**—ANTLR 参考手册**

# 献给

项目领导和最高导师

Terence Parr

旧金山大学

支持站点

jGuru.com

Your View of the Java Universe

初期代码获益于

John Lilly, Empathy Software

C++ 代码生成器

Peter Wells 和 Ric Klaren

C# 代码生成

Micheal Jordan, Kunle Odutola 和 Anthony Oguntimehin。

Python's 方面的扩展来自于

Wolfgang Häfelinger and Marq Kole

基础软件支撑来自 Perforce:

世界上最好的源码控制系统之一

感谢以下朋友贡献了他们的聪明才智

Loring Craymer

Monty Zukowski

Jim Coker

Scott Stanchfield

John Mitchell

Chapman Flack (UNICODE, 流部分)

关于 Eclipse 和 NetBeans 方面的源码改进来自于

Marco van Meegen and Brian Smith

ANTLR 2.7.5 版

2004 年 12 月 22 日

# 目录

## 前言 ANTLR 是什么

ANTLR, 语言识别的另一个工具(ANother Tool for Language Recognition ),(前身是 **PCCTS**)是一种语言工具，它提供了一个框架，可以通过包含 Java,C++,或 C#动作（action）的语法描述来构造语言识别器，编译器和解析器。

计算机语言的解析已经变成了一种非常普遍的工作。传统的计算机语言的编译器和工具（如 C 或 Java）仍旧需要被构造，它们的数量与需要开发的那些成千上万的小语言的识别工具和解析工具相比是相形见拙。程序员为了解析数据格式，图形文件（如，PostScript,AutoCAD），文本文件（如，HTML,SGML 等）而需要构造解析器。ANTLR 被设计出来处理所有这些转换工作。

Terence Parr 从 1989 年就和他的同事开始了 ANTLR 方面的工作，在编译理论和语言工具构造方面做出了巨大的贡献，引发了基于 LL(k)文法识别工具的苏醒。

这儿有一个按年份排列的软件历史和 ANTLR/PCCTS。的贡献者的列表。

这儿是 ANTLR 的软件授权。

可以获得入门教程，从 ANTLR FAQ at jguru.com 可以找到你的一些问题的答案。可以参见 http://www.ANTLR.org 和术语表。

# 第 1 章 ANTLR 规范：元语言（Meta-Language）

ANTLR 接受 3 类语法规范——语法分析器（parsers），词法分析器（lexers），和树分析器（tree-parsers）(也叫树遍历器 tree-walkers)。 由于 ANTLR 使用 LL(k) 分析所有的 3 种语法变型，并且语法说明相似，因而产生的 lexers 和语法分析程序也很类似。另外产生的识别程序可读性很好，你可以查看输出的内容来明白很多关于 ANTLR 的机理。

## 1.1 元语言词汇表（Meta-Language Vocabulary）

### 空格(Whitespace)

空格、tab 符号和换行符是分隔符，在 ANTLR 中可以分隔诸如标识符这样的词汇符号，但除此之外，它们会被忽略。例如，“FirstName LastName”对 ANTLR 来说是两个记号引用（token reference）序列而不是一个记号（token），空格，然后再接着一个记号（token）。

## 注释（Comments）

ANTLR 接受 C 语言风格的块注释和 C++风格的行注释。在语法类和规则中，Java 风格的文档注释也是可以接受的，在需要的时候，这些注释可以被传递给生成的输出文件。例如：

```
/**此文法识别简单的表达式
 * @作者 Terence Parr
 */
class ExprParser;

/**匹配因子 */
factor : ... ;
```

## 字符集（Characters）

字符常量像 Java 中那样被确定。它们包含八进制转义字符集(e.g.,'\377')、Unicode 字符集(e.g.,'\uFF00')和能被 Java 识别的常用的字符转义('\b','\r','\t','\n','\f','\'', '\\')。在词法分析器规则中，单引号代表一个可以在输入字符流中能得到匹配的字符。单引号的字符在语法分析器中是不被支持的。

## 文件结束标志（EOF）

EOF 记号（Token）可以用如下语法分析器规则自动生成：

```
rule : (statement)+ EOF;
```

你可以在词法分析器规则的动作（action）中检测 EOF_CHAR：

```
// make sure nothing but newline or
// EOF is past the #endif
ENDIF
{
  boolean eol=false;
}
```

```
    :   "#endif"
        ( ('\n' | '\r') {eol=true; } )?
        {
          if (!eol) {
            if (LA(1)==EOF_CHAR) {error("EOF"); }
            else {error("Invalid chars"); }
          }
        }
    ;
```

同时你可以把文件结束符当作一个字符来检测，但它实际上并不是一个字符，而是一种条件。你应该在你的词法分析器语法中重载 CharScanner。uponEOF()函数：

```
/** 此方法由 YourLexer.nextToken()当文法分析器
 *  遇到 EOF 条件时调用。
 *  EOF 并不是字符。
 *  当在处理语法谓词或一般的词法规则时到达 EOF，并不会调用此方法，
 *  因为可能会抛出 IOException。这是通常 EOF 条件的陷阱。
 *  在全部对先前所有的记号求值后，并且当分析程序请求在 EOF 后的
 *  非 EOF 记号时，uponEOF()方法会被调用。
 *  你也许希望抛出记号或字符流异常，可能因为这是一个过早的 EOF，
 *  即事实上并未到达文件结尾，或者到达文件结尾后，想回到文件开始
 *  重新引用文件。
 */
public void uponEOF()
  throws TokenStreamException, CharStreamException
{
}
```

文件结束的情形有点让人困惑（从 2.7.1 版本开始），因为 Terence 将-1 当作一个字符而不是一个整数(-1 是 '\uFFFF'...)。

## 字符串（Strings）

字符串常量是一个由双引号括起来的字符序列。字符串中的字符可以是字符集中合法的转义字符(八进制，Unicode 等)。目前 ANTLR 并不允许 Unicode 出现在字符串常量中（你不得不用转义符)，这是因为在 anglr.g 文件中设定 charVocabulary 选项为 ascii。

在词法分析规则中，字符串被理解为在输入流中将要进行匹配的字符序列 (例如：“for” 等效于 ‘f’ ‘o’ ‘r’)。

在语法分析规则中，字符串代表记号（tokens）,并且每个唯一的字符串被分派给一个记号类型。然而，ANTLR 并不创建词法分析规则来匹配字符串。相反，ANTLR 将这些字符串输入到一张与词法分析器相关联的字符表中。在将记号传送给语法分析器前，ANTLR 将产生检测代码来检测字符表中的每个记号的内容，每遇到一个匹配都会修改记号的类型。你也可以执行手动检测－－自动代码的生成由词法分析器选项控制。

你也许想在你的动作（action）中使用某个字符的记号类型值，例如在错误处理的同步部分。对于那些只由字母字符组成的字符串来说，这些字符串的值将是一个形如 LITERAL_xxx 的常量值，这里 xxx 是这个记号的名字。例如，字符串“return”将有一个 LITERAL_return 值与之关联。你也可以为记号节（tokens section）中使用的字符分派一个特定的标号。

## 记号引用（Token references）

以大写字符开头的标识符称为记号引用。接下来的字符可以是任意字符、数字或下划线。在语法分析规则中一个记号引用将引起匹配特定的记号。在词法分析规则中的一个记号引用将引起一个匹配记号的字符的词法规则的调用。换句话说，词法分析器中的记号引用被看作是一个规则引用。

## 记号定义（Token definitions）

在词法分析器中的记号定义与语法规则的语法定义是相同的,但是指向记号而不是语法规则。例如：

```
class MyParser extends Parser;
idList : ( ID )+;    // 解析规则定义


class MyLexer extends Lexer;
```

```
ID : ( 'a'..'z' )+ ;    // 记号定义
```

## 规则引用（Rule references）

以小写字母开头的标识符是对 ANTLR 的语法规则的引用。接下来的字符可以是任意字母，数字或下划线。词法规则不能引用语法规则。

## 动作（action）（Actions）

在花括号中的字符序列（可能是嵌套的）是语义动作（action）。在字符串和字符中的花括号并不是动作（action）分隔符。

## 动作（action）参数（Arguments Actions）

在方括号中的字符序列（可能是嵌套的）是动作（action）参数。在字符串和字符中的方括号不是动作（action）分隔符。在[]中的参数是用生成语言的语法定义的，并且用逗号分开。

```
codeBlock
[int scope, String name] // 输入参数
returns [int x]          // 返回参数
:... ;

// pass 2 args, get return
testcblock
{int y; }
      :       y=cblock[1,"John"]
      ;
```

许多人倾向于我们使用普通的括号来表示参数，但是括号在 EBNF 中已经被很好的用来定义语法组符号(grammatical grouping symbols)。

## 符号（Symbols）

下面的表统计了在 ANTLR 中使用的标点符号和关键字。

| 符号 | **描述** |
| --- | --- |

| | |
|---|---|
| (...) | 子规则 |
| (...)* | 闭包子规则（零和多个） |
| (...)+ | 正闭包子规则（一个和多个） |
| (...)? | 可选（零个和一个） |
| {...} | 语义动作（action） |
| [...] | 规则参数 |
| {...}? | 语义谓词 |
| (...)=> | 语法谓词 |
| \| | 可选符 |
| .. | 范围符 |
| ~ | 非 |
| . | 通配符 |
| = | 赋值 |
| : | 标号符，规则开始 |
| ; | 规则结束 |
| <...> | 元素选项 |
| class | 语法类 |
| extends | 指定语法基类 |
| returns | 指定规则返回类型 |
| options | options 段 |

| | |
|---|---|
| **tokens** | tokens 段 |
| **header** | header 段 |
| **tokens** | token 定义段 |

## 1.2 Header 段（Header Section）

一个 header 段包含了任何由 ANTLR 生成的代码在被输出到语法分析器前需要被替换的源码（译者注：形为类似 include、import）。这个主要用在 C++的输出中，因为 C++需要一些元素在引用之前必须被声明。在 Java 中，这可以用来为最后的语法分析指定一些包文件。一个 header 段看起来像下面这样：

```
header {
  source code in the language generated by ANTLR;
}
```

header 段是语法文件的第一节。根据选择的目标语言的不同，会有不同类型 header 段。请参考相应的附录。

## 1.3 语法分析类的定义（Paser　Class Definitions）

所有的语法规则必须与一个语法分析类关联。一个语法文件（.g）只包含一个语法分析类的定义（以及词法分析程序和树分析程序），一个语法分析类的定义先于其选项（options）和规则定义。语法文件中的语法分析类的定义通常如下所示：

```
{ optional class code preamble }
class YourParserClass extends Parser;
options
tokens
{ optional action for instance vars/methods }
parser rules...
```

当在面向对象语言中生成代码时，语法分析类将在输出中生成一个类，规则都会变成这个类的成员函数。在 C 中，类将生成一个结构，一些名字分配（name-mangling）的算法将使生成的规则函数是全局唯一的。

前面的可选类可以是包含在{}中的任意文本。前面的可选类，如果存在的话，将被直接输出到生成类文件中，并且在类定义之前。

封闭的花括号不能用来分隔类，因为它很难将一个文件底部的左花括号与这个文件顶部的花括号联系起来。然而，一个语法分析类被认为是连续的，直到遇到下一个类的语句。

你可以指定语法分析器的基类，它将作为语法分析器中生成代码所需的基类。这个基类必须完全可信并在双引号中，它自己必须是 ANTLR.LlkParser 的子类。例如：

```
class TinyCParser extends Parser("ANTLR.debug.ParseTreeDebugParser");
```

## 1.4 词法分析类定义（Lexcal Analyzer Class Definitions）

一个语法分析器类将产生一个将相关语法结构应用于输入流中的记号集的语法分析对象。为了执行词法分析，你需要指定一个词法分析类，它描述了如何将字符输入流分解成记号流。它的语法类似于语法分析类：

```
{ optional class code preamble }
class YourLexerClass extends Lexer;
options
tokens
{ optional action for instance vars/methods }
lexer rules...
```

词法分析类中的词法规则成为生成类中的成员方法。每个语法文件（.g）只包含一个词法分析类。语法分析类和词法分析类可以以任意顺序出现在语法文件中。

前面的可选类（optional class code preamble）是在{}中的任意文本。前面部分的可选类，如果存在，将输出到生成类的文件中，在类定义的之前。

你可以定义一个词法分析类的超类，它可以被用来作为生成词法分析类的超类。这个超类必须是完全可信的(fully-qualified)，并且在双引号中，而它本身是 ANTLR.CharScanner 子类。

## 1.5 树分析类定义（Tree-parser Class Definitions）

一个树分析器就像一个语法分析器，不同的是树分析器处理的是二维的由结点组成的抽象语法树（Abstract Syntax Tree），而不是处理由记号组成的记号流。树分析器定义类似于语法分析类，不同的是规则定义中可能包含特殊形式来指示其递归下降树。同样，一个特定的语法文件（.g）中只能包含一个树分析器。

```
{ optional class code preamble }
class YourTreeParserClass extends TreeParser;
options
tokens
{ optional action for instance vars/methods }
tree parser rules...
```

你可以定义一个树分析器的超类，它可以被用来作为生成树解析器的超类。这个超类必须是完全可信的（fully-qualified），并且在双引号中，它本身是 ANTLR.TreeParser 子类。

## 1.6 选项段（Option Section）

并不是让程序员给分析程序生成器指定多个的命令行参数，文法中的选项段本身就可以达到此目的。这种方法更受欢迎，因为它将需要的选项关联到文法而不 ANTLR 的调用。这部分以 options 关键字开关，包含多个的选项/值赋值语句。可以为每个文件、每个文法、每个规则和每个子规则指定一个选项段。

同时你也可以为一个元素指定一个选项段，例如记号引用。

## 1.7 记号段（Tokens Section）

如果你需要定义一个“虚拟”的记号，也即没有对应实际输入的符号与其关联，可以使用记号段来定义它们。虚拟记号通常用于标识树结点，该类树节点用于标记或组织根据实际

输入生成的子树。例如，你可能希望让 EXPR 结点成为每一个子树表达式的根结点，DECL 表示子树的声明，这样在树的遍历时更容易引用它们。因为 EXPR 没有对应的输入符号，你就不能在文法中通过引用来隐含地定义它。使用如下方法来定义那些虚拟的记号。

```
tokens {
    EXPR;
    DECL;
}
```

通常的语法是：

```
tokenSpec : "tokens" LCURLY
            (tokenItem SEMI)+
            RCURLY
          ;

tokenItem : TOKEN ASSIGN STRING (tokensSpecOptions)?
          | TOKEN  (tokensSpecOptions)?
          | STRING (tokensSpecOptions)?
          ;
tokensSpecOptions
          : "<"
              id ASSIGN optionValue
              ( SEMI id ASSIGN optionValue )*
            ">"
          ;
```

在 token 段中你还可以定义字面值，更重要的是，给它们赋与一个有效的标签，如下例所示。

```
tokens {
    KEYWORD_VOID="void";
    EXPR;
    DECL;
```

```
    INT="int";
}
```

以这种方式定义的字符串会被认为你已经在分析程序中对它们进行了引用。

如果文法导入了包含一个记号的词汇表，比如记号 T，然后你可以简单地通过在该文法的记号段中添加表达式 T = “字符串常量”来将一个字符串常量关联到该记号类型(也即 T)。类似地，如果导入的词汇表中定义了一个字面值，比如"_int32"，但没有相关联的标签，你可以在记号段中关联一个标签，例如 INT32="_int32"。你可以为在记号段中定义的记号定义选项。目前可用的选项仅有 AST=*class-type-to-instantiate*。

```
// 定义需要创建的多个 AST 结点
// 可以在文法中实际引用时重载
tokens {
    PLUS<AST=PLUSNode>;
    STAR<AST=MULTNode>;
}
```

## 1.8 语法继承（Grammar Inheritance）

面向对象编程语言，例如 C++和 Java，允许你定义一个新的对象，当它与已经存在的对象有区别时，这种方法提供了很多好处。“根据差异编程”节省了开发/测试的时间，并且将来对基类的修改也会自动传递给子类。ANTLR 支持语法继承，也就是基于一个基类来创建新的文法类的机制。文法相关的语法结构和动作（action）均可以单独被修改。

## 1.9 规则定义（Rule Definitions ）

因为 ANTLR 把词法分析看作是对字符流的分析，所以词法分析规则的语法规则可以同时讨论。一般讨论规则时，我们使用术语 atom 代表输入流中的一个元素(可能是字符或记号)。输入流中 atoms 的结构通过多个互相引用的规则来指定。每一个规则有一个名字，一些可选的参数，一个可选"throws"子句，一个可选的初始化动作（action）（init-action），一个可选的返回值，和一个或多个可选项。

ANTLR 规则的基本形式为：

```
rulename
    :   alternative_1
    |   alternative_2
   ...
    |   alternative_n
    ;
```

如果规则需要参数，使用如下形式：

```
rulename[formal parameters] : ... ;
```

如果你希望从规则返回一个值，使用 returns 关键字：

```
rulename return [type id] : ... ;
```

这里 type 是一个生成语言的类型指定符，id 是生成语言的一个有效标识符。Java 中一个单一的类型指定符能够满足大部分的应用,但是例如返回一个字符串的数组将需要一对方括号：

```
id return [String[] s]: ( ID {...} )* ;
```

同样，如果生成 C++，返回类型可能会很复杂，例如：

```
id return [char *[] s]: ... ;
```

return 语句的 id 会传递给输出代码。动作（action）可能直接对此 id 赋值来设置返回值。不要在动作（action）中使用 return 指令。

为了指明你的分析器（或树分析规则）可以抛出非 ANTLR 指定的异常，使用异常子句。例如：下面是一个简单的通过规则指定的分析程序，该分析程序抛出 MyException：

```
class P extends Parser;

a throws MyException
  : A
  ;
```

ANTLR 为规则 a 生成如下代码：

```
    public final void a()
        throws RecognitionException,
               TokenStreamException,
               MyException
    {
        try {
            match(A);
        }
        catch (RecognitionException ex) {
            reportError(ex);
            consume();
            consumeUntil(_tokenSet_0);
        }
    }
```

词法规则可能并不指定异常。

初始化动作（action）（Init-actions）在冒号前指定。初始化动作（action）（Init-actions)与一般的动作(action)不同,因为它们总会执行,并推测模式(guess mode)无关。另外，它们适合于局部变量的定义。

```
rule
{
    init-action
}
    :   ...
    ;
```

**词法分析规则（Lexer rules）。**词法分析文法中定义的规则必须有一个以大写字母开头的名字。这些规则隐含地匹配输入流的字符，而不是记号流中的记号。被引用的文法元素包括记号引用（token references）（隐含的词法分析规则引用），字符和字符串。词法分析规则按照与语法分析规则完全相同的方式被处理，可能会指定参数和返回值，未来，词法分析

规则同样可以使用局部变量和递归使用。更多关于词法规则请参考第 2 章 ANTLR 的词法分析。

**语法分析规则（Parser rules）。**语法规则将结构应用于记号流，而词法规则将结构应用于字符流。语法分析规则不能引用字符的字面值。语法分析规则中双引号括起的字符会被认为是记号引用（token references）和迫使 ANTLR 将字符串常量存储在表中，该表可以由相关词法分析程序中的动作（action）来检查。

所有的语法分析规则必须以小写字母开头。

**树分析规则（Tree-parser rules）。**树分析规则中，一个额外的特殊的语法允许被用来指定二维结构的匹配。一个语法分析规则类似：

```
rule : A B C;
```

意思是“依次匹配 A B C”。一个树分析规则可能会使用如下语法：

```
rule : #(A B C);
```

意思是“匹配一个类型 A 的结点，然后下降它的子结点列表，匹配 B 和 C”。这个符号可以任意嵌套，可以在 EBNF 结构能够使用的地方使用#（…），例如：

```
rule : #(A B #(C D (E)*) );
```

## 1.10 原子的产生式元素（Atomic Production Elements）

**字数常量（Character literal）。**字数常量仅仅可以在词法分析规则中被引用。单个的字符会在字符输入流被匹配。不需要转义正则表达式中的元符号，因为正则表达式并不是用来匹配词法原子符号的。例如，当你指定字面字符来匹配时，‘{’并不需要转义符。字数常量和字符串常量外的元符号被用来指定词法结构。

你所引用的所有字符会隐含地添加到全局字符词汇表中（具体请参考 charVocabulary 节）。当你引用通配符时，如‘.’或‘~c’（除 c 外的任意字符），词汇表此时就会起作用。

你不需要特别地处理 Unicode 字符。例如，下面是一个名为 LETTER 的规则，此规则匹配被认为是 Unicode 字母的字符：

```
protected
LETTER
    :   '\u0024' |
```

```
        '\u0041'..'\u005a' |
        '\u005f' |
        '\u0061'..'\u007a' |
        '\u00c0'..'\u00d6' |
        '\u00d8'..'\u00f6' |
        '\u00f8'..'\u00ff' |
        '\u0100'..'\u1fff' |
        '\u3040'..'\u318f' |
        '\u3300'..'\u337f' |
        '\u3400'..'\u3d2d' |
        '\u4e00'..'\u9fff' |
        '\uf900'..'\ufaff'
    ;
```

你可以在其它规则中引用上述规则：

```
ID  :   (LETTER)+
    ;
```

ANTLR 将生成代码来检查输入字符而不是 lexer 对象生成的字符集。

**字符串常量（String literal）**。语法分析规则中对字符串常量的引用会为此字符串常量定义一个记号类型，并且导致字符串常量在相关 lexer 的哈希表中被替换。相关的 lexer 将会自动检查每一个被匹配的记号，以查看该记号是否匹配一个字面值。如果匹配，此记号的记号类型会被设为从语法分析程序（parser）导入的为该字面值定义的记号类型。你可以关掉自动检查，然后在一个类似 ID 的简单规则手动检查。在语法分析程序中对字符串常量的引用会被添加一个元素类型的后缀，具体参考下面的记号引用章节。

词法规则中字符串的引用会特定的字符序列，是一种简写方式。例如，考虑下面的词法规则定义：

```
BEGIN : "begin" ;
```

这个规则可以以另外一种功能相同的方式重写：

```
BEGIN : 'b' 'e' 'g' 'i' 'n' ;
```

没有必要转义正则表达式中的符号，因为正则表达式并不是用来匹配词法分析程序（lexer）中字符。

**记号引用（Token Reference）**。语法分析规则中的记号引用意味着你希望使用特定的记号类型来识别一个记号。实际上这并不会调用相关的词法规则——词法分析阶段将记号流传递给语法分析程序（parser）。

词法分析规则中的记号引用意味着对该规则的一个调用方法，执行与语法分析程序中的规则引用相同的语义分析。这样的话，你可以指定规则参数和返回值。详情请参考下一规则引用章节。你同样可以指定记号引用上的选项。例如，下面的规则指引 ANTLR 从 INT 的引用创建 INTNode 对象：

```
i : INT<AST=INTNode> ;
```

