FUNCTIONAL PROGRAMMING
NO.4 MODULES

Tatsuya Hagino
hagino@sfc.keio.ac.jp

lecture slide URL

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

Modules

- Haskell programs are divided into modules.

- Each module consists of:
  - functions
  - variables
  - types

- Importing modules
  - Before using modules, you need to import them.

```
import Module
```

echo command

echo.hs

```
import System.Environment

main = do args <- getArgs
          putStr $ unwords args
```

- This command outputs its command arguments to the standard output.

```
% stack ghc echo.hs
...
% ./echo a b c
a b c
% ./echo This    is    a   pen.
This is a pen.
% ./echo "This    is    a    pen."
This    is    a    pen.
%
```

Main and Prelude Modules

- Main module
  - main variable belongs to Main module.
  - If you do not specify a module, everything you write belongs to Main module.

- Prelude module
  - Defines basic types, functions and variables.
  - It is implicitly imported by the system.
  - getContents, lines, unlines, ... are defined in Prelude module.

unwords function

```
unwords:: [String] -> String
```

- Concatenates strings in the list by adding spaces in between.
  - `unwords ["a", "b", "c"]` → `"a b c"`
  - `unwords ["a(1,", "2,", "3)"]` → `"a(1, 2, 3)"`
  - `unwords ["This", "is", "a", "pen."]` → `"This is a pen."`
  - `unwords ["a\n", "b"]` → `"a\n b"`
  - `unwords []` → `""`

System.Environment actions

```
getArgs:: IO [String]
```

- An action to read given command arguments.
- If the action is successfully executed, the result is a string list.

```
getProgName:: IO String
```

- An action to read the program name.
- If the action is successfully executed, the result is a string.

Exercise 4-1

- Write **`sum.hs`** which adds numbers given as command arguments.

sum.hs

```
import System.Environment

main = do args <- getArgs
          print $ .... args
```

- In order to convert a string to a number, use **`read`** function.
  - **`(read "123")::Int`**
  - **`read`** return type is polymorphic, so that a type needs to be specified.

```
% stack ghc sum.hs
...
% ./sum 1 5 3
9
% ./sum 123 248
371
%
```

fgrep command

- fgrep outputs the lines which contain a given string.
  - Outputs lines in USA-states.txt which contains ar.

```
% ./fgrep ar < USA-states.txt
CT      Connecticut     Hartford
DE      Delaware        Dover
MD      Maryland        Annapolis
NC      North Carolina  Raleigh
ND      North Dakota    Bismarck
NV      Nevada  Carson City
PA      Pennsylvania    Harrisburg
SC      South Carolina  Columbia
WV      West Virginia   Charleston
%
```

fgrep command

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
```

main action and fgrep function

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

fgrep :: String -> String -> String
fgrep pattern cs = unlines $ filter match $ lines cs
  where
    ...
```

- **`main`** action
  - **`getArgs`** reads the command argument and binds it **`args`**
  - Reads the standard input and binds to **`cs`**
  - **`head`** takes the first element of **`args`**
  - The first element and **`cs`** are passed to **`fgrep`**
  - Outputs the result
- **`fgrep`** function
  - **`lines`** divides the whole file into lines.
  - **`unlines`** put them back.
  - **`filter`** is a higher-order function to select elements from a list which satisfies a condition.

head, tail and filter functions

- **`head`** function
  - **`head :: [a] -> a`**
  - Returns the first element of the list
  - **`head [1, 2, 3]`** → **`1`**
  - **`head [2, 3]`** → **`2`**
  - **`head [3]`** → **`3`**
  - **`head []`** → run time error

- **`tail`** function
  - **`tail :: [a] -> [a]`**
  - Removes the first element of the list
  - **`tail [1, 2, 3]`** → **`[2, 3]`**
  - **`tail [2, 3]`** → **`[3]`**
  - **`tail [3]`** → **`[]`**
  - **`tail []`** → run time error

- **`filter`** function
  - **`filter :: (a -> Bool) -> [a] -> [a]`**
  - **`filter f xs`** collects element **`x`** in **`xs`** which make **`f x True`**
  - **`filter odd [1, 2, 3, 4, 5]`** → **`[1, 3, 5]`**
  - **`filter odd [2, 4, 6, 8, 10]`** → **`[]`**
  - **`filter odd []`** → **`[]`**
  - where **`odd n`** return **`True`** if **`n`** is an odd number

where clause

```
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
```

- **`where`** clause allows to define functions which can be used in the expression (defining local functions)

expression **`where`** $definition_1$
$definition_2$
$definition_3$

  - *expression* can use functions define in $definition_1$, $definition_2$ and $definition_3$

- **`match`** and **`prefixp`** functions are defined.
  - **`match`** and **`prefixp`** can only be used inside **`fgrep`**.
  - **`match`** and **`prefixp`** can use variables of **`fgrep`**.

