

计算概论A—实验班

函数式程序设计

Functional Programming

胡振江，张 伟

北京大学 计算机学院

2022年09~12月

# 第7章：高阶函数

## High-order Function

# High-order Function

A function is called **higher-order** if it takes a function as an argument or returns a function as a result.

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)
```

- **twice** is higher-order,
  - because it takes a function as its first argument

# Why High-order Function

- ❖ Common programming idioms can be encoded as functions within the language itself.
- ❖ Domain specific languages can be defined as collections of higher-order functions.
- ❖ Algebraic properties of higher-order functions can be used to reason about programs.

# The `map` Function

- \* The higher-order library function called `map` applies a function to every element of a list.

```
map :: (a -> b) -> [a] -> [b]
```

```
ghci> map (+1) [1,2,3,4,5]  
[2,3,4,5,6]
```

# The `map` Function

- \* The `map` function can be defined in a particularly simple manner using a list comprehension:

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

- \* Alternatively, for the purposes of proofs, the `map` function can also be defined using recursion:

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

# The **filter** Function

- \* The higher-order library function **filter** selects every element from a list that satisfies a predicate.

```
filter :: (a -> Bool) -> [a] -> [a]
```

```
ghci> filter even [1..10]  
[2,4,6,8,10]
```

# The `filter` Function

\* `filter` can be defined using a list comprehension:

```
filter :: (a -> Bool) -> [a] -> [a]
map pred xs = [x | x <- xs, p x]
```

\* Alternatively, it can be defined using recursion:

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter pred (x:xs)
  | pred x = x : filter pred xs
  | otherwise = filter pred xs
```



# The `foldr` Function

这里的`foldr`，大概对应于前文引入的`foldl`



别人都叫`foldr`，你为什么还要叫`foldl`

我觉得，`foldl` 更美！

其实：Prelude中的`foldr`的抽象级别更高



# The **foldr** Function on Lists

- \* A number of functions on lists can be defined using the following simple pattern of recursion:

$$\begin{aligned} \mathbf{f} \ [] &= \mathbf{v} \\ \mathbf{f} \ (x:xs) &= x \oplus \mathbf{f} \ xs \end{aligned}$$

- **f** maps the empty list to some value **v**, and any non-empty list to some function  $\oplus$  applied to its head and **f** of its tail.

$$f [] = v$$

$$f (x:xs) = x \oplus f xs$$

\* For example:

$$\text{sum} [] = 0$$

$$\text{sum} (x:xs) = x + \text{sum} xs$$

$$\text{product} [] = 1$$

$$\text{product} (x:xs) = x * \text{product} xs$$

$$\text{and} [] = \text{True}$$

$$\text{and} (x:xs) = x \ \&\& \ \text{and} \ xs$$

# The **foldr** Function

- \* The higher-order library function **foldr** (fold right) encapsulates this simple pattern of recursion, with the function  $\oplus$  and the value **v** as arguments.
- \* For example:

```
sum      = foldr (+) 0
product = foldr (*) 1
or       = foldr (||) False
and      = foldr (&&) True
```

# The **foldr** Function

```
class Foldable t where
```

[# Source](#)

```
foldr :: (a -> b -> b) -> b -> t a -> b
```

[# Source](#)

Right-associative fold of a structure, lazy in the accumulator.

In the case of lists, **foldr**, when applied to a binary operator, a starting value (typically the right-identity of the operator), and a list, reduces the list using the binary operator, from right to left:

```
foldr f z [x1, x2, ..., xn] == x1 `f` (x2 `f` ... (xn `f` z)...) 
```

Note that since the head of the resulting expression is produced by an application of the operator to the first element of the list, given an operator lazy in its right argument, **foldr** can produce a terminating expression from an unbounded list.

For a general **Foldable** structure this should be semantically identical to,

```
foldr f z = foldr f z . toList
```

# The **foldr** on lists can be defined using recursion

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

- \* However, it is best to think of **foldr** *non-recursively*, as simultaneously replacing each **(:)** in a list by a given function, and **[]** by a given value.

