林氏物语.技术乱弹之hand in hand with antlr

版权声明：

本文由林氏原创，遵循GPL许可，你可以自由地对本文进行任何目的的修改、转载、引用和发布，但基于此文所作的任何修改、转载、引用和发布品也需要遵循GPL许可，并要求在开头保留本声明。

联系作者：如果您有什么样的批评与指教，欢迎发email到 workspace.public@gmail.com;

如需获取最新的更新，请在email的subject填写get 010402001发送至 workspace.public@gmail.com;

如需获取林氏物语系列文章的列表，请在email的subject填写get list发送至:workspace.public@gmail.com;

如需获取林氏物语系列文章，请在email的subject填写get all发送至:workspace.public@gmail.com;

由于没有自己的邮件服务器，没有找到带附件自动回复的免费邮件服务器，采用客户端带附件自动回复，所以是非即时性回复，最迟回复时间尽量不超过一周，请勿发送多次。

l 本文编号:010402001

l 所属文集:林氏物语

l 所属类别:技术乱弹

l 所属细类:编译原理

[第一章 前言 3](#_Toc361871682)

[第二章 安装篇: 4](#_Toc361871683)

[第一节 安装前的检查: 4](#_Toc361871684)

[第1小节 是否已经安装了java 1.5 或者更高的版本。 4](#_Toc361871685)

[第2小节 是否已经下载了eclipse 3.x. 4](#_Toc361871686)

[第3小节 建立本教程演示目录。 5](#_Toc361871687)

[第二节 安装过程 6](#_Toc361871688)

[第1小节 下载antworks： 6](#_Toc361871689)

[第2小节 下载antlr的jar包 当前的版本是 v3.3 7](#_Toc361871690)

[第3小节 安装eclipse插件。 8](#_Toc361871691)

[第4小节 确认你的安装是否正确。 11](#_Toc361871692)

[第三章 工具环境介绍篇： 12](#_Toc361871693)

[第一节 先从antlrworks开始体验antlr. 13](#_Toc361871694)

[第1小节 创建语法文件 13](#_Toc361871695)

[第2小节 运行调试语法文件 15](#_Toc361871696)

[第二节 再开始品尝anltrv3ide 18](#_Toc361871697)

[第1小节 新建一个antlr项目. 18](#_Toc361871698)

[第2小节 创建语法文件 19](#_Toc361871699)

[第3小节 进行或者检查插件的设置.. 23](#_Toc361871700)

[第4小节 编写测试的程序 26](#_Toc361871701)

[第三节 Antlrworks工具的各部分使用说明 30](#_Toc361871702)

[第四节 Antlrv3ide插件的各部分使用说明 31](#_Toc361871703)

[第四章 使用篇 33](#_Toc361871704)

[第一节 无责任乱弹 33](#_Toc361871705)

[第二节 关于语法解析的一些基本概念 35](#_Toc361871706)

[第1小节 什么是词法分析，什么是语法分析，这两者有什么不同？ 36](#_Toc361871707)

[第2小节 巴克斯-诺尔范式,产生式,最左推导,最右推导,左递归,右递归 38](#_Toc361871708)

[第3小节 语法解析的自顶向下和自底向上的策略 39](#_Toc361871709)

[第三节 经典的表达式教程解析 44](#_Toc361871710)

[第1小节 界定需求 44](#_Toc361871711)

[第2小节 设计语法 45](#_Toc361871712)

[第3小节 实现语义 51](#_Toc361871713)

[第四节 状态机代码生成框架实例分析 52](#_Toc361871714)

[第五章 引用 52](#_Toc361871715)

# 前言

讨论组地址: <http://groups.google.com/group/antlr_chinese/>

仔细数起来,coding生涯也有不少年头了,从99年到现在,虽然中途离开过几年,但始终没有停留过关注.

在这不长不短的岁月里,在指尖流过的语言也有不少了,从学生年代的c/cpp,object pascal,asm,sql等到工作中用过的vbs/js,java,html,xml, schema,R,erlang和一些即兴学的shell script等，年轻时偏执于语言的表层，略显轻狂,并以之为傲。后来修习图灵机，lambda算子,组合子之类的计算模型，尝试着对语言的本质做了一些深入的思考,才发觉其实自己一直都是井底之蛙。在接触了古老的lex +yacc后,为其高度抽象的简洁与优美所折服.遂不能自拔。

一直都比较喜欢类似lex+yacc之类产生式工具，由于工作主要是java相关的,也接触了antlr.并对其保持比较浓厚的兴趣,时隔几年后重回it时,因为一个小产品的开发需要，对词法语法解析工具选型，选了几个工具，javacc ，antlr ，jflex+CUP，最终还是选择了antlr。

关于工具的选择方面. 一直认为,工欲善其事，必先利其器，但何为利器，如何评价，一直没有清晰的定义，有些东西，明明很好很强大，但是就是无法推广，比如基于强大lambda算子的函数式语言及其平台如lisp ，schema 等。很多年后，才明白，评价一个一个东西，不仅仅看点上的表现，还需要看面上的表现，比如周边的环境支撑等 ，哪怕点上的表现稍逊一筹，只要面上的表现优异，也能弥补其缺陷。

一个工具，被选择的首当其冲的标准当属于其生产能力，评价选择一个工具的生产能力，除了要看它的主要功能外，还要看它对你的生产流程的覆盖程度。覆盖的越全，也就认为其产出效率越高。凭借eclipse这个强大的工具平台。Antlr 做到了从前端语法设计到调试到集成测试的无缝覆盖，这确实是我选择其的一个关键决定因素。

贸然的使用一个不成熟的工具，或者没有实践过的新解决方案，不管它在别人那里是如何成功的，你都需要支付一定风险的。利器从来都是双刃的，伤人亦能伤己，用的好，强虏灰飞烟灭，用的不好，伤痕累累，欲罢不能。诚然，形式化的产生式编程方式如 antlr的确有其强大且简洁优雅的一面，但凡事有利必有弊，一旦遇到问题，其背后隐藏的抽象的复杂度足够让一个项目或产品栽倒在时间压力面前。这也是这个工具鲜被采用的原因。

开发结束后,对使用这个工具所遇到的问题以及一些解决方法，作了一些总结和整理。前车之鉴，后事之师，不管是为人还是为己。于是萌生了写个文档的想法，说是写，实际上是不过是将所找的零碎资料拼装而已。在找资料的过程中发现，鲜有中文教程，只有一本v2.75的中文版规范。 虽然官方的英文文档不算少，但对于在非英语母语的我们来说，对英文信息的敏感度较低，从那堆信息里，寻找一条合适的学习路径略显艰难。于是就学习吾友陈民的做法，尽力写一些教程像他以前帮助我一样帮助其他人。另外在这里也特别感谢甘草同志(healer.kx.yu@gmail.com)，加入这个阵营…

这篇教程定位为帮助零基础的初学者入门到掌握编译前端，为熟悉antlr的人，提供一些问题备忘录，以及共享一些对antlr本身设计的看法。希望能对您有所帮助。

# 安装篇:

选择antlr确实有很大的优势。 有官方自带的antlrworks ，有 民间的antlrv3ide ，和 netbeans什么的插件等等。还有一些收费的eclipse插件等，具体就不去细列了，这个教程主要集中在antlrworks 和 antlrv3ide+eclipse等两个 ide 上面。

通过本篇的学习，你应该能搭建好一个可以实战的平台。

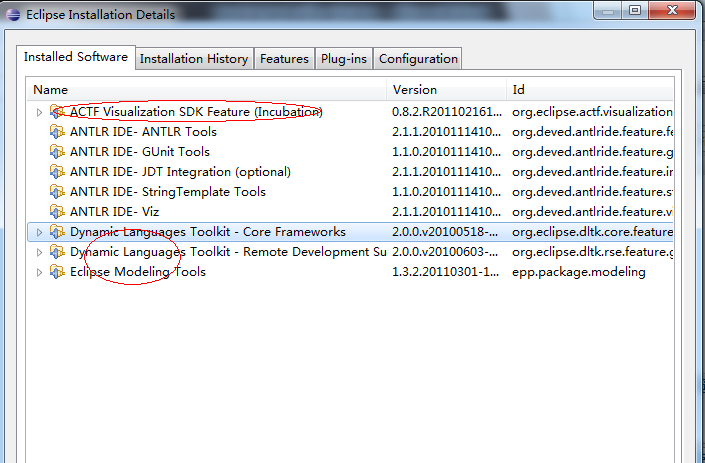
## 安装前的检查:

### 是否已经安装了java 1.5 或者更高的版本。

确认方法 java –version 和javac –version，如果不能工作，或者版本不对，请检查环境变量的JAVA\_HOME

### 是否已经下载了eclipse 3.x.

教程使用的是 eclipse 3.6.2(helios) Eclipse Modeling Tools 那个版本，最好保持和教程一致的。不能确定antlrv3ide是否适用了gef 或者gmf 但建议如果你使用其它的类型的eclipse ，将modeling 模块升级到最新版本。同时还需要安装一个dynamic languages toolkit core frameworks.这个是antlrv3ide依赖的插件.本教程所用的插件如下:（画圈圈的是没有安装插件前就要存在的。如果没有，你需要在 help->install new software 中选择 helios这个更新源进行更新）



