chapter 1

① 2+2　　(+) 2 2

2 * (-3)　　2 * -3 ×

② && 与　　|| 或

不等于 /=　不是 !=

not True

不确定加括号 ☆

③ 列表 [1,2,3]，所有元素同 type ☆

④ [1..10] = [1,2,3,4,5,6,7,8,9,10]

enumeration 列举

[1,4..15] = [1,4,7,10,13]

两个最前两个表示元素之间大小.

⑤ [3,1,3] (++) [3,7] = [3,1,3,3,7]

++ 连接两个列表

⑥ : 往前加一个元素到列表头部.

1 : [2,3] = [1,2,3]

△⇒必须单个元素，第二个必须列表

⑦ \n 换行　\t 制表符

必须用 putStrLn，才能换行.

直接输入，就直接显示 \n \t.

⑧ let a = ['l','o','t','s']

a == "lots"

"" == []

'a' : "bc" == "abc"

单字符 ''，字符串 ""

"foo" ++ "bar" == "foobar"

⑨ 类型名：大写字母开头.

变量：小写字母开头.

⑩ :: [Char]

最后一次求值的结果，ghci 辅助功能

⑪ :m + Data.Ratio. 加载模块　①

:m = :module.

:unset +t 取消.

:type 'a'.

:t 3+3.

⇒ 3+3 :: Num a ⇒ a. ⇒ Chapter 6.

Chapter 2

① type { strong：不会强制转换，拒绝元素不匹配
static：在编译期知道 value and type 表达式
automatically inferred.

True && "False" ⇒ 报错.

2点⇒ 小错误少，不可能运行时出现类型错误，更安全.

② 常用类型：Char：单个 unicode 字符.

Bool：True False

Int：fixed-width 固定 32 64 bits

Integer：不限长度带符号整数.

Double：浮点数.

Float：不推荐，慢.

③ :: 类型签名

如果不显式，那么就自动推导.

④ (compare 2 3) == LT

⇒ True.

compare (sqrt 3) (sqrt 6)

⇒ LT　不加括号报错.

⑤ String = [Char]. 别名.

⑥ head [1,2,3] ⇒ 1　it :: Num a ⇒ [a]

tail [1,2,3] ⇒ [2,3]

tail [1,2] ⇒ [2]

⑥ 列表的通用型 ⇒ 多态 (polymorphic)

[a] ⇒ 类型为a的列表.

[[Int]] ⇒ [Int] 类型的列表

⑦ 元组 ☆. (1964, "Labyrinths")

长度固定，包含不同类型的值

用括号包围.

(True, "hello") :: (Bool, [Char])

() ⇒ 零个元素的元组.

命名: 2-元组

⑧ 只有当元组(数量 位置 类型)相同.

⇒ 元组的类型相同.

(False, 'a') :: (Bool, Char)

(True, 'b') :: (Bool, Char)

⑨ 应用场景:

① 若一个函数返回多个值，可以将其包装到一个元组，然后返回元组作为函数的值.

② 当要事先长容器，但又必须用自定义类型，可以用元组.

⑩ 处理列表元组的函数.

☆take: 数字 n, 列表 l 返回一个包含 l 前 n 个元素的列表

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

⇒ [1,2]

drop: 返回丢掉了前n个元素的列表.

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

⇒ [3,4,5]

⑪ fst, snd ⇒ 都接受一个元组参数

返回元组第一个和第二个元素.

fst (1, 'a') ⇒ 1

snd (1, 'a') ⇒ 'a'

⑫ lines :: String → [String]

lines 函数接受单个字符串，按'\n'

换文字分割成多个字符串

lines "the quick\nbrown fox\njumps"

⇒ ["the quick", "brown fox", "jumps"]

⑬ 不纯函数都以IO开头.

:t readFile

⇒ readFile :: FilePath → IO String

⑭ 变量，一旦变量绑定了某个表达式，那么变量的值不会改变

不允许多次赋值

⑮ myDrop n xs = if n<=0 || null xs

then xs

else myDrop (n-1) (tail xs)

不同分支类型相同

⑯ 惰性求值：直到被需要才求值

Chapter 3

① data 定义新数据类型 即 赋 值

data BookInfo = Book Int String [String]
deriving (Show)

BookInfo: 类型构造器 > 名字相同
Book: 值构造器

:t Book 1 "abc" ["abc", "def"]
⇒ Book 1 "abc" ["abc", "def"] :: BookInfo.

