Haskell Notes v.0

Ichi Kanaya

2025

0.1 Haskell

TK. Haskell について.

0.2 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．以下，誤解のおそれがない限り整数型を $\mathbb{Z}$ で，浮動小数点型を $\mathbb{R}$ で表す.[*2]

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 $x :: \mathbb{Z}$ と書く．これは記号 $\in$ を別の用途に用いるためである.[*3]

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

[*1] Haskell では `x = 1` と書く.
[*2] Haskell ではそれぞれ `Int` および `Double` を用いる.
[*3] Haskell では `x :: Int` と書く.

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．

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．

0.3 関数

整数 x に 1 を足す**関数** f は次のように定義できる．[*4]

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

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．[*5]

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

関数 f が引数をふたつ取る場合は，次のように書く．[*6]

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

[*4] Haskell では `f x = x+1` と書く．
[*5] Haskell では `z = f x` と書く．
[*6] Haskell では `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．

TK. 有名な関数，実数編．

0.4 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

$$z = g(fx) \tag{5}$$

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．[*7]

$$k = g \bullet f \tag{6}$$

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．

関数合成演算子 $\bullet$ は以下のように**左結合**する．

$$k = h \bullet g \bullet f \tag{7}$$
$$= (h \bullet g) \bullet f \tag{8}$$
$$\tag{9}$$

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

[*7] Haskell では `k = g . f` と書く．

関数合成演算子よりも優先順位が低い．例を挙げる．[*8]

$$z = h \S (g \bullet f)x \tag{10}$$
$$= h\,((g \bullet f)x) \tag{11}$$

0.5 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 関数をスラント体で *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 1.5 乗を求める関数 f は次のように定義できる．[*9]

$$fx = x^{1.5} \tag{12}$$

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．[*10]

$$main = print \bullet f \bullet \mathrm{read} \heartsuit getLine \tag{13}$$

[*8] Haskell では `z = h $ (g . f) x` と書く．
[*9] Haskell では `f x = x ** 1.5` と書く．
[*10] Haskell では `main = print . f . read =<< getLine` と書く．

ここに関数 read は**文字列**であるユーザ入力を数に変換する関数である．また演算子 ▽ は新たな関数合成演算子で，アクションとアクションを合成するための特別な演算子である．詳細は「**モナド**」の章で述べる．

0.6 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．[*11]

$$f = \backslash x \mapsto x + 1 \tag{14}$$

ラムダ記号は一般的には λ が用いられるが，本書ではすべてのギリシア文字を予約しておきたいので，Haskell の記法に倣って \ を用いる．

本書では無名変数 ◊ を用いた以下の書き方も用いる．[*12]

$$f = (\Diamond + 1) \tag{15}$$
$$= \backslash x \mapsto x + 1 \tag{16}$$

無名変数が 2 回以上登場した場合は，その都度新しいパラメタを生成する．たとえば次のとおりである．[*13]

$$f = \Diamond + \Diamond \tag{17}$$
$$= \backslash x \mapsto \backslash y \mapsto x + y \tag{18}$$

なお $\backslash x \mapsto \backslash y \mapsto x + y$ は $\backslash x\, y \mapsto x + y$ と書いても良い．[*14]

[*11] Haskell では `f = \x -> x+1` と書く．

[*12] 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．

[*13] Haskell では `f = (+)` と書く．

[*14] Haskell では `\x y -> x+y` と書く．

0.7　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．[*15]

$$z = \mathsf{let}\ \{y = 1\}\ \mathsf{in}\ x + y \tag{19}$$

ローカル変数はラムダ式のシンタックスシュガーである．式 (19) は次の式と等価である．

$$z = (\backslash y \mapsto x + y)1 \tag{20}$$

ローカル変数の定義は次のように後置できる．[*16]

$$z = x + y\ \mathsf{where}\ \{y = 1\} \tag{21}$$

0.8　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．たとえば

$$fn = \backslash x \mapsto n + x \tag{22}$$

のように関数を定義して良い．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．

$$n = 3 \tag{23}$$

$$g = fn \tag{24}$$

[*15] 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．

[*16] 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．

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．

0.9 型

すべての変数，関数には**型**がある．代表的な型には整数型，浮動小数点型，ブール型，文字型がある．整数型を $\mathbb{Z}$ で，浮動小数点型を $\mathbb{R}$ で表すことはすでに述べたとおりである．

