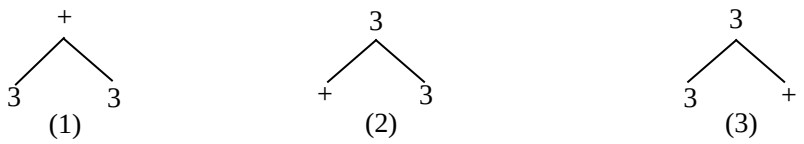
第六章 构造语法树

我们已经学习了词法分析、语法分析和嵌入规则的 Actions，现在我们可以定义文法分析输入的内容，并且可以在规则中嵌入 Actions 实现一些自己的操作，如：分析过程中提示语法错误，在分析过程中收集一些我们需要的信息。但是我们的分析器对输入的内容进行了分析后，还没有得到分析后的结果。上一章讲到可以利用 Actions 收集一些信息，但这些信息是分散的没有体系。比如我们分析 SQL 语句后收集了查询表和查询字段的信息，并分别放入了两个列表中，这时表与字段之间的所属关系在两个列表中体现不出来，不知道某个字段属于哪一个表。另外规则中的 Actions 也受执行时间的限制，Action 不知道在它执行之后发生了什么。比如分析编程语言时，先被分析的代码中可以引用了后面定义的类或方法，这时如何判断本句代码是否正确，这就需要在分析了全部内容之后才能判断 。Actions 只是小代码段在局部起作用，如果整个分析的结果的建立不应该用 Action 来实现，不使用 Actions 该如何获得语法分析的结果呢。我们需要一种能够既能表示符号信息又能表示语法结构的数据结构。这种结构在编译原理中就是语法树，树形的数据结构。

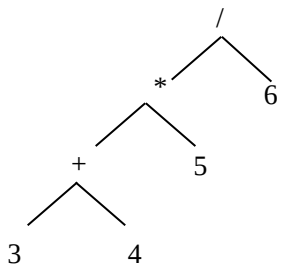
在第一章中我们介绍过语法树的概念，上一章也讲到过语法分析中的推导树与函数调用的流程相似的事实。一个规则推导出 0 到多个规则就像是一个树的节点拥有 0 到多个子节点。规则是推导树的模板，决定着推导树的结构，推导树是规则对于某种输入内容的体系结构的描述。语法树与规则的关系和推导树与规则的关系是非常类似的，如表达式 3 + 3 以“+”为根节点的语法树是图 5.1(1)所示的形式，语法树并没有规则定什么符号应该做根节点，图 5.1(2) (3)中是另外两种语法树的形式。但是由于语法树既要体现符号信息又要体现符号间语法结构的原故，图 5.1(1)以“+”作为根节点的形式是应该采用的形式，“+”作为根节点可以表示这棵子树是一个加算运算的子树，子节点就是操作数。这样的表示方法比较容易理解，程序操作起来也比较方便。

图 6.1



6.1 语法树的表示方法

图 5.1 表示出语法树的形式，使用图片固然可以很好地表示树形结构，但在文法无法使用。为了比较容易地书写和表示语法树结构，我们可以将语法树写成字符串的形式。下面介绍两种表式方法：第一种是表达式 3 + 3 可以写成 3 +^ 3 的形式，在“+”后面加入了一个“^”符号，“^”符号表示“+”作为树的根节点。没有“^”符号标记的符号为“+”的子节



点，要注意的是一棵树中不能有多个根节点。如果是更复杂的表达式比如：(3 + 4) * 5 / 6，应该写成((3 +^ 4) *^ 5) /^ 6

图 6.2

还有别一种语法树的表示方法，表达式 3 + 3 可以写成“+ 3 3”的形式。根节点的符号写在前面，其后的符号都属于它的子节点。字符串“- 3”可以代表一棵子树，根节点是“-”有一个子节点“3”。字符串“3”可以代表一棵子树，根节点为符号“3”没有子节点。我们用这种表示方法对更复杂的表达式比如：(3 + 4) * 5 / 6，进行改写，应该写成“/ (* (+ 3 4) 5) 6”的形式使用括号嵌套得将子树包括在“（）”内，这样就可以表示多层树的结构。由于语法树中的嵌套关系已经所映出运算的优先级，所表达式的中圆括号“（）”没有必要写出来。

由此可见在分析器程序在生成语法树时，主要要指出语法树的根节点并且要表示出语法树的层次结构。

6.2 默认情况下的语法树生成

让 ANTLR 生成语法树的方法我们曾在第一章讲到过，在文法的 options 设置中加入 output=AST 项目。下面看一个简单的表达式的文法 E[expression]。

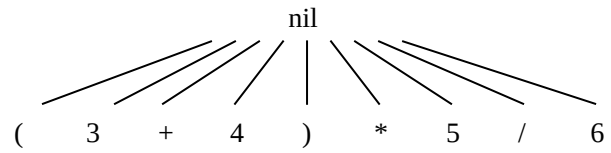
```
grammar E;
options { output=AST; }
expression : multExpr (('+' |'-' ) multExpr)*;
multExpr : atom (('*' | '/') atom)*;
atom : INT | '(' expression ')';
INT : '0'..'9' + ;
WS : (' ' |'\t' |'\n' |'\r' )+ {skip();};
```

生成分析器代码，使用如下代码来运行示例。

```
public class run
{
   public static void main(String[] args) throws Exception
   {
      ANTLRInputStream input = new ANTLRInputStream(System.in);
      ELexer lexer = new ELexer(input);
      CommonTokenStream tokens = new CommonTokenStream(lexer);
      EParser parser = new EParser(tokens);
      EParser.expression_return r = parser.expression();
      System.out.println(((BaseTree)r.getTree()).toStringTree());
   }
}
```

