第三章 词法分析

本章讲述 ANTLR 的词法分析部分。词法分析是编译过程的第一步，是编译过程的基础。词法分析除了上一章讲过它为语法分析提拱记号流，滤掉编译过程不关心的内容以外，还有一个重要的作用是有了词法分析可以大大提高编译的效率。可能有人曾有过疑问，为什么一定要有词法分析？词法分析和语法分析的关系与其它编译过程有些不同，如：语义分析，生成代码在编译过程中是独立的步骤与其它步骤有明显的区别。而词法分析和语法分析在形式上很相似，都要用文法规则去定义语言的结构，为什么不统一起来呢？可以想象一下如果把词法分析和语法分析合并会有什么不同，也就是说我们直接对源代码做语法分析。如 C#源代码中当我们遇到一个字符“c”这时它可能会是关键字“class”标识符“c1”等等。如果是 class 关键字那么接下是要分析一个类代码，如果是标识符那要看它的具体上下文而定。这样的话与一个字符“c”有可能对应的情况太多了，分析起来效率会很低。所以要先做一遍词法分析把源程序的基础组成单位先分析出来。词法分析是语法分析的一个缓冲，可以大大提高编译效率。

3.1 词法分析的规定

在 ANTLR 中词法分析定义的规则名必须以大写字母开头如“LETTER”，“NewLine”。我们在第一章示例中的词法分析部分与语法分析部分合写到一个文件中，ANTLR 允许把词法分析部分和语法分析部分分别写到两个文件中。

T.g 文件存放语法定义：

```
grammar T;
Options {tokenVocab = T2;}
a : B*;
```

T2.g 文件存放词法定义：

```
lexer grammar T2;
B : 'b';
```

将词法分析放到单独的文件中时文法的名称也要和文件的名称相同，在 grammar 关键字之前要加入 lexer 关键字。上例中的 T.g 文件生成语法分析类 TParser，T2.g 文件生成词法分析类 T2Lexer。在 T.g 中要加入一个设置项 tokenVocab 来指定语法文件所需的词法单词是来自 T2.g。这样就可以按照第一章示例中的方法编译运行分析器了。

3.2 字符编码定义

词法分析与源代码直接接触，因为源代码是由字符串组成的，所以我们需要定义字符。ANTLR 有两种方法定义字符，第一种方法是 ANTLR 可以直接使用字符本身很简单直观的定义。

CHAR：‘a’ | ‘b’ | ‘c’;

但这种定义只限于 ASCII 码的字符，下面的定义是不合法的。

CHAR：‘代码’;

定义汉字这样除 ASCII 码以外的字符只能用第二种方法十六进制编码定义法。使用“\u”开头加四位十六进制数定义来定义一个字符。

CHAR：‘\u0040’;

C# 中使用 String.Format("{0:x} {1:x}", Convert.ToInt32(' 代 '), Convert.ToInt32('码'));可以获得汉字的编码。如上面的 CHAR：‘代码’;我们可以定义为：

CHAR：'\u4ee3' '\u7801';

编码有很多种 GB2312 的编码范围是 A1A1 ～ FEFE，去掉未定义的区域之后可以理解为实际编码范围是 A1A1 ～ F7FE。GBK 的整体编码范围是为 8140 ～ FEFE。 BIG5 字符编码范围是 A140 ～ F97E。在 ANTLR 词法规则中定义字符时用 16 进制编码就可以了不用标识出编码的类型。

字符集	编码范围
GB2312	A1A1 ～ 7E7E
GBK	8140 ～ FEFE
BIG5	A140 ～ F97E
Unicode	000000 ～ 10FFFF
UTF-8	000000 ～ 10FFFF

其中汉字在 GB2312 中的编码范围为：‘\uB0A1' .. '\uF7FE'，汉字在 Unicode 编码范围为：’\u4E00’ .. ‘\u9FA5’ | ‘\uF900’ .. ‘\uFA2D’。

