语法分析实验

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

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。

本实验选用 ANTLR 工具，根据语法规则自动生成词法分析和语法分析器。ANTLR4 能够生成 ALL(*)分析器，从而能从源代码文件中直接生成解析树。使用 ANTLR 中提供的 Listener 接口，可以在遍历解析树的同时，使用 BIT-MiniCC 定义 AST 节点构造 AST，将解析树转换成 AST。

3. 实验过程及步骤

3.1 定义 ANTLR 支持的语法规则

ANTLR 使用 g4 格式的 EBNF 语法来描述文法规则。C 语言的语法规则，可以粗略的分为表达式、声明语句、语句块 3 类，使用 EBNF 描述的部分语法示例如下:

```
// 2 声明语句
declaration
    : declarationSpecifierList initDeclaratorList? ';'
    ;
// 2.1 声明语句说明符
declarationSpecifierList
    : (storageClassSpecifier | typeSpecifier | typeQualifier | functionSpecifier) declarationSpecifierList?
    ;
// 2.2 带初始化的声明列表
initDeclaratorList
    : initDeclarator
    | initDeclaratorList ',' initDeclarator
    ;
// 2.3 带初始化的声明语句
initDeclarator
    : declarator
    | declarator '=' initializer
    ;
```

3.2 根据文法规则生成词法、语法分析器

```
java -jar antlr-4.8-complete.jar C.g4
```

生成的分析器如下:

3.3 实现 Listener 接口，通过解析树构造 AST

Listener 模式是 ANTLR 提供的对解析树的一种遍历模式。使用 ParseTreeWalker 遍历树的过程中（深度优先），每次进入和退出规则节点时，触发对应的 enterRule/exitRule 方法。

定义映射表 ParseTreeProperty astMap。在遍历 ParseTree 退出规则节点时，以当前的 PareTree 为 key, 以对应的 ASTNode 为 value 构造键值对，插入到映射表 astMap 中。在构造 ASTNode 时，可以通过 astMap 查找它的子节点对应的 ASTNode 实现当前 ASTNode 的构造。及 astMap 中存储了 ParseTree 中的规则节点的“历史”信息。最终文法公理的规则节点 Start 对应的 ASTNode——ASTCompilationUnit 就是对应 AST 的根节点。部分代码示例如下：

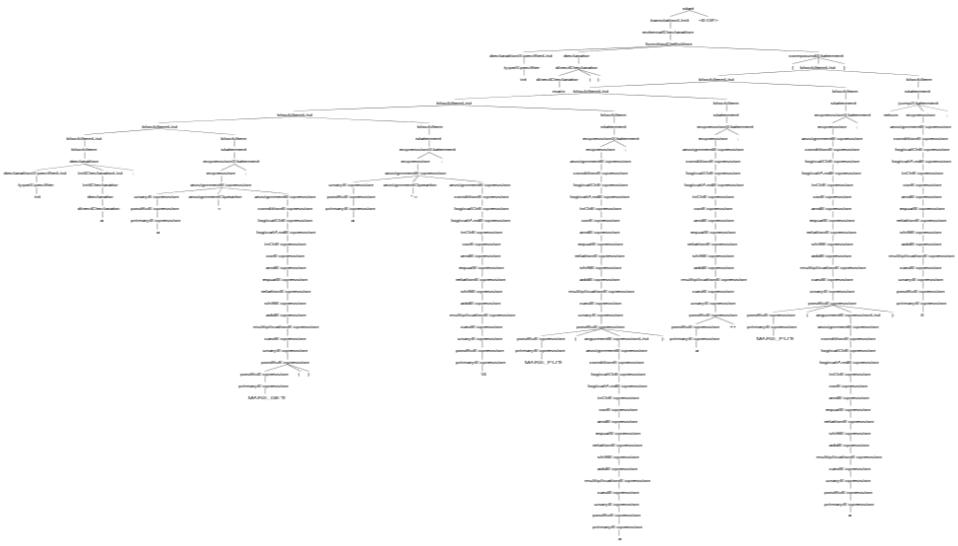
```java
public class MiniCCListener extends CBaseListener{
    private ParseTreeProperty astMap = new ParseTreeProperty();
    private ASTCompilationUnit compilationUnit;

    public ASTCompilationUnit getAST() {
        return this.compilationUnit;
    }

    @Override public void exitStart(CParser.StartContext ctx) {
        ASTCompilationUnit.Builder builder = new ASTCompilationUnit.Builder();
        if (ctx.translationUnit() != null) {
            builder.addNode(this.astMap.get(ctx.translationUnit()));
        }
        this.compilationUnit = builder.build();
        this.astMap.put(ctx, this.compilationUnit);
    }

    @Override public void exitTranslationUnit(CParser.TranslationUnitContext ctx) {
        List<ASTNode> ast = new LinkedList<>();
        if (ctx.translationUnit() != null) {
            ast.addAll((Collection<? extends ASTNode>) this.astMap.get(ctx.translationUnit()));
        }
        ast.add((ASTNode) this.astMap.get(ctx.externalDeclaration()));
        this.astMap.put(ctx, ast);
    }
```

