Hack Your Language!

CSE401 Winter 2016

Introduction to Compiler Construction

Ras Bodik
Alvin Cheung
Maaz Ahmad
Talia Ringer
Ben Tebbs

Lecture 3: Scoping and Iterators

Lexical Scoping and its Interpreter

Simple Objects

Iterators

Announcements

PA1 will be due this Sunday at **11PM**

How did HW1 go?

Please sign up on Piazza!

- https://piazza.com/class#winter2016/cse401
- From now on, post on piazza instead of staff mailing list and catalyst discussion board
- Questions can be marked as private on piazza

Announcements

Final quiz

- Time TBD (either last lecture or last section)
- Only covers second half of the class
- You will be well prepared if you work on the assignments, the project, and attend lectures

Next Monday is a holiday

- We will give an extra lecture this week on Thurs or Fri
- Look out for piazza poll email

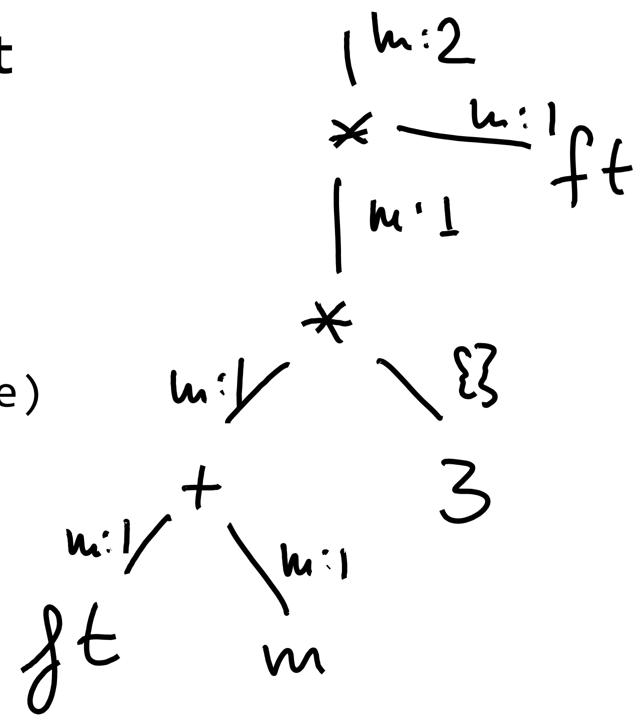
Outline for today

- More scoping
- Adding objects to our language
- Iterators

An interpreter for the unit language

Now we want to evaluate **`(ft +m)*3*ft`**

```
def eval(e):
    if type(e) == type(1):   return (e,{})
    if type(e) == type(1.1): return (e,{})
    if type(e) == type('m'): return lookupUnit(e)

def lookupUnit(u):
    return {
        'm'     : (1,      {'m':1}),
        'ft'    : (0.3048, {'m':1}),
        's'     : (1,         {'s':1}),
        'year'  : (31556926, {'s':1}),
        'kg'    : (1,            {'kg':1}),
        'lb'    : (0.45359237, {'kg':1})
        }[u];
```



how values are propagated up the tree

Rest of code at : http://bitbucket.org/bodik/cs164fa09/src/c73c51cfce36/L3-ConversionCalculator/Prep-for-lecture/ConversionCalculator.py

A simple core language

All constructs will be desugared to this language.

Key constructs in the language:

- *first-class* functions (i.e., they can be passed as values)
- definitions of local variables (variable binding)
- objects aside, these are sufficient to build a DSL like d3

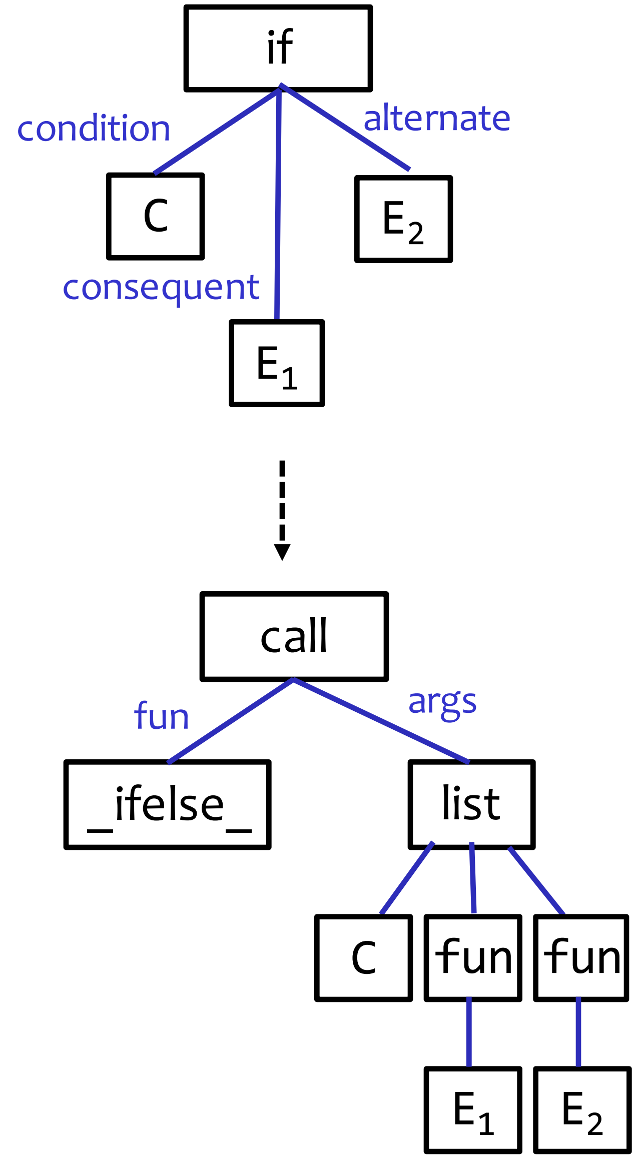
The grammar

```
E := n
   |  id                           // reference to a variable
   |  E+E | E-E | E/E | E*E
   |  function (id,…,id) { E }    // (anon.) function value
   |  E(E,…,E)                     // application (call)
   |  var id=E                     // var introduction
```

