すごいHaskellたのしく学ぼう! 読書会 #4

第5章 高階関数
第6章 モジュール

おがさわらなるひこ

第5章

高階関数

衝撃の事実!

- 今までの関数はすべて**一引数の関数**だった!
  - replicate 5 'a' とかも実はそう
  - どゆこと?

カリー化関数

- ただひとつの引数を取れる関数
- 一つの引数を受け取って、残りの引数を受け取る「関数」を返す
- ???

max関数にお出まし願おう

- max x y ← x と y のうち大きい方を返す

```
Prelude> max 4 5
5
```

- でもこれは実際はこう読むんだな

```
Prelude> (max 4) 5
5
```

  - 「max関数に4を適用した関数」に5を適用する
  - 型シグニチャ的には:
    max :: (Ord a) => a -> (a -> a)
  - Ord型クラスのインスタンス型aがあるとして、
    aを取って、「aを取ってaを返す関数」を返す関数

部分適用！

- さっきの「max関数に4を適用」して、新たな関数を作り出すことを「部分適用」と呼ぶのよ
- 関数をお手軽に作り出して、それを別の関数に渡すことができる

```
$ cat multThree.hs
multThree :: Int -> Int -> Int -> Int
multThree x y z = x * y * z
$ ghci
…
Prelude> :l multThree
[1 of 1] Compiling Main             ( multThree.hs, interpreted )
Ok, modules loaded: Main.
*Main> multThree 3 5 9
135
*Main> let multTwoWithNine = multThree 9
*Main> multTwoWithNine 2 3
54
```

部分適用した関数を変数に代入して再利用できる

セクション

- 中置関数の部分適用
- 演算子と括弧で囲う

```
$ cat divideByTen.hs
divideByTen :: (Floating a) => a -> a
divideByTen = (/10)
$ ghci
...
Prelude> :l divideByTen
[1 of 1] Compiling Main             ( divideByTen.hs, interpreted )
Ok, modules loaded: Main.
*Main> divideByTen 200
20.0
*Main> (/10) 200
20.0
```

部分適用した関数をGHCiで表示

- たとえばこうやると……

```
*Main> multThree 3 5

<interactive>:7:1:
    No instance for (Show (Int -> Int)) arising from a use of `print'
    Possible fix: add an instance declaration for (Show (Int -> Int))
    In a stmt of an interactive GHCi command: print it
```

- あれ？ 例外だ。
- なんで？ というのは、宿題①で。

ちょっとオフトピ：Scalaと部分適用

- 部分適用というのはHaskellに特別なものじゃなくて例えばScalaにもある

```
scala> def sum(a: Int, b: Int, c: Int) = a + b + c
sum: (a: Int, b: Int, c: Int)Int

scala> val x = sum _
x: (Int, Int, Int) => Int = <function3>

scala> x(1, 2, 3)
res0: Int = 6

scala> val y = sum(1, _: Int, 3)
y: Int => Int = <function1>

scala> y(10)
res2: Int = 14
```

- けど「書けば書ける」という感は否めない
  - Haskell（MLもそうらしい）は部分適用をごく普通に使うんだけど、Scalaの場合はJavaみたいな構文を使う関係上しゃーないってコップ本にあった

高階実演

- こんな関数を考えてみよう

```
applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)
```

  - '->' は右結合なので今までの使い方では括弧を省略できたけど、今回は括弧が必要

- ではいろいろ渡してみよう!

```
*Main> applyTwice (+3) 10
16
*Main> applyTwice (++ " HAHA") "HEY"
"HEY HAHA HAHA"
*Main> applyTwice ("HAHA " ++) "HEY"
"HAHA HAHA HEY"
*Main> applyTwice (multThree 2 2) 9
144
*Main> applyTwice (3:) [1]
[3,3,1]
```

カリー化関数超便利!

- こんなふうに「一引数の関数」を渡すときにささっと作って渡せる
- 関数の再利用がたやすくできる
- 関数合成

zipWithの実装

- こんな感じ

```
zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' _ [] _ = []
zipWith' _ _ [] = []
zipWith' f (x:xs) (y:ys) = f x y : zipWith' f xs ys
```

- aとbを引数の型としてc型を返す関数を一つとa、bのリストを取り、c型のリストを返す関数
  - わかんなくなったらGHCiの :t で型をチェック
- すっかり書き忘れていたけど「関数を引数にとる関数」のことを高階関数（high order function）っていうんだよ

flip!

