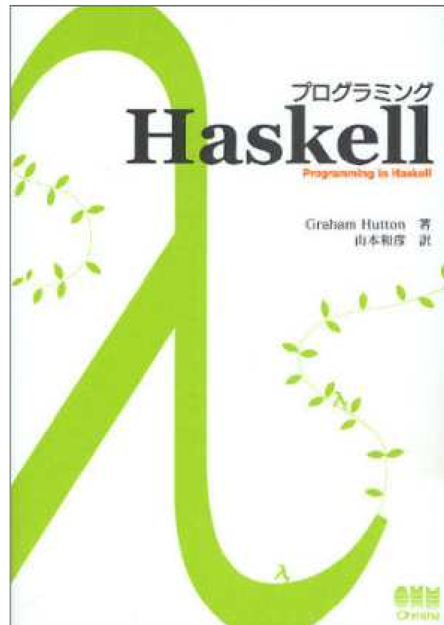
関数プログラミングが教えてくれる規律

山本和彦

自己紹介1

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

自己紹介2



プログラムには2つある

使い捨てる
コード

保守する
コード

使い捨てるコードは
自由に書いてよい

保守するコードは
規律を守って書くべき

なぜなら

プログラムは
書くより読む方が難しいから

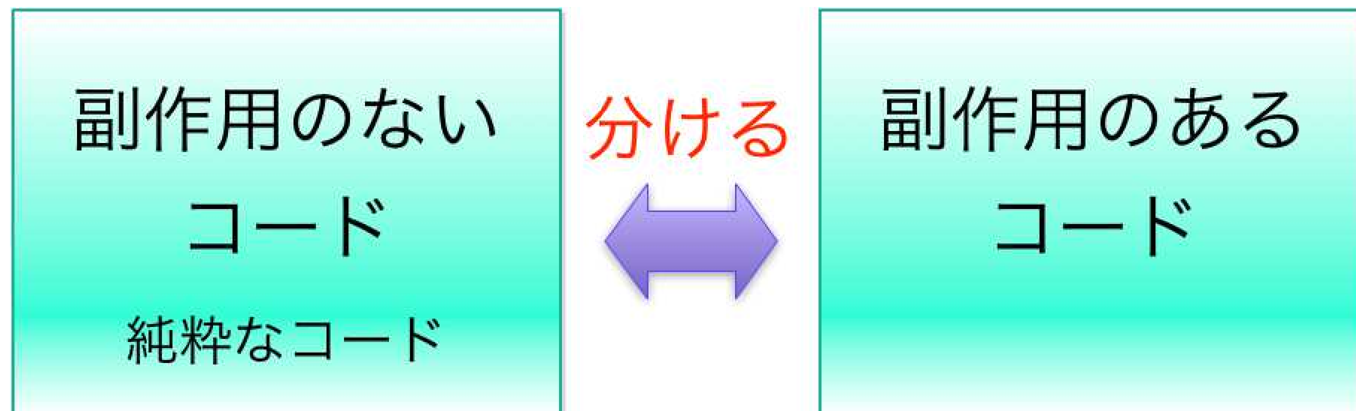
規律とは何か？

コーディングスタイルなどを
議論するのではレベルが低い

講義の内容

関数プログラミングが教えてくる
プログラミングの規律を学ぶ

もっとも大事なこと



まぜるな

危険

まぜられない

Haskell を使って説明します

純粋関数型言語

まぜられない

純粋な関数から副作用のある
関数を呼び出せない

Haskell

関数型言語

まぜられる

純粋な関数から副作用の
ある関数を呼び出せる

OCaml, SML
F#, Scala

今回は静的型付き関数型言語のみ

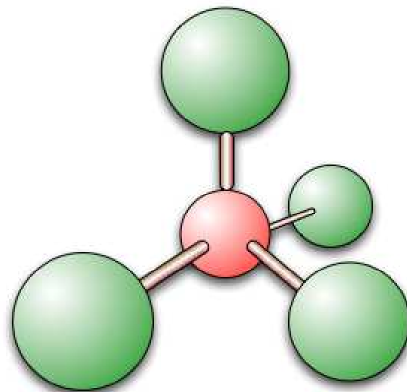
Haskell は
プログラミングをうまくなるための
養成ギプスとして使える

副作用とは何か？

入出力	ディスプレイ ファイル ネットワーク
状態の変更	グローバルな状態 ローカルな状態

グローバルな状態の変更

- グローバル変数への再代入(破壊的な代入)
- 同じグローバル変数を使う関数は強く結合する
  - 再利用は難しい
  - バグの温床

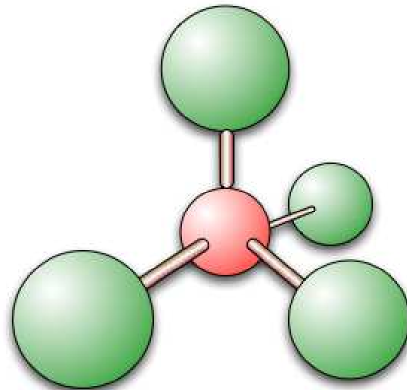


グローバル変数は必要悪？

- 基本的にグローバル変数は不要
  - 状態は引数として関数へ渡せばよい
