编译原理与设计实验报告

姓名：卜梦煜　　学号：1120192419　　班级：07111905

1. 实验名称

语法分析实验

2. 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。

3. 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。

4. 实验环境

Intellij IDEA Community 2021.3.2 + ANTLR 4.8

5. 实验过程与步骤

本实验首先阅读和理解 BITMiniCC 中 AST 结点定义，根据 C11 标准中对 C 语言语法、AST 结点定义、四个测试用例，定义了一个 C 语言语法子集，写出相应语法产生式，并通过文法等价变换消除左递归；之后使用递归下降分析方法，基于 BITMiniCC 手动编码实现语法分析器，并将语法树输出为 JSON 文件；最后运行四个测试用例，验证语法分析器的正确性。

5.1 定义 C 语言语法子集

首先根据实验说明书中 AST 结点定义，结合 BITMiniCC 内部语法分析器运行测试用例输出的 JSON 文件，较为深入地理解了 BITMiniCC 中各 AST 结点含义、组成，以及 AST 结点之间继承、实现、聚合、组合关系。根据理解，将 BITMiniCC 中 AST 结点大致分为五大

类，理解如下：

BITMiniCC结点类型

没有值的属性，数组用[]表示，其他用null表示

- 表达式结点：
  - ASTExpression，表达式及基本类型入口
    - ASTUnaryExpression，一元运算表达式结点，包含op、expr
    - ASTUnaryTypename，sizeof(类型)运算，包含op、typename
    - ASTBinaryExpression，二元运算表达式，包含op、expr1、expr2
    - ASTFunctionCall，函数调用，包含funcname、argList
    - ASTArrayAccess，数组访问，包含arrayName、elements
    - ASTPostfixExpression，后缀表达式，包含expr、op
    - ASTCastExpression，显式类型转换表达式，包含typename、expr
    - ASTConditionExpression，条件运算表达式，包含condExpr、trueExpr、falseExpr
    - ASTMemberAccess，成员访问，包含master、op、member
- 语句结点：
  - ASTStatement，语句入口
    - ASTBreakStatement，break语句，无属性
    - ASTContinueStatement，continue语句，无属性
    - ASTGotoStatement，goto语句，无属性
    - ASTReturnStatement，return语句，无属性
    - ASTLabeledStatement，标志语句，包含label、stat，eg：k : break;
    - ASTCompoundStatement，复合语句，包含blockItems，eg：{ goto k; break; }
    - ASTExpressionStatement，表达式语句，包含exprs
    - ASTSelectionStatement，选择语句，包含cond、then、otherwise，仅if-else
    - ASTIterationStatement，无声明的循环语句，包含init、cond、step、stat，仅for
    - ASTIterationDeclaredStatement，有声明的循环语句，包含init、cond、step、stat，仅for
- 声明结点：
  - ASTDeclarator，包含type、declarator
    - ASTVariableDeclarator，变量声明，主要用于引用变量名，包含identifier，eg：a
    - ASTArrayDeclarator，数组声明，主要用于引用数组元素，包含declarator、qualifierList、expr，eg：a[2]
    - ASTFunctionDeclarator，函数声明，主要用于引用函数名，包含declarator、params，eg：func(double, int a)
    - ASTParamsDeclarator，参数声明，主要用于引用参数名，包含specfiers、declarator，eg：func(double, int a)中double、int a
  - ASTInitList，初始化列表，包含declarator、exprs
  - ASTDeclaration，声明语句，包含specifiers、initLists
- 叶结点：所有叶结点都需要包括type、value、tokenId
  - ASTIntegerConstant、ASTCharConstant、ASTFloatConstant、ASTStringLiteral，常量结点
  - ASTIdentifier，标识符结点
  - ASTToken，标记结点，主要为数据类型、运算符等关键词，如int、double、void、<、++
- 其它结点：
  - 函数定义：ASTFunctionDefine，包含specifiers、declarator、declarations、body
  - 语法分析入口：ASTCompilationUnit
  - 类型名：ASTTypename，只出现在ASTUnaryTypename、ASTCastExpression，包含specfiers、declarator

根据 C11 标准中对 C 语言语法、AST 结点定义，以四个测试用例中涉及的 C 语言语法要求为基础，修改实验四中定义的 C 语言语法子集，得到新的自定义的 C 语言语法子集如