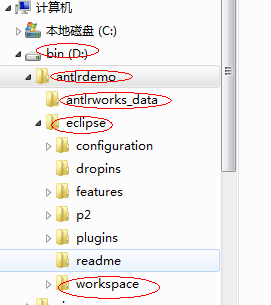
### 建立本教程演示目录。

建立一个目录(d:\antlrdemo),用作本教程的演示目录，这个目录可以更换到其他的地方，但是请注意，请不要带有中文或者其他非英文字母。

在anltrdemo下面建立一个antlrworks\_data目录。

将eclipse拷贝至 antlrdemo目录下。

并在eclipse目录下建立 一个workspace的目录，并在启动eclipse的时候，将工作空间设置为这个目录的工作空间。这样做的目的便于管理资料。



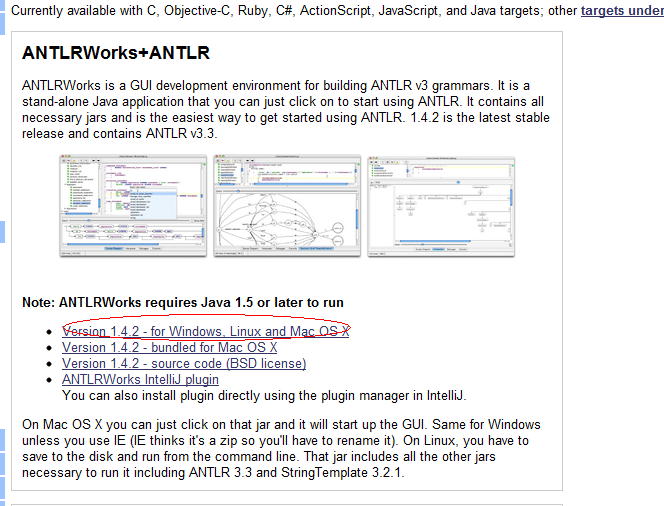
就绪之后，来开始我们的安装过程。。。

## 安装过程

### 下载antworks：

下载 antlrworks 这个集成调试工具，

在官方的网址 <http://www.antlr.org/download.html> 上面

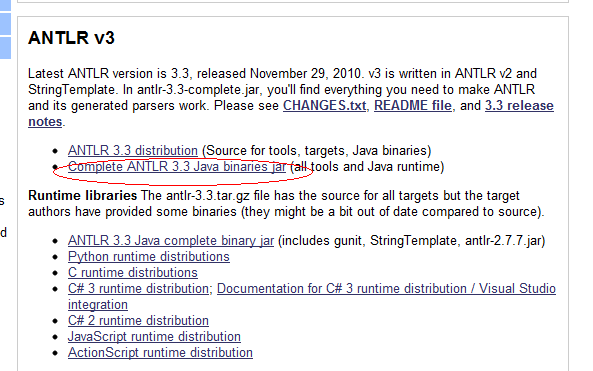


具体的包地址: http://www.antlr.org/download/antlrworks-1.4.2.jar

下载之后扔到 antlrdemo目录下。

### 下载antlr的jar包 当前的版本是 v3.3

在官方的网址 <http://www.antlr.org/download.html> 上面

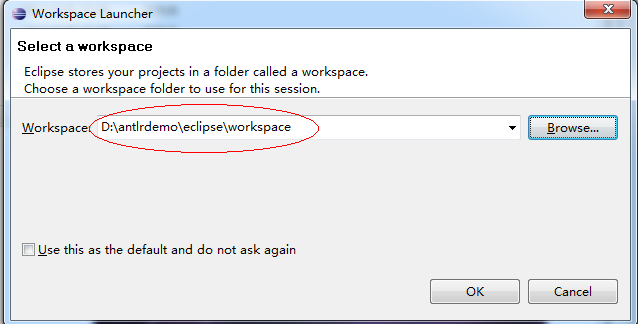


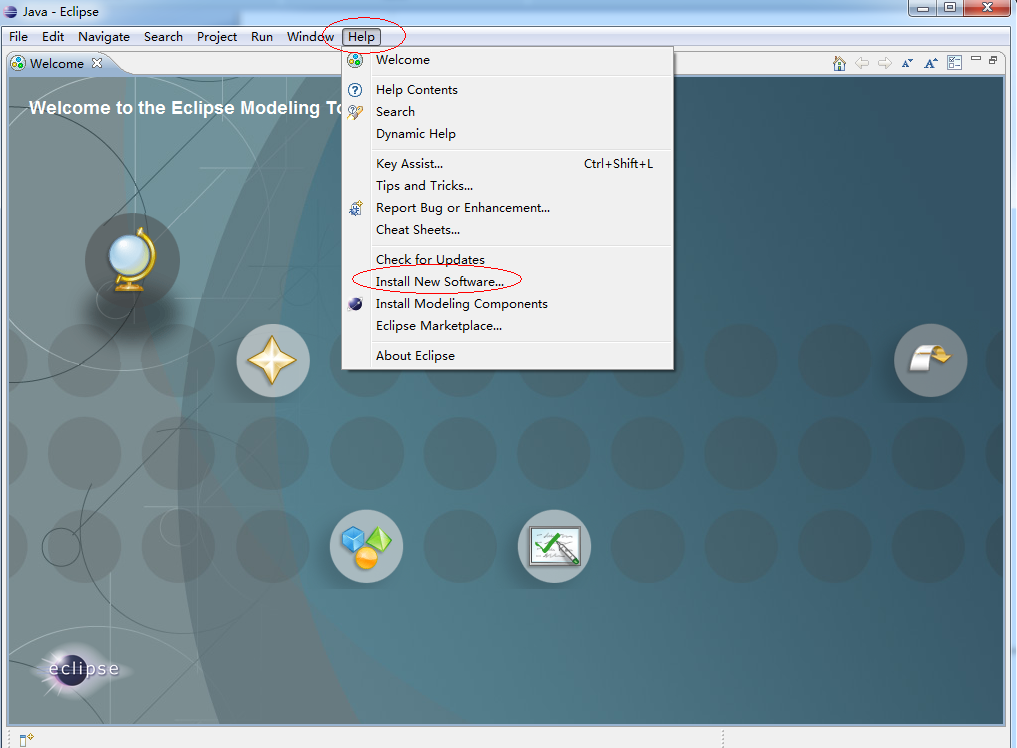
具体的jar包位置:http://www.antlr.org/download/antlr-3.3-complete.jar

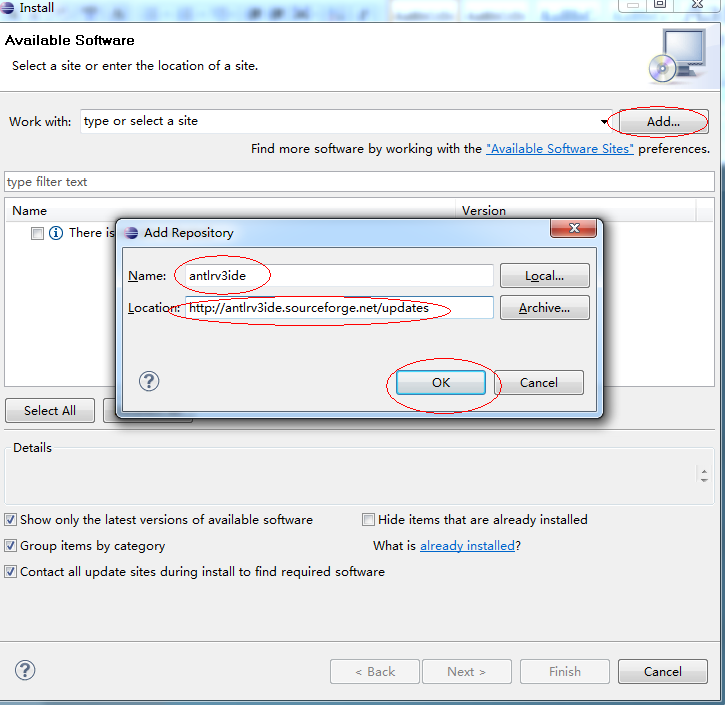
下载之后扔到 antlrdemo目录下。

### 安装eclipse插件。

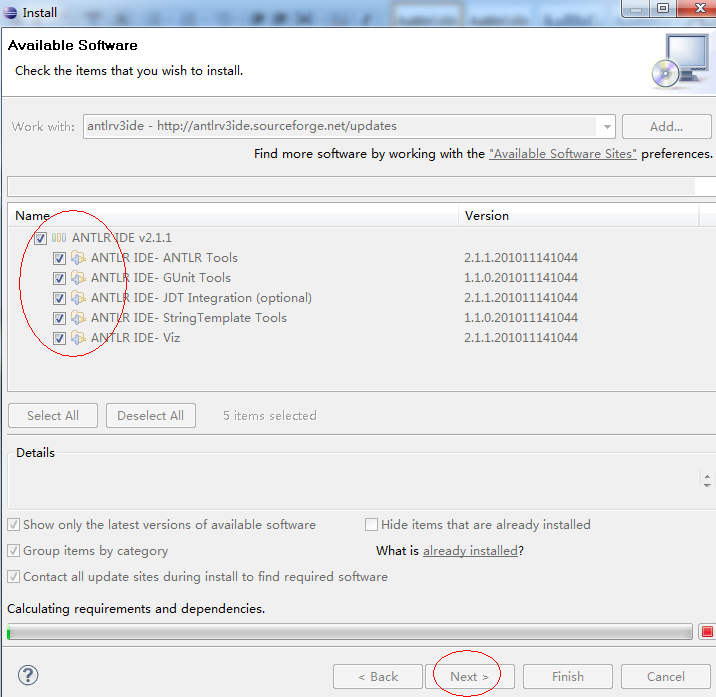
运行eclipse 3.6.2 ，如果是第一次，请设置 workspace的位置为 antlrdemo下面的eclipse的workspace







Ps: antlrv3ide 的插件updaesuite 的地址: <http://antlrv3ide.sourceforge.net/updates>

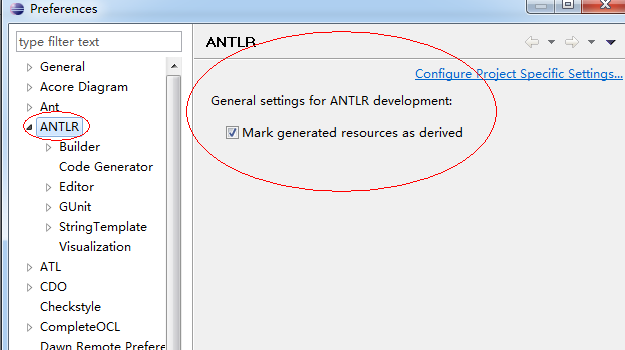


毫无悬念地全选，同意什么的，就安装完了。

### 确认你的安装是否正确。

确认antlrworks :双击antlrworks ，如果看到可视界面，就ok了。

确认antlrv3ide:打开eclipse ，安装插件成功后， 打开window🡪perfermance 可以看到



看到这个，差不多就安装正确了. 到目前为止，antlrdemo目录有两个目录：eclipse 和antlrworks\_data和两个jar文件: antlrworks-1.4.2.jar 和antlr-3.3-complete.jar;

# 工具环境介绍篇：

虽然一般的开发里面用antlrv3ide就足够了，但是，个人觉得antlrworks在调试语法的时候比较清爽。而且antlrv3ide不能显示抽象语法树，但antlrworks 可以，所以，一般都是双管齐下。前期的语法规则分析和调试之类的，就用antlrworks ，涉及到语义动作相关，需要嵌入代码的时候，就切入antlrv3ide 。

比较列表

|  |  |  |
| --- | --- | --- |
| 差异点 | Antlrworks | Antlrv3ide |
| 运行形式 | 独立运行的java程序 | 需要依赖eclipse |
| 使用配置复杂性 | 傻瓜式，在界面内全部展现出来 | 分散隐蔽，对不熟悉的人寻找有一定的难度。 |
| 图形化界面支持 | 全图形化 | 基本图形化，但对抽象语法树支持不好. |
| 环境支持 | 几乎不能融入其它环境，理论上可以通过设置classpath等来设置引用的jar或者class等，但很难用，靠使用者个人手工编译java的能力来保证。 | 可以和在eclipse上面开发的项目无缝集成在一起。 |

作为初级阶段，这里的使用篇就以经典的表达式解析来展现。

就绪后，就开始我们的使用之旅吧…

通过本篇的学习，你应该可以运行示例的demo文件，并了解工具的各部分的构成。

## 先从antlrworks开始体验antlr.

### 创建语法文件

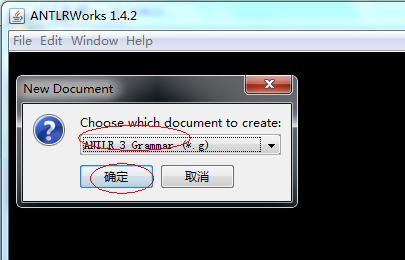
#### 可选一:下载语法文件.

下载 ( <http://www.antlr.org/works/help/tutorial/content/Expr.g> )到antlrworks\_data

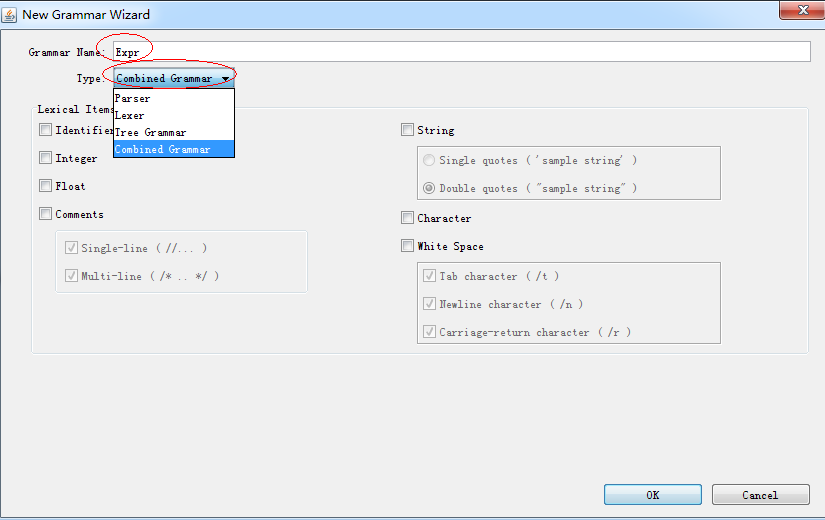
(注意:文件名不要改变，仍然为Expr.g)

然后用antlrworks 打开这个Expr.g文件: file -> open ,选择antlrworks\_data目录的Expr.g

#### 可选二:手工输入语法文件。



打开antlrworks， 新建一个 antlr\_3 Grammer \*.g 文件



语法名字叫Expr

类型名字叫combined grammar

保存在 antlrworks\_data目录下。

在将下面内容拷贝到编辑器。

//拷贝开始

grammar Expr;

@header {

package test;

import java.util.HashMap;

}

@lexer::header {package test;}

@members {

/\*\* Map variable name to Integer object holding value \*/

HashMap memory = new HashMap();

}

prog: stat+ ;

stat: expr NEWLINE {System.out.println($expr.value);}

| ID '=' expr NEWLINE

{memory.put($ID.text, new Integer($expr.value));}

| NEWLINE

;

expr returns [int value]

: e=multExpr {$value = $e.value;}

( '+' e=multExpr {$value += $e.value;}

| '-' e=multExpr {$value -= $e.value;}

)\*

;

multExpr returns [int value]

: e=atom {$value = $e.value;} ('\*' e=atom {$value \*= $e.value;})\*

;

atom returns [int value]

: INT {$value = Integer.parseInt($INT.text);}

| ID

{

Integer v = (Integer)memory.get($ID.text);

if ( v!=null ) $value = v.intValue();

else System.err.println("undefined variable "+$ID.text);

}

| '(' e=expr ')' {$value = $e.value;}

;

ID : ('a'..'z'|'A'..'Z')+ ;

INT : '0'..'9'+ ;

