趣味でやる

Haskellλ門1

～基礎文法編～

hiruishi

Haskellの雑な紹介

バグらずに沢山の機能を表現できるよ！

というか（慣れたら）ほかの言語で書くより簡潔に書けるよ！

プログラミングやったことのない人向け情報

Windowsの人はスタートメニューからWindows PowerShell、MacやLinuxの人はAppメニュー辺りから「端末」だとか「ターミナル」だとかを起動してください。

左側に今いるディレクトリ（Windowsだとフォルダ。ファイル閲覧してる時と同じ）があるので、**cd（ディレクトリ名）**で移動、**ls** で今いるディレクトリのファイルを一覧表示します。とりあえずこれから示すコマンドなどは**cd Documents**とか実行してから行うと無難です（コマンドやディレクトリの名前は**Tabキーで補完できる**のでバンバン押してください）。一個上のディレクトリに戻るときは **cd ../** で戻れます。

ついでに書いておくと、関数f(x)のxのことを引数（ひきすう）といいます。

環境構築

stack（コンパイラと外部パッケージの管理）、

intero（テキストエディタに作用してエラーとか表示するやつ）、

ghci（対話モード）

の三つをセットアップします。

環境構築1

--MacでHomebrewが入ってる--

brew install haskell-stack ; stack install intero ; stack ghci

実行後にCtrl+d（または:qと入力）で抜けられます（エラーが出た人は次頁）。

--Linux、またはHomebrewの入っていないMac--

curl -sSL https://get.haskellstack.org/ | sh ; stack install intero ; stack ghci

上と同様の方法で抜けられます（Macでエラーが出た人は次頁）

--Windows--

https://docs.haskellstack.org/en/stable/README/#how-to-install　から
64bitインストーラをダウンロードしてインストールしたのち、
powershellまたはコマンドプロンプトにて

stack install intero ; stack ghci

を実行、初回セットアップが終わったら上二つと同様の方法で抜けられます

環境構築2

macでxcodeのエラーが出る人は、

xcode-select --install

を実行してください。

その他詳細は

https://docs.haskellstack.org/en/stable/install_and_upgrade/

を参照してください。

エディタの拡張

- VisualStudioCodeならHaskero または Haskellyプラグインを導入。
- emacsなら Intero for Emacs (https://haskell-lang.org/intero 2019/02/25リンク更新)の

  指示に従って.emacsを編集（またリンク切れるかもしれないので**次頁参照**）。

  （初回セットアップが終わったら追加部分に関しては最後の一行以外コメントアウトしても大丈夫っぽい）

- vimの人は https://github.com/Fyrbll/intero-vim ,neovimの人は https://github.com/parsonsmatt/intero-neovim からプラグインを導入。

emacsの補足

Mac,またはLinuxについてはホームディレクトリの.emacsに以下を追記

```
;; If you don't have MELPA in your package archives:
(require 'package)
(add-to-list
'package-archives
'("melpa" . "http://melpa.org/packages/") t)
(package-initialize)
(package-refresh-contents)

;; Install Intero
(package-install 'intero)
(add-hook 'haskell-mode-hook 'intero-mode)
```

emacsについては基本マウスで操作しながら
F10キーでメニュー
Ctrl+xのあとにCtrl+sで保存
Ctrl+xのあとにCtrl+fでファイルを開く
Ctrl+xのあとにCtrl+cで抜ける
Alt+wでコピー、Ctrl+yで貼り付け
ができるって覚えていけばとりあえず困らないよ。

初回起動時以降は**最終行以外**の行頭に;を付ければ起動が早くなります・

最小限の構成

適当な場所に拡張子が".hs"のファイルを作ります（文字コードはutf-8）。

今回はBasic.hsで作ったとします（ファイル名の頭文字は必ず大文字！）。

そのディレクトリにて、

stack ghci Basic.hs を実行すると対話モードでBasic.hsの内容をテストすることができます。

~~なんでHello,world!しなきゃいけないんですか？~~

試しに 1 + 1 と入力してみましょう。

Main>1 + 1

2

stack ghciを電卓代わりにも使える（少なくとも筆者は使う）。

stack ghci （ファイル名を指定しないで実行もできる）

Prelude> 1 + 1 * 2

3

文法基礎編

型の定義と関数の定義で1セット！（とはいえ型の定義は省略も可能）

```
foo :: Int -> Int --型シグネチャ

foo x = x + 1    --関数の定義
```

