摘 要

本 SYSY 编译器是一个将 C 语言子集（SYSY）源代码转换为 RISC-V 指令集的编译工具。它采用 Flex 和 Bison++进行词法和语法分析，通过遍历抽象语法树（AST）生成类 LLVM 中间代码并划分基本块转化为控制流图。接着，编译器使用 Mem2reg 过程将内存操作转化为寄存器操作，生成静态单赋值（SSA）形式的中间表示（IR），并进行活跃性分析、插入 phi 函数、死代码消除、部分冗余消除、CFG 基本块的简化、循环优化、内联优化、伪递归消除、消除不重要边、数组相关的优化等。最终，将中端传递的 IR 进行模式匹配，并通过寄存器分配和其他后端优化，生成最终的 RISC-V 汇编代码。

关键词：SYSY 语言、LLVM、中间代码优化、指令选择、寄存器分配、RISC-V

目 录

第一章 综设目标：实现对 SYSY 语言的编译器

1.1 SYSY 语言简介

SysY 语言由 C 语言的一个子集扩展而成。程序的源码存储在一个扩展名为 sy 的文件中。该文件中有且仅有一个名为 main 的主函数定义，还可以包 含若干全局变量声明、常量声明和其他函数定义。SysY 语言支持 int/float

类型和元素为 int/float 类型且按行优先存储的多维数组类型，其中 int 型整数为 32 位有符号数；float 为 32 位单精度浮点 数；const 修饰符用于声 明常量。SysY 支持 int 和 float 之间的隐式类型，但是无显式的强制类型转 化支持。其 I/O 是以运行时库方式提供，库函数可以在 SysY 程序中的函数内 调用。关于在 SYSY 程序中可用的库函数，请参见 SYSY 运行时库文档。

- 函数：

SYSY 支持带参数和不带参数的函数，参数类型可以是 int 类型或数组类型。函数可以返回 int 类型的值，或声明为 void 类型表示不返回值。对 于 int 类型的参数，采用按值传递；对于数组类型的参数，传递的是数 组的起始地址，并且在形参中，只有数组的第一维长度可以省略。函数 体内包含变量声明和一系列语句。

- 变量与常量声明：

在 SYSY 中，可以在一条声明语句中声明多个变量或常量，并且可以在声明时为它们提供初始化表达式。所有变量和常量必须先声明后使用。全 局变量/常量在函数外声明，局部变量/常量在函数内声明。

- 语句：

SYSY 支持多种类型的语句，包括赋值语句、表达式语句（表达式可以为 空）、语句块、if 语句、while 语句、break 语句、continue 语句和

return 语句。语句块可以包含多个变量声明和语句。

- 表达式：

SYSY 支持基本的算术运算（+、-、*、/、%）、关系运算（==、!=、

<、>、<=、>=）和逻辑运算（ !、&&、 ||）。在 SYSY 中，非零值表示 真，零值表示假。关系运算和逻辑运算的结果是 1 表示真，0 表示假。 算符的优先级、结合性以及计算规则（包括逻辑运算的“短路计算 ”） 与 C 语言一致。

- 主函数与全局声明：

SYSY 支持主函数的定义，并且可以包含多个全局变量声明、常量声明以 及

其他函数定义。SYSY 语言支持 int 类型和元素类型为 int 的多维数组 类型，数组元素按行优先存储。int 类型的整数是 32 位有符号数，

const 修饰符用于声明常量。

- 如图（图 1.1-1）为 SYSY 文法的结构梳理：

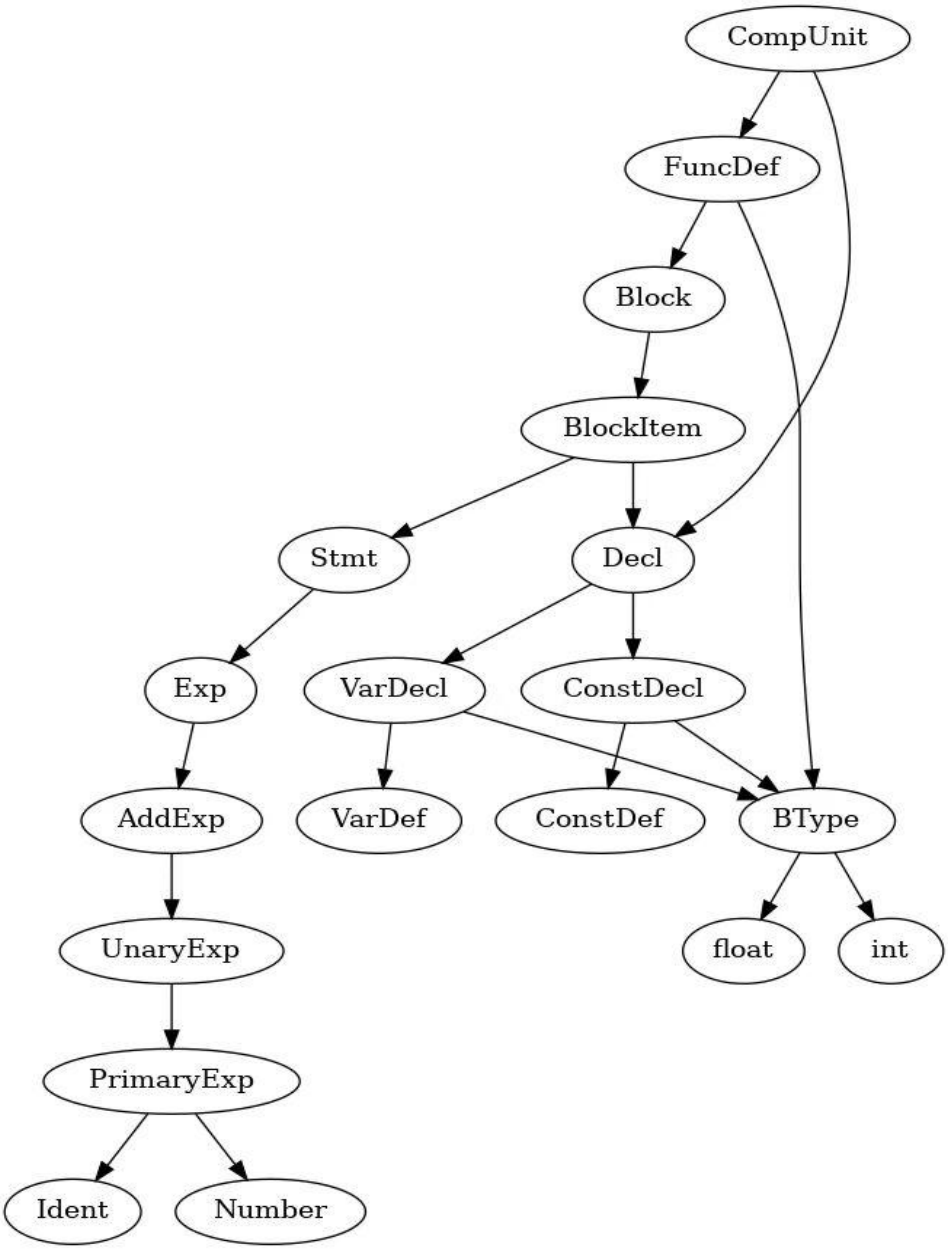


图 1. 1-1

1.2 实现平台与运行平台简介

编译在 Linux 环境下进行，采用 RISC-V 架构，所有文件均使用 UTF-8 编码。 词法分析采用 Flex 工具实现，语法分析则使用 Bison++工具。

CMake 构建工具进行编译管理。

采用 C++语言实现。

1.3 针对复杂工程问题的方案实现

本项目的最终目标是实现一个能够将类似 C 语言的源代码转换为汇编代码 RISC-V 的编译器。这将有助于我们更好地理解编译过程，同时提升我们的编程和团队合作能力。通过使用 GitHub 平台进行代码版本管理和协同工作，能够确保代码的版本控制和团队协作的顺畅。这种合作方式不仅提升了个人技能，也加速了整个项目的开发进程。

第二章 需求分析及实现方法

2.1 需求分析

2.1.1 项目背景

本项目旨在设计并实现一个面向 SysY 语言的高效、稳定的编译器。SysY 是一种为教学与实验目的而设计的简化版 C 语言子集，语法结构清晰，语义规则相对简单，非常适合用作编译原理课程的实验语言。通过构建这个编译器，项目希望实现对 SysY 源程序的完整处理能力，最终能够将其准确地翻译为 RISC-V 架构下可运行的汇编代码，从而为学习者提供一个具备实际意义的编译实践平台，也为编译技术的深入理解打下基础。

2.1.2 引言

本报告的目的是系统性地阐述 SysY 编译器的功能需求与设计方向，帮助项目相关人员更全面地了解该编译器的构建目标与整体架构。通过对核心功能的梳理与分析，报告为开发过程中的协作、测试与管理提供统一的参考依据，确保各阶段工作的协调推进，为编译器的顺利实现奠定基础。

2.1.3 编写目的

本报告的编写旨在明确 SysY 编译器在功能上的基本需求与实现范围，为后续的开发工作提供清晰的方向指引。通过对需求的系统整理，开发团队能够在统一理解的基础上高效协作，从而为项目的有序推进和最终落地提供坚实的基础支撑。

2.1.4 技术挑战

在编译器的开发过程中，性能与稳定性始终是核心考量。这不仅依赖于所选

编程语言本身的特性，也受到编译优化策略和底层实现方式的直接影响。若未合理设计，可能导致生成代码效率低下或系统在大规模输入下不稳定。此外，为了提升编译器的适用性与推广价值，项目需要实现跨平台支持，能够在 Linux、Windows 等主流操作系统上生成正确的目标代码。这一需求不仅增加了调试与适配的复杂性，也带来了诸如文件路径处理差异、系统调用接口不一致、汇编器行为不同等平台相关的技术挑战。如何在保证统一语义的前提下兼顾不同平台的兼容性，是本项目面临的重要技术难题之一。

2.1.5 文档约定

- 正文风格：宋体、小四
- 一级标题：黑体、小三
- 二级标题：黑体、四号
- 三级标题：宋体、小四（加粗）

2.1.6 预期读者和阅读建议

本报告面向的读者主要包括开发人员、测试人员以及项目管理人员。对于开发人员而言，理解报告中所提出的功能需求、设计规范以及系统架构至关重要，这将直接关系到编译器各模块的正确实现与整体系统的一致性。测试人员则应重点关注功能需求的覆盖情况、性能目标的达成程度以及具体的测试用例设计，从而保障编译器在不同输入和运行环境下的稳定性与可靠性。而对于项目管理人员来说，全面了解本项目的开发范围、时间安排以及潜在风险，是制定合理计划和动态监控项目进展的基础。不同角色的读者应结合自身职责，聚焦相关内容，以实现高效协同与项目顺利推进。

2.1.7 综合描述

SysY 编译器是一款面向教学用途的编译工具，负责将 SysY 语言编写的源代码逐步转换为中间表示形式，并最终生成能够在目标平台上运行的汇编代码或机器代码。作为编译原理课程的实践项目，它不仅承担语法分析、语义检查和中间代码生成等基本功能，还力求在

性能表现、系统结构以及平台适应性方面达到较高标准。为了更好地服务于多样化的开发场景，编译器的设计强调模块化和可扩展性，便于在未来集成更多优化技术或支持更复杂的语言特性。同时，系统也注重跨平台移植能力，确保在 Linux、Windows 等主流操作系统上均能稳定运行，从而为不同学习和应用环境下的编译实践提供统一、可靠的技术支持。

2.1.8 系统概述

SysY 编译器整体结构采用传统的前端-中端-后端架构，功能涵盖从源代码接收到目标代码生成的完整流程。系统首先接收以 SysY 语言编写的源程序文件作为输入，通过前端模块依次完成词法分析、语法分析和语义分析，从而构建出程序的语法树及其对应的中间表示（IR）。在前端处理的基础上，系统还会进一步将程序结构转化为控制流图（CFG），为后续优化打下基础。

中端模块承担着优化和转换的核心任务。它会将初始 IR 转化为静态单赋值形式（SSA），以便更高效地执行常见优化技术，如常量传播、死代码消除和控制流简化等，从而提升程序在运行时的执行效率。此外，中端还可能实现基础的数据流分析，为后端的指令生成提供支持信息。

在完成中间优化后，后端模块负责将优化后的中间代码翻译为 RISC-V 架构下的汇编指令。这一阶段不仅包括基本的指令选择和映射，还涉及寄存器分配、指令调度以及目标代码优化等关键步骤，以确保最终生成的汇编代码在执行效率和资源利用方面达到较优水平。整体上，系统各模块紧密协作，共同实现从高层语言到低层机器指令的准确、高效转换。

2.1.9 系统非功能需求

在设计 SysY 编译器时，除了满足基本的功能需求外，系统的非功能需求同样至关重要。性能方面，编译器应能在合理的时间内完成对源代码的处理，尤其是在面对较大规模的代码时，依然要保证足够的编译效率，以满足教学和实验的实际需求。代码的可维护性也是设计的重要考虑因素，系统结构需要保持清晰和模块化，这不仅有助于团队成员之间的协作，也方便后续功能的扩展与问题的调试。为了增强系统的适用性，兼容性要求编译器能够在主流的操作系统环境中稳定运行，包括但不限于 Linux 和 Windows，确保不同平台的开发者均能顺利使

用。最后，可靠性方面，编译器必须具备完善的错误检测与处理机制，能够准确识别非法输入并给予合理反馈，避免程序因异常情况崩溃，从而保障整体系统的稳定性和用户体验。

2.1.10 假设和约束

本系统的设计和开发基于若干前提假设和约束条件。首先，编译器需运行于支持 LLVM 生态系统的操作系统上，典型代表包括 Linux 和 Windows，这为系统的底层功能提供了兼容性保障。与此同时，开发团队需要具备对 LLVM 编译器架构及其相关 API 的深入理解，这对于后续的功能扩展和定制开发至关重要，因为只有充分掌握这些技术细节，才能有效地利用 LLVM 提供的强大功能，实现高效且灵活的编译流程。另一方面，编译器的性能和稳定性在很大程度上受所选编程语言以及具体的编译优化算法影响，因此在开发过程中必须谨慎权衡各种实现方案，确保最终系统既高效又可靠。最后，系统设计强调高度的可移植性，这不仅使编译器能够适应多种平台的运行环境，也为未来的扩展和部署提供了便利，保证项目能够灵活应对不断变化的技术需求和应用场景。

2.2 实现方法

2.2.1 具体实现流程

本编译器的实现过程遵循经典的编译器设计思路，整体分为多个阶段以保证系统的模块化和可维护性。实现采用以下流程：

前端部分：

在前端部分，通过使用 Flex 工具对源代码进行词法分析，将连续的字符序列转换成具有语义的词法单元（Token），为后续的语法处理奠定基础。紧接着，设计并维护符号表，用于存储程序中标识符的名称及其属性信息，这为语义分析和代码生成提供必要的数据支持。随后，利用 Bison++ 工具完成语法分析，解析程序的结构并构建出抽象语法树（AST），这是程序语义的直观表达形式。语义分析阶段对 AST 进行进一步检查和验证，确保程序符合语言的语义规则，并捕

获潜在的错误。基于符号表和抽象语法树，编译器接着生成类似 LLVM 风格的中间代码，这种中间代码便于进行后续的优化和目标代码生成。为了更好地管理程序流程，系统设计了核心类，如 User、Use 和 Value，这些类帮助表达和管理中间代码中的各种操作及其依赖关系。最后，通过划分基本块并生成控制流图（CFG），编译器建立起程序的流程控制结构，为中端优化阶段提供坚实基础，确保后续的代码转换和优化工作能够高效且准确地展开。

中端部分：

在编译器的中端阶段，系统重点聚焦于程序的优化与中间代码的改进，以提升生成代码的执行效率和资源利用率。首先，编译器通过分析控制流图，构建支配树，准确识别程序中各基本块之间的支配关系，这为后续的各种数据流分析和优化奠定了基础。借助 mem2reg 优化技术，系统插入 Phi 函数，完成变量的重命名过程，使中间代码转换为标准的静态单赋值（SSA）形式。SSA 形式不仅简化了变量的跟踪，也极大地促进了常量传播和数据流分析等优化手段的实施。基于此，编译器进行一系列标量优化操作，包括死代码消除和公共子表达式消除，目的是剔除冗余计算和无用代码，提升执行效率。同时，还实施部分冗余消除（SSAPRE）和基本块的简化等更深入的优化措施，以进一步精简程序结构和优化性能。针对程序中循环结构的特点，中端优化模块还包含丰富的循环优化技术，如循环分析、循环简化、循环旋转、循环展开、循环删除以及循环并行等，这些优化手段有效地提升了循环执行效率，减少了不必要的开销，为最终生成高效的目标代码提供坚实的保障。

后端部分：

在编译器的后端阶段，系统的主要任务是将优化后的中间代码转换为能够直接在目标硬件上运行的机器指令。首先，指令选择模块通过模式匹配技术，根据不同的操作类型和语句结构，选择最合适的目标指令序列，从而实现高效的指令映射。随后，寄存器分配环节采用图着色算法，将中间代码中使用的虚拟寄存器合理映射到有限的物理寄存器上，以减少内存访问，提高执行速度。为了进一步提升生成代码的质量，后端还会执行一系列优化措施，包括 Phi 函数的消除、代码的简化以及对 SIMD 指令的支持和优化，这些步骤能够有效减少冗余指令并

增强并行计算能力。最终，后端模块通过遍历基本块，生成符合目标架构规范的机器码，完成从高级语言到底层硬件指令的完整转换，确保编译器输出的代码既正确又高效。

本方案编译器实现具体流程如图所示（图 2.2.1-1）：

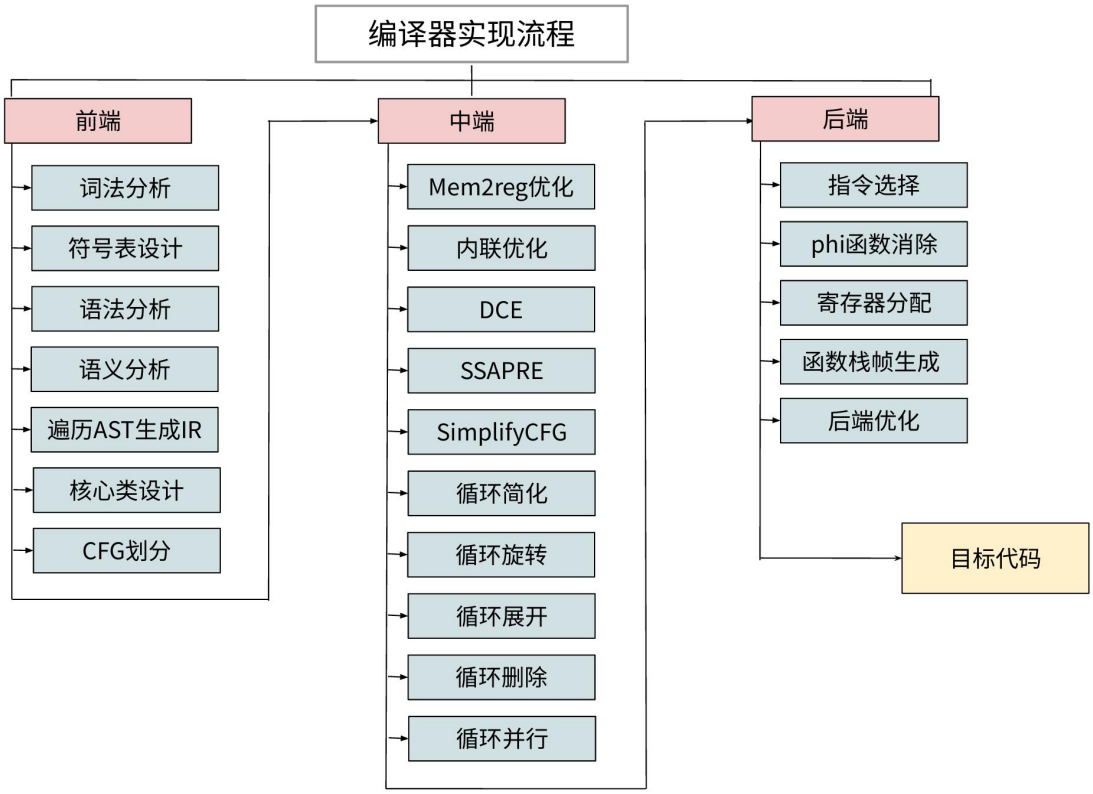


图2.2.1-1

2.2.2 小组合作方式

本项目的团队合作方式注重高效的协作与沟通，确保编译器开发过程有条不紊。首先，团队统一使用 GitHub 作为代码托管平台，借助其完善的版本控制和协作功能，实现代码的共享与管理。通过这一平台，团队成员能够及时同步更新、解决冲突，并记录开发历程，保障项目的透明度和可追溯性。为了促进沟通交流，团队定期召开会议，汇报进展、讨论技术难题，及时协调资源与解决方案，保证问题得到快速响应和处理。

在日常工作中，团队遵循严格的代码提交规范，制定明确的任务分配机制，确保每位成员根据自身技术专长和兴趣承担相应模块的开发责任。这种分工不仅提高了工作效率，也促进了成员间的协调配合，使得各个功能模块能够顺利集成。为了

保证代码质量，团队定期开展代码审查，利用 GitHub 的 Pull Request 功能对每次代码变更进行详细讨论和评审。这样的过程不仅增强了代码的可维护性，也促进了知识的共享与传递。

此外，团队重视技术培训与经验交流，定期组织分享会和学习活动，帮助成员掌握新技术、巩固已有技能，推动整体技术水平的提升。通过持续的知识共享，团队形成了积极进取、协同合作的良好氛围。任务分配方面，团队注重灵活调整，结合项目进度和成员状态进行合理分工，确保每个模块都由最合适的人员负责，从而最大限度地提升合作效率和项目质量。整体而言，这种系统化、规范化的协作方式为项目的顺利推进提供了坚实保障。

第三章 实现过程

3.1 源文件的输入

```
// 该代码仅仅只做测试相关 void swap(int a,int b)
{
int temp=a; a=b;
b=temp;
}
int a=110;
int main(){
int x=111;
int y=222;
swap(x,y);
return 0;
}
```

代码 3.1-1 输入的基础源文件代码

3.2 输入预处理

词法分析是编译器前端的第一步，其主要任务是将源代码中的字符序列转换成有意义的词法单元（token）。在本项目中，我们采用了 Flex（Fast Lexical Analyzer）这一广泛使用的工具来实现词法分析器。Flex 通过定义正则表达式模式，自动生成能够识别和匹配输入源代码中各类单词的分析代码，从而大大简化了词法分析器的开发工作。具体而言，词法分析器的功能主要体现在三个方面：首先，它能够准确标记每个单词的属性类型，区分关键字、标识符、常量、运算符等不同类别；其次，系统将输入的字符流转换成一系列<单词类型，单词内容>形式的二元组，作为后续语法分析的基础输入；最后，词法分析器还具备统计功能，能够统计并输出每种类型单词的数量，帮助开发和测试人员直观了解程序的词法结构。通过这些功能的实现，词法分析模块为整个编译过程提供了可靠的输入保障。

3.2.1 Flex 文件设计

首先，我们定义 SysY 语言中的关键字、操作符、标识符、常数等元素， 并使用正则表达式描述它们的模式，如代码 3.2.1-1。

```
    // 代码仅为节选实现 {Decimal}
    { yylval.iVal = strtol(yytext, nullptr, 0); current.line =
    yylineno; current.col = yylineno - yyleng + 1; return num_INT; CurCol
+= yyleng; }
    {nDecimal}
    { yylval.iVal = strtol(yytext, nullptr, 0); current.line =
    yylineno; current.col = yylineno - yyleng + 1; return num_INT; CurCol
+= yyleng; }
    {Float}
    { yylval.fVal = strtol(yytext, nullptr, 0); current.line =
    yylineno; current.col = yylineno - yyleng + 1; return num_FLOAT; CurCol
+= yyleng ; }
    {nFloat}
    { yylval.fVal = strtol(yytext, nullptr, 0); current.line =
    yylineno; current.col = yylineno - yyleng + 1; return num_FLOAT; CurCol
+= yyleng; }
```

代码 3.2.1-1 部分正则表达式演示代码

3.2.2 解释和流程

关键字：如 int, return, while, if 等，通过正则表达式 "int" 来匹配。

标识符：由字母和数字组成，[a-zA-Z_][a-zA-Z_0-9]* 表示。

常量：包括整数常量，正则表达式 [0-9]+ 匹配。

操作符：例如 =, ==, +, -, *, / 等。

注释：单行注释 // 和多行注释 /* ... */，注释内容被忽略。

Flex 会根据这些规则将源代码转化为不同的 token(如标识符、整数、关 键字、操作符等)。然后通过 print_token() 函数输出每个 token 的类型和 对应的文本。

为了验证词法分析模块的正确性，选取一段简单的测试用例代码进行分析。经过词法分析器处理后，源代码被分解成一系列结构化的 Token 序列，每个 Token 都包含其类型和对应的词素文本。例如，对于该测试代码，词法分析器依次识别出了整型关键字 <T_INT,"int">、标识符 <T_IDENTIFIER,"main">、左括号 <T_LPAREN, "("> 等基本组成部分。该序列准确反映了源代码的语法结构

和词法组成，从声明函数到变量定义、赋值，再到函数调用及返回语句，每个词素都被正确分类。测试用例代码经过词法分析后产生如下的Token序列，如代码3.2.2-1：

```
 <T_INT, "int">
 <T_IDENTIFIER,  "main">
 <T_LPAREN, "(">
 <T_RPAREN, ")">
 <T_LBRACE, "{">
 <T_INT, "int">
 <T_IDENTIFIER, "x">
 <T_ASSIGN, "=">
 <T_CONST, "111">
 <T_SEMICOLON, ";">
 <T_INT, "int">
 <T_IDENTIFIER, "y">
 <T_ASSIGN, "=">
 <T_CONST, "222">
 <T_SEMICOLON, ";">
 <T_IDENTIFIER,  "swap">
 <T_LPAREN, "(">
 <T_IDENTIFIER,  "x">
 <T_COMMA, ",">
 <T_IDENTIFIER,  "y">
 <T_RPAREN, ")">
 <T_SEMICOLON, ";">
 <T_RETURN,  "return">
 <T_CONST, "0">
<T_SEMICOLON, ";"> <T_RBRACE, "}">
```

代码 3.2.2-1 输出的Token 序列

通过该 Token 序列的输出，可以清晰看到词法分析器如何将原始代码中的字符流转换成可供后续语法分析处理的标准化单元。这不仅验证了词法规则的完整性和匹配的准确性，也为语法分析阶段的正确进行奠定了基础。此外，对每个Token 进行类型标注，有助于编译器在后续的语义分析和错误检测中实现精确定位和处理，从而提高整体编译流程的鲁棒性。

3.2.3 语法分析说明

语法分析（Parsing）是编译过程中的一个重要环节，指的是根据语言的文 法规则对词法分析器输出的单词序列进行分析，并构建出源代码的语法结构。

语法分析的核心任务是检查输入的代码是否符合语言的语法规则，并生成一个 语法结构（通常是语法分析树或抽象语法树，AST）。语法分析器通常依赖于词 法分析器提供的单词流，将其作为输入来构建语法树。

语法分析器的主要工作是按照文法的产生式规则，通过自上而下（Top-

Down）或自下而上（Bottom-Up）的方法，检查输入的单词序列是否能够匹配文 法规则。如果匹配成功，生成相应的语法树；如果匹配失败，则表示输入的代 码不符合语法规则。自上而下的分析方法是一种试探性的方法，从文法的开始 符号出发，尝试通过不同的产生式来匹配输入串。

3.2.4 Bison++使用

Bison++ 是一种通用的解析器生成工具，它能够将注释清晰的上下文无关文法（CFG）转换为基于 LALR(1) 解析器表的高效解析器。生成的解析器能够对输入的词法单元序列进行语法分析，并构建对应的抽象语法树（AST），为后续的语义分析和代码生成提供结构化的数据基础。在 Bison++ 中，文法规则的定义主要包含几个部分：首先是关键的 token 声明，通常使用 %token 来定义基本的词法单元，比如整数、标识符等；其次是非终结符的声明，通过 %nterm 关键字标识语法中的非终结符符号；最后是规则与语法动作的定义部分，文法规则与相应的动作代码通过 %% 进行分隔。动作代码在解析过程中触发，用于根据匹配的规则执行特定操作，例如构造 AST 节点，从而将语法结构具体化为程序内部可操作的数据形式。通过这种方式，Bison++ 不仅实现了文法的形式化表达，也为编译器的语法分析模块提供了强大支持。

3.2.5 Bison++部分代码说明

```
%token Y_INT Y_FLOAT Y_VOID Y_CONST Y_IF Y_ELSE Y_WHILE Y_BREAK
Y_CONTINUE Y_RETURN
%token  Y_EQ  Y_NOTEQ  Y_GREAT  Y_LESS  Y_GREATEQ  Y_LESSEQ  Y_AND
Y_OR %token <StrVal> Y_ID _string
%token <iVal> num_INT
%token            <fVal>
num_FLOAT %type <TypeVal>
Type

%type <AstVal> CompUnit CompUnitt FuncDef Block BlockItems
BlockItem Stmt FuncParam Decl ConstDecl VarDecl VarDecll VarDeff
VarDef LVal Exp UnaryExp PrimaryExp Number UnaryOp AddExp MulExp
RelExp EqExp LAndExp LOrExp FuncParams FuncParamm CallParams Cond
String ConstDecll ConstDeff ConstDef ConstExp
%type <InitVal> InitVal InitVals ConstInitVal ConstInitVall


%start
CompUnit %%
```

```
CompUnit: CompUnitt {
auto node = new CompUnitAST;
node->comp_unit = $1;
node->position.line = current.line;
node->position.col = current.col;

ast = move(node); }
```