  - グローバル変数を使ってきた人には面倒ですが
- 緊急事の必要悪
  - 例) 30 分以内にバグを直して、
    コードを置き換えないといけない。
    引数を増やしている暇はない

ローカルな状態の変更

- 関数に static な変数への再代入
- オブジェクトのインスタンス変数への再代入
- 同じインスタンス変数を使う関数は強く結合する
  - 再利用は難しい



関数プログラミングと変数

- 再代入可能な変数は、ほとんど使わない
  - 状態は引数として渡す
  - 値を破壊したくなったら、新たに作り出す
  - これだけでも多くの問題は解ける
- 再代入可能な型も用意されている
  - 参照型 (IORef)
  - 副作用のある関数の中で使う
  - 参照型も引数として渡す
    - グローバルな参照型は、緊急時にしか使わない

関数の目的

純粋な関数

式で構成する
計算をする

引数が同じなら同じ
結果を返す

副作用のある関数

文で構成する
副作用を起こす

引数が同じでも異なる
結果を返すこともある

一方向性

- 文から式を呼び出すのは OK
  - 副作用のある関数から純粋な関数は呼び出してよい
- 式から文を呼び出すのは NG
  - 純粋な関数から副作用のある関数は呼び出してはならない
  - 呼び出すと副作用のある関数になってしまうから

純粋関数型言語	一方向性を必ず要求 黒魔術としての抜け道はあり Haskell
関数型言語	式から文を呼び出せる OCaml, SML, F#, Scala

Haskell では型で純粋か分かる

- 純粋な関数

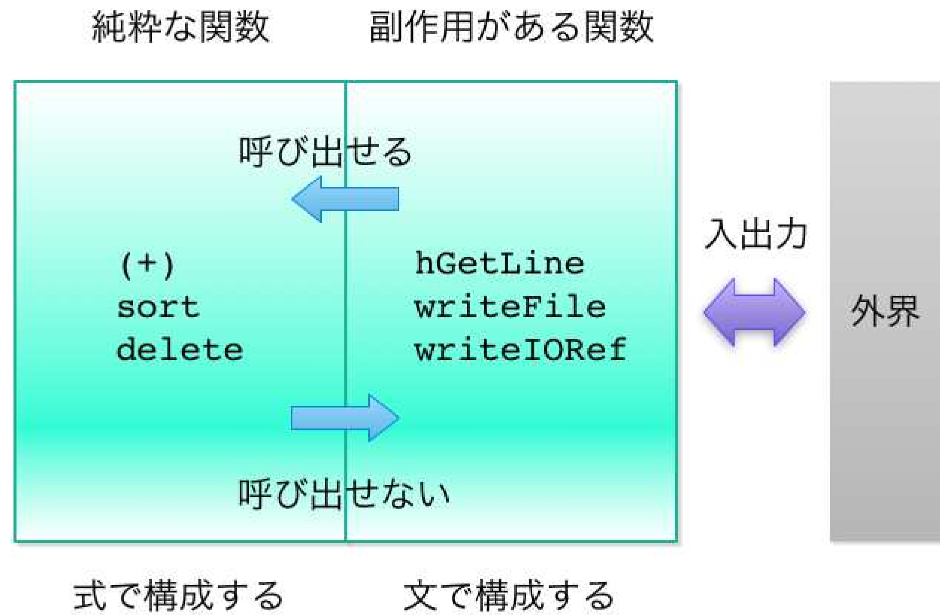
```
(+) :: Num a => a -> a -> a
sort :: Ord a => [a] -> [a]
delete :: Eq a => a -> [a] -> [a]
```