下：

C语言简单子集：

- 数据类型：支持基本数据类型(int float double char string 数组 指针)，不支持复杂数据类型(enum union struct)
- 数据操作：支持单目运算(++ -- & * + - ! sizeof . ->)，支持双目运算：赋值运算(= *= /= %= += -=)、算术运算(+ - * / %)、逻辑运算(&& | | !)、关系运算(< > == <= >= !=)，支持三目运算(?:)，支持显示类型转换，支持自定义函数，支持运算符优先级(参考C11标准)，不支持位运算
- 程序结构：循环语句仅支持for循环、分支语句仅支持if-else循环
- 关键字：void int float double char string signed unsigned
- 符号集合：( ) [ ] { } ; 数据操作中定义的符号
- 部分不支持的内容：宏、头文件、注释

根据自定义的 C 语言语法子集，参考 C11 标准，得到该子集的语法产生式如下：

语法产生式

- program -> functionList
- functionList -> functionDefine functionList | e
- functionDefine -> typeSpecifiers declarator '(' arguments ')' compoundStatement
  - typeSpecifiers -> typeSpecifier typeSpecifiers | e
    - typeSpecifier -> 'void' | 'int' | 'float' | 'double' | 'char' | 'string' | 'signed' | 'unsigned'
  - arguments -> argumentList | e
    - argumentList -> arguement ',' argumentList | argument
      - argument ->typeSpecifiers declarator
  - compoundStatement -> '{' blockItemList '}'
    - blockItemList -> declaration blockItemList | statement blockItemList | e
      - declaration -> typeSpecifiers initDeclaratorList ';'
- initDeclaratorList -> initDeclarator | initDeclarator ',' initDeclaratorList
  - initDeclarator -> declarator | declarator '=' initializer
    - declarator -> identifier postDeclarator
      - postDeclarator -> '[' assignmentExpression ']' postDeclarator | '[' ']' postDeclarator | e
    - initializer -> assignmentExpression | '{' expression '}'
- statement -> breakStatement | continueStatement | gotoStatement | returnStatement | compoundStatement | selectionStatement | iterationStatement | iterationDeclaredStatement | expressionStatement
  - breakStatement -> 'break' ';'
  - continueStatement -> 'continue' ';'
  - gotoStatement -> 'goto' identifier ';'
  - returnStatement -> 'return' expression ';' | 'return' ';'
  - selectionStatement -> 'if' '(' expression ')' statement | 'if' '(' expression ')' statement 'else' statement
  - iterationStatement -> 'for' '(' expression ';' expression ';' expression ')' statement | 'for' '(' expression ';' expression ';' ')' statement
  - iterationStatement -> 'for' '(' declaration ';' expression ';' expression ')' statement | 'for' '(' declaration ';' expression ';' ')' statement
  - expressionStatement -> expression ';' | ';'

        - expression -> assignmentExpression ',' expressionStatement | assignmentExpression
- assignmentExpression -> conditionalExpression | unaryExpression assignmentOperator assignmentExpression
    - assignmentOperator -> '=' | '*=' | '/=' | '%=' | '+=' | '-='
    - conditionalExpression -> logicalOrExpression | logicalOrExpression '?' expression ':' conditionalExpression
    - logicalOrExpression -> logicalAndExpression | logicalAndExpression '||' logicalOrExpression
    - logicalAndExpression -> equalityExpression | equalityExpression '&&' logicalAndExpression
    - equalityExpression -> relationalExpression | relationalExpression '==' equalityExpression | relationalExpression '!=' equalityExpression
    - relationalExpression -> additiveExpression | additiveExpression '<' relationalExpression | additiveExpression '>' relationalExpression | additiveExpression '<=' relationalExpression | additiveExpression '>=' relationalExpression
    - additiveExpression -> multiplicativeExpression | multiplicativeExpression '+' additiveExpression | multiplicativeExpression '-' additiveExpression
    - multiplicativeExpression -> castExpression | castExpression '*' multiplicativeExpression | castExpression '/' multiplicativeExpression | castExpression '%' multiplicativeExpression
    - castExpression -> unaryExpression | '(' typeSpecifiers ')' castExpression
    - unaryExpression -> postfixExpression | unaryOperator castExpression
        - unaryOperator -> '++' | '--' | '&' | '*' | '+' | '-' | '!' | 'sizeof'
    - postfixExpression -> primaryExpression postfixExpressionPost
        - postfixExpressionPost -> '[' expression ']' postfixExpressionPost | '(' expression ')' postfixExpressionPost | '(' ')' postfixExpressionPost | '.' postfixExpressionPost | '->' postfixExpressionPost | '++' postfixExpressionPost | '--' postfixExpressionPost | e
    - primaryExpression -> identifier | integerConstant | floatingConstant | characterConstant | stringLiteral | '(' expression ')'

