编译器设计文档

一、参考编译器介绍（Pascal 编译器）

1.1 Pascal 编译器概述

Pascal 编译器是一个经典的教学用编译器，其设计遵循传统的编译原理。典型的 Pascal 编译器（如 Free Pascal）采用以下架构：

1.2 总体结构

- **词法分析器（Scanner）**：将源代码分解为 Token 流
- **语法分析器（Parser）**：构建抽象语法树（AST）
- **语义分析器（Semantic Analyzer）**：类型检查、符号表管理
- **中间代码生成器**：生成中间表示（通常是三地址码或 P-Code）
- **代码优化器**：进行各种优化 Pass
- **代码生成器**：生成目标机器代码

1.3 接口设计

Pascal 编译器通常采用模块化设计：

- 各阶段通过明确的数据结构（Token 流、AST、符号表）进行通信
- 使用错误处理机制统一管理编译错误
- 提供灵活的配置选项控制编译过程

1.4 文件组织

```
src/
├── scanner/        # 词法分析
├── parser/         # 语法分析
├── semantic/       # 语义分析
├── codegen/        # 代码生成
├── optimize/       # 优化模块
└── utils/          # 工具类
```

二、编译器总体设计

2.1 项目概述

本编译器是一个面向类 C 语言（SysY 语言）的完整编译系统，使用 Java 实现，能够将源代码编译为 MIPS 汇编代码。编译器采用现代化的 LLVM IR 风格中间表示，并实现了多种代码优化技术。

2.2 总体结构

编译器采用经典的多遍扫描架构，分为以下几个主要阶段：

```
源代码 → 词法分析 → Token 流 → 语法分析 → AST
     → 语义分析 → IR 生成 → 优化 Pass → MIPS 代码生成 → 目标代码
```

2.2.1 编译流程（Compiler.java:14）

```
public static void main(String[] args) {
    1. Config.init()                    // 配置初始化
    2. Lexer.analyze()                  // 词法分析
    3. Parser.analyze()                 // 语法分析
    4. ErrorHandler.compUnitError()     // 语义分析+错误检查
    5. LLVMGenerator.visitCompUnit()    // IR 生成
    6. PassModule.runIRPasses()         // 优化 Pass
    7. MipsGenModule.genMips()          // MIPS 代码生成
}
```

2.3 接口设计

2.3.1 单例模式

所有主要模块均采用单例模式，保证全局唯一实例：

- `Lexer.getInstance()`
- `Parser.getInstance()`
- `ErrorHandler.getInstance()`
- `LLVMGenerator.getInstance()`
- `IRModule.getInstance()`
- `PassModule.getInstance()`
- `MipsGenModule.getInstance()`

2.3.2 模块间通信

- **Lexer → Parser**：通过 `List<Token>` 传递 Token 流
- **Parser → ErrorHandler**：通过 `CompUnitNode`（AST 根节点）传递语法树
- **Parser → LLVMGenerator**：通过访问者模式遍历 AST
- **LLVMGenerator → PassModule**：通过 `IRModule` 共享 IR 数据结构
- **PassModule → Backend**：优化后的 `IRModule` 传递给后端

2.3.3 数据结构接口

```
// Token 结构 (token/Token.java)
class Token {
    TokenType type;       // Token 类型
    int lineNumber;       // 行号
    String content;       // 词素内容
}

// AST 节点接口
interface ASTNode {
```

```
    void print();          // 打印语法树
    NodeType getType();    // 获取节点类型
}

// IR Value 基类 (ir/values/Value.java)
class Value {
    Type type;             // 类型信息
    String name;           // 名称
    List<Use> usesList;    // 使用链
}
```

2.4 文件组织

```
me2/
├── Compiler.java                    # 主编译器入口
├── config/
│   └── Config.java                  # 编译配置管理
├── frontend/
│   ├── Lexer.java                   # 词法分析器
│   └── Parser.java                  # 语法分析器
├── token/
│   ├── Token.java                   # Token 定义
│   └── TokenType.java               # Token 类型枚举
├── node/                            # AST 节点定义
│   ├── CompUnitNode.java            # 编译单元节点
│   ├── FuncDefNode.java             # 函数定义节点
│   ├── StmtNode.java                # 语句节点
│   └── ...                          # 其他节点类型
├── smybol/                          # 符号表系统
│   ├── Symbol.java                  # 符号基类
│   ├── FuncSymbol.java              # 函数符号
│   └── ArraySymbol.java             # 数组符号
├── error/
│   ├── Error.java                   # 错误类
│   ├── ErrorType.java               # 错误类型
│   └── ErrorHandler.java            # 错误处理+语义分析
├── ir/                              # 中间代码表示
│   ├── IRModule.java                # IR 模块（全局管理）
│   ├── LLVMGenerator.java           # IR 生成器
│   ├── types/                       # 类型系统
│   │   ├── Type.java
│   │   ├── IntegerType.java
│   │   ├── ArrayType.java
│   │   └── ...
│   ├── values/                      # Value 类型
│   │   ├── Value.java               # Value 基类
│   │   ├── Function.java            # 函数
│   │   ├── BasicBlock.java          # 基本块
│   │   ├── GlobalVar.java           # 全局变量
│   │   ├── BuildFactory.java        # IR 构建工厂
│   │   └── instructions/            # 指令类型
│   │       ├── Instruction.java
```

```
│   │       ├── BinaryInst.java      # 二元运算
│   │       ├── mem/                 # 内存操作
│   │       │   ├── AllocaInst.java
│   │       │   ├── LoadInst.java
│   │       │   └── StoreInst.java
│   │       └── terminator/          # 终止指令
│   │           ├── BrInst.java
│   │           ├── RetInst.java
│   │           └── CallInst.java
│   └── analysis/                    # IR 分析
│       ├── DomAnalysis.java         # 支配树分析
│       ├── LoopInfo.java            # 循环信息
│       └── AliasAnalysis.java       # 别名分析
├── pass/                            # 优化 Pass
│   ├── Pass.java                    # Pass 接口
│   ├── PassModule.java              # Pass 管理器
│   └── ir/
│       ├── GVNGCM.java              # 全局值编号+代码移动
│       ├── DeadCodeElimination.java # 死代码消除
│       ├── BranchOptimization.java  # 分支优化
│       └── InterProceduralAnalysis.java # 过程间分析
├── backend/
│   └── MipsGenModule.java           # MIPS 代码生成
└── utils/                           # 工具类
    ├── IOUtils.java                 # IO 工具
    ├── IList.java                   # 侵入式链表
    ├── Pair.java                    # 二元组
    └── Triple.java                  # 三元组
```

2.5 设计特点

1. **模块化设计**：各阶段职责清晰，接口明确
2. **单例模式**：保证全局状态一致性
3. **访问者模式**：AST 遍历和 IR 生成
4. **SSA 形式**：中间代码采用 SSA（Static Single Assignment）形式
5. **Use-Def 链**：完善的使用-定义链管理
6. **侵入式数据结构**：使用侵入式链表提高效率
7. **可配置性**：通过 Config 类控制各阶段输出

三、词法分析设计

3.1 编码前的设计

3.1.1 设计目标

- 将源代码文本转换为 Token 流
- 识别关键字、标识符、常量、运算符、分隔符
- 处理注释（单行 `//` 和多行 `/* */`）
- 记录行号信息用于错误报告
- 检测词法错误（如不完整的 `&` 和 `|`）

3.1.2 Token 类型设计

定义 `TokenType` 枚举包含：

- **关键字**：`MAINTK`, `CONSTTK`, `INTTK`, `VOIDTK`, `IFTK`, `ELSETK`, `FORTK`, `BREAKTK`, `CONTINUETK`, `RETURNTK`, `PRINTFTK`, `STATICTK`
- **标识符**：`IDENFR`
- **常量**：`INTCON`（整数），`STRCON`（字符串）
- **运算符**：`PLUS`, `MINU`, `MULT`, `DIV`, `MOD`, `LSS`, `LEQ`, `GRE`, `GEQ`, `EQL`, `NEQ`, `ASSIGN`, `NOT`, `AND`, `OR`
- **分隔符**：`SEMICN`, `COMMA`, `LPARENT`, `RPARENT`, `LBRACK`, `RBRACK`, `LBRACE`, `RBRACE`

3.1.3 算法设计

采用**有限状态自动机（DFA）**方法：

- 使用单个主循环扫描字符
- 根据当前字符类型进入不同处理分支
- 使用前向查看（lookahead）识别双字符运算符（如 `==`, `!=`, `<=`, `>=`, `&&`, `||`）

3.1.4 数据结构设计

```
class Lexer {
    List<Token> tokens;                    // Token 列表
    Map<String, TokenType> keywords;       // 关键字表

    void analyze(String content);          // 主分析函数
    void printLexAns();                    // 输出 Token 流
}
```

3.2 编码完成后的修改

3.2.1 实现细节（frontend/Lexer.java）

- **单例模式**：使用 `getInstance()` 获取唯一实例
- **关键字识别**：使用 `HashMap` 存储关键字，通过 `getOrDefault()` 区分关键字和标识符
- **行号跟踪**：遇到 `\n` 时自动递增 `lineNumber`

3.2.2 错误处理增强

在编码过程中添加了对以下词法错误的检测：

```
// 不完整的 & 运算符 (frontend/Lexer.java:95-102)
if (c == '&') {
    if (next == '&') {
        tokens.add(new Token(TokenType.AND, lineNumber, "&&"));
    } else {
        tokens.add(new Token(TokenType.AND, lineNumber, "&&"));
        ErrorHandler.getInstance().addError(new Error(lineNumber, ErrorType.a));
    }
}
```

3.2.3 注释处理优化

```
// 单行注释 (frontend/Lexer.java:118-121)
if (next == '/') {
    int j = content.indexOf('\n', i + 2);
    if (j == -1) j = contentLength;  // 处理文件末尾没有换行的情况
    i = j - 1;
}

// 多行注释 (frontend/Lexer.java:122-130)
if (next == '*') {
    for (int j = i + 2; j < contentLength; j++) {
        if (e == '\n') lineNumber++;  // 注释中的换行也要计数
        if (e == '*' && content.charAt(j + 1) == '/') {
            i = j + 1;
            break;
        }
    }
}
```

3.2.4 字符串处理

```
// 字符串常量识别 (frontend/Lexer.java:76-88)
if (c == '\"') {
    String s = "\"";
    for (int j = i + 1; j < contentLength; j++) {
        if (content.charAt(j) != '\"') {
            s += content.charAt(j);
        } else {
            s += "\"";
            i = j;
            break;
        }
    }
    tokens.add(new Token(TokenType.STRCON, lineNumber, s));
}
```