代码 3.2.5-1 Bison++代码

在 Bison++ 中，%token 用于定义终结符，也就是词法分析器所识别的最基本的语言单元。这些终结符代表了语言的核心元素，包括关键字、操作符、标识符等。比如，Y_INT、Y_FLOAT 和 Y_VOID 等表示 C 语言中的基本关键字如 int、float 和 void，它们通常由词法分析器在扫描源代码时识别生成。类似地，Y_EQ、Y_NOTEQ、Y_GREAT 和 Y_LESS 等符号代表各种比较操作符，比如等于、不等于、大于和小于等。Y_ID 代表程序中的标识符，例如变量名和函数名，而 _string 则表示字符串常量。常量类型还包括整数常量 num_INT 和浮点数常量 num_FLOAT，它们分别对应不同的数据类型。见代码 3.2.5-1。

除了简单的标记，%token 声明还可以附带类型信息，方便在语法分析过程中传递数据。例如，Y_ID 使用 <StrVal> 类型，表明它携带的是字符串类型的信息，num_INT 和 num_FLOAT 则分别使用 <iVal> 和 <fVal> 来表示整数和浮点数的值。这种类型标注机制为语法规则中的语义动作提供了数据支持，使得编译器能够在解析过程中携带和处理具体的值。

语法规则部分则通过 %% 进行分隔，定义了语言的句法结构及其对应的语法动作。每条规则的右侧描述了规则的具体构成，当输入符合该规则时，语法动作代码会被执行，通常用来构建抽象语法树的节点或者进行语义检查，从而将输入的语言结构转化为编译器内部可以处理的数据结构。这种机制使得 Bison++ 不仅能够验证代码的语法正确性，还能辅助后续的编译流程实现。

在语法规则的解析中，左侧的 CompUnit: CompUnitt 表示 CompUnit 这一非终结符是由 CompUnitt 组成的。作为整个语法分析的顶层规则，CompUnit 代表着整个程序或者一个完整的编译单元，它是语法树构建的起点。右侧的 CompUnitt 则具体定义了构成 CompUnit 的基本组成元素，通常包括声明、函数定义等代码结构。

在该规则对应的动作代码部分，首先通过 auto node = new CompUnitAST; 创建了一个新的抽象语法树（AST）节点对象 CompUnitAST，该节点用于表示顶层的编译单元结构。随后，node->comp_unit = $1; 将语法规则右侧匹配到的

CompUnitt 内容赋值给该节点的成员变量 comp_unit，实现语法树节点之间的关联和层级构建。为了方便错误定位和调试，动作代码中还为该 AST 节点记录了源代码中对应的位置信息，包括当前处理的行号和列号，分别存储于 node->position.line 和 node->position.col。最后，通过 ast = move(node); 语句将构造好的节点 node 移交给整体抽象语法树 ast，完成顶层节点的生成。这样的设计不仅保证了语法结构的准确表达，也为后续的语义分析和代码生成阶段提供了清晰的语法基础。

3.2.6 AST 结构

抽象语法树（AST）是一种高层次的、与具体编程语言无关的数据结构，它 捕捉了程序的结构和语义信息。AST 可以用来进行后续的分析和优化，如语法 检查、类型检查、代码优化等。在编译过程中，AST 通常是生成中间代码的基 础。

在实验中，AST 节点采用多叉树结构，通过基类指针存储。每个 AST 节点 代表程序中的一个语法结构，比如一个声明、一个赋值语句、一条控制语句等。在生成中间代码时，我们可以通过递归遍历 AST 来生成目标代码。例如，CompUnit 和 FuncDef 类型的节点会生成相应的中间代码；Stmt 类型的节点会生成语句执行的代码。

图示（代码 3.2.6-1）的 AST 结构展示了 CompUnit 作为语法分析的起始点，如何逐步构建出语法树，并根据需要生成中间代码:

```
CompUnit
    ├── Decl (int a = 110)
    ├── FuncDef (main)
    │   └── Block
    │   ├── Stmt (int x = 111)
    │   ├── Stmt (int y = 222)
    │   ├── Stmt (swap(x, y))
    │   └── Stmt (return 0)
```

代码 3.2.6-1 AST结构

测试用例通过语法分析后，产生了如代码3.2.6-2 所示抽象语法树。

```
CompUnitAST
```

```
{ FuncDefAST {
,  swap,  FuncFParamsAST
{ FuncFParamAST {
Btype:INT, a
}(FuncFParamAST  ends)  FuncFParamAST
{ Btype:INT, b
}(FuncFParamAST ends)
}(FuncFParamsAST ends) BlockAST {
BlockItemAST
{ DeclAST {
VarDeclAst {
vartype:INTtemp = { {ExpAST {
LOrExpAST {
LAndExpAST {
EqExpAST {
RelExpAST {
AddExpAST {
MulExpAST {
UnaryExpAST {
PrimaryExpAST
{ LValAST {
a
}(LValAST ends)
}(PrimaryExpAST ends)
}(UnaryExpAST ends)
}(MulExpAST ends)
}(AddExpAST ends)
}(RelExpAST ends)
}(EqExpAST ends)
}(LAndExpAST ends)
}(LOrExpAST
ends)      }(ExpAST
ends) }}  }(VarDecl
AST ends)
}(DeclAST ends)
```

代码 3.2.6-2 部分结果展示

3.2.7 符号表设计

符号表作为编译器中的核心数据结构，承担着对程序中所有标识符信息的系统管理与维护。在本项目中，我们采用了高效的哈希表实现符号表，这种结构能够快速完成标识符的查找和插入操作，从而满足编译过程中对符号频繁访问的需求。符号表中的每个条目都由标识符的名称与其对应的属性域组成，这些属性包括变量或函数的具体数据类型、作用域范围、内存地址以及其它相关的语义信息。通过将源代码中的每个标识符与其声明和使用的详细信息绑定，符号表实现了对程序语义信息的集中管理。在整个编译过程中，符号表不断被更新和查询，收集源程序中各个语法符号的类型特征及其作用环境，保证了语义分析的准确性和完整性。这样的设计不仅有助于语义检查的顺利进行，还为

后续的代码生成和优化环节提供了必要的支持，确保生成的目标代码能够正确反映程序的语义意图。

符号表在编译器设计中扮演着至关重要的角色，其主要功能体现在对程序中符号信息的系统管理。首先，符号表负责记录程序中所有变量的名称，并收集与之相关的详细属性信息，包括变量的存储分配位置、数据类型、作用域范围等。此外，对于函数符号，符号表还维护参数的数量、类型、传递方式以及函数的返回类型等关键信息，这些数据为后续的语义分析和代码生成提供了基础支持。其次，符号表在编译过程中被频繁查找，以验证符号的合法性，这对于上下文语义的正确性检查至关重要，确保变量或函数的使用符合语言规则，避免出现未定义或类型不匹配等错误。最后，在目标代码生成阶段，符号表中保存的地址分配信息为寄存器分配和内存布局提供依据，确保生成的代码能够正确访问和操作相应的数据。综上所述，符号表不仅是语义分析的核心组件，也是实现高效目标代码生成的关键基础。

符号表在编译器的各个阶段都发挥着重要作用，承担着不同的职责。在词法分析阶段，符号表主要用于记录标识符的基本信息，比如名称和在源代码中的位置，这一阶段通常只需要将尚未出现的标识符插入符号表中，信息相对简单。进入语法分析阶段，符号表的作用进一步强化，编译器会查询符号表以确保每个标识符都符合语法规则，特别是检查它们是否已经被正确声明，这就要求符号表能够支持高效的查找操作以满足分析需求。

到了语义分析阶段，符号表的功能更加复杂。除了插入新的变量或函数标识符外，还需要详细记录它们的类型信息、作用域范围、初始化状态等属性，尤其要能处理嵌套的作用域结构，如函数内部或代码块中的变量定义。这些详细的语义信息不仅确保标识符的正确使用，还为类型检查、作用域管理和语义

在编译器实现中，符号表除了要具备高效的查找和插入能力，还需要兼顾作用域的管理。为了处理程序中的作用域嵌套问题，本项目设计了一套基于栈结构的作用域管理机制。在进入新的作用域时，会将一个新的符号表压入栈中，用于记录该层作用域中的所有标识符。当离开该作用域时，借助记录的栈顶指针，符号表管理器能够准确定位并将该层的符号表出栈，从而清除当前作用域的所有标识符信息，确保符号不会被误用。这种机制通过栈结构与作用域一一对应，使作用域的进入与退出具备良好的可控性和一致性。

从数据结构的角度看，符号表内部采用哈希表实现，这种结构因其极高的查找与插入效率而广泛应用于编译器设计中。在哈希表中，标识符的名称作为键，对应的值则是包含该标识符各类属性的信息结构体。为了支持作用域嵌套，我们采用了分层符号表的设计方式。每个作用域对应一个独立的哈希表，通过指针与上一层作用域相连接，从而构成一个作用域层级结构。在这种设计中，全局符号表负责管理全局变量和函数声明，而函数体或语句块中的局部变量则保存在其对应的局部符号表中，两者协同运作，既保证作用域隔离，又支持符号的逐级查找。

符号表中存储的内容不仅包含标识符的基本信息，例如其名称、所属作用域以及定义的位置，还包括更详细的语义属性。例如，对于变量而言，需要记录其数据类型（如 int、float）、是否已初始化、分配在内存还是寄存器中的地址信息，以及在数组类型时的维度、大小等补充数据。而对于函数符号，则需进一步存储其返回值类型、参数列表（包括参数类型、顺序和是否具备默认值）、是否为内联函数以及是否具有外部链接属性等。通过这些信息的完整管理，符号表不仅为语义分析提供精准支持，也为后续的中间代码生成与优化阶段提供了丰富的数据依据。

3.2.8 本项目具体实现的符号表示例

类 Symbol：

本项目设计的符号表类 Symbol 如下（代码 3.2.8-1）：

```
class Symbol {
friend class SymbolTable;
bool GlobalFlag;
public:
int TempId;
SymType SymbolType;
bool ConstFlag;
bool MemoryFlag;
VarInfo VarArrtributes;
ArrayInfo* ArrAttributes;
FunctionInfo* FunctionAttributes;
Symbol(bool IsConst, bool IsInMemory, int val)
: SymbolType(SymType::INT32), ConstFlag(IsConst),
MemoryFlag(IsInMemory), VarArrtributes(val)
{
FunctionAttributes       =
NULL; }
//省略了整型逻辑
Symbol(bool IsConst, std::vector<float> dim, InitValTree<float>*
initval = NULL, InitValTree<BaseAST*>* initexp = NULL)
: SymbolType(SymType::Float32Array), ConstFlag(IsConst),
```

```
 MemoryFlag(true)
 {
 if (initexp != NULL) {
 ArrAttributes = new ArrayInfo(dim, initexp); }
 else {
 ArrAttributes = new ArrayInfo(dim, initval); }
 FunctionAttributes       =
 NULL; }
 Symbol(ValueType ret, std::vector<ArgsType> args)
 :   SymbolType(SymType::FuncName)
 { ArrAttributes = NULL;
 FunctionAttributes = new FunctionInfo(ret, args); }

bool GetGlobalFlag() const;
std::string GetLabelStr(std::string SymName) const; ……};
```

代码 3.2.8-1 符号表设计 I：类 Symbol

Symbol 类是用于描述和管理编译过程中符号信息的重要数据结构，其设计涵盖了对标量、数组和函数三类语言实体的支持，并能区分全局与局部符号，满足中间代码生成和语义分析的多样化需求。在成员变量方面，该类包含了 GlobalFlag 字段，它是一个布尔类型的私有变量，用于标识该符号是否为全局变量；而 TempId 是一个整型标识符，主要在中间代码生成过程中使用，用于对临时变量进行编号和区分。SymbolType 成员采用自定义类型 SymType，用于标识符号的种类，例如整数、浮点数、数组、函数等，具体包括如 INT32、FLOAT32、Int32Array、Float32Array 以及 FuncName 等不同的符号类型。

此外，ConstFlag 用于记录该符号是否为常量，MemoryFlag 则表示该符号是否已经完成了内存分配。在具体属性的管理上，标量变量的信息被封装在 VarAttributes 中，其类型为 VarInfo，包括变量的值及其语义细节；数组变量则由 ArrAttributes（类型为 ArrayInfo*）进行管理，支持记录数组的维度、初值等信息；函数类型的符号属性则存储在 FunctionAttributes（类型为 FunctionInfo*）中，涵盖函数的返回值类型、参数列表及其他附加属性。

为了适配不同类型符号的初始化需求，Symbol 类提供了多个重载的构造函数。对于标量变量，分别有整数型和浮点型两种构造函数，均接收是否为常量、是否已分配内存以及变量初始值作为参数，从而构造出具有完整语义信息的符号实体。对于数组类型，构造函数分别支持整型和浮点型数组的初始化，除了是否为常量以外，还需传入一个维度向量和可选的初始化值结构（支持常量树与表达式树），以支持静态与动态初始化的两种情况。函数构造函数则更加简洁，只需传入函数的返回值类型与参数类型列表即可完成符号的创建。这些构造函数的设

计为符号表支持不同类型的语言元素提供了灵活性和扩展性。

在功能接口方面，Symbol 类提供了若干成员函数。例如，GetGlobalFlag() 是一个常量成员函数，用于查询当前符号是否为全局变量。而 GetLabelStr(std::string SymName) 则根据全局或局部标识，在符号名称前加上特定前缀（如 % 或 @），用于生成中间代码中的标识符标签，保持命名一致性。类的析构函数 ~Symbol() 负责释放在函数信息（FunctionAttributes）等成员中动态分配的内存，避免内存泄漏。

总的来说，Symbol 类设计完整，功能丰富，核心目标是统一封装各种语言符号的属性信息，为前后端的编译处理提供统一的数据接口。它支持多维数组、复杂函数签名、作用域控制等高级特性，并通过对动态内存管理的封装提高了系统的健壮性和可维护性，是整个编译器符号管理系统的重要组成部分。

类 SymbolTable：

SymbolTable 类借助 Symbol 类实现了一个符号表，用于管理程序中的符号 及其作用域。该符号表支持嵌套作用域和全局符号的管理,如 代码 3.2.8-2。

```
class SymbolTable{

public:
static bool AddSymbol(std::string name,Symbol* sym);
static Symbol* FindSymbol(std::string name);
static Symbol* FindLocalScopeSymbol(std::string name);
static void EnterScope();
static void ExitScope();
static void InitTable();
static std::string AddConstString(std::string &str);
static void PrintConstStringDeclare();
static void AddGlobalSym(std::string name,Symbol* sym);
private:
static std::vector<std::map<std::string,Symbol*>> TableVec;
static std::map<std::string, std::string> ConstStringMap;
static int UnNamedStringCounter;
};
```

代码 3.2.8-2 符号表设计Ⅱ：类 SymbolTable

SymbolTable 类作为编译器中核心的符号管理模块，承担着符号记录、作用域控制和常量字符串维护等多项职责。其设计充分考虑了作用域嵌套、符号查找效率以及符号信息的完整性，是整个语义分析与中间代码生成过程的基础支撑。

在成员变量方面，SymbolTable 内部维护了一个静态成员 TableVec，它本

质上是一个栈结构，每个元素都是一个 std::map<std::string, Symbol*> 类型的映射，用于记录当前作用域下的所有符号。这种设计使得作用域的进入与退出操作可以通过栈顶元素的压栈与弹栈来高效实现，从而支持语言中常见的块级作用域结构。此外，为了统一管理程序中的常量字符串，该类还引入了 ConstStringMap，用于保存字符串文本与其唯一标识符之间的映射；同时，通过 UnNamedStringCounter 静态变量，为每一个未命名字符串生成递增编号，以便在后续生成中间代码或汇编时引用唯一标识。

在功能接口方面，SymbolTable 提供了多个静态成员函数来实现符号的增查操作。AddSymbol 函数将一个符号添加到当前作用域，如果该符号在该作用域中已存在，则返回 false，防止重复定义。FindSymbol 函数则从当前作用域开始，依次向上查找符号，直到找到目标或搜索到全局作用域；而 FindLocalScopeSymbol 专门限制在当前作用域中进行查找，便于在语义分析中准确处理局部变量与重名符号的冲突问题。针对作用域管理，EnterScope 和 ExitScope 分别用于新建和退出一个作用域，它们通过栈结构操作来动态维护作用域嵌套关系；而 InitTable 则用于在编译器初始化阶段建立全局作用域，并作为符号表的起始环境。此外，AddGlobalSym 可直接向全局符号表中插入符号，常用于函数声明、全局变量等符号的注册。

在常量字符串的管理上，AddConstString 会将字符串插入 ConstStringMap，为其生成唯一标识并返回，避免了重复存储同一字符串的问题；PrintConstStringDeclare 则负责在最终生成代码中输出这些字符串的声明，通常对应于 .data 段的内容，保证运行时能够正确引用这些字符串资源。

整体来看，SymbolTable 与 Symbol 两个类紧密配合，共同构成了编译器中符号系统的基础架构。前者负责作用域管理和符号生命周期控制，后者负责封装每一个符号的具体属性信息。这一设计不仅支持多种符号类型的统一管理（包括整型、浮点型、数组、函数等），而且通过栈式作用域管理策略，实现了对局部和全局符号查找的精确控制。在实现细节上，项目还借助动态内存管理策略，为数组与函数等复杂类型符号提供高效而安全的内存支持；同时通过字符串常量池机制，优化了对常量字符串的存储与引用效率。这样的架构设计在保证编译器正确性与可维护性的同时，也为后续的扩展与优化留下了良好的接口和结构基础。

3.3 中间代码生成

3.3.1 中间代码说明

中间代码是编译器的重要组成部分，是编译过程中的一种中间表示形式，介于源代码和目标代码之间。其主要作用是简化编译器的设计，提高编译器的可移植性和模块化，方便进行一系列优化。本项目中间代码形式类LLVM。

中间代码在现代编译器体系中扮演着承上启下的重要角色，其核心价值在于将前端与后端逻辑解耦，提升系统的灵活性与可维护性。首先，中间代码的引入极大地简化了前端和后端之间的耦合关系。在不使用中间代码的传统编译器中，前端生成的结构往往需要与具体目标平台高度贴合，导致每新增一个目标架构就必须重写大量前端逻辑。而有了中间代码作为过渡层，前端负责将源代码翻译为一种通用、平台无关的中间表示，而后端再基于这种中间表示生成特定平台的目标代码，从而使得整个编译器结构更加模块化。例如，在本项目中，我们通过将 SysY 语言翻译为类 LLVM 的中间表示，便实现了前后端之间的明确分工，使得未来更换目标架构（如从 RISC-V 切换为 ARM 或 x86）只需替换后端模块，而无需重写词法、语法等前端流程。

其次，中间代码为编译优化提供了良好的平台。相比于直接在源代码或汇编层进行优化，中间代码由于其抽象层次适中，既保留了程序的逻辑结构，又规避了底层架构的复杂细节，因此更便于实现各种中端优化策略。例如常量传播、死代码消除、公用子表达式消除、循环展开等优化技术，通常都在这一阶段完成。更重要的是，这些优化与具体硬件无关，具有较强的可重用性和跨平台通用性，从而进一步增强了编译器的适配能力。

再者，中间代码显著提升了编译器的可移植性。通过引入中间代码这一中立表示，我们可以统一前端的设计逻辑，而通过编写针对不同架构的后端翻译模块，实现对多种平台的支持。这种“编译一次，到处执行”的机制，使得编译器的维护和拓展变得更加高效，也更契合当前跨平台开发的趋势。

最后，中间代码的使用有效降低了编译器的整体复杂度。将整个编译过程划分为多个阶段，每个阶段围绕中间代码进行开发，使得开发者可以专注于各

自模块的实现和优化。例如语义分析人员只需确保中间代码的正确性，而后端人员则基于已有的中间表示进行机器码的生成。这种分层设计既促进了团队协作，又提高了系统的可测试性和可维护性，特别适用于教学、研究与工程实践等多种场景。

综上所述，中间代码不仅是编译器设计中的一种工程权衡，更是系统可扩展性、可优化性与跨平台能力的关键支撑。它在本项目中所承担的功能，不仅体现为技术实现的桥梁，更体现为整个编译架构设计的核心思想。

常见的中间代码表示：

在编译器实现中，中间代码的表示方式多种多样，每种形式都根据不同阶段的处理需求具备独特优势。为了更有效地完成语义分析、优化以及目标代码生成等任务，编译器通常采用以下几种主流的中间表示形式。

一种最常用的表示形式是**三地址码（Three-Address Code, TAC）**，它采用类汇编风格的结构，每条指令最多包含三个操作数：两个源操作数和一个目标地址。这种形式易于线性表示并能直接映射为机器指令，因此非常适合后端处理。在 TAC 中，一个复杂的表达式会被分解成一系列简单的操作，例如 `a = b + c * d` 会转化为 `t1 = c * d; t2 = b + t1; a = t2;`。由于每条指令执行一个操作，它在优化和代码生成阶段提供了极高的清晰度和控制力。

另一种重要的中间表示是**抽象语法树（Abstract Syntax Tree, AST）**，它主要服务于语法分析和语义检查阶段。AST 并不关注具体的指令顺序，而是以语法结构为核心，用树形结构表示程序中各个语法成分之间的层级关系。每个节点代表一种语法结构（如表达式、语句、函数定义等），而子节点则表示其组成部分。通过遍历 AST，编译器能够完成类型检查、作用域验证和符号绑定等语义处理，同时也为后续中间代码生成提供结构基础。

在程序流程控制方面，**控制流图（Control Flow Graph, CFG）**则显得尤为关键。CFG 将程序划分为若干个基本块，即在执行过程中始终顺序执行的一组语句，并用边表示这些块之间的跳转关系。通过构造 CFG，编译器能够分析程序的执行路径、判断语句的活跃性、发现循环结构以及进行基于路径的优化处理。CFG

也是数据流分析和高级优化（如环检测、循环不变代码外提等）的基础。

最后，**静态单赋值形式（Static Single Assignment, SSA）**是一种非常强大的中间表示，在大多数现代编译器中被广泛采用。在 SSA 形式中，程序中每个变量在其作用范围内只被赋值一次，每次赋值都会引入一个新的变量版本号。这种形式使得变量的数据流关系变得异常清晰，极大简化了依赖分析与优化处理。例如，常量传播、死代码消除和全局值编号等优化策略在 SSA 形式下实现更加自然高效。此外，为了支持控制流中多个路径合并时的变量赋值，SSA 引入了特殊的 phi 函数，使得变量合并变得显式可控。

这些中间表示通常并非彼此独立使用，而是按阶段逐步转化、层层衔接。在本项目中，我们从 AST 结构出发，生成三地址码，再构造控制流图并进一步转换为 SSA 形式，以便进行多种中间优化处理，最后再映射为目标平台的汇编指令。通过这样的多层次中间表示体系，我们不仅提升了编译器的模块化程度，也为实现高质量、高性能的目标代码生成提供了坚实的基础。

本项目实现的中间代码：

在本项目中，中间代码的实现融合了抽象语法树（AST）、控制流图（CFG）以及静态单赋值形式（SSA）三种常见的中间表示方式，力求在保持结构清晰的基础上，兼顾调试、优化和代码生成的灵活性与可维护性。

首先，**AST** 作为中间表示的初始形式，用于保留源代码的语法结构。在完成语法分析后，AST 承载着程序的层次化组织信息，为后续的符号分析和语义检查提供了结构基础。每个节点对应一个语法成分，比如函数定义、变量声明、表达式等。通过遍历 AST，可以高效地完成符号表构建、中间代码的初步生成，并为 CFG 构造提供直接支撑。

在中间阶段，项目将程序从 AST 进一步转化为 **控制流图**。CFG 将程序划分为多个基本块，并清晰地表达了各个块之间的跳转关系。这种图状结构不仅更符合程序执行过程的真实路径，也为多种路径敏感的优化打下了基础。通过构建 CFG，项目能够执行基本块划分、支配边界分析、环检测等工作，同时也便于后

续的 SSA 形式转换。

进一步地，在 CFG 的基础上，我们将中间代码转换为 **静态单赋值形式**。SSA 的引入是本项目中间表示的重要特色，它使每个变量在其作用域中只赋值一次，从而极大简化了变量的使用关系。借助 SSA 形式，项目可以顺利实现诸如常量传播、死代码消除、公用子表达式消除、部分冗余消除（SSAPRE）等中端优化策略。同时，Phi 函数的引入，使得程序在多路径合并处的值选择问题得到良好解决，进一步提高了中间代码在优化阶段的表现力和处理效率。

这一中间表示体系的设计兼顾了多方面需求。首先是**简洁性与可读性**，在调试和测试阶段，开发者可以清晰观察每一条中间指令的生成与执行路径，有效定位错误或性能瓶颈。其次是**兼容性与可移植性**，中间代码的抽象层级恰到好处，不依赖具体平台特性，为后端生成 RISC-V 或未来扩展至 ARM/x86 等平台的代码提供了通用支撑。最后，该中间代码体系具有良好的**可扩展性**，未来若需要加入函数内联、循环矢量化、并行化等更高级的优化策略，只需在 SSA 层或 CFG 层进行扩展，无需更改前端或词法语法分析逻辑。

综上，本项目设计与实现的中间代码体系不仅满足教学实验对可读性和结构化的需求，更具备工程意义上的稳定性与灵活性，为编译器后续的优化与平台适配打下了坚实基础。

3.3.2 中间代码生成的过程

中间代码的生成基于语法分析和语义分析的结果。通过遍历抽象语法树（AST），逐步翻译为中间代码。在遍历过程中从符号表中读取需要的变量属性 等信息。

中间代码生成最大程度上参照了 llvm 的处理方式，生成的是类 SSA 形式的 IR，这使得下一步转为严格 SSA 并优化更容易，并且前端最终生成的前端代码 可使用 llvm 工具链进行检验，可以快速找到问题所在，极大简化了后期测试。

常见的中间指令：（类 LLVM）

中间代码的生成建立在语法分析与语义分析的基础之上。在完成抽象语法树

（AST）的构建后，编译器会对 AST 进行深度遍历，根据各个语法节点的语义含义逐步翻译为中间指令。在这一过程中，符号表提供了关键支持，编译器通过查询符号表来获取变量的类型信息、作用域属性及是否初始化等信息，确保生成的中间代码既语义正确，又具备可进一步优化的潜力。

本项目在中间代码的设计上大量借鉴了 LLVM 的处理方式，生成的 IR（中间表示）具备类 SSA（静态单赋值）特性。这种设计不仅增强了表达能力，也使得前端代码与后端优化的对接变得更加自然。类 SSA 的 IR 极大地简化了后续转化为严格 SSA 格式的过程，为中端的各类优化打下良好基础。此外，由于前端 IR 的风格与 LLVM 高度相似，开发者可以利用 LLVM 工具链对生成的中间代码进行验证与调试，快速发现并定位语义错误和逻辑问题，这在项目测试与调优过程中表现出了显著优势。

在具体的中间代码实现中，以下几类指令是最为常见和基础的组成部分：

1. 内存管理指令

Alloca 指令用于在函数栈帧中为局部变量分配内存空间，其返回值是一个指针，指向该内存区域。例如：%ptr = alloca i32 表示为一个 i32 类型的变量分配内存，%ptr 为其地址。

Store 指令用于将寄存器中的值写入某个内存地址。例如：store i32 %value, i32* %ptr 表示将变量 %value 的值存储到指针 %ptr 所指向的位置。

Load 指令则执行与 store 相反的操作，它从内存中读取一个值并加载到寄存器中，返回一个新的 SSA 值。例如：%value = load i32, i32* %ptr 表示从指针 %ptr 指向的内存读取一个 i32 类型的值并存储到 %value。

2. 算术与逻辑指令

Binary 指令用于处理二元运算，例如加、减、乘、除以及逻辑运算等。例如，%sum = add i32 %a, %b 表示对 %a 和 %b 执行整数加法，结果保存在 %sum 中。逻辑操作如 %and = and i32 %x, %y 表示按位与操作。

值得注意的是，一些单操作数的运算（如取负）在中间代码中可转换为特殊形式的二元运算。例如，取负操作 -%x 可表达为 sub i32 0, %x，即用 0 减去 %x 来实现。

3. 控制流指令

控制程序流程的关键在于 br 和 ret 等指令。br 是跳转指令，它既可以是无条件跳转（br label %target），也可以是条件跳转（br i1 %cond, label %trueBlock, label %falseBlock）。条件跳转根据布尔值 %cond 的真假，跳转到不同的基本块，是实现 if/else 和循环结构的基础。

Call 指令用于函数调用。例如：%result = call i32 @functionName(i32 %arg1, float %arg2) 表示调用名为 functionName 的函数，并传入两个参数，返回一个 i32 类型的结果。

Ret（Return）指令则标志着函数的结束，并返回指定值。若函数无返回值，则使用 ret void 表示。

这些指令类型构成了中间代码的基本语义表达系统。通过它们的组合，编译器可以完整地表示源程序的控制流与数据流，为下一阶段的优化与代码生成打下坚实的基础。

3.3.3 本项目样例生成的中间代码展示

测试用例（代码 3.3.3-1）经过中间代码生成后生成内存形式的中间代码，导出如代码3.3.3-2形式的类llvm 中间代码文本形式。

```
void swap(int a,int b)
{
inttemp=a;
a=b;
b=temp;
}
int a=110;
int main()
{
int x=111;
int y=222;
swap(x,y);
return 0;
}
```

代码 3.3.3-1 具体的测试样例

```
define void @swap(i32 %0, i32 %1)
{ %3 = alloca i32 , align 4
store i32 %0, i32 * %3
store i32 %1, i32 %0  %4 = load i32,i32* %3 store i32 %4, i32 %1  }
@a = global i32 110, align 4 define i32 @main(){
%1 = alloca i32 , align 4
store i32 111, i32* %1
%2 = alloca i32 , align 4
store i32 222, i32* %2 %3 = load i32,i32* %1
%4 = load i32,i32* %2
```

```
call void @swap(i32 %3, i32 %4) ret i32 0
}
```

