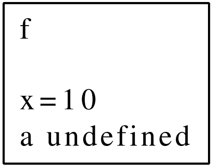
Different Execution Concepts: Generators and Coroutines

Carl Friedrich Bolz, David Schneider
Dynamische Programmiersprachen
Heinrich-Heine-Universität Düsseldorf
Sommersemester 2010

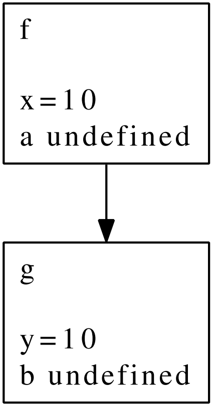
Stack-Discipline

```python
def f(x):
    a = g(x)
    return a + 1

def g(y):
    b = h(y + 5)
    return b * 2

def h(z):
    return z - 1
```

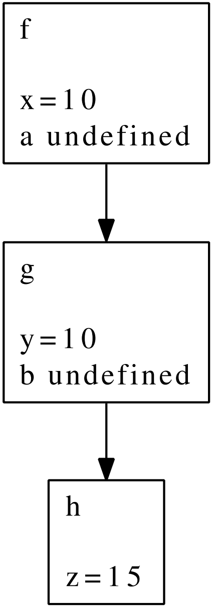
Execution Stack (1)



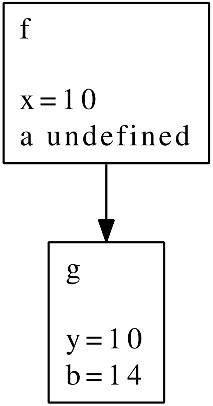
Execution Stack (2)



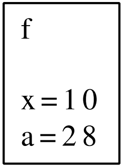
Execution Stack (3)



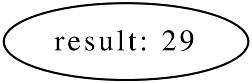
Execution Stack (4)



Execution Stack (5)



Execution Stack (6)



Stack-Discipline

- The boxes in the diagrams are called *frames*
- A frame holds the execution state of a function
- Frames of functions (in imperative languages) are typically organized as a *stack* - a function starts running - it then runs for a while, possibly starting other functions - it can only stop running after the functions it started have stopped
- is this really necessary?

Generators

- make it possible to have suspended frames around in a limited way
- in Python: new keyword `yield`, which suspends the current function
- using `yield` makes the function a generator
- calling a generator yields an object that can be used to resume the function
- function is resumed with the `.next()` method
- at the end, generator throws a `StopIterator` exception

Stacks with Generators

```
def f(x):
    g = g(x)
    a = g.next()
    b = g.next()
    return a + b

def g(y):
    yield y + 1
    yield h(y)

def h(z):
    return z + 2
```

Stack-Behaviour of Generators

- one main stack of frames
- can have any number of suspended frames
- suspended frames can only be suspended at their top level

Usecases for Generators

- implement iterators in a natural way
- “threading” with explicit scheduling
- ...

Co-routines

- any number of frame stacks
- no restriction!
- jump randomly to any other frame stack
- confuse yourself in arbitrary ways
- in Python: `greenlet` module
- rarely used

Languages Supporting Coroutines

old ones:

- Simula
- Modula-2

new ones:

- Lua
- Go
- Io
- Icon
- Scheme
- ...

Usecases for Coroutines

- lightweight threads
- can do everything a generator does
- actor-model: all objects have their own coroutine
- confusion