■ 連載

Haskell プログラミング

構文解析器結合子

山下 伸夫（(株) タイムインターメディア）
nobsun@sampou.org

文字列の分解

計算機のプログラムの多くはプログラム外からなにがしかの文字列（あるいはバイト列）で表現された入力を受け取る．その中に含まれている情報を取り出し，それを処理し，結果の情報を文字列（あるいはバイト列）で表現して外へ出力する．入力される文字列は，1 つあるいは複数のデータを表現している．これを解釈し情報を取り出すには，何らかの規則によって複数の（1 つの場合もあり得る）塊に分解しなければならない．外から文字列を受け取ったプログラムの最初の仕事は，その文字列を，データ構成の基本単位を表現する塊に分解することである．

単純な分解

最も単純な分解は，たとえば，行単位で分解することである．Haskell ではこの仕事をする関数 `lines` は Prelude で以下のように定義されている．

```
lines :: String -> [String]
lines "" = []
lines s  = let (l,s') = break ('\n'==) s
           in l : case s' of
                    []      -> []
                    (_:s'') -> lines s''
```

白空白で区切られた単語（白空白でない文字のならび）に分解するなら，同じく Prelude で定義されている `words` が使える．

```
words :: String -> [String]
words s  = case dropWhile Char.isSpace s of
             "" -> []
             s' -> w : words s''
                   where (w, s'') = break Char.isSpace s'
```

ここで `break` は，リストを辿って条件に適合する要素が出現する直前までの部分と，その要素以降の部分に分解する．

```
*Main> break ('c'==) "abcde"
("ab","cde")
```

区切り文字を白空白以外の文字にしたい場合には，その区切り文字を判別する述語でパラメータ化した `words2`

を定義しておくと便利である☆1.

```
words2 :: (Char -> Bool) -> String -> [String]
words2 p s  = case dropWhile isSpace s of
               "" -> []
               s' -> w : words2 p s''
                     where
                       (w, s'') = case break p s' of
                                    (v,_:s'') -> (v,s'')
                                    vs        -> vs
```

words2☆2 を用意しておけば，words は，

```
words = words2 isSpace
```

と定義できる．また，CSV (comma separated values) 形式の文字列を分解する関数 csv も以下のように定義できる．

```
csv :: String -> [String]
csv = map trim . words2 (','==)

trim = reverse . dropWhile isSpace . reverse
```

さらに句読点と空白を区切り文字にして分解したいときは，

```
wordsInSentence :: String -> [String]
wordsInSentence = words2 (`elem` " ,.!?")
```

のように定義する．

```
*Main> wordsInSentence "Above all,programming in Haskell is great fun, isn't it?"
["Above","all","programming","in","Haskell","is","great","fun","isn't","it"]
```

複雑な分解

少し複雑な例として，Apache という HTTP サーバのアクセスログを考える．Apache のアクセスログは，クライアントからの要求とそれに対する応答 1 組について 1 行でファイルに書き込まれる．次の 1 行はその例である．

```
10.0.0.10 - - [10/Dec/2005:16:58:46 +0900] "GET /kahua.cgi HTTP/1.1" 200 967
```

この 1 行にはデータの構成要素としての次の 7 つの塊がある☆3.

1. リモートホスト
2. リモートログ名
3. リモートユーザ
4. リクエスト受け付け時刻
5. リクエストライン
6. ステータス
7. レスポンスボディ長

☆1 この記事のプログラムは，http://www.sampou.org/ipsj/ から取ることができる．
☆2 文字述語 isSpace を使うために Char モジュールをインポートしておく必要がある．
☆3 Apache のログのフォーマットについては大抵の場合，httpd.conf という設定ファイルに書かれている．ここで用いているのは common という名前のフォーマットである．

