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

语义分析实验

2. 实验目的

（1）熟悉 C 语言的语义规则，了解编译器语义分析的主要功能；

（2）掌握语义分析模块构造的相关技术和方法，设计并实现具有一定分析功能的 C 语言语义分析模块；

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

3. 实验内容

语义分析阶段的工作为基于语法分析获得的分析树构建符号表，并进行语义检查。如果存在非法的结果，请将结果报告给用户，其中语义检查的内容主要包括：变量使用前是否进行了定义；变量是否存在重复定义；break 语句是否在循环语句中使用；函数调用的参数个数和类型是否匹配；函数使用前是否进行了定义或者声明；运算符两边的操作数的类型是否相容；数组访问是否越界；goto 的目标是否存在等。

本次语义检查的前（1）-（3）为要求完成内容，而其余为可选内容。

4. 实验环境

Intellij IDEA Community 2021.3.2

5. 实验过程与步骤

本实验首先封装符号表类、错误处理类。然后基于符号表类与错误处理类，封装抽象语法树的访问类，实现对每个 AST 结点接口的访问方法，并同时进行语义检查。最后编写语义分析总控程序，实现对抽象语法树的语义检查与错误报告。

5.1 封装符号表类与错误处理类

考虑到符号表中局部表、全局表等属性，以及表中变量插入等操作十分频繁，因此有必要对符号表相关的属性和方法进行封装。错误处理类同理。

对符号表分析，属性包括存储符号表项的 map 与当前符号表的父表，分别记录变量信息与父表信息；方法包括添加变量与查找变量是否定义。根据分析，对符号表定义如下。

```
class variable {
    public String name;
    public String type;

    public variable() {
        this.name = null;
        this.type = null;
    }

    public variable(String name, String type) {
        this.name = name;
        this.type = type;
    }
}

public class SymbolTable {
    public SymbolTable father;
    public Map<Object, Object> items = new LinkedHashMap<>();

    public void addVariable(String name, String type) {
        variable var = new variable(name, type);
        items.put(name, var);
    }

    public boolean findPresent(String name) { return this.items.containsKey(name); }

    public boolean findAll(String name) {
        if (this.items.containsKey(name)) {
            return true;
        } else if (this.father != null) {
            return this.father.findAll(name);
        } else {
            return false;
        }
    }
}
```

对错误处理分析，属性包括存储程序中出现的错误的列表；方法包括不同错误类型的错误添加方法，以及错误打印方法。根据分析，对错误处理类的封装如下：

```java
public class ErrorInfo {
    public ArrayList<String> errors = new ArrayList<~>();

    // 变量使用前是否进行了定义
    public void addES01Identifier(String name) {
        String error = "ES01 > Identifier [" + name + "] is not defined";
        this.errors.add(error);
    }

    public void addES01FunctionCall(String functionName) {
        String error = "ES01 > FunctionCall [" + functionName + "] is not defined";
        this.errors.add(error);
    }

    // 变量是否存在重复定义
    public void addES02Declaration(String name) {
        String error = "ES02 > Declaration [" + name + "] is defined";
        this.errors.add(error);
    }

    public void addES02FunctionDefine(String functionName) {
        String error = "ES02 > Function [" + functionName + "] is defined";
        this.errors.add(error);
    }

    // break语句是否在循环语句中使用
    public void addES03() {
        String error = "ES03 > breakstatement must be in a loopstatement";
        this.errors.add(error);
    }

    public void outputError() {
        if (errors != null) {
            System.out.println("Error:");
            for (String error : this.errors) {
                System.out.println(error);
            }
        }
    }
}
```

5.2 封装抽象语法树访问类

对抽象语法树，按深度优先顺序遍历，即对每个 AST 结点访问其属性。这个过程需要需要符号表类、错误分析类。深度优先访问示例如下。

```java
// functionList -> functionDefine functionList | e
@Override
public void visit(ASTCompilationUnit program) throws Exception {
    program.scope = this.globalSymbolTable;
    for (ASTNode astNode : program.items) {
        if (astNode instanceof ASTDeclaration) {
            this.visit((ASTDeclaration) astNode);
        } else if (astNode instanceof ASTFunctionDefine) {
            this.visit((ASTFunctionDefine) astNode);
        }
    }
}
```

