関数プログラミング入門

IIJ II
山本和彦

2010.12.17 14:50～16:00

プログラマーは二つに分類される

もっとプログラミングが
うまくなりたいと
思う人

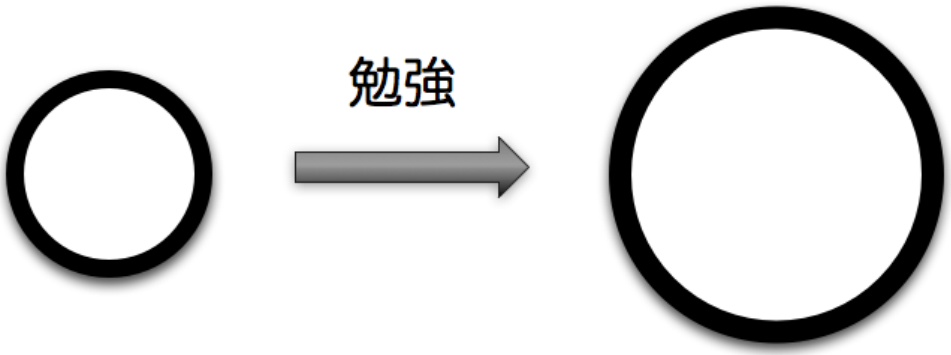
もっとプログラミングが
うまくなりたいとは
思わない人

うまくなりたいとは思わない人のために

すべてのプログラムの構造は、
これだけで実現できる。

以上、終わり！

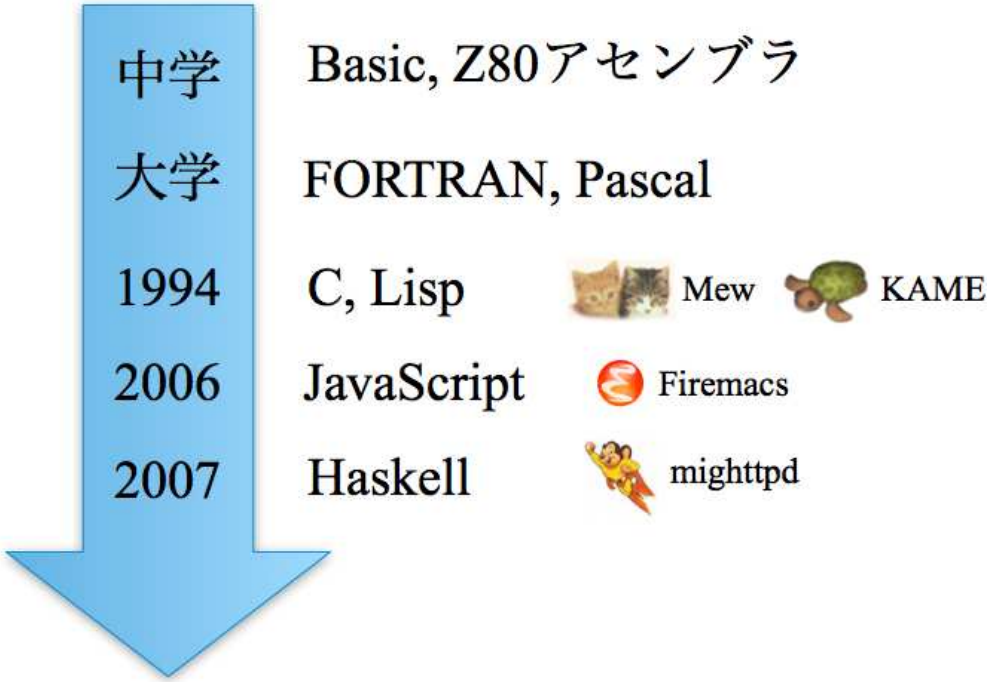
うまくなりたいと思う人のために



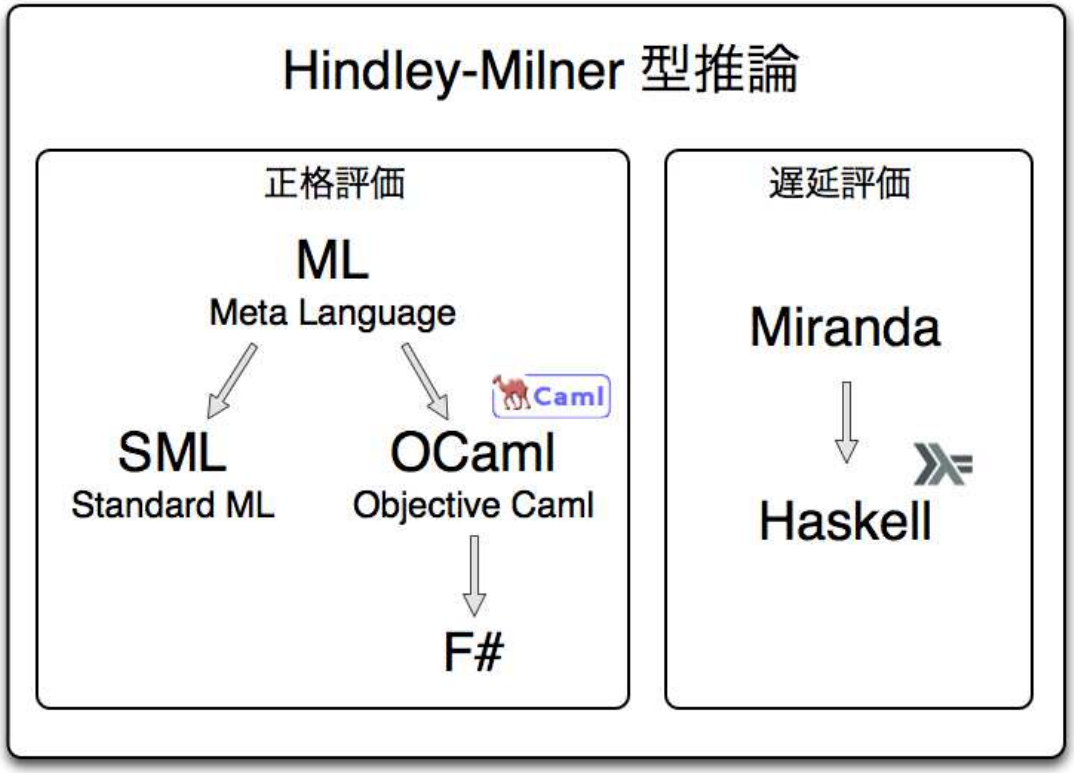
今日学ぶこと明日学ぶべきこと

自己紹介

- 山本和彦 <kazu@iij.ad.jp>
- (株) IIJ イノベーションインスティテュート
  主幹研究員



関数型言語の仲間達



今日は Haskell を使います

パラダイムの違い

命令プログラミング	関数プログラミング
命令を列挙する `A; B; C;`	関数を引数に適用する
状態がある	状態はない
破壊的代入を使う	(値を破壊したくなったら) 新たな値を作る

例題

- 入力として整数のリスト 10, 20, 30, 40, 50 がある
- 0 から数えて n 番目の要素には n を掛ける
- それらをすべて足し合わせる

- つまり、以下のような計算をする

10 * 0 + 20 * 1 + 30 * 2 + 40 * 3 + 50 *4 = 400

10 * 0 + 20 * 1 + 30 * 2 + 40 * 3 + 50 *4 = 400

命令型プログラマーなら
for 文か類似のループで
問題を解く

