関数型プログラミングと代数構造

金谷一朗

2017 年 8 月 27 日

目次

表 1　凡例

種類	字体・表記法	例
定数	イタリック，大文字	A, B, C
有名な定数	ローマン，大文字	True, False
変数	イタリック，小文字	u, v, w, x, y, z
有名な変数	ローマン，小文字	first, rest
リスト変数	肩に星印をつける	$x^\star$
Maybe 変数	肩に?をつける	$u^?$
Ether 変数	肩に!をつける	$u^!$
アクション	ギリシア文字	α, ω
有名なアクション	スラント・小文字	*putstr*, *readln*
関数	イタリック，小文字	f, g, h, i, j, k
有名な関数	ローマン，小文字	sin, cos
コンテナ関数	ギリシア文字	ϕ, ψ
添字	ローマン，大文字	$x_{\mathrm{Left}}, x_{\mathrm{Right}}$
集合	ボールドローマン，大文字	$\mathbf{A}, \mathbf{B}, \mathbf{C}$
有名な集合	ブラックボード，大文字	$\mathbb{B}, \mathbb{Z}, \mathbb{R}$
単位元	ローマンに斜線，大文字	$\emptyset$
型パラメタ	ボールドローマン，小文字	$\mathbf{a}, \mathbf{b}, \mathbf{c}$
型	ボールドローマン，大文字	**Bool**, **Int**, **Float**
コンテナ型	左肩にコンテナ名をつける	$^{\mathbf{Maybe}}[\![\mathbf{a}]\!]$, $^{\mathbf{IO}}[\![\mathbf{a}]\!]$
有名なコンテナ型	特別な括弧で包む	$[\mathbf{a}], \langle\!\langle \mathbf{a} \rangle\!\rangle_{\mathbf{r}}$
型クラス	スモールキャピタル	Eq, Ord, Num
値コンストラクタ	肩書付き括弧で包む	$^{\mathrm{Just}}[\![x]\!]$, $^{\mathrm{Right}}[\![x]\!]$
有名な値コンストラクタ	特別な括弧で包む	$[x], \langle\!\langle x \rangle\!\rangle$
型コンストラクタ	イタリック，大文字	*List*, *Maybe*
関手	ボールドイタリック，大文字	***List***, ***Maybe***
有名な関手	カリグラフィ，大文字	$\mathcal{I}$
キーワード	サンセリフ，小文字	if, otherwise
do 記法中のキーワード	タイプライタ	`let`
リスト	ブラケットで包む	$[x, y, z]$
集合	ブレースで包む	$\{x, y, z\}$
タプル	丸括弧で包む	(x, y, z)

表 2　記号一覧

記号	意味	最初に登場するページ
$\backslash, \rightarrow$	ラムダ式	
$\diamond$	無名パラメタ	
::	集合の元	
$\in$	集合の元	
$\mapsto$	写像	
$\bigstar$	任意の二項演算子	
$\dot{+}$	一般加法演算子	
$\centerdot$	関数合成	
\$	関数適用	
$\rightsquigarrow$	関数の右適用	
$\bigcup$	左畳み込み	
$\bigsqcup$	右畳み込み	
:	結合 (cons)	
$\oplus$	リスト結合 (append)	
$\circledast$	ジップ	
$\flat$	平坦化 (concat)	
$\sharp$	ジョイン	
[]	空リスト	
$\emptyset$	空 Maybe（ナッシング）	
$\varnothing$	空	
$\bullet$	一般マップ	
$\odot$	リストのマップ	
$\boxdot$	Maybe のマップ	
$\circ$	関数のマップ	
$\times$	一般アプリカティブマップ	
$\otimes$	リストのアプリカティブマップ	
$\boxtimes$	Maybe のアプリカティブマップ	
$\bowtie$	関数のアプリカティブマップ	
$\clubsuit$	リストのバインド演算子	
$\spadesuit$	Maybe のバインド演算子	
$\heartsuit$	モナドのバインド演算子	
$\diamondsuit$	関数のバインド演算子	

第 I 部

代数的構造とプログラミング

第 1 章

はじめに*

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，特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．

1.1 Haskell という森

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，例えば 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．

さて，登山口にどのプログラミング言語を選ぶのが適当であろうか．IEEE Spectrum の “The 2015 Top Ten Programming Languages” という記事によると「ビッグ 5」として Java, C, C++, Python, C# が挙げられている．このうち C は「多くのプログラマが読める」以外にメリットが無く，その唯一のメリットさえ最近は怪しくなっているため，登山口候補から外す．残るは Java, C++, C# グループと Python ということになるが，シンプルであり，かつ Haskell と対極にある言語である Python（バージョン 3）を登山口に選ぶことにした．

本書では Python コードはこのように登場する．

Python
```
print("Hello, world.")
```

本書に示すコードは擬似コードではなく，すべて実行可能な本物のコードである．

ところで，一部の章でどうしても型に触れないといけない部分がある．Python は動的型付け言語であり，型の説明には不適切であるため，この部分だけ理解の助けとして C++14 によるコードを例示した．この部分はコードを読まなくても先に進める．Java ではなく C++ を採用したのは macOS でも簡単に試せるようにである．型システムに関する限り C++ よりも Java のほうが簡潔に記述できるが，致し方無い．

✠

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．例えば変数 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，初めて Haskell コードを読むときに大きな問題になり得

る．例えば 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 $[\mathbf{a}]$ と区別出来たところである．

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．

1.2　関数型プログラミング

プログラマはなぜ Haskell を習得しなければならないのだろう．それは Haskell と **関数型プログラミング** の間に密接な関係があるからである．

関数型プログラミングとはプログラミングにおける一種のスローガンのようなもので，どの言語を用いたから関数型でどの言語を用いたから関数型ではない，というものではない．しかし，関数型プログラミングを強くサポートする言語と，そうでない言語とがある．ここら辺の事情はオブジェクト指向プログラミングとプログラミング言語の関係と似ている．Haskell は関数型プログラミングを強くサポートし，Python はほとんどサポートしない．

関数型プログラミングの特徴を一言で言えば，プログラム中の **破壊的代入** を禁止することである．変数 x に 1 という数値が一度代入されたら，変数 x の値をプログラム中に書き換える，すなわち破壊的代入をすることはできない．この結果，変数の値はプログラムのどこでも，どの時点で読み出しても同じであることが保証される．これを変数の **参照透過性** と呼ぶ．

プログラム全体に参照透過性があると，そのプログラムはブロックに分割しやすく，各々のブロックは再利用しやすい．またプログラムのどの断片から読み始めても，全体の構造を見失いにくい．これが関数型プログラミングとそれを強くサポートする Haskell を習得する理由である．

参照透過性がもたらすもう一つのボーナスは変数の **遅延評価** である．変数はいつ評価しても値が変わらないのだから，コンパイラは変数をできるだけ遅く評価してよい．この遅延評価によって，Haskell コンパイラは他の言語に見られない **無限リスト** を扱う能力を獲得している．

✠

ここで数学とプログラミングの関係について述べておこう．ある方程式を解くためにコンピュータによって数値シミュレーションを行うとか，非常に複雑な微分を機械的に行うとか，プログラミングによって数学をサポートすることは計算機科学の主たる分野の一つであるが，ここではもっと根源的な話をする．

数学者もプログラマも **関数** をよく使う．数学者が使う関数とは，引数がいくつかあって，その結果決まる戻り値があるようなものだ．一方でプログラマが使う関数というのは，引数と戻り値はだいたい同じとして，中身に条件分岐があったり，ループがあったり，外部変数を書き換えたり，入出力をしたりする．

どちらも同じ関数であるのに，なぜこうもイメージが違うのだろうか．

もし，我々が関数型プログラミングの原則を忠実に守り，プログラム中のいかなる破壊的代入をも禁止するとすると，両者の関数は全く同じ性格になる．逐次実行も条件分岐もループも，それどころか定数さえ，**ラムダ式** という式だけで書けるようになる．あらゆるプログラムが，最終的には単一のラムダ式で書ける．

ところが，入出力，状態変数，例外など，プログラミングに使われる多くのテクニックは関数の副作用を前提としている．参照透過性と副作用を統一的に扱うためには **モナド** という数学概念が必要である．Haskell はモナドを陽に扱うプログラミング言語である．

...

1.3　Haskell コンパイラの準備*

本書の第 I 部は Haskell コンパイラ無しで読み進めることが出来る．とは言え，前もって Haskell コンパイラを用意しておくことは無意味とも言えない．

—

Haskell コンパイラの中で最もよく使われているものは Glasgow's Haskell Compiler (GHC) である．本書に登場する Haskell コードもすべて GHC でテストをしている．GHC は 2015 年にリリースされたバージョン 7.10 とそれまでのバージョンとの間に大きな違いがある．本書はバージョン 7.10 以降を対象としているため，一部は古い書籍と互換性がない．

GHC は公式サイト [1] からダウンロード可能であるが，GNU/Linux システム（以下 Linux），macOS はそれぞれのパッケージマネージャからインストールすることを勧める．インストール方法は以下の参考文献にアクセスしてもらいたい．

Linux　パッケージマネージャから ghci をインストールする．Debian 系なら APT，Red Hat 系なら Yum を使うのが一般的であろう．文献 [2] を参考にしてもらいたい．

macOS　文献 [3] に Homebrew のインストール方法が書かれているので，Homebrew のインストールまでしておき，ターミナルで

```
$ brew install ghci
```

とする．（本書執筆時点で Haskell Stack という新しいパッケージングシステムが登場してきている．将来的には GHC 本体も Haskell Stack からインストールすることが望ましくなるだろう．Haskell Stack を用いた GHC のインストールは macOS では Homebrew を用いて

```
$ brew install haskell-stack
$ stack install ghc
```

とすれば良い．本書はひとまず GHC を直接インストールすることを前提に進める．）

Windows 10　GHC 公式サイト [1] からパッケージをインストールする．

1.4　余計な話：本書の構成*

ところで，各章の終りにはこのような「余計な話」の節を設けている．余計な話には，本書を読み進めるに当たって本質的ではない話を詰め込んでいる．ではなぜ書くかというと，これは筆者が頭を整理するために書くのである．なので，筆者が何を思って本書を執筆していたかを知りたいときが万が一来れば，目を通してもらいたい．

最初の余計な話は本書の構成である．

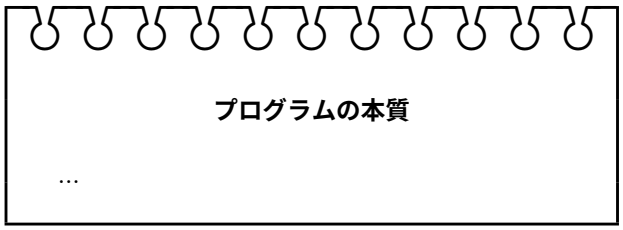
そもそも本の構成など，目次を見ればわかることであるが，執筆時点では目次がまだない．そこで，筆者がときどきこの節を読み返し，全体の構成を確認しているのである．

本書は 3 部構成になっている．第 I 部はプログラミング言語から抽象数学への登山である．第 II 部は抽象数学の山頂から Haskell の森へ下っていく道である．第 III 部はこれまで歩いた道全体を俯瞰する．

第 I 部は対応する Haskell のコードを脚注に記載している．脚注部分は Haskell を一通り覚えた後で読み返してもらいたい．[*1]

1.5　この章のまとめ*

1. ...
2. 各章にはこのように「この章のまとめ」を置く．



[*1] Haskell のコードは `x = 1` のようにタイプライタ体で書く．

第 2 章

カリー風な書き方*

本書では一般の数学書やプログラミングの教科書からは少し異なった記法を用いる．ある概念が発明されてからずっと後になって正しい記法が見つかり，それがきっかけとなって正しく理解されるという現象は歴史上よくあることである．本書でも様々な新しい記号，記法を導入するが，この章では Haskell に近い記法から始めることにする．

2.1 関数

数学やプログラミング言語には書き方に一定の決まりがある．この章ではまず「カリー風の」数式記述方式を見てみることにする．「カリー風」というのは，数学者ハスケル・カリーから名前を借りた言い方で，筆者が勝手に命名したものだ．

カリー風の書き方は数学の教科書やプログラミングの教科書で見かけるものとは若干違うが，圧倒的にシンプルで Haskell との親和性も高く，慣れてくると非常に読みやすいものなので，本書でも全面的に採用する．

まずは **関数** から見ていくことにしよう．Python や一般的な数学書では引数 x をとる関数 f を

Python
```
f(x)
```

と書くが，括弧は冗長なので今後は

$$fx \tag{2.1}$$

と書くことにする．[*1]

関数 f に引数 x を「食わせる」ことを **関数適用** と呼ぶ．もし fx と書いてあったら，それは f と x の積，つまり $f*x$ ではなく，従来の $f(x)$ すなわち関数 f に引数 x を与えているものと解釈する．高校生向けの数学書でも $\sin x$ のように三角関数に限ってはカリー風に書くことになっているので，まるで馴染みがないということもないだろう．なお，関数はいつも引数の左側に書くことにする．これを「関数 f が変数 x の **左から作用する**」と言い，また関数 f のことを **左作用素** とも呼ぶ．

複数引数をとる関数を Python や一般的な数学の教科書では

Python
```
g(x, y)
```

と書くが，これも括弧が冗長なので今後は gxy と書く．この場合式 gxy は左を優先して結合する．つまり

$$gxy = (gx)y \tag{2.2}$$

である．これは引数 y に関数 (gx) が左から作用していると解釈する．関数 (gx) は引数 x に関数 g を作用させて作った関数である．引数に「飢えた」関数 (gx) を **部分適用** された関数と呼ぶ．

このように式の左側を優先的に演算していくことを **左結合** と呼ぶ．Haskell の場合，関数適用はいつも左結合である．

2.2 ラムダ式

引数 x をとり値 $1+x$ を返す **ラムダ式** を Python では

Python
```
lambda x: 1+x
```

と書くが，我々はより簡潔に

$$\backslash x \to 1 + x \tag{2.3}$$

[*1] Haskell では関数 `f` に引数 `x` を適用させることを `f x` と書く．

と書くことにする．この式は多くの書物で $\lambda x\,.\,1+x$ と記述されるところである．しかし我々はすべてのギリシア文字を変数名のために予約しておきたいのと，ピリオド記号 (.) が今後登場する二項演算子 · と紛らわしいため，上述の記法を用いる．[*2]

ラムダ式は関数である．ラムダ式を適用するには，ラムダ式を括弧で包む必要がある．例を挙げる．

$$(\backslash\, x \to 1+x)2 \tag{2.4}$$

この式は結果として 3 を返す．

複数引数をとるラムダ式は例えば

$$\backslash\, xy \to x+y \tag{2.5}$$

のように引数を並べて書く．

本書では新たに，次のラムダ式記法も導入する．式中に記号 $\diamond$ が現れた場合，その式全体がラムダ式であるとみなす．記号 $\diamond$ の部分には引数が入る．第 n 番目の $\diamond$ には第 n 番目の引数が入る．例えばラムダ式 $\backslash\, xy \to x+y$ は

$$\diamond + \diamond \tag{2.6}$$

と書いても良い．式を左から読んで 1 番目の $\diamond$ が元々の x すなわち第 1 引数を，2 番目の $\diamond$ が元々の y すなわち第 2 引数を意味する．この省略記法はプログラミング言語 Scheme における cut プロシジャに由来する．[*3]

2.3 パタンマッチ・ガード・条件分岐

関数の定義は，基本的にはラムダ式の変数への代入である．引数 x をとり値 $(\sin x)/x$ を返す関数 f は

$$f = \backslash\, x \to (\sin x)/x \tag{2.7}$$

と定義できる．ただし，この省略形として

$$fx = (\sin x)/x \tag{2.8}$$

と書いても良い．この書き方は今後頻出するので，是非覚えておいてもらいたい．[*4]

関数に **スペシャルバージョン** がある場合はそれらを列挙する．例えば引数が 0 の場合は特別に戻り値が 1 であり，その他の場合は関数 f と同じ振る舞いをする関数 f' を考える．このとき f' は

$$\begin{cases} f'0 = 1 \\ f'x = (\sin x)/x \end{cases} \tag{2.9}$$

ように定義する．これを関数の **パタンマッチ** と呼ぶ．[*5]

関数のパタンマッチは，関数の内部に書いても良い．関数内部にパタンマッチを書きたい場合は

$$f'x = \mathsf{case}\, x\, \mathsf{of} \begin{cases} 0 \dashrightarrow 1 \\ \_ \dashrightarrow (\sin x)/x \end{cases} \tag{2.10}$$

のように書く．ここに _ は任意の値の意味である．パタンマッチは上から順番にマッチングしていくため，この場合は 0 以外を意味する．[*6]

関数定義にパタンマッチではなく **場合分け** が必要な場合は **ガード** を用いる．例えば引数の値が負の場合は -1 を，0 の場合は 1 を，それ以外の場合は関数 f と同じ振る舞いをする関数 f'' は

$$\begin{aligned} f''x\,|_{x<0} &= 0 \\ |_{x\equiv 0} &= 1 \\ |_{\mathsf{otherwise}} &= (\sin x)/x \end{aligned} \tag{2.11}$$

という風に定義する．[*7]

関数定義の場合分けを駆使すれば **条件式** はなくても構わないが，条件式の記法があるのは便利である．Python には

[*2] Haskell ではラムダ式 $\backslash\, x \to 1+x$ を `\x -> 1+x` と書く．ラムダ式は元々は $\hat{x}\,.\,x+1$ のように書かれていた．これが次第に $\hat{}\,x\,.\,x+1$ となり，$\lambda x\,.\,x+1$ に変化していったと言われている．Haskell が λ の代わりに $\backslash$ 記号を使うのは，その形が似ているからである．

[*3] この記法は Haskell にはない．

[*4] Haskell では $f = \backslash\, x \to (\sin x)/x$ を `f = \x -> (sin x)/x` と書き，一方 $fx = (\sin x)/x$ を `f x = (sin x)/x` と書く．

[*5] Haskell では

```
f' 0 = 1
f' x = (sin x)/x
```

と書く．Haskell ではプライム記号 (') はアルファベットと同一視される．

[*6] Haskell では

```
f' x = case x of
  0 -> 1
  _ -> (sin x)/x
```

と書く．

[*7] Haskell では

```
f'' x| x<0       = 0
     | x==0      = 1
     | otherwise = (sin x)/x
```

と書く．

Python
```
def f(x):
  if x==0:
    return 1
  else:
    return sin(x)/x
```

のような **制御構造** としての条件文があるが，我々は値を持つ **条件式** を考える．

我々の条件式とは

$$fx = \mathsf{if}\ x \equiv 0\ \mathsf{then}\ 1\ \mathsf{else}\ (\sin x)/x \tag{2.12}$$

のように if 節，then 節，及び else 節からなるものであって，then 節も else 節も省略できないものとする．if 節の式の値が真 (True) であれば then 節の式が評価され，偽 (False) であれば else 節の式が評価される．我々の条件式は C/C++ における条件演算子（三項演算子）と等しく見えるが，Haskell の場合は遅延評価が行われるため，結果として条件式の **短絡評価** が行われる点が異なる．[*8]

if 節の中身は真理値を返す関数であれば良いので，関数 p を

$$p = (\equiv)0 \tag{2.13}$$

としておき，

$$fx = \mathsf{if}\ px\ \mathsf{then}\ 1\ \mathsf{else}\ (\sin x)/x \tag{2.14}$$

と書く方法もしばしば用いられる．

条件式は紙面が許せば次のように書いても良い．

$$fx = \begin{cases} 1 & \mathsf{if}\ x \equiv 0 \\ (\sin x)/x & \mathsf{otherwise} \end{cases} \tag{2.15}$$

左ブレース（中括弧）の位置が関数のスペシャルバージョンの定義の時と異なっていることに注目しよう．[*9]

2.4 余計な話：局所変数

数式が長く続くとき，読みやすさのために局所変数を導入すると便利である．例えば

$$y = f(1 + x) \tag{2.16}$$

という式のうち，先に $1 + x$ の部分を計算して x' のように名前をつけておきたいこともあるであろう．そんなときは

$$y = \mathsf{let}\ x' \triangleq 1 + x\ \mathsf{in}\ fx' \tag{2.17}$$

と書くことにする．このようにして導入された x' を **局所変数** と呼ぶ．[*10]

式 (2.17) は局所変数を後ろに回して

$$y = fx'\ \mathsf{where}\ x' \triangleq 1 + x \tag{2.18}$$

のように書いても良い．[*11]

式 (2.17) や式 (2.18) はラムダ式を使った **糖衣構文**（シンタックスシュガー）であり，一般に

$$\mathsf{let}\ x' \triangleq gx\ \mathsf{in}\ fx' = fx'\ \mathsf{where}\ x' \triangleq gx \tag{2.19}$$

$$= (\backslash x' \to fx')(gx) \tag{2.20}$$

である．

2.5 この章のまとめ*

1. 変数 x に関数 f を適用することを fx と書く．
2. 変数 x, y に関数 g を適用することを gxy と書く．関数適用は左結合するので $gxy = (gx)y$ である．この関数 (gx) は部分適用された関数と呼ぶ．
3. 関数はラムダ式で定義する．
4. ラムダ式は $\backslash x \to (\sin x)/x$ のように表記する．
5. ラムダ式は $f\diamond$ のように無名パラメタ $\diamond$ を用いて表記しても良い．
6. 関数定義 $f = \backslash x \to (\sin x)/x$ は $fx = (\sin x)/x$ と省略表記できる．
7. 関数定義にはパタンマッチが使える．例えば $f'0 = 1; f'x = (\sin x)/x$ と定義できる．
8. パタンマッチは関数本体に書いても良い．関数 f' の例で言えば $f' = \mathsf{case}\ x\ \mathsf{of}\ 0 \dashrightarrow 1; \_ \dashrightarrow (\sin x)/x$ と定義できる．
9. 関数定義にはガードが使える．例えば $f''|_{x<0} = 0; f''|_{x\equiv 0} = 1; f''|_{\mathsf{otherwise}} = (\sin x)/x$ と定義できる．

[*8] Haskell では $fx = \mathsf{if}\ x \equiv 0\ \mathsf{then}\ 1\ \mathsf{else}\ (\sin x)/x$ を `f x = if x==0 then 1 else (sin x)/x` と書く．

[*9] Haskell ではいつも `if`, `then`, `else` を使った式にする．

[*10] Haskell では $y = \mathsf{let}\ x' \triangleq 1 + x\ \mathsf{in}\ fx'$ を `y = let x' = 1+x in f x'` と書く．

[*11] Haskell では $y = fx'\ \mathsf{where}\ x' \triangleq 1 + x$ は `y = f x' where x' = 1+x` と書く．

10. 条件分岐は $\mathsf{if}\, x\, \mathsf{then}\, y\, \mathsf{else}\, z$ と書き，$x \equiv \mathrm{True}$ の時には y が，$x \equiv \mathrm{False}$ のときには z が式の値になる．
11. 局所変数は $y = \mathsf{let}\, x' \triangleq \cdots \mathsf{in}\, fx'$ または $y = fx' \,\mathsf{where}\, x' \triangleq \cdots$ という書き方で導入できる．

式の評価順序

Haskell は参照透過な言語なので，式がいつ評価されるかを考える必要はない．一方で，参照透過でない言語は式の評価順序をいつも気にしておく必要がある．例えば C は関数引数の評価順序を定めていないので，次のコード

C

```
int i = 0;
printf("%d, %d\n", ++i, ++i);
```

は画面に 1，2 を出力する場合もあるし，2，1 を出力する場合もある．

第 3 章

さらにカリー風な書き方*

我々は関数とラムダ式の「カリー風」な書き方を見てきた．この章ではさらに演算子，関数合成，条件分岐についても「カリー風」な書き方を見ていく．

...

3.1 演算子

演算子 は関数の特別な姿である．演算子は **作用素** と呼んでも良い．どちらも英語の operator の和訳である．演算子は普通アルファベット以外のシンボル 1 個で表現し，変数や関数の前に置いて直後の変数や関数に作用させるか，2 個の変数や関数の中間に置いてその両者に作用させる．例えば $-x$ のマイナス記号 $(-)$ は変数の前に置いて直後の変数 (x) に作用する演算子であり，$x + y$ のプラス記号 $(+)$ は 2 個の変数の間に置いてその両者 (x, y) に作用する．

1 個の変数または関数に作用する演算子を **単項演算子** と呼び，2 個の変数または関数に作用する演算子を **二項演算子** と呼ぶ．本書では単項演算子はすべて変数の前に置く，すなわち **前置** する．前置する演算子のことを **前置演算子** と呼ぶが，数学者は同じものを左作用素と呼ぶ.

