Haskell 筆記

副檔名 .hs　　軟體安裝：https://www.haskell.org/platform/

使用環境：
GHCi, version 8.2.2: http://www.haskell.org/ghc/ :? for help

進入cmd line, ghci 或是 ghc [input_file]

:l baby.sh

負數要加括號 5*(-3)

succ 9 + max 5 4 + 1 -> 9++ 加上 5 加上1

＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝
第二章
用let 宣告變樹
　let lostNumbers = [4,8,15,16,23,42]
用++ operator可以連接資料
　[1,2,3,4] ++ [9,10,11,12]
　＝>[1,2,3,4,9,10,11,12]
用： operator也可以連接資料
　5:[1,2,3,4,5]
　＝>[5,1,2,3,4,5]
用！！ 讀取list內的element
　[9.4,33.2,96.2,11.2,23.25] !! 1
　=>33.2
用><=比較大小：會用lexicographical order比較
　lexicographical order：按照字典順序排序 (第一個字相等的話就比較第二個字，比到最後不一樣長的話短　的放前面)
用head/ tail/ last/ init 選擇list的第一個/扣掉第一個/ 最後一個/ 扣掉最後一個 element
用length 得到list 長度 用null判斷是否為null list
用reverse 反轉list
用take n 選擇list前n個element
用drop n 把list的前n-1個element忽略 (從第n個開始)
用minimum/ maximum/ sum/ product 選擇list的最大/ 最小element，計算list的總和/ 乘積
用n elem 指令判斷n是否為list中的一個member
用[1..n]建立1~n的range:
　[1..20] => [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
　['a'..'z'] => “abcdefghijklmnopqrstuvwxyz”
　[3,6..20] => [3,6,9,12,15,18]
　要建立20~1的list要用[20,19..1] 不能用 [20..1]
用cycle [list]建立一個list的循環
用repeat [n]建立一個element的循環
　注意這兩個都沒有停止條件，要用take (提取前n個element)
　take 10 (cycle [1,2,3])
　=>[1,2,3,1,2,3,1,2,3,1]

take 10 (repeat 5)
=>[5,5,5,5,5,5,5,5,5,5]
用replicate也可以
replicate 3 10
=> [10,10,10]
另外一種建立list的方法：用domain define的方式
[x*2 | x <- [1..10]] (x 等於 1~10之中的數，乘以二)
=>[2,4,6,8,10,12,14,16,18,20]
[x | x <- [50..100], x `mod` 7 == 3] (x 等於在 50~100之中，mod 7會等於3的數)
=>[52,59,66,73,80,87,94]
boomBangs xs = [if x < 10 then "BOOM!" else "BANG!" | x <- xs, odd x] 輸入list之中，只挑選小於10的奇數,
=> boomBangs [7..13]
=> ["BOOM!","BOOM!","BANG!","BANG!"]
[x | x <- [10..20], x /= 13, x /= 15, x /= 19] 10到20之中 不等於13 15 19
=>[10,11,12,14,16,17,18,20]
let nouns = ["hobo","frog","pope"]
let adjectives = ["lazy","grouchy","scheming"]
[adjective ++ " " ++ noun | adjective <- adjectives, noun <- nouns] 所有形容詞跟名詞的組合
=>["lazy hobo","lazy frog","lazy pope","grouchy hobo","grouchy frog",
"grouchy pope","scheming hobo","scheming frog","scheming pope"]
length' xs = sum [1 | _ <- xs]
_ (底線)代表一個參數我們之後也不會用到，不在意它的名稱
length' 代表的意義：把輸入xs的每一個element都用1取代，再sum就是這個list的長度了
removeNonUppercase st = [c | c <- st, c `elem` ['A'..'Z']] 輸入st當中 elem [A..Z]的element放到c
removeNonUppercase "Hahaha! Ahahaha!"
=>"HA"

tuple(元組，當你想要把several value存成一個pair，許多pair來組成一個list的時候)
[("Christopher", "Walken", 55), ("alumi","yu",20)] => ok
[("Christopher", "Walken", 55), ("alumi","yu",)] => error
tuple之間也可以比較大小 (但是要size一樣大才能比較)
list之間不需要一樣size也可以比較大小
用fst指令可以選擇pair的第一個element
fst (8,11)
=>8
用snd指令可以選擇pair的第二個element
snd (8,11)
=>11
用zip指令可以把兩個list變成一個list，合成的list的element都是兩個list組成的pair
=>zip [1 .. 5] ["one", "two", "three", "four", "five"]
=>[(1,"one"),(2,"two"),(3,"three"),(4,"four"),(5,"five")]
＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝
小結：
如果想要找一個三角刑，三邊長都小於10，周長等於24
let rightTriangles' = [(a,b,c) | c <- [1..10], b <- [1..c], a <- [1..b], a^2 + b^2 == c^2, a+b+c == 24]
=>rightTriangles'
=>[(6,8,10)]
＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝
第三章

資料形態：可以用 :t [x] 去查詢x的資料形態
return 的時候會看到 ： ： 意思是has a type of
=>:t "HELLO!”
=>"HELLO!" :: [Char]
為function宣告形態
=>addThree :: Int -> Int -> Int -> Int 用->分隔 最後一個是return 前面都是input parameter
=>addThree x y z = x + y + z
資料形態跟常見的差不多 int 跟 integer的差別注意一下
一個有趣的地方，去查function head的型態：
(head 是從input list裡面return 第一個element，預期應該是char->char)
=>:t head
=>head :: [a] -> a ??為什麼是a?應該不是一個型態
=> a 是一個type variable，就是a可以當成任何型態
=> 使用這種type variable的function就是polymorphic functions，很好用
=> 所以head就是讀入任何一種形態，然後回傳那個形態的第一個element

typeclass: function的行為定義
Eq
:t (==)
(==) :: (Eq a) => a -> a -> Bool
看到=>這個符號表示：class constraint
==這個function就是兩個型態為a(兩個要一樣)的parameter，output一個boolean

