Haskell Notes

目次

1 Haskell について 2
2 変数・関数・型 2
2.1 変数 2
2.2 変数の型 3
2.3 関数 3
2.4 有名な関数 4
2.5 関数の型 4
2.6 関数合成 4
2.7 IO サバイバルキット 1 5
3 Test Part 5

1 Haskell について

本書はプログラミング言語 Haskell の入門書である．それと同時に，本書はプログラミング言語を用いた代数構造の入門書でもある．プログラミングと代数構造の間には密接な関係があるが，とくに関数型プログラミングを実践する時にはその関係を意識する必要が出てくる．本書はその両者を同時に解説することを試みる．

これからのプログラマにとって Haskell を無視することはできない．Haskell の「欠点をあげつらうことも，攻撃することもできるが，無視することだけはできない」のだ．それは Haskell がプログラミングの本質に深く関わっているからである．

Haskell というプログラミング言語を知ろうとすると，従来のプログラミング言語の知識が邪魔をする．モダンで，人気があって，Haskell から影響を受けた言語，たとえば Ruby や Swift の知識さえ，Haskell を学ぶ障害になり得る．ではどのようにして Haskell の深みに到達すればいいのだろうか．

その答えは，一見遠回りに見えるが，一度抽象数学の高みに登ることである．

と言っても，あわてる必要はない．

近代的なプログラミング言語を知っていれば，すでにある程度抽象数学に足を踏み入れているからである．そこで，本書では近代的なプログラマを対象に，プログラミング言語を登山口に抽象数学の山を登り，その高みから Haskell という森を見下ろすことにする．

ところで，プログラムのソースコードは現代でも ASCII 文字セットの範囲で書くことが標準的である．Unicode を利用したり，まして文字にカラーを指定したり，書体や装飾を指定することは一般的ではない．たとえば変数 `a` のことを a と書いたり $\boldsymbol{a}$ と書いたり $\tilde{a}$ と書いたりして区別することはない．

Haskell プログラマもまた，多くの異なる概念を同じ貧弱な文字セットで表現しなければならない．これは，はじめて Haskell コードを読むときに大きな問題になりえる．たとえば Haskell では `[a]` という表記をよく扱う．この `[a]` は `a` という変数 1 要素からなるリストのこともあるし，`a` 型という仮の型から作ったリスト型の場合もあるが，字面からでは判断できない．もし変数はイタリック体，型はボールド体と決まっていれば，それぞれ $[a]$ および $[\boldsymbol{a}]$ と区別できたところである．

本書は，異なる性質のものには異なる書体を割り当てるようにしている．ただし，どの表現もいつでも Haskell に翻訳できるように配慮している．実際，本書執筆の最大の困難点は，数学的に妥当で，かつ Haskell の記法とも矛盾しない記法を見つけることであった．

2 変数・関数・型

2.1 変数

変数 x に値 1 を代入するには次のようにする．

$$x = 1 \tag{1}$$

- Haskell では

```
x = 1
```

と書く．

変数という呼び名に反して，変数の値は一度代入したら変えられない．そこで変数に値を代入するとは呼ばずに，変数に値を束縛するという． 式 1 の右辺のように数式にハードコードされた値をリテラルと呼ぶ．

リテラルや変数には型がある．型は数学者の集合と似た意味で，整数全体の集合 $\mathbb{Z}$ に相当する整数型や，実数全体の集合 $\mathbb{R}$ に相当する浮動小数点型がある．整数と整数型，実数と浮動小数点型は異なるため，整数型を **Int** で，浮動小数点型を **Double** で表すことにする．

- Haskell では **Int** および **Double** をそれぞれ `Int` および `Double` と書く．

数学者は変数 x が整数であることを $x \in \mathbb{Z}$ と書くが，本書では $x :: \mathbf{Int}$ と書く．これは記号 $\in$ を別の用途に用いるためである．

- Haskell では $x :: \mathbf{Int}$ を

```
x :: Int
```

と書く．

本書では変数名を原則 1 文字として，イタリック体で表し w, x, y, z のような n 以降のアルファベットを使う．

変数の値がいつでも変化しないことを参照透過性と呼ぶ．プログラマが変数の値を変化させたい，つまり破壊的代入を行いたい理由はユーザ入力，ループ，例外，内部状態の変化，大域ジャンプ，継続を扱いたいからであろう．しかし，後に見るようにループ，例外，内部状態の変化，大域ジャンプ，継続に変数の破壊的代入は必要ない．ユーザ入力に関しても章を改めて取り上げる．参照透過性を強くサポートするプログラミング言語をを関数型プログラミング言語と呼ぶ．

