# Antlr v3规范翻译以及批注

这不仅仅是一个翻译...但它还是什么，取决于你对它的认识。

--林氏按语

本文采用BSD授权，涉及英文部分版权归属于Terence Parr<parrt@cs.usfca.edu>，中文部分归属于alan<[workspace.public@gmail.com](mailto:workspace.public@gmail.com)>。

你可以随意引用，修改，转载，再发布本文的任何部分，只需保留上行BSD版权申明即可。

促译而成，工糙技浅，琢玉有瑕，恐误阅者，敬请行家，不吝指教，拨冗斧正，感激涕零。

主要贡献者列表:

翻译&批注:流氓九段(alan2lin) 校对:甘草, icy 排版: 娜达莎

其他贡献者列表:

这是原文的网址:

http://www.antlr.org/wiki/display/ANTLR3/ANTLR+v3+printable+documentation

这是Terence Parr的回信，确定了版权没有问题。

On Aug 17, 2011, at 11:24 PM, [workspace.public@gmail.com](mailto:workspace.public@gmail.com) wrote:  
  
> alan.lin wrote:  
>   
> Dear Mr.Terence Parr :  
>     Thank you for creating antlr. I like it very much,so i build a groups with my friends in [groups.google.com](http://groups.google.com/) (url:<http://groups.google.com/group/antlr_chinese>). I hope we can introduce antlr to more chinese.  
  
great! :)  
  
> Could we translate your articles and the reference material which in ( [www.antlr.org](http://www.antlr.org/)) so that we can put it out on antlr\_chinese groups with lgpl license?Or could you give us some other suggestion for our group?  
>     Looking forward your soon response ...  
  
Sure. But, I would prefer that you use the BSD or MIT license or something equally easy to understand. lgpl is a very complicated restricted license.  
Ter  
>   
>   
> Best Regards!  
>   
>                            Your sincerely Alan lin  
>                           2011-08-18

[**ANTLR v3 printable documentation**](http://www.antlr.org/wiki/display/ANTLR3/ANTLR+v3+printable+documentation)

* Added by [Thomas Aigner](http://www.antlr.org/wiki/display/~tomia), last edited by [Terence Parr](http://www.antlr.org/wiki/display/~admin) on Jan 16, 2007

An inline version of [ANTLR v3 documentation](http://www.antlr.org/wiki/display/ANTLR3/ANTLR+v3+documentation).

Antlr V3 可印刷的文档

Thomas aigner 添加, Terence Parr 在 2007年1月16日最后修改.

一份在线版的ANTLR v3 文档。

**ANTLR3 Code Generation Targets**

**Antlr3 目标语言代码生成**

Code generation for the following target languages is currently in development, testing or is complete. Visit the page for each target language for more information - hopefully the persons dealing with each target language will update their respective rows in this table with their current status.

See also [Target API documentation](http://www.antlr.org/api) and [How to build an ANTLR code generation target](http://www.antlr.org/wiki/display/ANTLR3/How+to+build+an+ANTLR+code+generation+target).

下面目标语言的代码生成(功能) 正在开发、测试或者已经完成。每一个目标语言的详情请访问相关页面-希望每一个目标语言维护者用他们当前状态更新这个表格中的各自的行。

另请参阅 [Target API documentation](http://www.antlr.org/api)  与  [How to build an ANTLR code generation target](http://www.antlr.org/wiki/display/ANTLR3/How+to+build+an+ANTLR+code+generation+target).

|  |  |  |
| --- | --- | --- |
| **Language 语言** | **Irresponsible Person 无责任人** | **Status 状态** |
| [Ada](http://www.antlr.org/wiki/display/ANTLR3/Antlr3AdaTarget) | Luke A. Guest | Currently dormant  目前静止(更新). |
| [ActionScript](http://www.antlr.org/wiki/display/ANTLR3/Antlr3ActionScriptTarget) | George Scott (initial port, not actively maintaining初始的移植，非活跃维护) | In sync up to 3.2, but currently not in active development.  已同步到3.2，但当前开发不活跃 |
| [C](http://www.antlr.org/wiki/display/ANTLR3/ANTLR3+Code+Generation+-+C) | Jim Idle | In sync with ANTLR3 development. Use the .tgz files under the dist subdirectory to build the runtime.  与antlr3开发同步，使用.tgz文件在dist子目录下构建运行时(代码). |
| [C++](http://www.antlr.org/wiki/pages/viewpage.action?pageId=29130834) | Gokulakannan Somasundaram  (was Jim Idle & Ric Klaren) | Created on antlr-3.4 and hence in sync with only antlr-3.4  在antlr3.4时创建，此后只和antlr3.4同步 |
| [C#; C# 2](http://www.antlr.org/wiki/display/ANTLR3/Antlr+3+CSharp+Target) | 维护者: Johannes Luber  (贡献者: Kunle Odutola 和Micheal Jordan) | In sync with ANTLR3 Development to 3.3, but a few errors make it beta for 3.3. There are separate targets for .NET 1.1 and .NET 2.  和antlr3开发同步到3.3，但一些错误导致它只是3.3的beta版。 .net1.1和.net2分别有不同的生成目标代码. |
| [C# 3](http://www.antlr.org/wiki/display/ANTLR3/Antlr3CSharpReleases) | 维护者: Sam Harwell | (Added post-release 3.1.3) In sync with ANTLR3 Development, except no support for the -debug or -profile flags yet  已增加快速发行版3.1.3，除了尚不支持 -debug 和 -profile标志外，和antlr3开发同步。 |
| [D](http://www.mbutscher.de/antlrd/) | ? | ? |
| [Emacs ELisp](http://www.antlr.org/wiki/display/ANTLR3/ANTLR+v3+printable+documentation) | Ola Bini | He's working on this at the moment; 他正在这里搞... <http://github.com/olabini/antlr-elisp> |
| [Objective C](http://www.antlr.org/wiki/display/ANTLR3/ANTLR3+Objective-C+Target) | Alan Condit, Kay Roepke | Current with 3.3 version. 当前版本是3.3 |
| [Java](http://www.antlr.org/wiki/display/ANTLR3/ANTLR+v3+printable+documentation) | Terence (parrt at cs usfca edu) | In sync with ANTLR3 Development 与antlr3开发同步 |
| [JavaScript](http://www.antlr.org/wiki/display/ANTLR3/ANTLR3JavaScriptTarget) | Joey Hurst | In sync with ANTLR3 Development 与antlr3开发同步 |
| [Python](http://www.antlr.org/wiki/display/ANTLR3/Antlr3PythonTarget) | Benjamin Niemann | Current with 3.1.3 当前版本是3.1.3 |
| [Ruby](http://www.antlr.org/wiki/display/ANTLR3/Antlr3RubyTarget) | Kyle Yetter, previously Martin Traverso | Current with 3.3 当前版本是3.3 |
| [Perl6](http://www.antlr.org/wiki/display/ANTLR3/Antlr3Perl6Target) | Bernhard Schmalhofer Bernhard.Schmalhofer@gmx.de | Inactive. No code produced yet. Takers wanted. 不活跃，尚未有代码产出。接受者想要这样? |
| [Perl](http://www.antlr.org/wiki/display/ANTLR3/Antlr3PerlTarget) | Ron Blaschke ron at rblasch.org | Early prototyping.  Simple lexer is working. 早期原型中，简单的词法分析器可用。 |
| [PHP](http://code.google.com/p/antlrphpruntime) | Sidharth Kuruvila, Yauhen Yakimovich, Geoff Speicher, Rolland Brunec | Primary milstone is aimed at verification of Lexer, Parser generation. The work towards implementation of StringTemplate is in progress 主要的里程碑目标是词法分析器的校验，语法分析器的代码生成.工作正进展到实现字符串模板。 |
| [Oberon (yes, Oberon)](http://www.antlr.org/wiki/display/ANTLR3/ANTLR+v3+printable+documentation) | Dominik Holenstein | Planning and analyzing. First version expected for  Q1/2007. 计划和分析中，第一个版本预计在2007年第一季度推出。 |
| [Scala](http://www.antlr.org/wiki/display/ANTLR3/ANTLR+v3+printable+documentation) | Matthew Lloyd |  |

**Command line options**

**命令行选项**

Usage:

用法：

java org.antlr.Tool [args] file.g [file2.g file3.g ...]

|  |  |
| --- | --- |
| **Option 选项** | description 描述 |
| -o outputDir  输出目录 | specify output directory where all output is generated; search for token vocabularies in here also  为产生的所有输出指定输出目录. token 词汇表的查找也在这里. |
| -fo outputDir  输出目录 | same as -o but force even files with relative paths to dir  和 –o 一样,只是强制每一个文件带输出目录的相对路径 |
| -depend | generate file dependencies; don't actually run antlr  产生文件依赖; 不实际运行anltr.  注:这个只是产生一个文件依赖的报告。  格式:产生的文件名(目标源代码文件或者词汇表):所依赖的.g文件。  例子:  ExprParser.java : Expr.g  .\Expr.tokens : Expr.g  ExprLexer.java : Expr.g |
| -lib dir  目录 | specify location of token files and important grammars  指定 token 文件和重要语法的位置. |
| -report | print out a report about the grammar(s) processed  输出一份关于语法处理的报告.  注:输出每一个语法规则的信息。  格式大约如下: 语法名.语法规则:行数:列数 决策顺序:k值  例子:  Expr.prog:15:9 decision 1: k=1  Expr.stat:17:5 decision 2: k=2  Expr.expr:27:9 decision 3: k=1  Expr.multExpr:31:70 decision 4: k=1  Expr.atom:35:5 decision 5: k=1 |
| -print | print out the grammar without actions  输出不带动作的语法。  注:这个选项在非常有用,当语法文件实现到后期，混合了一大堆语义动作时，antlrworks无法调试这些语法。这个选项能够让剥离出动作，在antlrworks 中调试。  当然比这个选项更好的是:把语义实现放到抽象语法树中去处理能让设计更清晰。建议不要因为这个选项的便利而放任糟糕的设计习惯。 |
| -trace | generate a parser with trace output - if the default output is not enough, you can override the traceIn and traceOut methods  产生一个带追踪输出的解析器-如果默认的输出不合心意，你可以重写traceIn 和 traceOut方法。  注: 这里的trace 是anltr为了追踪词法语法解析过程在进入/退出一个语法过程时，往外输出一些便于辅助分析观察的信息，可以认为是一种独特的日志。 |
| -debug | generate a parser that emits debugging events  产生可以发射调试事件的解析器。  注：antlr 集成了高级层面的语法调试功能。采用事件模式来实现。打开调试模式，将会在解析器生成代码中的特定位置，插入 触发调试事件的代码。  当事件发生时，代码控制权将转交给调试器接管，而调试器的实现是完全独立于antlr的产生的代码。这样的设计使得antlr能够支持远程调试(只需调试器允许远程ui接入)。 |
| -profile | generate a parser that computes profiling information  产生带计算描述信息的解析器。  注:  在代码生成时同时生成profile的代码，格式如下。  ANTLR Runtime Report; Profile Version 3 //版本信息  parser name test.ExprParser //解析器名字  Number of rule invocations 7 //调用的规则数  Number of unique rules visited 5 // 访问过的唯一规则数  Number of decision events 9 //决策事件数  Overall average k per decision event 1.0 //每一个决策事件k值的总平均数。  Number of backtracking occurrences (can be multiple per decision) 0 //回溯发生的次数。  Overall average k per decision event that backtracks NaN  Number of rule invocations while backtracking 0 // 当回溯时调用的规则数。  num decisions that potentially backtrack 0 //存在可能回溯的决策数。  num decisions that do backtrack 0 //执行回溯的决策数。  num decisions that potentially backtrack but don't 0 //可以回溯但是不回溯的决策数。  average % of time a potentially backtracking decision backtracks NaN //可以回溯的决策回溯时的平均时间  num unique decisions covered 5 //唯一决策的数量  max rule invocation nesting depth 5 //规则调用嵌套深度的最大层数  rule memoization cache size 0 //规则内存缓冲大小。  number of rule memoization cache hits 0 //规则内存缓冲命中次数  number of rule memoization cache misses 0 //规则内存缓冲未命中次数  number of tokens 4 //token数量  number of hidden tokens 0 //隐藏的token数量  number of char 0 //字符数  number of hidden char 0 //隐藏字符数  number of syntax errors 0 //语法错误数  location位置 n 个数 avgk 平均k值 maxk最大k值 synpred语法断言 sempred语义断言 canbacktrack可回溯 |
| -nfa | generate an NFA for each rule  为每一个规则产生 NFA  注: 这个选项产生的 是 DOT格式，需要用graphviz之类工具生成图片，可以用GVedit 打开，再作转换。 |
| -dfa | generate a DFA for each decision point  为每一个决策点产生DFA。 |
| -message-format name | specify output style for messages  为信息指定输出的样式。 |
| -X | display extended option list  显示扩展选项列表 |