二項演算子のうちよく使われるものは和 $(+)$，積 $(*)$，論理和 $(\vee)$，論理積 $(\wedge)$，同値 $(\equiv)$，大なり $(>)$，小なり $(<)$ 等である．二項演算子はたとえ積記号であっても省略できない．二項演算子は多数あるので，その都度説明する．

二項演算子は **中置** することが基本であるが，括弧で包むことで前置することも可能である．任意の二項演算子 ★ について $x \bigstar y$ 及び $(\bigstar)xy$ は全く同じ意味である．すなわち

$$(\bigstar)\, xy = x \bigstar y \tag{3.1}$$

である．従って，二項演算子と 2 引数関数に本質的な差はない．本書では演算子と関数という用語は全く同じ意味で用いる．[*1]

一般の関数が左結合であることを思い出すと，二項演算子を関数に見立てた $(\bigstar)$ も

$$(\bigstar)xy = ((\bigstar)x)y \tag{3.2}$$

であるから，部分適用が可能である．式 (3.2) から第 2 引数 y を取り除いて $(\bigstar)x$ という「餓えた」1 引数関数を取り出せる．例えば関数 $((+)1)$ は引数に 1 を加える関数である．[*2]

二項演算子の部分適用に限って **セクション** と言う記法も用いられる．二項演算子 ★ に対して $(\bigstar\, x)$ および $(x \bigstar)$ はそれぞれ

$$(\bigstar\, x) = \diamond \bigstar x \tag{3.3}$$
$$(x \bigstar) = x \bigstar \diamond \tag{3.4}$$

である．例えば $(1+)$ は $((+)1)$ と等価である． [*3]

なお，二項演算子の結合性，すなわち左結合か右結合かは，演算子によって異なる．また演算の優先順位を明示的に与えるために括弧が用いられる．

本書では 3 個以上の変数に同時に作用する演算子は扱わない.

3.2 関数合成と関数適用

ある変数に複数の関数を順に適用することはよくあることである．例えば

[*1] Haskell では任意の二項演算子を括弧で包むことで前置演算子として使うことができる．例えば `x+y` と `(+)x y` は同じ結果を返す．逆に任意の 2 引数関数 `f` は `x `f` y` と書くことで中置することができる.

[*2] Haskell では $((+)1)$ を `((+)1)` と書く.

[*3] Haskell では $(1+)$ を `(1+)` と書く.

Python
```
y = f1(x)
z = f2(y)
```

あるいは，同じことであるが

Python
```
z = f2(f1(x))
```

とすることがある．本書の記法で書けば

$$z = f_2(f_1 x) \tag{3.5}$$

である．式 (3.5) から括弧を省略して $z = f_2 f_1 x$ としてしまうと，関数適用は左結合するから $z = (f_2 f_1) x$ の意味になってしまう．関数 f_2 が引数に関数を取るのでない限り $(f_2 f_1)$ は無意味なので，式 (3.5) の括弧は省略できない．

ここで，引数のことは忘れて，関数 f_1 と関数 f_2 を先に **合成** しておきたいとしよう．その合成を $f_2 \centerdot f_1$ と書く．演算子 $\centerdot$ は **関数合成演算子** と呼ぶ．合成はラムダ式を使って

$$f_2 \centerdot f_1 = f_2(f_1 \diamond) \tag{3.6}$$

と定義できる．関数合成演算子 $\centerdot$ は関数適用よりも優先順位が高く，$(f_2 \centerdot f_1)\, x$ は単に $f_2 \centerdot f_1 x$ と書いても良い．この記法は括弧の数を減らすためにしばしば用いられる．式 (3.5) は関数合成演算子を用いると

$$z = f_2 \centerdot f_1 x \tag{3.7}$$

と書ける．[*4]

関数合成演算子とは逆に，結合の優先順位の低い **関数適用演算子** も考えておくと便利なこともある．関数適用演算子 $ を関数合成演算子と同じように次のように定義しておく．

$$f_2 \,\$\, f_1 = f_2\,(f_1 \diamond) \tag{3.8}$$

演算子 $ の優先順位は関数適用も含めあらゆる演算子よりも低いものとする．関数適用演算子を用いて式 (3.5) を書き直すと

$$z = f_2 \,\$\, f_1 x \tag{3.9}$$

となる．演算子 $ の優先順位は足し算よりも低いので $f(x+1)$ は $f \,\$\, x + 1$ と書くこともできる．演算子 $ を閉じ括弧のいらない開き括弧と考えてもよい．[*5]

関数適用演算子のもう一つの興味深い使い方は，関数適用演算子の部分適用である．セクション $(\$\, x)$ を用いると

$$(\$\, x) f = f \,\$\, x \tag{3.10}$$

であるから，関数適用演算子を用いて引数を関数に渡すことができる．[*6]

3.3　高階関数*

関数を引数に取ったり，あるいは関数を返す関数のことを **高階関数** と呼ぶことがある．関数合成演算子と関数適用演算子は高階関数の好例である．

他に例えば, 引数として整数 a を取り, 関数 $fx = a+x$ を返すような関数 g を

$$ga = a + \diamond \tag{3.11}$$

のように定義することも可能である．このとき,

$$f = g(100) \tag{3.12}$$
$$r = f1 \tag{3.13}$$

とすれば $r = 101$ を得る．[*7]

高階関数は今後度々顔をだすことになる．第 5 章に登場するマップ演算子や畳込み演算子は高階関数の一種である．

3.4　余計な話：クロージャ

ラムダ式をサポートするほとんどのプログラミング言語は，**レキシカルクロージャ** をサポートする．レキシカルクロージャとは，ラムダ式が定義された時点での，周囲の環境をラムダ式に埋め込む機構である．例えば

$$a = 100 \tag{3.14}$$
$$h = a + \diamond \tag{3.15}$$

というラムダ式があるとする．当然我々は関数 h がいつも $h = 100 + \diamond$ であることを期待するし，Haskell においてはいつも保証される．

[*4] Haskell では関数 `f2` と関数 `f1` の合成は `f2.f1` である．式 $z = f_2 \centerdot f_1 x$ は `z = f2.f1 x` と書く．

[*5] Haskell では $f_2 \,\$\, f_1 x$ を `f2 $ f1 x` と書く．

[*6] Haskell では $(\$\, x) f$ を `($x)f` と書く．

[*7] Haskell では $ga = a + \diamond$ を $ga = \backslash x \to a + x$ と展開しておいて `g a = \x -> a+x` と書く．

ところが，参照透過性のない言語，言い換えると変数への破壊的代入が許されている言語では，変数 a の値がいつ変わっても不思議ではない．そこで，それらの言語では関数 h が定義された時点での a の値を，関数 h の定義に含めておく．これがレキシカルクロージャの考え方である．

Haskell ではそもそも変数への破壊的代入がないので，関数 h がレキシカルクロージャであるかどうか悩む必要はない．あえて言えば，Haskell ではラムダ式はいつもレキシカルクロージャである．もしあなたのそばの C++ プログラマが「え？ Haskell にはレキシカルクロージャが無いの？」などと聞いてきたら，「ええ，Haskell には破壊的代入すらありませんから」と答えておこう．

3.5　この章のまとめ*

1. 関数と演算子は同じものである．
2. 任意の二項演算子 ★ について $x \bigstar y$ と $(\bigstar)xy$ は全く同じ意味である．
3. 任意の二項演算子 ★ について $(\bigstar x) = \diamond \bigstar x$ であり，$(x \bigstar) = x \bigstar \diamond$ である．これらの記法をセクション記法と呼ぶ．
4. 関数 f と関数 g を合成した関数を $f \mathbin{.} g$ と書く．
5. 式 $f \mathbin{\$} gx$ は $f(gx)$ の意味である．
6. ...
7. Haskell のラムダ式はいつもレキシカルクロージャである．

C のブロック拡張

C は公式にはラムダ式を採用していない．しかし C にラムダ式を持たせる拡張はいくつか提案されている．その一つが Apple 社が自社 OS の Grand Central Dispatch 機能のために行った「ブロック拡張」である．このブロック拡張を用いたクロージャの例を見てみよう．

C

```
#include <Block.h>
#include <stdio.h>
typedef int (^int_to_int)(int);
int_to_int make_plus_n(int n) {
  return Block_copy(^(int x) {
      return n+x;
    });
}
int main(void) {
  int_to_int make_plus_10
    = make_plus_n(10);
  int x = make_plus_10(1);
  printf("x = %d\n", x);
  return 0;
}
```

コード中の `^(int x) { return n+x; }` がブロック（クロージャ）である．C のブロック拡張はオープンソースコミュニティに還元されているため，GCC や clang で使用可能である．

第 4 章

型*

プログラマのいう型とは，数学者のいう集合のことである．

…

4.1 データ型

型 とは変数が取りうる値に言語処理系が与えた制約のことである．Haskell を含む多くのコンパイラ言語は **静的型付け** と言って，コンパイル時までに変数の型が決まっていることをプログラマに要求する．一方，Python のようなインタプリタ言語はたいてい **動的型付け** と言って，プログラムの実行時まで変数の型を決めない．

変数に型の制約を設ける理由は，プログラム上のエラーが減ることを期待するためである．例えば真理値が必要とされるところに整数値の変数が来ることは悪い予兆である．一方で C 言語のように全ての変数にいちいち型を明記していくのも骨が折れる．

数学者や物理学者は変数に型の制約を求める一方，新しい変数の型は明記せず読者に推論させる方法をしばしばとる．例えば，質量 m は「スカラー」という型を持つし，速度 v は「3 次元ベクトル」という型を持つ．スカラーと 3 次元ベクトルの間に足し算は定義されていないため，例えば $m + v$ という表記を見たときに，両者の型を知っていれば直ちにエラーであることがわかる．

✠

Haskell にはよく使う型が予め用意されている．例えば **論理型** は論理値すなわち真 (True) または偽 (False) という値をとる変数の型である．ある変数 x が論理型であることを，Haskell では

$$x :: \mathbf{Bool} \tag{4.1}$$

と書く．数学者なら同じことを

$$x \in \mathbb{B} \tag{4.2}$$

と書くところであるが，ここは Haskell の流儀に従おう．[*1]

他に整数を表す **整数型** がある．整数型には 2 種類あって，その一つは **Int** である．この **Int** は C の `int` と似た「計算機にとって都合の良い整数」である．計算機にとって都合の良い整数とは，例えば 64 ビット計算機の場合 -2^{63} から $2^{63} - 1$ の間の整数という意味である．

もう一つの整数型は **Integer** である．この **Integer** は計算機にとっては非常識なぐらい大きな，あるいは小さな値を表すことができる．

計算機は残念ながら無限精度の実数を扱えない．そこで標準精度（単精度）の **浮動小数点数型** である **Float** と，**倍精度浮動小数点型** である **Double** が提供される．

もう一つ，計算機ならではの型がある．それは **Int** とよく似ているが，特別に文字を扱うために考えられた **文字型** **Char** である．文字といってもその中身は整数である．整数ではあるが，わざわざ別な型とするのには理由がある．

理由の第一は，文字が小さな整数であるため，文字型を独立して定義しておくことでメモリを節約できるのである．特にメモリが高価であった時代はこれが唯一の理由であった．現在でも，整数が一般に 64 ビットを消費するのに対し，UTF8 文字エンコードを用いている場合，アルファベットは 8 ビットしか消費しない．

理由の第二は，単純に整数と文字が異なるからであ

[*1] Haskell では $x :: \mathbf{Bool}$ を `x :: Bool` と書く．

る．文字を表す変数に整数を代入するのは悪い兆しである．

理由の第三は，文字が数値にエンコードされる方式が可変長である場合に備えて，整数と区別しておくためである．例えば UTF8 文字エンコードは可変長エンコーディングを行う．

関数にも型がある．例えば整数引数を一つ取り，整数を返す関数 f は

$$f :: \mathbf{Int} \mapsto \mathbf{Int} \tag{4.3}$$

という型を持つ．上式は

$$f :: \underbrace{\mathbf{Int}}_{x} \mapsto \underbrace{\mathbf{Int}}_{fx} \tag{4.4}$$

のようにイメージすると良い．これは関数 f が集合 **Int** から集合 **Int** への **写像** であると読む．

4.2　カリー化

Haskell では，どのような関数であれ引数は 1 個しかとらない．引数が 2 個あるように見える関数として，例えば gxy があったとしよう．ここに g は関数, x, y は変数である．関数適用が左結合であるから，これは $(gx)\,y$ である．ここに (gx) は引数 y をとる関数であると見ることができる．つまり，関数 f とは引数 x をとり「引数 y をとって値を返す関数 (gx) を返す」関数であると言える．

二項演算 $x + y$ は $(+)xy$ とも書けたことを思い出そう．これも左結合を思い出すと

$$(+)xy = ((+)x)\,y \tag{4.5}$$

であるから，y という引数を $((+)x)$ という関数に食わせていると解釈できる．

ラムダ式の場合は話はもっと単純で，形式的に

$$\backslash\, xy \to x + y = \backslash\, x \to (\backslash\, y \to x + y) \tag{4.6}$$

のように展開すれば 1 引数にできる．矢印 $\to$ は **右結合** である．そこでこのラムダ式は括弧を省略して

$$\backslash\, xy \to x + y = \backslash\, x \to \backslash\, y \to x + y \tag{4.7}$$

とも書かれる．

複数引数をとる関数を 1 引数関数に分解することを **カリー化** と呼ぶ．これはこの分野の先駆者であるハスケル・カリーの名前に由来する．

整数引数を二つ取り，整数を返す関数 g は

$$g :: \mathbf{Int} \mapsto \mathbf{Int} \mapsto \mathbf{Int} \tag{4.8}$$

という型を持つ．写像の矢印記号は右結合するので，これは

$$g :: \mathbf{Int} \mapsto (\mathbf{Int} \mapsto \mathbf{Int}) \tag{4.9}$$

と同じ意味である．上式は

$$g :: \underbrace{\mathbf{Int}}_{x} \mapsto \underbrace{\overbrace{\mathbf{Int}}^{y} \mapsto \overbrace{\mathbf{Int}}^{(gx)y}}_{gx}$$

のようにイメージすると良い．自然言語で考えると **Int** 型の引数を一つ取り，**Int** 型の引数を一つ取って **Int** 型の値を返す関数を返す，と読める．

4.3　多相型と型クラス

整数型 (**Int**) と浮動小数点型 (**Float**) はよく似ている．どちらも値同士を比較可能で，それ故どちらにも等値演算子が定義されている．

整数型の等値演算子は

$$(\equiv) :: \mathbf{Int} \mapsto \mathbf{Int} \mapsto \mathbf{Bool} \tag{4.10}$$

であり，浮動小数点型の等値演算子は

$$(\equiv) :: \mathbf{Float} \mapsto \mathbf{Float} \mapsto \mathbf{Bool} \tag{4.11}$$

である．

このように型が異なっても（だいたい）同じ意味で定義されている演算子のことを **多相的** な演算子と呼ぶ．等値演算子は多相的な演算子の例である．

具体的な型を指定せずに，仮の変数で表したものを **型パラメタ** と呼ぶ．我々は型パラメタをボールド体で表す．いま型を表す仮の変数を **a** として，等値演算子の型を

$$(\equiv) :: \mathbf{a} \mapsto \mathbf{a} \mapsto \mathbf{Bool} \tag{4.12}$$

と表現してみよう．このような型パラメタを用いた型を総称して **多相型** と呼ぶ．

実は式 (4.12) は不完全なものである．このままでは型 **a** に何の制約もないため，等値演算の定義されていない型が来るかもしれないからである．

そこで，型自身が所属する，より大きな型があるとしよう．そのような型を我々は **型クラス** と呼ぶ．例えば型 **Bool**, **Int**, **Integer**, **Float**, **Double** は全て等値演算が定義できるので，型クラス Eq に属すとする．この関係を我々は

$$\mathrm{Eq} \supset \mathbf{Bool}, \mathbf{Int}, \mathbf{Integer}, \mathbf{Float}, \mathbf{Double} \tag{4.13}$$

と書く．ここに $\mathrm{Eq} \supset \mathbf{a}$ と書いて「型 **a** は型クラス Eq の **インスタンス** である」と読む．

式 (4.12) に型クラスの制約を加えてみよう．型 **a** は型クラス Eq に属さなければならないから，新たな記号 $\Rightarrow$ を使って

$$(\equiv) :: (\mathrm{Eq} \supset \mathbf{a}) \Rightarrow \mathbf{a} \mapsto \mathbf{a} \mapsto \mathbf{Bool} \tag{4.14}$$

と書くことにする．[*2]

型 **a** の変数同士の間で大小関係が定義されている場合，かつその型が型クラス Eq に属する場合，その型は型クラス Ord にも属する．型クラス Ord に属する型は比較演算子 $<, \leq, \geq, >$ を提供する．例えば型 **Int** は型クラス Ord に属すが，型 **Bool** は型クラス Ord に属さない．

型 **a** の変数同士の間で四則演算関係が定義されている場合，かつその型が型クラス Eq に属する場合，その型は型クラス Num にも属する．型クラス Num に属する型は二項演算子 $+, -, *, /$ を提供する．ここに $-$ は二項演算子のマイナスである．

型 **a** が型クラス Ord 及び型クラス Num に属しているとき，かつそのときに限り，型 **a** は型クラス Real にも属する．

型 **a** の変数について，一つ小さい値を返す関数 pred と一つ大きい値を返す関数 succ が定義されているとき，かつそのときに限り，型 **a** は型クラス Enum に属する．

型 **a** が型クラス Real 及び型クラス Enum に属しているとき，かつそのときに限り，型 **a** は型クラス Integral にも属する．

これらの関係を表にまとめたものが表 4.1 である．この表から，型 **Int** は型クラス Integral, Real, Ord, Num, Enum, Eq に属しているのに対し，型 **Bool** は Ord, Enum, Eq にのみ属しているのがわかる．

[*2] Haskell では $(\equiv) :: (\mathrm{Eq} \supset \mathbf{a}) \Rightarrow \mathbf{a} \mapsto \mathbf{a} \mapsto \mathbf{Bool}$ を `(==) :: (Eq a) => a -> a -> Bool` と書く．記号 $\supset$ は省略する．

4.4 余計な話：モノイド*

整数全てからなる **集合** を $\mathbb{Z}$ で表すことにする．計算機科学で整数と言うと，本当の整数と，例えば -2^{63} から $2^{63}-1$ までの間の整数の意味と両方あるが，今は前者の意味である．

集合 $\mathbb{Z}$ の任意の **元**（**要素**）z を

$$z :: \mathbb{Z} \tag{4.15}$$

と書く．

二つの整数 $z_1, z_2 :: \mathbb{Z}$ があるとしよう．両者の間には **足し算** $(+)$ が定義されており，その結果すなわち **和** もまた整数である．ここで

$$z_1 + z_2 :: \mathbb{Z} \tag{4.16}$$

であるとき，演算子 $+$ が集合 $\mathbb{Z}$ に対して **全域性** を持つと言う．一般に集合 **A** の元に対して二項演算子 $\bigstar$ が定義されていて，$a_1, a_2 :: \mathbf{A}$ のときに

$$a_1 \bigstar a_2 :: \mathbf{A} \tag{4.17}$$

である場合，つまり演算子 $\bigstar$ が集合 **A** に対して全域性を持つ場合，組み合わせ $(\mathbf{A}, \bigstar)$ を **マグマ** と呼ぶ．組み合わせ $(\mathbb{Z}, +)$ はマグマの例であり，$(\mathbb{Z}, *)$ もマグマの例である．

他に **論理集合** $\mathbb{B} = \{\mathrm{True}, \mathrm{False}\}$ に対して，**論理和** $(\vee)$ は全域性を持つから，組み合わせ $(\mathbb{B}, \vee)$ はマグマであるし，同様に **論理積** $(\wedge)$ も全域性を持つから，組み合わせ $(\mathbb{B}, \wedge)$ もマグマである．論理集合 $\mathbb{B}$ とは論理型 **Bool** を数学風に言い換えたものである．

マグマのうち，演算を 2 回続ける場合，その順序によって結果が異ならない，つまり

$$(a_1 \bigstar a_2) \bigstar a_3 = a_1 \bigstar (a_2 \bigstar a_3) \tag{4.18}$$

ただし $a_1, a_2, a_3 :: \mathbf{A}$ のとき，組み合わせ $(\mathbf{A}, \bigstar)$ のことを **半群** と呼ぶ．この式 (4.18) で表される性質を **結合性** と呼ぶ．組み合わせ $(\mathbb{Z}, +)$, $(\mathbb{Z}, *)$, $(\mathbb{B}, \vee)$, $(\mathbb{B}, \vee)$ はすべて半群である．

表 4.1 型と型クラス

型クラス	Integral											
	Real										Enum	
	Ord					Num						
	Eq					Eq						
インタフェース	≡	<	≤	≥	>	≡	+	−	∗	/	pred	succ
Char	✓	✓	✓	✓	✓	✓					✓	✓
Bool	✓	✓	✓	✓	✓	✓					✓	✓
Int	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Integer	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Float	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		
Double	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		

ところで，整数全体の集合 $\mathbb{Z}$ には特別な元 $0 :: \mathbb{Z}$ がある．この元 0 は $z :: \mathbb{Z}$ のとき

$$0 + z = z + 0 = z \tag{4.19}$$

という性質を持つ．この 0 を演算 + における **単位元** と呼ぶ．足し算のことを **加法** とも言うので 0 のことは **加法単位元** と呼ぶこともあるし，文字通り **零元** と呼ぶこともある．

一般に，$a, \emptyset_{\text{Left}}, \emptyset_{\text{Right}} :: \mathbf{A}$ として

$$\emptyset_{\text{Left}} \dot{+} a = a \dot{+} \emptyset_{\text{Right}} = a \tag{4.20}$$

であるとき，元 $\emptyset_{\text{Left}}, \emptyset_{\text{Right}}$ を単位元と呼ぶ．我々は多くの場合 $\emptyset_{\text{Left}} = \emptyset_{\text{Right}}$ であるケースを扱うので，二つの単位元を区別する必要はほとんどないが，必要な場合は $\emptyset_{\text{Right}}$ を **左単位元**，$\emptyset_{\text{Right}}$ を **右単位元** と呼ぶ．

組み合わせ $(\mathbf{A}, \dot{+}, \emptyset_{\text{Left}}, \emptyset_{\text{Right}})$ のことを **モノイド** または **単位的半群** と呼ぶ．例えば $(\mathbb{Z}, +, 0, 0)$ はモノイドであるし，$(\mathbb{Z}, *, 1, 1)$，$(\mathbb{B}, \vee, \text{False}, \text{False})$，$(\mathbb{B}, \wedge, \text{True}, \text{True})$ もモノイドである．これらのモノイドは全て左単位元と右単位元が同じなので，それぞれ $(\mathbb{Z}, +, 0)$, $(\mathbb{Z}, *, 1)$, $(\mathbb{B}, \vee, \text{False})$, $(\mathbb{B}, \wedge, \text{True})$ とも書く．

このように，数学者は数の性質を抽象化し，集合とその集合に対する演算というものの見方をよく行う．プログラミングの言葉で言えば，複数のクラスに共通のインタフェースを定義するようなものである．

表 4.2 モノイド（単位的半群）

型	演算子	単位元
Bool	$\vee$	False
Bool	$\wedge$	True
Int	+	0
Int	∗	1
Float	+	0
Float	∗	1

表 4.2 に型と対応するモノイドの単位元，演算子の一覧を示す．

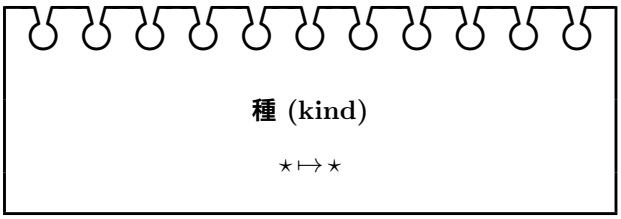
組み合わせ $(\mathbf{Int}, +, 0)$ はモノイドである．同様に $(\mathbf{Int}, *, 1)$，$(\mathbf{Float}, +, 0)$，$(\mathbf{Float}, *, 1)$，$(\mathbf{Bool}, \vee, \text{False})$, $(\mathbf{Bool}, \wedge, \text{True})$ もモノイドである．

4.5 この章のまとめ*

1. 集合 $\mathbf{A}$ の元 $a_1, a_2 :: \mathbf{A}$ について，二項演算子 ★ があり $a_1 \bigstar a_2 :: \mathbf{A}$ であるとき，すなわち演算子が全域性を有する場合 $(\mathbf{A}, \bigstar)$ のことをマグマと呼ぶ．
2. マグマ $\mathbf{A}$ の元 $a_1, a_2, a_3 :: \mathbf{A}$ について $(a_1 \bigstar a_2) \bigstar a_3 = a_1 \bigstar (a_2 \bigstar a_3)$ である場合，すなわち演算子が結合性を有する場合 $\mathbf{A}$ を半群と呼ぶ．
3. 半群のうち $\emptyset :: \mathbf{A}$ なる元 $\emptyset$ があり，任意の $a :: \mathbf{A}$

に対して $\emptyset \bigstar a = a \bigstar \emptyset = a$ である場合，すなわち単位元が存在する場合 **A** をモノイドと呼ぶ.

