第 07 章：Recursive Function

✧ Function

As we have seen, many functions can naturally be defined in terms of other functions.

```
fac :: Int -> Int

fac n   =   product [1..n]
```

✧ Recursive Function / 递归函数

In Haskell, functions can also be defined in terms of themselves. Such functions are called **recursive.**

```
fac :: Int -> Int

fac 0 = 1

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

✧ **Why** Recursive Function ？

- Some functions, such as factorial, are simpler to define in terms of other functions.
- As we shall see, however, many functions can naturally be defined in terms of themselves.
- Properties of functions defined using recursion can be proved using the simple but powerful mathematical technique of induction.

✧ Recursive Function on List

Recursion is not restricted to numbers, but can also be used to define functions on lists.

```
product :: Num a => [a] -> a
product []  =  1
product (n:ns)  =  n * product ns
```

Using the same pattern of recursion as in product we can define the length function on lists.

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

Using a similar pattern of recursion we can define the reverse function on lists.

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

✧ Example: 插入排序

```
isort :: Ord a => [a] -> [a]
isort []  =  []
isort (x:xs)  =  insert x (isort xs)


insert :: Ord a => a -> [a] -> [a]
insert x []  =  [x]
insert x (y:ys) | x <= y  =  x:y:ys
                | otherwise  =  y:(insert x ys)
```

✧ 多参数递归

Functions with more than one argument can also be defined using recursion.

Example: Zipping the elements of two lists

```
zip :: [a] -> [b] -> [(a,b)]

zip [] _  =  []

zip _ []  =  []

zip (x:xs) (y:ys)  =  (x,y) : zip xs ys
```

Example: Remove the first n elements from a list

```
drop :: Int -> [a] -> [a]

drop 0 xs  =  xs

drop _ []  =  []

drop n (_:xs)  =  drop (n-1) xs
```

Example: Appending two lists

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

✧ Multiple Recursion

Functions can also be defined using multiple recursion, in which a function is applied more than once in its own definition.

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n - 2) + fib (n - 1)
```

```
qsort :: Ord a => [a] -> [a]
qsort []  =  []
qsort (x:xs)  =  qsort smaller ++ [x] ++ qsort larger
  where
    smaller = [a | a <- xs, a <= x]
    larger  = [b | b <- xs, b >  x]
```

```
     qsort [3,2,4,1,5]
=== qsort [2,1]                   ++  [3]  ++  qsort [4,5]
=== qsort [1] ++ [2] ++ qsort [] ++  [3]  ++  qsort [] ++ [4] ++ qsort [5]
===       [1] ++ [2] ++       [] ++  [3]  ++        [] ++ [4] ++       [5]
```

✧ Mutual Recursion

Functions can also be defined using mutual recursion, in which two or more functions are all defined recursively in terms of each other.

```
even :: Int -> Bool
even 0 = True
even n = odd (n-1)

odd :: Int -> Bool
odd 0 = False
odd n = even (n-1)
```

作业 01

Without looking at the standard prelude, define the following library functions using recursion:

1. Decide if all logical values in a list are true

   `and :: [Bool] -> Bool`

2. Concatenate a list of lists

   `concat :: [[a]] -> [a]`

3. Select the nth element of a list (starting from 0)

   `(!!) :: [a] -> Int -> a`

4. Produce a list with n identical elements

   `replicate :: Int -> a -> [a]`

5. Decide if a value is an element of a list

   `elem :: Eq a => a -> [a] -> Bool`

作业 02

Define a recursive function

`merge :: Ord a => [a] -> [a] -> [a]`

that merges two sorted lists of values to give a single sorted list.

For example:

```
ghci> merge [2,5,6] [1,3,4]
[1,2,3,4,5,6]
```

作业 03

Define a recursive function

```
msort :: Ord a => [a] -> [a]
```

that implements merge sort, which can be specified by the following two rules:

A. Lists of length <= 1 are already sorted;

B. Other lists can be sorted by sorting the two halves and merging the resulting lists