第4章 モナド

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

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

4.1 参照透明性と副作用

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

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

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

と

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

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

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

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

例えば、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。

```haskell
getChar :: IO Char

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

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

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

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

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

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

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

4.2 モナドとは

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

具体的にはモナドとは

return :: a → M a

(>>=) :: M a → (a → M b) → M b

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

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

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

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

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

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

4.3 モナドと型クラス

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

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

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

4.4 IO モナド

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

```
-- type IO a ≒

-- instance Monad IO where
-- -- 注：実際には type で定義された型名を instance には使えない。
--   return a = \ w -> (a,w)
--   m >>= k  = \ w -> let (a,w1) = m w in k a w1
```

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

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

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

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

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

特に getContents は遅延評価で標準入力の内容を読み込む。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。

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

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

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

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

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

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

```

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

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

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

以降のプログラムでは、このように括弧を省略する。

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

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

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

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

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

do *{ e }* ⇒ *e*

do *{ e; stmts }* ⇒ *e >>= \ _ -> **do** { stmts }*

do *{ x <- e; stmts }* ⇒ *e >>= \ x -> **do** { stmts }*

do *{* ***let*** *decls; stmts }* ⇒ ***let*** *decls* ***in*** ***do*** *{ stmts }*

（実は、この翻訳規則はリストの内包表記の翻訳規則とほとんど同じである。）

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

1. 一行だけ行を読み込んで、それをオウム返しするプログラム
2. 一行だけ行を読み込んで、それを2回オウム返しするプログラム

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

```
import Data.Char

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

capitalizeLine line
    = unwords (capitalizeFirst (words line))

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

問 4.4.2 IO モナドを利用して次のようなプログラムを作成せよ。（次節で紹介している関数を使用するかもしれない。）

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

4.5 参照型と可変な配列型

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

```
-- import Data.IORef が必要

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

次に、IORef を利用したプログラムの例をあげる。

```
import Data.IORef

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

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

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

```
{- foo の自然な定義は以下の通り -}
foo i = if i <= 0 then return ()
        else do putStrLn (show i)
                foo (i - 1)

main = foo 10
```

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

```
-- import Data.Array.IO が必要

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

次に、Data.Array.IO を利用したプログラムの例をあげる。

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

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

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

```
        loop (i1 + 1) i2 arr

accumArray arr = do
    (i1, i2) <- getBounds arr
    loop i1 i2 0 arr
  where
    loop i1 i2 v arr | i1 > i2 = return ()
                     | otherwise = do
       a <- readArray arr i1
       let w = a + v
       writeArray arr i1 w
       loop (i1 + 1) i2 w arr

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

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

ここで `randomArray` は乱数で初期化した配列を作成する関数、`printArray` は配列の要素を出力する関数、`accumArray` は配列を破壊的に書き換えて足し算をする関数である。このプログラムを実行すると、例えば次のような出力が得られる。

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

4.6 続・有用なリスト処理関数

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

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

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

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

-- unwords は words の逆操作である。
-- 各文字列を空白で区切って結合する。
```

```haskell
unwords         :: [String] -> String

-- take n xs は xs の長さ n の先頭部分を返す。
-- n > length xs のときは xs自身を返す。
take            :: Int -> [a] -> [a]

-- drop n xs は xs の最初の n 個の要素を除いた末尾部分を返す。
-- n > length xs のときは [] を返す
drop            :: Int -> [a] -> [a]

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

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

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

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

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

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

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

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

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

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

```haskell

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

Q 4.6.1 次の式の値は何か?

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

4.7 さらに詳しく知りたい人のために...

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

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

第5章 モナドと命令型言語の意味

この章では、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 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。具体的には命令型プログラミング言語から Haskell へのコンパイラーを作成する。

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 Util コンパイラー

この節では、コンパイラーの実装を紹介していく。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。

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。

実際のコンパイラー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。)

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。

5.1.1 構文規則

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

ファイルRecType.hs

```
1 type Decl = (String, Expr)
2 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。）

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。

ただし、_（アンダーバー）から始まる変数名はコンパイラー内部で使用するために予約済みとする。

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 字句解析・構文解析関数

次のような関数が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 ↦ if b1 then b2 else False
b1 || b2 ↦ if b1 then True else 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。

5.1.3 ターゲット言語

コンパイラーのターゲット言語である Haskell のサブセットの構文木を表現する型 Target 型を定義しておく。

ファイルTarget.hs

```
type TDecl = (String, Target)
data Target = TLit Literal              -- 定数
            | TVar String               -- 変数
            | TIf Target Target Target  -- if 文
            | TLet [TDecl] Target       -- let 文
            | TLambda1 String Target    -- ラムダ式
            | TApp1 Target Target       -- 関数適用
            | TReturn Target            -- return に相当
            | TBind Target Target       -- (>>=) に相当
           deriving (Show,Eq)
data Literal = Str String | Int Integer | Frac Rational
             | Char Char              deriving (Show,Eq)

```

Util のコンパイラーとは次のような型を持つ関数である。

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

5.1.4 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 →

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

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`, `Target` 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。

5.1.5 コンパイラーの定義

関数 `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直截である。

ファイルRecCompiler.hs

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

関数 comp を理解するために、変換前と変換後を代数的データ型ではなく、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。）

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 °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）	`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°)
delay *m*	return 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°

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 が付くこと、関数適用は (>>=) 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。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）	`return (\ x ->` `  return (\ y ->` `  return (x ⊗ y)))`

この `comp` 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

```
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。

```
fact = \ n ->
  ((return (\ x -> return (\ y -> return (x == y)))
>>=
      \ _f -> return n >>= \ _x -> _f _x)
    >>= \ _f -> return 0 >>= \ _x -> _f _x)
   >>=
   \ _b ->
     if _b then return 1 else
       (return (\ x ->
          return (\ y -> return (x * y))) >>=
          \ _f -> return n >>= \ _x -> _f _x)
         >>=
         \ _f ->
           (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 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。

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 UtilI プログラムをコンパイルして実行する（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。

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 つだけ更新可能な "参照" 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、`__`という結果が得られる。

> **参考:**なお、Util に参照を指す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参照に割り当てる）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、参照を指す変数として宣言されていると仮定する。

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。

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 (\ s -> (a, s))
  (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))
```

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

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

ここで set は状態を書き換える、また get は状態の値の一部を複製する関数である。

また

```
evalST st s = fst (unST st s)
```

と定義する。

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

```
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	$\acute{p}$ `>>= \ _p ->` $\acute{m}$ `>>= \ _x ->` `set _p _x`
`get` p	$\acute{p}$ `>>= \ _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 Util 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
```

`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、パラメーター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

```
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)
```

ファイル MyIO.hs

```haskell
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

```haskell
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
```

をコンパイルした結果は、

```
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、補助関数を以下のように定義したとき、

```
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.
```
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.
```
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）を模倣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 = Nothing | Just 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 = k a
  Nothing  >>= k = Nothing
```

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)
             x `mydiv` n
```

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 では `1 / 0` の部分式は＿＿＿＿＿に全体の結果が 0 となるが、 UtilErr では次のような 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。

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

```
1 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、

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

```
  2         `mplus` (return 99999)
```

であり、`bar 0` の結果は `Just 99999.0` である。

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。

```
 1 -- Prelude で宣言済み
 2 instance Monad [] where
 3   return a = [a]
 4   []     >>= k = []
 5   (x:xs) >>= k = k x ++ (xs >>= k)
 6
 7 -- モジュール Control.Monad で宣言済み
 8 instance MonadPlus [] where
 9   mzero = []
10   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。

問 5.7.1 `[]` 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

```
1 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)
```

をコンパイルすると、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 でコンパイルすると、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 5.7.2 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、一度初期化すれば二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。

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

```
- myAppend([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 `myAppend` の逆向きの計算もできるということである。

```
?- myAppend(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 `myAppend` 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

第P章 Prolog 超簡易入門

Prologは、＿＿＿＿と呼ばれるプログラミング言語の仲間のうち、もっとも代表的なものである。論理型言語では、「～ならば～」という論理式の集まりをプログラムとみなし、論理式の証明をプログラムの実行とみなす。

Prolog などの論理型言語に特徴的な概念としては、＿＿＿＿＿＿（単一化）、＿＿＿＿＿＿（後戻り）、＿＿＿＿などが挙げられる。

P.1 Prolog でのプログラミング

Prolog では「～ならば～」という論理式の集まりをプログラムとみなすが、この「～」の部分を構成するのが、次のような形式である。

```
述語(項1,…, 項n)
```

このかたちを＿＿＿＿(アトム（Atom）、 複合項などと呼ぶこともある。)と呼ぶ。「項」は数・文字列などの定数や変数、あるいはリストなどのデータ構造などである。「述語」は直観的には項 $_1$,..., 項$_n$ の間の関係を表す。例えば、`father(adam, cain)` という素論理式は「Adam は Cain の父である」という関係を表す。

Prolog のプログラムは、＿＿＿＿（Horn clause）と呼ばれる形式の集まりである。ホーン節とは、

```
素論理式0 :-  素論理式1, …, 素論理式n.
```

という形である。（最後に必ずピリオド（.）が必要である。）「:-」は左向きの矢印（←）と考えれば良い。つまり、これは素論理式$_1$, ..., 素論理式$_n$ がすべて成り立つならば、素論理式$_0$ も成り立つという規則を表している。

ホーン節の「:-」の左側には素論理式は一つのみであり、これを＿＿＿（head）と呼ぶ、「:-」の右側の素論理式の並びは＿＿＿＿（body）と呼ぶ。

例えば、

```
1 grandchild(X, Z) :- child(X, Y), child(Y, Z).
```

は、`X` と `Y`, `Y` と `Z` の間に `child` という関係がある（`X` が `Y` の子であり、 `Y` が `Z` の子である）ならば、`X` と `Z` の間に `grandchild` という関係がある（`X` は `Z` の孫である）という規則を述べている。ここで `X`, `Y`, `Z` は変数である。Prolog の変数は`X`, `Xs`, `Y` のように、アルファベットの＿＿＿（あるいはアンダースコア「_」）からはじまる識別子（名前）を使用する。一方、小文字からはじまる識別子は、述語や構成子などの定数に利用される (Haskell と全く逆なので注意す

る。)。変数はどのような項に具体化しても良いので、変数を使用した規則は、実際には無数の規則を表していることになる。

ホーン節のうちボディーがないものを　　（fact）と言う。このときは、「:-」も省略する。（文末のピリオドは必要である。） 例えば、

```
child(hidetada, ieyasu).
```

は hidetada（秀忠）が ieyasu（家康）の子である、という事実を表明している。ここで hidetada, ieyasu は小文字からはじまるので定数である。

Prolog のプログラムは、このようなホーン節（事実を含む）を集めたものである。例えば徳川家の家系図を表すプログラムの一部は次のようになる。

```
grandchild(X, Z) :- child(X, Y), child(Y, Z).

child(hideyasu, ieyasu).
child(hidetada, ieyasu).
child(yoshinao, ieyasu).
child(yorinobu, ieyasu).
child(yorifusa, ieyasu).

child(iemitsu, hidetada).
child(tadanaga, hidetada).
child(masayuki, hidetada).

child(ietsuna, iemitsu).
child(tsunayoshi, iemitsu).
```

このようなプログラムをファイルに作成し、処理系にロード (Prolog では伝統的に consult という。)する。例えば Windows 上で動作する SWI-Prolog という処理系ではメニューの「File」—「Consult ...」 でプログラムファイルをロードすることができる。プログラムは、このようなホーン節の集まりに対して質問を発することにより起動される。Prolog の処理系は「?- 」というプロンプトを出力するので、このあとに質問を入力する。（質問も最後に必ずピリオドが必要である。）

```
?- grandchild(hidetada, ieyasu).

No
?- grandchild(iemitsu, ieyasu).

