

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 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}

$$f = \lambda x \mapsto x + 1 \quad (5)$$

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

$$f = (\diamond + 1) \quad (6)$$

$$= \lambda x \mapsto x + 1 \quad (7)$$

^{*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 $f = \lambda x \rightarrow x+1$ 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。

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

$$z = \text{let } \{y = 1\} \text{ in } x + y \quad (8)$$

ローカル変数の定義は次のように後置できる.*⁸

$$z = x + y \text{ where } \{y = 1\} \quad (9)$$

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。

$$f n = \lambda x \mapsto n + x \quad (10)$$

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。

$$n = 3 \quad (11)$$

$$g = f n \quad (12)$$

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 $z = \text{let } \{y = 1\} \text{ in } x+y$ 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 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}

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

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

$$f :: \mathbb{Z} \rightarrow \mathbb{Z} \rightarrow \mathbb{Z} \quad (15)$$

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 $\mathbb{Z} \rightarrow \mathbb{Z} \rightarrow \mathbb{Z}$ 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.*¹⁵

$$f :: (\mathbb{Z} \rightarrow \mathbb{Z}) \rightarrow (\mathbb{Z} \rightarrow \mathbb{Z}) \quad (16)$$

なお後半の括弧は省略可能なので

$$f :: (\mathbb{Z} \rightarrow \mathbb{Z}) \rightarrow \mathbb{Z} \rightarrow \mathbb{Z} \quad (17)$$

と書いても良い。

0.7 リテラル

リテラルは次のように書く.*¹⁶

$$z = 1 \quad (18)$$

*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 = 1` と書く。

0.8 条件

条件分岐は次のように書く.^{*17}

$$z = \text{if } x > 0 \text{ then } x \text{ else } -x \quad (19)$$

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.^{*18}

$$f = \text{case } x \text{ of } \begin{cases} 1 \rightarrow 1 \\ _ \rightarrow 0 \end{cases} \quad (20)$$

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.^{*19}

$$\begin{cases} f1 = 1 \\ f_ = 0 \end{cases} \quad (21)$$

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

^{*17} Haskell では `z = if x>0 then x else -x` と書く.

^{*18} 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
```

^{*19} Haskell では次のように書く.

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

きる.*²⁰

$$\begin{cases} fx \mid x > 0 = x \\ \mid \text{otherwise} = -x \end{cases} \quad (22)$$

0.9 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．*²¹

$$\begin{cases} \text{fib } 0 = 0 \\ \text{fib } 1 = 1 \\ \text{fib } n = \text{fib}(n-1) + \text{fib}(n-2) \end{cases} \quad (23)$$

0.10 関数合成

関数の合成は次のように書く．*²²

$$k = g \bullet f \quad (24)$$

*²⁰ Haskell では次のように書く．

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

*²¹ 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)
```

*²² 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.

$$f = f_3 \bullet f_2 \bullet f_1 \quad (25)$$

$$= (f_3 \bullet f_2) \bullet f_1 \quad (26)$$

$$(27)$$

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.*²³

$$k = h \$ g \bullet f \quad (28)$$

$$= h \bullet (g \bullet f) \quad (29)$$

いま任意の関数 f に対して

$$\text{id } f = f \quad (30)$$

なる関数 id があり, かつ任意の関数 f, g, h に対して

$$(h \bullet g) \bullet f = h \bullet (g \bullet f) \quad (31)$$

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.

0.11 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.*²⁴

$$z = (x, y) \quad (32)$$

*²³ Haskell では $k = h \$ g.f$ と書く.

*²⁴ 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。^{*25}

$$z :: (\mathbb{Z}, \mathbb{Z}) \quad (33)$$

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。^{*26}

$$z = () \quad (34)$$

ユニットの型は**ユニット型**で、型注釈を次のように書く。^{*27}

$$z :: () \quad (35)$$

0.12 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。^{*28}

$$x_s = [] \quad (36)$$

任意のリストは次のように構成する。

$$x_s = x_0 : x_1 : x_2 : \cdots : [] \quad (37)$$

^{*25} In Haskell, $z :: (\text{Int}, \text{Int})$.

