第5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 で使用するときには、背景となるカテゴリ理論のことを知っている必要はない。

5.1　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、____ 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、`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 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” であり、単なる値とは異なる型を持っている。副作用をもつ C 言語などの関数に対応する関数は、このアクションを返す関数として考える。（つまり “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、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 :: _________
putChar :: ___________________
```

この IO という型構成子がアクションの型を表す。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、後述するように、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。

5.2 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

$$\begin{aligned} (unitM\ a)\ `bindM`\ k &= k\ a \\ m\ `bindM`\ (\lambda a.unitM\ a) &= m \\ (m_1\ `bindM`\ k_1)\ `bindM`\ k_2 &= m_1\ `bindM`\ (\lambda a.(k_1\ a\ `bindM`\ k_2)) \end{aligned}$$

の 3 つの等式（monad law）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* ··· 値 *a* をそのまま返す＿＿＿＿＿＿＿＿＿＿を表す。
- *m 'bindM' k* ··· まず、*m* :: *M a* を評価し、その結果の値を関数 *k* :: *a* → *M b* に渡して、続けて評価する計算を表す。

5.3 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                    -- 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。

5.4 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である。型クラス Monad のインスタンスである。ただし、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。（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。）

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（lazy に読み込まれることに注意）
```

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、破壊的代入が可能な参照型（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 のプログラムを、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。

```
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)
```

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、_____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)
```

説明を簡単にするため、以下のプログラム例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。

5.5 モナドと命令型言語の意味

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。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。

以下では、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）を定義し、モナドを利用してその意味を与えることにする。Haskell で意味を与えるということは、結局はラムダ計算で意味を与えることになる。具体的には命令型プログラミング言語から Haskell へのコンパイラを作成する。

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**til: **T**iny **I**mperative **L**anguage)[1]と呼び、必要により、UtilErr, UtilST, UtilCont, ... などのようにバージョンを表す接尾語をつけることにする。いろいろな特徴を導入していくにつれ、その “計算” を表すモナドの定義が変わることになる。

実際のコンパイラ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 Haskell でも 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。再帰下降構文解析法（あるいは LR 構文解析法）などの方法を利用する。(ただし、再帰下降法で構文解析部を記述するときに、後述のようにモナドを利用することができる。)

[1] PHP, GNU などの略語の由来も参照すること。

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。

Util の構文木のデータ構造として、次のような Haskell データ型を使用する。

```
type Decl = (String, Expr)
data Expr = Const Target            -- 定数（Target は後述）
          | Var String              -- 変数
          | If Expr Expr Expr       -- if 文
          | While Expr Expr         -- while 文
          | Begin [Expr]            -- ブロック
          | Let [Decl] Expr         -- let 式（関数定義）
          | Val Decl Expr           -- val 式（変数定義）
          | Lambda String Expr      -- ラムダ式
          | Delay Expr              -- delay 式（後述）
          | App Expr Expr           -- 関数適用
         deriving Show
```

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)、ラムダ式 (Lambda)、関数適用 (App) などからなる。（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* | **while** *Expr* **do** *Expr* | **begin** *Exprs* **end**
| **let** *Decls* **in** *Expr* | **val** *Decl* **in** *Expr*
| \ *Var* -> *Expr* | *Expr Expr*
| *Expr* + *Expr* | *Expr* * *Expr* | ··· （他の中置演算子）···

Exprs → *Expr* | *Expr* ; *Exprs*

Decl → *Var* = *Expr*

Decls → *Decl* | *Decl* ; *Decls*

ここに示されていないが、定数 *Const* と変数 *Var* の字句の定義は Haskell と同じとする。ただし、_（アンダーバー）から始まる変数名はコンパイラ内部で使用するために予約済みとする。

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
```

- “**val** x=2*2 **in val** y=x*x **in** y*y” というソースプログラムは Expr 型のデータとして次のように構文解析される。

```
Val ("x", (App (App times (Const (TLit (Int 2)))) (Const (TLit (Int 2)))))
    (Val ("y", (App (App times (Var "x")) (Var "x")))
         (App (App times (Var "y")) (Var "y")))
```

