

# 编译2025春：FDMJ2025编译器 期末报告

22307110035 蒋会成

## 编译2025春：FDMJ2025编译器 期末报告

### 一、引言

- 1.1 报告内容概述
- 1.2 目标读者
- 1.3 编译器的背景与重要性
- 1.4 报告的组织方式
- 1.5 编译器的实现环境
- 1.6 编译器整体流程

### 二、词法解析

#### 2.1 词法分析

- 2.1.1 定义FDMJ2025词法规则
- 2.1.2 扫描输入与词法单元识别
- 2.1.3 示例

#### 2.2 语法分析

- 2.2.1 上下文无关文法 (CFG) 概述
- 2.2.2 FDMJ2025支持的CFG语法规则
- 2.2.3 AST节点类的详细说明
- 2.2.4 Bison的语法规则
- 2.2.5 符号优先级和结合性
- 2.2.6 错误处理
- 2.2.7 示例

### 三、语义分析

#### 3.1 访问者模式

- 3.1.1 访问者模式概述
- 3.1.2 访问者模式的优势
- 3.1.3 访问者模式的实现(以AST\_Name\_Map\_Visitor为例)

#### 3.2 符号表构建

#### 3.3 语义上下文记录

#### 3.4 FDMJ类型说明

- 3.4.1 数据类型
- 3.4.2 值类型
- 3.4.3 名称类型
- 3.4.4 类型转换

#### 3.5 语义分析与类型检查

#### 3.6 示例

### 四、转换为中间表示 (IR)

#### 4.1 关键数据结构

- 4.1.1 ASTToTreeVisitor 类成员变量
- 4.1.2 类表 (Class Table)
- 4.1.3 方法变量表 (Method Variable Table)
- 4.1.4 接口函数实现

#### 4.2 TREE节点类的详细说明

#### 4.3 运算与数组类型的转换

- 4.3.1 运算
- 4.3.2 程序结构相关
- 4.3.3 数组操作

#### ▼ 4.4 关于类类型的转换

- 4.4.1 关键处理
- 4.4.2 类操作转换

#### ■ 4.5 示例

### ▼ 五、指令选择以生成四元组表示

#### ■ 5.1 四元组中间语言的描述

#### ▼ 5.2 准备工作：IR的规范化

- 5.2.1 规范化的目的
- 5.2.2 规范化的过程
- 5.2.3 规范化IR的示例

#### ▼ 5.3 关键结构

- 5.3.1 Tree2Quad类
- 5.3.2 Quad基本类型QuadTerm
- 5.3.3 变量管理

#### ■ 5.4 主要转换逻辑

#### ■ 5.5 示例

### ▼ 六、将Quad转换为SSA形式

#### ■ 6.1 静态单赋值形式

#### ■ 6.2 准备工作：Quad的分块

#### ▼ 6.3 将分块后的Quad转成SSA形式

- 6.3.1 删除不可抵达的代码块
- 6.3.2 放置Phi函数
- 6.3.3 重命名变量
- 6.3.4 清理未使用的Phi函数

#### ■ 6.4 示例

### ▼ 七、SSA的活跃性分析

#### ▼ 7.1 用于活跃性分析的数据结构

- 7.1.1 ControlFlowInfo
- 7.1.2 DataFlowInfo

#### ▼ 7.2 控制流图分析

- 7.2.1 控制流图的构建
- 7.2.2 控制流图的计算

#### ▼ 7.3 活跃性分析

- 7.3.1 活跃性
- 7.3.2 数据流方程
- 7.3.3 计算活跃性的算法

#### ▼ 7.4 块流图和干扰图分析

- 7.4.1 块流图
- 7.4.2 干扰图
- 干扰图的计算

#### ■ 7.5 示例

### ▼ 八、寄存器分配

#### ■ 8.1 寄存器分配准备

#### ▼ 8.2 寄存器分配的关键要点

- 8.2.1 核心数据结构
- 8.2.2 入口函数

#### ▼ 8.3 寄存器分配步骤

- 8.3.1 简化 (simplify)
- 8.3.2 合并 (coalesce)
- 8.3.3 冻结 (freeze)
- 8.3.4 溢出 (spill)
- 8.3.5 选色 (select)

- 8.4 示例
- ▼ 九、RPI (ARM) 汇编代码生成
  - 9.1 堆栈组织框架
  - ▼ 9.2 四元式到RPI指令的转换
    - 9.2.1 转换函数概述
    - 9.2.2 溢出处理机制
    - 9.2.3 具体转换逻辑
  - 9.3 示例
- ▼ 十、总结
  - 10.1 本学期工作总结
  - ▼ 10.2 编译器构建与使用说明
    - 10.2.1 目录结构
    - 10.2.2 使用方法
    - 10.2.3 注意事项

# 一、引言

## 1.1 报告内容概述

本报告详细介绍了我在编译H（2025春）课程学习中构建的完整编译器的实现过程。该编译器的目标是将FDMJ语言的源代码编译为RPI (ARM) 汇编代码。编译器的实现涉及多个阶段，每个阶段都对应着编译过程中一个关键步骤。从源代码的解析到最终的汇编代码生成，整个过程通过精心设计和实现，以确保编译器的正确性和高效性。

报告将详细阐述编译器的各个组成部分，包括解析器、类型检查器、中间表示 (IR) 生成器、指令选择器、静态单赋值 (SSA) 转换器、寄存器分配器以及汇编代码生成器。每个部分的实现细节、数据结构、算法以及它们之间的交互关系都将在报告中进行详细说明。

此外，报告还将通过具体的FDMJ示例程序，展示编译器在每个阶段的处理过程和输出结果。这些示例将帮助读者更好地理解编译器的工作原理和实现细节。

## 1.2 目标读者

本报告的目标读者是计算机科学专业的学生，特别是那些正在学习编译原理课程的学生。这些读者可能对编译器的实现细节不太熟悉，因此报告的撰写方式注重清晰性和易懂性，旨在帮助理解编译器的各个组成部分及其工作原理。

报告假设读者已经具备的C++编程基础和计算机科学知识，但对编译器的具体实现过程可能缺乏深入了解。因此，报告将从基础概念入手，逐步深入到具体的实现细节，确保读者能够跟上思路并理解编译器的全貌。

## 1.3 编译器的背景与重要性

编译器是计算机科学中的一个重要工具，它将高级语言编写的程序转换为计算机可以直接执行的低级语言代码。编译器的设计和实现不仅涉及复杂的编程技术，还涉及到对语义、数据结构、算法以及计算机体系结构的深刻理解。

在现代软件开发中，编译器的作用越来越重要。它不仅需要正确地将源代码转换为目标代码，还需要优化代码的性能，减少运行时间和内存占用。此外，编译器还需要具备良好的错误检测和报告能力，帮助程序员快速定位和修复代码中的问题。

## 1.4 报告的组织方式

本报告将按照编译器的实现阶段进行组织，每个阶段都有详细的解释和示例程序的处理过程。以下是报告的主要结构：

1. 引言：介绍报告的背景、目标读者、编译器的整体过程以及报告的组织方式。
2. 解析：详细说明词法分析和语法分析的实现，以及如何生成抽象语法树 (AST)。
3. 类型检查：介绍类型检查的机制，符号表的构建以及类型错误的报告。
4. 中间表示 (IR) 生成：说明如何将AST转换为IR+，以及转换过程中构建的表格。
5. 指令选择与四元组生成：介绍IR+到四元组的转换过程，使用的瓦片以及指令的“定义”和“使用”集合。

6. **静态单赋值 (SSA) 转换**: 说明SSA的实现过程，包括Phi函数的插入。
7. **四元组程序准备**: 介绍如何进行活跃性分析，构建控制流图、块流图和干扰图。
8. **寄存器分配**: 说明寄存器分配的步骤，包括简化、合并、冻结和溢出处理。
9. **RPI (ARM) 汇编代码生成**: 说明如何生成最终的汇编代码，包括堆栈框架的组织和溢出处理。
10. **结论**: 总结本学期的工作，并提供编译器的构建和使用说明。

## 1.5 编译器的实现环境

本编译器的实现环境是64位Ubuntu 20 (或更高版本) Linux操作系统，基于x86架构。开发过程中使用了以下工具：

- **Lex/Yacc**: 用于词法分析和语法分析。
- **Make/CMake**: 用于项目构建和管理。
- **QEMU**: 用于模拟RPI (ARM) 环境，运行生成的汇编代码。

## 1.6 编译器整体流程



## 二、词法解析

解析阶段是编译器设计中的关键步骤，它将源代码转换为编译器可以进一步处理的内部表示形式。解析阶段分为两个主要部分：**词法分析**和**语法分析**。

### 2.1 词法分析

词法分析是编译过程的第一步，它将源代码文本分解成一系列有意义的词法单元 (tokens)，这些tokens是源代码中的基本构造块，如关键字、标识符、字面量和操作符。在本项目中，我们使用Flex工具来生成词法分析器，以处理FDMJ语言的词法规则。

#### 2.1.1 定义FDMJ2025词法规则

##### • FDMJ2025支持的关键字

**字**: public、int、main、return、if、else、while、continue、break、true、false、length、getint、getch、getarray、putint、putch、putarray、starttime、stoptime、this、class、extends。

##### • FDMJ2025的标识符

- **定义**: 由字母或下划线开头，后跟任意数量的字母、数字或下划线。
- **示例**: x、variableName、\_classVar、myFunction。

##### • FDMJ2025的数字常量

- **定义**: 非负整数，可以以 0 开头，或以 1-9 开头后跟任意数量的数字。
- **示例**: 0、123、4567。

##### • FDMJ2025支持的操作符:

###### ◦ 单元运算符

- ! (逻辑非)
- - (负号)

###### ◦ 双元运算符

###### ▪ 算术运算符:

- + (加法)
- - (减法)
- \* (乘法)
- / (除法)

###### ▪ 关系运算符:

- == (等于)
- != (不等于)
- < (小于)

- `<=` (小于等于)
- `>` (大于)
- `>=` (大于等于)
- **逻辑运算符:**
  - `&&` (逻辑与)
  - `||` (逻辑或)

• **FDMJ2025的分隔符:** 分号 (`;`)、逗号 (`,`)、圆括号 (`(`、`)`)、花括号 (`{`、`}`)、方括号 (`[`、`]`)。

• **FDMJ2025的注释:**

- **单行注释:** 以 `//` 开头，直到行尾。
- **多行注释:** 以 `/*` 开头，以 `*/` 结束。

## 2.1.2 扫描输入与词法单元识别

词法分析器从源代码的开始扫描，逐个字符地读取输入，并根据定义的词法规则识别词法单元。在借助Flex工具基础上，该过程的重点实现方案如下：

### 1. 定义词法单元类型

在 `AST_YSTYPE` 类中，定义词法单元可以持有的各种类型的值。这些类型包括整数 (`i`)、标识符 (`s`)、各种表达式和声明的指针等。这种设计使得词法分析器能够灵活地处理不同类型的词法单元。

### 2. 复制词法单元值和位置

`copyValue` 函数用于将识别出的词法单元的值复制到 `yyval` 中，而 `copyLocation` 函数用于将词法单元的位置信息复制到 `yyloc` 中。这两个函数在每次匹配模式后被调用，以便记录词法单元的值和位置。

### 3. 词法分析器的API

`ASTLexer` 类提供了词法分析器的API，包括构造函数和 `yylex` 函数。构造函数接受一个输入流和一个调试标志，用于初始化词法分析器。`yylex` 函数是词法分析器的主要接口，它接受两个参数：`yyval` 和 `yyloc`。`yyval` 用于存储识别出的词法单元的值，`yyloc` 用于存储词法单元的位置信息。

### 4. 处理注释和空白

在词法分析阶段，我们需要忽略源代码中的注释和空白字符。我们定义了两个状态 `COMMENT1` 和 `COMMENT2` 来处理单行和多行注释。当词法分析器进入这些状态时，它将忽略所有输入直到遇到注释结束标记。

## 2.1.3 示例

考虑以下FDMJ源代码片段：

```
public int main() {
    int x = 5;
    while (x > 0) {
        putint(x);
        x = x - 1;
    }
}
```

词法分析过程结果如下：

1. 识别 `public` 作为关键字。
2. 识别 `int` 作为关键字。
3. 识别 `main` 作为标识符。
4. 识别 `(` 作为分隔符。
5. 识别 `)` 作为分隔符。
6. 识别 `{` 作为分隔符。
7. 识别 `int` 作为关键字。
8. 识别 `x` 作为标识符。
9. 识别 `=` 作为操作符。
10. 识别 `5` 作为数字常量。
11. 识别 `;` 作为分隔符。

12. 识别 `while` 作为关键字。
13. 识别 `(` 作为分隔符。
14. 识别 `x` 作为标识符。
15. 识别 `>` 作为操作符。
16. 识别 `0` 作为数字常量。
17. 识别 `)` 作为分隔符。
18. 识别 `{` 作为分隔符。
19. 识别 `putint` 作为关键字。
20. 识别 `(` 作为分隔符。
21. 识别 `x` 作为标识符。
22. 识别 `)` 作为分隔符。
23. 识别 `;` 作为分隔符。
24. 识别 `x` 作为标识符。
25. 识别 `=` 作为操作符。
26. 识别 `x` 作为标识符。
27. 识别 `-` 作为操作符。
28. 识别 `1` 作为数字常量。
29. 识别 `;` 作为分隔符。
30. 识别 `}` 作为分隔符。
31. 识别 `}` 作为分隔符。

通过这种方式，词法分析器将源代码分解为一系列词法单元，为后续的语法分析阶段提供了基础。

## 2.2 语法分析

语法分析是编译过程的第二步，它涉及将词法单元序列转换为抽象语法树（AST）。在FDMJ2025编译器中，我们使用了Yacc/Bison工具来生成语法分析器。Yacc/Bison是一个强大的语法分析器生成工具，它可以根据定义的CFG自动生成解析器代码。通过这种方式，我们可以高效地实现复杂的语法分析规则，并生成高质量的AST。

### 2.2.1 上下文无关文法 (CFG) 概述

上下文无关文法是形式语言理论中的一种文法，它描述了一类语言的语法结构。CFG由一组产生式规则组成，每个产生式规则包含一个非终结符（左侧）和一个或多个终结符和非终结符的序列（右侧）。CFG能够描述的语言比正则文法描述的语言更加复杂，包括大多数编程语言。

在编译原理中，CFG通常用于定义编程语言的语法结构。编译器的语法分析器根据CFG规则解析源代码，构建出抽象语法树（Abstract Syntax Tree, AST）。AST是一种树形结构，它表示源代码的语法结构，是编译器进行语义分析和代码生成的基础。

CFG的一个关键特性是它能够处理递归定义，这使得它能够描述具有递归结构的语言，如算术表达式和程序结构。CFG的另一个重要特性是它能够处理嵌套结构，这对于编程语言中的块结构和函数调用非常重要。

### 2.2.2 FDMJ2025支持的CFG语法规则

FDMJ2025支持的语法规则展示如下：

```

Prog -> MainMethod ClassDeclList
MainMethod -> public int main '(' ')' '{' VarDeclList StmtList '}'
VarDeclList -> ε | VarDecl VarDeclList
VarDecl -> class id id ';' // id <=> [a-z_A-Z][a-z_A-Z0-9]*
    | int id ';' | int id '=' Const ';'
    | int '[' ']' id ';' | int '[' ']' id '=' '{' ConstList '}' ';'
    | int '[' 'NUM ']' id ';' | int '[' 'NUM ']' id '=' '{' ConstList '}' ';' // NUM <=> [1-9][0-9]*|0
Const -> NUM | '-' NUM
ConstList -> ε | Const ConstRest
ConstRest -> ε | ',' Const ConstRest
StmtList -> ε | Stmt StmtList
Stmt -> '{' StmtList '}'
    | if '(' Exp ')' Stmt else Stmt | if '(' Exp ')' Stmt
    | while '(' Exp ')' Stmt | while '(' Exp ')' ';'
    | Exp '=' Exp ';'
    | Exp '[' ']' '=' '{' ExpList '}' ';'
    | Exp '.' id '(' ExpList ')' ';'
    | continue ';' | break ';'
    | return Exp ';'
    | putint '(' Exp ')' ';' | putch '(' Exp ')' ';'
    | putarray '(' Exp ',' Exp ')' ';'
    | starttime '(' ')' ';' | stoptime '(' ')' ';'
Exp -> NUM | true | false | length '(' Exp ')'
    | getInt '(' ')' | getch '(' ')'
    | getarray '(' Exp ')'
    | id | this
    | Exp op Exp | '!' Exp | '-' Exp
    | '(' Exp ')' | '(' '{' StmtList '}' Exp ')'
    | Exp '.' id
    | Exp '.' id '(' ExpList ')'
    | Exp '[' Exp ']'
ExpList -> ε | Exp ExpRest
ExpRest -> ε | ',' Exp ExpRest
ClassDeclList -> ε | ClassDecl ClassDeclList
ClassDecl -> public class id '{' VarDeclList MethodDeclList '}'
    | public class id extends id '{' VarDeclList MethodDeclList '}'
MethodDeclList -> ε | MethodDecl MethodDeclList
MethodDecl -> public Type id '(' FormalList ')' '{' VarDeclList StmtList '}'
Type -> class id | int | int '[' ']'
FormalList -> ε | Type id FormalRest
FormalRest -> ε | ',' Type id FormalRest

