Hack Your Language!

CSE401 Winter 2016

Introduction to Compiler Construction

Ras Bodik
Alvin Cheung
Maaz Ahmad
Talia Ringer
Ben Tebbs

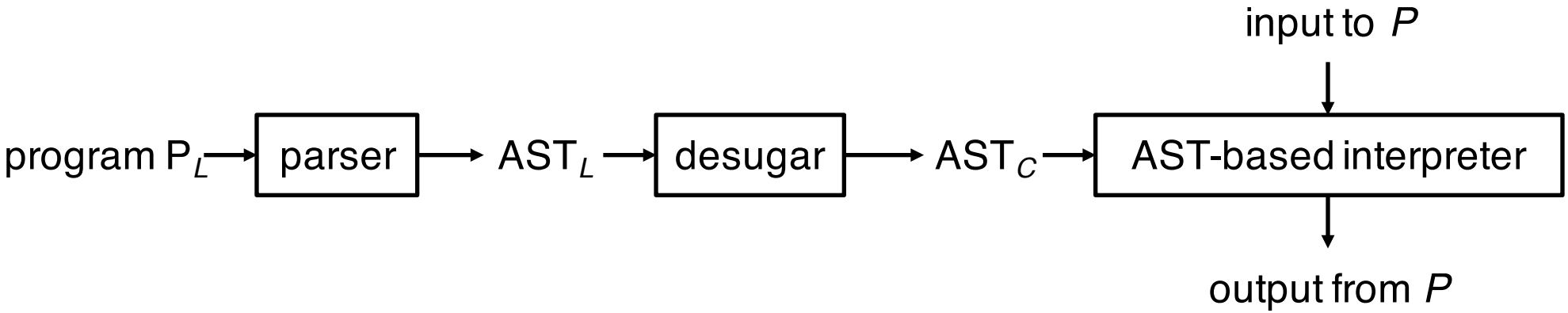
Lecture 5: Bytecode Compiler

Implementing Coroutines
Compile AST to bytecode
Bytecode interpreter

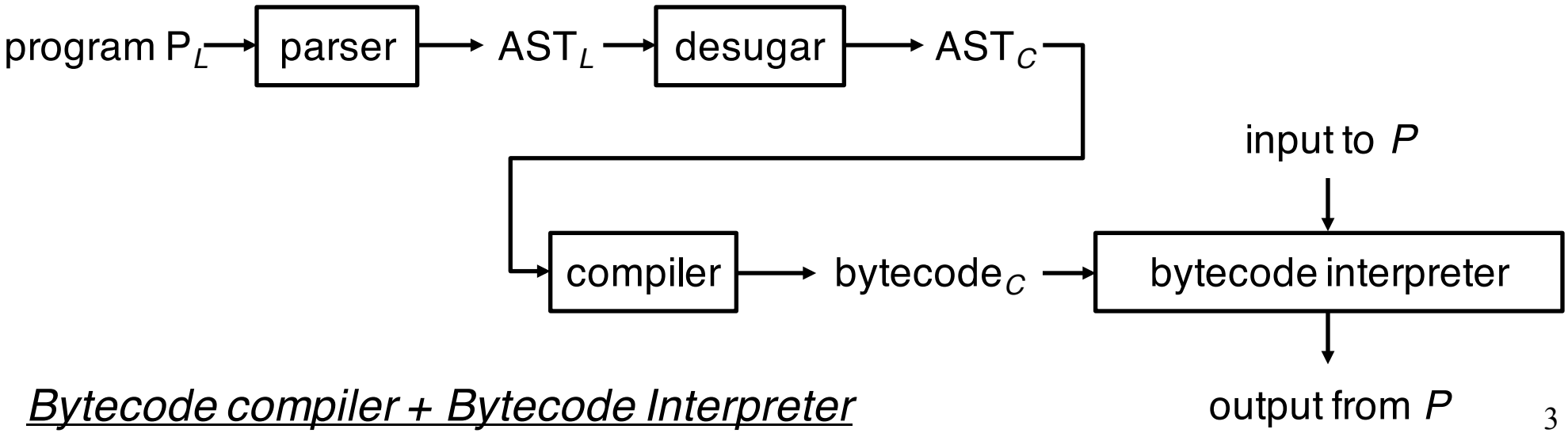
Announcements

- HW2 and PA2 will be out next Monday
- Maaz will have OH after class

What you will learn today



AST Interpreter



Bytecode compiler + Bytecode Interpreter

What you will learn today

Implement Asymmetric Coroutines

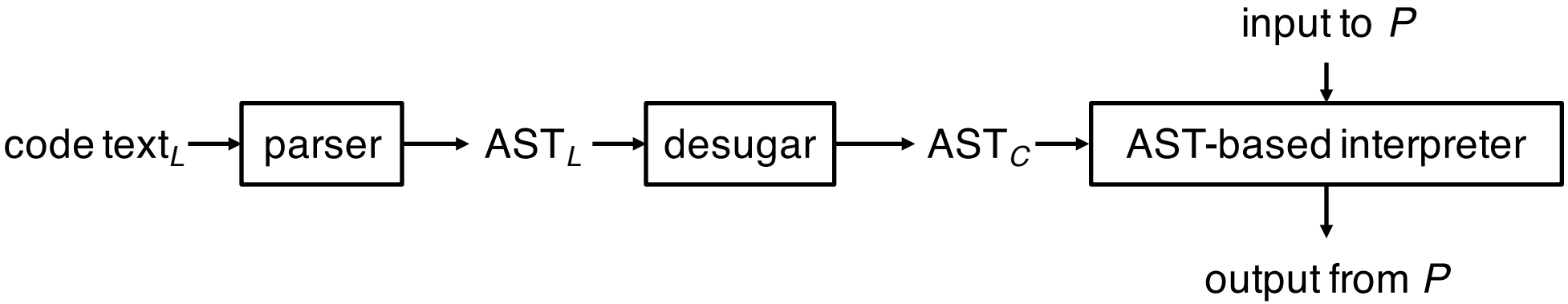
- why a recursive interpreter with implicit stack won't do

Compiling AST to bytecode

- this is your first compiler; compiles AST to "flat" code

Bytecode Interpreter

- bytecode can be interpreted without recursion
- hence no need to keep interpreter state on the call stack



Taking a Closer Look at Coroutines

Review: Why are coroutines useful?

Loop calls the iterator function to get the next item

- e.g., the next token from the input stream

The iterator maintains state between two such calls

- e.g., pointer to the input stream

Maintenance of that state may be difficult

- see the permutation iterator in previous lecture

Industrial-strength justification of Python generators

- restricted Python's coroutines

Review: Three coroutine constructs

co = coroutine(body) function → handle

creates a coroutine whose body is the argument function, returns a handle to the coroutine, which is ready to run

yv = resume(co, rv) handle × value → value

resumes the execution into the coroutine co, passing rv to co's yield expression (rv becomes the value of yield)

rv = yield(yv) value → value

transfers control to the master (ie, the coroutine that resumed into the current coroutines), passing yv to resume

Example

```
def f(x,y) {
  cg=coroutine(g)
  resume(cg,x,y)
}

f(1,2)
```

```
def g(x,y) {
  ck=coroutine(k)
  def h(a,b) {
    3 + yield(resume(ck,a,b))
  }
  h(x,y)
}
```

```
def k(x,y) {
   yield(x+y)
}
```

The call `f(1,2)` evaluates to __.

What was going on

```
def f(x,y) {
  cg=coroutine(g)
  resume(cg,x,y)









}

f(1,2)
```

```
def g(x,y) {
  ck=coroutine(k)
  def h(a,b) {
    3 +
    yield(
      resume(ck,a,b))
  }

  h(x,y)
}
```

```
def k(x,y) {

yield(x+y)

}
```

What was going on

```
def f(x,y) {
  cg=coroutine(g)
  resume(cg,x,y)









}

f(1,2)
```

```
def g(x,y) {
  ck=coroutine(k)
  def h(a,b) {
    3 +
    yield(
      resume(ck,a,b))
  }

  h(x,y)
}
```

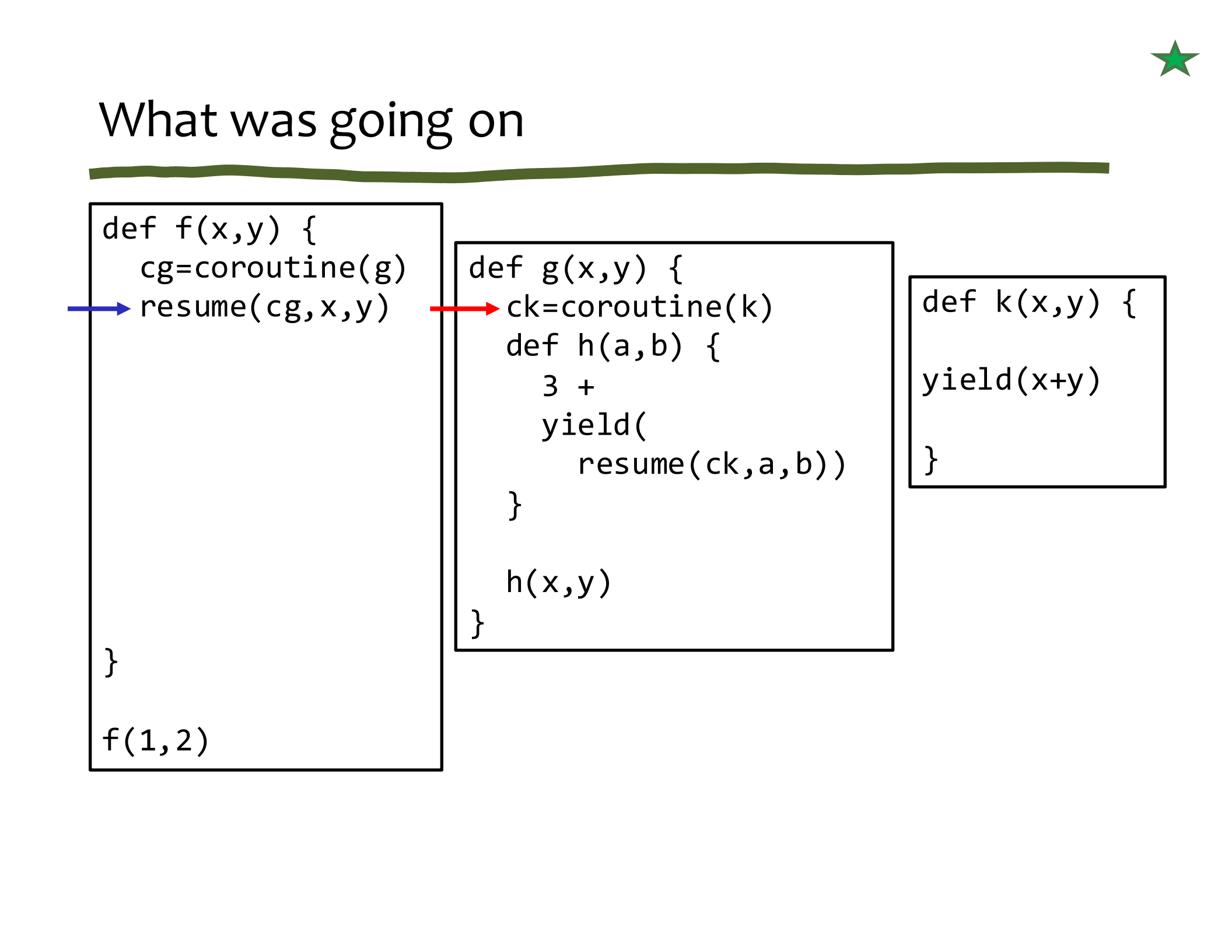
```
def k(x,y) {

yield(x+y)

}
```

What was going on

```
def f(x,y) {
  cg=coroutine(g)
  resume(cg,x,y)




}

f(1,2)
```

```
def g(x,y) {
  ck=coroutine(k)
  def h(a,b) {
    3 +
    yield(
      resume(ck,a,b))
  }

  h(x,y)
}
```

```
def k(x,y) {

yield(x+y)

}
```

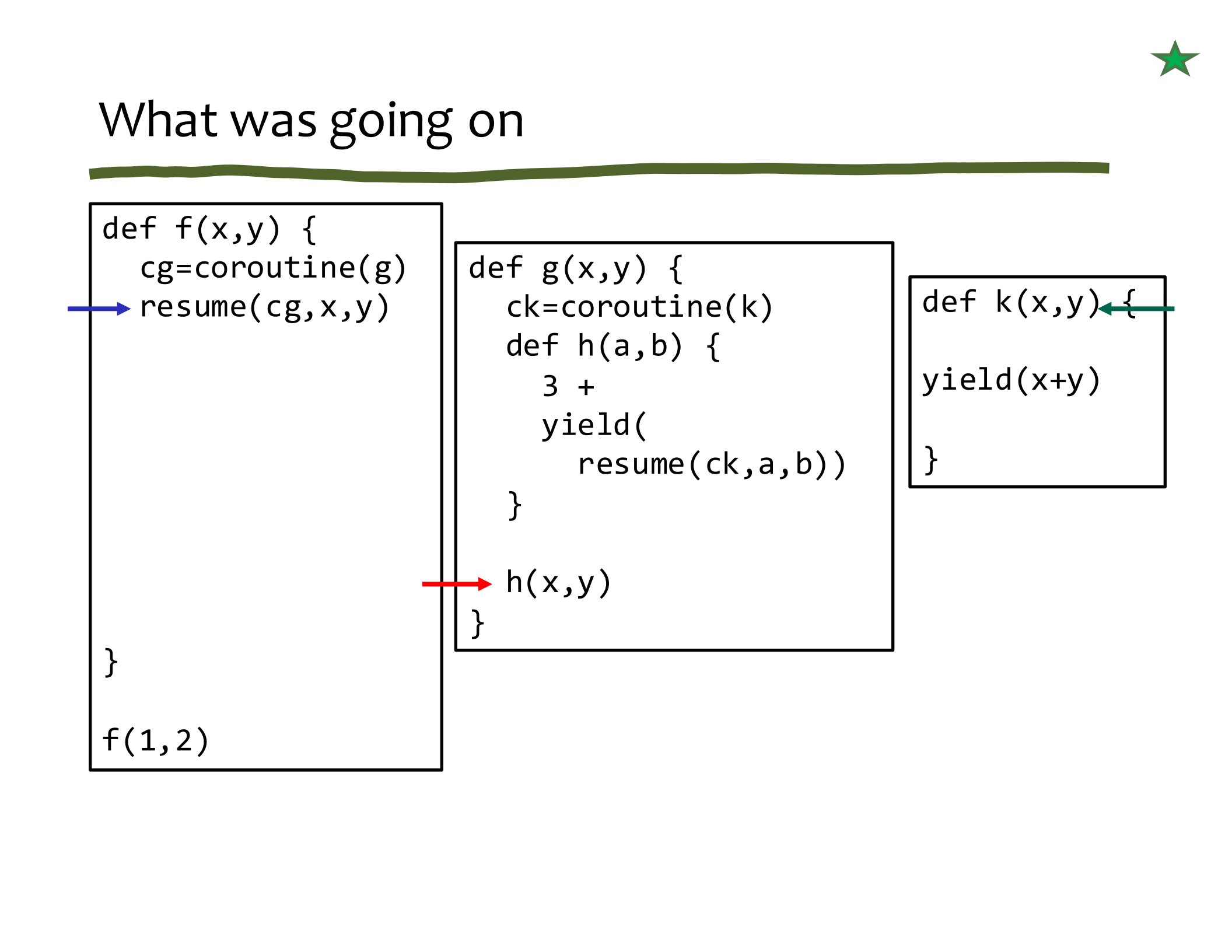


What was going on

```
def f(x,y) {
  cg=coroutine(g)
  resume(cg,x,y)




}

f(1,2)
```

```
def g(x,y) {
  ck=coroutine(k)
  def h(a,b) {
    3 +
    yield(
      resume(ck,a,b))
  }

  h(x,y)
}
```

```
def k(x,y) {

yield(x+y)

}
```


