関数型プログラミング
第5回　タプルとパターンマッチ

萩野　達也
hagino@sfc.keio.ac.jp

Slide URL

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

型と値

- 値は型ごとに分類されている
  - 型は値の集合

- Haskellは静的な型チェックを行う
  - コンパイル時に型をチェックしてくれる
  - 型が合わないとエラーになる

- Haskellは型推論を行う
  - 明示的に型を指定しなくとも推論して補ってくれる

基本的な型

- 基本的な型には以下のようなものがある.
  - 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

関数の型

第1引数の型 -> 第2引数の型 -> ···· -> 返り値の型

- `add2 x = x + 2`
  - `Int -> Int`

- `gcd x y = if y == 0 then x else gcd y (x `mod` y)`
  - `Int -> Int -> Int`

- ```
  suum xs = if null xs then 0
            else head xs + suum(tail xs)
  ```
  - `[Int] -> Int`
  ```

型変数

- `length` 関数
  - `length [1, 2, 3]`
  - `length ['a', 'b']`
  - `length ["abc", "def"]`
  - 色々な型のリストに適用可能
  - 多層型（polymorphic）関数
  - `length`関数の型
    - `[a] -> Int`
    - `a`は型変数

- `map` 関数
  - `map square [1, 2, 3]`
  - `map` 関数の型
    - `(a->b)->[a]->[b]`

ghciを使って型を調べる

```
% stack ghci
Prelude> :type map
map :: (a->b)->[a]->[b]
```

型推論

- 関数適用では，実引数と関数の引数の型が一致している必要がある.
  - 型推論が行われることがある.

- `length` の型は `[a]->Int`
  - `length [1,2,3]`
  - `[1,2,3]` が `[Int]` であることから，型変数 `a` は `Int` であると推論される.

- `map length`
  - `map` の型 `(a->b)->[a]->[b]`
  - `length` の型は `[a]->Int`
  - `map length` の型は `[[a]]->[Int]` と推論される.

型の宣言

```
変数名 :: 型
```

- 変数の型を指定する

```
関数名 :: 第1引数の型 -> 第2引数の型 -> ···· -> 返り値の型
```

- 関数の型を指定する
  - 関数の定義の時にチェックされる
  - 型推論がうまくいかないときに指定する

```
length  :: [a] -> Int
reverse :: [a] -> [a]
map     :: (a->b)->[a]->[b]
(+)     :: Int -> Int -> Int
(++)    :: [a] -> [a] -> [a]
putStrLn:: String -> IO()
```

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"])`

練習問題5－1

- **zip** を自分で定義してみなさい.
  - `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")]`

ziip.hs

```
ziip xs ys = if not(null xs) && not(null ys)
             then ...
             else []
```

- **xs** および **ys** に関する分割統治で解く.
  - **xs** あるいは **ys** が空リストならば，**[]**
  - そうでない場合には，**xs** および **ys** を一つ短くしたリストに分割

練習問題5－2

- **unzip** を自分で定義してみなさい.
  - `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"])`

unziip.hs

```
unziip ts = if null ts then ([], [])
            else ...
```

- **ts** に関する分割統治で解く.
  - **ts** が空リストならば `[]` のタプル
  - そうでない場合には，**ts** を一つ短くしたリストに分割し，その結果を利用

let 式

```
let { 定義1; 定義2; 定義3; ... } in 式
```

- let 式を使うと，その式の中だけで有効な束縛を導入できる
  - 定義された束縛を行って式を評価する
  - 式の外では定義を参照することはできない

```
f n = let { x = n + 1;
            y = n + 2;
            z = n + 3 }
      in x * y * z
```

```
(let x = 2 in x + 3)*(let x = 3 in x + 4)
```

```
5 * 7
```

unziip.hs

```
unziip ts = if null ts then ([], [])
            else let { t = head ts;
                       p = unzip(tail ts) }
                 in (fst t:fst p, snd t:snd p)
```

let 式と where 節

- where 節