我们可以看到此分析器的 expression 规则函数返回 expression_return 类型的对象，expression_return 类是从 ParserRuleReturnScope 类继承的，就是上一章讲的规则返回值中规则函数返回的类型，不过 expression_return 类中添加了 getTree()方法用来获得语法树。toStringTree 用于将语法树以字符串的形式输出。由于我们没有对生成树的结构进行指定，所以 ANTLR 会按默认的方式生成语法树。

比如我们输入：“(3+4)*5/6”，输出：“(3 + 4) * 5 / 6”。图 5.3 表示了在默认的情况下语法树的结构。



下面我们再看一下有了设置项 output=AST 之后，生成分析器的代码有什么变化。下面是此文法生成的代码，我们只给出到 expression 规则为止的部分代码：

```
public class EParser extends Parser {
    ……
   protected TreeAdaptor adaptor = new CommonTreeAdaptor();
   public void setTreeAdaptor(TreeAdaptor adaptor) {
      this.adaptor = adaptor;
   }
   public TreeAdaptor getTreeAdaptor() { return adaptor; }
   public static class expression_return extends ParserRuleReturnScope {
      Object tree;
```

```
    public Object getTree() { return tree; }
};
public final expression_return expression() throws RecognitionException
                                    {
  expression_return retval = new expression_return();
   Object root_0 = null;
  Token set2=null;
  multExpr_return multExpr1 = null;
  multExpr_return multExpr3 = null;
   Object set2_tree=null;
  try {
     root_0 = (Object)adaptor.nil();
    ......
    multExpr1=multExpr();
     adaptor.addChild(root_0, multExpr1.getTree());
    do {
      ......
      switch (alt1) {
                  case 1 :
        { ......
              set2=(Token)input.LT(1);
                adaptor.addChild(root_0, adaptor.create(set2));
              ......
              multExpr3=multExpr();
                adaptor.addChild(root_0, multExpr3.getTree());
         } break;
      }
    } while (true);
     retval.tree = (Object)adaptor.rulePostProcessing(root_0);
     adaptor.setTokenBoundaries(retval.tree,
                          retval.start, retval.stop);
  }
```

```
        ……
    return retval;
}
```

ouput=AST 只影响语法分析类代码，在语法分析类 EParser.java 中增加了有关生成语法树的代码。首先增加了一个 CommonTreeAdaptor 类型的成员 adaptor。

public class CommonTreeAdaptor : BaseTreeAdaptor

CommonTreeAdaptor 类是 ANTLR 中默认的对语法树适配器实现类，它从 BaseTreeAdaptor 抽象类继承。adaptor 对象作为 EParser 的成员使语法分析类具有语法树适配器的功能。接下来 setTreeAdaptor 方法用于向语法分析器设置自定义的语法树适配器，我们可以自定义自己的适配器来替换默认的适配器实现一些自定义的语法树操作 。getTreeAdaptor 方法用来获得语法分析器的语法树适配器。

```
public static class expression_return extends ParserRuleReturnScope {
    Object tree;
    public Object getTree() { return tree; }
};
```

语法分析器类中定义了 expression_return 类，在上一章介绍过它的另一功能是用来返回规则的返回值。如果加入 output=AST 的设置后所有的语法分析器中的规则函数都会返回 XXX_return 类，目的是返回相应规则的语法树。

在 expression()规则函数中 multExpr (('+' |'-') multExpr)* 规则部分分别对应着 multExpr1=multExpr()、set2=(Token)input.LT(1)和 multExpr3=multExpr()语句。文法中第一个 multExpr 规则对应 multExpr1=multExpr()代码，multExpr()规则函数返回的是 multExpr_return 类的实例，multExpr_return 类与 expression_return 类是相同的概念。set2=(Token)input.LT(1)中利用 LT 方法获取当前的一个 Token 对象它对应的应该是“+”或“-”号，词法规则用 Token 类来表示。

前面讲到了规则是语法树的模板，ANTLR 在每一个在规则函数中都会生成与本规则相同的语法树，做为启始规则的 expression()返回的语法树就是整个输入生成的语法树，它是由 multExpr1=multExpr()、set2=(Token)input.LT(1)和 multExpr3=multExpr()这三个语法树组成的，每一个规则生成的语法树都是由下一级规则函数生成的语法树组成的，就这样一级一级从下向上生成了一棵完整的语法树。下面我们看一下 expression()函数中如何将 multExpr1、multExpr3 和 set2 组成语法树。

root_0 = (Object)adaptor.nil();

语法树适配器 adaptor 调用 nil 方法生成一个空的树节点作为根节点。在没有对语法树的结构进行指定时，ANTLR 使用一个空节点作为语法树的根节点。

adaptor.addChild(root_0, multExpr1.getTree());

语法树适配器 adaptor 调用 addChild 方法将 multExpr1 的语法树加到根节点中，addChild 方法有两个参数第一个参数是根节点，第二个参数是子树的根节点。multExpr3 的语法树也是用相同的方法加到 root_0 根节点的这里不在重叙。对词法规则的 Set2 的操

作与multExpr1和multExpr3有些不同，adaptor.addChild(root_0, adaptor.create(set2));中addChild的第二个参数使用了create方法。create方法可以根据Token对象生成语法树节点类型BaseTree。这样就可以将词法规则信息也加入到语法树中。

代码中adaptor对象还调用了rulePostProcessing方法，其作用是如果语法树是nil空节点作为根节点并且只有一棵子树的情况下返回这棵子树。因为在这种情况下nil根节点没有存在的意思。setTokenBoundaries方法用于设置一棵语法树的启止Token，也就是一棵语法子树对应Token流中的启止位置。

到此我们知道词法符号和语法符号都可以参与语法树的建立成为语法树的节点。成为语法根节点的符号不可以是一棵有子节点的子树。下面给出BaseTree的主要成员的定义。

```
public abstract class BaseTree : ITree
{
   protected internal IList children;
   public virtual void AddChild(ITree t);
   public virtual ITree GetChild(int i);
   public virtual void SetChild(int i, BaseTree t);
   public virtual string ToStringTree();
   public virtual int CharPositionInLine { get; }
   public virtual int ChildCount { get; }
   public virtual bool IsNil { get; }
   public virtual int Line { get; }
   public abstract string Text { get; }
   public abstract int TokenStartIndex { get; set; }
   public abstract int TokenStopIndex { get; set; }
   public abstract int Type { get; }
}
```

