Functional Programming

Lau Chi Yung

HKOI

2017/02/25

Outline

- You will learn to write programs will less bugs
- You will learn to solve a problem more elegantly
- You will learn a new programming language called Haskell
- You will not learn new algorithms
- You will not learn to solve new problems

Outline

Which of the followings are functions?

A

```
void hello() {
  printf("hello");
}
```

B

```
int count() {
  static int i = 0;
  return i++;
}
```

C

```
int cube(int x) {
  return x * x * x;
}
```

D

```
int one() {
  return 1;
}
```

E

```
int identity(int x) {
  int i = 0;
  while (i < 1e9)
    i++;
  return x;
}
```

Pure function

$$f(input) = output$$

A function is said to be pure if it

- always evaluates to the same value *and*
- does not cause any *side effects*
  - standard output
  - write to harddisks
  - change of global variables
  - time delay is not considered as a side effect

Purely functional programming

$$f(x) = 2x^2 + 1$$
$$g(x) = 3x - 1$$
$$main = f(3)g(4) + g(4)$$

Programming with exclusive use of pure functions

- easier to separate into independent parts
- easier to debug
- allows more compiler optimizations
  - memoization
  - parallelism
  - lazy evaluation

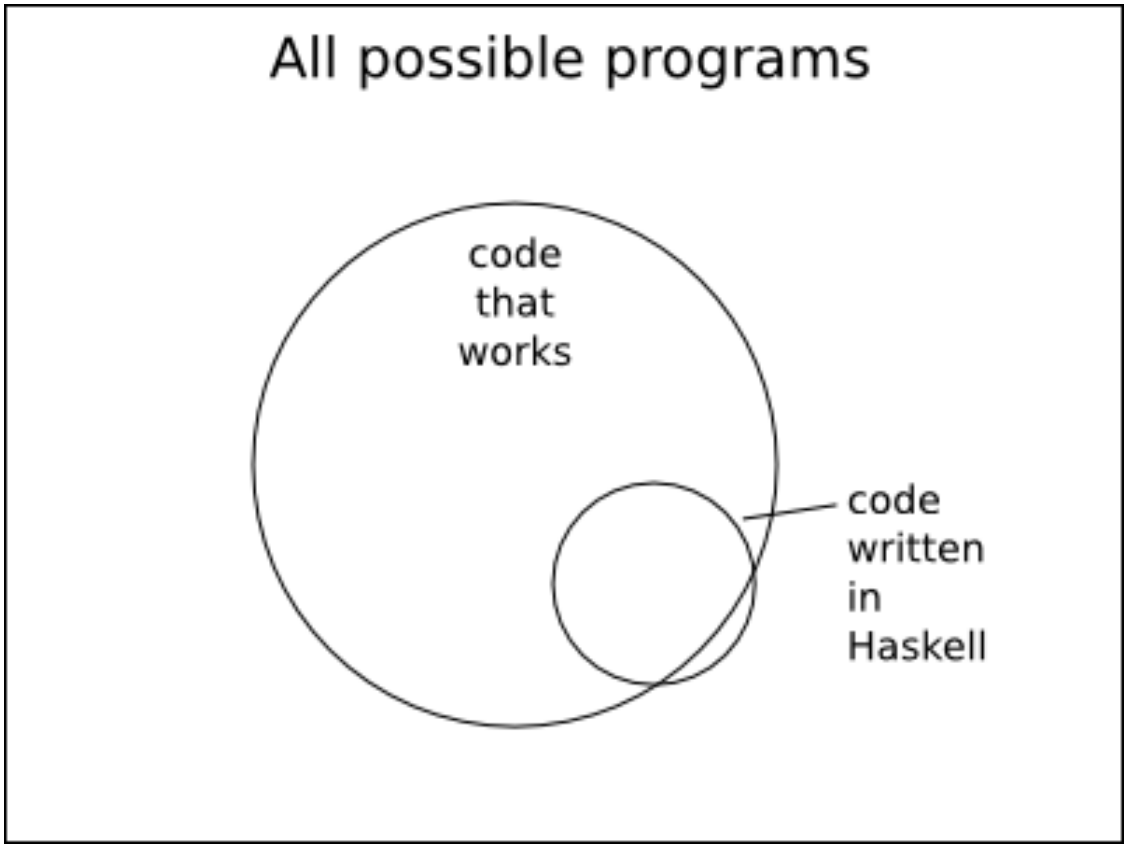
Higher-order function

Functions that

- take a function as input
  - std :: sort (a, a + n, cmp);
- or output a function
  - Differentiation
  - $f(x) = 2x^2 + 1$