^{*26} Haskell では $z = ()$ と書く。

^{*27} Haskell では $z :: ()$ と書く。

^{*28} 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.*29

$$x_s :: [\mathbb{Z}] \quad (38)$$

リストは次のように構成することもできる.*30

$$x_s = [1, 2, \dots, 100] \quad (39)$$

リストの構成には**内包表記**が使える。例を挙げる.*31

$$x_s = [x^2 \mid x \in [1, 2 \dots 100], x > 50] \quad (40)$$

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.*32

$$z_s = x_s \# y_s \quad (41)$$

0.13 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.*33

$$z_s = f \otimes x_s \quad (42)$$

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

*29 Haskell では `xs :: [Int]` と書く。

*30 Haskell では `xs = [1, 2..100]` と書く。

*31 Haskell では次のように書く。

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

*32 Haskell では `zs = xs ++ ys` と書く。

*33 Haskell では `zs = f `map` xs` と書く。

ように書ける.*³⁴

$$z = \bigcup_0^{(\diamond+\diamond)} x_s \quad (43)$$

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

$$\bigcup_a^{\star} x_s = a \star x_0 \star x_1 \dots x_{n-1} \star x_n \quad (44)$$

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.*³⁵

$$\bigsqcup_a^{\star} x_s = a \star (x_0 \dots (x_{n-2} \star (x_{n-1} \star x_n))) \quad (45)$$

0.14 Maybe

計算は失敗する可能性がある. 例えば

$$z = y/x \quad (46)$$

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

*³⁴ Haskell では $z = \text{foldl1 } 0 \ (+) \ xs$ と書く.

*³⁵ Haskell では foldr を用いる.

る．すると関数 f の定義は次のようになる．

$$f y x = \text{if } x \neq 0 \text{ then } y/x \text{ else } \emptyset \dots (\text{不完全}) \quad (47)$$

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

$$\tilde{f} y x = \text{if } x \neq 0 \text{ then } \text{Just } \langle y/x \rangle \text{ else } \emptyset \quad (48)$$

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．^{*36}

$$x_? :: ?\langle\mathbb{Z}\rangle \quad (49)$$

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

$$x_? = \text{Just } \langle 1 \rangle \quad (50)$$

のように書く．^{*37}

Maybe 変数が値を持たない場合は

$$x_? = \emptyset \quad (51)$$

と書く．^{*38}

^{*36} Haskell では `xm :: Maybe Int` と書く．

^{*37} Haskell では `xm = Just 1` と書く．

^{*38} Haskell では `xm = Nothing` と書く．

0.15 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。^{*39}

$$z? = (\diamond + 1) \textcircled{S} x? \quad (52)$$

ここに演算子 \textcircled{S} は

$$\text{Just} \langle fx \rangle = f \textcircled{S} \text{Just} \langle x \rangle \quad (53)$$

$$\emptyset = f \textcircled{S} \emptyset \quad (54)$$

と定義される。

0.16 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。^{*40}

$$f :: \mathbf{a} \rightarrow \mathbf{a} \quad (55)$$

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。^{*41}

$$f :: \mathfrak{Num} \supset \mathbf{a} \Rightarrow \mathbf{a} \rightarrow \mathbf{a} \quad (56)$$

^{*39} Haskell では $zm = (+1) \langle \$ \rangle xm$ と書く。

^{*40} Haskell では $f :: \mathbf{a} \rightarrow \mathbf{a}$ と書く。

^{*41} Haskell では $f :: \mathfrak{Num} \mathbf{a} \Rightarrow \mathbf{a} \rightarrow \mathbf{a}$ と書く。

型クラスは型に制約を与える.

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

TK. 型クラスの例.

0.17 関手

型 \mathbf{a} のリストの変数は

$$x_s :: [\mathbf{a}] \quad (57)$$

という型注釈を持つ. 型 \mathbf{a} 型の Maybe の変数は

$$x_? :: ? \llbracket \mathbf{a} \rrbracket \quad (58)$$

という型注釈を持つ.

普段遣いの関数

$$f :: \mathbf{a} \rightarrow \mathbf{a} \quad (59)$$

をリスト変数 x_s に適用する場合は

$$z_s = f \otimes x_s \quad (60)$$

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

$$z_? = f \circledcirc x_? \quad (61)$$

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 $\mathfrak{F}unctor$ 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．^{*42}

$$x_* :: \mathfrak{F}unctor \multimap \mathbf{f} \langle \mathbf{a} \rangle \quad (62)$$