3.2.5 性能考虑

- 使用 `StringBuilder` 替代字符串拼接可以提高性能（未来优化方向）
- 一次性读入所有源代码，避免频繁 IO

3.2.6 与 Parser 的接口

```
// Parser 获取 Token 流 (frontend/Parser.java:25-28)
public void setTokens(List<Token> tokens) {
    this.tokens = tokens;
    now = tokens.get(index);  // 初始化当前 Token
}
```

四、语法分析设计

4.1 编码前的设计

4.1.1 设计目标

- 将 Token 流转换为抽象语法树（AST）
- 检测语法错误
- 支持错误恢复（分号、括号缺失）
- 实现递归下降分析

4.1.2 文法设计

基于 SysY 语言文法，采用 LL(1) 文法：

```
CompUnit      → {Decl} {FuncDef} MainFuncDef
Decl          → ConstDecl | VarDecl
ConstDecl     → 'const' BType ConstDef {',' ConstDef} ';'
VarDecl       → ['static'] BType VarDef {',' VarDef} ';'
FuncDef       → FuncType Ident '(' [FuncFParams] ')' Block
MainFuncDef   → 'int' 'main' '(' ')' Block
Block         → '{' {BlockItem} '}'
Stmt          → LVal '=' Exp ';'
              | [Exp] ';'
              | Block
              | 'if' '(' Cond ')' Stmt ['else' Stmt]
              | 'for' '(' [ForStmt] ';' [Cond] ';' [ForStmt] ')' Stmt
              | 'break' ';' | 'continue' ';'
              | 'return' [Exp] ';'
              | 'printf' '(' StringConst {',' Exp} ')' ';'
Exp           → AddExp
AddExp        → MulExp {('+' | '−') MulExp}
MulExp        → UnaryExp {('*' | '/' | '%') UnaryExp}
UnaryExp      → PrimaryExp
              | Ident '(' [FuncRParams] ')'
              | UnaryOp UnaryExp
Cond          → LOrExp
LOrExp        → LAndExp {'||' LAndExp}
LAndExp       → EqExp {'&&' EqExp}
```

4.1.3 AST 节点设计

为每个非终结符设计对应的 AST 节点类：

- 节点类命名规则：`<NonTerminal>Node`（如 `CompUnitNode`，`StmtNode`）
- 每个节点包含：
  - 对应的子节点（非终结符）
  - Token 信息（终结符）
  - `print()` 方法用于输出语法树

4.1.4 算法设计

采用**递归下降分析**：

- 每个非终结符对应一个解析函数
- 使用 `match()` 函数匹配终结符
- 使用前向查看解决冲突（如区分函数调用和变量引用）

4.2 编码完成后的修改

4.2.1 核心解析机制（frontend/Parser.java）

Token 匹配机制：

```
// match 函数实现 (frontend/Parser.java:73-96)
private Token match(TokenType tokenType) {
    if (now.getType() == tokenType) {
        Token tmp = now;
        if (index < tokens.size() - 1) {
            now = tokens.get(++index);  // 前进到下一个 Token
        }
        return tmp;
    } else if (tokenType == TokenType.SEMICN) {
        // 错误恢复：合成缺失的分号
        ErrorHandler.getInstance().addError(new Error(
            tokens.get(index - 1).getLineNumber(), ErrorType.i));
        return new Token(TokenType.SEMICN,
            tokens.get(index - 1).getLineNumber(), ";");
    }
    // ... 类似处理右括号、右方括号
}
```

4.2.2 左递归消除

原始文法中的左递归需要改写：

改写前（左递归）：

```
AddExp → AddExp ('+' | '−') MulExp | MulExp
```

改写后（frontend/Parser.java:532-552）：

```
private AddExpNode AddExp() {
    // 等价于：AddExp → MulExp {('+' | '−') MulExp}
    AddExpNode left = new AddExpNode(null, MulExp(), null);
    while (now.getType() == TokenType.PLUS || now.getType() == TokenType.MINU) {
        Token ope = match(now.getType());
        MulExpNode right = MulExp();
        left = new AddExpNode(left, right, ope);  // 构建左结合的树
    }
    return left;
}
```

类似处理应用于：

- `MulExp` (frontend/Parser.java:508-530)
- `RelExp` (frontend/Parser.java:559-585)
- `EqExp` (frontend/Parser.java:587-607)
- `LAndExp` (frontend/Parser.java:609-626)
- `LOrExp` (frontend/Parser.java:628-637)

4.2.3 前向查看优化

区分函数定义与变量声明（frontend/Parser.java:102-110）：

```
// CompUnit 中区分 Decl 和 FuncDef
while (tokens.get(index + 2).getType() != TokenType.LPARENT) {
    declNodes.add(Decl());  // 第三个 Token 不是 '(' 则为变量声明
}
while (tokens.get(index + 1).getType() != TokenType.MAINTK) {
    funcDefNodes.add(FuncDef());  // 第二个 Token 不是 'main' 则为函数定义
}
```

区分赋值语句与表达式语句（frontend/Parser.java:448-475）：

```
// Stmt 解析中扫描到分号前寻找 '='
boolean containsAssign = false;
for (int i = index; i < tokens.size(); i++) {
    if (tokens.get(i).getType() == TokenType.ASSIGN) {
        containsAssign = true;
    }
    if (tokens.get(i).getType() == TokenType.SEMICN) {
        break;
    }
}
if (containsAssign) {
    // LVal '=' Exp ';'
    return new StmtNode(StmtNode.StmtType.LValAssignExp, ...);
} else {
    // [Exp] ';'
    return new StmtNode(StmtNode.StmtType.Exp, ...);
}
```

区分函数调用与变量引用（frontend/Parser.java:703-722）：

```
private UnaryExpNode UnaryExp() {
    if (now.getType() == TokenType.IDENFR &&
        tokens.get(index + 1).getType() == TokenType.LPARENT) {
        // 函数调用：Ident '(' [FuncRParams] ')'
        return new UnaryExpNode(ident, leftParen, funcRParams, rightParen);
    } else if (now.getType() == TokenType.PLUS ||
               now.getType() == TokenType.MINU ||
               now.getType() == TokenType.NOT) {
        // 一元运算：UnaryOp UnaryExp
```

```
            return new UnaryExpNode(unaryOp, unaryExp);
        } else {
            // 基本表达式：PrimaryExp
            return new UnaryExpNode(primaryExp);
        }
    }
```

4.2.4 错误恢复机制

实现了三种常见语法错误的自动恢复：

1. **缺失分号（错误类型 i）**（frontend/Parser.java:80-84）
2. **缺失右小括号（错误类型 j）**（frontend/Parser.java:85-88）
3. **缺失右中括号（错误类型 k）**（frontend/Parser.java:89-92）

错误恢复策略：

- 在期望位置"合成"缺失的 Token
- 记录错误但继续分析
- 不前进 `index`，避免跳过后续 Token

4.2.5 AST 设计优化

语句节点的多态设计（node/StmtNode.java）：

```
class StmtNode {
    enum StmtType {
        LValAssignExp,  // 赋值语句
        Exp,            // 表达式语句
        Block,          // 块语句
        If,             // if 语句
        For,            // for 循环
        Break,          // break
        Continue,       // continue
        Return,         // return
        Printf          // printf
    }

    StmtType type;
    // 根据不同类型包含不同字段
}
```

这种设计使得：

- 类型安全：通过枚举区分不同语句类型
- 灵活性：一个类处理所有语句类型
- 便于后续语义分析和 IR 生成

4.2.6 CompUnit 解析策略

```
private CompUnitNode CompUnit() {

    // CompUnit → {Decl} {FuncDef} MainFuncDef
```

```
    List<DeclNode> declNodes = new ArrayList<>();
    List<FuncDefNode> funcDefNodes = new ArrayList<>();
    MainFuncDefNode mainFuncDefNode = null;

    // 阶段1：解析全局声明（通过前向查看区分）
    while (tokens.get(index + 2).getType() != TokenType.LPARENT) {
        declNodes.add(Decl());
    }

    // 阶段2：解析函数定义
    while (tokens.get(index + 1).getType() != TokenType.MAINTK) {
        funcDefNodes.add(FuncDef());
    }

    // 阶段3：解析 main 函数
    mainFuncDefNode = mainFuncDef();

    return new CompUnitNode(declNodes, funcDefNodes, mainFuncDefNode);
}
```

4.2.7 For 循环特殊处理

```
// ForStmt → LVal '=' Exp { ',' LVal '=' Exp }
// 支持多个赋值：for (i = 0, j = 0; i < 10; i = i + 1, j = j + 1)
private ForStmtNode ForStmt() {
    List<LValNode> lValNodes = new ArrayList<>();
    List<Token> equals = new ArrayList<>();
    List<ExpNode> expNodes = new ArrayList<>();
    List<Token> commas = new ArrayList<>();

    // 第一个赋值
    lValNodes.add(LVal());
    equals.add(match(TokenType.ASSIGN));
    expNodes.add(Exp());

    // 后续赋值
    while (now.getType() == TokenType.COMMA) {
        commas.add(match(TokenType.COMMA));
        lValNodes.add(LVal());
        equals.add(match(TokenType.ASSIGN));
        expNodes.add(Exp());
    }

    return new ForStmtNode(lValNodes, equals, expNodes, commas);
}
```

五、语义分析设计

5.1 编码前的设计

5.1.1 设计目标

- **符号表管理**：维护标识符的作用域和属性
- **类型检查**：检查表达式和语句的类型正确性
- **语义错误检测**：识别各种语义错误
- **常量计算**：计算编译期常量表达式

5.1.2 符号表设计

采用**分层符号表**结构：

- 使用栈式符号表管理作用域嵌套
- 每个作用域维护一个 `Map<String, Symbol>`
- 支持变量、常量、函数、数组的符号记录

符号类型：

```
class Symbol {
    String name;          // 标识符名称
    String symbolType;    // 符号类型 ("var", "const", "func", "array")
}

class FuncSymbol extends Symbol {
    FuncType returnType;          // 返回值类型
    List<FuncParam> params;       // 参数列表
}

class ArraySymbol extends Symbol {
    int dimension;                // 维度
    List<Integer> dims;           // 各维大小
}
```

5.1.3 需要检测的语义错误

根据编译要求，需要检测以下错误类型：

错误代码	错误类型	说明
a	非法符号	格式字符串中出现非法字符
b	名字重定义	同一作用域内重复定义标识符
c	未定义的名字	使用未声明的标识符
d	函数参数个数不匹配	函数调用参数个数错误
e	函数参数类型不匹配	函数调用参数类型错误
f	无返回值的函数存在不匹配的 return 语句	void 函数有返回值
g	有返回值的函数缺少 return 语句	int 函数缺少 return
h	不能改变常量的值	对 const 变量赋值
i	缺少分号	（词法/语法层面已处理）
j	缺少右小括号	（词法/语法层面已处理）
k	缺少右中括号	（词法/语法层面已处理）
l	printf 中格式字符与表达式个数不匹配	%d 数量与参数不符
m	在非循环块中使用 break/continue	break/continue 不在循环内

