関数プログラミング用言語としてのHaskell

山下伸夫

2017-08-20

関数プログラミングとは

関数プログラミング言語を使う

関数プログラミングとは

関数プログラミング言語を使う

なぜ関数プログラミング言語？

関数プログラミングとは

副作用をほとんど使わない

関数プログラミングとは

副作用をほとんど使わない

で？ どうする？

関数プログラミングとは

関数を用いるプログラミング

なぜ関数？

関数プログラミングとは

データから欲しい情報を構成する

仕組みは？

関数プログラミングとは

データから欲しい情報を構成する

関数がその手段

なぜ関数プログラミングしたい

「判りやすい」プログラミング

- 抽象的に判りやすく考えたい
- 簡潔で読みやすいプログラム

なぜ関数プログラミングしたい

「判りやすい」プログラミング

- 対象を抽象化し構造化して考えやすい
- 高度な抽象概念の記述に組織的で簡潔な記述が使える

Haskell

- 言語仕様： Haskell 2010 Language Report
- 言語実装： Glasgow Haskell Compiler 8.2.1

- 汎用の純粋関数プログラミング言語
- 高階関数，非正格の意味論，静的多相型付け，
  利用者定義の代数的データ型，パターン照合，
  リストの内包表記，モジュールシステム，モナド I/O システム
- リスト，配列，任意倍長整数，固定倍長整数，浮動小数点数
- 遅延評価型関数型言語

Haskell で関数プログラミング

ポイントは,

- 欲しい値 (value) の仕様は型で構成する
- 欲しい値 (value) は式 (expression) で構成する

Haskell で関数プログラミング

コードの読み方のポイントは関数の型シグネチャ

- 関数の型シグネチャは　`f :: a -> b`　なら
  - 関数適用対象（引数）の型が　`a`
  - 関数適用結果（返値）の型が　`b`
- 型シグネチャ中の `->` は型構成演算子で右結合
  - `g :: a -> b -> c`　は　`g :: a -> (b -> c)`
  - 関数適用対象は　`a`　適用結果は　`b -> c`
  - 関数適用は左結合なので　`g x y`　は　`(g x) y`

Haskell で関数プログラミング

お題：

コンソールから入力された得点をそのつど評価して
成績を表示，入力終了後成績分布を表示

Haskell で関数プログラミング

```haskell
grading :: String -> String
grading = undefined
--
-- 関数合成演算子 (.)
-- (.) :: (b -> c) -> (a -> b) -> (a -> c)
-- (f . g) x = f (g x)
--
--
--
--
--
--
--
--
```

Haskell で関数プログラミング

```haskell
grading :: String -> String
grading =  promptResult . gradeGathering . divideIntoInts
--
-- 関数合成演算子 (.)
-- (.) :: (b -> c) -> (a -> b) -> (a -> c)
-- (f . g) x = f (g x)
--
--
--
--
--
--
--
--
```

Haskell で関数プログラミング

```haskell
grading :: String -> String
grading =  promptResult . gradeGathering . divideIntoInts

divideIntoInts :: String -> [Int]
divideIntoInts =  undefined

data Grade = A | B | C | D deriving (Show)
type Stat  = (Int, Int, Int, Int)

gradeGathering :: [Int] -> (Stat, [Grade])
gradeGathering = undefined

promptResult :: (Stat, [Grade]) -> String
promptResult = undefined
```

Haskell で関数プログラミング

```haskell
divideIntoInts :: String -> [Int]
divideIntoInts =  takeWhile (0 <) . map read . lines
--
-- lines :: String -> [String]
--
-- map :: (a -> b) -> ([a] -> [b])
-- map read :: [String] -> [Int]
--
-- takeWhile :: (a -> Bool) -> ([a] -> [a])
-- takeWhile (0 <) :: [Int] -> [Int]
--
--
--
--
```

Haskell で関数プログラミング

```haskell
gradeGathering :: [Int] -> (Stat, [Grade])
gradeGathering = mapAccumL gradeGather (0,0,0,0)
--
-- mapAccumL :: (acc -> a -> (b, acc)) -> acc -> [a] -> (acc,
-- mapAccumL gradeGathering (0,0,0,0) :: [Int] -> (Stat, [Grad
--
-- gradeGather :: Stat -> Int -> (Stat, Grade)
--
--
--
--
--
--
--
```

Haskell で関数プログラミング

```haskell
gradeGathering :: [Int] -> (Stat, [Grade])
gradeGathering = mapAccumL gradeGather (0,0,0,0)

gradeGather :: Stat -> Int -> (Stat, Grade)
gradeGather stat score = (upd g stat, g)
  where
    upd A (a,b,c,d) = (a+1,b,c,d)
    upd B (a,b,c,d) = (a,b+1,c,d)
    upd C (a,b,c,d) = (a,b,c+1,d)
    upd D (a,b,c,d) = (a,b,c,d+1)
    g | score < 50 = D
      | score < 65 = C
      | score < 80 = B
      | otherwise  = A
```

Haskell で関数プログラミング

```haskell
promptResult :: (Stat, [Grade]) -> String
promptResult ~(stat, gs)
  = "Score? " ++ concatMap prompt gs ++ pprStat stat

prompt :: Grade -> String
prompt g = show g ++ "\nScore? "

pprStat :: Stat -> String
pprStat (a,b,c,d)
  =  intercalate ", " (zipWith ppr [A,B,C,D] [a,b,c,d]))
  ++ "\n"

ppr :: Grade -> Int -> String
ppr g c = show g ++ ": " ++ show c
```

Haskell で関数プログラミング

```
import Data.List
import System.IO

main :: IO ()
main =  hSetBuffering stdout NoBuffering
     >> interact grading
--
--
--
--
--
--
--
--
```

Haskell で命令プログラミング

```haskell
import System.IO
import Text.Printf

main :: IO ()
main = do
  { hSetBuffering stdout NoBuffering
    putStr "Score? "
    (a,b,c,d) <- loop (0,0,0,0)
    printf "A: %d, B: %d, C: %d, D: %d\n" a b c d
  }
--
--
--
--
```

Haskell で命令プログラミング

```
loop :: Stat -> IO Stat
loop (a,b,c,d) = do
  { input <- getLine
    let score = read input :: Int
    case score of
      _ | score < 50 -> do
            { print D; loop (a,b,c,d+1) }
        | score < 50 -> do
            { print C; loop (a,b,c+1,d) }
        | score < 50 -> do
            { print B; loop (a,b+1,c,d) }
        | otherwise  -> do
            { print A; loop (a+1,b,c,d) } }
```