Rewrite rules

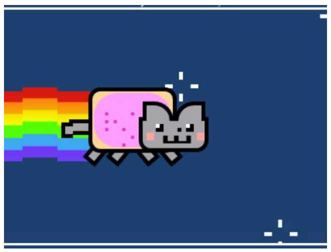
```
if (C) E₁ E₂
    is rewritten into
_ifelse_(C, function(){E₁},
            function(){E₂})
```

is an example of a rewrite rule



Desugaring happens between parsing and interpretation.

We will introduce a DSL to express these rules in PA



How to look up values of variables?

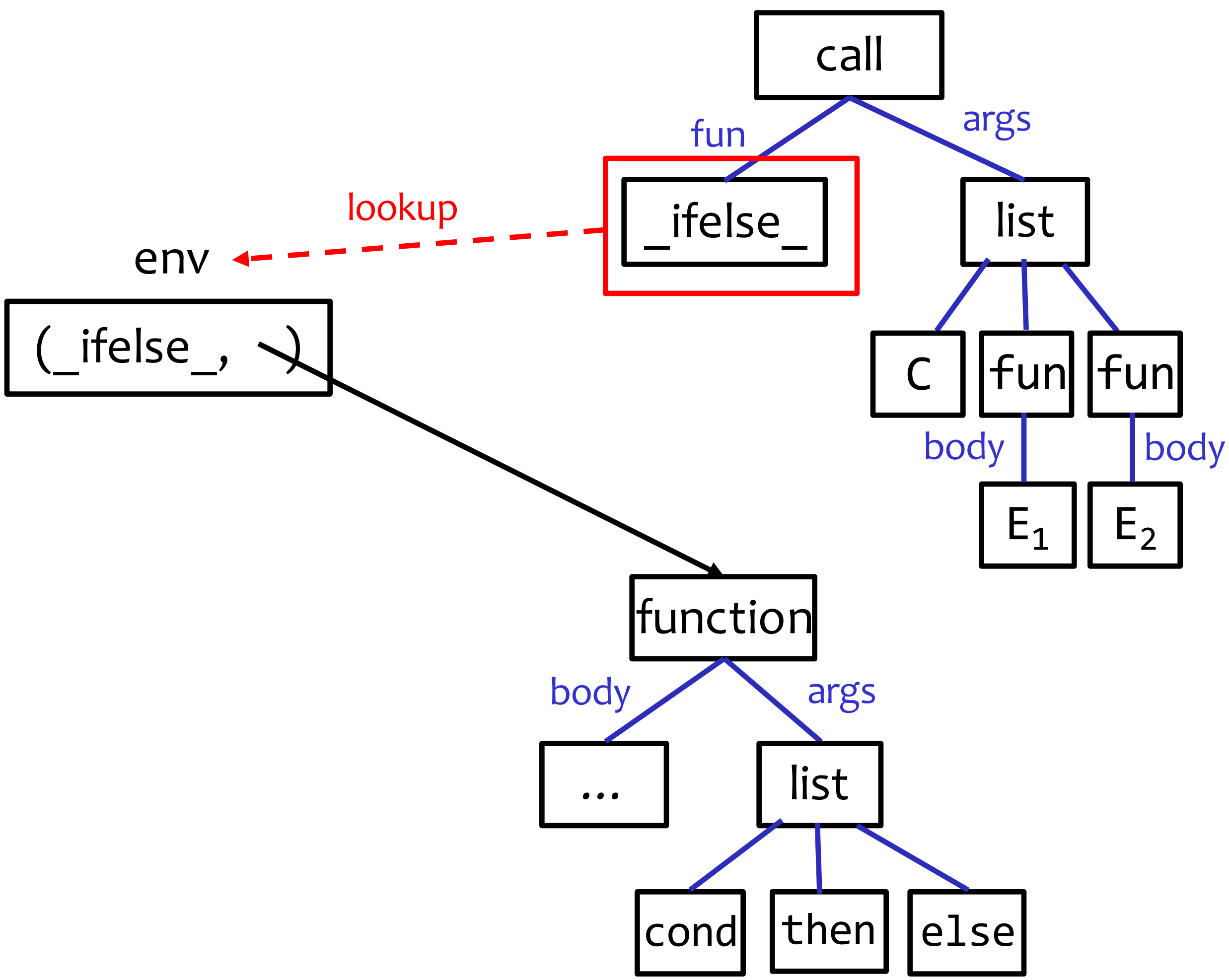
Environment: maps symbols (var names) to values

- Consists of (symbol, value) pairs
- env.lookup(sym) returns the value of the *first* sym in env
- the first variable shadows the pairs with the same name

What “first” means matters!