6.3 ^与！操作号

了解了语法树是如何创建之后，我们看一下在ANTLR中如何指定义语法树的结构。本章开始时讲述了两种表示语法树的方式，与第一种方法类似在ANTLR文法中可以使用“^”操作号来指定作为语法树的根节点的规则。“！”操作号用来指定不出现在语法树中的规则。“^”和“！”操作符作为后缀插入到规则的后面，请看下面文法：

```
grammar ETree1;
options { output=AST; }
```

```
expression : multExpr (('+'^ |'-'^ ) multExpr)*;
multExpr : atom (('*'^ | '/'^) atom)*;
atom : INT | '('! expression ')'!;
INT : '0'..'9' + ;
WS : (' ' |'\t' |'\n' |'\r' )+ {skip();};
```

运行代码如下：

```
import org.antlr.runtime.*;
import org.antlr.runtime.tree.*;
import org.antlr.runtime.ANTLRFileStream;

public class run
{
   public static void main(String[] args) throws Exception
   {
      ANTLRInputStream input = new ANTLRInputStream(System.in);
      ETree1Lexer lexer = new ETree1Lexer(input);
      CommonTokenStream tokens = new CommonTokenStream(lexer);
      ETree1Parser parser = new ETree1Parser(tokens);
      ETree1Parser.expression_return r = parser.expression();
      System.out.println(((BaseTree)r.getTree()).toStringTree());
      printTree((BaseTree)r.getTree());
   }
   static String pStr = "";
   public static void printTree(BaseTree tree) {
      pStr += "    ";
      System.out.println(pStr + tree.getText());
      for (int i = 0; i < tree.getChildCount(); i++) {
         BaseTree currTree = (BaseTree)tree.getChild(i);
         printTree(currTree);
      }
      pStr = pStr.substring(0, pStr.length() - 4);
   }
```

```
}
```

main 函数中调用了 printTree 函数用来按格式输出语法树的信息。printTree 使用递归从根节点遍历语法树。语法树节点的类型为 BaseTree 的派生类默认情况下为 CommTree 类，利用 getChild 方法可以获得树的某一个子节点。getChildCount 方法可以获得子节点的个数。

在命令行输入：

```
(3 + 4) * 5 / 6
^Z
```

命令行输出：

```
(/ (* (+ 3 4) 5) 6)
   /
      *
         +
            3
            4
         5
      6
```

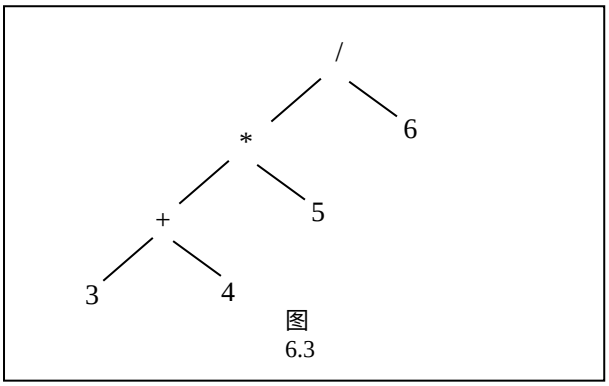
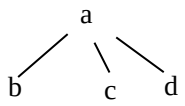


图 6.3

从输出中可以看出“/”作为语法树的根节点，树结构如图 6.3。括号没有出现在语法树中。下面我们仔细研究一下“^”操作符在语法树中的作用，下面是“^”功能的详细描述。

“^”操作符使节点在规则中成为子树的根在中，子树的高度增 1。如果下一个符号也标识了“^”操作符，则当前子树将成为下一个符号的第一个子树。如果“^”操作符标识的是一个规则，则规则只能返回一个符号，而不能返回一棵子树，此符号将成为根节点。“^”操作符不受子规则的限制。

```
grammar ETree2;
options { output=AST; }
mainrule : a^ b c d;
a : 'a';
b : 'b';
c : 'c';
d : 'd';
```



下面修改文法查看各种写法所生成树的情况：

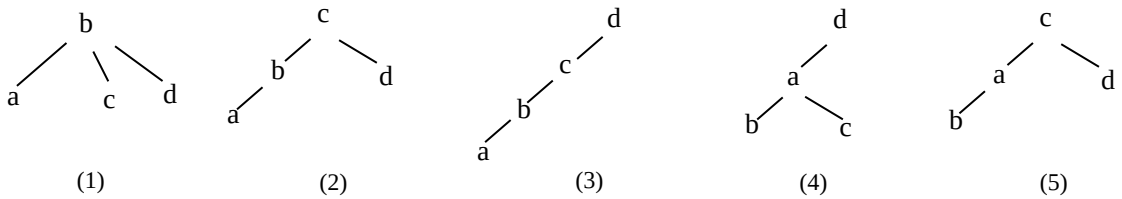


图 6.4

mainrule : a^ b^ c d; 图 6.4(1)

mainrule : a^ b^ c^ d; 图 6.4(2)

mainrule : a^ b c^ d; 图 6.4(3)

mainrule : a^ b^ c^ d^; 图 6.4(4)

mainrule : a^ b c d^; 图 6.4(5)

“^”操作符作用于一个规则中它不受子规则的影响。我们将上面的规则 mainrule:a b c d 加入子规则，看一下语法是否有变化。

mainrule : a^ (b^ c) d;

mainrule : a^ (b^ c d);

mainrule : (a^ b^ c) d;

加入子规则后这三个规则生成的语法树还是图 6.4(1)所示的结构。“^”操作符后缀的规则如果返回一棵子树而不是单一符号的情况，如下面的文法：

mainrule : a^ b^ c d;

a : c d;

b : 'b';

c : 'c';

d : 'd';