What was going on
def f(x,y) {
cg=coroutine(g)
resume(cg,x,y)
}
f(1,2)
def g(x,y) {
ck=coroutine(k)
def h(a,b) {
3 +
yield(
resume(ck,a,b))
}
h(x,y)
}
def k(x,y) {
yield(x+y)
}

What was going on

```
def f(x,y) {
  cg=coroutine(g)
  resume(cg,x,y)




}

f(1,2)
```

```
def g(x,y) {
  ck=coroutine(k)
  def h(a,b) {
    3 +
    yield(
      resume(ck,a,b))
  }

  h(x,y)
}
```

```
def k(x,y) {

yield(x+y)

}
```

What was going on

```
def f(x,y) {
  cg=coroutine(g)
  resume(cg,x,y)




}

f(1,2)
```

```
def g(x,y) {
  ck=coroutine(k)
  def h(a,b) {
    3 +
    yield(
      resume(ck,a,b))
  }

  h(x,y)
}
```

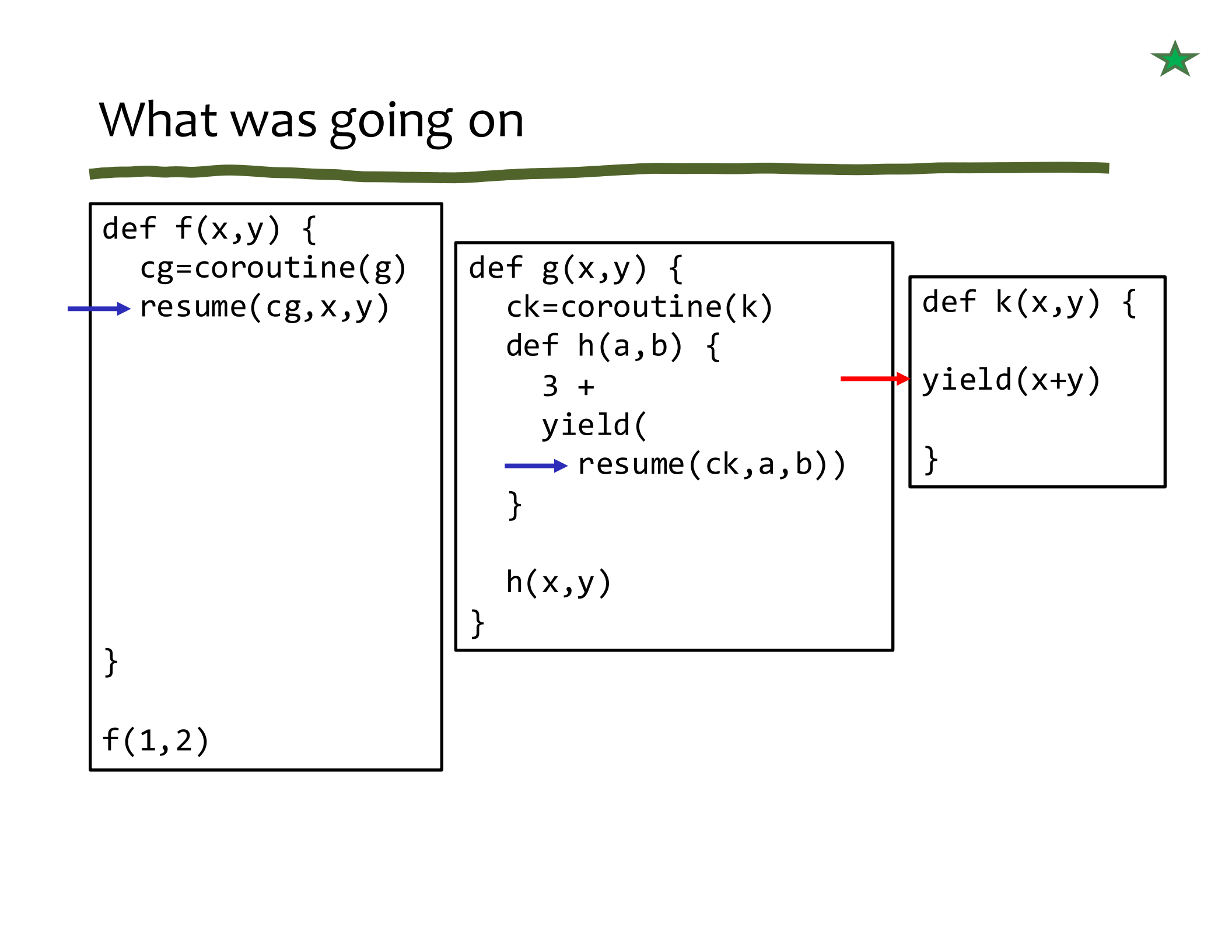
```
def k(x,y) {

yield(x+y)

}
```

What was going on

```
def f(x,y) {
  cg=coroutine(g)
  resume(cg,x,y)




}

f(1,2)
```

```
def g(x,y) {
  ck=coroutine(k)
  def h(a,b) {
    3 +
    yield(
    resume(ck,a,b))
  }

  h(x,y)
}
```

```
def k(x,y) {

yield(x+y)

}
```

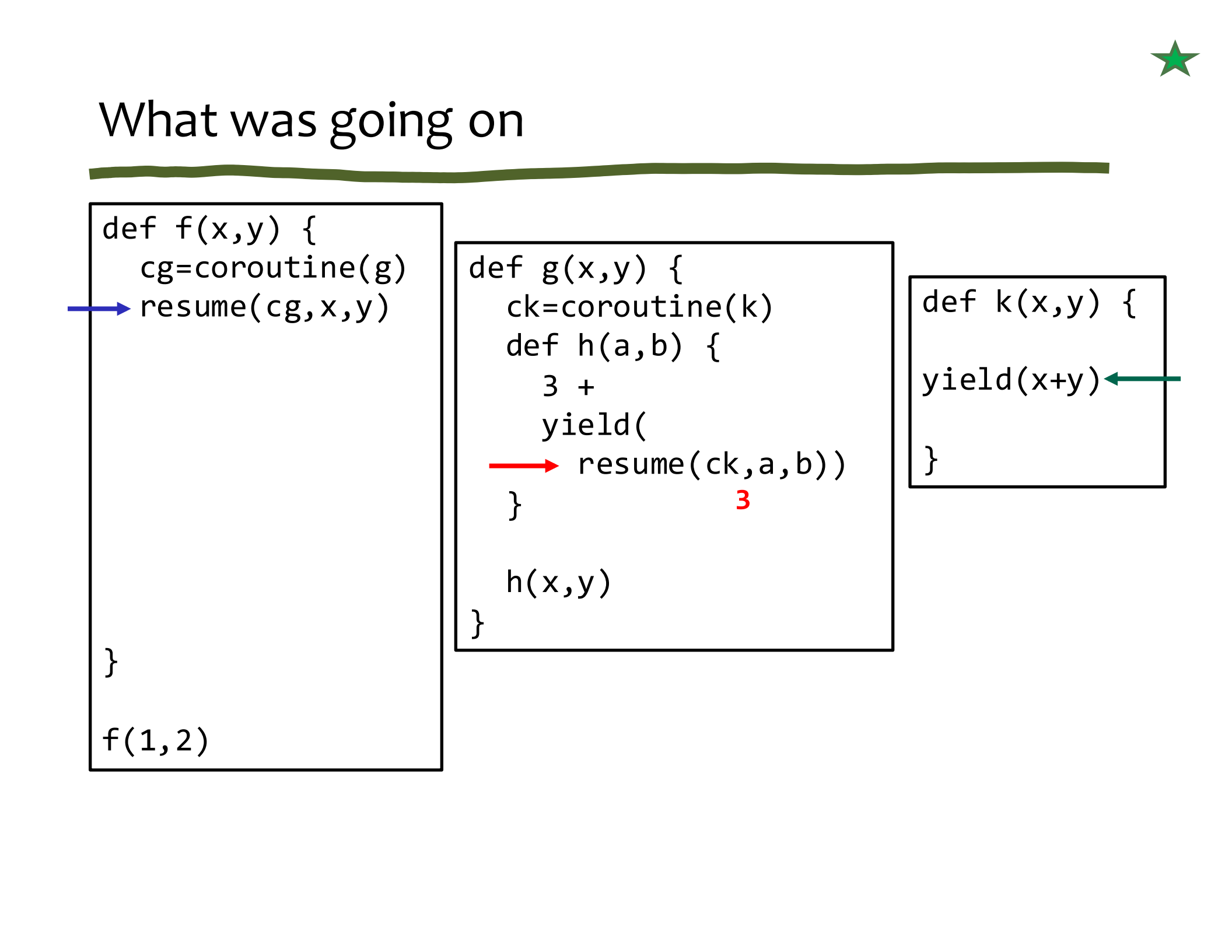


What was going on

```
def f(x,y) {
  cg=coroutine(g)
  resume(cg,x,y)




}

f(1,2)
```

```
def g(x,y) {
  ck=coroutine(k)
  def h(a,b) {
    3 +
    yield(
    resume(ck,a,b))
  }

  h(x,y)
}
```

```
def k(x,y) {

yield(x+y)

}
```

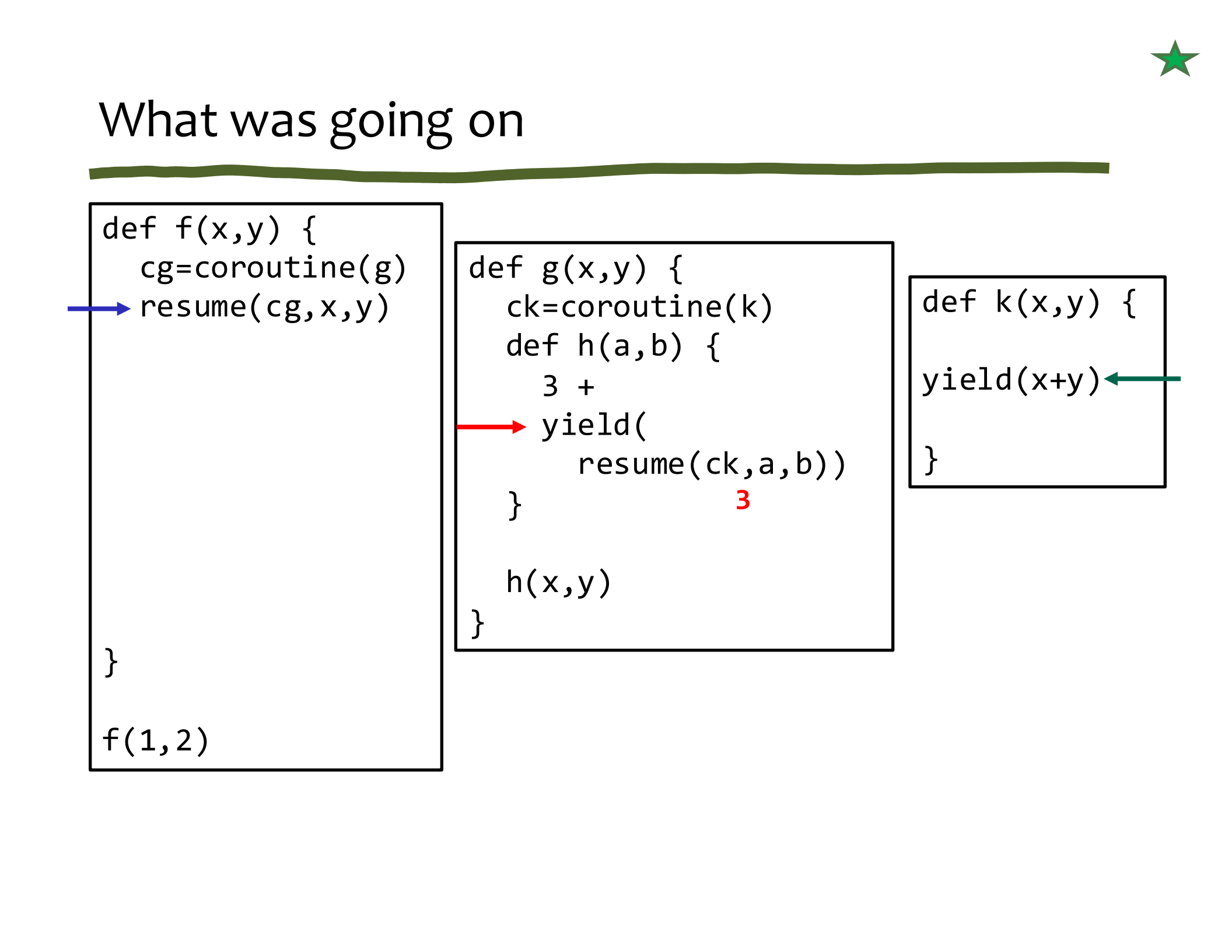


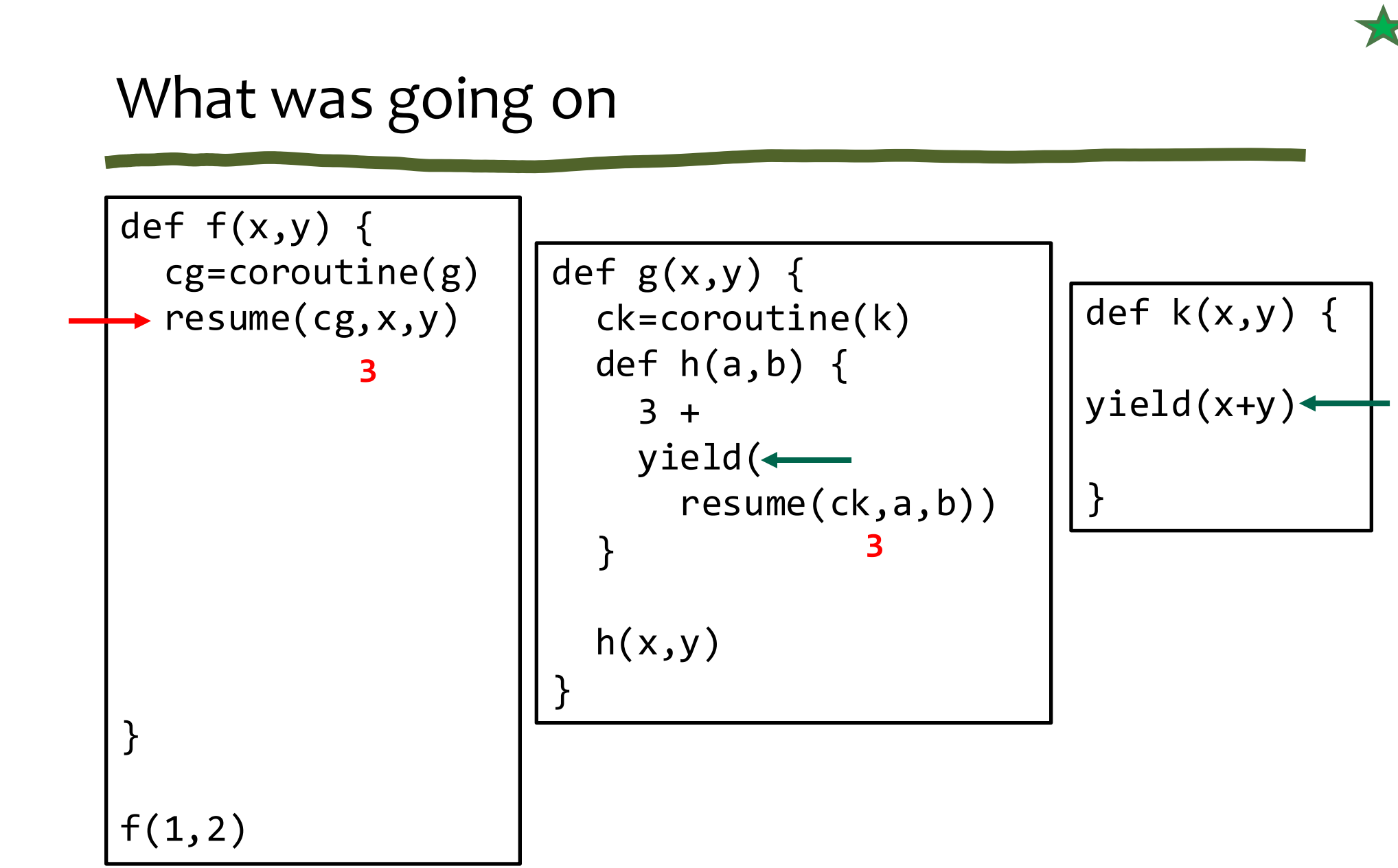
What was going on

```
def f(x,y) {
  cg=coroutine(g)
  resume(cg,x,y)




}

f(1,2)
```

```
def g(x,y) {
  ck=coroutine(k)
  def h(a,b) {
    3 +
    yield(
    resume(ck,a,b))
  }                  3

  h(x,y)
}
```

```
def k(x,y) {

yield(x+y)

}
```


What was going on
def f(x,y) {
cg=coroutine(g)
resume(cg,x,y)
}
f(1,2)
def g(x,y) {
ck=coroutine(k)
def h(a,b) {
3 +
yield(
resume(ck,a,b))
}
h(x,y)
}
3
def k(x,y) {
yield(x+y)
}


What was going on
def f(x,y) {
cg=coroutine(g)
resume(cg,x,y)
3
}
f(1,2)
def g(x,y) {
ck=coroutine(k)
def h(a,b) {
3 +
yield(
resume(ck,a,b))
3
}
h(x,y)
}
def k(x,y) {
yield(x+y)
}

A corner-case contest

Identify cases for which we haven't defined behavior:

1) yield executed in the main program

   let's define main as a coroutine; yield at the top level thus behaves as exit()

2) resuming to itself*

   illegal; can only resume to coroutines suspended in yield expressions or at the beginning of their body

3) return statement

   the (implicit) return statement yields back to resumer and terminates the coroutine

*exercise: test your PA2 on such a program

States of a coroutine

How many states do we want to distinguish?

suspended, running, terminated

Why do we want to distinguish them?

to perform error checking at runtime:

- do not resume into a running coroutine
- do not resume into a terminated coroutine

Are coroutines like calls?