附加说明：

1. GBK (GB2312/GB18030)

x00-xff GBK 双字节编码范围

x20-x7f ASCII

xa1-xff 中文

x80-xff 中文

2. UTF-8 (Unicode)

x3130-x318F (韩文)

xAC00-xD7A3 (韩文)

u0800-u4e00 (日文)

ps: 韩文是大于[u9fa5]的字符

3.3 终结符定义方法

LETTER：‘A’ | ‘B’ | ‘C’ | ‘D’ | ‘E’ | ‘F’ | ‘G’ | ‘H’ | ‘I’ | ‘J’ | ‘K’ | ‘L’ | ‘M’ | ‘N’ | ‘O’ | ‘P’ | ‘Q’ | ‘R’ | ‘S’ | ‘T’ | ‘U’ | ‘V’ | ‘W’ | ‘X’ | ‘Y’ | ‘Z’ | ‘a’ | ‘b’ | ‘c’ | ‘d’ | ‘e’ | ‘f’ | ‘g’ | ‘h’ | ‘i’ | ‘j’ | ‘k’ | ‘l’ | ‘m’ | ‘n’ | ‘o’ | ‘p’ | ‘q’ | ‘r’ | ‘s’ | ‘t’ | ‘u’ | ‘v’ | ‘w’ | ‘x’ | ‘y’ | ‘z’;

“..”符号，从上面的 LETTER 示例可以看出，定义一个表示英文字母的符号写起来非常繁琐。为了使定义变得简单 ANTLR 加入“..”符号通过指定首尾的字符可以很方便的定义一定范围内的字符。

LETTER：‘A’ .. ‘Z’ | ‘a’ .. ‘z’;

“~”符号，如果我们想表示除某些符号以外的符号时，可以使用“~”符号。“~”代表取反的意思。

A：~ ‘B’;

符号 A 匹配除字符“B”以外的所有字符。

A：~ (‘A’ | ‘B’);　　B：~(‘A’ .. ‘B’);　　C：~‘\u00FF';

这个的例子中定义三个符号。符号 A 匹配除字符“A”和“B”以外的所有字符，符号 B 匹配除大写字符母以外的所有字符。符号 C 匹配除编码为“u00FF”的字符以外的所有字符。

“.”符号，ANTLR 中可以用“.”表示单个任意字符，起通配符的作用。

A：.;　　B：.*;　　C：.* ‘C’;　　D：~ .;//error

这个例子中符号 A 匹配一个任意字符，符号 B 符号匹配 0 到多个任意字符，符号 C 匹配 0 到多个任意字符直到遇到字符“C”为止。D 的定义是错误的，不能定义任意字符以外的字符。

3.4 skip()方法

有些字符是不属于源程序范畴内的，这些字符在分析过程中应该忽略掉。在 ANTLR 中可以在词法定义中加入 skip();，(如果是 C#为目标语言为 Skip();)。在规则的定义的之后与表示定义结束的分号之前加入“{skip();}”。例如下面定义了一个跳过空白的词法定义。

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

空白符号 WS 中有空格符、制表符、回车和换行符，当遇到这些字符时词法分析程序会调用 skip()方法跳过这些字符，在语法分析时就可以不考虑这些空白字符的存在了。下面再看另一个 skip()的例子。

B： 'A' | 'B' {Skip();} | 'C' ;

这个例子中符号 B 只在匹配字符“B”时跳过，从这个例子可以看出{Skip();}要写在忽略内容的后面，如果它处于某选择分支中那么它只对某分支起作用。下面我们定义一些实际中经常出现的词法定义。

INT **:** DIGIT+;

DIGIT **:** ‘0’ .. ‘9’;

INT 定义了整型数，整型数是由 1 个或多 0 到 9 的数字组成的。下面我们来定义浮点数，浮点数的整数部分至少要有一位数字，小数部分是可有可无的，如要有小数部分则至少要有 1 位小数位。

FLOAT **:** DIGIT+ (**‘.’** DIGIT+)?;

在编程语言中注释是必不可少的，各种编程语言都有它的注释方法。下面看 java 语言中注释的定义。

COMMENT : '/*' . * '*/' {skip();} ;

LINE_COMMENT : '//' ~ ('\n' | '\r') * '\r'? '\n' {skip();} ;

/* */ 中和 //后代表的注释部分应该被忽略掉，下面我们给出完全的示例并运行它。

```
grammar T;
options {
 language=CSharp;
 output=AST;
}

a : INT;

INT : DIGIT+;
DIGIT : ‘0’ .. ‘9’;
COMMENT : '/*' . * '*/' {Skip();} ;
LINE_COMMENT : '//' ~ ('\n' | '\r') * '\r'? '\n' {Skip();} ;
WS : ( ' ' | '\t' | '\n' | '\r' ) + {Skip();} ;
```

