JavaCC基本使用介绍

目录

[一、引言 2](#_Toc407994566)

[1.1 JavaCC简介： 2](#_Toc407994567)

[1.2 词法分析器概念 2](#_Toc407994568)

[1.3 语法分析器概念 3](#_Toc407994569)

[1.4 JavaCC采取的语法分析方法 3](#_Toc407994570)

[二、示例使用流程 6](#_Toc407994571)

[2.1加法解析器 6](#_Toc407994572)

[2.2 扩展的加法解析器 10](#_Toc407994573)

[2.3 计算器解析器 12](#_Toc407994574)

2015年1月2日星期五

# 一、引言

## 1.1 JavaCC简介：

JavaCC(Java Compiler Compiler)是一个用JAVA开发的能生成词法和语法分析器的工具。这个工具可以读取上下文无关且有着特殊意义语法的文本输入并把它转换成可以识别该语法的JAVA程序。JavaCC可以在JVM 1.2或更高的版本上使用，它由纯Java代码开发，可以在多种平台上运行。JavaCC还提供JJTree工具来帮助我们建立语法树，JJDoc工具为我们的源文件生成BNF范式(巴科斯-诺尔范式)文档(Html)。

## 1.2 词法分析器概念

词法分析器又称扫描器。词法分析是指将我们编写的文本输入流解析为一个个的记号（Token），分析得到的记号以供后续[语法分析](http://baike.baidu.com/view/487037.htm" \t "_blank)使用。词法分析器的工作是低级别的分析，

就是将一系列字符分成一个个的Token，并标记Token的分类。

例如：

输入：

int main() {

return 0;

}

输出：

“int”, “ ”, “main”, “(”, “)”, “ ”, “{”, “\n”, “\t”, “return”, “ ”, “0”, “;”, “\n”, “}”

工作流程如图1所示：

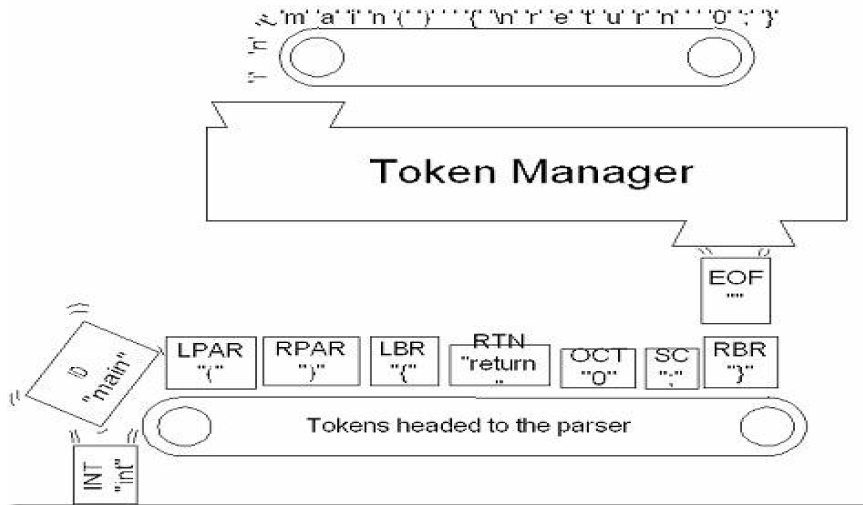


图1：词法分析器工作流程

## 1.3 语法分析器概念

语法分析（Syntacticanalysis or Parsing）是根据某种给定的形式文法对由单词序列构成的[输入文本](http://baike.baidu.com/view/1140843.htm" \t "_blank)进行分析并确定其语法结构的一种过程。[语法](http://baike.baidu.com/view/135635.htm" \t "_blank)[分析](http://baike.baidu.com/view/239473.htm)是编译过程的一个[逻辑](http://baike.baidu.com/view/1838.htm" \t "_blank)阶段。简单来说：[语法](http://baike.baidu.com/view/135635.htm)[分析](http://baike.baidu.com/view/239473.htm)的[任务](http://baike.baidu.com/view/135914.htm" \t "_blank)就是在[词法分析](http://baike.baidu.com/view/487040.htm" \t "_blank)的基础上将[单词](http://baike.baidu.com/view/419292.htm" \t "_blank)序列组合成各类语法短语，判断输入在结构上是否正确。