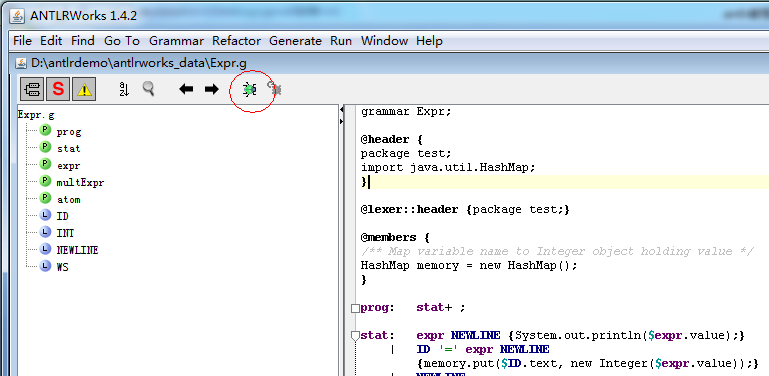
NEWLINE:'\r'? '\n' ;

WS : (' '|'\t')+ {skip();} ;

//拷贝结束

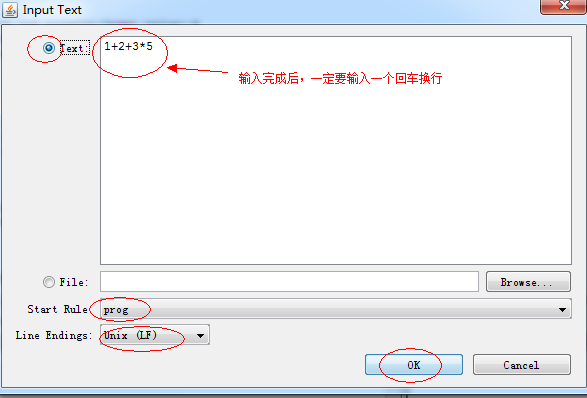
### 运行调试语法文件

点击 图标中的小甲虫。

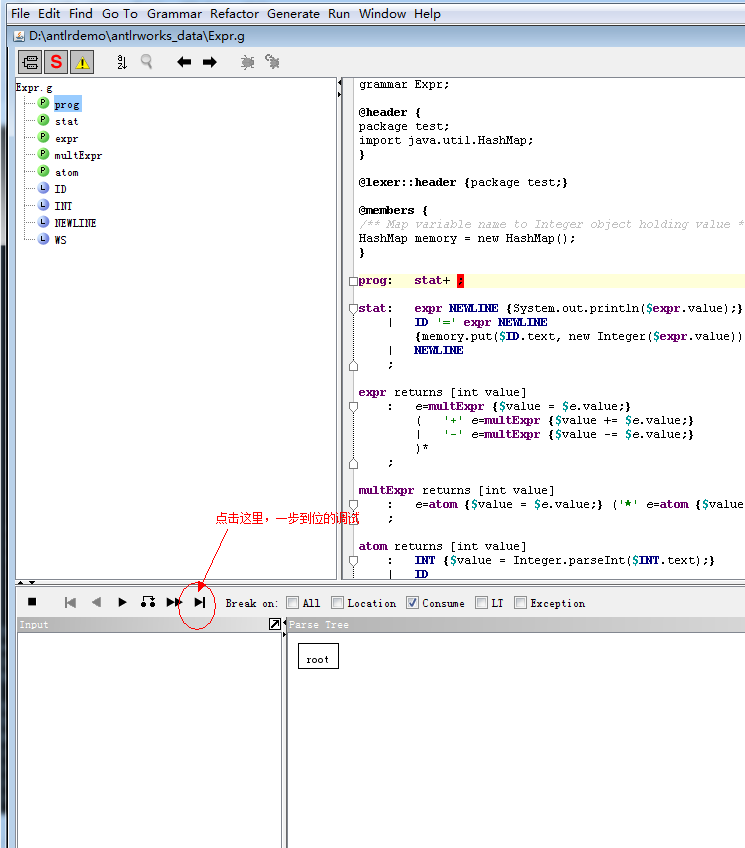


在弹出来的调试界面中，选择 text

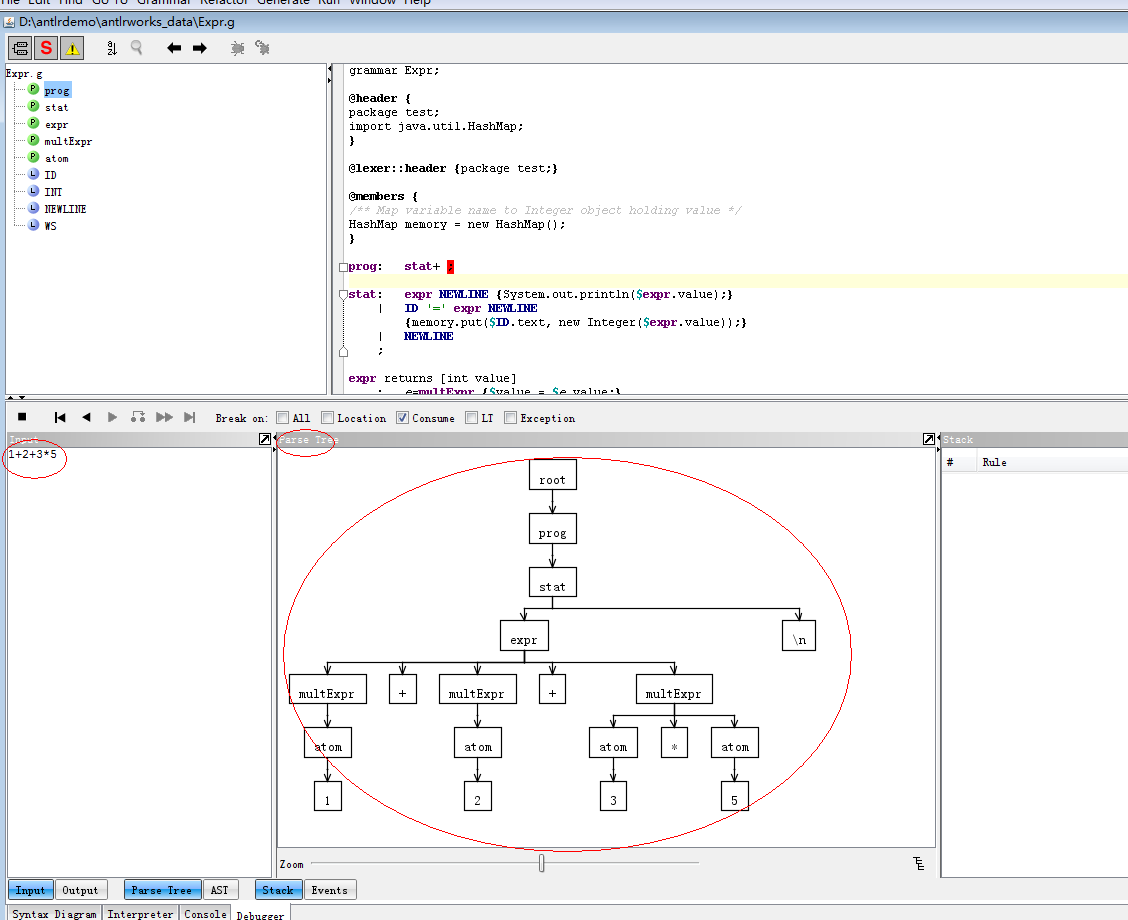
输入 1+2+3\*5 之后，还需要输入一个回车换行。



在就绪的界面中点击 “执行到底”的那个按钮，英文提示:”go to end”



之后将会在output窗口看到被识别出来的token流，和具体语法分析树的结果。



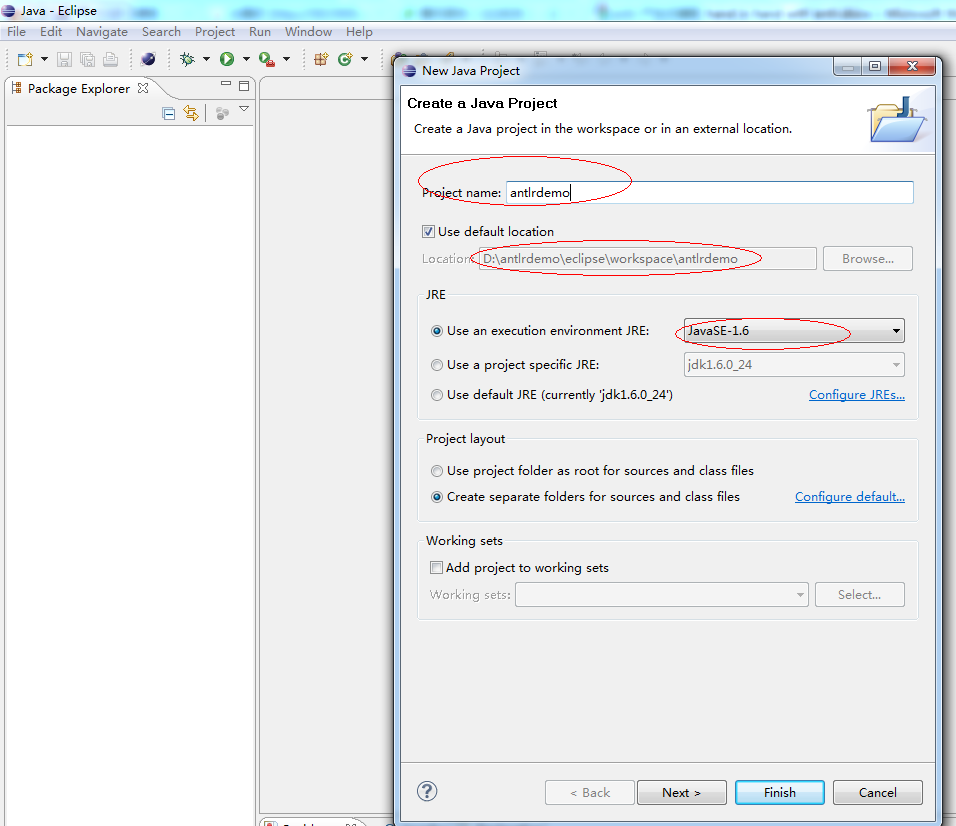
到此，就是简单使用anltrworks用语法文件来解析输入数据的过程了。

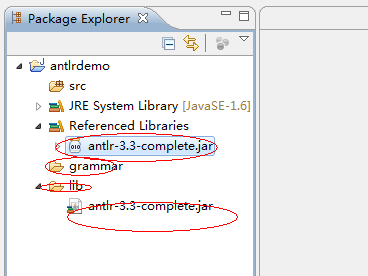
现在到antlrv3ide 登场了，这个会稍微有点儿麻烦。

## 再开始品尝anltrv3ide

### 新建一个antlr项目.

新建一个名叫antlrdemo的java项目,增加两个目录叫grammar 和lib，并将antlr.3.3-complete.jar拷贝到lib，在jar包上右键，build Path-> add to build path





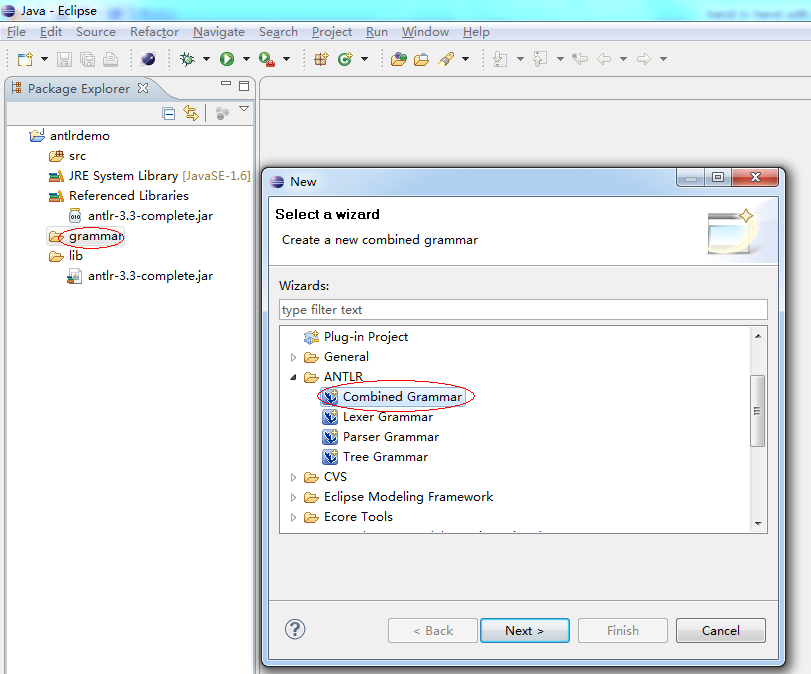
### 创建语法文件

#### 选择一:和antlrworks中的下载一样.

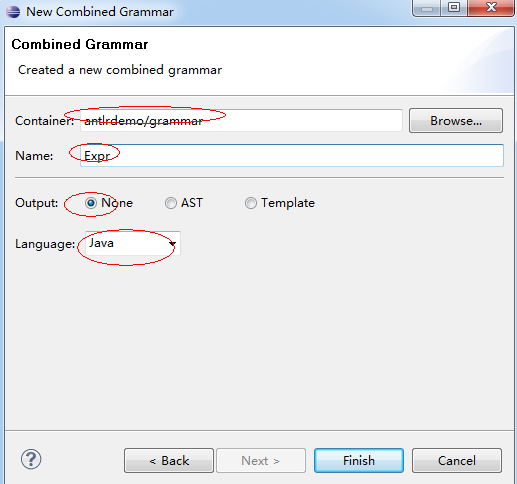
如法炮制，将刚才在antlrworks中的Expr.g文件拷贝到grammar文件夹中.

