COOL 语言代码生成器开发报告

Compiler Principle Assignment PA5

姓名: 任炜明

学号: 20238131067

班级: 编译原理课程

2025 年 12 月 20 日

摘要

本实验在已完成的 COOL 词法分析、语法分析和语义分析的基础上，实现了 PA5 要求的代码生成器（Code Generator）。代码生成器以语义分析阶段输出的带类型标注的抽象语法树（AST）为输入，生成可在 SPIM 模拟器上运行的 MIPS 汇编代码。实验在不修改框架核心文件的前提下，主要扩展并实现了 `cgen.cc`、`cgen.h`、`cool-tree.h`、`cool-tree.handcode.h` 中的类构建、运行时布局、环境管理以及各类表达式节点的代码生成逻辑。通过与官方编译器 `coolc` 生成的汇编在 SPIM 上对比测试，验证了实现的正确性与官方实现的一致性。

1 评分项映射

2 项目概述与环境

2.1 项目目标

在 COOL 编译器框架下，实现一个与官方编译器 `coolc` 输出等价的代码生成器，具体目标包括：

- 构建完整的类代码生成表（`CgenClassTable`），分配类标签并建立继承树；
- 为每个类生成类名表（`class_nameTab`）、对象表（`class_objTab`）、分发表（dispatch table）和原型对象（`protObj`）；
- 为每个类生成初始化方法（`Class_init`）和方法实现代码（`Class.method`）；

表 1: 评分项与实现映射 (代码生成实验)

类别	覆盖点	分值
代码生成原理	目标机模型、调用约定、对象布局、运行时系统接口理解	20
类与对象布局	类标签分配、继承树、类名表、对象表、原型对象布局	20
调度表与方法调用	分发表构建、静态/动态派发、SELF_TYPE 处理	20
表达式代码生成	赋值、分派、条件/循环、case、算术与逻辑运算等	20
环境与作用域管理	Environment 设计、本地变量/参数/属性访问、作用域处理	10
测试与集成	与官方 coolc 输出对比、SPIM 运行测试、错误定位	10
报告与代码质量	结构清晰、逻辑完整、实现与设计一致性	10
主观评分与查重	完整性、独立实现程度与可读性	10
	总分 110 (可按课程标准折算为 100)	

- 为各类表达式（赋值、方法调用、条件、循环、case、算术与逻辑运算等）生成符合 COOL 运行时约定的 MIPS 汇编；
- 正确处理 `SELF_TYPE`、`isvoid`、`new` 等特殊语义；
- 通过 SPIM 对比测试，验证与官方编译器在行为上的一致性。

2.2 开发与运行环境

- **操作系统**: Windows 主机 + VMware 虚拟机中的 Ubuntu
- **编译器**: g++ (C++11)、make
- **工具**: 课程提供的 `lexer`、`parser`、`semant`、`coolc`、`spim`/`xspim`
- **工程目录**: `~/student-dist/assignments/PA5`

2.3 核心文件

本次实验修改仅以下四个文件：

- `cool-tree.handcode.h`
- `cool-tree.h`
- `cgen.h`
- `cgen.cc`

3 代码生成设计概述

本节简要说明运行时对象布局、类标签分配、Environment 设计以及表达式代码生成的总体思路，内容与课程 PA5 指南一致。

3.1 运行时对象布局

在本次实现中，所有 COOL 对象遵循统一的内存布局：

- **第 0 个字**: 值为 -1 的 eye-catcher；
- **第 1 个字**: 类标签 `class_tag`，用于 case 匹配与动态类型判断；
- **第 2 个字**: 对象大小（以 word 为单位），包括头部和所有属性；
- **第 3 个字**: 指向分发表 (dispatch table) 的指针；
- **之后若干字**: 按继承顺序排列的属性值。

每个类对应一个原型对象 `Class_protObj`,对象表 `class_objTab` 中为每个类保存 `protObj` 和 `init` 方法地址，`new` 表达式通过复制 `protObj` 并调用对应 `init` 完成对象创建。