对符号表，包括全局符号表、局部符号表和函数符号表。全局符号表记录最外层的全局变量信息，局部符号表记录当前作用域的局部变量信息，函数符号表记录全局函数的信息。当进入 ASTFunctionDefine、ASTIterationDeclaredStatement 或 ASTIterationStatement、ASTCompoundStatement 时发生作用域切换，需新建局部作用域，并设置 father 属性，记录作用域间嵌套关系。作用域切换示例如下。

```java
// compoundStatement -> '{' blockItemList '}'
// blockItemList -> declaration blockItemList | statement blockItemList | e
@Override
public void visit(ASTCompoundStatement compoundStat) throws Exception {
    // 复合语句形成新的作用域
    compoundStat.scope = this.localSymbolTable;
    this.localSymbolTable = new SymbolTable();
    this.localSymbolTable.father = compoundStat.scope;

    for (ASTNode astNode : compoundStat.blockItems) {
        if (astNode instanceof ASTDeclaration) {
            visit((ASTDeclaration) astNode);
        } else if (astNode instanceof ASTStatement) {
            visit((ASTStatement) astNode);
        }
    }

    // 还原作用域
    this.localSymbolTable = compoundStat.scope;
}
```

对错误信息，包括“变量未定义”、“变量重复定义”、“break 语句不在循环中”三种。“变量未定义”包括普通变量未定义，通过查询局部表与全局表判断，以及函数调用未定义，通过查询函数表判断。“变量重复定义”包括普通变量重定义，通过查询当前局部表判断，以及函数重定义，通过查询函数表判断。“break 语句不在循环中”可通过记录层数判断，进入 for 循

环时层数加 1，退出 for 循环时层数减 1，若层数为 0 时出现 break 语句则判断为“break 语句不在循环中”错误。“变量未定义”错误处理示例如下。

```
@Override
public void visit(ASTArrayAccess arrayAccess) throws Exception {
    String arrayName = "";
    ASTExpression compoundArrayName = arrayAccess.arrayName;
    ASTExpression index = arrayAccess.elements.get(0);

    while (true) {
        // a[i]中i未定义
        if (index instanceof ASTIdentifier && !localSymbolTable.findAll(((ASTIdentifier) index).value)) {
            errorInfo.addES01Identifier(((ASTIdentifier) index).value);
            return;
        }

        // a[i]中a未定义
        if (!localSymbolTable.findAll(arrayName)) {
            errorInfo.addES01Identifier(arrayName);
            return;
        }

        // a[i][j]...[k]
        if (compoundArrayName instanceof ASTArrayAccess) {
            index = ((ASTArrayAccess) compoundArrayName).elements.get(0);
            compoundArrayName = ((ASTArrayAccess) compoundArrayName).arrayName;
        } else {
            arrayName = ((ASTIdentifier) compoundArrayName).value;
            break;
        }

    }
}
```

“变量重复定义”错误示例如下。

```
// declaration -> typeSpecifiers initDeclaratorList ';'
// initDeclaratorList -> initDeclarator | initDeclarator ',' initDeclaratorList
// initDeclarator -> declarator | declarator '=' initializer
// declarator -> identifier postDeclarator
// postDeclarator -> '[' assignmentExpression ']' postDeclarator | '[' ']' postDeclarator | e
// initializer -> assignmentExpression | '{' expression '}'
@Override
public void visit(ASTDeclaration declaration) throws Exception {
    declaration.scope = this.localSymbolTable;

    String typeSpecifiers = "";
    for (ASTToken typeSpecifier : declaration.specifiers) {
        typeSpecifiers += " " + typeSpecifier.value;
    }

    for (ASTInitList initDeclarator : declaration.initLists) {
        this.visit(initDeclarator);

        String name = initDeclarator.declarator.getName();

        // declarator是否重定义
        if (this.localSymbolTable.findPresent(name)) {
            errorInfo.addES02Declaration(name);
            return;
        }

        if (declaration.scope == this.globalSymbolTable) {
            this.globalSymbolTable.addVariable(name, typeSpecifiers);
        } else {
            this.localSymbolTable.addVariable(name, typeSpecifiers);
        }
    }
}
```

“break 语句不在循环中”错误示例如下。

```
@Override
public void visit(ASTBreakStatement breakStat) throws Exception {
    if (iterationDepth <= 0) {
        errorInfo.addES03();
    }
}
```

