第4章　パーサコンビネータ

4.1　後戻りつきの再帰下降構文解析

パーサ（構文解析器）については、後戻りつきの再帰下降構文解析系として非決定性と状態のモナドを用いて定義することが可能である。この節ではパーサのモナドのアイデアを紹介する。

例として、最も単純な言語 Util0 用のパーサ部分を紹介する。Util0 は定数、足し算、掛け算のみを構成要素として持つ言語である。Util0 の具象構文は以下の通りである。

$$
\begin{aligned}
\mathit{Expr} &\rightarrow \mathit{Term\ Expr1} \\
\mathit{Expr1} &\rightarrow \mathit{+\ Term\ Expr1} \mid \varepsilon \\
\mathit{Term} &\rightarrow \mathit{Factor\ Term1} \\
\mathit{Term1} &\rightarrow \mathit{*\ Factor\ Term1} \mid \varepsilon \\
\mathit{Factor} &\rightarrow \mathit{Const} \mid (\ \mathit{Expr}\ )
\end{aligned}
$$

また、Util0 の構文木のデータ型は次のように定義する。

```
data Expr  = Add Expr Expr | Mult Expr Expr | Const Value
data Value = Num Integer
```

この節で紹介するパーサは別に用意されたスキャナ (字句解析器) と組み合わせて使うと仮定する。つまり、次のように字句を表すデータ型があり、

```
data Token = Single Char | Symbol String | Number Integer
```

また、次のような字句解析関数が用意されていると仮定する。

```
lexer :: String -> [Token]
```

例えば `lexer "1*(2+3)"` は `[Number 1, Symbol "*", Single '(', Number 2, Symbol "+", Number 3, Single ')']` という字句リストになる。(この字句解析関数も単純な状態変換 (state transformer) のモナドを使って定義できるはずである。)

問 4.1.1 *Util0* の字句解析器 *lexer* を定義せよ。

まず、パーサのモナドの基本演算を導入する。

```
type MyParser d a =  d -> [(a, d)]

unitP :: a -> MyParser d a
unitP a = \ str -> [(a, str)]

bindP :: MyParser d a -> (a -> MyParser d b) -> MyParser d b
m `bindP` k = \ str -> concat (map (uncurry k) (m str))
```

次にパーサを構築する上で基本的な演算を導入する。

```
mplusP :: MyParser d a -> MyParser d a -> MyParser d a   -- 非決定的選択
m1 `mplusP` m2 = \ str -> m1 str ++ m2 str

choiceP :: MyParser d a -> MyParser d a -> MyParser d a  -- バイアスつき選択
m1 `choiceP` m2 = \ str -> case m1 str of
                             [] -> m2 str
                             ds -> ds

infixr 1 `choiceP`

mzeroP :: MyParser d a    -- 失敗
mzeroP = \ str -> []

check :: (a -> Bool) -> MyParser [a] a  -- Tokenが条件を満たすかどうか
check p (h:t) = if p h then [(h, t)] else []
check p []    = []

many :: MyParser d a -> MyParser d [a]    -- 0回以上の繰返し
many m  = let q = (m `bindP` \ x -> q `bindP` \ xs -> unitP (x:xs))
                  `choiceP` unitP []     in q

many1 :: MyParser d a -> MyParser d [a]   -- 1回以上の繰返し
many1 m = m `bindP` \ x -> many m `bindP` \ xs -> unitP (x:xs)
```

これ以降はUtil0の構文に固有の構文解析部である。

```
parseExpr   = parseTerm  `bindP` \ t ->
              parseExpr1 `bindP` \ ts ->
              unitP (foldl1 Add (t:ts))

parseExpr1  = (check (== Symbol "+") `bindP` \ _ ->
               parseTerm             `bindP` \ t ->
               parseExpr1            `bindP` \ ts ->
               unitP (t:ts))
              `choiceP`
              unitP []

parseTerm   = parseFactor `bindP` \ f ->
              parseTerm1  `bindP` \ fs ->
              unitP (foldl1 Mult (f:fs))

parseTerm1  = (check (== Symbol "*") `bindP` \ _ ->
               parseFactor           `bindP` \ f ->
               parseTerm1            `bindP` \ fs ->
               unitP (f:fs))
              `choiceP`
              unitP []

parseFactor = (check (\ tok -> case tok of Number _ -> True ; _ -> False)
                                                    `bindP` \ (Number n) ->
               unitP (Const (Num n)))
              `choiceP`
              (check (== Single '(') `bindP` \ _ ->
               parseExpr `bindP` \ e ->
               check (== Single ')') `bindP` \ _ ->
               unitP e)
```

