

# すごいHaskellたのしく学ぼう!

## 読書会 #1

イントロダクション  
第1章 はじめの第一歩  
第2章 型を信じろ!


おがさわらなるひこ

# イントロダクション

Haskellは  
おもしろい。  
以上!

だって、そう書いてあるんだもん……。

# イントロダクション(2)

- Haskellは**純粋関数型プログラミング言語** 
  - 「関数型言語」いうな
- 「何をするか」ではなく「何であるか」を伝える
- **副作用を持たない = 参照透明性**
- Haskellは怠け者 = **遅延評価 (lazy evaluation)**
- **静的型付け言語**
- **強力な型推論**
- とりあえず **Haskell Platform** 入れなさい

# 第1章

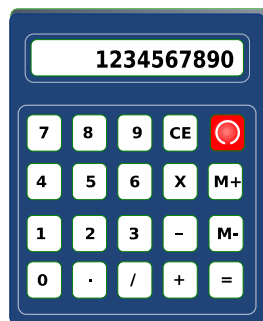
はじめの第一歩

# とりあえずGHCi起動

- インストールしてるよね

```
$ ghci
GHCi, version 7.6.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> 2 + 15
17
Prelude> 3 * 7
21
...
```

- 適当に式を入力すると電卓になるよ



# 電卓でいろいろ遊ぶ

- 数式は普通に書けます
  - 演算子順序は予想通り
- 括弧も使えるよ
- ただし負の数は注意

＋ －  
÷ ×

```
Prelude> 3 * -10  
  
<interactive>:2:1:  
  Precedence parsing error  
    cannot mix `*' [infixl 7] and prefix `-' [infixl 6]  
                                in the same infix expression  
  
Prelude> 3 * (-10)  
-30
```

括弧を  
つければ  
OK

中置演算子 '\*'  
と  
前置演算子 '-'  
は一緒に  
使えないよ!

# 関数

- 実は '+' や '\*' も関数
  - '\*' は「引数を(両側に)二つ取り、その二つを掛けた数を返す関数
  - 中置関数
- 多くの関数は「前置」
  - 名前の通り「前に置く」

```
Prelude> succ 100  
101
```

successor  
の意味  
引数の「次」を  
返す

- 括弧は要りませんよ!

# 関数(続き)

- Haskellでは関数「呼び出し」ではなく関数「適用」(apply)という方が普通
  - 数学の言葉遣いと一緒に(だよね?)
- 関数適用は他の演算子より結びつきが強い



```
Prelude> succ 99 * max 32 56  
5600
```

- バッククォート ` を使うと中置で関数を呼べる

```
Prelude> 100 / 7  
14.285714285714286  
Prelude> div 100 7  
14  
Prelude> 100 `div` 7  
14
```

'/' は浮動小数点型の割り算

div関数は整数の割り算

`div` と書くと中置できるよ!



# 関数を作ろう

- GHCiでは関数定義が(普通には)できない
  - ファイルにして :l (load) コマンドで読もう

```
$ cat baby.hs
doubleMe x = x + x
doubleUs x y = x * 2 + y * 2
$ ghci
```

引数xをとる関数  
doubleMe の  
定義は……

```
...
Prelude> :l baby.hs
[1 of 1] Compiling Main
Ok, modules loaded: Main.
*Main>
```

( baby.hs, interpreted )

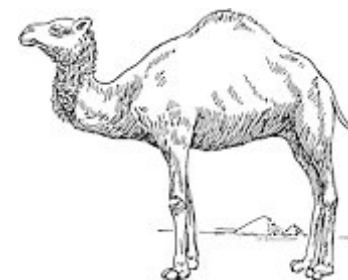
'Main' の意味はまたいつか

- 作ってしまえば呼ぶのは普通の関数と同じ

```
*Main> doubleMe 100
200
*Main> 10 `doubleUs` 1000
2020
```

