Lecture 5

Implementing Coroutines

regexes with coroutines, compile AST to bytecode, bytecode interpreter

Ras Bodik
Mangpo and Ali

Hack Your Language!

CS164: Introduction to Programming Languages and Compilers, Spring 2013

UC Berkeley

Administrativia

Exam dates:

midterm 1: **March 19** (80 min, during lecture)

midterm 2: **May 2** (80 min, during last lecture)

final exam (poster and demo): **May 15**, Wed, 11:30am,
the Woz, (3 hours)

What you will learn today

Regexes with coroutines

- backtracking in regex matching is concise w/ coroutines

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

PA2

PA2 was released today, due Monday

- bytecode interpreter of coroutines
- after PA2, you will be able to implement iterators
- in PA3, you will build Prolog on top of your coroutines

Before PA2, you may want some c/r exercises

1) lazy list concatenation
2) regexes

Solve at least the lazy list problem before you start on PA2

To find links to these problems, look for slides titled Test Yourself.

“Honors” cs164?

I am looking for a few Über-hacker teams to do a variant of the cs164 project.

Not really a harder variant, just one for which we don’t have a full reference implementation.

In PA2: instead of an coroutine interpreter, you’d build a compiler.

Another, simpler variant: implement cs164 project in JS, not Python.

Review of Coroutines

Review of L4: Why Coroutines

Loop calls iterator function to get the next item

- eg the next token from the input stream

The iterator maintains state between two such calls

- eg pointer to the input stream.

Maintenance of that state may be difficult

- see the permutation iterator in L4

See also Python justification for coroutines

- called generators in Python

Review of L4: Three coroutine constructs

co = coroutine(body) lambda --> handle

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

yv = resume(co, rv) handle x 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 coroutine that resumed into the current coroutine, passing yv to resume

