Haskell与代数数据类型

data关键字

```
data Point = MakePoint Double Double
```

数据类型　　构造函数

```
MakePoint :: Double -> Double -> Point
```

```
-- 构造数据
pointA = MakePoint 3.0 4.0

-- 解构数据
case pointA of MakePoint x y -> ... x ... y

getX :: Point -> Double
getX (MakePoint x y) = x
```

中缀构造函数

```
data Point = Double :.: Double
```

数据类型　　构造函数

```
(:.:) :: Double -> Double -> Point
```

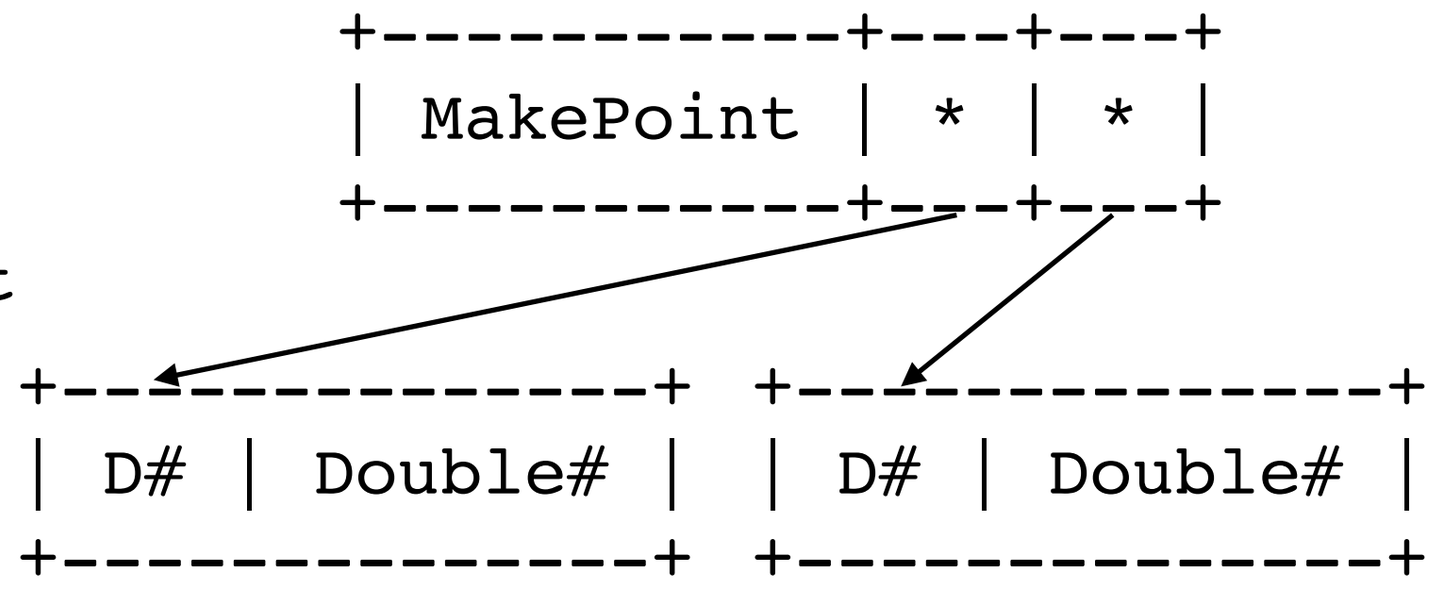
```
-- 构造数据
pointA = 3.0 :.: 4.0

-- 解构数据
case pointA of x :.: y -> ... x ... y

case pointA of (:.:) x y -> ...
```

记录语法

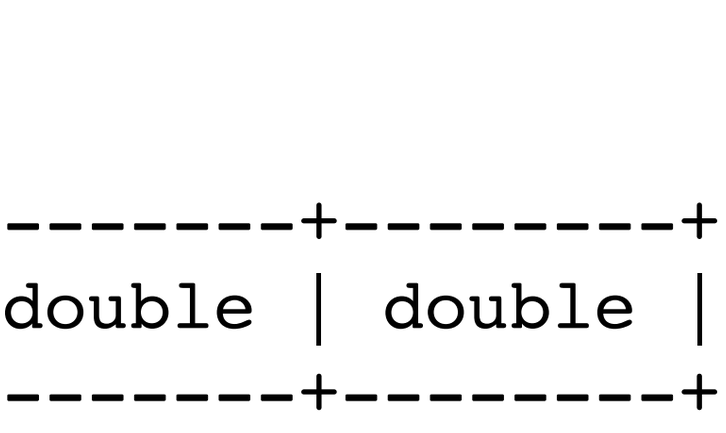
```
-- 添加新的绑定的时候都可以对数据进行解构
let MakePoint x y = pointA in ... x ... y
MakePoint x y = pointA

-- 使用记录语法，可以让编译器自动生成解构函数
data Point = MakePoint
     { getX :: Double
     , getY :: Double
     }

-- getX, getY :: Point -> Double

-- 构造数据的时候可以通过标签函数指定顺序
pointB = MakePoint{getY = 4.0, getX = 3.0}
```

c struct?

```
-- Haskell
data Point = MakePoint
     { getX :: Double
     , getY :: Double
     }
```




```
data Double = D# Double#

// In C
typedef struct {
   double x;
   double y;
} point;
```