# 関数応用編

- 関数の名前
  - 先頭大文字NG (いわゆるcamelCase)
  - クォート記号OK(naruoga'sFunction)
    - ある関数  $f$  の派生系に  $f'$  って名前をつけることがよくある
- GHCi内での関数定義は `let` を使う
- 関数内で場合分け
  - Haskellでは `if` も関数なんでこんな感じ



```
Prelude> let doubleSmallNumber x = if x > 100 then x else x * 2
Prelude> doubleSmallNumber 500
500
Prelude> doubleSmallNumber 50
100
```

# リスト!

- こういう奴 [1,2,3,4,5]
  - 全部の要素が同じ型じゃないとダメ
- いろいろ演算

```
Prelude> let a = [1,2,3,4,5]
Prelude> a
[1,2,3,4,5]
Prelude> a ++ [6,7,8,9,10]
[1,2,3,4,5,6,7,8,9,10]
Prelude> 0:a
[0,1,2,3,4,5]
```

代入と参照

連結

先頭に一個だけ足す(cons)

- 実は文字列もリスト

```
Prelude> "Hello"
"Hello"
Prelude> "Hello" ++ ", " ++ "world"
"Hello, world"
Prelude> 'Y':"Hello"
"YHello"
```

# もっとリスト

- 先頭から何番目の要素をくれないな

```
Prelude> [1,2,3,4,5] !! 3  
4
```

先頭は0始まりです

```
Prelude> [1,2,3,4,5] !! 5
```

```
*** Exception: Prelude.(!!): index too large
```

範囲外だと怒られる

- リスト入りリスト(リスト自体も型だしね)

```
Prelude> let b = [1,2,3,4,5]:[]
```

```
Prelude> b
```

```
[[1,2,3,4,5]]
```

できた

空のリスト [] にconsすると  
リストを作れる

```
Prelude> [9,9,9]:b
```

```
[[9,9,9],[1,2,3,4,5]]
```

整数のリストなら  
要素数が違っても同じ型

```
Prelude> b ++ [[0,-1,-2]]
```

```
[[1,2,3,4,5],[0,-1,-2]]
```

当然結合も可

```
Prelude> b ++ [['a','b']]
```

文字列のリストだと?

```
<interactive>:23:8:
```

(省略するけど型が違うので怒られます)

# もっともっとリスト

## • リストを比べるよ

```
Prelude> [3,2,1] > [2,1,0]
True
Prelude> [3,2,1] > [2,1,100]
True
Prelude> [3,4,2] > [3,4,3]
False
Prelude> [3,4,2] > [3]
True
Prelude> [3,4,2] > []
True
Prelude> [1,2,3] == [1,2,3]
True
Prelude> [1,2,3] == [3,2,1]
False
Prelude> [1,2,3] /= [3,2,1]
True
```