  ただし、*times* は *に対応する Expr の式である。

- “\ f -> \ x -> f x” という式は、

```
Lambda "f" (Lambda "x" (App (Var "f") (Var "x")))
```

というデータに構文解析される。

- &&, ||は、それぞれ、
  `b1 && b2` ↦ ＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿
  `b1 || b2` ↦ ＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿
  という糖衣構文であるように構文解析されるようにしておく。

問 5.5.1 なぜ、&&や||はプリミティブ関数として定義すると良くないのか?

Target 型はコンパイラのターゲット言語である Haskell（のサブセット）を表現する型である。

```
type TDecl = (String, Target)
data Target = TLit Literal                -- 定数
            | TVar String                 -- 変数
            | TIf Target Target Target    -- if 文
            | Let [TDecl] Target          -- let 式（変数・関数定義）
            | TLambda1 String Target      -- ラムダ式
            | TApp1 Target Target         -- 関数適用
            | TUnitM Target
            | TBindM Target Target
           deriving (Show,Eq)
data Literal = Str String | Int Integer | Frac Rational | Char Char
             deriving (Show,Eq)
```

Util のコンパイラとは次のような型を持つ関数である。

```
comp :: Expr -> Target   -- コンパイラ
```

5.6 抽象構文と具象構文

ところで、上の Util の構文規則は ＿＿＿＿（ambiguous）である。通常は曖昧さを避けるために、

$$
\begin{aligned}
\mathit{Expr} &\rightarrow \mathit{Expr} + \mathit{Term} \mid \mathit{Term} \\
\mathit{Term} &\rightarrow \mathit{Term} * \mathit{Factor} \mid \mathit{Factor} \\
\mathit{Factor} &\rightarrow \mathit{Const} \mid (\ \mathit{Expr}\ )
\end{aligned}
$$

のように曖昧さを避けるために構文規則を工夫する。この後者のように実際に構文解析に用いるための構文を ＿＿＿＿＿＿（concrete syntax）という。

それに対して、いったん構文解析が終了してしまえば、曖昧さを避けるための補助的な仕掛けは必要なくなり、本質的な構造のみを扱えばよい。そのため、前者のような構文規則で十分である。このように構成要素の本質的な関係を記述した構文のことを ＿＿＿＿＿＿（abstract syntax）という。

この章で “構文” と呼んでいるのは、この抽象構文のことである。データ型 Expr の定義は、抽象構文を代数的データ型として直訳したものである。

5.7 コンパイラの定義

comp 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比較的平易である。

```
comp :: Expr -> Target
comp (Const c)          = TUnitM c
comp (Var x)            = TUnitM (TVar x)
comp (Val (x, m) n)     = comp m `TBindM` TLambda1 x
                          (comp n)
comp (Let decls n)      = TLet (map (\ (x, m) -> let TUnitM c = comp m
                                                 in (PVar x, c)) decls)
                                (comp n)
comp (App f x)          = comp f `TBindM` TLambda1 "_f"
                          (comp x `TBindM` TLambda1 "_x"
                          (TApp1 (TVar "_f") (TVar "_x")))
comp (Lambda x m)       = TUnitM (TLambda1 x (comp m))
comp (Delay m)          = TUnitM (comp m)
comp (If e1 e2 e3)      = comp e1 `TBindM` TLambda1 "_b"
                          (TIf (TVar "_b") (comp e2) (comp e3))
comp (While e1 e2)      = TLet [(PVar "_while",  body)] (TVar "_while")
   where body = comp e1 `TBindM` TLambda1 "_b"
                (TIf (TVar "_b") (comp e2 `TBindM` TLambda1 (TVar "_while"))
                                 (TUnitM (TVar "()")))
comp (Begin [e])        = comp e
comp (Begin (e:es))     = comp e `TBindM` TLambda1 (TVar "_")
                          (comp (Begin es))
```

