JavaCC 简介

javacc是什么？下面是javacc官方文档上的定义
Java Compiler Compiler (JavaCC [tm]) is the most popular parser generator for use with Java applications. A parser generator is a tool that reads a grammar specification and converts it to a Java program that can recognize matches to the grammar. In addition to the parser generator itself, JavaCC provides other standard capabilities related to parser generation such as tree building (via a tool called JJTree included with JavaCC), actions, debugging, etc.
不过在俺看来就是一个代码生成器,生成的代码可以再处理其它别的数据或是代码.相当的方便.

官方的网址是javacc，使用方法:
1.编辑*.jj文件
2.javacc *.jj
3.javac *.java
4.使用生成的CLASS文件

简介

Theodore S. Norvell 写的一本《The JavaCC Tutorial》的第一章Introduction to JavaCC

JavaCC 入门

1. JavaCC 和分析器生成程序

JavaCC 是一个能生成语法和词法分析器的生成程序。语法和词法分析器是字符串处理软件的重要组件。编译器和解释器集成了词法和语法分析器来解释那些含有程序的文件，不管怎样，词法和预防分析器被广泛用于各种应用，所以我希望这本书中的示例能阐述清楚。

那么什么是词法和语法分析器呢？词法分析器能把一串字符切分成一溜叫做*token* 的子串并把它们归类。看一个用C 语言写的小程序。

```
int main() {
return 0 ;
}
```

C 编译器的词法分析器会把上面的程序切割成下面的一串token

“int”, “ ”, “main”, “(”, “)”,

“ ”, “{”, “\n”, “\t”, “return”

“ ”, “0”, “ ”, “;”, “\n”,

“}”, “\n”, “” .

词法分析器还会识别每个token 的类型；在这个例子中这个token 串的类型可能是

KWINT, SPACE, ID, OPAR, CPAR,

SPACE, OBRACE, SPACE, SPACE, KWRETURN,

SPACE, OCTALCONST, SPACE, SEMICOLON, SPACE,

CBRACE, SPACE, EOF .

EOF 这种token 表示输入文件结束。Token 流将会传给分析器。在这个C 语言的例子中，分析器并不需要所有的token；本例那些被归为SPACE 的 token 不会传给分析器。分析器将会分析token 流以决定程序的结构。通常在编译器中，分析器输出一棵代表程序结构的树。这棵树将会作为编译器语法分析和代码生成的输入的一部分。考虑某个程序里的一个语句：

fahrenheit = 32.0 + 9.0 * celcius / 5.0 ;

分析器根据语法规则分析这个表达式并生成一棵树：

图1fahrenheit = 32.0 + 9.0 * celcius / 5.0 （译注：原文缺失该图，此图为译者根据前后语境所画）

如果输入不遵循语言的词法或句法规则，词法分析器同时也负责产生出错信息。

JavaCC 本身并不是一个词法分析器或是语法分析器，而是一个分析程序生成器。这就是说它可以根据输入的语言规范输出一个词法和语法分析器。JavaCC 生成的是用Java 写成的词法和语法分析器。见下图TBD：

DIAGRAM TBD

词法和语法分析器本身就是一个冗长而又复杂的组件。手工编写一个这样的程序需要仔细考虑各种语法规则的相互影响。比如在一个C 的词法分析器中，处理整数的代码和处理浮点常量的代码不能相互独立，因为整数和浮点数的开头都是数字。而使用像JavaCC 这一点分析程序生成器时，处理整数的规则和处理浮点数的是分开书写的，而它们之间公共的代码在生成器处理时被抽取出来了。模块性的增加意味着语言规范比手写的Java 程序更容易编写、阅读和修改。通过使用JavaCC 这样的分析程序生成器，使软件工程师节约了不少时间，同时也增强了所编写软件的质量。

2. 第一个例子——整数相加

作为第一个例子，我们把一串数字加起来，像这样

99 + 42 + 0 + 15

我们忽略所有数字和符号间的空格和换行符，除此之外，我们不接受除了10个数字和加号之外的其他字符。

本节后面的代码都出自一个叫“adder.jj”的文件。这个文件含有符合JavaCC 规范的词法和语法说明，用来作为JavaCC 的输入。

2.1. 选项和类声明

文件的第一部分

```
/* adder.jj Adding up numbers */

options {

STATIC = false ;

}

PARSER BEGIN(Adder)

class Adder {

public static void main( String[] args )
```

```
throws ParseException, TokenMgrError {

Adder parser = new Adder( System.in ) ;

parser.Start() ; }

}

PARSER END(Adder)
```

