Lab.5 语法分析实验报告

一、实验目的和内容

1. 实验目的

- 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。
- 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。
- 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。
- 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 BITMiniCC 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。

二、实验过程和步骤

1. 语法设计

对实验中的文法进行了补全和修正，支持各种运算表达式、基本变量声明、数组声明、赋值语句、循环语句、分支语句等。

部分语法规则如图 1 所示。

```
grammar MyCGrammar;

primaryExpression
    :   tokenId//Identifier
    |   tokenConstant//Constant
    |   tokenStringLiteral//StringLiteral+
    |   '(' expression ')'
    |   genericSelection
    |   '__extension__'? '(' compoundStatement ')' // Blocks (GCC extension)
    |   '__builtin_va_arg' '(' unaryExpression ',' typeName ')'
    |   '__builtin_offsetof' '(' typeName ',' unaryExpression ')'
    ;
tokenId
    : Identifier
    ;
tokenConstant
    : Constant
    ;
tokenStringLiteral
    : StringLiteral+
    ;

genericSelection
    :   '_Generic' '(' assignmentExpression ',' genericAssocList ')'
    ;

genericAssocList
    :   genericAssociation
    |   genericAssocList ',' genericAssociation
    ;

genericAssociation
    :   typeName ':' assignmentExpression
    |   'default' ':' assignmentExpression
    ;
```

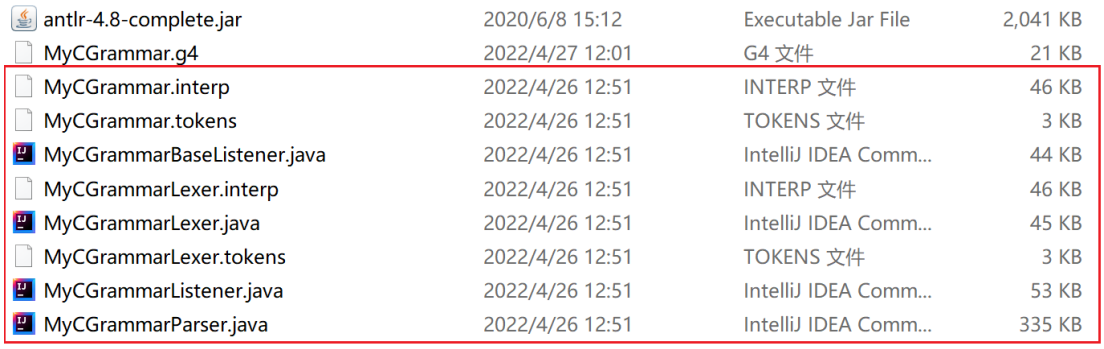
图 1：语法规则

2. 自动生成

将 BITMiniCC/lib 中的 antlr-4.8-comple.jar 复制到 MyCGrammar.g4 所在路径下，然后在该路径下启动控制台，执行命令

```
java -jar antlr-4.8-comple.jar MyCGrammar.g4
```

生成的文件如图 2 所示。



antlr-4.8-complete.jar	2020/6/8 15:12	Executable Jar File	2,041 KB
MyCGrammar.g4	2022/4/27 12:01	G4 文件	21 KB
MyCGrammar.interp	2022/4/26 12:51	INTERP 文件	46 KB
MyCGrammar.tokens	2022/4/26 12:51	TOKENS 文件	3 KB
MyCGrammarBaseListener.java	2022/4/26 12:51	IntelliJ IDEA Comm...	44 KB
MyCGrammarLexer.interp	2022/4/26 12:51	INTERP 文件	46 KB
MyCGrammarLexer.java	2022/4/26 12:51	IntelliJ IDEA Comm...	45 KB
MyCGrammarLexer.tokens	2022/4/26 12:51	TOKENS 文件	3 KB
MyCGrammarListener.java	2022/4/26 12:51	IntelliJ IDEA Comm...	53 KB
MyCGrammarParser.java	2022/4/26 12:51	IntelliJ IDEA Comm...	335 KB

图 2：自动生成

将生成的文件和 antlr-4.8-complete.jar 构建为 MyCGrammar.jar，并添加为 BITMiniCC 的库文件。

3. 词法分析

在软件包 bit.minisys.minicc.scanner 中新建 Java 类 MyParser。

为了嵌入 BITMiniCC 框架，MyParser 类实现了 IMiniCCScanner 中的 run 方法，用于调用 ANTLR 自动生成的词法分析器 MyCGrammarLexer。

