Haskell Notes v.0

Ichi Kanaya

2025

0.1 変数

変数 x に値を代入するには次のようにする.[*1]

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

変数という呼び名に反して，変数の値は変えられない．そこで変数に値を代入するとは呼ばずに，変数名に値を**束縛**するという.

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．プログラマーが変数の値を変化させたい理由はユーザー入力，ループ，例外，内部状態，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，大域ジャンプ，継続に変数の破壊的代入は必要ない．ユーザー入力に関しても章を改めて取り上げる.

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 のようなアルファベット後半の文字を使う.

0.2 関数

関数 f は次のように定義できる.[*2]

$$fx = x + 1 \tag{2}$$

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.

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

[*1] Haskell では `x = 1` と書く.
[*2] Haskell では `f x = x+1` と書く.

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．[*3]

$$z = fx \tag{3}$$

関数 f が引数をふたつ取る場合は，次のように書く．[*4]

$$z = fxy \tag{4}$$

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．

0.3　ラムダ

関数とは，変数名に束縛された**ラムダ式**である．ラムダ式は次のように書く．[*5]

$$f = \backslash x \mapsto x + 1 \tag{5}$$

本書では無名変数 $\Diamond$ を用いた以下の書き方も用いる．[*6]

$$f = (\Diamond + 1) \tag{6}$$

$$= \backslash x \mapsto x + 1 \tag{7}$$

[*3] Haskell では `z = f x` と書く．

[*4] Haskell では `z = f x y` と書く．

[*5] Haskell では `f = \x -> x+1` と書く．

[*6] 無名変数は Haskell には無いが，代わりに「セクション」という書き方ができる．式 $(\Diamond + 1)$ は Haskell では `(+1)` と書く．

0.4 ローカル変数

関数内で**ローカル変数**を使いたい場合は以下のように行う．[*7]

$$z = \mathsf{let}\,\{y = 1\}\,\mathsf{in}\,x + y \tag{8}$$

ローカル変数の定義は次のように後置できる．[*8]

$$z = x + y\,\mathsf{where}\,\{y = 1\} \tag{9}$$

0.5 クロージャ

ラムダ式を返す関数は，ラムダ式内部に値を閉じ込めることができる．

$$fn = \backslash x \mapsto n + x \tag{10}$$

関数 f に引数 n を与えると，新たな 1 引数関数が得られる．例を挙げる．

$$n = 3 \tag{11}$$
$$g = fn \tag{12}$$

この例では，関数 g の中に値 $n = 3$ が閉じ込められているため $g1$ は 4 と評価される．値を閉じ込めたラムダ式を**クロージャ**と呼ぶ．

[*7] Haskell では `z = let {y = 1} in x+y` と書く．`let` 節内の式がひとつの場合，中括弧は省略可能である．式が複数になる場合は `;` で区切る．

[*8] Haskell では `z = x+y where {y = 1}` と書く．`where` 節内の式が一つの場合，中括弧は省略可能である．式が複数になる場合は `;` で区切る．

0.6　型

すべての変数，関数には**型**がある．代表的な型にはブール型，整数型，浮動小数点型，文字型がある．以降，ブール型を $\mathbb{B}$ で，整数型を $\mathbb{Z}$ で表す．[*9]

浮動小数点型は実数全体を表現できないが，本書では実数全体を意味する $\mathbb{R}$ で表すことにする．[*10]

本書では対応する，あるいは近い数学概念がある場合，型名をブラックボード体 1 文字で書く．文字型のように対応する数学概念がない場合はボールドローマン体を用いる．文字型は **Char** とする．[*11]

