操作链表的艺术

List 语法糖

```
[1..5]
-- [1,2,3,4,5]

[1,0..(-5)]
-- [1,0,-1,-2,-3,-4,-5]
```

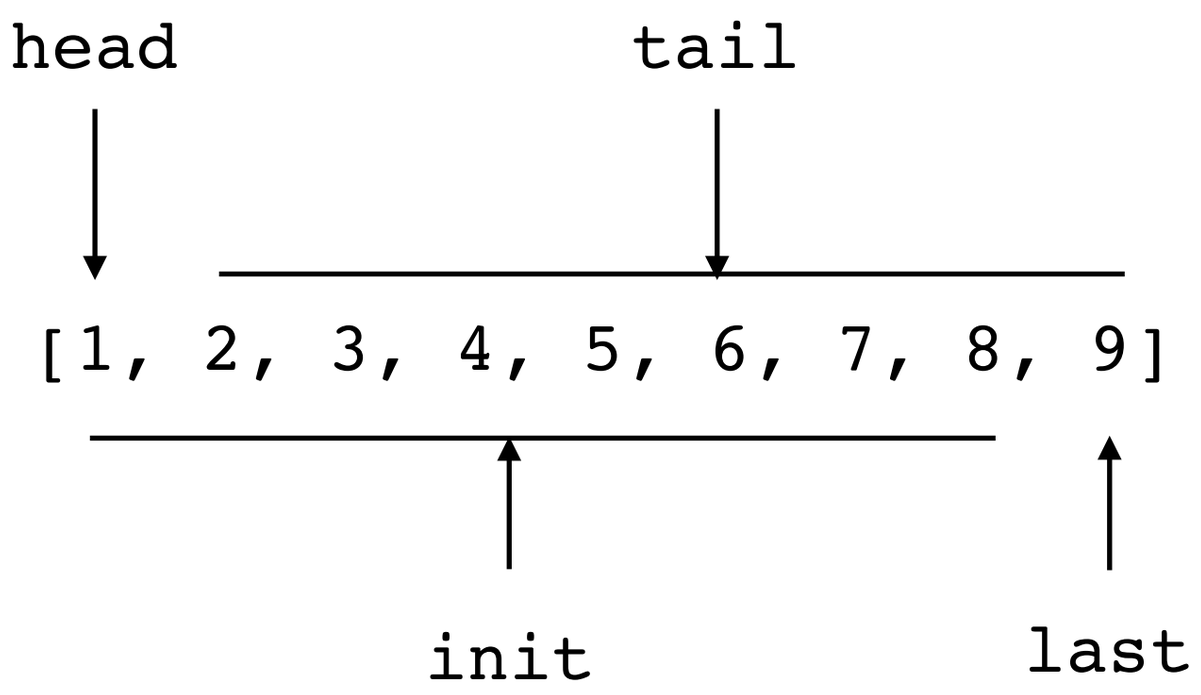
```
[1, 1.1 .. 2]
-- [1.0,1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.8,1.9,2.0]

[1,3..]
-- [1,3,5,7,9...]

[x*y | x <- [1,2,3], y <- [4,5,6]]
-- [4,5,6,8,10,12,12,15,18]
```

常见的 List 操作

```
head :: [a] -> a
head (x:_) = x
head []    = error "empty list"
```



