Table of Contents

ANTLR 4简明教程

本书是根据最新的ANTLR 4语法编写的简明教程，源于我在阅读《The Definitive ANTLR 4 Reference》时做的读书笔记，增加了部分全新的内容。

本书以开源的形式编写，源码托管在Github上，欢迎参与维护：https://github.com/dohkoos/antlr4-short-course 。

在线阅读：《ANTLR 4简明教程》。

如果在阅读过程中发现有什么问题，请到这里发言。

安装ANTLR

ANTLR是由Java写成的，所以在安装ANTLR前必须保证已经安装有Java 1.6或以上版本。你可以到 http://www.antlr.org/download.html 下载ANTLR的最新版本，或者也可使用命令行工具。

```
curl -O http://www.antlr.org/download/antlr-4.5.1-complete.jar
```

antlr-4.5.1-complete.jar包含运行ANTLR工具的所有必要依赖，以及编译和执行由ANTLR生成的识别器所需的运行库。ANTLR工具将由语法文件描述的语法转换成识别程序，识别程序利用ANTLR运行库中的某些支持类识别输入的语句。该jar包还包含两个支持库：TreeLayout（一个复杂的树布局库）和StringTemplate（一个用于生成代码和其它结构化文本的模板引擎）。

现在来测试下ANTLR工具是否工作正常：

```
java -jar antlr-4.5.1-complete.jar  # 启动org.antlr.v4.Tool
```

如果正常会看到以下帮助信息：

```
ANTLR Parser Generator  Version 4.5.1
 -o ___              specify output directory where all output is generated
 -lib ___            specify location of grammars, tokens files
 ...
```

每次运行ANTLR工具都要输入这么长的命令是否有些痛苦？写个脚本来解放我们的手指吧！

```
#!/bin/sh
java -cp .:./antlr-4.5.1-complete.jar:$CLASSPATH org.antlr.v4.Tool $*
```

把它保存为antlr.sh，以后就可以使用下列命令来运行ANTLR工具了：

```
antlr
```

开始

先看下面这段用于识别像hello world那样短语的简单语法：

```
grammar Hello;              // 定义文法的名字

s  : 'hello' ID ;           // 匹配关键字hello，后面跟着一个标志符
ID : [a-z]+ ;               // 匹配小写字母标志符
WS : [ \t\r\n]+ -> skip ;   // 跳过空格、制表符、回车符和换行符
```

把以上语法保存为Hello.g，然后执行以下命令来生成识别器：

```
antlr Hello.g
```

该命令会在相同目录下生成后缀名为tokens和java的六个文件：

```
Hello.tokens        HelloLexer.java         HelloParser.java
HelloLexer.tokens   HelloBaseListener.java  HelloListener.java
```

现在开始准备编译由ANTLR生成的Java代码。先写个脚本把编译命令包装起来：

```
#!/bin/sh
javac -cp .:./antlr-4.5.1-complete.jar:$CLASSPATH $*
```

把它保存为compile.sh文件，然后你就可以用以下命令编译代码了：

```
compile *.java
```

到此，我们已经有了一个可以被HelloParser和HelloLexer利用的可执行的识别器，只缺一个主程序去触发语言识别了。

ANTLR运行库有提供称之为TestRig的测试工具，可以让你不创建主程序就能测试语法。TestRig使用Java反射调用编译后的识别器，它能显示关于识别器如何匹配输入的大量信息。

同样地，创建一个脚本grun.sh来简化以后的打字数：

```
#!/bin/sh
java -cp .:./antlr-4.5.1-complete.jar:$CLASSPATH org.antlr.v4.gui.TestRig $*
```

现在，让我们来打印出识别期间创建的那些记号（记号是指像关键字hello和标识符world那样的词汇符号）：

```
grun Hello s -tokens
```

敲入上述命令并按回车，接着输入以下内容：

```
hello world  # 输入并按回车
EOF          # Unix系统输入Ctrl+D或Windows系统输入Ctrl+Z并按回车
```

TestRig会打印出记号列表，每一行输出表示一个记号以及它的有关信息：

```
[@0,0:4='hello',<1>,1:0]
[@1,6:10='world',<2>,1:6]
[@2,13:12='<EOF>',<-1>,2:0]
```

这里详细讲解下[@1,6:10='world',<2>,1:6]的意义。@1表示记号索引（从0开始）；6:10表示记号开始与结束的位置（从0开始）；<2>表示记号类型，具体数值和类型存储在后缀名为tokens的文件中；最后的1:6表示记号在第一行（从1开始），从第6个字符开始（从0开始，制表符作为单个字符计算）。

除此之外，还可以以LISP风格的文本形式查看记号：

```
grun Hello s -tree
```

它会输出如下形式的记号：

```
(s hello world)  # (root children)
```

你也可以以可视化的方式查看语法分析树：

```
grun Hello s -gui
```

以下是TestRig可用的所有参数：

- -tokens 打印出记号流。
- -tree 以LISP风格的文本形式打印出语法分析树。
- -gui 在对话框中可视化地显示语法分析树。
- -ps file.ps 在PostScript中生成一个可视化的语法分析树表示，并把它存储在file.ps文件中。
- -encoding encodingname 指定输入文件的编码。
- -trace 在进入/退出规则前打印规则名字和当前的记号。
- -diagnostics 分析时打开诊断消息。此生成消息仅用于异常情况，如二义性输入短语。
- -SLL 使用更快但稍弱的分析策略。

概念

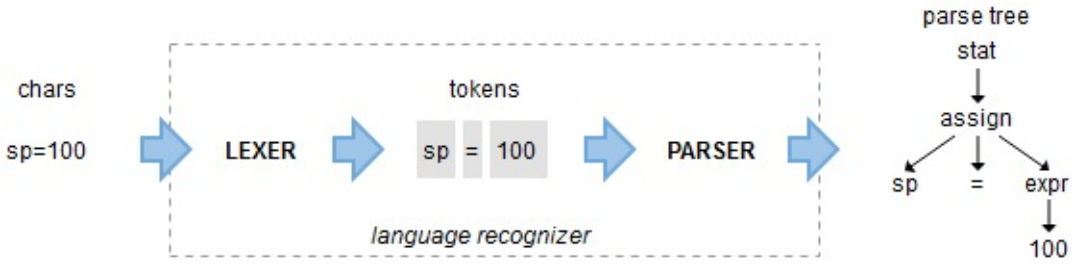
一门语言由有效的句子组成，一个句子由短语组成，一个短语由子短语和词汇符号组成。要实现一门语言，我们必须构建一个能读取句子以及对发现的短语和输入符号作出适当反应的应用。

这样的应用必须能识别特定语言的所有有效的句子、短语和子短语。识别一个短语意味着我们能确定短语的各种组件并能指出它与其它短语的区别。例如，我们把输入a=5识别为赋值语句，这就意味着我们知道a是赋值目标以及5是要存储的值。识别赋值语句a=5也意味着应用认为它是明显不同于，比如说，a+b语句的。在识别后，应用将执行适当的操作，例如performAssignment("a", 5)或者translateAssignment("a", 5)。

识别语言的程序被称为语法分析器。语法指代控制语言成员的规则，每条规则都表示一个短语的结构。为了更容易地实现识别语言的程序，通常我们会把识别语言的语法分析拆解成两个相似但不同的任务或阶段。

把字符组成单词或符号（记号）的过程被称为词法分析或简单标记化。我们把标记输入的程序称为词法分析器。词法分析器能把相关的记号组成记号类型，例如INT（整数）、ID（标志符）、FLOAT（浮点数）等。当语法分析器只关心类型的时候，词法分析器会把词汇符号组成类型，而不是单独的符号。记号至少包含两块信息：记号类型（确定词法结构）和匹配记号的文本。

第二阶段是真正的语法分析器，它使用这些记号去识别句子结构，在本例中是赋值语句。默认情况下，ANTLR生成的语法分析器会构建一个称为语法分析树或语法树的数据结构，它记录语法分析器如何识别输入句子的结构和它的组件短语。下图阐明了语言识别器的基本数据流：



语法分析树的内部节点是分组和确认它们子节点的短语名字。根节点是最抽象的短语名字，在本例中是prog（“program”的缩写）。语法分析树的叶子节点永远是输入记号。

通过生成语法分析树，语法分析器给应用的其余部分提供了方便的数据结构，它们含有关于语法分析器如何把符号组成短语的完整信息。树是非常容易处理的，并且也能被程序员很好的理解。更好的是，语法分析器能自动地生成语法分析树。

通过操作语法分析树，需要识别相同语言的多个应用能重用同一个语法分析器。当然，你也可以选择直接在语法中嵌入特定应用的代码片段，这是语法分析器生成器传统的做法。ANTLR v4仍然允许这样做，但是语法分析树有助于更简洁更解耦的设计。

语法分析树对于需要多次树遍历的转换也是非常有用的，因为在计算依赖关系的阶段通常会需要前一个阶段的信息。相比于在每个阶段都要准备输入字符，我们只需要遍历语法分析树多次，更具有效率。

因为我们用一套规则指定短语，语法分析树子树根节点对应于语法规则名。这里的语法规则对应于上图中assign子树的第一层：

```
assign : ID '=' expr ;    // 匹配赋值语句像"a=5"
```

明白ANTLR如何把这些规则转换为人类可读的语法分析代码是使用和调试语法的基础，因此让我们深入地挖掘语法分析是如何工作的。

实现语法分析器

ANTLR工具根据语法规则，例如我们刚才看到的assign，生成递归下降语法分析器。递归下降语法分析器只是递归方法的一个集合，每个规则一个方法。下降这个术语指的是分析从语法分析树的根开始向着叶子进行（记号）。我们首先调用的规则，即prog符号，成为语法分析树的根。那也就意味着对前面部分的语法分析树来说需要调用方法prog()。这类分析更通用的术语是自顶向下分析：递归下降语法分析器仅仅是自顶向下语法分析器实现的一种。

要了解递归下降语法分析器是什么样子，这里是ANTLR为规则assign生成的方法（稍微整理）：

```
// assign : ID '=' expr ;
void assign() {    // 根据规则assign生成的方法
    match(ID);     // 比较ID和当前输入符号然后消费
    match('=');
    expr();        // 通过调用expr()匹配表达式
}
```

递归下降语法分析器最酷的部分是通过调用方法prog()、assign()和expr()跟踪出的调用关系图反映了内部的语法分析树节点。match()的调用对应语法分析树叶子。为了在一个手工构建的语法分析器中手动构建一颗语法分析树，我们需要在每个规则方法的开始处插入“添加新子树根”操作，以及给match()一个“添加新叶子节点”操作。

方法assign()只是检查确保所有必要的记号存在且以正确的顺序。当语法分析器进入assign()时，它不必在多个选项之间进行选择。选项是规则定义右边的选择之一。例如，调用assign的prog规则可能有其它类型的语句。

```
/** 匹配起始于当前输入位置的任何语句 */
prog
    : assign    // 第一个选项（'|'是选项分隔符）
    | ifstat    // 第二个选项
    | whilestat
    ...
    ;
```

prog的分析规则看起来像一条switch语句：

```
void prog() {
    switch ( «current input token» ) {
        CASE ID : assign(); break;
        CASE IF : ifstat(); break;    // IF是关键字'if'的记号类型
        CASE WHILE : whilestat(); break;
        ...
        default : «raise no viable alternative exception»
    }
}
```