```
定義0 where { 定義1; 定義2; 定義3; .... }
```

  - 定義の中だけで有効な束縛を定義の後で導入する

- let 式

```
let { 定義1; 定義2; 定義3; ... } in 式
```

  - 式の中だけで有効な束縛を式の前に導入する

練習問題5－3

- 次の **where** 節を **let** 式を使って書き直しなさい.

divisors.hs

```
divisors x = filter divisible [1..x]
  where divisible y = x `mod` y == 0
```

divisors2.hs

```
divisors x = let ...
             in ...
```

パターンマッチ

- 値のパターンによる場合分け
  - 関数定義や case 式で用いることができる.

```
map :: (a -> b) -> [a] -> [b]
map f xs = if null xs then []
           else f(head xs):map f(tail xs)
```

```
map f []     = []
map f (x:xs) = f x : map f xs
```

- パターンの種類
  - 変数パターン
  - 「_」パターン（ワイルドカード）
  - リテラルパターン
  - タプルパターン
  - リストパターン
  - データコンストラクタパターン

変数パターン・「_」パターン

- 変数パターン
  - どんな値にでもマッチする
  - 変数をマッチした値に束縛する

```
id :: a -> a
id x = x
```

- 「_」パターン
  - ワイルドカードとも呼ばれる
  - どんな値にでもマッチする
  - マッチした値の変数への束縛などはない

```
const :: a -> b -> a
const x _ = x

map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

リテラルパターン・タプルパターン

- リテラルパターン
  - 値と指定したリテラルが等しいときにマッチする
  - 数値リテラル，文字リテラル，文字列リテラルを使うことができる

```
expandTab :: Char -> Char
expandTap '\t' = '@'
expandTab c    = c
```

- タプルパターン
  - タプルにマッチするパターン
  - タプルの各要素とマッチします
  - タプル内には任意のパターンを使うことができます
  - 「(パターン$_1$，パターン$_2$，パターン$_3$，・・・・)」

```
format :: (Int, String) -> String
format (n, line) = rjust 6 (show n) ++ " " ++ line
```

リストパターン・データコンストラクタによるパターン

- リストパターン
  - リストにマッチするパターン
  - 「[パターン$_1$, パターン$_2$, パターン$_3$, ・・・・]」

```
last []     = error "last []"
last [x]    = x
last (_:xs) = last xs
```

- データコンストラクタによるパターン
  - リストは空リスト「[]」と「:」によって作られています

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

練習問題5－4

- リストを結合する append をデータコンストラクタパターンを使って定義しなさい.

append.hs

```
append xs ys = if null xs then ys
               else (head xs):(append(tail xs) y)
```

append2.hs

```
append [] ys     = ...
append (x:xs) ys = ...
```

「@」パターン・ガード

- 「@」パターン
  - アズパターンともいわれる
  - 「変数名@パターン」
  - パターンにマッチさせ，値全体が変数名に束縛される

```
lstrip str@(c:cs) = if isSpace c then lstrip cs else str
```

- ガード
  - パターンの後に「|」を書き，その後ろにBool型の式を書くことでBool式がTrueの場合だけに限定できます．
  - 「$パターン_1$ $パターン_2$ ・・・・ | ガード」

```
joinPath :: String -> String -> String
joinPath a b | null a              = pathSep : b
             | last a == pathSep = a ++ b
             | otherwise           = a ++ pathSepStr ++ b
```

case式

```
case 式 of {
  パターンA | ガードA1 -> 式A1
            | ガードA2 -> 式A2
                :          :
            | ガードAn -> 式An;
  パターンB | ガードB1 -> 式B1
            | ガードB2 -> 式B2
                :          :
}
```

- 「式」の値でパターン$_?$にマッチさせガード$_{??}$がTrueとなる最初の式$_{??}$の値となる.

```
case str of {
  ""     -> "";
  (c:cs) -> toUpper c : cs
}
```