输入：cdbcd 分析器出现异常：“java.lang.RuntimeException: more than one node as root”。分析器不允许用一棵子树作为根节点。下面我们看一下加入“^”操作符后，生成分析器代码有什么变化。我们还是以 mainrule : a^ b^ c d;为例生成的代码如下：

```
public final mainrule_return mainrule() throws RecognitionException {
    mainrule_return retval = new mainrule_return();
    retval.start = input.LT(1);
    Object root_0 = null;
    a_return a1 = null;
    b_return b2 = null;
    c_return c3 = null;
```

```
 d_return d4 = null;
    try {
        root_0 = (Object)adaptor.nil();
        pushFollow(FOLLOW_a_in_mainrule19);
        a1=a();      _fsp--;
        root_0 = (Object)adaptor.becomeRoot(a1.getTree(), root_0);
        pushFollow(FOLLOW_b_in_mainrule22);
        b2=b();      _fsp--;
        root_0 = (Object)adaptor.becomeRoot(b2.getTree(), root_0);
        pushFollow(FOLLOW_c_in_mainrule25);
        c3=c();      _fsp--;
        adaptor.addChild(root_0, c3.getTree());
        pushFollow(FOLLOW_d_in_mainrule27);
        d4=d();      _fsp--;
        adaptor.addChild(root_0, d4.getTree());
        retval.stop = input.LT(-1);
        retval.tree = (Object)adaptor.rulePostProcessing(root_0);
        adaptor.setTokenBoundaries(retval.tree,
                                   retval.start, retval.stop);
    }  catch (RecognitionException re) {
        reportError(re);
        recover(input,re);
    }
    finally {    }
    return retval;
}
```

mainrule 规则函数的开始定义 a、b、c、d 四个规则的返回值对象。函数会先建立一个默认的 nil 的空根节点，这与以前没有什么不同。不同的是“^”操作符后缀的规则 a 和 b 对应 root_0 = (Object)adaptor.becomeRoot(a1.getTree(), root_0);语句。语法树适配器的 becomeRoot 方法的作用是用新的根节点替代旧的根节点，并将原根节点作为新的根节点的子节点，原根节为 nil 空根节点时抛弃。该方法有两个参数第一个参数为新根节点，第二个参数为原根节点。

象“+”这样的运算是从左到右的，如 3 + 4 + 5 先计算 3 + 4 语法树是如图 6.5(1)所示的语法树。但是有些运算从右到左的。如在 C 语言中幂运算 3^4，这里“^”表示 3 的 4 次幂。表达式 3^4^5 的运算顺序是从右到左先计算 4^5，语法树为图 6.5(2)所示的语法树。

图 6.5

这时对于幂运算规则，如果写成 3 '^'^ 4 '^'^ 5 时创建的语法树与“+”运算的结构相同，不能创建出从右到左的运算顺序的语法树。解决这个问题办法是使用右递归的定义方法来定义此规则，请看如下规则。

```
pow : INT ('^' ^ pow)? ;
```

右递归定义的 pow 规则使语法树只能向右子树发展。

6.4 重写规则（rewrite rule）

构造语法树的别一种方法是使用重写规则，重写规则是在规则定义之后重新书写关于语法树构造的部分。“^”符号和“!”符号夹杂在规则之中使规则的书写显得不够整洁。用重写规则来代替“^”符号方式可以提高文法的可读性，使文法变得列整洁干净。重写规则的写法与我们在本章开始时讲述的语法树表示方法的第二种类似，将根节点写在前面，子节点写在根节点之后。用“（）”来表示树的层次。下面给出了重写规则的书写形式，规则定义中每一个并列选择的分支都可以定义重写规则，定义了重写规则的分支分析器会按照重写规则指示的形式来生成语法树：

```
rule: alt1 -> rewriterule1
| alt2 -> rewriterule2
...
| altN -> rewriteruleN
;
```

下面我们使用变量定义的文法来演示重写规则如何构造语法树。

```
grammar ETree3;
options {output=AST;}
stat : variable assignState;
variable : type ID (',' ID)* -> ^(type ID+);
```

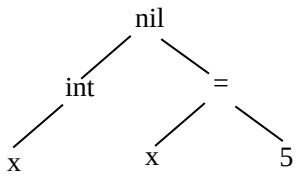
```
assignState : ID '=' INT -> ^('=' ID INT);

type : 'int' | 'float';

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

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

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

黑体字标识出了文法中的变化，文法中利用^(type ID)指定了 variable 规则构造以 type 为根节点以 ID 为子节点的语法树，利用^('=' ID INT)指定了 assignState 规则中构造以“=”为根节点 ID 和 INT 为子节点的语法树。运行些示例：

输入：int x; x= 5;

输出：

```
(int x) (= x 5)
   null
      int
         x
      =
         x
         5
```



利用“-> ^(type ID+)”重写规则很好地指定的语法树的结构并过滤掉了“，”，因为总体上 variable 规则至少要有一个 ID，所以使用了“+”表示 ID 的个数。按照惯例我们还是看一下文法生成的代码，看一下重写规则使生成的代码发生了什么变化。我们只给出 variable 规则的代码，黑体字标出了与重写规则有关的代码。

```
public final variable_return variable() throws RecognitionException {
    variable_return retval = new variable_return();
    retval.start = input.LT(1);
    Object root_0=null;
    Token ID4=null;
    Token char_literal5=null;
    Token ID6=null;
    type_return type3=null;
    Object ID4_tree=null;
    Object char_literal5_tree=null;
    Object ID6_tree=null;
    RewriteRuleTokenStream stream_7=
```

