第九章 错误处理

我们在开发分析器和使用分析器时会遇到各种错误，这些错误可能是开发过程中编写文法的错误，也可能是在使用分析器时遇到了不符合文法的语句而出现的错误。用户在使用分析器时对于语法错误语句子应该给出正确的错误信息。就象我们用 java 或 C#编程时编译器会提示出易于理解的出错信息让我们可以很快解决语法上的错误（现在 Java 和 C# 的提示的错误信息要比以前的 Turbo C 提示的信息要好很多）。我们在开发分析器时也要尽可能给用户提示易于理解的信息。本章我们讲述关于分析器使用过程中的错误住处提示以及关于分析器如何进行错误恢复的内容。

9.1 ANTLR 分析时异常

分析器分析时遇到用户输入的错误句子时提示错误信息。其中有些错误是用户的输入不符合特定语法或语义造成的，对分析器来说并没有发生异常，这种情况要我们自己完成异常的创建和引发。还有绝大多数错误是在分析器引发异常时我们获得并整理提示给用户的。这种情况下我们首先捕获 ANTLR Runtime 抛出的异常，根据异常的类型和其中提拱的行列信息来生成一个正确的错误信息给用户。下面我们看一个示例，其中错误信息中包括错误发生地点的行列信息和一段错误的描述信息。

```
grammar TestE
prog: stat+ ;
stat: expr ';'
        {System.out.println("found expr: "+$stat.text);}
        | ID '=' expr ';'
        {System.out.println("found assign: "+$stat.text);}
       ;
expr : multExpr (('+' |'-' ) multExpr)*;
multExpr : atom ('*' atom)*;
atom: INT
         | '(' expr ')';
ID : ('a'..'z' |'A'..'Z' )+ ;
INT : '0'..'9' + ;
WS : (' ' |'\t' |'\n' |'\r' )+ {skip();} ;
```

本示例中定义了简单的表达式文法，我们运行此分析器输入一个正确句子，看一下 ANTLR 的提示。

```
java TestE
(3);
found expr: (3);
```

输入(3);输出为 found expr: (3);ANTLR 提示成功地找到了 expr 表达式(3)，其中 expr 是规则名提示的(3)在文法为一个 expr。本例用 java 开发分析器时默认可以出现此提示信息。如果用.net 开发此示例在默认情况下可能不会出现这些提示，.net 版本的 ANTLR Runtime 没有设置成自动提示信息。所以我们在用 C#开发时要加入下面的代码。

```
@members {
public override string GetErrorMessage(RecognitionException e,
                    string[] tokenNames)
{
 msg = base.GetErrorMessage(e, tokenNames);
   System.Console.WriteLine(msg);
   return stack+" "+msg;
}
public override string GetTokenErrorDisplay(IToken t) {
   return t.ToString();
}
}
```

利用 @members 定义分析器类的重载方法 GetErrorMessage 和 GetTokenErrorDisplay。在 GetErrorMessage 方法中调用基类的 GetErrorMessage 获得提示信息，然后再输出信息。这样就可以看到提示信息了。

9.1.1 MismatchedTokenException

下面输入一个错误的句子看一下 ANTLR 提示的错误信息：

```
(3;
a=2;
^Z
BR.recoverFromMismatchedToken
line 1:2 [prog, stat, expr, multExpr, atom]
 mismatched input [@2,2:2=';',<7>,1:2
] expecting ')'
found expr: (3;
```

输入是一个错误句子（3;分析器输出 mismatched input expecting ')'提示错误的输入缺少“）”。对于我们开发人员来说知道分析器抛出的异常类型更有用，当前的这个错误的异常类型为 MismatchedTokenException。

MismatchedTokenException 异常有一个 Expecting 属性用来指示当前期待的符号的索引值，我们可以在.tokens 文件中找到相应的字符。还有 UnexpectedType 属性用来指示当前遇到的不期待的符号的索引值

9.1.2 MismatchedCharException

MismatchedTreeNodeException 异常同 MismatchedTokenException 不同的是它是在词法分析时找到不匹配的字符时引发的异常。

9.1.2 MismatchedTreeNodeException

MismatchedTreeNodeException 异常同 MismatchedTokenException 只是它是在 Tree Parser 遍历语法树时找到不匹配的树节点时引发的异常。

9.1.3 NoViableAltException

下面我们再学习一下其它异常。运行上面的分析器，输入另一个错误句子：“3 +”,下面是运行后输出的内容：

```
3+;
^Z
line 1:2 [prog, stat, expr, multExpr, atom]
no viable alt; token=[@2,2:2=';',<7
>,1:2] (decision=5 state 0) decision=<<49:1: atom : ( INT | '(' expr
')' );>>
found expr: 3+;
```

