

Scope

Deian Stefan

(adopted from my & Edward Yang's CSE242 slides)



Substitution model

- Way of giving semantics to the λ -calculus
 - E.g., $(\lambda x.f\ x\ x)\ (\lambda y.z) \rightarrow_{\beta} f\ (\lambda y.z)\ (\lambda y.z)$
- Translate this knowledge to JavaScript functions
 - $(x \Rightarrow f(x)(x))\ (y \Rightarrow z) \rightarrow_{\beta} f(y \Rightarrow z)(y \Rightarrow z)$

Substitution model

- Why would you, in practice, not really want to do function application in this way for a language like JavaScript?
 - It's super slow! Why?
 - It's actually nonsensical sometimes! When?

Substitution gone wrong

- Consider variable mutation in JavaScript:

let y = 1;	
let z = 0;	...
z++;	→ β. 0++;
console.log(z);	...

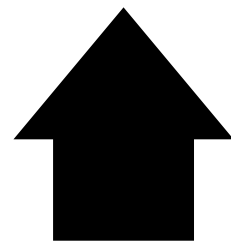
- There is nothing wrong with substitution per say
 - It's symbolic evaluation/computation
 - Problem is JavaScript has mutation and not amendable to symbolic evaluation

What can we do?

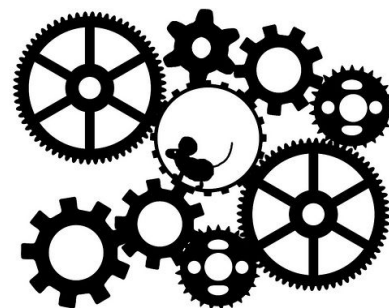
λ -calculus



environment model



machine model



The environment model (by example)

- Anatomy of a scope
- First-order functions
- Free variables
- High-order functions (bonus)

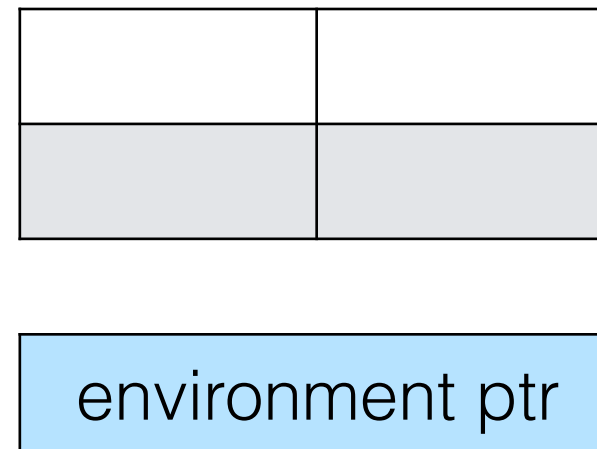
Anatomy of a scope

- What's the point of a scope (e.g., block scope)?

Anatomy of a scope

- Recall our previous example:

```
let y = 1;  
let z = 0;  
z++;  
console.log(z);
```

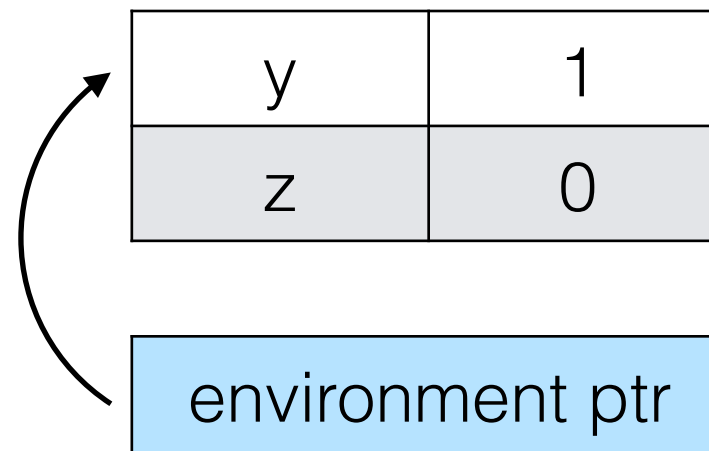


- In this model, we associate an environment (activation record) with the code we're executing
 - Environment contains entries of all variables in scope
 - Environment/stack ptr: points to cur activation record

Anatomy of a scope

- Recall our previous example:

```
let y = 1;  
let z = 0;  
z++;  
console.log(z);
```



- In this model, we associate an environment (activation record) with the code we're executing
 - Environment contains entries of all variables in scope
 - Environment/stack ptr: points to cur activation record

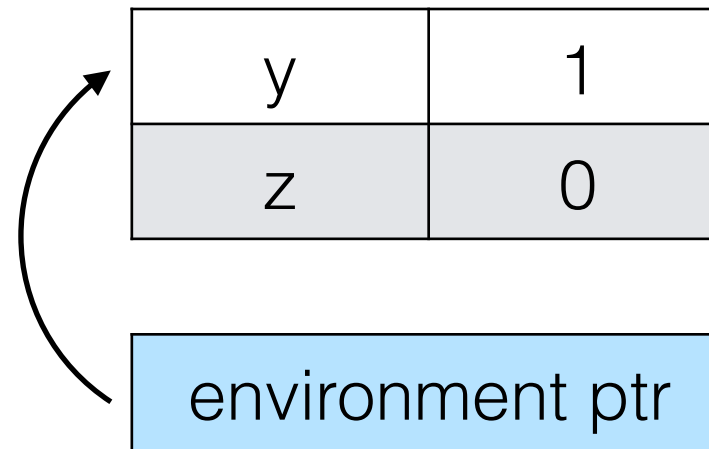
Anatomy of a scope

- In the environment model, we can distinguish between values and locations
 - r-values: plain old values; we can reason about them using substitution semantics
 - l-values: refer to locations where r-values are stored; they persist beyond single expressions.
- Why is this important?
 - It tells us the kind of values operators like ++ must take. A: r-values. **B: l-values**

Anatomy of a scope

- What's the process for executing `z++`:

```
let y = 1;  
let z = 0;  
z++;  
console.log(z);
```

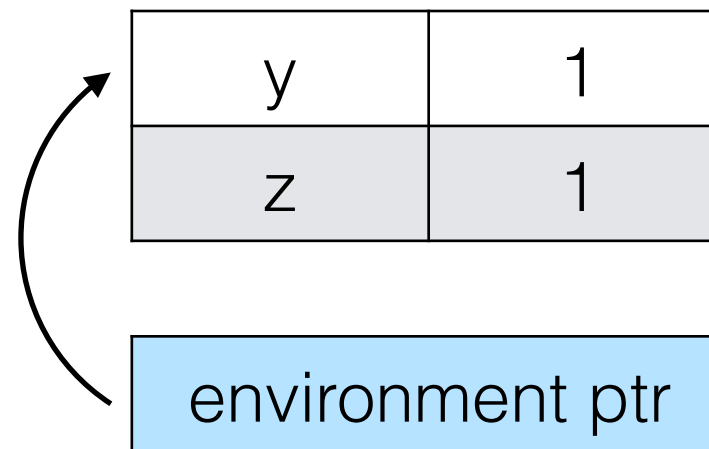


- Algorithm:
 - Find the current environment
 - Check to see if variable being reference is in env: if so, mutate!

Anatomy of a scope

- What's the process for executing `console.log(z)`

```
let y = 1;  
let z = 0;  
z++;  
console.log(z);
```



- Algorithm:
 - Find the current environment
 - Check to see if variable being reference is in env: if so, read it!

Anatomy of a scope

- This sounds slow!
 - It is!
 - But remember: this is not the machine model, this is still an abstract model!
- Not too far off from machine model
 - In x86, you dereference %esp to figure out where stack is and use offset to that location
 - In JavaScript, you often do table lookup to find location of variables

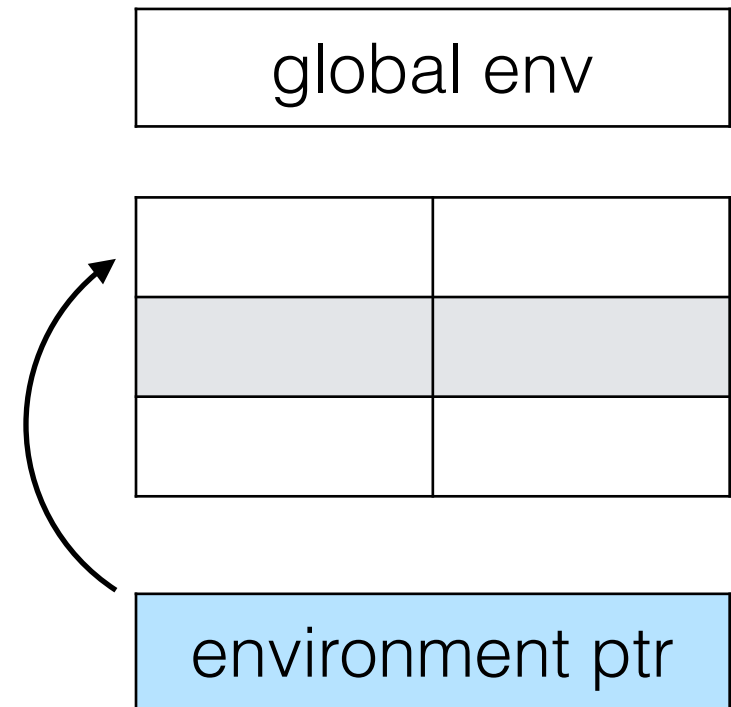
The environment model (by example)

- Anatomy of a scope ✓
- First-order functions
- Free variables
- High-order functions (bonus)

First-order functions

- Consider activation record when calling function:

```
function fact(n) {  
  if (n <= 1) {  
    return 1;  
  } else {  
    return n * fact(n-1);  
  }  
}  
fact(3);
```

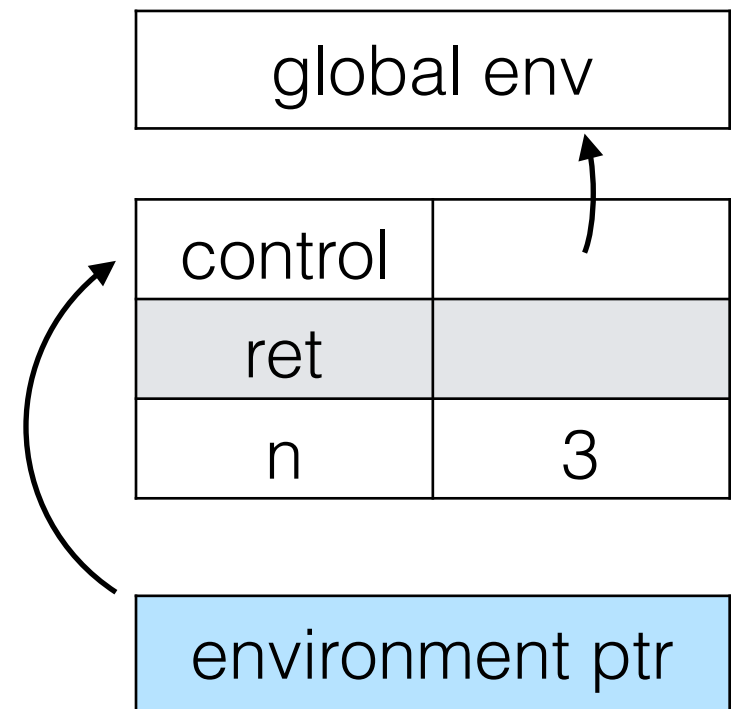


- What else do we need to keep track of?

First-order functions

- Consider activation record when calling function:

```
function fact(n) {  
  if (n <= 1) {  
    return 1;  
  } else {  
    return n * fact(n-1);  
  }  
}  
fact(3);
```



- What else do we need to keep track of?