:t (elem)
=>(elem) :: (Eq a, Foldable t) => a -> t a -> Bool
因為elem是逐一比較list當中的element和輸入的element是否相等，同上，list內的element和比較的element要相等
Order
:t (>)
(>) :: (Ord a) => a -> a -> Bool
==這個function就是兩個型態為a(兩個要一樣)的parameter，output一個boolean

也可以用’compare
5 `compare` 3
=>GT
Show/ Read
*Main> :t show
show :: Show a => a -> String
讀入任意形態，return為show的member，這個member是一個string
*Main> :t read
read :: Read a => String -> a
讀入一個string，return為任意形態的char (注意由於read不知道到底要return什麼形態，會根據你之後的運算決定)
所以如果你把read的直拿來運算就ＯK
read "5" - 2
=>3 此時read的return值是一個int
但是如果你
read “5"
=> exception read不知道要return什麼形態
像read這種沒有指定return type的function我們叫做type annotations

如果沒有運算，也可以用：：來指定回傳型態
ghci> read "5" :: Int
=>5
ghci> read "5" :: Float
=>5.0
Enum 可以把list連續輸出
*Main> succ ‘B'
=>‘C'
*Main> pred ‘B'
=>'A'
Bound 可以知道上下界
minBound :: Int
=> -2147483648
minBound :: Bool
=> False
也可以用在tuple
maxBound :: (Bool, Int, Char)
=>(True,2147483647,'\1114111’)
Num
注意這種形態的運動
比方說你在使用(5 :: Int) * (6 :: Integer)的時候就會出現error
或是你想要把一個list的length加上float的時候也會出現error
length [1,2,3,4] + 3.2
這個時候可以用一個很好用的fromIntegra，把形態轉成type variable
fromIntegral (length [1,2,3,4]) + 3.2
=> 7.2
＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝
小結：
:t 可以看結構
注意型態
＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝
第四章
function的語法
用pattern分開function行為 (一個pattern對應一個body，是不同的function行為)
sayMe :: (Integral a) => a -> String
sayMe 1 = "One!”
sayMe 2 = "Two!”
sayMe 3 = "Three!”
sayMe 4 = "Four!”
sayMe 5 = "Five!”
sayMe x = "Not between 1 and 5”
要記得寫exception case，但是expection case不要寫在最前面不然依序執行會每次都跑進去
練習重寫之前的數列乘積
factorial :: (Integral a) => a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)
一個比較奇特的pattern寫法：
let xs = [(1,3), (4,3), (2,4), (5,3), (5,6), (3,1)] xs是一個pari的list
ghci> [a+b | (a,b) <- xs] 把input xs的每一個pair都加總
=>[4,7,6,8,11,4] output是一個list

有另外一種pattern寫法：
xs@(x:y:ys) 把xs用：隔開

Guard: 用來判斷statement是否成立 (類似if else)
用”|"之後的一個判斷是來決定，true的話就跑進body，false的話就跑下一個guard
在最後面可以用where 來補充變數定義，例如
```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | bmi <= skinny = "You're underweight, you emo, you!"++show((fromRational(3.14)))
    | bmi <= normal = "You're supposedly normal. Pffft, I bet you're ugly!”
    | bmi <= fat = "You're fat! Lose some weight, fatty!”
    | otherwise                 = "You're a whale, congratulations!”
    where bmi = weight/height^2
          (skinny, normal, fat) = (18.5, 25.0, 30.0)
```
注意最後的空格對齊，不要用tab
另外一個有趣的例子，把字串的第一個char提出來
```
initials :: String -> String -> String
initials firstname lastname = [f] ++ ". " ++ [l] ++ ".”
    where (f:_) = first name
          (l:_) = lastname
```
Let… In…
在let裡面定義的變數可以在in裡面直接用，也記得要空格對齊
和之前的where做的事情大致一樣
不同之處：let in 是expression 可以用在statement
```
    =>4 * (let a = 9 in a + 1) + 2
    =>42
```
where只能用在function後面的敘述
要是不想要用空格對齊，也可以用”;”分開
```
    (let a = 100; b = 200; c = 300 in a*b*c, let foo="Hey "; bar = "there!" in foo ++ bar)
    (6000000,"Hey there!")
```
let 也可以用在domain裡面
```
    calcBmis :: (RealFloat a) => [(a, a)] -> [a]
    calcBmis xs = [bmi | (w, h) <- xs, let bmi = w / h ^ 2, bmi >= 25.0]
```
Case
```
case expression of pattern -> result
                                                    pattern -> result
                   pattern -> result
                   …
```
範例：case xs of pattern -> result
```
describeList :: [a] -> String
describeList xs = "The list is " ++ case xs of [] -> "empty.”
                                                    [x] -> "a singleton list.”
                                               xs -> "a longer list.”
```
另外一種更乾淨的寫法
```
describeList :: [a] -> String
describeList xs = "The list is " ++ what xs
    where what [] = "empty.”
          what [x] = "a singleton list.”
          what xs = "a longer list.”
```
＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝

小結：
function 裡面可以用where/ let…in… / case
＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝
第五章
Recursion
傳統要extract max of a list: tmp = a[0]，然後逐一比較每一個element，用比較大的那一個取代tmp，最後tmp就是答案
在heskell中我們用 myMax(x:xs) = max(x, myMax(xs))，把第一個element跟後面的sub-array的max來比較
要記得考慮邊境條件
其他範例：

```
replicate' n x => output list = x: replicate' n-1 x
        boundary condition: return [] when n = 0
take' n x:xs => output list = x: take' n-1 xs
        boundary condition: return [] when n = 0, return [] when list = empty
reverse' x:xs => output list = reverse' xs ++ [x]
        boundary condition: return [] when list = empty
repeat' x => output = x: repeat' x
        no boundary condition, same as standard library
zip' (x:xs) (y:ys) = (x,y): zip' (xs,ys)
        boundary condition: xs == empty || ys == empty
elem' a (x:xs)
        | a == x = True
        | otherwise = elem' a xs
        boundary condition: output False when list is empty
quick sort (ascending)
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =  把第一個當作pivot
      let smallerSorted = quicksort [a | a <- xs, a <= x]  a是xs之中，小於等於pivot x的element
      biggerSorted = quicksort [a | a <- xs, a > x]  a是xs之中，大於pivot x的element
      in  smallerSorted ++ [x] ++ biggerSorted
```

＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝
第六章
High Order Function = input parameter是function/ output是一個function

```
curried funtion: 所有的funtion都只接受一個input 參數
我們看到的那些兩個以上參數的function，實際上都是接受一個參數，return一個partial function 去接下一個參數
這個partial function也只接受一個input 參數
所以你看到multiThree 3 5 9，跟(((multiThree 3) 5) 9) 這兩個表示的東西是一樣的

infix function: function 放中間 (相較於 prefix/ postfix function)

其他的high order function
-把function 當作參數：
applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)
        注意這裡的宣告，第一個括號(a -> a)是必要的，表示我們的第一個參數是一個function，該function是a->a
```

接下來是第二個和第三個參數 a 和 a

所以這個function做的事：一個function跟一個type當作input，會把這個input套用這個function兩次之後在輸出

```
zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c] 跟zip類似，不過提供function當作參數，不同的zip operator
zipWith' _ [] _ = []
zipWith' _ _ [] = []
zipWith' f (x:xs) (y:ys) = f x y : zipWith' f xs ys
```

注意邊際條件

```
flip' :: (a -> b -> c) -> b -> a -> c  把輸入參數的順序調換
flip' f y x = f x y
```

原型是：

```
flip' :: (a -> b -> c) -> (b -> a -> c)
flip' f = g
        where g x y = f y x
map :: (a -> b) -> [a] -> [b]  input parameter是一個function和a 輸出b
map _ [] = []
map f (x:xs) = f x : map f xs
```

作用：把list當中每一個element都套用function f

其實map (+3) [1,5,3,1,6] 等價於[x+3 | x <- [1,5,3,1,6]]

(優點：可讀性？)

```
ghci> map (+3) [1,5,3,1,6]
[4,8,6,4,9]
ghci> map (++ "!") ["BIFF", "BANG", "POW”]
["BIFF!","BANG!","POW!”]
ghci> map (replicate 3) [3..6]
[[3,3,3],[4,4,4],[5,5,5],[6,6,6]]
ghci> map (map (^2)) [[1,2],[3,4,5,6],[7,8]]
[[1,4],[9,16,25,36],[49,64]]
ghci> map fst [(1,2),(3,5),(6,3),(2,6),(2,5)]
[1,3,6,2,2]
```

```
filter :: (a -> Bool) -> [a] -> [a]  input parameter是一個boolean function和一個list 輸出一個list
filter _ [] = []
filter p (x:xs)
    | p x     = x : filter p xs
    | otherwise = filter p xs
```

作用：用guard來判斷list當中的element x，如果true就把x放進output，然後繼續otherwise就跳過

可以把list當中符合boolean function (predicate function)的element filter出來

```
ghci> filter (>3) [1,5,3,2,1,6,4,3,2,1]
[5,6,4]
ghci> filter (==3) [1,2,3,4,5]
[3]
ghci> filter even [1..10]
[2,4,6,8,10]
ghci> let notNull x = not (null x) in filter notNull [[1,2,3],[],[3,4,5],[2,2],[],[],[]]
[[1,2,3],[3,4,5],[2,2]]
ghci> filter (`elem` ['a'..'z']) "u LaUgH aT mE BeCaUsE I aM diFfeRent”
"uagameasadifeent”
ghci> filter (`elem` ['A'..'Z']) "i lauGh At You BecAuse u r aLL the Same”
“GAYBALLS"
```

map 和 filter很好用，多練習
takeWhile :: (a -> Bool) -> [a] -> [a] 輸出從a的開頭到predicate function成立的最後一個element
(filter 會把每一個element都套用，takeWhile一路套用到false就停)
如果我們想要：find the sum of all odd squares that are smaller than 10,000 小於10000的所有奇數平方和
sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
=>166650
用domain define也可以做到一樣的事
sum (takeWhile (<10000) [n^2 | n <- [1..], odd (n^2)])
=>166650
練習：Collatz sequences (從一個數開始，如果是偶數就除以2，奇數就乘以3＋1，最後收斂到1)

