

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 n 以降のアルファベットを使う。

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}

$$z = fx \quad (3)$$

関数 f が引数をふたつ取る場合は、次のように書く。^{*4}

$$z = fxy \quad (4)$$

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 IO survival kit

$$f :: \mathbb{Z} \rightarrow \mathbb{Z} \quad (5)$$

$$f = \dots \quad (6)$$

$$main = putStrLn \bullet show \bullet f \bullet (read :: \mathbf{String} \rightarrow \mathbb{Z}) \heartsuit getLine \quad (7)$$

or

$$f :: \mathbb{Z} \rightarrow \mathbb{Z} \quad (8)$$

$$f = \dots \quad (9)$$

$$main = print \bullet f \bullet (read :: \mathbf{String} \rightarrow \mathbb{Z}) \heartsuit getLine \quad (10)$$

or

$$main = print \bullet f \heartsuit (read \textcircled{\mathbf{S}} getLine :: \mathbf{IO} \llbracket \mathbb{Z} \rrbracket) \quad (11)$$

^{*3} Haskell では $z = f\ x$ と書く。

^{*4} Haskell では $z = f\ x\ y$ と書く。

IO survival kit 2.

1 2 3

4 5 6

getContents

lines

words($\textcircled{\text{S}}$)

(read($\textcircled{\text{S}}$))($\textcircled{\text{S}}$)

$f x_s = \text{sqrt} \bullet \text{fromIntegral} \bullet \text{sum} \S (\backslash x \mapsto x * x) \otimes x_s$

$f(\textcircled{\text{S}})$

print

$$f :: \mathbb{Z} \rightarrow \mathbb{Z} \tag{12}$$

$$f = \dots \tag{13}$$

$$\text{ri} :: \mathbf{String} \rightarrow \mathbb{Z} \tag{14}$$

$$\text{ri} = \text{read} \tag{15}$$

$$\text{main} = \text{print} \heartsuit (((f \bullet \text{ri} \otimes) \otimes) \bullet (\text{words} \otimes) \bullet \text{lines} \textcircled{\text{S}} \text{getContents}) \tag{16} \blacksquare$$

`f :: Int -> Int`

`f = (2*)`

`readInt :: String -> Int`

`readInt = read`

`main = print . ((f <$>) <$>) . ((readInt <$>) <$>) . (words <$>) .`

0.4 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}

$$f = \lambda x \mapsto x + 1 \quad (17)$$

本書では無名変数 \diamond を用いた以下の書き方も用いる。^{*6}

$$f = (\diamond + 1) \quad (18)$$

$$= \lambda x \mapsto x + 1 \quad (19)$$

0.5 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}

$$z = \text{let } \{y = 1\} \text{ in } x + y \quad (20)$$

ローカル変数の定義は次のように後置できる。^{*8}

$$z = x + y \text{ where } \{y = 1\} \quad (21)$$

^{*5} Haskell では $f = \lambda x \rightarrow x + 1$ 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。

^{*7} Haskell では $z = \text{let } \{y = 1\} \text{ in } x + y$ 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 $z = x + y \text{ where } \{y = 1\}$ と書く。where 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。

0.6 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。

$$f n = \lambda x \mapsto n + x \quad (22)$$

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。

$$n = 3 \quad (23)$$

$$g = f n \quad (24)$$

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 $g 1$ 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。

0.7 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

^{*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。

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}

$$x :: \mathbb{Z} \quad (25)$$

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}

$$f :: \mathbb{Z} \rightarrow \mathbb{Z} \quad (26)$$

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。^{*14}

2 引数関数の方は次のように注釈できる。^{*15}

$$f :: \mathbb{Z} \rightarrow \mathbb{Z} \rightarrow \mathbb{Z} \quad (27)$$

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 $\mathbb{Z} \rightarrow \mathbb{Z} \rightarrow \mathbb{Z}$ は $\mathbb{Z} \rightarrow (\mathbb{Z} \rightarrow \mathbb{Z})$ と解釈される。