```

## 2.2.3 AST节点类的详细说明

### 1. 程序结构类

#### Program 类

- 表示整个程序。
- 包含：
  - MainMethod: 程序的主方法。
  - ClassDecl: 类声明的列表。

#### MainMethod 类

- 表示程序的主方法。
- 包含：
  - VarDecl: 变量声明的列表。
  - Stmt: 语句的列表。

#### ClassDecl 类

- 表示类声明。

- 包含：
  - IdExp：类的标识符。
  - IdExp：扩展类的标识符（用于继承）。
  - VarDecl：类变量的列表。
  - MethodDecl：方法的列表。

## 2. 类型和声明类

### Type 类

- 表示类型信息。
- 包含：
  - TypeKind：类型分类（INT、CLASS、ARRAY）。
  - IdExp：类的标识符（对于 CLASS 类型）。
  - IntExp：数组的维度（对于 ARRAY 类型）。

### VarDecl 类

- 表示变量声明。
- 包含：
  - Type：变量的类型。
  - IdExp：变量的标识符。
  - 初始值（IntExp 或 IntExp 的数组）。

### MethodDecl 类

- 表示方法声明。
- 包含：
  - Type：返回类型。
  - IdExp：方法的标识符。
  - Formal：形式参数的列表。
  - VarDecl：局部变量的列表。
  - Stmt：语句的列表。

### Formal 类

- 表示方法声明中的形式参数。
- 包含：
  - Type：参数的类型。
  - IdExp：参数的标识符。

## 3. 语句类

### Stmt (基类)

- 所有语句类型的基类。

#### 控制流语句：

- **If**：if-else 语句。
- **While**：while 循环语句。
- **Nested**：语句块。
- **Continue**：continue 语句。
- **Break**：break 语句。
- **Return**：return 语句。

#### 赋值和调用语句：

- **Assign**：赋值语句。
- **CallStmt**：方法调用语句。

#### 输入输出语句：

- **PutInt**：打印整数语句。
- **PutCh**：打印字符语句。
- **PutArray**：打印数组语句。
- **GetInt**：读取整数语句。
- **GetCh**：读取字符语句。
- **GetArray**：读取数组语句。

#### 计时语句：

- **Starttime:** 开始计时语句。
- **StopTime:** 停止计时语句。

#### 4. 表达式类

##### Exp (基类)

- 所有表达式类型的基类。

##### 运算符:

- **BinaryOp:** 二元运算表达式。
- **UnaryOp:** 一元运算表达式。
- **OpExp:** 运算符的表示。

##### 访问表达式:

- **ArrayExp:** 数组访问表达式。
- **ClassVar:** 类变量访问表达式。
- **CallExp:** 方法调用表达式。
- **Length:** 数组长度表达式。

##### 字面量表达式:

- **BoolExp:** 布尔字面量。
- **IntExp:** 整数字面量。
- **IdExp:** 标识符表达式。

##### 特殊表达式:

- **This:** `this` 引用。
- **Esc:** 带有语句块的表达式。

## 2.2.4 Bison的语法规则

### 1. 语法规则的定义

语法规则在 Bison 文件中定义，每条规则对应一个语法规则，规则的右侧是产生式，左侧是规则的名称。例如：

```
PROG: MAINMETHOD CLASSDECLLIST
{}
```

- **左侧:** `PROG` 是语法规则的名称，表示程序的根节点。
- **右侧:** `MAINMETHOD CLASSDECLLIST` 是产生式，表示程序由主方法和类声明列表组成。
- **动作代码:** `{...}` 中的代码用于创建对应的 AST 节点，并将其存储到 `result` 中。

### 2. 动作代码

动作代码是语法规则的核心部分，用于创建 AST 节点。例如：

```
MAINMETHOD: PUBLIC INT MAIN '(' ')' '{' VARDECLLIST STMLIST '}'
{
    $$ = new MainMethod(p, $7, $8) ;
}
```

- `$1` 到 `$n` 表示产生式中各个符号对应的语义值。
- `$$` 表示当前规则的语义值。
- `new MainMethod(p, $7, $8)` 创建了一个 `MainMethod` 类型的 AST 节点，并将其赋值给 `$$`。

## 2.2.5 符号优先级和结合性

为了避免语法冲突，在语法分析之前设计以下操作符的优先级和结合性规则：

```
%right '='
%left OR
%left AND
%left EQ NE
%left LT LE GT GE
%left ADD MINUS
%left TIMES DIVIDE
%right NOT UMINUS
%right ARRAY_ACCESS
%right METHOD_CALL
%left '.' '(' '[' ']'
%precedence THEN
%precedence ELSE
```

详细说明如下：

- **最高优先级：括号和成员访问**

示例：`a.b(c[d]);`

`a.b` 是成员访问，`c[d]` 是数组访问，`()` 是方法调用。这些操作符的优先级高于其他运算符。

- **方法调用和数组访问**

示例：`a.b().c[d];`

`a.b()` 是方法调用，`c[d]` 是数组访问。右结合性确保先解析 `b()`，再解析 `c[d]`。

- **算术运算符**：乘除法高于加减法

- **比较运算符**：优先级低于算术运算符

示例：`a + b < c * d;`

先计算 `a + b` 和 `c * d`，然后进行比较。

- **逻辑运算符**：优先级低于比较运算符

示例：`a < b && c > d || e == f;`

先计算比较表达式 `a < b`、`c > d` 和 `e == f`，然后进行逻辑与和逻辑或。

- **赋值运算符**：优先级最低，并且是右结合的

示例：`a = b = c;`

右结合性确保先计算 `b = c`，然后将结果赋值给 `a`。

- **单目运算符**：逻辑非和负号，具有右结合性

示例：`!a + -b;`

先计算 `!a` 和 `-b`，然后进行加法。

- **特殊优先级**

```
%precedence THEN
%precedence ELSE
```

- **THEN** 和 **ELSE** 是用于处理 `if` 和 `else` 语句的特殊优先级，以避免悬垂 `else` 问题。

- **示例：**

```
if (a) if (b) c; else d;
```

`ELSE` 的优先级确保 `else` 与最近的 `if` 结合。

## 2.2.6 错误处理

为了提供准确的错误报告和调试信息，我们跟踪每个词法单元在源代码中的位置。我们使用 `ast_location` 类来跟踪当前行号和列号。每次遇到换行符时，行号递增，列号重置。

语法分析器在遇到错误时会调用 `ASTParser` 类的 `error` 函数。例如：

```
void ASTParser::error(const location_t &location, const std::string &message)
{
    std::cerr << "Error at lines " << location << ":" << message << std::endl;
}
```

其中，location 提供了错误发生的位置信息。message 是错误描述。

一个报错输出示例(漏输一个分号):

```
Error at lines [3:3-3:5]: syntax error, unexpected INT, expecting ';'
```

## 2.2.7 示例

考虑如下的输入代码:

```
public int main() {
    int x = 10;
    return x;
}
```

1. 首先，词法分析器将输入代码转换为词法单元序列:

```
[PUBLIC, INT, MAIN, '(', ')', '{', INT, ID('x'), '=', NONNEGATIVEINT(10), ';', RETURN, ID('x'), ';', '}]
```

2. 语法分析器根据语法规则逐步构建 AST:

- 匹配规则 PROG :

```
PROG: MAINMETHOD CLASSDECLLIST
```

创建 Program 节点。

- 匹配规则 MAINMETHOD :

```
MAINMETHOD: PUBLIC INT MAIN '(' ')' '{' VARDECLLIST STMLIST '}'
```

创建 MainMethod 节点。

- 匹配规则 VARDECL :

```
VARDECLLIST: // empty
| VARDECL VARDECLLIST
VARDECL: INT ID '=' CONST ';'
```

创建 VarDecl 节点，以及在AST\_YYSTYPE中声明的varDeclList结构。

- 同理匹配规则 STM :

```
STMLIST: // empty
|
STM STMLIST
STM: RETURN EXP ';'
```

创建 Return 节点表示，返回语句。

3. 最终生成的 AST:

```
Program
└── MainMethod
    ├── vector<VarDecl> — VarDecl: int x = 10
    └── vector<Stm> — Return: x
    └── vector<ClassDecl> — (空)
```

总的来说，解析阶段是编译器设计中的基础，它为后续的语义分析、优化和代码生成等阶段提供了必要的输入。通过精心设计的词法分析器和语法分析器，我们可以确保编译器能够正确地理解和处理源代码，从而生成高效、可靠的目标代码。

## 三、语义分析

语义分析的主要目标是确保程序的逻辑和类型正确性，它主要需要两个步骤：

- 符号表构建：构建符号表，用于存储变量、方法和类的声明信息，以便后续的类型检查和代码生成。
- 类型检查：确保每个表达式和变量的类型符合语言的语义规则。在发现类型错误时，提供清晰的错误信息，并指出错误的位置。

### 3.1 访问者模式

在此之前，我们需要先了解访问者模式的原理以及其优势，便于理解后续的代码。

在语义分析阶段，访问者模式被用于构建符号表和执行类型检查（见 `AST_Name_Map_Visitor` 和 `AST_Semant_Visitor`）。通过定义不同的访问者类，可以轻松地为语法树的每个节点添加新的语义分析操作，而无需修改节点类的代码。

#### 3.1.1 访问者模式概述

访问者模式是一种行为设计模式，用于分离对象的结构和操作。它允许在不修改对象结构的情况下，为对象结构中的元素添加新的操作。访问者模式的核心思想是将数据结构和作用于结构上的操作解耦，使得操作可以在不改变数据结构的前提下独立变化。

访问者模式的主要组成部分包括：

- **访问者 (Visitor)**：定义了对数据结构中每个元素的访问操作。
- **具体访问者 (ConcreteVisitor)**：实现了访问者接口，为每种元素提供具体的访问操作。
- **元素 (Element)**：定义了接受访问者的接口。
- **具体元素 (ConcreteElement)**：实现了元素接口，接受访问者并调用访问者的访问方法。
- **对象结构 (ObjectStructure)**：包含元素的集合，可以遍历这些元素并接受访问者的访问。

#### 3.1.2 访问者模式的优势

1. **分离数据结构和操作**：访问者模式将数据结构和操作分离，使得操作可以在不改变数据结构的前提下独立变化。这符合开闭原则（对扩展开放，对修改封闭）。
2. **易于添加新操作**：通过添加新的访问者类，可以轻松地为现有数据结构添加新的操作，而无需修改数据结构本身。
3. **集中相关操作**：访问者模式将相关操作集中在一个访问者类中，使得代码更加模块化，便于维护和扩展。
4. **支持双分派 (Double Dispatch)**：访问者模式可以通过双分派机制，根据对象的类型动态选择合适的操作。这在多态和动态绑定中非常有用。

#### 3.1.3 访问者模式的实现(以`AST_Name_Map_Visitor`为例)

##### 1. 定义访问者接口

访问者接口定义了对每种元素的访问操作。

```
class AST_Visitor {
public:
    virtual void visit(Program* node) = 0;
    // ...
};
```

##### 2. 定义具体访问者

具体访问者实现了访问者接口，为每种元素提供具体的访问操作。

```

class AST_Name_Map_Visitor : public AST_Visitor {
private:
    Name_Maps *name_maps;
    string current_class;
    string current_method;
public:
    AST_Name_Map_Visitor() {
        name_maps = new Name_Maps();
    }
    Name_Maps* getNameMaps() { return name_maps; }
    void visit(Program* node) override {
        // 具体实现
    }
    void visit(MainMethod* node) override {
        // 具体实现
    }
    // 其他访问方法的实现...
};

```

### 3. 定义元素接口

元素接口定义了接受访问者的接口。

```

class AST {
public:
    virtual void accept(AST_Visitor &visitor) = 0;
};

```

### 4. 定义具体元素

具体元素实现了元素接口，接受访问者并调用访问者的访问方法。

```

class Program : public AST {
public:
    void accept(AST_Visitor &visitor) override {
        visitor.visit(this);
    }
};

class MainMethod : public AST {
public:
    void accept(AST_Visitor &visitor) override {
        visitor.visit(this);
    }
};
// 其他具体元素的实现...

```

### 5. 使用访问者模式

在应用中，通过创建具体访问者对象并调用其方法来对对象结构进行操作。

```

int main() {
    Program* program = new Program();
    AST_Name_Map_Visitor visitor;
    program->accept(visitor);
    return 0;
}

```

## 3.2 符号表构建

符号表用于存储程序中所有变量、方法和类的声明信息。在本实现中，符号表通过 `Name_Maps` 类构建，该类用于维护程序中所有名称及其关系的全面映射。它的构成成分如下：

**关键组成部分：**

- **classes:** 所有类名的集合
- **classHierarchy:** 类继承关系的映射
- **methods:** 类-方法名对的集合
- **classVar:** 类变量及其声明的映射
- **methodVar:** 方法变量及其声明的映射
- **methodFormal:** 方法形式参数及其声明的映射
- **methodFormalList:** 方法形式参数列表的映射

**关键方法：**

• **类管理：**

- `is_class()`：检查类是否存在
- `add_class()`：添加新类
- `add_class_hierarchy()`：添加类继承关系
- `get_ancestors()`：检索祖先类

• **方法管理：**

- `is_method()`：检查方法是否存在
- `add_method()`：添加新方法

• **变量管理：**

- `is_class_var()`：检查类变量是否存在
- `add_class_var()`：添加类变量
- `get_class_var()`：检索类变量声明
- `is_method_var()`：检查方法变量是否存在
- `add_method_var()`：添加方法变量
- `get_method_var()`：检索方法变量声明

• **形式参数管理：**

- `is_method_formal()`：检查形式参数是否存在
- `add_method_formal()`：添加形式参数
- `get_method_formal()`：检索形式参数声明
- `add_method_formal_list()`：添加形式参数列表
- `get_method_formal_list()`：检索形式参数列表

• **工具：**

- `print()`：打印所有名称映射

## 3.3 语义上下文记录

实验中使用 `AST_Semant` 类表示抽象语法树（AST）节点的语义信息，这对于类型检查和中间表示（IR）生成至关重要。它的构成成分如下：

**关键组成部分：**

- **Kind 枚举：** 定义语义实体的类型（值、方法名、类名）。
- **TypeKind：** 存储类型信息（类/对象、整数、数组）。
- **type\_par：** 变体，可以存储额外的类型参数（类名或数组维度）。
- **Ivalue：** 表示表达式是否是左值。

**方法：**

- 构造函数：初始化所有语义属性。

- `get_kind()`：返回语义实体的类型。
- `get_type()`：返回类型种类。
- `get_type_par()`：返回类型参数。
- `is_lvalue()`：检查表达式是否是左值。
- `s_kind_string()`：将 Kind 枚举转换为字符串表示。

此外，一个全局 `AST_Semant_Map` 将 AST 节点映射到它们对应的语义信息（`AST_Semant` 对象），用于维护在整个编译过程中的语义上下文。它包括：

- `semant_map`：存储 AST 节点与 `AST_Semant` 关联的映射结构。
- `name_maps`：名称映射信息的引用。

`AST_Semant_Visitor` 类实现用于 AST 的语义分析的访问者模式。它遍历 AST，执行语义检查，并为所有子表达式维护语义信息。它包括：

- `semant_map`：存储所有子表达式的语义信息。
- `name_maps`：程序中所有名称的映射。
- `current_class/method`：跟踪当前上下文。
- `in_a_while_loop`：用于处理嵌套 `while` 循环的计数器。

## 方法：

- 构造函数：使用名称映射初始化。
- `getSemantMap()`：返回语义映射。
- `visit()` 方法：为每种 AST 节点类型覆盖以执行特定的语义分析。

## 3.4 FDMJ类型说明

### 3.4.1 数据类型

- **基本类型**（整数）
- **数组类型**（整数数组）
- **类类型**（也称为对象类型，具有类名）
  - **类变量**
    - **类级变量**
      - 在 Java 中使用 `static` 关键字（例如 `class C { static int a; ... }`）
      - **FDMJ 没有这些**
    - **对象（实例）变量**
      - 在 Java 中更像一个记录（例如 `class C { int a; }` ）
      - **FDMJ 只有公有对象变量**
  - **方法**
    - Java 方法也具有类级别或对象级别（静态方法；非静态方法）
    - **FDMJ 只有非静态公有方法**
  - **继承**
    - Java
      - 子类继承（超）类的变量和方法。这种继承关系是传递的。
      - 可以覆盖（公有）变量和方法
        - 变量：通过同名覆盖。
        - 方法：通过相同的名称和签名（多态性）覆盖。
    - **FDMJ**
      - 子类继承父类的变量，但不能重复声明同名变量
      - 子类继承父类的方法，可以重写方法的实现，但方法的签名（返回值和参数列表）不能改变。例如父类是 `int f(int a, int[] b)`，子类只能是 `int f(int, int[])` 的签名，参数名可以不一样但函数名、类型、参数类型顺序必须一样。
      - 只能单继承，不允许多继承，不允许成环