4. 関数は集合 **A** から集合 **B** への写像という型を持つ.
5. 複数引数を取る関数はカリー化によって，1 引数をとる複数の関数へ分解される.
6. 型のインタフェースをまとめたものを型クラスと呼ぶ.



第 5 章

リスト

型から作る型をコンテナと呼ぶ．代表的なコンテナはある型のホモジニアスな配列であるリストである．この章ではリストと，リストに対する重要な演算である畳み込み，マップを取り扱う．

5.1 リスト

同じ型の値を一列に並べたもの，つまりホモジニアスな配列のことを **リスト** と呼ぶ．Python ではリスト `ls` を

Python
```
ls = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

のように定義できる．

我々も 0 から始まり 9 まで続く整数のリストを $[0,1,2,3,4,5,6,7,8,9]$ と書くことにしよう．ただし，これでは冗長なので **等差数列** に限って簡略化した書き方を許す．例えば 0 から 9 までのリストは $[0,1\cdots 9]$ と書いても良い．[*1]

リストの中身の一つ一つの値のことを **要素** と呼ぶ．要素のことは元と呼んでも良いが，本書では要素と呼ぶことにする．要素も元も英語の element の和訳である．

複数の型の要素が混在してもよい配列のことをヘテロジニアスな配列と呼び，ホモジニアスな配列とは区別する．

今後，リストを指す変数は，リストであることを忘れないように変数名の肩に星印をつけて

$$x^\star = [0,1\cdots 9] \tag{5.1}$$

のように書くことにしよう．なお，変数 x とリスト変数 $x^\star$ は異なる変数であるとする．[*2]

Python ではリスト内包表記が使える．例えば 0 から 9 までの倍数のリストは次のように作った．

Python
```
ls = [x*2 for x in range(0, 10)]
```

ここに `range(a, b)` は `a` から増加する方向に連続する `b` 個の整数からなるリストを返す Python の関数である．

我々は内包表記のかわりにガードを使って

$$x^\star = \left[x*2\,|_{x\,\in\,[0,1\cdots 9]}\right] \tag{5.2}$$

のように書こう．ここに右辺のリストから一つずつ要素を取り出して左辺に代入する演算子 $\in$ を用いた．[*3]

ガードの中の式は複数あっても良い．例えば

$$x^\star = \left[x+y\,|_{x\,\in\,[0,1\cdots 9],\,y\,\in\,[0,1\cdots 5],\,x+y>3}\right] \tag{5.3}$$

は $0 \leq x \leq 9$ かつ $0 \leq y \leq 5$ の範囲で $x+y>3$ となる x 及び y から $x+y$ を並べたリストである．これは Python でいう

Python
```
ls = [x+y for x in range(0, 10) \
  for y in range(0,6) if x+y > 3]
```

のことである．[*4]

リストは **結合** できる．例えばリスト $x^\star$ とリスト $y^\star$

[*1] Haskell では `[0, 1..9]` と書く．ピリオドの数に注意しよう．

[*2] Haskell では星印のかわりに `s` を変数名にくっつけて `xs = [0, 1..9]` のように書く習慣がある．

[*3] Haskell では $x^\star = \left[x*2\,|_{x\,\in\,[0,1\cdots 9]}\right]$ を `xs = [x*2 | x<-[0,1..9]]` と書く．

[*4] Haskell では $x^\star = \left[x+y\,|_{x\,\in\,[0,1\cdots 9],\,y\,\in\,[0,1\cdots 5],\,x+y>3}\right]$ を `xs = [x+y | x<-[0,1..9], y<-[0,1..5], x+y>3]` と書く．

を結合したリストは

$$x^\star \oplus y^\star \tag{5.4}$$

と表現する.[*5]

リストは **無限個** の要素を持っても良い．例えば自然数全体を表すリスト $n^\star$ は

$$n^\star = [1, 2 \cdots] \tag{5.5}$$

のように定義して良い.[*6]

空リスト は [] で表す.[*7]

任意の型を **a** とするとき，**a** 型のリスト型を [**a**] と書く．型 **a** から型 [**a**] を生成する演算子を **リスト型コンストラクタ** と呼んで $List$ と書き

$$[\mathbf{a}] = List\,\mathbf{a} \tag{5.6}$$

とする．この等式の両辺は変数ではなく型名であることに注意しよう．型コンストラクタの概念は Python には無い（必要無い）が，静的型付け言語である C++ の「クラステンプレート」が相当する.[*8]

a 型の変数 x を入れた [**a**] 型の変数を作る演算子を **リスト値コンストラクタ** と呼ぶ．[**a**] 型の変数のことを **リスト変数** とも呼ぶ．**a** 型の変数 x からリスト値コンストラクタを使ってリスト $x^\star$ を作ることは

$$x^\star = [x] \tag{5.7}$$

と書く.[*9]

リスト型を表す [**a**] と，1 要素のリストである $[x]$ の違いにはいつも気をつけておこう．本書では中身がボールドローマン体ならばリスト型，中身がイタリック体ならリスト値である．またリストを保持する変数にも，本書では $x^\star$ のように星印をつけておくことにする．

ある型を包み込んだ別の型を一般に **コンテナ型** または単に **コンテナ** と呼ぶ．コンテナ型の変数を **コンテナ変数** と呼ぶ．コンテナ型は多相型の一種である．

[*5] Haskell では $x^\star \oplus y^\star$ を `xs++ys` と書く.
[*6] Haskell では $n^\star = [1, 2 \cdots]$ を `ns = [1, 2..]` と書く.
[*7] Haskell では空リストを `[]` で表す.
[*8] Haskell では表記上コンテナ型 [**a**] と型コンストラクタ式 $List\,\mathbf{a}$ を区別せず，両者とも `[a]` と書く.
[*9] Haskell では `xs = [x]` と書く.

5.2 畳み込み

我々はよくリストの総和を表現するために総和演算子 ($\sum$) を使う．総和演算子とはリスト $[x_0, x_1 \cdots x_n]$ に対して

$$\sum [x_0, x_1 \cdots x_n] = x_0 + x_1 + \cdots + x_n \tag{5.8}$$

で定義される演算子である．この表現を一般化してみよう．リスト $[x_0, x_1 \cdots x_n]$ が与えられたとき，任意の二項演算子を ★ として

$$\bigcup_a^\star [x_0, x_1 \cdots x_n] = a \bigstar x_0 \bigstar x_1 \bigstar \cdots \bigstar x_n \tag{5.9}$$

であると定義する．

この新しい演算子 $\bigcup$ は **畳み込み演算子** と呼ばれる．変数 a は **アキュムレータ** と呼ぶ．アキュムレータは右側の引数が空であった場合のデフォルト値と考えても良い.[*10]

Python 2.7 には畳み込み演算子に相当する `reduce` 関数があり，リスト `ls` の総和 `s` を

Python
```
# Python 2.7
ls = [0, 1, 2, 3, 4, 5]
s = reduce(lambda x, y: x+y, ls, 0)
```

のように求めることができる．この `reduce` 関数は Python バージョン 3 では非推奨になっているが，Ruby には受け継がれていて，Ruby では

Ruby
```
ls = [0, 1, 2, 3, 4, 5]
s = ls.inject(0) { |x, y| x+y }
```

と書ける．

リストの総和をとる演算子 $\sum$ は

$$\sum x^\star = \bigcup_0^+ x^\star \tag{5.10}$$

とすれば得られる．この式は両辺の $x^\star$ を省略して

$$\sum = \bigcup_0^+ \tag{5.11}$$

[*10] Haskell では $\bigcup_a^\star x^\star$ を `foldl (*) a xs` と書く.

とも書く．このように，ラムダ式を使わずに引数を省略してしまう書き方を **ポイントフリースタイル** と呼ぶ．ポイントフリースタイルは今後も頻出するので，是非慣れておいてもらいたい．[*11]

リストの要素のすべての積をとる演算子 $\prod$ は

$$\prod = \bigcup_{1}^{*} \tag{5.12}$$

とすれば得られる．

畳み込み演算子は第 1（上）引数に **a** 型と **b** 型の引数を取り **a** 型の戻り値を返す二項演算子，第 2（下）引数に **a** 型，第 3（右）引数に **b** 型のリストすなわち [**b**] 型を取り，**a** 型の値を返す．従って畳み込み演算子の型は

$$\bigcup :: (\mathbf{a} \mapsto \mathbf{b} \mapsto \mathbf{a}) \mapsto \mathbf{a} \mapsto [\mathbf{b}] \mapsto \mathbf{a} \tag{5.13}$$

である．

畳み込み演算子には次のようなもう一つのバリエーションがある．

$$\bigsqcup_{a}^{\bigstar} [x_0, x_1 \cdots x_n] = (x_0 \bigstar (x_1 \bigstar \cdots \bigstar (x_n \bigstar a))) \tag{5.14}$$

これは **右畳み込み** と呼ばれる演算子である．[*12]

畳み込み演算子の面白い応用例を示そう．リストの結合演算子 $(\oplus)$ を使うと

$$\bigcup_{[\,]}^{\oplus} [[0,1,2],[3,4,5],\cdots] = [0,1,2,3,4,5,\cdots] \tag{5.15}$$

であるから，演算子 $\bigcup_{[\,]}^{\oplus}$ はリストを平坦化する **平坦化演算子** である．平坦化演算子は concat 演算子とも呼ばれることもあるが，基本的な演算子であるため特別な記号をつけておこう．我々は

$$\flat = \bigcup_{[\,]}^{\oplus} \tag{5.16}$$

と定義することにする．[*13]

[*11] Haskell はポイントフリースタイルをサポートする．
[*12] Haskell では $\bigsqcup_a^* x^\star$ を `foldr (*) a xs` と書く．
[*13] Haskell では演算子 $\flat$ の代わりに `concat` 関数を使う．

5.3 マップ

リストの各要素に決まった関数を適用したい場合がある．Python ではリスト `ls` に関数 `f` を適用するときには

Python
```
map(f, ls)
```

のように `map` 関数を用いる．例えば

Python
```
f = lambda x: 100+x
ls = [1, 2, 3, 4, 5]
ms = map(f, ls)
```

とすると，結果として `ms` には $[101, 102, 103, 104, 105]$ が入る．

このように引数として関数 f とリスト $[x_0, x_1 \cdots x_n]$ を取り，戻り値として $[fx_0, fx_1 \cdots fx_n]$ を返す演算子 $\odot$ を考えよう．このとき

$$f \odot [x_0, x_1 \cdots x_n] = [fx_0, fx_1 \cdots fx_n] \tag{5.17}$$

であると定義する．この演算子 $\odot$ をリストの **マップ演算子** と呼ぶ．[*14]

リストのマップ演算子の型は

$$\odot :: (\mathbf{a} \mapsto \mathbf{b}) \mapsto [\mathbf{a}] \mapsto [\mathbf{b}] \tag{5.18}$$

である．矢印 $\mapsto$ は右結合なので，これは

$$\odot :: (\mathbf{a} \mapsto \mathbf{b}) \mapsto ([\mathbf{a}] \mapsto [\mathbf{b}]) \tag{5.19}$$

の意味でもある．念のため上式に注釈を加えると

$$\odot :: \underbrace{(\mathbf{a} \mapsto \mathbf{b})}_{f} \mapsto \underbrace{[\mathbf{a}]}_{[x_0, x_1 \cdots x_n]} \mapsto \underbrace{[\mathbf{b}]}_{[fx_0, fx_1 \cdots fx_n]} \tag{5.20}$$

である．

ここで f と $f \odot$ の型を並べてみると

$$f :: \mathbf{a} \mapsto \mathbf{b} \tag{5.21}$$
$$f \odot :: [\mathbf{a}] \mapsto [\mathbf{b}] \tag{5.22}$$

となり，マップ演算子が何をしているのか一目瞭然になる．

[*14] Haskell では $f \odot x^\star$ を `map f xs` または `f <$> xs` と書く．ただし演算子 `<$>` は `fmap` 演算子の中置バージョンである．

具体例を見てみよう．先程の Python コードの例にあわせて

$$f = \backslash x \to 100 + x \tag{5.23}$$
$$x^\star = [1, 2, 3, 4, 5] \tag{5.24}$$
$$y^\star = f \bullet x^\star \tag{5.25}$$

とすると $y^\star$ の値は $[101, 102, 103, 104, 105]$ となる．

5.4　余計な話：リストの実装

ここでリストの実装について述べておこう．紙上ではリストは自由に考えられるが，計算機上ではそれほど自由ではないからである．我々はリストを LISP におけるリストと同じ構造を持つものとする．LISP におけるリストとは変数 first と変数 rest からなるペアの集合である．変数 first がリストの要素を参照し，変数 rest が次のペアを参照する．リストの最後のペアの rest は空リストを参照する特別な値を持つ．

リストのための特別な表現

$$\text{first} : \text{rest} \tag{5.26}$$

を用い，リファレンス first はリストが保持する型，リファレンス rest はリスト型であるとする．演算子 : を **結合演算子** と呼ぶ．[*15]

要素 rest はリストまたは空リストであるから，一般にリストは次のように展開できることになる．

$$[x_0, x_1, x_2 \cdots x_n] = x_0 : [x_1, x_2 \cdots x_n] \tag{5.27}$$
$$= x_0 : x_1 : [x_2 \cdots x_n] \tag{5.28}$$
$$= x_0 : x_1 : x_2 : \cdots : x_n : [\,] \tag{5.29}$$

結合演算子 (:) は右結合する．すなわち $x_0 : x_1 : x_2 = x_0 : (x_1 : x_2)$ である．

マップ演算子の実装は，リストの実装に踏み込めば簡単である．空でないリストは必ず $x : x^\star$ へと分解できるから

$$\begin{cases} f \odot (x : x^\star) = (fx) : (f \odot x^\star) \\ f \odot [\,] = [\,] \end{cases} \tag{5.30}$$

とマップ演算子 ($\odot$) を定義できる．つまりマップ演算子 ($\odot$) は結合演算子 (:) から作ることができる．換言すれば，マップ演算子はシンタックスシュガーである．

[*15] Haskell でも要素 `x` をリスト `xs` の先頭に追加することを `x:xs` と書く．

Haskell では任意のリスト $x^\star$ に対し，次の関数が用意されている．

$\text{head}\, x^\star \ldots x^\star$ の先頭要素
$\text{tail}\, x^\star \ldots x^\star$ の 2 番目以降の要素からなるリスト

これらは LISP の `car` 関数，`cdr` 関数と同じものであり，この二者を用いればどのようなリストの処理も可能である．

このように基本的な関数から高機能な関数を実装する方法はよく行われる．この例では結合演算子からマップ演算子を合成した．

なお，リストを引数にとる関数はいつでも

$$f(x : x^\star) = \cdots \tag{5.31}$$

という風にパタンマッチを行えるが，式の右辺でリスト全体すなわち $(x : x^\star)$ を参照したい場合もあるであろう．そのような場合は

$$fy^\star @ (x : x^\star) = \cdots \tag{5.32}$$

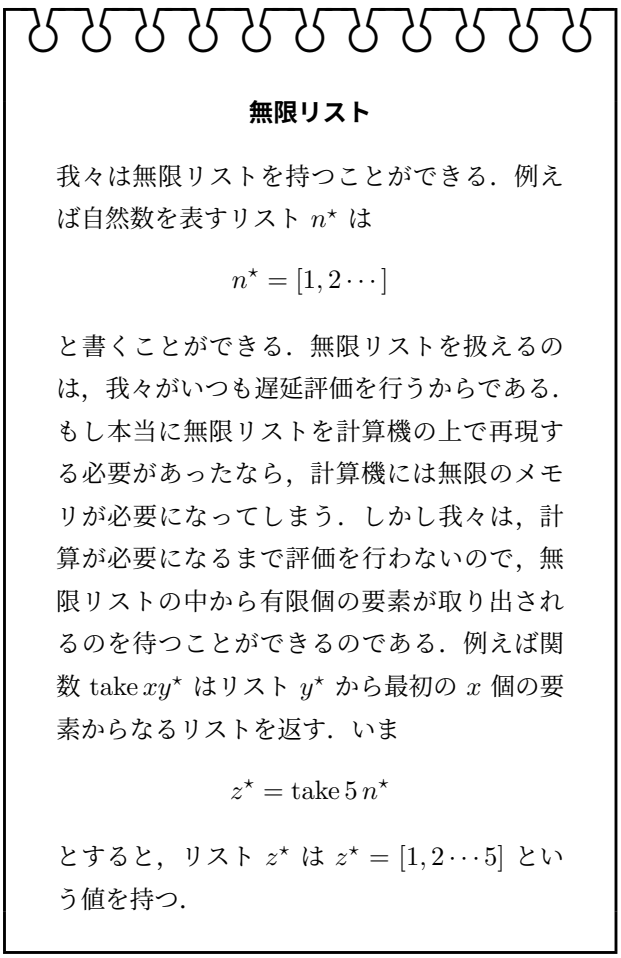
として，変数 y でリスト全体を参照することも可能である．このような記法を **as パタン** と呼ぶ．[*16]

5.5　この章のまとめ

1. 配列はリストである．配列は $[x_0, x_1, x_2]$ のように表記する．
2. 等差数列の配列は $[0, 1 \cdots 10]$ のように途中を $\cdots$ で省略できる．
3. 配列は無限長であってもよい．無限長配列は $[0, 1 \cdots]$ のように表記する．
4. 1 要素のリストは $[x]$ のように表記する．この括弧をリスト値コンストラクタと呼ぶ．
5. 空リストは $[\,]$ と表す．
6. **a** 型のリストを [**a**] 型で表す．
7. リスト型コンストラクタ *List* は型パラメタ **a** に作用して [**a**] を生成する．
8. リストは先頭要素と続くリストから定義される．先頭要素を x とし，続くリストを $x^\star$ とすると $x : x^\star$ はリストである．ここに演算子 : は結合 (cons) 演算子である．

[*16] Haskell では $fy^\star @ (x : x^\star)$ を `f ys@(x:xs)` と書く．

9. リストはリスト結合 (append) できる．リスト $y^\star$ とリスト $z^\star$ のリスト結合は $y^\star \oplus z^\star$ である．
10. 左畳み込み演算子 $\bigcup_a^\bigstar [x_0, x_1 \cdots x_n]$ は $a \bigstar x_0 \bigstar x_1 \bigstar \cdots \bigstar x_n$ を返す．
11. 右畳み込み演算子 $\bigsqcup_a^\bigstar [x_0, x_1 \cdots x_n]$ は $(x_0 \bigstar (x_1 \bigstar \cdots \bigstar (x_n \bigstar a)))$ を返す．
12. 平坦化演算子 $\flat = \bigcup_{[]}^\oplus$ はリストを平坦化する．
13. 関数 $f :: \mathbf{a} \mapsto \mathbf{a}$ はマップ演算子 $\odot$ を用いて $f \odot [x_0, x_1 \cdots x_n] = [fx_0, fx_1 \cdots fx_n]$ のようにリスト $[x_0, x_1 \cdots x_n]$ に適用できる．
14. マップ演算子 $\odot$ の型は $(\mathbf{a} \mapsto \mathbf{b}) \mapsto \mathbf{a} \mapsto \mathbf{b}$ である．
15. $\mathrm{head}(x : x^\star) = x, \mathrm{tail}(x : x^\star) = x^\star$ である．



無限リスト

我々は無限リストを持つことができる．例えば自然数を表すリスト $n^\star$ は

$$n^\star = [1, 2 \cdots]$$

と書くことができる．無限リストを扱えるのは，我々がいつも遅延評価を行うからである．もし本当に無限リストを計算機の上で再現する必要があったなら，計算機には無限のメモリが必要になってしまう．しかし我々は，計算が必要になるまで評価を行わないので，無限リストの中から有限個の要素が取り出されるのを待つことができるのである．例えば関数 $\mathrm{take}\, x y^\star$ はリスト $y^\star$ から最初の x 個の要素からなるリストを返す．いま

$$z^\star = \mathrm{take}\, 5\, n^\star$$

とすると，リスト $z^\star$ は $z^\star = [1, 2 \cdots 5]$ という値を持つ．

第 6 章

再帰*

A...

6.1 関数の再帰適用

関数は内部で自分自身を適用しても良い．例えば x の階乗 $(x!)$ を返す関数 fact は

$$\mathrm{fact}\, x = \begin{cases} 1 & \mathsf{if}\ x \equiv 0 \\ x * \mathrm{fact}(x-1) & \mathsf{otherwise} \end{cases} \tag{6.1}$$

と定義できる．関数が自分自身を適用することを関数の **再帰適用** と呼ぶ．

これで我々は関数の適用，変数の代入，ラムダ式，条件式，再帰の方法を学んだわけである．これだけあれば，原理的にはどのようなアルゴリズムも書くことができる．今日からはカリー風な数学であらゆるアルゴリズムを表現できるのである！

—

結合演算子 (:) は関数引数のパタンにも使える．これは，例えばリストの和をとる関数 sum は

$$\begin{cases} \mathrm{sum}\,[\,] = 0 \\ \mathrm{sum}(x : x^\star) = x + \mathrm{sum}\, x^\star \end{cases} \tag{6.2}$$

のようにも定義できるということである．

なお，関数は再帰させるたびに計算機のスタックメモリを消費する．これを回避するためのテクニックが，次節で述べる末尾再帰である．

6.2 末尾再帰

計算機科学者は，同じ再帰でも **末尾再帰** という再帰のスタイルを好む．末尾再帰とは，関数の再帰適用を関数定義の末尾にすることである．この章に出てきた階乗関数 fact を例にとろう．階乗関数 fact は

$$\mathrm{fact}\, x = \begin{cases} 1 & \mathsf{if}\ x \equiv 0 \\ x * \mathrm{fact}(x-1) & \mathsf{otherwise} \end{cases} \tag{6.3}$$

のような形をしていた．末尾の関数をよりはっきりさせるために演算子 $(*)$ を前置にして

$$\mathrm{fact}\, x = \begin{cases} 1 & \mathsf{if}\ x \equiv 0 \\ (*)x(\mathrm{fact}(x-1)) & \mathsf{otherwise} \end{cases} \tag{6.4}$$

と書いてみよう．この定義の末尾の式は

$$(*)x(\mathrm{fact}(x-1)) \tag{6.5}$$

である．これだと末尾の関数は fact ではなく演算子 $(*)$ なので，末尾に再帰適用を行ったことにはならない．

そこで，次のように形を変えた階乗関数 fact' を考えてみる．

$$\mathrm{fact}'\, a = \begin{cases} 1 & \mathsf{if}\ x \equiv 0 \\ \mathrm{fact}'(a * x)(x-1) & \mathsf{otherwise} \end{cases} \tag{6.6}$$

こうすれば末尾の関数がもとの fact' と一致する．

関数 fact と違い関数 fact' は引数を 2 個とる．関数 fact' を使って x の階乗を求める場合は $\mathrm{fact}'\, 1x$ と第 1 引数に 1 を与えることにする．この第 1 引数 a はアキュムレータという．アキュムレータが演算の途中経過を引き渡していくイメージを描けば，末尾再帰の意味が理解できるだろう．

計算機科学者が末尾再帰を好む理由は，Haskell を含む幾つかのプログラミング言語処理系が **末尾再帰最適化** を行うからである．末尾再帰最適化とは，一言で言うと再帰を計算機が扱いやすいループに置き換えることである．では最初から我々もループで関数を表現しておけば，と思われるかもしれないが，再帰以外の方法でルー

プを表現する場合には必ず変数（ループカウンタ）への破壊的代入が必要になるため，我々は末尾再帰に慎ましくループを隠すのである．

6.3 アルゴリズムの本質*

逐次実行，条件分岐，繰り返し．

…

6.4 余計な話：遅延評価*

Haskell は，意図しない限り遅延評価を行う．これは特に左畳み込み演算子 ($\bigcup$) を使う場合に問題となる．いま $x^\star = [x_0, x_1, x_2, x_3]$ とすると，左畳み込み演算 $\bigcup_0^+ x^\star$ は

$$\bigcup_0^+ x^\star = \bigcup_0^+ (x_0 : x_1 : x_2 : [\,]) \tag{6.7}$$

$$= \bigcup_{0+x_0}^+ (x_1 : x_2 : x_3 : [\,]) \tag{6.8}$$

$$= \bigcup_{(0+x_0)+x_1}^+ (x_2 : x_3 : [\,]) \tag{6.9}$$

$$= \bigcup_{((0+x_0)+x_1)+x_2}^+ (x_3 : [\,]) \tag{6.10}$$

$$= \bigcup_{(((0+x_0)+x_1)+x_2)+x_3}^+ [\,] \tag{6.11}$$

$$= (((0 + x_0) + x_1) + x_2) + x_3 \tag{6.12}$$

と展開される．遅延評価のために，Haskell 処理系は値ではなく式をメモリにストアしなければならないが，左畳み込み演算は大きなメモリを必要としがちである．もし例えば予め $0 + x_0$ を計算しておくなど左畳み込みだけ先に評価しておけば，大いにメモリの節約になる．そのために Haskell は「遅延評価無し」の左畳み込み演算子を用意している．[*1]

6.5 この章のまとめ

1. 関数は再帰適用できる．ループは再帰適用によって実現する．
2. 末尾再帰はスタックを消費しないように最適化される．
3. 関数はいつも遅延評価される．そのため無限リストを扱うことも可能である．

クイックソート

我々の記法を使うと，クイックソートは次のように定義できる．

$$\begin{cases} \mathrm{srt}\,[\,] = [\,] \\ \mathrm{srt}(x : x^\star) = (\mathrm{srt}\,a^\star) \oplus [x] \oplus (\mathrm{srt}\,b^\star) \\ \qquad \text{where} \\ \qquad\quad a^\star \triangleq [a \,|_{a \in x^\star, a \leq x}] \\ \qquad\quad b^\star \triangleq [b \,|_{b \in x^\star, b > x}] \end{cases}$$

Haskell では

Haskell
```
srt :: [a] -> [a]