- 二つの引数を持つ関数を引数に取り、その二つの引数をひっくり返す関数
- 型は簡単で flip' :: (a -> b -> c) -> (b -> a -> c)
- 実装部は:

```
flip' f = g
    where g x y = f y x
```

- あれ、f = g なんだったら別に g いらなくね?

```
flip' f y x = f x y
```

- flip それ自体が部分適用のときに超便利

関数プログラマの道具箱 mapとfilter

- 知らなきゃモグリ
  - map: 関数をリストの全要素に適用
  - filter: booleanの関数（＝述語）をリストの全要素に適用してTrueになるものだけを残す
  - どっちもScalaにもRubyにもあるよ
- 前回のquicksortをfilterで書き直すとこんな感じ

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
    let smallerOrEqual = filter (<= x) xs
        larger = filter (> x) xs
    in  quicksort smallerOrEqual ++ [x] ++ quicksort larger
```

関数プログラマの道具箱 mapとfilter（活用編）

- 10万以下から3829で割り切れる数を探す

```
largeDivisible :: Integer
largeDivisible = head (filter p [100000,99999..])
    where p x = x `mod` 3829 == 0
```

  - 10万以下のリストを作っておいてfilterしてhead
  - 一見無駄にリスト生成して遅そうだけど、Haskellは怠け者（遅延評価）なので実際は答えが見つかるまでしか評価されない
- 10000 より小さいすべての奇数の平方数の和
  - takewhile使えば一撃

無限リストなのに注目！

```
*Main> sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
166650
```

関数プログラマの道具箱 mapとfilter（活用編②）

- コラッツ数で遊ぶ
  - 任意の自然数から開始
  - 1なら、終了 / 偶数なら、2で割る / 奇数なら、3倍して1を足す
  - 終わりまで繰り返し
- 1~100までの数のウチ、コラッツ数の「長さ」（＝収束するまでに経由する数）が15以上になるものはいくつある？
  - 練習問題②へ

関数プログラマの道具箱 mapとfilter（活用編③）

- mapに複数の引数を取る関数を与えたらどうなるのかな？

```
Prelude> let listOfFuns = map (*) [0..]
Prelude> (listOfFuns !! 4) 5
20
```

- mapの引数に渡されたリストの各要素に関数が部分適用されたリストが作られる
  - ので、そこから要素を取り出して残りの引数を与えてやると結果が得られるというわけ

ラムダ!

- ラムダ式とは使い捨て関数を作る**無名関数**
  - 名前の源はChurchのlambda演算だけど、実際のところあんま関係ないよね
  - モダンな言語にはだいたい存在する
  - Javaにも8から入ります
- whereの置き換えなんかに使えるよ

```
largeDivisible :: Integer
largeDivisible = head (filter p [100000,99999..])
    where p x = x `mod` 3829 == 0
```

```
largeDivisible' :: Integer
largeDivisible' = head (filter (\x -> x `mod` 3829 == 0)
                               [100000,99999..])
```

ラムダ続き

- ラムダ式は複数の引数を取れるよ!

```
Prelude> zipWith ( \ a b -> (a * 30 + 3) / b) [5,4,3,2,1] [1,2,3,4,5]
[153.0,61.5,31.0,15.75,6.6]
```

- パターンマッチもできちゃうよん

```
Prelude> map ( \ (a,b) -> a + b) [(1,2),(3,5),(6,3),(2,6),(2,5)]
[3,8,9,8,7]
```

  - ただし複数のパターンを持つようなケースはダメ

- 単なるカリー化で済むところをラムダ式を使うのは記法が煩雑になるだけだが、「ある値は関数だ」と明示したいときには使ってもいい

```
flip' (a -> b -> c) -> (b -> a -> c)
flip' f = \x y -> f y x
```

再帰と畳込み(fold)

- 再帰のよくあるパターン
  - 基底部が存在する
  - （多くの場合）リストの先頭要素に対する処理をし、残りを再帰で処理したものと合成する
- このパターンをHaskellではfold（畳み込み）として関数化してある
  - Scalaにもあるよね
- 引数「2引数の関数」と「アキュムレータ（累算器）」「畳込み対象のリスト」を取る

あなたの畳み込み、左から？それとも右から？

- foldl = 左畳み込み（fold left）
  - 型チェック