#### 选择二:手工创建

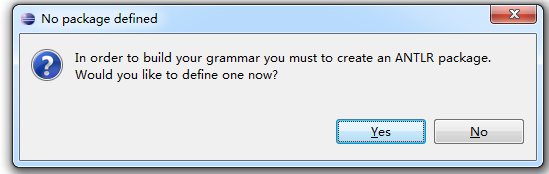
在grammar上面右键 ， new -> other,选择 antlr中的 combined grammar



输入 语法文件名Expr

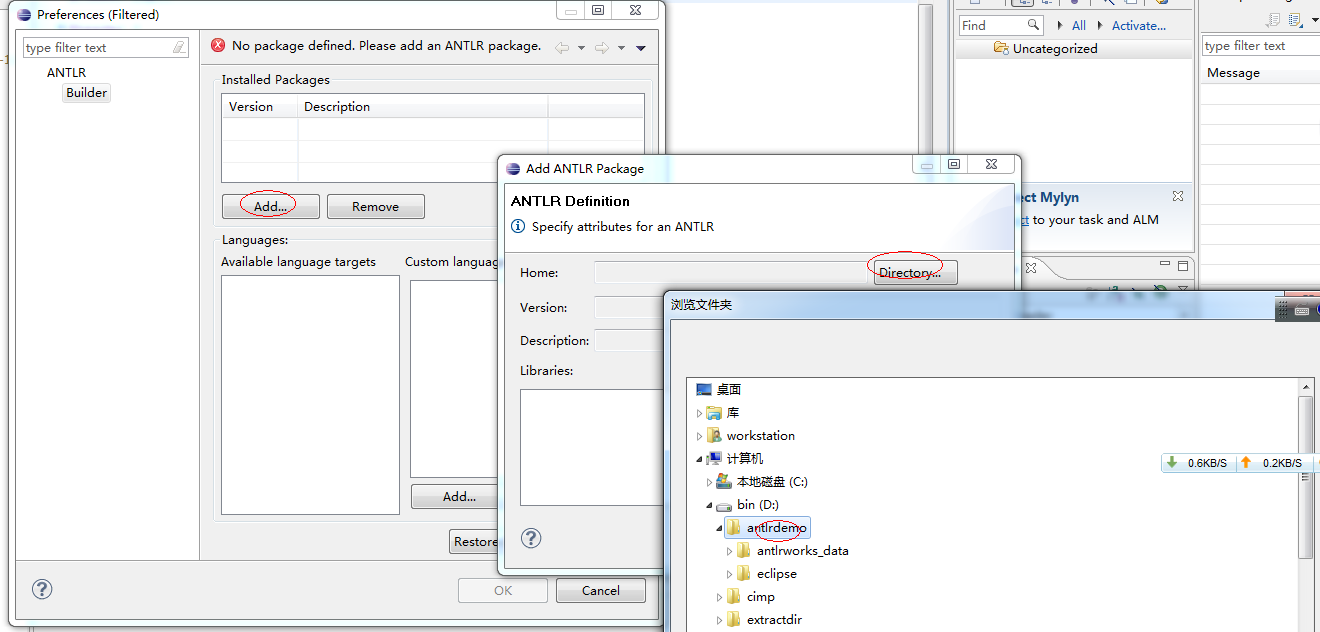


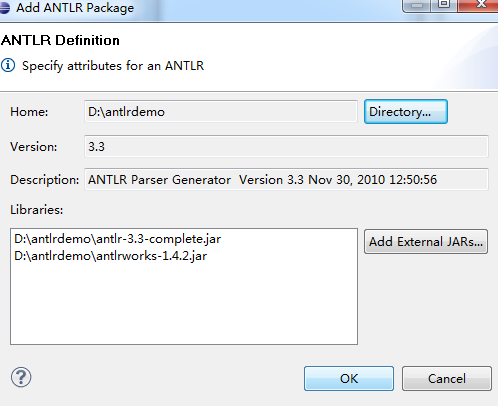
确定后会有一个提示框

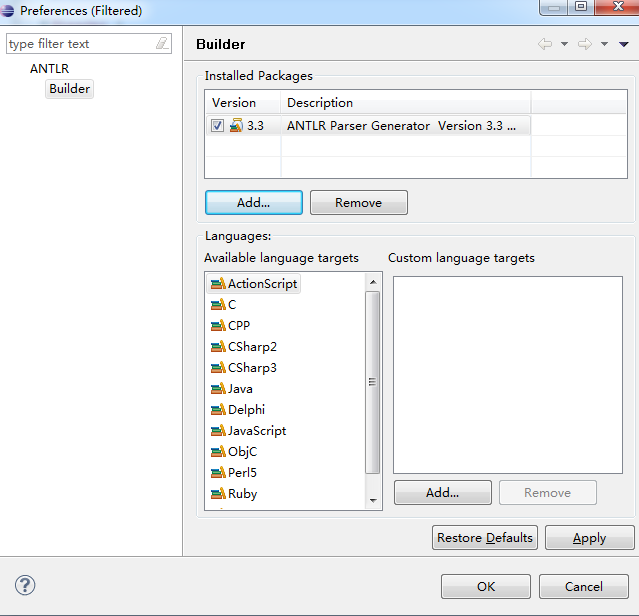


选择yes

此举是配置antlr的jar包位置的，必做。



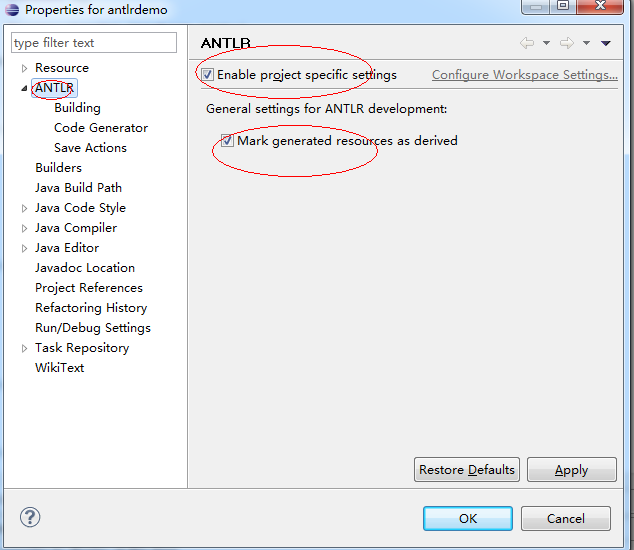




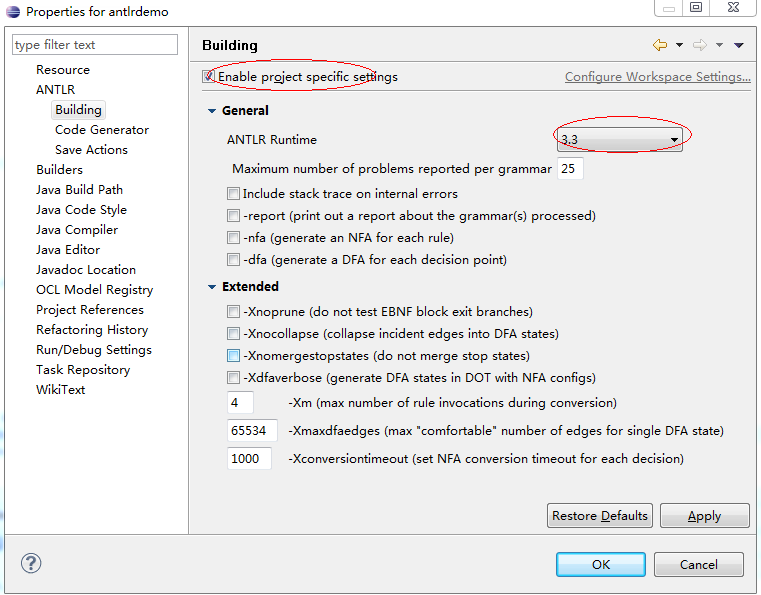
设置好后，将例子中的Expr.g文件的内容(参照antlrworks中关于Expr.g内容)输入到文件中。

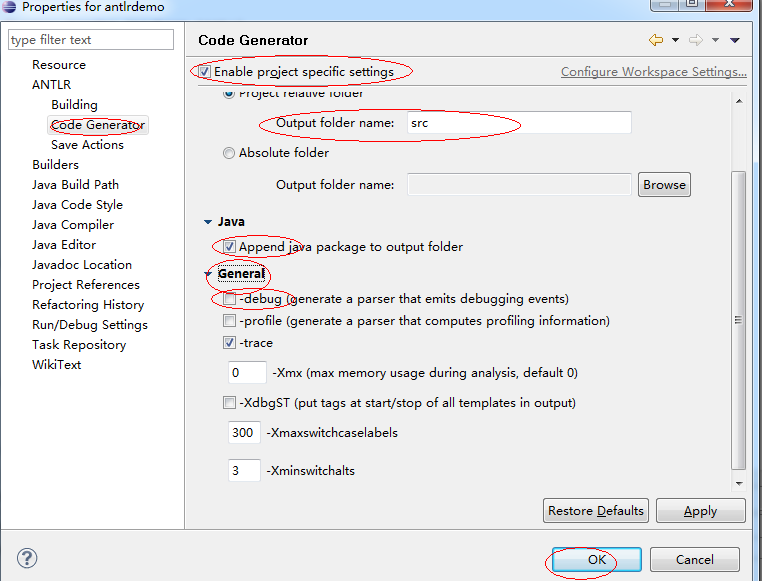
### 进行或者检查插件的设置..

在antlrdemo上右键，选择properties ，在弹出来的对话框中，左边选择 antlr，在右边钩上enable xxx，

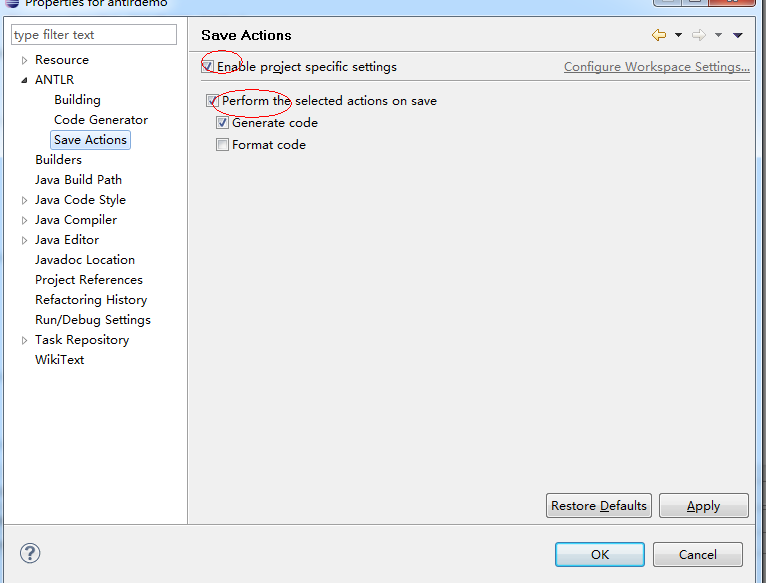


子节点的各个选项都钩上，enable



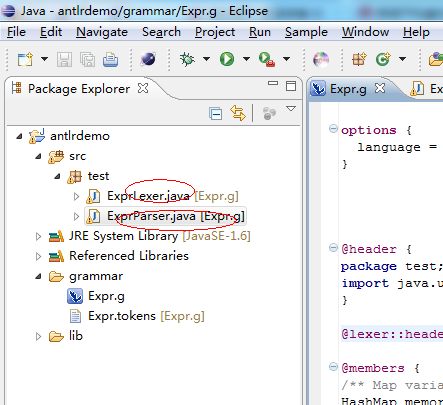


请注意，此时不要选择 debug,(原因呢，以后再告诉你…)。



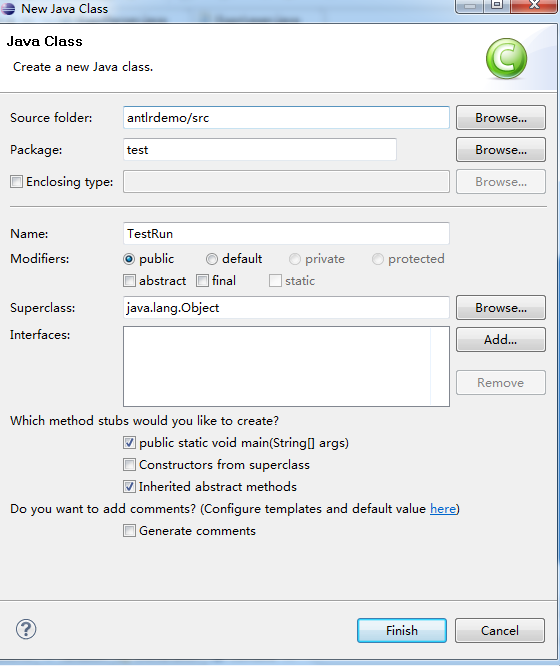
### 编写测试的程序

如果设置正确, 则会在test目录下顺利生成两个java的源码文件 ExprLexer.java 和 ExprParser.java。



如果看不到这两个文件，那么随便改动一下Expr.g文件 ，如果配置全都正确的话，保存即可自动生成这两个源代码文件，如果配置不正确，那请google之。

建立一个TestRun.java的测试程序,(可以比对一下这个文件和D:\antlrdemo\antlrworks\_data\output\ \_\_Test\_\_.java的异同，这以让你一窥antlr的调试功能的实现)。