Haskell には 2 種類の整数型がある．ひとつは**固定長整数型**で，もうひとつは**多倍長整数型**である．Haskell では前者を `Int` で，後者を `Integer` で表す．多倍長整数型はメモリの許す限り巨大な整数を扱えるので，整数全体の集合に近いのであるが，本書では $\mathbb{Z}$ と書いて固定長整数型を意味することにする．

浮動小数点型には**単精度浮動小数点型**と**倍精度浮動小数点型**があり，Haskell では前者を `Float` で，後者を `Double` で表現するが，単精度浮動小数点型はめったに用いられないため，今後 $\mathbb{R}$ と書けば倍精度浮動小数点型の意味とする．

ブール型は論理値 True または False のいずれかしか値をとれない型で，今後 $\mathbb{B}$ と書く．[*17]

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 $\mathbf{Char}$ とする．[*18]

[*17] Haskell ではブール型を `Bool` と書く．
[*18] Haskell では Unicode 文字型を `Char` と書く．

変数 x の型が $\mathbb{Z}$ のとき，以下のように**型注釈**を書く．[*19]

$$x :: \mathbb{Z} \tag{25}$$

1 引数関数の型は次のように注釈できる．[*20]

$$f :: \mathbb{Z} \to \mathbb{Z} \tag{26}$$

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．[*21]

2 引数関数の方は次のように注釈できる．[*22]

$$f :: \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z} \tag{27}$$

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．

$(\mathbb{Z} \to \mathbb{Z})$ 型の関数を受け取り，$(\mathbb{Z} \to \mathbb{Z})$ 型の関数を返す関数は次の型を持つ．[*23]

$$f :: (\mathbb{Z} \to \mathbb{Z}) \to (\mathbb{Z} \to \mathbb{Z}) \tag{28}$$

なお後半の括弧は省略可能なので

$$f :: (\mathbb{Z} \to \mathbb{Z}) \to \mathbb{Z} \to \mathbb{Z} \tag{29}$$

と書いても良い．

[*19] Haskell では `x :: Int` と書く．

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

[*21] 正確には → は型コンストラクタである．

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

[*23] Haskell では以下のように書く．

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

0.10 条件

条件分岐は次のように書く.[*24]

$$z = \text{if } x > 0 \text{ then } x \text{ else } -x \tag{30}$$

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.[*25]

$$f = \text{case } x \text{ of } \begin{cases} 1 \to 1 \\ \_ \to 0 \end{cases} \tag{31}$$

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.[*26]

$$\begin{cases} f1 = 1 \\ f\_ = 0 \end{cases} \tag{32}$$

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

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

[*25] 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
```

[*26] Haskell では次のように書く.

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

きる．[*27]

$$\begin{cases} f\,x \mid_{x>0} = x \\ \quad \mid_{\text{otherwise}} = -x \end{cases} \tag{33}$$

ここに otherwise は _ の別名である．

0.11 関数の再帰呼び出し

関数は再帰的に呼び出せる．$n \geq 0$ を前提とすると，n 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．[*28]

$$\begin{cases} \mathrm{fib}\,0 = 0 \\ \mathrm{fib}\,1 = 1 \\ \mathrm{fib}\,n = \mathrm{fib}(n-1) + \mathrm{fib}(n-2) \end{cases} \tag{34}$$

0.12 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．[*29]

$$z = (x, y) \tag{35}$$

[*27] Haskell では次のように書く．

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

[*28] 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 `n` 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 n = fib (n-1) + fib (n-2)
```

[*29] Haskell では `z = (x, y)` と書く．

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．[*30]

$$z :: (\mathbb{Z}, \mathbb{Z}) \tag{36}$$

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．[*31]

$$z = () \tag{37}$$

ユニットの型は**ユニット型**で，型注釈を次のように書く．[*32]

$$z :: () \tag{38}$$

0.13 リスト

任意の型について，その型の要素を並べた列を**リスト**と呼ぶ．

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．[*33]

$$x_\mathrm{s} = [\,] \tag{39}$$

任意のリストは次のように : 演算子を用いて構成する．

$$x_\mathrm{s} = x_0 : x_1 : x_2 : \cdots : [\,] \tag{40}$$

[*30] In Haskell, `z :: (Int, Int)`.