Example

looking up _ifelse_ in the environment
returns the (anonymous) function that is bound to _ifelse_



Scoping

A question of x's:

```
var x=1
function f(callback) {
    var x=2
    callback()
}
f( function(){ x } )
```

Which x should this return?

Note: f is a high-order function:

it accepts other functions as arguments

Scope

We must define where a variable is visible (its scope)

Dynamic scoping:

Return the value that was last declared during execution

Static (lexical) scoping:

Return the value that is the most local in terms of scope

Scope

We must define where a variable is visible (its scope)

Dynamic scoping:

Variable is visible globally until the end of its lifetime.

The environment is a **stack**. New bindings are pushed.

The lookup will proceed from the top of stack.

Static (lexical) scoping:

A function carries its own env (fun+env is called *closure*).

vars defined by different functions are kept separate.

Env is a **tree**; lookup proceeds from a leaf towards the root.

Interpreter for dynamic scoping

The dynamic-scoping interpreter

```
var env = […]                  // env is global; initially an empty stack
function eval(n) {
   switch (n.op) {
   case “int”:          return n.val
   case “id”:           return lookup(env,n.name)
   case “+”:            return eval(n.arg1) + eval(n.arg2)
   ...
   // function (id) { E }
   case “function”:  return { “ast_node”:  n }  // this dict is our fun value
   // E(E)
   case “call”:   var f = eval(n.fun) var a = eval(n.arg)
                  check if f is a function value. If not, exit with error
                  env.push(f.ast_node.param.name, a)
                  var ret = eval(f.ast_node.body)
                  env.pop()     // end the life time of the parameter
                  return ret    }}
```

Problems with dynamic scoping

Dynamic scoping

In dynamic scoping, `env.lookup("x")` returns

- the last x added to env
- that is still live.

Problem with dynamic scoping:

```
var x=1
f( function(){ x } )      ← 2 is returned!! Why?
function f(callback) {
    var x=2
    callback()
}
```

Note: hof is a high-order function:

- It accepts other functions as arguments

Dynamic scoping illustration

```
var x=1
function f(callback) {
        var x=2
        callback()
}
f( function(){ x } )
```

env (stack)

(x, 1)	← top

→ current program counter

```
var x=1
function f(callback) {
        var x=2
        callback()
}
f( function(){ x } )
```

env

(x, 1)	
(f, …)	
(x, 2)	← top

```
var x=1
function f(callback) {
        var x=2
        callback()
}
f( function(){ x } )
```

env

(x, 1)	
(f, …)	
(x, 2)	← top

What value of x is returned by f?

Learn more

- Find a language with dynamic scoping
- Study its tutorial and find useful applications of dynamic scoping
- Efficiency of name lookup in dynamic scoping:

  Our lookup must traverse the entire stack. Can you think of a constant-time algorithm for finding a variable in env.

Static scoping with closures

Why static scoping?

We want to look up var values in the environment where the function was defined

Not where it is called, as we saw in dynamic scoping

To do so, we need functions to "remember" where they were defined

- i.e., they need to "carry around" their env with them
- this is done using closures

Closures

Closure: a pair (function, environment)

the representation of function value in modern languages

the function:

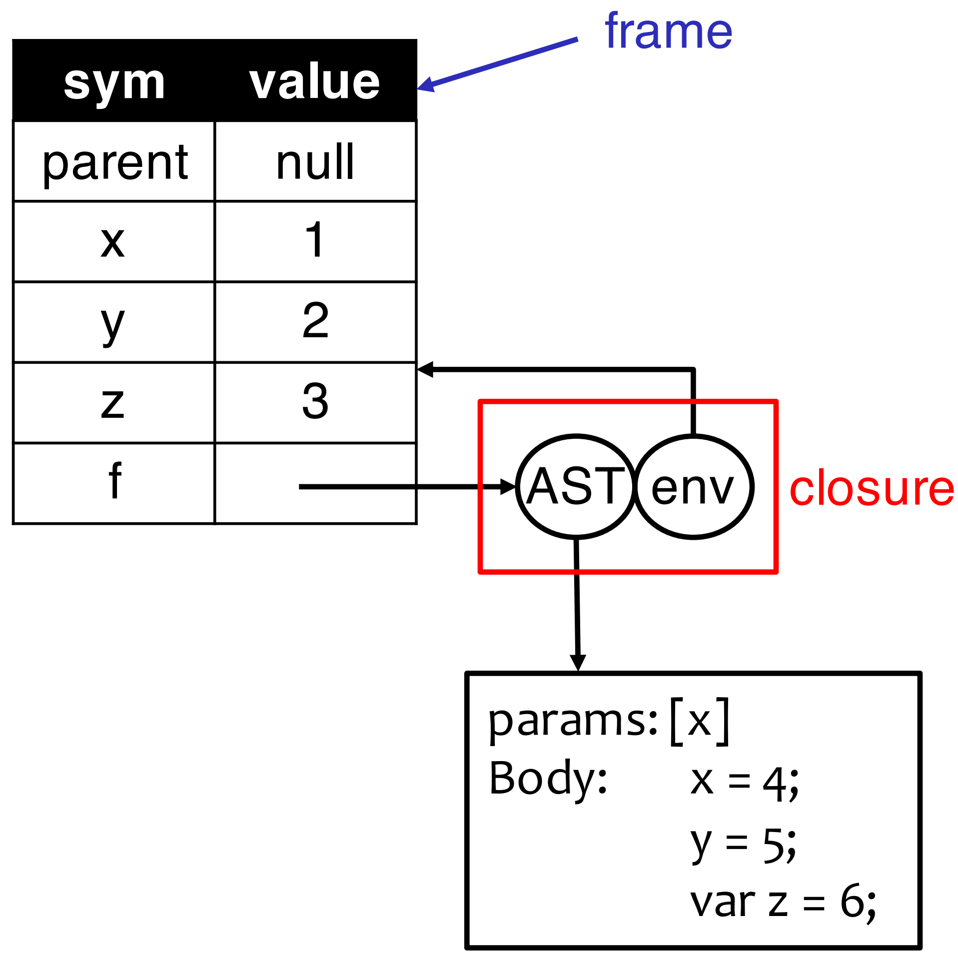
- "pointer" to function code
- Includes parameter names and the code of the body
- may have free variables, ex: `y` in `function(x) { x+y }`
  - these are resolved (looked up) using the function's environment

