第4章 モナド

モナド (monad) は、Haskell（あるいは他の関数型言語）で破壊的代入（変数の値など状態を変更すること）や入出力のような、他の言語では“副作用” (side effect) として実現される特徴を扱うための手法である。

もともとは数学のカテゴリー理論（圏論）で使われている言葉を借用したものであるが、Haskell で使用するときには、背景となるカテゴリー理論のことを知っている必要はない。

4.1 参照透明性と副作用

純粋な関数型言語には、式は値を表すためだけのものであり、「変数の出現はその定義（変数 = 式）の右辺の式で置き換えても全体の意味は変わらない」という性質がある。このような性質を＿＿＿＿＿＿(referential transparency) と呼び、プログラムの“意味”を考察していく上でとても重要な性質である。（参照透明性のおかげで、帰納法などによる証明が容易になる。） 一方、副作用は、式がその＿＿＿＿＿＿＿＿＿＿＿＿のことである。式が副作用を持ちうるということは、その言語では参照透明性が成り立たない、ということである。Haskell の式は副作用を持っていない。

C のような言語では、入出力や破壊的代入を扱う部分では、副作用を使っているため参照透明性は成り立たない。例えば、C で、

```
c = getchar(); putchar(c); putchar(c);  // C-code ①
```

と

```
putchar(getchar()); putchar(getchar()); // C-code ②
```

とは、`getchar()` が副作用を持っているために異なるプログラムである。（上のプログラムでは 1 文字、下のプログラムではキーボードから 2 文字の入力が消費される。）

一見、参照透明性と入出力や破壊的代入は相容れない性質のように見える。しかし、Haskell では次のように考える。

> 入出力や、変数などの状態の変更は、何らかの“アクション”である。副作用を持つ C 言語などの関数に対応する Haskell の関数は、このアクションを含めて戻り値として返す関数として考える。（つまりアクションを“副”作用ではなく、一人前の値として扱う。） この“アクション”の型は、単なる値とは異なる型を持っていて、アクションに対応している文脈でしか使用することができない。従って、アクションと値は区別され、参照透明性が保たれる。

例えば、 C 言語の `getchar`, `putchar` に対応する Haskell の関数は、`getChar`, `putChar` という名前で、それぞれ次のような型を持っている。

```

```

```
getChar ::

putChar ::
```

この IO という型構成子が入出力に関するアクションの型を表す。また、() はユニット型と読み、意味のある値を持たない型である。 そもそも、C 言語の putchar(getchar()) という式に対応する putChar getChar のような Haskell の式は、型エラーになってしまう。 IO 型に用意されている演算子「>>=」（バインド、と読む） を用いて、次のように書かなければいけない。

```
getChar >>= (\ c -> putChar c)
```

ここで、 (>>=) の型は IO a -> (a -> IO b) -> IO b である。（後述するように、実際にはより一般的な型を持っている。）この式では、 getChar というアクションの結果得られる Char 型の値が、 c という変数に束縛され、続いて putChar c というアクションが実行される。アクションの型 (IO Char) を持つ式から Char 型の値だけを抽出するには、 「>>=」のような演算子を介する必要がある。Char 型の変数 c に対して、c = getChar と書けるわけではないので、型の面からも c が getChar と置き換えられないことが明白である。

Q 4.1.1 C-code ① と C-code ② に対応する Haskell プログラム（の断片）を getChar, putChar, >>=, >> を使って書け。ただし >> は次のように定義される演算子である。

```
m >> n    =    m >>= (\ _ -> n)
```

副作用を持つプログラムは実行する順番により意味が変わるので、並列に実行されるプログラムでは問題となることがある。参照透明性を保ちながらアクションを扱えることは並列実行しても意味が変わらないので理論的に有用であるだけでなく、並列コンピューティングなど実用面でも必要となってきている。

4.2 モナドとは

このような、さまざまな言語で副作用として実現される特徴（入出力・破壊的代入・例外処理・非決定性など）に対するアクションの型が、実は共通の構造を持っていて、同じ構造を持つ演算子で取り扱えることがわかっている。この共通の構造を持つ型（正確には型構成子）のことを _____(monad)という。つまり、モナドはアクションの型である。上記の IO もモナドである。ただし、モナドの中にはアクションという言葉がふさわしくないものもあるので、以下では代わりに"計算" (computation)という言葉を使う。