変数 x の型が $\mathbb{Z}$ のとき，以下のように**型注釈**を書く．[*12]

$$x :: \mathbb{Z} \tag{13}$$

同じことを数学者は $x \in \mathbb{Z}$ と書くことを好むが，記号 $\in$ は別の用途で使うため :: を用いる．

1 引数関数の型は次のように注釈できる．[*13]

$$f :: \mathbb{Z} \to \mathbb{Z} \tag{14}$$

ここで関数 f は整数型の引数をひとつとり，整数型の値を返す．

[*9] Haskell ではブール型を `Bool`，整数型を `Int`，多倍長整数型を `Integer` と書く．

[*10] Haskell では単精度浮動小数点型を `Float`，倍精度浮動小数点型を `Double` と書く．

[*11] Haskell では Unicode 文字型を `Char` と書く．

[*12] Haskell では `x :: Int` と書く．

[*13] Haskell では `f :: Int -> Int` と書く．

2 引数関数の方は次のように注釈できる. [14]

$$f :: \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z} \tag{15}$$

ここで関数 f は整数型の引数をふたつとり，整数型の値を返す. 型 $\mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$ は $\mathbb{Z} \to (\mathbb{Z} \to \mathbb{Z})$ と解釈される.

$(\mathbb{Z} \to \mathbb{Z})$ 型の関数を受取，$(\mathbb{Z} \to \mathbb{Z})$ 型の関数を返す関数は次の型を持つ. [15]

$$f :: (\mathbb{Z} \to \mathbb{Z}) \to (\mathbb{Z} \to \mathbb{Z}) \tag{16}$$

なお後半の括弧は省略可能なので

$$f :: (\mathbb{Z} \to \mathbb{Z}) \to \mathbb{Z} \to \mathbb{Z} \tag{17}$$

と書いても良い.

0.7 条件

条件分岐は次のように書く. [16]

$$z = \mathsf{if}\, x > 0\, \mathsf{then}\, x\, \mathsf{else} -x \tag{18}$$

条件分岐の代わりに以下のような**パターンマッチ**も使える. [17]

$$f = \mathsf{case}\, x\, \mathsf{of} \begin{cases} 1 \dashrightarrow 1 \\ \_ \dashrightarrow 0 \end{cases} \tag{19}$$

---

[14] Haskell では `f :: Int -> Int -> Int` と書く.

[15] Haskell では以下のように書く.

```
f :: (Int -> Int) -> (Int -> Int)
```

[16] Haskell では `z = if x>0 then x else -x` と書く.

[17] Haskell では以下のように書くのが一般的である.

```
f = case x of 1 -> 1
              _ -> 0
```

この場合 $x \equiv 1$ ならば f は 1 を，そうでなければ f は 0 を返す．ここに _ はすべてのパターンに一致する記号である．パターンマッチは上から順に行われる．

関数定義にもパターンマッチを使える．[*18]

$$\begin{cases} f 1 = 1 \\ f \_ = 0 \end{cases} \tag{20}$$

関数定義には次のように**ガード**と呼ばれる条件を付与することができる．[*19]

$$\begin{cases} f x \mid_{x>0} = x \\ \quad\;\; \mid_{\mathsf{otherwise}} = -x \end{cases} \tag{21}$$

[*18] Haskell では次のように書く．

```
f 1 = 1
f _ = 0
```

[*19] Haskell では次のように書く．

```
f x | x > 0     = x
    | otherwise = -x
```

0.8 関数の再帰呼び出し

関数は再帰的に呼び出せる．$x \geq 0$ を前提とすると，x 番目のフィボナッチ数を計算する関数 fib を次のように定義できる．[*20]

$$\begin{cases} \mathrm{fib}\,0 = 0 \\ \mathrm{fib}\,1 = 1 \\ \mathrm{fib}\,x = \mathrm{fib}(x-1) + \mathrm{fib}(x-2) \end{cases} \tag{22}$$

0.9 関数合成

関数の**合成**は次のように書く．[*21]

$$f = f_2 \bullet f_1 \tag{23}$$

関数合成演算子 $\bullet$ は以下のように右結合する．

$$\begin{aligned} f &= f_3 \bullet f_2 \bullet f_1 && (24)\\ &= (f_3 \bullet f_2) \bullet f_1 && (25)\\ & && (26) \end{aligned}$$

[*20] Haskell では次のように書く．ただし Haskell には符号なし整数型がないために `x` が正であることを別に担保する必要がある．またこのコードは無駄な再帰呼び出しを行っており実用的ではない．

```
fib 0 = 0
fib 1 = 1
fib x = fib (x-1) + fib (x-2)
```

