Haskell Notes v.0

Ichi Kanaya

2025

目次

0.1 The New Flow

1. Function. $z = fx$
2. List. $z_{\mathrm{s}} = f \otimes x_{\mathrm{s}} = [fx \mid x \in x_{\mathrm{s}}] = [\![fx_{\mathrm{s}}]\!]$
3. Functor map. $z_\star = f \mathbin{Ⓢ} x_\star = \langle fx \mid x \leftarrow x_\star \rangle = [\![fx_\star]\!]$
4. Monadic function. $z_\star = f^\dagger x$
5. Bind. $z_\star = f^\dagger \heartsuit x_\star = (f^\dagger x \mid x \leftarrow x_\star)$
6. App. $z_\star = f_\star \circledast x_\star = \langle fx \mid f \leftarrow f_\star, x \leftarrow x_\star \rangle = [\![fx_\star \mid f \leftarrow f_\star]\!]$

0.2 Haskell

TK. Haskell について.

本書はプログラミング言語 Haskell の入門書である．それと同時に，本書はプログラミング言語を用いた代数構造の入門書でもある．プログラミングと代数構造の間には密接な関係があるが，とくに**関数型プログラミング**を実践する時にはその関係を意識する必要が出てくる．本書はその両者を同時に解説することを試みる．

これからのプログラマにとって Haskell を無視することはできない．Haskell の「欠点をあげつらうことも，攻撃することもできるが，無視することだけはできない」のだ．それは Haskell がプログラミングの本質に深く関わっているからである．

Haskell というプログラミング言語を知ろうとすると，従来のプログラミング言語の知識が邪魔をする．モダンで，人気があって，Haskell から影響を受けた言語，たとえば Ruby や Swift の知識さえ，Haskell を

学ぶ障害になり得る．ではどのようにして Haskell の深みに到達すればいいのだろうか．

その答えは，一見遠回りに見えるが，一度抽象数学の高みに登ることである．

と言っても，あわてる必要はない．

近代的なプログラミング言語を知っていれば，すでにある程度抽象数学に足を踏み入れているからである．そこで，本書では近代的なプログラマを対象に，プログラミング言語を登山口に抽象数学の山を登り，その高みから Haskell という森を見下ろすことにする．

ところで，プログラムのソースコードは現代でも ASCII 文字セットの範囲で書くことが標準的である．Unicode を利用したり，まして文字にカラーを指定したり，書体や装飾を指定することは一般的ではない．たとえば変数 a のことを $\mathbf{a}$ と書いたり $\underline{\mathrm{a}}$ と書いたり $\hat{a}$ と書いたりして区別することはない．

Haskell プログラマもまた，多くの異なる概念を同じ貧弱な文字セットで表現しなければならない．これは，はじめて Haskell コードを読むときに大きな問題になりえる．たとえば Haskell では [a] という表記をよく扱う．この [a] は a という変数 1 要素からなるリストのこともあるし，a 型という仮の型から作ったリスト型の場合もあるが，字面からでは判断できない．もし変数はイタリック体，型はボールド体と決まっていれば，それぞれ $[a]$ および $[\mathbf{a}]$ と区別できたところである．

本書は，異なる性質のものには異なる書体を割り当てるようにしている．ただし，どの表現もいつでも Haskell に翻訳できるように配慮している．実際，本書執筆の最大の困難点は，数学的に妥当で，かつ Haskell の記法とも矛盾しない記法を見つけることであった．

0.3 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．整数と整数型，実数と浮動小数点型は異なるため，整数型を **Int** で，浮動小数点型を **Double** で表すことにする.[*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 $x :: \mathbf{Int}$ と書く．これは記号 $\in$ を別の用途に用いるためである.[*3]

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.

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．参照透過性を強くサポートするプログラミング言語をを**関数型プログラミング言語**と呼ぶ.

[*1] Haskell では `x = 1` と書く.

[*2] Haskell ではそれぞれ `Int` および `Double` と書く.

[*3] Haskell では `x :: Int` と書く.