More bookkeeping

- The parts of an activation record when calling function
 - control link: records where to switch the environment pointer to when we finish evaluating in this scope.
 - Do we need this for block scopes too? A: yes, B:no
 - return value: l-value where the return value of function should be stored
 - parameters: l-value for each formal parameter
 - local variables: l-values for each let+const declaration

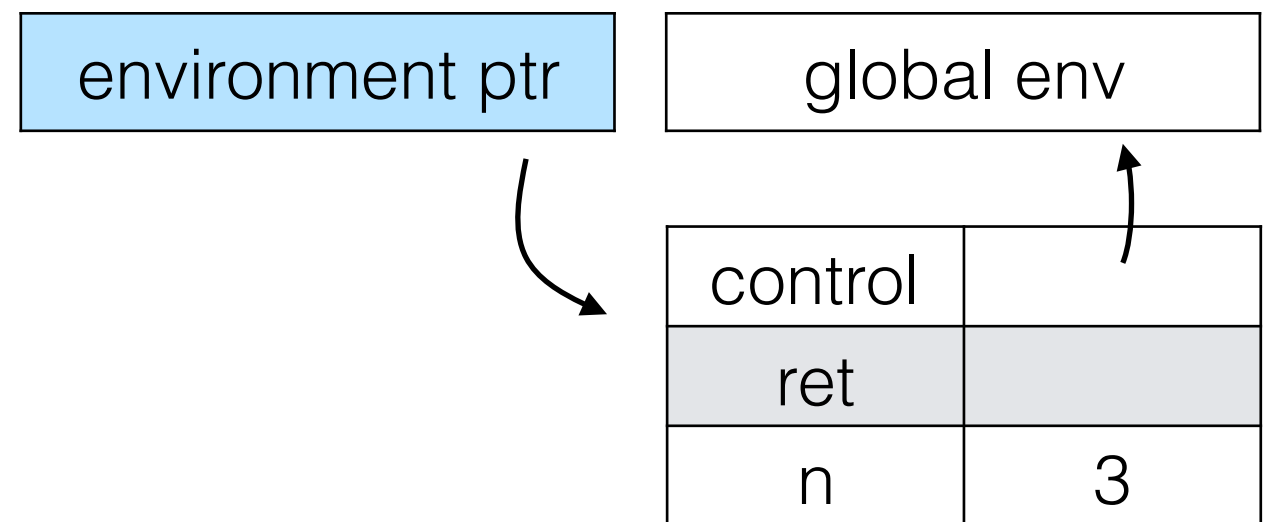
More bookkeeping

- Anything else?
 - Yes! Typically activation records will store the return address where to resume code execution — we'll talk about this in the control flow lecture

Let's look at how evaluation works

- Consider activation records when calling function:

```
function fact(n) {  
  if (n <= 1) {  
    return 1;  
  } else {  
    return n * fact(n-1);  
  }  
}  
fact(3);
```

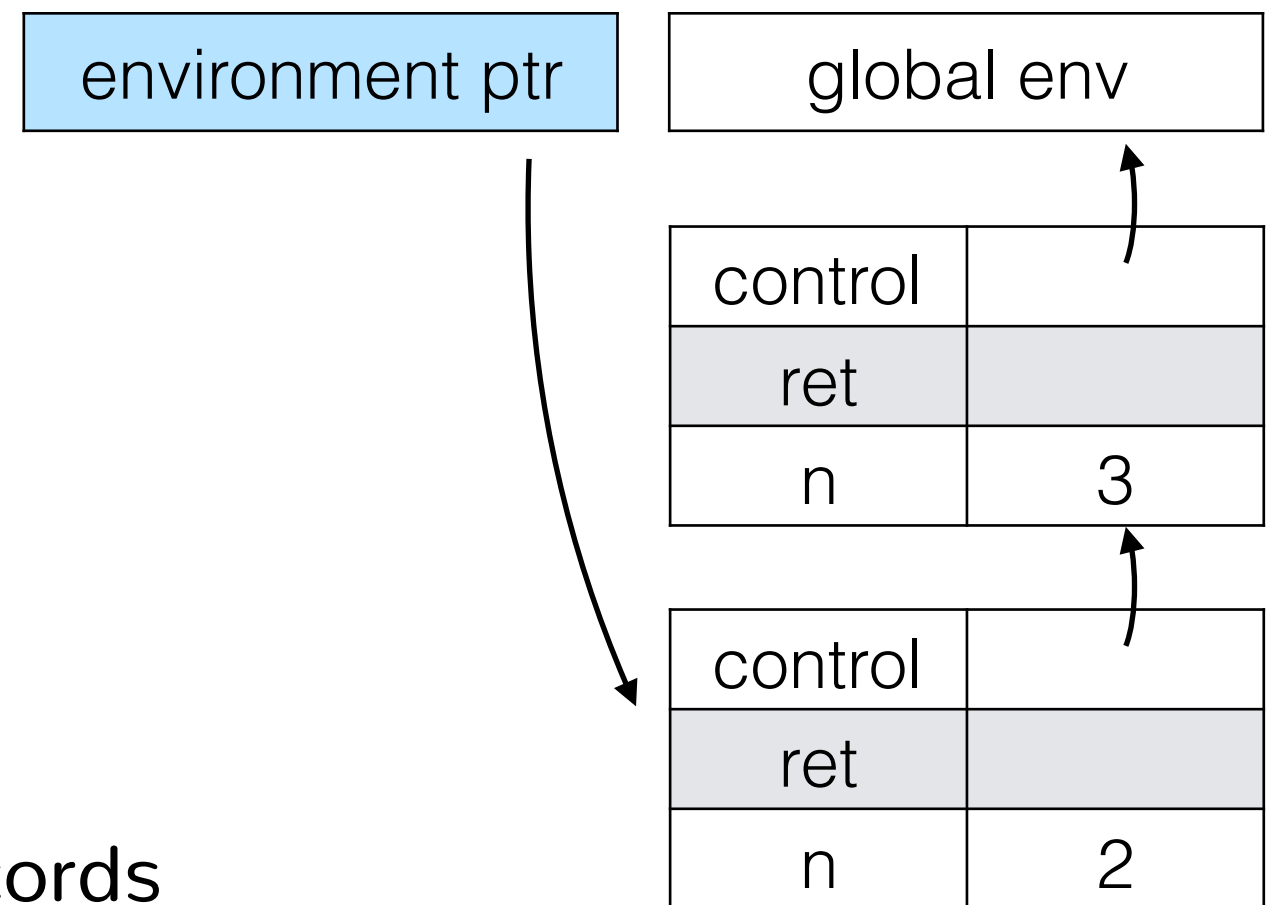


- Do we keep the activation records on the stack after evaluation?
A: yes, B: no

Let's look at how evaluation works

- Consider activation records when calling function:

```
function fact(n) {  
  if (n <= 1) {  
    return 1;  
  } else {  
    return n * fact(n-1);  
  }  
}  
fact(3);
```

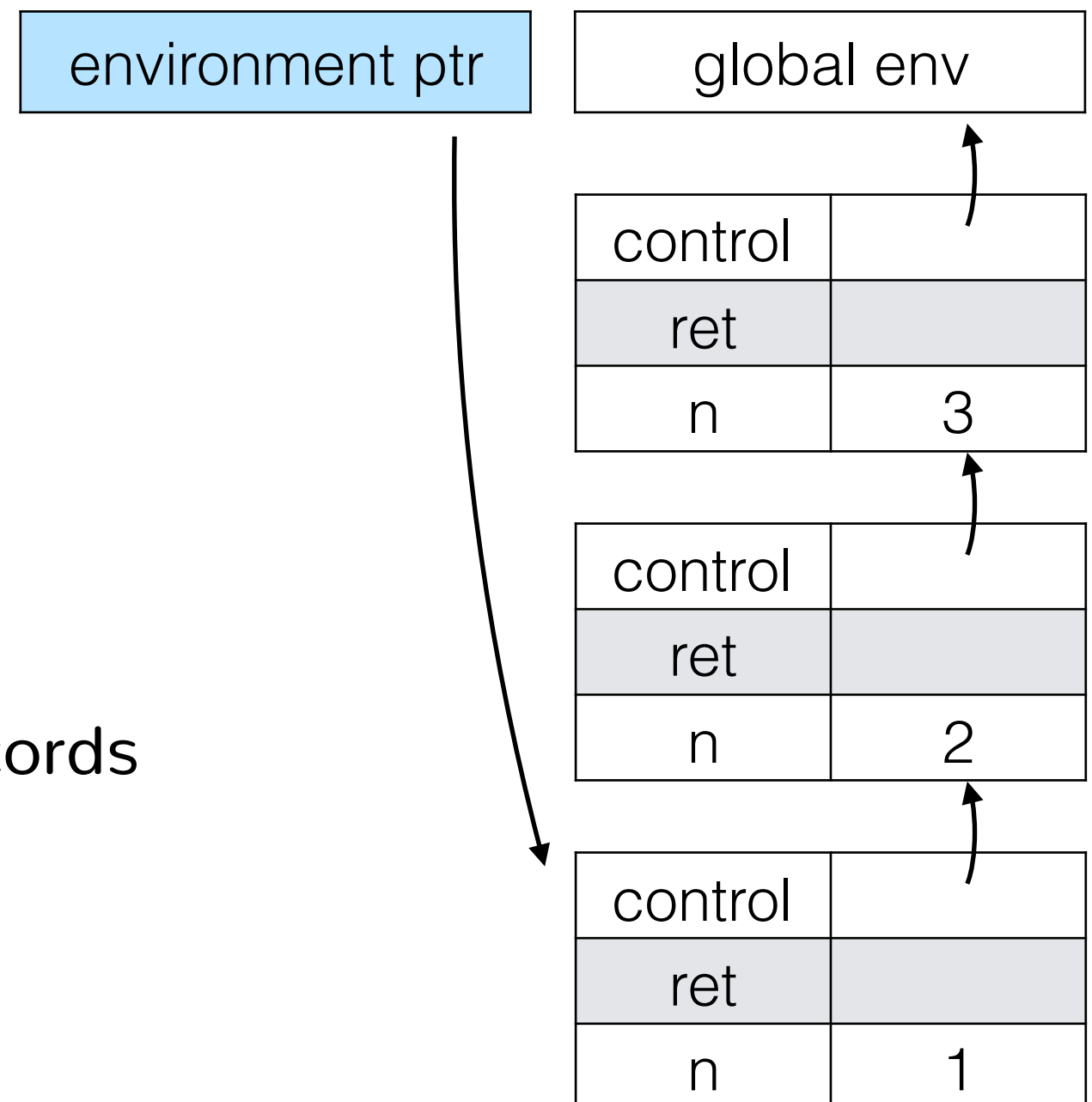


- Do we keep the activation records on the stack after evaluation?
A: yes, B: no

Let's look at how evaluation works

- Consider activation records when calling function:

```
function fact(n) {  
  if (n <= 1) {  
    return 1;  
  } else {  
    return n * fact(n-1);  
  }  
}  
fact(3);
```

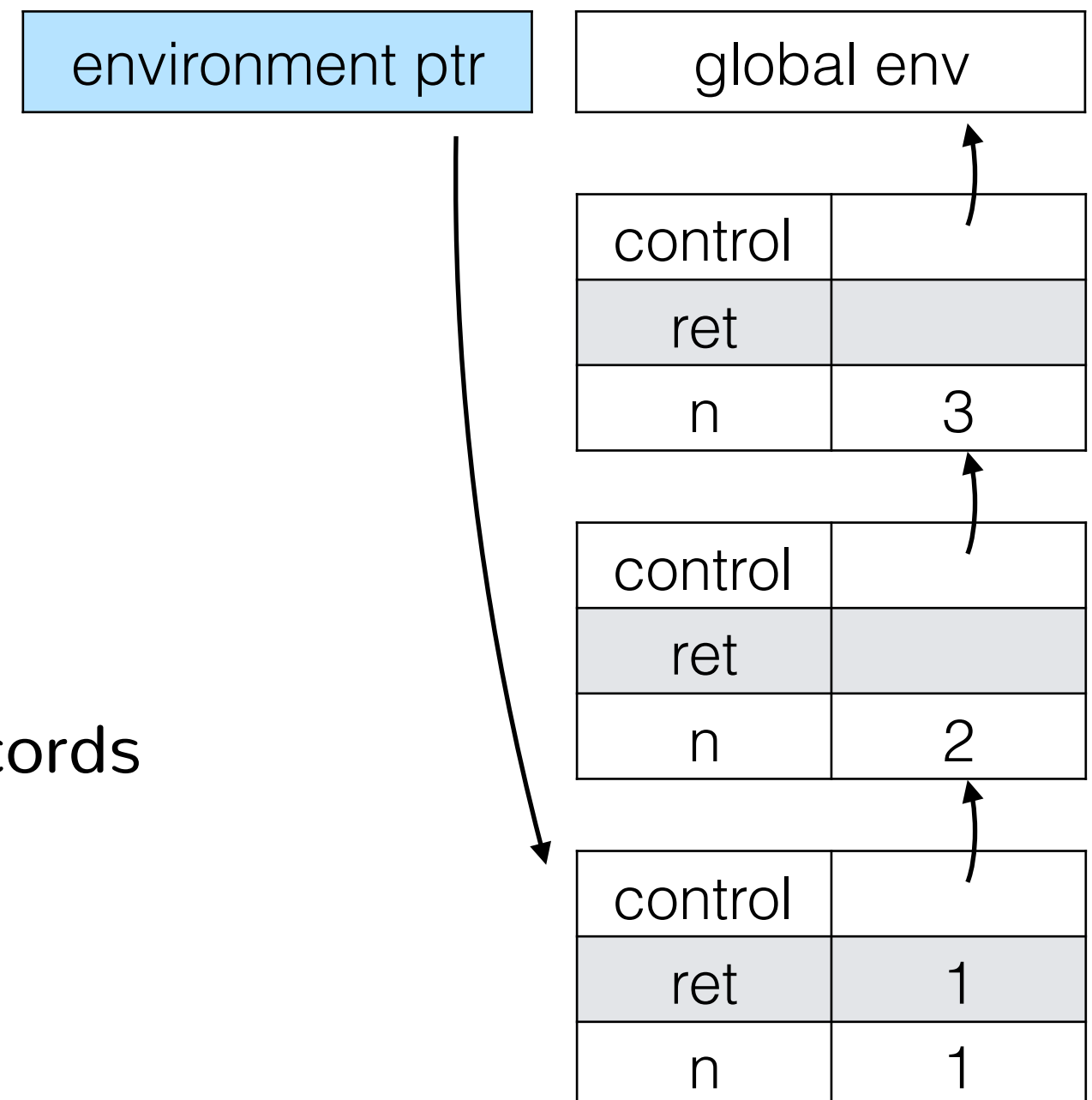


- Do we keep the activation records on the stack after evaluation?
A: yes, B: no

Let's look at how evaluation works

- Consider activation records when calling function:

```
function fact(n) {  
  if (n <= 1) {  
    return 1;  
  } else {  
    return n * fact(n-1);  
  }  
}  
fact(3);
```

