第四章 语法分析

语法分析是编译过程的第二步，在词法分析提供的记号流的基础上，对源代码的结构做总体的分析。无论分析的内容有多大语法分析总是由一个启始规则开始的，最后总是生成一棵语法树。一般情况语法规则是一个文法的主体部分，也是编写文法的难点。本章用几个示例来讲述如何用 ANTLR 定义语法规则。

4.1 语法分析的方法

在 ANTLR 中语法分析定义的规则名必须以小写字母开始大写如“baseClass”，“subfixSymbol”。如果词法规则与语法规则写在同一个文件时，虽然 ANTLR 中并没有严格定义规则的先后顺序，但一般情况下语法规则写到词法规则的上面，因为整个文法的启始规则是从语法规则开始的，这样可以从上到下查看整个文法。

ANTLR 中语法定义的方法与词法基本相同请看下面一个 SQL 文法的片段示例:

```
grammar Test;

sqlStatement : selectStatement | insertStatement | deleteStatement;

selectStatement : SELECT (ALL | DISTINCT)?  SelectList FROM tableSource;

SelectList : SelectItem+;

tableSource : TableName | '(' selectStatement ') ';
```

4.2 递归定义

定义文法时通常出现递归的情况，比如上例中 tableSource 中的子查询就是一个递归定义。在 C++，C#，java 语言中类名这样的符号是用“.”分隔的多个标识符组成的，如 java.IO，System.Web.UI 等这种情况需要使用递归的方法来定义，递归有左递归和右递归。

左递归:

```
qualifiedName : qualifiedName '. ' Identifier;

qualifiedName : Identifier;

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

右递归:

```
qualifiedName : Identifier '.' QualifiedName
qualifiedName : Identifier;
Identifier : ('a'.. 'z' | 'A'.. 'Z' | '_') ('a'.. 'z' | 'A'.. 'Z' | '_'| '0'.. '9')*;
```

不过在 ANTLR 中不允许左递归定义，ANTLR 会提示:“rule is left-recursive”错误 。ANTLR 中还有别一种定义方法，象类名这样的符号递归定义使用这种方法是最好的方案。原因我们会在后面章节讲述。

```
qualifiedName : Identifier ('.' Identifier)*;
Identifier : ('a'.. 'z' | 'A'.. 'Z' | '_') ('a'.. 'z' | 'A'.. 'Z' | '_' | '0'.. '9')*;
```

4.3 java 方法的文法示例

下面来看一个定义 java 方法的文法，些文法是从 ANTLR 自带的 java.g 示例中选出的一段:

```
methodDeclaration :type Identifier
        (('(' (variableModifier* type formalParameterDeclsRest?)? ')')
            ('[' ']')*
        ('throws' qualifiedNameList)?
        (   methodBody
        |   ';'
        )) ;
formalParameterDeclsRest :
                variableDeclaratorId
                (',' (variableModifier* type formalParameterDeclsRest?))?
                |   '...' variableDeclaratorId
                ;
type : Identifier (typeArguments)?
                    ('.' Identifier (typeArguments)? )* ('[' ']')*
                |                           primitiveType ('[' ']')*
                ;
variableModifier : 'final'  |  annotation ;

formalParameterDeclsRest :
```

```
            variableDeclaratorId
            (',' (variableModifier* type formalParameterDeclsRest?))?
            |  '...' variableDeclaratorId
            ;
variableDeclaratorId : Identifier ('[' ']')* ;

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

typeArguments : '<' typeArgument (',' typeArgument)* '>' ;

typeArgument : type
        | '?' (('extends' | 'super') type)? ;

qualifiedNameList :qualifiedName (',' qualifiedName)* ;

qualifiedName :Identifier ('.' Identifier)* ;
```

methodDeclaration 为方法定义的起始规则，后面 type Identifier 为方法的返回值和方法名，('(' (variableModifier* type formalParameterDeclsRest?)? ')') ('[' ']')* 中的 variableModifier 为参数前的标识符，type 为参数的类型 formalParameterDeclsRest 是对参数其余部分的定义，参数表在“（）”中，后面是“[]”字符，这是数组类型为返回值时需要的，因为数据可以是多维的后是用('[' ']')*来定义。

formalParameterDeclsRest 用来定义参数表部分，variableDeclaratorId 为参数名，如何有一个以上参数(',' (variableModifier* type formalParameterDeclsRest?))?部分字义了第二个到第 N 个参数的规则，这里使用了右递归方法。Java1.5 中支持可变参数 '...' variableDeclaratorId 为可变参数的定义。

另外 typeArguments 配合 Identifier 来定义泛型类型，使用“<>”括起来一个类型列表，如 List<string> 。 typeArgument 与 type 形成了递归定义，如 Hashtable<string,List<stirng>>。

4.4 | 作用范围

上例中的 typeArgument：type | '?' (('extends' | 'super') type)? 规则无须写成 typeArgument:type | ('?' (('extends' | 'super') type)?)。在同一规则或子规则中“|”使其两侧的内容为并列的选择关系，如果有多个“|”则一起为并列的选择关系，需要改变优先顺序时才使用“（）”。

4.5 SELECT 语句文法示例

下面给出一个 SQL SELECT 语句文法片段的例子。

grammar Select;

statement

: selectStatement (SEMICOLON)?;

selectStatement

: queryExpression (computeClause)? (forClause)? (optionClause)?;

queryExpression

: subQueryExpression (unionOperator subQueryExpression)* (orderByClause)?

;

subQueryExpression

: querySpecification | '(' queryExpression ')'

;

querySpecification

: selectClause (fromClause)? (whereClause)? (groupByClause (havingClause)?)?

;

selectClause

: SELECT (ALL | DISTINCT)? (TOP Integer (PERCENT)?)? selectList

;

whereClause

: WHERE searchCondition

;

orderByClause

: ORDER BY expression (ASC | DESC)? (COMMA expression (ASC | DESC)?)*

;

groupByClause

:GROUP BY (ALL)? expression (COMMA expression)* (WITH (CUBE |

