関数プログラミングの道しるべ

2011.11.11

山本和彦

自己紹介

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

お品書き

- 関数プログラミング
- 永続データプログラミング
- リスト
- 再帰
- MapReduce
- 文字列
- 木
- コンビネータ

「神へ参るこそ本意なれと思ひて、
山までは見ず」にならないように！
今日は僕が「先達」だよ。

☆ 関数プログラミング

よくある偏見

Functional Programming

関数プログラミング

Functional

↓

実用的

Functional Programming

↓

実用的プログラミング

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。

関数型の手法とは？

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


関数型の手法とは
永続データを使った
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

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

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

func([10,20,30,40,50]);
→ 400
```

- 関数プログラマーには仕事のやり過ぎに見える

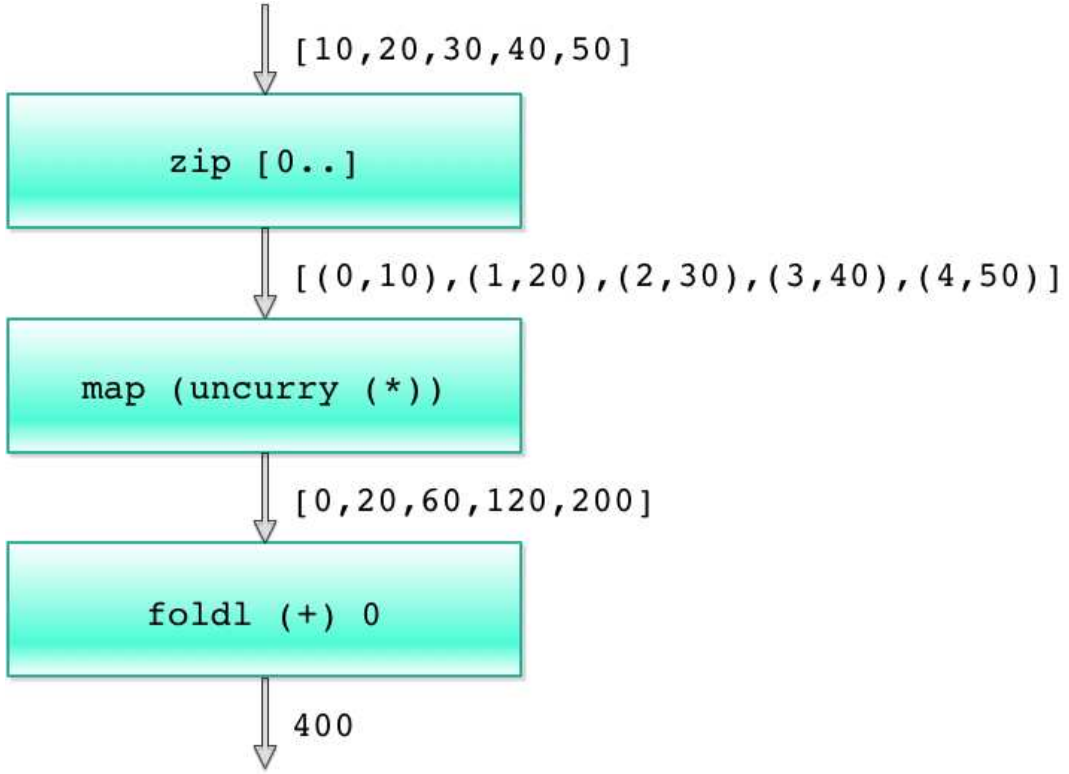
Haskell で MapReduce

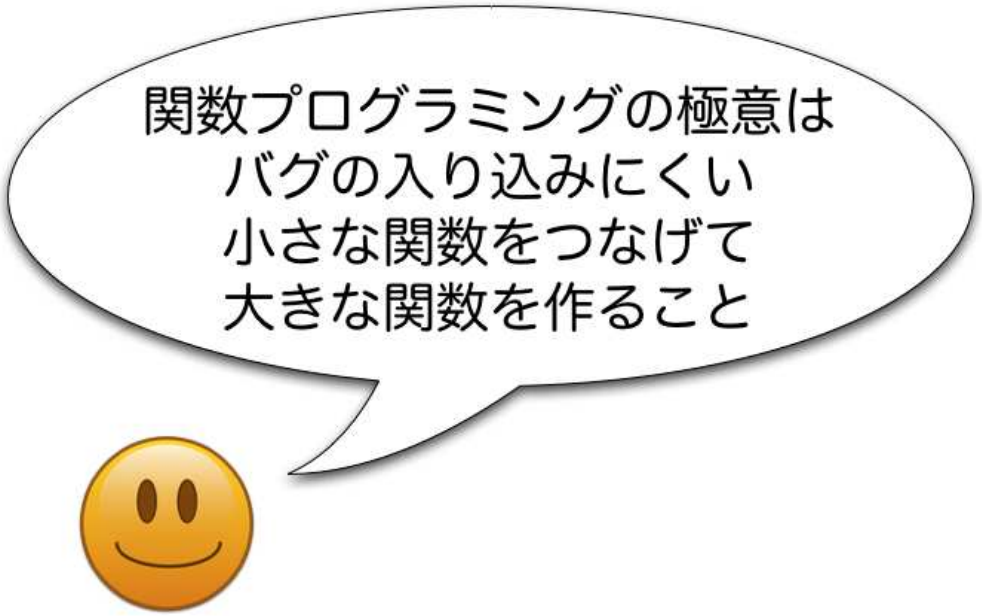
```
zip [0..] [10,20,30,40,50]
→ [(0,10),(1,20),(2,30),(3,40),(4,50)]

