第3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 で使用する時には、背景となるカテゴリ理論のことを知っている必要は一切ない。

3.1　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ひじょうに重要な性質である。（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、式がその値以外に持っている “効果” のことである。Haskell の式は副作用を持っていない。

C や ML のような言語では、入出力や破壊的代入を扱う部分では参照透明性は成り立たない。例えば、C で、

```
c = getchar();  putchar(c);  putchar(c);
```

と

```
putchar(getchar()); putchar(getchar());
```

とは、異なるプログラムである。

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、単なる値ではなく何らかの “アクション” であり、単なる値とは異なる型を持っている。この “アクション” の型は、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、Haskell で `getchar`, `putchar` に対応する関数は、それぞれ次のような型を持っている。

```
getChar :: IO Char
putChar :: Char -> IO ()
```

この IO という型構成子がアクションの型である。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>>=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。（実際には、以下で説明するように、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。アクションの型を持つ式から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。c = getChar と書けるわけではないので、文法の上からも型の面からも c が getChar と置き換えられないことが明白である。

3.2 モナドとは

さまざまな言語で副作用とされる特徴（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、同じような演算子で取り扱えることがわかってきた。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言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

$$\begin{aligned} unitM &:: a \to M\ a \\ bindM &:: M\ a \to (a \to M\ b) \to M\ b \end{aligned}$$

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、*unitM, bindM* が

$$\begin{aligned} (unitM\ a)\ `bindM`\ k &= k\ a \\ m\ `bindM`\ (\lambda a \to unitM\ a) &= m \\ m_1\ `bindM`\ (\lambda a \to (m_2\ `bindM`\ (\lambda b \to m_3))) &= (m_1\ `bindM`\ (\lambda a \to m_2))\ `bindM`\ (\lambda b \to m_3) \end{aligned}$$

の 3 つの等式を満たす必要がある。

直観的には $M\ a$ という型は、_____を意味する。また、*unitM* と *bindM* の直観的な意味は次のとおりである。

- *unitM a* $\cdots$ 値 *a* をそのまま返す _____を表す。
- m `bindM` $k \cdots$ まず、$m :: M\ a$ を評価し、その結果の値を関数 $k :: a \to M\ b$ に渡して、続けて評価する計算を表す。

3.3 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 *unitM, bindM* の定義ももちろん異なる。しかし、*unitM, bindM* は *M* をパラメータとして、決められた型を持つので、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                  -- 上の unitM に対応
   (>>=)  :: m a -> (a -> m b) ->  m b -- 上の bindM に対応
```

Monad は型ではなく、型構成子に対する型クラスになっている。

3.4 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）で使用可能なモナド型である。その定義は一般のプログラマからは見えない組込みの型となっている。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。putChar, getChar の他にも、主なプリミティブとして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        -- 一行を入力する
getContents :: IO String        -- EOF まで入力する
```

これらの他に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。

さらに、Data.IORef というモジュールを import することで、次のような参照型（IORef）を扱う関数も用意される。

```
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
```

Haskell で、GHCi のような対話環境ではなく、実行可能ファイルにコンパイルして実行する時、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ことになっているが、main 関数は、IO τ という形の型（τ は任意の型）を持たなければいけないことになっている。

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。

```
module Main where

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)
                               >>= (\ _ -> putStr (map toUpper s)))
```

ラムダ抽象（\ ... -> ... ）は、どんな中置演算子よりも優先順位が低いので、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できる。

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

実は、Haskell では Monad クラスに対して、_____という特別な記法を用意していて、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)
          putStr (map toUpper s)
```

3.5 モナドとインタプリタ

IO は効率のために Haskell の処理系で特別扱いされる組込みのモナドであるが、他の言語の副作用を模倣するためにユーザがモナドを定義することも可能である。以下では、モナドを利用して簡単な命令型プログラミング言語（つまり副作用を持つ言語）のインタプリタを作成する。モナドを用いる利点は、“計算” の意味が変わっても、モナドの標準的な関数 *unitM, bindM* のみを用いている部分は、変更する必要がないところである。簡単な言語からはじめて様々な特徴をもつ言語を定義していく。名前がないと不便なので、これらの対象言語を Util (**U**ntyped **T**iny **I**mperative **L**anguage) と呼び、必要により、Util1, UtilErr, ... などのように接尾語をつけることにする。

説明を簡単にするため、ここではモナドを型クラス Monad のインスタンスとして宣言していない。この宣言を追加して、do 記法を使うプログラムに書き直すことは暗黙の練習問題とする。

実際のインタプリタにはフロントエンド、つまり_____や_____が必要である。字句解析や構文解析の原理は C 言語などの命令型言語で記述するときと変わりはない。再帰下降構文解析（あるいは LR 構文解析）などの方法を利用する。ただ、再帰下降法で構文解析部を記述するときに、モナドとして構成すると便利である。

しかし、ここではこれらフロントエンドの作り方は既知のものとして、構文木ができた状態から話をはじめることにする。

構文木のデータ構造として、次のようなデータ型を使用する。

