FUNCTIONAL PROGRAMMING
NO.8 FUNCTIONS

Tatsuya Hagino
hagino@sfc.keio.ac.jp

Slide URL

https://vu5.sfc.keio.ac.jp/slide/

Function Definition

```
fun pat_A1 pat_A2 .... | guard_A1 = exp_A1
                       | guard_A2 = exp_A2
                           :          :
fun pat_B1 pat_B2 .... | guard_B1 = exp_B1
                       | guard_B2 = exp_B2
                           :          :
```

- defines a function with pattern match
- Function name and variables are identifiers:
  - starts with a small alphabet letter
  - follows small and upper alphabet letters, numbers, underscore and single quote.
- The following identifiers are reserved and cannot be used:
  - **`case`**, **`class`**, **`data`**, **`default`**, **`deriving`**, **`do`**, **`else`**, **`if`**, **`import`**, **`in`**, **`infix`**, **`infixl`**, **`infixr`**, **`instance`**, **`let`**, **`module`**, **`newtype`**, **`of`**, **`then`**, **`type`**, **`where`**, **`_`**

Recursion

- Functional programming languages do not have for or while statements for repetition.
- Instead, use recursion to implement repetition.

```
factorial n = if n == 0 then 1
                        else n * factorial(n - 1)
```

recursive call of **`factorial`**

- Recursive call
  - Call itself directly or indirectly.
  - Divide and Conquer
    - Divide a big program into smaller ones.
    - Smaller ones are the same problem but small size.
    - Apply recursively until the size becomes small enough.

Recurrence Formula and Sequence

- Sequence defined by a recurrence formula.
  - a_n is defined in terms of a_{n-1} and/or a_{n-2} .

$$a_n = a_{n-1} + a_{n-2}$$

- Can be implemented by using recursive calls.

```
fib n | n == 0     = 1
      | n == 1     = 1
      | otherwise = fib(n - 1) + fib(n - 2)
```

Example

- Define a function **`sumn`** which adds numbers from 1 to n.

- Using **`sum`** :

```
sumn n = sum [1..n]
```

- The sum of 1 to n is equal to the sum of 1 to $n - 1$ and add n. Therefore, using recursion:

```
sumn n = if n == 0 then 0
                   else n + sumn(n-1)
```

Exercise 8-1

comb.hs

```
import System.Environment

main = do args <- getArgs
          print $ comb (read $ args !! 0) (read $ args !! 1)

comb n r | r == 0    = 1
         | n == r    = 1
         | otherwise = ...
```

- Define ${}_nC_r$ which is the number of combinations taking r elements from n elements.
  - ${}_nC_0 = {}_nC_n = 1$
  - ${}_nC_r = {}_{n-1}C_r + {}_{n-1}C_{r-1}$

```
% stack ghc comb.hs
...
% ./comb 10 5
252
%
```

The number of ways for the change

- If you buy a drink of 170 yen and pay 200 yen, you get 30 yen change.
- There are many ways of giving 30 yen change:
  - three 10 yen coins
  - six 5 yen coins
  - thirty 1 yen coins
  - ten 1 yen coins, two 5 yen coins and one 10 yen coins
  - and so on

- Let us calculate the number of ways for the change.
  - Let $a(n)$ be the number of ways to pay n yen by 1 yen coins only:
    - $a(n) = 1$
  - Let $b(n)$ be the number of ways to pay n yen with 1 yen and 5 yen coins:
    - $b(n) = a(n) \quad (n < 5)$
    - $b(n) = a(n) + b(n-5) \quad (n \geq 5)$
  - Let $c(n)$ be the number of ways to pay n yen with 1 yen, 5 yen and 10 yen coins:
    - $c(n) = b(n) \quad (n < 10)$
    - $c(n) = b(n) + c(n-10) \quad (n \geq 10)$

Exercise 8-2

change.hs

```
import System.Environment

main = do args <- getArgs
          print $ change 500 $ read $ head args

change 500 n = if n < 500 then change 100 n else ...
change 100 n = ...
change 50 n = ...
...
change 1 n = 1
```

- **`change c n`** is the number of ways to hand **`n`** yen with less than or equal to **`c`** yen coins.

```
% stack ghc change.hs
...
% ./change 30
16
% ./change 1000
248908
%
```

Recursive Call for Lists

- Any list consists of the empty list `[]` and `(:)`.
  - `[] :: [a]`
  - `(:) :: a -> [a] -> [a]`
  - `[1, 2, 3] = 1:(2:(3:[]))`
  - `1:[2]` → `[1, 2]`
  - `5:[]` → `[5]`

