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]

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

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 IO survival kit

$$f :: \mathbb{Z} \to \mathbb{Z} \tag{5}$$

$$f = \ldots \tag{6}$$

$$main = print \bullet f \bullet (\mathrm{read} :: \mathbf{String} \to \mathbb{Z}) \heartsuit\, getLine \tag{7}$$

or

$$main = print \bullet f \heartsuit (\mathrm{read} ⓢ\, getLine :: {}^{\mathbf{IO}} \langle\!\langle \mathbb{Z} \rangle\!\rangle) \tag{8}$$

[*3] Haskell では `z = f x` と書く．

[*4] Haskell では `z = f x y` と書く．

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]

$$f = \backslash x \mapsto x + 1 \tag{9}$$

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

$$\begin{aligned} f &= (\Diamond + 1) && (10) \\ &= \backslash x \mapsto x + 1 && (11) \end{aligned}$$

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]

$$z = \mathsf{let}\ \{y = 1\}\ \mathsf{in}\ x + y \tag{12}$$

ローカル変数の定義は次のように後置できる．[*8]

$$z = x + y\ \mathsf{where}\ \{y = 1\} \tag{13}$$

[*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．

[*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 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．

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

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．

$$n = 3 \tag{15}$$
$$g = f n \tag{16}$$

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]

$$x :: \mathbb{Z} \tag{17}$$

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]

$$f :: \mathbb{Z} \to \mathbb{Z} \tag{18}$$

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]

$$f :: \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z} \tag{19}$$

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．[*16]

$$f :: (\mathbb{Z} \to \mathbb{Z}) \to (\mathbb{Z} \to \mathbb{Z}) \tag{20}$$

なお後半の括弧は省略可能なので

$$f :: (\mathbb{Z} \to \mathbb{Z}) \to \mathbb{Z} \to \mathbb{Z} \tag{21}$$

[*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．

[*14] 正確には $\to$ は型コンストラクタである．

[*15] Haskell では `f :: Int -> Int -> 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]

$$x = 1 \tag{22}$$

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]

$$x :: \mathbb{Z} = 1 \tag{23}$$

同様に，浮動小数点型のリテラルは次のように書く.[*19]

$$x = 1.0 \tag{24}$$

文字型のリテラルは次のように書く.[*20]

$$x = \texttt{'a'} \tag{25}$$

論理型のリテラルは次のように書く.[*21]

$$x = \mathrm{True} \tag{26}$$

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]

$$z = \text{if } x > 0 \text{ then } x \text{ else } -x \tag{27}$$

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]

$$f = \text{case } x \text{ of } \begin{cases} 1 \dashrightarrow 1 \\ \_ \dashrightarrow 0 \end{cases} \tag{28}$$

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]

$$\begin{cases} f1 = 1 \\ f\_ = 0 \end{cases} \tag{29}$$

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]

$$\begin{cases} fx \mid_{x>0} = x \\ \quad\;\; \mid_{\text{otherwise}} = -x \end{cases} \tag{30}$$

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]

$$\begin{cases} \mathrm{fib}\,0 = 0 \\ \mathrm{fib}\,1 = 1 \\ \mathrm{fib}\,n = \mathrm{fib}(n-1) + \mathrm{fib}(n-2) \end{cases} \tag{31}$$

0.11　関数合成

関数の**合成**は次のように書く.[*27]

$$k = g \bullet f \tag{32}$$

[*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.

$$
\begin{aligned}
k &= h \bullet g \bullet f && (33)\\
&= (h \bullet g) \bullet f && (34)\\
& && (35)
\end{aligned}
$$

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.[*28]

$$
\begin{aligned}
z &= h \,\S\, (g \bullet f)x && (36)\\
&= h\,((g \bullet f)x) && (37)
\end{aligned}
$$

いま任意の関数 f に対して

$$
\mathrm{id} f = f \tag{38}
$$

なる関数 id があり, かつ任意の関数 f, g, h に対して

$$
(h \bullet g) \bullet f = h \bullet (g \bullet f) \tag{39}
$$

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]

$$
z = (x, y) \tag{40}
$$

[*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]

$$z :: (\mathbb{Z}, \mathbb{Z}) \tag{41}$$

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]

$$z = () \tag{42}$$

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]

$$z :: () \tag{43}$$

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]

$$x_{\mathrm{s}} = [\,] \tag{44}$$

任意のリストは次のように構成する．

$$x_{\mathrm{s}} = x_0 : x_1 : x_2 : \cdots : [\,] \tag{45}$$

[*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]

$$x_{\mathrm{s}} :: [\mathbb{Z}] \tag{46}$$

リストは次のように構成することもできる．[*35]

$$x_{\mathrm{s}} = [1, 2, \ldots, 100] \tag{47}$$

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]

$$z_{\mathrm{s}} = x_{\mathrm{s}} \# y_{\mathrm{s}} \tag{48}$$

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]

$$f :: [\mathbb{Z}] \to \mathbb{Z} \tag{49}$$

$$f(x : x_{\mathrm{s}}) = x \tag{50}$$

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]

$$x_{\mathrm{s}} = \left[x^2 \mid x \in [1, 2 \ldots 100], x > 50\right] \tag{51}$$