不格好な for 文

- Douglas Crockford のエッセイ
  「JavaScript: 世界で最も誤解されているプログラミング言語」
  http://www.crockford.com/javascript/javascript.html

波括弧や不格好な for 文がある
JavaScript の C ライクな文法を見ると、
通常の命令型言語のように思える。

これは誤解を与えやすい。
なぜなら、JavaScript は、
C や Java とよりも
関数型言語 Lisp や Scheme との方が
共通点が多いからだ。

for 文って不格好なの？

for の秘密

- 山本和彦は、for について授業で熱く語っていた
- for の意味
  - for は期間の for だ！
  - 例) for two days
- 非対称範囲を使え！
  - `for (i = 0; i < N; i++) { }`
  - 左の境界は入るが、右の境界は入らない
  - 0 から始めると配列と相性がよい
  - N をそのまま使える
  - 個数 = 最後 - 最初 + 1
    - (N - 1) - 0 + 1 = N
  - 不等号を使うと安全性
    - コンピュータでは、1/3 + 1/3 + 1/3 ≠ 1

JavaScript で不格好な for 文

```
function func(inp) {
  var ret = 0;
  for (var i = 0; i < inp.length; i++) {
      ret = ret + inp[i] * i;
  }
  return ret;
}

func([10,20,30,40,50]);
→ 400
```

