関数型プログラミング
第6回　基本的な値

萩野　達也
hagino@sfc.keio.ac.jp

基本的な値

- Haskellで取り扱うことができる基本的な値には以下のようなものがあります.
  - 真偽値
    - `Bool`型
  - 数値
    - `Int`型, `Integer`型, `Float`型, `Double`型
  - 文字
    - `Char`型
  - 文字列
    - `String`型 = `[Char]`型
  - タプル
    - `(a,b)`型
  - ユニット
    - `()`型
  - リスト
    - `[a]`型
  - 関数
    - `a -> b`型

真偽値

- **Bool型**

- 真偽値の値
  - **True**
  - **False**

- **Bool型に関する関数**

関数	凡例	意味
`not::Bool -> Bool`	`not x`	**x**が**True**なら**False**を返す. **x**が**False**なら**True**を返す.
`(&&)::Bool->Bool->Bool`	`x && y`	**x**と**y**の両方が**True**ならば**True**を返す. それ以外のときは**False**を返す.
`(||)::Bool->Bool->Bool`	`x || y`	**x**と**y**のいずれかが**True**ならば**True**を返す. それ以外のときは**False**を返す.

練習問題6－1

- notをifを使って定義すると次のようになります.

```
not x = if x then False else True
```

- パターンマッチで書くとどうなるでしょう？

```
not True = ...
not False = ...
```

- (&&)と(||)についてもifで定義するとどうなりますか?
- パターンマッチで書くとどうなりますか?

logic.hs

```
x && y = if x then ...
x || y = if x then ...

True && True = ...          True || True = ...
True && False = ...         True || False = ...
False && True = ...         False || True = ...
False && False = ...        False || False = ...
```

数値

- 整数型
  - `Int`型　　比較的小さな符号付整数
  - `Integer`型　　制限のない整数
- 整数リテラル
  - 10進表記　　`5`, `999`, `12345678901234567890`
  - 8進表記　　`0o644`
  - 16進表記　　`0x1f`
  - リテラルは文脈で`Int`型か`Integer`型が決まる
  - `(16::Int)`などで明示的に指定も可能
- 浮動小数点型
  - Float型　　単精度浮動小数点
  - Double型　　倍精度浮動小数点
- 浮動小数点リテラル
  - `1.5`
  - `3.141592`
  - `0.1543e+2`
  - `1343e-3`
  - `Float`か`Double`かは文脈で決まる
  - `(1.5::Double)`

数値の演算

使用例	意味
`x + y`	xとyの和
`x - y`	xとyの差
`x * y`	xとyの積
`x / y`	xとyの商（浮動小数点同士のみ）
`x `div` y`	xとyの整除（整数同士のみ，負の無限大に向かってまるめる）
`x `quot` y`	xとyの整除（整数同士のみ，ゼロに向かってまるめる）
`x `mod` y`	`(x `div` y)`のあまり（整数同士のみ） yの符号と一致
`x `rem` y`	`(x `quot` y)`のあまり（整数同士のみ） xの符号と一致
`x ^ y`	xのy乗
`-x`	xの符号反転
`negate x`	xの符号反転
`subtract x y`	`(y - x)`と同じ
`abs x`	xの絶対値
`odd x`	xが奇数ならTrue，偶数ならFalse
`even x`	xが偶数ならTrue，奇数ならFalse

数値の型の変換

- 整数型を他の型に変換する関数

使用例	意味
`toInteger x`	`Int`型の値`x`を`Integer`型に変換
`fromInteger x`	`Integer`型の値`x`を数値型に変換（返り値の型は文脈で決まる）
`fromIntegral x`	`Int`型または`Integer`型の値xを数値型に変換（返り値の型は文脈で決まる）

- 浮動小数点型を整数型に変換する関数

使用例	意味
`ceiling x`	`x`を下回らない最小の整数
`floor x`	`x`を上回らない最大の整数
`trancate x`	`x`と0との間で`x`に最も近い整数（`x`自身を含む）
`round x`	`x`に最も近い整数（2つある場合は偶数）