UNPACK to rescue!

```haskell
-- Haskell
data Point = MakePoint
     { getX :: {-# UNPACK #-} !Double
     , getY :: {-# UNPACK #-} !Double
     }
data Double = D# Double#
```

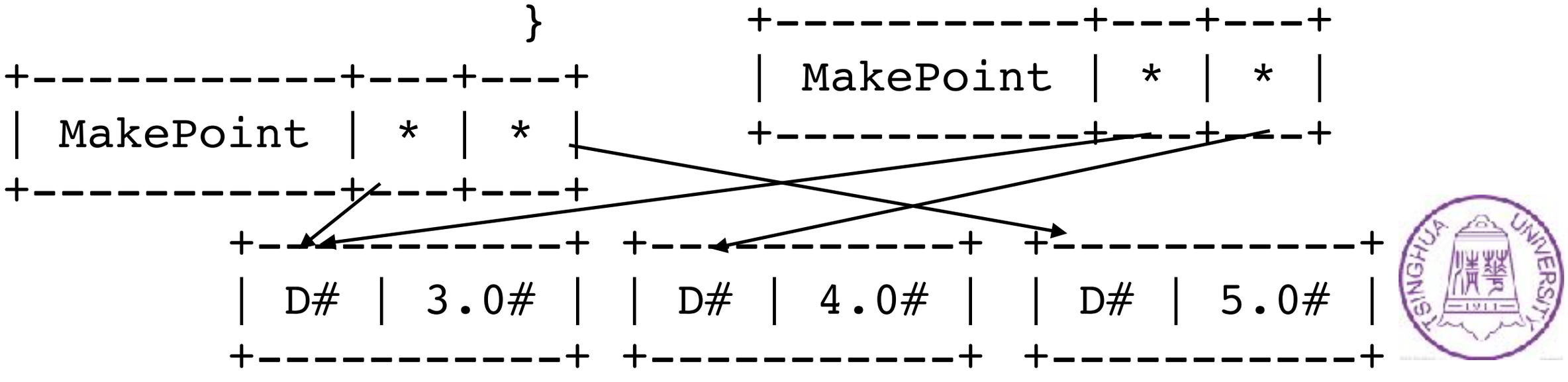
```
+-----------+---------+---------+
| MakePoint | Double# | Double# |
+-----------+---------+---------+
```

```c
// In C
typedef struct {
   double x;
   double y;
} point;
```

```
+--------+--------+
| double | double |
+--------+--------+
```

『更新』数据

```
-- Haskell中数据默认是不可变的，只能根据原有的数据创建新的
pointB :: Point
pointB = MakePoint (getX pointA) 5.0

-- 记录语法提供了创建新数据的语法糖
pointB = pointA {getY = 5.0}
-- 等价于
pointB = MakePoint{ getX = getX pointA
                  , getY = 5.0
                  }
```



抽象出类型?

```
-- 使用其他数值类型?
data FloatPoint = MakeFloatPoint Float Float
data IntPoint = MakeIntPoint Int Int
...
```

```
-- 使用类型变量抽象盒子的数据类型!
data Point a = MakePoint a a

type IntPoint = Point Int
type FloatPoint = Point Float

-- Point Int 和 Point Float 是不同的类型
-- Point Int :: Type
-- Point :: Type -> Type
```

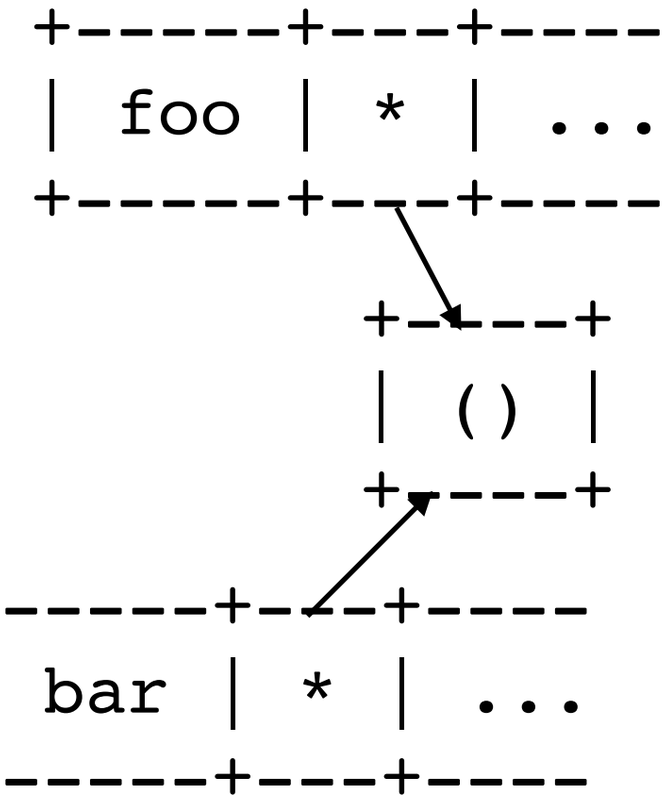
抽象出类型?

```
-- 使用 Point a 定义，我们可以抽象出一些通用的操作
flipXY :: Point a -> Point a
flipXY (MakePoint x y) = MakePoint y x

-- flipXY 可以工作在 IntPoint, FloatPoint...之上
```

```
-- 假如我们可以要求a类型的一些性质，就可以实现更有趣的操作
moveX :: Num a => Point a -> a -> Point a
moveX (MakePoint x y) dx = MakePoint (x + dx) y

-- 因为 moveX 使用到了 (+) :: Num a => a -> a -> a
-- 即要求 a 类型是一个数字 (Num a => ...)
-- 所以 moveX 也必须要求 a 类型是一个数字
```