5.3 语义分析总控程序

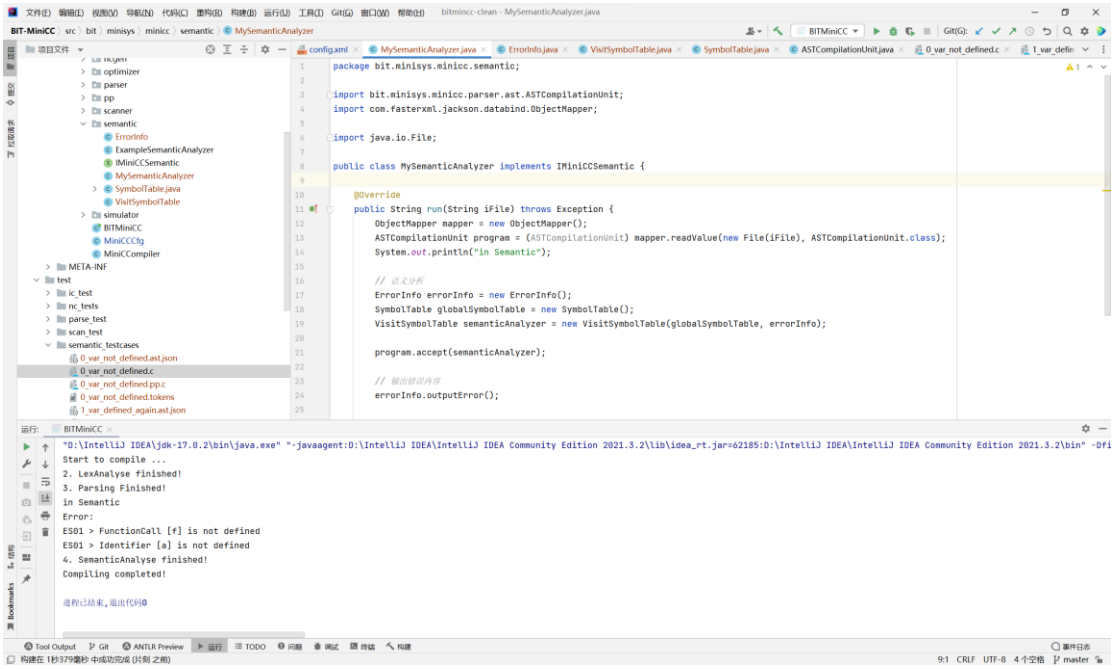
创建用于语法树分析的 VisitSymbolTable 实例，对语法树入口 ASTCompilationUnit，调用 accept()方法对语法分析树遍历，遍历结束后输出错误信息，完成语法分析。

```java
public class MySemanticAnalyzer implements IMiniCCSemantic {

    @Override
    public String run(String iFile) throws Exception {
        ObjectMapper mapper = new ObjectMapper();
        ASTCompilationUnit program = (ASTCompilationUnit) mapper.readValue(new File(iFile), ASTCompilationUnit.class);
        System.out.println("in Semantic");

        // 语义分析
        ErrorInfo errorInfo = new ErrorInfo();
        SymbolTable globalSymbolTable = new SymbolTable();
        VisitSymbolTable semanticAnalyzer = new VisitSymbolTable(globalSymbolTable, errorInfo);

        program.accept(semanticAnalyzer);

        // 输出错误内容
        errorInfo.outputError();

        System.out.println("4. SemanticAnalyse finished!");

        return null;
    }
}
```

6. 实验结果与分析

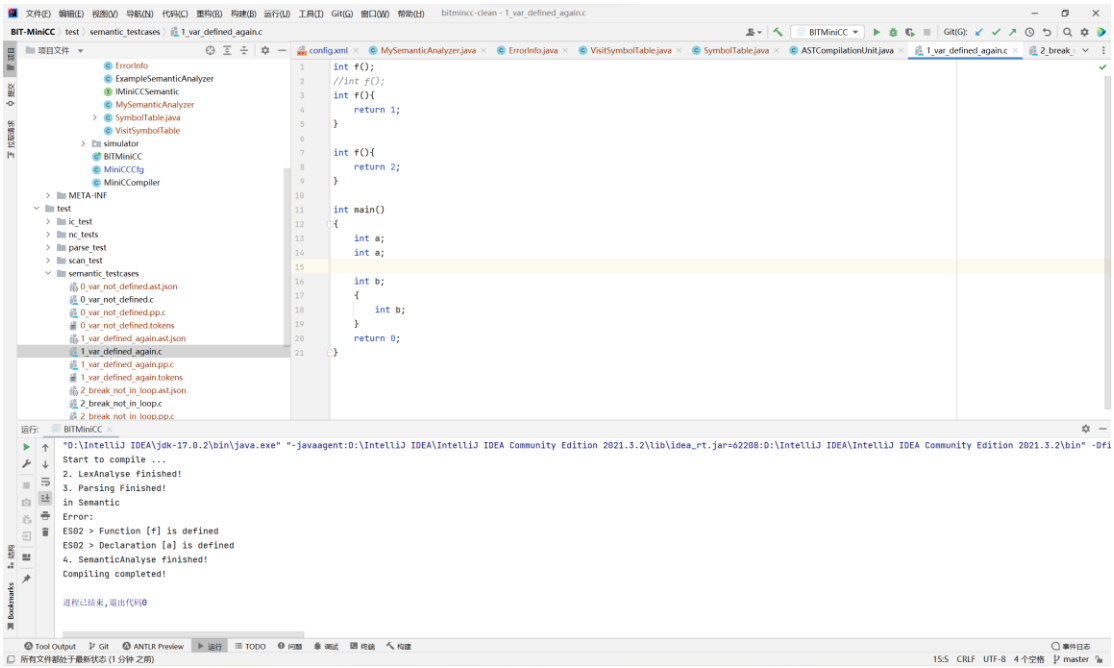
运行“test/semantic_testcases”中测试用例 0、1、2，观察运行结果。

