HARVARD
John A. Paulson
School of Engineering
and Applied Sciences

CS153: Compilers
Lecture 24:
Compiling Control Flow

Stephen Chong

https://www.seas.harvard.edu/courses/cs153

Pre-class Puzzle

- Consider Python generators, as shown in this program, which outputs the numbers 1 through 10

```
def my_gener(x):
  yield x
  yield x+1
  for y in range(2, 10):
    yield x+y

for value in my_gener(1):
  print(value)
```

- How would you compile generators?

Announcements

- Project 7 out
  - Due Thursday Nov 29 (2 days)
- Project 8 out
  - Due Saturday Dec 8 (11 days)
- Final exam: Wed December 12, 9am-12pm, Emerson 305
  - Covers everything except guest lectures
    - Lec 1-21, 23, 24, and all projects are fair game!
  - 30 multiple choice questions
  - Open book, open note, open laptop
  - No internet (except to look up notes, etc.),
    - No looking up answers, no communicating with anyone

Today

- Compiling control flow
  - Break and continue
  - Exceptions
    - "Zero cost" exceptions
  - Generators

Control Flow

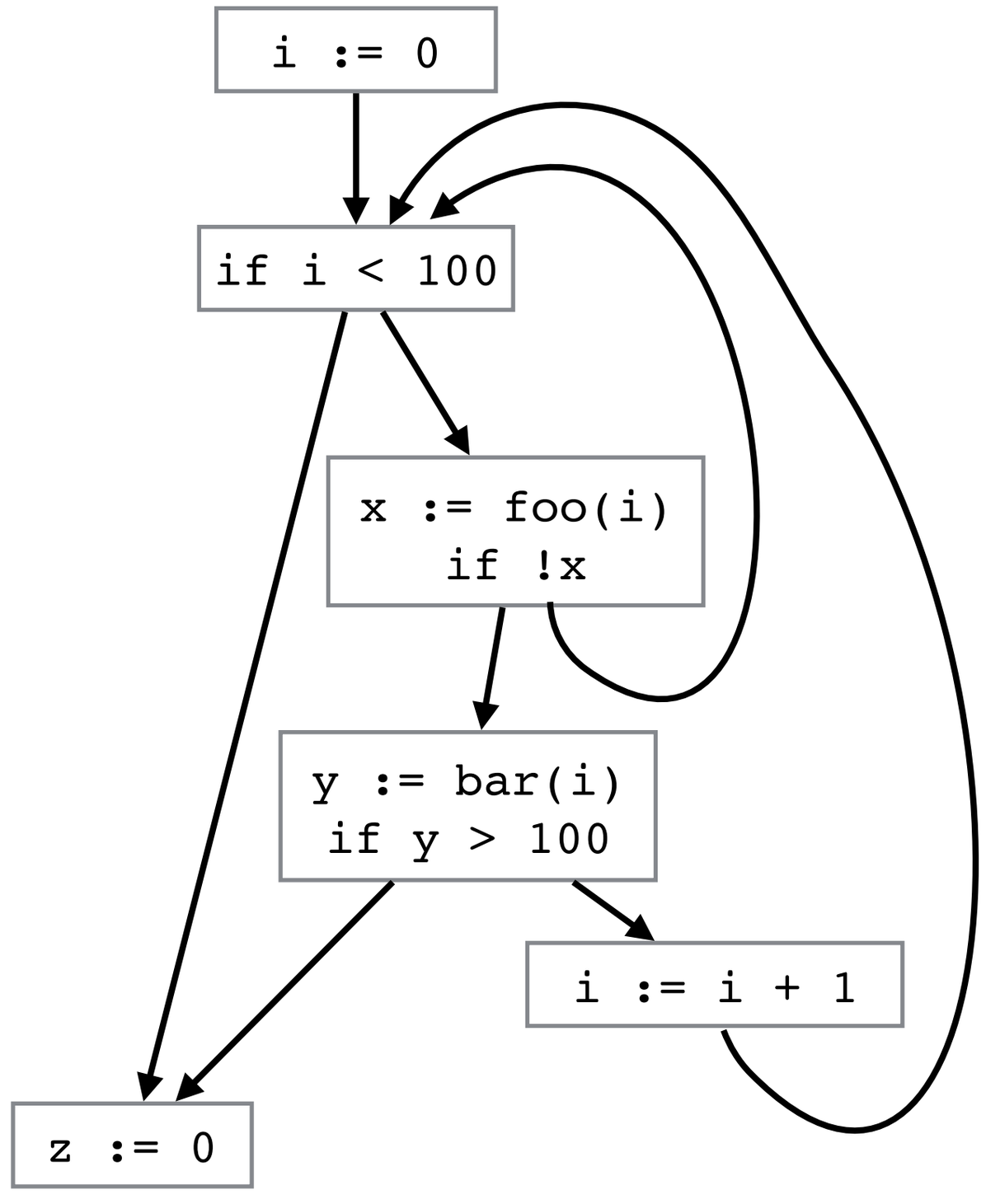
- So far we have seen standard control flow constructs
  - Sequence, selection, iteration
- But modern languages have additional control flow constructs! How do we deal with them?