分析器提示“no viable alt”表示输入不是正确的可选项中的一个。根据文法定义表达式中操作符“+”后面可以输入另一个表示式，包括 INT、（但是分析器遇到的是“;”,所以分析器抛出了 NoViableAltException 异常。NoViableAltException 异常是在分析器时根据当前符号决定使用哪个规则进行匹配但所有规则又都不匹配时引发的异常。生成分析器的源代码中经常用类似下面的代码来进行决定匹配的规则。

```
switch ( input.LA(1) )
{
case INT: { alt7 = 1; }
   break;
case ID: { alt7 = 2; }
   break;
case 13: { alt7 = 3; }
   break;
default:
      NoViableAltException nvae_d7s0 =
         new NoViableAltException("", 7, 0, input);
      throw nvae_d7s0;
}
```

NoViableAltException 有两个属性，decisionNumber 属性用来记录当前错误的决定数。stateNumber 属性用来记录当前的状态值。grammarDecisionDescription 属性

用来记录当前决定的描述信息。关于决定数和状态值我们后面详细讲解。

9.1.4 EarlyExitException

下面我们学习另一个错误类型，请看下面的文法：

```
grammar TestE2;
typeCmd : 'type' fileName+;
fileName : ID;
ID : ('a'..'z' |'A'..'Z' )+ ;
WS : (' ' |'\t' |'\n' |'\r' )+ {skip();} ;
```

这个文法定义了 DOS 命令 type 的格式，type 命令用来打印输出文件的内容，单词“type”后面为一个或多个文件名，注意至少要有一个文件名。所以我们输入一个没有文件名的错误的句子来看一下分析器生成什么错误信息。

输入：type
输出：
line 0:-1 required (...)+ loop did not match anything at input '<EOF>'

分析器提示需要(...)+ loop 但没有匹配。这个异常的类型是 EarlyExitException 类。下面是生成的分析器代码中的片段。

```
switch (alt1) {
    case 1 :{
      pushFollow(FOLLOW_fileName_in_typeCmd12);
      fileName();
      state._fsp--;
     }
   break;
     default : {
      if ( cnt1 >= 1 ) break loop1;
          EarlyExitException eee = new EarlyExitException(1, input);
          throw eee;
    }
}
```

因为 fileName+ 至少要匹配一次所以在没有文件名的情况下分析器抛出了 EarlyExitException 异常。

EarlyExitException 异常有一个 decisionNumber 属性用来记录当前错误的决定数。

9.1.5 MismatchedRangeException

MismatchedRangeException 是当输入的符号不在定义的一范围内时引发的异常。下面我们编写一个简单的文法来看一下生成的代码

```
grammar TestE3;
number : Digital+;
Digital : '0'..'8';
```

编译运行分析器，如果输入：9，分析器会引发 MismatchedRangeException 异常。我们打开词法分析器 TestE3Lexer.java 文件看一下其中的 mDigital()函数的代码，mDigital 为了匹配'0'..'8'的范围字符使用了 Lexer 基类中的方法 matchRange 来实现 。matchRange 方法对于超出范围的 9 引发了 MismatchedRangeException 异常。

```
public final void mDigital() throws RecognitionException {
    try {
        int _type = Digital;
        int _channel = DEFAULT_TOKEN_CHANNEL;
        {
                                                    matchRange('0','8');       //Antlr.Runtime.MismatchedRangeException
        }
        state.type = _type;
        state.channel = _channel;
    }
    finally {
      }
}
```

9.1.6 MismatchedSetException

MismatchedSetException 异常与 MismatchedRangeException 类型用来在输入不匹配一系列值时引发。修改上面的文法来实现看一下 Digital 规则生成的代码。

```
number : Digital+;
Digital : 'A' | 'B' | 'C';

 public void mDigital() // throws RecognitionException [2]
 {
     try
     {
       int _type = Digital;
       int _channel = DEFAULT_TOKEN_CHANNEL;
       {
           if ( (input.LA(1) >= 'A' && input.LA(1) <= 'C') )
       {
```

```
            input.Consume();
          } else {
            MismatchedSetException mse
                                = new MismatchedSetException(null,input);
            Recover(mse);
            throw mse;}
          }
          state.type = _type;
          state.channel = _channel;
        }
        finally { }
    }
```

运行分析器输入 CD，分析器会抛出 MismatchedRangeException。这里要注意的是如果输入是 D 没有 C 则分析器会抛出 EarlyExitException 原因我们在讲 EarlyExitException 时已经讲过了。另外 9.1.4 和 9.1.5 这两节的文法中如果加上 WS：(' ' |'\t' |'\n' |'\r')+ {skip();}; 规则，同样的输入分析器则不会抛出 MismatchedRangeException 和 MismatchedSetException 异常，而会确发 NoViableAltException 事件，请大家自己思考其原因。

9.1.7 MismatchedNotSetException

MismatchedNotSetException 异常与 MismatchedSetException 相似，分析器在不能够匹配如~（'A' | 'B' | 'C'）的反转符号集时引发的异常。

9.1.8 FailedPredicateException

FailedPredicateException 异常是在做句法判定时出现错误抛出的异常。句法判定是 antlr 应用的高级语题我们会在后面的章节详细讲述。

9.1.9 UnwantedTokenException

UnwantedTokenException 异常是在分析器遇到了不希望遇到符号时抛出，Unwanted TokenException 是从 MismatchedTokenException 继承的所以与 MismatchedToken Exception 的习性相同。