该语法的选项为：

<*option*=*value*; *option*=*value*; ...>

**通配符（Wildcard）**。语法分析规则（parser rule）中的‘.’通配符代表任意一个记号；在词法分析规则（lexer rule）中，它代表任意一个字符。例如，‘.’代表任意一个在 B 和 C 之间的记号：

```
r : A B . C;
```

## 1.11 简单的产生式元素（Simple Production Elements）

**规则引用（Rule reference）**。对规则的引用意味着在语法分析程序中该位置处对该规则的一个方法调用。你可以传递参数和获取返回值。例如，形参和实参在方括号中被指定：

```
funcdef
    :   type ID "(" args ")" block[1]
    ;
block[int scope]
    :   "begin" ... {/*use arg scope/*} "end"
    ;
```

存储在变量中的返回值使用简单的赋值符返回：

```
set
{ Vector ids=null; }  // init-action
    :  "(" ids=idList ")"
    ;
idList returns [Vector strs]
{ strs = new Vector(); }   // init-action
```

```
:  id:ID
   { strs.appendElement(id.getText()); }
   (
      ",” id2:ID
      { strs.appendElement(id2.getText()); }
   )*
;
```

**语义动作（Symatic action）**。动作（action）是括在花括号（curly braces）中的源代码块（以目标语言来表示）。这段代码会在前面的产生元素已经识别之后，后续元素识别之前执行。动作（action）通常被用来产生输出，构造树或者修改符号表。动作（action）的位置决定了它什么时候被识别，相对于周围的文法元素。

如果动作（action）是产生式的第一个元素，它将在此产生式中任何其它元素之前被执行，除非此产生式由超前查看（lookahead）预测。

EBNF 子规则的第一个动作（action）后面可能紧跟着‘:’。这样做是为了指定此动作（action）是一个初始化动作（*init-action*），把它与子规则关联成为一个整体，而不是任意的产生式。一旦进入子规则，它就会被执行——在超前查看（lookahead）为子规则替换而进行预测之前——并且即使中预测过程中（检查语谓词）也会执行。例如：

```
(   {init-action}:
    {action of 1st production} production_1
|   {action of 2nd production} production_2
)?
```

不管可选的子规则中将匹配什么，初始化动作（init-action）都会执行。

初始化动作放置中在为子规则(...)+和(...)*生成的循环中。

## 1.12 产生式元素操作符（Production Element Operators）

**元素求反（Element complement）**。取反一元操作符’~’只能用于原子元素，比如记号标识符。对一些原子的记号（token）T，~T 将匹配除文件结束符（end-of-file）和 T 以外的任何记号。有词法分析规则（lexer rules）中，~’a’将匹配任何非’a’字符。”~.”（不是任何东西）毫无意义，同时也是不允许的。

词汇表中的空格对取反操作符来说很重要。在语法分析程序（parser）中，完整的记号类型列表对 ANTLR 来说是已知的，于是，ANTLR 简单地设置和清除标记的元素。对字符来说，如果你想使用取反操作符，你必须指定字符的词汇表。注意对类似 Unicode 字符块的庞大的词汇表来说，最坏情况下，对一个字符的取反意味着创建 2 ^16（2 的 16 次方）个元素集（大

约 8k）。字符的词汇表是 charVocabulary 选项指定的词汇表与所有在词法分析规则（lexer rules）中引用的字符的并集。下面是一个字符词汇表选项的简单使用例子：

```
class L extends Lexer;
options { charVocabulary = '\3'..'\377'; } // LATIN

DIGIT : '0'..'9';
SL_COMMENT : "//" (~'\n')* '\n'; （译者注：单行注释的文法）
```

**集合取反（Set complement）**。通过对其它集合取反，非操作符（not operator）同样可以用来构造一个记号集或字符集。最大的用处的就是当你希望匹配多个记号或多个字符，直到遇到特定的分隔符。并不是为这类集合引入特殊的语法，ANTLR 允许将~放在仅由简单元素且没有动作构成的子规则前，以此来生成这类集合。在这类特定的情况下，ANTLR 并不会生成子规则，而是创建一个集合匹配。简单的元素可以是记号引用，记号范围，字数常量，或者字符范围。例如：

```
class P extends Parser;
r : T1 (~(T1|T2|T3))* (T1|T2|T3);

class L extends Lexer;
SL_COMMENT : "//" (~('\n'|'\r'))* ('\n'|'\r);

STRING : '"' (ESC | ~('\\'|'"'))* '"';
protected ESC : '\\' ('n' | 'r');
```

**范围操作符（Rang operator）**。范围二元操作符意味着一定范围内的原子元素可能被匹配。词法分析程序中的表达式'c1'..'c2'匹配包含在此范围内（包括'c1'和'c2'）的所有字符。语法分析程序中的表达式 *T..U* 匹配任何记号类型包含在此范围内（包含 T 和 U）的记号，该范围是不确定的值，除非记号类型是在外部生成的。

**抽象语法树根结点操作符（AST root operator）**。当生成抽象语法树（ASTs）时，以根结点操作符"^"为后缀的记号引用将此结点强制生成并添加为当前树的根结点。这个符号仅仅当 buildAST 选项设置时有效。更多关于 ASTs 的信息是可以得到的，请参考后面相关的章节。

**AST 排除操作符（AST exclude operator）**。当生成抽象语法树（ASTs）时，以排除操作符"!"为后缀的记号引用并不会包含在为相应规则而构造的抽象语法树（AST）中。规则引用同样也可以以排除操作符为后缀，这意味着当为引用的规则构造树时，它并不会链接到为引用的

规则构造的树。同样，这个符号仅仅当 buildAST 选项设置时有效。更多关于 ASTs 的信息是可以得到的，请参考后面相关的章节。

## 1.13 记号类

通过使用范围操作符、非操作符、或者仅仅由原子的元素构成的子规则，你可以隐含地定义匿名的记号或字符类——具有很好时间和空间效率的集合。例如，你可以如下地定义一个词法分析规则：

```
OPS : (PLUS | MINUS | MULT | DIV) ;
```

或

```
WS  : (' '|'\n'|'\t') ;
```

这些单独地描述了记号和字符集合，这种集合很容易被优化为简单、单一的位的集合，而不是一系列的记号和字符的比较。

## 1.14 谓词

**语义谓词（Semantic predicate）**。语义谓词是在分析能够继续传递它们之前必须满足的条件。语义谓词的功能会在接下来的章节中详细地说明。语义谓词的语法就是以问号符(?)为后缀的语义动作：

```
{ 表达式 }?
```

其中的表达式不能有副作用，求值必须能够得到 true 或者 false（Java 中的 boolean 值或者 C++中的 bool 值）。既然语义谓词能够在预测时执行，它们不应该依赖动作的返回值或规则的参数。

**语法谓词（Syntactic predicate）**。语法谓词指定了被用来预测可替代项的超前预测分析语言（lookahead language）。语法谓词的功能会在接下来的章节中详细地说明。语法谓词的语法形式为以=>操作符为后缀的子规则：

```
( lookahead-language ) => production
```

这里的超前预测分析语言（lookahead language）可以是任何有效的 ANTLR 结构，包括对其它规则的引用。尽管如此，在语法谓词求值过程中，动作并不会被执行。

## 1.15 元素标签

任何原子的或规则引用的产生式元素可以用标识符进行标识（大小写并有重要）。在原子的元素带标签的情况，标识符在语义动作中被使用，以此来访问相关的 Token 对象或者字符。例如：

```
assign
    :   v:ID "=" expr ";"
        { System.out.println(
            "assign to "+v.getText()); }
    ;
```

在动作中对标签的引用并不需要"$"操作符，与 PCCTS 1.xx 版本中一样。

在动作中，一个记号引用可以这样被访问，就像通过*标签*访问 Token 对象，或通过*#标签*访问为该记号生成的 AST。为一个规则引用生成的 AST 结点在动作中可以以*#标签*来访问。

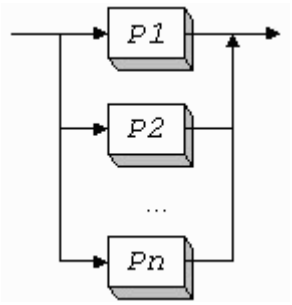
记号引用的标签同样可以在关联的语法分析异常处理中使用，来指定当记号不能被匹配时做什么。

规则引用的标签同样也可以在关联的语法分析异常中使用，因此任何在执行标识的规则时产生的异常能够被捕获到。

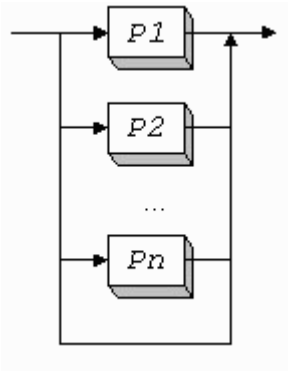
## 1.16 扩展的 BNF 规则元素（EBNF Rule Elements）

ANTLR 支持与下面四个子规则语法或语法图相应的扩展的 BNF 符号：

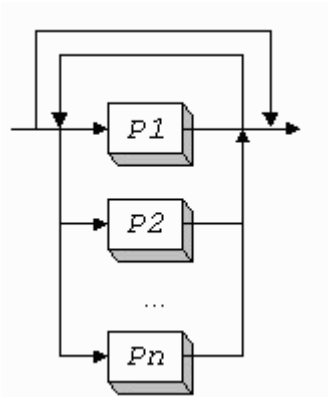
( *P1* | *P2* | ... | *Pn* )



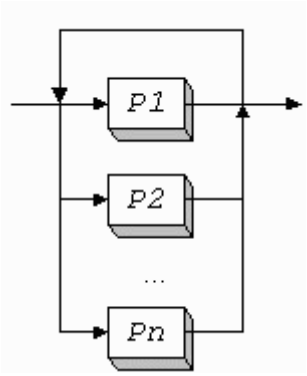

( *P1* | *P2* | ... | *Pn* )?




( *P1* | *P2* | ... | *Pn* )*




( *P1* | *P2* | ... | *Pn* )+




## 1.17 语义动作的解释（Interpretation Of Semantic Actions）

语义动作被逐字的复制到输出的语法分析程序中适当的位置，并且可能会抛出 AST action translation 异常。没有从 PCCTS 1.xx 开始的$-变量符号（$-variable notation）引入到 ANTLR 中。

## 1.18 语义谓词（Semantic Predicates）

语义谓词指定了在分析能够继续处理之前必须满足的条件（运行时）。我们需要区别两种类型的语义谓词：(i)确认（*validating*）谓词，如果在分析产生式时条件没有得到满足，就

抛出异常的谓词（类似断言 assert）；(ii)消除歧义（*disambiguating*）的谓词，提升到相关产生式的谓词汇表达式中的谓词。从语法上来说，语义谓词就是带有问号标记符为后缀的语义动作：

{语义谓词汇表达式}？ （{ semantic-predicate-expression }?）

此处的表达式可以使用任何程序员提供的或者 ANTLR 生成的符号，表达式在输出中出现的地方可用的符号。

谓词在产生式中的位置决定了它是哪种类型。例如，考虑下面的确认谓词（出现在任何非左边的位置），该谓词确保一个标识符号语法上是一种类型名：

```
decl: "var" ID ":" t:ID
      { isTypeName(t.getText()) }?
    ;
```

当确认谓词失败时，会产生语法分析异常。抛出的异常是 SemanticException。你可以在异常处理者（exception handler）中捕获此异常和其它的异常。

消除歧义的谓词在一个产生式中总是第一个元素，因为它们不能提升到动作、记号、规则引用之上。例如下述规则的第一个产生式有一个消除歧义的谓词，可以提升到谓词汇表达式中，作为第一个可供选择的：

```
stat:   // declaration "type varName;"
        {isTypeName(LT(1))}? ID ID ";"
    |   ID "=" expr ";"            // assignment
    ;
```

如果我们将此文法限制为 LL(1)，从语法上来说，它是不确定的，因为常见的左前缀：ID。尽管如此，语义谓词正确地提供附加的信息来消除分析决策时的歧义。分析逻辑将是：

```
if ( LA(1)==ID && isTypeName(LT(1)) ) {
    match production one
}
else if ( LA(1)==ID ) {
    match production one
}
else error
```

通常，在 PCCTS 1.xx 中，语义谓词代表了一个产生式的语义上下文。如此，语义和语法上下文（超前预测分析）能够被提升到其它规则中。在 ANTLR 中，谓词并不会被提升到包含它们的规则之外。因此，类似下面的规则：

```
type : {isType(t)}? ID ;
```

毫无意义。换句话说，这种语义上下文的特点给许多 PCCTS 1.xx 的版本产生了不可忽视的歧义。

## 1.19 语法谓词（Syntactic Predicates）

偶尔会有通过有限的预测不能呈现为确定的语法分析决策。例如：

```
a   :   ( A )+ B
    |   ( A )+ C
    ;
```

在 k 为任何值的 LL(k)情况下，通常的左前缀会造成两个产生式不确定。明显的是，这两个产生式可以从左因式分解为（*left-factored*）：

```
a   :   ( A )+ (B|C)
    ;
```

而不改变已经识别的语言。尽管如此，当动作嵌入在文法中时，从左因式分解（left-factoring）并不总是可能的。进一步来说，从左因式分解和其它文法上的处理不会产生自然（可读的）文法。

解决方法是在少数有限的 LL(k)（k>1）不足够的情况下，简单地使用任意的超前预测分析。ANTLR 允许你通过可能的无限字符串来以下述语法来指定超前预测分析语言：

```
( prediction block ) => production
```

例如，考虑下面的规则，该规则区分集合（逗号分隔的单词列表）和并列赋值（一个列表赋值给另外一个）：

```
stat:   ( list "=" )=> list "=" list
    |   list
    ;
```

如果一个紧跟着一个赋值符的列表在输入流在被发现，第一个产生式被预测。如果不是，会尝试第二个可供选择的产生式。

语法谓词是一种选择性的可返回（selective backtracking）的形式，因此，当对一个语法谓词求值时，动作会被关掉，所以动作没必要是未完成的。

语法谓词是使用目标语言中的异常来实现的，如果存在异常的话。当生成 C 代码（C 中没有异常）时，会使用 longjmp 来实现。

对任何在文法中发现的非 LL(k)决策，我们本可以选择简单地使用任意的超前预测分析。尽管如此，在文法中显示地使用任意超前预测分析很有用，因为你不必去猜测语法分析程序在做什么。更重要的是，存在模棱两可的语言结构，因为存在非确定的文法！例如，声名狼藉的 if-then-else 结构对任何 k 都没有 LL(k)文法。现在的文法是模棱两可的，不确定的：

```
stat:   "if" expr "then" stat ( "else" stat )?
    |   ...
```

;

在一个非确定的决策中，给定在两个产生式中的一个选择，我们简单地选择第一个。在大部分情况下，这样工作得很好。强制这个决策使用任意的超前预测分析会降低分析的效率。

### 1.19.1 固定深度的超前预测分析和语法谓词（Fixed depth lookahead and syntactic predicates）

ANTLR 并不能确保哪种超前预测分析可以跟在语法谓词后面(唯一的逻辑可能性是不管什么都可以跟在谓词预测的可选择项后，但是错误的输入等使之更复杂)，ANTLR 假设什么都可以跟在语法土谓词后。这种情形类似于当遇到记号规则定义结束时的词法超前预测分析的计算。

考虑带(...)*的谓词，其隐含的退出分支强行计算什么跟在循环的后面，这种情况下是语法谓词的末尾。

```
class parse extends Parser;
a       :       (A (P)*) => A (P)*
        |       A
        ;
```

超前预测分析在退出分支时人为地设为“任意的记号”。通常 P 与这“任意的记号”会产生冲突，但是 ANTLR 知道你的意思是匹配一系列的 P 记号，如果它们同时出现，并不产生警告。

在任何一个决策中如果不止一条路径能够通向谓词的结尾，ANTLR 会产生一个警告。下面的规则会产生两个警告。

```
class parse extends Parser;
a       :       (A (P|)*) => A (P)*
        |       A
        ;
```

空的可选项可以间接地成为这个循环的开始，与 P 相冲突。进一步来说，ANTLR 检测到了这个问题，就是有两路径可以到达谓词的结尾。生成的语法分析程序会发出警告但从不会终止（P|*）循环。

k>1 的超前预测分析中，情况会更复杂。当第 n 个超前预测分析到达谓词结尾时，它会记录原因，然后代码生成器会忽略此深度的超前预测分析。

```
class parse extends Parser;
options {
        k=2;
```

```
}
a       :       (A (P B|P )*) => A (P)*
        |       A
        ;
```

ANTLR 从谓词 (..)*里生成如下形式的一个决策：

```
if ((LA(1)==P) && (LA(2)==B)) {
    match(P);
    match(B);
}
else if ((LA(1)==P) && (true)) {
    match(P);
}
else {
    break _loop4;
}
```

这种计算在所有的文法类型中都会起作用。

## 1.20 ANTLR 元语言文法（ANTLR-meta Lanuage Grammar）

请参考 antlr/antlr.g 来了解文法，此文法描述 ANTLR 语言本身中的输入文法的语法。

**Version: $Id: //depot/code/org.ANTLR/release/ANTLR-2.7.6/doc/metalang.html#1 $**

# 第 2 章 使用 ANTLR 进行词法分析（Lexical Analysis with ANTLR）

词法分析器（通常称为扫描器）将输入的字符流分解为词汇表中的一个个的符号，然后输出到语法分析器，语法分析器将语法结构应用于那些符号流。因为 ANTLR 为词法分析、语法分析和树分析引入了相同的识别机制，ANTLR 生成的词法分析器比基于 DFA 词法分析器更强大，比如 DLG 和 lex 生成的词法分析器。

词法分析能力的提高是在一些词法分析器规范上的不方便所引起的花费，以及确实要求一个严格地关于词法分析的思维转变。请参考关于 LL(k)和基于 DFA 的词法分析的比较。

ANTLR 生成超前预测分析 LL(k)的词法分析器，这意味着你可以有一些语义和语法的谓词，并且可以使用 k>1 的超前预测分析。其它的优点在于：

- 你可以阅读和调试输出代码，因为它与你手工创建的很相似。

- 指定词法结构的语法对词法分析器（lexers）、语法分析器（parsers）和树分析器（tree parsers）来说都是相同的。
- 在识别单个记号的过程中，你可以让动作执行。
- 你可以识别复杂的记号，比如 HTML 标记，或者“可运行的”注释，像在/** ... */注释中 javadoc @-tags。词法分析器有一个堆栈，不像 DFA 那样，所以你可以匹配嵌套的结构，比如嵌套的注释。

一个词法分析器的总体结构如下：

```
class MyLexer extends Lexer;
options {
  some options
}
{
  lexer class members
}
lexical rules
```

## 2.1 词法规则（Lexical Rules）

在一个词法分析器文法中定义的规则必须有一个以大写字母开关的名字。这些规则隐示地匹配输入流的字符，而不是记号流中的记号。引用的文法元素包括记号引用（隐示地词法分析规则引用）、字符和字符串。词法分析规则按照与语法分析规则完全相同的方式处理，可以指定参数和返回值；更进一步说，词法分析规则同样可以有局部变量和使用递归。下面的规则定义了一个名为 ID 的规则，该规则名作为一个记号类型在语法分析器是可用的。

```
ID : ('a'..'z')+
   ;
```

此规则将成为最终的词法分析器的一部分，并将以一个名为 mID()的方法出现，类似如下方法：

```
    public final void mID(...)
        throws RecognitionException,
               CharStreamException, TokenStreamException
    {
        ...
        _loop3:
        do {
            if (((LA(1) >= 'a' && LA(1) <= 'z'))) {
                matchRange('a','z');
            }
```

```
        } while (...);
        ...
    }
```

熟悉 ANTLR 的输出是一个好主意——生成的词法分析器是可读的，并使很多概念变得更加清晰。

### 2.1.1 跳过字符（Skipping characters）

为了使被某个规则匹配的字符被忽略掉，设置记号类型为 Token.SKIP。例如：

```
WS : ( ' ' | '\t' | '\n' { newline(); } | '\r' )+
     { $setType(Token.SKIP); }
   ;
```

被跳过的记号迫使词法分析器复位并尝试其它的记号。被跳过的记号永远不会传递给语法分析器。

## 2.1.2 词法分析规则的区别（Distinguishing between lexer rules）

与大部分类似 lex 的词法分析器生成器一样，你只需简单地列表匹配记号的词法规则的集合。工具会自动地生成代码来将下一个输入字符映射到规则可能匹配的字符。因为 ANTLR 生成递归下降的词法分析器，就像它对语法分析器和树分析器做的一样，ANTLR 自动地为一个假想的规则生成一个称为 nextToken 的方法，以通过查看超前预测分析的字符来预测你的词法分析规则将匹配的字符。你可以把这方法想像成一个大的"switch"语句，其路径识别流向合适的规则（尽管其代码可能比一个简单的 switch 语句复杂很多）。nextToken 方法是 TokenStream（在 Java 中）的唯一方法：

```
public interface TokenStream {
    public Token nextToken() throws TokenStreamException;
}
```

语法分析器填充超前预测分析的缓冲区，并且缓冲区来自任何 TokenStream。考虑如下两个词法分析规则：

```
INT : ('0'..'9')+;
WS : ' ' | '\t' | '\r' | '\n';
```

你将会在 ANTLR 生成的词法分析器中看到一些如下的类似方法：

```
public Token nextToken() throws TokenStreamException {
    ...
```

```
    for (;;) {
        Token _token = null;
        int _ttype = Token.INVALID_TYPE;
        resetText();
        ...
        switch (LA(1)) {
          case '0': case '1': case '2': case '3':
          case '4': case '5': case '6': case '7':
          case '8': case '9':
            mINT(); break;
          case '\t': case '\n': case '\r': case ' ':
            mWS(); break;
          default: // error
        }
        ...
    }
}
```

**当相同的字符预测到不止一个词法规则时会怎样？**在冲突的规则之间，ANTLR 产生一个非确定的警告，指明你需要确保你的规则之间没有相同的左前缀。ANTLR 并不遵循常见地"第一个定义优先"词法分析规则（尽管如此，规则中的可供选择的项之间依然遵循此规则）。相反，足够地权力被赋于给处理两种最常见模棱两可的情况，也就是“关键字 vs 标识符”以及“常见的前缀”；对于特别恶心的情况，你可以使用语法或语义谓词。

**如果你希望将一个复杂的规则定义分解为多条规则，该怎样？**这种情况下，你肯定不希望每条规则都产生一个完整的 Token 对象。一些规则仅仅是用来帮助其它规则构造记号。为了区分那些“协助”规则与产生记号的规则，使用 protected 修饰符。这重载的 Java 权限访问控制术语出现了，因为如果这规则是不可见的，那它就不能被语法分析器“看到”。请参考什么是受保护的词法分析规则。

另外一个更实用的看待这种情况的方法是注意仅仅非受保护的规则由 nextToken 来调用，也就是仅仅非受保护的规则能产生可传递到通向 TokenStream 的管道的记号。