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

foldl (+) 0 上の式
→ ((((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は
バグの入り込みにくい
小さな関数をつなげて
大きな関数を作ること

☆リスト

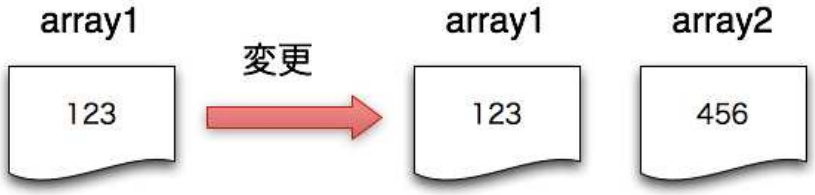
警告：このセクションは退屈です

リストの三種の神器

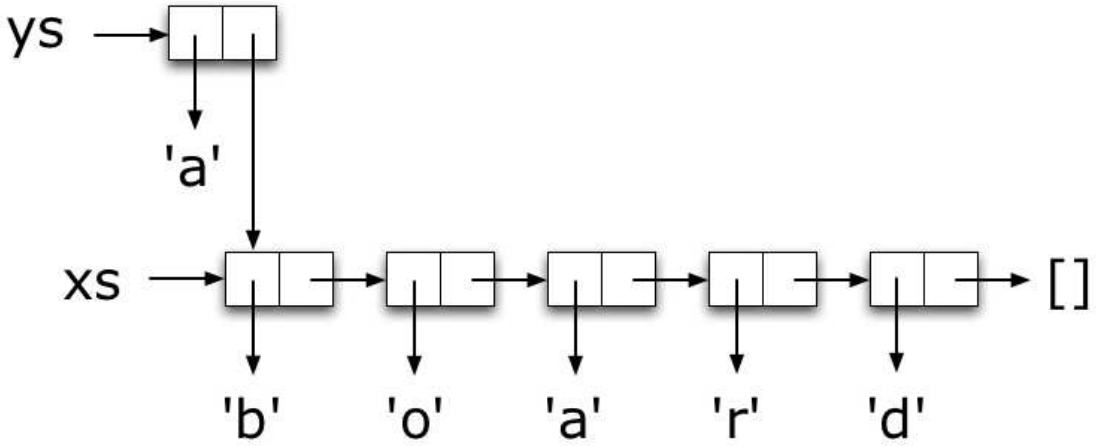
- head (car, first)
  - リストの先頭の要素を取り出す
- tail (cdr, rest)
  - 先頭のリストを除いた残りのリストの取り出す
- : (cons)
  - リストの先頭に要素を加える