面临选择?

```
-- 一个只有一个居民(inhabitant)的类型
data () = ()

-- GHC 会优化空盒子，程序里所有的()都指向
-- 静态内存的一个地址
```




```
-- 一个拥有两个居民的类型
data Bool = True | False

case x of True  -> ...
          False -> ...

-- 上述 Bool 类型就是标准库里的逻辑值类型
-- 程序里所有的 True, False 都指向静态内存的两个地址
```

Sum&Product

```
-- 在类型之间选择的类型，又被称为和类型(Sum Type)
data Either a b = Left a | Right b
-- Either a b 居民的数量，是 a 和 b 的居民数量之和

-- 同时包含若干类型的类型，又被称为积类型(Product Type)
data (a, b) = (a, b)
-- (a, b) 居民的数量，是 a 和 b 的居民数量之积

-- 代数数据类型里的『代数』，指的就是和类型和积类型。
```

"sum" is alternation (A | B, meaning A or B but not both)
"product" is combination (A B, meaning A and B together)

重要的代数类型

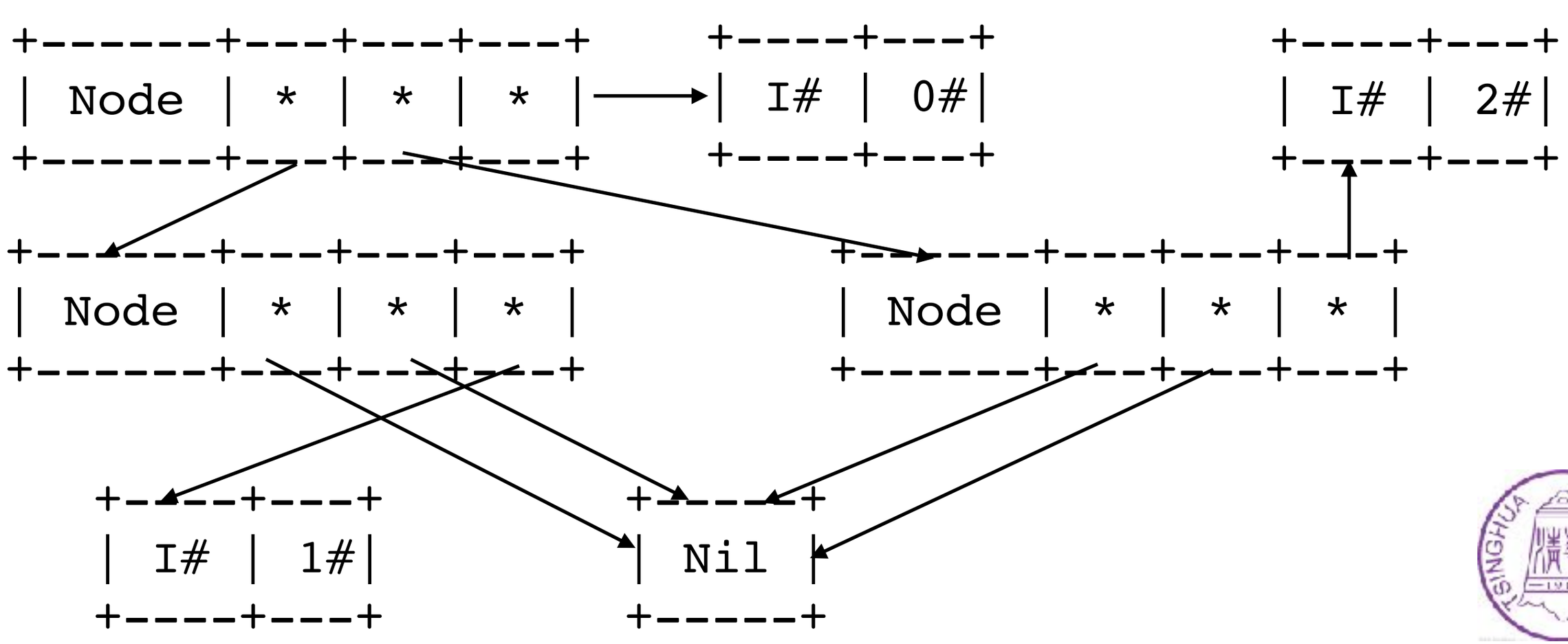
```
-- 标准库里表示可能不存在的值的类型 Maybe a
data Maybe a = Just a | Nothing

divMaybe x y | y == 0 = Nothing
             | otherwise = Just (x `div` y)

case x `divMaybe` y of Just ... -> ...
                       _        -> ...

-- 标准库里的单链表
data [a] = a : [a] | []
-- [1,2,3,4]
1:2:3:4:[]
```