5.2 编码实现递归下降分析器

使用递归下降分析方法，基于 BITMiniCC 中的 ExampleParser.java，手动编码实现语法分析器，并将语法树输出为 JSON 文件。部分关键代码如下：

语法分析入口的定义：

```
// program -> functionList
public ASTNode program() {
    ASTCompilationUnit p = new ASTCompilationUnit();
    ArrayList<ASTNode> fl = functionList();
    if (fl != null) {
        //p.getSubNodes().add(fl);
        p.items.addAll(fl);
    }
    p.children.addAll(p.items);
    return p;
}
```

函数定义结点的定义：

```java
// functionDefine -> typeSpecifiers declarator '(' arguments ')' CompoundStatement
public ASTNode functionDefine() {
    ASTFunctionDefine fdef = new ASTFunctionDefine();

    ArrayList<ASTToken> specifiers = typeSpecifiers();
    fdef.specifiers = specifiers;
    fdef.children.addAll(specifiers);

    ASTFunctionDeclarator fdecl = new ASTFunctionDeclarator();
    ASTDeclarator decl = declarator();
    fdecl.declarator = decl;
    fdecl.children.add(decl);

    matchToken( type: "'('");

    ArrayList<ASTParamsDeclarator> pdl = arguments();
    if (pdl != null) {
        fdecl.params = pdl;
        fdecl.children.addAll(pdl);
    }
    fdef.declarator = fdecl;
    fdef.children.add(fdecl);

    matchToken( type: "')'");

    ASTCompoundStatement body = compoundStatement();
    fdef.body = body;
    fdef.children.add(body);

    return fdef;
}
```

语句结点的定义：

```java
// statement -> breakStatement | continueStatement | gotoStatement | returnStatement |
//              compoundStatement | selectionStatement | iterationStatement | expressionStatement
public ASTStatement statement() {
    nextToken = tknList.get(tokenIndex);
    return switch (nextToken.type) {
        case "'break'" -> breakStatement();
        case "'continue'" -> continueStatement();
        case "'goto'" -> gotoStatement();
        case "'return'" -> returnStatement();
        case "'{'" -> compoundStatement();
        case "'if'" -> selectionStatement();
        case "'for'" -> iterationStatement();
        default -> expressionStatement();
    };
}
```

