□ 实验报告：基于 Go 的中间代码生成器设计与实现

一、实验目的

本实验旨在设计并实现一个小型语言的中间代码生成系统，支持以简洁 Go 模块为基础的源代码解析、语义分析与三地址码（TAC）生成，为后续编译优化与代码生成打下基础。

二、实现功能概述

本系统支持如下语言特性对应的中间代码生成：

功能类别	支持情况	示例
变量赋值 `d = E`	□	`x = a + b * c - d;`
return 语句	□	`return x + 1;`
函数调用 `d(Ř)`	□	`foo(a + 1, b);`
数组赋值 `a[Ě] = E`	□	`a[i+1,j*2,4] = 66;`
条件语句 if	□	`if (x < y) a = 1;`
if-else	□	`if (x < y) a = 1; else a = 0;`
while 循环	□	`while (x < 10) y = y + 1;`
布尔表达式 &&, ||, !	□	`if ((x < 1 y < 2) && !z)`
块语句 `{ S1; S2; }`	□	`if (...) { a = 1; b = 2; }`

三、项目模块结构与职责划分

```
project/
├── main.go             // 主函数：读取 JSON 测试用例并驱动生成
├── test_case.json      // 含测试目的 + 代码的输入数据集
├── lexer/lexer.go      // 词法分析：将代码字符串切分为 token 序列
├── parser/parser.go    // 顶层分发：转发给 stmt 模块
├── stmt/               // 各类语句模块定义
│   ├── dispatch.go         // 统一分发入口 Dispatch([]string)
│   ├── if.go               // if / if-else 的解析与跳转生成
│   ├── while.go            // while 结构的入口与循环体生成
│   ├── return.go           // return 的表达式求值与生成
│   ├── call.go             // 函数调用：参数倒序传参 + CALL
│   ├── array_assign.go     // 数组赋值表达式及偏移计算
│   ├── stmtlist.go         // 块结构 `{ ... }` 多语句处理器
│   └── util.go             // 通用辅助函数，如括号匹配等
├── boolean/expr.go     // 逻辑表达式短路生成器：支持 &&, ||, !, ()
├── expr/expr.go        // 表达式求值器：支持算术表达式与临时变量生成
├── generator/tac.go    // 初期测试用静态样例输出（现已弃用）
```

四、核心模块实现要点

1. `lexer.go`：支持多字符符号识别

- 自动识别 `!=`, `<=`, `>=`, `==` 为单 token，避免分裂成 `!` 与 `=`。

2. `expr.go`：表达式左结合求值

- 利用栈式遍历生成临时变量 `t1`, `t2`, ...
- 支持如 `a + b * c - d` 的算术表达式组合。

3. `boolean/expr.go`：支持短路逻辑的布尔表达式生成

- 实现了如下语义：
  - `B → B1 || B2`：左真跳真，左假跳右
  - `B → B1 && B2`：左假跳假，左真跳右
  - `B → !B`：真假跳转互换
  - `B → (B)`：递归展开

4. `stmt/stmtlist.go`：支持复合语句 `{ ... }`

- 维护语法嵌套层级 `level`，仅在分号层级为 0 时切分子句
- 自动识别内部语句是否为控制结构或嵌套语句块

5. `dispatch.go`：统一语句调度器

- 代替原 `parser.go` 中分发逻辑
- 调用 `GenerateIfElse`、`GenerateWhile`、`GenerateReturn` 等模块
- 解决模块循环 import 的设计冲突

五、main.go 测试入口

- 支持读取 `test_case.json` 文件格式如下：

```
[
  {
    "purpose": "if-else 结构测试",
    "code": "if (x < y) a = 1; else a = 0;"
  },
  ...
]
```

- 每个用例输出结构清晰：
  - □ 测试目的

- □ 输入语句
- □ 三地址代码

六、模块关系图

```
main.go
  ↓
  lexer.Tokenize()
  ↓
  parser.ParseAndGenerateTAC()
  ↓
  stmt.Dispatch()
    ├── if.go / while.go / return.go ...
    ├── stmtlist.go（用于 {} 递归块）
    └── expr / boolean 中调用 GenerateXXX()
```