在第一个注释之后的是选项段；除了STATIC 这一项（缺省为true），所有其他的JavaCC选都为默认值。关于JavaCC 选项的更多信息，请参考JavaCC 的文档、本书的以后的章节和FAQ。接下来定义了一个叫做Adder 的Java 类，但在这你所看到的不是Adder 类的全部；JavaCC 会在处理时为这个类添加其他代码。main 方法宣称可能在运行时隐式的抛出两个异常：ParseException 和TokenMgrError；这些类都会由JavaCC生成。

2.2. 详述词法分析器

我们待会儿再看那个main函数，现在我们首先来定义一个词法分析器。在这个简单的例子中，词法分析器的定义只有4行：

```
SKIP : { " " }

SKIP : { "\n" | "\r" | "\r\n" }

TOKEN : { < PLUS : "+" > }

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

第一行说明了空格是一个token,但是会被忽略。所以解析器并不会收到任何单独的空格。

第二行也说了差不多的事情，只不过被忽略的是换行符，换行符会因操作系统而不同。Unix/Linux 采用LF (linefeed)字符；DOS 和Windows 则用CR+LF (carriage + linefeed)，在老的Macintoshes 机子上，就用一个回车表示。我们要告之JavaCC 所有的可能，就如上面用一个小竖线"|" 把不同的匹配模式隔开。

第三行告诉JavaCC一个单独的加号是一个token,而且给这个Token取了一个名字：PLUS。

最后一行告诉JavaCC数字的语法并为它们取名为NUMBER。如果你熟悉Perl或者Java的正则表达式包，就不难明白这些式子的含义。让我们仔细看一下这个表达式(["0"-"9"])+。圆括号中的 ["0"-"9"] 是一个匹配任意数字的正则表达式，这表明unicode编码中的0-9之间的字符都能被匹配。一个形如 (x)+ 的正则式可以匹配任意重复的x 串。所以表达式 (["0"-"9"])+ 就可以匹配任意连续数字串。这四行每一行都是一个正则表达式实例(regular expression production)。

还有一种由词法分析器生成的token，它的名字是EOF，正如其名，它代表了输入的终止。不能，也不需要任何对EOF的匹配，JavaCC会自动生成它们。

考虑一个包含如下字符串的输入文件：

"123 + 456\n"

我们定义的词法分析器将会找到7个token: NUMBER, 空格, PLUS, 又一个空格, 另一个数字,一个换行, 然后是EOF。当然，标记了SKIP的token不会被传到解析器。所以，解析器只会看到这些东西:

NUMBER, PLUS, NUMBER, EOF

设想一个包含未定义字符的输入文件，例如：

“123 – 456\n”

在处理完第一个空格之后，我们的可爱的词法分析器将遇到一个不认识的字符：减号。由于没有任何token的定义是以减号打头，词法分析器会扔出一个TokenMgrError 异常。

现在我们看看另一种情况：

“123++456\n”

我们的词法分析器会提交一个这样的串：

NUMBER,PLUS,PLUS,NUMBER,EOF

词法分析器还没有智能到判断一个token 序列是否有意义，这通常是语法分析器的工作。我们接下来要讨论的解析器会在词法分析器提交第二个PLUS 之后发觉这个错误，然后拒绝处理之后的任何token。所以解析器实际上处理的只有：

NUMBER,PLUS,PLUS

同时，跳过(skip)一个token并不代表忽略(ignore)它。考虑下列输入：

“123 456\n”

词法分析器会识别出3个token：两个NUMBER和夹在它们中间的空格；然后报错。

2.3. 详述语法分析器

语法分析器的定义使用了一种叫BNF范式的东西，这看起来有点像Java的方法定义：

