实验五、语法分析实验

> *学号：1120180207*
>
> *姓名：唐小娟*
>
> *班级：07111801*

1. 实验目的

（1）熟悉C语言的语法规则，了解编译器语法分析器的主要功能；

（2）熟练掌握典型语法分析器构造的相关技术和方法，设计并实现具有一 定复杂度和分析能力的C 语言语法分析器；

（3）了解 ANTLR 的工作原理和基本思想，学习使用工具自动生成语法分析器；

（4）掌握编译器从前端到后端各个模块的工作原理，语法分析模块与其他模块之间的交互过程。

2. 实验内容

该实验选择 C 语言的一个子集，基于 BIT-MiniCC 构建 C 语法子集的语法分析器，该语法分析器能够读入词法分析器输出的存储在文件中的属性字符流，进行语法分析并进行错误处理，如果输入正确时输出 JSON格式的语法树，输入不正确时报告语法错误。

提交的文件包括：

src下所有源码：其中txjparser、txj_parse.g4、OutputJson.java、txjParseMain.java是本人写的。

config.xml、classpath文件（anltr4采用的4.9.2版本，所以在lib和classpath中我更改了版本）

本实验主要包括两部分：

1. ANTLR4自动生成工具生成了语法分析器。
2. 通过创建OutputJosn类（继承txj_parseBaseVisitor类）修改遍历语法树的方式，生成与老师所给json文件格式相同的语法树。

   注意：最后对nc_tests中的样例代码进行语法分析后得到的json文件和老师执行结果完全一致，基本目标已经达成。

3. 语法分析

采用ANLTR自动生成工具，编写相应的txj_Parse.g4文件，生成语法分析器。由于在实验四中已经完成了文法设计实验，g4文件只需要修改部分格式即可。但是为了符合老师ast文件中的类，也修改了相应的文法。由于和实验四有重合的部分，这里只介绍部分关键的代码。

注意：ANLTR4支持左递归和EBNF，语法定义首字母小写，词法定义大写。除此之外，语法定义中的每一个产生式后面可跟#标签，在之后的遍历中，利用此标签执行相应的函数。

3.1 Program

C语言程序主要是由functionDefine和declaration组成，也就是函数定义和外部变量声明。这也是语法分析的根节点。

```
program:translationUnit?EOF;
translationUnit: externalDeclaration+;
externalDeclaration:functionDefine
                    |declaration ';';
```

3.2 FunctionDefine

函数定义包含了返回类型Type、说明符、参数列表、复合语句等，这里的参数列表可为空。

```
functionDefine:Type declarator '(' arguments? ')'
body=compoundStatement;
arguments : fundeflist?;
fundeflist : declarator
           |declarator ',' fundeflist
           ;
```

3.3 Declarator

声明这里主要包括了四部分，分别是标变量、数组、函数声明、参数声明。

```
//声明
declarator : Identifier
    #VariableDeclarator
           |declarator '[' expr ']'
   #ArrayDeclarator
           |declarator '(' arguments? ')'
   #FunDeclarator
           |Type declarator
   #ParamsDeclarator
           ;
```

3.4 Stat

语句这里基本包括了11中，其中还有空语句。

```
stat : breakStatement
      |continueStatement
      |gotoStatement
      |returnStatement
      |labeledStatement
      |compoundStatement
      |selectionStatement
      |expressionStatement
      |iterationStatement
      |iterationDecStatement
      |';'
      ;
```