② type: 设置别名

type BookRecord = (BookInfo, BookReview)

data Bool = False | True

③. 元组不代表类型. b:([Char],[Char])

a = ("abc", "def")
b = ("def", "abc")
a=b.
:t a ⇒ a :: ([Char], [Char])

data Cola = Cola String String
data Fur = Fur String String

c = Cola "Por" "Grey"
d = Fur "Por" "Gray"
:t c ⇒ c :: Cola
:t d ⇒ d :: Fur
c≠d

④ 枚举

data Rygbw = Red
| Orang
| Yellow
| Green
deriving(Eq, Show)

⑤ 模式匹配 ☆ ③

sumList (x:xs) = x + sumList xs
sumList [] = 0.

⑥ 通配符 _

要包括所有情况, 包括空构造器

⑦ data Maybe a = Just a
| Nothing

⑧ data Tree a = ~~Node~~
= Node a (Tree a) (Tree a)
| Empty
deriving (Show)

⑨ mySecond xs = if null (tail xs)
bug ← then error "list too sh
else head (tail xs)

⇒ 如果要指出某种特定失败
⇒ 用 Nothing 构造器. 反之用 Just

safeSecond [] = Nothing
safeSecond xs = if null (tail xs)
then Nothing
else Just (head (tail xs)
不写.

safeSecond [1] ⇒ Nothing
safeSecond [1,2] ⇒ Just 2

(10) 引入局部变量 (1) let ·

in 标识这个区块的结束.

```
foo = let a = 1
      in let b = 2
      in a + b
```

(2) where:

```
lend x amount balance =
    if amount < reserve * 0.5
    then Just newBalance
    else Nothing
  where reserve = 100
        newBalance = balance - amount
```

Chapter 4.

1. IntParse

```
import Data.Char (digitToInt)

asInt xs = loop 0 xs

loop :: Int -> String -> Int
loop acc [] = acc : 终止情形
loop acc (x:xs) = let acc' = acc * 10
                      + digitToInt x
                  in loop acc' xs
```

```
asInt "33"
=> 33
```

(2) square :: [Double] -> [Double]

```
square (x:xs) = x * x : square xs
square [] = []
```

(3) upper case

```
import Data.Char (toUpper)

upperCase :: String -> String

upperCase (x:xs) = toUpper x : upperCase xs
upperCase [] = [].
```

{ base case
  recursive case : 处理头部, 处理其它

(4) map ☆

输入: 一个函数, 一个列表.
将函数运用到列表的每个元素, 输出这个列表.

```
square2 xs = map squareOne xs
  where squareOne x = x * x

upperCase2 xs = map toUpper xs
```

:t map ☆☆

=> map :: (a -> b) -> [a] -> [b]
a, b 类型可能不一样

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

⑤ 筛选列表元素，保留满足条件的.

例. 返回列表以有奇数元素. guard

```
oddList :: [Int] -> [Int]
oddList (x:xs) | odd x = x : oddList xs
               | otherwise = oddList xs
oddList [] = []
```

⑥ filter：接受一个谓词函数，并将它应用到列表每个元素，只有返回为True的元素才被保留.

⇒ 返回为Bool的函数

```
filter :: (a -> Bool) -> [a] -> [a]
filter odd [1..10]
⇒ [1,3,5,7,9]
```

⑦ 收集器. 对列表所有元素求和.

```
mySum xs = helper 0 xs
  where helper acc (x:xs) = helper (acc+x) xs
        helper acc [] = acc
```

⑧ foldl：对一个列表所有元素做累计处理，并且一边处理元素，一边累积，最后处理完整个列表，返回累积结果的值.