先頭要素から  
順に  
大きいほうが  
大きい

長さが  
違ってても  
OK

空リストは  
どんなリストより  
小さい

等しいか  
どうかは  
==

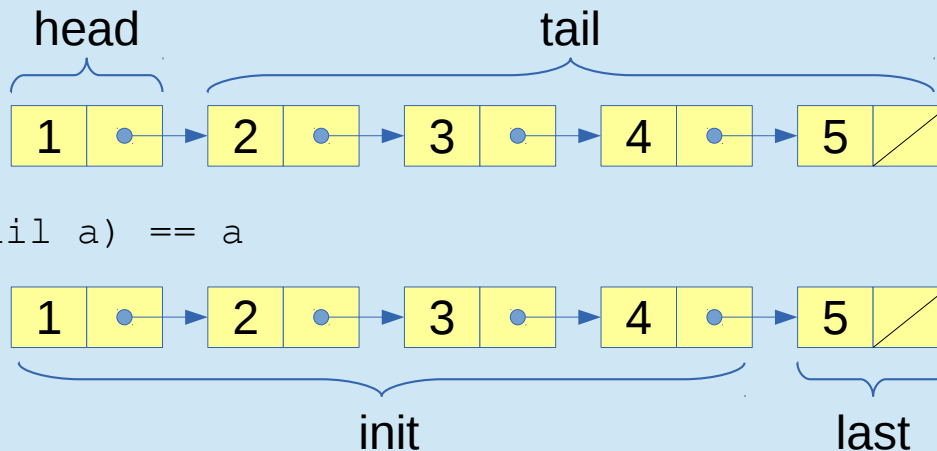
順序には  
意味がある

不等号は  
!= じゃないよ

# もっともっともっとリスト

- head, tail, init, last

```
Prelude> a
[1,2,3,4,5]
Prelude> head a
1
Prelude> tail a
[2,3,4,5]
Prelude> (head a):(tail a) == a
True
Prelude> init a
[1,2,3,4]
Prelude> last a
5
Prelude> (init a) ++ [last a]
[1,2,3,4,5]
```



- 空のリストはこれらは適用できないよ!

```
Prelude> head []
*** Exception: Prelude.head: empty list
```

# もっともっともっともっとリスト

- まだまだあるぞ便利関数 (スクショ省略)

length l	リスト l の長さを知る
null l	リスト l が空リストかどうかを調べる
reverse l	リスト l をひっくり返す
take n l	リスト l から先頭 n 個の要素を取り出す
drop n l	リスト l から先頭 n 個の要素を捨てる
maximum l	リスト l の各要素の最大値を返す
minimum l	リスト l の各要素の最小値を返す
sum l	リスト l の各要素の総和をとる
product l	リスト l の各要素の全ての積をとる
e `elem` l	リスト l 内に要素 e があるかどうか調べる

# レンジでチン!

- 今日日の電子レンジはチン!っていいません :)
- 範囲指定する構文
  - 刻み(ステップ)も指定できるお
  - 整数に限らないお

```
Prelude> [1..20]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
Prelude> [2,4..20]
[2,4,6,8,10,12,14,16,18,20]
Prelude> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
```

- 逆順するときはちょっと注意

```
Prelude> [20..1]
[]
Prelude> [20,19..1]
[20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1]
```

?

ステップを指定するとうまくいく



# 無限リスト!

- 「端っこを指定しない範囲」で無限リストが作れる
- takeと組み合わせると超便利!

```
Prelude> take 10 [13,26..]  
[13,26,39,52,65,78,91,104,117,130]
```

13の倍数を  
10個ちょうだい!

- 遅延評価ばんざい
  - take 10 を評価して「実際に値を取り出す」ときまでリストの実体を作らないということ
    - Scalaにもlazyっていうvalの修飾子があるによる
- あと便利な関数たち
  - cycle [1,2,3] / repeat 'a' / replicate 100 'a'

# リスト内包表記

- 数学の集合の表記に似た感じでリストを作る

```
Prelude> [x*2 | x <- [1..10]]  
[2,4,6,8,10,12,14,16,18,20]  
Prelude> [x*2 | x <- [1..10], x*2 >= 12]  
[12,14,16,18,20]  
Prelude> [x | x <- [50..100], x `mod` 7 == 0]  
[56,63,70,77,84,91,98]
```

基本形

条件も  
書ける

- 条件のことを「フィルター」っていいます
  - 条件はいっぱいかけます(論理和になるよ)
- 関数にもできるよん

```
Prelude> let boomBang xs = [if x < 10 then "BOOM!" else "BANG!"  
                           | x <- xs, odd x]  
  
Prelude> boomBang [7..13]  
["BOOM!", "BOOM!", "BANG!", "BANG!"]
```

oddは  
奇数を  
与えると  
真

# 複数のリストを与えたら

- こんなものも作れるよん

```
Prelude> [x + y | x <- [1,2,3], y <- [10,100,1000]]  
[11,101,1001,12,102,1002,13,103,1003]  
Prelude> [ x*y | x <- [2,5,10], y <- [8,10,11]]  
[16,20,22,40,50,55,80,100,110]  
Prelude> [ x*y | x <- [2,5,10], y <- [8,10,11], x*y > 50]  
[55,80,100,110]
```