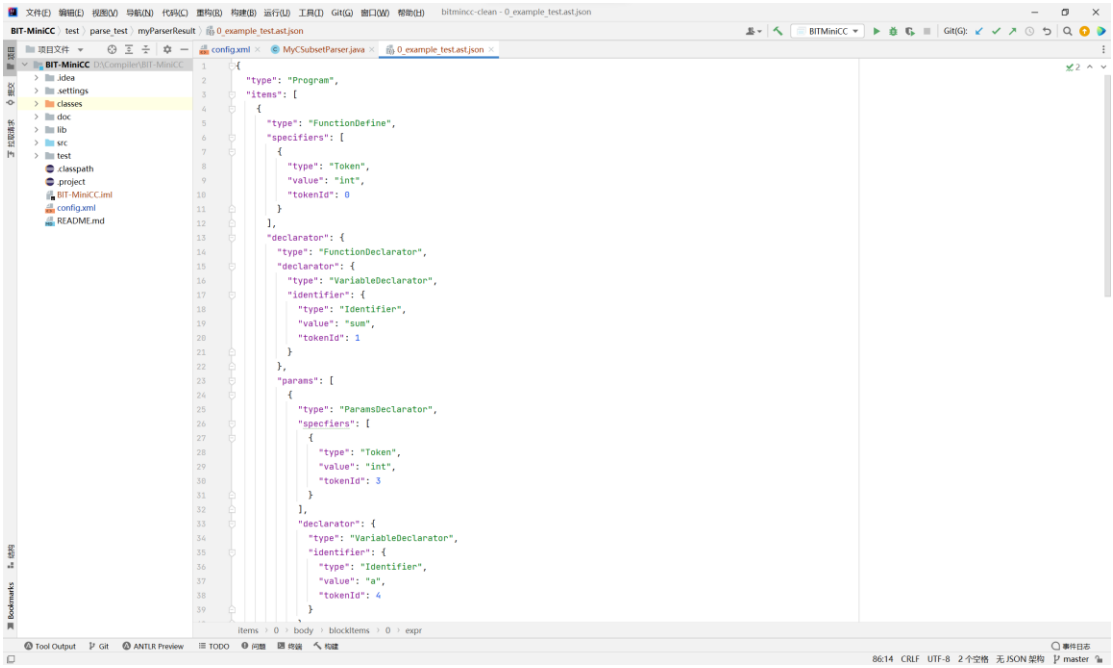
5.3 实验测试

用 BITMiniCC 中的四个语法分析器测试用例对自定义的语法分析器进行测试，四个测试

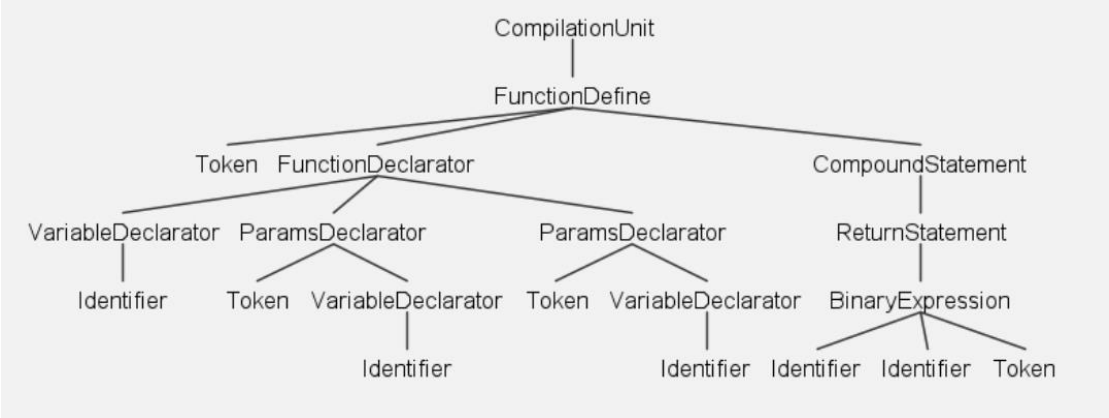
用例上测试结果分别如下：

（1）测试用例一：

生成 JSON 文件如下，与 BITMiniCC 内部语法分析器分析该测试用例生成的 JSON 文件内容相同：

