

関数型プログラミング

第4回 モジュール

萩野 達也

hagino@sfc.keio.ac.jp

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
```

echoコマンド

```
echo.hs
```

```
import System.Environment

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

- コマンドの与えられた引数をそのまま標準出力に出す。

```
% 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 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

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 []` → `""`

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4-1

```
sum.hs
```

```
import System.Environment

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

- 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。
- 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。

```
% ghc sum.hs
...
% ./sum 1 5 3
9
% ./echo 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。
 - USA-states.txtの中からarを含む行を抜き出す。

```
% ./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 コマンドのプログラム

```
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
```

- 上記のプログラムを入力しなさい。
 - USA-states.txtからarを含む行を抜き出さなさい
 - erを含む行はどうですか

main アクションと fgrep 関数

```
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**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

head と tail と filter 関数

- head関数

- `head :: [a] -> a`
- リストの先頭の要素を返す
- `head [1, 2, 3] → 1`
- `head [2, 3] → 2`
- `head [3] → 3`
- `head [] → 実行時エラー`

- tail関数

- `tail :: [a] -> [a]`
- リストから先頭要素を取り除いたリストを返す
- `tail [1, 2, 3] → [2, 3]`
- `tail [2, 3] → [3]`
- `tail [3] → []`
- `tail [] → 実行時エラー`

- filter関数

- `filter :: (a -> Bool) -> [a] -> [a]`
- `filter f xs`はリスト`xs`の要素`x`のうち`f x`が`True`である要素だけを集めたリストを返す
- `filter odd [1, 2, 3, 4, 5] → [1, 3, 5]`
- `filter odd [2, 4, 6, 8, 10] → []`
- `filter odd [] → []`
- ただし`odd n`は`n`が奇数の時に`True`

where 節

```
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節は式の中で必要な関数などを後で定義する構文

```
式 where 定義1
         定義2
         定義3
```

- 式の中で定義1～定義3で定義した関数などを利用することができる。
- fgrep関数の中で使うmatch関数とprefixp関数を定義している
 - matchやprefixpはfgrep外のmainなどでは利用できない
 - matchやprefixpではfgrepの変数なども参照可能

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

any関数, tails関数, isPrefixOf関数

• any関数

- `any :: (a -> Bool) -> [a] -> Bool`
- `any f xs`はリスト`xs`の各要素に`f`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

練習問題4-2

```
fgrep.hs
```

```
import System.Environment
import Data.List

main = ...

fgrep :: String -> String -> String
fgrep pattern cs = unlines $ myfilter 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fgrep.hsを作りなさい。
 - たとえばUSA-states.txtの中でarを含む行を抜き出したときにAraskaも含まれていますか？
 - 大文字を小文字に変換するためには練習問題3-2のlowerを用いなさい。

```
lower :: Char -> Char
lower 'A' = 'a'
...
lower c = ...
```

練習問題4-3

```
uniq.hs
```

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

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

- 同じ行が連続していたらそれらを1行にまとめるuniqコマンドを完成させなさい。
 - これまでに授業で出てきた関数のみを使用すること。

```
% ghc uniq.hs
```

```
...
```

```
% ./uniq
```

```
ABCD
```

```
ABCD
```

```
EFG
```

```
XXX
```

```
ABCD
```

```
XXX
```

```
XXX
```

```
^D
```



出力

```
ABCD
```

```
EFG
```

```
XXX
```

```
ABCD
```

```
XXX
```


練習問題4-4

```
uniq2.hs
```

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

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

- 連続していなくても、ファイル内に重複する行がある場合には、最初の1行だけを出力するuniq2コマンドを完成させなさい。

```
% ghc uniq2.hs
```

```
...
```

```
% ./uniq2
```

```
ABCD
```

```
ABCD
```

```
EFG
```

```
XXX
```

```
ABCD
```

```
XXX
```

```
XXX
```

```
^D
```



出力

```
ABCD
```

```
EFG
```

```
XXX
```

関数およびアクションのまとめ

関数名	適用例	意味
<code>unwords</code>	<code>unwords xs</code>	文字列のリスト <code>xs</code> を空白を挟んで連結する
<code>any</code>	<code>any f xs</code>	リスト <code>xs</code> の要素 <code>x</code> のうち <code>f x</code> が <code>True</code> になるものがあれば <code>True</code> を返す
<code>filter</code>	<code>filter f xs</code>	リスト <code>xs</code> の要素 <code>x</code> のうち <code>f x</code> が <code>True</code> になるものを集めたリストを返す
<code>head</code>	<code>head xs</code>	リスト <code>xs</code> の先頭要素を返す
<code>tail</code>	<code>tail xs</code>	リスト <code>xs</code> の第2要素以降のリストを返す
<code>Data.List.tails</code>	<code>tails xs</code>	<code>[xs, (tail xs), (tail (tail xs)), ...]</code> を返す
<code>Data.List.isPrefixOf</code>	<code>xs `isPrefixOf` ys</code>	リスト <code>xs</code> がリスト <code>ys</code> の先頭部分に一致するときに <code>True</code>

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