    $$f'(x) = \frac{\mathrm{d}}{\mathrm{d}x}(2x^2 + 1) = 4x$$

Haskell



Haskell

- Purely functional programming language
- Available in many online judges
  - HKOI Online Judge
  - Codeforces
  - Timus
- Available in Facebook Hacker Cup, Google Code Jam
- Not available in TFT, HKOI, NOI, IOI, ACM-Local

My advice

- Try to forget about C, C++ and Pascal
- Think in Mathematical way, not computer way

Variables


```
x = 123
y = x * 2 + 3
-- single-line comment
```

Variables


```
x = 123
y = x * 2 + 3
-- single-line comment
```

WRONG

```
x = 123
x = 456
-- What is the value of x?
```

- Variable names must start with a lower-case letter

Types


```
a :: Int          -- at least 30-bit integer
a = 123

b :: Float                -- single-precision
b = 1.23

c = (1.23 :: Double) -- double-precision

d = 'a'
d :: Char
```

- Data types start with a capital letter

Types


```
a = 1.23                    -- inferred as Float
b = 1.23 :: Double

c = 1.23 + a :: Float
d = 1.23 + b                -- inferred as Double
e = a + b
-- Compilation error: what is the type of e?
```

- Type annotations are optional
- The compiler can guess the type automatically, aka Type inference
- All variables must have a consistent type

List


```
list1 = [1, 2, 3, 4] :: [Int]

list2 = ['a', 'b', 'c'] :: [Char]

-- equivalent
list3 = "abc" :: String

list4 = list1 ++ [5, 6]
-- [1, 2, 3, 4, 5, 6]
```

List


```
a = [10..15] :: [Int]
-- [10, 11, 12, 13, 14, 15]

b = [10,20..] :: [Int]
-- [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 1
-- infinite list

c = take 6 b
-- same as a
-- lazy evaluation

d = [2 * x | x <- [0..], x < 5]
-- [0, 2, 4, 6, 8]
```

$$d = \{2x \mid x \in \mathbb{N}, x < 5\}$$

Tuple


```
tuple :: (Int, Char, String)
tuple = (123, 'B', "Hello")

notTuple :: (Int)
notTuple = (123)          -- just an Int

nullTuple :: ()
nullTuple = ()
```

Function


```
e :: Int                          -- nullary function
e = 10

f :: Int -> Int                   -- unary function
f x = x * x

g :: Int -> Int -> Int            -- binary function
g x y = x + y

h = g 10 (f 10)
```

- Compact function syntax
- Function names must start with lower case too
- What is the value of h?

Function


```
e :: Int                          -- nullary function
e = 10

f :: Int -> Int                   -- unary function
f x = x * x

g :: Int -> Int -> Int            -- binary function
g x y = x + y

h = g 10 (f 10)
```

- Compact function syntax
- Function names must start with lower case too
- What is the value of h?
- h = 110

Anonymous function


```
f :: Int -> Int
f = \x -> x + 1
-- f x = x + 1

g :: Int -> Int -> Int
g = \x y -> x + y
-- g x y = x + y
```

- "\" stands for λ (lambda)
- Originated from *Lambda Calculus*, the basis of functional programming

if then else


```
f :: Int -> Int
f n =
  if n == 0
    then 1
    else n * f (n - 1)
```

- There must be an **else** part

case of


```
f :: Char -> Int
f c =
  case c of
    '1' -> 1
    '2' -> 2
    '3' -> 3

one = f '1'      -- 1
four = f '4'     -- runtime error
```

case of


```
f :: Char -> Int
f c =
  case c of
    '1' -> 1
    '2' -> 2
    '3' -> 3
    x   -> 0
one = f '1'      -- 1
four = f '4'     -- 0
```

case of


```
f :: Char -> Int
f c =
  case c of
    '1' -> 1
    '2' -> 2
    '3' -> 3
    _   -> 0
one = f '1'      -- 1
four = f '4'     -- 0
```

```
f :: Char -> Char -> Int
f '1' '2' = 12
f '1' '3' = 13
f '1' '4' = 14
f _ _     = 0

one = f '1' '2'     -- 12
four = f '4' '4'    -- 0
```