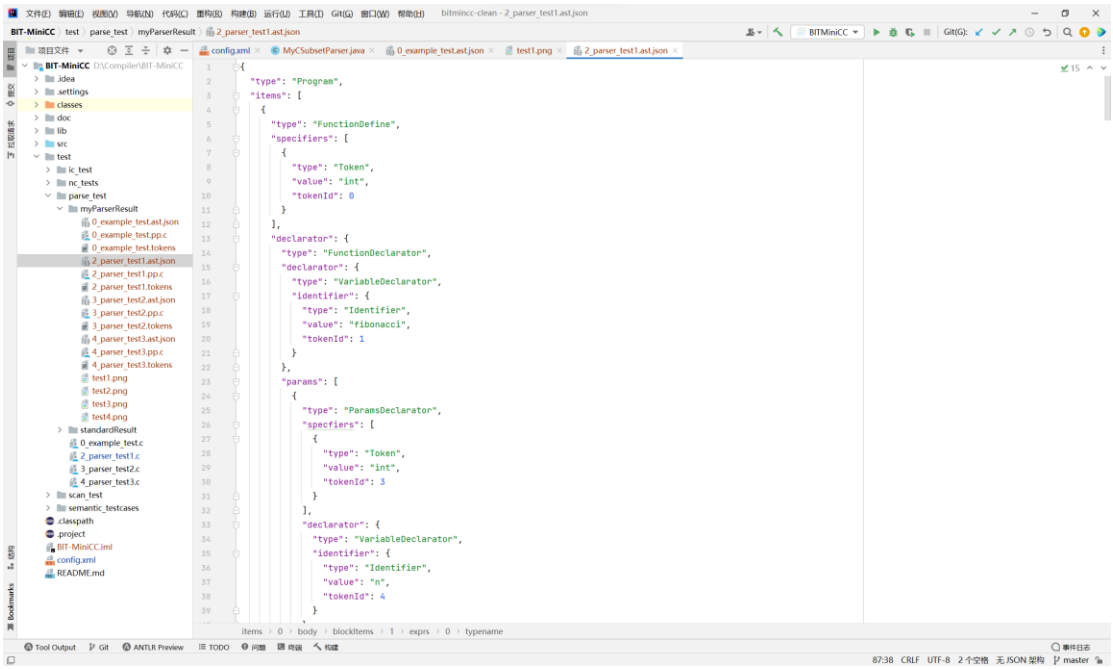


生成的语法树结构如下：

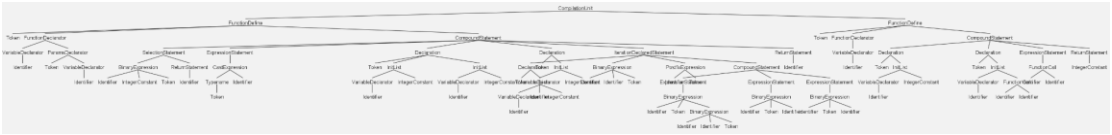


（2）测试用例二：

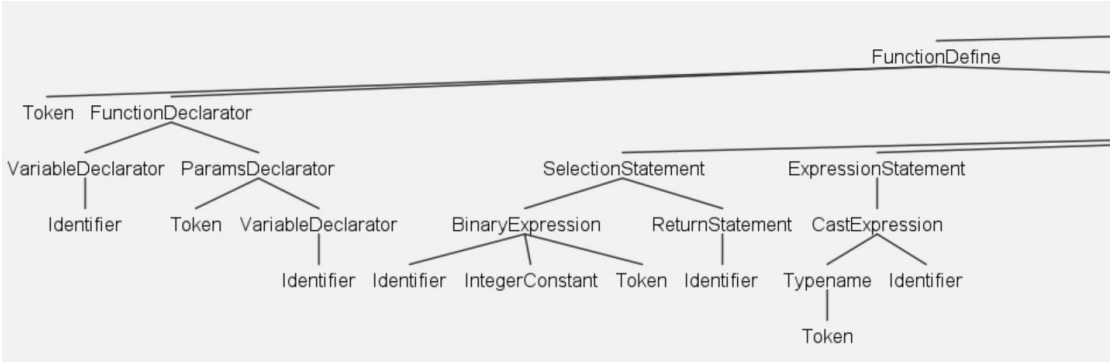
生成 JSON 文件如下，与 BITMiniCC 内部语法分析器分析该测试用例生成的 JSON 文件内容相同：