### 2.1.3 返回值（Return values）

所有的规则都会自动返回记号对象，此对象至少包含为规则匹配的文字和它的记号类型。为了指定一个用户自定义的返回值，可以定义一个返回变量，然后在动作中设置其值：

```
protected
```

```
INT returns [int v]
    :   ( '0' .. '9' )+ { v=Integer.valueOf($getText); }
    ;
```

注意仅仅受保护的规则可以有一个返回类型，因为正则词法分析规则通常是由 nextToken() 调用的，并且语法分析器不能访问返回值，这会导致冲突。

## 2.2 含谓词的 LL(k)词法分析

词法分析规则允许你的语法分析器匹配输入字符流中的上下文无关结构，而不是更弱的正则结构（使用 DFA－确定的有限状态自动机）。例如，考虑下面的情况，使用 DFA 来匹配嵌套的花括号可能使用计数器来实现，而嵌套的花括号是很平凡地被上下文无关文法所匹配

```
ACTION
    :   '{' ( ACTION | ~'}' )* '}'
    ;
```

从 ACTION 规则到 ACTION 的递归当然是一个死循环，并不是一个普通的词法分析规则。

因为同样的算法被用来分析词法分析规则和语法分析规则，词法分析规则可能使用不止一个超前预测分析的符号，可以使用语义谓词，并且也可以语法谓词来进行任意地超前查看，也就是，提供了在 LL(k)语言外、上下文相关的识别能力。下面是一个简单的要求 k>1 的超前预测分析：

```
ESCAPE_CHAR
    :   '\\' 't' // two char of lookahead needed,
    |   '\\' 'n' // due to common left-prefix
    ;
```

为了说明为词法分析规则的语法谓词使用，考虑 Pascal 中浮点数和范围的区分问题。输入 3..4 极可能被分解成 3 个记号：INT，RANGE，接下来是 INT。另一方面，输入 3.4,极可能作为一个 REAL 发送到语法分析器。麻烦在于第一个‘.’前的数字序列可以是任意长。扫描器必须消耗掉第一个‘.’来下一个字符是不是一个‘.’也就暗示了它必须回退，并把第一个数字序列当作是一个整数。使用不能回退跟踪的词法分析器使这个任务变得非常困难：没有回退跟踪，你的词法分析器必须一次能够响应不止一个的记号。尽管如此，一个语法谓词可以被用来指定何种任意的超前预测分析是需要的：

```
class Pascal extends Parser;

prog:   INT
        (   RANGE INT
            { System.out.println("INT .. INT"); }
```

```
        |   EOF
            { System.out.println("plain old INT"); }
        )
    |   REAL { System.out.println("token REAL"); }
    ;

class LexPascal extends Lexer;

WS  :   (' '
    |   '\t'
    |   '\n'
    |   '\r')+
        { $setType(Token.SKIP); }
    ;

protected
INT :   ('0'..'9')+
    ;

protected
REAL:   INT '.' INT
    ;

RANGE
    :   ".."
    ;

RANGE_OR_INT
    :   ( INT ".." ) => INT  { $setType(INT); }
    |   ( INT '.' )  => REAL { $setType(REAL); }
    |   INT                  { $setType(INT); }
    ;
```

ANTLR 词法分析规则甚至能够处理 FORTRAN 的赋值语句以及其它复杂的词法结构。考虑下面的 DO 循环：

```
DO 100 I = 1,10
```

如果中间的逗号替换成句点，循环语句将成为一个对一个称为"DO100I"的超乎寻常的变量的赋值语句：

```
DO 100 I = 1.10
```

下面的规则正确地区别了这两种情况：

```
DO_OR_VAR
    :   (DO_HEADER)=> "DO" { $setType(DO); }
    |   VARIABLE { $setType(VARIABLE); }
```

```
    ;
protected
DO_HEADER
options { ignore=WS; }
    :   "DO" INT VARIABLE '=' EXPR ','
    ;

protected INT : ('0'..'9')+;

protected WS : ' ';

protected
VARIABLE
    :   'A'..'Z'
        ('A'..'Z' | ' ' | '0'..'9')*
        { /* strip space from end */ }
    ;

// just an int or float
protected EXPR
    :   INT ( '.' (INT)? )?
    ;
```

前面的例子讨论了如何区分词法规则与大量超前预测分析（固定 k 或任意）。还有你需要打开和关闭特定的词法分析规则（使特定记号有效和失效）的其它情形，依赖于前面的上下文内容或语义信息。一个最好的例子是匹配一个记号，仅仅当它从一行的左边开始（也就是第一列）。如果不能检测词法分析器的列计数器，你就无法很好地完成此项工作。下面是一个简单的 DEFINE 规则，仅仅当语义谓词为真时才被匹配。

```
DEFINE
    :   {getColumn()==1}? "#define" ID
    ;
```

在单个可供选择的词法规则左边的语义谓词被提升进 nextToken 的预测机制。将谓词添加到一个规则使其不是一个识别的候选项，直至谓词为真。这种情况下，为 DEFINE 产生的方法将永远不会进入，即使当列数大于 1 时，超前预测分析预测到#define。

另一个有用的例子包括上下文相关识别，比如当你希望仅仅当你的词法分析器在一个特定的的上下文中时才匹配一个记号（例如，词法分析器先前匹配的一些触发序列）。如果你正在匹配分隔数据行的记号，比如"----"，你可能仅仅希望当“开始表（begin table）”序列已经被找到时才匹配这个记号。

```
BEGIN_TABLE
```

```
    :   '[' {this.inTable=true;} // 进入表上下文
    ;

ROW_SEP
    :   {this.inTable}? "----"
    ;

END_TABLE
    :   ']' {this.inTable=false;} //退出表上下文
    ;
```

这种谓词提升能力是一种对基于 DFA 的、类似 lex 的词法分析器生成器的仿真，虽然谓词更强大。（你甚至可以根据分析的阶段打开特定的规则）。:）

## 2.3 关键字和字面值（Keywords and literals）

许多语言有一个通用的“标识符”识别词法规则，关键字是标识符模式的特例情况。一个典型的标识符如下定义：

```
ID : LETTER (LETTER | DIGIT)*;
```

这通常与关键字相冲突。ANTLR 通过让你把关键字放在一个字面值表中来解决这个问题。在每个记号被匹配后，会检查字面值表（在词法分析器中通常以 hash 表来实现），所以字面值能够有效地覆盖更普通的标识符模式。字面值以下面两种方法中的一种来创建。首先，任何在语法分析器使用的双引号括起来的字符串自动地政相关词法分析器的字面值表。其次，通过 literal 选项（literal  option）的方式在词法分析规则中指定字面值。另外，testLiterals 选项（testLiterals option）能够让你精切地控制字面值测试代码的生成。

## 2.4 常见的前缀（Common prefixes）

通过增加词法分析器超前预测分析的深度，词法分析规则中固定长度的常见前缀能够很好地被处理。例如，一些来自 Java 的操作符：

```
class MyLexer extends Lexer;
options {
  k=4;
}
GT : ">";
GE : ">=";
RSHIFT : ">>";
RSHIFT_ASSIGN : ">>=";
```

```
UNSIGNED_RSHIFT : ">>>";
UNSIGNED_RSHIFT_ASSIGN : ">>>=";
```

## 2.5 记号定义文件（Token definition files）

通过记号定义文件的方式，记号定义能够从一种文法被转移到另一文法。这是通过importVocab和exportVocab选项实现的。

## 2.6 字符类（Character classes）

使用~操作符来对一个字符或字符集取反。例如，为了匹配任何除换行符外的其它字符，下面的规则引用了~'\n'。

```
SL_COMMENT: "//" (~'\n')* '\n';
```

~操作符同样可以被用来对一个字符集取反：

```
NOT_WS: ~(' ' | '\t' | '\n' | '\r');
```

范围操作符可以被用来创建一系列的序列字符集合：

```
DIGIT : '0'..'9' ;
```

## 2.7 记号属性（Token Attributes）

请参考下一章节。

## 2.8 词法超前分析和记号结束符（Lexical lookahead and the end-of-token symbol）

当分析词法的文法时,一个独特的情况会出现,类似于在分析正则文法时的文件结束符条件。考虑为分析在如下规则B中的子规则('b'|)，你将如何计算超前预测分析集合：

```
class L extends Lexer;

A: B 'b'
;

protected // 仅仅通过其它lex规则调用
B: 'x' ('b' | )
        ;
```

子规则的第一个可供选择的项的超前预测分析很清楚的是‘b’。第二个可供选择的项为空，

超前预测分析集合是所有能够跟在子规则的引用后面的字符的集合，此子规则是规则 B 的 follow 集合。这种情况中，字符‘b’跟在 B 的引用后，所以是空可选项的间接的超前预测分析集合。因为‘b’开始于两个可供选择的项，此子规则的分析决策是我们有时说的非确定或模棱两可的。ANTLR 会正确地产生一个对此规则的警告（除非你使用了 warnWhenFollowAmbig 选项）。

现在，如果规则 A 并不存在，规则 B 也不是 protected（它是一个完整的记号而不是一个“子记号”），超前预测分析会有什么意义：

```
B  :  'x' ('b' | )
   ;
```

这种情况中，空的可选择项仅仅查找到规则的结束作为超前预测分析，并且没有其它的规则引用它。更糟糕的情况中，任何字符可以跟在此规则后（也就是，下一记号或错误序列的开始）。所以那么空的可选择项的超前预测分析就不应该整个字符词汇表？以及这不应该产生一个非确定性的警告，因为它肯定与‘b’可选项冲突？从概念上来说，两个问题的答案都是肯定的。尽管如此，从一个实际的立场来说，你会很清楚地说：“嗯，在记号 B 的结束处匹配‘b’，如果你找到一个的话。”我讨论过不应该产生警告，ANTLR 匹配元素的策略会尽快做到这点。

另外一个不把超前预测分析表现为整个词汇表的原因是，'\u0000'..'\uFFFF'的词汇表实在太庞大了（一个 2 的 16 次方再除以 32 个长字的内存集合）。任何在其超前预测分析集合中含'<标记结束符（end-of-token）>'的可选择项将被代码生成器压入 ELSE 或 DEFAULT 从句中，因此庞大的位集可以避免。

总结是单纯由遇到词法分析规则结束而得到的超前预测分析不能是导致非确定的一个原因。下表总结了一系列的情况，有助于帮助你弄明白何时 ANTLR 将抱怨，何时不会。

| | |
|---|---|
| `X: 'q' ('a')? ('a')?`<br>`  ;` | 第一个子规则是不确定的，因为第二个子规则（标记结束符）里的'a'在退出分支(...)?的超前预测分析中。 |
| `X: 'q' ('a')? ('c')?`<br>`  ;` | 确定的。 |
| `Y: 'y' X 'b'`<br>`  ;`<br>`protected`<br>`  X: 'b' \|`<br>`  ;` | 规则 X 中存在非确定性。 |
| `X: 'x'`<br>`('a' \| 'c' \| 'd')+ \| 'z'` | 没有非确定性，因为循环的退出分支查看单纯根据标记结束符计算得到的超前预测分析。 |

| | |
|---|---|
| ('a')+<br>;| |
| Y:'y' ('a')+ ('a')?<br>; | (...)+中的'a'和退出分支之间存在非确定性，因为退出时能够看到可选子规则的'a'。即使('a')?简单地是'a'，这也将是一个问题。(...)*会产生相同的问题。 |
| X:'y' ('a' 'b')+ 'a' 'c'<br>; | 在 k=1 时，对来说(...)?，这是一个非确定的，因为'a'能够预测继续循环和退出循环。在 k=2 时，没有非确定性。 |
| Q: 'q' ('a' \| )?<br>; | 这里，在一个可选的子规则中存在一个空的可供选择的项。会报告存在一个非确定性，因为两条路径都可能预测标记结束符。 |

你也许想知道为什么下面的第一个子规则是模棱两可的：

```
('a')? ('a')?
```

答案是 NFA 到 DFA 的转换会导致含‘a’的转移的一个 DFA 合并到一个单独的状态转移中去。对一个除了在一个完整的匹配后，你不能有动作（action）的 DFA 来说，这样没问题。记住 ANTLR 允许你如下使用规则：

```
('a' {do-this})? ('a' {do-that})?
```

另外还有一件其它的事情知道很重要。在词法分析规则中的可选项的重新调用会根据它们超前预测分析的要求重新排序，从最高到最低。

```
A  :  'a'  |  'a' 'b'
   ;
```

在 k=2 时，ANTLR 可以看到第一个可选项的‘a’后面跟着‘<标记结束符（end-of-token）>’，以及第二个可选项的‘a’后面跟着‘b’。对第一个可选项深度为 2 的超前预测分析是‘<标记结束符（end-of-token）>’并抑制了一个警告，深度为 2 能够匹配第一个可选项的任意字符。当没有警告产生时，为了行为自然和生成好的代码，ANTLR 对可选项重新排序，所以生成的代码类似如下代码：

```
A() {
        if ( LA(1)=='a' && LA(2)=='b' ) { // 可选项 2
                match('a'); match('b');
        }
        else if ( LA(1)=='a' ) { // 可选项 1
                match('a')
        }
        else {error;}
}
```

注意可选项 1 的深度为 2 的超前预测分析的缺失。当出现一个空的可选项时，ANTLR 将其移到末尾。例如：

```
A       :       'a'
        |
        |       'a' 'b'
```

```
        ;
```

产生的类似如下的代码：

```
A() {
        if ( LA(1)=='a' && LA(2)=='b' ) { // alt 2
                match('a'); match('b');
        }
        else if ( LA(1)=='a' ) { // alt 1
                match('a')
        }
        else {
        }
}
```

注意这里无法出现词法分析错误（这样做有意义，因为此规则是可选的——虽然这个规则仅仅当是 protected 时有意义）。

当可选项根据超前预测分析的深度排序时，语义谓词会与其相关的可选项一起移动。如果一个{true}?谓词（隐示地存在于每一个可选项）的增加改变了词法分析器识别的内容，这会很诡异。下列规则被重新排序，所以可选项 2 首先被检测。

```
B  :  {true}? 'a'
        |  'a' 'b'
        ;
```

语法谓词不会被重新排序。说起规则后的谓词，它与结果在不明确性上存在冲突，比如此条规则中：

```
F  :  'c'
    | ('c')=> 'c'
        ;
```

尽管如此，其它的可选项会关于语法谓词重新排序，即使为 LL(1)组件生成了 switch 语句并语法谓词被压入 default 语句中。下面的规则解释了这点。

```
F  :  'b'
    |   {/* empty-path */}
    |   ('c')=> 'c'
    |   'c'
    |   'd'
    |   'e'
   ;
```

规则 F 的决策会生成为如下所示：

```
        switch ( la_1) {
        case 'b':
        {
            match('b');
            break;
```

```
        }
        case 'd':
        {
            match('d');
            break;
        }
        case 'e':
        {
            match('e');
            break;
        }
        default:
            boolean synPredMatched15 = false;
            if (((la_1=='c'))) {
                int _m15 = mark();
                synPredMatched15 = true;
                guessing++;
                try {
                    match('c');
                }
                catch (RecognitionException pe) {
                    synPredMatched15 = false;
                }
                rewind(_m15);
                guessing--;
            }
            if ( synPredMatched15 ) {
                match('c');
            }
            else if ((la_1=='c')) {
                match('c');
            }
            else {
                if ( guessing==0 ) {
                    /* empty-path */
                }
            }
        }
```

注意在检测‘c’可选项后，空路径是如何被移动的?

## 2.9 扫描二进制文件（Scanning Binary Files）

字符常量并不限于可打印的 ASCII 字符。为了说明这个概念，假如你想解析一个包含字符串和短整型整数的二进制文件。为了区分它们，根据下列格式使用了的标记字节：

| 格式 | 描述 |
| --- | --- |
| '\0' *高位 低位* | 短整型 |
| '\1' *非'\2'的字符串 字符* '\2' | 字符串 |

简单的输入（274 后面接着是“a test”）可能如下十六进制所示（UNIX 命令 od -h 的输出）：

```
0000000000    00 01 12 01 61 20 74 65 73 74 02
```

或者以字符形式查看：

```
0000000000    \0 001 022 001 a      t  e  s  t 002
```

语法分析器，很一般地，仅仅就是一个关于两种输入标记类型的(...)+：

```
class DataParser extends Parser;

file:   (   sh:SHORT
            {System.out.println(sh.getText());}
        |   st:STRING
            {System.out.println("\""+
               st.getText()+"\"");}
        )+
    ;
```

所有有趣的事情发生在词法分析器中。首先，定义类并且设置词汇表为所有的 8 位二进制值：

```
class DataLexer extends Lexer;
options {
    charVocabulary = '\u0000'..'\u00FF';
}
```

然后，根据说明定义两个标记，字符串带有多个标记字节，短整型前有一个标记字节：

```
SHORT
    :   // match the marker followed by any 2 bytes
        '\0' high:. lo:.
        {
        // pack the bytes into a two-byte short
        int v = (((int)high)<<8) + lo;
        // make a string out of the value
        $setText(""+v);
        }
    ;
```

```
STRING
    :   '\1'!   // begin string (discard)
        ( ~'\2' )*
        '\2'!   // end string (discard)
    ;
```

为了调用语法分析器，使用如下类似的程序：

```
import java.io.*;

class Main {
    public static void main(String[] args) {
        try {
            // use DataInputStream to grab bytes
            DataLexer lexer =
              new DataLexer(
                new DataInputStream(System.in)
              );
            DataParser parser =
                new DataParser(lexer);
            parser.file();
        } catch(Exception e) {
            System.err.println("exception: "+e);
        }
    }
}
```

Version: $Id: //depot/code/org.antlr/release/antlr-2.7.6/doc/lexer.html#1 $

# 第 3 章 ANTLR 的树分析器

## 曾经的 SORCERER

在 ANTLR 2。xx 版本中，只要增加一些树操作符,就可以帮助你建立一种中间形式的树结构(抽象语法树) 来重写语法规则和语义动作（action）。ANTLR 同样允许你去指定 AST 树的文法结构，因此，可以通过操作或简单遍历树结点的方式来进行文法翻译。

以前，树分析器用一个单独的工具 SORCERER 来生成，但是 ANTLR 已经取代了它的功能。ANTLR 现在可以为字符流，记号流，以及树结点来建立识别器。

## 3.1 什么是树分析器？

分析是将语法结构应用于输入的记号流的过程。ANTLR 在这方面比大多数工具考虑的都要深，它把一颗树看作是二维的结点流。实际上，在 ANTLR 中，对记号流进行分析和对树的进行分析生成的代码生成过程来说，真正仅有的区别就变成了对超前扫描，规则方法定义头部的检测，以及对二维树结构代码生成模板的指定上。

## 3.2 可以分析什么类型的树？

ANTLR 树分析器可以遍历实现了 AST 接口的任何树。AST 接口是一种基于类似儿子-兄弟结点的树通用结构，有如下重要的制导方法：

- getFirstChild: 返回第一个子结点的引用。
- getNextSibling: 返回下一个兄弟结点的引用。

每一个 AST 结点有一个子女列表，一些文本和一个"记号类型"。每个树的结点都是一棵树，因此我们说树是自相似的（也即树是递归定义的：译者注）。AST 接口的完整定义如下：

```
/** 最小 AST 结点接口用于 ANTLR 的 AST 成生和树遍历
 */
public interface AST {
    /** 添加一个子结点到最右边 */
    public void addChild(AST c);
    public boolean equals(AST t);
    public boolean equalsList(AST t);
    public boolean equalsListPartial(AST t);
    public boolean equalsTree(AST t);
    public boolean equalsTreePartial(AST t);
    public ASTEnumeration findAll(AST tree);
    public ASTEnumeration findAllPartial(AST subtree);
    /** 得到第一个子结点； 如果没有子结点则返回 null */
    public AST getFirstChild();
```

```
    /** 得到本结点的下一个兄弟结点 */
    public AST getNextSibling();
    /** 得到本结点的记号文本 */
    public String getText();
    /** 得到本结点的记号类型 */
    public int getType();
    /** 得到本结点的子结点总数； 如果是叶子结点，返回 0 */
    public int getNumberOfChildren();
    public void initialize(int t, String txt);
    public void initialize(AST t);
    public void initialize(Token t);
    /** 设置第一个子结点。 */
    public void setFirstChild(AST c);
    /** 设置下一个兄弟结点。 */
    public void setNextSibling(AST n);
    /** 设置本结点的记号文本 */
    public void setText(String text);
    /** 设置本结点的记号类型 */
    public void setType(int ttype);
    public String toString();
    public String toStringList();
    public String toStringTree();
}
```

## 3.3 树的语法规则

正如 PCCTS1。33 的 SORCERER 工具和 ANTLR 记号语法中所看到的，树语法是一个嵌入语义动作（action），语义断言和句法断言的 EBNF 规则的集合。

```
规则:       可选产生式 1
    |       可选产生式 2
```

```
    ...
      |         可选产生式 n
      ;
```

每一个可选的产生式都是由一个元素列表所组成，列表中的元素是加入了树模式的 ANTLR 正则表达式语法中的项，有如下的形式：

```
#( 根结点 子结点 1 子结点 2 ... 子结点 n )
```

例如：下列的树模式匹配一个以 PLUS 为根结点，并有两个 INT 子结点的简单树结构：

```
#( PLUS INT INT )
```

树模式的根结点必须是一个记号引用，但是子结点元素不限于此，它甚至可以是子规则。例如，一种常见结构是 if-then-else 树结构，其中的 else 子句声明的子树是可选的：

```
#( IF expr stat (stat)? )
```

值得一提的是，当指定树模式和树语法后，通常，会进行满足条件的匹配而不是精确的匹配。一旦树满足给定的模式，不管剩下多少没有分析，都会报告一次匹配。例如，#( A B )，对于像#( A #(B C) D)这样有相同结构的树，不管有多长，都会报告一次匹配。

## 3.4 句法断言

ANTLR 树分析器在工作时仅使用一个单独的超前扫描记号，这在通常情况下不是一个问题，因为这种中间形式被明确设计成利于遍历的结构。然而，偶尔也需要区别出相似的树结构。句法断言就是被用来克服有限确定的超前扫描所带来的限制。例如：在区分一元和二元减号时，可以为每一种类型的减号都创建不同记号的操作结点，但赋与相同的根结点，这样的处理方法可以工作的很好。使用句法断言可以区分以下结构：

```
expr:   ( #(MINUS expr expr) )=> #( MINUS expr expr )
      |   #( MINUS expr )
    ...
     ;
```

赋值的次序很重要，因为第二个可选产生式是第一个可选产生式的“子集”。

## 3.5 语义断言

在可选产生式开始部分的语义断言，只是简单地与可选断言表达式合成一体，就像合成正则文法一样。产生式中间的语义断言，当失败时，也会像正则文法一样抛出异常。

## 3.6 一个树遍历器的例子

考虑一下如何去建立一个简单的计算器。一个方法是建立一个分析器，识别输入并计算表达式的值。为了说明这种方法，我们将会建立一个分析器来为输入的表达式创建一棵树，并把表达式以这种中间形式表示，然后树分析器遍历这个中间表达式，并计算出结果。

