第五章 嵌入文法的 Actions

在 ANTLR 中词法规则和语法规则都是一些上下文无关的规则，它们不能满足语法分析中的一些高级需求或特殊需求。如:我们可能要判断一个表达式中的变量之前是否定义了，没有定义要给出编译错误信息，或者我们要在语法分析时记录一些信息以后使用。这时我们就要在文法中嵌入 Actions。Actions 就是嵌入文法中的程序代码，它与我们在 asp 或 jsp 页面 html 代码中嵌入 VB 和 java 的代码非常类似，在生成代码后 actions 会在恰当的位置起到恰到好处的作用。在 ANTLR 文法中嵌入的程序代码要用 Options{ language= }的设置项的目标语言编写，ANTLR 并不会去过多分析 actions 中的代码这全靠编写文法的人自己控制。

5.1 嵌入简单的代码

ANTLR 中 Action 必须写在“{}”花括号中，其中可以有一行或多行代码。我们首先在文法中加入简单的 Action 编写一个示例，下面有一个变量定义的文法，在 variable 规则的结束符“;”之前加入了{System.out.println("a action of variable");}用于输出一行字符。

grammar SimpleAction;

variable : type ID ';' **{System.out.println("a action of variable");}**;

type : 'int' | 'float';

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

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

运行代码如下：

ANTLRInputStream input = new ANTLRInputStream(System.in);

SimpleActionLexer lexer = new SimpleActionLexer(input);

CommonTokenStream tokens = new CommonTokenStream(lexer);

SimpleActionParser parser = new SimpleActionParser(tokens);

parser.variable();

程序运行后输入：int x; 输出：a action of variable

本示例没有让语法分析程序生成语法树，System.out.println 方法是在分析程序执行到该处时被执行的，所以我们可以直接看到效果。

5.2 Actions 中的符号引用

在 Action 中可以引用规则符号来获得符号当前的信息，如：我们将上例中的 Action 修改一下来获得变量的具体和变量名。

```
variable : type ID ';' {System.out.println($type.text + " " + $ID.text);};
```

程序运行后输入：int x; 输出：int x。

引用规则符号时使用字符“$”加上符号名来表示，“$”是一个标识，表明了它是一个符号引用而不是普通的嵌入式代码。

5.3 生成的分析器代码

为了保证本章后面的学习，现在有必要讲解一下 ANTLR 生成的代码与文法之间的大概关系。第一章的示例中就涉及到了语法树，第二章也讲了自顶向下推导树。ANTLR 是一个自顶向下的分析工具，从树的根节点向下，按照左子树到右子树的深度遍历的顺序进行分析。下面看一个简化的表达式文法。

```
expr : subExpr '+' subExpr | subExpr;

subExpr : ATOM '*' ATOM | ATOM;

ATOM : '0'.. '9';
```

我们用这个文法来分析一个表达式：3 * 4 + 5，分析其推导树看图 4.1（左）。

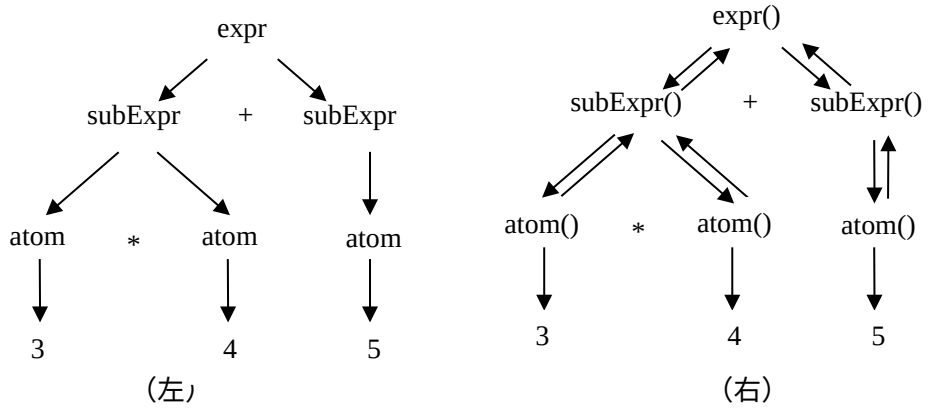


图 4.1

ANTLR 用自顶向下的分析方法从语法树的根节通过输入的字符判断生成推导树，向推导树的子节点分析伸展，直到匹配终结字符。匹配终结字符后会向树的父节点返回，如图 4.1（左）。这种下推回代的过程在数据结构中要用栈来保存临时信息，下推时压栈返回时出栈。ANTLR 使用了函数调用的方式来简洁地实现了这个过程，因为计算机语言中函数调用时系统会自动压栈，函数执行结束返回时系统会自动出栈。这样 ANTLR 就没有必要自己去实现这样的一个栈的构造。如图 4.1（右）让我们把生成推导树的过程想象成一些函数

之间调用的过程。

ANTLR 在生成分析器代码时把每一个规则（包括语法规则和词法规则）生成同名的函数，我们可以方便地叫它规则函数。例如这个简单的 expr 示例生成的代码大致为：

```
public final void expr() throws RecognitionException {
    try   {
            ……//经过词法分析程序对输入字符的判断，决定分支
        case 1 :// subExpr '+' subExpr
            subExpr();
            match(input,5,FOLLOW_5_in_expr12); //匹配 ‘+’
            subExpr();
            break;
         case 2 : // subExpr
            subExpr();
            break;
    } catch (RecognitionException re) {…… }
    finally {       }
}
public final void subExpr() throws RecognitionException {
    try   {
            ……//经过词法分析程序对输入字符的判断，决定分支
         case 1 : // ATOM '*' ATOM
            {
            match(input,ATOM,FOLLOW_ATOM_in_subExpr25);
            match(input,6,FOLLOW_6_in_subExpr27); //匹配‘*’
            match(input,ATOM,FOLLOW_ATOM_in_subExpr29);
            }   break;
         case 2 :// ATOM
        {
            match(input,ATOM,FOLLOW_ATOM_in_subExpr33);
         }  break;
      }
    }
```

```
    catch (RecognitionException re) {…… }
    finally {       }
  }
  public final void mATOM() throws RecognitionException {
    try {
      int _type = ATOM;
      { matchRange('0','9'); }
      this.type = _type;
    }
    finally {       }
  }
```

expr()函数和 subExpr()函数存在于 SimpleExprParser 类中，mATOM()函数存在于 SimpleExprLexer 类中，词法规则生成的方法名前增加了一个“m”字符。

ANTLR 生成代码时我们加入的 Actions 也就嵌入了生成的代码之中。前面的示例 variable : type ID ';' {System.out.println("a action of variable");}生成的代码为：

```
public final void variable() throws RecognitionException {
  try {
      ……
    match(input,ID,FOLLOW_ID_in_variable12);
    match(input,6,FOLLOW_6_in_variable14);
    System.out.println("a action of variable");
    }
  } catch
      ……
}
```