方法prog()必须通过检查下一个输入记号作出分析决定或预测。分析决定预判哪一个选项将会成功。在本例中，当看到WHILE关键字时会预判是规则prog的第三个选项。规则方法prog()然后就会调用whilestat()。你以前可能听说过术语预读记号，那只是下一个输入记号。预读记号可以是语法分析器在匹配和消费它之前嗅探的任何记号。

有时候，语法分析器需要一些预读记号去预判哪个选项会成功。它甚至必须考虑从当前位置直到文件结尾的所有的记号！ANTLR默默地为你处理所有的这些事情，但是对决策过程有个基本的了解是有帮助的，可以让调试生成的语法分析器更容易。

为更好地理解分析决定，想象有个单一入口和单一出口的迷宫，有单词写在地板上。每个沿着从入口到出口路径的单词序列表示一个句子。迷宫的结构与定义一门语言的语法规则类似。为测试一个句子在一门语言中的成员身份，我们在穿越迷宫时把句子的单词和沿着地板的单词作比较。如果通过句子的单词我们能到达出口，那么句子是有效的。

为了通过迷宫，我们必须在每个岔口选择一条有效路径，正如我们必须在语法分析器中选择选项。我们必须决定该走哪条路，通过把我们句子中下一个单词（们）和沿着来自每个岔口的每条路径上可见的单词比较。我们能从岔口看到的单词与预读记号类似。当每条路径以唯一的单词开始时决定是相当容易的。在规则prog中，每个选项从唯一的记号开始，因此prog()可以通过查看第一个预读记号识别选项。

当单词从一个岔口重叠部分开始每条路径时，语法分析器需要继续往前看，扫描可以识别选项的单词。ANTLR根据需要为每个决定自动上下调节预读数量。如果预读的结果是多条同样的到出口的路径，即当前的输入短语有多种解释。解决这样的二义性将是我们的下一个主题。

二义性

一个模棱两可的短语或句子通常是指它有不止一种解释。换句话说，短语或句子能适配不止一种语法结构。要解释或转换一个短语，程序必须要能唯一地确认它的含义，这意味着我们必须提供无歧义的语法，以便生成的语法分析器能用明确的一个方法匹配每个输入短语。

在这里，让我们展示一些有歧义的语法以便让二义性的概念更具体。如果你以后在构建语法时陷入二义性，你可以参考本节的内容。

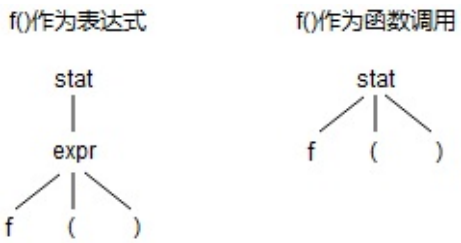
一些明显有歧义的语法：

```
assign
    : ID '=' expr    // 匹配一个赋值语句，例如f()
    | ID '=' expr    // 前面选项的精确复制
    ;

expr
    : INT ;
```

大多数时候二义性是不明显的，如同以下的语法，它能通过规则stat的两个选项匹配函数调用：

```
stat
    : expr          // 表达式语句
    | ID '(' ')'    // 函数调用语句
    ;

expr
    : ID '(' ')'
    | INT
    ;
```

这里是两个输入f()的解释，从规则stat开始：



左边的语法分析树显示f()匹配规则expr。右边的语法分析树显示f()匹配规则stat的第二个选项。

因为大部分语言它们的语法都被设计成无歧义的，有歧义的语法类似于编程缺陷。我们需要识别语法以便为每个输入短语提交单一选择给语法分析器。如果语法分析器发现一个有歧义的短语，它必须选一个可行的选项。ANTLR通过选择涉及决定的第一个选项解决二义性。在本例中，语法分析器将选择与左边的语法分析树有关的f()的解释。

二义性可以发生在词法分析器中也能发生在语法分析器中，但ANTLR可以自动地解决它们。ANTLR通过使输入字符串和语法中第一个指定的规则匹配来解决词法二义性。为了明白这是如何工作的，让我们看看对大部分编程语言都很普遍的二义性：在关键字和标志符规则中的二义性。关键字begin（后面有个非字母）也是标志符，至少词法上，因此词法分析器可以匹配b-e-g-i-n到两者中的任何一个规则。

```
BEGIN : 'begin' ;    // 匹配b-e-g-i-n序列，即把二义性解析为BEGIN
ID    : [a-z]+ ;     // 匹配一个或多个任意小写字母
```

注意，词法分析器会试着为每个记号尽可能匹配最长的字符串，这意味着输入beginner将仅匹配规则ID。词法分析器不会把beginner匹配成BEGIN随后ID匹配输入ner。

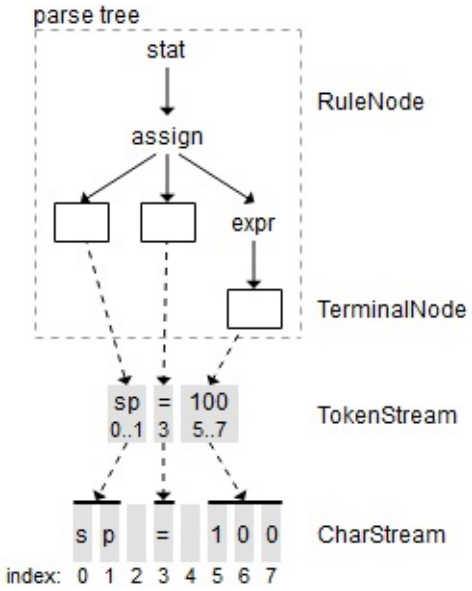
有时候语言的语法就明显有歧义，没有任何的语法重组能改变这个事实。例如，算术表达式的自然语法可以用两种方式解释输入像1+2*3这样，要么执行运算符从左到右，要么像大部分语言那样按优先级顺序。

C语言展示了另一种二义性，但我们可以使用上下文信息比如标志符如何被定义来解决它。考虑代码片段i*j;。在语法上，它看起来像是一个表达式，但它的含义或者语义依赖i是类型名还是变量。如果i是类型名，那么这个片段不是表达式，而是一个声明为指向类型i的指针变量j。

语法分析树

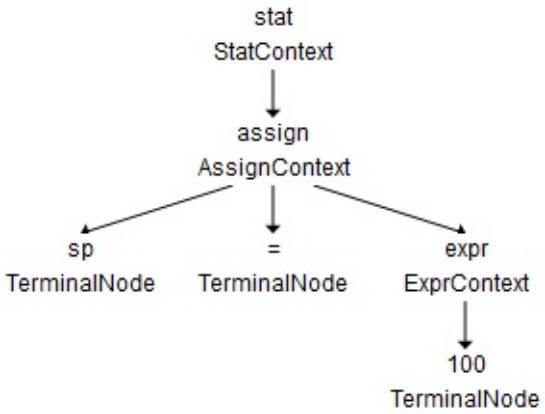
为制作语言应用，我们必须为每个输入短语或子短语执行一些适当的代码，那样做最简单的方法是操作由语法分析器自动创建的语法分析树。

早些时候我们已经学习了词法分析器处理字符和把记号传递给语法分析器，然后语法分析器分析语法和创建语法分析树的相关知识。对应的ANTLR类分别是CharStream、Lexer、Token、Parser和ParseTree。连接词法分析器和语法分析器的管道被称为TokenStream。下图说明了这些类型的对象如何连接到内存中其它的对象。



这些ANTLR数据结构分享尽可能多的数据以便节省内存的需要。上图显示在语法分析树中的叶子（记号）节点含有在记号流中记号的点。记号记录开始和结束字符在CharStream中的索引，而不是复制子串。这里没有与空格字符有关的记号，因为我们假设我们的词法分析器扔掉了空格。

下图显示的是ParseTree的子类RuleNode和TerminalNode以及它们所对应的子树根节点和叶子节点。RuleNode包含有方法如getChild()和getParent()等，但RuleNode并不专属于特定语法所有。为支持更好地访问在特定节点中的元素，ANTLR为每个规则生成一个RuleNode子类。下图为我们显示了赋值语句例子的子树根节点的特定类，它们是ProgContext，AssignContext和IntContext：



因为它们记录了我们知道的通过规则对短语识别的每件事，所以这些被称为上下文对象。每个上下文对象知道被识别短语的开始和结束记号以及提供对所有短语的元素的访问。例如，AssignContext提供方法ID()和INT()去访问标志符节点和表达式子树。

给出了具体类型的描述，我们可以手工写代码去执行树的深度优先遍历。当我们发现和完成节点时我们可以执行任何我们想要的动作。典型的操作是诸如计算结果，更新数据结构，或者生成输出。相比每次为每个应用写同样的树遍历样板代码，我们可以使用ANTLR自动生成的树遍历机制。

Visitor和Listener

ANTLR在它的运行库中为两种树遍历机制提供支持。默认情况下，ANTLR生成一个语法分析树Listener接口，在其中定义了回调方法，用于响应被内建的树遍历器触发的事件。

在Listener和Visitor机制之间最大的不同是：Listener方法被ANTLR提供的遍历器对象调用；而Visitor方法必须显式的调用visit方法遍历它们的子节点，在一个节点的子节点上如果忘记调用visit方法就意味着那些子树没有得到访问。

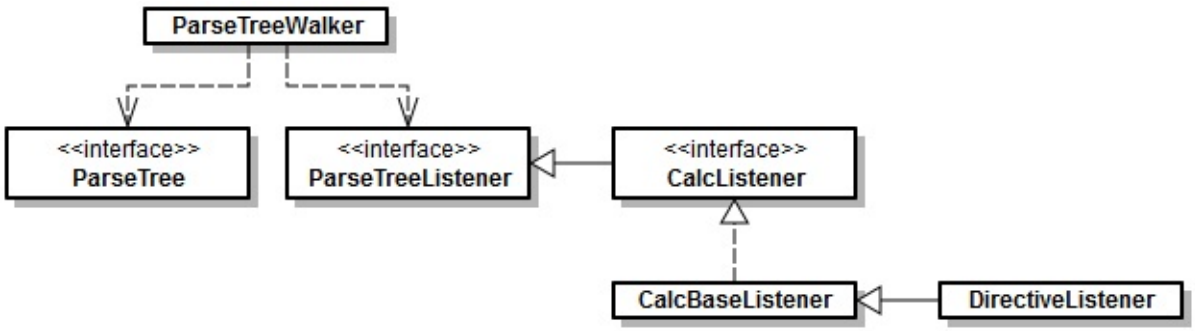
让我们首先从Listener开始。在我们了解Listener之后，我们也将看到ANTLR如何生成遵循Visitor设计模式的树遍历器。