[*31] Haskell では `z = ()` と書く．

[*32] Haskell では `z :: ()` と書く．

[*33] Haskell では `xs = []` と書く．

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.[*34]

$$x_{\mathrm{s}} :: [\mathbb{Z}] \tag{41}$$

リストは次のように構成することもできる.[*35]

$$x_{\mathrm{s}} = [1, 2, \ldots, 100] \tag{42}$$

なお次のような**無限リスト**を構成しても良い.[*36]

$$x_{\mathrm{s}} = [1, 2, \ldots] \tag{43}$$

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.[*37]

$$z_{\mathrm{s}} = x_{\mathrm{s}} \# y_{\mathrm{s}} \tag{44}$$

関数はリストを受け取ることができる．次の書き方では，関数 f は整数リストの最初の要素 x と残りの要素 x_{s} を別々に受け取り，先頭要素だけを返す.[*38]

$$f :: [\mathbb{Z}] \to \mathbb{Z} \tag{45}$$
$$f(x : x_{\mathrm{s}}) = x \tag{46}$$

先頭から 2 個の要素を受け取り，それらの和を返す場合は次のように書く.

$$f :: [\mathbb{Z}] \to \mathbb{Z} \tag{47}$$
$$f(x : y : y_{\mathrm{s}}) = x + y \tag{48}$$

[*34] Haskell では `xs :: [Int]` と書く.
[*35] Haskell では `xs = [1, 2..100]` と書く.
[*36] Haskell では `xs = [1, 2..]` と書く.
[*37] Haskell では `zs = xs ++ ys` と書く.
[*38] Haskell では `f (x:xs) :: [Int] -> Int = x` と書く.

リストのリストは次のように構成できる．

$$z_{\mathrm{ss}} = [[1,2],[3,4]] \tag{49}$$

0.14 内包表記

リストの構成には**内包表記**が使える．例を挙げる．[*39]

$$x_{\mathrm{s}} = \left[x^2 \mid x \in [1,2\ldots100], \mathrm{even}\, x\right] \tag{50}$$

関数 even は引数が偶数の場合にだけ True を返す関数である．この例では数列 $[1,2\ldots100]$ のうち偶数だけを 2 乗したリストを作っている．

0.15 文字列

文字型のリストを文字列型と呼び **String** で表す．**String** 型は次のように予約語 type を用いて，**型シノニム**すなわち型の別名として次のように定義される．

$$\mathsf{type}\ \mathbf{String} = [\mathbf{Char}] \tag{51}$$

文字列型のリテラルは次のように書く．[*40]

$$x_{\mathrm{S}} = \texttt{"Hello, World!"} \tag{52}$$

TK. String に関する有名な関数.

[*39] Haskell では次のように書く．

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

[*40] Haskell では `xs = "Hello, World!"` と書く．

0.16　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．[*41]

$$z_\mathrm{s} = f \otimes x_\mathrm{s} \tag{53}$$

式 (53) は次の式と同じである．[*42]

$$z_\mathrm{s} = [fx \mid x \in x_\mathrm{s}] \tag{54}$$

リスト x_s の各要素を先頭から順番に二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．たとえば整数リストの和は次のように書ける．[*43]

$$z = \bigcup_{0}^{(\Diamond+\Diamond)} x_\mathrm{s} \tag{55}$$

Haskell では

$$\sum = \bigcup_{0}^{(\Diamond+\Diamond)} \tag{56}$$

として関数 $\sum$ が定義されている．[*44]

リスト x_s が $x_\mathrm{s} = [x_0, x_1, \ldots, x_n]$ のとき，一般に

$$\bigcup_{a}^{\Diamond✠\Diamond} x_\mathrm{s} = a ✠ x_0 ✠ x_1 \ldots x_{n-1} ✠ x_n \tag{57}$$

[*41] Haskell では `zs = f `map` xs` と書く．
[*42] Haskell では `zs = [f x | x <- xs]` と書く．
[*43] Haskell では `z = foldl 0 (+) xs` と書く．
[*44] Haskell では関数 $\sum$ を `sum` と書く．