Yes
```

1 番目の質問は、Hidetada（秀忠）は Ieyasu（家康）の孫か? という質問である。これはプログラム中の規則から導出できないので、処理系は No と答えている。2 番目の質問は、Iemitsu（家光）は家康の孫か? という質問である。これは、child(hidetada, ieyasu). と child(iemitsu, hidetada). という二つの事実と

```
grandchild(X, Z) :- child(X, Y), child(Y, Z).
```

というホーン節から導出できるので Yes と答えている。

さらに Prolog では質問の中に変数を含めることができる。すると Prolog の処理系は、その質問を成り立たせる＿＿＿＿＿＿を出力する。例えば、

```
1 ?- grandchild(X, ieyasu).
```

という質問に対しては、

```
1 X = iemitsu
```

という解を出力する。ここで、リターンキーを押すとこれで終ってしまうが、「;」を入力すると、さらに別解を表示させることができる。

```
1 X = iemitsu ;
2
3 X = tadanaga ;
4
5 X = masayuki ;
6
7 No
```

質問には一般に複数の素論理式を並べることができる。

```
?-  素論理式1, …, 素論理式n.
```

このような形を＿＿＿＿＿＿と呼ぶ。直観的には、並べた素論理式がすべて成り立つか?（成り立たせるような変数への代入が存在するか?）という質問を表している。

P.2 単一化（ユニフィケーション）と後戻り

この節では、Prolog のプログラムの実行方法を解説する。解説を簡単にするために、まず、最初にいくつかの言葉を定義しておく。

2 つ以上の（通常変数を含む）素論理式があり、変数に適切な代入をして、これらの素論理式をまったく同一のものにすることができるとき、これらの素論理式は＿＿＿＿＿＿（unifiable）であるという。また、その時の代入を＿＿＿＿＿＿（unifier）という。例えば、次の 2 つの素論理式

```
p(a, Y, Z) と p(X, b, Z)
```

は `X = a, Y = b` を単一化代入として単一化可能である。単一化可能なときは、通常必要なのは最汎（最も一般的な）単一化代入（most general unifier, ＿＿＿＿）である。例えば、上の例では、`X = a, Y = b, Z = c` という代入も単一化代入であるが、前者の方がより一般的である。実際、この例の場合は `X = a, Y = b` が MGU になる。

プログラム中のホーン節

```
P :- Q1,  Q2, …, Qn,
```

のヘッド P が素論理式（G）と単一化可能なとき、このホーン節は G に＿＿＿＿という。

Prolog のプログラムの実行方法は、次のようにまとめられる。

1. ゴール節中の＿＿＿＿＿＿素論理式に対し、適用できるホーン節を選ぶ。適用できるホーン節が複数あるときは、プログラム中に＿＿＿＿＿＿ホーン節から試みる（Prolog を定理証明系として見たときには、ここで、「最も左」、「先に書かれている」という選択をするために、証明系としての力が弱くなってしまう。つまり、このような制限をしなければ証明できるはずの論理式が証明できなくなってしまう。しかし、プログラミング言語として見たときは、効率を確保するために必要な制限である。）。
2. 選んだ適用可能なホーン節に対して、その MGU をゴール節の残りの素論理式に適用し、さらにゴール節中の最左素論理式を、上で選んだホーン節のボディに MGU を適用したものと置き換える。（ホーン節のボディーがないときは、最左素論理式を消す。）
3. もし適用できるホーン節がなければ、後戻り（＿＿＿＿＿＿）して、ひとつの前のゴール節中の素論理式がある状態からやり直す。そして、その素論理式に適用できるホーン節のうち次の候補を選ぶ。
4. ゴール節に素論理式がなくなれば、プログラムの実行を終了し、その時得られた変数への代入を表示する。素論理式がまだあれば、1. に戻る。

具体的に、

```
1 ?- grandchild(X, ieyasu).
```

というゴール節での動作を説明することにする。まずこのゴール節の最左（1 つしかないが）素論理式は `grandchild(X, ieyasu)` である。これに適用できるホーン節は、今のプログラムでは、

```
grandchild(X, Z) :- child(X, Y), child(Y, Z).
```

しかなく、最汎単一化代入は `Z = ieyasu` である。するとゴール節は

```
1 ?- child(X, Y), child(Y, ieyasu).  … ①
```

に変換される。

次に、この最左の `child(X, Y)` に適用できる最初のホーン節は、

```
child(hideyasu, ieyasu).
```

であり、MGU は `X = hideyasu, Y = ieyasu` である。すると、ゴール節は、

```
?- child(ieyasu, ieyasu).
```

に変換される。しかし、このゴール節に適用できるホーン節はない。

そこで、上の ① のところまでバックトラックし、次の候補である

```
child(hidetada, ieyasu).
```

の適用を試みる。しばらくのあいだ同様にバックトラックが続き、最終的に① のゴール節に対し、

```
child(iemitsu, hidetada).
```

というホーン節を選ぶ。そのとき MGU は `X = iemitsu, Y = hidetada` となる。すると、ゴール節は、

```
?- child(hidetada, ieyasu).
```

に書き換えられる。これはすぐに適用できるホーン節が見つかり、ゴール節が空になって、結果として代入:

```
X = iemitsu
```

が出力される。ここで「;」が入力されると、またバックトラックが起こる。

ホーン節の適用は命令型言語・関数型言語の関数呼出しのようなものだと考えることもできるが、単一化により呼び出される側（ホーン節のヘッド）から呼び出す側（ゴール節）に情報が流れることがある、というところが論理型言語に特徴的なところである。例えば、

```
?- child(X, ieyasu).
```

という呼出しに対しては、`X = hideyasu`（**あるいは** `hidetada, yoshinao ...`）という情報が、呼び出された側から、呼び出した側に伝えられる。

Prolog の変数は命令型言語の変数と異なり、いったん単一化により値が代入されれば、以降は値を変更されることはない。ただし、（純）関数型言語の変数とも異なり、最初はいったん不定 (unknown) な状態を取ることができる。このような振舞いをする変数は一般に＿＿＿＿(logical variable) と呼ばれる。

P.3 Prologでのリスト処理

Prolog でのリストの記法は要素を「,」で区切り、角括弧（`[` と `]`）で囲んで、`[1, 2, 3]` のように書く。

また、先頭の要素が `X` で、先頭を除く残りの要素からなるリストが `Xs` であるようなリストは＿＿＿＿と表記される。これは Haskell の `x:xs` という書き方に相当する。また、`[1, 2, 3]` は＿＿＿＿＿＿＿＿の略記法である。

Prolog でのリストを連接 (append) するプログラムは次のように記述できる。

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

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。

例えば, `[1, 2]` と `[3, 4]` の連接は次のように求められる。

```
?- myappend([1, 2], [3, 4], X).

X = [1, 2, 3, 4] ;

No
```

問 P.3.1 実際に、上のような結果が出ることを、上に示した Prolog の実行方法で1ステップずつ確認せよ。

上の問を解いてみるとわかるが、変数 Z_0 は後ろの要素が不定（変数）のまま、前の要素から順に定まっていく。

後ろの要素がもっと後で定まるようなプログラムを書くことも可能である。例えば、

```
myconcat([], []).                                % ①
myconcat([Xs|Xss], Ys)
   :- myappend(Xs, Zs, Ys), myconcat(Xss, Zs). % ②
```

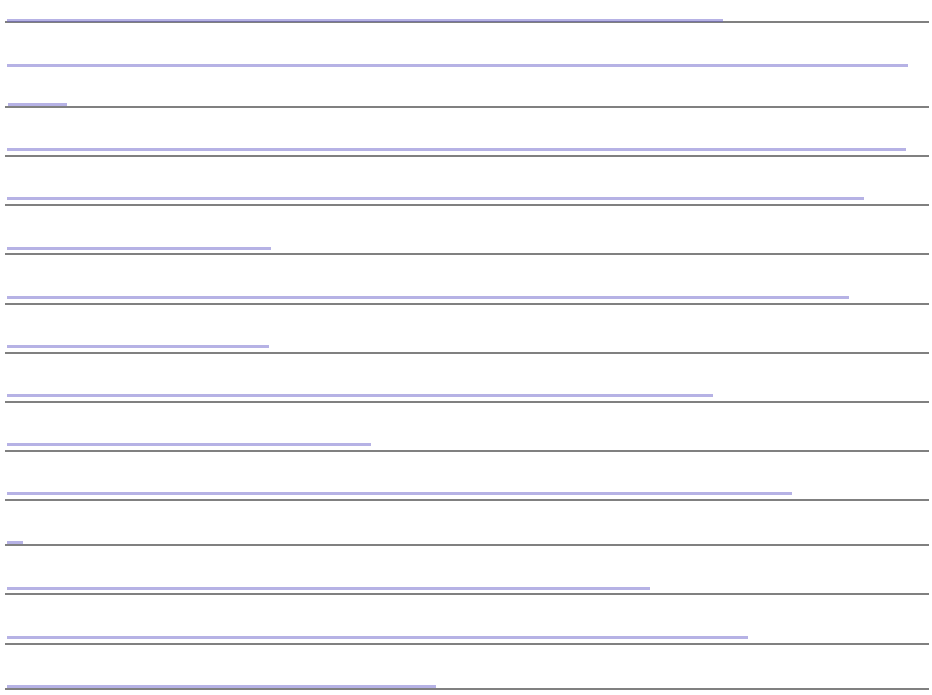
これは、リストのリストを平坦なリストに変換するプログラムである。

```
?- myconcat([[1, 2], [3, 4, 5], [6, 7]], Ys).

Ys = [1, 2, 3, 4, 5, 6, 7] ;

No
```

問 P.3.2 実際に、上のような結果が出ることを、上に示した Prolog の実行方法で1ステップずつ確認せよ。



この `myappend` の第 2 引数の `Zs` はあとの `myconcat` の呼出しで値が定まる。また `myappend` の第 3 引数の `Ys` はしばらくの間、最後が論理変数で終る形で扱われることになる。

このように尾 (tail) の部分が論理変数になっているリストを＿＿＿＿（open list）という。開リストは Prolog でリストを扱う時に良く使われるイディオムである。

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

```
?- myappend(X, Y, [1, 2, 3]).

X = []
Y = [1, 2, 3] ;

X = [1]
Y = [2, 3] ;

X = [1, 2]
Y = [3] ;

X = [1, 2, 3]
Y = [] ;

No
```

この例では連接して `[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クが起こり別解を表示する。

> **問 P.3.3** 逆向きの `myappend` の計算ができることを、前節で示した Prolog の実行方法で実際に 1 ステップずつ確認せよ。

第6章 接続 (continuation)

この章では、**goto** や **break**, **continue** などのジャンプ命令に意味を与えるために　(continuation)の概念を導入する。

> **（補足）** “continuation” の日本語訳は「　」のほうが一般的だが、ここでは「接続」を採用する。

接続は直観的には（ジャンプなどがないときに）　を表す。例えば、次のような C のプログラムでは:

```
int main(int argc, char** argv) {
    printf("The result is %d.\n", 1 + fact(10));
    return 0;
}
```

下線の部分の接続は、プログラムの残りの部分 — 1 を足してその結果を出力する、という操作である。

どのようなプログラム処理系でも、プログラムの実行中は何らかの形でこの接続の情報を保持しているはずである。機械語レベルでは、接続は　（program counter）と　の組に相当する。ジャンプ命令を解釈するためには、この接続の概念を明示的に扱う必要がある。

また、　や　など一部の言語は、接続をプログラマーが明示的に扱うことを可能にしている。これによってコルーチン（coroutine）など、さまざまな自明でない制御構造を実現することができる。

この章では接続の概念を導入し、そのさまざまな応用を紹介する。

6.1 接続のモナド

接続（continuation）のモナドは単独では次のような型になる。
ファイル Cont.hs

