编译原理 实验五 词法分析实验

1120180488 王梓丞

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。

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。

实验过程和步骤

文法构造

本次使用了 antlr 来构造文法规则，文法为上一次实验书写的 antlr 的改造，来自 C11 标准。终结符（词法分析）参考了 antlr 的官方 C 语言。

为了和目标的语法树有更好的对应关系，在原有的文法基础上加上了别名，以方便节点的遍历。

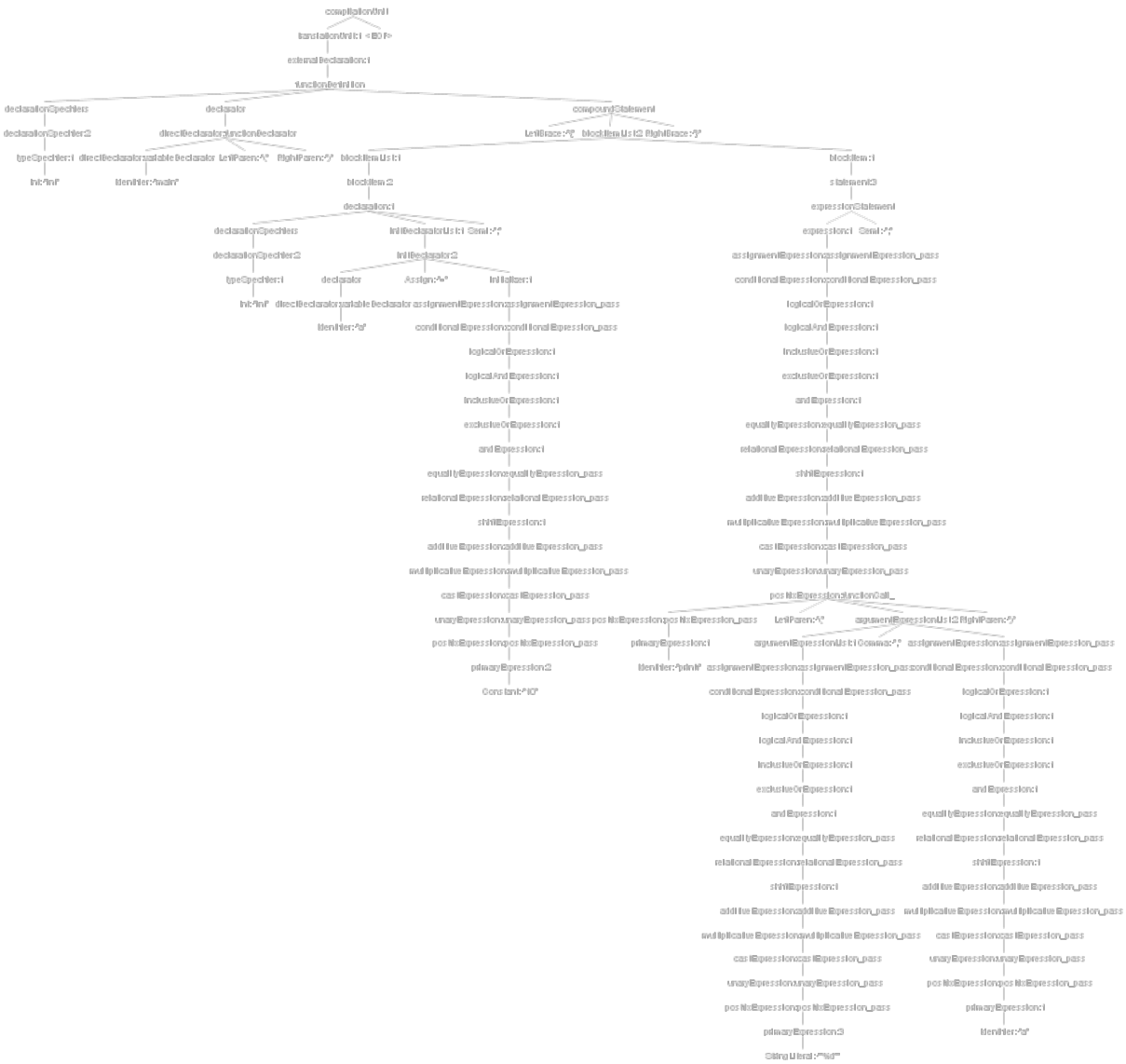
别名示例：

```
directDeclarator:
    Identifier                                                          # variableDeclarator
    | '(' declarator ')'                                                # variableDeclarator
    | directDeclarator '[' typeQualifierList? assignmentExpression? ']'         # arrayDeclarator
    | directDeclarator '[' 'static' typeQualifierList? assignmentExpression ']' # arrayDeclarator
    | directDeclarator '[' typeQualifierList 'static' assignmentExpression ']'  # arrayDeclarator
    | directDeclarator '[' typeQualifierList? '*' ']'                   # arrayDeclarator
    | directDeclarator '(' parameterTypeList ')'                        # functionDeclarator
    | directDeclarator '(' identifierList? ')'                          # functionDeclarator;
```