- Hard coding becomes easy
- "_" is a special character to denote something that we don't care

Guards


```
f :: Int -> Int
f x | x == 0      = 0
    | x /= 0      = 1
    | otherwise   = 2    -- never reached
```

$$f(x) = \begin{cases} 0 & \text{if } x = 0 \\ 1 & \text{if } x \neq 0 \\ 2 & \text{otherwise} \end{cases}$$

- Haskell is indentation sensitive
- Conditions are tested from top to bottom
- Runtime error if nothing is matched

Binary function?


```
g :: Int -> Int -> Int
g x y = x + y
```

```
-- (->) is right-associative
g :: Int -> (Int -> Int)
g x = h
  where
    h :: Int -> Int
    h y = x + y
```

- The above are equilvalent
- All functions are unary functions!
- Life is much easier

Partial application


```
g :: Int -> Int -> Int
g x y = x + y

three :: Int
three = g 1 2

addOne :: ?          -- Which type is it?
addOne = g 1

four :: Int
four = addOne three
```

Partial application


```
g :: Int -> Int -> Int
g x y = x + y

three :: Int
three = g 1 2

addOne :: Int -> Int
addOne = g 1

four :: Int
four = addOne three
```

Partial application


```
g :: Int -> Int -> Int -> Int
g x y z = (x + y + z) * (x + y + z)

f :: Int -> Int -> Int
f x y = (x + y) * (x + y)
```

Exercise

By making use of g, come up with a simpler definition of f

Partial application


```
g :: Int -> Int -> Int -> Int
g x y z = (x + y + z) * (x + y + z)

f :: Int -> Int -> Int
f x y = (x + y) * (x + y)
```

Exercise

By making use of g, come up with a simpler definition of f

```
f :: Int -> Int -> Int
f = g 0
```

Function as input


```
evalAtHalf :: (Double -> Double) -> Double
evalAtHalf f =
  f :: ?                    -- which type?
  f 0.5

square :: Double -> Double
square x = x * x

a :: Double
a = evalAtHalf square    -- 0.25
```

Function as input


```
evalAtHalf :: (Double -> Double) -> Double
evalAtHalf f =
  f :: Double -> Double
  f 0.5

square :: Double -> Double
square x = x * x

a :: Double
a = evalAtHalf square    -- 0.25
```

Composition

$$(f \circ g)(x) = f(g(x))$$

```
f '-' = -0.5
f '+' = 0.5

g x = if x < 0 then '-' else '+'

compose :: ?

fg = compose f g
fg 10      -- 0.5
```

Exercise

Implement the function "compose"

Composition

$$(f \circ g)(x) = f(g(x))$$

```
f '-' = -0.5
f '+' = 0.5

g x = if x < 0 then '-' else '+'

compose :: (Char -> Double) ->
           (Int  ->   Char) -> (Int -> Double)
compose f g = \x -> f (g x)

fg = compose f g
fg 10     -- 0.5
```

Type variable


```
evalAtHalf :: -- (Double -> Double) -> Double ?
evalAtHalf f =
  f :: --Double -> Double ?
  f 0.5

square :: Double -> Double
square x = x * x

sign :: Double -> Char
sign x = if x < 0  then '-' else '+'

a = evalAtHalf square  -- 0.25
b = evalAtHalf sign    -- '+'
```

Type variable


```
evalAtHalf :: (Double -> a) -> a
evalAtHalf f =
  f :: Double -> a
  f 0.5

square :: Double -> Double
square x = x * x

sign :: Double -> Char
sign x = if x < 0  then '-' else '+'

a = evalAtHalf square  -- 0.25
b = evalAtHalf sign    -- '+'
```

