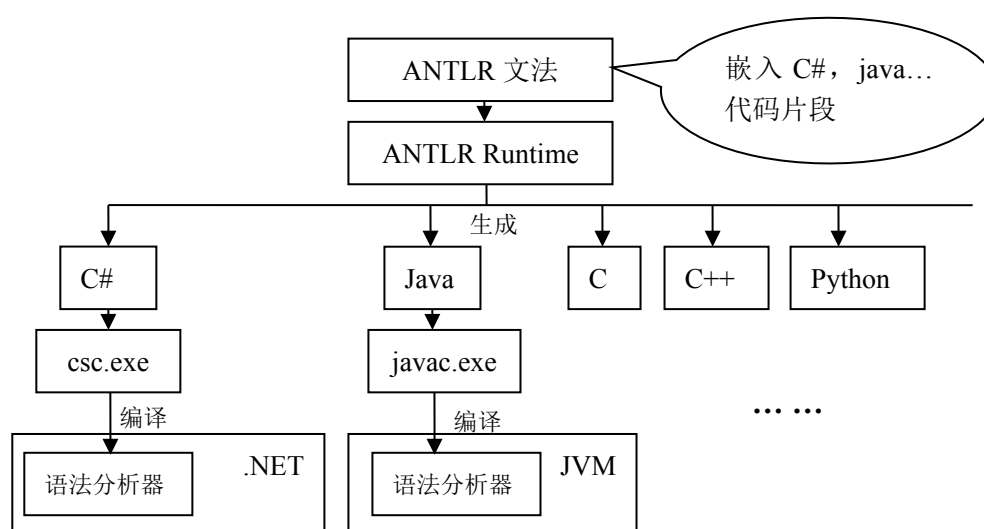


第一章 Hello World

ANTLR 是 ANother Tool for Language Recognition 的缩写“又一个语言识别工具”，读['æntlə]。从名字上可以看出在 ANTLR 出现之前已经存在其它语言识别工具了（如 LEX¹，YACC²）。ANTLR 的官方定义为：根据一种可以嵌入如 Java，C++或 C#等辅助代码段的文法，来构筑出相对该文法的识别器，编译器或翻译器的一种语言工具框架。这个定义说明了 ANTLR 的功能是根据给定文法自动生成编译器，其过程为先编写相应语言的文法然后生成相应语言编译器。定义提到的语言识别器，编译器和翻译器我们以后统称为语法分析器。事实上 ANTLR 是生成相应语言编译器的源代码，我们还需要编译它。那么 ANTLR 可以生成哪些语言的语法分析器源代码语言的代码呢？这是程序员很关心的问题。幸运的是 ANTLR 现在已经支持了多种当前流行的开发语言，包括 Java、C#、C、C++、Objective-C、Python 和 Ruby. 1 等。你可以根据需要生成其中任何一种语言的语法分析器。本书主要介绍 java，C# 两种语言，有详细的操作步骤包括如何编译、执行和如何使用 ANTLRWorks 开发环境编写文法等。读者可以顺利上手，避免实际操作障碍。后面章节还会指出在 Java 和 C#开发中应注意的细微差别，确保程序的顺利运行。



1.1 开发 Hello World 示例

本章将开发一个简单示例让读者对 ANTLR 有一个初步的认识，并搭建开发环境以便后续

¹ 一种词法分析器（分词器）的自动产生系统，用正则表达式来描述文法结构。

² 一种语法分析程序的自动产生器，可以处理能用 LALR(1)文法表示的上下文无关语言。

的学习。读者在示例中遇到不懂的地方也不必担心，我们的目的是搭建开发环境学会编译运行语法分析器。用 ANTLR 开发一个语法分析器大致分三步，第一步：写出要分析内容的文法。第二步：用 ANTLR 生成相对该文法的语法分析器的代码。第三步：编译运行语法分析器。