There are a bunch of less often used "extended" options as well

这是一些较少用的”扩展”选项

|  |  |
| --- | --- |
| **Extended option**  **扩展选项** | description  描述 |
| -Xgrtree | print the grammar AST  输出语法的抽象语法树(AST) |
| -Xdfa | print DFA as text  以文本方式输出确定型有限状态机(dfa) |
| -Xnoprune | do not test EBNF block exit branches  不检测扩展巴斯科范式(EBNF)块的退出分支。 |
| -Xnocollapse | collapse incident edges into DFA states  把 缩进边 折叠到 dfa 状态里。 |
| -Xdbgconversion | dump lots of info during NFA conversion  导出 NFA 转换的大量信息 |
| -Xmultithreaded | run the analysis in 2 threads  起两个线程运行分析 |
| -Xnomergestopstates | do not merge stop states  不要合并停止状态 |
| -Xdfaverbose | generate DFA states in DOT with NFA configs  用NFA的配置产生dot格式的DFA状态图 |
| -Xwatchconversion | print a message for each NFA before converting  在每一个nfa转换前输出一个消息 |
| -XdbgST | put tags at start/stop of all templates in output  在输出中，放一个标签在 所有的模板 开始/停止 处 |
| -Xm m | max number of rule invocations during conversion  在转换中，规则调用的最大数 |
| -Xmaxdfaedges m | max "comfortable" number of edges for single DFA state  单个dfa状态图的边 最舒适的数量 |
| -Xconversiontimeout t | set NFA conversion timeout for each decision  为每一个决策 设置nfa 转换 超时时间 |
| -Xmaxinlinedfastates m | max DFA states before table used rather than inlining  用内联形式比 用表格形式 要好的dfa 状态的最大数。  注: 超过这个数量最好用table 形式 |
| -Xnfastates | for nondeterminisms, list NFA states for each path  对不确定情况，对每一个路径列出 nfa 状态。 |

Attribute and Dynamic Scopes

# 属性与动态域

# Token attributes

## Token 属性

|  |  |
| --- | --- |
| **attribute 属性** | **description**  **描述** |
| text  文本 |  |
| type  类型 |  |
| line  行号 |  |
| index  索引 |  |
| pos  位置 |  |
| channel  通道 |  |
| tree  树 |  |
| int  整数值 |  |

## Rule attributes

## 规则属性

语法解析器

Parsers

|  |  |
| --- | --- |
| **attribute**  **属性** | **description**  **描述** |
| text  文本 |  |
| start  开始 |  |
| stop  结束 |  |
| tree  树 |  |
| st  字符串模板 |  |

Tree parsers

树语法解析器

|  |  |
| --- | --- |
| **attribute**  **属性** | **description**  **描述** |
| text  文本 |  |
| start  开始 |  |
| tree  树 |  |
| st  字符串模板值 |  |

Lexers

词法解析器

|  |  |
| --- | --- |
| **attribute**  **属性** | **description**  **描述** |
| text  文本 |  |
| type  类型 |  |
| line  行数 |  |
| index  索引 |  |
| pos  位置 |  |
| channel  通道 |  |
| start  开始 |  |
| stop  停止 |  |
| int  整型值 |  |

### The Rule text Attribute in Tree Grammars

### 在树语法中规则的text属性

In a parser grammar, the relationship between the elements matched by a rule and the associated input text is very clear. A rule begins parsing at a particular token and stops parsing at a particular token. The text attribute for a rule, $text, is simply the concatenated text from all tokens in that range, including hidden channel tokens. What does $text mean in a tree grammar, though?

在一个语法解析器的文法中，一个规则匹配的元素和与之关联的输入的文本之间的关系是非常清晰的。一个规则开始，一个规则于某个token的解析终于另一个token的解析。$text是这个规则的文本属性，它是规则范围内的所有token包括隐藏的通道的token的文本的简单拼接，可在树语法中，$text是什么呢?

Tree grammar rules match nodes and trees not tokens. Fortunately, each node has an associated token start and stop index (See TreeAdaptor). As the parser builds trees, each rule sets the token indexes for its return AST to the start and stop token of that rule. We can then define the text attribute for a tree grammar rule to be the text concatenated from the range of tokens indicated by the range in the root of the first tree matched by the rule. This rule may seem strange, but is the most efficient implementation and works in almost all situations. Here are a few examples:

树文法规则匹配节点和树而不是那些token.幸运地是每一个节点有一个关联的token 的start和stop index (参阅 TreeAdaptor)。当语法解析器构建树时，每一个规则为它返回的AST 设置token的索引到该规则的开始token 和结束token. 我们之后可以把树文法规则的text 属性定义成 与规则匹配的第一棵树的根节点下面的token集合的文本的拼接串.这个规则可能看起来奇怪,但却是最高效的实现和几乎在所有情况下适用.这是一个例子.

/\*\* match tree created from, e.g., "int x;"

\* $text would, therefore, be "int x;"

\* $start node is VAR node.

\*/

/\*\* 匹配由例子"int x"创建的树

\* $text 将无疑是"int x"

\* $start节点是 VAR节点

\*/

variable

: ^(VAR type ID) // $text derived from indexes in VAR node

//$text 从VAR节点下的索引获得

;

/\*\* match tree node created from, e.g., "int"

\* $text would, therefore, be "int"

\* $start node is 'int' node.

\*/

/\*\* 匹配由例子"int"创建的树节点

\* $text将无疑是 "int"

\* $start节点是 'int'节点.

\*/

type: 'int' // $text derived from indexes in 'int'

// $text 从'int'节点的索引获得。

node

| 'void'

;

The following code embodies the text attribute definition. The token range from a rule's start node defines the range of text for the entire rule.

下面的代码包含了text属性的定义，token的范围从一个规则的开始节点定义整个规则的文本范围。

// input is a TreeNodeStream implementation

// 输入是一个 treenodestream的实现

int start = input.getTreeAdaptor().getTokenStartIndex($start);

int stop = input.getTreeAdaptor().getTokenStopIndex($start);

String text = input.getTokenStream().toString(start, stop);

Be careful when referencing the text of a rule that happens to be the root of a tree. The text of a rule is the text of all tokens underneath the first rout matched by the rule. In the following example, rule @r op matches a single node, but $op.text will include the text associated with the two operands as well. The parser that build the plus and multiply operator nodes will set the token range to include all tokens for that expression.

当引用一个树的根节点的规则的text时候需要小心些, text是匹配到的规则下面的所有的token。在下面的例子里，规则 @r op 匹配一个单节点，但是  $op.text 将包含关联的两个操作数的text。构建加法和乘法运算符节点的语法解析器将根据那个表达式设置token范围到包含所有的token

/\*\* match subtrees for + and \* created from input such as "1+4\*2"

\* $text and $op.text is "1+4\*2" for first alternative.

\* $text is just the INT node for second alternative.

\*/

/\*\* 在形如"1+4\*2"的输入里面，为+和\* 匹配子树

\* $text 和 $op.text 在第一个可选路径上都是 "1+4\*2"

\* $text 在第二个可选路径上仅仅是INT节点

\*/

expr: ^(op expr expr) ; // $op.text is same as $text!

// $op.text 同于 $text!

| INT

;

op : o='+' | o='\*' // $text includes text of operands

// $text 包含操作数的text

; // $o.text is just node's text

// $o.text 仅只是节点的text

Note that the text for a node label is always just the string returned from getText() invoked on that node whereas the text for a rule reference is always the text for the tree rooted at that labeled node.

请注意，一个节点标签的text通常只是在那个节点上调用getText()返回的字符串，一个规则的text 通常是涉及那个标签节点所在树根的所有text，这两者是有别的。

Finally, here is the case where the definition of the text attribute does not do what you expect. The text attribute is derived from the first node matched by a rule, but a rule such as rule slist that matches multiple subtrees has an ill-defined text attribute because it only gives you the text for the first statement subtree:

最后，这是一个text属性定义不按预期工作的地例子。text属性内容源自规则匹配到的第一个节点，但形如slist这种匹配多个子树的规则有一个不合常规的text属性定义，因为它仅给你第一个语句的子树的text。

func: 'void' ID '()' slist ; // $slist.text is text from first tree only

//$slist.text仅是第一个树的text

slist: stat+ ;

In general, you just need to keep this in mind--the text attribute is natural in most cases.