具体的にはモナドとは

return :: a → M a
(>>=) :: M a → (a → M b) → M b

という型の関数の存在する型構成子 *M* のことである。より厳密には、 *return*, *(>>=)* が

(*return* a) >>= k = k a
m >>= (λ a → *return* a) = m
(m_1 >>= k_1) >>= k_2 = m_1 >>= (λ a → (k_1 a >>= k_2))

の 3 つの等式 (monad law) を満たす必要がある。これらの等式は、直感的には *return* が *(>>=)* の "単位元"であること、 *(>>=)* が結合則を満たすこと、を示している。つまり、数に対する、加算 (+) と加算の単位元 0 の次の等式に相当する。

0 + x = x
x + 0 = x
(x + y) + z = x + (y + z)

直観的には *M a* という型は、＿＿＿＿＿＿または "*a* のアクション"の型を意味する。また、*return* と *(>>=)* の直観的な意味は次の通りである。

- *return a* … 値 *a* をそのまま返す何もしない （アクションが実質ない）計算を表す。
- *m >>= k* … まず、*m :: M a* を計算し、その結果のアクションを除いた値の部分を関数 *k :: a → M b* に渡して、続けて計算することを表す。また、アクションは *m*, *k* のアクションを続けて実行するアクションに相当する。

4.3 モナドと型クラス

モナド *M* の定義は模倣したい副作用により異なるし、付随する関数 *return*, *(>>=)* の定義ももちろん異なる。しかし、 *return*, *(>>=)* は *M* をパラメーターとして、決まった型を持つので、型クラスを用いてアドホック多相な関数として定義することができる。

```
class Monad m where
   return :: a -> m a
   (>>=)  :: m a -> (a -> m b) ->  m b
```

実際は、`Monad` は型ではなく、型構成子に対する型クラス（型構成子クラス, type constructor class）になっている。

4.4 `IO` モナド

`IO` は Haskell の Prelude（最初から読み込まれているライブラリー）に入っているモナドである。型クラス `Monad` のインスタンスである。その定義は一般のプログラマーからは見えない組込みの型となっている。ただし、直観的には次のような定義を持つ型だと考えることができる。

```
-- type IO a ≒ ＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿

-- instance Monad IO where
-- -- 注： 実際には type で定義された型名を instance には使えない。
```

```
--    return a = \ w -> (a,w)
--    m >>= k  = \ w -> let (a,w1) = m w in k a w1
```

ただし RealWorld は、コンピューター全体のファイルなどの状態を表す型である。IO は関数の型だが、引数の RealWorld 型のデータは隠されていて、他の計算で使われないことが保証できるため、破壊的に書き換えて、戻り値の RealWorld 型のデータのために使っても良い、ということである。

IO は Haskell で入出力や状態を効率的に扱うために、処理系で特別な扱いを受ける。主なプリミティブとして、putChar, getChar の他にも、以下のような関数がある。

```
putStr   :: String -> IO () -- 文字列を出力する
putStrLn :: String -> IO () -- 文字列を出力して改行する

getLine     :: IO String    -- 一行を読み込む
getContents :: IO String    -- EOF まで読み込む

print  :: Show a => a -> IO ()
                    -- 文字列に変換して出力し、改行する
readLn :: Read a => IO a
                    -- 一行を読み込んでパースする
```

特に getContents は遅延評価で標準入力の内容を読み込む。言い換えれば、必要にならなければ入力を求めない。これらの他に、標準入出力ではなくファイルに対して入出力するための関数なども用意されている。

ところで、Haskell のプログラムを、GHCi のような対話環境ではなく、 ghc コマンドで実行可能ファイルにコンパイルして実行するとき、C と同じように main という名前の関数から実行が開始される約束になっている。そして main 関数は、IO *t* という形の型（*t* は任意の型）を持たなければいけないことになっている。

つまり次のようなプログラムでは、標準入力のすべてを読み込むことはなく、最初の 1 行のみを出力する。

```
main :: IO ()
main = getContents >>=
       (\ all -> let ls = lines all
                 in putStrLn (head ls))
```

