Compiler Lab Final Report

Name: 韩周吾

ID: 22307130440

Date: 2025.06.19

1. 引言

本报告旨在系统说明我在《RCOMP130014h.01 编译原理》课程中完成的**FDMJ语言编译器**的设计与实现过程。该编译器以 FDMJ（一种类 Java 的教学语言）为输入，最终生成适用于 Raspberry Pi 平台（RPi，基于 ARM 架构）的汇编代码，并可通过 QEMU 模拟器运行。整个编译过程严格遵循课程提供的阶段划分，自底向上逐步实现了词法分析、语法分析、类型检查、中间表示生成（IR+）、四元式转换（Quad）、静态单赋值形式（SSA）、寄存器分配以及最终的目标代码生成。

1.1 报告目标与读者对象

本报告面向与我具有相似背景但**尚未修读本课程**的计算机专业学生。它不仅详细解释了编译器的整体结构与实现细节，还介绍了代码如何通过不同阶段处理源程序，直至生成可执行的目标代码。读者无需预先掌握课程中的全部理论细节，也能通过报告中的示例和解释理解编译器各模块的功能与实现逻辑。

1.2 编译器整体流程概述

本编译器的整体工作流程如下图所示：

```
FDMJ 源程序
    ↓
词法分析（Lexer）
    ↓
语法分析（Parser）→ 构建抽象语法树（AST）
    ↓
类型检查（Type Checker）→ 构建符号表
    ↓
中间表示生成（AST → IR+）
    ↓
指令选择（IR+ → Quad 四元式）
    ↓
SSA 变换（插入 φ 函数，变量重命名）
    ↓
活跃性分析 → 干涉图构建
    ↓
图着色寄存器分配
    ↓
目标代码生成（RPi 汇编）
```

该编译器的实现涵盖了**完整的后端管线**，能够从任意符合语法与语义的 FDMJ 程序中，自动生成结构清晰、可执行的 ARM 汇编文件，并在虚拟环境中运行以获得程序输出。

1.3 示例程序与输出说明

在整个报告中，我将使用如下简洁示例程序作为统一的演示样例，贯穿各阶段进行讲解。该程序涵盖了基本的变量定义、条件语句、循环、函数调用等核心语言结构，有助于展示编译器的处理过程与中间产物输出。

示例 FDMJ 程序（简化版）：

```
class Main {
  public static void main() {
    int a;
    a = 3;
    if (a > 0) {
      a = a + 1;
    }
    return;
  }
}
```

本程序将用于说明如下输出文件的生成过程，包括：

- 抽象语法树（AST）
- 中间表示（IR+）
- 四元式（Quad）
- SSA转换结果
- 活跃性分析与寄存器分配
- 最终生成的汇编代码

1.4 报告结构说明

章节	内容概述
第2章	描述词法与语法分析，AST构建与错误处理
第3章	说明类型检查过程及符号表的构建
第4章	AST到IR+的转换过程，涵盖中间表示设计
第5章	IR+到四元式指令的翻译策略与指令覆盖
第6章	静态单赋值形式（SSA）的插入与变量版本管理
第7章	活跃性分析、控制流图与干涉图构建
第8章	图着色寄存器分配算法与溢出处理
第9章	最终RPi汇编代码生成及栈帧布局说明
第10章	总结、构建方法与使用说明（编译器用户手册）

---

2. 语法分析

2.1 词法分析器（Lexer）

词法分析的任务是将源代码序列划分为有意义的“词法单元（token）”，并赋予类别标签。在本项目中，我使用 **Flex** 编写了 `lexer.ll` 文件，其作用如下：

- 支持关键字识别：如 `class`, `if`, `else`, `return`, `int`, `boolean`, `true`, `false` 等；
- 支持标识符与常量的识别：
  - 标识符正则：`[a-zA-Z_][a-zA-Z_0-9]*`
  - 整数常量：`[0-9]+`
- 支持符号分隔符：如 `{`, `}`, `(`, `)`, `;`, `=`, `+`, `-`, `*`, `/`, `&&`, `||`, `<`, `>` 等；
- 支持注释与空白过滤（会被忽略）；
- 每个 token 会通过 `yylval` 传递其词法值，并记录所在的行号/列号（用于错误提示）。