- これが不格好と思わない人は、
  かっこいい解決方法を知らないのだろう。。。

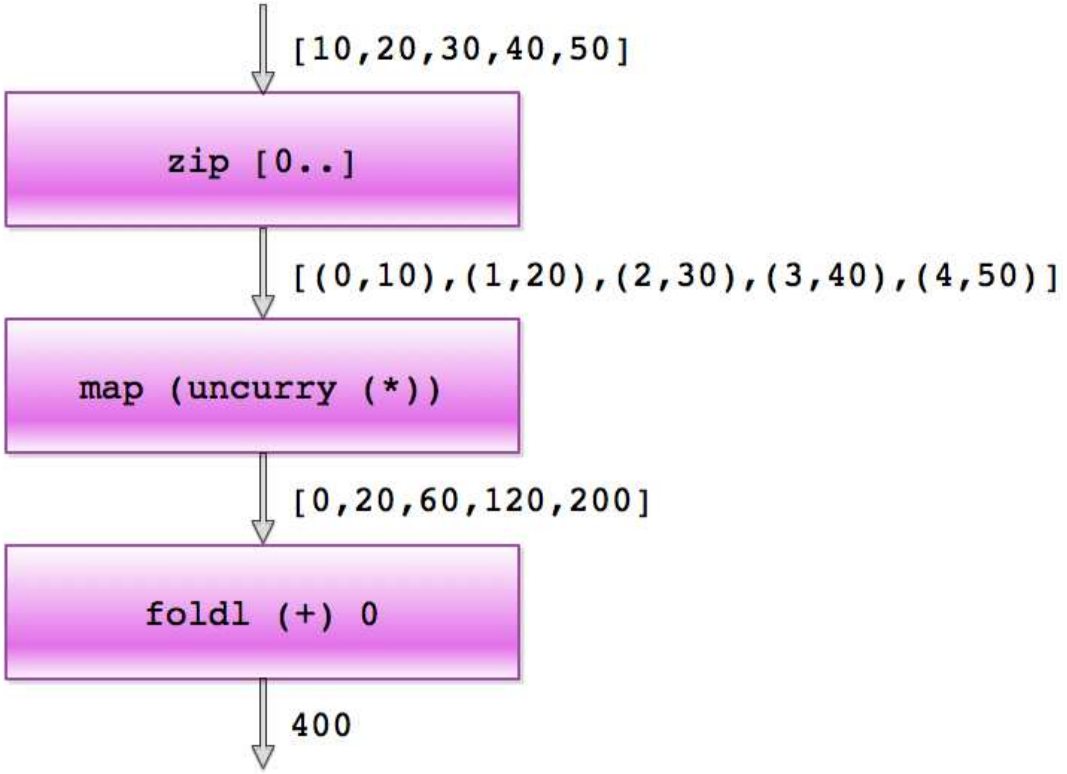
Haskell で map & fold

```
  zip [0..] [10,20,30,40,50]
→ [(0,10),(1,20),(2,30),(3,40),(4,50)]

  map (\(i,x) -> x*i) (上記の式)
→ [0,20,60,120,200]

  foldl (+) 0 (上記の式)
→ ((((0 + 0) + 20) + 60) + 120) + 200
→ 400
```

- 関数を合成する

```
  func = foldl (+) 0
       . map (\(i,x) -> x*i)
       . zip [0..]

  func [10,20,30,40,50]
→ 400
```

関数プログラミングと信号回路



- 関数プログラミングの極意
  - バグの入り込みにくい小さな関数をつなぎ合わせる

関数プログラミングとは
「関数を引数に適用すること」だと言う
プログラミング手法だとみなせる。

そして、関数型言語とは、
関数型の手法を提供し奨励している
プログラミング言語である。

今日のテーマ

型に守られたプログラミング
型を考えることがプログラミング

動機 其の一

なぜバグはなくならないのか？

動機 其の二

なぜ人の書いたコードは読みにくいのか？

その理由は
あまりにも自由に
プログラムを書くから

規律あるプログラミングのススメ

- 使い捨てのプログラム
  - 自由に書いてもよい
- 保守するプログラム
  - 規律を守って書く
  - 背筋を伸ばして書く
  - 他人が理解できるように
  - 未来の自分は他人です

型から規律とは何かを考えてみる

型の神話

型を書くのは面倒	あなたの言語の文法が 冗長なだけ
型なんて 役に立たない	あなたの言語の型システムが 貧弱で役に立たないだけ
コンパイル するのは面倒	コンパイルしなくても Haskell のコードは動かせる