- FDMJ 语法中，只有一个 `extend`，自然只有单继承、不允许许多继承
- FDMJ 加入如下限制：若 `a extends b`，则 `b` 必须事先定义。因此保证了继承关系的方向性（父类先于子类存在），而且单继承形成的是树结构，没有回溯的可能性，因此不会存在循环继承的问题

### 3.4.2 值类型

- **基本值**（缩写为“值”）
  - 只有 `int` 的变量或常量才有值
  - 变量如：`int a;`
  - 常量如：`1`
- **指针值**（缩写为“指针”）
  - 只有数组或类的对象或匿名对象才有指针
  - 对象如：`int[] a;`（数组对象），`class C;`（类对象）
  - 匿名对象如：`{1, 2, 3}`（匿名数组对象）
- **位置值**（所有数据类型，缩写为“位置”）
  - 对于内存位置（无论是在栈中还是在堆空间中）。
  - 每个位置都专用于特定的数据类型，并持有32位（对于本课程）。
  - 只有左值（变量或对象）有位置，右值（常量或匿名对象）没有位置
- **方法值**
  - 表示函数入口位置（指令空间）
  - 注意我们没有在 FDMJ 中使用 `goto` 语句。

### 3.4.3 名称类型

1. **变量名：**
  - 与3种数据类型中的1种相关联
  - 携带2个值：一个位置值和一个基本/指针值（如果已定义）
  - 变量名代表以下之一：（例如变量名是 `a`）
    - 基本类型具有位置值和整数值（例如 `int a;`）
    - 数组类型具有位置值和指针值（例如 `int[] a;`）
    - 类类型具有位置值和指针值（例如 `class A a;`）
  - 变量仅在其声明的作用域内
    - 在方法中，或（包括传入的和声明的；FDMJ 传值（整数/指针），不传引用（位置））
    - 在类中
2. **类名：**
  - 是类类型的别名
  - 不与任何值相关联
  - 类名用于定义类类型的变量（例如“`class A o;`”）
3. **方法名：**
  - 是一个指针值，这是“函数”（或方法）的“地址”。没有关联位置值。
  - 方法的确切“地址”
    - 取决于调用它的对象（也基于继承层次结构）。
    - 或在链接时确定（如果是外部函数）。

### 3.4.4 类型转换

FDMJ 允许在赋值期间进行隐式基本类型转换和类类型向上转型

- **隐式基本类型转换**
  - 例如 `int a = true + 2;` 最终 `a` 被赋值为 `3`
- **类类型upcasting**
  - 向上转型相当于放弃子类的一些特性，转而成为父类类型
  - 例如，允许类赋值操作：`class Father f; class Son s; f = s;`

- `f` 的位置和值都是 `Father` 类型，因此，之后 `f` 只能调用父类中存在的变量和方法；但如果调用了子类重写的方法，则会调用子类的实现
- 在类型检查中，我们不关心它调用了什么函数，只检查右边是否是左边的类型或者是左边类型的子类型

## 3.5 语义分析与类型检查

为了在处理过程中保持一致性，我们做出**两个重要的前提假设**：

- `main` 函数视为：在 `_^main^_` 类的方法
- 函数返回值视为：名称为 `_^return^_+$current_method` 的方法参数

在此基础上，我们在执行符号构建和语义分析的过程中，做出以下16种关键的类型检查，并给出对应的报错信息以及位置信息：

### 1. 子类不允许声明和父类一样的变量

- **检查思路**：当访问到子类的变量声明时，检查该变量是否已经在父类中声明。通过查询父类和子类中的变量名来检测重复声明。
- **错误信息**：`Error: Duplicate class variable with ancestors: " << var_name << " in class " << current_class`

### 2. 检查子类重载父类方法，是否违反签名（参数列表类型，返回值类型）

- **检查思路**：当访问到子类的方法声明时，检查该方法是否与父类中的方法签名一致。使用符号表来存储方法签名信息，通过比较子类和父类中的方法签名来检测签名冲突。
- **错误信息**：
  - `Error: Method formal number mismatch`
  - `Error: Method return type mismatch`
  - `Error: Method formal type mismatch`

### 3. If/While语句内类型为INT

- **检查思路**：当访问到条件表达式时，检查其类型是否为INT。
- **错误信息**：`Error: Condition in if/while loop must be of type int`

### 4. Assign语句中检查

- **检查思路**：
  - 左值是否为左值（lvalue）。
  - 左右值类型是否相等。
  - 如果是类类型，是否满足upcast。
- **错误信息**：
  - 左值错误：`Error: Left-hand side of assignment is not an lvalue`
  - 类型不匹配：`Error: Type mismatch in assignment`
  - 类型upcast错误：`Error: Inheritance relationship violated in assignment`

### 5. Assign语句中数组赋值

- **检查思路**：在符号表中查找左值数组，并更新其长度。
- **错误信息**：`Error: Array Assignment: left value is not an id`

### 6. CallStmt/CallExp语句检查

- **检查思路**：
  - 点号前是否是类的实例对象。
  - 根据继承关系查找方法是否存在。
  - 参数列表的类型是否按顺序匹配。
- **错误信息**：
  - 对象错误：`Error: Object is not a class instance`
  - 参数未定义：`Error: Parameter not defined in CallStm!`

- 参数不匹配: Error: Parameter type mismatch in method call

## 7. Continue/Break检查是否在while循环中

- 检查思路: 使用一个标志变量 in\_a\_while\_loop 来跟踪是否在 while 循环中。当访问到 Continue 或 Break 语句时，检查它们是否位于 while 循环中。
- 错误信息: Error: Continue/Break statement not within a while loop

## 8. Return语句检查

- 检查思路:
  - 返回值类型与声明的是否匹配。
  - 类返回值是否满足upcast。
- 错误信息:
  - 类型不匹配: Error: Return type mismatch
  - 类型upcast错误: Error: Inheritance relationship violated in return statement

## 9. PutInt/PutCh/PutArray检查

- 检查思路:
  - PutInt/PutCh参数是否为int。
  - PutArray中n为int类型，arr为array类型。
- 错误信息:
  - 参数错误: Error: PutInt/PutCh expression must be an integer
  - 数组错误: Error: PutArray size and array type mismatch

## 10. BinaryOp检查

- 检查思路:
  - 左右值类型是否相等。
  - 不允许类进行二元运算。
- 错误信息: Error: Type mismatch in binary operation, Error: class type is not allowed in binaryop operation

## 11. UnaryOp检查操作数只能是INT

- 检查思路: 当访问到一元运算符时，检查其操作数是否为INT类型或数组类型
- 错误信息: Error: class type is not allowed in unary operation

## 12. ArrayExp检查

- 检查思路:
  - arr是数组类型。
  - index是int类型。
- 错误信息: Error: ArrayExp base is not an array., Error: Array index is not an integer.

## 13. ClassVar检查

- 检查思路:
  - 点号前是否是类的实例对象。
  - 点号后是否是类或其祖先的成员。
- 错误信息:
 

Error: ClassVar object is not a class instance.,  
Error: Class variable '" <> node->id->id <> "' not found in class " <> obj\_class\_name

## 14. Length检查参数只能是数组类型

- 检查思路: 当访问到 Length 函数时，检查其参数是否为数组类型。
- 错误信息: Error: Length expression must be an array

## 15. GetArray检查参数只能是数组类型

- **检查思路:** 当访问到 GetArray 函数时, 检查其参数是否为数组类型。
- **错误信息:** Error: GetArray parameter must be an array

## 16. BoolExp直接标记为INT类型

- **检查思路:** 当访问到 BoolExp 时, 将其标记为整数类型, 相当于进行了隐式转换。
- **错误信息:** Error: Boolean expression implicitly converted to int

## 3.6 示例

考虑以下的源程序:

```
public int main() {  
    int[] a={1,2,3,4,5,6,7,8,9,10};  
    int[] aa={3, 4, 5};  
  
    if (a[aa[0]]<1)  
        aa=a;  
    return aa[9>10];  
}
```

生成的带有语义信息的ast树为:

```

<?xml version="1.0" encoding="UTF-8"?>
<Program>
    <MainMethod>
        <VarDeclList>
            <VarDecl>
                <Type typeKind="ARRAY">
                    <Arity val="0"/>
                </Type>
                <IdExp id="a"/>
                <IntInitList>
                    <IntExp val="1"/>
                    <IntExp val="2"/>
                    <IntExp val="3"/>
                    <IntExp val="4"/>
                    <IntExp val="5"/>
                    <IntExp val="6"/>
                    <IntExp val="7"/>
                    <IntExp val="8"/>
                    <IntExp val="9"/>
                    <IntExp val="10"/>
                </IntInitList>
            </VarDecl>
            <VarDecl>
                <Type typeKind="ARRAY">
                    <Arity val="0"/>
                </Type>
                <IdExp id="aa"/>
                <IntInitList>
                    <IntExp val="3"/>
                    <IntExp val="4"/>
                    <IntExp val="5"/>
                </IntInitList>
            </VarDecl>
        </VarDeclList>
        <StmList>
            <If>
                <BinaryOp s_kind="Value" typeKind="INT" lvalue="false">
                    <ArrayExp s_kind="Value" typeKind="INT" lvalue="true">
                        <IdExp s_kind="Value" typeKind="INTARRAY" lvalue="true" arity="0" id="a"/>
                        <ArrayExp s_kind="Value" typeKind="INT" lvalue="true">
                            <IdExp s_kind="Value" typeKind="INTARRAY" lvalue="true" arity="0" id="aa"/>
                            <IntExp s_kind="Value" typeKind="INT" lvalue="false" val="0"/>
                        </ArrayExp>
                    </ArrayExp>
                    <OpExp op="&lt;;" />
                    <IntExp s_kind="Value" typeKind="INT" lvalue="false" val="1"/>
                </BinaryOp>
                <Assign>
                    <IdExp s_kind="Value" typeKind="INTARRAY" lvalue="true" arity="0" id="aa"/>
                    <IdExp s_kind="Value" typeKind="INTARRAY" lvalue="true" arity="0" id="a"/>
                </Assign>
            </If>
            <Return>
                <ArrayExp s_kind="Value" typeKind="INT" lvalue="true">
                    <IdExp s_kind="Value" typeKind="INTARRAY" lvalue="true" arity="0" id="aa"/>
                    <BinaryOp s_kind="Value" typeKind="INT" lvalue="false">
                        <IntExp s_kind="Value" typeKind="INT" lvalue="false" val="9"/>
                        <OpExp op="&gt;" />
                        <IntExp s_kind="Value" typeKind="INT" lvalue="false" val="10"/>
                    </BinaryOp>
                </ArrayExp>
            </Return>
        </StmList>
    </MainMethod>
</Program>

```

```

        </BinaryOp>
    </ArrayExp>
</Return>
</StmList>
</MainMethod>
<ClassDeclList/>
</Program>

```

**以 return aa[9>10]为例分析：**代码首先访问return节点，进而访问其表达式arrayexp，这时需要首先分别对数组名idexp和整型索引进行访问，在索引中继续递归访问二元运算符的左右值和符号，对两个整型操作数附上对应的语义信息前，需要先检测两者类型是否一样，进而赋值给binaryop语义信息，并在arrayexp中检查索引是否为整型，并完成arrayexp的语义赋值，其中需要注意Idexp和ArrayExp本身是左值，而整数常量，计算表达式都不能为左值。

## 四、转换为中间表示 (IR)

在编译过程中，中间表示 (IR) 是一种介于源代码和目标代码之间的程序表示形式。它提供了一种与平台无关的方式来描述程序的语义，同时去除了源代码中与特定硬件无关的细节。IR+是IR的一种扩展形式，它增加了一些特定于编译器后端优化和目标代码生成的特性。

### 4.1 关键数据结构

#### 4.1.1 ASTToTreeVisitor 类成员变量

```

tree::Tree *visit_tree_result = nullptr;           // 用于返回stm节点
Tr_Exp* visit_exp_result = nullptr;               // 用于返回exp节点
Temp_map* tm = nullptr;                          // 用于生成临时变量和标签
Class_table* class_table = nullptr;                // 类表
Method_var_table* mvt = nullptr;                  // 方法变量表
AST_Semant_Map* semant_map = nullptr;             // 语义信息映射map
Name_Maps* nm = nullptr;                         // 名称信息映射map
string current_class;                            // 记录当前类
string current_method;                           // 记录当前方法
stack<tree::Label*> loop_entry_labels;          // 循环入口标签栈
stack<tree::Label*> loop_exit_labels;            // 循环退出标签栈
tree::Exp* last_tmp = nullptr;                    // 记录最后一个构造的temp

```

#### 4.1.2 类表 (Class Table)

类表是编译器中用于映射每个类变量和方法到地址偏移的数据结构。它采用了一种“通用类”方法，即所有类使用相同的类表，其中列出了所有类的所有可能的变量和方法。

- 类表使用两个映射map来存储变量和方法的名称及其对应的偏移量。这种设计使得编译器能够快速查找变量和方法的偏移信息。
- 提供 `get_var_pos` 和 `get_method_pos` 方法，用于获取变量和方法的偏移量。这些方法通过变量或方法的名称作为键，快速返回对应的偏移量。
- `get_malloc_size` 方法返回类表中方法的总偏移量，用于确定内存分配的大小。

**生成类表 (generate\_class\_table)**：生成类表的过程涉及遍历所有类和方法，为每个类变量和方法分配一个唯一的偏移量，并存储在类表中。这个过程确保了每个类变量和方法都有一个唯一的地址偏移，从而在运行时可以正确地访问它们。

#### 4.1.3 方法变量表 (Method Variable Table)

方法变量表用于存储方法中的形式参数和局部变量的信息，包括它们的临时标识符和类型。

- 方法变量表使用两个映射（`map`）来存储变量的临时标识符和类型。这种设计使得编译器能够快速查找变量的类型和临时标识符。
- 提供 `get_var_temp` 和 `get_var_type` 方法，用于获取变量的临时标识符和类型。这些方法通过变量的名称作为键，快速返回对应的临时标识符和类型。
- 在方法变量表中，局部变量可以覆盖同名的形式参数。这种设计确保了方法内部变量的正确处理。
- 方法的返回值也被视为一个特殊的形式参数，并存储在方法变量表中。这种设计确保了方法返回值的正确处理。

**生成方法变量表 (generate\_method\_var\_table)**：生成方法变量表的过程涉及为每个方法的形式参数和局部变量分配临时标识符和类型，并存储在方法变量表中。如果局部变量的名称与形式参数冲突，则使用局部变量。这个过程确保了方法内部变量的正确处理。

#### 4.1.4 接口函数实现

```
tree::Program* ast2tree(fdmj::Program* prog, AST_Semant_Map* semant_map) {
    ASTToTreeVisitor visitor; // 构建visitor实例
    visitor.semant_map = semant_map; // 存储semant_map
    visitor.nm = semant_map->getNameMaps(); // 获得namemap
    prog->accept(visitor); // 访问所有ast节点并构造tree
    return dynamic_cast<tree::Program*>(visitor.getTree()); // 将生成的树根节点返回
}
```

## 4.2 TREE节点类的详细说明

### 程序结构类

- tree::Program**  
程序的顶级容器，存储所有函数声明（`FuncDecl`）的集合。
- tree::FuncDecl**  
表示函数声明，包含函数名、参数列表、基本块集合、返回类型等元信息。
- tree::Block**  
基本块，包含入口标签、出口标签列表和语句序列，用于控制流的线性区域划分。

### 语句类（`Stm` 派生类）

- tree::Seq**  
语句序列，将多个语句按顺序组合成单一语句节点。
- tree::LabelStm**  
标签语句，标记程序中的跳转目标位置（如循环开始/结束位置，条件判断跳转位置）。
- tree::Jump**  
无条件跳转，直接跳转到指定标签（类似 `goto`）。
- tree::Cjump**  
条件跳转，根据二元比较结果跳转到真/假分支标签。
- tree::Move**  
赋值操作，将右侧表达式结果存储到左侧目标位置（hw4中默认是变量）。
- tree::Phi**  
用于不同控制流路径下的变量值合并。
- tree::ExpStm**  
表达式语句，用来忽略 `tree::Exp` 的返回值，仅保留副作用，让 `tree::Exp` 变成 `tree::Stm`。
- tree::Return**  
函数返回语句，携带返回值表达式。