和多数编译书籍一样，本章也用解析简单的表达式作为示例。要解析的表达式中有二种数据类型：整数 如 “23”，“5” 和字符串 如 “Hello World”。表达式中以算术表达式为主也包括赋值表达式。我们列举两个表达式语句：

`23+4*(5+1);` `str= “Hello World” ;`

第一条语句是一个算术表达式，括号改变了运算顺序，计算结果不赋给任何变量。第二条是一个赋值表达式，将字符串赋给一个变量。后面我们要开发一个语法分析器来分析这两条语句。在开发之前先简单提一下语法树的概念，在语法分析中一般用树来表示语法结构，表达式的语法树是以操作符为根节点操作数为子节点的树形结构，`23+4*(5+1)` 的语法树根据图 1.1 所示的操作符的优先级如下。

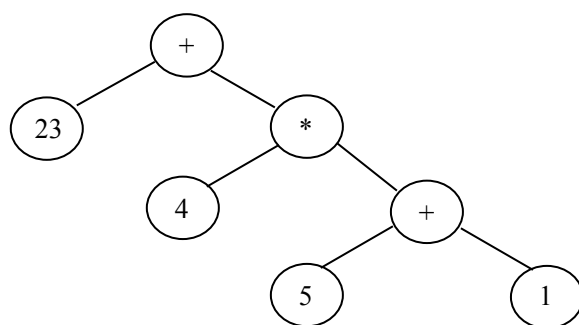


图 1.1

算术表达式先计算 `5+1`，`5+1` 在括号中操作符的优先级最高在语法树中的深度最大，然后是 `4*(5+1)`，最后是 `23+4*(5+1)`。可以看出语法树的求值顺序是从下向上的，先计算深度大的操作符 `5+1` 结果为 6，然后是 `4*6` 结果为 24，然后是 `23+24` 表达式的结果为 47。下面再看一下赋值表达式的语法树结构：

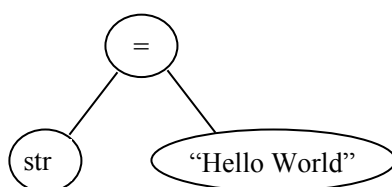


图 1.2

赋值操作符 “=” 做为根节点变量 `str` 作为左子树，而字符串表达式 “Hello World” 作为右子树。了解了语法树后我们开始录入文法源文件。ANTLR 中文法文件是扩展名为 “.g” 的文本文件，“.g” 文件就是我们的源文件。这里新建一个叫 “E.g” 的文法文件，在文件中输入如下文法定义：

```

grammar E;
options{ output=AST;}

program : statement + ;

statement: (expression | VAR '=' expression) ';' ;

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

multExpr : atom ('*' atom)*;

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

VAR : ('a'..'z' | 'A'..'Z' )+ ;

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

STRING : '"' (('A'..'Z' | 'a'..'z' | ' ') +) '"' ;

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

```

文件的第一行 `grammar E` 的 `E` 为文法的名称它与文件名一致。第二行是文法的设置部分 `options{ output=AST;}`，`output=AST` 表示让语法分析器返回包含语法树的信息。第三行开始是文法定义部分，文法是用 EBNF¹ 推导式来描述的（有关 EBNF 会在后面章节中讲解），文法定义中分两大部分以小写字母开头的语法描述和全大写的词法描述。其中每一行都是一个规则（rule）或叫做推导式、产生式，每个规则的左边是文法中的一个名字，代表文法中的一个抽象概念。中间用一个 “:” 表示推导关系，右边是该名字推导出的文法形式。下面逐一介绍文法的规则定义：

```
statement : (expression | VAR '=' expression) ';' ;
```

`statement` 代表表达式语句，前面说了语句有两种，在推导式中以 “|” 分隔代表并列可选的关系。表达式本身是合法的语句，表达式也可以出现在赋值表达式中组成赋值语句，两种语句都以 “;” 字符结束。

```
expression: (multExpr (('+' | '-' ) multExpr)*) | STRING
```