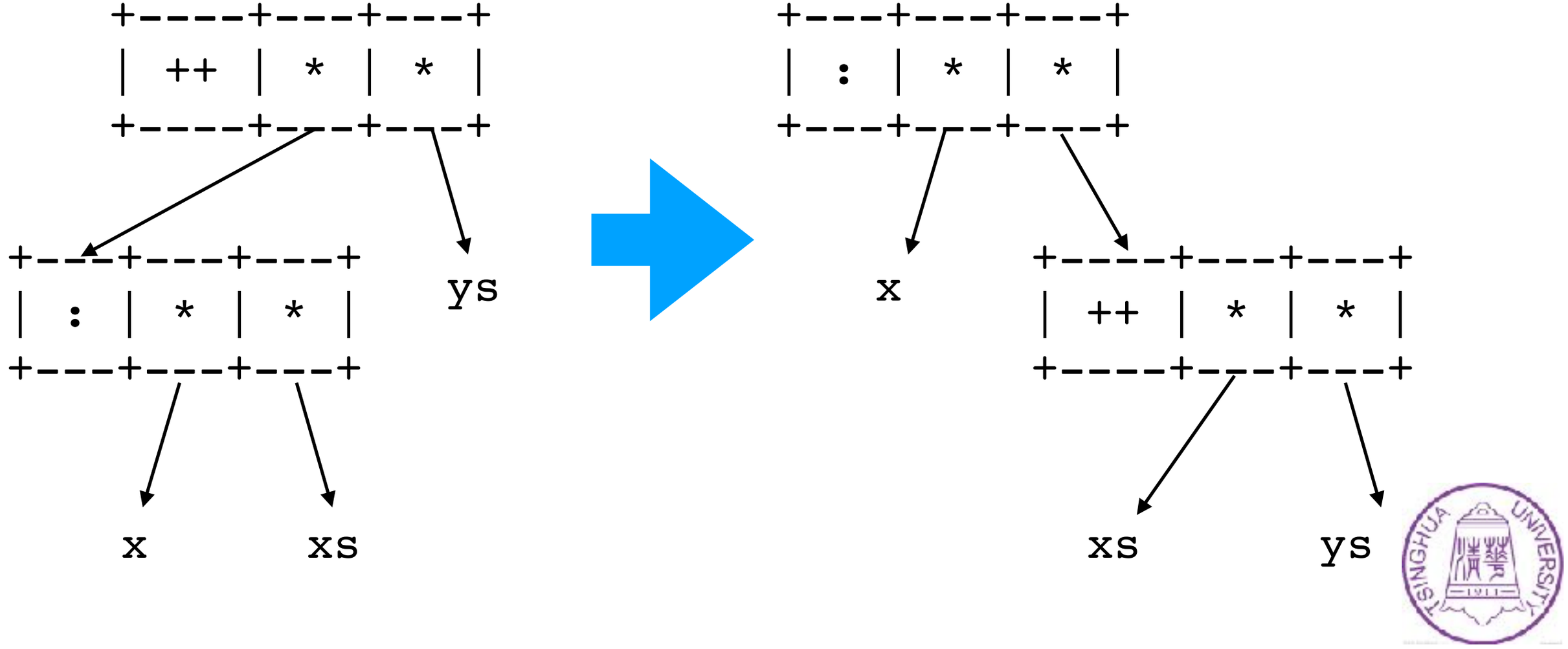
常见的 List 操作

```
drop :: Int -> [a] -> [a]
drop _ [] = []
drop n xs@(_:xs') | n <= 0     = xs
                  | otherwise = drop (n-1) xs'
```

```
take :: Int -> [a] -> [a]
take _ [] = []
take n (x:xs) | n <= 0     = []
              | otherwise = x : take (n-1) xs
```