代码 3.3.3-2 测试样例生成的中间代码dump出的.ll文本形式

3.4 核心类设计

3.4.1 核心类设计

在中间代码生成过程中，设计一个合理的中间表示结构是连接前端语法树与后端目标代码的重要桥梁。通过遍历抽象语法树（AST），编译器逐步构建出基本的中间代码表示。为了保持模块之间的解耦与中间表示的灵活性，本项目在中间表示的设计上深度借鉴了 LLVM 的内存模型与对象体系，构建了一套核心类结构，包括 Use、Value、User、Instruction、BasicBlock、Function 与 Module 等关键类。这些类之间通过继承与组合关系，共同构成了一个清晰、可扩展的中间代码系统。

其中，Use 类用来表示一个 Value 实体在程序中的使用位置。它不仅指向具体被使用的 Value，还维护着这个使用点与使用者（如指令）之间的关系，有助于实现数据流分析、引用追踪等中端优化操作。

Value 是整个中间表示体系的抽象基类。它表示程序中所有具有计算意义的实体，如变量、常量、临时值等。所有参与指令运算的数据都被统一封装为 Value 对象，保证了操作数之间的一致性。

User 是一个用于表示“使用其他 Value”的抽象实体的类，其本质上继承自 Value，因为它既作为一个值被其他指令使用，同时又持有对其他 Value 的引用。User 的设计使得如 Instruction 这类类可以追踪它使用的每一个操作数。

Instruction 类继承自 User，表示程序中的单条指令。每一条指令既是一个值（例如可以参与到后续运算中），又是一个使用者（它的操作数是其他 Value）。每条指令通过指定操作类型（如加法、跳转、函数调用等）来表述特定的行为。

BasicBlock 类用于描述基本块，即控制流图中一段连续执行、只包含唯一入口和唯一出口的指令序列。每个基本块包含多条 Instruction，并通过前驱与

后继关系形成完整的控制流图（CFG）。基本块是进行控制流分析与局部优化的基本单位。

Function 类代表程序中的函数实体。每个函数由若干个基本块构成，具备入口块、返回类型与参数列表等属性，是进行跨基本块优化、寄存器分配及后端代码生成的重要结构。

其中，Module 是最顶层的容器类，用于表示一个完整的程序模块，内部包含多个 Function 实例，统一管理整个编译单元的中间表示与全局信息。

整体而言，这一组核心类共同构建出中间代码的骨架，它们的合理设计不仅提升了中间代码的表达能力和可维护性，也为后续的中端优化与目标代码生成奠定了良好的结构基础。

在 LLVM 当中几乎所有的类都是Value，在我们的数据结构中它用来承载基 础的信息。如果一个Value被使用了那么使用它的数据结构我们称为User，同 时注意，User很可能也会在其他地方作为Value使用，所以他需要继承于 Value类。

核心类的关系如图（图 3.4.1-1）：

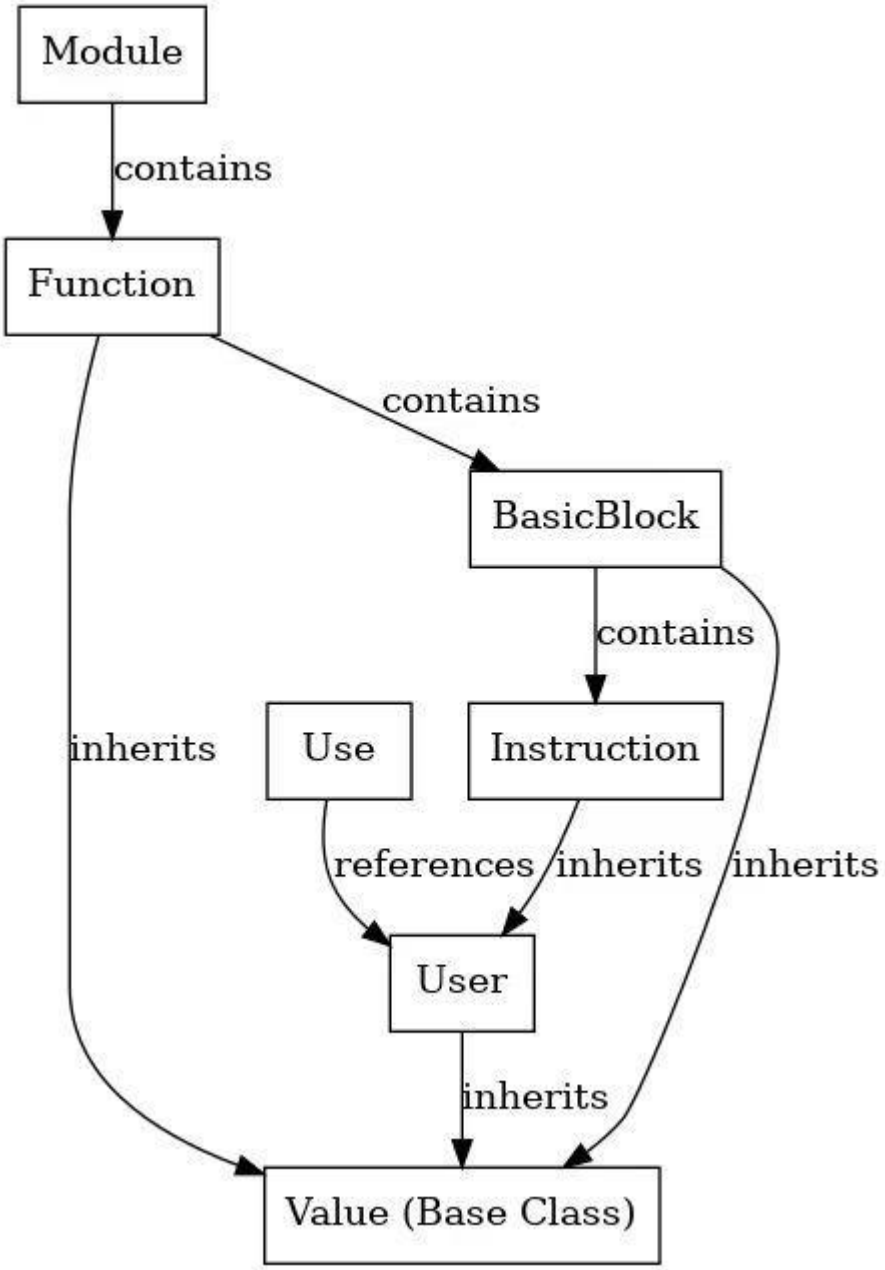


图 3.4.1-1

同时根据 LLVM 的Use-Def 设计，我们还需要建立一个Use 类以及相应的UseList(*Use)来管理 Value 和User 的这种关系，它就相当于一条边连接Value和User。

通过上图我们可以看到一个 Instruction包含了多个Use、一个User、并 通过Use连接了多个Value对象。

设计一套类似于LLVM 的核心中间类库是一个复杂且关键的任务,所以我们小组 采用的方式为：前端设计类的基本框架，中端根据自己的部分所需功能添加修 改，后端能理解使用。

LLVM 的核心类（如 Value、User、Instruction 等）是面向中间表示（IR）的，体现了面向对象的编译设计思想，方便实现优化和代码生成。接下 来将会展示我们设计的User-Use-Value基本框架，是一个简化的、能够自洽的 中间核心类设计，使用C++描述。

在核心类设计中，我们确立了三个主要设计目标：**可扩展性**、**模块化** 和 **自洽性**。可扩展性意味着整个 IR 系统应具备灵活性，便于在后续开发中添加新的指令类型、数值类型或数据结构，不需对现有框架进行大规模重构。模块化则强调各组件之间职责清晰、边界明确，彼此通过定义良好的接口协作，从而降低耦合、提升代码的可维护性。自洽性是确保系统内部结构合理、数据流通畅的关键，即类之间能够自然协作，共同完成中间代码的构建、组织和使用。

从类的层次结构来看，Value 作为所有中间表示实体的抽象基类，奠定了统一的接口基础。而 User 是继承自 Value 的重要子类，表示那些**依赖其他值**的实体，其中最典型的就是 Instruction 类，它不仅作为 IR 中的具体执行语句，还承担了使用多个 Value 的职责。容器类如 BasicBlock、Function 和 Module 则从底到上构成了中间代码的组织体系，每一层都对下层内容进行封装与管理：BasicBlock 管理一系列指令，Function 管理若干基本块，而 Module 作为顶层结构，汇总所有函数，最终构成完整的中间代码模块。

在类之间的协作机制中，Value 和 Use 搭建了 IR 数据流追踪的基础：Use 记录了 Value 的使用位置，使得一个值被引用的所有地方都可以被追踪与分析。这种机制对于后续的数据流分析、变量替换和优化极为重要。与此同时，Instruction 与 BasicBlock 紧密耦合，使得指令能够正确地归属于逻辑上的控制结构中，形成良好的流程结构。

为了支持优化与后端生成的需求，该套核心类系统专门留有接口和设计空间。前端提供了基础的类定义和通用接口，而中端可以在不破坏结构的前提下，扩展新的 Instruction 类型，或定义新的 Value 以适配更复杂的计算需求。这一特性也使得该 IR 系统在兼容 LLVM 设计理念的同时，具有高度的可塑性与实践价值。

3.4.2 Use-Def 链

Use-Def 链(双向链表)是编译器设计和中间表示（IR）管理中的重要概念。它将每个 Value（定义）与其对应的 Use（使用点）连接起来，从而支持 数据流分析、优化过程以及SSA形式的转换。

主要通过基类Value 中定义的UseList(Use*)实现，支持：1、Value 查询被哪些User使用(双向链表)；2、User 查询使用了哪些Value(地址偏移) 。如下示意图清晰展示了这两种关系：

1、 Value查询被哪些User使用（图 3.4.2-1）：

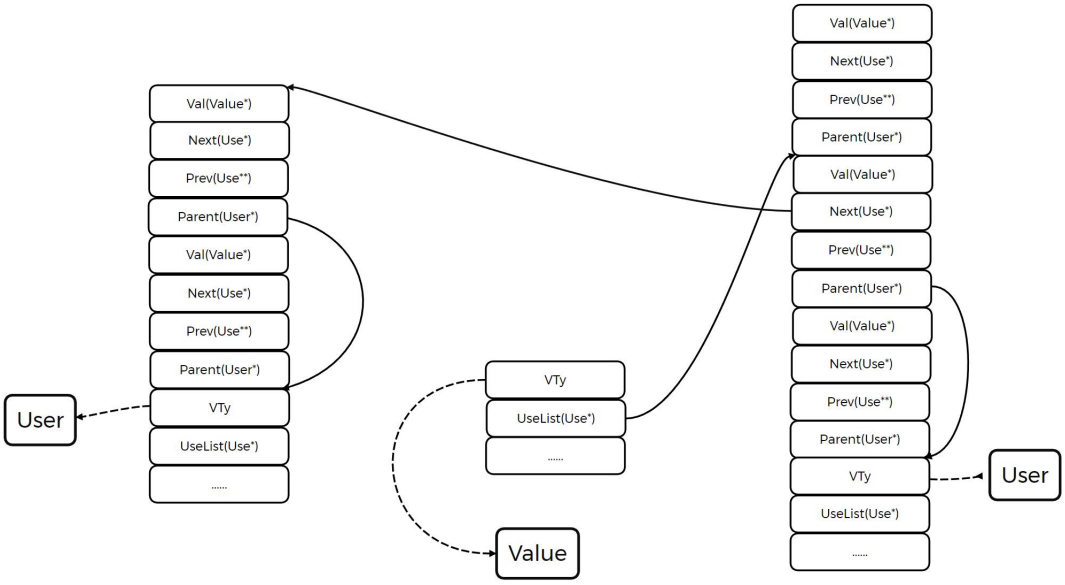


图 3.4.2-1

2、 User 查询使用了哪些Value（图 3.4.2-2）：

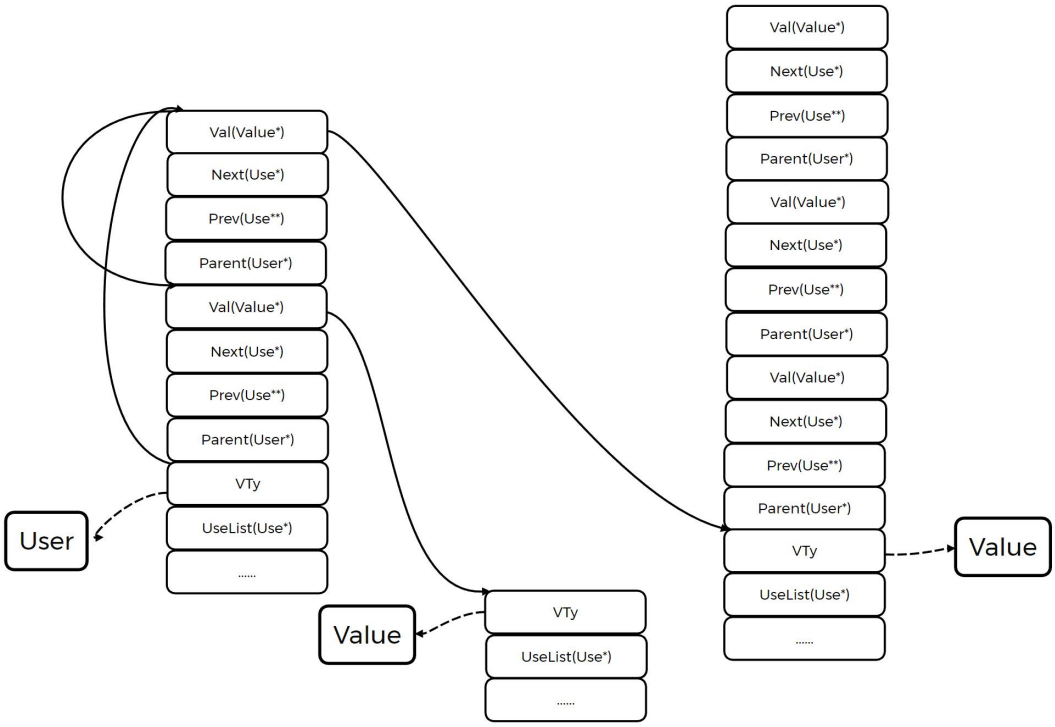


图 3.4.2-2

这套核心类设计在实际编译器实现中有着广泛的应用场景。首先，在数据流分析阶段，它通过精确跟踪每一个 Value 的使用点，为死代码删除（DCE）和常量传播等关键优化提供了坚实的数据基础。利用 Use 结构，编译器能够识别那些未被实际使用的计算结果，从而有效地剔除无用代码，提高生成代码的效率和质量。

其次，在生成静态单赋值（SSA）形式时，这些类同样发挥着重要作用。通过明确标识每个变量的定义位置及其所有使用点，简化了 SSA 形式的构造过程，使变量的重命名和插入 phi 函数变得更为直观和高效。这为后续的优化步骤铺平了道路，增强了编译器对代码的分析和重写能力。

此外，核心类系统也极大支持了代码优化阶段的多种技术。例如，在进行公共子表达式消除（CSE）时，系统能够准确地分析和替换相同的 Value，避免重复计算，进一步精简代码。所有对同一 Value 的引用都能被有效管理，确保优化过程的正确性和完整性。

最后，这些类还促进了指令的简化，通过追踪 Value 之间的依赖关系，识别并替换冗余计算，降低代码复杂度和执行开销。总体来看，这种紧密结合的类设计不仅提升了中间代码的表达能力，也为高效的编译优化提供了坚实的结构保

障。

3.4.3 前端与中端的交接协作——具体的类接口

介绍完主要的数据结构，代码 3.6.3-1 以我们设计的基本Value、Use、User框架为例，介绍了重要接口及其实现。注意本代码只展示了重要接口。

```
class Use {

public:
Use(User* u, Value* v) : user(u), value(v) {}
private:
~Use() {
if (Val)
removeFromList();
 }

private:
// 指向被使用的 Value
Value *Val = nullptr;
// 指向下一个关联相同 Value 的 Use 边
Use *Next = nullptr;
// 指向上一个关联相同 Value 的 Use 边（二级指针方便替换）
Use **Prev = nullptr;
// 指向使用该值的 User  User *Parent = nullptr;

void  addToList(Use  **List)
{ Next = *List;
if (Next)
Next->Prev = &Next;
Prev = List;
*Prev = this; }
void removeFromList() {
*Prev = Next;
if (Next)
Next->Prev = Prev; }
.............
}
class   Value
{ private:
// 使能够找到使用该 Value 的 User
Use *UseList;

//能够实现替换相关优化的基础函数
void Value::doRAUW(Value *New, ReplaceMetadataUses
 ReplaceMetaUses)
 {
while    (!materialized_use_empty())
{ Use &U = *UseList;
U.set(New);
 }
}
void addUse(Use &U) { U.addToList(&UseList); }
```

```
 public:
 void   Value::replaceAllUsesWith(Value   *New)
 { doRAUW(New,ReplaceMetadataUses::Yes);
 }
 ..............
};
```

代码 3.4.3-1 中间代码的重要接口及其实现

3.5 划分基本块和控制流图的构建

在编译器的中间表示阶段，对程序进行基本块的划分和控制流图（CFG，Control Flow Graph）的构建是至关重要的步骤。基本块是指一段连续执行的指令序列，其特点是在块内没有分支跳转，且只有一个入口和一个出口。通过将程序划分为多个基本块，可以更清晰地反映程序的执行路径，有助于后续的分析和优化。

3.5.1 基本块划分

基本块（BasicBlock）是指一组顺序执行的指令，这些指令中间没有分支和跳 转，只有入口和出口。基本块的划分过程首先通过识别程序中的分支指令、跳转目标和函数入口等位置，将连续的指令序列切分成若干个互不重叠的基本块。每个基本块内的指令按顺序执行，中途不发生跳转，确保执行的线性性。基本块通常满足以下条件：

1. 只有第一个指令可能作为入口。
2. 只有最后一个指令可能跳转到其他基本块。

划分步骤：

1. **标记基本块的边界：**
    1. 标记程序的入口指令。
    2. 标记跳转指令的目标为基本块的入口。
    3. 标记跳转指令的下一条指令为基本块的入口。

2. **根据边界划分基本块：**

   从入口指令开始，将顺序的指令划入同一个基本块，直到遇到下一条入口指令或跳转指令。

3.5.2 控制流图（CFG）构建

控制流图(Control Flow Graph, CFG)是程序基本块之间控制流关系的有 向图。每个节点表示一个基本块，边表示可能的控制流路径。在基本块划分完成后，构建控制流图，即用图的形式表示基本块之间的跳转关系。图中的每个节点代表一个基本块，边表示可能的控制流转移路径，如条件跳转或无条件跳转。控制流图清晰地描述了程序中所有可能的执行路径，是数据流分析、死代码删除、循环优化等多种高级优化的基础。

构建步骤：

1. **构造基本块：** 根据基本块划分结果，为每个基本块分配唯一标识。
2. **添加边：** 根据跳转指令和顺序执行逻辑，添加基本块之间的两种边。
   1. 顺序边：基本块的最后一条指令不是跳转指令时，指向下一个基本块。
   2. 跳转边：跳转指令指向目标基本块。

图形化输出：使用工具如 Graphviz，可以生成 .dot 文件并可视化控制流图。

通过基本块和 CFG 的建立，编译器能够更好地进行静态分析和代码转换。例如，在生成 SSA 形式时，需要依赖 CFG 来准确插入 phi 函数，以保证变量定义与使用的正确性。此外，基于 CFG，优化器可以执行跨基本块的优化，识别循环结构和控制依赖，进一步提升生成代码的效率和质量。

总之，基本块的划分和控制流图的构建不仅为中间代码的结构化提供了清晰的框架，也为编译器的中后端各种复杂优化策略奠定了坚实的基础，使整个编译流程更加科学和高效。

3.6 SSA 形式的转化

3.6.1 中端的主要任务

中端的主要任务是对中间代码(IR)进行一系列的优化和处理，生成更加高效的中间代码(IR)，让后端可以更加好地生成目标机器代码，使性能更优，以 下是中端具体需要实现的相关内容。

静态单赋值（SSA）转换： SSA形式是一种中间表示，其中每个变量在程序中只被赋值一次，所有 对变量的引用都必须是该变量的单一赋值。这使得数据流分析和优化变 得更加简单。 编译器会通过mem2reg等技术，将内存中的变量提升为寄存器变量，并 且插入 Φ 函数(Φ-function）来处理不同控制流路径中的变量合并问 题。

数据流分析（Data Flow Analysis)： 数据流分析通过跟踪程序中数据的传递情况，帮助编译器识别可能的优 化点。例如，确定哪些变量的值没有改变，可以安全地消除冗余计算， 或者哪些变量永远不会被使用。数据流分析包括常见的技术如活跃变量分析、常量传播、别名分析、死 代码消除等。

中间表示优化（IR Optimization)：中端的重要任务之一是优化中间表示，以提高程序的性能或减小其体积。常见的优化策略包括：常量折叠，将表达式中的常量计算提前到编译时，减少运行时计算。死代码消除，移除那些不影响程序输出的代码。循环优化，如循环展开、循环分割、循环合并等，以减少循环开 销。内联函数，将小函数直接嵌入到调用点，避免函数调用的开销。复制传播：消除无意义的变量复制。代码移动，将某些代码从频繁执行的位置移动到不常执行的位置，从而提高性能，等诸多优化策略。

3.6.2 构建支配树，寻找支配边界

为了构建 SSA 的形式，需要寻找支配边界来确定phi 函数需要插入的位置，而为了寻找支配边界，则需要构建支配树。

支配关系（Dominance）的定义：

在编译器中，支配关系是控制流图（Control Flow Graph, CFG）中的一种 重要关系。对于一个有向图（CFG）中的每个节点来说，节点 A 支配节点 B，意味着从入口节点(通常是入口函数或程序的开始)到节点 B 的每一条路径，都必须经过节点 A。支配：如果在控制流图中从起始节点到达节点 B 的任何路径都必须经过 节点 A，那么我们就说“节点 A 支配节点 B”。如果节点 A 不支配节点 B，则存在一条路径从起始节点到节点 B，但并不经过节点 A。

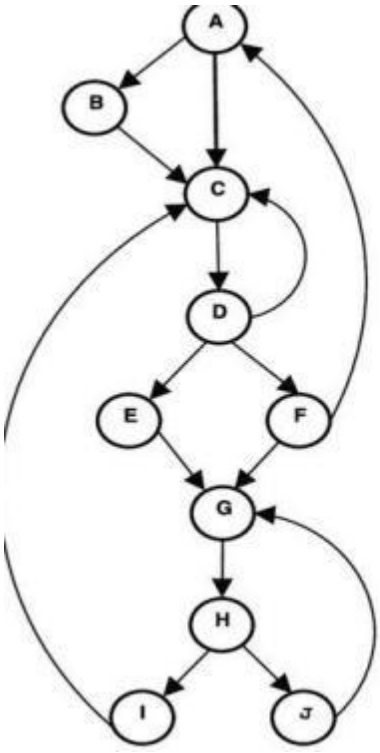
符号表示：节点 A 支配节点 B 通常表示为 A → B 或 A dominates B。

支配树（Dominance Tree）（见图 3.8.2-1）

支配树是控制流图（CFG）中一个重要的结构，它反映了节点间的支配关系。

定义：在支配树中，节点 A 是节点 B 的“支配者”，当且仅当 A 支配 B，并且没有其他节点支配 B 且比 A 更接近入口节点。

具体来说，如果节点 A 支配节点 B 且 B 没有其他支配者比 A 更接近 入口节点，那么在支配树中 A 将是 B 的父节点。



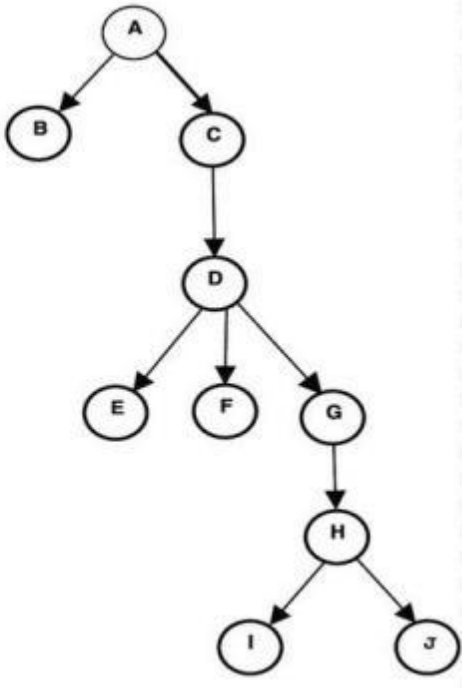


图 3.6.2-1 将 CFG 流图转化成支配树

支配树的根节点是入口节点，因为入口节点支配整个程序。

支配树的性质：

每个节点都与其支配节点之间有一条有向边。

对于每个节点，其所有的直接支配节点构成其父节点集合。

如果节点 A 是节点 B 的支配节点，则所有从 B 到任何其他节点 的路径，都会经过节点 A。

支配边界（Dominance Frontier）（见图 3.6.2-2）

支配边界是一个重要概念，通常用于静态单赋值(SSA)形式的构建和优化

定义：对于某个节点 A，节点 B 属于 A 的支配边界，当且仅当：
B 不是 A 的支配子节点（即 A 不直接支配 B）。
存在至少一条路径，从 A 到 B，其中这条路径经过 A，但并不通 过 A 的子节点。

直观地讲，支配边界包含了一个节点的所有“可能”影响的控制流路径的结束点。在编译器的优化和SSA 中，支配边界通常与 Φ 函数（Phi-function） 相关。

Φ 函数的插入位置通常位于支配边界。Φ 函数用于合并不同控制流路径 上的同一变量的赋值，确保在程序控制流图中每个分支（if/else 或循 环）中变量的值能被正确地合并。

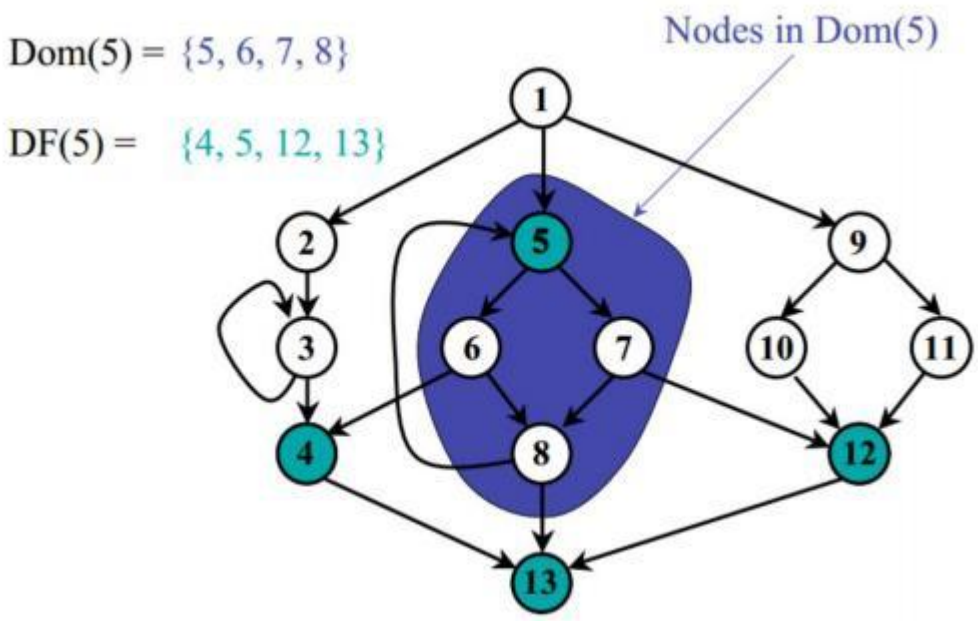


图 3.6.2-2 支配树

支配树实现的相关算法

在实现支配树的构造上，我们使用很经典的Lengauer-Tarjan算法，可以 有效地计算支配关系并构建支配树，尤其是当程序的控制流图较为复杂时，能 够比传统的迭代方法（如基于交集的支配集计算方法）更快地完成任务。

LT算法具体实现步骤

深度优先搜索：

在控制流图上进行深度优先搜索（DFS），并且记录下每个节点的发现时 间。对于每个节点 vvv，其发现时间（DFS 编号）记录在 dfs(v)中。深度优先搜索结果为计算后续的路径压缩和支 配树的构建奠定了基础，见代码 3.6.2-1。

```
void DFS(int pos) {
node[pos].dfnum = count;
node[pos].sdom = count; // 每个节点的 sdom先初始化为自己
vertex[count] = pos; // 记录每一个dfnum对应的结点
count++;
```

```
 for (auto p : node[pos].des) { if
 (node[p].dfnum == 0) {
 DFS(p);
 node[p].father = pos; }
 }
}
```

代码 3.6.2-1 获取 DFS 树的序号

支配集初始化：对每个节点，初始化支配集为整个图中的所有节点，即每个节点的支配集 一开始包含图中的所有节点，见代码 3.6.2-2。

```
 function BuildDominanceSets(G, D, Dom):
 for each u in V:
 for each v in Neighbors(u):
 if v not in Dom[u]:
 Dom[u].add(v)
 if D[v] != v:
Dom[v].add(u)

 D[v] = u
return D, Dom;
```

代码 3.6.2-2 构建支配集的伪代码

路径压缩：

在计算过程中，Lengauer-Tarjan 算法通过路径压缩优化支配集计算过程。路径压缩的基本思想是：对于一个节点 vvv，如果我们已经知道 vvv 支配 某个节点 uuu，并且 uuu 有多个前驱节点，那么我们可以将节点 uuu 的支配 集（即支配路径）合并成一个更小的支配集，这个过程通过路径压缩来实现。

并查集（Union-Find）数据结构：

使用并查集数据结构（Union-Find）来加速支配集的合并操作。在计算支 配集时，合并和查找操作可以通过并查集的路径压缩优化，使得算法在大规模 图上运行时更加高效。