# The **foldr** on lists: Examples

```
sum = foldr (+) 0
```

```
sum [1,2,3]
=
foldr (+) 0 [1,2,3]
=
foldr (+) 0 (1:(2:(3:[])))
=
1+(2+(3+0))
=
6
```

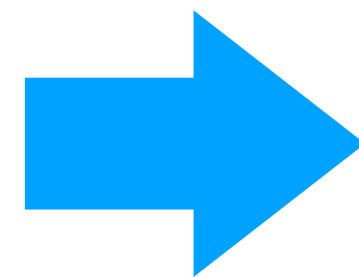
```
product = foldr (*) 1
```

```
product [1,2,3]
=
foldr (*) 1 [1,2,3]
=
foldr (*) 1 (1:(2:(3:[])))
=
1*(2*(3*1))
=
6
```

# The **foldr** on lists: Examples

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

```
length [1,2,3]
=
length (1:(2:(3:[])))
=
1+(1+(1+0))
=
3
```



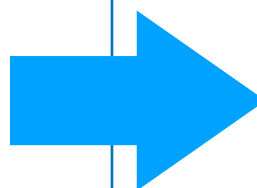
```
length :: [a] -> Int
length = foldr (\ _ n -> 1+n) 0
```



# The **foldr** on lists: Examples

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

```
reverse [1,2,3]
=
reverse (1:(2:(3:[])))
=
(([] ++ [3]) ++ [2]) ++ [1]
=
[3,2,1]
```



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

# The **foldr** on lists: Examples

- \* Finally, we note that the append function **(++)** has a particularly compact definition using **foldr**:

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

遗憾的是：Haskell似乎不支持这种定义方式  
“error: Parse error in pattern: ++ys”

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

```
(++) :: [a] -> [a] -> [a]
(++) = flip $ foldr (:)
```

# Why foldr

- \* Some recursive functions on lists, such as `sum`, are simpler to define using `foldr`.
- \* Properties of functions defined using `foldr` can be proved using algebraic properties of `foldr`, such as *fusion* and the *banana split* rule.
- \* Advanced program optimizations can be simpler if `foldr` is used in place of explicit recursion.

# The **foldl** Function on Lists

- \* It is also possible to define recursive functions on lists using an operator that is assumed to **associate to the left**.

$$\begin{aligned} f \ v \ [] &= v \\ f \ v \ (x:xs) &= f \ (v \oplus x) \ xs \end{aligned}$$

- **f** maps the empty list to the *accumulator* value **v**, and any non-empty list to the result of recursively processing the tail using a new accumulator value obtained by applying an operator  $\oplus$  to the current value and the head of the list.

# The **foldl** Function on Lists

\* **foldl** on lists itself can be defined using recursion:

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f v [] = v
foldl f v (x:xs) = foldl f (f v x) xs
```

# The `foldl` Function

```
class Foldable t where
```

[# Source](#)

```
foldl :: (b -> a -> b) -> b -> t a -> b
```

[# Source](#)

Left-associative fold of a structure, lazy in the accumulator. This is rarely what you want, but can work well for structures with efficient right-to-left sequencing and an operator that is lazy in its left argument.

In the case of lists, `foldl`, when applied to a binary operator, a starting value (typically the left-identity of the operator), and a list, reduces the list using the binary operator, from left to right:

```
foldl f z [x1, x2, ..., xn] == (...((z `f` x1) `f` x2) `f` ...) `f` xn
```

Note that to produce the outermost application of the operator the entire input list must be traversed. Like all left-associative folds, `foldl` will diverge if given an infinite list.

If you want an efficient strict left-fold, you probably want to use `foldl'` instead of `foldl`. The reason for this is that the latter does not force the *inner* results (e.g. `z `f` x1` in the above example) before applying them to the operator (e.g. to `(`f` x2)`). This results in a thunk chain  $\mathcal{O}(n)$  elements long, which then must be evaluated from the outside-in.

For a general `Foldable` structure this should be semantically identical to:

```
foldl f z = foldl f z . toList
```

# Other Library Functions: `(.)`

- \* The library function `(.)` returns the composition of two functions as a single function.

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

- \* For example:

```
odd :: Int -> Bool  
odd = not . even
```

# Other Library Functions: **all**

```
all :: Foldable t => (a -> Bool) -> t a -> Bool
```

Determines whether all elements of the structure satisfy the predicate.

\* **all** on lists can be defined as

```
all :: (a -> Bool) -> [a] -> Bool  
all p xs = and [p x | x <- xs]
```



# Other Library Functions: **any**

```
any :: Foldable t => (a -> Bool) -> t a -> Bool
```

Determines whether any element of the structure satisfies the predicate.

\* **any** on lists can be defined as

```
any :: (a -> Bool) -> [a] -> Bool  
any p xs = or [p x | x <- xs]
```

# Other Library Functions: **takeWhile**

- \* The library function **takeWhile** selects elements from a list while a predicate holds of all the elements.

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile _ [] = []
takeWhile p (x:xs)
  | p x = x : takeWhile p xs
  | otherwise = []
```

```
ghci> takeWhile (/= ' ') "abc def"
"abc"
```