srt[]     = []
srt(x:xs) = srt as ++ [x] ++ srt bs
              where
                as = [a| a<-xs, a<=x]
                bs = [b| b<-xs, b>x]
```

と書く．このコードはしばしば Haskell のパワーを示すために紹介される．しかし，クイックソートのピボットとして常にリストの先頭要素を用いているため，必ずしも良いコードではない．

[*1] 「遅延評価無し」の左畳み込み演算子を Haskell では `foldl'` と書く．

第 7 章

Maybe*

この章では計算結果が正しいかもしれないし，正しくないかもしれないという曖昧な状況を表す型を導入する．手始めに Python でクラス `Possibly` を実装し，それがカリー風の数式で綺麗に書けることを示す．またリストとの共通点についても見ていくことにする．

7.1 Possibly

計算の途中で，計算にまつわる状態を残りの計算に引き継ぎたくなる場合がある．例えば，整数 x, y, z があり $x = y/z$ なる値を続く計算で利用したいとする．だが $z \equiv 0$ のときには x は正しく計算されない．こんなときプログラマが取れる手段は

- $x = y/z$ を計算した時点で **ゼロ除算例外** を発生させ，プログラムの制御を他の場所へ移す（大域ジャンプを行う）
- グローバル変数にゼロ除算エラーが起こったことを記録しておき，x にはとりあえずの数値，例えば 0 を代入しておいて，計算を続行させる
- x にエラー状態を示す印を新たにつけておいて，計算を続行させる

といったところだろう．

大域ジャンプも，グローバル変数の書き換えも破壊的代入を伴うものであり，受け入れがたい．そこで我々は第三のエラー状態を示す印をつける方法を採用することにする．普通変数が整数だろうが実数だろうが，計算機表現には余分なビットが残っていないので，変数をラップする次のようなクラス `Possibly` を導入することにしよう．メンバ変数 `value` が値を，メンバ変数 `valid` がエラーの有無を表す．

Python
```
class Possibly:
  def __init__(self, a_valid, a_value = 0):
    self.valid = a_valid
    self.value = a_value
```

例えば整数値 123 を持つ `Possibly` クラスの値 `p` は

Python
```
p = Possibly(True, 123)
```

として生成できるし，`Possibly` 値 `p` が計算エラーを表す場合は

Python
```
p = Possibly(False)
```

と初期化できる．

ここで，引数に 1 を加えて返す関数 `f` があるとしよう．関数 `f` の定義は次の通りである．

Python
```
f = lambda x: 1+x
```

関数 `f` に直接 `Possibly` 値 `p` を食わせるとランタイムエラーを引き起こす．

Python
```
q = f(p) # エラー!!
```

これは関数 `f` が引数として数値を期待していたにもかかわらず，`Possibly` クラスの値が渡されたからである．もし関数 `f` のほうをいじりたくないとすれば，次のような関数 `map_over` を使って

Python
```
q = map_over(f, p)
```

というふうに間接的に関数適用を行う必要がある．

関数 map_over(f, p) はもし p がエラーを表す値でなければ中身の値を関数 f に適用し，その結果を Possibly クラスに包んで返す．もし p がエラー値を表す値であれば，結果もエラー値である．関数 map_over の実装は次のようになる．

Python
```
def map_over(f, p):
  if p.valid == True:
    return Possibly(True, f(p.value))
  else:
    return Possibly(False)
```

さて，次節では以上のようなことを抽象数学的に綺麗に描いてみよう．

7.2 Maybe コンテナ

もう一度振り出しに戻る．

整数 x, y, z があり $x = y/z$ という式があるとする．この式は $z \equiv 0$ のときにはゼロ除算エラーである．しかし「例外」は内部状態の書き換えであり，我々の計算に入れたくない．そこで変数 x が正しく計算されたかもしれないし，されていないかもしれないということを $u^?$ のように肩に?をつけた変数に入れて，忘れないようにしておこう．

ここで変数 $u^?$ が取り得る値は正しく計算された値 x をラップしたものか，あるいはエラーを表す値 $\emptyset$ である．このように計算結果に「意味付け」をすることを **文脈** に入れると言う．定数 $\emptyset$ は「ナッシング」と呼ぶ．

この変数 $u^?$ はもはや整数 (**Int**) 型とは言えない．そこでこの $u^?$ の型を $^{\mathbf{Maybe}}[\![\mathbf{Int}]\!]$ と表して「Maybe 整数 (おそらく整数)」型と呼ぶことにしよう．型 **a** から型 $^{\mathbf{Maybe}}[\![\mathbf{a}]\!]$ を生成するこには **型コンストラクタ** *Maybe* を用いて

$$^{\mathbf{Maybe}}[\![\mathbf{a}]\!] = \mathit{Maybe}\,\mathbf{a} \tag{7.1}$$

とする．[*1]

a 型の変数を $^{\mathbf{Maybe}}[\![\mathbf{a}]\!]$ 型の変数に代入するには，次の値コンストラクタを用いて

$$u^? = {}^{\mathrm{Just}}[\![x]\!] \tag{7.2}$$

と書く．[*2]

変数 x が一度ゼロ除算の危険性に「汚染」された場合，その後ずっと Maybe 変数に入れ続けなければいけない．そこで，普通の変数を引数にとる関数 f に Maybe 変数 $u^?$ を食わせるには，リストの時と同じようなマップ演算子が必要になる．具体的には，変数 x が **Int** 型として，Maybe 変数 $u^? = {}^{\mathrm{Just}}[\![x]\!]$ が与えられたとき

$$f \boxdot u^? = {}^{\mathrm{Just}}[\![fx]\!] \tag{7.3}$$

となるような Maybe バージョンのマップ演算子 $\boxdot$ を用いる．ここに $f \boxdot u^?$ の型は，もし $f :: \mathbf{Int} \mapsto \mathbf{Float}$ ならば $^{\mathbf{Maybe}}[\![\mathbf{Float}]\!]$ である．

実際には $u^? \equiv \emptyset$ の可能性も考えなければならないから，Maybe バージョンのマップ演算子は

$$f \boxdot u^? = \begin{cases} {}^{\mathrm{Just}}[\![fx]\!] & \text{if } u^? \equiv {}^{\mathrm{Just}}[\![x]\!] \\ \emptyset & \text{otherwise} \end{cases} \tag{7.4}$$

でなければならない．この Maybe バージョンのマップ演算子 $\boxdot$ は

$$\begin{cases} f \boxdot {}^{\mathrm{Just}}[\![x]\!] = {}^{\mathrm{Just}}[\![fx]\!] \\ f \boxdot \emptyset = \emptyset \end{cases} \tag{7.5}$$

と定義すれば得られる．

今後，普通の（引数に Maybe が来ることを想定していない）関数 f を Maybe 型である変数 $u^?$ に適用させるときには，必ず

$$v^? = f \boxdot u^? \tag{7.6}$$

のように Maybe バージョンのマップ演算子 $\boxdot$ を用いることにする．これはプログラムの安全性のためである．変数が一旦ゼロ除算の可能性に汚染されたら，最後まで Maybe に包んでおかねばならない．[*3]

Python で Maybe の概念を忠実になぞることは難しい．と言うのも Python は動的型付け言語であるため，

[*1] Haskell では表記上コンテナ型 $^{\mathbf{Maybe}}[\![\mathbf{a}]\!]$ と型コンストラクタ式 $\mathit{Maybe}\,\mathbf{a}$ を区別せず，両者とも Maybe a と書く．

[*2] Haskell では $u^? = {}^{\mathrm{Just}}[\![x]\!]$ を u = Just x と書く．Maybe を表す疑問符は省略する．

[*3] Haskell では Maybe バージョンのマップ演算子に特別な記号，関数名が与えられていない．その代わり第 8 章で述べる一般マップ演算子 • に相当する fmap 関数を用い v = fmap f u または v = f <$> u のように書く．Haskell は型推論を行なうため変数 u が Maybe であれば Maybe バージョンのマップ演算子（関数）が適用され，もし u がリストであれば通常のマップ関数である map が適用される．

型コンストラクタという概念が無いからだ．一方で Maybe の概念を静的型付け言語である C++ や Java で実現することはできる．そこで C++ の本物のコードで示しておこう．ただしポインタを使わないでおいたので C++ プログラマも Java プログラマも参考にできるだろう．

Maybe は次の maybe クラステンプレートで表現できる．（Java プログラマへの注意：これは maybe<a>クラスの定義と同じ意味である．）

C++
```
template <class a> class maybe {
  private:
    a value;
    bool valid;
  public:
    maybe(): value(0), valid(false) { }
    maybe(a a_value):
      value(a_value), valid(true) { }
    a get_value() const { return value; }
    bool is_valid() const { return valid; }
}
```

デフォルトコンストラクタは例外的な状況を表す $\emptyset$ を生成し，1 引数コンストラクタは maybe で包んだ引数値を生成する．

C++ プログラムで良く見かける クラス設計と違い，この maybe クラスはコンストラクタ以外に中身を書き換える手段が提供されていない．これが破壊的代入の禁止が意味することである．

当然我々には Maybe バージョンのマップ演算子が必要である．ここでは関数 map_over として書いてみよう．（Java プログラマへの注意：関数 map_over はどのクラスにも属していないが，それで正解なのである．）

C++
```
template <class a, class b, class fn>
maybe<b> map_over(fn f, maybe<a> u) {
  if (u.is_valid()) {
    return maybe<a>(f(u.get_value()));
  }
  else {
    return maybe<b>();
  }
}
```

テンプレートの 2 番目の引数 fn は関数 f を受け取るために必要である．C++ はコンパイル時までにすべての変数の型が決定していないといけないが，関数 f の型は関数 map_over 設計時には確定できないため，このようにテンプレートにしている．（C++03 プログラマへの注意：関数 f の代わりに関数オブジェクトを渡しても良い．）

整数 x から Maybe 値 $u^? = {}^{\mathrm{Just}}[\![x]\!]$ を作り，関数 $gx = 1 + x$ を Maybe 値 $u^?$ に食わせて Maybe 値 $v^?$ ただし

$$v^? = g \boxdot u^? \tag{7.7}$$

を得ることを C++ では次のように書くことになる．

C++
```
int x = 123;
maybe<int> u(x);
auto g = [](int x) -> int { return 1+x; };
maybe<int> v = map_over(g, u);
```

注意してほしいのは g(x) も map_over(g, u) も正当なコードだが g(u) は型エラーであることだ．また g(u.get_value()) は正当なコードだが，わざわざ u が持つ文脈を捨てることになる．（C++ プログラマへの注意：int 型の使用は現在では非推奨である．本書ではコードの読みやすさのためにあえて int を使用している．）

7.3 リストと Maybe

関数 f を Mayby 値 $u^?$ に適用するために

$$v^? = f \boxdot u^? \tag{7.8}$$

のような Maybe バージョンのマップ演算子 ($\boxdot$) を使った．一方で，同じ関数 f をリスト $x^\star$ に適用するには

$$y^\star = f \odot x^\star \tag{7.9}$$

のようなリストバージョンのマップ演算子 ($\odot$) を使った．

リストバージョンのマップ演算子 ($\odot$) をもし C++ で書くとしたら，次のようなコードになる．ここでリスト型として C++ の標準テンプレートライブラリ (STL) の list クラスを流用した．

C++
```
using namespace std;
template <class a, class b, class fn>
list<b> map_over(fn f, list<a> xs) {
  list<b> ys(xs.size());
  auto i = xs.cbegin();
  auto j = ys.begin();
  while (i != xs.cend()) {
    *j = f(*i);
    ++i; ++j;
  }
  return list<b>(ys);
}
```

この関数 map_over の中身部分はどうでもよろしい．それよりも，リストバージョンのマップ演算子の C++ 関数のインタフェースと，Maybe バージョンのマップ演算子の C++ 関数のインタフェースを見比べてみよう．

C++
```
// List
template <class a, class b, class fn>
list<b> map_over(fn f, list<a> xs);
// Maybe
template <class a, class b, class fn>
maybe<b> map_over(fn f, maybe<a> u);
```

やはりそっくりである．であるならば，うまく統一したい．C++14 以降であれば次のような書き方ができる．

C++
```
template <class a, class b,
  template<class> X, class fn>
X<b> map_over(fn f, X<a> x);
```

これは一見上手く行きそうに見えるが，このコードは map_over のインスタンス化で躓くため，次のように b 型のダミー変数が必要になる．

C++
```
template <class a, class b,
  template<class> X, class fn>
X<b> map_over(fn f, X<a> x, b dummy);
```

残念なことに, いずれのコードにしてもリストと Maybe の本質的な抽象化にはなってない．型 X がマップ可能なコンテナであることをテンプレート機構を使って保証することができないためである．この問題は C++20 で導入予定の「コンセプト」機能によって解決する見込みである．

一方で，数学者たちが見つけた圏という代数的構造が，リストも Maybe も統一的に扱うことを可能にしている．これを発見したのは Eugenio Moggi を始めとする計算機科学者たちである．この人類の英知は第 8 章から見ていくことにしよう．

7.4 余計な話：Either

Maybe とよく似た型に Either がある．Maybe が **a** 型または $\emptyset$ のいずれかの値をとったように，Either は **a** 型または **b** 型のいずれかの値を取る．**a** 型または **b** 型を取る Either 型の変数 $e^!$ があるとすると，

$$e^! :: {}^{\mathbf{Either}}[\![\mathbf{a}, \mathbf{b}]\!] \tag{7.10}$$

と書く．Either 型は型 **a** および **b** から型コンストラクタを用いて

$${}^{\mathbf{Either}}[\![\mathbf{a}, \mathbf{b}]\!] = \mathit{Either}\,\mathbf{ab} \tag{7.11}$$

のように作られる．[*4]

Ether には値コンストラクタが 2 種類あり，それぞれ ${}^{\mathrm{Right}}[\![x]\!]$ と ${}^{\mathrm{Left}}[\![x]\!]$ である．値コンストラクタは

$$e^! = {}^{\mathrm{Right}}[\![x]\!] \tag{7.12}$$

または

$$e^! = {}^{\mathrm{Left}}[\![x]\!] \tag{7.13}$$

のように使う．[*5]

Either はより複雑な計算エラーが発生する場合に用いる．Maybe が単に失敗を表す $\emptyset$ しか表現できなかったのに対し，Either は任意の型の変数で表現できる．習慣的に，正しい (right) 計算結果は ${}^{\mathrm{Right}}[\![x]\!]$ 値コンストラクタで格納し，エラーの情報は ${}^{\mathrm{Left}}[\![x]\!]$ 値コンストラクタで格納する．

Either 型は C の共有型 (union) に近い．

7.5 この章のまとめ

1. ある型 **a** からその Maybe 型 ${}^{\mathbf{Maybe}}[\![\mathbf{a}]\!]$ を作ることを ${}^{\mathbf{Maybe}}[\![\mathbf{a}]\!] = \mathit{Maybe}\,\mathbf{a}$ と書く．ここに *Maybe* は Maybe 型コンストラクタである．

[*4] Haskell では ${}^{\mathbf{Either}}[\![\mathbf{a}, \mathbf{b}]\!]$ も $\mathit{Either}\,\mathbf{ab}$ も区別せずに `Either a b` と書く．

[*5] Haskell ではそれぞれ `e = Right x` および `e = Left x` と書く．

2. ある変数 x から Maybe 変数 $u^?$ を作るには $u^? = {}^{\mathrm{Just}}[\![x]\!]$ とする．ここに ${}^{\mathrm{Just}}[\![\cdots]\!]$ は Maybe 値コンストラクタである．
3. Maybe 変数は ${}^{\mathrm{Just}}[\![x]\!]$ のような値か，かまたは $\emptyset$ なる「ナッシング」値のかどちらかを持つことができる．
4. 普通の関数 $f :: \mathbf{a} \mapsto \mathbf{b}$ を Maybe 値に適用することはできない．関数 f を Maybe 値に適用するには $f \boxdot u^?$ のように Maybe マップ演算子が必要であり，この関数適用の結果は ${}^{\mathbf{Maybe}}[\![\mathbf{b}]\!]$ 型である．
5. Either 変数は二つの型のいずれかを持つことができ，$e^! = {}^{\mathrm{Right}}[\![x]\!]$ または $e^! = {}^{\mathrm{Left}}[\![x]\!]$ のように生成する．

テンプレートとジェネリック

…

第 8 章

関手*

A...

8.1 圏と関手*

型 $\mathbf{a}$ の変数 $x, y :: \mathbf{a}$ について，関数 $f :: \mathbf{a} \mapsto \mathbf{a}$ があり

$$y = fx \tag{8.1}$$

であるとしよう．このように型 $\mathbf{a}$ で閉じた世界を仮に $\mathbf{a}$ 世界と呼ぶことにする．

型 $^{\mathbf{Maybe}}[\![\mathbf{a}]\!]$ の変数 $u^?, v^? :: {}^{\mathbf{Maybe}}[\![\mathbf{a}]\!]$ について，関数 $g :: {}^{\mathbf{Maybe}}[\![\mathbf{a}]\!] \mapsto {}^{\mathbf{Maybe}}[\![\mathbf{a}]\!]$ があり

$$v^? = gu^? \tag{8.2}$$

であるとしよう．このように $^{\mathbf{Maybe}}[\![\mathbf{a}]\!]$ で閉じた世界を仮に $^{\mathbf{Maybe}}[\![\mathbf{a}]\!]$ 世界と呼ぶことにする．

ここで，変数 x, y と Maybe 変数 $u^?, v^?$ は Maybe 値コンストラクタによって

$$u^? = {}^{\text{Just}}[\![x]\!] \tag{8.3}$$
$$v^? = {}^{\text{Just}}[\![y]\!] \tag{8.4}$$

の関係にあるとしよう．値コンストラクタは値を $\mathbf{a}$ 世界から $^{\mathbf{Maybe}}[\![\mathbf{a}]\!]$ 世界へとジャンプさせる機能を持っている．

他に $\mathbf{a}$ 世界から $^{\mathbf{Maybe}}[\![\mathbf{a}]\!]$ 世界へジャンプさせるものがあるだろうか．よく考えてみると，マップ演算子もそうである．いま $u^? = {}^{\text{Just}}[\![x]\!], v^? = {}^{\text{Just}}[\![y]\!]$ なのだから，$\mathbf{a}$ 世界の関数 f と $^{\mathbf{Maybe}}[\![\mathbf{a}]\!]$ 世界の関数 g は無関係ではなく

$$v^? = gu^? = f \boxdot u^? \tag{8.5}$$

であり，

$$g = f \boxdot \tag{8.6}$$

である．つまりマップ演算子 $\boxdot$ が関数 f を $\mathbf{a}$ 世界から $^{\mathbf{Maybe}}[\![\mathbf{a}]\!]$ 世界へとジャンプさせているのである．

いま「世界」と呼んだものを，数学者は **圏** と呼ぶ．圏とは **対象** と **射** の組み合わせである．本書では「対象」とは型のことであり，射とは関数だと思えば良い．コンテナに入れられた関数も射に含まれる．そして，圏から圏へとジャンプさせるものを **関手** と呼ぶ．この例で言えば値コンストラクタ $^{\text{Just}}[\![x]\!]$ とマップ演算子 $\boxdot$ が関手である．値コンストラクタ $^{\text{Just}}[\![x]\!]$ は $\mathbf{a} \mapsto {}^{\mathbf{Maybe}}[\![\mathbf{a}]\!]$ という型を持ち，マップ演算子 $\boxdot$ は $(\mathbf{a} \mapsto \mathbf{b}) \mapsto ({}^{\mathbf{Maybe}}[\![\mathbf{a}]\!] \mapsto {}^{\mathbf{Maybe}}[\![\mathbf{b}]\!])$ という型を持つ．

同じことはリストにも言える．値コンストラクタ $[x]$ とマップ演算子 $\odot$ もまた関手である．この場合値コンストラクタは $\mathbf{a} \mapsto [\mathbf{a}]$ という型を持ち，マップ演算子も同じく $(\mathbf{a} \mapsto \mathbf{b}) \mapsto ([\mathbf{a}] \mapsto [\mathbf{b}])$ という型を持つ．

✠

Haskell ではマップ演算子が定義された型を関手と呼ぶ．具体的には，マップ演算子が定義された全ての型は Functor 型クラスのインスタンスであるとする．つまり，Functor 型クラスには一般化されたマップ演算子が定義されており，そのインスタンスであるリストや Maybe は独自のマップ演算子を定義しなければならないということである．

一般化されたマップ演算子を $\bullet$ で表そう．この $\bullet$ 演算子は

$$(\bullet) :: (\text{Functor} \supset fn) \Rightarrow (\mathbf{a} \mapsto \mathbf{b}) \mapsto {}^{\mathbf{fn}}[\![\mathbf{a}]\!] \mapsto {}^{\mathbf{fn}}[\![\mathbf{b}]\!] \tag{8.7}$$

という型を持つ．もし型コンストラクタがリスト型コンストラクタであれば

$$(\odot) :: (\mathbf{a} \mapsto \mathbf{b}) \mapsto [\mathbf{a}] \mapsto [\mathbf{b}] \tag{8.8}$$

であるし，もし型コンストラクタが Maybe 型コンスト

ラクタであれば

$$(\boxdot) :: (\mathbf{a} \mapsto \mathbf{b}) \mapsto {}^{\mathbf{Maybe}}\,[\![\mathbf{a}]\!] \mapsto {}^{\mathbf{Maybe}}\,[\![\mathbf{b}]\!] \tag{8.9}$$

である．

...

8.2 アプリカティブ関手

マップ演算子をさらに汎用性のあるものにするために，**アプリカティブマップ演算子** という演算子を考えてみる．

いま関数のリスト $[f, g, h]$ と変数のリスト $[x, y, z]$ があるとする．リストバージョンのアプリカティブマップ演算子 $\otimes$ を次のように定義する．

$$[f, g, h] \otimes [x, y, z] = [fx, fy, fz, gx, gy, gz, hx, hy, hz] \tag{8.10}$$

リストバージョンのアプリカティブマップ演算子はこのように，左引数のリスト内のすべての関数を順番に右引数のリスト内の変数に適用し，その結果をリストとして返す．

リストバージョンのアプリ化ティブマップ演算子 $\otimes$ の型は $[\mathbf{a} \mapsto \mathbf{b}] \mapsto [\mathbf{a}] \mapsto [\mathbf{b}]$ である．これは

$$\otimes :: \underbrace{[\mathbf{a} \mapsto \mathbf{b}]}_{[f_0, f_1 \cdots f_n]} \mapsto \underbrace{[\mathbf{a}]}_{[x_0, x_1 \cdots x_n]} \mapsto \underbrace{[\mathbf{b}]}_{[f_0x_0, f_0x_1 \cdots f_0x_n, f_1x_0, f_1x_1 \cdots f_nx_n]} \tag{8.11}$$

と解釈すれば良い．

リストバージョンのアプリカティブマップ演算子 $(\otimes)$ の特別な場合として，左引数のリストの要素数が 1 の場合を考えると

$$[f] \otimes [x, y, z] = [fx, fy, fz] \tag{8.12}$$

であり，通常のマップ演算子 $(\odot)$ を使ったマップすなわち

$$f \odot [x, y, z] = [fx, fy, fz] \tag{8.13}$$

と右辺が一致する．つまり，マップ演算子はアプリカティブマップ演算子の特別な場合と考えることができる．実際，リストバージョンのアプリカティブマップ演算子は

$$f \odot x^\star = [f] \otimes x^\star \tag{8.14}$$

と定義できる．

Maybe バージョンについても考えてみよう．Maybe に包まれた関数 $i^?$ を Maybe な変数 $u^?$ にマップするアプリカティブマップ演算子 $\boxtimes$ を

$$i^? \boxtimes u^? = \begin{cases} j \boxdot u^? & \text{if } i \equiv {}^{\text{Just}}\,[\![j]\!] \\ \emptyset & \text{otherwise} \end{cases} \tag{8.15}$$

で定義する．この Maybe バージョンのアプリカティブマップ演算子 $(\boxtimes)$ から Maybe バージョンのマップ演算子 $(\boxdot)$ は

$$f \boxdot u^? = {}^{\text{Just}}\,[\![f]\!] \boxtimes u^? \tag{8.16}$$

のように導出できる．

これらの関係を一般化して

$$f \bullet w = \langle f \rangle \times w \tag{8.17}$$

となるような **一般アプリカティブマップ演算子** $(\times)$ を考える．ここに f は関数，w はリストや Maybe といったコンテナ型の変数すなわち **コンテナ変数** である．一般アプリカティブマップ演算子 $(\times)$ から一般マップ演算子 $(\bullet)$ を導き出すには，式 (8.17) のように値コンストラクタが必要である．この一般化された値コンストラクタを **ピュア演算子** と呼ぶ．アプリカティブマップ演算子とピュア演算子を持つ型クラスを **アプリカティブ関手** と呼び，Applicative 型クラスと定義する．[*1]

ピュア演算子をピュア値コンストラクタと呼ばないのは，単純に「ピュア値」というものがないからである．Functor 型クラスはリスト型や Maybe 型を抽象化したものであって，直接変数を生成できない．型クラスは，C++ の用語で言えば純粋仮想クラスのようなものであるし，Objective-C の用語で言えばメタクラスであるからである．もちろんリストのピュア演算子は $[x]$ であるし，Maybe のピュア演算子は ${}^{\text{Just}}\,[\![x]\!]$ であり，それぞれ具体的な変数を生成する．しかし変数 x にピュア演算子を適用した $\langle x \rangle$ は抽象的な概念であり，そのような変数は実在しない．[*2]

一般アプリカティブマップ演算子 $(\times)$ は多様性によってそれぞれリストバージョンのアプリカティブマッ

[*1] Haskell では一般アプリカティブマップ演算子を <*> と書く．

[*2] Haskell は一般のピュア演算子の実装を与えていない．変数の型に応じて対応する関数が適用される．

表 8.1 リストと Maybe の関係

型の名前	型	型コンストラクタ	マップ	値コンストラクタ
リスト	$[\mathbf{a}]$	*List*	$\odot$	$[x]$
Maybe	$^{\mathbf{Maybe}}[\![\mathbf{a}]\!]$	*Maybe*	$\boxdot$	$^{\text{Just}}[\![x]\!]$

プ演算子 ($\otimes$) や Maybe バージョンのアプリカティブマップ演算子 ($\boxtimes$) にオーバーライドされ，それぞれリスト値コンストラクタ ($[x]$), Maybe 値コンストラクタ ($^{\text{Just}}[\![x]\!]$) を用いることでリストバージョンのマップ演算子 ($\odot$), Maybe バージョンのマップ演算子 ($\boxdot$) を生成することができる．リスト値コンストラクタ，Maybe 値コンストラクタはそれぞれピュア演算子 ($\langle x\rangle$) をオーバーライドしたものであるから，結局，一般アプリカティブマップ演算子とピュア演算子のふたつがあれば，任意のクラスのマップ演算子を生成することができる．

この節の最後に **アプリカティブスタイル** という記法を紹介しておこう．アプリカティブマップ演算子は連続して

$$\langle f\rangle \times u \times v \tag{8.18}$$

のように使える．もし $u \equiv \varnothing$ もしくは $v \equiv \varnothing$ であれば式の値は $\varnothing$ になる．式 (8.18) からピュア演算子を消すには，最初のアプリカティブマップ演算子をマップ演算子に置き換えて

$$w = f \bullet u \times v \tag{8.19}$$

とすれば良い．このようにアプリカティブマップ演算子を並べる書き方をアプリカティブスタイルと呼ぶ．

8.3 関手としての関数*

関数は関手である．関手とはマップ演算子を持つ型クラスのことであった．そこで，関数がどのようなマップ演算子を持つのか考えてみる．

ま関数 f が

$$f :: \mathbf{r} \mapsto \mathbf{a} \tag{8.20}$$

という型を持っているとする．全く形式的に，$Func_{\mathbf{r}}$ なる型コンストラクタがあるとして

$$\mathbf{r} \mapsto \mathbf{a} = Func_{\mathbf{r}}\,\mathbf{a} \tag{8.21}$$

であると考えてみる．型コンストラクタによって型 $(\mathbf{r} \mapsto \mathbf{a})$ が作られると考えるのだ．

マップ演算子の型は

$$(\bullet) :: (\textsc{Functor} \supset fn) \Rightarrow (\mathbf{a} \mapsto \mathbf{b}) \mapsto {}^{\mathbf{fn}}[\![\mathbf{a}]\!] \mapsto {}^{\mathbf{fn}}[\![\mathbf{b}]\!] \tag{8.22}$$

であって，いま $fn = Func_{\mathbf{r}}$ を考えているから，関数のマップ演算子を $\circ$ とすると

$$(\circ) :: (\mathbf{a} \mapsto \mathbf{b}) \mapsto Func_{\mathbf{r}}\,\mathbf{a} \mapsto Func_{\mathbf{r}}\,\mathbf{b} \tag{8.23}$$