语法分析树Listener

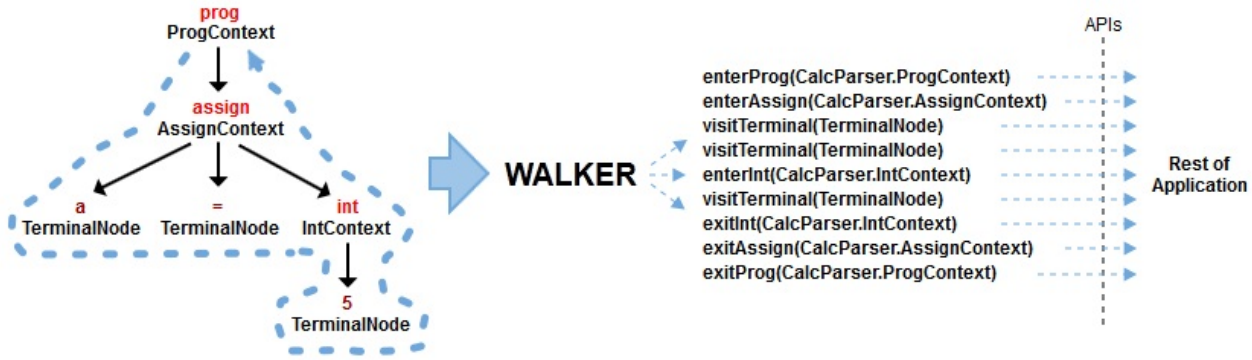
在Calc.java中有这样两行代码：

```
ParseTreeWalker walker = new ParseTreeWalker();
walker.walk(new DirectiveListener(), tree);
```

类ParseTreeWalker是ANTLR运行时提供的用于遍历语法分析树和触发Listener中回调方法的树遍历器。ANTLR工具根据Calc.g中的语法自动生成ParseTreeListener接口的子接口CalcListener和默认实现CalcBaseListener，其中含有针对语法中每个规则的enter和exit方法。DirectiveListener是我们编写的继承自CalcBaseListener的包含特定应用代码的实现，把它传递给树遍历器后，树遍历器在遍历语法分析树时就会触发DirectiveListener中的回调方法。



下图左边的语法分析树显示ParseTreeWalker执行了一次深度优先遍历，由粗虚线表示，箭头方向代表遍历方向。右边显示的是语法分析树的完整调用序列，它们由ParseTreeWalker触发调用。当树遍历器遇到规则assign的节点时，它触发enterAssign()并且给它传递AssignContext语法分析树节点。在树遍历器访问完assign节点的所有子节点后，它触发exitAssign()。

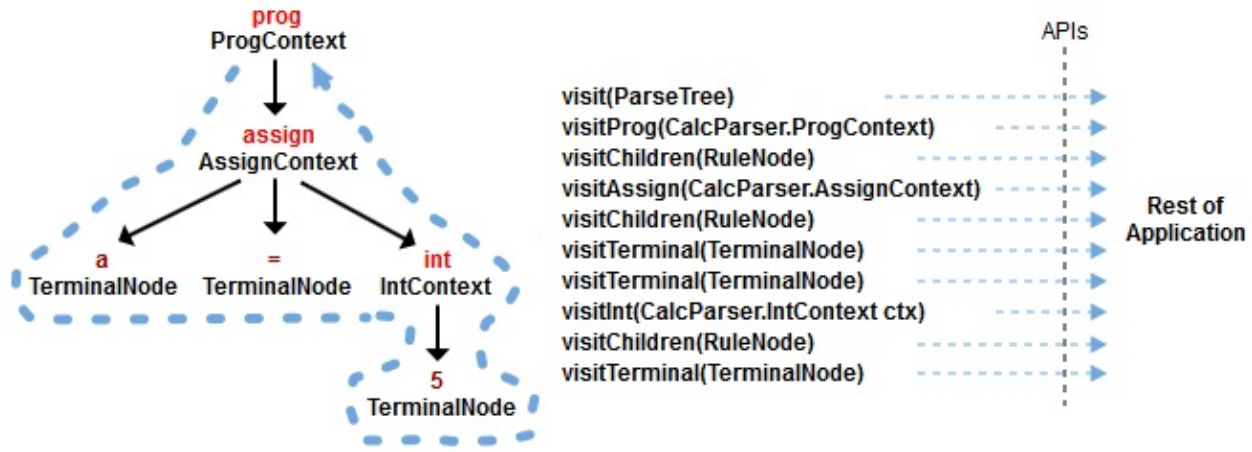


Listener机制的强大之处在于所有都是自动的。我们不必要写语法分析树遍历器，而且我们的Listener方法也不必要显式地访问它们的子节点。

语法分析树Visitor

有些情况下，我们实际想要控制的是遍历本身，在那里我们可以显式地调用visit方法去访问子树节点。选项-visitor告诉ANTLR工具从相应语法生成Visitor接口和默认实现，其中含有针对语法中每个规则的visit方法。

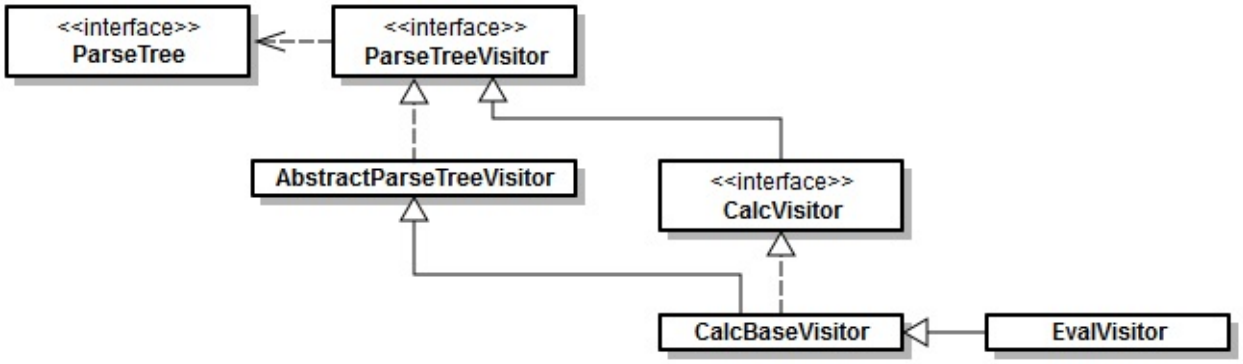
下图是我们熟悉的Visitor模式操作在语法分析树上。左边部分的粗虚线表示语法分析树的深度优先遍历，箭头方向代表遍历方向。右边部分指明Visitor中的方法调用序列。



下面是Calc.java中的两行代码：

```
EvalVisitor eval = new EvalVisitor();
// To start walking the parse tree
eval.visit(tree);
```

我们首先初始化自制的树遍历器EvalVisitor，然后调用visit()去访问整棵语法分析树。ANTLR运行时提供的Visitor支持代码会在看到根节点时调用visitProg()。在那里，visitProg()会把子树作为参数调用visit方法继续遍历，如此等等。



ANTLR自动生成的Visitor接口和默认实现可以让我们为Visitor方法编写自己的实现，让我们避免必须覆写接口中的每个方法，让我们仅仅聚焦在我们感兴趣的方法上。这种方法减少了我们学习ANTLR必须要花费的时间，让我们回到我们所熟悉的编程语言领域。

在语法中嵌入任意的操作

如果我们不想付出构建语法分析树的开销，或者想要在分析期间动态地计算值或把东西打印出来，那么可以通过在语法中嵌入任意代码实现。它的比较困难的，因为我们必须明白在语法分析器上的操作的影响，以及在哪里放置这些操作。

为了解释嵌入在语法中的操作，让我们先来看下文件rows.txt中的数据：

```
parrt   Terence Parr    101
tombu   Tom Burns       020
bke     Kevin Edgar     008
```

这些列是由TAB分隔的，每一行用一个换行结束。匹配这种类型的输入在语法上还是相当简单的。下面是此语法文件Rows.g的内容：

```
file : (row NL)+ ;    // NL is newline token: '\r'? '\n'
row  : STUFF+ ;
```

我们需要创建一个构造器以便我们能传递我们想要的列号（从1开始计数），所以我们需要在规则中添加一些操作来做这些事情：

```
grammar Rows;

@parser::members {    // add members to generated RowsParser
    int col;
    public RowsParser(TokenStream input, int col) {    // custom constructor
        this(input);
        this.col = col;
    }
}

file: (row NL)+ ;

row
locals [int i=0]
    : ( STUFF
        {
        $i++;
        if ( $i == col ) System.out.println($STUFF.text);
        }
      )+
    ;

TAB  :  '\t' -> skip ;    // match but don't pass to the parser
NL   :  '\r'? '\n' ;      // match and pass to the parser
STUFF:  ~[\t\r\n]+ ;      // match any chars except tab, newline
```

在上述语法中，操作是被花括号括起来的代码片段；members操作的代码将会被注入到生成的语法分析器类中的成员区；在规则row中的操作访问的$i是由locals子句定义的局部变量，该操作也用$STUFF.text获取最近匹配的STUFF记号的文本内容。STUFF词法规则匹配任何非TAB或换行的字符，这意味着在列中可以有空格字符。

现在，是时候去思考如何使用定制的构造器传递一个列号给语法分析器，并且告诉语法分析器不要构建语法分析树了：

```
public class Rows {

    public static void main(String[] args) throws Exception {
        ANTLRInputStream input = new ANTLRInputStream(System.in);
        RowsLexer lexer = new RowsLexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        int col = Integer.valueOf(args[0]);
        RowsParser parser = new RowsParser(tokens, col);    // pass column number!
        parser.setBuildParseTree(false);    // don't waste time bulding a tree
        parser.file();
    }
}
```

现在，让我们核实下我们的语法分析器能否正确匹配一些示例输入：

```
antlr -no-listener Rows.g  # don't need the listener
compile *.java
run Rows 1 < rows.txt
```