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl step zero (x:xs) = foldl step (step zero x) xs
foldl _ zero [] = zero
```

foldl 接受一个step函数，一个累积器的初始值，一个列表，step function 每次用累积器和列表中的一个元素作为参数，并计算新累加器，(stepper)，再计算，完成

⑤

```
foldSum xs = foldl step 0 xs
  where step acc x = acc + x
```

step 只是加两个数.

⇒

```
niceSum :: [Integer] -> Integer
niceSum xs = foldl (+) 0 xs
```

```
niceSum [1,2,3]
== foldl (+) 0 (1:2:3:[])
⇒ foldl (+) (0+1) (2:3:[])
⇒ foldl (+) ((0+1)+2) (3:[])
⇒ foldl (+) (((0+1)+2)+3) []
⇒ (((0+1)+2)+3)
```

foldr :: (a->b->b) -> b -> [a] -> b

```
foldr step zero (x:xs) = step x (foldr step zero xs)
foldr _ zero [] = zero
niceSumFoldr :: [Int] -> Int
niceSumFoldr xs = foldr (+) 0 xs
niceSumFoldr [1,2,3]
= foldr (+) 0 (1:2:3:[])
= 1 + foldr (+) 0 (2:3:[])
= 1+(2 + foldr (+) 0 (3:[]))
= 1+(2+(3 + foldr (+) 0 ([])))
= 1+(2+(3+0))
```

⑨ filter 用 foldr 来写

```
myFilter p xs = foldr step [] xs
  where step x ys | p x = x : ys
                  | otherwise = ys
```

后面可以用 foldr 定义的 ⇒ primitive recursive

:t - dropWhile isSpace

```
dropWhile isSpace :: [Char] -> [Char]
```

将一个参数传给一个函数时，最前面的元素就被给除掉.

部分函数应用：Currying

```
niceSumPartial :: [Int] -> Int
niceSumPartial = foldl (+) 0

niceSumPartial [1..10]
```

⇒ 55.

⑩ map 用 foldr 定义

```
myFoldrMap :: (a->b) -> [a] -> [b]
myFoldrMap f xs = foldr step [] xs
  where step x ys = f x : ys
```

⑪ foldl 用 foldr 写

```
myFoldl :: (a->b->a) -> a -> [b] -> a
myFoldl f z xs = foldr step id xs z
  where step x g a = g (f a x)
```

⑫ append

```
append :: [a] -> [a] -> [a]
append xs ys = fold (:) ys xs
```

⑬ 用 foldl'

第一个参数：如何处理

第二个参数：当空列空，递推来指代.

foldr

⑭ Lambda 函数.

\ 开始，函数体定义 →.

```
isInAny2 needle haystack =
  any (\s -> needle `isInfixOf` s) haystack
```

只能定义一条.

⑮ 部分函数和柯里化

```
:t dropWhile
dropWhile :: (a -> Bool) -> [a] -> [a]
```

⑯ .section .串. 使用括号包围一个操作符，通过在括号里面提供左操作对象或右操作对象，可产生一个部分应用函数.

```
map (2^) [3,5]
```

⇒ [8, 32]

```
map (^2) [3,5]
```

⇒ [9, 25]

⑰ tails "foobar"

⇒ ["foobar", "oobar", "obar", "bar", "ar", "r", ""]

⑱ suffixes :: [a] → [[a]]

suffixes xs@(_:xs') = xs : suffixes xs'

suffixes [] = []

⇒ 如果输入值能匹配@符号右边的模式(_:xs'), 那么就将这个值绑定到@符号左边的变量中(xs)

⑲ init : 返回一个列表中除了最后一个元素之外的其他元素.

suffixes2 xs = init (tails xs)

⇒ compose :: (b→c) → (a→b) → a→c

compose f g x = f (g x)

(.) = compose

let suffixes' = init . tails

:type (.) 右关联的

⇒ (.) :: (b→c) → (a→b) → a→c

:type suffixes'

⇒ suffixes' :: [a] → [[a]]

自动派生

data Color = Red | Green | Blue

deriving (Read, Show, Eq, Ord)

newtype : declae types, with only one constructor, with only one argument

Chapter 6

Show : type show

⇒ show :: Show a ⇒ a → String

data Color = Red | Green | Blue in

instance Show Color where

show Red = "Red"

show Green = "Green"

show Blue = "Blue"

Haskell

① XOR

```
xor :: Bool -> Bool -> Bool
xor a b = (a || b) && not (a && b)
```

✓	✓	x
✓	x	✓
x	✓	✓
x	x	x

②
```
head [] = error "error"
head (x:_) = x
tail [] = error "error"
tail (_:xs) = xs
```

③
```
append [] list = list
append (e:es) list = e : append es list
```

④ reverse [3,2,1] . [1,2,3]
```
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