# Other Library Functions: **dropWhile**

- \* Dually, the function **dropWhile** removes elements while a predicate holds of all the elements.

```
dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile _ [] = []
dropWhile p xs@(x:xs')
    | p x = dropWhile p xs'
    | otherwise = xs
```

```
ghci> dropWhile (== ' ') " abc"
"abc"
```

# 应用1: Binary String Transmitter

## \* 2进制数 转换到 10进制数

```
ghci> bin2int [1,0,1,1]
13
```

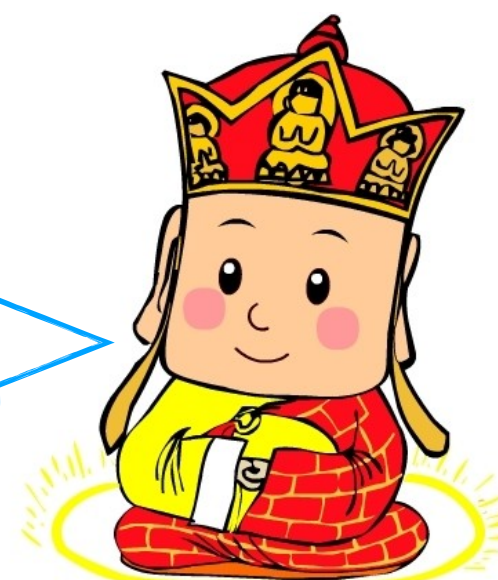
```
type Bit = Int

bin2int :: [Bit] -> Int
bin2int bits = sum [w * b | (w, b) <- zip weights bits]
  where weights = iterate (* 2) 1

-- iterate is defined in Prelude
-- iterate :: (a -> a) -> a -> [a]
-- iterate f x = x : iterate f (f x)
```

```
bin2int :: [Bit] -> Int
bin2int = foldr (\x y -> x + 2 * y) 0
```

还有更简洁的定义方式吗



# 应用1: Binary String Transmitter

\* 10进制数 转换到 8位2进制数

```
ghci> int2bin8 13  
[1,0,1,1,0,0,0,0]
```

```
int2bin :: Int -> [Bit]  
int2bin 0 = []  
int2bin n = mod n 2 : int2bin (div n 2)
```

```
make8 :: [Bit] -> [Bit]  
make8 bits = take 8 $ bits ++ repeat 0  
-- repeat is defined in Prelude  
-- repeat :: a -> [a]  
-- repeat x = xs where xs = x : xs
```

```
int2bin8 :: Int -> [Bit]  
int2bin8 = make8 . int2bin
```

# 应用1: Binary String Transmitter

## \* 文字序列编码

```
ghci> encode "abc"  
[1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0,1,1,0,0,0,1,1,0]
```

```
encode :: String -> [Bit]  
encode = concat . map (make8 . int2bin . ord)
```

# 应用1: Binary String Transmitter

## \* 2进制序列解码

```
ghci> decode [1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0,1,1,0,0,0,1,1,0]
"abc"
```

```
decode :: [Bit] -> String
```

```
decode = map (chr . bin2int) . chop8
```

```
chop8 :: [Bit] -> [[Bit]]
```

```
chop8 [] = []
```

```
chop8 bits = take 8 bits : chop8 (drop 8 bits)
```

# 应用2: 投票算法 之 First past the post

\* In this system, each person has one vote, and the candidate with the largest number of votes is declared the winner.

```
votes :: [String]
votes = ["Red", "Blue", "Green", "Blue", "Blue", "Red"]
```

```
ghci> result votes
[(1,"Green"),(2,"Red"),(3,"Blue")]
ghci> :type result
result :: Ord a => [a] -> [(Int, a)]
```

```
ghci> winner votes
"Blue"
ghci> :type result
result :: Ord a => [a] -> [(Int, a)]
```



# 应用2: 投票算法 之 First past the post

```
votes :: [String]
votes = ["Red", "Blue", "Green", "Blue", "Blue", "Red"]

result :: Ord a => [a] -> [(Int, a)]
result vs = sort [ (count v vs, v) | v <- rmdups vs ]
-- The sort function is defined in Data.List

rmdups :: Eq a => [a] -> [a]
rmdups [] = []
rmdups (x:xs) = x : filter (/= x) (rmdups xs)

count :: Eq a => a -> [a] -> Int
count x = length . filter (== x)

winner :: Ord a => [a] -> a
winner = snd . last . result
```

# 应用2: 投票算法 之 **Alternative vote**

- \* In this voting system, each person can vote for as many or as few candidates as they wish, listing them in preference order on their ballot (1st choice, 2nd choice, and so on).

```
ballots :: [[String]]
ballots = [ ["Red", "Green"],
            ["Blue"],
            ["Green", "Red", "Blue"],
            ["Blue", "Green", "Red"],
            ["Green"] ]
```

```
ghci> winner' ballots
"Green"
ghci> :type winner'
winner' :: Ord a => [[a]] -> a
```