例如，以下代码片段：

```
int a;
a = 3;
```

会被词法分析器识别为如下 token 序列：

```
INT → ID(a) → SEMICOLON
ID(a) → ASSIGN → INT_CONST(3) → SEMICOLON
```

2.2 语法分析器（Parser）

语法分析使用 **Bison** 实现，定义在 `parser.yy` 中。我采用自顶向下递归规则，支持 FDMJ 语言的完整文法，包括类定义、主方法、变量定义、赋值、条件分支、循环、函数调用与表达式等。

核心结构设计：

- 每一个文法规则构造相应的 AST 节点（如 `new IfStmt(...)`, `new BinaryExpr(...)` 等）。
- 语义值（`%union`）用于在 `yylval` 和规则中传递结构化信息，例如 AST 指针、符号名称、整数值等。
- 报错机制通过 `%error-verbose` 与 `yyerror()` 提供清晰的错误信息。

例如，下列产生式定义了 `if` 语句的结构：

```
statement:
  IF LPAREN expression RPAREN statement %prec THEN {
    $$ = new IfStmt($3, $5, nullptr);
  }
| IF LPAREN expression RPAREN statement ELSE statement {
    $$ = new IfStmt($3, $5, $7);
  }
```

表示：`if (cond) stmt;` 或 `if (cond) stmt1 else stmt2;`，对应构造了 `IfStmt` 抽象语法树节点。

2.3 AST 构建机制

AST（抽象语法树）由语法分析器在产生式规约过程中逐步构建。

- 所有节点类继承自统一的 `ASTNode` 接口，常见的子类有：
  - `Program`, `ClassDecl`, `MethodDecl`, `VarDecl`
  - `IfStmt`, `WhileStmt`, `ReturnStmt`, `AssignStmt`
  - `BinaryExpr`, `UnaryExpr`, `CallExpr`, `ConstExpr`, `VarExpr`

示例：如下 FDMJ 程序段：

```
if (a > 0) {
  a = a + 1;
}
```

将被解析为如下 AST 结构（简化）：

```
IfStmt
├── condition: BinaryExpr (">")
│   ├── VarExpr("a")
│   └── ConstExpr(0)
└── then_body: AssignStmt
    ├── VarExpr("a")
    └── BinaryExpr("+")
        ├── VarExpr("a")
        └── ConstExpr(1)
```

2.4 错误处理机制

语法错误由 `yyerror()` 统一处理，并打印包含**行号与出错 token**的信息。例如：

```
Syntax error at line 4: unexpected token 'else'
```

通过在 `parser.yy` 中使用 `%locations` 和自定义位置字段，可以更精确地指出错误位置。错误发生后，parser 会尝试恢复并继续分析，以报告更多问题。

2.5 本阶段输出文件说明

执行 `make run` 后，将输出 `.2.ast` 文件，表示带有语义信息的抽象语法树。每个 AST 节点会打印其类型与子结构，并嵌入符号表引用信息，供后续类型检查与 IR 转换阶段使用。

例如，`test/hw1test1.fmj.2.ast` 文件可能输出：

```
Program
└── ClassDecl(Main)
    └── MethodDecl(main)
        ├── VarDecl(int a)
        └── IfStmt
            ├── BinaryExpr(>)
            │   ├── VarExpr(a)
            │   └── ConstExpr(0)
            └── AssignStmt
                ├── VarExpr(a)
                └── BinaryExpr(+)
                    ├── VarExpr(a)
                    └── ConstExpr(1)
```

---

3. 类型检查

3.1 名称映射构建（Symbol Tables）

在进行类型检查前，我们需要先收集所有变量、方法、类的定义信息，构建统一的**名称映射表结构（`Name_Maps`）**。这一过程由 `AST_Name_Map_Visitor` 在 `setnamemaps.cc` 中实现，并通过如下数据结构组织：

- `classes: set<string>`：记录所有类的名称。
- `methods: map<(class, method)>`：记录所有类中定义的方法。
- `classVar: map<(class, var), VarDecl*>`：类变量定义。
- `methodVar: map<(class, method, var), VarDecl*>`：方法局部变量。

- `methodFormal: map<(class, method, var), Formal*>`：方法形参。
- `methodFormalList: map<(class, method), vector<string>>`：形参名称列表。
- `methodType: map<(class, method), Type*>`：方法返回类型。
- `classHierarchy: map<child_class, parent_class>`：类的继承关系。