文法中的 COMMENT、LINE_COMMENT 和 WS 规则只是为了滤掉相应内容，没有必要与语法规则关联，这样它们也可以正确的工作。COMMENT 符号使用了“.”符号来匹配一切字符直到匹配“/”字符为止。LINE_COMMENT 从“//”字符开始匹配除了“\r\n”以外的所有字符直到匹配“\r\n”为止。其中回车换行符中可以没有回车符所以“\r”是可选项。

将如下源代码保存到与 exe 同目录的 f1.txt 文件中。

```
/* 这是
```

注释 */

234//这是注释

/* 这是

注释 */

下面是运行分析器的代码。

```
TestSkipLexer lex = new TestSkipLexer(new ANTLRFileStream("f1.txt"));
CommonTokenStream tokenStream = new CommonTokenStream(lex);
TestSkipParser parser = new TestSkipParser(tokenStream);
TestSkipParser.a_return aReturn = parser.a();
```

在 Visual Studio.NET 2005 中单步执行上面几行代码，当执行到 TestSkipParser.a_return aReturn = parser.a();时用内存监视查看 aReturn.Tree 可以看到语法树根节点只有 234 一个子节点，这说明所有空白和注释的内容都被忽略了，没出现在语法树中。

3.5 $channel = HIDDEN

有时象注释这样的部分在编译时并不是完全放弃的，如 java 中在/**….*/的注释形式中的内容可以添加到最终生成程序的文档中，在.NET 中使用 /// 注释形式也有类似的功能。也就是说我们在编译代码时忽略这些注释，而在分析生成文档时又要用到这些注释。这样的话就不能使用 skip();方法了，skip()方法是抛弃其内容。所以 ANTLR 中加入了 channel 属性用来指定当前匹配的内容是属于哪一个频道，分析过程中关心哪个频道就可以只接收哪个频道的内容。ANTLR 中定义有两个频道常量 DEFAULT_CHANNEL 和 HIDDEN_CHANNEL，一般情况下匹配的内容都中放到 DEFAULT_CHANNEL 中。我们可以指定让当前匹配的内容放到哪个频道中。

前面示例中的词法规则可以改成如下形式。

```
COMMENT : '/*' .* '*/' {$channel=HIDDEN;} ;
LINE_COMMENT : '//' ~ ('\n' | '\r') * '\r'? '\n' {$channel=HIDDEN;} ;
WS : ( ' ' | '\t' | '\n' | '\r' ) + {$channel=HIDDEN;};
```

COMMENT、LINE_COMMENT 和 WS 词法规则后面加入{$channel=HIDDEN;}，这样匹配的字符就会被存放到 HIDDEN 频道中。下面将这个修改后的文法编译运行，其结果和使用 skip()方法时是一样的，注释部分没有出现在语法树中。这就是说存在于 HIDDEN 频道中的内容被语法分析器忽略了，因为在默认情况下 ANTLR 语法分析程序只分析 DEFAULT_CHANNEL 中的内容。

Channel 这个概念是介于词法分析和语法分析之间的一个概念，将存放匹配的内容存

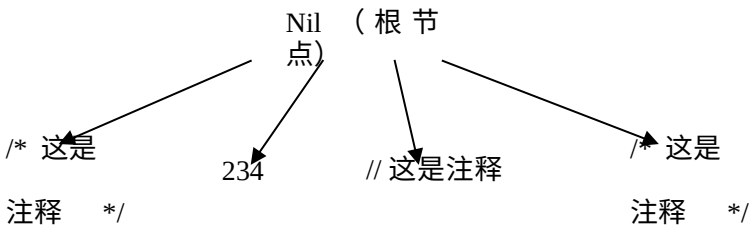
放到各个频道中是在词法分析时进行的，而语法分析时可以有选择的读某一个频道中的内容。那么如何让语法分析程序处理示例中的注释内容呢？我们再次修改文法因为我们只关心注释所以把 WS：(' ' | '\t' | '\n' | '\r') + {Skip();};改回成 skip()方法因为空白我们是不关心的，下面给出全部的文法。

```
grammar TestHidden;
options {
 language=CSharp;
 output=AST;
}
a : b INT b;

b : (COMMENT | LINE_COMMENT)*;

INT : DIGIT+;
DIGIT : '0' .. '9';
COMMENT : '/*' . * '*/' {$channel=HIDDEN;} ;
LINE_COMMENT : '//' ~ ('\n' | '\r') * '\r'? '\n' {$channel=HIDDEN;} ;
WS : ( ' ' | '\t' | '\n' | '\r' ) + {Skip();} ;
```