⑤
```
getNthTale _ [] = error
getNthTale n (x:xs) =
  | n < 0 = error
  | n == 0 = x
  | n > 0 = getNthTale (n-1) xs
```

⑥
```
instance Show Card where show = showCard
showCard :: ...
```

⑦
```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * (factorial n-1)
```

⑧
```
myElement :: Eq a => a -> [a] -> Bool
myElement _ [] = False
myElement n (x:xs) = True
myElement n (x:xs) = myElem n xs
```

⑨ transpose

```
transpose :: [[a]] -> [[a]]
transpose [] = error "e"
transpose list@(xs:xxs)
  | len > 0 = transpose' len list
  | otherwise = error ""
  where len = length xs
transpose' len [] = replicate len []
transpose' len (xs:xss)
  | len == length xs = zipWith (:) xs
                       (transpose' len xss)
  | otherwise = error ""
```

---

⑩
```
(length, sum, sumsq)
stats [] = (0, 0, 0)
stats (n:ns) =
  let (len, sum, sumsq) = stats ns
  in (len+1, sum+n, sumsq+n*n)
```

⑪ zipWith
```
zipWith :: (a->b->c) -> [a] -> [b] -> [c]
zipWith f [] _ = []
zipWith f _ [] = []
zipWith f (x:xs) (y:ys) =
    f x y : zipWith f xs ys
```

⑫
```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl _ base [] = base
foldl f base (x:xs) =
  let newbase = f base x in
  foldl f newbase xs
```

(13) foldr :: (a -> b -> b) -> b -> [a] -> b

```
foldr f base [] = base
foldr f base (x:xs) =
    let fxs = foldr f base xs
    in f x fxs
```

```
main :: IO ()
main = do
  putStrLn "String"
  len <- readLen
  putStrLn "..." ++ show len

readLen :: IO Int
readLen = do
  str <- getLine
  return (length str)
```

(14) greet :: IO ()

```
greet =
    putStr "Name"
  >>=
  \_ -> getLine
  >>=
  \name -> (putStr "town?"
          >>=
          \_ -> getLine
          >>=
          \town ->
          let msg = name ++ town
          in putStrLn msg
          )
```

```
greet :: IO ()
greet = do
  putStr "Name"
  name <- getLine
  putStr "Town?"
  town <- getLine
  let msg = Name ++ town
  putStrLn msg
  -- putStrLn (Name ++ town)
```

Prolog

① len([], 0).
len([_|T], N) :-
  len(T, T), N is T + 1.

leng(List, N) :- aulen(List, 0, N)

② aulen([], A, A). △
aulen([_|T], A, N) :-
  Anew = A + 1,
  aulen(T, Anew, N).

=<

③ accMax(List, Max) :- △
  List = [H|_],
  accMax(List, H, Max).

accMax([], M, M).
accMax([H|T], A, M) :-
  (H < A →
    accMax(T, A, M)
  ; H >= A =<
    accMax(T, H, M)
).

④ append([], A, A).
append([H|T], A, [H|L]) :-
  append(T, A, L).

⑤ reverse △

reverse([A|B], C) :-
  sameLength([A|B], C),
  reverse1([A|B], C).

sameLength([], []).
sameLength([_|T], [_|M]) :-
  sameLength(T, M).

reverse1([], Z).
reverse1([A|B], C) :-
  reverse1(B, CB),
  append(CB, [A], CBA).

tail
reverseNew(List, Rev) :-
  accRev(List, [], Rev).

accRev([], Rev, Rev).
accRev([H|T], A, R) :-
  accRev(T, [H|A], R).

⑥ prefix(P, L) :- append(P, _, L).
suffix(S, L) :- append(_, S, L).

sublist(SubL, L) :- prefix(S, L),
  suffix(SubL, S).

⑦ take(N, List, Front) :-
  length(Front, N),
  append(Front, _, List).

(8) member(Elt, Lst) :-
append(_, [Elt|_], Lst).

member2(Elt, [Elt|_]).
member2(Elt, [_|T]) :-
member2(Elt, T).