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．[*45]

$$\bigsqcup_{a}^{\Diamond \maltese \Diamond} x_{\mathrm{s}} = a \maltese (x_0 \ldots (x_{n-2} \maltese (x_{n-1} \maltese x_n))) \tag{58}$$

0.17 IO サバイバルキット 2

1 行ごとに 3 次元ベクトルが並べられた，以下の入力ファイルがあるとする．

input.txt

```
1.0 2.0 3.0
4.5 5.5 6.5
```

このようなファイル形式は計算機科学者にとって見慣れたものである．

各行つまり各ベクトルごとに，そのノルムを計算して出力するプログラムを書きたいとしよう．まず数列を受け取ってそのノルムを返す関数 norm を次のように定義する．[*46]

$$\mathrm{norm} :: [\mathbb{R}] \to \mathbb{R} \tag{59}$$

$$\mathrm{norm}[\,] = 0.0 \tag{60}$$

$$\mathrm{norm}\, x_{\mathrm{s}} = \mathrm{sqrt}\left(\sum [x * x \mid x \in x_{\mathrm{s}}]\right) \tag{61}$$

---

*45 Haskell では `foldr` を用いる．

*46 Haskell では次のように書く．

```
norm :: [Double] -> Double
norm [] = 0.0
norm xs = sqrt (sum [x * x | x <- xs])
```

入力ファイル全体を受け取るにはアクション *getContents* を用いる．入力ファイルを 1 行毎のリストにするには関数 lines を用いる．

各行を空白で区切ってリストに格納するには関数 words を用いる．

各文字列を数に変換するには次の関数 $\mathrm{read}_{\mathbb{R}} :: \mathbf{String} \to \mathbb{R}$ を用いる．[*47]

$$\underset{\mathbb{R}}{\mathrm{read}} :: \mathbf{String} \to \mathbb{R} \tag{62}$$

$$\underset{\mathbb{R}}{\mathrm{read}} = \mathrm{read} \tag{63}$$

入力ファイルの各行に書かれたベクトルを対象に関数 norm を適用して，結果を書き出すには次のように書く．[*48]

$$\begin{aligned} \mathit{main} = \mathit{print} \bullet (\mathrm{norm} \otimes) \bullet ((\underset{\mathbb{R}}{\mathrm{read}} \otimes) \otimes) \\ \bullet (\mathrm{words} \otimes) \bullet \mathrm{lines} \heartsuit \mathit{getContents} \end{aligned} \tag{64}$$

アクション *print* に代えて次の *printEach* を用いると，入力と出力を同じ形式にできる．[*49]

$$\mathit{printEach}\, x_{\mathrm{s}} = \mathit{print} \clubsuit x_{\mathrm{s}} \tag{65}$$

[*47] Haskell では次のように書く．

```
readDouble :: String -> Double
readDouble = read
```

[*48] Haskell では次のように書く．

```
main = print
  . (norm <$>)
  . ((readDouble <$>) <$>)
  . (words <$>)
  . lines
  =<< getContents
```

[*49] Haskell では `printEach xs = print ‘mapM‘ xs` と書く．

演算子 ♣ はアクション版のマップ演算子である．

0.18 Maybe

計算は失敗する可能性がある．たとえば

$$z = y/x \tag{66}$$

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．

$$fyx = \mathsf{if}\ x \neq 0\ \mathsf{then}\ y/x\ \mathsf{else}\ \varnothing \ldots \text{（不完全）} \tag{67}$$

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

$$f^{\dagger}yx = \mathsf{if}\ x \neq 0\ \mathsf{then}\ {}^{\mathrm{Just}}\langle y/x \rangle\ \mathsf{else}\ \varnothing \tag{68}$$

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 ${}^{\mathrm{Maybe}}\langle\!\langle \mathbb{Z} \rangle\!\rangle$ 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．[*50]

$$x_? :: {}^{\mathrm{Maybe}} \langle\!\langle \mathbb{Z} \rangle\!\rangle \tag{69}$$

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

$$x_? = {}^{\mathrm{Just}} \langle 1 \rangle \tag{70}$$

のように書く．[*51]

Maybe 変数が値を持たない場合は

$$x_? = \varnothing \tag{71}$$

と書く．[*52]