```
newtype K r a = K ((a -> r) -> r)

unK :: K r a -> (a -> r) -> r
unK (K c) = c

instance Monad (K r) where
  return a      = K (\ c -> c a)
  (K m) >>= k   = K (\ c -> m (\ a -> unK (k a) c))
  -- K, unK がなければ、
  -- m >>= k = \ c -> m (\ a -> k a c)
```

直観的には `r` が"結果"の型、`a -> r` が接続（"以後実行すべき操作"）の型になる。`return a` は、接続（c）に `a` 渡す。`m >>= k` は、接続（c）の `m` という接続（`\ 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。

6.2 UtilCont — 接続の導入

Util に **break**, **continue** などを導入するために、`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。
ファイル ContType.hs

```haskell
data Expr = Const Target | Var String
          | If Expr Expr Expr | While Expr Expr
          | Let [Decl] Expr | Val Decl Expr
          | Lambda String Expr | Delay Expr
          | App Expr Expr
          -- ここまでは、Util1と同じ
          | Begin [LabeledExpr]      -- ブロック
          | Break                    -- break 文
          | Continue                 -- continue 文
          | Goto String              -- goto 文
          deriving Show
type LabeledExpr = (Maybe String, Expr) -- ラベル付きの式
```

これに対する具象構文としては、

Expr → **begin** *LabeledExprSeq*
| **break** | **continue** | **goto** *Var*
LabeledExprSeq → *LabeledExpr* **end** | *LabeledExpr* ; *LabeledExprSeq*
LabeledExpr → *Expr* | *Var* : *Expr*

を想定する。

実際の UtilCont では接続とともに変数の破壊的代入や入出力も扱いたいので、計算のモナドは、`K` そのものではなく、"状態への操作"を"結果"として持つ `K (WithIO s -> r) a`（と同型の型）とする。ここで、`s` は状態の型である。

ファイル Cont.hs

```haskell
newtype KIO r s a
  = KIO ((a -> WithIO s -> r) -> WithIO s -> r)

unKIO :: KIO r s a -> (a -> WithIO s -> r)
                   -> WithIO s -> r
unKIO (KIO m) = m
```

この `KIO` の定義にあわせて、`set` など状態に関する関数も書き直しておく。
ファイル Cont.hs

```haskell
instance MyState (KIO r) where
  get p    = KIO (\ c (s, i, o) -> c (fst (p s)) (s, i, o))
  set p v  = KIO (\ c (s, i, o) -> c () (snd (p s) v, i, o))

instance MyStream (KIO r s) where
  readChar   = KIO (\ c (s, ch : i, o) -> c ch (s, i, o))
  eof        = KIO (\ c (s, i, o) -> c (null i) (s, i, o))
  writeStr v = KIO (\ c (s, i, o) -> c () (s, i, o ++ v))

abort :: (WithIO s -> r) -> KIO r s a
abort f = KIO (\ c -> f)
```

すると、`set p v` は、状態の `p` で表される位置に `v` をセットし、`()` と新しい状態を接続に渡す。

また、`abort f` は現在の接続を無視して `f` 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。

`Const`, `Var`, `Let` などに対しては `comp` は変更する必要はない。変更された部分のうち、 `Goto`, `Break`, `Continue` に対する `comp` の定義は以下のようになる。
ファイル ContCompiler.hs

```haskell
comp (Goto lbl)         = mkGoto lbl "()"
comp Break              = mkGoto "_break" "()"
comp Continue           = mkGoto "_while" "_break"

mkGoto lbl v = TApp1 (TVar "abort")
                     (TApp1 (TVar lbl) (TVar v))
```

ここで **goto**, **break**, **continue** について、変換前と変換後をそれぞれ Util と Haskell の文法で記述すると次の表になる。

ソース（Util）	ターゲット（Haskell）
goto label	abort (label ())
break	abort (_break ())
continue	abort (_while _break)

一方 **goto** `label`, **break** はそれぞれ、現在の接続は無視して、 `label`, `_break` という識別子に束縛されている接続を起動する式に翻訳される。これが"ジャンプ"に相当する。

また **continue** も、現在の接続は無視して、`_while` という識別子に束縛されている計算に `_break` という接続を渡す。

ところで **while** 〜 **do** 〜 に対しては、**break** に対応する接続を変数に格納する必要があるため、定義がやや複雑になる。
ファイル ContCompiler.hs

```
comp (While e1 e2)      = compWhile e1 e2

compWhile e1 e2 = TApp1 (TVar "KIO")
  (TLambda1 "_break"
    (TLet [(PVar "_while", TApp1 (TVar "unKIO") body)]
          (TApp1 (TVar "_while") (TVar "_break"))))
  where body = comp e1 `TBind` TLambda1 "_b"
          (TIf (TVar "_b") (comp e2 `TBind`
                 TLambda1 "_"
                 (TApp1 (TVar "KIO") (TVar "_while")))
                 (TReturn (TVar "()")))
```

ソース（Util）	ターゲット（Haskell）
while c **do** t	KIO (\ _break -> **let** KIO _while = $\hat{c}$ >>= \ _b -> **if** _b **then** $\hat{t}$ >>= \ _ -> KIO _while **else** return () **in** _while _break)

ここで、_break は＿＿＿＿＿＿＿＿を表す接続で、_while _break は＿＿＿＿＿＿を表す接続である。これらの接続が、それぞれ **break**, **continue** に対応する。

例えば、UtilCont プログラム（右は対応する C プログラム）：

```
foo y = begin
  xP := 1; yP := y;
  while get yP > 0 do
  begin
    val x = get xP in
    val y = get yP in
    if y == 10 then
     break
    else if y == 3 then
    begin
      yP := y - 1;
      continue
    end else ();
    xP := x * y;
    yP := y - 1
  end;
  get xP
end
```

```
int foo(int y) {
  int x = 1;
  while (y > 0) {



    if (y == 10)
      break;
    else
    if (y == 3) {
      y--;
      continue;
    }
    x = x * y;
    y--;
  }
  return x;
}
```

は階乗の関数の変な変形であるが、これをコンパイルすると、次の Haskell プログラムが得られる。

```
foo = \ y ->
  set xP 1              >>= \ _ ->
  set yP y              >>= \ _ ->
  KIO (\ _break ->
    let KIO _while
        = get yP        >>= \ y ->
```

```
          if y > 0 then
              get xP              >>= \ x ->
              get yP              >>= \ y ->
              (if y == 10 then abort (_break ())
                 else if y == 3 then
                      set yP (y - 1)    >>= \ _ ->
                      abort (_while _break)
                   else return ()) >>= \ _ ->
              set xP (x * y)      >>= \ _ ->
              set yP (y - 1)      >>= \ _ ->
              KIO _while
            else return ()
      in _while _break)  >>= \ _ ->
  get xP
```

この foo の型は`Integer -> KIO a (Integer,Integer) a Integer` であるから、値を取り出すためには、整数と初期接続（通常、`\ a s -> a`）、初期状態（`((0,0),"","")` など）を渡す必要がある。ここで

```
evalKIO m s = unKIO m (\ a s -> a) (s,"","")
```

と定義すると、`evalKIO (foo 9) (0,0)` の結果は、＿＿＿＿に、`evalKIO (foo 11) (0,0)` は ＿になる。（参考: $9! = 362880$）

さらに **goto** に対する意味を与えるためには、ブロック（**begin** ～ **end**）のなかで、"ラベル"に適切な接続を与える必要があるが、`Begin` に対する `comp` の定義は長くなってしまうので、ここに示さず、 変換前と変換後の形の例のみを示す。

ソース（Util）	ターゲット（Haskell）
begin	KIO (\ _end -> **let**
lbl1: s_1	lbl1 = \ _ -> unKIO $\mathring{s}_1$ lbl2
lbl2: s_2	lbl2 = \ _ -> unKIO $\mathring{s}_2$ lbl3
lbl3: s_3	lbl3 = \ _ -> unKIO $\mathring{s}_3$ _end
end	**in** lbl1 ())

この s_1, s_2, s_3 の中には、 **goto** lbl1, **goto** lbl2, **goto** lbl3 が含まれているかもしれない。ターゲット（Haskell）プログラム中の識別子 lbl1,lbl2, lbl3 に束縛されているのはそれぞれ、同名のラベル lbl1, lbl2, lbl3 に対応する接続である。

例えば、次の UtilCont プログラム（右は対応する C プログラム）:

```
bar _ = begin
    xP := 1;
  label1:
    if get xP > 100
      then goto label2
      else ();
    xP := get xP * 2;
    goto label1;
```

```
int bar(void) {
   int x = 1;
 label1:
   if (x > 100)
     goto label2;

   x = x * 2;
   goto label1;
```

```
  label2:
    get xP
  end
```

```
 label2:
   return x;
}
```

は次のような Haskell プログラムにコンパイルされる。

```
bar = \ _ ->
        KIO (\ _end ->
          let label1 = \ _ -> unKIO (
                get xP              >>= \ x ->
                (if x > 100 then abort (label2 ())
                   else return ()) >>= \ _ ->
                get xP              >>= \ x ->
                set xP (x * 2)      >>= \ _ ->
                abort (label1 ())) label2
              label2 = \ _ -> unKIO (get xP) _end
          in unKIO (set xP 1) label1)
```

そして、`evalKIO (bar ()) (0,0)` を評価すると、結果は ___ になる。

問 6.2.1 次の C の関数とほぼ同等な Haskell の関数をモナドを用いて作成せよ。

```
int hoge(int n) {
  int i = 1, sum = 0;
  while (i <= n) {
    sum = sum + i;
    if (sum > 21) {
      sum = 0;
      break;
    }
    i = i + 1;
  }
  return sum;
}
```

今日では構造化プログラミングが一般的になり、`goto` 文はほとんど使われることはない。つまり `goto` 文を多用したスパゲッティーコードは、結局、相互再帰（例えば、A の定義に B を利用し、B の定義に C を利用し、C の定義に A を利用するような状況）を多用することと同等であり、プログラムの意味がわかりにくくなると言える。

6.3 `callcc` とは

ここで紹介する `callcc` は Scheme や Ruby などが採用している、プログラマーが接続を直接操作することができるプリミティブである。Scheme では `call-with-current-continuation` または、省略して `call/cc` と書く。（Scheme では "-" も "/" も関数名の一部として使えることに注意する。）

1 引数の関数 f に対して、`callcc f` のように使用すると ________ を引数として、f を呼び出す。f のなかで、この接続を呼び出せば、呼出しの接続は無視されて（= ジャンプして）、`callcc` が呼ばれたときの接続にその値が返される。一方 f が接続を呼び出さなければ、f 自身の戻り値が `callcc` 式全体の戻り値になる。まず、トリビアルな例として Util の記法で

```
baz x = callcc (\ k ->
    100 + (if x >= 10 then x else k x))
```

という関数を考える。 ここで `baz 10` を評価すると普通に足し算が計算され、値は ___ になる。一方、`baz 1` の場合は、接続 `k` が 呼び出されるので 100 を足す部分はスキップされて、戻り値は _ となる。

また `callcc` のよくある使い方は、`try` 〜 `catch` と同じような大域脱出である。

```
multlist xs =
  let aux xs k = begin
        xP := 1; yP := xs;
        while not (null (get yP)) do
          val y = get yP in val n = head y in
          if n == 0 then k 0 else
            begin xP := get xP * n; yP := tail y;
                  writeStr " ";
                  write n
            end;
        get xP
    end in
  val result = callcc (\ k -> aux xs k) in begin
    writeStr "; result = ";
    write result
  end
```

この関数はリストの要素の掛け算を求める。要素の中に 0 が見つかると、大域脱出して `multlist` 全体は _ と出力する。

しかし、このような大域脱出だけならば、言語の仕様に `callcc` のような大がかりな仕掛けをいれておく必要はない。`callcc` の本当の価値はコルーチンなどの普通でない制御構造を実現できるところにある。

6.4 コルーチン（coroutine）

コルーチンとは、2 つ以上のプログラムの実行単位が、 ________________ ながら実行されていく方式のことである。サブルーチン（subroutine）のように、実行単位の間に主と副といった従属関係はなく、コルーチンを構成する個々のルーチンは互いに対等な関係である。

例えば、