Which of { resume, yield } behaves like a function call?

`resume` is more like call:

- like in regular call, control is guaranteed to return to `resume` (unless the program runs into an infinite loop)
- we can specify the target of the call (via coro. handle)

`yield` is more like return:

- like return, yield cannot control where it returns (always returns to its resumer's `resume` expression)
- no guarantee that the coroutine will be resumed after yield (eg, when a loop decides it need not iterate further)

Example from last lecture

Find the best first move in Scrabble given some time:

```
s = [‘a‘,‘f‘,...]          // 7 letter tiles
for p in permgen(s) {
    for s in subsets(p) {
      if (legalWord(s, wordDict)) {
            // check score of s
            // exit when out if time
        }
    }
}
```

Test yourself

Implement lazy list concatenation

(http://bit.ly/1U0Eaqv)

Language design question (1)

Since resume behaves like a call, do we need to introduce a separate construct to resume into a coroutine? Could we just do this?

```
co = coroutine(…)     // creates a coroutine
co(arg)               // resume to the coroutine
```

Yes, we could. But we will keep `resume` for clarity.

Compare: in Python generators, resume is a method call .next() in a generator object. Do you like the fact that resume appears to be a call?

Language design question (2)

In 164/Lua, a coroutine is created with corutine(lam).

In Python, a coroutine is created by a call to function that syntactically contains a yield:

```
def fib():          # function fib is a generator/corou
    a, b = 0, 1
    while 1:
        yield b          # … because it contains a yield
        a, b = b, a+b

it = fib()           # create a coroutine
print it.next();    # resume to it with .next()
print it.next(); print it.next()
```

Language design question (2, cont'd)

Python creates coroutine by calling a function with yield? How does it impact programming?

What if the function with yield needs to call itself recursively, as in permgen from Lecture 4?

```
def permgen(a,n) { ... yield(a) ... permgen(a,n-1) ... }
def permutations(a) {
    def co = coroutine( permgen )
    lambda () { resume(co, a) }
}
```

You should know a workaround from this limitation

That is, how would you write permgen in Python?

See how Python writes recursive tree generators

Exercise on Lua vs. Python

A recursive Lua Fibonacci iterator:

```
def fib(a,b) {
     yield b
     fib(b,a+b)
}
def fibIterator() {
    def co = coroutine( fib )
    lambda () { resume(co, 0, 1) }
}
```

Rewrite it into a recursive Python generator.

Symmetric vs. asymmetric coroutines

Symmetric: one construct (yield)

- as opposed to resume and yield
- yield(co,v): has the power to transfer to any coroutine
- all transfers happen with yield (no need for resume)

Asymmetric:

- resume transfers from resumer (master) to corou (slave)
- yield(v) always returns to its resumer

A language sufficient to explain coroutines

Language with functions of two arguments:

```
P ::= D*, E
D ::= def f(ID,ID) { P }

E ::= n
    | ID(E,E)
    | def ID = coroutine(ID)
    | resume(ID,E,E)
    | yield(E)
```

sequence of declarations followed by an expression

Implementation notes

Stackless Python is a controversial rethinking of the Python core (PEP 255)

Summary

Coroutines support backtracking.

- In match, backtracking happens in seq, where we pick a match for patt1 and try to extend it with a match for patt2.
- If we fail, we backtrack and pick the next match for patt1, and again see if it can be extended with a match for patt2.

Test yourself

Draw the state of our coroutine interpreter

(link to an excercise)

Why a recursive interpreter cannot implement (deep) coroutines

The plan

We will attempt to extend Section 2 to support coroutines

- treating resume/yield as call/return

This extension will fail

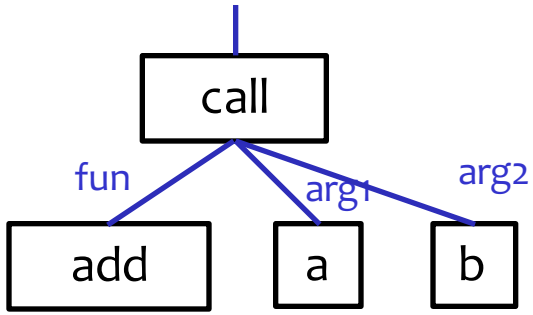
- motivating a non-recursive interpreter architecture

This failure should be unsurprising

- but will clarify the difference between PA2 and Sec 2 interpreters

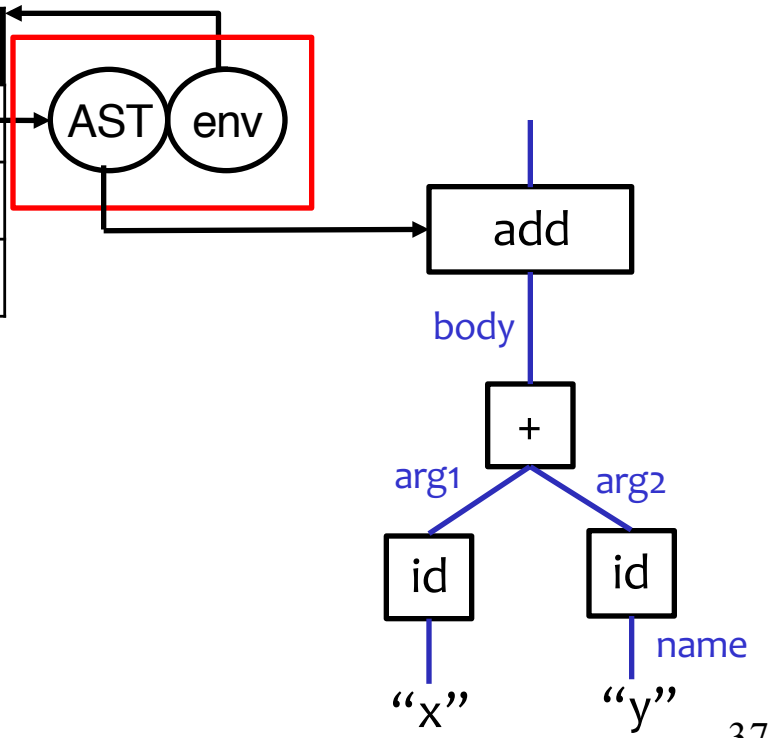
Example

```
function add(x,y) { x + y }
a = 1
b = 2
add(a, b)
```



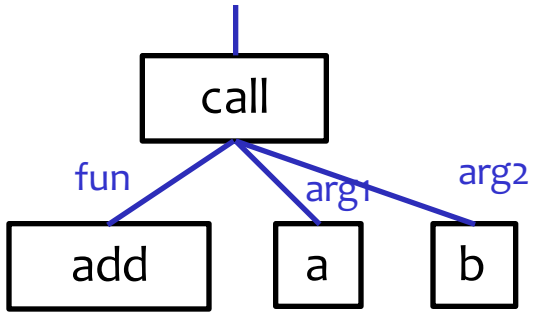

```
case 'call': // E(E,E)
  def fn = eval(node.fun)
  def t1 = eval(node.arg1)
  def t2 = eval(node.arg2)

  return call(fn.env, fn, t1, t2)
}
def call(p, fun, v1, v2) {
 f = Frame(p)
 f.push(fun.body.arg1, v1)
 f.push(fun.body.arg2, v2)
 eval(fun.body, f)
}
```

sym	value
add	
a	1
b	2



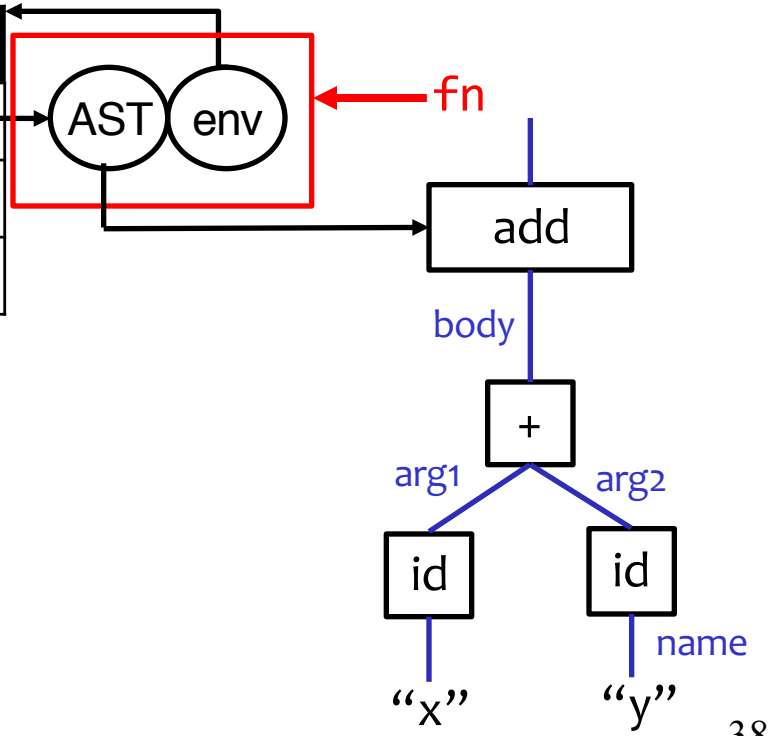
Example

```
function add(x,y) { x + y }
a = 1
b = 2
add(a, b)
```



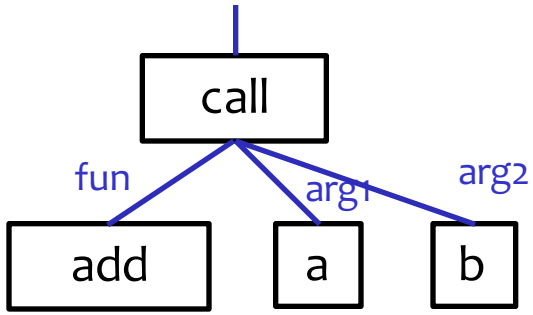

```
case 'call': // E(E,E)
  def fn = eval(node.fun)
  def t1 = eval(node.arg1)
  def t2 = eval(node.arg2)

  return call(fn.env, fn, t1, t2)
}
def call(p, fun, v1, v2) {
 f = Frame(p)
 f.push(fun.body.arg1, v1)
 f.push(fun.body.arg2, v2)
 eval(fun.body, f)
}
```

sym	value
add	
a	1
b	2



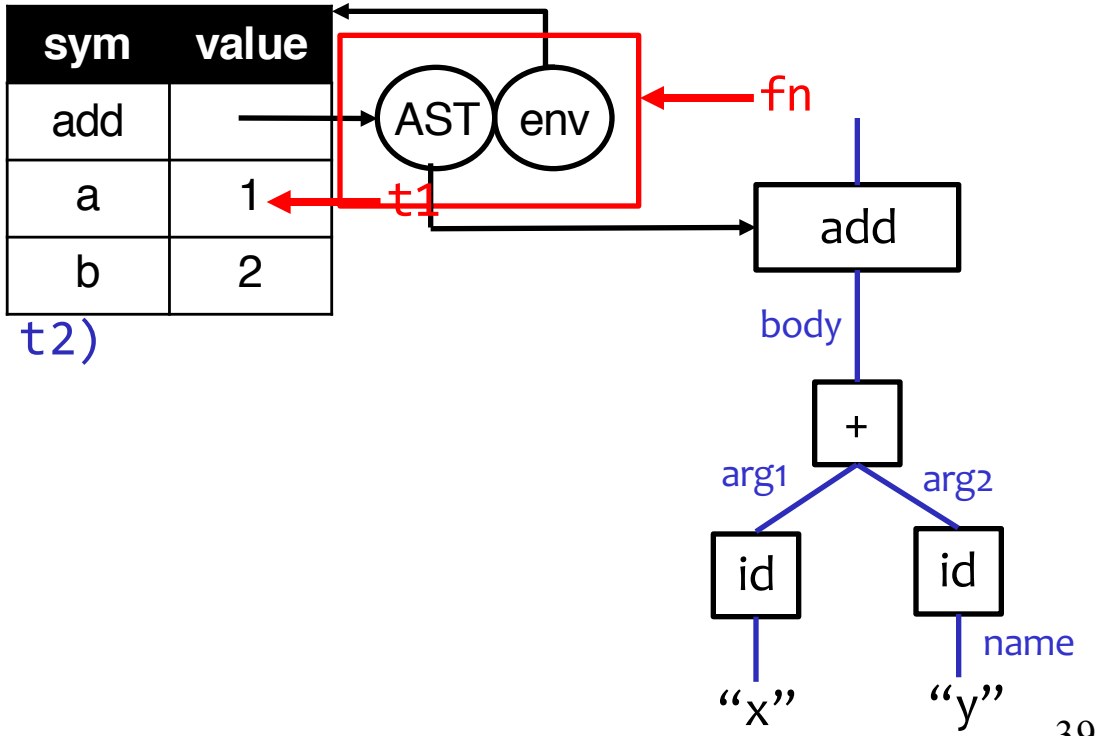
Example

```
function add(x,y) { x + y }
a = 1
b = 2
add(a, b)
```



