コンパイラ資料(JavaCC)　佐々木晃

構文解析器を生成する JavaCC

生成系(generator)
形式的記述をもとにした仕様記述からプログラムを自動生成する。コンパイラ・コンパイラ(compiler compiler)とも呼ばれる。生成系の代表例を挙げる。

- 字句解析器生成系: 正規表現→字句解析器
  - JFlex
  - lex,flex (for C, etc)
- 構文解析器生成系: BNF→構文解析器
  - JavaCC, Antlr(LL), JCup, jay(LR) (for java)
  - yacc, bison (for C, etc)
- 意味解析器の生成系: 属性文法など
  - Rie

JavaCC の概略、使い方。

- LL(k)文法の拡張 BNF に基づく記述から構文解析器を自動生成する。
- 字句解析器も自動生成する。

JavaCC についての説明は下記。ソフトウェアもダウンロード可能。
http://cis.k.hosei.ac.jp/~nakata/lectureCompiler/JavaCC/index.html

例題による javacc の紹介

(参考プログラム: http://cis.k.hosei.ac.jp/~asasaki/lect/compiler/2007-1113.zip)
ここでは、正整数をオペランドとする加算式を表す言語を考え、その構文解析器を作成してみよう。まず、言語を定義する。
トークンの定義

```
<NUM> ::= (0|1|2|...9)*
<ADDOP> ::= '+'
```

構文の定義

```
E → E <ADDOP> F;
F → <NUM>
```

例えば次の式は、この文法に従っている。
sample1.txt

```
123+456
```

sample2.txt

```
123+456+789
```

上記の文法定義に基づいて LL(1)構文解析器を作るためには、構文の左再帰を除去するか、括り出しをする必要がある。左再帰除去をした場合は、

```
E → F E'
E' → <ADDOP>F  E' | ε;
F → <NUM>
```

となる。この文法に対応した構文解析プログラムを手で書いた場合は、配布した test1/Parser1.java である。

練習問題：コマンドラインで java のコンパイルと実行を行ってみよ。

```
javac *.java
java Parser1
```

JavaCC による構文解析器の作成。

(A) 左再帰を除去した文法に基づく構文解析器

上記の例を javacc のプログラムとして記述すると下記のようになる。javacc では、文脈自由文法(BNF)の記述から構文解析プログラムを生成する。字句解析器も正規表現に基づいた記述から自動で生成される。(SKIP,TOKEN の箇所)

プログラム test1.jj

下記の①, ②, ③, ④からなる。

①Javacc のオプションなどの設定

```
options {
 JDK_VERSION="1.5";
}
```

②メインルーチン（構文解析器インスタンスの作成と呼び出し）

```
PARSER_BEGIN(Sample)
import java.io.*;

public class Sample {
  public static void main(String args[]) {
    try {
       Sample parser = new Sample( new FileReader(args[0] ));
       parser.start();
    }catch(Exception ex){
    ex.printStackTrace();
    System.err.println("Failed to parse:"+ex.getMessage());
    }
  }
}
PARSER_END(Sample)
```

③字句解析器の定義（トークンの定義)

```
SKIP:{
     <SPACE: " "|"¥t">
     | <NL:"¥n"|"¥r"|"¥r¥n">
}

TOKEN:{
     <NUMBER:(["0"-"9"])+>
     |<ADDOP:"+">
}
```

④構文解析器の定義（この場合、単に解析するだけ。→入力された言語が、文法に従っているかどうかを判定)

```
void start(): {}
{
  e()
}

void e():{}
{
  f() e2()
}


void e2():{}
```

```
{
  <ADDOP> f() e2()
  | {}
}

void f():{}
{
  <NUMBER>
}
```

<自動生成の方法>　コマンドを使って１ステップずつ作成していく方法で説明する。

(1) javacc は、上記のプログラムを java のプログラムに変換する。

```
javacc test1.jj
```

さまざまなファイルが生成(generate)されるが Sample.java が本体である。(②の PARSER_BEGIN(Sample)の「Sample」が出力するプログラム名となる。)
その中身を見ると、自動的に変換されたものなので多少読みにくいが LL(1)構文解析器が作られていることがわかる。

(2) これをコンパイルし実行するには、生成された java ファイルをすべてコンパイルし、Sample クラスの static main メソッドを呼び出せば良い。sample1.txt は入力である。

```
javac *.java
java Sample sample1.txt
```

(例) 逆ポーランド記法への変換
逆ポーランド記法への変換は、文法と合わせて次のように表すことができる。[と]で囲まれた部分でその文字列を表示することを示している。<NUM>.image はトークン<NUM>に対する実際の文字列である。

```
E → F E'
E' → <ADDOP> E' [+]F | ε;
F → <NUM> [<NUM>.image]
```

javacc では、"{"と"}"で囲まれた部分に java の文（アクション）を記述することができる。(下記は変更部のみ)

```
void e2():{}
{
  <ADDOP> f() {System.out.print(" +");} e2()
  | {}
}

void f():{Token token;}
{
  token=<NUMBER> {System.out.print(" "+ token.image);}
}
```

アクション。F を解析後プラスを表示。

ローカル変数 token を宣言。<NUMBER>で解析した字句が代入される。token.image で、字句 (この場合、もとの数字列) を取り出すことが出来る。