通常你只需要记住:text属性在大多数情况是正常的。

**Rule scopes**

**规则的作用域**

**Global shared scopes**

**全局共享的作用域**

**Lexical filters**

**词法分析过滤器**

ANTLR has a lexical filter mode that lets you sift through an input file looking for certain grammatical structures. The rules are prioritized in the order specified in case an input construct matches more than a single rule, with the first rule having the highest priority. The filter proceeds character-by-character looking for a match among the rules. If no match, consume that char and try again. The following example, prints found var foo for every field foo in the input:

antlr有一个词法分析过滤模式，能让你从一个输入文件筛分出要找的一些语法结构。这些规则按照特定的顺序排优先次序，第一个规则优先级最高，这样以免一个输入结构匹配多个规则，。过滤器开始一个字符一个字符寻找符合规则的匹配。如果没有合适的匹配，消耗掉那个字符然后重新尝试。下面的例子，对输入中的每一个域foo，打印找到的var foo 。

注:有时候，只对一些敏感语法结构感兴趣，但这部分语法结构所在的整个语法结构很复杂，这时候，使用filter=true 就可以避免写出整个复杂的语法结构，然后再关注感兴趣的部分。比如，抽取函数原形。开启过滤模式后，是按照逐字符匹配的，若没有匹配的规则，就消耗掉这个字符，或者叫略过也行

lexer grammar FuzzyJava;

options {filter=true;}

FIELD

: TYPE WS name=ID '[]'? WS? (';'|'=')

{System.out.println("found var "+$name.text);}

;

fragment

TYPE : ID ('.' ID)\*

;

fragment

ID : ('a'..'z'|'A'..'Z'|'\_') ('a'..'z'|'A'..'Z'|'\_'|'0'..'9')\*

;

WS : (' '|'\t'|'\n')+

;

Don't forget that you must ignore text in comments, so add another rule:

别忘你必须忽略注释中的文本，所以增加另一个规则：

COMMENT

: '/\*' (options {greedy=false;} : . )\* '\*/'

{System.out.println("found comment "+getText());}

;

**Grammars**

语法

**Grammar syntax**

**语法的句法**

All grammars are of the form:

所有的语法是这种格式:

/\*\* This is a grammar doc comment \*/  
/\*\* 这是语法文档注释 \*/  
*grammar-type* grammar *name*;  
[options](http://www.antlr.org/wiki/display/ANTLR3/Grammar+options) { *name1* = *value*; *name2* = *value2*; ... }  
import *delegateName1*=*grammar1*, ..., *delegateNameN*=*grammarN*; // can omit delegateName  
tokens { *token-name1*; *token-name2* = *value*; ... }  
scope *global-scope-name-1* { «*attribute-definitions*» }  
scope *global-scope-name-2* { «*attribute-definitions*» }  
...  
@header { ... }  
@lexer::header { ... }  
@members { ... }

«*rules*»

The type of the grammar, specified via the *grammar-type* modifier above, can be one of: lexer, parser, tree, and combined (no modifier). To set the superclass of the generated parser class, use the superClass option. See [Grammar options](http://www.antlr.org/wiki/display/ANTLR3/Grammar+options) for a list of valid grammar options and their semantics.

语法类型通过上面的grammar-type修饰词指定，可以是lexer，parser，tree 和combined(没有修饰词)之一。要设置产生的语法解析类的超类，用superClass选项。有效的语法选项及其含义的清单请参阅 语法选项

**Rule syntax**

/\*\* rule comment \*/  
/\*\* 规则注释 \*/  
*access-modifier* *rule-name*[«arguments»] returns [«return-values»] throws *name1*, *name2*, ...  
[options](http://www.antlr.org/wiki/display/ANTLR3/Rule+and+subrule+options) {...}  
scope {...}  
scope *global-scope-name*, ..., *global-scope-nameN*;  
@init {...}  
@after {...}  
    : «*alternative-1*» -> «*rewrite-rule-1*»  
    | «*alternative-2*» -> «*rewrite-rule-2*»  
    ...  
    | «*alternative-n*» -> «*rewrite-rule-n*»  
    ;  
    catch [«*exception-arg-1*»] {...}  
    catch [«*exception-arg-2*»] {...}  
    finally {...}

See [Rule and subrule options](http://www.antlr.org/wiki/display/ANTLR3/Rule+and+subrule+options) for a list of valid rule options and their semantics.

有效的规则选项及其含义，参阅 规则与子规则选项

**Lexer rules**

**词法规则**

Rules in a lexical grammar are token names:

在词汇的语法中的规则是token名字:

ID : ('a'..'z'|'A'..'Z'|'\_') ('a'..'z'|'A'..'Z'|'0'..'9'|'\_')\* ;

Here are some common lexical rules for programming languages:

这是一些编程语言常用的词汇规则:

WS : (' '|'\r'|'\t'|'\u000C'|'\n') {$channel=HIDDEN;}

;

COMMENT

: '/\*' .\* '\*/' {$channel=HIDDEN;}

;

LINE\_COMMENT

: '//' ~('\n'|'\r')\* '\r'? '\n' {$channel=HIDDEN;}

;

The $channel=HIDDEN; action places those tokens on a hidden channel. They are still sent to the parser, but the parser does not see them. Actions, however, can ask for the hidden channel tokens. If you want to literally throw out tokens then use action skip(); (see [org.antlr.runtime.Lexer.skip()](http://www.antlr.org/api/Java/classorg_1_1antlr_1_1runtime_1_1_lexer.html#2a0d5c0c2539d27e2a86a0ec45ec784e)).

$channel=HIDDEN;这个语义动作把那些token送到一个隐藏的通道。它们(那些token)仍然送给语法解析器，但语法解析器看不到它们。然而语义动作可以从隐藏的通道中取出那些token。如果你想逐个的推出那些token，那么用动作skip();(参阅[org.antlr.runtime.Lexer.skip()](http://www.antlr.org/api/Java/classorg_1_1antlr_1_1runtime_1_1_lexer.html#2a0d5c0c2539d27e2a86a0ec45ec784e) )

Sometimes you will need some help or rules to make your lexer grammar more readable. Use the fragment modifier in front of the rule:

有时候你会需要一些帮助或者规则让你的词法解析器的语法更可读。那就在规则前面使用fragment修饰词吧。

HexLiteral : '0' ('x'|'X') HexDigit+ ;

fragment HexDigit : ('0'..'9'|'a'..'f'|'A'..'F') ;

In this case, HexDigit is not a token in its own right; it can only be called from HexLiteral.

在这种情况下 右边的HexDigit 不是一个token，它只能被HexLiteral调用。

**Tree grammar rules**

**树语法规则**

Rules in tree grammars are identical to parser grammars except that they can specify a tree element to match. The syntax is ^( *root child1 child2 ... childn* ). For example:

在树语法中的规则和语法解析器的语法规则是一样，除了它们能制定一个树的元素来匹配外。句法是^( *root child1 child2 ... childn* ).例子:

decl : ^(DECL type declarator) {System.out.println($type.text+" "+$declarator.text);}

;

**Attribute scope syntax**

**属性作用域句法**

Attribute scopes are a set of attribute definitions of the form:

属性的作用域是一个按以下格式定义的属性的集合:

scope *name* {  
    *type1* *attribute-name1*;  
    *type2* *attribute-name2*;  
}

**Grammar action syntax**

**文法动作句法**

Grammar actions are, in general, of the form:

文法动作动作通常是这种格式:

@*action-name* { ... }  
@*scope-name*::*action-name* { ... }

The default *scope-name* is parser. For instance, @header is the same as @parser::header. Valid *scope-name* 's differ depending on the [target](http://www.antlr.org/wiki/display/ANTLR3/Code+Generation+Targets), but most targets should support parser and lexer. Two common *action-name* 's are header and members. The header action is placed at the top of a generated class definition and the members action is inserted within the body of a generated class definition. For example, the following grammar actions would ensure generated parser and lexer Java classes include a package declaration:

默认的scope-name是语法解析器的名字，对于语法实例 @header 和 @parser::header是一样的，合法的scope-name因目标代码而不同，但大多数的目标代码应该支持语法解析器和词法解析器。两个公共的action-name是header和members。  
header语义动作放置在生成类定义的顶部，member语义动作插入生成类定义的类体。举个例子下面的语法动作将保证产生的语法解析器和词法解析器的java类包含一个包的申明。

@parser::header { package my.example.package; }，

@lexer::header { package my.example.package; }

**Rule elements**

**规则元素**

Rules may reference:

规则可能涉及到:

|  |  |
| --- | --- |
| Element 元素 | Description  描述 |
| *T* | Token reference. An uppercase identifier; lexer grammars may use optional arguments for fragment token rules.  Token的引用，一个大写的标识符;词法解析器语法可能会为fragment修饰的token规则使用可选的参数 |
| *T*<node=*V*> or *T*<*V*> | Token reference with the optional token option node to indicate tree construction note type; can be followed by arguments on right hand side of -> rewrite rule  带可选项的node 的token的引用。可选项node用来显示树构造的标注类型。在重写规则->的右边时可以跟随参数。 |
| *T*[«args»] | Lexer rule (token rule) reference. Lexer grammars may use optional arguments for fragment token rules.  词法解析器规则(token规则)的引用。对fragment修饰的token规则，词法解析器语法可能使用可选参数。 |
| *r* [«args»] | Rule reference. A lowercase identifier with optional arguments.  规则引用，一个带可选参数的小写的标识符 |
| '«one-or-more-char»' | String or char literal in single quotes. In parser, a token reference; in lexer, match that string.  用单引号引起来的字符串或者字符序列，在语法解析器中，作为一个token引用；在词法解析器中，匹配那个字符串。 |
| {«action»} | An action written in target language. Executed right after previous element and right before next element.  用目标代码语言编写的语义动作，。在前一个元素之后和下一个元素之前执行。  注：语义动作所嵌在的位置决定了它的执行时机，在它所紧跟的规则元素被匹配完成后才会开始执行。 |
| {«action»}? | Semantic predicate.  语义断言  注:它通常被用来做检查，也称校验语义断言(validating semantic predicate),作用类似c语言中的assert宏。 它的形式跟语义动作一样，区别在于，1、它的最终执行结果是boolean值的。2、如果是false它会抛出一个断言异常。  有一种情况，当可选分支的followset是同一个token的时候，放置在可选分支前面的语义断言称为消除二义性断言(Disambiguating Semantic Predicates)，它的作用是帮助选择不同的可选分支，虽然都是同一个token，但决策路径不同。如果信息足够，它会补全未定义语义断言分支的断言。如果信息不够，它会给出警告并短路某些可选分支。 有一种情况需要特别注意的是，在消除二义性断言的情况中，如果可选规则是子规则.那么子规则语义断言也会被合并到当前规则,这会导致子规则的语义断言可能被执行两次，如果该断言改变了全局变量或者改变了上下文相关的内容，两次的结果可能会不一样，如果你这么做了，那就别忘记，当然最好的习惯是别这么做。 |
| {«action»}?=> | Gated semantic predicate.  门语义断言  注:它的作用犹如一个门开关，只有断言成真的时候，其所在的选择分支才会被可见，请注意是可选分支。 它跟语义断言不同的地方是，它的假值不会抛出异常，而是不进行后面的语法规则元素的识别。 |
| («subrule»)=> | Syntactic predicate.  句法断言  注:这个跟前面三个不一样的地方在于,它类似于预先搜索这个规则，如果成立才会继续这个规则元素，同时，断言内的语义动作会被忽略。 |
| («x»|«y»|«z») | Subrule. Like a call to a rule with no name.  子规则，类似调用一个匿名的规则。  注: 子规则用括号括起来，类似一个内容等同的匿名的规则。 |
| («x»|«y»|«z»)? | Optional subrule.  可选的子规则。  注:括号界定它是子规则，问号界定它是可选的，意义同正则。 |
| («x»|«y»|«z»)\* | Zero-or-more subrule.  0个或者多个的子规则。  注:同正则的克林闭包。 |
| («x»|«y»|«z»)+ | One-or-more subrule.  1个或者多个的子规则  注:同正则正闭包。 |
| «x»? | Optional element.  一个可选的元素 |
| «x»\* | Zero-or-more element.  0或者多个可选的元素。 |
| «x»+ | One-or-more element.  1个或者多个元素。 |

