Adapted from Graham's Lecture slides.

第七章：高阶函数

基本概念

处理序列的常用高阶函数，foldr/foldl

两个应用问题

Introduction

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 Are They Useful?

- 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]
```

For example:

```
> map (+1) [1,3,5,7]

[2,4,6,8]
```

The map function can be defined in a particularly simple manner using a list comprehension:

```
map f xs = [f x | x ← xs]
```

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

```
map f []     = []
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]
```

For example:

```
> filter even [1..10]

[2,4,6,8,10]
```

Filter can be defined using a list comprehension:

```
filter p xs = [x | x ← xs, p x]
```

Alternatively, it can be defined using recursion:

```
filter p [] = []
filter p (x:xs)
   | p x       = x : filter p xs
   | otherwise = filter p xs
```

The Foldr Function

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

```
f []     = v
f (x:xs) = x ⊕ f xs
```

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

For example:

```
sum []     = 0
sum (x:xs) = x + sum xs
```

$v = 0$

$\oplus = +$

```
product []     = 1
product (x:xs) = x * product xs
```

$v = 1$

$\oplus = *$

```
and []     = True
and (x:xs) = x && and xs
```

$v = \text{True}$

$\oplus = \&\&$

The higher-order library function foldr (fold right) encapsulates this simple pattern of recursion, with the function ⊕ and the value v as arguments.

For example:

```
sum = foldr (+) 0

product = foldr (*) 1

or = foldr (||) False

and = foldr (&&) True
```

Foldr itself 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.

For example:

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

Replace each (:) by (+) and [] by 0.

For example:

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

Replace each (:) by (*) and [] by 1.

Other Foldr Examples

Even though foldr encapsulates a simple pattern of recursion, it can be used to define many more functions than might first be expected.

Recall the length function:

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

For example:

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

Replace each
(:) by λ _ n → 1+n, and
[] by 0.

Hence, we have:

```
length = foldr (λ_ n → 1+n) 0
```

Now recall the reverse function:

```
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

For example:

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

Replace each
(:) by λx xs $\rightarrow$ xs ++ [x], and
[] by [].

Hence, we have:

```
reverse = foldr (λx xs → xs ++ [x]) []
```

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

```
(++ ys) = foldr (:) ys
```

Replace each (:) by (:), and [] by ys.

Why Is Foldr Useful?

- 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 optimisations can be simpler if foldr is used in place of explicit recursion.

The Foldl Function

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

```
f v []     = v
f v (x:xs) = f (v ⊕ x) xs
```

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 ⊕ to the current value and the head of the list.

Foldl itself can be defined using recursion:

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

It is a tail recursion, like a loop.

Other Library Functions

The library function (.) returns the composition of two functions as a single function.

```
(.) :: (b → c) → (a → b) → (a → c)
f . g = λx → f (g x)
```

For example:

```
odd :: Int → Bool
odd = not . even
```

The library function <u>all</u> decides if every element of a list satisfies a given predicate.

```
all :: (a → Bool) → [a] → Bool
all p xs = and [p x | x ← xs]
```

For example:

```
> all even [2,4,6,8,10]

True
```

Dually, the library function <u>any</u> decides if at least one element of a list satisfies a predicate.

```
any :: (a → Bool) → [a] → Bool
any p xs = or [p x | x ← xs]
```

For example:

```
> any (== ' ') "abc def"

True
```

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

```
takeWhile :: (a → Bool) → [a] → [a]
takeWhile p [] = []
takeWhile p (x:xs)
    | p x       = x : takeWhile p xs
    | otherwise = []
```

For example:

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

"abc"
```

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

```
dropWhile :: (a → Bool) → [a] → [a]
dropWhile p [] = []
dropWhile p (x:xs)
    | p x          = dropWhile p xs
    | otherwise = x:xs
```

For example:

```
> dropWhile (== ’ ’) "   abc"

"abc"
```

应用1: Binary String Transmitter

- 2进制到10进制的转换

  > 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 f x = [x, f x, f (f x), f (f (f x)), ...]
-- iterate f x = x : iterate f (f x)
-- defined in prelude
```

```
bin2int = foldr (\x y -> x + 2*y) 0
```

应用1: Binary String Transmitter

- 10进制数字到8位2进制的转换

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

```
make8 :: [Bit] -> [Bit]
make8 bits = take 8 (bits ++ repeat 0)
```

```
Int2bin8 :: Int -> [Bin]
Int2bin8 = make8 . int2bin
```

应用1: Binary String Transmitter

- 文字序列编码

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

```
> 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]
```

应用1: Binary String Transmitter

- 2进制序列解码

```
decode :: [Bit] -> String
decode = map (chr . bin2int) . chop8
```

```
chop8 :: [Bit] -> [Bit]
chop8 bits = ...?
```

```
> 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"
```

应用2: 投票算法 (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"]
```

```
> result votes
[(1,"Green"), (2,"Red"), (3,"Blue")]

> winner votes
"Blue"
```

应用2: 投票算法

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

```
result :: Ord a => [a] -> [(Int,a)]
result vs = sort [(count v vs, v) | v <- rmdups vs]

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)
```

应用2: 投票算法 (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"]]
```

应用2: 投票算法

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

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

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

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.

```
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

7-1 Express the comprehension [f x | x ← 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.