### 表达式类（`Exp` 派生类）

- tree::Binop**  
二元运算类，支持算术、逻辑和比较操作。

**2. tree::Mem**

内存访问操作，表示对指针解引用。

**3. tree::TempExp**

用于转换id (tree::Temp) 为 tree::Exp。

**4. tree::Eseq**

先执行语句块，保留其副作用，再返回表达式结果。

**5. tree::Name**

将标签转换为地址表达式，用于函数指针或跳转表。

**6. tree::Const**

表示整型常量值，用于转换num (int) 为 tree::Exp。

**7. tree::Call**

用于方法调用，包含目标函数名、对象指针和参数列表。

**8. tree::ExtCall**

用于外部函数调用，如 `getint`, `putint`, `starttime` 等。

## 4.3 运算与数组类型的转换

### 4.3.1 运算

运算结果属于表达式，用 `visit_exp_result (Tr_ex*)` 传递结果。

首先，分别计算左右操作数的表达式。然后，根据运算类型进行不同处理：

#### 算数运算（加、减、乘、除、负号）

- 输入 Tr\_ex 类、输出 Tr\_ex 类。
- 根据运算符生成对应的 tree::Binop 表达式
- 对于负号，看作0-exp。

#### 比较运算（等于、不等于、大于、大于等于、小于、小于等于）

- 输入 Tr\_ex 类、输出 Tr\_cx 类。
- 创建两个 Patch\_list，分别作为 true\_list 和 false\_list，待后续填充。
- 根据比较运算符生成条件跳转语句 (tree::Cjump)，存储为 Tr\_cx 节点

#### 逻辑运算（与、或、非）

- 输入 Tr\_cx 类、输出 Tr\_cx 类。
- 准备右值的 right\_label 用于短路时跳转。
- 对于逻辑与 (&&)，需短路计算：左假则整个假，左真则取决于右（跳转至 right\_label）
- 对于逻辑或 (||)，需短路计算：左真则整个真，左假则取决于右（跳转至 right\_label）
- 对于逻辑非 (!)，看作 exp==0，生成条件跳转语句 (tree::Cjump)，存储为 Tr\_cx 节点

#### 赋值

- Assign节点：**

- 计算左右侧表达式的值。
- 将两侧的 Tr\_ex\* 节点调用 unEx 函数转换为 Tr\_ex\* 并取出 exp。其中，Tr\_cx\* 在转换过程中进行 patch\_list 的填充。
- 构建 tree::Move 节点

- VarDecl节点：**

- 左侧从 Method\_var\_table 中拿到变量的 temp 并构建成 tree::TempExp 表达式
- 右侧表达式转换为 Tr\_ex\* 并取出 exp
- 构建 tree::Move 节点

## 4.3.2 程序结构相关

### 条件 (if)

- 生成条件判断的true分支标签和false分支标签以及最终合并标签
- 将条件表达式的patch\_list填充对应的label。
- 控制流：计算条件表达式的值。根据条件表达式的布尔值，跳转到true分支标签或false分支标签，执行完对应分支，跳转或直接进入合并标签。

## 循环 (while、break、continue)

- **while节点：**
  - 生成循环的条件判断标签、入口标签和出口标签
  - 入口标签和出口标签存入栈中
  - 处理完循环体内语句之后，从栈顶弹出入口标签和出口标签
  - 控制流：在循环体的开始处，计算条件表达式的值。根据条件表达式的布尔值，跳转到循环体入口标签或出口标签。
- **break节点：**
  - 无条件跳转至栈顶的循环出口标签
- **continue节点：**
  - 无条件跳转至栈顶的循环入口标签

## 外部函数调用

- **PutInt, PutCh节点**
  - 存在参数，先访问参数
  - 提取参数表达式中的exp，构建 tree::ExtCall 节点并用 Tr\_ex\* 包裹。
  - 再用 tree::ExpStm 包裹，转成stm类型。
- **Starttime、StopTime、GetInt、GetCh节点**
  - 不存在参数，直接构建 tree::ExtCall 节点并用 Tr\_ex\* 包裹。
  - 其中，Starttime、StopTime是stm类，需再次用 tree::ExpStm 包裹。

## 4.3.3 数组操作

- **数组初始化**  
对于形如 `int[] a={ConstList}` 的语句，调用 `std::new` 分配比数组长度多1的空间，每个位置的类型为 `int`，在第一个位置放置数组长度，然后通过计算偏移并通过使用 `tree::Mem` 为其他位置赋值。
- **数组存取 ( exp[exp] )**  
计算偏移，然后从第一个位置读取数组长度，判断索引值是否小于长度，否则数组越界将调用exit退出程序，如果长度检查通过，利用 `tree::Mem` 取对应位置的内容进行读取或赋值
- **数组长度 ( length(exp) )**  
通过 `tree::Mem` 取第一个位置位置获得长度
- **数组运算**
  - 二元运算：首先获得两个数组的长度，检查长度是否相同，不同则exit程序，相同则继续分配新的数组空间，大小为 `(size+1)*int_length`，并在第一个位置存入长度，接着循环遍历每一个索引值，计算两个数组对应位置数值的结果存入新数组对应位置。
  - 一元运算：不需要检查数组长度，其余同二元运算。

## 4.4 关于类类型的转换

### 4.4.1 关键处理

1. **重命名method:** class\_name + "^" + method\_name
2. **参数列表:** main method的参数列表为空，class method的参数列表第一个元素为class指针 (This)，其后为函数的其他形式参数。
3. **处理This:** 在构造method\_var\_table时，首先构造This变量作为temp=100(main函数除外)，当运行到this有关的语句时，访问this节点，从method\_var\_table中获得其对应的变量和类型。
4. **记录class的变量和方法:** 依次遍历namemap中所有的变量和方法，将不同名的变量和方法分别存入对应的map中，利用一个全局的offset记录偏移量。将main函数记录在最后。

5. **处理多态**: 在寻找类变量和方法的时候, 首先从namemap中找到其所有祖先的vector, 并将自身加入vector的第一个元素, 按照vector的顺序 (从子类到父类) 遍历查找该变量或方法, 找到第一个为止。

## 4.4.2 类操作转换

- **类实例初始化**:

- 首先分配class\_table长度的空间, 用于存放类变量和方法的指针。然后从子类向上查找, 得到所有该类的变量和方法, 注意函数重载只保留最先遇到的那个。
- 先遍历成员变量, 如果遇到需初始化的整型或数组, 则递归调用VarDecl的访问函数, 进行构造
- 遍历方法: 将重命名的方法名存入对应的地址 (从class\_table的偏移量中获得)

- **类变量访问 ( ClassVar )**

- 首先访问类实例, 得到其地址, 然后从class\_table中获得类变量的偏移量, 计算得到存储该变量的地址
- 从语义信息表中获得类实例的语义信息, 从而得到其所属的类, 再在该类及其祖先中遍历, 查找这个变量的类型
- 最后, 使用tree::Mem从该地址取出变量

- **类方法调用 ( CallExp 、 CallStm )**

- 首先访问类实例, 得到其地址, 然后从class\_table中获得函数方法的偏移量, 计算得到存储方法名的地址
- 从语义信息表中获得类实例的语义信息, 从而得到其所属的类, 再在该类及其祖先中遍历, 查找这个方法的信息, 获得方法的返回值
- 最后构造tree::Call树节点进行方法调用。

## 4.5 示例

考虑下面的源代码:

```
public int main() {
    int[] w;
    class D d;
    return(1);
}

public class C {
    int x=1;
    int[] a;
    class D y;
}

public class D extends C {
    int[] z = {1,2,3};
}
```

使用通用类记录 (Unified Object Record) 规则生成类表如下:

Class Table Contents:	
Variable Name	Offset
a	0
x	4
y	8
z	12

方法变量表如下:

```
(_^main^_^main) Method Variable Table Contents:
```

Variable Name	Temp	Type
_^return^_main	t102	INT
d	t100	PTR
w	t101	PTR

最终转换成的中间代码如下所示，部分原理已经展示在代码中：

```

<?xml version="1.0" encoding="UTF-8"?>
<Program>
  <FunctionDeclaration name="_^main^_main" return_type="INT" last_temp="107" last_label="100">
    <Blocks>
      <Block entry_label="100">
        <Sequence>
          <Label label="100"/>
          <Move>
            <Temp type="PTR" temp="100"/>
            <ExtCall extfun="malloc" type="PTR">
              <Arguments>
                <Const value="16"/> // 类表大小为16
              </Arguments>
            </ExtCall>
          </Move>
          <Move>
            <Memory type="PTR">
              <BinOp op="+" type="PTR">
                <Temp type="PTR" temp="100"/>
                <Const value="0"/> // a对应的偏置
              </BinOp>
            </Memory>
          <ESeq>
            <Sequence>
              <Move>
                <Temp type="PTR" temp="103"/>
                <ExtCall extfun="malloc" type="PTR">
                  <Arguments>
                    <Const value="4"/>
                  </Arguments> // 数组a长度为0, 故开 (0+1) *4大小
                </ExtCall>
              </Move>
              <Move>
                <Memory type="INT">
                  <Temp type="PTR" temp="103"/>
                </Memory>
                <Const value="0"/> // 0号位填入长度0
              </Move>
            </Sequence>
            <Temp type="PTR" temp="103"/>
          <ESeq>
        </Move>
        <Move>
          <Memory type="PTR">
            <BinOp op="+" type="PTR">
              <Temp type="PTR" temp="100"/>
              <Const value="4"/> // 变量x对应偏置
            </BinOp>
          </Memory>
        <ESeq>
          <Sequence>
            <Move>
              <Temp type="INT" temp="105"/>
              <Const value="1"/> // 变量x初始化
            </Move>
          </Sequence>
          <Temp type="INT" temp="105"/>
        <ESeq>
      </Move>
    </Blocks>
  </FunctionDeclaration>
</Program>

```

```

<Move>
  <Memory type="PTR">
    <BinOp op="+" type="PTR">
      <Temp type="PTR" temp="100"/>
      <Const value="12"/> // 数组变量z对应偏置
    </BinOp>
  </Memory>
<ESeq>
  <Sequence>
    <Move>
      <Temp type="PTR" temp="106"/>
      <ExtCall extfun="malloc" type="PTR">
        <Arguments>
          <Const value="16"/> // 数组大小 (3+1) *4=16
        </Arguments>
      </ExtCall>
    </Move>
    <Move>
      <Memory type="INT">
        <Temp type="PTR" temp="106"/>
      </Memory>
      <Const value="3"/> // 填入z的长度
    </Move>
    <Move>
      <Memory type="INT">
        <BinOp op="+" type="PTR">
          <Temp type="PTR" temp="106"/>
          <Const value="4"/>
        </BinOp>
      </Memory>
      <Const value="1"/> // z[0]
    </Move>
    <Move>
      <Memory type="INT">
        <BinOp op="+" type="PTR">
          <Temp type="PTR" temp="106"/>
          <Const value="8"/>
        </BinOp>
      </Memory>
      <Const value="2"/> // z[1]
    </Move>
    <Move>
      <Memory type="INT">
        <BinOp op="+" type="PTR">
          <Temp type="PTR" temp="106"/>
          <Const value="12"/>
        </BinOp>
      </Memory>
      <Const value="3"/> // z[2]
    </Move>
  </Sequence>
  <Temp type="PTR" temp="106"/>
<ESeq>
<Move>
  <Temp type="PTR" temp="101"/>
  <ExtCall extfun="malloc" type="PTR">
    <Arguments>
      <Const value="4"/> // main方法变量w
    </Arguments>

```

```

        </ExtCall>
    </Move>
    <Move>
        <Memory type="INT">
            <Temp type="PTR" temp="101"/>
        </Memory>
        <Const value="0"/> // w长度为0
    </Move>
    <Return>
        <Const value="1"/> // 返回值1
    </Return>
</Sequence>
</Block>
</Blocks>
</FunctionDeclaration>
</Program>

```

## 五、指令选择以生成四元组表示

在编译器的优化和代码生成阶段，将中间表示（IR）转换为四元组（Quad）表示是一种常见的做法。四元组是一种低级别的中间语言，它为编译器的后端提供了一种简洁、高效的指令集，便于进行进一步的优化和目标代码生成。

### 5.1 四元组中间语言的描述

四元组中间语言是一种基于四元组的表示方法，每个四元组包含了一个操作符和最多三个操作数，以及可能的目标位置。这种表示方法既能够表达复杂的控制流和数据流，又能够保持足够的简洁性，便于编译器的后端处理。

四元组中间语言的结构通常包括以下几种类型的指令：

- **QuadProgram**: 顶层结构，包含一个或多个函数定义。
- **QuadFuncDecl**: 函数定义，包含了该函数的所有基本块和控制流指令。
- **QuadBlock**: 函数体中的基本结构，它包含了一系列的语句。
- **QuadMove**: 数据传输指令，如 `temp <- temp`。
- **QuadLoad**: 从内存加载数据到寄存器，如 `temp <- mem(temp)`。
- **QuadStore**: 将寄存器中的数据存储到内存，如 `mem(temp) <- temp`。
- **QuadMoveBinop**: 执行二元操作并将结果存储到寄存器，如 `temp <- temp op temp`。
- **QuadCall**: 函数调用指令，忽略返回结果。
- **QuadExtCall**: 外部函数调用指令，忽略返回结果。
- **QuadMoveCall**: 函数调用并将结果存储到寄存器。
- **QuadMoveExtCall**: 外部函数调用并将结果存储到寄存器。
- **QuadLabel**: 标签指令，用于控制流的跳转。
- **QuadJump**: 无条件跳转指令。
- **QuadCJump**: 条件跳转指令，如 `cjump relop temp, temp, label, label`。
- **QuadPhi**: Phi函数，用于在控制流合并点生成新的临时变量（本步骤使用不到，无需关注）
- **QuadReturn**: Return函数，返回值。

### 5.2 准备工作：IR的规范化

在编译器的优化和代码生成阶段之前，一个重要的步骤是中间表示（IR）的规范化。规范化过程的目的是将IR转换为一种标准形式，这有助于简化后续的优化和代码生成过程。规范化后的IR更容易分析，因为它具有更少的变体和更统一的结构。

#### 5.2.1 规范化的目的

规范化IR的主要目的包括：

1. **消除 Eseq 节点:** Eseq 节点表示一个表达式序列，其中表达式的结果被忽略。规范化过程将这些序列分解为单独的语句，使得每个表达式的效果更加明确。
2. **统一IR结构:** 通过消除复杂的IR结构，如嵌套的序列和混合的语句/表达式，规范化过程使得IR的结构更加统一和简洁。
3. **简化优化:** 规范化后的IR更容易进行优化，因为优化算法可以更直接地应用于简化后的结构。

## 5.2.2 规范化的过程

规范化过程涉及以下几个关键步骤：

1. **线性化嵌套的序列 (SEQ) :**
  - 将所有嵌套的序列结构线性化为单个序列，这有助于简化控制流的表示。
2. **消除 Eseq 节点:**
  - Eseq 节点包含一个语句和一个表达式，其中表达式的结果被忽略。规范化过程将这些节点分解为单独的语句和表达式，确保每个语句和表达式的效果都被正确处理。
3. **重新排序语句和表达式:**
  - 在代码块中，将所有语句移动到前面执行，然后使用表达式。这涉及到使用临时变量来存储表达式的值，然后将这些值用于后续的计算。

## 5.2.3 规范化IR的示例

以下所示的代码为例：

```
y = (x = x + 1)
```

在IR中，这可能被表示为：

```
Move(y, Eseq(x=x+1, x))
```

规范化过程将消除 Eseq 节点，将其转换为一系列简单的操作：

```
x = x + 1;
move(y, x)
```

这一过程不仅简化了IR的结构，还使得每个操作的效果更加明确，便于后续的优化和代码生成。

## 5.3 关键结构

### 5.3.1 Tree2Quad类

```
class Tree2Quad : public Visitor {
public:
    QuadProgram* quadprog;           // 接口函数返回值
    vector<QuadStm*> *visit_result; // 存储block里的所有Quad语句
    QuadTerm *output_term;          // 传递temp/name/const
    Temp_map *temp_map;             // 用于生成新的temp
}
```

转换采用Visitor模式实现，通过遍历IR Tree的各个节点，生成对应的Quad指令。主要组件包括：

- Tree2Quad 类：核心转换器，继承自Visitor接口
- QuadTerm 类：表示Quad中的基本项（临时变量、常量、名称）

### 5.3.2 Quad基本类型QuadTerm

表示Quad中的基本项，有三种类型：

```
enum class QuadTermKind {
    TEMP,      // 临时变量
    CONST,     // 常量
    NAME       // 名称(标签)
};
```

### 5.3.3 变量管理

#### 1. Temp\_map

`Temp_map` 用于管理临时变量和标签。它负责生成唯一的临时变量名和标签，以避免命名冲突，并确保每个变量或标签在整个程序中具有唯一性。

- **临时变量**: 在中间表示中，临时变量通常用于存储中间计算结果。例如，在表达式 `a + b` 中，可能需要一个临时变量来存储加法的结果，然后再用于后续的计算。
- **标签**: 标签用于控制流指令，如跳转和条件跳转。它们标识程序中的特定位置，以便执行跳转操作。

