関数型プログラミング
第7回　基本的な構文

萩野　達也
hagino@sfc.keio.ac.jp

Slide URL

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

コメント

- 1行コメント
  - 「--」から行末まではコメントになり，コンパイラは無視します．

```
square n = n * n -- nの2乗
```

- ブロック形式のコメント
  - 「{-」から「-}」まではコメントになり，コンパイラは無視します．
  - ネストさせることも可能です．

```
{-
この関数は使っていないので一時的に削除
square n = n * n {- 2乗する関数 -}
-}
```

リテレイト形式

- プログラムにコメントを入れるのではなく，コメントの中にコードを書くこともできます．
  - 説明文書中にプログラムを書く．
  - 拡張子は.lhsにするのが慣例

```
> main = print $ square 5

関数squareは数値nの2乗を返す．

> square :: Int -> Int
> square n = n * n
```

```
\begin{code}
main = print $ square 5
\end{code}

関数squareは数値nの2乗を返す．
\begin{code}
square :: Int -> Int
square n = n * n
\end{code}
```

レイアウト構文とブレース構文

- レイアウト構文をこれまで使ってきました.
  - インデントをそろえることで複数の式をまとめることができます.

cat.hs

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

- 「{ }」と「;」を使うことでインデントに関係なく，複数式を束ねることができます.

最後にセミコロンはない

```
main = do { cs <- getContents;
            putStr cs }
```

```
main = do { cs <- getContents; putStr cs }
```

```
main = do {
cs <- getContents
; putStr cs }
```

オフサイドルールと式の継続

- レイアウト構文によって複数の式をまとめるときにインデントをそろえますが，インデントの場所を**オフサイドライン**といいます．

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

do式のオフサイドライン

Mainモジュールのオフサイドライン

- 現在のオフサイドラインより深くインデントした行は，継続行とみなされ，前の行の続きになります．

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

do式のオフサイドライン

if式

```
if 条件式 then 式1 else 式2
```

- 条件式は**Bool**型の式でないといけません.
- 条件式が**True**の時に$式_1$の値を返し，**False**の時に$式_2$の値を返します.
- $式_1$と$式_2$は式であり，コードブロック（複数の式をまとめたもの）ではありません.
  - 式を1つだけ書くことができます.

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ます.

```
map :: (a -> b) -> [a] -> [b]
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
```

「@」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
                :          :
  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
```

- of以降はコードブロックとみなされる.

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

関数定義

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 $パターン_{B1}$ $パターン_{B2}$ ・・・・ | $ガード_{B1}$ = $定義_{B1}$
| $ガード_{B2}$ = $定義_{B2}$
⋮
⋮

- パターンマッチを使って関数を定義
- 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
- 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`**, **`-`**

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`

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
```

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

```
resolverY2K y = base + y where base = 1900
```

```
expandTab :: Int -> String -> String
expandTab width cs = concatMap translate cs
  where
    translate '\t' = replicate width ' '
    translate c    = [c]
```

練習問題7－1

- 引数に与えられた西暦年がうるう年であればTrueを，そうでない場合にはFalseを出力するプログラムを書きなさい．
  - 4で割り切れる年はうるう年です．
  - ただし，100で割り切れる年はうるう年ではありません．
  - しかし，400で割り切れる年はうるう年です．

leap.hs

```
import System.Environment

main = do args <- getArgs
          print $ leap $ read $ head args

leap::Int -> Bool
leap y = if ...
```

```
% ghc leap.hs
...
% ./leap 2016
True
% ./leap 2017
False
%
```

練習問題7－2

- 引数に与えられた西暦と月から，その月が何日あるかを出力するプログラムを書きなさい．
  - たとえば，2020年2月はうるう年だったので29日ありました．
    - 2月はうるう年かどうかによって28日だったり29日だったりします．
  - 2018年3月は31日ありました．
  - **(xs !! n)** は **xs** の **n** 番目の要素を返します．

monthday.hs

```
import System.Environment

main = do args <- getArgs
          print $ monthDay (read $ args !! 0) (read $ args !! 1)

monthDay ... = ...
```

- 月ごとの場合分けは関数のパターンマッチやcase式などを用いなさい．if式でももちろんかまいません．

```
% ghc monthday.hs
...
% ./monthday 2020 2
29
% ./monthday 2018 11
30
%
```

練習問題7－3

- 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

```
import System.Environment

main =
  do args <- getArgs
     print $ days (read $ args !! 0) (read $ args !! 1) (read $ args !! 2)

yearDay year = if leap year then 366 else 365

monthDay year month = ...

days year month day = ...
```

```
% ./days 2018 11 14
737012
%
```

- 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．

練習問題7-4

- 誕生日を与えて，次の記念日を出力するプログラムを作成しなさい：
  - 生まれてから10日目
  - 生まれてから100日目
  - 生まれてから1000日目
  - 生まれてから10000日目

anniversary.hs

```
import System.Environment

main = do args <- getArgs
          let year = read $ args !! 0
          let month = read $ args !! 1
          let day = read $ args !! 2
          putStrLn $ yearStr year month (day + 10)
          putStrLn $ yearStr year month (day + 100)
          putStrLn $ yearStr year month (day + 1000)
          putStrLn $ yearStr year month (day + 10000)

yearStr:: Int -> Int -> Int -> String
yearStr year month day = ...
```

```
% ./anniversary 2001 1 1
2001/1/11
2001/4/11
2003/9/28
2028/5/19
%
```

練習問題7－5

- 引数に与えられた西暦年と月から，その月のカレンダーを出力するプログラムを書きなさい．

cal.hs

```
import System.Environment

main = do args <- getArgs
          purStr $ cal (read $ args !! 0)
                       (read $ args !! 1)

...
```

```
% ./cal 2018 11
Su Mo Tu We Th Fr Sa
                1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30
%
```

- ヒント
  - 練習問題7－3を使ってその月の最初の日の曜日を求める．
  - 練習問題7－2によってその月の日数を計算する．
  - カレンダーとしてきれいに表示する．
    - たとえば，リストを作成して週ごとに折り曲げるとか？