9.1.10 异常基类 RecognitionException

RecognitionException 类是前面讲述的 ANTLR 中关于分析器过程异常的基类。它提拱了这些异常一些基本的属性和方法。RecognitionException 类是从 ANTLR 所有异常类的基类 ANTLRException 类继承的。RecognitionException 类型的定义如下：

```
public class RecognitionException : ANTLRException {
  // Fields
    public int column;
  public string fileName;
  public int line;
  // Methods
  public RecognitionException();
  public RecognitionException(string s);
  public RecognitionException(string s, string fileName_,
                              int line_, int column_);
  public virtual int getColumn();
   public virtual string getErrorMessage();
  public virtual string getFilename();
  public virtual int getLine();
  public override string ToString();
}
```

其中 getLine()方法用来获得出错位置的行号。getColumn 方法用来获得出错位置的列号。getErrorMessage()方法用来获得异常的错误信息。getFilename()方法用来获得出错的文件名。

下面是 ANTLR 中 RecognitionException 异常及子类的继承关系图。

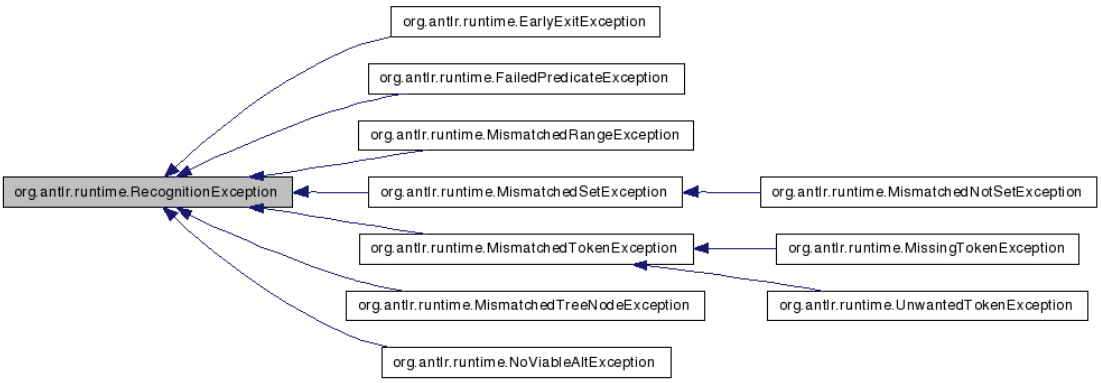


图 9.1

9.1.12 区分 Mismatched 与 NoViableAlt

9.1 节中开发的分析器输入不同的句子会出现不同的异常，为了让大家更加清楚 ANTLR 分析中出现的异常，下面我们再进一步地研究异常的产生。由于 MismatchedToken Exception 与 NoViableAltException 这两个异常导致的原因一般情况难以区分，所以我们来分析一下这两个异常。

```
grammar TestE
prog: stat+ ;
stat: expr ';'
       | ID '=' expr ';'
      ;
expr : multExpr (('+' |'-' ) multExpr)*;
multExpr : atom ('*' atom)*;
atom: INT
        | '(' expr ')';
……
```

TestE 文法的分析器对于(3 的输入抛出 MismatchedTokenException 异常，对于(3+的输入抛出 NoViableAltException。MismatchedTokenException 异常的原因是当前需要匹配的规则没能匹配。(3 这个输入后没有应该匹配的)，所以出现这个异常。但是 (3+ 输入后面也是缺少应该匹配的输入为什么抛出的不是 MismatchedTokenException 呢？NoViableAlt Exception 异常是在匹配下一个符号时需要决定使用哪一个规则来匹配后面的内容时抛出的异常。从文法中可以看出当输入是(3 时分析器会使用 atom 规则的'(' expr ')'分支来匹配，'('匹配后因为 3 是一个合法的 expr 符号串，所以（3 匹配'(' expr。接下来应该没有悬念必须匹配')'。但输入中没有）所以分析器抛出的是 MismatchedTokenException 异常。引发 MismatchedTokenException 异常在于没有其它选择必须匹配的内容的缺失。分析器 atom()函数中的代码如下：

```
Match(input,12,FOLLOW_12_in_atom133);//匹配 (
expr();//匹配 3
Match(input,13,FOLLOW_13_in_atom137)//匹配 )出现
MismatchedTokenException
```

我们看一下为什么输入(3+就会引发 NoViableAltException。分析器先匹配(，然后使用 expr 规则匹配 3 和符号+，从文法中可以看出在+之后可以匹配一个嵌套的(expr) 或 INT 注意这时是一个决定使用什么来匹配的时刻，但是没有符合候选的输入。下面是源代码中对应的部分。

```
public void atom() {
    if ( (LA5_0 == INT) ) { //匹配 INT
        alt5 = 1;
    }
    else if ( (LA5_0 == 12) ) {//匹配 (
    alt5 = 2;
    }
    else { //抛出 NoViableAltException 异常
        NoViableAltException nvae_d5s0 =
          new NoViableAltException("", 5, 0, input);
        throw nvae_d5s0;
    }
}
```