#### 2. 记录变量的定义 (def) 和使用 (use) 集合

定义 (def) 和使用 (use) 集合是数据流分析中的两个基本概念，它们对于理解变量的生命周期和优化代码至关重要。

- **定义 (def)** : 变量的定义是指在程序中给变量赋值的位置。在四元组表示中，定义通常出现在 `MOVE`、`LOAD`、`STORE` 和 `CALL` 等指令中，这些指令将值赋给变量或内存位置。
- **使用 (use)** : 变量的使用是指在程序中读取变量值的位置。在四元组表示中，使用通常出现在需要读取变量值以进行计算或比较的指令中。

记录这些集合有助于编译器进行多种优化，如死代码消除（如果一个变量定义后从未被使用，那么这个定义可以被消除）、公共子表达式消除（如果一个表达式多次出现且中间没有修改其变量的操作，那么可以只计算一次并重复使用结果）等。

## 5.4 主要转换逻辑

#### 1. 程序结构转换

- **Program节点**: 转换为QuadProgram，包含函数声明列表
- **FuncDecl节点**: 转换为QuadFuncDecl，包含基本块列表和函数参数
- **Block节点**: 转换为QuadBlock，包含指令序列和入口/出口标签

#### 2. 指令转换

- **Move指令**: 根据目标类型分为多种情况处理：
  - 内存存储: 转换为QuadStore指令
  - 外部调用结果: 转换为QuadMoveExtCall指令
  - 函数调用结果: 转换为QuadMoveCall指令
  - 二元运算结果: 转换为QuadMoveBinop指令
  - 简单赋值: 转换为QuadMove指令
  - 内存加载: 转换为QuadLoad指令
- **内存访问分为两种模式**
  - 加载模式(Mem作为源): 生成QuadLoad指令
  - 存储模式(Mem作为目标): 生成QuadStore指令
- **控制流指令**:
  - Cjump: 转换为条件跳转指令QuadCJump，保留relop和标签信息
  - Jump: 转换为无条件跳转指令QuadJump
  - LabelStm: 转换为标签指令QuadLabel
- **函数调用**:
  - Call: 处理对象和参数，生成QuadCall指令
  - ExtCall: 处理外部函数调用，生成QuadExtCall指令
  - 区分有返回值和无返回值的调用

### 3. 表达式转换

- **Binop**: 处理二元运算，生成QuadMoveBinop语句，并产生中间临时变量
- **Mem**: 处理内存访问，生成QuadLoad/QuadStore指令
- **TempExp/Const/Name**: 转换为QuadTerm

## 5.5 示例

考虑这段源程序：

```
public int main() {  
    int x = 1;  
    x = 1 + 2 * 3 > 3 || 4 && 5==0;  
    return(({x=x+1;} x));  
}
```

它的IR表示如下所示，其中包含很多的Eseq节点，嵌套结构很复杂。

```

<?xml version="1.0" encoding="UTF-8"?>
<Program>
  <FunctionDeclaration name="_^main^_main" return_type="INT" last_temp="102" last_label="111">
    <Blocks>
      <Block entry_label="111">
        <Sequence>
          <Label label="111"/>
          <Move>
            <Temp type="INT" temp="100"/>
            <Const value="1"/>
          </Move>
          <Move>
            <Temp type="INT" temp="100"/>
          </Move>
          <ESeq>
            <Sequence>
              <Move>
                <Temp type="INT" temp="102"/>
                <Const value="0"/>
              </Move>
              <Sequence>
                <CJump relop="&gt;" true="108" false="107">
                  <BinOp op="+" type="INT">
                    <Const value="1"/>
                    <BinOp op="*" type="INT">
                      <Const value="2"/>
                      <Const value="3"/>
                    </BinOp>
                  </BinOp>
                  <Const value="3"/>
                </CJump>
                <Label label="107"/>
                <Sequence>
                  <CJump relop="!=" true="104" false="109">
                    <Const value="4"/>
                    <Const value="0"/>
                  </CJump>
                  <Label label="104"/>
                  <CJump relop="==" true="108" false="109">
                    <Const value="5"/>
                    <Const value="0"/>
                  </CJump>
                </Sequence>
              </Sequence>
              <Label label="108"/>
              <Move>
                <Temp type="INT" temp="102"/>
                <Const value="1"/>
              </Move>
              <Label label="109"/>
            </Sequence>
            <Temp type="INT" temp="102"/>
          </ESeq>
        </Move>
        <Return>
          <ESeq>
            <Sequence>
              <Move>
                <Temp type="INT" temp="100"/>
                <BinOp op="+" type="INT">

```

```

<Temp type="INT" temp="100"/>
<Const value="1"/>
</BinOp>
</Move>
</Sequence>
<Temp type="INT" temp="100"/>
</ESeq>
</Return>
</Sequence>
</Block>
</Blocks>
</FunctionDeclaration>
</Program>

```

转化为Quad表示为：

```

Function _^main^_main() last_label=111 last_temp=104:
Block: Entry Label: L111
Exit labels:
LABEL L111; def: use:
MOVE t100:INT <- Const:1; def: 100 use:
// 计算1+2*3>3
MOVE t102:INT <- Const:0; def: 102 use:
// 先算2*3, 存入定义的中间变量t103
MOVE_BINOP t103:INT <- (*, Const:2, Const:3); def: 103 use:
// 再算1+t103, 存入定义的中间变量t104
MOVE_BINOP t104:INT <- (+, Const:1, t103:INT); def: 104 use: 103
// 使用计算结果t104 与 3进行大小比较
CJUMP > t104:INT Const:3? L108 : L107; def: use: 104
LABEL L107; def: use:
CJUMP != Const:4 Const:0? L104 : L109; def: use:
LABEL L104; def: use:
CJUMP == Const:5 Const:0? L108 : L109; def: use:
LABEL L108; def: use:
MOVE t102:INT <- Const:1; def: 102 use:
LABEL L109; def: use:
MOVE t100:INT <- t102:INT; def: 100 use: 102
MOVE_BINOP t100:INT <- (+, t100:INT, Const:1); def: 100 use: 100
RETURN t100:INT; def: use: 100

```

显然，转化为Quad表示，代码更加简洁清晰，复杂的过程用中间变量进行拆解，为后续的优化和代码生成阶段提供支持。

## 六、将Quad转换为SSA形式

静态单赋值 (Static Single Assignment, SSA) 优化是一种编译器优化技术，它通过确保**每个变量在整个程序中只被赋值一次**来简化变量的分析和优化。在SSA形式的程序中，每个变量定义对应一个唯一的赋值，这有助于消除公共子表达式和死代码。

在本实验中，我们实现了一个将Quad转换为SSA形式的过程。这个过程涉及到几个关键步骤，包括**删除不可达代码块、放置Phi函数、重命名变量和清理未使用的Phi函数**。

### 6.1 静态单赋值形式

SSA形式通常包含以下元素：

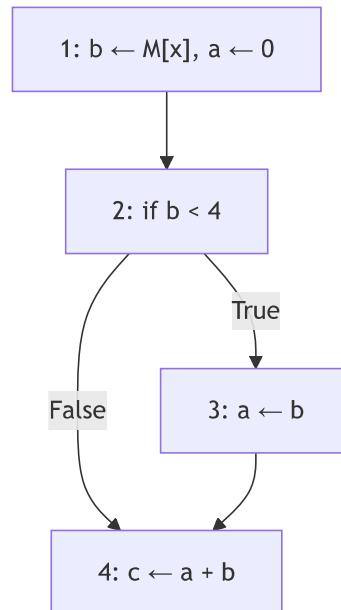
- Phi函数**: Phi函数用于在多个前驱块 (predecessors) 合并变量的值。它们通常出现在控制流图中的合并点 (如循环的头部)。

- **版本号**: 每个变量的定义都有一个版本号, 表示该变量的特定版本。
- **重命名**: 变量的重命名反映了它们的版本号, 例如  $x_1$ ,  $x_2$  等。

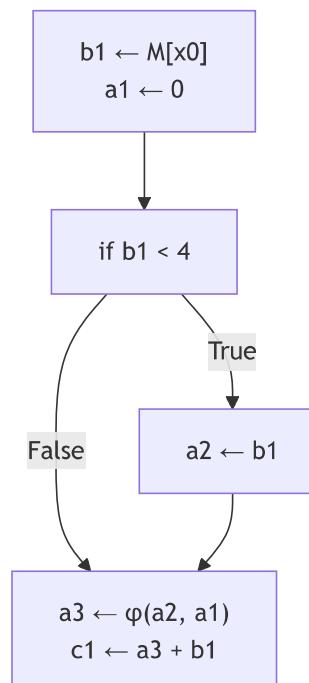
## SSA的示例

### 原始程序

1.  $b$  从内存位置  $x$  加载值。
2.  $a$  被初始化为0。
3. 检查  $b$  是否小于4。
4. 如果条件为真,  $a$  被赋值为  $b$ 。
5. 否则,  $c$  被赋值为  $a$  和  $b$  的和。



在SSA形式中, 我们需要为每个变量的每个新版本创建一个新的变量。这涉及到引入Phi函数来处理变量值的合并。



1.  $b_1$  从内存位置  $x_0$  加载值。
2.  $a_1$  被初始化为0。
3. 检查  $b_1$  是否小于4。
4. 如果条件为真,  $a_2$  被赋值为  $b_1$ 。
5. 否则,  $a_3$  被赋值为Phi函数的结果, 该函数根据控制流的来源选择  $a_1$  或  $a_2$  的值。

6. `c1` 被赋值为 `a3` 和 `b1` 的和。

通过这种方式，我们确保了每个变量的每个版本都是唯一的，从而简化了后续的优化步骤，如寄存器分配和死代码消除。

### SSA形式的主要优势：

1. **简化优化**: SSA形式使得编译器更容易识别和消除公共子表达式，因为它清楚地显示了每个变量的值是如何计算的。
2. **寄存器分配**: SSA形式简化了寄存器分配问题，因为每个变量的每个版本都是独立的，这有助于编译器决定哪些变量可以存储在寄存器中。
3. **死代码消除**: SSA形式有助于识别和消除死代码，因为编译器可以更容易追踪哪些变量的值实际上从未被使用。

## 6.2 准备工作：Quad的分块

Blocking过程旨在通过将控制流图中的基本块分割成更小的块，来优化寄存器的使用。具体实现再 `blocking.cc` 中：

- **blocking 函数**: 作为整个过程的入口点，接收一个 `QuadProgram` 对象，该对象包含一系列的函数声明（`QuadFuncDecl`）。该函数的目标是将这些函数声明转换为blocking形式。如果输入的程序对象为空，则直接返回原始程序对象。否则，创建一个新的函数声明列表 `new_func_decl_list`，用于存储转换后的函数声明。接着，遍历原始程序中的每个函数声明，并调用 `blocking` 函数对每个函数进行转换。转换后的函数声明若不为空，则添加到 `new_func_decl_list` 中。最后，使用转换后的函数声明列表创建一个新的 `quadProgram` 对象，并返回这个新程序对象。
- **blocking 静态函数**: 负责将单个函数声明转换为blocking形式。它首先检查输入的函数声明是否为空，如果为空，则直接返回原始函数声明。然后，创建一个新的 `Temp_map` 对象，用于管理新的临时变量和标签的生成。初始化一个新的基本块列表 `new_blocks`，用于存储分割后的基本块。遍历函数中的每个基本块，并对每个块调用 `split_block` 函数进行分割。如果分割后的基本块列表不为空，则将其添加到 `new_blocks` 列表中。使用分割后的基本块列表创建一个新的函数声明 `new_func_decl`，并返回这个新函数声明。
- **split\_block 静态函数**: 负责将单个基本块分割成更小的块，以适应控制流的变化。首先，检查输入的基本块是否为空，如果为空，则返回空指针。创建一个新的基本块列表 `result_blocks`，用于存储分割后的基本块。初始化一个当前语句列表 `current_stmts`，用于存储当前块中的语句。遍历基本块中的每个语句，并根据语句的类型进行不同的处理：如果语句是标签（`LABEL`），则可能需要在此处分割基本块。如果语句是跳转（`JUMP`）、条件跳转（`CJUMP`）、返回（`RETURN`）或外部调用（`EXTCALL`），则在此处分割基本块，因为这些语句会改变控制流。在分割点处，创建一个新的基本块，并将当前语句列表中的语句添加到这个新块中。继续处理剩余的语句，直到所有语句都被处理完毕。最后，如果 `current_stmts` 列表中还有剩余的语句，创建一个最终的基本块，并将剩余的语句添加到这个块中。

## 6.3 将分块后的Quad转成SSA形式

### 6.3.1 删除不可抵达的代码块

1. **遍历函数的代码块**:
  - 从函数的代码块列表中逐个检查每个块。使用一个索引 `i` 来遍历列表，并检查每个块的入口标签 `entry_label`。
2. **识别不可达块**:
  - 对于每个块，检查其入口标签编号是否在 `unreachableBlocks` 集合中。如果不在集合中，说明该块是可达的，继续检查下一个块。
3. **删除不可达块**:
  - 如果块的入口标签在 `unreachableBlocks` 集合中，说明该块是不可达的。从函数的代码块列表中删除该块，并更新索引 `i`，以跳过被删除的块。
4. **重新计算控制流信息**:
  - 删除不可达块后，可能需要重新计算控制流图的某些部分，如所有块、前驱块、后继块、支配块等。因此，如果相应的集合不为空，调用相应的方法来重新计算这些信息。

### 6.3.2 放置Phi函数

#### 插入 $\phi$ 函数的算法

##### 1. 数据流信息初始化

首先，需要对函数进行数据流分析，收集以下信息：

- **变量集合**: 找出函数中所有出现的变量。
- **活跃性信息**: 计算每个基本块的变量活跃性信息, 包括变量在哪些基本块中是活跃的 (即在基本块结束时仍然可能被使用的变量)。

## 2. 构建变量的定义点集合

对于每个变量, 记录它在哪些基本块中被定义。这可以通过分析函数的控制流图 (CFG) 来完成, 将每个变量的定义点存储在一个集合中。

## 3. 初始化工作集合

对于每个变量, 初始化一个工作集合  $w$ , 初始值为该变量的所有定义点。这个集合用于记录需要进一步处理的基本块。

## 4. 主循环: 处理工作集合

当工作集合  $w$  不为空时, 重复以下步骤:

- **取出一个基本块**: 从工作集合  $w$  中取出一个基本块  $n$ 。
- **检查支配边界点**: 对于基本块  $n$  的每个支配边界点  $y$  (支配边界点是指从基本块  $n$  到基本块  $y$  的所有路径都经过  $n$ ) , 执行以下操作:
  - 检查变量是否需要在  $y$  中插入 $\phi$ 函数:
    - 如果变量在  $y$  的活跃出集 (即从  $y$  流出时仍然活跃的变量集合) 中, 且尚未在  $y$  中插入 $\phi$ 函数, 则需要插入 $\phi$ 函数。
  - 插入 $\phi$ 函数:
    - 创建一个新的 $\phi$ 函数, 其定义变量为当前处理的变量。
    - $\phi$ 函数的参数为该变量在  $y$  的所有前驱基本块中的值。
    - 将 $\phi$ 函数插入到基本块  $y$  的顶部。
  - 更新信息:
    - 将变量标记为已在  $y$  中插入 $\phi$ 函数。
    - 如果变量在  $y$  中有其他定义点, 则将这些定义点加入工作集合  $w$ , 以便后续处理。
- 重复上述步骤, 直到工作集合  $w$  为空, 表示所有需要插入 $\phi$ 函数的位置都已处理完毕。

### 6.3.3 重命名变量

#### 1. 全局数据结构

```
// 全局定义
vector<int> params_nums; // 记录函数参数的temp_num, 无需rename
set<int> is_def; // 记录当前已定义的变量, 若无def就use, 则无需rename
map<int, tree::Type> temp2type; // 记录变量与类型的映射表, 用于给phi赋值类型。
```

#### 2. 变量重命名逻辑

- (1) **检查是否为参数**
  - 如果当前变量是函数参数 (通过检查其编号是否在参数编号集合中), 则直接将其加入到  $new\_set$  中, 并返回原始变量表达式。参数不需要重命名, 因为它们在函数入口处是唯一的。
- (2) **检查是否需要重命名**
  - 如果当前操作不是定义操作 ( $def = false$ ), 并且该变量尚未被定义 (通过检查  $is\_def$  集合), 则直接将其加入到  $new\_set$  中, 并返回原始变量表达式。未定义的变量不需要重命名。
- (3) **获取版本号**
  - 如果当前操作是定义操作 ( $def = true$ ):
    - 从  $varCount$  中获取当前变量的版本号计数器值, 并将其作为新的版本号。
    - 将新的版本号压入  $varStack$  的对应变量栈中。
    - 将版本号计数器加1, 为下一次定义操作做准备。
  - 如果当前操作不是定义操作 ( $def = false$ ):
    - 从  $varStack$  的对应变量栈中获取栈顶的版本号, 作为当前变量的版本号。
- (4) **生成新的变量**
  - 根据变量编号和版本号生成新的变量编号 (版本化编号)。