```
                                    ROLLUP) )?
    ;
havingClause
    : HAVING searchCondition
    ;
SEMICOLON : ';';
```

我们对 SELECT 语句都比较熟悉，看一下 SELECT 语句文法的大体结构。selectStatement 规则表示整个 SELECT 语句体系，queryExpression 表示查询语句可能由多个子查询用 UNION 到一起 unionOperator 代表 UNION 或 UNION ALL 关键字，orderByClause 子句出现在 SELECT 语句的最后，这里 subQueryExpression 和 queryExpression 之间形成了递归定义，子查询中还可以有子查询。whereClause 子句和 havingClause 子句后面都是查询过滤表达式后成它们共用 searchCondition 规则。

4.6 HTML 文法示例

下面再看一个 HTML 文法片段示例:

```
grammar HTML;
options  {language=CSharp; output=AST;}
document :    OHTML  body  CHTML;
body :        obody  (body_content)*  CBODY ;
body_content :  body_tag | text;
body_tag :    block; // | heading | ADDRESS
text :    text_tag;
text_tag :      form;// phrase | special | font
block :      table;//paragraph | list | preformatted | div | center | blockquote
                                    | HR |
table :     otable (tr)+ CTABLE;
tr :     o_tr (th_or_td)* (C_TR)? ;
th_or_td :     o_th_or_td (body_content)* (C_TH_OR_TD )?;
form :     oform (form_field | body_content)* CFORM;//
oform :     '<form' (ATTR)* '>';
form_field :    inputField;// | select | textarea;
inputField :  '<input' (ATTR)* '>';
```

```
obody :   '<body' (ATTR)* '>';

CBODY :    '</body';
OHTML :    '<html>';
CHTML :    '</html>';
otable :    '<table' (ATTR)* '>';
CTABLE :    '</table>';
o_tr :    '<tr' (ATTR)* '>';
C_TR :    '</tr>';
o_th_or_td :    ('<th' | '<td') (ATTR)* '>';
C_TH_OR_TD :  '</th>' | '</td>';
CFORM :    '</form>';
ATTR :    WORD ('=' (WORD ('%')? | ('-')? INT | STRING | HEXNUM))?;
HEXNUM :    '#' ('0'..'9' | 'a'..'f')+;
INT :    (DIGIT)+;
DIGIT :    '0'..'9';
WORD :    (LCLETTER | '.' | '/')  (LCLETTER | DIGIT | '.')+;
STRING :    '"' (~'"')* '"' | '\'' (~'\'')* '\'';
LCLETTER :    'a'..'z';
WS : ( ' ' | '\t' | '\n' | '\r' ) + {Skip();} ;
```

这个可运行的简化的 HTML 文法中只支持<form>表单和<table>表格。O 开头的符号如 OHTML、otable 表示标记开头<html>和<table>，C 开头的符号是表示结束标记如:CHTML，CTABLE 表示</html>和</table>。document 规则表示整个 HTML 文档其包括 body 部分。Body 主要包括两种内容 text 和 body_tag，body_tag 中包括<table>表格，text 中包含<form>表单。词法规则可以和语法规则混合书写这是允许的。

文法中的 th_or_td 规则表示表格中的一个 cell，其中又包含了 body_content 这个递归的定义使得表格中的第一个小格都可以嵌套的包含 HTML 标记。生成分析器后可以对下面的 HTML 文件进行分析。

