


# すごいHaskell つくばで学ぼう!

2016年2月1日-2日  
村主 崇行

# 質問は

- 村主まで！
  - 1.いつでも直接声をかけてください。
  - 2.または [muranushi@gmail.com](mailto:muranushi@gmail.com)
- <http://ja.stackoverflow.com/questions/tagged/haskell>
- Haskell関係のメーリングリスト(英語)
  - 1.雑談用 [haskell-cafe@haskell.org](mailto:haskell-cafe@haskell.org)
  - 2.初心者用 [beginners@haskell.org](mailto:beginners@haskell.org)
  - "Here, there is no such thing as a 'stupid question.'"

# 準備

- 今回使用する材料は以下のレポジトリに置いてあります。
- <https://github.com/nushio3/learn-haskell>
-  ボタン、またはgitを使って取得してください。
- `setup-tsukuba-2016.sh` の中身を確認して実行してください。stack(必須)とz3(オプション)をインストールします。
- 自前環境の方、スクリプトがうまく動かない方は <https://github.com/Z3Prover/z3/releases> からz3をインストールしてください。
- 上記を実行すると2.7GBくらいのディスク容量を持っていかなれますので、残り容量が不安な方は環境にもともと入っているcabal/ghcを使ってください。

# Chapter 1

# Haskellの使い方

# Haskell is ...

今回みなさんと学ぶプログラミング言語

- Lazy Evaluation
- Type Class
- Sexy Types
- 一貫していて類推できるUI
- 進化が速い

Glasgow Haskell Compiler

Haskell処理系のデファクトスタンダード

# Haskell is not magic

- Xmonad (Haskellで書かれたWindow Manager) に脆弱性があった。

<http://jvndb.jvn.jp/ja/contents/2013/JVNDB-2013-006660.html>

“想定される影響

第三者により、Web ページのタイトルを介して、ユーザが xmobar ウィンドウのタイトルをクリックした際にコマンドをアクティブにされることで、任意のコマンドを実行される可能性があります。”

# Haskellを使う上で非常によく使う Webサイト1つ

- **Stackage** <https://www.stackage.org/>  
ある時点で互換性が確認済みのLTS(long term support)ライブラリを週ごとにまとめている。Hackage, Hoogleの機能も持つ
- **Hackage** <https://hackage.haskell.org/package/base>  
Haskellのライブラリ集積サイトであり、ドキュメントを見ることができる。
- **Hoogle** <https://www.haskell.org/hoogle/>  
Haskellのライブラリをキーワードや、型で検索することができる。
- **Tryhaskell** <http://tryhaskell.org/>  
オンラインでHaskellプログラムを実行できる。

# コマンドラインツールstack

- stack.yamlの指定に従い、必要なバージョンのコンパイラ・依存ライブラリ等をダウンロード・コンパイルし環境を整えてくれる。
- stack登場以前、ghcやライブラリのバージョンが上がると相性問題が頻発していた問題を解決。
- Haskellのインタプリタ・コンパイラ、Haskellスクリプトの実行などはすべてstack経由で行える



# Stackの使い方 (インタプリタ篇)

```
~$ stack ghci
```

```
Run from outside a project, using implicit global project  
config
```

```
Using resolver: lts-4.0 from implicit global project's  
config file: /home/nushio/.stack/global-project/stack.yaml  
Error parsing targets: The specified targets matched no  
packages.
```

```
Perhaps you need to run 'stack init'?
```

```
Warning: build failed, but optimistically launching GHCi  
anyway
```

```
Configuring GHCi with the following packages:
```

```
GHCi, version 7.10.3: http://www.haskell.org/ghc/ :? for  
help
```

```
Ok, modules loaded: none.
```

```
> 1+1
```

```
2
```

```
> putStrLn "hello world!"
```

```
hello world!
```

```
>
```

# Stackの使い方(スクリプトファイル篇)

```
$ cat hello-world.hs
```

```
main :: IO ()
```

```
main = interact $ const "Hello World!\n"
```

```
$ runhaskell hello-world.hs
```

```
Hello World!
```

```
$ stack runhaskell hello-world.hs
```

```
Run from outside a project, using implicit global project  
config
```

```
Using resolver: lts-4.0 from implicit global project's config  
file: /home/nushio/.stack/global-project/stack.yaml
```

```
Hello World!
```

```
$
```

# Stackの使い方 (プロジェクト 篇)

- Stackは、Haskellのプロジェクトをテンプレートから生成してくれる機能も備えている。テンプレートの一覧はstack templatesで表示。

生成するプロジェクト名    テンプレート名

```
$ stack new hello-world simple
```

```
$ tree hello-world
```

```
hello-world
├── LICENSE
├── Setup.hs
├── hello-world.cabal
├── src
│   └── Main.hs
└── stack.yaml
```

- ライセンスファイル
- インストールスクリプト
- ソースコードの位置や、コンパイルオプションなどビルドに必要な設定を記したファイル
- ソースコード
- 使用するコンパイラのバージョンや各ライブラリのバージョン、その取得先などを記した環境設定ファイル



## 練習問題

- `stack new` プロジェクト名 `simple` コマンドで新しいプロジェクトを作り、`putStrLn`を使って文字列"`hello world`"を表示するプログラムを作り、ビルドし、実行してください。
- 上記で作った`Main.hs`を、`runhaskell`を使ってスクリプトとして実行してみてください。

Haskellのプログラムは  
3種類の要素からできている

値(value)  
型(type)  
型クラス(type class)

The diagram illustrates the components of the Haskell expression `(+) :: Num a => a -> a -> a` using red arrows and labels:

- A red double-headed arrow labeled **値** (value) spans the `(+)` operator.
- A red double-headed arrow labeled **型クラス** (type class) spans the `:: Num a` part.
- A long red double-headed arrow labeled **型** (type) spans the entire right-hand side `=> a -> a -> a`.

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

# Haskellは値、型、型クラスだ！ という証拠

<https://www.haskell.org/onlinereport/syntax-iso.html>

- Haskell文法定義より。

トップレベル宣言の構文定義を見ると...

```
topdecl    ->
              type simpletype = type
              | data [context =>] simpletype = constrs [deriving]
              | newtype [context =>] simpletype = newconstr [deriving]
              | class [scontext =>] tycls tyvar [where cdecls]
              | instance [scontext =>] qtycls inst [where idecls]
              | default (type1 , ... , typen)  (n>=0)
              | decl
```

型を作る構文

型クラス定義

// インスタンス

値の定義

# 値を調べる方法

- GHCiに表示させる

```
$ ghci
```

```
> 3+7
```

```
10
```

- `hackage`で調べる

# 型を調べる方法

- GHCiに表示させる

```
$ ghci
```

```
> :t 1
```

```
1 :: Num a => a
```

```
> :t "hello"
```

```
"hello" :: [Char]
```

