関数型プログラミング
第6回　UNIXコマンドを作る（1）

萩野　達也
hagino@sfc.keio.ac.jp

Slide URL

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

プログラム開発環境

- プログラム開発環境CUI vs GUI
  - CUI（Character User Interface）またはCLI（Command Line Interface）
    - 単純で軽い
    - コンパイラとライブラリを使ってプログラム開発
    - テキストエディタを使ってプログラムを書く
  - GUI（Graphical User Interface）
    - 現代的だが重い
    - エディタ，コンパイラ，デバッガなどが一体となっている
    - 例：eclipse, Xcode, Visual Studio

- CUI
  - UNIX (Linux): シェル（sh, csh, tcsh, bash）
  - Mac OS X: ターミナル
  - Windows: コマンドプロンプト

- テキストエディタ
  - UNIX (Linux): vi (vim), emacs
  - Mac OS X: TextEdit, mi, emacs
  - Windows: notepad, xyzzy
  - 非依存：atom

UNIXの基本コマンド

- CUIの基本
  - 実行するコマンドを入力する
  - コマンド名と引数を与える
  - current working directoryを正しく設定すること
  - folder = directory

% *command arg1 arg2 arg3*

プロンプト

引数

- シェルの基本コマンド

コマンド	意味
pwd	current working directoryを表示（print working directory）
cd *dir*	ディレクトリを*dir*に変更（change directory）
ls *dir*	*dir*にあるファイルの一覧を表示（list）
ls -l *dir*	*dir*にあるファイルの一覧を詳しく表示（long list）
cat *file*	*file*の中身を表示（concatenate）
more *file*	*file*の中身を1ページずつ表示
mkdir *dir*	新しいディレクトリ*dir*を作成（make directory）
rmdir *dir*	ディレクトリ*dir*を削除する（remove directory）
rm *file*	*file*を削除する（remove）
command < *file*	*command*の入力を*file*からにする（入力リダイレクション）
command > *file*	*command*の出力を*file*にする（出力リダイレクション）

削除したものを戻すことはできない
ゴミ箱に移すのとは異なる

ファイルの中身を表示する

- UNIXのcatコマンドに似たものをHaskellで書いてみましょう.
  - あたえられたファイルの中身を表示する.

caat.hs

```
main = getContents >>= putStr
```

```
% stack ghc caat.hs
...
% ./caat < caat.hs
main = getContents >>= putStr
%
```

- "./" は現在のディレクトリを表す.
- "./caat" は現在のディレクトリの "caat" プログラムを意味する.
- Windowsでは "./caat" のかわりに ".¥caat.exe" としてください.
- "< caat.hs" は入力を端末からでなく, ファイルに切り替えるシェルによるリダイレクション.

cat プログラム

```
main = getContents >>= putStr
```

- **getContents**
  - 端末からの入力を文字列とするアクション
- **putStr**
  - 文字列を端末に出力するアクション
  - **putStr** は文字列を受け取る関数
  - **putStrLn** は最後に改行したが，**putStr** はしない
- **getContents >>= putStr**
  - **getContents** のアクションが成功した場合，その値（文字列）を **putStr** に渡す

```
main = putStr(getContents)
```

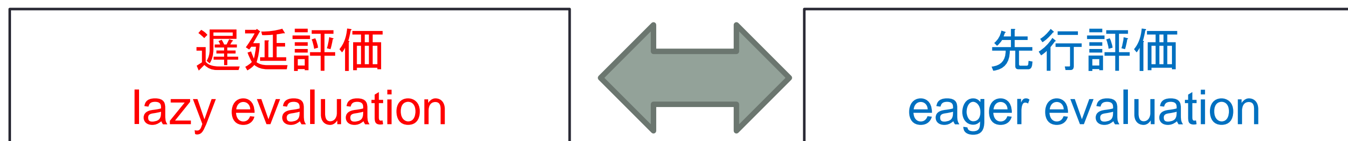
- **getContents** の値を直接 **putStr** に与える
- うまく動かない
- 関数型では一般に評価順序が決まっていないので，**getContents** が先に評価されるわけではない
- >>= を使って評価順序を与える必要がある（モナド）