运行“0_var_not_defined.c”，运行结果如下。



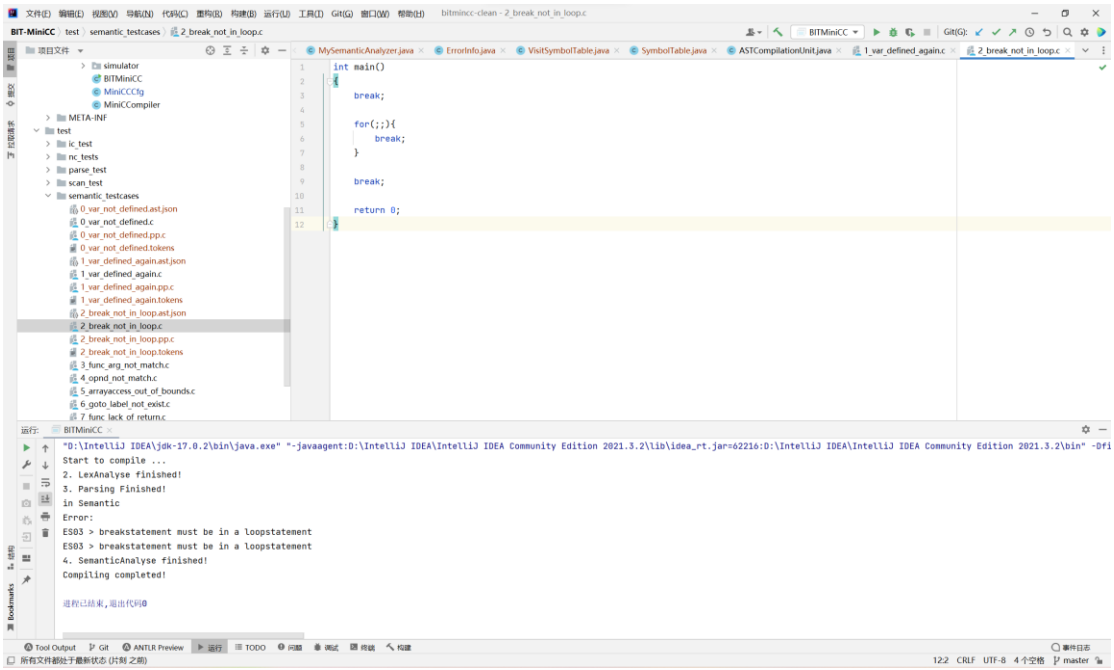
由运行结果可知，语义分析程序能够检测出测试用例中一遍扫描时存在函数未定义问题，以及变量未定义问题。将 f()函数定义移到 main()函数前，则不存在函数未定义问题，运行结果与此相同。

运行“1_var_defined_again.c”，运行结果如下。



由运行结果可知，语义分析程序能够检测出测试用例中 f()函数定义两次的问题，以及变量定义两次问题。

运行“2_break_not_in_loop.c”，运行结果如下。



由运行结果可知，语义分析程序能够检测出测试用例中两个 break 不在循环中的问题。

7. 个人心得体会

本次实验代码难度不大，难度主要在于语义分析程序的设计与写法上。虽然课上已经学过语义分析相关的符号表、作用域等内容，但由于对 JAVA 语法的不熟悉，以及对 BITMiniCC 中需要对语义分析程序编写的框架不清楚，导致刚开始无从下手。在与同学交流、查阅相关

资料之后，对 BITMiniCC 中语义分析部分框架与 JAVA 中访问者模式有了一定熟悉，后面的代码编写难度并不是很大。通过本次实验，我的收获如下。

（1）对 C 语言中常见的语义错误及其产生原因、语义分析规则、编译器语义分析的主要功能有了一些了解。

（2）初步掌握了构造语义分析模块的相关技术与方法，能够设计具有一定错误分析能力的 C 语言语义分析器。