右辺で使われている _f, _x, _b, _while などの識別子は、Util ソースプログラム中に使われている識別子と衝突しないように選んでいる。

comp 関数を理解するために、変換前と変換後をそれぞれ Util と Haskell の文法で記述したのが次の表である。（詳細はソースプログラムを参照すること。）ただし bindM や unitM など、末尾に M が付く関数名は、対象のモナドに応じて適切な名前に置き換えられるものとする。

この表のなかで ソース中で *Italic* フォントで示されている *m, n* などは任意の Util の式で、ターゲット中で *m', n'* のように'（プライム）が付いている式は、その comp による変換後の Haskell の式を表す。なお、delay については内部的に使用されるので、実際には Util のソースプログラムに現れることはない。

ソース（Util）	ターゲット（Haskell）
c （ただし c は定数）	`unitM c`
x （ただし x は変数）	`unitM x`
val x = *m* **in** *n*	*m'* `` `bindM` `` `\ x -> n'`
let f = \ x -> *m* g = \ y -> *n* **in** *n*	**let** f = \ x -> *m'* x = \ y -> *n'* **in** *n'*
f x	*f'* `` `bindM` `` `\ _f ->` *x'* `` `bindM` `` `\ _x ->` `_f _x`
\ x -> *m*	unitM (\ x -> *m'*)
delay *m*	unitM *m'*
if *c* **then** *t* **else** *e*	*c'* `` `bindM` `` `\ _b ->` **if** _b **then** *t'* **else** *e'*
while *c* **do** *t*	**let** _while = *c'* `` `bindM` `` `\ _b ->` **if** _b **then** *t'* `` `bindM` `` `\ _ ->` _while **else** () **in** _while
begin *s*; *t*; *u* **end**	 *s'* `` `bindM` `` `\ _ ->` *t'* `` `bindM` `` `\ _ ->` *u'*

また、Util プログラム中の+, -, * などの二項演算子は、他の関数が unitM (\ x -> ...) という形に変換されること、戻り値はアクションを持たないことから、同名の Haskell 内のオペレータを用いて、それぞれ

```
unitM (\ x -> unitM (\ y -> unitM (x+y)))
unitM (\ x -> unitM (\ y -> unitM (x-y)))
unitM (\ x -> unitM (\ y -> unitM (x*y)))
```

という Haskell の式に置換するようにしておく。すると、comp 関数による他の部分の変換と整合する。

ソース（Util）	ターゲット（Haskell）
⊗ （ただし ⊗ は二項演算子）	unitM (\ x -> unitM (\ y -> unitM (x ⊗ y)))

この comp を用いて、例えば次の Util プログラムを変換[2]すると

```
let fact = \ n -> if n==0 then 1 else n*fact(n-1)
  in fact 9
```

次のような Haskell のプログラムが得られる。

[2]ただし、この fact 関数は副作用を含んでいないので、この変換自体にはあまり意味はない。

```
let fact
      = \ n ->
          ((unitM (\ x -> unitM (\ y -> unitM (x == y))) `bindM`
              \ _f -> unitM n `bindM` \ _x -> _f _x)
             `bindM` \ _f -> unitM 0 `bindM` \ _x -> _f _x)
            `bindM`
            \ _b ->
              if _b then unitM 1 else
                (unitM (\ x -> unitM (\ y -> unitM (x * y))) `bindM`
                   \ _f -> unitM n `bindM` \ _x -> _f _x)
                  `bindM`
                  \ _f ->
                    (unitM fact `bindM`
                       \ _f ->
                         ((unitM (\ x -> unitM (\ y -> unitM (x - y))) `bindM`
                             \ _f -> unitM n `bindM` \ _x -> _f _x)
                            `bindM` \ _f -> unitM 1 `bindM` \ _x -> _f _x)
                           `bindM` \ _x -> _f _x)
                      `bindM` \ _x -> _f _x
  in unitM fact `bindM` \ _f -> unitM 9 `bindM` \ _x -> _f _x
```