此外还实现了继承分析与方法签名检查，如 `inherit_var()`、`check_method_signature()` 等。

3.2 类型检查访问器（Semantic Analyzer）

类型检查过程由 `AST_Semant_Visitor` 实现，定义在 `semantanlyzer.cc` 中，其核心思路如下：

主要职责：

- 遍历 AST，收集每个节点的类型信息（封装为 `AST_Semant` 对象）；
- 检查变量是否重复声明；
- 检查方法定义是否重名；
- 检查表达式、语句的类型一致性；
- 检查赋值、函数调用、返回语句的合法性；
- 对错误信息给出明确提示（包含行列号、错误类型、建议说明等）。

类型信息封装结构：

```
class AST_Semant {
  enum Kind { Value, MethodName, ... };
  TypeKind type; // INT, CLASS, ARRAY
  variant<monostate, string, int> type_param; // 若为类则是类名，数组是维度
  bool is_lvalue;
};
```

每个 AST 节点的语义信息都会被存入一个 `semant_map` 中，供后续阶段使用。

3.3 类型检查的典型规则与流程

变量声明检查：

- 检查变量名是否重复定义。

- 检查变量名是否与参数冲突。
- 若继承了父类变量，警告是否覆盖（shadow）。

表达式检查：

以二元运算 `BinaryOp` 为例：

```
if (op == "+" || op == "-") {
  if (lhs_type != INT || rhs_type != INT) {
    // 错误：加减操作数必须为整型
  }
}
```

对于 `==` 或 `!=` 运算，则允许数组、类等类型进行比较，但要求同维或同名。

赋值语句检查：

- 左值必须是变量或数组元素（检查 `is_lvalue == true`）。
- 右值类型必须与左值一致，类类型需验证继承兼容性。

函数调用检查：

- 检查方法是否存在；
- 检查参数个数与类型是否匹配；
- 检查返回类型是否符合上下文预期。

返回语句检查：

- 方法应返回与声明类型一致的值；
- 类与数组类型返回时要求兼容；
- `void` 方法不应带返回值，反之亦然。

3.4 示例：类型不匹配错误的检测

以以下 FDMJ 程序为例：

```
int a;
a = "hello";
```

在 `Assign` 节点上，类型检查器会执行：

```
if (lhs_type != rhs_type) {
  cerr << "Error: Type mismatch in assignment" << endl;
}
```

实际输出：

```
Error at line 2: Type mismatch in assignment - cannot assign CLASS to INT
```

3.5 本阶段中间输出与调试信息

类型检查阶段不会生成单独文件，但所有类型错误与符号冲突都会通过 `stderr` 输出，内容结构为：

- 错误类型（Type mismatch, Duplicate declaration, etc.）
- 所在类与方法名
- 精确行号与列号
- 错误说明 + 建议修复信息

如：

```
Error at line 5, column 10: Variable 'x' in method 'main' is already declared
Error: Parameter 1 of method 'foo' in class 'B' has type 'int', but in parent class 'A' it has type 'boolean'
```

---

4. AST 到中间表示 IR+ 的转换

4.1 IR+ 表达形式概述

IR+ 表达形式为一组树状结构，其核心包括：

- `Stm`：语句类节点，如 `SEQ`, `LABEL`, `JUMP`, `CJUMP`, `MOVE` 等。
- `Exp`：表达式类节点，如 `BINOP`, `MEM`, `TEMP`, `CALL`, `CONST` 等。

节点种类参考了标准的 Tiger 编译器 IR 设计，适合进行 tiling 和 pattern matching。

4.2 IR 生成流程总览

在文件 `ast2tree.cc` 中，类 `AST2TreeVisitor` 实现了从 AST 节点到 IR 树的转换逻辑。

基本思路：

- 使用访问者模式（Visitor），为每种 AST 节点定义对应的 IR 转换规则。
- 使用 `TrExp` 类封装 IR+ 的三种表达：表达式（`Ex`）、语句（`Nx`）、有值条件表达式（`Cx`）。
- 每个 `visit()` 方法返回一个 `TrExp*` 对象，包含其对应的 IR+ 树节点结构。