であり，これはすなわち

$$(\circ) :: (\mathbf{a} \mapsto \mathbf{b}) \mapsto (\mathbf{r} \mapsto \mathbf{a}) \mapsto (\mathbf{r} \mapsto \mathbf{b}) \tag{8.24}$$

のことである．

いま関数 $f :: \mathbf{r} \mapsto \mathbf{a}$ とは別な関数 $g :: \mathbf{a} \mapsto \mathbf{b}$ があったとしよう．関数 f と関数 g の合成 $g \centerdot f$ の型は

$$g \centerdot f :: \mathbf{r} \mapsto \mathbf{b} \tag{8.25}$$

であるから，

$$(\centerdot) :: (\mathbf{a} \mapsto \mathbf{b}) \mapsto (\mathbf{r} \mapsto \mathbf{a}) \mapsto (\mathbf{r} \mapsto \mathbf{b}) \tag{8.26}$$

である．つまり関数のマップ演算子 ($\circ$) と関数の合成演算子 ($\centerdot$) は同じ型を持つ．

幸い，我々は関数のマップ演算子の実装に関しては，型さえ守っていれば自由に選べる．そこで

$$g \circ f = g \centerdot f \tag{8.27}$$

としておこう．これは

$$g \circ f = \backslash x \to g(fx) \tag{8.28}$$

と書いても同じことである．

✠

関数はアプリカティブ関手でもある．アプリカティブ関手には，アプリカティブマップ演算子とピュア演算

子が定義されるのであった．そこで，関数版のアプリカティブマップ演算子を $\bowtie$ とし，関数版のピュア演算子を $\langle\!\langle \cdots \rangle\!\rangle$ と書くことにしよう．

ピュア演算子は $\mathbf{a} \mapsto (\mathbf{r} \mapsto \mathbf{a})$ 型を持たなければならない．従って関数版のピュア演算子は変数から関数を作るとも考えられる．我々は

$$\langle\!\langle x \rangle\!\rangle = \setminus \_ \to x \tag{8.29}$$

を採用する．

関数版のアプリカティブマップ演算子を $\bowtie$ とすると

$$\bowtie :: Func_{\mathbf{r}}(\mathbf{a} \mapsto \mathbf{b}) \mapsto Func_{\mathbf{r}}\,\mathbf{a} \mapsto Func_{\mathbf{r}}\,\mathbf{b} \tag{8.30}$$

つまり

$$\bowtie :: (\mathbf{r} \mapsto \mathbf{a} \mapsto \mathbf{b}) \mapsto (\mathbf{r} \mapsto \mathbf{a}) \mapsto (\mathbf{r} \mapsto \mathbf{b}) \tag{8.31}$$

である．

我々は関数版アプリカティブマップ演算子として

$$g \bowtie f = \setminus x \to gx(fx) \tag{8.32}$$

とする．関数版マップ演算子は，関数版ピュア演算子と関数版アプリカティブマップ演算子から合成できる．これは

$$\begin{aligned} \langle\!\langle g \rangle\!\rangle \bowtie f &= \setminus x \to \langle\!\langle g \rangle\!\rangle x(\phi f) && (8.33)\\ &= \setminus x \to (\setminus \_ \to g)x(fx) && (8.34)\\ &= \setminus x \to g(fx) && (8.35)\\ &= g \centerdot f && (8.36) \end{aligned}$$

とすれば確かめられる．

8.4 余計な話：アプリカティブマップ演算子の実装*

リストと Maybe のアプリカティブマップ演算子は，それぞれのマップ演算子から定義することができる．リストのアプリカティブマップ演算子の定義は次の通り．

$$\begin{cases} (f : f^\star) \otimes x^\star = \flat\,((f \odot x^\star) : (f^\star \otimes x^\star)) \\ [\,] \otimes x^\star = [\,] \end{cases} \tag{8.37}$$

ここに $(f : f^\star)$ は関数のリストであり，$x^\star$ はリスト変数である．

Maybe のアプリカティブマップ演算子の定義は次の通り．

$$g^? \boxtimes u^? = \begin{cases} h \boxdot u^? & \text{if } g^? \equiv {}^{\mathrm{Just}}[\![h]\!] \\ \emptyset & \text{otherwise} \end{cases} \tag{8.38}$$

ここに $g^?$ は Maybe コンテナに入れられた関数，$u^?$ は Maybe 変数である．

8.5 この章のまとめ*

…

ファンクター

関手は英語でファンクター (functor) と言うが, C++ の関数オブジェクト (function object) もかつてはファンクター (functor) と呼ばれていた．C++ のファンクターとはクロージャの粗末な代用品のことで，本書で述べる関手とは異なる概念である．混同しないように注意しよう．

第 9 章

モナド*

A...

9.1 バインド演算子

一般マップ演算子をピュア演算子と一般アプリカティブマップ演算子に分解することで，式の見通しを良くすることができるアプリカティブスタイルという記法を採用できた．アプリカティブスタイルでは

$$f \bullet u \times v \times w \tag{9.1}$$

という風にコンテナ変数 u, v, w に関数 f を適用させることができる．コンテナ変数 u, v, w のいずれかが $\varnothing$ であれば式全体の値が $\varnothing$ になる．これは 3 個の計算を並列に行って，その結果をそれぞれ u, v, w に入れておき，最後に関数 f に投げるという **計算構造** を具現化したものである．（関数 f は C で言えば `main` 関数に相当するであろう．）

しかしながら，アプリカティブスタイルでは変数に文脈を与えるタイミングがコンテナ変数を作るときのそれぞれ 1 回に限られている．そこで，任意のタイミングで変数に文脈を与えられるように，別な方法で一般マップ演算子を分解してみよう．

Maybe の例を思い出そう．Maybe 型の変数 $u^?$ はラップされた値 ${}^{\mathrm{Just}}[\![x]\!]$ を持つのか，エラーを表す $\emptyset$ を持つのかを選べる．そこで，引数 x をとり何らかの計算をする関数 g を考えよう．この関数 g は引数 x の値次第ではエラーを表す $\emptyset$ を返す．例えば

$$gx = \begin{cases} {}^{\mathrm{Just}}[\![1/x]\!] & \text{if } x \neq 0 \\ \emptyset & \text{otherwise} \end{cases} \tag{9.2}$$

といった関数が考えられる．変数 x は文脈を持っていないが，関数 g を適用した結果である gx は文脈を持っていることに注意しよう．いま gx は Maybe という文脈を持っているから，我々は

$$v^? = gx \tag{9.3}$$

という風に結果を Maybe 変数に保存しなければならない．今まで見てきた $y = fx$ や $v^? = f \boxdot u^?$ の関係とは異なることに注意しよう．

関数 g の型は

$$g :: \mathbf{a} \mapsto {}^{\mathbf{Maybe}}[\![\mathbf{a}]\!] \tag{9.4}$$

である．ということは，関数 g を Maybe 変数に適用させるようと思っても，我々が既に知っているマップ演算子 $\boxdot$ やアプリカティブマップ演算子 $\boxtimes$ が使えないということである．前者は第 1 引数に $\mathbf{a} \mapsto \mathbf{b}$ 型の関数を取るし，後者は第 1 引数に ${}^{\mathrm{Just}}[\![(\mathbf{a} \mapsto \mathbf{b})]\!]$ 型の関数（Maybe 関数）を取るからである．

そこで，新しいマップ演算子を発明する．いま Maybe 変数 $u^?$ を $u^? = {}^{\mathrm{Just}}[\![x]\!]$ としよう．新しいマップ演算子 $\spadesuit$ を使って

$$v^? = g \spadesuit u^? \tag{9.5}$$

とする．この新しいマップ演算子 $\spadesuit$ のことを Maybe の **バインド演算子** と呼ぶ．ここで $v^? = {}^{\mathrm{Just}}[\![gx]\!]$ である．

演算 $(g \spadesuit u^?)$ の結果は Maybe 値であるから，バインド演算子は連続して用いることができる．通常の引数を取って Maybe 値を返すもう一つの関数 h があるとすると

$$v^? = h \spadesuit (g \spadesuit u^?) \tag{9.6}$$

のように連続して関数を適用できる．バインド演算子は右結合するので，上式は

$$v^? = h \spadesuit g \spadesuit u^? \tag{9.7}$$

のように簡潔に書ける．このスタイルなら演算子もすべて統一できていて，かつどの関数でも戻り値を $\emptyset$ に切り替えられるので，アプリカティブスタイルよりも強力と言える．

具体例で考えてみよう．関数 g を

$$gx = \begin{cases} {}^{\mathrm{Just}}\,[\![1/x]\!] & \mathsf{if}\ x \neq 0 \\ \varnothing & \mathsf{otherwise} \end{cases} \tag{9.8}$$

とする．また，関数 h を

$$hy = \begin{cases} {}^{\mathrm{Just}}\,[\![\tan y]\!] & \mathsf{if}\ -\frac{\pi}{2} < y < \frac{\pi}{2} \\ \varnothing & \mathsf{otherwise} \end{cases} \tag{9.9}$$

とする．このとき $u^? = {}^{\mathrm{Just}}\,[\![4/\pi]\!]$ とすると

$$v^? = h \spadesuit g \spadesuit u^? \tag{9.10}$$

の計算結果として $v^? = {}^{\mathrm{Just}}\,[\![1]\!]$ を得る．一方で $u^? = {}^{\mathrm{Just}}\,[\![0]\!], u^? = {}^{\mathrm{Just}}\,[\![1/\pi]\!], u = \emptyset$ などの場合は $v^? = \emptyset$ となり，計算できなかったという結果を得る．

というわけで，Applicative 型クラスをさらに拡張して，一般のバインド演算子を持たせることを考えらてみよう．我々はこの新しい型クラスを **モナド** 型クラスと呼び Monad で表す．

9.2　モナド

Functor 型クラスは一般マップ演算子 ($\bullet$) を持っていた．Applicative 型クラスはピュア演算子 ($\langle x \rangle$) と一般アプリカティブマップ演算子 ($\times$) を持っており，このふたつの演算子から一般マップ演算子を合成できた．新しい Monad 型クラスは，**一般バインド演算子** ($\heartsuit$) とピュア演算子を持つものとしよう．後で見るように，一般マップ演算子はピュア演算子と一般バインド演算子から合成できるし，一般アプリカティブマップ演算子もまたピュア演算子と一般バインド演算子から合成できる．

関数 i が次の形をしているとする．

$$ix = \begin{cases} \langle fx \rangle & \text{嬉しいとき} \\ \varnothing & \mathsf{otherwise} \end{cases} \tag{9.11}$$

ここに x は非コンテナ変数で，関数 f も「普通の」（コンテナではない）関数である．より厳密に言えば $x :: \mathbf{a}$ かつ $f :: \mathbf{a} \mapsto \mathbf{b}$ である．従って，関数 i の型は $\mathbf{a} \mapsto \langle \mathbf{b} \rangle$ である．

関数 i をコンテナに入った変数 $u = \langle x \rangle$ に適用させるのが一般バインド演算子 ($\heartsuit$) の役割である．計算結果をコンテナ変数 a に格納するとすると，関数 i のコンテナ変数 u への適用は

$$a = i \heartsuit u \tag{9.12}$$

と書ける．うまく行けば $a = \langle fx \rangle$ となるし，そうでなければ $a = \varnothing$ となる．

さて $u = \langle x \rangle, v = \langle y \rangle$ として，かつ $y = fx$ であるとき

$$v = f \bullet u = \langle f \rangle \times u \tag{9.13}$$

であった．式 (9.11) から $ix = \langle fx \rangle$ すなわち $i = \langle f \diamond \rangle$ の関係を抜き出すと式 (9.12) は

$$a = \langle f \diamond \rangle \heartsuit u \tag{9.14}$$

となる．いま $y = fx$ であったから $a = v$ であり，最終的に

$$v = f \bullet u = \langle f \rangle \times u = \langle f \diamond \rangle \heartsuit u \tag{9.15}$$

を得る．これが一般バインド演算子と一般アプリカティブマップ演算子，一般マップ演算子の関係である．

9.3　関手則・アプリカティブ関手則・モナド則*

関手，アプリカティブ関手，モナドにはそれぞれ従う規則がある．これらは自然法則ではなく，定義である．

関手の一般マップ演算子 ($\bullet$) は次の規則に従う．

1. 単位元の存在: $\mathrm{id} \bullet u = u$
2. 合成則: $(f \mathbin{.} g) \bullet = (f \bullet) \mathbin{.} (g \bullet)$

ただし関数 id は $\mathrm{id}\, x = x$ で定義される．もちろん $\mathrm{id} = \diamond$ と定義しても同じである．

例えば $\mathrm{id} \bullet x^\star$ は $\mathrm{id}\, x^\star$ であり，結局は $x^\star$ である．単位元の存在とは，そのような関数 id があるという規則である．

合成則のほうは $x^\star = [x]$ を例に考えるとわかりや

すく，

$$(f\bullet)\centerdot(g\bullet)x^\star = (f\bullet(g\bullet\diamond))x^\star \tag{9.16}$$
$$= f\bullet(g\bullet x^\star) \tag{9.17}$$
$$= f\bullet[gx] \tag{9.18}$$
$$= [(f\centerdot g)x] \tag{9.19}$$
$$= (f\centerdot g)\bullet[x] \tag{9.20}$$
$$= (f\centerdot g)\bullet x^\star \tag{9.21}$$

のような関係を一般化したものだと考えれば良い．

✠

アプリカティブ関手の一般アプリカティブマップ演算子 $(\times)$ およびピュア演算子は次の規則に従う．

1. 単位元の存在: $\langle\mathrm{id}\rangle\times u = u$
2. 準同型則: $\langle f\rangle\times\langle x\rangle = \langle fx\rangle$
3. 合成則: $\langle(\centerdot)\rangle\times\phi\times\psi\times v = \phi\times(\psi\times v)$
4. 交換則: $\varphi\times\langle y\rangle = \langle(\$\,y)\rangle\times\varphi$

ここでも $\mathrm{id} = \diamond$ である．

アプリカティブマップ演算子の準同型則は $w = \langle fx\rangle$ とした時に，

$$x \xrightarrow{\langle\cdots\rangle} \langle x\rangle \xrightarrow{\langle f\rangle\times\cdots} w \tag{9.22}$$
$$x \xrightarrow{f} fx \xrightarrow{\langle\cdots\rangle} w \tag{9.23}$$

のように x からスタートして，どちらのルートを辿っても w に行き着くという意味である．

アプリカティブマップ演算子の合成則も注釈が必要であろう．仮に $\phi = \langle g\rangle, \psi = \langle h\rangle, v = \langle z\rangle$ とすると，合成則の左辺は

$$\langle(\centerdot)\rangle\times\phi\times\psi\times v = \langle(\centerdot)g\rangle\times\psi\times v \tag{9.24}$$
$$= \langle(\centerdot)gh\rangle\times v \tag{9.25}$$
$$= \langle g\centerdot h\rangle\times v \tag{9.26}$$
$$= \langle g\centerdot hz\rangle \tag{9.27}$$

となる一方，合成則の右辺は

$$\phi\times(\psi\times v) = \phi\times\langle hz\rangle \tag{9.28}$$
$$= \langle g\centerdot hz\rangle \tag{9.29}$$

となり一致する．合成則とは，このような関係が満たされるように一般アプリカティブマップ演算子を定義しておきなさいという意味だ．

✠

モナドの一般バインド演算子 $(\heartsuit)$ は次の規則に従う．

1. 右単位元の存在: $i\heartsuit\langle x\rangle = ix$
2. 左単位元の存在: $\langle\mathrm{id}\rangle\heartsuit u = u$
3. 結合則: $i\heartsuit(j\heartsuit v) = (i\heartsuit(j\diamond))\heartsuit v$

結合則についてのみ解説しておこう．

$$i\heartsuit\langle z\rangle = \langle fz\rangle,\ j\heartsuit\langle z\rangle = \langle gz\rangle,\ v = \langle y\rangle \tag{9.30}$$

とすると

$$i\heartsuit(j\heartsuit v) = i\heartsuit\langle gy\rangle \tag{9.31}$$
$$= \langle f(gy)\rangle \tag{9.32}$$
$$= \langle f\centerdot gy\rangle \tag{9.33}$$

である一方，$k = i\heartsuit(j\diamond)$ とすると

$$kz = (i\heartsuit(j\diamond))z \tag{9.34}$$
$$= i\heartsuit(jz) \tag{9.35}$$
$$= \langle f(gz)\rangle \tag{9.36}$$
$$= \langle f\centerdot gz\rangle \tag{9.37}$$

であるから

$$k = i\heartsuit(j\diamond) = \langle f\centerdot g\diamond\rangle \tag{9.38}$$

を得る．ここで

$$\langle f\centerdot g\diamond\rangle\heartsuit v = (f\centerdot g)\bullet v \tag{9.39}$$
$$= \langle f\centerdot gy\rangle \tag{9.40}$$

であるから，結合則

$$i\heartsuit(j\heartsuit v) = (i\heartsuit(j\diamond))\heartsuit v \tag{9.41}$$

を得ることになる．

9.4 余計な話：モナドとしての関数*

関数はアプリカティブ関手であった．関数のピュア演算子とアプリカティブマップ演算子はそれぞれ

$$\langle\!\langle x\rangle\!\rangle = \backslash\_\to x \tag{9.42}$$
$$g\bowtie f = \backslash x\to gx(fx) \tag{9.43}$$

であった．関数ピュア演算子は，任意の引数 x を関数に置き換える．ただし，その関数は引数を捨てて元の変数 x を返す．例えば

$$x = \langle\!\langle x\rangle\!\rangle y \tag{9.44}$$

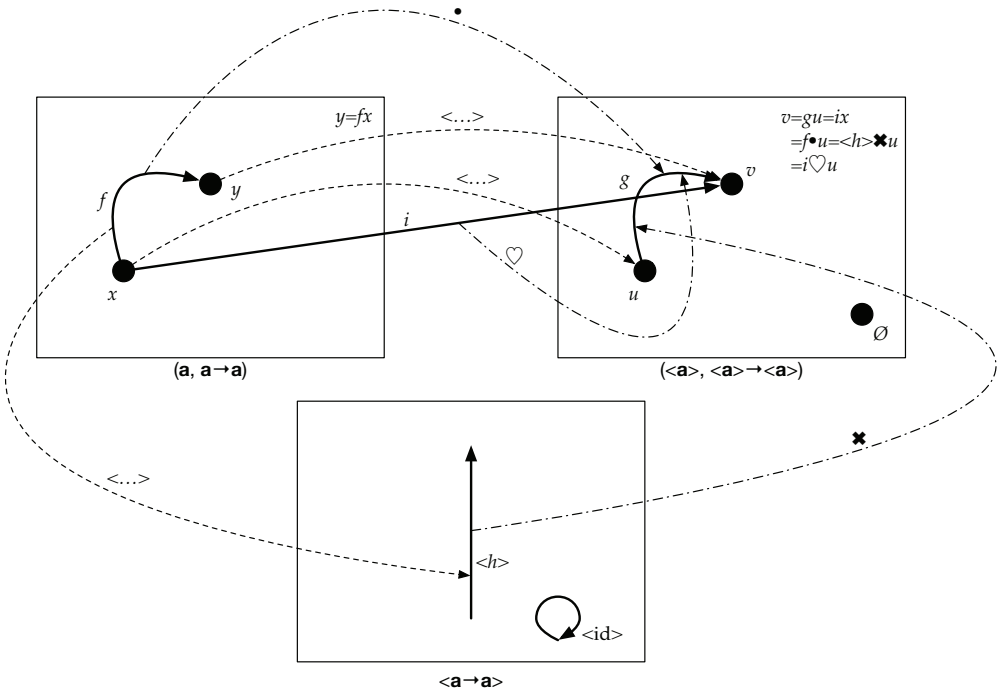


図 9.1　...

表 9.1　型と型クラスの関係

型 \ 型クラス	MonadPlus					
	Monad				Monoid	
	Applicative					
				Functor		
一般コンテナ	♡	$\langle x \rangle$	×	•		
一般モノイド					Ø	∔
リスト	♣	$[x]$	⊗	⊙	[ ]	⊕
Maybe	♠	$^{\text{Just}}[\![x]\!]$	⊠	⊡	∅	⋯
関数	♢	$\langle\langle x \rangle\rangle$	⋈	○	◇	▪
整数					0	+
整数					1	*

と，ダミー変数 y を使って中身の x を取り出せる．

関数のバインド演算子 $\diamondsuit$ を考えておこう．関数のバインド演算子は

$$g \diamondsuit f = \backslash x \to g(fx)x \tag{9.45}$$

と定義する．ピュア演算子とバインド演算子から，関数のマップ演算子を合成できる．

$$\begin{aligned}
\langle\!\langle g\diamond \rangle\!\rangle \diamondsuit f &= \backslash x \to \langle\!\langle g\diamond \rangle\!\rangle (fx)x && (9.46)\\
&= \backslash x \to \langle\!\langle g(fx) \rangle\!\rangle x && (9.47)\\
&= \backslash x \to (\backslash _ \to g(fx))x && (9.48)\\
&= \backslash x \to g(fx) && (9.49)\\
&= g \centerdot f && (9.50)\\
&= g \circ f && (9.51)
\end{aligned}$$

なお関数はモノイドとしての性質も持つ．関数 $\mathrm{id} = \diamond$ とすると，任意の関数 f に対して

$$\mathrm{id} \centerdot f = f \centerdot \mathrm{id} = f \tag{9.52}$$

であるから，関数全体の集合を $\mathbb{F}$ で表すと，組み合わせ $(\mathbb{F}, \centerdot, \mathrm{id})$ はモノイドである．

モナドでありモノイドである型クラスを**モナドプラス**と呼ぶ．関数はモナドプラスである．

今まで出てきた型と型クラスの関係を表 9.3 に示す．Maybe に関しては **a** 型がモノイドである場合に限って $\mathbf{Maybe}\ [\![\mathbf{a}]\!]$ 型もモノイドである．

—

$$\begin{aligned}
\mathrm{const}\, x &= \backslash _ \to x && (9.53)\\
\langle\!\langle f \rangle\!\rangle &= \mathrm{const}\, f && (9.54)\\
i \heartsuit f' &= \backslash r \to i(f'r)r && (9.55)
\end{aligned}$$

$$\begin{aligned}
i \circ f' &= \langle\!\langle i\diamond \rangle\!\rangle \heartsuit f' && (9.56)\\
&= (\mathrm{const}(i\diamond)) \heartsuit f' && (9.57)\\
&= (\mathrm{const} \centerdot i) \heartsuit f' && (9.58)\\
&= \backslash r \to (\mathrm{const} \centerdot i)(f'r)r && (9.59)\\
&= \backslash r \to \mathrm{const}(i(f'r)r && (9.60)\\
&= \backslash r \to i(f'r) && (9.61)\\
&= \backslash r \to (i \centerdot f')r && (9.62)\\
&= \langle\!\langle i \centerdot f' \rangle\!\rangle && (9.63)
\end{aligned}$$

9.5　この章のまとめ*

若干の歴史的な混乱について

Haskell には最初に関手が導入され，その次に関手を拡張する形でモナドが導入された．そしてその次に，関手を拡張しなおす形でアプリカティブ関手が導入された．それゆえ，モナドとアプリカティブ関手には概念的重複があるにもかかわらず，別々に定義されるという悲劇が暫くの間続いた．アプリカティブ関手のピュア演算子と，モナドのユニット演算子は概念的に同じものであるにもかかわらず，別々の演算子として定義されていたのである．この状態は GHC v7.10 以降で，モナドがアプリカティブ関手を拡張する形に改められたことで解消した．

第 10 章

IO

A...

10.1 アクション

計算機の状態を変えることを **副作用** と呼ぶ．副作用とは変数への破壊的代入に他ならない．例えば，計算機は画面やプリンタに何かを出力するが，それは画面やプリンタという「変数」を書き換えていることになる．また例えば擬似乱数の生成も副作用である．呼び出されるたびに異なる値を返す擬似乱数生成関数は，そのたびに計算機の内部状態を書き換えているのである．

副作用を持つ関数を **アクション** と呼ぶ．アクションは値を持つが，その値は計算の実行時までわからない．例えば擬似乱数を生成するアクション $rand$ があるとしよう．これを変数 α に

$$\alpha = rand \tag{10.1}$$

と代入しても，変数 α に擬似乱数が代入されるわけではない．「擬似乱数を生成する」というアクションが α に代入されたのだ．

では，いつ「擬似乱数を生成する」アクションが実行されるのだろうか．それは，プログラムがまさに計算機の状態を変えるタイミング，つまりプログラムが実行されるタイミングなのである．そのためには，プログラムそのものをアクションで表しておかないといけない．我々はプログラム全体を ω で表すことにしよう．

プログラムが計算機状態を変える一例として，画面に値を出力することを考える．画面に値を出力するアクションを $putstr$ と名付けよう．例えば次のプログラム例は，画面に “Hello, world.” と書き出すものとする．

$$\omega = putstr\, s \text{ where } s \triangleq \text{“Hello, world.”} \tag{10.2}$$

アクション $putstr$ は変数を一つとり，その値を画面へ出力すなち破壊的代入を行う．では $\omega = putstr\, \alpha$ とすればアクション α が実行されて，晴れて擬似乱数が生成され，その値が画面へ出力されるだろうか．もちろんそうはならないのである．

ここに α はアクションであり，変数ではない．一方で，アクション $putstr$ は変数を受け取る．つまり，アクションから何らかの方法で値を「安全に」抜き取らないといけない．ここで安全性にこだわるのは，アクション α が副作用を持つからである．副作用を参照透過な変数へ伝播させてはいけない．副作用を持つアクションは，副作用を持つアクションへのみ受け継がれなければならない．

この話は何かと似ていないだろうか．そう，Maybe である．一度ゼロ除算の可能性に汚染されたコンテナ変数は，コンテナから出すことが許されないのである．Maybe を返す関数 f の戻り値

$$fx = \begin{cases} ^{\text{Just}}[\![1/x]\!] & \text{if } x \neq 0 \\ \emptyset & \text{otherwise} \end{cases} \tag{10.3}$$

を，別の関数

$$gy = 1 + y \tag{10.4}$$

に渡そうと思ったら，

$$g \heartsuit fx \tag{10.5}$$

のようにバインド演算子で合成しなければならなかった．

我々はアクションにもバインド演算子を拡張して，

$$\omega = putstr \heartsuit \alpha \tag{10.6}$$

とする．これは第 9 章で見たバインド演算子と同じものである．同じバインド演算子が使えるカラクリは，次節で見ていくことにする．

10.2 IO モナド*

ある関数 c が引数を取らず，いつも決まった **Double** 型の数を返すとしよう．そうすると，関数 c の型は単純に **Double** である．

呼び出すたびに異なる擬似乱数を返すアクション $rand$ もまた毎回 **Double** 型の数を返す．そこで $rand$ も **Double** 型としたいところだが，こちらは関数ではなくアクションである．そこを区別するために，$rand$ の型は $^{\mathbf{IO}}[\![\mathbf{Double}]\!]$ 型と $^{\mathbf{IO}}[\![\cdots]\!]$ に入れて区別する．[*1]

呼び出しても何も返さないアクションはどのような型を持つべきだろう．何も返さない関数というものは無意味だが，アクションは副作用を持つので，何も返さないものがあっても良いのである．何も返さないことを「空っぽ」を返すと読み替えて，何も返さないアクションの型を $^{\mathbf{IO}}[\![()]\!]$ 型としよう．ここに () は「空っぽ」の意味で，**ユニット型** と読む．$^{\mathbf{IO}}[\![()]\!]$ 型のアクションの例は，画面に値を出力する $putstr$ アクションである．[*2]

キーボードからの入力を受け取るアクションもある．そのアクションを $readln$ としよう．アクション $readln$ は文字列型を返すので，その型は $^{\mathbf{IO}}[\![\mathbf{String}]\!]$ である．[*3]

アクション $putstr, readln, rand$ は計算機の状態を変化させる．アクション $putstr, readln$ は OS のシステムコールを発行して，前者ならビデオメモリの値を書き換えるし，後者ならシリアルインタフェースの入力バッファをフラッシュする．アクション $rand$ は呼ばれるたびに内部のカウンタ値を一つ進める．これらのアクションは参照透過性を破壊する．そのために，プログラムの他の部分から隔離されねばならない．その隔離のメカニズムを提供するのがモナドである．

[**a**] 型が **a** 型のリストであるように，$^{\mathbf{IO}}[\![\mathbf{a}]\!]$ 型は **a** 型の **IO** である．そしてリストがモナドであるように，IO もまたモナドである．モナドには，バインド演算子と，アプリカティブ関手から引き継いだピュア演算子の二つが必要であった．アプリカティブ関手から引き継いだアプリカティブマップ演算子はバインド演算子とピュア演算子の二つから合成できるし，関手から引き継いだマップ演算子もまたアプリカティブマップ演算子とピュア演算子から合成できるから，アプリカティブマップ演算子とマップ演算子は改めて実装しておく必要はない．

$^{\mathbf{IO}}[\![()]\!]$ 型の場合，ピュア演算子は何も返す必要がないから

$$\langle \_ \rangle = () \tag{10.7}$$

であるとする．ここに () は空のタプルで，「ユニット」と呼ぶ．一方で $^{\mathbf{IO}}[\![()]\!]$ 型のバインド演算子は，処理系の奥深くに隠されている．

10.3 do 記法

モナドを使った書き方が従来のプログラムの記法からあまりにもかけ離れていることに，Haskell の設計者は気づいていたようで，Haskell には次に述べる **do 記法** という記法が用意されている．この do 記法はもちろんシンタックスシュガーで，新しいことは何もない．

例えば $\omega = putstr \heartsuit \alpha$ where $\alpha \triangleq rand$ を do 記法を用いて書き直すと

$$\omega = \mathsf{do}\,\{a \leftarrow rand \Rrightarrow putstr\, a\} \tag{10.8}$$

となる．[*4]

ここで，同様なことをする Python プログラムを見てみよう．

Python