2.2 変数の型

TK

$$x :: \mathbf{Int} \tag{2}$$

$$x = 1 \tag{3}$$

$$x :: \mathbf{Int} = 1 \tag{4}$$

2.3 関数

整数 x に 1 を足す関数 f は次のように定義できる．

$$fx = x + 1 \tag{5}$$

ここに x は関数 f の引数である．引数は括弧でくるまない．

- Haskell では $fx = x + 1$ を

```
f x = x + 1
```

と書く．

本書では関数名を原則 1 文字として，イタリック体で表し，f, g, h のようにアルファベットの f 以降の文字を使う．ただし有名な関数についてはローマン体で表し，文字数も 2 文字以上とする．たとえば sin などの三角関数や指数関数がそれにあたる．

変数 x に関数 f を適用する場合は次のように書く．ここでも引数を括弧でくるまない．

$$z = fx \tag{6}$$

- Haskell では $z = fx$ を

```
z = f x
```

と書く.

関数 f が引数をふたつ取る場合は，次のように書く.

$$z = fxy \tag{7}$$

- Haskell では $z = fxy$ を

```
z = f x y
```

と書く.

なお fxy は $(fx)y$ と解釈される．前半の (fx) は 1 引数の関数とみなせる．2 引数関数を連続した 1 引数関数の適用とみなす考え方を，関数のカリー化と呼ぶ.

2.4 有名な関数

TK

2.5 関数の型

TK

2.6 関数合成

変数 x に関数 f と関数 g を連続して適用したい場合 $z = g(fx)$ とするところであるが，事前に関数 f と関数 g を合成しておきたいことがある.

関数の合成は次のように書く.

$$k = g \bullet f \tag{8}$$

関数の連続適用 $g(fx)$ と合成関数の適用 $(g \bullet f)x$ は同じ結果を返す.

- Haskell では $k = g \bullet f$ を

```
k = g . f
```

と書く.

関数合成演算子 . は以下のように左結合する.

$$\begin{aligned} k &= h \bullet g \bullet f \\ &= (h \bullet g) \bullet f \end{aligned} \tag{9}$$

関数適用のための特別な演算子 § があると便利である．演算子 § は関数合成演算子よりも優先順位が低い．例を挙げる.

$$\begin{aligned} z &= h \bullet (g \bullet f)x \\ &= h \,\S\, g \bullet fx \end{aligned} \tag{10}$$

- Haskell では $z = h \,\S\, g \bullet fx$ を

```haskell
z = h $ (g . f) x
```

と書く．

2.7 IO サバイバルキット 1

プログラムとは合成された関数である．多くのプログラミング言語では，プログラムそのものに main という名前をつける．本書では「IO モナド」の章で述べる理由によって，main 関数をサンセリフ体で main と書く．

実用的なプログラムはユーザからの入力を受け取り，関数を適用し，ユーザへ出力する．Haskell ではユーザからの 1 行の入力を getLine で受け取り，変数の値を print で書き出せる．ここに getLine と print は関数（ファンクション）ではあるが，特別に「アクション」とも呼ぶ．関数 main もアクションである．

引数 x の 2 乗を求める関数 f は次のように定義できる．

$$\begin{aligned} f &:: \mathbf{Double} \to \mathbf{Double} \\ fx &= x \times x \end{aligned} \tag{11}$$

- Haskell では 式 11 を

```haskell
f :: Double -> Double
f x = x * x
```

と書く．

ユーザからの入力に関数 f を適用してユーザへ出力するプログラムを Haskell で書くと次のようになる．

$$\mathsf{main} = \mathsf{print} \bullet f \bullet \mathrm{read} \heartsuit \mathsf{getLine} \tag{12}$$

3 Test Part

Function application.

$$z = fx \tag{13}$$

```haskell
z = f x
```

List map.

$$x_{\mathrm{s}} = f \star x_{\mathrm{s}} \tag{14}$$

```haskell
zs = f `map` xs
```

Functor map.

$$x_* = f * x_* \tag{15}$$

```haskell
zm = f <$> xm
```

Applicative map.

$$x_* = x_* \circledast x_* \tag{16}$$

```haskell
zm = fm <*> xm
```

Apllicative map 2.

$$x_* = x_* \circledast x_* \circledast x_* \tag{17}$$

```haskell
zm = gm <*> xm <*> ym
```

Applicative map 2 (another version).

$$x_* = g * x_* \circledast x_* \tag{18}$$

```haskell
zm = g <$> xm <*> ym
```

Monadic function application.

$$x_* = x^\dagger x \tag{19}$$

```haskell
zm = f x
```

Bind.

$$x_* = x^\dagger \heartsuit x_* \tag{20}$$

```haskell
zm = f =<< xm
```

Double bind.

$$x_* = x^\dagger \heartsuit x^\dagger \heartsuit x_* \tag{21}$$

```haskell
zm = g =<< f =<< xm
```