関数型プログラミング
第7回　UNIXコマンドを作る(2)

萩野　達也
hagino@sfc.keio.ac.jp

Slide URL

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

モジュール

- Haskellプログラムはモジュール単位に分割されている

- モジュールでは以下のものが定義されている:
  - 関数
  - 変数
  - データ型

- モジュールの読み込み
  - モジュールは，利用する前に読み込む（インポート）する必要がある

```
import Module
```

Main と Prelude モジュール

- **Main** モジュール
  - **main** 変数は **Main** モジュールに属する
  - モジュールを指定しない場合，すべてのものは **Main** モジュールに属する

- **Prelude** モジュール
  - 基本的な型，変数，関数を定義している．
  - 暗黙でシステムに読み込まれる
  - **getContents**, **lines**, **unlines**, ... は **Prelude** モジュールで定義されている

module 宣言

```
module name where {
   ...
}
```

- ***name*** のモジュールを宣言する.
  - モジュール名は大文字で始めること
  - 複数の名前を「.」でつなぐこともできる.
  - モジュールで定義されたエンティティ(変数, 関数, 型, クラスなど)が外部にエキスポートされる.

例

```
module FileUtils where {
   data SomeType = ConsA String | ConsB Int;
   makePath = ....;
   forceRemove = ...
}
```

Mainモジュール

- モジュール宣言で始まらないファイルは，次の **Main** モジュールの宣言が最初になされたものとみなされる．

```
module Main(main) where
```

- **Main** モジュール
  - **main** のみがエキスポートされる．
  - **main** の型は **(IO a)** でなくてはいけない．
  - **main** はプログラムのエントリーポイントとなる．

echoコマンド

echoo.hs

```
import System.Environment

main = getArgs >>= putStrLn . unwords
```

- コマンドの与えられた引数をそのまま標準出力に出す.

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

unwords 関数

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

- 文字列を空白は挟んでつなげる
  - `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 []` → `""`

練習問題7－1

- 文字列を空白は挟んでつなげる **unwords** を自分で書きなさい.
  - 関数名は **uunwords** にします.

uunwords.hs

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

- uunwords ss を ss についての分割統治で解きましょう.
  - ss が空リストのとき
  - ss が単一の文字列だけのとき
  - ss が2つ以上の文字列を含むとき

System.Environment アクション

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

- コマンド引数を読み込むアクション
- アクションが成功すると，文字列のリストが返される

```
getProgName:: IO String
```

- プログラム名を読み込むアクション
- アクションが成功すると，文字列が返される

練習問題7－2

- 引数に数字が与えられたときに，その合計を出力するプログラム **sum.hs** を完成させなさい．

sum.hs

```
import System.Environment

main = getArgs >>= \args -> print $ sumArgs args
  where sumArgs args = ...
```

- 文字列を数字にするには **read** を用います．
  - `read "123"::Int`
  - **read**は返す値が多層型なので欲しい型を指定する必要がある場合があります
- リストの数字を合計するには **sum** を用います．
  - `sum::[Int]->Int`
  - `sum [1,2,3,4,5]` ⇒ `19`

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

fgrep コマンド

- 特定の文字列を含む行だけを入力から抜き出し出力する.
  - airline-code.txtの中からJapanを含む行を抜き出す.

```
% fgrep Japan < airline-code.txt
NV      Air Central     AIR CENTRAL     Japan
NQ      Air Japan       AIR JAPAN       Japan
EL      Air Nippon      ANK AIR Japan
EH      Air Nippon Network Co. Ltd.     ALFA WING       Japan
DJ      AirAsia Japan   WING ASIA       Japan
HD      AIRDO   AIR DO  Japan
NH      All Nippon Airways      ALL NIPPON      Japan
...
7G      Star Flyer      STARFLYER       Japan
JW      Vanilla Air     VANILLA Japan
```

fgrep コマンドのプログラム

fgreep.hs

```
import System.Environment
import Data.List

main = getArgs >>= \args ->
       getContents >>= \cs ->
       putStr $ fgrep (head args) cs

fgrep :: String -> String -> String
fgrep pattern cs = unlines $ filter match $ lines cs
  where {
    match line = any prefixp $ tails line;
    prefixp line = pattern `isPrefixOf` line
  }
```