```
import random
define main:
  a = random.random() # (1)
  print(a)            # (2)
```

コード中の (1) の行で $a \leftarrow rand$ を実行し，(2) の行で $putstr\, a$ を実行していると思えば，このコードと式 (10.8) の順序はそっくり同じである．もちろんこの do 記法はバインド演算子を使った式を切り貼りして，順序を入れ替えただけである．

[*1] Haskell では $^{\mathbf{IO}}[\![\mathbf{Double}]\!]$ のことを `IO Double` と書く．
[*2] Haskell では $^{\mathbf{IO}}[\![()]\!]$ のことを `IO ()` と書く．
[*3] Haskell では $^{\mathbf{IO}}[\![\mathbf{String}]\!]$ のことを `IO String` と書く．

[*4] Haskell では

```
main = do
  a <- rand
  putStr a
```

と書く．

do 記法には ← の他に，我々の let とよく似た `let` という構文が用意されている．この `let` は局所変数の導入に用いられて，例えば

$$\omega = \mathsf{do}\,\{\texttt{let}\,y = fx \Rrightarrow \alpha y\} \tag{10.9}$$

のように使う．

以下に，do 記法を使った例を示す．

$$\mathsf{do}\,\{\alpha x\} = \alpha x \tag{10.10}$$
$$\mathsf{do}\,\{y \leftarrow \alpha x \Rrightarrow \beta y\} = \beta \heartsuit \alpha x \tag{10.11}$$
$$\mathsf{do}\,\{\alpha x \Rrightarrow \beta y\} = (\backslash \_ \to \beta y) \heartsuit \alpha x \tag{10.12}$$
$$\mathsf{do}\,\{\texttt{let}\,y = fx \Rrightarrow \alpha y\} = \alpha y \;\mathsf{where}\; y \triangleq fx \tag{10.13}$$
$$\mathsf{do}\,\{y \leftarrow \alpha x \Rrightarrow \texttt{let}\,z = fy \Rrightarrow \beta z\} = \beta \heartsuit (f \bullet \alpha x) \tag{10.14}$$

より複雑な例も挙げる．

$$\mathsf{do}\,\{y \leftarrow \alpha x \Rrightarrow y' \leftarrow \alpha' x' \Rrightarrow \texttt{let}\,z = fyy' \Rrightarrow \beta z\} = \beta \heartsuit (f \bullet \alpha x \times \alpha' x') \tag{10.15}$$

最後に do 記法中に変数を 2 回以上使いまわす例を表 10.4 に示す．この例では変数 y, y' が 2 回使われている．

我々はバインド演算子を使った通常の記法と do 記法のいずれか読みやすい方を採用すれば良い．

10.4　余計な話：関数であるということ*

IO モナドの変数は，たとえ値を読み出すだけであっても必ず関数適用が必要である．それは，IO モナドの変数が「自分が読み出されたこと」を知る必要があるからである．IO モナドの変数は，自分が読み出されたタイミングで副作用を発生させる．これは Objective-C や Swift に見られる getter メソッドと同じ考え方である．

IO モナドの変数値を読み出すために行われる関数呼び出しはダミーである場合があり，戻り値はしばしば捨てられる．

変数をダミーの関数で包み，それをさらに IO モナドで包んだのが IO モナド変数である．

もうひとつ，アクションが関数でなければならない理由がある．Haskell はいつも遅延評価を行うことを思い出してもらいたい．Python であれば，式は書かれた順に評価される．しかし，例えば

$$y_1 = f_1 x_1 \tag{10.16}$$
$$y_2 = f_2 x_2 \tag{10.17}$$

という式があった場合，関数 f_1 と f_2 のどちらが先に評価されるか，あるいは同時に評価されるかは Haskell では未定義である．Haskell で唯一計算順序が保証されているのは，関数適用である．例えば

$$y = g_2(g_1 x) \tag{10.18}$$

であれば，確実に関数 g_1 が関数 g_2 よりも先に評価される．

直列に評価したい関数の戻り値が，いつも次の関数の引数の型と一致しているとは限らないし，次の関数（アクション）が引数を取らない可能性もある．もし関数 g_2 が引数を取らなければ

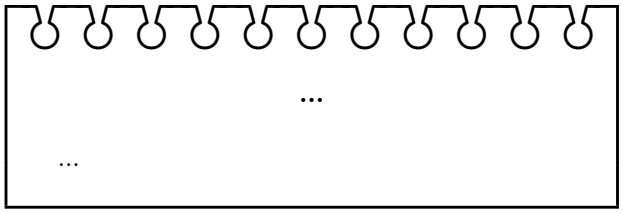
$$\langle\!\langle g_2 \rangle\!\rangle (g_1 x) \tag{10.19}$$

とする．[*5]

—

10.5　この章のまとめ*

1. do 記法.



[*5] C では `void` 型を返す関数 `void g1(int);` を 引数を取らない関数 `int g2(void);` に「食わせる」ことが可能で，`g2(g1(x));` は正しいコードである．もっとも C プログラマは `g1(x), g2();` という書き方の方を好むであろう．

表 10.1　do 記法中に変数を 2 回以上使いまわす例

$$\mathsf{do}\,\{y \leftarrow \alpha x \Rrightarrow y' \leftarrow \alpha' x' \Rrightarrow \mathtt{let}\; z = fyy' \Rrightarrow \beta z \Rrightarrow \mathtt{let}\; z' = f'yy' \Rrightarrow \beta' z'\}$$
$$= (\setminus yy' \to ((\setminus \_ \to \mathsf{let}\; z \triangleq fyy'\; \mathsf{in}\; \beta z) \heartsuit (\mathsf{let}\; z' \triangleq f'yy'\; \mathsf{in}\; \beta' z')))(\alpha x)(\alpha' x')$$

第 11 章

データ型の定義*

...

11.1 データ型

我々はしばしば新しい集合を考える必要に迫られる．例えば，イチ，ニ，サン，タクサンからなる集合

$$\mathbf{Num} \stackrel{\text{def}}{=} \{\mathrm{One}, \mathrm{Two}, \mathrm{Three}, \mathrm{Many}\} \tag{11.1}$$

を考えることがあるだろう．集合 $\mathbf{Num}$ の元 $n :: \mathbf{Num}$ は $\mathrm{One}, \mathrm{Two}, \mathrm{Three}, \mathrm{Many}$ のいずれかの値を取ることになる．

数学者は新しい集合を定義するが，Haskell プログラマは新しい **データ型** を定義する．データ型というのは，型の上品な言い方に過ぎない．集合 $\mathbf{Num}$ の定義の代わりに，我々はデータ型 $\mathbf{Num}$ を

$$\underline{\mathsf{data}}\mathsf{type}\ \mathbf{Num} = \mathrm{One} \curlyvee \mathrm{Two} \curlyvee \mathrm{Three} \curlyvee \mathrm{Many} \tag{11.2}$$

のように書いて定義するものとする．式の先頭にある $\underline{\mathsf{data}}\mathsf{type}$ は，この式がデータ型の定義であることを示すタグである．部分的にアンダーラインが引かれているのは，Haskell が $\underline{\mathsf{data}}\mathsf{type}$ を `data` と省略してしまうからである．これは不本意なことだが，文字数節約のために仕方ない．ここに $\mathbf{Num}$ は型コンストラクタ，$\mathrm{One}, \mathrm{Two}, \mathrm{Three}, \mathrm{Many}$ は値コンストラクタである．[*1]

この節で紹介したデータ型の定義は C で言う `enum` に近い．他に C で言う `struct` や `union` すなわち構造体や共用体もこの $\underline{\mathsf{data}}\mathsf{type}$ 文で定義出来るが，これは次節で見ることにする．

[*1] Haskell では $\underline{\mathsf{data}}\mathsf{type}$ を `data` と書く．式 $\underline{\mathsf{data}}\mathsf{type}\ \mathbf{Num} = \mathrm{One} \curlyvee \mathrm{Two} \curlyvee \mathrm{Three} \curlyvee \mathrm{Many}$ は `data Color = One | Two | Three | Many` と書く．

11.2 レコード構文*

型の定義.

$$\underline{\mathsf{data}}\mathsf{type}^{\dagger}\ \mathbf{Shape} = {}^{\mathrm{Circle}}[\![\mathbf{Float}\ \mathbf{Float}]\!] \curlyvee {}^{\mathrm{Triangle}}[\![\mathbf{Float}\ \mathbf{Float}\ \mathbf{Float}]\!] \tag{11.3}$$

$$\underline{\mathsf{type}}\mathsf{synonym}\ \mathbf{Vector2D} = (\mathbf{Float}, \mathbf{Float}) \tag{11.4}$$

$$\begin{aligned} \underline{\mathsf{data}}\mathsf{type}\ \mathbf{Rectangle} = {}^{\mathrm{Rectangle}}\{ & \quad (11.5)\\ \mathit{origin} :: \mathbf{Vector2D}, & \quad (11.6)\\ \mathit{size} :: \mathbf{Vector2D} & \quad (11.7)\\ \} & \quad (11.8) \end{aligned}$$

$$\underline{\mathsf{data}}\mathsf{type}^{\dagger}\ \mathit{Either}\ \mathbf{ab} = {}^{\mathrm{Left}}[\![\mathbf{a}]\!] \curlyvee {}^{\mathrm{Right}}[\![\mathbf{b}]\!] \ \mathsf{deriving}\ (\textsc{Eq}, \textsc{Ord}) \tag{11.9}$$

11.3 型クラスとインスタンス化

型クラスとは，複数の型が持つ共通のインタフェースである．その複数の型をいま $\mathbf{a}$ で表すこととして，型 $\mathbf{a}$ が型クラス $\textsc{Eq}$ に属すとしよう．型クラス $\textsc{Eq}$ は等号 $(\equiv)$ と不等号 $(\neq)$ をインタフェースとして持つ．このことを

$$\begin{aligned} \mathsf{typeclass}\ \textsc{Eq} \supset \mathbf{a}\ \mathsf{where}\ (\equiv) :: \mathbf{a} \mapsto \mathbf{a} \mapsto \mathbf{Bool} & \quad (11.10)\\ (\neq) :: \mathbf{a} \mapsto \mathbf{a} \mapsto \mathbf{Bool} & \quad (11.11)\\ x \equiv y = \neg(x \neq y) & \quad (11.12)\\ x \neq y = \neg(x \equiv y) & \quad (11.13) \end{aligned}$$

と書く．最初の 2 行は型クラス Eq が等号と不等号を持つことを宣言しており，続く 2 行がその性質を宣言している．[*2]

ここで宣言したのは等号，不等号というインタフェースが「ある」という事だけで，その実装は未定義である．等号，不等号の実装は次に述べる instance 文を使う．

型クラス Eq に属す型 **a** は等号と不等号を持つ．我々の型 **Num** が型クラス Eq に属すことを宣言し，型クラス Eq が備えるべき等号，不等号を実装するには

$$\begin{aligned}&\mathsf{instance}\ \mathrm{EQ} \supset \mathbf{Num}\\ &\qquad \mathsf{where}\\ &\qquad\quad \mathrm{One} \equiv \mathrm{One} = \mathrm{True}\\ &\qquad\quad \mathrm{Two} \equiv \mathrm{Two} = \mathrm{True}\\ &\qquad\quad \mathrm{Three} \equiv \mathrm{Three} = \mathrm{True}\\ &\qquad\quad \mathrm{Many} \equiv \mathrm{Many} = \mathrm{True}\\ &\qquad\quad \_ \equiv \_ = \mathrm{False}\end{aligned}$$

とする．これを **型クラスのインスタンス化** と呼ぶ．[*3]

型クラスは **継承関係** を持てる．型クラス Ord は型クラス Eq から等号，不等号を継承し「小なりイコール」 ($\leq$) を追加する．

$$\begin{aligned}&\mathsf{typeclass}\ (\mathrm{EQ} \supset \mathbf{a}) \Rightarrow \mathrm{ORD}\ \mathbf{a}\\ &\qquad\qquad \mathsf{where}\ (\leq) :: \mathbf{a} \mapsto \mathbf{a} \mapsto \mathbf{Bool}\end{aligned} \tag{11.14}$$

型クラス Ord に属する型は，等号，不等号，小なりイコールと，それらから派生させることの出来る大なり ($>$)，大なりイコール ($\geq$)，小なり ($<$)，および最大値をとる関数 max と最小値をとる関数 min を持つ．[*4]

deriving

—

11.4　余計な話：型シノニム*

データ型には **シノニム**（別名）がつけられる．例えば **Char** のリスト [**Char**] は

$$\mathsf{typesynonym}\ \mathbf{String} = [\mathbf{Char}] \tag{11.15}$$

とすることで，別名 **String** を与えることが出来る．Haskell では typesynonym を type と省略してしまう．これは残念なことだが，C の typedef のようなものだと思って割り切るしかない．[*5]

11.5　この章のまとめ*

メタクラス

オブジェクト指向言語の多くが「クラス」と呼ぶものを，Haskell は「型」と呼ぶ．

[*2] Haskell では typeclass を class と書く．また ⊃ を省略して，