BNFから単純明快にパーサが構築できることがわかるだろう。ただし、構文エラーがあった時のエラーメッセージは考慮されていないので、このままでは実用的なパーサを書くことはできない。

4.2 パーサコンビネータライブラリ Parsec

Parsec はモナドに基づくパーサコンビネータライブラリである。基本的な考え方は前節で紹介したパーサモナド MyParser と大きくは違わないが、Parsec は実行効率が高く、また、品質の良いエラーメッセージを出力することができる、この節では Parsec の使用方法を説明する。

Parsec を使用するためには、次のような import 文を書く必要がある。

```
import Text.ParserCombinators.Parsec
```

パーサの型は Parser という名前のモナドである。これは次のように型の別名として定義されている。

```
type Parser a = GenParser Char () a
```

ここで、GenParser tok st a は、型 tok のリストを入力として受け取り、ユーザが設定する状態の型 st を持つモナドである。

4.2.1 Text.ParserCombinators.Parsec.Prim

このモナドに関連する基本的な関数は、モナドの基本オペレータの return, (>>=) に加えて、runParser, (<|>), try, token, tokenPrim, (<?>) などである。これらの基本関数は Text.ParserCombinators.Parsec.Prim モジュールに定義されている。(Text.ParserCombinators.Parsec を import すると、同時に import される。)

これらの関数については、Parsec のドキュメント「Parsec, a fast combinator parser」から説明を抜粋する。

runParser
:: GenParser tok st a -> st -> FilePath -> [tok] -> Either ParseError a

The most general way to run a parser. (runParser p state filePath input) runs parser p on the input list of tokens input, obtained from source filePath with the initial user state st. The filePath is only used in error messages and may be the empty string. Returns either a ParseError (Left) or a value of type a (Right).

runParser は初期値を設定して、パーサを実行する関数である。

(<|>) :: GenParser tok st a -> GenParser tok st a -> GenParser tok st a
infixr 1 <|>

This combinator implements choice. The parser (p <|> q) first applies p. If it succeeds, the value of p is returned. If p fails *without consuming any input*, parser q is tried. This combinator is defined equal to the mplus member of the MonadPlus class.

The parser is called predictive since q is only tried when parser p didn't consume any input (i.e.. the look ahead is 1). This non-backtracking behaviour allows for both an efficient implementation of the parser combinators and the generation of good error messages.

```
try  :: GenParser tok st a -> GenParser tok st a
```

The parser (try p) behaves like parser p, except that it pretends that it hasn't consumed any input when an error occurs.

This combinator is used whenever arbitrary look ahead is needed. Since it pretends that it hasn't consumed any input when p fails, the (<|>) combinator will try its second alternative even when the first parser failed while consuming input.

"選択" を表すオペレータは(<|>) である。しかし、(<|>) の左辺のオペレータは、失敗の時には入力を消費してはいけない。入力を消費してから失敗する可能性がある時は、try オペレータで全体を囲む必要がある。

```
token  :: (tok -> String) -> (tok -> SourcePos) -> (tok -> Maybe a)
          -> GenParser tok st a
```

The parser (token showTok posFromTok testTok) accepts a token t with result x when the function testTok t returns Just x. The source position of the t should be returned by posFromTok t and the token can be shown using showTok t.

```
tokenPrim  :: (tok -> String) -> (SourcePos -> tok -> [tok] -> SourcePos)
              -> (tok -> Maybe a) -> GenParser tok st a
```