和类型的记录语法?

```
-- 和类型里也可以使用记录语法，但非常不推荐
data Candidate = Fresh School
               | Experienced
                   { getCompany :: Company
                   , getPosition :: Position }

getCompany :: Candidate -> Company
getCompany (Experienced comp _) = comp

-- getCompany (Fresh _) = ???
-- 在和类型里使用记录语法，会引入部分(Partial)函数!
-- 在运行过程中会发生异常：
-- *** Exception: No match in record selector
getCompany
```

递归的数据定义

```
-- 二叉树
data BinTree a = Node (BinTree a) (BinTree a) a
               | Nil

Node (Node Nil Nil 1) (Node Nil Nil 2) 0
```

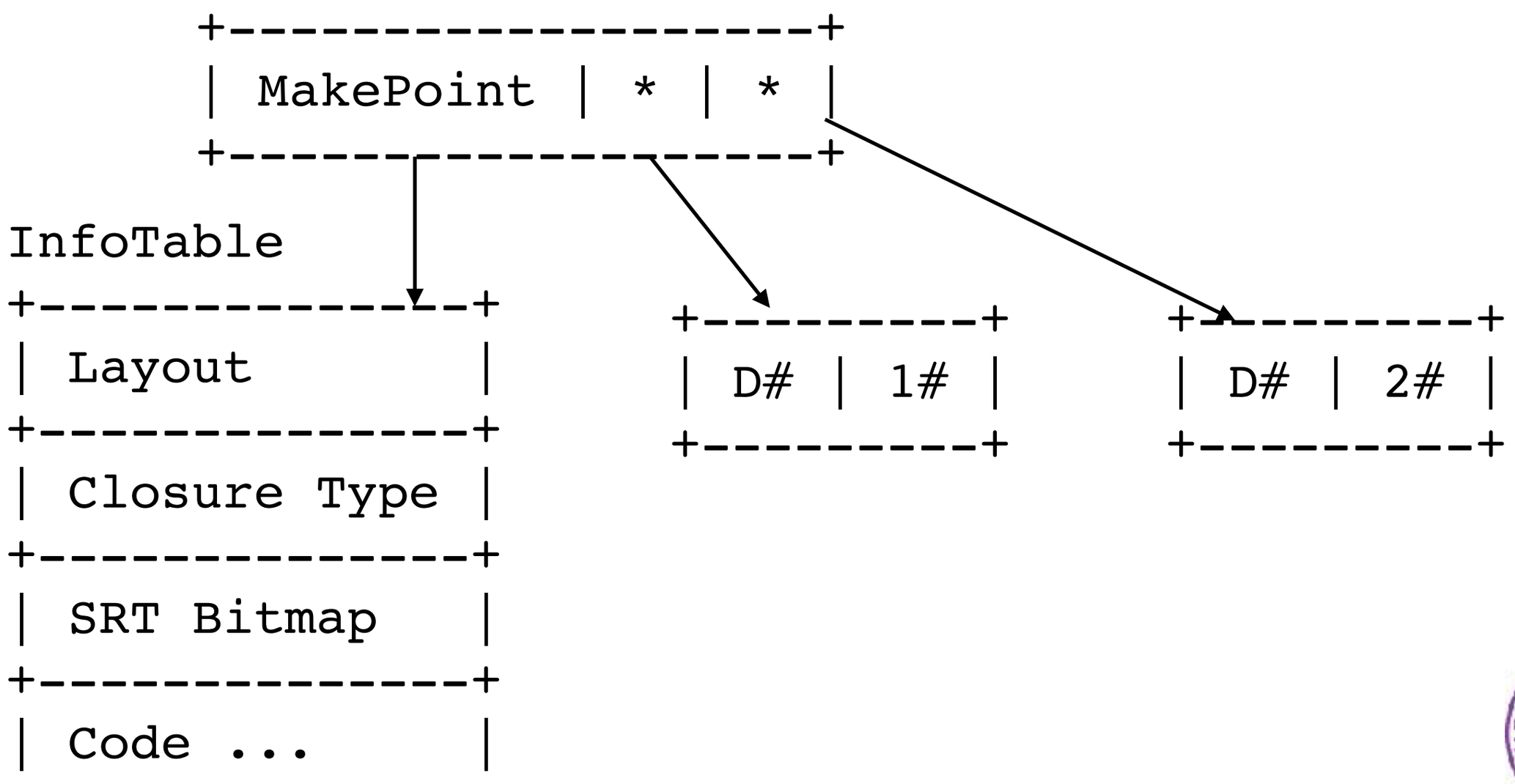


递归数据递归处理

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

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

countNode (Node (Node (Node Nil Nil 1)
                      (Node Nil Nil 2) 0)
-- countNode (Node Nil Nil 1)
     + countNode (Node Nil Nil 2) + 1
-- (countNode Nil + countNode Nil + 1)
     + (countNode Nil + countNode Nil + 1) + 1
-- (0 + 0 + 1) + (0 + 0 + 1) + 1
```

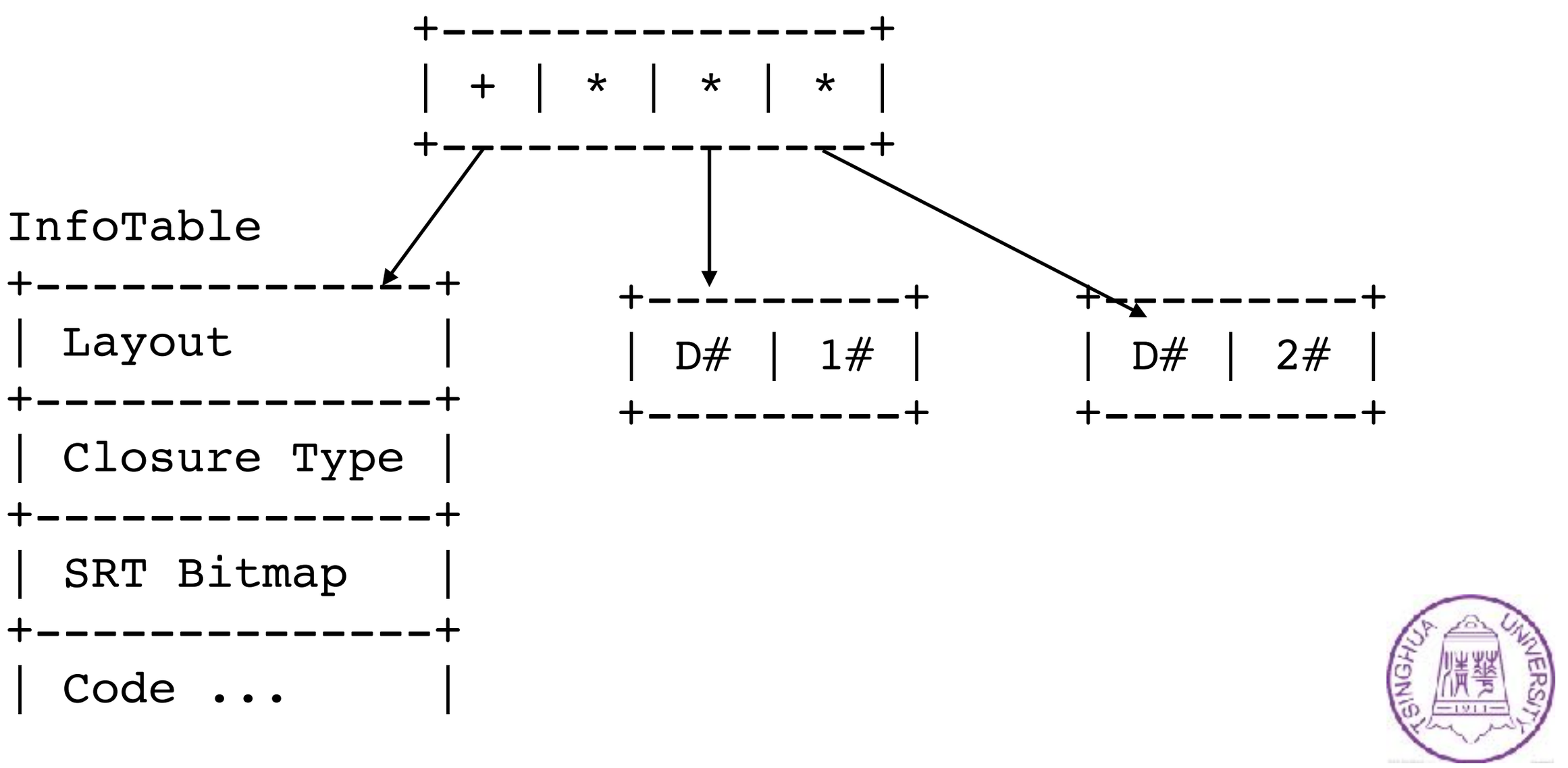
盒子比喻