MismatchedNotSetException MismatchedRangeException MismatchedSetException 与 MismatchedTokenException 异常相似分析器处在抉择时都会引发 NoViableAlt Exception 异常而不会引发先面的四种异常。

9.2 错误恢复

ANTLR 开发的分析器并不是遇到错误就停止执行，当然分析器遇到错误后也可以提示错误信息后停止分析。但更好的方案是从错误中恢复继续分析并提示后面可能出现的错误。试想编译器每一次编译只能提示一个错误信息，用户修改然后再重新编译再修改下一个错误，这样一个个地改正错误肯定是不合理的。所以分析器要具有从错误之中恢复继续分析的能力。错误恢复是一种在分析中遇到如多余、缺少、错误的输入符号时越过错误符号尽可能找到后续匹配符号的行为。

我们可以看到 9.1.1 例中虽然我们输入了错误的句子分析器提示了出错信息，但是分析器还是可以找到 expr (3;。这说明分析器已经从错误中恢复并自动添加了缺少的“)”这样就找到了表示式(3)。下面我们来介绍一下 ANTLR 所开发的分析器的错误恢复能力。

9.2.1 异常的恢复策略

Mismatched 系列异常指的是前面我们讲过的以 Mismatched 开头的异常，ANTLR 分析器对 Mismatched 系列异常的恢复有两种方法，一种是删除符号继续分析，一种是插入符号继续分析。

9.2.1.1 删除符号恢复策略

当分析器遇到不匹配的 Mismatched 异常时，首先分析器采用删除符号继续分析的策略。查看当前符号的下一个符号是否匹配。如果匹配则认为当前符号为多余的符号，将当前符号删除然后继续分析。如前面的示例中输入(3)); 分析器遇到第二个) 时出错异常，这时查看其后的符号为 ；可以匹配 stat 规则所以这时) 被认为是多余符号。

9.2.1.2 插入符号恢复策略

如果删除符号策略不可行也就是说当前符号后面的符号无法匹配的情况下，分析器会采用插入符号的策略。插入符号策略是查看当前符号是否与是下一个要匹配的符号匹配，如果匹配分析器会认为当前错误是缺少符号，分析器会插入这个符号然后继续分析。使用前面的示例对于(3;的输入当遇到 ;时分析出错这时;后面没有符号所以不能用删除;的方法来恢复错误。但是 ;可以匹配 stat 规则说明当前缺少)符号，所以分析器会插入)恢复这个错误。

9.2.2 动态 Follow 集恢复策略

当分析时出现 NoViableAlt 异常或使用插入和删除符号的方法无法恢复错误时。分析器会采用动态 Follow 集恢复策略。一般编译原理书中介绍 LL 分析方法时会讲解 Follow 集，是一个符号的后续需要匹配的符号集合，举个简单的例子。

```
a : '[' b ']'
    | '(' b ')';
b : c '&';
c : ID;
```

c 的 Follow 集为['&']，b 的 Follow 集为**[**']', ')'**]**，ANTLR 在分析器时采用的是动态 Follow 集是在分析过程中动态计算出来的，而且 ANTLR 中的 Follow 集中不仅包含特定符号的直接 Follow 符号，还包括所有嵌套规则的 Follow 集的符号。比如本例的文法对于输入[ok&]，c 的动态 Follow 集为**[**'&', ']'**]**，因为 ok 符号的直接后续符号是'&'并且分析过程中选择的是 a 规则的第一个分支所以后续符号还有']'。下面完善方法：

```
grammar TestE7;
options {
   language=CSharp;
}
start : a+;
a : '[' b ']' {System.Console.WriteLine("[" + $b.text + "]");}
    | '(' b ')';
b : c '&';
c : ID;
ID : ('a'..'z' |'A'..'Z' )+ ;
WS : (' ' |'\t' |'\n' |'\r' )+ {Skip();};
```

a 规则中加入了 Action 可以显示是否找到了正确的句子。运行分析器后输入如下两行字符串：

```
[ok]
[ok&]
```

第一行是一个错误的句子，第二行是正确的句子。如果分析器可以在第一行的错误中恢复，分析器就一定可以找到第二行的句子。下面就是运行后分析器输出的结果：

```
[]
[ok&]
```

可以看到分析器从第一行的错误恢复并找到了第二个句子。下面我们仔细研究一下错误的恢复过程。当分析器分析“[”字符后由 a 规则进入 b 规则时分析器会将“]”字符插入动态到 Follow 集中，这时 Follow 集为**[[ ]**。下面的代码中显示出在进入规则函数 b()之前先调用 PushFollow 方法将当前分支的后续符号“]”放到 Follow 集中在，Follow 集在内部是以一个栈的形式存储的，PushFollow 作用是将“]”符号压栈。state.