0.4 変数の型

TK. 変数の型

$$x :: \mathbf{Int} \tag{2}$$

$$x = 1 \tag{3}$$

$$x :: \mathbf{Int} = 1 \tag{4}$$

a[*4]

$$x :: \mathbf{Int} = 1 \tag{5}$$

a[*5]

$$x :: \mathbf{Double} = 1.0 \tag{6}$$

0.5 関数

整数 x に 1 を足す**関数** f は次のように定義できる.[*6]

$$fx = x + 1 \tag{7}$$

[*4] Haskell では `x :: Int = 1` と書く.
[*5] Haskell では `x :: Double = 1.0` と書く.
[*6] Haskell では `f x = x+1` と書く.

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．[*7]

$$z = fx \tag{8}$$

関数 f が引数をふたつ取る場合は，次のように書く．[*8]

$$z = fxy \tag{9}$$

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．

0.6 関数の型

TK. 関数の型

$$f :: \mathbf{Int} \to \mathbf{Int} \tag{10}$$

[*7] Haskell では `z = f x` と書く．
[*8] Haskell では `z = f x y` と書く．

$$f :: \mathbf{Int} \to \mathbf{Int} \to \mathbf{Int} \tag{11}$$

$$f :: \mathbf{Int} \to \mathbf{Int} \tag{12}$$

$$f :: \mathbf{Int} \to \mathbf{Int} \to \mathbf{Int} \tag{13}$$

0.7 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

$$z = g(fx) \tag{14}$$

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．[*9]

$$k = g \bullet f \tag{15}$$

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．

$$k = h \bullet g \bullet f \tag{16}$$
$$= (h \bullet g) \bullet f \tag{17}$$
$$\tag{18}$$

[*9] Haskell では `k = g . f` と書く．

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．[*10]

$$z = h \,\S\, (g \bullet f)x \tag{19}$$
$$= h\,((g \bullet f)x) \tag{20}$$

0.8 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 2 乗を求める関数 f は次のように定義できる．[*11]

$$f :: \mathbf{Double} \to \mathbf{Double} \tag{21}$$
$$fx = x * x \tag{22}$$

[*10] Haskell では `z = h $ (g . f) x` と書く．

[*11] Haskell では

```
f :: Double -> Double
f x = x*x
```

と書く．

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．[*12]

$$main = print \bullet f \bullet \mathrm{read} \diamond getLine \tag{23}$$

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．

TK. readDouble を導入する．

0.9 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．[*13]

$$f = \backslash x \mapsto x + 1 \tag{24}$$

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．[*14]

$$f = (\Diamond + 1) \tag{25}$$
$$= \backslash x \mapsto x + 1 \tag{26}$$

[*12] Haskell では `main = print . f . read =<< getLine` と書く．

[*13] Haskell では `f = \x -> x+1` と書く．

[*14] 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．

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．[*15]

$$f = \Diamond + \Diamond \tag{27}$$
$$= \backslash x \mapsto \backslash y \mapsto x + y \tag{28}$$

なお $\backslash x \mapsto \backslash y \mapsto x + y$ は $\backslash x\, y \mapsto x + y$ と書いても良い．[*16]

0.10 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．[*17]

$$z = \mathsf{let}\ \{y = 1\}\ \mathsf{in}\ x + y \tag{29}$$

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 (29) は次の式と等価である．

$$z = (\backslash y \mapsto x + y)1 \tag{30}$$

ローカル変数の定義は次のように後置できる．[*18]

$$z = x + y\ \mathsf{where}\ \{y = 1\} \tag{31}$$

[*15] Haskell では `f = (+)` と書く．

[*16] Haskell では `\x y -> x+y` と書く．

[*17] 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．

[*18] 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．

0.11 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

$$fn = \backslash x \mapsto n + x \tag{32}$$

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．

$$n = 3 \tag{33}$$
$$g = fn \tag{34}$$

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．

0.12 型

TK. 一部前倒し

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 **Int** で，浮動小数点型を **Double** 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，本書では主に固定長整数型を用いる．

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．単精度浮動小数点型はめったに用いられないため，本書では浮動小数点型と言えば倍精度浮動小数点型を意味することにする．

論理型は論理値 True または False のいずれかしか値をとれない型である．論理型のことを **Bool** と書く．[*19]

文字型を **Char** と書く．[*20]

変数 x の型が **Int** のとき，以下のように**型注釈**を書く．[*21]

$$x :: \mathbf{Int} \tag{35}$$

TK. 一部前倒し

1 引数関数の型は次のように注釈できる．[*22]

$$f :: \mathbf{Int} \to \mathbf{Int} \tag{36}$$

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．[*23]