工作流程如图2所示：

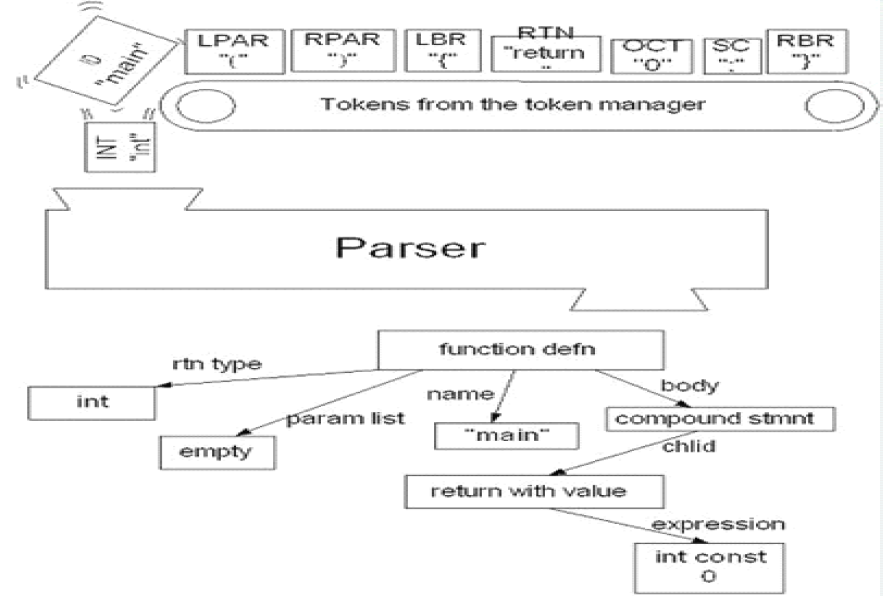


图2：语法分析器工作流程

## 1.4 JavaCC采取的语法分析方法

JavaCC采用的是自顶向下的分析方法，而且没有回溯功能，因此如何解决冲突的问题，是程序员的责任。

关于语法分析方法的分类如图3所示：

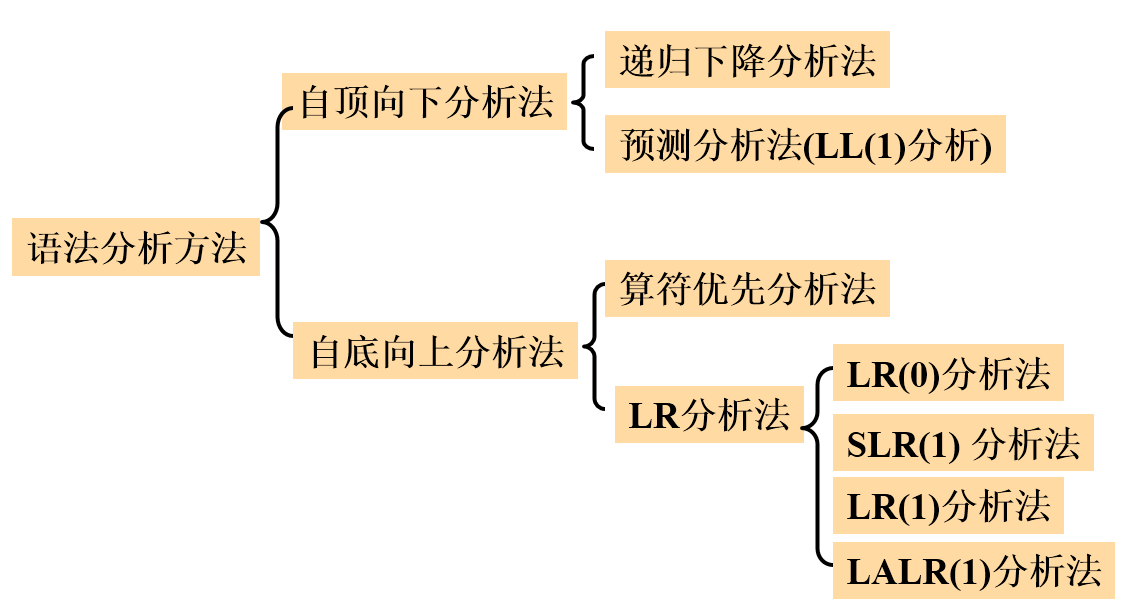


图3：语法分析方法简单分类

**什么是自顶向下的分析方法？什么是自底向上的分析方法？**请看图4，图5示例。



图4：自顶向下的分析方法 图5：自底向上的分析方法

**自顶向下的分析法会遇到的问题：**

（1）存在形如A→αβ1|αβ2的产生式，即有多个候选式的前缀相同（称为**公共左因子**，或左因子），则可能造成虚假匹配，使得在分析过程中可能需要进行大量回溯。

（2）存在**左递归**（文法中有形如A→Aα的产生式），分析过程又使用最左推导，就会使分析过程陷入死循环 。

左递归的类型：

直接左递归：A→A…

间接左递归：A→Bβ，B→Aα 即A→+A…

潜在左递归：A→αAβ，且α→\*ε

**针对上述问题的一些解决方法：**

（1）关于公因子的问题，可以改写算法（提取公因子），也可以通过展望（look ahead）更多符号的方法来解决。例将形如：A→αβ1|αβ2|…|αβn|γ1|γ2|…|γm，改写为：A→αA′|γ1|γ2|…|γm，A′→β1|β2|…|βn。

（2）关于左递归的问题，一般采用经典的修改文法的方法解决。

消除直接左递归：

假定关于A的全部产生式是：A→Aα1|Aα2|…|Aαm|β1|β2|…|βn

消除直接左递归后改写为：

A→β1 A′|β2 A′|…|βn A′

A′→α1 A′|α2 A′|…|αm A′|ε

消除间接左递归：

去除那些从开始符号出发永远无法到达的非终结符号的产生规则，为非终结符号编号，再采用代入法将间接左递归变为直接左递归，从而再消除直接左递归。

**当在JavaCC中碰到冲突时，可以采用下面的方法之一解决问题：**

（1）修改语法，使之成为LL(1)语法。