源码如图 3 所示。

```java
package bit.minisys.minicc.scanner;

import MyCGrammar.MyCGrammarLexer;
import bit.minisys.minicc.MiniCCCfg;
import bit.minisys.minicc.internal.util.MiniCCUtil;
import org.antlr.v4.runtime.ANTLRInputStream;
import org.antlr.v4.runtime.CommonTokenStream;

import java.io.FileInputStream;
import java.io.FileWriter;

public class MyScanner implements IMiniCCScanner{
    @Override
    public String run(String iFile) throws Exception {
        MyCGrammarLexer lexer = new MyCGrammarLexer(new ANTLRInputStream(new FileInputStream(iFile)));
        CommonTokenStream tokens = new CommonTokenStream(lexer);

        String oFile = MiniCCUtil.removeAllExt(iFile) + MiniCCCfg.MINICC_SCANNER_OUTPUT_EXT;
        FileWriter fileWriter = new FileWriter(oFile);
        for (int i = 0; i < tokens.getNumberOfOnChannelTokens(); i++) {
            fileWriter.write(tokens.get(i).toString());
            fileWriter.write( str: "\n");
        }
        fileWriter.close();

        return oFile;
    }
}
```

图 3：MyScanner 源码

4. 语法分析

在软件包 bit.minisys.minicc.scanner 中新建 Java 类 MyListener，继承了由 ANTLR 自动生成的虚类 MyCGrammarBaseListener，并实现了其中的方法。

部分源码如图 4 所示。

```java
public class MyListener extends MyCGrammarBaseListener {
    public String oFile;
    public Stack<ASTNode> NodeStack = new Stack<>();

    @Override
    public void enterCompilationUnit(MyCGrammarParser.CompilationUnitContext ctx) {
        ASTCompilationUnit node = new ASTCompilationUnit();
        NodeStack.push(node);
    }

    @Override
    public void exitCompilationUnit(MyCGrammarParser.CompilationUnitContext ctx) {
        ASTNode node = NodeStack.pop();
        node.children.addAll(((ASTCompilationUnit) node).items);

        ObjectMapper mapper = new ObjectMapper();
        try {
            mapper.writerWithDefaultPrettyPrinter().writeValue(new File(oFile), node);
        } catch (IOException e) {
            e.printStackTrace();
        }
        NodeStack.push(node);
    }

    @Override
    public void enterInitDeclaratorList(MyCGrammarParser.InitDeclaratorListContext ctx) {
        ASTNode node = new ASTInitList();
        NodeStack.push(node);
    }

    @Override
    public void exitInitDeclaratorList(MyCGrammarParser.InitDeclaratorListContext ctx) {
        ASTNode node = NodeStack.pop();
        ASTNode ParentNode = NodeStack.peek();
        if (((ASTInitList) node).declarator != null) {
            node.children.add(((ASTInitList) node).declarator);
        }
        if (((ASTInitList) node).exprs != null) {
```

图 4：MyListener 源码

然后在同一软件包中新建 Java 类 MyParser，并实现接口 IMiniCCParser 中的 run 方法。

在 run 方法的实现中调用了由 ANLTR 自动生成的语法分析器 MyCGrammarParser，得到测试用例对应的具体语法树。然后调用 ANTLR 运行时 API 中的 ParseTreeWalker 中的 walk 方法。walk 方法会根据参数中传入的 MyListener 实例，按照 MyListener 类中各方法定义的规则遍历具体语法树，生成相应的 AST 树。

源码如图 5 所示。

```java
public class MyParser implements IMiniCCParser{
    @Override
    public String run(String iFile) throws Exception {
        System.out.println("Parsing...");

        String oFile = MiniCCUtil.removeAllExt(iFile) + MiniCCCfg.MINICC_PARSER_OUTPUT_EXT;

        String temp = MiniCCUtil.removeAllExt(iFile) + ".c";
        ANTLRInputStream iStream = new ANTLRInputStream(new FileInputStream(temp));
        MyCGrammarLexer lexer = new MyCGrammarLexer(iStream);
        CommonTokenStream tokens = new CommonTokenStream(lexer);

        MyCGrammarParser parser = new MyCGrammarParser(tokens);
        ParseTree tree = parser.compilationUnit();

        TreeViewer treeViewer = new TreeViewer(Arrays.asList(parser.getRuleNames()), tree);
        treeViewer.open();

        ParseTreeWalker walker = new ParseTreeWalker();
        MyListener listener = new MyListener();
        listener.oFile = oFile;
        walker.walk(listener, tree);

        System.out.println("Finished!");

        return oFile;
    }
}
```

