PA5 代码生成器实现报告

课程名称：**编译原理**

项目名称：PA5 - COOL 代码生成器实现

学生姓名：**陈翰林**

学号：**20238131070**

一、概述

1.1 背景

COOL（Classroom Object-Oriented Language）是一门为教学设计的面向对象编程语言，其编译过程分为词法分析、语法分析、语义分析和代码生成四个核心阶段。本项目（PA5）是 COOL 编译器的最后一个核心部件——代码生成器，承接 PA4 语义分析器的输出结果，将经过类型检查的抽象语法树（AST）转换为可在 MIPS 架构模拟器（SPIM）上运行的汇编代码。

1.2 目标

核心目标是实现一个完整、正确、高效的 COOL 语言代码生成器，具体目标包括：

- 接收 PA4 语义分析器输出的、带有类型标注的 AST，正确识别其中的类定义、属性、方法及各类表达式；
- 遵循 MIPS 汇编指令集规范，生成对应的汇编代码，支持 COOL 语言的所有核心语法特性（类继承、方法调用、属性访问、各类表达式运算等）；
- 实现内存管理机制，支持对象的创建、初始化与内存分配，兼容课程提供的垃圾回收（GC）模块；
- 保证生成代码的正确性，能够通过所有标准测试用例（包括基础语法测试、继承测试、多态测试、异常处理测试等）；
- 优化生成代码的效率，减少冗余指令，合理利用寄存器资源。

1.3 核心功能范围

本代码生成器需支持 COOL 语言的全部核心功能，具体包括：

- 类与继承机制：支持类定义、父类继承、方法重写、基础类（Object、Int、Bool、Str、IO）的正确实现；
- 属性与方法：支持属性的定义与初始化、方法的定义与调用（动态分派、静态分派）；
- 表达式运算：支持算术运算（+、-、*、/、neg）、比较运算（<、≤、=）、逻辑运算、条件表达式（cond）、循环表达式（loop）、类型判断表达式（typcase）、let 表达式、块表达式（block）等；
- 常量处理：支持整数、布尔值、字符串常量的正确存储与访问；
- 内存管理：支持对象的创建（new）、内存分配、垃圾回收接口对接；
- 输入输出：支持 IO 类的标准输入（in_int、in_string）与输出（out_int、out_string）方法。

二、需求分析

2.1 功能需求

2.1.1 AST 解析与遍历

需正确解析 PA4 输出的 AST 结构，遍历所有节点（类节点、属性节点、方法节点、表达式节点等），并根据节点类型生成对应的汇编代码。要求遍历过程中能够准确获取节点的类型信息、子节点关系、属性/方法名称及参数信息。

2.1.2 基础类代码生成

需自动生成 COOL 语言预定义基础类（Object、Int、Bool、Str、IO）的汇编代码，包括其属性（如 Int 的 val、Str 的 str_field）、方法（如 Object 的 copy、type_name，Str 的 length、concat 等）的实现，确保基础类功能符合 COOL 语言规范。

2.1.3 用户自定义类代码生成

支持用户自定义类的代码生成，包括：

- 类的继承关系处理：正确继承父类的属性与方法，支持方法重写；
- 属性初始化：生成属性的初始化代码，确保属性在对象创建时被正确赋值；
- 方法实现：将方法体中的 AST 转换为汇编代码，支持方法内的各类表达式运算。

2.1.4 表达式代码生成

针对 COOL 语言的各类表达式，生成正确的汇编代码，确保运算逻辑准确、类型匹配：

- 算术表达式：确保操作数为 Int 类型，生成对应的 MIPS 算术指令（add、sub、mul、div 等）；
- 比较表达式：生成比较指令（blt、bleq、beq 等），确保比较结果为 Bool 类型；
- 条件表达式：正确处理分支跳转，确保 then 分支与 else 分支的类型兼容性；
- 循环表达式：生成循环跳转指令，支持循环条件的判断与循环体的执行；
- 类型判断表达式：正确处理 typcase 的分支匹配，支持按类标签进行跳转；
- 方法调用表达式：支持动态分派（dispatch）与静态分派（static_dispatch），正确处理参数传递与返回值获取。

2.1.5 内存管理与垃圾回收

实现对象的内存分配与管理机制，包括：

- 对象结构定义：按照 COOL 语言规范，定义对象的内存布局（类标签、对象大小、分派表指针、属性等）；
- 内存分配：对接课程提供的 GC 模块，支持通过 GC 接口申请内存；
- 垃圾回收对接：生成 GC 所需的标记代码，支持 GC 对对象的遍历与回收。

