HOF

Nested Functions

```
def maxpower(n,exp):
    def count(m,cnt):
        if m >= exp:
            return cnt
        return count(m*n, 1+cnt)
    return count(1,0)
```

```
>>> maxpower(2,1000)
10
>>> maxpower(3,1000)
7
```

Nested Functions

```
def maxpower(n,exp):
    def count(m,cnt):
        if m >= exp:
            return cnt
        return count(m*n, 1+cnt)
    return count(1,0)
```

Which variables live in the scope of **count** but not a local variable of **count**?

n **exp**

Nested Functions

```
def partial(op):
    def action(a,b):
      print(op(a,b))
    return action
f1 = partial(lambda x,y : x*2 + y*2)
```

```
>>> f1(5,10)
30
>>> f2 = partial(f1)
>>> f2(5,10)
30
None
```

Nested Functions

```
def partial(op):
    def action(a,b):
      print(op(a,b))
    return action
f1 = partial(lambda x,y : x*2 + y*2)
```

Which variables live in the scope of **`action`** but not local in **`action`**?

`op`

Lambda Expressions

A function definition: `lambda a,b,c : a + b`

Takes in 3 arguments : Evaluates to a number

A function application:

`(lambda a,b,c : a + b)(1,2,3)`

- (a+b) {{a→1,b→2,c→3}}
- (1+2)
- 3

A closure containing binding environment, An abstract concept

Lambda Expressions

A function definition:

```
lambda a,b,c : a + b
```

keyword

parameters

body

Similar to:

parameters

```
def f(a, b, c):
    return a + b
```

keyword

body

Lambda Expressions

a b missing c

```
>>> (lambda a,b,c : a + b + c)(1,2)
```

```
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    (lambda a,b,c : a + b + c)(1,2)
TypeError: <lambda>() missing 1 required positional
argument: 'c'
```

Lambda Expressions

```
(lambda a : lambda b : a + b)
```

Takes one argument, a

Returns a function (which takes one argument, b; and returns the result a + b)

```
(lambda a : lambda b : lambda c : a + b + c)
```

Takes one argument, a

Returns a function (which takes one argument, b; and returns another function which takes one argument, c, and returns the result a + b + c).

Lambda Expressions

```
(lambda a : lambda b : a + b)(1)
```

➢ (lambda b : a + b) {{ a→1 }}

```
>>> x = (lambda a : lambda b : a + b)(1)
>>> x
<function <lambda>.<locals>.<lambda> at 0x016A8C90>
>>> x(3)
4
>>>
```

(lambda b : a + b {{ a→1 }})(3)

X is assigned to the return value of calling this function, and this return value is itself a function:
(lambda b : a + b) {{ a→1 }}

Lambda Expressions

```
(lambda a : lambda b : a + b)(1)
```

- ➢ (lambda b : a + b) {{ a→1 }}

```
(lambda a : lambda b : a + b)(1)(2)
```

- ➢ (lambda b : a + b {{ a→1 }})(2)
- ➢ a + b {{ b→2 , a→1 }}
- ➢ 1 + 2
- ➢ 3

Lambda Expressions

```
(lambda a : lambda b : a + b)(1)(2)
```

Consecutive applications of function is treated as left associative.

Lambda Expressions

`(lambda x: x (lambda y: y))(lambda z : z)(1)`

- `(x (lambda y: y) {{ x → lambda z : z }}) (1)`
- ((lambda z : z) (lambda y: y)) (1)
- (z {{ z → lambda y : y }}) (1)
- (lambda y : y) (1)
- y {{ y → 1 }}
- 1