3.5 Expr

表达式包括了前缀表达式、二元表达式、一元表达式、条件表达式以及常量等。刚开始我是采用递归层层递进的方式，但是ANTLR可能在某些递归层数过多的情况下出现错误，之后查找资料发现，默认前面的优先级大于后面的，所以我就直接采用了二元的方式来表示。

```
expr : expr op = ('++'|'--')
    #Postfix_expression
     | expr '[' exprs ']'
   #Array_access
     | op = ('++'|'--'|'sizeof'|Unary_operator) expr
    #Unary_expression
     | op = 'sizeof' '(' typename ')'
   #Unary_typename
     | '(' typename ')' expr
    #Cast_expression
     | expr op = ('*'|'/'|'%') expr
   #Binary_expression
     | expr op = ('+'|'-')  expr
    #Binary_expression
     | expr op = ('<<'|'>>') expr
   #Binary_expression
     | expr op = ('>'|'<'|'<='|'>=') expr
   #Binary_expression
     | expr op = ('=='|'!=') expr
   #Binary_expression
     | expr op = '&' expr
   #Binary_expression
     | expr op = '^' expr
   #Binary_expression
     | expr op = '|' expr
   #Binary_expression
     | expr op = '&&' expr
    #Binary_expression
```

```
15      | expr op = '||' expr
     #Binary_expression
16      | expr '?' trueExpr=exprs ':' falseExpr=expr
    #Conditional_expression
17      | expr op = Assignment_operator expr
     #Binary_expression
18      | functionCall
     #MyFunctionCall
19      | IntegerConstant
     #IntegerConstant
20      | CharConstant
     #CharConstant
21      | FloatConstant
     #FloatConstant
22      | Identifier
     #Identifier
23      | StringLiteral
     #StringLiteral
24      |'(' expr ')'
     #MyExpr
25      ;
```

4. 输出格式

将自己的语法树转化为老师的抽象语法树，主要采用了visitor的遍历方式，visitor具有返回值，同时可以按照自己设定的遍历方式进行遍历。主要思想就是递归。接下来以几个简单的例子来说明我设计的过程。

4.1 Program

根据我的文法设计：`program:translationUnit?EOF;`先创建一个对象，访问Program的孩子节点translationUnit得到ASTCompilationUnit p的成员ArrayList<ASTNode> f1，并将得到的孩子节点添加到p中。代码如下：

```
1  public ASTNode visitProgram(txj_parseParser.ProgramContext ctx)
2  {
3          ASTCompilationUnit p = new ASTCompilationUnit();
4          if(ctx.translationUnit()!=null)
5          {
6              ArrayList<ASTNode> f1 = (ArrayList<ASTNode>)
   visitTranslationUnit(ctx.translationUnit());
7              p.items.addAll(f1);
8              p.children.addAll(p.items);
9          }
10         return p;
11 }
```

4.2 FunctionDefine

在函数定义中：`functionDefine:Type declarator '(' arguments? ')' body=compoundStatement;`创建一个functionDefine对象，由于Type是叶子节点，直接获得Token之后转化为ASTToken，之后访问 declarator和arguments节点。

```
        public ASTFunctionDefine visitFunctionDefine(txj_parseParser.FunctionDefineContext ctx) {
        if(ctx==null)
            return null;
        ASTFunctionDefine fdef = new ASTFunctionDefine();
        Token spetoken = ctx.Type().getSymbol();
        ASTToken s = new ASTToken();
        s.tokenId = spetoken.getTokenIndex();
        s.value = spetoken.getText();
        fdef.specifiers.add(s);

        ASTFunctionDeclarator fdec = new ASTFunctionDeclarator();
        ASTDeclarator dec = (ASTDeclarator) visit(ctx.declarator());
        fdec.declarator = dec;
        fdef.children.add(dec);
        ArrayList<ASTParamsDeclarator> pl = visitArguments(ctx.arguments());
        if(pl!=null)
        {
            fdec.params.addAll(pl);
            fdec.children.addAll(pl);
        }
        ASTCompoundStatement cs = visitCompoundStatement(ctx.compoundStatement());

        fdef.declarator = fdec;
        fdef.children.add(fdec);
        fdef.body = cs;
        fdef.children.add(cs);
        return fdef;
    }
```