2.2 非功能需求

2.2.1 正确性

生成的汇编代码需能在 SPIM 模拟器上正确运行，准确实现 COOL 源程序的逻辑，无语法错误、逻辑错误及运行时错误。需通过所有标准测试用例，包括空字符串、零值、极端循环条件等边界场景。

2.2.2 高效性

生成的汇编代码应尽可能简洁，减少冗余指令；合理利用 MIPS 寄存器（如 ACC、SELF、T1-T3 等），减少内存访问次数，提升程序运行效率。

2.2.3 可维护性

代码生成器的实现应采用模块化设计，各功能模块（如 AST 遍历模块、表达式生成模块、内存管理模块）职责清晰，代码注释完整，便于后续维护与扩展。

2.2.4 兼容性

生成的汇编代码需兼容 SPIM 模拟器的所有版本，支持课程提供的各类 GC 模块（NoGC、GenGC、ScnGC），可通过配置参数切换 GC 策略。

2.3 接口需求

2.3.1 输入接口

输入为 PA4 语义分析器输出的、带有类型标注的 AST，具体为 `Program` 类型的对象，包含所有类定义（`Classes`）、属性（`Feature`）、方法（`method`）及表达式（`Expression`）节点。

2.3.2 输出接口

输出为标准 MIPS 汇编代码文件（.s），包含.data 段（存储常量、全局变量、对象原型等）和.text 段（存储函数/方法的汇编指令）。输出文件可直接被 SPIM 模拟器加载并运行。

2.3.3 外部模块接口

需与以下外部模块对接：

- GC 模块：通过调用 GC 提供的初始化函数（_GenGC_Init 等）、内存分配函数（_gc_check 等）、回收函数（_GenGC_Collect 等）实现内存管理；
- SPIM 系统调用：通过 MIPS 的 syscall 指令实现输入输出功能（如打印字符串、读取整数等）；
- 语义分析器：接收语义分析器输出的类型标注信息，确保生成代码的类型安全性。

三、总体设计

3.1 设计理念

本代码生成器采用“模块化+访问者模式”的设计理念，核心思路如下：

- 模块化设计：将代码生成器划分为 AST 遍历模块、基础类生成模块、用户类生成模块、表达式生成模块、内存管理模块、辅助工具模块等，各模块职责单一、接口清晰；
- 访问者模式：基于 AST 的结构特点，为每个 AST 节点类型（如 class__class、

method_class、assign_class 等）定义对应的代码生成方法（code()），通过遍历 AST 节点时调用对应方法，实现“节点类型-代码生成逻辑”的一一映射；

分层生成：先生成.data 段的全局数据（常量、类名表、对象原型、分派表等），再生成.text 段的代码（初始化函数、方法实现、主函数等），确保内存布局合理、代码执行顺序正确；

- 寄存器优化：基于 MIPS 寄存器的特点，划分专用寄存器（如 ACC 用于存储计算结果、SELF 用于指向当前对象），减少寄存器的频繁保存与恢复，提高代码效率。

3.2 系统架构

本代码生成器的系统架构分为五层，从顶层到底层依次为：

1. 入口层：包含 `program_class::cgen()`方法，作为代码生成器的入口，负责初始化全局符号、创建类表、触发代码生成流程；
2. 类表管理层：包含 `CgenClassTable` 类，负责管理所有类的信息（继承关系、属性列表、方法列表等），构建继承树与分派表，协调各模块的代码生成；
3. 节点生成层：包含 `CgenNode` 类及各类 AST 节点的 `code()`方法，负责将具体的 AST 节点转换为汇编代码，是代码生成的核心层；
4. 辅助工具层：包含各类辅助函数（如 `emit_*`系列函数），负责生成具体的 MIPS 指令、管理标签、处理常量等；
5. 外部对接层：负责与 GC 模块、SPIM 系统调用对接，实现内存分配、垃圾回收、输入输出等功能。

系统架构图如下：

```
mermaid
graph TD
    A[入口层: program_class::cgen()] --> B[类表管理层:
CgenClassTable]
    B --> C[节点生成层: CgenNode + AST 节点 code()方法]
    C --> D[辅助工具层: emit_*函数]
    D --> E[外部对接层: GC 模块 + SPIM 系统调用]
    B --> E
    C --> E
```

3.3 核心数据结构

本代码生成器的核心数据结构包括类表、节点、符号表及各类辅助结构，具体如下：

3.3.1 CgenClassTable

类表，继承自`SymbolTable<Symbol,CgenNode>`，用于管理所有类的信息，核心成员包括：

- `nds`：存储所有类节点（CgenNode）的列表；
- `str`：输出流，用于写入汇编代码；
- `stringclasstag/intclasstag/boolclasstag`：基础类（Str、Int、Bool）的类标签，用于对象的类型识别；
- `class_num`：类的总数；
- `label_index`：标签索引，用于生成唯一的跳转标签。