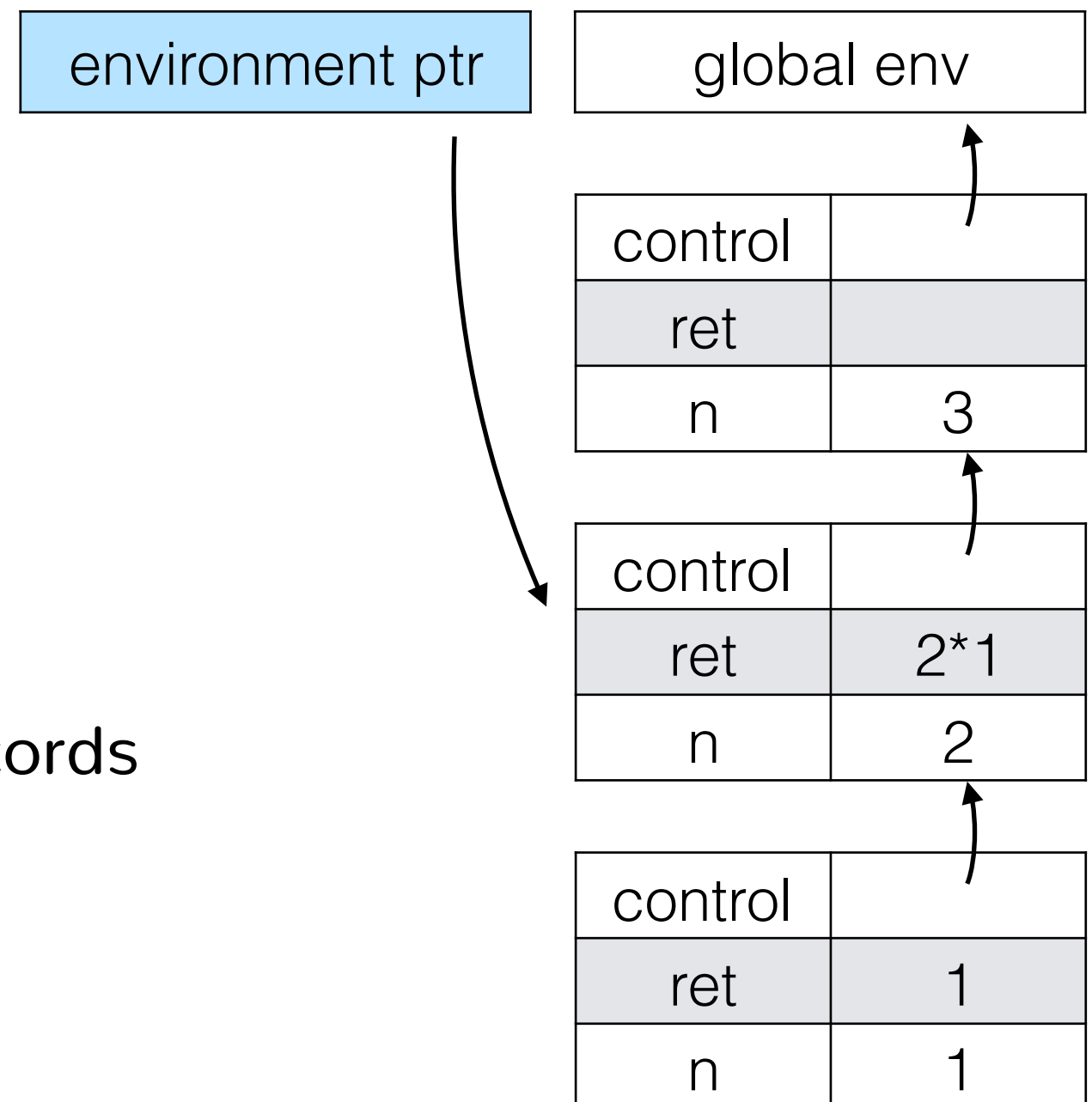


- Do we keep the activation records on the stack after evaluation?
A: yes, B: no

Let's look at how evaluation works

- Consider activation records when calling function:

```
function fact(n) {  
  if (n <= 1) {  
    return 1;  
  } else {  
    return n * fact(n-1);  
  }  
}  
fact(3);
```

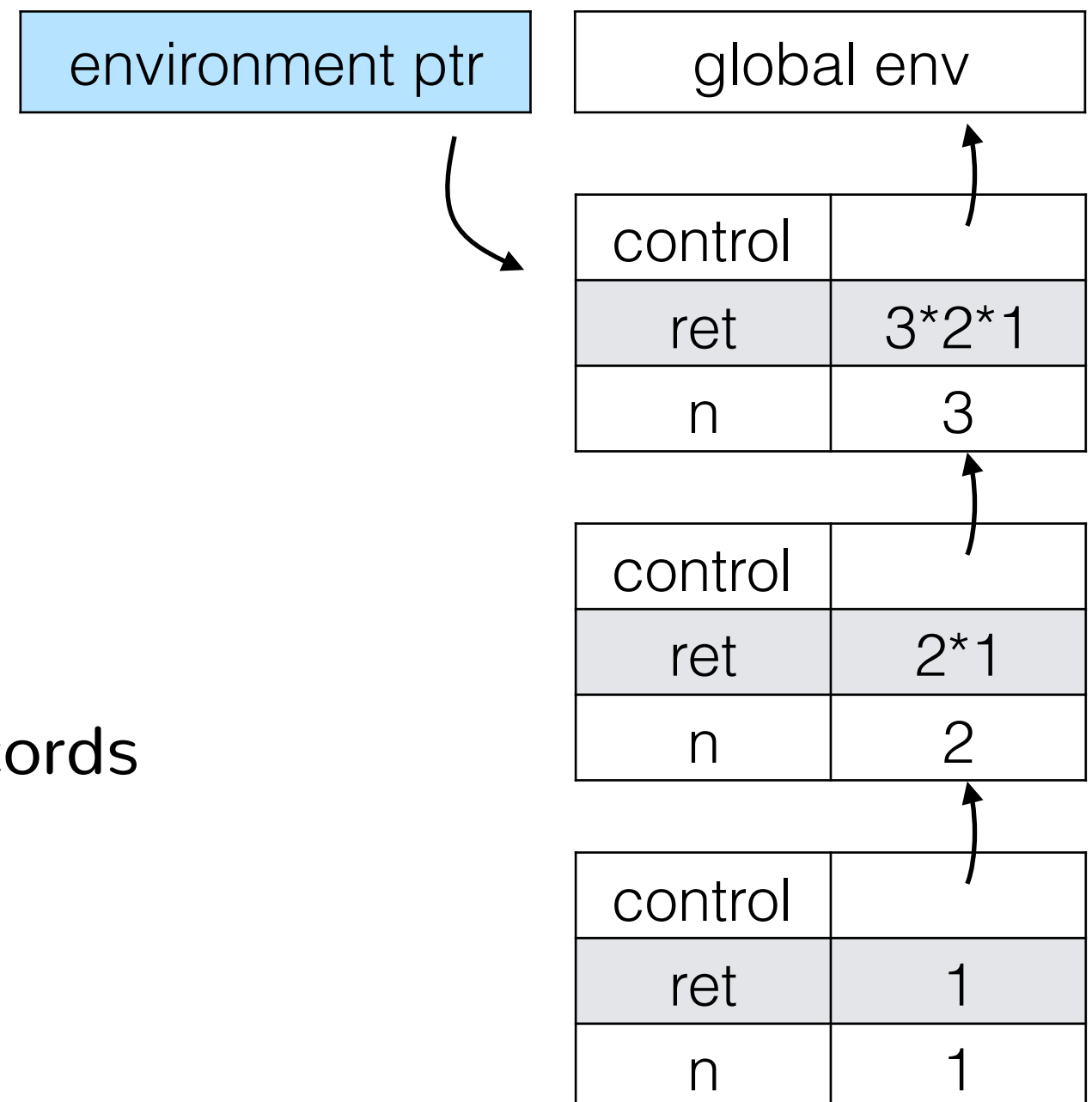


- Do we keep the activation records on the stack after evaluation?
A: yes, B: no

Let's look at how evaluation works

- Consider activation records when calling function:

```
function fact(n) {  
  if (n <= 1) {  
    return 1;  
  } else {  
    return n * fact(n-1);  
  }  
}  
fact(3);
```



- Do we keep the activation records on the stack after evaluation?

A: yes, **B: no**

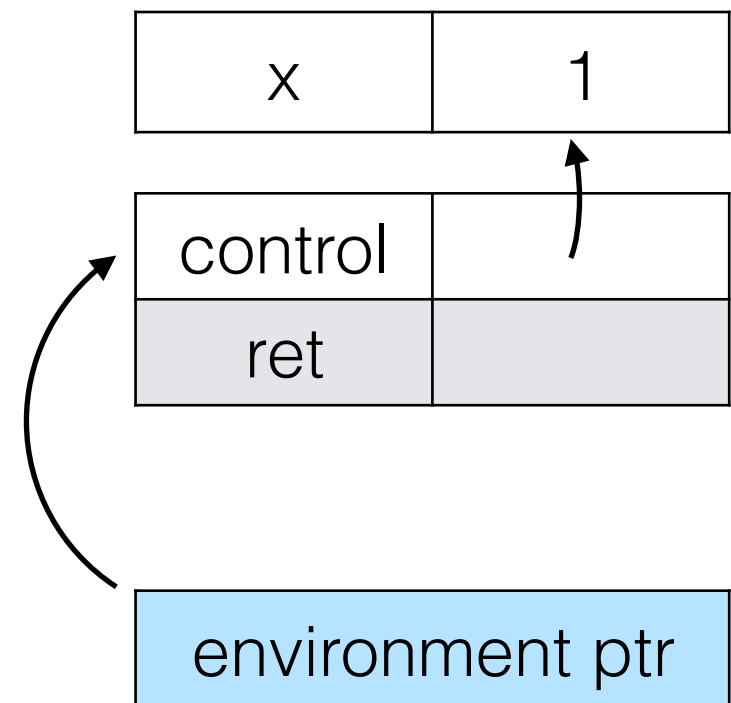
The environment model (by example)

- Anatomy of a scope ✓
- First-order functions ✓
- Free variables
- High-order functions (bonus)

Free variables

- Consider activation records when calling f:

```
let x = 1;  
function f() {  
  console.log(x)  
}  
f();
```



- Should we lookup x via the control link?
 - A: yes
 - B: no

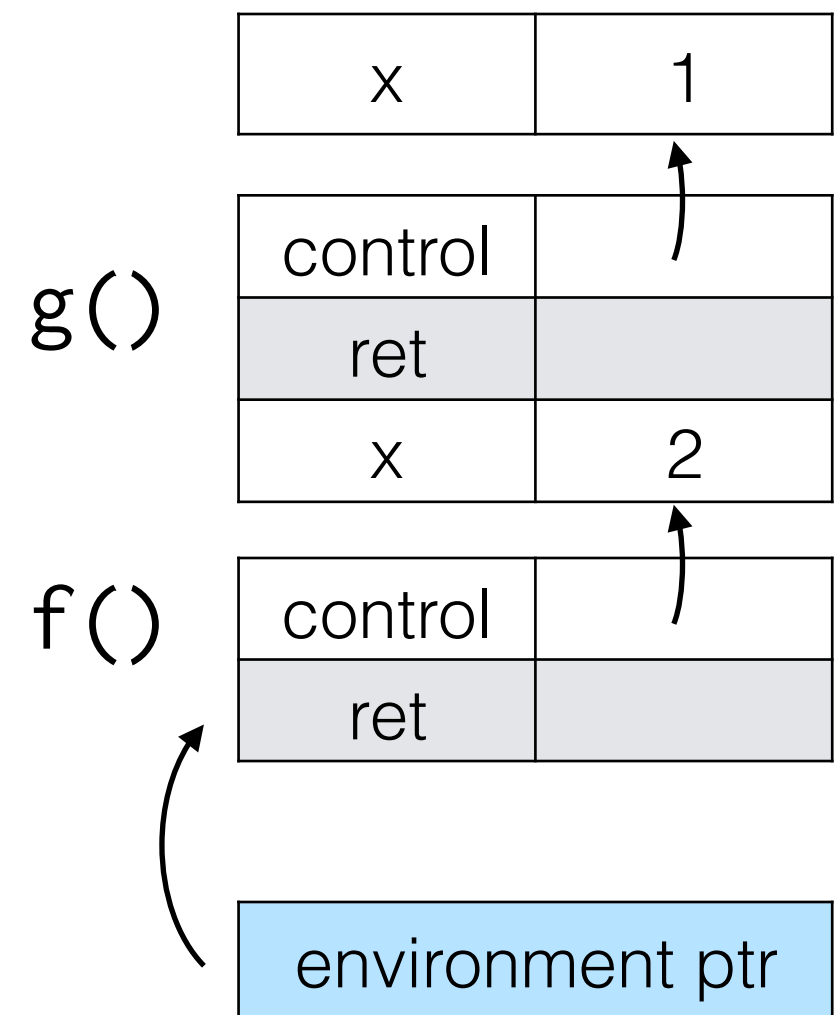
Free variables

- Consider activation records when calling g:

```
let x = 1;  
function f() {  
  console.log(x)  
}
```