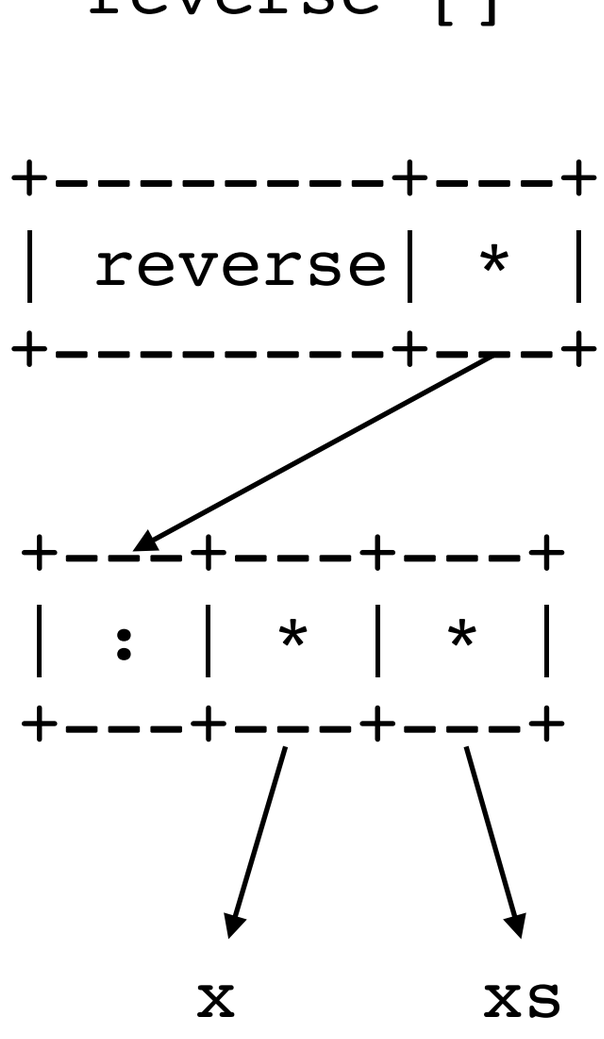
++ is lazy!

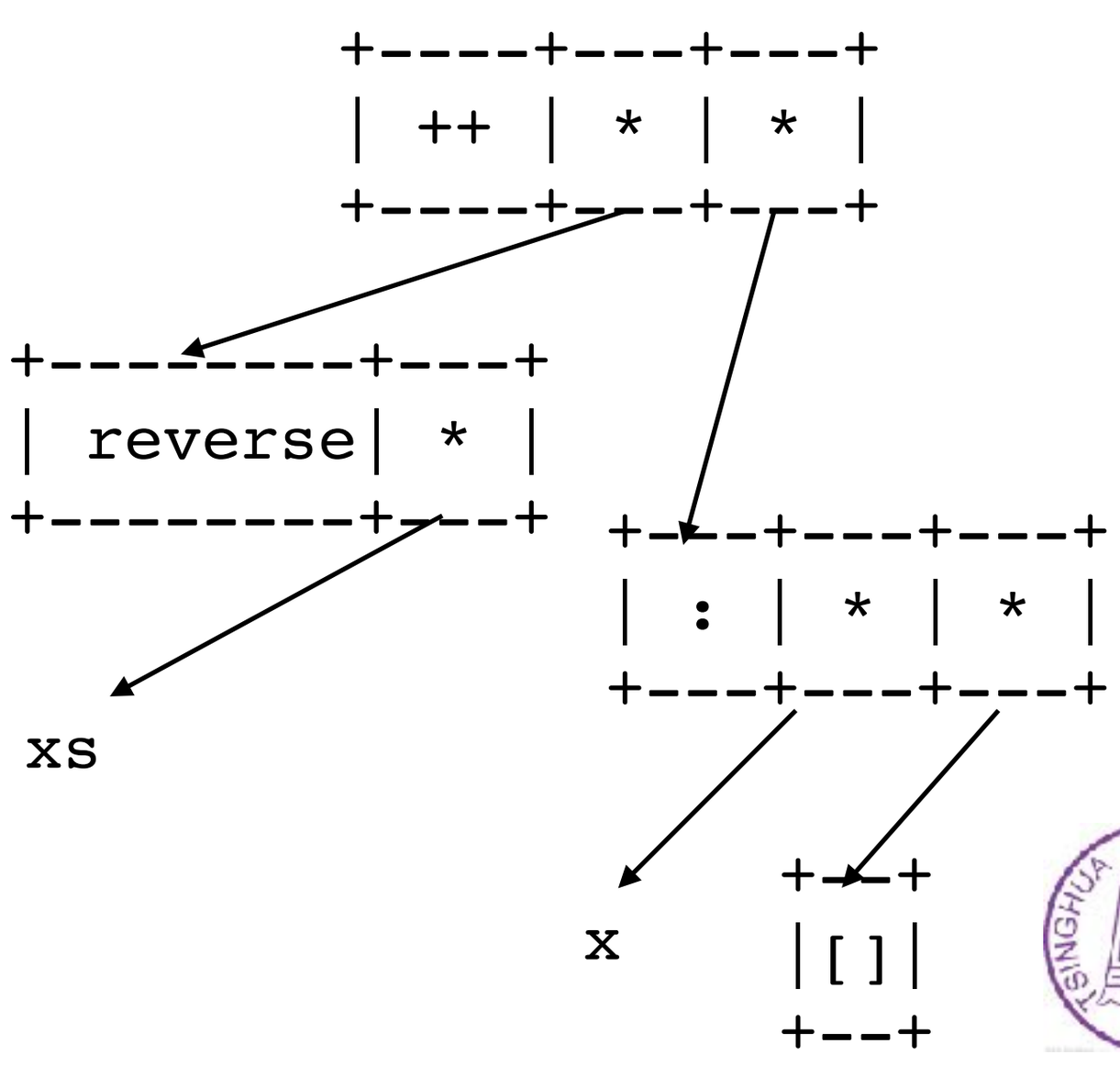
```
(++) :: [a] -> [a] -> [a]
(x:xs) ++ ys = x : (xs ++ ys)
[]     ++ ys = ys
```