TrExp 类型说明：

类型	含义
`Ex`	表达式类型，有返回值
`Nx`	单纯语句，无返回值
`Cx`	条件表达式，带跳转结构（布尔）

4.3 常见语法结构的转换逻辑

以下列出关键语句的 IR 构建方式。

变量定义与赋值

```
TrExp* AST2TreeVisitor::visit(AssignStmt* s)
```

- 目标变量地址转换为 `MEM`，`TEMP` 或 `ESEQ`。
- 源表达式转换为 `Ex`。
- 生成 `MOVE` 语句 IR：`MOVE(dst, src)`。

条件语句 `if`

```
TrExp* AST2TreeVisitor::visit(IfStmt* s)
```

- 条件为 `Cx`，表示跳转控制；
- 使用 `LABEL`、`SEQ` 和 `JUMP` 构建分支；
- 有无 `else` 分支分别对应两个构造模式。

循环语句 `while`

```
TrExp* AST2TreeVisitor::visit(WhileStmt* s)
```

- 构建 `test`, `body`, `done` 三个跳转点；
- 条件转换为 `Cx`，控制是否进入 `body`；
- IR 结构如下：

```
LABEL test
CJUMP cond → body / done
LABEL body
  body-stms
JUMP test
LABEL done
```

表达式语句（如 `a + b`）

```
TrExp* AST2TreeVisitor::visit(BinaryExpr* e)
```

- 转换两边为 `Ex`；
- 构建 `BINOP(op, lhs, rhs)` 表达式节点；
- 运算类型在 `semant_map` 中已确认为合法整型。

4.4 函数结构转换

函数体（即方法）在 `MethodDecl` 的转换中：

- 使用 `IR::Label` 指定方法标签；
- 每个方法的参数和局部变量均映射到 `TEMP` 临时变量；
- 方法返回值通过 `MOVE` 到指定位置（如 `$RV`）实现；
- 使用 `SEQ` 构造整个方法体为顺序结构。

4.5 示例：AST 到 IR+ 转换过程

以以下 FDMJ 程序片段为例：

```
int a;
a = 3 + 4;
```

构建的 IR+ 表达式如下：

```
SEQ(
  MOVE(TEMP a, CONST 3),
  MOVE(TEMP a, BINOP(PLUS, CONST 3, CONST 4))
)
```

如果嵌入于函数中，还会包括方法标签、返回、栈帧入口等结构。

4.6 中间输出文件

执行 `make run` 后，`.3.tree` 或 `.3.ir` 文件将包含每个函数转换后的 IR+ 表达结构。

示例输出：

```
LABEL main
MOVE TEMP a CONST 3
MOVE TEMP a BINOP(PLUS, CONST 3, CONST 4)
```

每一行表示一条中间语句，结构紧凑，易于用于后续 tiling 翻译。

---

5. 中间代码生成

5.1 四地址码（Quad Code）概述

每一条四地址码语句被表示为一种 `QuadKind` 的语句，形式为：

```
(op, dst, src1, src2) 或类似形式
```

系统支持的关键指令包括：

指令类型	含义
`MOVE`	赋值
`LOAD` / `STORE`	内存访问
`MOVE_BINOP`	二元操作符运算
`CALL` / `EXTCALL`	函数调用
`CJUMP` / `JUMP`	控制流跳转
`LABEL`	标签定义
`PHI`	SSA形式合流节点
`RETURN`	函数返回

5.2 转换器结构设计

`tree2quad.cc` 实现了一个 IR+ 到四地址码的转换器，主要结构如下：

- `class Tree2Quad`：核心转换器，接受 `tree::Program` 为输入。
- `visit()` 系列函数对不同的 IR 节点进行转换，递归构造四地址码结构。
- 中间变量统一为 `Temp` 表示，具备唯一编号及类型信息。
- 所有指令归纳为 `QuadStm` 子类，由 `QuadBlock` 组织为基本块集合，最终构成 `QuadFuncDecl` 和 `QuadProgram`。

5.3 转换流程细节

表达式转指令

```
QuadTerm* Tree2Quad::exp2term(tree::Exp* e)
```

- 对 `CONST`, `TEMP`, `MEM`, `CALL` 表达式生成对应 `QuadTerm`；
- 若为复杂表达式（如 `BINOP`），先转换为指令序列，并返回结果寄存器对应 `TEMP`。