- ちょっとお遊び

```
Prelude> let nouns = [ "hobo", "frog", "pope"]  
Prelude> let adjectives = ["lazy","grouchy", "scheming"]  
Prelude> [adjective ++ " " ++ noun  
          | adjective <- adjectives, noun <- nouns]  
["lazy hobo","lazy frog","lazy pope","grouchy hobo",  
 "grouchy frog","grouchy pope","scheming hobo","scheming frog",  
 "scheming pope"]
```

# リスト内包表記応用編

- lengthの再定義

- `length' xs = sum [1 | _ <- xs]`

- アンダースコア `_` は無名変数

- これによって `[1 | _ <- [3, 4, 5]] => [1, 1, 1]`

- `sum xs` は `xs` の各要素を全部足しあわせたものを返す

- 文字列操作もリスト操作

- `removeNonUpperCase st =  
[ c | c <- st, c `elem` ['A'..'Z']]`

- 見ればわかるので解説は省略w

# たぶる!

- ○ tuple × TaPL <http://www.amazon.co.jp/dp/4274069117>
- リストとよく似ている
  - リスト: [1,2,3]、タプル: (1,2,3)
  - 括弧が違うだけかい!
- 違いはなんだ!
  - 1. 型が混在できる  
○ (1, "abc", [7, 8, 9]) × [1, "abc"]
  - 2. 要素の種類が違うタプルは型が異なる  
○ [(1,2),(3,4)] × [(1,2),(3,4,5)] × [(1,2),(3,'a')]

# タプルの使い道

- 構造体っぽく使う
  - (“小笠原”, “徳彦”, 42)
- 型がきっちりしてるのでリストに同じ形式のデータを放り込みたいときに嬉しい
  - 座標列とかね

# ダブル = 値が二つのタプル

- よく使うので特別な関数が用意されています

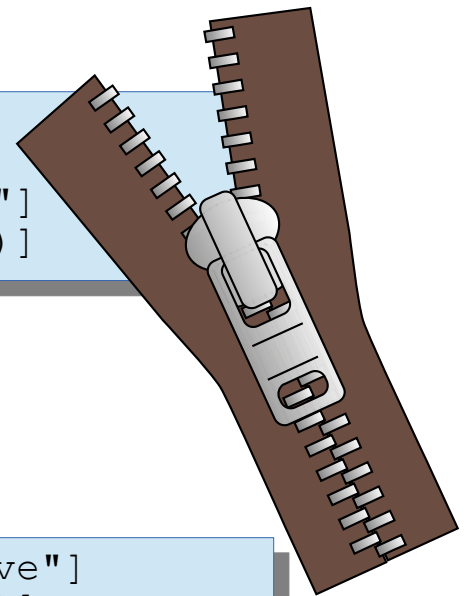
```
Prelude> fst (1,100)
1
Prelude> snd (1,100)
100
```

- zip – リストをジッパーで閉じる

```
Prelude> zip [1,2,3] [5,5,5]
[(1,5),(2,5),(3,5)]
Prelude> zip [1..5] ["one","two","three","four","five"]
[(1,"one"),(2,"two"),(3,"three"),(4,"four"),(5,"five")]
```

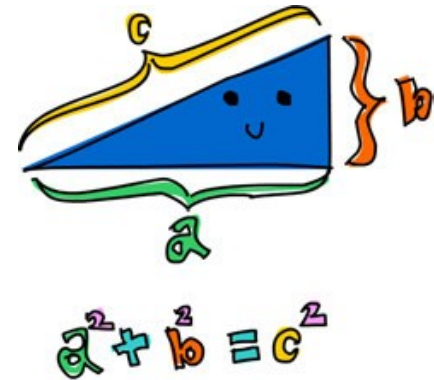
- 長さが不一致でも大丈夫
  - 短いほうが優先

```
Prelude> zip [1..100] ["one","two","three","four","five"]
[(1,"one"),(2,"two"),(3,"three"),(4,"four"),(5,"five")]
```