2 引数関数の方は次のように注釈できる．[*24]

$$f :: \mathbf{Int} \to \mathbf{Int} \to \mathbf{Int} \tag{37}$$

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 $\mathbf{Int} \to \mathbf{Int} \to \mathbf{Int}$ は $\mathbf{Int} \to (\mathbf{Int} \to \mathbf{Int})$ と解釈される．

[*19] Haskell ではブール型を `Bool` と書く．
[*20] Haskell では Unicode 文字型を `Char` と書く．
[*21] Haskell では `x :: Int` と書く．
[*22] Haskell では `f :: Int -> Int` と書く．
[*23] 正確には → は型コンストラクタである．
[*24] Haskell では `f :: Int -> Int -> Int` と書く．

$(\mathbf{Int} \to \mathbf{Int})$ 型の関数を受け取り，$(\mathbf{Int} \to \mathbf{Int})$ 型の関数を返す関数は次の型を持つ. [*25]

$$f :: (\mathbf{Int} \to \mathbf{Int}) \to (\mathbf{Int} \to \mathbf{Int}) \tag{38}$$

なお後半の括弧は省略可能なので

$$f :: (\mathbf{Int} \to \mathbf{Int}) \to \mathbf{Int} \to \mathbf{Int} \tag{39}$$

と書いても良い.

Haskell ではすべての変数，関数に型があり，型はコンパイル時に決定されていなければならない．ただし，式から**型推論**が行える場合は型注釈を省略できる.

0.13 条件

TK. 旧文章を確認する.

条件分岐は次のように書く. [*26]

$$z = \mathsf{if}\, x > 0 \,\mathsf{then}\, x \,\mathsf{else}\, -x \tag{40}$$

TK. パターンマッチを独立させる

[*25] Haskell では以下のように書く.

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

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

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.[*27]

$$f = \mathsf{case}\, x\, \mathsf{of} \begin{cases} 1 \text{ -> } 1 \\ \_ \text{ -> } 0 \end{cases} \tag{41}$$

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．

TK. Bool の説明.

関数定義にもパターンマッチを使える.[*28]

$$\begin{cases} f1 = 1 \\ f\_ = 0 \end{cases} \tag{42}$$

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.[*29]

$$\begin{cases} fx \mid_{x>0} = x \\ \quad \mid_{\mathrm{otherwise}} = -x \end{cases} \tag{43}$$

ここに otherwise は True の別名である.

[*27] 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
```

[*28] Haskell では次のように書く.

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

[*29] Haskell では次のように書く.

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

TK. ガードは上から順にマッチされる.

0.14 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.[*30]

$$\begin{cases} \mathrm{fib}\,0 = 0 \\ \mathrm{fib}\,1 = 1 \\ \mathrm{fib}\,n = \mathrm{fib}(n-1) + \mathrm{fib}(n-2) \end{cases} \tag{44}$$

TK. プログラムの本質

0.15 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.[*31]

$$z = (x, y) \tag{45}$$

[*30] 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)
```

[*31] 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 **Int** が 2 個からなるタプルの型は次のようになる．[*32]

$$z :: (\mathbf{Int}, \mathbf{Int}) \tag{46}$$

TK. タプルの受け取り

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．[*33]

$$z = () \tag{47}$$

ユニットの型は**ユニット型**で，型注釈を次のように書く．[*34]

$$z :: () \tag{48}$$

0.16　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．[*35]

$$x_{\mathrm{s}} = [\,] \tag{49}$$

[*32] In Haskell, `z :: (Int, Int)`.
[*33] Haskell では `z = ()` と書く．
[*34] Haskell では `z :: ()` と書く．
[*35] Haskell では `xs = []` と書く．

任意のリストは次のように**リスト構築演算子** : を用いて構成する．

$$x_{\mathrm{s}} = x_0 : x_1 : x_2 : \cdots : [\,] \tag{50}$$

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．[*36]

$$x_{\mathrm{s}} :: [\mathbf{Int}] \tag{51}$$

リストは次のように構成することもできる．[*37]

$$x_{\mathrm{s}} = [1, 2, \ldots, 100] \tag{52}$$

なお次のような**無限リスト**を構成しても良い．[*38]

$$x_{\mathrm{s}} = [1, 2, \ldots] \tag{53}$$

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．[*39]