followingStackPointer--;用来移动 Follow 栈的指针作用相当于是弹栈，在 b()函数完成后将“]”符号从栈中删除。

```
PushFollow(FOLLOW_b_in_a33);//压栈 ]
b1 = b();
state.followingStackPointer--;//弹出 ]
```

接着分析器进入 b()规则函数，然后直接进入 c 规则函数，在调用 c()函数的前后分析的代码为：

```
PushFollow(FOLLOW_c_in_b52);// 压栈 &
c();
state.followingStackPointer--;// 弹出 &
Match(input,10,FOLLOW_10_in_b54);//匹配&
```

我们可以看到进入 c 规则函数之前将“&” 压栈，这时 Follow 集为**[**], &**]**。接下来在 c 规则中输入的符号 ok 匹配 ID 规则然后从 c()中退出并企图匹配“&”。由于没有&所以分析会引发 MismatchedTokenException 异常，在默认情况下异常会被分析自身捕获并调用 Recover 恢复这个异常。

```
try
{
    PushFollow(FOLLOW_c_in_b56);
  c();
  state.followingStackPointer--;
        Match(input,10,FOLLOW_10_in_b58);  // 匹配“&”引发
                                    MismatchedTokenException
}
catch (RecognitionException re)
{
  ReportError(re);
  Recover(input,re);//恢复错误。
}
```

Recover 方法会从 Follow 集中扫描当前输入的句子。去寻找可以匹配的字符。本例中第一行输入中的最后“]”字符在 Follow 集中并且“]”在 a 规则中，所以这个错误可以在 a 规则中恢复。分析器找到一个不完整的句子之后就可以继续分析第二行的内容了。

9.3 提供友好的错误信息

良好的错误信息可以让用户迅速找出错误所在，更快地解决错误。良好的出错信息包括四方面的信息：出了什么错、哪出了错、出错的原因和如何解决。编写描述错误时我们要用既简短准确的语言，错误的描述说明出了什么错。而错误描述是否友好准确取决于开发者是否能够正确地获得错误。本章一开始提到过有一部分错误是开发者自己判断并抛出的，下面举一个这种情况的例子，比如我们开发一个命令行工具（Dos 常做的事）用来合并多

个文件成一个文件，命令格式为“combine 文件名, 文件名……”其中文件名必须是两个或多个。下面编写文法。

```
command : 'combine' FileName+;
FileName : NAME '. ' EXT;
```

根据上面的文法当用户输入一个文件名时并不违反语法规则，这就要开发者自己写代码来判断。下面给文法加入 Actions：

```
command : 'combine' (flist+=FileName)+
{ if($flist.Count == 1) {
  throw new Exception("only one FileName!");
 }
};
FileName : NAME '.' EXT;
fragment NAME : ('A'..'Z' | 'a'..'z') ('A'..'Z' | 'a'..'z'| '0'..'9') *;
fragment EXT : ('A'..'Z' | 'a'..'z') ('A'..'Z' | 'a'..'z'| '0'..'9')? ('A'..'Z'
| 'a'..'z'| '0'..'9')?;
WS : (' ' |'\t' |'\n' |'\r' )+ {Skip();} ;
```

这种自已抛出的异常在开发一个分析器时会大量使用，但是自己抛出的异常由于没有在 ANTLR 异常的基础上所以没有行列信息。用户不知道错误出在什么地方很难解决错误。下面我们改进一个文法使我们自己的异常也能有行列信息。

```
command : pos='combine' (flist+=FileName)+
{
 if($flist.Count == 1) {
   throw new Exception($pos.Line + ":" + ($pos.CharPositionInLine
 + 8)
                          + " 本命令行需要两个或两以上的文件名!");
 }
};
FileName : NAME '.' EXT;
……
```

解决办法将出错位置附近的符号的(IToken)赋予一个变量中，然后再读取这个变量的行列值。

还有一部分错误是来自分析器自身抛出的 ANTLR 异常。前一节已经介绍了 ANTLR 分析过程中可能抛出的各种异常。如何捕获这些异常并在这些异常的基础上给出更良好的错误信息提示给用户是非常重要的。第五章 5.14 catch[]与 finally 介绍了如何改变分析器默认的异常处理代码，下面我们来修改文法：

```
command : pos='combine' (flist+=FileName)+
{ if($flist.Count == 1) {
  throw new Exception($pos.Line + ":" + ($pos.CharPositionInLine + 8)
                          + " 本命令行需要两个或两以上的文件名!");
 }
```

```
};
catch[RecognitionException re] {
  if(re is EarlyExitException) {
     throw new Exception(re.Line + ":" + re.CharPositionInLine
                         + " 缺少文件名!");
  }
}
FileName : NAME '.' EXT;
```

先面讲了当一个多重性规则完全没有匹配时分析器会抛出 EarlyExitException，所以这时 catch 语句捕捉的异常如果是 EarlyExitException 说明用户没有输入任何文件名。这时可以给出相应的提示。注意 EarlyExitException 异常提供的列值一般为-1 这不利于正确地提示出错位置，所以我们还是用取附近一个词法符号的方式来提示位置。