5.1.4 算法设计

- **遍历 AST**：深度优先遍历语法树
- **作用域管理**：进入块时压栈，退出块时弹栈
- **符号查找**：从当前作用域向上查找符号
- **类型推导**：自底向上计算表达式类型

5.2 编码完成后的修改

5.2.1 符号表实现（error/ErrorHandler.java）

数据结构（error/ErrorHandler.java:23-24）：

```
// 使用 Quadruple 存储：<符号表，是否函数作用域，函数类型，作用域编号>
private List<Quadruple<LinkedHashMap<String, Symbol>, Boolean, FuncType, Integer>>
    symbolTables = new ArrayList<>();
```

作用域管理：

```
// 添加新作用域 (error/ErrorHandler.java:26-29)
private void addSymbolTable(boolean isFunc, FuncType funcType) {
    symbolTables.add(new Quadruple<>(
        new LinkedHashMap<>(), isFunc, funcType, getTotalArea() + 1));
```

```
}

// 移除作用域 (error/ErrorHandler.java:41-44)
private void removeSymbolTable() {
    printSymbolTables.add(symbolTables.get(symbolTables.size() - 1));
    symbolTables.remove(symbolTables.size() - 1);
}

// 查找符号 (error/ErrorHandler.java:50-57)
private boolean contains(String ident) {
    for (int i = symbolTables.size() - 1; i >= 0; i--) {
        if (symbolTables.get(i).getFirst().containsKey(ident)) {
            return true;
        }
    }
    return false;
}

// 仅在当前作用域查找 (error/ErrorHandler.java:46-48)
private boolean containsInCurrent(String ident) {
    return symbolTables.get(symbolTables.size() - 1)
        .getFirst().containsKey(ident);
}
```

5.2.2 语义错误检测实现

名字重定义检测（错误类型 b）：

```
// 在声明时检查当前作用域是否已存在
if (containsInCurrent(identName)) {
    errors.add(new Error(lineNumber, ErrorType.b));
} else {
    put(identName, new Symbol(identName, "const/var/func"));
}
```

未定义名字检测（错误类型 c）：

```
// 使用标识符时检查是否已声明
if (!contains(identName)) {
    errors.add(new Error(lineNumber, ErrorType.c));
}
```

函数调用检查（错误类型 d, e）：

```
// 检查参数个数
if (actualParamCount != formalParamCount) {
    errors.add(new Error(lineNumber, ErrorType.d));
}

// 检查参数类型（维度匹配）
for (int i = 0; i < params.size(); i++) {
    if (actualDim[i] != formalDim[i]) {
        errors.add(new Error(lineNumber, ErrorType.e));
        break;
    }
}
```

常量修改检测（错误类型 h）：

```
// 赋值语句检查左值
Symbol symbol = get(lvalIdent);
if (symbol.getSymbolType().equals("const")) {
    errors.add(new Error(lineNumber, ErrorType.h));
}
```

return 语句检查（错误类型 f, g）：

```
// void 函数不应有返回值（错误类型 f）
if (funcType == FuncType.VOID && returnExp != null) {
    errors.add(new Error(lineNumber, ErrorType.f));
}

// int 函数最后一条语句必须是 return（错误类型 g）
// 在 Block 结束时检查
if (funcType == FuncType.INT && !lastStmtIsReturn) {
    errors.add(new Error(blockEndLine, ErrorType.g));
}
```

printf 格式检查（错误类型 l）：

```
// 统计格式字符串中 %d 的个数
int formatCount = 0;
for (int i = 0; i < formatStr.length() - 1; i++) {
    if (formatStr.charAt(i) == '%' && formatStr.charAt(i + 1) == 'd') {
        formatCount++;
    }
}

// 比较参数个数
if (formatCount != expCount) {
    errors.add(new Error(lineNumber, ErrorType.l));
}
```

循环语句检查（错误类型 m）：

```
// 维护循环深度计数器
private int loopDepth = 0;

// 进入循环时增加
loopDepth++;
visitLoop();
loopDepth--;

// 遇到 break/continue 时检查
if (loopDepth == 0) {
    errors.add(new Error(lineNumber, ErrorType.m));
}
```

5.2.3 常量表达式计算

在语义分析阶段计算常量表达式的值：

```
// 常量表同符号表一起管理 (error/ErrorHandler.java:92-100)
private Map<String, Integer> getCurConstTable() {
    return symbolTables.get(symbolTables.size() - 1).getSecond();
}

private void addConst(String name, Integer value) {
    getCurConstTable().put(name, value);
}

// 常量表达式求值
private Integer calcConstExp(ConstExpNode node) {
    // 递归计算表达式值
    // 支持：整数常量、常量标识符、算术运算
}
```

应用场景：

- 数组维度必须是常量表达式
- case 标签必须是常量表达式
- 初始化器中的常量折叠

5.2.4 类型系统设计

维度信息管理：

```
// 维度 0: 普通变量
// 维度 1: 一维数组
// 维度 2: 二维数组

class FuncParam {
    int dimension;      // 参数维度

    // 对于数组参数
    // int a[]     → dimension = 1
    // int a[][]   → dimension = 2
}
```

函数签名匹配:

```
// 参数类型检查考虑维度
// 例如: void f(int a[], int b)
// 调用: f(arr, x)
// 需要检查: arr 是一维数组, x 是普通变量
```

5.2.5 作用域特殊处理

函数参数作用域:

```
// 函数参数与函数体在同一作用域
private void visitFuncDef(FuncDefNode node) {
    addSymbolTable(true, funcType);  // 创建函数作用域

    // 添加函数参数到符号表
    for (FuncFParamNode param : params) {
        put(paramName, new Symbol(paramName, "param"));
    }

    // 处理函数体（不再创建新作用域）
    visitBlock(node.getBlock(), false);

    removeSymbolTable();
}
```

for 循环变量作用域:

```
// for 循环的初始化、条件、更新和循环体共享同一作用域
private void visitForStmt(StmtNode node) {
    addSymbolTable(false, null);

    visitForStmt(node.getForStmt1());  // 初始化
    visitCond(node.getCond());         // 条件
    visitForStmt(node.getForStmt2());  // 更新
    visitStmt(node.getStmt());         // 循环体

    removeSymbolTable();
}
```

5.2.6 数组处理

数组声明:

```
// int a[10][20];
ArraySymbol symbol = new ArraySymbol(
    name: "a",
    dimension: 2,
    dims: [10, 20]
);
```

数组使用检查:

```
// 检查下标个数是否匹配
// a[i][j] → 使用维度 2，符号维度 2 ✓
// a[i]    → 使用维度 1，符号维度 2 ✓（数组切片）
// a[i][j][k] → 使用维度 3，符号维度 2 ✗
```

5.2.7 错误输出格式

```
// 错误按行号排序输出 (error/ErrorHandler.java)
public void printErrors() throws IOException {
    errors.sort(Comparator.comparingInt(Error::getLineNumber));
    for (Error error : errors) {
        IOUtils.write(error.toString());  // 格式：行号 错误类型
    }
}
```

六、中间代码生成设计

6.1 编码前的设计

6.1.1 设计目标

- 将 AST 转换为 LLVM IR 风格的中间代码

- 采用 SSA（Static Single Assignment）形式
- 支持基本的控制流结构（if, for, while）
- 处理数组和函数调用
- 为后续优化提供良好的 IR 表示

6.1.2 IR 设计选择

选择 LLVM IR 风格的理由：

- **SSA 形式**：便于数据流分析和优化
- **类型化**：每个值都有明确的类型
- **三地址码**：指令简单，易于优化和目标代码生成
- **成熟的设计**：LLVM IR 是工业级编译器的标准

6.1.3 类型系统设计

```
abstract class Type {
    // 基本类型
    - IntegerType (i32)
    - VoidType (void)
    - LabelType (label, 用于基本块)

    // 复合类型
    - ArrayType ([n x Type])
    - PointerType (Type*)
    - FunctionType (RetType (ParamTypes))
}
```

6.1.4 Value 体系设计

```
abstract class Value {
    Type type;
    String name;
    List<Use> usesList;  // 使用链 (SSA 的基础)
}

// Value 的子类层次
Value
├── User (使用其他 Value 的值)
│   ├── Instruction (指令)
│   │   ├── BinaryInst (二元运算)
│   │   ├── MemInst (内存操作)
│   │   │   ├── AllocaInst (栈分配)
│   │   │   ├── LoadInst (加载)
│   │   │   ├── StoreInst (存储)
│   │   │   ├── GEPInst (GetElementPtr, 数组索引)
│   │   │   └── PhiInst (φ 函数, SSA 合并点)
│   │   └── TerminatorInst (终止指令)
│   │       ├── BrInst (分支)
│   │       ├── RetInst (返回)
│   │       └── CallInst (函数调用)
│   ├── Function (函数)
```

```
│   └── BasicBlock（基本块）
├── Const（常量）
│   ├── ConstInt（整数常量）
│   ├── ConstString（字符串常量）
│   └── ConstArray（数组常量）
└── GlobalVar（全局变量）
```

6.1.5 Use-Def 链设计

采用**双向链接**的 Use-Def 关系：

```
class Use {
    Value value;     // 被使用的值（Def）
    User user;       // 使用者（Use）
    int operandNo;   // 在 user 中是第几个操作数
}

class User extends Value {
    List<Value> operands;  // 操作数列表
}

class Value {
    List<Use> usesList;    // 所有使用该值的 Use
}
```

优点：

- 快速查找所有使用点（用于优化）
- 替换值时自动更新所有使用点
- 支持高效的死代码消除

6.1.6 IR 生成策略

- **访问者模式**：遍历 AST 生成 IR
- **符号表**：维护标识符到 Value 的映射
- **基本块管理**：维护当前基本块
- **临时变量**：使用 SSA 寄存器表示临时结果

6.2 编码完成后的修改

6.2.1 核心数据结构实现

IRModule 全局管理（ir/IRModule.java）：

```
class IRModule {
    private static final IRModule module = new IRModule();  // 单例

    private List<GlobalVar> globalVars;              // 全局变量列表
    private IList<Function, IRModule> functions;     // 函数列表

    public void addGlobalVar(GlobalVar gv);
    public IList<Function, IRModule> getFunctions();
    public void refreshRegNumber();  // 重新分配寄存器编号
}
```

Function 结构（ir/values/Function.java）：

```
class Function extends User {
    private FunctionType functionType;
    private IList<BasicBlock, Function> basicBlocks;  // 基本块列表
    private List<Value> args;                         // 参数列表
    private boolean isLibraryFunction;                // 库函数标记

    public void addBasicBlock(BasicBlock bb);
}
```