```
Prelude> :t foldl
foldl :: (a -> b -> a) -> a -> [b] -> a
```

- 試しに sum' を実装しよう
  - 再帰版

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

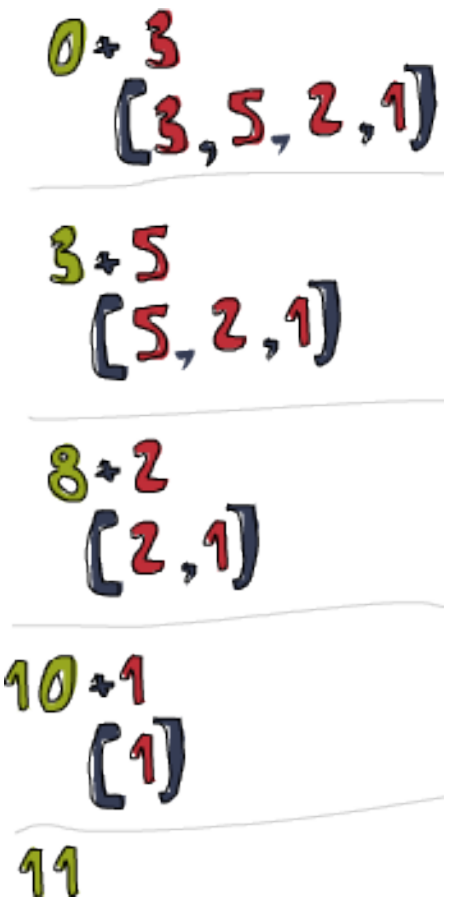
  - foldl版

accumulatorの意味

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

foldlの動きを見てみる

- アキュムレータに対して左からラムダ式で渡した関数が適用されてどんどん値が足されていくのです



お次は右畳み込み

- 関数foldr（fold right）=畳み込みの方向が違う以外はfoldlとほぼ同じ
  - なんで二つ分けてあるの?という話はこの後で
- 例えば map を実装してみる
  - 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 (\x acc -> f x : acc) [] xs
```

なんで右左どっちもあるんだよ!めんどくさいな!

- 例えばmap''はこうやって実装することもできなくはない……んだけど。

```
map'' :: (a -> b) -> [a] -> [b]
map'' f xs = foldl (\acc x -> acc ++ [f x]) [] xs
```

  - 右から結果を得た場合、consが使えずappendしなければいけない
  - appendが遅いのでこれを避けたい
  - 一方、sum' の場合は + が可換なので、与えられたリスト xs の走査が早いほうがいい
- 問題:elem関数をfold(どっちでも可)で実装しよう!

foldl1とfoldr1

- foldlとfoldrに似ている
- 違いは、アキュムレータの初期値がいらない
- 端っこの値を初期値とみなす
- 例えばさっきのsum'はこんなふうに実装可能

```
sum'' :: (Num a) => [a] -> a
sum'' xs = foldl1 (\acc x -> acc + x) xs
```

  - よいこのみなさんは気づいてると思うけどfoldl1 / foldr1 は空リストだと動かないから注意な!

scanl と scanr

- foldl / foldr とそっくし
- 違いは畳み込まないで、全部の途中結果をリストで返すこと

```
Prelude> scanl (+) 0 [5,3,1,8]
[0,5,8,9,17]
Prelude> scanr (+) 0 [5,3,1,8]
[17,12,9,8,0]
Prelude> scanl1 (+) [5,3,1,8]
[5,8,9,17]
```

- 途中結果を全部欲しいときにつかいます

$

- 関数適用演算子
  - へ？ 関数適用って普通にスペースでええのんちゃうの？
- 違い
  - スペースによる関数適用は非常に高い演算子優先順位を持つが、$ は最低の優先順位
  - スペースは左結合 (f g h x = (((f g) h) x) だけど $ は右結合
  - 括弧の節約になるよ!

```
sum (map sqrt [1..130]) ===> sum $ map sqrt [1..130]
sqrt (3 + 4 + 9) ===> sqrt $ 3 + 4 + 9
sum (filter (> 10) (map (*2) [2..10]))
                            ===> sum $ filter (> 10) (map (*2) [2..10])
                            ===> sum $ filter (> 10) $ map (*2) [2..10]
```