```
increase n k =
    if n > 10 then ()
    else begin writeStr " i:"; write n;
    increase (n + 1) (callcc k) end;
decrease n k =
    if n < 0 then ()
    else begin writeStr " d:"; write n;
    decrease (n - 1) (callcc k) end
```

という 2 つの関数を定義して

```
increase 0 (decrease 10)
```

という式を実行すると、

と出力される。

注意: 実は、この Util プログラムを Haskell に変換すると型エラーになり、そのままではコンパイル・実行できない。これは、`callcc` の引数の型と戻り値の型が同一になる必要があるからである。

いくつかトリッキーな変換をすると、意味を変えずに型付け可能な定義に書き換えが可能で、上記のような実行結果になる。しかし、ここはコルーチンのアイデアを説明するのが本旨なので、型付けをするためのトリックの詳細には立ち入らないことにする。

なお、Scheme では、

```
(define (increase n k)
   (if (> n 10) '()
   (begin (display " i:") (display n)
          (increase (+ n 1) (call/cc k)))))
(define (decrease n k)
   (if (< n 0) '()
   (begin (display " d:") (display n)
          (decrease (- n 1) (call/cc k)))))
```

のように対応する関数を定義して

```
(increase 0 (lambda (k) (decrease 10 k)))
```

という式を実行すると上記の出力が得られる。

ここでは `increase` と `decrease` という 2 つの関数が交互に実行されていることがわかる。スレッドと似ているが、2 つのルーチンが同時に実行される訳ではない。

また `callcc` は、コルーチンの他にこれまでに紹介したエラー処理（try ～ catch）や非決定性などのプリミティブも、`callcc` を用いて定義できることが

わかっている。ある意味でオールマイティーなプリミティブである。しかし、その詳細の解説については、ここでは割愛する。

6.5 callcc の表現

我々の言語 UtilCont に callcc 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 に対応する関数の定義は次のようになる。
ファイル Cont.hs

```
callcc :: ((a -> KIO v s b) -> KIO v s a) -> KIO v s a
callcc h = KIO (\ c -> let k a = KIO (\ d -> c a)
                       in unKIO (h k) c)
-- KIO, unKIO がなければ
-- callcc h = \ c -> let k a = \ d -> c a
--                       in h k c
```

この callcc の定義中で用いられている k は現在の接続（d）を捨て、キャプチャーされた接続（c）を呼び出すという関数である。

コンパイラーは単に callcc という名前の UtilCont の関数を Haskell の callcc にコンパイルすれば良い。

ソース（Util）	ターゲット（Haskell）
callcc *m*	*m*' >>= \ _x -> callcc _x

また、head, tail, null, not, show などの 1 引数で副作用を持たない関数は、次のようにコンパイルされる。

ソース（Util）	ターゲット（Haskell）
funWithOneArg *m*	*m*' >>= \ _x -> return (funWithOneArg _x)

例えば、先に紹介した UtilCont プログラム multlist をコンパイルすると、次の Haskell プログラムが得られる。

```
multlist = \ xs ->
  let aux = \ xs ->
        return (\ k ->
          set xP 1                >>= \ _ ->
          set yP xs               >>= \ _ ->
          KIO (\ _break ->
             let KIO _while =
                   get yP                 >>= \ y ->
                   if not (null y) then
                       get yP             >>= \ y ->
                       (if head y == 0 then k 0 else
                          get xP          >>= \ x ->
                          set xP (x * head y)
                                          >>= \ _ ->
                          set yP (tail y) >>= \ _ ->
                          writeStr " "    >>= \ _ ->
                          write (head y)) >>= \ _ ->
                       KIO _while
```

```
                    else return ()
                in _while _break) >>= \ _ ->
          get xP)
    in callcc (\ k -> aux xs >>= \ _f ->
                        _f k)      >>= \ result ->
       writeStr "; result = "    >>= \ _ ->
       write result
```

このとき、プログラムの出力を取り出すために

```
evalKIO2 m s = unKIO m (\ a (_,_,o) -> o) (s,"","")
```

と定義すると、`evalKIO2 (multlist [1,2,3,4,5]) (0,[])` は、" `1 2 3 4 5; result = 120`" となり、

一方、`evalKIO2 (multlist [1,2,3,0,4,5]) (0,[])` は" `1 2 3; result = 0`" となる。つまり、0 が現れた時点で乗算を打ち切っていることがわかる。

6.6 さらに詳しく知りたい人のために...

接続に関する文献は数多くあるが、文献（Reynolds 1993） は接続の「発見」について、振り返っている珍しいものである。文献（Filinski 1994） は、`call/cc` がある意味で「オールマイティー」であることについての説明 を与えている。 文献（Sekiguchi et al. 2001）は、 Java などの命令型言語に、`call/cc` のような接続を扱うオペレーター を導入する方法を述べている。 文献（Erköok & Launchbury 2000）は `mfixU` などの不動点演算子について解説している。

（Reynolds 1993） John C. Reynolds, "The Discoveries of Continuations" Lisp and Symbolic Computation, 6, (233–247). 1993 年

（Filinski 1994） Andrzej Filinski, "Representing Monads" 21st ACM Symposium on Principles of Programming Languages. 1994 年

（Sekiguchi et al. 2001） T. Sekiguchi, T. Sakamoto, and A. Yonezawa, "Portable Implementation of Continuation Operators in Imperative Languages by Exception Handling" Advances in Exception Handling Techniques. Springer-Verlag, LNCS 2022. 2001年
http://www.yl.is.s.u-tokyo.ac.jp/amo/

（Erköok & Launchbury 2000） Levent Erköok, and John Launchbury, "Recursive Monadic Bindings" Proc. of the International Conference on Functional Programming. 2000 年

第S章 Scheme 超簡易入門

Scheme は、Lisp の一方言である。Scheme は関数型言語であるが、Haskell と異なり、変数への代入など命令的な特徴を残している。また遅延評価ではなく、関数の引数を先に評価する、先行評価を採用している。

S.1 Scheme でのプログラミング

関数適用

関数適用 (function application)は次のような形である。

- （関数 引数$_1$ 引数$_2$ … 引数$_n$）のような＿＿＿＿(parenthesis) でくくった式の列

Scheme では ＋ や × などの算術演算子に、通常の＿＿＿＿(infix notation) ではなく、＿＿＿＿(prefix notation) を用いることが特徴的である。例えば、`(+ 1 2)`という式では、「+」が関数（function）, `1` と `2` が引数である。

変数と代入

例えば、

```
(define x 5)
```

という式で、5 という値の入った “x”という名前の変数を用意する。これ以降は `x` という変数は `5` に評価される。

Scheme の場合、変数名の中には、アルファベット、数字の他に

```
+ - . * / < = > ! ? : $ % _ & ~ ^
```

などの記号を用いることができる。（もちろん空白はダメ）アルファベットの大文字と小文字は＿＿＿＿。（つまり、`Japan` と `japan` は＿＿変数である。）

また `set!` という命令によって、変数の値を変更する（代入するという）ことができる。（C 言語の 「=」演算子に対応する。）

```
(set! x 4)          ; 変数 xの値を 4に変更する。
                    ; それ以前に xを defineしておく必要がある。
```

これは、Scheme が＿＿＿＿としての側面を持つことを示す。なお、Scheme では「;」から行末までがコメントである。

リスト

リストを入力するためには、組み込み関数 `list` を用いる。`list` は任意の数の引数を取ることができる。

```
> (list 1997 5 6)
(1997 5 6)
> (list "kagawa" "university")
("kagawa" "university")
```

単に `(1997 5 6)` と入力すると、Scheme の処理系は、`1997` という関数を `5` と `6` という引数に適用しているのだと判断してしまう。

> このように、Scheme（一般に Lisp）では小括弧「`(`」、「`)`」が 2 つの意味に使われる。ユーザが入力するときは「＿＿＿＿」の意味に、処理系が出力するときは「＿＿＿＿」の意味になる。もっと正確に言うとユーザが「リスト」を入力すると、処理系はそれを「関数適用」だと解釈するのである。
>
> このような処理系の振舞いは Lisp の強力さの源であるが、一方で混乱のもとでもある。

上記のデータは「`'`」（クォート記号・引用記号）を用いて次のようにも入力できる。

```
> '(1997 4 22)
(1997 4 22)
```

「**`'`式**」は、「`(quote` **式**`)`」とも書く。（むしろ、後者が正式な書き方である。）

```
> (quote (1997 5 6))
(1997 5 6)
```

ここで `quote` は、その＿＿＿＿＿＿＿＿＿＿ことを表す。だから、*(1997 5 6)* は関数適用ではなくリストと解釈される。

空リスト（要素を 1 つも含まないリスト）は `'()` または `(list)` のように入力する。

```
> '()
()
> (list)
()
```

また `cons`（＿＿＿＿＿＿と読む）, `car`（＿＿＿＿＿＿と読む）, `cdr`（＿＿＿＿＿＿と読む）などが、リストを操作するための最も基本的な関数である。 ここで `cons` はリストを組み立てるための関数、`car` と `cdr` はリストを分解するための関数である。

- `cons` — 第 2 引数として与えられるリストの先頭に、第 1 引数として与えられる要素を付け加えたリストを返す
- `car` — リストの先頭の要素を返す
- `cdr` — リストの先頭を除いた残り（のリスト）を返す
- `null?` — リストが空ならば真、空でなければ偽を返す

関数定義

関数の定義には次の形式の `define` を用いる。

```
(define (関数名 変数1 … 変数n) 定義)
```

変数$_1$ … 変数$_n$はこの関数の仮引数である。

```
1 > (define (square x) (* x x))
2 square
3 > (square 4)
4 16
```

条件判断

条件判断は次のような形式で行なう。

```
(if 条件式 式1 式2)
```

条件式が＿＿＿＿を、＿＿＿＿を評価（計算）する。（C の **if** 文と異なり、値を返すことに注意する。むしろ、C の「`?:`」オペレーターに対応する。）

逐次実行

```
(begin 式1 式2 … 式n)
```

式$_1$から、式$_n$を順に評価し、最後の式$_n$の値を全体の値として返す。通常、式$_1$から、式$_{n-1}$は＿＿＿＿のために実行される。 C や JavaScript のブロック `{ 〜 }` と意味は似ているが、C や JavaScript のブロックは“文”の一種であるので値を持たないのに対し、Scheme の `begin` 式は値を持つ。

なお、関数の定義の本体で、

```
1 (define (hen_na_square x)
2     (begin (set! x (* x x))
3            x))
```

のように順に式を評価するときは、上のように `begin` を使う必要はなく、

```
1 (define (hen_na_square x)
```

```
2       (set! x (* x x))
3        x)
```

のように単に式を並べて書くだけで良い。（これを “暗黙”の begin という。）

局所変数 (let)

関数の定義の他に let という構文で局所変数を導入することができる。

```
(let ((変数1 式1)
      …
      (変数m 式m))
 式0)
```

この let 文では、式$_1$から式$_m$を評価した結果が、変数$_1$から変数 $_m$に入れられ、最後に式$_0$を評価する。変数$_1$,..., 変数$_m$のスコープは式$_0$である。

ラムダ式（匿名関数）

```
(lambda (変数1 … 変数n) 定義)
```

これは 変数$_1$ ... 変数$_n$を引数とする関数である。例えば、(lambda (x) (* x x)) は 2乗する関数である。((lambda (x) (* x x)) 2) は 4になる。lambdaはギリシャ文字の λ のことである。これらは define を用いて定義した名前付きの関数:

```
(define (square x) (* x x))
```

の square と同じ関数になる。つまり、(define (square x) (* x x)) は (define square (lambda (x) (* x x))) と同じ意味なのである。

S.2 Scheme の call-with-current-continuation

Scheme では、プログラマが接続を直接操作することができる。このことを Scheme は＿＿＿＿＿＿(first-class continuation) を持つという言い方をするときもある。