```
class Eq a where (==) :: a -> a -> Bool
                 (/=) :: a -> a -> Bool
                 x==y = not(x/=y)
                 x/=y = not(x==y)
```

と書く．

[*3] Haskell では

```
instance Eq Num
  where One==One     = True
        Two==Two     = True
        Three==Three = True
        Many==Many   = True
        _==_         = False
```

と書く．

[*4] Haskell では

```
class (Eq a) => Ord a
  where (<=) :: a -> a -> Bool
```

と書く．

[*5] Haskell では typesynonym **String** = [**Char**] を `type String = [Char]` と書く．

第 12 章

多相型の定義*

...

12.1 パラメトリックなデータ型

Maybe のように型パラメタを取る型はどのように定義されるかと言うと，

$$\underline{\text{data}}\text{type}^{\dagger}\ \ \mathit{Maybe}\,\mathbf{a} = {}^{\mathrm{Just}}[\![\mathbf{a}]\!] \curlyvee \emptyset \tag{12.1}$$

のように，やはり datatype を使って定義される．念のため型パラメタを取る場合は datatype† と区別しておこう．[*1]

Maybe が型クラス Eq に属すものとして，Eq からのインスタンス化をしておこう．ここでも型パラメタ付きの instance を仮に instance† とすると次のように書けそうである．

$$\begin{aligned}&\text{instance}^{\dagger}\ \mathrm{EQ} \supset (\mathit{Maybe}\,\mathbf{a})\\ &\qquad\text{where}\\ &\qquad\quad {}^{\mathrm{Just}}[\![x]\!] \equiv {}^{\mathrm{Just}}[\![y]\!] = x \equiv y\\ &\qquad\quad \emptyset \equiv \emptyset = \mathrm{True}\\ &\qquad\quad \_ \equiv \_ = \mathrm{False}\end{aligned}$$

残念ながら，この式は $\mathbf{a}$ 型の変数 x, y の間に等号 $(\equiv)$ が定義されていることが隠れた前提になっているため，正しくない．我々は $\mathbf{a}$ が型クラス Eq に属すことを要求するので，次のように言い換える．

$$\begin{aligned}&\text{instance}^{\dagger}\ (\mathrm{EQ} \supset \mathbf{a}) \Rightarrow \mathrm{EQ} \supset (\mathit{Maybe}\,\mathbf{a})\\ &\qquad\text{where}\\ &\qquad\quad {}^{\mathrm{Just}}[\![x]\!] \equiv {}^{\mathrm{Just}}[\![y]\!] = x \equiv y\\ &\qquad\quad \emptyset \equiv \emptyset = \mathrm{True}\\ &\qquad\quad \_ \equiv \_ = \mathrm{False}\end{aligned}$$

この $(\mathrm{EQ} \supset \mathbf{a}) \Rightarrow$ の部分が「以下 $\mathbf{a}$ 型は Eq 型クラスに属すものとして」という意味になる．[*2]

Maybe の間に新たに定義された等号 $(\equiv)$ は次の型を持つ．

$$(\equiv) :: (\mathrm{EQ} \supset \mathbf{a}) \Rightarrow {}^{\mathbf{Maybe}}[\![\mathbf{a}]\!] \mapsto {}^{\mathbf{Maybe}}[\![\mathbf{a}]\!] \mapsto \mathbf{Bool} \tag{12.2}$$

—

自己参照型

Tree

12.2 関手の拡張

型クラス Functor はマップ演算子 $(\bullet)$ を提供する．

型ではなく型コンストラクタを定義する typeclass† を使ってみよう．

$$\begin{aligned}&\text{typeclass}^{\dagger}\ \ \textsc{Functor} \supset \mathit{List}\\ &\qquad\qquad\text{where}\ \bullet :: (\mathbf{a} \mapsto \mathbf{b}) \mapsto [\mathbf{a}] \mapsto [\mathbf{b}]\end{aligned} \tag{12.3}$$

[*3]

$$\text{instance}^{\dagger}\ \ \textsc{Functor} \supset \mathit{List}\ \text{where}\ \bullet \triangleq \odot \tag{12.4}$$

[*1] Haskell は datatype と datatype† を区別せず，datatype† $\mathit{Maybe}\,\mathbf{a} = {}^{\mathrm{Just}}[\![\mathbf{a}]\!] \curlyvee \emptyset$ を `data Maybe a = Just a | Nothing` と書く．

[*2] Haskell では

```
instance (Eq a) => Eq (Maybe a)
  where Just x == Just y = x==y
        Nothing==Nothing = True
        _==_             = False
```

と書く．

[*3] Haskell では

```
class Functor []
  where map :: (a -> b) -> [a] -> [b]
```

と書く．

*4

—

Maybe の場合は次のようになる．

$$\underline{\mathsf{typeclass}}^{\dagger}\ \textsc{Functor} \supset \mathit{Maybe}$$
$$\mathsf{where}\ \bullet :: (\mathbf{a} \mapsto \mathbf{b}) \mapsto {}^{\mathbf{Maybe}}[\![\mathbf{a}]\!] \mapsto {}^{\mathbf{Maybe}}[\![\mathbf{b}]\!] \quad (12.5)$$

*5

$$\begin{aligned}&\mathsf{instance}^{\dagger}\ \textsc{Functor} \supset \mathit{Maybe}\\ &\qquad \mathsf{where}\\ &\qquad\quad f \bullet {}^{\mathrm{Just}}[\![x]\!] = {}^{\mathrm{Just}}[\![fx]\!]\\ &\qquad\quad f \bullet \emptyset = \emptyset\end{aligned}$$

*6

—

Either の場合．

$$\underline{\mathsf{typeclass}}^{\dagger}\ \textsc{Functor} \supset (\mathit{Either}\ \mathbf{a})$$
$$\mathsf{where}\ \bullet :: (\mathbf{a} \mapsto \mathbf{b}) \mapsto {}^{\mathbf{Either}}[\![\mathbf{a}, \mathbf{a}]\!] \mapsto {}^{\mathbf{Either}}[\![\mathbf{a}, \mathbf{b}]\!] \quad (12.6)$$

$$\begin{aligned}&\mathsf{instance}^{\dagger}\ \textsc{Functor} \supset (\mathit{Either}\ \mathbf{a})\\ &\qquad \mathsf{where}\\ &\qquad\quad f \bullet {}^{\mathrm{Right}}[\![x]\!] = {}^{\mathrm{Right}}[\![fx]\!]\\ &\qquad\quad f \bullet {}^{\mathrm{Left}}[\![x]\!] = {}^{\mathrm{Left}}[\![x]\!]\end{aligned}$$

*7

—

一般化すると次のようになる．

$$\underline{\mathsf{typeclass}}^{\dagger}\ \textsc{Functor} \supset \mathit{fn}$$
$$\mathsf{where}\ \bullet :: (\mathbf{a} \mapsto \mathbf{b}) \mapsto {}^{\mathbf{fn}}[\![\mathbf{a}]\!] \mapsto {}^{\mathbf{fn}}[\![\mathbf{b}]\!] \quad (12.7)$$

ここで，型 ${}^{\mathbf{fn}}[\![\mathbf{a}]\!]$ は型 $\mathbf{a}$ と型コンストラクタ fn から

$${}^{\mathbf{fn}}[\![\mathbf{a}]\!] = \mathit{fn}\mathbf{a} \quad (12.8)$$

のように作られる．*8

12.3 余計な話: newtype

リストに新しいマップ演算子 $\circledast$ を定義したいとしよう．この演算子 $\circledast$ は

$$[f, g, h] \circledast [x, y, z] = [fx, gy, hz] \quad (12.9)$$

のように働くとする．新しいマップ演算子 $\circledast$ のことを我々は **ジップ演算子** と呼ぶことにする．

ジップ演算子を定義するにはどうしたら良いだろうか．汎用性を考えると，ジップ演算子もまたリストのアプリカティブマップ演算子であって欲しい．ところが，リストには既に $\otimes$ というアプリカティブマップ演算子が定義されている．念のためにアプリカティブマップ演算子を用いると

$$[f, g, h] \otimes [x, y, z] = [fx, fy, fz, gx, gy, gz, hx, hy, hz] \quad (12.10)$$

である．ある型のインスタンス化は一種類しか行えないので，リスト型に新しいアプリカティブマップ演算子を追加することは出来ない．

そこで $\underline{\mathsf{data}}\mathsf{type}^{\dagger}$ を使ってリスト型をラップした新しい型を作る．例えば新しい型を ${}^{\mathbf{Ziplist}}[\![\mathbf{a}]\!]$ とすると

$$\underline{\mathsf{data}}\mathsf{type}^{\dagger}\ \mathit{Ziplist}\ \mathbf{a} = {}^{\mathrm{Ziplist}}\{\mathit{getlist} :: [\mathbf{a}]\} \quad (12.11)$$

のように定義することになる．念のため，左辺の *Ziplist* は型コンストラクタ，右辺の Ziplist は値コンストラクタである．すなわち

$${}^{\mathbf{Ziplist}}[\![\mathbf{a}]\!] = \mathit{Ziplist}\ \mathbf{a} \quad (12.12)$$

---

*4 Haskell では instance† FUNCTOR *List* where • ≙ ⊙ を，instance と instance† は区別せずに `instance Functor [] where fmap = map` と書く．

*5 Haskell では

```
class Functor Maybe
  where map :: (a -> b) -> Maybe a -> Maybe b
```

と書く．

*6 Haskell では

```
instance Functor Maybe
  where fmap f (Maybe x) = Maybe (f x)
        fmap f Nothing = Nothing
```

と書く．

*7 Haskell では

```
instance Functor (Either a)
  where fmap f (Right x) = Right (f x)
        fmap f (Left x)  = Left x
```

と書く．

*8 Haskell では ${}^{\mathbf{fn}}[\![\mathbf{a}]\!]$ も $\mathit{fn}\mathbf{a}$ も同様に `fn a` と書く．そこで式 (12.7) は