$$z_{\mathrm{s}} = x_{\mathrm{s}} \# y_{\mathrm{s}} \tag{54}$$

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．

$$f :: [\mathbf{Int}] \to \mathbf{Int} \tag{55}$$

$$f(x : x_{\mathrm{s}}) = x \ldots (\text{不完全}) \tag{56}$$

ただし，引数のリストが空リストである可能性を考慮して，式 (56) は次のように書き直すべきである．

$$\begin{cases} f[\,] = 0 \\ f(x : x_{\mathrm{s}}) = x \end{cases} \tag{57}$$

[*36] Haskell では `xs :: [Int]` と書く．
[*37] Haskell では `xs = [1, 2..100]` と書く．
[*38] Haskell では `xs = [1, 2..]` と書く．
[*39] Haskell では `zs = xs ++ ys` と書く．

$f[\,]$ が 0 を返すのは不自然だが，関数 f の戻り型を整数型としているためこれは仕方がない．エラーを考慮する場合は 0.21 節で述べる Maybe を使う必要がある．

TK. head は非推奨

TK. リスト先頭 2 要素の和

リストのリストは次のように構成できる．

$$z_{\mathrm{ss}} = [[1,2],[3,4]] \tag{58}$$

TK. 線形リストであることと vector の話し．

0.17　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．[*40]

$$x_{\mathrm{s}} = \left[x^2 \mid x \in [1,2\ldots100], \mathrm{even}\, x\right] \tag{59}$$

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．

[*40] Haskell では次のように書く．

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

0.18 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．

$$\text{type } \mathbf{String} = [\mathbf{Char}] \tag{60}$$

文字列型のリテラルは次のように書く．[*41]

$$x_{\mathrm{S}} = \texttt{"Hello, World!"} \tag{61}$$

TK. String に関する有名な関数.

リストに対するすべての演算は文字列にも適用可能である．

0.19 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．[*42]

$$z_{\mathrm{s}} = f \otimes x_{\mathrm{s}} \tag{62}$$

式 (62) は次の式と同じである．[*43]

$$z_{\mathrm{s}} = [f x \mid x \in x_{\mathrm{s}}] \tag{63}$$

[*41] Haskell では `xs = "Hello, World!"` と書く.
[*42] Haskell では `zs = f ‘map‘ xs` と書く.
[*43] Haskell では `zs = [f x | x <- xs]` と書く.

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．[*44]

$$z = \bigcup_{0}^{(\Diamond+\Diamond)} x_{\mathrm{s}} \tag{64}$$

Haskell では

$$\sum = \bigcup_{0}^{(\Diamond+\Diamond)} \tag{65}$$

として関数 $\sum$ が定義されている．[*45]

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

$$\bigcup_{a}^{\Diamond\maltese\Diamond} x_{\mathrm{s}} = a \maltese x_0 \maltese x_1 \ldots x_{n-1} \maltese x_n \tag{66}$$

である．ここに ✠ は任意の二項演算子である．

畳み込み演算子には次の右結合バージョンが存在する．[*46]

$$\bigsqcup_{a}^{\Diamond\maltese\Diamond} x_{\mathrm{s}} = a \maltese (x_0 \ldots (x_{n-2} \maltese (x_{n-1} \maltese x_n))) \tag{67}$$

0.20 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．

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

[*45] Haskell では関数 $\sum$ を `sum` と書く．

[*46] Haskell では `foldr` を用いる．

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．[*47]

$$\mathrm{norm} :: [\mathbf{Double}] \to \mathbf{Double} \tag{68}$$

$$\begin{cases} \mathrm{norm}[\,] = 0.0 \\ \mathrm{norm}\, x_{\mathrm{s}} = \mathrm{sqrt}\left(\sum [x * x \mid x \in x_{\mathrm{s}}]\right) \end{cases} \tag{69}$$

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 readDouble :: **String** → **Double** を用いる．[*48]

$$\mathrm{readDouble} :: \mathbf{String} \to \mathbf{Double} \tag{70}$$

$$\mathrm{readDouble} = \mathrm{read} \tag{71}$$

関数 readDouble は標準関数 read に型注釈を付けたものである．

[*47] Haskell では次のように書く．

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

[*48] Haskell では次のように書く．

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

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．[*49]

$$main = print \bullet (\mathrm{norm} \otimes) \bullet ((\mathrm{readDouble} \otimes) \otimes) \bullet (\mathrm{words} \otimes) \bullet \mathrm{lines} \heartsuit getContents \quad (72)$$

