第5章 モナドと命令型言語の意味

この章では、命令型言語のさまざまな副作用に対して、モナドを利用してその意味を Haskell で与えることにする。Haskell で意味を与えるということは、等式による推論が可能になるということである。

前章で紹介した `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ーが独自の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 *return* と *(>>=)* 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。

5.1 ミニ命令型言語 Util

この章で紹介する副作用をすべて備え持つ命令型言語は現実には存在しないので、説明のための独自のミニ命令型プログラミング言語を導入する。簡単な言語からはじめて様々な副作用を持つ言語を定義していく。状況に応じて C, Java などに加えて、このようなミニ命令型プログラミング言語のプログラムの意味を Haskell で説明する。

名前がないと不便なので、これらのミニ命令型言語をUtil (**U**til: **T**iny **I**mperative **L**anguage) ((いわゆる再帰的頭字語 (recursive acronym) である。PHP, GNU などの略語の由来も参照すること。))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。いろいろな副作用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。

Util については、この章では、必要な事柄の説明だけにとどめて、補足の第 U 章で実装 （Util から Haskell へのコンパイラー）の詳細を紹介する。ただし、実装の詳細は把握しなくても、ソースプログラムとターゲットプログラムを比べれば、副作用を持つプログラミング言語がどのように変換されるか感覚的に掴めるはずである。

5.1.1 構文規則

Util の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`** *Binds* **`in`** *Expr*
　　| \ *Vars* -> *Expr* | *Expr Expr*
　　| *Expr* + *Expr* | *Expr* * *Expr* | … （他の中置演算子）…
Exprs → *Expr* | *Expr* ; *Exprs*
Decl → *Vars* = *Expr*

Decls → *Decl* | *Decl* ； *Decls*
Bind → *Var* <- *Expr*
Binds → *Bind* | *Bind* ； *Binds*
Vars → *Var* | *Var Vars*

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。

Haskell と異なり **`while`** ～ **`do`** 式や **`begin`** ～ **`end`** 式があるところが命令型言語らしいところである。

5.1.2 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。つまり `1 + 2 * 3` は足し算と掛け算のどちらを先に実行するか、決定できない。通常は曖昧さを避けるために、

Expr → *Expr* + *Term* | *Term*
Term → *Term* * *Factor* | *Factor*
Factor → *Const* | （ *Expr* ）

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。

5.1.3 Util から Haskell への翻訳

命令型言語である Util から Haskell への翻訳を考えることができる。補足第 U 章では、さらに詳しく Util から Haskell へのコンパイラーの実装を紹介するが、ここでは対応表の形で説明する。

次の表のソース中で *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 ⟦ ⟧ が付いている式は、その変換後の Haskell の式を表す。

ソース (Util)	ターゲット (Haskell)
`c`（ただし `c` は定数）	`return c`
`x`（ただし `x` は変数）	`return x`
`val` `x <-` *m* **`in`** *n*	⟦*m*⟧ `>>= \ x ->` ⟦*n*⟧
`let` `f = \ x ->` *m* `g = \ y ->` *n* **`in`** *k*	**`let`** `f = \ x ->` ⟦*m*⟧ `g = \ y ->` ⟦*n*⟧ **`in`** ⟦*k*⟧
f a	⟦*f*⟧ `>>= \ _g ->` ⟦*a*⟧ `>>= \ _x ->`

	`_g _x`
`\ x -> m`	`return (\ x -> ⟦m⟧)`
if *c* **then** *t* **else** *e*	`⟦c⟧ >>= \ _b ->` **if** `_b` **then** `⟦t⟧` **else** `⟦e⟧`

while *c* **do** *t*	**let** `_while = ⟦c⟧ >>= \ _b ->`
	`  **if** _b **then** ⟦t⟧ >>= \ _ ->`
	`  _while`
	`  **else** return ()`
	in `_while`

begin
s;
t;
u
end
	
	`⟦s⟧ >>= \ _ ->`
	`⟦t⟧ >>= \ _ ->`
	`⟦u⟧`

この翻訳の要点は、変数や定数の出現など"副作用"が発生しないところには `return` がついた形に翻訳されること、関数適用や **begin** 〜 **end** による逐次実行は `(>>=)` を使った式に翻訳されること、などである。