ところがこの 1 行を words で分解すると 10 の塊に分解される．'[' と ']' との間にある日付データが，その中にある空白によって 2 つに分解され，'"' と '"' との間にある HTTP リクエストのリクエストラインデータがその間にある空白により，3 つに分解されたからである．この形式では，'[' と ']' や '"' と '"' は，通常は区切り文字を含むデータ塊の範囲を示すために用いられている．このようなデータを正しく分解するには，'[' と ']' の対や '"' と '"' の対の間では分解が起こらないようにする．

```
accessLog :: String -> [String]
accessLog s = case dropWhile isSpace s of
                ""     -> []
                '[':cs -> case break (']'==) cs of
                            (v,"")    -> error "closing ']' not found"
                            (v,_:cs') -> v : accessLog cs'
                '"':cs -> case break ('"'==) cs of
                            (v,"")    -> error "closing '\"' not found"
                            (v,_:cs') -> v : accessLog cs'
                ccs    -> case break isSpace ccs of
                            (v,s')    -> v : accessLog s'
```

accessLog を使えば，上の 1 行のログデータは 7 つの塊に分解される．

```
*Main> aLogLine
"10.0.0.10 - - [10/Dec/2005:16:58:46 +0900] \"GET /kahua.cgi HTTP/1.1\" 200 967"
*Main> accessLog aLogLine
["10.0.0.10","-","-","10/Dec/2005:16:58:46 +0900","GET /kahua.cgi HTTP/1.1","200","967"]
```

words2 では区切り文字をパラメータ化して定義した．同様に words3 では塊の開始を表す文字と終了を表す文字との組（のリスト）をパラメータ化する．

```
words3 :: [(Char, Char)] -> (Char -> Bool) -> String -> [String]
words3 ps p s
 = case dropWhile isSpace s of
     ""          -> []
     ccs@(c:cs) -> case lookup c ps of
                     Nothing -> case break p ccs of
                                  (v,"")    -> [trim v]
                                  (v,_:cs') -> trim v : words3 ps p cs'
                     Just cl -> case break (cl==) cs of
                                  (v,"")    -> error ("closing '"++cl:"' not foud")
                                  (v,_:cs') -> v : words3 ps p cs'
```

これを使えばログの 1 行を分解する先ほどの accessLog は次のように定義できる．

```
accessLog :: String -> [String]
accessLog = words3 [('[',']'),('"','"')] isSpace
```

字句解析器

文字列データを塊に分解するプログラムは，言語処理系では字句解析器と呼ぶものである．字句解析器は入力文字列を字句 (lexeme) に分解し，トークン (token) 列を生成する．前章の例では塊と呼んでいたものが字句である．

字句解析器を 1 入力文字列をデータ上意味のある最小単位（字句）に分解し，字句の列を作るのが字句解析器である．1 つの字句だけを取り出す小さい解析器をつなぎ合わせてこの字句解析器全体を作る．前章のアクセスログの例

では，空白を含まない文字列，'[' と ']' で囲まれた文字列，'"' と '"' で囲まれた文字列の 3 種類の字句を扱った．これらの種類の字句を識別できる 3 種類の字句解析器をつないで全体を作るのである．まず型を考える．

```
type Lexeme = String
type Lexer  = String -> Lexeme
```

字句解析器は，文字列を取り字句を返す関数として考えるのが単純そうであるが，これでは，字句解析器のつなぎあわせを表現していない．最初の字句解析器が入力文字列の最初のいくらかの部分を消費して字句を取り出し，次の字句解析器が残りの文字列を消費して字句を取り出すように作る．そこで，Lexer を文字列を取り，字句と残りの文字列を返す関数と表現する．

```
type Lexer = String -> (Lexeme, String)
```

これは 8 月号 [3] で状態の受渡し処理を行うために定義した State と同じ構成である．

```
type State s t = s -> (t, s)
type Lexer = State String Lexeme
```

ここでは，9 月号 [1] で示されているように，Haskell のライブラリで提供されている MonadState クラスのインスタンス State を使う．

```
newtype State s a = State { runState :: s -> (a, s)}

instance MonadState (State s) s where
  get   = State $ \ s -> (s,s)
  put s = State $ \ _ -> ((),s)
```