# 応用！

- 二等辺三角形の辺の組を探そう
  - 辺の長さは10以下の整数
  - 以下のようにトリプルのリストが作れる
    - $[(a,b,c) \mid a \leftarrow [1..10], b \leftarrow [1..10], c \leftarrow [1..10]]$
  - $b < a < c$  になるように制限を加える
    - $[(a,b,c) \mid c \leftarrow [1..10], a \leftarrow [1..c], b \leftarrow [1..a]]$
  - ピタゴラスの定理を満たす条件を加える
    - $[(a,b,c) \mid c \leftarrow [1..10], a \leftarrow [1..c], b \leftarrow [1..a],$   
 $a^2 + b^2 == c^2]$
  - ほい実行！
    - $[(4,3,5),(8,6,10)]$





# 第1章まとめ

- GHCi はHaskellの対話型環境 (REPL)
- GHCi 電卓でHaskellに馴染もう
- Haskellでは関数を「適用」する
- 関数の定義も簡単
- リスト重要
- タプルもあるですよ
- 「どんどん条件を増やして答えを絞り込む」のが関数型プログラミングのスタイル！

# 第2章

型を信じろ！

# 型がない世界なんかない！

- 静的か動的か
  - コンパイル時にすべての変数の型が決まっているか、実行時に決まるかどうか
    - 静的型付けだと型エラーはコンパイル時に検知できる
    - 動的型付けだと型エラーは実行時エラー（例外）になる
- 強い型付け vs 弱い型付け
  - 型の不一致があったときにどうするか
    - エラーにするのが強い型付け
    - 不一致はなんとなく解決するのが弱い型付け
      - $1 + '2' = ?$

# 型を調べる!

- GHCi の :t コマンド
  - 型を調べる

```
Prelude> :t 'a'
'a' :: Char
Prelude> :t True
True :: Bool
Prelude> :t "HELLO!"
"HELLO!" :: [Char]
Prelude> :t (True, 'a')
(True, 'a') :: (Bool, Char)
Prelude> :t 4 == 5
4 == 5 :: Bool
```

なぜ素直に  
数値の  
型を  
調べないかは  
秘密

- 関数にも「型」がある!

```
Prelude> :t removeNonUpperCase
removeNonUpperCase :: [Char] -> [Char]
```

「[Char] (文字列)を  
受取り  
[Char] を  
返す関数」  
という型

# 関数の定義と型シグニチャ

- 関数の型宣言の表記＝「型シグニチャ」

```
$ cat addThree.hs  
addThree :: Int -> Int -> Int -> Int  
addThree x y z = x + y + z
```

- なにこの  $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$ ?
  - 今のところは「Int型の引数を三つ取ってIntを返す」と読んでおけばいい
  - 詳しい話は後ほど
- ちょっと横入りだけどHaskellの型シグニチャは記法が綺麗でちょっとうらやましい
  - Scala（に限らず普通の言語）だと型情報が埋もれてしまう  
`Int addThree(Int x, Int y, Int z)`

# よく出てくる型たち

- Int: 整数 **有界** (最小・最大がある)
- Integer: 整数 **有界でない**
- Float: 単精度浮動小数点数
- Double: 倍精度浮動小数点数
- Bool: 真偽値
- Char: Unicode文字
- タプル:

# 型変数

- 例として二つほど

```
Prelude> :t head
head :: [a] -> a
Prelude> :t fst
fst :: (a, b) -> a
Prelude> :t snd
snd :: (a, b) -> b
```

- head の `[a] -> a` とは「何でもいい型 `a` のリストを取って、型 `a` の値を返す」の意味
- `fst` と `snd` の例だと、ダブルの場合二つの型 `a`, `b` は違うかもしれないけどそれぞれと同じってこと
- 型変数を用いた関数 = **多相的(polymorphic)関数**

# 型と暮らそう型クラス 初級編

(その冗談つままないよ……)

- なんらかの振る舞いを定義するインターフェース
- ある型クラスのインスタンスである型は、型クラスで定義されたインターフェースを実装する
- 例として (==) 見よう

```
Prelude> :t (==)  
(==) :: Eq a => a -> a -> Bool
```