```
void Start() :
{}
{
<NUMBER>
(
<PLUS>
<NUMBER>
)*
<EOF>
}
```

这个BNF范式声明了一个正确的输入序列的模式。我们解释一下它的意思：它以NUMBER开头的序列，以EOF结束，中间存在零个或多个由一个PLUS 后面跟一个NUMBER 组成的子序列。

正如所见，语法分析器只会检查一个输入序列是否合法，而并没有真的把数字加起来。待会儿我们还会修改这个语法分析器，但现在我们先让它生成Java 组件，然后run 起来。

2.4. 生成一个解析器和一个词法分析器

我们现在用JavaCC根据我们写好的adder.jj 文件生成分析器。具体怎么做依赖于操作系统。下面是在Windows NT, 2000 和 XP 上完成的。首先使用“命令提示符”程序（CMD.EXE）运行JavaCC：

D:\home\JavaCC-Book\adder>javacc adder.jj

Java Compiler Compiler Version 2.1 (Parser Generator)

Copyright (c) 1996-2001 Sun Microsystems, Inc.

Copyright (c) 1997-2001 WebGain, Inc.

(type "javacc" with no arguments for help)

Reading from file adder.jj . . .

File "TokenMgrError.java" does not exist. Will create one.

File "ParseException.java" does not exist. Will create one.

File "Token.java" does not exist. Will create one.

File "SimpleCharStream.java" does not exist. Will create one.

Parser generated successfully.

这个操作生成了七个Java 类，每一个在独立的文件中：

l TokenMgrError 是一个简单的错误类；词法分析器用它来侦测错误，父类是Throwable.

l ParserException 是另一个错误类；解析器用它侦测错误，父类是Exception，因此也是Throwable 的子类。

l Token 是一个表示token 的类。每个Token 对象都有一个整数域kind 表示token的类型（PLUS, NUMBER, 或者EOF），和一个String 域image，存储token 所代表的内容。

l SimpleCharStream 是一个把字符串提交给词法分析器的接口转换类。

l AdderConstants 是一个接口，定义了一组在词法分析器和解析器中都要用到的类。

l AdderTokenManager 就是词法分析器。

l Adder 是解析器。

现在我们可以用一个Java 编译器编译这些类了：

D:\home\JavaCC-Book\adder>javac *.java

2.5. 让它跑起来

现在我们换个角度来看**Adder**类的main 方法。

```
static void main( String[] args )
throws ParseException, TokenMgrError {
Adder parser = new Adder( System.in ) ;
parser.Start() ;
}
```

最先注意到main 可能会抛出继承自Throwable 的两个子类（译注：TokenMgrError 和ParserException）中的任意一个。这风格不是很好，我们应该捕捉这些异常。但是为了保持第

一个例子简洁（译注：为了让读者能迅速把握要点，而不是陷入无穷的细节之中），我们忽略了这些东西。

第一个语句创建了一个解析器实例，构建函数使用了自动生成的接受一个java.io.InputStream的重载。其实还有一个（更好的）接受Reader实例的重载（java建议在处理字符串时尽量使用Reader(Writer)而不是InputStream(OutputStream),这样能更好的避免字符编码带来的问题——译者如是说）。这个构建函数创建了一个SimpleCharStream对象和一个词法分析器AdderTokenManager的实例。这样，词法分析器通过SimpleCharStream顺利地获取到了我们的输入。

第二句调用了一个由JavaCC生成的方法Start()。对语法规范中的每个BNF产生式，JavaCC都会生成一个对应的方法。这个方法负责尝试在输入序列中寻找符合模式的输入。例如，调用Start时会使解析器试图寻找一个匹配下面模式的输入序列：

<NUMBER>(<PLUS><NUMBER>)*<EOF>

我们可以准备一个合适的输入然后运行这条命令

D:\home\JavaCC-Book\adder>java Adder <input.txt

我们运行程序，输入表达式以后，会出现以下三种不同的情况：

1. 出现词法错误。本例中，词法错误只出现在遇到未知字符时。我们可以通过下面的输入引发一个词法错误：

“123-456\n”

这种情况下，程序会抛出一个TokenMrgError异常。这个异常的message域是：Exception in thread “main” TokenMgrError: Lexical error at line 1,column 5. Encountered: “-“ (45), after : “”

2. 出现一个解析错误。这发生在输入序列不符合Start的BNF范式时。例如

“123++456\n”

或者

“123 456\n”

或者

“\n”

这时，程序会扔出一个ParseException异常。这种异常的第一条信息分别是：

Exception in thread “main” ParseException: Encountered ”+” at line 1, column 6.

Was expecting:

<NUMBER> ...

3. 输入串符合Start的定义。这时，程序不抛出任何异常，只会默默的停止。

由于解析器除了挑错什么都不做，所有现在这个程序除了检查输入合法性以外什么都做不了。在下一节，我们将会做一些改变让它更有用。

2.6. 生成的代码

为了了解JavaCC生成的代码是如何工作的，最好的办法是看看它生成的代码。

final public void Start() throws ParseException {

jj consume token(NUMBER);