The parser (token showTok nextPos testTok) accepts a token t with result x when the function testTok t returns Just x. The token can be shown using showTok t. The position of the next token should be returned when nextPos is called with the current source position pos, the current token t and the rest of the tokens toks, (nextPos pos t toks).

token と tokenPrim は、前節の MyParser の check に類似の基本関数である。第 1 引数と第 2 引数はエラーメッセージに関する関数である。

```
(<?>)  :: GenParser tok st a -> String -> GenParser tok st a
infix 0 <?>
```

The parser (p <?> msg) behaves as parser p, but whenever the parser p fails without consuming any input, it replaces expect error messages with the expect error message msg.

This is normally used at the end of a set alternatives where we want to return an error message in terms of a higher level construct rather than returning all possible characters.

(<?>) も、やはりエラーメッセージに関するオペレータである。

4.2.2 Text.ParserCombinators.Parsec.Combinator

Text.ParserCombinators.Parsec.Combinator モジュールは基本オペレータを組み合わせて、より高度なパーサを作成するための演算が用意されている。このモジュールも Text.ParserCombinators.

Parsec を import すると、同時に import される。多くの便利な演算が用意されているが、ここでは代表的な演算のみ紹介する。

many `:: GenParser tok st a -> GenParser tok st [a]`

`(many p)` applies the parser `p` *zero* or more times. Returns a list of the returned values of `p`.

many1 `:: GenParser tok st a -> GenParser tok st [a]`

`(many p)` applies the parser `p` *one* or more times. Returns a list of the returned values of `p`.

sepBy `:: GenParser tok st a -> GenParser tok st sep -> GenParser tok st [a]`

`(sepBy p sep)` parses *zero* or more occurrences of `p`, separated by `sep`. Returns a list of values returned by `p`.

eof `:: Show tok => GenParser tok st ()`

This parser only succeeds at the end of the input.

4.2.3 Text.ParserCombinators.Parsec.Char

このモジュールは

```
type CharParser st a = GenParser Char st a
```

つまり文字ストリーム（String）を対象とするパーサ用の関数群である。このモジュールも Text.ParserCombinators.Parsec を import すると、同時に import される。やはり代表的な関数だけ紹介する。

char `:: Char -> CharParser st Char`

`(char c)` parses a single character `c`. Returns the parsed character (i.e. `c`).

```
semiColon  = char ';'
```

string `:: String -> CharParser st String`

`(string s)` parses a sequence of characters given by `s`. Returns the parsed string (i.e. `s`).

```
divOrMod    =   string "div"
            <|> string "mod"
```

satisfy :: (Char -> Bool) -> CharParser st Char

The parser (satisfy f) succeeds for any character for which the supplied function f returns True. Returns the character that is actually parsed.

```
digit     = satisfy isDigit
oneOf cs  = satisfy (\ c -> c `elem` cs)
```

4.2.4 Text.ParserCombinators.Parsec.Expr

四則演算など、本来は再帰下降構文解析法では記述するのが面倒な部分に対して、演算子順位法と同じように演算子の優先順位と結合性を指定することでパーサを作成するためのモジュールである。このモジュールは Text.ParserCombinators.Parsec と別に import する必要がある。

少し長くなるがドキュメントから引用する。

Assoc

This data type specifies the associativity of operators: left, right or none.

```
data Assoc  = AssocNone | AssocLeft | AssocRight
```

Operator tok st a

This data type specifies operators that work on values of type a. An operator is either binary infix or unary prefix or postfix. A binary operator has also an associated associativity.

```
data Operator tok st a
  = Infix (GenParser tok st (a -> a -> a)) Assoc
  | Prefix (GenParser tok st (a -> a))
  | Postfix (GenParser tok st (a -> a))
```

OperatorTable tok st a

An (OperatorTable tok st a) is a list of (Operator tok st a) lists. The list is ordered in descending precedence. All operators in one list have the same precedence (but may have a different associativity).

```
type OperatorTable tok st a  = [[Operator tok st a]]
```

buildExpressionParser :: OperatorTable tok st a -> GenParser tok st a
-> GenParser tok st a

(buildExpressionParser table term) builds an expression parser for terms term with operators from table, taking the associativity and precedence specified in table into account. Prefix and postfix operators of the same precedence can only occur once (i.e. --2 is not allowed if - is prefix negate). Prefix and postfix operators of the same precedence associate to the left (i.e. if ++ is postfix increment, than -2++ equals -1, not -3).

