第4章　命令型言語の意味の記述

この章では簡単な命令型プログラミング言語を定義し、その意味を前章で紹介した Haskell で与えることにする。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。

ここでは Haskell でいろいろなプログラミング言語のインタプリタを作成する。以下では、簡単な言語からはじめてだんだんと複雑な構造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、Util0, UtilVar, ... 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。

4.1　Util0 – 最初のインタプリタ

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。しかし、ここではこれら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。

まず、対象とするプログラミング言語 Util の構文木のデータ型を Haskell で定義する。あとから必要な構文要素を追加することにして、最初は簡単な構文からはじめる。

```
data Expr = Add Expr Expr | Mult Expr Expr | Const Value; -- Value 型は後述
```

つまり、(最初のバージョンでは) Util は定数 (Const) と足し算 (Add)、掛け算 (Mult) のみを構成要素として持つ。

このような言語を解釈した結果の型としては次のようなデータ型を考える。

```
data Value = Num Double;
```

つまり、結果は数値 (Num) である。(最初のバージョンでは数値だけだが、後のバージョンで他の型の結果も扱えるように Value 型の構成子を増やしていく。)

次のような関数は既に定義されているものとする。

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

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 -> Value;  -- インタプリタ本体
```

4.2　抽象構文と具象構文

ところで、Util0 の構文規則を定義する時、BNF で

$$Expr \quad \rightarrow \quad Expr + Expr \mid Expr * Expr \mid Const$$

のように定義したのでは、_____（ambiguous）な文法になってしまう。そこで通常、

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

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。

問 4.2.1 上の具象構文に基づいて、*Util0* のパーサ

```
myParse :: String -> Expr;
```

を定義せよ。

4.3 Util0（つづき）

この構文に対するインタプリタはひじょうに単純である。

```
interp :: Expr -> Value;
interp (Const c)  = c;
interp (Add m n)  = let {
                      Num c = interp m;
                      Num d = interp n
                    } in
                    Num (c+d);
interp (Mult m n) = let {
                      Num c = interp m;
                      Num d = interp n
                    } in
                    Num (c*d);
```

たとえば、`interp (myParse "1+2*3")` の値は `Num 7.0` になる。

4.4 UtilVar – 変数の導入

定数と四則演算だけでは電卓の代わりにもならないので、もっとプログラミング言語らしくするために変数を利用できるようにしよう。

新しい `Expr` の定義は次のとおりである。

```
type Decl = (String, Expr);
data Expr = Add Expr Expr | Mult Expr Expr | Const Value
          | _________________ | _____________;
```

対応する（曖昧な）BNF は次のようになる。

$$
\begin{aligned}
\mathit{Expr} &\rightarrow \mathit{Expr} + \mathit{Expr} \mid \mathit{Expr} * \mathit{Expr} \mid \mathit{Const} \mid \mathit{Var} \\
&\mid \texttt{let}\ \mathit{Decl}\ \texttt{in}\ \mathit{Expr} \\
\mathit{Decl} &\rightarrow \mathit{Var} = \mathit{Expr}
\end{aligned}
$$

例えば、"**let** x=2*2 **in let** y=x*x **in** y*y" のような式を Expr 型のデータとして次のように表現する。

```
Let ("x", Mult (Const (Num 2)) (Const (Num 2)))
    (Let ("y", Mult (Var "x") (Var "x"))
         (Mult (Var "y") (Var "y")))
```

対象言語 Util に変数を導入すると、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' :: String -> [(String, a)] -> a;
lookup' x ((n,v):rest) = if n==x then v else lookup' x rest;
```

lookup' x e は ＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿。

すると、インタプリタ interp の型は

```
interp :: Expr -> Env -> Value;
```

となる。

また、interp の定義は次のようになる。

```
interp (Const c) e      = c;
interp (Add m n) e      = let {
                            Num c = interp m e;
                            Num d = interp n e
                          } in
                          Num (c+d);
interp (Mult m n) e     = let {
                            Num c = interp m e;
                            Num d = interp n e
                          } in
                          Num (c*d);
interp (Var x) e        = lookup' x e;
interp (Let (x, m) n) e = let { v = interp m e } in
                          interp n ((x,v):e);
```

interp (Let (x, m) n) では、n の評価は、拡張した新しい環境 (x,v):e で行なっている。interp (Var x) は単に現在の環境を lookup する。