**Grammar options**

**语法选项**

Taken from [/org/antlr/tool/Grammar.java](http://fisheye2.atlassian.com/browse/antlr/tool/src/main/java/org/antlr/tool/Grammar.java?r=head) all allowed options are

从[/org/antlr/tool/Grammar.java](http://fisheye2.atlassian.com/browse/antlr/tool/src/main/java/org/antlr/tool/Grammar.java?r=head) 摘取的所有允许的选项是:

|  |  |
| --- | --- |
| Option  选项 | Description  描述 |
| language  语言 | The target language for code generation. Default is Java. See [Code Generation Targets](http://www.antlr.org/wiki/display/ANTLR3/Code+Generation+Targets) for list of currently supported target languages.  代码生成的目标语言。默认是java ，请参阅 代码生成目标语言 中的当前支持的目标语言清单。 |
| tokenVocab  词汇表 | Where ANTLR should get predefined tokens and token types. Tree grammars need it to get the token types from the parser that creates its trees. *TODO: Default value? Example?*  Antlr可以找到预定义的token和token类型的地方。树语法需要它来取得那些语法解析器创建树时的token类型.待做:默认值？例子? |
| output  输出 | The type of output the generated parser should return. Valid values are AST and template. *TODO: Briefly, what are the interpretations of these values? Default value?*  返回产生的解析器的输出的类型，有效地值是AST和template。待做:简短地说，这些值得解释是什么？默认值？ |
| ASTLabelType | Set the type of all tree labels and tree-valued expressions. Without this option, trees are of type Object. *TODO: Cross-reference default impl (*[*org.antlr.runtime.tree.CommonTree*](http://www.antlr.org/api/Java/classorg_1_1antlr_1_1runtime_1_1tree_1_1_common_tree.html) *in Java)?*  设置所有属的标签和树求值得表达式的类型。不设置这个选项，树的类型是object。待做:交叉引用默认实现*(在java中*[*org.antlr.runtime.tree.CommonTree*](http://www.antlr.org/api/Java/classorg_1_1antlr_1_1runtime_1_1tree_1_1_common_tree.html) *)*? |
| TokenLabelType | Set the type of all token-valued expressions. Without this option, tokens are of type [org.antlr.runtime.Token](http://www.antlr.org/api/Java/interfaceorg_1_1antlr_1_1runtime_1_1_token.html) in Java (IToken in C#).  设置所有token求值得表达式的类型，不设置这个选项，token类型在java中是类型[org.antlr.runtime.Token](http://www.antlr.org/api/Java/interfaceorg_1_1antlr_1_1runtime_1_1_token.html) (在C#是IToken) |
| superClass | Set the superclass of the generated recognizer. *TODO: Default value (*[*org.antlr.runtime.Parser*](http://www.antlr.org/api/Java/classorg_1_1antlr_1_1runtime_1_1_parser.html) *in Java)?*  设置产生的识别器的超类。待做:缺省值(在java中[*org.antlr.runtime.Parser*](http://www.antlr.org/api/Java/classorg_1_1antlr_1_1runtime_1_1_parser.html)) |
| filter | In the lexer, this allows you to try a list of lexer rules in order. The first one that matches, wins. This is the token that nextToken() returns. If nothing matches, the lexer consumes a single character and tries the list of rules again. See [Lexical filters](http://www.antlr.org/wiki/display/ANTLR3/Lexical+filters) for more.  在词法解析器，这个允许尝试有序尝试词法规则列表。被匹配的第一个赢。这个就是nexttoken()返回的token。如果没被匹配到，词法解析器消耗一个字符，然后再尝试这个规则列表。更多请参阅[Lexical filters](http://www.antlr.org/wiki/display/ANTLR3/Lexical+filters). |
| rewrite | Valid values are true and false. Default is false. Use this option when your translator output looks very much like the input. Your actions can modify the TokenRewriteStream to insert, delete, or replace ranges of tokens with another object. Used in conjunction with output=template, you can very easily build translators that tweak input files.  合法值是true和false.默认是false.当你的翻译器的输出和输入非常像的时候，使用这个选项。你的动作可以修改TokenRewriteStream来插入，删除，或者用其他对象来替换某个范围的token。和output=template联用的时候，你能很轻易构建变换输入文件的翻译器。 |
| k | Limit the lookahead depth for the recognizer to at most k symbols. This prevents the decision from using acyclic LL\* DFA.  限制识别器向前看的深度最多k个字符。这个阻止决策使用非循环的LL\*DFA。 |
| backtrack | Valid values are true and false. Default is false. Taken from <http://www.antlr.org:8080/pipermail/antlr-interest/2006-July/016818.html> : The new feature (a big one) is the backtrack=true option for *grammar*, *rule*, and *block* that lets you type in any old crap and ANTLR will backtrack if it can't figure out what you meant. No errors are reported by antlr during analysis. It implicitly adds a syn pred in front of every production, using them only if static grammar LL\* analysis fails. Syn pred code is not generated if the pred is not used in a decision. This is essentially a rapid prototyping mode. It is what I have used on the java.g. Oh, it doesn't memoize partial parses (i.e. rule parsing results) during backtracking automatically now. You must also say memoize=true. Can make a HUGE difference to turn on.  合法值是true或者false。默认实false。选自<http://www.antlr.org:8080/pipermail/antlr-interest/2006-July/016818.html> :一个新的大的特征是为语法，规则，和块设立的backtrack=true选项，它能让你输入任何老套的废话，如果antlr不能理解你的意思，它将会backtrack(回溯)。在antlr分析期间，不会报告错误。它隐式的增加一个语法断言在每一个产生式的前面，只有在静态语法LL\*分析失效的时候才会使用它们。如果决策中用不到断言，那么不会产生语法断言的代码。这本是一个快速原形模式，我曾在java.g中用的就是这玩意儿。噢，在自动回溯的过程中，它并不记住部分的分析(也即规则分析的结果)。你必须还要对语法解析器说memoize=true。打开这个选项能产生很大的差别。 |
| memoize | Valid values are true and false. When backtracking, remember whether or not rule references succeed so that the same input position cannot be parsed more than once by the same rule. This effectively guarantees linear parsing when backtracking at the cost of more memory. *TODO: Default value (false)?*  合法值是true和false.当回溯时，记住规则引用是否成功，这样，同一规则对同样的输入位置就不用分析多次。事实上在回溯时这用更多的内存空间来保证线性分析。 |

**Rule and subrule options  
规则和子规则选项**

|  |  |
| --- | --- |
| option  选项 | description  描述 |
| k | Specify the exact lookahead to be used by the subrule.  指定子规则用到的向前看的精确深度。 |
| greedy | Valid values are true and false. Default is true. Normally symbols are matched greedily so that if a symbol can be matched immediately or by exiting the subrule the parser chooses to match the symbol immediately.  有效值是true和false.缺省值是true.正常情况下，符号被贪婪地匹配以便如果一个符号可以被立即匹配或者在离开子规则时语法解析器选择立刻匹配符号。  注: 不明所言。根据译者对贪婪模式和非贪婪模式的了解，贪婪模式是在保证整体匹配成功的情况下，尽可能多地匹配，非贪婪模也是在保证整体匹配成功的情况下尽可能少的匹配。 |
| backtrack | Rule-specific version of the backtrack grammar option. See [Grammar options](http://www.antlr.org/wiki/display/ANTLR3/Grammar+options) for details.  特殊规则版本的backtrack语法选项，详情请参阅[Grammar options](http://www.antlr.org/wiki/display/ANTLR3/Grammar+options) |
| memoize | Rule-specific version of the memoize grammar option. See [Grammar options](http://www.antlr.org/wiki/display/ANTLR3/Grammar+options) for details.  特殊规则版本的memozie语法选项，详情请参阅[Grammar options](http://www.antlr.org/wiki/display/ANTLR3/Grammar+options) |

**Special symbols in actions  
动作中的特定的符号**

This table describes the complete set of special symbols you can use in actions within your grammar. These are translated by the codegen/action.g ANTLR v3 grammar (in filter mode). The rules mentioned below are found in action.g

这个表格描述了在你的语法动作中使用到的特殊符号的全集。它们是 codegen/action.g antlr v3 语法(过滤器模式)来翻译的。

|  |  |
| --- | --- |
| Syntax  句法 | Description  描述 |
| $enclosingRule.attr | x is enclosing rule, y is a return value, parameter, or predefined property. Rule ENCLOSING\_RULE\_SCOPE\_ATTR. x是封闭规则，y是一个返回值，参数，或者预定义属性。  r[int i] returns [int j]  : {$r.i, $r.j, $r.start, $r.stop, $r.st, $r.tree}  ;  注:我凌乱了... 我感觉文档错了... 这一段没有提及xy什么的。 |
| $tokenLabel.prop $tokenRef.prop | token scope attribute. Rule TOKEN\_SCOPE\_ATTR.  token域属性。 |
| $rulelabel.attr $ruleref.attr | Rule RULE\_SCOPE\_ATTR. |
| $label | either a token label or token/rule list label like label+=expr. Rule LABEL\_REF.  一个token标签或者形如label+=expr的标签列表 |
| $tokenref | in a non-lexer grammar ISOLATED\_TOKEN\_REF  用在非词法解析器语法中 |
| $lexerruleref | Yields a Token object created from that rule or fragment rule. Rule ISOLATED\_LEXER\_RULE\_REF.  从规则或者fragment修饰的规则产生一个token对象。 |
| $y | return value, parameter, predefined rule property, or token/rule reference within enclosing rule's outermost alt. y must be a "local" reference; i.e., it must be referring to something defined within the enclosing rule. Rule LOCAL\_ATTR. 在封闭规则的最外头的可选分支内的返回值，参数，预定义规则属性或者token/规则等的引用。  Y必须是一个“本地”引用，也就是，它必须引用那些定义在封闭规则内的东西。  r[int i] returns [int j]  : {$i, $j, $start, $stop, $st, $tree}  ; |
| $x::y | the only way to access the attributes within a dynamic scope regardless of whether or not you are in the defining rule. Rule DYNAMIC\_SCOPE\_ATTR.  这是访问动态作用域内的属性的唯一方法而不管是作用域否在定义的规则内。  scope Symbols { List names; }  r  scope {int i;}  scope Symbols;  : {$r::i=3;} s {$Symbols::names;}  ;  s : {$r::i; $Symbols::names;}  ; |
| $x[-1]::y | previous (just under top of stack). Rule DYNAMIC\_NEGATIVE\_INDEXED\_SCOPE\_ATTR.  前一个(仅在堆栈顶元素下)  注:这个一般都是嵌套规则对全局共享域的引用。 解析器为每一个全局共享作用域维护一个堆栈，每一个对全局共享作用域引用的规则，解析器都会为其初始化一个和全局共享作用域结构一样的对象，并压入当前栈。离开该规则时，弹出堆栈。 |
| $x[-i]::y | top of stack - i where the '-' MUST BE PRESENT; i.e., i cannot simply be negative without the '-' sign! Rule DYNAMIC\_NEGATIVE\_INDEXED\_SCOPE\_ATTR.  栈顶-i处，'-'号必须有。也即 没有'-'号就没有负数。 |
| $x[i]::y | absolute index i (0..size-1). Rule DYNAMIC\_NEGATIVE\_INDEXED\_SCOPE\_ATTR.  绝对的索引i(0..size-1)的位置 |
| $x[0]::y | is the absolute 0 indexed element (bottom of the stack). Rule DYNAMIC\_NEGATIVE\_INDEXED\_SCOPE\_ATTR.  绝对位置0索引的元素(堆栈底部) |
| $x.size() | returns the size of the current stack of the scope. Note: This particular syntax is target-dependent. Look at the target page for other targets than Java. 返回作用域当前堆栈的大小。注:这个特殊的语法是依赖于目标平台的。其它的非java目标的请查看目标平台页面。 |
| $r | r is a rule's dynamic scope or a global shared scope. Isolated $rulename is not allowed unless it has a dynamic scope **and** there is no reference to rulename in the enclosing alternative, which would be ambiguous. Rule ISOLATED\_DYNAMIC\_SCOPE.  R是一个规则的动态作用域或者一个全局共享作用域。独立的$rulename是不允许的，除非他有一个动态域，并且在封闭可选分支内没有对该规则名的能引起二义性的引用。 |