練習問題6－2

- IPv6ではアドレス空間が128bitになりましたが，これによってどれだけのものがインターネット上で区別可能になったのかを計算して出力しなさい．

ipv6.hs

```
main = print $ ...
```

- 税抜き123円のものが消費税8%を加えるといくらになるかを計算するプログラムを作成しなさい．
  - 小数点以下の端数は四捨五入することにします．

vat.hs

```
main = print $ ...
```

文字型

- Char型
  - ユニコード文字
- 文字リテラル
  - 'a'
  - '*'
  - ' '
- エスケープシーケンス

記述例	意味
'¥t'	タブ
'¥n'	改行
'¥r'	復帰
'¥v'	垂直タブ
'¥f'	改ページ
'¥a'	ベル
'¥b'	バックスペース

記述例	意味
'¥NNN'	10進数表記の数値NNNに対応する文字
'¥oNN'	8進数表記の数値NNに対応する文字
'¥xNN'	16進数表記の数値NNに対応する文字
'¥^X'	コントロールX
'¥''	シングルクォート
'¥"'	ダブルクォート
'¥¥'	バックスラッシュ(あるいは¥記号)

文字列型

- **`String`型**
  - 文字のリスト
  - **`[Char]`型**

- 文字列リテラル
  - **`"string"`**
  - **`"Keio"`**
  - **`"abc¥ndef¥n¥"hello¥65¥tend"`**

文字に関する関数(1)

- 文字種のテスト(型はすべてChar -> Bool)
  - 利用するにはData.Charモジュールをimportする必要があります.

使用例	意味
`isAlpha c`	文字cがUnicodeのアルファベットならば`True`
`isLower c`	文字cがUnicodeのアルファベット小文字ならば`True`
`isUpper c`	文字cがUnicodeのアルファベット大文字ならば`True`
`isAlphaNum c`	文字cがUnicodeのアルファベットまたは数字ならば`True`
`isDigit c`	文字cが数字(`0`～`9`)ならば`True`
`isHexDigit c`	文字cが16進数表記で使用する文字(`0`～`9`, `a`～`f`, `A`～`F`)ならば`True`
`isOctDigit c`	文字cが8進数表記で使用する文字(`0`～7)ならば`True`
`isSpace c`	文字cが空白類文字(空白やタブ, 改行など)ならば`True`
`isAscii c`	文字cがAsciiに含まれる文字(`'¥0'`～`'¥127'`)ならば`True`
`isLatin1 c`	文字cがLatin 1の文字(`'¥0'`～`'¥255'`)ならば`True`
`isPrint c`	文字cがUnicodeで表示可能文字ならば`True`
`isControl c`	文字cがUnicodeで表示不可能文字ならば`True`

文字に関する関数(2)

- 大文字・小文字の変換

関数	使用例	意味
`toLower :: Char -> Char`	`toLower c`	文字cがアルファベットの大文字のとき小文字を返す．それ以外の場合には文字c自身を返す．
`toUpper :: Char -> Char`	`toUpper c`	文字cがアルファベットの大文字のとき小文字を返す．それ以外の場合には文字c自身を返す．

- 文字と整数との変換

関数	使用例	意味
`ord :: Char -> Int`	`ord c`	文字cの文字コードを返す．
`chr :: Int -> Char`	`chr n`	文字コードがnの文字を返す．

タプル

- タプル（tuple）とは
  - いくつかの値の組．色々な型の値を組み合わせることが可能
  - 要素の個数と順序まで含めて型が決まる

- タプルの例
  - `(3, "string")` `:: (Int, String)`
  - `("lucky", 7)` `:: (String, Int)`
  - `(1, "string", [5, 4, 3])` `:: (Int, String, [Int])`
  - `('a', "string", (1, 3))` `:: (Char, String, (Int, Int))`