```
(call-with-current-continuation thunk)
(call/cc thunk)
```

この call-with-current-continuation という名前は長いので、省略形の call/ccがよく使われる (Schemeは、「-」や「/」 のような文字も変数の名前の中でに使用できるので、call-with-current-continuation や call/cc でひとつの名前である。ただし、call/cc は Scheme の標準仕様には含まれていないので、処理形によっては、(define call/cc call-with-current-continuation) のように自分で定義しておく必要がある。)。

ここで *thunk* は 1 引数の関数であり、(call/cc *thunk*) は＿＿＿＿＿＿を引数として、*thunk* を呼び出す。この *thunk* のなかで、この接続を呼び出せば、

そのときの接続は無視されて（= ジャンプして）、`call/cc` が呼ばれたときの接続にその値が返される。また *`thunk`* が接続を呼び出さなければ、*`thunk`* 自身の戻り値が `call/cc` 式全体の戻り値になる。

例えば、

```
(define (bar x)
  (call/cc (lambda (k)
    (+ 100 (if (= x 0) 1 (k x))))))
```

という関数を考える。`(bar 0)` を評価すると普通に足し算が計算され、値は ___ になる。一方、`(bar 1)` の場合は、接続 `k` が 呼び出されるので 100 を足す部分はスキップされて、戻り値は __ となる。

よくある `call/cc` の使い方は、`try` 〜 `catch` と同じような大域脱出である。

```
(define (multlist xs)
   (call/cc (lambda (k)
       (define (aux xs)
           (if (null? xs)     1
               (if (= 0 (car xs))
                   (k 0)
                   (* (car xs) (aux (cdr xs))))))

       (aux xs))))
```

この関数はリストの要素の掛け算を求める。要素の中に 0 が見つかると、大域脱出して `multlist` 全体の値は __ になる。

しかし、このような大域脱出だけならば、言語の仕様に `call/cc` のような大がかりな仕掛けをいれておく必要はない。本当の `call/cc` の価値はコルーチンなどの普通でない制御構造を実現できるところにある。

S.3 コルーチン (coroutine)

コルーチンとは、2 つ以上のプログラムの実行単位が、_____ ながら実行されていく方式のことである。サブルーチン (subroutine) のように、実行単位の間に主と副といった従属関係はなく、コルーチンを構成する個々のルーチンは互いに対等な関係である。

例えば、

```
(define (increase n k)
   (if (> n 10) '()
   (begin (display " i:") (display n)
          (increase (+ n 1) (call/cc k)))))
(define (decrease n k)
   (if (< n 0) '()
   (begin (display " d:") (display n)
          (decrease (- n 1) (call/cc k)))))
```

という 2 つの関数を定義して

```
(increase 0 (lambda (k) (decrease 10 k)))
```

という式を実行すると、

```

```

というように画面へ出力される。2 つの関数 `increase` と `decrease` が交互に実行されていることがわかる。

このように `call/cc` はひじょうに強力なプリミティブで、コルーチンの他にこれまでに紹介したエラー処理（try ～ catch）や非決定性などのプリミティブも、 `call/cc` を用いて定義できることがわかっている。ある意味でオールマイティなプリミティブである。

しかし、`call/cc` は効率的な実装の難しいプリミティブでもある。素直な実装では `call/cc` を実現するためには、スタック全体のコピーを行なう必要がある。一方、はじめからスタックをヒープの中に取り、スタックのコピーを行なわないという方式もある。この方式では不要になったスタック領域もで回収する。

S.4 さらに詳しく知りたい人のために...

文献（R^{6}RS） は Scheme の仕様書である。通常、略して R^{6}RS と呼ばれる。中に `call/cc` の簡単な解説もある。

> **（R^{6}RS）** Richard Kelsey, William Clinger, and Jonathan Rees (Editors), “Revised^6 Report on the Algorithmic Language Scheme” http://www.r6rs.org/

第J章 JavaScript 超簡単入門

JavaScript (ECMAScript) の基本をごくごく簡単に説明する。

J.1 JavaScript の基本

変数

JavaScript には型チェックはないので、変数の宣言に型の情報は必要なく、 ___ というキーワードで変数を宣言する。

```
var i = 0;
```

演算子

次のような演算子 +, –, *, /, %, ++, ––, =, +=, == の意味は C 言語や Java とほぼ同じである。また「+」演算子は文字列の連接にも使用できる。

制御構造

条件判断 （**if** 文）, 繰返し （**while** 文, **for** 文, **do** ～ **while** 文）はほとんど C 言語や Java と同じである。

関数の定義

関数の定義も C 言語と良く似ているが、JavaScript では戻り値の型を書く必要がないので、C 言語で関数の戻り値の型を書く部分に、キーワード _____ を用いるところだけが異なる。また、仮引数の型を宣言する必要もない。return 文の書き方も C 言語と同じである。

```
1 // JavaScript
2 function cube(n) {
3   return n * n * n;
4 }
```

```
1 /* （参考）C 言語 */
2 double cube(double n) {
3   return n * n * n;
4 }
```

もう一つ C 言語と異なるところとして、JavaScript では、関数定義のなかに別の関数を定義することができる。

匿名関数

JavaScript でも無名の関数を定義することができる。JavaScript では次のような形を用いる。

```
function （変数1, …, 変数n） { 宣言・文の並び }
```

つまり、**function** というキーワードと括弧の間に関数名がない。

J.2 ジェネレーター

ECMAScript 6（2015 年）よりジェネレーター (generator) 関数という機構が導入された。ジェネレーター関数はコルーチンの一種 (stackless coroutine) を提供する。

ジェネレーター関数は `function*` というキーワードで定義される。

```
function* gfib(n) {
  var a = 1, b = 1;
  while (a < n) {
    yield a;
    var tmp = b;
    b += a;
    a = tmp;
  }
}
```

ジェネレーター関数の内部では `yield` というキーワードを使って値を生成する。

ジェネレーター関数を呼び出すと、すぐに関数内部のコードが実行されるのではなく、一旦、ジェネレーター (generator) オブジェクトが作られて返される。このジェネレーターオブジェクトの `next` メソッドを呼び出すと、ジェネレーター関数内部のコードが実行され、`yield` された値を `value` プロパティーとして持つオブジェクトを返す。さらに `next` メソッドを呼び出すと yield 式のつづきから実行が再開され、やはり、次の `yield` された値を `value` プロパティーとして持つオブジェクトを返す。

```
var gen = gfib(100);
console.log(gen.next().value);  // 1 を出力する
console.log(gen.next().value);  // 1 を出力する
console.log(gen.next().value);  // 2 を出力する
console.log(gen.next().value);  // 3 を出力する
```

ジェネレーターオブジェクトは `for ～ of` 文の中でも使うことができる。

```
for （変数 of 式） {
    文のならび
}
```

この `for ～ of` 文 はジェネレーターオブジェクト（より一般的には iterable オブジェクト）の `next` メソッドによって返されるオブジェクトの `value` プロパティーを変数に代入してループする。ジェネレーター関数の中のコードの実行が return するか、関数を抜けるとループを終了する。

```
for (v of gfib(10)) {
    console.log(v); // 1, 1, 2, 3, 5, 8 を出力する。
}
```

J.3 HTML タグ

JavaScript は HTML の `<script type="text/javascript">` 〜 `</script>` というタグの間に書く。

```
<script type="text/javascript">
// ここに JavaScript のプログラムを書く。
</script>
```

あるいは、

```
<script type="text/javascript" src="nantoka.js">
</script>
```

で、別ファイル (*nantoka*.js) の JavaScript ソースを読み込める。

J.4 JavaScript プログラムのデバッグ

JavaScript はプログラムの実行前にエラーを見つけてくれないので、気軽に試せる反面、デバッグがしにくい、という短所がある。

問題があったときでも、実行が止まってしまって画面に何も表示されず、そのままでは手がかりが何も得られないことが多い。そういう場合はブラウザーの「コンソール」という画面を表示しておけば、実行時に表示されるエラーメッセージを見ることができる。「コンソール」の表示の仕方はブラウザーによって異なるが、Firefox の場合は、「ツール」－「Web 開発」－「Web コンソール」、Internet Explorer の場合は「ツール」－「F12 開発者ツール」で表示されるようである。

コンソール画面にメッセージを出力するためには `console.log` という関数を使う。また、`alert` という関数を呼び出すと、画面に警告ダイアログを開くことができる。（`alert` の場合、警告ダイアログを閉じるまで、次の処理に進まない。）

```
console.log("Hello!");
```

```
alert("Hello!");
```

これらの関数呼び出しをプログラム中に適宜挿入しておいて実行すると、どこまで実行されたか、どこで実行が止まっているか、などを確認することができる。

J.5 さらに詳しく知りたい人のために ...

文献（ECMA 262） は、JavaScript (ECMAScript) の仕様書である。

> **（ECMA 262）**ECMA International “ECMAScript Language Specification” http://www.ecma-international.org/publications/standards/Ecma-262.htm

第7章 Continuation-Passing Style (CPS)

この章では接続の概念の応用を説明する。

7.1 Continuation-Passing Styleとは

Continuation-Passing Style（CPS）とは常に関数に接続（に相当するもの）を引数として受け渡すプログラムの書き方のことである。次のような使い途がある。

- `call/cc` のない言語でコルーチンなどを実現したいときに用いる
- プログラムを効率の良い形に変換したいときに、変換の途中の中間形式で用いる

また、JavaScript で非同期の関数 (`XMLHttpRequest` の `send` など) を呼び出すときには、CPS に準じてプログラムを書かざるを得ないときもある。（なお ECMAScript 2017 以降では、`async`, `await` を用いて非同期の処理が書き易くなった。）

CPS のプログラムは次のような制限に従う。

- 関数呼び出しが。（つまり、関数呼び出しの引数は関数呼び出し (ただし、「`+`」や「`*`」のような プリミティブな関数の呼び出しは除く。)になっていることがない。）だから、結果が評価順によらない

CPS 変換とは、接続として恒等関数（`\ x -> x`）を渡したときに、意味が同じになるような CPS のプログラムに変換することである。例えば、
ファイル ProdPrimes.js

```
function prodPrimes(n) {
  if (n == 1) return 1;
  else if (isPrime(n)) return n * prodPrimes(n - 1);
  else return prodPrimes(n - 1);
}
```

という関数を考える。これは 1 から n までの範囲に存在する素数の積を求める関数である。ここで `isPrime` は素数かどうかを判定する関数とする。これを、CPS 変換すると、次のような関数 `prodPrimesCPS` に変換される。（ここには定義を示していないが、`isPrime` を CPS 変換した関数を `isPrimeCPS` とする。）
ファイル ProdPrimes.js

```
function prodPrimesCPS(n, c) {
  if (n == 1) return c(1);
```

```
  else return isPrimeCPS(n, function(b) {
    if (b)
       return prodPrimesCPS(n - 1,
                 function (p) { return c (n * p); });
    else return prodPrimesCPS(n - 1, c);
       });
}
```

（JavaScript の記法で紹介しているが、他のプログラミング言語でも同様の変換は可能である。）

```
prodPrime(n) = prodPromeCPS(n, function (x) { return x; })
```

という関係が成り立つ。 ここで `isPrimeCPS` を呼び出すときに、戻ってきたときに行なうべき処理を接続:

```
function (b) {
   if (b)
      return prodPrimesCPS(n - 1, function (p) {
        return c(n * p);
      });
   else return prodPrimesCPS(n - 1, c);
}
```

として `isPrimeCPS` に渡している。さらにこの接続の中で、 `prodPrimesCPS` を呼び出すときに、`b` の値に応じて、`n` を掛けてから `c` に渡すという接続: **function** `(p) { return c(n * p); }`、またはもとのままの接続である `c` を渡している。

CPS はコンパイラーの中間言語として用いられることがある。これは関数の呼び出しの順番が明確になり、関数の呼び出しを単なるジャンプ命令で実現して良いという性質があるからである。

プログラムを CPS に変換するには、だいたい次のような手順で行なう。

1. すべての関数定義に＿＿＿＿＿＿を一つ追加する
   function `prodPrimes(n) { … }` ⇒ **function** `prodPrimesCPS(n, c) { … }`
2. 関数の戻り値に相当する位置にある単純な式は、接続に渡す。（ここで**単純な式**とは …
   定数、変数、ラムダ式、プリミティブオペレーター（「-」, 「==」など）を単純な式に適用した式、のいずれか）
   … **return** `1;` … ⇒ … **return** `c(1);` …
3. 関数の戻り値に相当する位置にある（単純な式でない）関数適用は、接続を引数として渡す。
   … **return** `prodPrimes(n - 1);` … ⇒ … **return** `prodPrimesCPS(n - 1, c)`

4. その他の位置にある（単純な式でない）関数適用は、 "適切な" (("適切な"接続の正確な定義をここで与えることは断念する。 要するに恒等関数（$\lambda x.x$）を接続として渡されたときに元のプログラムと意味が変わらず、CPS 変換を施す目的が達成できれば良い。))接続を明示的に受け取る形に変換する。
   … **return** `n * prodPrimes(n - 1);` …⇒ … **return** `prodPrimesCPS(n - 1, function (p) { return c(n * p); })` …

正式な CPS 変換の定義は、UtilCont に対する変換 `comp` そのものである。つまり、UtilCont を Haskell にコンパイルし、ただし通常は`return` を "`\ a c -> c a`"、`(>>=)` を "`\ c -> m (\ a -> k a c)`" に置き換え、さらに見易いかたちにするための β 簡約を実施すれば CPS 変換になる。 主な部分を `return` と `(>>=)` を置き換えた上で、再構成すると次の表のようになる。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 $\mathring{m}, \mathring{n}$ のように ˚（ドット）が付いている式は、その CPS 変換後の式を表す。

ソース（Util）	ターゲット (CPS)
`x`（ただし`x`は定数・変数）	`\ _c -> _c x`
val `x =` *m* **in** *n*	`\ _c ->` $\mathring{m}$ `(\ x ->` $\mathring{n}$ `_c)`
f a	`\ _c ->` $\mathring{f}$ `(\ _g ->` `  `$\mathring{a}$ `(\ _x ->` `  _g _x _c))`
`\ x ->` *m*	`\ _c -> _c (\ x ->` $\mathring{m}$`)`
if *b* **then** *t* **else** *e*	`\ _c ->` $\mathring{b}$ `(\ _b ->` `  **if** _b **then** `$\mathring{t}$ `_c` `  **else** `$\mathring{e}$ `_c)`
begin *s;* *t;* *u* **end**	`\ _c ->` $\mathring{s}$ `(\ _ ->` `  `$\mathring{t}$ `(\ _ ->` `  `$\mathring{u}$ `_c))`

ターゲット言語は Haskell でなくても、ラムダ式を持っていれば良いので、UtilCont から UtilCont への変換と見なすことも出来る。あるプログラミング言語（例えば JavaScript）が UtilCont と同等の制御構造を持っていれば、JavaScript と UtilCont の間の変換を介して、JavaScript から JavaScript への CPS 変換を考えることが出来る。（関数呼出しがネストしないので、ターゲット言語の評価戦略が遅延評価か先行評価かは関係ない。）

ソース (JavaScript)	ターゲット (JavaScript)
`x`（ただし`x`は定数・変数）	**function** `(_c) { return _c(x); }`
var `x =` *m;* *n*	**function** `(_c) {` `  **return** $\mathring{m}$`(`**function** `(x) {` ` **return** $\mathring{n}$`(_c);` `  });` `}`
f(*a*)	**function** `(_c) {` `  **return** $\mathring{f}$`(`**function** `(_g) {` ` **return** $\mathring{a}$`(`**function** `(_x) {`

	`return _g(_x)(_c);` `});` `});` `}`
`function (x) { m }`	`function (_c) {` `  return _c(function (x) {` `  return m̊;` `  });` `}`
`if (b) { t } else { e }`	`function (_c) {` `  return b̊(function (_b) {` `  if (_b) {` `  return t̊(_c);` `  } else {` `  return e̊(_c);` `  }` `  });` `}`
`s;` `t;` `return u;`	`function (_c) {` `  return s̊(function (_) {` `  return t̊(function (_) {` `  return ů(_c);` `  });` `  });` `}`

7.2 CPS の応用—再帰呼出しの繰返しへの変換

CPS を利用してプログラムの変換を行なうことがある。例として再帰的関数を CPS を経由して繰返しへ変換する場合を取り上げる。

変換の対象は、次のように定義された階乗の関数である。

```
1 function fact(n) {
2   if (n == 0) return 1;
3   else return n * fact(n - 1);
4 }
```

これは数学的な記法の定義:

$$0! = 1$$
$$n! = n \times (n-1)! \quad (n > 0)$$

に直接対応していてわかりやすいが、実行時には n に比例するスタック領域が必要にある。

この fact を CPS に変換すると次のようなプログラムになる。
ファイル Fact.js

```
1 function factCPS(n, c) {
2   if (n == 0) return c(1);
3   else
4     return factCPS(n - 1,
5                    function (r) { return c(n * r); });
6 }
```

さらに、これは末尾再帰なので、次のように繰返しに書き換えることができる。
ファイル Fact.js

```
function aux(n, c) {
  return function(r) {
    return c(n * r);
  };
}

function factCPS(n, c) {
  while (n > 0) {
    c = aux(n, c);  n--; // 注†
  }
  return c(1);
}
```

†ここは `c = function (r) { return c(n * r); };` と書くことはできない。 JavaScript のセマンティクスでは、右辺の変数 `n, c` の値も変わってしまうからである。`aux` 関数を介すると `n, c` の値がコピーされるため安全である。

繰返しに変換されたが、`c` がどんどん大きくなってしまうので、領域の節約にはならない。しかし、良く観察すると `c` は常に次のような形の関数であることがわかる。

```
function (r) { return n * (n - 1) * … * m * r; }
```

つまり、`fact` の場合、第 2 引数として本当の接続を受け渡さなくても、この`n * (n-1) * … * m` で接続を表現可能ということである。このことを考慮に入れてさらにプログラムを変換すると、次の定義が得られる。
ファイル Fact.js

```
function factCPS(n, m) {
  while (n > 0) {
    m *= n; n--;
  }
  return m;
}
```

これは、通常の繰返しによる階乗関数の定義である。このように非末尾再帰を除去する場合、まず CPS に変換して末尾再帰のかたちにし、繰り返しに書き換え、それから“接続”を同等のオブジェクトに置き換えるとよい。

7.3 CPS の応用—Web プログラミング

Servlet や JavaScript などで WWW 上のインタラクティブなアプリケーションを作成するときに、プログラムの任意の場所でユーザーの入力を待って、続きか

ら実行するという書き方ができない（必ず doGet などの関数のはじめから実行されてしまう）という制約がある。

そこで、インタラクティブなプログラムを実現するために、さまざまなテクニックが必要になるが、CPS への変換はある意味でオールマイティーな（つまり、どんな場合にも適用可能な）手段である（もちろん、言語に最初から call/cc が用意されていれば、このような面倒なことをする必要がない。）。

トリッキーな例として JavaScript のハノイの塔のプログラム:
ファイル Hanoi0.js

```
function move(n, a, b) {      // 非 CPS 版
  document.form.textarea.value
    += ("move " + n + " from " + a + " to " + b);
  return 0; // 形をそろえるため 0 を return する
}

function hanoi(n, a, b, c) { // 非 CPS 版
  if (n > 0) {
    hanoi(n - 1, a, c, b);
    move(n, a, b);
    hanoi(n - 1, c, b, a);
  }
  return 0;
}
```

を「ボタンを押したら 1 行表示する」というバージョンに書き換える、ということを考える。つまり、

```
<form name="form">
<input type="button" onClick="exec()" value="実行"><br>
<textarea name="textarea" cols="20" rows="32">
</textarea>
</form>
```

というフォームの「実行」ボタンを押せばテキストエリアに 1 行表示するようにする。

まず、hanoi を CPS に書き換える。
ファイル Hanoi.js

```
function moveCPS(n, a, b, k) {      // 暫定版（説明用）
  document.form.textarea.value
    += ("move " + n + " from " + a + " to " + b +
"\n");
  return k(0);
}

function hanoiCPS(n, a, b, c, k) { // 最終版
  if (n > 0) {
    return hanoiCPS(n - 1, a, c, b, function(ignore) {
      return moveCPS(n, a, b, function(ignore) {
        return hanoiCPS(n - 1, c, b, a, k);
```

```
      });
    });
  } else {
    return k(0);
  }
}
```

しかし、ここで、

```
function exec() {                  // 暫定版（説明用）
  return hanoiCPS(5, 'a', 'b', 'c',
                  function(n) { return n; });
}
```

のように、hanoiCPS を呼び出しても、これまで通り一気に最後まで出力してしまうだけである。そこで moveCPS を次のように書き換える。

```
function moveCPS(n, a, b, k) {      // 最終版
  document.form.textarea.value
    += ("move " + n + " from " + a + " to " + b +
"\n");
  return k;      // k(0) ではない。
}
```

つまり、最後に接続を呼び出してしまわず、いったん呼び出し側に接続を戻り値として返す。（このような手法をトランポリンと言うらしい。）これで call/cc と同じような接続を明示的に扱う効果が得られる。この接続を利用するために exec を次のように書き換える。

```
function doEnd(n) {                 // 最終版
  document.form.textarea.value += "end\n"; // 最後の処理
  return doEnd;
}

// 最初のエントリーポイント
var restart = function(ignore) {
  return hanoiCPS(5, 'a', 'b', 'c', doEnd);
};

function exec() {                   // 最終版
  restart = restart(0);
}
```

この exec は restart(0) の実行結果を新しい restart の値として保存するだけである。これで「実行」ボタンを押すたびに move が 1 回ずつ実行されるようになる。

もう一つの例として、次のフィボナッチ数列を計算する関数を CPS 化してみる。
ファイル Fib0.js

```
function showArgument(m) {      // 非 CPS 版
  document.form.textarea.value += ("argument = " + m);
  return 0;
}

function showResult(m, r) {     // 非 CPS 版
  document.form.textarea.value
    += ("result for argument: " + m + " = " + r);
  return 0;
}

function fib(m) {               // 非 CPS 版
  showArgument(m);
  var r;
  if (m < 2) {
    r = 1;
  } else {
    r = fib(m - 1) + fib(m - 2);
  }
  showResult(m, r);
  return r;
}

function exec() { fib(5); }
```

このプログラムは、計算途中の引数と戻り値を表示するようになっている。まずは、非 CPS 版と意味が変わらない、途中で止まらないバージョンを作成する。ここで `fib` は戻り値を持つので、接続は値を受け取る必要がある。

ファイル Fib.js

```
function showArgumentCPS(m, k) {     // 暫定版
  document.form.textarea.value
    += ("argument = " + m + "\n");
  return k(0);
}

function showResultCPS(m, r, k) {    // 暫定版
  document.form.textarea.value
    += ("result for argument: " + m + " = " + r +
"\n");
  return k(0);
}

function fibCPS(m, k) {              // 最終版
  return showArgumentCPS(m, function(ignore) {
    function tmp(r) {
      return showResultCPS(m, r, function(ingore)   {
        return k(r);
      });
    }
    if (m < 2) {
      return tmp(1);
    } else {
      return fibCPS(m - 1, function(r1) {
        return fibCPS(m - 2, function(r2) {
```

```
          return tmp (r1 + r2);
        });
      });
    }
  });
}

function doEnd(n) {                    // 暫定版
  document.form.textarea.value
    += "final result is " + n + " end\n";
}

function exec() { fibCPS(5, doEnd); }
```

これをハノイの塔のときと同じテクニックを使って途中で止まるように、プログラムを書き換える。

```
function showArgumentCPS(m, k) {     // 最終版
  document.form.textarea.value
    += ("argument = " + m + "\n");
  return k; // k(0) ではない
}

function showResultCPS(m, r, k) {    // 最終版
  document.form.textarea.value
    += ("result for argument: " + m + " = " + r +
"\n");
  return k; // k(0) ではない
}