```
-- 在 ghc 的堆(heap)上，除了构造函数像一个盒子
pointA = MakePoint 1 2
```



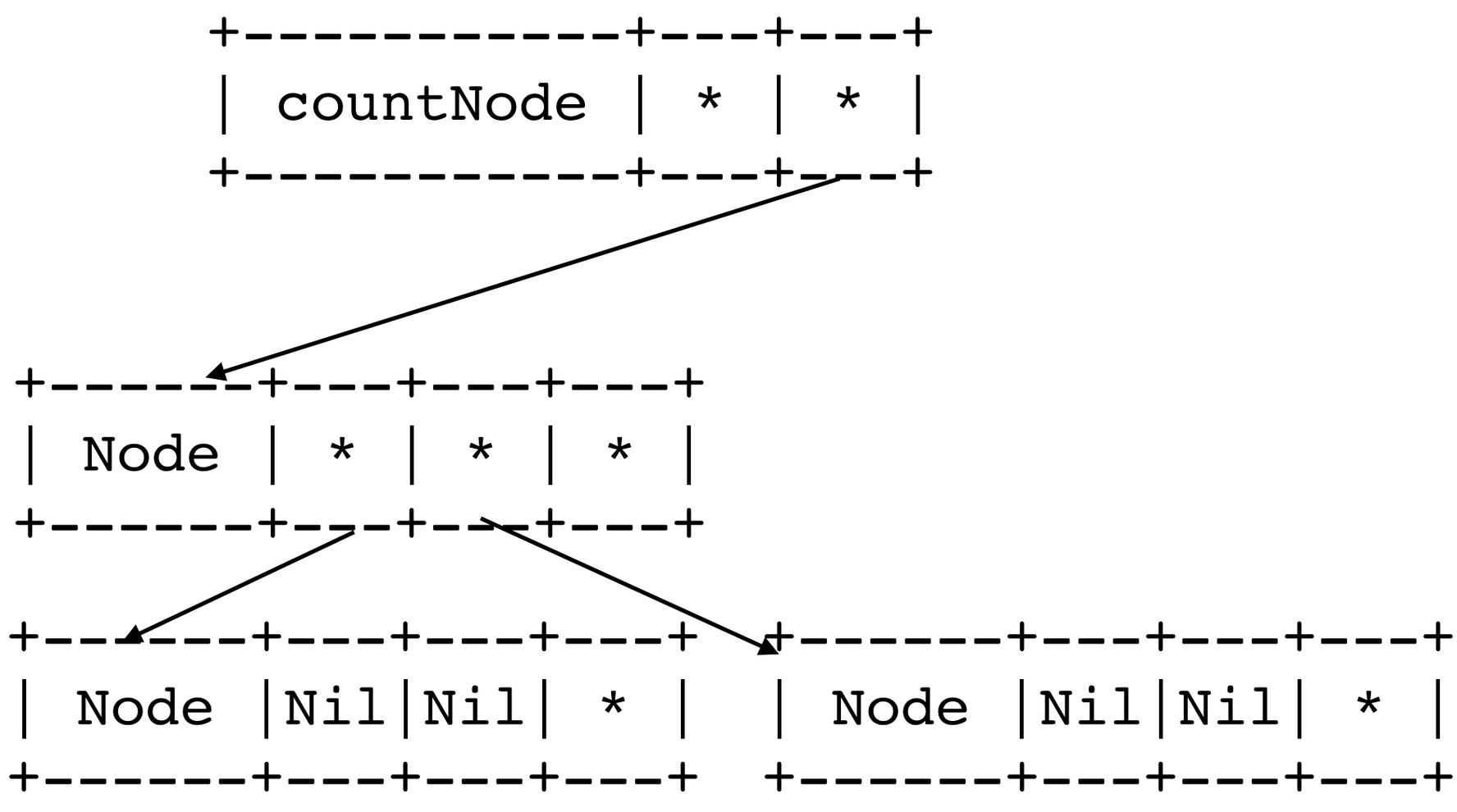
任务盒?

```
-- 在 ghc 的堆(heap)上，除了构造函数之外，有一类特除的盒子
1 + 2 :: Double
```



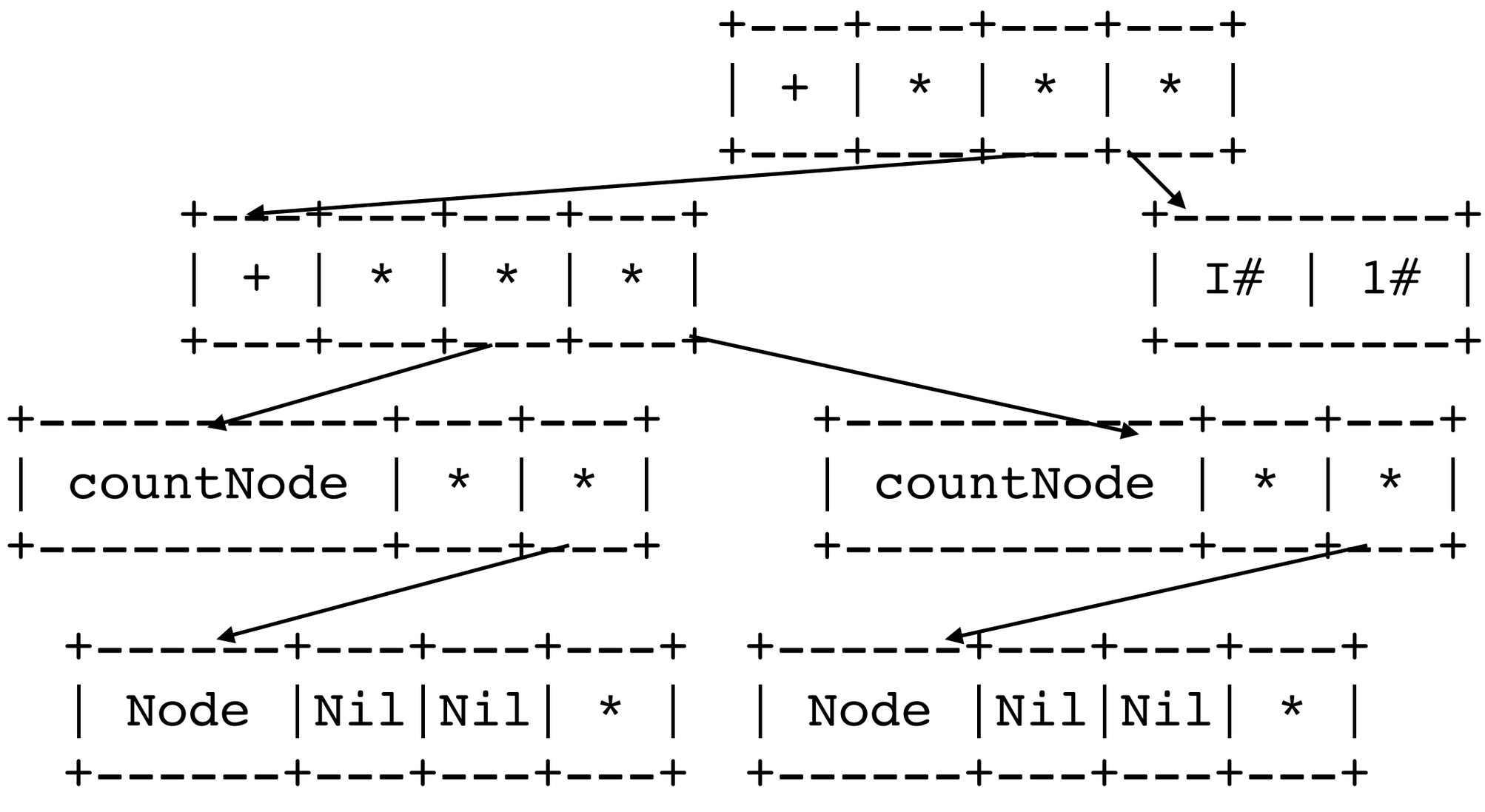
递归求解任务盒