遅延評価（lazy evaluation）

caat.hs

```
main = getContents >>= putStr
```

- **getContents** は端末からの入力のすべてを一度に読み込むわけではない．
  - 端末からの入力を受け取り終わってから，それを文字列にして **putStr** に渡すわけではない．
  - 端末からの入力から作られる文字列になるであろうものを **putStr** に渡す．
  - 実際の文字列の中身は **putStr** が出力しようとしてから読み込まれる．



必要になった時にはじめて評価する
なるべく評価を遅らせる

先に評価してしまう
積極的に評価する

- 端末から入力があるごとに出力される
  - 端末からの入力はOSが1行ごとにまとめてアプリケーションに送っている
  - 行ごとに読み込むようなループを書いたわけではない

ファイルの行数を数える

- UNIXではwcコマンドを使って，ファイルの行数を数えることができる.

```
% wc countline.hs
  5  21  111  countline.hs
```

- ファイルの行数を数えるプログラムをHaskellで書いてみよう.
  - 行の終わりには改行文字(`'\n'`)があるので，その数を数えればよい.

countline.hs

```
main = getContents >>= print . countLine

countLine []          = ...
countLine ('\n':cs) = ...
countLine (_:cs)      = ...
```

- 上記のプログラムを実行

```
% stack runghc countline.hs < countline.hs
5
```

ファイルの行数を数える(2)

- 高階関数 **`filter`** を用いるとどうなる?
- リストの中から条件に合うものだけを選ぶ.
  - `filter::(a->Bool)->[a]->[a]`
  - `filter p xs`
    - `p` は真偽値を返す関数
    - `p` が `True` となる `xs` の要素だけを選ぶ.
- ファイル(文字列)から改行文字(`'\n'`)だけを取り出して数えれば良いのでは?

countline2.hs

```
main = getContents >>= print . countLine

countLine cs = length(filter eqln cs)
  where eqln ch = ...
```

$ 演算子

```
countLine cs = length(filter eqln cs)
```

- '$' 演算子の利用
  - 'f $ x' の意味は '(f x)'

```
countLine cs = length $ filter eqln cs
```

  - 右結合の演算子なので括弧を省略することが可能
    - 'f $ g $ x' は 'f $ (g $ x)' を意味する

```
head(tail(tail(tail xs)))
```

```
head $ tail $ tail $ tail xs
```

ファイルの行数を数える(3)

- '$' 演算子を使うことで少しすっきりした.

countline3.hs

```
main = getContents >>= print . countLine

countLine cs = length $ filter eqln cs
  where eqln ch = ch == '\n'
```

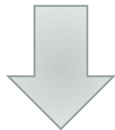
- でも, まだ, 疑問は残る
  - 「`print . countLine`」の「.」は何？
  - 「`filter eqln`」で filter の述語は常に `where` で書かないといけないの?

無名関数

```
\パターン1 パターン2 ‥‥ -> 式
```

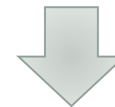
- 関数名を与えずに関数を作ることができる.
  - 関数定義＝関数作成＋変数束縛
- 使用例
  - 関数の値を作成する.
  - 一度しか使わない関数に名前を与える必要はない.