[*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．

$$\text{type } \mathbf{String} = [\mathbf{Char}] \tag{52}$$

文字列型のリテラルは次のように書く．[*39]

$$x :: \mathbf{String} = \texttt{"Hello, World!"} \tag{53}$$

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]

$$z_\mathrm{s} = f \otimes x_\mathrm{s} \tag{54}$$

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]

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

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

$$\bigcup_{a}^{✠} x_\mathrm{s} = a ✠ x_0 ✠ x_1 \ldots x_{n-1} ✠ x_n \tag{56}$$

[*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]

$$\bigsqcup_{a}^{\maltese} x_{\mathrm{s}} = a \maltese (x_0 \ldots (x_{n-2} \maltese (x_{n-1} \maltese x_n))) \tag{57}$$

0.17 Maybe

計算は失敗する可能性がある．例えば

$$z = y/x \tag{58}$$

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．

$$f y x = \text{if } x \neq 0 \text{ then } y/x \text{ else } \varnothing \ldots (\text{不完全}) \tag{59}$$

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

$$f^{\dagger} y x = \text{if } x \neq 0 \text{ then } {}^{\mathrm{Just}} \langle y/x \rangle \text{ else } \varnothing \tag{60}$$

[*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 ${}^{?}\langle\!\langle \mathbb{Z} \rangle\!\rangle$ 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．[*43]

$$x_? :: {}^{?}\langle\!\langle \mathbb{Z} \rangle\!\rangle \tag{61}$$

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

$$x_? = {}^{\text{Just}}\langle 1 \rangle \tag{62}$$

のように書く．[*44]

Maybe 変数が値を持たない場合は

$$x_? = \varnothing \tag{63}$$

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 Ⓢ を用いる．[*46]

$$z_? = (\Diamond + 1) \circledS x_? \tag{64}$$

[*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 Ⓢ は

$$ {}^{\mathrm{Just}}\langle fx\rangle = f \circledS {}^{\mathrm{Just}}\langle x\rangle \tag{65} $$

$$ \varnothing = f \circledS \varnothing \tag{66} $$

と定義される.

0.19 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.

$$ x_? = {}^{\mathrm{Just}}\langle[1, 2, \ldots, 100]\rangle \tag{67} $$

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.[*47]

$$ z_? = (f\otimes) \circledS x_? \tag{68} $$

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]

$$ f :: \mathbf{a} \to \mathbf{a} \tag{69} $$

[*47] 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.

[*48] 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．[*49]

$$f :: \mathfrak{Num} \supset \mathbf{a} \Rightarrow \mathbf{a} \to \mathbf{a} \tag{70}$$

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.

0.21 関手

型 $\mathbf{a}$ のリストの変数は

$$x_{\mathrm{s}} :: [\mathbf{a}] \tag{71}$$

という型注釈を持つ．これは

$$x_{\mathrm{s}} :: {}^{[]}\langle\!\langle \mathbf{a} \rangle\!\rangle \tag{72}$$

のシンタックスシュガーである．

型 $\mathbf{a}$ 型の Maybe の変数は

$$x_{?} :: {}^{?}\langle\!\langle \mathbf{a} \rangle\!\rangle \tag{73}$$

という型注釈を持つ．

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

普段遣いの関数

$$f :: \mathbf{a} \to \mathbf{a} \tag{74}$$

をリスト変数 x_s に適用する場合は

$$z_\mathrm{s} = f \otimes x_\mathrm{s} \tag{75}$$

とする．同じく関数 f を Maybe 変数 $x_?$ に適用する場合は

$$z_? = f \circledS x_? \tag{76}$$

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．[*50]

$$x_\star :: \mathfrak{Functor} \supset \mathbf{f} \Rightarrow {}^{\mathbf{f}}\langle\!\langle \mathbf{a} \rangle\!\rangle \tag{77}$$