核心方法包括类的安装（`install_classes`）、继承树构建（`build_inheritance_tree`）、全局数据生成（`code_global_data`）、方法表生成（`build_methodtab`）等。

3.3.2 CgenNode

类节点，继承自`class__class`，用于存储单个类的详细信息，核心成员包括：

- `parentnd`：父类节点指针；
- `children`：子类节点列表；
- `basic_status`：标记是否为基础类（Basic/NotBasic）；
- `class_tag`：类标签，用于对象的类型识别；
- `attr_list`：属性列表；
- `method_list`：方法列表；
- `method_symbtab`：方法符号表，用于快速查找方法。

核心方法包括属性/方法表构建（`build_feature_tab`）、对象原型生成（`code_class_protobj`）、初始化代码生成（`code_init`）、方法代码生成（`code_method`）等。

3.3.3 符号表（SymbolTable）

用于存储符号（如类名、属性名、方法名）与对应信息（如类节点、属性偏移、方法偏移）的映射，支持作用域管理，确保符号的正确查找与重名处理。

3.3.4 辅助结构

- `BoolConst`：布尔常量类，用于生成布尔值的定义与引用代码；

- `StringEntry/IntEntry`：字符串/整数常量表项，用于管理常量的存储与代码生成。

3.4 代码生成流程

本代码生成器的核心流程分为初始化、全局数据生成、代码生成、收尾四个阶段，具体流程如下：

6. **初始化阶段**：
    - 调用`initialize_constants()`初始化预定义符号（如 self、Object、Int、Bool 等）；
    - 创建`CgenClassTable`实例，传入 AST 的类列表与输出流；
    - 安装基础类（`install_basic_classes`）与用户自定义类（`install_classes`）；
    - 构建继承树（`build_inheritance_tree`）与属性/方法表（`build_feature_tab`）。
7. **全局数据生成阶段**（.data 段）：
    - 生成全局数据定义（`code_global_data`），包括类名表、对象原型、常量等的全局声明；
    - 选择并生成 GC 相关代码（`code_select_gc`）；
    - 生成常量代码（`code_constants`），包括字符串、整数、布尔值常量；
    - 生成类名表（`code_class_nametab`）、对象原型（`code_protoobj`）、分派表（`build_methodtab`）、对象表（`code_objtab`）。
8. **代码生成阶段**（.text 段）：
    - 生成全局文本定义（`code_global_text`），包括堆起始地址、初始化函数、方法的全局声明；
    - 生成类初始化代码（`code_init`），包括父类初始化调用与属性初始化；
    - 生成方法代码（`code_method`），遍历所有用户自定义类的方法，将方法体 AST 转换为汇编代码。
9. **收尾阶段**：
    - 写入代码结束标记；
    - 释放相关资源，完成代码生成。

代码生成流程图如下：

```mermaid
graph TD
    A[初始化预定义符号] --> B[创建 CgenClassTable]
    B --> C[安装基础类与用户类]
    C --> D[构建继承树与属性/方法表]
    D --> E[生成.data 段全局数据]
    E --> E1[生成全局数据声明]
    E1 --> E2[生成 GC 代码]
    E2 --> E3[生成常量代码]
    E3 --> E4[生成类名表/分派表/对象原型]
    E4 --> F[生成.text 段代码]
    F --> F1[生成全局文本声明]
    F1 --> F2[生成类初始化代码]
    F2 --> F3[生成方法代码]
    F3 --> G[收尾：写入结束标记]
```

四、核心模块实现

4.1 初始化模块实现

4.1.1 预定义符号初始化

调用 `initialize_constants()`函数初始化 COOL 语言的预定义符号，包括关键字（self、SELF_TYPE）、基础类名（Object、Int、Bool、Str、IO）、方法名（cool_abort、copy、type_name）、属性名（val、str_field）等。这些符号被添加到全局符号表（idtable）中，供后续代码生成过程中快速查找。符号表的核心实现依赖 `stringtab.h` 中定义的符号管理机制，配合 `cool-tree.cc` 中提供的符号创建与复制接口，确保符号的唯一性与可追溯性。相关核心代码片段如下（来自 `cool-tree.cc`）：

```cpp
// 符号相关的节点复制实现，确保符号在 AST 遍历与代码生成中的一致性
Class_ class__class::copy_Class_()
{
   return new class__class(copy_Symbol(name), copy_Symbol(parent),
features->copy_list(), copy_Symbol(filename));
}
```

```
Feature method_class::copy_Feature()
{
   return new method_class(copy_Symbol(name), formals-
>copy_list(), copy_Symbol(return_type), expr->copy_Expression());
}

// 基础符号创建接口，为代码生成提供统一的符号引用
Symbol copy_Symbol(Symbol s) {
    if (s == nullptr) return nullptr;
    return s->copy();
}
```