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



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

```
let square n = n * n
in map square [1, 2, 3, 4, 5]
```



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

無名関数(つづき)

- 複数引数の無名関数

```
add x y = x + y

add = \x y -> x + y

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

- パターンマッチを利用することも可能
  - ただし一つのパターンしか書くことができない
  - ガードも使うことができない

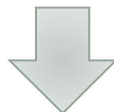
```
add2 (x, y) = x + y

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]
```

let も無名関数？

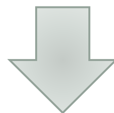
```
let x = 2
in x * x
```

```
(\x -> x * x) 2
```

```
let square n = n * n
in map square [1, 2, 3, 4, 5]
```



```
let square = \n -> n * n
in map square [1, 2, 3, 4, 5]
```



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

ファイルの行数を数える(4)

- 無名関数を使って **where** を消す

countline4.hs

```
main = getContents >>= print . countLine

countLine cs = length $ filter (\ch -> ch == '\n') cs
```

- でも，まだ，疑問は残る
  - 「**print . countLine**」の「**.**」は何？
  - 「**\ch -> ch == '\n'**」は良いけど，もっと何とかならないの？

部分適用

- 関数に引数は一度に渡す必要はない
  - `addThree i j k = i + j + k`
  - 「`addThree 5`」は`addThree`に最初の引数を与えた部分適用状態
    - 残り2つの引数が与えられるのを待っている
- 部分適用
  - 関数に一部の引数を与えた状態のこと

```
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
```

セクション

- 二項演算子の部分適用
- 例:
  - 「`(+ 1)`」は「+」の2つ目の引数を部分適用したもの
  - 「`(1 +)`」は「+」の1つ目の引数を部分適用したもの
  - `(+ 1) 2` ⇒ `3`
- 注意:
  - **`(-)` 二項演算子でもあり単項演算子でもある**
    - 「`(- 1)`」は単に「`-1`」を意味する
    - 「`(subtract 1)`」を使うこと

```
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"
```

ファイルの行数を数える(5)

- セクションを使う

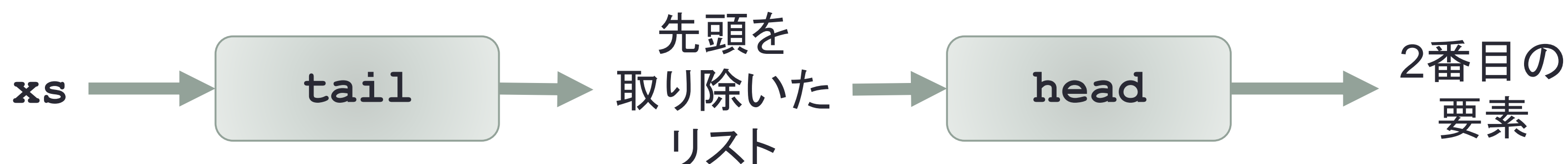
countline5.hs

```
main = getContents >>= print . countLine

countLine cs = length $ filter (== '\n') cs
```

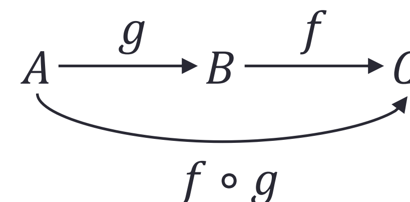
- 最後の疑問
  - 「`print . countLine`」の「`.`」は何？

関数の合成

```
headTail xs = head(tail xs)
```



- '.' 演算子
  - '`f . g`' は関数 '`f`' と '`g`' を合成した関数
  - '`(f . g) x`' の意味は '`(f (g x))`'




```
headTail xs = (head . tail) xs
```

```
headTail = head . tail
```

関数合成

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

- 2つの関数を合成して新しい関数を作る
  - `(f . g) x = f (g x)`
  - `f . g = \x -> f (g x)`

```
headTail :: [a] -> a
headTail xs = head $ tail xs
```

```
headTail :: [a] -> a
headTail = head . tail
```

- ($)との違い
  - `($) :: (a -> b) -> a -> b`
  - `f $ x = f x`

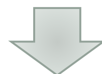
ファイルの行数を数える(6)

- 関数合成を使う