[*21] Haskell では `f = f2.f1` と書く．

優先順位の低い関数合成演算子もあると便利である．そのために演算子 $ を導入する．例を挙げる．[*22]

$$
\begin{aligned}
f &= f_3 \,\$\, f_2 \bullet f_1 && (27)\\
&= f_3 \bullet (f_2 \bullet f_1) && (28)
\end{aligned}
$$

0.10　タプル

複数の変数をまとめてひとつの**タプル**にすることができる．例を挙げる．[*23]

$$z = (x, y) \tag{29}$$

タプルの型は，要素の型をタプルにしたものである．例えば $\mathbb{Z}$ が 2 個からなるタプルの型は次のようになる．[*24]

$$z :: (\mathbb{Z}, \mathbb{Z}) \tag{30}$$

要素を含まないタプルを**ユニット**と呼ぶ．ユニットは次のように書く．[*25]

$$z = () \tag{31}$$

ユニットの型は**ユニット型**で，型注釈を次のように書く．[*26]

$$z :: () \tag{32}$$

[*22] Haskell では `f = f3 $ f2.f1` と書く．
[*23] Haskell では `z = (x, y)` と書く．
[*24] In Haskell, `z :: (Int, Int)`.
[*25] Haskell では `z = ()` と書く．
[*26] Haskell では `z :: ()` と書く．

0.11 リストと内包表記

ある変数が**リスト**であるとき，その変数がリストであることを忘れないように x_{s} と小さく s を付けることにする．

空リストは次のように定義する．[*27]

$$x_{\mathrm{s}} = [\,] \tag{33}$$

任意のリストは次のように構成する．

$$x_{\mathrm{s}} = x_0 : x_1 : x_2 : \cdots : [\,] \tag{34}$$

リストの型はその構成要素の型をブラケットで包んで表現する．[*28]

$$x_{\mathrm{s}} :: [\![\mathbb{Z}]\!] \tag{35}$$

リストは次のように構成することもできる．[*29]

$$x_{\mathrm{s}} = [1, 2, \ldots, 100] \tag{36}$$

リストの構成には**内包表記**が使える．例を挙げる．[*30]

$$x_{\mathrm{s}} = \left[x^2 \mid x \in [1, 2 \ldots 100], x > 50\right] \tag{37}$$

リストとリストをつなぐ場合は**リスト結合演算子** $+\!\!\!+$ を用いる．[*31]

$$z_{\mathrm{s}} = x_{\mathrm{s}} +\!\!\!+ y_{\mathrm{s}} \tag{38}$$

[*27] Haskell では `xs = []` と書く．
[*28] Haskell では `xs :: [Int]` と書く．
[*29] Haskell では `xs = [1, 2..100]` と書く．
[*30] Haskell では次のように書く．

```
xs = [x^2 | x <- [1, 2..100], x>50]
```

[*31] Haskell では `zs = xs ++ ys` と書く．

0.12　マップと畳み込み

リスト x_s の各要素に関数 f を適用して，その結果をリスト z_s に格納するためには次のように**マップ演算子** $\otimes$ を用いる．[*32]

$$z_\mathrm{s} = f \otimes x_\mathrm{s} \tag{39}$$

リスト x_s の各要素を先頭から順番に 2 項演算子を適用して，その結果を得るには畳み込み演算子を用いる．例えば整数リストの和は次のように書ける．[*33]

$$z = \bigcup_{0}^{(\lozenge+\lozenge)} x_\mathrm{s} \tag{40}$$

リスト x_s が $[x_0, x_1, \ldots, x_n]$ のとき，一般に

$$\bigcup_{a}^{\maltese} x_\mathrm{s} = a \maltese x_0 \maltese x_1 \ldots x_{n-1} \maltese x_n \tag{41}$$

である．

畳み込み演算子には次の右結合バージョンが存在する．[*34]

$$\bigsqcup_{a}^{\maltese} x_\mathrm{s} = a \maltese (x_0 \ldots (x_{n-2} \maltese (x_{n-1} \maltese x_n))) \tag{42}$$