计算支配树：

在构建支配树时,依据节点的支配关系来决定父子节点的结构。每个节点 的支配树父节点就是它的唯一支配节点。通过路径压缩和并查集，我们能够高 效地找出一个节点的唯一支配节点，进而构建出支配树，见代码 3.6.2-3。

```
void dominance::computeDF(int x) {
for (auto de : node[x].des) { if
(node[de].idom != x) {
df[x].df.push_front(de);
}
}
for (auto child : node[x].idom_child)
{ computeDF(child);
for (auto frontier : df[child].df) {
if (node[frontier].idom != x || x == frontier)
{ df[x].df.push_front(frontier);
}

}}}
```

代码 3.6.2-3 计算支配边

3.6.3 中端的第一个优化 mem2reg

中端的第一个优化是mem2reg，mem2reg是要对前端生成的目标代码进行变 量的提升，主要是alloca，store，load语句的消除，将这些操作转化成更加 高效的寄存器操作，从而提升程序的执行效率，并且要求插入phi 函数，将中间代码彻底转化为最标准的SSA 的形式。

Mem2reg 的优化过程，对alloca，store，load 的promote：

标识栈变量的生命周期：mem2reg 首先会分析哪些变量的内存只在一个基本块或函数范围内被使 用。如果一个栈变量的值只在一个基本块内被访问，并且没有跨越基本块或函 数调用的边界，它可以被提升到寄存器中。

替换 alloca 为寄存器变量：对于符合条件的 alloca，mem2reg 会将其分配的栈空间转换为寄存器，从 而消除对 alloca 的调用。

消除冗余的 store 操作：如果某个变量在寄存器中已经存在，不需要重复存储到栈中，编译器就会 消除不必要的 store 操作。

消除冗余的 load 操作：如果变量已存储在寄存器中，编译器会避免从栈中重新加载该变量,消除冗余的 load 操作。对是否可以Promote 进行检查,见代码 3.6.3-1。

```
bool promoteMemoryToRegister(Function &func, dominance &dom)
{ std::vector<AllocaInst *> Allocas;
auto BB = func.GetBasicBlock(); // BB 是一个
std::vector<BasicBlockPtr>
while (true) {
Allocas.clear();
for (auto &it : BB) {
List<User> &insts = it->GetInsts(); for
(auto &Instruct : insts) {
User *user = Instruct.get(); if
(auto allocaInst =

dynamic_cast<AllocaInst *>(user)) //确保是 alloc指令

if (IsAllocaPromotable(allocaInst))
Allocas.push_back(allocaInst);
}
}
if(Allocas.empty())//当前没有可以 promote 的 alloca 指令
break;
RunPromoteMem2Reg(dom,Allocas); }

}
```

代码 3.6.3-1 寄存器的提升

Mem2reg 在支配边界插入 phi 函数的算法实现：

在支配边界插入 phi 指令：假设在 B 处 store，那么在 DF(B) 中的基本块

被支配，可能有其它分支的 值流向它，需要插入 phi。插入 phi 后，流向支配边界的值被汇总，这个基本块对应的支配边界 DF(DF(B))又有分支。因此要求出一个支配边界的闭包 DF+(B)，然后在闭包内所有的基本块中插入 phi

插入 phi 函数 后，进行重命名：对 CFG 进行 DFS，并在 DFS 的每个结点中记录每个 alloca 对应的值 IncomingVals[]。对于遇到的指令进行如下操作：

Alloca：直接删除

Load：读取 IncomingVals[]

Store 或者 Phi：写入 IncomingVals[]

遍历完一个基本块后，遍历它的后继，向所有后继的 phi 中插入当前基本 块流向该后继的边，值为当前结点 IncomingVals[] 中的值。假设有多个基本块 A B 流入一个基本块 D，那么所有值都会在 phi 中汇聚在遍历 D 时，需要汇聚的值在 IncomingVals[]中的记录为对应的 phi。

因此上一步为 phi 添加边可以处理所有情况，遍历顺序无关紧要（可以先遍历 D 再遍历 A B）

其伪代码的实现如下，代码 3.6.3-2：

```
Insert_phi(x){
    alreadyList = {}
    workList = {Set of Nodes Assigning X}
    everOnWorkList = { Set of Nodes Assigning X}
    While workList !={}
        Remove N from workList
        for each M in DF(N)
            if M not in alreadyList
                Insert phi at M for variable X Add
                M to alreadyList
                if M not in everOnWorkList
```

```
                Add M to everOnWorkList
                Add M to workList
                }
```

代码 3.6.3-2 插入 phi 函数

Mem2reg实现展示，见代码3.6.3-3：

```
bool PromoteMem2Reg::promoteMemoryToRegister()
{
    for(int AllocaNum = 0; AllocaNum != Allocas.size(); ++AllocaNum){
        AllocaInst* AI = Allocas[AllocaNum];
        removeLifetimeIntrinsicUsers(AI);
       if(!AI->isUsed()){
            delete AI;
            RemoveFromAList(AllocaNum);
            continue; // 可有可无
        }
        if(Info.DefBlocks.size() == 1) // 仅仅只有一个定义的基本块
        {
            if(rewriteSingleStoreAlloca(Info,AI,BkInfo))
 {   RemoveFromAList(AllocaNum);
                continue;  }}
        if(PhiBlocks.size() > 1)
            std::sort(PhiBlocks.begin(),PhiBlocks.end(),
                    [this](BasicBlock* A,BasicBlock* B){
                    return BBNumbers.at(A) < BBNumbers.at(B);  });
       for(int i = 0, e = PhiBlocks.size(); i !=e; i++){
            QueuePhiNode(PhiBlocks[i],AllocaNum);
        }
```

```
    }
    RenamePassData::ValVector Values(Allocas.size());
    for(int i = 0, e =Allocas.size(); i!=e; ++i)

RenamePassWorkList.emplace_back(*(_func->begin()),nullptr,std::move(Va
lues));
    do{
        auto tmp = std::move(RenamePassWorkList.back());
        RenamePassWorkList.pop_back();


        // core function for rename
        // transfer WorkList
        RenamePass(tmp.BB,tmp.PredBB,tmp.Values,RenamePassWorkList);
    }while(!RenamePassWorkList.empty());
    NewPhiNodes.clear();
    return true;
}
```

代码 3.6.3-3 mem2reg 的简化代码

mem2reg 优化是中端第一个优化也是很重要的一个优化，有了中端优化 mem2reg 的优化，之后的其他优化也可以进行开展，经过 mem2reg 的优化，中端代码的形式已经彻底转化成了 SSA IR 的形式。

3.7 函数优化

3.7.1 内联优化

在本编译器的中端优化阶段，内联（Inline Expansion）优化作为一项重要的

局部优化手段，被用于消除频繁的小函数调用所带来的运行时开销。其基本思想是将被调用函数的函数体直接插入到调用点，从而避免运行时的函数调用机制带来的诸如压栈、跳转、保存返回地址等一系列控制转移和栈帧管理操作。通过在编译期完成函数展开，不仅能提升运行效率，还能为后续的优化策略（如常量传播、死代码删除等）提供更好的上下文信息，从而提高优化空间。

在编译器的中端阶段，即在语义分析和目标代码生成之间，我们使用一种类三地址码（TAC）作为中间表示，所有函数在语法分析后被翻译成相应的中间代码形式。内联优化以该中间代码为操作对象，在不破坏程序语义的前提下，将目标函数的中间代码插入到调用点所在的位置，并进行必要的变量重命名和作用域处理。相比在前端阶段进行宏替换式展开，或者在后端阶段进行函数内联建议，本项目的中端内联更有利于维持代码语义的准确性与优化策略的连贯性。

为实现该优化，我们首先对所有函数进行扫描与特征分析，判断其是否适合被内联展开。判断标准主要考虑函数的长度、递归性、返回行为和副作用等方面。对于满足条件的函数，我们将其标记为“可内联”，并在随后中间代码的处理阶段对其调用点进行替换操作。内联替换并非简单地进行代码复制，而需要完成形参与实参的绑定、中间变量的重命名，以及返回值传递与控制流回接等一系列转换。

内联优化的整体流程如下图（图 3.7.1-1）所示：

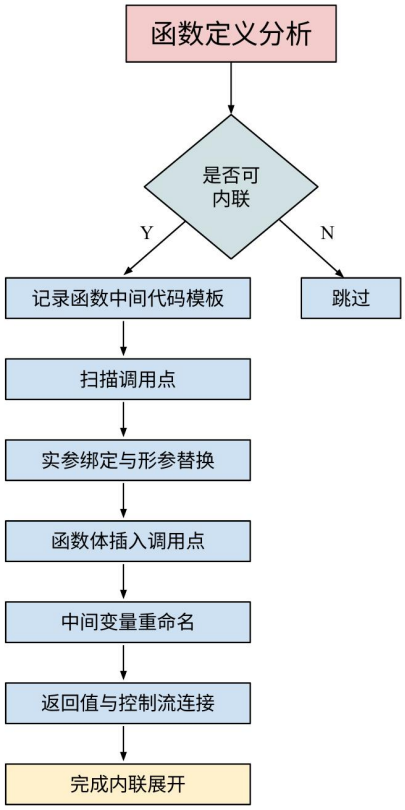


图 3.7.1-1 内联优化的整体流程

通过如上流程，内联优化在保持语义等价性的前提下，有效地将函数调用转化为直接执行的语句序列，消除了运行时的不必要开销，并为中间代码优化提供更丰富的上下文环境。

在执行内联优化前，必须对所有函数进行可内联性分析。并非所有函数都适合展开，否则可能造成代码爆炸（code bloat）或引发语义错误。因此，在本编译器中，我们设定了一系列约束条件和判断策略，用于对函数的结构和语义进行静态评估，仅对满足条件的函数进行内联处理。

首先，函数体的长度是一个重要的参考指标。我们采用的中间代码为类三地址形式，因此函数体的“长度”可以自然地度量为其所对应的中间代码条数。一般而言，如果一个函数对应的中间指令条数超过某一阈值（例如 10 条），则会被认为展开成本过高而被拒绝。对于如下简短的辅助函数：

```
int square(int x) {
```

```
    return x * x;
}
```

其对应的中间代码如下：

```
t1 = x * x
return t1
```

只有两条，显然适合内联。而对于以下结构复杂、控制流多分支的函数：

```
int compute(int a, int b) {
    if (a > b)
        return a * b;
    else
        return a - b;
}
```

其对应的中间代码可能包含多个标签、条件跳转与返回语句，在没有结构展开能力的情况下，难以完整、正确地嵌入到调用点。对此类函数，编译器将拒绝进行内联。

其次，函数不应具有副作用，尤其是不可预期的全局状态修改或 I/O 操作。例如，若函数内部调用了 printf 或修改了外部全局变量，那么内联后可能改变程序的行为顺序或破坏原有封装语义：

```
int debug(int x) {
    printf("x = %d\n", x); // 存在副作用，不可内联
    return x + 1;
}
```

同样地，递归函数也被明确排除在可内联范围之外。由于递归的展开具有不确定的深度，会导致编译期展开过程无限进行，造成逻辑死循环甚至编译器崩溃。因此，在函数定义分析阶段，我们会建立函数调用图并检测循环边。任何出现在调用图环路中的函数将被标记为“递归”，并跳过内联处理。

以上判断策略可简要总结为以下决策流程图（图 3.7.1-2）：

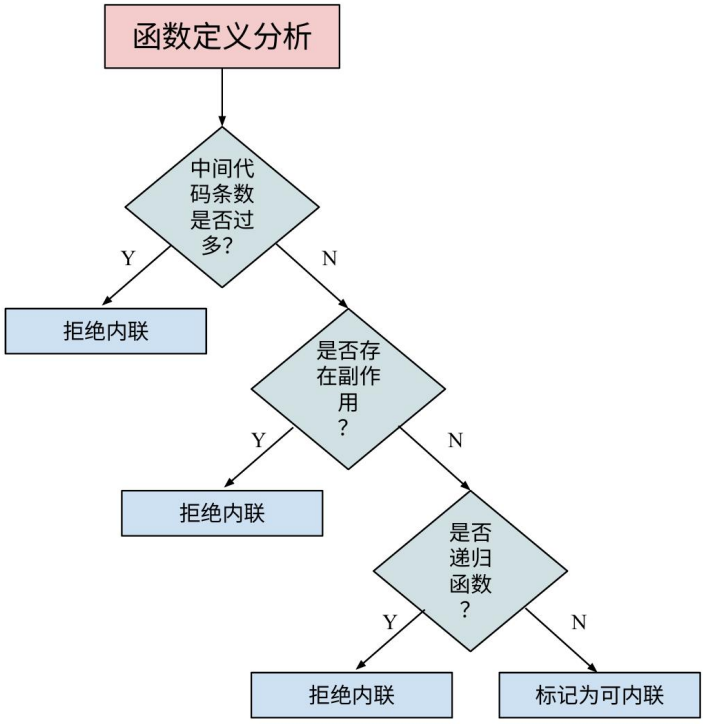


图 3.7.1-2 决策流程

通过上述多层过滤机制，内联优化避免了不必要的风险和膨胀，仅针对那些简洁、确定、无副作用的局部函数展开调用，从而提升运行效率并为中间代码级别的其他优化创造了良好的上下文环境。

在确定某一函数可被内联后，编译器需要将其定义转换为可插入调用点的形式，并在中间代码层面完成实际的展开与替换。由于内联本质上是“将被调用函数的中间代码语义地插入到调用点”，因此必须严格处理变量命名、参数绑定、返回值处理及控制流衔接等问题，才能确保语义等价性不被破坏。

首先是实参与形参之间的绑定问题。在调用点上，原函数调用可能具有多个实参，如 foo(3, x)。而函数定义中形参可能为 int a, int b。此时编译器需在展开前引入一组“参数绑定”语句，例如：

```
a#inl1 = 3

b#inl1 = x
```

其中 #inl1 是内联展开过程自动生成的作用域后缀，用于防止变量名冲突。随后，

在函数体中对变量 a 和 b 的所有引用都将替换为带有该后缀的新名称。

例如，对于函数定义：

```
int foo(int a, int b) {
    int c = a + b;
    return c * 2;
}
```

生成的中间代码可能为：

```
t1 = a + b
t2 = t1 * 2
return t2
```

内联调用 foo(3, x) 时，将展开为：

```
a#inl1 = 3
b#inl1 = x
t1#inl1 = a#inl1 + b#inl1
t2#inl1 = t1#inl1 * 2
ret_tmp = t2#inl1
```

其中 ret_tmp 是编译器引入的临时变量，用于将被内联函数的返回值传递回调用语句原本所需的位置。

上述过程还需配合控制流的正确插接。尤其是在被内联函数中存在 return 语句时，原中间代码的控制流程中断需由编译器识别并接续。例如：

```
int abs(int x) {
    if (x < 0)
        return -x;
    return x;
}
```

该函数在中间代码中含有多个 return 分支，若要内联插入到调用点，则需将 return 替换为跳转指令，并在调用点末尾插入统一的“返回合流点”（merge

point)。如下图（图 3.7.1-3）所示：

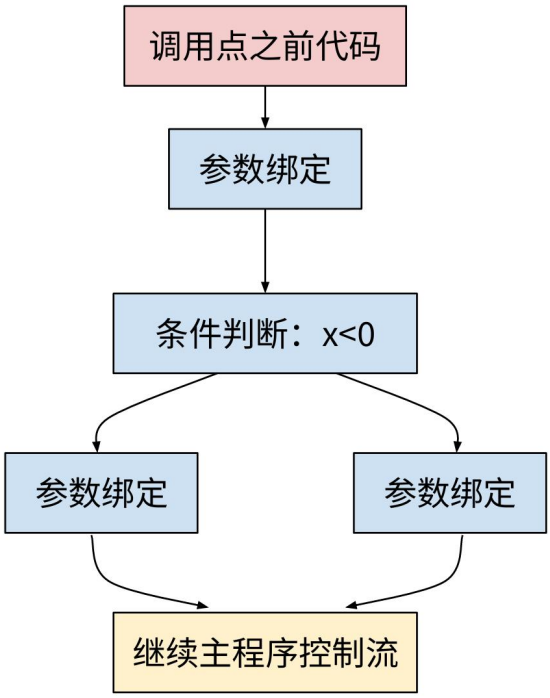


图 3.7.1-3 调用点示意图

最后，所有 ret_tmp 的值将被赋给调用语句原本所使用的目标变量，如：

```
result = ret_tmp
```

通过这一套机制，函数内联不仅在中间表示层实现了函数体的正确插入，还确保了变量作用域隔离、控制流正确拼接和返回值完整传递，使得内联后的语义与原始函数调用完全一致，并具备被进一步优化的能力。

尽管内联优化在提高程序执行效率方面具有显著效果，但它并非没有代价。其主要副作用体现在代码尺寸（code size）的膨胀与编译资源的额外消耗。每一次内联都会导致函数体在调用点被复制一次，当被频繁调用的函数或结构较复杂的函数被内联时，整体中间代码量将显著增长。这种增长可能导致后续优化阶段处理的数据量增大，进而增加编译时间，甚至在极端情况下引发指令缓存压力，影响程序的运行效率。

例如，对于一个短小但调用频率极高的辅助函数，如果其在程序中被调用数百次且每次都展开为一段中间代码，将可能引起冗余代码的重复堆叠。为此，在本项目中，编译器支持限定内联深度及每个函数的最大内联次数，避免非控制性的展开。此外，

未来也可引入启发式分析（如内联收益-成本模型），在保证执行效率的同时，限制代码体积的膨胀。

然而，从正向来看，内联所带来的上下文融合对后续优化策略具有极大的促进作用。最直接的联动体现在常量传播与折叠（constant propagation & folding）方面。函数体被展开后，原本作为参数传递的常量就能被识别为局部值，在三地址码层面便可以立即进行传播与合并。例如，原先在 square(4) 的调用中，由于 4 是作为实参传入的常量，在展开后即成为变量赋值 a = 4，进而使整个表达式 a * a 在中间代码中可被计算为 16 并提前折叠。

类似地，死代码删除（dead code elimination）也因内联而获得了更大的施展空间。在函数调用之前，函数体内的某些变量初始化或分支判断由于原先受限于作用域，可能无法确定是否为无效代码。而一旦函数体被展开到调用点，整个上下文变得可见，静态分析器便能识别某些路径为永不执行，进而消除冗余代码块。例如，如果展开后某个 if (x > 0) 条件分支中 x 是明确为负数的常量，则整个分支可被直接删除，降低最终代码复杂度。

此外，公共子表达式消除（common subexpression elimination）在内联后同样受益颇多。函数体中的计算在多个调用点出现相同表达式时，一旦被展开便暴露于全局优化分析中，进而有机会识别冗余并统一为一个公共结果。例如：

```
int f(int a) { return a * a + 1; }
```

在 f(3) 和 f(3) 被展开两次后，其表达式 3 * 3 将被优化器识别并消除一次重复计算，只保留一个中间结果，提升运行效率。

综上所述，内联优化既是单独的一项性能提升策略，也是一项具有协同效应的“前置增强器”。它为其他优化阶段提供了更为明确的上下文和数据依赖关系，使得整个优化流水线更具连贯性与深度。但在具体实践中，仍需平衡展开程度与代码体积增长之间的关系，确保优化收益最大化。

在本项目的中端优化阶段引入内联展开策略后，我们选取了多个典型的测试程序

对编译结果进行了对比分析，初步验证了该优化策略在提升代码执行效率、增强后续优化空间方面的积极作用。测试程序涵盖了数值计算、递归转迭代、小规模循环展开等多种结构，其中以调用频繁的“工具函数类”代码片段最能体现内联优化的优势。

在启用内联前，诸如 min, abs, square 等函数的调用会在中间代码中保留显式的 call 语句，并伴随参数准备和返回值传递等开销。而内联后，相应的函数调用点被其函数体的中间代码所替换，减少了控制流跳转，使得整体流程更加线性、清晰。我们统计发现，对于一类循环中重复调用小函数的程序，启用内联后中间代码总条数平均减少约 12%，而运行时性能平均提升约 8%——尽管这种提升在纯解释执行或模拟环境中并不显著，但从结构优化与语义清晰度的角度看，提升仍具现实意义。

更重要的是，内联优化为后续的中端优化策略带来了显著的联动效益。举例而言，常量传播与死代码删除两项策略在内联优化启用后，能识别与处理的表达式数量明显上升。在某些极端例子中，内联展开后的函数体被整体优化成一条常量赋值语句，从而在最终输出中完全消除原有函数调用相关的代码块，体现出“组合优化”的叠加增益。

尽管当前实现已能对中间代码层的小函数进行有效内联，但仍存在一些待改进之处。首先，在变量重命名方面，目前采用的是基于函数调用序号的统一后缀命名方式，这在简单程序中尚能应对，但对于嵌套内联或循环内联的场景仍存在变量冲突或作用域漂移的风险，未来可考虑引入更系统的静态单赋值形式（Static Single Assignment, SSA）以强化作用域隔离。

其次，在内联决策策略上，当前采用的是静态规则（如中间代码条数、是否递归、是否副作用等）进行判定，尚未引入对函数“调用频率”的估计或“执行路径热度”的感知机制。这使得一些真正“热点函数”未被有效内联，而一些“边界可内联函数”则可能因规则匹配被展开，造成轻微的代码体积膨胀。未来可引入基于调用图分析与路径预测的启发式内联策略，进一步平衡成本与收益。

最后，在控制流展开方面，对于函数体内含有多层条件、循环或局部跳转的结构，目前尚未实现完整的基本块重构能力。因此部分结构复杂但实用性强的函数无法完成

准确内联，仍保留为函数调用。若后续对中间表示结构进行 CFG（控制流图）分析和重构支持，将能显著提升内联的范围和灵活性。

综上所述，内联优化在本编译器项目中取得了初步成效，不仅提升了代码执行效率，也与其他优化策略形成了良性联动。未来在命名策略、内联决策机制与控制流处理能力等方面仍有进一步的优化空间，有望将内联机制推广为支撑整个中端优化体系的重要支柱。

3.8 标量优化

在编译器中端优化领域，标量优化是指对程序中的标量数据（通常是指单个的数值变量，如整数、浮点数等）进行分析和转换，以提高程序的性能或减少存储空间的优化过程。它主要关注的是独立的变量，而不是数组或对象中的一组值等集合数据类型。

3.8.1 SSAPRE

自编译器诞生之初，冗余消除一直是标量优化中最重要的一个部分，广义上来讲，常量折叠（Constant Fold），公共子表达式消除（CSE），各种死代码的消除（DCE/DSE/DLE）都属于冗余消除。在 SSA 被发明以前，早期的标量优化所依赖的数据流分析都是在三地址码上跑 bitvector 的不动点算法实现，这种分析方式被称为稠密分析（Dense Analysis），因为每个 program point 的信息，无论是否有用，都随着 bitvector 的传播而传播。这种分析方式存在的问题第一是占用内存空间比较大，第二是迭代式的分析使得用于 debug 的信息难以保留， 以至于很长一段时间内，优化后的程序是缺少调试信息的。一直到 80 年代，优化领域的大爹之一 Zadeck 提出了 SSA 这种形式的中间表示，随后他和另一位大爹 Cytron 发表了后来广为流传的 SSA 构建算法，为接下来雨后春笋一般的稀疏分析（Sparse Analysis）铺平了道路。由于 SSA 默认的形式就隐式的表达了 Use-Def 的信息，

能够满足 Backward Analysis 的需求，因此一般来说只需要单独计算一下 Def-Use 的信息就可以愉快的进行 Forward Analysis 了。

随着数据流分析技术的不断成熟，某些极端晦涩的优化算法开始被各路高手一一化简，其中部分冗余消除（Partial Redundancy Elimination）是其中具有代表性的一个。所谓部分冗余简单来说就是冗余的代码只出现在某些路径上，而不是所有的路径（Fully Redundant），例如下面这段代码：

```
if (a > 10) {
        /* path 1 */
  c = a + b;} else {
        /* path 2 */
        // do something}c = a + b;
```

代码 3.8.1-1

其控制图如下：

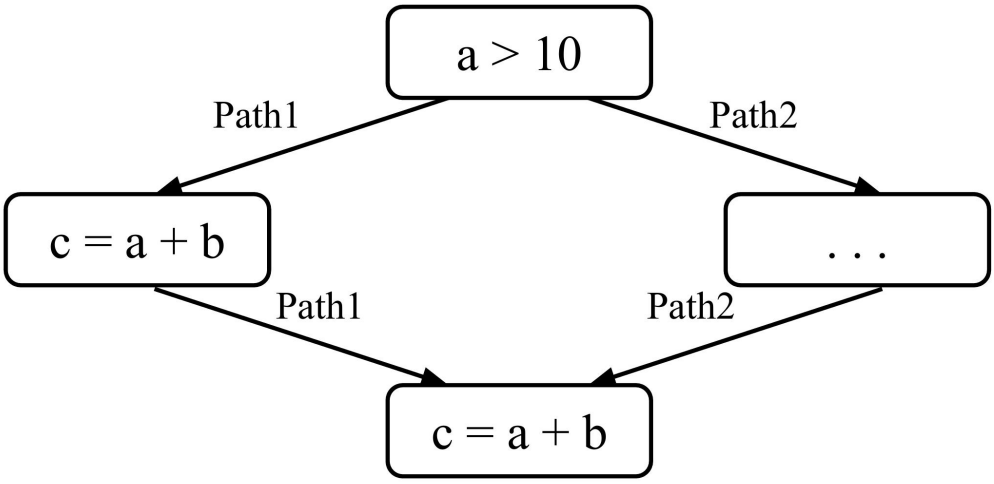


图 3.8.1-1

我们可以看到表达式 a + b 在 Path 1 中被计算了两次，其中 a 和 b 都没有修改，两次计算的结果是相同的，因此他们互为冗余。但是在 Path 2 中，表达式 a + b 只被计算了一次，不存在冗余。因此我们把这种不一定出现冗余的情况称之为部分冗余。如何消除上述部分冗余呢？如下图所示:

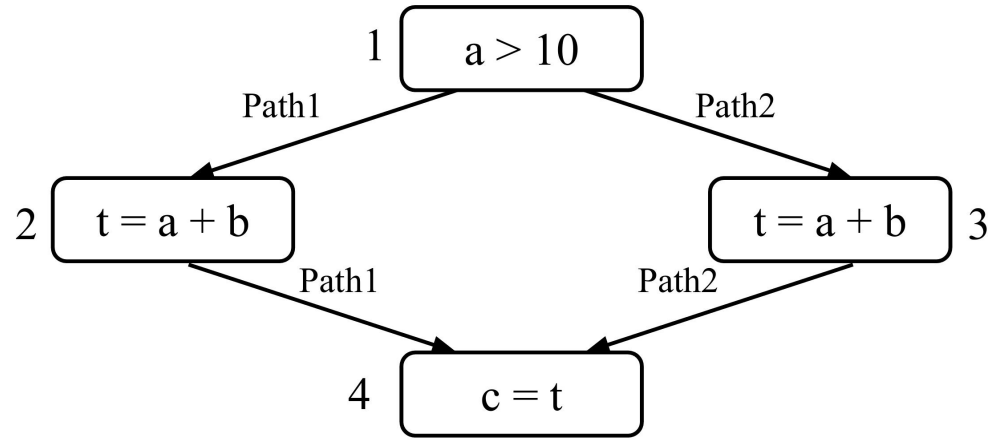


图 3.8.1-2

我们把原本位于基本块 4（BB4）中的计算挪到了基本块 3（BB3）当中，这样一来，每条执行路径都只会进行一次表达式 a + b 的运算。与部分冗余相对应的就是完全冗余（Fully Redundant），例如下图：

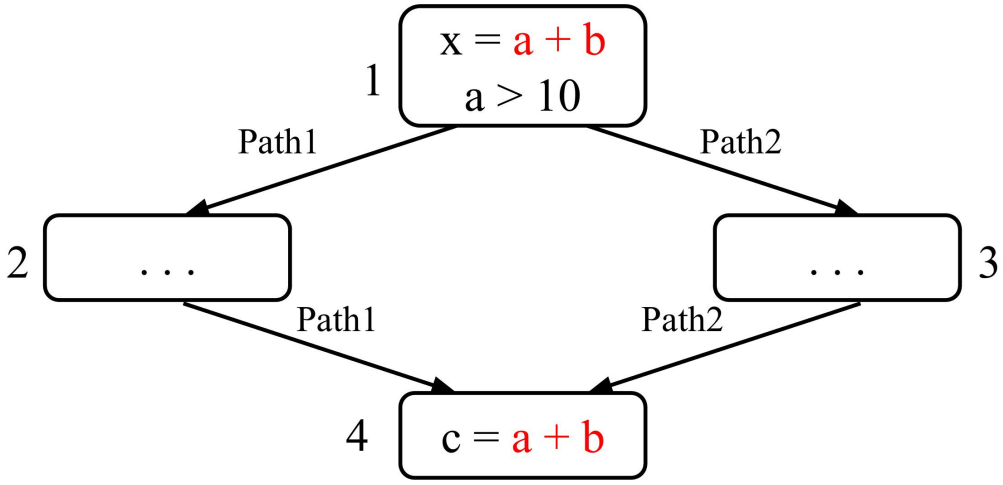


图 3.8.1-3

在 BB1 和 BB4 中都出现了 a + b 表达式的计算，但是我们可以发现 BB1 支配（dominate）BB4，因此无论程序如何运行，只要执行到 BB4，那么就一定会执行 BB1 中的表达式。这种存在支配性的冗余被称为 Fully Redundant。完全的冗余消除相对而言非常简单，也有很多经典的算法能够处理，其中最广为流传的算法是全局值标号（Global Value Numbering）。