これで前章の accessLog を再構成する．

```
import Data.Char
import Control.Monad.State

type Lexeme  = String
type Lexer l = State String l

runLexer :: Lexer l -> String -> (l, String)
runLexer = runState

lexResult :: Lexer l -> String -> l
lexResult = evalState

lexWord :: Lexer Lexeme
lexWord = State $ break isSpace . dropWhile isSpace

lexSkipChar :: Lexer ()                          -- 入力文字列の先頭の 1 文字を読み捨てる
lexSkipChar = modify $ \ s -> if null s then s else tail s

-- modify は Control.Monad.State で定義されている．
-- modify :: (MonadState s m) => (s -> s) -> m ()
-- modify f = do { s <- get; return (f s) }

lexUntilSep :: Char -> Lexer Lexeme          -- 指定した区切り文字の直前までを読み込む
lexUntilSep sep = State $ break (sep==)
```

```
lexBracketed :: Char -> Char -> Lexer Lexeme -- 指定した文字に挟まれている部分を読み込む
lexBracketed open close
 = do { lexUntilSep open                    -- 字句開始位置手前まで読み飛ばす
      ; lexSkipChar                         -- 字句開始文字を読み捨てる
      ; b <- lexUntilSep close              -- 字句終了位置手前まで読み込む
      ; lexSkipChar                         -- 字句終了文字を読み捨てる
      ; return b                            -- 字句を返す
      }

lex_h, lex_l, lex_u, lex_t, lex_r, lex_s, lex_b :: Lexer Lexeme
lex_h = lexWord                -- リモートホスト
lex_l = lexWord                -- リモートログ名
lex_u = lexWord                -- リモートユーザ
lex_t = lexBracketed '[' ']'   -- リクエスト受付け時刻
lex_r = lexBracketed '"' '"'   -- リクエストライン
lex_s = lexWord                -- ステータス
lex_b = lexWord                -- レスポンスボディ長

accessLog :: String -> [String]
accessLog = lexResult $ sequence [lex_h,lex_l,lex_u,lex_t,lex_r,lex_s,lex_b]
```

構文解析器結合子

1 行あるいは数行が 1 つの記録単位で含まれる字句の順序や数が定まっている場合にはデータの分解は単純な字句解析器で行えば十分であった．しかし，たとえば，四則演算式になると通常字句の数や並び方は単純ではない．四則演算式の生成規則（文法）にしたがって解析する必要がある．

構文解析器の型

字句解析器の連結と同じように部品となる構文解析器の型から考える．構文解析器はトークン（型 `t`）列を取り解析結果（型 `a`）を返すと考えると，

```
[t] -> a
```

となる．前章の議論と同様，入力トークン列（`[t]`）を消費して残りを次に渡したいので，

```
[t] -> (a, [t])
```

とする．さらに構文解析器は入力によっては失敗して欲しい．たとえば，四則演算を表す文字列を構文解析しているとしよう．ある位置では数字か開き括弧があり得る．このとき数字を認識する解析器を適用し，成功すればそのまま続け，失敗すれば開き括弧を認識する解析器をあらためて適用したい．この失敗はプログラマが意図するものなので，解析結果がないという表現が必要である．そこでリストを使う．リストの中身が解析結果で，空リストで解析結果がないことを表現する．

```
[t] -> [(a,[t])]
```

通常は結果があるが，結果がない場合もあるという計算は，検索などの場面でよく現れる．このような計算は `MonadPlus` モナドで抽象化できる．リストがこのモナドのインスタンスであることは `Control.Monad` モジュールで宣言されている．

```
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a

instance MonadPlus [] where
  mzero = []
  mplus = (++)
```

上述の構文解析器の構成はMonadStateクラスのインスタンスStateの構成とMonadPlusクラスのインスタンスであるリストとを合成したものに見える．実際，モジュールControl.Monad.StateではMonadStateと任意のモナドを合成して新たなMonadStateを作る手段StateTが提供されている．これを用いてMonadStateとMonadPlusを合成して，MonadPlusの性質を持つMonadStateを作れることが分かる．

