编译原理第二次实验报告

姓名：李博文　　学号：2022111189

一、实验内容

本次实验在已完成的词法分析（lexical.l）和语法分析（syntax.y）的基础上，进一步实现了语义分析模块。主要目标是：

1. 对给定的 C-- 源代码进行语义检查；

2. 检测并准确报告包括变量重复定义、类型不匹配、函数未定义等在内的 17 类语义错误；

3. 输出每个语义错误对应的行号与错误类型；

4. 构建并维护符号表以支持作用域管理、函数/结构体识别；

5. 利用抽象语法树（AST）辅助实现语义规则遍历与分析。

二、实验环境与编译方式

- 开发环境：macOS + VSCode + GNU Flex & Bison

- 编译命令：在 Code/ 目录下执行 make 即可完成编译，生成可执行文件 parser。

三、模块结构说明

1. 词法分析器 (lexical.l)

- 使用 Flex 实现；

- 定义了关键词、标识符、常量等正则规则；

- 输出 token 传递给 Bison 构建语法树。

2. 语法分析器 (syntax.y)

- 使用 Bison 实现；

- 定义了完整的 C-- 语法产生式；

- 在每个语义规则中调用 buildTree() 构建抽象语法树。

3. **抽象语法树模块** (tree.c, tree.h)

- 每个非终结符都构造成 TreeNode；
- 支持多子节点构建与递归打印；
- 提供结构打印函数 printTree() 用于调试分析。

4. **语义分析模块** (semantic.c, semantic.h)

- 维护符号表栈结构，支持作用域嵌套；
- 对 AST 进行 DFS 遍历，识别函数/结构体/变量声明与使用；
- 精准定位 17 种语义错误类型并打印格式统一的报错。

5. **主程序入口** (main.c)

- 启动词法、语法分析；
- 构建语法树后调用语义分析器进行语义校验；
- 最终打印所有语义错误并输出。

四、测试示例

使用提供的 test.cmm 输入样例测试，语义错误检测如下所示（因为十七个样例太多故省略输出部分）：

```
$ ./parser test.cmm
Error type 3 at Line 5: Redefined variable "x".
Error type 5 at Line 8: Type mismatched for assignment.
...
```

五、实验亮点与总结

- 使用栈式符号表支持作用域嵌套，适应函数/结构体复杂场景；

- 错误去重机制避免一处多报；

- 模块解耦性强，便于调试与维护；

- 实现了全套 17 种错误类型，全部通过 OJ 验证。

六、实验体会

本次语义分析实验是编译器从“语法正确”迈向“语义合理”的关键步骤。通过实现符号表管理与语义规则校验，切实感受到编译器在语义层面所需考虑的复杂性。这一实验提升了我对作用域、类型系统、符号绑定等概念的理解，也为后续中间代码生成与优化打下了坚实基础。