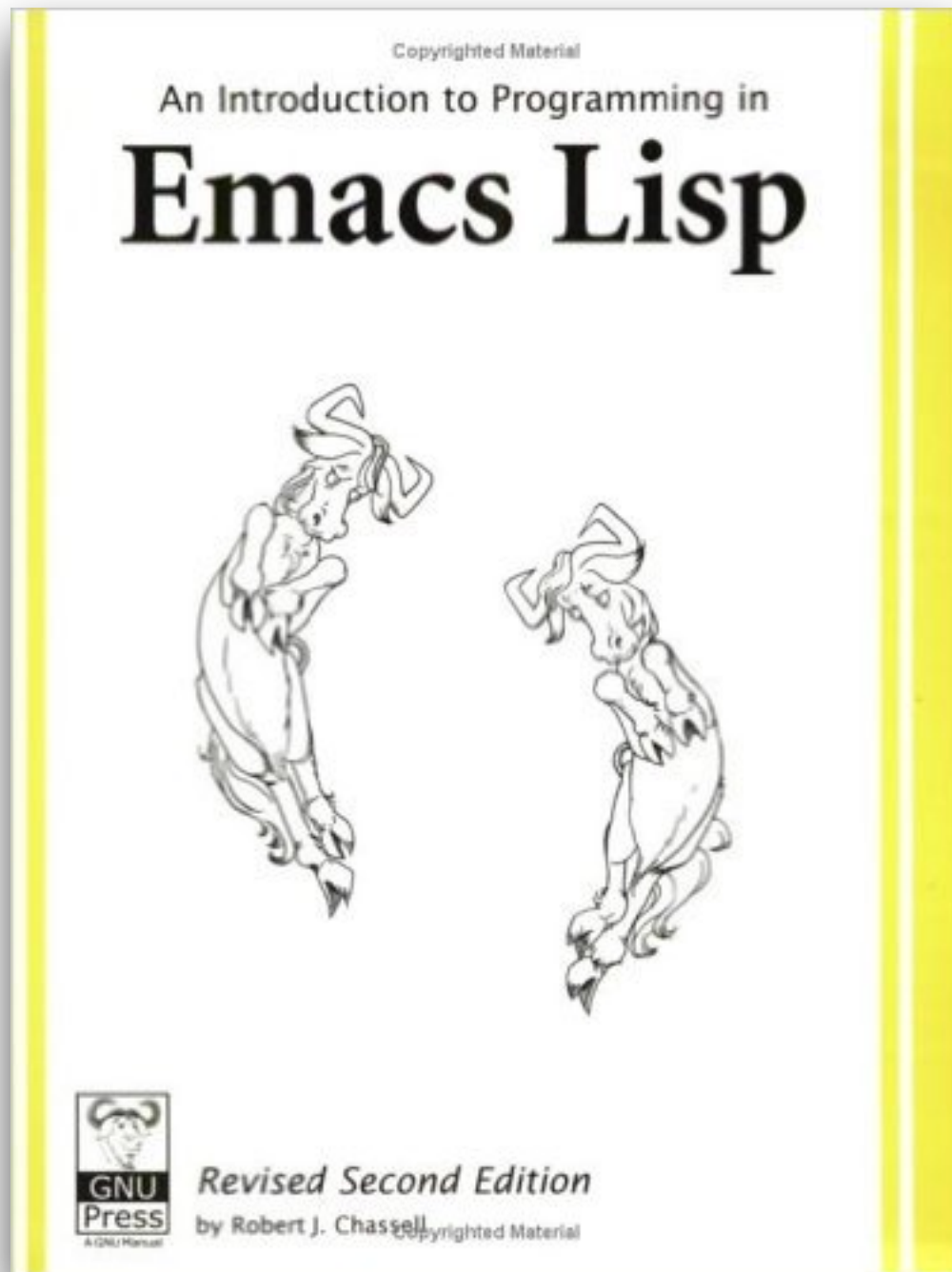
```
function g() {  
  let x = 1;  
  f();  
}
```

```
g();
```



- What happens when we follow the control link?

Congrats, you did it!



You invented dynamic scoping!

How do we “fix” this?

- We need more bookkeeping!
 - access link: reference to activation record of closest enclosing lexical scope
- Modify our lookup algorithm:
 - Find the current environment
 - Check to see if variable being reference is in env
 - If not, follow the access link and repeat

Retry with access links

- Consider activation records when calling g:

```
let x = 1;  
function f() {  
  console.log(x)  
}
```

```
function g() {  
  let x = 2;  
  f();  
}
```

```
g();
```

x	1
---	---

environment ptr



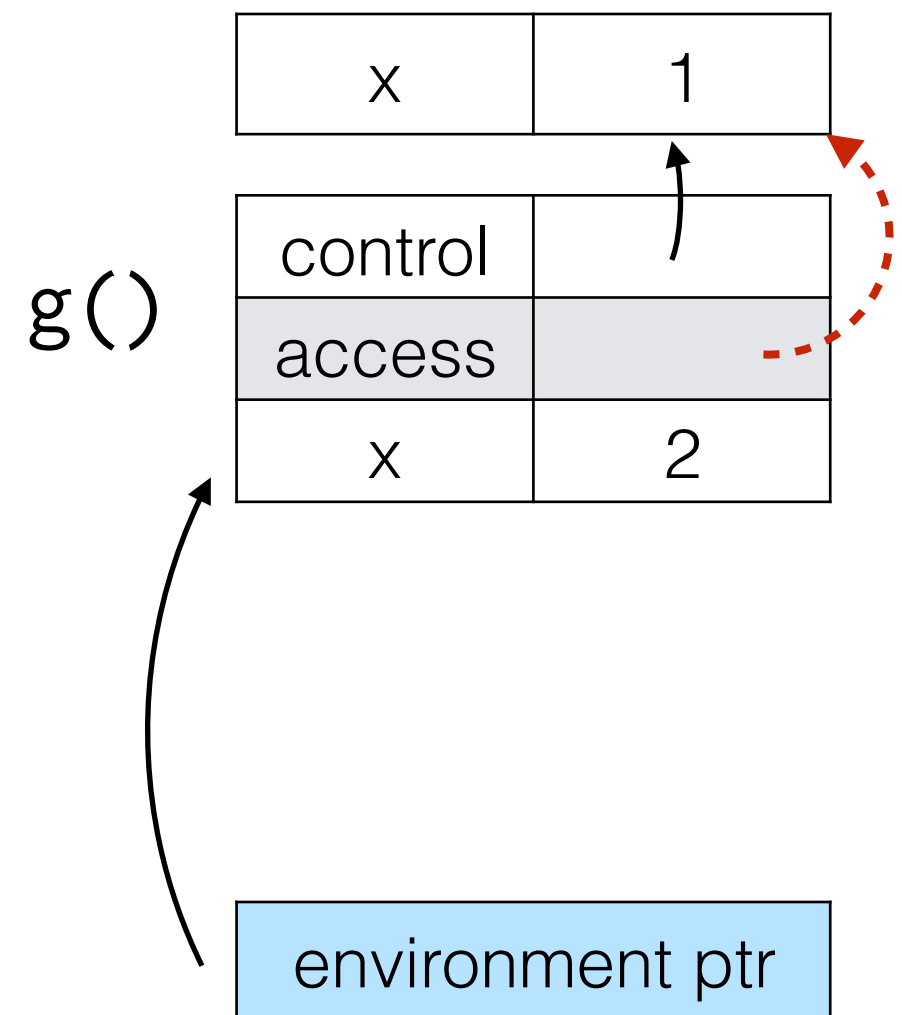
Retry with access links

- Consider activation records when calling g:

```
let x = 1;  
function f() {  
  console.log(x)  
}
```

```
function g() {  
  let x = 2;  
  f();  
}
```

```
g();
```



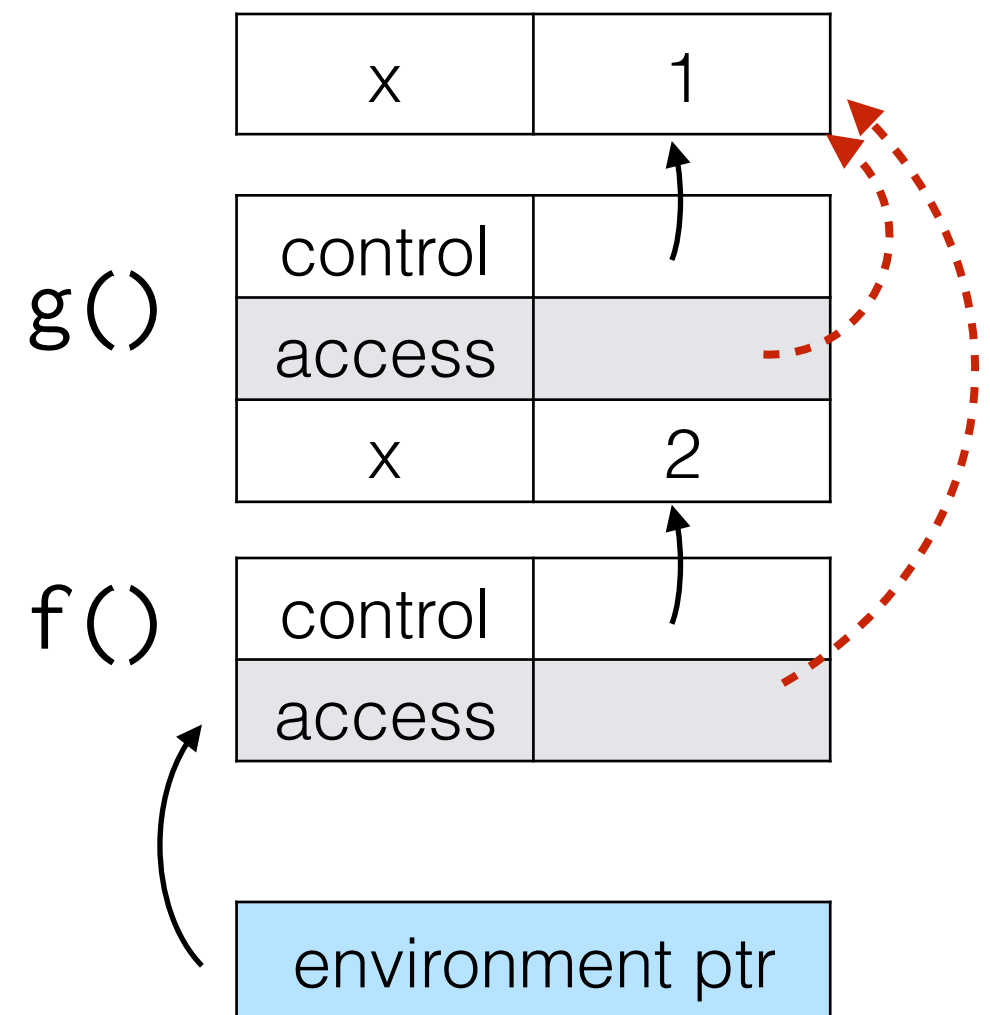
Retry with access links

- Consider activation records when calling g:

```
let x = 1;  
function f() {  
  console.log(x)  
}
```

```
function g() {  
  let x = 2;  
  f();  
}
```

```
g();
```



Functions are data!