Haskellの場合、型は単なる記憶領域確保のためのものではなく、数学における

「集合」と同じような意味を持つ（写像の始域と終域、覚えていますか？）。

ちなみに関数名は必ず小文字から、型の名前とファイル名は必ず大文字から！

（型や変数、ファイル名もすべてキャメルケースで書きましょう）

CamelCase や camelCaseのように単語の区切りを大文字にして区切ってください。

コメント

--と書くとそれより右側はコメント扱いになり、メモとしていろいろ書いてもエラーにならない。

{- と -}で挟んでも良い。この場合は複数行を一度にコメント扱いできる。

```
foo x = x + 1 -- コメントをここに書く

{-

ここにもコメントを書くことができる。関数の役割のメモなどにどうぞ

-}
```

Basic.hs

```
foo :: Int -> Int

foo x = x + 1

const1 :: Int -- 定数関数

const1 = 1
```

この内容を書き終えたらghciで :r を実行して再読み込み。

foo 2 や foo const1 を実行してみましょう（引数を括弧でくくる必要はない！）。

インデント(字下げ)について

pythonと同じくインデントの深さで範囲(例えば関数の定義を何行にわたって書いているか、C言語でいう{})が決定されるので必ずTabキーやSpaceでインデントを入れましょう。ちなみに通常の関数定義は字下げの深さ0です。

```
foo :: Int

    -> Int  -- :: と -> が揃うと見やすさがアップ

foo x =

        x +

        1

foo :: Int -> Int

foo x = x  +

1   -- foo と 1が同じ字下げ(この場合深さ0)だとエラーになる。
```

ちなみにEmacsのinteroだとTabキー押せば一発でそれっぽいインデントにしてくれる(VSCodeだとなぜかできなかった)

ghciの使い方

:t で型を見ることができる。

Main> :t foo

Int -> Int

Main> :t foo 1

Int

Main> :t const1

Int

ghciの使い方

:i で型や関数の定義を表示（このスライド中のわからない要素も:iや:tで調べましょう）

Main> :i Int

Main> :i map

:q（またはCtrl+d）でghciを終了

:h でヘルプ

ちなみに変数名や型名はTabキーで補完できます。

:l (ファイル名) で読み込み。

ghciの使い方、おまけ

いざghciで何か関数を定義したとき、型の定義を入れたらエラーになったと思います。そういうときは複数行定義の記号を使って

:{

（型の定義）

（関数の定義）

:}

とするとよいです。

関数を引数に取る関数とか

Main> :t map

(a -> b) -> ([a] -> [b])（aやbのように小文字で始まる型は「任意の型」という意味）

１変数関数を引数に取り、「リストからリストへの関数」にしてくれる。

Main>:t map foo

[Int] -> [Int] （[Int]でIntのリストという意味）

Main> map foo [1,2,3,4,5,6]（ちなみに、map foo [1 .. 6]という書き方もOK!）

……これってなんか２変数関数みたいじゃね？

2変数関数、3変数関数、……

bar :: Int -> Int -> Int

bar x y = x + y

bar 1 2 のように書く。

ちなみに内部的には

bar 1 :: Int -> Int の関数が生成した後、

(bar 1) 2 :: Int と関数が適用され、

最終的にInt型の値が出てくる（カリー化による部分適用）。

3変数関数も同じ理屈。

まぁ正直

bar :: Int -> Int -> Int の

前二つのIntが引数の型で、
最後のIntが関数の戻り値って
覚えておいても
そんなに困らないけどな！

2変数関数おまけ

bar 1　のように要求されているよりも少ない引数を渡すと残りの引数を引数としてとる関数になる（この場合Int -> Int）。(bar 1) 2 で3になる。

二項演算子もこの規則が適用されるので、

(+ 1)　と書けば+1してくれる無名関数になる。

Main> (+ 1) 2

3

ちなみに関数を二項演算子にしたいときは ` ` をつかう。

Main> 1 `bar` 2

無名関数

前ページの通り、多変数関数の引数を不足させたものも関数の扱いになる。

Main> (bar 1) 2

Main>:t (+)