对于程序：

```
int main ( ) {
    int a = 10;
    printf("%d",a);
}
```

其生成的语法树示例：

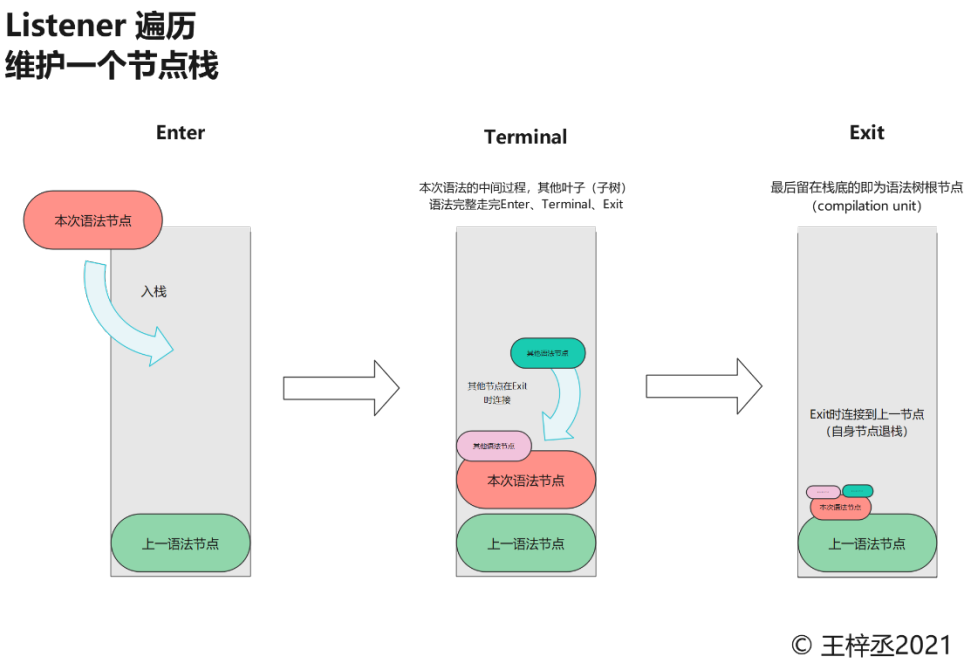


Parser 构造

语法结构构造完毕后，接下来就是通过这个语法规则构造规定的输出。在我看来这一个步骤相当于“遍历自己的语法树，在此过程中将已经定义 AST 类互相连接起来，生成需求的语法树”。

我选择了使用 antlr 提供的 listener 方法自动递归遍历语法树，通过对 CListener 程序的覆写来实现自定义构造规则的调用。

在具体实现上，我选择了使用一个栈来保存节点的历史数据，以方便节点的连接，栈顶的节点必定是链接好的节点，在遍历完成后，栈内剩余的一个函数一定是语法树根节点。



代码示例：

进入节点后向栈内推入此节点：

```java
@Override
public void enterEqualityExpression_(EqualityExpression_Context ctx) {
    nodeStack.push(new ASTBinaryExpression());
}

@Override
public void enterRelationalExpression_(RelationalExpression_Context ctx) {
    nodeStack.push(new ASTBinaryExpression());
}

@Override
public void enterAdditiveExpression_(AdditiveExpression_Context ctx) {
    nodeStack.push(new ASTBinaryExpression());
}

@Override
public void enterMultiplicativeExpression_(MultiplicativeExpression_Context ctx) {
    nodeStack.push(new ASTBinaryExpression());
}
```

