Recursion

Ken Q Pu, Faculty of Science, Ontario Tech University

Recursion

Modern programming languages support recursion using **symbol binding**. Namely, we need to create a name for the expression that will use the name as well.

One way to expression recursion without symbol binding is to define it as a fixed point of some other unnamed function.

```
def factorial(n):
    if n == 0 then
        1
    else
        factorial(n-1) * n
```

But this does not work in LC because all expressions are *unnamed*.

Recursion

Modern programming languages support recursion using **symbol binding**. Namely, we need to create a name for the expression that will use the name as well.

One way to expression recursion without symbol binding is to define it as a fixed point of some other unnamed function.

```
def factorial(n):
    if n == 0 then
        1
    else
        factorial(n-1) * n
```

But this does not work in LC because all expressions are *unnamed*.

Recursion as fixed point

How do we express factorial as the solution of the following equation:

factorial = **H**(**factorial**)

It turns out to be rather trivial.

```
def factorial(n):
    if n == 0 then
        1
    else
        factorial(n-1) * n


def H(f, n):
    if n == 0 then
        1
    else
        f(n-1) * n


H(factorial) = ... = factorial
```

Fixed point function does not require bindings

Since H relies on parameters, it can be expressed using pure LC:

H = \f.\n. **IfElse** (**IsZero** n) **1** (**Mult** n (f (**Pred** n)))

But we want the factorial function, which is the fixed point of H.

$$Y = \lambda f.\,(\lambda x.\,f\;(x\;x))\;(\lambda x.\,f\;(x\;x))$$

This was an outstanding problem for many years until Haskell Curry discovered the Y-combinator.

Paul Graham named his VC fund after this concept in 2005.