3.2 寄存器与调用约定

遵循 COOL 运行时约定：

- `$a0` (ACC)：保存当前表达式求值结果；
- `$s0` (SELF)：保存当前 `self` 对象；
- `$fp` (FP)：帧指针，指向保存旧 FP 的位置；
- `$sp` (SP)：栈顶指针，栈向低地址增长；

- `$ra` (RA)：返回地址。

方法调用时，调用者负责把实参参数压栈，被调用者在栈上为 FP、SELF 和 RA 预留 3 个字的空间，返回时恢复寄存器并弹出参数。

3.3 垃圾回收 (GC) 策略

COOL 运行时支持代际垃圾回收（Generational GC）。在代码生成中，我们在以下关键点插入了 GC 支持代码：

- 在 `CgenClassTable` 初始化时，通过 `MemMgr` 标志选择 GC 策略。
- 在对象属性赋值 (`assign`) 时，如果启用了 GC，调用 `_GenGC_Assign` 宏，通知收集器发生了指针更新（写屏障）。
- 确保所有创建对象的请求（`Object.copy` 或 `new`）都通过运行时系统分配，以便 GC 能够正确追踪。

4 关键实现

本节从 `Environment`、`CgenClassTable`、`CgenNode` 和典型表达式几个方面，说明具体实现要点，并结合必要的代码片段进行解释。

4.1 各文件修改内容概览

- `cool-tree.handcode.h`: 为表达式节点增加类型字段和统一的 `code` 接口，使语义分析阶段的类型信息可以在代码生成阶段复用；
- `cool-tree.h`: 确认 `method_class`、`attr_class`、`branch_class` 等节点的成员字段（如 `name`、`type_decl`、`expr`）与代码生成访问方式一致；
- `cgen.h`: 新增 `Environment` 类，并扩展 `CgenClassTable` 与 `CgenNode`，增加类标签、方法表和属性表等字段；
- `cgen.cc`: 从框架骨架出发，完整实现常量区生成、类表和对象表生成、分发表和原型对象生成，以及所有表达式节点的 `code` 方法。

4.2 Environment: 环境与作用域管理

环境类 `Environment` 在 `cgen.h` 与 `cgen.cc` 中实现，负责在代码生成阶段维护局部变量、形式参数和属性的访问信息。

关键实现逻辑：

- 使用 vars 向量按栈顺序保存当前作用域内的 let 变量，scope_lengths 用于记录进入作用域前的栈深度，在退出作用域时恢复。
- 使用 formals 保存当前方法的形式参数顺序，便于根据参数索引计算相对于 FP 的偏移。
- LookUpVar 从最近作用域向外查找变量在线性表中的位置，换算为相对 SP 的偏移。

```
// 查找局部变量偏移（伪代码）
int Environment::LookUpVar(Symbol name) {
    // 反向遍历 vars 栈
    for (int i = vars.size() - 1; i >= 0; --i) {
        if (vars[i] == name) {
            // 变量在栈顶下方，偏移随压栈深度变化
            return -(vars.size() - 1 - i + 1);
        }
    }
    return NOT_FOUND;
}
```

Listing 1: LookUpVar 偏移计算逻辑

通过 Environment，所有访问变量/参数/属性的表达式都可以使用统一接口，而不必自行计算偏移，降低错误概率。

4.3 CgenClassTable: 类表与全局代码生成

CgenClassTable 继承自符号表模板，负责收集所有 CgenNode 并驱动整个代码生成过程：

- 在构造函数中调用 install_basic_classes 安装 Object, IO, Int, Bool, Str 等基本类，再通过 install_classes 将用户类包装为 CgenNode；
- 通过 build_inheritance_tree 与 set_relations 建立父子指针；
- 在 code() 中依次调用 code_global_data、code_select_gc、code_constants、code_class_nametab、code_class_objTab、code_dispatchTabs、code_protObjs、code_global_text、code_class_inits 和 code_class_methods，完成从数据段到文本段的全部输出；