- 副作用がある関数

```
hGetLine :: Handle -> IO String
writeFile :: FilePath -> String -> IO ()
writeIORef :: IORef a -> a -> IO ()
```

- IO から純粋な関数は呼び出せる
- 純粋な関数から IO は呼び出せない

これまでのまとめ



例1) Ruby でフィボナッチ数列

- 純粋な関数だが文で構成されている例

```
def fibonacci (n)
  x = 1; y = 1; i = 3;
  while i <= n do
    y = x + y;     ← 式を文として利用
    x = y - x;     ← 式を文として利用
    i += 1;        ← 式を文として利用
  end
  y;
end
```

例1) Haskell でフィボナッチ数列

- 純粋な関数は式で構成される

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

- 繰り返しには再帰を使う

例2) 文字列を単語に分割

- 実行例

```
> split " foo bar   baz  "
["foo","bar","baz"]
```

- 実装

```
split :: String -> [String]
split s
  | s’ == ""  = []
  | otherwise = w : split s’’
  where
    s’ = dropWhile isSpace s
    (w, s’’) = break isSpace s’
```

例3) ファイルの単語数を数える

```
module Main where

import Data.Char

main :: IO ()              -- 副作用がある
main = do
    cs <- getContents      -- ファイルの中身を全部読む
    print (wc cs)          -- 純粋な関数を呼び出す

wc :: String -> Int        -- 純粋
wc xs = length (split xs)

split :: String -> [String]
split ...
```

失敗と例外

失敗する可能性はあるか？

- 型からは失敗する可能性があるのか分からない例

```
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

NULL とその仲間たち

- NULL, null, nil, None
- プログラマは、必ず NULL を踏む
- 21世紀の言語は、NULL を排除すべき

NULL なしで
どうやって失敗を表現するのか？

純粋な関数

全域関数にする

すべての引数に対して
値を返す

副作用のある関数

例外を使う

基本的には

純粋な部分関数を書いてはならない

純粋な関数は
失敗できないのか？

失敗を表すためには

直和型

を使う

直和型の例

- 真理値

```
data Bool = False | True
```

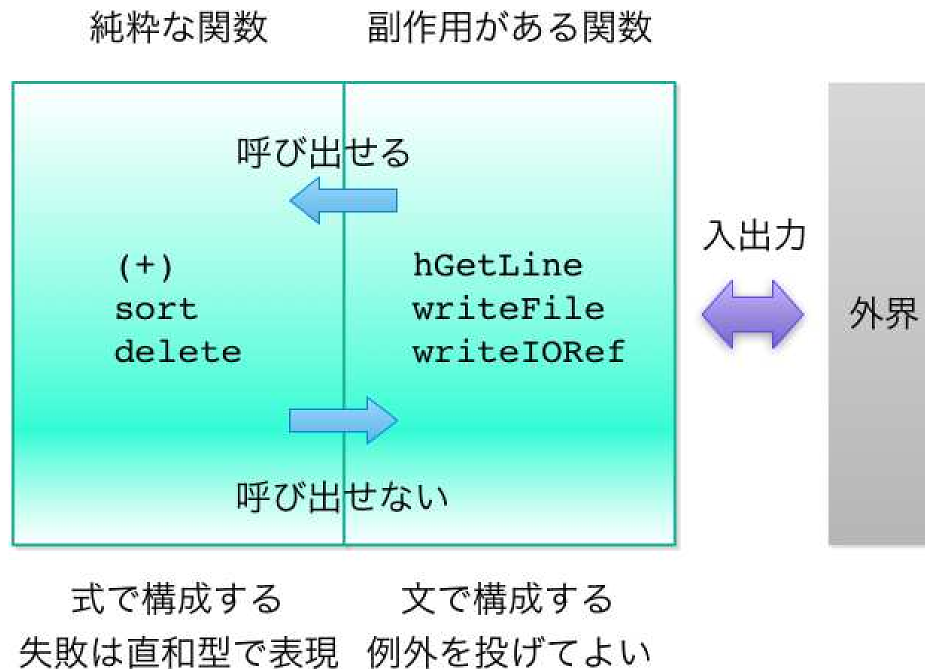
- 失敗と成功

