文脈自由文法とパーサコンビネータ

Pasberth

December 20, 2013

概 要

この文書は文脈自由文法とパーサコンビネータの簡単な教科書である．これを読めば，きっと，文脈自由文法とパーサコンビネータを理解する手助けとなるであろう．本書は主に Haskell におけるパーサコンビネータを扱う．たとえば，Parsec や Attoparsec，Trifecta の使い方を簡単に勉強するのが本書の目標である．もしあなたがパーサコンビネータに興味があるなら，ぜひ手にとってみてほしい．きっと役に立つだろう．

目 次

1　はじめに

1.1　ソースコード

この文書はオープンソースである．完全なソースコードは，2013 年 12 月 20 日現在，`https://github.com/pasberth/pasberth.github.io/tree/default/papers/brief-intro-parser-combinators` で公開されている．この文書自体のソースコードや，例に使用されるソースコードがそこから入手できる．

1.2　ソースコードの入手方法

もっとも簡単な方法は git を使用することである．次のようなシェルコマンドで簡単に入手できる．

```
git clone https://github.com/pasberth/pasberth.github.io.git
cd pasberth.github.io
git checkout default
cd papers/brief-intro-parser-combinators
```

2　文脈自由文法

言語の規定のために用いる概念である文脈自由文法 (*context-free grammer*) を導入する．文脈自由文法は略して文法 (*grammer*) ということもある．

文法は，たいていの形式言語の構成要素がもつ階層構造を，自然な形で表現したものである．たとえば，Java の if-else 文は，

if (式) 文 **else** 文

である．すなわち，if-else 文は，キーワード **if**，左かっこ，式，右かっこ，文，キーワード **else**，そして 1 つの文を並べたものである．式をあらわすための変数を *expr*，文をあらわすための変数を *stmt* とすると，この構造化規則は，次のように表現できる．

$$stmt \longrightarrow \textbf{if} \ (\ expr\ )\ stmt\ \textbf{else}\ stmt$$

このような規則を**生成規則**とよぶ．キーワード **if** や “(” のような字句表現を**終端記号**とよぶ．*expr* や *stmt* のような変数を**非終端記号**とよぶ．

2.1 文法の定義

文法は次の 4 つの要素からなる．

- 終端記号 (*terminal symbol*) の集合．
- 非終端記号 (*nonterminal symbol*) の集合．
- 生成規則 (*production*) の集合．
- 開始記号としての，1 つの非終端記号の指定．

文法は，生成規則を列挙することによってさだめられる．その際，開始記号に対する生成規則を先頭におくようにする．**if** や **else** のように，太字であらわされた記号は終端記号である．“(”といった符号は終端記号である．*expr* や *stmt* のようにイタリック体であらわされた記号は非終端記号である．生成規則を，

$$\alpha \longrightarrow \beta_1\beta_2 \ldots \beta_n$$

のように書き，“α は $\beta_1\beta_2 \ldots \beta_n$ という形式をもつことができる”と読む．α はその生成規則の**頭部**または**左辺**とよばれる．$\beta_1\beta_2 \ldots \beta_n$ はその生成規則の**本体**または**右辺**とよばれる．

文脈自由文法で S 式を定義する簡単な例を述べる．まず，自然言語で S 式を定義する．

1. 加算個の変数 $x_1, x_2, \ldots, x_n$ は S 式である．
2. M と N が S 式ならば，$(M.N)$ は S 式である．

この定義から，たとえば，$(x_1.x_2)$ は S 式であるとわかる．これを文脈自由文法を用いて記述する．

1. $\mathit{variable} \longrightarrow x_1$
   $\mathit{variable} \longrightarrow x_2$
   :
   $\mathit{variable} \longrightarrow x_n$
2. *s-expression* $\longrightarrow$ *variable*
3. *s-expression* $\longrightarrow$ (*s-expression* . *s-expression*)

2.2 導出

文法は，開始記号から出発して，非終端記号をそれに対する生成規則の本体でおきかえる，という操作の繰り返しによって，記号列を導出する．開始

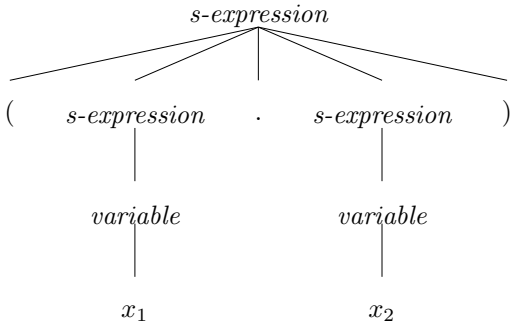
記号から導出できることのできる終端記号列が，その文法によって定義される言語を形成する．

たとえば，$(x_1.x_2)$ が *s-expression* であることは，次のようにして説明できる．

1. 生成規則 (1) から x_1 は *variable* なので，生成規則 (2) から x_1 は *s-expression* である．
2. 生成規則 (1) から x_2 は *variable* なので，生成規則 (2) から x_2 は *s-expression* である．
3. x_1 と x_2 は *s-expression* なので，生成規則 (3) から $(x_1.x_2)$ は *s-expression* である．