Action 中引用规则符号的示例中 variable : type ID ';' {System.out.println ($type.text + " " + $ID.text);};生成的代码为：

```
public final void variable() throws RecognitionException {
  Token ID2=null;
  type_return type1 = null;
  try {
    pushFollow(FOLLOW_type_in_variable10);
```

```
        type1=type();
        _fsp--;
        ID2=(Token)input.LT(1);
        match(input,ID,FOLLOW_ID_in_variable12);
        match(input,6,FOLLOW_6_in_variable14);
          System.out.println(input.toString(type1.start,type1.stop) + " "
                                        + ID2.getText());
    }
    catch (RecognitionException re) {
        reportError(re);
        recover(input,re);
    }
    finally {    }
    return ;
}
```

原 action 中的$type.text 使生成的代码与第一个示例相比生成了变化，在上面的的代码中已用黑体字提示出来。type()函数返回了一个 type_return 类型的对象，对象名为叫 type1，而原来 Action 中的$type.text 变成了 input.toString(type1.start,type1.stop)。关于$ID.text 在生成的代码中首先用(Token)input.LT(1)返回一个 Token 对象名为 ID2，原来 Action 中的$ID.text 变成了 ID2.getText()。可以看出语法规则 type 和词法规则 ID 生成的代码是不同的。

我们先解释一个语法规则中$ID.text 对应的代码，代码中的 input 就是运行示例的 main 函数中 CommonTokenStream tokens = new CommonTokenStream(lexer);语句中的 tokens 对象。它是词法分析后生成的记号流，记号流曾经在第二章开始时讲过，我们这里不再重复。input.LT(1)功能是从记号流中获得下一个词法符号，由于 variable 规则第一个要匹配符号是 type 类型标识符，第二个要匹配的是 ID 标识符,所以分析程序先调用 type()，并

```
abstract public class Token
{
    abstract String getText();//符号内容。
    abstract void setText(String text);
    abstract int getType();//符号在分析程序中相对应的整数值。
    abstract void setType(int ttype);
    abstract int getLine();//符号所在的行数。
    abstract void setLine(int line);
    abstract int getCharPositionInLine();//符号所在的列数。
    abstract void setCharPositionInLine(int pos);
    abstract int getChannel();//符号所在的频道，对应 CHANNEL 设置。
    abstract void setChannel(int channel);
    abstract int getTokenIndex();//符号在输入中的位置。
    abstract void setTokenIndex(int index);
}
```

认为 type()返回的对象是 type 类型标识符（我们后面再讲 type()函数中的情况）。在 typ1=type()之后分析程序再次调用 input.LT(1)，认为其返回的对象是 ID 标识符并交给 ID2 变量。接下来分析程序调用 match()方法检验这两个符号是不是要匹配的符号，如果不是会报出语法分析错误。我们在 Action 中所写的$ID 就是 ID2 是 Taken 类型的对象，ID2 调用了它的 getText()方法来取得匹配 ID 标识符名。下面给出 Taken 类和它的属性。

在 C#中 Token 是一个接口。

```
public interface IToken
{
    int Channel {get; set;}
    int CharPositionInLine {get; set;}
    int Line {get; set;}
    string Text {get; set;}
    int TokenIndex {get; set;}
    int Type {get; set;}
}
```

有关 ANTLR 类的说明请参见 http://www.antlr.org/api/Java 官方网址。

要想弄清楚语法规则$type.text 是对应的代码，我们还要看一下生成的 type()函数代码。

```
public type_return type()
{
    type_return retval = new type_return();
    retval.start = input.LT(1);
    try {
        if ( (input.LA(1) >= 7 && input.LA(1) <= 8) )
      {   input.Consume();
        errorRecovery = false;
      } else {
        MismatchedSetException mse = new
                            MismatchedSetException(null,input);
          RecoverFromMismatchedSet(input,mse,FOLLOW_set_in_type0);
            throw mse;
      }
      retval.stop = input.LT(-1);
    } catch (RecognitionException re) {
```

```
            ReportError(re);
            Recover(input,re);
        } finally {      }
        return retval;
    }
    public class type_return : ParserRuleReturnScope { };
```

type()函数中与$type.text 有关的代码已经用黑体字标记出来。函数中先创建了 type()函数的返回值实例 type_return retval = new type_return()，retval 对象作为返回值赋给了 variable()函数中的 type1 对象。type_return 类型也在生成的分析程序代码中下面是 type_return 类与其基类的代码。type_return 类继承了 ParserRuleReturnScope 类，而 ParserRuleReturnScope 类又继承了 RuleReturnScope 类。其中 ParserRuleReturnScope 类与 RuleReturnScope 类是 ANTLR 中定义的类（下面看一下 C#语言的定义）。

```
public class ParserRuleReturnScope : RuleReturnScope
{   public IToken start; //规则匹配的第一个词法符号
    public IToken stop; //规则匹配的第一个词法符号
    public ParserRuleReturnScope();
}
public class RuleReturnScope
{   public RuleReturnScope();
    public virtual object Start { get; }
    public virtual object Stop { get; }
    public virtual object Template { get; }
    public virtual object Tree { get; }//规则生成的语法树
}
```

在 variable() 函数中 $type.text 变成了 input.toString(type1.start,type1.stop)，toString(Token, Token)方法是将一个语法规则匹配的第一个词法符号和最后一个词法记号之间所有匹配的输入内容转换成一个字符串。语法分析程序利用这个方法可以获得 type 规则所匹配的类型信息。

5.4 使用变量

除了直接使用“$”符号加规则名的方法引用规则以外，ANTLR 文法可以将规则符号赋值到一个变量中，然后引用变量就等于引用规则符号。修改 SimpleAction 文法：

variable : t=type id=ID ';'

{System.out.println("type: " + $t.text + " ID: " + $id.text);};

对变量的引用是使用“$”符号再加上变量名的形式。下面看一下生成的代码：

```
public void variable() // throws RecognitionException [1]
{   IToken id = null;
   type_return t = null;
      ……
   t = type();
   id = (IToken)input.LT(1);
        System.out.println(input.ToString(t.start,t.stop) + " " +
                                    id.Text);
      ……
```

5.5 +=的用法

修改 SimpleAction 文法，使能够在类型标识符后可以定义多个变量。这文法中加入了“+=”操作符功能是将所有定义的变量收集到一个集合当中。

```
grammar SimpleAction;
variable : type ids+=ID (',' ids+=ID)* ';'
            {   System.out.println($type.text);
                for(Object t : $ids)
                   System.out.print(" " + ((Token)t).getText()); }
          ;
```