/* fibCPS は変更なし */

function doEnd(n) {                    // 最終版
  document.form.textarea.value
    += "final result is " + n + " end\n";
  return function(ignore) { return doEnd(n); };
}

var restart = function(ignore) {
  return fibCPS(5, doEnd);
};
function exec() { restart = restart(0); }
```

これで、1 行表示するたびに停止する。

問 7.3.1 上のやり方にならって（CPS を使って）、次の関数を「ボタンを押したら一つの線分を表示する」というバージョンに書き換えよ。
ファイル Sierpinski.js

```
var ctx;
```

```
var x = 256, y = 256, dx = 8, dy = 0, h = 0;

function forward() {
  ctx.strokeStyle = "hsla(" + h + ", 100%, 50%, 0.8)";
  h++;
  ctx.beginPath(); ctx.moveTo(x, y);
  ctx.lineTo(x += dx, y += dy);
  ctx.closePath(); ctx.stroke();
  return 0;
}

function turnLeft() {
  var tmp = dx; dx = dy; dy = -tmp;
  return 0;
}

function turnRight() {
  var tmp = dx; dx = -dy; dy = tmp;
  return 0;
}

function sierpinski(n) {
  zig(n); zig(n);
  return 0;
}

function zig(n) {
  if (n <= 1) {
    turnLeft(); forward(); turnLeft(); forward();
  } else {
    zig(n / 2); zag(n / 2); zig(n / 2); zag(n / 2);
  }
  return 0;
}

function zag(n) {
  if (n <= 1) {
    turnRight(); forward(); turnRight(); forward();
    turnLeft(); forward();
  } else {
    zag(n / 2); zag(n / 2); zig(n / 2); zag(n / 2);
  }
  return 0;
}

function exec() {
  var canvas = document.getElementById('canvas');
  ctx = canvas.getContext("2d");
  sierpinski(16);
}
```

ヒント: 関数 `forward`, `sierpinski`, `zig`, `zag` を CPS に変換する必要がある。関数 `turnLeft`, `turnRight` については、（この問題では） CPS にする必要はない。

問 7.3.2 関数 `sierpinski` をジェネレーター関数を使って、「ボタンを押したら一つの線分を表示する」というバージョンに書き換えよ。

接続の表現

JavaScript は匿名関数（ラムダ式）を持っているため、CPS への変換は比較的容易であったが、ラムダ式を持たない言語や効率を重視する場合では、＿＿＿＿＿＿を明示的に使用し、そのなかに接続に対応するデータを格納する必要がある。次のプログラムは、接続を `n, a, b, c` の各パラメーターと次に実行を開始すべき場所（`pc`）から構成されるデータとしてハノイの塔を表現したものである（このように `while` 文と `switch` ～ `case` 文を用いて、 `goto` 文を模倣する書き方のことは switch-in-a-loop construct というらしい。）。

```
// move は非 CPS 版と同じなので省略

var stack = new Array();
stack.push(new Array(5, 'a', 'b', 'c', 0));

function hanoiStack(n, a, b, c, pc) { // 明示スタック版
  while (n>0) {
    switch (pc) {
    case 0:
      stack.push(new Array(n, a, b, c, 1));
      var tmp = c; c = b; b = tmp; n--;
      continue;
    case 1:
      stack.push(new Array(n - 1, c, b, a, 0));
      move(n, a, b);
      return 0;
    }
  }
  return exec();
}