（2）只要将LOOKAHEAD=k写到Options块中即可。

JavaCC产生的分析器就可以分析LL(K)语法生成的语言。采用第一种方法的好处是效率非常高，易于维护；采用第二种方法的好处是语法更加直观，但是却不易维护。有时候采用第一种方法是无法解决冲突的，第二种方法是唯一的选择。

**什么是LL(1)语法？**

第一个L代表从左向右扫描输入符号串，第二个L代表产生最左推导，1代表在分析过程中执行每一步推导都要向前查看一个输入符号。LL(1)文法既不是二义性的，也不含左递归，对LL(1)文法的所有句子均可进行确定的自顶向下语法分析。

**LL(1)语法的具体定义（不必详究）：**

对文法G的句子进行确定的自顶向下语法分析的充分必要条件是，G的任意两个具有相同左部的产生式A—>α|β 满足下列条件：

（1）如果α、β均不能推导出ε，则 FIRST(α) ∩ FIRST(β) = Φ。

（2）α 和 β 至多有一个能推导出 ε。

（3）如果 β \*═> ε，则 FIRST(α) ∩ FOLLOW(A) = Φ。

将满足上述条件的文法称为LL(1)文法。

**基于LL(1)文法的分析过程（不必详究）**：

给定LL(1)文法，对输入串进行有效的无回溯的自上而下的分析过程是：假设要用非终结符A进行匹配，面临的输入符号为a，A的候选式为A→α1|α2|…|αn

（1）若a∈First(αi)，则指派αi去执行匹配任务；

（2）若a不属于任何一个候选式的终结首符集，则：

①若ε属于某个First(αi)，且a∈Follow(A)，则让A与ε自动匹配；

②否则，a的出现是一种语法错误。

# 二、示例使用流程

JavaCC的输入文档是一个词法和语法的规范文件，其中也包括一些动作的描述，它的后缀是jj或者jjt。一个jj文档包括以下部分：

（1）Options{}部分，这个部分对产生的语法分析器的特性进行说明，例如向前看的token的个数（用来解除冲突）。这部分可以省略的，因为每一个选项都有默认值，当我们没有对某个选项进行说明时，它就采用默认值

