# **中间代码优化器 (Optimizer) 设计与实现文档**

## **1. 概述**

Optimizer 是一个为编译器设计的关键后端模块，其主要任务是在目标代码生成之前，对前端产生的原始四元式（中间代码）进行改进，以生成更高效、更精简的代码。

该优化器采用了经典的基本块（Basic Block）**和**控制流图（Control Flow Graph, CFG）作为分析基础，并在此之上实现了多种局部优化（Local Optimization）技术。

### **1.1. 主要功能与优化技术**

* **基本块划分**: 将线性的四元式序列划分为多个最大化的、无分支的基本块。
* **控制流图构建**: 分析基本块之间的跳转关系，构建有向图来表示程序的控制流。
* **数据流分析**:
  + **活跃变量分析 (Liveness Analysis)**: 精确计算在程序的每个点上，哪些变量的未来值是必需的。这是实现死代码消除的关键。
* **局部优化 (基于DAG)**: 对每个基本块内部进行深度优化。
  + **有向无环图 (DAG)**: 将基本块内的表达式构建为DAG，直观地表示计算和数据依赖。
  + **公共子表达式消除 (Common Subexpression Elimination, CSE)**: 复用已经计算过的表达式结果，避免重复计算。
  + **常量折叠 (Constant Folding)**: 在编译期直接计算出常量表达式的结果。
  + **拷贝传播 (Copy Propagation)**: 通过重命名和别名机制，消除不必要的赋值语句。
  + **死代码消除 (Dead Code Elimination)**: 基于活跃性分析的结果，移除那些计算结果从未被使用的无效指令。
* **最终清理**: 移除在优化过程中产生的、未被任何跳转指令使用的“悬空”标签。

## **2. 核心概念与数据结构**

### **2.1. BasicBlock (基本块)**

基本块是优化的最小单元，它是一段满足以下条件的连续四元式序列：

1. 只有一个入口点（序列的第一条指令）。
2. 只有一个出口点（序列的最后一条指令）。

这意味着代码只能从块的开头进入，并且一旦进入，就会按顺序执行完所有指令，直到块的末尾才可能发生跳转。

struct BasicBlock {  
 int id;  
 std::vector<Quadruple> quads; // 块内的四元式  
 std::set<std::string> use; // 在本块中被使用前未被定义的变量  
 std::set<std::string> def; // 在本块中被赋值的变量  
 std::set<std::string> live\_in; // 入口活跃变量  
 std::set<std::string> live\_out; // 出口活跃变量  
 std::vector<int> successors; // 后继块ID列表  
 std::vector<int> predecessors; // 前驱块ID列表  
 // ...  
};

### **2.2. DagNode (DAG节点)**

DAG节点用于表示基本块内的表达式或值。

* **叶子节点 (op == "leaf")**: 代表一个变量或一个常量。
* **内部节点 (op为运算符)**: 代表一个计算操作，其子节点是该操作的操作数。

一个节点的 labels 向量存储了所有当前值为该节点所代表的值的变量名。

struct DagNode {  
 int id;  
 std::string op;  
 DagNode \*left = nullptr, \*right = nullptr;  
 std::vector<std::string> labels;  
 // ...  
};

例如，对于 a = b + c 和 d = b + c，b 和 c 是叶子节点，+ 是一个内部节点。a 和 d 最终都会成为 + 节点的标签，这正是公共子表达式消除的基础。

## **3. 优化流程 (Pipeline)**

Optimizer::optimize() 函数是整个优化的入口，它严格按照以下顺序执行各个阶段：

### **第1步: 划分基本块 (divide\_into\_basic\_blocks)**

此阶段的目标是将一维的四元式流切分为多个基本块。算法如下：

1. **识别入口指令 (Leaders)**：
   * 第一条四元式是入口指令。
   * 任何跳转指令（JUMP, JUMPF）的目标指令是入口指令。
   * 紧跟在任何跳转或返回指令之后的指令是入口指令。
2. **生成基本块**: 从一个入口指令开始，到下一个入口指令（或代码结束）之前的所有指令，构成一个基本块。