```
main = getContents >>= print . countLine
```

```
main = getContents >>= \cs -> print(countLine cs)
```

```
countLine cs = length $ filter (== '\n') cs
```



```
countLine = length . filter (== '\n')
```

countline6.hs

```
main = getContents >>= print . length . filter (== '\n')
```

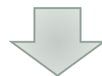
lines 関数

- 'lines cs' 関数
  - 文字列 cs を行ごとに分ける
  - lines::String -> [String]
  - lines "aaa\nbbb\nccc\n" → ["aaa", "bbb", "ccc"]
  - lines "aaa\n" → ["aaa"]
  - lines "aaa" → ["aaa"]
  - lines "\n" → [""]
  - lines "" → []

- cs の行数を数える
- cs を行ごとに分けたリストにして，その長さを求める

```
countLine cs = length(lines cs)
```

```
countLine cs = length $ lines cs
```



```
countLine = length . lines
```

airline-code.txt

- 世界の航空会社のIATAコード
  - See https://en.wikipedia.org/wiki/List_of_airline_codes

タブ

```
airline-code.txt
Q5      40-Mile Air     MILE-AIR        United States
W9      Abelag Aviation         ABG     Belgium
M3      ABSA Cargo      Turismo Brazil
MO      Abu Dhabi Amiri Flight SULTAN  United Arab Emirates
GB      ABX Air ABEX    United States
ZA      AccessAir       CYCLONE United States
VX      ACES Colombia   ACES    Colombia
...
C4      Zimex Aviation ZIMEX   Switzerland
3J      Zip     ZIPPER  Canada
Z4      Zoom Airlines   ZOOM    Canada
```

- タブで区切られている
  - IATAコード，航空会社，コールサイン，国名

ファイルの先頭10行を表示

head.hs

```
main = getContents >>= putStr . firstNLines 10

firstNLines n cs = unlines $ take n $ lines cs
```

- 上記プログラムを実行

```
% stack runghc head.hs < airline-code.txt
Q5        40-Mile Air       MILE-AIR United States
W9        Abelag Aviation   ABG      Belgium
M3        ABSA Cargo        Turismo  Brazil
MO        Abu Dhabi Amiri Flight     SULTAN   United Arab Emirates
GB        ABX Air  ABEX     United States
ZA        AccessAir         CYCLONE  United States
VX        ACES Colombia     ACES     Colombia
KI        Adam Air ADAM SKY Indonesia
Z7        ADC Airlines      ADCO     Nigeria
JP        Adria Airways     ADRIA    Slovenia
```

'unlines' と 'take'

- 'unlines xs' 関数
  - 'lines' 関数の逆.
  - リスト xs の文字列を改行で区切りながらつなげる.
  - unlines::[String] -> String
  - unlines ["aaa", "bbb", "ccc"] → "aaa\nbbb\nccc\n"
  - unlines ["aaa"] → "aaa\n"
  - unlines [""] → "\n"
  - unlines [] → ""
  - unlines ["aaa\n"] → ["aaa\n\n"]

- 'take n xs' 関数
  - リスト xs の先頭から n 要素を取り出したリストを作る.
  - リスト xs が n より短い時には, リストをそのまま返す.
  - take::Int -> [a] -> [a]
  - take 3 [5, 2, 4, 6, 8] → [5, 2, 4]
  - take 3 [5] → [5]
  - take 3 [] → []
  - take 3 "string" → "str"
  - take 0 [1, 2, 3] → []

練習問題6-1

head.hs

```
main = getContents >>= putStr . firstNLines 10

firstNLines n cs = unlines $ take n $ lines cs
```

- 上のプログラムの firstNLines の本体を '`$`' を使って書き直しなさい.
- 上のプログラムの firstNLines の本体を '`.`' を使って書き直すとどうなりますか.
- **`take`** を自分自身で定義してみなさい. 関数名は **`taake`** としましょう.

taake.hs