这时你会看到rows.txt文件的第1列内容被输出：

```
parrt
tombu
bke
```

如果将上面命令中的1换成2，你会看到rows.txt文件的第2列内容被输出；如果换成3，那么rows.txt文件的第3列内容将会被输出。

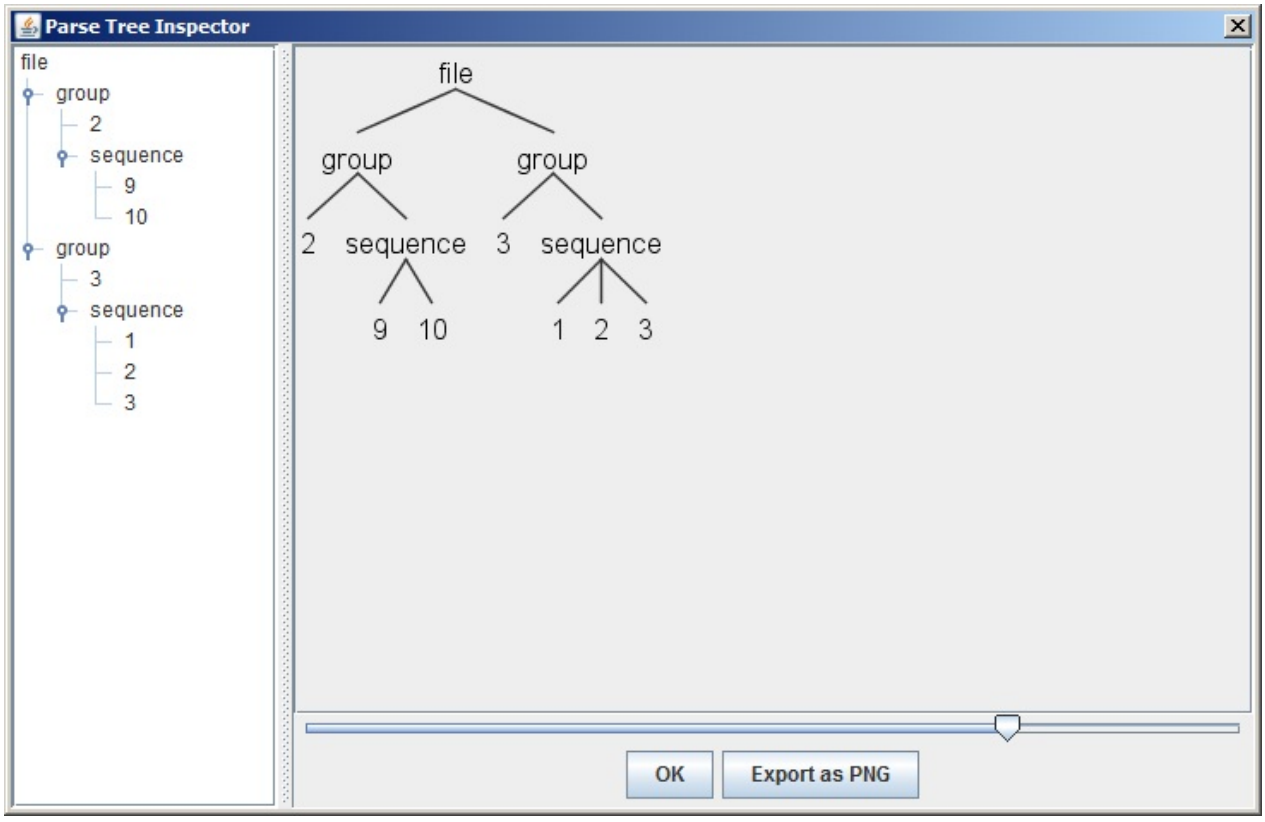
使用语义谓词改变语法分析

有一个读入整数序列的语法，它的玄机是由输入的部分指定有多少个整数组合在一起，所以我们必须等到运行时才能知道有多少整数被匹配。这里是示例输入文件idata.txt的内容：

```
2 9 10 3 1 2 3
```

第1个数字表示匹配后续两个数字9和10；紧跟10的数字3表示匹配接下来的三个数字。我们的目的是设计一个语法IData.g，把9和10组合在一起，把1、2和3组合在一起。在语法上执行以下命令后显示的语法分析树能够清楚地标识出整数的分组，就像下图显示的那样：

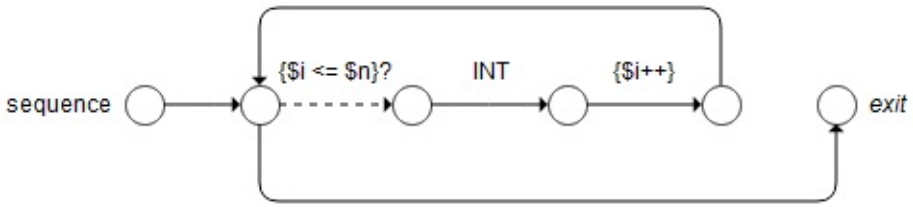
```
antlr -no-listener IData.g
compile *.java
grun IData file -gui idata.txt
```



要达成这个目标，以下语法中的关键是一个被称为语义谓词的布尔值操作：{$i <= $n}?。当谓词计算结果为true时，语法分析器匹配整数直到超过序列规则参数n要求的数量；当计算结果为false时，谓词让相关的选项从生成的语法分析器中“消失”。 在这个案例中，值为false的谓词让(...)*循环从规则序列里终止并返回。

```
grammar IData;

file : group+ ;

group: INT sequence[$INT.int] ;

sequence[int n]
locals [int i = 1;]
     : ( {$i<=$n}? INT {$i++;} )*    // match n integers
     ;

INT  : [0-9]+ ;  // match integers
WS   : [ \t\n\r]+ -> skip ;    // toss out all whitespace
```

被语法分析器使用的规则序列的内部语法表示看起来就像下图这样：



虚线表明谓词可以剪断那条路径，只给语法分析器留下一个选择：退出的路径。

虽然大部分时间我们不需要这样的微管理，但它至少让我们知道我们有这样的武器可以处理病理分析问题。

处理同一文件中的不同格式

目前我们看到的输入文件都只包含一种语言，但在实际应用中我们会遇到有些包含多种语言的常用文件格式。例如，Java的文档注释，XML文件等。这些环绕着模板表达式的文本需要不同的处理方式，它们被称为孤岛语言。

ANTLR有提供一个称之为“词法模型”的词法分析器特性，它让我们可以很容易地处理包含混合格式的文件。基本思路是：当词法分析器看到特殊的哨兵字符序列时，让它在模式之间来回切换。

XML是一个很好例子，它通常会在同一个文件中包含不同的词法结构。一个XML语法分析器会把除标签和实体引用（例如♥）之外的任何东西当作文本块。当词法分析器看到 `<` 时，它切换到“inside”模式；当它看到 `>` 或 `/>` 时，就切换回默认模式。以下语法展示了该特性是如何工作的：

```
lexer grammar XMLLexer;

// Default "mode": Everything OUTSIDE of a tag
OPEN        :   '<'                 -> pushMode(INSIDE) ;
COMMENT     :   '<!--' .*? '-->'    -> skip ;
EntityRef   :   '&' [a-z]+ ';' ;
TEXT        :   ~('<'|'&')+ ;    // match any 16 bit char minus < and &

// ----------------- Everything INSIDE of a tag ----------------------
mode INSIDE;

CLOSE       :   '>'                 -> popMode ;    // back to default mode
SLASH_CLOSE :   '/>'                -> popMode ;
EQUALS      :   '=' ;
STRING      :   '"' .*? '"' ;
SlashName   :   '/' Name ;
Name        :   ALPHA (ALPHA|DIGIT)* ;
S           :   [ \t\r\n]           -> skip ;

fragment
ALPHA       :   [a-zA-Z] ;

fragment
DIGIT       :   [0-9] ;
```

把上述语法保存为XMLLexer.g文件，然后使用包含以下内容的t.xml文件作为输入来测试它：

```
<tools>
  <tool name="ANTLR">A parser generator</tool>
</tools>
```

以下是构建和运行测试的命令：

```
antlr XMLLexer.g
compile *.java
grun XML tokens -tokens t.xml
```

这里是输出的内容：

```
[@0,0:0='<',<1>,1:0]
[@1,1:5='tools',<10>,1:1]
[@2,6:6='>',<5>,1:6]
[@3,7:10='\r\n  ',<4>,1:7]
[@4,11:11='<',<1>,2:2]
[@5,12:15='tool',<10>,2:3]
[@6,17:20='name',<10>,2:8]
[@7,21:21='=',<7>,2:12]
[@8,22:28='"ANTLR"',<8>,2:13]
[@9,29:29='>',<5>,2:20]
[@10,30:47='A parser generator',<4>,2:21]
[@11,48:48='<',<1>,2:39]
[@12,49:53='/tool',<9>,2:40]
[@13,54:54='>',<5>,2:45]
[@14,55:56='\r\n',<4>,2:46]
[@15,57:57='<',<1>,3:0]
[@16,58:63='/tools',<9>,3:1]
[@17,64:64='>',<5>,3:7]
[@18,65:66='\r\n',<4>,3:8]
[@19,67:66='<EOF>',<-1>,4:0]
```

上面输出的每一行代表一个记号，包含记号索引、开始和结束字符、记号文本、记号类型，最后的行和字符位置则告诉我们词法分析器如何标记化输入。

在命令行中，XML tokens序列处通常是一个语法名字后面跟着开始规则，但在这里，我们使用语法名字后面跟着特殊的规则名字tokens来告诉TestRig应该运行词法分析器而不是语法分析器。接着使用选项-tokens打印出匹配的记号列表。

重写输入流

现在准备要构建一个工具，用来把前面idata.txt里的数据按group分行显示，就像这样：

```
2 9 10
3 1 2 3
```

我们可以借助语法分析树的Listener机制来对词法分析结束后生成的记号流进行改写，我们不需要实现每一个Listener接口方法，只需要在捕获到group的时候把换行符插到它末尾就行。实现改写的代码如下所示：

```
import org.antlr.v4.runtime.TokenStream;
import org.antlr.v4.runtime.TokenStreamRewriter;

public class RewriteListener extends IDataBaseListener {
    TokenStreamRewriter rewriter;

    public RewriteListener(TokenStream tokens) {
        rewriter = new TokenStreamRewriter(tokens);
    }

    @Override
    public void enterGroup(IDataParser.GroupContext ctx) {
        rewriter.insertAfter(ctx.stop, '\n');
    }
}
```

接着就是写一个小程序来调用我们上面的改写类：

```
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;
import java.io.FileInputStream;
import java.io.InputStream;

public class IData {

    public static void main(String[] args) throws Exception {
        InputStream is = args.length > 0 ? new FileInputStream(args[0]) : System.in;

        ANTLRInputStream input = new ANTLRInputStream(is);
        IDataLexer lexer = new IDataLexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        IDataParser parser = new IDataParser(tokens);
        ParseTree tree = parser.file();

        RewriteListener listener = new RewriteListener(tokens);

        System.out.println("Before Rewriting");
        System.out.println(listener.rewriter.getText());

        ParseTreeWalker walker = new ParseTreeWalker();
        walker.walk(listener, tree);

        System.out.println("After Rewriting");
        System.out.println(listener.rewriter.getText());
    }
}
```

这里的关键是TokenStreamRewriter对象知道如何在不修改流的情况下提供一个记号流的修改过的视图。它把所有的操作方法当作指令并把它们排进队列，等到在遍历记号流把它作为文本渲染回去的时候延迟执行。每次我们调用getText()时rewriter就会执行那些指令。

最后就是构建和测试应用：

```
antlr IData.g
compile *.java
run IData idata.txt
```

以下是输出结果：

```
Before Rewriting
29103123
After Rewriting
2910
3123
```

仅用几行代码，我们就能够没有任何烦恼地对某些内容做轻微的调整。这种策略对于源代码检测或重构这类一般性的问题是非常有效的。TokenStreamRewriter是一个非常强大且有效的操作记号流的方法。

发送记号到不同的通道

对于大多数语法，注释和空格都是语法分析器可以忽略的东西。如果我们不想让注释和空格在语法中到处都是，那么就需要让词法分析器把它们扔掉。不幸的是，这意味着任何后续处理步骤都不能再访问注释和空格。安全地忽略掉注释和空格的方法是把这些发送给语法分析器的记号放到一个“隐藏通道”中，语法分析器仅需要调协到单个通道即可。我们可以把任何我们想要的东西传递到其它通道中。这里是如何实现的语法：

```
COMMENT
    : '/*' .*? '*/' -> channel(HIDDEN)    // match anything between /* and */
    ;

WS  : [ \r\t\n]+    -> channel(HIDDEN)
    ;
```

就像我们前面讨论过的 `-> skip` 那样， `-> channel(HIDDEN)` 也是一个的词法分析器指令。在这里，它设置那些记号的通道号码以便这些记号可以被语法分析器忽略。记号流仍然维护着原始的记号序列，但在喂食给语法分析器时则略过离线通道中的记号。