- Hackageで調べる



# 型クラスを調べる方法

- GHCiで調べる

> **:info Num**

class Num a where 型クラスの名前

(+) :: a -> a -> a 型クラスに属するメソッド

(-) :: a -> a -> a

(\*) :: a -> a -> a

negate :: a -> a

abs :: a -> a

signum :: a -> a

fromInteger :: Integer -> a

-- Defined in `GHC.Num`

型クラスの代表的なインスタンス

instance Num Word -- Defined in `GHC.Num`

instance Num Integer -- Defined in `GHC.Num`

- Hackageで調べる

# 絶対に覚えていただきたい ghciの基本操作3つ

値を調べる	> 式
型を調べる	> :t 型名
型クラスを調べる	> :info 型クラス名
その他のghciコマンドを調べる	:help

# ライブラリの使い方の調べ方

- ライブラリの鍵となる関数を探す(自分のしてほしいことをしてくれる関数)
- その関数に渡す引数の作り方を調べる。
- 使って確かめる。



# 練習問題

- binary-searchというライブラリがあります。

<https://hackage.haskell.org/package/binary-search-1.0.0.2/docs/Numeric-Search.html>

- いったい何をするライブラリでしょう？
- どの関数が鍵関数だろうか？
- それに渡す引数は何があるだろうか？

# Prelude

- Haskellプログラムにデフォルトで読み込まれているモジュール。頻出の値、型、型クラスが一通り定義されている。

<https://hackage.haskell.org/package/base/docs/Prelude.html>

# Haskellプログラムの基本

```
openFile :: FilePath -> IOMode -> IO Handle
```

- $a \rightarrow b$ 型の関数に $a$ 型の値を渡すと $b$ 型になる。
- `do`の後ろに一行ずつ、`IO a`型の値を並べる。
- `IO a`が返す値は`<-`記号で取得できる。

```
main :: IO ()
```

```
main = do
```

```
    putStrLn "May I have your name?"
```

```
    name <- getLine
```

```
    putStrLn $ "Nice to meet you, " ++ name
```



# 文字列入出力

<https://hackage.haskell.org/package/base-4.8.2.0/docs/System-IO.html>

- `putStr :: String -> IO ()`  
Stringの内容を標準出力に書き出す。
- `print :: Show a => a -> IO ()`  
型aの値を文字列に変換して表示する。
- `getContents :: IO String`  
標準入力を全部読み込む
- `interact :: (String -> String) -> IO ()`  
文字列から文字列への関数をプログラムに変える



# ファイル入出力

<https://hackage.haskell.org/package/base-4.8.2.0/docs/System-IO.html>

- `readFile :: FilePath -> IO String`  
ファイルの内容をStringとして読み込む
- `writeFile :: FilePath -> String -> IO ()`  
ファイルに文字列を書き出す
- `hGetContents :: Handle -> IO String`  
ファイルハンドルからファイルの中身を読み込む
- `stdin, stdout, stderr :: Handle`
- `openFile :: FilePath -> IOMode -> IO Handle`
- `type FilePath = String`





# 演習問題

exercise-1-hello-cat

- `stack new` プロジェクト名 `simple` コマンドで新しいプロジェクトを作り、`putStrLn`を使って文字列`"hello world"`を表示するプログラムを作り、ビルドし、実行してください。
- 同様にして、標準入力を1行ずつ読み込んでそのまま出力するプログラム`cat`を作ってください。
- `hello-world`と`cat`を、`interact`を使って実装してみてください。(ヒント:`const`, `id` という関数を使おう。)

# Chapter 2

## 基本文法

# Haskellの基本構文要素7つ

- 変数 `x, y, +, *, map, >>=`
- リテラル `1, 3.14, '漢', "string"`
- 関数適用 `sin x`
- Let式 `let x = 9 in x*x`
- λ式 `\x -> x*x`
- Case式 `case b of True -> 1`
- 型 `1 :: Int`

この7つさえ覚えればどんなHaskellプログラムでも読めます！  
他の構文は糖衣構文で、上記のいずれかに還元されます。

# Haskellの基本構文要素は たった7つであるという証拠:

<https://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler/CoreSynType>

```
data Expr b-- "b" for the type of binders,
  = Var    Id
  | Lit    Literal
  | App    (Expr b) (Arg b)
  | Lam    b (Expr b)
  | Let    (Bind b) (Expr b)
  | Case   (Expr b) b Type [Alt b]
  | Cast   (Expr b) Coercion
  | Tick   (Tickish Id) (Expr b)
  | Type   Type
```

# Haskellの予約語

パターンマッチ

**case of**

型クラス

**class instance  
deriving**

型

**data newtype  
type**

モナド

**do**

if文

**if then else**

モジュール

**module  
import**

局所変数の定義

**let in  
where**

演算子の優先順位と結合方向の指定

**infix  
infixl infixr**

# GHC(7.10.3)の言語拡張106個・・・

- `OverlappingInstances`, `UndecidableInstances`, `IncoherentInstances`, `DoRec`, `RecursiveDo`, `ParallelListComp`, `MultiParamTypeClasses`, `MonomorphismRestriction`, `FunctionalDependencies`, `Rank2Types`, `RankNTypes`, `PolymorphicComponents`, `ExistentialQuantification`, `ScopedTypeVariables`, `PatternSignatures`, `ImplicitParams`, `FlexibleContexts`, `FlexibleInstances`, `EmptyDataDecls`, `CPP`, `KindSignatures`, `BangPatterns`, `TypeSynonymInstances`, `TemplateHaskell`, `ForeignFunctionInterface`, `Arrows`, `Generics`, `ImplicitPrelude`, `NamedFieldPuns`, `PatternGuards`, `GeneralizedNewtypeDeriving`, `ExtensibleRecords`, `RestrictedTypeSynonyms`, `HereDocuments`, `MagicHash`, `TypeFamilies`, `StandaloneDeriving`, `UnicodeSyntax`, `UnliftedFFITypes`, `InterruptibleFFI`, `CapiFFI`, `LiberalTypeSynonyms`, `TypeOperators`, `RecordWildCards`, `RecordPuns`, `DisambiguateRecordFields`, `TraditionalRecordSyntax`, `OverloadedStrings`, `GADTs`, `GADTSyntax`, `MonoPatBinds`, `RelaxedPolyRec`, `ExtendedDefaultRules`, `UnboxedTuples`, `DeriveDataTypeable`, `DeriveGeneric`, `DefaultSignatures`, `InstanceSigs`, `ConstrainedClassMethods`, `PackageImports`, `ImpredicativeTypes`, `NewQualifiedOperators`, `PostfixOperators`, `QuasiQuotes`, `TransformListComp`, `MonadComprehensions`, `ViewPatterns`, `XmlSyntax`, `RegularPatterns`, `TupleSections`, `GHCForeignImportPrim`, `NPlusKPatterns`, `DoAndIfThenElse`, `MultiWayIf`, `LambdaCase`, `RebindableSyntax`, `ExplicitForAll`, `DatatypeContexts`, `MonoLocalBinds`, `DeriveFunctor`, `DeriveTraversable`, `DeriveFoldable`, `NondecreasingIndentation`, `SafeImports`, `Safe`, `Trustworthy`, `Unsafe`, `ConstraintKinds`, `PolyKinds`, `DataKinds`, `ParallelArrays`, `RoleAnnotations`, `OverloadedLists`, `EmptyCase`, `AutoDeriveTypeable`, `NegativeLiterals`, `BinaryLiterals`, `NumDecimals`, `NullaryTypeClasses`, `ExplicitNamespaces`, `AllowAmbiguousTypes`, `JavaScriptFFI`, `PatternSynonyms`, `PartialTypeSignatures`, `NamedWildCards`, `DeriveAnyClass`

# でもghcやghciが必要な 言語拡張を教えてくれるから大丈夫！

> **type ShowRead x = (Show x, Read x)**

Illegal constraint synonym of kind:  
'GHC.Prim.Constraint'