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．[*51]

$$z_\star = f \circledS x_\star \tag{78}$$

演算子 $\circledS$ の型は次のとおりである．

$$\lozenge \circledS \lozenge :: (\mathbf{a} \to \mathbf{b}) \to {}^{\mathbf{f}}\langle\!\langle \mathbf{a} \rangle\!\rangle \to {}^{\mathbf{f}}\langle\!\langle \mathbf{b} \rangle\!\rangle \tag{79}$$

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

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

もし変数 $x_\star$ の型がリストであれば

$$ⓢ = \otimes \tag{80}$$

であると解釈する．

Function of parametric type with functor class:[*52]

$$f :: \mathfrak{Functor} \supset \mathbf{f} \Rightarrow \mathbf{a} \to {}^{\mathbf{f}}\langle\!\langle \mathbf{a} \rangle\!\rangle \tag{81}$$

Example function application:[*53]

$$z_\star = (\Diamond + 1)\ ⓢ\ {}^{\mathrm{Just}}\langle x \rangle \tag{82}$$

0.22　関手としての関数

$$f :: \mathbf{q} \to \mathbf{r} \tag{83}$$

Function as a functor:[*54]

$$f :: (\blacklozenge \to \mathbf{r})\, \mathbf{q} = {}^{(\blacklozenge \to \mathbf{r})}\langle\!\langle \mathbf{q} \rangle\!\rangle \tag{84}$$

Thus,

$$f_2 \bullet f_1 \equiv f_2\ ⓢ\ f_1 \tag{85}$$

$$\mathrm{id} \bullet f = \mathrm{id} f = f \tag{86}$$

$$(h \bullet g) \bullet f = ((h\bullet) \bullet (g\bullet))f \tag{87}$$

$$= h \bullet (g \bullet f) \tag{88}$$

*52 In Haskell, `f :: Functor f => a -> f a`.
*53 In Haskell, `zm = (+1) <$> Just x`.
*54 In Haskell, `f :: ((->) r) q`.

0.23 アプリカティブ関手

Pure:[*55]

$$z_\star = {}^\star\langle x\rangle \tag{89}$$

Applicative map:[*56]

$$z_\star = f_\star \circledast x_\star \tag{90}$$

where

$$f_\star :: {}^{\mathbf{f}}\langle\!\langle \mathbf{a} \to \mathbf{b} \rangle\!\rangle \tag{91}$$

Applicative style:[*57]

$$z_\star = {}^\star\langle f\rangle \circledast x_\star \circledast y_\star \tag{92}$$

or[*58]

$$z_\star = f \circledS x_\star \circledast y_\star \tag{93}$$

or[*59]

$$z_\star = [\![f\, x_\star\, y_\star]\!] \tag{94}$$

---

*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*.

$$. \tag{95}$$

Returning *Maybe*:[*60]

$$f :: \mathbb{Z} \to {}^{?} \langle\!\langle \mathbb{Z} \rangle\!\rangle \tag{96}$$

$$fx = {}^{\text{Just}} \langle x \rangle \tag{97}$$

Returning *monad*:

$$f :: \mathbb{Z} \to {}^{\mathbf{m}} \langle\!\langle \mathbf{a} \rangle\!\rangle \tag{98}$$

$$fx = {}^{\star} \langle x \rangle \tag{99}$$

Returning monadic value:[*61]

$$f :: \mathfrak{Monad} \supset \mathbf{m} \Rightarrow \mathbf{a} \to {}^{\mathbf{m}} \langle\!\langle \mathbf{a} \rangle\!\rangle \tag{100}$$

Monadic function binding:[*62]

$$z_\star = x_\star \twoheadrightarrow f_1 \twoheadrightarrow f_2 \tag{101}$$

where

$$f_1 :: \mathbb{Z} \to {}^{?} \langle\!\langle \mathbb{Z} \rangle\!\rangle \tag{102}$$

$$f_2 :: \mathbb{Z} \to {}^{?} \langle\!\langle \mathbb{Z} \rangle\!\rangle . \tag{103}$$

[*60] In Haskell, `f :: Int -> Maybe Int` and `f x = Just x`.
[*61] In Haskell, `f :: Monad m => a -> m a`.
[*62] In Haskell, `zm = xm >>= f1 >>= f2`.