例えば、interp (myParse "let x=2*2 in let y=x*x in y*y") [] という式を評価すると Num 256.0 という結果が得られる。

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

4.5 UtilFun – 関数の導入

対象言語 Util に環境の概念を導入したので、さらに関数の定義と関数適用を導入することもできる。Expr の定義に次のように構成子を追加する。

```
data Expr = Add Expr Expr | Mult Expr Expr | Const Value
          | Let Decl Expr | Var String
          | ____________________ | _________________;
```

新たにラムダ式（Lambda）と関数適用（App）を表す構成子を追加している。例えば具象構文で、ラムダ式と関数適用をそれぞれ

$$Expr \quad \rightarrow \quad \ldots \mid \backslash\ Var \text{ -> } Expr \mid Expr\ Expr$$

という形で表すことにすると、“\ f -> \ x -> f x” という式は、

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

という Expr 型のデータになる。

こんどのインタプリタは関数を値として扱う必要があるため Value 型を拡張する必要がある。

```
data Value = Num Double | ____________________________;
```

新しい構成子に対する interp の定義は次のようになる。

```
 ...
interp (App f x) e    = case interp f e of {
                          Fun g -> g (interp x e)
                        };
interp (Lambda x m) e = Fun (\ v -> interp m ((x,v):e));
```

ラムダ式 Lambda x m の本体 m は、その周りの環境 e に変数 x に対する値を追加した環境: (x,v):e で評価される。

例えば、interp (myParse ("let sq = \\ x -> x*x in sq 2")) [] という式を評価すると Num 4.0 という答が返ってくる。

ところで、関数の概念を導入したので Expr 型の中で足し算と掛け算を特別扱いする必要はなくなる。Expr から構成子 Add と Mult を取り除き、

```
data Expr = Const Value | Let Decl Expr | Var String
          | Lambda String Expr | App Expr Expr;
```

"1+2"という式を App (App (Var "+") (Const (Num 1))) (Const (Num 2)) に構文解析するように myParse を定義し、初期環境に

```
"+"  ↦  Fun (\ (Num c) -> Fun (\ (Num d) -> Num (c+d)))
"*"  ↦  Fun (\ (Num c) -> Fun (\ (Num d) -> Num (c*d)))
     ...
```

のようなプリミティブ関数用の対応を加えておけば良い。

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

こうしておくと、プリミティブ関数を比較的容易に増やすことができる。

ついでに、真偽値や組（pair）などの他のデータ型と if〜then〜else〜式なども対象言語 UtilFun に追加しておくことにする。

```
data Value = Num Double | Fun (Value -> Value) | _____ | ___________
           | _____________ | ___________ | ____________________;
data Expr = Const Value | Let Decl Expr | Var String
          | Lambda String Expr | App Expr Expr | ____________________;
```

Expr → ... | `if` *Expr* `then` *Expr* `else` *Expr*

また、++, ==, <, <=, >=, >, 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。&&, ||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。また、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。

if〜then〜else〜式に対する interp の定義は次のようになる。

```
 ...
interp (If m1 m2 m3) e = case interp m1 e of {
                            Bool b -> if b then interp m2 e
                                           else interp m3 e;
                         }
```

条件式 m1 を評価してから、m2 または m3 のいずれかを評価していることに注意する。

問 4.5.1 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?

4.6 UtilRec – Letrec の導入

関数の再帰的な定義が可能となるように対象言語 Util に再帰を許す変数宣言（letrec）を追加する。

```
data Expr = Const Value | Let Decl Expr | Var String
          | Lambda String Expr | App Expr Expr | If Expr Expr Expr
          | ____________________;
```

Expr → ... | `letrec` *Decl* `in` *Expr*

Let に関する interp の定義を見てみると、

```
interp (Let (x, m) n) e = let { v  = interp m _ } in
                          interp n ((x,v):e);
```

m は環境 e の中で評価しているので、当然 m の中では、変数 x を参照できない。“let fact = \ n

-> if n==0 then 1 else n*fact(n-1) in fact 7" という式を計算しようとしても再帰呼出しのところで fact という関数の定義がみつからずにエラーになってしまう。

そこで、Letrec に対する interp の定義では、x に対する対応を追加した新しい環境 e1 で m を評価するようにする。

```
interp (Letrec (x, m) n) e = let {
                               v  = interp m ___;
                               e1 = (x,v):e
                             } in interp n e1;
```