`expression` 代表表达式，第二个 “|” 的左边是算术表达式的形式，第二个 “|” 的右边是字符串表达式。我们通过规则的推导顺序可以看出，规则按操作符的优先级首先推导优先级最低的运算 “+”，“-” 运算。

```
multExpr : atom ('*' atom)*;
```

然后是 “*” 的运算。表达式中没有除法运算。

¹ EBNF: BNF 是被用来形式化定义语言的语法，以使其规则没有歧义。EBNF 在 BNF 基础上改进了定义形式比 BNF 更写法更简洁。

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

最后是“()”运算，括号中又可以是一个表达式，这样也就实现了括号的嵌套关系。以“|”分隔与括号并列的会出现参与运算的整型数。

```
VAR : ('a'..'z' | 'A'..'Z') + ;  
INT : '0'..'9' + ;  
STRING : '"' (('A'..'Z' | 'a'..'z' | ' ') +) '"' ;  
WS : (' ' | '\t' | '\n' | '\r') + {skip();} ;
```

以大写形式表达的是词法描述部分，VAR 表示变量由一个或多个大小写字母组成，INT 表示整型，整型由一个或多个 0 到 9 的数字组成，STRING 表示字符串，和变量类似一个或多个大小写字母组成但要用“”括起来。WS 表示空白，它的作用是可以滤掉空格、TAB、回车换行这样的无意义字符。{skip();}的作用是跳过这些字符。

1.2 下载 ANTLR

本章我们的目的是运行第一个示例，后面章节会详细介绍文法定义的写法，所以暂时有不清楚的地方不必担心。这里应该注意的是：文法中单词的大小写，ANTLR 文法是大小写敏感的，文法名称要和文件名一致。下面我们用 ANTLR 生成该文法的 java 分析器代码。我们先下载 ANTLR 的 Runtime 和开发环境，到 <http://www.antlr.org/download.html> ANTLR 的下载页，www.antlr.org 为 ANTLR 的官方网站，ANTLR 是一个开源项目可以免费下载。图 1.1 为开发环境 ANTLRWorks 的下载页面，图 1.2 为开发环境 ANTLR Runtime3.01 的下载页面。您的机器上需要安装 JDK1.5 或更高版本的 java SDK。其中 ANTLRWorks 的下载文件名叫 antlrworks-1.1.7.jar 安装 JDK 后可以直接双击运行打开 ANTLRWorks 开发环境。

ANTLR v3

[Home](#)
[Download](#)
[ANTLRWorks](#)
[News](#)
[Using ANTLR](#)
[Documentation](#)
[FAQ](#)
[Articles](#)
[Grammars](#)
[File Sharing](#)
[Runtime API](#)
[Tech Support](#)
[Bug Tracking](#)
[About ANTLR](#)
[What is ANTLR](#)
[Why use ANTLR](#)
[Showcase](#)
[Testimonials](#)
[Getting Started](#)
[Software License](#)
[ANTLR WebLogs](#)
[ANTLR Workshops](#)
[StringTemplate](#)


SEARCH

ANTLR Software Download

Currently available with C, C#, and Java targets; other [targets under development](#).

ANTLRWorks+ANTLR

ANTLRWorks is a GUI development environment for building ANTLR v3 grammars. It is a stand-alone Java application that you can just click on to start using ANTLR. It contains all necessary jars and is the easiest way to get started using ANTLR.



Note: ANTLRWorks requires Java 1.5 or later to run

- [Version 1.1.7 - for Windows, Linux and Mac OS X](#) ←
- [Version 1.1.7 - bundled for Mac OS X](#)
- [Version 1.1.7 - source code \(BSD license\)](#)
- [ANTLRWorks IntelliJ plugin](#)
You can also install plugin directly using the plugin manager in IntelliJ.