永続データとしてのリスト

- 命令型言語のように配列を使うと悲惨
  - 配列の変更は、配列全体のコピーを意味する



- リストの先頭に要素を追加しても、元のリストは破壊されない



リストの表記

- 略記 -- `[1,2,3]`
- 正式 -- `1 : 2 : 3 : []`
- : (cons) は右結合 -- `1 : (2 : (3 : []))`

- : の利用例

```
'a' : ['b','o','a','r','d']
→ ['a','b','o','a','r','d']
```

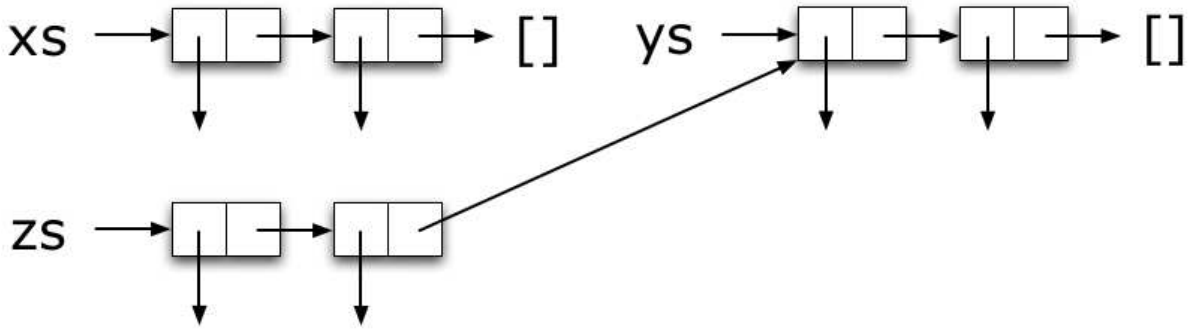
- パターンマッチがあれば、
  head や tail はあまり使わない

```
sum []     = 0
sum (x:xs) = x + sum xs
```

リストの連結(1)

- ++ (append) の左のリストはコピーされる

```
zs = xs ++ ys
```



- ++ の原則1) なるべく使うな

リストの連結(2)

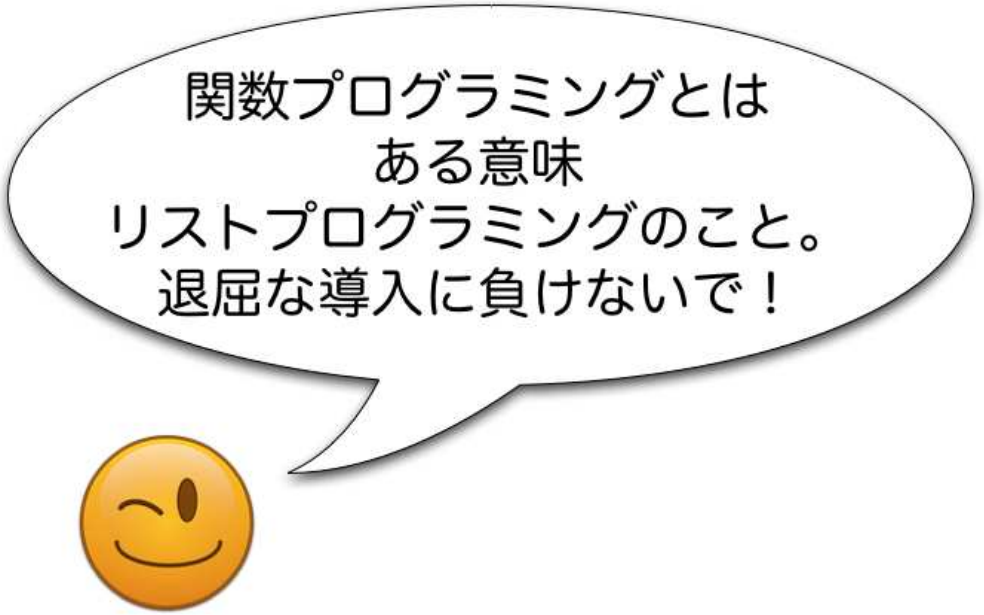
- ++ は右結合で使うと O(N)

```
xs ++ (ys ++ zs)
```