- ユニット
  - 0要素のタプル
  - `()` `:: ()`

タプルを扱う関数

- `fst :: (a, b) -> a`
  - 2要素のタプルの第1要素を返す
  - `fst (1, 2)` → `1`
  - `fst ("key", "value")` → `"key"`
- `snd :: (a,b) -> b`
  - 2要素のタプルの第2要素を返す
  - `snd (1, 2)` → `2`
  - `snd ("key", "value")` → `"value"`
- `zip :: [a] -> [b] -> [(a, b)]`
  - `zip xs ys` はリスト`xs`とリスト`ys`の各要素を横につないだタプルのリストを返す
  - `zip [1, 2, 3] [4, 5, 6]` → `[(1, 4), (2, 5), (3, 6)]`
  - `zip [1, 2, 3] ["a", "b"]` → `[(1, "a"), (2, "b")]`
- `unzip :: [(a, b)] -> ([a], [b])`
  - `zip`関数の逆で，タプルのリストをリストのタプルに分解する
  - `unzip [(1, 4), (2, 5), (3, 6)]` → `([1, 2, 3], [4, 5, 6])`
  - `unzip [(1, "a"), (2, "b")]` → `([1, 2], ["a", "b"])`

リスト

- 同じ型の値をならべたもの
  - 色々な型のデータをまぜることはできません.
  - 単方向リストのため前から後ろにたどることしかできません. 逆にはたどれません.

- リスト型
  - `[a]`
  - `[Char]`
  - `[Int]`

- リストの例
  - `[]`
  - `[1, 2, 3]`
  - `['a', 'b', 'c']`
  - `["aa", "bb", "cc"]`
  - `[['a', 'a'], ['b', 'b'], ['c', 'c']]`

- 文字のリストは文字列リテラルとして書くこともできる.
  - `"Hello, World¥n"`

「:」演算子

- `(:) :: a -> [a] -> [a]`
  - `x : xs`
  - リスト`xs`の先頭に`x`を追加したリストを返す

- 例
  - `1 : [2, 3]` → `[1, 2, 3]`
  - `'a' : "bc"` → `"abc"`

- 「:」演算子は右から順に結合する
  - `[1, 2, 3] = 1 : 2 : 3 : []`

リストの数列表記

- 一定の範囲の整数や文字などを含むリストを生成する特別な記法
  - `[1..7] = [1, 2, 3, 4, 5, 6, 7]`
  - `['a'..'e'] = ['a', 'b', 'c', 'd', 'e']`

- 一定間隔で要素を並べることもできる
  - `[1, 3..11] = [1, 3, 5, 7, 9, 11]`

- 無限のリストを作ることもできる
  - `[1..] = [1, 2, 3, 4, 5, 6, 7, 8, ····]`
  - `[1, 3..] = [1, 3, 5, 7, 9, 11, ····]`

リストに関する関数

関数名	適用例	意味
`length::[a]->Int`	`length xs`	リスト**xs**の長さを返す
`take::Int->[a]->[a]`	`take n xs`	リスト**xs**の先頭の**n**要素を取り出したリストを返す
`reverse::[a]->[a]`	`reverse xs`	リスト**xs**を反転させたリストを返す
`(++)::[a]->[a]->[a]`	`xs ++ ys`	リスト**xs**とリスト**ys**を連結したリストを返す
`concat::[[a]]->[a]`	`concat xs`	リストのリスト**xs**の要素を連結したリストを返す
`replicate::Int->a->[a]`	`replicate n x`	要素**x**を**n**個ならべたリストを返す
`lines::String->[String]`	`lines cs`	文字列**cs**を行ごとに分割したリストを返す
`unlines::[String]->String`	`unlines xs`	行ごとに分割されたリスト**xs**を改行を挟んでつなげた文字列を返す
`words::String->[String]`	`words cs`	文字列**cs**を単語ごとに分割したリストを返す
`unwords::[String]->String`	`unwords xs`	単語のリスト**xs**を空白を挟んでつなげた文字列を返す
`map::(a->b)->[a]->[b]`	`map f xs`	リスト**xs**の要素に**f**を適用した結果のリストを返す
`concatMap::(a->[b])->[a]->[b]`	`concatMap f xs`	リスト**xs**の要素に**f**を適用してできたリストを連結したリストを返す
`filter::(a->Bool)->[a]->[a]`	`filter f xs`	リスト**xs**の要素で**f**を適用して**True**のものだけを抽出したリストを返す
`any::(a->Bool)->[a]->Bool`	`any f xs`	リスト**xs**の要素で**f**を適用して**True**のものがあれば**True**を返す
`head::[a]->a`	`head xs`	リスト**xs**の先頭の要素を返す
`tail::[a]->[a]`	`tail xs`	リスト**xs**から先頭の要素を取り除いたリストを返す
`null::[a]->Bool`	`null xs`	リスト**xs**が空リストなら**True**を返す