On Mac OS X you can just click on that jar and it will start up the GUI. Same for Windows

(图 1.3) ANTLRWorks 下载页面

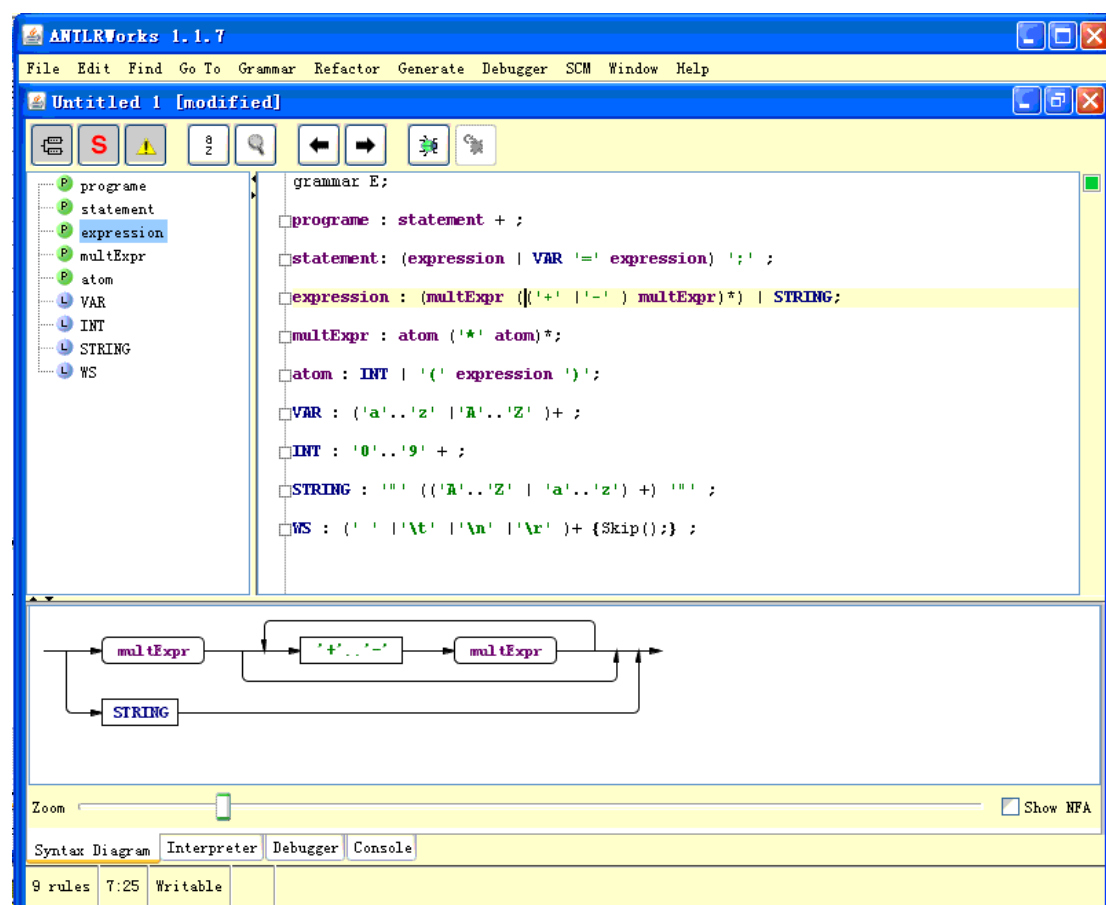
ANTLR v3

Latest ANTLR version is 3.0.1, released August 13, 2007. v3 is written in ANTLR v2 and StringTemplate so you need those jars as well. The source distribution includes all jars (including v3's jar) necessary to run ANTLR from the command line.