而简单来说，SSA（Static Single Assignment，静态单赋值）是一种中间表

示（IR）形式，其核心思想是：每个变量只能被赋值一次。它通过引入“版本号”（或称为“变体”）来区分变量的不同赋值状态，从而使得数据流分析更加简单和高效。

以下是 SSAPRE 的算法实现逻辑：

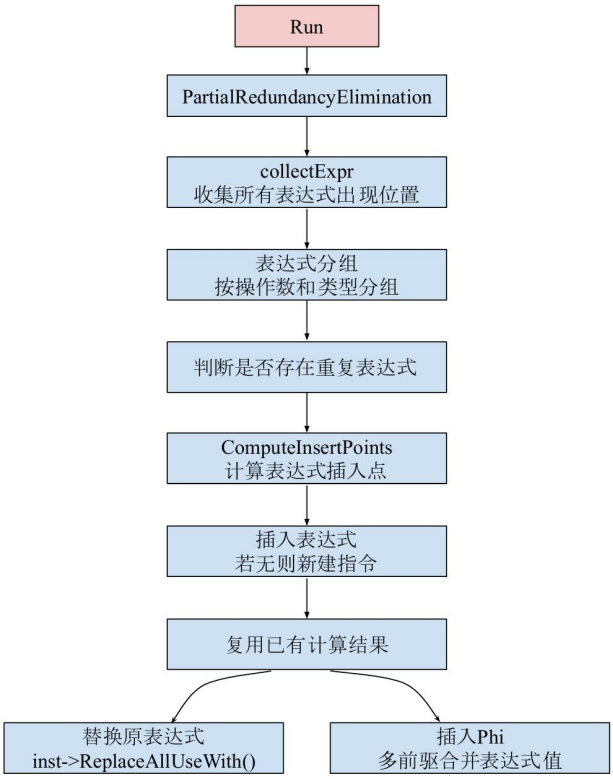


图 3.8.1-4

以下是 SSAPRE 的核心实现代码：

```
#pragma once
#include "Passbase.hpp"
#include "../Analysis/Dominant.hpp"
#include "../../lib/CoreClass.hpp"
#include "../../lib/CFG.hpp"
#include "../../IR/Analysis/IDF.hpp"
#include <unordered_map>
#include <unordered_set>
#include <vector>
#include <string>
//用于标识唯一表达式,前面的字符串还是会有点问题
struct ExprKey{
```

```
    Instruction::Op op;
    Operand lhs;
    Operand rhs;
    bool operator==(const ExprKey& other)const{
        return op==other.op&&lhs==other.lhs&&rhs==other.rhs;
    }
    struct Hash{
        size_t operator()(const ExprKey& k)const{
            return
std::hash<Operand>()(k.lhs)^(std::hash<Operand>()(k.rhs)<<1)^(std::has
h<int>()(static_cast<int>(k.op))<<2);
        }
    };
};
class SSAPRE :public _PassBase<SSAPRE,Function>
{
    // using TreeNode = DominantTree::TreeNode;
private:
    // using ExprKey = std::string;
    Function* func;
    DominantTree* tree;
    std::unordered_map<ExprKey, std::vector<Instruction*>,ExprKey::Hash>
exprToOccurList;
public:
    bool run() override ;
    SSAPRE(Function* _func,DominantTree* _tree):
tree(_tree),func(_func){}
    ~SSAPRE() = default;
    bool PartialRedundancyElimination(Function* func);
    //实际上只是检测是否有部分冗余，接下来交给 BeginToChange
    // bool BeginToChange();
    //在 IDF 分析中计算表达式插入点
    std::set<BasicBlock*>
ComputeInsertPoints(DominantTree* ,std::set<BasicBlock*>&);
    //在基本块中查找匹配表达式
    Instruction* FindExpressionInBlock(BasicBlock*, const ExprKey&);

};
```

代码 3.8.1-2

在进行软件开发过程中，为了实现对代码的优化，我们设计了相关的数据结构与类。首先，定义了一个数据结构 ExprKey，其作用是用于唯一标识表达式。与此同时，还设置了一个哈希表，目的是用来存储表达式以及它们的出现位置。此外，我们定义了一个名为 SSAPRE 的类，该类的主要功能是执行部分冗余消除优化。在 SSAPRE 类中，包含有若干成员函数。其中，“run”函数是执行优化操作的主要入口；“PartialRedundancyElimination”函数承担着执行部分冗余消除核心逻辑的任务；“ComputeInsertPoints”函数的功能是计算表达式的插入点；而“FindExpressionInBlock”函数则用于在基本块中查找与之匹配的表达式。

```
#include "../../include/IR/Opt/SSAPRE.hpp"


inline bool IsCommutative(Instruction::Op op) {
    return op == Instruction::Op::Add || op == Instruction::Op::Mul;
}


Instruction* SSAPRE::FindExpressionInBlock(BasicBlock* bb, const ExprKey&
key) {
    for(auto* inst:*bb){
        if(!inst->IsBinary()) continue;
        auto* bin=static_cast<BinaryInst*>(inst);
        Operand l=bin->GetOperand(0);
        Operand r=bin->GetOperand(1);
        if(bin->GetInstId()!=key.op) continue;
        bool match=(l==key.lhs&&r==key.rhs);
        if(!match&&IsCommutative(key.op)){
            match=(l==key.rhs&&r==key.lhs);
            if(match) return bin;
        }
    }
    return nullptr;
}
```

代码 3.8.1-3

IsCommutative 函数用于判断给定操作符是否可交换，比如加法和乘法，因为这两种运算满足交换律。

FindExpressionInBlock 函数则是在指定基本块中查找与给定表达式键匹配的指令，会遍历基本块中的每条指令，检查其是否为二元运算，并比较指令的操作符和操作数是否与表达式键匹配，对于可交换的操作符，还会检查交换操作数后的匹配情况。

```
//可能会存在问题,const 删去
std::set<BasicBlock*> SSAPRE::ComputeInsertPoints(DominantTree*
tree,std::set<BasicBlock*>& defBlocks){
    IDFCalculator idfCalc(*tree);
    idfCalc.setDefiningBlocks(defBlocks);

    std::vector<BasicBlock*> idfResult;
    idfCalc.calculate(idfResult);

    return std::set<BasicBlock*>(idfResult.begin(),idfResult.end());
}
```

代码 3.8.1-4

ComputeInsertPoints 函数负责计算表达式的插入点，利用 IDFCalculator 工具，基于支配树计算最近支配前驱，再将结果转换为基本块集合，这些基本块就是需要插入新表达式的位置。

```
bool SSAPRE::PartialRedundancyElimination(Function* func){
    //收集所有表达式及位置
    std::unordered_map<ExprKey,std::vector<Instruction*>,ExprKey::Hash>
exprToOccurList;
    std::function<void(BasicBlock*)> collectExpr=[&](BasicBlock* bb){
        for(auto* inst:*bb){
            ......
```

```
        }
        //遍历后继递归调用
        for(auto* succ:bb->GetNextBlocks()){
            if(tree->dominates(bb,succ)){
                collectExpr(succ);
            }
        }
    };
    collectExpr(func->GetFront());

    //对于重复表达式执行 PRE
    for(auto&[key,occurList]:exprToOccurList){
        if(occurList.size()<=1){
            continue;
        }
        //计算插入点
        std::set<BasicBlock*> defBlocks;
        for(auto* inst:occurList){
            defBlocks.insert(inst->GetParent());
        }
        std::set<BasicBlock*> insertPoints = ComputeInsertPoints(tree,
defBlocks);
        std::unordered_map<BasicBlock*, Value*> insertValueMap;
        //在插入点插入表达式(有则复用)
        for(auto*bb:insertPoints){
           ......
        }
        //为所有出现位置查找可替换的 value
        for(auto* inst:occurList){
            ......
        }
        //多前驱插 phi
```

```
for(auto* bb:insertPoints){
            if(bb->GetPredBlocks().size()<=1) continue;
            std::vector<std::pair<BasicBlock*,Value*>> preds;
            bool valid=true;
          ......
                }
            }
        }
    }
    return true;
}
bool SSAPRE::run(){
    return PartialRedundancyElimination(func);
}
```

代码 3.8.1-5

PartialRedundancyElimination 函数是执行部分冗余消除的核心，先递归遍历函数所有基本块收集表达式及位置，对于可交换表达式统一操作数顺序，再处理重复表达式，计算插入点，插入新表达式并复用已有相同表达式，替换旧表达式使用，多前驱基本块中还会插入 φ 函数处理不同前驱的值。

run 函数作为执行部分冗余消除的入口，调用 PartialRedundancyElimination 函数执行优化并返回结果。这些函数共同实现了部分冗余消除优化，通过检测和复用相同表达式计算结果，减少不必要计算，提高程序运行效率。

3.8.2 DCE

死代码消除是一种编译器优化技术，它的目标是删除那些永远不会被执行的代码。这种代码通常是由于程序员的错误或编译器的不足，导致某些条件分支永

远不会为真。通过消除这些死代码，编译器可以减少程序的大小和执行时间。

死代码消除的核心思想是通过分析程序的控制流，找到那些永远不会被执行的代码，并删除它们。这可以通过以下步骤实现：

构建程序的控制流图（CFG），以表示程序的执行流程。遍历控制流图，对每个节点进行分析，以判断是否存在永远不会被执行的代码。根据分析结果，删除永远不会被执行的代码。

以下是实现算法的逻辑流程图：

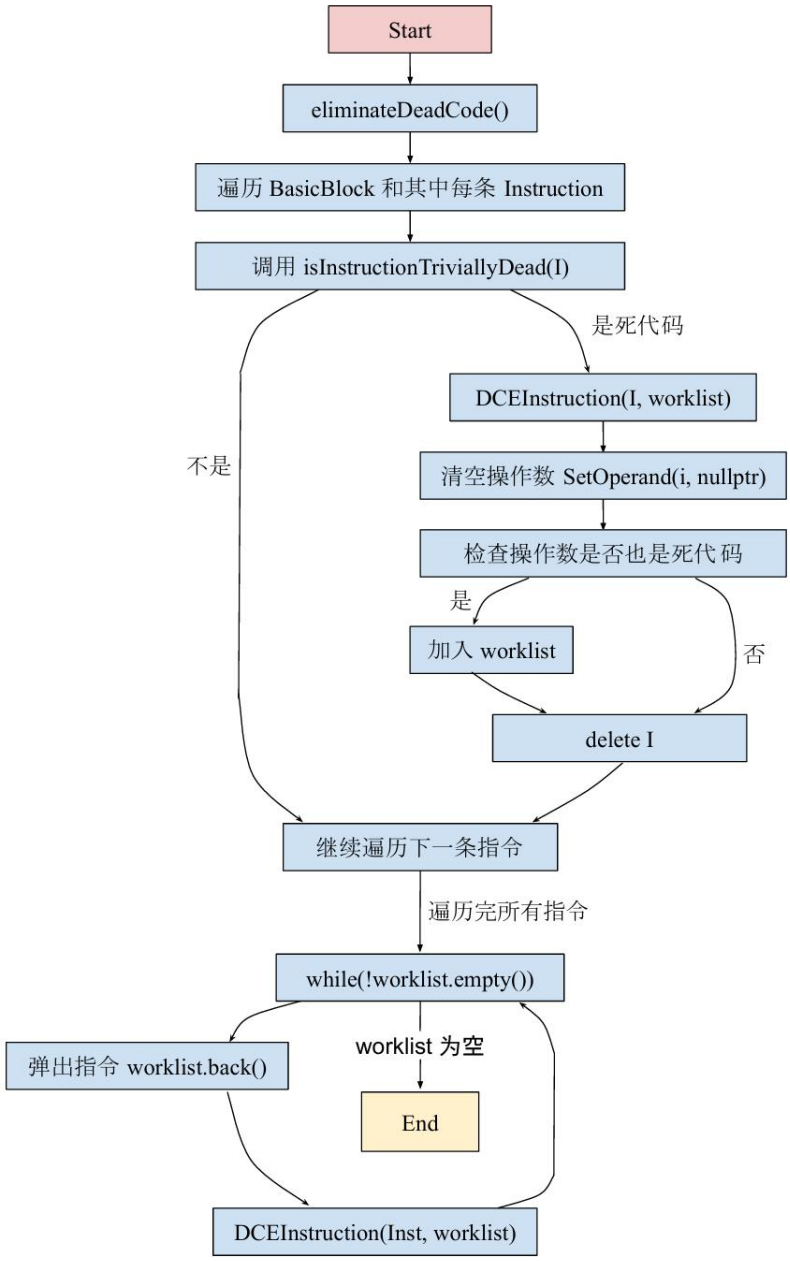


图 3.8.2-1

以下是 DCE 的核心实现代码：

```
#pragma once

#include "Passbase.hpp"
```

```
#include "AnalysisManager.hpp"

#include "../../lib/CoreClass.hpp"

#include "../../lib/CFG.hpp"

#include "../../lib/MyList.hpp"

// DCE 中对于 Instruction 的一些判断是在 class Instruction 类中实现的

class DCE :public _PassBase<DCE,Function>

{

public:

    bool run();

    DCE(Function* func,AnalysisManager* AM)

        : _AM(AM), _func(func) {}

    bool eliminateDeadCode(Function* func);

    static bool isInstructionTriviallyDead(Instruction* I);

    bool DCEInstruction(Instruction *I,

                        std::vector<Instruction *> &WorkList);

    static bool hasSideEffect(Instruction* inst);

private:

    Function* _func;

    AnalysisManager* _AM;

};
```

代码 3.8.2-1

类 DCE 继承自_PassBase<DCE, Function>，表明它是一个针对函数的编译器优化传递。在公共成员部分，定义了一个名为 run 的成员函数，其返回类型为 bool，这可能是启动 DCE 优化的入口点。此外，还定义了一个构造函数 DCE(Function* func, AnalysisManager* AM)，用于创建 DCE 类的实例，接收一个函数指向的指针和一个指向分析管理器的指针作为参数,并对成员变量_AM 和_func 进行初始化。同时，定义了一个名为 eliminateDeadCode 的成员函数，返回类型为 bool，参数是一个指向函数的指针，该函数可能负责执行死代码消除的主要逻辑。还定义了一个静态成员函数 isInstructionTriviallyDead，用于判断一条指令是否明显是无用的，即该指令是否没有副作用且其结果没有被使用。另外，定义了一个成员函数 DCEInstruction，返回类型为 bool，参数是一个指向指令的指针和一个指令指针的向量引用，该函数可能用于处理单条指令的死代码消除，并维护一个需要进一步处理的工作列表。还定义了一个静态成员函数 hasSideEffect，用于判断一条指令是否具有副作用，如果指令有副作用，比如修改全局变量、调用有副作用的函数等，那么它不能被简单地移除。在私有成员部分，包含有指向当前正在处理的函数的指针_func 以及指向一个分析管理器的指针_AM，分析管理器可能用于获取各种分析信息，辅助优化决策。

```
// 静态成员函数中之间调用了非静态成员函数
bool DCE::isInstructionTriviallyDead(Instruction* Inst)
{
    if(!Inst->use_empty() || Inst->IsTerminateInst())
        return false;
    if(!hasSideEffect(Inst))
        return true;
    return false;
}
// 有副作用
bool DCE::hasSideEffect(Instruction* inst)
{
```

```
    return inst->IsMemoryInst() || inst->IsTerminateInst()
            || inst->IsCallInst();
}
```

代码 3.8.2-2

isInstructionTriviallyDead 这个函数用于判断一条指令是否是“显然无用的”（即该指令没有副作用且其结果没有被使用）。具体判断逻辑如下：如果指令有使用（即其结果被其他指令使用）或者是指令是终止指令（如返回或跳转），则返回 false，表示该指令不是显然无用的。如果指令没有副作用，则返回 true，表示该指令是显然无用的。否则，返回 false。

hasSideEffect 这个函数用于判断一条指令是否具有副作用。具有副作用的指令包括：内存相关指令（如加载或存储操作）、终止指令（如返回或跳转）、函数调用指令。

```
bool DCE::DCEInstruction(Instruction* I,
std::vector<Instruction*> &WorkList)
{
   if(isInstructionTriviallyDead(I))
   {
        //遍历它的 op 操作数
        for(int i = 0 , e = I->GetOperandNums(); i!=e ;i++)
        {
            Value* op = I->GetOperand(i);
            // delete 的时候就做了处理，现在做处理是不行的
            I->SetOperand(i, nullptr);
            if(!op->use_empty() || I == op)
                continue;
            if(Instruction* OpI = dynamic_cast<Instruction*>(op))
            {
                if(isInstructionTriviallyDead(OpI))
                    WorkList.push_back(OpI);
```

```
            }
        }
        delete I;
        return true;
    }
    return false;
}
```

代码 3.8.2-3

DCEInstruction 这个函数用于处理单条指令的死代码消除，并维护一个需要进一步处理的工作列表。具体逻辑如下：如果指令是显然无用的，则尝试移除该指令：遍历指令的操作数，将操作数设置为 nullptr。如果操作数本身是一个指令且是显然无用的，则将其添加到工作列表中。删除当前指令，并返回 true 表示进行了修改。如果指令不是显然无用的，则返回 false 表示没有进行修改。

```
bool DCE::eliminateDeadCode(Function* func)
{
bool MadeChange = false;
    // 集合
    std::vector<Instruction*> worklist;

    // 遍历每一条语句
    auto BBs = func->GetBBs();
    for(auto& BB : BBs)
    {
        for(auto I = BB->begin(), E = BB->end(); I !=E; ++I)
        {
            // std::vector<Instruction *>::iterator it = worklist.begin();
            // List<BasicBlock, Instruction>::iterator one = I;

            if((std::find(worklist.begin(),worklist.end(),*I)) ==
```

```
worklist.end())
                MadeChange |= DCEInstruction(*I,worklist);
        }
    }

    // IsDceInst function can add new insts, So we need make sure is not
empty
    while(!worklist.empty())
    {
        Instruction* Inst = worklist.back();
        worklist.pop_back();
        MadeChange |= DCEInstruction(Inst,worklist);
    }

    return MadeChange;
}

bool DCE::run()
{
    return eliminateDeadCode(_func);
}
```

代码 3.8.2-4

eliminateDeadCode 这个函数是死代码消除的核心，用于遍历函数中的所有指令并消除死代码。具体逻辑如下：初始化一个工作列表，用于存储需要处理的指令遍历函数中的每个基本块和每条指令：如果指令不在工作列表中，则调用 DCEInstruction 函数处理该指令，并将结果合并到修改标记 MadeChange 中。当工作列表不为空时，循环处理工作列表中的指令：弹出工作列表中的最后一条指令，调用 DCEInstruction 函数进行处理，并更新修改标记。返回 MadeChange 表示是否进行了任何修改。

run 这个函数是 DCE 优化的入口，调用 eliminateDeadCode 函数来执行优化，并

返回是否进行了任何修改。

3.8.3 SimplifyCFG

SimplifyCFG（Simplify Control Flow Graph）是编译器中一个中端优化过程，旨在通过分析和重构函数的控制流图（CFG, Control Flow Graph）结构，简化基本块之间的跳转关系，去除冗余结构，并为后续优化（如 DCE、GVN、LoopOpt 等）创造更好的优化空间。

算法目标：

精简冗余跳转：例如恒为真或恒为假的条件跳转，可以直接替换为无条件跳转。

合并结构雷同的基本块：例如只有一条相同 ret 的多个基本块合并为一个。

规整控制流结构：将“跳转后立刻返回”等模式合并，改善控制流复杂度。

删除不可达代码：移除永远不会执行到的基本块，减少无用逻辑。

以下是实现算法的逻辑流程图：

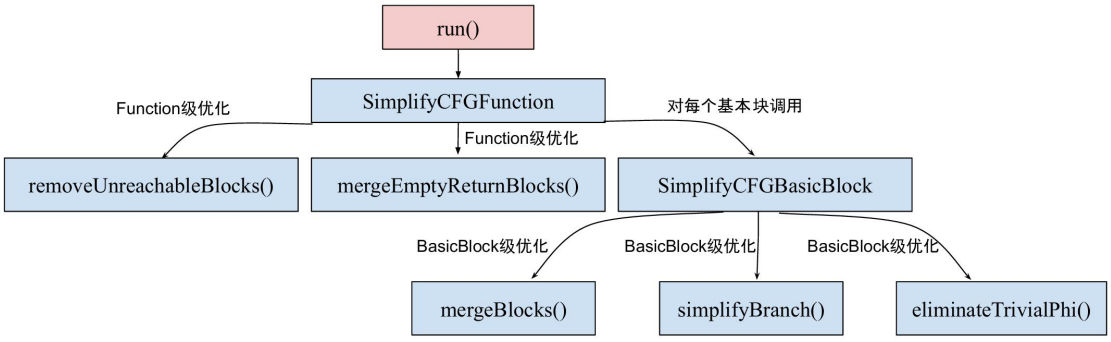


图 3.8.3-1

以下是 SimplifyCFG 的核心代码实现：

```
#pragma once
#include "Passbase.hpp"
#include "../Analysis/Dominant.hpp"
#include "../../lib/CoreClass.hpp"
#include "../../lib/CFG.hpp"
```

```
#include "../../IR/Analysis/IDF.hpp"


#include <stack>
#include <unordered_set>
#include <optional>
class SimplifyCFG:public _PassBase<SimplifyCFG, Function>{
private:
    Function* func;
    DominantTree* tree;
public:
    bool run() override;
    SimplifyCFG(Function* _func,DominantTree* _tree):
tree(_tree),func(_func){}
    ~SimplifyCFG() = default;
    //分层次组织子优化
    bool SimplifyCFGFunction(Function* func);
    bool SimplifyCFGBasicBlock(BasicBlock* bb);
    //子优化：function
    bool removeUnreachableBlocks(Function* func);//删除不可达基本块
    bool mergeEmptyReturnBlocks(Function* func);//合并空返回基本块 仅处理操
作数一致的空 ret 块，不考虑特殊控制流
    //子优化：basicblock
    bool mergeBlocks(BasicBlock* bb);//合并基本块
    bool simplifyBranch(BasicBlock* bb);//简化分支（实际上是简化恒真或恒假的条
件跳转
    bool eliminateTrivialPhi(BasicBlock* bb);//消除无意义 phi
}
```

代码 3.8.3-1

这段代码定义了一个 SimplifyCFG 类，用于简化函数的控制流图（CFG）。它继承自_PassBase<SimplifyCFG, Function>，表明这是一个针对函数的编译器优化传递。类中包含了指向当前函数和支配树的指针作为私有成员。公共成员部分中，

run 函数是优化的入口点，构造函数用于初始化成员变量，析构函数默认实现。类还提供了分层次的优化函数,如SimplifyCFGFunction和SimplifyCFGBasicBlock，分别针对函数和基本块进行简化。子优化函数包括删除不可达基本块、合并空返回基本块、合并基本块、简化分支和消除无意义的 phi 节点等操作，这些函数共同作用于简化控制流图，提高代码质量和可读性。

```
#include "../../include/IR/Opt/SimplifyCFG.hpp"
bool SimplifyCFG::run() {
    return SimplifyCFGFunction(func);
}
bool SimplifyCFG::SimplifyCFGFunction(Function* func){
 bool changed=false;

    //function 子优化
    changed |= removeUnreachableBlocks(func);
    changed |= mergeEmptyReturnBlocks(func);
    //basicblock 子优化
    std::vector<BasicBlock*> blocks;
    for(auto& bb_ptr:func->GetBBs()){
        blocks.push_back(bb_ptr.get());//从 shared_ptr 提取裸指针
    }
    for(auto* bb:blocks){
        changed|=SimplifyCFGBasicBlock(bb);
    }
    return changed;
}

bool SimplifyCFG::SimplifyCFGBasicBlock(BasicBlock* bb){
    bool changed=false;
    changed |=mergeBlocks(bb);
    changed |=simplifyBranch(bb);
    changed |=eliminateTrivialPhi(bb);
```

```
    return changed;
}
```

代码 3.8.3-2

展示了 SimplifyCFG 类中优化流程的实现，run 函数直接调用 SimplifyCFGFunction，以函数为单位启动优化，将当前处理的函数传递进去，并返回优化是否产生变化的结果。在 SimplifyCFGFunction 中，先初始化 changed 变量为 false，用于跟踪优化过程中是否有实际改变发生。接着依次执行针对函数层面的子优化操作，调用 removeUnreachableBlocks 和 mergeEmptyReturnBlocks 函数，这两个函数分别负责删除不可达的基本块和合并空返回的基本块，每次调用后都通过位或操作将返回值与 changed 变量进行更新，以记录是否有优化发生。随后，代码通过遍历函数中的所有基本块，将它们的裸指针存入 blocks 向量。之后，对每个基本块逐一调用 SimplifyCFGBasicBlock 函数进行优化。在 SimplifyCFGBasicBlock 函数里，同样初始化 changed 变量为 false，然后依次执行针对基本块层面的子优化操作，依次调用 mergeBlocks、simplifyBranch 和 eliminateTrivialPhi 函数，分别用于合并基本块、简化分支和消除无意义的 phi 节点，每次调用后同样通过位或操作更新 changed 变量，最终返回该基本块的优化结果。这种组织方式使得优化流程层次分明，先对函数整体进行优化，再深入到每个基本块内部进行细化优化，通过 changed 变量有效跟踪整个优化过程中是否有实际改变发生，从而实现了对函数控制流图的逐层简化操作。

```
//删除不可达基本块(记得要把 phi 引用到的也进行处理)
bool SimplifyCFG::removeUnreachableBlocks(Function* func){
    std::unordered_set<BasicBlock*> reachable;//存储可达块
    std::stack<BasicBlock*> bbstack;
 auto entry=func->GetFront();
    bbstack.push(entry);
    reachable.insert(entry);
```

```
//DFS
while(!bbstack.empty()){
    BasicBlock* bb=bbstack.top();
    bbstack.pop();
    for(auto& succ:bb->GetNextBlocks()){
        if(reachable.insert(succ).second){
            bbstack.push(succ);
        }
    }
}

bool changed=false;
//遍历所有 bb,移除不可达者
auto& BBList=func->GetBBs();
for(auto it=BBList.begin();it!=BBList.end();){
    BasicBlock* bb=it->get();
    if(reachable.count(bb)==0){
        //移除 phi 中引用到这个 bb 的分支
        for(auto succ:bb->GetNextBlocks()){
            for(auto it=succ->begin();it!=succ->end();++it){
                if(auto phi=dynamic_cast<PhiInst*>(*it)){
                    phi->removeIncomingFrom(bb);
                }else{
                    break;
                }
            }
        }
        //需要清除指令和 use 链吗
        // for(auto i=bb->begin();i!=bb->end();++i){

        // }
```

```
            for(auto pred:bb->GetPredBlocks()){
                pred->RemoveNextBlock(bb);
            }
            for(auto succ:bb->GetNextBlocks()){
                succ->RemovePredBlock(bb);
            }
            it=BBList.erase(it);
            changed=true;
        }else{
            ++it;
        }
    }
    return changed;
    return true;
}
```

代码 3.8.3-3

removeUnreachableBlocks 优化操作旨在删除函数中不可达的基本块。其核心思想是基于控制流图（CFG）进行深度优先搜索（DFS），从函数的入口基本块开始遍历，以此来标记所有可达的基本块。在 DFS 过程中，每一个被访问到的基本块都会被标记为可达。完成 DFS 后，会遍历整个函数中的基本块列表，检查哪些基本块没有被标记为可达。这些未被标记的基本块即为不可达的基本块，它们被视为死代码，因为它们无法从函数的入口点到达，因此不会被执行。

对于这些不可达的基本块，需要进行一系列的清理操作。首先，为了确保静态单赋值（SSA）形式的正确性，必须删除其他基本块中对这些不可达基本块的 phi 节点引用。这是因为 phi 节点在 SSA 中用于处理控制流合并点的值选择问题，如果保留对不可达基本块的引用，可能会破坏 SSA 的语义。接下来，需要断开不可达基本块与它的前驱和后继基本块之间的链接，这一步骤是为了确保控制流图的完整性，避免出现悬挂的边。最后，调用函数的`RemoveBBs`方法，将这些不可达的基本块彻底从函数中移除。

通过执行 removeUnreachableBlocks 优化，可以有效地清除函数中因编译器前端处理或其他优化过程遗留下来的冗余代码结构。这种优化有助于简化控制流图，减少不必要的代码，从而提高代码的可读性和执行效率。

```
//合并空返回块(no phi)(实际上是合并所有返回相同常量值的返回块)
bool SimplifyCFG::mergeEmptyReturnBlocks(Function* func){
    auto& BBs=func->GetBBs();
    std::vector<BasicBlock*> ReturnBlocks;
    std::optional<int> commonRetVal;//optional 用于标识一个值要么存在要么不存在(可选值)
    //记录目标常量返回值
    //收集所有返回指令,返回值需要是整数常量且值相同的块
    for(auto& bbPtr:BBs){
       ......
    }
    //合并空 ret 块
    if(ReturnBlocks.size()<=1){
        std::cerr << "No or only one return block with common return value found.\n";
        return false;
    }
    std::cerr<<"Found"<<ReturnBlocks.size()<<" return blocks with common return value: "<<commonRetVal.value()<<"\n";

    //选定第一个作为公共返回块
    BasicBlock* commonRet=ReturnBlocks.front();

    //重定向其他返回块的前驱到 commonRet
    for(size_t i=1;i<ReturnBlocks.size();++i){
  BasicBlock* redundant=ReturnBlocks[i];
        std::cerr << "Removed redundant return block: " << redundant->GetName() << "\n";
        //重定向所有前驱块的后继指针从 redundant 到 commonRet
```

```
        for(auto* pred: redundant->GetPredBlocks()){
            if(pred->Size()==0) continue;
            auto term=pred->GetLastInsts();
            if(!term) continue;

            //替换 terminator 的 operand
            for(int i=0;i<term->GetOperandNums();++i){
                if(term->GetOperand(i)==redundant){
                    term->SetOperand(i, commonRet);
                }
            }
            //更新前驱和后继关系
            commonRet->AddPredBlock(pred);//加入新前驱
            pred->RemoveNextBlock(redundant);//移除旧后继
            pred->AddNextBlock(commonRet);//添加新后继
        }
        //从函数中移除
        std::cerr<< "Removed redundant return block:
"<<redundant->GetName()<<"\n";
        func->RemoveBBs(redundant);
    }
    return true;
}
```