(Use ConstraintKinds to permit this)

In the type declaration for 'ShowRead'

> **:set -XConstraintKinds**

> **type ShowRead x = (Show x, Read x)**

> (成功)

ソースコードで言語拡張を使うときは次のように書きます。

```
{-# LANGUAGE ConstraintKinds #-}
```

# 識別子名の規則

<https://www.haskell.org/onlinereport/syntax-iso.html>

	通常文字(Letter)	記号文字(Symbol)
大文字(large)	A,B,Д,... any Unicode uppercase or titlecase letter	: (半角コロン)
小文字(small)	a,b,д,πひ,ら,ガ,ナ,漢,字,... any Unicode lowercase letter	(その他の記号) any Unicode symbol or punctuation

- 変数名、型変数名は小文字から始まる
- 型名、型クラス名は大文字から始まる
- 通常文字の列は変数、関数としてparseされる
- 記号文字の列は中置演算子としてparseされる
- `div` のように通常文字列を囲むと記号文字列
- (+) のように記号文字列を囲むと通常文字列





# 練習問題

- インタプリタを使って、Haskellの4種類の識別子名を試してみてください。一番おもしろい識別子名を考えた奴が優勝

ヒント：インタプリタに右のように入力して実際に識別子を定義しようとすると、正しい識別子名なら通ります。

間違っているとエラーになる。

```
> let 百倍にする x = 100 * x
> let a ++ b = a + a*b + b
> type A型 = Integer
> type a : 株 b = String
```

```
> type a : 株 b = String
<interactive>:6:10:
    Unexpected type `株 b'
    In the type declaration for `: 株'
    A type declaration should have form
      type : 株 a b = ...
>
```

# 計算式

> **1+1**

2

> **"cheese" ++ "cake"**

"cheesecake"

> **1+2\*3-4/5**

6.2

> **4^4**

256

# 関数適用

```
> sin 1
```

```
0.8414709848078965
```

```
> sqrt 3
```

```
1.7320508075688772
```

```
> show 123
```

```
"123"
```

```
> read "456" :: Integer
```

```
456
```

```
>
```

# Let式

- let a=b in c

a=bであるときの式c

> **let x = 7 in x \* x**

49

# ラムダ式

- $\lambda x \rightarrow e$

xを受け取ってeという式になる関数

```
> (\x -> x * x) 7
```

49

```
> :t \x -> x * x
```

```
\x -> x * x :: Num a => a -> a
```

# Case式

- パターンマッチ(場合分け)をする式

```
> case 1 < 2 of {True -> "Yes"; False -> "No"}  
"Yes"
```

```
> case 1+1 of {2 -> "Okay"; _ -> "What?"}  
"Okay"
```

- ソースファイルでは複数行に分けて書くこともできる。

```
case userAge >= 20 of  
  True -> "課金承認"  
  False -> "おやごさんの きょかを もらってね!"
```



## 練習問題

- いまから画面にHaskellの式を表示しますので、その値をインタプリタに入力して計算した結果を教えてください。

# Chapter 3

## 型と型クラス



# Haskellの型

- 要素型 `Int`, `Char`, `Integer`, `Float`, ...
- 関数 `Double -> Double`, ...
- リスト `[Char]`, `[[Int]]`, ...
- タプル `(Int, String)`, `(Int, Int, Int)`
- 型コンストラクタと型適用 `Maybe Int`,  
`Either String Int`, `IO String`,  
`Map Telephone (FirstName, LastName)`



# 練習問題

1

- の型は何でしょう？
- いったいそれはどういう意味でしょう？

# 型クラスNum

> **:info Num**

class Num a where

(+) :: a -> a -> a

(-) :: a -> a -> a

(\*) :: a -> a -> a

negate :: a -> a

abs :: a -> a

signum :: a -> a

fromInteger :: Integer -> a

型  $a$  が型クラス Num のインスタンスであるとは  
以下のが定義されていることである。

足し算:  $a \rightarrow a \rightarrow a$

- おおむね「整数っぽいもの」の型クラス
- Haskellの大抵の演算子は何らかの型クラスに付属しており、型クラスに属する型なら何でも演算できる。



# 練習問題

- PythonやRubyなどのスクリプト言語
- C++言語
- Haskell

ではいずれも+演算子を多様な型に対して使えるよう定義できる仕組みがあるが、これら3種類の言語の方式の違いはなんだろうか？



# Preludeの主な数値演算型クラス

型クラス	主なメソッド
Num a 整数っぽいクラス	$(+)$ , $(-)$ , $(*)$ :: $a \rightarrow a \rightarrow a$ <code>fromInteger</code> :: <code>Integer</code> $\rightarrow$ $a$
Real a 分数っぽいクラス	<code>toRational</code> :: $a \rightarrow$ <code>Rational</code>
Integral a 商や余りがあるクラス	<code>div</code> , <code>mod</code> :: $a \rightarrow a \rightarrow a$
Fractional a 割り算ができるクラス	$(/)$ :: $a \rightarrow a \rightarrow a$
Floating a 実数っぽいクラス	<code>pi</code> :: $a$ <code>exp</code> , <code>sin</code> :: $a \rightarrow a$ $(**)$ :: $a \rightarrow a \rightarrow a$

<http://hackage.haskell.org/package/numeric-prelude> には数学で使う型クラスが沢山あります。



# Preludeの主な型

型	値
Char 文字	'a'
Int ...固定長整数	32, 9223372036854775807
Integer 多倍長整数	$2^{(300^3)}$
Float, Double 浮動小数	3.1415
Rational 分数	$3 \div 5$
String 文字列	"hello", ['つ','く','ば']
[a] aのリスト	[1,2,3], ['a'..'z'], [1..]
Maybe a aがいっこまで	Just 5, Nothing
(a,b) aとbのペア	(444444, "Domohorn")
Either a b aかbかどっちか	Left "error", Right 42



# Preludeの主な型クラス

型クラス	主なメソッド
<code>Eq a</code> 等号が定義されている	<code>(==), (/=) :: a -&gt; a -&gt; Bool</code>
<code>Ord a</code> 大小比較ができる	<code>(&lt;), (&lt;=) ... :: a -&gt; a -&gt; Bool</code>
<code>Show a</code> 文字列へ変換できる	<code>show :: a -&gt; String</code>
<code>Read a</code> 文字列から変換できる	<code>read :: String -&gt; a</code>
<code>Enum a</code> 前要素、後要素がある	<code>succ, pred :: a -&gt; a</code> <code>enumFrom :: a -&gt; [a]</code>



# Preludeの主な型変換関数

- Haskellは強い型付け言語（処理系が勝手に型変換をしない）のため、似たような型同士でも必ず明示的に変換関数で変換してやらないといけない。

正直な話、ちょっと煩雑に感じるときもある。

```
fromInteger :: Num a => Integer -> a
fromIntegral :: (Integral a, Num b) => a -> b
fromRational :: Fractional a => Rational -> a
realToFrac :: (Fractional b, Real a) => a -> b
show :: Show a => a -> String
read :: Read a => String -> a
```



# 型クラス則(type class laws)

- 型クラスには、「このような法則を満たすようにしてほしい」というコメントが備わっていることがあります。
- 満たすかどうかはインスタンス次第(満たせない・敢えて満たさない場合もある)

