Haskell Notes

目次

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．

3 Test Part

Function application.

$$z = fx \tag{2}$$

```
z = f x
```

List map.

$$z_{\mathrm{s}} = f \star x_{\mathrm{s}} \tag{3}$$

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

Functor map.

$$z_* = f * x_* \tag{4}$$

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

Applicative map.

$$z_* = f_* \circledast x_* \tag{5}$$

```
zm = f <*> xm
```

Apllicative map 2.

$$z_* = f_* * x_* \circledast y_* \tag{6}$$

```
zm = f <$> xm <*> ym
```

Monadic function application.

$$z_* = f^\dagger x \tag{7}$$

```
zm = f xm
```

Bind.

$$z_* = f^\dagger \heartsuit x_* \tag{8}$$

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

Double bind.

$$z_* = g^\dagger \heartsuit f^\dagger \heartsuit x_* \tag{9}$$

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