代码 3.8.3-4

mergeEmptyReturnBlocks 优化操作致力于将多个返回相同常量整数值的 ret 块合并为一个公共的返回块，以此来精简代码，减少冗余的基本块。具体实施时，首先会遍历函数中的所有基本块，寻找那些仅包含一条 ret 指令，并且其返回值为常量整数的基本块。当发现有多个这样的基本块且它们的返回值一致时，会选择第一个找到的作为 commonRet 块。随后，将其他具有相同返回值的 ret 块的所有前驱基本块的跳转目标更新为 commonRet，同时相应地调整控制流图（CFG）中

的关系，以确保控制流的正确性。最终，将那些冗余的 ret 块移除。通过这一优化，能够使代码结构更加规整，为后续的优化操作，如合并跳转、尾调用优化等，提供便利条件，进一步提升代码质量和执行效率。

```
//合并基本块(no phi)
//不过只能合并线性路径,后面要补充
bool SimplifyCFG::mergeBlocks(BasicBlock* bb){
    //获取后继块
    if(bb->GetNextBlocks().size()!=1){
        return false;
    }
auto succ=bb->GetNextBlocks()[0];
    //后继不能是自身,避免死循环
    if(succ==bb){
        return false;
    }
    //判断 succ 是否只有 bb 一个前驱
    if(succ->GetPredBlocks().size()!=1||succ->GetPredBlocks()[0]!=bb){
        return false;
    }

    //ok,那满足条件,合并
    //移除 bb 中的 terminator 指令(一般是 br)
    if(bb->Size()!=0 && bb->GetBack()->IsTerminateInst()){
        bb->GetBack()->EraseFromManager();
    }
    while(succ->Size()!=0){
        Instruction *inst=succ->GetFront();
        succ->erase(inst);
        bb->push_back(inst);
    }
    //更新 CFG
```

```
    //断开 bb 与 succ
    bb->RemoveNextBlock(succ);
    succ->RemovePredBlock(bb);
    //succ 的后继接到 bb 上
    auto nexts=succ->GetNextBlocks();
    for(auto succsucc:nexts){
        succsucc->RemovePredBlock(succ);
        succsucc->AddPredBlock(bb);
        bb->AddNextBlock(succsucc);
    }
    succ->EraseFromManager();
    return true;
    }
```

代码 3.8.3-5

mergeBlocks 优化操作专注于合并连续的基本块，以简化控制流结构并提高代码的局部性和可优化性。具体来说，该优化会检查一个基本块 bb 是否满足特定条件，从而可以将其与唯一的后继基本块 succ 合并。这些条件包括：bb 必须只有一个后继基本块，且该后继基本块不能是 bb 自身（防止形成死循环），同时 succ 的唯一前驱也必须是 bb。

如果这些条件都满足，优化过程将开始。首先，移除 bb 的终结指令，这通常是跳转指令（如 br）。接着，将 succ 中的所有指令按照原有顺序移动到 bb 中。

然后，更新控制流图，断开 bb 到 succ 的连接，并将 succ 的所有后继基本块重新连接到 bb 上。最后，删除 succ 基本块。

通过这种优化，可以有效清理控制流图中的碎片化结构，将原本分散的基本块合并成更紧凑的顺序结构。这不仅有助于提高指令的局部性，减少跳转指令的开销，还能为后续的优化操作提供更简洁的代码结构，从而进一步提升代码的执行效率和可维护性。

```
bool SimplifyCFG::simplifyBranch(BasicBlock* bb){
```

```
    if(bb->Size()==0){
        return false;
    }
    //获取基本块最后一条指令
    Instruction* lastInst=bb->GetBack();

    //判断是否条件跳转指令
    bool is_cond_branch = lastInst && lastInst->id==Instruction::Op::Cond;
    if(!is_cond_branch){
        return false;
    }
    //获取条件操作数和两个基本块
    Value* cond=lastInst->GetOperand(0);
    BasicBlock*
trueBlock=dynamic_cast<BasicBlock*>(lastInst->GetOperand(1));
    BasicBlock*
falseBlock=dynamic_cast<BasicBlock*>(lastInst->GetOperand(2));
    //确认`目标基本块合法
    if(!trueBlock||!falseBlock){
        return false;
    }
    //判断条件是否是常量函数
    auto* c=dynamic_cast<ConstIRBoolean*>(cond);
    if(!c){
        std::cerr << "Not a constant condition\n";
        return false;
    }
    BasicBlock* targetBlock=c->GetVal() ? trueBlock:falseBlock;
    //创建无条件跳转指令,替换原条件跳转指令
    auto oldInst=bb->GetLastInsts();
    bb->erase(oldInst);
    Instruction* uncondBr=new UnCondInst(targetBlock);
```

```
    bb->push_back(uncondBr);


    //更新 CFG
    bb->RemoveNextBlock(trueBlock);
    bb->RemovePredBlock(falseBlock);
 targetBlock->RemovePredBlock(bb);
    bb->AddNextBlock(targetBlock);
    targetBlock->AddPredBlock(bb);


    std::cerr << "Simplified to: br label %" << targetBlock->GetName() <<
"\n";
    return true;
}
```

代码 3.8.3-6

SimplifyBranch 优化操作致力于简化控制流中的恒定条件跳转，即将类似 if (true)或 if (false)的恒定布尔条件跳转转换为无条件跳转。这一过程有助于简化代码结构，减少不必要的条件判断，从而提高代码的执行效率并降低指令路径长度。

具体实施时，首先会检查当前基本块是否以条件跳转指令（CondInst）结尾。如果条件跳转的条件是一个布尔常量（ConstIRBoolean），即条件恒为真或恒为假，那么就可以进行优化。优化的具体步骤包括：删除原有的条件跳转指令，然后插入一条无条件跳转指令，跳转到对应的分支。同时，为了保持控制流图（CFG）的正确性，还需要更新 CFG 中的前驱和后继关系，确保控制流的正确性和完整性。

通过这种优化，可以有效清理代码中的冗余条件判断，使控制流更加直接和高效。这不仅有助于提高代码的执行效率，还能为后续的优化操作提供更简洁的代码结构，进一步提升代码质量和可维护性。

```
//消除无意义 phi
bool SimplifyCFG::eliminateTrivialPhi(BasicBlock* bb){
```

```
bool changed=false;

//遍历当前基本块中所有指令
for(auto it=bb->begin();it!=bb->end();){
    Instruction* inst=*it;

    //isphi?
    if(inst->id==Instruction::Op::Phi){
        Value* same=nullptr;
        bool all_same=true;

        //遍历所有 phi 的输入值
        for(size_t i=0;i<inst->GetOperandNums();i+=2){
            Value* val=inst->GetOperand(i);
            if(!same){
             same=val;
            }else if(val!=same){
                all_same=false;
                break;
            }
        }
        //所有输入值相同,可以替换
        if(all_same&&same){
            inst->ReplaceAllUseWith(same);

            auto to_erase=it;
            ++it;
            bb->erase(*to_erase);

            changed=true;
            continue;
```

```
            }
        }
        ++it;
    }
    return changed;
}
```

代码 3.8.3-7

EliminateTrivialPhi 优化操作旨在消除静态单赋值（SSA）形式中冗余的 phi 节点。在 SSA 中，phi 节点用于处理控制流合并点的值选择问题。然而，当 phi 节点的所有输入值都相同时，无论这些值来自哪个前驱块，该 phi 节点就变得冗余，可以被优化掉。

具体实施时，会遍历当前基本块中的所有 phi 指令。对于每个 phi 指令，检查其所有输入值是否为同一个变量或常量。如果是这样，就用该变量或常量替换 phi 指令的所有使用点，然后删除该 phi 指令。同时，记录下这种修改情况，以便驱动后续的迭代优化。

这种优化在控制流图（CFG）被规整后特别有效，例如在合并基本块之后，往往会产生大量冗余的 phi 节点。通过消除这些无意义的 phi 节点，可以进一步简化 CFG，减少不必要的指令，提高代码的可读性和执行效率。这不仅有助于提升代码质量，还能为后续的优化操作提供更简洁的代码结构，是常见的 CFG 简化策略之一。

3.9 循环优化

循环优化（Loop Optimization）是编译器优化中的核心部分，旨在提高循环结构的执行效率，减少不必要的计算和内存访问。由于程序的大部分执行时间通常消耗在循环上，因此高效的循环优化能显著提升程序性能。

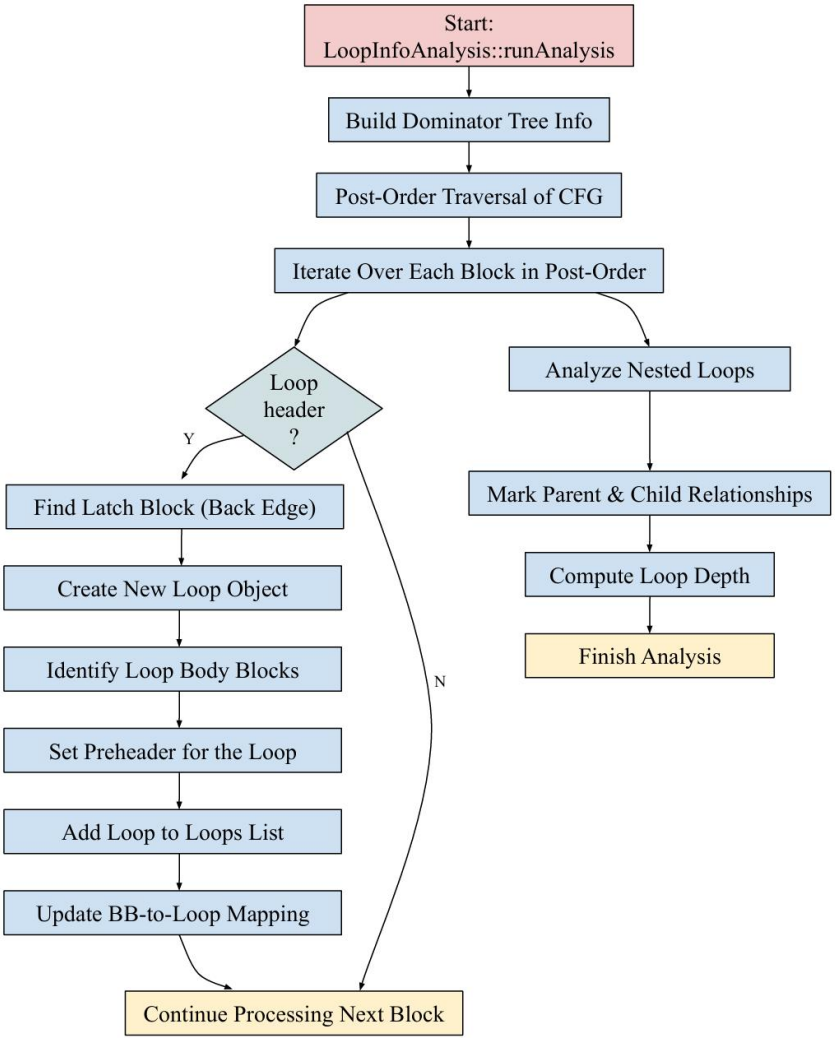


图 3.9-1

3.9.1 构建循环分析，进行循环边界判断

循环分析（LoopInfoAnalysis）主要采用了基于支配树（Dominator Tree）的后序遍历算法来识别循环结构，并结合了控制流图（CFG）的特性进行循环边界的判断。

支配树（Dominator Tree）与后序遍历：

后序遍历的作用：后序遍历（Post-order）确保在处理当前节点前，其所有子节点已被访问。这在循环分析中的意义包括：

自底向上分析：优先处理子循环（内层循环），再处理父循环（外层循环），

避免重复分析。

依赖关系解决：例如，循环头的支配性质需要先确认其子节点的支配关系

循环识别条件：

回边检测是识别程序控制流图中循环结构的关键技术，其严格定义需要满足两个相互关联的条件：首先，在控制流边 A→B 中，基本块 B 必须支配基本块 A，这意味着从程序入口到 A 的所有执行路径都必须经过 B，这种支配关系保证了 B 作为循环头的必然执行；其次，必须存在一条从 B 回到 A 的可达路径，这样才能形成完整的循环回路。

在实际的编译器实现中，检测回边通常结合支配树分析和深度优先搜索来完成，当发现某条边同时满足支配性和可达性时，即可确认其为回边。基于检测到的回边，循环体的收集采用反向遍历算法，从回边的起始节点 A（称为 Latch 节点）开始，沿着控制流的反向边进行遍历，期间记录所有经过的基本块，直到遇到循环头 B 为止。这种反向遍历需要特别注意控制流的分支情况，当遇到多个前驱节点时，必须验证这些节点同样被循环头 B 所支配，以确保它们属于当前循环体的一部分。

整个收集过程本质上是在构建循环的强连通分量，最终得到的节点集合构成了完整的循环体，包括循环头、循环尾以及所有中间执行路径上的基本块。这种算法不仅能处理简单循环，还可以正确识别嵌套循环结构，为后续的循环优化提供准确的程序分析基础。

嵌套循环处理：

父循环关联：通过 setLoopsHeader 和 addLoopsBody 方法将嵌套循环关联到外层循环中。

度计算：LoopsDepth 记录嵌套深度，用于优化优先级判断。

核心算法：

该函数用于识别控制流图中的循环结构，包括简单循环和嵌套循环。该算法基于支配树（Dominator Tree）和后序遍历，通过检测回边（Back Edge）来识别循环头（Header）和循环体（Body），并处理嵌套循环关系。

```
void LoopInfoAnalysis::runAnalysis()
{
  // 通过后续遍历 CFG，来获取每个基本块的后序遍历
  // 然后通过支配树来查找每个基本块的前驱，通过 for 循环查找，来判断前驱中哪些是支配这个基本块
  // 如果找到前驱，那就将 curbb 存储在 latch 中，也就是目前正在遍历的基本块
  // 通过这种方法，构建不同的 latch 前驱序列
  //
  for (auto curbb : PostOrder)
  {
    std::vector<BasicBlock *> latch;            // 回边列表
    for (auto succbb : _dom->getPredBBs(curbb)) // 查找每个基本块的前驱
    {
      if (_dom->dominates(curbb, succbb)) // 如果 curbb 支配 succbb
      {
        latch.push_back(succbb);
      }
    }
    if (!latch.empty())
    {
      Loop *loop = new Loop(curbb);
      std::vector<BasicBlock *> WorkList = {latch.begin(), latch.end()};

      while (!WorkList.empty())
      {
        auto bb = WorkList.back();
        WorkList.pop_back();
        auto node = _dom->getNode(bb);
  //查找这个基本块，是否还有别的循环
        if (auto iter = Loops.find(bb); iter != Loops.end())
        {
          // 找到嵌套循环的最外层
          auto tmp = iter->second;
          while (tmp->GetLoopsHeader() != nullptr)
          {
            tmp = tmp->GetLoopsHeader();
          }
          if (tmp == loop)
            continue;
          tmp->setLoopsHeader(loop);
```

```
        loop->addLoopsBody(tmp);

        BasicBlock *header = tmp->getHeader();
        for (auto n : _dom->getPredBBs(header))
        {
          if (auto iter_ = Loops.find(header); iter != Loops.end())
            WorkList.push_back(header);
        }
      }  else
      {
        Loops.emplace(bb, loop);
        if (bb == curbb)

          continue;
        loop->addLoopBody(bb);
        WorkList.push_back(bb);
      }
    }
    loops.push_back(std::move(loop));
  }
 }
}
```

代码 3.9.1-1

后序遍历支配树：遍历每个基本块 curbb（按后序顺序）。检查 curbb 的所有前驱（_dom->getPredBBs(curbb)）。如果 curbb 支配某个前驱 succbb（即 _dom->dominates(curbb, succbb)），则 succbb 是一个回边（Back Edge），表明 curbb 可能是一个循环头（Header）。

回边检测与循环识别：使用 WorkList 进行广度优先搜索（BFS），从 latch（回边前驱）开始，反向遍历所有能到达 curbb 的基本块。这些基本块构成循环体。

嵌套循环处理:如果某个基本块 bb 已经属于另一个循环(Loops.find(bb) != Loops.end())，则当前循环可能是其外层循环。通过 GetLoopsHeader() 找到最外层循环，并建立父子关系。

通过递归或迭代方式计算循环的嵌套深度（LoopsDepth），用于后续优化（如

循环展开优先级）。

3.9.2 循环简化

循环简化是编译器优化中的一个重要阶段，它通过规范化循环结构为后续优化（如循环展开）创造有利条件。

```
bool Run() {
  bool changed = false;
  m_dom = AM.get<dominance>(m_func); // 获取支配树信息
  loopAnlay = AM.get<LoopAnalysis>(m_func, m_dom, std::ref(DeleteLoop));
  // 先处理内层循环（后序遍历）
  for (auto iter = loopAnlay->begin(); iter != loopAnlay->end(); iter++)
{
    auto loop = *iter;
    changed |= SimplifyLoopsImpl(loop); // 实际简化实现
    AM.AddAttr(loop->GetHeader(), Simplified);
  }
  SimplifyPhi(); // PHI 节点简化
  return changed;
}
```

代码 3.9.2-1

这个 Run 函数是循环简化优化的主入口，主要干三件事：首先它通过 PassManager 获取当前函数的支配树和循环分析结果，这是后续优化的基础信息。

然后按照后序遍历的顺序处理所有循环，从最内层循环开始往外处理，这样能保证嵌套循环的正确简化。每次循环处理都会调用 SimplifyLoopsImpl 这个核心实现，并标记已经简化过的循环头。

最后还会额外处理一下 PHI 节点，清理循环简化后可能产生的冗余 PHI 指令。整个函数返回一个布尔值，告诉我们这次优化是否真的改变了代码。这种从内到

外的处理顺序很关键，因为内层循环简化后可能会影响外层循环的结构，反过来就不行了。

函数最后那个 SimplifyPhi 的调用也很有必要，因为循环简化经常会留下一些没用的 PHI 节点，需要专门清理。

循环简化实现 (SimplifyLoopsImpl)：

```
bool SimplifyLoopsImpl(LoopInfo *loop) {
  // 1. 处理 preheader（确保每个循环有唯一的前导块）
  if (!preheader) {
    InsertPreHeader(L); // 插入 preheader 块
    changed |= true;
  }
  // 2. 处理 exit blocks（规范化循环出口）
  for (auto exit : loopAnlay->GetExit(L)) {
    FormatExit(L, exit); // 格式化出口块
    changed |= true;
  }
  // 3. 处理 latch/back-edge（确保唯一回边）
  if (Latch.size() > 1) {
    FormatLatch(loop, preheader, Latch); // 合并多个 latch
    changed |= true;
  }
}
```

代码 3..2-2

这个 SimplifyLoopsImpl 函数是循环简化的核心实现，主要处理三个关键问题。首先看 preheader 的处理，如果发现循环还没有 preheader 块，就调用 InsertPreHeader 插入一个。这个 preheader 很重要，它是循环的唯一入口块，所有进入循环的跳转都要先经过它，这样后续优化才有统一的操作点。然后是 exit

blocks 的处理，循环可能有多个出口，这里通过 FormatExit 函数把它们规范化，保证每个出口块的结构一致。

最后处理 latch 块的问题，有些循环会有多个回边，这时候 FormatLatch 函数就派上用场了，它把多个 latch 合并成一个，让循环结构更规整。每次修改都会把 changed 标志置为 true，这样外层就知道确实做了优化。

这三个步骤做完，原来乱七八糟的循环结构就变得整齐规范了，后面的优化 pass 处理起来就方便多了。这种规范化处理虽然看起来简单，但对后续的循环展开、向量化这些优化来说特别重要，没有这个基础，很多高级优化都做不了。

关键辅助函数：

插入 Preheader（InsertPreHeader）：

```
void InsertPreHeader(LoopInfo *loop) {
  // 收集所有外部前驱块
  std::vector<BasicBlock *> OutSide;
  // 创建新的 preheader 块
  BasicBlock *preheader = new BasicBlock();
  m_func->InsertBlock(Header, preheader);
  // 重定向外部前驱到 preheader
  for (auto target : OutSide) {
    if (auto cond = dynamic_cast<CondInst *>(condition)) {
      cond->RSUW(i, preheader); // 替换操作数
    }
  }
  // 更新 PHI 节点
  for (auto inst : *Header) {
    if (auto phi = dynamic_cast<PhiInst *>(inst))
      UpdatePhiNode(phi, work, preheader);
  }
```

```
  // 更新支配关系
  UpdateInfo(OutSide, preheader, Header, loop);
}
```

代码 3.9.2-3

这个 InsertPreHeader 函数负责给循环插入一个统一的前导块，是循环规范化的重要步骤。函数首先会收集所有从循环外部跳转到循环头的那些基本块，这些块都需要被重定向。

然后创建一个全新的 preheader 块，插在循环头前面。接下来就是重定向工作，把原来那些直接跳转到循环头的外部前驱块，都改成跳转到新创建的 preheader 块。这里要特别注意条件跳转的情况，得把对应的跳转目标改过来。PHI 节点的处理也很关键，因为循环头里的 PHI 节点原来是从各个外部前驱块接收值的，现在这些前驱都统一到 preheader 了，所以 PHI 节点的输入也需要相应调整。

最后还要更新支配关系信息，毕竟新加了一个块，支配树的结构也跟着变了。这个 preheader 的引入看似简单，但作用很大，它把循环的入口统一起来，让后续的循环优化可以基于一个清晰的结构来做，不用再处理各种乱七八糟的入口情况。特别是对循环外提、循环展开这些优化来说，有了 preheader 会让代码生成简单很多。

PHI 节点更新 (UpdatePhiNode)：

```
void UpdatePhiNode(PhiInst *phi, std::set<BasicBlock *> &worklist,
BasicBlock *target) {
  // 检查所有输入值是否相同
  if (所有输入值相同) {
    // 直接替换为统一值
    phi->updateIncoming(ComingVal, target);
  } else {
    // 创建新的 PHI 节点合并不同输入
    PhiInst *pre_phi = PhiInst::NewPhiNode(...);
```

```
    phi->updateIncoming(pre_phi, target);
  }
}
```

代码 3.9.2-4

UpdatePhiNode 这个函数专门处理循环规范化过程中 PHI 节点的调整问题。当我们在循环前面插入 preheader 块后，原来循环头 PHI 节点的输入来源就发生了变化。

函数首先会检查所有外部前驱块传入 PHI 节点的值是否都一样，如果值都相同那事情就简单了，直接把 PHI 节点替换成这个统一值就行。但实际情况往往没那么理想，不同前驱块传进来的值可能各不相同，这时候就需要在 preheader 里新建一个 PHI 节点来合并这些不同的输入值。这个新 PHI 节点会把原来分散在各个前驱块的值收集起来，然后再统一传给循环头里的 PHI 节点。这种处理方式既保证了程序语义不变，又让循环结构更加规范。

特别要注意的是，新建 PHI 节点时要确保类型匹配，还要正确设置各个前驱块对应的输入值，一个不小心就会搞出编译器错误。这个函数虽然不长，但在循环优化里扮演着关键角色，它让后续的优化 pass 可以不用操心 PHI 节点的特殊情况，直接按照标准化的循环结构来处理就行。

循环简化优化前的 IR 表示：

```
; 多前驱循环头
bb1:
  br label %header
bb2:
  br label %header
header:
  %phi = phi i32 [0, %bb1], [1, %bb2]
```

代码 3.9.2-5

优化后的版本通过插入 preheader 块，把所有外部入口统一归拢。现在 bb1

和 bb2 都改为跳转到 preheader，而 preheader 内部的%phi.pre 节点负责合并所有外部输入值，再通过单一分支进入循环头。新的循环头 header 现在只需处理来自 preheader 的单一输入，结构变得干净利落。自 preheader 的单一输入，结构变得干净利落。

3.9.3 循环旋转

循环旋转优化的核心思想其实很直观，就是把常见的 while 循环改造成 do-while 的形式。具体来说，就是把原本放在循环开头的条件判断挪到循环末尾去。这么一改有个明显的好处，就是循环体至少能执行一次，省去了首次循环前的条件检查。

在实现上，编译器会先把循环头里的不变指令提到外面去，减少循环内部的计算量。不过这个改造过程要特别注意控制流和数据流的处理，特别是那些 PHI 节点，一个处理不当就会破坏程序逻辑。旋转后的循环结构更紧凑，执行效率也更高，特别是对那些循环次数较多的情况，能省下不少条件判断的开销。这个优化虽然原理简单，但实际实现起来要考虑很多细节问题，比如循环出口的处理、嵌套循环的相互影响等等。

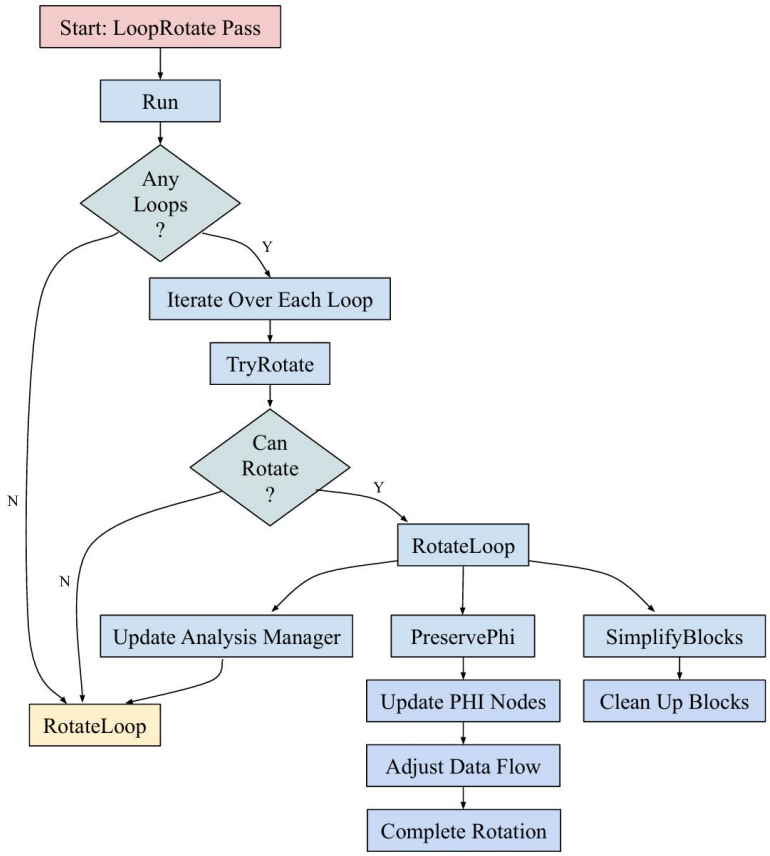


图 3.9.3-1

主入口函数：

```
bool Run() {
  bool changed = false;
  auto sideeffect = AM.get<SideEffect>(&Singleton<Module>());
  m_dom = AM.get<dominance>(m_func); // 获取支配树信息
  loopAnlasis = AM.get<LoopAnalysis>(m_func, m_dom,
std::ref(DeleteLoop));

  // 遍历所有循环
  for (auto loop : loopAnlasis->GetLoops()) {
    bool Success = TryRotate(loop); // 尝试旋转
    if (RotateLoop(loop, Success) || Success) { // 执行旋转
      changed |= true;
      AM.AddAttr(loop->GetHeader(), Rotate);
    }
  }
  return changed;
```

```
}
```

代码 3.9.3-1

获取当前函数的所有循环结构（通过 LoopAnalysis），对每个循环尝试进行旋转优化（TryRotate 判断是否可以旋转），如果成功，则调用 RotateLoop 进行实际变换，标记该循环已旋转，供后续优化使用。

RotateLoop() 方法：循环旋转核心实现：

```
bool RotateLoop(LoopInfo *loop, bool Succ) {
  auto prehead = loopAnlasis->GetPreHeader(loop);
  auto header = loop->GetHeader();
  auto latch = loopAnlasis->GetLatch(loop);

  auto cond = dynamic_cast<CondInst *>(header->back());
  auto New_header = dynamic_cast<BasicBlock *>(cond->GetOperand(1));
  auto Exit = dynamic_cast<BasicBlock *>(cond->GetOperand(2));

  // 外提不变指令
  for (auto inst : *header) {
    if (LoopAnalysis::IsLoopInvariant(contain, inst, loop) &&
CanBeMove(inst)) {
      inst->EraseFromParent();
      It.insert_before(inst);
    }
  }

  // 更新控制流
  prehead->back()->RSUW(1, New_header);
  prehead->back()->RSUW(2, Exit);
```

```
  // 维护 PHI 节点
  PreservePhi(...);

  // 更新循环信息
  loop->setHeader(New_header);
  AM.ChangeLoopHeader(header, New_header);
  SimplifyBlocks(header, loop);
}
```

代码 3.9.3-2

获取关键基本块：

preheader：循环前的基本块

header：当前循环头

latch：循环尾，通常包含增量更新操作

提取条件跳转指令：

假设循环以条件跳转结束条件指令 cond 是最后一个指令，从中提取 New_header 和 Exit 块。

外提不变指令：

对于 header 中的每条指令，若满足：是循环不变量（Loop Invariant）或者可以外提（由 CanBeMove() 判断），则将其移动到 preheader。

控制流重构：

修改 preheader 的条件跳转目标：把原来指向 header 的分支改为指向 body 或 exit，这样就实现了将判断条件提前到循环外部执行

维护 PHI 节点：

确保 PHI 指令的 incoming 值仍然对应正确的来源块，在新旧 header 之间建立映射关系。

更新循环元信息：

设置新的循环头，删除旧 header 中多余的指令或简化其结构。

CanBeMove() 方法：判断指令是否可外提：

```
bool CanBeMove(User *I) {
  if (auto call = dynamic_cast<CallInst *>(I)) {
    return !call->HasSideEffect(); // 无副作用的调用可以外提
  }
  return isa<BinaryInst>(I) || isa<GetElementPtrInst>(I); // 算术和地址计
算可外提
}
```