4.1.2 类安装与继承树构建

类安装过程分为基础类安装与用户自定义类安装：

- **基础类安装**：通过`install_basic_classes()`方法安装 Object、Int、Bool、Str、IO 五个基础类。基础类的属性与方法由系统预设（如 Int 类的 val 属性、IO 类的 out_int 方法），无需用户定义。安装过程中，将基础类节点添加到类表中，并标记为 Basic 类型。
- **用户自定义类安装**：通过`install_classes()`方法遍历 AST 中的用户类列表，为每个用户类创建`CgenNode`实例，标记为 NotBasic 类型，并添加到类表中。

继承树构建通过`build_inheritance_tree()`与`set_relations()`方法实现：遍历所有类节点，查找其父类节点，设置父类指针（parentnd）与子类列表（children），构建完整的继承关系树。这确保了后续属性/方法的继承与重写处理的正确性。

4.1.3 属性/方法表构建

通过`build_feature_tab()`方法构建每个类的属性表（attr_list）与方法表（method_list）：

- **属性表构建**：从父类开始递归收集属性，确保子类继承父类的属性，且子类新增属性排在父类属性之后。属性表记录了属性的名称与偏移量，用于对象内存布局的确定与属性访问代码的生成。
- **方法表构建**：同样从父类递归收集方法，支持方法重写（子类方法覆盖父类同名方法）。方法表记录了方法的名称与偏移量，用于分派表的生成与方法调用时的地址查找。

4.2 全局数据生成模块实现

全局数据生成模块负责生成.data 段的内容，包括全局变量、常量、类名表、分派表、对象原型等，这些数据是程序运行的基础。

4.2.1 全局数据声明

通过 `code_global_data()`方法生成全局数据的声明，包括:

- 类名表（CLASSNAMETAB）的全局声明;
- 基础类与 Main 类对象原型的全局声明;
- 布尔常量（true/false）的全局声明;
- 基础类标签（INTTAG、BOOLTAG、STRINGTAG）的全局声明，用于对象类型的快速识别。

4.2.2 常量代码生成

通过 `code_constants()`方法生成字符串、整数、布尔值常量的代码:

- **字符串常量**：遍历字符串表（stringtable），为每个字符串生成对应的内存布局（包括类标签、大小、分派表指针、字符串长度、字符串内容等），并对齐到字边界。核心实现依赖 `cgen_supp.cc` 中定义的字符串常量发射函数 `emit_string_constant`，该函数会根据字符类型自动选择`.ascii` 或`.byte` 指令生成汇编代码，处理特殊字符（如换行、制表符、引号等）。核心代码片段如下（来自 `cgen_supp.cc`）：

```
cpp
static int ascii = 0;

void ascii_mode(ostream& str)
{
  if (!ascii)
    {
      str << "\t.ascii\t\"";
      ascii = 1;
    }
}

void byte_mode(ostream& str)
{
  if (ascii)
    {
      str << "\"\n";
      ascii = 0;
```

```
    }
}

void emit_string_constant(ostream& str, char* s)
{
  ascii = 0;
  while (*s) {
    switch (*s) {
    case '\n':
      ascii_mode(str);
      str << "\\n";
      break;
    case '\t':
      ascii_mode(str);
      str << "\\t";
      break;
    case '\\':
      byte_mode(str);
      str << "\t.byte\t" << (int) ((unsigned char) '\\') << endl;
      break;
    case '"' :
      ascii_mode(str);
      str << "\\\"";
      break;
    default:
      if (*s >= ' ' && ((unsigned char) *s) < 128)
{
  ascii_mode(str);
  str << *s;
}
      else
{
  byte_mode(str);
  str << "\t.byte\t" << (int) ((unsigned char) *s) << endl;
}
      break;
    }
    s++;
  }
  byte_mode(str);
  str << "\t.byte\t0\t" << endl;
}
```

- **整数常量**：遍历整数表（inttable），为每个整数生成内存布局（包括类标签、大

小、分派表指针、整数值等）。

- **布尔常量**：生成 true 与 false 两个布尔常量的内存布局，包括类标签、大小、分派表指针、布尔值等。布尔常量的命名与存储遵循 `emit.h` 中定义的规范，核心定义如下（来自 `emit.h`）：

```cpp
#define BOOLCONST_PREFIX     "bool_const"
#define BOOLTAG              "_bool_tag"
#define STRINGTAG            "_string_tag"
#define INTTAG               "_int_tag"
// 布尔常量的内存布局大小定义
#define BOOL_SLOTS        1
#define DEFAULT_OBJFIELDS 3  // 基础对象头：标签、大小、分派表指针
```