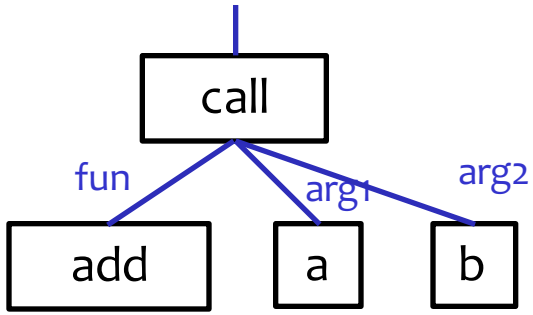

```
case 'call': // E(E,E)
  def fn = eval(node.fun)
  def t1 = eval(node.arg1)
  def t2 = eval(node.arg2)

  return call(fn.env, fn, t1, t2)
}
def call(p, fun, v1, v2) {
 f = Frame(p)
 f.push(fun.body.arg1, v1)
 f.push(fun.body.arg2, v2)
 eval(fun.body, f)
}
```

sym	value
add	
a	1
b	2

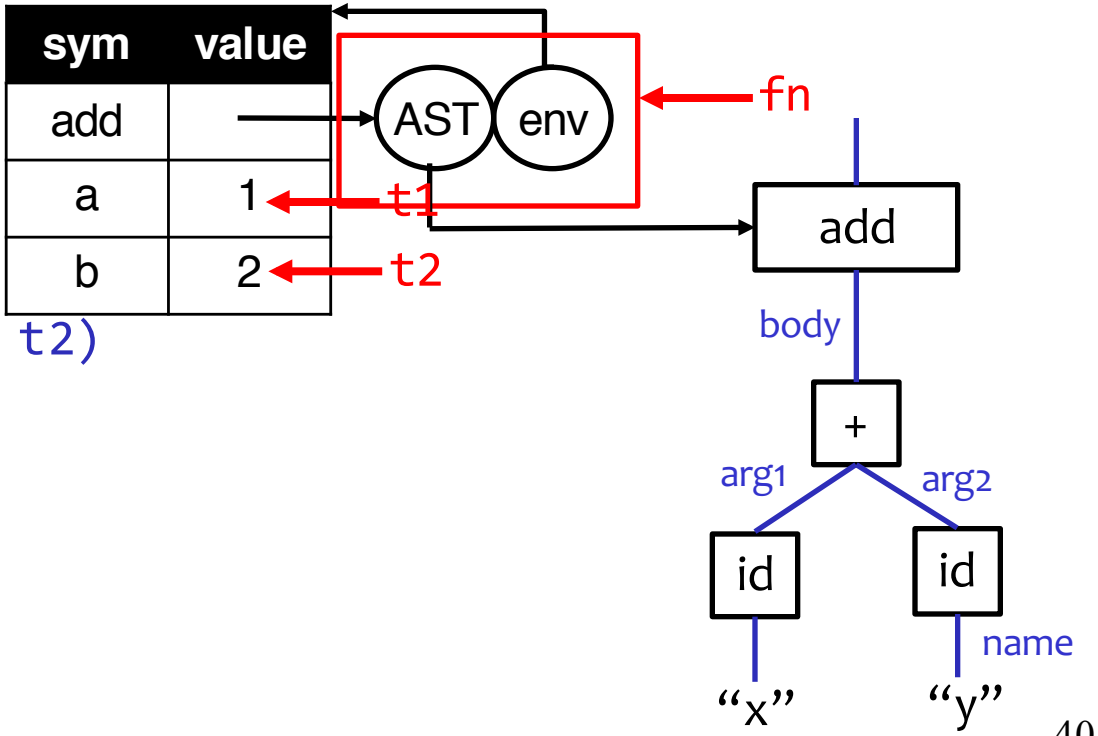


Example

```
function add(x,y) { x + y }
a = 1
b = 2
add(a, b)
```

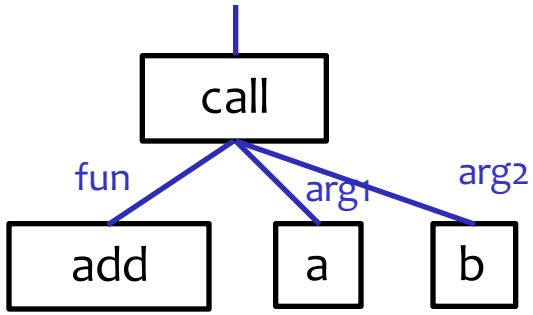



```
case 'call': // E(E,E)
  def fn = eval(node.fun)
  def t1 = eval(node.arg1)
  def t2 = eval(node.arg2)

  return call(fn.env, fn, t1, t2)
}
def call(p, fun, v1, v2) {
 f = Frame(p)
 f.push(fun.body.arg1, v1)
 f.push(fun.body.arg2, v2)
 eval(fun.body, f)
}
```

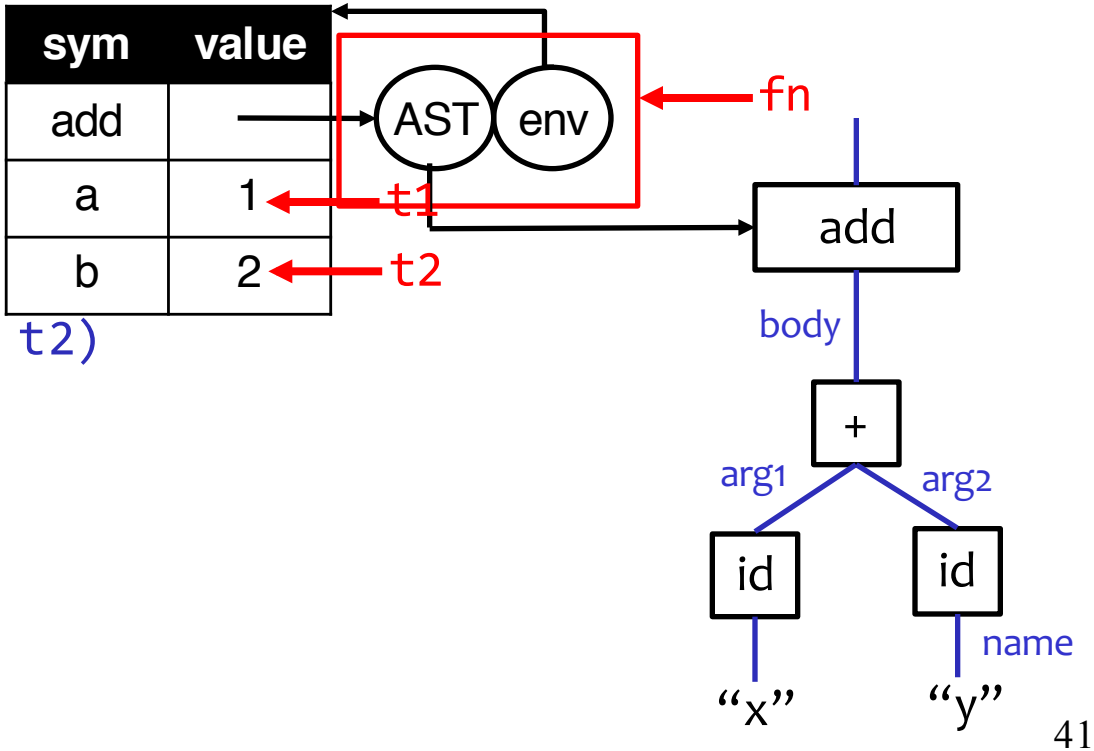


Example

```
function add(x,y) { x + y }
a = 1
b = 2
add(a, b)
```

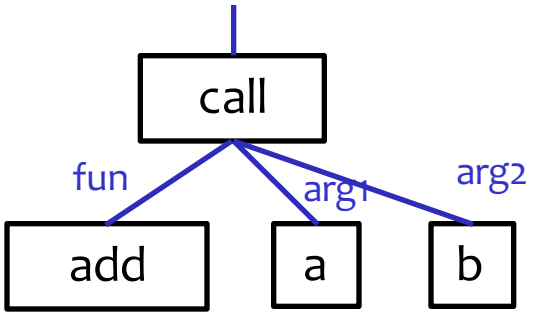



```
case 'call': // E(E,E)
  def fn = eval(node.fun)
  def t1 = eval(node.arg1)
  def t2 = eval(node.arg2)

  return call(fn.env, fn, t1, t2)
}
def call(p, fun, v1, v2) {
 f = Frame(p)
 f.push(fun.body.arg1, v1)
 f.push(fun.body.arg2, v2)
 eval(fun.body, f)
}
```

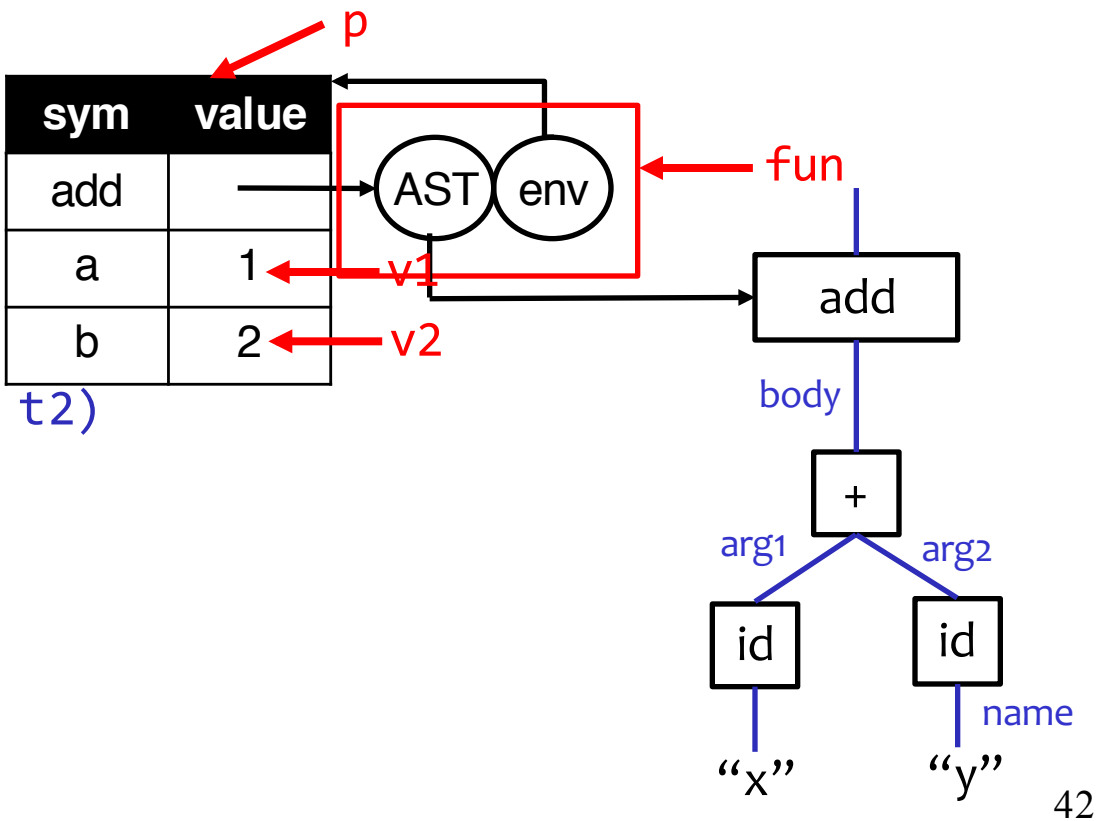


Example

```
function add(x,y) { x + y }
a = 1
b = 2
add(a, b)
```

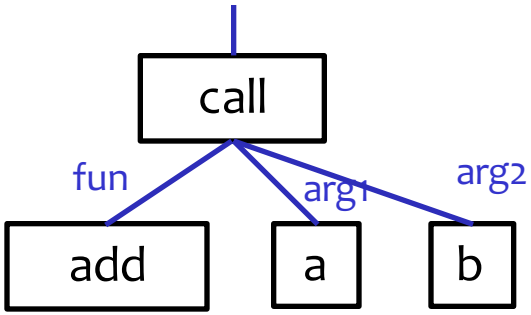



```
case 'call': // E(E,E)
  def fn = eval(node.fun)
  def t1 = eval(node.arg1)
  def t2 = eval(node.arg2)

  return call(fn.env, fn, t1, t2)
}
def call(p, fun, v1, v2) {
 f = Frame(p)
 f.push(fun.body.arg1, v1)
 f.push(fun.body.arg2, v2)
 eval(fun.body, f)
}
```

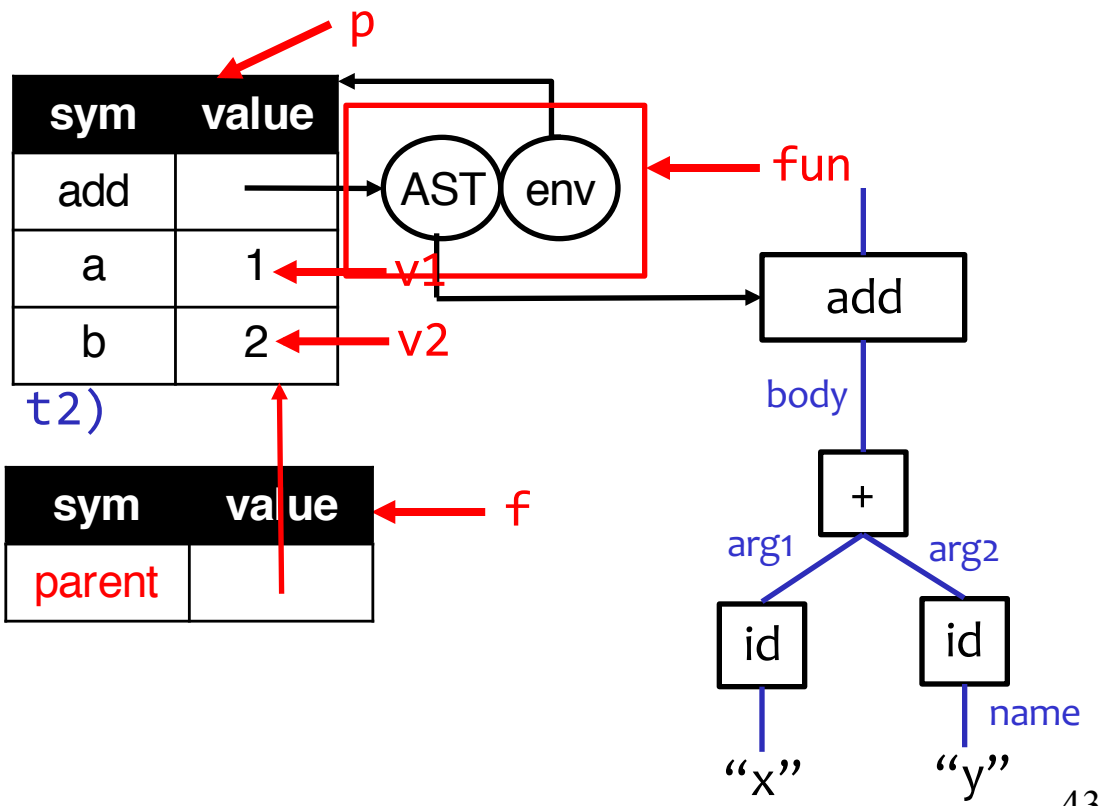


Example

```
function add(x,y) { x + y }
a = 1
b = 2
add(a, b)
```

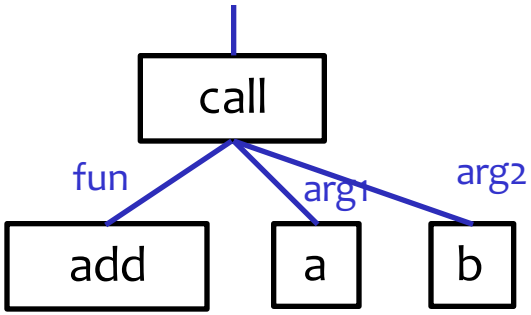



```
case 'call': // E(E,E)
  def fn = eval(node.fun)
  def t1 = eval(node.arg1)
  def t2 = eval(node.arg2)

  return call(fn.env, fn, t1, t2)
}
def call(p, fun, v1, v2) {
  f = Frame(p)
  f.push(fun.body.arg1, v1)
  f.push(fun.body.arg2, v2)
  eval(fun.body, f)
}
```