reverse in O(n)?

```
  reverse :: [a] -> [a]
  reverse (x:xs) = reverse xs ++ [x]
  reverse []     = []
```





reverse in O(n)?

```
reverse :: [a] -> [a]
reverse xs = go xs []
  where go []     acc = acc
        go (x:xs) acc = go xs (x:acc)

-- 此处板书说到了 η 转换，但参数位置错了

-- reverse [1,2,3]
-- go (1:2:3:[])   []
-- go (2:3:[])   1:[]
-- go (3:[])   2:1:[]
-- go []     3:2:1:[]
-- [3,2,1]
```

map [a] to [b]

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

```
-- map (+1) (1:2:3:[])
-- (+1) 1 : map (+1) (2:3:[])
-- (+1) 1 : (+1) 2 : map (+1) (3:[])
-- (+1) 1 : (+1) 2 : (+1) 3 : map (+1) []
-- (+1) 1 : (+1) 2 : (+1) 3 : []
```

more list, more power

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith _ [] _ = []
zipWith _ _ [] = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

```
-- zipWith (+) [1,2,3] [4,5]
-- (+) 1 4 : zipWith (+) [2,3] [5]
-- (+) 1 4 : (+) 2 5 : zipWith (+) [3] []
-- (+) 1 4 : (+) 2 5 : []
```

merge sort

```
mergeSort :: Ord a => [a] -> [a]
mergeSort = ?

split :: [a] -> ([a], [a])
split = ?
```

fibonacci sequence

```
fib :: Int -> [Int]
fib 0 = 1
fib 1 = 1
fib x = fib (x-1) + fib (x-2)
```

```
fib :: [Int]
fib = ?
```

take half of a list

```
half :: [a] -> [a]
half = ?

const :: a -> b -> a
const x _ = x
```

```
zipWith const :: [a] -> [b] -> [a]

-- 回忆之前的split?
zipWith const xs (split xs)
```

zipper

```
data ListZipper a = ListZipper [a] [a]

fromList :: [a] -> ListZipper a
fromList xs = ListZipper [] xs

moveRight :: ListZipper a -> ListZipper a
moveRight lz@(ListZipper _ []) = lz
moveRight (ListZipper rs (x:xs)) = ListZipper (x:rs) xs

moveLeft :: ListZipper a -> ListZipper a
moveLeft lz@(ListZipper [] _) = Liz
moveLeft (ListZipper (r:rs) xs) = ListZipper rs (x:xs)

toList :: ListZipper a -> [a]
toList (ListZipper rs xs) = reverse rs ++ xs
```