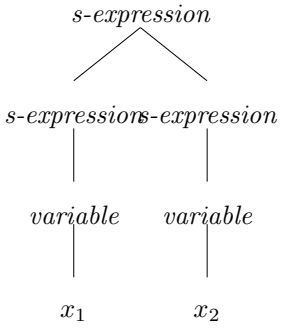
2.3 解析木

解析木 (*parse tree*) は，言語に含まれる文字列が，その文法の開始記号から，どのように導出できるかを図式したものである．解析木のことを，構文木 (*syntax tree*) ということもある．解析木を形式的に定義することもできるが，ここでは例を見て理解してほしい．次の図は，$(x_1.x_2)$ の解析木である．



2.4 抽象構文木

抽象構文木 (*abstract syntax tree*) は，解析木からよけいな文字を取り除いたものである．たとえば，$(x_1.x_2)$ の解析木に “(”や “.”や “)” といった文字が含まれるが，情報として必要とは思えない．なぜならば，これがあったとしてもなかったとしても，それが定義 (3) のものだとはっきりわかるからである．しかし，x_1 や x_2 は情報として必要である．“(”や “.”や “)” といった文字を取り除き，x_1 や x_2 のみを含む木を抽象構文木という．ここでは例をみて理解してほしい．



3　パーサコンビネータ

パーサコンビネータは，文脈自由文法をそのままに書けるパーサだと思えばよろしい．パーサコンビネータの実装には，Haskell では，Parsec[1]，Attoparsec[2]，Trifecta[3] などがある．たとえば，冒頭で述べた Java の if-else 文を，Haskell の Parsec であらわすと，次のようになる．

```
import qualified Text.Parsec as P

expr :: Monad m => P.ParsecT String u m ()
expr = return ()

stmt :: Monad m => P.ParsecT String u m ()
stmt = P.string "if" >>
       P.char '(' >> expr >> P.char ')' >>
       stmt >> P.string "else" >> stmt
```

キーワード **if** は`P.string "if"`のようにしてあらわす．カッコは`P.char '('`のようにしてあらわす．変数は Haskell における識別子であらわす．そして，それぞれの記号を “>>”で繋ぐ．記号を置き換えて読んでみれば，このコードが生成規則をそのまま写したものであるように見えるのはわかるであろう．

3.1　正規表現とパーサコンビネータ

もし正規表現がわかるなら，正規表現を翻訳してみるのがパーサコンビネータを理解する早道である．たとえば，正規表現`/abc/`を考えてみよう．これは，正規表現の文法にしたがえば，abc という文字列にマッチする．これをパーサコンビネータに翻訳すると，`string "abc"`となる．正規表現という

[1] `http://hackage.haskell.org/package/parsec`
[2] `http://hackage.haskell.org/package/attoparsec`
[3] `http://hackage.haskell.org/package/trifecta`

ならば，当然*や+ といったオペレータをもつのではないかと疑問に思うだろう．たとえば，/a*/はどのように翻訳されるのか？　答えは，many (string "a") である．簡単な対応表を以下に示す．ここで $\alpha \Longrightarrow \beta$ は “α は β に翻訳される”と読む．ただし，正規表現は Perl のものを意図している．

- /x/ $\Longrightarrow$ string "x"
- /x*/ $\Longrightarrow$ many (string "x")
- /x+/ $\Longrightarrow$ many1 (string "x") または some (string "x") (実装によって異なることに注意する．)
- /x?/ $\Longrightarrow$ optional (string "x")
- /x(?R)|$/ $\Longrightarrow$ let r = do { string "x"; r <|> string "\n" }

4　付録

簡単な S 式パーサの完全な実装を Parsec，Attoparsec，Trifecta それぞれで例として示す．例のコードはすべて同じ仕様である．例のコードの仕様は，引数なしでコマンドラインから起動されると，まず 1 行入力を受け付ける．そしてその行を S 式としてパースし，問題なくパースできたか失敗したかを人間が目で見て判断できるような結果を出力する．

たとえば，Trifecta の例は，1 行目に次のように与えると，

```
(x1.x2)
```

次のように出力する．これは抽象構文木を文字の列として表現したものである．

```
Cons (Sym "x1") (Sym "x2")
```

1 行目に次のように与えると，

```
(x1.x2
```

次のように出力する．これはエラーメッセージである．

```
(interactive):1:7: error: unexpected EOF, expected: ")",
    white space
(x1.x2<EOF>
      ^
```

付録のサンプルコードをビルドするには，まず本書のソースコードを手に入れて，本書のソースコードがあるディレクトリまでいく．そしてそのあと，cabal-dev を使用してビルドできる．./cabal-dev/bin 以下にバイナリが用意される．

```
git clone https://github.com/pasberth/pasberth.github.io.git
cd pasberth.github.io
git checkout default
cd papers/brief-intro-parser-combinators
cabal-dev install
ls ./cabal-dev/bin
```