```
type Decl = (String, Expr)
data Expr = Const Value | Var String | If Expr Expr Expr
          | Let Decl Expr | Letrec Decl Expr
          | Lambda String Expr | App Expr Expr
```

つまり、式 (Expr) とは、定数 (Const) または、変数 (Var) または、if 式 (If)、let 式 (Let)、再帰的な let 式 (Letrec)、ラムダ式 (Lambda)、関数適用 (App) からなる。さらに、あとから必要に応じて構文要素を追加することにする。

具体的な構文としては次のような BNF で定義されていると仮定する。(演算子の優先順位なども適切に宣言されているとする。)

Expr → *Const* | *Var* | (*Expr*)
| `if` *Expr* `then` *Expr* `else` *Expr* | `letrec` *Decl* `in` *Expr* | `let` *Decl* `in` *Expr*
| \ *Var* -> *Expr* | *Expr Expr* | *Expr* + *Expr* | *Expr* * *Expr* | ···(他の中置演算子)···

そして、次のような関数も既に定義されているものと仮定する。

```
myParse :: String -> Expr  -- 字句解析・構文解析の関数
```

Value は "値" の型である。Expr を評価した結果は Value 型になる。ここでは、数値・真偽値・文字列・文字・関数・組を扱えるようにする。

```
data Value = Num Double | Bool Bool | Str String | Char Char
           | Fun (Value -> M Value) | Pair Value Value | Unit
```

インタプリタの実行中に変数の値を知る必要があるため、変数の名前と値の対応をデータとして持っておく必要がある。この対応を表すデータのことを _____ (environment) と呼ぶ。ここでは環境を単に変数名 (String) と値 (Value) の対のリスト[1]として表す。

```
type Env = _________________________
```

この Env 型に対して次のような基本演算を用意しておく。

```
lookup' :: a -> [(a, b)] -> b
lookup' x ((n,v):rest) = if n==x then v else lookup' x rest
```

また、初期環境として+, *などのプリミティブ関数に対する定義を用意しておく。

```
initEnv = [("+", Fun (\ (Num c) -> Fun (\ (Num d) -> Num (c+d)))),
           ("*", Fun (\ (Num c) -> Fun (\ (Num d) -> Num (c*d)))),
           ... ]
```

この他に==, <, <=, >=, >, True, False, pair, fst, snd, toString, ++などのプリミティブの定義を initEnv に追加しておく。toString は数値などを文字列に変換する関数、++は文字列の連接オペレータである。

ここでの目標は次のような型を持つ関数を定義することである。

```
interp :: Expr -> Env -> M Value;  -- インタプリタ本体
```

この型のなかの M がモナドであり、対象言語 Util の "計算" の型を表す。つまり、interp は Value 型の値を返すだけではなく、Value 型の計算を返す必要があるということである。今後、いろいろな特徴を導入していくにつれ、この M の定義が変わることになる。

---

[1] このようなリストを連想リスト (association list, a-list) と言う。現実のインタプリタではもっと効率の良いデータ構造を用いる。

3.6 最初のインタプリタ – Util1

最初のバージョン Util1 では、 M はトリビアルな計算（何もしない計算）としておく。つまり、Util1 は副作用を持たない言語である。

```
type M a = a

unitM :: a -> M a
unitM a = a

bindM :: M a -> (a -> M b) -> M b
m `bindM` k = k m
```

interp の定義は次のようになる。個々の構文要素に対する定義は（Letrec を除いて）比較的平易である。

```
interp :: Expr -> Env -> M Value
interp (Const c) e          = unitM c
interp (Var x)   e          = unitM (lookup' x e)
interp (Let (x, m) n) e     = interp m e `bindM` \ v ->
                              interp n ((x,v):e)
interp (App f x) e          = interp f e `bindM` \ (Fun g) ->
                              interp x e `bindM` \ y ->
                              g y
interp (Lambda x m) e       = unitM (Fun (\ v -> interp m ((x,v):e)))
interp (If m1 m2 m3) e      = interp m1 e `bindM` \ (Bool b) ->
                              if b then interp m2 e
                                   else interp m3 e
interp (Letrec (x, m) n) e =
    mfixU (\ v -> interp m ((x, Fun v):e) `bindM` \ (Fun v) ->
                  unitM v) `bindM` \ v  ->
    interp n ((x, Fun v):e)
```

Letrec に対する定義で使われている mfixU は次のように定義されている関数である。

```
mfixU :: ((a -> M b) -> M (a -> M b)) -> M (a -> M b)
mfixU f = f (\ a -> mfixU f `bindM` \ g -> g a)
```

この定義はトリッキーだが、以下に示す不動点演算子 fix のバリエーションと考えておけば良い。

```
fix  :: (a -> a) -> a
fix f = f (fix f)
```

このとき、

```
interp (myParse "letrec fact = \\ n -> if n==0 then 1 else n*fact(n-1) in fact 5")
       initEnv
```

という式を評価すると Num 120.0 という値になる。

3.7 UtilErr – エラー処理の導入

Util1 ではさまざまなエラーの場合を無視していたので、プログラムのなかに間違い（例えば、宣言していない変数を使用する・0 で割算する・関数以外のものを関数の位置で使用するなど）がある時は、不可解なエラーメッセージを出力したり、予想できないような振舞いをしてしまったりする。そ

こでインタプリターにエラー処理を導入し、プログラム中の間違いに対しては適切なエラーメッセージを表示できるようにする必要がある。

エラーと正常な振舞いを区別するために、次のようにデータ型 Err を定義する。

```
data Err a = ____________ | ____________________
```

正常な振舞いは Success という構成子で表す。エラーの場合は状況を表す文字列をパラメータとする Failure という構成子を用いる。この型に対して次のような補助関数を定義しておく。