- ++ は左結合で使うと O(N^2)

```
(xs ++ ys) ++ zs
```

- ++ の原則2) 使うなら右結合で使え


関数プログラミングとは
ある意味
リストプログラミングのこと。
退屈な導入に負けないで！

☆ 再帰

再帰の例

- 階乗の漸化式
  factorial(0) = 1
  factorial(n) = n * factorial(n - 1)

- 基底部で終了する
- 再帰部で仕事をする

```
factorial :: Int -> Int
-- 基底部
factorial 0 = 1
-- 再帰部
factorial n = n * factorial (n-1)
```

再帰のこころ

- 一つ前ができていたら、次はどうする？
  - (C) nobsun

1) `factorial n` で n! を表す
2) `factorial (n - 1)` まで、できているとする
3) 次はどうする？
   → それに `n` をかければよいはず

```
factorial n = n * factorial (n-1)
```

型検査と再帰

- 再帰は式
  - 部分も式

```
length :: [Int] -> Int
length []     = 0
length (x:xs) = 1 + length xs
```

- 静的型付けであれば型検査の恩恵を受ける
  - コンパイル時にたくさんの間違いを発見できる

二種類の再帰

- リストを生成する関数
  - (末尾でない)再帰を使う
- 数値を生成する関数
  - 末尾再帰を使う

リストを生成する再帰

- (末尾でない)再帰
  - 分岐の末端で、一番外の関数が自分自身ではない再帰
  - Haskell では = の直後に自分自身がない再帰

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)

{- あるいは
 (++) (x:xs) ys = (:) x (xs ++ ys)
-}
```

- これは遅延評価と相性がいい
  - 正格評価を採用している言語では delay と force を使う

数値を生成する再帰

- (末尾でない)再帰で書いてみる

```
sum :: Num a => [a] -> a
sum []     = 0
sum (x:xs) = x + sum xs

{- あるいは
sum (x:xs) = (+) x (sum xs)
-}
```

- スタックが溢れる

```
  sum [1,2,3,4]
= 1 + sum [2,3,4]
= 1 + (2 + sum [3,4])
= 1 + (2 + (3 + sum [4]))
= 1 + (2 + (3 + (4 + sum [])))
= 1 + (2 + (3 + (4 + 0)))
```

末尾再帰

- 末尾再帰で書き直す
  - 分岐の末端で、一番外の関数が自分自身である再帰
  - 引数に蓄積変数を追加するのが定石

```
sum :: Num a => [a] -> a
sum xs = sum' 0 xs
  where
    sum' r []     = r
    sum' r (y:ys) = sum' (y+r) ys
```

- スタックは溢れない

```
   sum'  0 [1,2,3,4]
= sum'  1 [2,3,4]
= sum'  3 [3,4]
= sum'  6 [4]
= sum' 10 []
= 10
```


再帰的に考えろ！
そうすれば
型システムのご加護がある！

☆ MapReduce

二つの畳み込み

- 右からの畳み込み

```
[1,2,3,4]
1 + (2 + (3 + (4 + 0)))
```

- 左からの畳み込み

```
[1,2,3,4]
(((0 + 1) + 2) + 3) + 4
```

右からの畳み込み

- foldr の実装

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _  ini []     = ini
foldr op ini (x:xs) = op x (foldr op ini xs)
```

- これは(末尾でない)再帰
  - リストを生成する関数と相性がよい

- ++ を foldr で実装する

```
(++) :: [a] -> [a] -> [a]
xs ++ ys = foldr (:) ys xs
```

左からの畳み込み

- foldl の実装

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl _  acc []     = acc
foldl op acc (x:xs) = foldl op (op acc x) xs
```

- これは末尾再帰
  - 数値を生成する関数と相性がよい
- sum を foldl で実装する

```
sum :: Num a => [a] -> a
sum xs = foldl (+) 0 xs
```

高階関数 map

- 再帰版

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

- foldr 版