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

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

instance (Monad m) => MonadState s (StateT s m) where
  get   = StateT $ \s -> return (s, s)
  put s = StateT $ \_ -> return ((), s)

instance (MonadPlus m) => MonadPlus (StateT s m) where
  mzero       = StateT $ \_ -> mzero
  m `mplus` n = StateT $ \s -> runStateT m s `mplus` runStateT n s
```

このStateTを使って，構文解析器を構成する．

```
import Data.Char
import Control.Monad
import Control.Monad.State

type Parser t a = StateT [t] [] a

runParser = runStateT
```

基本部品となる構文解析器

failureは常に失敗する構文解析器である．succeedは常に成功し，入力トークン列を消費することなく指定した解析結果とトークン列を返す構文解析器である．itemはトークン列を取り，最初のトークンを無条件で受け付ける．トークン列が空なら失敗する．satは最初のトークンが指定した条件を満たしていればそれを受け付ける．

```
failure :: Parser t a
failure = mzero

succeed :: a -> Parser t a
succeed = return

item :: Parser t t
item = do { (x:xs) <- get
          ; put xs
          ; return x
          }
```

```
sat :: (t -> Bool) -> Parser t t
sat p = do { x <- item
           ; if p x then return x else failure
           }
```

たとえば，hello は最初のトークンが "Hello" であればそれを受理し，さもなければ失敗する．

```
hello :: Parser String String
hello = sat ("Hello"==)
```

実行例は次のとおり．

```
*Main> runParser item "Parser"
('P',"arser")
*Main> runParser item ["Hello","world"]
("Hello",["world"])
*Main> runParser (sat ("Hello"==)) ["Hello","world"]
[("Hello",["world"])]
*Main> runParser (sat ("Hello"==)) ["Good-bye","world"]
[]
```

選択および連接

次に 2 つの構文解析器を合成する構文解析器結合子を考える．2 つの構文解析器を選択（alt）によって合成すると同じトークン列を 2 つの構文解析器に別々にかけ，それぞれの成功した解析結果を全部返す構文解析器ができる．

```
alt :: Parser t a -> Parser t a -> Parser t a
alt = mplus
```

Parser t（すなわち State [t] []）が Monad なので，2 つ（以上）の構文解析器の連接は do 構文を使って書くことができる．

greeeting → *helloOrGoodby item*
helloOrGoodby → Hello
| Goodby

この文法では *greeting* は *helloOrGood* と *item* の連接であり，*helloOrGoodby* は Hello と Goodby の選択である．対応する構文解析器は

```
greeting :: Parser String (String,String)
greeting = do { hg  <- helloOrGoodby
              ; x   <- item
              ; return (hg,x)
              }

helloOrGoodby :: Parser String String
helloOrGoodby =      hello
               `alt` goodbye

hello, goodbye :: Parser String String
hello   = sat ("Hello"  ==)
goodbye = sat ("Goodby" ==)
```

と定義できる.

```
*Main> runParser greeting ["Hello","world"]
[(("Hello","world"),[])]
*Main> runParser greeting ["Goodby","sadness"]
[(("Goodby","sadness"),[])]
*Main> runParser greeting ["Bonjour","tristesse"]
[]
```

繰り返し

指定した構文解析器を 0 回以上繰り返す構文解析器結合子 many と 1 回以上繰り返す many1 を定義する.