我们的识别器, CalcParser, 通过如下的代码来定义：

```
class CalcParser extends Parser;
options {
    buildAST = true;    // // 默认使用 CommonAST
}
expr:   mexpr (PLUS^ mexpr)* SEMI!
    ;
mexpr
    :   atom (STAR^ atom)*
    ;
atom:   INT
    ;
```

PLUS 和 STAR 记号是操作符，因此把它们作为子树的根结点，在它们后面注释上字符'^'。SEMI 记号后缀有字符'!'，表明它不应该被加入到树中。

这个计算器的词法分析器定义如下：

```
class CalcLexer extends Lexer;
WS :        (' '
   |        '\t'
   |        '\n'
```

```
    |       '\r')
            { _ttype = Token。SKIP;  }
    ;
LPAREN:     '('
    ;
RPAREN:     ')'
    ;
STAR:       '*'
    ;
PLUS:       '+'
    ;
SEMI:       '; '
    ;
INT         :       ('0'..'9')+
    ;
```

识别器生成的树是一棵简单的表达式树。例如，输入"3*4+5"将产生形如#( + ( * 3 4 ) 5 )的树。为了给这种形式的树建立树遍历器，你必须要为 ANTLR 递归地描述树的结构：

```
class CalcTreeWalker extends TreeParser;
expr :      #(PLUS expr expr)     //PLUS 为根结点,两个 expr 分别为左右子结点
    |       #(STAR expr expr)
    |       INT
    ;
```

一旦指定了结构，你就可以嵌入语义动作（action）来计算正确的结果。一个简单的实现办法就是使 expr 规则返回一个整型的值，然后让每一条可选产生式来计算每个子树的值。下面的树文法和动作（action）达到了我们期望的效果：

```
class CalcTreeWalker extends TreeParser;
expr returns [int r]
{
```

```
    int a,b;
    r=0;
}
    :       #(PLUS a=expr b=expr) {r = a+b; }
    |       #(STAR a=expr b=expr) {r = a*b; }
    |       i:INT                 {r = Integer。parseInt(i。getText()); }
    ;
```

注意到当计算表达式值得时候，没有必要指定优先级，因为它已经隐含在树的结构中了。这也解析了为什么以中间树形式的表示比以树的形式复制输入的表示要重要。输入的记号确实作为结点储存在树结构中，而且这种结构隐含了结点之间的关系。

要想执行分析器和树遍历器，还需要以下的代码：

```
import java.io.*;
import ANTLR.CommonAST;
import ANTLR.collections.AST;

class Calc {
    public static void main(String[] args) {
        try {
            CalcLexer lexer =
                new CalcLexer(new DataInputStream(System。in));
            CalcParser parser = new CalcParser(lexer);
            // 分析输入的表达式
            Parser.expr();
            CommonAST t = (CommonAST)parser.getAST();
            // 以 LISP 记号的形式输出树
            System.out.println(t.toStringList());
            CalcTreeWalker walker = new CalcTreeWalker();
            // 遍历由分析器建立的树
            int r = walker.expr(t);
```

```
            System.Out.println("value is "+r);
        } catch(Exception e) {
            System.err.println("exception: "+e);
        }
    }
}
```

## 3.7 翻译

树分析器对检查树或者从一棵树产生输出来说是非常有用，但必须要为它们添加处理树转换的代码。就像正则分析器一样，ANTLR 树分析器支持 buildAST 选项，这类似于 SORCERER 的翻译模式。不需要程序员的参与，树分析器自动把输入树拷贝到结果的树中。每一个规则都隐含(自动定义的)一颗结果树。通过 getAST 方法，我们可以从树分析器中获得此树的开始记号。在一些可选产生式和文法元素后面注释上"!"，将意味着不被自动输出到输出树。部分或全部子树都可以被重写。

嵌入到规则中的语义动作（action）可以根据测试和树结构来对结果树进行设置。参考文法动作（action）翻译章节。

## 3.8 一个树翻译的例子

再来看一下上面提到的简单计算器的例子，我们可以执行树翻译来代替计算表达式的值。下面树文法中的动作（action）优化了加法的恒等运算(加 0)。

```
class CalcTreeWalker extends TreeParser;
options{
    buildAST = true;  // "翻译"模式
}

expr:!  #(PLUS left:expr right:expr)
        // '!'关闭自动翻译
        {
```

```
            // x+0 = x
            if ( #right.getType()==INT &&
                 Integer.parseInt(#right.getText())==0 )
            {
                #expr = #left;
            }
            // 0+x = x
            else if ( #left.getType()==INT &&
                      Integer.parseInt(#left.getText())==0 )
            {
                #expr = #right;
            }
            // x+y
            else {
                #expr = #(PLUS, left, right);
            }
        }
    |   #(STAR expr expr)  // 使用自动翻译
    |   i:INT
    ;
```

执行分析器和树翻译器的代码如下：

```
import java.io.*;
import ANTLR.CommonAST;
import ANTLR.collections.AST;

class Calc {
    public static void main(String[] args) {
        try {
```

```
            CalcLexer lexer =
                new CalcLexer(new DataInputStream(System。in));
            CalcParser parser = new CalcParser(lexer);
            // 分析输入的表达式
            Parser.expr();
            CommonAST t = (CommonAST)parser.getAST();
            // 以 LISP 记号的形式输出树
            System.Out.println(t。toLispString());

            CalcTreeWalker walker = new CalcTreeWalker();
            // 遍历由分析器建立的树
            walker.expr(t);
            // 遍历，并得到结果
            t = (CommonAST)walker.getAST();
            System.Out.println(t.toLispString());
        } catch(Exception e) {
            System.err.println("exception: "+e);
        }
    }
}
}
```

## 3.9 检查/调试 AST

当开发树分析器的时候，经常会遇到分析错误。不幸的是，你的树通常异乎寻常的大，使得很难去确定 AST 结构错误到底在哪里。针对这种情况(当创建 Java 树分析器的时候,我发现它非常有用)，我创建了一个 ASTFrame 类(一个 JFrame 类)，这样，你就可以用 Swing 树视图来查看你的 AST。它没有拷贝这棵树，而是用了一个 TreeModel。以应用程序方式运行 ANTLR.debug.misc.ASTFrame 去或者看看 Java 代码 Main。java。就像不确定如何去调试

一样,我不确定它们在相同的包下,总之,将会在以后的 ANTLR 版本中给出。这里有一个简单的使用例子：

```
public static void main(String args[]) {
  // 创建树结点
  ASTFactory factory = new ASTFactory();
  CommonAST r = (CommonAST)factory.create(0, "ROOT");
  r.addChild((CommonAST)factory.create(0, "C1"));
  r.addChild((CommonAST)factory.create(0, "C2"));
  r.addChild((CommonAST)factory.create(0, "C3"));

  ASTFrame frame = new ASTFrame("AST JTree Example", r);
  frame.setVisible(true);
}
```

Version: $Id: //depot/code/org。ANTLR/release/ANTLR-2.7.6/doc/sor.html#1 $

# 第 4 章 记号流（Token Streams）

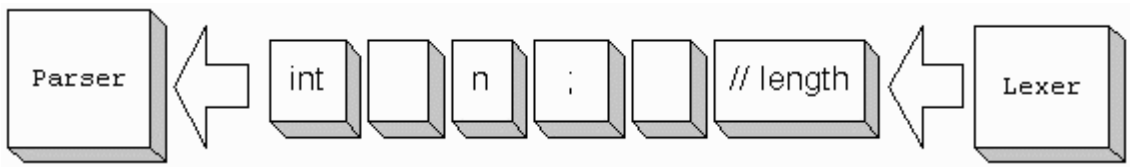
长久以来, 词法分析器和语法分析器是紧紧耦合在一起的;也就是说, 你不可以在他们中间做任何事情,也不能修改记号流。但是，用记号流来处理词法分析器和语法分析器之间的连接的话，会给代码识别和翻译带来极大的帮助。这个想法类似于 Java 的 I/O 流，利用 I/O 流你可以以管道的方式将大量的流对象组织更高层次的数据流。

## 4.1 引言

ANTLR 能识别任何满足 TokenStream 接口的记号流对象(2。6 以前的版本, 这个接口叫做 Tokenizer)；也就是说记号流对象要实现以下的方法：

```
Token nextToken();
```

分析过程中，从某种角度上说，从词法分析器(生产者)到语法分析器(消费者)的普通记号流如下图所示：




最普通的记号流是一个词法分析器，但是想象一下，如果在词法分析器和语法分析器中间有一个流的实体，你就可以做一些有趣的事情。例如，你可以：

- 过滤掉不想要的记号
- 插入一些辅助的记号，帮助语法分析识别一些模棱两可的结构
- 把一个流分成多个流，把某些感兴趣的记号传送到不同的流中
- 把多个记号流合并成一个流，从而“模拟”PCCTS,lex 等词法分析工具的状态。

记号流的概念的意义在于词法分析器和语法分析器不在互相影响--它们只不过是流的生产者和消费者。流对象是消费者用来产生、处理、合并或者分离记号流的过滤器。可以使已有的词法分析器和语法分析器在不修改的情况下合并成一种新的工具。

这份文档正式提出了记号流的概念，详细描述了一些非常有用的流过滤器。

## 4.2 自由通过记号流

一个记号流可以是任何满足下面接口的对象：

```
public interface TokenStream {
  public Token nextToken()
    throws java.io.IOException;
}
```

例如, 一个"无操作"或者说仅仅传递记号的过滤器就像如下这样：

```
import ANTLR。*;
import java.io.IOException;

class TokenStreamPassThrough
    implements TokenStream {
  protected TokenStream input;
```

```
    /** Stream to read tokens from */
    public TokenStreamPassThrough(TokenStream in) {
      input = in;
    }

    /** This makes us a stream */
    public Token nextToken() throws IOException {
      return input.nextToken();  // "short circuit"
    }
  }
```

你可以使用一个简单的流对象从词法分析器中获得记号，然后语法分析器再从这个流对象中获得记号，就像下面的 main()程序一样：

```
  public static void main(String[] args) {
    MyLexer lexer =
      new MyLexer(new DataInputStream(System。in));
    TokenStreamPassThrough filter =
      new TokenStreamPassThrough(lexer);
    MyParser parser = new MyParser(filter);
    Parser.startRule();
  }
```

## 4.3 记号流过滤

多数情况下，你希望词法分析器丢弃掉空白符和注释，然而，如果你还希望在语法分析器必须使用注释的情况下重用词法分析器呢？这时，你只需要设计一个将空白符和注释与普通记号一起传递给语法分析器的简单的词法分析器来满足大多应用。然后，当你想忽略空白符的时候，只要在词法分析器和语法分析器中间加入一个过滤器，过滤掉空白符。

针对这种情况,ANTLR 提供了 TokenStreamBasicFilter。你可以在不修改词法分析器的情况下让它过滤掉任何类型的记号或记号集。下面 TokenStreamBasicFilter 的用法的例子中过滤掉了注释和空白符。

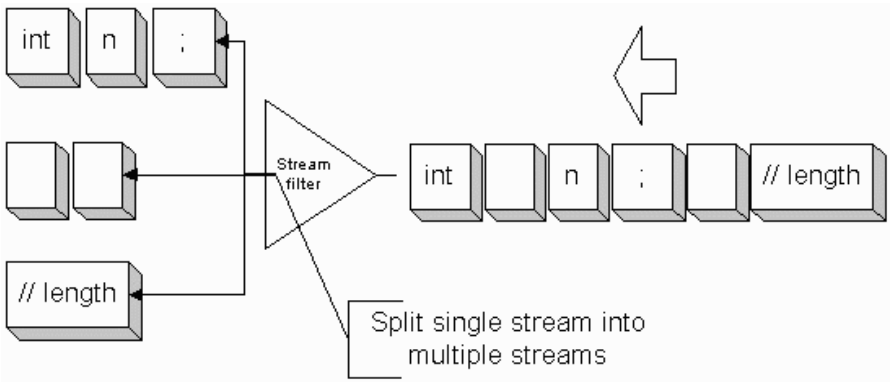
```
public static void main(String[] args) {
  MyLexer lexer =
    new MyLexer(new DataInputStream(System。in));
  TokenStreamPassThrough filter =
    new TokenStreamPassThrough(lexer);
  filter.discard(MyParser。WS);
  filter.discard(MyParser。COMMENT);
  MyParser parser = new MyParser(filter);
  parser.startRule();
}
```

可以看到，它比修改词法分析器的词法结构要来的有效，你也会这么做的吧，因为这样你不用去构建一个记号对象。另一方面，采用这种过滤流的方法使词法分析器的设计更加灵活。

## 4.4 记号流分离

有时，在识别阶段，你想让翻译器忽略而不是丢弃输入的部分记号。比如说，你想在语法分析时忽略注释，但在翻译时又需要注释。解决办法是将注释发送到一个隐藏的记号流中，所谓隐藏，就是语法分析器没有对它进行监听。在识别期间，通过动作（action）来检查这些隐藏的流，收集注释等等。流分离过滤器就像棱镜把白光分离成彩虹。

下面的图中示出了把一个记号流分成三个的情况。




让语法分析器从最上面的流中获得记号。

用流分离器可以实现很多功能。比如，"Y-分离器"像有线电视 Y 连接器一样，复制记号流。如果过滤器是线程安全的而且有缓冲器缓冲，过滤器就可以同时为多个语法分析器提供记号。

这一节描述 ANTLR 提供的一个叫做 `TokenStreamHiddenTokenFilter` 的流过滤器,它类似于给一堆硬币分类,把一分的放到一个箱子里,把一角的放到另一个箱子里,等等。这个过滤器把输入流分离成两个流，一个包含主要记号，另一个被缓冲以便以后可以访问。因为这种实现方式，无论怎么做，你都无法让语法分析器直接访问隐藏流。下面你将会看到，过滤器实际上把隐藏记号交织在主记号中。

### 4.4.1 例子

考虑以下的简单文法,该文法用来声明整型变量。

```
decls: (decl)+
     ;
decl : begin:INT ID end:SEMI
     ;
```

比如说有以下输入:

```
int n;  // list length
/** doc */
int f;
```

假定词法分析器忽略空白符,你可以用过滤器把注释分离到一个隐藏流。那么现在如果语法分析器从主记号流中获得记号，它只会看到"INT ID SEMI FLOAT ID SEMI",注释在隐藏

流中。语法分析器可以忽略注释，而语义动作（action）可以从过滤器中查询隐藏流中的记号。

第一次调用文法规则 decl 前后,begin 记号都没有对隐藏记号的引用,但

filter.getHiddenAfter(end)

返回一个对下面记号的引用

```
// list length
```

接下来就会访问到

```
/** doc */
```

第二次调用文法规则 decl 时

```
filter.getHiddenBefore(begin)
```

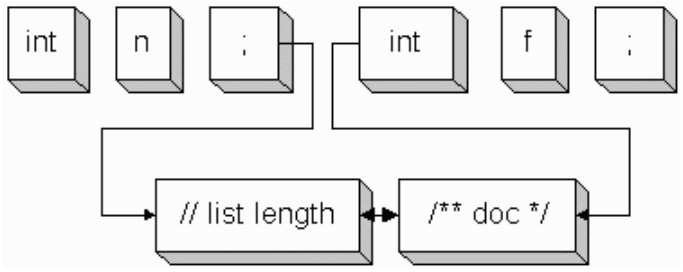
指向

```
/** doc */
```

的引用

### 4.4.2 过滤器实现

下图阐述了记号对象实际上是如何组织记号来模拟两个不同的流：




随着记号的读取，TokenStreamHiddenTokenFilter 对象通过链表来连接隐藏记号和主记号。过滤器只提供了一个物理上的记号流，通过交叉指针维护和管理记号次序信息。

因为额外的指针需要把记号连接到一起，必须要用一个叫 CommonHiddenStreamToken 的特殊记号对象(普通记号对象叫做 CommonToken)。前面曾说过，可以用如下方法指定词法分析器为特定的类创建记号：

```
lexer.setTokenObjectClass("classname");
```

从技术上讲，不需要特殊的记号对象，也可以实现同样功能的过滤器，但这样实现非常有效而且它很容易告诉词法分析器去生成什么样的记号。进一步说，这样实现使得很容易地自动创建树的结点，同时保留隐藏流的信息。

这个过滤器影响 ANTLR 的延缓消耗（lazy-consume）。在识别每一个主记号之后，TokenStreamHiddenTokenFilter 必须查看下一个记号是不是隐藏记号。因此，这个过滤器在交互程序(比如命令行)下工作得不是很好。

### 4.4.3 如何使用这个过滤器

要使用 TokenStreamHiddenTokenFilter，你所要做的是：

- 创建词法分析器，让它创建链接隐藏记号的记号对象。

```
MyLexer lexer = new MyLexer(some-input-stream);
lexer.setTokenObjectClass(
  "ANTLR.CommonHiddenStreamToken"
);
```

- 创建一个 TokenStreamHiddenTokenFilter 对象，从前面创建的词法分析器中读取记号。

```
TokenStreamHiddenTokenFilter filter =
  new TokenStreamHiddenTokenFilter(lexer);
```

- 告诉 TokenStreamHiddenTokenFilter 要隐藏哪些记号，要丢弃哪些记号。例如,

```
filter.discard(MyParser。WS);
filter.hide(MyParser。SL_COMMENT);
```

- 创建一个语法分析器，从 TokenStreamHiddenTokenFilter 而不是从词法分析器中读取记号。

```
MyParser parser = new MyParser(filter);
try {
```

```
      parser.startRule();  // parse as usual
    }
    catch (Exception e) {
      System.err.println(e.getMessage());
    }
```

可以查看 ANTLR 指南，在 preserving whitespace 处有一个完整的例子。

### 4.4.4 树的创建

最后，在翻译阶段会需要这些隐藏的流记号，通常也就是遍历树的时候。怎么做才能在不打乱树文法的情况下把隐藏流的信息送给翻译器呢？很简单： 用 AST 结点储存这些隐藏流记号。ANTLR 定义了 CommonASTWithHiddenTokens 来自动连接隐藏流中的记号到树结点；有方法可以访问与树结点相关的隐藏记号。你所需要做的是告诉语法分析器去创建这种类型的树结点而不是默认的 CommonAST 类型的结点：

```
parser.setASTNodeClass("ANTLR.CommonASTWithHiddenTokens");
```

树结点作为记号对象的功能被创建。当 ASTFactory 创建树结点的时候，树结点的 initialize()方法会被调用。根据包含隐藏记号的记号创建的树结点也会包含相同的隐藏记号。你没必要使用这结点定义，但它在很多翻译任务中起作用：

```
package ANTLR;

/** CommonAST 在初始化时把从记号中获得
 *  的隐藏记号的信息复制，用来创建结点
 */
public class CommonASTWithHiddenTokens
  extends CommonAST {
  // 指向隐藏记号
  protected Token hiddenBefore, hiddenAfter;

  public CommonHiddenStreamToken getHiddenAfter() {
```

```
    return hiddenAfter;
  }
  public CommonHiddenStreamToken getHiddenBefore() {
    return hiddenBefore;
  }
  public void initialize(Token tok) {
    CommonHiddenStreamToken t =
      (CommonHiddenStreamToken)tok;
    super。initialize(t);
    hiddenBefore = t.getHiddenBefore();
    hiddenAfter  = t.getHiddenAfter();
  }
}
```

注意到这种结点的定义假设你使用了 CommonHiddenStreamToken 对象。如果你没有让词法分析器创建 CommonHiddenStreamToken 对象，就会出现运行时类型转换异常。

### 4.4.5 垃圾回收

通过分离输入流以及把隐藏记号流与主记号流分离出来，GC（Garbage Collection）可以在此记号流上起作用。在上面整数声明的例子中，当没有对第一个 SEMI 记号以及第二个 INT 记号的更多引用时，注释记号将会作为垃圾回收的候选。如果所有的记号是连在一起的，一个单独的对任意记号的引用会阻止任何记号被回收。在 ANTLR 实现中，事实并非如此。

### 4.4.6 附注

翻译时,过滤器在保存空白符和注释方面做得很好，但在处理输出和输入差别很大的情况下，用过滤器并不是一个好的办法。例如，有 3 个注释分散在一个输入语句中，你想在翻译阶段把注释合并到输出声明语句的头部。与通过查看每一个已分析的记号来确定其周围的注释相比，更好的办法是有一个真正的、物理上分开的流来缓存注释以及一种方法来联系分析好的记号组与注释流记号组。你或许会支持像"给我在注释流上从开始分析到结束分析时最初出现的所有记号" 的问题。

这个过滤器实现了同 JavaCC 中特殊记号一样的功能。Sriram Sankar（JavaCC 之父）关于特殊记号有一个非常好的想法，在 1997 的 Dr. T's Traveling Parsing Revival and Beer Tasting Festival，出席者把这种想法扩展到更广泛的记号流概念。现在 JavaCC 特殊记号的功能正是另一个 ANTLR 的流过滤器，好处是你不必修改词法分析器来指定哪些记号是特殊的。

## 4.5 记号流多路技术 (又叫 "词法分析器多状态")

现在，考虑一下相反的问题，你需要的是把多个流合并成一个流而不是把一个流分解成多个流。当你的输入中包含差别很大的代码片段时，比如说 Java 和 JavaDoc 的注释，你会发现仅用一个词法分析器去识别所有的输入段很困难。这主要是因为合并不同部分的记号定义会造成二义性词法语言或者识别出一些错误的记号。例如，"final"在某些部分里是一个关键字，但在另一个部分里它可能会是一个标识符。同样，"@author"是一个合法的 javadoc 注释里的记号，但在 Java 代码中，它是不合法的。

很多人为了解决这个问题，为词法分析器设定了很多状态，在不同的部分里切换到不同的状态（例如，在"读取 Java 模式"和"读取 JavaDoc 模式"中间切换）。词法分析器开始是以 Java 模式工作的，然后在遇到"/**"后切换到 JavaDoc 模式； "*/"强制切换回 Java 模式。

### 4.5.1 多词法分析器

让一个词法分析器可以运行在多个状态下可以解决上述的问题，但让多个词法分析器协同工作，在一个记号流上进行多路分析，能够更好地解决问题，因为独立的词法分析器更容易重用(不是剪切粘贴到一个新的词法分析器，而是让流的多路切换器来切换到不同的词法分析器)。例如，JavaDoc 词法分析器可以在解决任何有 JavaDoc 注释的语言问题时得到重用。

ANTLR 提供了一个预定义的 `TokenStreamSelector` 记号流，可以用它在多个词法分析器间进行切换。不同词法分析器中定义的动作（action）控制选择器如何切换输入流。考虑下面的 Java 代码片段。

```
/** Test。
 *  @author Terence
```

```
 */
int n;
```

给定两个词法分析器：JavaLexer 和 JavaDocLexer，两个词法分析器的动作（action）序列看上去可能如下：

```
JavaLexer: 匹配 JAVADOC_OPEN, 切换到 JavaDocLexer
JavaDocLexer: 匹配 AUTHOR
JavaDocLexer: 匹配 ID
JavaDocLexer: 匹配 JAVADOC_CLOSE, 切换回 JavaLexer
JavaLexer: 匹配 INT
JavaLexer: 匹配 ID
JavaLexer: 匹配 SEMI
```