Type constraint


```
evalAtHalf :: -- (Double -> a) -> a ?
evalAtHalf f =
  f :: -- Double -> a ?
  f 0.5

square :: Double -> Double
square x = x * x

sign :: Float -> Char
sign x = if x < 0 then '-' else '+'

a = evalAtHalf square  -- 0.25
b = evalAtHalf sign    -- '+'
```

Type constraint


```
evalAtHalf :: (b -> a) -> a
evalAtHalf f =
  f :: b -> a
  f 0.5

square :: Double -> Double
square x = x * x

sign :: Float -> Char
sign x = if x < 0  then '-' else '+'

a = evalAtHalf square  -- 0.25
b = evalAtHalf sign    -- '+'
```

Type constraint


```
evalAtHalf :: Floating b => (b -> a) -> a
evalAtHalf f =
  f :: Floating b => b -> a
  f 0.5

square :: Double -> Double
square x = x * x

sign :: Float -> Char
sign x = if x < 0 then '-' else '+'

a = evalAtHalf square  -- 0.25
b = evalAtHalf sign    -- '+'
```

Integral


```
x :: Int          -- at least 30-bit integer
x = 1000

z :: Integer      -- arbitrary size
z = 99999999999999999999999999999999999999999999

f :: Integral a => a -> a -> a
f a b | a < b  = f b a
      | b == 0 = a
      | b /= 0 = f (a - b) b
```

Type classes

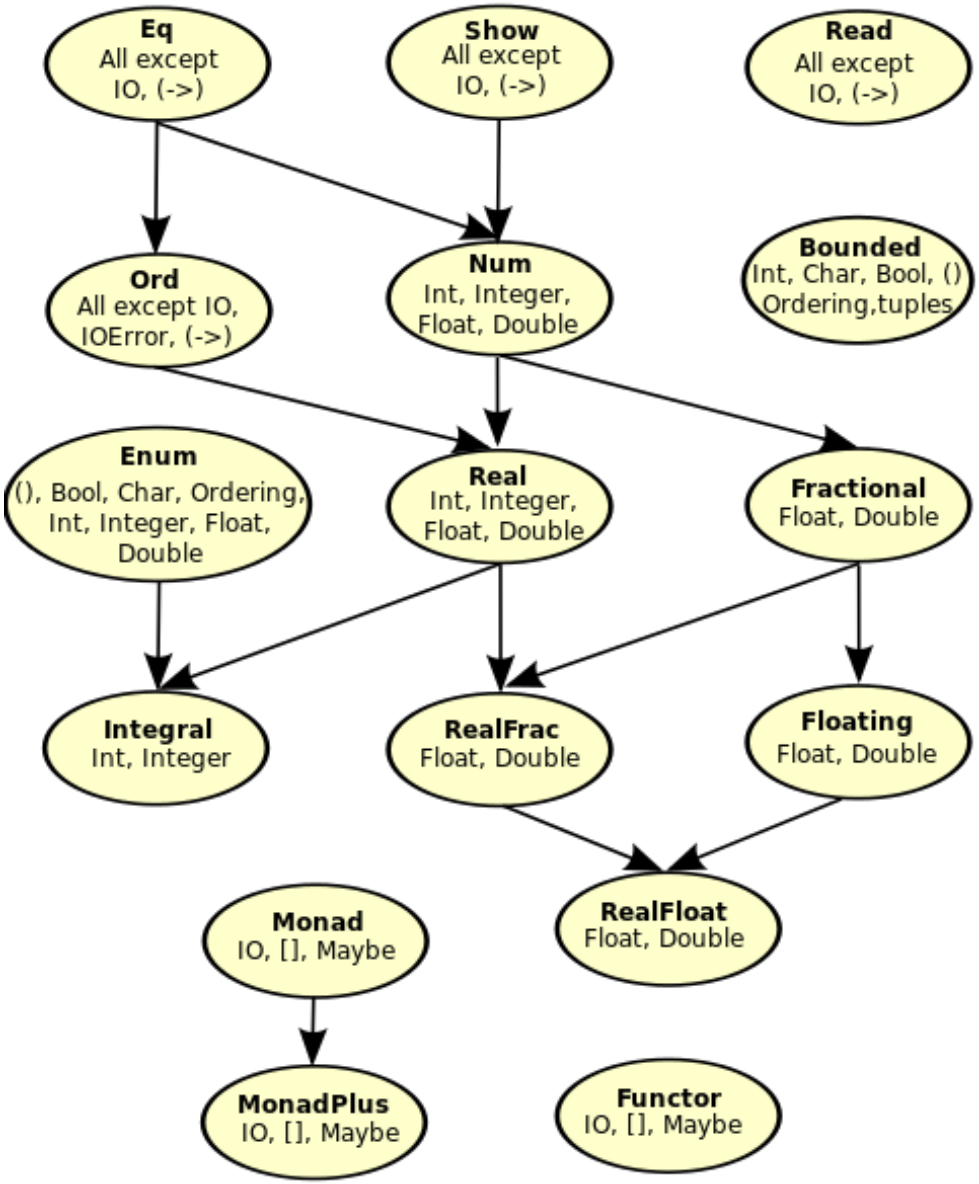

```
(==) :: Eq a => a -> a -> Bool

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

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

- Eq, Ord, Num, Floating, Integral are **Type classes**
- Int, Integer, Float, Double, Char, Bool are **Concrete types**

Type hierarchy



Defining types