アクション。読み込んだトークンの字句を空白の後に表示

ただし、アクションを記述する部分は通常 javacc では解析されないので、もし間違って記述した場合には、javacc コマンドでエラーが検出されず、javac でのコンパイル時に発見される。例えば、System.out を Systm.out とかに変えて javacc を実行してみてどうなるか試してみよ。

練習問題: マイナスが使えるように拡張してみよ。

```
E → F E'
E' → <ADDOP> E' [+]F | <MINUSOP> E' [-]F | ε;
F → <NUM> [<NUM>.image]
```

(B) 括り出しを利用する場合

javacc では正規右辺文法(拡張 BNF)を利用することができる。左再帰を除去した文法よりシンプルになる。
今後のコンパイラ作成演習では、こちらの方法を使う方が楽である。

```
E → F {<ADDOP> F}
F → <NUM>
```

配布した test2/Parser2.java が、この文法に従って作成した解析プログラムであるが、javacc では次のようにシンプルに書くことができる。(下記④の変更部のみ。void e2(){…}は必要なし)

```
void e():{}
{
  f() (<ADDOP> f())*
}

void f():{}
{
  <NUMBER>
}
```

逆ポーランド記法に変換する表現は、

```
E → F {+ F [+]}
F → <NUM> [<NUM>.image]
```

```
void e():{}
{
  f() (<ADDOP> f() {System.out.print(" +");})*
}

void f():{Token token;}
{
  token=<NUMBER> {System.out.print(" "+ token.image);}
}
```

練習問題:マイナスが使えるように拡張せよ

```
E → F {+ F [+] | - F [-]}
F → <NUM> [<NUM>.image]
```

(C)メソッドの返り値を利用した値の受け渡し

配布した test2/Parser2.java では、構文解析をしながら部分式の計算結果をメソッドの返り値として受け渡すことができるようにしている。(test2/Parser1.java と比較せよ)。Javacc では、このような値の受け渡しの機構も持っている。

```
void start(): {int result;}
{
  result=e()
  {System.out.println("result="+result);}
}

int e():{int result; int num;}
{
  num=f() {result = num; }
      (<ADDOP> num=f() {result += num;})*
  {return result;}
}
```

```
int f():{Token token;}
{
  token=<NUMBER> {return Integer.parseInt(token.image);}
}
```

練習問題: マイナスが使えるように拡張せよ。

サンプルその２ (javacc の説明ページ掲載のもの)

正規右辺文法(拡張 BNF)を解析する例

```
LIST = {EXP "¥n"}                          // 改行を終了文字とした 0 個以上の EXP の並び
EXP = NUM {"+" NUM}                        // EXP=NUM | NUM+NUM | ... | NUM+NUM+ .. +NUM
```

sample.jj, (java-1.5 の機能を使う人は最初の 3 行必須。)

```
options {
 JDK_VERSION="1.5";
}

PARSER_BEGIN(Sample)
import java.io.*;

public class Sample {
  public static void main(String args[]) {
    try {
       Sample parser = new Sample( new FileReader(args[0] ));
       parser.list();
    }catch(Exception ex){
    System.err.println("Failed to parse:"+ex.getMessage());
    }
  }
}
PARSER_END(Sample)

SKIP:{
     <SPACE: " "|"¥t">
}

TOKEN:{
     <NUMBER:(["0"-"9"])+>
    | <NL:"¥n"|"¥r"|"¥r¥n">
    |<ADDOP:"+">
}

void list(): {Integer value;}
{
  (value=expression()<NL>  {System.out.println(value);}
  )*
}

Integer expression():{int result;Token tokenNum;}
{
  tokenNum=<NUMBER>     {result=Integer.parseInt(tokenNum.image);}
  (<ADDOP> tokenNum=<NUMBER>
```

```
            {result+=Integer.parseInt(tokenNum.image);}
  )*        {return new Integer(result);}
}
```

テスト入力(2行目の最後は改行でないと駄目)

```
3 + 40 + 20
40 + 50
```

下記は、もとの文法だけを抜き出して記述したもの(最初の宣言は省略)。この場合、入力が文法にあっている場合は、何もせず終了する。あっていない場合はエラー表示する。

```
SKIP:{
     <SPACE: " "|"¥t">
}

TOKEN:{
     <NUMBER:(["0"-"9"])+>
    |<NL:"¥n"|"¥r"|"¥r¥n">
    |<ADDOP:"+">
}

void list():{}
{
  (expression()<NL>
  )*
}

void expression():{}
{
  <NUMBER>
  (<ADDOP> <NUMBER>
  )*
}
```

逆ポーランドに変換するバージョン。解析の途中にコードを埋めることができる。手続きの結果をローカル変数に渡すことができる。変更点は下線。

```
void list():{}
{
  (expression()<NL> {System.out.println();}
  )*
}

void expression():{Token tokenNum;}
{
  tokenNum=<NUMBER>     {System.out.print(tokenNum.image+" ");}
  (<ADDOP> tokenNum=<NUMBER>
            {System.out.print(tokenNum.image+" ");
             System.out.print("+ ");
            }
  )*
}
```