Function binding of monadic function and non-monadic function:[*63]

$$z_\star = x_\star \mathrel{-\!\!\heartsuit\!\!\!\rightarrow} f \mathrel{-\!\!\heartsuit\!\!\!\rightarrow} g' \text{ where } \{g'w = {}^\star \langle gw \rangle\} \tag{104}$$

or

$$z_\star = x_\star \mathrel{-\!\!\heartsuit\!\!\!\rightarrow} (f \rightarrowtail g') \text{ where } \{g'w = {}^\star \langle gw \rangle\} \tag{105}$$

where

$$f :: \mathbb{Z} \to {}^{?}\langle\!\langle \mathbb{Z} \rangle\!\rangle \tag{106}$$
$$g :: \mathbb{Z} \to \mathbb{Z}. \tag{107}$$

Another solution is:

$$z_\star = (g^* \bullet f) \heartsuit x_\star \tag{108}$$

where g^* means `liftM g` in Haskell.[*64]

0.25　種

$$\star \to \star \tag{109}$$

*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]

$$\textbf{data}\,\textbf{Suit} = \mathrm{Spade} \vee \mathrm{Heart} \vee \mathrm{Club} \vee \mathrm{Diamond} \tag{110}$$

Data with parameters:[*66]

$$\textbf{data}\,\mathbf{V^2} = \mathrm{V}^2\,\{x :: \mathbb{Z}, y :: \mathbb{Z}\} \tag{111}$$

0.27 型クラスとインスタンス

0.28 IO

IO example:[*67]

$$\mathit{main} = \mathit{getLine} \overset{\heartsuit}{\rightarrow} \mathit{print} \gg {}^{\star}\,\langle 0 \rangle \tag{112}$$

[*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]

$$z_\star = \mathsf{do}\,\{x' \leftarrow x_\star; y' \leftarrow y_\star; f x'; g y'\} \tag{113}$$

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

$$i \bigcirc x = x \ldots \text{（単位元の存在）} \tag{114}$$

$$(x \bigcirc y) \bigcirc z = x \bigcirc (y \bigcirc z) \ldots \text{（結合律）} \tag{115}$$

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．

$$\mathrm{id} \bullet f = f \ldots \text{（単位元の存在）} \tag{116}$$

$$(h \bullet g) \bullet f = h \bullet (g \bullet f) \ldots \text{（結合律）} \tag{117}$$

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．

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

0.31　関手則

関手のマップ演算子 Ⓢ は以下の**関手則**に従う．

$$\mathrm{id} \,Ⓢ\, x_\star = \mathrm{id}x_\star \tag{118}$$
$$(g \bullet f) \,Ⓢ\, x_\star = ((gⓈ) \bullet (fⓈ))x_\star \tag{119}$$
$$= g \,Ⓢ\, (f \,Ⓢ\, x_\star) \tag{120}$$

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

$$\mathrm{id}_{\boldsymbol{X}}, \mathrm{id}_{\mathfrak{F}\boldsymbol{X}} \to \mathrm{id} \tag{121}$$
$$fⓈ \to \mathfrak{F}f \tag{122}$$

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 $\circledast$ は以下の規則に従う．

$$^{\star}\langle \mathrm{id} \rangle \circledast x_{\star} = x_{\star} \tag{123}$$

$$^{\star}\langle f \rangle \circledast {}^{\star}\langle x \rangle = {}^{\star}\langle fx \rangle \tag{124}$$

$$f_{\star} \circledast {}^{\star}\langle x \rangle = {}^{\star}\langle \Diamond \, \S \, x \rangle \circledast f_{\star} \tag{125}$$

$$^{\star}\langle \Diamond \bullet \Diamond \rangle \circledast h_{\star} \circledast g_{\star} \circledast f_{\star} = h_{\star} \circledast (g_{\star} \circledast f_{\star}) \tag{126}$$

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．

$$f^{\dagger} \heartsuit {}^{\star}\langle x \rangle = f^{\dagger} x \tag{127}$$

$$^{\star}\langle \Diamond \rangle \heartsuit x_{\star} = x_{\star} \tag{128}$$

$$(g^{\dagger} \heartsuit f^{\dagger}) \heartsuit x_{\star} = g^{\dagger} \heartsuit (f^{\dagger} \heartsuit x_{\star}) \tag{129}$$

次の**クライスリスター**すなわち

$$f^{\bigstar} = (f^{\dagger} \heartsuit \Diamond) \tag{130}$$

を用いると，モナド則は次のように書き換えられる．

$$\left(f^{\bigstar}\right) {}^{\star}\langle x \rangle = f^{\dagger} x \tag{131}$$

$$({}^{\star}\langle \Diamond \rangle)^{\bigstar} x_{\star} = x_{\star} \tag{132}$$

$$\left(g^{\bigstar} f^{\dagger}\right)^{\bigstar} x_{\star} = g^{\bigstar} \left(f^{\bigstar} x_{\star}\right) \tag{133}$$