9.3.1 使用自定义异常

一般在做开发时需要自定义异常类来描述与自己业务有关的异常，来与系统异常相区别，来可以自定义一些特定的属性来满足需要。自定义异常首先有一个自定义的异常基类。下面结合自定义异常修改完善前面的例子。捕获文件名是否合法，文件是否存在，缺少文件名，文件中个数是否正确四个错误。我们可以在工程中新建一个名为 Classes.cs 的文件将自定义的文件

```
public class CombineException : System.Exception
{
   public CombineException(string msg) : base(msg)
   {
   }
   private int line;
   public int Line
   {
      get { return line; }
      set { line = value; }
   }
   private int pos;
   public int Pos
   {
      get { return pos; }
      set { pos = value; }
   }
}
public class NotExistFileException : CombineException
{
   public NotExistFileException(string msg) : base(msg)
   {
```

```
    }
    private string notExistFileName;
    public string NotExistFileName
    {
        get { return notExistFileName; }
        set { notExistFileName = value; }
    }
}
public class IncorrectFileException : CombineException
{
    public IncorrectFileException(string msg) : base(msg)
    {
    }
    private string incorrectFieldName;
    public string IncorrectFieldName
    {
        get { return incorrectFieldName; }
        set { incorrectFieldName = value; }
    }

}
public class NoFileNameException : CombineException
{
    public NoFileNameException(string msg) : base(msg)
    {
    }
}
public class OnlyOneFileNameException : CombineException
{
    public OnlyOneFileNameException(string msg) : base(msg)
    {
    }
}
```

CombineException 类为本命令行所有自定义异常的基类，它继承了.net 的 Exception 类所以 CombineException 类也就成为了一个.net 的异常类，CombineException 类也定义了行列属性来标识出错位置。NotExistFileException 类表示文件不存在异常，这是一个自行判断获得的异常，因为分析器本身不可能知道文件是否存在。NotExistFileException 异常类有一个 NotExistFileName 属性用来保存这个不存在的文件名。IncorrectFileException 类表示文件名不正确（不合法的文件名）的异常，也有一个属性用来 NotExistFileName 不正确的文件名。NoFileNameException 类表示用户没有输入文件名的异常。OnlyOneFileNameException 类表示用户只输入了一个文件名的异常。

grammar TestE9;

```
options {
   language=CSharp;
}
@lexer::namespace {Chapter9Sample}
@parser::namespace {Chapter9Sample}
@lexer::members {
   public override void ReportError(RecognitionException e)
   {
      base.DisplayRecognitionError(this.TokenNames, e);
      throw e;
   }
}
FileName : NAME '.' EXT;
command : pos='combine'
       (flist+=FileName
       {
        if(System.IO.File.Exists($FileName.text) == false) {
            NotExistFileException notExistEx = new NotExistFileException(
                                $FileName.text + " file not exist.");
            notExistEx.NotExistFileName = $FileName.text;
            notExistEx.Line = $FileName.Line;
            notExistEx.Pos = $FileName.CharPositionInLine;
            throw notExistEx;
         }
       }
       )+
{
 if($flist.Count == 1) {
                  OnlyOneFileNameException onlyOneExt = new
OnlyOneFileNameException(
            $pos.Line + ":" + ($pos.CharPositionInLine + 8) + " only
            one FileName!");
   onlyOneExt.Line = $pos.Line;
   onlyOneExt.Pos = $pos.CharPositionInLine;
   throw onlyOneExt;
 }
};
catch[NoViableAltException re] {
   IncorrectFileException incorrectExt = new IncorrectFileException(
              re.Line + ":" + re.CharPositionInLine + " incorrect FileName!");
   incorrectExt.Line = re.Line;
   incorrectExt.Pos = re.CharPositionInLine;
   throw incorrectExt;
}
```

```
    catch[RecognitionException re] {
      if(re is EarlyExitException) {
         NoFileNameException noFileNameEx = new NoFileNameException(
                $pos.Line + ":" + $pos.CharPositionInLine + " have no
FileName!");
         noFileNameEx.Line = $pos.Line;
         noFileNameEx.Pos = $pos.CharPositionInLine;
         throw noFileNameEx;
      } else if(re is MismatchedTokenException) {
         IncorrectFileException incorrectExt = new IncorrectFileException(
               re.Line + ":" + re.CharPositionInLine + " incorrect FileName!");
         incorrectExt.Line = re.Line;
         incorrectExt.Pos = re.CharPositionInLine;
         throw incorrectExt;
      }
    }
    fragment NAME : ('A'..'Z' | 'a'..'z') ('A'..'Z' | 'a'..'z'| '0'..'9') *;
    fragment EXT : ('A'..'Z' | 'a'..'z') ('A'..'Z' | 'a'..'z'| '0'..'9')? ('A'..'Z' | 'a'..'z'|
'0'..'9')?;
    WS : (' ' |'\t' |'\n' |'\r' )+ {Skip();} ;
```

