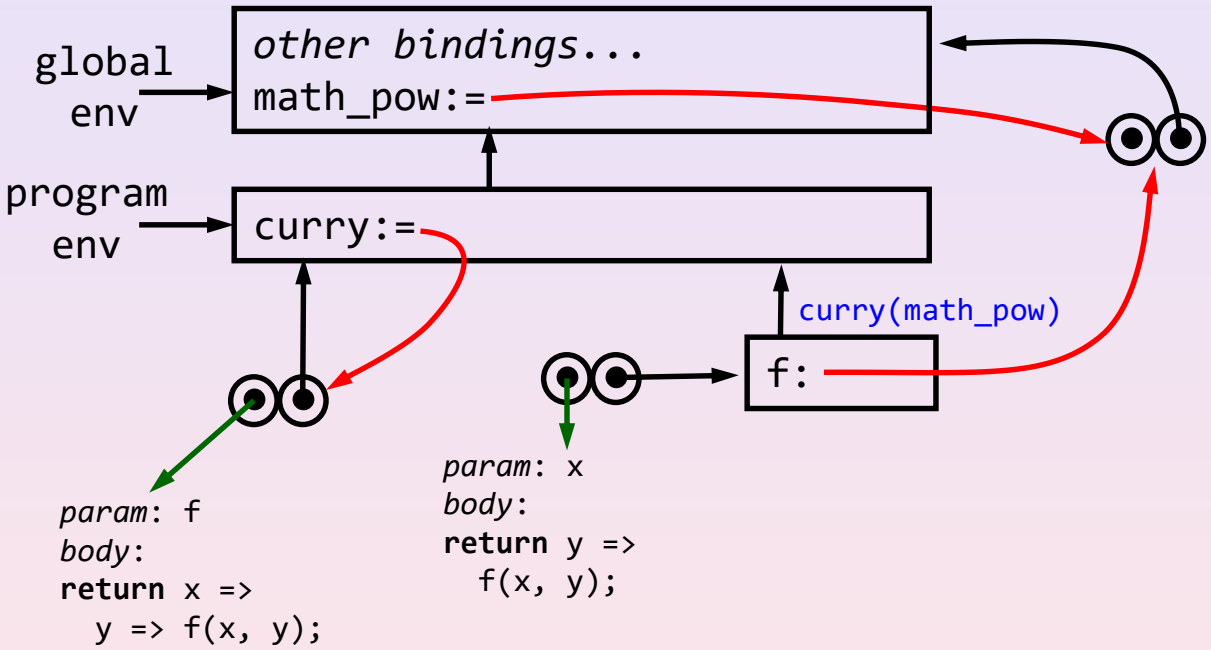
curry Example




```
function curry(f) {
    return x => y => f(x, y);
}
curry(math_pow)(3)(4);
```

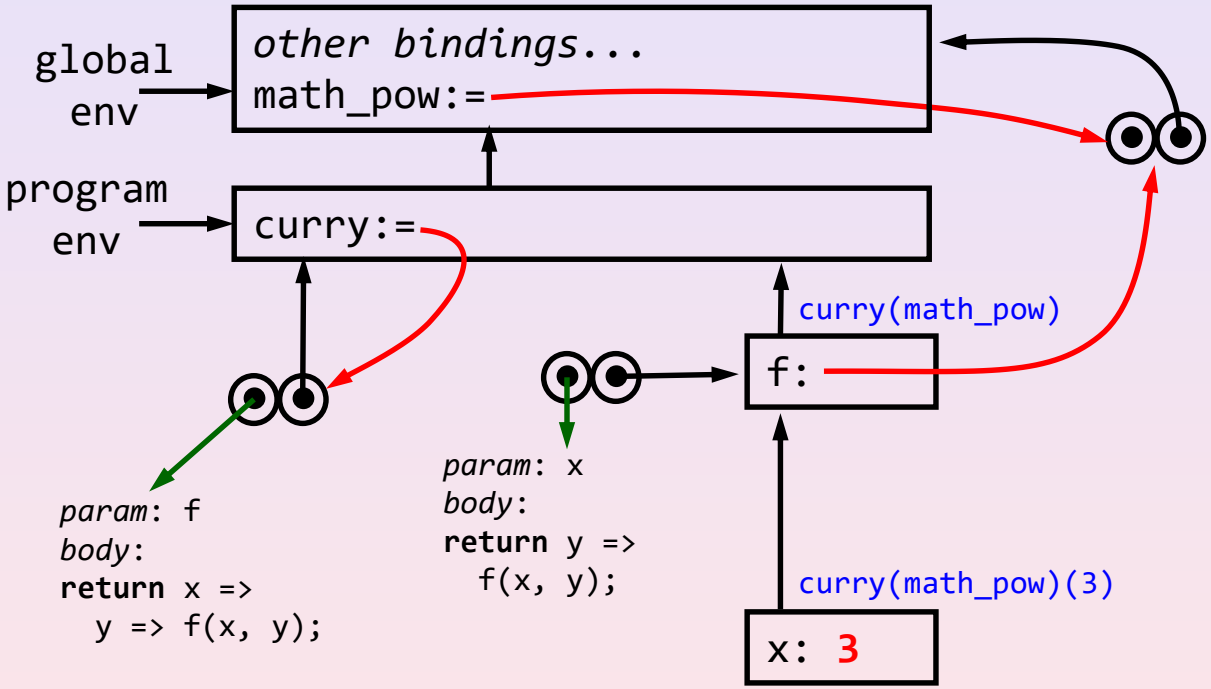
param: f
body:
return x =>
  y => f(x, y);