（2）分析器类的声明，这个部分指定了分析器类的名字，以及其他类中成员的声明，这个部分是必需有的。

（3）词法分析器，主要定义了SKIP和TOKEN。

（4）语法分析器，语法中的每一个非终结符都对应一个函数，利用函数调用来表示非终结符之间的结构关系。

## 2.1加法解析器

首先建立一个adder.jj文件，然后实现如下四个部分。

**（1）Options{}部分：**

*options {*

*STATIC = false ; //生成非静态类*

*LOOKAHEAD = 2;//向前看2个字母,消除歧义用的*

*DEBUG\_PARSER = true;//以debug形式生成,便于调试*

*}*

**（2）分析器类的声明：**

*PARSER\_BEGIN(Adder)*

*class Adder {*

*static void main( String[] args ) throws ParseException, TokenMgrError*

*{*

*Adder parser = new Adder( System.in ) ;*

*parser.Start() ;*

*}*

*}*

*PARSER\_END(Adder)*

**（3）词法分析器：**

*SKIP : { " " }*

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

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

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

*TOKEN : { < NUMBER : <DIGITS> |*

*<DIGITS> "." <DIGITS> |*

*<DIGITS> "." |*

*"." <DIGITS>> }*

*TOKEN : { < #DIGITS : (["0"-"9"])+ > }//*在词法声明部分，以#开头的token只是在词法分析时使用，不能作为语法分析的输入，也就是说，它相对词法分析是局部的。

**简单释意：**

第一二行表示空格和回车换行是不会传给语法分析器的。

第三行声明了一个Token，名称为PLUS，符号为“+”。

第四行声明了一个Token，名称为NUMBER，符号位一个或多个0-9的数的组合。

如果词法分析器分析的表达式如下：

“123 + 456\n”，则分析为NUMBER, PLUS, NUMBER,EOF

“123 - 456\n”，则报TokenMgrError，因为“-”不是一个有效的Token.

