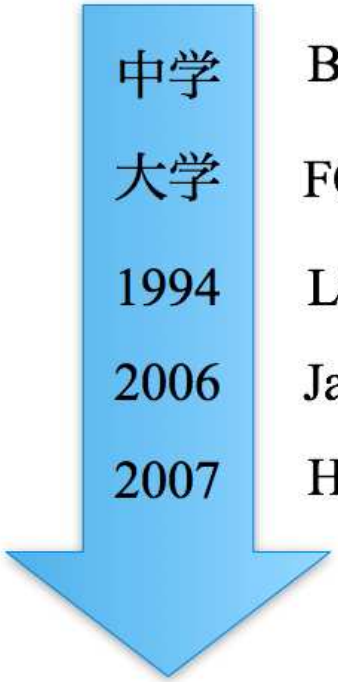
Ruby の聖地で Haskell を語る

山本和彦

山本和彦はこんなプログラマーです



中学	Basic, Z80アセンブラ	
大学	FORTRAN, Pascal	
1994	Lisp, C	 Mew  KAME
2006	JavaScript	 Firemacs
2007	Haskell	 mighttpd

愛すべき島根の銘酒


We are here
鳥取県
島根県
岡山県
広島県
山口県
I was born here

完全アウェー

Ruby と Haskell は真逆？

オブジェクト指向型	関数型
動的型付け	静的型付け

よくある偏見

Functional Language

関数型言語

Functional

↓

実用的

Functional language

↓

実用的言語

Haskell ではなんでも実装できます

イカしたアプリを知りたい人は
田中英行さんの資料を見て下さい

http://groups.google.com/group/
start-haskell/browse_thread/thread/782124d66eef9b6d