MOVE 指令

```
case tree::MOVE:
```

- 判断目标为 `TEMP` / `MEM`；
- 若源表达式为复杂结构，先转换为中间寄存器，再 `MOVE`。

例：

```
a = b + c;
```

生成指令：

```
t1 <- b
t2 <- c
t3 <- PLUS, t1, t2
a <- t3
```

即：

```
MOVE t1 <- b
MOVE t2 <- c
MOVE_BINOP t3 <- (PLUS, t1, t2)
MOVE a <- t3
```

控制流与跳转

```
QuadCJump, QuadJump, QuadLabel
```

- 控制结构由 `tree::CJUMP` 构建；
- 真/假分支跳转至 `Label` 指定块。

生成示例：

```
CJUMP LT t1 t2 ? Ltrue : Lfalse
LABEL Ltrue
...
JUMP Lend
LABEL Lfalse
...
LABEL Lend
```

函数调用

- 内部调用（`CALL`）：生成 `QuadCall` 或 `QuadMoveCall`；

- 外部函数（如库函数）：使用 `EXTCALL`。

参数按顺序转换为 `QuadTerm`，保存在 `args` 向量中。

5.4 SSA 预处理支持（PHI 指令）

在后续 SSA 转换前，每个基本块的 `QuadBlock` 会根据 dominator tree 与变量活跃信息插入 `PHI` 节点。该阶段通过辅助工具插入形式为：

```
PHI t3 <- (t1, L1; t2, L2)
```

由 `QuadPhi` 类封装，其语义为：t3 根据前驱块选择不同来源的 t1/t2。

5.5 输出示例

输出的 `.4-xml.quad` 文件内容示例如下：

```
Function main(t1, t2) last_label=5 last_temp=10:
  Block: Entry Label: L0
    MOVE t3:INT <- t1:INT;
    MOVE_BINOP t4:INT <- (PLUS, t3:INT, t2:INT);
    RETURN t4:INT;
```

可视化结构中，每个函数包括入口标签、参数、四地址码指令列表等信息，便于后续构建 CFG 与 SSA 图。

---

6. SSA转换与数据流分析

6.1 总体流程

整个SSA转换过程由以下几个步骤组成：

1. **删除不可达基本块（deleteUnreachableBlocks）**
2. **插入Phi函数（placePhi）**
3. **变量重命名（renameVariables）**
4. **清除未使用Phi函数（cleanupUnusedPhi）**

这些步骤封装在 `quad2ssa` 函数中，对每个函数的 Quad 中间表示依次进行转换。

6.2 控制流分析与支配边界

我们使用 `ControlFlowInfo` 类构造控制流图（CFG）并计算每个基本块的：

- 支配树
- 支配边界（Dominance Frontier）
- 前驱与后继关系

这些信息是插入Phi函数和变量重命名的核心基础。

6.3 Phi函数插入

对于所有在多个基本块中有定义的变量 `t`，我们在其支配边界插入Phi函数。

示例（变量 `t4` 被定义在 `B1` 和 `B2` 中）：

```
if (...) { t4 = ...; } else { t4 = ...; }
// 在合流点 B3 插入：t4 = φ(t4@B1, t4@B2)
```

插入位置为基本块中Label后的第一条语句前。

6.4 SSA重命名机制

SSA变量版本通过以下方式生成：

```
newNum = origNum * 100 + version
```

每个原始临时变量 `Temp* t` 在每次新定义时创建新版本，并维护：

- `tempVersions`：所有版本的映射
- `currentVersion`：当前作用域中使用的版本
- `tempSsaIds`：SSA编号记录

对所有使用该变量的位置，包括表达式、函数调用参数、条件跳转、Phi函数参数等，都会替换为当前版本。

6.5 清除未使用Phi

最后阶段会清理所有未被使用的Phi函数，通过反复遍历 use-def 关系，标记必要的Phi，剔除冗余。

6.6 示例

以如下 FDMJ 代码为例：

```
int f(int x) {
    int y;
    if (x > 0) {
        y = x + 1;
    } else {
        y = x - 1;
    }
    return y;
}
```

在转换为Quad表示后，`y` 会在两个分支中定义，因此在合流点插入：

```
t200 = φ(t101@L1, t102@L2)
return t200
```

通过SSA形式，后续的寄存器分配能精确追踪变量活跃范围，减少冲突。

7.寄存器分配与数据流分析

7.1 模块概述

本阶段的主要任务包括两部分：

- **控制流分析与数据流分析**：基于中间代码基本块生成控制流图，进一步计算各类支配关系、支配边界、活跃变量集合等信息。
- **寄存器分配**：在数据流信息支持下，构建干涉图并进行图着色，为中间代码中的临时变量分配实际寄存器。

