# 《计算机程序的构造和解释》
# Lab 04: Data Abstraction, Trees, and Mutable Values

助教：李晨曦、李煦阳、吴羽、徐鼎坤、张天昀

2022 年 10 月 19 日

# Anonymous Recursion

Consider this code:

```
def add1(n):
    return n + 1
```

## Insight

1 can be *generalized* to an arbitrary integer.

```
def add(n1):
    return lambda n:n + n1
```

## Question

how can we use add to define add1 ?

```
add1 = add(1)
```

# Anonymous Recursion

## Intuition 1

Generalize a term $t$ in a function $f$ yields a higher order function $g$ that is more generalized than $f$. And $f$ can be recovered from $g$ and $t$ by:

$$f = g(t)$$

# Anonymous Recursion

Consider this code:

```
def factor(n):
    return 1 if n == 0 else n * factor(n - 1)
```

## Insight

Underlined `factor` can be generalized to an arbitrary function.

```
def factor2(f):
    return lambda n:1 if n == 0 else n * f(n - 1)
```

## Question

How can we use `factor2` to define `factor` ?

# Anonymous Recursion

Use the Intuition 1:

```
factor = factor2(factor2)
```

But it's not correct.

```
factor2(factor2)
= lambda n:1 if n == 0 else n * factor2(n - 1)
```

`factor2` should be applied to a **function**, not integer.

## Question

What is that **function** ?

# Anonymous Recursion

Make some fix:

```
def factor3(f):
    return lambda n:1 if n == 0 else n * f(f)(n - 1)

factor = factor3(factor3)
```

Does it work ?

```
factor3(factor3)(3)
= 1 if 3 == 0 else 3 * factor3(factor3)(2)
= 3 * factor3(factor3)(2)
```

Wow, it just work fine !

# Anonymous Recursion

We can define a anonymous recursion function which computes factorial.

```
factor3(factor3)
= (lambda f: lambda n:1 if n == 0 else n * f(f)(n - 1))
  (lambda f: lambda n:1 if n == 0 else n * f(f)(n - 1))
```

# Anonymous Recursion

## Question

How can we use `factor2` to define `factor` ?

# Anonymous Recursion

```
def f(n):
    n * n + n
```

Consider this expression:

```
a = 5 * 5 + 5 + 1
```

## Insight

Underlined expression can be *folded* into `f(5)`

```
a = f(5) + 1
```

# Anonymous Recursion

It seems that ...

```
factor3
= lambda f: lambda n:1 if n == 0 else n * f(f)(n - 1)
= lambda f: factor2(f(f))
```

So,

```
factor
= factor3(factor3)
= (lambda f: factor2(f(f)))(lambda f: factor2(f(f)))
```

**Insight**

Underlined `factor2` can be generalized to an arbitrary function.

# Anonymous Recursion

Now, we get a super power function, `y`. `y` makes every "factor2 like" function to be a recursive function.

```
y = lambda f: (lambda x: f(x(x)))(lambda x: f(x(x)))
```

`y` has a very interesting property:

$$y(f) = f(y(f))$$

In fact,

```
y(factor2) = factor3(factor3)
           = factor2(factor3(factor3))
```

That's why `factor3` works.

# Anonymous Recursion

But wait, there's something wrong:

```
y(factor2)
= factor3(factor3)
= (lambda x: factor2(x(x)))(lambda x: factor2(x(x)))
= factor2((lambda x:(factor2(x(x))))
          (lambda x:(factor2(x(x)))))
= factor2(factor3(factor3))
= factor2(factor2(factor3(factor3)))
= ...
```

## Question

Why `factor3(factor3)`(the first version) works?

# Anonymous Recursion

What will `factor3(factor3)`(the first version) be evaluated to ?

```
factor3(factor3)
= (lambda n: 1 if n == 0 then n * factor3(factor3)(n - 1))
```

**Insight**

Python will not evaluate anything inside a function.

So, just fold a "smaller" term:

```
factor3
= lambda f: lambda n: 1 if n == 0 else n * f(f)(n - 1)
= lambda f: lambda n: factor2(f(f))(n)
```

# Anonymous Recursion

Finally, we get:

```
def y(f):
    a = lambda x: lambda n: f(x(x))(n)
    return a(a)
```

or

```
lambda f: (lambda a: a(a))(lambda x: lambda n: f(x(x))(n))
```

another form: ("Z combinator")

```
lambda f: (lambda a: a(a))(lambda x: f(lambda n: x(x)(n)))
```

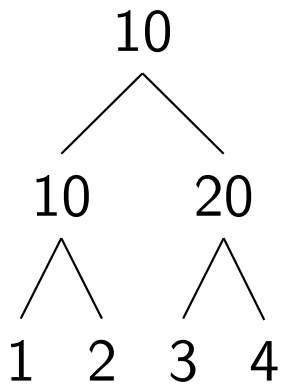
# Mobile

图: Mobile(sculpture)

# Mobile

25 min to finish.

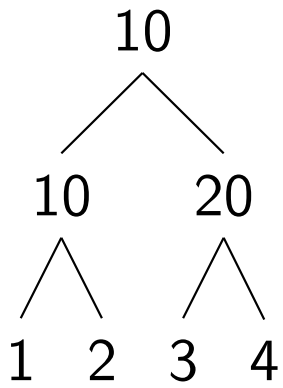
# Preorder




See document 2.4(Tree) for more information.

# Preorder

Catamorphism, or tree fold, or "recursion left, recursion right, then combine".




15 min to finish.

# Insert

Be carful!
15 min to finish.