生成的语法树结构如下：

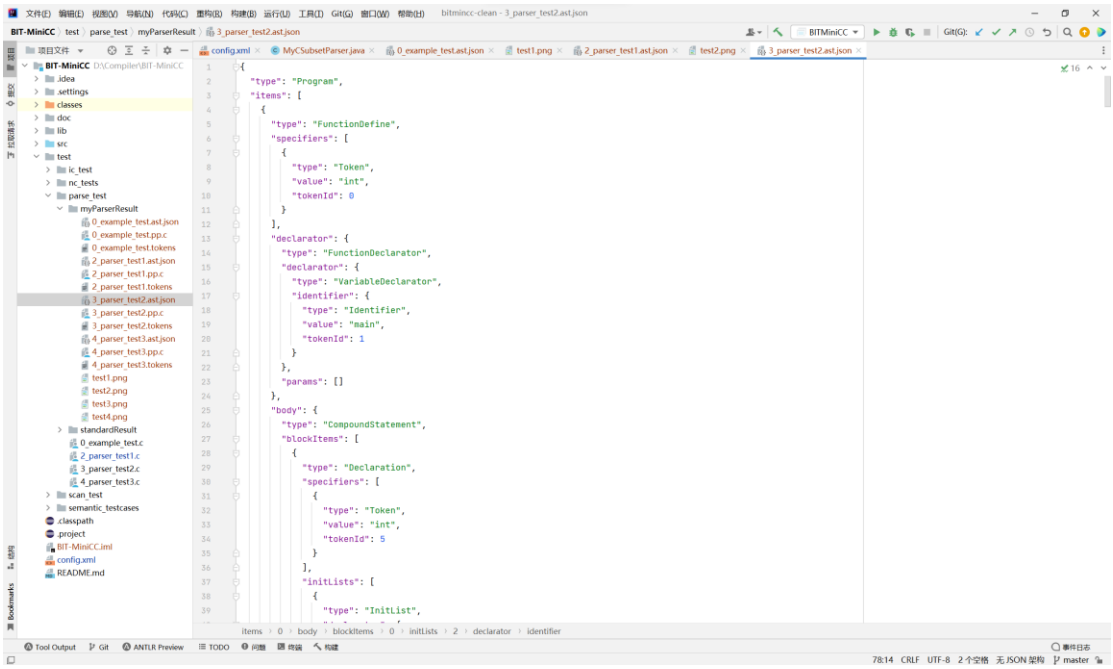


部分结构如下：

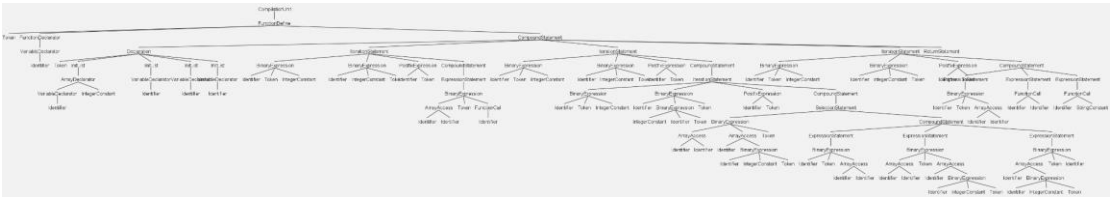


（3）测试用例三：

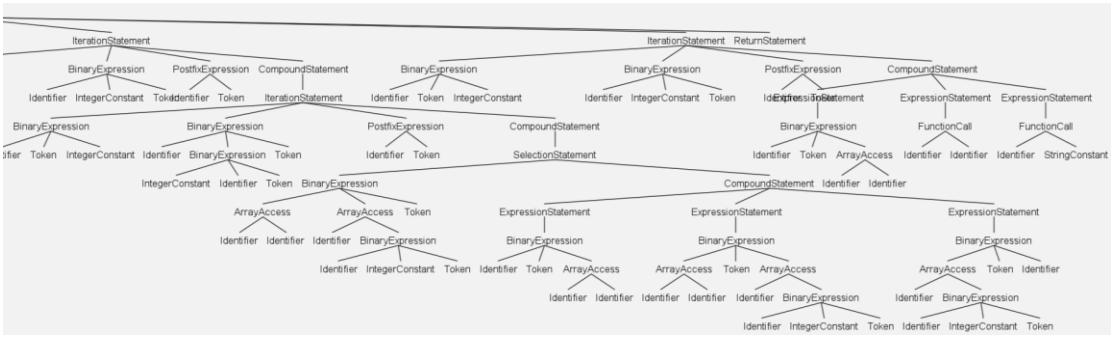
生成 JSON 文件如下，与 BITMiniCC 内部语法分析器分析该测试用例生成的 JSON 文件内容相同：