运行此示例输入：int x, y, z; 输出：int x y z　我们来看一下生成的代码。

```
public final void variable() throws RecognitionException {
    Token ids=null;
  List list_ids=null;
  type_return type1 = null;
  try {
    pushFollow(FOLLOW_type_in_variable9);
    type1=type();
```

```
_fsp--;
ids=(Token)input.LT(1);
match(input,ID,FOLLOW_ID_in_variable13);
if (list_ids==null) list_ids=new ArrayList();
 list_ids.add(ids);
loop1:
do{ int alt1=2;
   int LA1_0 = input.LA(1); if ( (LA1_0==6) ) { alt1=1; }
   switch (alt1) {
      case 1 : {
         match(input,6,FOLLOW_6_in_variable16);
         ids=(Token)input.LT(1);
         match(input,ID,FOLLOW_ID_in_variable20);
         if (list_ids==null) list_ids=new ArrayList();
              list_ids.add(ids);
      } break;
      default :  break loop1;
      }
   } while (true);
   match(input,7,FOLLOW_7_in_variable24);
   System.out.println(input.toString(type1.start,type1.stop));
            for(Object t : list_ids) System.out.print(" " +
                                ((Token)t).getText());
}
catch (RecognitionException re)    ……
}
```

代码中添加了一个 List 类型的 list_ids 对象，每匹配一个 ID 符号时就加入到 list_ids 中。List 中存放的是 Object 类型所以我们输出时要加类型转换（ANTLR 下个版本是否加入泛型?）

5.6 规则的参数

规则被翻译成函数自然可以加参数。ANTLR 中利用规则的参数可以实现上下文信息的传递来弥补上下文无关文法的缺点。下面用一个简单的示例来演示参数的用法。

```
grammar SimpleParam;
variable : type idList[$type.text] ';';
idList[String typeName]
 : ID '=' (con=INT | con=FLOAT)
   {  if(typeName.equals("int") && $con.text.indexOf(".") != -1) {
            System.out.println("error: value of float  cannot assgin to var of
                                    int.");
       }
   };
type : 'int' | 'float';
INT : ('0'..'9')+;
FLOAT : ('0'..'9')+ ('.' ('0'..'9')+)?;
ID :  ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')* ;
WS :  ( ' ' | '\t' | '\r' | '\n' )+ { $channel = HIDDEN; } ;
```

分析器运行后输入：int x = 5.5; 输出：error: float value cannot assgin to int type var。variable 规则中将 type 匹配的类型名传递给 idList 规则使在 idList 规则中可以判断是否将浮点型付给了整型，如果是则提示错误信息。下面是生成的分析程序代码：

```
public final void variable() throws RecognitionException {
   ......
   idList(input.toString(type1.start,type1.stop));
}
public final void idList(String typeName) throws RecognitionException {
   ......
   if(typeName.equals("int") && con.getText().indexOf(".") != -1) {
      System.out.println("error: value of float  cannot assgin to var of int.");
    }
   ......
}
```

规则参数也可以写成“$”本规则名为前缀的写法。$idList.typeName 与 typeName 是等价的。

5.7 规则的返回值

规则也可以有返回值，规则只有单个返回值时规则函数会从 void 类型变成有返回值的函数。规则可以返回多个值，实际上返回多值的功能是通过返回一个具有多个属性的对象来实现。不管返回值是单个值对象还是由对象携带多个返回值，在文法中的写法都是一样的，以“ruleRetObj.RetValue”带有规则名前缀的形式来引用。返回对象就是我们前讲过的 ParserRuleReturnScope 类的派生类对象。我们用一个示例来看一下返回值的用法。

```
grammar SimpleReturn;
variable : type idList[$type.text]
{ System.out.println($idList.retList + "\r\n" + $idList.count); }';';
idList[String typeName] returns [List retList, int count]
 : ids+=ID (',' ids+=ID)*
   {  $retList = $ids;
      $count = $ids.size();
   };
type : 'int' | 'float';
INT : ('0'..'9')+;
FLOAT : ('0'..'9')+ ('.' ('0'..'9')+)?;
ID :   ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')* ;
WS :   ( ' ' | '\t' | '\r' | '\n' )+ { $channel = HIDDEN; } ;
```

运行后输入：int x, y, z

输出：[[@2,4:4='x',<4>,1:4], [@5,7:7='y',<4>,1:7], [@8,10:10='z',<4>,1:10]]

3

variable 规则从 idList 规则返回定义的变量名集合和变量的个数，集合中保存的是 Token 对象。下面看一下生成的程序代码。

```
public static class idList_return extends ParserRuleReturnScope {
   public List retList;
   public int count;
};
public final void variable() throws RecognitionException {
  type_return type1 = null;
  idList_return idList2 = null;
     ......
  type1=type();
     ......
```

```
 idList2=idList(input.toString(type1.start,type1.stop));
     ……
 System.out.println(idList2.retList + "\r\n" + idList2.count);
     ……
}
 public final idList_return idList(String typeName)
                                        throws RecognitionException {
     idList_return retval = new idList_return();
       ……
     list_ids.add(ids);
           ……
     if (list_ids==null) list_ids=new ArrayList();
     list_ids.add(ids);
       ……
    retval.retList = list_ids;
    retval.count = list_ids.size();
   return retval;
}
```

规则 idList 的两个返值放到了 idList_return 的对象中返回，idList_return 类加入了 retList 和 count 两个属性保存了返回值。

5.8 @header 用法

ANTLR 中每一个文法规则都生成对应的函数，这些函数是分析器类的方法。第一章已经介绍过语法分析器和词法分析器都分别生成 XXXParser 类和 XXXLexer 类。当需要向类的顶端加入代码时可以用@header 定义。一般情况下（如 java）用@header 来加入包名和导入 imports 其它 java 类。

```
grammar SimpleMember;

@options { language=Java; }
@header { package Simple; }
……
```

@header 在默认的情况下会设置 XXXParser 类代码，也就是语法分析类。我们也可以显示的指明设置词法类还是设置语法类。

```
@parser::header { package Simple.Parser; }

@lexer::header { package Simple.Lexer; }
```

在 C#中命名空间由于有“{}”的存在所以无法用来@header 定义，@header 中不

用输入不闭合的花括号。可以用如下方法来定义：

```
@lexer::namespace {
  Simple.Lexer
}
@parser::namespace {
  Simple.Parser
}
```

::namespace 只能用于 C#。还要注意一点我们在后面还会讲到树文法（tree grammar）在树文法中也可以使用@namespace {OKProject}来设置命名空间。

5.9 @members 用法

在文法中可以用@members 定义分析器类的成员，和@header、Actions 一样并没有具体的限制，你可以在其中任何内容如属、方法、事件、内部类和注释等，还有一个重要的应用是可以用@members 重写基类的方法实现一些高级功能。