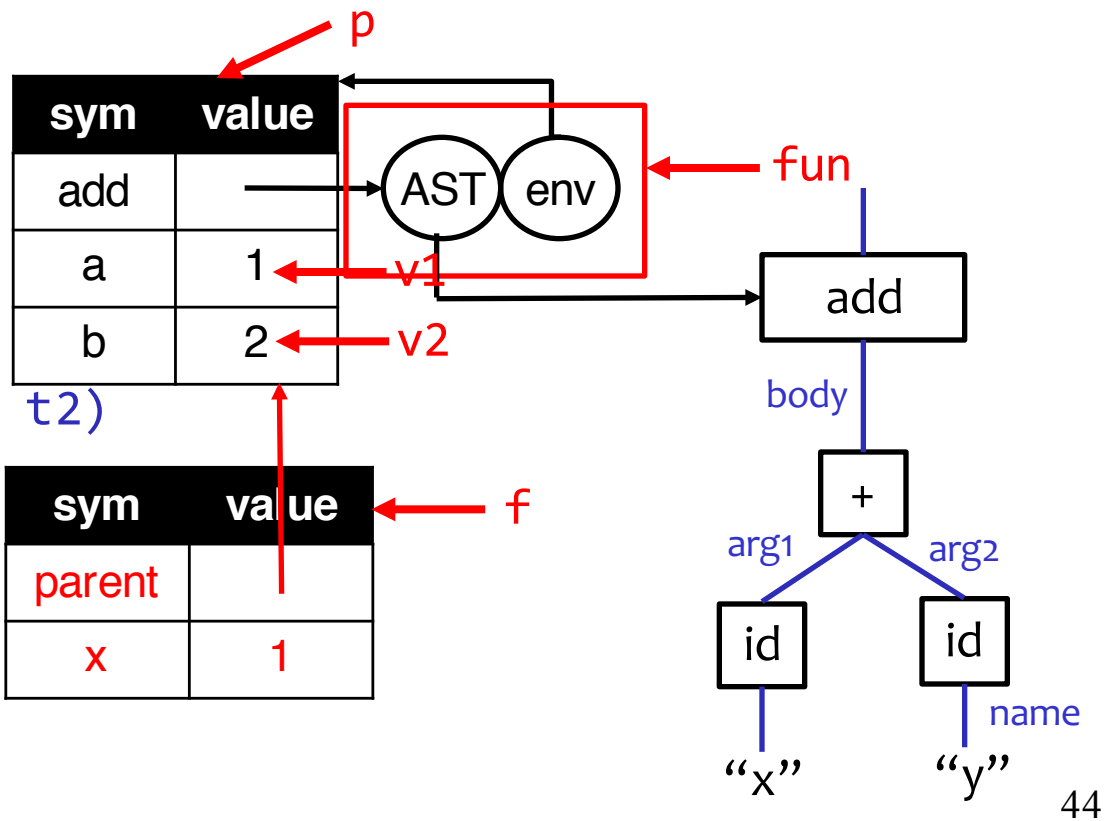


Example

```
function add(x,y) { x + y }
a = 1
b = 2
add(a, b)
```

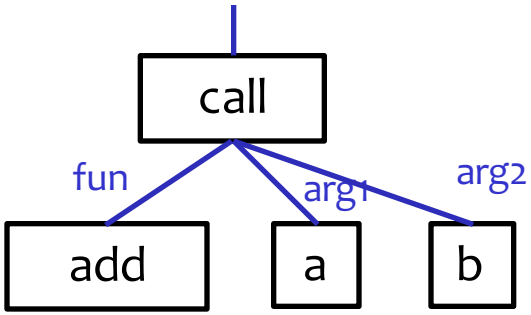



```
case 'call': // E(E,E)
  def fn = eval(node.fun)
  def t1 = eval(node.arg1)
  def t2 = eval(node.arg2)

  return call(fn.env, fn, t1, t2)
}
def call(p, fun, v1, v2) {
 f = Frame(p)
 f.push(fun.body.arg1, v1)
 f.push(fun.body.arg2, v2)
 eval(fun.body, f)
}
```

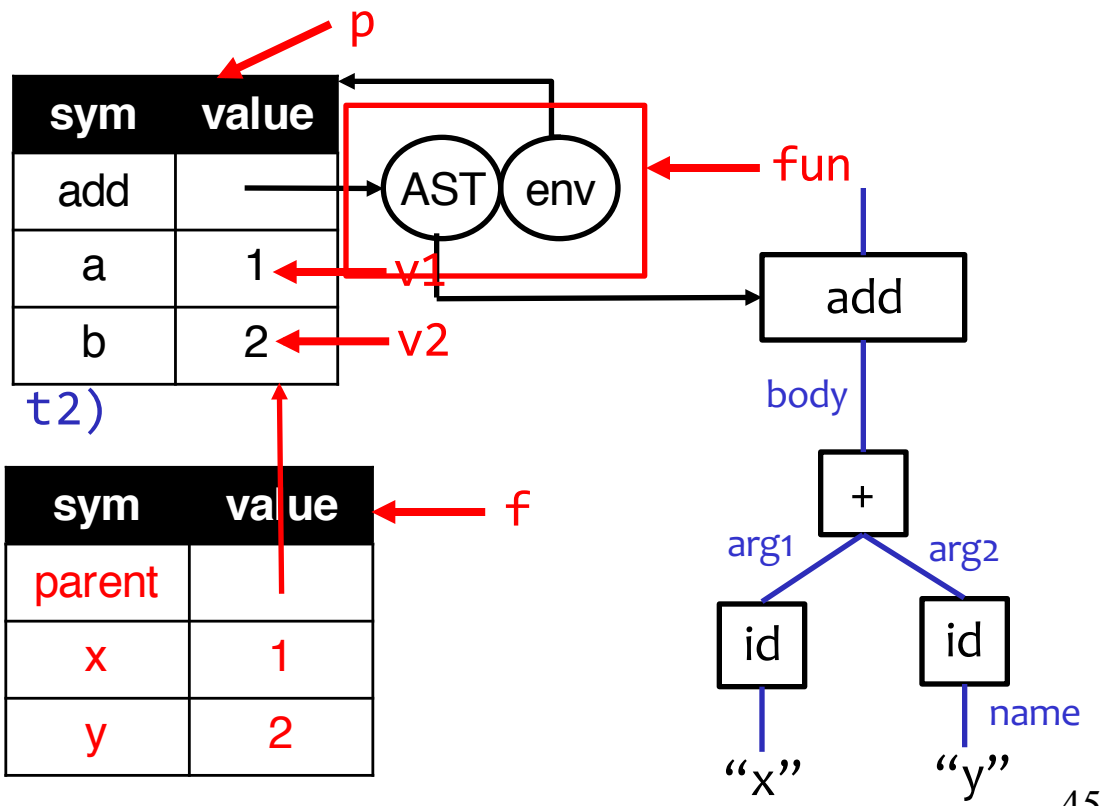


Example

```
function add(x,y) { x + y }
a = 1
b = 2
add(a, b)
```

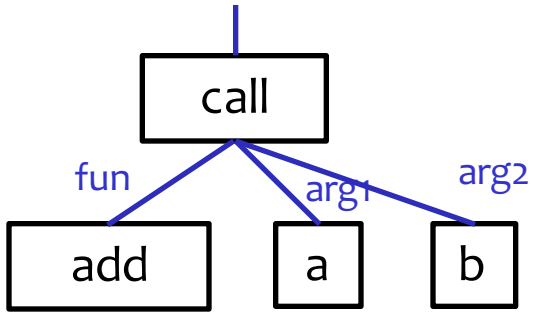



```
case 'call': // E(E,E)
  def fn = eval(node.fun)
  def t1 = eval(node.arg1)
  def t2 = eval(node.arg2)

  return call(fn.env, fn, t1, t2)
}
def call(p, fun, v1, v2) {
 f = Frame(p)
 f.push(fun.body.arg1, v1)
 f.push(fun.body.arg2, v2)
 eval(fun.body, f)
}
```

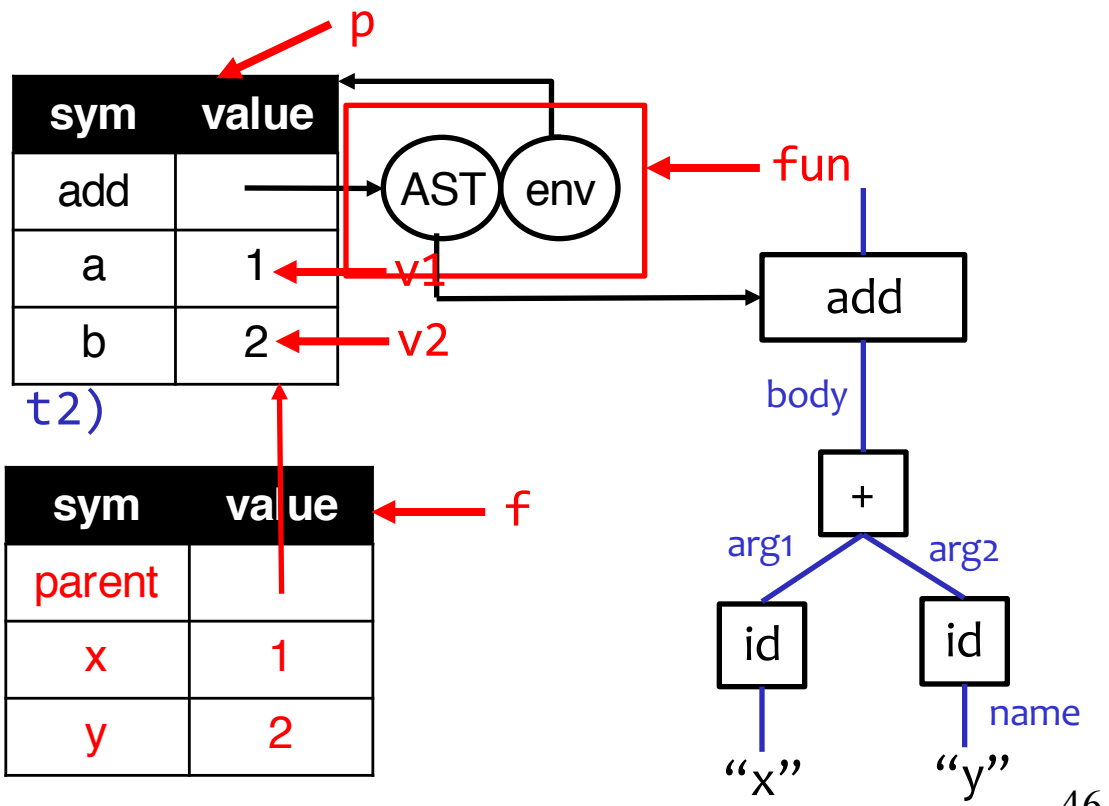


Example

```
function add(x,y) { x + y }
a = 1
b = 2
add(a, b)
```




```
case 'call': // E(E,E)
  def fn = eval(node.fun)
  def t1 = eval(node.arg1)
  def t2 = eval(node.arg2)

  return call(fn.env, fn, t1, t2)
}
def call(p, fun, v1, v2) {
 f = Frame(p)
 f.push(fun.body.arg1, v1)
 f.push(fun.body.arg2, v2)
 eval(fun.body, f)
}
```



The recursive Sec 2 interpreter

Program is represented as an AST

- evaluated by walking the AST bottom-up

State (context) of the interpreter:

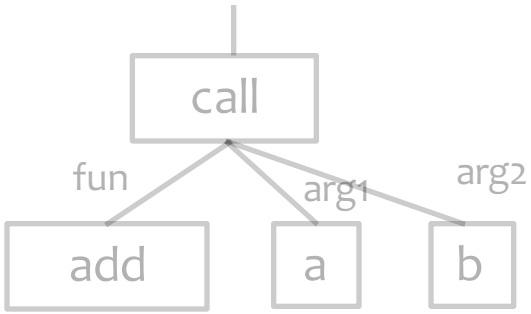
- control: what AST nodes to evaluate next (the "PC")
- data: intermediate values (arguments to operators, calls)

Where is this state maintained?

- control: on interpreter's calls stack (one PC per level)
- data: in interpreter's variables (eg, t1, t2)

Example

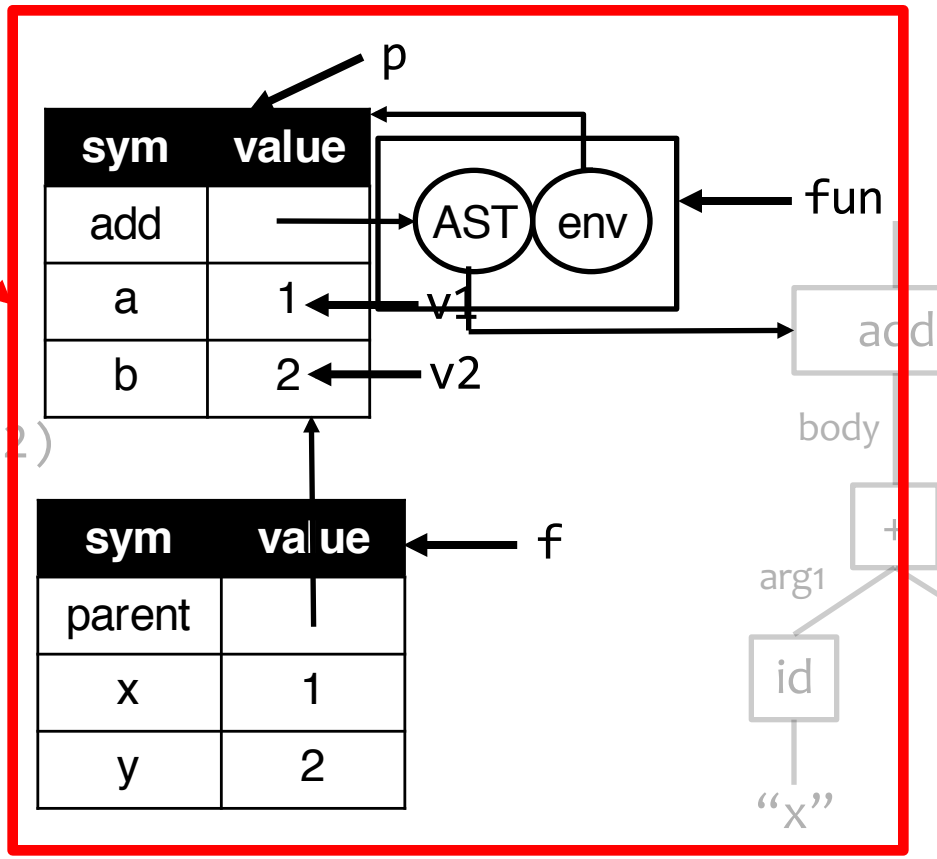
```
function add(x,y) { x + y }
a = 1
b = 2
add(a, b)
```



Call stack

Interpreter state

```
case 'call': // E(E,E)
  def fn = eval(node.fun)
  def t1 = eval(node.arg1)
  def t2 = eval(node.arg2)

  return call(env, fn, t1, t2)
}
def call(p, fun, v1, v2) {
  f = Frame(p)
  f.push(fun.body.arg1, v1)
  f.push(fun.body.arg2, v2)
  eval(fun.body, f)
}
```



An attempt for a recursive coro. interpreter

```
def eval(node) {
    switch (node.op) {
    case 'resume': // resume E, E
                   evaluate the expression that gives coroutine handle -- OK
                   def co = eval(n.arg1)
                   def t1 = eval(n.arg2)
                   now resume to c/r by evaluating its body
                   (resumes in the right point only on first resume to co)
                   call(co.body, t1)
    case 'yield': // yield E
                  return eval(node.arg)
                  oops: this returns to eval() in the same coroutine,
                  rather than to the resumer coroutine as was desired
    }
}
```

Draw call stacks at this point

```
def f(x,y) {
  cg=coroutine(g)
  resume(cg,x,y)
}

f(1,2)
```

```
def g(x,y) {
  ck=coroutine(k)
  def h(a,b) {
    3 + yield(resume(ck,a,b))
  }
  h(x,y)
}
```

```
def k(x,y) {
  yield(x+y)
}
```

call(env_f, f, 1, 2)
eval(env_1, resume)
resume(env_2, cg, 1, 2)

call(env_g, g, 1, 2)
call(env_h, h, 1, 2)
eval(env_2, add)
eval(env_2, yield)
resume(env_2, ck, 1, 2)

call(env_k, k, 1, 2)
eval(env_3, yield)

Draw call stacks at this point

```
def f(x,y) {
  cg=coroutine(g)
  resume(cg,x,y)
}

f(1,2)
```

```
def g(x,y) {
  ck=coroutine(k)
  def h(a,b) {
    3 + yield(resume(ck,a,b))
  }
  h(x,y)
}
```

```
def k(x,y) {
   yield(x+y)
}
```

Draw the calls stacks in the interpreter:

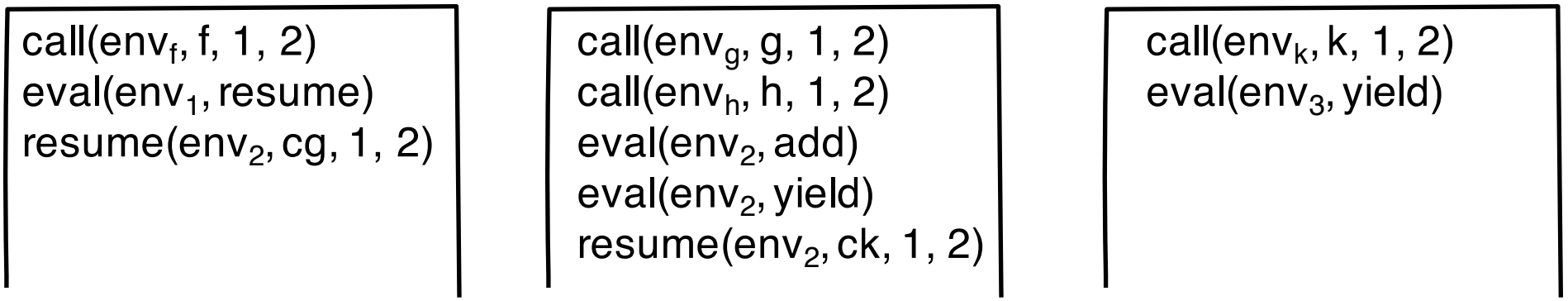
The interpreter's call stack contains the recursive eval() calls.