ただし、lines :: String -> [String] は文字列を改行文字で分割する関数、head :: [a] -> a はリストの先頭要素を返す関数である。

たとえば、標準入力から読み込んだ文字列の大文字を小文字に変換したものと小文字を大文字に変換したものを出力するプログラムは以下のようになる。

```
import Data.Char   -- Data.Charモジュールを importする

-- toLower, toUpper :: Char -> Char
--       は大文字・小文字の変換関数
main :: IO ()
main = getContents >>=
         (\ s -> putStr (map toLower s
                         ++ map toUpper s))
```

上記の定義は括弧を省略し、さらにレイアウトを整えて、次のように書くことが多い。

```
main = getContents              >>= \ s ->
       putStr (map toLower s ++ map toUpper s)
```

以降のプログラムでは、このように括弧を省略する。

実は、Haskell では `Monad` クラスに対して、`do` 記法という糖衣構文を用意していて、この関数は次のように書くこともできる。

```
main = do { s <- getContents;
            putStr (map toLower s ++ map toUpper s) }
```

この **do** 式も第 A 章で紹介したレイアウトルールの対象であり、適切にレイアウトすればブレースやセミコロンを省略することができる。

```
main = do s <- getContents
          putStr (map toLower s ++ map toUpper s)
```

そして **do** 式は次のルールで翻訳される。（下線部に翻訳規則を再帰的に適用していく。）

do *{ e }*	⇒ *e*
do *{ e; stmts }*	⇒ *e >>= \ _ ->* ***do*** *{ stmts }*
do *{ x <- e; stmts }*	⇒ *e >>= \ x ->* ***do*** *{ stmts }*
do *{* ***let*** *decls; stmts }*	⇒ ***let*** *decls* ***in*** ***do*** *{ stmts }*

（実は、この翻訳規則はリストの内包表記の翻訳規則とほとんど同じである。）

Q 4.4.1 つぎのようなプログラムを上記の `IO` に関する関数を用いて定義せよ。

ヒント: 入力には `getContents` ではなく、`getLine` を使用せよ。

1. 一行だけ行を読み込んで、それをオウム返しするプログラム
2. 一行だけ行を読み込んで、それを 2 回オウム返しするプログラム
3. 整数を読み込んで、その 3 倍を出力するプログラム
4. 正の整数を読み込んで、その数を辺の長さとして、`*` で三角形を表示するプログラム

なお、**for** 文などの繰返しのための構文はないので、繰り返しが必要なときは再帰関数を定義する必要がある。例えば、入力文字列の各行の先頭の単語だけを大文字にするプログラムは `map` を使って次のように定義することができる。

```
import Data.Char

capitalizeFirst []       = []
capitalizeFirst (xs:xss) = (map toUpper xs) : xss

capitalizeLine line
    = unwords (capitalizeFirst (words line))

main = do all <- getContents
          let xs = lines all
              ys = map capitalizeLine xs
          putStr (unlines ys)
```

問 4.4.2 `IO` モナドを利用して次のようなプログラムを作成せよ。（あとの節で紹介しているライブラリー関数を使用するかもしれない。）

1. 入力文字列を行毎に大文字と小文字に変換して出力する。（例えば、「Hello,↲World↲」 が 「hello,↲HELLO,↲world↲WORLD」になる。ただし、"↲" は改行文字を表すとする。）
2. 入力文字列中の数字 (`'0'` ～ `'9'`) の出現回数をカウントして出力する。
3. 入力文字列中の `'@'` が出現する最初の 10 行だけを出力する。

4.5 参照型と可変な配列型

さらに、`Data.IORef` というモジュールを import することで、破壊的代入が可能な参照 (reference) 型 (`IORef`) を扱う関数も用意される。

```
-- import Data.IORef が必要

newIORef    :: a -> IO (IORef a)      -- 新しい参照の作成
readIORef   :: IORef a -> IO a        -- 参照の読出し
writeIORef  :: IORef a -> a -> IO ()  -- 参照への書込み
```