0.13　Maybe

計算は失敗する可能性がある．例えば

$$z = y/x \tag{43}$$

[*32] Haskell では `zs = f ‘map‘ xs` と書く．

[*33] Haskell では `z = foldl 0 (+) xs` と書く．

[*34] Haskell では `foldr` を用いる．

のときに $x \equiv 0$ であったとしたら，この計算は失敗する．プログラムが計算を失敗した場合，たいていのプログラマは大域ジャンプを試みる．しかし大域ジャンプは変数の書き換えを行うことであるから，別の方法が望まれる．Haskell では失敗する可能性がある場合には Maybe という機構が使える．

いま関数 f が引数 x と y を取り，$x \neq 0$ であるならば y/x を返すとする．もし $x \equiv 0$ であれば失敗を意味する $\varnothing$（ナッシング）を返すとする．すると関数 f の定義は次のようになる．

$$f\,y\,x = \mathsf{if}\,x \neq 0\,\mathsf{then}\,y/x\,\mathsf{else}\,\varnothing \tag{44}$$

残念ながら上式は不完全である．なぜならば $x \neq 0$ のときの戻り値は数であるのに対して，$x \equiv 0$ のときの戻り値は数ではないからである．そこで

$$f\,y\,x = \mathsf{if}\,x \neq 0\,\mathsf{then}\,{}^{\mathrm{Just}}\langle y/x\rangle\,\mathsf{else}\,\varnothing \tag{45}$$

とする．ここに ${}^{\mathrm{Just}}\langle y/x\rangle$ は数 y/x から作られる，Maybe で包まれた数である．

整数型 $\mathbb{Z}$ を Maybe で包む場合は ${}^{?}\langle\!\langle\mathbb{Z}\rangle\!\rangle$ と書く．Maybe で包まれた型を持つ変数は $x_?$ のように小さく ? をつける．例を挙げる．[*35]

$$x_? :: {}^{?}\langle\!\langle\mathbb{Z}\rangle\!\rangle \tag{46}$$

Maybe で包まれた型を持つ変数は，値を持つか $\varnothing$（ナッシング）であるかのいずれかである．値をもつ場合は

$$x_? = {}^{\mathrm{Just}}\langle 1\rangle \tag{47}$$

[*35] Haskell では `xm :: Maybe Int` と書く．

のように書く．[*36]

Maybe 変数が値を持たない場合は

$$x_? = \varnothing \tag{48}$$

と書く．[*37]

0.14 Maybe に対する計算

Maybe 変数に，非 Maybe 変数を受け取る関数を適用することは出来ない．そこで特別な演算子 Ⓢ を用いる．[*38]

$$z_? = (\Diamond + 1) \circledS x_? \tag{49}$$

ここに演算子 Ⓢ は

$$^{\text{Just}}\langle fx \rangle = f \circledS {}^{\text{Just}}\langle x \rangle \tag{50}$$

$$\varnothing = f \circledS \varnothing \tag{51}$$

と定義される．

0.15 型パラメタ

型をパラメタとして扱うことができる．任意の型を $\mathbf{a}$ と，ボールド体小文字で書く．ある型 $\mathbf{a}$ の引数を取り，同じ型を返す関数の型は次のように書ける．[*39]

$$f :: \mathbf{a} \to \mathbf{a} \tag{52}$$

[*36] Haskell では `xm = Just 1` と書く．
[*37] Haskell では `xm = Nothing` と書く．
[*38] Haskell では `zm = (+1) <$> xm` と書く．
[*39] Haskell では `f :: a -> a` と書く．

型パラメタには制約をつけることができる．型の集合を**型クラス**と呼び，フラクチュール体で書く．たとえば数を表す型クラスは $\mathfrak{Num}$ である．型パラメタ $\mathbf{a}$ が型クラス $\mathfrak{Num}$ に属するとき，上述の関数 f の型注釈は次のようになる．[*40]

$$f :: \mathfrak{Num} \supset \mathbf{a} \Rightarrow \mathbf{a} \to \mathbf{a} \tag{53}$$

型クラスは型に制約を与える．

TK. `Num a => x :: a` ならば `x` が持つべき演算子.

TK. 型クラスの例.