- 创建一个新的变量对象（`Temp`），并将其加入到 `new_set` 中。
- 返回一个新的变量表达式，其类型与原始变量表达式相同，但变量对象是新生成的版本化变量。

### 3. 递归处理子块重名

- (1) 备份当前状态

在处理当前基本块之前，备份当前的变量版本号栈（`varStack`）和已定义变量集合（`is_def`）。这一步是为了在完成当前基本块的处理后，能够恢复到处理前的状态，以便不影响其他基本块的处理。

- (2) 遍历基本块中的语句

逐条处理基本块中的语句，先后处理use和def集合，根据语句的类型（如赋值、加载、存储、条件跳转等）分别进行处理，调用 `renameQuadTerm` 或 `renameTempExp` 函数。并在处理完后更新语句的def和use集合。

- (3) 处理  $\phi$  函数

对于当前基本块的后继基本块中的  $\phi$  函数：

- 遍历后继基本块的前驱节点，找到当前基本块的位置。
- 对于后继基本块中每个  $\phi$  函数，根据当前基本块中的变量版本号，更新  $\phi$  函数的参数。
- 如果变量未定义，则不添加版本号，直接使用原始变量编号。

- (4) 递归处理子节点

根据支配树信息，处理当前基本块的子节点（即在支配树中由当前基本块支配的其他基本块）

- (5) 恢复状态

在完成当前基本块及其子节点的处理后，恢复备份的变量版本号栈（`varStack`）和已定义变量集合（`is_def`），以便不影响其他基本块的处理。

#### 6.3.4 清理未使用的Phi函数

在放置Phi函数时要求：所有Phi节点都是在变量在该块live-out集合的情况下放置的，因此不会存在未使用的 Phi 函数，此步骤无需操作。

### 6.4 示例

考虑源Quad如下所示：

```
Function _^main^_^main() last_label=105 last_temp=104:
  Block: Entry Label: L105
  Exit labels:
  LABEL L105; def: use:
  MOVE t101:INT <- Const:19; def: 101 use:
  MOVE t103:INT <- Const:0; def: 103 use:
  CJUMP > t101:INT Const:1? L102 : L103; def: use: 101
  LABEL L102; def: use:
  MOVE t103:INT <- Const:1; def: 103 use:
  LABEL L103; def: use:
  EXTCALL putch(t103:INT); def: use: 103
  RETURN t101:INT; def: use: 101
```

基于跳转、退出等指令进行分块后变成：

```

Function _^main^_main() last_label=105 last_temp=104:
  Block: Entry Label: L105
    Exit labels: L102 L103
    LABEL L105; def: use:
    MOVE t101:INT <- Const:19; def: 101 use:
    MOVE t103:INT <- Const:0; def: 103 use:
    CJUMP > t101:INT Const:1? L102 : L103; def: use: 101
  Block: Entry Label: L102 // add
    Exit labels: L103 // add
    LABEL L102; def: use:
    MOVE t103:INT <- Const:1; def: 103 use:
    JUMP L103; def: use: // add
  Block: Entry Label: L103 // add
    Exit labels: // add
    LABEL L103; def: use:
    EXTCALL putch(t103:INT); def: use: 103
    RETURN t101:INT; def: use: 101

```

转换为SSA形式后的Quad为：

```

Function _^main^_main() last_label=105 last_temp=104:
  Block: Entry Label: L105
    Exit labels: L102 L103
    LABEL L105; def: use:
    MOVE t10100:INT <- Const:19; def: 10100 use:
    MOVE t10300:INT <- Const:0; def: 10300 use:
    CJUMP > t10100:INT Const:1? L102 : L103; def: use: 10100
  Block: Entry Label: L102
    Exit labels: L103
    LABEL L102; def: use:
    MOVE t10301:INT <- Const:1; def: 10301 use:
    JUMP L103; def: use:
  Block: Entry Label: L103
    Exit labels:
    LABEL L103; def: use:
    PHI t10302:INT <- (t10301, L102; t10300, L105); def: 10302 use: 10300 10301 // 这里插入phi节点 合并分别来自L102和L105
    EXTCALL putch(t10302:INT); def: use: 10302
    RETURN t10100:INT; def: use: 10100

```

可见，在转化SSA后，相同的变量被赋予了不同的版本号，并在汇聚时通过Phi函数进行选择。

## 七、SSA的活跃性分析

在编译器的寄存器分配阶段，理解和管理控制流信息是至关重要的。控制流信息帮助编译器理解程序中各个基本块（Basic Blocks）之间的关系，如前驱（Predecessors）、后继（Successors）、支配者（Dominator）等。这些信息对于寄存器分配算法来说非常重要，因为它们决定了哪些变量可以共享相同的寄存器，哪些变量需要单独的寄存器，以及在寄存器不足时哪些变量需要溢出到内存中。

### 7.1 用于活跃性分析的数据结构

#### 7.1.1 ControlFlowInfo

`ControlFlowInfo` 类用于存储和计算控制流信息，包括：

- `func`：指向当前处理的函数声明。
- `labelToBlock`：将标签编号映射到对应的基本块。

- `entryBlock`：函数的入口块编号。
- `allBlocks`、`unreachableBlocks`、`predecessors`、`successors`、`dominators`、`immediateDominator`、`dominanceFrontiers`、`domTree`：存储控制流图的各种信息，如所有块、不可达块、前驱块、后继块、支配者、支配边界等。

## 相关函数

- `computeAllBlocks()`：计算函数中的所有基本块。
- `computeUnreachableBlocks()`：计算不可达的基本块。
- `eliminateUnreachableBlocks()`：从函数中删除不可达的基本块。
- `computePredecessors()`：计算每个基本块的前驱块。
- `computeSuccessors()`：计算每个基本块的后继块。
- `computeDominators()`：计算每个基本块的支配者。
- `computeImmediateDominator()`：计算每个基本块的直接支配者。
- `computeDominanceFrontiers()`：计算支配边界。
- `computeDomTree()`：构建支配树。
- `computeEverything()`：计算所有控制流信息。

## 7.1.2 DataFlowInfo

`DataFlowInfo` 类用于存储和计算数据流信息，包括：

- `func`：指向当前处理的函数声明。
- `allVars`：函数中使用的所有变量集合。
- `defs`、`uses`、`liveout`、`livein`：变量的定义点、使用点、活跃出集合和活跃入集合。

## 相关函数

- `findAllVars()`：找到函数中使用的所有变量。
- `computeLiveness()`：计算每个语句的活跃性信息。
- `printLiveness()`：打印活跃性信息。

## 7.2 控制流图分析

控制流图（Control Flow Graph, CFG）是程序的一种表示，它展示了程序中各个基本块之间的控制流关系。控制流图通常由节点（代表基本块）和有向边（代表控制流）组成。控制流分析的主要目标是识别程序中的所有可达块，以及它们之间的支配关系。

### 7.2.1 控制流图的构建

控制流图的构建通常遵循以下步骤：

1. **识别基本块**：将程序的每个独立执行路径段（如条件语句、循环体等）识别为一个基本块。
2. **确定控制流**：分析程序的控制结构，确定基本块之间的控制流关系，并在控制流图中添加相应的边。

控制流图具有以下重要属性：

1. **可达性**：可达性分析用于识别程序中所有可能执行的基本块。
2. **支配关系**：支配关系用于确定哪些块在控制流图中对其他块有控制影响。一个块  $A$  支配另一个块  $B$ ，如果从程序的开始到块  $B$  的每条路径都必须经过块  $A$ 。
3. **支配边界**（Dominance Frontier）是指在控制流图中，从一块到另一块的路径上，所有在路径上但不在路径终点块内的块的集合。
4. **支配树**：支配树是一种树形结构，用于表示控制流图中块之间的支配关系。

### 7.2.2 控制流图的计算

控制流图的计算包括以下步骤：

1. **计算所有块**：遍历程序，识别并记录所有基本块。
2. **计算不可达块**：识别并记录在控制流图中不可达的基本块。

3. **计算支配者**: 对于每个基本块, 计算其支配者集合。
4. **计算支配边界**: 对于每个基本块, 计算其支配边界。
5. **构建支配树**: 基于支配关系构建支配树。

## 7.3 活跃性分析

### 7.3.1 活跃性

**活跃性**: 如果从某个边 (即控制流图中的一条有向边) 到变量的某个使用点存在一条有向路径, 并且这条路径不经过任何对该变量的定义 (def), 则该变量在该边上是活跃的。如果一个变量在某个节点的任何入边 (即指向该节点的边) 上都是活跃的, 那么该变量在该节点是活跃进入 (live-in) 的; 如果它在该节点的任何出边上都是活跃的, 那么该变量在该节点是活跃出去 (live-out) 的。

在编译器设计中, 理解变量的活跃性对于优化寄存器使用和中间代码生成至关重要。通过分析变量的活跃性, 编译器可以确定在任何给定时间点哪些变量是必要的, 从而进行更有效的寄存器分配和消除不必要的变量存储。

### 7.3.2 数据流方程

1. **活跃进入 (Live-In) 方程**:

$$\text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n])$$

这个方程表示节点  $n$  的活跃进入集合由两部分组成: 节点  $n$  的使用集合 ( $\text{use}[n]$ ) , 以及节点  $n$  的活跃出去集合 ( $\text{out}[n]$ ) 减去定义集合 ( $\text{def}[n]$ ) 。

2. **活跃出去 (Live-Out) 方程**:

$$\text{out}[n] = \bigcup_{s \in \text{succ}[n]} \text{in}[s]$$

这个方程表示节点  $n$  的活跃出去集合是所有后继节点  $s$  的活跃进入集合的并集。

### 7.3.3 计算活跃性的算法

```

for each n
    in[n] ← {};
    out[n] ← {};
repeat
    for each n
        in'[n] ← in[n];
        out'[n] ← out[n]
        in[n] ← use[n] ∪ (out'[n] - def[n])
        out[n] ← ∪_{s ∈ succ[n]} in[s]
until in'[n] = in[n] and out'[n] = out[n] for all n

```

解释:

- 初始化每个节点的活跃进入和活跃出去集合为空集。
- 重复以下步骤直到活跃进入和活跃出去集合不再变化:
  - 对于每个节点  $n$ , 将上一次迭代的活跃进入和活跃出去集合赋值给临时集合  $\text{in}'[n]$  和  $\text{out}'[n]$ 。
  - 更新节点  $n$  的活跃进入集合为节点  $n$  的使用集合与节点  $n$  的活跃出去集合减去定义集合的并集。
  - 更新节点  $n$  的活跃出去集合为节点  $n$  的所有后继节点  $s$  的活跃进入集合的并集。
- 检查是否所有节点的活跃进入和活跃出去集合都已稳定 (即本次迭代与上次迭代结果相同) 。

## 7.4 块流图和干扰图分析

块流图 (Block Flow Graph) 和干扰图 (Interference Graph) 是编译器中用于优化代码的重要工具，特别是在寄存器分配和指令调度阶段。

### 7.4.1 块流图

块流图是一种图表示，它描述了程序中基本块（基本块是控制流图中的节点）之间的数据依赖关系。块流图的构建基于活跃性分析的结果，用于确定哪些变量在程序的不同部分之间流动。

#### 块流图的构建

1. **识别活跃变量**: 基于活跃性分析的结果，识别在基本块之间传递的变量。
2. **确定数据依赖**: 分析程序中变量的读写操作，确定哪些变量在不同基本块之间存在数据依赖。
3. **构建图结构**: 根据数据依赖关系，构建块流图，图中的节点表示基本块，边表示数据依赖。

#### 块流图重要属性

1. **数据依赖**: 块流图中的边表示基本块之间的数据依赖关系。
2. **可达性**: 通过块流图可以分析程序中哪些基本块是可达的。
3. **循环**: 块流图可以帮助识别程序中的循环结构。

#### 块流图的计算

1. **计算活跃变量**: 基于活跃性分析的结果，计算每个基本块的活跃变量集。
2. **确定数据依赖**: 对于每个基本块，确定哪些变量被读或写，并分析这些操作之间的依赖关系。
3. **构建图**: 基于数据依赖关系，构建块流图。

### 7.4.2 干扰图

干扰图是寄存器分配中使用的一种图表示，它描述了程序中哪些变量可能在同一时间点被访问，从而可能发生寄存器冲突。

#### 干扰图的构建

1. **识别活跃变量**: 基于活跃性分析的结果，识别在基本块中活跃的变量。
2. **确定冲突**: 分析程序中变量的读写操作，确定哪些变量在同一时间点被访问，从而可能发生冲突。
3. **构建图结构**: 根据冲突关系，构建干扰图，图中的节点表示变量，边表示冲突。

#### 干扰图的属性

1. **冲突**: 干扰图中的边表示变量之间的冲突关系。
2. **寄存器分配**: 干扰图用于指导寄存器分配，避免冲突。
3. **优化**: 通过分析干扰图，可以进行指令调度和寄存器分配优化。

#### 干扰图的计算

1. **计算活跃变量**: 基于活跃性分析的结果，计算每个基本块的活跃变量集。
2. **确定冲突**: 对于每个基本块，确定哪些变量在同一时间点被访问，从而可能发生冲突。
3. **构建图**: 基于冲突关系，构建干扰图。

## 7.5 示例

考虑一段Quad源程序：

```

Function _^main^_main() last_label=105 last_temp=125:
Block: Entry Label: L105
Exit labels: L100
LABEL L105; def: use:
MOVE_EXTCALL t100:PTR <- malloc(Const:20); def: 100 use:
STORE Const:4 -> Mem(t100:PTR); def: use: 100
MOVE_EXTCALL t101:INT <- getarray(t100:PTR); def: 101 use: 100
LOAD t104:INT <- Mem(t100:PTR); def: 104 use: 100
MOVE_BINOP t116:INT <- (+, t104:INT, Const:1); def: 116 use: 104
MOVE_BINOP t111:INT <- (*, t116:INT, Const:4); def: 111 use: 116
MOVE_EXTCALL t105:PTR <- malloc(t111:INT); def: 105 use: 111
STORE t104:INT -> Mem(t105:PTR); def: use: 104 105
MOVE t106:INT <- Const:4; def: 106 use:
MOVE_BINOP t117:INT <- (+, t104:INT, Const:1); def: 117 use: 104
MOVE_BINOP t107:INT <- (*, t117:INT, Const:4); def: 107 use: 117
JUMP L100; def: use:
Block: Entry Label: L100
Exit labels: L101 L102
LABEL L100; def: use:
CJUMP < t106:INT t107:INT? L101 : L102; def: use: 106 107
Block: Entry Label: L101
Exit labels: L100
LABEL L101; def: use:
MOVE_BINOP t118:PTR <- (+, t105:PTR, t106:INT); def: 118 use: 105 106
MOVE_BINOP t119:PTR <- (+, t100:PTR, t106:INT); def: 119 use: 100 106
LOAD t120:INT <- Mem(t119:PTR); def: 120 use: 119
MOVE_BINOP t121:INT <- (-, Const:0, t120:INT); def: 121 use: 120
STORE t121:INT -> Mem(t118:PTR); def: use: 118 121
MOVE_BINOP t106:INT <- (+, t106:INT, Const:4); def: 106 use: 106
JUMP L100; def: use:
Block: Entry Label: L102
Exit labels: L103 L104
LABEL L102; def: use:
MOVE t100:PTR <- t105:PTR; def: 100 use: 105
EXTCALL putarray(Const:4, t100:PTR); def: use: 100
MOVE t115:PTR <- t100:PTR; def: 115 use: 100
LOAD t108:INT <- Mem(t100:PTR); def: 108 use: 100
CJUMP >= Const:0 t108:INT? L103 : L104; def: use: 108
Block: Entry Label: L103
Exit labels:
LABEL L103; def: use:
EXTCALL exit(Const:-1); def: use:
Block: Entry Label: L104
Exit labels:
LABEL L104; def: use:
MOVE_BINOP t122:INT <- (+, Const:0, Const:1); def: 122 use:
MOVE_BINOP t123:INT <- (*, t122:INT, Const:4); def: 123 use: 122
MOVE_BINOP t124:PTR <- (+, t115:PTR, t123:INT); def: 124 use: 115 123
LOAD t125:INT <- Mem(t124:PTR); def: 125 use: 124
RETURN t125:INT; def: use: 125