型クラス $\mathfrak{F}unctor$ に属する型は $\textcircled{\mathbf{S}}$ 演算子を持たねばならない．演算子 $\textcircled{\mathbf{S}}$ は次の形を持つ．^{*43}

$$z_* = f \textcircled{\mathbf{S}} x_* \quad (63)$$

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

$$\diamond \textcircled{\mathbf{S}} \diamond :: (\mathbf{a} \rightarrow \mathbf{b}) \rightarrow \mathbf{f} \langle \mathbf{a} \rangle \rightarrow \mathbf{f} \langle \mathbf{b} \rangle \quad (64)$$

もし変数 x_* の型がリストであれば

$$\textcircled{\mathbf{S}} = \otimes \quad (65)$$

であると解釈する．

`fmap`．

関手は関手則に従う．

$$\text{id} \textcircled{\mathbf{S}} x_* = \text{id} x_* \quad (66)$$

$$(g \bullet f) \textcircled{\mathbf{S}} x_* = ((g \textcircled{\mathbf{S}}) \bullet (f \textcircled{\mathbf{S}})) x_* \quad (67)$$

$$= g \textcircled{\mathbf{S}} (f \textcircled{\mathbf{S}} x_*) \quad (68)$$

^{*42} Haskell では `xm :: Functor f => f a` と書く．

^{*43} In Haskell, `zm = f <$> xm`．

Function of parametric type with functor class:^{*44}

$$f :: \mathfrak{F}\mathbf{unctor} \supset \mathbf{f} \Rightarrow \mathbf{a} \rightarrow \mathbf{f} \llbracket \mathbf{a} \rrbracket \quad (69)$$

Example function application:^{*45}

$$z_{\star} = (\diamond + 1) \textcircled{\mathbb{S}}^{\text{Just}} \langle x \rangle \quad (70)$$

0.18 関手としての関数

$$f :: \mathbf{q} \rightarrow \mathbf{r} \quad (71)$$

Function as a functor:^{*46}

$$f :: (\blacklozenge \rightarrow \mathbf{r}) \mathbf{q} = (\blacklozenge \rightarrow \mathbf{r}) \llbracket \mathbf{q} \rrbracket \quad (72)$$

Thus,

$$f_2 \bullet f_1 \equiv f_2 \textcircled{\mathbb{S}} f_1 \quad (73)$$

$$\text{id} \bullet f = \text{id } f = f \quad (74)$$

$$(h \bullet g) \bullet f = ((h \bullet) \bullet (g \bullet)) f \quad (75)$$

$$= h \bullet (g \bullet f) \quad (76)$$

^{*44} In Haskell, `f :: Functor f => a -> f a`.

^{*45} In Haskell, `zm = (+1) <$> Just x`.

^{*46} In Haskell, `f :: ((->) r) q`.

0.19 アプリカティブ関手

Pure:^{*47}

$$z_{\star} = \star \langle x \rangle \quad (77)$$

Applicative map:^{*48}

$$z_{\star} = f_{\star} \otimes x_{\star} \quad (78)$$

where

$$f_{\star} :: \mathbf{f} \langle \mathbf{a} \rightarrow \mathbf{b} \rangle \quad (79)$$

Applicative style:^{*49}

$$z_{\star} = \star \langle \diamond + \diamond \rangle \otimes x_{\star} \otimes y_{\star} \quad (80)$$

or^{*50}

$$z_{\star} = (\diamond + \diamond) (\textcircled{\text{S}} x_{\star} \otimes y_{\star}) \quad (81)$$

or

$$z_{\star} = \llbracket f x_{\star} y_{\star} \rrbracket \quad (82)$$

^{*47} In Haskell, `zm = pure x`.

^{*48} In Haskell, `zm = f <*> xm`.

^{*49} In Haskell, `zm = pure (+) <*> xm <*> ym`.

^{*50} In Haskell, `zm = (+) <$> xm <*> ym` or `zm = liftA2 (+) xm ym`.

0.20 モナド

Returning *List*.

$$\cdot \quad (83)$$

Returning *Maybe*:^{*51}

$$f :: \mathbb{Z} \rightarrow ? \langle \mathbb{Z} \rangle \quad (84)$$

$$fx = \text{Just } \langle x \rangle \quad (85)$$

Returning *monad*:

$$f :: \mathbb{Z} \rightarrow \mathbf{m} \langle \mathbf{a} \rangle \quad (86)$$

$$fx = \text{return } \langle x \rangle \quad (87)$$

Returning monadic value:^{*52}

$$f :: \mathcal{M}\text{onad} \supset \mathbf{m} \Rightarrow \mathbf{a} \rightarrow \mathbf{m} \langle \mathbf{a} \rangle \quad (88)$$

Monadic function binding:^{*53}

$$z_* = x_* \multimap f_1 \multimap f_2 \quad (89)$$

where

$$f_1 :: \mathbb{Z} \rightarrow ? \langle \mathbb{Z} \rangle \quad (90)$$

$$f_2 :: \mathbb{Z} \rightarrow ? \langle \mathbb{Z} \rangle. \quad (91)$$

^{*51} In Haskell, $f :: \text{Int} \rightarrow \text{Maybe Int}$ and $f\ x = \text{Just } x$.