the environment:

- the environment in which the function was created
- where the function finds vars from its enclosing scope

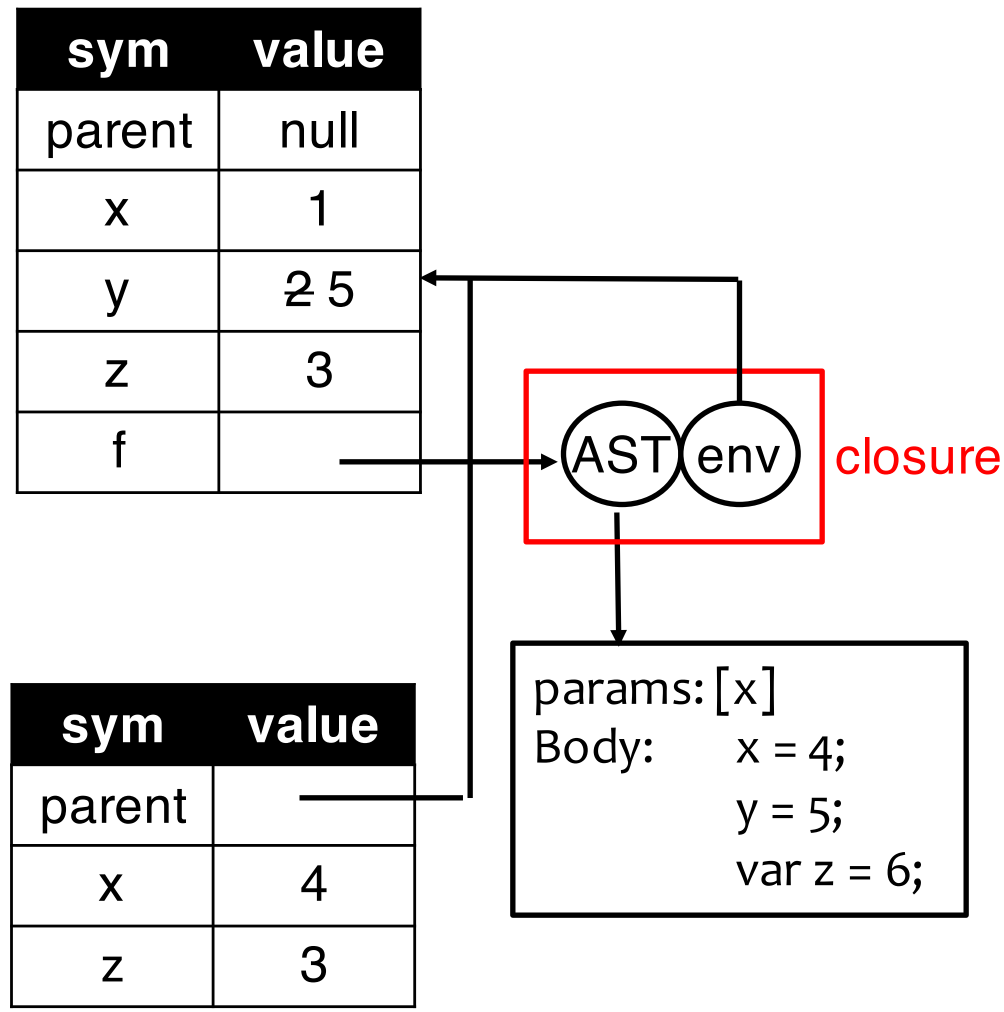
Example

```
var x = 1;
var y = 2;
var z = 3;
var f = function(x) {
      x = 4;
      y = 5;
      var z = 6;

}
// execution point 1
f(x)
```