$(\mathbb{Z} \rightarrow \mathbb{Z})$ 型の関数を受け取り、 $(\mathbb{Z} \rightarrow \mathbb{Z})$ 型の関数を返す関数は次の型を持つ。^{*16}

$$f :: (\mathbb{Z} \rightarrow \mathbb{Z}) \rightarrow (\mathbb{Z} \rightarrow \mathbb{Z}) \quad (28)$$

なお後半の括弧は省略可能なので

$$f :: (\mathbb{Z} \rightarrow \mathbb{Z}) \rightarrow \mathbb{Z} \rightarrow \mathbb{Z} \quad (29)$$

^{*11} Haskell では Unicode 文字型を **Char** と書く。

^{*12} Haskell では $x :: \text{Int}$ と書く。

^{*13} Haskell では $f :: \text{Int} \rightarrow \text{Int}$ と書く。

^{*14} 正確には \rightarrow は型コンストラクタである。

^{*15} Haskell では $f :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$ と書く。

^{*16} Haskell では以下のように書く。

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

と書いても良い.

0.8 リテラル

整数型のリテラルは次のように書く.*¹⁷

$$x = 1 \tag{30}$$

変数の束縛と型注釈を組み合わせると次のようになる.*¹⁸

$$x :: \mathbb{Z} = 1 \tag{31}$$

同様に、浮動小数点型のリテラルは次のように書く.*¹⁹

$$x = 1.0 \tag{32}$$

文字型のリテラルは次のように書く.*²⁰

$$x = 'a' \tag{33}$$

論理型のリテラルは次のように書く.*²¹

$$x = \text{True} \tag{34}$$

論理型リテラルは True の他に False がある. なお True と False は正しくは**値コンストラクタ**である.

*¹⁷ Haskell では `x = 1` と書く.

*¹⁸ Haskell では `x :: Int = 1` と書く.

*¹⁹ Haskell では `x = 1.0` と書く.

*²⁰ Haskell では `x = 'a'` と書く.

*²¹ Haskell では `x = True` と書く.

0.9 条件

条件分岐は次のように書く.*²²

$$z = \text{if } x > 0 \text{ then } x \text{ else } -x \quad (35)$$

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.*²³

$$f = \text{case } x \text{ of } \begin{cases} 1 \rightarrow 1 \\ _ \rightarrow 0 \end{cases} \quad (36)$$

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.*²⁴

$$\begin{cases} f1 = 1 \\ f_ = 0 \end{cases} \quad (37)$$

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

*²² Haskell では `z = if x>0 then x else -x` と書く.

*²³ 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
```

*²⁴ Haskell では次のように書く.

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

きる.*25

$$\begin{cases} fx \mid x>0 = x \\ \mid \text{otherwise} = -x \end{cases} \quad (38)$$

ここに otherwise は `_` の別名である.

0.10 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.*26

$$\begin{cases} \text{fib } 0 = 0 \\ \text{fib } 1 = 1 \\ \text{fib } n = \text{fib } (n-1) + \text{fib } (n-2) \end{cases} \quad (39)$$

0.11 関数合成

関数の**合成**は次のように書く.*27

$$k = g \bullet f \quad (40)$$

*25 Haskell では次のように書く.

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

*26 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)
```

*27 Haskell では `k = g.f` と書く.

関数合成演算子 \bullet は以下のように右結合する.

$$k = h \bullet g \bullet f \quad (41)$$

$$= (h \bullet g) \bullet f \quad (42)$$

$$(43)$$

関数適用のための特別な演算子 \S があると便利である. 演算子 \S は関数合成演算子よりも優先順位が低い. 例を挙げる.*²⁸

$$z = h \S (g \bullet f)x \quad (44)$$

$$= h ((g \bullet f)x) \quad (45)$$

いま任意の関数 f に対して

$$\text{id}f = f \quad (46)$$

なる関数 id があり, かつ任意の関数 f, g, h に対して

$$(h \bullet g) \bullet f = h \bullet (g \bullet f) \quad (47)$$

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.

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

$$z = (x, y) \quad (48)$$

*²⁸ Haskell では $z = h \$ (g.f) x$ と書く.