The following symbols relate to StringTemplate templates.  
以下的符号和String Template的模板有关。

|  |  |
| --- | --- |
| Syntax  句法 | Description  描述 |
| %foo(a={},b={},...) | Create instance of template foo, setting attribute arguments. Rule TEMPLATE\_INSTANCE.  创建一个模板foo的实例，设置属性参数。 |
| %({name-expr})(a={},...) | indirect template ctor reference. Rule INDIRECT\_TEMPLATE\_INSTANCE.  间接模板构造引用。 |
| %x.y = z; | set template attribute y of x (always set never get attr) to z [languages like python without ';' must still use the ';' which the code generator is free to remove during code gen]. Rule SET\_ATTRIBUTE.  设置模板x的属性y(总是设置属性而不会读取)成z.[语言像python那样的没有';',必须还得用';' 因为代码生成器生成代码时可以自由去掉] |
| %{expr}.y = z; | template attribute y of StringTemplate-typed expr to z. Rule SET\_EXPR\_ATTRIBUTE.  设置字符模板类型的表达式expr所指的模板的属性y 成z。 |
| %{string-expr} | anonymous template from string expr. Rule TEMPLATE\_EXPR.  字符串表达式string-expr所引用的匿名模板。 |

**Template construction  
模板构造**

ANTLR v3 has built-in support for constructing StringTemplate templates. there are two forms: [Special symbols in actions](http://www.antlr.org/wiki/display/ANTLR3/Special+symbols+in+actions) and rewrite rules similar to AST construction. I am including a number of rules from the mantra example.

antlr v3 内置支持构造String Template 模板。有两种形式: 在语义动作中的特殊符号和类似AST构造那样重写规则，我正在将一些经典的例子的规则包含进来。

Sometimes you just need a string to become a template:  
有时，你只需要一个字符串成为一个模板:

'void' -> {%{"void"}}

The following tree grammar rule illustrates some of the basic rewrite rules:  
下面的树语法规则展示了一些基本的重写规则。

primary

: ID -> {%{$ID.text}}

// create template from token text

//从token的文本中创建模板。

// create template using rule results as template attributes

//使用规则结果作为模板属性来创建模板。

| ^('new' typename args=expressionList)

-> new(type={$typename.st},args={$args.st})

| listliteral -> {$listliteral.st}

// reuse template built for listliteral

//为listliteral重用模板构建

// create template using token text as template attribute

//使用token的text作为末班属性来创建模板。

| NUM\_INT -> int\_literal(v={$NUM\_INT.text})

;

And here are some more complicated examples:  
这是一些更复杂的例子:

assignment

: // special case "a[i] = expr;"

//特殊的例子"a[i] = expr;"

^('=' ^(EXPR ^(INDEX a=expression i=expression)) rhs=completeExpression)

-> indexed\_assignment(list={$a.st}, index={$i.st}, rhs={$rhs.st})

| ^('=' lvalue completeExpression)

-> assignment(

lhs={$lvalue.st},

rhs={$completeExpression.st})

| ^(assign\_op lvalue completeExpression)

-> assignment\_with\_op(

type={$assign\_op.start.type.name},

op={$assign\_op.text},

lhs={$lvalue.st},

rhs={$completeExpression.st})

;

When you need to append multiple strings or templates into a another template use the += operator for a rule's return value (former use of toTemplates no longer required). For example adding variable declarations inside a struct template :

当你需要增加多个字符串或者模板到另一个模板的时，在规则返回值处使用 +=操作符(以前的toTemplates的用法不再需要了)。举个在结构体模板内增加变量申明的例子：

structDeclaration

: name=Ident (decls+=typeDecls)+ -> structDecl(name={$name.text},declList={$decls});

    ;

and the templates for a struct declaration may be something like this :  
结构体声明的模板可能是像这样的:

structDecl(name,declList) ::= <<

struct <name> {

<declList; separator="\n">

}

More on string templates can be found here: [String Template](http://www.antlr.org/wiki/display/ST/StringTemplate+3.1+Documentation)  
更多的关于字符串模板的能在这里找到: [String Template](http://www.antlr.org/wiki/display/ST/StringTemplate+3.1+Documentation)

注:这坑爹的文档...貌似就不想让人学会怎么使用...我找源代码翻了翻，大约总结出用法。  
首先你要在options里面设置 output=template . 然后定义一个模板文件stg，把你需要用到的模板定义好，才能够在.g文件中使用。当设置为output=template时，antlr会为.g文件生成一个templateLib的setter和getter.这个就是把你定义的模板文件和语法解析器连接起来东西，根据stg文件实例化出一个String Template Group对象，然后setTemplateLib()。  
在每一个语法规则返回值里面，都生成一个getTemplate()，可以使用这种方式，当然规则返回值的toString函数也实现了把模板转成字符串，之所以定义两种方式是因为有时候并不是需要字符串这种最终结果。

**Tree construction  
树构造**

There are two mechanisms in v3 for building abstract syntax trees (ASTs): operators and rewrite rules.   
在v3中有两种机制用来构建抽象语法树：运算符和重写规则。

**Operators  
运算符**

Nodes created for unmodified tokens and trees for unmodified rule references are added to the current subtree as children.  
为不变的tokens创建节点和为不变的规则引用创建树，并做为子节点加入到当前子树。

|  |  |
| --- | --- |
| Operator 运算符 | Description  描述 |
| ! | do not include node or subtree (if referencing a rule) in subtree  不包含节点或者子树(如果是引用一个规则的情况)到子树。 |
| ^ | make node root of subtree created for entire enclosing rule even if nested in a subrule  设置子树的根节点，这颗子树从整个封闭规则甚至是嵌套在子规则中的封闭规则创建。 |

additiveExpression

: multiplicativeExpression ('+'^ multiplicativeExpression)\*

;

That is the same as the following in rewrite notation from the following section:  
那(这？)和紧跟着的下一节的重写标记一样。

additiveExpression

: (a=multiplicativeExpression->$a) // set result

//设置结果

( '+' b=multiplicativeExpression

-> ^('+' $additiveExpression $b)

// use previous rule result

//使用前一个规则结果

)\*

;

注:这一段颇有一点难以理解，它的语法功能是 连加，解析过程如下:只有一个multiplicativeExpression 的时候，结果$additiveExpression 为 $a, 有两个的时候，令第二个为$b\_0 则结果为 ^('+' $additiveExpression $b\_0) 也即是^('+' $a $b\_0)，有三个的时候,令第三个为 $b\_0 ，则结果为 ^('+' $additiveExpression $b\_1) 也即是 ^('+' ^('+' $a $b\_0) $b\_1) 如果把括号内第一个元素当做根节点，就可以构造出一个连加的抽象语法树,它是一个二叉树。

**Rewrite rules  
重写规则**

The rewrite syntax is more powerful than the operators. It suffices for most common tree transformations.  
While the parser grammar specifies how to recognize input, the rewrites are generational grammars, specifying how to generate output. ANTLR figures out how to map input to output grammar. To create an imaginary node, just mention it like the following example (UNIT is a node created from an imaginary token and is used to group the compilation unit chunks):

重写语法比运算符更强大，对大多数常用的树变换来说足够了。  
正如语法解析器语法是指定如何识别输入流的一样，重写也是一类指定如何产生输出的语法。

compilationUnit

: packageDefinition? importDefinition\* typeDefinition+

-> ^(UNIT packageDefinition? importDefinition\* typeDefinition\*)

;

ANTLR tracks all elements with the same name into a single implicit list:  
antlr用同一个名字标记所有的元素到单个隐式的列表。

formalArgs

: formalArg (',' formalArg)\* -> formalArg+

|

;

If the same rule or token is mentioned twice you generally must label the elements to distinguish them. If you want to combine multiple elements into a single list, list labels are very handy (though in this case since they have the same name ANTLR will automatically combine them):  
如果 同样的规则或者token被提到两次，通常你必须打上标签以便于识别。如果你要组合多个元素城一个单个列表，列出标签是非常方便的(虽然在这个例子里面因为他们有相同的名字，antlr会自动的组合他们。)

('implements' i+=typename (',' i+=typename)\*)?

注：亲...你看到那个i了么...对它就是typename被打上的标签。由于使用了+=符号， 它在两处所代表的typename被自动组合了。

Here is the entire rule:  
这是整个规则：

classDefinition[MantraAST mod]

: 'class' cname=ID

('extends' sup=typename)?

('implements' i+=typename (',' i+=typename)\*)?