```

按照上述算法，进行分析，结果如下所示，可以自行验证。

```
-----  
Computing all blocks in function: _^main^_main  
#blocks = 6  
All blocks in function: _^main^_main  
100 L100  
101 L101  
102 L102  
103 L103  
104 L104  
105 L105  
-----  
Predecessors:  
Block: 100 <- 101 105  
Block: 101 <- 100  
Block: 102 <- 100  
Block: 103 <- 102  
Block: 104 <- 102  
-----  
Successors:  
Block: 100 -> 101 102  
Block: 101 -> 100  
Block: 102 -> 103 104  
Block: 105 -> 100  
-----  
Computing dominators for: _^main^_main  
Initial Dominators for: _^main^_main  
Block: 100 <- 100 101 102 103 104 105  
Block: 101 <- 100 101 102 103 104 105  
Block: 102 <- 100 101 102 103 104 105  
Block: 103 <- 100 101 102 103 104 105  
Block: 104 <- 100 101 102 103 104 105  
Block: 105 <- 105  
Final Dominators for: _^main^_main  
Block: 100 <- 100 105  
Block: 101 <- 100 101 105  
Block: 102 <- 100 102 105  
Block: 103 <- 100 102 103 105  
Block: 104 <- 100 102 104 105  
Block: 105 <- 105  
Block: 100 <- 100 105  
Block: 101 <- 100 101 105  
Block: 102 <- 100 102 105  
Block: 103 <- 100 102 103 105  
Block: 104 <- 100 102 104 105  
Block: 105 <- 105  
-----  
Start to find immediate dominators for: _^main^_main  
Immediate Dominators for: _^main^_main  
Block: 100 -> 105  
Block: 101 -> 100  
Block: 102 -> 100  
Block: 103 -> 102  
Block: 104 -> 102  
Block: 105 -> -1  
Block: 100 -> 105  
Block: 101 -> 100  
Block: 102 -> 100  
Block: 103 -> 102  
Block: 104 -> 102
```

```

Block: 105 -> -1
-----
Computing dominator tree for: _^main^_^main
Final Dominator Tree for: _^main^_^main
Block: 100 -> 101 102
Block: 101 ->
Block: 102 -> 103 104
Block: 103 ->
Block: 104 ->
Block: 105 -> 100
-----
Dominator Tree:
Block: 100 -> 101 102
Block: 101 ->
Block: 102 -> 103 104
Block: 103 ->
Block: 104 ->
Block: 105 -> 100
-----
Computing dominance frontier for: _^main^_^main
Final Dominance Frontier
Block: 100 -> 100
Block: 101 -> 100
-----
Dominance Frontier:
Block: 100 -> 100
Block: 101 -> 100
-----
```

## 八、寄存器分配

寄存器分配是编译器优化过程中的一个重要步骤，旨在将程序中的变量分配到有限的寄存器资源上，以减少内存访问次数，提高程序运行效率。本部分中，我们将探讨如何进行寄存器分配，包括执行的步骤有：简化（simplify）、合并（coalesce）、冻结（freeze）和溢出（spill）处理。

### 8.1 寄存器分配准备

在编译器的寄存器分配阶段之前，需要对中间表示（SSA）进行一系列准备工作，以确保寄存器分配过程能够顺利进行。这些准备工作包括处理函数入口、返回值、Phi节点、调用指令和其他指令。以下是这些准备工作的详细描述：

- **prepareRegAlloc 函数**

**整个寄存器分配准备工作的入口点。** 它接收一个 `QuadProgram` 对象，该对象包含了一系列的函数声明（`QuadFuncDecl`），并以 SSA形式的 `Quad` 作为输入。

- **处理每个函数声明：** 遍历原始程序中的每个函数声明，并为每个函数调用 `prepareRegAlloc` 函数进行寄存器分配的准备工作。

- **克隆函数声明：** 为了避免修改原始函数声明，首先对每个函数声明进行克隆操作，以创建一个新的函数声明。

- **调用 handleEntry、handleReturn、handlePhi、handleCall 和 handleOtherStm 函数：** 对克隆后的函数声明分别调用这些函数，以处理函数入口、返回值、Phi节点、调用指令和其他指令。

- **handleEntry 函数**

- **用于处理函数入口。** 它为每个函数声明添加指令，将寄存器 `r0` 到 `r3` 的值移动到对应的临时变量中。这些移动指令被添加到函数的第一个基本块的第一个指令之后。

- **handleReturn 函数**

- **用于处理函数返回指令。** 它为每个返回指令添加指令，将返回值移动到寄存器 `r0` 中，并将返回指令更改为 `r0`。

- **handlePhi 函数**

- **用于处理Phi节点。** Phi节点用于在控制流图中的合并点合并变量的值。对于每个Phi节点，添加移动指令，将Phi值从原始块移动到Phi函数的临时变量中。
- **handleCall 函数**
  - **用于处理调用指令。** 它为每个调用指令添加指令，将参数从寄存器 `r0` 到 `r3` 移动到对应的临时变量中，并将调用返回值移动到 `r0` 中。
  - **调用者寄存器 (Caller Registers)**  
调用者寄存器用于在函数调用时保存参数和接收返回值。在ARM架构中，前四个寄存器 (`r0-r3`) 通常用于传递参数，而返回值通常存储在`r0`中。
    - **保存参数：** 在函数调用时，调用者需要将参数移动到指定的寄存器中。这可以通过创建移动指令 (QuadMove) 来实现，将参数从临时变量移动到寄存器。
    - **接收返回值：** 函数返回时，返回值通常存储在`r0`中。调用者需要从`r0`中移动返回值到目标变量中。
  - **被调用者寄存器 (Callee Registers)**  
被调用者寄存器用于在函数内部存储局部变量和临时值。在函数执行期间，这些寄存器可能会被覆盖，因此在函数调用时需要保存这些寄存器的值。
    - **保存寄存器：** 在函数开始时，被调用者需要保存其使用的寄存器的值，以便在函数结束时恢复。这通常通过将寄存器的值移动到栈上来实现。
    - **恢复寄存器：** 在函数结束时，被调用者需要从栈中恢复这些寄存器的值。
- **handleOtherStm 函数**
  - **用于处理其他类型的指令**，如存储指令、二元操作指令等。它为这些指令添加必要的移动指令，以确保操作数被正确地移动到寄存器中。

通过这些准备工作，编译器为寄存器分配阶段准备了优化的中间表示，确保了寄存器分配过程能够顺利进行。

## 8.2 寄存器分配的关键要点

### 8.2.1 核心数据结构

代码中使用以下核心数据结构辅助寄存器分配工作的顺利进行：

- `graph`：邻接表表示的干扰图，记录变量间的冲突关系
- `movePairs`：活跃的移动指令集合(如 $x=y$ )
- `simplifiedNodes`：简化阶段移除的节点栈
- `coalescedMoves`：记录合并关系的映射表
- `spilled`：需要溢出的变量集合
- `colors`：最终寄存器分配结果

### 8.2.2 入口函数

如下展示的coloring函数十分简洁，确实最为核心的函数。它利用干扰图和着色数量`k`，依次进行简化、合并、冻结、溢出处理，直至不发生变化，开始进行选色填充。

```
Coloring *coloring(InterferenceGraph *ig, int k) {
    Coloring *c = new Coloring(ig, k);
    while (c->simplify() || c->coalesce() || c->freeze() || c->spill()) { // Keep simplifying, coalescing, freezing, and spilling
    }
    c->select(); // Final selection of colors
    return c;
}
```

## 8.3 寄存器分配步骤

### 8.3.1 简化 (simplify)

实现逻辑：

1. 遍历干扰图，寻找所有度数小于k(可用寄存器数)且不参与移动指令的非机器寄存器节点
2. 将这些节点压入 `simplifiedNodes` 栈并从图中移除
3. 移除节点时会自动处理其所有边(通过 `eraseNode` )

#### 关键代码：

```

bool changed = false;
vector<int> to_remove;
for (const auto& [node, neighbors] : graph) {
    if (!isMachineReg(node) && neighbors.size() < k && !isMove(node)) {
        to_remove.push_back(node);
    }
}
for (int node : to_remove) {
    simplifiedNodes.push(node);
    eraseNode(node);
    changed = true;
}
return changed;

```

### 8.3.2 合并 (coalesce)

#### 实现逻辑：

1. 遍历所有移动指令对(u,v)，跳过已有冲突的(u和v相邻)
2. 确定主导节点(lead node):
  - 优先选择机器寄存器
  - 否则选择度数较大的节点
3. 检查Briggs合并准则：
  - 合并后高度数( $\geq k$ )邻居数量必须小于k
  - 合并后机器寄存器邻居数量必须小于k
4. 执行合并：
  - 将被合并节点的邻居转移到主导节点
  - 更新 `coalescedMoves` 记录递归合并关系
  - 更新相关移动指令对

#### 关键代码：

```

for (auto it = movePairs.begin(); it != movePairs.end(); ++it) {
    int u = it->first;
    int v = it->second;
    if(isAnEdge(graph, u, v)) continue;
    // 确定主导节点(lead node)
    int lead, other;
    // 规则1: 机器寄存器必须作为主导节点
    if (isMachineReg(u)) {
        lead = u;
        other = v;
    } else if (isMachineReg(v)) {
        lead = v;
        other = u;
    }
    // 规则2: 都不是机器寄存器时, 选择度数较大的作为主导节点
    else {
        lead = (graph[u].size() >= graph[v].size()) ? u : v;
        other = (lead == u) ? v : u;
    }
    // 检查合并后是否满足Briggs准则
    set<int> combined_neighbors;
    if (graph.count(lead)) combined_neighbors.insert(graph[lead].begin(), graph[lead].end());
    if (graph.count(other)) combined_neighbors.insert(graph[other].begin(), graph[other].end());

    int high_degree_count = 0;
    int machine_reg_count = 0;
    for (int n : combined_neighbors) {
        if (isMachineReg(n)){
            machine_reg_count++;
        }
        if (graph[n].size() >= k) {
            high_degree_count++;
            if (high_degree_count >= k)
                break;
        }
    }

    if (high_degree_count < k && machine_reg_count < k) {
        // 执行合并操作
        // ...
        break;
    }
}

```

### 8.3.3 冻结 (freeze)

实现逻辑:

1. 查找可冻结的移动指令(两端节点度数都小于k)
2. 从 movePairs 中移除该指令
3. 检查两端节点是否可简化(度数小于k且不再参与其他移动)
4. 将可简化节点压栈并移除

关键点:

- 冻结后可能产生新的可简化节点
- 每次调用只处理一对可冻结指令
- 冻结是当无法简化或合并时的折衷方案

**关键代码:**

```
// 查找可以冻结的移动指令 (两个节点的度数都 < k)
for (auto it = movePairs.begin(); it != movePairs.end(); it++) {
    int u = it->first;
    int v = it->second;

    bool u_in_graph = graph.count(u);
    bool v_in_graph = graph.count(v);

    if ((!u_in_graph || graph[u].size() < k) &&
        (!v_in_graph || graph[v].size() < k)) {

        // 冻结这条移动指令
        it = movePairs.erase(it);
        changed = true;

        // 冻结后可能允许继续简化
        if (u_in_graph && graph[u].size() < k && !isMove(u)) {
            simplifiedNodes.push(u);
            eraseNode(u);
        }
        if (v_in_graph && graph[v].size() < k && !isMove(v)) {
            simplifiedNodes.push(v);
            eraseNode(v);
        }
        break;
    }
}
```

### 8.3.4 溢出 (spill)

**实现逻辑:**

1. 选择溢出候选节点:
  - 非机器寄存器且未被合并
  - 当前度数最高的节点
2. 将节点压入 simplifiedNodes 栈
3. 从图中移除节点并清理相关移动指令

**关键点:**

- 这是“潜在”溢出，实际决定在select阶段做出
- 优先选择干扰最多的节点(启发式策略)
- 溢出后可能使其他节点变得可简化

**关键代码:**

```

// 选择溢出候选节点（使用简单启发式：最高度数）
int spill_candidate = -1;
size_t max_degree = 0;

for (const auto& [node, neighbors] : graph) {
    if (!isMachineReg(node) && !coalescedMoves.count(node)) {
        size_t degree = neighbors.size();
        if (degree > max_degree) {
            max_degree = degree;
            spill_candidate = node;
        }
    }
}

if (spill_candidate != -1) {
    // 将节点压入栈并从图中移除（实际溢出在select阶段决定）
    simplifiedNodes.push(spill_candidate);
    eraseNode(spill_candidate);
    // 删除所有有关的movePair
    if(isMove(spill_candidate)){
        for (auto it = movePairs.begin(); it != movePairs.end(); ) {
            if (it->first == spill_candidate || it->second == spill_candidate) {
                it = movePairs.erase(it);
            } else{
                it++;
            }
        }
    }
}

```

### 8.3.5 选色 (select)

**实现逻辑：**

1. 首先为图中剩余节点(包括机器寄存器)着色
2. 逆序处理 `simplifiedNodes` 栈中的节点：
  - 收集邻居已使用的颜色
  - 分配第一个可用颜色
  - 无可用颜色则标记为实际溢出
3. 处理合并节点的颜色传播
4. 最终验证着色有效性

**关键代码：**

```

while (!simplifiedNodes.empty()) {
    int node = simplifiedNodes.top();
    simplifiedNodes.pop();
    // 收集邻居已使用的颜色
    set<int> used_colors;
    set<int> neighbors = getNeighbors(node);
    for (int neighbor : neighbors) {
        if (colors.count(neighbor)) {
            used_colors.insert(colors[neighbor]);
        }
    }
    // 尝试分配可用颜色
    for (int c = 0; c < k; c++) {
        if (used_colors.find(c) == used_colors.end()) {
            colors[node] = c;
            break;
        }
    }
    // 分配失败，标记为实际溢出
    if (!colors.count(node)) {
        spilled.insert(node);
        colors[node] = 0;
        eraseNode(node);
    } else {
        // 处理合并节点
        if(coalescedMoves.count(node)){
            for(const auto& coalesced: coalescedMoves[node]){
                if(!isAnEdge(graph, coalesced, node))
                    colors[coalesced] = colors[node];
            }
        }
    }
}

```

## 8.4 示例

在一个干扰图如下所示的问题上进行寄存器分配：

```
Interference graph for function: _^main^_^main
Interference Graph: size=33
Node 0: 1 2 3 10000 10001 10400
Node 1: 0 10000 10001 10400
Node 2: 0 10000 10001 10400
Node 3: 0 10000 10001 10400
Node 126: 10000
Node 127: 10000 10400 11600
Node 128: 10000 10500 10600 11700
Node 129: 10000 10500 10601 10700 11800 12000
Node 130: 10800 11500
Node 131: 11500
Node 132: 11500 12200
Node 10000: 0 1 2 3 126 127 128 129 10100 10400 10500 10600 10601 10602 10700 11100 11600 11700 11800 11900 12000 12100
Node 10001: 0 1 2 3
Node 10100: 10000
Node 10400: 0 1 2 3 127 10000 10500 10600 11100 11600
Node 10500: 128 129 10000 10400 10600 10601 10602 10700 11700 11800 11900 12000 12100
Node 10600: 128 10000 10400 10500 10700 11700
Node 10601: 129 10000 10500 10700 11800 11900 12000 12100
Node 10602: 10000 10500 10700
Node 10700: 129 10000 10500 10600 10601 10602 11800 11900 12000 12100
Node 10800: 130 11500
Node 11100: 10000 10400
Node 11500: 130 131 132 10800 12200 12300
Node 11600: 127 10000 10400
Node 11700: 128 10000 10500 10600
Node 11800: 129 10000 10500 10601 10700 11900 12000 12100
Node 11900: 10000 10500 10601 10700 11800
Node 12000: 129 10000 10500 10601 10700 11800
Node 12100: 10000 10500 10601 10700 11800
Node 12200: 132 11500
Node 12300: 11500
Node 12400:
Node 12500:
Move pairs:
(0, 10100) (0, 10500) (0, 11100) (0, 12500) (10001, 10500) (10001, 11500) (10100, 0) (10500, 0) (10500, 10001) (10600, 10601)
```

根据上述的原理，执行的过程如下：

第一步：简化节点: 126 127 128 130 131 132 10800 11600 11700 12200 12300 12400 此时无法继续简化，进入合并阶段  
第二步：合并 Coalesced: 10100 into 0  
第三步：回到简化阶段，无法简化，继续合并 Coalesced: 11100 into 0  
第四步：回到简化阶段，无法简化，继续合并 Coalesced: 12500 into 0  
第五步：回到简化阶段，无法简化，继续合并 Coalesced: 11500 into 10001  
第六步：回到简化阶段，无法简化；合并阶段无法合并；冻结阶段无法冻结；最后进入溢出，选择潜在溢出节点 10000  
第七步：回到简化阶段，简化节点：11900 12100  
第八步：回到简化阶段，无法简化；合并阶段无法合并；冻结阶段无法冻结；最后进入溢出，选择潜在溢出节点 10500  
第九步：回到简化阶段，简化节点：129 10001 11800 12000  
第十步：回到简化阶段，简化节点：10700  
第十一步：无法继续简化，进入合并阶段 Coalesced: 10601 into 10600  
第十二步：回到简化阶段，无法简化，继续合并 Coalesced: 10602 into 10600  
第十三步：回到简化阶段，简化节点：10600  
第十四步：回到简化阶段，简化节点：10400  
第十五步：回到简化阶段，无法简化；合并阶段无法合并；冻结阶段无法冻结；溢出无法溢出；此时进入最终选色阶段

-----  
Interference Graph: with #colors=5:

Node 0: 1 2 3

Node 1: 0

Node 2: 0

Node 3: 0

-----  
Simplified nodes (reversed order from the simplification process): 10400 10600 10700 12000 11800 10001 129 10500 12100 11900 12200 12300 12400

-----  
Coalesced moves:

Nodes: {10100, 11100, 12500} are coalesced to node 0.