BasicBlock 结构（ir/values/BasicBlock.java）：

```
class BasicBlock extends Value {
    private IList<Instruction, BasicBlock> instructions;  // 指令列表
    private Function parent;
    private List<BasicBlock> predecessors;   // 前驱块
    private List<BasicBlock> successors;     // 后继块
}
```

侵入式链表（utils/IList.java, utils/INode.java）： 为了高效的插入/删除操作，使用侵入式双向链表：

```
class IList<T, P> {
    private INode<T, P> head;
    private INode<T, P> tail;

    // 支持 O(1) 的插入和删除
}

class INode<T, P> {
    private T value;
    private P parent;
    private INode<T, P> prev;
    private INode<T, P> next;
    private IList<T, P> parentList;

    public void removeFromList();  // 从链表中移除自己
}
```

6.2.2 IR 生成器实现（ir/LLVMGenerator.java）

生成器状态管理（ir/LLVMGenerator.java:21-50）：

```
class LLVMGenerator {
    private BuildFactory buildFactory;        // IR 构建工厂

    // 基本块管理
    private BasicBlock curBlock;              // 当前基本块
    private BasicBlock curTrueBlock;          // 条件真分支目标
    private BasicBlock curFalseBlock;         // 条件假分支目标
    private BasicBlock continueBlock;         // continue 跳转目标
    private BasicBlock curWhileFinalBlock;    // break 跳转目标

    private Function curFunction;             // 当前函数

    // 临时变量
    private Integer saveValue;                // 常量折叠的值
    private Operator saveOp;                  // 暂存运算符
    private Value tmpValue;                   // 临时值

    // 状态标志
    private boolean isGlobal;                 // 是否全局作用域
    private boolean isConst;                  // 是否常量表达式
    private boolean isArray;                  // 是否数组初始化
    private boolean isStatic;                 // 是否静态变量

    // 符号表和常量表
    private List<Pair<Map<String, Value>, Map<String, Integer>>> symbolTable;
}
```

符号表接口（ir/LLVMGenerator.java:65-110）：

```
// 当前作用域符号表
private Map<String, Value> getCurSymbolTable() {
    return symbolTable.get(symbolTable.size() - 1).getFirst();
}

// 添加符号
private void addSymbol(String name, Value value) {
    getCurSymbolTable().put(name, value);
}

// 查找符号（向上查找所有作用域）
private Value getValue(String name) {
    for (int i = symbolTable.size() - 1; i >= 0; i--) {
        if (symbolTable.get(i).getFirst().containsKey(name)) {
            return symbolTable.get(i).getFirst().get(name);
        }
    }
    return null;
}
```

```
// 常量表（用于编译期常量计算）
private void addConst(String name, Integer value);
private Integer getConst(String name);
```

BuildFactory 工厂模式（ir/values/BuildFactory.java）：

```
class BuildFactory {
    private static final BuildFactory instance = new BuildFactory();

    // 类型创建
    public IntegerType getI32Type();
    public ArrayType getArrayType(Type elementType, int size);
    public PointerType getPointerType(Type pointeeType);

    // 指令创建
    public BinaryInst createBinaryInst(Operator op, Value lhs, Value rhs, BasicBlock bb);
    public AllocaInst createAllocaInst(Type type, BasicBlock bb);
    public LoadInst createLoadInst(Value pointer, BasicBlock bb);
    public StoreInst createStoreInst(Value value, Value pointer, BasicBlock bb);
    public GEPInst createGEPInst(Value pointer, List<Value> indices, BasicBlock bb);
    public BrInst createBrInst(Value cond, BasicBlock trueBB, BasicBlock falseBB);
    public RetInst createRetInst(Value value, BasicBlock bb);
    public CallInst createCallInst(Function func, List<Value> args, BasicBlock bb);
}
```

6.2.3 表达式生成

算术表达式：

```
// 访问 AddExp: MulExp {('+' | '-') MulExp}
private Value visitAddExp(AddExpNode node) {
    Value left = visitMulExp(node.getMulExp());  // 左操作数

    if (node.getAddExp() != null) {  // 有加减运算
        Value right = visitAddExp(node.getAddExp());  // 右操作数
        Operator op = (node.getOp().equals("+")) ? Operator.ADD : Operator.SUB;

        // 常量折叠
        if (left instanceof ConstInt && right instanceof ConstInt) {
            int result = (op == Operator.ADD)
                ? ((ConstInt) left).getValue() + ((ConstInt) right).getValue()
                : ((ConstInt) left).getValue() - ((ConstInt) right).getValue();
            return new ConstInt(result);
        }

        // 生成指令
        return buildFactory.createBinaryInst(op, left, right, curBlock);
    }


    return left;
```

```
}
```

短路求值（逻辑与 `&&`）：

```
// LAndExp → EqExp {'&&' EqExp}
private Value visitLAndExp(LAndExpNode node) {
    if (node.getLAndExp() == null) {
        return visitEqExp(node.getEqExp());  // 单个 EqExp
    }

    // 需要短路求值
    BasicBlock currentBlock = curBlock;
    BasicBlock nextBlock = buildFactory.createBasicBlock(curFunction);
    BasicBlock savedTrueBlock = curTrueBlock;
    BasicBlock savedFalseBlock = curFalseBlock;

    // 左操作数为假则直接跳转到 false 分支
    curTrueBlock = nextBlock;
    curFalseBlock = savedFalseBlock;
    visitLAndExp(node.getLAndExp());

    // 右操作数
    curBlock = nextBlock;
    curTrueBlock = savedTrueBlock;
    curFalseBlock = savedFalseBlock;
    visitEqExp(node.getEqExp());

    return null;
}
```

数组访问（使用 GEP 指令）：

```
// a[i][j] → GEP 指令计算地址
private Value visitLVal(LValNode node) {
    String name = node.getIdent().getContent();
    Value arrayPtr = getValue(name);  // 获取数组指针

    if (node.getExps().isEmpty()) {
        // 没有下标，直接返回（数组名）
        return arrayPtr;
    }

    // 计算下标值
    List<Value> indices = new ArrayList<>();
    indices.add(new ConstInt(0));  // 第一个索引总是 0（LLVM 规定）

    for (ExpNode exp : node.getExps()) {
        indices.add(visitExp(exp));
    }


    // 生成 GEP 指令
```

```
    Value elementPtr = buildFactory.createGEPInst(arrayPtr, indices, curBlock);

    // 生成 Load 指令
    return buildFactory.createLoadInst(elementPtr, curBlock);
}
```

6.2.4 语句生成

赋值语句:

```
private void visitAssignStmt(StmtNode node) {
    LValNode lval = node.getLVal();
    ExpNode exp = node.getExp();

    // 计算右值
    Value rvalue = visitExp(exp);

    // 获取左值地址
    Value lvaluePtr = getLValuePointer(lval);

    // 生成 Store 指令
    buildFactory.createStoreInst(rvalue, lvaluePtr, curBlock);
}
```

if 语句:

```
private void visitIfStmt(StmtNode node) {
    BasicBlock thenBlock = buildFactory.createBasicBlock(curFunction);
    BasicBlock elseBlock = node.hasElse()
        ? buildFactory.createBasicBlock(curFunction)
        : null;
    BasicBlock mergeBlock = buildFactory.createBasicBlock(curFunction);

    // 条件表达式
    curTrueBlock = thenBlock;
    curFalseBlock = (elseBlock != null) ? elseBlock : mergeBlock;
    visitCond(node.getCond());

    // then 分支
    curBlock = thenBlock;
    visitStmt(node.getThenStmt());
    if (!curBlock.isTerminated()) {
        buildFactory.createBrInst(mergeBlock, curBlock);
    }

    // else 分支（如果有）
    if (elseBlock != null) {
        curBlock = elseBlock;
        visitStmt(node.getElseStmt());
        if (!curBlock.isTerminated()) {

            buildFactory.createBrInst(mergeBlock, curBlock);
```

```
        }
    }

    curBlock = mergeBlock;
}
```

for 循环:

```
private void visitForStmt(StmtNode node) {
    // for (init; cond; update) body

    // 初始化
    if (node.getInit() != null) {
        visitForStmt(node.getInit());
    }

    BasicBlock condBlock = buildFactory.createBasicBlock(curFunction);
    BasicBlock bodyBlock = buildFactory.createBasicBlock(curFunction);
    BasicBlock updateBlock = buildFactory.createBasicBlock(curFunction);
    BasicBlock exitBlock = buildFactory.createBasicBlock(curFunction);

    // 跳转到条件块
    buildFactory.createBrInst(condBlock, curBlock);

    // 条件块
    curBlock = condBlock;
    if (node.getCond() != null) {
        curTrueBlock = bodyBlock;
        curFalseBlock = exitBlock;
        visitCond(node.getCond());
    } else {
        buildFactory.createBrInst(bodyBlock, curBlock);
    }

    // 循环体
    BasicBlock savedContinue = continueBlock;
    BasicBlock savedBreak = curWhileFinalBlock;
    continueBlock = updateBlock;
    curWhileFinalBlock = exitBlock;

    curBlock = bodyBlock;
    visitStmt(node.getBody());
    if (!curBlock.isTerminated()) {
        buildFactory.createBrInst(updateBlock, curBlock);
    }

    // 更新块
    curBlock = updateBlock;
    if (node.getUpdate() != null) {
        visitForStmt(node.getUpdate());
    }

    buildFactory.createBrInst(condBlock, curBlock);
```

```
    // 恢复
    continueBlock = savedContinue;
    curWhileFinalBlock = savedBreak;
    curBlock = exitBlock;
}
```

函数调用:

```
private Value visitFuncCall(UnaryExpNode node) {
    String funcName = node.getIdent().getContent();
    Function function = (Function) getValue(funcName);

    // 计算实参
    List<Value> args = new ArrayList<>();
    if (node.getFuncRParams() != null) {
        for (ExpNode exp : node.getFuncRParams().getExps()) {
            args.add(visitExp(exp));
        }
    }

    // 生成 Call 指令
    return buildFactory.createCallInst(function, args, curBlock);
}
```

6.2.5 全局变量和数组初始化

全局变量:

```
private void visitGlobalVarDecl(VarDeclNode node) {
    for (VarDefNode varDef : node.getVarDefs()) {
        String name = varDef.getIdent().getContent();
        Type type = getType(varDef);

        // 计算初始值
        Value initializer = null;
        if (varDef.getInitVal() != null) {
            isGlobal = true;
            isConst = true;
            initializer = visitInitVal(varDef.getInitVal(), type);
            isConst = false;
            isGlobal = false;
        } else {
            initializer = getZeroInitializer(type);
        }

        // 创建全局变量
        GlobalVar gv = new GlobalVar(name, type, initializer);
        IRModule.getInstance().addGlobalVar(gv);
        addSymbol(name, gv);

    }
```

```
}
```