- [ANTLR 3.0.1 source distribution](#) ←
- [ANTLR 3.0.1 tool+Java runtime binary only jar](#) (included in source distribution)

Runtime libraries

(图 1.4) ANTLR Runtime 下载页面



(图 1.5) ANTLRWorks IDE

1.3 Java 环境的运行

下面录入文法文件，运行 ANTLRWorks 点击“File - New”菜单新建文法文件，在新文件中将前面的文法录入。（我的网站中有本书所有示例源代码，但我建议您还是手工录入一遍。这样您会有更好的学习效果。）录入文法后点击“File - Save”菜单文件名为“E.g”。然后点击“Generate - Generate Code”，如果 ANTLRWorks 提示“The grammar has been successfully generated in path...”说明 ANTLRWorks 成功生成了语法分析器的代码。会在“E.g”的当前目录中生成“ELexer.java”、“EParser.java”、“E.tokens”和“E_.g”四个文件，其中有两个 java 源文件。“ELexer.java”为词法分析部分的代码，“EParser.java”为语法分析部分的代码。那么为什么会生成 java 代码呢？ANTLR 在不指定目标语言的情况下默认是 java 语言。我们也可以在文法文件中加入 options 项指定目标语言。

```
grammar E;
options { output=AST; language=Java; }
program : statement + ;
```

.....

生成了代码后，我们来编译运行语法分析器。刚才生成的是 java 代码，所以先来看看 java 如何编译运行。首先要有一个执行程序的 main 方法，这个类如下：

```
import org.antlr.runtime.*;
import org.antlr.runtime.tree.*;
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.program_return r = parser.program();
        System.out.println(((BaseTree)r.getTree()).toStringTree());
    }
}
```

把这段代码存入 run.java 文件中，main 方法功能是从命令行接收输入的表达式，通过词法分析和语法分析两个步骤来获得这个表达式的语法树，并以字符串的形式输出语法树的内容。

```
ANTLRInputStream input = new ANTLRInputStream(System.in);
ELexer lexer = new ELexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
```

词法分析步骤是从命令行接收输入的表达式，通过 ANTLR 内部 ANTLRInputStream 类，生成一个 ANTLRInputStream 类的实例 input，再将 input 传给 ELexer 类。ELexer 类是词法分析类，把 input 中的输入内容进行词法分析，词法分析后会产生记号流（token stream）交给语法分析类，为语法分析提供前提。

```
EParser parser = new EParser(tokens);
EParser.program_return r = parser.program();//此处进行了语法分析
System.out.println(((BaseTree)r.getTree()).toStringTree());
```

语法分析步骤是根据词法分析产生的记号流生成语法分析类的实例 parser。然后调用 parser 的方法 program()。这个方法名和我们文法中的第一条规则 program : statement + ; 的名字是一致的，这说明我们要用整个文法进行分析，program 是文法的起点。调用 program() 方法后就进行了语法分析，program() 方法返回语法分析的信息其中包括语法树。r.getTree() 可以返回语法树，getTree() 返回的是 Object 类型所以这里做一个类型转换 (BaseTree)r.getTree() 并调用其 toStringTree() 方法获得语法树的字符串形式输出。

到现在完成了源代码的录入工作，接下来编译所有的代码。编译的命令行字符串为：

```
javac -classpath .;.....\antlr-3.0.1\lib\antlr-3.0.1.jar *.java
```

run.java 中的 `import org.antlr.runtime.*;import org.antlr.runtime.tree.*;` 所引用的类包含在 `antlr-3.0.1.jar` 中，解压我们之前下载的 `antlr-3.0.1.tar.gz` 文件，在其中的 `lib` 目录中可以找到 `antlr-3.0.1.jar`。编译字符串中的 `-classpath` 参数中给出 `.....\antlr-3.0.1\lib\antlr-3.0.1.jar` 的实现路径。运行程序的命令行字符串与编译字符串相似：

```
java -classpath .;.....\antlr-3.0.1\lib\antlr-3.0.1.jar run
```

好的我们来执行这两个字符串来编译并执行程序，执行程序后命令行光标会等待输入，把之前准备分析的两个表达式输入，然后按 `Ctrl+Z`（Windows 系统 `Ctrl+Z`，UNIX 系统 `Ctrl+D`）表示输入结束，然后回车。

```
23+4*(5+1);
str="hello world";
^Z
23 + 4 * ( 5 + 1 ) ; str = "hello world" ;
```