```
many :: Parser t a -> Parser t [a]
many p = many1 p `alt` return []

many1 :: Parser t a -> Parser t [a]
many1 p = do { x  <- p
             ; xs <- many p
             ; return (x:xs)
             }
```

例：計算木の再構成

部品と糊がひと通り揃ったので，簡単な構文解析器を作る．題材として 2005 年 5 月号 [2] で扱った計算木を使う．前回は計算木 Term から対応する印字表現（**図 -1**）へ変換する show メソッドを作成し，Term を Show クラスのインスタンスと宣言した．今回は逆に四則演算式を表す文字列から計算木を再構成するメソッド readsPrec を作り，Term を Read クラスのインスタンスとして宣言する☆4.

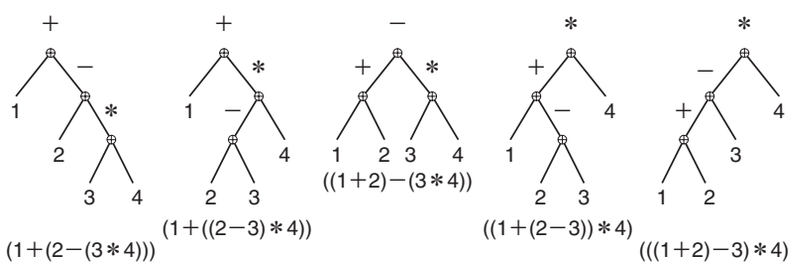

図-1　計算木と対応する印字表現

計算木 Term の定義と Term に対する show メソッドの定義は以下のとおりであった.

```
data Term = Val Char
          | App Char Term Term

instance Show Term where
  show (Val c)     = [c]
  show (App o l r) = "(" ++ show l ++ [o] ++ show r ++ ")"
```

show の定義より Term の印字表現の文法は,

☆4　Show と Read との関係は 2005 年 11 月号 [4] で簡単に述べた.

term → *app* | *var*
app → (*term bop term*)
var → *dig*
bop → [+-*/]
dig → [0123456789]

である．これは直截に構文解析器に翻訳できる．

```
pterm, papp, pval :: Parser Char Term
pterm = papp `alt` pval

papp = do { sat ('('==)
          ; l <- pterm
          ; o <- pbop
          ; r <- pterm
          ; sat (')'==)
          ; return (App o l r)
          }

pval = do { c <- sat (`elem` "0123456789")
          ; return (Val c)
          }

pbop :: Parser Char Char
pbop = do { o <- sat (`elem` "+-*/")
          ; return o
          }
```

readsPrec メソッドは構文解析器 pterm を用いて以下のように定義する．

```
instance Read Term where
  readsPrec _ = runParser pterm
```

実際に評価してみる．

```
*Main> read "(1+2)" :: Term
(1+2)
*Main> read "(1+(2-(3*4)))" :: Term
(1+(2-(3*4)))
*Main> read "((1+(1/9))*9)" :: Term
((1+(1/9))*9)
```

構文解析器がうまく働いている．read と show は逆関数になるように定義したので入力した文字列がそのまま出力されている．read で再構成した Term を評価して確かめることもできる．評価器 eval は 2005 年 5 月号[2] で作成したものを再利用する．

```
*Main> eval $ read "(1+2)"
(3,1)
*Main> eval $ read "(1+(2-(3*4)))"
(-9,1)
*Main> eval $ read "((1+(1/9))*9)"
(90,9)
```

eval は以下のように定義されていた．

```
type Rat = (Int,Int)

eval :: Term -> Rat
eval (Val x)      = ctor x
eval (App o l r) = (ctoo o) (eval l) (eval r)

ctor :: Char -> Rat
ctor x = (ord x - ord '0',1)

ctoo :: Char -> (Rat -> Rat -> Rat)
ctoo '+' (x,y) (z,w) = (x*w+z*y,y*w)
ctoo '-' (x,y) (z,w) = (x*w-z*y,y*w)
ctoo '*' (x,y) (z,w) = (x*z,y*w)
ctoo '/' (x,y) (z,w) = if z == 0 then (0,0) else (x*w,y*z)
```

Maybe 版構文解析器

ここまでの構文解析器では，解析結果がないことを示したいのでリストを使って解析結果を表現した．解析結果がないことを示すためには MonadPlus クラスであればよいはずなので，これを別の MonadPlus クラスのインスタンス Maybe に置き換えてもよさそうである．実際，