0.19 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 Ⓢ を用いて，次のように計算する．[*53]

$$z_? = (\Diamond + 1) \,Ⓢ\, x_? \tag{72}$$

ここに演算子 Ⓢ は

$${}^{\mathrm{Just}} \langle f\, x \rangle = f \,Ⓢ\, {}^{\mathrm{Just}} \langle x \rangle \tag{73}$$

$$\varnothing = f \,Ⓢ\, \varnothing \tag{74}$$

と定義される．

[*50] Haskell では `xm :: Maybe Int` と書く．
[*51] Haskell では `xm = Just 1` と書く．
[*52] Haskell では `xm = Nothing` と書く．
[*53] Haskell では `zm = (+1) <$> xm` と書く．

0.20 Maybe の中のリスト

リストが Maybe の中に入っている場合は，リストの各要素に関数を適用することができる．例を挙げる．

$$x_? = ^{\mathrm{Just}} \langle [1, 2, \ldots, 100] \rangle \tag{75}$$

のとき，リストの各要素に関数 $f :: \mathbb{Z} \to \mathbb{Z}$ を適用するには次のように書く．[*54]

$$z_? = (f \otimes) \textcircled{\scriptsize S} x_? \tag{76}$$

0.21 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．[*55]

$$f :: \mathbf{a} \to \mathbf{a} \tag{77}$$

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

[*54] Haskell では `zm = (f <$>) <$> xm` と書く．最初の `<$>` はリストの各要素に関数 f を適用する演算子，2 番目の `<$>` は Maybe の中のリストの各要素に関数 f を適用する演算子である．

[*55] Haskell では `f :: a -> a` と書く．

型注釈は次のようになる.[*56]

$$f :: \mathfrak{Num} \supset \mathbf{a} \Rightarrow \mathbf{a} \to \mathbf{a} \tag{78}$$

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.

0.22 関手

型 **a** のリストの変数は

$$x_{\mathrm{s}} :: [\mathbf{a}] \tag{79}$$

という型注釈を持つ. これは

$$x_{\mathrm{s}} :: {}^{[]}\langle\!\langle \mathbf{a} \rangle\!\rangle \tag{80}$$

のシンタックスシュガーである.

型 **a** の Maybe の変数は

$$x_{?} :: {}^{\mathrm{Maybe}}\langle\!\langle \mathbf{a} \rangle\!\rangle \tag{81}$$

という型注釈を持つ.

普段遣いの関数

$$f :: \mathbf{a} \to \mathbf{a} \tag{82}$$

[*56] 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

$$z_{\mathrm{s}} = f \otimes x_{\mathrm{s}} \tag{83}$$

とする．同じく関数 f を Maybe 変数 $x_?$ に適用する場合は

$$z_? = f \textcircled{\scriptsize S} x_? \tag{84}$$

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．[*57]

$$x_\star :: \mathfrak{Functor} \supset \mathbf{f} \Rightarrow {}^{\mathbf{f}}\langle\!\langle \mathbf{a} \rangle\!\rangle \tag{85}$$

型クラス $\mathfrak{Functor}$ に属する型は Ⓢ 演算子を持たねばならない．演算子 Ⓢ は次の形を持つ．[*58]

$$z_\star = f \textcircled{\scriptsize S} x_\star \tag{86}$$

演算子 Ⓢ の型は次のとおりである．

$$\Diamond \textcircled{\scriptsize S} \Diamond :: (\mathbf{a} \to \mathbf{b}) \to {}^{\mathbf{f}}\langle\!\langle \mathbf{a} \rangle\!\rangle \to {}^{\mathbf{f}}\langle\!\langle \mathbf{b} \rangle\!\rangle \tag{87}$$

もし変数 $x_\star$ の型がリストであれば

$$\textcircled{\scriptsize S} = \otimes \tag{88}$$

[*57] Haskell では `xm :: Functor f => f a` と書く．

[*58] In Haskell, `zm = f <$> xm`.

であると解釈する.