4.2.5 Text.ParserCombinators.Parsec.Token

Yacc や Bison などで構文解析器を作成する時は、字句解析器を別に Lex などで作成し、構文解析器は字句（トークン）の列を受け取るように設計するのが一般的である。しかし、Parsec では字句解析器と構文解析器をわけず、直接 String を入力とするパーサを作成することが多い。この場合、識別子、コメント、数値リテラルなどの字句を認識するための関数が必要となる。Text.ParserCombinators.Parsec.Token は、そのような字句パーサを作成するためのモジュールである。このモジュールは Text.ParserCombinators.Parsec と別に import する必要がある。

このモジュールはただ一つの関数 makeTokenParser から構成される。makeTokenParser 関数は LanguageDef st型の引数を受け取って、TokenParser st型を返す。

TokenParser st

The type of the record that holds lexical parsers that work on character streams with state st.

```
data TokenParser st
 = TokenParser{
     identifier  :: CharParser st String
   , reserved    :: String -> CharParser st ()
   , operator    :: CharParser st String
   , ...  -- 省略
   }
```

LanguageDef st

The LanguageDef type is a record that contains all parameterizable features of the Text.ParserCombinators.Parsec.Token module.

```
data LanguageDef st
  = LanguageDef
  { commentStart   :: String
  , commentEnd     :: String
  , ...  -- 省略
  , reservedNames  :: [String]
  , reservedOpNames:: [String]
  , caseSensitive  :: Bool
  }
```

makeTokenParser :: LanguageDef st -> TokenParser st

The expression (makeTokenParser language) creates a TokenParser record that contains lexical parsers that are defined using the definitions in the language record.

The use of this function is quite stylized – one imports the appropiate language definition and selects the lexical parsers that are needed from the resulting TokenParser.

```
module Main where

import Text.ParserCombinators.Parsec
import qualified Text.ParserCombinators.Parsec.Token as P
import Text.ParserCombinators.Parsec.Language (haskellDef)

-- The parser
 ...

expr  =   parens expr
      <|> identifier
      <|> ...


-- The lexer
lexer       = P.makeTokenParser haskellDef

parens      = P.parens lexer
braces      = P.braces lexer
identifier  = P.identifier lexer
reserved    = P.reserved lexer
```

4.2.6 TokenParser のメンバ

TokenParser レコード型のメンバを通常の関数のように紹介する。(やはり主なもののみ抜粋する。)

whiteSpace :: CharParser st ()

Parses any white space. White space consists of zero or more occurrences of a space, a line comment or a block (multi line) comment. Block comments may be nested. How comments are started and ended is defined in the LanguageDef that is passed to makeTokenParser.

lexeme :: CharParser st a -> CharParser st a

(lexeme p) first applies parser p and than the whiteSpace parser, returning the value of p. Every lexical token (lexeme) is defined using lexeme, this way every parse starts at a point without white space. Parsers that use lexeme are called *lexeme* parsers in this document.

The only point where the whiteSpace parser should be called explicitly is the start of the main parser in order to skip any leading white space.

```
mainParser  = do{ whiteSpace
                ; ds <- many (lexeme digit)
                ; eof
                ; return (sum ds)
                }
```

symbol :: String -> CharParser st String

Lexeme parser (symbol s) parses (string s) and skips trailing white space.

parens :: CharParser st a -> CharParser st a

Lexeme parser (parens p) parses p enclosed in parenthesis, returning the value of p. It can be defined as:

```
parens p  = between (symbol "(") (symbol ")") p
```

identifier :: CharParser st String

This lexeme parser parses a legal identifier. Returns the identifier string. This parser will fail on identifiers that are reserved words. Legal identifier (start) characters and reserved words are defined in the LanguageDef that is passed to makeTokenParser. An identifier is treated as a single token using try.

reserved :: String -> CharParser st String

The lexeme parser (reserved name) parses (symbol name), but it also checks that the name is not a prefix of a valid identifier. A reserved word is treated as a single token using try.