```
label 1:
while (true) {
jj consume token(PLUS);
jj consume token(NUMBER);
switch ((jj ntk == -1) ? jj ntk() : jj ntk) {
case PLUS:
;
break;
default:
jj la1[0] = jj gen;
break label 1;
}
}
jj consume token(0);
}
```

方法jj_consume_token将试图从输入中读取一个指定类型的token，如果得到的token与期望的类型不符，则抛出一个异常。表达式

(jj_ntk==-1)?jj_ntk():jj_ntk

计算下一个未读token的类型。而最后一行则要求匹配一个类型0的token；JavaCC 总是用0 来编码EOF类型。

2.7. 增强解析器

像上文中提到的start方法一样的，由JavaCC根据BNF文法生成的方法，在默认情况下仅仅是检查了输入是否符合规则。但是我们可以在BNF中间夹杂Java代码，这些代码将来会被包含在生成的方法中。JavaCC为我们提供了一个骨架，而我们要让它有血有肉。

下面我们改变adder.jj中的BNF规范，为Start 添加一些声明和Java 代码。新的文件叫做adder1.jj。添加或改变的部分用黑体标出：

```
 int start() throws NumberFormatException:
{
Token t;
int i;
int value;
}
{
t = <NUMBER>
```

```
{ i = Integer.parseInt(t.image); }

{ value = i; }

(

<PLUS>

t = <NUMBER>

{ i = Integer.parseInt(t.image);}

{ value += i; }

)*

<EOF>

{ return value; }

}
```

首先，我们定义了BNF产生式的返回类型，这样生成的方法就从void 变为int。然后还声明了NumberFormatException可能会在运行时抛出。我们定义了三个变量。变量t 是一个Token，Token 是一个生成的类用来表示token；Token 类的image 域记录了匹配的字符串。当一个token 匹配上了一个BNF 产生式，我们就能通过赋上一个引用来记下这个Token 对象。像这样

t = <NUMBER>

我们可以在BNF产生式的大括号里添加任意的Java语句，这些语句会原封不动的copy到生产的代码里面。

由于更改了Start的返回类型，我们有必要更改一下我们的main函数：

```
static void main( String[] args )

throws ParseException, TokenMgrError, NumberFormatException {

Adder parser = new Adder( System.in ) ;

int val = parser.Start() ;

System.out.println(val);

}
```

在结束这个例子前，我们再做一点小小的改进。下面的代码在start中出现了两次：

```
t = <NUMBER>

{ i = Integer.parseInt( t.image ) ; }
```

虽然在这个例子中不会引起太大的差异，仅仅涉及两行代码，但这种重复会导致维护的问题。所以我们把这两行提出来作为另一个BNF 产生式，叫做Primary。最新的修改依旧用黑体标出。

```
int start() throws NumberFormatException:

{
```

```
int i;
int value;
}
{
value = Primary()
(
  <PLUS>
  i = Primary()
  { value += i; }
)*
<EOF>
{ return value; }
}
int Primary() throws NumberFormatException :
{
Token t ;
}
{
t=<NUMBER>
{ return Integer.parseInt( t.image ) ; }
}
```

这时我们再来看看JavaCC所生成的代码：

```
final public int Start() throws ParseException, NumberFormatException {
int i ;
int value ;
value = Primary();
label 1:
while (true) {
switch ((jj_ntk==-1)?jj_ntk():jj_ntk) {
case PLUS:
;
break;
default:
```

```
jj la1[0] = jj gen;
break label 1;
}
jj consume token(PLUS);
i = Primary();
value += i ;
}
jj consume token(0);
{ if (true) return value ; }
throw new Error(”Missing return statement in function”);
}

final public int Primary() throws ParseException, NumberFormatException {
Token t ;
t = jj consume token(NUMBER);
{ if (true) return Integer.parseInt( t.image ); }
throw new Error(”Missing return statement in function”);
}
```

待会儿我们还能看到如何向BNF产生式传递参数。

3. 第二个例子：运算器

接下来，我们继续改进我们的adder,使它成为一个简易的四则运算计算器。

第一步，我们让它能够和我们进行交互，把每行作为一个单独的表达式，并计算输出。稍后，我们会考虑加法之外的其他操作，减法，乘法和除法。

3.1. 选项和类定义

calculator0.jj的开头如下：