Function of parametric type with functor class:[*59]

$$f :: \mathfrak{Functor} \supset \mathbf{f} \Rightarrow \mathbf{a} \to {}^{\mathbf{f}}\langle\!\langle \mathbf{a} \rangle\!\rangle \tag{89}$$

Example function application:[*60]

$$z_\star = (\lozenge + 1)\ \text{Ⓢ}^{\,\text{Just}}\ \langle x \rangle \tag{90}$$

0.23 関手としての関数

$$f :: \mathbf{q} \to \mathbf{r} \tag{91}$$

Function as a functor:[*61]

$$f :: (\blacklozenge \to \mathbf{r})\,\mathbf{q} = {}^{(\blacklozenge \to \mathbf{r})}\langle\!\langle \mathbf{q} \rangle\!\rangle \tag{92}$$

Thus,

$$f_2 \bullet f_1 \equiv f_2\ \text{Ⓢ}\ f_1 \tag{93}$$

$$\begin{aligned} \mathrm{id} \bullet f &= \mathrm{id}\, f = f && (94)\\ (h \bullet g) \bullet f &= ((h\bullet) \bullet (g\bullet))f && (95)\\ &= h \bullet (g \bullet f) && (96) \end{aligned}$$

[*59] In Haskell, `f :: Functor f => a -> f a`.
[*60] In Haskell, `zm = (+1) <$> Just x`.
[*61] In Haskell, `f :: ((->) r) q`.

0.24 アプリカティブ関手

演算子 Ⓢ は関手型クラスの値に 1 引数関数を適用することを可能にした．一方で 2 引数関数を適用するのは若干面倒である．いま関数 f が 2 引数をとるとし，関手型クラスの変数 $x_\star$ と $y_\star$ があるとする．関数 f に変数 $x_\star$ を部分適用して関数 f' すなわち

$$f' = f \,Ⓢ\, x_\star \tag{97}$$

を作ると，定義によって関数 f' は関手型クラスになる．そこで，関手型クラスの関数を関手型クラスの変数に適用する新しい演算子が必要になる．このような演算子を**アプリカティブマップ演算子**と呼び $\circledast$ で表す．アプリカティブマップ演算子を用いると

$$z_\star = f' \circledast y_\star \tag{98}$$
$$= f \,Ⓢ\, x_\star \circledast y_\star \tag{99}$$

と書ける．

任意の変数または関数を関手型クラスに入れる**ピュア演算子**があり，次のように書く．[*62]

$$z_\star = {}^\star\langle x \rangle \tag{100}$$

なおピュア演算子の名称は「純粋 (pure)」であるが，意味合いはむしろ「不純 (impure)」のほうが近い．

ピュア演算子を用いると，式 (99) は

$$z_\star = {}^\star\langle f \rangle \circledast x_\star \circledast y_\star \tag{101}$$

[*62] Haskell では `zm = pure x` と書く．

と書ける.

Applicative map:[*63]

$$z_\star = f_\star \circledast x_\star \tag{102}$$

where

$$f_\star :: {}^{\mathbf{f}} \langle\!\langle \mathbf{a} \to \mathbf{b} \rangle\!\rangle \tag{103}$$

Applicative style:[*64]

$$z_\star = {}^\star \langle f \rangle \circledast x_\star \circledast y_\star \tag{104}$$

or[*65]

$$z_\star = f \mathbin{\textcircled{\scriptsize S}} x_\star \circledast y_\star \tag{105}$$

or[*66]

$$z_\star = [\![f\, x_\star\, y_\star]\!] \tag{106}$$

0.25　モナド

Returning *List*.

$$. \tag{107}$$

[*63] In Haskell, `zm = f <*> xm`.
[*64] In Haskell, `zm = pure (+) <*> xm <*> ym`.
[*65] In Haskell, `zm = f <$> xm <*> ym`.
[*66] In Haskell, `zm = liftA2 f xm ym`.

Returning *Maybe*:[*67]

$$f :: \mathbb{Z} \to {}^{\text{Maybe}} \langle\!\langle \mathbb{Z} \rangle\!\rangle \tag{108}$$

$$fx = {}^{\text{Just}} \langle x \rangle \tag{109}$$

Returning *monad*:

$$f :: \mathbb{Z} \to {}^{\mathbf{m}} \langle\!\langle \mathbf{a} \rangle\!\rangle \tag{110}$$

$$fx = {}^{\star} \langle x \rangle \tag{111}$$

Returning monadic value:[*68]

$$f :: \mathfrak{Monad} \supset \mathbf{m} \Rightarrow \mathbf{a} \to {}^{\mathbf{m}} \langle\!\langle \mathbf{a} \rangle\!\rangle \tag{112}$$