TestRun.java的文件内容:

**package** test;

**import** java.io.\*;

**import** org.antlr.runtime.\*;

**public** **class** TestRun {

**public** **static** **void** main(String[] args) {

ExprLexer lex = **new** ExprLexer(**new** ANTLRStringStream("1+2+3\n"));

CommonTokenStream tokens = **new** CommonTokenStream(lex);

ExprParser g = **new** ExprParser(tokens);

**try** {

g.prog();

} **catch** (Exception e) {

e.printStackTrace();

}

}

}

保存，后运行，会输出以下日志

enter INT 1 line=1:0

exit INT + line=1:1

enter prog [@0,0:0='1',<6>,1:0]

enter stat [@0,0:0='1',<6>,1:0]

enter expr [@0,0:0='1',<6>,1:0]

enter multExpr [@0,0:0='1',<6>,1:0]

enter atom [@0,0:0='1',<6>,1:0]

enter T\_\_9 + line=1:1

exit T\_\_9 2 line=1:2

exit atom [@1,1:1='+',<9>,1:1]

exit multExpr [@1,1:1='+',<9>,1:1]

enter INT 2 line=1:2

exit INT

line=1:3

enter multExpr [@2,2:2='2',<6>,1:2]

enter atom [@2,2:2='2',<6>,1:2]

enter NEWLINE

line=1:3

exit NEWLINE ? line=2:0

exit atom [@3,3:3='\n',<4>,1:3]

exit multExpr [@3,3:3='\n',<4>,1:3]

exit expr [@3,3:3='\n',<4>,1:3]

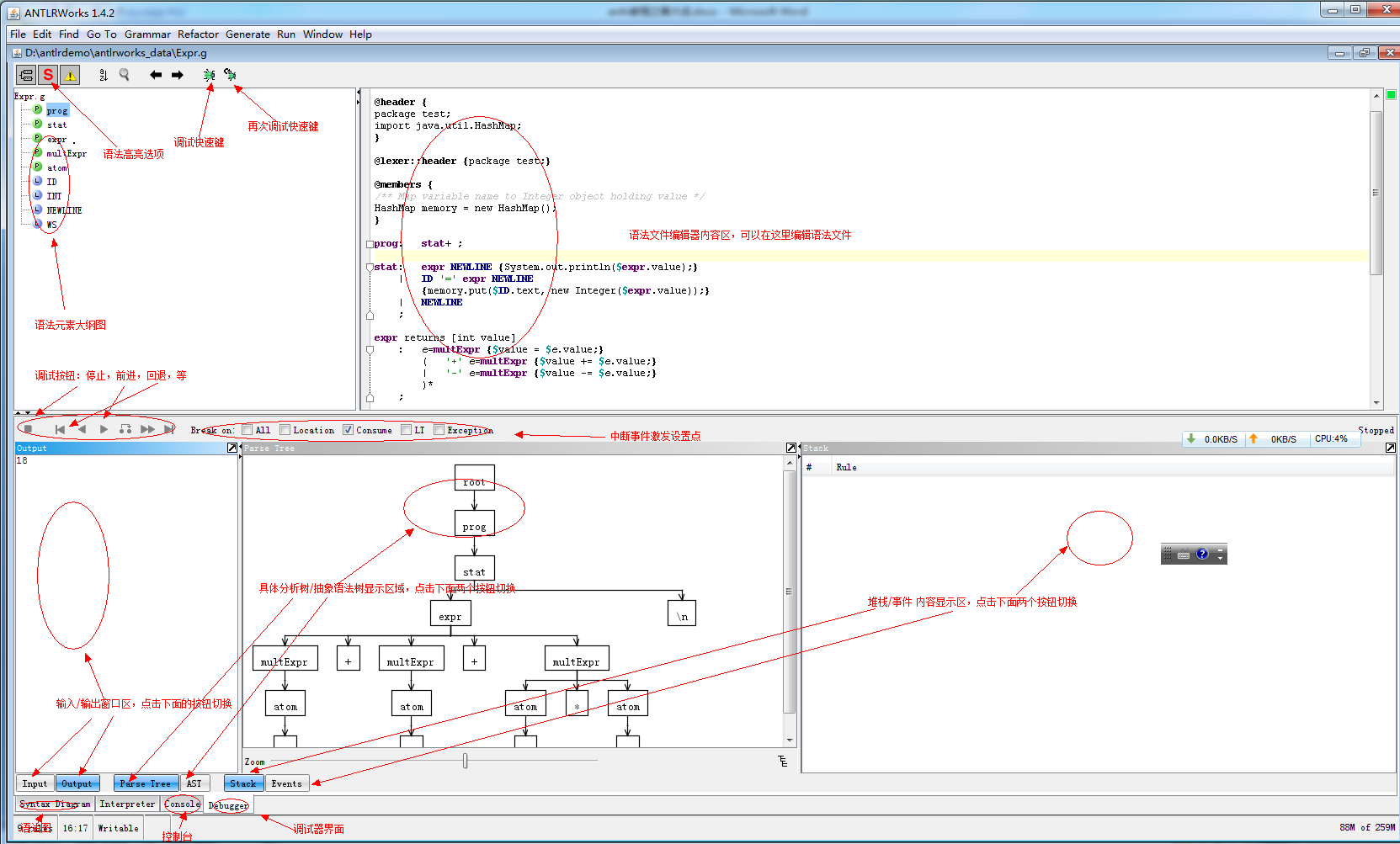
3

exit stat [@4,4:4='<EOF>',<-1>,2:0]

exit prog [@4,4:4='<EOF>',<-1>,2:0]

这些日志信息代表的含义将会在使用进阶里面解释，需要关注的是倒数第三行 数字3，这个是输出的结构，你可以通过配置选项关闭其他信息的输出，只保留结果的输出，这些将会在antlr的配置项中给以解释。

## Antlrworks工具的各部分使用说明



**语法元素大纲图:** 将语法文件的语法元素以大纲的方式列在左边。上面还有一个左右箭头按钮，用来在语法元素间切换的。

**语法文件编辑器：**编辑语法文件，与普通的编辑器无异，多了一个输入时，会自动对齐的功能。

**语法高亮按钮：**控制语法编辑器是否高亮显示语法关键字。

**调试快速按钮:**这个可以在菜单中找到，但经常用，就列在了界面上。

**调试按钮:**调试时候，进行单步前进，单步后退，回到最开始位置，或者执行到最后位置。

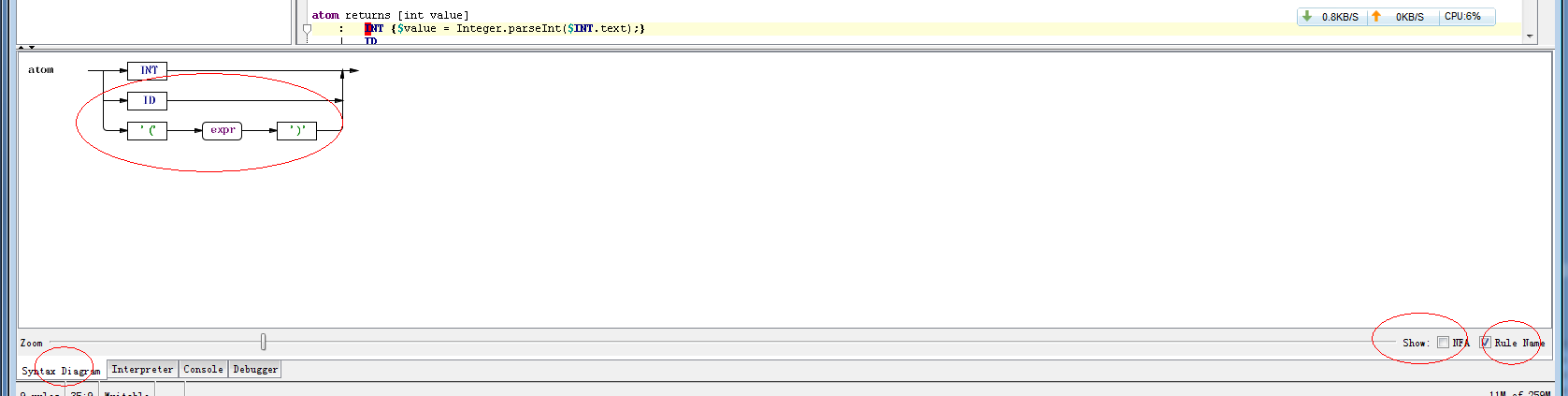
**中断事件激发点：**设置什么时候进行发出调试中断陷阱信号给调试器。默认设置为token消耗时，也就是当一个词法被识别出来的时候，形成一个token被语法调用消耗时，触发调试中断陷阱。

**输入/输出窗口:**输入输出窗口共用一块显示区域，不能同时显示，任意一个时间，最多只能显示其中的一个。输入输出窗口的作用主要是关于输入流的显示和被辨认出来的token流的输出，由于是ll(k)算法 ( ll(k) 的意思是 scans input from left-to-right, produces a leftmost derivation top-down, k symbols of lookahead .而lr(k)的意思是a grammar in which purser scans the input from left to right and generate the reverse rightmost derivation, and it can take a decision about reduction by looking next k symbol only. That is why; it is called LR (k) grammar.),所以这两者几乎都是同步的，但由于可能会存在一些token被skip或者hide了，所以，output的窗口内容会少于input窗口内容。

**分析树和抽象语法树窗口:** 以树状的形式来展现解析的结果，分析树贴近源语法，是源语法的一个直接映射，抽象语法语法树是对源语法进行抽象后的一个体现。比如，一般整数和实数，在分析树中会对应不同的类型，但是在抽象语法树中，可能会被抽象成操作数这样的统一类型。