在 Java 词法分析器的文法中，你需要定义一个规则去切换到 JavaDoc 词法分析器(把需要切换的词法分析器记录在堆栈中)：

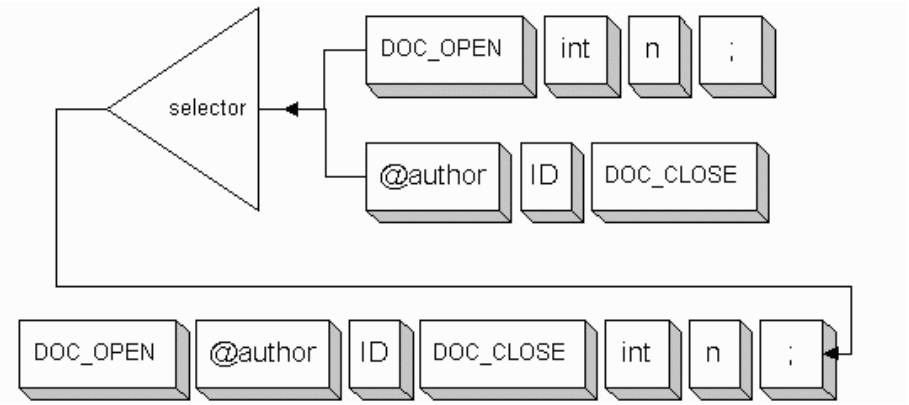
```
JAVADOC_OPEN
    :   "/**" {selector。push("doclexer"); }
    ;
```

同样地，在 JavaDoc 词法分析器中定义一个规则切换回去：

```
JAVADOC_CLOSE
    :   "*/" {selector。pop(); }
    ;
```

选择器中有一个堆栈，所以 JavaDoc 词法分析器不需要知道谁调用了它。

如图，选择器把两个词法分析流合并成一个流并提供给后续的语法分析器：




选择器会为你维护流列表，所以你可以通过名字或者实际对象的引用来切换到另一个输入流。

```
public class TokenStreamSelector implements TokenStream {
  public TokenStreamSelector() {…}
  public void addInputStream(TokenStream stream,
    String key) {…}
  public void pop() {…}
  public void push(TokenStream stream) {…}
  public void push(String sname) {…}
  /** Set the stream without pushing old stream */
  public void select(TokenStream stream) {…}
  public void select(String sname)
    throws IllegalArgumentException {…}
}
```

使用选择器很容易:

- 创建一个选择器。

```
TokenStreamSelector selector =
  new TokenStreamSelector();
```

- 为流命名(不是一定要命名--在切换的时候你可以使用流对象的引用来避免使用哈希表查找)。

```
selector.addInputStream(mainLexer, "main");
selector.addInputStream(doclexer, "doclexer");
```

- 选择哪一个词法分析器先读取字符流。

```
// start with main java lexer
selector.select("main");
```

- 将语法分析器与选择器关联而不是与每一个词法分析器关联。

```
JavaParser parser = new JavaParser(selector);
```

### 4.5.2 词法分析器共享同一字符流

在介绍语法分析器如何使用选择器之前,注意两个词法分析器都要从同一个输入流中读取字符。在 ANTLR2.6.0 以前的版本中，每一个单独的词法分析器都有它自己的记录行号的变量、输入字符流变量等等。为了共享同样的输入状态，ANTLR2.6.0 代理词法分析器的部分功能，将输入的字符输出到一个 LexerSharedInputState 对象中，从而可以被 n 个词法分析器共享(单线程)。为了让多个词法分析器共享状态,你需要创建第一个词法分析器，获得它的输入状态对象，然后在构建其它词法分析器并且需要共享输入状态的时候使用它:

```
// 创建 Java 词法分析器
JavaLexer mainLexer = new JavaLexer(input);
// 创建 javadoc 词法分析器； 使用
// java 词法分析器的共享输入状态
JavaDocLexer doclexer =
  new JavaDocLexer(mainLexer。getInputState());
```

### 4.5.3 分析多元记号流

就像一个词法分析器从多个差别很大的输入片段中产生一个独立的流时会遇到很多麻烦，一个语法分析器在处理多记号流的时候也会遇到一些麻烦。同样，一个记号在一个词法分析器中可能是一个关键字，在另一个词法分析器中可能会是一个标识符。将语法分析器根

据不同的输入段分解成子分析器，为每一个输入片段单独处理它们的单词汇表，这样做很有意义，同时也利于文法的重用。

下面的语法分析文法使用主词法分析器的记号词汇表(用 importVocab 指定)，在遇到 JAVADOC_OPEN 的时候，它创建并且调用一个 JavaDoc 分析器来处理后面在注释中的记号流。

```
class JavaParser extends Parser;
options {
    importVocab=Java;
}

input
    :   ( (javadoc)? INT ID SEMI )+
    ;

javadoc
    :   JAVADOC_OPEN
        {
        // 创建一个分析器去处理 javadoc 注释
        JavaDocParser jdocparser =
          new JavaDocParser(getInputState());
        jdocparser。content();  // 用 jdocparser 继续分析
        }
        JAVADOC_CLOSE
    ;
```

你会发现从 2.6.0 版本起，ANTLR 语法分析器也共享记号输入流状态。当创建"子分析器"时，JavaParser 告诉它从同一输入状态对象中获取记号。

JavaDoc 分析器匹配大量的标签:

```
class JavaDocParser extends Parser;
```

```
options {
    importVocab=JavaDoc;
}

content
    :   (   PARAM // includes ID as part of PARAM
        |   EXCEPTION
        |   AUTHOR
        )*
    ;
```

当子分析器的 content 规则结束后，控制权自然地返回给调用它的方法,也就是 Java 分析器中的 javadoc。

### 4.5.4 多记号流超前扫描的效果

如果语法分析器需要超前查看 JavaDOc 注释起始位置后的两个记号，会发生什么呢？换句话说，以主分析器来看，JAVADOC_OPEN 之后的记号是什么呢？当然是记号 JAVADOC_CLOSE!主分析器把任何 JavaDoc 注释看作是一个单一实体,不管这个注释有多复杂;它不会去查看注释记号流内部情况,也不需要这么做——子分析器会处理注释记号流。

子分析器中，content 规则后是什么记号呢？是"End of file"记号。子分析器的分析过程不能确定你的代码中将会调用怎样的方法。但这不是一个问题，因为一般情况会有一个单独的记号标识子分析器的结束。即使因为某种原因 EOF 被载入到分析过程，EOF 也不会出现在记号流中。

### 4.5.5 多词法分析器 vs 调用另一条词法规则

词法分析器的多个状态经常也被用来处理那些非常复杂的单个记号,比如嵌入有转义字符的字符串，输入的"\t"应该被识别为一个字符串。针对这种情况，典型的做法是在第一个引号之后，词法分析器切换到"字符串状态"，在识别完字符串之后再切换回"普通状态"。

所谓的“模式”编程，就是根据不同的模式代码完成不同的事情，这通常是一个不好编程方式。在处理复杂记号的情况下，最好是使用多个规则显式地指定复杂的记号。下面是一个什么时候该用和什么时候不该用多记号流的黄金规则：

**复杂的单个记号应该通过调用另一个(protected)词法规则来匹配，而对来自差别很大的输入片段的记号流来说，应该用多个词法分析器处理相同的输入流并提供给分析器。**

例如，词法分析器中的字符串定义应该只是调用另一个规则来处理转义字符的情况：

```
STRING_LITERAL
    :    '"' (ESC|~('"'|'\\'))* '"'
    ;

protected // 不是一个记号； 仅仅被另一个规则调用
ESC
    :    '\\'
        (    'n'
        |    'r'
        |    't'
        |    'b'
        |    'f'
        |    '"'
        |    '\''
        |    '\\'
        |    ('u')+
             HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT
        …
       )
    ;
```

## 4.6 TokenStreamRewriteEngine 简单的语法制导翻译

在很多情况下，你希望在原代码基础上修改或者增加一段程序或数据文件。ANTLR 2。7。3 引进了一个(只有 Java/C#版本)非常简单但非常强大的类 TokenStream 处理以下问题：

1. 输出语言和输入语言相似
2. 语言元素的相对次序不改变

参见 ANTLR 网站上的 Syntax Directed TokenStream Rewriting。

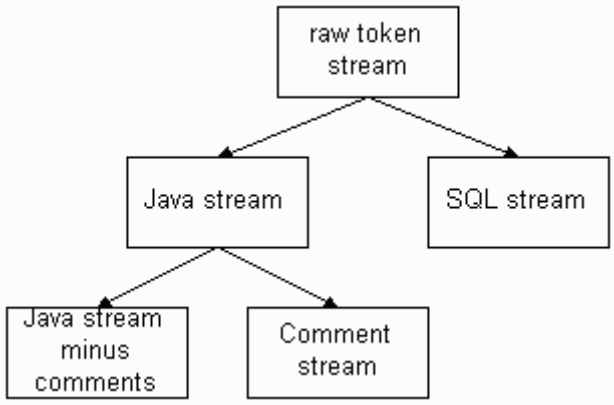
## 4.7 未来

ANTLR 2.6 版本为记号流的使用提供了一个基本框架，一旦我们有经验使用记号流，今后的版本将会更加强大。

当前的"隐藏记号"流过滤器对"忽略但保存空白符"的问题解决得很好，但它在很多情况下不能很好的处理注释。例如，在真正的翻译过程中，为了更好地理解，你想把不同树结点上的注释收集起来(像 DECL 或者 METHOD)，而不是让它们分布在树中。你确实需要一个流分离器缓存一个单独流中的注释，这时你就可以说"**给我在识别这个规则上所用掉的所有注释**"或者 "**给我这两个记号之间的所有注释**。" 这几乎是你在翻译注释时所需要的。

记号流会带来很多便利。大部分人不习惯去思考关于记号流，使得很难想象记号流有什么优点。让思维更开阔一些。怎样处理嵌入语言的输入片段，就像你所能看到的 Java 中嵌入 SQL(输入的每一个部分都可能被分解并通过不同的流)。怎么样分析含有和不含有调试信息的 Java 。class 文件？如果你有一个可以分析不含调试信息的。class 文件分析器，而你想分析含有调试信息的。class 文件，不用去管这个分析器，为你的词法分析器新增处理调试信息的结构。然后用一个过滤器分离调试信息记号到另一个流，这样，对两种类型的。class 文件，原来的分析器就可以正常工作了。

稍后，我会增加一种"看法(perspective)"，这确实是另一种考虑过滤器的方式。想象一下从词法分析器(最初看法)中输出一个原始加工的记号（Token）流。我们可以非常容易地构建一棵树根据最初看法。例如，给出一个嵌有 SQL 的 Java 程序，为了分析或翻译你可能需要输入流的不同思考角度，如下图所示：




你可以把 SQL 流或者去掉注释的 Java 流交给带有查询注释流动作（action）的语法分析器处理。

将来，还会增加分析器的另一个功能，生成记号（Token）流(或文本)作为输出，就像现在建立树一样。这样，多路传递分析变得十分自然和简单，因为语法分析器也变成了流的生产者。一个语法分析器的输出可以是另一个语法分析器的输入。

Version: $Id: //depot/code/org。ANTLR/release/ANTLR-2.7.6/doc/streams.html#1 $

# 第 5 章 记号（token）词汇表

每一种文法都指定了带有规则(子结构)和单词符号（symbol）的语言结构。为了有效地比较,这些符号(symbol)在运行时被转换成整型的"记号(token)类型"。定义从符号(symbol)到记号(token)类型映射的文件对执行 ANTLR 和 ANTLR 生成的分析器来说是基础。这份文档描述了 ANTLR 使用和生成的这类文件，还介绍了用于控制词汇表的选项。

## 5.1 引言

在分析时，一个语法分析器文法通过符号（symbol）来引用在词汇表里的记号（token），该记号符合由词法分析器或其他记号流生成的 Token 对象。分析器比较赋值给每一个符号(symbol)的唯一整数记号类型和储存在记号对象中的记号类型。如果分析器正在查找的记号类型 23，但发现第一个超前扫描的记号的记号类型，`LT(1).getType()`，不是 23，这时分析器抛出 `MismatchedTokenException` 异常。

一个文法可能有一个导入词汇表，通常也会有一个导出词汇表，可以被其他文法引用。导入的词汇表永远不会被修改，表示词汇表的"初始状态"。别混淆 importVocabular 选项。

**下面列出了最常见的问题:**

## 5.1.1 ANTLR 如何决定哪个词法符号是什么记号类型?

每个文法都有一个记号管理器来管理文法的导出词汇表。使用文法的 importVocab 选项，符号管理器可以符号/记号类型对的形式预先被预载。这个选项强制 ANTLR 查找有如下映射关系的文件：

PLUS=44

没有 importVocab 选项的话，文法的记号管理器为空(稍后会看见一个警告)。

文法中任何没有预赋值的记号会根据遇到的顺序依次赋值。例如，在下面的文法中，记号 A 和 B 分别是 4 和 5:

```
class P extends Parser;
a : A B ;
```

词法文件以如下形式命名： NameTokenTypes。txt。

### 5.1.2 为什么记号类型从 4 开始

因为 ANTLR 在分析过程中需要一些特殊的记号类型，用户自定义的记号类型必须在 3 后开始。

### 5.1.3 ANTLR 生成什么样的词汇表相关的文件

ANTLR 为单词 V 生成 VTokenTypes.txt 和 VTokenTypes.java，V 是文法的名字或者是在 exportVocab＝V 选项中指定。文本文件有点像一个简化的记号管理器，表示 ANTLR 需要的回归状态，允许其它文件中的文法查看该文法包括字符串常量在内的文法词汇表。Java 文件是一个包含了记号类型常量定义的接口。ANTLR 生成的分析器实现了其中的一个接口，以获得所需要的记号类型定义。

### 5.1.4 ANTLR 怎样同步在同一文件和不同文件里文法的符号类型映射

一个文法的导出词汇表必须是另一个文法的导入词汇表或者两个文法必须共享一个公共的导入词汇表。

设想 p.g 中的一个语法分析器 P:

```
// yields PTokenTypes.txt
class P extends Parser;
// options {exportVocab=P; } ---> default!
decl : "int" ID ;
```

l.g 中有一个词法分析器 L

```
class L extends Lexer;
options {
  importVocab=P;  // reads PTokenTypes。txt
}
ID : ('a'..'z')+ ;
```

即使 L 使用的是 P 的词汇表中的值,但 ANTLR 还是会生成 LTokenTypes.txt 和 LTokenTypes。

不同文件中的文法必须共享同样的记号类型空间,应该使用 importVocab 选项去预加载相同的词汇表。

如果这些文法在同一文件中，ANTLR 会用同样的方法处理它。然而，你也可以通过设置它们的导出词汇表到同一文件（允许它们都可以使用相同的记号空间）来使这两个文法共享同一个词汇表。例如，P 和 L 在一个文件中，你可以这样做：

```
// yields PTokenTypes.txt
class P extends Parser;
// options {exportVocab=P; } ---> default!
decl : "int" ID ;
class L extends Lexer;
options {
  exportVocab=P;  // shares vocab P
}
ID : ('a'..'z')+ ;
```

如果你没有为 L 指定词汇表，它将会选择共享文件中导出的第一个词汇表；在下面的例子中，它将共享 P 的词汇表：

```
// yields PTokenTypes.txt
class P extends Parser;
decl : "int" ID ;
// shares P's vocab
class L extends Lexer;
ID : ('a'..'z')+ ;
```

记号类型映射文件就像下面这样：

```
P    // exported token vocab name
LITERAL_int="int"=4
ID=5
```

## 5.2 文法继承和词汇表

子文法会继承父文法的规则，动作（action）和选项，但子文法使用什么样词汇表和记号词汇表呢？ANTLR 对子文法的处理就像把父文法的所有非重载规则复制粘贴到子文法中，就像使用 include 一样。因此，子文法的记号集合是父文法记号集合和子文法记号集合的并集。所有的文法都导出到一个词汇表文件，所以子文法导出并使用一个与父文法不同的词汇表文件。子文法通常导入父文法的词汇表，除非你使用 importVocab 选项覆盖它。

继承 P 的文法 Q 会预先根据 P 的词汇表设置它的词汇表，就好像 Q 使用了 importVocab=P 选项一样。例如，下面的文法有 2 个记号符号。

```
class P extends Parser;
a : A Z ;
```

子文法 Q 最初有与父文法相同的词汇表，但增加了一个额外的符号。

```
class Q extends P;
f : B ;
```

上面的情况中，Q 定义了一个额外的符号,B，使得 Q 的词汇表为{A,B,C}。

子文法的词汇表通常是父文法的词汇表的超集（译者注：也即包括父文法的词汇表）。注意重载规则并不影响最初的词汇表。

如果你的子文法需要父文法未使用过的新词法结构，你或许需要让子语法分析器使用一个子词法分析器。使用指定子词法分析器词汇表的 importVocab 选项来覆盖初始的词汇表。例如，假设语法分析器 P 使用词法分析器 PL。没有 importVocab 覆盖，Q 的词汇表将使用 P 的词汇表，进而使用 PL 的词汇表。如果你想让 Q 使用另一个词法分析器的记号类型，比如说 QL，可以如下做：

```
class Q extends P;
options {
  importVocab=QL;
}
f : B ;
```

Q 的词汇表现在和 QL 的词汇表相同或者是 QL 词汇表的超集。

## 5.3 识别器生成顺序

如果所有的文法在一个文件中，你就没必要担心 ANTLR 最先处理哪一个文法文件，不过你仍需要担心 ANTLR 处理文件中文法的顺序。如果你尝试去导入一个由文件中后面一个文法导出的词汇表，ANTLR 将提示它不能加载这个文件。下面的文法文件会造成 ANTLR 出错：

```
class P extends Parser;
options {
importVocab=L;
}

a : "int" ID;

class L extends Lexer;
ID : 'a';
```

ANTLR 将提示不能找到 LTokenTypes.txt，因为在文法文件中还没有看到文法 L。另外，如果 LTokenTypes.txt 存在(文法文件中还没有 P 文法的时候 ANTLR 运行生成的？)，ANTLR 将为 P 加载这个文件，然后在处理 L 文法的时候覆盖它。ANTLR 必须假设是要加载的词汇表由另一个文件生成，因为它不知道接下来会是哪个文法在同一文件中。

通常来说，如果你想让文法 B 使用文法 A 的记号类型(不管什么文法类型)，你必须首先在文法上 A 运行 ANTLR。例如，一个使用了分析器的文法词汇表的树文法应该在 ANTLR 生成了分析器之后再运行。

例如，当你想让一个词法分析器和一个语法分析器共享同一个词汇表空间的时候，你要做的就是去把它们放到同一个文件中，设置它们的导出词汇表指向同一个空间。如果它们在的不同的文件中，把语法分析器的导入词汇表选项设置为词法分析器的导出词汇表，除非语法分析器产生了大量的字面常量。这时，交换一下导入/导出的关系让词法分析器使用语法分析器的导出词汇表。

## 5.4 词汇表的一些使用技巧

如果你的文法在不同的文件中，你仍想让它们共享全部或部分记号空间，该怎么办呢？有 2 种解决方法：(1) 让文法导入相同的词汇表 (2) 让文法继承同一父文法，该父文法含有共享的记号空间。

第一种方法适合于下面的情况，比如有 2 个词法分析器和 2 个语法分析器，必须分析截然不同的输入的部分。ANTLR 2.6.0 发行版 examples/java/multiLexer 中的例子就属于这种情况。javadoc 注释和 Java 代码分别由不同的词法分析器和语法分析器分析。Javadoc 的词法分析器有必要识别"*/"中止注释的词法结构，但它一般通过 Java 的语法分析器使用打开/关闭的记号引用来嵌套加载 javadoc 语法分析器：

```
javadoc
    : JAVADOC_OPEN
      {
      DemoJavaDocParser jdocparser =
      new DemoJavaDocParser(getInputState());
      jdocparser.content();
```

```
    }
    JAVADOC_CLOSE
  ;
```

问题在于：javadoc 的词法分析器定义了 JAVADOC_CLOSE，即也定义了它的记号类型。不幸的是 Java 的语法分析器的词汇表是基于 Java 的词法分析器而不是 javadoc 的词法分析器。要让 javadoc 的词法分析器和 java 的词法分析器都可以看到 JAVADOC_CLOSE（并且有同样的记号类型），2 个词法分析器都要导入含有这种记号类型定义的词汇表。这里有 DemoJavaLexer 和 DemoJavaDocLexer 的头部：

```
class DemoJavaLexer extends Lexer;
options {
  importVocab = Common;
}
...
class DemoJavaDocLexer extends Lexer;
options {
  importVocab = Common;
}
...
```

CommonTokenTypes。txt 包含:

```
Common // name of the vocab
JAVADOC_CLOSE=4
```

共享词汇表的第二种方法适合于下面的情况，比如有 1 个语法分析器和 3 个不同的词法分析器(比如说为不同风格的 C)。为了空间效率，你只想使用一个语法分析器，这个语法分析器必须可以访问 3 个不同词法分析器的所有词汇表，去掉文法上不需要的结构(可能使用语义断言)。给定 CLexer，GCCLexer 和 MSCLexer，使 CLexer 作为父文法并定义所有记号的集合。例如，如果 MSCLexer 需要"_int32"，可以预留一个对 CLexer 中所有词法分析器都可见的记号类型：

```
tokens {
```

```
    INT32;
  }
```

在 MSCLexer 中，你可以给它赋与一个字符。

```
  tokens {
    INT32="_int32"
  }
```

通过这种方法，不同的词法分析器可以共享同一记号空间，允许你用一个语法分析器识别多种不同风格 C 的输入。

Version: $Id: //depot/code/org。ANTLR/release/ANTLR-2.7.6/doc/vocab。html#1 $

# 第 6 章 错误处理及恢复

所有的句法和语义错误都会引起解析器异常的抛出。特别是，当用来匹配解析器基类(或者其它类）中记号的方法出现错误时，会抛出 MismatchedTokenException 异常。如果预测分析在解析器或者 Lexer 之间没有更好地选择，会抛出 NoViableAltException 异常。Lexer 基类中用来匹配字符串的方法在出现错误时会抛出类似的异常。