Data.Listモジュール　関数名	適用例	意味
`tails::[a]->[[a]]`	`tails xs`	`[xs, (tail xs), (tail(tail xs)), ...]`を返す
`isPrefixOf::(Eq a)=>[a]->[a]->Bool`	`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**
`sort::(Ord a)=>[a]->[a]`	`sort xs`	リストxsの要素を昇順に並び替えたリストを返す
`group::(Eq a)=>[a]->[[a]]`	`group xs`	リストxsに連続して同じ要素が現れるとそれらをまとめたリストを返す

リスト内包表記

- 条件を満たす要素を集めてリストにする
- 例
  - `[abs x | x <- xs ]`
    - リスト`xs`の各要素xについて`(abs x)`を集める
  - `[(x, y) | x <- [1, 2, 3], y <- ['a', 'b', 'c']]`
    - `[1, 2, 3]`の各要素`x`と`['a', 'b', 'c']`の各要素`y`のすべての組み合わせについて`(x, y)`を集める
    - `[(1, 'a'), (1, 'b'), (1, 'c'), (2, 'a'), (2, 'b'), (2, 'c'), (3, 'a'), (3,'b'), (3, 'c')]`
- クイックソート関数

```
qsort []     = []
qsort (p:xs) = qsort lt ++ [p] ++ qsort gteq
                 where
                   lt   = [x | x <- xs, x < p]
                   gteq = [x | x <- xs, x >= p]
```

練習問題6－3

- 1から99までの奇数の2乗の和を計算しなさい．**map**を用いなさい．
  - $1^2+3^2+5^2+\cdots+99^2$

os1.hs

```
square n = n * n
main = print $ sum ...
```

- リスト内包表記を利用すると

os2.hs

```
main = print $ sum [ ... | ... ]
```

- 組み込みの関数やListモジュールの関数を用いるのではなく，再帰呼び出しを使って自分で計算する関数osを定義しなさい．

os3.hs

```
main = print $ os 99

os n = if n == 1 then ... else ...
```

実習: cat -nコマンド

- UNIXのcatコマンドでは-nオプションを付けることで行番号を付けて出力してくれます.
- これと同じ働きをするcatn.hsを作成してみましょう.

catn.hs

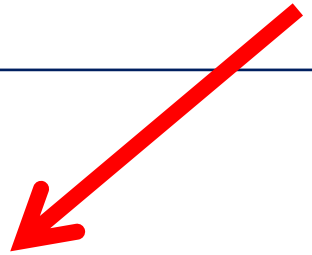
```
main = do cs <- getContents
          putStr $ numbering cs

numbering :: String -> String
numbering cs = unlines $ map format $ zipLineNumber $ lines cs

zipLineNumber :: [String] -> [(Int, String)]
zipLineNumber xs = zip [1..] xs

format :: (Int, String) -> String
format (n, line) = rjust 4 (show n) ++ " " ++ line

rjust :: Int -> String -> String
rjust width s = replicate (width - length s) ' ' ++ s
```

catn.hs解説

- **numbering cs**
  - 文字列**cs**を行ごとに分割し，行番号を付けた行をもう一度つなげる
- **zipLineNumber xs**
  - 行番号を振っていく
  - **zip [1..] xs**
  - **1**から始まる無限リストと行のリスト**xs**をタプルで組にする
- **format (n, line)**
  - 行**line**の先頭に行番号nを付加する
  - 行番号が4文字になるように空白でパディングする
- **rjust width s**
  - 文字列**s**の先頭に空白を補って**width**の長さになるようにする
- **show**関数
  - **show :: (Show a) => a -> String**
  - **show x**
  - 値**x**を文字列に変換して返す

練習問題6－4

- 与えられた引数を含む行を表示する fgrep.hs を修正して，一致した行番号を出力する行の前に追加しなさい．
  - 行番号は入力されるファイルの行番号であり，出力の行番号ではありません．



fgrepn.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
```