(9) fact(N, F) :-
(N == 0 ->
F = 1
; N > 0 ->
N1 is N - 1,
F1 is F * N,
fact(N1, F1)
).

fact(N, F) :- fact1(N, A, F).
fact1(N, A, F) :-
(N = 0 ->
F = A
; N > 0,
N1 is N - 1,
A1 is A * N,
fact1(N1, A1, F)
).

(10) map(_, [], []).
map(P, [X|Xs], [Y, Ys]) :-
call(P, X, Y),
map(P, Xs, Ys).

(11) adjacent(E1, E2, Lst) :-
append(_, [E1, E2|_], Lst).
adjacent(E1, E2, [E1, E2|_]).
adjacent(E1, E2, [_|Tail]) :-
adjacent(E1, E2, Tail).

(12) SumList(List, Sum) :-
SumList(List, 0, Sum).
SumList([], A, A).
SumList([H|T], A, S) :-
A1 is A + H,
SumList(T, A1, S).

12-1

(a) :t filter :: (a->Bool) -> [a] -> [a] ✓

(b) :t (+) Num a => a -> a -> a ✓

(c) [1,2] ++ [] ++ [3] → [1,2,3] [1,2,3].

(d) map (\x -> [x]) [2] => [[2]] [[2]]

(e) foldl (++) [4] [[1,2],[]]

=> [4,1,2] ✓

(f) let x = x+1 in 1+1 => 2 ✓

13-1

(a) :t (+3) ~~Integer~~ Num a => a -> a

(b) :t foldr (a->b->b) -> b -> [a] -> b

(c) :t return :: ~~Monad a~~ Monad m => a -> m a ☆

(d) filter (\x->True) [2] [[~~True~~]] [2]

(e) [Nothing] ++ [Nothing] ~~[Nothing]~~ [Nothing, Nothing].

(f) [Just] [Just] ~~[Just Just]~~ type error

:t (>>=) Monad m => ma -> (a -> m b) -> m b ☆

getChar :: IO Char

getLine :: IO String

putChar :: Char -> IO()

putStr :: String -> IO ()

putStrLn :: String -> IO ()

print :: (Show a) => a -> IO ()

10-2.

1. (a) Ord a ⇒ a → a → Bool ✓ :t (<)

(b)

map :: (a → b) → [a] → [b] :t map (+3)

(+3)

map +3 :: Num a ⇒ [a] → [a] ✓

(c) type error ✓ map (length<3) [[1], [1,2,3]]

(d) [1,2] ✓ filter (not . (==3)) [1,2,3]

(e) 3 ✓ let e = head [] in 3

10-1

(a) :t map :: (a→b) →[a]→[b] ✓

(b) map tail [[1,2],[]] type error ✓

(c) filter (length . (==2)) [[1,2],[]] type error ✓ (==2). length

(d) foldl (+) 4 [1,2] ⇒ 7 ✓

(e) ~~let e = 3:e in head e~~ ⇒ 3 ✓

11-1

a) :t (++) [a] → [a] → [a] ✓

(b) :t filter (a→Bool) →[a] → [a] ✓

(c) :t take ~~a~~ → [a] → [a] ~~≠~~ Int →[a]→[a]

(d) map (:[4]) [1,2] [[1,4],[2,4]] ✓

(e) ((drop 7) . (take 10)) [1..] [8,9,10] ✓

(f) foldl (++) [4] [[1,2],[]]

(++) [4,1,2] ✓

foldr (++) [4] [[1,2],[]]

① odd 3 ⇒ True Odd :: Integral a ⇒ a → Bool
odd 6 ⇒ False

② compare 2 3 ⇒ LT
compare 3 3 ⇒ EQ
compare 3 2 ⇒ GT

③ sqrt 3.

④ head：取出列表第一个元素
☆ head [1,2,3,4] ⇒ 1.
tail：取出列表除了第一个，其它元素.
tail [1,2,3,4] ⇒ [2,3,4]

⑤ chop 2 [1,2,3,4,5] 列表
⇒ [3,4,5]
take 2 [1,2,3,4,5]
⇒ [1,2]
take :: Int → [a] → [a]

⑥ 元组 ☆
fst (1, 'a') ⇒ 1
snd (1, 'a') ⇒ 'a'
fst :: (a,b) ⇒ a.