Example

```
var x = 1;
var y = 2;
var z = 3;
var f = function(x) {
      x = 4;
      y = 5;
      var z = 6;
   // execution point 2
}

f(x)
```



Example with higher order functions

From the book *Programming in Lua*

```
names = { "Peter", "Paul", "Mary" }
grades = { Mary: 10, Paul: 7, Paul: 8 }
sort(names, function(n1,n2) {
      grades[n1] > grades[n2]
})
```

Sorts the list names based on grades.

grades not passed to sort via parameters but via closure

A cool closure

```
function derivative(f,delta)
    function(x) {
        (f(x+delta) - f(x))/delta
    }
}

var c = derivative(sin, 0.001)

print(cos(10), c(10))
   --> -0.83907, -0.83907
```

Summary of key concepts

- Idea: allow nested functions + allow access only to nonlocals in *parent* (ie statically outer) functions
- The environment: frames on the parent chain
- Name resolution for x: first x from on parent chain
- Solves modularity problems of dynamic scoping
- Functions are now represented as closures, a pair of (function code, function environment)
- Frames created for a function's locals survive after the function returns
- This allows creating data on the heap, accessed via functions (eg a closure that increments its counter)

The interpreter for static scoping

Static-scoping interpreter

This part is the same as in dynamic scoping

except that env is passed into recursive calls to eval, which is cleaner than updating the global env

```
function eval(n, env) {
        switch (n.op) {
        case “int”: return n.arg1
        case “id”:  return env.lookup(n.arg1)
        case “+”:   return eval(n.arg1, env) + eval(n.arg2, env)
        …
}}
```

The lexical-scoping interpreter

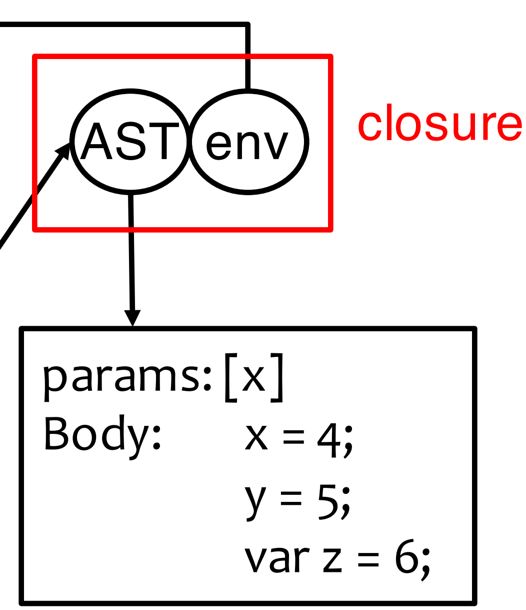
```
eval(program, { "parent": null })  // env with an empty frame
function eval(n, env) {
 switch (n.op) {
 ...
 case "function":  // construct and return the closure
     return { "ast_node": n, "env": env }
```

Example code

pc

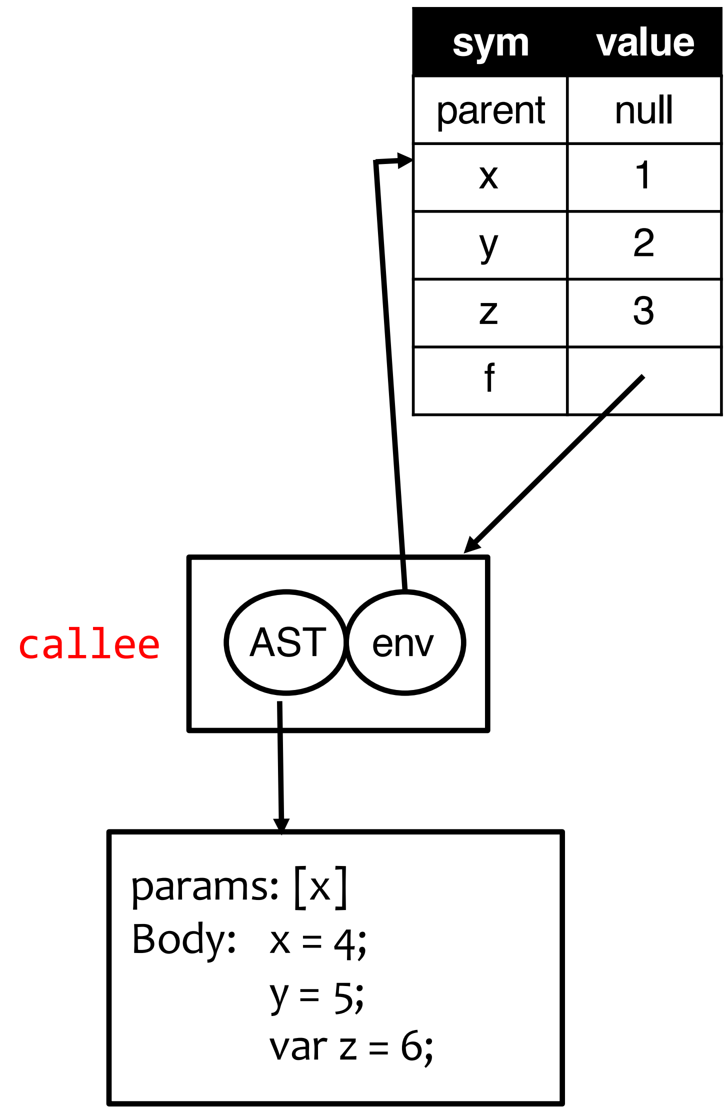
```
var x = 1;
var y = 2;
var z = 3;
var f = function(x) {
        x = 4;
        y = 5;
        var z = 6;
}

f(x)
```

sym	value
parent	null
x	1
y	2
z	3
f	



The lexical-scoping interpreter

```
case "call":
  var callee = eval(n.fun, env)
```