また、Util プログラム中の +, –, * などの（副作用を持たない）プリミティブな二項演算子は、他の関数が `return (\ 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 内のオペレーターを用いて、それぞれ

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

という Haskell の式に置換するようにしておくと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
副作用のない二項演算子）
	`return (\ x ->`
	`  return (\ y ->`
	`  return (x ⊗ y)))`

この対応表を用いて、例えば次の Util プログラムを変換（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。）すると

```
1  fact n = if n == 0 then 1 else n * fact (n - 1)
```

次のような Haskell のプログラムが得られる。

```
1  fact = \ n ->
2   ((return (\ x -> return (\ y -> return (x == y))) >>=
3       \ _f -> return n >>= \ _x -> _f _x)
4     >>= \ _f -> return 0 >>= \ _x -> _f _x)
5    >>=
6    \ _b ->
7      if _b then return 1 else
8        (return (\ x ->
9           return (\ y -> return (x * y))) >>=
10          \ _f -> return n >>= \ _x -> _f _x)
11         >>=
12         \ _f ->
13           (return fact >>=
```

```
                \ _f ->
                  ((return (\ x ->
                     return (\ y -> return (x - y)))
                     >>= \ _f -> return n
                       >>= \ _x -> _f _x)
                         >>= \ _f -> return 1
                           >>= \ _x -> _f _x)
                    >>= \ _x -> _f _x)
              >>= \ _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、多くの `return` や `(>>=)` が消えて、次のような Haskell の式が得られる。

```
fact = \ n -> if n == 0 then return 1 else
                    fact (n - 1) >>= \ _x ->
                    return (n * _x)
```

5.2 トリビアルなバージョン — 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。

ファイル Id.hs

```
newtype I a = I a

instance Monad I where
  return a = I a
  (I m) >>= k = k m
```

ここで、 `newtype` は、Haskell の新しい型の宣言の形式の一つである。`data` 宣言と似ているが、フィールドが一つの構成子を一つしか持つことができない。また `data` 宣言の構成子と異なり、`newtype` 宣言で導入される構成子は型変換の意味しか持たず、実行時の計算を伴わない。また、型の別名を宣言する `type` 宣言との違いは、型の変換が構成子によって明示的になる点である。（型クラスのインスタンスには、`type` 宣言で導入された型の別名は指定できない。例えば リストに通常の辞書式順序と異なる順序（`Ord` のインスタンス宣言）を与えたいときは `newtype` を使って別名を用意し、別名に対してインスタンス宣言する必要がある）

上記の `newtype` 宣言では型構成子と構成子に同じ `I` という名前を使っているが、文脈でどちらか判断することができるので問題ない。

なお、ある型構成子を `Monad` のインスタンスと宣言するとき、同時に `Monad` のスーパークラスである `Functor` と `Applicative` に対してもインスタンス宣言をする必要がある。今後紹介するモナドに対してもすべて同一なので `I` に対するインスタンス宣言だけを代表として紹介しておく。

ファイル Id.hs

```
instance Functor I where
```

```
  fmap f m = m >>= \ x -> return (f x)

instance Applicative I where
  pure = return
  g <*> m = g >>= \ f -> m >>= \ x -> return (f x)
```

このとき、

```
fact n = if n == 0 then 1 else n * fact (n - 1)
```

という Utill プログラムを Haskell に翻訳して実行する `unI (fact 9)` と、同じプログラムを Haskell として実行したときと全く同じ `362880` という値になる。これは何の意味もないように思えるが、副作用がなければ、Util と Haskell は意味が変わらないという確認にはなる。

ただし、`unI` は次のように定義された型変換関数である。

ファイル Id.hs