另外一個partial function 的例子
ghci> let listOfFuns = map (*) [0..]
ghci> (listOfFuns !! 4) 5
=>20
[0..] 是一個無窮list, 0,1,2,3,4,5,6…
map (*) [0..] 製造了一個無窮list 0*,1*,2*,3*,…
listOfFuns !! 4 就是無窮list 的第四個element (4*)
(listOfFuns !! 4) 5 = (4*)5

Lambda: 有點類似_變數，anonymous function 之後不會再用到，不用特地去宣告他
用\表示(形狀像？)，之後接參數，之後接->，後面的事function本體
範例：
zipWith (\a b -> (a * 30 + 3) / b) [5,4,3,2,1] [1,2,3,4,5] lambda function: 兩個參數，function body: (a * 30 + 3) / b
=>[153.0,61.5,31.0,15.75,6.6]
map (\(a,b) -> a + b) [(1,2),(3,5),(6,3),(2,6),(2,5)]
=>[3,8,9,8,7]

fold: 由於遞迴處理list的時候 很常出現 x: xs 的結構 (挑出第一個element，再遞迴處理剩下的sub-list)
所以用fold來簡化這個過程
語法： foldl (binary function) stating_value head_of_list binary function：有兩個input的function
foldl 會用這個binary function 依序處理list
sum' :: (Num a) => [a] -> a
sum' xs = foldl (\acc x -> acc + x) 0 xs 0是stating_value xs是head_of_list
所以0先assign給acc 然後acc + list的第一個element
加總的結果再assign給axx 然後acc + list的下一個element
…
由於curried 的關係，也可以寫成
sum' :: (Num a) => [a] -> a
sum' = foldl (+) 0
省略head_of_list
Generally, if you have a function like foo a = bar b a, you can rewrite it as foo = bar b, because of currying.
另外一個fold的例子
elem' :: (Eq a) => a -> [a] -> Bool
elem' y ys = foldl (\acc x -> if x == y then True else acc) False ys
初始化的acc是false，預設element y 不在ys裡面
binary function是逐一對照ys裡面的element是不是==y

如果是的話就True，否則acc (如果前面有找到，acc就是True，沒有找到就是False)

foldl vs foldr 通常我們想要從某一個list建立一個list的時候用foldr比較有利

從尾端處理之後的值存在acc，處理前一個再：接上acc

當然也可以用foldl從頭開始，處理第一個，存在acc，處理第二個再acc＋接上

但是＋的花費成本比：大，

“Folds can be used to implement any function where you traverse a list once, element by element, and then return something based on that. Whenever you want to traverse a list to return something, chances are you want a fold”

foldl1 跟 foldr1: 省略starting value，把第一個element當作starting_value，從第二個開始fold

在處理empty list的時候會有問題

例如：

foldr (+) 0 [1..100] => 5050

foldr1 (+) [1..100] => 5050 兩個都ＯＫ

但是

foldr (+) 0 [] => 0

foldr1 (+) [] => ERROR

另外一組function: scan，跟fold一樣，只是會把過程中的acc都output出來

scanl/ scanr/ scanl1/ scanr1 就對應 foldl/ foldr/ foldl1/ foldr1

ghci> scanl (+) 0 [3,5,2,1]

=>[0,3,8,10,11]

ghci> scanr (+) 0 [3,5,2,1]

=>[11,8,3,1,0]

這個例子更容易看出來left跟right的方向，只是scanl順向把最後一個acc放在最後

scanr逆向把最後一個acc放在第一個

$ function

($) :: (a -> b) -> a -> b

f $ x = f x …好像沒有用耶

定義上沒有用，但是在operator的優先順序上我們把$設成最低，所以 “ “空白 “+”加減乘除 都會先處理

運用上可以幫我們省掉一些括號，增加可讀性，$以右的都會先運算，當成是$左邊function的input

sqrt 3 => 1.7320508

sqrt 3 + 4 + 9 => 14.7320508

sqrt (3 + 4 + 9) => 4

sqrt $ 3 + 4 + 9 => 4

例子

sum (filter (> 10) (map (*2) [2..10])) 等於 sum $ filter (> 10) $ map (*2) [2..10]

另一個特殊用法：

map ($ 3) [(4+), (10*), (^2), sqrt]

[7.0,30.0,9.0,1.7320508075688772]

function composition：跟數學定義一樣,

(.) :: (b -> c) -> (a -> b) -> a -> c

f . g = \x -> f (g x) 不過要確定 g 的 output型態是 f 的 input型態

所以map (\x -> negate (abs x)) [5,-3,-6,7,-3,2,-19,24]

等價於map (negate . abs) [5,-3,-6,7,-3,2,-19,24]

把list當中的數字絕對值之後再負號

上面都是一個input的例子，如果遇到兩個input:

sum (replicate 5 (max 6.7 8.9))

由於curried的關係，等價於
(sum . replicate 5 . max 6.7) 8.9
也等價於
sum . replicate 5 . max 6.7 $ 8.9
理解的方式：apply 8.9 to (max 6.7), and apply to replicate 5, and apply to sum

point free style 省略輸入參數，像是
sum' :: (Num a) => [a] -> a
sum' xs = foldl (+) 0 xs
可以寫成
sum' = foldl (+) 0 省略xs
但是遇到類似
fn x = ceiling (negate (tan (cos (max 50 x))))
不能把x直接省略，這樣右邊會變成 (max 50)
改用composition來表示：
fn = ceiling . negate . tan . cos . max 50

注意chain不要拉太長，可讀性很差，有時候用let … in 還比較好
oddSquareSum :: Integer
oddSquareSum = sum . takeWhile (<10000) . filter odd . map (^2) $ [1..]
wtf?
oddSquareSum :: Integer
oddSquareSum =
    let oddSquares = filter odd $ map (^2) [1..]
      belowLimit = takeWhile (<10000) oddSquares
    in sum belowLimit

＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝
小結：
haskell內部都是curried function (只吃一個參數，回傳一個partial function)
high order function 可以把function當做參數輸入
map f xs 跟filter p xs 很好用
lambda \ 代表匿名函數，叫什麼不重要，之後也不會用到
你如果想要element by element的處理一個list內的東西，就用fold (scan會逐步輸出成list)
$: 右邊的先運算，當成是左邊function的input
“.” 可以作function composition，但是注意不要用太長，可讀性很差
＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝
第七章
module
前面使用的所有function都是default module: Prelude module
可以在.hs 檔的最前面加入import Data.List
或是在cmd line 中加入，使用:m + Data.List 增加 (:m - Data.List 刪減該module)
同時加入數個module:
  :m + Data.List Data.Map Data.Set
只想要加入module中的特定function:
  import Data.List (nub, sort)
想要加入特定module，但是不想要其中的特定function: (通常用在想要用自己的function取代該function)
  import Data.List hiding (nub)
有時候import進來的module會有一些function name跟現存的一樣，此時要使用qualified import
  import qualified Data.Map

之後使用的時候Data.Map.XXX 就是 import進來的function，單純使用XXX就是現存的function
每次都用Data.Map.XXX很麻煩，使用
import qualified Data.Map as M
之後使用M.XXX 就是import進來的function，單純使用XXX就是現存的function
常用的Data.List function (都是處理一些list)
intersperse a [a] 把element a 插入list中每一個element之間
intercalate [a] [[a]] 把一個list [a]插入一個list of list 的每一個list的中間，輸出一個大list
transpose 把list內的list當作矩陣，column row互換
比方說好幾個多項式相加的時候3x2 + 5x + 9, 10x3 + 9 and 8x3 + 5x2 + x - 1
= map sum $ transpose [[0,3,5,9],[10,0,0,9],[8,5,1,-1]]
foldl'/ foldr' 和 foldl/ foldr 一樣，不過處理大list的時候不會overflow (舊版實際上沒有每一個iteration都更新acc)
concat 為一個list of list解除nesting (把list當中的list連起來)
concatMap f [a] 先做map (把function apply到list的每一個element，再連起來)
concatMap (replicate 4) [1..3] => [1,1,1,1,2,2,2,2,3,3,3,3]
concat ((replicate 4) [1..3]) => [1,2,3,1,2,3,1,2,3,1,2,3]
and/or [a] 處理一個list的 boolean，輸出and/or 全部element的結果
and $ map (>4) [5,6,7,8] => True
any/ all p [a] 用 predicate function來判斷 list當中所有的element 一般我們喜歡用any/all instead of and/or
any (==4) [2,3,5,6,1,4] => True
iterate f a 把function apply 到 starting_value，再apply 一次，製造無窮數列
take 10 $ iterate (*2) 1 => [1,2,4,8,16,32,64,128,256,512]
splitAt num [a] 把list 在num的地方切開，輸出一個tuple，裡面有兩個list (前半＋後半)
takeWhile p [a] 從頭開始輸出到不符合predicate funtcion的地方
dropWhile p [a] takeWhile的相反，從頭開始drop，直到第一個符合prodicate的地方
span p [a] 跟takeWhile做一樣的事，不過最後會輸出一個pair (takeWhile, dropWhile)
break p [a] 跟takeWhile很像，不過 takeWhile會切在第一個不符合p的地方 breal會切在第一個符合p的地方
break p 等價於 span (not . p)
sort [a] 排序，list當中的element要是Ord 的typeclass (否則無法比較大小，也就無法排序)
group [a] 把list當中的element組成list，最後輸出list of list
組合使用：想要知道list當中的element出現幾次
map (\l@(x:xs) -> (x,length l)) . group . sort $ [1,1,1,1,2,2,2,2,3,3,2,2,2,5,6,7]
=>[(1,4),(2,7),(3,2),(5,1),(6,1),(7,1)]
inits/ tails 從頭到尾/從尾到頭 分解list，每一個element都是一次分解的結果
ghci> inits "w00t"
["","w","w0","w00","w00t"]
ghci> tails "w00t"
["w00t","00t","0t","t",""]
isInfixOf/ isPrefixOf/ isSuffixOf [a] [a] 判斷特定sub-list是不是在list當中/ 最前面/ 最後面
(elem是判斷特定的element是不是在list當沖)
partition p [a] 用predicate function去partition list，輸出兩個list，符合p的放前面，不符合的放後面
find p [a] drop直到第一個符合p 的element (有點像dropWhile，但是find的return 值是Maybe a)
Maybe a，如果input是Int，輸出就是Maybe Int，輸出結果會是一個Just X或是Nothing
elemIndex a [a] 跟elem一樣，確定特定element是不是在list當中，找到的話回傳該element的位置，return值也是Maybe a
所以輸出格式: Just X或是Nothing (如果找不到就是Nothing)
elemIndices a [a] 跟elemIndex一樣，尋找特定的element，會回傳element的位置 (一個數字的list)
如果完全找不到，return []

findIndex p [a] 回傳第一個符合p的element的index

輸出格式: Maybe X或是 Nothing

findIndices p [a] 回傳符合p 的element的位置 (會回傳一個list)

