

# HaskeIIプログラミングチュートリアル

## Table of contents

1	純粋関数型言語HaskeII	2
2	HaskeIIキホンのキホン	2
2.1	まずは、おおまかな字句構造から：	2
2.2	HaskeIIプログラムを構成するものたち(1)：関数定義と型宣言	3
2.3	HaskeIIプログラムを構成するものたち(2)：型定義	5
3	そのほか必要なこと	8
3.1	let, where -- ローカルな関数の定義	8
3.2	抽象による無名関数	8
3.3	関数を書くには!? HaskeIIにおけるプログラム記述の慣習	9
4	型クラス：関数オーバーロードでさらなる利便性	9
4.1	オーバーロード：型ごとに違う振る舞いをする関数を定義したい	9
4.2	Eq:同値演算子を提供する型クラス	10
4.3	クラス拡張で型クラスを継承	11
5	おわりに	11

## 1. 純粋関数型言語Haskell

Haskellは、次のような特徴をもった関数型言語です：

- ・ 強力な型システム
  - ・ パラメータ多相のサポート
  - ・ 型クラスによるアドホック多相
- ・ 非正格で純粋な関数型言語
  - ・ 関数は非正格
  - ・ 遅延評価：無限リストなどのデータ構造の扱いが容易
  - ・ 純粋：参照透明性が保存され、副作用がない
- ・ モナドを用いたプログラミング

今回は、Haskellプログラムの基本要素である**関数**と**型**について、順を追って解説していきたいと思います。

## 2. Haskellキホンのキホン

### 2.1. まずは、おおまかな字句構造から：

- ・ コメントの書き方については、C、JavaやMLのそれと変わりません。
- ・ それぞれの関数定義や宣言は、セミコロン ';' で区切られ、ブレース '{}' でグループ化されます。
- ・ しかし、Haskellのレイアウトという構文により、セミコロンを省いた記述が可能です。いまのところ、行頭に文字があったらその行から新しい宣言くらいに思っておけば大丈夫です。

```
-- '--' 以降はコメント

{- 複数行のコメントは
   こんな風に入ります -}

-- 基本的に、トークンの区切りになる任意の場所で改行できる。
-- ただし、2行目以降は1行目から1文字以上、字下げすること。
fact n = if n==0
  then 1
  else n * fact (n-1)

add x y = x+y -- 一列目に 'a' が来たのでここから新しい宣言

f x =
  let g y = y*y -- let, where, of は特別；次に始まる文字(ここではg)が
    h z = z*2 -- 新しいグループの開始カラムになる
  in g x / h x -- 開始カラムより字下げが浅くなればグループ終了
```

## Haskell プログラミングチュートリアル

は、コメントを省くと以下と同じです（現在のところ、letなどは気にする必要はありません。） Haskellプログラムの文はセミコロンで区切られ、レイアウトによりセミコロンを省略する記法が提供されているということです。：

```
;fact n = if n==0
  then 1
  else n * fact (n-1)

;add x y = x+y      -- 開始カラムに文字があれば、直前に; を挿入

;f x =
  let {g y = y*y    -- let, where, of の直後に{を,
      ;h z = z*2    -- 開始カラムに文字があれば、直前に; を挿入
      }            -- 字下げが浅くなったので}を挿入
  in g x / h x
```

### Note:

Haskellでは、タブ文字は8文字の空白として処理されます。レイアウトで正しくインデントされているように見えるプログラムでも、意図したように解釈されずエラーになることがあります。注意してください。

## 2.2. Haskellプログラムを構成するものたち(1)：関数定義と型宣言

### 2.2.1. すべての基礎：関数定義

Haskellプログラムはいくつもの等式による関数の定義から成り立ちます。

```
-- 値も関数も第一級の対象(first class object) として同様に扱える
i = 1

-- 関数名に続き、空白で区切られた仮引数のリスト、 '=', 関数本体の式が続く
double x = x*2
add x y = x+y

-- よくある階乗の例； 再帰関数にもletrecのような特別扱いはありません。
fact n = if n==0
  then 1
  else fact (n-1)

-- パターンマッチングの例。文字列の長さを返す関数 'len' を定義してみる
-- 文字列はキャラクタのリスト； ':' は 'cons', "" は空文字列
len (s:ss) = 1 + len ss
len ""     = 0

-- 新しい中置記法の演算子を導入することもできる。 やりすぎると可読性が下がる :- (
x +* y = (x + y) * y
```

'=' の左辺に関数名と仮引数のリスト，右辺に関数本体の式を書きます．  $x+y$  などの演算子については他の言語と同様に直感的なのですが，手続き型における「関数呼び出し」は `add 3 4` のように，括弧を省いて，空白で実引数を区切って並べます (**関数適用**)．関数適用は他の演算子より強く結合します．ですので，`fact (n-1)` のように 必要に応じて括弧で囲んで書きましょう．