下面请看示例:

```
grammar SimpleMember;

options {
 language=CSharp;
}
@header {
using System.Collections.Generic;
}
@members {
private List<IToken> RetList = new List<IToken>();
private int Count = 0;
}
variable : type idList
{
foreach(IToken t in RetList)
   System.Console.WriteLine(t.Text);
System.Console.WriteLine(Count);
}';';

idList
 : ids+=ID (',' ids+=ID)*
  {
    RetList = new List<IToken>();
    foreach(object t in $ids)
       RetList.Add((IToken)t);
```

```
        Count = $ids.Count;
      };
    type : 'int' | 'float';
    INT : ('0'..'9')+;
    FLOAT : ('0'..'9')+ ('.' ('0'..'9')+)?;
    ID :  ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')* ;
    WS :  ( ' ' | '\t' | '\r' | '\n' )+ { $channel = HIDDEN; } ;
```

这个示例实现了与之前的示例相同的功能，但没有使用返回值，idList()函数中利用分析器类的成员 RetList 和 Count 来保存变量的信息，然后在 variable()函数中输出。类成员是在不同的成员函数间传递信息的一种方法，但要注意对类成员赋值和取值的先后顺序，不然会在分析过程中出现异常。该文法生成了如下的代码：

```
using System.Collections.Generic;
......
public class SimpleMemberParser : Parser
{
   private List<IToken> RetList = new List<IToken>();
   private int Count = 0;
   public void variable()
   {
        ......
            foreach(IToken t in RetList)
               System.Console.WriteLine(t.Text);
            System.Console.WriteLine(Count);
        ......
   }
   public void idList()
   {
      IToken ids = null;
      IList list_ids = null;
          ......
          if (list_ids == null) list_ids = new ArrayList();
          list_ids.Add(ids);
              ......
              RetList = new List<IToken>();
              foreach(object t in list_ids)
                 RetList.Add((IToken)t);
              Count = list_ids.Count;
              ......
      }
```

5.10 scope 属性

规则的参数和返回值可以实现在规则之间传递信息，但要跨越多个规则传递时，每一个规则都要定义同样的参数和返回值一级一级的传递，这样比较麻烦也不太现实 。@member 定义的类成员也可以实现信息传递，但是类的成员在类的范围内是全局可见的，大量地使用类成员代替参数对程序结构带来的坏处是显然的。

我们可以把规则看成是象类一样的程序单位，如果把 variable、idList 和 type 看成是三个类，我们可以定义这些类的属性在文法中使用 scope{...}来定义。看下面的示例：

```
grammar SimpleScope;
options { language=CSharp; }
@header {
    using System.Collections.Generic;
}
variable
scope {
    List<IToken> RetList;
    int Count;
}
: type {$variable::RetList = new List<IToken>();} idList
{
    foreach(IToken t in $variable::RetList)
    System.Console.WriteLine(t.Text);
    System.Console.WriteLine($variable::Count);
}';';

idList : ids+=ID (',' ids+=ID)*
  {
    foreach(object t in $ids)
       $variable::RetList.Add((IToken)t);
    $variable::Count = $ids.Count;
  };

type : 'int' | 'float';
INT : ('0'..'9')+;
FLOAT : ('0'..'9')+ ('.' ('0'..'9')+)?;
ID :   ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')* ;
WS :   ( ' ' | '\t' | '\r' | '\n' )+ { $channel = HIDDEN; } ;
```

这个示例使用 scope 属性代替之前的类成员和返回值，实现了规则之间的信息传递。在使用 scope 属性时在属性名前要加“$”符号规则名和“::”，下面看一下生成的代码。

```
protected class variable_scope
{
   protected internal List<IToken> RetList;
   protected internal int Count;
```

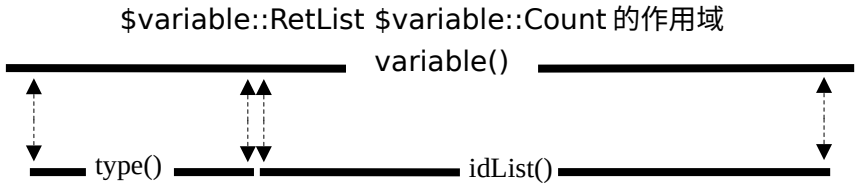
```
}
protected Stack variable_stack = new Stack();
public void variable() // throws RecognitionException [1]
{
   variable_stack.Push(new variable_scope());
     ......
   ((variable_scope)variable_stack.Peek()).RetList =  new List<IToken>();
         idList();
     ......
   foreach(IToken t in ((variable_scope)variable_stack.Peek()).RetList)
            System.Console.WriteLine(t.Text);

System.Console.WriteLine(((variable_scope)variable_stack.Peek()).Count);
     ......
      finally
     {
          variable_stack.Pop();
      }
}
public void idList()
{
......
     foreach(object t in list_ids)
        ((variable_scope)variable_stack.Peek()).RetList.Add((IToken)t);
   ((variable_scope)variable_stack.Peek()).Count =  list_ids.Count;
.....
}
```

从生成的代码中可以看出 scope 属性的实现有三部分，首先定义了一个新的内部类 variable_scope，类中定义了 RetList 和 Count 两个属性，然后定义了一个栈对象 variable_stack，最在在 variable()函数开始时创建一个 variable_scope 的对象并压栈，在 variable()结束时弹出 variable_scope 的对象。对 scope 属性的操作则是操作栈顶对象的属性。

@scope{}中定义的属性可能直接初始化对象，原因我想大家也可以明白。所以我们只能在 idList 子规则执行前创建 RetList 对象的实例。还要注意的是这个版本的 ANTLR 以 C#为目标语言时@scope{}中不能加入注释。

5.10.1 scope 属性的生命期和作用域

因为拥有 scope 属性的对象实例是在规则函数开始执行时创建，结束执行时删除。所以 scope 属性的生命期覆盖了其子规则函数的执行时间。本示例中 variable 规则的 RetList 和 Count 属性的生命期跨越了 type()和 idList()函数。



scope 属性只能在语法规则中定义，所以它的作用域是整个语法分析类。

关于 scope 属性的功能还有重要的一点，就是当规则出现递归时 scope 属性可以很好处理嵌套结构中属性生命期和作用域。下面给出一个关于分析 SELECT 语句的极度精简的示例。我们都很熟悉 SQL SELECT 语句，SELECT 语句可以嵌套定义主查询可以有子查询，子查询还可以有子查询。当我们解析 SELECT 语句时我们要主查询和子查询分别都查询了哪些表。无论是主查询还是子查询都是嵌套定义的一套规则，规则中的 action 代码要处理自己在不同嵌套层中的数据。在这种情况下可以用 scope 属性所具有的特点来轻松处理作用域的问题。

```
grammar SimpleSelect;
options { language=CSharp; }
@header {
using System.Collections.Generic;
}
selectStatement
scope {Dictionary<string, string> tableSourceList;}
:
{$selectStatement::tableSourceList = new Dictionary<string,
string>();}
    selectClause (fromClause)?
    { foreach(KeyValuePair<string, string> keyValue in
                              $selectStatement::tableSourceList)
      System.Console.WriteLine($selectStatement.text +
                    ":( " + keyValue.Key + " " + keyValue.Value + " )");
    }
      ;
    selectClause
      : 'SELECT' ('ALL' | 'DISTINCT')? '*'
      ;
    fromClause
      : 'FROM' tableSource (',' tableSource)*
      ;
    tableSource
      : tn=TableName
        {$selectStatement::tableSourceList.Add($tn.text, $tn.text);}
      | '(' ss=selectStatement ')' 'AS' tn=TableName
        {$selectStatement::tableSourceList.Add($tn.text, $ss.text);}
```

```
    ;
TableName : Identifier;
Identifier : ('a'..'z' | 'A'..'Z' | '_')
           ('a'..'z' | 'A'..'Z' | '_' | '0'..'9')*;
WS :  ( ' ' | '\t' | '\r' | '\n' )+ {Skip(); } ;
```