- the control context (what code to execute next)

The interpreter's environment has variables t1 and t2 (see prev slide).

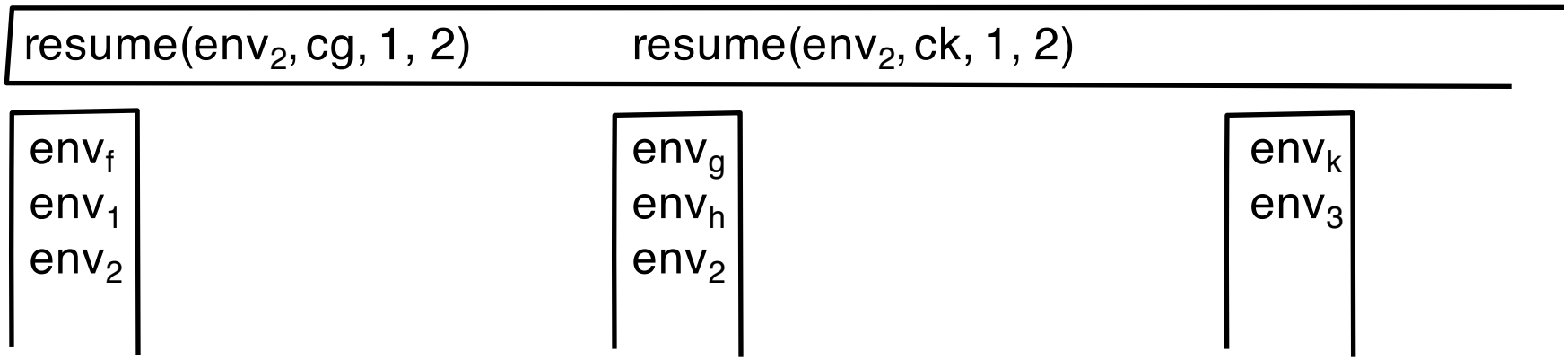
- the data context (values of intermediate values)

Draw call stacks at this point



recursive interpreter (can't yield without losing state)

bytecode interpreter (keeps state on the stack of each coroutine)



Summary

If it exits when yielding, it loses this context

- this exit would need to pop all the way from recursion

The interpreter for a coroutine c must return

- when c encounters a yield

The interpreter will be called again

- when some coroutine invokes resume(c, …)

Summary

A recursive, Sec 2-like interpreter cannot *yield*

- We need a separate call stack for each coroutine.
- To preserve the coroutine context while it is suspended.

Why multiple call stacks are impossible in PA2 interp?

- The interpreter can only use one call stack – because the host language (JS) does not have threads or coroutines.

Intermission

The real deal

Must re-architect the interpreter

We need to create a non-recursive interpreter

- non-recursive ==> yield can become a return to resumer

all the context stored in separate data structures

- explicit stack data structure is OK

The plan

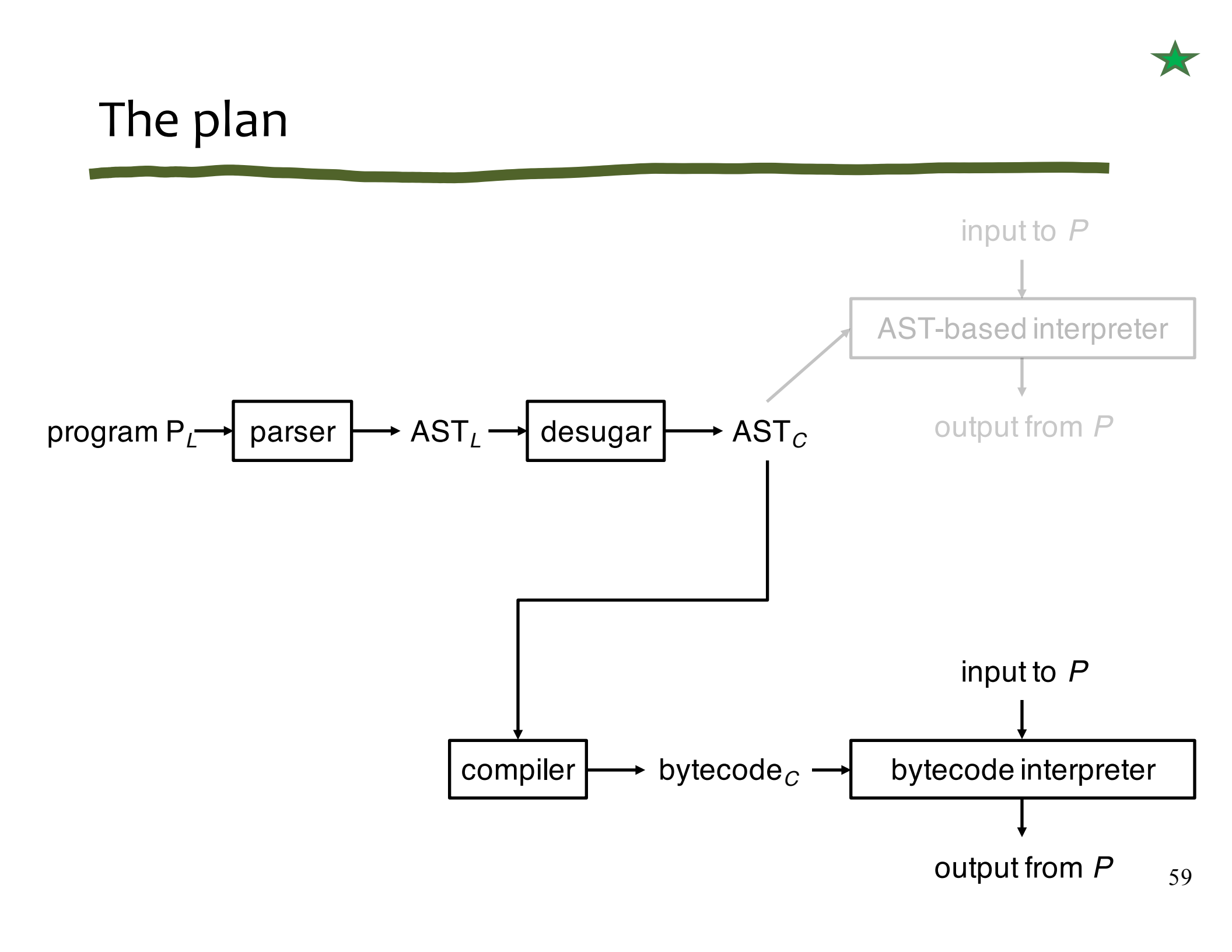
Linearize the AST so that the control context can be kept as a program counter (PC)

- a stack of these PCs will be used to implement call/return

Store intermediate values in program's variables

- in PA2, they were in the interpreter's variables

These two goals are achieved by translating the AST into register-based bytecode



The plan

input to *P*

AST-based interpreter

output from *P*

program P_L → parser → AST_L → desugar → AST_C

compiler → $bytecode_C$ → bytecode interpreter

input to *P*

output from *P*

AST-to-bytecode compiler

AST vs. bytecode

Abstract Syntax Tree:

values of sub-expressions do not have explicit names

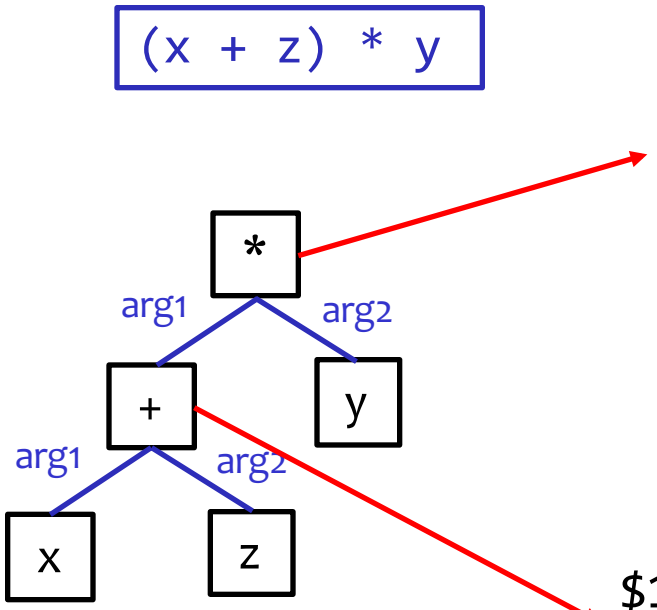
Bytecode:

- list of instructions of the form x = y + z
- x, y, z can be temporary variables invented by compiler
- temporaries store values that would be in the variables of a recursive interpreter

Example: AST `(x+z)*y` translates to bytecode:

```
$1 = x+z    // $1, $2: temp vars
$2 = $1*y   // return value of AST is in $2
```

Example




```
$1 = x+z    // $1, $2: temp vars
$2 = $1*y   // return value of AST is in $2
```

```
$1 = x+z
// $1: temp var
```

Compile AST to bytecode

Traverse AST *a*, emitting code *c* rather than evaluating

when the generated code c is later executed,
it evaluates the original AST *a*

We'll write a translation function **b** with signature:

b : AST -> (Code, RegName)

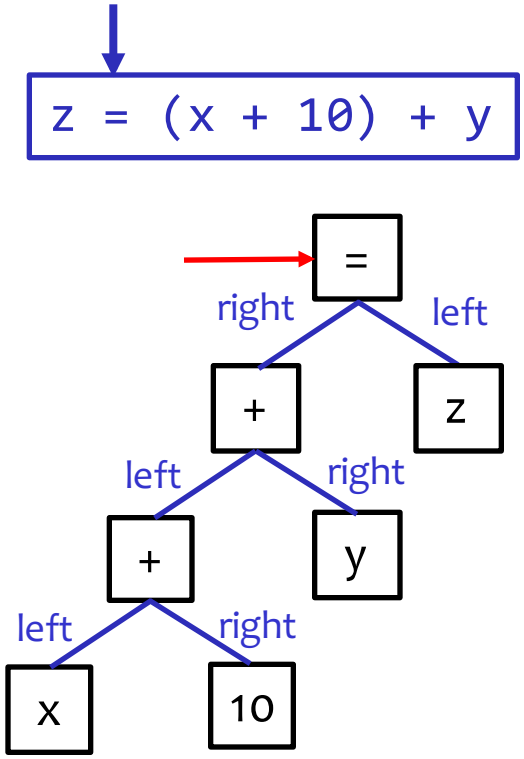
The function produces code c and register name r s.t.:

when c is executed, the value of AST is stored in register r

Compile AST to bytecode (cont)

```
def b(n):
        switch n.op:
        +:      (c1,r1) = b(n.left)
                (c2,r2) = b(n.right)
                def r = freshReg()
                def instr = "%s = %s + %s\n" % (r, r1, r2)
                return (c1 + c2 + instr, r)
        n:      integer literal (allows compiling 2+3+4)
                def r = freshReg()
                return ("%s = %s\n" % (r, n.val), r)
        ID:     variable name (allows compiling x+3+y)
                return ("", n.name)
        =:      ID = E (allows compiling x=1+3+x)
                (c,r) = b(n.right)
                return (c + "%s=%s\n" % (n.left.name, r),
                                                n.left.name)
```

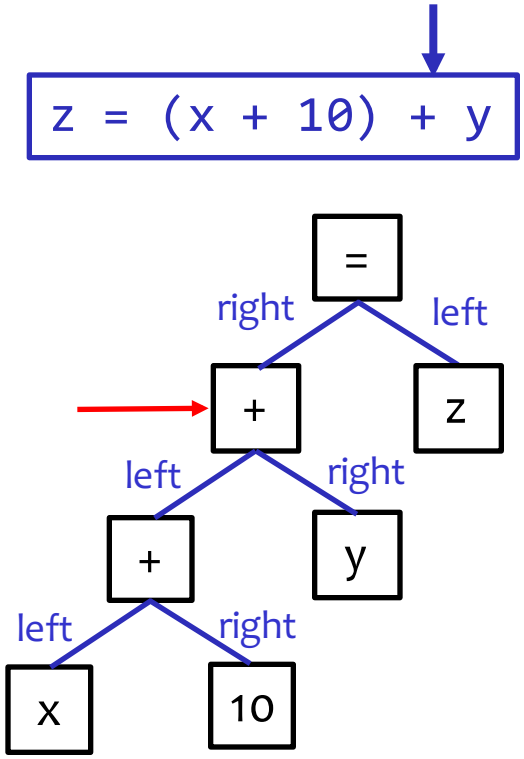
Example




```
def b(n):
        switch n.op:
        +:      (c1,r1) = b(n.left)
                (c2,r2) = b(n.right)
                def r = freshReg()
                def instr = "%s=%s+%s\n" % (r, r1, r2)
                return (c1 + c2 + instr, r)
        n:      // integer literal (allows compiling 2+3+4)
                def r = freshReg()
                return ("%s=%s\n" % (r, n.val), r)
        ID:     // variable name (allows compiling x+3+y)
                return ("", n.name)
        =:      // ID = E (allows compiling x=1+3+x)
                (c,r) = b(n.right)
                return (c + "%s=%s\n" %
                        (n.left.name, r), n.left.name)
```