- For lists, `[]` and `(:)` pattern match are used:

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

recursive call for **`map`**

Recursive Calls (Examples)

- **`length`** (calculate the length of a list)
  - `length [] = 0`
  - `length (x:xs) = 1 + length xs`
- **`sum`** (calculate the sum of the elements in a list)
  - `sum [] = 0`
  - `sum (x:xs) = x + sum xs`
- **`(++)`** (concatenate the two lists)
  - `(++) :: [a] -> [a] -> [a]`
  - `(++) [] ys = ys`
  - `(++) (x:xs) ys = x : ((++) xs ys)`
- **`concat`** (concatenate all the lists in a list)
  - `concat :: [[a]] -> [a]`
  - `concat [] = []`
  - `concat (x:xs) = x ++ concat xs`

Exercise 8-3

reverse.hs

```
import System.Environment

main = do args <- getArgs
          print $ myreverse $ map read args

myreverse :: [Int] -> [Int]
myreverse []     = []
myreverse (x:xs) = ...
```

- Write your own **myreverse** which reverses a list.
  - The reverse of the empty list is the empty list.
  - If you want to reverse **(x:xs)** , what is necessary after reversing **xs** .

```
% stack ghc reverse.hs
...
% ./reverse 1 2 3 4 5
[5,4,3,2,1]
%
```

Exercise 8-4

sort.hs

```
import System.Environment

main = do args <- getArgs
          print $ mysort $ map read args

mysort :: [Int] -> [Int]
mysort []     = []
mysort (x:xs) = ...

myinsert :: Int -> [Int] -> [Int]
myinsert x []     = [x]
myinsert x (y:ys) = ...
```

- Write your own **`mysort`** which sorts an integer list from small to big.
  - The empty list is the empty list if you sort it.
  - If you want to sort **`(x:xs)`** , insert **`x`** after sorting **`xs`**.

```
% stack ghc sort.hs
...
% ./sort 5 3 7 2
[2,3,5,7]
%
```

Function Binding

```
square n = n * n
```

- Define **`square`** function which calculates the square of a given argument.

- Bind **`square`** to a function which calculates the square of a given argument.
  - just like **`a = 10`** binds a to the constant 10
  - **`square = ...`**

Higher Order Function

- Functions as values
  - can be used as arguments to functions
  - can be returned from functions

```
map square [1,2,3,4,5] ⇒ [1,4,9,16,25]
```

- **`map`** takes a function as an argument

- **`map`** returns a function
  - **`(map square)`** is a function which takes a list

Anonymous Function

```
\pat1 pat2 .... -> exp
```

- Create a function without giving a name
  - Function binding = create a function + bind it a variable
- Usage
  - Create function values
  - Create a function which can be used only once

```
square = \n -> n * n

map (\n -> n * n) [1, 2, 3, 4, 5]
```

Anonymous Function (cont.)

```
add x y = x + y

add = \x y -> x + y

(\x y -> x + y) 2 3 ⇒ (\y -> 2 + y) 3 ⇒ 2 + 3 ⇒ 5
```

- Can use pattern match.
  - Only one pattern is allowed

```
add2 (x, y) = x + y

add2 = \(x, y) -> x + y

map (\(x, y) -> x + y) [(1,11),(2,12),(3,13)]
      ⇒ [(1+11),(2+12),(3+13)]
      ⇒ [12,14,16]
```

Function Composition

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
Usage: f . g
```

- Compose two functions and create a new function
  - `(f . g) x = f (g x)`
  - `f . g = \x -> f (g x)`

```
numberOfLines :: String -> Int
numberOfLines cs = length $ lines cs
```

```
numberOfLines :: String -> Int
numberOfLines = length . lines
```

- Difference with `($)`
  - `($) :: (a -> b) -> a -> b`
  - `f $ x = f x`

Function Composition (cont.)

```
sortLines :: String -> String
sortLines cs = unlines $ sort $ lines cs
```

- Using function composition

```
sortLines = unlines . (sort . lines)
```

- `(.)` is right associative

```
sortLines = unlines . sort . lines
```

- Another example:

```
tac :: String -> String
tac cs = unlines $ reverse $ reverse $ reverse $ lines cs

tac = unlines . reverse . reverse . reverse . lines
```

Partial Application

- Arguments are not necessarily given at the same time.
  - **`addThree i j k = i + j + k`**
  - **`addThree 5`** is a partial application of **`addThree`** with the first argument

- Partial Application
  - Give some of the arguments, not all of them