```
 /* calculator0.jj An interactive calculator. */
options {
STATIC = false ;
}

PARSER BEGIN(Calculator)
import java.io.PrintStream ;
```

```
class Calculator {

static void main( String[] args )

throws ParseException, TokenMgrError, NumberFormatException {

Calculator parser = new Calculator( System.in ) ;

parser.Start( System.out ) ;

}

double previousValue = 0.0 ;

}

PARSER END(Calculator)
```

类Calculator 的previousValue 域用于保存前一行的计算结果，我们的下一版本将允许在表达式中使用美元符号($)表示这个值。import语句可以写在PARSER_BEGIN和PARSER_END之间，他们将被复制到生成的类文件中，包定义同样也在这时声明。

3.2. 词法定义

词法定义的改变不大，首先，换行符不再被忽略，而声明成一个token，这使得换行符可以被解析器处理。

```
SKIP:{" "}

TOKEN:{< EOL : "\n"|"\r"|"\r\n" >}

TOKEN:{< PLUS : "+">}
```

第二，我们将允许小数参与运算，所以我们要更改NUMBER的定义使得它允许小数点被匹配。一共有4种形式（用竖线隔开）：没有小数部分，既有小数部分又有整数部分，只有小数点和小数部分，只有整数部分和小数点。于是我们声明如下：

```
TOKEN:{< NUMBER :

(["0"-"9"])+|

(["0"-"9"])+"." (["0"-"9"])+ |

(["0"-"9"])+"."|

"." (["0"-"9"])+

>}
```

我们又发现相同的正则表达式出现了好多次，这显然不是个好现象，所有我们可以给这部分重复的表达式起一个名字。这个名字仅仅在这个词法分析器中有效，而且不代表任何token类型，这样的正则表达式在定义中用# 标记。前一表达式等价于以下代码：

```
TOKEN:{< NUMBER : <DIGITS>| <DIGITS>"." <DIGITS> | <DIGITS>"."|"." <DIGITS> >}

TOKEN : {< #DIGITS : (["0"-"9"])+ >}
```

3.3. 解析器定义

解析器的输入包括了若干行序列，每行都包含一个表达式。用BNF（下一章我们还会讨论（译注：没有下一章了））表示这种结构就是：

```
Start ->(Expression EOL)* EOF
```

下面给出Start BNF 产生式的框架：

```
void Start():
{}
{
(
Expression()
<EOL>
)*
<EOF>
}
```

我们会在这个框架之上添加了一些Java代码，让它能记录并打印出每行表达式的值：

```
void Start(PrintStream printStream) throws NumberFormatException :
{}
{
(
previousValue = Expression()
<EOL>
{ printStream.println(previousValue); }
)*
<EOF>
}
```

每个表达式都包括了一个或者多个由加号（目前它还只认加号）分隔的数字序列。BNF表示如下：

expression -> primary (PLUS primary)*

这里的primary现在只表示数字。这个BNF翻译成JavaCC 的记法就是（增加的用粗体显示）：

```
double Expression() throws NumberFormatException :
{
double i;
double value;
}
```

```
{
value = primary()
(
<PLUS>
i = primary()
{ value += i;}
)*
{ return value; }
}
```

这个和adder1.jj中Start的定义惊人的相似啊，不过我们吧int改成了double。

primary的定义也和adder1.jj中的差不多，用BNF表示非常简单：

Primary -> NUMBER

除了它现在能计算双精度数字外一切如前:

```
double primary() throws NumberFormatException:
{
Token t;
}
{
t = <NUMBER>
{ return Double.parseDouble(t.image); }
}
```

总结一下我们用到的BNF：

Start ->(Expression EOL)* EOF

expression -> primary (PLUS primary)*

Primary -> NUMBER

至此，我们已经完成了calculator.jj。下面我们要测试一下它。

3.4. 增加减法操作

为了得到一个功能丰富的计算器，我们需要能执行更多的操作，比如减法、乘法和除法。我们先从减法开始。

在词法定义里添加一个新的产生式：

TOKEN :{ < MINUS : “-“ > }

在定义EOL和NUMBER时我们使用了小竖线分割不同选项，现在我们要使用同样的方法吧减号添加进EXPRESSION的定义中，我们的BNF如下：

Expression -> Primary((PLUS|MINUS) Primary)*

还有另外一种等价形式：

Expression -> Primary(PLUS Primary |MINUS Primary)*

因为第二种形式处理起来更简单些，所有我们用第二种形式。这样我们就得到了新的JavaCC代码：