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．[*50]

$$printEach\, x_{\mathrm{s}} = print \clubsuit x_{\mathrm{s}} \quad (73)$$

演算子 ♣ はアクション版のマップ演算子である．

0.21 Maybe

計算は失敗する可能性がある．たとえば

$$z = y/x \quad (74)$$

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

[*49] Haskell では次のように書く．

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

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

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．

$$f\,x\,y = \mathsf{if}\, x \neq 0 \,\mathsf{then}\, y/x \,\mathsf{else}\, \varnothing \ldots (\text{不完全}) \tag{75}$$

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

$$f^{\dagger}\,x\,y = \mathsf{if}\, x \neq 0 \,\mathsf{then}\, {}^{\mathrm{Just}}\langle y/x \rangle \,\mathsf{else}\, \varnothing \tag{76}$$

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．[*51]

整数型 **Int** を Maybe で包む場合は ${}^{\mathrm{Maybe}}\langle\!\langle \mathbf{Int} \rangle\!\rangle$ 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．[*52]

$$x_? :: {}^{\mathrm{Maybe}}\langle\!\langle \mathbf{Int} \rangle\!\rangle \tag{77}$$

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

$$x_? = {}^{\mathrm{Just}}\langle 1 \rangle \tag{78}$$

のように書く．[*53]

[*51] Haskell では `f x y = if x /= 0 then Just y/x else Nothing` と書く．

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

[*53] Haskell では `xm = Just 1` と書く．

Maybe 変数が値を持たない場合は

$$x_? = \varnothing \tag{79}$$

と書く.[*54]

一度 Maybe になった変数を非 Maybe に戻すことは出来ない.

0.22 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.[*55]

$$z_? = f \,Ⓢ\, x_? \tag{80}$$

ここに関数 f は 1 引数関数で，演算子 Ⓢ は

$$^{\mathrm{Just}}\langle f x\rangle = f \,Ⓢ\, {}^{\mathrm{Just}}\langle x\rangle \tag{81}$$

$$\varnothing = f \,Ⓢ\, \varnothing \tag{82}$$

と定義される.

0.23 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.

$$x_? = {}^{\mathrm{Just}}\langle [1, 2, \ldots, 100]\rangle \tag{83}$$

[*54] Haskell では `xm = Nothing` と書く.
[*55] Haskell では `zm = (+1) <$> xm` と書く.

のとき，リストの各要素に関数 $f :: \mathbf{Int} \to \mathbf{Int}$ を適用するには次のように書く．[*56]

$$z_? = (f\otimes)\ \textcircled{S}\ x_? \tag{84}$$

0.24 型パラメタと型クラス

型をパラメタとして扱うことができる．任意の型を **a** と，ボールド体小文字で書く．ある型 **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．[*57]

$$f :: \mathbf{a} \to \mathbf{a} \tag{85}$$

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 **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．[*58]

$$f :: \mathfrak{Num} \supset \mathbf{a} \Rightarrow \mathbf{a} \to \mathbf{a} \tag{86}$$

ここに $\mathfrak{Num}$ 型クラスには，整数型 **Int**，浮動小数点型 **Double** が含まれる一方，論理型 **Bool** は含まれない．

TK. 型の条件

[*56] 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．

[*57] Haskell では `f :: a -> a` と書く．

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

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.

0.25 関手

型 **a** のリストの変数は

$$x_{\mathrm{s}} :: [\mathbf{a}] \tag{87}$$

という型注釈を持つ. これは

$$x_{\mathrm{s}} :: {}^{[]}\langle\!\langle \mathbf{a} \rangle\!\rangle \tag{88}$$

のシンタックスシュガーである.[*59]

型 **a** の Maybe の変数は

$$x_{?} :: {}^{\mathrm{Maybe}}\langle\!\langle \mathbf{a} \rangle\!\rangle \tag{89}$$

という型注釈を持つ.

普段遣いの関数

$$f :: \mathbf{a} \to \mathbf{a} \tag{90}$$

をリスト変数 x_{s} に適用する場合は

$$z_{\mathrm{s}} = f \otimes x_{\mathrm{s}} \tag{91}$$

[*59] Haskell では `xs :: [] a` と書く.