生成的语法树结构如下：

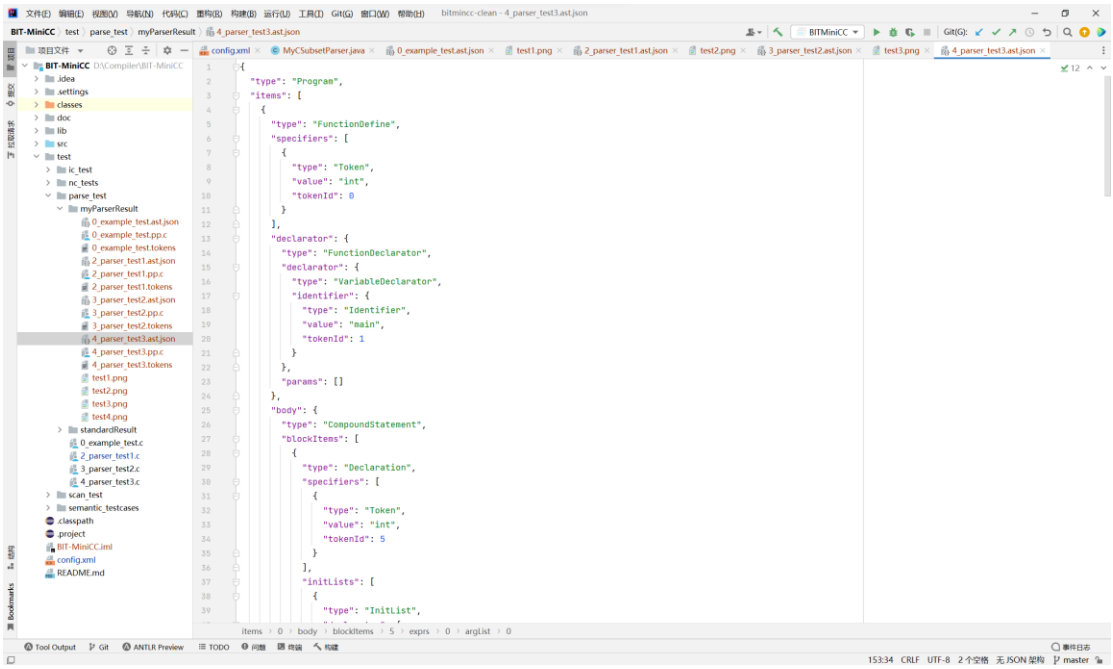


部分结构如下：

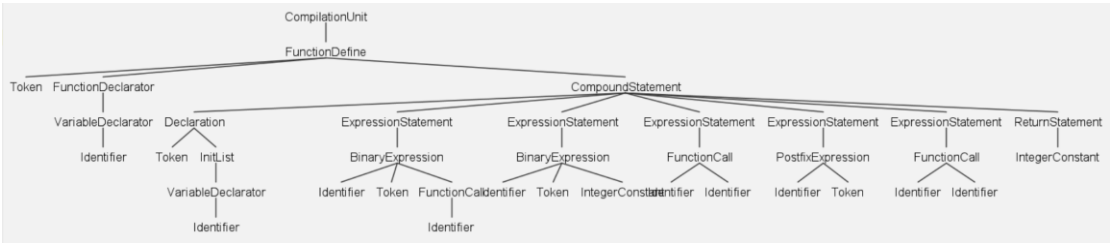


（4）测试用例四：

生成 JSON 文件如下，与 BITMiniCC 内部语法分析器分析该测试用例生成的 JSON 文件内容相同：



生成的语法树结构如下：



6. 实验心得体会

本次实验是目前遇到的难度最大的一次编译原理实验。实验中我首先采用了 ANTLR 自动生成工具，但在如何嵌入 BITMiniCC 的过程中遇到了一些困难，包括 token 文件读入方式，以及如何将 AST 结点写入语法分析器中，进而生成符合 BITMiniCC 定义的 JSON 文件。第一个问题可以通过手动编写符合结构的输入函数解决。第二个问题是主要问题，因为需要完全改写自动生成的语法分析器，并改写访问者模式的 visit()函数，工作量较大。因此换用了手动编写递归下降分析器的方式编写能够嵌入 BITMiniCC 的语法分析器。

通过本次实验，我有如下收获：

（1）在实验四自定义的 C 语言语法子集的基础上，进一步加深了对 C11 标准中定义的 C 语言语法结构的理解。

（2）对 BITMiniCC 中语法分析器的 AST 结点定义有了更深入的理解，基本理解了 BITMiniCC 中语法分析器的结构，包括语法分析器的输入输出、语法产生式与 AST 结点的关系、JSON 文件的理解、JAVA 访问者模式的理解等等。