Break and Continue

- Many languages have a statement to exit a loop early
  - Typically `break`
- Also have statement to continue with next iteration
  - Typically `continue`
- Straightforward to construct CFG
  - `continue` statements become jumps to loop header
  - `break` statements become jumps to statement after loop

Example

```
i := 0;
while (i < 100) {
  x := foo(i);
  if (!x) {
    continue;
  }
  y := bar(i);
  if (y > 100) {
    break;
  }
  i := i+1;
}
z := 0
```



Labeled Loops

- Typically **`break`** and **`continue`** are with respect to the closest loop (**`while`**, **`for`**, **`do-while`**, etc.)
- C, C++, Java, etc., allow labeled loops
  - e.g.,
```
L: while (e) {
       while (e') {
           ...
           break L;
           ...
       }
   }
```
- To compile, maintain stack of loops as processing statements
  - For each loop, record label of loop, **`break`** target and **`continue`** target
  - When **`break`** or **`continue`** is encountered, use the loop info at top of stack
  - When **`break L`** or **`continue L`** is encountered, use the loop info for the labeled loop

Exceptions

- Many languages support **exceptions** and **exception handling**
- A mechanism to indicate and handle errors or unusual conditions
- Typically a construct to `raise` or `throw` a value, and a construct to indicate how to handle exceptions
  - In Java, `try {...} catch (Exception e) {...}`
  - In OCaml, `try ... with ...`
- E.g.