0.16 関手

型 $\mathbf{a}$ のリストの変数は

$$x_{\mathrm{s}} :: [\![\mathbf{a}]\!] \tag{54}$$

という型注釈を持つ．型 $\mathbf{a}$ 型の Maybe の変数は

$$x_{?} :: {}^{?}\langle\!\langle\mathbf{a}\rangle\!\rangle \tag{55}$$

という型注釈を持つ．

普段遣いの関数

$$f :: \mathbf{a} \to \mathbf{a} \tag{56}$$

[*40] Haskell では `f :: Num a => a -> a` と書く．

をリスト変数 x_s に適用する場合は

$$z_\mathrm{s} = f \otimes x_\mathrm{s} \tag{57}$$

とする．同じく関数 f を Maybe 変数 $x_?$ に適用する場合は

$$z_? = f \circledS x_? \tag{58}$$

とする．

リストも Maybe も元の型 $\mathbf{a}$ から派生しており，関数適用のための特別な演算子を持つことになる．そこで，リストや Maybe は**関手**という型クラスに属する，型パラメタを伴う型であるとする．関手の型クラスを $\mathfrak{Functor}$ で表す．関手型クラスの $\mathbf{a}$ 型の変数を次のように型注釈する．[*41]

$$x_\star :: \mathfrak{Functor} \supset \mathbf{f} \Rightarrow {}^{\mathbf{f}}\langle\!\langle \mathbf{a} \rangle\!\rangle \tag{59}$$

型クラス $\mathfrak{Functor}$ に属する型は $\circledS$ 演算子を持たねばならない．演算子 $\circledS$ は次の形を持つ．[*42]

$$z_\star = f \circledS x_\star \tag{60}$$

演算子 $\circledS$ の型は次のとおりである．

$$\Diamond \circledS \Diamond :: (\mathbf{a} \to \mathbf{b}) \to {}^{\mathbf{f}}\langle\!\langle \mathbf{a} \rangle\!\rangle \to {}^{\mathbf{f}}\langle\!\langle \mathbf{b} \rangle\!\rangle \tag{61}$$

もし変数 $x_\star$ の型がリストであれば

$$\circledS = \otimes \tag{62}$$

[*41] Haskell では `xm :: Functor f => f a` と書く．
[*42] In Haskell, `zm = f <$> xm`.

であると解釈する.

fmap.

関手は関手則に従う.

Function of parametric type with functor class:[*43]

$$f :: \mathfrak{Functor} \supset \mathbf{f} \Rightarrow \mathbf{a} \rightarrow {}^{\mathbf{f}}\langle\!\langle \mathbf{a} \rangle\!\rangle \tag{63}$$

Example function application:[*44]

$$z_\star = (\Diamond + 1) \circledS {}^{\mathrm{Just}} \langle x \rangle \tag{64}$$

0.17 関手としての関数

Function as a functor:[*45]

$$f :: (\blacklozenge \rightarrow \mathbf{r})\, \mathbf{q} = {}^{(\blacklozenge \rightarrow \mathbf{r})} \langle\!\langle \mathbf{q} \rangle\!\rangle \tag{65}$$

Thus,

$$f_2 \bullet f_1 \equiv f_2 \circledS f_1 \tag{66}$$

0.18 アプリカティブ関手

Pure:[*46]

$$z_\star = {}^\star \langle x \rangle \tag{67}$$

---

*43 In Haskell, `f :: Functor f => a -> f a`.
*44 In Haskell, `zm = (+1) <$> Just x`.
*45 In Haskell, `f :: ((->) r) q`.
*46 In Haskell, `zm = pure x`.

Applicative map:[*47]

$$z_\star = f_\star \boxtimes x_\star \tag{68}$$

where

$$f_\star :: {}^{\mathbf{f}}\langle\!\langle \mathbf{a} \to \mathbf{b} \rangle\!\rangle \tag{69}$$