Main> (+) 1 2

ラムダ式によって無名関数を作ることもできる。

bar2 :: Int -> Int -> Int

bar2 = \x y -> x + y　-- \x y -> x + y がラムダ式。1変数なら \x -> x + 1 とか。

データ型（直積型、あるいはデカルト積型）

直積集合は高校でやる積集合とは違うから気を付けてね！

```
data NingenSama =   --こっちは型コンストラクタ
  NingenSama        --ややこしいけどこっちはデータ（値）コンストラクタ
  { age :: Int
  , name :: String
  } deriving Show --この行はghciで画面に表示するのに必要
```

NingenSama 10000 “akachan” とするとNingenSama型の値になる。

直積型のおまけ

age は NingenSama -> Int の関数になる。nameは NingenSama -> String の関数。

age $ NingenSama 10000 “akachan”

$は文末までの括弧と同じと思ってよい（本当は違うけど）。

ちなみにageやnameを省略して

data NingenSama = NingenSama Int String　deriving Show

とやってもよいが、当然あったほうが便利だし何やってるかわかりやすい。

（データコンストラクタ NingenSamaの引数部分には定義の時には型を書くけど実際呼び出すときには値を書くから混乱するよ！　気を付けてね）

データ型(直和型)

```
data Tensuu = TensuuSuuji Int
            | NanrakanoJiko String   deriving Show
```

TensuuSuuji 98 も NanrakanoJiko “report hyousetsu ga bare ta” も同じTensuu型

直和型と直積型を組み合わせればいろいろ作れる。

```
data Zoo = Animals Int String | UchuKaraKitaAlien | ZooT Tensuu | ZooN NingenSama
```

いろいろやってみましょう。

NingenSama型やTensuu型をBasic.hsにて定義し、

ghciで動作を確認してみましょう(:rで再読み込み)。

余談……型に別名を付けるだけならデータ型を使わず typeを使う

実際の例 : type String = [Char]

標準で定義されているデータ型

タプル

(a,b) -- aとbは任意の型

(1,”abc”)などの値が作れる。

リスト (aは任意の型)

```
data [a]

  = []  |  a : [a]
```

:という二項演算子のデータコンストラクタが定義されていて、先頭から追加できる。例えば[1,2,3,4,5]は1:2:3:4:5:[] と同じ。

Maybe a 型

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

aは任意の型。
エラー処理に使ったりする。
Maybe IntならIntが入っている(Just 1など)かもしれないが、エラーにより何も入っていない (Nothing)かもしれない、という用法。

ユニット型

()という型で、()という値しか入らない。 ほかの言語におけるvoid型と同じ使い方をするけど、本来の意味は違う。

分岐処理

if文

```
baz :: Bool -> Int

baz b =

  if b

  then 1

  else  0
```

（なんかあんまり使った記憶がない ……後述のガード文のほうが便利）

分岐処理

ガード文（数学で見たことあるやつ）

```
reLU :: Int -> Int  -- ニューラルネットワークの活性化関数として使われる

reLU x
    | x < 0 = 0
    | otherwise = x
```

otherwiseはghciで調べるとTrueと同じ意味。なので上の行でマッチしなかったら

ここで必ずマッチする。

分岐処理

パターンマッチ（データ型を利用した分岐とかに使える）

```
anzen :: Zoo -> Bool
anzen z = case z of
  ZooN (NingenSama a n)  -> if a > 10000 then False else True
  _ -> False
```

nは定義したけど使わなかったので _（ワイルドカードパターン）を使って

ZooN (NingenSama a _) ->　　みたいに書くとよい（中の値はメモリから消される）。

case文の条件に適当な変数名や _を持ってくると必ずマッチするので、この場合 NingenSama a nに

マッチしなかったら必ず Falseになる。

分岐処理（併用）

パターンマッチとガードの併用（さっきはif文と併用したけど、ガードで書き直してみる）

```
anzen :: Zoo -> Bool
anzen z = case z of
  ZooN (NingenSama a _) -- ワイルドカードパターンのところは使わない
        | a > 10000 -> False  --ガードとの併用
        | otherwise -> True
  _ -> False                  --ガードを併用しないとき
```