```
<html>
<body id="doc">
<table>
  <tr>
    <td>
```

```
        <form action="login.jsp" >
        <input id="txt1" value="" />
      </td>
    </tr>
</table>
</form>
</body>
</html>
```

4.7 Skip()的效果

在 HTML 文法示例中的开始标记的定义是用语法规则定义的，而结束是用词法规则定义的 ANTLR 中规则可以灵活定义不拘一格，结束标记只是固定的字符串定义成词法规则就可以了。而开始标记中要包含属性的定义所以用语法规则来定义。其中有一点要注意的是如果我们把开始标记也定义成词法规则会出现什么情况呢？如下面改变一下 HTML 文法。

OFORM : ‘<form’ (ATTR)* ‘>’

将<form>表单的开始标记改为词法规则，这时分析器不能正确分析上面的 html 文件。在分析<form ation=””>行时异常。运行这个示例的语句如下:

HTMLLexer lex = new HTMLLexer(new ANTLRFileStream("t.html"));

ITokenStream tokens = new CommonTokenStream(lex);

HTMLParser parser = new HTMLParser(tokens);

HTMLParser.document_return dReturn = parser.document();

这是因为空白规则 WS 中的 Skip()语句的效果是对语法分析而言的。在词法分析阶段 Skip()并没有去掉空格，我们可以 java 和.net 的开发环境中查看 tokens 对象了解词法分析阶段的结果，这可以帮助我们分析问题。对于输入的“<form ation="">”来说 form 和 ation 之间有一个空格这时词法分析器会失败，因为我们在这个规则中没有定义空白，除非我们这样写这个词法规则。

OFORM : ‘<form’ WS (ATTR)* WS? ‘>’

把所有可能出现空格地方都加下显示地定义空白才行。这也就是我们在上一章中的 FuzzyJava2 示例中为什么词法规则中加了很多 WS 的原因。

4.7 语法规则中的字符常量

可能读者早已发现我们在语法规则中直接写入了需要匹配的字符。如:

oform : '<form' (ATTR)* '>';

‘<form’和‘>’都属于词法范畴，ANTLR 为了使书写简单直观，允许在语法规则中直接写出需要匹配字符。在生成分析器代码时这些常量会自动被放入词法分析程序中，语法分析程序中的字符串会用生成的序号代替。oform : '<form' (ATTR)* '>'与下面的写法是等价的。

oform : t10 (ATTR)* t11;

t10 :‘<form';

t11 : ‘>';

4.8 C-语言示例

C-语言是 C 语言的一个子集，是 ANTLR 的一个经典示例。下面看一下它的文法。

```
grammar CMinus;
program
   :  declaration+
   ;
declaration
   :  variable  |  function
   ;
variable
   :  type ID ';'
   ;
type:  'int'  |  'char'
   ;
function
   :  type ID
      '(' ( formalParameter (',' formalParameter)* )?  ')'
      block
   ;
formalParameter
   :  type ID
   ;
```

```
block
   :   '{' variable* stat* '}'
   ;
stat
     : forStat  |  ifStat  |  expr ';'  |  block  |  assignStat ';'  |  ';'
   ;
ifStat
   : 'if'  '('  expr  ')'  stat  ('else'  stat)?
   ;
forStat
   :  'for' '(' assignStat ';' expr ';' assignStat ')'  block
   ;
assignStat
   :   ID '=' expr
   ;
expr:   condExpr ;

condExpr
   :   aexpr ( ('==' | '!=' ) aexpr )?
   ;
aexpr
   :   mexpr ('+'  mexpr)*
   ;
mexpr
   :   atom ('*'  atom)*
   ;
atom:   ID
   |   INT
   |   '(' expr ')'
   ;
ID  :   ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')* ;
INT :   ('0'..'9')+ ;
```

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

启始规则 program 表示整个 C-程序，declaration 表示语言中的声明项，有函数和变量两种声明 variable 和 function。变量声明由类型 type 加标识符 ID 组成，function 函数声明是由返回类型 type 函数名 ID 和参数表'(' (formalParameter (',' formalParameter)*)? ')'，block 为函数体。函数体内可以有变量 variable 和语句 stat。

语句 stat 包括 for 语句 forStat、表达式 expr、语句块 block 和赋值语句 assignStat。表达式 expr 和第一章中的 HelloWorld 示例类似推导顺序与操作符优先顺序同反，在表达式的末端规则 atom 有可出现标识符 ID、整数 INT 和嵌套的子表达式，语句块 block 与函数体 block 是同一规则，这是一个递归定义使得语句块可以嵌套书写。

生成分析器代码编译运行，来分析如下的 C-代码。

```
char c;
int x;
int foo(int y, char d) {
  int i;
  for (i=0; i!=3; i=i+1) {
    x=3;
    y=5;
  }
  if(5 == 4)
      if(5 == 4)
          i = 1;
       else
          i = 2;
}
```

4.9 小结

本章讲述了 ANTLR 的语法分析，语法分析中的递归定义的应用，用 SQL、java、HTML 和 C-几个实际的例子来让读者对语法分析有更好的理解。通过对 Skip() 效果的示例说明了词法部分的操作与语法分析的关系。