```
addThree i j k = i + j + k

addThree 5 = \j k -> 5 + j + k

(addThree 5) 6 = \k -> 5 + 6 + k

((addThree 5) 6) 7 = 5 + 6 + 7
```

Section

- Partial application of binary operators

- Example:
  - `(+ 1)` is a partial application of `(+)` giving the second argument
    - `(+ 1) 2 ⇒ 2 + 1 ⇒ 3`
  - `(1 +)` is a partial application of `(+)` giving the first argument
    - `(1 +) 2 ⇒ 2 + 1 ⇒ 3`

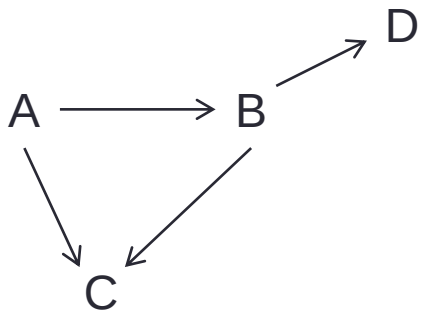
- Note:
  - `(-)` is both binary and unary operator.
    - `(- 1)` is just `-1`
    - use `(subtract 1)`

```
map (+ 7) [1,2,3,4,5]
        ⇒ [8,9,10,11,12]

filter (/= '\r') "aaa\r\nbbb\r\nccc\r\nddd\r\neee\r\n"
        ⇒ "aaa\nbbb\nccc\nddd\neee\n"
```

Point-Free Style

- Category theory
  - theory of objects and arrows
  - point = value



- Point-free style
  - not using values, but using function compositions only

fgrep.hs

```
import System.Environment
import Data.List

main = do args <- getArgs
          cs <- getContents
          putStr $ fgrep (head args) cs

fgrep :: String -> String -> String
fgrep pattern cs = unlines $ filter match $ lines cs
  where
    match :: String -> Bool
    match line = any prefixp $ tails line

    prefixp :: String -> Bool
    prefixp line = pattern `isPrefixOf` line
```

Convert to point-free style

```
fgrep :: String -> String -> String
fgrep pattern cs = unlines $ filter match $ lines cs
  where
    match :: String -> Bool
    match line = any prefixp $ tails line

    prefixp :: String -> Bool
    prefixp line = pattern `isPrefixOf` line
```

- without using **`where`** clause

```
fgrep :: String -> String -> String
fgrep pattern cs = unlines $ filter (match pattern) $ lines cs

match :: String -> String -> Bool
match pattern line = any (prefixp pattern) $ tails line

prefixp :: String -> String -> Bool
prefixp pattern line = pattern `isPrefixOf` line
```

Convert to point-free style (cont.)

```
fgrep :: String -> String -> String
fgrep pattern cs = unlines $ filter (match pattern) $ lines cs
```

```
fgrep pattern = unlines . filter (match pattern) . lines
```

```
prefixp :: String -> String -> Bool
prefixp pattern line = pattern `isPrefixOf` line
```

```
prefixp pattern = (pattern `isPrefixOf`)
```

```
match :: String -> String -> Bool
match pattern line = any (prefixp pattern) $ tails line
```

```
match pattern = any (prefixp pattern) . tails
```

```
match pattern = any (pattern `isPrefixOf`) . tails
```

Convert to point-free style (cont.)

fgrep.hs

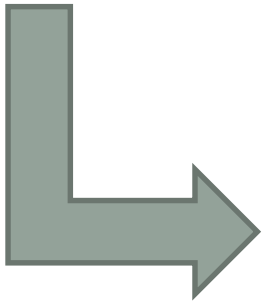
```
import System.Environment
import Data.List

main = do args <- getArgs
          cs <- getContents
          putStr $ fgrep (head args) cs

fgrep :: String -> String -> String
fgrep pattern cs = unlines $ filter match $ lines cs
  where
    match :: String -> Bool
    match line = any prefixp $ tails line

    prefixp :: String -> Bool
    prefixp line = pattern `isPrefixOf` line
```

Point-free style



fgrep.hs

```
import System.Environment
import Data.List

main = do args <- getArgs
          cs <- getContents
          putStr $ fgrep (head args) cs

fgrep :: String -> String -> String
fgrep pattern = unlines . filter (match pattern) . lines

match :: String -> String -> Bool
match pattern = any (pattern `isPrefixOf`) . tails
```

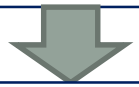
Convert further to point-free style

- In order not to refer the argument of **`match`**, the order of function application needs to be changed.

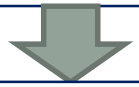
```
f . g
```

```
(. g) $ f
```

```
match pattern = any (pattern `isPrefixOf`) . tails
```



```
match pattern = any (isPrefixOf pattern) . tails
```



```
match pattern = ((any . isPrefixOf) pattern) . tails
```

```
match pattern = (. tails) $ (any . isPrefixOf) pattern
```

```
match = (. tails) . any . isPrefixOf
```

Exercise 8-5

- Make fgrep function point-free completely.