文法用中使用@lexer::namespace 和@parser::namespace 将分析器的命名空间设成 Chapter9Sample 与前面的自定义异常相同这样分析器就可以使用这些异常类 。OnlyOne FileNameException 和 NoFileNameException 的获得和抛出与以前一样只是换成了这两个新的异常类，并且赋予了 Line 和 Pos 属性。

文法中新加入了对 NotExistFileException 异常处理，由于 NotExistFileException 异常是自行获得的，所以在 FileName 后直接加入 Action 来使用 File. Exists 判断文件是否存在，行列就取当前 FileName 记号的行列值所以位置是很准确的。然后将 FileName 记号的 text 属性赋给异常的 NotExistFileNament 属性，用户就可以通过这个属性获得不存在的文件名是哪个文件（因为用户会输入多个文件），方便用户的使用。

最后介绍一下对错误文件名异常的处理（如+.txt 就是一个非法的），错误文件名就是指用户输入了非法的文件名。错误文件名异常的获得首先要捕获分析器本身抛出的 NoViableAltException 异常和 MismatchedTokenException 异常，因为文件名出错可能造成这两个异常，造成 MismatchedTokenException 异常很容易理解，而错造成 NoViableAltException 异常就有些奇怪其原因比较复杂，下面详细解释为何产生 NoViableAltException 异常和如何捕获 NoViableAltException 异常。

通过文法我们知道 command 规则中匹配 FileName 词法规则，FileName 词法规则又包含 NAME 和 EXT 词法规则。在规则函数 command()中有如下的代码。

```
    pos=(IToken)Match(input,8,FOLLOW_8_in_command60); //匹配 combine
    …
    do {
    …
```

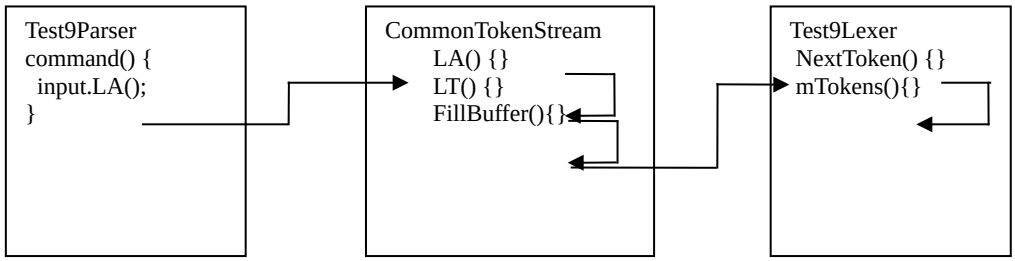
```
int LA1_0 = input.LA(1);//从 Token 流中获得一个记号
if ( (LA1_0 == FileName) ) {//判断是否文件名
   alt1 = 1;
}
```

语法分析类 Test9Parser 中的 input 是词法分析后生成的记号流一般为 CommonToken Stream 类继承自 ITokenStream 接口。CommonTokenStream 有 LA 方法用来获取当前的下一个记号。而 LA 方法中间接调用了词法分析程序（当前示例为 Test9Lexer）的 NextToken 方法，NextToken 方法又会调用 mTokens 方法。mTokens 方法是一个抽象方法，第一个具体的词法分析类都会有对这个方法有具体的实现。我们可以在 Test9Lexer 类中找到这个方法的实现。下面就是 Test9Lexer 类中的 mTokens()方法定义。

```
public override void mTokens() // throws RecognitionException
{   int alt5 = 3;
   alt5 = dfa5.Predict(input);
   switch (alt5) {
      case 1 : mT__8(); break;
      case 2 : mFileName(); break;
      case 3 : mWS(); break;
     }
}
```

可以看到 mTokens 方法是词法分析程序的一个总体调度方法由这里决定调用哪个词法规则函数，决策由 dfa5.Predict(input)语句进行。如果 Predict 方法没有找到匹配的分支就会抛出 NoViableAltException 异常。这就是不合法文件名造成 NoViableAltException 异常的原因，为了帮助理解给出下面的示意图，理解这个调用过程还对语法分析和词法分析之间如何交互理解有帮助所以很重要。

图 9.1



在了解了 NoViableAltException 如何抛出之后，一个新的问题就是如何捕捉这个异常。目录 antlr 版本中在默认情况下 NoViableAltException 异常并不会抛出到 Test9Parser，在 Test9Lexer 类的 NextToken()中的实现可以看到调用 mTokens()中捕获了异常但没有抛出，我们必须修改默认的实现让异常可以抛出。

```
public virtual IToken NextToken()

{
……
   try {
      this.mTokens();
```

```
            ……
        } catch (NoViableAltException exception)
        {//捕获了 NoViableAltException 异常但没有再抛出。
            this.ReportError(exception);
            this.Recover(exception);
        }
    ……
    }
```

ReportError()作用是报告错误，但没有继续抛出异常。不过 ReportError 是一个可重载的函数，我们可以在词法分析器中重载并继续抛出异常，下面是文法中重载的代码。

```
@lexer::members {
    public override void ReportError(RecognitionException e)
    {
        base.DisplayRecognitionError(this.TokenNames, e);
        throw e;
    }
}
```

