# Thinking recursively

# Haskell语法快速回顾

```
-- 构造、解构
data Maybe a = Just a | Nothing
case m of Just x -> ... x
          Nothing -> ...
let Foo ... = foo
```

```
-- 词法作用域
let x = ...
in let x' = ...
   in ...

x = ... y ...
  where
    y =...
```

# Haskell语法快速回顾

-- 标识符字符集

- 小写开头（包括_）、字母、数字、'
- 大写开头、字母、数字、'

-- 中缀

- !、#、$、%、&、⋆、+、.、/、<、=、>、?、@、\、^、|、-、~、:
- : 开头的是中缀构造函数

```
-- 模块，导入，qualified 导入
import Foo.Bar
import qualified Foo.Bar as Bar

... Bar.qux ...
```

# Haskell语法快速回顾

```
-- 空白代表函数应用，左结合，最高优先级(10)
f x y === (f x) y

-- lambda
\ x y z -> ... x ... y ... z
```

```
-- 中缀函数可以指定优先级（0~9）
($) :: (a -> b) -> a -> b
f $ x = f x                    -- 此处板书有问题，注意结合性
infixr 0 $                     -- f(x)(y) != f $ x $ y

(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \ x -> f (g x)
infixr 9 .
```

# Haskell语法快速回顾

```
-- Guard 语法

foo x y z | ... = ...
          | ... = ...
          ...

case x of ... | ... ->
              | ... ->
              ...
          ... -> ...

otherwise :: Bool
otherwise = True
```

# 递归

- 递归是实现重复的一种方法

```
factorial 4 = 4 × 3 × 2 × 1
factorial 3 =     3 × 2 × 1
factorial 2 =         2 × 1
factorial 1 =             1
```

- 组成递归的两个部分：
  - 递归定义（两次递归之间的关系？）
  - 终止条件（何时退出？可选）

```
factorial 1 = 1
factorial n = n * factorial (n - 1)
```

# 和循环的关系?

```
-- c
int i;
for (i = 0; i < 10; i++){
    ...
}
```

```
-- Haskell
foo i | i < 10 = ...
                 foo (i+1)
      | otherwise = ...

foo 0
```

-- 循环初始化i， 递归接受初始值i
-- 递归可以调用自己若干次
-- 递归传递i，循环修改i

# What to do with stack?

大部分语言的调用约定 (calling convention)，调用方 (caller) 把参数推到执行栈上，然后进入被调用方(callee)，callee 在 return 时复位 SP。

```
fact(4)
4 * fact(3)
4 * 3 * fact(2)
4 * 3 * 2 * fact(1)
```

连续把4,3,2,1推到栈上，每次return复位一个stack frame

如果递归深度超过栈的最大深度，那么等不到 return 就会出现内存越界，i.e. stack overflow。

# tail recursive to rescue!

```
int fact(int acc, int n){
    if (n <= 1){return acc};
    else {return fact(n*acc, n-1);}
}
```

```
fact(1, 4)
fact(4, 3)
fact(12, 2)
fact(24, 1)
24
```

// 每次fact调用，都不再需要之前栈上的参数，而每次递归调用push参数的顺序是固定的，所以我们可以在没有return之前复位SP，用新的参数覆盖之前的参数，复用栈空间。

# 尾递归和Haskell

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)

-- GHC的栈是分段链表，默认上限非常大，通过+RTS -K参数缩小一些
> ghci +RTS -K1M
> Prelude> factorial 100000
> *** Exception: stack overflow

factorial :: Integer -> Integer -> Integer
factorial acc 0 = acc
factorial acc n = factorial (n*acc) (n - 1)
> factorial 100000
> *** Exception: stack overflow

-- Haskell惰性求值，导致生产了一大堆互相引用得任务盒
-- 最后进入任务盒时同样会消耗参数栈
```

# BangPattern to rescue!