“123 ++ 456\n”，则分析为NUMBER, PLUS, PLUS, NUMBER, EOF，词法分析正确，后面的语法分析将会错误。

**其他：**

JavaCC中的语法表示吸收了UNIX中正规文法的一些记号：

[]： 其中的内容是可选的

+：前面的内容出现一次或多次

\*: 前面的内容出现0次或多次

?： 前面的内容出现0次或一次

|： 前面或后面

()：改变运算的优先级，把其中的内容作为一个整体

**（4）语法分析器**

*void Start() :*

*{}*

*{*

*<NUMBER>*

*(*

*<PLUS>*

*<NUMBER>*

*)\**

*<EOF>*

*}*

**（5）编译adder.jj**

adder.jj文件具体内容如图6所示：



图6：adder.jj文件内容

编译结果如图7所示：

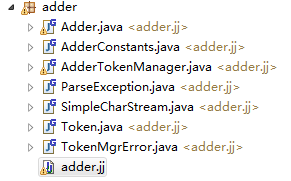


图7：adder.jj编译结果

**SimpleCharStream.java：**用于将一系列字符串传入词法分析器。

**Token.java：**代表词法分析后的一个个Token。Token对象有一个整型域kind，来表示此Token的类型(PLUS,NUMBER, EOF)，有一个String类型的域image，来表示此Token的值。

**AdderConstants.java：**一些常量，如PLUS, NUMBER,EOF等。

**AdderTokenManager.java：**词法分析器。

**Adder.java：**语法分析器。其中的main函数是完全从adder.jj中拷贝的，而start函数是利用javacc由adder.jj描述的规则生成的。

**TokenMgrError.java：**用于在词法分析错误的时候抛出

**ParseException.java：**用于在语法分析错误的时候抛出。

**运行结果如图8（a-c）所示：**

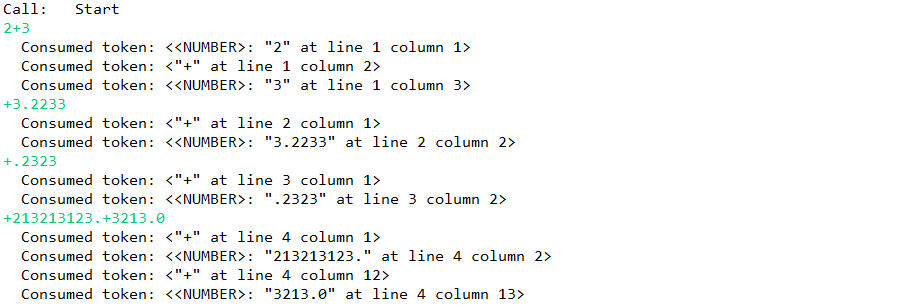


图8（a）：adder.java运行结果—正常情况

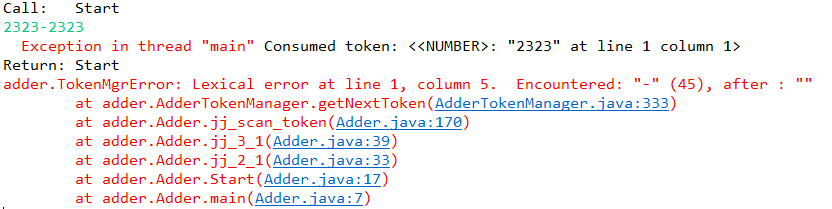


图8（b）：adder.java运行结果—词法错误

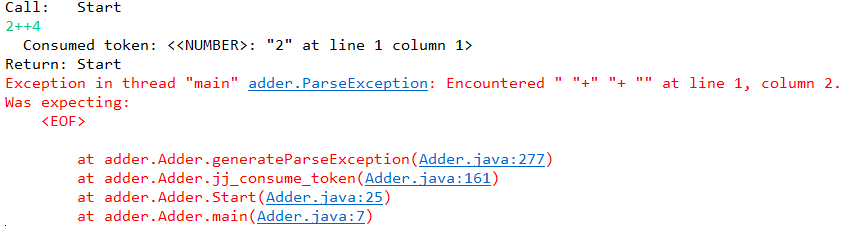


图8（c）：adder.java运行结果—语法错误

## 2.2 扩展的加法解析器

在上面的例子中的start函数中，我们仅仅通过语法分析器来判断输入的语句是否正确。我们可以扩展BNF表达式，加入Java代码，使得经过语法分析后，得到我们想要的结果或者对象。我们将start函数改写为：