下面是运行代码，运行代码中加入了 tokenStream.SetTokenTypeChannel 方法。

```
TestSkipLexer lex = new TestSkipLexer(new ANTLRFileStream("f1.txt"));
CommonTokenStream tokenStream = new CommonTokenStream(lex);
TestSkipParser parser = new TestSkipParser(tokenStream);
tokenStream.SetTokenTypeChannel(TestSkipLexer.COMMENT,
                                Token.DEFAULT_CHANNEL);
tokenStream.SetTokenTypeChannel(TestSkipLexer.LINE_COMMENT,
                                Token.DEFAULT_CHANNEL);
TestSkipParser.a_return bReturn = parser.a();
```

SetTokenTypeChannel 方法是将指定的频道中的符号加入到分析程序关心的范围内，第一个参数是词法符号，第二个参数是这个符号所处频道。两条 SetTokenTypeChannel 语句将 DEFAULT_CHANNEL 频道中的 COMMENT、LINE_COMMENT 加入到语法分析程序 channelOverrideMap 集合中，语法分析程序会分析 channelOverrideMap 集中注



册的类型，此示例运行后的语法树有四个子节点。这说明分析程序分析了注释的内容并将其加入到了语法树中。

3.6 options {greedy=false;}

ANTLR 词法分析中还可以加入 greedy 设置项，当 greedy=true 时当前规则会尽可能的匹配可以匹配的输入。ANTLR 中默认情况 greedy 为 true，所以 COMMENT： '/*' . * '*/' {$channel=HIDDEN;} ;规则中符号“.”可以匹配“*/”字符，也就是说当遇到“*/”字符时它是匹配“.”还是匹配’*/’出现了二义的情况，前面的例子中已经显示出在这种情况下分析器是可以正确分析的，但加入 greedy=false 后可以消除这种二义性，就是说 greedy=false 时分析器遇到“*/”字符时会作为注释结束符号来匹配，这样的话就消除了二义性。

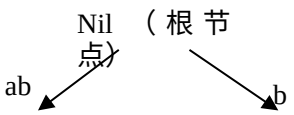
COMMENT : '/*' (options {greedy=false;} : .) * '*/' {$channel=HIDDEN;} ;

options {greedy=false;}：是一种局部设置的写法，其格式为：(options {«option-assignmen ts»}：«subrule-alternatives»)。关键字 options 后面用“{}”将设置项括起来，使用“:”符号表示对后面部分进行设置，这种设置的方法必须在子规则中设置它只能在子规则中有效。如：'/*' options {greedy=false;}：. * '*/' 这样的定义是不无法的。

下面给出一个更加明显的例子。

```
grammar TestGreedy;
options {
  language=CSharp;
  output=AST;
}
c : A B;
A : 'a' 'b' ?;
B : 'b';
```

此例子中 c 规则有 A 和 B 两个词法符号，其中 A 匹配的是“a”或“ab”字符，B 匹配的是“b”字符。这时会出现一个问题就是当输入为“ab”时这个“b”是规则 A 的 b 还是规则 B 的 b，这属于二义性问题。此示例在实际运行时如果输入为“abb”时可以正确生成有两个子节点的语法树。



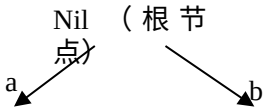
如果输入为“ab”会出现 MismatchedTokenException 异常，因为现在分析器的

greedy 属性为 true。这时分析器会尽可能的匹配输入的字符“ab”被规则 A 匹配，这时规则 B 就没有了可匹配的字符，所以出现了 MismatchedTokenException 异常。尽可能匹配的意思是所有的可选项分析器都认为是需要的。