型クラス	型クラス則
Eq	$a == b \Leftrightarrow b == a$
Num	$(a+b)+c == a+(b+c)$ $a*(b+c) == a*b+a*c$
Show, Read	$\text{show (read } x) == x$
Integral, Num	$\text{toInteger (fromInteger } x) == x$

# Preludeに含まれる部分関数

- 部分関数(partial function)...入力型の一部についてしか定義されておらず、実行時エラーを起こす可能性のある関数。安全な版と使い分けよう。

危険: `read :: Read a => String -> a`  
安全: `readMaybe :: Read a => String -> Maybe a`  
安全: `readEither :: Read a => String -> Either String a`

危険: `head :: [a] -> a`  
安全: `headMay :: [a] -> Maybe a`

# 型を自分で作ろう

kazu.hs

- 3までしか数えられない数を作ってみましょう。
  - 次のように入力すると、新しい型Countが作れます
- ```
> data Count = One | Two | Three | Huh  
    deriving (Eq, Show)
```
- Count型の値は、One, Two, Three, Huhのいずれかです。(ひとつ、ふたつ、みっつ、ハアわからん)
  - インタプリタに入力する場合に全体を1行で入力して下さい。

# 型を自分で作ろう

- 次のように入力すると、新しい型Countが作れます。

```
data 型 = 値 | 値 | 値 | 値
```

```
> data Count = One | Two | Three | Huh
```

Preludeの既存の型も、まったく同じ構文で定義されています。

```
> :info Bool
```

```
data 型 = 値 | 値
```

```
data Bool = False | True -- Defined in  
`GHC.Types'
```

# Count型の値の計算をするには？

## CountをNumのインスタンスにします。

instance Num Count where

One + One = Two

One + Two = Three

Two + One = Three

\_ + \_ = Huh

関数定義などのパターンマッチにおける  
アンダースコア \_ 記号は「その他」の意味です。

One \* x = x

x \* One = x

\_ \* \_ = Huh

パターンマッチにおいて x などの変数を用いると、パターンの特定の位置にくる値に名前をつけられます。



引き算(-)、絶対値 abs、符号数signum、整数からの変換fromInteger  
といった、Numクラスの残りの関数も実装してみよう。(kazu.hsには実装済みです。)

# Count型は普通のHaskellの型と同様に扱えます。

```
$ ghci kazu.hs
```

```
> :t One
```

```
One :: Count
```

```
> :t Huh
```

```
Huh :: Count
```

```
> One + Two
```

```
Three
```

```
> Two + Two
```

```
Huh
```

```
> :info Count
```

```
data Count = One | Two | Three | Huh    -- Defined at kazu.hs:1:1
```

```
instance Eq Count -- Defined at kazu.hs:2:22
```

```
instance Num Count -- Defined at kazu.hs:4:10
```

stack経由で試す場合は、

```
$ stack ghci xxx
```

と書くと、「xxxというフォルダ内のプロジェクトの環境でghciを起動せよ」という意味になってしまうので、次のように、ghciを起動したあと `:l` (コロンえる) でファイルを読み込んでください。

```
$ stack ghci
```

```
...
```

```
> :l "kazu.hs"
```

```
...
```

```
> Two * One - One
```

```
One
```

- 1,2,3 ... 等がCount型のリテラルとして使えますが、少し難しい計算をさせると途中でオーバーフローしてHuh?になってしまいます。
- このことから、数式全体がCount型で演算されていることがわかります。Integerで計算してからCountに変換されているのではない。

```
> One + One + 1
Three
> 4
4
> 4 :: Count
Huh
```

```
> 1 + 2 - 1 :: Count
Two
> 1 + 3 - 1 :: Count
Huh
> 3 - 1 + 1 :: Count
Three
> 1 - 2 + 3 :: Count
Huh
```

# 再帰的なインスタンス定義

- 型クラスのメンバ関数定義の右辺では、通常の間数定義とおなじく、任意の式をつかうことができます。

```
> :{      「a がNumのインスタンスなら、Maybe a もNumのインスタンスだよ！」
*Main|    instance Num a => Num (Maybe a) where
*Main|      Just x + Just y = Just (x + y)
*Main|      _      + _      = Nothing
*Main|    :}      型Maybe aの足し算を、型aの足し算を用いて、定義しています
```

```
<interactive>:17:10: Warning:
  No explicit implementation for
    '*', 'abs', 'signum', 'fromInteger', and (either
'negate' or '-')
  In the instance declaration for 'Num (Maybe a)'
```

>

ところで、`:{ ... :}` を使うと、インタプリタに複数行を入力できます



# 再帰的なインスタンス定義

- Maybe Integerばかりか、Maybe (Maybe Integer)などもNumのインスタンスになりました！

```
> Just 4 + Just 9
```

```
Just 13
```

```
> Just 4 + Nothing
```

```
Nothing
```

```
> Just (Just 4) + Just (Just 1)
```

```
Just (Just 5)
```



# 演習問題

exercise-3-string-Num

- String型を、(+)演算子で文字列結合できるようにしてしまいましょう。
- ヒント: 型クラスNumのインスタンスにすればOK.

# Chapter 4

## 応用文法

# kind

- kind = 型の型。類、種類などともいう。
- :k コマンドで調べられる。

```
> :k Int
```

```
Int :: *
```

```
> :k []
```

```
[] :: * -> *
```

```
> :k (,)
```

```
(,) :: * -> * -> *
```

```
> :k Show
```

```
Show :: * -> GHC.Prim.Constraint
```

```
> :t 3
```

```
3 :: Num a => a
```

```
> :t []
```

```
[] :: [t]
```

```
> :t (,)
```

```
(,) :: a -> b -> (a, b)
```

# カリー化と関数の部分適用

- Haskellでは、関数の引数はつねに1個しかない
- $a \rightarrow b \rightarrow c$  は右結合  $a \rightarrow (b \rightarrow c)$
- 引数を1個ずつ渡して $\rightarrow$ がなくなると完成。
- 型レベル関数も同様。

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

```
> :t (3+)
(3+) :: Num a => a -> a
```

```
> :t (3+5)
(3+5) :: Num a => a
```

```
> (3+5)
```

```
8
```

```
> :k (,)
(,) :: * -> * -> *
```

```
> :k (,) Int
(,) Int :: * -> *
```

```
> :k (,) Int Char
(,) Int Char :: *
```

```
> (3, '五') :: (,) Int Char
(3, '\20116')
```

# 演算子と関数の相互変換

- 関数をバッククオートで囲むと中置演算子に、演算子を括弧で囲むと関数にできる。
- 括弧式で、一部を省略するとそういう関数になる。

```
> 100 `divMod` 7
(14, 2)
> (/3) 369
123.0
> :set -XtupleSections
> :t (,,)
(,,) :: a -> b -> c -> (a, b, c)
> (2,,4,,6) 3 5
(2, 3, 4, 5, 6)
```

# ガード

- `case`にBool式を追加できる構文糖衣。

```
case maybeAge of Just x | x >= 20 -> "OK!"
```

以下のような式と等価。

```
case maybeAge of Just x ->  
    case x >= 20 of True -> "OK!"
```

- `if`文も同様の構文に対応(MultiWayIf拡張)

```
if | age >= 20 -> maxBound  
   | age >= 16 -> 10000  
   | otherwise -> 5000
```