**Note:**

値や関数の名前(識別子)は，小文字で始まらなければなりません．例： `func`, `aLongNameFunc`, `fooBarBaz`, `anotherLongLongInt...` (大文字で始まる名前は，後述するデータ構築子のために用います．)

**2.2.2. 型推論と型宣言：型宣言を効果的に使ってバグがないプログラムを！**

Haskell の値はすべて単一の型を持ちます．C や Java などと同様に，関数の引数に誤った型の値を与えたり，文字列に1を足そうとするような誤りは，すべて型エラーとして報告されます．

ある値 `a` がある型 `T` を持つとき，Haskell では `a :: T` のように書きます．これを **型シグネチャ宣言** と呼びますが，Haskell では **型推論** により自動的に型が推論されますので，大抵の場合必要ありません．

しかし，適切な型宣言を書いておけばプログラムの可読性が向上し，またバグをコンパイル時に発見する良い手助けになります．

**Note:**

型の名前は，大文字で始まらなければなりません．例： `Tree`, `Address`, `Person`, `UF0`, `Dog`, ... しかし型と値の名前空間は別個のもので，名前が衝突することはありません．

```
i :: Int
i = 1 -- 関数の型は (引数) -> (戻り値) のように書く

fact :: Int -> Int
fact n = ... (略)

len :: String -> Int
len (s:ss) = ... (略)

-- 多引数の関数は，(引数1) -> (引数2) -> ... -> (戻り値)
add :: Int -> Int -> Int
add x y = x+y

-- 高階関数: -> は右結合なので () で囲んで関数型を区別
map :: (a -> b) -> [a] -> [b]
map f [] = []
```

```
map f (x:xs) = f x : map f xs
```

### 2.2.3. 多相型って？

ここでは、Haskellプログラムの再利用性を促進する多相型についてお話しします。

例えば、リストのような型には、長さを求めたり、特定の位置の要素を取り出したり、並べ替えたりと、要素(中身)の型に依存しない操作が多くあります。

```
head [1,2,3,4]
  ==> 1 :: Int
head ["Jan", "Feb", "Mar"]
  ==> "Jan" :: String
```

そのため、リストは**要素の型が全称修飾された型** `[a]` として定義されています。(ここで`a`は**型変数**といい、具体的な型と区別されます。)

#### Note:

この多相性を「**パラメトリック多相**」といい、C++のtemplateやJava 5.0のGenericsなどもこれにあたります。Java 5.0では、Haskellなどと同様に `List<String>` としてリストの要素の型を明示することができます。

## 2.3. Haskellプログラムを構成するものたち(2)：型定義

### 2.3.1. 標準ライブラリで定義されている型

Haskell言語と標準ライブラリで提供されている型には、例えば

- `Int`(固定長の整数型)
- `Integer`(任意長の整数型)
- `Float`(浮動小数点数)
- `Char`(文字型)
- `String`(文字列型；`[Char]`と同じ)
- `Bool`(真偽値)
- `[a]`(リスト)
- `(a,b)`, `(a,b,c)`, `(a,b,c,d)`, ... (タプル)

...などがあります。といっても、これらと、次に紹介するユーザが定義した型はほとんど変わりません。特別に`325`, `1.414`, `'c'`, `"string"`などの文字列や数値リテラルがこれらの型に属していることを除けば、ユーザ定義の型と区別はありません。

### 2.3.2. data宣言によるデータ型定義

### 2.3.2.1. まずは単純な列挙型

Haskellでは、`data`宣言によりユーザ定義のデータ型を定義することができます。例えば、amazonで売っている商品の種類を表す型 `Goods` を次のように定義してみます。

```
data Goods =
  CD | Book | Video |
  Toy | Furniture | Electronics
```

上の**型定義**で、型`Goods`は値 `CD`, `Book`, `Video`, ... を持つ、と定義している事になります。この`CD`や`Book`などを**データ構築子**と呼びます。これらはパターンマッチングで照合させることができます：

```
toString :: Goods -> String
toString CD    = "CD"
toString Book  = "Book"
toString Video = "Video"
...
```

パターンマッチングは引数のほか、`case ... of`**構文**でも可能です。例えば、上の `toString` は次のように書いても同じです：

```
toString s =
  case s of
    CD    -> "CD"
    Book  -> "Book"
    Video -> "Video"
    ...
```

#### Note:

上で見たように、**データ構築子の名前は大文字で始まらなければなりません**。データ構築子も値ですが、後で見るように関数の様にも扱うことができます。通常の間数や値とデータ構築子を区別するのは、パターンマッチによる照合が可能なことと、識別子が必ず大文字で始まることです。