これは多くの冗長な部分を含んでいるので、前述の monad law などを利用して単純化すると、次のような式が得られる。

```
let fact
      = \ n ->
          if n == 0 then unitM 1 else
               fact (n - 1) `bindM` \ _x ->
               unitM (n * _x)
  in fact 9
```

5.8 最初のバージョン – Util1

最初のバージョン Util1 では、モナド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 I a = a

unitI :: a -> I a
unitI a = a

bindI :: I a -> (a -> I b) -> I b
m `bindI` k = k m
```

このとき、

```
let fact = \ n -> if n==0 then 1 else n*fact(n-1) in fact 9
```

という Util プログラムをコンパイルして実行すると、同じプログラムを Haskell として実行したときと同じ 362880 という値になる。

5.9 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” X と Y 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。

例えば、

```
begin setX 1; setX (getX ()+3); getX () end
```

という UtilST 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” を定義する必要がある。まず 次の ST を定義する。

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

unitST :: a -> ST s a
unitST a = _____________________

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

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。

プリミティブは次のように定義する。

```
setXST :: x -> ST (x, y) ()
setXST v = \ (x, y) -> ((), (v, y))

setYST :: y -> ST (x, y) ()
setYST v = \ (x, y) -> ((), (x, v))

getXST :: () -> ST (x, y) x
getXST () = ___________________________________

getYST :: () -> ST (x, y) y
getYST () = ___________________________________
```

setXST, setYST は状態を書き換え、また getXST, getYST は状態の値の一部を複製している。

UtilST の setXM, getXM, ... という関数は、そのまま Haskell の setXM, getXM, ... にコンパイルされるようにしておく[3]。

ソース（Util）	ターゲット（Haskell）
setXM *m*	*m*’ `bindM` \ _x -> setXM _x
getXM ()	getXM ()

UtilST プログラム（右は対応する C プログラム）:

[3] このプリント中では、getXM のように “M” という接尾辞を使っているときは、一般のモナドに適用可能な事柄であり、getXST のように “M” 以外の（例えば “ST”）という接尾辞を使っているときは、特定のモナド ST に関する事柄であるものと約束する。つまり、M という接尾辞がついている識別子は実際に使うときに適宜名前を付け替える。

```
let fact = \ n ->
  begin
    setXM 1; setYM n;
    while getYM () > 0 do begin
      setXM (getXM () * getYM ());
      setYM (getYM () - 1)
    end;
    getXM ()
  end
  in fact 9
```

```
int fact(int y) {

  int x = 1;
  while (y > 0) {
    x = x * y;
    y = y - 1;
  }
  return x;
}
```

をコンパイルすると次のような Haskell の関数（一部、見易くするために変数名の変更などをしている）になり、

```
let fact n = setXST 1 `bindST` \ _ ->
             setYST n `bindST` \ _ ->
             (let _while = getYST () `bindST` \ y ->
                           if y > 0 then
                                getXST ()          `bindST` \ x ->
                                getYST ()          `bindST` \ y ->
                                setXST (x*y)       `bindST` \ _ ->
                                getYST ()          `bindST` \ y ->
                                setYST (y-1)       `bindST` \ _ ->
                                _while
                           else unitST ()
                in _while)   `bindST` \ _ ->
             getXST ()
  in fact 9
```

fact 9 を実行する（fst (fact 9 (0, 0))）とその結果は 362880 になる。

階乗の場合、普通に関数的な定義を書いた方が簡潔だが、パラメータの数が多い場合などは、このような命令的な書き方の方が簡潔になる場合もありうる。

この ST 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 ST と（後述する）Maybe の定義を合成する必要がある。次のようなモナドになる。