*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}

$$z :: (\mathbb{Z}, \mathbb{Z}) \quad (49)$$

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}

$$z = () \quad (50)$$

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}

$$z :: () \quad (51)$$

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}

$$x_s = [] \quad (52)$$

任意のリストは次のように構成する。

$$x_s = x_0 : x_1 : x_2 : \cdots : [] \quad (53)$$

^{*30} In Haskell, $z :: (\text{Int}, \text{Int})$.

^{*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}

$$x_s :: [\mathbb{Z}] \quad (54)$$

リストは次のように構成することもできる.^{*35}

$$x_s = [1, 2, \dots, 100] \quad (55)$$

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.^{*36}

$$z_s = x_s \# y_s \quad (56)$$

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.^{*37}

$$f :: [\mathbb{Z}] \rightarrow \mathbb{Z} \quad (57)$$

$$f(x : x_s) = x \quad (58)$$

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.^{*38}

$$x_s = [x^2 \mid x \in [1, 2 \dots 100], x > 50] \quad (59)$$

^{*34} Haskell では `xs :: [Int]` と書く.

^{*35} Haskell では `xs = [1, 2..100]` と書く.

^{*36} Haskell では `zs = xs ++ ys` と書く.

^{*37} Haskell では `f (x:xs) :: [Int] -> Int = x` と書く.

^{*38} Haskell では次のように書く.

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

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**として定義される.

$$\text{type String} = [\text{Char}] \quad (60)$$

文字列型のリテラルは次のように書く.*39

$$x :: \text{String} = \text{"Hello, World!"} \quad (61)$$

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.*40

$$z_s = f \otimes x_s \quad (62)$$

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.*41

$$z = \bigcup_{0}^{(\diamond + \diamond)} x_s \quad (63)$$

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

$$\bigcup_a^* x_s = a \bowtie x_0 \bowtie x_1 \dots x_{n-1} \bowtie x_n \quad (64)$$

*39 Haskell では `x :: String = "Hello, World!"` と書く.

*40 Haskell では `zs = f 'map' xs` と書く.

*41 Haskell では `z = foldl 0 (+) xs` 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.*42

$$\bigsqcup_a x_s = a \star (x_0 \dots (x_{n-2} \star (x_{n-1} \star x_n))) \quad (65)$$

0.17 Maybe

計算は失敗する可能性がある. 例えば

$$z = y/x \quad (66)$$

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 \emptyset (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.

$$f y x = \text{if } x \neq 0 \text{ then } y/x \text{ else } \emptyset \dots (\text{不完全}) \quad (67)$$

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

$$f^\dagger y x = \text{if } x \neq 0 \text{ then } \text{Just } \langle y/x \rangle \text{ else } \emptyset \quad (68)$$

*42 Haskell では `foldr` を用いる.

とする．ここに $\text{Just } \langle y/x \rangle$ 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 $? \langle \mathbb{Z} \rangle$ 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．^{*43}

$$x? :: ? \langle \mathbb{Z} \rangle \quad (69)$$

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

$$x? = \text{Just } \langle 1 \rangle \quad (70)$$

のように書く．^{*44}

Maybe 変数が値を持たない場合は

$$x? = \emptyset \quad (71)$$

と書く．^{*45}

0.18 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 \textcircled{S} を用いる．^{*46}

$$z? = (\diamond + 1) \textcircled{S} x? \quad (72)$$

^{*43} Haskell では `xm :: Maybe Int` と書く．

^{*44} Haskell では `xm = Just 1` と書く．

^{*45} Haskell では `xm = Nothing` と書く．

^{*46} Haskell では `zm = (+1) <$> xm` と書く．

ここに演算子 \textcircled{S} は

$$\text{Just } \langle fx \rangle = f \textcircled{S} \text{Just } \langle x \rangle \quad (73)$$

$$\emptyset = f \textcircled{S} \emptyset \quad (74)$$

と定義される.