```
countNode (Node (Node (Node Nil Nil 1)
                      (Node Nil Nil 2) 0)
```



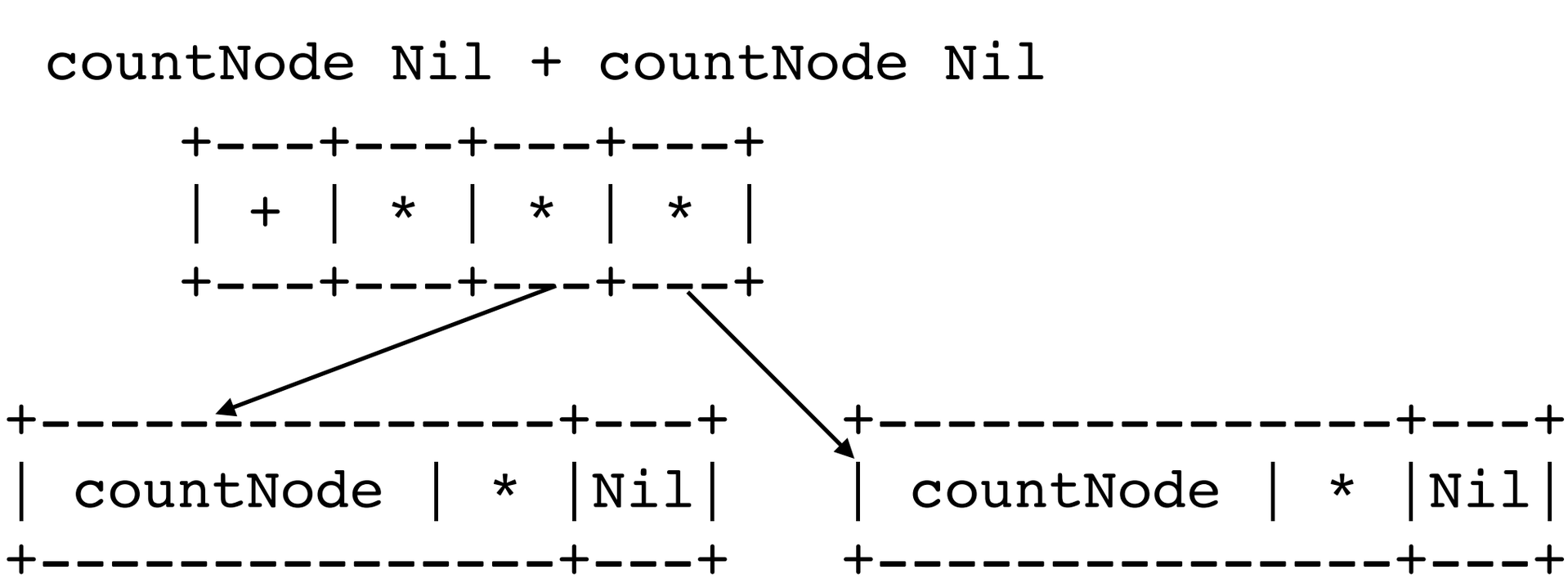
递归求解任务盒