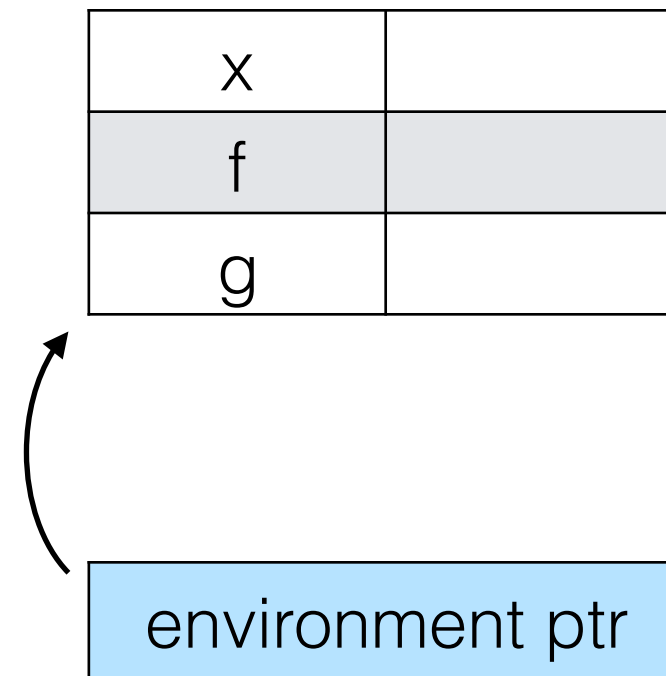
The act of defining a function should include the act of recording the access link associated with the function

Treating functions as data

- Let's look at the example again, with minor rewrite

```
let x = 1;  
let f = () => {  
  console.log(x)  
}
```

```
let g = () => {  
  let x = 2;  
  f();  
}  
  
g();
```



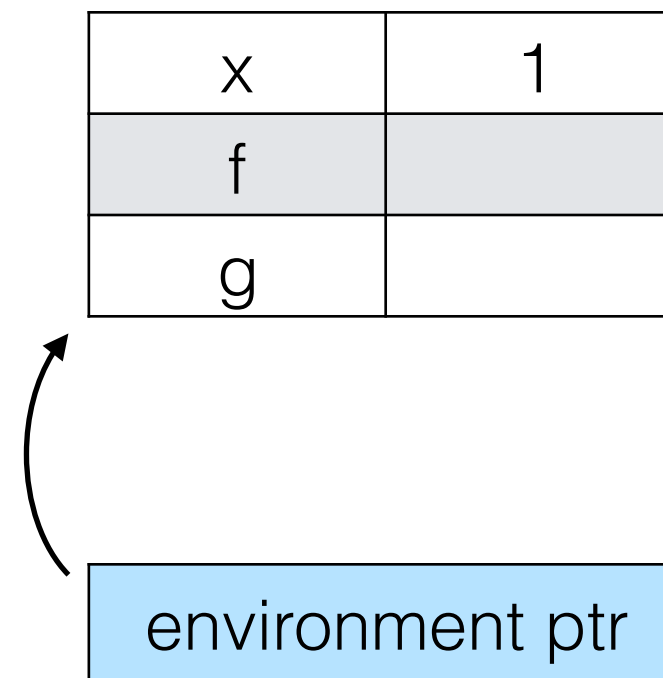
- Function as data = closures = (current env ptr, code pointer)

Treating functions as data

- Let's look at the example again, with minor rewrite

```
let x = 1;  
let f = () => {  
  console.log(x)  
}
```

```
let g = () => {  
  let x = 2;  
  f();  
}  
  
g();
```



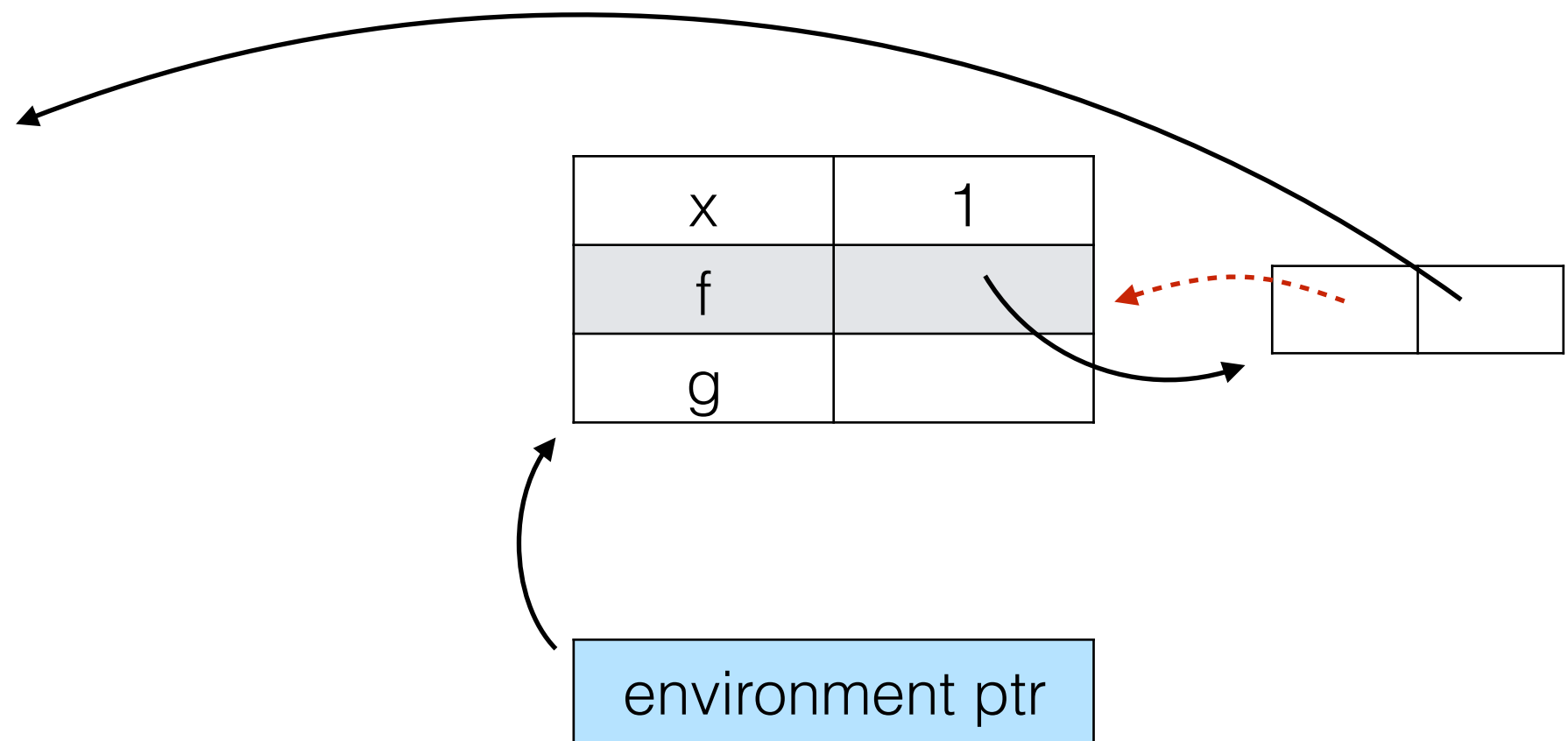
- Function as data = closures = (current env ptr, code pointer)

Treating functions as data

- Let's look at the example again, with minor rewrite

```
let x = 1;  
let f = () => {  
  console.log(x)  
}
```

```
let g = () => {  
  let x = 2;  
  f();  
}  
  
g();
```



- Function as data = closures = (current env ptr, code pointer)

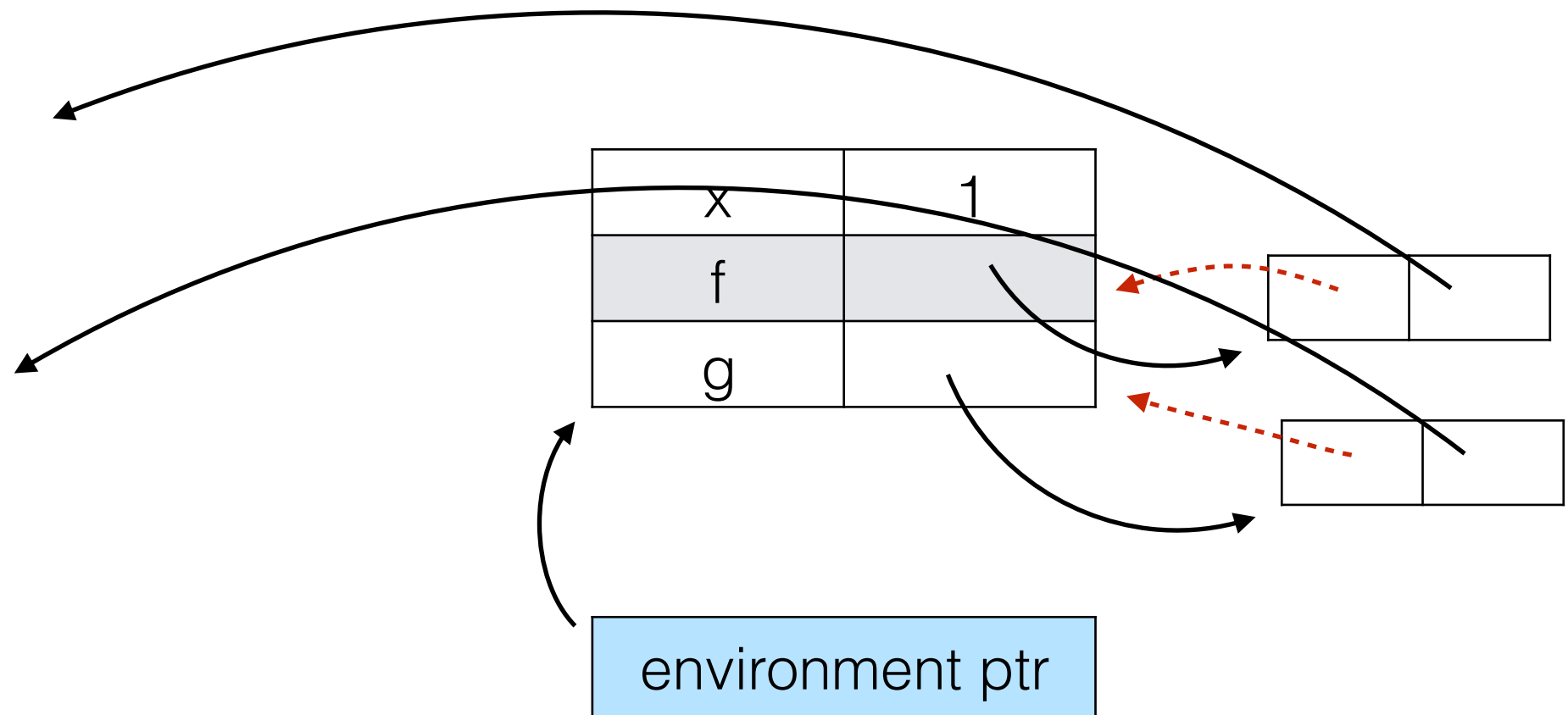
Treating functions as data

- Let's look at the example again, with minor rewrite

```
let x = 1;
let f = () => {
  console.log(x)
}
```

```
let g = () => {  
  let x = 2;  
  f();  
}
```

g();



- Function as data = closures = (current env ptr, code pointer)

Treating functions as data

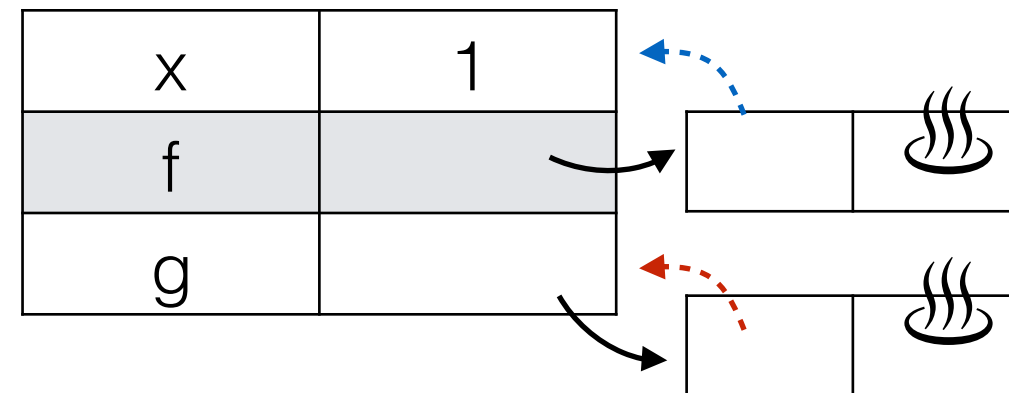
- When we evaluate function, the access link is set to the pointer in the closure

```
let x = 1;  
let f = () => {  
  console.log(x)  
}
```

```
let g = () => {  
  let x = 2;  
  f();  
}
```

```
g();
```

environment ptr



Treating functions as data

- When we evaluate function, the access link is set to the pointer in the closure