```
data Either a b = Left a | Right b
```

- 理由を示さない失敗と成功

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

これまでのまとめ



IO での例外

- 例外を投げる

```
receive :: Socket -> IO String
receive s = do
    cs <- recv s 4096
    if cs == "" then
        throwIO ConnectionClosedByPeer
      else
        return cs
```

- 例外を捕まえる

```
-- serv は内部で receive を使っている
server :: Socket -> IO ()
server s = catch (serv s) logIt
  where
    logIt :: SomeException -> IO ()
    logIt e = print e
```

純粋関数での失敗

- 連想リストを検索する lookup

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

- 利用例

```
> lookup 5 [(2,'a'),(5,'b')]
Just 'b'

> lookup 1 [(2,'a'),(5,'b')]
Nothing
```

- 実装

```
lookup _key [] =  Nothing
lookup  key ((x,y):xys)
   | key == x    =  Just y
   | otherwise   =  lookup key xys
```

失敗とパターンマッチ

- Just a の a を取り出すにはパターンマッチ使う
- Nothing も処理しないとコンパイラが警告する

```
getValue :: Int -> [(Int, Char)] -> Char
getValue key db = case lookup key db of
    Just c  ->  c
    Nothing -> 'z'
```

- 失敗の処理を忘れない

純粋な関数の効能

高いエラー検出率

テストが容易

保守が容易

式

式と式の型の関係は検査される

文

文と文の型の関係は検査されない

型推論

- 明記されてない型を推論する
- 型の整合性を検査する
  - 外側から推測される型
  - 内側から推測される型

```
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

- あらゆる部分が検査される

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, null, nil, None

Haskell にはこれらがない

型安全

Haskell では
コンパイルが通れば
型に関する誤りがない

Haskell のコンパイルはテスト

純粋な関数での網羅率は 100 %

副作用のある関数でも高い網羅率

Haskell では
コンパイルが通れば
だいたい思い通りに動く

Haskell では保守が容易

- 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

書き換えるには
同じ形のピースが必要

Haskell でのテスト

純粋な関数

状態がないのでテストが容易
性質テストも可能

副作用のある関数

他の言語と同じ
HUnit, Hspec

性質テスト

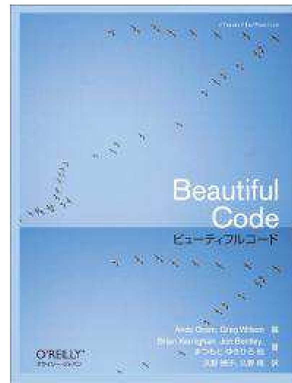
- Haskell では QuickCheck が有名
- 関数の性質を記述する

```
prop_doubleSort :: [Int] -> Bool
prop_doubleSort xs = sort xs == sort (sort xs)
```

- テストケースを乱数で生成してくれる

```
> quickCheck prop_doubleSort
+++ OK, passed 100 tests.
```

- 純粋な関数は性質を見つけやすい
  - 副作用のある関数は性質を見つけにくい



「ビューティフルコード」
7章 ビューティフル・テスト

二分探索に対するテスト

二分探索のアイディアは
1946年に出された

バグがない実装ができたのは12年後

美しきテストたち

- 線形探索を使いながら二分探索をテストする

スモークテスト	境界テスト
ランダムテスト	突然変異テスト

Haskeller にしてみれば

二分探索が線形探索と同じ

といっているだけ

QuickCheck による美しきテスト

- QuickCheck だと、仕様はモデル実装と同じと表現するだけ！

```
prop_model x xs =
  linearSearch x xs == binarySearch x xs
```