```
unitErr :: a -> Err a
unitErr = Success

bindErr :: Err a -> (a -> Err b) -> Err b
(Success a) `bindErr` k = ____
(Failure s) `bindErr` k = ____________
```

m `bindErr` k は、まず m を計算し、その計算が正常終了すれば、その値を k という関数に渡す。しかし、いったん m でエラーが起こると、k は評価されず、_____ことを表している。

UtilErr の計算の型 M はこの Err そのものである。

```
type M a = Err a

unitM :: a -> M a
unitM = unitErr

bindM :: M a -> (a -> M b) -> M b
bindM = bindErr
```

さらに、補助関数を定義しておく。

```
failM :: String -> M a
failM = Failure

lookupM :: String -> Env -> M Value
lookupM x ((n,v):rest)  = if n==x then unitM v else lookupM x rest
lookupM x []            = failM ("Variable: "++x++" is not found")
```

lookupM は Util1 で使用した lookup' に相当する関数であるが、環境中に変数の束縛が見つからなかった場合は、エラーとなるようにしている。failM は _____時に用いる関数である。

初期環境（initEnv）に定義しておく“プリミティブ関数”もエラー処理を利用するように書き換えることができる。例えば、割り算（/）に対応する関数は次のように定義すれば良い。

```
Fun (\ v -> case v of
              Num c -> unitM (Fun (\ w ->
                         case w of
                           Num 0 -> ____________________________
                           Num d -> unitM (Num (c/d))
                           _     -> failM "Number expected"))
              _     -> failM "Number expected")
```

これで引数が数値でない場合や、0 で割ろうとした場合にはエラーが報告される。

interp の定義自体もエラーが起こる可能性のあるところを次のように書き換えておくと良い。

```
interp (App f x) e         = interp f e `bindM` \ g ->
                             case g of
                               Fun h -> interp x e `bindM` \ y ->
                                          h y
                               _     -> failM "Function expected."
interp (If m1 m2 m3) e     = interp m1 e `bindM` \ v ->
                             case v of
                               Bool b -> if b then interp m2 e
                                               else interp m3 e
                               _      -> failM "Boolean expected"
interp (Letrec (x, m) n) e = mfixU (\ v ->
                                    interp m ((x, Fun v):e) `bindM` \ v1 ->
                                    case v1 of
                                      Fun f -> unitM f
                                      _     -> failM "function expected"
                                    ) `bindM` \ v  ->
                             interp n ((x, Fun v):e)
```

例えば App の場合は、関数の位置に出現する式が実際に関数でないときにはエラーを報告する。If の場合は条件式の位置に来る式が真偽値を取らない時にエラーとなる。

なお、UtilErr は（Haskell のような）遅延評価ではなくて、関数の引数を必ず先に評価する ______ ______（strict evaluation）をシミュレートすることに注意する必要がある。例えば“(\x -> 0) (1/0)”のような式は、Haskell では ____が、UtilErr では

____。

Java の try〜catch のように例外を捕捉する構文を導入することも可能である。

```
data Expr = ...  | ______________ | __________
```

BNF には以下の構文を追加する。

$$Expr \quad \rightarrow \quad \ldots \mid \texttt{try}\ Expr\ \texttt{catch}\ Expr \mid \texttt{throw}\ Expr$$

Try *m1 m2* は *m1* を評価し、エラーがなかった場合は、その戻り値を try 式の戻り値とする。しかし *m1* の評価中にエラーが生じた場合は、代わりに *m2* を評価する。Fail *e* は明示的にエラーを発生させる。（Java の throw に対応する。） これらの構文に対する interp の定義は次のようになる。

```
interp (Try m1 m2) e = ____________________________________
interp (Fail m1) e   = interp m1 e `bindM` \ v ->
                       failM (showValue v)
```

（ここで showValue は Value 型のオブジェクトの文字列としての表現を返す Value -> String 型の関数であるとする。）tryM は ____ ______関数である。

```
tryM :: M a -> M a -> M a
tryM (Success v) m = __________
tryM Failure     m = _
```

例えば

```
interp (Try (App (App (Var "/") (Const (Num 1))) (Const (Num 0)))
            (Const (Num 99999)))
       initEnv
```

の値は Success (Num 99999.0) になる。

3.8 UtilST – 状態の導入

Util に更新（代入）可能な状態の概念を導入する。C 言語や Java 言語のように、変数に対して代入を導入することも可能であるが、非本質的な部分が多くなってしまうので、ここで紹介する例では、2 つだけ更新可能な “参照” を導入することにする。2 という数は別に本質的なものではなく、いくつにすることも可能である。

Expr 型には、この参照への代入（SetX, SetY）と参照（GetX, GetY）を追加するとともに、2 つの制御構造 – 繰り返し（While）と逐次実行（Begin）– のための構文を導入しておく。

```
data Expr = ...  | _____ | ____________ | _____ | ____________
          | ____________________ | _________________
```

これに対する具象構文は次のようなものを想定する。

$$
\begin{aligned}
\mathit{Expr} &\rightarrow \ldots \mid \texttt{setX}\ \mathit{Expr} \mid \texttt{getX} \mid \texttt{setY}\ \mathit{Expr} \mid \texttt{getY} \\
&\mid \texttt{while}\ \mathit{Expr}\ \texttt{do}\ \mathit{Expr} \mid \texttt{begin}\ \mathit{ExprSeq} \\
\mathit{ExprSeq} &\rightarrow \mathit{Expr}\ \texttt{end} \mid \mathit{Expr}\ \texttt{;}\ \mathit{ExprSeq}
\end{aligned}
$$

例えば、"begin setX 1; setX (getX+3); getX end"というプログラムを評価すると、_____という結果が得られる。

状態を導入するために、やはり “計算の型”M の定義を変更する必要がある。まず 次の ST を定義する。