算术表达式语言

了解ANTLR最好的方法就是实例。构建一个简单的计算器是个不错的主意。为了使它容易理解且保持简单，我们将只允许基本的算术运算符（加、减、乘、除）、括号表达式、整数和变量。

```
grammar Calc;

prog
    : stat+
    ;

stat
    : expr
    | ID '=' expr
    ;

expr
    : expr ('*'|'/') expr
    | expr ('+'|'-') expr
    | INT
    | ID
    | '(' expr ')'
    ;

ID  : [a-zA-Z]+ ;

INT : [0-9]+ ;

WS  : [ \t\r\n]+ -> skip ;    // toss out whitespace
```

在上述的语法中，程序是由空格（换行符也被当作空格）终止的语句序列，语句可以是表达式或者赋值。那些以小写字母开头的像stat和expr是语法规则；由大写字母开头的诸如ID和INT为词法规则，用于识别标志符和整数这样的记号。我们用“|”分隔规则的选项，我们也可以用“()”把符号分组成子规则。例如，子规则 `('*'|'/')` 匹配乘法符号或者除法符号。

ANTLR v4最重要的新特性是它有能力处理（大多数类型的）左递归规则。例如，规则expr前两个选项就在左边缘递归地调用了expr自身。这种指定算术表达式表示法的方法比那些典型的自顶向下语法分析器策略更容易。当然，在这种策略下，我们需要定义多个规则，每个运算符优先级一个规则。

记号定义的表示法对那些有正则表达式经验的应该很熟悉。唯一不寻常的是在WS规则上的 `-> skip` 指令，它告诉词法分析器去匹配但丢弃空格，不要把它们放到记号流中，这样在语法分析树上空格就不会有对应的记号。（每个可能的输入字符都必须被至少一个词法规则匹

配。）我们通过使用形式化的ANTLR表示法避免捆绑语法到某个特定的目标语言，而不是在语法中插入任意代码片段来告诉词法分析器去忽略。

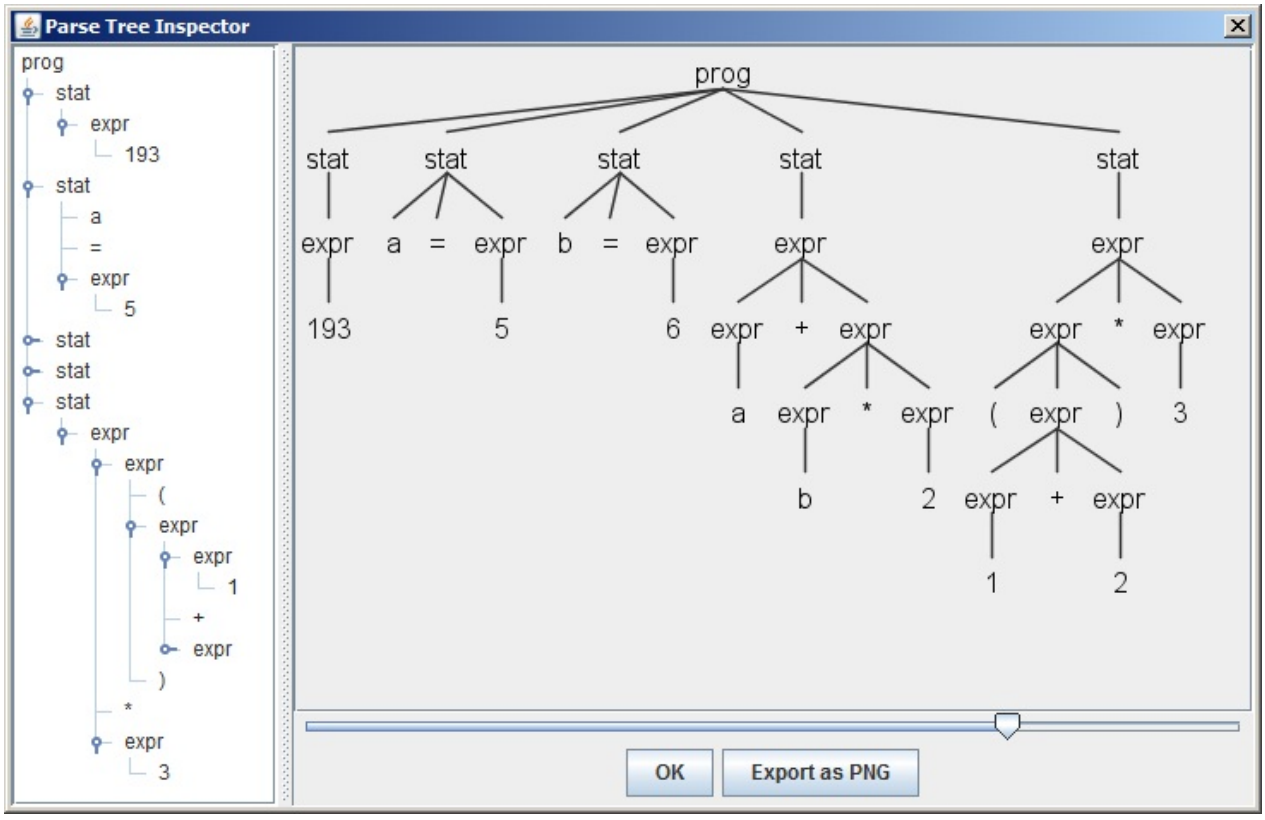
这里是一些用来评估所有语法特性的测试序列：

```
193
a=5
b=6
a+b*2
(1+2)*3
```

把它们放入文件calc.txt中，然后执行以下命令：

```
antlr Calc.g
compile *.java
grun Calc prog -gui calc.txt
```

TestRig会弹出一个显示语法分析树的窗口：



使用Visitor模式计算结果

为了给前面的算术表达式语法分析器计算出结果，我们还需要做些其它的事情。

ANTLR v4鼓励我们保持语法的整洁，使用语法分析树Visitor和其它遍历器来实现语言应用。不过在接触这些之前，我们需要对语法做些修改。

首先，我们需要用标签标明规则的选项，标签可以是和规则名没有冲突的任意标志符。如果选项上没有标签，ANTLR只会为每个规则生成一个visit方法。

在本例中，我们希望为每个选项生成一个不同的visit方法，以便每种输入短语都能得到不同的事件。在新的语法中，标签出现在选项的右边缘，且以“#”符号开头：

```
stat
    : expr                  # printExpr
    | ID '=' expr           # assign
    ;

expr
    : expr op=(MUL|DIV) expr # MulDiv
    | expr op=(ADD|SUB) expr # AddSub
    | INT                   # int
    | ID                    # id
    | '(' expr ')'          # parens
    ;
```

接下来，让我们为运算符字面量定义一些记号名字，以便以后可以在visit方法中引用作为Java常量的它们：

```
MUL : '*' ;

DIV : '/' ;

ADD : '+' ;

SUB : '-' ;
```

现在，我们有了一个增强型的语法。接下来要做的事情是实现一个EvalVisitor类，它通过遍历表达式语法分析树计算和返回值。

执行下面的命令，让ANTLR生成Visitor接口和它的默认实现，其中-no-listener参数是告诉ANTLR不再生成Listener相关的代码：

```
antlr -no-listener -visitor Calc.g
```

所有被标签标明的选项在生成的Visitor接口中都定义了一个visit方法：

```
public interface CalcVisitor<T> extends ParseTreeVisitor<T> {
    T visitProg(CalcParser.ProgContext ctx);
    T visitPrintExpr(CalcParser.PrintExprContext ctx);
    T visitAssign(CalcParser.AssignContext ctx);
    ...
}
```

接口定义使用的是Java泛型，visit方法的返回值为参数化类型，这允许我们根据表达式计算返回值的类型去设定实现的泛型参数。因为表达式的计算结果是整型，所以我们的EvalVisitor应该继承 `CalcBaseVisitor<Integer>` 类。为计算语法分析树的每个节点，我们需要覆写与语句和表达式选项相关的方法。这里是全部的代码：

```
public class EvalVisitor extends CalcBaseVisitor<Integer> {
    /** "memory" for our calculator; variable/value pairs go here */
    Map<String, Integer> memory = new HashMap<String, Integer>();

    /** ID '=' expr */
    @Override
    public Integer visitAssign(CalcParser.AssignContext ctx) {
        String id = ctx.ID().getText();  // id is left-hand side of '='
        int value = visit(ctx.expr());   // compute value of expression on right
        memory.put(id, value);           // store it in our memory
        return value;
    }

    /** expr */
    @Override
    public Integer visitPrintExpr(CalcParser.PrintExprContext ctx) {
        Integer value = visit(ctx.expr()); // evaluate the expr child
        System.out.println(value);         // print the result
        return 0;                          // return dummy value
    }

    /** INT */
    @Override
    public Integer visitInt(CalcParser.IntContext ctx) {
        return Integer.valueOf(ctx.INT().getText());
    }

    /** ID */
    @Override
    public Integer visitId(CalcParser.IdContext ctx) {
        String id = ctx.ID().getText();
        if ( memory.containsKey(id) ) return memory.get(id);
```

```
        return 0;
    }

    /** expr op=('*'|'/') expr */
    @Override
    public Integer visitMulDiv(CalcParser.MulDivContext ctx) {
        int left = visit(ctx.expr(0));  // get value of left subexpression
        int right = visit(ctx.expr(1)); // get value of right subexpression
        if ( ctx.op.getType() == CalcParser.MUL ) return left * right;
        return left / right; // must be DIV
    }

    /** expr op=('+'|'-') expr */
    @Override
    public Integer visitAddSub(CalcParser.AddSubContext ctx) {
        int left = visit(ctx.expr(0));  // get value of left subexpression
        int right = visit(ctx.expr(1)); // get value of right subexpression
        if ( ctx.op.getType() == CalcParser.ADD ) return left + right;
        return left - right; // must be SUB
    }

    /** '(' expr ')' */
    @Override
    public Integer visitParens(CalcParser.ParensContext ctx) {
        return visit(ctx.expr()); // return child expr's value
    }
}
```

以前开发和测试语法都是使用的TestRig，这次我们试着编写计算器的主程序来启动代码：

```
public class Calc {
    CharStream input = CharStreams.fromFileName(fileName);
    public static void main(String[] args) throws Exception {
        InputStream is = args.length > 0 ? new FileInputStream(args[0]) : System.in;

        ANTLRInputStream input = new ANTLRInputStream(is);
        CalcLexer lexer = new CalcLexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        CalcParser parser = new CalcParser(tokens);
        ParseTree tree = parser.prog();

        EvalVisitor eval = new EvalVisitor();
        // 开始遍历语法分析树
        eval.visit(tree);

        System.out.println(tree.toStringTree(parser));
    }
}
```

antlr 4.7版本后改用

创建一个运行主程序的脚本：

```
#!/bin/sh
java -cp .:./antlr-4.5.1-complete.jar:$CLASSPATH $*
```

把它保存为run.sh后，执行以下命令：

```
compile *.java
run Calc calc.txt
```

然后你就会看到文本形式的语法分析树以及计算结果：

```
193
17
9
(prog (stat (expr 193)) (stat a = (expr 5)) (stat b = (expr 6))
 (stat (expr (expr a) + (expr (expr b) * (expr 2)))) (stat (expr
 (expr ( (expr (expr 1) + (expr 2)) )) * (expr 3))))
```

使用Listener模式计算结果

在上一节中的计算器是以解释的方式执行的，现在我们想要把它转换成以编译的方式执行。编译执行和解释执行相比，需要依赖于特定的目标机器。在这里我们假设有一台这样的机器，它用堆栈进行运算，支持如下表所示的几种指令：