```
                new RewriteRuleTokenStream(adaptor,"token 7");
RewriteRuleTokenStream stream_ID=
                new RewriteRuleTokenStream(adaptor,"token ID");
RewriteRuleSubtreeStream stream_type=
                        new RewriteRuleSubtreeStream(adaptor,"rule
                            type");
  try {
     pushFollow(FOLLOW_type_in_variable25);
     type3=type();         _fsp--;
      stream_type.add(type3.getTree());
     ID4=(Token)input.LT(1);
     match(input,ID,FOLLOW_ID_in_variable27);
      stream_ID.add(ID4);
     loop1:
     do {
        int alt1=2;      int LA1_0 = input.LA(1);
        if ( (LA1_0==7) ) { alt1=1; }
        switch (alt1) {
           case 1 : {
              char_literal5=(Token)input.LT(1);
              match(input,7,FOLLOW_7_in_variable30);
                stream_7.add(char_literal5);
              ID6=(Token)input.LT(1);
              match(input,ID,FOLLOW_ID_in_variable32);
                stream_ID.add(ID6);
           } break;
           default :  break loop1;
        }
     } while (true);
     retval.tree = root_0;
     RewriteRuleSubtreeStream stream_retval=
      new RewriteRuleSubtreeStream
```

```
        (adaptor,"token retval",retval!=null?retval.tree:null);
      root_0 = (Object)adaptor.nil();
      Object root_1 = (Object)adaptor.nil();
       root_1=(Object)adaptor.becomeRoot(stream_type.nextNode(), root_1);
      if ( !(stream_ID.hasNext()) ) {
         throw new RewriteEarlyExitException();
      }
      while ( stream_ID.hasNext() ) {
          adaptor.addChild(root_1, stream_ID.next());
      }
      stream_ID.reset();
      adaptor.addChild(root_0, root_1);
    retval.stop = input.LT(-1);
    retval.tree = (Object)adaptor.rulePostProcessing(root_0);
    adaptor.setTokenBoundaries(retval.tree, retval.start, retval.stop);
  }.....
  return retval;
}
```

从代码中可以看到重写规则不同于使用“^”符号直接用 adapter 建立语法树，而是先或收集语法树的节点到 RewriteRuleTokenStream 类或 RewriteRuleSubtreeStream 对象中（这些节点可能是语法规则返回的子树，也可能是词法规则返回的 Token 对象），然后再将这些节点建立成一棵语法树。

RewriteRuleTokenStream 类与 RewriteRuleSubtreeStream 类都是从 RewriteRuleNode Stream 继承的用于收集语法树节点的类。我们在第五章嵌入规则的 Actions 一节中讲到过生成的分析器代码结构与规则定义有对应关系，是有规律有寻的。根据规则定义形式的不同，生成语法分析器代码也会有不同的结构（分支或循环结构），使用“^”符号建立语法树时，建立语法树的过程是在语法分析代码的执行过程中进行的，在语法分析的同时建立语法树，所以建立语法树的代码比较简单。但使用重写规则建立语法树时创建语法树的过程与语法分析过程是分开的，在语法分析程序执行完之后才开始建立语法树。这样就需要在语法分析时事先收集树节点，放到 RewriteRuleNodeStream 类型的对象中。语法分析完成后，再进行语法树的建立工作。本示例中的 stream_7.add、stream_ID.add 和 stream_type.add 分别收集“,”的 Token 对象、ID 的 Token 对象和 type 子树。

在 variable 规则中 type 为语法规则，ID 和“，”为语法规则，RewriteRuleSubtree Stream 用于收集语法规则子树，所以 type()返回的子树使用

RewriteRuleSubtreeStream 的对象 stream_type 来收集。ID 和“，”对应的 Token 对象使用 RewriteRuleTokenStream 的对象 stream_7 和 stream_ID 来收集。

我们可以看到 variable 规则中的(',' ID)*重复定义对应生成了循环结构的代码，在循环 stream_7.add(char_literal5)和 stream_ID.add(ID6)语句将 0 到多个 ID 与“，”收集到 stream 类中，这正体现了 RewriteRuleNodeStream 收集的过程和原因。

收集了节点信息后 adaptor 调用 becomeRoot 方法使 type 成为根节点，利用 stream_type 类的 nextNode()方法可以取出收集到的 type 节点 type3。nextNode()方法会确保取出的是一个节点而不是有子节点的树，原因前面已经讲过。接下来在一个循环中将收集到的 ID 信息节点作为 type 的子节点加入到语法树中 stream_ID 类的 next()方法可以取出收集到的 ID 节点 ID6 对象，next()方法与 nextNode()方法不同的是 next 方法可以返回有子节点的树。

6.6 重写规则的使用

下面我们来详细研究一下重写规则的用法和技巧。

6.6.1 放弃节点

重写规则为用户提拱了一个重组语法树的机会，我们可以挑选需要的节点建立语法树，不需要的节点自然就会被放弃。请看下面的文法：

```
variable : type ID (',' ID)* -> ^(type ID+); //放弃“,”符号
prog : stat ';' -> stat; //放弃“;”符号
expr : '(' expr ')' -> expr; //放弃“()”符号
```

也可以让规则不返回语法树：

```
nullstat : ';' -> ;
```

6.6.2 重组语法树

使用重写规则建立语法树的同时可以按照需要重新排列元素的顺序，下面的文法显示了将 Pascal、Basic 和 C 风格的变量定义规则生成相同形式的语法树。

```
decl : 'var' ID ':' type  -> type ID ;
decl : 'dim' ID 'As' type -> type ID ;
decl : type ID            -> type ID ;
```

6.6.3 指定根节点和树的层次

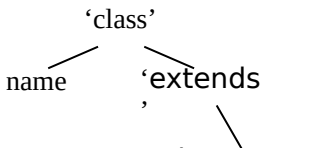
指定语法树的根节点是生成语法树的关键。使用重写规则指定根节点的方法是将所有节点用括号括起来，根节点写在最前面，括号的前面用“^”符号标识形式为^(root node node…)。请看下面的文法：

stat : 'return' expr ';' -> ^('return' expr) ;//使用 return 关键字做为根节点。

decl : 'Dim' ID 'As' type -> ^('Dim' type ID) ;//使用 Dim 作为根节点。

^(…)形式可以嵌套定义来指明语法树层次结构。请看下面一个定义类声明的文法，继承基类作为子树建立。