とする．同じく関数 f を Maybe 変数 $x_?$ に適用する場合は

$$z_? = f \textcircled{S} x_? \tag{92}$$

とする．

リストも Maybe も元の型 **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 **a** 型の変数を次のように型注釈する．[*60]

$$x_\star :: \mathfrak{Functor} \supset \mathfrak{f} \Rightarrow {}^{\mathfrak{f}}\langle\!\langle \mathbf{a} \rangle\!\rangle \tag{93}$$

型クラス $\mathfrak{Functor}$ に属する型は Ⓢ 演算子を必ず持つ．演算子 Ⓢ は次の形を持つ．[*61]

$$z_\star = f \textcircled{S} x_\star \tag{94}$$

演算子 Ⓢ の型は次のとおりである．

$$\Diamond \textcircled{S} \Diamond :: \mathfrak{Functor} \supset \mathfrak{f} \Rightarrow (\mathbf{a} \to \mathbf{b}) \to {}^{\mathfrak{f}}\langle\!\langle \mathbf{a} \rangle\!\rangle \to {}^{\mathfrak{f}}\langle\!\langle \mathbf{b} \rangle\!\rangle \tag{95}$$

もし変数 $x_\star$ の型がリストであれば

$$\textcircled{S} = \otimes \tag{96}$$

であると解釈する．

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

[*61] Haskell では `zm = f <$> xm` と書く．

0.26 関手としての関数

TK. 移動する.

$$f :: \mathbf{q} \to \mathbf{r} \tag{97}$$

Function as a functor:[*62]

$$f :: (\blacklozenge \to \mathbf{r})\,\mathbf{q} = {}^{(\blacklozenge \to \mathbf{r})}\langle\!\langle \mathbf{q} \rangle\!\rangle \tag{98}$$

Thus,

$$f_2 \bullet f_1 \equiv f_2 \,\textcircled{\scriptsize S}\, f_1 \tag{99}$$

$$\begin{aligned} \mathrm{id} \bullet f = \mathrm{id}\, f &= f && (100)\\ (h \bullet g) \bullet f = ((h\bullet) \bullet (g\bullet))f &&& (101)\\ = h \bullet (g \bullet f) &&& (102) \end{aligned}$$

0.27 アプリカティブ関手

演算子 Ⓢ は関手型クラスの型の値に 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

[*62] In Haskell, `f :: ((->) r) q`.

数 f が 2 引数をとるとし，関手型クラスの型の変数 $x_\star$ 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

$$f' = f \,\textcircled{\scriptsize S}\, x_\star \tag{103}$$

を作ると，定義によって関数 f' は関手型クラスの型の変数になる．そこで，関手型クラスの型の関数を関手型クラスの型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

$$z_\star = f' \circledast y_\star \tag{104}$$
$$= f \,\textcircled{\scriptsize S}\, x_\star \circledast y_\star \tag{105}$$

と書ける．

任意の変数または関数を関手型クラスの型に入れる**ピュア演算子**があり，次のように書く．[*63]

$$z_\star = {}^\star\langle x \rangle \tag{106}$$

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 (105) は

$$z_\star = {}^\star\langle f \rangle \circledast x_\star \circledast y_\star \tag{107}$$

と書ける．[*64]

式 (107) はかつて

$$z_\star = [\![f\, x_\star\, y_\star]\!] \tag{108}$$

[*63] Haskell では `zm = pure x` と書く．
[*64] Haskell では `zm = (pure f) <*> xm <*> ym` と書く．

のように書くことも提案されたが，普及はしなかった．[*65]

ピュア演算子とアプリカティブマップ演算子を必ず持つ関手のことを**アプリカティブ関手**と呼び $\mathfrak{Applicative}$ で表す．

いま関数 $f :: \mathbf{a} \to \mathbf{b}$ に対して，新たな関数 $f_\star$ ただし

$$f_\star = {}^\star \langle f \rangle \tag{109}$$

を作ったとすると，関数 $f_\star$ は

$$f_\star :: \mathfrak{Applicative} \supset \mathfrak{f} \Rightarrow {}^{\mathfrak{f}} \langle\!\langle \mathbf{a} \to \mathbf{b} \rangle\!\rangle \tag{110}$$

という型を持つ．アプリカティブマップ演算子は変数

$$x_\star :: \mathfrak{Applicative} \supset \mathfrak{f} \Rightarrow {}^{\mathfrak{f}} \langle\!\langle \mathbf{a} \rangle\!\rangle \tag{111}$$