分岐処理　例題

データ型 data Zoo' = Animals' Int String | UchuKaraKitaAlien' deriving Show

が与えられているときに、anzen'関数 (anzen' :: Zoo' -> Bool)を

自分で定義してみましょう。

if文は使わなくていいけど、パターンマッチ（case文）と

ガード文は必ず使いましょう。

(' は定義が被らないようにつけたけど、Basic.hsに

さっきのZooの定義を書いてなかったらつけなくていいよ)

(ていうかふつうはanzenじゃなくてisSafeとかのほうがわかりやすい気がする .....)

再帰による繰り返し処理

ここから2つの例題については後で述べる参考文献を参照

```
factorial :: Integer -> Integer
factorial = go 1
  where
    go a n
      | n <= 0    = a
      | otherwise = go (n * a) (n - 1)
```

Int型はほかの言語同様上限と下限があるが、Integerにはない。

ちなみにfactorialは「階乗」な。

--局所定義。ここに書いた関数goはfactorialの定義以外で参照できない。

goの第一引数は蓄積引数といって、現時点での正しい計算結果を記憶する。

再帰による繰り返しとリスト

Basic.hsの先頭に

import Data.Char (digitToInt)

を追加してください。

digitToInt :: Char -> Int

Main> digitToInt ‘a’

再帰による繰り返しとリスト – 前提知識

文字列型StringはCharのリスト[Char]と同じ。

[Char]型の値は、

[] または

a : b : c : ・・・ : [] (これを[a,b,c, ・・・ ,z]と書く)

（ただし変数a,b,c,...には何か文字が入っているとする）

例えば文字列”abc”(=[‘a’ , ’b’ , ’c’])は、

‘a’ : ‘b’ : ‘c’ : []

遅延評価
(take関数はリストの先頭から指定した数だけ持ってくる）
take 10 [1 ..] のように無限リストを使った式でも、リストの中身は要求されるまで生成されないのでメモリがいっぱいになったりしない。

↓

なんでこんなめんどくさい定義になってるかというと、遅延評価をしたいから。

再帰による繰り返しとリスト

16進数の数値を表す文字列 を数値（数値）に直す関数readHex’を作ってください。

readHex’ :: [Char] -> Int

readHex’ = undefined -- undefinedと書くことで定義をまだ書いていない関数について、エラーを出さないようにできる。

〈ヒント〉: さっきの facotrialを参考に。

〈ヒント〉: 変数xxsに[Char]型の値が入っている場合のパターンマッチ

```
case xxs of
  x : xs -> undefined  -- xはxxsの先頭（Char型）、xsはその残り（[Char]型）
  [] -> undefined -- リストが空っぽ、つまり全部評価し終えた時の処理。
```

補足：undefinedについて

前頁に書いた通り、undefinedと書くとまだ定義されていない部分についてエラーを出さないままにできる（ただしそのまま放っておくと実行時にエラーになる）。

畳み込み関数を使った繰り返し

Main> :t foldl

foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b

foldl :: (b -> a -> b) -> b -> [a] -> b として使うことができる（Foldableについては次回やります）。

foldl（今の値と新情報による更新処理）（初期値）（新情報のリスト）みたいに使う。

readHex'' :: String -> Int をfoldlを使って定義してみましょう。

参考文献

haskell-tiny-intro （マイコンクラブOBの人が書いた教材）

https://github.com/khibino/haskell-tiny-intro/blob/master/exercise/Basic.hs

baz、factorical と readHex’の例題等ここから持ってきました。

すごいHaskellたのしく学ぼう！ オーム社

めちゃんこ分厚いけどわかりやすいと評判の本、

私はちょっとしか読んでないけど。

次回予定

趣味でやるHaskell2　～実用編～

・hoogleを使ってみよう

・型クラス

・入出力と逐次処理

・リストによるループ処理の簡略化などTips

・プロジェクトの立ち上げとビルド

・外部パッケージの導入、代表的なパッケージの紹介