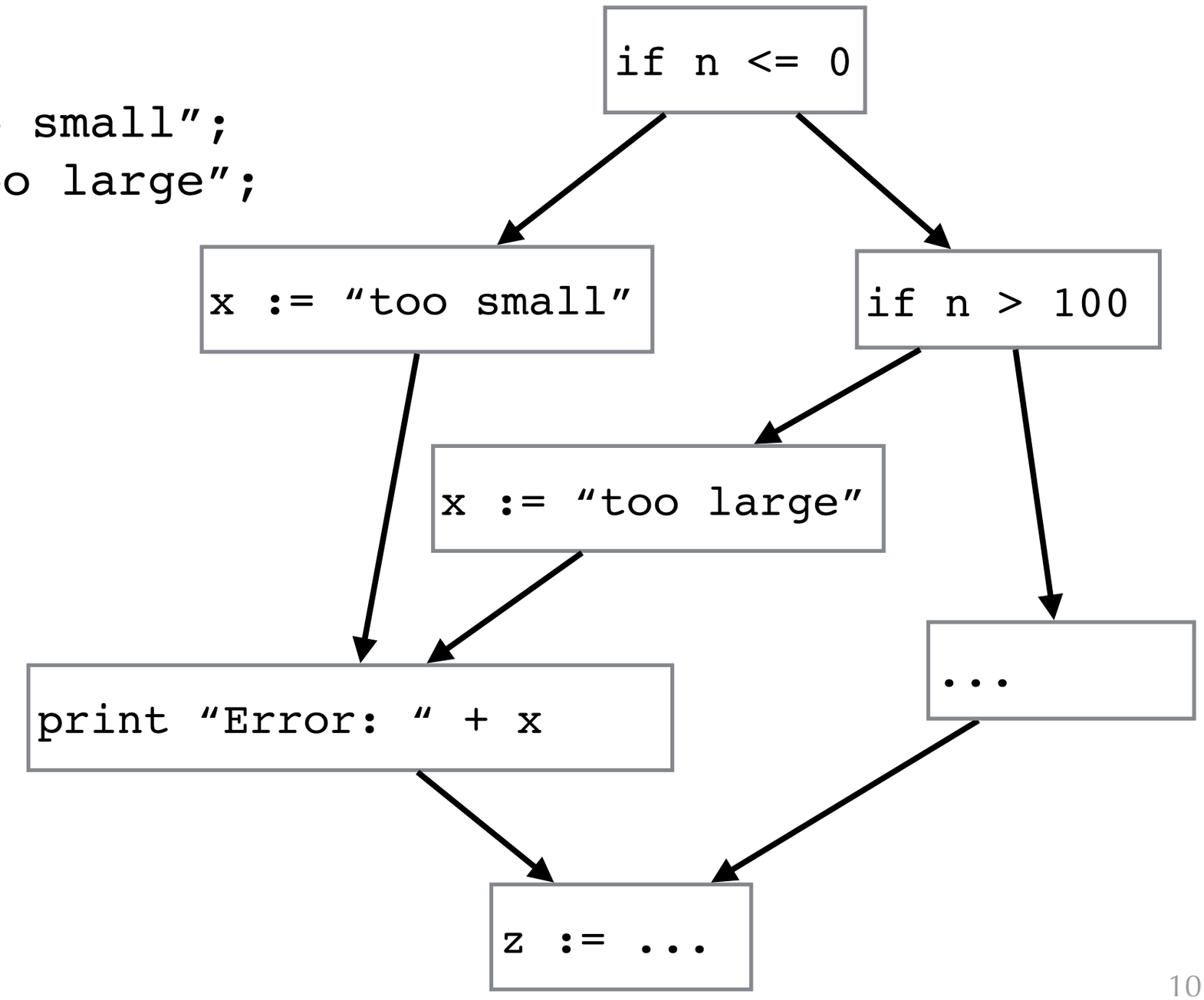
```
try {
    if (n <= 0) throw “too small”;
    if (n > 100) throw “too large”;
    ...
}
catch x {
    print “Error: “ + x;
}
```

- How to compile exceptions and handlers?

Handling Exceptions Within A Procedure

- Straightforward! Just affects CFG...

```
try {
   if (n <= 0) throw “too small”;
   if (n > 100) throw “too large”;
   ...
}
catch x {
   print “Error: “ + x;
}
z := ...
```



Handling Exceptions Across Calls?

```
void foo() {
  try {
    x := bar();
    ...
  }
  catch y {
    x := 0;
  }
  ...
}
```

```
void bar() {
  ...
  if (...) {
    throw “Uh oh!”;
  }
  ...
}
```

- Function `bar` might return normally or with an exception! How do we handle this?