4.2.3 分派表与对象原型生成

- **分派表生成**：通过 `build_methodtab()`方法为每个类生成分派表，分派表是一个函数指针数组，存储该类所有方法的地址。分派表的生成确保了动态分派的正确性——方法调用时，通过对象的分派表指针查找对应的方法地址。分派表的命名规范严格遵循 `emit.h` 中的定义，核心代码逻辑需结合类表中的方法偏移量，生成对应的函数指针列表。相关核心定义如下（来自 `emit.h`）：

```cpp
// 分派表与方法命名规范，确保代码生成的一致性
#define DISPTAB_SUFFIX       "_dispTab"
#define METHOD_SEP           "."
#define CLASSINIT_SUFFIX     "_init"
#define PROTOBJ_SUFFIX       "_protObj"
// 全局数据段相关指令宏定义
#define GLOBAL        "\t.globl\t"
#define ALIGN         "\t.align\t2\n"
#define WORD          "\t.word\t"
#define LABEL               ":\n"
```

- **对象原型生成**：通过 `code_protoobj()`方法为每个类生成对象原型（protoobj），对象原型是该类所有对象的模板，包含类标签、对象大小、分派表指针、属性初始值等。对象创建时，通过复制对象原型实现初始化。对象原型的内存布局需严格遵循 `emit.h` 中定义的对象头结构，确保与 GC 模块的内存管理逻辑兼容。

4.3 类与方法代码生成模块实现

4.3.1 类初始化代码生成

通过`code_init()`方法生成每个类的初始化代码，初始化代码的核心逻辑包括:

- 保存调用者保存的寄存器（如 FP、SELF、$s1、RA），确保函数调用的栈帧完整性;
- 调用父类的初始化方法，确保父类的属性被正确初始化;
- 初始化当前类的属性，将属性的初始值存储到对应的内存偏移量;
- 恢复保存的寄存器，返回当前对象（SELF）。

4.3.2 方法代码生成

通过`code_method()`方法与各类`method_class::code_method()`方法生成方法的汇编代码，核心逻辑包括:

- 栈帧初始化：保存调用者保存的寄存器，设置栈帧指针（FP）与栈指针（SP）;
- 参数处理：将方法的形式参数添加到形式参数列表（formal_list），记录参数的偏移量，便于参数访问;
- 方法体代码生成：调用方法体表达式的`code()`方法，将 AST 转换为汇编代码;
- 返回值处理：确保方法的返回值存储在 ACC 寄存器中;
- 栈帧清理：恢复保存的寄存器，调整栈指针，返回调用者。

方法代码生成的核心依赖`cool-tree.cc`中各类表达式节点的`copy_Expression()`方法确保 AST 结构的完整性，同时结合`emit.h`中定义的寄存器与指令宏，生成规范的 MIPS 汇编代码。以算术表达式中的加法为例，其代码生成逻辑需调用`plus_class::copy_Expression()`复制表达式节点，再生成对应的 MIPS 加法指令。相关核心代码如下（来自`cool-tree.cc`）:

```
cpp
// 加法表达式节点的复制，为代码生成提供完整的表达式结构
Expression plus_class::copy_Expression()
{
   return new plus_class(e1->copy_Expression(), e2-
>copy_Expression());
}

// 加法表达式的 dump 方法（辅助调试），代码生成阶段可参考其节点遍历逻辑
void plus_class::dump(ostream& stream, int n)
{
```

```
   stream << pad(n) << "plus\n";
   e1->dump(stream, n+2);
   e2->dump(stream, n+2);
}
```

对应的 MIPS 指令生成需遵循 `emit.h` 中定义的寄存器规范，核心指令宏如下（来自 `emit.h`）：

```cpp
// 寄存器命名规范，确保方法调用与表达式运算中的寄存器使用一致性
#define ZERO "$zero"// Zero register
#define ACC  "$a0"// Accumulator （存储运算结果与返回值）
#define A1   "$a1"// For arguments to prim funcs
#define SELF "$s0"// Ptr to self (callee saves)
#define T1   "$t1"// Temporary 1
#define T2   "$t2"// Temporary 2
// 算术运算指令宏
#define ADD   "\tadd\t"
#define ADDI  "\taddi\t"
#define ADDU  "\taddu\t"
#define SUB   "\tsub\t"
#define MUL   "\tmul\t"
#define DIV   "\tdiv\t"
```

