HW 4 Jianzhong Chen 23478230

part 1 d b

~~part 2~~

~~part 3~~

part 2 >> 1
>> 2

part 3
Because with bytecode, we can maintain a process pointer, which allows us to exit a function call without losing the state of main call stack.

Part 4
The third resume will cause an error. The interpreter need to track the number of yields or return in the coroutine.

part 5
a yield after resume ~~will~~ will cause an error. ~~If the number of resume reaches the number of yield in the coroutine,~~ we need to implement the coroutine as an object, so that we will know a yied is inside a coroutine or not

part 6.
For each coroutine, we need to store
① A coroutine state (running, suspended, dead)
② A call stack
③ For each frame in the call stack, we need:
1) The programm Counter
2) The bytecode it to which the PC indexes
3) The environment.

part 7. ex coroutine1 = coroutine(sourceCoFun)
coroutine2 = coroutine(processCoFun).

① print "create source coroutine"
coroutine1 suspend
coroutine2 (~~was~~ undefine)
frames: 1) the frame for call to sourceCoFun.
PC before "yield(1)"

② print "Running process coroutine"
coroutine1 suspended
coroutine2. Running
frames: 1) the frame for sourceCoFun
PC After "yield(1)"
2) the frame for call to processCoFun.
pc After yield("First" + x1).

③ print "Running source coroutine."
Coroutine1 Running
coroutine2. suspended
frames: 1) the frame for call to sourceCoFun.
PC After print "Running source coroutine".
2) the frame for call to processCoFun.
PC After yield("First" + x1).

④ print ~~"Finish"~~ "Finish process coroutine".
coroutine1 dead
coroutine2 Running
frames: 1) the frame for call to ~~process fun~~ processCoFun
pc after print "Finish process coroutine"

⑤ print "End"
Coroutine1 dead
Coroutine 2 dead.
frames: None