関数合成

- 名前の通り関数を合成すること

$$(f \circ g)(x) = f(g(x))$$

- Haskellだと “.” を使う
  - 定義こんなの

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \ x -> f (g x)
```

  - 使い方こんな感じ

```
Prelude> (negate . (*3)) 10
-30
```

関数合成で関数即席作成

- ある適当な整数のリストを全部負にしたい
  - ぱっと思いつくのは絶対値を取ってマイナスにする
  - もちろん \x -> negate (abs x) とやってもいい
  - けどこっちのほうがずっと簡単

```
Prelude> map (negate . abs) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]
```

さらに関数合成

- 関数合成自体も演算で、結合則を満たす
  - 右結合
- なので以下のコードは:

```
Prelude> map ( \ xs -> negate (sum (tail xs))) [[1..5],[3..6],[1..7]]
[-14,-15,-27]
```

以下のように変形できちゃうのだよっと

```
Prelude> map (negate . sum . tail) [[1..5],[3..6],[1..7]]
[-14,-15,-27]
```

多引数関数の関数合成

- “.” 演算子の定義から分かるように引数が一個な関数しか合成できません
- ので、部分適用によって一引数な関数にしましょう

複雑な式を変形してみよう

```
replicate 2 (product (map (*3) (zipWith max [1,2] [4,5])))
```

- 常に一番内側 (この場合は zipwith max [1,2] [4,5]）に注目
- それが適用されている関数（map *3）に着目
  - これを $ 演算子で簡約

```
replicate 2 (product (map (*3) $ zipWith max [1,2] [4,5]))
```

- 今簡約した式に適用されてる関数はproduct
  - 関数合成で簡約可能

```
replicate 2 (product . map (*3)) $ zipWith max [1,2] [4,5]
```

- 同様に展開してこうなる!

```
replicate 2 . product . map (*3) $ zipWith max [1,2] [4,5]
```

ポイントフリースタイル

- たびたび登場sum'

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

  - この右辺値と左辺値には両方引数xsがある
  - 関数はカリー化されているのでxsは省略可能

```
sum' :: (Num a) => [a] -> a
sum' = foldl (\acc x -> acc + x) 0
```

- このような、引数としての一次変数（＝ポイント）をないかたちで関数宣言することを**ポイントフリースタイル**と呼ぶ
  - ポイントフリースタイルの関数定義に関数合成が役立つ

ポイントフリースタイルと関数合成でハッピー♪

- 次のような関数定義があったして:

```
fn x = ceiling (negate (tan (cos (max 50 x))))
```

- 関数合成＆ポイントフリーでこんな感じに♪

```
fn = ceiling . negate . tan . cos . max 50
```

- いかにも「合成した関数を fn に代入して新たに関数を作っている」感がして素敵♪
  - でもやり過ぎには注意
  - 適当な単位で let を使ってまとめましょう

もうひとつ関数合成の例

- さっき出てきた「10000より小さい……」の例

```
oddSquareSum :: Integer
oddSquareSum = sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
```

- これを関数合成と $ を使うとこんな感じに

```
oddSquareSum :: Integer
oddSquareSum = sum . takeWhile (<10000) . filter odd $ map (^2) [1..]
```

- 読み慣れないと??
  - $ の右辺 map (^2) [1..] を先に読んで
  - それを filter odd して
  - takeWhile (<10000) して
  - 最後に sum する、と読めるようになる

第6章

モジュール

Haskellのモジュールとは

- いくつかの**関数**や**型**、**型クラス**を定義したファイル
- **プログラム**はモジュールの集合
- モジュールは関数や型を任意に選んで**エクスポート**できる
  - つまりモジュールの外から利用可能にするってこと
- よい子は互いに強く依存しない（粗結合な）モジュールにプログラムをうまく分割しようね
- 今まで使ってきた標準の関数、型、型クラスは**Prelude**というモジュールの配下

```
Prelude> map (negate . abs) [5,-3,-6,7,-3,2,-19,24]
```

これ!

モジュールを使ってみる

- ファイルから呼び出す場合：import文
- とりま使ってみましょう