今日の話題

関数型

静的型付け

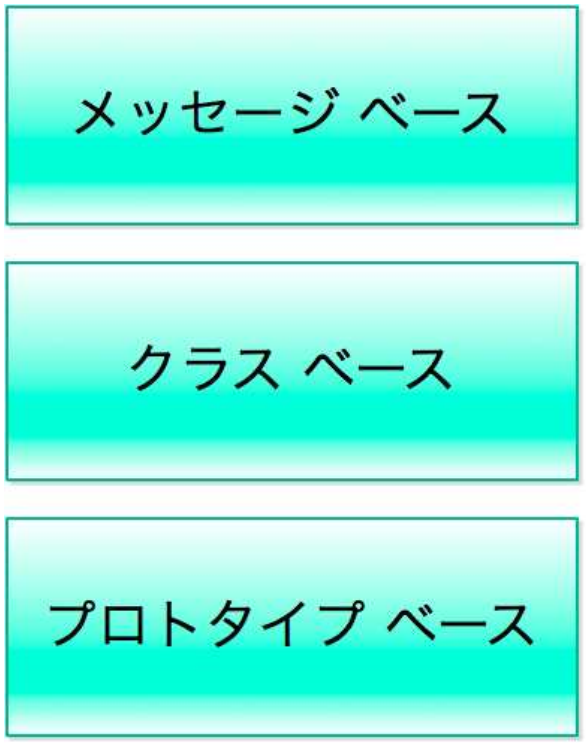
関数型を語る

共通定義のないオブジェクト指向

- それぞれの人が適当なサブセットを選んで、オブジェクト指向と呼ぶ

  http://practical-scheme.net/trans/reesoo-j.html

あなたのオブジェクト指向はどれ？



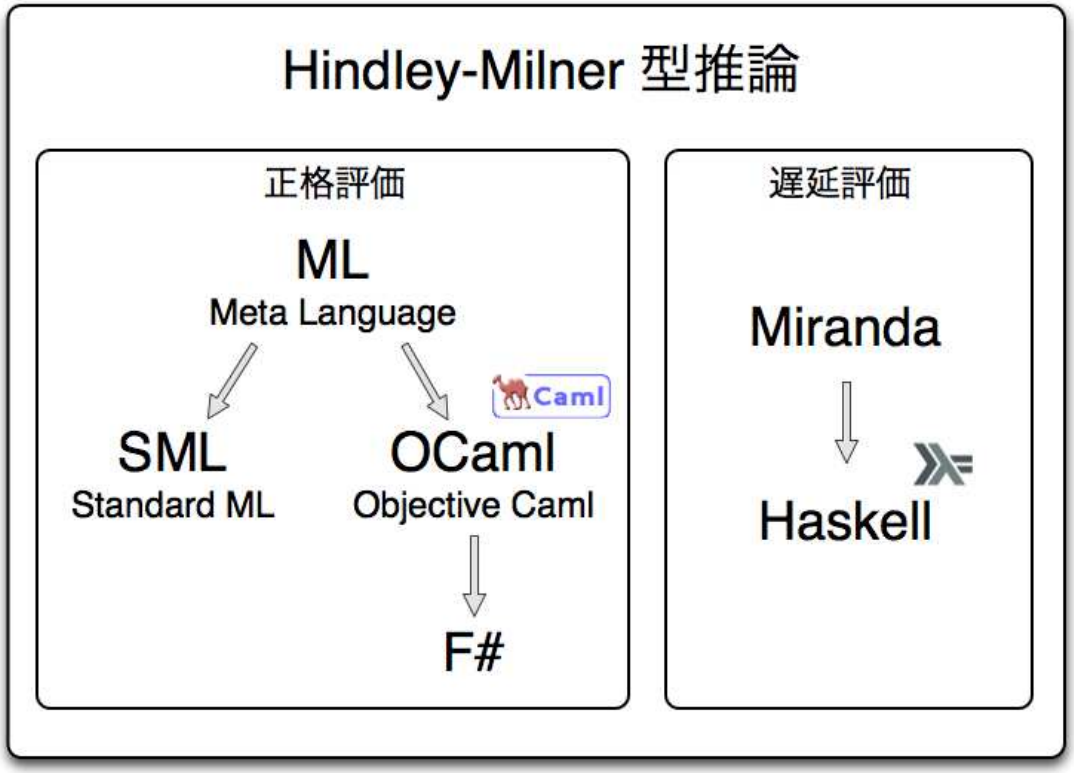
分類が間違っていたらごめんなさい

共通定義のない関数型言語

- それぞれの人が適当なサブセットを選んで、関数型と呼ぶ

関数が第一級の値	無名関数	クロージャ
カリー化	関数の再帰的定義	末尾再帰の最適化
破壊的代入がない	参照透明性	遅延評価

僕の関数型



「すべては式」が活かされている言語

大昔

ロジック

データ

構造化定理

ロジック

構造化定理

- 逐次
- 分岐
- 反復

データ

オブジェクト
指向型

関数型

変わりうる状態
(データ)

オブジェクト指向型

関数型

パラダイムの違い

関数プログラミング = 永続データプログラミング

例題

- 入力として整数のリストあるいは配列
  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

Ruby で逐次＆反復

- inject は使わない場合

```
def func (ar)
  sum = 0;                ← 命令の列挙
  i = 0;                  ← 命令の列挙
  ar.each{|x|
    sum += x * i;         ← 破壊的代入
    i += 1;               ← 破壊的代入
  }
  sum;
end
```

- 実行

```
func([10,20,30,40,50]);
→ 400
```

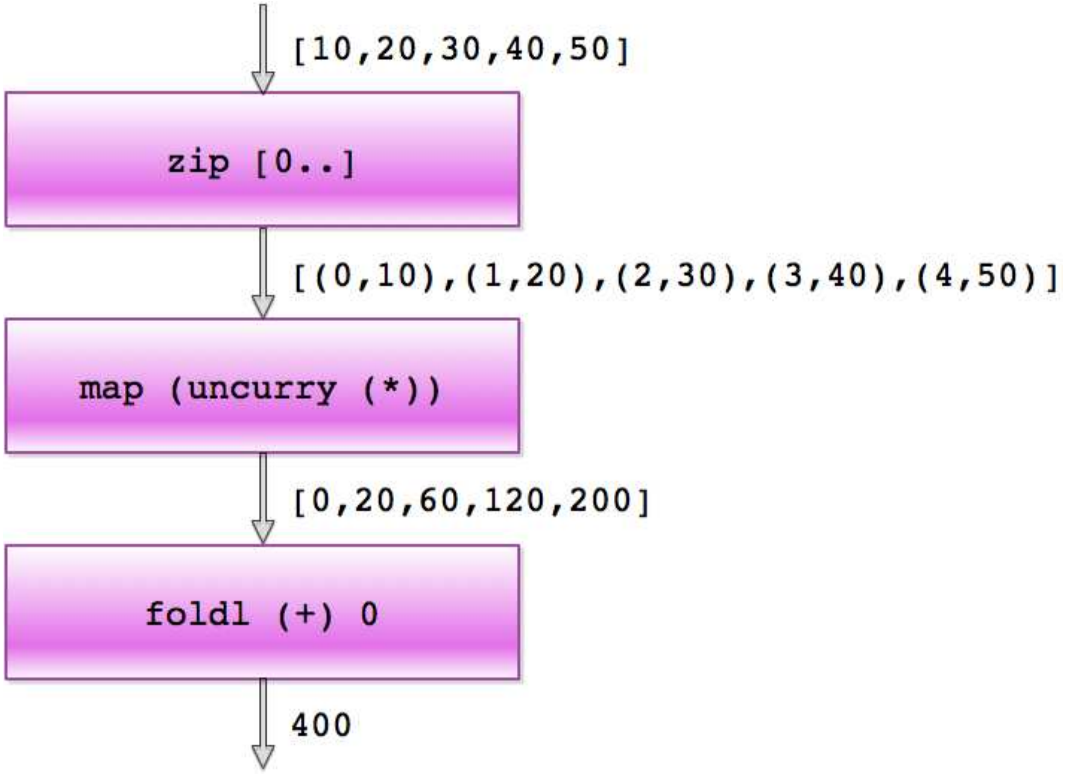
Haskell で map & reduce

```
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

Ruby で map & reduce