指令	说明	操作数个数	用途
LDV	Load Variable	1	变量入栈
LDC	Load Constant	1	常量入栈
STR	Store Value	1	栈顶一个元素存入指定变量
ADD	Add	0	栈顶两个元素出栈，求和后入栈
SUB	Subtract	0	栈顶两个元素出栈，求差后入栈
MUL	Multiply	0	栈顶两个元素出栈，求积后入栈
DIV	Divide	0	栈顶两个元素出栈，求商后入栈
RET	Return	0	栈顶一个元素出栈，计算结束

做这个最简单的方法是使用ANTLR的语法分析树Listener机制实现DirectiveListener类，然后它通过监听来自树遍历器触发的事件，输出对应的机器指令。

Listener机制的优势是我们不必要自己去做任何树遍历，甚至我们不必要知道遍历语法分析树的运行时如何调用我们的方法，我们只要知道我们的DirectiveListener类得到通知，在与语法规则匹配的短语开始和结束时。这种方法减少了我们学习ANTLR必须要花费的时间，让我们回到我们所熟悉的编程语言领域。

这里不需要创建新的语法规则，还是继续沿用前文Calc.g所包含的语法，标签也要保留：

```
grammar Calc;

prog
    : stat+
    ;

stat
    : expr                   # printExpr
    | ID '=' expr            # assign
    ;

expr
    : expr op=(MUL|DIV) expr # MulDiv
    | expr op=(ADD|SUB) expr # AddSub
    | INT                    # int
    | ID                     # id
    | '(' expr ')'           # parens
    ;

MUL : '*' ;

DIV : '/' ;

ADD : '+' ;

SUB : '-' ;

ID  : [a-zA-Z]+ ;

INT : [0-9]+ ;

WS  : [ \t\r\n]+ -> skip ;    // toss out whitespace
```

然后，我们可以运行ANTLR工具：

```
antlr Calc.g
```

它会生成后缀名为tokens和java的六个文件：

```
Calc.tokens         CaclLexer.java          CalcParser.java
CalcLexer.tokens    CalcBaseListener.java   CalcListener.java
```

正如这里我们看到的，ANTLR会为我们自动生成Listener基础设施。其中CalcListener是语法和Listener对象之间的关键接口，描述我们可以实现的回调方法：

```
public interface CalcListener extends ParseTreeListener {
    void enterProg(CalcParser.ProgContext ctx);
    void exitProg(CalcParser.ProgContext ctx);
    void enterPrintExpr(CalcParser.PrintExprContext ctx);
    ...
}
```

CalcBaseListener则是ANTLR生成的一组空的默认实现。ANTLR内建的树遍历器会去触发在Listener中像enterProg()和exitProg()这样的一串回调方法，如同它对语法分析树执行了一次深度优先遍历。为响应树遍历器触发的事件，我们的DirectiveListener需要继承CalcBaseListener并实现一些方法。我们不需要实现全部的接口方法，我们也不需要去覆写每个enter和exit方法，我们只需要去覆写那些我们感兴趣的回调方法。

在本例中，我们需要通过覆写6个方法对6个事件——当树遍历器exit那些有标签的选项时触发——作出响应。我们的基本策略是当这些事件发生时打印出已转换的指令。以下是完整的实现代码：

```
public class DirectiveListener extends CalcBaseListener {

    @Override
    public void exitPrintExpr(CalcParser.PrintExprContext ctx) {
        System.out.println("RET\n");
    }

    @Override
    public void exitAssign(CalcParser.AssignContext ctx) {
        String id = ctx.ID().getText();
        System.out.println("STR " + id);
    }

    @Override
    public void exitMulDiv(CalcParser.MulDivContext ctx) {
        if (ctx.op.getType() == CalcParser.MUL) {
            System.out.println("MUL");
        } else {
            System.out.println("DIV");
        }
    }

    @Override
    public void exitAddSub(CalcParser.AddSubContext ctx) {
        if (ctx.op.getType() == CalcParser.ADD) {
            System.out.println("ADD");
        } else {
            System.out.println("SUB");
        }
    }

    @Override
    public void exitId(CalcParser.IdContext ctx) {
        System.out.println("LDV " + ctx.ID().getText());
    }

    @Override
    public void exitInt(CalcParser.IntContext ctx) {
        System.out.println("LDC " + ctx.INT().getText());
    }
}
```

为了让它运行起来，余下我们唯一需要做的事是创建一个主程序去调用它：

```
public class Calc {

    public static void main(String[] args) throws Exception {
        InputStream is = args.length > 0 ? new FileInputStream(args[0]) : System.in;

        ANTLRInputStream input = new ANTLRInputStream(is);
        CalcLexer lexer = new CalcLexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        CalcParser parser = new CalcParser(tokens);
        ParseTree tree = parser.prog();

        ParseTreeWalker walker = new ParseTreeWalker();
        walker.walk(new DirectiveListener(), tree);

        // print LISP-style tree
        System.out.println(tree.toStringTree(parser));
    }
}
```

这个程序和前文Calc.java中的代码极度相似，区别只在12-13行。这两行代码负责创建树遍历器，然后让树遍历器去遍历那颗从语法分析器返回的语法分析树，当树遍历器遍历时，它就会触发调用到我们的DirectiveListener中实现的方法。此外，通过传入一个不同的Listener实现我们能简单地生成完全不同的输出。Listener机制有效地隔离了语法和语言应用，使语法可以被其它应用再次使用。

现在一切完备，让我们尝试着去编译和运行它吧！下面是完整的命令序列：

```
compile *.java
run Calc calc.txt
```

编译的输出结果如下所示：

```
LDC 19
RET

LDC 5
STR a
LDC 6
STR b
LDV a
LDV b
LDC 2
MUL
ADD
RET

LDC 1
LDC 2
ADD
LDC 3
MUL
RET
```

语法设计

在聚焦到具体的语法规则内部结构之前，我们要先讨论下语法的整体剖析以及如何形成一套初始的语法骨架。

文法文件通常是由一个命名文法的头和一系列可以彼此调用的规则组成。就像下面的那样：

```
grammar MyG;
rule1 : «stuff» ;
rule2 : «more stuff» ;
...
```

设计语法就是要搞清楚«stuff»是什么？哪个规则是开始规则。这要求我们需要知道给定语言的一系列代表性的输入例子。当然，从语言参考手册或者其它语法分析器生成器格式而来的语法也是有帮助的。

正确设计语法的方法是借鉴功能分解或者自顶向下的设计，从粗粒度级别到细粒度级别逐步定义语言结构并把它们编码为语法规则。所以，我们的第一个任务就是找到粗粒度语言结构的名字，同时它也是开始规则。在英语中我们使用sentence，对于XML文件来说它则是document。

设计开始规则的内容是用英语伪代码描述整个输入格式的问题。例如，“a comma-separated-value (CSV) file is a sequence of rows terminated by newlines.”这段文字，在is a左边的至关重要的单词file是规则名字，在is a右边的所有内容则成为在规则定义右则的«stuff»的内容：

```
file : «sequence of rows that are terminated by newlines» ;
```

然后我们通过描述在开始规则右侧被确定的元素来进行下一个粒度级别的设计。在规则右侧的名词通常是对记号或尚未定义的规则的引用，这些记号是那些我们在正常情况下视为单词、标点符号、运算符的元素。就像单词是英语句子中的原子成分那样，记号在语法规则中也是如此。规则引用则涉及到像row那样需要被分解为更详细部分的其它语言结构。

进入细节的另外一层，我们可以说row是一系列被逗号分隔的field，而field则是一个数字或字符串。就像以下所示：

```
row   : «sequence of fields separated by commas» ;
field : «number or string» ;
```

当没有规则需要再定义时，我们就得到了语法的一个粗略的草图。

如果有其它格式的语法作为参考的话设计语法会容易的多，但小心不要盲目地遵循它，否则你会误入歧途的。非ANTLR格式的语法只是让你知道别人是如何决定分解语言中的短语的，它最大的作用就是可以给我们一份规则名称的列表用作参考。

不推荐从参考手册上复制粘贴语法到ANTLR，然后再通过细微的调整让它工作。把它当作一套指南而不是一段代码是更好的办法。为了清晰地描述语法，参考手册通常是相当松散的。这意味着语法能识别大量不在语言中的句子，或者语法可能不够明确，可以用多种方法匹配相同的输入序列。例如，语法可能会说表达式可以调用一个构建器或者访问一个函数，问题是像T(i)这样的输入可以同时匹配两者。理想情况下，在语法中是不能有这样的二义性的，每个输入句子我们只需要一种解释。

在另一个极端，参考手册中的语法有时过于明确地说明了规则。有些约束是需要在分析完输入后实施的，而不是试图对语法结构实施约束。例如，W3C XML语法就显式地指定标签中什么地方必须要有空格以及什么地方的空格可以省略。但事实是我们可以简单地让词法分析器在把记号发送给语法分析器之前去除空格，不需要在语法中到处测试它。

规格还说<?xml ...>标签可以有两个附加属性encoding和standalone。我们需要知道约束，但它是很容易去允许任何属性名字，然后在语法分析后检查语法分析树，以确保所有这些限制都满足的。 归根结底，XML只是嵌在文本中的一对标签，因此它的语法结构是相当直白的。唯一的挑战是如何分别对待什么在标签内以及什么在标签外。

识别语法规则并用伪代码表示它们的右侧部分最初是个挑战，但当你为更多的语言构建语法后它会变得越来越容易。一旦我们有了伪代码，我们就需要把它转换成ANTLR表示法，以便能得到一个可工作的语法。

常用语言模式

现在，我们已经有了一个自顶向下的草拟出语法的通用策略，下面我们要专注于一些常用的语言模式。尽管在过去几十年里有大量的语言被发明，但仍然只有较少的基本语言模式需要被处理。这是因为人们趋向于设计遵循自然语言约束的语言，语言也会因为设计者遵循数学上的常用表示法而趋向于相似。甚至在词法级别，语言趋向于重用一些相同的结构，例如标志符、整数、字符串等。这些单词顺序和依赖的约束来源于自然语言，并逐渐演化成为四种抽象的语言模式：

模式：序列

这是像数组初始值设定项中的值那样的一系列元素，也是在计算机语言中最常见的结构。例如，下面是登录到POP服务器时的序列：

```
USER parrt
PASS secret
RETR 1
```

这些命令本身也是序列。大部分命令是一个关键词（保留标志符，例如USER和RETR）跟随一个运算元再跟随一个换行符。为了在语法中指定此类序列，我们可以按照顺序简单地列出各个元素。以下是检索命令的序列（其中INT表示整数记号类型）：

```
retr : 'RETR' INT '\n' ;
```

我们可以给RETR序列打上retr规则的标签，这样在语法的其它地方，我们就能使用规则名字作为简写来引用RETR序列。

对于任意长度的序列像矢量[1 2 3]这样的简单整数列表，虽然它是一个有限序列，但我们不可能通过像INT INT INT ...这样的规则片段来列出所有可能的整数列表。为了编码这样的一个或者多个元素，我们使用“+”子规则运算符。例如，{INT}+表示任意长度的整数序列，或者使用简写INT+也可以。至于可以为空的列表，我们则使用零个或者多个运算符“*”。