- `ghc`に `-ddump-ds` フラグを与えると脱糖された姿を出力させることができます。

# リスト内包表記

- リストから値を取り出したり、条件判定やパターンマッチをしながら新しいリスト値を作る式が書ける。

```
> let factors n = [x | x <- [1..n], n `mod` x == 0]
```

```
> factors 14
```

```
[1,2,7,14]
```

```
> let multi13 = [x | x <- [1..], 13 <- factors x]
```

```
> takeWhile (<100) $ multi13
```

```
[13,26,39,52,65,78,91]
```



# where

- let式と同様、局所的に変数を定義できるが、本体の後ろに定義を置ける構文。

```
huge :: Integer
```

```
huge = big * big
```

```
  where
```

```
    big = large * large
```

```
      where
```

```
        large = 10000
```

# 関数定義

- 型を別行で定義したり、パターンマッチと組み合わせたりできる。

```
factorial :: Integer -> Integer
```

```
factorial 0 = 1
```

```
factorial x | x < 0 = undefined
```

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



# Haskell頻出関数

- `id :: a -> a` 恒等関数

```
id x == x
```

- `const :: a -> b -> a` 定数関数

```
const x y == x
```

- `($) :: (a -> b) -> a -> b`

関数適用の演算子。優先順位が低いので括弧の代わりに。

```
> print $ sqrt $ 7 - 4
```

```
1.7320508075688772
```

- `(.) :: (a -> b) -> (b -> c) -> a -> c` 関数合成

```
> (*2) . (+100) . sqrt $ 9
```

```
206.0
```



# リスト頻出関数

完全なリストはData.Listモジュールを参照しよう！

- `map :: (a -> b) -> [a] -> [b]`

関数をリストのすべての要素に適用する

- `filter :: (a -> Bool) -> [a] -> [a]`

リストのなかから条件を満たすものだけを抽出する

- `reverse :: [a] -> [a]`

リストを逆順にする

- `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`

2つのリストを関数で噛みあわせる

- `foldr1 :: (a -> a -> a) -> [a] -> a`

リスト全体を二項演算子で1つにくっつける



# リスト頻出関数

- `minimum, maximum :: Ord a => [a] -> a`  
リストの中の最小・最大要素を探す
- `lines, words :: String -> [String]`  
文字列を行、列で分割する
- `repeat :: a -> [a]`  
無限リストを作る
- `take, drop :: Int -> [a] -> [a]`  
リストの先頭 $n$ 要素だけを採用 (または先頭 $n$ 個を落とす)
- `length :: [a] -> Int`  
リストの長さを返す



# 演習問題

exercise-4-brain-twister

- Scientists say that the brain recognizes words mainly by how they begin and end (which is why *Esilgnh* is *slitl pltcefrey lbilege wehn* you *wtrie it lkie tihs*.)
- 本当かな？
- 英語で書かれたテキストファイルを読み込んで、各単語の先頭と末尾以外の文字を逆順にしてしまうプログラムを作しましょう。

# Chapter 5

## モジュール

# モジュール

- コードを分割し、名前を管理する単位
- 1モジュールが1ファイルに対応
- 同じ名前の関数でも、異なるモジュールに置けば共存できる。むしろ、似たような機能にはあえて同じ名前をつけることが多い。
- Haskellの無数のライブラリはモジュールがあることで共存できる。



# モジュールの利用

- `import Data.List`

モジュール内のすべての識別子をインポート

- `import qualified Data.Map as M`

`Data.Map`内の識別子に`M.`をつけてインポート

- `import Data.Vector (Vector)`

特定の識別子だけインポート

- `import Prelude hiding ((++))`

特定の識別子だけを隠してインポート



# ライブラリ紹介: Data.Text

- Unicode文字列の処理に最適化された文字列ライブラリ。Preludeの文字列(Charのリスト)よりも高速に演算できる。

```
pack :: String -> Text
```

```
unpack :: Text -> String
```

```
replace :: Text -> Text -> Text -> Text
```

- Data.Textモジュールが見つからないと言われるとき -> **stack install text**
- GHC拡張のOverloadedStringsを有効にすると、文字列リテラルの型がIsString a => aになって便利。

# StringをTextへ変更

```
main :: IO ()
main = do
  name <- getLine
  putStrLn $ "Hello, " ++ name
```

```
import Data.Text      (pack)
import Data.Monoid    ((<>))
import Data.Text.IO   (getLine, putStrLn)
import Prelude hiding (getLine, putStrLn)
```

PreludeのgetLine, putStrLnを隠して、  
Data.Textから同名の関数をインポート

```
main :: IO ()
main = do
  name <- getLine
  putStrLn $ pack "Hello, " <> name
```

`pack :: String -> Text`      `(<>) :: Monoid m => m -> m -> m`

# StringをTextへ変更(2)

```
main :: IO ()
main = do
  name <- getLine
  putStrLn $ "Hello, " ++ name
```

```
import Data.Monoid  ((<>))
import qualified Data.Text      as T
import qualified Data.Text.IO  as T

main :: IO ()
main = do
  name <- T.getLine
  T.putStrLn $ T.pack "Hello, " <> name
  putStrLn "This is String"
```

# StringをTextへ変更(3)

```
main :: IO ()
main = do
    name <- getLine
    putStrLn $ "Hello, " ++ name
```

```
{-# LANGUAGE OverloadedStrings #-}
import Data.Monoid    ((<>))
import qualified Data.Text      as T
import qualified Data.Text.IO  as T

main :: IO ()
main = do
    name <- T.getLine
    T.putStrLn $ "Hello, " <> name
    putStrLn "This is String"
```



# 演習問題

exercise-5-1-fast-reverse

- Let us now carry out large-scale experiment on whether the brain recognizes words mainly by how they begin and end.
- 英語で書かれたテキストファイルを読み込んで、各単語の先頭と末尾以外の文字を逆順にしてしまうプログラムをData.Textを使って作しましょう。
- exercise-4が未完成の場合は、授業でつくった次ページの解を基にしてください。

# 上田直之さんの解を基にした exercise-4の参考回答

[https://github.com/naoyuky/learn\\_haskell/blob/master/exercise-4-brain-twister/twister.hs](https://github.com/naoyuky/learn_haskell/blob/master/exercise-4-brain-twister/twister.hs)

```
1 import System.IO (isEOF)
2
3 wordreverse :: String -> String
4 wordreverse xs
5     | length xs < 2 = xs
6     | otherwise = hajime ++ manaka ++ owari
7     where
8         hajime = take 1 xs
9         manaka = reverse $ drop 1 $ take (length xs - 1) xs
10        owari  = drop (length xs - 1) xs
11
12 main :: IO ()
13 main = do
14     strings <- getLine
15     putStrLn $ unwords $ map wordreverse $ words strings
16     f <- isEOF
17     if f then return () else main
```



# 練習問題

exercise-5-2-confuse-prelude (発展問題)

- confuse-preludeフォルダにごく普通の計算をするHaskellプログラムが入っています。
- 新しいモジュールを作り、Main.hsの中の+と\*の意味を入れ替えてください(+が乗算、\*が加算になるようにする)。Main.hsにはimport文しか追加してはいけません。
- 同じMain.hsで、通常のPreludeの+, \*演算子も使ってください。