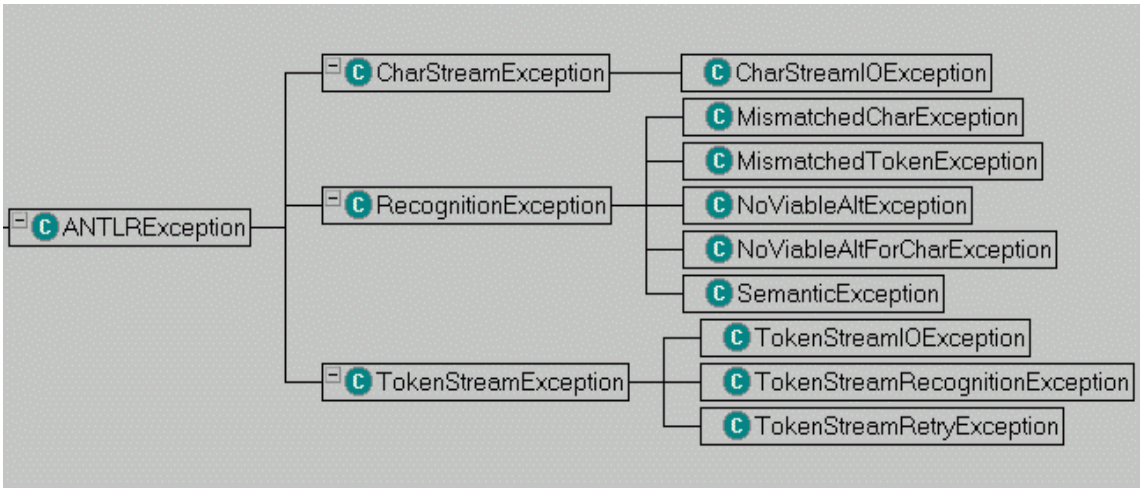
ANLTR 可以产生默认的错误处理代码，当然你也可以指定自己的异常处理代码。上述任意一种情况，ANLTR 都会生成 try/catch 语句块（当然这也需要编程语言的支持）。这样的 try 语句块会在生成代码的重要文法元素的周围，如规则、选择、记号参考（reference）、规则参考等文法元素。如果没有指定相应的异常处理（默认的或其它的），异常将会被抛出给解析器外的上一级调用程序。

ANLTR 默认的异常处理能够很好地处理大部分异常，但是，如果你编写自定义的异常处理代码，你将能更多地控制错误报告和同步异常。

注意：PCCTS 1。33 的'@'异常规范并不适用于 ANTLR。

## 6.1、ANLTR 的异常体系结构

基于 ANTLR 生成器的解析器通过抛出异常来表明出现了识别错误或其他流问题。所有的异常都是继承于 ANTLRException。下图展示了 ANLTR 的异常体系结构：




| 异常 | 描述 |
|---|---|
| ANTLRException | 所有异常处理类的基类。如果你想自定义异常处理类，你可以直接从该派生，除非自定义的异常处理类与下面已定义的异常处理类很相似 。 |
| CharStream-Exception | 字符输入流中发生了一些错误。大多数情况下是由于 IO 故障引起，但你也可以为来自对话框或其它方式的输入定义该异常。 |
| CharStreamIO-Exception | 字符输入流中发生了 IO 异常（例如：方法 CharBuffer。fill()会抛出该异常）。如果方法 nextToken()捕获到该异常，它将会把该异常转换为 TokenStreamIOException 异常。 |
| Recognition-Exception | 输入流中一个常见的识别问题。在 main 函数中或其它调用解析器（parser）、lexer、树解析器（treeparser）的方法中，以该来“捕获一切”异常。所有的解析规则均能抛出该异常。 |
| MismatchedChar-Exception | 当方法 CharScanner。match()在输入流中查找一个字符，但搜索到的却是另外一个字符，即查找不到匹配字符时，会抛出该异常。 |
| MismatchedToken Exception | 当方法 Parser。match()在输入流中查找一个符号，但搜索到的却是另外一个符号，即查找不到匹配符号时，会抛出该异常。 |
| NoViableAlt-Exception | 解析器发现一个未定义的符号，也就是说，解析器发现了一个符号，但该符号并不在当前决策中开始任何一个选择。 |
| NoViableAltFor-CharException | lexer 发现一个未定义的字符，也就是说，lexer 发现了一个字符，但该字符并不在当前决策中开始任何一个选择。 |
| Semantic- | 用来表明语法结构有效，但在输入流中出现了无语法意义或其它错误的 |

| | |
|---|---|
| Exception | 输入。当验证语义谓词失败时，该异常会被自动抛出。例如：<br>a : A {false}? B ;<br>ANTLR 会产生如下代码：<br>match(A);<br>if (!(false)) throw new<br>SemanticException("false");<br>match(B);<br>在解析过程中，如果其中一个活动发现输入是无效的，你也可以抛出该异常。 |
| TokenStream-Exception | 表示在产生符号流的时出现了错误。 |
| TokenStreamIO-Exception | 隐含着在 TokenStreamException 异常中同时存在 IOException 异常。 |
| TokenStream-Recognition-Exception | TokenStreamException 异常中隐含着 RecognitionException 异常，所以你可以将该异常传递给一个流。 |
| TokenStream-RetryException | 信号终止了当前符号的识别，然后尝试再次识别。TokenStreamSelector。retry() 方法使用该异常来强制流的 nextToken()方法重新读取并重新识别。具体可以参考 examples / java / includeFile 目录。<br>这是一个处理嵌套的包含文件等的很好方式，或者用来通过尝试多种语法查找来对数据进行匹配。类似当不知道何时会出现何种数据时，你可以在一个套接字上对多种输入类型进行侦听。 |

典型的 main 函数或解析调用者会在调用附近产生如下类似的 try-catch 语句块：

```
try {
        ...
    }
    catch(TokenStreamException e) {
```

```
  System.err.println("problem with stream: "+e);
}
catch(RecognitionException re) {
  System.err.println("bad input: "+re);
}
```

Lexer 规则会抛出如下异常：RecognitionException，CharStreamException 及 TokenStreamException。解析规则会抛出如下异常：RecognitionException 和 TokenStreamException。

## 6.2 借助文法来修改默认的错误消息

在 lexer 中使用的符号名或定义对识别器或翻译器的用户来说几乎毫无意义。例如：

```
T.java:1:9: expecting ID, found ';'
```

经过解析器解析后会生成如下代码：

```
T.java:1:9: expecting an identifier, found ';'
```

ANTLR 提供了一种简单的方法来指定一个字符串以代替符号名。在符号“ID”的定义中，使用如下文法：

```
ID
options {
  paraphrase = "an identifier";
}
  : ('a'..'z'|'A'..'Z'|'_')
    ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*
  ;
```

请注意，该文法被加入到记录符号类型的文法文件中（ANTLR 的文法文件）。换而言之，也就是任何使用该词汇表的文法分析同样会使用该文法。

## 6.3 解析异常处理

ANTLR 生成递归下降的文法识别器。既然递归下降的文法识别器是通过递归调用规则匹

配方法来运行，那么将会产生一个存放递归下降调用的方法内容的调用栈。文法解析规则的异常处理就像 C++或 Java 中的异常处理一样。也就是，当异常被抛出时，线程的正常执行会被停止，然后调用栈中的函数依次退出，直至捕获该异常的函数。一旦异常被捕获，程序将会在异常被捕获的地方继续执行。

ANTLR 中，解析异常会在以下情况下被抛出：

1) 语法错误；

2) 语法谓词的有效性验证失败；

3) 用户抛出异常。

在上述所有情况在，调用栈中递归下降的函数会依次退出，直至遇到捕获该异常的类或其父类（非面向对象语言中，异常处理的层次结构并不是以类的层次来实现的）。异常处理类可以由两种方法生成。第一种方法是，如果你不做任何处理，ANTLR 会为每一个解析规则产生默认的异常处理方法。默认的异常处理方法会报告一个错误，并同步到后续的规则集，然后从该规则返回。第二种方法是，你可以根据稍后介绍的各种方法来自定义异常处理方法。

一旦你为一个规则自定义了异常处理方法，就不会生成该规则的默认异常处理方法。另外，你也可以通过 per-grammar 或 per-rule 选项控制默认异常方法的生成。

## 6.4 指定解析异常处理方法

你可以为一个规则、一个选择、或一个标签元素指定异常处理方法。指定异常处理方法的一般格式是：

```
exception [label]
catch [exceptionType exceptionVariable]
  { action }
catch ...
catch ...
```

这里的标签仅仅用于为标签元素指定异常处理方法。exceptionType 是欲捕获的异常，exceptionVariable 是被捕获异常的变量名，进而方法可以处理指定的异常。下面是一个为规则、选择、标签元素自定义异常处理方法的例子：

```
rule:   a:A B C
    |   D E
```

```
        exception // for alternate
          catch [RecognitionException ex] {
            reportError(ex。toString());
        };
    exception // for rule
    catch [RecognitionException ex] {
       reportError(ex。toString());
    }
    exception[a] // for a:A
    catch [RecognitionException ex] {
       reportError(ex。toString());
    }
```

注意选择和标签元素的异常并不会导致规则解析的退出。匹配和控制流会继续执行，似乎异常并没有发生。因此，你必须注意，不要使用任何在异常发生时已经因为匹配而被设置的变量。

## 6.5 Lexer 中的默认异常处理

通常情况下，你希望 Lexer 不断的尝试在词法错误的情况下生成一个有效的符号。那样的话，解析器就没有必要处理词法错误以及生成一个其它的符号。有时候，你可能希望异常被抛出到 Lexer 之外，大部分情况是你希望在出现词法错误时中止整个解析过程。为了使 ANTLR 产生将 RecognitionException 作为 TokenStreamException 传递给解析器的 lexer，可以使用 defaultErrorHandler=false 的文法选项。注意 IO 异常被作为 TokenStreamIOException 传回给 lexer，而忽略 defaultErrorHandler=false 的文法选项。

下面的例程使用不存在的语义异常（RecognitionException 的子类）来阐述 lexer 的异常抛出（blasting out of the lexer）：

```
class P extends Parser;
{
  public static void main(String[] args) {
          L lexer = new L(System。in);
```

```
        P parser = new P(lexer);
        try {
                parser.start();
        }
        catch (Exception e) {
                System.err.println(e);
        }
  }
}

start : "int" ID (COMMA ID)* SEMI ;

class L extends Lexer;
options {
        defaultErrorHandler=false;
}
{int x=1; }
ID  : ('a'..'z')+ ;
SEMI: '; '
      {if ( expr )
       throw new
          SemanticException("test",
                            getFilename(),
                            getLine()); } ;

COMMA:',' ;
WS  : (' '|'\n'{newline(); })+
      {$setType(Token.SKIP); }
    ;
```

当输入“int b;”时，你会得到如下输出：

ANTLR.TokenStreamRecognitionException: test

Version: $Id: //depot/code/org.ANTLR/release/ANTLR-2.7.6/doc/err。html#1 $

# 第 7 章 Java Runtime Model

# 第 8 章 C++ Runtime Model

# 第 9 章 C# Runtime Model

# 第 10 章 Python Runtime Model

# 第 11 章 ANTLR 树构建

ANTLR 通过语法注释指出哪些记号（tokens）为子树,哪些记号（tokens）为叶子和哪些是在树结构中应该忽略，从而帮助你生成中间形式的树,或者说是抽象语法树(ASTs)。就像 PCCTS1。33,你就可以使用 tree grammar action 来操作语法树。

程序员经常遇到这样的情况,或者已经拥有了树定义或者需要一个特殊的物理结构来阻止 ANTLR 定义抽象语法树（AST）节点的实现。ANTLR 只定义了一个接口来描述最基本的行为。你在树的实现中必须实现这个接口以便让 ANTLR 知道如何来处理你的树。另外,你必须告诉解析器树节点的名字或者提供一个树“工厂”，这样 ANTLR 才能知道怎么使用正确的类型来创建节点(而不是在所有新的 AST() 表达式中进行硬编码)。ANTLR 能构建和遍历任何满足的 AST 接口的树。少量通用的树定义已经提供。不幸的是,ANTLR 不能解析 xml 的 dom 树,因为我们的方法名字与之相冲突(例如,getfirstchild())；ANTLR 首先在这里非常不尽人意,

唉!

## 11.1 注释

在此文档以及其它的文档中,树结构由类似于 LISP(LISP 是一种比较简单的动态语言)的符号表示,比如:

#(A B C)

是一棵以 A 为根节点,B 和 C 是儿子的树。这个结构可以嵌套表示任意结构的树,比如:

#(A B #(C D E))

是一棵以 A 为根节点,B 是第一个儿子,和整个子树作为第二个儿子的树,而这颗子树,是以 C 为根节点和 D,E 为儿子。

## 11.2 控制 AST 构建

在 ANTLR 解析器中的 AST 构建或者在树解析器中的 AST 转换的使用与不使用是通过 buildAST 选项来控制的。

从 AST 构建和遍历的角度来看，ANTLR 认为所有的树节点都是一样的(也就是它们表现的是没有区别的)。通过树“工厂”或规则，然而你也可以指示 ANTLR 创建出不同类型的节点。见下一节关于异构树。

## 11.3 构建 AST 的语法注释

### 11.3.1 叶节点

因为内附规则，ANTLR 把所有在产生树中没有后辍或并列的 token 作为叶子结点。如果在语法中完全没有后辍，解析器就会构建一个 tokens 连接表（退化的 AST），树解析器就会拷贝输入的 AST。

### 11.3.2 根节点

所有以“^”操作符为后辍的 token 都被认为是根节点。为 token 建立一个节点，并把它作为一个原来那部分树的一个根节点。

```
a : A B^ C^ ;
```

产生的树为 #(C #(B A))。

首先 A 被匹配当作一个独立的的儿子节点，跟着 B 被匹配作为当前树 A 的父亲节点。最后 C 被匹配并作为当前树的父亲节点，并作为 B 节点的父节点。注意，没有任何操作符的同样的规则产生的是平行的树 A B C。

### 11.3.3 关闭标准树的构建

使用“!”作为 token 的后辍以避免将与该 token 关联的节点加到产生树中(该 token 的 AST 节点依然会被构建，可能会在 action 中引用，只是不会自动被添加到产生树中)。使用“!”作为后辍的规则指明调用该规则所构建的树不会被链接到当前规则构建的树结构中。

使用“!”作为一个规则定义的后辍指明该规则对应的树是失效的。该规则引用的规则和 tokens 依然会创建 ASTs,但是它们不会被连接到产生树中。下列规则并没有自动创建树。必须使用 actions 来设定返回的 AST 值，如：

```
begin!
    :   INT PLUS i:INT
        { #begin = #(PLUS INT i);  }
    ;
```

好的做法是以可选的“!”作为前辍来为另一种选择关闭树的构造。这种做法非常有用，例如，如果你有大量的选择但你只想选用一个来手动构建树：

```
stat:
        ID EQUALS^ expr    // 自动构建
  ... some alternatives ...
    |!  RETURN expr
        {#stat = #([IMAGINARY_TOKEN_TYPE] expr); }
   ... more alternatives ...
    ;
```

### 11.3.4 树节点构建

当自动构建树的选项为 off 的时候(但是使用 buildAST)，你必须构建你自己的树节点和在内含的 actions 中联结它们加入到树结构。有几种方法在一个 action 中生成树节点：

1. 使用 T(arg),T 是你的树节点类型，arg 是单一的 token 类型，一个 token 类型与 token 原文和一个 token 中的其中一个。
2. 使用 ASTFactory。create(arg)，T 是你的树节点类型，arg 是单一的 token 类型，一个 token 类型与 token 原文和一个 token 中的其中一个。使用工厂比直接生成一个新节点普遍，因为节点类型的定义由工厂来控制，可以容易地更改整个语法。
3. 使用简化符号#[TYPE]或#[TYPE,”text”]或 [TYPE,”text”,ASTClassNameToConstruct]。简化符号导致对 ASTFactory。create() 填入特定参数的调用。
4. 使用简明符号#id，id 是被匹配的 token，记号和规则引用其一。

从节点集合里构建树，你可以根据你自己来设定首子节点和下一个兄弟节点,或者调用工厂执行方法,或者使用下面说到的#(…)符号。

### 11.3.5 AST Action 换化

在一般解析器和树解析器中，当 buildAST 设为 true,为使在 actions 中创建 ASTs 为我简化 ANTLR 会转化部份的用户 action。特别地，以下以”#”为开始的构建将会被转化：

#label

Token 引用或规则引用与 AST 关联可以通过#label 来访问。被翻译为一个含有由 token 或者规则返回的 AST 值创建的 AST 节点。

#rulb

当一个规则是一个内部规则的名字的时候，ANTLR 会把它翻译为含有 AST 结果的规则的变量。由此充许你设定规则的返回 AST 结果或者在一个 action 里审查它。当 AST 生成为开启状态或者支持规则简化或其它。例如：

```
r! : a:A {#r = #a; }
```

当与普通树构建组合的时候设置返回的树是非常有用的,因为你可以让 ANTLR 做所有的工作(创建树和添加虚的根节点)如：

```
decl : ( TYPE ID )+
```

```
  { #decl = #([DECL,"decl"], #decl); }
;
```

ANTLR 充许你在任何地方给把另外一个 rule 赋给#rule。ANTLR 确保对在 action 内部的规则的引用和赋值在解析器内部的 AST 的构建中达到稳定的状态。在你给#rule 赋值之后，当 ANTLR通过#tule创建完树根节点后解析器自动构建的AST变量的状态将会被设置。例如：任何在 action 执行完之后被添加的子节点将会被添加成#rule 的子节点。

#label_in

在树解析中，由标签 token 或者规则引用的输入 AST 可以通过形如#label_in 来访问。相应的解析为一个由该规或 token 归纳出的包含 AST 型的输入树节点的变量。输入变量比较少用。你总是会使用#label 来替代#label_in。

#id

ANTLR 支持把没有标注的 token 引用作为简化符号来进行转化,只要该 token 在选定范围中唯一。在这种情况下,使用一个没有标注的 token 引用如同使用一个标注符号,例如:

```
r! : A { #r = #A; }
```

等同于:

```
r! : a:A { #r = #a; }
```

#id_in 的使用与#label_in 的使用一样

#[TOKEN_TYPE] or #[TOKEN_TYPE,"text"] or #[TYPE,"text",ASTClassNameToConstruct] 是 AST 节点构造简化。相应的作用是对 ASTFactory。create()方法的调用。例如,#[T]转化为:ASTFactory.create(T)

#(root,c1,...,cn)

是 AST 树的构造的简化。ANTLR 会自动查找到逗号来分隔树的参数。在方法中的逗号被称为树的元素,它会被适当地处理。如:元素 foo(#a,34)是正确的,它不会与在同一棵树中的其它的树元素的分隔符相冲突。这个树结构会被翻译成”创建树”调用。该“创建树”调用因为需要构造与 java 相似的变量参数所以比较复杂,由此它的型式会像如下：

ASTFactory。make(root,c1,...,cn);

除了对 #(…) 作为一个整体的转化外,根节点与每一个子节点也会被转化。在一个#(…)结构的上下文里,你可以使用:

id 或者 label as 的简化符号替代 #id 或 #label。

[...] 的简化符号替代 #[...]。

(...) 的简化符号替代 #(...)。

目标代码生成器会使用特殊的解析动作（action）的语法执行这个转化,同时告诉代码生成器来为每一个转化项创建适当的替代。这个语法分析会使用一些关于符号名字的限制(思考 C/C++预处理向导)

## 11.4 执行解析创建树

假设你已经在你的程序中定义了一个语法 L 和一个解析器 P,你可以对你的系统输入顺序地执行它们像下面一样:

```
L lexer = new L(System。in);
P parser = new P(lexer);
parser。setASTNodeType("MyAST");
parser。startRule();
```

如果你已经在你的转化程序中设定 buildAST=true,这样它会创建一个 AST 节构,这个结构可以通过 parser.getAST()来访问到。如果你已经定义了一个树解析器叫作 T,你可以用以下方式执行它:

```
T walker = new T();
walker。startRule(parser.getAST());  //遍历树
```

另外,如果你在你的树解析器中已经设定 buildAST=true 来开启转化模式,这样你可以通过树遍历器来访问到 AST。

```
AST results = walker.getAST();
DumpASTVisitor visitor = new DumpASTVisitor();
visitor.visit(results);
```

DumpASTVistor 是一个预定义的 ASTVistitor 实现,用于把树简单地打印到标准终端上。你也可以用 String s = parser.getAST().toStringList(); 这种获取一个列表型式来打印树。

## 11.5 AST 工厂

ANTLR 使用工厂模式来创建与连接 AST 节点。这个主要用来巧妙地把它从解析器中分隔

开树结构,但是留下了用于操作解析器与树节点结构的回调接口。通过继承可以改变用于创建的方法。

如果你只关注于在运行时指定 AST 节点的类型,在解析器或工厂中使用

setASTNodeType(String className)

方法。默认地,组成树的节点都是 ANTLR 型的。CommonAST。(你必须使用完全符合规范的类名字)。你也可以为每个 token 类型定义一个不同的类名字生成不同类型的树。

```
/** 定义一个 "重写方法"给为特殊的 token 而创建的 Java AST 对象。
 *  它的是为了方便于你定义一个动态类型而创建的。
 *  ANTLR 设定被从 tokens{...}部份自动映射的 token 类型,
 *  但是你要以改变被这个方法所映射的东西。
 *  ANTLR 尽力地为解析产生器使确定节点的类型静态化,
 *  但是它不能处理动态的类型,如#[LT(1)]
 *  在这种情况下, 它依赖于映射本身。
 *  请注意 tokens{...}
 *  部份与你通过此方法设置的部份。
 *  确保它们是一样的。
 *
 *  移除该映射,请设类名字为空。
 *
 *  @since 2.7.2
 */
public void setTokenTypeASTNodeType(int tokenType, String className)
        throws IllegalArgumentException;
```

ASTFactory 主要有以下常用的方法:

```
/** 用相同的 java AST 对像来复制一个单独的节点。
 *  忽略 tokenType->类映射,因为你知道节点的类型
 *  ,t.getClass(),和执行 dup 操作。
 *
```

```
 *  clone() 没有被使用,因为我们目的是所有的 AST 创建都通过工厂来实现,
 *  这样,创建工作可以被记录。如果 t 为 null,返回 null。
 */
public AST dup(AST t);

/** 复制树,复制的内容包括根节点的同级节点
 */
public AST dupList(AST t);
/**复制树, 假假这个是
 * 树的根节点—复制该节点和
 * 它下级的东西; 忽略它的同级节点。
 */
public AST dupTree(AST t);
```

## 11.6 异类 ASTs

在 AST 中的每个节点必须把这类节点的信息进行编码; 例如,是否是一个 ADD 操作或者一个 INT 的叶子节点?有两种方法来编码这些信息:使用一个 java(或 C++等)类型。也就是说,你是否有一个单独的类型和很多的 token 类型或者是没有 token 类型但是有很多类类型?当缺少比较好的条件的时候,我(Terence)一直使用单独的类型相似树和许多类型相异的树来调用 ASTs。

对于不同的节点使用不同的类的原因是要处理执行一堆人工编码的树的遍历或者你的节点储存着没有共同特性的各种数据。我使用的例子是一个表达式树,该树的每个节点重写了 value()方法,所以 root。value()是对输入表达式的运算结果。从对创建树的预处理与使用一个生成树解析器遍历，最好的方法是假设每一个节点都是相同的 AST 节点。这样，对于异类与同类的 AST 的相异处被合适地处理。

ANTLR 同时支持两种树节点！如果你只设置了”buildAST=true”选项，你得到了一棵同构树。随后，如果你只想使用了物理结构不一样的类类型来处理节点，你只须定义创建树的语法。接着，你可以拥有世界上最好的东西—自动构建树，但是你可以为变量节点提供不同的方法与存储相异的数据。注意，树的结构没有被影响，只是节点的类型被改变了。