Example

```
f(1,2)

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

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

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

The call `f(1,2)` evaluates to 3.

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 waiting in yield statements 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?

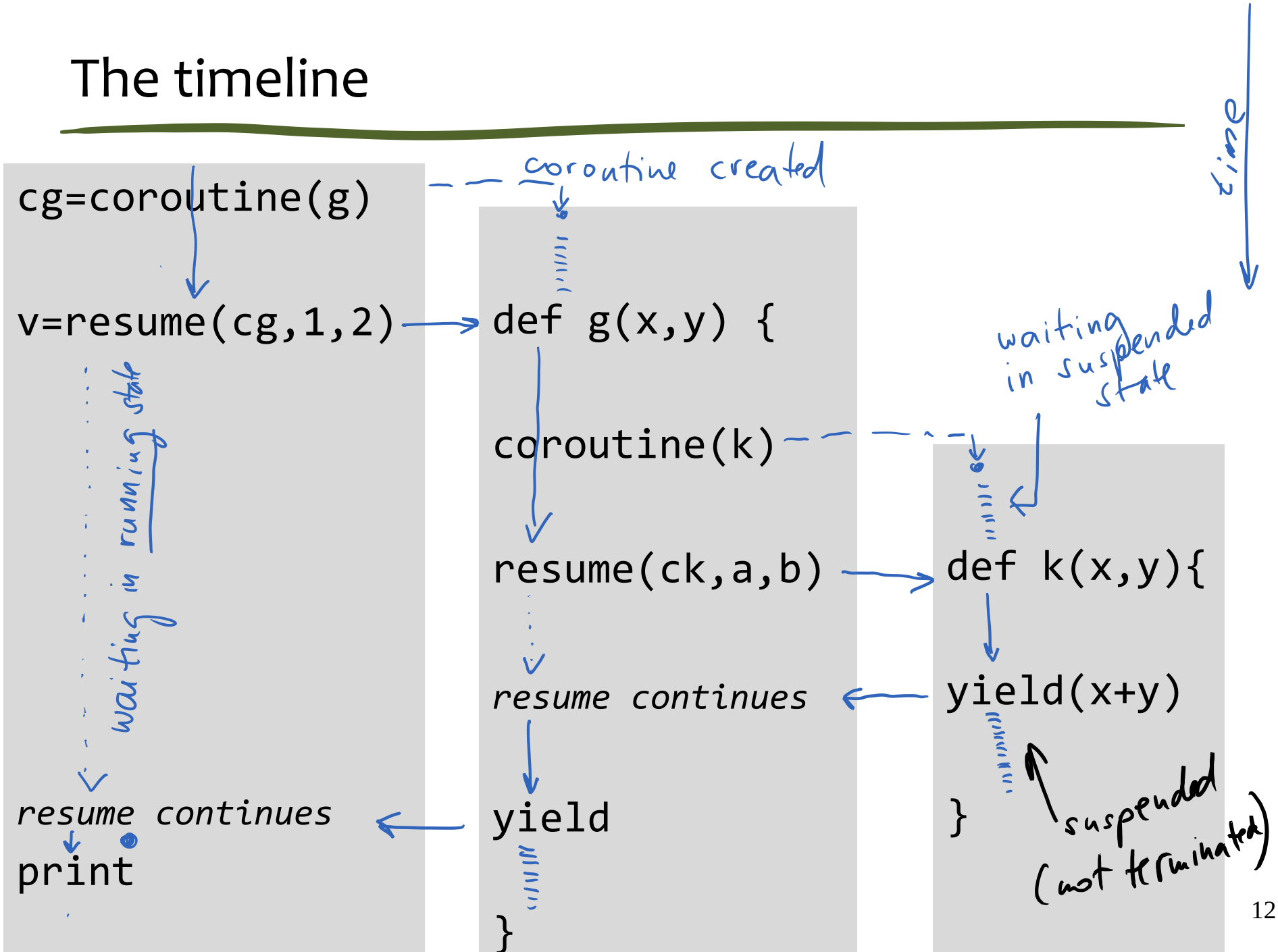
suspended, running, terminated

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

The timeline



Are coroutines like calls?

Which of resume, yield behaves like a call?

`resume`, for two reasons:

- 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 corou. handle)

`yield` is more like return:

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

Test yourself

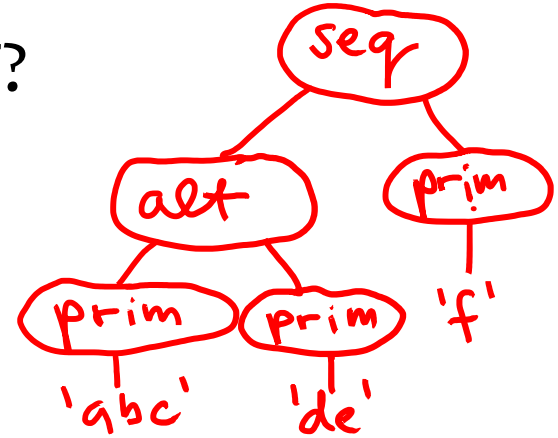
Implement lazy list concatenation

(link to the excercise)

Implement regular expression with c/r

(abc|de)f

parses into what AST?



which we compile into this Lua code

```
match("def", seq(alt(prim("abc"),prim("de")),prim("f")))
```

Overview of c/r-based regex matching

The nested function calls in the expression

```
seq(alt(prim("abc"), prim("de")), prim("f"))
```

create a closure that is the body of a coroutine C.

C will be resumed into by the matcher's top loop with the goal of iterating over all possible ways in which the pattern `(abc|de)f` matches the input "def".

The matcher loop will stop when a full match (from beginning to the end) has been found.

For more details, see section 5.3 in Revisiting Coroutines.

Regexes with coroutines

```
match("def",
seq(alt(prim("abc"),prim("de")),prim("f")))

def match(S, patt) {
    // wrap turns a lambda into c/r iterator
    def m=coroutine.wrap(lambda(){ patt(S,0) })
    for (pos in m) {  // for all matches
        if (pos==len(S)) { // matched all input?
            return true
    }   }
    return false
}
```

prim

-- matching a string literal (primitive goal)

```
def prim(str) {
    // this lmbda will be the body of c/r
    lambda(S,pos) {
        deflen = len(str)
        // does str match S[pos:pos+len]?
        if (sub(S,pos,pos+len-1)==str) {
            yield(pos+len)
}   }   }
```

Let's implement `alt`

What we came up with in the lecture:

```
def alt(patt1,patt2) {
   lambda(S,pos) {
      def m = coroutine.wrap(lambda(){ patt1(S,pos) })
      for (pos in m) {  // for all matches
          yield pos
      }
      m = coroutine.wrap(lambda(){ patt2(S,pos) })
      for (pos in m) {  // for all matches
          yield pos
} } }
```

The code on next slide is semantically equivalent (and simpler).

alt

```
-- alternative patterns (disjunction)

def alt(patt1,patt2) {
   lambda(S,pos) {
        patt1(S,pos)
        patt2(S,pos)
   }
}
```

seq

-- sequence of sub-patterns (conjunction)

```
def seq(patt1,patt2) {
    lambda(S,pos) {
        def btpoint=coroutine.wrap(
            lambda(){ patt1(S,pos) }
        )
        for npos in btpoint {
            patt2(S,npos)
        }
    }
}
```

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 PA1 to support c/r

- treating resume/yield as call/return

This extension will fail,

- motivating a non-recursive interpreter architecture

This failure should be unsurprising,

- but may clarify the contrast between PA1 and PA2

Pseudocode of PA1 recursive interpreter

```
def eval(node) {
    switch (node.op) {
      case ‘n’:    // integer lieteral
                   return node.val
      case ‘call’: // E(E,E)
                   def fn = eval(node.fun)
                   def t1 = eval(node.arg1)
                   def t2 = eval(node.arg2)
                   return call(fn, t1, t2)
    }
    def call(fun, v1, v2) {
      // set up the new environment, mapping arg vals to params
      eval(body of fun, new_env)
    }
}
```

The recursive PA1 interpreter

Program is represented as an AST

- evaluated by walking the AST bottom-up

State (context) of the interpreter:

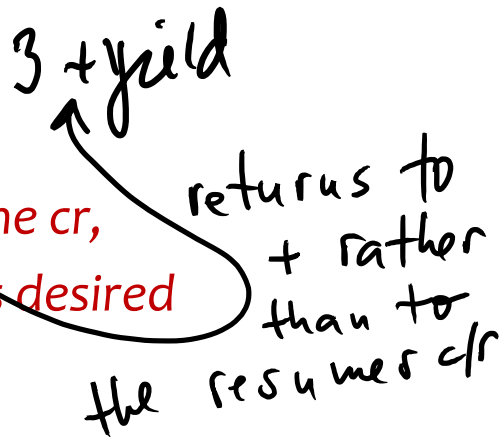
- control: what AST nodes to evaluate next (the "PC")
- data: intermediate values (arguments to operators, calls)

Where is this state maintained?

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

An attempt for a recursive corou. interpreter

```
def eval(node) {
    switch (node.op) {
    case 'resume': // resume E, E
                   evaluate the expression that gives c/r hadle  -- 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 cr,
                  rather than to the resumer c/r as was desired
    }
}
```



Draw env and call stacks at this point

```
f(1,2)
def f(x,y) {
  cg=coroutine(g)
  resume(cg,1,2)
}
```

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

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

Draw the environment frames and 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)

Summary

A recursive PA1-like interpreter cannot *yield*

because it maintains its control stack on the host call stack

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, …)

The plan

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 PA1, they were in the interpreter's variables

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

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
```

Compile AST to bytecode

Traverse AST, emiting code rather than evaluating

- when the generated code is executed, it evaluates the AST
- (see L3 for discussion of "non-standard" evaluation)

What's the type of translation function **b**?

- b : AST -> (Code, RegName)

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

- when c is executed, the value of tree resides in register r

Compile AST to bytecode (cont)

```
def b(tree):
        switch tree.op:
        +:      (c1,r1) = b(t.left)
                (c2,r2) = b(t.right)
                def r = freshReg()
                def instr = "%s = %s + %s\n" % (r, r1, r2)
                return (c1 + c2 + instr, r)
        n:      complete this so that you can correctly compile AST 2+3+4


        =:      complete this so that you can compile AST x=1+3+4; y=x+2
```

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

We had seen stack-based bytecode

- intermediate values were on data stack
- see "Compiler" in Lecture 3

Summary (2)

Bytecode translation recursively linearizes the AST

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

Test yourself

Complete the bytecode copiler on slide 44.

Bytecode interpreter

Bytecode interpreter

Data (ie, environments for lexical scoping):

same as in PA1

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, 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 PA1)
                    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