0.19 Maybe 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.

$$x? = \text{Just } \langle [1, 2, \dots, 100] \rangle \quad (75)$$

のとき, リストの各要素に関数 $f :: \mathbb{Z} \rightarrow \mathbb{Z}$ を適用するには次のように書く.^{*47}

$$z? = (f \otimes) \textcircled{S} x? \quad (76)$$

0.20 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.^{*48}

$$f :: \mathbf{a} \rightarrow \mathbf{a} \quad (77)$$

^{*47} Haskell では $\mathbf{zm} = (f \text{ <\$>}) \text{ <\$> } \mathbf{xm}$ と書く. 最初の $\text{<\$>}$ はリストの各要素に関数 f を適用する演算子, 2 番目の $\text{<\$>}$ 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.

^{*48} Haskell では $f :: \mathbf{a} \rightarrow \mathbf{a}$ 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 Num である．型パラメタ \mathbf{a} が型クラス 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．^{*49}

$$f :: \text{Num} \supset \mathbf{a} \Rightarrow \mathbf{a} \rightarrow \mathbf{a} \quad (78)$$

型クラスは型に制約を与える．

TK. $\text{Num } \mathbf{a} \Rightarrow \mathbf{x} :: \mathbf{a}$ ならば \mathbf{x} が持つべき演算子．

TK. 型クラスの例．

0.21 関手

型 \mathbf{a} のリストの変数は

$$x_s :: [\mathbf{a}] \quad (79)$$

という型注釈を持つ．これは

$$x_s :: [] \llbracket \mathbf{a} \rrbracket \quad (80)$$

のシンタックスシュガーである．

型 \mathbf{a} 型の Maybe の変数は

$$x_? :: ? \llbracket \mathbf{a} \rrbracket \quad (81)$$

という型注釈を持つ．

^{*49} Haskell では $f :: \text{Num } \mathbf{a} \Rightarrow \mathbf{a} \rightarrow \mathbf{a}$ と書く．

普段遣いの関数

$$f :: \mathbf{a} \rightarrow \mathbf{a} \quad (82)$$

をリスト変数 x_s に適用する場合は

$$z_s = f \otimes x_s \quad (83)$$

とする. 同じく関数 f を Maybe 変数 $x_?$ に適用する場合は

$$z_? = f \textcircled{\text{S}} x_? \quad (84)$$

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 `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.^{*50}

$$x_* :: \text{Functor} \supset \mathbf{f} \Rightarrow^{\mathbf{f}} \langle\langle \mathbf{a} \rangle\rangle \quad (85)$$

型クラス `Functor` に属する型は $\textcircled{\text{S}}$ 演算子を持たねばならない. 演算子 $\textcircled{\text{S}}$ は次の形を持つ.^{*51}

$$z_* = f \textcircled{\text{S}} x_* \quad (86)$$

演算子 $\textcircled{\text{S}}$ の型は次のとおりである.

$$\diamond \textcircled{\text{S}} \diamond :: (\mathbf{a} \rightarrow \mathbf{b}) \rightarrow^{\mathbf{f}} \langle\langle \mathbf{a} \rangle\rangle \rightarrow^{\mathbf{f}} \langle\langle \mathbf{b} \rangle\rangle \quad (87)$$

^{*50} Haskell では `xm :: Functor f => f a` と書く.

^{*51} In Haskell, `zm = f <$> xm`.

もし変数 x_* の型がリストであれば

$$\textcircled{\text{S}} = \otimes \quad (88)$$

であると解釈する.

Function of parametric type with functor class:^{*52}

$$f :: \mathfrak{F}\text{unctor} \supset \mathbf{f} \Rightarrow \mathbf{a} \rightarrow \mathbf{f} \llbracket \mathbf{a} \rrbracket \quad (89)$$

Example function application:^{*53}

$$z_* = (\diamond + 1) \textcircled{\text{S}}^{\text{Just}} \langle x \rangle \quad (90)$$

0.22 関手としての関数

$$f :: \mathbf{q} \rightarrow \mathbf{r} \quad (91)$$

Function as a functor:^{*54}

$$f :: (\blacklozenge \rightarrow \mathbf{r}) \mathbf{q} = (\blacklozenge \rightarrow \mathbf{r}) \llbracket \mathbf{q} \rrbracket \quad (92)$$

Thus,

$$f_2 \bullet f_1 \equiv f_2 \textcircled{\text{S}} f_1 \quad (93)$$

$$\text{id} \bullet f = \text{id} f = f \quad (94)$$

$$(h \bullet g) \bullet f = ((h \bullet) \bullet (g \bullet)) f \quad (95)$$

$$= h \bullet (g \bullet f) \quad (96)$$

^{*52} In Haskell, $f :: \text{Functor } f \Rightarrow a \rightarrow f \ a$.