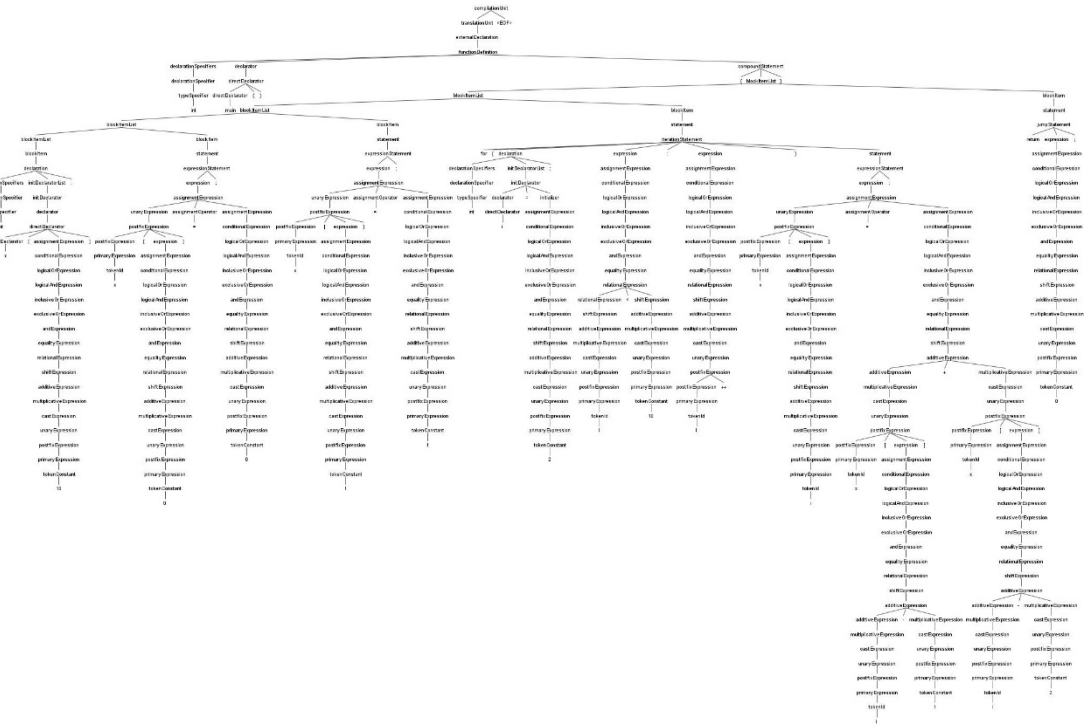
图 5：MyParser 源码

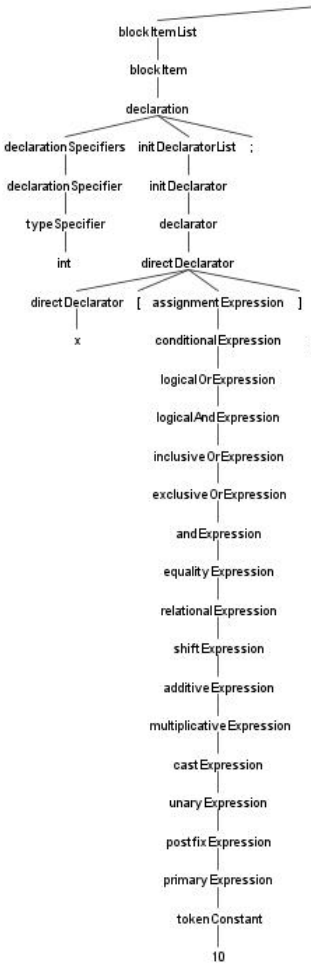
三、运行效果截图

测试用例

```c
int main() {
    int x[10];

    x[0] = 0;
    x[1] = 1;

    for (int i=2; i<10; i++)
        x[i] = x[i-1] + x[i-2];

    return 0;
}
```

具体语法树




blockItemList
blockItem
declaration
declarationSpecifiers
initDeclaratorList
;
declarationSpecifier
initDeclarator
typeSpecifier
declarator
int
directDeclarator
directDeclarator
[
assignmentExpression
]
x
conditionalExpression
logicalOrExpression
logicalAndExpression
inclusiveOrExpression
exclusiveOrExpression
andExpression
equalityExpression
relationalExpression
shiftExpression
additiveExpression
multiplicativeExpression
castExpression
unaryExpression
postfixExpression
primaryExpression
tokenConstant
10

部分. json 文件


```
{
  "type" : "Program",
  "items" : [ {
    "type" : "FunctionDefine",
    "specifiers" : [ {
      "type" : "Token",
      "value" : "int",
      "tokenId" : 0
    } ],
    "declarator" : {
      "type" : "FunctionDeclarator",
      "declarator" : {
        "type" : "VariableDeclarator",
        "identifier" : {
          "type" : "Identifier",
          "value" : "main",
          "tokenId" : 1
        }
      },
      "params" : [ ]
    },
    "body" : {
      "type" : "CompoundStatement",
      "blockItems" : [ {
        "type" : "Declaration",
        "specifiers" : [ {
          "type" : "Token",
          "value" : "int",
          "tokenId" : 5
        } ],
        "initLists" : [ {
          "type" : "InitList",
          "declarator" : {
            "type" : "ArrayDeclarator",
            "declarator" : {
              "type" : "VariableDeclarator",
              "identifier" : {
                "type" : "Identifier",
                "value" : "x",
                "tokenId" : 6
              }
            },
            "expr" : {
              "type" : "IntegerConstant",
              "value" : 10,
```

四、实验心得体会

在本次实验中，尽管使用了自动生成工具 ANTLR，但是由于语法规则扩充得过于复杂，导致 MyListener 的代码量变得极其庞大，耗费了大量时间。

在嵌入框架的过程中，由于未能找到将词法分析器生成的.tokens 文件还原为 CommonTokenStream 的方法，导致在 MyParser 中再次调用了词法分析器对源文件进行分析，产生了代码冗余。