Example code

```
var x = 1;
var y = 2;
var z = 3;
var f = function(x) {
        ...
}
f(x)
```

pc →

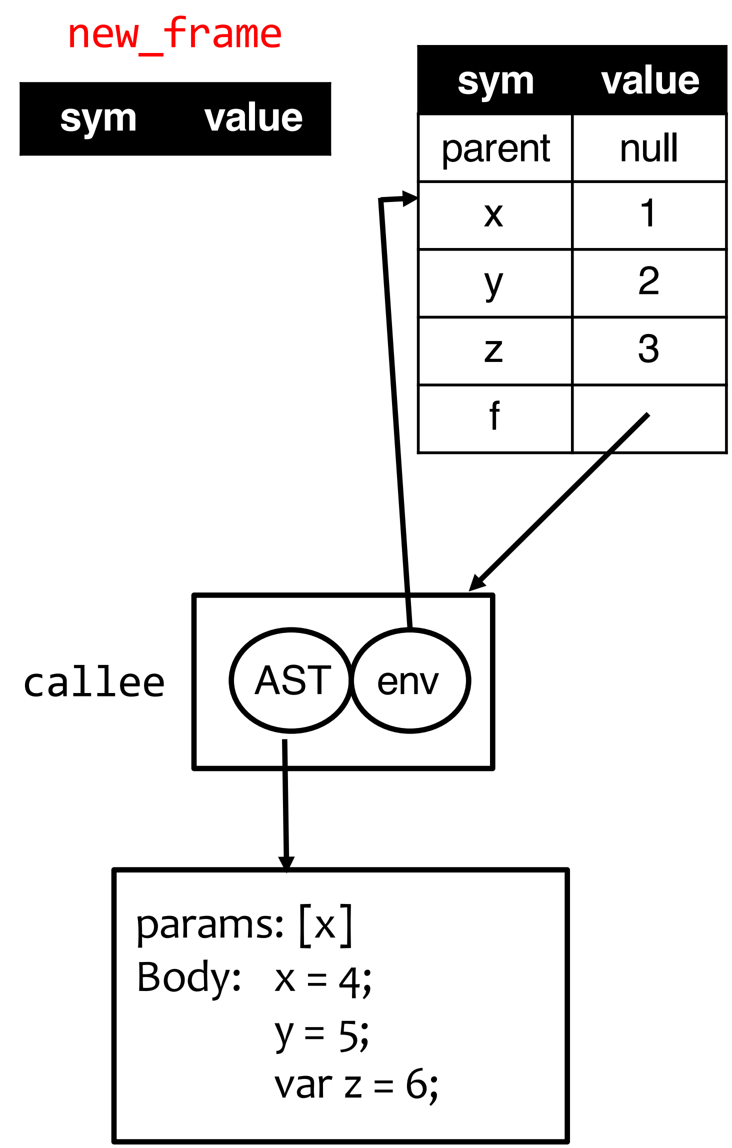
The lexical-scoping interpreter

```
case "call":
  var callee = eval(n.fun, env)
  var arg = eval(n.arg, env)
```
check if f is a function value. If not, exit with error!
```
  var new_frame = Frame()
```

Example code

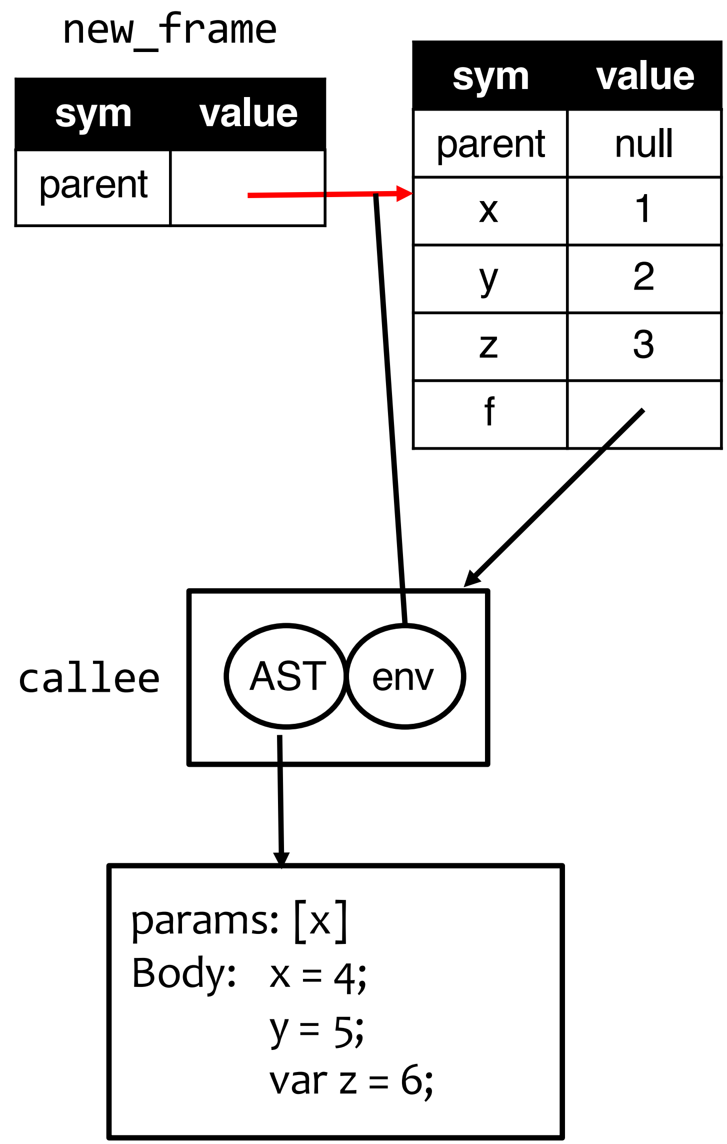
```
var x = 1;
var y = 2;
var z = 3;
var f = function(x) {
       ...
}
f(x)
```