实现文件包括：

- `controlflowinfo.cc`：构造控制流图，计算支配、支配边界等
- `dataflowinfo.cc`：活跃变量分析，输出 live-in/live-out 信息
- `regalloc.cc`：使用图着色算法进行寄存器分配
- `prepareregalloc.cc`：准备变量重命名、寄存器使用分析

7.2 控制流图构造与分析

控制流分析由 `ControlFlowInfo` 类完成，步骤如下：

1. **基本块收集**：
   - `computeAllBlocks()`：收集函数中的所有基本块并编号，存储为 `allBlocks` 和 `labelToBlock`。
2. **控制流边计算**：
   - `computeSuccessors()`：为每个基本块计算后继集合；
   - `computePredecessors()`：计算前驱集合。
3. **支配关系分析**：
   - `computeDominators()`：使用迭代算法求出每个块的支配集合；
   - `computeImmediateDominator()`：计算直接支配者；
   - `computeDomTree()`：构造支配树；
   - `computeDominanceFrontiers()`：构造支配边界（φ函数插入依赖）。
4. **不可达块处理**：
   - `computeUnreachableBlocks()` + `eliminateUnreachableBlocks()`：移除 CFG 中不可达的基本块。
5. **汇总接口**：
   - `computeEverything()` 一次性执行全部上述过程。

7.3 活跃变量分析（Liveness Analysis）

`DataFlowInfo` 模块完成变量活跃信息的收集与分析：

1. **变量使用与定义集收集**：
   - `findAllVars()` 收集所有变量的 `use` 和 `def` 集合。
2. **活跃信息迭代分析**：
   - `computeLiveness()` 中通过迭代方式计算每条语句的 `live-in` 与 `live-out`：

     $in[s]=use[s]\cup(out[s]-def[s])$

     $out[s]=\cup_{s'\in succ(s)} in[s']$
   - 通过 `stmtToBlock`、`nextStmtInBlock` 映射连接语句级别的数据流依赖。
3. **分析结果打印**：
   - `printLiveness()` 输出每条语句的活跃变量集合，用于调试或展示。

7.4 干涉图构建与寄存器分配

`regalloc.cc` 负责从活跃信息出发，构建干涉图并分配寄存器：

1. **干涉图构建**：
   - `buildIg(func)` 在 `coloring.cc` 内部完成（调用未在此文件出现，但逻辑可推断）；
   - 每个变量对应图中一个节点，若两个变量在某语句 `live-out` 集中共存，则连边。
2. **图着色算法分配寄存器**：
   - `coloring(func, k)` 为函数中所有变量在 k 个寄存器下分配着色；
   - `coloring(prog, k)` 将所有函数统一处理，生成 XML 格式输出。
3. **输出结果**：
   - 若启用 `output_ig`，可打印干涉图结构以供调试；
   - 每个变量对应的颜色即寄存器编号，可用于最终代码生成。

---

8. 寄存器分配

本章介绍如何将 SSA 格式的中间表示进行变量着色（graph coloring）以实现寄存器分配。该部分构建于干涉图（interference graph）之上，采用了经典的图着色算法，并结合了 coalescing、simplify、freeze、spill 等多种策略，使得在可用寄存器数量有限的情况下，尽可能减少溢出。

8.1 干涉图与目标

我们使用 `InterferenceGraph` 建立变量之间的冲突关系（即不能被分配相同寄存器的变量对），并在该图的基础上进行寄存器着色。每个变量视为图中的一个节点，存在冲突关系的变量间连有边。

输入为SSA形式的Quad程序和参数 `k`，代表可用寄存器数（0-9），其中 r0-r8 可分配，r9-r10 用作溢出，r11-r15 保留特殊用途。

8.2 寄存器分配流程

我们实现的着色器 `Coloring` 包括以下阶段，每阶段循环执行直到无变化为止：

1. Simplify

从干涉图中移除所有度数小于 `k` 且不参与 move 指令的节点，并压入 `simplifiedNodes` 栈中，为后续选择着色提供顺序。

```
if (degree < k && !isMove(node))
    simplifiedNodes.push(node);
```