如果我们将规则 A 中可选的’b’定义为 greedy=false;代码如下。

```
c ： A B;
A ： 'a' (options {greedy=false;} : 'b')?;
B ： 'b';
```

当输入“ab”和“abb”时生成的语法树同样为：



规则 A 中的’b’由于是可选项并且其后规则 B 中也需要匹配’b’字符所以在 Greedy 属性为 false 时规则 A 不去匹配字符’b’。这样输入的字符串中第一个’b’字符与规则 B 匹配，第二个’b’字符无处匹配。

3.7“.”相关注意

词法定义中有一些与通配符“.”有关的注意事项，下面给出一个示例。

```
d :C ANY PLUS;
C : 'c';
PLUS : '+';
ANY : ( options {greedy=false;} : . ) *;
```

这个文法是要匹配输入字符“c”和“+”，字符“c”与“+”之间可以是一些不确定的内容所以用通配符“.”进行匹配，如前面学到的应该使用 greedy=false 设置。但是此文法运行时会出现死循环情况。我们可以使用其它方法改写定义，由于字符“c”与“+”之间不应该出现“c”与“+”字符以防出现二义性，所以 ANY 规则可以修改成以下的定义方法。

```
ANY : ~(C | PLUS)*;
```

不过如何正确使用“.”通配符呢？在使用通配符定义规则时要在当前的规则中明确的指定开始和结束字符。前面的示例中定义/**/注释时就是指定了开始结束字符。本示例只能改成：

```
d : CANYPLUS;
CANYPLUS : ‘c’ ( options {greedy=false;} : . ) * ‘+’;
```

所以“.”的使用范围是有限的，请大家注意。

3.8 fragment 词法规则

ANTLR 文法中语法规则是在词法规则基础上建立的。但不一定每个词法规则都会被语法规则直接引用到。这就象一个类的公有成员和私有成员，公有成员是对外公开的会被外界直接调用。而私有成员不对外公开只能通达公有成员间接调用。在词法规则中那些不会被语法规则直接调用的词法规则可以用一个 fragment 关键字来标识，fragment 标识的规则只被其它词法规则引用。

```
grammar TestFragment;
options {
  language=CSharp;
  output=AST;
}
a : INT;
INT : DIGIT+;
fragment DIGIT : '0' .. '9';
```

此示例中词法符号 INT 定义了整型数，而 DIGIT 定义了一位数字。语法规则 a 对 DIGIT 规则的使用是间接的，这时使用 fragment 来标识 DIGIT 规则，DIGIT 规则将不能被语法规则直接调用。如果如下例语法规则 a 直接调用 DIGIT 规则，运行时语法分析器会死循环。

```
a : DIGIT;
INT : DIGIT+;
fragment DIGIT : '0' .. '9';
```

在不使用 fragment 关键字的时候，有时语法规则也不能直接使用某些词法规则。请看下面的示例。

```
a : DIGIT;
INT : DIGIT+;
DIGIT : '0' .. '9';
```

运行后输入“9”分析器生成的语法树为空，使用 java 命令行运行时提示“line 1:0 mismatched input '0' expecting DIGIT ”。这是因为 DIGIT 规则匹配的内容对 INT 规则来说也是匹配的，INT 规则干扰了 DIGIT 规则的匹配，造成 DIGIT 规则没有匹配的内容。

```
a : DIGIT;
fragment INT : DIGIT+;
DIGIT : '0' .. '9';
```

如果在 INT 前加入 fragment 则分析器可以生成有一个节点“9” 的语法树，不过在实际中不应该这样定义。在第五章我们会学到 fragment 词法规则是可以带参数的，这样可以在词法分析的过程中灵活的控制，fragment 词法规则具有更大的意义。

3.9 filter=true

我们在分析源代码时有时只想获得其中一部分信息，例如：我们想知道一个 java 文件中的类的类名，类有哪些方法，方法的参数和返回值及其类型，属性及其类型和此类继承了什么类。但是我们必须写出 java 的全部文法才可以分析 java 文件，这很有难度也没有必要。ANTLR 中加入了一个叫 filter 的设置项，filter 为布尔类型默认为 false。filter 为 true 时词法分析器可以处于一种过滤状态，我们只需定义出我们关心部分的词法结构，其它部分全部被忽略掉。

filter=true 模式中的词法规则的先后顺序很重要，写在最前面的词法规则优先级最高之后从高到低依次排列。这样规定是因为有些规则会被其它规则所包含造成一些规则无法被识别出来。请看下面的示例：

```
lexer grammar TestFilter;
options {
filter=true;
language=CSharp;
}
A : aText=AA{Console.Out.WriteLine("A: "+$aText.Text);};
B : bText=BB{Console.Out.WriteLine("B: "+$bText.Text);};
AA : 'ab';
BB : 'a';
```