*void start() throws NumberFormatException :*

*{*

*Token t ; double i ; double value ;*

*}*

*{*

*t= <NUMBER>*

*{ i = Double.parseDouble( t.image ) ; }*

*{ value = i ; }*

*(*

*<PLUS>*

*t= <NUMBER>*

*{ i = Double.parseDouble( t.image ) ; }*

*{ value += i ; }*

*{ System.out.println(value); }*

*)\**

*}*

**注意：**t.kind表示单词的类别，t.image表示单词值

**运行结果如图9所示：**

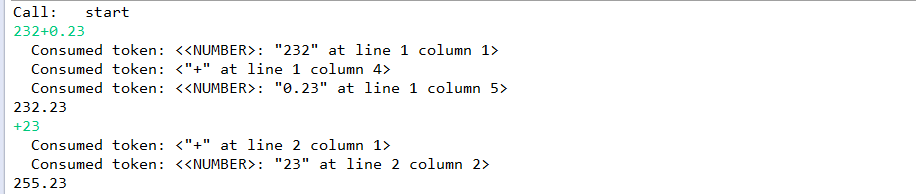


图9：扩展的加法解析器1运行结果

从上面的例子，我们发现，把一个NUMBER取出，并解析为double型这一步是可以共用的，所以可以抽象为一个函数。

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

*void start() throws NumberFormatException :*

*{*

*Token t ; double i ; double value ;*

*}*

*{*

*(*

*i = Primary()*

*{ value = i ; }*

*<PLUS>*

*i = Primary()*

*{ value += i ; }*

*{ System.out.println(value); } <EOL>*

*)\* < EOF>*

*}*

*double Primary() throws NumberFormatException :*

*{*

*Token t ;*

*}*

*{*

*t= <NUMBER>*

*{ return Double.parseDouble( t.image ) ; }*

*}*

运行结果与图10所示：

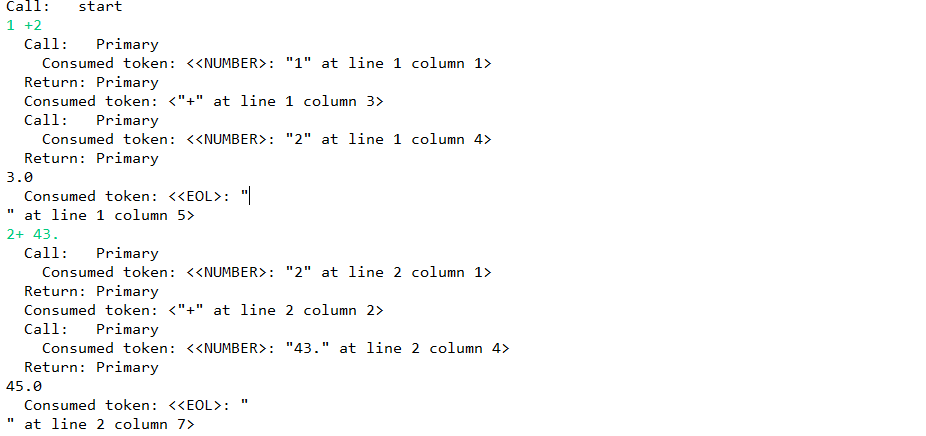


图10：扩展的加法解析器2运行结果

## 2.3 计算器解析器

首先建立一个calculator.jj文件，具体内容如下：

*options {*

*}*

*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 ) ;*

*}*

*}*

*PARSER\_END(Calculator)*

*SKIP : { " " }*

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

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

*TOKEN : { < MINUS : "-" > }*

*TOKEN : { < TIMES : "\*" > }*

*TOKEN : { < DIVIDE : "/" > }*

*TOKEN : { < NUMBER : <DIGITS> |*

*<DIGITS> "." <DIGITS> |*

*<DIGITS> "." |*

*"." <DIGITS>> }*

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

*void start(PrintStream printStream) throws NumberFormatException:*

*{*

*double previousValue;*

*}*

*{*

*(*

*previousValue = Expression() <EOL>*

*{ printStream.println( previousValue ) ; }*

*)\* <EOF>*

*}*

*double Expression() throws NumberFormatException :*

*{*

*double i ;*

*double value ;*

*}*

*{*

*value = Term()*

*(*

*<PLUS>*

*i= Term()*

*{ value += i ; }*

*|*

*<MINUS>*

*i= Term()*

*{ value -= i ; }*

*)\**

*{ return value ; }*

*}*

*double Term() throws NumberFormatException :*

*{*

*double i ;*

*double value ;*

*}*

*{*

*value = Primary()*

*(*

*<TIMES>*

*i = Primary()*

*{ value \*= i ; }*

*|*

*<DIVIDE>*

*i = Primary()*

*{ value /= i ; }*

*)\**

*{ return value ; }*

*}*

*double Primary() throws NumberFormatException :*

*{*

*Token t ;*

*}*

*{*

*t= <NUMBER>*

*{ return Double.parseDouble( t.image ) ; }*

*}*

**运行结果如图11所示：**

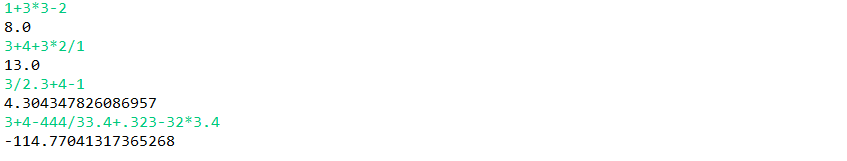


图11：计算器示例运行图