程序输出 `23 + 4 * (5 + 1) ; str = "hello world" ;`；这表示程序执行成功了。我们的语法分析器已经正确的解析了这两个表达式。您可以试着用不规则的格式输入两个表达式会得到相同的输出，因为已经正确分析了表达式的语法，所以输出格式化的字符串对我们的分析器来说是很简单的事情了。

```
23    +4*(    5+    1
);str
= "  hello world  "
;
^Z
23 + 4 * ( 5 + 1 ) ; str = "  hello world  " ;
```

1.4 .NET 环境的运行

我们再说 .NET 的编译执行。首先生成 C# 的语法分析器代码，在文法中的 `options` 设置中修改目标语言为 `CSharp`，还要把 `WS` 中的 `skip()` 改成 `Skip()`。Java 版和 .NET 版的 `ANTLR Runtime` 都使用了各自语言的命名规范，所以名称上略微有些区别。

```
options { output=AST; language=CSharp; }
.....
```



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

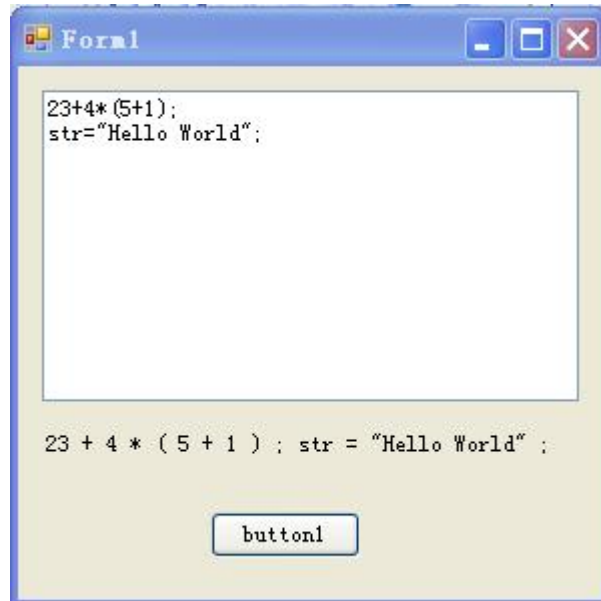
然后用 ANTLRWorks 中的 Generate 菜单生成代码，这时目录中会生成四个文件“ELexer.cs”、“EParser.cs”、“E.tokens”和“E__.g”。E.tokens 和 E__.g 文件与之前 Java 开发中生成的两个同名文件是一样的，另外 ELexer.cs 为词法分析器，EParser.cs 为语法分析器。在 .NET 开发中我们采用 Visual Studio.NET2005 做为开发环境，让我们的示例和真正的开发贴近一些。在 Visual Studio.NET2005 中先新建一个名称为“HelloWorld”的 C# WindowApplication 项目，将生成的 ELexer.cs、EParser.cs 文件拷贝并加入到项目中。再将 .NET 版的 ANTLR Runtime 的 DLL 引用到项目中来。本示例需要 Antlr3.Runtime.dll 和 antlr.runtime.dll，这两个文件在 antlr-3.0.1.tar.gz 解压后的 antlr-3.0.1\antlr-3.0.1\ runtime\CSharp\bin\net-2.0 目录中可以找到。这些操作都完成之后，我们在程序窗体上放一个多行文本框，一个按钮和一个 Label。在窗体的代码文件 Form1.cs 中加入：

```
using Antlr.Runtime;  
using Antlr.Runtime.Tree;
```

我们要实现在文本框中输入表达式语句，点击按钮语法树结果显示在 Label 控件中。在按钮的事件中加入如下代码：

```
ICharStream input = new ANTLRStringStream(textBox1.Text);  
ELexer lex = new ELexer(input);  
CommonTokenStream tokens = new CommonTokenStream(lex);  
EParser parser = new EParser(tokens);  
EParser.program_return progReturn = parser.program();  
label1.Text = ((BaseTree)progReturn.Tree).ToStringTree();
```