Applicative style:[*48]

$$z_\star = {}^{\star}\langle \lozenge + \lozenge \rangle \boxtimes x_\star \boxtimes y_\star \tag{70}$$

or[*49]

$$z_\star = (\lozenge + \lozenge) \circledS x_\star \boxtimes y_\star \tag{71}$$

0.19　モナド

Returning *Maybe*:[*50]

$$f :: \mathbb{Z} \to {}^{?}\langle\!\langle \mathbb{Z} \rangle\!\rangle \tag{72}$$

$$fx = {}^{\text{Just}}\langle x \rangle \tag{73}$$

Returning *monad*:

$$f :: \mathbb{Z} \to {}^{\mathbf{m}}\langle\!\langle \mathbf{a} \rangle\!\rangle \tag{74}$$

$$fx = {}^{\text{return}}\langle x \rangle \tag{75}$$

[*47] In Haskell, `zm = f <*> xm`.
[*48] In Haskell, `zm = pure (+) <*> xm <*> ym`.
[*49] In Haskell, `zm = (+) <$> xm <*> ym` or `zm = liftA2 (+) xm ym`.
[*50] In Haskell, `f :: Int -> Maybe Int` and `f x = Just x`.

Returning monadic value:[*51]

$$f :: \mathfrak{Monad} \supset \mathbf{m} \Rightarrow \mathbf{a} \rightarrow {}^{\mathbf{m}} \langle\!\langle \mathbf{a} \rangle\!\rangle \tag{76}$$

Monadic function binding:[*52]

$$z_\star = x_\star \mathrel{-\!\heartsuit\!\!\rightarrow} f_1 \mathrel{-\!\heartsuit\!\!\rightarrow} f_2 \tag{77}$$

where

$$f_1 :: \mathbb{Z} \rightarrow {}^{?} \langle\!\langle \mathbb{Z} \rangle\!\rangle \tag{78}$$

$$f_2 :: \mathbb{Z} \rightarrow {}^{?} \langle\!\langle \mathbb{Z} \rangle\!\rangle \,. \tag{79}$$

Function binding of monadic function and non-monadic function:[*53]

$$z_\star = x_\star \mathrel{-\!\heartsuit\!\!\rightarrow} f \mathrel{-\!\heartsuit\!\!\rightarrow} g' \qquad \text{where } \{g'w = {}^{\star} \langle gw \rangle\} \tag{80}$$

where

$$f :: \mathbb{Z} \rightarrow {}^{?} \langle\!\langle \mathbb{Z} \rangle\!\rangle \tag{81}$$

$$g :: \mathbb{Z} \rightarrow \mathbb{Z}. \tag{82}$$

Another solution is:

$$z_\star = (g^* \bullet f) x_\star \tag{83}$$

where g^* means `liftM g` in Haskell.[*54]

[*51] In Haskell, `f :: Monad m => a -> m a`.

[*52] In Haskell, `zm = xm >>= f1 >>= f2`.

[*53] In Haskell,

```
zm = xm >>= f >>= g'
  where g' w = pure (g w)
```

[*54] In Haskell, `zm = (liftM g . f) xm`.

0.20　種

$$\bigstar \to \bigstar \tag{84}$$

0.21　Data

Data:[*55]

$$\textsf{data}\,\mathbf{Suit} = \mathrm{Spade} \vee \mathrm{Heart} \vee \mathrm{Club} \vee \mathrm{Diamond} \tag{85}$$

Data with parameters:[*56]

$$\textsf{data}\,\mathbf{V^2} = \mathrm{V}^2\,\{x :: \mathbb{Z}, y :: \mathbb{Z}\} \tag{86}$$

[*55] In Haskell,

```
data Suit = Spade | Heart | Club | Diamond
```

[*56] In Haskell,

```
data V2 = V2 { x :: Int, y :: Int}
```

or `data V2 = V2 Int Int`.

0.22 型クラスとインスタンス

0.23 IO

IO example:[*57]

$$main = getLine \mathrel{-\!\!\heartsuit\!\!\rightarrow} print \gg^{\text{return}} \langle 0 \rangle \tag{87}$$

0.24 Do 構文

Do notation:[*58]

$$z_{\star} = \mathsf{do}\,\{x' \leftarrow x_{\star}; y' \leftarrow y_{\star}; fx'; gy'\} \tag{88}$$

[*57] In Haskell, `main = getLine >>= print >> return 0`.
[*58] In Haskell, `z = do {x’ <- x; y’ <- y; f x’; g y’}`.