pc



The lexical-scoping interpreter

```
case "call":
  var callee = eval(n.fun, env)
  var arg = eval(n.arg, env)
  check if f is a function value. If not, exit with error!
  var new_frame = Frame()
  new_frame.parent = callee.env
```



Example code

```
var x = 1;
var y = 2;
var z = 3;
var f = function(x) {
      ...
}
f(x)
```

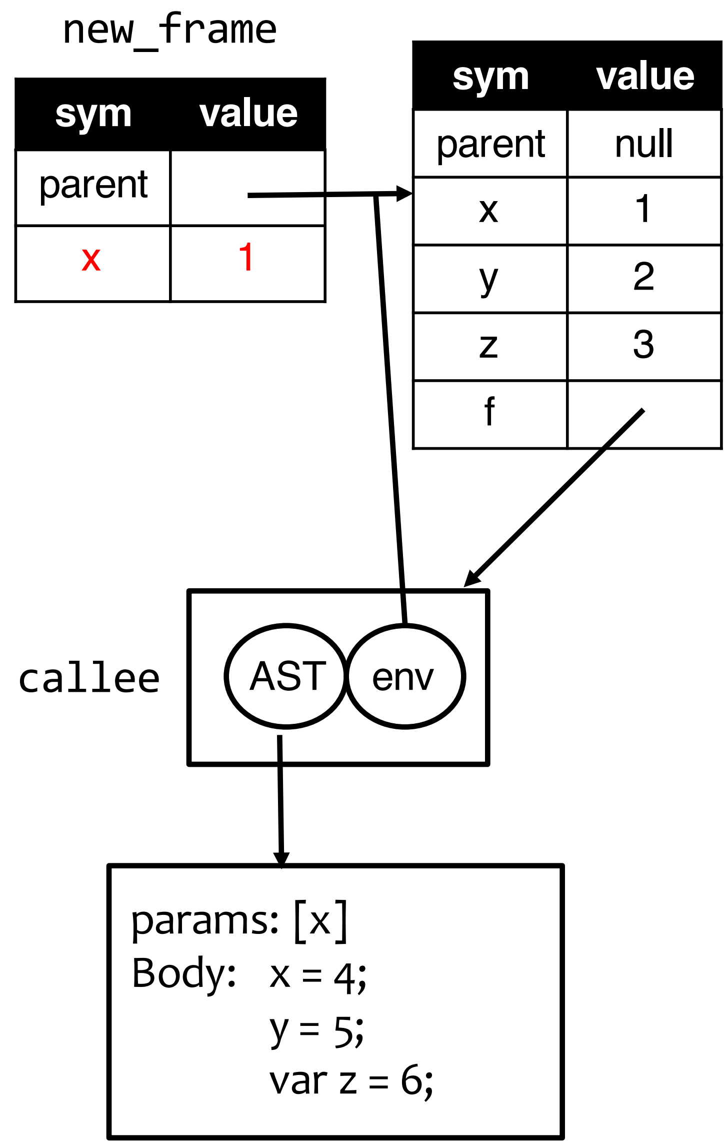
pc → f(x)

The lexical-scoping interpreter

```
case "call":
  var callee = eval(n.fun, env)
  var arg = eval(n.arg, env)
```

check if f is a function value. If not, exit with error!

```
  var new_frame = Frame()
  new_frame.parent = callee.env
  new_frame[callee.ast_node.param.name] = arg
```