示例中 selectStatement 规则定义了一个 tableSourceList 属性用来存放当前查询所查询的别名和表名，分析完当前查询后输出所有查询表的信息。FROM 子句的 tableSource 规则可以匹配表名和子查询，子查询又是 selectStatement 规则形成了查询的嵌套定义。(.net 中的 Dictionary<T, T>相当于 java 中的 Map<T,T>)

在子查询分析期间子查询创建了 tableSourceList 属性对象并入栈，这时主查询创建的 tableSourceList 属性对象已经不是栈顶的元素。所以在子查询分析期间主查询的属性被隐藏，当前所分析规则的信息被很好的保存。

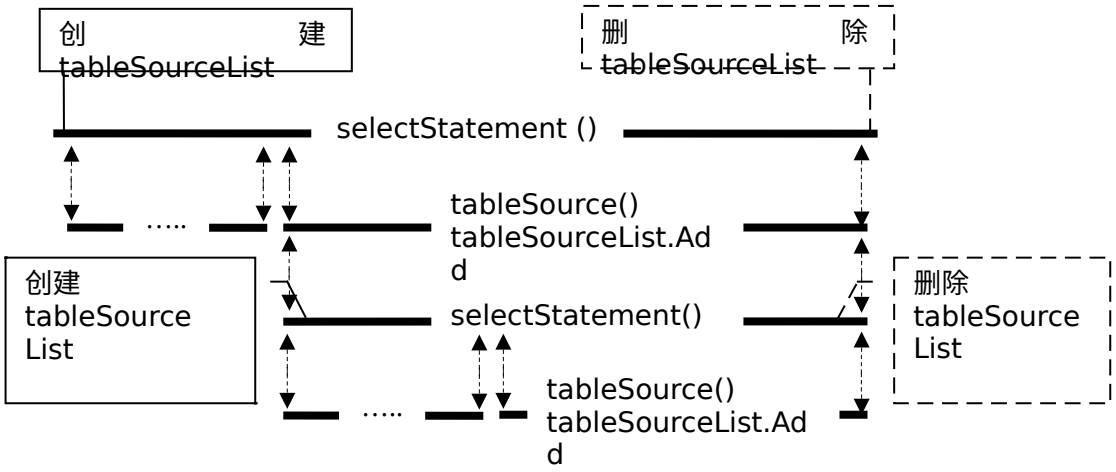


图 5.1

ANTLR 允许直接操作栈中的任何 scope 属性对象，在属性名后加方括号指定属性对象在栈中的位置。如果索引为负数，代表从栈顶向栈底方向的第 N 个元素。

如：$selectStatement[-2]::tableSourceList;

生成的代码为：

((selectStatement_scope)selectStatement_stack[**selectStatement_stack.Count-2-1]**).tableSourceList

如果索引为正数，代表从栈底向栈顶方向的第 N 个元素。

如：$selectStatement[1]::tableSourceList;

生成的代码为：

((selectStatement_scope)selectStatement_stack[**1**]).tableSourceList

其中如果索引为 0，代表的是栈底的元素。生成的代码为：

((selectStatement_scope)selectStatement_stack[**0**]).tableSourceList

5.11 @init

ANTLR 规则中可以使用@init 定义规则函数的初始化代码，@init 定义的代码将出现在在其它 actions 的代码之前，我们可以将一些局部变量定义、属性初始化等代码写在@init 中。如我们在上一节 scope 属性代替返回值的示例 SimpleScope 中使用了 action 去创建对象，有@init 可以把文法改写为：

```
variable
scope {
List<IToken> RetList;
int Count;
}
@init {
$variable::RetList = new List<IToken>();
}
: type idList
{
foreach(IToken t in $variable::RetList)
   System.Console.WriteLine(t.Text);
System.Console.WriteLine($variable::Count);
}';';
```

在生成的代码中可以看到，创建对象的代码放到了合适的位置。

```
public void variable()
{
   variable_stack.Push(new variable_scope());
   ((variable_scope)variable_stack.Peek()).RetList = new
List<IToken>();
   ......
}
```

5.12 @after

与@init 相对 ANTLR 文法中可以用@after 定义规则函数最后执行的代码。可以在@after{}中做一些删除对象输出结果等收尾工作。我们再修改 SimpleScope 示例使用了@after{}输出结果，文法如下：

```
variable
scope {
List<IToken> RetList;
int Count;
}
```

```
@init {
$variable::RetList = new List<IToken>();
}
@after{
foreach(IToken t in $variable::RetList)
   System.Console.WriteLine(t.Text);
System.Console.WriteLine($variable::Count);
}
: type idList ';';
```

@after 和@init 的位置谁先谁后没有关系都可以正确生成代码，@after 中的代码被放到了 try{}的最后，在异常处理范围之内。下面看一下生成后的代码：

```
public void variable()
{  ......
   try
   {
         ......
      foreach(IToken t in
((variable_scope)variable_stack.Peek()).RetList)
            System.Console.WriteLine(t.Text);
      System.Console.WriteLine(((variable_scope)variable_stack.Pee
      k()).Count);
      }
      catch (RecognitionException re) {......}
      finally {......}
      return ;
}
```

5.13 全局 scope 属性

scope 属性受规则的生命期的限制，规则开始时创建规则结束时删除。规则可以引用本规则的 scope 属性，在规则生命期内只有其子规则可以引用父规则的 scope 属性。上一节的嵌套示例中子查询统计自己查询表 tableSourceList 属性时，父查询的 tableSourceList 被屏蔽，虽然可以用 tableSourceList[-1]访问，但是是分开的列表。

但是有时我们需要用一个属性来统计某个规则下的所有规则（包括子规则）的信息。ANTLR 中可以定义一种全局 scope 属性，能够在多个规则中共享中使用。全局 scope 属性是使用 scope 关键字加上一个全局 scope 名称，其后与 scope 属性一样在{}中可以定义一到多个属性。请看下面的文法：

```
grammar SimpleGlobalScopes;
scope CScope {
    String name;
    List<String> symbols;
}
```

```
prog
scope CScope;
@init {
    $CScope::symbols = new ArrayList<String>();
    $CScope::name = "global";
}
@after {
    System.out.println("global symbols = "+$CScope::symbols);
}
: variable* func*;
func : 'void' ID '(' ')' '{' variable* stat+ '}';
block :
    '{' variable* {$CScope::symbols.add($variable.text);}
    stat+ '}';
stat: ID '=' INT ';' | block;
variable : type ID (ID)* ';';
type : 'int' | 'float';
INT : '0'..'9' + ;
ID  :   ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')* ;
WS  :   ( ' ' | '\t' | '\r' | '\n' )+ { $channel = HIDDEN; } ;
```