代码 3.9.3-3

这个 CanBeMove 函数是判断指令能否从循环里提到外面去的关键函数。它主要看两种指令：一种是函数调用，只要这个调用没有副作用，比如不修改全局变量、不做 IO 操作，就可以安全地外提；另一种是普通的算术运算和地址计算指令，像加减乘除或者数组下标计算这种，这些指令本身不会改变程序状态，提出来也不会影响程序逻辑。

实际处理的时候要特别小心，有些看着像纯计算的指令可能暗藏玄机，比如除零检查或者指针越界这种潜在问题。所以这个函数虽然代码简单，但在循环优化里起着把关的作用，决定了哪些指令能往外提，哪些必须老老实实留在循环里。用的时候还得结合具体上下文，有时候即使指令本身可以外提，但如果依赖循环内部的值，那还是不能随便动。

PreservePhi() 方法：维护 PHI 节点：

```
void PreservePhi(...) {
  // 1. 更新后继块的 PHI 节点
  for (auto succ : header->successors()) {
    for (auto phi : succ->phis()) {
      if (PreHeaderValue.count(phi->incomingValueForBlock(header)))
        phi->setIncomingValue(PreHeaderValue[...], preheader);
```

```
    }
  }
  // 2. 创建新的 PHI 节点处理数据流
  auto new_phi = PhiInst::NewPhiNode(new_header->front(), new_header,
type);
  new_phi->addIncoming(cloned, preheader);
  new_phi->addIncoming(inst, Latch);
  // 3. 更新使用点
  for (auto use : inst->uses()) {
    if (shouldRewrite(use))
      use->set(new_phi);
  }
}
```

代码 3.9.3-4

这个 PreservePhi 函数专门处理循环旋转时 PHI 节点的维护工作，是保证程序正确性的关键。函数主要做三件事：首先更新循环后继基本块里的 PHI 节点输入值，把原来来自循环头的输入改成来自新创建的 preheader 块的值。

要在新的循环头里建立新的 PHI 节点，这个节点会接收两个来源的值 - 一个是从 preheader 块传来的初始值，另一个是从 latch 块传来的循环更新值。最后还要把所有用到原值的地方替换成新的 PHI 节点，确保数据流正确传递。这里最麻烦的是要准确判断哪些使用点需要更新，特别是当循环里有多个基本块的时候，很容易漏掉某些边界的特殊情况。

函数里那些 PreHeaderValue 的维护和 shouldRewrite 的判断条件，都是为了避免在旋转过程中破坏原有的数据依赖关系。这个处理虽然看起来只是简单的 PHI 节点替换，但实际上要保证旋转后的循环和原来语义完全一致，需要仔细处理各种边界情况。

循环旋转优化前的 IR 表示：

```
; 标准 while 循环
preheader:
  br label %header
header:
  %phi = phi i32 [0, %preheader], [%inc, %latch]
  %cmp = icmp slt i32 %phi, 100
  br i1 %cmp, label %body, label %exit
body:
  br label %latch
latch:
  %inc = add i32 %phi, 1
  br label %header
```

代码 3.9.3-5

代码是典型的 while 循环结构，每次循环开始都要先检查条件（%cmp），然后才决定是否进入循环体。这种结构有个明显的效率问题——就算循环肯定要执行，第一次也得先做条件判断。

循环旋转优化后的 IR 表示：

```
; do-while 形式
preheader:
  %0 = icmp slt i32 0, 100  ; 条件外提
  br i1 %0, label %body, label %exit


body:
  %phi = phi i32 [0, %preheader], [%inc, %latch]
  ...
  %inc = add i32 %phi, 1
  %cmp = icmp slt i32 %inc, 100
```

```
br i1 %cmp, label %body, label %exit
```

代码 3.9.3-6

旋转优化后的 IR 变成了 do-while 形式，直接把第一次的条件判断提到循环外面（%0 = icmp slt i32 0, 100）。这样做的好处很明显：首先，循环体内减少了一次条件判断指令，特别是对于循环次数较多的情况，这个节省就很可观；其次，新的结构让循环体可以连续执行，现代处理器的分支预测在这种结构下表现更好。不过代价是要多复制一次循环条件（%cmp = icmp slt i32 %inc, 100），放在循环末尾用于控制是否继续循环。

3.9.4 循环展开

循环展开（Loop Unrolling）是一种经典的编译器优化技术，通过减少循环控制开销（如分支预测、迭代计数）来提高程序执行效率。本报告基于 LoopUnrolling 类的实现，分析其核心算法、优化策略及实际应用。

整体流程概述：

循环展开优化的实现主要分为三个关键阶段，整个过程在 LoopUnrolling 类中完成。首先进行循环识别阶段，这个阶段会利用支配树和循环分析的结果，找出程序中那些结构规整、适合展开的循环结构。

确定目标循环后，紧接着进入展开可行性检查阶段，这里会对循环体的指令开销进行仔细评估，同时分析循环的迭代模式，确保展开后不会导致代码过度膨胀或者引入性能问题。

最后是实际的循环展开执行阶段，这个阶段通过函数内联的技术手段，把循环体复制多份来实现展开效果。整个流程在 run()函数中串联起来：先获取支配树和循环分析信息，然后遍历所有循环结构，对每个循环依次进行识别、检查、展开的操作。

特别值得注意的是，处理顺序是从内层循环往外层循环进行的，这样可以避免嵌套循环之间的相互干扰。展开完成后还会进行一些清理工作，比如简化多余

的 PHI 节点，确保生成的代码保持整洁高效。这种分阶段处理的方式既保证了优化的可靠性，又能针对不同循环特征灵活调整展开策略。

```
bool LoopUnrolling::run() {
    // 1. 构建支配树和循环分析
    DominantTree dom(_func);
    dom.InitNodes();
    dom.BuildDominantTree();
    LoopInfoAnalysis loopAnalysis(_func, &dom, DeleteLoop);
 // 2. 遍历所有可展开循环
    for (auto currLoop : loopAnalysis->loops) {
        if (!CanBeUnroll(currLoop)) continue;  // 可行性检查

        // 3. 获取循环体并执行展开
        auto unrollbody = GetLoopBody(currLoop);
        if (unrollbody) {
            Unroll(currLoop, unrollbody);  // 核心展开逻辑
            return true;
        }
    }
    return false;
}
```

代码 3.9.4-1

关键函数解析：

可展开性检查 (CanBeUnroll)：

功能：验证循环是否符合展开条件

核心逻辑：

循环结构验证：

```
    if (header != latch) return false;  // 仅支持单入口/单出口循环
```

代码 3.9.4-2

迭代次数静态可计算：

```
// 必须为常量边界（ConstIRInt）
if (!dynamic_cast<ConstIRInt*>(loop->trait.initial) ||
           !dynamic_cast<ConstIRInt*>(loop->trait.boundary))
   return false;
```

代码 3.9.4-3

指令开销评估：

```
int cost = CaculatePrice(body, _func, Lit_count);
if (cost > MaxInstCost) return false;  // 超过阈值则放弃展开
```

代码 3.9.4-4

迭代次数计算示例（加法循环）：

```
case BinaryInst::Op_Add:
   // 公式: (bound - initial + step - 1) / step
   Lit_count = (bound - initial + step + (step > 0 ? -1 : 1)) / step;
   break;
```

代码 3.9.4-5

循环体提取 (GetLoopBody)：

功能：将循环体封装为可重用的函数单元

关键技术：

PHI 节点处理：

```
// 提取循环头部的 PHI 节点（归纳变量）
auto phi = dynamic_cast<PhiInst*>(header->GetLastInsts());
if (!phi || phi->getNumIncomingValues() != 2) return nullptr;
```

代码 3.9.4-6

外部变量参数化：

```
// 识别循环依赖的外部变量
if (Judge(operand)) {
    Val2Arg[operand] = new Var(Var::Param, operand->GetType(), "");
}
```

代码 3.9.4-7

函数克隆与调用：

```
auto unrollFunc = new Function(ty, "loop_unroll");
auto callinst = new CallInst(unrollFunc, args);  // 生成调用指令
```

代码 3.9.4-8

循环展开执行 (Unroll)：

功能：通过函数内联实现循环体复制

核心步骤：

迭代次数计算：

```
// 根据操作类型计算迭代次数
  switch (operation)
  {
  case BinaryInst::Op_Add:
    loop_iterations = (boundary_value - initial_value + step_value +
(step_value > 0 ? -1 : 1)) / step_value;
    break;
  case BinaryInst::Op_Sub:
    loop_iterations = (initial_value - boundary_value + step_value +
(step_value > 0 ? -1 : 1)) / step_value;
    break;
case BinaryInst::Op_Mul:
    loop_iterations = static_cast<int>(std::log(boundary_value /
initial_value) / std::log(step_value));
```

```
    break;
  case BinaryInst::Op_Div:
    loop_iterations = static_cast<int>(std::log(initial_value /
boundary_value) / std::log(step_value));
    break;
  default:
    assert(false && "Unsupported operation type");
  }
```

代码 3.9.4-9

参数映射与内联：

```
// 建立参数到实际值的映射
    mapArguments(next_call, current_step_value, current_result);
// 执行内联展开
    auto inline_result = _func->InlineCall(call_inst, Arg2Orig);
```

代码 3.9.4-10

清理冗余代码：

```
// 删除原始归纳变量
loop->trait.indvar->ReplaceAllUseWith(UndefValue::Get(...));
delete loop->trait.indvar;
```

代码 3.9.4-11

关键技术亮点：

代价模型 (CaculatePrice)：

采用加权指令计数控制代码膨胀：

```
int cost = 0;
for (auto inst : *bb) {
    if (isa<LoadInst>(inst)) cost += 4;  // 内存访问权重较高
    else if (isa<CallInst>(inst)) cost += 10;
```

```
    else cost += 1;  // 普通算术指令
}
return cost * Lit_count;  // 总开销=单次迭代开销*迭代次数
```

代码 3.9.4-12

控制流重构：

基本块克隆：

```
auto newHeader = header->clone();  // 创建展开后的新头部
```

代码 3.9.4-13

分支指令调整：

```
                    // 将条件跳转指向展开后的块
                cond_inst->SetOperand(1, newHeader);
```

代码 3.9.4-14

嵌套循环支持：

通过 LoopInfoAnalysis 维护父子循环关系：

```
if (auto parentLoop = loop->GetParentLoop()) {
    parentLoop->addSubLoop(unrolledLoop);
}
```

代码 3.9.4-15

原始代码示例：

```
int i = 0;
int sum = 0;
while (i < 4) {  // 循环头 (Header)
    sum += arr[i];  // 循环体 (Body)
    i++;          // 循环尾 (Latch)
}
```

代码 3.9.4-16

对应的 IR 表示：

```
; 初始化
entry:
  store i32 0, ptr %i
  store i32 0, ptr %sum
  br label %header
; 循环头（条件判断）
header:
  %i.val = load i32, ptr %i
  %cmp = icmp slt i32 %i.val, 4
  br i1 %cmp, label %body, label %exit
; 循环体
body:
  %sum.val = load i32, ptr %sum
  %ptr = getelementptr i32, ptr %arr, i32 %i.val
  %elem = load i32, ptr %ptr
  %new_sum = add i32 %sum.val, %elem
store i32 %new_sum, ptr %sum
  br label %latch
; 循环尾（迭代更新）
latch:
  %i.next = add i32 %i.val, 1
  store i32 %i.next, ptr %i
  br label %header
; 退出
exit:
  ...
```

代码 3.9.4-17

原始代码是标准的循环结构，包含初始化、条件判断、循环体和迭代更新四个部分。每次循环都要重复执行条件检查（%cmp）和指针计算（%ptr），还要处理循环变量的更新（%i.next）和存储。这种结构虽然通用，但执行效率不高，特别是循环次数较少时，循环控制的开销占比很大。

完全展开（迭代 4 次）：

```
entry:
  ; 迭代 1
  %ptr0 = getelementptr i32, ptr %arr, i32 0
  %val0 = load i32, ptr %ptr0
  %sum1 = add i32 0, %val0

  ; 迭代 2
  %ptr1 = getelementptr i32, ptr %arr, i32 1
  %val1 = load i32, ptr %ptr1
  %sum2 = add i32 %sum1, %val1

  ; 迭代 3
  %ptr2 = getelementptr i32, ptr %arr, i32 2
  %val2 = load i32, ptr %ptr2
  %sum3 = add i32 %sum2, %val2

  ; 迭代 4
  %ptr3 = getelementptr i32, ptr %arr, i32 3
  %val3 = load i32, ptr %ptr3
  %sum4 = add i32 %sum3, %val3

  ; 直接跳转到退出
```

```
br label %exit
```

代码 3.9.4-18

展开后的版本直接把循环体复制了四次，完全消除了循环控制的开销。现在可以看到,每个迭代的计算都是顺序执行的:先计算数组元素地址(%ptr0 到%ptr3)，然后加载元素值（%val0 到%val3)，最后累加到总和（%sum1 到%sum4)。这种线性结构对 CPU 流水线非常友好，省去了所有分支预测和循环变量的维护操作。

3.9.5 循环删除

条件检测（CanBeDelete)：

```
bool CanBeDelete(LoopInfo* loopInfo) {
    // 检查退出块 PHI 节点的值一致性
    for (PhiInst* Phi : ExitBlock->phis()) {
        if (!所有出口块提供相同值) return false;
        if (!可循环不变外提) return false;
    }

    // 检查副作用
    for (BasicBlock* BB : 循环体) {
        if (指令有副作用) return false;
    }
    return true;
}
```

代码 3.9.5-1

验证所有循环出口块到退出块的 PHI 节点输入值是否相同，确保循环内无副作用指令（存储/调用/终止符等)，通过 makeLoopInvariant 判断值能否外提。

循环不变式外提（makeLoopInvariant）

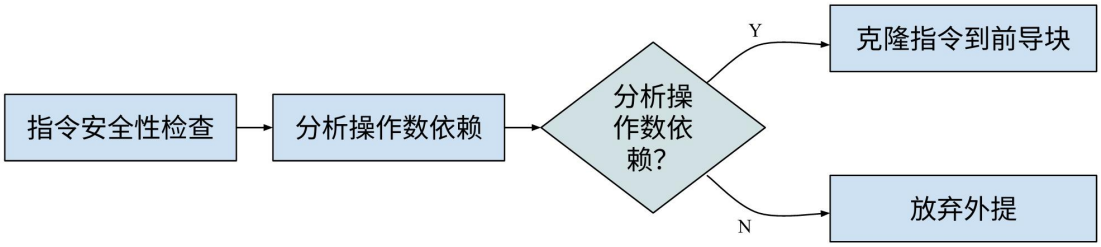


图 3.9.5-1

递归处理操作数依赖链，特殊处理 PHI 节点（合并相同值），通过 IsSafeToMove 过滤危险指令。

循环删除：

```
void TryDeleteLoop(LoopInfo* loopInfo) {

    // 重定向前导块跳转

    PreHeader->Terminator->replaceJumpTo(Header, ExitBlock);



    // 清理 PHI 节点引用

    for (PhiInst* Phi : ExitBlock->phis()) {

        Phi->replaceIncoming(ExitingBlocks[0], PreHeader);

    }

    // 删除循环体基本块

    for (BasicBlock* BB : LoopBody) {

        BB->deleteAllInstructions();

        DominatorTree->removeNode(BB);

    }

}
```

代码 3.9.5-2

这个函数实现了删除无用循环的核心逻辑，主要处理三个关键操作。首先把循环前导块的跳转目标从原来的循环头直接改到退出块，相当于把整个循环短路

掉。然后要处理退出块里的 PHI 节点，这些节点原本接收来自循环内部块的输入值，现在需要改成接收前导块的值。

最后是清理工作，把循环体内所有基本块的指令逐个删除，同时更新支配树结构，把这些块从控制流图中彻底移除。这里要特别注意删除顺序，得确保先处理好所有数据依赖再删指令，不然可能会破坏 SSA 形式的正确性。整个过程看似简单，但实际上要维护好控制流和数据流的一致性，特别是在有多个退出边的情况下，PHI 节点的更新很容易出错。这种循环删除优化在遇到空循环或者循环条件恒假的情况下特别有用，能直接把无效代码清除干净。

3.9.6 循环并行

并行条件检测（CanBeParallel）：

```
bool CanBeParallel(LoopInfo* loop) {
    // 必要条件检查
    if (!已旋转 || 非标准比较操作) return false;
 // 迭代次数分析
    if (迭代次数<=100) return false; // 小循环不并行
    // 数据依赖检查
    if (!DependencyAnalysis(loop)) return false;
    // 副作用检查
    for (auto BB : 循环体) {
        if (存在存储/调用等副作用指令) return false;
    }
    return true;
}
```

代码 3.9.6-1

仅处理带有 Rotate 标记的规范化循环，迭代次数需大于 100 次（避免并行开销），循环内无不可并行化的副作用指令，通过数据依赖分析验证可并行性。

数据依赖分析（DependencyAnalysis）：

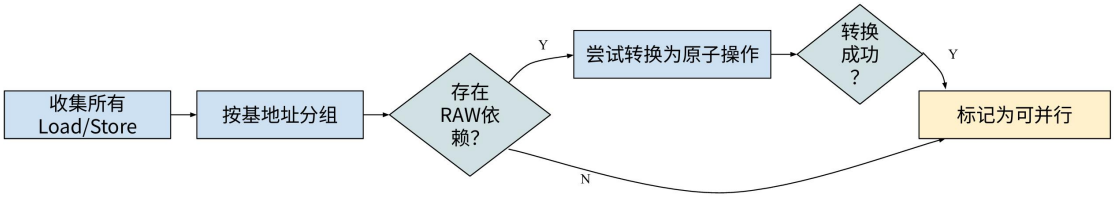


图 3.9.6-1

典型模式转换：

```
// 原始代码

x = arr[i];

x = x + delta;

arr[i] = x;



// 转换为原子操作

atomic_add(&arr[i], delta);
```

代码 3.9.6-2

循环体提取（ExtractLoopParallelBody）：

```
Function* ExtractLoopParallelBody(LoopInfo* loop) {

   // 1. 创建新函数

   Function* ParallelFunc = new Function(...);

// 2. 参数化处理

   for (Value* val : 外部引用值) {

      Val2Arg[val] = new Parameter(val->type);

   }
```

```
    // 3. 指令迁移
    for (BasicBlock* BB : 循环体) {
        BB->moveToFunction(ParallelFunc);
    }
    // 4. 生成调用点
    CallInst* call = new CallInst(ParallelFunc, 参数列表);
    return call;
}
```

代码 3.9.6-3

这个函数负责把循环体提取出来做成一个独立的并行函数，是循环并行化的核心步骤。首先它会创建一个新的函数对象，用来装循环体的代码。然后处理那些在循环里用到、但在外面定义的变量，把这些变量都变成新函数的参数，这样循环体搬出去后还能拿到这些值。

接下来把循环里的所有基本块整个搬到新函数里去，保持原来的控制流结构不变。最后在原位置生成一个调用指令，调用这个新创建的函数，并传给它需要的参数值。

这个过程中最麻烦的是要处理好那些跨循环内外的数据依赖，特别是循环索引变量和归约操作的处理要特别小心，稍不注意就会破坏程序的正确性。这种提取操作虽然增加了函数调用的开销，但为后续的并行执行打下了基础，让循环体可以独立地在不同线程上跑。

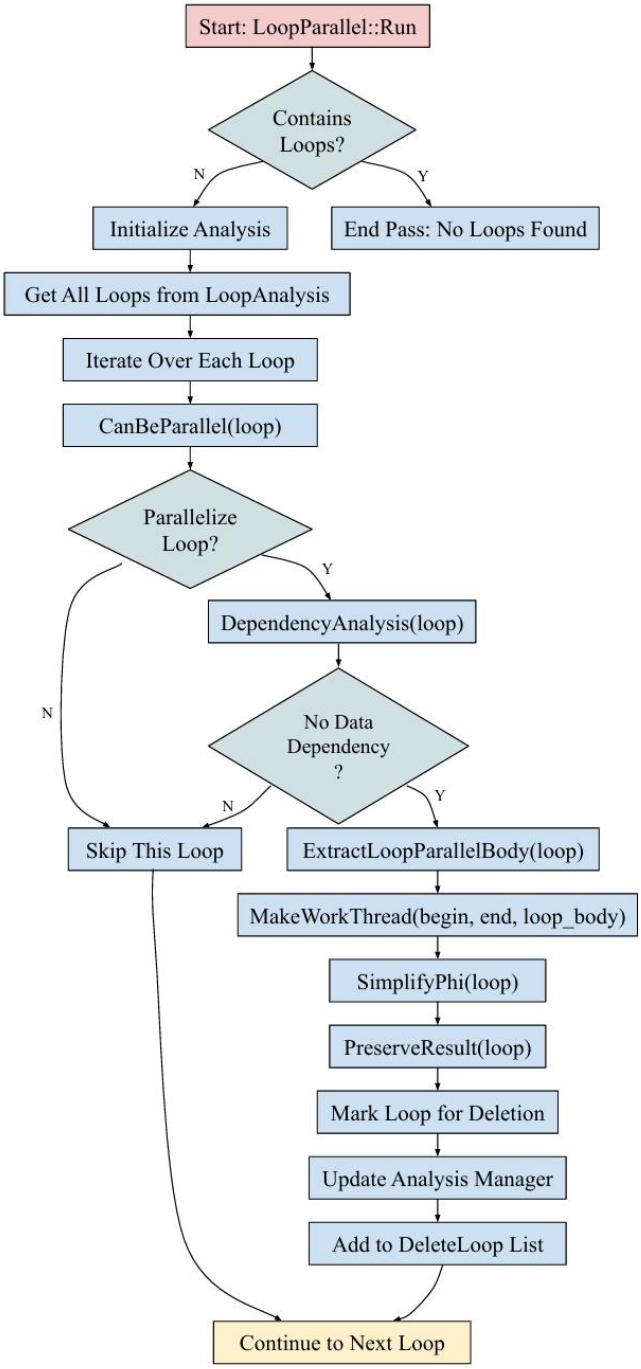

Start: LoopParallel::Run
Contains Loops?
N
Y
Initialize Analysis
End Pass: No Loops Found
Get All Loops from LoopAnalysis
Iterate Over Each Loop
CanBeParallel(loop)
Parallelize Loop?
Y
N
DependencyAnalysis(loop)
No Data Dependency ?
N
Y
Skip This Loop
ExtractLoopParallelBody(loop)
MakeWorkThread(begin, end, loop_body)
SimplifyPhi(loop)
PreserveResult(loop)
Mark Loop for Deletion
Update Analysis Manager
Add to DeleteLoop List
Continue to Next Loop

图 3.9.6-2

后端：RISC-V 目标代码生成（图 1）：

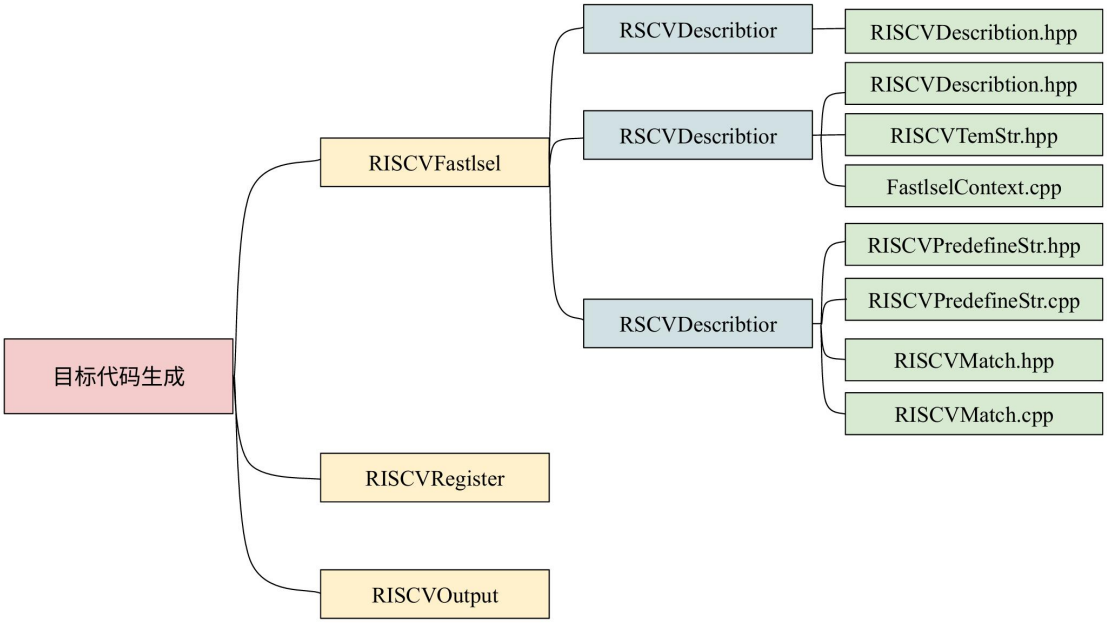


图 1

3.10 指令选择

在编译器设计中，将高级编程语言的语句转换成目标机器架构的具体指令的过程被称为指令选择。高级语言旨在提供一种接近自然语言的、易于人类理解的编程接口，而目标机器指令则是特定于硬件平台的低级命令，用于直接控制计算机硬件执行具体任务。无论是结构较为简单的编译器——它们直接将源代码翻译成目标机器码，还是具备复杂优化能力的编译器——这些编译器先将源代码转换为中间表示（IR）形式，进行一系列优化后再生成目标机器指令，都会经过指令选择，见图 3.10-1。

模式匹配：寻找所有可以匹配的指令序列。采用“单指令匹配”

模式选择：当有多个匹配序列存在的时候，根据需要选择其中的一个模式作为匹配结果。

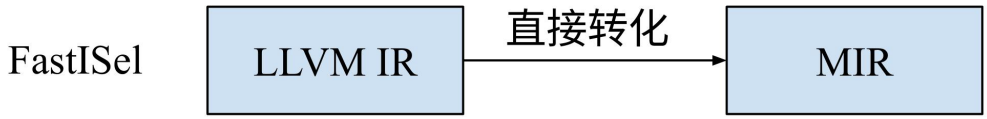


图 3.10-1

3.10.1 模式匹配的目的

中端的每一条语句都需要正确的传递到后端，在高级语言中可能表示简单的一条 int a = 10; 的 一条简单的赋值语句，在中端会被翻译成 alloca 和 store 语句，而到了后端又会对中端的 alloca 和 store 语句进行匹配，生成 li 和 sw 的汇编指令语句(见图 3.10.1-1)。

图 3.10.1 -1

3.10.2 核心的设计模式

中端传递过来的语句，后端需要针对中端传入的语句（具体语句类型见 代码 3.10.2-1）进行特定的处理。

```
中端的指令：
LoadInst
StoreInst
AllocaInst
RetInst
BinaryInst
ZextInst
SextInst
TruncInst
CallInst
CondInst
UnCondInst
MaxInst
```

```
MinInst
SelectInst
GepInst
FP2SIInst  // float -> int
SI2FPInst  // int -> float
```

代码 3.10.2-1 中端语句

后端的基础类的设计，RISCV 后端有自己的基础类，中端传入的每条语句在后端都会被分配指定的类进行承载。后端设计的类有助于之后的遍历与查找。

核心的基础设计类（见 代码 3.10.2-2）如下：

```
class RISCVOp
{
    std::string name;
public:
    RISCVOp() = default;
    RISCVOp(std::string _name):name(_name) {}
    RISCVOp(float tmpf)
    {
        uint32_t n;
        memcpy(&n, &tmpf, sizeof(float)); // 直接复制内存位模式
        name = std::to_string(n);
    }
    virtual ~RISCVOp() = default;
template<typename T>
    T* as()
    {
        return static_cast<T*> (this);
    }
```

```
    void setName(std::string _string)
    {
        name = _string;
    }

    std::string& getName()
    {
        return name;
    }
};
```

代码 3.10.2-2 基础类

我所设计的最基础类是 RISCVOp，之后的 Imm 类，和 Register（见代码 3.10.2-3）以及 BasicBlock ，和 Function，也都是继承自 RISCVOp 的实现。

```
class Imm: public RISCVOp
{
    Value* val;
    ConstIRFloat* fdata;
public:
    Imm(Value* _val): val(_val),RISCVOp(_val->GetName()) { }
    Imm(ConstIRFloat *_fdata) : fdata(_fdata),RISCVOp(_fdata->GetVal())
                              { }

    Imm(std::string name):RISCVOp(name) { }
};

// 虚拟实际寄存器封装到一起
class Register:public RISCVOp
```

```
{
public:
    static int VirtualReg;
 enum FLAG
    {
        real,
        vir
    };
    Register(std::string _name,bool Flag = vir):RISCVOp(_name)
    {
        if(Flag)
            VirtualReg++;
    }
};
```