```
type MyState = (Value, Value)
type ST a = MyState -> (a, MyState)

unitST :: a -> ST a
unitST a = _________________

bindST :: ST a -> (a -> ST b) -> ST b
m `bindST` k = ____________________________________________________
```

ST は、MyState 型の状態の書換えを表す型である。。unitST a は状態（s）の変更を行なわず、a をそのまま返す計算である。m `bindST` k は、m で変更された状態（s1）をそのまま、k に受渡す計算である。

UtilST での計算の型 M は、ST そのものになる。

```
type M a = ST a

unitM = unitST
bindM = bindST
```

すると、Const, Var, Let, Letrec に対する interp の定義は基本的に 3.6 節のものを変更する必要はない。

SetX や GetX のような新しいプリミティブの解釈は次のように行なう。

```
interp (SetX m1) e = interp m1 e `bindM` \ v ->
                     \ (x, y) -> (x, (v, y))
interp (SetY m1) e = interp m1 e `bindM` \ v ->
                     \ (x, y) -> (y, (x, v))

interp GetX e = _____________________________

interp GetY e = _____________________________
```

SetX, SetY は MyState を書き換え、また GetX, GetY は MyState の値の一部を複製している。

Begin や While などの制御構造に対する定義は次のようになる。

```
interp (Begin [m1]) e   = interp m1 e
interp (Begin (f:fs)) e = interp f e `bindM` \ _ ->

                                ____________________________
                          }
interp (While m1 m2) e = interp m1 e `bindM` \ (Bool b) ->
                            if b then interp m2 e `bindM` \ _ ->

                                                ____________________________
                                  else unitM Unit
```

run という関数を

```
run :: String -> String
run prog = showValue (fst (interp (myParse prog) initEnv (Unit, Unit)))
```

のように定義する。つまり run はプログラムソースを構文解析し、初期環境（initEnv）と初期状態（(Unit, Unit)）を与えて、interp を実行し、その結果を取り出す関数である。この run に対して、

```
run ("let fact = \\ n ->         "++
     "  begin                    "++
     "    setX 1; setY n;        "++
     "    while getY > 0 do begin"++
     "      setX (getX*getY);    "++
     "      setY (getY-1)        "++
     "    end;                   "++
     "    getX                   "++
     "  end in                   "++
     "fact 9                     ")
```

の結果は"362880.0"になる。

問 3.8.1 この *M* の定義では、エラー処理を考慮していない。エラー処理を行なうためには、この *M* の定義にさらに *Err* を合成する必要がある。

```
type M a = MyState -> Err (a, MyState)

unitM :: a -> M a
unitM a = \ s -> unitErr (a, s)

bindM :: M a -> (a -> M b) -> M b
m `bindM` k = \ s0 -> case m s0 of
                        Success (a, s1) -> k a s1
                        Failure err     -> Failure err
```

この *M* の定義に対して、*interp* を定義せよ。

3.8.1 newtype 宣言

上の例では簡単のため型クラスを利用していないが、型構成子 ST を、型構成子クラス Monad のインスタンスとして宣言するために、

```
instance Monad ST where
  return = unitST
  (>>=)  = bindST
```

と書くことはできない。ST は **type** 宣言で導入された型シノニム（型の別名）であり、（パラメータが不足している）型シノニムを型クラスのパラメータとして使うことはできないからである。

この場合、次のように **newtype** 宣言を使うことができる。

```
newtype ST s a = ST { unST :: s -> (a, s) }
-- 以下の定義と同等
-- newtype ST s a = ST (s -> (a, s))
-- unST (ST m) = m

unitST :: a -> ST s a
unitST a = ST (\ s -> (a, s))

bindST :: ST s a -> (a -> ST s b) -> ST s b
(ST m) `bindST` k = ST (\ s0 -> let (a, s1) = m s0 in unST (k a) s1)

instance Monad (ST s) where
  return = unitST
  (>>=)  = bindST
```

newtype 宣言は構成子を 1 種類しか持たない **data** 宣言と見かけは似ているが、構成子（のようなもの）ST および unST は型変換の役割のみを持ち、実行時にコストはかからない。

3.9 UtilIO – 入出力の導入

入出力は、入出力ストリームを状態の一種と考えれば、3.8 節の UtilST と同じ方法で取り扱うことができる。

まず、Expr 型の定義に入出力のプリミティブを追加する。

```
data Expr = Const Value | Let Decl Expr | Var String
          |  ...
          | _____ | _____________
```

計算の型 M の定義は 3.8 節と基本的に同じだが、状態に入力と出力のストリームを表す String 型の部分を追加しておく。

```
type MyState = ((Value, Value), String, String)
```

入出力のプリミティブの定義は次のようになる。

```
getX :: Value -> M Value
getX = ____________________________________________

setX :: Value -> M Value
setX x1 = ____________________________________________

readM :: M Value
readM = ____________________________________________

writeM :: Value -> M Value
writeM v  = ____________________________________________
```

readM は入力ストリームから 1 文字を取出し、writeM v は出力ストリーム o に v を String に変換したものを追加している。

新しいプリミティブ Read と Write に対する interp は次のようになる。