### 2.3.3. レコード型をためそう

もちろん、ある型の値が他の型の値を含むような、いわゆるCの構造体のような型も定義することができます。個人アドレス帳のエントリを表す型`Person`を定義します。ひとつのデータ構築子`MkPerson`を与え、これは管理`id`を表す 整数、名前、住所からなる、とします。前の例とは違い、データ構築子に続き構成要素の型名が続きます。

```
data Person = MkPerson Int String String
```

データ構築子MkPersonは 型 `Int -> String -> String -> Person`を持つ関数と見なすこともできます。型Personの値に対するパターンマッチングは次のようにします：

```
-- '++' はリスト連結 (つまり文字列連結) 演算子,  
-- showはInt型の値を文字列に変換する関数  
showPerson :: Person -> String  
showPerson (Person id name addr) =  
  "id:" ++ show id ++ " name:" ++ name ++ " address:" ++ addr
```

また、データ構築子の構成要素に名前を付けて直接アクセスすることもできます。

```
data Person2 = Person2 {pid::Int, pname, paddress::String}
```

Cの構造体やJavaのクラスのフィールド名に近いです。これを**フィールドラベル**と呼び、関数として扱います。(pid::Person->Int, pname::Person->Stringのように型付けされます。) 例えば `pname (Person2 5 "Michael" "somewhere")` ==> "Michael"となります。

### Note:

また、型とデータ構築子の名前空間は異なるため、型とデータ構築子の名前を同じにすることがよくありますが、上の例で = の右辺の Person2はデータ構築子、IntやStringは型であることに注意してください。

### 2.3.4. 多相型・再帰型をつくろう

多相型も定義することができます。例えば、型aの2項組で平面上の座標 (x,y)を表す型 Point aは：

```
data Point a = Pt {x,y :: a}
```

一方、再帰的かつ多相的な型の例として、型aを枝と葉にもつ2分木を表す型 Tree aは次のように定義できます：

```
data Tree a =  
  Branch a (Tree a) (Tree a)  
  | Leaf a
```

Tree は型を一つ引数に取り、完全な型となります。型の識別子は**型構築子**とも呼ばれます(無引数のものも含めて)。a は**型変数**で、型構築子に適用される任意の型引数を表します。型引数は = の右辺に現れることができます。

多相型の代表的な例としてリストがありますが、ほかに関数型 ( $\rightarrow$ ) も 2 引数の型構築子です。それぞれ 型式を  $[] a$  や  $(\rightarrow) a b$  と書くこともできます。

### 3. そのほか必要なこと

#### 3.1. let, where -- ローカルな関数の定義

let whereを用いて、ある関数のスコープの範囲内のみで有効な関数を定義することができる。例：

```
func x = let g y = ...
          h z = ...
          in (関数の本体)

func x = (関数の本体)
  where
    g y = ...
    h z = ...
```

#### 3.2. 抽象による無名関数

抽象を用いて、無名の関数をその場で作り適用させる、といったことが可能である。記法は、 $\$x \rightarrow x+1$  のように、 $\$$ の次に仮引数を置き、 $\rightarrow$ に続き式を書く。例えば上のaddは、次のように書くこともできる：

```
add :: Int -> Int -> Int
add = \$x -> \$y -> x+y
```

ほかに、例えば、引数を取るデータ構築子に対し、次のような定義も可能である。これを**パターン束縛**という。パターン束縛により値を抽出するだけのcase文を簡潔に書くことが出来る：

```
(id,name) = (100, "Bob")
```

この2つを組み合わせ、値を展開しつつ無名関数をつくる、といったことも可能だ：

```
-- getPerson :: Int -> Person とする

showPersonName :: Int -> String
showPersonName id = (\(Person _ name _) -> name) getPerson id

-- タプル値を取り出して使うのに良い：
(\(a,b) -> a+b) (10, 63) ==> 73
```

### 3.3. 関数を書くには!? Haskellにおけるプログラム記述の慣習

標準プレリュードに定義されている関数はどんどん使おう。例えば関数適用演算子(\$):

```
infixr 0 $
($) :: (a -> b) -> a -> b
f $ x = f x
```

一見ほとんど「何もしない」関数に見えるが、結合性宣言 `infixr`により優先レベルが最低に設定されているため、これを用いて余分な括弧を省くことができる:

```
f (g (h (l++r)))
==
f $ g $ h $ l++r
```

他に `flip`, `curry`, `uncurry`, `zip`, . などを用いると、タプル型や関数合成を駆使し、不要な `let` 束縛や 抽象を削除することができる。以下の関数がどのように振る舞うのか、皆さん考えてみてください:

```
flip :: (a -> b -> c) -> b -> a -> c
flip f = \b -> \a -> f a b

curry :: ((a, b) -> c) -> a -> b -> c
curry f = \a -> \b -> f (a,b)

uncurry :: (a -> b -> c) -> (a, b) -> c
uncurry f = \ab -> f ab

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

zip :: [a] -> [b] -> [(a, b)]
zip (a:as) (b:bs) = (a,b) : zip as bs
zip _ _ = []
```