```
map :: (a -> b) -> [a] -> [b]
map f xs =
  foldr (\y ys -> f y:ys) [] xs
```

高階関数 filter

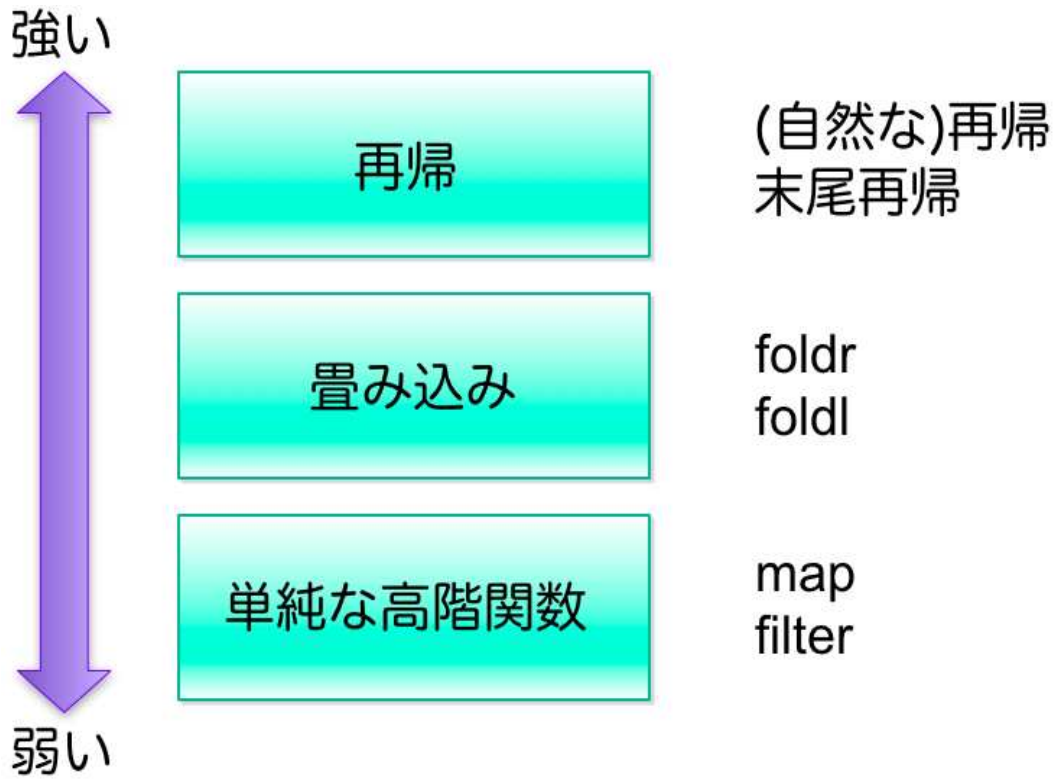
- 再帰版

```
filter :: (a -> Bool) -> [a] -> [a]
filter _    [] = []
filter prd (x:xs)
  | prd x       = x : filter prd xs
  | otherwise   = filter pred xs
```

- foldr 版

```
filter :: (a -> Bool) -> [a] -> [a]
filter prd xs = foldr filter' [] xs
  where
    filter' y ys
      | prd y     = y : ys
      | otherwise = ys
```

力の階層と節度



なるべく力の弱いものを使え！

どちらが分かりやすい？

- 再帰で書く

```
sumEven :: [Int] -> Int
sumEven xs = sumEven' xs 0
  where
    sumEven' []     r = r
    sumEven' (y:ys) r
      | even y        = sumEven' ys (y + r)
      | otherwise     = sumEven' ys r
```

- MapReduce で節度を守って書く

```
sumEven :: [Int] -> Int
sumEven xs = foldl (+) 0 (filter even xs)

{- あるいは
sumEven = foldl (+) 0 . filter even
-}
```


(再帰で）考えるな！
（MapReduce を）感じろ！

☆ 文字列

文字列プログラミング

- 「文字列は文字のリスト」であるとする
  - `"String" = ['S','t','r','i','n','g']`
  - 文字列が文字のリストでなくても考え方は同じ
- お題
  - メールのヘッダを解析する
  - 最長重複文字列を探す

メールのヘッダを解析する

- フィールドのキーと値を取り出す