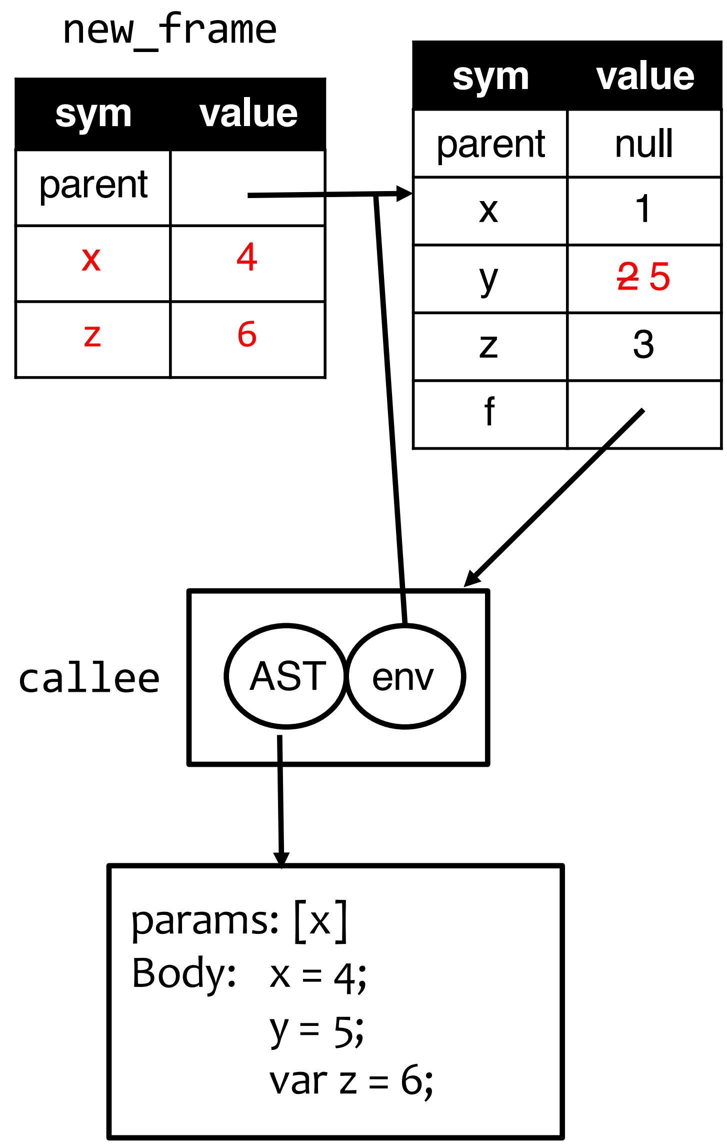
Example code

```
var x = 1;
var y = 2;
var z = 3;
var f = function(x) {
      ...
}
f(x)
```

pc →

The lexical-scoping interpreter

```
case "call":
  var callee = eval(n.fun, env)
  var arg = eval(n.arg, env)
```
check if f is a function value. If not, exit with error!
```
  var new_frame = Frame()
  new_frame.parent = callee.env
  new_frame[callee.ast_node.param.name] = arg
  return eval(callee.ast_node.body, new_frame)
```



Example code

```
var x = 1;
var y = 2;
var z = 3;
var f = function(x) {
      ...
}
f(x)
```
pc → f(x)

Intermission

Introduction to Objects

Recall from CSE 143

What are objects

- state (attributes) and
- code (methods)
- objects belong to classes, and classes can be inherited

Why objects?

abstraction: hide implementation using encapsulation

Why inheritance?

reuse: specialization of an object's behavior reuses its code

For now, let's add simple objects – dicts!

Three operations:

- {}
- obj[k]
- obj[k]=v

What about arrays?

- Objects whose keys are numeric indices!
- This is actually how it's done in Lua.

Iterators

Iterators

Whenever a language includes collections

or allows you to build one

we also want constructs for iterating over them

Example: d3 selections (sets of DOM nodes)

The each operator in

```
aSelection.each(aFunction)
```

is an iterator (implemented as a function)

Let's design a `for` iterator

We need to worry about two things:

- What data can **for** iterate over?
- What can be in the body?

Let's design a `for` iterator (behavior)

Desired behavior: say want to iterate from 1 to 10:

```
for x in iter(10) { print x }
```

Q1: Is `iter` a keyword in the language? No, a function.

Q2: What does it return? An iterator function.

```
function iter(n) {
     var i = 0
     function () {
          if (i < n) { i = i + 1; i }
          else { null }
}    }
```

Let's design a `for` iterator (generality)

Q3: In general, what constructs to permit in __ ?

```
for x in __ { print x }
```

A: Any expression that returns an iterator function.

- the syntax of `for` is thus: **`for`** *`ID`* **`in`** *`E`* **`{`** *`S`* **`}`**
- these are all legal programs:

```
for x in myIter { S }
for x in myIterArray[2] { S }
for x in myIterFactory() { S }
for x in myIterFactoryFactory()() { S }   // ☺
```

Let's design a `for` iterator (scoping)

Q4: What is the scope x?

```
for x in E { S }
```

Q5: In what environment should *E* be evaluated?

In particular, should the environment include x?

E should be evaluated in e, the environment of **for**.

S should be evaluated in e extended with the binding for x.

Implementing the **for** iterator

We are done with the design of behavior (semantics).

Now to implementation. We'll desugar it, of course.

```
for ID in E { S }
        ↓
{  // a block to introduce new scope
   var t1 = E
   var ID = t1()
   while (ID != null) {
         S
         ID = t1()
   }
}
```

Side note: the block scope

A new scope can be introduced by desugaring, too:

```
{ S }  → ( function(){ S } )()
```

This trick is used in JS programs to restrict symbol visibility, i.e. to implement a simple *module* construct.