function exec() {                    // 明示スタック版
  if (stack.length > 0) {
    var args = stack.pop();
    return hanoiStack(args[0], args[1], args[2],
                      args[3], args[4]);
  } else {
    document.form.textarea.value += "end\n";
    return 0;
  }
}
```

ここまでやってしまうとプログラムの実行途中で"接続"をファイルに保存したり、別のコンピューターで起動することさえ可能になる。

第L章 ラムダ計算（λ-calculus）

L.1 ラムダ計算とは？

ラムダ記法とは、関数を簡潔に記述するための記法であり、ラムダ計算とはラムダ記法を用いて、関数の性質について論じるための体系である。ラムダ計算は、単純で、しかも形式的な扱いが可能である。なお、ラムダはギリシャ文字の λ（大文字は Λ）のことであり、関数を表記するのに、この λ という文字を用いることから、この名で呼ばれている。

ここでは型無し（untyped）のラムダ計算を紹介することにする。

ラムダ計算はひじょうに単純な体系であるが、____ や ____ などの計算モデルと同等の計算能力を有することを知られている。つまり、現実のコンピューターと理論的には同等の計算能力を持つ。（メモリーは必要に応じて増やせると仮定している。）

問 L.1.1 チャーチの提唱（Church's thesis）という言葉について、その意味を調べよ。

____

____

____

ラムダ記法・ラムダ計算は理論的な計算機科学のなかで、良く使われる。この章ではラムダ計算のごく基本的な部分を紹介する。

L.2 ラムダ式のかたち（文法）

変数を表す記号が $x, y, z, \ldots$ のように定められている（通常はアルファベット 1 文字）として、次のような”かたち”をしているものをラムダ式と呼ぶ。

1. 変数 — 任意の変数記号はラムダ式である。
2. 関数適用 — M, N をラムダ式とするとき、(MN)もラムダ式である。
   この式の直観的な意味は、____（関数 M を引数 N で呼び出した結果）である（C 言語などでは関数適用は `f(a)` と書くが、 ラムダ計算では、引数の周りの括弧が必要ない代わりに全体に括弧が必要である。）。
3. 関数抽象 — M をラムダ式、x を変数とするとき、$(\lambda x. M)$ もラムダ式である。
   この式の直観的な意味は、____である。

BNF で表現すると以下のとおりである。

$$M ::= V \mid \text{“(”} MM \text{“)”} \mid \text{“(”} \text{“}\lambda\text{”} V \text{“.”} M \text{“)”}$$
$$V \ ::= \text{“}x\text{”} \mid \text{“}y\text{”} \mid \text{“}z\text{”} \mid \cdots$$

この他に 1, 2, 3, ... や $+$, $-$ などの定数を導入する場合もあるが、しばらくは、変数・関数適用・関数抽象の 3 つのみからなる純ラムダ計算を紹介する。

ラムダ式の例

1. $(\lambda x.\, x)$ — これは x という引数を受け取って、x をそのまま返すので、恒等関数を表している。Scheme の記法ならば `(lambda (x) x)` という関数であり、C ならば、

   ```
   int id(int x) { return x; }
   ```

   という関数 `id` に相当する。(λ 式や Scheme では x が `int` 型という制限はないが、C ではこのように型の制限のない関数を定義することはできないので、便宜上 `int` 型にしている。)

2. $(\lambda x.\, (\lambda y.\, x))$ — これは、x という引数を受け取り、$(\lambda y.\, x)$ という関数を返す関数である。そして、$(\lambda y.\, x)$ という関数は、y という引数を受け取り、（これを無視して）x を返す関数である。つまり、式全体は高階関数である。

   λ 記法には、多引数の関数を表す記法はないので、このような"関数を返す関数"で、多引数関数の代用とすることが行われる。つまり、C の記法で、

   ```
   int foo(int x, int y) { return x; }
   ```

   と表される 2 引数の関数を $(\lambda x.\, (\lambda y.\, x))$ と書いてしまうのである。このように、多引数関数を"関数を返す関数"として表現することを、＿＿＿＿と言う。カリー (Curry) は、著名な数理論理学者 Haskell B. Curry (1900–1982) の名にちなんでいる。

3. $(\lambda f.\, (\lambda x.\, (f(fx))))$ — 関数 f とデータ x を受け取って、f を x に 2 回適用する関数である。

L.3 ラムダ計算のきまり（計算規則）

算数で $1 + 2 \times 3 \to 1 + 6 \to 7$ というように計算の規則があるように、ラムダ計算も計算規則（書き換え規則）が決められている。例えば $(\lambda x.\, x)$ は恒等関数であり、任意のラムダ式 M に対して、$((\lambda x.\, x)M) \to M$と書き換えることができる。また、
$(((\lambda f.\, (\lambda x.\, (f(fx))))M)N) \to ((\lambda x.\, (M(Mx)))N) \to (M(MN))$である。

ラムダ計算の変換規則を正式に紹介する前に、いくつかの必要な用語を説明しておく。

束縛変数・自由変数

$(\lambda x.\, M)$ という部分式があるとき、x はこの部分式で束縛され（bound）ているという。このとき、M の中で出現（occur）する変数 x を＿＿＿＿（bound variable）という。束縛変数でない（自由に出現する）変数を＿＿＿＿（free variable）という。（なお、λ のすぐあとに書かれる変数は、そもそも出現にカウントしない。）

例えば、$(\lambda x.(xy))$ の x は束縛変数だが、y は自由変数である。また、$((\lambda z.z)z)$ の z は束縛された形でも、自由にも出現している。一番右端の z は $(\lambda z.\cdots)$ という形の中に入っていないからである。

この最後の例のように、束縛変数と自由変数に同じ名前が使われていると、混乱の元である。そこで、以下の議論では束縛変数は自由変数と名前がぶつからないように、適宜、名前の付け替えをするものと仮定する。

> **Q L.3.1** 以下のラムダ式の変数の出現のうち、どれが自由変数、どれが束縛変数か？
>
> 1. $(\lambda x.(yx))$
> 2. $(a(\lambda b.b))$
> 3. $((\lambda w.w)w)$
> 4. $(\lambda x.(\lambda y.((xy)(zy))))$
>
> ---
>
> ---

一般にプログラミング言語で仮引数の名前は、他の変数とぶつからない限り、付け替えても良い。ラムダ記法でも同様であり、$(\lambda x.(yx))$ と $(\lambda z.(yz))$ は同じものと見なされる。（このように変数の名前を付け替えることを _____(_____) と呼ぶことがある。）ただし、$(\lambda x.(yx))$ と $(\lambda y.(yy))$は別物である。このように名前の衝突する α 変換 は許されない。

> **Q L.3.2** 以下のうち、α 変換によって同等となるラムダ式を選べ。
>
> 1. $(\lambda x.(xy))$ と $(\lambda z.(zy))$
> 2. $(\lambda x.(\lambda y.(xy)))$ と $(\lambda y.(\lambda x.(yx)))$
> 3. $(\lambda x.(\lambda y.y))$ と $(\lambda z.(\lambda y.y))$
> 4. $(\lambda a.(\lambda b.b))$ と $(\lambda b.(\lambda a.b))$

置換

ラムダ式 M, N と変数 x があるとき、$M[x := N]$ という記法を M の中の自由な x の出現をすべて N で置き換えて得られるラムダ式を表すものとする。（この $\_[\_ := \_]$ という記法自体はラムダ式の枠外の、"メタ"な記法である。）

例えば、$(\lambda y.(xy))[x := (\lambda z.z)]$は、$(\lambda y.((\lambda z.z)y))$ となるが、$(\lambda y.(xy))[y := (\lambda z.z)]$ は、$(\lambda y.(xy))$ のままである。y は $(\lambda y.(xy))$ の中に自由に出現していないからである。

β 簡約

ラムダ計算の計算規則は、基本的に β 簡約（β 変換、β reduction）と呼ばれる変換規則のみである。直感的には、関数の仮引数を実引数で置き換える操作である。これは、ラムダ式の中の

$$((\lambda x.M)N)$$

という形をした部分式を

$$M[x := N]$$

に書き換える変換である。この書き換えが適用可能な部分式のことを ____(β redex) と呼ぶ。

例: $(((\lambda f.(\lambda x.(f(fx))))(\lambda y.y))z) \xrightarrow{\beta} ((\lambda x.((\lambda y.y)((\lambda y.y)x)))z)$
$\xrightarrow{\beta} ((\lambda y.y)((\lambda y.y)z)) \xrightarrow{\beta} ((\lambda y.y)z) \xrightarrow{\beta} z$

Q L.3.3 次のラムダ式を（1 ステップ）β 簡約せよ。

1. $((\lambda x.(xy))(\lambda z.(zy)))$
2. $((\lambda x.(\lambda y.x))(\lambda z.z))$
3. $((\lambda y.(xy))(\lambda w.w))$

もっと面白い例として $((\lambda x.(xx))(\lambda x.(xx)))$ というラムダ式は、β 簡約の結果、自分自身に戻る。ラムダ計算には通常のプログラミング言語にあるような繰り返し文や再帰がないが、それでも止まらない計算を表現することができるということがわかる。（あとで、止まる繰り返しも紹介する。）

これ以上 β 簡約を施すことができないラムダ式を_____という。M から β 簡約を繰り返して、N という正規形に到達するとき、N を M の正規形と呼ぶ。上の例でわかるように正規形を持たないラムダ式というものも存在する。

L.4 ラムダ式の略記法

ここまで説明したラムダ式の文法は、計算規則の説明のためには、括弧をつけたり外したりする必要がなくて都合がよい。ただ、このままだと、括弧が多くなりすぎるので、次のような略記法の約束を導入して、括弧の数を節約する。

1. $\lambda x_1 x_2 \cdots x_n.M \equiv (\lambda x_1.(\lambda x_2.(\cdots(\lambda x_n.M)\cdots)))$ つまり、λ 抽象が続く場合は λ を節約して 1 つだけ書く。
2. $M_1 M_2 M_3 \cdots M_n \equiv ((\cdots((M_1 M_2)M_3)\cdots)M_n)$ つまり、関数適用は左に結合する。

また、λ 抽象よりも関数適用の方が優先度が高い。つまり、$\lambda xy.M_1 M_2 M_3$ は $(\lambda x.(\lambda y.((M_1 M_2)M_3)))$ の略記となる。$(\lambda x.M_1)M_2$ は括弧を省略してしまうと、$\lambda x.(M_1 M_2)$ と区別がつかなくなってしまうので、括弧は省略できない。

BNF で表現すると以下のようになる。
M ::= F | “λ” W “.” M
F ::= A | F A
A ::= V | “(” M “)”
W ::= V | V W

V ::= “x” | “y” | “z” | $\cdots$

例:

例えば $\lambda fx.\, f(fx)$ は $(\lambda f.\, (\lambda x.\, (f(fx))))$ の略記であり、$(\lambda x.\, xx)(\lambda x.\, xx)$ は $((\lambda x.\, (xx))(\lambda x.\, (xx)))$ の略記である。

また β 簡約などをするときには、略記法をいちど（頭の中で）正式な記法に戻して、 β 簡約し、再度略記法にする必要がある。

Q L.4.1 次のラムダ記法の正式記法を、できるだけ括弧を少なくした略記法に変換せよ。

1. $(\lambda x.\, (\lambda y.\, ((xy)(xy))))$
2. $(\lambda x.\, (((\lambda y.\, x)(\lambda z.\, z))x))$

Q L.4.2 次のラムダ記法の略記法を正式記法に変換せよ。

1. $\lambda x.\, (\lambda y.\, y)x$
2. $\lambda xy.\, xxy$

L.5 ラムダ計算の性質

よく知られているラムダ計算の性質を証明なしで紹介する。

チャーチ・ロッサー（Church-Rosser）の定理

ひとつのラムダ式に幾通りもの β 簡約が可能なことがある。このとき、異なる β 簡約を行なうと、別の形に枝分かれしてしまう。

しかし、うまく何回か β 簡約するとこの枝分かれしたものを再び合流させることができる、

ということを述べている定理である。

これは同時に、あるラムダ式に正規形が存在するならば、それは一つしかない（α 変換による違いを除く）ということを保証している。

最左戦略

正規形が存在するラムダ式でも、下手に（上手に?）β 基を選んでいけば、いつまでも β 簡約をし続けることがありうる。しかし、最も左からはじまる β 基を選んで行けば、正規形の存在するラムダ式ならば、必ず正規形に到達することが可能である。

例 1: ラムダ式 $(\lambda xy.\, y)((\lambda x.\, xx)(\lambda x.\, xx))$ は、$(\lambda x.\, xx)(\lambda x.\, xx)$ の部分を β 簡約していると、いつまでも正規形に到達しないが、最左 β 基を選ぶとすぐに正規形 $\lambda y.\, y$ になる。

ただし、最左戦略が正規形に到達するために最も効率の良い（つまり β 簡約の少ない）方法とは限らない。（むしろ、そうでないことのほうが多い。）

例 2: ラムダ式 $(\lambda x.\, fxx)((\lambda yz.\, z)w)$ は、右側の β 基を選ぶと $(\lambda x.\, fxx)(\lambda z.\, z)$ になるが、最左 β 基を選ぶと $f((\lambda yz.\, z)w)((\lambda yz.\, z)w)$ になる。最終的にはどちらも $f(\lambda z.\, z)(\lambda z.\, z)$ になる。

> **Q L.5.1** 以下のラムダ式を（1 ステップ）最左簡約せよ。
>
> 1. $((\lambda x.\, ((\lambda y.\, x)x))(\lambda z.\, z))$
> 2. $((\lambda x.\, (xx))((\lambda y.\, y)z))$

L.6 おもしろいラムダ式

いろいろなデータの表現

純粋なラムダ計算には、整数などの組み込みのデータ型がないため、一見したところ意味のある計算ができるようには見えない。しかし、実際には真偽値・整数・組などのデータは純ラムダ計算の中で表現することができる。

真偽値 ラムダ式 $\lambda tf.\, t$, $\lambda tf.\, f$ をそれぞれ $true$, $false$ と呼ぶことにする。また、$\lambda cte.\, cte$ というラムダ式を if と呼ぶことにする。

$if\ true\ M_1\ M_2 \overset{\beta}{\twoheadrightarrow} M_1$ であり、 $if\ false\ M_1\ M_2 \overset{\beta}{\twoheadrightarrow} M_2$ である。

問 L.6.1 上記の β 簡約を 1 ステップずつ書いて確かめよ。つまり、

$$if\ true\ M_1\ M_2 \equiv (\lambda cte.\ cte)\ true\ M_1\ M_2 \to (\lambda te.\ true\ t\ e)\ M_1\ M_2 \to \ldots \to M_1$$

であることを示せ。

チャーチの数（Church numeral） ラムダ式 $\lambda fx.\ x$, $\lambda fx.\ fx$, $\lambda fx.\ f(fx)$, ... を 0, 1, 2, ... という整数に対応するという意味で、c_0, c_1, c_2, ... と呼ぶ。一般に c_n は

$$\lambda fx.\ \underbrace{f(f(\cdots(f}_{n個}x)\cdots))$$

というラムダ式である。ここで $plus$ というラムダ式を次のように定義する。

$$\lambda mnfx.\ mf(nfx)$$

すると、$plus\ c_m\ c_n \overset{\beta}{\twoheadrightarrow} c_{m+n}$ となる。

問 L.6.2 上記の β 簡約を、m = 3, n = 2 などの具体例を用いて 1 ステップずつ書いて確かめよ。つまり、

$$(\lambda mnfx.\ mf(nfx))(\lambda fx.\ f(f(fx)))(\lambda fx.\ f(fx)) \to \cdots \to \lambda fx.\ f(f(f(f(fx))))$$

となることを示せ。

引き算・かけ算などもやや難しくなるが定義することが可能である。

問 L.6.3 次の関数をチャーチの数に対するラムダ式として定義せよ。

1. $zero$ — 0 であるかどうかを判定する述語
2. $mult$ — かけ算
3. $pred$ — 1 を引く関数 （難）
4. sub — 引き算 （$pred$ を使えば簡単）

組 ここで $pair$ を $\lambda fsd.\ dfs$ というラムダ式と定義する。また、fst, snd をそれぞれ、$\lambda p.\ p(\lambda fs.\ f)$, $\lambda p.\ p(\lambda fs.\ s)$ とする。 $fst\ (pair\ M_1\ M_2) \overset{\beta}{\twoheadrightarrow} M_1$ であり、 $snd\ (pair\ M_1\ M_2) \overset{\beta}{\twoheadrightarrow} M_2$ となる。

問 L.6.4 上記の β 簡約を 1 ステップずつ書いて確かめよ。

問 L.6.5 リストを表現するために、$cons$, nil, $isNull$, car, cdr に対応するラムダ式を定義せよ。

Y コンビネーター

ラムダ式 $\lambda f.\ (\lambda x.\ f(xx))(\lambda x.\ f(xx))$ を Y と呼ぶ。
$YF \overset{\beta}{\to} (\lambda x.\ F(xx))(\lambda x.\ F(xx))$ であるが、この右辺を U と置くと、
$U \overset{\beta}{\to} FU \overset{\beta}{\to} F(FU) \overset{\beta}{\to} \cdots \overset{\beta}{\to} F(F(\cdots(FU)\cdots))$ となることがわかる。 U は F の不動点と考えられるため、Y のことを不動点演算子（fixed point operator）とも呼ぶ。このような Y は再帰関数を定義するのに用いることができる。

例えば、$fact$というラムダ式を次のように定義する。

$$Y\ (\lambda fx.\ if\ (zero\ x)\ c_1\ (mult\ x\ (f\ (pred\ x))))$$

これは、おなじみの階乗の関数を定義している。

問 L.6.6 上の $fact$ が階乗の関数を表現していることを、c_3 などの具体的なチャーチ数を用いて、確かめよ。ただし $zero, mult, pred$ などのラムダ式はすでに定義されているものと仮定して良い。（つまり、$pred\, c_3 \xrightarrow{\beta} c_2$ などは途中のステップを書かなくて良い。）

$$
\begin{aligned}
fact\, c_3 &\equiv Y\,(\lambda f x.\, if\,(zero\, x)\, c_1\,(mult\, x\,(f\,(pred\, x))))c_3 \\
&\xrightarrow{\beta} Uc_3 \text{ここで } F \stackrel{\text{def}}{\equiv} \lambda f x.\, if\,(zero\, x)\, c_1\,(mult\, x\,(f\,(pred\, x))) \\
&\qquad\qquad U \stackrel{\text{def}}{\equiv} (\lambda x.\, F(xx))(\lambda x.\, F(xx)) \\
&\xrightarrow{\beta} FUc_3 \\
&\xrightarrow{\beta} if\,(zero\, c_3)\, c_1\,(mult\, c_3\,(U\,(pred\, c_3))) \\
&\xrightarrow{\beta} \cdots
\end{aligned}
$$

L.7 この章のまとめ

以上でラムダ計算が、単純な体系ながら、強力なプログラミング言語とみなすことができるということがわかる。少なくとも再帰と条件分岐などの制御構造、整数などのデータ型をラムダ計算の中で表現することが可能である。また、2 つのラムダ式が等価であるという議論も比較的容易にできる《本当は、 2 つのラムダ式が等価であるという議論が簡単にできるのは、正規形が存在する場合だけである。 正規形が存在しないラムダ式の場合には、互いに β 簡約できないのに、 "同じ"としか考えられないラムダ式が存在する。 これが、D_∞ や Pω などの領域 (domain) に関する理論が必要な理由である。》。

原理的には、より複雑なプログラミング言語の意味をラムダ式として表現することも可能である。しかしながら、実際にすべての計算を純粋なラムダ計算で記述すると、量が多くなりすぎてたいへんである。 Haskell は、基本的にはラムダ計算に、いろいろな便利な構文（構文上の糖衣）と高度な型システムを導入した（だけの）プログラミング言語である。

L.8 さらに詳しく知りたい人のために...

文献 （高橋 1991）は、ラムダ計算について丁寧に解説している。文献 （セシィ 1995）の 12 章にもラムダ計算の解説がある。

（高橋 1991） 高橋 正子 「計算論 — 計算可能性とラムダ計算」近代科学社, 1991年, ISBN4-7649-0184-6

（セシィ 1995）ラビ・セシィ （神林 靖 訳）「プログラミング言語の概念と構造」アジソン・ウェスレイ, 1995 年, ISBN4-7952-9663-4