```
import Data.List

numUniques :: (Eq a) => [a] -> Int
numUniques = length . nub
```

  - Date.Listモジュールはリストを操作する便利な関数をたくさん用意したモジュール
  - nub はリストから重複を除去したリストを返す関数
- Hoogle（http://www.haskell.org/hoogle/）使うといいよ！

モジュールを使ってみる: GHCi編

- :m でモジュールを追加できる

```
Prelude> :m + Data.List
Prelude Data.List> nub [1,5,2,3,1,2,5]
[1,5,2,3]
Prelude Data.List> length . nub $ [1,5,2,3,1,2,5]
4
```

- 同時に何個もモジュールを追加可能

```
Prelude> :m Data.List Data.Map Data.Set
Prelude Data.List Data.Map Data.Set>
```

- 今のGHCiはimport文も使えるよ!

```
Prelude> import Data.List(nub)
Prelude Data.List> length . nub $ [1,5,2,3,1,2,5]
4
```

部分インポートと修飾インポート

- 何でもかんでもモジュールを突っ込むと名前がぶつかりまくってえらいことに
  - いろいろ方法あるよ!

やりたいこと	名称	構文
使いたい関数だけインポート	部分インポート	import Data.List (nub,sort)
使いたくない関数以外をインポート		import Data.List *hiding* (nub)
インポートするときに名前空間を分ける	修飾インポート	import *qualified* Data.Map (参照するときは Data.Map.filter)
インポートするときに名前空間を分け、別名をつける		import *qualified* Data.Map *as* M (参照するときは M.filter)

※関数合成の“.”とはスペースの有無で区別する

標準関数で遊ぼう
1. 単語を数える

文字列内にある単語を抽出し
その出現頻度をカウント

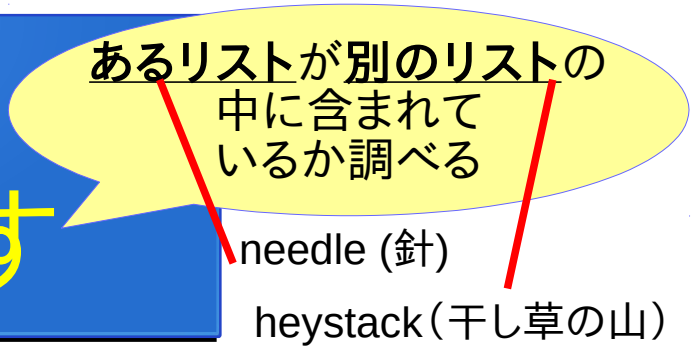
モジュール	関数	意味	
Data.List	words	文字列を単語で切る	> words "cat elephant dog" ["cat", "elephant", "dog"]
Data.List	group	同じ塊をグループ化	> group [1,1,1,2,2,2,2,3,3,3,4,4,5] [[1,1,1],[2,2,2,2],[3,3,3],[4,4],[5]] > group [1,1,2,2,1] [[1,1],[2,2],[1]] <-- イケてない
Data.List	sort	リストをソート	> sort [5,7,2,1,8] [1,2,5,7,8] > sort ["cat", "elephant", "dog"] ["cat", "dog", "elephant"]

- 組み合わせるとこんな感じ!

```
import Data.List

wordNums :: String -> [(String, Int)]
wordNums = map (\ws -> (heads ws, length ws)) . group . sort . words
```

標準関数で遊ぼう
2. 干し草の山から針を探す



モジュール	関数	意味	
Data.List	tails	あるリストにtailを繰り返し適用したリストを得る	> tails “party” [“party”, “arty”, “rty”, “ty”, “y”, “”]
Data.List	isPrefixOf	あるリストの先頭が別のリストで始まっているかどうか調べる	> "hawaii" `isPrefixOf` "hawaiian" True > "hawaii" `isPrefixOf` "hawa" False
Data.List	any	与えられたリストに述語を渡して、一個でもTrueならTrue	> any (>10) [1, 2, 3] False > any (== 'F') “Frank” True

- 組み合わせるとこんな感じ!