```
unI :: I a  -> a
unI (I a) = a
```

5.3 UtilST — 状態の導入

Util に更新（破壊的代入）可能な状態の概念を導入する。 ここで紹介する例では、2 つだけ代入可能な**参照** (reference) `xP` と `yP` 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。参照は `:=` 演算子で値を代入し、`get` で値を取り出すことができる。

例えば、

```
begin xP := 1; xP := get xP + 3; get xP end
```

という UtilST プログラムを評価すると、`4` という結果が得られる。

> **参考:**なお、Util に代入可能な変数をそれ以外の変数と区別するような宣言（例えば Kotlin の `var` と `val`）を導入したり、変数名のカテゴリーを区別（例えば、特定の文字セットから始まる変数名を代入可能な変数に割り当てる）したりすれば、 `get` の適用を省略して、
>
> ```
> begin μ := 1; μ := μ + 3; μ end
> ```
>
> のように書くことができるように言語仕様を変更することも可能である。こうすれば、C 言語により近い記法でプログラムを記述することも可能である。以下では、`μ`, `ν` のようにギリシア文字を使った変数は、代入可能な参照を指す変数として宣言されていると仮定する。 しかし `get` を使う書き方にも、副作用が存在している箇所が明確になると利点がある。

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 という型構成子を定義する。

ファイル ST.hs

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

unST :: ST s a -> s -> (a,s)
unST (ST s) = s

instance Monad (ST s)  where
  return a      = ST (                )
  (ST m) >>= k = ST (\ s0 -> let { (a,s1) = m s0 }
                                 in unST (k a) s1)
  -- ST, unST がなければ、次のようになる
  -- m >>= k = \ s0 -> let (a,s1) =  m s0 in k a s1
```

この型構成子は `ST s` という形で s というパラメーター付きのモナドになる。このモナドは State Transformer モナドと呼ばれる。 `return 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。このモナドの `m >>= 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。

問 5.3.1 ST に対して、次の monad law が成り立つことを確認せよ。

```
        (return a) >>= k  =  k a
m >>= (\ a -> return a)   =  m
      (m1 >>= k1) >> k2   =  m1 >>= (\ a -> (k1 a >>= k2))
```

参照は次のように定義する。
ファイルMyState.hs

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

xP :: Pos (x,y) x
xP = \ (x,y) -> (x, \ x1 -> (x1,y))

yP :: Pos (x,y) y
yP = \ (x,y) -> (y, \ y1 -> (x,y1))
```

`Pos s a` は s という全体の状態の型の中で、a という型を指している参照の型である。 ここで xP と yP は、それぞれペアの第 1、第 2 成分にアクセスするための参照である。参照に対する読み出し・書き込みのメソッドは、後で別のモナドでも使用するので、型クラス MyState のメソッドとして定義しておくことにする。

ファイル MyState.hs

```
class MyState m where
  get :: Pos s a -> m s a
  set :: Pos s a -> a -> m s ()
```

そして ST をこの MyState クラスのインスタンスとして宣言する。
ファイル ST.hs

```haskell
instance MyState ST where
  get p   = ST (\ s -> (fst (p s), s))
  set p v = ST (\ s -> ((), snd (p s) v))

-- 例えば get xP ≡ ST (\ (x,y) -> (x,(x,y)))
--       set xP x1 ≡ ST (\ (x,y) -> ((),(x1,y)))
```

ここで `set` は参照先の状態を書き換える、また `get` は参照先の値を複製する関数である。

また、最終的に `ST` から値を取り出すために

```haskell
evalST :: ST s a -> s -> a
evalST st s = fst (unST st s)
```

という関数を定義しておく。第 2 引数の `s` は初期状態である。

Q 5.3.2 次のように `ST` を使った関数 `foo`, `bar` を定義する。