variable	value

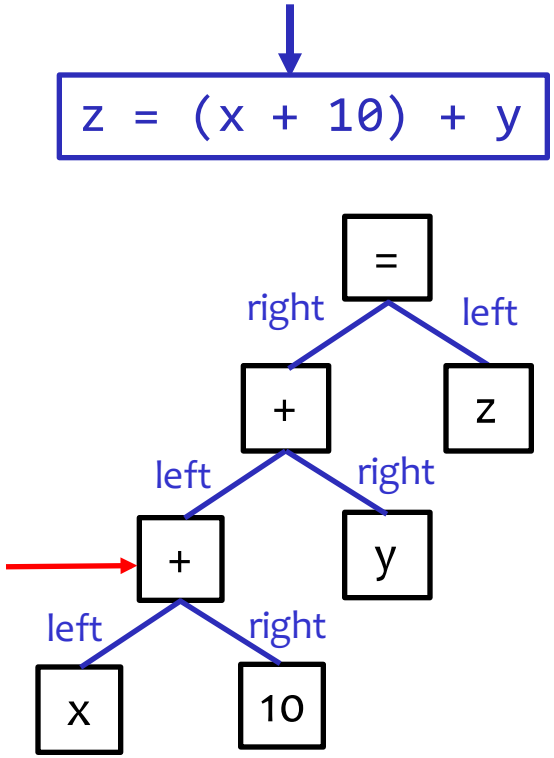
Example




```
def b(n):
        switch n.op:
        +:      (c1,r1) = b(n.left)
                (c2,r2) = b(n.right)
                def r = freshReg()
                def instr = "%s=%s+%s\n" % (r, r1, r2)
                return (c1 + c2 + instr, r)
        n:      // integer literal (allows compiling 2+3+4)
                def r = freshReg()
                return ("%s=%s\n" % (r, n.val), r)
        ID:     // variable name (allows compiling x+3+y)
                return ("", n.name)
        =:      // ID = E (allows compiling x=1+3+x)
                (c,r) = b(n.right)
               return (c + "%s=%s\n" %
                        (n.left.name, r), n.left.name)
```

variable	value

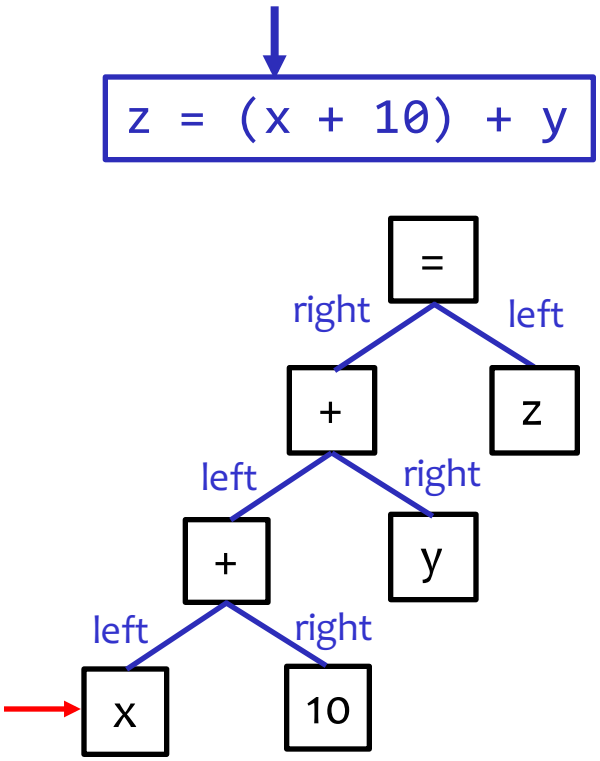
Example




```
def b(n):
        switch n.op:
        +:      (c1,r1) = b(n.left)
                (c2,r2) = b(n.right)
                def r = freshReg()
                def instr = "%s=%s+%s\n" % (r, r1, r2)
                return (c1 + c2 + instr, r)
        n:      // integer literal (allows compiling 2+3+4)
                def r = freshReg()
                return ("%s=%s\n" % (r, n.val), r)
        ID:     // variable name (allows compiling x+3+y)
                return ("", n.name)
        =:      // ID = E (allows compiling x=1+3+x)
                (c,r) = b(n.right)
               return (c + "%s=%s\n" %
                        (n.left.name, r), n.left.name)
```

variable	value

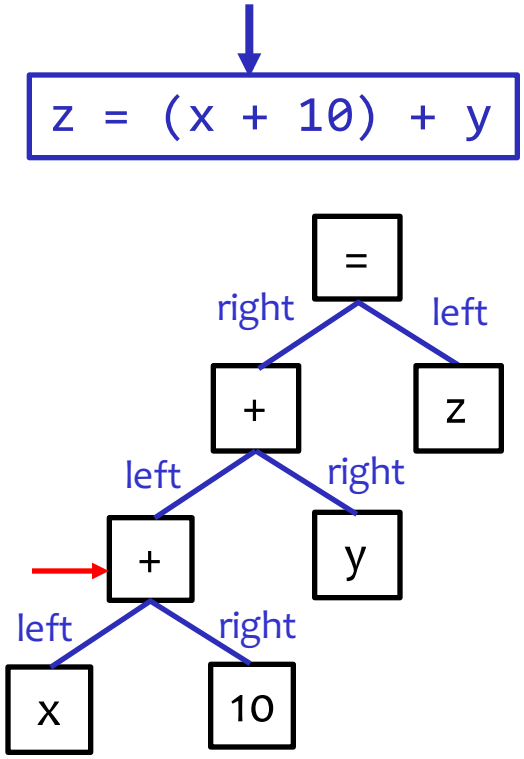
Example




```
def b(n):
        switch n.op:
        +:      (c1,r1) = b(n.left)
                (c2,r2) = b(n.right)
                def r = freshReg()
                def instr = "%s=%s+%s\n" % (r, r1, r2)
                return (c1 + c2 + instr, r)
        n:      // integer literal (allows compiling 2+3+4)
                def r = freshReg()
                return ("%s=%s\n" % (r, n.val), r)
        ID:     // variable name (allows compiling x+3+y)
                return ("", n.name)
        =:      // ID = E (allows compiling x=1+3+x)
                (c,r) = b(n.right)
                return (c + "%s=%s\n" %
                        (n.left.name, r), n.left.name)
```

variable	value
return value	("", x)

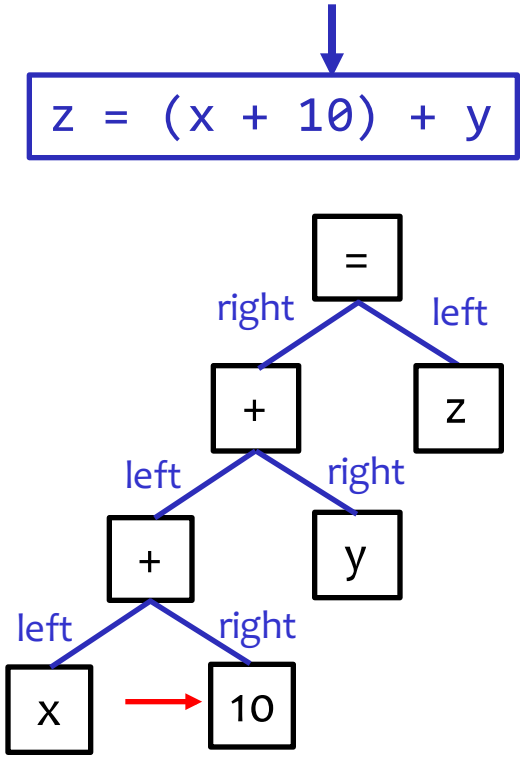
Example




```
def b(n):
        switch n.op:
        +:      (c1,r1) = b(n.left)
                (c2,r2) = b(n.right)
                def r = freshReg()
                def instr = "%s=%s+%s\n" % (r, r1, r2)
                return (c1 + c2 + instr, r)
        n:      // integer literal (allows compiling 2+3+4)
                def r = freshReg()
                return ("%s=%s\n" % (r, n.val), r)
        ID:     // variable name (allows compiling x+3+y)
                return ("", n.name)
        =:      // ID = E (allows compiling x=1+3+x)
                (c,r) = b(n.right)
                return (c + "%s=%s\n" %
                        (n.left.name, r), n.left.name)
```

variable	value
(c1, r1)	("", x)

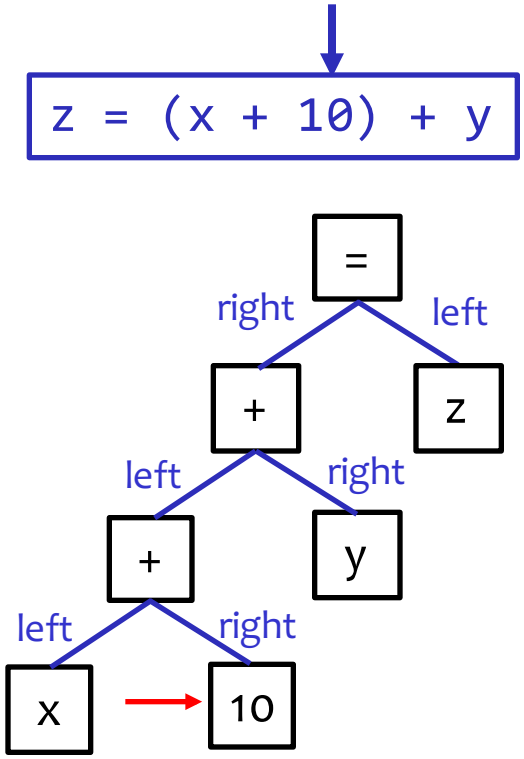
Example




```
def b(n):
        switch n.op:
        +:      (c1,r1) = b(n.left)
                (c2,r2) = b(n.right)
                def r = freshReg()
                def instr = "%s=%s+%s\n" % (r, r1, r2)
                return (c1 + c2 + instr, r)
        n:      // integer literal (allows compiling 2+3+4)
                def r = freshReg()
                return ("%s=%s\n" % (r, n.val), r)
        ID:     // variable name (allows compiling x+3+y)
                return ("", n.name)
        =:      // ID = E (allows compiling x=1+3+x)
                (c,r) = b(n.right)
                return (c + "%s=%s\n" %
                        (n.left.name, r), n.left.name)
```

variable	value
r	$1

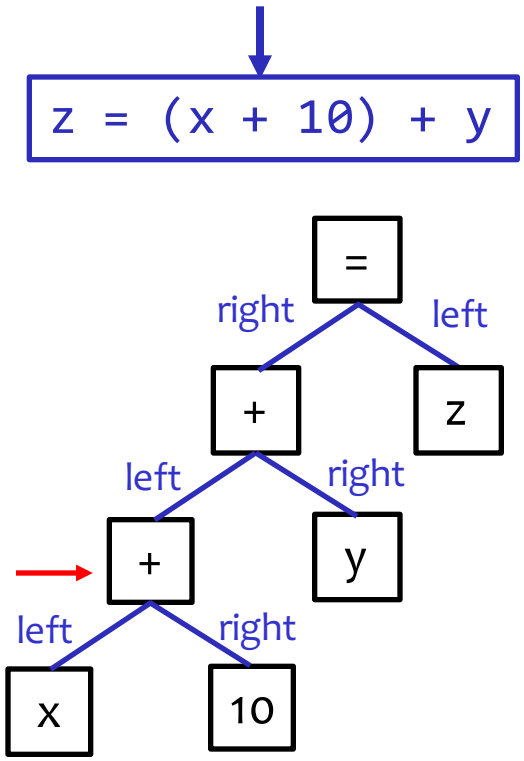
Example




```
def b(n):
        switch n.op:
        +:      (c1,r1) = b(n.left)
                (c2,r2) = b(n.right)
                def r = freshReg()
                def instr = "%s=%s+%s\n" % (r, r1, r2)
                return (c1 + c2 + instr, r)
        n:      // integer literal (allows compiling 2+3+4)
                def r = freshReg()
                return ("%s=%s\n" % (r, n.val), r)
        ID:     // variable name (allows compiling x+3+y)
                return ("", n.name)
        =:      // ID = E (allows compiling x=1+3+x)
                (c,r) = b(n.right)
                return (c + "%s=%s\n" %
                        (n.left.name, r), n.left.name)
```

variable	value
r	$1
return value	($1=10, $1)

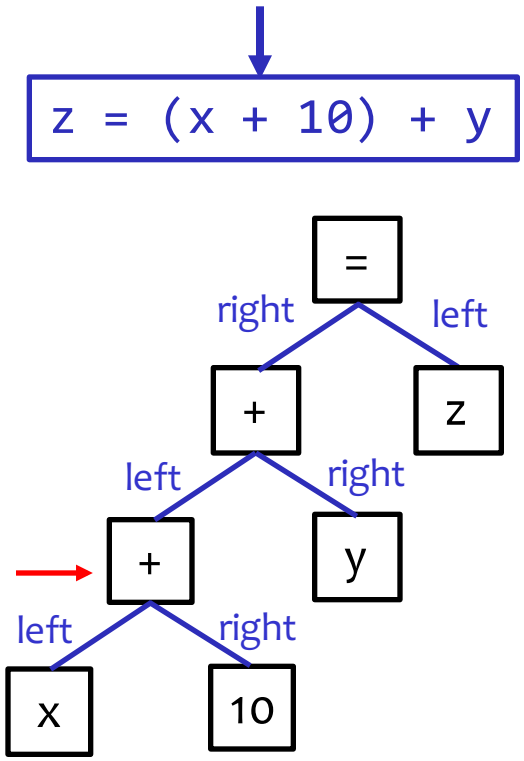
Example




```
def b(n):
        switch n.op:
        +:      (c1,r1) = b(n.left)
                (c2,r2) = b(n.right)
                def r = freshReg()
                def instr = "%s=%s+%s\n" % (r, r1, r2)
                return (c1 + c2 + instr, r)
        n:      // integer literal (allows compiling 2+3+4)
                def r = freshReg()
                return ("%s=%s\n" % (r, n.val), r)
        ID:     // variable name (allows compiling x+3+y)
                return ("", n.name)
        =:      // ID = E (allows compiling x=1+3+x)
                (c,r) = b(n.right)
                return (c + "%s=%s\n" %
                        (n.left.name, r), n.left.name)
```

variable	value
(c1, r1)	("", x)
(c2, r2)	($1=10, $1)
r	$2

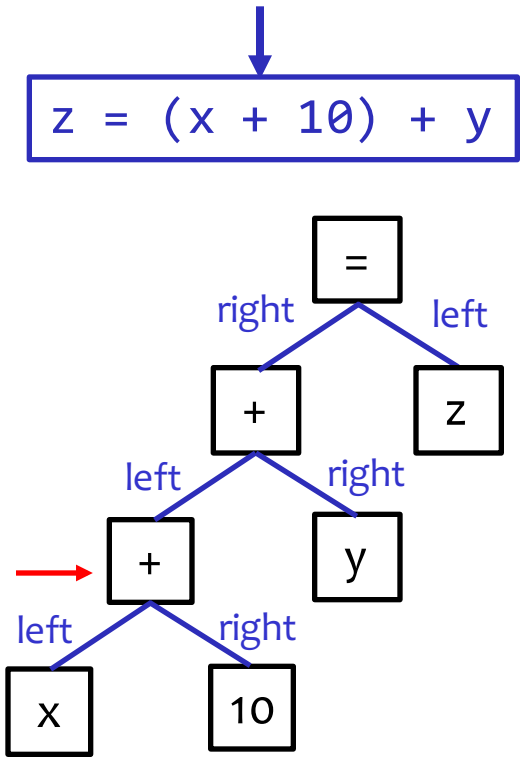
Example




```
def b(n):
        switch n.op:
        +:      (c1,r1) = b(n.left)
                (c2,r2) = b(n.right)
                def r = freshReg()
                def instr = "%s=%s+%s\n" % (r, r1, r2)
                return (c1 + c2 + instr, r)
        n:      // integer literal (allows compiling 2+3+4)
                def r = freshReg()
                return ("%s=%s\n" % (r, n.val), r)
        ID:     // variable name (allows compiling x+3+y)
                return ("", n.name)
        =:      // ID = E (allows compiling x=1+3+x)
                (c,r) = b(n.right)
                return (c + "%s=%s\n" %
                        (n.left.name, r), n.left.name)
```

variable	value
(c1, r1)	("", x)
(c2, r2)	($1=10, $1)
r	$2
instr	$2=x+$1

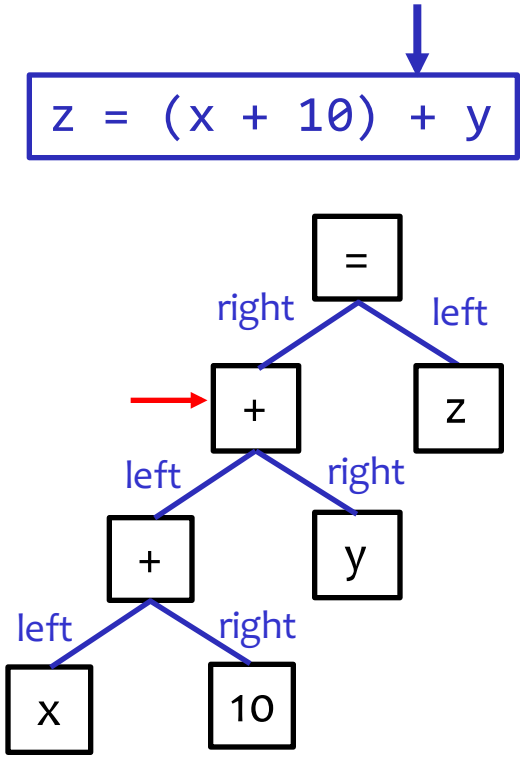
Example




```
def b(n):
        switch n.op:
+:          (c1,r1) = b(n.left)
            (c2,r2) = b(n.right)
            def r = freshReg()
            def instr = "%s=%s+%s\n" % (r, r1, r2)
            return (c1 + c2 + instr, r)
n:          // integer literal (allows compiling 2+3+4)
            def r = freshReg()
            return ("%s=%s\n" % (r, n.val), r)
ID:         // variable name (allows compiling x+3+y)
            return ("", n.name)
=:          // ID = E (allows compiling x=1+3+x)
            (c,r) = b(n.right)
           return (c + "%s=%s\n" %
                    (n.left.name, r), n.left.name)
```

variable	value
(c1, r1)	("", x)
(c2, r2)	($1=10, $1)
r	$2
instr	$2=x+$1
return value	($1=10 $2=x+$1, $2)