```
classDef
: 'class' name 'extends' basename
-  >  ^('class'  name  ^('extends'  basename));
```



规则定义中的“+”、“?”和“*”等操作符同样可以在重写规则中使用。它们代表相同的意思。下面给出一个更加贴近实际的类这义规则：

```
classDef : 'class' ID ('extends' supernName)?
('implements' interfaceName (',' interfaceName)*)?
->  ^('class'  ID  ^('extends'  supernName)?  ^('implements'  interfaceName+)?
    variableDefinition* ctorDefinition* methodDefinition*
  )
```

有些时候我们完全可以把规则拷贝到重写规则这边来，或做一些小修改就可以实现重写规则的语法树的建立。

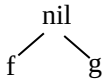
6.6.4 规则中的重写规则

重写规则可以写在选择的分支内。重写规则在分支中对其所在分支起构造语法树的作用，没有重写规则的分支语法树将按照默认方式进行构造。我们可以在不同的分支中使用不同的构造方法。请看下面的文法

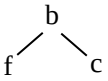
```
grammar mutlRewriteRule;
options { output=AST; }
a : b c -> ^(b c)
  | d e^
```

```
    | f g;
b : 'b';
c : 'c';
d : 'd';
e : 'e';
f : 'f';
g : 'g';
```

运行后分别输入：“bc”，“de”，“fg”，生成如下的三种语法树。



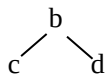
重写规则也可以写在子规则中，子规则中的重写规则可以创建将子规则以及子规则之前的规则符号创建成语法树。请看下面的规则：




```
a : b (c d -> ^(b c d)) ;
```

由于子规则可能在规则（）的中间位置，重写规则也就可以写在规则的中间。在规则中间的重写规则只能使用重写规则之前的规则符号建立语法树。有前面的对关于重写规则生成代码的理解，理解这一点并不难，因为重写规则创建语法树时在重写规则之后的规则符号还没有被收集，也就不可能加入到语法树中。请写下面的规则：

```
a : b (c d -> ^(b c d)) e;
```



修改上面的示例：

```
a : 'a' (b c -> ^(b c)
      | d e^
      | f g);
```

6.6.5 收集节点统一输出

前面的示例已经展示了，对于输入中的相同类型的节点可以在重写规则中统一处理。这种方式使构造语法树的过程更加的直观整洁。

```
variable   : 'var' ID (',' ID)* ':' type  -> type ID+ ;
```

需要注意的是重写规则中的符号的多重性要与规则中对应符号的多重性相符。要根据

符号可能出现的个数正确地使用“+”，“*”符号。修改上面的规则使 ID 的多重性是*，这样写是不很正确的，但并不会造成执行错误：

variable : 'var' ID (',' ID)* ':' type -> type ID* ;

但是如果规则 decl : ‘var’ ID* -> type ID+; 当输入“var ”时则会出现 RewriteEarlyExit Exception 异常。

6.6.6 重复与分散节点

有时在构造语法树时需要将输入的单一节点复制多份在语法树上。这时使用重写规则非常容易实现，只需把同一符号写多次就可以了。请看下面 variable 规则将第一个 ID 都独立生成一棵语法树，对于每一个 ID 都配备一个 type 根节点，type 复制了与 ID 相同的个数。

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

下面看一下生成代码：

```
public final variable_return variable() throws RecognitionException {
    ......
    try {
       type1=type(); //type
       stream_type.add(type1.getTree());
       ID2=(Token)input.LT(1); //ID
       stream_ID.add(ID2);
       do {
          ......
          char_literal3=(Token)input.LT(1);//“,”
          stream_7.add(char_literal3);
          ......
          ID4=(Token)input.LT(1); //ID
          stream_ID.add(ID4);
       } while (true);
       if ( !(stream_ID.hasNext() || stream_type.hasNext()) ) {
          throw new RewriteEarlyExitException();
       }
       while ( stream_ID.hasNext() || stream_type.hasNext() ) {
```

```
            Object root_1 = (Object)adaptor.nil();
                                                    root_1 =
                                (Object)adaptor.becomeRoot(stream_type.n
                                extNode(), root_1);
            adaptor.addChild(root_1, stream_ID.next());
            adaptor.addChild(root_0, root_1);
          }
        stream_ID.reset();
        stream_type.reset();
          ......
      }
      ......
      return retval;
  }
```

从代码中可以看出 type 节点从输入中收集一次，但在生成语法树时与 ID 一起循环了多次，被复制了与 ID 相同的个数。

6.6.7 重写规则中的 Actions

重写规则中也可以插入 Actions 主要可以用来在重写规则中引用其它规则、引用变量、建立自定义的节点的功能。

6.6.7.1 重写规则中引用变量

在重写规则中可以引用变量，在第五章讲到的变量我们可以在重规则中引用就象使规则名称一样。6.5.5 节中的规则 variable 修改成如下的写法我们可以得到同样的语法树。

```
variable : t=type ids+=ID (',' ids+=ID)* -> ^($t $ids+) ;
```

下面是一个建立类语法树的文法示例。语法树的以类名为根节点，假设类中只能有一个构造函数，我们把构造函数找出并作为第一个子节点。

```
grammar constructorNode;
options { output=AST; language=Java;}
classDef
@init {
```

```
  Object constructor = null;
}
: 'class' className '{'
  (methodName
  { if($methodName.text.equals($className.text))
       constructor = $methodName.tree;
  }
  )*
  '}' -> ^('class' {constructor} methodName+) ;
className : ID;
methodName : ID;
ID : ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')* ;
WS : ( ' ' | '\t' | '\r' | '\n' )+ { skip(); };
```

输入：