如果完全找不到，return []

zip/ zipWith/ zip3/ zipWith3/ … 最多到7

lines string 把string用"\n" (換行符號)切開，輸出一個string的list

words string 把string用 “ “ (空白符號)切開，輸出一個string的list

unwords 把一個list組成一個string，中間會用空白隔開

nub [a] 把list當中的重複的element刪掉，只留下第一個

delete a [a] 刪掉list特定的第一個element

\\ 比較兩個list，輸出兩個list不重複的element

用法 [list_a] \\ [list_b] 會把list_b當中的element拿去比對list_a，有出現就刪掉他

[1..10] \\ [2,5,9] 等價於 delete 2 . delete 5 . delete 9 $ [1..10]

union 把兩個list union (重複的不會疊加)

用法 [list_a] union [list_b]

[1..7] `union` [5..10] => [1,2,3,4,5,6,7,8,9,10]

intersect 找出兩個list都有的element

insert a [a] 會把a插入到list當中第一個比a大的element之前 (整體不保證sorted)

但是如果insert a 進入一個 sorted list，still sorted

length, take, drop, splitAt, !! and replicate 這些只能接受int格式,

用genericLength, genericTake, genericDrop, genericSplitAt, genericIndex and genericReplicate代替,

像是 (!!) 1 ['A'..'Z'] 會失敗

genericIndex ['A'..'Z'] 0 => ‘A’

像是let xs = [1..6] in sum xs / length xs 會失敗 sum輸出a length輸出Int，不能直接拿來除

let xs = [1..6] in sum xs / genericLength xs => 3.5

nub, delete, union, intersect and group 也有generic

用nubBy, deleteBy, unionBy, intersectBy and groupBy

XXBy f [a] 用function描述的方式去篩選list

在使用這些XXBy的function的時候很常搭配 on function

let values = [-4.3, -2.4, -1.2, 0.4, 2.3, 5.9, 10.5, 29.1, 5.3, -2.4, -14.5, 2.9, 2.3]

groupBy (\x y -> (x > 0) == (y > 0)) values 把value用正負號分group (同時大於0或是同時小於0的一組)

[[-4.3,-2.4,-1.2],[0.4,2.3,5.9,10.5,29.1,5.3],[-2.4,-14.5],[2.9,2.3]]

也可以寫成

ghci> groupBy ((==) `on` (> 0)) values value中 (>0) 的結果一致 (==) 的同一組

[[-4.3,-2.4,-1.2],[0.4,2.3,5.9,10.5,29.1,5.3],[-2.4,-14.5],[2.9,2.3]]

on 的定義

f `on` g = \x y -> f (g x) (g y)

所以(==) `on` (> 0) 等價於 \x y -> (x > 0) == (y > 0)

sort, insert, maximum and minimum也有generic

用sortBy, insertBy, maximumBy and minimumBy

很漂亮的寫法：

let xs = [[5,4,5,4,4],[1,2,3],[3,5,4,3],[],[2],[2,2]]

ghci> sortBy (compare `on` length) xs 想要把xs內的list 用長度排序

[[],[2],[2,2],[1,2,3],[3,5,4,3],[5,4,5,4,4]]

常用的Data.Char function (都是處理字串)

isControl 判斷是否control character (換行/ null/ CF/ EOF/ …)

isSpace 判斷是否space (包括tab 跟換行)

isLower/ isUpper/ isAlpha/ isAlphaNum/ isPrint/ isDigit/ isOctDigit/ isHexDigit/ isLetter判斷是否

小寫字母/ 大寫字母/ 字母/ 字母數字/ 可以print/ 數字/ 十進位 / 16進味/ 是不是字母 (控制字元就不是Printable)

isMark/ isNumber/ isPunctuation/ isSymbol/ isSeparator/ isAscii/ isLatin1/ isAsciiUpper/ isAsciiLower/ 判斷是否

uniMark/ 數字 / 標點符號/ 符號 / 分隔符號/ ascii/ latin /大寫 ascii/ 小寫 ascii

(ascii 在unicode 前128bits. latin在unicode 前256 bits)

generalCategory 可以查詢 category

toUpper/toLower 把char全部轉成大寫/小寫，其他不變

toTitle 把char轉成title case，基本上是大寫

digitToInt 轉成數字，只接受 '0'..'9' 'a'..'z' 'A'..'Z' 的input

map digitToInt "34538" => [3,4,5,3,8]

intToDigit 數字(1..15)轉成digit 字串

ord/ chr 把char轉成對應數字/ 把對應數字轉成char

encode x string/ decode x string 凱薩加密 (只是shift x)

常用的Data.Map function

用於dictionaries (一個key跟一個value)

最常使用的: findKey/ lookup (考慮到可能會找不到，使用maybe 當作回傳值)

```
findKey :: (Eq k) => k -> [(k,v)] -> Maybe v
findKey key [] = Nothing
findKey key ((k,v):xs) = if key == k
                         then Just v
                         else findKey key xs
```

使用fold的寫法：

```
findKey :: (Eq k) => k -> [(k,v)] -> Maybe v
findKey key = foldr (\(k,v) acc -> if key == k then Just v else acc) Nothing
```

fromList [(k,v)] 把list輸出成 map list，注意用這種方法的時候key要可以Ord (普通list只要可以Eq就好)