curry Example



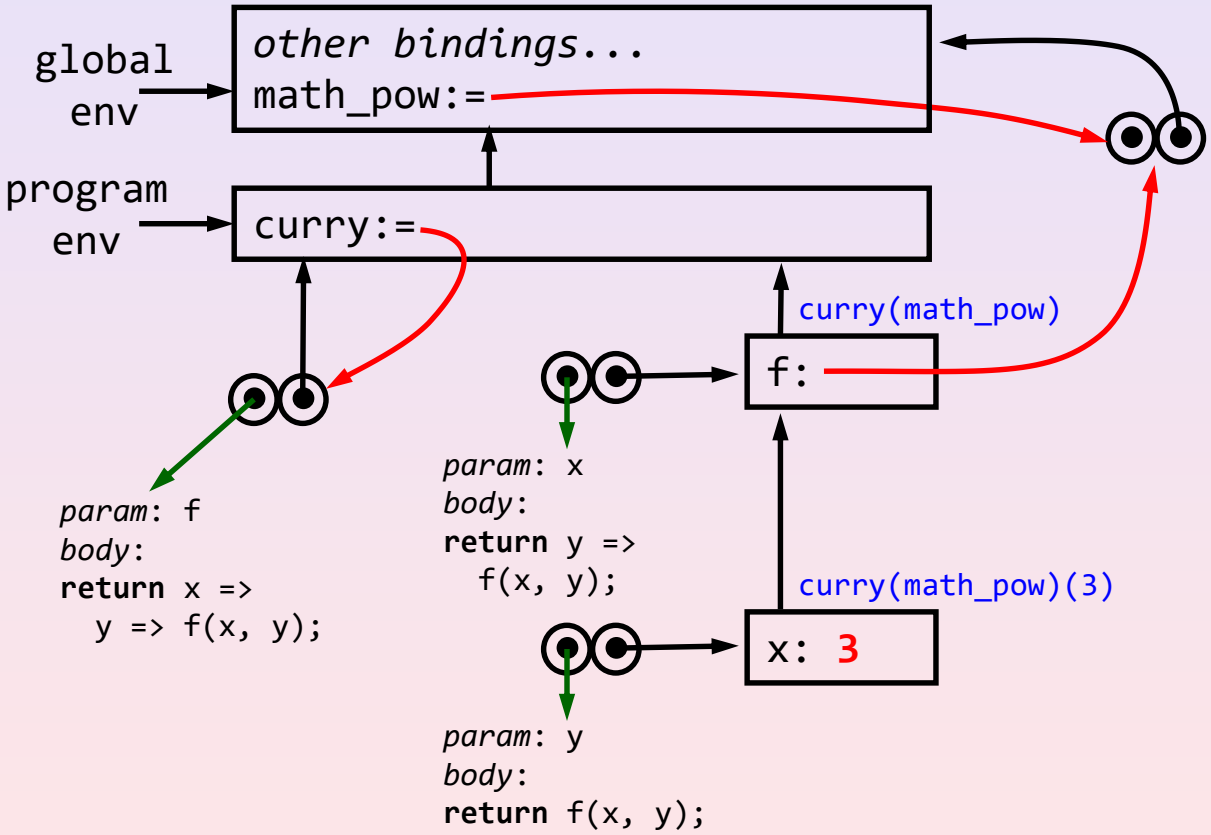

```
function curry(f) {
    return x => y => f(x, y);
}
curry(math_pow)(3)(4);
```

