编译原理 Lab6: 语义分析实验

郑子帆 1120200822

北京理工大学 计算机学院 07112002 班

日期：2023 年 5 月 19 日

摘　　要

本文为北京理工大学《编译原理与设计 2023》课程的 Lab6 实验报告。在本文中，我们在前面实验所得到的 AST 语法树的基础上，构建了符号表，并进行了三种类型的简单的语义检查。最后，分别在三个测试程序上进行了实验。

1　实验简介 [1]

1.1　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。

1.2　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；

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。在本次实验中，我仅实现了（1）-（3）三个基本功能。

2 实验过程

本次实验是在语法分析实验的基础上，通过遍历 AST 语法树来构建符号表并基于二者进行语义检查。并能够输出相应的错误编号和错误信息。遍历 AST 树是通过重写 ASTVisitor 的 visit 方法。

2.1 构建符号表

在这里我们分为了全局符号表和局部符号表。其中局部符号表可以划分为多层，每层以一对大括号 {} 作为区分。下面为设计的符号表类。

```
class Symbol{
    public String name;   // 该标识符对应函数还是变量
    public String type;   // 该标识符的数据类型
    public List<String> params = new ArrayList<>();   // 若该标识符为函数，存
}
```

其中，name 有三种取值，分别是 Variable, FunctionDeclaration 和 FunctionDefine，用来区分标识符对应的是变量，函数声明还是函数定义。type 用于记录该标识符的数据类型。params 用来存储函数的参数数据类型列表。

2.2 遍历 AST 节点

获取到 AST 语法树根结点 program 后，首先编写 MyVistor 类实现 ASTVisitor 接口函数，然后通过 Vistor 对象对 program 的子节点进行深度优先遍历。下面代码为对于根节点的遍历：

```
@Override
public void visit(ASTCompilationUnit program) throws Exception {
    for(int i = 0; i < program.items.size(); i++ ){
        ASTNode node = program.items.get(i);
        if(node instanceof ASTDeclaration) {
            this.visit((ASTDeclaration)node);
        }else if(node instanceof  ASTFunctionDefine) {
            this.visit((ASTFunctionDefine)node);
```

```
        }
    }
    outputEnd();
}
```

2.3 功能实现

在本实验中我们只实现了 ES01、ES02、ES03 的相关报错功能。

2.3.1 变量是否先被定义 (ES01)

每个标识符在使用前必须进行定义操作，所以需要在每个标识符被使用前对符号表进行查找。如果该变量没有出现在任何的符号表中，则进行相关报错。另外，还要主要不同类型的标识符是否被混用，如一个标识符被定义为变量，但是却以函数的形式调用等等。核心代码如下：

```
@Override
public void visit(ASTIdentifier idToken) throws Exception {
    String idName = idToken.value;
    Symbol idSymbol = null;
    boolean found = false;
    for(int index = scopeIdx-1; index >= 0; index--){
        Map<String, Symbol> localSymbols = localSymbolTable.get(index);
        if( localSymbols.get(idName) != null ){
            idSymbol = localSymbols.get(idName);
            found = true;
        }
    }
    if( !found ){
        if(globalSymbolTable.get(idName) == null ){
            display("Identifier: " + displayName(idName) + " is not defin
            exprKind = "int";
            return;
        }else{
            idSymbol = globalSymbolTable.get(idName);
        }
    }
    if(!idSymbol.name.equals("Variable")){
```

```
            display("Identifier: " + displayName(idName) + " is defined as fu
        }
        exprKind = idSymbol.type;
    }
```

2.3.2 变量是否被重复定义 (ES02)