這裡為了建立Tree儲存map，需要比大小

empty 建立empty map

insert k v map 插入一筆 (k,v) 進入map

```
Map.empty
fromList []
ghci> Map.insert 3 100 Map.empty
=> fromList [(3,100)]
```

null map 檢查map是否為empty

size map 檢查map有多少組element

singleton k v 建立一個map，裡面含有一組 (k,v)

lookup k map 回傳map中key = k的value，回傳值是maybe，所以找不到該key的話回傳Nothing

member k map 檢查 map是否有key 是k 的pair，這是一個predicate function (T/F)

map f map 和list的map類似，把特定function套用到map中的每一個value

filter p map 和list的filter類似，回傳 map中符合p的value (會回傳map)

toList map fromList的相反，把map轉成List

keys 回傳keys (以list型態) 等價於 map fst . Map.toList

elems 回傳value (以list型態) 等價於 map snd . Map.toList

fromListWith f list 跟fromList一樣，但是增加一個function，遇到相同key的時候要如何處理

```
phoneBookToMap :: (Ord k) => [(k, String)] -> Map.Map k String
phoneBookToMap xs = Map.fromListWith (\number1 number2 -> number1 ++ ", " ++ number2) xs
```

遇到名字一樣的時候 把號碼用字串＋＋串起來

Map.fromListWith max [(2,3),(2,5),(2,100),(3,29),(3,22),(3,11),(4,22),(4,15)]
fromList [(2,100),(3,29),(4,22)]
遇到key 一樣的時候，選value最大的
insertWith f k v 就跟insert k v 一樣，增加的function是在遇到key相同的時候要如何處理
Map.insertWith (+) 3 100 $ Map.fromList [(3,4),(5,103),(6,339)]
fromList [(3,104),(5,103),(6,339)]
常用的Data.Set function
一個set當中的每一個element都是unique的
text1 = "I just had an anime dream. Anime... Reality... Are they so different?”
let set1 = Set.fromList text1
=> set1 fromList " .?AIRadefhijlmnorstuy" 每一個用到的char都算一個element，不會加到重複的，排序過
intersection Set1 Set2 回傳兩個set重複的部分
difference Set1 Set2 回傳兩個set差異的部分
union Set1 Set2 把兩個set Union起來
其他 null/ size/ member/ empty/ singleton/ insert/ delete 都異樣，檢查一個set是否存在/大小/是否存在a/ 是否[]/ 建立一個set/ 插入set/ 刪除特定element
set_1 isSubsetOf set_2 set_2是否包含set_1的每一個元素
set_1 isProperSubsetOf set_2是否包含set_1的每一個元素，還有其他元素
map f set/ filter p set 把f apply到set的每一個元素/ 把set中符合p的元素留下來
建立自己的module
XXX.hs

```
module XXX
( f_name1
,f_name2
,f_name3
) where

f_name1:: a->a
…
```

然後只要import XXX就可以使用當中的function
可以在不同的module中使用相同的function name，只要記得他們實作是不同的，還有呼叫時的命名
＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝
小結：
import進來的module，有簡化名稱，可以只加入特定function或是不想加入特定function
想要查詢特定module的用法：Hoogle
Data.List
find/ findIndex/ findIndices 還有 elem/ elemIndex/ elemIndices 的差異
Data.Map, Data.Set
在使用map跟set的時候要小心，很多跟prelude and Data.List有重複，用qualified import比較好
import qualified Data.Map as Map和import qualified Data.Set as Set
＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝
第八章
Type/ Typeclass
data Bool = False | True
data Int = -2147483648 | -2147483647 | ... | -1 | 0 | 1 | 2 | ... | 2147483647 實際上不是這樣寫的，純說明用
試著自己定義一個type: Circle和Rectangle
data Shape = Circle Float Float Float | Rectangle Float Float Float Float
(圓形的中心_x, 圓形的中心_y, 半徑) (左上角_x, 左上角_y, 右下角_x, 右下角_y)

在宣告的最後面加上deriving (Show)：可以讓你的形態自動變成part of Show，就可以show出來了
漂亮的寫法：把Point也宣告成Type
data Point = Point Float Float deriving (Show)
data Shape = Circle Point Float | Rectangle Point Point deriving (Show)
在宣告一個function，根據shape計算面積
surface :: Shape -> Float
surface (Circle _ r) = pi * r ^ 2
surface (Rectangle (Point x1 y1) (Point x2 y2)) = (abs $ x2 - x1) * (abs $ y2 - y1)
也可以把自己定義的data type寫在module裡面輸出
module Shapes
(Point(..) … 就是把Point的所有結構輸出
, Shape(..)
, surface
, nudge
, baseCircle
, baseRect
) where

Record Syntax:
當你要建立一個很大的data type的時候，直覺做法是
data Person = Person String String Int Float String String deriving (Show)，然後為了要提取每一個member，需要定義function
firstName :: Person -> String firstName (Person firstname _ _ _ _ _) = first name
lastName :: Person -> String lastName (Person _ lastname _ _ _ _) = lastname
age :: Person -> Int age (Person _ _ age _ _ _) = age
height :: Person -> Float height (Person _ _ _ height _ _) = height
phoneNumber :: Person -> String phoneNumber (Person _ _ _ _ number _) = number
flavor :: Person -> String flavor (Person _ _ _ _ _ flavor) = flavor …起笑
Haskell提供更好的辦法
data Person = Person { firstName :: String
, lastName :: String
, age :: Int
, height :: Float
, phoneNumber :: String
, flavor :: String
} deriving (Show)
定義data type的時候用大括號括起來，每一個name用 :: 後接資料形態，Haskell會自動幫忙建立以name為名的function

Type parameter
一般的Value constructor: take input，and produce value (在C裡面建構子：呼叫class的時候建立)
Type contractor: take parameter(a type), and produce type
像是data Maybe a = Nothing | Just a
Maybe就是一個Type constructor，a 就是一個type，如果最後的結果有找到(不是Nothing)的話，就會回傳一個Just a
Just a 的a會根據input的a決定，注意沒有一種形態叫做Maybe，他不可能單獨存在，否則沒有parameter