ANTLR 为创建过程的需要，定义了一个在某范围内有效的运算式，用于为特殊的 AST 节点来确定类型。默认的类型为 CommonAST。你可以使用解析器的接口覆盖该属性，使用调用：myParser。setASTNodeType("com。acme。MyAST")；

这里，你必须使用符合规范的类名字。

在这个语法中，你可以通过设定根据特殊的输入字串生成的节点的类型来重定义默认的类型。在 tokens 部份使用选项<AST=typename>：

```
tokens {
    PLUS<AST=PLUSNode>;
    ……
}
```

更进一步，你可以通过一个在你的解析语法中特殊的 token 记号注释来重写类型：

```
anInt : INT<AST=INTNode>
```

涉汲到的这个重写对于 tokens 如 ID(该 ID 你在一个上下文环境中可能想转化为一个 TYPENAME 节点，在另外一个环境中转化为 VARREF)非常有用的。ANTLR 使用 AST 工厂来创建所有的 AST 节点，即使知道它的类型。换一种说法，ANTLR 应用如下的方式来生成结点：

```
ANode tmp1_AST = (ANode)astFactory.create(LT(1),"ANode");
```

从

```
a : A<AST=ANode> ;
```

## 11.6.1 一棵表达式树例子

这个例子包括了一个用于构建表达式 ASTs 的解析器，一个比较常用的语法分析器和一些 AST 节点类的定义。

让从描述 AST 结构与节点类型来开始我们的工作吧。表达式中有加，乘操作，有整数。操作将作为子树的根节点(非叶子节点)，整数将被作为叶子节点。例如：输入 3+4*5+21 生成一棵树结构如下：

```
(  +(  +  3(  *  4  5))  21)
```

或者:

```
  +
  |
```

```
  +--21
  |
  3--*
     |
     4--5
```

所有的 AST 节点都是 CalcAST 的子类，CalcAST 属于 BaseAST，同样提供 value()方法。Value()方法计算从属于该节点的值。自然地，对于整型节点，value()方法只是简单地返回该节点存储的值。下面就是 CalcAST:

```
public abstract class CalcAST
    extends ANTLR.BaseAST
{
    public abstract int value();
}
```

AST 操作节点必须联合它两个子树计算出来的结果。它必须对于它下级的节点执行一个深度优先的树遍历。为使操作有趣点和更显而易见，操作节点定义用 left()和 right()替代，这样使它们与同级的子树代表符号显示得更不一样。因此，这些表达式树可以被当作同构的同级子树和异构的表达式树。

```
public abstract class BinaryOperatorAST extends
    CalcAST
{
    /** 使其看起来像异构树 */
    public CalcAST left() {
        return (CalcAST)getFirstChild();
    }

    public CalcAST right() {
        CalcAST t = left();
        if ( t==null ) return null;
        return (CalcAST)t.getNextSibling();
    }
```

```
}
```

在该树中简单的节点如下:

```
import ANTLR.BaseAST;
import ANTLR.Token;
import ANTLR.collections。AST;
import java.io.*;

/** 替代 INT 的简单节点 */
public class INTNode extends CalcAST {
    int v=0;

    public INTNode(Token tok) {
        v = Integer.parseInt(tok。getText());
    }

    /** 计算子树的值，这个是异构的部份
     */
    public int value() {
        return v;
    }

    public String toString() {
        return " "+v;
    }

    // 实现 BaseAST 的虚方法
    public void initialize(int t, String txt) {
    }
    public void initialize(AST t) {
    }
```

```
    public void initialize(Token tok) {
    }
}
```

操作衍生于 BinaryOperatorAST 和按照 left()与 right()定义了 value()。例如, 这里是 PLUSNode:

```
import ANTLR.BaseAST;
import ANTLR.Token;
import ANTLR.collections。AST;
import java.io.*;

/** 替代 PLUS 操作的简单节点 */
public class PLUSNode extends BinaryOperatorAST {
    public PLUSNode(Token tok) {
    }

    /** 计算子树的值;
     *这里是相异的部份 :)
     */
    public int value() {
        return left().value() + right().value();
    }

    public String toString() {
        return " +";
    }

    // 实现 BaseAST 的虚方法
    public void initialize(int t, String txt) {
    }
    public void initialize(AST t) {
```

```
    }
    public void initialize(Token tok) {
    }
}
```

解析器非常简单,除了你需要添加配置选项来告诉 ANTLR 你想根据匹配到的输入流创建哪些类型。Tokens 部份列出了操作和与之追加到它们定义的选项元素。这告诉 ANTLR 为所有的在输入流中发现的 PLUS tokens 创建 PLUSNode 对象，例如。示范中，INT 没有被包含在 tokens 部份—详细而精确的 token 引用被选择项标注下标来指定该创建于 INT 节点应该为 INTNode 型(当然，如果只有一个对 INT 的引用，效果是一样的)。

```
class CalcParser extends Parser;
options {
    buildAST = true;  // uses CommonAST by default
}

// 定义一堆特别的 AST 节点用于以下构建
// 可以覆盖目前在下面语法中的对 tokens 的引用
tokens {
    PLUS<AST=PLUSNode>;
    STAR<AST=MULTNode>;
}

expr:   mexpr (PLUS^ mexpr)* SEMI!
    ;

mexpr
    :   atom (STAR^ atom)*
    ;

// 示范 tokens 中的选项
atom:   INT<AST=INTNode>
```

;

像往常一样执行解析器。 通过调用根的 value()方法，可以完全地计算 AST 的结果。

```
import java.io.*;
import ANTLR.CommonAST;
import ANTLR.collections.AST;

class Main {
    public static void main(String[] args) {
        try {
            CalcLexer lexer =
                new CalcLexer(
                  new DataInputStream(System。in)
                );
            CalcParser parser =
                new CalcParser(lexer);
            // 解析表达式
            parser.expr();
            CalcAST t = (CalcAST)parser。getAST();

            System.out.println(t.toStringTree());

            // 计算值与返回
            int r = t.value();
            System.out.println("value is "+r);
        } catch(Exception e) {
            System.err.println("exception: "+e);
            e.printStackTrace();
        }
```

```
    }
}
```

For completeness, here is the lexer:

```
class CalcLexer extends Lexer;

WS  :   (' '
    |   '\t'
    |   '\n'
    |   '\r')
        { $setType(Token。SKIP);  }
    ;

LPAREN: '(' ;

RPAREN: ')' ;

STAR:   '*' ;

PLUS:   '+' ;

SEMI:   ';' ;

protected
DIGIT
    :   '0'..'9' ;

INT :   (DIGIT)+ ;
```

### 11.6.2 使用语法描述异构树

这个方法与同构的树构建有什么区别呢？最大的区别是，你需要一个树语法来描述表达式树和计算结果值。但是，它的优点是它的“可执行文件”和不用你手动去为树解析编码(value()方法)。如果你使用同构树，以下的是你除解析器/语法分析器外需要的计算表达式的值的代码： [这些代码来自 examples/java/calc directory。]

```
class CalcTreeWalker extends TreeParser;


expr returns [float r]
{
    float a,b;
    r=0;
}
    :   #(PLUS a=expr b=expr)    {r = a+b; }
    |   #(STAR a=expr b=expr)    {r = a*b; }
    |   i:INT
        {r = (float)
         Integer.parseInt(i.getText()); }
    ;
```

因为 Terence 希望你用语法树，即使是构建异构树 ASTs 的时候()(避免手动编写实现深度优先的搜索方法)，在各种 AST 节点类中，实现以下的方法:

```
    /** 获取这个节点的 token 信息 */
    public String getText();
    /** 获取这个节点的 token 类型 */
    public int getType();
```

那就是怎样使用异构树结合语法树。 注意，从你解析器中导入的的 token 类型必须匹配 PLUS 和 STAR token 类型。例如,确保 PLUSNode.getType()返回 CalcParserTokenTypes。PLUS。 通过 ANTLR 提供的接口而创建，如以下方式

```
public interface CalcParserTokenTypes {
```

```
        ...
        int PLUS = 4;
        int STAR = 5;
        ......
}
```

## 11.7 AST（XML）序列化

[Oliver Zeigermann olli@zeigermann。de 提供最初的序列化实现。他的 XTAL XML 转化最早被检出； 特别地为读 XML-serialized ASTs 的支持。]

由于各种各样的理由,你可能想存储一个 AST 或者把它传给另外一个程序或电脑。类 ANTLR。BaseAST 是可序列化的，它使用 Java 代码生成器,这样你就可以把 ASTs 用标准的 java 写方式写到硬盘上。 你也可以使用以下 BaseAST 的方法把 ASTs 写成 XML 的型式:

- public void xmlSerialize(Writer out)
- public void xmlSerializeNode(Writer out)
- public void xmlSerializeRootOpen(Writer out)
- public void xmlSerializeRootClose(Writer out)

所有的方法抛出 IOException。

你可以重写 xmlSerializeNode 这样你就可以改变节点输出的方式。默认地，序列化动作(action)使用类类型名字作为目标名字和拥有属性文本与类型来存储本文和该 token 节点的类型。

执行简单的异构树的例子的输出 examples/java/heteroAST, 生成:

```
( + ( + 3 ( * 4 5 ) ) 21 )
<PLUS><PLUS><int>3</int><MULT>
<int>4</int><int>5</int>
</MULT></PLUS><int>21</int></PLUS>
value is 44
```

LISP 程序风格的树展示着结构与内容。各种异构的节点重写着打开与关闭标签，修改

了子叶节点的序列化，使用<int>值</int>代替了单独节点的属性标签。

下面是生成 XML 的代码:

```
Writer w = new OutputStreamWriter(System.out);
t.xmlSerialize(w);
w.write("\n");
w.flush();
```

## 11.8 AST 枚举

AST 的 findAll 和 findAllPartial 方法返回了你可以遍历的树结点的枚举。接口：

ANTLR.collections.ASTEnumeration

和

class ANTLR.Collections.impl.ASTEnumerator

实现该功能。下面是例子：

```
// 打印所有的在树 t 中的 a-subtree-of-interest 实例。
ASTEnumeration enum;
enum = t.findAll(a-subtree-of-interest);
while ( enum.hasMoreNodes() ) {
  System.out.println(
    enum.nextNode().toStringList()
  );
}
```

## 11.9 一些例子

```
sum :term ( PLUS^ term)*
    ;
```

PLUS 的后辍“^”告诉 ANTLR 生成一个额外的节点，让它作为根，无论哪个子树被构建，直到这个 sum 规则结束。由于对该项的引用，子树的返回被作为该额外的节点的儿子而添加。如果子树没有匹配，被关联的节点将不会被添加到树中。该规则返回与首项引用相匹配的树或 RLUS-rooted 树。

语法注释可以被看成是操作，不是静态的规则。在上述的例子中，每个(...)*的遍历将创建一个新的 PLUS 根，前一个树在左边，从新项中生成的树在右边，这样，保留着往常的“+”的含义。

参考下面的规则，该规则关闭了默认的树构建。

```
decl!:
    modifiers type ID SEMI
        { #decl = #([DECL], ID, ([TYPE] type),
                    ([MOD] modifiers) );  }
    ;
```

在这个例子中，有一个声名被匹配。结果的 AST 有一个“虚构”的 DECL 节点在根上,还有三个子节点。 第一个子节点是定义的 ID。第二个节点是一个拥有虚构型节点作为根节点的子树,AST 的 type 规则作为它的子节点。 第三个子节点是以一个虚构的 MOD 作为根节点的子树,modifiers 规则作为它的子节点。

## 11.10 标签子规则

[这将不会被作为标签子规则来实现...最终我们会做其它的工作。]

在 2.00 版本的 ANTLR 中, 每个规则精确对应于一个树。子规则只是简单地给树添加元素而作为内部规则,该子规则是平常你需要的东西。例如, 表达式树可以通过如下方式简单地构建:

```
expr: ID ( PLUS^ ID )*
    ;
```

然而，很多情况下你会希望子规则的元素能产生一棵不依赖于此规则树的树。 在计算系数的乘法之前，必须先调用该 exponent 的运算。下面的文法正确地匹配该语法。

```
// 匹配指数的运算，如： "3*x^4"
eterm
    :   expr MULT ID EXPONENT expr
```

```
    ;
```

然而，为了产生正确的 AST,你通常会把 ID EXPONENT expr 分离开来成为另外一种规则，如下：

```
eterm:
    expr MULT^ exp
    ;

exp:
        ID EXPONENT^ expr
    ;
```

由这种方式，每个操作将会成为适当子规则的根节点。对于输入 3*x^4,该树将会如下：

```
#(MULT 3 #(EXPONENT ID 4))
```

然而，如果你想让语法保持一样的规则：

```
eterm
    :   expr MULT^ (ID EXPONENT^ expr)
    ;
```

两个"^" 根的操作将会对受影响的树作相同的修改

```
#(EXPONENT #(MULT 3 ID) 4)
```

这棵树以操作符作为根节点，但是它被错误的操作数所关联。 使用一个标签子规则允许原始的规则生成正确的树。

```
eterm
    :   expr MULT^ e:(ID EXPONENT^ expr)
```

```
    ;
```

在这个情况下，对于相同的输入 3*xˆ4，标注型子规则会创建它自己的子树，同时作为 eterm 规则的 MULT 树的操作数。现有的标注改变了子规则里面的元素的 AST 代码的生成，使它的操作更像普通的规则。"ˆ" 符号使为该 token 创建的节点引用到由该子规则生成的树的根。

标注型子规则拥有一个与普通规则具有相同访问方式来访问的 AST 结果。例如, 我们可以使用标注型子规则来重写上面的 decl 例子。(注意在子规则开始的地方使用！来禁止该子规则的自动构建):

```
decl!:
    m:(! modifiers { #m = #([MOD] modifiers); } )
    t:(! type { #t = #([TYPE] type); } )
    ID
    SEMI;
    { #decl = #( [DECL] ID t m ); }
    ;
```

怎么处理子规则的死循环？ 相同的规则应用于一个闭合子规则中--该循环有一棵独立的树，参考由 AST 操作注释的这个循环的元素来构建。例如，考虑以下的规则。

```
term:   Tˆ i:(OPˆ expr)+
    ;
```

对于输入 T OP A OP B OP C, 下面的树结构将会被创建:

```
#(T #(OP #(OP #(OP A) B) C) )
```

这个可以画成如下的图形界面

```
T
|
OP
|
OP--C
|
OP--B
|
A
```

首先需要关注的是子规则中每一个循环过程在相同的树中进行操作。在循环结束后返回的结果树与子规则的标注所关联。上面的标注型子规则的结果树如下：

```
#(OP #(OP #(OP A) B) C)
```

其次要注意的事是,因为 T 首先被匹配，而且在该规则中后面紧跟着一个根操作，所以如果 T 不是针对该子树的标注，T 将会在树的底部。

循环一般是用于创建子树的列表。例如，如果你希望得到一列多项式赋值语句来生成一个同级的列 ASSIGN 子树，通常下面的规则你可以把它分成两个规则。

```
interp
    :   ( ID ASSIGN poly ″; ″ )+
    ;
```

通常地, 需要下面的部份

```
interp
    :   ( assign )+
    ;
assign
```

```
    :   ID ASSIGN^ poly ″; ″!
    ;
```

标注一个子规则允许你更简单地写出上面的例子：

```
interp
    :   ( r:(ID ASSIGN^ poly ″; ″) )+
    ;
```

每一个对子规则的识别作用于树，如果子规则被循环所嵌套，所有的树的返回是以一个树列的型式。(也就是子树的根节点是同级的)。如果标注型子树是以”!”为后辍的，那么由子规则创建的树将不会被连接到和闭合规则或子规则相关联的树。

标注型子规则内部的标注型子规则使树连接到周围的子规则树。例如，下面的规则导致树以下面的形式存在 X #( A #(B C) D) Y。

```
a   :   X r:( A^ s:(B^ C) D) Y
    ;
```

非标注型子规则内部的标注型子规则使树连接到周围的规则树。例如，下面的规则使树以下面的形式存在#(A X #(B C) D Y)。

```
a   :   X ( A^ s:(B^ C) D) Y
    ;
```

## 11.11 引用节点

**没有实现**。 在一棵树中的一个节点只引用到另一个节点。比较好的做法是隐藏相同的树到各种列中。

## 11.12 必需的 AST 功能与形式

由数据结构表明你的树可以拥有任意的形式或类名字，只要它们实现了 AST 的接口：

```
package ANTLR。collections;

/** ANTLR 中最小形式的 AST 节点接口
*  AST 生成与树遍历。
 */
public interface AST {
        /** 获取该节点的 token 类型*/
    public int getType();

        /** 设置该节点的 token 类型 */
    public void setType(int ttype);

    /** 获取该节点的 token 文本 */
    public String getText();

    /** 设置该节点的 token 文本 */
    public void setText(String text);

    /** 获取该节点的第一个孩子;
     *  如果没有则返回 NULL */
    public AST getFirstChild();

    /** 设置节点的第一个孩子 */
    public void setFirstChild(AST c);

    /** 顺序地获取该节点的同属节点
     */
    public AST getNextSibling();

    /**设置该节点的下一同属节点 */
```

```
public void setNextSibling(AST n);

/** 给节点添回 (最右边的) 的儿子 */
public void addChild(AST node);
/** 判断两个节点是否相等? */
public boolean equals(AST t);
/** 两个列表中的节点或子树确实在结构和内容方面相等? */
public boolean equalsList(AST t);
/**两个列表中的节点或子树确实部份相等? 也就是说 'this'
 * 的内容比't' 丰富?
 */
public boolean equalsListPartial(AST t);
/**两个节点或子树确实部份相等? */
public boolean equalsTree(AST t);
 /**两个列表中的节点或子树确实部份相等? 也就是说 'this'
 * 的内容比't' 丰富?
 */
public boolean equalsTreePartial(AST t);
/** 在'this'这树里面返回一个精确匹配于树的遍历     */
public ASTEnumeration findAll(AST tree);
/** 在'this'这树里面返回一个精确匹配于树的遍历
 */
public ASTEnumeration findAllPartial(
    AST subtree);
/** 使用类型与文本来初始化一个节点*/
public void initialize(int t, String txt);
/**以 t 的内容来初始化一个节点*/
public void initialize(AST t);
/** 以 t 的内容来初始化一个节点*/
public void initialize(Token t);
```

```
    /** 转化该节点为可打印的形式*/
    public String toString();
    /** 把’this’当作列表（也就是,
     *  把’this’认为是同属），同时转化为可打印的形式
     *
     */
    public String toStringList();
    /**
     * 把’this’当作根（也就是,不把’this’认为是同属），同时转化为可打印的形式
     */
    public String toStringTree();
}
```

这个方案不排除对异构树的处理使用同构树的方式。然而，你需要写额外的代码来创建异构树(通过一个 ASTFactory 的子类)或者在 token 关联的位置或 tokens 部份中定义节点类型，反之，异构树将会被释放。

Version: $Id: //depot/code/org。ANTLR/release/ANTLR-2.7.6/doc/trees.html#1 $

# 第 12 章 语法继承（Grammar Inheritance）

## 12.1 语法继承（Grammar Inheritance）

像 C++和 Java 这样的面向对象的编程语言都提供了一种机制允许你定义一个新的对象继承自一个已有对象，并且还能为其提供一些不同于已有对象的新功能。这也是"Programming by difference"的关键，它可以节省大量的开发/测试周期，而且将来对基类或者超类进行的修改能够自动的映射到派生类或子类上面。

### 简介和动机（Introduction and motivation）

允许 ANTLR 程序员定义一个新的语法可以继承自某个已存在的语法，并且还能够提供自己所特有的功能，将会大大的减少开发周期，因为程序员只需关心那些不同的或者需要

新增加的语法规则。而且在将来，当基础语法规则改变的时候，所有继承自该语法的派生语法规则都会自动的具备新功能。语法继承同时也提供了一种非常有效的方法去更改某个已存在语法的已有行为。这种机制允许具有相同语法结构的一组规则具有不同行为。

语法继承最显著的作用就是可以使用同一个语言来描述多个不同的方言（Dialect）。以前此类问题的解决办法通常是创建多个版本的语法规则，或者只创建一个语法规则来代表所有的方言，并使用语义来区分它们。而如果使用语法继承，你只需要写一个基本语法包含所有公共属性，然后再为每一种方言都单独写一个派生语法就可以了。这种共享编码（Code Sharing）将体现在语法和最后所输出的解析程序（Parser）两个级别之上。

考虑一下下面这个简单的 English 的子集：

```
class PrimarySchoolEnglish;
sentence
    :   subject predicate
    ;
subject
    :   NOUN
    ;
predicate
    :   VERB
    ;
```

这个语法可以识别这样的句子：Dilbert speaks.