这种模式的变体有带终结符的序列和带分隔符的序列，CSV文件就很好地示范了这两者。

```
file : (row '\n')* ;           // 带一个“\n”终结符的序列
row  : field (',' field)* ;    // 带一个“,”分隔符的序列
field: INT ;                   // 假设字段只是整数
```

规则file使用带终结符模式的列表去匹配零个或者多个row '\n'序列，记号“\n”终结序列的每个元素。规则row使用带分隔符模式的列表去匹配一个field后面有零个或者多个',' field序列，记号“,”分隔各个字段。

最后，还有个特殊类型的零个或者一个序列，用“?”指定。可以使用它去表达可选的构造体。

模式：选择

这是一个在多个可供替代的短语之间的选择，比如在编程语言中不同种类的语句。为了在语言中表示选择的这个概念，我们使用“|”作为ANTLR中的“or”运算符去分隔被称为“选项”的语法选择。

回到CVS语法，我们可以通过整数或者字符串的选择让规则field变得更灵活。

```
field: INT | STRING ;
```

任何时候，如果你发现正在说“语言结构x可以是这个或者那个”，那么你就可以确定应该使用选择模式，在规则x中使用“|”。

模式：记号依赖

记号依赖表示一个记号的存在需要在短语的其它地方有它的对等物的存在，比如匹配的左右括号。前面我们曾经使用INT+去表达在矢量[1 2 3]中的整数非空序列。为指定周围有方括号的矢量，我们需要一种方法去表达记号中的依赖。如果我们在句子中看到一个符号，那么我们必须在句子的其它地方找到它的对等物。为表达这种语法，我们必须使用同时指定对等符号的序列，它们通常包围或分组着其它元素。在这个案例中，我们这样指定矢量：

```
vector : '[' INT+ ']' ;    // [1], [1 2], [1 2 3], ...
```

扫视任何有效的代码，你会看到必须成对出现的各种分组符号：(...)，[...]，{...}。但是要牢记，依赖符号并不是必须配对的，类C语言都有的a ? b : c三元运算符就指定了当看到“?”符号时需要在接下来的短语中看到“:”符号。

模式：嵌套短语

嵌套短语有一个自相似的语言结构，它的子短语也遵循相同的结构。表达式是典型的自相似语言结构，由被运算符分隔的嵌套子表达式组成。类似地，while的代码块是嵌套在外部代码块内的一个代码块。我们在语法中使用递归规则表达自相似的语言结构。因此，如果规则的伪代码引用它自身，我们将需要一个递归的自引用规则。

让我们来看下代码块的嵌套是如何工作的。while语句是关键词while跟随一个在括号中的条件表达式再跟随一个语句。我们也可以把多条语句包裹在花括号里当作一个单块语句。表达语法如下所示：

```
stat: 'while' '(' expr ')' stat    // 匹配WHILE语句
    | '{' stat* '}'                // 匹配在括号中的语句块
    ;
```

这里的stat可以是单条语句或者被花括号括起来的一组语句。规则stat是直接递归的，因为它在两个选项中直接引用它自身。如果我们把第二个选项移到它自己的规则中，规则stat和block将是双向间接递归的。语法如下所示：

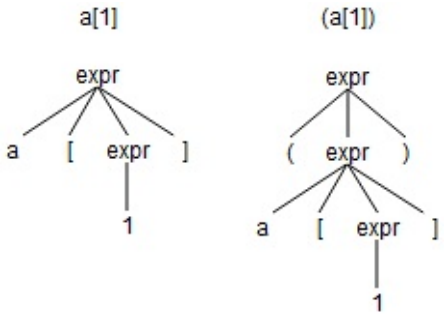
```
stat: 'while' '(' expr ')' stat    // 匹配WHILE语句
    | '{' stat* '}'                // 匹配语句块
    ;
block: '{' stat* '}' ;             // 匹配在括号中的语句块
```

看下面仅有3类表达式（索引数组引用、括号表达式和整数）的简单语言的语法：

```
expr: ID '[' expr ']'    // a[1], a[b[1]], a[(2*b[1])]
    | '(' expr ')'       // (1), (a[1]), (((1))), (2*a[1])
    | INT                // 1, 94117
    ;
```

注意递归是如何自然地发生的。数组索引表达式的索引组件是表达式本身，因此我们只需要在选项中引用expr即可。

下图是关于两个例子输入的语法分析树：



分析树中的内部树节点是规则引用，叶子是记号引用。从树根到任何节点的路径表示元素的规则调用栈（或者ANTLR生成的递归下降语法分析器调用栈）。路径代表递归嵌套的子树有多个相同规则的引用。规则节点是其下方子树的标签。根节点是expr，所以整棵树是一个表达式。在1之前的那棵expr子树会把整数当作一个表达式。

为实现这些模式，我们只需要由选项、记号引用、规则引用组成的语法规则即可。我们还可以把这些元素组成子规则，子规则是裹在括号内的行内规则。我们也可以将子规则标记为“?”或“*”或“+”循环去识别被包围的语法片段多次。

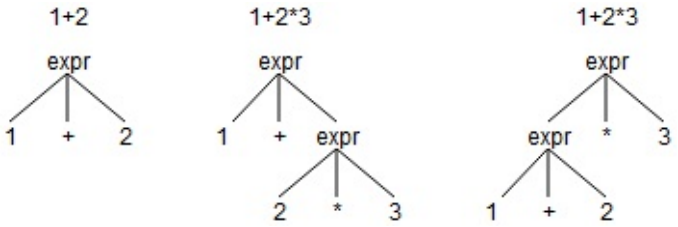
优先级，左递归以及相关性

用自顶向下的语法指定和通过手工的递归下降语法分析器识别表达式一直是个麻烦。首先是因为大部分自然语法是模糊的，其次是因为大部分自然语法规格使用一种被称为左递归的特殊类型递归。所以自顶向下的语法和语法分析器不能处理传统形式上的左递归。

为了阐明这个问题，设想一个算术表达式语言，它只有乘法和加法运算符以及整数。表达式是自相似的。也就是说，一个乘法表达式是由“*”运算符连接的两个子表达式。同样的，一个加法表达式是由“+”运算符连接的两个子表达式。我们也可以把整数看作表达式。整个语法规则看起来就像以下显示的那样：

```
expr : expr '*' expr     // 匹配由“*”运算符连接的子表达式
     | expr '+' expr     // 匹配由“+”运算符连接的子表达式
     | INT               // 匹配简单整数
     ;
```

问题是上述规则对于某些输入短语来说是模棱两可的。换句话说，这个规则能用多种方法匹配单个输入流。对于简单的整数和像1+2和1*2这样的单运算符表达式是没问题的，因为只有一种方法能去匹配它们。例如，规则可以仅用第二个选项匹配1+2。就像下图左边所示的那样：



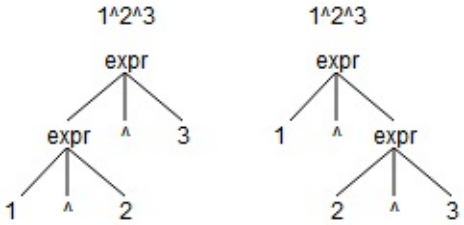
问题是指定的规则可以像中间和右边语法分析树描绘的那样用两种方法解释1+2*3这样的输入。两者的解释是不同的，因为中间的树说加1到2乘3的结果上，而右边的树说3乘以1加2的结果。这是一个运算符优先级的问题，但常规语法根本没有指定优先级的方法。大部分语法工具使用额外的符号来指定运算符优先级。

与之相反的是，ANTLR解决二义性有利于首先给出的选项，隐式地允许我们指定运算符优先级。规则expr有一个乘法选项在加法选项之前，因此，ANTLR解决1+2*3的运算符二义性有利于乘法。

默认情况下，ANTLR从左到右结合运算符，然而某些像指数群这样的运算符则是从右到左。因此，我们必须使用选项assoc手动指定运算符记号上的相关性。这里是一个能正确地把输入 `2^3^4` 解释成 `2^(3^4)` 的表达式规则：

```
expr : expr '^'<assoc=right> expr    // 运算符是右结合的
     | INT
     ;
```

下图中的语法分析树阐明了运算符左右结合版本的不同。右边的语法分析树是惯常的解释：



为了把所有这三个运算符合并成一条规则，我们把指数表达式选项放在其它表达式选项之前，因为它的运算符比乘法和加法都有更高的优先级。合并后的语法如下所示：

```
expr : expr '^'<assoc=right> expr    // 运算符是右结合的
     | expr '*' expr                 // 匹配由“*”运算符连接的子表达式
     | expr '+' expr                 // 匹配由“+”运算符连接的子表达式
     | INT                           // 匹配简单整数
     ;
```

不像其它常规的语法分析器生成器那样，ANTLR v4是可以处理直接左递归的。左递归规则是指直接或者间接调用在选项左边缘的自身的规则。规则expr是直接左递归的，因为除INT选项外的其它所有选项都开始于规则expr自身的引用。如果规则expr的引用处在某些选项的右边缘，那么它就是右递归的。虽然ANTLR v4可以处理直接左递归，但它不能处理间接左递归。这意味着我们不能把expr因子化为语法等效规则。

```
expr : expo ;    // 通过expo左递归地间接调用expr
expo : expr '^'<assoc=right> expr ;
```

ANTLR v4可以简化直接左递归的表达式规则的工作。这种新的机制不仅更有效率，而且表达式规则也更小和更容易理解。

常用词法结构

编程语言在词法上看起来惊人地相似，无论是函数式、过程式、声明式还是面向对象语言，看起来几乎都是一样的。这很棒，因为我们只需要学习一次如何描述标志符和整数，没有太大的变化，就可以把它们应用到大多数编程语言上。正如语法分析器以及词法分析器使用规则去描述各种语言构造体一样，我们要使用基本相同的表示法。唯一的区别是语法分析器识别在记号流中的语法结构，而词法分析器识别在字符流中的语法结构。

因为词法分析和语法分析有相似的结构，ANTLR允许我们把两者合并在单个语法文件中。但是因为词法分析和语法分析是语言识别的两个不同阶段，我们必须告诉ANTLR每个规则是和哪个阶段相关联的。我们能够通过以大写字母开始的词法规则名字和以小写字母开始的语法规则名字做到这点。例如，ID是一个词法规则名字，expr则是一个语法规则名字。

当开始构建一个新的语法时，对于那些常用的词法构造体：标志符、数字、字符串、注释以及空格等，我们可以从已经存在的语法中拷贝粘贴规则。然后，通过一些细微的调整，就可以让它运行起来。几乎所有的语言，甚至像JSON和XML这样的非编程语言，都有这些记号的变体。例如，C语言的词法分析器完全可以标记化以下的JSON代码：

```
{
  "title":"Cat wrestling",
  "chapters":[ {"Intro":"..."}, ... ]
}
```

另一个例子就是块注释。在C语言中，它们是被/ ... /括起来的。而在XML里，注释是被括起来。但它们除了开始和结束符号之外，或多或少都有相同的词法构造。

对于关键词、运算符和标点符号，我们不需要词法规则，因为我们可以在语法分析器规则中直接引用它们，用单引号括起来，就像`'while'`、`'*'`、`'++'`这样。有些开发者更喜欢使用像MUL而不是字面量`'*'`这样的词法规则引用，这些都没问题，因为它们都有相同的记号类型。