- $=>$  を「型クラス制約」と呼ぶ
  - 「型クラス Eq のインスタンスである型 a があって a を二つ取り bool を返す関数」と読める

(==): ==演算子も関数であり(等値性関数)、Haskellの文法でデフォルトで中置になる。関数として見るときにはカッコで囲う



# え、どういうこと？

- Eq というのは「等値であるといえる」というインタフェースを持った型クラス
  - なので Eq のインスタンスは == 演算子で比較ができるということ
- インタフェースという考え方自体は Java のインタフェースとそんなには変わらない
- けど、Haskell の場合「型の性質」に着目した型クラスを多数用意する(定義もできる)ことで、型という概念をより強固にかつ柔軟にしている

# よく出てくる型クラスたち

名前	インターフェース	意味
Eq	==, /=	等値性をテストできる
Ord	<, >, <=, >=	順序性をテストできる
Show	show	値を文字列に変換できる
Read	read	文字列を値に変換できる
Enum	succ, pred	順番に並んでいる。「次」と「前」が指定できる。レンジで使える
Bounded	minbound, maxbound	上限と下限を持つ (有界)
Num	(数値演算)	数を表す
Floating		浮動小数点数
Integral		整数

# 型クラスと型推論と型注釈

- 型クラス制約のついた多相関数は型推論で答えを出す……けど出せないときも

```
Prelude> :t read
read :: Read a => String -> a
Prelude> read "5.3"
```

```
<interactive>:21:1:
  ...
```

```
Prelude> read "5.3" + 3.2
8.5
```

read 関数は  
文字列を受け取り  
Read 型クラスのインスタンスである  
型 a の値を返す

単に文字列を  
渡すと怒られた!

こうするとOK  
「浮動小数点と足せるなら  
浮動小数点」と  
推論できるから

- 推論できないときの「型注釈」

```
Prelude> read "5.3" :: Float
5.3
```

結果は  
Floatですよ!と  
「注釈」してやる

# ちょっと注意

- 演算子 (+) の型を見ると

```
Prelude> :t (+)
Num a => a -> a -> a
```

- すべての型 a は同じでないといけない

```
Prelude> 1 + 2.3
3.3
Prelude> (1 :: Int) + (2 :: Integer)
```

```
<interactive>:32:15:
  Couldn't match expected type `Int' with actual type `Integer'
  In the second argument of `(+)', namely `(2 :: Integer)'
  In the expression: (1 :: Int) + (2 :: Integer)
  In an equation for `it': it = (1 :: Int) + (2 :: Integer)
```

これはOK  
型推論により 1 は Float に

- さすがにこの間違いはありえない

こっちは  
明示的に違う型を  
指定してるのでアウト

- けど関数定義のときには狭い型を使わないよう注意

## 第2章まとめ

- 関数の明示的な型宣言は「型シグニチャ」と呼ぶ
- Haskellの一般的な型をいくつか見てきた
- 型変数と多相的関数
- 型クラス
  - 一つの型は複数の型クラスのインスタンスになることに注意

# 練習問題①

- 「Haskell」の名前の由来をしらべてください。

## 練習問題②

- cycle, repeat, replicate のそれぞれの意味と実行例を示してください。
- また、replicate は他の関数の組み合わせで実現できます。relicate と同じ動きをする replicate' を実装してみてください。

## 練習問題③

- 次のような動きをする関数 `fizzBuz` を作りましょう。  
整数のリストを受け取り「3の倍数のときは Fizz」  
「5の倍数のときは Buzz」「3と5の倍数のときは FizzBuzz」「それ以外は数字を文字列にしたもの」を返します。  
数字を文字列にするには関数 `show` が使えます。

```
Prelude> fizzBuzz [1..30]
["1","2","Fizz","4","Buzz","Fizz","7","8","Fizz","Buz
z","11","Fizz","13","14","FizzBuzz","16","17","Fizz",
"19","Buzz","Fizz","22","23","Fizz","Buzz","26","Fizz
","28","29","FizzBuzz"]
```



## 練習問題④

- 先に示した関数:

```
removeNonUpperCase st =  
  [ c | c <- st, c `elem` ['A'..'Z']]
```

について、型シグニチャを指定してください。