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

^{*53} In Haskell, $zm = xm \gg= f1 \gg= f2$.

Function binding of monadic function and non-monadic function:^{*54}

$$z_* = x_* \multimap f \multimap g' \quad \text{where } \{g'w = * \langle gw \rangle\} \quad (92)$$

where

$$f :: \mathbb{Z} \rightarrow ? \langle \mathbb{Z} \rangle \quad (93)$$

$$g :: \mathbb{Z} \rightarrow \mathbb{Z}. \quad (94)$$

Another solution is:

$$z_* = (g^* \bullet f) \multimap x_* \quad (95)$$

where g^* means `liftM g` in Haskell.^{*55}

0.21 種

$$\star \rightarrow \star \quad (96)$$

^{*54} In Haskell,

```
zm = xm >>= f >>= g'
```

```
where g' w = pure (g w)
```

^{*55} In Haskell, `zm = (liftM g . f) xm.`

0.22 Data

Data:^{*56}

$$\text{data Suit} = \text{Spade} \vee \text{Heart} \vee \text{Club} \vee \text{Diamond} \quad (97)$$

Data with parameters:^{*57}

$$\text{data V}^2 = V^2 \{x :: \mathbb{Z}, y :: \mathbb{Z}\} \quad (98)$$

0.23 型クラスとインスタンス

0.24 IO

IO example:^{*58}

$$\text{main} = \text{getLine} \multimap \text{print} \gg \text{return } 0 \quad (99)$$

^{*56} In Haskell,

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

^{*57} In Haskell,

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

or `data V2 = V2 Int Int.`

^{*58} In Haskell, `main = getLine >=> print >> return 0.`

0.25 Do 構文

Do notation:^{*59}

$$z_{\star} = \text{do } \{x' \leftarrow x_{\star}; y' \leftarrow y_{\star}; f x'; g y'\} \quad (100)$$

0.26 Notes

Monoid law.

$$\text{id} \bullet f = f \quad (101)$$

$$(h \bullet g) \bullet f = h \bullet (g \bullet f) \quad (102)$$

Functor law.

$$\text{id} \textcircled{\text{S}} x_{\star} = \text{id } x_{\star} \quad (103)$$

$$(g \bullet f) \textcircled{\text{S}} x_{\star} = ((f \textcircled{\text{S}}) \bullet (g \textcircled{\text{S}})) x_{\star} \quad (104)$$

$$= f \textcircled{\text{S}} (g \textcircled{\text{S}} x_{\star}) \quad (105)$$

Applicative law.

$$\text{id} \otimes x_{\star} = x_{\star} \quad (106)$$

$$^{\star} \langle f \rangle \otimes ^{\star} \langle x \rangle = ^{\star} \langle f x \rangle \quad (107)$$

$$f_{\star} \otimes ^{\star} \langle x \rangle = ^{\star} \langle \diamond \$ x \rangle \otimes f_{\star} \quad (108)$$

$$^{\star} \langle \diamond \bullet \diamond \rangle \otimes h_{\star} \otimes g_{\star} \otimes f_{\star} = h_{\star} \otimes (g_{\star} \otimes f_{\star}) \quad (109)$$

$$= h_{\star} \otimes ^{\star} \langle g \bullet f \rangle \quad (110)$$

^{*59} In Haskell, `z = do {x' <- x; y' <- y; f x'; g y'}`.

Monadic law.

$$\tilde{f} \heartsuit^{\text{return}} \langle x \rangle = \tilde{f}x \quad (111)$$

$$\text{return} \langle \diamond \rangle \heartsuit x_{\star} = x_{\star} \quad (112)$$

$$\tilde{g} \heartsuit (\tilde{f} \heartsuit x_{\star}) = (\tilde{g} \heartsuit \tilde{f}) \heartsuit x_{\star} \quad (113)$$