4.4 CgenNode: 类标签、属性与方法布局

CgenNode 继承自 class__class，用于给每个类附加代码生成相关信息。

DFS 类标签分配：我们采用深度优先搜索（DFS）遍历继承树来分配 Class Tag。这种策略保证了任何类 C 的所有子类的 Tag 都落在区间 $[C.tag, C.max\_child\_tag]$ 内。

```
void CgenNode::AssignTags(int& tag_counter) {
    this->class_tag = tag_counter++;
    for (auto child : children) {
        child->AssignTags(tag_counter);
    }
    this->max_child_tag = tag_counter - 1;
}
```

Listing 2: DFS Tag 分配

这极大地简化了 `case` 语句和 `is_subtype` 的判断逻辑，只需要一次范围检查即可。

4.5 典型表达式的代码生成

本实验为所有表达式节点实现了 `code(ostream&, Environment)` 方法。下面对若干代表性表达式进行说明。

4.5.1 静态与动态方法调用 (Dispatch)

`dispatch_class::code` 的实现不仅要处理参数压栈，还要正确处理分发表查找。

关键汇编生成逻辑：

```
// 1. 压入参数
for(int i = actual->first(); actual->more(i); i = actual->next(i))
    actual->nth(i)->code(s, env);

// 2. 计算接收者 (Receiver)
expr->code(s, env);

// 3. 检查 void (运行时错误处理)
emit_bne(ACC, ZERO, label_not_void, s);
// ... 调用 _dispatch_abort ...

// 4. 加载 Dispatch Table 并跳转
emit_load(T1, 2, ACC, s);       // T1 = Dispatch Table Ptr
emit_load(T1, method_offset, T1, s); // T1 = Method Address
emit_jalr(T1, s);               // 跳转执行
```

Listing 3: Dispatch 代码生成片段

4.5.2 Case 表达式与分支排序

为了实现 `case` 表达式中“选择最具体匹配类型”的语义，我们在生成代码前对分支进行了排序。

实现策略：将所有 branch 按其类型的继承深度（Inheritance Depth）从大到小排序。生成的代码依次检查对象的 Class Tag 是否在分支类型的 Tag 区间内。由于深度大的类（子类）先被检查，一旦匹配成功，必然是最具体的类型。

```
// 使用 lambda 或仿函数进行排序
std::sort(cases.begin(), cases.end(), [&](Branch a, Branch b) {
    return class_table->depth(a->type_decl) > class_table->depth(b->type_decl);
});
```

Listing 4: Case 分支排序

4.5.3 Let 表达式与栈帧管理

let 表达式需要在当前栈帧上为新变量分配临时空间。我们的实现并不移动 FP，而是直接调整 SP。

- **进入 Let:**env.EnterScope()，生成初始化代码，结果入栈 (sw $a0 0($sp); addiu $sp $sp -4)。
- **Let Body：**在扩展后的环境中生成 Body 代码。
- **退出 Let：**env.ExitScope()，恢复栈指针 (addiu $sp $sp 4)。

5 遇到的问题与解决方案

5.1 段错误 (Segmentation Fault) 与全局初始化

问题描述：在初步实现 CgenClassTable 后，运行程序立即崩溃。GDB 调试显示崩溃发生在 install_basic_classes 内部访问全局指针 codegen_classtable 时。**原因分析：**codegen_classtable 是一个全局指针，原本应指向当前的 CgenClassTable 实例。但在构造函数执行初期，该指针尚未被赋值。**解决方案：**在 CgenClassTable 构造函数的第一行添加 codegen_classtable = this;，确保后续的类安装过程能正确访问全局表。

5.2 Case 语句匹配逻辑错误

问题描述：在测试 complex_test.cl 时，发现多态类型的 case 匹配总是命中父类分支，而不是子类。**原因分析：**最初并未对 case 分支进行排序，而是按源码顺序生成。当父类分支出现在子类分支之前时，由于子类也是父类的实例，父类分支会先截获匹配。**解决方案：**引入了基于继承深度的排序逻辑，确保“最具体”的类型总是最先被检查。