这里由于表达式是从界面文本框中获得，所以第一行代码和上面 java 的示例有些不同使用 ANTLRStringStream 类来接收录入的内容。后面的代码和 java 版本中的几乎一致，只是有一些 java 与 .NET 在命名规则方面的区别。Java 方法名首字母为小写而 .NET 是大写。



(图 1.6) .NET 版 HelloWorld 的运行结果

1.5 改进示例

到此我们已经学习了 java 和 .NET 开发语法分析器的全过程。读者可能觉得做完这个示例成就感不大，因为输入和输出是一样的，并没有看到前面提到的语法树结构。下面我们修进一步示例在文法中添加一些构造语法树的符号，使程序构造出如图 1.1、图 1.2 的语法树。文法修改如下：

```

grammar E;
options{ output=AST;}
program : statement + ;
statement: (expression | VAR '=' ^ expression) ';' ;
expression : (multExpr (('+' ^ | '-' ^ ) multExpr)*) | STRING;
multExpr : atom ('*' ^ atom)*;
atom : INT | '(' ! expression ')' !;
VAR : ('a'..'z' | 'A'..'Z' )+ ;
INT : '0'..'9' + ;
STRING : '"' (('A'..'Z' | 'a'..'z' | ' ') +) '"' ;
WS : (' ' | '\t' | '\n' | '\r' )+ {skip();} ;

```

修改后的文法中所有操作符的后面都添加了一个“^”符号，这表示操作符会在构造语法树时作为根节点。“statement: (expression | VAR '=' ^ expression) ';' !;”一行中的“;”字符与“atom : INT | '(' ! expression ')' !;”的“()”字符后面添加了“!”符号，表示不让“()”和“;”出现在语法树中，因为语法树已经体现了操作求值顺

序，所以括号没有必要出现在语法树中。代表语句结束的“;”是为语法分析服务的，语法分析后语法树中的也没有必要加入“;”。我们会在以后的章节中更详细讲解如何构造语法树。现在先用修改后的文法来生成代码，编译运行程序，输入同样的表达式我们会得到如下结果

```
23+4*(5+1);
str="hello world";
^Z
(+ 23 (* 4 (+ 5 1))) (= str "hello world")
```

程序的输出结果了发生了变化：算术表达式的语法树输出字符串形式为：(+ 23 (* 4 (+ 5 1))) 我们已经使“()”不出现在语法树中了，所以输出字符串中的括号并不是我们输入的表达式括号，这些括号表示语法树的结构。由于我们让操作符为根节点，所以这里“+”、“*”操作符出现在前面，其后是它的左子树，再往后是它的右子树，内层括号是外号括号子树。按照这个规则我们可以绘出语法树：

```

      +
     / \
    23  *
       / \
      4  +
         / \
        5  1
```

绘出语法树和前面图 1.1 中的语法树是一样的，这说明我们已经正确的生成了语法树。赋值表达式亦然。我们可以在 Visual Studio.NET 2005 中运行。在程序中设置断点并查看 progReturn.Tree 的对象内存情况。