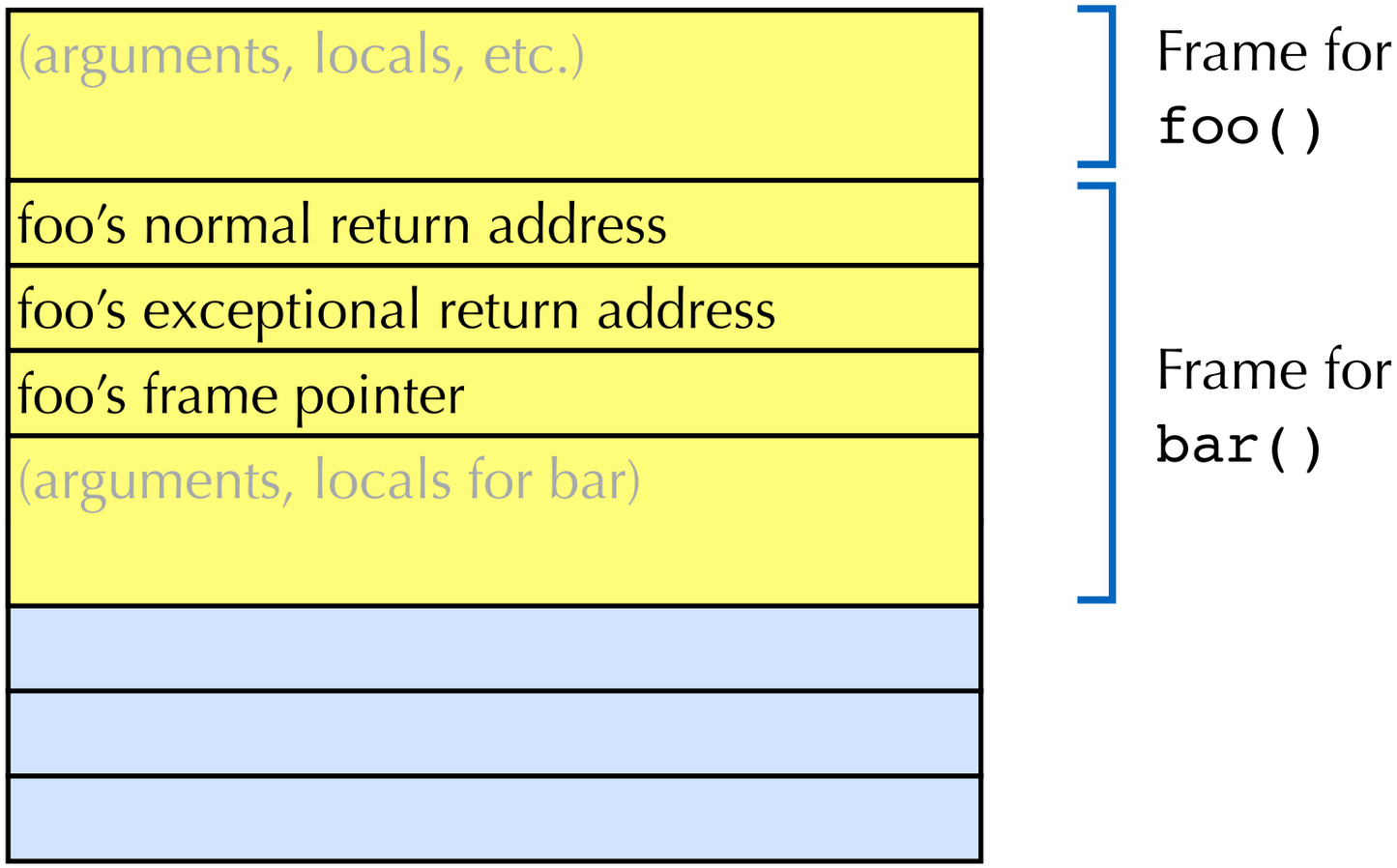
Handler Adresses

- Callee could return normally or exceptionally
- Key idea: provide two return addresses!
  - One for returning normally
  - One for returning exceptionally
- Note: This is a change in calling convention

Example

```
void foo() {
  try {
    x := bar();
    ...
  }
  catch y {
    x := 0;
  }
  ...
}

void bar() {
  ...
  if (...) {
    throw “Uh oh!”;
  }
  ...
}
```



Exception Mechanism Expensive

- Exceptions are exceptional
  - i.e., typically occur rarely
- But for every function call we need to push an exceptional return address on the stack
  - Expensive!
- Can we reduce the cost in the normal case?
  - i.e., when no exception is thrown?