```
taake 0 _      = ...
taake _ []     = ...
taake n (x:xs) = ...
```

'reverse' と 'words'

- '`reverse xs`' 関数
  - リスト `xs` の要素の順番を逆転させたリストを返す.
  - `reverse [1, 2, 3]` → `[3, 2, 1]`
  - `reverse []` → `[]`
  - `reverse "string"` → `"gnirts"`
  - `reverse ""` → `""`
  - `reverse ["abc", "def", "ghi"]`
    → `["ghi", "def", "abc"]`
- '`words cs`' 関数
  - 文字列 `cs` を単語に分割する.
  - 空白(タブ, 改行を含む)で単語は区切られているものとする.
  - `words "This is a pen."` → `["This", "is", "a", "pen."]`
  - `words " a(1, 2, 3) "` → `["a(1,", "2,", "3)"]`
  - `words "a\nb\nc\n"` → `["a", "b", "c"]`
  - `words ""` → `[]`

練習問題6ー2

countbyte.hs

```
main = getContents >>= print ...
```

- ファイルの文字数を出力する.

練習問題6ー3

countword.hs

```
main = getContents >>= print ...
```

- ファイルの単語数を出力する.

練習問題6－4

reverse.hs

```
main = getContents >>= putStr ...
```

- ファイルの行を逆順に出力するプログラムを完成させなさい.
  - 関数合成を使って書いてみなさい.

```
% stack runghc reverse.hs < airline-code.txt
Z4        Zoom Airlines     ZOOM     Canada
3J        Zip       ZIPPER   Canada
C4        Zimex Aviation    ZIMEX    Switzerland
C4        Zimex Aviation    ZIMEX    Switzerland
Q3        Zambian Airways   ZAMBIANA Zambia
...
Q5        40-Mile Air       MILE-AIR United States
```

練習問題6—5

tail.hs

```
main = getContents >>= putStr . lastNLines 10

lastNLines n cs = unlines $ takeLast n $ lines cs

takeLast n xs = ...
```

- ファイルの最後の10行を出力するプログラムを完成させなさい.
  - 関数合成を使うとどうなりますか.

```
% stack runghc tail.hs < airline-code.txt
R3        Yakutia Airlines AIR YAKUTIA       Russia
YL        Yamal Airlines   YAMAL    Russia
Y8        Yangtze River Express     YANGTZE RIVER     China
IY        Yemenia  YEMENI   Yemen
2N        Yuzhmashavia     YUZMASH  Ukraine
Q3        Zambian Airways  ZAMBIANA Zambia
C4        Zimex Aviation   ZIMEX    Switzerland
C4        Zimex Aviation   ZIMEX    Switzerland
3J        Zip      ZIPPER   Canada
Z4        Zoom Airlines    ZOOM     Canada
```

練習問題6−6

oddline.hs

```
main = getContents >>= putStr . oddLines

oddLines ...
```

- ファイル奇数行だけを出力するプログラムを完成させなさい.

```
% stack runghc oddline.hs < airline-code.txt
Q5      40-Mile Air     MILE-AIR        United States
M3      ABSA Cargo      Turismo Brazil
GB      ABX Air ABEX    United States
VX      ACES Colombia   ACES    Colombia
Z7      ADC Airlines    ADCO    Nigeria
A3      Aegean Airlines AEGEAN  Greece
EI      Aer Lingus      SHAMROCK        Ireland
E4      Aero Asia International AERO ASIA       Pakistan
JR      Aero California AEROCALIFORNIA  Mexico
AJ      Aero Contractors        AEROLINE        Nigeria
...
```

関数とアクションのまとめ

関数	意味
`putStr cs`	文字列 **cs** を出力するアクションを返す
`putStrLn cs`	文字列 **cs** を出力し，改行を出力するアクションを返す．
`print x`	**x** の値を出力するアクションを返す．
`length xs`	リスト **xs** の長さを返す．
`take n xs`	リスト **xs** の先頭から **n** 要素だけのリストを返す．
`reverse xs`	リスト **xs** を逆順に並び替えたリストを返す．
`lines cs`	文字列 **cs** を行ごとに分割したリストを返す．
`unlines xs`	リスト **xs** の文字列を改行を挟んでつなげた文字列を返す．
`words cs`	文字列 **cs** を単語のリストに分割する．

アクション	意味
`getContents`	標準入力から読み込み文字列とするアクション