需要检查变量或函数是否存在重复定义，同样需要进行对符号表的查询。对于函数，只需要在函数表中查找是否存在对应项即可；对于变量，需要对当前局部符号表进行查找。因为例如全局变量名与某一函数的变量名即使相同也并不是重复定义。核心代码实现如下：

```
private boolean checkSymbolAdd(String symbolKey, Symbol symbol, Map<Strin
    if( symbolTable.get(symbolKey) == null ){
        symbolTable.put(symbolKey, symbol);
        return true;
    }
    output(symbol.name + ": " + outputName(symbolKey) + " is redefined.",
    return false;
}

public void visit(ASTFunctionDeclarator functionDeclarator, String type)
    String functionName = ((ASTVariableDeclarator) functionDeclarator.dec
    if(scopeId != 0){
        output("FunctionDeclaration: " + outputName(functionName) + "  sc
        return;
    }
    Symbol symbol = new Symbol();
    symbol.type = type;
    symbol.name = "FunctionDeclaration";
    symbol.params = getParams(functionDeclarator.params);
    checkSymbolAdd(functionName, symbol, symbolTableGlobal);
}

public void visit(ASTVariableDeclarator variableDeclarator, String type)
        Symbol symbol = new Symbol();
        symbol.type = type;
```

```
        symbol.name = "Variable";
        String variableName = variableDeclarator.identifier.value;
        if(scopeId == 0){
            checkSymbolAdd(variableName, symbol, symbolTableGlobal);
        }else{
            checkSymbolAdd(variableName, symbol, symbolTableLocal.get(sco
        }
    }
```

2.3.3 break 是否在循环中 (ES03)

由于 break 语句的父节点不一定是循环语句，有可能是其祖先节点是循环语句。故我们用一个栈来保存当前所有在执行中的语句块的种类。当语句块开始时，将该语句块的类型入栈；当语句块结束时，栈顶元素出栈。而当 break 语句出现时，我们只需要遍历当前栈中所有种类，判断是否含有循环语句即可。核心代码如下：

```
public void visit(ASTBreakStatement breakStat) throws Exception {
    for(int i = stateList.size()-1; i >= 0; i-- ){
        if(stateList.get(i).equals("Iteration")) return;
    }
    output("BreakStatement: \"break\" must be in a LoopStatement.", 3);
}

public void visit(ASTIterationDeclaredStatement iterationDeclaredStat) th
    Map<String, Symbol> symbolTable = new HashMap<>();
    symbolTableLocal.add(symbolTable);
    scopeId++;
    this.visit(iterationDeclaredStat.init);
    if(iterationDeclaredStat.cond != null) {
        for (ASTExpression cond : iterationDeclaredStat.cond) {
            this.visit(cond);
        }
    }
    if(iterationDeclaredStat.step != null) {
        for (ASTExpression step : iterationDeclaredStat.step) {
            this.visit(step);
        }
    }
```

```
    stateList.add("Iteration");
    this.visit(iterationDeclaredStat.stat);
    stateList.remove(stateList.size()-1);
    scopeId--;
    symbolTableLocal.remove(scopeId);
}
```

3 实验结果

3.1 配置 *config.xml*

配置 *config.xml*，具体内容如下。

```
<?xml version="1.0" encoding="UTF-8"?>
<config name="config.xml">
  <phases>
    <phase>
      <phase skip="true" type="java" path="" name="preprocess" />
      <phase skip="false" type="java" path="" name="scan" />
      <phase skip="false" type="java" path="" name="parse" />
      <phase skip="false" type="java" path="bit.minisys.
      minicc.parser.MySemantic.java" name="semantic" />
      ...
    </phase>
  </phases>
</config>
```

3.2 ES01 检查展示

对 0_var_not_defined.c 文件进行语义分析，结果如下。

```
-----------------------------------------------------
ES1 >> FunctionCall: "f" is not defined.
ES1 >> Variable: "a" is not defined.
-----------------------------------------------------
Semantic Finished!
Compiling completed!
```

3.3 ES02 检查展示

对 1_var_defined_again.c 文件进行语义分析，结果如下。

```
-----------------------------------------------------
ES2 >> FunctionDefine: "f" is redefined.
ES2 >> Variable: "a" is redefined.
-----------------------------------------------------
Semantic Finished!
Compiling completed!
```

3.4 ES03 检查展示

对 2_break_not_in_loop.c 文件进行语义分析，结果如下。

```
-----------------------------------------------------
ES3 >> BreakStatement: "break" must be in a LoopStatement.
ES3 >> BreakStatement: "break" must be in a LoopStatement.
-----------------------------------------------------
Semantic Finished!
Compiling completed!
```

4 实验心得与体会

通过这次实验，使我对语义分析的流程和方法有了更清晰的认识。同时，对编译器各个模块的工作原理有了更具体的了解，这也让我更深入地理解了课本中的知识。

在实验中，我也遇到了一些问题，比如回溯节点比较麻烦，于是采取了栈存储的方式进行了遍历，让我对各种数据结构和 java 代码编写也有了进一步的学习与体会。

参考文献

[1] *Lab 6 语义分析说明及要求.pdf*. zh. 2023.