**堆栈和事件窗口:**堆栈窗口展现语法规则被访问的深度关系。就像函数的调用堆栈一样。事件窗口展现每一步发生的事件，比如进入那一个规则，消耗那一些token等。

语法图窗口:默认以bnf图方式展示语法规则。Show nfa可以选择是否显示非确定型状态机。

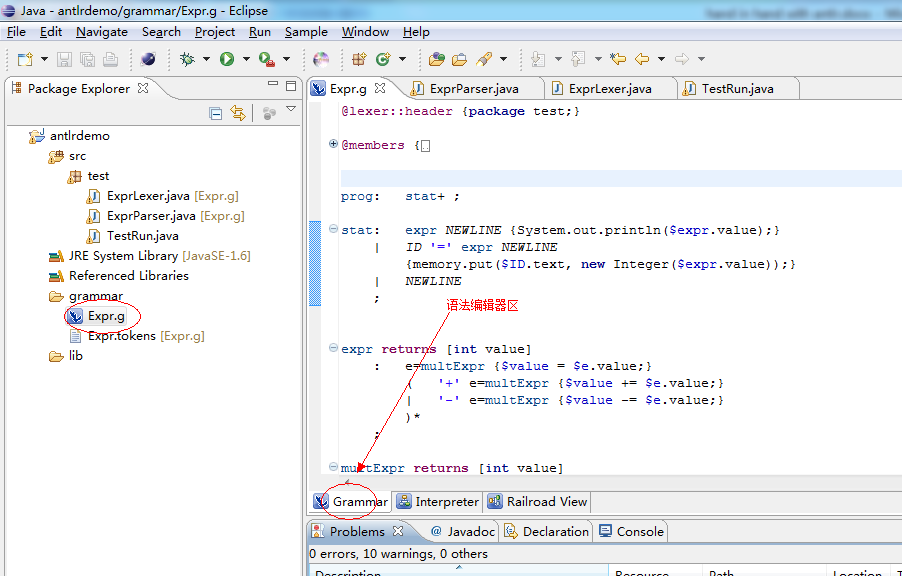


解释器窗口:可以单独调试某一条规则。

## Antlrv3ide插件的各部分使用说明

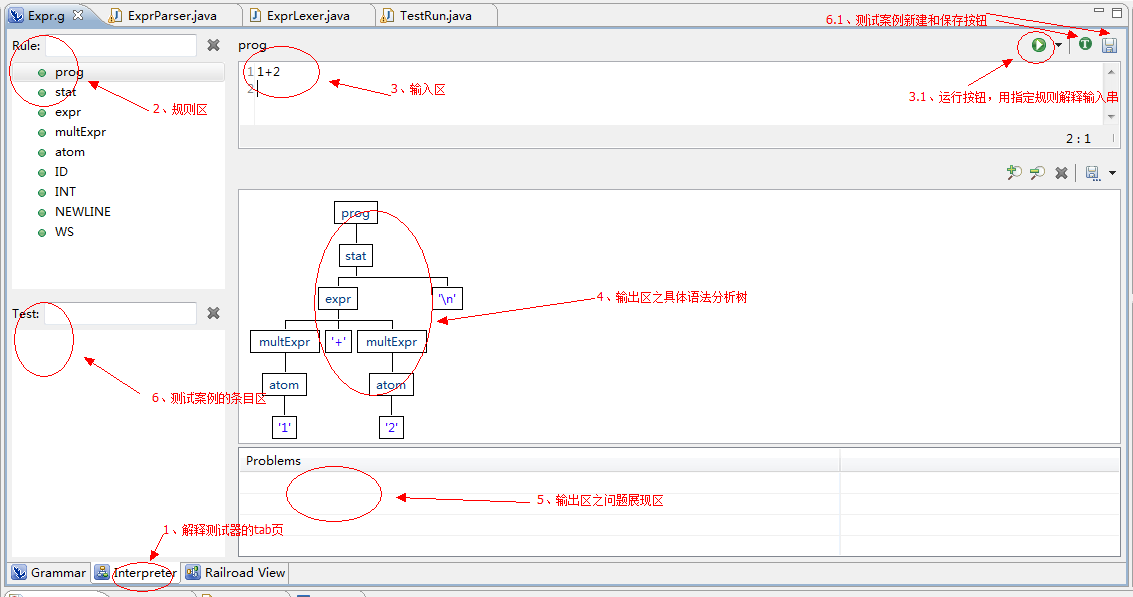
双击Expr.g文件，即可进入antlrv3ide的插件关联的视图。下面是各组成部分的说明。

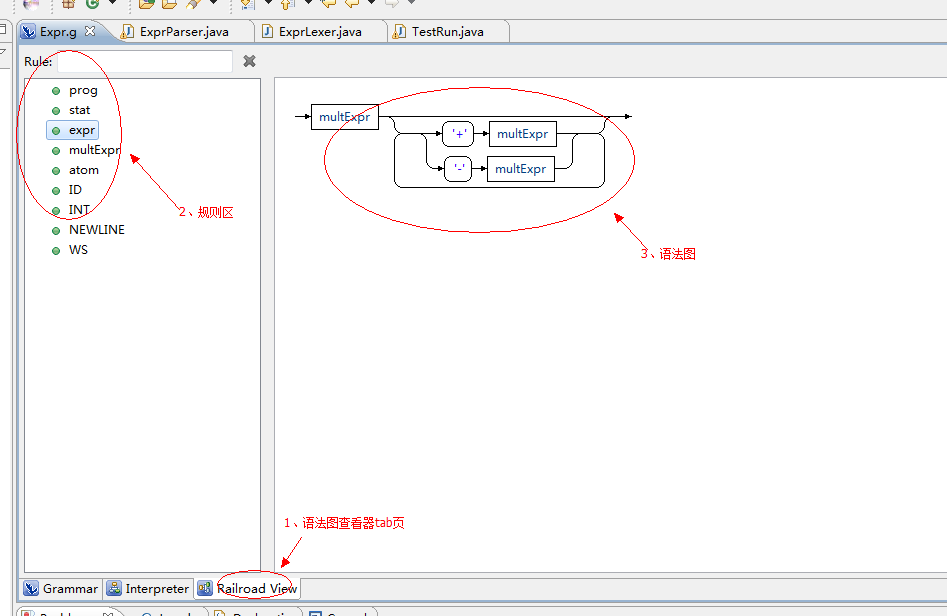
语法编辑器区：这部分只是辅助编辑.g的语法文件。



**解释调试区:** 这部分可以解释调试各种规则。选择需要调试的的规则，在输入区输入待调试的字符串， 点击运行按钮，就可以在输出区看到具体语法树或者问题了。

如果有需要，还可以以新建用例的方式将该次测试保存为gunit测试用例。以供多次测试用。Gunit用例可以转化成junit用例….





# 使用篇

在安装完成后，并通过简单的使用，熟悉了antlr工具antlrworks以及插件antlrv3ide的各部分，已利其器了，剩下的就是如何欲善其事了。

本篇的安排拟这样: 先吹吹水，了解一下一些历史典故，然后对一些基本概念作一个大体介绍，对经典的表达式进行剖析，再安排一个具体的场景《状态机代码生成框架》，中间会穿插一些编译原理知识。

通过本篇的学习后，你应该能够使用antlr把工作中需求转化成设计，如果不能，就给我来email吧，我们一起探讨一下如何改进这个教程。

## 无责任乱弹

一般古代出征时总会祭拜先祖或者什么大神之类的，求乞庇护。这里我们也烧几柱香来拜祭一下相关的先祖们。

上一个世纪是一个伟人井喷的世纪.

希尔伯特提出了著名的23个著名的问题,在第二个问题中，并企图建立一套完备的独立的相容的公理体系,使得一起的数学命题在这套公理体系内经过有限步骤地推导都能得到真伪的判断，这个世界大同式得梦想让当时所有的数学家都振奋不已。但在1931年，哥德尔不完备定理的出现，让这一共产主义式的梦想破灭。哥德尔令人信服的证明了：任何相容(无矛盾)的公理系统，只要包含初等算术的陈述，必定存在一个不可判定的问题，用这组公理不能判断其真假。也就是说，“相容”和“完备”犹如鱼和熊掌，不可得而兼之。吹开迷雾后才发现理想的公理系统就像一件捉襟见肘的华丽外衣。

1936年，图灵和邱奇为了第10个问题，分别构造了两种殊途同归的计算模型，图灵机和lambda算子(貌似还有一个数学家提出了递归函数)。图灵机以感性直观的方式展现了逻辑的机械性质，而lambda算子则以理性的方式揭示了逻辑的抽象变换本质。图灵停机问题和lambda算子中的不动点算子(Y combinator)为哥德尔德不完备定理提供了有力的佐证。(图灵机和lambda算子之于计算理论与波粒二象性之于物理比较像,在基本粒子身上同时体现出了两种特性)。

图灵的计算模型将思维的运动转化成了机械的运动方式，为计算机的制造提供了可能。1944 ，冯诺依曼要计算原子弹的反应传播雇佣了100多名女计算员，日以继夜的计算仍然不能满足计算需求，偶然机会他了解到了ENIAC计算机的研制计划，很感兴趣，改造了一下，用逻辑等价的稳定可靠的基于二进制设计的电路 代替了复杂的多进制逻辑电路，成了，然后再给那台机器加了个内存条，就变成了传说中的冯诺依曼机（笑侃，莫当真，冯诺依曼机是指一种结构）。

有了计算机，自然就催生了程序设计语言，甚至在还没有计算机之前，就已经有了程序员，第一个程序员是一个非常有才华的美女:Ada，拜伦的女儿。这世界上，聪明的女人本来就很少，美丽的聪明女人更是少之又少。可惜英年早逝，三十六岁就down掉了，每每想起此事，未免黯然叹息，天妒红颜阿。但她天才的预见留存来下来，并在之后的一百多年后一一得到验证。每次读到这个ada这个单词就 仿佛看到在皎洁的月光下，有一名婉约的女子，站在世界之巅瞻望，她火热眼睛里面映照着单调的01世界精彩的未来，在那里不仅有子程序，流程图，还有音乐，图画….

在描述程序设计语言的本身的时候，不可避免地涉及到巴克斯-诺尔范式.

半路出家貌似是很多大拿的路子。或许是因为他们的聪明才智是特地为计算机而生的，之前，都是在为遇见的那一刻准备。巴克斯和诺尔就是这样的仁兄。

据说巴克斯年少时不爱读书，混了个中学毕业。在大学时由于无知选择了化学，但也没怎么学好...(感谢当年同学的不杀之恩哪。要不编程语言历史得倒退多少年阿)，不过却年少方刚比较热血，二战时头脑一热跑去当陆军医疗兵了，在这中间检查出了脑瘤，接受了手术。兵是当不成了，鬼子也是打不了了，但光活着也不是办法，于是他想转职做个无线电工了此残生算了。在上岗前的培训中，忽然着了魔般的对数学感兴趣了，(鬼知道是不是那个脑瘤影响了他大脑的结构)，后来回炉考研重造，学成后跑去IBM，后来的事，大家都知道了....就这样一代天才终于找到了自己的天赋和使命，找到了上帝交给他的任务，开发了第一个高级编程语言，并抽象出了一个描述高级程序设计语言的规范:巴克斯范式。

诺尔更是一个意外的客串.丫当时根本就没想过要在计算机界闯出个名堂来，而醉心于夜观天象。他本职也是一个天文学家，由于需要处理大量的天文数据，也就被逼成了计算机专家。在巴黎的著名的algol大会上，诺尔大幅度的简化了巴斯克范式，并作为大会总结人用它优雅简洁的描述了algol的特性，于是就产生了那份在编程语言史上有划时代意义的<算法语言algol60报告>，这个报告展示了从接近人类语言的视角去和机器沟通，而不是用机器语言的视角去和机器沟通的方式。换句话说，他们成功的让所有的开发活动从以机器为中心转移到了以人的逻辑思维为中心，从此解放了机器特性对人的思维逻辑的束缚，这是个了不起的质的飞跃，就像从活着是为了吃饭跃迁到吃饭是为了活着的境界。

诺尔还是一个才华横溢的人，足迹涉及了天文学，心理学，生物学，古典音乐等，但在其他领域都不如他在计算机上的名声大，后来也就慢慢的专业搞计算机了。还有一点有意思的是，诺尔虽然得了图灵奖，但却始终不承认图灵关于人工智能的想法。世界上的事往往都是这么赶巧，无心插柳的，却荫了一大片。宿命阿。

接着该到乔姆斯基出场了。我眼中的乔大叔堪比金庸笔下的风清扬。他虽然不是计算机科学家，只是哲学家和语言学家，但他成功地把语法结构统一到了状态机的计算模型上来，向世人揭示了语法的数学本质。在没有看到乔姆斯基的形式文法前，我从未意识到感性的语言还有这么理性的内涵。当然这中间可能很多都是来自于他的老师哈里斯关于语言结构[线性算子](http://zh.wikipedia.org/wiki/%E7%BA%BF%E6%80%A7%E7%AE%97%E5%AD%90)方面的发现。

好了，祭祀大典结束...接下来我们该分猪肉了。

## 关于语法解析的一些基本概念

虽然现代语文到处都是主谓宾什么的，但是很奇怪，在古代却鲜有分析语法的，说文解字的很多，谋篇布局的也很多，可是研究语法的却很少，貌似这是一个很自然的事情，这件事确实好玩，古人的文章都是没有标点的，全靠语义断句，和现代的先判断语法再判断语义几乎是违背的。后来马建忠照着西方的语法研究体系框架，对中文做了一个全面的审视，横空出了一本<马氏文通>，奠定了现代汉语研究的基础。但是这种词本位的研究方法，终究不是太合适，以至于到了现在，都还没有一种靠谱的分词方法，那些号称准确到95%以上的多数都是基于统计的方法，也就是说，它们的本质基本都是基于现有使用频率而定，不具备推导性。我隐隐约约的感觉到(ps:只是隐隐约约的感觉到)，中文体系的机制有点类似lambda算子，英文体系的机制有点象图灵机体系。

:) 扯了那么多，是想告诉你... 本教程不打算研究中文，哈哈。如果你打算做中文语法解析，下面的内容，你可以不用看了。

本篇并不打算以严格的形式介绍编译原理的前端的内容，因为，严格的话也是抄龙虎鲸书等,而且，愿意啃哪几本书的，基本也用不着本教程了。也就从需求的角度信手即乱弹了。

通常，我们要表达一个系统的观点，都会用文章来表达，一篇文章，是由若干个段落组成， 段落是有若干有逻辑联系的句子组成。句子是有若干词语构成的，就像一层一层的洋葱。 词语或者叫单词就是那些无法再拆分的最小的具有概念意义的单位。词语是由字母有序组成的，但字母不具备概念意义。到了这里，就有了一条严格的分水线了.... 以单词为分界线，单词之上的，组成结构的元素都是具有概念意义的。单词以下，组成结构的元素都是没有概念意义的。（请再一次注意，我说的是英文体系的...不是中文体系) 于是这里就引出了两个概念，词法分析和语法分析。

### 什么是词法分析，什么是语法分析，这两者有什么不同？

**词法分析:**一门研究无意义的字母如何组成有意义的单词的技术。

**语法分析:**一门研究有意义的单词如何组成更复杂意义的句子，文章的技术。

例子:

hello,my name is alan, i come from china.

