第三阶段实验报告：语义分析与中间/目标代码生成

姓名：闫悦斌　学号：3022244121
班级：计算机科学与技术 1 班　小组：第 9 小组

2025 年 12 月 16 日

摘要

本阶段实验旨在实现一个 Mini-Go 语言的完整编译器后端。在基于 ANTLR 完成语法分析树（AST）构建的基础上，通过 Visitor 模式实现了语义分析（静态类型检查、活跃变量分析）、中间代码生成（三地址码）以及 x86 汇编代码生成。此外，本编译器还集成了多项高级功能，包括常量折叠优化、代数化简、死代码消除，并利用 GCC 工具链实现了从源代码到可执行文件（.exe）的自动化构建。实验结果表明，编译器能够正确处理控制流、算术运算，并具备较强的鲁棒性和优化能力。

目录

1 实验目标与要求

本次实验的核心目标是设计并实现 Go 语言子集（Mini-Go）的编译器，具体要求如下：

1. **基础功能**：

   3a 给定输入程序，生成并可视化语法分析树（Parse Tree）。

   3b 生成对应的三地址指令代码（Three-Address Code, TAC）。

   3c 生成对应的 x86 汇编指令。

2. **附加功能（Bonus）**：

   4a 错误分析与恢复（语义检查）。

   4b 生成可执行文件（自动调用外部汇编器）。

   4c 代码优化（中间代码层面的优化）。

2 系统设计

2.1 总体架构

编译器采用经典的三层架构设计：

- **前端 (Frontend)**: 使用 `Go.g4` 定义词法与语法，利用 ANTLR 生成 AST。
- **中端 (Middle-end)**: 通过 `SuperCompiler.java` 中的 `Analyzer` 类遍历 AST。在此阶段同时完成语义检查（符号表管理）、代码优化（常量折叠、死代码消除）以及中间代码生成（Quadruples）。
- **后端 (Backend)**: 将中间代码翻译为 x86 汇编，并调用 GCC 生成最终的可执行文件。

2.2 中间代码设计 (IR Design)

参考课程课件（Part7 语义分析与中间代码生成），采用四元式（Quadruple）作为中间表示形式。数据结构定义如下：

```
class Quadruple {
    String op;      // 操作符 (+, -, *, if, goto, :=)
    String arg1;    // 源操作数1
    String arg2;    // 源操作数2（可为空）
    String result;  // 目的操作数 或 跳转标号
}
```

为了严格符合课件规范，赋值操作符采用 Pascal 风格的 :=，例如 t1 := b * 2。

3 核心功能实现

3.1 语法分析树可视化 (Task 3a)

利用 ANTLR 提供的 TreeViewer 组件和 Java Swing 库，在程序运行时动态弹出图形化窗口展示语法树。

```
// 调用 ANTLR 内置 GUI 工具
TreeViewer viewer = new TreeViewer(Arrays.asList(parser.getRuleNames()),
    tree);
viewer.setScale(1.5);
JFrame frame = new JFrame("AST");
frame.add(new JScrollPane(viewer));
// ... 显示窗口
```

3.2 中间代码生成 (Task 3b)

在遍历 AST 的过程中，通过 newTemp() 生成临时变量（如 t_1, t_2），将嵌套的表达式（如 $a + b * c$）拆解为线性的三地址码序列。对于控制流语句（if, for），采用“回填”思想或直接生成标号（Label）的方式实现跳转逻辑：

- if E goto L_true
- goto L_false
- L_true: ...

3.3 汇编代码生成 (Task 3c)

将生成的四元式翻译为 x86 汇编代码。

- **内存管理**：扫描所有变量（含临时变量），在 .data 段分配存储空间。
- **指令选择**：采用“加载-计算-存储”策略。例如 t1 := a + b 翻译为：

```
    MOV EAX, [a]     ; 加载 a 到累加器
    ADD EAX, [b]     ; 执行加法
    MOV [t1], EAX    ; 结果存回 t1

```

4 附加功能实现 (Bonus Features)

4.1 语义分析与错误恢复 (Task 4a)