```
isIn :: [a] -> [a] -> Boolean
needle `isIn` heystack = any (needle `isPrefix`) (tails heystack)
```

  - でもこの関数、Data.Listに isInfixOf としてあるんだよねえ……

標準関数で遊ぼう
3. シーザー暗号

- シーザー暗号知ってるよね？
  - 各文字を一定の数だけシフトした暗号

モジュール	関数	意味	
Data.Char	ord	文字から文字コードへ	> ord 'a' 97
Data.Char	chr	文字コードから文字へ	> chr 97 'a'

- エンコードはこんな感じで

このラムダ式は関数合成
chr . (+offset) . ord に書き換え可

```
encode :: Int -> String -> String
encode offset msg = map (\c chr $ ord(c) + offset)
```

- デコードは足すのを引くように変えればOK

```
decode :: Int -> String -> String
decode offset msg = map (\c chr $ ord(c) - offset)
```

まだまだ標準モジュールは あるけど省略してfoldの話

- fold便利だよね！
  - でも時には困るときも……
  - 遅延評価なので、末尾再帰最適化が実際の評価が行われるまで行われず、スタックを食いつぶして死ぬ
    - foldl (+) 0 (replicate 100000000 1) してみたらGHCiがOOM Killerに殺られました ><;
- そんなあなたに Data.Listのfoldl' / foldr' ！
  - 遅延（lazy）でない正格（strict）評価なfold
    - 正格評価のことを「先行評価（eager）」といったりします

```
Prelude Data.List> foldl' (+) 0 (replicate 100000000 1)
100000000 <------ やった！
```

キーと値の紐付けする話（多分）

- いわゆる連想リスト（association list、A-list）
  - 実装はリストの中にダブル（＝二要素のタプル）でOK

```
phoneBook =
    [("betty", "555-2938")
    ,("bonnie", "452-2928")
    ,("patsy", "493-2928")
    ,("lucille", "205-2928")
    ,("wendy", "939-8282")
    ,("penny", "853-2492")
    ]
```

  - 検索はこんな感じで大丈夫

```
findKey :: (Eq k) => k -> [(k, v)] -> v
findKey key xs = snd . first . filter (\(k, v) -> k == key) $ xs
```

  - ……ん？ ホントか？

キーと値の紐付けする話（多分ね）

- さっきの式、キーに対応する値がなかったらどう？

```
findKey :: (Eq k) => k -> [(k, v)] -> v
findKey key xs = snd . first . filter (\(k, v) -> k == key) $ xs
```

後ろのfilterが空リストだと
このfirstが例外を吐く

  - 例外はイヤン
  - 「多分値が得られるはず」な型
    Maybe を使いましょう

```
findKey :: (Eq k) => k -> [(k, v)] -> Maybe v
findKey key [] = Nothing
findKey key (k,v):xs
    | key == k = Just v
    | otherwise = findKey key xs
```

  - うーんいかにもfold使いたくなる再帰だ

```
findKey :: (Eq k) => k -> [(k, v)] -> Maybe v
findKey key = foldr
                (\(k, v) acc -> if key == k then Just v else acc)
                Nothing
```

キーと値の紐付けする方法はもうあったんや!

- 実はData.Mapってデータ構造もうあるねん
- 連想リストからMap作れるよ

Data.MapはPreludeと被るのでqualifiedでインポート

```
import qualified Data.Map as Map
Prelude Map> Map.fromList [(3,"shoes"),(4,"trees"),(9,"bees")]
... (なんか出るけど気にしない)
fromList [(3,"shoes"),(4,"trees"),(9,"bees")]
Prelude Map> Map.fromList [(1,"hoge"),(1,"fuga"),(1,"fumu")]
fromList [(1,"fumu")]
```

同じキーは後勝ち

- Map.fromListの型シグニチャ

```
:t Map.fromList
Map.fromList :: Ord k => [(k, a)] -> Map.Map k a
```

  - キーがOrdであることに注意
  - Eqだけではなく順序性を持たないとダメ

phoneBookふたたび

- Mapを使ってphoneBookを再度表現してみる

```
import qualified Data.Map as Map

phoneBook :: Map.Map String String
phoneBook = Map.fromList $
    [("betty", "555-2938")
    ,("bonnie", "452-2928")
    ,("patsy", "493-2928")
    ,("lucille", "205-2928")
    ,("wendy", "939-8282")
    ,("penny", "853-2492")
    ]
```

- 検索はMap.lookup

```
Prelude Map> :t Map.lookup
Map.lookup :: Ord k => k -> Map.Map k a -> Maybe a
Prelude Map> Map.lookup "bonnie" phoneBook
Just "452-2928"
Prelude Map> Map.lookup "john" phoneBook
Nothing
```

Maybeなので……

結果はJustでwrapされる

見つからない時は
Nothing

phoneBookふたたび(2)

- 電話帳に項目を追加しよう