```
class cls1 {
  getA
  getB
  cls1
  getC
}
^Z
```

输出：

```
(class cls1 getA getB cls1 getC)
   class
      cls1
      getA
      getB
      cls1
      getC
```

语法规则$methodName.tree 是一个 CommTree 类型的对象，重写规则在建立语法树时 adaptor.AddChild 方法将收集的 CommTree 类型的对象加入语法树中，了解这个情况后我们可以将$methodName.tree 赋予变量 constructor，在重写规则建立语法树时将 constructor 加入到语法树中。在重写规则中加入的 Actions“{}”内的内容将被放入 adaptor.AddChild()的“（）”之内。也就是说在重写规则中 Actions 中提拱的是 AddChild 参数。

6.6.7.2 参数传递语法树

可以将规则返回的语法树作为参数传递给其它规则。请看下面的示例，类和接口定义中修饰符是写在前面的，由于 class 和 interface 都有共同的修饰符，所以在规则中修饰

符不包括在类和接口的定义中，添加一个 typeDefinition 规则，使修饰符 modifiers 规则可以单独定义。但是类和接口的语法树中需要修饰符的信息，modifiers 不在 classDefinition 和 interfaceDefinition 规则中无法直接将 modifiers 子树加入到类和接口的语法树中，这时可以使用传参数的方法，将修饰符 modifiers 规则返回的语法树作为参数传递给 classDefinition 和 interfaceDefinition 规则，然后作为子树建立到语法树中。

```
typeDefinition
: modifiers! classDefinition[$modifiers.tree];
| modifiers! interfaceDefinition[$modifiers.tree];

classDefinition[CommonTree mod]
: 'class' ID ('extends' superClass=typename)?
('implements' baseClasses+=typename (',' baseClasses+=typename)*)?
'{'
   ( variableDefinition | methodDefinition | ctorDefinition )*
'}'
-> ^('class' ID {$mod} ^('extends' superClass)? ^('implements'
                                    $baseClasses+)?
variableDefinition* ctorDefinition* methodDefinition*
)
;
```

6.6.7.2 自已创建树节点

前面讲过重写规则中的 Actions“{}”中插入的代码是语法树节点，它将作为 adaptor.AddChild()的“（）”之内的参数。了解这个情况之后我们就可以在“{}”中插入创建节点的代码。请看下面的规则：

```
nullStat : SMEI -> {new CommonTree(new CommonToken(SMEI,
"NULLStatement"))} ;
SMEI : ';'
```

这个重写规则的作用是当遇到空语句时向语法树中建立 NULLStatement 节点，语法树的节点的类型是 CommonTree 类，CommonTree 类可以用一个 Token 对象来创建。而 CommonToken 类可使用两个参数来构造，第一个是 Token 对应的类型值，它是一个代表要建立 Token 对象类型的整型值，SMEI 在语法分析器的代码中是一个常量，你可以在生成的分析器代码中找到。第二个参数是生成的 Token 对象的 text 属性值，text 值将成为树节点的 text 属性值。

6.6.7.3 引用前一规则的变量

我们在本章 6.2 ^ 与！操作号一节中的 ETree1 [expression]示例中使用“^”很容易地构造了四则运算的语法树。但是我们如何用重写规则来建立示例的语法树呢？到现在为止我们做不到这一点，对于 expression 规则和 multExpr 规则来说创建语法树需要本规则的上一次分析的语法树结果。如果把 expression 规则和 multExpr 规则写成右递归的话就比较好理解。

expression : multExpr (('+' |'-') multExpr)*;

等价于：

expression : multExpr | **expression** ('+' |'-') multExpr;

//此写法只为说明问题在 ANTLR 中是非法的。

当 expression 匹配 expression ('+'|'-') multExpr 分支时，很显然建立语法需要先前对 expression 返回语法树。重写规则可以利用对规则的引用来实现这个功能。请看下面的规则：

expression : multExpr | expression '+' multExpr

-> ^('+' $expression multExpr); //此写法只为说明问题 在 ANTLR 中是非法的。

根据这规律我们返回到先前的写法，请看下面的规则：

expression : (multExpr **-> multExpr**) ((**opter=**'+' | **opter=**'-') **rightN=**multExpr

-> ^(**$opter $expression $rightN**))*;

这个规则可以正确生成语法树，规则的第一个 multExpr 生成的语法树与本身相同只有一个 multExpr 子树，规则后半部的(('+' |'-') multExpr)*;首先将“+”和“-”赋于 opter 变量中，opter 变量为 Token 类型。由于重写规则中不允许用形如('+'|'-')的子规则作为根节点，所以使用一个$opter 代表根节点。处理第二个 multExpr 时为了在建立语法树时与第一个 multExpr 冲突，将第二个 multExpr 赋予 rightN 变量中。然后在(('+' |'-') multExpr)*;子规则中加入了->^($opter $expression $rightN)建立语法树，其中 $expression 代表 expression 规则前一次的生成的语法树结果，前一次生成的语法树成为了当前语法树的左子树，这是一个迭代的过程。

其中需要注意的是第一个 multExpr 要使用重写规则(multExpr **-> multExpr**)即使看起来这个重写规则好像没什么必要，如果没有第一个 multExpr 将不会创建语法树，这是因为 expression 规则后半部分已经有了重写规则，如果第一个 multExpr 没有重写规则则第一个 multExpr 将被忽略掉。

6.7 建立虚拟节点

在语法树中建立虚拟节点很重要，只是输入的符号并不能很好的描述一个语法子树代表的意思，很多时候需要加入一个代表抽象意思的虚拟节点作为根节点，使容易理解整个语法子树的意义。请看下面的文法：

```
grammar ERewriteRule;

options { output=AST; language=CSharp;}

tokens {

   EXPER;

}

expression : (multExpr -> multExpr) ((opter='+' |opter='-' )
                                    rightN=multExpr

        -> ^(EXPER $expression $opter $rightN) )* ;


multExpr : atom (('*' | '/') atom)*;

atom : INT | '(' expression ')';