```
class functor fn
  where fmap :: (a -> b) -> fn a -> fn b
```

と書く．

である．

こうしておいて，あとは

$$\mathsf{instance}^{\dagger}\ \textsc{Applicative} \supset \mathit{Ziplist}\ \mathbf{a} \tag{12.13}$$

$$\mathsf{where} \tag{12.14}$$

$$\langle x \rangle = {}^{\mathrm{Ziplist}}[\![\mathrm{repeat}\ x]\!] \tag{12.15}$$

$${}^{\mathrm{Ziplist}}[\![f^{\star}]\!] \times {}^{\mathrm{Ziplist}}[\![x^{\star}]\!] \tag{12.16}$$

$$= {}^{\mathrm{Ziplist}}[\![\mathit{zipwith}(\backslash\, fx \to fx) f^{\star} x^{\star}]\!] \tag{12.17}$$

とすれば上手くいく．ここに関数 *zipwith* は

$$\mathit{zipwith}\ \_[\,]\_ = [\,] \tag{12.18}$$

$$\mathit{zipwith}\ \_\_[\,] = [\,] \tag{12.19}$$

$$\mathit{zipwith}\ f(x : x^{\star})(y : y^{\star}) = fxy : \mathit{zipwith}\ f x^{\star} y^{\star} \tag{12.20}$$

であり，関数 *zipwith* の型は

$$\mathit{zipwith} :: (\mathbf{a} \mapsto \mathbf{b} \mapsto \mathbf{c}) \mapsto [\mathbf{a}] \mapsto [\mathbf{b}] \mapsto [\mathbf{c}] \tag{12.21}$$

である．

関数 repeat は引数を無限回繰り返すリストを返す関数である．念のため関数 repeat の実装方法を書いておくと

$$\mathrm{repeat}\ x = x : \mathrm{repeat}\ x \tag{12.22}$$

である．関数 repeat の型は

$$\mathrm{repeat} :: [\mathbf{a}] \mapsto [\mathbf{a}] \tag{12.23}$$

である．

任意のリスト $x^{\star}$ と任意の関数リスト $f^{\star}$ ただし

$$x^{\star} = [x_0, x_1, \cdots] \tag{12.24}$$

$$f^{\star} = [f_0, f_1, \cdots] \tag{12.25}$$

の両方をそれぞれ値コンストラクタで包んでアプリカティブマップ演算子を適用すると

$${}^{\mathrm{Ziplist}}[\![f^{\star}]\!] \times {}^{\mathrm{Ziplist}}[\![x^{\star}]\!] = [f_0 x_0, f_1 x_1, \cdots] \tag{12.26}$$

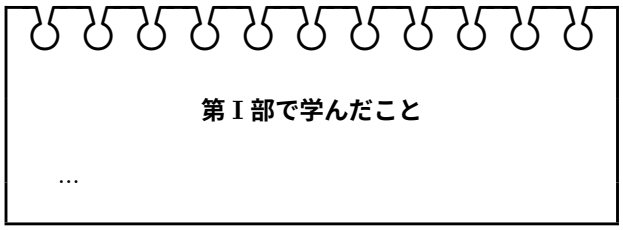
のようにジップ演算子を適用できる．

このままでも問題はないのだが，式 (12.11) を次のように書き換えることがより好ましい．

$$\mathsf{newtype}^{\dagger}\ \ \mathit{Ziplist}\ \mathbf{a} = {}^{\mathrm{Ziplist}}\{\mathit{getlist} :: [\mathbf{a}]\} \tag{12.27}$$

やったことはキーワード datatype† を newtype† に置き換えたことである．これは，字面に反して，型 **Ziplist** が何一つ「新しくない」ことを Haskell コンパイラに伝えるためである．

12.4　この章のまとめ*



第 II 部

Haskell プログラミング

第 13 章

プログラム

A...

13.1 文字セットとコメント

Haskell コンパイラを含む多くのコンパイラが Unicode 文字セットに対応しているものの，従来からの習慣や英語圏での使いやすさを考慮してか，ASCII 文字セットだけでプログラムを書けるようにしているし，またそれを推奨している．

Haskell プログラムもまた, 文字定数を除いては ASCII 文字セットの範囲で書くことが普通である．そこで我々もその習慣に従うことにしよう．例えば円周率を代入する変数を π と書きたいところだが，我々は `pi` と書く．

数学記号のほとんども，ASCII 文字セットの中から記号を組み合わせるか，さもなくば言葉で表現する．例えば Haskell では $\in$ の代わりに `<-` を使うし，$\wedge$ の代わりに `and` を使う．

また計算機科学者たちの絶えざる努力にもかかわらず，プログラム中の文字の装飾はこれまでほとんど受け入れられていない．我々は本書で $\mathrm{f}, f, \mathbf{f}, \mathsf{f}, f$ を使い分けてきたが，Haskell プログラム中では全て `f` と書く．

以上のような制約にもかかわらず，Haskell プログラムと我々が見えきた「カリー風の」数学記法は本質的に差がない．本書を読み進めてきた読者なら，Haskell プログラムを読むのに苦労はいらないだろう．

なお Haskell では `--` から行の終わりまでがコメントとして扱われる．

13.2 main 関数と一般の関数定義

Python インタプリタはプログラムを頭から実行していくので，*main* 関数は書かなくてもよいが，アプリケーションプログラマにとっての一番の関心事は *main* 関数の書き方だろう．iOS アプリケーション開発のように，基本的には *main* 関数をプログラマが触れないというスタイルもあるが，それでもデバッグの時には *main* 関数から辿ることになるので，*main* 関数のありかを知っておくことはいつでも重要だ．

Haskell コンパイラも *main* 関数をアプリケーションプログラムのエントリポイントとして認識する．というよりも，Haskell プログラムとは *main* という一つの関数である．*main* 関数を含む一般の関数はこれまで通り

$$main = \cdots \tag{13.1}$$

のように関数名のあとに等号 $(=)$ を置いて定義する．

C で *main* 関数の型が OS の都合で `int main(int, const char *const *);` と決められているように，Haskell でも *main* 関数の型はあらかじめ決められている．その型は $\mathbf{IO}\ [\![\mathbf{Int}]\!]$ である．

13.3 プログラムの本質

プログラムに与えられた引数やファイルを x とし，プログラムの出力を y とすると，副作用のないプログラムの本質は

$$y = \mathrm{main}\, x \tag{13.2}$$

という main 関数であり，それを

$$\mathrm{main} = f_1 \centerdot f_2 \centerdot \cdots \centerdot f_n \tag{13.3}$$

と部分関数に分解することがプログラマの能力である．副作用のないプログラムとは，例えば標準入力から文字列を読み取り，標準出力へ文字列を書き出す UNIX のフィルタプログラムのようなものである．例えば `wc` という UNIX プログラムは，標準入力に与えられた文字列から行数，単語数，文字数を数えて，その数値を標準出力へ書き出す副作用のないプログラムである．

副作用のない関数を合成するのはわけのないことだ．もし Python ならば

Python
```
y = f2(f1(x))
```

のように関数を入れ子にしても良いし，読みづらければ途中経過を一時変数にして

Python
```
y1 = f1(x)
y2 = f2(y1)
```

としてもよい．副作用のない関数の場合，これが関数合成の規則である．

プログラムが副作用を持つ場合は，プログラム自身が IO モナドであるため，関数へと分解できないのであった．副作用を持つプログラムは

$$w = main \heartsuit \langle x \rangle \tag{13.4}$$

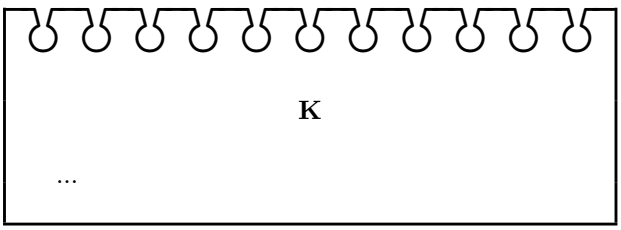
という $main$ 関数であり，それを

$$main = f_1 \heartsuit f_2 \heartsuit \cdots \heartsuit f_n \tag{13.5}$$

と部分関数に分解することがプログラマの能力となる．

13.4　余計な話：パイプライン演算子*

13.5　この章のまとめ*



第 14 章

インタープリタで遊ぶ*

第 15 章

新しい型の導入*

15.1 型推論*

第 16 章

IO*

16.1 ダークサイド*

第 17 章

データ構造*

第 18 章

例外*

第 19 章

状態*

第 20 章

非決定性*

第 21 章

乱数*

第 22 章

...

第 23 章

...

第 24 章

プリミティブ型*

24.1 余計な話：アンボックス化型*

第 III 部

補講

第 25 章

代数的構造

この章では「代数的構造」を見ていくことにする．代数的構造とは，四則演算のような数に関する基本的な性質を抽象化していくことで，数の背後にある基本的なメカニズムを抽出したものである．代数的構造はあらゆるプログラミング言語に明示的，あるいは非明示的に見られる要素である．

25.1 数

これから各種の **代数的構造** を見ていくことにする．代数的構造と言っても，身構える必要はない．それは，我々プログラマが日々接している概念に，共通した名前を与えたにすぎない．

まず最初に，我々にとって一番身近な代数的構造である **数** を見てみよう．数の代表例は **実数** であるから，実数を例にとって考えてみよう．実数全体の集合を $\mathbb{R}$ で表すことにする．また任意の実数を x, y, z で表すこととする．このことを数学者は $x, y, z \in \mathbb{R}$ と書くが，本書ではこれまで通り

$$x, y, z :: \mathbb{R} \tag{25.1}$$

と表すことにする．

以下に実数の備える代数的性質を列挙する．どれも当たり前のことに見えるが，ひとつひとつ見ていこう．ここで $x, y, z :: \mathbb{R}$ とする．

実数の性質 1. **加法の全域性**　任意の x と任意の y の **加法**（足し算）の結果すなわち **和** $x + y$ は $\mathbb{R}$ の元すなわち実数である．演算の結果が同じ集合の元になることを **全域性** と呼ぶ．

実数の性質 2. **加法の結合性**　任意の x, y, z について

$$(x + y) + z = x + (y + z) \tag{25.2}$$

である．これを加法の **結合性**（結合律）と呼ぶ．

実数の性質 3. **零元（加法単位元）の存在**　特別な実数 $0 :: \mathbb{R}$ があり

$$0 + x = x + 0 = x \tag{25.3}$$

である．この 0 は足し算の **単位元** である．**零元** または **加法単位元** と呼ぶこともある．単位元が存在することを単位律と呼ぶこともある．

実数の性質 4. **負元（加法逆元）の存在**　任意の x に対して $-x :: \mathbb{R}$ があり

$$-x + x = 0 \tag{25.4}$$

である．この $-x$ は x の加法の **逆元** である．**負元** または **加法逆元** と呼ぶこともある．逆元が存在することを消約律と呼ぶこともある．

実数の性質 5. **加法の可換性**　任意の x, y について

$$x + y = y + x \tag{25.5}$$

である．このことを加法の **可換性**（可換律）と呼ぶ．

実数の性質 6. **乗法**　任意の x と任意の y の **乗法**（掛け算）の結果すなわち **積** $x * y$ は $\mathbb{R}$ の元すなわち実数である．

実数の性質 7. **乗法の結合性**　任意の x, y, z について

$$(x * y) * z = x * (y * z) \tag{25.6}$$

である．

実数の性質 8. **単位元の存在**　特別な実数 $1 :: \mathbb{R}$ があり

$$1 * x = x * 1 = x \tag{25.7}$$

である．この 1 を乗法の単位元または **乗法単位元** と呼ぶ．

実数の性質 9. 逆元の存在　任意の x に対して $x^{-1} :: \mathbb{R}$ があり

$$x^{-1} * x = 1 \tag{25.8}$$

である．この x^{-1} は x の乗法の逆元である．**乗法逆元** と呼ぶこともある．ただし性質 11 で述べる通り，加法単位元については逆元がなくても良い．

実数の性質 10. 乗法の可換性　任意の x, y について

$$x * y = y * x \tag{25.9}$$

である．このことを乗法の可換性と呼ぶ．

実数の性質 11. 加法単位元の乗法逆元　加法単位元に対する乗法の逆元は存在しなくても良い．（つまり 0^{-1} のことは考えなくて良い．）

実数の性質 12. 分配律　加法と乗法が混在する場合

$$(x + y) * z = (x * z) + (y * z) \tag{25.10}$$

と乗法を **分配** する．

以上が実数の代数的性質の全てである．我々がよく使う引き算，割り算は数学上はシンタックスシュガーである．

上述の 12 個の条件が当てはまる数には **有理数** や **複素数** がある．この 12 個の性質をまとめて，数学では **体** と呼ぶ．

体の要素は，集合 $\mathbb{K}$，二項演算子 $+$，二項演算子 $+$ の単位元 0，二項演算子 $+$ の逆元生成演算子 $-$，もう一つの二項演算子 $*$，二項演算子 $*$ の単位元 1，二項演算子 $*$ の逆元生成演算子 $^{-1}$ であるから，体はそれらを列挙して $\left(\mathbb{K}, +, 0, -, *, 1, ^{-1}\right)$ と表現する．

体の性質から言えることを一つ紹介しよう．これから

$$z \uparrow n = \underbrace{z * \cdots * z}_{n} \tag{25.11}$$

なる二項演算子 $\uparrow$（**クヌースの矢印**）を使う．ここに z を体の元，n を自然数とした．さて $z \uparrow 2$ は

$$z \uparrow 2 = z * z \tag{25.12}$$

であるから，いま $z = x + y$ とすると

$$\begin{aligned}
z \uparrow 2 &= z * z && (25.13)\\
&= z * (x + y) && (25.14)\\
&= z * x + z * y \text{ —分配律} && (25.15)\\
&= (x + y) * x + (x + y) * y && (25.16)\\
&= x * x + y * x + x * y + y * y \text{ —分配律} && (25.17)\\
&= x \uparrow 2 + x * y + y * x + y \uparrow 2 && (25.18)
\end{aligned}$$

となり

$$(x + y) \uparrow 2 = x \uparrow 2 + x * y + y * x + y \uparrow 2 \tag{25.19}$$

を得る．式 (25.19) は体の性質だけを使って導いた関係なので，実数だけでなく有理数や複素数にもそのまま使える．実際には式 (25.19) は体の性質のうち分配律だけを使っているので，体以外にも応用が利く式でもある．[*1]

25.2　群

体の性質を若干緩めたい場合がある．さもなければ，**整数**，**正方行列**，**クォータニオン**（四元数），**論理値**，**ベクトル**，ベクトルの **変換**，集合から集合への **写像** と言った重要な概念が数の概念からこぼれてしまうからである．例えば整数の掛け算の逆元は（単位元の逆元を除いて）整数の中には存在しないし，正則行列やクォータニオンの場合は掛け算が可換ではない．

だいたいどの辺まで制約を緩めたものを数の仲間に入れるかというのは見解の分かれるところでもあるが，体から性質 9（乗法の逆元），性質 10（乗法の可換性），性質 11（加法単位元の乗法逆元）を取り除いたものを **環** と呼び，環の性質を持つものを数の仲間に入れることが一般的である．環の性質を持つものは，体である実数，有理数，複素数に加えて，整数，正方行列，クォータニオン，論理値などがある．

制約を少しずつ緩める代わりに，制約をその構成要素に分解するほうがさらなる応用が利きそうである．体には二つの二項演算子 $+$ と $*$ が登場した．その片方にのみ注目してみたらどうなるだろう．それがこの節で取り上げる **群** である．

[*1] Knuth の矢印 (↑) は Haskell では演算子 ^ として提供されている．

表 25.1 代表的な代数的構造の性質 (1)

代数的構造	$+$	$+$ の単位元	$+$ の逆元	$*$	$*$ の単位元	$*$ の逆元
体	可換	あり	あり	可換	あり	あり
環	可換	あり	あり	非可換	あり	なし

形式的に体と環の性質を並べたものが表 25.1 である．これを見ると，各演算子について「可換・単位元あり・逆元あり」の組み合わせが二つペアになったもの（体）か，「可換・単位元あり・逆元あり」の組み合わせと「非可換・単位元あり・逆元なし」の組み合わせがペアになったもの（環）があることがわかる．

いま集合 $\mathbf{G}$ があり，$x, y, z :: \mathbf{G}$ であるとし，二項演算子を $\bigstar$ と書くことにして，体の性質の前半分を書き下してみよう．

性質 1. 任意の x と任意の y の演算の結果 $x \bigstar y$ は $\mathbf{G}$ の元である．

性質 2. 任意の x, y, z について

$$(x \bigstar y) \bigstar z = x \bigstar (y \bigstar z) \tag{25.20}$$

である．

性質 3. 特別な元 $\emptyset :: \mathbf{G}$ があり

$$\emptyset \bigstar x = x \bigstar \emptyset = x \tag{25.21}$$

である．

性質 4. 任意の x に対して $\ominus x :: \mathbf{G}$ があり

$$\ominus x \bigstar x = \emptyset \tag{25.22}$$

である．

性質 5. 任意の x, y について

$$x \bigstar y = y \bigstar x \tag{25.23}$$

である．

このような性質が満たされる時，組み合わせ $(\mathbf{G}, \bigstar, \emptyset, \ominus)$ を **可換群** または **加群** と呼ぶ．この可換群が最初の構成要素「可換・単位元あり・逆元あり」の正体である．

例えば $(\mathbb{R}, +, 0, -)$ は可換群である．整数全体の集合を $\mathbb{Z}$ とすると $(\mathbb{Z}, +, 0, -)$ も可換群である．また，集合 $\mathbb{R}$ から 0 だけを取り除いた集合を $\mathbb{R} \setminus 0$ とするとき $\left(\mathbb{R} \setminus 0, *, 1, ^{-1}\right)$ も可換群である．

可換群は代表的な代数的構造のひとつであり，他にも数学のあちこちに顔を出している．例えば回転角を t とする二次元の回転変換を R_t として，回転変換 R_t すべてからなる集合 $\mathbf{R}$ を考えてみよう．回転の合成を $\cdot$ で表すとすると

$$R_{t_1} \cdot R_{t_2} = R_{(t_1+t_2)} \tag{25.24}$$

であるから，回転を合成した結果も回転である．また式 (25.24) から

$$R_{t_1} \cdot (R_{t_2} \cdot R_{t_3}) = (R_{t_1} \cdot R_{t_2}) \cdot R_{t_3} \tag{25.25}$$

であるから，回転変換は結合性も満たしている．

次にに回転変換に単位元があるかどうか調べてみよう．回転しない変換は **恒等変換** とも言い，しばしば I で表す．何もしない回転変換は 0 度の回転であるから $I = R_0$ である．このとき式 (25.24) から

$$I \cdot R_t = R_t \cdot I = R_t \tag{25.26}$$

であるから，I は回転変換の単位元であると言える．

最後に回転変換に逆元があるかも調べてみよう．t 回転の逆は明らかに $-t$ であるから

$$R_{-t} \cdot R_t = R_t \cdot R_{-t} = I \tag{25.27}$$

が成り立つ．そこで

$${R_t}^{-1} = R_{-t} \tag{25.28}$$

として R_t の逆元 ${R_t}^{-1}$ を定義することができる．

このように，組み合わせ $\left(\mathbb{R}, \cdot, I, ^{-1}\right)$ も可換群である．（回転 R_t 全体の集合 $\mathbf{R}$ が群を形成することは，パラメタ t が所属する実数全体の集合 $\mathbb{R}$ が群を形成することに大いに頼っている．この部分を詳細に調べるとリー群という美しい代数的構造が見つかる．）

回転されるものをベクトルと呼ぶ．ベクトルや回転変換の実装方法はいくつかあり，例えば第 1 座標値を u，第 2 座標値を v としたときにベクトル $\vec{p}$ を

$$\vec{p} = \begin{bmatrix} u \\ v \end{bmatrix} \tag{25.29}$$

と表すことにしよう．矢印は変数 p がベクトルであることを忘れないようにするための飾りである．このとき，回転変換は

$$R_t = \begin{bmatrix} \cos t & -\sin t \\ \sin t & \cos t \end{bmatrix} \tag{25.30}$$

と行列で表すことになり，回転後のベクトル $\vec{p'}$ は

$$\vec{p'} = R_t * \vec{p} \tag{25.31}$$

となる．ここに演算子 $*$ は行列の積である．また，この場合変換の合成 $\cdot$ は行列積 $*$ となる．

他にもベクトルを複素数で表現する方法もある．いま $\vec{p}$ を

$$\vec{p} = u + Iv \tag{25.32}$$

と表して，回転変換 R_t を

$$R_t = \cos t + I \sin t \tag{25.33}$$

とすると，回転後の $\vec{p'}$ はやはり

$$\vec{p'} = R_t * \vec{p} \tag{25.34}$$

と書ける．ここに演算子 $*$ は複素数の積である．また，この場合変換の合成 $\cdot$ も複素数積 $*$ となる．

任意次元のベクトル全体からなる集合を $\mathbf{V}$ として，零ベクトルを $\vec{0}$ で表すことにしよう．ここでも矢印はベクトルであることを忘れないようにするための飾りである．ベクトル同士の加算を二項演算子 $+$ で表し，向きを反転させた逆ベクトル作る演算子を $-$ とすると，組み合わせ $\left(\mathbf{V}, +, \vec{0}, -\right)$ もまた可換群である．

可換群の性質のうち最初の 4 項目だけを満たすものを群と呼ぶ．可換群は群の特別な場合である．現代の数学では $x \bigstar y \neq y \bigstar x$ のように演算子の前後を入れ替えると結果が異なるような演算をよく取り扱うので，一般の群は可換群よりもよく取り上げられ，それ故より短い名前が付けられている．

もう一度組み合わせ $(\mathbf{G}, \bigstar, O, \ominus)$ が群である条件を少し緩め，逆元が存在しなくても良い「緩やかな群」を考えてみる．この「緩やかな群」のことを **単位的半群** または **モノイド** と呼ぶ．これが構成要素「非可換・単位元あり・逆元なし」の正体である．

表 25.2　代表的な代数的構造の性質 (2)

代数的構造	★	★ の単位元	★ の逆元
可換群	可換	あり	あり
群	非可換	あり	あり
単位的半群	非可換	あり	なし
半群	非可換	なし	なし

単位的半群の性質は次の三つである．ただし x, y, z が集合 $\mathbf{M}$ の元であるとする．

単位的半群の性質 1.　任意の x と任意の y の演算の結果 $x \bigstar y$ は $\mathbf{M}$ の元である．

単位的半群の性質 2. **結合性**　任意の x, y, z について

$$(x \bigstar y) \bigstar z = x \bigstar (y \bigstar z) \tag{25.35}$$

である．

単位的半群の性質 3. **単位元の存在**　特別な元 $\emptyset :: \mathbf{M}$ があり

$$\emptyset \bigstar x = x \bigstar \emptyset = x \tag{25.36}$$

である．

このとき，組み合わせ $(\mathbf{M}, \bigstar, \emptyset)$ が単位的半群である．可換群とこの単位的半群を組み合わせたのが環，可換群二つを組み合わせたのが体であった．

なお，これまで単位元の定義として

$$\emptyset \bigstar x = x \bigstar \emptyset = x \tag{25.37}$$

を掲げているが，厳密には単位元は

$$\emptyset_{\text{Left}} \bigstar x = x \bigstar \emptyset_{\text{Right}} = x \tag{25.38}$$

のように，**左単位元** と **右単位元** を区別しても良い．

単位的半群の性質からさらに性質 3 を消したものを **半群** と呼ぶ．可換群，群，単位的半群，半群を一覧にしたものを表 25.2 に掲げる．

25.3　圏

これまでは集合の元同士に対する二項演算を考えてきた．集合 $\mathbf{M}$ が単位的半群であるとき，集合 $\mathbf{M}$ の元

$x, y :: \mathbf{M}$ に対して $x \bigstar y :: \mathbf{M}$ であった．見方を変えると，演算子 $\bigstar$ とは集合 $\mathbf{M}$ の元 2 個から出発して，集合 $\mathbf{M}$ の元 1 個へとジャンプさせる **写像** であると言える．これを

$$\bigstar :: (\mathbf{M} \times \mathbf{M}) \rightarrowtail \mathbf{M} \tag{25.39}$$

と書く．ここに $\mathbf{X} \times \mathbf{Y}$ は集合 $\mathbf{X}$ と集合 $\mathbf{Y}$ の **直積集合** である．直積集合と元の集合はもはや別な集合であることに注意しよう．

写像は $\mathbf{M} \times \mathbf{M} \rightarrowtail \mathbf{M}$ に限ったもの出はなく，集合 $\mathbf{M}$ から集合 $\mathbf{M}$ への写像 $\star$ ただし

$$\star :: \mathbf{M} \rightarrowtail \mathbf{M} \tag{25.40}$$

があっても良い．実はこれまでにも登場した逆元を作る演算子はまさに $\mathbf{M} \rightarrowtail \mathbf{M}$ という写像である．

またベクトルには，実数倍や回転といった写像がある．これらは実数のパラメタを一つとるので，ベクトル全体の集合を $\mathbf{V}$，実数全体の集合を $\mathbb{R}$ として，実数のパラメタを r としたときに

$$\star_r :: (\mathbb{R} \times \mathbf{V}) \rightarrowtail \mathbf{V} \tag{25.41}$$

と書ける．例えば回転の場合は $\star_r = R_r$ である．

このようにとある集合（例えば $\mathbf{M} \times \mathbf{M}$ や $\mathbb{R} \times \mathbf{V}$）から異なる別な集合（例えば $\mathbf{M}$ や $\mathbf{V}$）へという写像を一般化するとどうなるだろうか．いま，集合 $\mathbf{X}$ から集合 $\mathbf{Y}$ への写像 f があり，集合 $\mathbf{Y}$ から集合 $\mathbf{Z}$ への写像 g があるとする．すなわち

$$f :: \mathbf{X} \rightarrowtail \mathbf{Y} \tag{25.42}$$
$$g :: \mathbf{Y} \rightarrowtail \mathbf{Z} \tag{25.43}$$

があるとする．また写像同士を二項演算子 $\circledcirc$ で **合成** できるものとする．例えば f と g の合成写像は $\mathbf{X}$ を出発点に $\mathbf{Y}$ を経由して $\mathbf{Z}$ へと行くので

$$f \circledcirc g :: \mathbf{X} \rightarrowtail \mathbf{Z} \tag{25.44}$$

と書ける．ここで合成演算子は結合性を満たすものとしておこう．

さらに

$$I_{\mathbf{X}} :: \mathbf{X} \rightarrowtail \mathbf{X},\ I_{\mathbf{Y}} :: \mathbf{Y} \rightarrowtail \mathbf{Y},\ I_{\mathbf{Z}} :: \mathbf{Z} \rightarrowtail \mathbf{Z} \tag{25.45}$$

という写像もあるとしよう．ここで

$$I_{\mathbf{Y}} \circledcirc f = f \circledcirc I_{\mathbf{X}} \tag{25.46}$$

とすると，写像 $I_{\mathbf{X}}$ と写像 $I_{\mathbf{Y}}$ はそれぞれ写像の合成演算子 $\circledcirc$ に対して単位元のように振る舞う．写像 g については

$$I_{\mathbf{Z}} \circledcirc g = g \circledcirc I_{\mathbf{Y}} \tag{25.47}$$

であるとする．このような写像 $I_{\mathbf{X}}, I_{\mathbf{Y}}, I_{\mathbf{Z}}$ を **恒等写像** と呼ぶ．

集合 $\mathbf{X}$，集合 $\mathbf{Y}$，集合 $\mathbf{Z}$ の集合 $\mathbf{C} = \{\mathbf{X}, \mathbf{Y}, \mathbf{Z}\}$ と，写像 f と写像 g の集合 $\mathbf{P} = \{f, g\}$ と，写像合成演算子 $\circledcirc$ と，恒等写像の集合 $\mathbf{I}$ の組み合わせ $(\mathbf{C}, \mathbf{P}, \circledcirc, \mathbf{I})$ を **圏** と呼ぶ．写像合成演算子 $(\circledcirc)$ と恒等写像の集合 $(\mathbf{I})$ は自明であるためしばしば省略され，組み合わせ $(\mathbf{C}, \mathbf{P})$ を圏とする書き方もよくされる．

圏を考えるとき，変換や写像は全て **射** と呼ぶ決まりである．

いまある単位的半群 $(\mathbf{M}, \bigstar, O)$ があるとする．集合 $\mathbf{C}$ を $\mathbf{M}$ 及び $\mathbf{M} \times \mathbf{M}$ を元とする集合すなわち

$$\mathbf{C} = \{\mathbf{M}, \mathbf{M} \times \mathbf{M}\} \tag{25.48}$$

とし，集合 $\mathbf{P}$ を $\bigstar$ のみを元とする集合すなわち

$$\mathbf{P} = \{\bigstar\} \tag{25.49}$$

とすると，組み合わせ $(\mathbf{C}, \mathbf{P})$ は圏になっている．

25.4 余計な話：束*

25.5 この章のまとめ

	Totality	Associativity	Identity	Divisibility	Commutativity
Semicategory		✓			
Category		✓	✓		
Groupoid		✓	✓	✓	
Magma	✓				
Quasigroup	✓			✓	
Loop	✓		✓	✓	
Semigroup	✓	✓			
Monoid	✓	✓	✓		
Group	✓	✓	✓	✓	
Abelian Group	✓	✓	✓	✓	✓

表 25.3　代数的構造

	全域性	結合性	単位律	消約律	可換性
半圏 (semicategory)		✓			
圏 (category)		✓	✓		
亜群 (groupoid)		✓	✓	✓	
マグマ (magma)	✓				
擬群 (quasigroup)	✓			✓	
ループ (loop)	✓		✓	✓	
半群 (semigroup)	✓	✓			
モノイド (monoid)	✓	✓	✓		
群 (group)	✓	✓	✓	✓	
可換群 (Abelian group)	✓	✓	✓	✓	✓

第 26 章

ラムダ*

26.1 条件式

条件式は一種のシンタックスシュガーである．次のように関数 $t, f,$ if を定義すると，それぞれ真，偽，条件分岐のように振る舞う．

$$t = \backslash xy \to x \tag{26.1}$$
$$f = \backslash xy \to y \tag{26.2}$$
$$\text{if}\, pxy = pxy \tag{26.3}$$

本当かどうか試してみよう．

$$\text{if}\, txy = txy \tag{26.4}$$
$$= (\backslash xy \to x)xy \tag{26.5}$$
$$= x \tag{26.6}$$
$$\text{if}\, fxy = fxy \tag{26.7}$$
$$= (\backslash xy \to y)xy \tag{26.8}$$
$$= y \tag{26.9}$$

確かに if t は 1 番目の引数だけを，if f は 2 番目の引数だけを残す．これは条件分岐そのものである．

26.2 整数*

26.3 Y コンビネータ*

26.4 余計な話：継続*

26.5 この章のまとめ*

第 27 章

モナドとプログラミング言語*

第 28 章

カリー=ハワード同型対応

28.1 型付きラムダ計算

第 29 章

整理中*

C++14

```
auto lambda_exp
  = [](auto x, auto y) { return x+y; };

auto lambda_exp = [u = 1] { return u; };
```

29.1 オブジェクト指向*

複雑な構造に対する単純な操作について考える．

オブジェクト指向（クラス指向と呼ぶべきだが歴史はそうはならなかった）は，複雑な構造に対する単純な操作をうまく抽象化する．

例えば，複素数（実数に比べれば複雑だ）の足し算（単純だ）は，C89/90 ならば以下のようになる．

優秀な C プログラマならすぐに構造体と関数を導入するだろう．しかし，よりよい方法がある．C++ を使うことだ．（C99 ならば複素数を扱えるが，言語の抽象度が C89/90 よりも高いわけではない．）C++98 ならば次のようにする．

複素数はあらかじめ用意されていたが，プログラマは独自の複雑な構造（行列だとか）を自前で作っておくことが出来る．複雑な構造に対する単純な操作を陽に扱えることは，特に GUI の設計，実装にとっては決定的となる．近代的な GUI は，複雑な構造（ビュー，コントローラ，モデルのそれぞれに当てはまる）の間をメッセージが単純に行き交うモデルであるからである．

（いま述べたのはオブジェクト指向のうちカプセル化についてだけである．オブジェクト指向の本当の力はポリモーフィズムにある．それについては後半で触れる．）

次は，単純な構造に対する複雑な操作について考える．

C++ のような素朴なオブジェクト指向機能だけでは，単純な構造に対する複雑な操作をうまく抽象化できない．次の Scheme のコードを見てもらいたい．

関数（Scheme では手続きと呼ぶ）make-plus-n は「引数に n を足す」という操作を作る．

とすると 5 が印刷される．数値（単純だ）に対する，飢えた足し算演算子（複雑な気分になる）を作ったのだ．C++98 で make-plus-n を作ることは可能であるが，簡潔とは言い難い．準備段階として標準関数オブジェクトクラステンプレート std::plus を使ってみる．

として作ったオブジェクト plus2 は関数風に使える．例えば

は 5 を印字する．（あるいはより簡単に

と書いても同じことである．）もう一段の抽象化が C++ 版の make-plus-n である．

は 5 を印字する．驚くべきことに，優れた C++ プログラマは上のコードをさらさらと書く．

C++0x やアップルの Grand Central Dispatch (GCD) 対応版 C 言語では不格好ながらラムダ抽象が導入される．例えば C++0x では make_plus_n は次のように書けるようになるはずである．

GCD では同様のラムダ式をこう書く．

さて，もう一度複雑な構造に対する単純な操作を振り返ってみよう．

オブジェクト指向の本質は関数のディスパッチである．メジャーなオブジェクト指向言語（C++ を含む）は単一ディスパッチ（第 1 引数が指す型ポインタによって実際の呼び出し先関数が決定される）であるが，LISP 用オブジェクトシステム CLOS（これは LISP 上に構築されている）は複数ディスパッチをサポートする．Clojure における例は「Closer to Clojure: ポリモーフィズム」から見てもらいたい．

もし，オブジェクト指向かクロージャ指向かのどちらかの言語を選べと言われたら，クロージャ指向を取ろう．どちらの言語でも，その言語の上にもう一方のパラダイムを築くことは出来る．ただし，オブジェクト指向言語でクロージャ指向をサポートするのは骨の折れることだ．それに対し，クロージャ指向の言語でオブジェクト指向をサポートするのはたやすい．

—

29.2 余計な話：C によるクロージャの実装*

簡単なクロージャの例として，引数に n を足す関数を生成する関数を C 言語で考える．やりたいことは Scheme で言えば

Scheme
```
(define (make-plus-n n) (lambda (x) (+ n x)))
```

なのだが，レキシカルクロージャを持たない C 言語では自前で変数をラップする必要がある．そこで，こんな構造体を作ってみる．

C
```
struct make_plus_n_context_t {
  int _n;
  int (*_func)
    (const struct make_plus_n_context_t *,
    int);
};
typedef struct make_plus_n_context_t
  MAKE_PLUS_N_CONTEXT_T;
```

次に，足し算関数の実体を用意しておく．

C
```
static int plus_n(
  const MAKE_PLUS_N_CONTEXT_T *context,
  int x) {
    return context->_n + x;
}
```

最後に，関数 make_plus_n を定義する．

C
```
MAKE_PLUS_N_CONTEXT_T *make_plus_n(int n) {
  MAKE_PLUS_N_CONTEXT_T *context;
  context = (MAKE_PLUS_N_CONTEXT_T *)
    malloc(sizeof(MAKE_PLUS_N_CONTEXT_T));
  context->_n = n;
  context->_func = plus_n;
  return context;
}
```

この関数は make_plus_n_context_t 構造体をメモリを新たに確保して返す．この構造体から _func を呼んでやるのは，次のようなマクロを用意すると便利である．

C
```
#define FUNC_CALL(context, param)
  ((context)->_func((context), (param)))
```

本マクロは次のように使う．

C
```
int main(void) {
  MAKE_PLUS_N_CONTEXT_T *plus_2
    = make_plus_n(2);
  int y = FUNC_CALL(plus_2, 1);
  printf("%dn", y);
  free(plus_2);
  return 0;
}
```

これが C 版クロージャの一例である．驚くべきことに C ウィザードはこのようなことは朝飯前にやってしまう．

29.3 この章のまとめ*

オブジェクト指向とクロージャ

…

第 30 章

もっと勉強したい人へ*

キーワード一覧

キーワード対訳表

参考文献

[1] Haskell.org; https://www.haskell.org/, as of 2016.

[2] 大角祐介: 新しい Linux の教科書; SB クリエイティブ, 2015.

[3] 金谷一朗: 「新しい Linux の教科書」を Mac で実践する; http://bit.ly/brew-on-mac, 2016.

[4] 金谷一朗: ファンクション + アクション=プログラム; 工学社, 2011.

[5] Miran Lipovaca: "Learn You a Haskell for Great Good: A Beginner's Guide"; No Starch Press, 2011.

[6] Benjamin C. Pierce: Types and Programming Languages; The MIT Press, 2002.

[7] Ryan Lemmer: Haskell Design Patterns; Packt Publishing, 2015.

[8] Bryan O'Sullivan: "Real World Haskell: Code You Can Believe In"; O'Reilly Media, 2008.

[9] Graham Hutton: Programming in Haskell; Cambridge University Press; 2007.

[10] Richard Bird: Thinking Functionally with Haskell; Cambridge University Press, 2014.

[11] Paul Hudak, John Peterson, Joseph Fasel: A Gentle Guide to Haskell; https://www.haskell.org/tutorial/index.html, 1999.