2. Coalesce

若两个变量之间存在 move 指令，并且它们之间不直接冲突，可尝试合并为一个变量以节省寄存器使用。

- 优先合并非机器寄存器
- 合并后更新图结构
- 每轮只合并一对，确保控制

```
if (!isAnEdge(u, v) && safeToCoalesce)
    merge u and v
```

3. Freeze

若某些节点度数 < k 且无法合并，则冻结该变量对应的 move 指令，使得它能被 simplify 考虑。

```
if (isMove(node) && degree < k)
    remove move pairs involving node
```

4. Spill

若以上都无法执行，则选择度数最大的节点进行“软溢出”，即从图中移除，标记为稍后尝试分配或 spill 到内存。

```
spillNode = argmax{ degree(node) }
simplifiedNodes.push(spillNode)
```

5. Select & Assign Colors

根据简化栈顺序回溯分配寄存器颜色，若某变量的所有邻居颜色已经用完，则视为实际溢出。

```
for color in [0, ..., k-1]:
    if color not used by neighbors:
        assign to current node
    else:
        mark as spilled
```

溢出变量将使用 `r9` 和 `r10` 存储，并通过栈进行访存管理。

8.3 色彩映射与检查

每个变量被映射到一个颜色值（即寄存器号），最终通过 `checkColoring()` 验证所有冲突边连接的节点是否使用不同颜色，未分配的变量是否确实已溢出。

若 `checkColoring()` 返回 false，表示存在冲突或遗漏，编译器将终止或打印错误。

8.4 示例：三变量冲突

考虑如下 SSA 表达式（简化）：

```
t1 = ...
t2 = ...
t3 = φ(t1, t2)
return t3
```

若 `t1`, `t2`, `t3` 均在同一作用域活跃，将建立如下冲突图：

```
t1 -- t3
t2 -- t3
```

假设 k=2，则不能对三者分配两个颜色，必有一个变量溢出。

8.5 输出结果

最终结果会输出到 XML 中，包含：

- 每个变量的寄存器号 `<Color node=X color=Y>`
- 所有溢出变量 `<Spill node=X>`
- 干涉图结构与简化顺序
- Coalesce 合并信息

---

9. RPi 汇编代码生成

9.1 汇编生成的整体流程

整个汇编生成由以下几个步骤构成：

1. **块串联与控制流规整化**（`trace()`）；
2. **四地址指令转汇编（每句转换）**；
3. **变量到寄存器的映射与临时变量处理**；

4. **栈帧生成：溢出变量与返回地址管理**；
5. **输出 `.s` 汇编文件或字符串。**

主入口为：

```
string quad2rpi(QuadProgram* quadProgram, ColorMap *cm);
```

或输出文件版本：

```
void quad2rpi(QuadProgram* quadProgram, ColorMap *cm, string filename);
```

9.2 控制流规整：`trace()` 函数

在转换前，所有基本块将被串联为一个连续的块以简化跳转指令处理：

- 收集所有 block 至 `allBlocks`；
- 按照程序执行顺序拼接 block 中的语句至 `sl`；
- 最终构成一个新的 QuadBlock 替代原有结构。

该阶段还分析了函数用到的寄存器与溢出变量数量，用于栈帧构造。

9.3 汇编转换的核心函数：`convertQuadStm`

每条四地址指令由 `convertQuadStm()` 转换为相应的 RPi 汇编片段，具体规则包括：

- **赋值**：

  ```
  t1 = t2 → MOV rX, rY
  ```

- **常数加载**：

  ```
  t1 = const → MOV rX, #imm8
  ```

- **二元操作**：

  ```
  t1 = t2 + t3 → ADD rX, rY, rZ
  ```

- **条件跳转**：

```
if t1 == t2 goto L → CMP rX, rY; BEQ L
```

- **函数调用**：
  - 参数最多 3 个，使用 r0–r2；
  - 返回值在 r0；
  - 使用 `BL` 进行调用。
- **返回语句**：

```
return t1 → MOV r0, rX; BX lr
```

9.4 栈帧组织

每个函数按以下方式组织其栈帧：

- **保存返回地址与旧帧指针**；
- **分配溢出槽空间**（通过 `spill_slot_num`）；
- **按寄存器调用约定保存/恢复 callee-save 寄存器**；
- **从 `ColorMap` 中读取变量是否分配寄存器或栈槽。**