'{'

( variableDefinition

| methodDefinition

| ctorDefinition

)\*

'}'

-> ^('class' ID {$mod} ^('extends' $sup)? ^('implements' $i+)?

variableDefinition\* ctorDefinition\* methodDefinition\*

)

;

Note that using a simple action in a rewrite means evaluate the expression and use as a tree node or subtree. The mod argument is a set of modifiers passed in from an enclosing rule.  
请注意，在重写中使用单个语义动作意味着对表达式求值并把结果当做树节点或者子树。mod参数是从封闭规则传入的一个修饰符集合。

Deleting tokens or rules is easy: just don't mention them:  
删除token或者规则是很容易的:只要不提及他们就可以了。

packageDefinition

: 'package' classname ';' -> ^('package' classname)

;

If you need to build different trees based upon semantic information, use a semantic predicate:  
如果你需要根据语义信息构建不同的树，使用语义断言:

variableDefinition

: modifiers typename ID ('=' completeExpression)? ';'

-> {inMethod}? ^(VARIABLE ID modifiers? typename completeExpression?)

-> ^(FIELD ID modifiers? typename completeExpression?)

;

where inMethod is set by the method rule.  
这里的inMethod在method规则中被设置了。

Often you will need to build a tree node from an input token but with the token type changed:  
你经常会需要从一个输入的token构造一个树节点，但树节点的token的类型和以前不一样。

compoundStatement

: lc='{' statement\* '}' -> ^(SLIST[$lc] statement\*)

;

SLIST by itself is a new node based upon token type SLIST but it has no line/column information nor text. By using SLIST[$lc], all information except the token type is copied to the new node.

SLIST自身是一个token类型是SLIST的新节点，只是他没有行列信息，也没有文本。通过使用SLIST[$lc]，除了token类型外的所有信息都拷贝到新节点。

Using a rewrite rule at a non-extreme-right-edge-of-production location is ok, but it still always sets the overall subtree for the enclosing rule.  
在产生式非最右边的位置使用重写规则是可以的，但对于那个封闭规则，它还是设置整个的子树。

'if' '(' equalityExpression ')' s1=statement

( 'else' s2=statement -> ^('if' ^(EXPR equalityExpression) $s1 $s2)

| -> ^('if' ^(EXPR equalityExpression) $s1)

)

注:亲，括号里面的两个可选分支，虽然所处的位置是一个子规则里面，但是它们构造的树，是对整个规则而言的。

You may reference the previous subtree for the enclosing rule using $rulename syntax  
对于封闭规则，你可以使用$rulenam语法来引用前面子树。

postfixExpression

: (primary->primary) // set return tree

( lp='(' args=expressionList ')' -> ^(CALL $postfixExpression $args)

| lb='[' ie=expression ']' -> ^(INDEX $postfixExpression $ie)

| dot='.' p=primary -> ^(FIELDACCESS $postfixExpression $p)

| c=':' cl=closure[false] -> ^(APPLY ^(EXPR $postfixExpression) $cl)

)\*

;

**Imaginary nodes  
虚构节点**

References to tokens with rewrite not found on left of -> are imaginary tokens.  
重写规则中引用到的但在->左边没有找到的的token 就是虚构节点。

d : type ID ';' -> ^(DECL type ID) ; // DECL is imaginary

or

call : lp='(' ID args ')' -> ^(CALL[$lp] ID args) ;

Here, the CALL node has its line/column info set with info from '(' token. CALL node is "derived" from the '('.  
这里，CALL节点有它的行/列信息，那个信息是用'('token信息设置的。CALL节点继承自'('.

Even tokens referenced within alternative result in nodes disassociated with tokens from left of -> if you put arguments on the references:  
每一个在可选分支被引用到的token，如果你在引用上加参数，会导致节点和->左边的token脱离关联。

a : INT -> INT["99"] ; // node created from adaptor.create(INT, "99")

// 从adaptor.create(INT,"99") 创建节点。

**Tree construction during tree parsing  
在树解析时的树构造**

ANTLR 3.0.1 could not create trees during tree parsing. 3.1 introduces the ability to create a new AST from an incoming AST using rewrites rules:  
antlr 3.0.1在树分析期间不能创建树，3.1引入使用重写规则从源抽象语法树创建一个新的抽象语法树的能力。

* Each rule returns a new tree.
* An alternative without a rewrite duplicates the incoming tree.
* The tree returned from the start rule is the new tree.
* The new tree created with output=AST in a tree grammar is completely independent of the input tree as all nodes are duplicated (with and without rewrite -> operator).
* 每一个规则返回一个树
* 一个没有重写规则的可选分支原样复制源树
* 开始规则返回的树是一个新树
* 在树语法中使用 output=AST 创建的新树 完全独立于输入的树，所有都是拷贝的（不管有没有重写->运算符）

The rewrites work just like they do for normal parsing:  
在正常的解析中，重写规则一般像下面这样工作:

a : INT ; // duplicate INT node and return

//复制INT节点并返回

a : ID -> ; // delete ID node from tree

//从树中删除ID节点

a : INT ID -> ID INT ; // reorder nodes

//重新排序节点

a : ^(ID INT) -> ^(INT ID) ; // flip order of nodes in tree

//在树中翻转节点顺序。

a : INT -> INT["99"] + // make new INT node

//创建一个新INT节点

a : (^(ID INT))+ -> INT+ ID+ ; // break apart trees into sequences

//把一棵树分解成序列

Predicates can be used to choose between rewrites as well:  
断言可以用来在重写规则中选择，像下面这样:

a : ^(ID INT) -> {some test}? ^(ID["ick"] INT)

-> INT

;

Don't forget the wildcard (smile)  
别忘了通配符

s : ^(ID c=.) -> $c ;

// new tree is whatever matched wildcard  
//新树由通配符匹配的任意的东西。

**Polynomial differentiation example  
多项式微分的例子**

For translations whose input and output languages are the same, it often makes sense to build a tree and them morph it towards the final output tree, which can then be converted to text. Polynomial differentiation is a great example of this. Recall that:  
对输入和输出语言都是一样的翻译来说，它的意义通常是构建一棵树然后变形成可以变换成文本的最终输出树。多项式微分是这方面一个很好地例子。记得那些变换公式:

* d/dx(n) = 0
* d/dx(x) = 1
* d/dx(nx) = n
* d/dx(nx^m) = nmx^m-1
* d/dx(foo + bar) = d/dx(foo) + d/dx(bar)

Ok, here's a parser that builds nice trees.  
好了，这是一个构建新树的规则解析器。

grammar Poly;

options {output=AST;}

tokens { MULT; } // imaginary token

//虚构token

poly: term ('+'^ term)\*

;

term: INT ID -> ^(MULT["\*"] INT ID)

| INT exp -> ^(MULT["\*"] INT exp)

| exp

| INT

| ID

;

exp : ID '^'^ INT

;

ID : 'a'..'z'+ ;

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

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

Then we differentiate:  
然后我们微分:

tree grammar PolyDifferentiator;

options {

tokenVocab=Poly;

ASTLabelType=CommonTree;

output=AST;

// rewrite=true; // works either in rewrite or normal mode

//在重写或者正常模式都能工作

}

poly: ^('+' poly poly)

| ^(MULT INT ID) -> INT

| ^(MULT c=INT ^('^' ID e=INT))

{

String c2 = String.valueOf($c.int\*$e.int);

String e2 = String.valueOf($e.int-1);

}

-> ^(MULT["\*"] INT[c2] ^('^' ID INT[e2]))

| ^('^' ID e=INT)

{

String c2 = String.valueOf($e.int);

String e2 = String.valueOf($e.int-1);

}

-> ^(MULT["\*"] INT[c2] ^('^' ID INT[e2]))

| INT -> INT["0"]

| ID -> INT["1"]

;

then we simplify (a little anyway):  
然后我们化简(不管如何多少有点)

tree grammar Simplifier;

options {

tokenVocab=Poly;

ASTLabelType=CommonTree;

output=AST;

backtrack=true;

// rewrite=true; // works either in rewrite or normal mode

//在重写或者正常模式都能工作

}

/\*\* Match some common patterns that we can reduce via identity

\* definitions. Since this is only run once, it will not be perfect.

\* We'd need to run the tree into this until nothing

\* changed to make it correct.

\*/

/\*\* 匹配一些通用的模式，一些我们能够通过特征定义来精简(表达式)的模式。

\* 因为这个仅止运行一次，所以它不会完美。

\* 我们需要把这棵树运行到没有什么可以再调整。

\*/

poly: ^('+' a=INT b=INT) -> INT[String.valueOf($a.int+$b.int)]

| ^('+' ^('+' a=INT p=poly) b=INT)

-> ^('+' $p INT[String.valueOf($a.int+$b.int)])

| ^('+' ^('+' p=poly a=INT) b=INT)

-> ^('+' $p INT[String.valueOf($a.int+$b.int)])

| ^('+' p=poly q=poly)-> {$p.tree.toStringTree().equals("0")}? $q

-> {$q.tree.toStringTree().equals("0")}? $p

-> ^('+' $p $q)

| ^(MULT INT poly) -> {$INT.int==1}? poly

-> ^(MULT INT poly)

| ^('^' ID e=INT) -> {$e.int==1}? ID

-> {$e.int==0}? INT["1"]

-> ^('^' ID INT)

| INT

| ID

;

Finally we walk the tree to print it back out using simple templates:  
最后遍历这个树，用简单模板打印，推出。

tree grammar PolyPrinter;

options {

tokenVocab=Poly;

ASTLabelType=CommonTree;

output=template;

}

poly: ^('+' a=poly b=poly) -> template(a={$a.st},b={$b.st}) "<a>+<b>"

| ^(MULT a=poly b=poly) -> template(a={$a.st},b={$b.st}) "<a><b>"

| ^('^' a=poly b=poly) -> template(a={$a.st},b={$b.st}) "<a>^<b>"

| INT -> {%{$INT.text}}

| ID -> {%{$ID.text}}

;

Here is a test rig:  
这是测试的脚手架:

[?](http://www.antlr.org/wiki/display/ANTLR3/ANTLR+v3+printable+documentation)

|  |
| --- |
| import org.antlr.runtime.\*;  import org.antlr.runtime.tree.\*;    public class Main {      public static void main(String[] args) throws Exception {          CharStream input = null;          if ( args.length>0 ) {              input = new ANTLRFileStream(args[0]);          }          else {              input = new ANTLRInputStream(System.in);          }            // BUILD AST  //构建抽象语法树          PolyLexer lex = new PolyLexer(input);          CommonTokenStream tokens = new CommonTokenStream(lex);          PolyParser parser = new PolyParser(tokens);          PolyParser.poly\_return r = parser.poly();          System.out.println("tree="+((Tree)r.tree).toStringTree());            // DIFFERENTIATE  //微分          CommonTreeNodeStream nodes = new CommonTreeNodeStream((Tree)r.tree);          nodes.setTokenStream(tokens);          PolyDifferentiator differ = new PolyDifferentiator(nodes);          PolyDifferentiator.poly\_return r2 = differ.poly();          System.out.println("d/dx="+((Tree)r2.tree).toStringTree());            // SIMPLIFY / NORMALIZE  //化简 /标准化          nodes = new CommonTreeNodeStream((Tree)r2.tree);          nodes.setTokenStream(tokens);          Simplifier reducer = new Simplifier(nodes);          Simplifier.poly\_return r3 = reducer.poly();          System.out.println("simplified="+((Tree)r3.tree).toStringTree());          // CONVERT BACK TO POLYNOMIAL  //转回到多项式          nodes = new CommonTreeNodeStream((Tree)r3.tree);          nodes.setTokenStream(tokens);          PolyPrinter printer = new PolyPrinter(nodes);          PolyPrinter.poly\_return r4 = printer.poly();          System.out.println(r4.st.toString());      }  } |

Running the rig on "2x+3x^5" shows:  
在"2x+3x^5"上运行脚手架代码将显示:

tree=(+ (\* 2 x) (\* 3 (^ x 5)))

d/dx=(+ 2 (\* 15 (^ x 4)))

simplified=(+ 2 (\* 15 (^ x 4)))

2+15x^4

**Rewriting an existing AST  
重写一个存在的抽象语法树**

For efficiency, option rewrite=true does an in-line replacement for rewrite rules so you can avoid making a copy of an entire tree just to tweak a few nodes. For example, if you have a huge expression tree but only want to rewrite ^('+' INT INT) to be a single INT node, it's better not to duplicate the entire huge tree. The rewrite mode behaves exactly the same as nonrewrite mode except that rewrites stitch changes into the incoming tree. Nodes are not duplicated for rules w/o rewrites.