### **第2步: 构建CFG与计算Use/Def集 (build\_cfg\_and\_compute\_use\_def)**

此阶段在基本块的基础上建立它们之间的联系。

1. **构建CFG**: 遍历每个基本块，检查其最后一条指令：
   * 如果是 JUMP L1，则目标标签 L1 所在的基本块是当前块的后继。
   * 如果是 JUMPF cond, L1，则目标标签 L1 所在块和紧邻的下一个块都是当前块的后继。
   * 否则，紧邻的下一个块是后继。
2. **计算 use 和 def 集合**:
   * **def集**: 一个块中所有被赋值的变量的集合。
   * **use集**: 一个块中所有在被赋值前就被使用的变量的集合。

### **第3步: 活跃变量分析 (run\_liveness\_analysis)**

这是一个经典的反向数据流分析迭代算法，用于计算每个块的 live\_in 和 live\_out 集合。

* **live\_out[B]**: 在基本块 B 的出口处，后续可能被使用的变量集合。它的值等于其所有**后继块** live\_in 集合的并集。
* live\_in[B]: 在基本块 B 的入口处，必须是活跃的变量集合。它的计算公式为：  
  live\_in[B] = use[B] ∪ (live\_out[B] - def[B])

算法会持续迭代，直到所有块的 live\_in 和 live\_out 集合不再变化为止。

### **第4步: 块内优化 (optimize\_block)**

这是优化的核心，它对每个基本块独立进行。

1. **构建DAG**:
   * 遍历块内的每一条四元式。
   * 对于表达式（如 a = b + c），查找或创建代表操作数 b 和 c 的叶子节点。
   * 检查是否存在一个具有相同操作符（+）和相同子节点（b和c的节点）的内部节点。
     + 若存在（**公共子表达式**），则将变量 a 添加到该节点的 labels 列表。
     + 若不存在，则创建一个新的内部节点，并将 a 作为其标签。
   * 对于赋值（如 a = b），找到 b 对应的节点，并将 a 也添加到该节点的 labels 列表中。
   * **常量折叠**: 在构建过程中，如果发现一个表达式的左右操作数均为常量，则立即计算其值，并创建一个代表该结果常量的叶子节点来取代原有的表达式节点。
   * **副作用指令**: CALL, PRINT, STORE\_AT 以及对**全局变量**的赋值等指令被视为具有副作用，它们被从DAG构建流程中分离出来，确保其执行。
2. **识别必需节点**:
   * 一个节点的计算结果是“必需的”，当且仅当：
     + 它的某个标签是该块的出口活跃变量（在 live\_out 集合中）。
     + 它的计算结果被一个具有副作用的指令所使用（例如 PRINT a 中的 a）。
   * 从这些“根”节点开始，通过反向追溯其所有依赖的子节点，就能标记出DAG中所有必需的节点。
3. **从DAG生成代码**:
   * 采用**后序遍历**的方式访问所有DAG节点。
   * **死代码消除**: 如果一个内部节点**未被标记为必需的**，则跳过它，不为它生成任何代码。
   * **代码生成**: 如果一个节点是必需的，则为其生成对应的计算四元式。
   * 最后，将在第1步中分离出来的副作用指令附加到优化后代码的末尾。

### **第5步: 清理未使用标签**

在优化过程结束后，一些原本被跳转指令引用的标签可能因为分支被优化掉而变得不再需要。最后一步会遍历所有优化后的四元式，移除所有未被引用的 (LABEL, ...) 指令。

## **4. 如何使用**

// 1. 已有一个包含原始四元式的 vector<Quadruple> quads  
// 2. 已有一个构建好的符号表 SymbolTable& st  
  
// 创建优化器实例  
Optimizer optimizer(quads, st);  
  
// 执行优化  
vector<Quadruple> optimized\_quads = optimizer.optimize();  
  
// optimized\_quads 即为最终优化后的四元式序列