```
def func (ar)
  ar.zip((0..ar.length).to_a) \
    .map{|(x,i)|x*i} \
    .reduce(:+);
end

func([10,20,30,40,50]);
→ 400
```

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。

静的型付けを語る

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

Haskell の型は簡潔

- 型に別名を付ける

```
type FilePath = String
```

- ある型を別の型にする

```
newtype PostalCode = PostalCode Int
```

  - Java で基本型をクラスで包むのに相当

- 新しい型を作る

```
data Tree a = Leaf
            | Node a (Tree a) (Tree a)
```

- 関数のシグニチャ

```
lookup :: k -> Map k v -> Maybe v
lookup = ...
```

役に立つ静的型付け

動的型付け言語	型は書かない エラーは実行時に見つかる
一般的な静的型付け言語	型を書く コンパイル時にあまり エラーが見つからない
静的型付け関数型言語	型を書く コンパイル時に多く エラーが見つかる

Glasgow Haskell Compiler (GHC)

コンパイラー

```
% ghc foo.hs
→ foo
```

インタープリター

```
% ghci
> 1 + 1
2
```

スクリプト

```
% runghc foo.hs
```

型の意味

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

ドキュメント

設計図

型を考えることが
プログラミング


型はともだち
こわくないよ

型は仕様

すべては式である

```
(defun fibonacci (n)
  (let ((x 1) (y 1) (i 3))
    (while (<= i n)
      (setq y (+ x y))
      (setq x (- y x))
      (setq i (1+ i)))
    y))
```

式を文として利用
式を文として利用
式を文として利用

```
fibonacci :: Int -> Integer
fibonacci n = fib 1 0 1
 where
  fib m x y
    | n == m     = y
    | otherwise = fib (m + 1) y (x + y)
```

コンパイルはテスト

- あらゆる場所で式と式の型の関係が検査される

```
fibonacci :: Int -> Integer
fibonacci n = fib 1 0 1
 where
  fib :: Int->Integer->Integer->Integer
  fib m x y
    | n == m     = y
    | otherwise = fib (m + 1) y (x + y)
```

推論された型

- コンパイルがたくさんのバグを発見する
- コンパイルに通れば概ね思い通りに動く

ユルふわプログラミングにようこそ！

型は保守性の向上

保守性の向上

- 型は書いてあるので忘れない
- 理解してない変更はコンパイラーが禁止する

型はドキュメント

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

nullは
10億ドルの失敗

Tony Hoare

型はドキュメント

- 失敗するかもしれない型 Maybe
  ```
  data Maybe a = Nothing | Just a
  ```
  - `Int` は失敗しない型
  - `Maybe Int` は、失敗するかもしれない型
  - `Nothing` は失敗
  - `Just Int` は成功
- 失敗するかもしれない関数
  ```
  lookup :: k -> [(k,v)] -> Maybe v
  ```
- `Maybe Int` から `Int` を取り出すには `Nothing` も処理する必要がある

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

Haskell は型安全

- 型システムを台無しにするもの

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

- Haskell にはこれらがない

型は設計図

型は設計図

- 型を考えることがプログラミング

```
data TLV = TLV Type Length Value
newtype Type = Type Int
newtype Length = Length Int
newtype Value = Value [Int]

tlv :: Parser TLV
tlv = undefined

typ :: Parser Type
typ = undefined

len :: Parser Length
len = undefined

val :: Parser Value
val = undefined
```

まとめ

関数型言語による規律プログラミング

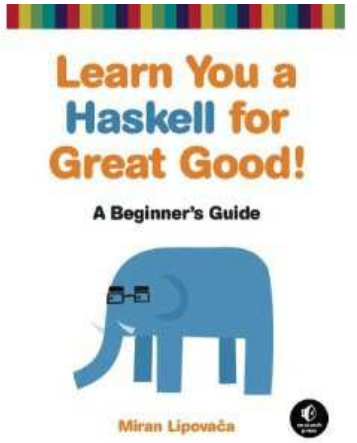
- プログラムは「書く」より「読む」方が難しい
- 保守するプログラムは自由に書いてはならない
- 関数型は、ほどよい制約 = 規律をプログラマーに課す
- バグは少なくなり、読みやすくなる
- プログラミングはコンパイラーが教えてくれる

僕は Haskell を覚えて
はじめてプログラミングとは
何か分かりました

コンパイラーは先生

- GHCはプログラミングを無料で
  徹底的に教えてくれる

お勧めの書籍



- プログラミングHaskell
  - オーム社
- Learn You a Haskell for Great Good!
  - No Starch Pr
  - 翻訳される予定です
- Real World Haskell
  - O'REILLY