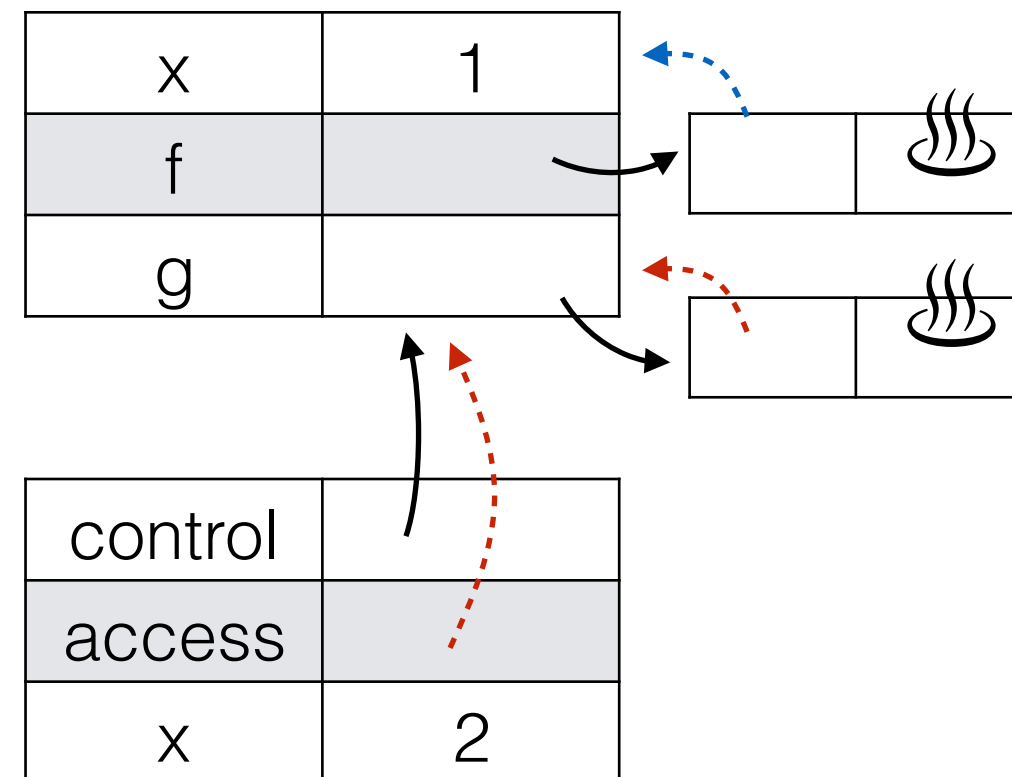
```
let x = 1;  
let f = () => {  
  console.log(x)  
}
```

```
let g = () => {  
  let x = 2;  
  f();  
}
```

```
g();
```

environment ptr

g()



Treating functions as data

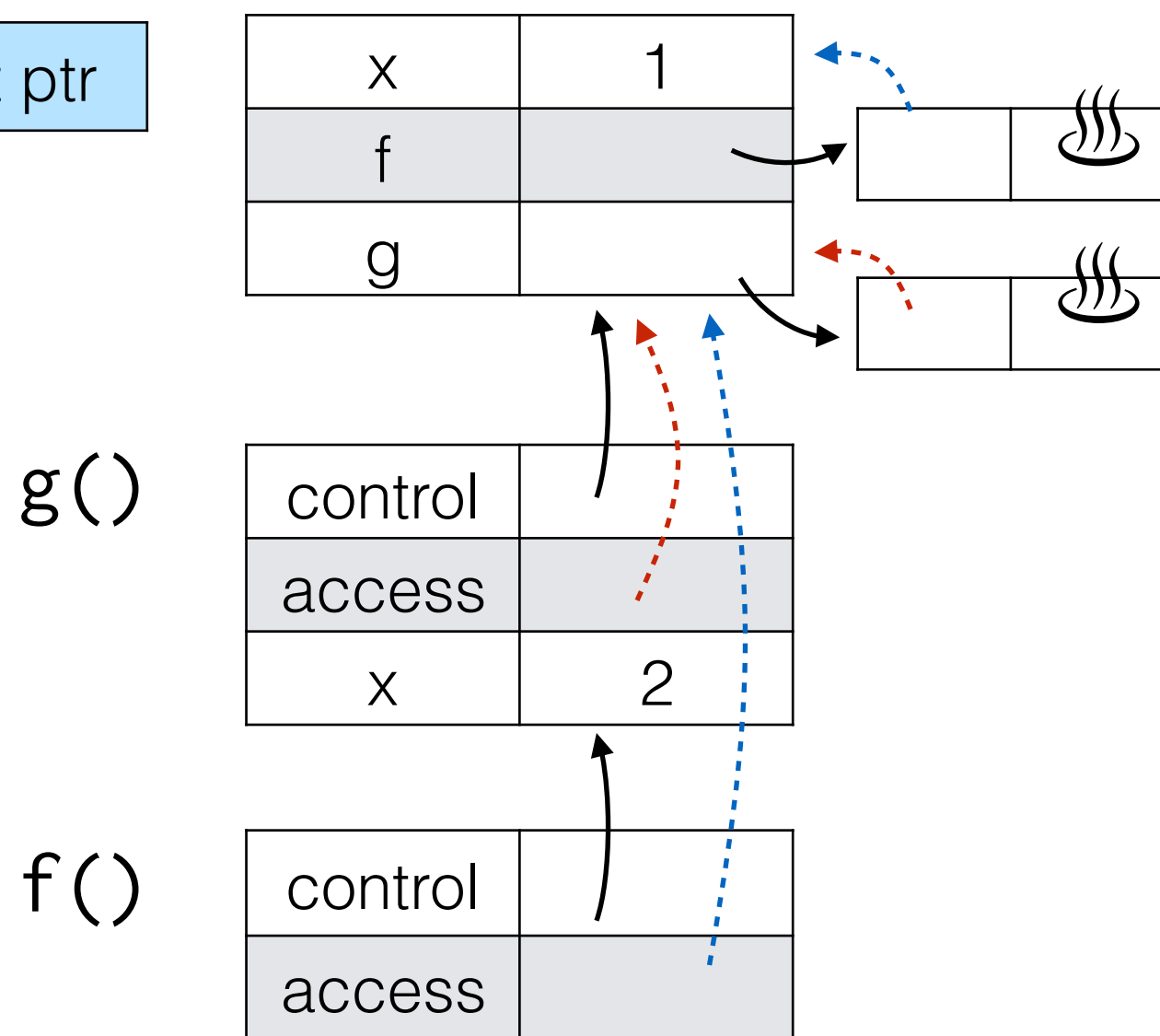
- When we evaluate function, the access link is set to the pointer in the closure

```
let x = 1;  
let f = () => {  
  console.log(x)  
}
```

```
let g = () => {  
  let x = 2;  
  f();  
}
```

```
g();
```

environment ptr



Treating functions as data

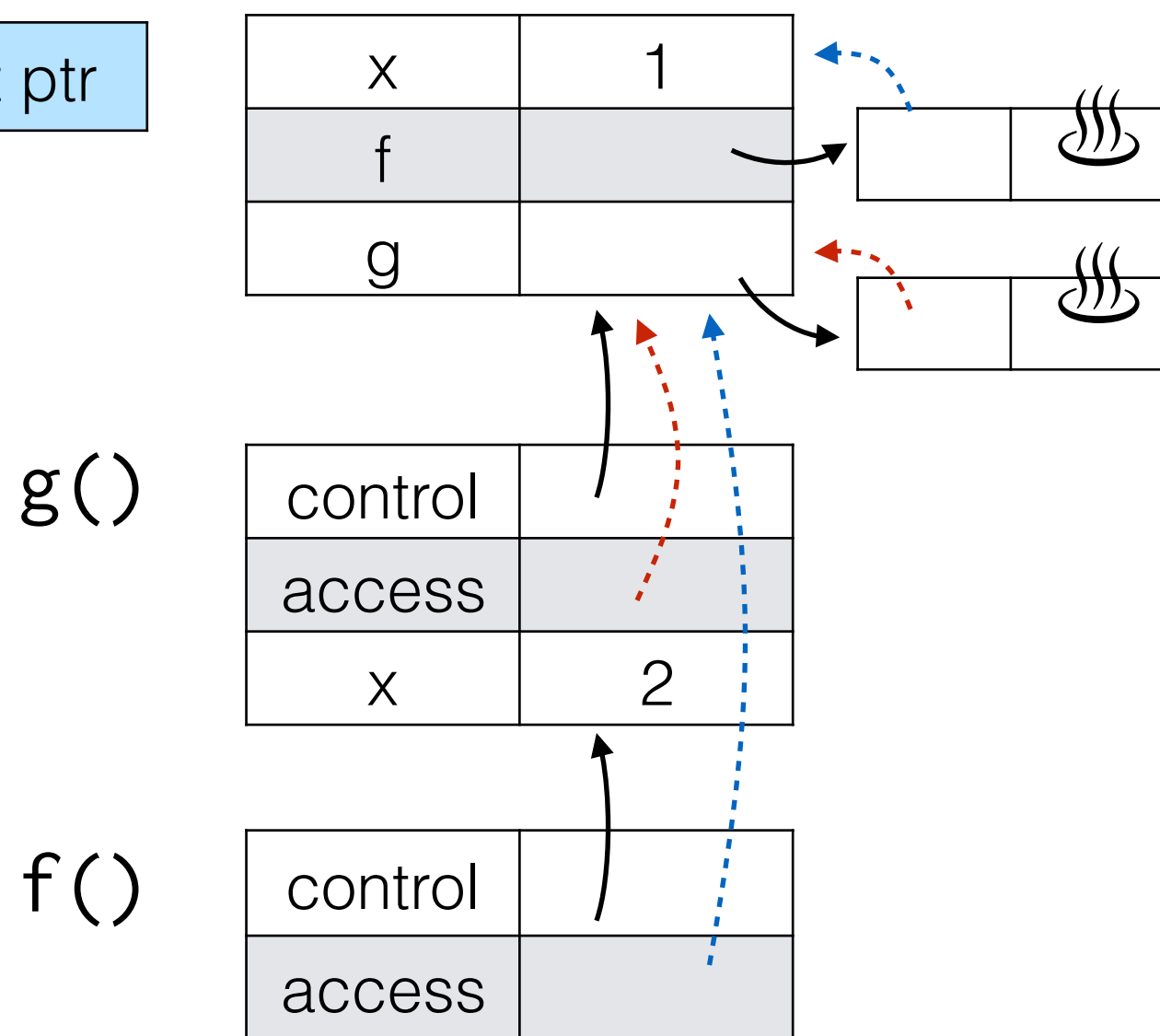
- When we evaluate function, the access link is set to the pointer in the closure

```
let x = 1;  
let f = () => {  
  console.log(x) // 1  
}
```

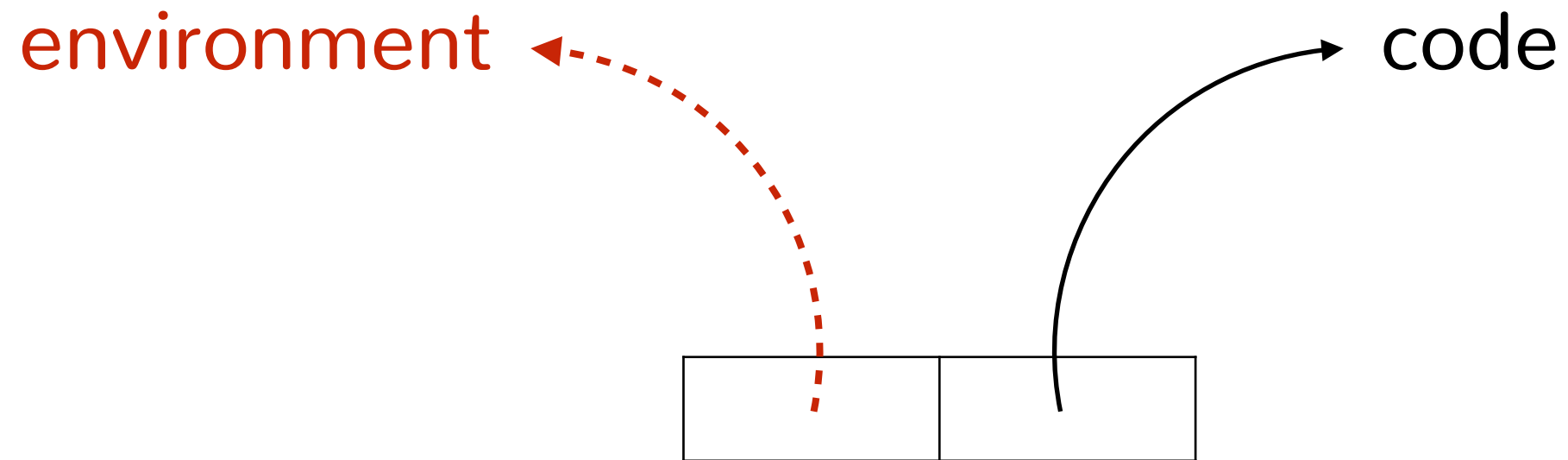
```
let g = () => {  
  let x = 2;  
  f();  
}
```

```
g();
```

environment ptr



Closures



The environment model (by example)