これだと v と e1 の定義が互いに依存しているが、もともと Haskell の **let** ではこのような再帰的定義が可能である[2]。

例えば、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 という答になる。

問 4.6.1 相互再帰関数が定義できるように、*letrec* で複数の変数を同時に宣言できるようにせよ。

4.7 UtilErr – エラーの導入

ここまでのインタプリタのバージョンは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、次のようなデータ型 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。

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

lookupM :: String -> Env -> Err Value;
lookupM x ((n,v):rest)  = if n==x then Success v else lookupM x rest;
lookupM x []            = Failure ("Variable: "++x++" is not found");
```

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。

lookupM は UtilVar 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。

インタプリタ interp の型は

[2] インタプリタの解釈の対象としている言語 Util とメタ言語である Haskell の話を混同しないように気をつける必要がある。

```
interp :: Expr -> Env -> Err Value;
```

となる。また、interp の定義は次のようになる。

```
interp (Const c) e      = Success c;
interp (Var x)   e      = lookupM x e;
interp (Let (x, m) n) e = interp m e `bindErr` \ v ->
                          interp n ((x,v):e);
interp (App f x) e      = interp f e `bindErr` \ g ->
                          case g of {
                            Fun h -> interp x e `bindErr` \ y ->
                                     h y;
                            _     -> Failure "Function expected."
                          };
interp (Lambda x m) e   = Success (Fun (\ v -> interp m ((x,v):e)));
interp (If m1 m2 m3) e  = interp m1 e `bindErr` \ v ->
                          case v of {
                            Bool b -> if b then interp m2 e
                                           else interp m3 e;
                            _      -> Failure "Boolean expected"
                          };
-- interp (Letrec ...) は後述
```

ここで、Value 型のなかの Fun 構成子の定義も、interp の型の変更に連動して、次のように変更しておく。

```
data Value = ...  | Fun (Value -> Err Value) | ...
```

4.8 モナドの導入

この UtilErr に対する interp の型のうち、Err という型構成子は、いわば対象言語の “計算” の型である。今後、対象言語 Util にいろいろな特徴を導入していくにつれ、この部分の定義が変わることになる。

そこで今後 interp 自体の書き換えができるだけ少なくて済むように、次のような型の別名を導入しておく。

```
type M a = Err a;
```

また、interp の型を

```
interp :: Expr -> Env -> M Value;
```

と書くことにする。この M に対して、UtilErr では 2 つの関数を次のように定義し、

```
unitM :: a -> M a;
unitM = Success;

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

interp の定義を次のように書き換えておく。

```
interp :: Expr -> Env -> M Value;
```

```
interp (Const c) e      = unitM c;
interp (Var x)   e      = lookupM 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` \ g ->
                          case g of {
                            Fun h -> interp x e `bindM` \ y ->
                                     h y;
                            _     -> failM "Function expected."
                          };
interp (Lambda x m) e   = unitM (Fun (\ v -> interp m ((x,v):e)));
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"
                          };
```

また、これに連動して、Value 型のなかの Fun 構成子の定義を次のようにわずかに変更しておく。

```
data Value = ...  | ______________________________ | ...
```

すると今後は、M, unitM, bindM の定義さえ変更すれば、interp 自体の型と定義はほとんど変更しなくても済むようになる。

failM は ______________________________________ 時に用いる UtilErr の計算の型に特有の関数である。

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

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。

これまで紹介したバージョン（Util0, UtilVar, UtilFun, UtilRec）でも、実は、

```
type M a = a;

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

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

のように M を定義しておけば、この interp の定義をほとんどそのまま利用できることに注意する。ただし、failM を使用している部分（App と If）については、以下のバージョンを使用する必要がある。

```
interp (App f x) e     = interp f e `bindM` \ (Fun h) ->
                        (interp x e `bindM` \ y ->
                         h y);
interp (If m1 m2 m3) e = interp m1 e `bindM` \ (Bool b) ->
                         if b then interp m2 e else interp m3 e;
```

なお、bindM は、通常、上の例のように中置演算子として用いられる。ラムダ抽象（\ ... -> ... ）は、____ ので、上記の interp (App ...) は、実は括弧を省略して例えば次のように書くことができる。

```
interp (App f x) e = interp f e `bindM` \ (Fun h) ->
                     interp x e `bindM` \ y ->
                     h y;
```

以降のプログラムでは、このように括弧を省略する。

さらに、Letrec についても、次の型を持つ関数 mfixU を導入する。

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

mfixU f は f :: (a -> M b) -> M (a -> M b) の "不動点" を表す。直観的には mfixU は以下の問の fix（あるいは λ 計算の Y コンビネータ）のバリエーションと考えておけば良い。

問 4.8.1 以下のように定義される関数: *fix*

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

を用いて定義される式:

```
fix (\ f -> \ n -> if n==0 then 1 else n * f(n-1))
```

は階乗の関数を表すことを示せ。

Letrec に対する interp の一般的な定義は、

```
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);
```

となる。

一般に、M a という型は、直観的には _____を意味する。interp :: Expr -> Env -> M Value の結果は、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。

また、unitM と bindM の直観的な意味は次のとおりである。

- unitM a — _____を表す。値 a をそのまま返す。
- m `bindM` k – 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。

このような、$unitM :: a \to M\ a$ と $bindM :: M\ a \to (a \to M\ b) \to M\ b$ という型の関数の存在する型構成子 M のことを _____（monad – より、正確には、*unitM, bindM* が

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

の 3 つの等式を満たすもの）という。

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（あるいは failM）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。

4.9 UtilErr（つづき）

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。

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）であ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 _____。

4.10 UtilCatch – 例外処理

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 = Const Value | Let Decl Expr | Var String
          | Lambda String Expr | App Expr Expr | If Expr Expr Expr
          | Letrec Decl Expr | ___________________ | ____________;
```

Expr → ... | try *Expr* catch *Expr* | 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

```
-- try 1/0 catch 99999
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。

もちろん tryM などの関数は、直接 Haskell でエラーを扱うプログラムを記述する時にも使うことができる。（つまらない例だが）上記の UtilCatch プログラムに対応する Haskell のプログラムは、

```
x `myDiv` y = if y==0 then failM "Division by 0"
                      else unitM (x/y);

tryTest = tryM (1 `myDiv` 0) (unitM 99999);
```

であり、その結果は Success 99999.0 である。

4.11 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 = Const Value | Let Decl Expr | Var String
          | Lambda String Expr | App Expr Expr | If Expr Expr Expr
          | Letrec Decl Expr
          | _____ | ____________ | _____ | ___________
          | ____________________ | _________________;
```

これに対する具象構文は次のようなものを想定する。

$$
\begin{array}{rcl}
\mathit{Expr} & \rightarrow & \mathit{Const} \mid (\mathit{Expr}) \mid \ldots \\
 & \mid & \texttt{setX}\ \mathit{Expr} \mid \texttt{getX} \mid \texttt{setY}\ \mathit{Expr} \mid \texttt{getY} \\
 & \mid & \texttt{while}\ \mathit{Expr}\ \texttt{do}\ \mathit{Expr} \mid \texttt{begin}\ \mathit{ExprSeq} \\
\mathit{ExprSeq} & \rightarrow & \mathit{Expr}\ \texttt{end} \mid \mathit{Expr}\ \texttt{;}\ \mathit{ExprSeq}
\end{array}
$$

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。そのために、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（State Transformer の略）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 4.8 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。

```
setX :: Value -> M Value;
setX v =  (x, y) -> (Unit, (v, y));

setY :: Value -> M Value;
setY v =  (x, y) -> (Unit, (x, v));

getX :: M Value;
getX   = ________________________________;

getY :: M Value;
getY   = ________________________________;
```

```
interp (SetX m1) e = interp m1 e `bindM` \ v ->
                     setX v;
interp (SetY m1) e = interp m1 e `bindM` \ v ->
                     setY v;
interp GetX e      = getX
interp GetY e      = getY
```

SetX, SetY は State を書き換え、また GetX, GetY は State 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。

なお、ST のようなモナドは、直接 Haskell で命令的なプログラムを記述するためにも使える。例えば、階乗の関数は次のように書くことができる。

```
factLoop :: ST Value;
factLoop = getY                          `bindM` \ (Num n) ->
           if n>0 then getX              `bindM` \ (Num p) ->
                       setX (Num (n*p)) `bindM` \ _ ->
                       setY (Num (n-1)) `bindM` \ _ ->
                       factLoop
           else        getX;

fact   :: Double -> ST Double;
fact n = setX (Num 1) `bindM` \ _ ->
         setY (Num n) `bindM` \ _ ->
         factLoop     `bindM` \ (Num r) ->
         unitM r;
```

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。

問 4.11.1 本節の *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。

4.12 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、4.11 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 4.11 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[3]。

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]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。

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。