Monadic function binding:[*69]

$$z_\star = x_\star \multimap f_1 \multimap f_2 \tag{113}$$

where

$$f_1 :: \mathbb{Z} \to {}^{\text{Maybe}} \langle\!\langle \mathbb{Z} \rangle\!\rangle \tag{114}$$

$$f_2 :: \mathbb{Z} \to {}^{\text{Maybe}} \langle\!\langle \mathbb{Z} \rangle\!\rangle . \tag{115}$$

Function binding of monadic function and non-monadic function:[*70]

$$z_\star = x_\star \multimap f \multimap g' \text{ where } \{g' w = {}^{\star} \langle g w \rangle\} \tag{116}$$

[*67] In Haskell, `f :: Int -> Maybe Int` and `f x = Just x`.

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

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

[*70] In Haskell,

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

or

$$z_\star = x_\star \multimap (f \rightarrowtail g') \text{ where } \{g'w = {}^\star \langle gw \rangle\} \tag{117}$$

where

$$f :: \mathbb{Z} \to {}^{\text{Maybe}} \langle\!\langle \mathbb{Z} \rangle\!\rangle \tag{118}$$

$$g :: \mathbb{Z} \to \mathbb{Z}. \tag{119}$$

Another solution is:

$$z_\star = (g^* \bullet f) \heartsuit x_\star \tag{120}$$

where g^* means `liftM g` in Haskell.[*71]

0.26 種

$$\star \to \star \tag{121}$$

0.27 Data

Data:[*72]

$$\textbf{data}\,\textbf{Suit} = \text{Spade} \vee \text{Heart} \vee \text{Club} \vee \text{Diamond} \tag{122}$$

Data with parameters:[*73]

$$\textbf{data}\,\mathbf{V^2} = \mathrm{V}^2\,\{x :: \mathbb{Z}, y :: \mathbb{Z}\} \tag{123}$$

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

[*72] In Haskell,

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

[*73] In Haskell,

0.28 型クラスとインスタンス

0.29 IO モナド

IO example:[*74]

$$main = getLine \mathbin{-\!\!\heartsuit\!\!\rightarrow} print \gg^{\star} \langle 0 \rangle \tag{124}$$

0.30 Do 構文

Do notation:[*75]

$$z_{\star} = \mathsf{do}\,\{x' \leftarrow x_{\star}; y' \leftarrow y_{\star}; fx'; gy'\} \tag{125}$$

0.31 モノイド

任意の関数 f に対して

$$\mathrm{id}\, f = f \tag{126}$$

なる関数 id があり，かつ任意の関数 f, g, h に対して

$$(h \bullet g) \bullet f = h \bullet (g \bullet f) \tag{127}$$

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

or `data V2 = V2 Int Int`.

[*74] In Haskell, `main = getLine >>= print >> return 0`.

[*75] In Haskell, `z = do {x' <- x; y' <- y; f x'; g y'}`.

が成り立つとする．このとき関数は**モノイド**であるという．

TK. 一般のモノイド．

0.32　モノイド則

型 $\mathbf{a}$ の変数 $x, y, z :: \mathbf{a}$ について，特別な変数 $i :: \mathbf{a}$ および二項演算子 $\bigcirc$ ただし $x \bigcirc y :: \mathbf{a}$ があり，

$$i \bigcirc x = x \ldots (\text{単位元の存在}) \tag{128}$$
$$(x \bigcirc y) \bigcirc z = x \bigcirc (y \bigcirc z) \ldots (\text{結合律}) \tag{129}$$

であるとき，組み合わせ $(\mathbf{a}, \bigcirc, i)$ をモノイドと呼ぶ．

組み合わせ $(\mathbb{Z}, +, 0)$ や $(\mathbb{Z}, \times, 1)$ はモノイドである．

同じ型から同じ型への 1 引数関数を改めて $\mathbf{a} \to \mathbf{a}$ で表し，特別な変数 i を関数 id，二項演算子を $\bullet$ とすると以下の関係が成り立つ．

$$\mathrm{id} \bullet f = f \ldots (\text{単位元の存在}) \tag{130}$$
$$(h \bullet g) \bullet f = h \bullet (g \bullet f) \ldots (\text{結合律}) \tag{131}$$