数组初始化：

```
// int a[2][3] = {{1, 2, 3}, {4, 5, 6}};
private Value visitArrayInitVal(InitValNode node, ArrayType arrayType) {
    List<Value> elements = new ArrayList<>();
    Type elementType = arrayType.getElementType();

    for (InitValNode subInit : node.getInitVals()) {
        if (elementType instanceof ArrayType) {
            // 嵌套数组
            elements.add(visitArrayInitVal(subInit, (ArrayType) elementType));
        } else {
            // 标量元素
            elements.add(visitExp(subInit.getExp()));
        }
    }

    // 填充未初始化的元素为 0
    while (elements.size() < arrayType.getSize()) {
        elements.add(new ConstInt(0));
    }

    return new ConstArray(arrayType, elements);
}
```

6.2.6 静态变量处理（static）

静态局部变量提升为全局变量：

```
// static int cnt = 0;  (在函数内部)
// 转换为:
// @func.cnt = internal global i32 0

private void visitStaticVarDecl(VarDeclNode node) {
    for (VarDefNode varDef : node.getVarDefs()) {
        String baseName = varDef.getIdent().getContent();

        // 生成唯一全局名（避免同名冲突）
        String globalName = curFunction.getName() + "." + baseName;

        // 如果多次声明，添加后缀
        int counter = staticVarCounter.getOrDefault(globalName, 0);
        if (counter > 0) {
            globalName = globalName + "." + counter;
        }
        staticVarCounter.put(globalName, counter + 1);

        // 创建全局变量

        Type type = getType(varDef);
```

```
        Value initializer = (varDef.getInitVal() != null)
            ? visitInitVal(varDef.getInitVal(), type)
            : getZeroInitializer(type);

        GlobalVar gv = new GlobalVar(globalName, type, initializer);
        IRModule.getInstance().addGlobalVar(gv);

        // 在当前作用域以原名注册
        addSymbol(baseName, gv);
    }
}
```

6.2.7 内置函数声明

库函数声明:

```
// 在 IR 开始时声明库函数
private void declareLibraryFunctions() {
    // declare i32 @getint()
    Function getint = new Function(
        "getint",
        new FunctionType(IntegerType.i32, new ArrayList<>()),
        true  // isLibraryFunction
    );
    IRModule.getInstance().getFunctions().add(getint);
    addGlobalSymbol("getint", getint);

    // declare void @putint(i32)
    List<Type> putintParams = new ArrayList<>();
    putintParams.add(IntegerType.i32);
    Function putint = new Function(
        "putint",
        new FunctionType(VoidType.voidType, putintParams),
        true
    );
    IRModule.getInstance().getFunctions().add(putint);
    addGlobalSymbol("putint", putint);

    // 类似地声明 putch, putstr 等
}
```

6.2.8 printf 处理

printf 转换为 putint/putch/putstr 调用:

```
private void visitPrintfStmt(StmtNode node) {
    String formatStr = node.getFormatString().getContent();
    List<ExpNode> exps = node.getExps();
    int expIndex = 0;


    // 解析格式字符串
```

```
    for (int i = 1; i < formatStr.length() - 1; i++) {  // 跳过引号
        char c = formatStr.charAt(i);

        if (c == '%' && i + 1 < formatStr.length() &&
            formatStr.charAt(i + 1) == 'd') {
            // %d → putint
            Value arg = visitExp(exps.get(expIndex++));
            Function putint = (Function) getValue("putint");
            buildFactory.createCallInst(putint, Arrays.asList(arg), curBlock);
            i++;  // 跳过 'd'
        } else if (c == '\\' && i + 1 < formatStr.length() &&
                   formatStr.charAt(i + 1) == 'n') {
            // \n → putch(10)
            Function putch = (Function) getValue("putch");
            buildFactory.createCallInst(putch,
                Arrays.asList(new ConstInt(10)), curBlock);
            i++;  // 跳过 'n'
        } else {
            // 普通字符 → putch
            Function putch = (Function) getValue("putch");
            buildFactory.createCallInst(putch,
                Arrays.asList(new ConstInt((int) c)), curBlock);
        }
    }
}
```

6.2.9 常量折叠

编译期常量计算:

```
private Value foldBinaryOp(Operator op, Value left, Value right) {
    if (!(left instanceof ConstInt) || !(right instanceof ConstInt)) {
        return null;  // 不是常量
    }

    int lval = ((ConstInt) left).getValue();
    int rval = ((ConstInt) right).getValue();
    int result;

    switch (op) {
        case ADD: result = lval + rval; break;
        case SUB: result = lval - rval; break;
        case MUL: result = lval * rval; break;
        case DIV:
            if (rval == 0) return null;  // 除零，不折叠
            result = lval / rval;
            break;
        case MOD:
            if (rval == 0) return null;
            result = lval % rval;
            break;

        default: return null;
```

```
    }

    return new ConstInt(result);
}
```

代数简化:

```
private Value simplifyBinaryInst(Operator op, Value left, Value right) {
    // x + 0 = x
    if (op == Operator.ADD && right instanceof ConstInt &&
        ((ConstInt) right).getValue() == 0) {
        return left;
    }

    // x * 1 = x
    if (op == Operator.MUL && right instanceof ConstInt &&
        ((ConstInt) right).getValue() == 1) {
        return left;
    }

    // x * 0 = 0
    if (op == Operator.MUL && right instanceof ConstInt &&
        ((ConstInt) right).getValue() == 0) {
        return new ConstInt(0);
    }

    // 更多简化规则...
}
```

七、代码优化设计

7.1 编码前的设计

7.1.1 设计目标

- 实现多种经典优化技术
- 采用 Pass 架构，便于扩展
- 优化分为局部、全局、过程间三个层次
- 保证优化的正确性

7.1.2 Pass 架构设计

```
interface Pass {
    interface IRPass {
        String getName();
        void run(IRModule module);
    }

    interface FunctionPass {
        String getName();
        void run(Function function);
    }
}
```

设计优点：

- **模块化**：每个 Pass 独立实现
- **可组合**：Pass 可以按任意顺序组合
- **可扩展**：添加新优化只需实现接口
- **可配置**：通过 Config 控制启用哪些 Pass

7.1.3 计划实现的优化

1. **死代码消除（DCE）**：删除无用代码
2. **公共子表达式消除（CSE）**：通过 GVN 实现
3. **常量传播**：在 GVN 中一并实现
4. **代数简化**：简化算术表达式
5. **分支优化**：消除不可达分支，合并基本块
6. **循环优化**：循环不变式外提（LICM）
7. **过程间分析**：分析函数调用关系

7.1.4 数据流分析框架

为优化提供必要的分析信息：

- **Use-Def 链**：已在 IR 设计中实现
- **支配树分析**：用于 SSA 构造和优化
- **循环检测**：识别自然循环
- **别名分析**：判断指针是否指向同一内存

7.2 编码完成后的修改

7.2.1 Pass 管理器实现（pass/PassModule.java）

```
public class PassModule {
    private static final PassModule INSTANCE = new PassModule();
    private final List<Pass.IRPass> irPasses = new ArrayList<>();

    public PassModule() {
        // 根据配置添加 Pass
        if (Config.GVNGCM) {
            irPasses.add(new GVNGCM());

        }
```

```
        if (Config.BranchOptimization) {
            irPasses.add(new BranchOptimization());
        }
        // 可以添加更多 Pass...
    }

    public void runIRPasses() {
        irPasses.forEach(p -> p.run(IRModule.getInstance()));
    }
}
```

Pass 运行流程（Compiler.java:47）：

```
// 在 IR 生成后，目标代码生成前运行优化
LLVMGenerator.getInstance().visitCompUnit(ast);
IOUtils.llvm_ir_raw(IRModule.getInstance().toString());  // 优化前
PassModule.getInstance().runIRPasses();                  // 运行优化
IOUtils.llvm_ir(IRModule.getInstance().toString());      // 优化后
```

7.2.2 死代码消除（DCE）

实现原理（pass/ir/DeadCodeElimination.java）：

```
public class DeadCodeElimination implements Pass.IRPass {
    @Override
    public void run(IRModule m) {
        // 1. 过程间分析：标记可达函数
        InterProceduralAnalysis preAnalysis = new InterProceduralAnalysis();
        preAnalysis.run(m);

        // 2. 删除不可达函数
        List<Function> funcToRemove = new ArrayList<>();
        for (Function func : m.getFunctions()) {
            if (!func.getName().equals("main") &&
                func.getPredecessors().isEmpty()) {
                funcToRemove.add(func);
            } else if (!func.isLibraryFunction()) {
                dce(func);  // 函数内 DCE
            }
        }

        // 3. 从 IR 中删除
        for (Function func : funcToRemove) {
            func.getNode().removeFromList();
        }
    }

    // 函数内死代码消除
    public void dce(Function func) {
        Set<Instruction> liveSet = new HashSet<>();
```

```
        // 1. 标记所有"必需"的指令
        for (BasicBlock bb : func.getBasicBlocks()) {
            for (Instruction inst : bb.getInstructions()) {
                if (dceNeed(inst)) {  // 有副作用的指令
                    findUsefulClosure(inst, liveSet);  // 递归标记依赖
                }
            }
        }

        // 2. 删除未标记的指令
        Set<Instruction> deadSet = new HashSet<>();
        for (BasicBlock bb : func.getBasicBlocks()) {
            for (Instruction inst : bb.getInstructions()) {
                if (!liveSet.contains(inst)) {
                    deadSet.add(inst);
                }
            }
        }

        // 3. 执行删除
        for (Instruction inst : deadSet) {
            inst.removeUseFromOperands();  // 从 Use-Def 链中移除
            inst.replaceUsedWith(null);
            inst.getNode().removeFromList();
        }

        // 4. 删除不可达基本块
        removeDeadBlock(func);
    }

    // 判断指令是否必需（有副作用）
    private boolean dceNeed(Instruction inst) {
        return inst instanceof TerminatorInst ||  // 控制流指令
               inst instanceof StoreInst ||       // 内存写入
               inst instanceof CallInst;          // 函数调用（可能有副作用）
    }

    // 递归标记依赖
    private void findUsefulClosure(Instruction inst, Set<Instruction> liveSet) {
        if (liveSet.contains(inst)) return;
        liveSet.add(inst);

        // 递归标记所有操作数
        for (Value operand : inst.getOperands()) {
            if (operand instanceof Instruction) {
                findUsefulClosure((Instruction) operand, liveSet);
            }
        }
    }
}
```