次に、`IORef` を利用したプログラムの例をあげる。

```
import Data.IORef

foo r = do i <- readIORef r
           if i <= 0 then return ()
             else do
               putStrLn (show i)
               writeIORef r (i - 1)
               foo r

main = do r <- newIORef 10
          foo r
```

ただし、これは無理矢理 `IORef` を使った例であり、`IORef` を使わずに同じ動作をする次のようなプログラムのほうが自然である。

```
{- foo の自然な定義は以下の通り -}
```

```
foo i = if i <= 0 then return ()
        else do putStrLn (show i)
                foo (i - 1)

main = foo 10
```

また Data.Array.IO というモジュールを import することで、破壊的代入が可能な配列を扱う関数も用意される。破壊的代入を使わない配列の操作は、配列をいちいちコピーすることになって極めて非効率的なので、配列には破壊的代入は実質的に不可欠である。（破壊的代入を使わないと、配列を書き換えるたびに新しい配列を用意することになる。）

```
-- import Data.Array.IO が必要

newArray   :: Ix i => (i, i) -> e -> IO (IOArray i e)
-- 添字の最小・最大の組と要素の初期値を受け取って
-- 新しい配列を作る
getBounds  :: Ix i => IOArray i e -> IO (i, i)
-- 配列の添字の最小と最大を返す
readArray  :: Ix i => IOArray i e -> i -> IO e
-- 配列と添字を受け取って、その要素を読む
writeArray :: Ix i => IOArray i e -> i -> e -> IO ()
-- 配列と添字と値を受け取って、要素に書き込む
```

次に、Data.Array.IO を利用したプログラムの例をあげる。ここで randomArray は乱数で初期化した配列を作成する関数、 printArray は配列の要素を出力する関数、accumArray は配列を破壊的に書き換えて足し算をする関数である。

```
{-# LANGUAGE FlexibleContexts #-}
import Data.Array.IO
import System.Random

randomArray ::
    Int -> Int -> Int -> Int -> IO (IOArray Int Int)
randomArray i1 i2 mn mx = do
    arr <- newArray (i1,i2) mn  -- 仮に mn で初期化する
    loop i1 i2 mn mx arr
    return arr
  where
    loop i i2 mn mx a
       | i > i2    = return ()
       | otherwise = do r <- randomRIO (mn, mx)
                        writeArray a i r
                        loop (i + 1) i2 mn mx a

printArray :: Show a => IOArray Int a -> IO ()
printArray arr = do
    (i1,i2) <- getBounds arr
    loop i1 i2 arr
  where
    loop i1 i2 arr | i1 > i2   = return ()
                   | otherwise = do
        a <- readArray arr i1
        putStr (show a)
        putStr ", "
        loop (i1 + 1) i2 arr

accumArray arr = do
    (i1, i2) <- getBounds arr
    loop i1 i2 0 arr
```

```
  where
    loop i1 i2 v arr | i1 > i2 = return ()
                     | otherwise = do
        a <- readArray arr i1
        let w = a + v
        writeArray arr i1 w
        loop (i1 + 1) i2 w arr

main = do arr <- randomArray 1 5 0 100
          printArray arr
          putStrLn ""
          accumArray arr
          printArray arr
          putStrLn ""
```

冒頭の `{-# LANGUAGE FlexibleContexts #-}` はプラグマというコンパイラーに与える指令で、ここでは詳細には立ち入らないが、`Data.Array.IO` を使うときは付けておくことをお勧めする。

このプログラムを実行すると、例えば次のような出力が得られる。

```
56, 36, 76, 44, 10,
56, 92, 168, 212, 222,
```