"Zero Cost" Exceptions

- Key insight: for a given normal return address, the handler address is always the same!

```
void foo() {
  try {
    x := bar();
    ...
    foo();
  }
  catch y {
    x := 0;
  }
  try { baz(); }
  catch y { ... }
  ...
}
```

For these call sites, the catch handler is always this one

For this call site, the catch handler is always this one

"Zero Cost" Exceptions

- Key insight: for a given normal return address, the handler address is always the same!
- So don't bother putting handler address on stack
- Instead, create table that maps ranges of normal return addresses to handler addresses
- No overhead during normal execution
  - So-called "zero cost"
- When a function needs to return exceptionally, use normal return address to look up in table and find corresponding handler address

Extensions

- Finally blocks
  - `try { ... } finally { ... }`
  - `Finally` block will always execute when `try` block exits (whether normally or exceptionally)
    - Useful to allow clean up of resources (also, e.g., restoring of callee-save registers, regardless of how function returns)
    - Typically construct CFG by copying the `finally` block
- Different kinds of exceptions
  - Languages allow catch handlers to catch only certain kinds of exceptions
- Some languages allow handlers to tell program to resume execution at point where exception was thrown

Generators

- **Iterators** allow sequentially processing of values in a collection
  - From CLU programming language, Barbara Liskov et al., 1974
  - E.g., in Java:

```
Iterator iter = list.iterator();
 while (iter.hasNext()) {
   Object n = iter.next(); ...
 }
```

- **Generators** are one way of writing iterators
  - Can **yield** multiple values to caller (as opposed to normal functions, which just return once)
  - Generator maintains its state between invocations
  - In languages such as Python, Javascript, C#

Generator Examples

- Traversing a binary tree

```
gen() {
  if (this.left) { this.left.gen(); }
  yield this.val;
  if (this.right) { this.right.gen(); }
}
```

- Note, in Python, it's a little different.
  Can only `yield` from generator function

```
class Node:
  left = None
  right = None
  val = None
  def gen(self):
    if self.left:
      for x in self.left.gen():
        yield x
    yield self.val
    if self.right:
      for x in self.right.gen():
        yield x
```

Generator Examples

- Fibonacci

```
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a+b
```

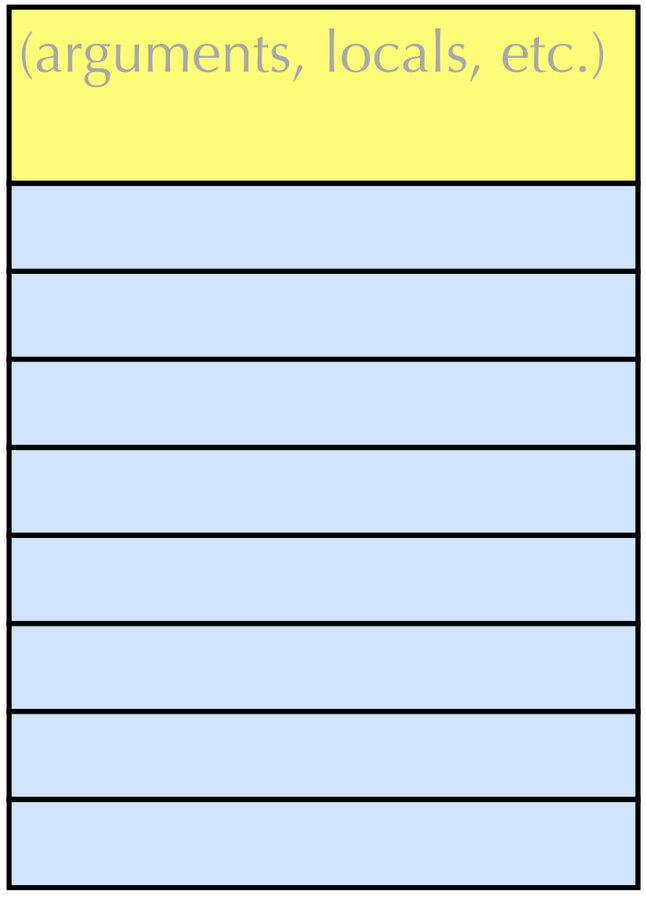
How to Compile Generators?