执行代码为：

```
ICharStream input = new ANTLRFileStream("t.txt");
FuzzyJava2Lexer lex = new FuzzyJava2Lexer(input);
ITokenStream tokens = new CommonTokenStream(lex);
tokens.ToString();
```

此文法中规则 AA 匹配字符串“ab”，规则 BB 匹配字符串“a”，规则 AA 包含规则 BB。t.txt 文件中的内容为“xxxxxxabxxxxx”。我们看一看是内容规则 A 匹配优先还是规则 B 优先匹配，这可以反应出优先关系。这个例中输出为：“A : ab”。这说明规则 A 优先匹配了。（aText 和 bText 叫例变量它们可以保存把 AA 和 BB 符号的内容，{ Console.Out…… }是嵌入在文法中的代码用于输出 AA 和 BB 中的内容。这会在第五章中讲述。）下面我们修改一下文法改变一个 A 和 B 规则的位置：

```
B : bText=BB{Console.Out.WriteLine("B: "+$bText.Text);};
```

```
A : aText=AA{Console.Out.WriteLine("A: "+$aText.Text);};
BB : 'a';
AA : 'ab';
```

相同的输入这次输出为： B：a。这说明规则 B 优先匹配了。但是如果将规则 BB 改为匹配字符“b”则输出为“A：ab”。

```
B : bText=BB{Console.Out.WriteLine("B: "+$bText.Text);};
A : aText=AA{Console.Out.WriteLine("A: "+$aText.Text);}; //输出为  A：ab
BB : 'b';
AA : 'ab';
```

下面给出一个从 java 文件中读取类，方法，属性信息的示例。

```
lexer grammar FuzzyJava2;
options {
    filter=true;
}
IMPORT : 'import' WS name=QIDStar WS? ';'
{Console.Out.WriteLine("Import: "+$name.Text);};

PACKAGE : 'package' WS name=QID WS?';'
            {Console.Out.WriteLine("Package: "+$name.Text);};

CLASS : 'class' WS name=ID WS? ('extends' WS QID WS?)?
        ('implements' WS QID WS? (',' WS? QID WS?)*)? '{'
     {Console.Out.WriteLine("Class: "+$name.Text);};

METHOD : type1=TYPE WS name=ID WS?
    '(' ( ARG WS? (',' WS? ARG WS?)* )? ')' WS?
    ('throws' WS QID WS? (',' WS? QID WS?)*)? '{'
       {Console.Out.WriteLine("Method:  "+type1.Text+"  "+$name.Text +
                              "()");};

FIELD : defvisi=DEFVISI WS TYPE WS name=ID '[]'? WS? (';'|'=')
                        {Console.Out.WriteLine("Field:"+$defvisi.Text+"    "+
```

```
                                    $name.Text+";");};

DEFVISI : 'public' | 'protected' | 'private';

USECOMMENT : '/**' (options {greedy=false;} : . )* '*/' ;

COMMENT : '/*' (options {greedy=false;} : . )* '*/'
        //{Console.Out.WriteLine("found comment "+Text);};
SL_COMMENT : '//' (options {greedy=false;} : . )* '\r'?'\n' ;

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

STRING : '"' (options {greedy=false;}: ESC | .)* '"';

CHAR : '\'' (options {greedy=false;}: ESC | .)* '\'';

fragment QID : ID ('.' ID)*;
fragment QIDStar : ID ('.' ID)* '.*'? ;
fragment TYPE : QID '[]'?;
fragment ARG : type=TYPE WS name=ID
            {Console.Out.WriteLine("Param:  "+$type.Text  +  "  "+
                $name.Text + " , ");};
fragment ID : ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
fragment ESC : '\\' ('"'|'\''|'\\');
```