（ほぼ）対応する C 言語のプログラムは以下のようになる。

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int* randomArray(int n, int min, int max) {
    srand((unsigned)time(NULL));
    int* arr = (int*)malloc(n * sizeof(int));
    for (int i = 0; i < n; i++) {
        arr[i] = min + rand() % (max - min + 1);
    }
    return arr;
}

void printArray(int n, int* arr) {
    for (int i = 0; i < n; i++) {
        printf("%d, ", arr[i]);
    }
}

void accumArray(int n, int* arr) {
    int v = 0;
    for (int i = 0; i < n; i++) {
        v += arr[i];
        arr[i] = v;
    }
}

int main(void) {
    int n = 5;
    int* arr = randomArray(n, 0, 100);
    printArray(5, arr);
    printf("\n");
    accumArray(5, arr);
    printArray(5, arr);
    printf("\n");

    return 0;
```

```
}
```

4.6 続・有用なリスト処理関数

モナドとは直接関係ないが、以前紹介した以外に文字列処理プログラムで有用と思われるリスト処理関数を、以下でいくつか紹介する。これらの関数は `sort` を除いて Prelude（標準ライブラリー）に定義済みである。

```haskell
-- lines は文字列を改行文字のところで分割して、文字列（単語）
-- のリストにする
-- 結果の文字列には改行文字は含まれない
lines           :: String   -> [String]

-- words は文字列を空白文字で分割して、文字列のリストにする
words           :: String   -> [String]

-- unlines は lines の逆操作である
-- 各文字列の末尾に改行文字を追加し、一つに結合する
unlines         :: [String] -> String

-- unwords は words の逆操作である
-- 各文字列を空白で区切って結合する
unwords         :: [String] -> String

-- takeWhile は述語 p とリスト xs を受け取り， p を満たす
-- 要素だけからなる xs の最も長い先頭部分（空の場合もある）を
-- 返す
-- > takeWhile (< 3) [1,2,3,4,-1,0,3,4] == [1,2]
-- > takeWhile (< 9) [1,2,3] == [1,2,3]
-- > takeWhile (< 0) [1,2,3] == []
takeWhile       :: (a -> Bool) -> [a] -> [a]

-- dropWhile p xs は takeWhile p xs の残りの末尾部分
-- を返す
-- > dropWhile (< 3) [1,2,3,4,5,1,2,3]
---           == [3,4,5,1,2,3]
-- > dropWhile (< 9) [1,2,3] == []
-- > dropWhile (< 0) [1,2,3] == [1,2,3]
dropWhile       :: (a -> Bool) -> [a] -> [a]

-- any は述語とリストを受け取り、リストのなかのどれかの要素が
-- 述語を満たすかどうか判定する
any             :: (a -> Bool) -> [a] -> Bool

-- all は述語とリストを受け取り、リストのなかのすべての要素が
-- 述語を満たすかどうか判定する
all             :: (a -> Bool) -> [a] -> Bool

-- head はリスト（空ではいけない）の先頭要素を取り出す
head            :: [a] -> a

-- last は（非空かつ有限な）リストの最後の要素を取り出す
last            :: [a] -> a

-- tail はリスト（空ではいけない）の先頭要素を除いた末尾部分
-- を取り出す
tail            :: [a] -> [a]

-- elem はリストの要素 (element) として含まれているか否か
-- を判定する
-- 通常 x `elem` xs のように中置記法で用いることが多い
elem            :: (Eq a) => a -> [a] -> Bool

```

```haskell
-- import Data.List が必要である
-- sort は安定なソートアルゴリズムの実装である
-- 比較関数を引数にとる sortBy 関数もある
sort            :: (Ord a) => [a] -> [a]

-- sequence はリスト中のアクションを順に実行する。
sequence        :: Monad m => [m a] -> m [a]
sequence []      = return []
sequence (m:ms) = m            >>= \ a ->
                  sequence ms >>= \ as ->
                  return (a:as)

-- mapM はアクションを返す関数をリスト中の全ての要素に
-- 適用する
-- ex.) mapM putStrLn ["ab", "xyzw", "12345"]
mapM         :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f as     = sequence (map f as)
```

Q 4.6.1 次の式の値は何か?

1. `drop 2 [1,4,5]`
2. `takeWhile (< 0) [-1,-3,2,-2,7]`
3. `any (> 0) [-1,-3,-5]`
4. `3 `elem` [2,5,3,1]`

4.7 さらに詳しく知りたい人のために...

文献 (Wadler 1990) は、モナドとリストの内包表記の関係について解説している。

[1] **(Wadler 1990)** Philip Wadler, “Comprehending Monads” ACM Conference on Lisp and Functional Programming, Nice (France), 1990 年 6 月