4.4 表达式代码生成模块实现

表达式代码生成是本项目的核心难点，需针对各类表达式的语义，生成正确的汇编代码。以下是关键表达式的实现细节：

4.4.1 算术与比较表达式

算术表达式（plus、sub、mul、div、neg）与比较表达式（lt、leq、eq）的生成逻辑如下：

- **类型校验**：确保操作数类型正确（如算术表达式的操作数必须为 Int 类型）；

示例：plus_class::code()方法先生成两个操作数的代码，将结果存入$s1 寄存器，然后调用 Object.copy 方法创建新的 Int 对象，将运算结果存入新对象的 val 属性，最后将新对象的地址存入 ACC。

4.4.2 条件与循环表达式

- **条件表达式（cond）**：生成条件判断指令，根据条件结果跳转到 then 分支或

else 分支。确保 then 分支与 else 分支的类型兼容性，将分支结果存入 ACC。

- **循环表达式（loop）**：生成循环标签与跳转指令，先判断循环条件，若条件为真则执行循环体，执行完成后跳回条件判断处；若条件为假则退出循环。

4.4.3 方法调用表达式

支持动态分派（dispatch）与静态分派（static_dispatch），核心实现如下：

- **动态分派**：通过对象的分派表指针查找方法地址。生成代码时，先获取对象的分派表指针（存储在对象的第 3 个偏移量），再根据方法偏移量查找对应的方法地址，最后通过 jalr 指令调用方法。
- **静态分派**：直接调用指定类的方法，无需通过分派表查找。生成代码时，直接跳转到指定类的方法地址（如 type_name.method_name）。

4.4.4 类型判断表达式（typcase）

typcase 表达式的生成逻辑如下：

- 先生成表达式的代码，获取表达式的对象地址；
- 获取对象的类标签，根据类标签排序分支（确保子类分支优先于父类分支）；
- 生成分支跳转指令，根据类标签匹配对应的分支并执行；
- 若没有匹配的分支，调用_case_abort 函数抛出异常。

4.4.5 赋值与对象访问表达式

- **赋值表达式（assign）**：先生成右值表达式的代码，再根据左值的类型（局部变量、形式参数、属性）查找对应的内存偏移量，生成存储指令（sw）将右值存入左值的内存地址。若开启 GC，还需生成 GC 赋值标记代码。
- **对象访问表达式（object）**：根据对象名称的类型（self、局部变量、形式参数、属性）查找对应的内存偏移量，生成加载指令（lw）将对象地址存入 ACC。

4.5 内存管理与 GC 对接模块实现

本代码生成器通过对接课程提供的 GC 模块实现内存管理，核心实现包括：

- **GC 初始化**：通过 `code_select_gc()`方法生成 GC 初始化代码，根据配置参数（cgen_Memmgr）选择对应的 GC 策略（NoGC、GenGC、ScnGC），生成 GC 初始化函数与回收函数的指针。GC 模块的对接依赖 `cgen_gc.h` 中定义的接口，核心初始化逻辑需在主函数中完成，相关主函数流程代码如下（来自 `cgen-phase.cc`）：

```cpp
#include "cgen_gc.h"

extern int optind;            // for option processing
extern char *out_filename;    // name of output assembly
extern Program ast_root;              // root of the abstract
syntax tree
FILE *ast_file = stdin;       // we read the AST from standard
input
extern int ast_yyparse(void); // entry point to the AST parser

int main(int argc, char *argv[]) {
  int firstfile_index;

  handle_flags(argc,argv);
  firstfile_index = optind;

  // 处理输出文件名
  if (!out_filename && optind < argc) {   // no -o option
      char *dot = strrchr(argv[optind], '.');
      if (dot) *dot = '\0'; // strip off file extension
      out_filename = new char[strlen(argv[optind])+8];
      strcpy(out_filename, argv[optind]);
      strcat(out_filename, ".s");
  }

  // 先完成 AST 解析，确保前端无错误后再生成代码
  ast_yyparse();

  // 生成汇编代码，包含 GC 初始化相关代码
  if (out_filename) {
      ofstream s(out_filename);
      if (!s) {
  cerr << "Cannot open output file " << out_filename << endl;
  exit(1);
      }
      ast_root->cgen(s); // cgen 方法中会调用 code_select_gc 完成 GC
初始化
  } else {
      ast_root->cgen(cout);
  }
}
```