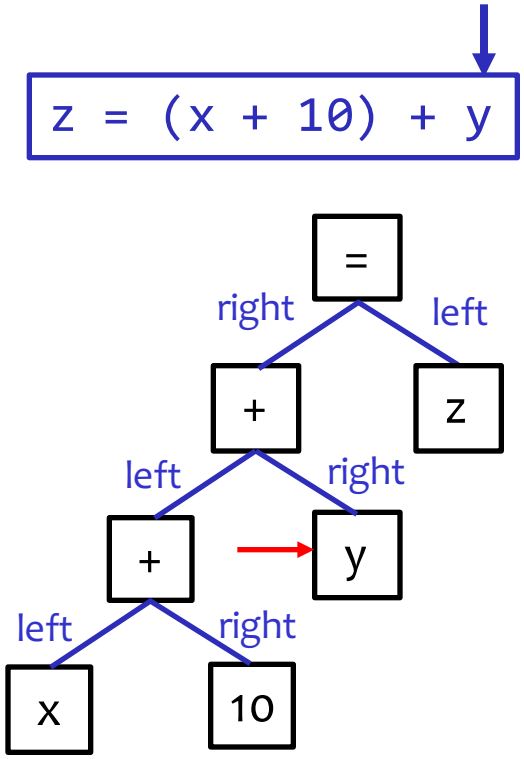
Example




```
def b(n):
        switch n.op:
        +:      (c1,r1) = b(n.left)
                (c2,r2) = b(n.right)
                def r = freshReg()
                def instr = "%s=%s+%s\n" % (r, r1, r2)
                return (c1 + c2 + instr, r)
        n:      // integer literal (allows compiling 2+3+4)
                def r = freshReg()
                return ("%s=%s\n" % (r, n.val), r)
        ID:     // variable name (allows compiling x+3+y)
                return ("", n.name)
        =:      // ID = E (allows compiling x=1+3+x)
                (c,r) = b(n.right)
                return (c + "%s=%s\n" %
                        (n.left.name, r), n.left.name)
```

variable	value
(c1, r1)	($1=10 $2=x+$1, $2)

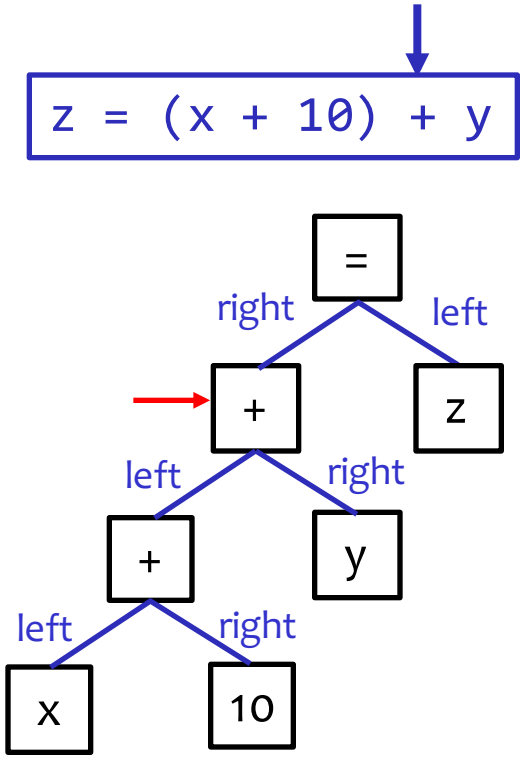
Example




```
def b(n):
        switch n.op:
        +:      (c1,r1) = b(n.left)
                (c2,r2) = b(n.right)
                def r = freshReg()
                def instr = "%s=%s+%s\n" % (r, r1, r2)
                return (c1 + c2 + instr, r)
        n:      // integer literal (allows compiling 2+3+4)
                def r = freshReg()
                return ("%s=%s\n" % (r, n.val), r)
        ID:     // variable name (allows compiling x+3+y)
                return ("", n.name)
        =:      // ID = E (allows compiling x=1+3+x)
                (c,r) = b(n.right)
                return (c + "%s=%s\n" %
                        (n.left.name, r), n.left.name)
```

variable	value
return value	("", y)

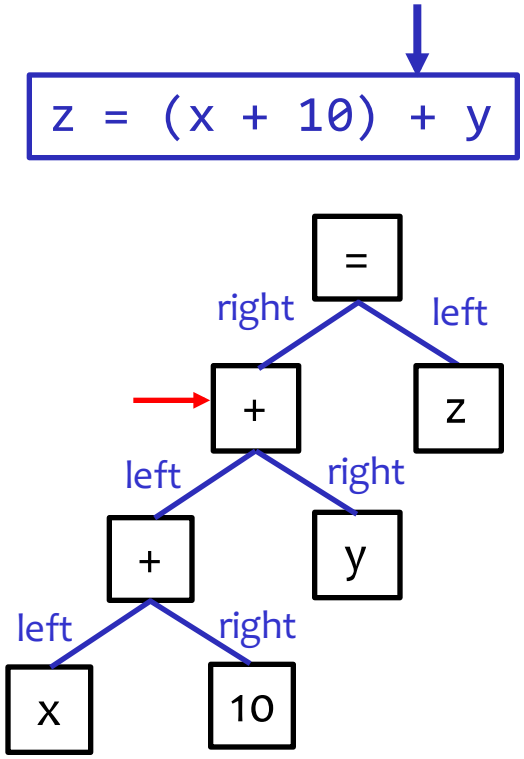
Example




```
def b(n):
        switch n.op:
        +:      (c1,r1) = b(n.left)
                (c2,r2) = b(n.right)
                def r = freshReg()
                def instr = "%s=%s+%s\n" % (r, r1, r2)
                return (c1 + c2 + instr, r)
        n:      // integer literal (allows compiling 2+3+4)
                def r = freshReg()
                return ("%s=%s\n" % (r, n.val), r)
        ID:     // variable name (allows compiling x+3+y)
                return ("", n.name)
        =:      // ID = E (allows compiling x=1+3+x)
                (c,r) = b(n.right)
                return (c + "%s=%s\n" %
                        (n.left.name, r), n.left.name)
```

variable	value
(c1, r1)	($1=10 $2=x+$1, $2)
(c2, r2)	("", y)
r	$3

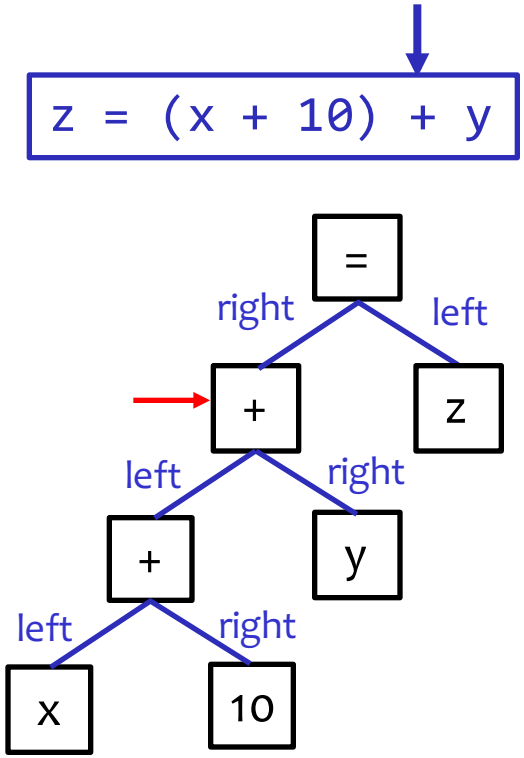
Example




```
def b(n):
        switch n.op:
        +:      (c1,r1) = b(n.left)
                (c2,r2) = b(n.right)
                def r = freshReg()
                def instr = "%s=%s+%s\n" % (r, r1, r2)
                return (c1 + c2 + instr, r)
        n:      // integer literal (allows compiling 2+3+4)
                def r = freshReg()
                return ("%s=%s\n" % (r, n.val), r)
        ID:     // variable name (allows compiling x+3+y)
                return ("", n.name)
        =:      // ID = E (allows compiling x=1+3+x)
                (c,r) = b(n.right)
                return (c + "%s=%s\n" %
                        (n.left.name, r), n.left.name)
```

variable	value
(c1, r1)	($1=10 $2=x+$1, $2)
(c2, r2)	("", y)
r	$3
instr	$3=$2+y

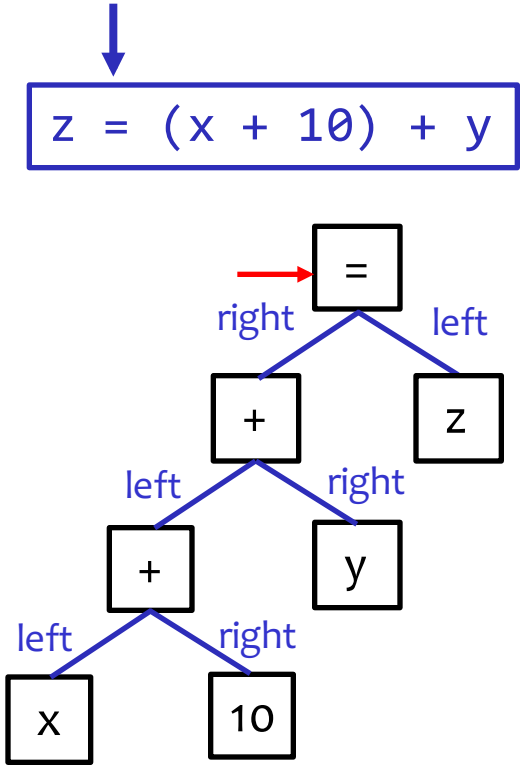
Example




```
def b(n):
        switch n.op:
        +:      (c1,r1) = b(n.left)
                (c2,r2) = b(n.right)
                def r = freshReg()
                def instr = "%s=%s+%s\n" % (r, r1, r2)
                return (c1 + c2 + instr, r)
        n:      // integer literal (allows compiling 2+3+4)
                def r = freshReg()
                return ("%s=%s\n" % (r, n.val), r)
        ID:     // variable name (allows compiling x+3+y)
                return ("", n.name)
        =:      // ID = E (allows compiling x=1+3+x)
                (c,r) = b(n.right)
                return (c + "%s=%s\n" %
                        (n.left.name, r), n.left.name)
```

variable	value
(c1, r1)	($1=10 $2=x+$1, $2)
(c2, r2)	("", y)
r	$3
instr	$3=$2+y
return value	$1=10 ($2=x+$1 $3=$2+y , $3)

Example




```
def b(n):
        switch n.op:
        +:      (c1,r1) = b(n.left)
                (c2,r2) = b(n.right)
                def r = freshReg()
                def instr = "%s=%s+%s\n" % (r, r1, r2)
                return (c1 + c2 + instr, r)
        n:      // integer literal (allows compiling 2+3+4)
                def r = freshReg()
                return ("%s=%s\n" % (r, n.val), r)
        ID:     // variable name (allows compiling x+3+y)
                return ("", n.name)
        =:      // ID = E (allows compiling x=1+3+x)
                (c,r) = b(n.right)
                return (c + "%s=%s\n" %
                        (n.left.name, r), n.left.name)
```

variable	value
(c, r)	$1=10 ($2=x+$1 $3=$2+y , $3)
return value	$1=10 ($2=x+$1 , z) $3=$2+y z=$3

Summary (1)

Bytecode compiler: two tasks

- translates AST into flat code (aka straight-line code)
- benefit: interpreter does not need to remember which part of the tree to evaluate next

Our bytecode is a "register-based bytecode"

- it stores intermediate values in registers
- these are virtual, not machine registers
- we implement them as local variables

There is also *stack-based* bytecode

- intermediate values pushed and popped onto *data stack*

Summary (2)

Bytecode translation recursively linearizes the AST

- the sub-translation picks a fresh temp variable and returns its name along with the generated code
- alternatively, caller tells the sub-translations in which temp variable the generated code must store its result

Translation to machine code not very different

- Sometimes done several nodes at a time, to find an instruction that executes both nodes

Bytecode interpreter

Bytecode interpreter

Data (ie, environments for lexical scoping):

same as in PA2

Control:

coroutine: each c/r is a separate interpreter

- these (resumable) interpreters share data environments
- each has own control context (call stack + next PC)

more on control in two slides …

first, let's examine the typical b/c interpreter loop

it's a loop now, not a recursive call

The bytecode interpreter loop

Note: this loop does not yet support resume/yield

```
def eval(bytecode_array) { // program to be executed
    pc = 0
    while (true) {
            curr_instr = bytecode_array[pc]
            switch (curr_instr.op) {
            case '+': …     // do + on args (as in PA2)
                      pc++  // jump to next bc instr
            …
            }
    }
}
```

Control, continued

call:

push PC to call stack

set PC to the start of the target function

return:

pops the PC from call stack

sets PC to it

Control, continued

yield *v*:

return from interpreter

passing *v* to resume

- PC after yield remains stored in c/r control context

resume *c*, *v*:

restart the interpreter of c

pass v to c

Discussion

Not quite a non-recursive interpreter

- while calls are handled with the interpreter loop, each **`resume`** recursively calls an interpreter instance

What do we need to store in the c/r handle?

- a pointer to an interpreter instance, which stores
- the call stack
- the next PC

Summary (1)

Control context:

- index into a *bytecode array*

Call / return: do not recursively create a sub-interpeter

- call: add the return address to the call stack
- return: jump to return address popped from call stack

Summary (2)

Create a coroutine: create a sub-interpreter

- initialize its control context and environment
- so that first resume start evaluating the coroutine
- the coroutine is suspended, waiting for resume

Resume / yield

- resume: restarts the suspended coroutine
- yield: store the context, go to suspend mode, return yield value

Test yourself

Extend the bytecode interpreter on slide 50 so that it supports yield and resume.

Hint: Note that this interpreter always starts at pc=0. Change eval so that it can start at any pc.

Reading

Required:

see the reading for Lecture 4

Recommended:

Implementation of Python generators

Fun:

continuations via continuation passing style

Summary

Implement Asymmetric Coroutines

- why a recursive interpreter with implicit stack won't do

Compiling AST to bytecode

- btw, compilation to assembly is pretty much the same

Bytecode Interpreter

- can exit without losing its state