# Chapter 6

## 型の作り方

# 型の作り方

- type

もとの型と互換して使える型の別名を作る。

- data

独自の型名、値コンストラクタを持った新しい型を作る。

- newtype

値コンストラクタが1つ、値の引数も1つしかない特別なdata。

# データ型の自作

```
data Person = Male String Int
            | Female String Int
            | Cat String Int Person
            value constructor argument type argument type argument type
```

- ユーザー定義のデータ型は、値コンストラクタと型の組み合わせで作る。
- 上記の例では、次のようなものがPerson型になる。

```
Male "nushio" 32
Female "miku" 16
Cat "Kuro" 5 (Male "nushio" 32)
```

# レコード構文

```
data Person = Human {name::String, age::Int}  
              | Cat {name::String, age::Int,  
                    kainushi::Person}
```

- データ型の要素の型に名前をつけてアクセスできる。
- レコード名は要素を取り出す関数、値コンストラクタはデータ型を作る関数。

```
> Human "nushio" 32  
Human {name = "nushio", age = 32}  
> :t Cat  
Cat :: String -> Int -> Person -> Person  
> :t name  
name :: Person -> String  
> let me = Human "nushio" 32  
> me{age = age me + 1}  
Human {name = "nushio", age = 33}
```

# 型クラスの自作

- 型クラスの作り方

```
class Greetable a where  
  greet :: a -> IO ()
```

- インスタンス宣言の仕方

```
instance Greetable Person where  
  greet (Male name _) = putStrLn $ "Hello, Mr. " ++ name  
  greet (Female name _) = putStrLn $ "Hello, Ms. " ++ name  
  greet (Cat name _ _) = putStrLn $ "Meow, " ++ name
```

- derivingを使うとインスタンスを自動導出できる。

```
data Cat = Cat String Int  
  deriving (Eq, Ord, Show, Read)
```



# 演習問題

exercise-6-1-data-Vec

- `data Vec a = Vec a a a`

のような3次元ベクトルのデータ型を作って、ベクトルとしての加算やスカラー倍を定義してみてください。

- 外積なんかはどうでしょう？
- `newtype Vec a = Vec [a]` で多次元ベクトルを作るなんてどうでしょう？



## 練習問題

- Personデータ型とGreetable型クラスの例を、なるべく中置演算子を使って書き直してみてください。
- ヒント: データ型、値コンストラクタ、型クラスはすべて大文字始まりの名前をつける必要があります。

# Chapter 7

## データ構造





# containers

- 頻出のデータ構造が揃っているライブラリ。
- グラフ、マップ、双方向 Queue、集合、ツリー



# vector

- ループ融合等の最適化が自動的に施される、高性能な配列ライブラリ。
- リストの演算はたいてい使える。

```
fromList :: [a] -> Vector a
```

```
generate :: Int -> (Int -> a) -> Vector a
```

```
map :: (a -> b) -> Vector a -> Vector b
```

```
filter :: (a -> Bool)
```

```
    -> Vector a -> Vector a
```

```
zipWith :: (a -> b -> c)
```

```
    -> Vector a -> Vector b -> Vector c
```



# 演習問題

exercise-7-nabe

- 鍋の美味しい季節です。

Data.Mapに鍋の具材と個数のデータ入れて、鍋の中身を管理するプログラムを作ってください。具の名前を入力すると(残っているならば)指定の具を一つ取り出します。鍋が空になったらメッセージを表示して終了します。

# Chapter 8

## 型クラス(2)

# Functor

```
> :info Functor
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b
```

- $a \rightarrow b$ と $f\ a$ を取って、 $f$ の中身の型 $a$ を他の型 $b$ に変える操作 $fmap$ を持つような型コンストラクタ $f$ が属する型クラス

ファンクターの型クラス則:

```
fmap id = id
fmap (f . g) = (fmap f) . (fmap g)
```

# Applicative

```
class Functor f =>
Applicative (f :: * -> *) where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

- fの中に入っている関数を、fの中に入っている値に適用できる型クラス

アプリカティブ則:

```
pure id <*> v = v
pure f <*> pure x = pure (f x)
u <*> pure y = pure ($ y) <*> u
u <*> (v <*> w) = pure (.) <*> u <*> v <*> w
```

# Monad

```
class Applicative m =>
Monad (m :: * -> *) where
    (>>=) :: m a -> (a -> m b) -> m b
    return :: a -> m a
    (>=>) :: (a -> m b) -> (b -> m c) -> a -> m c
    join :: m (m a) -> m a
```

- $a \rightarrow m b$  と  $b \rightarrow m c$  を合成して  $a \rightarrow m c$  が作れる  
ような  $m$  が属する型クラス

モナドの型クラス則:

$$\begin{aligned} \text{return} \gg f &= f \\ f \gg \text{return} &= f \\ (f \gg g) \gg h &= f \gg (g \gg h) \end{aligned}$$

# do記法

- do記法は>>=の糖衣構文  
左のプログラムは右に変換される

do

```
x <- getLine
y <- getLine
let z = x ++ y
putStrLn z
```

```
getLine >>= (\x ->
getLine >>= (\y ->
let z = x ++ y in
putStrLn z))
```



右側の式の型をチェックしよう。



# Monadって難しいの？

```
fmap    :: (a -> b) -> m a -> m b
(<*>)   :: m (a->b) -> m a -> m b
(>>=)   :: m a -> (a -> m b) -> m b
```

- 上記のような操作ができる型で、
- do記法という糖衣構文が用意されている  
という以上の深い意味はないです。
- 実践あるのみ！

# モナドの例: リスト

c.f. monad-family-tree.hs

$(\gg=) :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$

↓ リストに特化

$(\gg=) :: [a] \rightarrow (a \rightarrow [b]) \rightarrow [b]$

- aのリストがあつて、aのそれぞれに対して[b]のリストがあるなら、[b]のリストが作れる？

us :: [Hito]

us = ["村主", "亀山", "須永"]

parents :: Hito -> [Hito]

parents x = [x++"の父", x++"の母"]

# モナドの例: IO

c.f. world-state.hs

$(\gg=) :: IO\ a \rightarrow (a \rightarrow IO\ b) \rightarrow IO\ b$

- IOも、通常のnewtypeで作られた型なんです!

```
> import GHC.Types
> import GHC.Prim
> :info IO
newtype IO a = IO (State# RealWorld -> (# State# RealWorld, a #))
```

- State RealWorldをひとかたまりにし、#(unbox型の記号)を除いて読むと、IOの中身はただの関数。

`StateRealWorld -> (StateRealWorld, a)`

# モナドの例：構文解析器(Parser)

c.f. parser-sample.hs

```
(>>=) :: Parser a -> (a -> Parser b) -> Parser b
```

- Parserとは、文字列を受け取って、特定の構文規則にそって解釈するプログラム。
- たとえば、「自然数  $n$  があって、その後  $n$  個の単語があります」という文法は...