不可达基本块删除（pass/ir/DeadCodeElimination.java:67-90）：

```
private void removeDeadBlock(Function func) {
    // 1. 从入口块开始 DFS 标记可达块
    Set<BasicBlock> reachable = new HashSet<>();
    Stack<BasicBlock> stack = new Stack<>();
    stack.push(func.getEntryBlock());

    while (!stack.isEmpty()) {
        BasicBlock bb = stack.pop();
        if (reachable.contains(bb)) continue;
        reachable.add(bb);

        for (BasicBlock succ : bb.getSuccessors()) {
            stack.push(succ);
        }
    }

    // 2. 删除不可达块
    Set<BasicBlock> toRemove = new HashSet<>();
    for (BasicBlock bb : func.getBasicBlocks()) {
        if (!reachable.contains(bb)) {
            toRemove.add(bb);
        }
    }

    for (BasicBlock bb : toRemove) {
        // 清理 Phi 指令中对该块的引用
        for (BasicBlock succ : bb.getSuccessors()) {
            succ.removePredecessor(bb);
        }
        bb.getNode().removeFromList();
    }
}
```

7.2.3 GVN（全局值编号）+ GCM（全局代码移动）

GVN 原理（pass/ir/GVNGCM.java）：

```
public class GVNGCM implements Pass.IRPass {
    private final PairTable<Value, Value> valueTable = new PairTable<>();

    @Override
    public void run(IRModule m) {
        InterProceduralAnalysis ipa = new InterProceduralAnalysis();
        ipa.run(m);

        for (Function func : m.getFunctions()) {
            if (!func.isLibraryFunction()) {
                BranchOptimization bo = new BranchOptimization();

                do {
                    // 1. 死代码消除
                    DeadCodeElimination dce = new DeadCodeElimination();
```

```
                    dce.dce(func);

                    // 2. 别名分析
                    AliasAnalysis.run(func);

                    // 3. GVN
                    runGVN(func);

                    // 4. 清理 Use-Def 链
                    refreshUseList(func);
                    AliasAnalysis.clear(func);

                    // 5. 再次 DCE
                    dce = new DeadCodeElimination();
                    dce.dce(func);

                } while (bo.compute(func));  // 迭代直到不动点
            }
        }
    }

    // GVN 核心算法
    private void runGVN(Function func) {
        valueTable.clear();

        // 逆后序遍历基本块（保证定义在使用前被处理）
        List<BasicBlock> rpo = getReversePostOrder(func);

        for (BasicBlock bb : rpo) {
            for (Instruction inst : bb.getInstructions()) {
                Value simplified = simplifyInstruction(inst);

                if (simplified != null && simplified != inst) {
                    // 找到等价值，替换所有使用
                    inst.replaceUsedWith(simplified);
                    continue;
                }

                // 查找值表
                Value existingValue = valueTable.lookup(inst);
                if (existingValue != null) {
                    // 找到相同的表达式
                    inst.replaceUsedWith(existingValue);
                } else {
                    // 加入值表
                    valueTable.insert(inst, inst);
                }
            }
        }
    }
}
```

指令简化（pass/ir/GVNGCM.java）：

```
private Value simplifyInstruction(Instruction inst) {
    if (inst instanceof BinaryInst) {
        return simplifyBinaryInst((BinaryInst) inst);
    } else if (inst instanceof LoadInst) {
        return simplifyLoadInst((LoadInst) inst);
    } else if (inst instanceof GEPInst) {
        return simplifyGEPInst((GEPInst) inst);
    }
    return null;
}

private Value simplifyBinaryInst(BinaryInst inst) {
    Value lhs = inst.getOperand(0);
    Value rhs = inst.getOperand(1);
    Operator op = inst.getOperator();

    // 常量折叠
    if (lhs instanceof ConstInt && rhs instanceof ConstInt) {
        int l = ((ConstInt) lhs).getValue();
        int r = ((ConstInt) rhs).getValue();
        int result = 0;

        switch (op) {
            case ADD: result = l + r; break;
            case SUB: result = l - r; break;
            case MUL: result = l * r; break;
            case DIV: if (r != 0) result = l / r; else return null; break;
            case MOD: if (r != 0) result = l % r; else return null; break;
            default: return null;
        }

        return new ConstInt(result);
    }

    // 代数简化
    if (rhs instanceof ConstInt) {
        int c = ((ConstInt) rhs).getValue();

        // x + 0 = x, x - 0 = x
        if ((op == Operator.ADD || op == Operator.SUB) && c == 0) {
            return lhs;
        }

        // x * 0 = 0
        if (op == Operator.MUL && c == 0) {
            return new ConstInt(0);
        }

        // x * 1 = x
        if (op == Operator.MUL && c == 1) {
            return lhs;
        }
```

```
        // x / 1 = x
        if (op == Operator.DIV && c == 1) {
            return lhs;
        }

        // x - x = 0
        if (op == Operator.SUB && lhs.equals(rhs)) {
            return new ConstInt(0);
        }
    }

    return null;
}

private Value simplifyLoadInst(LoadInst inst) {
    Value pointer = inst.getPointer();

    // 查找最近的 Store（别名分析）
    BasicBlock bb = inst.getBasicBlock();
    INode<Instruction, BasicBlock> node = inst.getNode().getPrev();

    while (node != null) {
        Instruction prevInst = node.getValue();

        if (prevInst instanceof StoreInst) {
            StoreInst store = (StoreInst) prevInst;
            if (store.getPointer().equals(pointer)) {
                // 找到对同一地址的 Store
                return store.getValue();
            }
        }

        node = node.getPrev();
    }

    return null;
}
```

值表实现（使用哈希映射）：

```
class PairTable<K, V> {
    private Map<ValueWrapper, V> table = new HashMap<>();

    public void insert(K key, V value) {
        table.put(new ValueWrapper(key), value);
    }

    public V lookup(K key) {
        return table.get(new ValueWrapper(key));
    }


    public void clear() {
```

```
        table.clear();
    }

    // 用于哈希和相等性比较的包装类
    private static class ValueWrapper {
        private Value value;

        @Override
        public int hashCode() {
            if (value instanceof BinaryInst) {
                BinaryInst inst = (BinaryInst) value;
                return Objects.hash(inst.getOperator(),
                    inst.getOperand(0), inst.getOperand(1));
            }
            // 其他类型...
            return value.hashCode();
        }

        @Override
        public boolean equals(Object obj) {
            // 结构相等性判断
        }
    }
}
```

7.2.4 分支优化

实现（pass/ir/BranchOptimization.java）：

```
public class BranchOptimization implements Pass.IRPass {
    @Override
    public void run(IRModule m) {
        for (Function func : m.getFunctions()) {
            if (!func.isLibraryFunction()) {
                compute(func);
            }
        }
    }

    public boolean compute(Function func) {
        boolean changed = false;

        for (BasicBlock bb : func.getBasicBlocks()) {
            Instruction terminator = bb.getTerminator();

            if (terminator instanceof BrInst) {
                BrInst br = (BrInst) terminator;

                // 无条件分支优化
                if (!br.isConditional()) {
                    BasicBlock target = br.getTargetBlock();
```

```
                    // 合并只有一个前驱的块
                    if (target.getPredecessors().size() == 1 &&
                        target != func.getEntryBlock()) {
                        mergeBlocks(bb, target);
                        changed = true;
                    }
                }
                // 条件分支优化
                else {
                    Value cond = br.getCondition();

                    // 条件为常量
                    if (cond instanceof ConstInt) {
                        int val = ((ConstInt) cond).getValue();
                        BasicBlock target = (val != 0)
                            ? br.getTrueBlock()
                            : br.getFalseBlock();

                        // 替换为无条件分支
                        BrInst newBr = buildFactory.createBrInst(target, bb);
                        br.replaceUsedWith(newBr);
                        br.getNode().removeFromList();
                        changed = true;
                    }
                    // 两个分支相同
                    else if (br.getTrueBlock().equals(br.getFalseBlock())) {
                        BrInst newBr = buildFactory.createBrInst(
                            br.getTrueBlock(), bb);
                        br.replaceUsedWith(newBr);
                        br.getNode().removeFromList();
                        changed = true;
                    }
                }
            }
        }

        return changed;
    }

    // 合并基本块
    private void mergeBlocks(BasicBlock pred, BasicBlock succ) {
        // 将 succ 的所有指令移动到 pred（除了终止指令）
        Instruction predTerminator = pred.getTerminator();
        predTerminator.getNode().removeFromList();

        for (Instruction inst : succ.getInstructions()) {
            inst.getNode().removeFromList();
            pred.addInstruction(inst);
        }

        // 更新后继的前驱
        for (BasicBlock succSucc : succ.getSuccessors()) {

            succSucc.replacePredecessor(succ, pred);
```

```
        }

        // 删除 succ
        succ.getNode().removeFromList();
    }
}
```

7.2.5 过程间分析

实现（pass/ir/InterProceduralAnalysis.java）：

```
public class InterProceduralAnalysis implements Pass.IRPass {
    @Override
    public void run(IRModule m) {
        // 构建调用图
        buildCallGraph(m);

        // 分析函数属性
        analyzeFunctionProperties(m);
    }

    private void buildCallGraph(IRModule m) {
        for (Function caller : m.getFunctions()) {
            if (caller.isLibraryFunction()) continue;

            for (BasicBlock bb : caller.getBasicBlocks()) {
                for (Instruction inst : bb.getInstructions()) {
                    if (inst instanceof CallInst) {
                        CallInst call = (CallInst) inst;
                        Function callee = call.getCalledFunction();

                        // 记录调用关系
                        callee.addPredecessor(caller);
                        caller.addSuccessor(callee);
                    }
                }
            }
        }
    }

    private void analyzeFunctionProperties(IRModule m) {
        for (Function func : m.getFunctions()) {
            if (func.isLibraryFunction()) continue;

            // 分析函数是否有副作用
            boolean hasSideEffect = false;
            for (BasicBlock bb : func.getBasicBlocks()) {
                for (Instruction inst : bb.getInstructions()) {
                    if (inst instanceof StoreInst ||
                        inst instanceof CallInst) {
                        hasSideEffect = true;

                        break;
```

```
                    }
                }
                if (hasSideEffect) break;
            }
            func.setHasSideEffect(hasSideEffect);

            // 分析函数是否总是返回常量
            // ...
        }
    }
}
```

7.2.6 别名分析

实现（ir/analysis/AliasAnalysis.java）：

```
public class AliasAnalysis {
    // 简单的基于 Alloca 的别名分析
    public static void run(Function func) {
        Map<Value, Set<Value>> aliasMap = new HashMap<>();

        for (BasicBlock bb : func.getBasicBlocks()) {
            for (Instruction inst : bb.getInstructions()) {
                if (inst instanceof AllocaInst) {
                    // 每个 Alloca 创建独立的内存位置
                    Set<Value> aliasSet = new HashSet<>();
                    aliasSet.add(inst);
                    aliasMap.put(inst, aliasSet);
                } else if (inst instanceof GEPInst) {
                    // GEP 继承基址的别名集合
                    GEPInst gep = (GEPInst) inst;
                    Value base = gep.getPointer();
                    if (aliasMap.containsKey(base)) {
                        aliasMap.put(gep, aliasMap.get(base));
                    }
                }
            }
        }

        // 保存分析结果
        func.setAliasAnalysisResult(aliasMap);
    }

    public static boolean mayAlias(Value ptr1, Value ptr2, Function func) {
        Map<Value, Set<Value>> aliasMap = func.getAliasAnalysisResult();
        if (aliasMap == null) return true;  // 保守假设

        Set<Value> set1 = aliasMap.get(ptr1);
        Set<Value> set2 = aliasMap.get(ptr2);

        if (set1 == null || set2 == null) return true;
```

```
        // 检查别名集合是否相交
        return !Collections.disjoint(set1, set2);
    }

    public static void clear(Function func) {
        func.setAliasAnalysisResult(null);
    }
}
```