- 上記のプログラムを入力しなさい.
  - airline-code.txtからJapanを含む行を抜き出しなさい
  - airを含む行はどうですか

main アクションと fgrep 関数

```
main = getArgs >>= \args ->
       getContents >>= \cs ->
       putStr $ fgrep (head args) cs

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

- mainアクション
  - getArgsでコマンド引数を読みargsに束縛
  - 標準入力を読みcsに束縛
  - headでargsの先頭を取り出しcsとともにfgrepに渡す
  - その結果を出力する
- fgrep関数
  - linesで行うごとに分け，unlinesで行に戻す
  - filterは条件を満たすリストを取り出す高階関数

any 関数，tails 関数，isPrefixOf 関数

- any 関数
  - `any::(a -> Bool) -> [a] -> Bool`
  - `any p xs` は `xs` の各要素に `p` を適用し，そのいずれかの値が `True` なら `True` を返す．すべてが `False` ならば `False` を返す．
  - `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`
- Data.List.tails 関数
  - `tails::[a] -> [[a]]`
  - `tails xs` はリスト `xs` 自身，`xs` から2番目の要素以降のリスト，3番目の要素以降のリスト，‥‥をリストにして返す
  - `tails [1, 2, 3]` → `[[1, 2, 3], [2, 3], [3], []]`
  - `tails [1, 2]` → `[[1, 2], [2], []]`
- Data.List.isPrefixOf 関数
  - `isPrefixOf::(Eq a) => [a] -> [a] -> Bool`
  - `isPrefixOf xs ys` はリスト `xs` がリスト `ys` の先頭に一致するときに `True`
  - `xs` `` `isPrefixOf` `` `ys` は `isPrefixOf xs ys` と同じで2項演算子として使う方法

match 関数の実装

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

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

- **match line**
  - **line**(たとえば"**abcd**")の中に **pattern**（たとえば"**bc**")が含まれているかを調べる
    - **match "abcd"**
  - **tails** によって **line** を一文字ずつ短くした文字列を作る
    - **tails "abcd" → ["abcd", "bcd", "cd", "d", ""]**
  - **any**によって**prefixp**を満たすものがあるのかを調べる
  - **prefixp** は **pattern** で始まっているかを調べる
    - **prefixp "abcd" → "bc" `isPrefixOf` "abcd" → False**
    - **prefixp "bcd" → "bc" `isPrefixOf` "bcd" → True**
    - **prefixp "cd" → "bc" `isPrefixOf` "cd" → False**

練習問題7－3

- **any** 関数，**tails** 関数，**isPrefixOf** 関数を自分で定義してみなさい．
- `any::(a -> Bool) -> [a] -> Bool`
  - `any p xs` は `xs` の各要素に `p` を適用し，そのいずれかの値が `True` なら `True` を返す．すべてが `False` ならば `False` を返す．

```
any p [] = ...
any p (x:xs) = ...
```

- `tails::[a] -> [[a]]`
  - `tails xs` はリスト `xs` 自身，`xs` から2番目の要素以降のリスト，3番目の要素以降のリスト，‥‥をリストにして返す

```
tails [] = ...
tails (x:xs) = ...
```

- `isPrefixOf::(Eq a) => [a] -> [a] -> Bool`
  - `isPrefixOf xs ys` はリスト `xs` がリスト `ys` の先頭に一致するときに `True`

```
isPrefixOf [] ys = ...
isPrefixOf xs [] = ...
isPrefixOf (x:xs) (y:ys) = ...
```

練習問題7－4

fgrepi.hs

```
import System.Environment
import Data.List
import Data.Char

main = ...

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

- fgrep.hs では大文字と小文字を区別していましたが，区別しない形でマッチする行を探して出力する fgrepi.hs を作りなさい.
  - たとえば airline-code.txt の中で japan を含む行を抜き出せますか.
  - 大文字を小文字に変換するためには **Data.Char** モジュールの `toLower` を使いなさい.

```
% stack runghc fgrepi.hs japan < airline-code.txt
NV      Air Central     AIR CENTRAL     Japan
NQ      Air Japan       AIR JAPAN       Japan
EL      Air Nippon      ANK AIR Japan

...
%
```

all 関数

- all 関数
  - all:: (a -> Bool) -> [a] -> Bool
  - all p xs は xs の各要素に p を適用し，そのすべての値が True なら True を返す．いずれかが False ならば False を返す．
  - 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

- any 関数と似ている
  - any:: (a -> Bool) -> [a] -> Bool
  - any p xs は xs の各要素に p を適用し，そのいずれかの値が True ならTrueを返す．すべてが False ならば False を返す．
  - 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 p xs = not $ any (not . p) xs

any p xs = not $ all (not . p) xs
```

練習問題7－5

fgrepm.hs

```
import System.Environment
import Data.List

main = getArgs >>= \args ->
       getContents >>= \cs ->
       putStr $ fgrepm args cs

fgrepm :: [String] -> String -> String
fgrepm ps cs = unlines $ filter matchAll $ lines cs
  where {
    matchAll line = all match ps
      where {
        ...
```

- fgreep.hsでは与えられた引数1つにマッチする行を返しましたが，複数の引数すべてを含む行だけを出力するfgrepm.hsを作りなさい．

```
% stack runghc fgrepm.hs Japan All < airline-code.txt
NH      All Nippon Airways      ALL NIPPON      Japan
%
```

関数およびアクションのまとめ

関数名	適用例	意味
`unwords`	`unwords xs`	文字列のリスト `xs` を空白を挟んで連結する
`sum`	`sum xs`	リスト `xs` の要素を合計する
`read`	`(read s)::Int`	文字列を数字に変換する
`any`	`any p xs`	リスト `xs` の要素 `x` のうち `p x` が `True` になるものがあれば `True` を返す
`all`	`all p xs`	リスト `xs` のすべての要素 `x` に対して `p x` が `True` になれば `True` を返す
`Data.List.tails`	`tails xs`	`[xs, (tail xs), (tail(tail xs)), ...]` を返す
`Data.List.isPrefixOf`	`xs `isPrefixOf` ys`	リスト `xs` がリスト `ys` の先頭部分に一致するときに `True`

アクション名	意味
`System.Environment.getArgs`	コマンドライン引数を読み込むアクション
`System.Environment.getProgName`	コマンド名のアクション