curry Example



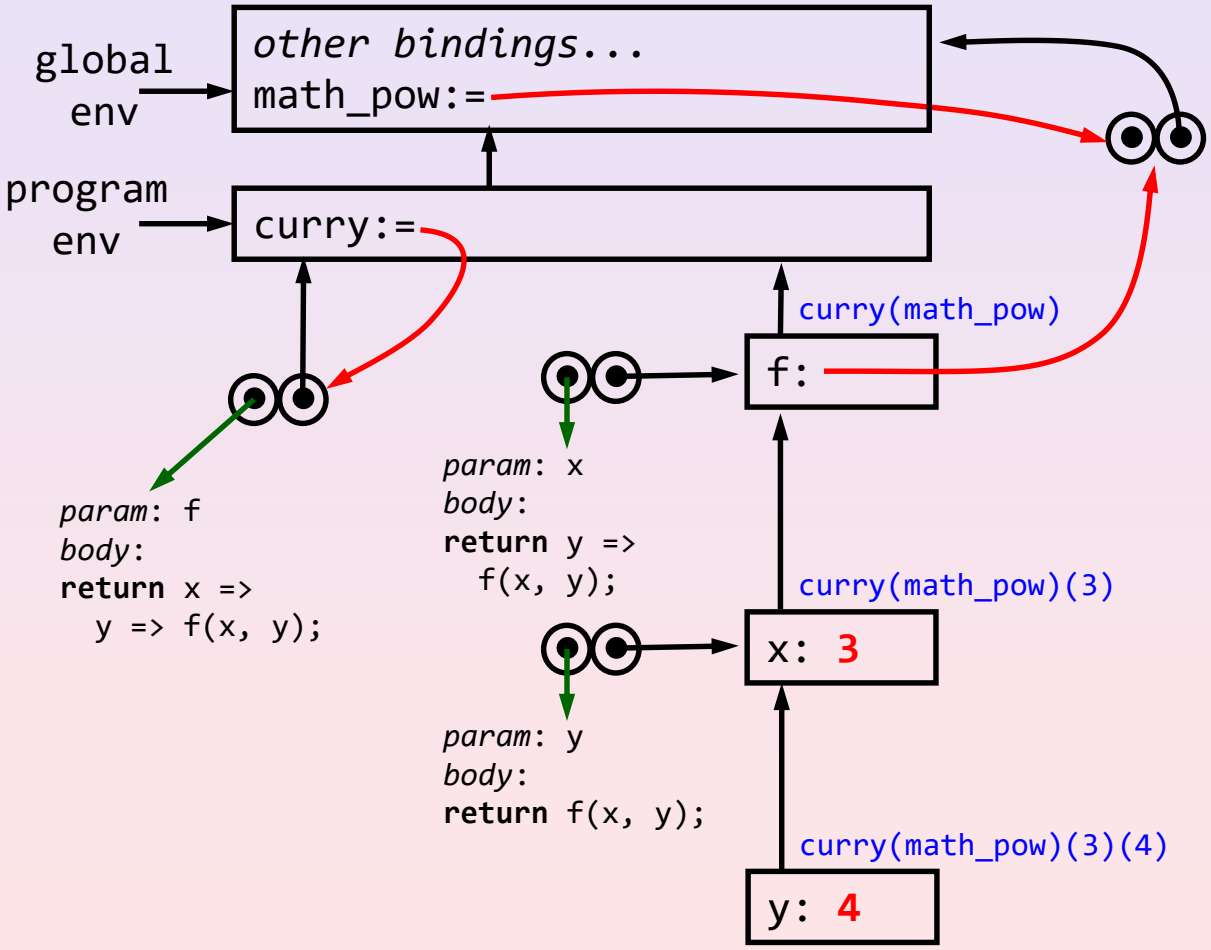

```
function curry(f) {
    return x => y => f(x, y);
}
curry(math_pow)(3)(4);
```

curry Example




```
function curry(f) {
    return x => y => f(x, y);
}
curry(math_pow)(3)(4);
```

curry Example




```
function curry(f) {
    return x => y => f(x, y);
}
curry(math_pow)(3)(4);
```

Showing all frames created during evaluation of program

evaluate `f(x, y)` ➔ `math_pow(3, 4)`

Summary

- **Assignment** allows us to create **state**
- **Substitution model** breaks down with **assignment**
- **Environment model** replaces **substitution model**
- When a **lambda expression** is evaluated, it creates a **function object** that remembers the **current environment**
- To evaluate a **function application**
  1) Create a **new frame** in the function's environment (right pointer of function object)
  2) Bind **parameters** in new frame
  3) Evaluate **body block** of function in the new environment