```
interp Read e       = readM
interp (Write m1) e = interp m1 e `bindM` \ v ->
                      writeM v
```

run を次のように定義する。

```
run :: String -> String -> String
run str i = let
    (_, (_, _, o)) = interp (myParse str) initEnv ((Unit,Unit), i, "")
in o
```

すると、次の式

```
run ("let sq = \\ x -> if x>0 then x*x else 0-x*x in "++
     "let r = sq 2 in                                "++
     "write r                                        ") ""
```

の値は、"4.0"になる。

3.9.1 出力のモナドの代替表現

上記の MyState 型では、出力ストリームを表現する文字列の後ろに新しい文字列を追加するのに ++オペレータを使っている。この場合、++オペレータの計算量が左オペランドの長さに比例するので、出力文字列が長くなるにしたがって効率が悪くなる。これを避けるための一つの方法として、次のような定義が考えられる。

```
type Writer a = (a, String -> String)

unitW :: a -> Writer a
unitW a = (a, \ s -> s)

bindW :: Writer a -> (a -> Writer b) -> Writer b
m `bindW` k = let (a, f1) = m; (b, f2) = k a in (b, \ s -> f1 (f2 s))

writeW :: String -> Writer ()
writeW s = ((), (s++))
```

これで++オペレータの左辺が長くなっていくことは避けられる。

3.10 UtilNonDet – 非決定性の導入

次に対象言語 Util に非決定性を導入する。非決定性（nondeterminism）とはプログラムの動作にいくつかの選択肢が存在することを言う。ある選択肢を選んだ結果、計算が失敗する場合がある、その場合は前の選択肢に戻って計算をやり直す（後戻り・バックトラック）。非決定性は探索型のゲームや構文解析プログラムなどで利用できる。非決定性をプリミティブな機能として提供する言語としては、論理型言語の Prolog が有名である。

Util に、新しく Amb と Fail の構文を追加する。

```
data Expr = ...  | ___________________ | ____________
```

Amb *m1 m2* は、*m1* または *m2* のいずれかを評価する計算を表す。Fail は計算の失敗を表す。これに対する具象構文としては、

$$Expr \quad \rightarrow \quad \ldots \mid \texttt{amb}\ Expr\ \texttt{or}\ Expr \mid \texttt{fail}\ Expr$$

を想定する。

非決定性の “計算” の型 M は次のように定義する。

```
data List a = Cons a (List a) | Failure String
type M a = List a

nil :: List a
nil = Failure ""

append :: List a -> List a -> List a
(Cons x xs) `append` ys = Cons x (xs `append` ys)
(Failure s) `append` (Failure "") = Failure s
(Failure s) `append` ys           = ys

unitM :: a -> M a
unitM a = Cons a nil

bindM :: M a -> (a -> M b) -> M b
(Cons x xs) `bindM` k = k x `append` (xs `bindM` k)
(Failure m) `bindM` k = Failure m
```

つまり、List a は基本的には a のリスト型（[a]）だが、空リストに String 型のメッセージが付加されている。unitM と bindM は、リストの内包表記を説明する時に使った

```
unit :: a -> [a]
bind :: [a] -> (a -> [b]) -> [b]
```

に類似の関数である。この M は複数の選択肢を単にリストとして表現している。

計算の失敗は Failure で表される。

```
failM :: String -> M a
failM message = ____________________
```

すると、Amb と Fail に対する interp は次のように定義される。

```
interp (Amb m1 m2) e = ____________________________________________
interp (Fail e1) e   = interp e1 e `bindM` \ v ->
                       failM (showValue v)
```

Amb m1 m2 は m1, m2 の 2 つの式の評価の結果をappend で単に連接しているだけである。

割算のように失敗する可能性のある計算は、上記の failM を用いて定義しておく。

```
Fun (\ v ->
      case v of
        Num c -> unitM (Fun (\ w ->
                    case w of
                      Num 0 -> failM "Division by 0"
                      Num d -> unitM (Num (c/d))
                      _     -> failM "Number expected"))
        _     -> failM "Number expected")
```

ここで、run という関数を

```
run :: String -> String
run prog = showL (interp (myParse prog) initEnv)

showL (Cons a (Cons b cs)) = showValue a ++ " or " ++ showL (Cons b cs)
showL (Cons a _)           = showValue a
showL (Failure str)        = "Failure: " ++ str
```

と定義しておくと、run "(amb 1 or 2) * (amb 3 or 4)"の値は、"3.0 or 4.0 or 6.0 or 8.0"となり、run "(amb 1 or 2) / (amb 0 or 4)"の値は"0.25 or 0.5"となる。失敗している計算については結果に現れていないことに注意する。

なお、run の定義を

```
run1 :: String -> String
run1 prog = showValue (____ (interp (myParse prog) initEnv))

hdL :: List a -> a
hdL (Cons a _) = a
```

のようにして、リストの頭部を取ることにより、成功した最初の計算だけを返すようにすることも可能である。すると、run1 "(amb 1 or 2) / (amb 0 or 4)"の値は "0.25"となる。この場合、実装言語の Haskell が _____を採用しているため、他の選択肢の計算は行なわれない。そのため選択肢が無限個あるような場合でも最初の選択肢の計算結果を出力することができる。

3.11 UtilCont – 接続の導入

Util に break, continue などのジャンプ命令を導入するために、Expr の定義に次のように構成子を追加する。また、goto 文を導入するため、ラベルも導入する。