# 应用2: 投票算法 之 **Alternative vote**

- \* To decide the winner,
  - any empty ballots are first removed,
  - then the candidate with the smallest number of 1st-choice votes is eliminated from the ballots,
  - and same process is repeated until only one candidate remains, who is then declared the winner.

```
ballots :: [[String]]
ballots = [ ["Red", "Green"],
             ["Blue"],
             ["Green", "Red", "Blue"],
             ["Blue", "Green", "Red"],
             ["Green"] ]
```

# 应用2: 投票算法 之 **Alternative vote**

- \* To decide the winner,
  - any empty ballots are first removed,
  - then the candidate with the smallest number of 1st-choice votes is eliminated from the ballots,
  - and same process is repeated until only one candidate remains, who is then declared the winner.

```
ballots :: [[String]]
ballots = [ ["Red", "Green"],
            ["Blue"],
            ["Green", "Red", "Blue"],
            ["Blue", "Green", "Red"],
            ["Green"] ]
```

# 应用2: 投票算法 之 **Alternative vote**

- \* To decide the winner,
  - any empty ballots are first removed,
  - then the candidate with the smallest number of 1st-choice votes is eliminated from the ballots,
  - and same process is repeated until only one candidate remains, who is then declared the winner.

```
ballots :: [[String]]
ballots = [ ["Green"],
            ["Blue"],
            ["Green", "Blue"],
            ["Blue", "Green"],
            ["Green"] ]
```

# 应用2: 投票算法 之 **Alternative vote**

- \* To decide the winner,
  - any empty ballots are first removed,
  - then the candidate with the smallest number of 1st-choice votes is eliminated from the ballots,
  - and same process is repeated until only one candidate remains, who is then declared the winner.

```
ballots :: [[String]]
ballots = [ ["Green"],
             ["Blue"],
             ["Green", "Blue"],
             ["Blue", "Green"],
             ["Green"] ]
```

# 应用2: 投票算法 之 **Alternative vote**

- \* To decide the winner,
  - any empty ballots are first removed,
  - then the candidate with the smallest number of 1st-choice votes is eliminated from the ballots,
  - and same process is repeated until only one candidate remains, who is then declared the winner.

```
ballots :: [[String]]
ballots = [ ["Green"],
            [],
            ["Green"],
            ["Green"],
            ["Green"] ]
```

# 应用2: 投票算法 之 **Alternative vote**

- \* To decide the winner,
  - any empty ballots are first removed,
  - then the candidate with the smallest number of 1st-choice votes is eliminated from the ballots,
  - and same process is repeated until only one candidate remains, who is then declared the winner.

```
ballots :: [[String]]
ballots = [ ["Green"],
            [],
            ["Green"],
            ["Green"],
            ["Green"] ]
```



# 应用2: 投票算法 之 **Alternative vote**

- \* To decide the winner,
  - any empty ballots are first removed,
  - then the candidate with the smallest number of 1st-choice votes is eliminated from the ballots,
  - and same process is repeated until only one candidate remains, who is then declared the winner.

```
ballots :: [[String]]
ballots = [ ["Green"],
             ["Green"],
             ["Green"],
             ["Green"] ]
```

## 应用2: 投票算法 之 **Alternative vote**

```
winner' :: Ord a => [[a]] -> a
winner' bs = case rank $ filter (/= []) bs of
  [c]      -> c
  (c:cs)   -> winner' $ map (filter (/= c)) bs

rank :: Ord a => [[a]] -> [a]
rank = map snd . result . map head
```

# 作业

# 作业

7-1 Express the comprehension  $[f\ x \mid x \leftarrow xs, p\ x]$  using the functions `map` and `filter`.

7-2 Redefine `map f` and `filter p` using `foldr`.

# 作业

- 7-3 Modify the binary string transmitter example to **detect simple transmission errors** using the concept of parity bits.
- ▶ That is, each eight-bit binary number produced during encoding is extended with a parity bit,
    - set to one if the number contains an odd number of ones, and to zero otherwise.
  - ▶ In turn, each resulting nine-bit binary number consumed during decoding is checked to ensure that its parity bit is correct, with the parity bit being discarded if this is the case, and a parity error being reported otherwise.

Hint: the library function `error :: String -> a` displays the given string as an error message and terminates the program; the polymorphic result type ensures that `error` can be used in any context.

# 第7章：高阶函数

## High-order Function

**就到这里吧**