```
double Expression() throws NumberFormatException :
{
double i;
double value;
}
{
value= primary()
(
<PLUS>
i = primary()
{ value+=I;}
|
<MINUS>
i = Primary()
{ value -= i; }
)*
{ return value;}
}
```

3.5. 增加乘法和除法操作

要增加乘除运算是件很简单是事情，我们只需要添加两个产生式

TOKEN:{< TIMES : "*" > }

TOKEN:{< DIVIDE : "/" > }

就像我们增加减法操作时所作的，我们还应该更改Expression的定义，现在它的BNF是：

Expression -> Primary(PLUS Primary | MINUS Primary | TIMES Primary| DIVIDE Primary)*

从纯粹的句法角度看，这个产生式一点问题都没有，但是它并不能正确的表达我们的意思，因为没有考虑运算符的优先级。例如我们输入

2 * 3 + 4 * 5

我们希望得到的是(2*3)+(4*5)但是我们却得到了((2*3)+4)*5！所有我们不得不使用另外一种表达方式：

Expression -> Term (PLUS Term |MINUS Term)*

Term -> Primary(TIMES Primary |DIVIDE Primary)*

这样表达式被分成了一连串的加减运算，加减的元素是Term.

2 * 3 + 4 * 5

我们要做的仅仅是把Expression中所有对Primary 的引用改为对Term 的引用:

```
double Expression() throws NumberFormatException :
{
double i;
double value;
}
{
value = Term()
(
<PLUS>
i = Term()
{ value += i;}
|
<MINUS>
i = Term()
{ value -= i; }
)*
{ return value; }
}
```

Term 的产生式类似：

```
double Term() throws NumberFormatException :
{
double i;
double value;
}
{
```

value = **Primary**()

(

<TIMES>

i = **Primary** ()

{ value *= i;}

|

<DIVIDE>

i = **Primary** ()

{ value /= i; }

)*

{ return value; }

}

3.6. 增加括号，单目操作符和历史记录

现在我们还需要添加少许其他功能使它变成一个真正的有用的计算器，我们需要括号支持，负数支持，还要允许使用美元符$表示上一次表达式计算的值。

更改词法定义变得水到渠成，我们只需增加几个产生式:

TOKEN:{< OPEN_PAR : "(" > }

TOKEN:{< CLOSE_PAR : ")" > }

TOKEN:{< PREVIOUS : "$" > }

我们不需要为取负做任何词法更改，因为我们只需要用到减号(MINUS)而已。

改变之后的Primary的一共有4种可能性：一个数，一个$，一个括号包起来的表达式，或者一个带负号的任意可能

使用BNF表示就是：

Primary -> NUMBER

| PERIVOUS

| OPEN_PAR Expression CLOSE_PAR

| MINUS Primary

这个BNF有两路递归，最后一个是直接递归，倒是第二个是间接递归。在BNF中使用递归是允许的，但是有若干限制。考虑下列表达式：

这将会被理解成：

在解析表达式时，每个小盒子被当成一个Primary。例如

12 * (42 + 19)

将会被理解成

通过这个我们可以看到，Primary这个BNF是如何被递归调用的。

现在我们给出Primary的JavaCC 实现：

```
double Primary() throws NumberFormatException :
{
Token t;
double d;
}
{
t = <NUMBER>
{ return Double.parseDouble( t.image ); }
|
<PREVIOUS>
{ return previousValue; }
  |
<OPEN_PAR> d = Expression() <CLOSE_PAR>
{ return d; }
|
<MINUS> d = Primary()
{ return -d; }
}
```

至此，我们终于完成了我们的计算器。完整的计算器声明见calculator1.jj。当然，我们能做的改进仍然很多，比如添加新的操作符，这些工作就留给各位读者了。

这种计算表达式的方式被称作“直接解释”，也就是说解析器自己把输入解析成数值然后计算出结果。对于简单的表达式来讲，这工作的很好，但是对于复杂的表达式来讲远远不够，比如当我们需要引入某种循环时。考虑下面的表达式：

sum i : 1..10 of i*i

这就是一个典型的数学上的求和运算

这时直接求值就不好使了，因为没有任何数字对于子表达式

i*i

对于这种情况，最好的办法就是让解析器把表达式表示成其他什么形式，比如树，或者某种字节码，在解析完成后再计算。

4. 文本处理

本章的最后一个例子有点不同。加法器和计算器的例子展示的是如何处理人工语言，这些语言将来完全可以拓展为一门完整的编程语言。而本小节的例子将说明JavaCC 在处理无结构的文本时也是非常有用的。

4.1. 一个滤词器

任务是用某些文本替换输入中特定模式的文本。我们根据声明逐字地搜索并替换四字母词，不管它是否合适。

选项和类声明

声明文件的初始部分声明了一个将String 映射到String 的静态方法。