```
data Expr = Const Value | Let Decl Expr | Var String
          | Lambda String Expr | App Expr Expr | If Expr Expr Expr
          | Letrec Decl Expr
          | GetX | SetX Expr | GetY | SetY Expr | GetZ | SetZ Expr
          | While Expr Expr | ____________________________ | _______
          | ___________ | _______ | _______________ | ________________

type LabeledExpr = (Maybe String, Expr)
```

これに対する具象構文としては、

$$
\begin{array}{rcl}
\mathit{Expr} & \rightarrow & \ldots \mid \texttt{begin}\ \mathit{LabeledExprSeq} \\
 & \mid & \texttt{break} \mid \texttt{continue} \mid \texttt{abort} \\
 & \mid & \texttt{goto}\ \mathit{Var} \mid \texttt{callcc}\ \mathit{Expr} \\
\mathit{LabeledExprSeq} & \rightarrow & \mathit{LabeledExpr}\ \texttt{end} \mid \mathit{LabeledExpr}\ \texttt{;}\ \mathit{LabeledExprSeq} \\
\mathit{LabeledExpr} & \rightarrow & \mathit{Expr} \mid \mathit{Var}\ \texttt{:}\ \mathit{Expr}
\end{array}
$$

を想定する。

次に abort, break, continue などを解釈するために接続の概念をインタプリタに導入する。接続（continuation）のモナドは単独では次のような型になる。

```
type K r a = ________________

unitK :: a -> K r a
unitK a = _____________

bindK :: K r a -> (a -> K r b) -> K r b
m `bindK` k = ________________________________

abortK :: r -> K r a
abortK v = ___________
```

直観的には a -> r が接続（“以後実行すべき操作”）の型になる。unitK a は、_____ _____。m `bindK` k は、_____（\ a -> k a c）を m に渡す。m は最後にこの接続を呼び出すのが普通だが、無視したり、他の接続を呼び出したりすることも可能である。これが、ジャンプなどの命令に対応する。例えば、abortK v は現在の接続を無視して v という値を全体の計算の結果としている。これは計算を途中で中止することに相当する。

実際に UtilCont で使用する計算の型 M では、接続とともに、状態を扱わなければいけないので、この r は状態の変化を表す型 ST () とする。このバージョンでは MyState は Value の三つ組であると定義しておく。（UtilST の時と同様、3 という数に特別の意味はない。）

```
type MyState = (Value, Value, Value)
type ST a = MyState -> (a, MyState)
type Result = ST ()
type M a = _____________

unitM :: a -> M a
unitM a = unitK a

bindM :: M a -> (a -> M b) -> M b
m `bindM` k = m `bindK` k
```

setX など状態に関する関数も、この M の定義にあわせて書き直しておく。

```
failM :: String -> M a
failM message = abortK (\ s -> (Str ("failure: "++message), Unit, Unit))

setXST :: Value -> ST Value
setXST v = \ (x, y, z) -> (Unit, (v, y, z))

setX :: Value -> M Value
setX v = \ c -> setXST v `bindST` c
-- setY, setZ も同様に定義する。

getXST :: ST Value
getXST = \ (x, y, z) -> (x, (x, y, z))

getX :: M Value
getX   = \ c -> getXST `bindST` c
-- getY, getZ も同様に定義する。
```

```
lookupM :: String -> Env -> M Value
lookupM x ((n,v):rest) = if n==x then unitM v else lookupM x rest
lookupM x [] = _______ ("Variable: "++x++" is not found")
```

failM はその時の環境・状態・接続はすべて無視して、“failure: . . . ” というメッセージをプログラムの結果とする。（便宜上、状態の第 1 要素にセットする。）setX v は状態の第 1 要素に v をセットし、Unit と新しい状態を接続に渡す。

interp を書き換える前に、接続を値として扱えるように Value 型を拡張しておく。これは break や continue や goto を実現するために、環境の中に接続を格納しておけると便利だからである。

```
data Value = ...  | ___________________________________
```

Const, Var, Letrec などに対しては interp は変更する必要はない。

Break, Continue は環境の中の break, continue という識別子に束縛されている接続を lookup し、現在の接続は無視して、その接続を起動する。これが “ジャンプ” に相当する。Abort は、接続を無視して現在の状態をそのまま計算の最終結果として返す。

```
interp Break    e = \ c0 -> lookupM "break" e

                                  ___________________________________________
interp Continue e = \ c0 -> lookupM "continue" e

                                  ___________________________________________

interp Abort    e = _____________________________
```

While に対しては、適切な接続を環境に格納する必要があるため、定義がやや複雑になる。

```
interp (While m1 m2) e =
  \ c1 ->
  let e1 = (("break", Cont c1) : e) in
  interp m1 e (\ v ->
  case v of
    Bool b -> if b then
                  let c2 = \ _ -> interp (While m1 m2) e c1 in
                  let e2 = (("continue", Cont c2) : e1) in
                  interp m2 e2 c2
              else c1 (Bool True)
    _      -> fairM "Boolean expected" c1)
```

ここで、c1 は＿＿＿＿＿＿＿＿＿を表す接続で、c2 は＿＿＿＿＿＿＿を表す接続である。これらの接続をそれぞれ、break, continue という識別子に束縛した環境（env）のもとで m2 を評価する。

これまでと同じように run という関数を