运行截图

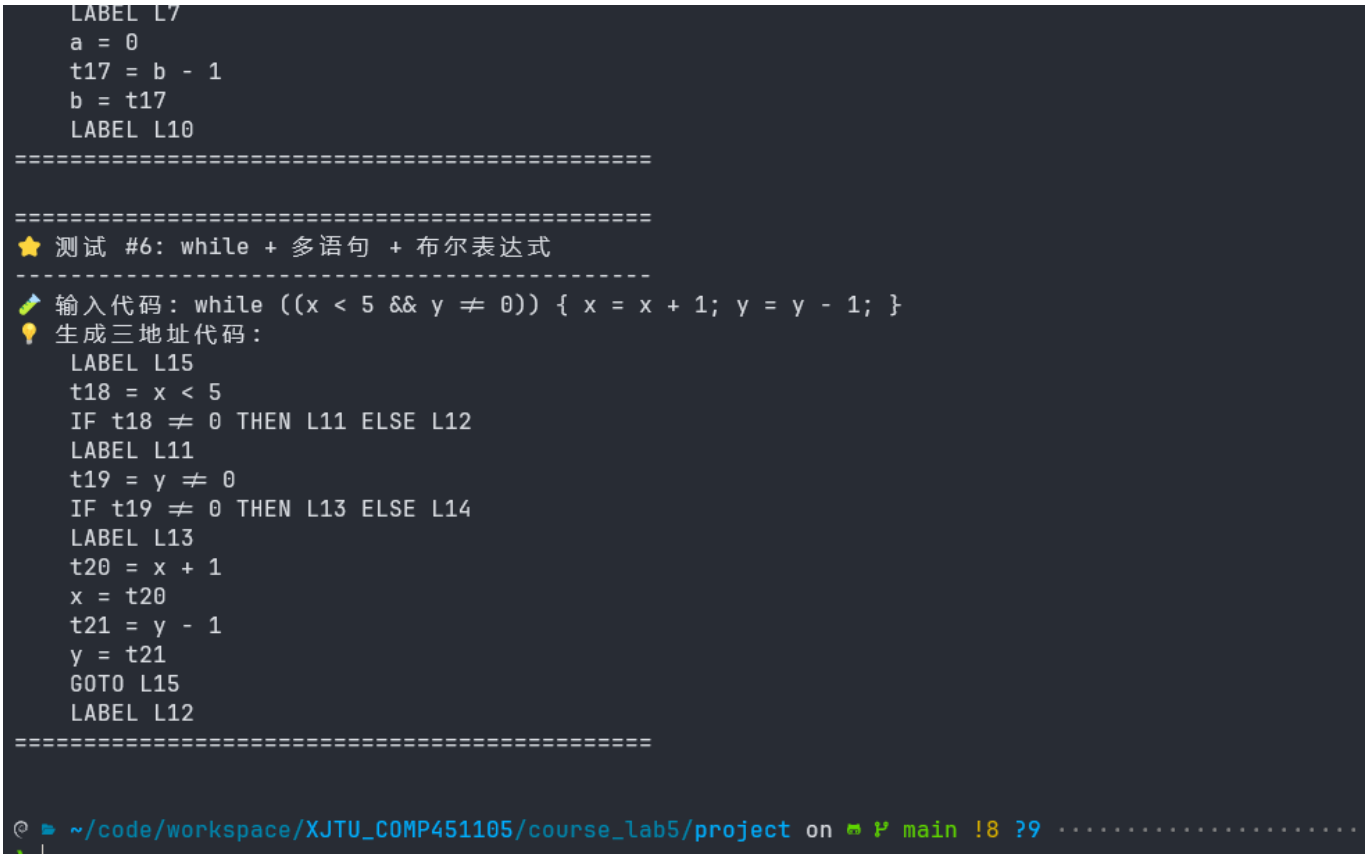
```
❯ cd code/workspace/XJTU_COMP451105/course_lab5/project
❯ go run main.go
================================================
⭐ 测试 #1: 简单赋值语句
------------------------------------------------
🧪 输入代码: x = a + 1;
💡 生成三地址代码:
    t1 = a + 1
    x = t1
================================================

================================================
⭐ 测试 #2: 函数调用
------------------------------------------------
🧪 输入代码: foo(a + 1, b, c * d);
💡 生成三地址代码:
    t2 = a + 1
    t3 = c * d
    PAR t3
    PAR b
    PAR t2
    t4 = CALL foo, 3
================================================

================================================
⭐ 测试 #3: 数组赋值
------------------------------------------------
🧪 输入代码: arr[i + 1, j * 2, 4] = 99;
💡 生成三地址代码:
    t5 = i + 1
    t6 = j * 2
    t7 = t5 * 5
    t8 = t7 + t6
    t9 = t8 * 20
    t10 = t9 + 4
    t11 = t10 * 4
    __val = 99
    arr[t11] = 99
================================================

================================================
⭐ 测试 #4: if-else + 简单布尔表达式
------------------------------------------------
🧪 输入代码: if (x < y) a = 1; else a = 0;
💡 生成三地址代码:
    t12 = x < y
    IF t12 ≠ 0 THEN L1 ELSE L2
    LABEL L1
    a = 1
    GOTO L3
    LABEL L2
    a = 0
    LABEL L3
================================================

================================================
⭐ 测试 #5: if-else + 复合布尔表达式与语句块
------------------------------------------------
🧪 输入代码: if ((x < 1 || y < 2) && !z) { a = 1; b = b + 2; } else { a = 0; b = b - 1; }
💡 生成三地址代码:
    t13 = x < 1
    IF t13 ≠ 0 THEN L4 ELSE L5
    LABEL L5
    t14 = y < 2
    IF t14 ≠ 0 THEN L6 ELSE L7
    LABEL L4
    t15 = !z ≠ 0
    IF t15 ≠ 0 THEN L8 ELSE L9
    LABEL L8
    a = 1
    t16 = b + 2
    b = t16
    GOTO L10
```



八、总结与展望

本系统在保持模块划分清晰、代码量简洁的前提下，成功实现了：

- 表达式求值、临时变量管理
- 短路布尔表达式逻辑
- 所有主要语句类型及语句块支持
- 多层嵌套控制流结构的正确处理

后续可拓展方向：

- 支持函数定义与作用域隔离
- 支持 AST 构建与优化
- 增加错误恢复机制与类型检查
- 输出四元式表结构用于优化与生成目标代码