可以抛出异常之后我们再加入 catch[NoViableAltException re]捕获这个异常就可以了。现在我们可以报告文件名不合法异常，但是并不能提示是哪个文件名不合法所以还不够理想。在很多时候上述引起的 NoViableAltException 异常可以提示出错误的发生位置。但无法获得出错位置的字符串。现在给出一个笨方法来获得出错位置的字符串，思路是把输入的字符串不但交给分析器分析还放到一个帮助类中保存，在这个帮助类中提供一个取得一个位置处字符串的方法。以供在分析器捕捉到异常时调用并提示给用户。

```
public static class inputText
{
    public static string Text;
    public static string GetToken(int x, int y, string sper)
    {
        string[] lines = Text.Split('\r', '\n');
        string line = lines[x - 1];
        string line2 = lines[x - 1].Substring(y - 1).Trim();
        string token = "";
        if (line2.IndexOf(" ") != -1)
            token = line2.Substring(0, line2.IndexOf(' '));
        else
            token = line2;
        return token;
    }
}
```

为了方便使用把类和方法定义成静态类。Text 字段用来保存用户的全部输入，GetToken 方法用来获得指定位置的一个以 sper 参数指定的字符串用为结束的子字符串。我们在分析之前把用户的输入赋给 Text 字段保存 inputText.Text = strInput;。然后分析器进行分析，下面是文法中对捕获 NoViableAltException 的代码修改。

```
catch[NoViableAltException re] {
    IncorrectFileException incorrectExt = new IncorrectFileException(re.Line
+ ":" + re.CharPositionInLine + " 不正确的文件名 " + i
nputText.GetToken(re.Line, re.CharPositionInLine, " ") + "!");
    incorrectExt.Line = re.Line;
    incorrectExt.Pos = re.CharPositionInLine;
    throw incorrectExt;
}
```

9.3.2 利用 antlr 异常的信息

在给用户提示错误信息时利用 antlr 异常的一些属性可能获得一些很关键的信息，比如前面示例中提示了的行列值。antlr 异常可利用的信息还有很多，比如 MismatchedToken Exception 异常的 UnexpectedType 属性用来指示当前遇到的不期待的符号的索引值，我们可以在.tokens 文件中找到相应的符号名。TestE2 文法分析器如下：

```
grammar TestE2;
typeCmd : 'type' fileName+;
fileName : ID;
ID : ('a'..'z' |'A'..'Z' )+ ;
WS : (' ' |'\t' |'\n' |'\r' )+ {skip();} ;
```

当输入 typ ff gg 时由于 type 少了一个 e 这会造成 MismatchedTokenException 异常，其 UnexpectedType 的值为 4，Expecting 属性的值为 6。我们看一下 TestE2.tokens 文件 4 是什么。

```
WS=5
ID=4
T__6=6
'type'=6
```

这时我们就可以向用户提示“当前期待 type 但遇到了 ID”，或是使用更好的提示“当前需要 type 关键字但遇到了标识符”。

NoViableAltException 异常和一些其它异常提供了 stateNumber、decisionNumber 和 grammar DecisionDescription。stateNumber 属性是状态数，decisionNumber 属性是决定数。grammarDecisionDescription 是关于些错误的提示信息。

```
grammar TestE10;
options {
   language=CSharp;
}
command : WS? 'dir' (WS path)? (WS param)? NEWLINE
{System.Console.WriteLine($path.text);
```

```
System.Console.WriteLine($param.text);};
param : SLASH2 PARAMCHAR;
path : DISK (SLASH NAME | SLASH)*;
DISK : ('A'..'Z' | 'a'..'z') ':';
PARAMCHAR   :  ('w' | 'p' | 'h' | 'a' | 'W' | 'P' | 'H' | 'A');
NAME : ('A'..'Z' | 'a'..'z' | '_' ) ('A'..'Z' | 'a'..'z'| '0'..'9' | '_' ) *;
//catch[NoViableAltException re] {
//}
SLASH : '\\';
SLASH2 : '/';
NEWLINE : '\r\n';
WS : (' ' |'\t')+;
```

9.3.3 嵌套规则中的异常处理

大多数文法中规则之间都存在多层的嵌套关系，我们要注意在合适的层次上处理错误。下面这个例子

```
grammar TestE
prog: stat+ ;
stat: expr ';'
        {System.out.println("found expr: "+$stat.text);}
        | ID '=' expr ';'
        {System.out.println("found assign: "+$stat.text);}
      ;
expr : multExpr (('+' |'-' ) multExpr)*;
multExpr : atom ('*' atom)*;
atom: INT
         | '(' expr ')';
ID : ('a'..'z' |'A'..'Z' )+ ;
INT : '0'..'9' + ;
WS : (' ' |'\t' |'\n' |'\r' )+ {skip();} ;
```

desionNumber antlrwork

合适的外层抛

异常后是否继续