```
run :: String -> String
run str = showValue (fst3 (snd (interp (myParse str) initEnv
                          (\ v (_, y, z) -> (v, y, z)) (Unit, Unit, Unit))))

fst3 (a, b, c) = a
```

と定義すると、例えば、

```
run ("let foo = \\ n -> begin              "++  -- C の記法では
     "  setX 1; setY n;                    "++  -- int foo(int n) {
     "  while getY > 0 do begin            "++  --   int r=1;
     "    if getY==10 then break           "++  --   while (n>0) {
     "    else if getY==3 then begin       "++  --     if (n==10) break;
     "      setY (getY-1); continue        "++  --     else if (n==3) {
     "    end else 1;                      "++  --       n--; continue;
     "    setX (getX*getY);                "++  --     }
     "    setY (getY-1)                    "++  --     r=r*n; n--;
     "  end;                               "++  --   }
     "  getX                               "++  --   return r;
     "end in                               "++  -- }
     "foo 9                                ")
```

の結果は、＿＿＿＿＿＿に、最後の行の foo 9 を foo 11 に変えると結果は＿＿＿＿＿になる。

自由な飛躍命令である goto に対する意味を与えるために、まず、“ラベル” を解釈する必要があるが、これに対する interp を定義をここに示すと長くなってしまうので、アイデアのみを示す。

例えば、

```
begin
  l1:  Exprs1 /*  --- この中に goto l1; goto l2; を含むかもしれない */
  l2:  Exprs2 /*  --- この中に goto l1; goto l2; を含むかもしれない */
end
```

のような文の意味は、c_1, c_2 を l1, l2 の接続とすると、

$$
\begin{aligned}
c_1 &= \lambda \_ \to \texttt{interp}\ \mathit{Exprs}_1\ e'\ c_2 \\
c_2 &= \lambda \_ \to \texttt{interp}\ \mathit{Exprs}_2\ e'\ c \\
e' &= e + \{\texttt{l1} \mapsto \texttt{Cont}\ c_1, \texttt{l2} \mapsto \texttt{Cont}\ c_2\}
\end{aligned}
$$

のような式で意味を与えられる。そして、

$$\texttt{interp (l1:}Exprs_1\texttt{ l2:}Exprs_2\texttt{)}\ e\ c = c_1\ \texttt{Unit}$$

と解釈する。この c, e はこの部分全体を解釈する時の接続と環境である。

goto はラベル名に対応する接続を環境から読み出して、現在の接続は無視して、単にその接続を起動する。

```
interp (Goto label) e = \ c0 -> lookupM label e _____________________________
```

例えば、

```
run (                                          -- /* C の記法では */
"begin                                   "++   -- {
"    set 1;                              "++   --    x = 1;
"  l1:                                   "++   -- l1:
"    if get > 100 then goto l2 else Unit; "++   --    if (x>100) goto l2;
"    set (get * 2);                      "++   --    x = x * 2;
"    goto l1;                            "++   --    goto l1;
"  l2:                                   "++   -- l2:
"    get                                 "++   --    return x
"end                                     ")    -- }
```

を評価すると、結果は _____になる。

3.12 call/cc の表現

我々の言語 UtilCont に Scheme の call/cc のようなプリミティブを導入するには、接続を関数として渡すためのコードを用意すれば良い。Callcc に対する interp の定義は次のようになる。

```
callccK :: ((a -> K r b) -> K r a) -> K r a
callccK h = ___________________________________________________
```

```
interp (Callcc m) e = interp m e `bindM` \ g ->
                      case g of
                        Fun f -> ____________________________________
                        _     -> failM ("Callcc: Function expected")
```

callccK の定義中で用いられている k は現在の接続（d）を捨て、キャプチャされた接続（c）を呼び出すという関数である。Callcc に対する interp の定義は、基本的に callccK を呼び出すだけである。

例えば、

```
run (
"let mult = \\ xs -> \\ k -> begin                                                  " ++
"    setX 1; setY xs; setZ  \"\";                                                   " ++
"    while isCons getY do begin                                                     " ++
"      let n = car getY in                                                          " ++
"      if n == 0 then k 0 else                                                      " ++
"        begin setX (getX*n); setY (cdr getY); setZ (getZ ++ \" \" ++ n) end       " ++
"    end;                                                                           " ++
"    getX                                                                           " ++
"  end in                                                                           " ++
```

```
  "let list   = cons 1 (cons 2 (cons 3 (cons 0 (cons 4 (cons 5 nil))))) in     " ++
  "let result = callcc (\\ k -> mult list k)  in                               " ++
  "pair result getZ                                                            ")
```

の結果は (Num 0.0, Str " 1.0 2.0 3.0") となる。

3.13 モナドの組合せ

複数の種類の “計算” を同時に扱う場合、いくつかのモナドを組み合わせる必要があるが、組み合わせ方に普遍的な方法があるわけではない。以下にいくつかの例を挙げる。

まず、基本となるモナドの定義を (一部手直しして) 再掲する。