本文法定义了简单的编程语言的结构，启始规则 prog 代表程序，程序下有变量和函数的定义，函数中可以有变量定义和语句，而语句中有语句块和赋值语句。语句块是包括在花括号“{}”内的一组语句，语句块又可以包含语句块它们是递归的关系。

文法中定义了两个全局 scope 属性 name 和 symbols，全局 scope 属性名为 CScope。Name 用来保存当前域的名称，symbols 用来保存当前域的所有变量的类型和名称，全局 scope 属性要写在所有规则之前。

在 prog 规则名后面加入了 scope CScope 定义;，scope CScope;代表 prog 规则拥有一个 CScop 全局 scope 属性的对象。分析程序会在规则开始时创建一个 CScope 属性对象，规则结束时删除 CScope 属性对象。我们看一下 scope CScope;对应生成的代码：

```
public final void prog() throws RecognitionException {
   CScope_stack.push(new CScope_scope());
     ......
   finally {
      CScope_stack.pop();
   }
}
```

拥有全局 scope 属性对象的规则可以在自己的@init、@after 中对全局 scope 属性进行初始化和收尾工作。本例中 prog 规则在@init 中创建了 symbols 集合的对象，并将 name 设置为”global”说明这是整个程序的全局变量集合。与 scope 属性一样对全局 scope 属性的引用也是使用“$”加全局 scope 名加“::”符号和属性名的形式，本例中 $CScope::symbols.add($variable.text);语句将变量类型与标识符加到 symbols 列表中。编译运行后输入：“int x, y,z;”输出：“global symbols = [int x, int y, int z]”。下面

有一个应用全局 scope 属性的实例，来说明何时使用全局 scope 属性。

许多编程语言的语句块中可以定义变量，语句块之间可以是嵌套关系。如果外层语句块定义了变量 x，这时如果内层语句块也定义变量 x 分析器应该提示变量名重复的错误信息。这就要求我们在分析内层语句块时在列表中也要有上屋语句块的变量信息，这样才能知道是否与上层变量重句，这时可以使用全局 scope 属性来实现这个功能。下面我们修改文法使分析器可以统计出二类变量，第一类是函数外的全局变量，第二类是函数中最前面定义的变量我们管这样的变量叫函数变量，作用域在函数之内。函数内变量还可以在函数内的语句块中定义其作用域在当前的语句块之内。对这两类变量的统计我们使用了全局 scope 属性来实现。请看下面文法：

```
grammar SimpleGlobalScopes;
scope CScope {
  String name;
  List<String> symbols;
}
prog
scope CScope;
@init {
  $CScope::symbols = new ArrayList<String>();
  $CScope::name = "global";
}
@after {
  System.out.println("global var = "+$CScope::symbols);
}
: variable* func*
;
func
scope CScope;
@init {
  $CScope::symbols = new ArrayList<String>();
  $CScope::name = "func";
}
@after {
  System.out.println("func var = "+$CScope::symbols);
}
: 'void' ID '(' ')' '{' variable* stat+ '}'
;
block
scope CScope;
@init {
  $CScope::symbols = new ArrayList<String>();
  $CScope::name = "block";
}
@after {
```

```
   System.out.println("block var = "+$CScope::symbols);
}
: '{' variable* stat+ '}'
;
stat: ID '=' INT ';' | block;
variable
: type a=ID {$CScope::symbols.add($type.text + " " + $a.text);}
 (',' b=ID {$CScope::symbols.add($type.text + " " + $b.text);} )* ';'
;
type : 'int' | 'float';

INT : '0'..'9' + ;
ID  :  ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')* ;
WS  :  ( ' ' | '\t' | '\r' | '\n' )+ { $channel = HIDDEN; } ;
```

分析下面内容：

```
int a,b;
void f() {
   int x;
   x = 1;
   {
      int y;
      y = 2;
      {
         int z;
         z = 3;
      }
   }
}
```

分析器输入输出：

```
func var = [int x, int y, int z]
global var = [int a, int b]
```

5.14 catch[]与 finally

从前面的例中已经看到规则函数中使用 try..catch()..finally 将代码包括到异常处理之中。ANTLR 允许我们修改默认的异常处理代码,可以在规则定义的后面使用 catch[]和 finally 定义自己的异常处理程序。下面我们还是修改 SimpleScope 示例请看下面的文法：

```
variable
scope {
List<IToken> RetList;
```

```
int Count;
}
@after{
foreach(IToken t in $variable::RetList)
   System.Console.WriteLine(t.Text);
System.Console.WriteLine($variable::Count);
}
@init {
$variable::RetList = new List<IToken>();
}
: type idList ';'
catch[RecognitionException re] {
System.Console.WriteLine("variable rule error: " + re.Message);
}
finally { System.Console.WriteLine("parsing variable rule finished."); }
```

异常处理的写法与 java，C#语言很类似只是 catch 后面使用的是方括号。但有两点需要注意，一是 catch 和 finally 的位置是在规则结束符号“;”的后面。二是 finally 中无需写出 scope 属性的删除代码，因为 ANTLR 已经自动生成了删除 scope 属性对象的代码。

到此我们学到了 ANTLR 文法中的各种嵌入代码的方式，下面我们总结一下：

```
parser grammar T;
@header {……}
@members {……}
scope scopeName {……}
rule1[int x] returns [int y]
scope{string Y;}
@init {……}
@after {……}
: {……} a=Rule2 {……};
catch[……] {
……
}
finally { …… }
```

其中要注意的是如果我们要在形如 classdef：method*这样的规则中 method 后加入 Actions 时不能写成 method{…}*，这时要写成(method{…})*的形式让 method 和 Actions 在子规则中。

5.14.1 @rulecatch

如果每一个规则的 catch 代码都是相同的，ANTLR 有一个比较简单的写法。在文法中与@member 同样级别可以加入一个@rulecatch 块加入关于 catch 的程序代码。

```
@rulecatch {
    catch (RecognitionException e) {
        throw e;
    }
}
```

5.15 嵌入规则的 Actions

规则中的 Actions 也可以看成是文法规则中的一项内容。它在规则中的位置与生成的代码执行的顺序是一致的。a：b Acion1 c 规则中 Action1 会在 a 之后 c 之前执行。下面我们研究一下在选择结构、重复和嵌套的规则定义中 Action 出现的位置对代码的影响。

```
grammar ActionInRule1;
a : A {System.out.println($A.text);} B {System.out.println($B.text);};
A : 'A';
B : 'B';
```