注意如果我们定义一个词法规则是由其它词法规则组成的，如： NODE：STRING | QID | NUM;这样写 ANTLR 会提示下面的错误信息“STRING,NUM 不可能被匹配，因为它们之前的规则已经匹配了同样的内容”而无法生成分析器代码。这种情况我们应该把 Node 改为语法规则写成： node：STRING | QID | NUM;或将 STRING、QID 和 NUM 改成 fragment 词法规则就可以了。

我们编译运行这个分析器来分析下面的 java 源代码。

```
/* [The "BSD licence"] Copyright (c) 2005-2006 Terence Parr
 All rights reserved.*/
package org.antlr.analysis;
import org.antlr.misc.IntSet;
import org.antlr.misc.OrderedHashSet;
import org.antlr.misc.Utils;
import org.antlr.tool.Grammar;
import java.util.*;
public class DFAState extends State {
    public DFA dfa;
      public DFAState(DFA dfa) {
       this.dfa = dfa;
    }
    public int addTransition(DFAState target, Label label) {
        transitions.add( new Transition(label, target) );
            return transitions.size()-1;
    }

    /** Print all NFA states plus what alts they predict */
      public int getLookaheadDepth() {
            return k;
      }
      public void setLookaheadDepth(int k) {
            this.k = k;
            if ( k > dfa.max_k ) { // track max k for entire DFA
                  dfa.max_k = k;
            }
      }
}
```

将上面的 java 程序分析后分析器会输出下面的结果。

```
Package: org.antlr.analysis
```

Import: org.antlr.misc.IntSet

Import: org.antlr.misc.OrderedHashSet

Import: org.antlr.misc.Utils

Import: org.antlr.tool.Grammar

Import: java.util.*

Class: DFAState

Field:public dfa;

Param: DFA dfa ,

Method: public DFAState()

Param: DFAState target ,

Param: Label label ,

Method: int addTransition()

Method: int getLookaheadDepth()

Param: int k ,

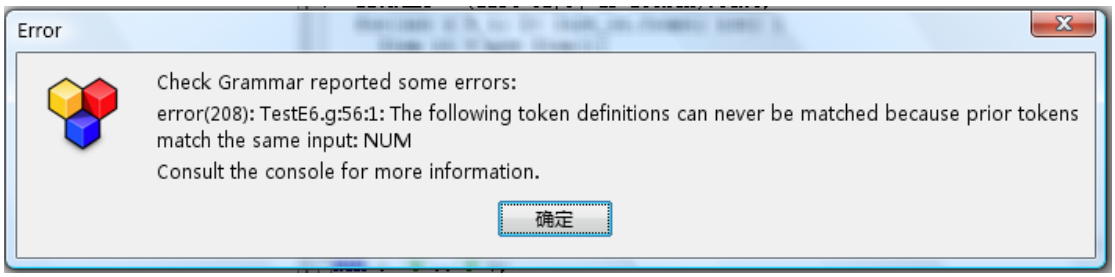
Method: void setLookaheadDepth()

3.10 词法规则的先后顺序

前面已经提到了关于词法规则先后顺序的问题，这里我们再举一个具体的例子来进一步说明。请看下面两个词法规则：

```
CHINESECHAR : ('A'..'Z' | 'a'..'z'| '0'..'9' | '(' | ')'
            | '\u4E00' .. '\u9FA5' | '\uF900' .. '\uFA2D') +;
NUM : '0'..'9'+;
```

CHINESECHAR 规则可以接受中文字符（不包括全角标点符号），而 NUM 规则可以接受正整数。包含这两个规则的文法生成分析器时 ANTLR 会提示错误信息：



由于在词法分析的过程中如果遇到两个规则都可以匹配当前字符串时，ANTLR 会按文法中词法规则书写的先后顺序来决定优先使用哪个规则来匹配。本例中 CHINESECHAR 包含'0'..'9'并且 CHINESECHAR 写在 NUM 之前，这样遇到数字也会用 CHINESECHAR 规