```
type ST s a = s -> (a, s) -- State Transformer

unitST :: a -> ST s a
unitST a = \ s -> (a, s)

bindST :: ST s a -> (a -> ST s b) -> ST s b
m `bindST` k = \ s -> let (a, s1) = m s in k a s1

tickST :: ST Integer ()
tickST = \ n -> ((), n+1)
```

```
type K r a = (a -> r) -> r -- Continuation

unitK :: a -> K r a
unitK a = \ c -> c a

bindK :: K r a -> (a -> K r b) -> K r b
m `bindK` k = \ c -> m (\ a -> k a c)
```

```
type E a = Either String a -- Error

unitE :: a -> E a
unitE a = Right a

bindE :: E a -> (a -> E b) -> E b
(Right a) `bindE` k = k a
(Left s)  `bindE` k = Left s
```

```
type L a = [a] -- Nondeterminism

unitL :: a -> L a
unitL a = [a]

bindL :: L a -> (a -> L b) -> L b
[] `bindL` k = []
(x:xs) `bindL` k = k x ++ (xs `bindL` k)
```

- State Transformer と Continuation (1)

```
type KST r s a  = K (ST s r) a

unitKST :: a -> KST r s a
unitKST = unitK

bindKST :: KST r s a -> (a -> KST r s b) -> KST r s b
bindKST = bindK

tickKST :: KST r Integer ()
tickKST = \ c -> tickST `bindST` c
```

- State Transformer と Continuation (2)

```
type STK r s a = s -> K r (a, s)

unitSTK :: a -> STK r s a
unitSTK a = \ s c -> c (a, s)

bindSTK :: STK r s a -> (a -> STK r s b) -> STK r s b
m `bindSTK` k = \ s c -> m s (\ (a, s1) -> k a s1 c)

tickSTK :: STK r Integer ()
tickSTK = \ s c -> c (tickST s)
```

- State Transformer と Error (1)

```
type STE s a = s -> E (a, s)

unitSTE :: a -> STE s a
unitSTE a = \ s -> unitE (a, s)

bindSTE :: STE s a -> (a -> STE s b) -> STE s b
m `bindSTE` k = \ s -> m s `bindE` \ (a, s1) -> k a s1
```

- State Transformer と Error (2)

```
type EST s a = s -> (E a, s)

unitEST :: a -> EST s a
unitEST a = \ s -> (unitE a, s)

bindEST :: EST s a -> (a -> EST s b) -> EST s b
m `bindEST` k = \ s -> let (ma, s1) = m s in
                       case ma of
                         Right a -> k a s1
                         Left s  -> (Left s, s1)
```

- State Transformer と Nondeterminism (1)

```
type STL s a = s -> [(a, s)]

unitSTL :: a -> STL s a
unitSTL a = \ s -> [(a, s)]

bindSTL :: STL s a -> (a -> STL s b) -> STL s b
m `bindSTL` k = \ s -> m s `bindL` \ (a, s1) -> k a s1
```

- State Transformer と Nondeterminism (2)

```
type LST s a = s -> ([a], s)

unitLST :: a -> LST s a
unitLST a = unitST (unitL a)

bindLST :: LST s a -> (a -> LST s b) -> LST s b
m `bindLST` k = m          `bindST` \ as ->
                mapST k as `bindST` \ bs ->
                unitST (concat bs)

mapST f []     = unitST []
mapST f (a:as) = f a        `bindST` \ b ->
                 mapST f as `bindST` \ bs ->
                 unitST (b:bs)
```

一部のモナドについては、モナド変換 (Monad Transformer) という形で扱うことも可能である。
(Haskell Libraries のソースから一部抜粋・簡略化)

```
newtype StateT s m a = StateT { runStateT :: s -> m (a,s) }

instance (Monad m) => Monad (StateT s m) where
  return a = StateT $ \ s -> return (a,s)
  m >>= k  = StateT $ \ s -> do
          (a, s') <- runStateT m s
          runStateT (k a) s'
```

```
newtype ContT r m a = ContT { runContT :: (a -> m r) -> m r }

instance (Monad m) => Monad (ContT r m) where
  return a = ContT ($ a)
  m >>= k  = ContT $ \ c -> runContT m (\ a -> runContT (k a) c)
```

```
newtype ErrorT m a = ErrorT { runErrorT :: m (Either String a) }

instance (Monad m) => Monad (ErrorT m) where
  return a = ErrorT $ return (Right a)
  m >>= k  = ErrorT $ do
          a <- runErrorT m
          case a of
             Left l  -> return (Left l)
             Right r -> runErrorT (k r)
```

```
newtype ListT m a = ListT { runListT :: m [a] }

instance (Monad m) => Monad (ListT m) where
  return a = ListT $ return [a]
  m >>= k  = ListT $ do
          a <- runListT m
          b <- mapM (runListT . k) a
          return (concat b)
```

当然ながら、モナド変換を適用する順番により、意味が異なる場合がある。

この章の参考文献

[1] Philip Wadler 「The essence of functional programming」
19th Annual Symposium on Principles of Programming Languages (invited talk), 1992 年 1 月
おもにモナドを用いてインタプリタを構築する方法を解説している。

[2] Philip Wadler 「Monads for functional programming」
Program Design Calculi, Proceedings of the Marktoberdorf Summer School, 1992 年 7–8 月
モナドを用いてパーサを構築する技法の解説がある。

[3] Philip Wadler 「Comprehending Monads」
ACM Conference on Lisp and Functional Programming, Nice (France), 1990 年 6 月
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。