この式の型は何？

分岐、繰り返し、逐次

型の書き方

- A -> B
  - 型 A のデータを取り、型 B のデータを返す

```
isDigit :: Char -> Bool

isDigit ’0’
→ True

isDigit ’a’
→ False
```

- A -> B -> C
  - 型 A と型 B のデータを取り、型 C のデータを返す

```
(+) :: Int -> Int -> Int

makeString :: Char -> Int -> String

makeString ’z’ 3
→ "zzz"
```

- 以下同様

if 式

- × else 節のない if 式があるとしたら

```
foo n = if n == 3 then n + 1
```

  - 条件式が真のとき `Int -> Int`
  - 条件式が偽のとき `Int -> ？`

- if 式には必ず then 節と else 節を書く

```
collatz :: Int -> Int
collatz n = if even n
            then n `div` 2
            else n * 3 + 1
```

繰り返し

- × for みたいな式があるとしたら

```
for (var i = 0; i < inp.length; i++) {
  if ... break;
  if ... continue;
}
```

  - 型がまったく分からない

- 繰り返しは再帰で書く

```
makeString :: Char -> Int -> String
makeString c 0 = []
makeString c n = c : makeString c (n - 1)
```

  - 再帰の末端で型検査を受ける！

注意

コードを理解しようとは
思わないで下さい

型だけ理解して下さい

型変数とコンテナー型

- 型変数 = どんな型にも当てはまる

```
length :: [a] -> Int

length [1,2,3,4]
→ 4
length [’a’,’b’,’c’,’d’]
→ 4
```

- コンテナー型： `m a`
  - リスト：`[a]` あるいは `[]a`
  - 入出力：`IO a`

逐次

- × もし逐次があるとしたら

```
c <- getChar;
putChar c;
```

  - この式全体の式の型は何？

- 逐次ではなく、関数適用
  - 同じコンテナーだけをつないでいける

```
getChar >>= putChar
```

```
getChar ::          IO Char
putChar :: Char -> IO ()
```

```
>>= :: IO a -> (a -> IO b) -> IO b
```

- 構文糖衣としての逐次 do

```
do c <- getChar
   putChar c
```

逐次と if

- もしファイルが存在すれば削除

```
do exist <- doesFileExist "file"
   if exist
      then removeFile "file"  -- IO ()
      else return ()          -- IO ()

doesFileExist :: FilePath -> IO Bool
removeFile :: FilePath -> IO ()
```

- 制御構造は自作できる

```
when :: Bool -> IO () -> IO ()
when p s = if p then s else return ()
```

- もしファイルが存在すれば削除

```
do exist <- doesFileExist "file"
   when exist (removeFile "file")
```

