

# メタプログラミングの光と闇

～ Haskell 編～

IIJ イノベーションインスティテュート

山本和彦

# Haskell は地球の裏からやってきた

命令型言語  
動的型付け  
正格評価



純粋関数型言語  
強い静的型付け  
遅延評価



Haskell は標準化された言語

Haskell 2010 Language Report

Haskell 98 はもう古い

# Glasgow Haskell Compiler

---

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
```

Haskell は  
型を書きたくなる言語

## 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 = ...
```

Haskell は  
「すべてが式である」  
を活かした言語

## 式を文として使うか否か

```
(defun fibonacci (n)
  (let ((x 1) (y 1) (i 3))
    (while (<= i n)
      (setq y (+ x y)) ← 式を文として利用
      (setq x (- y x)) ← 式を文として利用
      (setq i (1+ i))) ← 式を文として利用
    y))
```

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



# Haskell での コンパイルはテスト

## あらゆる場所で式と式の型の関係を検査

- Haskell のプログラムは、1つの大きな式

```
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
  - 型に関する間違い
  - 引数の数が間違っていないか
  - 名前が重なっていないか
- コンパイルに通れば概ね思い通りに動く

Haskell での開発は  
自然にテスト駆動となる

Haskell はツンデレです



型もあわせられない  
なんて最低ね。



型があっていて素敵。  
守ってあげたい。

## 型安全

---

- Haskell には型システムを台無しにするものがない

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

- コンパイルが通れば型に関する間違いがない

ソフトウェアを設計するには二つの方法がある。

一つは、とても簡潔にして**明らかに**欠陥がないようにする方法。

もう一つは、とても複雑にして**明らかな**欠陥がないようにする方法。

前者は後者より、はるかに困難である。



Tony Hoare



# ゆるふわプログラミングへようこそ

---

型を書くたび  
安心がふえるね



## QuickCheck

---

- コンパイルを通れば、型に関する間違いはないが、値に関する間違いは残る
- 値に関する間違いは QuickCheck で探す
- QuickCheck で性質を記述した例  
`qsort (qsort xs) == qsort xs`
- テストケースは自動的に生成される
- ある意味メタプログラミング？

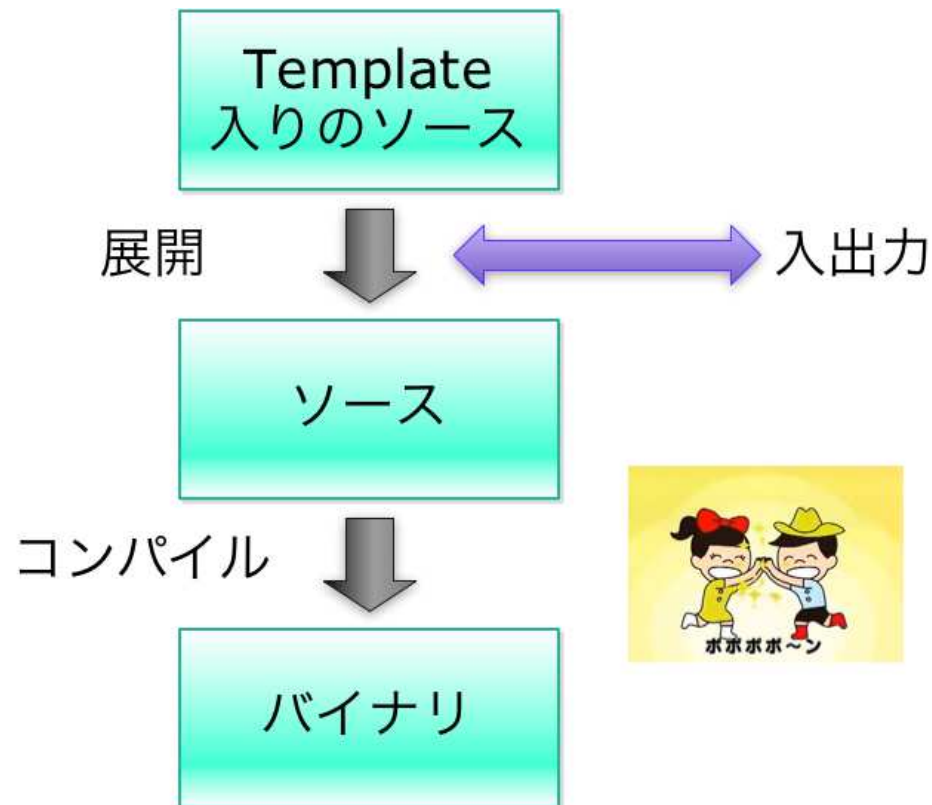
Haskell でのメタプログラミング

Template Haskell

準クォート

## Template Haskell の動作

- Template はコンパイル時に展開される



## バイラプレート

---

- 人を表す構造体 (種)