に対して，関数 $f_\star$ を

$$z_\star = f_\star \circledast x_\star \tag{112}$$

のように作用させる．変数 $z_\star$ の型は

$$z_\star :: \mathfrak{Applicative} \supset \mathfrak{f} \Rightarrow {}^{\mathfrak{f}} \langle\!\langle \mathbf{b} \rangle\!\rangle \tag{113}$$

である．

0.28　モナド

Returning *List*.

$$. \tag{114}$$

[*65] 現在の Haskell では `zm = liftA2 f xm ym` と書くことで代用されている．元の提案は `zm = [|f xm ym|]` であった．

Returning *Maybe*:[*66]

$$f :: \mathbf{Int} \to {}^{\mathrm{Maybe}} \langle\!\langle \mathbf{Int} \rangle\!\rangle \tag{115}$$

$$f\,x = {}^{\mathrm{Just}} \langle x \rangle \tag{116}$$

Returning *monad*:

$$f :: \mathbf{Int} \to {}^{\mathfrak{m}} \langle\!\langle \mathbf{a} \rangle\!\rangle \tag{117}$$

$$f\,x = {}^{\star} \langle x \rangle \tag{118}$$

Returning monadic value:[*67]

$$f :: \mathfrak{Monad} \supset \mathfrak{m} \Rightarrow \mathbf{a} \to {}^{\mathfrak{m}} \langle\!\langle \mathbf{a} \rangle\!\rangle \tag{119}$$

Monadic function binding:[*68]

$$z_\star = x_\star \mathbin{-\!\!\heartsuit\!\!\to} f_1 \mathbin{-\!\!\heartsuit\!\!\to} f_2 \tag{120}$$

where

$$f_1 :: \mathbf{Int} \to {}^{\mathrm{Maybe}} \langle\!\langle \mathbf{Int} \rangle\!\rangle \tag{121}$$

$$f_2 :: \mathbf{Int} \to {}^{\mathrm{Maybe}} \langle\!\langle \mathbf{Int} \rangle\!\rangle\,. \tag{122}$$

Function binding of monadic function and non-monadic function:[*69]

$$z_\star = x_\star \mathbin{-\!\!\heartsuit\!\!\to} f \mathbin{-\!\!\heartsuit\!\!\to} g' \text{ where } \{g'w = {}^{\star} \langle gw \rangle\} \tag{123}$$

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

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

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

[*69] In Haskell,

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

or

$$z_\star = x_\star \multimap (f \rightarrowtail g') \text{ where } \{g'w = {}^\star \langle gw \rangle\} \tag{124}$$

where

$$f :: \mathbf{Int} \to {}^{\mathrm{Maybe}} \langle\!\langle \mathbf{Int} \rangle\!\rangle \tag{125}$$

$$g :: \mathbf{Int} \to \mathbf{Int}. \tag{126}$$

Another solution is:

$$z_\star = (g^* \bullet f) \diamond x_\star \tag{127}$$

where g^* means `liftM g` in Haskell.[*70]

0.29 種

$$\star \to \star \tag{128}$$

0.30 Data

Data:[*71]

$$\mathsf{data}\, \mathbf{Suit} = \mathrm{Spade} \vee \mathrm{Heart} \vee \mathrm{Club} \vee \mathrm{Diamond} \tag{129}$$

Data with parameters:[*72]

$$\mathsf{data}\, \mathbf{V^2} = \mathrm{V}^2\, \{x :: \mathbf{Int}, y :: \mathbf{Int}\} \tag{130}$$

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

[*71] In Haskell,

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

[*72] In Haskell,

0.31 型クラスとインスタンス

0.32 IO モナド

IO example:[*73]

$$main = getLine \multimap print \gg^{\star} \langle 0 \rangle \tag{131}$$

0.33 Do 構文

Do notation:[*74]

$$z_{\star} = \mathsf{do}\,\{x' \leftarrow x_{\star}; y' \leftarrow y_{\star}; fx'; gy'\} \tag{132}$$

0.34 モノイド

任意の関数 f に対して

$$\mathrm{id}\, f = f \tag{133}$$

なる関数 id があり，かつ任意の関数 f, g, h に対して

$$(h \bullet g) \bullet f = h \bullet (g \bullet f) \tag{134}$$

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

or `data V2 = V2 Int Int`.