基于效率考虑，选项rewrite=true在重写规则时做一个在线的替换，所以你能避免仅微调几个节点就拷贝整个树。举一个例子，如果你有一个巨大的表达式树，但只是想要重写^('+' INT INT) 成单个的INT节点，最好是别复制整个巨大的树。除了重写模式对源树进行修修补补外，重写模式和非重写模式在行为几乎是一样的。对于不带重写的规则，不会复制节点。

The result of a rule with a rewrite is the newly created tree. The result of a rule w/o a rewrite is simply the incoming tree. For chains of rule invocations as in the next example, ANTLR copies rewrites upwards so that the action in rule 's' prints out the tree created in b:  
带重写的规则的结果是一个新近创建的树。不带重写的规则保持和源树一样。对在下一个例子中的规则调用链，antlr 重写规则是自底向上拷贝的，所以，在规则's'的语义动作里面可打印在b中创建的树。

tree grammar TP;

options {output=AST; ASTLabelType=CommonTree; tokenVocab=T; rewrite=true;}

s : a {System.out.println($a.tree.toStringTree());} ;

a : b ;

b : ID INT -> INT ID ;

**Heterogeneous tree nodes  
异构的树节点**

By default, with output=AST, ANTLR creates trees of type CommonTree. To create different nodes depending on the incoming token type, you can override create(Token) and YourTreeClass.dupNode(Object) and errorNode() in a subclass of CommonTreeAdaptor or implement your own TreeAdaptor. Unfortunately, this only allows you to change the node type based upon the token type and not grammatical context. Sometimes you want to have ID become a VarNode and sometimes they MethodNode object. As of v3.1, you can use the node token option to indicate node type (in both parsers **and** tree parsers):

在output=AST的默认情况下，antlr创建类型为CommonTree的树。要想创建不同的节点依赖于来源的token类型。你可以在CommonTreeAdaptor的子类中重写create(Token)和 YourTreeClass.dupNode(Object)和errorNode()，或者在你自己的TreeAdaptor中实现这些方法。不幸的是，这个只允许你在token类型上改变节点类型并且没有语法环境。有时候你有一个ID要变成VarNode对象，有时候要变成MethodNode对象。在v3.1中，你可以用节点token选项来指明类型(在语法解析器和树语法解析器中皆可)

decl : 'int'<node=TypeNode> ID<node=VarNode> ';' ;

or equivalently  
或等价的:

decl : 'int'<TypeNode> ID<VarNode> ';' ;

because node is assumed if there is only one option and it is not an option assignment. Token references with node options and invoke the following constructor during tree construction:  
因为节点假定如果只有一项并不是赋值时，token带着节点的选项，并调用下面的构造函数

[?](http://www.antlr.org/wiki/display/ANTLR3/ANTLR+v3+printable+documentation)

|  |
| --- |
| public V(Token t); // NEED SPECIAL CTOR for ID<V> on left of ->  //在->左边的ID<V>需要特殊的构造函数 |

You can specify the node type on any token reference including liberals:  
你能在任何的token引用包括自由出现的字符序列上指定节点类型。

a : ID<V> ';'<V>

The "become root" operator ^ is used following the token options:  
置根操作符^在下面的token的选项中使用:

e : INT '+'<PlusNode>^ INT ;

Labels are available as usual; e.g.,  
标签和平常一样其作用  
 x=ID<V> and x+=ID<V>.

Heterogeneous tree nodes are labeled on the right-hand side of the -> rewrite operator as well:  
异构树节点类型也可以在重写运算符->的右边标记出来。

decl : 'int' ID -> ^('int'<TypeNode> ID<VarNode>) ;

You can also specify arguments on node type constructors on the right of -> rewrite operator. For example, the following two token references:  
你还可以指定参数在重写运算符->右边的节点类型构造函数上。举个例子，下面的两个token引用:

ID<V>[42,19,30] ID<V>[$ID,99]

invoke the following two constructors of V:  
调用下面的两个V的构造函数

[?](http://www.antlr.org/wiki/display/ANTLR3/ANTLR+v3+printable+documentation)

|  |
| --- |
| public V(int ttype, int x, int y, int z)  public V(int ttype, Token t, int x) |

The TreeAdaptor is not called; instead for constructors are invoked directly. This is much more flexible because the list of arguments can change per type whereas the TreeAdaptor interface is fixed. Note that parameters are not allowed on token references to the left of ->:  
treeAdptor没有被调用，取而代之的是构造函数被直接调用。这样会灵活的多，因为参数列表能能根据每一个类型改变，而treeAdptor接口是固定的。注意,在->的左边的token引用不允许有参数。

a : ID<V>[23,21] ; // ILLEGAL

//不合法

Use imaginary nodes as you normally would, but with the addition of the node type:  
可以和往常一样使用虚构节点，但要带上节点类型这个条件限制。

block : lc='{' stat+ '}' -> ^(BLOCK<StatementList>[$lc] stat+) ;

Here is a complete simple example:  
这是一个完整的简单的例子:

grammar T;

options {output=AST;}

@members {

static class V extends CommonTree {

public int x,y,z;

public V(int ttype, int x, int y, int z) {

this.x=x; this.y=y; this.z=z; token=new CommonToken(ttype,"");

}

public V(int ttype, Token t, int x) { token=t; this.x=x; }

public String toString() {

return (token!=null?token.getText():"")+"<V>;"+x+y+z;

}

}

}

a : ID -> ID<V>[42,19,30] ID<V>[$ID,99] ;

ID : 'a'..'z'+ ;

WS : (' '|'\\n') {$channel=HIDDEN;} ;

Sometimes ANTLR must duplicate nodes to avoid cycles and to provide useful semantics. In the next example, the trees returned from rule type are type V. There is only one type specification in the input (e.g., "int a,b,c;") but multiple identifiers. to create multiple trees with 'int' at the root, that node must be duplicated by the rewrite rule ^(type ID)+.  
有时ANTLR必须复制节点避免循环和提供有用的语义，在下一个例子，规则type返回的树是类型V。在输入中只指明了一种类型(例如:"int a,b,c"),但多个标识符要创建多个以int为根的树，这样，重写规则必须复制节点。

grammar T;

options {output=AST;}

a : type ID (',' ID)\* ';' -> ^(type ID)+;

type : 'int'<V> ;

ID : 'a'..'z'+ ;

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

WS : (' '|'\\n') {$channel=HIDDEN;} ;

We want 3 trees, one for each identifier:  
我们需要三个树，每一个标识符一个。  
注:亲，我猜你一定已经注意到了 一个int 分裂成下面的三个int了...

[?](http://www.antlr.org/wiki/display/ANTLR3/ANTLR+v3+printable+documentation)

|  |
| --- |
| (int<V> a) (int<V> b) (int<V> c) |

We need to override dupNode() in our node class definition as well as two constructors:  
我们需要重写dupNode()还有两个构造函数在我们的节点类定义里。

[?](http://www.antlr.org/wiki/display/ANTLR3/ANTLR+v3+printable+documentation)

|  |
| --- |
| class V extends CommonTree {      public V(Token t) { token=t;}  // for 'int'<V>  // 用于'int'<V>的构造函数      public V(V node) { super(node); }  // for dupNode  //用于dupNode的构造函数      public Tree dupNode() { return new V(this); } // for dup'ing type  //用于复制类型      public String toString() { return token.getText()+"<V>";}  } |

Here is a test rig:  
这是一个测试的脚手架:

[?](http://www.antlr.org/wiki/display/ANTLR3/ANTLR+v3+printable+documentation)

|  |
| --- |
| public class Test {      public static void main(String[] args) throws Exception {          CharStream input = new ANTLRFileStream(args[0]);          TLexer lex = new TLexer(input);          TokenRewriteStream tokens = new TokenRewriteStream(lex);          TParser parser = new TParser(tokens);          TParser.a\_return r = parser.a();          if ( r.tree!=null ) {              System.out.println(((Tree)r.tree).toStringTree());              ((CommonTree)r.tree).sanityCheckParentAndChildIndexes();          }      }  } |

**Using custom AST node types  
使用自定义的抽象语法树节点类型**

[?](http://www.antlr.org/wiki/display/ANTLR3/ANTLR+v3+printable+documentation)

|  |
| --- |
| /\*\* An adaptor that tells ANTLR to build CymbalAST nodes \*/ /\*\* 一个告诉antlr来构建CymbalAST节点的适配器 \*/  public static TreeAdaptor cymbalAdaptor = new CommonTreeAdaptor() {      public Object create(Token token) {          return new CymbalAST(token);      }      public Object dupNode(Object t) {          if ( t==null ) {              return null;          }          return create(((CymbalAST)t).token);      }      public Object errorNode(TokenStream input, Token start, Token stop,                              RecognitionException e)      {          CymbalErrorNode t = new CymbalErrorNode(input, start, stop, e);          return t;      }  }; |

Here's a suitable error node:  
这是一个对应的错误节点：

[?](http://www.antlr.org/wiki/display/ANTLR3/ANTLR+v3+printable+documentation)

|  |
| --- |
| /\*\* A node representing erroneous token range in token stream \*/  /\*\* 在tooken流中表现错误的token范围的节点\*/  public class CymbolErrorNode extends CymbolAST {          org.antlr.runtime.tree.CommonErrorNode delegate;            public CymbolErrorNode(TokenStream input, Token start, Token stop,                                             RecognitionException e)          {                  delegate = new CommonErrorNode(input,start,stop,e);          }            public boolean isNil() { return delegate.isNil(); }            public int getType() { return delegate.getType(); }            public String getText() { return delegate.getText(); }          public String toString() { return delegate.toString(); }  } |