```haskell
import ST
import MyState

foo b y = do set xP 1
             set yP y
             let repeat =
                   do y1 <- get yP
                      if y1 > 0 then do
                        x1 <- get xP
                        set xP (x1 * b)
                        set yP (y1 - 1)
                        repeat
                      else return ()
                in repeat
             get xP

bar n = do set xP 1
           set yP 1
           let repeat =
                 do x1 <- get xP
                    if x1 < n then do
                      y1 <- get yP
                      let y2 = y1 + 1
                      set yP y2
                      set xP (x1 * y2)
                      repeat
                    else return ()
             in repeat
           get yP
```

これらの関数に対して、

1. `evalST (foo 3 4) (0,0)`
2. `evalST (bar 50) (0,0)`

の値を、まず実行する前に予想し、次に実際に実行して確認せよ。

UtilST の `:=` 演算子と `get` 関数は、 Haskell の `set`, `get` にそれぞれコンパイルされるようにしておく。

ソース (Util)	ターゲット (Haskell)
p `:=` m	⟦p⟧ `>>= \ _p ->` ⟦m⟧ `>>= \ _x ->` `set _p _x`
`get` p	⟦p⟧ `>>= \ _p ->` `get _p`

左の UtilST プログラム（右は対応する C プログラム）:

```
fact n = begin
  xP := 1; yP := n;

  while get yP > 0 do
begin
    xP := get xP * get yP;
    yP := get yP - 1
  end;
  get xP
end
```

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

  }
  return x;
}

```

参考: 次は `get` を省略する書き方にして、さらに C に近付けた UtilST プログラムである。

```
fact n = begin
  μ := 1; ν := n;
  while ν > 0 do begin
    μ := μ * ν;
    ν := ν - 1
  end;
  μ
end
```

を翻訳すると次のような Haskell の関数になる。

```
fact n = set xP 1 >>= \ _ ->
         set yP n >>= \ _ ->
         (let _while = get yP >>= \ y ->
                       if y > 0 then
                            get xP          >>= \ x ->
                            get yP          >>= \ y ->
                            set xP (x * y) >>= \ _ ->
                            get yP          >>= \ y ->
                            set yP (y - 1) >>= \ _ ->
                            _while
                       else return ()
            in _while)    >>= \ _ ->
         get xP
```

この翻訳した `fact` に対して `fact 9` を実行する `evalST (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のほうが簡潔だが、変数の数が多い場合などは、このような命令的な書き方が簡潔になる場合も考えられる。

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。参考までに、次のようなモナドになる。
ファイル EST.hs

```haskell
newtype EST s a = EST (s -> Maybe (a,s))

unEST :: EST s a -> s -> Maybe (a,s)
unEST (EST m) = m

instance Monad (EST s) where
  return a = EST (\ s -> return (a,s))
  (EST m) >>= k = EST (\ s0 ->
     case m s0 of
       Just (a,s1) -> unEST (k a) s1
       Nothing     -> Nothing)
```

問 5.3.3 次の C の関数とほぼ同等な UtilST の関数、または、 ST モナドを用いた Haskell の関数を定義せよ。

1.
```c
int foo(int n) {
  int i = 1, j = 1;
  while (i < n) {
    i = i + j;
    j = i - j;
  }
  return i;
}
```

2.
```c
int bar(int n) {
  int i = 0;
  while (n > 1) {
    i = i + 1;
    n = n / 2;
  }
  return i;
}
```

※ 整数の割り算は Haskell では `div` 演算子 になる。

5.4 （参考）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、前節の 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前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。
ファイル MyStream.hs

```haskell
type WithIO s = (s,String,String)
```

第 2 要素が入力ストリーム、第 3 要素が出力ストリームを表す部分である。

ファイル MyIO.hs

```
newtype MyIO s a = MyIO (WithIO s -> (a, WithIO s))

unMyIO :: MyIO s a -> WithIO s -> (a, WithIO s)
unMyIO (MyIO m) = m
```

参照のプリミティブの定義は次のようになる。
ファイル MyIO.hs

```
instance MyState MyIO
  get p   = MyIO (\ (s,i,o) -> (fst (p s),(s,i,o)))
  set p v = MyIO (\ (s,i,o) -> ((),(snd (p s) v,i,o)))
```

入出力に関するプリミティブも後で別のモナドで使用するので、型クラス `MyStream` のメソッドとして定義しておく。
ファイル MyStream.hs

```
class MyStream m where
  readChar   :: m Char
  eof        :: m Bool
  writeStr   :: String -> m ()
```