そこで組み合わせ $(\mathbf{a} \to \mathbf{a}, \bullet, \mathrm{id})$ はモノイドであると言える．

0.33　関手則

関手のマップ演算子 $\textcircled{\scriptsize S}$ は以下の**関手則**に従う．

$$\mathrm{id} \textcircled{\scriptsize S} x_\star = \mathrm{id}\, x_\star \tag{132}$$
$$(g \bullet f) \textcircled{\scriptsize S} x_\star = ((g \textcircled{\scriptsize S}) \bullet (f \textcircled{\scriptsize S})) x_\star \tag{133}$$
$$= g \textcircled{\scriptsize S} (f \textcircled{\scriptsize S} x_\star) \tag{134}$$

関手則は**関手（数学）**に由来する．

圏 $\mathcal{C}$ の**対象**を $\boldsymbol{X}$ とする．圏 $\mathcal{D}$ の対象は関手（数学）$\mathfrak{F}$ によって対象 $\boldsymbol{X}$ と関係づけられる．圏 $\mathcal{C}$ における**射** $f : \boldsymbol{X} \to \boldsymbol{Y}$ が $\mathfrak{F}f : \mathfrak{F}\boldsymbol{X} \to \mathfrak{F}\boldsymbol{Y}$ に対応し，次の関係を満たす．

- $\boldsymbol{X} \in \mathcal{C}$ に対して $\mathfrak{F}\,\mathrm{id}_{\boldsymbol{X}} = \mathrm{id}_{\mathfrak{F}\boldsymbol{X}}$
- $f : \boldsymbol{X} \to \boldsymbol{Y}$ および $g : \boldsymbol{Y} \to \boldsymbol{Z}$ に対して $\mathfrak{F}(g \bullet f) = (\mathfrak{F}g) \bullet (\mathfrak{F}f)$

いま

$$\underset{\boldsymbol{X}}{\mathrm{id}}, \underset{\mathfrak{F}\boldsymbol{X}}{\mathrm{id}} \to \mathrm{id} \tag{135}$$

$$f\text{Ⓢ} \to \mathfrak{F}f \tag{136}$$

と対応付けると，関手（数学）が満たす法則と関手則は一致する．

0.34　アプリカティブ関手則

アプリカティブ関手のマップ演算子 $\circledast$ は以下の規則に従う．

$$^{\star}\langle \mathrm{id} \rangle \circledast x_{\star} = x_{\star} \tag{137}$$

$$^{\star}\langle f \rangle \circledast {}^{\star}\langle x \rangle = {}^{\star}\langle fx \rangle \tag{138}$$

$$f_{\star} \circledast {}^{\star}\langle x \rangle = {}^{\star}\langle \Diamond \S x \rangle \circledast f_{\star} \tag{139}$$

$$^{\star}\langle \Diamond \bullet \Diamond \rangle \circledast h_{\star} \circledast g_{\star} \circledast f_{\star} = h_{\star} \circledast (g_{\star} \circledast f_{\star}) \tag{140}$$

0.35　モナド則

モナドのマップ演算子 $\heartsuit$ は以下の規則に従う．

$$f^{\dagger} \heartsuit {}^{\star}\langle x\rangle = f^{\dagger} x \tag{141}$$

$${}^{\star}\langle\diamond\rangle \heartsuit x_{\star} = x_{\star} \tag{142}$$

$$(g^{\dagger} \heartsuit f^{\dagger}) \heartsuit x_{\star} = g^{\dagger} \heartsuit (f^{\dagger} \heartsuit x_{\star}) \tag{143}$$

次の**クライスリスター**すなわち

$$f^{\bigstar} = (f^{\dagger} \heartsuit \diamond) \tag{144}$$

を用いると，モナド則は次のように書き換えられる．

$$\left(f^{\bigstar}\right)^{\star}\langle x\rangle = f^{\dagger} x \tag{145}$$

$$({}^{\star}\langle\diamond\rangle)^{\bigstar} x_{\star} = x_{\star} \tag{146}$$

$$\left(g^{\bigstar} f^{\dagger}\right)^{\bigstar} x_{\star} = g^{\bigstar}\left(f^{\bigstar} x_{\star}\right) \tag{147}$$