3.4 实现 MiniCCParser

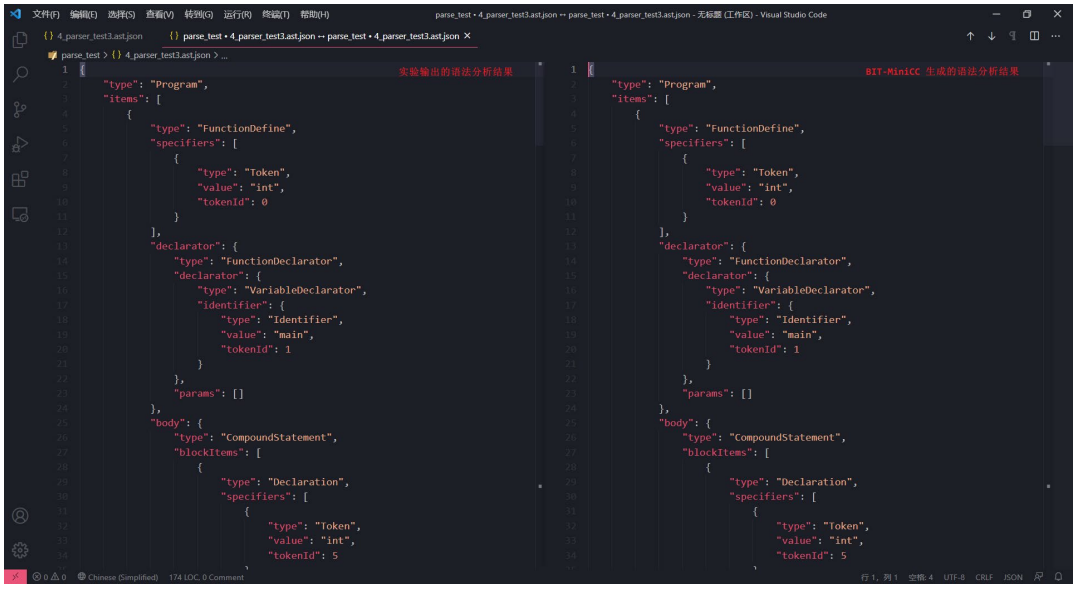
1) 使用 ANTLR 生成的 CLexer 进行词法分析，生成 CommonTokenStream。
2) 使用 ANTLR 生成的 CParser 进行语法分析，生成语法解析树 ParserTree。
3) 使用 ParserTreeWalker 对 ParserTree 进行遍历，遍历时使用构造的 Listener 将解析树转换为 AST。
4) 使用 ObjectMapper 将 AST 序列化为 json 文件。
5) 使用 ANTLR 的 gui 接口 TreeViewer 可视化解析树。

3. 实验结果

以 test\parse_test\4_parser_test3.c 为测试用例，运行结果如下：
（1）语法解析树可视化效果如下：



（2）实验生成的 ast.json 与 BIT-MiniCC 生成结果进行文件对比结果如下：



4. 实验心得体会

（1）进一步熟悉了 C 语言的语法。通过使用 EBNF 描述 C 语法规则，对 C 语言中的声明、表达式、语句块有了更清楚的认识。
（2）学习了 ANTLR 框架的使用，能够使用 ANTLR 通过语法规则生成相应规则的识别程序。
（3）学习了 ANTLR 中的 Listener 模式，体会到不需要显式的遍历也能实现树节点访问的便利。