**Error Node Insertion Upon Syntax Error  
根据语法错误插入Error Node**

Prior to v3.1, ANTLR AST-building parsers did not alter the resulting AST upon syntax error. After v3.1 ANTLR adds an error node as created by TreeAdaptor.errorNode(...) to represent the missing nodes or confusing input sequences. The first token in the error sequence is the token at which the parser first detected an error. The last token in the sequence is the last token consumed during error recovery. ANTLR creates a CommonErrorNode by default, but you can obviously create your own tree adapter and override this.  
v3.1之前，ANTLR 构建抽象语法树的分析器在语法错误后不会修改结果的抽象语法树。  
v3.1之后，ANTLR增加一个错误节点，它由TreeAdaptor.errorNode(...)创建来展现缺失的节点或者令人迷惑的输入序列。语法解析器第一次检测到错误的token是错误序列中的第一个。在错误恢复的时候消耗的最后一个token就是错误的序列中的最后一个。antlr默认创建的是CommonErrorNode，但显然，你可以创建自己的树适配器并重写这个。

Let me demonstrate the new mechanism by example. Referring to the attached SimpleC.g from the v3 examples, here is some good input:  
让我用例子展示这个新机制。涉及V3例子附带的simpleC.g，下面是合法的输入。

[?](http://www.antlr.org/wiki/display/ANTLR3/ANTLR+v3+printable+documentation)

|  |
| --- |
| int foo() {    for (i=0; i<3; i=i+1) {      x=9;    }  } |

That input results in the following tree output:  
那个输入结果是下面的树输出:

tree=(FUNC\_DEF (FUNC\_HDR int foo) (BLOCK (for (= i 0) (< i 3) (= i (+ i 1)) (BLOCK (= x 9)))))

The grammar and tree construction for the FOR loop is as follows:  
用于for循环的语法和数构造如下：

forStat

: 'for' '(' start=assignStat ';' expr ';' next=assignStat ')' block

-> ^('for' $start expr $next block)

;

Now, remove the first '(' of the or loop:  
现在移除for循环的第一个'(':

[?](http://www.antlr.org/wiki/display/ANTLR3/ANTLR+v3+printable+documentation)

|  |
| --- |
| int foo() {    for i=0; i<3; i=i+1) {      x=9;    }  } |

You will see that ANTLR detects an error, but magically inserts the missing token. In this case, the parser was not asked to insert the '(' into the tree so there is no evidence of the error in the output tree:  
你将会看到antlr检测到一个错误，但是神奇地插入缺失的token。在这个例子，语法解析其没有要求插入'(' 到树，所以输出树没有错误的迹象。

line 2:6 missing '(' at 'i'

tree=(FUNC\_DEF (FUNC\_HDR int foo) (BLOCK (for (= i 0) (< i 3) (= i (+ i 1)) (BLOCK (= x 9)))))

What about when you have a random extra token such as "22" before the '(':  
那当你在'('之前插入一个随机额外的token例如"22",又会如何呢？

[?](http://www.antlr.org/wiki/display/ANTLR3/ANTLR+v3+printable+documentation)

|  |
| --- |
| int foo() {    for 22 (i=0; i<3; i=i+1) {      x=9;    }  } |

line 2:6 extraneous input '22' expecting '('

tree=(FUNC\_DEF (FUNC\_HDR int foo) (BLOCK (for (= i 0) (< i 3) (= i (+ i 1)) (BLOCK (= x 9)))))

Again, ANTLR detects the error and is able to ignore the extraneous token to yield a valid tree.  
再一次的，antrl检测到错误并能忽略多余的token来产生一个合法的树。

If you forget a token that must go into the output tree, however, you will see an error node. Given a missing identifier at the start of the FOR loop:  
如果你忘记一个必须放入输出树的token，不管怎样，你都会看到一个错误节点。在for循环的开始位置，会给出一个缺失标识符。

[?](http://www.antlr.org/wiki/display/ANTLR3/ANTLR+v3+printable+documentation)

|  |
| --- |
| int foo() {    for (=0; i<3; i=i+1) {      x=9;    }  } |

The parser emits:

line 2:7 missing ID at '='

tree=(FUNC\_DEF (FUNC\_HDR int foo) (BLOCK (for (= <missing ID> 0) (< i 3) (= i (+ i 1)) (BLOCK (= x 9)))))

The toString() method of the error node yields "<missing ID>".  
错误节点的toString方法产生"<missing ID>"。

When the parser gets really confused, such as when it gets a NoViableAltException, you will see that it consumes a whole bunch of input and adds it to the tree as an error node (it indicates what tokens it consumes during resynchronization). Input:  
当语法解析器变得真正迷惑的时候，象当它遇到一个NoViableAltException，你会看到它消耗整束输入，并把它当作一个错误节点加入到树中(它展示了在同步的时候消耗了什么token)。输入:

);

int foo() {

for (i=0; i<3; i=i+1) {

x=9;

}

}

yields:  
输出:

line 1:0 required (...)+ loop did not match anything at input ')'

tree=<error: );

int foo() {

for (i=0; i<3; i=i+1) {

x=9;

}

}>

**Making custom error nodes  
自定义错误节点**

Just override errorNode() in TreeAdaptor. The default handling is as follows:  
仅重写treeAdaptor中的erroNode。默认处理如下:

[?](http://www.antlr.org/wiki/display/ANTLR3/ANTLR+v3+printable+documentation)

|  |
| --- |
| public Object errorNode(TokenStream input, Token start, Token stop,              RecognitionException e)  {      CommonErrorNode t = new CommonErrorNode(input, start, stop, e);      return t;  } |

Make sure that your error node type is a subclass of your node type so that you do not get class cast exceptions.  
确定你的错误节点类型是你的节点类型的子类，这样才不会有类型转换异常。

See the next section for example of how to override.  
看下一节举例展示如何重写。

**Turning off error node construction  
关闭错误节点的构建。**

To turn this off, just override errorNode:  
想把这个关掉，只要重写errorNode

[?](http://www.antlr.org/wiki/display/ANTLR3/ANTLR+v3+printable+documentation)

|  |
| --- |
| class MyAdaptor extends CommonTreeAdaptor {      public Object errorNode(TokenStream input, Token start, Token stop,                              RecognitionException e)      {          return null;      }  } |

and then set  
然后设置:

[?](http://www.antlr.org/wiki/display/ANTLR3/ANTLR+v3+printable+documentation)

|  |
| --- |
| parser.setTreeAdaptor(new MyAdaptor()); |

**What makes a language problem hard?  
是什么让一个语言问题变得很难?**

Given a source to target mapping. How can you characterize the difficulty of the translation?   
给一个源语言与目标语言的映射.你如何能描述出这个翻译的难点？

* Is the set of all input fixed? If you have a fixed set of files to convert, your job is much easier because the set of language construct combinations is fixed. For example, building a general Pascal to Java translator is much harder than building a translator for a set of 50 existing Pascal files.
* 所有的输入集合是固定的么？如果你有一个固定的文件集合要转换，你的工作是比较简单的，因为因为语言构造组合的集合是固定的。例如，构建一个通用的pascal到java的翻译器比构建一个翻译50个存在pascal文件集合的的翻译器难的多。
* Forward or external references? I.e., multiple passes needed? Pascal has a "forward" reference to handle intra-file procedure references, but references to procedures in other files via the USES clauses etc... require special handling.
* 超前引用或者外部引用？也即，多种途径都需要？ pascal有一个超前引用来处理内部文件的过程引用，但过程引用在别的文件里通过USES从句等...需要特殊的处理。
* Is input order of sentences close to output order? Are there multiple files to generate from a single input file or vice versa?
* 句子输入的顺序是否和输出的顺序接近? 是一个输入文件产生多个输出文件还是反过来?
* Context sensitive lexer? You can't decide what vocabulay symbol to match unless you know what kind of sentence you are parsing.
* 上下文敏感的词法分析器？你不能决定那个词汇表的符号去匹配，除非你知道你在分析的是那种类型的句子。
* Are delimiters non-fixed for things like strings and comments? That makes it tough to build an efficient lexer.
* 分隔符是像字符串或者注释那样非固定的吗？那样会使构建一个高效的词法器比较艰难。
* Is language big; like lots of statements?
* 语言大吗? 比如有很多语句？
* Are the source statements really similar; declarations vs expressions in C++?
* 源文件语句是不是很相近; 如c++中的声明之于表达式？
* Column sensitive input? E.g., are newlines significant like lines in a log file and does the position of an item change its meaning?
* 输入中的列位置敏感吗？比如，新行是否像行在日志文件那样重要？一项的位置改变会改变它的含义吗？
* Case sensitivity problems like fortran?
* 像fortran的大小写敏感问题？
* Do you need good error recovery? Good reporting?
* 你需要一个好的错误恢复嘛？好的报告呢？
* Well defined language or no manual; hacked for ages like gnucc by non-language designers? Is your language VisualBasic-like?
* 定义良好的语言或者不用手册的语言？为非语言设计者给一些古老的东西如gnucc提供包装？你的语言是否像vb？
* How fast does your translator have to be? It is often the case that building lots of translator phases simplifies your problem, but it can slow down the translation.
* 你的翻译器必须要多快？是否经常处于这样的情况：要构建许多翻译语法来简化你的问题但这拖慢了翻译。
* Does your input have comments as you do in programming languages that can occur anywhere in the input and need to go into the output in a sane location?
* 你的输入有像你在编程语言中的可以随处放但是输出时必须要放到一个合理的位置的注释？
* How much semantic information do you need to do the translation? For example, do you need to simply know that something is a type name or do you need to know that it is, say, an array whose indices are a set like (day,week,month) and contains records? Sometimes syntax alone is enough to do translation.
* 有多少语义信息你需要翻译的？例如，你需要只需要简单地知道某些东西是类型名还是你需要知道比如说，一个用来展现集合(天,星期,月份)的数组，和是否包含记录？有时候，只需要句法就足够翻译了。
* Equivalent syntaxes?  In C there are many different ways to dereference pointers.  You can normalize the language to a standard representation, but you might loose the original representation. The choice usually hinges on whether the output will be human-edited or not.  Designing the right tree structure has to incorporate decisions like this.
* 等价语法？在c中，有很多放方法去解引用指针。你能规范化语言成标准的表示，但你可能对原来的表示失去控制。选择的关键点是这些输出是否人易编辑的。设计正确的树结构不得不像这样合并决策。
* Jurgen Pfundt points out: The considered language might be small, but mapping is targeted for the conversion of huge files and this is really a challenge. An input file with a size of several megabytes restricts the usage of tree parsers or any other kind of memory consuming features. The transformation should be done in one single pass due to performance requirements and an extremly good and comfortable error reporting and error recovery is a must.
* Jurgen Pfundt指出:一个深思熟虑的语言应该是小的，但是是针对巨大文件转换的映射，这实在是一个挑战。一个几m输入文件约束树解析器的使用或者任何其他的消耗内存的特征。因为性能要求和必须有极其良好的错误输出和错误恢复，转换应该在一遍扫描中完成。

问题集

错误集:

Now, remove the first '(' of the or loop:

or-> for