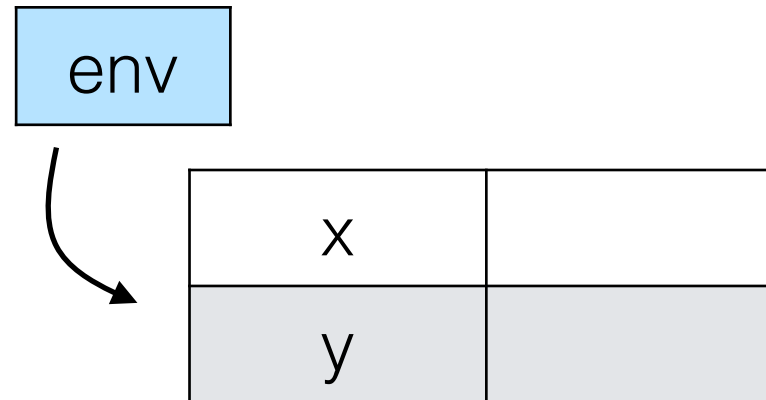
- Anatomy of a scope ✓
- First-order functions ✓
- Free variables ✓
- High-order functions (bonus)

Higher-order functions

- Consider the use of high-order mkCounter function

```
function mkCounter(c) {  
  return () => {  
    return c++;  
  };  
}
```

```
let x = mkCounter(0);  
let y = mkCounter(2);  
console.log(x());
```

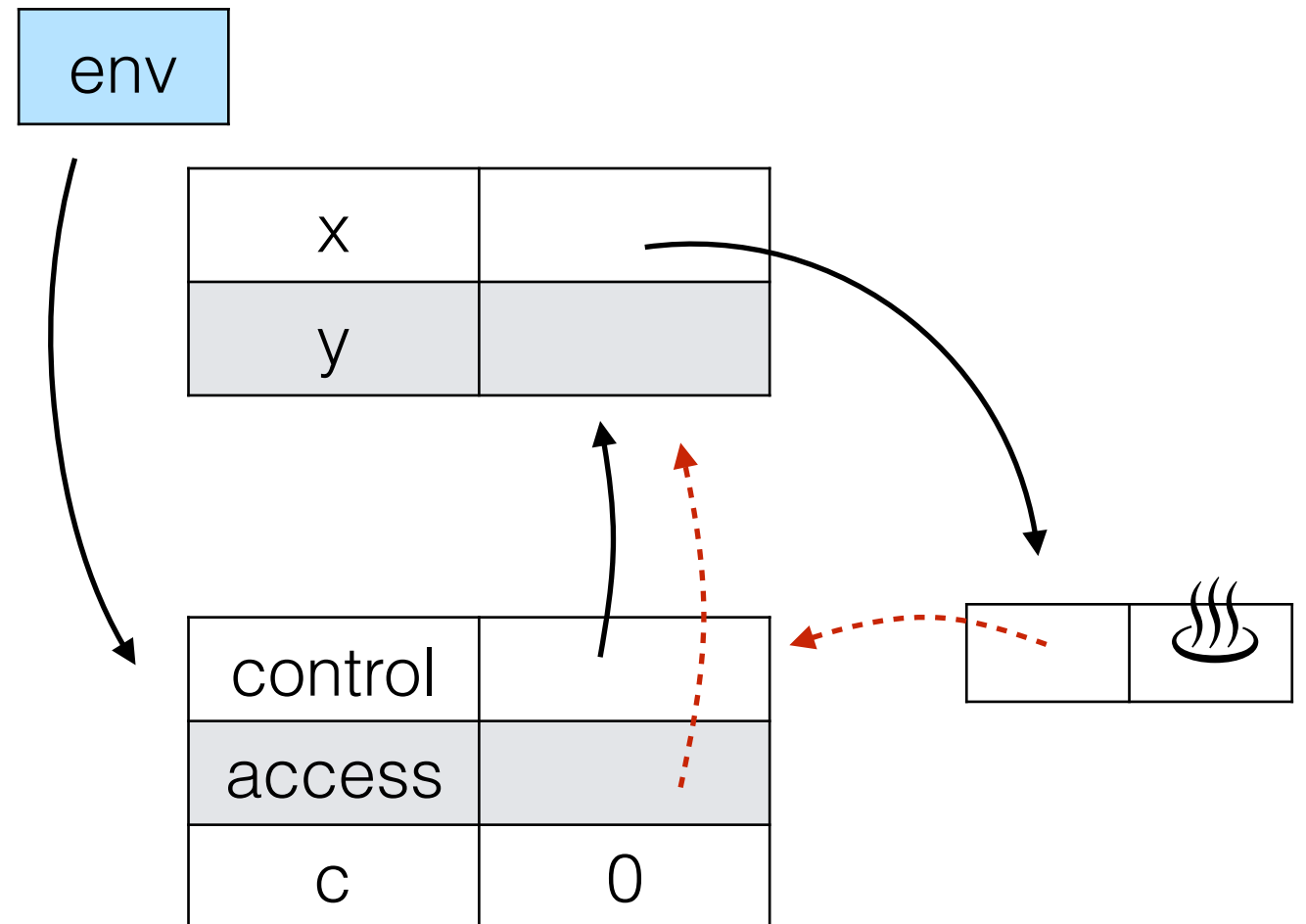


Higher-order functions

- Consider the use of high-order mkCounter function

```
function mkCounter(c) {  
  return () => {  
    return c++;  
  };  
}
```

```
let x = mkCounter(0);  
let y = mkCounter(2);  
console.log(x());
```

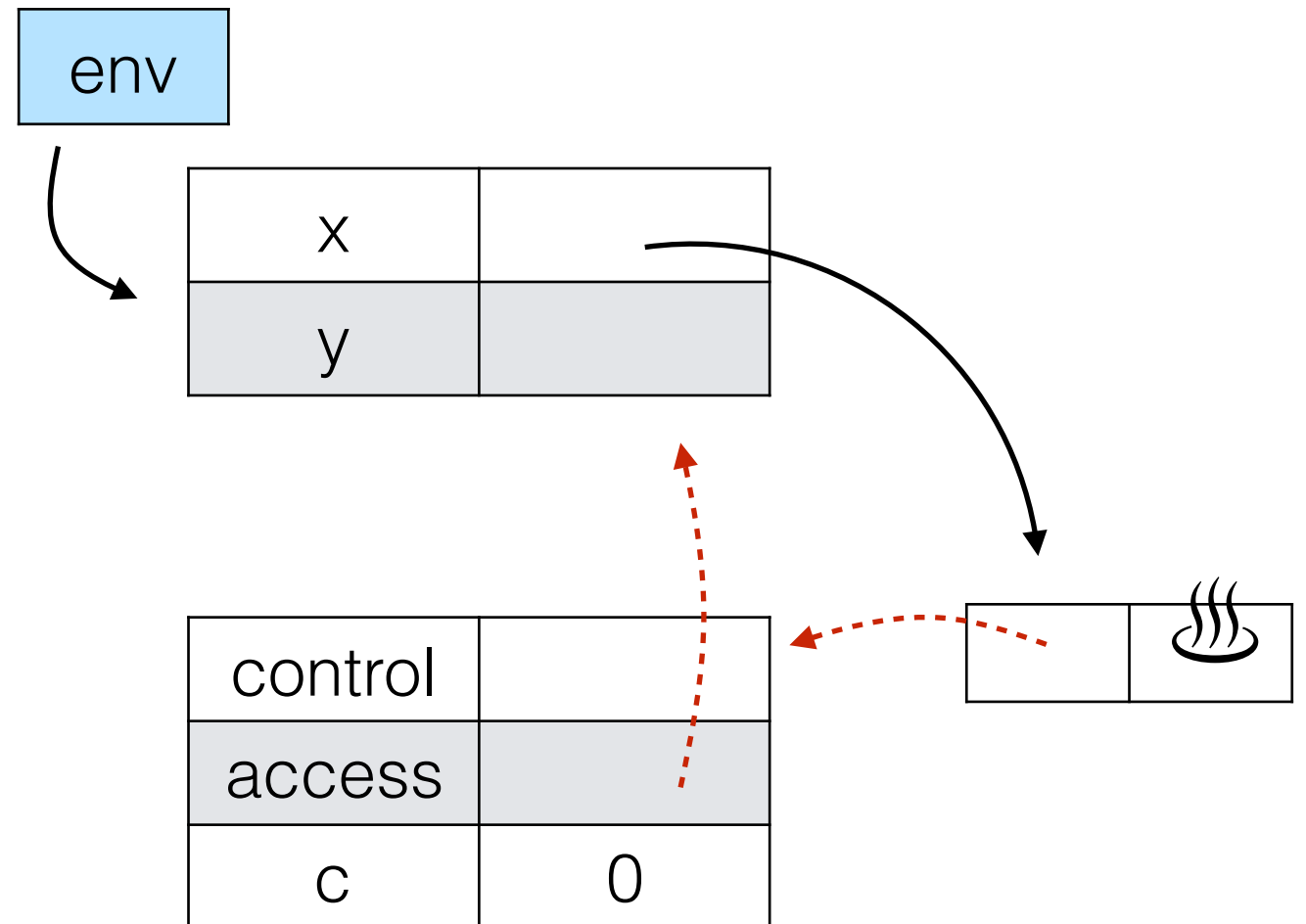


Higher-order functions

- Consider the use of high-order mkCounter function

```
function mkCounter(c) {  
  return () => {  
    return c++;  
  };  
}
```

```
let x = mkCounter(0);  
let y = mkCounter(2);  
console.log(x());
```

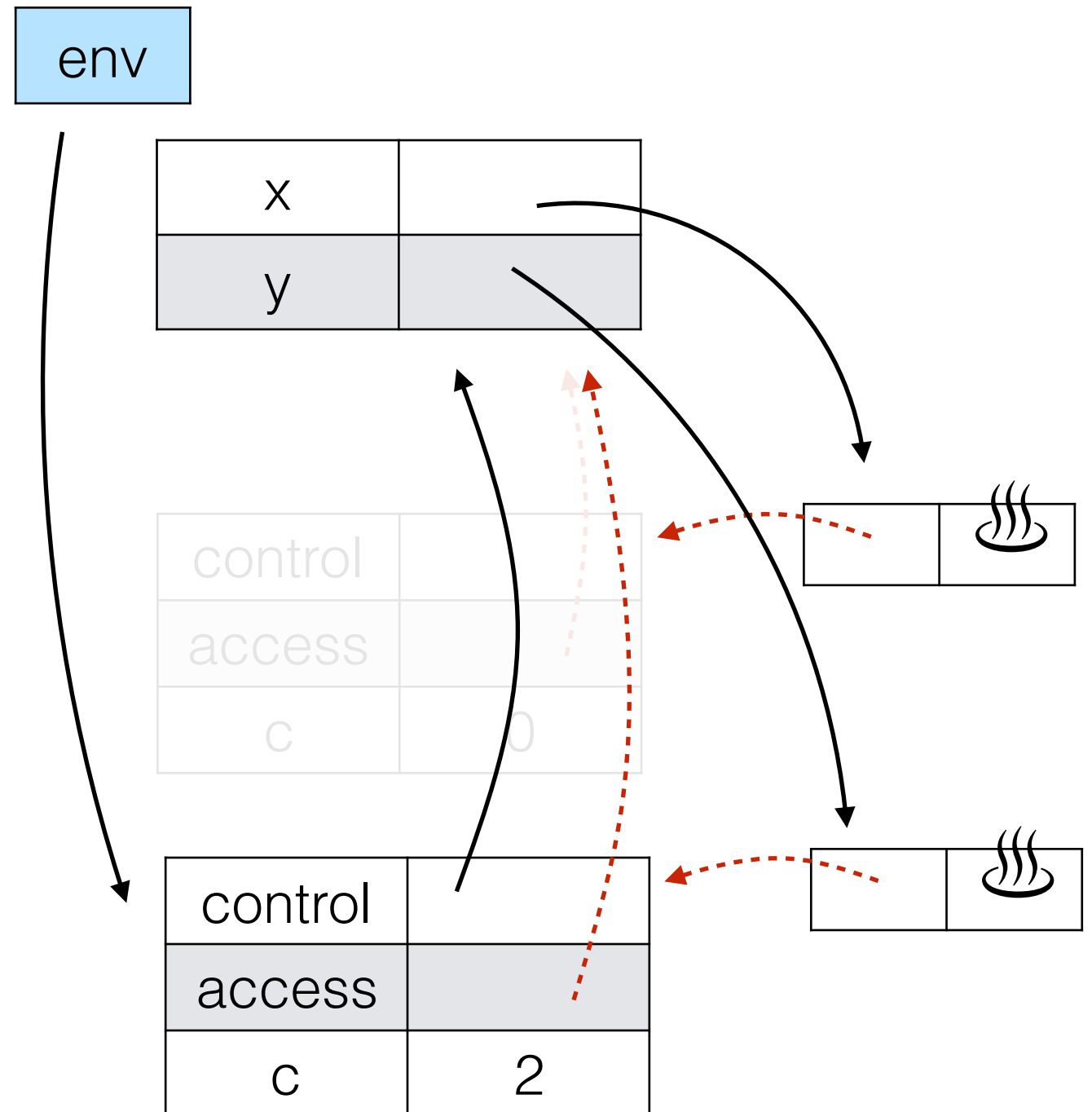


Higher-order functions

- Consider the use of high-order mkCounter function

```
function mkCounter(c) {  
  return () => {  
    return c++;  
  };  
}
```

```
let x = mkCounter(0);  
let y = mkCounter(2);  
console.log(x());
```