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

[*74] 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．

0.35　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，

$$i \bigcirc x = x \ldots (\text{単位元の存在}) \tag{135}$$
$$(x \bigcirc y) \bigcirc z = x \bigcirc (y \bigcirc z) \ldots (\text{結合律}) \tag{136}$$

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 $(\mathbf{Int}, +, 0)$ や $(\mathbf{Int}, \times, 1)$ 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．

$$\mathrm{id} \bullet f = f \ldots (\text{単位元の存在}) \tag{137}$$
$$(h \bullet g) \bullet f = h \bullet (g \bullet f) \ldots (\text{結合律}) \tag{138}$$

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．

0.36　関手則

関手のマップ演算子 Ⓢ は以下の**関手則**に従う．

$$\mathrm{id} \,Ⓢ\, x_\star = \mathrm{id}\, x_\star \tag{139}$$
$$(g \bullet f) \,Ⓢ\, x_\star = ((g Ⓢ) \bullet (f Ⓢ)) x_\star \tag{140}$$
$$= g \,Ⓢ\, (f \,Ⓢ\, x_\star) \tag{141}$$

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

$$\underset{\boldsymbol{X}}{\mathrm{id}}, \underset{\mathfrak{F}\boldsymbol{X}}{\mathrm{id}} \to \mathrm{id} \tag{142}$$

$$f\text{Ⓢ} \to \mathfrak{F}f \tag{143}$$

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．

0.37 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．

$$^{\star}\langle \mathrm{id} \rangle \circledast x_{\star} = x_{\star} \tag{144}$$

$$^{\star}\langle f \rangle \circledast {}^{\star}\langle x \rangle = {}^{\star}\langle fx \rangle \tag{145}$$

$$f_{\star} \circledast {}^{\star}\langle x \rangle = {}^{\star}\langle \Diamond \S\, x \rangle \circledast f_{\star} \tag{146}$$

$$^{\star}\langle \Diamond \bullet \Diamond \rangle \circledast h_{\star} \circledast g_{\star} \circledast f_{\star} = h_{\star} \circledast (g_{\star} \circledast f_{\star}) \tag{147}$$

0.38　モナド則

モナドのマップ演算子 $\heartsuit$ は以下の規則に従う．

$$f^{\dagger} \heartsuit {}^{\star} \langle x \rangle = f^{\dagger} x \tag{148}$$

$${}^{\star} \langle \diamondsuit \rangle \heartsuit x_{\star} = x_{\star} \tag{149}$$

$$(g^{\dagger} \heartsuit f^{\dagger}) \heartsuit x_{\star} = g^{\dagger} \heartsuit (f^{\dagger} \heartsuit x_{\star}) \tag{150}$$

次の**クライスリスター**すなわち

$$f^{\bigstar} = (f^{\dagger} \heartsuit \diamondsuit) \tag{151}$$

を用いると，モナド則は次のように書き換えられる．

$$\left(f^{\bigstar}\right)^{\star} \langle x \rangle = f^{\dagger} x \tag{152}$$

$$({}^{\star} \langle \diamondsuit \rangle)^{\bigstar} x_{\star} = x_{\star} \tag{153}$$

$$\left(g^{\bigstar} f^{\dagger}\right)^{\bigstar} x_{\star} = g^{\bigstar} \left(f^{\bigstar} x_{\star}\right) \tag{154}$$

0.39　クラスの定義

アプリカティブ関手は関手の拡張である．

$$\mathsf{class}\ \mathfrak{Functor} \supset \mathfrak{f} \Rightarrow \mathfrak{Applicative} \supset \mathfrak{f}\ \mathsf{where} \tag{155}$$

$$\begin{aligned} &{}^{\star} \langle \diamondsuit \rangle :: \mathbf{a} \to {}^{\mathfrak{f}} \langle\!\langle \mathbf{a} \rangle\!\rangle \\ &\diamondsuit \circledast \diamondsuit :: {}^{\mathfrak{f}} \langle\!\langle \mathbf{a} \to \mathbf{b} \rangle\!\rangle \to {}^{\mathfrak{f}} \langle\!\langle \mathbf{a} \rangle\!\rangle \to {}^{\mathfrak{f}} \langle\!\langle \mathbf{b} \rangle\!\rangle \end{aligned} \tag{156}$$