- Need to maintain local state of generator between yields of values
- Key idea: have a separate call stack for generator
  - Stack frames for generators are not reclaimed when generator yields a value
- Represent a generator as a call stack, resume address,
- When consumer invokes generator for next value:
  - Like a normal function call, but set up frame pointer and stack pointer for the generator's call stack, and jump to generator's resume address. Needs to save off frame pointer, stack pointer, and yield address
- When generator yields a value:
  - Restore frame pointer and stack pointer appropriately for consumer
  - Save resume address
  - Jump to the yield address, passing yielded value

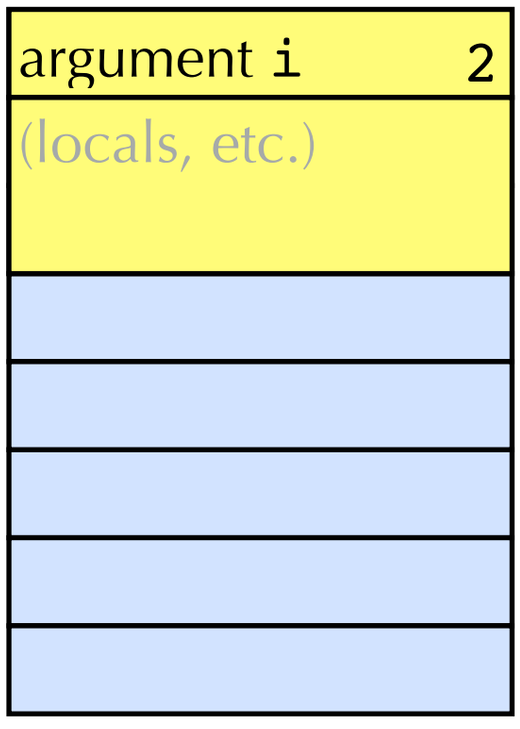
Example

```
    g = gen(2)
p1: print next(g)
p2: print next(g)
p3: print next(g)
```

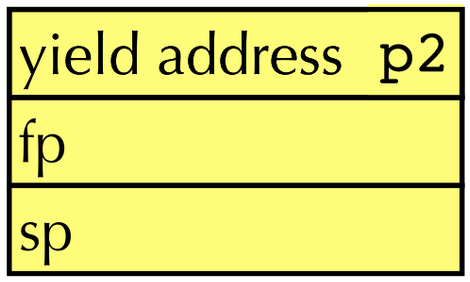
```
  def gen(i):
p4:yield 40+i
p5:yield 41+i
p6:yield 42+i
```



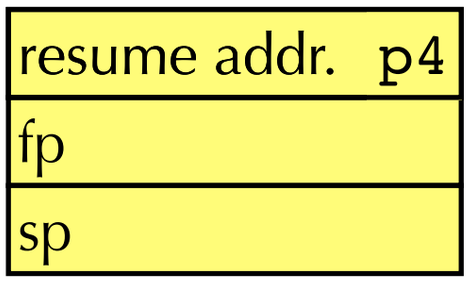
Stack



Consumer state



Generator state



Notes

- Generators are an example of **co-routines**
  - Co-routines can exit by invoking other code, and then later return to same program point
- **Continuations** are a useful way to think about and compile control flow constructs
  - See CS152!