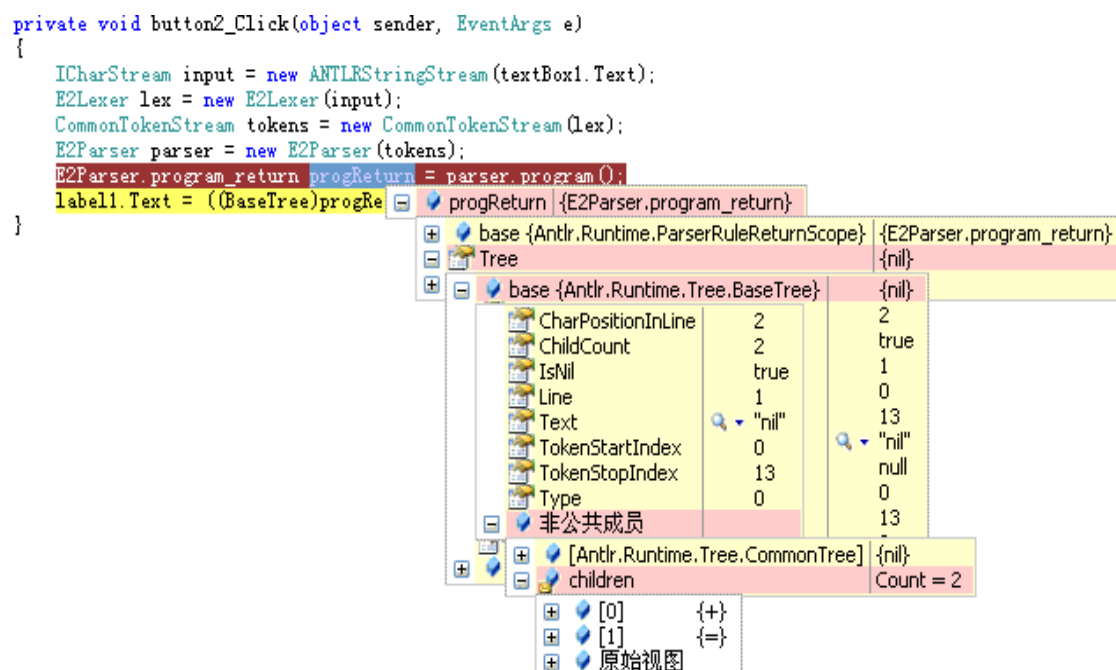


图 1.7

语法树是 `BaseTree` 类型，`BaseTree` 有一个 `children` 集合用来存放子语法树，子语法树也是 `BaseTree` 类型，可以利用 VS.NET 内存监视功能一级一级展开，看一看 ANTLR 的语法树的对象表现形式。`BaseTree` 类有一个 `GetChild(int i)` 方法可以获取第 `N` 个子树，还有一个 `ChildCount` 属性代表子树的个数。结合这两个属性和方法可以用如下的代码遍历子语法树。

```
for (int i = 0; i < tree.ChildCount; i++) {  
    BaseTree currTree = (BaseTree) tree.GetChild(i);  
}
```

1.6 本书结构（以后补充）

好，完成 Hello World 示例的开发后，介绍一下本书的结构。本书后面章节大体顺序是先学习文法、推导式等编译原理基础知识，使没有编译原理知识基础的读者铺平道路。然后全面学习 ANTLR 开发技术，主要包括词法、语法、语法树以及字符模板的内容。在 ANTLR 全学习之后再强化学习一下编译原理的知识（如 DFA）然后学习如何解决 ANTLR 开发中的较难的问题。

第二章 编译原理基础知识

第三章 词法分析

第四章 语法分析

第五章 嵌入文法的 Actions

第六章 构造语法树

第七章 遍历语法树及语义分析

第八章 使用字符串模板

第九章 编译错误处理

第十章：文法编写中的错误和解决方法

第十一章：ANTLR API

第十二章：一个总体的开发实例。

1.7 本章小结

本章开发表达式语法分析器示例详细地向读者介绍了 ANTLR 的开发过程，如何建立

ANTLR 的开发环境以及如何在 Java 和 .NET 环境中编译和运行程序。本书后面出现的示例希望读者都要亲手完成，只有亲手做出正确运行的程序才算是真正领悟了书中的内容。

从文件加载

```
TestE10Lexer lexer = new TestE10Lexer(new  
ANTLRFileStream("TestE10.txt"));  
CommonTokenStream tokens = new CommonTokenStream(lexer);  
TestE10Parser parser = new TestE10Parser(tokens);
```