Nodes: {11500} are coalesced to node 10001.

Nodes: {10601, 10602} are coalesced to node 10600.

-----  
Remaining move pairs:

-----  
Actual spill: 10500

Actual spill: 10000

最终的颜色分配结果如下，实际溢出的节点有两个10500和10000。

```

<?xml version="1.0" encoding="UTF-8"?>
<COLORING>
  <Coloring func="_^main^_main" k="5">
    <Colors>
      <Color node="0" color="0"/>
      <Color node="1" color="1"/>
      <Color node="2" color="2"/>
      <Color node="3" color="3"/>
      <Color node="126" color="0"/>
      <Color node="127" color="1"/>
      <Color node="128" color="2"/>
      <Color node="129" color="4"/>
      <Color node="130" color="1"/>
      <Color node="131" color="0"/>
      <Color node="132" color="1"/>
      <Color node="10000" color="0"/>
      <Color node="10001" color="4"/>
      <Color node="10100" color="0"/>
      <Color node="10400" color="4"/>
      <Color node="10500" color="0"/>
      <Color node="10600" color="0"/>
      <Color node="10601" color="0"/>
      <Color node="10602" color="0"/>
      <Color node="10700" color="1"/>
      <Color node="10800" color="0"/>
      <Color node="11100" color="0"/>
      <Color node="11500" color="4"/>
      <Color node="11600" color="0"/>
      <Color node="11700" color="1"/>
      <Color node="11800" color="3"/>
      <Color node="11900" color="2"/>
      <Color node="12000" color="2"/>
      <Color node="12100" color="2"/>
      <Color node="12200" color="0"/>
      <Color node="12300" color="0"/>
      <Color node="12400" color="0"/>
      <Color node="12500" color="0"/>
    </Colors>
    <Spills>
      <Spill node="10000"/>
      <Spill node="10500"/>
    </Spills>
  </Coloring>
</COLORING>

```

## 九、RPi (ARM) 汇编代码生成

在编译器的后端阶段，将四元组 (Quad) 中间表示转换为寄存器级指令 (RPI) 格式是必要的步骤。这个过程涉及到将Quad程序转换为RPI格式，同时考虑寄存器分配和颜色映射。

### 9.1 堆栈组织框架

堆栈用于存储函数调用过程中的各种信息，包括参数、局部变量、返回地址等。以下是对堆栈组织框架的详细说明：

#### 1. 堆栈帧 (Stack Frame)

堆栈帧是函数调用时在堆栈上分配的一块内存区域，用于存储函数的局部变量、参数、返回地址等信息。每个函数调用都会创建一个新的栈帧。

- Caller Stack (调用者栈)**：当一个函数被调用时，它的调用者（Caller）的栈帧位于被调用者（Callee）的栈帧之上。调用者栈帧包含调用者的局部变量、参数、返回地址等信息。
- Callee Stack (被调用者栈)**：被调用函数的栈帧位于调用者栈帧之下，包含被调用函数的局部变量、参数、返回地址等信息。

## 2. 函数调用时的栈操作

当一个方法被调用时，会发生以下操作：

- 保存调用者信息**：调用者（Caller）的栈帧中的某些寄存器（如帧指针 `fp` 和链接寄存器 `lr`）和寄存器（如 `r4-r11`）需要被保存到栈上，以便在函数返回时恢复。
- 创建被调用者栈帧**：为被调用函数（Callee）在栈上分配一个新的栈帧，用于存储被调用函数的局部变量和参数。
- 返回地址**：调用者的返回地址（即调用指令之后的地址）被压入栈中，以便函数执行完毕后能够返回到正确的位置。

## 3. 寄存器分配

在寄存器分配过程中，需要考虑如何有效地使用有限的寄存器资源：

- 寄存器分配**：尽可能将变量分配到寄存器中，以减少内存访问次数，提高程序执行效率。
- 溢出处理**：当变量数量超过可用寄存器数量时，需要将部分变量“溢出”到内存中。这涉及到在栈上分配额外的空间来存储这些变量。

## 4. 溢出处理

- 识别溢出变量**：在寄存器分配过程中，识别出那些无法放入寄存器中的变量。
- 分配内存空间**：为这些溢出变量在栈上分配内存空间。
- 加载和存储指令**：在生成加载（Load）和存储（Store）指令，以便在需要时将溢出变量从内存加载到寄存器中，或将修改后的值存储回内存中。

# 9.2 四元式到RPI指令的转换

## 9.2.1 转换函数概述

转换过程主要包括以下几个函数：

- `normalizeName`：用于规范化函数名称，使其符合汇编器的要求。对于主函数，它将名称从 `_^main^_^main` 转换为 `main`。
- `rpi_isMachineReg`：检查给定编号是否为机器寄存器。在本实验中，寄存器编号范围为0到15。
- `temp2str`：将临时变量转换为字符串形式，考虑颜色映射和寄存器编号。如果临时变量是机器寄存器，则直接返回寄存器名称；如果临时变量被溢出，则使用溢出偏移量；否则，使用颜色映射返回对应的寄存器名称。
- `term2str`：将四元组项转换为字符串形式，处理常数和名称。对于常数，它添加前缀 `#`；对于名称，它添加前缀 `@`。
- `load_temp` 和 `store_temp`：处理加载和存储临时变量到寄存器的指令。如果临时变量被溢出，则使用溢出偏移量；否则，直接加载或存储到寄存器。
- `convert`：将Quad函数声明转换为RPI格式。它遍历函数中的每个四元组指令，根据指令类型生成相应的RPI指令。
- `quad2rpi`：将整个Quad程序转换为RPI格式。它遍历程序中的每个函数声明，调用 `convert` 函数进行转换，并将结果添加到输出字符串中。

## 9.2.2 溢出处理机制

在先前寄存器分配过程中，已经将某些变量“溢出”到内存中，在转换汇编代码过程中，要对这些溢出的变量做特殊处理：

### • 变量溢出处理

`temp2str` 函数中，如果一个临时变量通过 `color->is_spill(temp->num)` 检查后发现它被溢出到内存，则使用 `color->get_spill_offset(t->num)` 获取溢出区域中的偏移量，并生成适当的加载和存储指令。

```
string temp2str(TempExp *temp, Color *color, int reg=9) {
    Temp *t = temp->temp;
    if (rpi_isMachineReg(t->num)) {
        return "r" + to_string(t->num);
    } else if (color->is_spill(t->num)) {
        return "r" + to_string(reg);
    }
    return "r" + to_string(color->color_of(t->num));
}
```

- 如果变量是寄存器 (`rpi_isMachineReg` 返回 `true`)，则直接返回寄存器编号。
- 如果变量被溢出到内存 (`color->is_spill` 返回 `true`)，则使用溢出区域的寄存器 `reg`。
- 否则，使用颜色映射中的寄存器编号。

#### • 插入加载和存储指令

在生成的机器代码中，插入必要的加载 (Load) 和存储 (Store) 指令，以在寄存器和溢出区域之间传输变量值。

- **加载指令**：当需要访问溢出到内存的变量时，首先从内存中加载变量到寄存器。
- **存储指令**：当修改了溢出变量的值后，需要将其存储回内存中的溢出区域。

`load_temp/load_term`：对于溢出的变量，生成加载临时变量到寄存器的 RPI 汇编代码。

`store_temp/store_term` 代码类似，用于存储溢出的变量。

```
string load_temp(TempExp *temp, Color *color, int indent, int reg=9) {
    // Load a temp into a register
    Temp *t = temp->temp;
    if (color->is_spill(t->num)) {
        // If the temp is spilled, use the spill offset
        int offset = color->get_spill_offset(t->num);
        return string(indent, ' ') + "ldr r" + to_string(reg) + ", [fp, #-" + to_string(offset) + "]\n";
    } else {
        return "";
    }
}

string load_term(QuadTerm *term, Color *color, int indent, int reg=9) {
    // Load a term into a register
    if (term->kind == QuadTermKind::TEMP)
        return load_temp(term->get_temp(), color, indent, reg);
    return "";
}
```

### 9.2.3 具体转换逻辑

根据四元式操作符的类型，分别进行转换：

- **标签 (LABEL)**：将四元式中的标签转换为RPI代码中的标签，用于标记跳转目标位置。如果发现上一条指令是跳转到当前标签的指令，则可以优化掉重复的跳转指令。
- **跳转 (JUMP)**：将无条件跳转指令转换为RPI的“b”指令。
- **条件跳转 (CJUMP)**：先将条件表达式的操作数加载到寄存器中，然后生成比较指令 (cmp)，根据条件关系生成相应的条件跳转指令 (如beq、bne等)，最后添加一条无条件跳转指令跳转到假标签。
- **赋值 (MOVE)**：如果目标变量和源变量映射到同一个寄存器，则跳过该赋值操作；否则，将源变量的值加载到目标变量对应的寄存器中，并根据需要将结果存储回内存。
- **加载 (LOAD)**：根据源操作数的类型，生成相应的加载指令。如果源操作数是变量名，则需要考虑变量的存储位置（如全局变量或局部变量）来生成正确的加载指令。同时，尝试优化与前面的“add”指令组合的情况，以减少指令数量。
- **存储 (STORE)**：与加载类似，根据目标操作数的类型生成存储指令，并尝试进行优化。
- **二元运算赋值 (MOVE\_BINOP)**：将操作数加载到寄存器中，根据运算符选择相应的RPI二元运算指令（如add、sub等），执行运算并将结果存储到目标变量对应的寄存器中，最后根据需要将结果存储回内存。
- **函数调用 (CALL)**：将被调用函数的对象（如函数指针）以及参数加载到寄存器中，然后使用“blx”指令进行函数调用。
- **带返回值的函数调用 (MOVE\_CALL)**：与CALL类似，但在函数调用后，将返回值存储到目标变量对应的寄存器中，并根据需要存储回内存。
- **外部函数调用 (EXTCALL)**：将外部函数的参数加载到寄存器中，然后使用“bl”指令调用外部函数。
- **带返回值的外部函数调用 (MOVE\_EXTCALL)**：与EXTCALL类似，但在调用后处理返回值。
- **返回 (RETURN)**：将返回值加载到寄存器中，恢复栈指针和寄存器状态，然后使用“pop”指令返回到调用者。

## 9.3 示例

8.4 的示例最终生成的汇编代码如下，由于有两个溢出节点，故sp在push进各种调用者寄存器后又-8，用于存放这两个溢出节点t10000和t10500。

```
.section .note.GNU-stack

@ Here is the RPI code

@ Here's function: _^main^_^main

.balign 4
.global main
.section .text

main:
    push {r4-r10, fp, lr}
    add fp, sp, #32
    sub sp, sp, #8

main$L105:
    mov r0, #20
    bl malloc
    mov r10, r0      // 这里将malloc返回的指针移入r10 然后存入t10000对应的栈位置
    str r10, [fp, #-36]
    mov r0, #4
    ldr r10, [fp, #-36]
    str r0, [r10]
    ldr r9, [fp, #-36] // 取出t10000变量到r9中
    mov r0, r9        // 参数移入r0
    bl getarray        // 调用getarray函数，r0中是参数
    ldr r9, [fp, #-36] // 再次取出t10000变量到r9中
    ldr r4, [r9]
    add r0, r4, #1
    mov r1, #4
    mul r0, r0, r1
    bl malloc
    mov r10, r0
    str r10, [fp, #-40] // t10500变量存入栈中
    ldr r10, [fp, #-40] // t10500从栈中取出
    str r4, [r10]
    mov r0, #4
    add r1, r4, #1
    mov r2, #4
    mul r1, r1, r2

main$L100:
    ...
...
```

## 十、总结

### 10.1 本学期工作总结

本学期的编译原理课程是一个充满挑战和收获的经历。通过构建一个完整的编译器，我不仅加深了对编译器各个组成部分的理解，还提高了解决复杂问题的能力。以下是本学期工作的总结：

#### 1. 编译器的构建

在本课程中，我成功构建了一个将FDMJ语言源代码编译为RPi (ARM) 汇编代码的编译器。这个编译器涵盖了编译器的多个关键阶段，包括词法分析、语法分析、中间表示 (IR) 生成、静态单赋值 (SSA) 转换、寄存器分配以及汇编代码生成。

## 2. 各阶段的实现

- 词法分析**: 实现了一个词法分析器，能够识别FDMJ语言的关键字、标识符、字面量和操作符等，并生成词法单元序列。
- 语法分析**: 构建了一个语法分析器，根据FDMJ语言的语法规则解析词法单元序列，生成抽象语法树 (AST)。
- 中间表示 (IR) 生成**: 将AST转换为中间表示 (IR)，包括基本块、控制流图、活跃性分析、块流图和干扰图的构建。
- 静态单赋值 (SSA) 转换**: 实现了SSA转换，确保每个变量在整个程序中只被赋值一次，简化了变量分析和优化。
- 寄存器分配**: 进行了寄存器分配，包括简化、合并、冻结和溢出处理，以优化寄存器的使用。
- 汇编代码生成**: 最终将四元组程序转换为ARM汇编代码，为生成可执行的机器代码做准备。

## 3. 收获和反思

通过本课程，我获得了以下收获：

- 深入理解编译原理**: 通过实践，我对编译器的工作原理和各个阶段有了更深入的理解。
- 编程技能提升**: 在实现编译器的过程中，我的编程技能，特别是C++编程能力得到了显著提升。
- 问题解决能力**: 面对复杂的编译器实现，我学会了如何分解问题、逐步解决并优化解决方案。

# 10.2 编译器构建与使用说明

## 10.2.1 目录结构

```
project/
├── build/          # 构建目录，用于存放编译生成的文件
├── docs/           # 文档目录，存放报告文件（如 report.pdf）
├── include/         # 头文件目录
├── lib/             # 库文件目录
├── test/            # 测试文件目录
│   └── fmj_normal/ # 存放测试文件的子目录
└── vendor/          # 第三方库目录
    └── libsysy/     # 第三方库文件
└── tools/           # 工具目录，包含主程序 main
```

## 10.2.2 使用方法

### 构建项目

运行以下命令以构建项目：

```
make build
```

此命令会在当前目录下创建一个 `build` 文件夹，并在其中运行 CMake 和 Make 命令，生成项目所需的可执行文件。

### 清理项目

运行以下命令以清理项目：

```
make clean
```

此命令会删除 `build` 文件夹，并清理 `test/fmj_normal` 目录中除 `.fmj` 文件以外的所有文件。

### 重新构建项目

运行以下命令以重新构建项目：

```
make rebuild
```

此命令会先执行 `clean` 操作，然后重新执行 `build` 操作。

## 编译测试文件

运行以下命令以编译测试文件：

```
make compile
```

此命令会使用 `tools/main/main` 程序编译 `test/fmj_normal` 目录下的所有 `.fmj` 文件。

## 运行测试文件

运行以下命令以运行测试文件：

```
make run
```

此命令会编译并运行 `test/fmj_normal` 目录下的所有 `.fmj` 文件，并使用 `qemu-arm` 模拟器运行生成的汇编程序。

## 编译单个测试文件

运行以下命令以编译单个测试文件：

```
make compile-one
```

此命令会编译指定的测试文件（默认为 `hw5test0`）。可以通过修改 `FILE` 变量的值来指定其他文件。

## 运行单个测试文件

运行以下命令以运行单个测试文件：

```
make run-one
```

此命令会编译并运行指定的测试文件（默认为 `hw5test0`）。可以通过修改 `FILE` 变量的值来指定其他文件。

## 打包提交

运行以下命令以打包提交：

```
make handin
```

此命令会检查 `docs/report.pdf` 文件是否存在。如果不存在，会提示用户先生成报告。如果存在，会提示用户输入“学号-姓名”，并生成一个包含报告和项目代码的压缩文件。

## 10.2.3 注意事项

1. 确保系统执行环境是64位Ubuntu 20（或更高版本）Linux操作系统，x86架构。
2. 确保系统已安装以下工具：
  - CMake
  - Make
  - GCC（用于交叉编译）
  - QEMU（用于模拟ARM架构）
3. 修改 `FILE` 变量的值时，请确保文件名与实际测试文件一致。
4. 在运行 `make handin` 命令之前，请确保 `docs/report.pdf` 文件已生成。