```
import Data.Maybe(maybeToList)                  -- maybeToList のインポート

type Parser t a = StateT [t] Maybe a            -- Parser 型を変更

instance Read Term where
  readsPrec _ = maybeToList . runParser pterm  -- readsPrec の実装の変更
```

の 3 個所の変更で先の Maybe 版の計算木構文解析器を試せる．

リスト版構文解析器と Maybe 版構文解析器の違いは選択結合子 (alt) の挙動にある．すでに見たように alt は StateT t m に対する MonadPlus クラスのメソッド mplus と同じである．

```
m `mplus` n = StateT $ \s -> runStateT m s `mplus` runStateT n s
```

右辺の mplus は StateT [t] [] a の場合はリストに対する mplus であり，StateT [t] Maybe a の場合には Maybe に対応する mplus である．それぞれの mplus は標準ライブラリ Monad で定義されている．

```
instance  MonadPlus []  where
  mzero = []
  mplus = (++)

instance MonadPlus Maybe where
  mzero              = Nothing
  Nothing `mplus` ys = ys
  xs      `mplus` ys = xs
```

リスト版の構文解析器結合子 alt で作られた構文解析器は与えられた 2 つの構文解析器の解析結果をすべて連結して返す．一方，Maybe 版の alt で作られた構文解析器は 1 つ目の構文解析器が成功すれば 2 つ目の構文解析の結果は採用しない．これにより扱える文法に差が出る．たとえば，

$S \rightarrow$ aBd

$B \rightarrow$ b | bc

という文法を考える．これに対応する構文解析器を以下のように定義する．

```
char :: Char -> Parser Char Char
char c = sat (c==)

pS = do { a   <- pa
        ; bc  <- pB
        ; d   <- pd
        ; return ([a]++bc++[d])
        }

pB =       (do { b <- pb; return [b] })
     `alt` (do { b <- pb; c <- pc; return ([b]++[c]) })

pa = char 'a'
pb = char 'b'
pc = char 'c'
pd = char 'd'
```

リスト版では

```
*Main> runParser pS "abcd"
[("abcd","")]
```

と成功するが，Maybe 版では

```
*Main> runParser pS "abcd"
Nothing
```

のように失敗する．これは，Maybe 版では後戻りの必要な構文解析ができないからである．

この 2 つの解析器はともに再帰下降解析を行う．このような解析器では左再帰を含むような文法に沿っての解析はできない．また上のような文法では後戻り解析が必要になる．通常，再帰下降解析を行えるような言語は LL(1) の文法クラスで設計される．LL(1) 文法には左再帰は含まれず，また後戻り解析も必要にならない．そのように注意深く設計されている言語に対しては Maybe 版の構文解析器が利用できる．

今回取り上げた構文解析器はシンプルなもので，エラー処理，効率についてはなにも対応していない．実際の応用場面では，再帰下降解析器としては GHC や Hugs で利用できる Parsec[5] というライブラリがある．

参考文献

1) 尾上能之 : 文字列間の距離―モナドを使って―，情報処理, Vol.46, No.9, pp.1053-1060 (2005). (http://www.ipsj.or.jp/07editj/promenade/4609.pdf)
2) 山下伸夫 : 木(tree)で遊ぶ, 情報処理, Vol.46, No.5, pp.564-570 (2005). (http://www.ipsj.or.jp/07editj/promenade/4605.pdf)
3) 山下伸夫 : 記憶(memo)する関数, 情報処理, Vol.46, No.8, pp.930-938 (2005). (http://www.ipsj.or.jp/07editj/promenade/4608.pdf)
4) 山下伸夫 : ペンシルパズルを解く, 情報処理, Vol.46, No.11, pp.1279-1288 (2005). (http://www.ipsj.or.jp/07editj/promenade/4611.pdf)
5) Leijen, D. : http://www.cs.uu.nl/~daan/parsec.html

（平成 17 年 12 月 22 日受付）