## 4. 型クラス: 関数オーバーロードでさらなる利便性

### 4.1. オーバーロード: 型ごとに違う振る舞いをする関数を定義したい

Java言語では、全ての親クラスであるObjectに、等価性を判定する `equals` や文字列表現を返す `toString`が定義されています。プログラマはそれぞれのクラスに合った等価性判定や文字列表現を、各メソッドをオーバーライドすること

で定義することができます。

Note:

オーバーロードは関数(メソッド)や演算子の**多重定義**を、オーバーライドは、スーパークラスで定義されたメソッドをサブクラスで**再定義**すること、とされています。

HaskeIIで同様のことは可能でしょうか？例えば、

```
data Color = Red | Green | Blue

equals :: Value -> Value -> Bool
equals Red Red = True
equals Green Green = True
equals Blue Blue = True
equals _ _ = False -- _ は 関数内で出現しない仮引数に用いることができる。

equals :: Tree Color -> Tree Color -> Bool
equals (Branch v a b) (Branch v' a' b') =
    equals v v' -- この equals は Color -> Color -> Bool
    && equals a a' && equals b b' -- この equals は再帰呼び出し
equals (Leaf v) (Leaf v') =
    equals v v' -- この equals も Color -> Color -> Bool
equals _ _ = False
```

上のようにある関数に複数の型を割り当てることは**できません**。HaskeIIでは、関数を多重に定義することはできないのでしょうか？同じ等価性を表す関数に、いちいち違う名前を付けなければいけないのでしょうか？

答えは Yes です。 **型クラス**を使ってこれを実現します。

型クラスは**型クラス定義**と**インスタンス宣言**からなります。直観的には、

- ・ 型クラス定義により、クラスの **名前** と **多重定義したい関数 (メソッド)** の型シグネチャを定める。
- ・ **インスタンス宣言**により、
  1. 型がそのクラスに属することを宣言し、
  2. メソッドの実装を与える。

という方法で、一つの関数を複数の型に対し定義することができます。では、標準ライブラリにある具体的な例を見ていきましょう。

#### 4.2. Eq: 同値演算子を提供する型クラス

先ほどの例を、型クラスを用いて表現してみましょう。標準プレリュードで、型クラス Eq は以下のように定義されています：

```
Eq = ...
```

## Haskell | プログラミングチュートリアル

```
class Eq a where
  (==) :: a -> a -> Bool
```

aは型変数で，インスタンスとして宣言する型がこれにあたります．次に，型 Color および Tree Color についてインスタンス宣言 を示します：

```
instance Eq Color where
  Red   == Red   = True
  Green == Green = True
  Blue  == Blue  = True
  _     == _     = False

instance Eq (Tree Color) where
  (Branch v a b) == (Branch v' a' b') =
    v == v' -- この == は Color -> Color -> Bool
    && a == a' && b == b' -- この == は再帰呼び出し
  (Leaf v) == (Leaf v') =
    v == v' -- この == も Color -> Color -> Bool
  _ == _ = False
```

しかしTree aは任意の型の上で同値性を定義できそうです．そこで，上の Tree Colorに代わって，

```
instance (Eq a) => Eq (Tree a) where
  ... (同様)
```

と定義することで，Eqのインスタンスである任意の型a について同値性を定義することができます．ここで (Eq a) => の部分を **文脈** といい，型変数aが満たすべき **制約** を示しています．

### 4.3. クラス拡張で型クラスを継承

一方， Haskell では全順序関係をチェックするためのメソッドを提供する型クラス Ord も提供しています．全順序なので，Eqクラスの==演算子も備えているべきですが，こういった場合型クラスを**継承**することができます．

```
class (Eq a) => Ord a where
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min           :: a -> a -> a
```

ここでEqはOrdの**スーパークラス**である，とといいます．Eqで可能な操作はすべて Ordで可能ですが，いっぽう， Ordのインスタンスになる型はEqのインスタンスでなければならないことも示しています．

## 5. おわりに

いかがだったでしょうか？ HaskeIIプログラムの基本的な書き方，型の概念について解説しました．次回は，これらを用いてどんなプログラムが書けるのか，また純粋関数型で非正格であることを利用した無限のデータ構造や，いかにHaskeIIが現実世界の副作用の記述を可能にしているのか，など，まだまだ面白いお話はたくさんありますので，そちらに焦点を当てて解説したいと思います．どうぞお楽しみに！