- **内存分配**：对象创建（new 表达式）时，通过调用 Object.copy 方法复制对象原

型，并对接 GC 的内存分配接口（_gc_check）申请内存。

- **垃圾回收触发**：生成测试 GC 的代码（`emit_test_collector`），支持手动触发 GC 回收；同时，在内存分配时，通过 GC 接口自动检查内存是否充足，必要时触发回收。
- **赋值标记**：若开启 GC，在属性赋值时生成 GC 赋值标记代码（`emit_gc_assign`），确保 GC 能够正确跟踪对象的引用关系。赋值操作的内存访问需遵循 `emit.h` 中定义的对象偏移量规范，核心定义如下（来自 `emit.h`）：

```cpp
cpp
// 对象头偏移量定义，确保 GC 能够正确识别对象结构
#define TAG_OFFSET 0
#define SIZE_OFFSET 1
#define DISPTABLE_OFFSET 2
// 内存访问指令宏
#define SW    "\tsw\t"  // 存储指令（用于赋值）
#define LW    "\tlw\t"  // 加载指令（用于属性访问）
#define HEAP_START           "heap_start"  // 堆起始地址，GC 内存管理的基础
```

五、测试验证

5.1 测试环境

- 操作系统：Linux Ubuntu 20.04 LTS
- 编译器：GCC 9.4.0
- 模拟器：SPIM 8.0
- 测试工具：COOL 语言标准测试用例集、diff 工具（用于对比生成代码与标准输出）
- 构建工具：Makefile（项目提供的构建脚本），核心构建逻辑如下（来自 `Makefile`）：

```makefile
makefile
ASSN = 5
CLASS= cs143
CLASSDIR= ../..
LIB= -lfl

AR= gar
```

```
ARCHIVE_NEW= -cr
RANLIB= gar -qs

# 项目核心源文件列表，包含代码生成器核心逻辑与依赖模块
SRC= cgen.cc cgen.h cgen_supp.cc cool-tree.h cool-tree.handcode.h
emit.h example.cl README
CSRC= cgen-phase.cc utilities.cc stringtab.cc dumptype.cc tree.cc
cool-tree.cc ast-lex.cc ast-parse.cc handle_flags.cc
TSRC= mycoolc
CGEN=
HGEN=
LIBS= lexer parser semant

# 待编译的 C++文件列表
CFIL= cgen.cc cgen_supp.cc ${CSRC} ${CGEN}
OBJS= ${CFIL:.cc=.o}
OUTPUT= good.output bad.output

# 编译参数与头文件路径
CPPINCLUDE= -I. -I${CLASSDIR}/include/PA${ASSN} -
I${CLASSDIR}/src/PA${ASSN}
CFLAGS=-g -Wall -Wno-unused -Wno-write-strings -Wno-deprecated
${CPPINCLUDE} -DDEBUG

# 代码生成器目标文件构建规则
cgen:${OBJS} parser semant
    ${CC} ${CFLAGS} ${OBJS} ${LIB} -o cgen

# 测试执行规则
dotest:cgen example.cl
    @echo "\nRunning code generator on example.cl\n"
    -./mycoolc example.cl

# 清理规则
clean :
    -rm -f ${OUTPUT} *.s core ${OBJS} cgen parser semant lexer *~
*.a *.o
```

5.2 测试用例设计

为全面验证代码生成器的正确性，设计了覆盖所有核心功能的测试用例集，包括:

5.2.1 基础语法测试用例

- hello_world.cl：输出字符串“Hello, World!”，验证基础输出功能；
- int_arithmetic.cl：测试整数的加减乘除运算，验证算术表达式生成的正确性；
- bool_operation.cl：测试布尔值的逻辑运算与比较运算，验证比较表达式生成的正确性。

5.2.2 类与继承测试用例

- simple_class.cl：定义简单的用户类，测试类的创建与属性访问；
- inheritance.cl：测试类的继承关系，验证子类对父类属性与方法的继承；
- method_override.cl：测试方法重写，验证子类方法对父类方法的覆盖。

5.2.3 多态与方法分派测试用例

- dynamic_dispatch.cl：测试动态分派，验证不同对象调用同名方法时的正确跳转；
- static_dispatch.cl：测试静态分派，验证直接调用指定类方法的正确性。

5.2.4 复杂表达式测试用例

- cond_loop.cl：测试条件表达式与循环表达式，验证分支跳转与循环执行的正确性；
- typcase.cl：测试类型判断表达式，验证分支匹配与异常处理的正确性；
- let_block.cl：测试 let 表达式与块表达式，验证局部变量作用域与代码块执行的正确性。

5.2.5 边界情况测试用例

- empty_string.cl：测试空字符串的处理；
- zero_int.cl：测试零值整数的运算；
- deep_inheritance.cl：测试多层继承（如 A→B→C），验证继承关系的正确性。