⑦ lines :: String → [String]
lines "The quick\n brown fox"
⇒ ["The quick", " brown fox"]

⑧ add a b = a + b.

⑨ null :: [a] → Bool
检查一个列表是否为空.
null [] ⇒ True.

(||) :: Bool → Bool → Bool
True || False ⇒ True

⑩ 列表最后一个元素 last :: [a] → a
last [1,2,3,4,5] ⇒ 5

⑪ :type (==)
(==) :: Eq a ⇒ a → a → Bool

⑫ sum [1,2] ⇒ 3

① isEmpty :: [a] → Bool
isEmpty [] = True
isEmpty (_ : _) = False
不关心

② ["a", "b"] ⇒ [[Char]]
& &

flip zip "Hello"
flip :: (a → b → c) → b → a → c
zip :: [d] → [e] → [(d, e)]
zip "Hello" [Char] :: [e] → [(Char, e)]
flip zip "Hello"
[Char] → [(Char, e)]

foldl :: (b → a → b) → b → [a] → b.

① map length

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

length :: [a] -> Int

map length :: [[a]] -> [Int]

② map (+3)

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

+3 :: Num a => a -> a

map (+3) :: Num a => [a] -> [a]

③ zip [True, False, False]

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

[b] -> [(Bool, b)]

④ flip filter "hello"

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

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

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

flip filter "Hello" :: (Char -> Bool) -> [Char]

[Char]

[a]

⑤ (. length)

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

length :: [a] -> Int

[Int -> c] -> [a] -> c

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

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

原 true 的保留下来

foldl :: (b->a->b) -> b -> [a] -> b

foldr :: (a->b->b) -> b -> [a] -> b

zipWith :: (a->b->c) -> [a] -> [b] -> [c]

* 对应元素回来，运行上面

length :: [a] -> Int

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

concatMap :: (a -> [b]) -> [a] -> [b]

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

concatMap f lst

= foldr (++) [] (map f lst)

any :: (a -> Bool) -> [a] -> Bool

all :: (a -> Bool) -> [a] -> Bool

(+) :: Num a => a -> a -> a

(++) :: [a] -> [a] -> [a]

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

map

(a->b) -> [a] -> [b]

map map (c->d) -> [c] -> [d]

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

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

zip "Hello"

:: [Char]

map zip "Hello"

[b] -> [(Char, b)]

```
p(1,2). p(2,3). p(2,6). p(3,4)

q(A,B) :- p(A,B).
q(A,C) :- q(A,B), q(B,C).

  p(1,2) . p(2,3). p(2,6). p(3,4)
+ q(1,2). q(2,3). q(2,6). q(3,4).

  q(1,3), q(1,6). q(2,4). q(1,4).

multiply(X,Y,XY) :-
    multiply(X,Y,0,XY).

multiply(X,Y,A,XY) :-
    ( X = 0 ->
      XY = A
    ; X1 is X-1,
      A1 is A+Y,
      multiply(X1,Y,A1,XY)
    ).

fact(N,A,F)

map(_,[],[]).
map(P,[X|Xs],[Y|Ys]) :-
    call(P,X,Y),
    map(P,Xs,Ys).

filter(_,[],[]).
filter(P,[X|Xs],Filtered) :-
    ( P(X) ->
      Filtered = [X|Filtered1]
    ; Filtered = Filtered1
    ),
    filter(P,Xs,Filtered1).
```

```
Map = F ⇒ L ⇒ Result,
| L = [_], Result = [_],
| L = [E], ES = [E],
| Map = F ⇒ ES ⇒ T,
| F = E ⇒ H.
| Result = [H], Result = T.

Filter = F ⇒ L ⇒ Result
| L = [_], Result = [_],
| L = [x], XS = [x],
| Filter = F ⇒ XS ⇒ FxS,
| F ⇒ X ⇒ bool,
| X = [x], Result = [x], FxS = [x],
| Result = FxS.
```

```
main :: IO ()
main = do
  putStrLn "Sirly ?"
  len ← readLen
  putStrLn $ "++ show len

readLen :: IO Int
readLen = do
  ser ← getLine,
  return (length ser)
```

```
greet :: IO ()
greet =
  putStr "Greet! What your name?"
  >>=
  \_ → getLine
  >>=
  \name → ( putStr "from"
            >>=
            \_ → getLine
            \town →
              let msg = name ++ town in putStrLn msg
```

```
greet :: IO ()
greet = do
  putStr "name ?"
  name ← getLine
  putStr "from ?"
  town ← getLine
  let msg = name ++ town
  putStrLn msg
```