运行后输入: AB

输出：A
B

生成代码为：

```
Token A1=null;
Token B2=null;
try {
   A1=(Token)input.LT(1);
   match(input,A,FOLLOW_A_in_a10);
   System.out.println(A1.getText());
   B2=(Token)input.LT(1);
   match(input,B,FOLLOW_B_in_a14);
   System.out.println(B2.getText());
}
```

从代码中看到 Action 与规则是顺序的，所以如果将 a 规则修改成:

```
a : A {System.out.println($A.text); System.out.println($B.text);} B ;
```

对$B 的引用是没有意义的，所以 ANTLR 给出错误提示“illegal forward reference: $B.text”。

继续修改文法使 A 和 B 成选择关系。

```
grammar ActionInRule2;
a : A {System.out.println($A.text);}
   | A {System.out.println($A.text);};
```

生成的代码为：

```
a : A {System.out.println($A.text);}
public final void a() throws RecognitionException {
```

```
    Token A1=null;
     Token A2=null;
     try {
           ……
            switch (alt1) {
          case 1 :  {
            A1=(Token)input.LT(1);
           match(input,A,FOLLOW_A_in_a9);
            System.out.println(A1.getText());
         } break;
         case 2 : {
            A2=(Token)input.LT(1);
        match(input,A,FOLLOW_A_in_a16);
            System.out.println(A2.getText());
         }   break;
       }……
}
```

选择关系的规则中生成的代码大体是一个分支结构。在这个示例中我们会在生成的代码中兴奋地发现 ANTLR 智能的区别了两个分支中的$A 引用，分别用 A1 和 A2 来代替。A1 和 A2 是在函数一开始定义的，所以它们的作用域还是整个函数。这个示例说明在并列的选择结构中同样的$A 引用，是分别引用了不同的变量。

```
grammar ActionInRule2;
a : v=A {System.out.println($v.text);}
   | v=A {System.out.println($v.text);};
```

如果将两个子规则 A 赋给同一个变量，这时的 v 则是同一个变量。这里我们不列出生成代码。根据之前的讲述读者应该可以理解。

```
grammar ActionInRule2;
a : A {System.out.println($A.text);}
   | B {System.out.println($A.text);};
```

如果选择的两个并列项是 A 和 B，而 B 的一端要引用$A。这时 B 分支生成的代码中$A.text 没能被 ANTLR 处理，生成的代码还是 System.out.println($A.text);原样，这当然是无法编译的。这说明一个选择项中无法用规则名直接引用别一个选择项的规则符号(这种情况只能使用变量)。

下面看一下规则中重复的文法：

```
grammar ActionInRule3;
variable : type ID {System.out.println($ID.text);}
(',' ID {System.out.println($ID.text);} )* ';';
type : 'int' | 'float';
ID  :  ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')* ;
WS  :  ( ' ' | '\t' | '\r' | '\n' )+ { $channel = HIDDEN; } ;
```

此文件生成代码时 ANTLR 提示错误信息“ID is a non-unique reference”，说明 ANTLR 不能区分两个$ID 的引用。这时我们只能把两个$ID 赋给两个变量然后再引用，请

看下面对 variable 规则的修改：

```
variable : type a=ID {System.out.println($a.text);}
(',' b=ID {System.out.println($b.text);} )* ';';
```

修改后的文法解决了这个问题，下面看一下生成的代码：

```
public final void variable() throws RecognitionException {
    Token a=null;
    Token b=null;
    try {
            ......
        a=(Token)input.LT(1);
        match(input,ID,FOLLOW_ID_in_variable14);
        System.out.println(a.getText());
        do {
                ......
            b=(Token)input.LT(1);
            match(input,ID,FOLLOW_ID_in_variable23);
            System.out.println(b.getText());
        } while (true);
            ......
    }......
}
```

生成代码中对应重复的是循环语句，“*”的作用是使用 do..while 语句实现的。那么为什么 ANTLR 对于此文法不能智能的区分两个$ID 引用呢？原因是 ID（',' ID）* 规则与之前的选择结构不同，第二个到第 N 个 ID 也就是子规则（',' ID）位置在第一个 ID 之后，所以在此子规则中引用第一个 ID 的需求是可能的，因此无法区分是要引用哪一个 ID，只能将其赋给两不变量来实现。

了解了规则中的 Actions 在生成代码中的位置后，我们可以更好的使用 Actions。因为 java 和 C#语言允许在任何语句块中定义新的对象，所以在 Actions 中我们也可以定义对象(当目标语言为 C 时会有限制)。根据 Actions 在生成代码中的位置我们就可以判断出定义的对象的作用域和生命期，这对正确做 Actions 有很大帮助。

```
variable : type a=ID {String theFirstID = $a.text;}
           (',' {List<String> moreIDs = new ArrayList<String>();}
           b=ID { moreIDs.add($b.text);} )* ';';
```

5.16 词法规则中的 Actions

词法规则中使用 Actions 和语法规则中不完全相同，前面我们讲解的都是语法规则中的 Actions。下面我们学习一下词法规则中如何使用 Actions。词法规则中的 Actions 用法和语法规中的用法大致相似，如语法规则相同词法规则也可以用规则符号名直接引用规则符号信息。

```
grammar ActionInTokenRule1;
```

```
variable : QID;
QID : ID {System.out.println($ID.text);} ;
ID : ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*;
WS : ( ' ' | '\t' | '\r' | '\n' )+ { $channel = HIDDEN; } ;
```

词法规则中也可以使用变量：

```
QID : id=ID {System.out.println($id.text);} ;
```

下面看一下生成的代码：

```
public final void mQID() throws RecognitionException {
  try {
      int _type = QID;
       Token id=null;
      int idStart15 = getCharIndex();
      mID();
       id = new CommonToken(input, Token.INVALID_TOKEN_TYPE,
                            Token.DEFAULT_CHANNEL,
                            idStart15, getCharIndex()-1);
       System.out.println(id.getText());
      this.type = _type;
   } finally { }
}
```

词法分析生成的规则函数中定义了一个 Token 类型的 id 对象，本章前面已经讲过词法规则返回的是 Token 类型，词法规则的信息是使用 Token 的对象来获得的。id = new CommonToken(input, Token.INVALID_TOKEN_TYPE, Token.DEFAULT_CHANNEL, idStart15, getCharIndex()-1);这条语句用来获得输入中与 ID 规则匹配的信息。这个处在词法分析器代码中的 input 对象是 ICharStream 对象的实例，是 main 函数中的第一条语句 ANTLRInputStream input = new ANTLRInputStream(System.in);的 input。（语法分析中的 input 是 main 函数中的第三条语句是 CommonTokenStream 的对象。）通过调用 CommonToken 的构造函数将输入流 input 中的 DEFAULT_CHANNEL 频道的 idStart15 开始到 getCharIndex()-1 位置的字符生成 CommonToken 对象。CommonToken 对象是 Token 的派生类，这样就可以使用 Token 的属性 text 来获得词法符号的内容。下面是 main 函数：

```
public static void main(String[] args) throws Exception
{
    ANTLRInputStream input = new ANTLRInputStream(System.in);
    ActionInTokenRule2Lexer lexer = new ActionInTokenRule2Lexer(input);
    CommonTokenStream tokens = new CommonTokenStream(lexer);
    ActionInTokenRule2Parser parser = new ActionInTokenRule2Parser(tokens);
    parser.variable();
}
```