```
"Subject: Hello world "

key = "Subject"
val = "Hello world"
```

JavaScript でメールのヘッダを解析する

- きっと正規表現を使う

```
var target = "Subject: Hello world ";
var x = target.match(/^(\w+):\s*(.*)/);
var key = x[1];
var val = x[2].replace(/\s*$/,"");
```

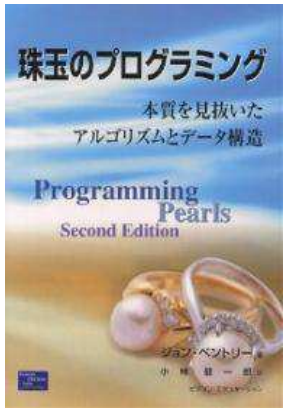
関数的にメールのヘッダを解析する

```
target = "Subject: Hello world "

break (==':') target
→ ("Subject", ": Hello world ")

tail (snd 上の式)
→ " Hello world "

dropWhile isSpace 上の式
→ "Hello world "

dropWhileEnd isSpace 上の式
→ "Hello world"
```

利用した高階関数

```
break :: (a -> Bool) -> [a] -> ([a],[a])
break _ []    = ([],[])
break p (x:xs)
  | p x       = ([],x:xs)
  | otherwise = (x:ys,zs)
  where
    (ys,zs) = break p xs

dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile _ [] = []
dropWhile p (x:xs)
  | p x         = dropWhile p xs
  | otherwise  = x:xs

dropWhileEnd :: (a -> Bool) -> [a] -> [a]
dropWhileEnd p = foldr op []
  where
    op x xs
      | p x && null xs = []
      | otherwise      = x:xs
```

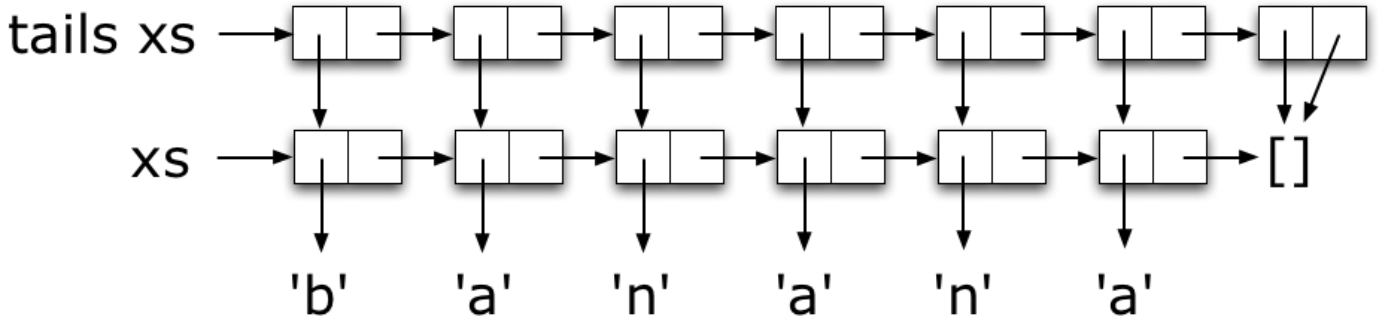
最長重複文字列



Ask not what your country can do for you,
but what you can do for your country.

最長重複文字列は "can do for you"

サフィックス(接尾辞)リスト



- tails の利用例

```
tails "banana"
→ ["banana","anana","nana","ana","na","a",""]
```

- tails の定義

```
tails :: [a] -> [[a]]
tails [] = [[]]
tails xs = xs : tails (tail xs)
```

最長重複文字列を求める(1)

```
tails "mississippi"
→["mississippi","ississippi",
   "ssissippi","sissippi",...

sort 上の式
→["","i","ippi","issippi",
   "ississippi","mississippi","pi",...