```
{-# LANGUAGE BangPatterns #-}

factorial :: Integer -> Integer -> Integer
factorial !acc !1 = acc
factorial !acc !n = factorial (n*acc) (n - 1)
```

```
> factorial 1 100000
> 2824229407960347874293421578024535518477494926091224850578918086542977950901063017872551771413831163...
```

-- !后跟着的模式会在进入RHS的表达式之前被求值到常态。

# STG Prim Type

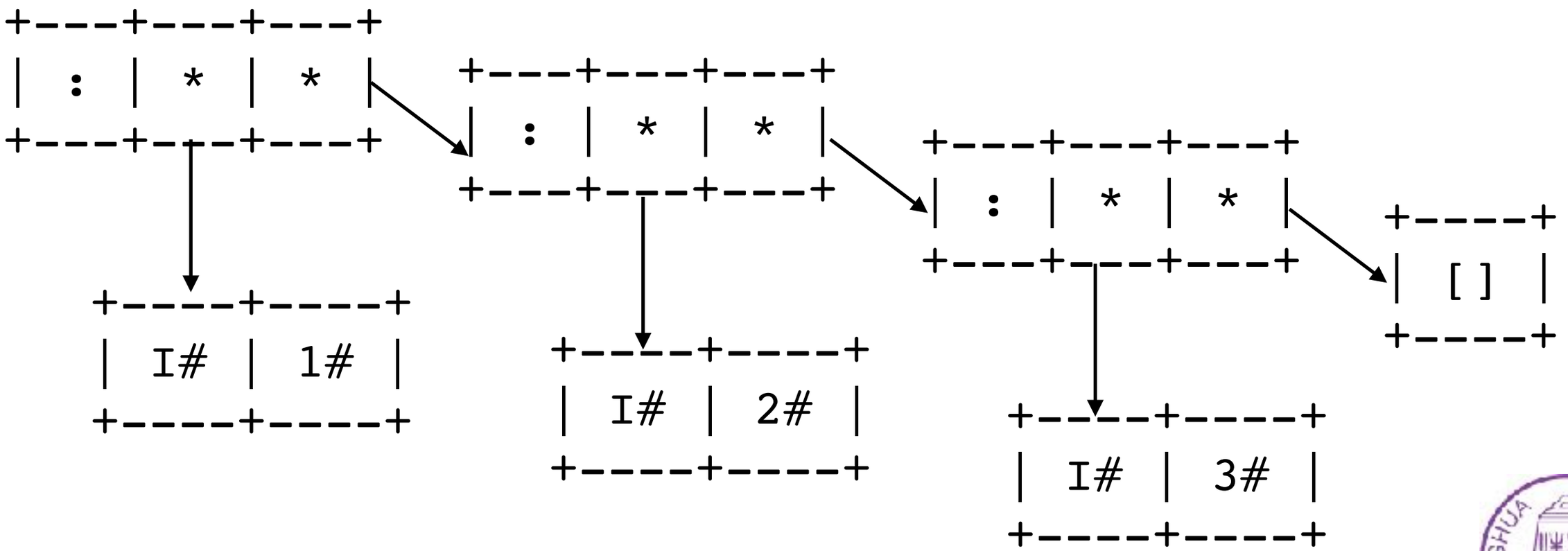
```
{-# LANGUAGE MagicHash #-}
-- from ghc-prim
import GHC.Prim
import GHC.Types

sumN# :: Int# -> Int#
sumN# 0# = 0#
sumN# n# = n# +# sumN# (n# -# 1#)
> I# (sum 100000#)
*** Exception: stack overflow

-- tail-recursive
sumN'# :: Int# -> Int# -> Int#
sumN'# acc# 0# = acc#
sumN'# acc# n# = sumN'# (n# +# acc#) (n# -# 1#)
> I# (sumN'# 0# 100000#)
5000050000
```

# Accumulator style

```
data BinTree a = Node (BinTree a) (BinTree a) a
               | Nil

-- direct style
countNode :: BinTree a -> Int
countNode (Node left right _) =
    countNode left + countNode right + 1
countNode Nil = 0

-- accumulator style
countNode' :: BinTree a -> Int -> Int
countNode' (Node left right _) !acc =
    countNode' left (countNode' right (acc+1))
countNode' Nil !acc = acc
```

# 单链表[a]

```
data [a] = a : [a] | []
-- data List a = Cons a (List a) | Nil

-- [1,2,3] == 1:2:3:[]
```



# 单链表[a]

```
data [a] = a : [a] | []


length :: [a] -> Int
length (_:xs) = 1 + length xs
length []     = 0

-- accumulator style
length :: [a] -> Int
length xs = lenAcc xs 0

lenAcc :: [a] -> Int -> Int
lenAcc []     n = n
lenAcc (_:ys) n = lenAcc ys (n+1)
```