練習問題5－5

- 練習問題4－3の **fizzBuzz** を **case** 式のガードを使って定義しなさい.

fizzBuzz.hs

```
fizzBuzz n = map fb [1..n]
  where fb n = if n `mod` 15 == 0 then "Fizz Buzz"
               else if n `mod` 5 == 0 then "Buzz"
               else if n `mod` 3 == 0 then "Fizz"
               else show n
```

fizzBuzz2.hs

```
fizzBuzz n = map fb [1..n]
  where fb n = case n of {




  }
```

関数定義

- パターンマッチを使って関数を定義

```
関数名 パターン_A1 パターン_A2 ···· | ガード_A1 = 定義_A1
                                    | ガード_A2 = 定義_A2
                                    :
                                    :
関数名 パターン_B1 パターン_B2 ···· | ガード_B1 = 定義_B1
                                    | ガード_B2 = 定義_B2
                                    :
                                    :
```

- 関数名および変数名は識別子
  - アルファベットの小文字で始まる
  - アルファベット大文字・小文字，数字，アンダースコア，シングルクォートからなる
- 次の予約語は使えない
  - `case`, `class`, `data`, `default`, `deriving`, `do`, `else`, `if`, `import`, `in`, `infix`, `infixl`, `infixr`, `instance`, `let`, `module`, `newtype`, `of`, `then`, `type`, `where`, `_`

練習問題5－6

- 練習問題4－3の **fizzBuzz** を関数定義のガードを使って定義しなさい.

fizzBuzz2.hs

```
fizzBuzz n = map fb [1..n]
  where fb n = case n of {
    m | m `mod` 15 == 0 -> "Fizz Buzz"
      | m `mod` 5 == 0 -> "Buzz"
      | m `mod` 3 == 0 -> "Fizz"
      | otherwise -> show m
  }
```

fizzBuzz3.hs

```
fizzBuzz n = map fb [1..n]
  where fb n ...
```

練習問題5－7

- 与えられた年がうるう年かどうか調べる **leapYear** を定義しなさい.
  - 4で割り切れる年はうるう年である.
  - ただし，100で割り切れる年はうるう年とはしない.
  - しかし，400で割り切れる年はうるう年とする.

leapYear.hs

```
leapYear year ...
```

練習問題5－8

- 年と月が与えられたとき，その月の日数を返す関数 **monthDays** を定義しなさい．
  - 2月はうるう年のときには29日，それ以外の年は28日
  - 4月，6月，9月，11月は30日
  - それ以外の月は31日
- パターンマッチやガードを使って定義しなさい．

monthDays.hs

```
monthDays year month = ...
```

二項演算子の定義

パターン$_1$ 演算子 パターン$_2$ = 式

- 関数定義と同じようにパターンを使って定義
  - 記号の組み合わせで新しい二項演算子を定義することができる
  - 関数名を二項演算子として扱うには「`関数名`」とする
  - 二項演算子を関数として扱うには「(演算子)」とする

```
(||) :: Bool -> Bool -> Bool
True  || _ = True
False || x = x
```

優先順位	左結合	非結合	右結合
9	!!		. .
8			^ ^^ **
7	* / `div` `mod` `rem` `quot`		
6	+ -		
5			: ++
4		== /= < <= > >= `elem` `notElem`	
3			&&
2			||
1	>> >>=		
0			$ $! `seq`

練習問題5－9

- 引数に与えられた年月日が，西暦1年1月1日から何日目かを出力するプログラムを書きなさい．
  - 西暦1年1月1日は1日目とします．
  - 現在のグレゴリウス歴がずっと使われていたものとします．

days.hs

```
leap year = ...

yearDay year = if leap year then 366 else 365

monthDays year month = ...

days year month day = ...
```

```
% stack ghci
Prelude> :load days.hs
*Main> days 2023 11 13
738837
%
```

- これを使うとその日の曜日を計算することができます．
  - 西暦1年1月1日は月曜日です．