以上是一个自我介绍的片段。

词法分析是识别出一个一个的单词 比如hello， my等。 单个的 h，e ,l,l,o 是不具备概念意义的， 只有有序组成hello的时候才具备一个概念意义。

那语法分析又是什么呢？ 我们假定这是一个标准的自我介绍的文章。这个文章由三段组成：问候语，名字介绍，归属地介绍 :) .

问候语采用 hello 或者 hello xxx 的形式。

名字介绍采用 my name is xxx 的形式。

归属地介绍采用 i come from xxx的形式。

那我们规定的这种标准介绍的格式，就是所谓的语法，每一条叫语法规则。 我们通常意义上的英文语法，本质上也是这样的，但他们的语法结构层次多点，各组成部分的划分多些和标准些。我们常说的英语句式，就是所谓的语法，但是语法不仅仅包括句式，还包括篇章段落结构等。

比如一个最简单的英语句式:主语谓语宾语。

主语由名词或者代词组成。

谓语由动词组成。

宾语由名词或者代词组成。

以下为简单起见就假设各个结构组成成分最多只有两个吧。

**名词：**专有名词或者普通名词。(参见http://baike.baidu.com/view/26580.htm)

专有名词：常见的人名或者地名:也见字典的专有名词列表。如**China**（[中国](http://baike.baidu.com/view/61891.htm)）、**Asia**（[亚洲](http://baike.baidu.com/view/2918.htm)）**Beijing**（[北京](http://baike.baidu.com/view/2621.htm)）

**普通名词:**个体名词或者物质名词。

**个体名词:** 单个人或者物的名词，如car 汽车 room 房间 fan 风扇photo 照片等。

**物质名词:** 表示物质或不具备确定形状和大小的个体的物质。fire 火 steel 钢 air 空气 water 水 milk牛奶。

**动词:**表示动作的词语。 一般见字典的动词表，比如is ，do， make 等等。

代词: it，i， she，he等词语。

也就是说，这个经过阉割的简单的主谓宾的句式，语法结构是这样的:



椭圆框内的是字母序列，是具体的单词例子。

方框是作为语法分析的基本单位，是词法分析出来的最后终结果。

圆角框是符合的语法结构。

像 it is tree ，就是这样的结构。



那如何来描述词法语法规则呢？ 这就涉及到了鼎鼎有名的巴克斯-诺尔范式。

### 巴克斯-诺尔范式,产生式,最左推导,最右推导,左递归,右递归

巴克斯发明了一种叫巴克斯范式(Backus Normal Form)的格式来标示语法词法，语法词法本质上其实是一样的，不过是组成单位是否有逻辑概念意义而已。

巴克斯范式后经诺尔改进，后人便合成巴克斯-诺尔范式 (Backus-Naur Form)简称都是叫BNF。

这个范式很简单(见http://zh.wikipedia.org/zh-cn/%E5%B7%B4%E7%A7%91%E6%96%AF%E8%8C%83%E5%BC%8F)，

规定了一个语法结构的描述是由一堆语法规则组成，

每一个规则符合这样的格式:

<符号> ::= <使用符号的表达式>

这里的 <符号> 是非终结符，而表达式由一个符号序列，或用指示选择的竖杠 '|' 分隔的多个符号序列构成，每个符号序列整体都是左端的符号的一种可能的替代。从未在左端出现的符号叫做终结符。

具体可以解读成:

<非终结符> ::= <可能选择表达式1> | <可能选择表达式2>...

BNF在描述上是存在着一些缺陷的， 它无力描述它所占用的<> ::= | 等字符，用现在的话说是没有转义功能。另外，它对可选结构(可有可无的选项)以及重复项的描述很罗嗦。于是就有了EBNF(扩展巴克斯范式)的出现。

具体看(http://zh.wikipedia.org/wiki/%E6%89%A9%E5%B1%95%E5%B7%B4%E7%A7%91%E6%96%AF%E8%8C%83%E5%BC%8F).

现在常用的，基本都EBNF的格式，但BNF和EBNF 两者能力等价。

一般采用产生式来做语法推导，可以认为产生式其实就是BNF的一条规则的另一种表达方式。产生式采用 前提条件->结论动作 这样的表达方式。

采用这种产生式的规则，可以很好的诠释语法的推导过程。

比如有一个这样的规则集合: {S->AB,A->a, B->b } 要分析的字符串 SB

最左推导是: SB->ABB->aBB->abB->abb;

最右推导是: SB->Sb->ABb->Abb->abb;

他们的区别是很简单的 从左边还是从右边开始按照规则替换。

在替换的过程中，可能会遭遇左递归或者右递归的情况，左递归是相对最左推导而产生的，右递归是相对最右推导而产生的。

举例说明,{A->Aa,A->b} 当使用最左推导时，A->Aa,就会陷入无限的分裂中 A->AAA...a。经典的处理方法是将这两个产生式转换成3个等价的产生式{A->bR,R->aR,R->空}; 具体可以自行验证一下，编译原理的书也有详细叙述。

右递归也类似这样。不过是形如(A->aA,A->b)的产生式规则 在运用最右推导的时候才会产生，解决方法也雷同。

### 语法解析的自顶向下和自底向上的策略

语法解析的过程，就算是拿这个纸带似的字符串序列按照一定规则扭成目标的语法结构，如果能扭成，就说明这个是字符串序列是符合这个语法的，否则就是不符合这个语法的。

那现在我们关注一下这个扭的过程。一般而言，有以下两种策略。

自顶向下的策略：在构造语法树(就是那个像一颗倒着的树的语法结构)的时候，从语法树的根节点开始，按照先根顺序构造各个节点，也就是说，叶结点总是后于根节点创建， 过程就像从上面(根节点)到下面(叶结点)建造，因此得名。

自底向上，刚好相反。

如何理解这两者的区别呢？

一个人，左手拿着一个语法结构，右手从一个管道里面取出 输入字符串的积木块。往语法结构里面叠放，最后叠出来了带输入字符串的语法结构，这个就叫自顶向下。

如果那个人左手拿着一段输入字符串，右手从一堆 零散的语法结构积木块里面取出合适的积木块，往输入字符串上叠放，最后得到一个带输入字符串的语法结构，这个就叫自底向上。

我们玩一个A说B说得游戏来具体说明这个过程吧。

我们定义这样的语法结构:

句式DEMO: 主语 谓语 宾语

主语: 人名，简单的以"A" "B" ..."Z"来命名。

谓语: "say" 和 "is" 两个动词。

宾语: that从句 或者 "over"这个词。

that从句："that" 主语 谓语 宾语。

现在我们来分析 这样的一个句子:

A say that B say that C is over;

先来看看自顶向上是怎样的过程：

我们先略去词法分析，假设A B C say is that over这些已经被识别为单词 。

左手拿着语法结构 句式demo ，右手准备从管道里面取出单词。

句式demo 要求的第一个元素是 主语：人名。 右手从管道里面取出的第一个单词是A ，刚好是人名，两者符合，于是到下一步，



句式demo 要求第二个元素是 谓语 say 或者 is 两个单词。 右手从管道取出的第二个单词是 say， 也符合要求，于是可以继续下一步。



句式demo 要求的第三个元素是 over 或者 以that开头的that从句。 右手从管道取出的是that.. 满足是that的从句的特征，但是此刻不能判断是不是正确的that从句，假定为that从句，跳到that从句的语法规则，继续下一步。



that从句要求that之后的元素是 主语:人名。右手从管道里面取出的是B ，符合要求，继续下一步...

重复以上步骤，刚刚好左手所有用到的规则都得到满足，而右手管道的内容也都空了。

于是我们就得到了这样的一棵语法结构树:



我们再来看看自底向上的过程:

把上面过程的左右手换过来.左手字符串的管道，右手语法规则堆。我们借助一种叫堆栈的结构工具，你可以把堆栈想象成像 子弹夹这样的东西，后压进去的子弹先出来，如果有子弹的话，永远有一颗子弹保持在顶端，我们把那个顶端位置叫做栈顶。

自底向上的处理过程总体是这样的，把左手的字符串中的元素按照顺序，像子弹一样，一个一个的压入进去，压入一个就从栈顶开始依次往栈底检查，如果符合右边语法规则堆里面的规则，就把符合部分的堆栈内容弹出，把那个抽象规则名，压入进去， 把字符串压入栈的过程叫**移动**，发现符合语法规则，变成抽象的语法名称实例(或者handle) ,然后再压入栈的过程叫**规约**。

具体例子过程如下:





## 经典的表达式教程解析

几乎每一本语法解析的教程都会用整数四则表达式来当作入门级别的语法教程，这节就以实用的视角去剖析这个教程，我们就当它是一个真实的case吧。

### 界定需求

假设你给某一个程序一串常用的三则混合运算的表达式，要求这个程序给你计算出结果，显示的要求如下:

1. 使用者传递文本传递文本给程序，程序解析完后在控制台输出结果，或者错误信息
2. 输入和结果为整型范围的整数。
3. 支持命名的表达式。表达式结果可以被其他表达式按名引用。支持多条表达式批量计算。
4. 支持加减乘三种运算符号,符号操作意义同数学，注:乘优先级比加减高。
5. 支持括号，括号改变原有的运算符号优先级。

这个case比较特殊，它的需求是简单直白的，故没经过什么抽象过程。然而，在真实的环境中，以用户用例界定功能边界，从用例中抽取出需求。这是一个很重要也很复杂的工作，这也超出了本书的范围与作者的能力，有兴趣的可以看看关于<需求的探索> <rup><分析与设计>等软件需求分析与设计的书。

一般用use case来捕捉描述需求，这里我拟总结出自己的一些经验法则，但有必要提醒一下，这只是个人经验法则，没有经过严格的证明是对的.请批判性的接受。

* 要找出参与者，语言是多方沟通的契约。所以要找出场景中的参与者。并对参与者进行归类。即要抽象出参与者的角色。（见需求1）
* 找出潜在的条件约束，以及沟通的目的，这个对语义实现很重要。比如在实时交易的场景下，对时间的要求很严格，这种就是潜在的约束条件。 沟通的目的又是什么呢？ 像表达式的例子的目的就是计算这个表达式最后的结果。(见需求2)
* 抽取出专业术语，这个对词法分析和语义实现很重要。这其实是一种取巧的做法，利用人的思维锤炼出来的结果。当一件事情被锤炼千百遍的时候，人脑一般都会进行抽象，用最精简的信息来描述这个事件，这就是常说的行话。精简的信息包含的大量的内涵，内涵决定了语义的实现。比如行话“调试(debug) ”就是包含了设置断点一步一步追踪检查程序代码每一处状态的一系列过程。(见需求3、4)
* 明确专业术语要求的上下文。比如隐含某些参数或者需要指定具体的参数，那些术语需要和那些术语配套使用。(见需求3、4、5)
* 判断描述的粒度。有一些场景，用一个句子就可以描述清楚了，有些场景需要用一段话来描述，而有一些场景，需要用一篇文章来描述。

### 设计语法

设计语法是一个很麻烦的过程，但也是一个最重要的过程。

1. **首先要列出所有的单词项，并设计出词法规则。表达式例子中单词项如下:**

标识符: 表达式名，由A到Z组成，位数不限。

整数项: 由0到9组成，位数不限，(实际上不考虑而已，本例不支持大数).