代码 3.10.2-3 基础类 Register

有了这些基础的 class 的设计，只需要对中端存在与内存之中的每一条语句进行一个匹配（见代码 3.10.2-4 ）就可以很容易的得到最终的结果。

```
if(auto inst = dynamic_cast<Instruction*>(val))
{
       // RISCVInst inst;
       if(auto   Inst   =   dynamic_cast<LoadInst*>(inst))      return
CreateLInst(Inst);
       if(auto    Inst=    dynamic_cast<StoreInst*>(inst))       return
CreateSInst(Inst);
       if(auto   Inst=   dynamic_cast<AllocaInst*>(inst))        return
CreateAInst(Inst);
       if(auto   Inst=   dynamic_cast<CallInst*>(inst))         return
CreateCInst(Inst);
```

```
        if(auto    Inst=    dynamic_cast<RetInst*>(inst))          return
 CreateRInst(Inst);

        if(auto    Inst=    dynamic_cast<CondInst*>(inst))         return
 CreateCondInst(Inst);

        if(auto    Inst=    dynamic_cast<UnCondInst*>(inst))       return
 CreateUCInst(Inst);

        if(auto    Inst=    dynamic_cast<BinaryInst*>(inst))       return
 CreateBInst(Inst);

        if(auto    Inst=    dynamic_cast<ZextInst*>(inst))         return
 CreateZInst(Inst);

        if(auto    Inst=    dynamic_cast<SextInst*>(inst))         return
 CreateSInst(Inst);

        if(auto    Inst=    dynamic_cast<TruncInst*>(inst))        return
 CreateTInst(Inst);

        if(auto    Inst=    dynamic_cast<MaxInst*>(inst))          return
 CreateMaxInst(Inst);

        if(auto    Inst=    dynamic_cast<MinInst*>(inst))          return
 CreateMinInst(Inst);

  if(auto    Inst=    dynamic_cast<SelectInst*>(inst))            return
 CreateSelInst(Inst);

        if(auto    Inst=    dynamic_cast<GepInst*>(inst))          return
 CreateGInst(Inst);

        if(auto    Inst    =    dynamic_cast<FP2SIInst*>(inst)    )    return
 CreateF2IInst(Inst);

        if(auto    Inst    =    dynamic_cast<SI2FPInst*>(inst)    )    return
 CreateI2Fnst(Inst);

        assert("have no right Inst type to match it");
}
```

代码 3.10.2-4 匹配

3.10.3 匹配的难题

模式匹配的难题在于，中端的一条指令可能有多种的匹配方式，也可能对应后端的多条指令，并且后端需要支持的指令有很多，例如，类型转化的指令，判断的指令，循环的指令等待都需要实现。

这里针对 if 判断的指令（见 代码 3.10.3-1 ）进行分析与总结。

```
RISCVInst* RISCVContext::CreateCondInst(CondInst *inst)
{
    auto CmpInst = inst->GetPrevNode();
    RISCVInst* RInst = nullptr;
    auto condition = CmpInst->GetOperand(0);
    if (condition->GetType() == IntType::NewIntTypeGet())
    {
        if (CmpInst && CmpInst->IsCmpInst())  {
            RISCVInst *RInst = mapTrans(CmpInst)->as<RISCVInst>();
            auto cmpOp2 = RInst->getOpreand(0);
            // Instruction
            switch (CmpInst->GetInstId())
            {
            case Instruction::Eq:
                extraDealBrInst(RInst,  RISCVInst::_bne,  inst,  CmpInst,
cmpOp2);
                break;
            case Instruction::Ne:
                extraDealBrInst(RInst,  RISCVInst::_bqe,  inst,  CmpInst,
cmpOp2);
                break;
            case Instruction::Ge:
```

```
                extraDealBrInst(RInst, RISCVInst::_blt, inst, CmpInst,
cmpOp2);
                break;
 case Instruction::L:
                extraDealBrInst(RInst, RISCVInst::_bge, inst, CmpInst,
cmpOp2);
                break;
            case Instruction::Le:
                extraDealBrInst(RInst, RISCVInst::_bgt, inst, CmpInst,
cmpOp2);
                break;
            case Instruction::G:
                extraDealBrInst(RInst, RISCVInst::_ble, inst, CmpInst,
cmpOp2);
                break;
            default:
                break;
            }
        }
        else
            assert("other conditions???");
    }
     else if(condition->GetType() == FloatType::NewFloatTypeGet()) {
            extraDealBeqInst(RInst,RISCVInst::_beq,inst);
    }
    else {
        assert("may be other conditions");
    }
```

```
    return RInst;
}
```

代码 3.10.3-1 if 语句的匹配

只针对 if 语句来说，需要依次匹配他们的 Op 操作符，针对他们的操作符进行匹配，才可以生成正确的汇编代码。

3.10.4 效果展示

经过了之前的模式匹配，可以将绝大多数的语句转变成目标的汇编代码，也完成了此阶段的任务。

对于如下的代码（ 见代码 3.10.4-1）：

```
int main()
{
  int a = 10;
  float b = 20.0;
 if ( a > b )
    return b;
  else
    return a;
}
```

代码 3.10.4-1 C 演示代码

其在此阶段所生成的汇编代码如下（见代码 3.10.4-2 ）所示：

```
main:
    addi  sp,sp,-32
    sd  s0,24(sp)
    addi  sp,sp,32
.BB.0:
```

```
    li  %.0,10
    sw  %.0,-20(s0)
    li  t0,1101004800
    fmv.w.x  %.1,t0
    fsw  %.1,-24(s0)
    lw  %.2,-20(s0)
    flw  %.3,-24(s0)
    fcvt.s.w  %.4,%.3
    flt.s  %.5,%.4,%.3
    beq  %.5,zero,.BB.8
.BB.7:
    flw  %.6,-24(s0)
    fcvt.w.s  %.7,%.6,rtz
    mv  a0,%.7
    ld  s0,24(sp)
    addi  sp,sp,32
    ret
.BB.8:
    lw  %.8,-20(s0)
    mv  a0,%.8
    ld  s0,24(sp)
    addi  sp,sp,32
    ret
  .size  main, .-main
```

代码 3.10.4-2 汇编代码

在指令匹配阶段的任务就是将中端相应的指令转化成后端所对应的汇编指令，需要针对特定的语句进行特定的分析，以及特定的用后端的相应 class 类和

指令标识符进行标记。

3.11 寄存器分配——图着色

任务:根据活跃变量分析的结果,用图着色算法进行寄存器分配,将虚拟寄存器一 一映射到物理寄存器中。

3.11.1 图着色算法

图着色算法涉及将一组特定的颜色按照一定的约束条件分配给图形中的各个部分，其中核心的规则是任何两个相邻的节点（即有直接连接的边的节点） 不能被赋予相同的颜色。这一问题在多个实际应用领域有着重要的作用，例如:

频率分配：在无线通信中，不同电台或频道之间需要分配不同的频率以 避免信号干扰。寄存器分配：编译器优化过程中，尝试将程序中的虚拟寄存器映射到有 限的实际硬件寄存器上，同时保证不会在同一时刻将两个相互关联的变 量放置在同一个寄存器中。制作考试时间表：安排考试时间时确保没有学生会在同一时间段内参加 两门或多门课程的考试。

根据解决问题的方法不同，图着色算法可以大致分为两大类：

1. 精确算法 (Exact Algorithms)：这类算法旨在找到一个最优解，也就是 使用尽可能少的不同颜色来正确地对图进行着色。然而，由于图着色问 题是一个NP完全问题，对于大型图来说，精确算法的计算复杂度非常高，这使得它们在处理大规模数据集时往往不切实际。
2. 启发式或近似算法 (Heuristic or Approximation Algorithms)：这些 算法并不保证能够找到一个全局最优解，而是致力于在一个可接受的时 间范围内提供一个足够好的解决方案。对于非常大的图，这种方法通常 更

为实用。启发式算法通过一系列策略和技巧来快速得到一个可行解， 尽管这个解可能不是最优的。

在编译器优化领域，特别是在寄存器分配任务中，我们倾向于使用启发式 图着色算法。这是因为虽然我们希望尽可能有效地利用有限数量的物理寄存器，但更关键的是我们需要一个可以在合理时间内完成并能良好运行的解决方案。启发式方法允许我们在性能和效率之间取得平衡，确保编译过程能够在有 限的时间内完成，同时生成的代码也具有较高的执行效率。

3.11.2 图着色的目的

我们经过了指令的选择与匹配，但是其实我们的代码任然有很重要的部分没有处理，在指令选择的阶段，我所封装的寄存器类中的内容包括了虚拟寄存器和实际的物理寄存器，且在这个阶段我们所有的寄存器都是虚拟寄存器（见代码3.11.2-1），而且我们默认寄存器是无限的，但是实际上的寄存器却是极其宝贵而且有限的。

```
class Register:public RISCVOp
{
public:
    static int VirtualReg;
    enum FLAG
    {
        real,
        vir
    };
    Register(std::string _name,bool Flag = vir):RISCVOp(_name)
    {
        if(Flag)
```

```
            VirtualReg++;
        }
};
```

代码 3.11.2-1 Register 类

riscv 的寄存器（见图表 3.11.2-1）如下：

Register	ABI	Saver	作用
x0	zero	-	硬编码恒为 0
x1	ra	Caller	调用的返回地址
x2	sp	Callee	堆栈指针
x3	gp	-	全局指针
x4	tp	-	线程指针
x5-x7	t0-2	Caller	临时寄存器
x8	s0/fp	Callee	保存寄存器
x9	s1	Callee	保存寄存器
x10-11	a0-1	Caller	函数参数
x12-17	a2-7	Caller	函数参数
x18-27	s2-11	Callee	保存寄存器
x28-31	t3-6	Caller	临时寄存器

图表 3.11.2-1

寄存器分配是生成目标的汇编代码最重要的一部分，只有分配了对应的目标寄存器才可以在相应的目标架构机器上进行运行，下面介绍的就是实现的总体逻辑了。

3.11.3 总体代码逻辑

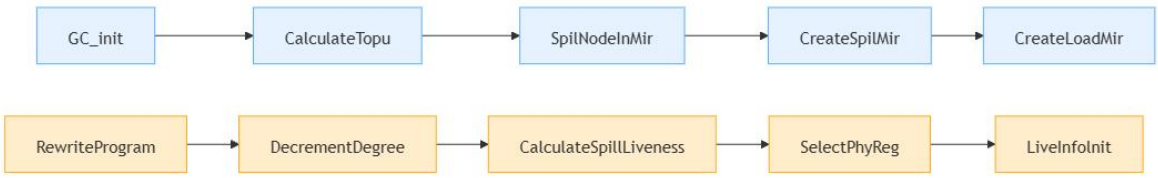


图 3.11.3-1

```
void GraphColor::RunOnFunc()
{
bool condition = true;
GC_init(); // 初始化图着色相关结构
for (auto b : *m_func)
{
 CalCulateSucc(b); // 计算每个基本块后继用于构建控制
CaculateTopu(m_func->GetFront()); // 对基本块做拓补排序,计算支配关系等
  std::reverse(topu.begin(), topu.end()); // 逆序方便数据流分析
while (condition)
{ condition = false;
CaculateLiveness(); // 活跃变量分析,构建干涉图
MakeWorklist(); // 将变量分到不同的工作集里（简化、冻结、合并、溢出）
do {
if (!simplifyWorkList.empty())
{
    simplify(); // 简化图，删除度数小于寄存器数的节点
} else if (!worklistMoves.empty())
{
coalesce(); // 合并 move 相关的变量，减少 move 指令
```

```
 } else if (!freezeWorkList.empty())

 { freeze(); // 冻结 move 指令相关的节点，改为普通节点

 } else if (!spillWorkList.empty())

 { spill(); // 溢出某个变量到内存 } }

  while     (!simplifyWorkList.empty()     ||     !worklistMoves.empty()
   || !freezeWorkList.empty() || !spillWorkList.empty());

      for (auto sp : SpillStack) { // 这是处理 spilled nodes 的部分 auto it
   = std::find(selectstack.begin(), selectstack.end(), sp); // 看它是否在
   着色选择栈中

      if (it == selectstack.end())

         assert(0);

         selectstack.erase(it);

 }

     std::reverse(SpillStack.begin(),SpillStack.end());

     selectstack.insert(selectstack.end(),          SpillStack.begin(),
SpillStack.end());

      AssignColors(); // 实际分配颜色(物理寄存器)

     if (!spilledNodes.empty())

     { SpillNodeInMir(); // 将spill节点在MIR中插入load/store指令 condition
       = true;

     }

}

      RewriteProgram();

 }
```

代码 3.11.3-1

该函数RunOnFunc（图 3.11.3-1）体现了图着色寄存器分配的整体逻辑（见代码 3.11.3-1），具体如下:初始化之后,遍历基本块计算每个基本块的后继,计算从第一个块开始 的拓补排序,反转拓补排序结果,以便从最后一个基本块开始处

理。主循环中，condition = false;：假设在这一轮迭代中不需要再进行溢出操作。

CaculateLiveness();：计算每个点的活跃信息（哪些变量在哪个点是活跃的），MakeWorklist();：根据活跃信息创建工作列表（worklists），这些列 表包含简化（simplify）、共用（coalesce）、冻结（freeze）和溢出（spill）的工作项，内层 do-while循环尝试尽可能多地简化图、合并颜色、冻结节点和选择溢出节点，直到所有的工作列表为空。根据不同工作列表的状 态调用相应的方法：simplify(), coalesce(), freeze(), 和 spill()。接下 来处理栈溢出，如果有溢出发生，更新 selectstack 以反映最新的溢出节点位 置，SpillStack 被反转并插入到 selectstack 的末尾，确保最先被溢出的节 点最后被重新考虑，接下来分配颜色，尝试为剩余未着色的节点分配颜色（即实际寄存器），如果 spilledNodes 不为空，则意味着还有溢出节点需要处理，因此设置 condition = true 继续下一轮迭代。一旦所有寄存器都已成功 分配颜色（包括任何必要的溢出处理），就重写程序以反映新的寄存器分配。

3.11.4 实现过程

图着色实现的核心函数（见代码 3.11.4-1）。

```
ValsInterval.clear();
freezeWorkList.clear();
worklistMoves.clear();
simplifyWorkList.clear();
spillWorkList.clear();
spilledNodes.clear();
initial.clear();
coalescedNodes.clear();
constrainedMoves.clear();
coalescedMoves.clear();
frozenMoves.clear();
coloredNode.clear;
AdjList.clear();
selectstack.clear;
belongs.clear();
```

```
activeMoves.clear();
alias.clear();
RegType.clear();
InstLive.clear();
Precolored.clear();
color.clear();
moveList.clear();
instNum.clear();
RegLiveness.clear();
```

代码 3.11.4-1

CaculateTopu ()：拓补排序，用于后续的寄存器分配过程。（拓扑排序是对 图中的节点进行排序的一种方法，使得对于图中的每一条有向边 (u, v)，节点 u 都出现在节点 v 之前。）它表示的是基本块的执行顺序：在基本块 mbb 被 执行之前，所有它的后继块必须已经执行过。然后 reserve 之后方便后续操作。

SpillNodeInMir ()：该函数用于在生成的中间代码（MIR，Machine Intermediate Representation）中处理被溢出（spill）到内存的寄存器节 点。包括 temps 集合的初始化、遍历基本块（topu）、跳过（函数调用 call，返回 ret，跳转指令_j）控制流指令（因为这些指令并不会涉及溢出操作，因 此直接跳过）、处理溢出的定义、处理溢出操作数、继续处理下条指令、清空 溢出节点集合，准备进行下一步的寄存器分配或其他处理。

CreateSpillMir ()：这个函数的主要作用是生成一个存储（spill）指令，用 于将虚拟寄存器的数据溢出到内存中。首先进行类型检查与转换使用 dynamic_cast 将传入的操作数 spill 转换为 VirRegister * 类型，并断言转 换必须成功，确保溢出的是一个虚拟寄存器，检查是否对于给定的虚拟寄存器 有特殊的使用情况（通过 GetSpecialUsageMOperand 方法）。如果有，创建一 个新的 MIR 指令来标记该寄存器为已死亡，并创建一个新的虚拟寄存器，然后 将其添加到 temps 集合中，最后返回这个 MIR 指令。如果没有特殊用途，则 创建一个新的虚拟寄

存器 reg 并根据原始寄存器的类型初始化它，同时将其加 入 temps 集合中。根据虚拟寄存器的数据类型选择正确的存储指令（如 _sd 或 _fsw），并创建相应的 RISCVMIR 指令对象 sd，然后构建存储指令 sd并返 回结果，用于后续生成机器代码。

CreateLoadMir()：该函数的作用是生成一个加载（load）指令，用于将溢出的 虚拟寄存器从内存中加载回寄存器。具体过程与CreateSpillMir类似。

RewriteProgram()：该函数作用是遍历程序中的所有基本块，并将虚拟寄存器 替换为物理寄存器（根据 color 映射）。同时，还会删除冗余的移动指令。遍 历所有基本块 topu 中的指令（MIR）、跳过函数调用指令（call）（因为函数 调用不会涉及寄存器替换）、替换虚拟寄存器为物理寄存器、删除冗余移动指 令（如果指令是 mv 或 _fmv_s（即寄存器间的移动指令），并且定义和操作数 相同（即没有实际的变化），则删除该指令，避免冗余。）。

DecrementDegree ()：该函数的作用是减少一个目标节点的度数，并且根据度 数变化后的状态更新相关的工作列表，如 spillWorkList、freezeWorkList 和 simplifyWorkList。该函数适用于两种类型的寄存器：整数寄存器见代码 3.10.3-1 （riscv_i32, riscv_ptr, riscv_i64）和浮点寄存器（riscv_float32）。

```
  if (target->GetType() == riscv_i32 || target->GetType() ==
riscv_ptr || target->GetType() == riscv_i64)

{
    if (Degree[target] == (GetRegNums(riscv_i32) -1)) {
          //  ...
    }
}
```

代码 3.11.4-2

这个条件判断适用于整数类型的寄存器。检查目标节点的度数是否等于整

数寄存器可用寄存器数量减去 1（即当该节点的邻居数量为寄存器数量减一时，表示该节点的邻居可能会使用一个寄存器，因而需要检查并更新相关状态）。

```
auto x = Adjacent(target);
std::unordered_set<MOperand>          tmp(x.begin(),          x.end());
tmp.insert(target);
```

代码 3.11.4-3

通过 Adjacent(target) 获取目标节点的邻居（即与目标节点有干扰的节点）。将目标节点和其邻居节点都放入 tmp 集合中。

```
for (auto node : tmp)
    for (auto mov : MoveRelated(node))
    {
        if (activeMoves.find(mov) != activeMoves.end())
        {
              activeMoves.erase(mov);PushVecSingleVal(worklistMoves,
          mov);
        }
}
```

代码 3.11.4-4

对于 tmp 中的每个节点,检查是否存在相关的移动操作(即该节点是否与 其他节点有移动操作)。如果存在并且该操作是活动的（即在 activeMoves 中），则将其移出 activeMoves 并加入到 worklistMoves。

```
        spillWorkList.erase(target);
        if (MoveRelated(target).size() != 0)
        {
            freezeWorkList.insert(target);
        }
        else
        {
           simplifyWorkList.push_back(target);
```

```
        }
```

代码 3.11.4 - 5

将目标节点从 spillWorkList 中移除，表示不再需要溢出处理。

如果目标节点有与其他节点相关的移动操作,将其添加到 freezeWorkList(冻 结工作列表)，否则加入到 simplifyWorkList（简化工作列表)。

对于浮点寄存器（riscv_float32)：

{ else if (target->GetType() == riscv_float32) {if (Degree[target] == (GetRegNums(riscv_float32) - 1)) }

如果目标节点是浮点类型寄存器(riscv_float32)，则类似地判断其度数 是否符合条件，并进行与整数寄存器类似的处理。

CaculateSpillLiveness ()：该函数用于计算溢出节点的生存区间，并为这些 节点分配一个溢出标记（spill token)。溢出标记用于追踪哪些节点是可以一 起溢出的（即它们在程序中没有干扰)。

SelectPhyReg ()：这个函数的目的是根据虚拟寄存器（vreg）和其类型（ty）选择一个合适的物理寄存器（PhyRegister)。它在寄存器分配过程中，根据是否存在移动操作、寄存器类型和当前寄存器的使用情况(通过 assist 集合表示）来做出选择。

LiveInfoInit()：该函数用于清空各成员变量，避免数据残留影响后续的计算。

3.11.5 结果展示

在经过图着色算法的处理之后，我们将之前的每一个虚拟寄存器都替换成了对应的物理寄存器，也依照函数调用的规约，将不能够被寄存器所保证的参数将其溢出到了栈帧当中，通过栈帧进行传递。

以如下的 C 语言例子（见码 3.11.5-1）进行举例：

```
int main()
```

```
{
  float a = 10.0;
  if ( a != 20.0)
    a = 5.0;
  else
    a = 30.0;
  return 0;
}
```

代码 3.11.5-1

如何没有寄存器分配所生成的汇编代码（见代码 3.11.5-2 ）：

```
main:
.LBB0:
   addi sp, sp, -32
   sd ra, 24(sp)
   sd s0, 16(sp)
   addi s0, sp, 32
   li t0, 1084227584
   fmv.w.x %6, t0
   li t0, 1106247680
   fmv.w.x %5, t0
   li t0, 1101004800
   fmv.w.x %4, t0
   li t0, 1092616192
   fmv.w.x %0, t0
   fsw %0, -20(s0)
   flw %1, -20(s0)
```

```
   feq.s %2, %1, %4
   seqz %3, %2
beq %3, zero, .LBB2
.LBB1:
   fsw %6, -20(s0)
   j .LBB3
.LBB2:
   fsw %5, -20(s0)
.LBB3:
   li a0, 0
   ld ra, 24(sp)
   ld s0, 16(sp)
   addi sp, sp, 32
   ret
    .size main, .-main
```

代码 3.11.5-2 虚拟寄存器

进行了寄存器分配生成的汇编代码（代码 3.11.5-3）如下：

```
main:
.LBB0:
    addi sp, sp, -32
    sd ra, 24(sp)
    sd s0, 16(sp)
    addi s0, sp, 32
    li t0, 1084227584
    fmv.w.x ft1, t0
    li t0, 1106247680
    fmv.w.x ft3, t0
```

```
    li t0, 1101004800
    fmv.w.x ft2, t0
    li t0, 1092616192
    fmv.w.x ft0, t0
    fsw ft0, -20(s0)
    flw ft0, -20(s0)
    feq.s t2, ft0, ft2
    seqz t2, t2
    beq t2, zero, .LBB2
.LBB1:
    fsw ft1, -20(s0)
    j .LBB3
.LBB2:
    fsw ft3, -20(s0)
.LBB3:
    li a0, 0
    ld ra, 24(sp)
    ld s0, 16(sp)
    addi sp, sp, 32
    ret
  .size main, .-main
```

代码 3.11.5-3 汇编

3.12 栈帧的分配和后端优化

进行目标代码翻译时，对于函数的调用需要考虑函数栈帧的开辟，在进入函数前计算好栈帧需要开辟的大小，用来保存局部变量或者是临时变量，并且需要

记录每个变量相对于栈帧首地址的偏移量。在进入函数内部调用到的变量需要在内存中查找，这时候需要提供变量在内存中的地址（也就是相对偏移量）

编译器后端优化是编译器优化过程中的重要环节，通过对中间代码的精细调整，可以显著提升最终生成的目标代码的性能。与中端优化不同，后端优化更关注于具体的硬件架构和算子级别的优化，例如指令向量化，利用 SIMD 指令，对多个数据进行并行计算，提高计算密集型任务的性能。

3.12.1 RISC-V 的函数栈帧

函数栈帧具有重要的意义，对 CSR（Callee Saved Register）进行处理，这是满足调用约定最主要的工作之一，完成栈帧布局，在栈帧布局确定之前栈对象通过栈索引进行访问（假定栈对象存放在一个数组中），当栈帧布局确定后，栈对象通过基于栈寄存器的偏移进行访问；函数前言（Prologue）/后序（Epilogue）生成，前言通常是为函数建立新的栈帧，后序则是销毁函数栈帧。
在我们的实现中，我们创造了对应的类来代表前言和后序的栈帧，并且在函数当中与他的前言和后序栈帧绑定（见代码 3.12.1-1）。

```
class RISCVPrologue:public RISCVOp
{
    using Instptr = std::shared_ptr<RISCVInst>;
    // std::vector<Instptr> proloInsts;
    typedef std::vector<Instptr> ProloInsts;
    ProloInsts proloInsts;
public:
    ProloInsts& getInstsVec()
    {
        return proloInsts;
    }
```

```
};


class RISCVEpilogue:public RISCVOp
{
    using Instptr = std::shared_ptr<RISCVInst>;
    // std::vector<Instptr> proloInsts;
    typedef std::vector<Instptr> EpilogueInsts;
    EpilogueInsts epiloInsts;
public:
    EpilogueInsts& getInstsVec()
    {
        return epiloInsts;
    }
};
```

代码 3.12.1-1 前言后序

函数之中放有前言与后序的指针（代码 3.12.1-2 ），方便找到该函数自己的前言与后序。

```
class RISCVFunction:public RISCVOp, public List<RISCVFunction, RISCVBlock>
{
    Function* func;
    RISCVBlock* exit;
    std::shared_ptr<RISCVPrologue> prologue;
    std::shared_ptr<RISCVEpilogue> epilogue;
}
```

代码 3.12.1-2

3.12.2 生成栈帧

针对简单的的样例（见代码 3.12.2-1 ）进行分析，只要有函数的存在，就可以生成对应的函数栈帧，前言与后序。

```
int main()
{
return 0;
}
```

代码 3.12.2-1 测试代码

生成对应的汇编代码（代码 3.12.2-2 ）。

```
main:
    addi  sp,sp,-32
    sd  s0,24(sp)
    addi  sp,sp,32
.BB.0:
    li  a0,0
    ld  s0,24(sp)
    addi  sp,sp,32
    ret
  .size  main, .-main
```

代码 3.12.2-2

3.12.3 后端优化

后端的优化与中端区别较大，在将中端代码逐条翻译成后端之后，可能会有冗余语句的多出，并且并不是所有的语句在后端都是可以支持的，phi 函数是只可以在中端进行支持，后端需要有专门的 phi 函数消除的优化去消除 phi。除此之外，后端的优化更多的是针对目标架构的，很多和处理器所支持的寄存器相关。

后端像中端一样由基础的优化类继承（见 代码 3.12.3-1 ）而来。

```
class BackendPassBase
{
public:
    virtual bool run()=0;
    BackendPassBase() = default;
    ~BackendPassBase() = default;
};
```

代码 3.12.3-1

phi 函数的消除（见代码 3.12.3-2）是因为，riscv 并没有支持 phi 函数的汇编指令。

```
class PhiEliminate : public BackendPassBase
{
    Function* func;
public:
    PhiEliminate(Function* _func) :func(_func) { }

    bool run() override;
};
```

代码 3.12.3-2

我们实现的编译器，在此时已经失去其在中端所保持的 Value，Use，User 的设计模式，其他的后端优化相对来说比较困难去实现，但是针对于特定的硬件，我们也会去实现相关的优化。

存在问题与解决方案

1. 复杂数组初始化的适配：中间代码层面很难在只有 store语句的情况下 完成复杂数组的初始化。考虑将复杂数组的初始化直接交给后端处理，可以直 接使用区间赋值操作。

2. Type System：由于中端优化的需要（要求判断类型是否相同使用指针 比较）和考虑到语法可能新增的struct语句，现在正在进行一定程度的微调。 考虑Type System使用有向图的形式存储。

3. 活跃变量分析仿照LLVM在 MachineCode 的基础上实现,但现决定在前 端生成的 IR上实现，还需要添加处理Phi 函数的情况，并根据Value-Use-User 关系修改相应的参数和函数内部的具体实现，并修改接口以便其他Part 使用。

4. 后端部分寄存器分配算法还有部分问题，需要更正调整。

5. 后端优化尚未完成,需要结合 RISC-V 处理器的目标架构进行特定的优化，例如结合向量化，实现 SIMD 优化。

执行情况与完成度

前端部分：高效实现了对 sysy语言的词法分析、语法分析、中间代码生成，核心类的设计。前端能全部跑通 sysy 官方给的功能样例。

中端部分：目前正在进行标量优化与循环优化，已经完成了死代码消除、部分冗余消除、CFG 简化等标量优化以及循环展开、循环并行和循环删除等循环优化部分，后续优化在逐步推进。

后端部分：后端部分基本实现，完成了指令的匹配，寄存器分配，栈帧生成等主要任务。

分工协作与交流情况

成员	贡献
刁寒	后端目标代码生成，指令选择，寄存器分配
刘铭辰	中端的循环优化
文雯	词法分析,语法分析,符号表设计,中间代码生成，CFG 划分
范耀晖	中端的标量优化

心得体会

编译器的设计与实现是一个非常具有挑战性的课题。在这个过程中，我们 不仅巩固了课堂上学习到的知识，也涉及到了许多未曾接触过的知识点，从中 收获了大量新的技能和理解。编译器项目的实现要求我们将复杂的任务拆解为 多个可管理的小问题，并通过逐步攻克这些问题，最终实现了复杂系统的构

建。这不仅加深了我们对编译原理的理解，还极大提升了我们的编程能力。

我们学习了最先进的编译器项目 llvm，通过学习 llvm 了解到了 Value，Use，User 等一套核心的设计模式，并且在中端了解到了 mem2reg，DCE，分支与循环相关的诸多优化，还在后端了解到了 RISC-V 的设计理念，涉及函数的调用和规约，函数栈帧的建立与销毁，如此诸多内容都让我们受益匪浅。

在设计与处理我们编译器的过程中，我们也了解到了很多好的设计模式，例如 C++的单例模式，诸多的模式都可以很好的让我们更好的理解 C++语言，更加熟悉 C++的很多特性。并且在此过程中，我们还实践了我们大一所学习的数据结构的链表，树，图等诸多内容。

此次的项目实践也让我们意识到了团队分工和合作的重要性，针对一个很大的项目，我们大家合理分工，最终才让我们可以很好的完成这个项目。

参考文献

[1] 周尔强 周帆 韩蒙 陈文宇，编译技术

[2] 彭成寒 李灵 戴贤泽 王志磊 俞佳嘉 深入理解LLVM:代码生成

[3] [美] Keith D.Cooper Linda Torczon， 编译器设计（第二版）

[4] [美] Alfred V.Aho Monica-S.Lam等，编译原理

[5] [美] David Patterson Andrew Waterman等，RISC-V手册

[6] THOMAS J. SAGER and SHIJEN LIN， AN IMPROVED EXACT GRAPH COLORING ALGORITHM

[7] ANDREW W.APPEL Princeton University with MAIA GINSBURG，Modern Compiler Implementation in C

[8] [美]Steven S.Muchnick，高级编译器设计与实现