^{*53} In Haskell, $zm = (+1) \langle \$ \rangle \text{ Just } x$.

^{*54} In Haskell, $f :: ((\rightarrow) \ r) \ q$.

0.23 アプリカティブ関手

Pure:^{*55}

$$z_{\star} = \star \langle x \rangle \quad (97)$$

Applicative map:^{*56}

$$z_{\star} = f_{\star} \otimes x_{\star} \quad (98)$$

where

$$f_{\star} :: \mathbf{f} \ll \mathbf{a} \rightarrow \mathbf{b} \gg \quad (99)$$

Applicative style:^{*57}

$$z_{\star} = \star \langle f \rangle \otimes x_{\star} \otimes y_{\star} \quad (100)$$

or^{*58}

$$z_{\star} = f \textcircled{\text{S}} x_{\star} \otimes y_{\star} \quad (101)$$

or^{*59}

$$z_{\star} = \llbracket f \ x_{\star} \ y_{\star} \rrbracket \quad (102)$$

^{*55} In Haskell, `zm = pure x`.

^{*56} In Haskell, `zm = f <*> xm`.

^{*57} In Haskell, `zm = pure (+) <*> xm <*> ym`.

^{*58} In Haskell, `zm = f <$> xm <*> ym`.

^{*59} In Haskell, `zm = liftA2 f xm ym`.

0.24 モナド

Returning *List*.

$$\cdot \quad (103)$$

Returning *Maybe*:^{*60}

$$f :: \mathbb{Z} \rightarrow ? \langle \mathbb{Z} \rangle \quad (104)$$

$$fx = \text{Just } \langle x \rangle \quad (105)$$

Returning *monad*:

$$f :: \mathbb{Z} \rightarrow ^{\mathbf{m}} \langle \mathbf{a} \rangle \quad (106)$$

$$fx = * \langle x \rangle \quad (107)$$

Returning monadic value:^{*61}

$$f :: \mathcal{M}onad \supset \mathbf{m} \Rightarrow \mathbf{a} \rightarrow ^{\mathbf{m}} \langle \mathbf{a} \rangle \quad (108)$$

Monadic function binding:^{*62}

$$z_* = x_* \multimap f_1 \multimap f_2 \quad (109)$$

where

$$f_1 :: \mathbb{Z} \rightarrow ? \langle \mathbb{Z} \rangle \quad (110)$$

$$f_2 :: \mathbb{Z} \rightarrow ? \langle \mathbb{Z} \rangle. \quad (111)$$

^{*60} In Haskell, $f :: \text{Int} \rightarrow \text{Maybe Int}$ and $f\ x = \text{Just } x$.

^{*61} In Haskell, $f :: \text{Monad } m \Rightarrow a \rightarrow m\ a$.

^{*62} In Haskell, $zm = xm \gg= f1 \gg= f2$.

Function binding of monadic function and non-monadic function:^{*63}

$$z_* = x_* \multimap f \multimap g' \text{ where } \{g'w = * \langle gw \rangle\} \quad (112)$$

or

$$z_* = x_* \multimap (f \multimap g') \text{ where } \{g'w = * \langle gw \rangle\} \quad (113)$$

where

$$f :: \mathbb{Z} \rightarrow ? \langle \mathbb{Z} \rangle \quad (114)$$

$$g :: \mathbb{Z} \rightarrow \mathbb{Z}. \quad (115)$$

Another solution is:

$$z_* = (g^* \bullet f) \heartsuit x_* \quad (116)$$

where g^* means `liftM g` in Haskell.^{*64}

0.25 種

$$* \rightarrow * \quad (117)$$

^{*63} In Haskell,

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

^{*64} In Haskell, `zm = (liftM g . f) xm`.