```
data Person = Person {  
    idnt :: Int  
    , name :: String  
}
```

- SQL のカラム (育てたい木)

```
fromPerson :: Person -> [SqlValue]  
fromPerson x = [  
    toSql (idnt x)  
    , toSql (name x)  
]
```

## Template Haskell の例

---

### ■ 構文木を種から育てる

```
mkFrom :: Name -> Q [Dec]
mkFrom t = do
  -- 構文木の種を得る
  TyConI (DataD [] name [] [RecC _ gs] [])
    <- reify t
  -- 構文木を育てる
  x <- newName "x"
  let fnm = mkName $ "from" ++ nameBase name
      tosql = mkName "toSql"
      trans (get,_,_) = VarE tosql `AppE`
                    (VarE get `AppE` VarE x)

      ds = map trans gs
      bdy = NormalB (ListE ds)
      cls = [Clause [VarP x] bdy []]
  return [FunD fnm cls]
```

## ボイラープレートの生成

---

```
data Person = Person {  
    idnt :: Int  
    , name :: String  
}
```

### ■ Before

```
fromPerson :: Person -> [SqlValue]  
fromPerson x = [  
    toSql (idnt x)  
    , toSql (name x)  
]
```

### ■ After

```
$(mkFrom ''Person)
```

メタプログラミングによる  
Domain Specific Language の例

Yesod



## Yesod の特徴

---

- Haskell の Web Application Framework

Web アプリ、CGI、  
FastCGI が作成可能

開発時はサーバの  
再起動が不要

nginx よりも速い

リンク切れフリー  
XSS フリー  
SQL Injection フリー

## 準クォートで DSL を囲む

---

### ■ Yesod でリンク切れを起こさない2つのページ

```
-- 準クォートとクォートされているDSL  
-- DSLを処理するパーサー名
```

```
$(mkYesod "Demo" [$parseRoutes |  
/      HomeR  GET  
/page1 Page1R GET  
| ])
```

```
-- DSLの中で変数展開
```

```
getHomeR = defaultLayout [$hamlet |  
<a href="@{Page1R}">Go to page 1.  
| ]
```

```
getPage1R = defaultLayout [$hamlet |  
<a href="@{HomeR}">Go home.  
| ]
```

## Haskell でパーサー

---

- JSON の BNF (RFC 4627)

value = object / array / number / string ...

- Parsec で JSON パーサーの定義

```
value :: Parser JSON
value = jsObject <|> jsArray
      <|> jsNumber <|> jsString ...
```

- BNF に従って実装すればよい(コツはいるけどね)
- Parsec を使って書くのは Haskell のコードそのもの
- コンパイラーのご加護がある

Haskell では  
Domain Specific Language  
の作成が簡単

Template Haskell  
準クオート  
パーサー

## 所感

---

- プログラムは書くより読む方が難しい
- メタプログラミングは読みにくい
- 結局、程度問題