ファイル MyIO.hs

```
instance MyStream (MyIO s) where
  readChar    = MyIO (\ (s,c:cs,o) -> (c,(s,cs,o)))
  eof         = MyIO (\ (s,i,o) -> (null i,(s,i,o)))
  writeStr v  = MyIO (\ (s,i,o) -> ((),(s,i,o ++ v)))
```

ここで `readChar` は入力ストリームから1文字を取出す。また `writeStr str` は出力ストリーム `o` に `str` を追加する（ただし「++」の計算量は左オペランドの長さに比例するので、 この定義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、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。）。そして `write` という関数を次のように定義しておく。
ファイルMyStream.hs

```
write :: (Show v, MyStream m) => v -> m ()
write v = writeStr (show v)
```

すると、次の UtilIO プログラム（ただし、「//」は整数の除算を表す演算子とする。）

```
foo n = begin
    xP := n;
    while get xP > 0 do begin
      write (get xP % 10);
      xP := get xP // 10
    end
  end
```

を Haskell に翻訳した結果は、

```haskell
foo n = set xP n >>= \ _ ->
        let _while
              = get xP >>= \ _x ->
                if _x > 0 then
                    get xP                >>= \ _x ->
                    write (_x `mod` 10)   >>= \ _ ->
                    get xP                >>= \ _x ->
                    set xP (_x `div` 10) >>= \ _ ->
                    _while
                else return ()
          in _while
```

となり、補助関数 `evalMyIO` を以下のように定義したとき、

```haskell
evalMyIO e s =
  let (_,(_,_,o)) = unMyIO e (s,"","") in o
```

関数 `foo` を `evalMyIO (foo 12345) (0,0)` のように実行すると、出力は `"54321"` になる。

問 5.4.1 次の C の関数とほぼ同等な UtilIO の関数、または、`MyIO` モナドを用いた Haskell の関数を定義せよ。

1.
```c
int baz(int n) {
  int i, j;
  for (i = 0; i < n; i++) {
    for (j = 0; j <= i; j++) {
      printf("*");
    }
    printf("\n");
  }
  return i;
}
```

2.
```c
int qux(int n) {
  int i, j;
  for (i = 0; i < n; i++) {
    printf("*");
    if (i % 3 == 0) {
      printf("!");
    }
    printf("\n");
  }
  return i;
}
```

5.5 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。このように 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) を模倣する。

エラーと正常な振舞いを区別するために、次のような標準ライブラリーに用意されているデータ型 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インスタンス宣言がされている。

```
-- Preludeに宣言済み
instance Monad Maybe where
  return a = Just a
  (Just a) >>= k =
  Nothing  >>= k =
```

このモナドの m >>= 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、＿＿＿＿＿＿していくことを表している。

問 5.5.1 Maybe に対して、次の monad law が成り立つことを確認せよ。

```
        (return a) >>= k  =  k a
m >>= (\ a -> return a)   =  m
      (m1 >>= k1) >> k2   =  m1 >>= (\ a -> (k1 a >>= k2))
```

さらに、MonadPlus というクラスに属するいくつかのメソッドを用意する。ここで mzero は＿＿＿＿＿＿状況をつくり出すときに用いる。

Maybe に対する mplus は第 1 引数を評価し、＿＿＿＿＿＿＿＿第 2 引数を評価する関数である。

```
-- モジュール Control.Monad に定義済み
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a

-- モジュール Control.Monad に宣言済み
instance MonadPlus Maybe where
  mzero = Nothing
  Just a  `mplus` _ = Just a
  Nothing `mplus` m = m
```

Q 5.5.2 次のように Maybe を使った関数 foo, bar を定義する。

```
import Control.Monad

mydiv :: Double -> Double -> Maybe Double
x `mydiv` y = if y == 0 then mzero else return (x / y)

foo :: Double -> Maybe Double
foo n = do x <- 6 `mydiv` n
           return (x * 2)

bar :: Double -> Double -> Maybe Double
bar m n = do x <- (4 `mydiv` m) `mplus` (return 3)
```

```
12              x `mydiv` n
13
```