```
data Color = Red | Green | Blue
c = Red :: Color

f :: Color -> Int
f Red   = 1
f Green = 2
f Blue  = 3
```

- Type names (Color) and value constructors (Red, Green, Blue) must start with a capital letter

Maybe


```
data Maybe a = Nothing | Just a

a = Just 'C' :: Maybe Char
b = Nothing  :: Maybe Float

f :: Int -> Maybe Int
f x | x > 0  = Just (x * y)
    | x == 0 = Just 1
    | x < 0  = Nothing
    where Just y = f (x - 1)
```

- Predefined in Haskell
- Helpful to represent something that might fail
- **Maybe Int** is a concrete type
- **Just** 123 is a concrete value

Either


```
data Either a b = Left a | Right b

f :: Int -> Either String Int
f x | x > 0  = Right (x * y)
    | x == 0 = Right 1
    | x < 0  = Left "x must not be negative"
  where Right y = f (x - 1)
```

- Predefined in Haskell
- "Right" also means "correct"
- "Left" usually stores error information

List


```
(:) :: a -> [a] -> [a]
(++) :: [a] -> [a] -> [a]
-- predefined in Haskell

a = (:) 123 ((:) 456 [])

b = 123 : 456 : []

c = [123, 456]

second :: [Int] -> Int
second [] = 0
second (x1:x2:xs) = x2
```

List


```
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (a:as) =
  qsort left ++ [a] ++ qsort right
  where left  = [x | x <- as, x < a]
        right = [x | x <- as, x >= a]
```

Prelude

- The standard library that is imported by default
- Defines a whole bunch of elementary functions and data types
- Let's glance through it

Bool

- **data Bool = False | True**
- **(&&) :: Bool -> Bool -> Bool**
- **(||) :: Bool -> Bool -> Bool**
- **not :: Bool -> Bool**
- **otherwise :: Bool**

Eq

- (==) :: **Eq** a =>a -> a -> **Bool**
- (/=) :: **Eq** a =>a -> a -> **Bool**

Ord

- (<) :: **Ord** a =>a -> a -> **Bool**
- (<=) :: **Ord** a =>a -> a -> **Bool**
- (>) :: **Ord** a =>a -> a -> **Bool**
- (>=) :: **Ord** a =>a -> a -> **Bool**
- **min** :: **Ord** a =>a -> a -> a
- **max** :: **Ord** a =>a -> a -> a

Bounded

- **minBound** :: **Bounded** a =>a
- **maxBound** :: **Bounded** a =>a

```
minBound :: Integer  -- compilation error

minBound :: Bool     -- False
maxBound :: Bool     -- True

minBound :: Int      -- -9223372036854775808
maxBound :: Int      --  9223372036854775807
```

Enum

- **succ** :: **Enum** a =>a −> a
- **pred** :: **Enum** a =>a −> a
- **toEnum** :: **Enum** a =>**Int** −> a
- **fromEnum** :: **Enum** a =>a −> **Int**