示例栈帧结构如下：

```
| ...              |
| spill slot[n]    |
| ...              |
| saved r4-r8      |
| saved fp         |
| return address   | ← sp before call
```

9.5 溢出变量的加载与存储

对于未被分配寄存器的变量（即“溢出”变量），通过以下函数处理其访存：

```
string loadSpilledTemp(int temp_num, Color* color, int reg_num, int indent);
string storeSpilledTemp(int temp_num, Color* color, int reg_num, int indent);
```

- 加载时将变量从其槽位加载至临时寄存器；
- 存储时反之；

- `r9` 与 `r10` 作为溢出操作的辅助寄存器。

9.6 示例输出

以如下 Quad 程序为例：

```
t1 = 1
t2 = 2
t3 = t1 + t2
return t3
```

汇编输出：

```
MOV r1, #1
MOV r2, #2
ADD r3, r1, r2
MOV r0, r3
BX lr
```

其中 r1, r2, r3 的编号由 `ColorMap` 指定。若 t3 被溢出，则需从栈读取至寄存器后再赋值给 r0。

9.7 编译选项与输出文件说明

最终 `.s` 文件可通过调用：

```
make build
make run
```

自动生成并执行。每个 `.fmj` 源文件将输出：

- `.7-rpi.s`：对应的 RPi 汇编文件；
- `.7-rpi.txt`：可读格式打印内容。

---

10. 总结与使用说明

10.1 项目总结

本学期我实现了一个完整的编译器，目标语言为教学语言 FDMJ（一个类 Java 的语言），并最终生成可在 RPi 架构模拟器（如 QEMU）上运行的 ARM 汇编代码。

整个编译器流程如下：

```
[ 源代码 ] → 词法分析 → 语法分析 → AST → 类型检查 → IR+ → Quad → SSA → 活跃变量分析
→ 寄存器分配 → RPi 汇编
```

本项目涵盖了所有课程涉及的编译器模块，并在每个阶段插入了格式化的中间结果输出，便于调试与验证。通过这个项目，我深入理解了编译器的结构、语言抽象的转换流程以及数据流分析在实际代码优化中的作用。

10.2 编译器结构与模块说明

完整的编译器项目结构如下：

```
FudanCompilerH2025/
├── include/              # 各模块头文件（frontend, ast, ir, quad, etc.）
├── lib/                  # 源码实现，按模块分为 lexer, parser, ast, ir, quad 等
├── tools/main/           # 编译主程序入口
├── test/                 # FDMJ 源代码样例 (*.fmj) 与输出文件 (*.ast, *.quad,
*.ssa, *.s)
├── vendor/               # 第三方库（如 tinyxml2）
├── CMakeLists.txt        # 构建配置文件
├── Makefile              # 构建与运行入口
└── README.md             # 使用说明文档
```

主要模块功能：

模块	功能说明
`frontend/`	词法与语法分析（使用 Flex/Bison）
`ast/`	抽象语法树构建、类型检查
`ir/`	AST → IR+ 转换
`quad/`	IR+ → Quad（四地址码）转换与 SSA 构建
`quadflow/`	控制流图、数据流分析与寄存器分配
`rpi/`	Quad → ARM 汇编生成，栈帧与溢出变量管理

10.3 使用说明（User Manual）

环境依赖：

- 64-bit Ubuntu 20.04+
- 安装工具：`flex`, `bison`, `make`, `cmake`, `qemu-arm`, `python3`

编译器构建与运行：

构建编译器：

```
make build
```

每个 `test/*.fmj` 文件将依次生成以下输出文件：

输出文件名	含义
`*.1-fmj.txt`	源码格式化后输出
`*.2-ast.xml`	带类型信息的 AST
`*.3-ir.xml`	IR+ 表示
`*.4-quad.xml`	四地址码表示
`*.4-ssa.quad`	SSA 转换后的四地址码
`*.5-color.xml`	变量着色与干涉图输出
`*.7-rpi.s`	RPi 汇编代码输出
`*.7-rpi.txt`	可读汇编文本

执行编译后的汇编程序（使用 QEMU）：

```
make run
```

10.4 示例程序使用

建议使用如下示例程序进行逐步验证：

```c
int main() {
    int a = 1;
    int b = 2;
    int c = a + b;
    return c;
}
```

执行后，可检查 `test/main.7-rpi.txt`，确认生成的汇编逻辑正确无误。