- 注意)加えてオーダーのテストも必要

豊かな型

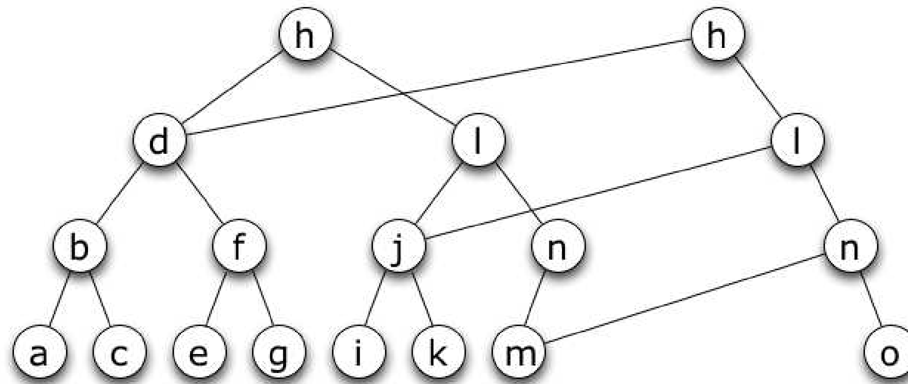
代数データ型

- 直和型
  - 複数の型に対して、どれか一つの型の値になる
  - または
  - C の union
  - `data Maybe a = Nothing | Just a`
- 直積型
  - 複数の型の値を同時に格納する
  - かつ
  - C の struct、オブジェクト
  - `data TLV a = TLV Tag Int a`
- 代数データ型
  - 直積型の直和型
  - `data Tree a = Leaf | Node (Tree a) a (Tree a)`

破壊的代入がないのに
どうやって木を操作するのか？

木に要素を加える

- 探索パスのノードを新たに作る
  - 10,000ノードの木だと約13ノードを新たに作ればよい



- 使われなくなったノードは GC される

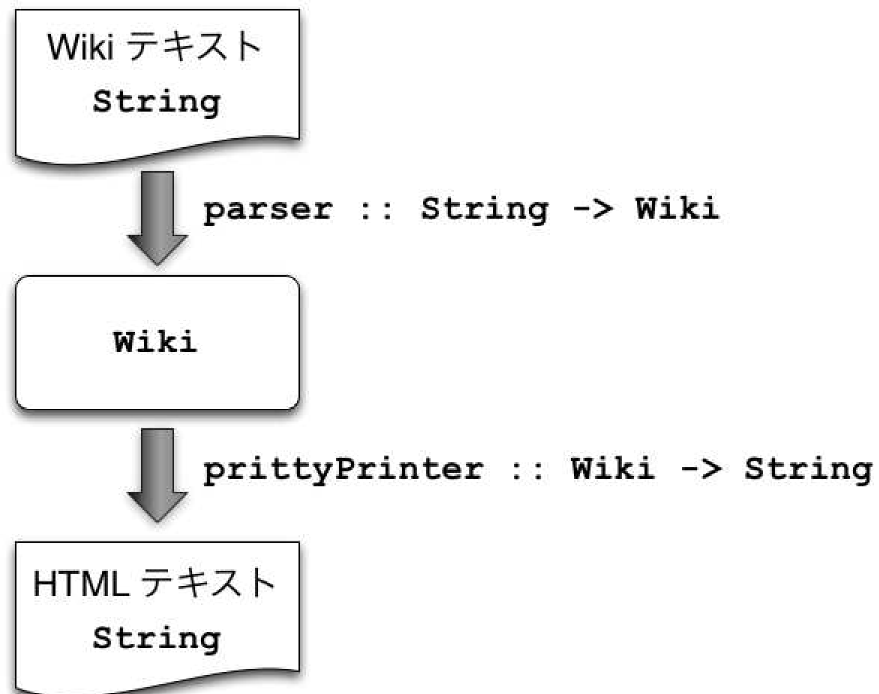
例) Wiki を作る

```
* 大見出し

行頭にキー文字がなければ段落

=============== 罫線
** 中見出し

\* で始まる段落

リンクは [タイトル URL] と書く

-箇条書き レベル1
--箇条書き レベル2
--箇条書き レベル2
-箇条書き レベル1
++番号付き箇条書き レベル2
++番号付き箇条書き レベル2
```

Wiki のデータ構造

```
data Wiki = HR
          | H Int HText
          | P HText
          | UOL Xlist

type HText = [PText]

data PText = Raw Char
           | Escaped Char  -- backslash
           | Anchor Title URL

type URL = String

type Title = String

data Xlist = Ulist [Xitem]
           | Olist [Xitem]
           | Empty

data Xitem = Item HText Xlist
```

豊かな型に対してプログラミングする



まとめ

- 関数プログラミング = 式で構成するプログラミング
- まぜるな危険！
- こんにちは 直和型、さようなら NULL
- コンパイルはテスト
- 性質テストでコーナーケースを探す
- 豊かな型に対してプログラミングする