退出时挂载本节点到父节点：

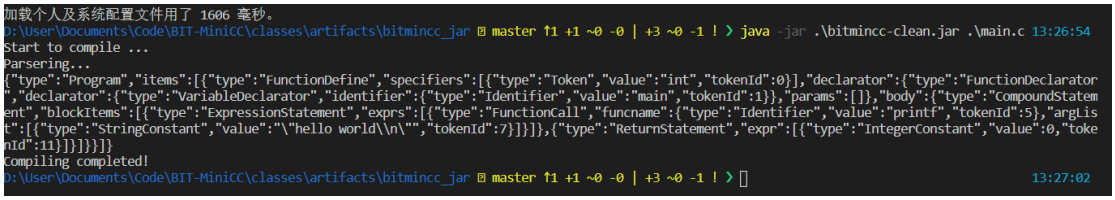
```java
public void mountNonExpression(ASTExpression thisNode, ASTNode parentNode) {
    // 把需要一个expression的非expression节点都放这里来
    if (parentNode.getClass() == ASTExpressionStatement.class) {
        if (((ASTExpressionStatement) parentNode).exprs == null) {
            ((ASTExpressionStatement) parentNode).exprs = new LinkedList<>();
        }
        ((ASTExpressionStatement) parentNode).exprs.add(thisNode);
    } else if (parentNode.getClass() == ASTInitList.class) {
        if (((ASTInitList) parentNode).exprs == null) {
            ((ASTInitList) parentNode).exprs = new LinkedList<>();
        }
        ((ASTInitList) parentNode).exprs.add(thisNode);
    } else if (parentNode.getClass() == ASTReturnStatement.class) {
        if (((ASTReturnStatement) parentNode).expr == null) {
            ((ASTReturnStatement) parentNode).expr = new LinkedList<>();
        }
        ((ASTReturnStatement) parentNode).expr.add(thisNode);
    } else if (parentNode.getClass() == ASTSelectionStatement.class) {
```

需要留意的是终结符的词法由于没有对应的 enter 和 exit 方法，需要在 visitTerminal 时将其挂载到可能的父节点上：

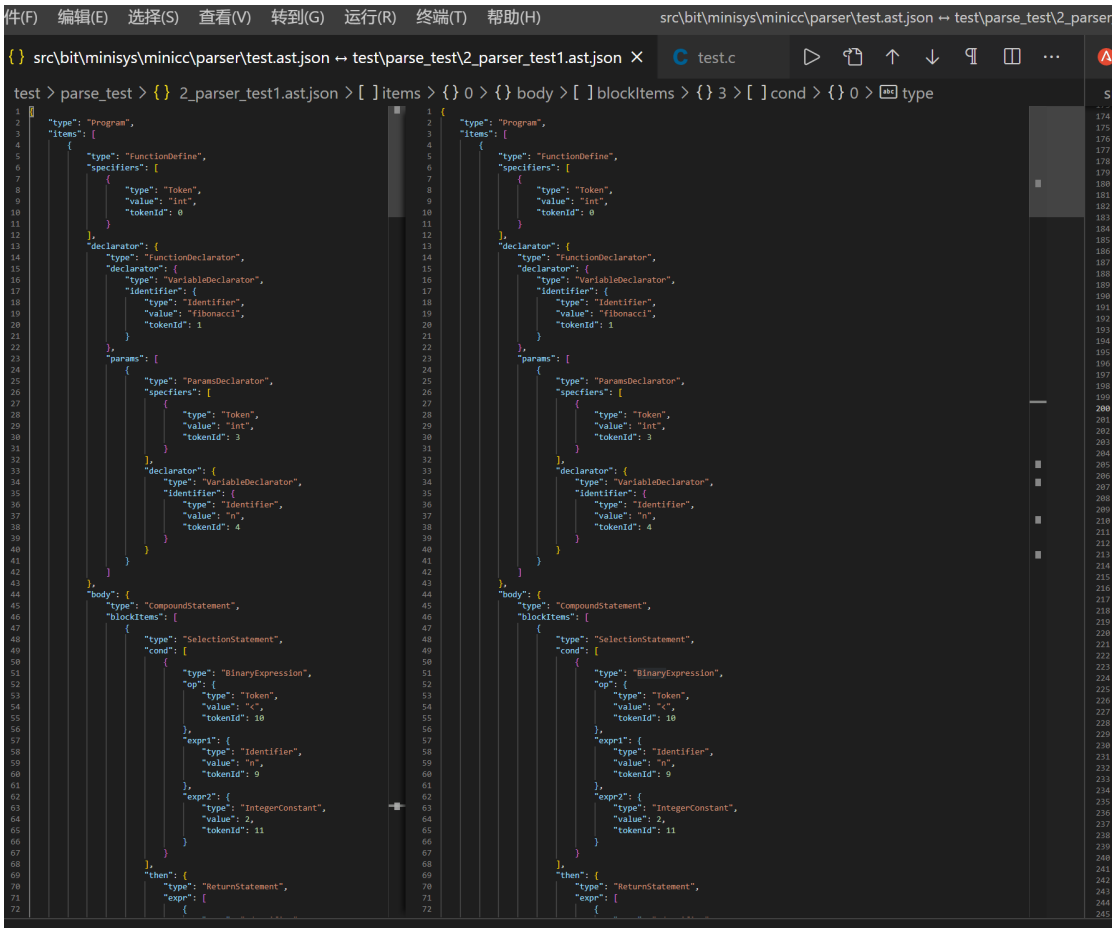
```java
@Override
public void visitTerminal(TerminalNode node) {
    parentNode = nodeStack.peek();
    switch (node.getSymbol().getType()) {
    case CLexer.Int:
    case CLexer.Double:
    case CLexer.Void:
    case CLexer.Float:
        if (parentNode.getClass() == ASTFunctionDefine.class) {
            ((ASTFunctionDefine) parentNode).specifiers
                    .add(new ASTToken(node.getSymbol().getText(), node.getSymbol().getTokenIndex()));
        } else if (parentNode.getClass() == ASTDeclaration.class) {
            ASTDeclaration astDeclaration = (ASTDeclaration) parentNode;
            if (astDeclaration.specifiers == null) {
                astDeclaration.specifiers = new LinkedList<ASTToken>();
            }
            astDeclaration.specifiers
                    .add(new ASTToken(node.getSymbol().getText(), node.getSymbol().getTokenIndex()));
        } else if (parentNode.getClass() == ASTParamsDeclarator.class) {
            ((ASTParamsDeclarator) parentNode).specfiers
                    .add(new ASTToken(node.getSymbol().getText(), node.getSymbol().getTokenIndex()));
        }
        break;
    case CLexer.Identifier:
```