zip 上の式 (tail 上の式)
→[("","i"),("i","ippi"),
   ("ippi","issippi"),
   ("issippi","ississippi"),...
```

最長重複文字列を求める(2)

- 先頭からの共通部分の長さを求める

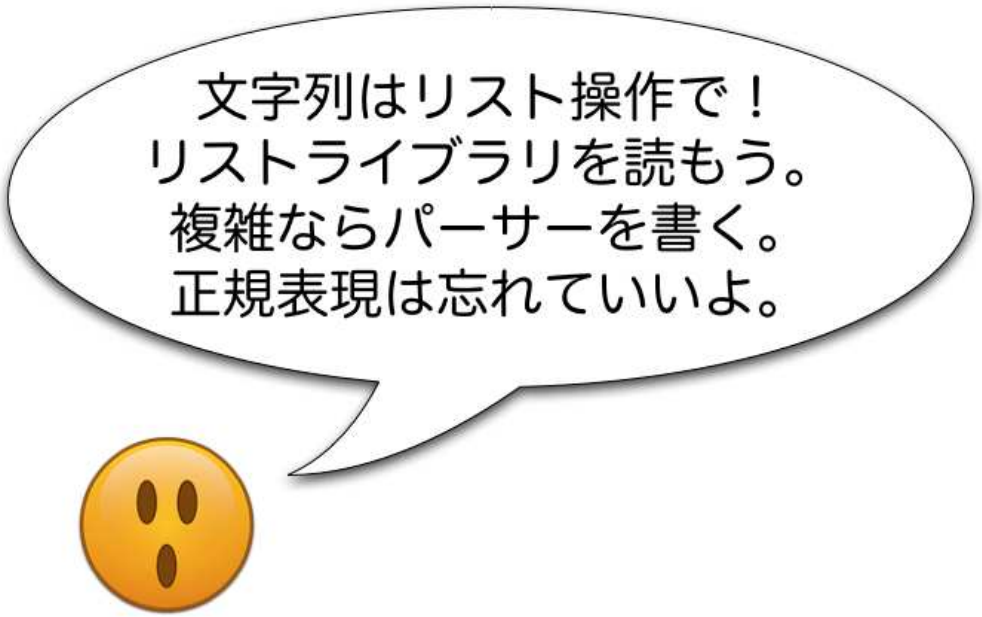
```
comlen :: Eq a => [a] -> [a] -> Int
comlen as bs = comlen' as bs 0
  where
    comlen' (x:xs) (y:ys) n
      | x == y    = comlen' xs ys (n+1)
    comlen' _ _ n = n
```

- 長さと文字列の組へ

```
lenstr :: Eq a => ([a], [a]) -> (Int,[a])
lenstr (xs,ys) = (len,xs)
  where
    len = comlen xs ys
```

最長重複文字列を求める(3)

```
→[("","i"),("i","ippi"),
   ("ippi","issippi"),
   ("issippi","ississippi"),...

map lenstr 上の式
→[(0,""),(1,"i"),(1,"ippi"),
   (4,"issippi"),(0,"ississippi"),...

maximumBy (comparing fst) 上の式
→(4,"issippi")

take (fst 上の式) (snd 上の式)
→"issi"
```

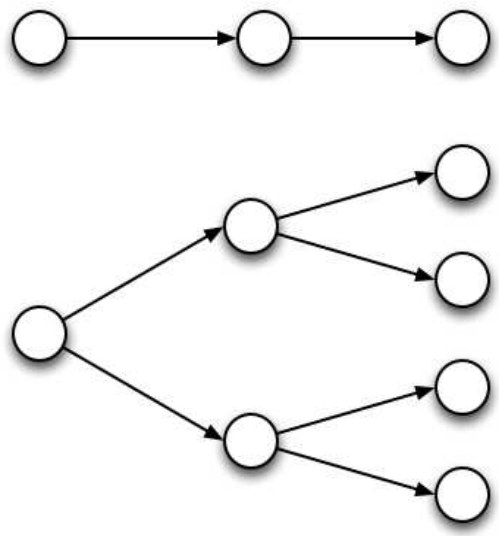

文字列はリスト操作で！
リストライブラリを読もう。
複雑ならパーサーを書く。
正規表現は忘れていいよ。

☆木

リストと木

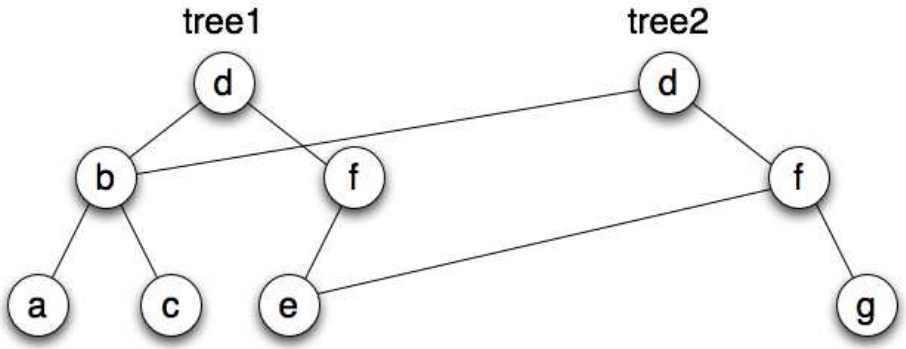
```
data List a =
    Nil | Cons a (List a)
data Tree a =
    Tip | Bin a (Tree a) (Tree a)
```

- 一分木か二分木か



永続データとしての木

- 木では更新時に多くの部分を共有できる
- 例えば要素を挿入する
  - 10,000ノードの木だと約13ノードを新たに作ればよい



走査(1)