为了阐明词法规则看起来像什么，让我们从标志符开始构建一个常用记号的简单版本。

匹配标志符

在语法伪代码中，一个基本的标志符是一个大写和小写字母的非空序列。根据已经学习到的知识，我们知道需要用(...)+表示法来表示这样的序列模式。因为序列元素可以是大写或小写字母，所以在子规则中我们需要使用选择运算符：

```
ID : ('a'..'z'|'A'..'Z')+ ;    // 匹配一个或多个大小写字母
```

唯一的新ANTLR表示法是范围运算符：'a'..'z'代表从a到z的任意字符。或者你也可以使用Unicode代码点字面量'\uXXXX'，这里的XXXX是Unicode字符代码点值的十六进制值。

作为字符集的一个简写，ANTLR支持我们使用更熟悉的正则表达式集合表示法：

```
ID : [a-zA-Z]+ ;    // 匹配一个或多个大小写字母
```

有时候我们会发现像下面的语法貌似存在冲突的现象：

```
enumDef : 'enum' '{' ... '}' ;
FOR : 'for' ;
ID : [a-zA-Z]+ ;    // 不匹配'enum'或者'for'
```

规则ID也可以同时匹配enum和for这样的关键词，这意味着同样的字符串能被多个规则匹配。但事实上，ANTLR处理这种混合语法时会把字符串字面量以及词法规则与语法规则分隔开，像enum这样的字面量就变成了词法规则并紧随在语法规则之后和在显式的词法规则之前。

ANTLR词法分析器通过偏爱首先指定的规则来解决词法规则间的二义性，这意味着ID规则应该定义在所有的关键词规则之后。ANTLR把隐式的为字面量生成的词法规则放在显式的词法规则之前，因此它们总是有更高的优先级。在这里，'enum'被自动赋予比ID高的优先级。

因为ANTLR会重新排序词法规则并让它在语法规则之后发生。所以上面的语法与下面的变体是相同的：

```
FOR : 'for' ;
ID : [a-zA-Z]+ ;    // 不匹配'enum'或者'for'
enumDef : 'enum' '{' ... '}' ;
```

匹配数字

描述像10这样的整型数字非常容易，因为它只是一个数字序列。

```
INT : '0'..'9'+ ;    // 匹配一个或多个数字
```

或者

```
INT : [0..9]+ ;    // 匹配一个或多个数字
```

浮点数要复杂的多，但如果我们忽略指数的话，可以很容易地制作一个简化版本。浮点数是数字序列后面跟着一个句点和一个可选的小数部分；或者以一个句点开始，然后是数字序列。单独一个句点是不合法的。因此我们的浮点规则使用一个选择和一些序列模式：

```
FLOAT: DIGIT+ '.' DIGIT*    // 匹配1. 39. 3.14159等等
     | '.' DIGIT+           // 匹配.1 .14159
     ;

fragment
DIGIT: [0-9] ;              // 匹配单个数字
```

这里我们使用了一个帮助规则DIGIT，因此我们不必到处去写[0-9]。通过在规则前面加上fragment前缀，我们让ANTLR知道该规则仅被其它词法规则使用。它本身不是一个记号，这意味着我们不能在语法规则中引用它。

匹配字符串字面量

计算机语言中共同具有的下一个常用记号是字符串字面量，例如"hello"。大部分使用双引号作分隔符，有些使用单引号或两者都使用。以双引号为分隔符而言，在语法伪代码中，一个字符串就是在双引号中的任意字符序列：

```
STRING : '"' .*? '"' ;    // 匹配在双引号中的任意字符
```

语法中的点是通配符运算符，它可以匹配任意单个字符。因此，“`.*`”是一个能够匹配任意零个或多个字符的序列的循环。当然，它也将消费字符直到文件结尾，所以不是很有用。幸运的是，ANTLR通过正则表达式表示法（?后缀）提供对非贪婪模式规则的支持。非贪婪模式意味着“直到看见在词法规则中跟在子规则后的字符时才停止吃掉字符”。更确切地说，非贪婪模式规则匹配最小数量的字符，同时仍然允许整个周围的规则被匹配。相反，“`.*`”被认为是贪婪模式，因为它贪婪地消费能够匹配循环内部的所有字符。

以上的STRING规则还做得不够好，因为它不允许字符串中有双引号。为了做到这点，大部分语言定义了以反斜杠开始的转义字符。在字符串中的双引号我们可以使用“\"”。为支持常用的转义字符，我们需要使用以下规则：

```
STRING: '"' (ESC|.)*? '"' ;

fragment
ESC : '\\"' | '\\\\' ;    // 匹配字符\"和\\
```

ANTLR自身也需要避开转义字符，所以这里我们需要用“\”去指定反斜杠字符。

现在，在STRING规则中的循环既可以通过调用fragment规则RULE去匹配转义字符序列，也可以通过点通配符去匹配任意字符。当看到一个非转义双引号字符时“`*?`”子规则运算符终止“`(ESC|.)*?`”循环。

匹配注释和空格

词法分析器会把匹配到的记号通过记号流传递给语法分析器，然后语法分析器检查流的语法结构。但我们希望当词法分析器匹配到注释和空格时能把它们扔掉。那样，语法分析器就不必为匹配无处不在的可选的注释和空格担心。例如，当WS是一个空格的词法规则时以下的语法规则就非常尴尬和容易出错：

```
assign : ID (WS|COMMENT)? '=' (WS|COMMENT)? expr (WS|COMMENT)? ;
```

定义这些被丢弃的记号和定义非丢弃的记号一样，我们只需要使用skip指令去表明词法分析器应该扔掉它们。以下是匹配那些衍生自C的语言的单行和多行注释的语法规则：

```
LINE_COMMENT : '//' .*? '\r'? '\n' -> skip ;    // 匹配"//" stuff '\n'
COMMENT      : '/*' .*? '*/'       -> skip ;    // 匹配"/*" stuff "*/"
```

在COMMENT中，“`.*?`”消费在“`/*`”和“`*/`”之间的任意字符。在LINE_COMMENT中，“`.*?`”消费“//”之后的任意字符，直到它看到一个换行符。

词法分析器接受若干跟随在->运算符后的指令，skip只是它们中的一个。例如，我们可以通过使用channel指令把传递给语法分析器的记号放进隐藏通道。

最后，让我们处理空格这个常用记号。大部分编程语言都把空格当作记号分隔符，但某些像Python这样的语言则把空格用作特殊语法目的。以下是告诉ANTLR如何扔掉空格的语法：

```
WS : (' '|'\t'|'\r'|'\n')+ -> skip ;    // 匹配一个或多个空格但丢弃
```

或者

```
WS : [ \t\r\n]+ -> skip ;    // 匹配一个或多个空格但丢弃
```

当换行既是要被忽略的空格又是命令终结符时，就会有个问题。换行是上下文有关的，在语法上下文中，我们需要扔掉换行，但在其它地方，我们需要把它传递给语法分析器以便让它知道某个命令已经结束。该问题以及它的具体解决方案我们将在以后讨论。

词法分析器和语法分析器的界线

因为词法规则可以使用递归，所以词法解析器在技术上和语法解析器一样强大。那意味着我们甚至可以在词法分析器中匹配语法结构。或者，在另一个极端，我们可以把字符当作记号，使用语法分析器去把语法结构应用到字符流（这种被称为无扫描语法分析器）。这导致什么在词法分析器中匹配和什么在语法分析器中匹配的界线在哪里并不是很明显。幸运的是，有几条经验法则可以让我们做出判断：

- 在词法分析器中匹配和丢弃任何语法分析器根本不需要见到的东西。例如，在词法分析器中识别和扔掉像空格和注释这些东西。否则，语法分析器必须经常查看是否有空格或注释在记号间。
- 在词法分析器中匹配诸如标志符、关键字、字符串和数字这样的常用记号。语法分析器比词法分析器有更多的开销，因此我们不必让语法分析器承受把数字放在一起识别成整数的负担。
- 把那些语法分析器不需要去辨别的词法结构合并成一个单独的记号类型。例如，如果我们的应用把整数和浮点数当作同一事物对待，然后把它们合并成记号类型NUMBER，那么就没必要向语法分析器发送单独的记号类型。
- 合并能被语法分析器视为一个单独实体的任何东西。例如，如果语法分析器不在乎XML标签里的内容，词法分析器可以把尖括号中的任何东西合并成一个单独的被称为TAG的记号类型。
- 如果语法分析器需要先分开一小块文本后才能去处理它，那么词法分析器应该传递独立的构件作为记号给语法分析器。例如，如果语法分析器需要处理一个IP地址的元素，词法分析器应该发送IP构件（整数和点）的独立的记号。

想象下现在需要处理Web服务器上的日志文件，每一行表示一条记录。让我们假设每条记录都有一个请求IP地址、HTTP协议命令和结果代码。这里是一个日志条目的示例：

```
192.168.209.85 "GET /download/foo.html HTTP/1.0" 200
```

如果想要统计文件中有多少行，那么我们可以忽略掉任何东西除了换行字符的序列：

```
file  : NL+ ;                // 匹配换行（NL）序列的语法规则
STUFF : ~'\n'+ -> skip ;     // 匹配和丢弃除'\n'外的任何东西
NL    : '\n' ;               // 返回NL给语法分析器或调用代码
```

词法分析器不必识别太多的结构，语法分析器会匹配换行记号的序列。

接下来，我们需要从日志文件中收集一系列的IP地址。这意味着我们需要一条规则去识别IP地址的词法结构。并且我们也可以提供其它记录元素的词法规则：

```
IP    : INT '.' INT '.' INT '.' INT ;    // 192.168.209.85
INT   : [0-9]+ ;                         // 匹配IP八位组或者HTTP结果代码
STRING: '"' .*? '"' ;                    // 匹配HTTP协议命令
NL    : '\n' ;                           // 匹配日志文件记录终结符
WS    : ' ' -> skip ;                    // 忽略空格
```

拥有一套完整的记号后，我们可以让语法规则匹配日志文件中的记录：

```
file  : row+ ;                // 匹配日志文件中行的语法规则
row   : IP STRING INT NL ;    // 匹配日志文件记录
```

更进一步，我们需要把文本IP地址转换成32位的数字。使用便利的库函数split('.')，我们可以把IP地址作为字符串传递给语法分析器让它去处理。但是，更好的做法是让词法分析器匹配IP地址的词法结构，然后把匹配出的构件作为记号传递给语法分析器。

```
file  : row+ ;                           // 匹配日志文件中行的语法规则
row   : ip STRING INT NL ;               // 匹配日志文件记录
ip    : INT '.' INT '.' INT '.' INT ;    // 在语法分析器中匹配IP地址
INT   : [0-9]+ ;                         // 匹配IP八位组或者HTTP结果代码
STRING: '"' .*? '"' ;                    // 匹配HTTP协议命令
NL    : '\n' ;                           // 匹配日志文件记录终结符
WS    : ' ' -> skip ;                    // 忽略空格
```

把词法规则IP切换成语法规则ip显示了我们可以多么轻易地移动这条分界线。

如果要求处理HTTP协议命令字符串的内容，我们可以遵循相同的思考过程。如果不需要检查字符串的部分，那么词法分析器可以把整个字符串作为一个单独的记号传递给语法分析器。如果我们需要抽出各种不同的部分，最好就是让词法分析器去识别那些部分后再把这些匹配出的构件传递给语法分析器。