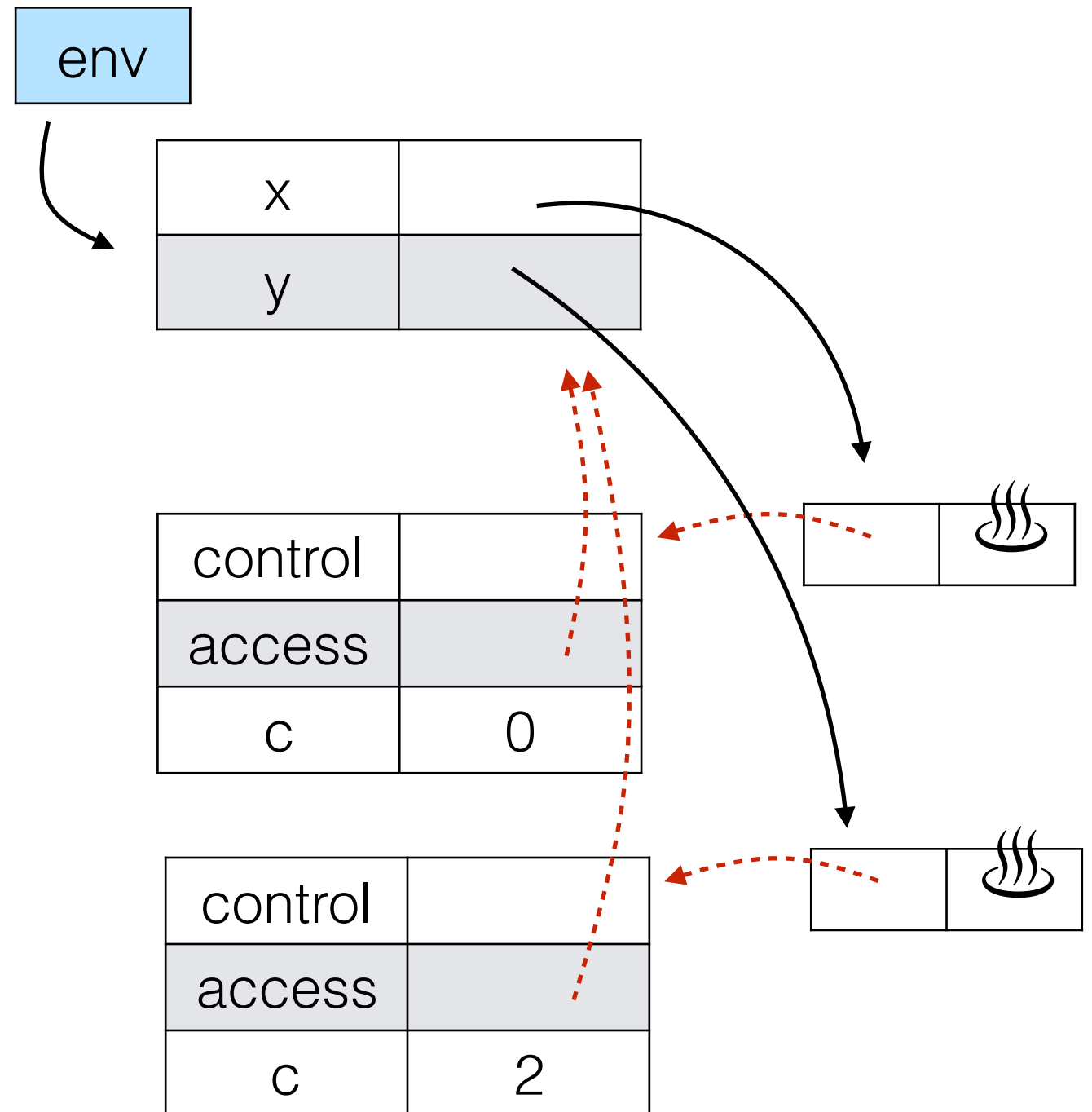


Higher-order functions

- Consider the use of high-order mkCounter function

```
function mkCounter(c) {  
  return () => {  
    return c++;  
  };  
}
```

```
let x = mkCounter(0);  
let y = mkCounter(2);  
console.log(x());
```

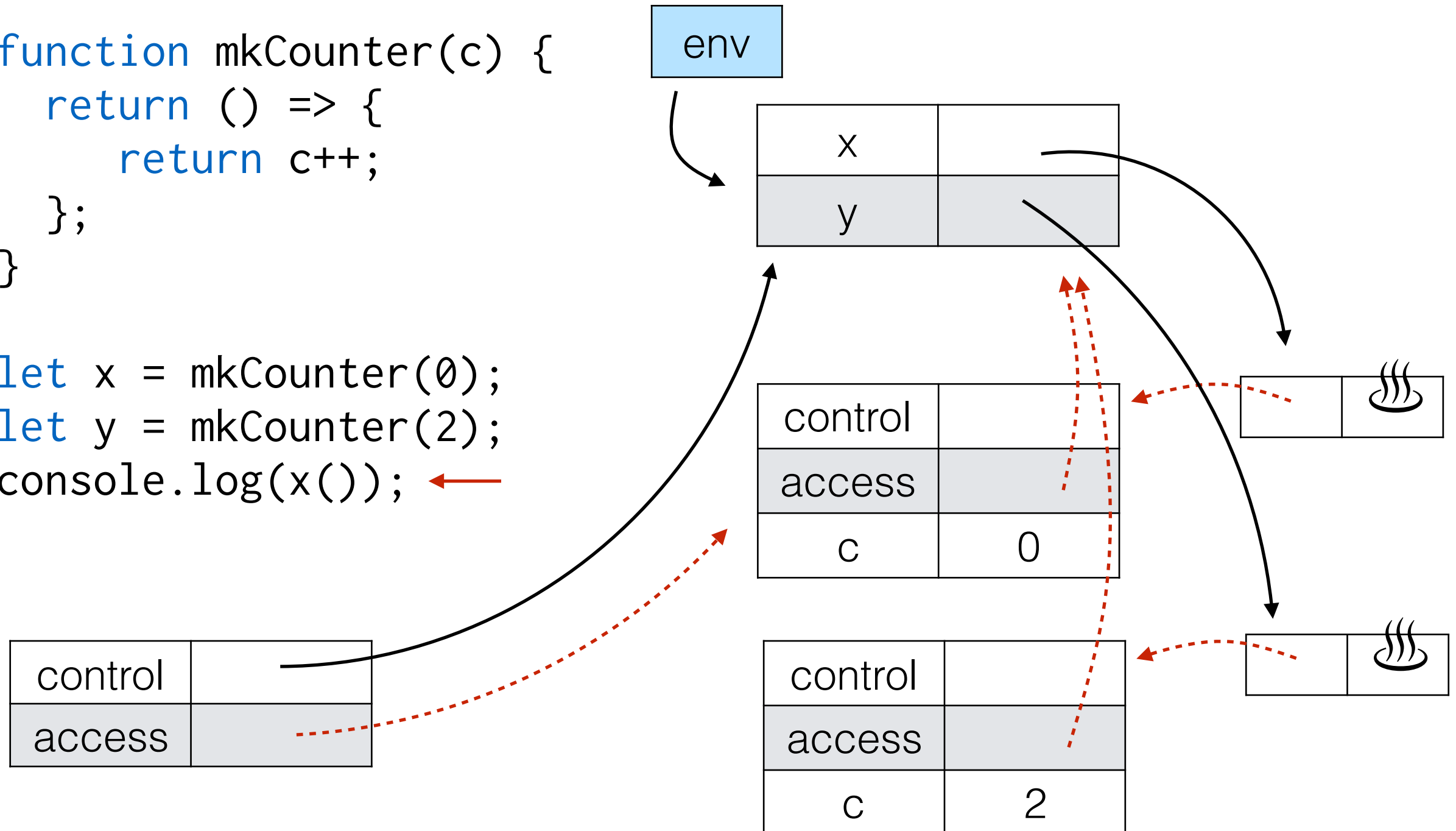


Higher-order functions

- Consider the use of high-order mkCounter function

```
function mkCounter(c) {  
  return () => {  
    return c++;  
  };  
}
```

```
let x = mkCounter(0);  
let y = mkCounter(2);  
console.log(x());
```

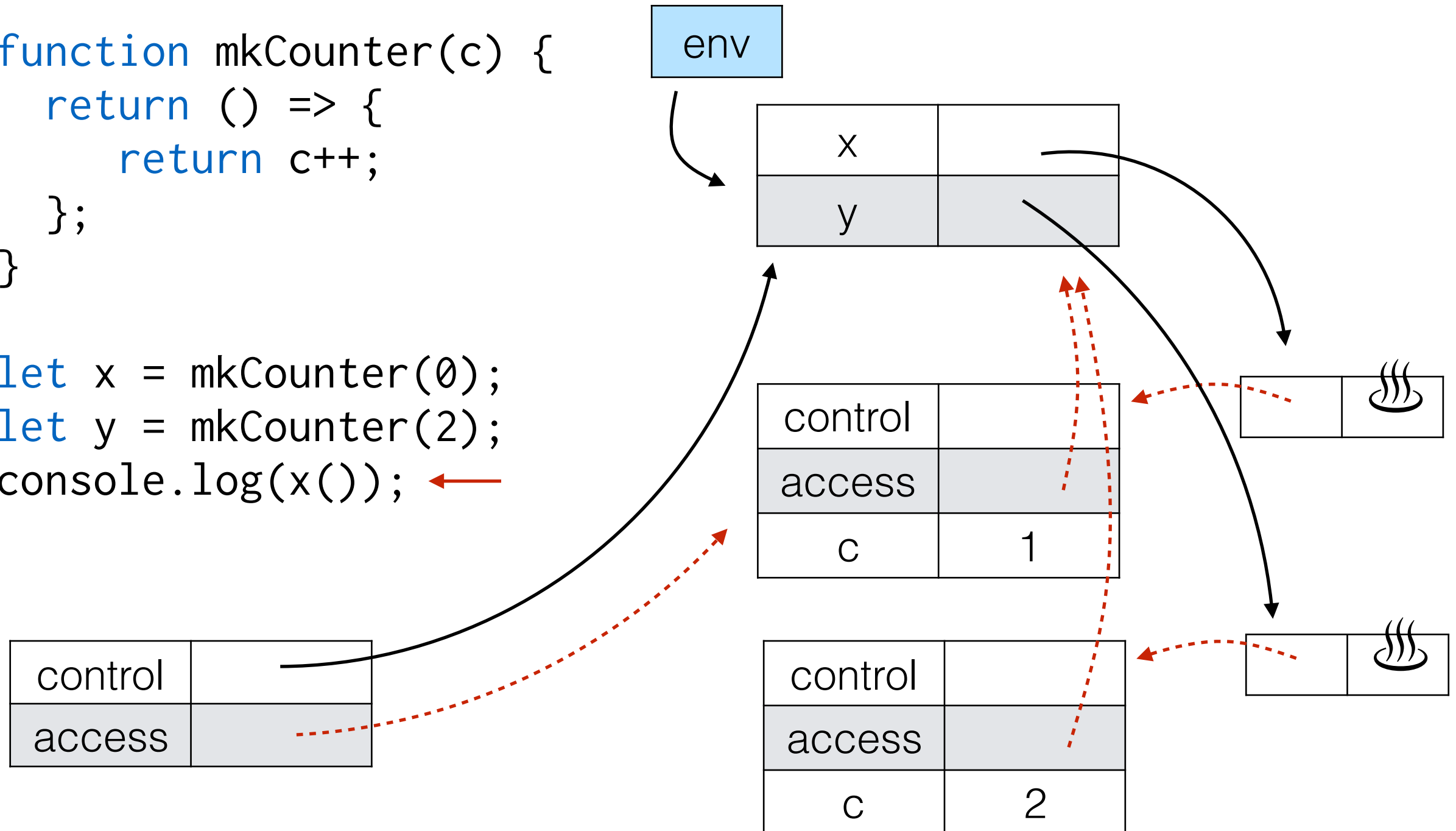


Higher-order functions

- Consider the use of high-order mkCounter function

```
function mkCounter(c) {  
  return () => {  
    return c++;  
  };  
}
```

```
let x = mkCounter(0);  
let y = mkCounter(2);  
console.log(x());
```



The environment model (by example)

- Anatomy of a scope ✓
- First-order functions ✓
- Free variables ✓
- High-order functions (bonus) ✓