则来匹配 NUM 规则就不可能匹配输入了。解决办法是把 NUM 规则放在 CHINESECHAR 规则之前。

3.11 常用词法规则

很多情况下我们需要定义的词法规则都是相似的，下面给出一组常用的词法规则定义：

```
grammar Abstract;
NAME :
  (LETTER | UNDERLINE | CHINESECHAR)
  (LETTER | UNDERLINE | DIGIT | CHINESECHAR)* ;
LETTER : ('A'..'Z' | 'a'..'z');
CHINESECHAR : '\u4E00' .. '\u9FA5' | '\uF900' .. '\uFA2D';
INT : DIGIT+;
DIGIT : '0' .. '9';
COLON : ':' ;
COMMA : ',' ;
SEMICOLON : ';' ;
LPAREN : '(' ;
RPAREN : ')' ;
LSQUARE : '[' ;
RSQUARE : ']' ;
LCURLY : '{';
RCURLY : '}';
DOT : '.' ;
UNDERLINE : '_';
ASSIGNEQUAL : '=' ;
NOTEQUAL1 : '<>' ;
NOTEQUAL2 : '!=' ;
LESSTHANOREQUALTO1 : '<=' ;
LESSTHAN : '<' ;
GREATERTHANOREQUALTO1 : '>=' ;
GREATERTHAN : '>' ;
```

```
DIVIDE : '/' ;
PLUS : '+' ;
MINUS : '-' ;
STAR : '*' ;
MOD : '%' ;
AMPERSAND : '&' ;
TILDE : '~' ;
BITWISEOR : '|' ;
BITWISEXOR : '^' ;
POUND : '#';
DOLLAR : '$';
COMMENT :  '/*'    . *  '*/' {$channel=HIDDEN;} ;
LINE_COMMENT : '//'  ~ ('\n' | '\r') *  '\r'? '\n' {$channel=HIDDEN;} ;
WS : ( ' ' | '\t' | '\n' | '\r' ) +  {Skip();} ;
```

3.12 大小写敏感

ANTLR3.01 中没有大小写是否敏感的设置项，所以只能象下面的词法规则这样定义大小写不敏感的单词。

```
SELECT : ('S'|'s')('E'|'e')('L'|'l')('E'|'e')('C'|'c')('T'|'t') ;
FROM : ('F'|'f')('R'|'r')('O'|'o')('M'|'m');
```

还有一种方法是重载输入的 Stream 类的 LA 方法在其中将输入的内容写为大小写不敏感。

```
package org.antlr.runtime;
import java.io.*;
/** @author Jim Idle */
public class ANTLRNoCaseFileStream  extends ANTLRFileStream {
    public ANTLRNoCaseFileStream(String fileName) throws IOException {
        super(fileName, null);
    }
    public ANTLRNoCaseFileStream(String fileName, String encoding)
    throws IOException {
            super(fileName, encoding);
```

```
  }
  public int LA(int i) {
    if ( i==0 ) {
      return 0; // undefined
    }
    if ( i<0 ) {
      i++; // e.g., translate LA(-1) to use offset 0
    }
    if ( (p+i-1) >= n ) {
      return CharStream.EOF;
    }
    return Character.toUpperCase(data[p+i-1]);
  }
}
```

C#代码如下：（ 注 可是在 lookahead 时临时转换成小写，原来输入字符不会受到影响。）

```
public class CaseInsensitiveStringStream : ANTLRStringStream {
        public CaseInsensitiveStringStream(char\[\] data, int
            numberOfActualCharsInArray) : base(data,
            numberOfActualCharsInArray) {}

  public CaseInsensitiveStringStream() {}
  public CaseInsensitiveStringStream(string input) : base(input) {}
  public override int LA(int i) {
    if (i == 0) {
      return 0;
    }
    if (i < 0) {
      i++;
    }
    if (((p + i) - 1) >= n) {
      return (int) CharStreamConstants.EOF;
```

```
            }
            return Char.ToLowerInvariant(data\[(p + i) - 1\]);
        }
    }
```

3.11 本章小结

本章讲述了 ANTLR 如何定义词法规则，包括：定义各种编码的字符，通配符“.”和“..”、“~”、“.”符号的用法，skip()方法的用法，$channel = HIDDEN 输入频道，greedy=false 的用法和注意事项以及 fragment 词法规则的用法。这些内容为 ANTLR 中词法分析中基础，后面章节还会讲述词法规则中潜入代码和词法中的二义性的内容。学完本章后读者应该可以定义一种语言的基本词法规则，下一章我们讲述如何在词法规则的基础上定义语法规则。