これらの関数に対して、(1) `foo 2`　(2) `foo 0`　(3) `bar 0 4`　(4) `bar 1 0`　の値を、まず実行する前に予想し、次に実際に実行して確認せよ。

いわゆる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、UtilErr の割り算 (/) は、0 で割るとエラーなので、次のような Haskell の式に置換されるようにする。

```
\ x -> return (\ y -> if y == 0 then mzero
                                else return (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

```
1 (\ x -> 999) (1 / 0)
```

のような式は、Haskell の式としても Util の式としても解釈できるが、 Haskell だと解釈すると、 `1 / 0` の部分式は＿＿＿＿＿に全体の結果が 999 となるが、 UtilErr だと解釈すると、次のような Haskell プログラムに翻訳され、

```
1 (if 0 == 0 then mzero
2            else return (1 / 0)) >>= \ _x ->
3 return 999
```

実行すると（エラーが起こったことを表す）＿＿＿＿＿という結果になる。つまり、先行評価をシミュレートしていることになる。

5.6 例外処理の導入

例外のモナド `Maybe` 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。

Util の BNF には以下の構文を追加する。

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

"**try** m **catch** n" 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、n を評価する。Util の "**try** m **catch** n" は "⟦m⟧ `` `mplus` `` ⟦n⟧"という式として構文解析されるようにしておく。

また、`fail` という Util の関数は、Haskell の `mzero` を返す関数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	⟦m⟧ `` `mplus` `` ⟦n⟧
`fail ()`	`mzero`

例えば

```
bar n = try 1 / n 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、

```
bar n = (if n == 0 then mzero else return (1 / n))
        `mplus` (return 99999)
```

であり、`bar 0` の結果は `Just 99999.0` である。

問 5.6.1 次の Java のメソッドとほぼ同等な Haskell の関数をモナドを用いて作成せよ。

```
  public static void hoge(int n) {
    int i;
    for (i = -n; i <= n; i++) {
      try {
        System.out.printf("%d ", 360 / i);
      } catch (Exception e) {
        System.out.printf("(/ by 0) ");
      }
    }
    System.out.println();
  }
```

5.7 UtilNonDet — 非決定性の導入

ここからは 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＿＿＿＿＿＿＿＿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（バックトラック）。非決定性は探索型のパズル・ゲームや構文解析プログラムなどで利用できる。バックトラック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＿＿＿＿が有名である。

非決定性の“計算”のモナドは通常のリストを使用する。

```
-- Prelude で宣言済み
instance Monad [] where
  return a = [a]
  []     >>= k = []
  (x:xs) >>= k = k x ++ (xs >>= k)

-- モジュール Control.Monad で宣言済み
instance MonadPlus [] where
  mzero = []
  mplus = (++)
```

このモナドは複数の選択肢を単にリストとして表現している。要するに、失敗は空リスト `[]` と表し、“ふつう”の成功は、シングルトン（要素数 1 のリスト）で表している。やり直しを「++」で表している。つまり、`Maybe` のときと違って、将来ダメだったときの選択を残している。

Q 5.7.1 次のように `Maybe` とリストを使った関数 `foo, bar` を定義する。（`foo, bar` は定義は同一で、型だけが異なる。）

```
import Control.Monad

-- mydiv は型を明示しない
x `mydiv` y = if y == 0 then mzero else return (x / y)

foo :: Double -> Maybe Double
foo m = do x <- (12 `mydiv` m) `mplus` (12 `mydiv` (m + 1))
           12 `mydiv` (4 - x)

bar :: Double -> [Double]
bar m = do x <- (12 `mydiv` m) `mplus` (12 `mydiv` (m + 1))
           12 `mydiv` (4 - x)

```

これらの関数に対して、① `foo 4` ② `bar 4` ③ `foo 3` ④ `bar 3` の値を、まず実行する前に予想し、次に実際に実行して確認せよ。