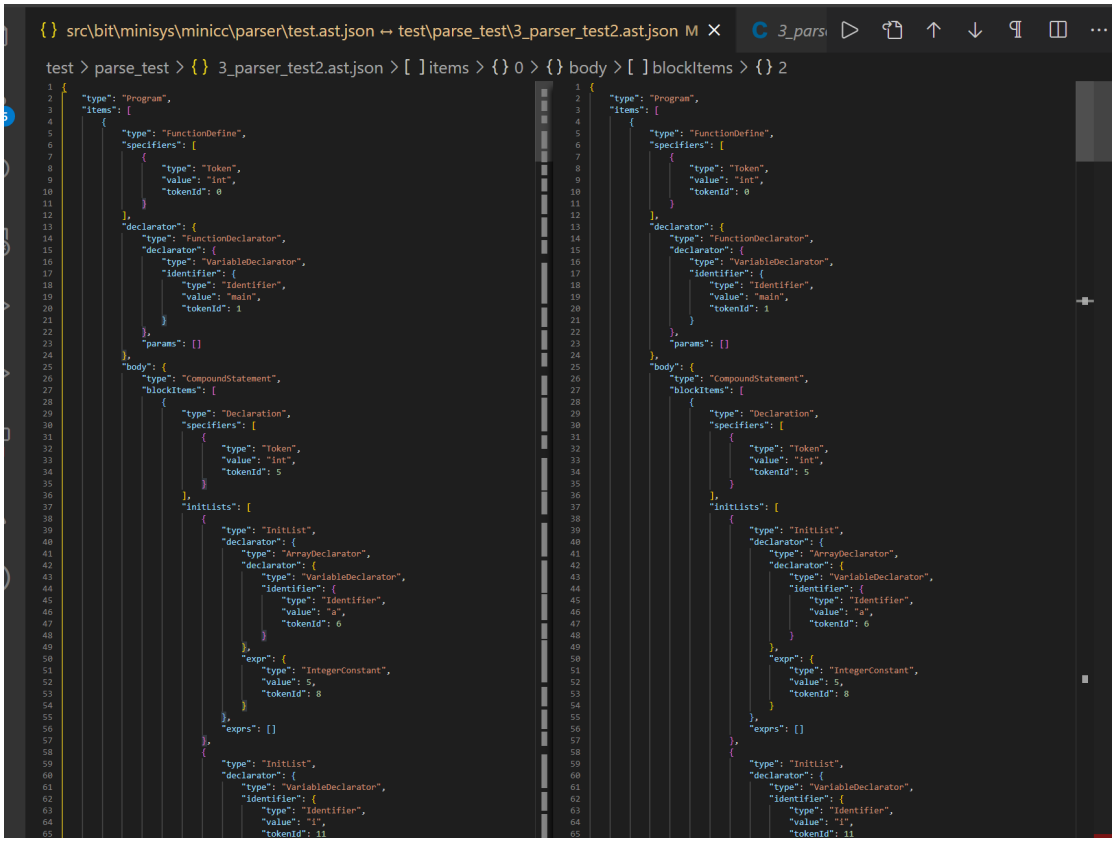
运行效果截图

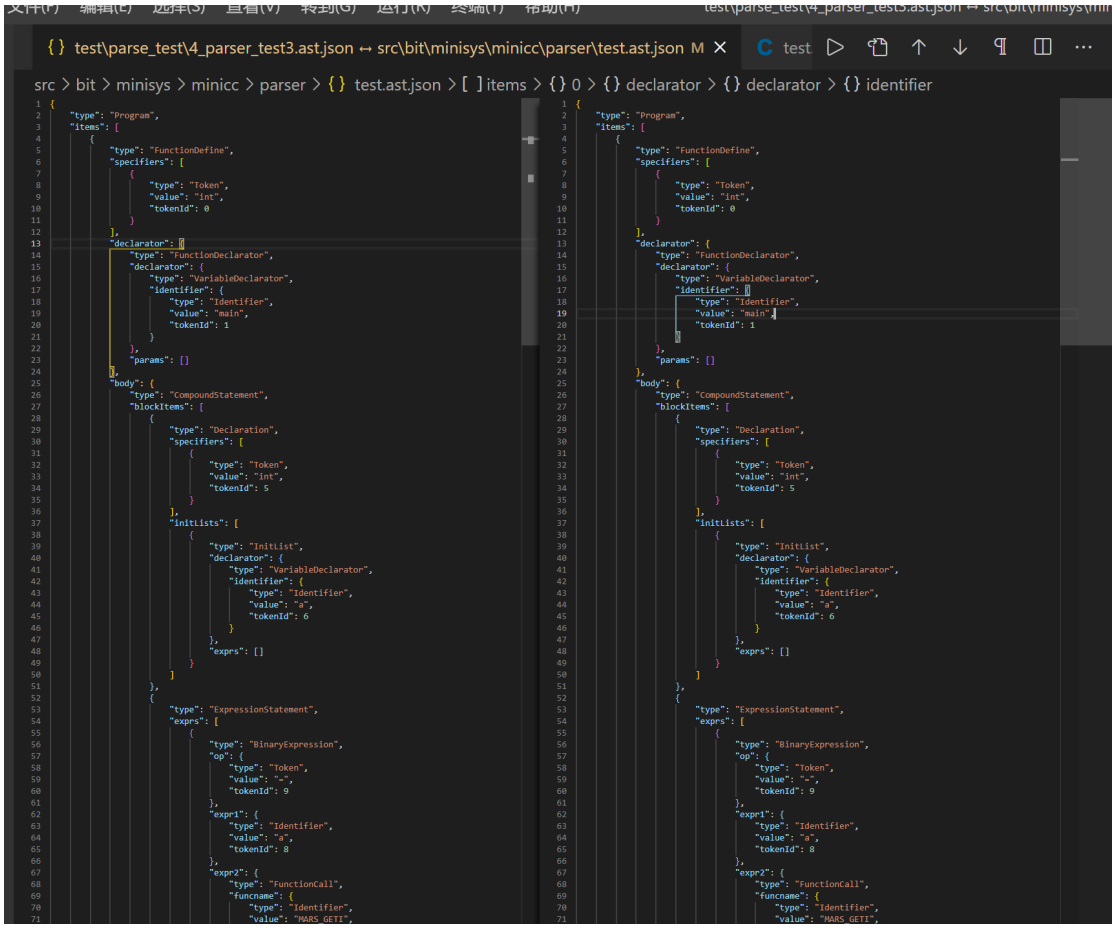
打包后的 Jar 包运行结果：



将其与 Parser_test 中的 C 语言文件与课程闭源的程序生成的 json 进行结果对比：




{ } src\bit\minisys\minicc\parser\test.ast.json ↔ test\parse_test\3_parser_test2.ast.json M
test > parse_test > { } 3_parser_test2.ast.json > [ ] items > { } 0 > { } body > [ ] blockItems > { } 2


{ } test\parse_test\4_parser_test3.ast.json ↔ src\bit\minisys\minicc\parser\test.ast.json M
src > bit > minisys > minicc > parser > { } test.ast.json > [ ] items > { } 0 > { } declarator > { } declarator > { } identifier

实验心得体会

通过本次实验，我学会了 antlr 的使用。在这次实验过程中，出现过一些比较令人困扰的情况，代码在实现语法过程中复杂度一度令人难以接受，因此在实现过程中我决定了重构一次代码。在重构代码后代码逻辑变得十分清晰，编写过程也十分顺利，运行结果也十分正确，让我意识到了写代码时良好的设计以及保持清醒的头脑的重要性。

同时，由于预定义的 AST 类有多重继承的关系，因此在这次实验中我深刻体验到了 Java 面向对象的类的多态性带来的程序的简洁性。