4.3 Declarator

根据文法设计中，声明有四种情况，所以在代码中，重写了四个函数分别是：visitVariableDeclarator、visitParamsDeclarator、visitArrayDeclarator、visitFunDeclarator，部分代码如下：

```
public ASTVariableDeclarator visitVariableDeclarator(txj_parseParser.VariableDeclaratorContext ctx) {
        ASTVariableDeclarator v = new ASTVariableDeclarator();
        ASTIdentifier t = new ASTIdentifier();
```

```
        Token vardec = ctx.Identifier().getSymbol();
        t.tokenId = vardec.getTokenIndex();
        t.value = vardec.getText();
        v.identifier = t;
        return v;
    }


 public ASTParamsDeclarator
visitParamsDeclarator(txj_parseParser.ParamsDeclaratorContext
ctx) {
        ASTParamsDeclarator t = new ASTParamsDeclarator();
        Token Typetoken = ctx.Type().getSymbol();
        ASTToken s = new ASTToken();
        s.tokenId = Typetoken.getTokenIndex();
        s.value = Typetoken.getText();
        t.specfiers.add(s);

        ASTDeclarator dec = (ASTDeclarator)
visit(ctx.declarator());
        t.declarator = dec;

        return t;

    }
```

4.4 Stat

和上面内容同理，由于语句可以有多种，比如返回、选择、循环等等，而老师所给类ASTStatement为虚类，所以重写每一类的函数，获得相应的节点内容。例如代码展示的返回语句，这里要注意一点，返回值可以是空，所以要判断是否为null。

```
public ASTReturnStatement
visitReturnStatement(txj_parseParser.ReturnStatementContext
ctx) {
        ASTReturnStatement rs = new ASTReturnStatement();
        if(ctx.exprs()!=null)
        {
            LinkedList<ASTExpression> exprs = new
LinkedList<ASTExpression>();
            List<ASTExpression> myexprs =
visitExprs(ctx.exprs());
            for(int i= 0;i<myexprs.size();i++)
                exprs.add(myexprs.get(i));
            rs.expr = exprs;
        }

        else
            rs.expr = null;


```

```
        return rs;
    }
```

5. 运行结果

执行老师nc_test目录下的8个文件，皆和老师运行结果一致。

测试文件：

```
int a[10] = {1,4,7,2,3,0,8,5,9,6};
void quick(int start,int end);
int partion(int low,int high);
void quickSort(int len){
    quick(0,len-1);
    return;
}
void quick(int start,int end){
    int par = partion(start,end);
    if(par > start + 1){
        quick(start, par-1);
    }
    if(par < end - 1){
        quick(par+1, end);
    }
    return;
}
int partion(int low, int high){
    int tmp = a[low];
    for(; low < high; ){
        for(; low < high && a[high] > tmp;){
            high --;
        }
        if(low >= high){
            break;
        }else{
            a[low] = a[high];
        }
        for(; low < high && a[low] < tmp;){
            low ++;
        }
        if(low >= high){
            break;
        }else{
            a[high] = a[low];
        }
    }
    a[low] = tmp;

    return low;
}
int main(){
```

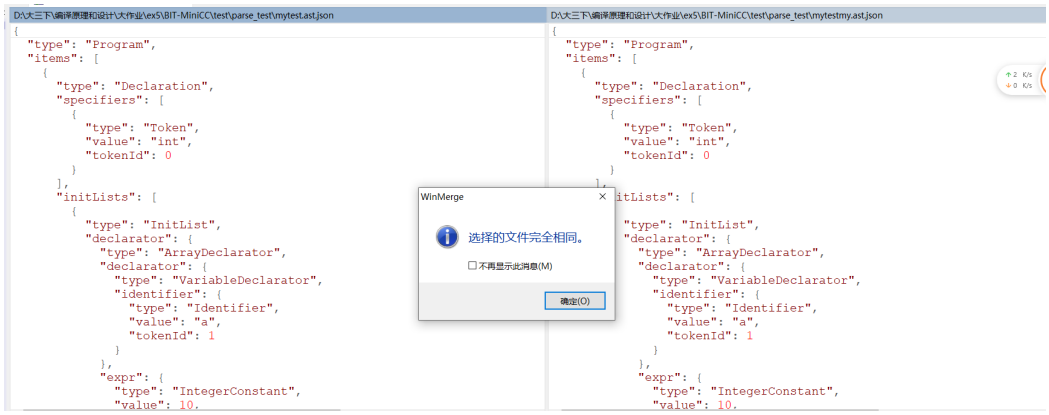
```
43      Mars_PrintStr("Before quicksort:\n");
44      for(int i = 0; i<10; i++){
45          Mars_PrintInt(a[i]);
46      }
47      quickSort(10);
48      Mars_PrintStr("\nAfter quicksort:\n");
49      for(int i = 0; i<10; i++){
50          Mars_PrintInt(a[i]);
51      }
52      return 0;
53  }
```