```
succ 'A'                -- 'B'
succ False              -- True
succ True               -- runtime error
pred 123                -- 122
fromEnum 'A'            -- 65
toEnum 66 :: Char       -- 'B'
```

Num

- $(+)$ :: **Num** a =>a −> a −> a
- $(-)$ :: **Num** a =>a −> a −> a
- $(*)$ :: **Num** a =>a −> a −> a
- **negate** :: **Num** a =>a −> a
- **abs** :: **Num** a =>a −> a
- **signum** :: **Num** a =>a −> a

Integral

- **div** :: **Integral** a =>a -> a -> a
  (truncate towards $-\infty$)
- **mod** :: **Integral** a =>a -> a -> a
- **gcd** :: **Integral** a =>a -> a -> a
- **lcm** :: **Integral** a =>a -> a -> a
- **even** :: **Integral** a =>a -> **Bool**
- **odd** :: **Integral** a =>a -> **Bool**
- **toInteger** :: **Integral** a =>a -> **Integer**
- **fromInteger** :: **Num** a =>**Integer** -> a
- **fromIntegral** :: (**Integral** a, **Num** b) =>a -> b

Floating

- (/) :: **Fractional** a =>a -> a -> a
- **recip** :: **Fractional** a =>a -> a
- **pi** :: **Floating** a =>a
- **exp** :: **Floating** a =>a -> a
- **log** :: **Floating** a =>a -> a
- **sqrt** :: **Floating** a =>a -> a
- (**) :: **Floating** a =>a -> a -> a --power
- **sin** :: **Floating** a =>a -> a
- **cos** :: **Floating** a =>a -> a
- **tan** :: **Floating** a =>a -> a
- **atan2** :: **Floating** a =>a -> a -> a
- **ceiling** :: (**Floating** a, **Integral** b) =>a -> b
- **floor** :: (**Floating** a, **Integral** b) =>a -> b

Miscellaneous

- **id** :: a -> a
- **const** :: a -> b -> a
- (.) :: (b -> c) -> (a -> b) -> a -> c
- **flip** :: (a -> b -> c) -> b -> a -> c
- ($) :: (a -> b) -> a -> b

```
a = (f . g) x
b = f . g $ x
```

Tuple

- **fst** :: (a, b) -> a
- **snd** :: (a, b) -> b
- **curry** :: ((a, b) -> c) -> a -> b -> c
- **uncurry** :: (a -> b -> c) -> (a, b) -> c
- (,) :: a -> b -> (a, b)
- (,,) :: a -> b -> c -> (a, b, c)
- (,,,) :: a -> b -> c -> d -> (a, b, c, d)
- (,,,,) :: a -> b -> c -> d -> e -> (a, b, c, d, e)
- ...
- Up to 62-tuple

List (I)

- (:) :: a -> [a] -> [a]
- (++) :: [a] -> [a] -> [a]
- **head** :: [a] -> a
- **last** :: [a] -> a
- **init** :: [a] -> [a]
- **tail** :: [a] -> [a]

```
list1 = 'a' : 'b' : 'c' : []
list2 = ['a', 'b', 'c']
a  = head list2
c  = last list2
ab = init list2
bc = tail list2
```

List (II)

- **reverse** :: [a] -> [a]
- **cycle** :: [a] -> [a]
- **repeat** :: a -> [a]
- **replicate** :: **Int** -> [a] -> [a]
- **take** :: **Int** -> [a] -> [a]
- **drop** :: **Int** -> [a] -> [a]
- **elem** :: a -> [a] -> **Bool**
- **notElem** :: a -> [a] -> **Bool**

```
a = cycle [1, 0]   -- [1, 0, 1, 0, 1, 0, 1, 0,
b = repeat 0       -- [0, 0, 0, 0, 0, 0, 0, 0,
c = take 5 a       -- [1, 0, 1, 0, 1]
d = elem 1 a       -- True
```

List (III)

- **length** :: [a] -> **Int**
- **sum** :: **Num** a =>[a] -> a
- **product** :: **Num** a =>[a] -> a
- **maximum** :: **Num** a =>[a] -> a
- **minimum** :: **Num** a =>[a] -> a
- **zip** :: [a] -> [b] -> [(a, b)]
- **unzip** :: [(a, b)] -> ([a], [b])