```
type EST s a = s -> Maybe (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 = \ s0 -> case m s0 of
                          Just (a, s1) -> k a s1
                          Nothing      -> Nothing
```

5.10 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、5.9 節の UtilST と同じ方法で取り扱うことができる。

計算のモナドの定義は 5.9 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 = ((Integer, Integer), String, String)
type MyIO a = ST MyState a          -- つまり、 MyState -> (a, MyState)
```

入出力のプリミティブの定義は次のようになる。

```
getXIO :: () -> ST ((x, y), i, o) x
getXIO () = _____________________________________________

setXIO :: x -> ST ((x, y), i, o) ()
setXIO x1 = _____________________________________________

readIO :: () -> MyIO Char
readIO () = _____________________________________________

writeIO :: Show s => s -> MyIO ()
writeIO v  = _____________________________________________

eofIO :: () -> MyIO Bool
eofIO () =  (s, i, o) -> (null i, (s, i, o))
```

readIO は入力ストリームから 1 文字を取出し、writeIO 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[4]。

すると、次の UtilIO プログラム

```
let sq = \ x -> if x>0 then x*x else 0-x*x
  in writeM (sq 2)
```

を実行したときの出力は、"4"になる。

5.11 UtilErr – エラー処理の導入

次に Util にエラー処理を導入する。UtilErr は、生真面目に (?) エラー処理を行ない、部分式にエラーがあれば式全体もエラーになるようにする。この場合、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 _____ （eager evaluation）をシミュレートすることに注意する必要がある。

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 Maybe を使用する。

```
-- Prelude に定義済み
data Maybe a = _____ | _____ deriving (Show,Eq)
```

正常な振舞いは Just という構成子で表す。エラーの場合は Nothing 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。

[4] このように文字列の後ろに新しい文字列を追加（++）していくと、++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。これを避けて効率の良い定義を与えることも可能であるが、ここでは簡単のために++を使った定義を採用する。

```
unitE :: a -> Maybe a
unitE a = Just a

bindE :: Maybe a -> (a -> Maybe b) -> Maybe b
(Just a) `bindE` k = ____
Nothing  `bindE` k = _________
```

m `bindE` 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。

さらに、補助関数を定義しておく。failE は ____________________________________ ときに用いる UtilErr の計算の型に特有の関数である。

```
failE :: String -> Maybe a
failE _ = Nothing
```

“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（/）は次のような Haskell の式に置換されるようにする。

```
\ x -> unitM (\ y -> if y==0 then failM "Division by 0"
                      else unitM (x/y))
```

これで 0 で割ろうとした場合にはエラーが報告される。

例えば “(\ x -> 0) (1/0)” のような式は、Haskell では ________________________________ ____________________ が、UtilErr では次のような Haskell プログラムに翻訳され、

```
(if 0 == 0 then failE "Division by 0"
           else unitE (1/0))        `bindE` \ _x ->
unitE 0
```

実行すると _________ という結果になる。

5.12 例外処理の導入

例外のモナド E を利用して、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。

BNF には以下の構文を追加する。

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

“**try** *m* **catch** *h*” は *m* 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 *m* の評価中にエラーが生じた場合は、*h* を評価する。“**try** *m* **catch** *h*” は “tryM (**delay** *m*) (**delay** *h*)” という式として構文解析されるようにしておく。

また、failM という Util の関数は、そのまま Haskell の failM にコンパイルされるようにしておく。この関数は、Java の throw 文に対応する。

ソース（Util）	ターゲット（Haskell）
try *m* **catch** *n*	tryM (delay *m’*) (delay *n’*)
failM m	*m’* `bindM` \ _x -> failM _x

tryE は ____
関数である。

```
tryE :: E e a -> E e a -> E e a
tryE (Just v) h = ________
tryE Nothing h  = _
```

例えば

```
try 1/0 catch 99999
```

という UtilErr プログラムをコンパイルすれば、出力される Haskell プログラムは、

```
tryE (if 0 == 0 then failE "Division by 0" else unitE (1/0))
     (unitE 99999)
```

であり、その結果は Just 99999.0 である。

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。