```
Prelude Map> :t Map.insert
Map.insert :: Ord k => k -> a -> Map.Map k a -> Map.Map k a
Prelude Map> let newBook = Map.insert "john" "341-9021" phoneBook
Prelude Map> Map.lookup "john" newBook
Just "341-9021"
```

破壊的操作がないので
新しいMapが
作られる

- サイズを確認

```
Prelude Map> Map.size newBook
7
Prelude Map> Map.size phoneBook
6
```

phoneBookふたたび(3)

- 何でお前電話番号を文字列で保持するねん
  - 番号だからIntegerの配列でいいでしょ?
  - ガツッと変換したいよね
    - まずは変換関数を実装しよう
    - HaskellではString = [Char]だからData.CharのdigitToIntをmapすれば良さげ
    - ハイフンはisDigitでフィルターすればいいんじゃね?

```
string2digits :: String -> [Int]
string2digits = map digitToInt . filter isDigit
```

    - ほいできた。

```
*Main> string2digits "123-4567"
[1,2,3,4,5,6,7]
```

phoneBookふたたび(4)

- 変換関数をMapの全要素に適用したい
  - Map.map関数

```
*Main Map> :t Map.map
Map.map :: (a -> b) -> Map.Map k a -> Map.Map k b
```

  - こいつで一撃!

```
*Main Map> let intBook = Map.map string2digits phoneBook
*Main Map> Map.lookup "betty" intBook
Just [5,5,5,2,9,3,8]
```

重複のあるphoneBook

- こんな電話帳があったらどうしよう？

```
phoneBook =
    [("betty", "555-2938")
    ,("betty", "342-2492")
    ,("bonnie", "452-2928")
    ,("patsy", "493-2928")
    ,("patsy", "943-2929")
    ,("patsy", "827-9162")
    ,("lucille", "205-2928")
    ,("wendy", "939-8282")
    ,("penny", "853-2492")
    ,("penny", "555-2111")
    ]
```

- fromListを使うと
  - 重複するキーは問答無用で削除されちゃう
  - ソレじゃ困ることもあるよね？

fromListWith

- 重複したキーがあったときにどうするかを指定する関数を受け取るfromListの派生系

```
*Main Map> :t Map.fromListWith
Map.fromListWith
  :: Ord k => (a -> a -> a) -> [(k, a)] -> Map.Map k a
```

- 例えば「重複するキーがある場合、カンマで区切って結合する」という処理にするなら:

```
import qualified Data.Map as Map

phoneBookToMap :: (Ord k)  =>  [(k, String)] -> Map.Map k String
phoneBookToMap xs = Map.fromListWith add xs
    where add number1 number2 = number1 ++ ", " ++ number2
```

  - 試しに実行:

```
*Main> Map.lookup "patsy" $ phoneBookToMap phoneBook
Just "827-9162, 943-2929, 493-2928"
*Main> Map.lookup "bonnie" $ phoneBookToMap phoneBook
Just "452-2928"
```

モジュールを作ろう!(1)

- モジュールは簡単に自作可能
- 関数を定義して**エクスポート**すればそのモジュールの外部から利用可能になる
- エクスポートしない**内部関数**も実装可能
- お試しに幾何学オブジェクトの体積と面積を求めるGeometryモジュールを作ってみよう

モジュールを作ろう!(2)

- モジュールを構成するファイルは
  <モジュール名>.hs
- まずはこんな感じでエクスポートする関数を列挙

```
module Geometry
( sphereVolume
, sphereArea
, cubeVolume
, cubeArea
, cuboidVolume
, cuboidArea
) where
```

  - sphere（球体）、cubic（立方体）、cuboid（直方体）それぞれの体積（volume）、表面積（area）を求める

モジュールを作ろう!(3)

- そいで関数の定義をダラダラっと書く