```
integer :: Parser Integer
```

```
n_words :: Integer -> Parser [String]
```

の2つをつかって

```
integer >>= n_words
```

と書ける!?

# IOモナドとは？

- 「副作用を持つ計算？」→すべての計算はメモリを書き換えるという意味では副作用を持つ。
- 処理系の自己判断で、実行順や実行回数が変わってしまっては困る計算が属するのがIOモナド
- 例：accelerate-cudaライブラリのrun関数は、呼び出すとGPUプログラムをファイルに生成したりするけど、IOじゃない。

`run :: Arrays a => Acc a -> a`



# 練習問題

もしもIOモナドがなくて、putStrLnの型がString -> ()だったら...

以下のプログラムは、どちらを先に表示するだろうか？

```
main :: [()]
```

```
main =
```

```
[print 1, print 2]
```

以下のプログラムは、文字列を何回表示するだろうか？

```
main :: [()]
```

```
main = [a, a]
```

```
where
```

```
a = putStrLn "hi"
```

# 様々なモナド

Maybeモナド、Errorモナド  
答えがないかもしれない計算

```
head [1] = Just 1, head [] = Nothing  
Left "404 not found"  
Right "<!doctype html><html><head>..."
```

Listモナド、Vectorモナド  
答えが複数ある計算

```
factors 14 = [1, 2, 7, 14]  
[(b, g) | b <- boys, g <- girls]
```

IOモナド  
副作用を伴う計算

```
getLine :: IO String  
putStrLn :: String -> IO ()
```

Stateモナド  
sという状態変数を読み書きしながら行う計算

```
class MonadState s m where  
  get :: m s  
  put :: s -> m ()
```

Parserモナド  
ソースコードから他の構造に変換する計算

```
spaces :: CharParsing m => m ()  
integer :: TokenParsing m => m Integer
```

# Facebookの内部処理言語Haxl

- 並列処理のためにApplicativeを使っている。





# 演習問題

exercise-8-1-safe-pred

- `pred :: Enum a => a -> a`は危険な部分関数です。

```
> pred True
```

```
False
```

```
> pred False
```

```
*** Exception: Prelude.Enum.Bool.pred: bad argument
```

```
>
```

- 安全なpred関数`predMay :: Enum a => a -> Maybe a`を用意しましたので、これを使って、ある値の「3つ前の値」、「n個前の値」を求める関数を作ってください。

```
pred3  :: Enum a => a -> Maybe a
```

```
predN  :: Enum a => Int -> a -> Maybe a
```



# 演習問題

exercise-8-2-learn-parser

- trifectaとparsersライブラリを使って、四則演算や括弧を含む式をパースし、評価してくれる簡易インタプリタ言語を作ってください。
- ヒント: `Text.Parser.Expression` モジュールの `buildExpressionParser` を使おう。

# Chapter 9

## 型クラス(3)

# Foldable

```
class Foldable (t :: * -> *) where  
    fold :: Monoid m => t m -> m  
    toList :: Foldable t => t a -> [a]
```

- $t$ の中のデータを1つに折りたためるような構造 $t$
- 実は、リストに対する関数(fold, maximum)のかなりの部分はFoldableに対して一般化されていたりする。

# Traversable

```
class Traversable (t :: * -> *) where
  traverse :: Applicative f =>
    (a -> f b) -> t a -> f (t b)
  sequenceA :: Applicative f => t (f a) -> f (t a)
```

- 構造 $t$ について、要素を計算する方法 $(a \rightarrow f b)$  がわかっているとき、 $t a$  をとって、全体を計算  $f (t b)$  してしまえる。
- $t (f a)$ を  $f(t a)$ に変換できる。



# 演習問題

exercise-9-traversable-Vec

- 以前に作ったVec型を、Applicative, Monad, Foldable, Traversableのインスタンスにしてみよう。
- IOモナドやParserモナドと組み合わせて使ってみよう。

# モナドトランスフォーマ

- あるモナドに別のモナドの機能を追加する

```
> :info MonadTrans
```

```
class MonadTrans (t :: (* -> *) -> * -> *) where  
  lift :: Monad m => m a -> t m a  
  -- Defined in `Control.Monad.Trans.Class`
```

## モナドクラス

- あるモナドの機能を使えるモナド

```
> :info MonadIO
```

```
class Monad m => MonadIO (m :: * -> *) where  
  liftIO :: IO a -> m a
```



# モナドトランスフォーマ

`StateT Int IO`という型について

- この型の類は？
- この型はMonad, MonadIOのインスタンスです。  
インスタンス宣言はどこにあるでしょう？
- この型はMonadState Intのインスタンスですね？  
ということは、getとputの型は何になるでしょう？

```
class Monad m => MonadState s m where  
  get  :: m s  
  put  :: s -> m ()
```





## 練習問題(Liveコーディング)

- 最近ディープラーニングというのが流行っているので、たくさんの画像を集めたい。
- 画像がたくさんありそうなWebページをどんどん探してきて、画像を取得するクローラを作るぞ。
- クローラの内部状態を管理したり、Webサイトを取得・解析するときの例外を処理したり、もちろんIOも必要なので、複数のモナドを組み合わせさせて使おう。



# Chapter 10

## ライブラリ紹介

Haskellは静的型付け(コンパイル時に型がつく)関数型言語だと思われるが……

- 動的型付けにもできる
- "Haskellは最高の手続き型言語です!" (モナド)
- オブジェクト指向もできる
- 自動定理証明もできます

数多くのプログラミングパラダイムは相反するものではない。それらを型というインターフェイスでつなぎ合わせられるglue言語がHaskell

# Haskellと動的型付け

## baseのData.Dynamicモジュール

- `toDyn :: a -> Dynamic`  
`fromDyn :: Dynamic -> a`  
`fromDynamic :: Dynamic -> Maybe a`  
`dynApp :: Dynamic -> Dynamic -> Dynamic`  
`dynApply ::`  
    `Dynamic -> Dynamic -> Maybe Dynamic`
- あらゆる型の値を、Dynamicという単一の型にしてしまえる。型がなかったら実行時エラーで落ちる。Haskellで動的プログラミングが可能に！
- 型検査できるMaybe版もあるよ。
- また別の手段として、-fdefer-type-errorsフラグをつけるとすべての型エラーがWarningに格下げになり、実行時まで遅延される

# ライブラリ紹介: spoon

`spoon :: NFData a => a -> Maybe a`

- 実行時エラーを起こすかもしれない式aのエラーを補足して、Maybeにできる。
- 危険な部分関数を使っているかもしれない処理を、安全にできる!
- 正格評価(遅延されている部分を全部計算してしまうこと)のしかたがわかる型`NFData a`が対象。

# Haskellとオブジェクト指向

- Haskellのレコード構文はあんまりいけてないので、様々なアプローチの解決が試みられている。

- lens

<https://hackage.haskell.org/package/lens>

```
> kuro . kainushi . age += 1
```

- record preprocessor

<https://github.com/nikita-volkov/record-preprocessor>

```
person :: Person
```

```
person = {!
```

```
    name = "Yuri Alekseyevich Gagarin",
```

```
    country = {! name = "Soviet",
```

```
                language = "Russian" }}
```

# Haskellと並列処理

- baseライブラリの[Control.Concurrent](#)  
軽量スレッド
  - <https://hackage.haskell.org/package/stm>  
software transactional memory。アトミックな操作を  
合成することで、ロックを使わず並行処理を記述する。
  - <https://hackage.haskell.org/package/parallel-io>  
お手軽並列処理 `parallel :: [IO a] -> IO [a]`
- ※Haskellで並列計算をするときは、`-threaded` オプションをつけてコンパイルし、実行時に`+RTS -N8`等で並列度を指定します。

# Haskellとストリーム処理

- Stsitneics say Esilgnh is slitl pltcefrey lbilege wehn you wtrie it lkie tihs.
- もし、処理したいデータが何TBもあったら？もし、Webから際限なくやってくるデータだったら？
- **pipes** - 無限データ処理(streaming)の決定版
- 文字コード変換、ネットワーク、メモリには収まらない大規模ファイルの処理などを組み合わせるのに使用
- ライブラリ自体にチュートリアルが同梱されている

<http://hackage.haskell.org/package/pipes-4.1.8/docs/Pipes-Tutorial.html>



# ライブラリ紹介: 高性能計算

- Repa: マルチコア配列計算ライブラリ