7.2.7 支配树分析

实现（ir/analysis/DomAnalysis.java）：

```
public class DomAnalysis {
    private Function function;
    private Map<BasicBlock, BasicBlock> idom;  // 直接支配节点
    private Map<BasicBlock, Set<BasicBlock>> domFrontier;  // 支配边界

    public void run(Function func) {
        this.function = func;
        this.idom = new HashMap<>();
        this.domFrontier = new HashMap<>();

        computeDominators();
        computeDominanceFrontier();
    }

    // 使用迭代算法计算支配关系
    private void computeDominators() {
        List<BasicBlock> blocks = function.getBasicBlocks();
        BasicBlock entry = function.getEntryBlock();

        // 初始化
        Map<BasicBlock, Set<BasicBlock>> dom = new HashMap<>();
        for (BasicBlock bb : blocks) {
            if (bb == entry) {
                Set<BasicBlock> entryDom = new HashSet<>();
                entryDom.add(entry);
                dom.put(entry, entryDom);
            } else {
                dom.put(bb, new HashSet<>(blocks));  // 所有块
            }
        }

        // 迭代直到不动点
        boolean changed = true;
        while (changed) {
            changed = false;

            for (BasicBlock bb : blocks) {
                if (bb == entry) continue;
```

```
                // Dom(bb) = {bb} ∪ (∩ Dom(pred) for pred in Predecessors(bb))
                Set<BasicBlock> newDom = new HashSet<>();
                newDom.add(bb);

                Set<BasicBlock> intersection = null;
                for (BasicBlock pred : bb.getPredecessors()) {
                    if (intersection == null) {
                        intersection = new HashSet<>(dom.get(pred));
                    } else {
                        intersection.retainAll(dom.get(pred));
                    }
                }

                if (intersection != null) {
                    newDom.addAll(intersection);
                }

                if (!newDom.equals(dom.get(bb))) {
                    dom.put(bb, newDom);
                    changed = true;
                }
            }
        }

        // 计算直接支配节点
        for (BasicBlock bb : blocks) {
            if (bb == entry) continue;

            Set<BasicBlock> dominators = new HashSet<>(dom.get(bb));
            dominators.remove(bb);

            // idom(bb) 是 dominators 中距离 bb 最近的节点
            BasicBlock idomBB = null;
            for (BasicBlock dominator : dominators) {
                if (idomBB == null ||
                    !dom.get(dominator).contains(idomBB)) {
                    idomBB = dominator;
                }
            }

            idom.put(bb, idomBB);
        }
    }

    // 计算支配边界
    private void computeDominanceFrontier() {
        for (BasicBlock bb : function.getBasicBlocks()) {
            Set<BasicBlock> df = new HashSet<>();

            for (BasicBlock pred : bb.getPredecessors()) {
                BasicBlock runner = pred;
                while (runner != null && runner != idom.get(bb)) {

                    df.add(bb);
```

```
                    runner = idom.get(runner);
                }
            }

            domFrontier.put(bb, df);
        }
    }

    public BasicBlock getImmediateDominator(BasicBlock bb) {
        return idom.get(bb);
    }

    public Set<BasicBlock> getDominanceFrontier(BasicBlock bb) {
        return domFrontier.get(bb);
    }
}
```

7.2.8 循环分析

实现（ir/analysis/LoopInfo.java）：

```
public class LoopInfo {
    private Function function;
    private List<IRLoop> loops;

    public void analyze(Function func) {
        this.function = func;
        this.loops = new ArrayList<>();

        // 1. 构建支配树
        DomAnalysis domAnalysis = new DomAnalysis();
        domAnalysis.run(func);

        // 2. 识别回边 (back edge)
        Set<Pair<BasicBlock, BasicBlock>> backEdges = new HashSet<>();
        for (BasicBlock bb : func.getBasicBlocks()) {
            for (BasicBlock succ : bb.getSuccessors()) {
                if (dominates(succ, bb, domAnalysis)) {
                    // bb → succ 是回边
                    backEdges.add(new Pair<>(bb, succ));
                }
            }
        }

        // 3. 对每条回边识别自然循环
        for (Pair<BasicBlock, BasicBlock> edge : backEdges) {
            BasicBlock tail = edge.getFirst();
            BasicBlock header = edge.getSecond();

            IRLoop loop = new IRLoop(header);
            findLoopBlocks(tail, header, loop);

            loops.add(loop);
```

```
        }
    }

    private void findLoopBlocks(BasicBlock tail, BasicBlock header, IRLoop loop) {
        loop.addBlock(header);

        if (tail == header) return;

        Stack<BasicBlock> stack = new Stack<>();
        stack.push(tail);
        loop.addBlock(tail);

        while (!stack.isEmpty()) {
            BasicBlock bb = stack.pop();

            for (BasicBlock pred : bb.getPredecessors()) {
                if (!loop.contains(pred)) {
                    loop.addBlock(pred);
                    stack.push(pred);
                }
            }
        }
    }

    private boolean dominates(BasicBlock dominator, BasicBlock bb,
                              DomAnalysis domAnalysis) {
        BasicBlock runner = bb;
        while (runner != null) {
            if (runner == dominator) return true;
            runner = domAnalysis.getImmediateDominator(runner);
        }
        return false;
    }

    public List<IRLoop> getLoops() {
        return loops;
    }
}
```

7.2.9 优化效果示例

优化前:

```
define i32 @func(i32 %a, i32 %b) {
entry:
  %t1 = add i32 %a, 10
  %t2 = mul i32 %t1, 0      ; 结果总是 0
  %t3 = add i32 %t2, 5      ; 5 + 0 = 5
  %t4 = sub i32 %t3, 0      ; 无效的减法
  %t5 = mul i32 %b, 1       ; 无效的乘法
  %t6 = add i32 %t4, %t5
  ret i32 %t6
}
```

优化后（经过 GVN + 代数简化 + DCE）：

```
define i32 @func(i32 %a, i32 %b) {
entry:
  %t6 = add i32 5, %b       ; 简化为 5 + b
  ret i32 %t6
}
```

八、目标代码生成设计

8.1 编码前的设计

8.1.1 设计目标

- 将 LLVM IR 转换为 MIPS 汇编代码
- 实现寄存器分配
- 处理函数调用约定
- 生成可在 MARS 模拟器上运行的代码

8.1.2 MIPS 架构特点

- **寄存器**：32 个通用寄存器（$0-31$）
  - $0: 常量 0
  - $v0-v1: 函数返回值
  - $a0-a3: 函数参数
  - $t0-t9: 临时寄存器
  - $s0-s7: 保存寄存器
  - $sp: 栈指针
  - $fp: 帧指针
  - $ra: 返回地址
- **指令类型**：R 型、I 型、J 型
- **内存访问**：load/store 架构

8.1.3 调用约定设计

遵循标准 MIPS 调用约定：

- 前 4 个参数通过 $a0-$a3 传递，其余通过栈传递
- 返回值通过 $v0 传递
- 调用者保存寄存器：$t0-$t9
- 被调用者保存寄存器：$s0-$s7
- 栈帧布局：

```
高地址
+------------------+
| 返回地址 ($ra)   |
+------------------+
| 保存的 $s 寄存器 |
+------------------+
| 局部变量         |
+------------------+
| 临时变量         |
+------------------+  ← $sp
低地址
```

8.1.4 寄存器分配策略

- **简单分配**：为每个虚拟寄存器分配一个栈位置
- **改进分配**：使用 t 和 s 寄存器，溢出到栈

8.2 编码完成后的修改

8.2.1 MIPS 生成器实现（backend/MipsGenModule.java）

加载 IR：

```
public class MipsGenModule {
    private static final MipsGenModule instance = new MipsGenModule();

    private IRModule irModule;
    private List<String> mipsCode;

    public void loadIR() {
        this.irModule = IRModule.getInstance();
        this.mipsCode = new ArrayList<>();
    }

    public void genMips() {
        // 1. 生成 .data 段
        genDataSegment();

        // 2. 生成 .text 段
        genTextSegment();

        // 3. 输出汇编代码
        outputMips();

    }
```

```
}
```

数据段生成:

```
private void genDataSegment() {
    emit(".data");

    // 生成全局变量
    for (GlobalVar gv : irModule.getGlobalVars()) {
        if (gv.getInitializer() instanceof ConstInt) {
            // 整数
            ConstInt init = (ConstInt) gv.getInitializer();
            emit(gv.getName() + ": .word " + init.getValue());
        } else if (gv.getInitializer() instanceof ConstArray) {
            // 数组
            ConstArray init = (ConstArray) gv.getInitializer();
            emit(gv.getName() + ": .word " +
                 init.getElements().stream()
                     .map(e -> String.valueOf(((ConstInt) e).getValue()))
                     .collect(Collectors.joining(", ")));
        } else if (gv.getInitializer() instanceof ConstString) {
            // 字符串
            ConstString init = (ConstString) gv.getInitializer();
            emit(gv.getName() + ": .asciiz \"" +
                 escapeString(init.getValue()) + "\"");
        }
    }
}
```

代码段生成:

```
private void genTextSegment() {
    emit(".text");
    emit("jal main");       // 跳转到 main
    emit("li $v0, 10");     // 系统调用：退出
    emit("syscall");

    // 生成所有函数
    for (Function func : irModule.getFunctions()) {
        if (!func.isLibraryFunction()) {
            genFunction(func);
        }
    }
}

private void genFunction(Function func) {
    emit(func.getName() + ":");

    // 1. 分配栈帧
    int frameSize = calculateFrameSize(func);

    emit("addiu $sp, $sp, -" + frameSize);
```

```
    // 2. 保存 $ra (如果函数有调用)
    if (funcHasCall(func)) {
        emit("sw $ra, " + (frameSize - 4) + "($sp)");
    }

    // 3. 保存 $s 寄存器 (根据使用情况)
    // ...

    // 4. 生成函数体
    for (BasicBlock bb : func.getBasicBlocks()) {
        genBasicBlock(bb);
    }
}

private void genBasicBlock(BasicBlock bb) {
    // 基本块标签
    if (bb != bb.getFunction().getEntryBlock()) {
        emit(bb.getName() + ":");
    }

    // 生成指令
    for (Instruction inst : bb.getInstructions()) {
        genInstruction(inst);
    }
}
```