0.26 Data

Data:^{*65}

data Suit = Spade ∨ Heart ∨ Club ∨ Diamond (118)

Data with parameters:^{*66}

data V² = V² { $x :: \mathbb{Z}, y :: \mathbb{Z}$ } (119)

0.27 型クラスとインスタンス

0.28 IO

IO example:^{*67}

main = *getLine* \rightarrow *print* >> *⟨0⟩ (120)

^{*65} In Haskell,

data Suit = Spade | Heart | Club | Diamond

^{*66} In Haskell,

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

or **data V2** = V2 Int Int.

^{*67} In Haskell, *main* = *getLine* >>= *print* >> *return* 0.

0.29 Do 構文

Do notation:^{*68}

$$z_* = \text{do} \{x' \leftarrow x_*; y' \leftarrow y_*; f x'; g y'\} \quad (121)$$



0.30 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 \circ ただし $x \circ y :: \mathbf{a}$ があり,

$$i \circ x = x \dots (\text{単位元の存在}) \quad (122)$$

$$(x \circ y) \circ z = x \circ (y \circ z) \dots (\text{結合律}) \quad (123)$$

であるとき, 組み合わせ (\mathbf{a}, \circ, 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 $\mathbf{a} \rightarrow \mathbf{a}$ 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.

$$\text{id} \bullet f = f \dots (\text{単位元の存在}) \quad (124)$$

$$(h \bullet g) \bullet f = h \bullet (g \bullet f) \dots (\text{結合律}) \quad (125)$$

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

^{*68} In Haskell, $z = \text{do} \{x' \leftarrow x; y' \leftarrow y; f x'; g y'\}.$

0.31 関手則

関手のマップ演算子 \textcircled{S} は以下の関手則に従う.

$$\text{id} \textcircled{S} x_* = \text{id} x_* \quad (126)$$

$$(g \bullet f) \textcircled{S} x_* = ((g \textcircled{S}) \bullet (f \textcircled{S})) x_* \quad (127)$$

$$= g \textcircled{S} (f \textcircled{S} x_*) \quad (128)$$

関手則は関手（数学）に由来する.

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

- $X \in \mathcal{C}$ に対して $\mathfrak{F}\text{id}_X = \text{id}_{\mathfrak{F}X}$
- $f: X \rightarrow Y$ および $g: Y \rightarrow Z$ に対して $\mathfrak{F}(g \bullet f) = (\mathfrak{F}g) \bullet (\mathfrak{F}f)$

いま

$$\text{id}_X, \text{id}_{\mathfrak{F}X} \rightarrow \text{id} \quad (129)$$

$$f \textcircled{S} \rightarrow \mathfrak{F}f \quad (130)$$

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.

0.32 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 \otimes は以下の規則に従う.

$$^*\langle \text{id} \rangle \otimes x_* = x_* \quad (131)$$

$$^*\langle f \rangle \otimes ^*\langle x \rangle = ^*\langle fx \rangle \quad (132)$$

$$f_* \otimes ^*\langle x \rangle = ^*\langle \diamond \S x \rangle \otimes f_* \quad (133)$$

$$^*\langle \diamond \bullet \diamond \rangle \otimes h_* \otimes g_* \otimes f_* = h_* \otimes (g_* \otimes f_*) \quad (134)$$

0.33 モナド則

モナドのマップ演算子 \heartsuit は以下の規則に従う.

$$f^\dagger \heartsuit ^*\langle x \rangle = f^\dagger x \quad (135)$$

$$^*\langle \diamond \rangle \heartsuit x_* = x_* \quad (136)$$

$$(g^\dagger \heartsuit f^\dagger) \heartsuit x_* = g^\dagger \heartsuit (f^\dagger \heartsuit x_*) \quad (137)$$

次のクライスリスターすなわち

$$f^\star = (f^\dagger \heartsuit \diamond) \quad (138)$$

を用いると, モナド則は次のように書き換えられる.

$$(f^\star)^\star \langle x \rangle = f^\dagger x \quad (139)$$

$$(^*\langle \diamond \rangle)^\star x_* = x_* \quad (140)$$

$$(g^\star f^\dagger)^\star x_* = g^\star (f^\star x_*) \quad (141)$$