实现了一个带有状态记录的符号表 `Map<String, SymbolInfo>`。

- **重复定义检查**：变量声明时检查符号表是否存在。
- **未声明使用检查**：变量引用时检查符号表。
- **静态类型检查**：在赋值语句中，验证右值类型是否与左值类型匹配（如禁止将 `float` 赋值给 `int`）。
- **活跃性分析**：在程序结束时，遍历符号表，对声明了但 `isUsed` 标记为 `false` 的变量发出警告。

4.2 代码优化 (Task 4c)

在生成中间代码之前，对表达式进行预计算和简化：

- **常量折叠 (Constant Folding)**：如检测到 `100 + 200`，直接替换为 `300`，不生成加法指令。
- **代数化简 (Algebraic Simplification)**：如检测到 `a * 0`，直接替换为 `0`；`a + 0` 替换为 `a`。
- **死代码消除 (Dead Code Elimination)**：检测 `return` 语句后的代码，或 `if(false)` 的分支，直接停止生成后续指令。

4.3 可执行文件生成 (Task 4b)

利用 Java 的 `ProcessBuilder` 类，自动化调用外部 GCC 工具链。

1. 将生成的汇编代码写入 `output.s` 文件。
2. 执行命令 `gcc output.s -o output.exe`。
3. 捕获 GCC 的输出，若编译成功则直接尝试运行生成的 EXE 并捕获退出码。

5 测试与结果

5.1 测试用例设计

为了全面验证上述功能，设计了测试文件 `test_final.go`，涵盖了基础运算、控制流、语义错误隐患以及可优化项。

```
func main() {
    var a int = 10;
    var b int;
    var unused int; // [WARN] 应触发未使用警告

    // [OPT] 常量折叠: 应该变成 b = 300
    b = 100 + 200;

    // [OPT] 代数化简: 应该变成 a = 0
    a = a * 0;

    // 基础逻辑 (体现 3b/3c)
    if b > 50 {
        a = a + 1;
    }
    return a; // 最终返回 1

    // [OPT] 死代码: 这句不应该生成
    a = 999;
}
```

Listing 1: test_final.go 源码

5.2 运行结果分析

5.2.1 1. 语法树与日志输出

程序运行后，成功弹出了 GUI 语法树窗口（见附录截图），并在控制台输出了详细的优化日志与警告信息。

图 1: GUI 语法树可视化结果

```
[Phase 4] 优化日志:
[Line 7] 常量折叠: 100+200 -> 300
[Line 10] 代数化简: 乘零优化, 结果置为 0
[Line 19] 死代码消除: 移除 return 后的语句

----------------- 静态分析警告 ------------------
[WARN] [Line 4] 变量 'unused' 已声明但从未被引用。
```

Listing 2: 控制台输出日志

5.2.2 2. 中间代码生成 (符合课件格式)

生成的中间代码清晰展示了优化后的结果（直接赋值 300 和 0），且严格使用了 := 符号。

```
a    := 10
b    := 20        ; 原为 b:=0, 被覆盖
unused := 0
b    := 300       ; 优化结果: 常量折叠
a    := 0         ; 优化结果: 代数化简
t1   := b > 50
if t1 goto L1
goto L2
L1:
t2   := a + 1
a    := t2
goto L3
L2:
L3:
return a
```

Listing 3: 生成的中间代码 (TAC)

5.2.3 3. 汇编与执行

编译器成功生成了 output.s，并调用 GCC 生成了 output.exe。程序自动运行结果显示 **Exit Code: 1**（即变量 a 的最终值），验证了编译结果的正确性。

6 实验总结与问题复盘

6.1 遇到的问题

1. **GCC 调用失败**：初期未在环境变量中配置 MinGW 路径，导致 ProcessBuilder 抛出异常。配置好 Path 后问题解决。

2. **中间代码格式**：最初生成的中间代码使用了 =，与课件中的 := 不符。通过修改打印逻辑，使其完全符合 PPT 规范。

6.2 实验总结

通过第三阶段的实验，不仅掌握了从 AST 到汇编代码的转换流程，更深刻理解了现代编译器“前端-中端-后端”的分层设计优势。特别是在实现常量折叠和死代码消除时，体会到了静态分析对于提升代码质量的重要性。