問 5.7.2 `[]` に対して、次の monad law が成り立つことを確認せよ。

```
(return a) >>= k  =  k a
m >>= (\ a -> return a)  =  m
(m1 >>= k1) >> k2  =  m1 >>= (\ a -> (k1 a >>= k2))
```

実はリストに対する `return` と `(>>=)` は、リストの内包表記を説明するときに使った `unit :: a -> [a]` と `bind :: [a] -> (a -> [b]) -> [b]` と全く同一の関数である。

計算の失敗は空リストで表される。

UtilNonDet は、構文は UtilErr と同じである。つまり、以下の構文を持つ。

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

UtilNonDet の `try` *m* `catch` *h* は、*m* を評価し、 *h* は"バックトラック"が起こったときに評価される。

UtilNonDet プログラム

```
test0 = (try 2 catch 3) * (try 5 catch 7)
```

をコンパイルすると、次の Haskell プログラムが得られる。

```
test0 = (return 2 `mplus` return 3) >>= \ x ->
        (return 5 `mplus` return 7) >>= \ y ->
        return (x * y)
```

この、`test0` は `[ x * y | x <- [2,3], y <- [5,7]]` というリスト内包表記と同じ意味になる。そして `test0` は、`[10,14,15,21]` となる。

また、次のUtilNonDet プログラム

```
test1 n = (try 1 catch 2) / (try n catch 4)
```

を翻訳すると、次の Haskell プログラムが得られる。

```
test1 n  = (return 1 `mplus` return 2) >>= \ x ->
           (return n `mplus` return 4) >>= \ y ->
           if y == 0 then mzero else return (x / y)
```

そして `test1 2` は、`[0.5,0.25,1.0,0.5]`, `test1 0` は `[0.25,0.5]` 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。同じプログラムを UtilErr で翻訳すると、UtilErr ではバックトラッキングが起こらないので、 `test1 0` は全体が失敗 (`Nothing`) に終わる。

なお、次の `head` を用い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ことも可能である。

```
-- Prelude に定義済み
head :: [a] -> a
head (x:_) = x
```

このとき `head (test1 0)` の値は `0.25` となる。この場合、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。

8 クイーンの問題も非決定性を用いて記述することができる。

```
safe1 xs n m = null xs ||
               val y  <- head xs;
                   ys <- tail xs in
               y /= n && y /= n + m && y /= n - m
                      && safe1 ys n (m + 1);

safe xs n = safe1 xs n 1;

range i j = if i > j then fail ()
            else try i catch range (i + 1) j;

queen n = if n == 0 then []
          else val p <- queen (n - 1);
                   n <- range 1 8 in
               if safe p n then (n:p) else fail ()
```

そして `queen 8` は、`[4,2,7,3,6,8,5,1]` から始まる 92 個の解を返す。

問 5.7.3 非決定性と状態の両方の特徴を持つ計算の型として、

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

の 2 つのバリエーションが考えられる。このそれぞれに対して、コンパイラーの定義を完成させ、2 つの違いを説明せよ。

5.8 Prolog と論理変数

論理型言語の Prolog には非決定性の他に、＿＿＿＿＿＿という特徴がある。論理変数 (logical variable) とは、最初は値が定まっておらず、単一化の制約により徐々に値が具体化していく変数である。何度も代入できる命令型言語の変数とも、一度初期化すればニ度と代入ができない関数型言語の変数とも異なる。Haskell では論理変数自体は、破壊的代入を模倣するために使った State Transformer モナドと同様の方法で模倣することができる。

例えば、Prolog では、リストを連接 (append) するプログラムは次のように記述される。

```
myappned([H|X], Y, [H|Z]) :- myappned(X, Y, Z).
myappned([], Y, Y).
```

これは、1 番目の引数と 2 番目の引数を連接した結果が 3 番目の引数になる、という関係を表している。

この `myappned` に対して、`[1,4,3]` と `[5,3]` の連接を求めるには、次のような問合せ（呼出し）をする。

```
- myappned([1,4,3], [2,5], R).

R = [1,4,3,2,5] ;

No
```