```
fgrep :: String -> String -> String
fgrep pattern cs = unlines $ filter (match pattern) $ lines cs
```

```
fgrep pattern = unlines . filter (match pattern) . lines
```


fgrep2.hs

```
import System.Environment
import Data.List

main = do args <- getArgs
          cs <- getContents
          putStr $ fgrep (head args) cs

fgrep :: String -> String -> String
fgrep = ...
```

Folding Functions

- List related functions are often defined as follows:

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

- In case of the empty list, the value is **v**.
- Otherwise, take out the head element **x**, apply itself to the rest of the list and combine the result by **op**.
- For example, the following functions are exactly in this style:

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

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

- By choosing **v** and **op**, various functions may be created.
- Create a higher order function which takes **v** and **op**, returns a function which does the above.

Right Associative Folding Function: foldr

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

- Using this folding function **`foldr`**, list related functions can easily be defined.
  - foldr takes two arguments: first one which creates the result and the second one for the empty value case.

```
sum = foldr (+) 0
prod = foldr (*) 1
```

- By choosing **`f`**, **`length`** can be defined by **`foldr`**.

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

```
length = foldr (\x n -> 1 + n) 0
```

Exercise 8-6

- Define **`(++)`** which connects two lists by **`foldr`**.

append.hs

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

append xs ys = foldr ....
```

- Define **`map`** function using **`foldr`**.

map2.hs

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

map2 f = foldr ....
```

- Define **`reverse`** which reverses a list by **`foldr`**.

rev.hs

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

rev = foldr ....
```

Left Associative Folding Function: foldl

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

- **`foldr`** is for right associative operators. There is a similar function for left associative operators: **`foldl`**.
- Creates values from left to right.
- Since **`(+)`** and **`(*)`** are left associative operators, it is natural to use **`foldl`**.

```
sum = foldl (+) 0
prod = foldl (*) 1
```

- **`reverse`** can be implemented much lighter.

```
reverse::[a] -> [a]
reverse xs = reverse2 [] xs

reverse2::[a] -> [a] -> [a]
reverse2 ys [] = ys
reverse2 ys (x:xs) = reverse2 (x:ys) xs
```

```
reverse = foldl (\xs x -> x:xs) []
```

Exercise 8-7

- Convert a binary digit number to a decimal number.
  - An argument is given as a string. Convert it to a list of numbers by **`digit`**.
  - Use **`foldl`** to combine the result.
  - Make **`digit`** and **`foldl`** both point-free.

b2d.hs

```
import System.Environment
import Data.Char

main = do args <- getArgs
          print $ b2d $ head args

digit::Char -> Int
digit ch = ord ch - ord '0'

b2d::String -> Int
b2d = foldl ...
```

```
% ./b2d 1010
10
% ./b2d 11111100100
2020
%
```

Exercise 8-8

- Roman numerals use I, V, X, L, C, D and M symbols to represent numbers.

Symbols	I	V	X	L	C	D	M
Value	1	5	10	50	100	500	1000

- Convert a roman numeral to a number by simply adding numbers which each symbol represents.

r2a.hs

```
import System.Environment

main = do args <- getArgs
          print $ r2a $ head args

r2a::String -> Int
r2a = ...
```

- Can you add rules like IV is 4, IX is 9, XL is 40, XC is 90, and so on?

```
% ./r2a MMXX
2020
% ./r2a CMXLIX
949
%
```

Exercise 8-9

- Convert an Arabic number (from 1 to 3999) to Roman.

example

```
% ./roman 1111
MCXI
% ./roman 1954
MCMLIV
% ./roman 1990
MCMXC
% ./roman 2020
MMXX
```

a2r.hs

```
import System.Environment

main = do args <- getArgs
          putStrLn $ roman $ read $ head args

roman::Int -> String
roman ....
```

- Roman number consists of the following 7 letters:

Letter	I	V	X	L	C	D	M
Number	1	5	10	50	100	500	1000

- The numbers of each letter are added.
- Starting from larger numbers.
- To avoid the repetition of 4 letters of the same one, the following subtraction rules are used:

String	IV	IX	XL	XC	CD	CM
Number	4	9	40	90	400	900