問 4.2.1 *Tiny C (http://www.watalab.cs.uec.ac.jp/tinyCdoc.txt)* は *C* 言語のサブセットとして定義されたプログラミング言語である。次に示す *Tiny C* の構文（*BNF*）を適切な代数的データ型として定義し、*Tiny C* のパーサーを完成させよ。

（*BNF* は改ページの都合で適宜分割して掲載する。）

```
Program      →  ExternalDeclaration | ExternalDeclaration Program
ExternalDeclaration → FunctionDef | FunctionDecl | VariableDecl
FunctionDef         → FunctionDeclarator Block
FunctionDecl        → FunctionDeclarator ";"
FunctionDeclarator  → TypeName FuncName "(" ParamList ")"
ParamList    → TypeName NameSpec ParamListTail | ε
ParamListTail→ "," TypeName NameSpec ParamListTail | ε
NameSpec     → Id | Id "[" IntConstant "]"
VariableDecl → TypeName NameSpec NameListTail ";"
NameListTail → "," NameSpec NameListTail | ε
```

```
Block        → "" StatementSeq ""
StatementSeq → Statement | Statement StatementSeq | ε
Statement    →
    Variable "=" Expression ";"
  | "if" "(" ConditionExp ")" Statement ElsePart
  | "while" "(" ConditionExp ")" Statement
  | FuncName "(" ExpressionList ")" ";"
  | Block   | "return" ";"
ElsePart     → "else" Statement | ε
ExpressionList → Expression ExpListTail | ε
ExpListTail  → "," Expression ExpListTail | ε
```

```
Expression   → SignPart Term ExpTail
ExpTail      → AddSub Term ExpTail | ε
Term         → Factor TermTail
TermTail     → MultDiv Factor TermTail | ε
Factor       → IntConstant | Variable | "(" Expression ")"
Variable     → Id SubscriptPart
SubscriptPart→ "[" Expression "]" | ε
ConditionExp → Expression CompOperator Expression
```

```
CompOperator → "==" | "!=" | ">" | ">=" | "<=" | "<"
AddSub       → "+" | "-"
MultDiv      → "*" | "/"
SignPart     → "+" | "-" | ε
FuncName     → Id
TypeName     → "int" | "void"
Id           → Alpha IdTail
IntConstant  → Digit ConstTail
IdTail       → Alpha IdTail | Digit IdTail | ε
Alpha →  "a"|"b"|"c"|"d"|"e"|"f"|"g"|"h"|"i"|"j"|"k"|"l"|"m"|
         "n"|"o"|"p"|"q"|"r"|"s"|"t"|"u"|"v"|"w"|"x"|"y"|"z"|
         "A"|"B"|"C"|"D"|"E"|"F"|"G"|"H"|"I"|"J"|"K"|"L"|"M"|
         "N"|"O"|"P"|"Q"|"R"|"S"|"T"|"U"|"V"|"W"|"X"|"Y"|"Z"
Digit →  "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
ConstTail    →  Digit ConstTail | ε
```

この章の参考文献

[1] Philip Wadler 「Monads for functional programming」
Program Design Calculi, Proceedings of the Marktoberdorf Summer School, 1992 年 7–8 月
モナドを用いてパーサを構築する技法の解説がある。

[2] Daan Leijen 「Parsec, a fast combinator parser」
`http://legacy.cs.uu.nl/daan/download/parsec/parsec.html`, 2001 年 4 月

[3] Daan Leijen and Erik Meijer 「Parsec: Direct Style Monadic Parser Combinators for the Real World」
Thechnical Report UU-CS-2001-35, Department of Computer Science, Universiteit Utrech, 2001 年