```
/* four-letter-words.jj A simple report writer. */
options {
STATIC = false ;
}

PARSER BEGIN(FLW)
import java.io.Reader ;
import java.io.StringReader ;
class FLW {
static String substitute( String inString ) {
Reader reader = new StringReader( inString ) ;
FLW parser = new FLW( reader ) ;
StringBuffer buffer = new StringBuffer() ;
try {
parser.Start( buffer ) ; }
catch( TokenMgrError e ) {
throw new IllegalStateException() ; }
catch( ParseException e ) {
throw new IllegalStateException() ; }
return buffer.toString() ; }
}
PARSER END(FLW)
```

try 语句的要点是它把ParseException 转变为那些不一定声明了的异常。原因是这个解析器绝对不会扔出ParseException（或者TokenMgrError）；任何一个字符串都是一个合法的输入。（如果Java 编译器支持assert 语句，我们可以用assert false; 语句替代throw 语句。）

词法分析器

词法分析器的声明部分是至关重要的一部分。我们把文件分成三种token：四字母词、多于四个字母的词和少于四个字母的词。我们一步步的来。

四字母词用正则表达式中的量词就很好声明。(x){n} 表示表达式x 恰好重复n 次。

TOKEN : { < FOUR LETTER WORD : (<LETTER>){4} > }

我们已经看到过(x)+ 表示表达式x至少重复一次。类似地，(x)* 表示表达式x重复若干次。这样，五字母词（译注：作者意指含至少五个字母的单词）可以这样声明：

TOKEN : { < FIVE OR MORE LETTER WORD : (<LETTER>){5} (<LETTER>)* > }

我们像这样声明数字[“0”-“9”]。我们可以用不止一种方法写出一个匹配任一单个字母的正则表达式，比如[“a”-“z”, “A”-“Z”]（注：为了简单起见，我们把字母表限定为52个大小写罗马字母。JavaCC 能完美地处理任意Unicode 字符，也就是说它能轻易地处理标音字母和其他字母表的字母）。

TOKEN : { < #LETTER : [”a”-”z”,”A”-”Z”] > }

我们也可以把所有的数字一个个列出来，[“0”-”9”] 就相当于

["0","1","2","3","4","5","6","7","8","9"]

更普遍地，我们能列出各种单独的字符或字符范围，比如

["0"-"9","a"-"z","A"-"Z","'","-"]

它将匹配任一数字、字母、撇号或者连字符。还可以写出字符集的补。比如

~["0"-"9","a"-"z","A"-"Z","'","-"]

匹配任意的非数字字符。一个极端的例子是空集。正则表达式[] 匹配任意空集中的任意字符，就是说，它什么都不匹配，或它匹配的字符序列的集合是空集。正则表达式[] 没什么用，但它的补~[] 能匹配任意不在空集中的单个字符，也就是说，它能匹配任一单字符。这正是我们捕获那些既不是四字母单词也不是长单词时所需要的。

TOKEN : { < OTHER : ~[] > }

最长匹配

考虑输入串”sinister”。我们有好几种办法把它打断成几部分，每一部分都匹配上我们那三种之一的正则表达产生式。例如，我们可以把它看成八个单独的字符。这时，它被断成八个OTHER类型的token。或者把它看成两个OTHER，一个FOUR_LETTER_WORD，以及另外两个OTHER。还可以把它看成包含一个FIVE_OR_MORE_LETTER_WORD，跟着零个、一个、两个、三个OTHER。（一共有17 种可能。）

我们想要的当然是它被当作一个FIVE_OR_MORE_LETTER_WORD匹配上，事实上也是如此，但重点是理解为什么。词法分析器总是试图尽可能多地把剩下的字符塞进下一个产生的token里面。这就是”maximal munch”（最长匹配）规则。假设输入”sinister cats”。三个产生式都能匹配上输入的开头部分：OTHER 捕获一个字符序列”s”； FIVE _LETTER_WORD 捕获头四个字符”sini”；而FIVE_OR_MORE_LETTER_WORD 可以捕获”sinis”，”sinist”，”siniste”，”sinister”中的任意一个。最长的可能匹配是头八个字符，留下” cats”。由于下一个字符不是一个字母，唯一能匹配上的就是OTHER。剩下的序列是”cats”。OTHER 和FIVE _LETTER_WORD 都可以匹配上，但由于最长匹配原则，FIVE _LETTER_WORD 胜出。剩下空串，产生一个EOF。

你可能会想如果两个产生式都能匹配上一个最长的可能匹配时最长匹配原则不就不适用了？在本例中是不会发生这样的情况，因为三个产生式分别匹配长度为一、四、五甚至更多的输入。但考虑一个Java 编程语言的词法分析器。我们写出如下产生式规则。

TOKEN : { < KWINT : ”int” > }

TOKEN : { < IDENTIFIER : (<LETTER> | "") (<LETTER> | <DIGIT> | "")* > }

当余下的输入为"int0 = 0; …" 时，由于最长匹配规则"int0"匹配上INDENTIFIER。而，当输入"int i; …"时，两条规则都能捕捉到最大数目（3个）的字符。在这个例子中，最先出现的规则拥有优先权。所以，"int" 被KWINT 匹配上。

在我们的规范中OTHER 的存在保证了词法解析器总会产生一些token。如果输入不为空，就会产生一个OTHER token（虽然可能其他产生式实际上更适合），而如果剩余输入为空串，将会产生一个EOF。所以我们的词法分析器永远不会产生TokenMgrError 异常。（注：在以后的章节中，我们会看到MORE 关键字的使用。当词法分析规范中用到MORE时，一个能匹配所有情况的产生式，比如我们的OTHER，不足以保证TokenMgrErrors 不会抛出。见JavaCC FAQ 了解更多。）

滤词器的解析

滤词器的解析器规范就直截了当了。这三种token以任意数目出现在任意位置。对FIVE _LETTER_WORD，我们在输出上以四个星号标识。而对于其他token，我们只是简单地回显出来。