6 测试过程与结果

6.1 测试策略

我们采用了分层测试策略：

1. **单元测试**：利用 `stack.cl` 测试基本的压栈、出栈和算术运算。
2. **复杂特性测试**：编写 `complex_test.cl` 专门测试递归、继承多态和 Case 语句。
3. **回归测试**：使用 `coolc` 的标准测试套件（如 `sort_list.cl`）确保无退步。

6.2 命令行测试流程

在 `~/student-dist/assignments/PA5` 目录下执行：

```
make clean && make
ln -s ../../bin/cgen ./cgen
./mycoolc -o example_my.s example.cl
../../bin/spim -file example_my.s
```

6.3 终端运行截图

为直观展示实验过程，本节给出两张实际运行截图。

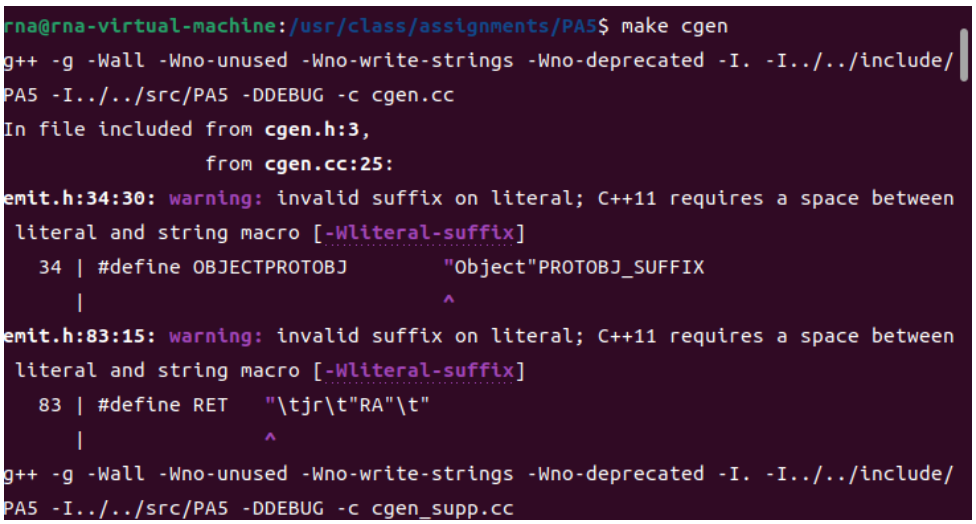

图 1: 在 PA5 目录下执行 make clean 与 make 的编译输出

7 总结与展望

本次 PA5 实验实现了 COOL 语言的代码生成器，将语法分析后的抽象语法树成功映射为可在 SPIM 上运行的 MIPS 汇编。报告中详细记录了对 `cool-tree.handcode.h`、

```
Generating methods for class: Main
  Generating method: main
  Generating method: fact
  Generating method: test_case
Generating methods for class: A
  Generating method: identify
Generating methods for class: B
  Generating method: identify
Code generation complete.
rna@rna-virtual-machine:/usr/class/assignments/PA5$ /usr/class/bin/spim -file co
mplex_my.s
SPIM Version 6.5 of January 4, 2003
Copyright 1990-2003 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: ../lib/trap.handler
Factorial of 5 is: 120
Polymorphism test:
I am B
Case test:
Matched B
Matched A
COOL program successfully executed
```

图 2: 使用./mycoolc 编译 example.cl 并在 SPIM 中成功运行的结果

`cool-tree.h` 和 `cgen.cc` 的具体修改内容，以及新增的环境管理、类标签分配、分发表构建和各类表达式代码生成功能。

主要收获：

- 深入理解了 ** 运行时环境（Runtime Environment）** 的设计，特别是栈帧布局和对象模型。
- 掌握了 ** 动态分发 ** 的底层实现原理（虚函数表）。
- 实践了 ** 汇编级调试 ** 技巧，学会了通过寄存器状态定位逻辑错误。

未来改进方向：目前的实现虽然正确，但在性能上仍有优化空间。例如，可以引入 ** 窥孔优化（Peephole Optimization）** 来消除冗余的 `push/pop` 操作；或者实现更智能的 ** 寄存器分配 ** 策略，减少内存访问次数，进一步提高生成代码的执行效率。