现在来扩展一下这个语法，使它可以包括大部分美国大学生都可以掌握的句子（Sentence）。我们可以直接添加一个新的对象到句子（Sentence）的定义当中，而无需拷贝和修改 PrimarySchoolEnglish 语法的任何代码，只需简简单单的继承它就可以了：

```
class AmericanCollegeEnglish extends
        PrimarySchoolEnglish;
```

```
sentence
    :   subject predicate object
    ;
object
    :   PREPOSITION ARTICLE NOUN
    ;
```

现在这个语法可以描述像“Dilbert speaks to a dog”这样的句子了。然而这实现起来似乎也没什么难度，例如如果输出的解析器的目标语言是 Java，那么只需添加一个 extends 关键字就可以完成任务。事实上并非想象的那么简单，因为在语法分析的过程中还需要确保文法的正确性。例如，为了能够生成正确的代码，ANTLR 需要下推到基本语法然后根据定义的重写规则修改被重写的语法。为了说明这个问题，请考虑一下下面这个简单语言的语法：

```
class Simple;

stat:   expr ASSIGN expr
    |   SEMICOLON
    ;

expr:   ID
    ;
```

显而易见，ID（Token）是识别 stat 子句第一种选择的必要前提。下面考虑一下这个 Simple 的派生方言：

```
class Derived extends Simple;

expr:   ID
    |   INT
    ;
```

在这个例子中，{ ID, INT }是 stat 子句第一种选择的前提。派生语法影响了从基本语法继承过来的识别规则，ANTLR 不仅仅需要在派生语法中重写 expr 语句，同时也必须重写基本语法中的 stat 语句，因为 stat 语句中使用到的 expr 在派生语法中被重写覆盖了。

想要知道由于派生语法的重写而影响到了基本语法中的哪些语法并不是件简单的事情。因此我们的实现过程只是简单的将基本语法拷贝到派生语法当中，然后经过适当的修改生成一个新的、完整的解析器。从程序员的角度来看，实现了编码/语法共享（code/grammar sharing）；但是从实现的角度来讲，只是对基本语法进行拷贝并没有进行真正的共享。

## 12.2 功能（Functionality）

语法 Derived 继承了语法 Base 中所有的规则（Rule）、选项（Option）和行为（Action）.Derived 可以重写任何已有的选项和规则也可以添加新的选项、规则和成员行为。子语法没有继承类或文件选项之外的任何行为（Action）。Base 语法定义如下：

```
class Base extends Parser;
options {
  k = 2;
}
{
  int count = 0;
}
a : A B {an-action}
  | A C
  ;
c : C
  ;
```

可以像下面这样派生一个新的语法：

```
class Derived extends Base;
options {
```

```
  k = 3;            // need more lookahead; override
  buildAST=true; // add an option
}
{
  int size = 0;    // override; no 'count' def here
}
a : A B {an-action}
  | A C {an-extra-action}
  | Z               // add an alt to rule a
  ;
b : a
  | A B D          // requires LL(3)
  ;
```

它等效于下面的语法，实际上 ANTLR 也是按照下面的内容解析这个子语法的：

```
class Derived extends Parser;
options {
        k=3;
        buildAST=true;
}
{
  int size = 0; // override Base action
}
a : A B {an-action}
  | A C {an-extra-action}
  | Z          // add an alt to rule a
  ;

b : a
```

```
  | A B D       // requires LL(3)
  ;

// inherited from grammar Base
c : C
  ;
```

规则的参数和返回类型也可以被重写，例如：

```
class Base extends Parser;
a[int x] returns [int y]
  : A
  ;

class Derived extends Base;
a[float z]
  : A
  ;
```

这种情况下，ANTLR 将会发出一个警告：

*warning: rule Derived.a has different signature than Base.a*

这样的功能，在 Java 中，子语法实际上并没有继承父语法。

## 12.3 父语法（Supergrammar）可以放置的位置

某语法 P 可以访问到的父语法（Supergrammar）位置包括与语法 P 处于同一个语法文件中的任何语法，或者使用如下 ANTLR 命令行包含的语法文件：

```
-glib f1.g;f2.g
```

注：如果上述语法文件和语法 P 所在的语法文件不在同一目录下需要指定该语法文件的路径。

那么父语法（Supergrammar）是如何创建的呢？父语法列表中涉及到的所有语法都会被读取然后创建一个继承层次结构，这个结构中的任何重复语法定义都被忽略。那些被指定的语法文件中的语法也会被添加到这个层次结构当中。如果这个层次结构不完整，ANTLR 将会发出错误信息。当 P 的父语法同时在几个地方定义的时候，与 P 处在同一个文件当中语法会被优先考虑。

语法的类型（Lexer,Parser,TreeParser）是由继承链中最高层次的语法决定。

## 12.4 错误信息（Error Messages）

如果输入的语法文件名称为 T.g（不是通过-glib 提供的文件），ANTLR 会为此文件创建一个名字为 expandedT.g 的文件。这个文件会包含全部的语法（整合了派生语法和父语法），所有的错误信息提示都是与这个文件相关的。在将来也许会提供更好的解决方案。

Version: $Id: //depot/code/org.ANTLR/release/ANTLR-2.7.6/doc/inheritance.html#1 $

# 第 13 章 选项（Options）

## 13.1 文件、语法和规则的选项（File, Grammar, and Rule Options）

程序员不需要再使用命令行的方式为解析器生成器指定相关的参数了，语法自身的选项（Option）部分就可以完成这样的功能。这种方法更加优越是因为它在语法中结合了所需要的参数，而不是通过 ANTLR 来进行相关的调用。选项部分是位于语法部分前面的用 options 关键字标识的一段区域，该区域是用大括号“{}”包围着，并且内部包含了多个选项/值形式的赋值语句。例如：

```
options {
```

```
    k = 2；
    tokenVocbaulary = IDL；
    defaultErrorHandler = false；
}
```

如果指定的选项区域（options section）紧接着位于 header 区域的下面，那么这些选项是对整个文件（.g）起作用的，如：

```
header { package X；  }
options {language="FOO"；}
```

如果指定的选项区域紧接着位于类描述语句结束分号“；”的后面，那么这些选项是对这个语法（Grammar）起作用的，如：

```
class MyParser extends Parser；
options { k=2；  }
```

如果指定的选项区域紧接着位于规则名称的后面，那么这些选项只对该规则（Rule）起作用，如：

```
myrule[args] returns [retval]
    options { defaultErrorHandler=false；  }
    :    // body of rule…
    ;
```

选项的名称不是 ANTLR 的关键字，但是它是 ANTLR 所定义的符号表中的实体。选项名称的范围仅限于在选项区域（options section）中使用才有意义，语法部分的标识符如果使用了选项的名称那么它将不再具有选项符号表中所代表的意义（译者注：正是因为如此，所以选项的名称不是关键字，因为关键字是不能在语法部分被当成标识符来使用的，而选项的名称可以，但意义不一样）。

不包括在 options 区域中的选项（命令行选项）是与语法本身无关的，它们都是用来指定 ANTLR 调用参数的。最好例子就是设置调试信息这个选项，一般来讲，程序员都想创建一个 makefile 文件用来指定 debug 还是 release 选项的。

### 13.1.1 ANTLR 中支持的选项（Options supported in ANTLR）

类型的缩写： F=file, G=grammar, R=rule, L=lexer, S=subrule, C=C++，标 C 表示该选项只对 C++起作用。

| 符号（Symbol） | 类型（Type） | 描述（Description） |
|---|---|---|
| language | F | 设置目标生成语言 |
| k | G | 设置 lookahead 向前探测的深度 |
| importVocab | G | 初始化语法词汇表 |
| exportVocab | G | 设置从语法导出的词汇表名称 |
| testLiterals | LG,LR | 是否生成探测常量的代码 |
| defaultErrorHandler | G,R | 是否使用默认的异常处理代码 |
| greedy | S | False 表示像(..)*和(..)+这样的循环子规则在它探测到后面的循环时退出。 |
| codeGenMakeSwitchThreshold | G | 控制代码的生成 |
| codeGenBitsetTestThreshold | G | 控制代码的生成 |
| buildAST | G | 设置自动创建 AST in Parser (AST 转换模式 in Tree-Parser) |
| analyzerDebug | G | 在执行文法分析的时候是否输出 Debug 信 |

| | | |
|---|---|---|
| | | 息 |
| codeGenDebug | G | 在生成代码的时候是否输出 Debug 信息 |
| ASTLabelType | G | 设置用户自定义的节点类型 |
| charVocabulary | LG | 设置词法分析器所使用的字符集 |
| interactive | G | 词法分析器和语法分析器都有一个 interactive 选项，并且默认值都为 false，可以查看前面的 Parser Speed 一节获取更多信息。 |
| caseSensitive | LG | 在 Lexer 指定该值表示 character 和 string literals 是否大小写敏感。如果在 Token 对象中指定该值表示输入字符流是否大小写敏感。 |
| ignore | LR | 是否忽略空格。这是一个词法规则选项。 |
| paraphrase | LR | 在错误处理的过程中，这个选项提供了一种简便的方法替换一个 Token 名字为指定的字符串。 |
| caseSensitiveLiterals | LG | 在比较 Token 和字符表的时候是否大小写敏感。 |
| classHeaderPrefix | G | class 的前缀描述符(如 Java 中的"public") 。 |
| classHeaderSuffix | G | Class 的后缀描述符，如 Java 中的 Lexer、Parser 或者 Tree-Walker 必须实现的接口。 |
| mangleLiteralPrefix | F | 设置 Token 类型定义的前缀，默认的前缀为"TOKEN_".. |

| | | |
|---|---|---|
| warnWhenFollowAmbig | S | 当包含有空子规则或者像(…)+、(…)*这样的循环闭包子规则时，该选项用来控制是否输出警告信息，默认值为 true。 |
| generateAmbigWarnings | S | 当为 true 的时候，不会生成不确定性的警告。使用该选项的时候需要非常小心，因为你可能由于设置这个选项而没有注意到一个隐含的不确定性因素。使用的时候一定要确保 ANTLR 能够正确的处理不确定的子规则，ANTLR 生成的语法分析器采用的是尽可能早的消化输入来解决语言不确定性（或者采用可选子规则列表中的第一个）。<br><br>查看 Java 和 HTML 相关文档中关于该属性的正确用法。 |
| filter | LG | 设置为 true 的时候，词法分析会忽略掉那些没有完全匹配不受保护的词法规则的输入。当为一个规则设置该选项时，解析器只解析那些有效的、重要的 Token 字符。 |
| namespace | FGC | 设置该选项，生成的所有的 C++代码都是在该命名空间下。 |
| namespaceStd | FGC | 设置该选项后，生成的 C++代码中宏 ANTLR_USE_NAMESPACE(std)会被该值取代。这个选项提供的目的只是为了提高可读性。 |
| namespaceANTLR | FGC | 同 namespaceStd，该值会修改生成的 C++代码中的 |

| | | |
|---|---|---|
| | | ANTLR_USE_NAMESPACE(ANTLR)。 |
| genHashLines | FGC | Boolean 类型,当设置为 true 的时候，#line <linenumber> "filename" 会被插入到生成的代码中。 |
| noConstructors | FGLC | 设置为 true 的时候，lexer/parser/treewalker 的默认构造器将会被忽略。用户需另行指定。 |

### 13.1.2 language: 设置生成的目标语言

ANTLR 内置了多种代码生成器，而且其中的任何一种代码生成器都可以通过设置 language 选项来调用它。默认的选项设置为“Java”，当然 ANTLR 同时也支持 "Cpp" 和 "CSharp" 等选项。language 选项只允许在文件级别（file-level）上指定，例如：

```
header { package zparse； }
options { language="Java"； }
... classes follow ...
```

### 13.1.3 k: 设置 lookahead（前瞻）的深度

可以使用 k= 选项来设置任何语法（包括 parser, lexer 或者 tree-walker）的 lookahead 深度，如：

```
class MyLexer extends Lexer；
options { k=3； }
...
```

通过设置 lookahead 深度可以更改当有多个可选方案或者在试探 EBNF 表达式 (...)?, (...)+, 和 (...)* 等的结束条件时允许向前探测 Token 的最大个数。预测分析与 LL(K) 分析不同，它是近似线性（linear approximate）的分析方法，考虑下面这个例子，它涉及到了更多的细节，其中 k=2：

```
r:  ( A B | B A )
  |  A A
  ;
```

对于 r 中的第一个表达式，LL(k) 分析会采用如下判断方法去探测分析并区别于其它表达式（注：LA(1) 就是向前探测一个 Token）：

```
if ( (LA(1)==A && LA(2)==B) || (LA(1)==B && LA(2)==A) )
```

然而，近似线性分析则会对每一个深度的 Token 采用 OR 的逻辑运算，判断语句如下：

```
if ( (LA(1)==A || LA(1)==B) && (LA(2)==A || LA(2)==B) )
```

显而易见，上面的这个判断语句是无法将 r 中的第一个表达式与 (A A) 表达式区分开的。正是由于这个原因，lookahead 深度设置的太大往往会起到不好的作用，这是因为 lookahead 设置的越大它所包括的可能性就越多。

### 13.1.4 importVocab: 初始化语法词汇表

[请查看文档中的 vocabularies 部分获取等多的信息]

可以通过在语法的选项区域设置 importVocab 选项为语法指定初始化词汇表（tokens, literals, 和 token types）。

```
class MyParser extends Parser;
options {
    importVocab=V;
}
```

ANTLR 会在当前目录下的 VTokenTypes.txt 文件和预先加载的 MyParser 的 Token 管理器附加信息中查找词汇表的相关信息。

这个选项是非常有用的，例如，你创建了一个外部词法分析器（lexer）并且想把它连接到一个 ANTLR 的解析器（parser）上。或者相反，你可以创建一个外部的语法解析器（parser）并且想用 ANTLR 的分析器（lexer）来分析你的 Token。你会发现这都是非常方便的事情，

因为你把语法文件放在了单独的文件当中。尤其是如果你有多个 tree-walkers 的时候你甚至不需要添加任何的 literal 到 Token 集合中。

词汇表文件的第一行用来命名这个词汇表，它下面的每一行都是如同 ID=value 或者 "literal"=value 这样的键/值对形式的语句，每行一个。如：

```
ANTLR // vocabulary name
"header"=3
ACTION=4
COLON=5
SEMI=6
...
```

注意：必须在运行 vocabulary-consuming 语法文件之前先运行 vocabulay-generating。

### 13.1.5 exportVocab: 指定导出词汇表的名称

[请查看文档中的 vocabularies 部分获取等多的信息]

一个语法的词汇表是由 importVocab 选项所指定的 Token 集合以及语法中定义的 Token 和 Literal 集合所组成的。ANTLR 默认情况下为每个语法导出一个以该语法名字命名的词汇表文件，例如，下面的这个语法名字为 P：

```
class P extends Parser;
a : A;
```

ANTRL 默认情况下（没有设置 exportVocab 选项的值）会生成 PTokenTypes。txt 和 PTokenTypes。java 两个文件。

可以通过设置 exportVocab 选项的值指定导出的词汇表的名字。例如，下面的语法导

出的词汇表是 V 而不是 P：

```
class P extends Parser；
options {
   exportVocab=V；
}
a : A；
```

所有在同一个文件中指定了相同的词汇表名称的语法会创建相同的词汇表（词汇表文件同样也一样）。如果这样的语法处在不同的语法文件当中，由于它们最终创建的词汇表文件名称一样，所以其中的一些将会被最后一个词汇表覆盖。例子如下，它们的词汇表名称都是 MyTokens：

```
class MyParser extends Parser；
options {
   exportVocab=MyTokens；
}
...

class MyLexer extends Lexer；
options {
   exportVocab=MyTokens；
}
...
```

### 13.1.6 testLiterals：是否生成常量检测代码

默认情况下，ANTLR 在所有的词法分析器中都会生成检测 Token 是否属于常量表中的常量的代码，一旦发现 Token 与常量表中的值匹配就会修改当前 Token 的类型。当然，也可以通过设置 testLiterals 选项来禁止生成这样的代码：

```
class L extends Lexer；
options { testLiterals=false；  }
...
```

如果你禁止了词法分析器的 testLiterals 选项，那么，你还可以单独的为 Rule 重新启用该功能。这是非常有用的功能，例如下面这种情况，除了 ID 之外其余的都是关键字。

```
ID
options { testLiterals=true；  }
    : LETTER (LETTER | DIGIT)*
    ;
```

如果只想检测一个 Token 字符串中的一部分，可以在 action 中使用如下方法：

```
    public int testLiteralsTable(String text, int ttype) {...}
```

例如，分析 HTML 记号的时候只想检测记号符号“<>”中的字符串是否是正确记号，就可以用该方法。

### 13.1.7 defaultErrorHandler: 设置默认的错误处理器

默认情况下，ANTLR 会为 Parser 和 Tree-Parser 规则生成默认的错误处理代码来处理当中的错误。生成的异常处理代码会捕获所有的解析错误，并同步到规则的集合当中，然后返回。这是一种简单的、经常使用的异常处理机制，但是它并不足够的灵活、并不适合所有情况。当你想要自己提供这个异常处理代码的时候，ANTLR 会在你指定了自己的异常处理代码的地方关闭默认的异常处理。你也可以精确的为每一个语法或者规则指定是否生成默认的异常处理代码，例如，下面的代码就是对整个语法关闭了默认的异常处理，但是对规则“r”启用了默认的异常处理。

```
class P extends Parser；
options {defaultErrorHandler=false； }

r
options {defaultErrorHandler=true； }
: A B C；
```

点击 这里 可以了解更多关于词法分析器的异常处理信息。

### 13.1.8 codeGenMakeSwitchThreshold: 控制代码的生成

ANTLR 对那些具有大量可选情况的规则进行了优化，使用 switch 语句代替了一大堆的 if/else 语句来预测后面的 Token。codeGenMakeSwitchThreshold 选项就是用来控制这个的（注：当可选情况大于或等于指定数字的时候就会使用 switch 语句代替 if 语句）。你可能想使用这个选项来优化解析器，但是你也可能由于 debug 的原因想要禁止这个功能，那么只需将它设置为一个很大的数值就可以了：

```
class P extends Parser；
options { codeGenMakeSwitchThreshold=999； }
...
```

### 13.1.9 codeGenBitsetTestThreshold: 控制代码的生成

与 codeGenMakeSwitchThreshold 选项类似，它也是为了优化解析器的，当遇到复杂的情况时它采用位集合（Bitset）的方式来代替 if 语句。可以通过修改该值来优化最终生成的 Parser：

```
class P extends Parser；
// 当遇到 5 种或 5 种以上的选择的时候启用 Bitset。
options { codeGenBitsetTestThreshold=5； }
...
```

当然如果为了 Debug 等目的也可以将它的值设置的非常大来关闭这个优化功能：

```
class P extends Parser；
options { codeGenBitsetTestThreshold=999； }
…
```

### 13.1.10 buildAST: 自动创建抽象语法树（AST）

在 Parser 中，可以让 ANTLR 根据识别的语法结构生成能够创建相应 AST 的代码。buildAST 选项设置为 true 就可以让 ANTLR 生成创建 AST 的代码。并且可以使用所有的创建 AST 的语法和方法。

在 Tree-Parser 中，这个选项打开了一种“转换模式”，将输入的 AST 转换并输出 AST。遍历树的时候，buildAST 选项的默认行为是将它所遍历到的输入 AST 的那部分生成一份拷贝作为输出 AST。树转换和在 Parser 中创建 AST 基本上是一样的，唯一不同的是一个的输入是 AST，另一个的输入是 Token 流。

### 13.1.11 ASTLabelType: 设置节点类型

当需要自定义的 AST 节点类型的时候，语法的 Action 中需要非常多的从 AST（用户自定义节点类型默认为 AST）到自定义类型的类型转换代码，例如：

```
decl : d:ID {MyAST t=(MyAST)#d； }
     ；
```

上段代码输入又麻烦而且又不容易阅读。为了避免这种情况，可以通过设置语法选项中的 ASTLabelType 让 ANTLR 自动的转换并定义合适的节点类型。

```
class ExprParser extends Parser；
```

```
options {
   buildAST=true；
   ASTLabelType = "MyAST"；
}

expr : a:term ；
```

上例中的 #a 的节点类型为 MyAST 而不是 AST。

### 13.1.12 charVocabulary: 设置词法分析器的字符表

ANTLR 处理 Unicode 字符，正因为如此，ANTLR 在处理的时候不能对实用的字符集做任何的假想，而且这样也会使生成的词法分析器非常庞大。相反，ANTLR 认为词法分析器中的字符常量、字符串常量以及所使用的字符范围组成了整个字符集。例如，如下的分析器：

```
class L extends Lexer；
A : 'a'；
B : 'b'；
DIGIT : '0' .. '9'；
```

这个词法分析器暗指的字符集为{ 'a', 'b', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9' }。如果你认为普通的 ASCII 字符字符集都允许使用则会产生意想不到的结果。例如，在下面的这个 Lexer 规则：

```
class L extends Lexer；
A : 'a'；
B : 'b'；
DIGIT : '0' .. '9'；
STRING: '"' (~'"')* '"'；
```

这个 Lexer 规则中的 STRING The lexer rule STRING will only match strings containing 'a', 'b' and the digits, which is usually not what you want。 可以通过设置 charVocbaulary 选项的值来控制词法分析器中所使用的字符集。例如下面的这个例子就是使用了一个 8 位的字符集（实际上就是 ASCII）。

```
class L extends Lexer；
options { charVocabulary = '\3'..'\377'；  }
...
```

下面的这个例子使用的是 ASCII 字符集和部分的 Unicode 字符：

```
class L extends Lexer；
options {
    charVocabulary = '\3'..'\377' | '\u1000'..'\u1fff'；
}
...
```

### 13.1.13 warnWhenFollowAmbig

[**警告：**在使用这个选项之前你必须确切的知道自己在做什么。我（ANTLR 的作者）故意的将关闭警告操作设置的非常麻烦，是为了让大家知道这不仅仅只是关掉了所有的警告。在实现这个特性之前我考虑了很长的时间。在使用这个选项之前建议大家好好看看下面的说明，它会向大家解析如何安全的关掉警告]

这个子规则选项的默认值为 true，当任何子规则的 FOLLOW 集合为空或者包含像(..)+和(...)* 等这样的闭包时，这个选项就是用来控制这种不确定性的警告是否产生。例如下面的这个简单规则包括了一个不确定的子规则，因为 ELSE 子句即可以和与他最近的一个 IF 语句匹配，也可以与最外层的 IF 语句匹配。

```
stat :    "if" expr "then" stat ("else" stat)?
```

```
    |   ID ASSIGN expr SEMI
    ;
```

由于这个语言本身的不确定性，所以它的上下文无关文法必定是不确定的并且产生的 Parser 也是不确定的（理论上是这样）。然而，像我们这样的有实际语言经验的人都知道，可以让 ANTLR 通过尽早的消化输入来解决这个冲突。不过，顺便说一下，我曾经遇到过一个例子采用这种办法解决不确定性是错误的。当把这个选项设置为 False 的时候，仅仅只是让 ANTLR 知道你已经确认了使用这种方法来解决语言中的不确定性，从而可以关闭遇到这从情况时产生的警告信息。下面就是一个不会发出警告信息的规则：

```
stat    :       "if" expr "then" stat
                (
                    // standard if-then-else ambig
                    options {
                        warnWhenFollowAmbig=false;
                    }
                :       "else" stat
                )?
        |       ID ASSIGN expr SEMI
        ;
```

需要注意的一点：这个选项不会影响到非空的选择（即子规则全都是不为空的）。例如，下面的例子仍然会产生警告信息：

```
(
        options {
                warnWhenFollowAmbig=false;
        }
:       A
|       B
|       A
)
```

另外，这个选项不会影响到 lookahead 操。只有有空选择作为候选子规则的时候才会关闭警告，所以，当 k=2 的时候，因为 ANTLR 能够查看前面的 Token，所以仍然可以产生警告。

## 13.2 命令行选项（Command Line Options）

| | |
|---|---|
| -o outputDir | 指定输出目录 |
| -glib supergrammarFile | 指定 supergrammar 文件 |
| -debug | 启动 ParseView 调试器。除非下载并且解压调试器文件到标准的 ANTLR 发布文件夹内，否则这个选项没有被编译不能使用。 |
| -html | 根据语法文件生成 HTML 文件，注意不包括其中的 action 等。它仅仅是一个原型（**prototype**）,但是它是非常有用处的。 不过它只对 Parser 起作用，对 Lexer 和 Tree-Parser 都不起作用。 |
| -docbook | 使用它生成的内容和 –html 选项一样，只不过它生成的是 SGML 文档文件。 |
| -diagnostic | 生成一个包含语法 debug 信息的文本文件。 |
| -trace | 让所有的规则都调用 traceIn/traceOut。 |
| -traceParser | 让 Parser 规则调用 traceIn/traceOut。 |
| -traceLexer | 让 Lexer 规则调用 traceIn/traceOut。 |
| -traceTreeParser | 让 Tree-Walker 规则调用 traceIn/traceOut。 |
| -h\|-help\|--help | 帮助信息 |

Version: $Id: //depot/code/org.ANTLR/release/ANTLR-2.7.6/doc/option.html#1 $