```
...
) where

sphereVolume :: Float -> Float
sphereVolume radius = (4.0 / 3.0) * pi * (radius ^ 3)

sphereArea :: Float -> Float
sphereArea radius = 4 * pi * (radius ^ 2)

cubeVolume :: Float -> Float
cubeVolume side = cuboidVolume side side side

cubeArea :: Float -> Float
cubeArea side = cuboidArea side side side

cuboidVolume :: Float -> Float -> Float -> Float
cuboidVolume a b c = rectArea a b  * c

cuboidArea :: Float -> Float -> Float -> Float
cuboidArea a b c = rectArea a b * 2 + rectArea a c * 2 +
rectArea c b * 2

rectArea :: Float -> Float -> Float
rectArea a b -> a * b
```

モジュールを作ろう!(4)

- 使うときには import でいい……と本には書いてあるけど、GHCi だとうまくいかん
  - :load だと読めるけど、エクスポートしてない関数まで呼べちゃうのが困りモノ
  - ファイルで import Geometry ってやるとうまく行きます

```
$ cat testGeometry.hs
import Geometry
ghci testGeometry.hs
GHCi, version 7.6.3: http://www.haskell.org/ghc/  :? for help
...
*Main> cubeVolume 10
1000.0
*Main> rectArea 10 20

<interactive>:3:1: Not in scope: `rectArea'
```

階層構造を持つモジュール (1)

- モジュールには階層構造を持たせることが可能
  - Data.Mapみたいなね
- 作り方は単にディレクトリ掘ってその下にファイル置くだけ
- 今回はこういう構造にしてみよう
  - Geometry
    - Sphere
    - Cube
    - Cuboid

```
$ tree Geometry
Geometry
├── Cube.hs
├── Cuboid.hs
└── Sphere.hs
```

階層構造を持つモジュール(2)

- SphereとCuboidは超簡単

```
module Geometry.Sphere
( volume
, area
) where

volume :: Float -> Float
volume radius = (4.0 / 3.0) * pi * (radius ^ 3)

area :: Float -> Float
area radius = 4 * pi * (radius ^ 2)
```

```
module Geometry.Cuboid
(  volume
, area
) where

volume :: Float -> Float -> Float -> Float
volume a b c = rectArea a b  * c

area :: Float -> Float -> Float -> Float
area a b c = rectArea a b * 2 + rectArea a c * 2 + rectArea c b * 2

rectArea :: Float -> Float -> Float
rectArea a b = a * b
```

階層構造を持つモジュール (3)

- Cubeは内部でCuboidを呼ぶのでimportが必要

```
module Geometry.Cube
( volume
, area
) where

import qualified Geometry.Cuboid as Cuboid

volume :: Float -> Float
volume side = Cuboid.volume side side side

area :: Float -> Float
area side = Cuboid.area side side side
```

階層構造を持つモジュール (4)

- とりあえず使ってみる

```
$ cat testGeometry2.hs
import Geometry.Sphere
$ ghci testGeometry2.hs
GHCi, version 7.6.3: http://www.haskell.org/ghc/  :? for help
...
Ok, modules loaded: Geometry.Sphere, Main.
*Main> volume 10
4188.7905
*Main> area 10
1256.6371
```

  - 使い分けしたいならqualifiedでインポート

```
import qualified Geometry.Sphere as Sphere
import qualified Geometry.Cube as Cube
import qualified Geometry.Cuboid as Cuboid
```

練習問題①

- GHCiで関数を部分適用すると:

```
*Main> multThree 3 5

<interactive>:7:1:
    No instance for (Show (Int -> Int)) arising from a use of `print'
    Possible fix: add an instance declaration for (Show (Int -> Int))
    In a stmt of an interactive GHCi command: print it
```

  以上のように例外がおきます。

- なぜだか理由を答えてください。
  - ヒント1: 例外のメッセージをよく見ましょう
  - ヒント2: GHCiで値を表示できる条件とは?

練習問題②

- 1~100までの数値でコラッツ数列が15を超える数値を列挙してください。
  - テキストのp.72～73に思いっきり答え書いてあるのでカンニングしちゃダメだよ
  - ただしコラッツ数列を求めるプログラムは以下

```
chain :: Integer -> [ Integer ]
chain 1 = [1]
chain n
    | even n = n : chain (n `div` 2)
    | odd  n = n : chain (n * 3 + 1)
```

  - もーつヒント:リストの長さは length 関数で得られます

練習問題③

- 組み込み関数elemと同じ動きをするelem'をfold（左右は作りやすい方で）で実装してみてください。

練習問題④

- 例で作成したGeometryモジュールのサブモジュールSphere、Cube、Cuboidを試しに実行した結果をデモしてください。