```
void Start( StringBuffer buffer ) :

{

Token t ;

}

{

(

<FOUR LETTER WORD>

{ buffer.append("****"); }

|

( t=<FIVE OR MORE LETTER WORD> | t=<OTHER> )

{ buffer.append( t.image ) ; }

)*

<EOF>

}
```

既然解析器接受任意词法分析器产生的token 串，它就不会抛出ParseException 异常。我们的词法分析器和解析器都是"全局的"：它们接受任意的输入串。

5. 总结

我们已经看到JavaCC 允许用正则表达式和BNF 产生式书写出简明的词法分析器和解析器规范。

词法分析器的输入是一串字符——用Java InputStream 对象或者Java Reader 对象表示。词法分析器的输出由JavaCC 确定：一串Token 对象序列。解析器的输入也是固定的，就是词法分析器产生的那串Token 对象序列。词法分析器和解析器之间的关系如下图所示

TBD

解析器的输出并不是由JavaCC 规定的，程序员想让它输出什么它就什么样，只要能用Java 表示出来。输入一般是一些抽象的表示。在加法器和计算器的例子中，输出是一个Java int 或double 类型的数字。不同的输入可能产生相同的数字。编译器中，解析器的输出可能是机器码或汇编码的形式。大多数编译器的解析器会生成输入程序的某种中间代码，这些中间表示日后会被编译器的其他部分用到。不同的应用会有不同的输出形式。例如，输出可能是一个字串，可能是输入串的修改版本，就像我们那个滤词器例子；如果输入是一个配置文件，输出也可能是表示配置的Java 对象；等等等等。

一种特别常见的情况是解析器的输出是一棵完全遵照方法调用的树。这种情况下，有一些额外的工具用来自动增强JavaCC 的输出。这些工具包括JJTree和JTB。

注意词法分析器的工作完全独立于解析器，这一点很重要。它把输入流切割成什么样的token不受解析器期望的影响，而是由它的规范完全确定。

最后，感谢那个烂尾而且喜欢乱吐槽的译者：

https://sites.google.com/site/beariceshome/the-javacc-tutorial

来自：xiaopengpeng > 《数码科技》

上一篇：[转] javacc简介
下一篇：[转] 笔记本电脑和台式电脑在家里没有装网也可以上网（...

发表评论：

您好，请 登录 或者 注册 后再进行评论

社交帐号登录：