match function

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

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

- **`match line`**
  - Checks whether **`line`** (e.g.**`"abcd"`**) contains **`pattern`** (e.g.**`"bc"`**) or not.
    - **`match "abcd"`**
  - **`tails`** creates a list of strings by shortening **`line`** one character by one character.
    - **`tails "abcd"` → `["abcd", "bcd", "cd", "d", ""]`**
  - **`any`** checks whether there is a string which satisfies **`prefixp`**
  - **`prefixp`** checks whether it starts with **`pattern`**
    - **`prefixp "abcd"` → `"bc" `isPrefixOf` "abcd"` → `False`**
    - **`prefixp "bcd"` → `"bc" `isPrefixOf` "bcd"` → `True`**
    - **`prefixp "cd"` → `"bc" `isPrefixOf` "cd"` → `False`**

any and all functions

- **any** function
  - **`any :: (a -> Bool) -> [a] -> Bool`**
  - **`any f xs`** applies **`f`** to each element of **`xs`**, and if one of it is **`True`**, then it returns **`True`**. If all the elements return **`False`**, it also returns **`False`**.
  - **`any odd [1, 2, 3, 4, 5]`** → **`True`**
  - **`any odd [1, 3, 5]`** → **`True`**
  - **`any odd [2, 4, 6]`** → **`False`**
  - **`any odd [3]`** → **`True`**
  - **`any odd []`** → **`False`**

- **`all`** function
  - **`all :: (a -> Bool) -> [a] -> Bool`**
  - **`all f xs`** applies **`f`** to each element of **`xs`**, and if all of them are **`True`**, then it returns **`True`**. If an any element elements return **`False`**, it also returns **`False`**.
  - **`all odd [1, 2, 3, 4, 5]`** → **`False`**
  - **`all odd [1, 3, 5]`** → **`True`**
  - **`all odd [2, 4, 6]`** → **`False`**
  - **`all odd [3]`** → **`True`**
  - **`all odd []`** → **`True`**

tails and isPrefixOf functions

- **`Data.List.tails`** function
  - **`tails :: [a] -> [[a]]`**
  - **`tails xs`** returns a list consits of **`xs`** itself, **`xs`** taking out the first element, **`xs`** taking out the first two elements, ...
  - **`tails [1, 2, 3]`** → **`[[1, 2, 3], [2, 3], [3], []]`**
  - **`tails [1, 2]`** → **`[[1, 2], [2], []]`**

- **`Data.List.isPrefixOf`** function
  - **`isPrefixOf :: (Eq a) => [a] -> [a] -> Bool`**
  - **`isPrefixOf xs ys`** is **`True`** if **`xs`** matches the first part of **`ys`**
  - **`xs `isPrefixOf` ys`** is same as **`isPrefixOf xs ys`**
    - makes a function a binary operator

Exercise 4-2

fgrepi.hs

```
import System.Environment
import Data.List

main = ...

fgrepi :: String -> String -> String
fgrepi pattern cs = unlines $ filter match $ lines cs
  where
    ...
```

- fgrep.hs distinguishes upper and lower letters. Write fgrepi.hs which does not distinguish.
  - For example, Arasuka can be selected when aR is given.
  - In order to convert upper letters to lower ones, use function lower which is defined in Exercise 3-2.

```
lower :: Char -> Char
lower 'A' = 'a'
lower 'B' = 'b'
...
lower 'Z' = 'z'
lower c = c
```

```
% ./fgrepi aR < USA-states.txt
AR      Arkansas        Little Rock
AZ      Arizona Phoenuix
CT      Connecticut     Hartford
DE      Delaware        Dover
...
%
```

Exercise 4-3

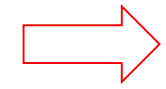
uniq.hs

```
main = do cs <- getContents
          putStr $ unlines $ uniq $ lines cs

uniq :: [String] -> [String]
uniq [] = []
uniq ...
```

- Write program uniq.hs which removes duplicate lines (i.e. the same line as the previous line).
  - Only use functions which we studied in the class.

```
% stack ghc uniq.hs
...
% ./uniq
ABCD
ABCD
EFG
XXX
ABCD
XXX
XXX
^D
```



output

```
ABCD
EFG
XXX
ABCD
XXX
```

Exercise 4-4

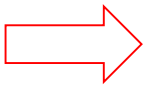
uniq2.hs

```
main = do cs <- getContents
          putStr $ unlines $ uniq $ lines cs

uniq :: [String] -> [String]
uniq [] = []
uniq ...
```

- Write program uniq2.hs which removes duplicate lines which have already appeared before.

```
% stack ghc uniq2.hs
...
% ./uniq2
ABCD
ABCD
EFG
XXX
ABCD
XXX
XXX
^D
```



output

```
ABCD
EFG
XXX
```

Summary of Functions and Actions

Function	Example	Meaning
`unwords`	`unwords xs`	Concatenates the strings by adding speaces
`sum`	`sum xs`	Add numbers in the list
`read`	`(read s)::Int`	Convert string to an integer
`any`	`any f xs`	Checks whether there is an element `x` which makes `f x True`
`all`	`all f xs`	Checks whether all the elements `x` makes `f x True`
`filter`	`filter f xs`	Selects elements from `xs` which makes `f x True`
`head`	`head xs`	Returns the head element of `xs`
`tail`	`tail xs`	Removes the head element of `xs`
`Data.List.tails`	`tails xs`	`[xs, (tail xs), (tail(tail xs)), ...]`
`Data.List.isPrefixOf`	`xs `isPrefixOf` ys`	Checkes whether `xs` is the head part of `ys`

Action	Meaning
`System.Environment.getArgs`	Reads command line arguments and the result is a string list
`System.Environment.getProgName`	Reads the command name and the result is a string