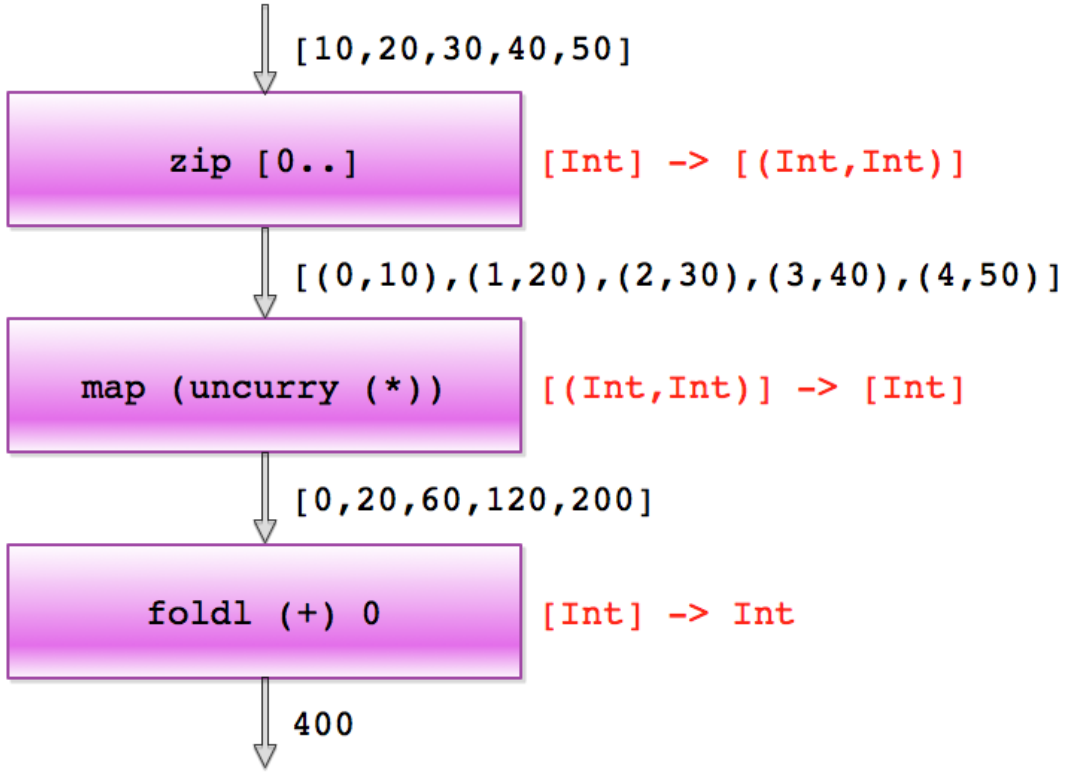
型推論と型検査

- トップレベルの関数には型を書く
- ローカルの関数の型は推論させる

```
qsort :: [a] -> [a]
qsort []   = []
qsort (p:ps) = qsort less ++ [p] ++ qsort more
  where
    less = filter (<p) ps     -- [a] と推論される
    more = filter (>=p) ps    -- [a] と推論される
```

- 型検査
  - コンパイラーは、すべてのつじつまが合うかを検査
  - すでに知っている関数の型
  - ユーザが指示した型
  - 推論された型

パラダイムの例で型検査



型の意味

設計図

型を考えることが
プログラミング

ドキュメント

仕様

コンパイルはテスト

保守性の向上

間違った変更を禁止する

型を書いておけば忘れない

型を台無しにするものたち

言外の型変換

unsigned int + int

スーパーな型

何でも表せる型

void *, Object

スーパーなデータ

どんな型にもなれるデータ

null, nil, None

うまく設計された言語には
こんなものはない

失敗する可能性はあるか？

- 型を見ても失敗する可能性があるのか分からない

```
FILE * fopen(const char *, const char *);
```

- 失敗しないときも FILE *
- 失敗するときも FILE *
  - 失敗すると NULL を返す
  - NULL の処理を忘れる ← バグの温床

Maybe

- 失敗するかもしれない型 Maybe

```
data Maybe a = Nothing | Just a
```

  - a は失敗しない型
  - Maybe a は、失敗するかもしれない型
  - Nothing は失敗
  - Just a は成功

- Maybe a から a を取り出すには Nothing も処理する必要がある

```
lookup :: a -> [(a, b)] -> Maybe b
```

```
case lookup key db of
   Nothing -> 0
   Just v -> v
```

空の木はどうやって表現するか？

- 空の木を NULL で表す

```
struct node {
  char key;
  int val;
  struct node* left;
  struct node* right;
}
```

- 空でないノードも struct node *
- 空のノードも struct node *
  - NULL の検査を忘れて NULL を使う

木の直接的な表現

```
data Tree k v = Empty
              | Node k v (Tree k v) (Tree k v)
```

- パターンマッチで両方を処理する必要がある

```
size tree = case tree of
  Empty -> 0
  Node k v l r -> size l + size r + 1
```

- トップレベルでパターンマッチ

```
size Empty = 0
size   (Node k v l r) = size l + size r + 1
```

それでもやはり
型検査をすり抜ける
バグも存在する orz

テストケースの自動生成

- 世間一般でのテストの自動化
  - ユニットテストのテストケースを手で書く
  - それを自動的に実行する
- QuickCheck
  - 性質を記述する
  - その性質を満たすテストケースを自動生成する

QuickCheck

- 整列されている

```
prop_ordered :: Ord a => [a] -> Bool
prop_ordered rs = ordered (qsort rs)
  where
    ordered []       = True
    ordered [x]      = True
    ordered (x:y:xs) = x <= y && ordered (y:xs)
```

- 二度整列しても整列されている

```
prop_sortsort :: Ord a => [a] -> Bool
prop_sortsort xs = qsort (qsort xs) == qsort xs
```

- モデル実装に対するテスト

```
prop_model :: Ord a => [a] -> Bool
prop_model xs = sort xs == qsort xs
```

まとめ

- 型は友達、コンパイラーは先生
- Haskell のプログラムは、1つの大きな式になる
  - 小さな式をつなぎ合わせる
- あらゆる部分が型検査を受ける
  - コンパイルはテスト
  - テスト駆動の開発
- 型検査をすり抜けたバグは QuickCheck で探す

お勧めの書籍

- プログラミングHaskell
  - オーム社
- Real World Haskell
  - O'REILLY