```
preorder :: Tree a -> [a]
preorder Tip         = []
preorder (Bin x l r) =
    [x] ++ preorder l ++ preorder r

inorder :: Tree a -> [a]
inorder Tip         = []
inorder (Bin x l r) =
    inorder l ++ [x] ++ inorder r

postorder :: Tree a -> [a]
postorder Tip         = []
postorder (Bin x l r) =
    postorder l ++ postorder r ++ [x]
```

走査(2)

```
preorder :: Tree a -> [a]
preorder t = go t []
  where
    go Tip xs         = xs
    go (Bin x l r) xs = x : (go l (go r xs))

inorder :: Tree a -> [a]
inorder t = go t []
  where
    go Tip xs         = xs
    go (Bin x l r) xs = go l (x : (go r xs))

postorder :: Tree a -> [a]
postorder t = go t []
  where
    go Tip xs         = xs
    go (Bin x l r) xs = go l (go r (x : xs))
```

その他の操作

- 挿入、削除
- 分割、マージ
- 回転し平衡化する

- 自分で勉強して下さい！

木の用途

探索木
O(log N) のハッシュテーブル

集合演算

キュー

優先順序付きキュー

両端キュー

巨大な文字列操作

永続データプログラミングの弱点

- 配列と相性が悪い
- 更新や検索が O(1) でできるハッシュテーブルがない
  - 辞書、連想配列、Finite Map
- 短命データを使う
  - 命令的な解決方法
- 融合変換、リサイクルを使う
  - 関数的な解決方法 (制限あり)
  - 中間の不要なデータをなくす
  - 「簡約 λ カ娘」が秀逸


リストだけに頼ってはダメ。
コンテナライブラリを読もう。

☆ コンビネータ

コンビネータとは何か？

- コンビネータ = 素敵な内部DSL
- 問題を表すデータ型を定義する

```
data Parser a =
    Parser (String -> (Maybe a, String))
```

- そのデータを返す小さな関数を書く

```
char :: Char -> Parser Char
char c = ...
```

- それらを組み合わせる関数を書く

```
(<|>) :: Parser a -> Parser a -> Parser a
p1 <|> p2 = ...
```

- 狭義には組み合わせる関数を
  広義にはこれら全体をコンビネータと呼ぶ

パーサーコンビネータ

- JSON の BNF (RFC 4627)

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

- パーサーコンビネータで 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(コツは必要)
  - パーサーコンビネータを使って書くのは
    その言語のコードそのもの
  - 型システムのご加護がある

さまざまなコンビネータ

パーサー

プリティプリンタ

SQL, XML, HTML

Functional
Reactive
Programming

ハードウェア記述

金融商品記述
(デリバティブ記述)


本殿へようこそ！

もっと勉強するために

- 書籍
  - 7つの言語 7つの世界
    - Scala、Erlang、Clojure、Haskell
  - Scheme 手習い
  - プログラミングHaskell
  - Scala スケーラブルプログラミング
  - プログラミングの基礎
    - OCaml、浅井健一著
  - Purely Functional Data Structure
- オンライン記事
  - 右も左も分かる再帰
  - Haskellの文法(再帰編)
  - なぜ関数プログラミングは重要か