有一個要注意的是：儘量不要在宣告data的時候加上constraint，到時候在function宣告的時候也要加，很麻煩

一個運用使用自己宣告的data例子

data Vector a = Vector a a a deriving (Show) 注意在這裡第一個Vector是Type condtructor a 是type parameter

第二個Vector是Value constructor

vplus :: (Num t) => Vector t -> Vector t -> Vector t input是兩個Vector，output是Vector，加總

(Vector i j k) `vplus` (Vector l m n) = Vector (i+l) (j+m) (k+n)

vectMult :: (Num t) => Vector t -> t -> Vector t input是Vector，output是Vector，乘以一個t

(Vector i j k) `vectMult` m = Vector (i*m) (j*m) (k*m)

scalarMult :: (Num t) => Vector t -> Vector t -> t input是兩個Vector，output是相乘相加

(Vector i j k) `scalarMult` (Vector l m n) = i*l + j*m + k*n

Derived instance

一個type是某一個type class的instance，表示這個type支援該type class的行為運算

(像是 Int 是instance of Eq，所以Int可以用 (==))

常見的Type class: Eq, Ord, Enum, Bounded, Show, Read

假設我們現在定義了一個

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     } deriving (Eq, Show, Read)
```

要在最後再加上這個 deriving (Eq) 才能直接把 person1 == person2

加上這個之後，在裡面的data type每一個都要支援Eq才有用，不過由於Int和String都ＯＫ，所以這裡安全

```
let mikeD = Person {firstName = "Michael", lastName = "Diamond", age = 43}
ghci> "mikeD is: " ++ show mikeD  支援show
=>"mikeD is: Person {firstName = \"Michael\", lastName = \"Diamond\", age = 43}"

read "Person {firstName =\"Michael\", lastName =\"Diamond\", age = 43}" :: Person 支援 read
=> Person {firstName = "Michael", lastName = "Diamond", age = 43}
```

練習一下 Ord，假設我們想要讓Boolean 可以比較大小

data Bool = False | True deriving (Ord) 宣告的時候 True擺比較後面：True比較大

maybe中Nothing 永遠比Maybe x 小

如果我們想要做一個enum的data type:

```
data Day = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
      deriving (Eq, Ord, Show, Read, Bounded, Enum)
ghci> show Wednesday  => "Wednesday”  可以用show
ghci> read "Saturday" :: Day => Saturday  可以用read
ghci> Saturday == Saturday => True  可以用 ==
ghci> Saturday > Friday => True  可以用Ord
ghci> Monday `compare` Wednesday => LT  可以用Ord
ghci> minBound :: Day => Monday  可以用bound
ghci> maxBound :: Day => Sunday
ghci> succ Monday => Tuesday  可以用enum
ghci> pred Saturday => Friday
ghci> [Thursday .. Sunday] => [Thursday,Friday,Saturday,Sunday]
```

ghci> [minBound .. maxBound] :: [Day] =>
[Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday]
Type Synonyms
用type String = [Char] 表示string 跟[char] 等價
所以當你有一個map是這樣
phoneBook :: [(String,String)] phoneBook = [("betty","555-2938") …]
為了增加可讀性，我們可以
type PhoneNumber = String
type Name = String
type PhoneBook = [(Name,PhoneNumber)]
我們要定義一個function的時候
inPhoneBook :: Name -> PhoneNumber -> PhoneBook -> Bool
inPhoneBook name pnumber pbook = (name,pnumber) `elem` pbook
但是要注意不要做過頭，不然你的程式就會充斥著一堆陌生的變數名稱

type也可以加入參數表示，像是
type AssocList k v = [(k,v)]
注意concrete type的意思是沒有a/b/c這種未指明形態的東西，像是Maybe String

type 也可以partial apply，像是
type IntMap v = Map Int v 可以定義一個map，key一定是Int，point free寫法：type IntMap = Map Int

Either a| b 的用法
data Either a b = Left a | Right b deriving (Eq, Ord, Read, Show)
當get Either a b 的data 之後才會決定是哪一個type (可能是a也可能是b)
跟Maybe a 的用法有點像，會回傳Nothing或是Just x
但是Nothing的時候我們什麼也不知道，只知道沒有找到
使用either的話，fail的情形是Left，成功的情形是Right
實際例子：map裡面存著locker編號，還有密碼，有人占用的話Taken，沒有人占用的話就是Free
map長這樣
lockers :: LockerMap
lockers = Map.fromList
[(100,(Taken,"ZD39I"))
,(101,(Free,"JAH3I")) …]
使用lookup，找到locker編號所對應的locker的密碼
這裡失敗的情形有兩種：編號不存在，或是編號存在，但是Taken
import qualified Data.Map as Map
data LockerState = Taken | Free deriving (Show, Eq)
type Code = String
type LockerMap = Map.Map Int (LockerState, Code)
在這裡使用 Either String Code，失敗的時候用String，成功用Code
lockerLookup :: Int -> LockerMap -> Either String Code
lockerLookup lockerNumber map =
case Map.lookup lockerNumber map of
Nothing -> Left $ "Locker number " ++ show lockerNumber ++ " doesn't exist!"
Just (state, code) -> if state /= Taken
then Right code
else Left $ "Locker " ++ show lockerNumber ++ " is already taken!"
Recursive Data Structure
to be continued...

＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝
小結：