5.3 测试结果与分析

本代码生成器通过了所有标准测试用例的验证，测试结果如下：

- 基础语法测试用例：所有用例均能正确生成汇编代码，在 SPIM 上运行后输出正确结果；
- 类与继承测试用例：子类能够正确继承父类的属性与方法，方法重写功能正常；

- 多态与方法分派测试用例：动态分派与静态分派均能正确跳转，多态功能正常；
- 复杂表达式测试用例：条件、循环、typcase、let 等表达式均能正确执行，逻辑无错误；
- 边界情况测试用例：空字符串、零值、多层继承等边界情况均能正确处理，无运行时错误。

测试过程中发现的问题及解决方案：

- 问题 1：动态分派时方法地址查找错误，导致程序崩溃。解决方案：修正方法偏移量的计算逻辑，确保从父类继承的方法偏移量正确累加。具体通过调试`build_methodtab()`方法中分派表的构建逻辑，结合`cool-tree.cc`中方法节点的复制与遍历代码，修正了方法偏移量的累加规则。
- 问题 2：typcase 表达式分支匹配顺序错误，子类分支被父类分支覆盖。解决方案：实现分支排序逻辑，按类标签从大到小排序（子类标签大于父类标签），确保子类分支优先匹配。排序逻辑的实现参考了`cool-tree.cc`中`cases`列表的处理方式，核心是对`branch_class`节点的类标签进行比较排序。
- 问题 3：对象初始化时属性值未正确赋值。解决方案：修正初始化代码中属性偏移量的计算，确保父类属性与子类属性的偏移量不重叠。通过核对`emit.h`中定义的对象头偏移量（`DEFAULT_OBJFIELDS`）与属性偏移量的累加规则，修正了`code_init()`方法中属性赋值的内存地址计算代码。
- 问题 4：字符串常量中特殊字符（如反斜杠、引号）处理错误，导致汇编代码语法错误。解决方案：优化`cgen_supp.cc`中`emit_string_constant`函数的特殊字符处理分支，补充了对反斜杠的转义逻辑，确保生成的`.ascii`指令中的字符串符合 MIPS 汇编规范。

六、优化与改进

6.1 已实现的优化

- **寄存器优化**：合理划分专用寄存器（ACC 存储结果、SELF 指向当前对象、T1-T3 存储临时值），减少寄存器的频繁保存与恢复，提高代码执行效率；
- **常量合并**：相同的常量在常量表中只存储一次，避免冗余的内存占用；
- **分支优化**：对条件表达式的分支跳转进行优化，减少不必要的跳转指令；
- **栈帧优化**：合理分配栈帧空间，只保存必要的寄存器，减少栈操作指令。

6.2 可改进方向

- **指令重排**：对生成的汇编指令进行重排，减少数据相关与控制相关，提高 MIPS 处理器的流水线效率；
- **常量折叠**：在代码生成阶段对编译期可知的常量表达式（如 1+2、3*4）进行计算，直接生成结果常量，减少运行时运算；
- **死代码消除**：识别并删除不会被执行的死代码（如条件恒为假的分支），减少代码体积；
- **GC 优化**：优化 GC 的调用时机，减少不必要的垃圾回收触发，提高程序运行效率；
- **支持更多优化选项**：提供优化级别参数（如-O0、-O1、-O2），允许用户根据需求选择不同的优化策略。

七、总结

本项目成功实现了一个完整、正确、高效的 COOL 语言代码生成器，完成了从 AST 到 MIPS 汇编代码的转换，支持 COOL 语言的所有核心功能。通过模块化设计与访问者模式，确保了代码的可维护性与可扩展性；通过严格的测试验证，保障了生成代码的正确性。

本项目的核心收获包括：

- 深入理解了编译器后端的工作原理，掌握了代码生成的核心技术与流程；
- 熟悉了 MIPS 汇编指令集的特点与应用，掌握了汇编代码的生成与优化方法；
- 理解了面向对象语言的实现机制，包括类继承、多态、动态分派等核心特性的底层实现；
- 掌握了内存管理与垃圾回收的基本原理，实现了与 GC 模块的对接；
- 提升了模块化设计与问题排查能力，能够快速定位并解决代码生成过程中的错误。
- **架构扩展**：支持更多架构的汇编代码生成（如 x86、ARM），提高代码生成器的可移植性；
- **调试支持**：生成调试信息（如 DWARF 格式），支持在 SPIM 或其他调试器中对 COOL 程序进行调试；
- **集成优化**：将代码生成器与前端（词法分析、语法分析、语义分析）集成，形成完整的 COOL 编译器，并实现端到端的优化。