~~Monad~~

Maybe Monad

Tree 同等

```
data Maybe t = Just t | Nothing

return x = Just x

(Just x) >>= f = f x

Nothing >>= _ = Nothing
```

MaybeOK

```
data MaybeOK = OK t | Error String

return x = OK x

(OK x) >>= f = f x

(Error m) >>= _ = Error m
```

Reading Input

```
getChar :: IO Char

getLine :: IO String
```

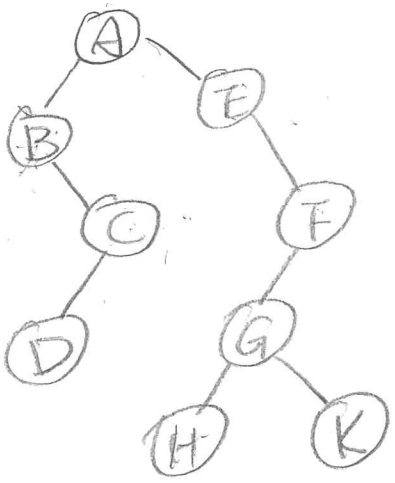
Writing

```
putChar :: Char -> IO ()

putStr :: String -> IO ()

putStrLn :: String -> IO ()

print :: (Show a) => a -> IO ()
```



pre-order：前根先，左右到右.

in-order：先左，根，右.

post-order：先左，右，根.

pre-order: A→B.C.D.E.F.G.H.K.

in-order: B D.C.A.E H.G.K.F.

post-order: D.C.B.H.K.G.F.E.A.

① Planked

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

② quicksort :: Ord a => [a] -> [a]

```
sort [] = []
sort (pivot:xs) = sort lessor ++ [pivot] ++ sort larger
  where lessor = filter (< pivot) xs
        larger = filter (>= pivot) xs
```

③ data Tree k v = Leaf | Node k v (Tree k v) (Tree k v)

⇒ same shape deriving (Eq, show).

```
Sameshape :: Tree a b -> Tree a b -> Bool
Shape Leaf (Node _ _ _ _) = False
Shape (Node _ _ _ _) Leaf = False
Shape Leaf Leaf = True
Shape (Node _ _ l1 r1) (Node _ _ l2 r2) =
  (Shape l1 l2) && (Shape r1 r2)
```

data Tree a = Empty | Node (Tree a) a (Tree a)

④ treesort :: Ord a => [a] -> [a]

```
treesort = inorder (int_bst List)

int_bst :: Ord a => [a] -> Tree a
int_bst [] = Empty
int_bst (x:xs) = insert x (int_bst xs)
```

⑤ map_tree :: (a->a) -> Tree a -> Tree

```
map_tree f Empty = Empty
map_tree f (Node l n r)
  = Node (map_tree f l)
         (f n)
         (map_tree f r)
```

⑥ foldr_tree :: (a->b->a->a) -> a -> Tree b -> a

```
foldr_tree f b Empty = b
foldr_tree f b (Node l n r)
  = f (foldr_tree f b l) n
      (foldr_tree f b r)

height_tree = foldr_tree node_1 0
  where node_1 _ l _ r = 1 + (max l r)

ternary :: (a->a->a) -> a->a->a->a
ternary f x y z = f x (f y z)
sum_tree = foldr_tree (ternary (+)) 0
```

```
int_insert :: Ord a => a -> Tree a -> Tree a
int_insert n Empty = Node Empty n Empty
int_insert n (Node l v r)
  | n < v  = Node (insert n l) v r
  | n >= v = Node l v (insert n r)

inorder :: Ord a => Tree a -> [a]
inorder Empty = []
inorder (Node l n r) =
  inorder l ++ [n] ++ inorder r
```

empty. tree(L,N,R). Prolog

① inset_member(N,set)

member(N, tree(_,N,_)).
member(N, tree(L,V,R)):-
(N<V →
member(N,L)
; N>V
member(N,R)
).