分析了词法分析的代码之后，我们来学习一下 Action 在词法规则中与语法规则的不同。下面我们象语法规则中那样直接引用本规则符号来输出本规则的匹配内容。

QID : ID {System.out.println(**$QID**.text);} ;

但是 ANTLR 提示："rule QID has no defined paramters."错误无法生成代码。为什么 ANTLR 会提示规则 QID 没有定义参数呢？在词法规则中$QID.text 代表 QID 规则的参数 text。语法规则中的 Actions 中不能直接引用本规则符号。

那么词法规则的参数据如何定义呢？在第三章词法分析中的 fragment 用法中曾经讲到过只有用 fragment 定义的词法规则才可以加入参数，但 fragment 词法规则不能被语法规则直接引用也不会出现在语法树中。为了能加入词法规则参数我们将 ID 词法规则加上 fragment 关键字并添加两个参数。

```
grammar ActionInTokenRule3;
variable : QID;
QID : ID["x", "y"] {System.out.println($ID.text);} ;
fragment ID[String X, String Y] :
('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*
{System.out.println("X:" + $ID.X + " Y:" + $ID.Y);};
WS : ( ' ' | '\t' | '\r' | '\n' )+ { $channel = HIDDEN; } ;
```

$ID.X 用于引用参数 X，也可以直接使用 X。下面是生成的代码：

```
public final void mQID() throws RecognitionException {
   try {
         ......
      mID("x",  "y");
         ......
   }
   finally {
   }
}
public final void mID(String X, String Y) throws RecognitionException {
   ......
   System.out.println("X:" + X + " Y:" + Y);
}
```

词法规则不能定义返回值，因为语法分析程序是通过类似(IToken)input.LT(1)的语句获得的，所以对于语法规则直接引用的语法规则程序没有对办法传递自定义参数。而 fragment 语法规则不能让语法规则直接引用只能在词法规则之间引用，所以可以使用参数。每一个词法规则的信息都是用一个 Token 类来表示。

编译器在提示错误信息时都会提示错误的行号和列号，我们可以利用 Token 类的 Line 和 CharPositionInLine 属性输出字符在输入中的位置的。我们修改 5.13 全局 scope 属性的 SimpleGlobalScopes 示例，使能够输入变量的位置。

```
grammar SimpleGlobalScopes2;
......
variable
```

```
: type a=ID
{$CScope::symbols.add($type.text + " " + $a.text + " " + $a.line + ":" +
$a.pos);}
 (',' b=ID
{$CScope::symbols.add($type.text + " " + $b.text + " " + $b.line + ":" +
$b.pos);}
)* ';'
;
```

与 SimpleGlobalScopes 使用相同的输入，输出为：

```
func var = [int x 3:8, int y 6:12, int z 10:16]
global var = [int a 1:4, int b 1:6]
```

在语法规则和词法规则中出现的字符串常量定义，与普通的词法规则是一样的，只需定义变量，赋值后即可获得相关信息。

```
type : 'int' | 'float' ;

block
: begin='{' variable* stat+ '}' { System.out.println($begin.line);};
……
WS  @init{ a=0; }
: ( ' ' | '\t' | a='\r' | '\n' )+
{ System.out.println(a); $channel = HIDDEN; } ;
```

词法规则中也可以使用变量注意有些变量要初始化：

```
type : 'int' | 'float' ;
```

5.17 Actions 中变量引用总结

下面总结一下各种 Actions 中引用变量的写法：

```
parser grammar T;
@header {……}
@members {……}
scope scopeName {string X;}
rule1[int x] returns [int y]
scope{string Y;}
@init { }
@after {……}
: a=Rule2
{ scopeName::X; rule1::Y; $rule1.text; $Rule2.text; $a.text;
  scopeName[-i]::X; scopeName[i]::X;rule1::Y[-i]; rule1::Y[i];
};
catch[……] {
```

```
……
}
finally { …… }
Rule2 : Rule3 {$Rule3.text;};
fragment Rule3[string P] : b='Rule2' {$b.text; $Rule3.P;}
```

5.18 表达式求值示例

下面是一个有加减乘除四种运算的表达式文法，我们使用 Actions 让这个文法具有表示式求值的能力：

```
grammar E;
options { output=AST; }
expression : multExpr (('+' |'-' ) multExpr)*;
multExpr : atom (('*' | '/') atom)*;
atom : INT | '(' expression ')';
INT : '0'..'9' + ;
WS : (' ' |'\t' |'\n' |'\r' )+ {skip();};
```

下面是对表达式文法使用 Actions 加入求值功能后的文法：

```
grammar EEvalue;
options{language=Java;}
expression returns[int v]
: val1=multExpr {$v=$val1.v;}
  ((op='+' |op='-' ) val2=multExpr
  { if($op.text.equals("+")) {
      $v = $val1.v + $val2.v;
    } else if($op.text.equals("-")) {
      $v = $val1.v - $val2.v;
    }
  }
  )*
  ;

multExpr returns[int v]
: val1=atom {$v=$val1.v;}
  ((op='*' | op='/') val2=atom
  { if($op.text.equals("*")) {
      $v = $val1.v * $val2.v;
    } else if($op.text.equals("/")) {
      $v = $val1.v / $val2.v;
    }
  }
  )*
```

```
  ;
atom returns[int v]
: INT {$v=Integer.parseInt($INT.text); System.out.println("v:"+$v);}
  | '(' expression {$v=$expression.v;} ')';

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

修改后的文法中 expression、multExpr、atom 规则都具有返回值，规则中的 Actions 计算并返回本规则所表示的运算的结果值，下面的运行代码中最后输出启始 expression 规则的返回值(因为是单个返回值，所以规则函数直接返回整型值。

```
EEvalueLexer lexer = new EEvalueLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
EEvalueParser parser = new EEvalueParser(tokens);
System.out.println(parser.expression());
```

5.19 本章小结

本章讲述了嵌入到规则中的 Actions，分析器如何相对规则生成代码。全面地介绍了有关语法规则中 Actions 的用法。相关代码在生成代码后的位置，读都学习本章后也应该对 ANTLR 生成的代码结构有所了解。有了嵌入代码的辅助，大大增强了语法分析器的能力，分析器也变得灵活可定义。由于 ANTLR 还没有达到世界一流开发工具的严谨程度，所以有些时候查看 ANTLR 生成的代码对理解 ANTLR 是很有用的。