```
countNode (Node Nil Nil 1)
    + countNode (Node Nil Nil 2) + 1
```

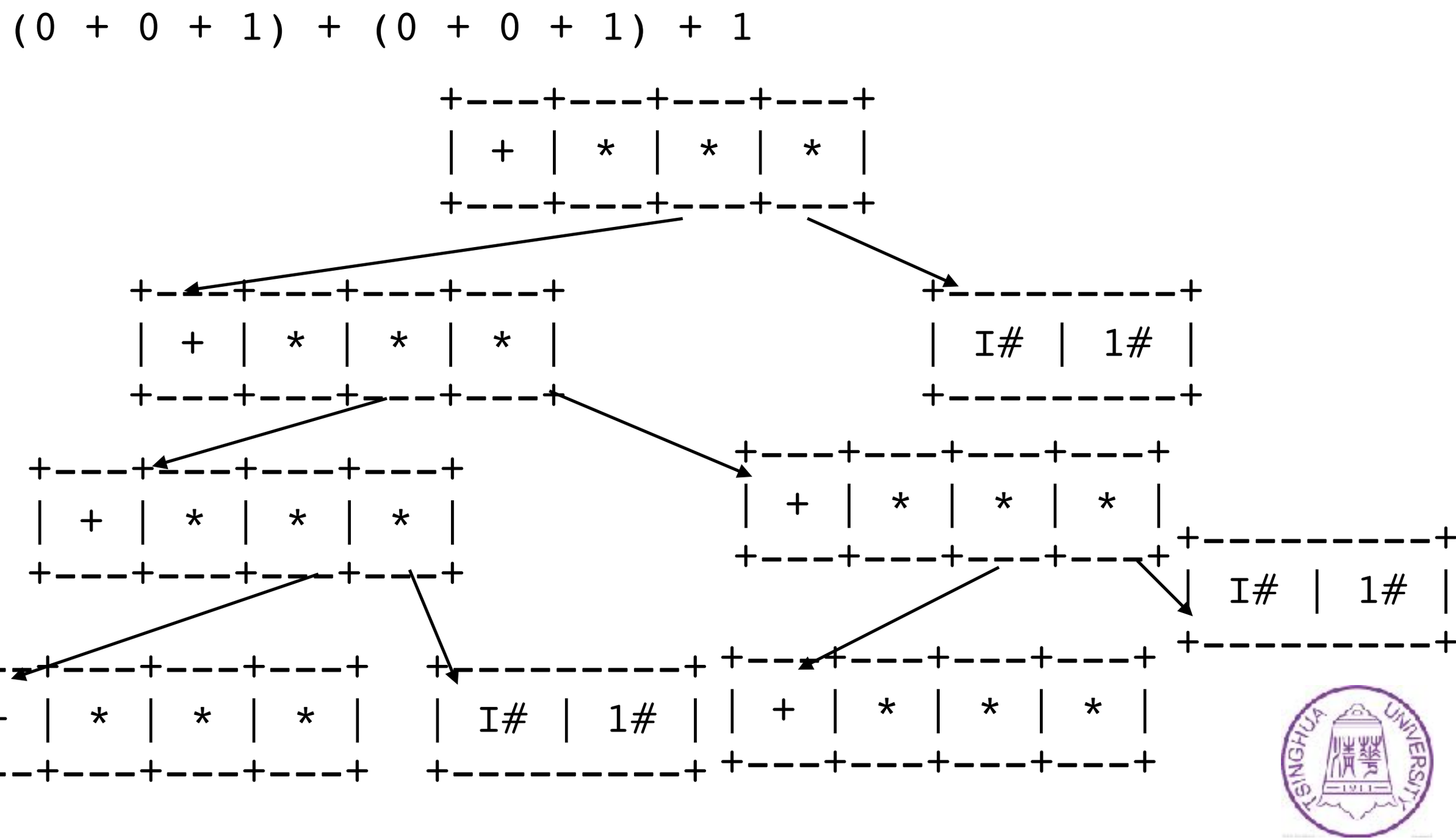


递归求解任务盒

```
(countNode Nil + countNode Nil + 1)
+ (countNode Nil + countNode Nil + 1)
+ 1
```

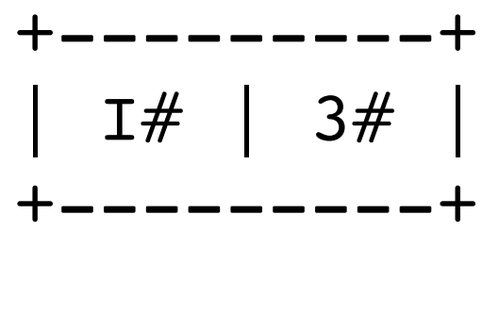


统一的内存表示



统一的内存表示

3



Thinking recursively

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

-- 对二叉树的结点求和
sumBinTree :: Num a => BinTree a -> a
sumBinTree = ?

-- 求二叉树的高度
binTreeHeight :: BinTree a -> Int
binTreeHeight = ?
```

Keep Invariant!

```
-- 递归结合数据结构的性质可以解决一些有趣的问题
-- 假如我们规定插入二叉树的节点保持比父节点 key 小的进左树
-- 否则进右树的性质

insertNode :: Ord k
           => BinTree (k, v)
           -> (k, v) -> BinTree (k, v)
insertNode
  (BinTree left right kv'@(k', _)) kv@(k, _)
    | k < k'  = BinTree (insertNode left kv) right kv'
    | k == k' = BinTree left right kv
    | k > k'  = BinTree left (insertNode right kv) kv'
insertNode Nil kv = BinTree Nil Nil kv
```

Keep Invariant!

```
-- 有了左右树和父节点大小的性质，递归搜索一个 k 就变得很简单

lookup :: Ord k => BinTree (k, v) -> k -> Maybe v
lookup k (Node left right (k',v))
      |   k < k' = lookup k left
      |  k == k' = Just v
      |   k > k' = lookup k right
lookup _ Nil = Nothing
```

递归的两种形式

```
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
```