```
computeP :: Array r1 sh e -> m (Array r2 sh e)
```

- Accelerate: GPU計算ライブラリ

```
dotp :: Acc (Vector Float) -> Acc (Vector Float) -> Acc (Scalar Float)
```

```
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

```
run :: Arrays a => Acc a -> a
```

- Paraiso: GPU向けのシミュレーションコードを自動生成・自動最適化してくれるライブラリ

# ライブラリ紹介:trifecta

- 綺麗なエラーメッセージが出るパーザコンビネータライブラリ。

```
atom :: Parser SExpr
atom = Atom <$> some alphaNum
      <*> spaces
```

```
list :: Parser SExpr
list = do
  symbol "("
  ret <- many sexpr
  symbol ")"
  return $ List ret
```

# GUI, ゲームプログラミング

- 今回要望は多かったのですが・・・HaskellのGUIライブラリは充実しているとはいえません・・・。

[https://wiki.haskell.org/Applications\\_and\\_libraries/GUI\\_libraries](https://wiki.haskell.org/Applications_and_libraries/GUI_libraries)

- 各種グラフィックライブラリへのバインディングはひととおりあります: GTK, qt, SDL, OpenGL, X11, curses(?!)
- ghcjsの登場により、かなりのHaskellプログラムをjavascriptにコンパイルしてブラウザ上で走らせられるようになりました。
- Threepennyはブラウザをディスプレイとして使うGUIライブラリです。面白そう。

僕がHaskellを覚えたころに書いたMonadiusというのがありますが、ソースコードはとても汚いので参考にしないで下さい。絶対に見るなよ？<sup>123</sup>

# ライブラリ紹介: quickcheck

- ある型の値をランダムに生成して、ある性質を満たさない反例があるか探す。見つかったら、なるべく小さな反例を探してくれる。

```
class Arbitrary a where
  arbitrary :: Gen a
  shrink :: a -> [a]
```

```
class Testable prop where
  property :: prop -> Property
```

```
instance (Arbitrary a, Show a, Testable prop)
=> Testable (a -> prop)
```

```
> quickCheck (\x y -> x >= 2 && y >= 2 ==> x*y /= (7 :: Int))
+++ OK, passed 100 tests.
```

```
> quickCheck (\x y -> x >= 2 && y >= 2 ==> x*y /= (8 :: Int))
*** Failed! Falsifiable (after 6 tests):
```

```
4
```

```
2
```

# Haskellと物理量の計算

- <https://github.com/adamgundry/uom-plugin>

GHCの型推論プラグイン機能を使い、Haskellの型システムに物理量の型を追加する。

- 質量に長さを足し算するといった間違いが、コンパイル時に検出できるように！

```
-- Declaring some base units and derived units
[u| ft = 0.3048 m, kg, m, s,
N = kg * m/s^2  |]

-- An integer constant quantity with units
myMass = [u| 65 kg |]

-- A rational constant, this time with a type signature
gravityOnEarth :: Quantity Double [u| m/s^2 |]
gravityOnEarth = [u| 9.808 m/(s*s) |]
```

# ライブラリ紹介: sbv

<https://hackage.haskell.org/package/sbv>

- `prove :: Provable a => a -> IO ThmResult` 定理を自動証明する
- `allSat :: Provable a => a -> IO AllSatResult` 解をすべて見つける

```
> prove $ \(x :: SInt32) -> x + x .>= x
```

```
Falsifiable. Counter-example:
```

```
  s0 = 1073741824 :: Int32
```

```
> prove $ \(x :: SInteger) -> x + x .>= x
```

```
Falsifiable. Counter-example:
```

```
  s0 = -1 :: Integer
```

```
> prove $ \(x :: SInteger) -> x + x .== 2*x
```

```
Q.E.D.
```

```
> allSat $ \(x :: SReal) -> x^8 + x^7 .== -1
```

```
No solutions found.
```

```
> allSat $ \(x :: SReal) -> x^8 + x^7 .== 2
```

```
Solution #1:
```

```
  s0 = 1.0 :: Real
```

```
Solution #2:
```

```
  s0 = root(1, x^7+2x^6+2x^5+2x^4+2x^3+2x^2+2x = -2) =  
-1.3069899769252657... :: Real
```

```
Found 2 different solutions.
```

# Further Reading...

- State of the Haskell ecosystem

Haskellの最新ライブラリ情報を分野別にまとめた文書

<https://github.com/Gabriel439/post-rfc/blob/master/sotu.md>

- Typeclassopedia: Monad等型クラスの解説記事

<https://wiki.haskell.org/Typeclassopedia>

- International Conference on Functional Programming



# 演習問題

exercise-10-free

- 自分の気になるHaskellのライブラリを調べて使って、独自のHaskellプログラムを作ってみてください。
- この課題は他の課題との選択制とします。つまり、exercise1 ~ 9 を提出するか、exercise10のみを提出するか、どちらかにかまいません。





# 演習課題の提出方法

- learn-haskellフォルダ以下、各exercise-\* フォルダのTASK.mdファイルの指示を読み、プログラムを作ったり加筆したりしてください。learn-haskellフォルダ直下にSOLUTION.md (.txt, .pdf, ...)という文書を1つだけ作り、貴方の氏名、学籍番号、やったことの目録、解説、授業のフィードバックや感想などを記入して下さい。
- 「exercise1 ~ 9 の発展問題以外」か、exercise10か、のどちらかは解いてみてください。両方やっても構いません。



# 演習課題の提出方法

- exerciseを解くにあたってわからないことが出てきたら、冒頭にあげたコミュニティで質問してください。質疑応答のメール等、経緯が分かるものを提出物に同梱して下さったらそれも採点の対象に加えます。質問力はHaskell力の不可欠な一部ですし、コミュニティでのQ&Aは共有の財産となるからです！
- プログラムが完成しているかどうかよりも、プログラミングを上達するために外界に働きかける経験をつむことのほうが重要だと私は考えます。



# 演習課題の提出方法

- learn-haskellフォルダ全体を圧縮して1つの圧縮ファイルとし、manabaから提出してください。提出手順の詳細は亀山先生のメールを参照してください。
- もしmanabaの容量上限がある場合、各exercise-\*フォルダの隠しフォルダ(.stack-work/)など、再現に不必要なファイルを削除してみてください。
- ✕切は2月15日午前9時です。複数回の提出が可能だそうですので、余裕を持って入稿してください（最新版を採点します）。

# Haskellライブラリの使い方

1. 望みの型を返す関数(例: `IO HTML`)を探す。
2. 呼びたい関数に引数があれば、その型の作り方を調べる。
3. 未知の型がなくなるまで繰り返す。

## ghciの基本操作3つ

|          |                           |
|----------|---------------------------|
| 値を調べる    | > 式                       |
| 型を調べる    | > <code>:t</code> 型       |
| 型クラスを調べる | > <code>:info</code> 型クラス |

つつく

Happy Hacking!