指令生成:

```
private void genInstruction(Instruction inst) {
    if (inst instanceof BinaryInst) {
        genBinaryInst((BinaryInst) inst);
    } else if (inst instanceof LoadInst) {
        genLoadInst((LoadInst) inst);
    } else if (inst instanceof StoreInst) {
        genStoreInst((StoreInst) inst);
    } else if (inst instanceof BrInst) {
        genBrInst((BrInst) inst);
    } else if (inst instanceof RetInst) {
        genRetInst((RetInst) inst);
    } else if (inst instanceof CallInst) {
        genCallInst((CallInst) inst);
    }
    // ... 其他指令类型
}

private void genBinaryInst(BinaryInst inst) {
    String dest = allocateRegister(inst);
    String lhs = getOperandReg(inst.getOperand(0));
    String rhs = getOperandReg(inst.getOperand(1));


    switch (inst.getOperator()) {
```

```
            case ADD:
                emit("add " + dest + ", " + lhs + ", " + rhs);
                break;
            case SUB:
                emit("sub " + dest + ", " + lhs + ", " + rhs);
                break;
            case MUL:
                emit("mul " + dest + ", " + lhs + ", " + rhs);
                break;
            case DIV:
                emit("div " + lhs + ", " + rhs);
                emit("mflo " + dest);  // 商在 lo
                break;
            case MOD:
                emit("div " + lhs + ", " + rhs);
                emit("mfhi " + dest);  // 余数在 hi
                break;
        }
    }

    private void genLoadInst(LoadInst inst) {
        String dest = allocateRegister(inst);
        Value pointer = inst.getPointer();

        if (pointer instanceof GlobalVar) {
            // 全局变量
            emit("lw " + dest + ", " + pointer.getName());
        } else {
            // 局部变量（栈上）
            int offset = getStackOffset(pointer);
            emit("lw " + dest + ", " + offset + "($sp)");
        }
    }

    private void genStoreInst(StoreInst inst) {
        String value = getOperandReg(inst.getValue());
        Value pointer = inst.getPointer();

        if (pointer instanceof GlobalVar) {
            emit("sw " + value + ", " + pointer.getName());
        } else {
            int offset = getStackOffset(pointer);
            emit("sw " + value + ", " + offset + "($sp)");
        }
    }

    private void genBrInst(BrInst inst) {
        if (!inst.isConditional()) {
            // 无条件跳转
            emit("j " + inst.getTargetBlock().getName());
        } else {
            // 条件跳转

            String cond = getOperandReg(inst.getCondition());
```

```
            emit("bne " + cond + ", $0, " + inst.getTrueBlock().getName());
            emit("j " + inst.getFalseBlock().getName());
        }
    }

    private void genRetInst(RetInst inst) {
        if (inst.hasReturnValue()) {
            // 将返回值放入 $v0
            String retVal = getOperandReg(inst.getReturnValue());
            if (!retVal.equals("$v0")) {
                emit("move $v0, " + retVal);
            }
        }

        // 恢复栈帧
        int frameSize = calculateFrameSize(inst.getFunction());

        // 恢复 $ra
        if (funcHasCall(inst.getFunction())) {
            emit("lw $ra, " + (frameSize - 4) + "($sp)");
        }

        // 恢复栈指针
        emit("addiu $sp, $sp, " + frameSize);

        // 返回
        emit("jr $ra");
    }

    private void genCallInst(CallInst inst) {
        Function callee = inst.getCalledFunction();
        List<Value> args = inst.getArguments();

        // 传递参数
        for (int i = 0; i < args.size(); i++) {
            String arg = getOperandReg(args.get(i));

            if (i < 4) {
                // 前 4 个参数通过 $a0-$a3
                emit("move $a" + i + ", " + arg);
            } else {
                // 其余参数通过栈
                emit("sw " + arg + ", " + ((i - 4) * 4) + "($sp)");
            }
        }

        // 调用函数
        if (callee.isLibraryFunction()) {
            genLibraryCall(callee, args);
        } else {
            emit("jal " + callee.getName());
        }
```

```
    // 获取返回值
    if (!inst.getType().isVoidType()) {
        String dest = allocateRegister(inst);
        if (!dest.equals("$v0")) {
            emit("move " + dest + ", $v0");
        }
    }
}

private void genLibraryCall(Function func, List<Value> args) {
    switch (func.getName()) {
        case "getint":
            emit("li $v0, 5");      // 系统调用：读整数
            emit("syscall");
            break;
        case "putint":
            emit("li $v0, 1");      // 系统调用：打印整数
            emit("syscall");
            break;
        case "putch":
            emit("li $v0, 11");     // 系统调用：打印字符
            emit("syscall");
            break;
        case "putstr":
            emit("li $v0, 4");      // 系统调用：打印字符串
            emit("syscall");
            break;
    }
}
```

寄存器分配：

```
// 简单策略：所有虚拟寄存器映射到栈
private Map<Value, Integer> stackOffsets = new HashMap<>();
private int currentOffset = 0;

private int getStackOffset(Value value) {
    if (!stackOffsets.containsKey(value)) {
        stackOffsets.put(value, currentOffset);
        currentOffset += 4;
    }
    return stackOffsets.get(value);
}

// 改进策略：使用临时寄存器
private Map<Value, String> regAlloc = new HashMap<>();
private Set<String> usedRegs = new HashSet<>();
private static final String[] tempRegs =
    {"$t0", "$t1", "$t2", "$t3", "$t4", "$t5", "$t6", "$t7", "$t8", "$t9"};

private String allocateRegister(Value value) {

    if (regAlloc.containsKey(value)) {
```

```
        return regAlloc.get(value);
    }

    // 尝试分配临时寄存器
    for (String reg : tempRegs) {
        if (!usedRegs.contains(reg)) {
            usedRegs.add(reg);
            regAlloc.put(value, reg);
            return reg;
        }
    }

    // 寄存器不足，溢出到栈
    int offset = getStackOffset(value);
    return offset + "($sp)";
}

private String getOperandReg(Value operand) {
    if (operand instanceof ConstInt) {
        // 常量立即数
        int val = ((ConstInt) operand).getValue();
        String tempReg = allocateTempReg();
        emit("li " + tempReg + ", " + val);
        return tempReg;
    } else {
        return allocateRegister(operand);
    }
}
```

栈帧大小计算:

```
private int calculateFrameSize(Function func) {
    int size = 0;

    // 1. 局部变量 (Alloca 指令)
    for (BasicBlock bb : func.getBasicBlocks()) {
        for (Instruction inst : bb.getInstructions()) {
            if (inst instanceof AllocaInst) {
                Type type = ((AllocaInst) inst).getAllocatedType();
                size += getTypeSize(type);
            }
        }
    }

    // 2. 保存的寄存器
    if (funcHasCall(func)) {
        size += 4;  // $ra
    }
    size += usedSRegs(func).size() * 4;  // $s 寄存器

    // 3. 对齐到 8 字节

    size = (size + 7) / 8 * 8;
```

```
    return size;
}
```

九、总结与展望

9.1 项目总结

9.1.1 完成的功能

本编译器实现了从源代码到目标代码的完整编译流程：

1. **前端**：
   - 词法分析：识别所有 Token 类型，处理注释
   - 语法分析：递归下降分析，构建 AST，支持错误恢复
   - 语义分析：类型检查，符号表管理，错误检测
2. **中端**：
   - IR 生成：生成 LLVM 风格的 SSA IR
   - 优化 Pass：死代码消除、GVN、分支优化、过程间分析
3. **后端**：
   - 目标代码生成：生成 MIPS 汇编代码
   - 寄存器分配：简单的寄存器分配策略

9.1.2 设计亮点

1. **现代化的 IR 设计**：
   - 采用 SSA 形式，便于优化
   - 完善的 Use-Def 链管理
   - 侵入式数据结构提高效率
2. **模块化架构**：
   - 单例模式保证全局一致性
   - Pass 架构易于扩展
   - 接口设计清晰
3. **完善的错误处理**：
   - 支持语法错误恢复
   - 全面的语义错误检测
   - 友好的错误报告
4. **实用的优化**：
   - GVN 消除冗余计算
   - 常量折叠和代数简化
   - 死代码消除

9.2 不足之处

1. **寄存器分配**：
    - 当前使用简单的栈分配策略
    - 缺少图着色等高级寄存器分配算法
2. **优化深度**：
    - 缺少循环优化（LICM、循环展开）
    - 没有实现内联优化
    - 过程间优化较为简单
3. **错误恢复**：
    - 语法错误恢复较为局限
    - 无法从某些严重错误中恢复
4. **代码质量**：
    - 生成的 MIPS 代码效率有待提高
    - 缺少窥孔优化

9.3 未来改进方向

1. **寄存器分配改进**：
    - 实现线性扫描寄存器分配
    - 或实现图着色寄存器分配
2. **更多优化 Pass**：
    - 循环不变式外提（LICM）
    - 循环展开
    - 函数内联
    - 尾调用优化
3. **更好的错误处理**：
    - 更全面的错误恢复策略
    - 更友好的错误信息
    - 警告系统
4. **目标代码优化**：
    - 指令调度
    - 窥孔优化
    - 更好的函数调用约定
5. **支持更多特性**：
    - 浮点数
    - 结构体
    - 指针运算
    - 更丰富的类型系统

9.4 学习收获

通过完成这个编译器项目，深入理解了：

- 编译原理的各个阶段及其相互关系
- 数据结构在编译器中的重要性（如 Use-Def 链、支配树）

- 优化技术的原理和实现
- 工程化的代码组织和架构设计
- 从理论到实践的转化过程

附录：关键文件列表

A.1 核心文件

文件路径	功能说明	行数
Compiler.java	编译器主入口	57
frontend/Lexer.java	词法分析器	175
frontend/Parser.java	语法分析器	729
error/ErrorHandler.java	错误处理+语义分析	~500
ir/LLVMGenerator.java	IR 生成器	~1000
ir/IRModule.java	IR 模块管理	~100
pass/PassModule.java	Pass 管理器	32
pass/ir/DeadCodeElimination.java	死代码消除	~150
pass/ir/GVNGCM.java	全局值编号	~300
pass/ir/BranchOptimization.java	分支优化	~150
backend/MipsGenModule.java	MIPS 代码生成	~800

A.2 总代码量估计

- 词法分析：~200 行
- 语法分析：~800 行
- 语义分析：~600 行
- IR 生成：~1500 行
- 优化 Pass：~800 行
- 目标代码生成：~1000 行
- 数据结构和工具类：~1000 行
- **总计：约 6000 行 Java 代码**