② inset_insert(N,set0,set)

insert(N, empty, tree(empty,N,empty)).
insert(N, tree(L,N,R), tree(L,N,R)).
insert(N, tree(L,V,R), Result):-
(N<V →
Result = tree(L1,V,R),
insert(N,L,L1)
; N>V,
Result = tree(L,V,R1),
insert(N,R,R1)
).

③ tree_list(Tree,List). tree → list

empty node(L,N,R)

tree_list(empty,[]).
tree_list(node(L,N,R),List):-
tree_list(L,List1),
tree_list(R,List2),
append(List1,[N|List2],List).

④ No append.

tree_list(Tree,List) = tree_list(Tree,[],List).

tree_list(empty,List,List).
tree_list(node(L,N,R),List_in,List_out):-
tree_list(R,List_in,List1),
tree_list(L,[N|List1],List_out).

⑤ balanced tree

list_tree(List,Tree)

list_tree([],empty).
list_tree([E|Tail], node(L,N,R)):-
length(Tail,Len),
Len1 = Len//2,
length(Front,Len1),
list_tree(Front,L),
list_tree(Back,R),
append(Front,[N|Back],[E|Tail]).

let (a, b) = tt' tree.

in a/b

Haskell 擡小写

prolog 大写.

tt' Nil = (0.0, 0.0)

tt' (Node n l m r)

= let (suml, cntl) = tt' l

(,) = tt' m

(,) = tt' r

in (n + suml ...)(1 + ...)

filter (\x -> x `mod` 2 == 0) [1,2,3,4]

filter (\x -> length x < 2) ["a", "abc"]

countnodes :: Tree -> Int
countnodes Leaf = 0
countnodes (tree k v l r) =
  let c1 = countnodes l
      c2 = countnodes r
  in 1 + c1 + c2

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

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

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

foldl :: (b -> a -> b) -> b -> [a] -> b

foldr :: (a -> b -> b) -> b -> [a] -> b

zipwith :: (a -> b -> c) -> [a] -> [b] -> [c]

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

length :: [a] -> Int

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

return :: Monad m => a -> m a

(>>=) :: Monad m => m a -> (a -> m b) -> m b

print :: Show a => a -> IO ()

take :: Int -> [a] -> [a]

[[1,4],[2,4]]

concatmap :: (a -> [b]) -> [a] -> [b]

any :: (a -> Bool) -> [a] -> Bool

all :: (a -> Bool) -> [a] -> Bool

Filter = F => L => Result,
L = [_], Result = [_],
L = [x], [Xs] = [X]
Filter = F => Xs => Fxs
F = X => Bool,
Result = [x], Fxs = [X],
Result = Fxs.

Map = F => L => Result,
L = [_], Result = [_],
L = [E], [Es] = [E],
Map = F => Es => T,
F = E => H,
Result = [H], Result = T,

intset - insert (N, empty. tree(empty, N, empty)).

intset - insert (N, tree(~~Node~~ Left, Val, Right), Result) :-

{ N = Val →
  Result = tree(Left. Val. Right)

; N < Val →
  ~~Result~~ = tree(Left1, Val. Right),
  intset - insert (N, Left, Left1)

; N > Val,
  Result = tree (Left, Val, Right1),
  intset - insert (N, Right, Right1)

}.

sumlist (Left, 0, Sum)

sumlist ([ ], Sum, Sum).

sumlist ([N|Ns], Sum0, Sum) :-

Sum1 = Sum0 + N,

sumlist (Ns, Sum1, Sum).

(<) :: Ord a => a -> a -> Bool

map (+3) :: Num a => [a] -> [a]

foldr :: (a -> b -> b) -> b -> [a] -> b

Nothing :: Maybe a

zip [True, True, False]

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

[Bool]

[a] -> [(Bool, a)]

flip filter "hello"

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

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

[a] -> (a -> Bool) -> [a]

[char] -> (Char -> Bool) -> [char]

[(b, True)]

[8, 4, 0, 0, 9]

[(3,8), (0,4), (0,0), (2,0), (0,9)]

(0,0), (0,4), (0,9), (2,0), (3,8)

(d,1), (e,2), (c,3), (b,4)

[(c,3), (d,1), (e,2), (b,4)]

[3, 1, 2, 4]

let (sum, count) =

in sum / count

let