加号:+

减号:-

乘号:\*

除号:/

左括号:(

右括号:)

词法规则，实际就是正则表达式来描述，不熟悉的人到网络上搜以下正则表达式的经典教材看看吧。这里不累述了。

1. **分析使用场景，根据需求，逐步分析出语法推导的产生式规则集合。**

从一堆杂乱的需求中去构造出精巧准确的语法结构，是一个由简入繁的不断反复迭代的过程，如抽茧剥丝后裁衣一样，大凡此类工作都有一定的艺术鉴赏性，但只有你沉浸其中时才能体味。

我们观察需求，对语法元素进行分类，按照其作用将涉及元素分成三类{整数项，标识符}作为操作项，{加减乘}作为运算符，{括号}作为辅助符号。

我们从每一个分类中取一个元素出来分析，构造出产生式描述的语法规则组。

然后再逐步的加入同类的元素。

先取 整数项，加法，和括号 三个元素。它们只有两种形式，

整数项+整数项+整数项+....

整数项+(整数项+整数项)+整数项+....

假设a代表一个整数项，

用这个两条产生式{S->S+a,S->a} 就可以描述第一条语法规则，有人可能会问这两条产生式是如何的，其实我也不知道，貌似自然而然的就抽象出来了，再仔细想了一想，我觉得用数学归纳法比较适合解释这个过程。

a,a+a,a+a+a,a+a+a+a,a+a+a+a+a....

=> a,{a}+a,{a+a}+a,{a+a+a}+a,{a+a+a+a}+a....(我们借助花括号来帮助界定成分)

括号里面的就是前面的式子。然后就推出了通项：A[0]=a A[n] = A[n-1]+a；

请注意：这里的+是一个字符串符号, a+a代表a连接+号连接a。

{S->S+a,S->a} 虽然直观却不适合最左推导，按照公式转一下(见上面的最左推导)。

{S->aR,R->+aR,R->空}

再看第二种形式； 通过观测可以看出来，括号里面的是一个子表达式。

于是{S->S+a,S->a,S->(S) }

转成符合最左推导的形式: { S->aR,R->+aR,R->空,S->(S)}

重新选取 整数项，减号，括号，进行同样的分析和构造，

可以得到 {S->S-a,S->a,S->(S) }

由于加减法对等，合并后可以得到{S->S-a,S->S+a,S->a,S->(S) }

这就是描述加减混合运算的式子，很容易通过实例来验证的。

同理的，我们可以把乘法也加入进来,可以得到这样的一个

{S->S-a,S->S+a,S->S\*a,S->a,S->(S) }

转成符合最左推导的形式: { S->aR,R->+aR, R->-aR, R->\*aR ,R->空,S->(S)}

如果，我只是说，如果...这个世界上只语法实现语法而不实现语义的话... 那就太完美了。但通常上帝都会给你留点遗憾的。这些规则完全可以合法的解析含加减乘的表达式，但是它们无法区分出需求中比较重要的一条:乘法优先级比加减的优先级要高。

乔boss说过一句经典名言：形式应该追随功能。很显然的，我们之前抽象归纳出来的语法规则的形式并不是追随功能的，那就变换一下吧。我们知道乘法的优先级比加减法高，故涉及乘法的表达式部分应该先结合成项，然后再作为加减算术符号的两边的操作项再形成加减表达式，故我们在乘法的基础{M->M\*a, M->a,M->(M)}上，垒上一层加减法看看，如下:

{M->M\*a, M->a,M->(M), S->M+M,S->M-M}

嗯，挺接近了，但这组语法规则无法解决两个问题:

1、无法处理多个连续的加减号。 形如a+a-a-a+a

2、从开始符号无法推导纯的连续乘表达式。

3、无法处理子表达式中超过一个加减号的。

针对问题1，分析一下特点，第一项必须为a ，第二项为第一项连接+a 或者第一项连接-a,因此推出通向项为前一项连接上+a或者连接上-a，于是我们就大致可以推断出规则:引入一个Expr 简写为E {E->M,E->E+M,E->E-M}来代替加减项，

于是得到规则组: {M->M\*a, M->a,M->(M), E->M,E->E+M, E->E-M,S->E }

同时因为加入一个E->M，也解决了问题2，即从开始符号推导出连续乘。

针对问题3，把M->(M) 改成 M->(E)。也即乘法中要有比乘法范围更广的包含了加减法的子表达式做为因子项，

于是得到语法规则组{M->M\*a, M->a,M->(E), E->M, E->E+M, E->E-M,S->E }

转化成符合最左推导的语法规则组是

{ M-> aR,R->\*aR,R->空,M->(E), E->MN, N->+MN, N->-MN,N->空,S->E }

至此，整数项，加减乘符，括号已经被加入进来了。

现在加入标识符号。

引入标识符id 简写为i，

加入标识符后， a 和 i 就成了同等意义的符号，可以抽象出一层，我们使用大写A来代表。

{A->a,A->i}

同时，匿名表达式仍然保持之前同样的语法结构，新增了一个命名表达式，

所以，用一个新的元素P{P->E,P->i=E}来代替，

于是规则变成

{ A->a,A->i,M->M\*a, M->a,M->(E), E->M, E->E+M, E->E-M, P->E,P->i=E, S->P }

转化成符合最左推导的语法规则组是:

{ A->a,A->i,M->AR,R->\*AR,R->空, M->(E), E->MN, N->+MN, N->-MN,N->空, P->E,P->i=E,S->P}

至此所有的重要符号都已经加入完成。

但对照需求3，还有批量表达式的支持，只需要对P的重复即可也即{S->P,S->SP} 。

最后合并在一起便是

{ A->a,A->i,M->M\*a, M->a,M->(E), E->M, E->E+M, E->E-M, P->E,P->i=E, S->P,S->SP }

转化成符合最优推导的语法规则组:

{ A->a,A->i,M->AR,R->\*AR,R->空, M->(E), E->MN, N->+MN, N->-MN,N->空, P->E,P->i=E,S->PQ,Q->PQ,Q->空}

至此，我们就从需求开始进行一步步地推导得到了最后的语法规则组。

如图表:我简单的总结了一些推导技巧以及映射的技巧

|  |  |  |  |
| --- | --- | --- | --- |
| 需求 | 正则 | 最右推导规则组 | 最左推导规则组 |
| 连接项或者与关系项 | ab | S->ab | S->ab |
| 或关系项 | a|b | S->a,S->b | S->a,S->b |
| 可选项 | a? | S->a,S->空 | S->a,S->空 |
| 至少重复一次(正闭包) | a+ | S->Sa,S->a | S->aR,R->aR,R->空 |
| 重复0次或者多次(克林闭包) | a\*  注: a\*=a+|空 | S->Sa,S->a,S->空 | S->aR,R->aR,R->空  S->空 |

1. **把语法规则产生式集合转化成antlr的语法描述**

接下来的工作是把之前所得到的语法规则组，转换成antlr的语法表示。关于antlr v3语法规则的规范请参照(http://www.antlr.org/wiki/display/ANTLR3/ANTLR+v3+printable+documentation),很抱歉，任何初学者都需要精心去读一遍该文档，没有任何的窍门可以走的。我也着手翻译和批注了一点，但由于去年一些突发事情耽搁了很久，v3还没翻译完，v4就出来了，我也在考虑是否继续翻译v3还是新翻译v4.无论

根据上一个步骤的总结表，对

{ A->a,A->i,M->AR,R->\*AR,R->空, M->(E), E->MN, N->+MN, N->-MN,N->空, P->E,P->i=E,S->PQ,Q->PQ,Q->空}

进行翻译，一下翻译采用表达式例子中的命名。

先翻译词法吧,毕竟这是基础。有一点需要记住的是，词法首字母必须大写，一般情况下，都用全部大写来标识词法单元。

ID : ('a'..'z'|'A'..'Z')+ ;//标识符是任意的大小写混合

INT : '0'..'9'+ ; //整数项是任意的数字

NEWLINE:'\r'? '\n' ; //换行符号， \r可选是为了应对window和unix的区别。

WS : (' '|'\t')+ {skip();} ; //空白符号，默认动作忽略。

还有几个符号 加减乘和括号，这几个默认以字符串的形式出现在语法解析中，其实也可以定义在这里的。我们就遵照demo行事吧。

现在开始翻译语法，语法中有一点需要注意的是语法是以小写开始的。

与整数项，标识符相关的:

{A->a,A->i}

atom

: INT

| ID

;

乘法相关的：

M->AR,R->\*AR,R->空, M->(E)

multExpr

: atom ('\*' atom )\* //注:处第一个'\*'是token第二个\*是克林闭包

|'(' expr ')'

;

此处，和demo不太一样。关于子表达式(E)的安置。demo中把整数项以及标识符当成看作是参与运算的原子项atom，并把子表达式也看作是原子项，我们的推导过程中，也把参与乘法的因子中的一个扩展为字表达式，这两者在变换形式上是等价的，只是含义解释不一样。

加减法相关的:

E->MN, N->+MN, N->-MN,N->空,

这个翻译过程有点绕，不是直观，你可以把 N->+MN, N->-MN 合并成 N->((+M)|(-M))N ，令 T->((+M)|(-M)) ,即有 {E->MN,N->TN,N->空} 即可利用总结表中格式转化成demo了。

expr

: multExpr

( '+' multExpr

| '-' multExpr

)\*

;

实际上也有另外一种语法形式， 把 N->+MN, N->-MN 合并成 N->(+|-)MN ，令 T->(+|-)M ,即有 {E->MN,N->TN,N->空}

expr

: multExpr

(( '+' | '-' ) multExpr )\*

;

这两种形式都可以的，demo的作者偏向于前者，也许那样比较好理解些吧。

关于命名和匿名表达式的:

P->E,P->i=E

这个没啥悬念的，直接翻译好了，demo多了一些newline ，无伤大雅。

stat: expr NEWLINE

| ID '=' expr NEWLINE

| NEWLINE

;

关于批量表达式的：

S->PQ,Q->PQ,Q->空

prog: stat+ ;

至此最重要的语法设计完成了，对照需求，一步一步地进行推导出产生式规则，最后翻译成antlr所支持的ebnf范式。

### 实现语义

在很多的场景里面，具体语法一般要转成抽象语法树，然后再在抽象语法树上实现语义动作，这样做的好处可以避免冗余的处理，不同的语法元素但具有相同的语义含义的，可以转化成统一格式合并处理。

但一般简单的语法可以直接嵌入语义动作处理就ok了，比如这个demo。

语义动作一般是以包含在{}的目标语言的代码。具体的规范参照: (http://www.antlr.org/wiki/display/ANTLR3/Grammars)

本节就针对重点地地方进行代码注释。

prog: stat+ ;

stat: expr NEWLINE {System.out.println($expr.value);} //打印匿名表达式值，$expr是引用这个expr这个语法元素，需要注意的是 value并不是expr的内建的属性，而是自定义的return的值，参见expr的语法表达式。

| ID '=' expr NEWLINE

{memory.put($ID.text, new Integer($expr.value));} //命名表达式的值存入一个hashmap对象。

| NEWLINE

;

expr returns [int value] //在此定义了这个expr对调用者提供一个value的返回值。

: e=multExpr {$value = $e.value;} //以e的方式引用第一个multExpr,语义动作在此把左边的表达式的值赋给返回值变量。注意:$e.value也不是内建的，是multExpr定义的返回值。

( '+' e=multExpr {$value += $e.value;}//以e的方式第二个multExpr，由于作用域不同，故无冲突，这个语义动作只有遇到+才会执行。语义动作式返回值累加e这个项。

| '-' e=multExpr {$value -= $e.value;}//同理，不过是累减。

)\* //需要特别注意的是，这个克林闭包，会让上面的两个语义动作，不断的重复。

;

multExpr returns [int value] //这个表达式和expr 无啥本质差别

: e=atom {$value = $e.value;} ('\*' e=atom {$value \*= $e.value;})\*

;

atom returns [int value]

: INT {$value = Integer.parseInt($INT.text);}//将词法的字符串值转换成整型，然后赋值给返回值，text是内建变量。

| ID

{

Integer v = (Integer)memory.get($ID.text);

if ( v!=null ) $value = v.intValue();

else System.err.println("undefined variable "+$ID.text);

}//如果是标识符，则从hashmap中按名取出来。

| '(' e=expr ')' {$value = $e.value;}

;

## 状态机代码生成框架实例分析

待写.....

有兴趣的可以访问：

http://www.github.com/alan2lin/makefsm

若有兴趣参与开发或者讨论的请加qq群: 281827854

# 引用

1. Antlrv3ide 的下载地址和文档: <http://antlrv3ide.sourceforge.net/>
2. Antlr 的下载地址: <http://www.antlr.org>
3. Antlr视频教程的地址(不能打开没有验证过): <http://javadude.com/articles/antlr3xtut/>