```
a = zip [1..] "abc"
-- [(1, 'a'), (2, 'b'), (3, 'c')]
```

List (IV)

- **map** :: (a -> b) -> [a] -> [b]
- **filter** :: (a -> **Bool**) -> [a] -> [a]
- **takeWhile** :: (a -> **Bool**) -> [a] -> [a]
- **dropWhile** :: (a -> **Bool**) -> [a] -> [a]
- **zipWith** :: (a -> b -> c) -> [a] -> [b] -> [c]

```
a = map (\x -> x * x) [1..5]
-- [1, 4, 9, 16, 25]

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

zip = zipWith (,)
```

String

- **lines** :: **String** -> [**String**]
- **unlines** :: [**String**] -> **String**
- **words** :: **String** -> [**String**]
- **unwords** :: [**String**] -> **String**
- **read** :: **Read** a =>**String** -> a
- **show** :: **Show** a =>a -> **String**

```
a = unlines ["hello", "world"]
-- "hello\nworld"

b = unwords ["hello", "world"]
-- "hello world"

c = read "123" :: Int
d = show c
```

Hello, World!


```
main = interact f

f :: String -> String
f _ = "Hello, World!"
```

- Ignore the first line for now
- You can submit this program to the online judge!
- What you need to do is just to write a pure f

Factorial

Input	Output
5	120

```
main = interact f

f :: String -> String
f = show . g . read

g n | n == 0 = 1
    | n /= 0 = n * (g (n - 1))
```

A + B problem

Input	Output
1 2	3

```
main = interact f

f :: String -> String
f = show . sum . map read . lines
```

Enumeration

Input	Output
4	1 2 3 4 4 6 8 10 9 12 15 18 16 20 24 28

```
f :: String -> String
f = unlines . g . read

g :: Int -> [String]
g n = [h n i | i <- [1..n]]

h :: Int -> Int -> String
h n i = unwords . map show $ take n [i^2, i^2+i..]
```

Inverse Problem

Output N distinct integers that sums to M

Input	Output
4 8	0 8 1 -1

```
f :: String -> String
f = unwords . map show . g . map read . words

g :: [Int] -> [Int]
g [n, 0] | even n = take n h
g [n, m] | even n = take n (0 : m : hm)
         | odd  n = take n (    m : hm)
  where hm = [hm | hm <- h, hm /= m, hm /= -m]

h :: [Int]
h = concat [[x,-x] | x <- [1..]]
```

Functor


```
fmap :: Functor f => (a -> b) -> f a -> f b

a = fmap succ (Just 10)          -- Just 11
b = fmap succ Nothing            -- Nothing

a = fmap succ (Left "Error")     -- Left "Error"
b = fmap succ (Right 10)         -- Right 11

a = fmap succ [1, 3, 6]          -- [2, 4, 7]
```

Monad


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

f :: Int -> Just Int
f x | x > 0  = Just (x * y)
    | x == 0 = Just 1
    | x < 0  = Nothing
    where Just y = f (x - 1)

a = Just 3 >>= f >>= f  -- Just 720
a = Nothing >>= f       -- Nothing

b = [1..] >>= (\x -> [x, -x])
-- [1, -1, 2, -2, 3, -3, 4, -4, 5, -5, 6, -6,
```

I/O

- **putChar** :: **Char** -> **IO** ()
- **putStr** :: **String** -> **IO** ()
- **putStrLn** :: **String** -> **IO** ()
- **getChar** :: **IO Char**
- **getLine** :: **IO String**
- **getContents** :: **IO String**
- **interact** :: (**String** -> **String**) -> **IO** ()

```
main :: IO a
main = putStrLn "Hello, World!"
```

I/O

- IO is a Functor
- fmap :: **Functor** f =>(a -> b) -> f a -> f b
- fmap :: (a -> b) -> **IO** a -> **IO** b
- IO is a Monad
- (>>=) :: **Monad** m =>m a -> (a -> m b) -> m b
- (>>=) :: **IO** a -> (a -> **IO** b) -> **IO** b

```
main :: IO a
main =
  getChar >>= putChar
```

```
main :: IO a
main =
  fmap succ getChar >>= putChar
```