我的结果：

老师的结果：

```
{
  "type": "Program",
  "items": [
    {
      "type": "Declaration",
      "specifiers": [
        {
          "type": "Token",
          "value": "int",
          "tokenId": 0
        }
      ],
      "initLists": [
        {
          "type": "InitList",
          "declarator": {
            "type": "ArrayDeclarator",
            "declarator": {
              "type": "VariableDeclarator",
              "identifier": {
                "type": "Identifier",
                "value": "a",
                "tokenId": 1
              }
```


在WinMerge软件中比较这两个文件，显示完全相同。



5. 实验心得

在本次实验中，我认为语法分析器的生成时不难的，因为有自动生成工具ANTLR4，而且由于idea中含有相关插件，可视化效果相当好，调试过程十分便利，因此简单生成语法分析器并不困难。

然而对我来说，具有挑战性的是如何转换成和老师格式一致的json文件。主要难点在于：

1. 文法设计中和老师的类不一致。
2. ANLTR中虽然可以生成json文件，但是根据的是它内嵌的遍历树的方式，所以要和老师生成一样的json文件是很困难的。
3. 网上对visitor和listenor的教程很零碎，学习过程有一定的难度。

最后我想说一下我解决这些问题的心路历程：

刚开始我查阅资料，看到网上有生成相应json文件的代码，我采用了他们的思想，但是由于在输出的类的名称上不一样（毕竟没有改变遍历节点的每一个类），所以这个解决办法行不通。

但是那份代码给了我灵感，因为它在主函数采用遍历树的方式去生成相应json文件的，所以我的突破点就是在遍历中，构造相应的类。但是困难又来了，我怎么判断遍历的节点是属于哪一类呢，难道在每次遍历的时候，我都要switch判断吗，这些变成体力活了，所以我不到万不得已不准备做这件事。

之后我又找网上的资料，看到一份计算器的样例，主要是通过visitor遍历树，对相应的产生式进行加减运算并返回值。这太棒了，因为我不需要再去判断哪个节点是哪一个节点了。

可是我高兴的太早了，因为我发现继承visitor类去更改遍历树的方式，不可以返回不同类型（有些是ArrayList类型），而listenor中又不能返回值，而且它不能去更改遍历方式，默认使用的是内嵌的遍历方式，虽然这样也可以使用堆栈的方法进行，可是我认为这个方式操作上很容易出bug。

之后，我和同学讨论我解决这个问题的想法，他后来尝试了一下，发现visitor可以返回不同类型！只要把模板的<T>删掉就好了！太惊喜了！于是这个问题我就有了头绪。

在写的过程我参考了老师给的ExampleParse的文件，里面虽然是用底层算法进行语法分析的，但是在返回相应类和节点的方式，和我的想法很相像，所以刚开始我模拟老师的方式，虽然有很多迷惑，遇到bug的时候，我就用很多不同的方法尝试，后来写了将近400多行的代码的时候，对这个遍历机制以及类的构造过程中有了更深刻的理解，后面写代码的速度也越来越快了。

总之，这次实验虽然我花了很多时间在输出格式上，也听说了周围同学采用了输出字符串的方式（我觉得太逆天了），中间也想过放弃，但是再一次当我去解决这些难题，踩在老师代码和网上样例代码的肩膀上，成功实现了。这带给我的不仅仅是这个实验的技能，更有解决问题的思路。这还是编译器的第二步，接下来的实验有更大的挑战，所以我需要更加认真学习编译知识。