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

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

建立虚拟节点很简单首先在 options 设置项下面添加 tokens{…}节，其中加入需要的虚拟节点标识符，标识符必须以大写字母开头。由于语法规则小写开头，词法规则大写开头，为了便于区分一般虚拟节点标识符用全大写表示。

运行后输入：

```
3 + 4 + 5
```

输出：

```
(EXPER (EXPER 3 + 4) + 5)
   EXPER
      EXPER
         3
         +
         4
      +
        5
```

EXPER 作为根节点后，在遍历语法树时遇到 EXPER 节点后就可以知道此子树是一个表达式的语法树，这样语法树变得更实用。

使用 Antlr 生成代码后会出现一个.token 文件，其中存放了所有词法符号和虚拟节点符

号以及它们对应的整型值。其它分析器可以共享这些符号，我们将在下一章讲解符号的共享。上面的 ERewriteRule[expression]示例中 EXPER 节点的 text 属性值句节点句相同为“EXPER”。我们下面将讲解如何设置虚拟节点的 text 属性值。

作为语法树节点的 BaseTree 类存放了行、列等信息，可以获得节点对应输入的位置，这在报告错误信息时很有用。但我们建立的虚拟节点没有这些信息。可以通过下面的方法将这些信息加入到虚拟节点中。

```
expression : (multExpr -> multExpr) ((opter='+' |opter='-' )
                                     rightN=multExpr
                -> ^(EXPER[$opter, "EXPER"] $opter $expression $rightN) )* ;
```

重写规则中的 EXPER 虚拟节点可以设置参数。第一个参数$opter 代表“+”、“-”符号，这可以让 EXPER 具有$opter 所属性的信息包括行、列信息。第二个参数是设置节点的 Text 属性，这样就可以用 Line 和 CharPositionInLine 属性来获得语法树的行列信息。

```
ERewriteRuleParser.expression_return r = parser.expression();
System.Console.WriteLine(((BaseTree)r.Tree).getLine() + ", "
                         ((BaseTree)r.Tree).getCharPositionInLine());
```

虚拟节点的创建有多种用法，重写规则中 EXPER[$opter, "EXPER"]对应生成的代码为：

```
adaptor.Create(EXPER, opter, "EXPER")
```

CommAdaptor 类的 Create 方法可以根据 Token 对象或 Token 类型创建 CommTree 树节点。创建虚拟节点时一共有如下四种方法：

EXPER	adaptor.Create(EXPER, "EXPER")
EXPER[]	adaptor.Create(EXPER, "EXPER")
EXPER[$opter]	adaptor.Create(EXPER, opter)
EXPER[$opter, "EXPER"]	adaptor.Create(EXPER, opter, "EXPER")

6.8 自定义语法树

ANTLR 中允许用户自定义自己的语法树。因为语法树是由节点组成的，所以自定义语法树就是自定义的语法树的节点类型。前面我们讲过生成语法树的分析器的代码，其中有一个 setTreeAdaptor 方法（在.net 中是 TreeAdpator 属性）可以设置分析器的语法树适配器类。由于语法树适配器用于创建语法树节点，所以我们可以设置自己的语法树适配器类从而创建自己树节点。第一步我们要创建一个自己的语法树适配器类，为了让示例简单我们的适配器类从 CommTreeAdaptor 类继承。请看下面的 java 代码：

```
import org.antlr.runtime.*;

import org.antlr.runtime.tree.*;

public class MyTreeAdaptor extends CommonTreeAdaptor {
```

```
    public Object create(Token token) {
        return new MyTree(token);
    }
}
```

MyTreeAdaptor 从 CommonTreeAdaptor 继承并重载了 create 方法（.net 中为 Create），用一个 Token 对象作为参数，这样 create 方法可以根据 token 对象创建树节点。在 create 方法中创建了 MyTree 类型的节点。下面是 MyTree 类代码：

```
import org.antlr.runtime.*;
import org.antlr.runtime.tree.*;

public class MyTree extends CommonTree {
    public String getLongText() {
        return _getLongText(this);
    }
    private String _getLongText(BaseTree bt) {
        String longT = "";
        if(bt.getChildCount() == 0) {
            return bt.getText();
        } else {
            for (int i = 0; i < bt.getChildCount(); i++) {
                BaseTree childTree = (BaseTree)bt.getChild(i);
                longT += " " + _getLongText(childTree);
            }
        }
        return longT;
    }
    public MyTree(Token t) {
        super(t);
    }
```

```
}
```

MyTree 类继承 CommTree 类，这也是为了让示例简单。我们可以直接继承 BaseTree 类。为了让 MyTree 树节点有意义，定义了 getLongText()方法用来获得相前节点对应输入的全部内容，有些时候这是很实用的当访问根节点时就可以知道整个子树的内容。如调用 EXPR 节点的 getLongText()方法可以获得整个表达式的内容“3+4+5”。在运行分析器的语句中，将我们的 MyTreeAdaptor 类设置到语法分析对象中。

```
ERewriteRuleLexer lexer = new ERewriteRuleLexer(input);

CommonTokenStream tokens = new CommonTokenStream(lexer);

ERewriteRuleParser parser = new ERewriteRuleParser(tokens);

MyTreeAdaptor adaptor = new MyTreeAdaptor();

parser.setTreeAdaptor(adaptor);

ERewriteRuleParser.expression_return r = parser.expression();

System.out.println(((MyTree)r.getTree()).getLongText());
```

输入：3+4+5 输出：3 + 4 + 5

6.9 小结

本章我们学习了 ANTLR 如何控制生成语法树。ANTLR 可以按照使用者的意愿生成语法树，主要有两种方法，使用“^”和使用重写规则。重写规则是主流方法，并可以加入虚拟节点使语法树更具有实用意义。语法树是语法分析是结果和最终目的，下一章我们将学习遍历语法树方法，如何利用语法分析的结果。