さらに Prolog のおもしろいところは `myappned` の逆向きの計算もできるということである。

```
?- myappned(A, B, [1,2,3]).

A = []
B = [1,2,3] ;

A = [1]
B = [2,3] ;

A = [1,2]
B = [3];

A = [1,2,3]
B = [] ;

No
```

この実行例では連接して `[1,2,3]` になる 2 つのリストの、すべての可能性を求めていることになる。ユーザーが「;」を入力するたびにバックトラッキングが起こり別解を表示する。

MicroKanren.hs (`https://github.com/rntz/ukanren`) は、μKanren という論理プログラミングのための埋め込み言語 (embedded language) を実装するための Haskell のライブラリーである。μKanren は miniKanren という言語のミニ

マルな実装である。 このライブラリーでは、非決定性と論理変数にプラスアルファの機能を加えたモナドが定義されている。ここで詳細な説明には立ち入らないが、このモナドは State Transformer と非決定性を組み合せたものである。このライブラリーを使えば、上の Prolog の `myappned` プログラムに相当するプログラムを Haskell で次のように記述できる。

```
myAppend a b ab =
        do { h <- fresh; t <- fresh; res <- fresh;
             ht <- cons h t; ht === a;
             hres <- cons h res; hres === ab;
             myAppend t b res }
    `mplus`
        do { n <- nil; n === a; b === ab }
```

ここで、`fresh` は新しい論理変数を生成する関数であり、「`===`」は両辺が単一化されることを示す演算子である。

ちなみに、Util の文法では次のように書くことができる。

```
myAppend a b ab =
  try val h = fresh() in val t = fresh() ;
          res = fresh() in
      begin
        cons h t === a; cons h res === ab;
        myAppend t b res
      end
  catch begin nil() === a; b === ab end
```

この `myAppend` に対して次のようなプログラムを実行する。ただし、`toLVList` は通常のリストの型から論理変数を含むリストの型への変換、`fromLVList` はその逆である。ただし `fromLVList` c `xs` は `xs` のなかの未定の論理変数を c に置き換えた通常のリストを返す。

```
exampleAppend a =
  val xs = fresh();
      ys = fresh();
      zs = toLVList [1,2,3] in begin
    myAppend xs ys zs;
    (fromLVList 0 xs,fromLVList 0 ys)
  end
```

その結果は `[([],[1,2,3]),([1],[2,3]),([1,2],[3]),([1,2,3],[])]` になる。

5.9 さらに詳しく知りたい人のために...

Parsec (Leijen & Meijer 2001) はモナドを利用した有名なパーサーライブラリーである。 (Wadler 1992a) はモナドを用いてインタプリターを構築する方法を解説している。 (Wadler 1992b) にも、モナドを用いてパーサーを構築する技法の解説がある。 (Hinze 1998) は、Prolog のカット等のオペレーターの意味を整理

している。 (Kiselyov et al. 2005) は、miniKanren を Haskell で実装するためのモナドの解説である。

(Leijen & Meijer 2001) Daan Leijen and Erik Meijer, “Parsec: Direct Style Monadic Parser Combinators for the Real World”
Technical Report UU-CS-2001-35, Dept. of Comp. Sci, Universiteit Utrecht, 2001 年, `http://www.cs.uu.nl/people/daan/parsec.html`

(Wadler 1992a) Philip Wadler, “The essence of functional programming”
19th Annual Symposium on Principles of Programming Languages (invited talk), 1992 年 1 月

(Wadler 1992b) Philip Wadler, “Monads for functional programming”
Program Design Calculi, Proceedings of the Marktoberdorf Summer School, 1992 年 7–8 月

(Hinze 1998) Ralf Hinze, “Prological Features in a Functional Setting Axioms and Implementations”
Third Fuji International Symposium on Functional and Logic Programming, 1998 年

(Kiselyov et al. 2005) Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman and Amr Sabry, “Backtracking, interleaving, and terminating monad transformers (functional pearl)”
ACM SIGPLAN Notices (Vol. 40, No. 9, pp. 192–203), ACM, 2005 年 9 月