4.1 Parsec

Parsec はパーサコンビネータの実装のひとつで，Haskell で書かれている．
以下に，Parsec で S 式をパースする完全な例を示す[4]．

```
import qualified Text.Parsec as P

data SExp
  = Sym String
  | Cons SExp SExp
  deriving (Show)

sym :: Monad m => P.ParsecT String u m SExp
sym = do
  s <- P.many1 $ P.noneOf "(.) \t"
  return $ Sym s

cons :: Monad m => P.ParsecT String u m SExp
cons = do
  P.char '('
  P.spaces
  x <- sexp
  P.spaces
  P.char '.'
  P.spaces
  y <- sexp
  P.spaces
  P.char ')'
  return $ Cons x y

sexp :: Monad m => P.ParsecT String u m SExp
```

[4]ファイルは https://raw.github.com/pasberth/pasberth.github.io/default/papers/brief-intro-parser-combinators/sexp-parsec.hs から入手できる．

```
sexp = cons P.<|> sym

main :: IO ()
main = do
  s <- getLine
  let result
        = P.parse
            (do { x <- sexp ; P.eof ; return x })
            "<stdin>"
            s
  print result
```

4.2 Attoparsec

Attoparsec はパーサコンビネータの実装のひとつで，Haskell で書かれている．ByteString を使用するので Parsec より高速である．以下に，Attoparsec で S 式をパースする完全な例を示す[5]．

```
{-# LANGUAGE OverloadedStrings #-}

import qualified Control.Applicative as A
import qualified Data.ByteString as B
import qualified Data.Attoparsec as P
import qualified Data.Attoparsec.ByteString.Char8 as P8

data SExp
  = Sym String
  | Cons SExp SExp
  deriving (Show)

sym :: P.Parser SExp
sym = do
  s <- P.many1 $ P8.satisfy (not . (`elem` "(.) \t"))
  return $ Sym s

cons :: P.Parser SExp
cons = do
  P8.char '('
```

[5]ファイルは https://raw.github.com/pasberth/pasberth.github.io/default/papers/brief-intro-parser-combinators/sexp-attoparsec.hs から入手できる．

```
  P.many' P8.space
  x <- sexp
  P.many' P8.space
  P8.char '.'
  P.many' P8.space
  y <- sexp
  P.many' P8.space
  P8.char ')'
  return $ Cons x y

sexp :: P.Parser SExp
sexp = cons A.<|> sym

main :: IO ()
main = do
  s <- B.getLine
  let result
        = P.parse sexp s
  print $ P.eitherResult result
```

4.3 Trifecta

Trifecta はパーサコンビネータの実装のひとつで，Haskell で書かれている．今回紹介した 3 つのなかではいちばん新しく，高度に抽象化されているので，今イチバンイケてる実装である．以下に，Trifecta で S 式をパースする完全な例を示す[6]．

```
import qualified Control.Applicative as A
import qualified Text.Trifecta as P

data SExp
  = Sym String
  | Cons SExp SExp
  deriving (Show)

sym :: P.Parser SExp
sym = do
  s <- A.some $ P.noneOf "(.) \t"
```

[6]ファイルは https://raw.github.com/pasberth/pasberth.github.io/default/papers/brief-intro-parser-combinators/sexp-trifecta.hs から入手できる．

```
  return $ Sym s

cons :: P.Parser SExp
cons = do
  P.char '('
  P.spaces
  x <- sexp
  P.spaces
  P.char '.'
  P.spaces
  y <- sexp
  P.spaces
  P.char ')'
  return $ Cons x y

sexp :: P.Parser SExp
sexp = cons A.<|> sym

main :: IO ()
main = do
  s <- getLine
  P.parseTest (do { x <- sexp ; P.eof ; return x }) s
```

4.4 おまけ: Applicative

Control.Applicative は Haskell の標準ライブラリである．これを使用すると，パーサコンビネータをよりスマートに利用できる．前の節で書いた Trifecta を使用したコードを Applicative を使用して書き直す例を示す[7].

```
import           Control.Applicative
import qualified Text.Trifecta as P

data SExp
  = Sym String
  | Cons SExp SExp
  deriving (Show)
```

[7]ファイルは https://raw.github.com/pasberth/pasberth.github.io/default/papers/brief-intro-parser-combinators/sexp-trifecta-applicative.hs から入手できる．

```
sym :: P.Parser SExp
sym = Sym <$> (some $ P.noneOf "(.) \t")

cons :: P.Parser SExp
cons = Cons <$> (P.char '(' *> P.spaces *> sexp)
            <* P.spaces <* P.char '.'
            <*> sexp <* P.spaces <* P.char ')'

sexp :: P.Parser SExp
sexp = cons <|> sym

main :: IO ()
main = do
  s <- getLine
  P.parseTest (sexp <* P.eof) s
```

参考文献

[1] コンパイラ―原理・技法・ツール