2025北航编译技术课设——SysY编译器设计文档

一、参考 PL/0 编译器源代码分析介绍

1 总体结构

PL/0 编译器采用单遍扫描+递归下降的编译架构，其中将部分操作封装为16个Procedure和2个Function

- error: 错误处理，打印错误位置和错误码。
- getsym: 词法分析主过程，获取下一个Token。
- getch: 读取下一个字符。
- gen: 生成中间代码指令。
- test: 测试当前符号是否在符号表s1中，否则报错并合并s1和s2，跳转到下一个期望符号。
- block: 分程序处理。
- enter: 将标识符加入符号表
- position: 在符号表中查找标识符，返回位置；未找到返回0
- constdeclaration: 处理常量声明
- vardeclaration: 处理变量声明
- listcode: 输出该分程序输出的代码
- statement: 语句处理
- expression: 表达式处理
- term: 项处理
- factor: 因子处理
- condition: 条件处理
- interpret: 解释执行生成的中间代码
- base: 沿着display区向上找到基址并返回

最后在主程序中进行如下操作

1. 执行源文件读取。

2. 初始化各种常量(保留字表，符号映射，指令助记符，符号集合等）、全局变量（错误计数，字符位置，当前字符等）
3. 调用 `getsym` 进行词法分析，逐个返回 `symbol` 枚举值。
4. 调用 `block` 递归处理声明和语句，过程中使用 `gen` 生成中间代码。
5. 所有语句分析完毕后，若没有错误则使用 `interpret` 解释运行 `code` 数组中的指令。

---

2 接口设计

2.1 词法分析

通过调用 `getsym` 内部封装了 `getch` 用于读取字符，将源程序字符流转换为词法符号流，供语法分析使用。

2.2 语法分析

通过调用 `getsym` 获取词法符号，采用递归下降方法进行语法分析，在 `block` 中调用 `statement` 等模块完成语法分析

2.3 语义分析与代码生成

根据语法分析得到的语法成分，调用 `enter` 和 `position` 建立并维护符号表 `table`，调用 `gen` 过程生成中间代码 `code`，最后在末尾调用 `listcode` 输出生成的代码

2.4 解释运行模块

`interpret` 过程独立运行，解释执行生成的中间代码 `code`，实现 PL/0 程序的运行时行为。

2.5 错误处理

在单遍扫描时调用 `error` 和 `test` 进行错误处理

---

3 文件组织

该编译器为单文件结构，自顶向下分为：程序头与常量定义、过程与函数定义和主程序。总体按照编译流程来进行组织（词法分析->语法语义分析->中间代码生成->解释执行），同时又采用嵌套过程特性如

```
block (最外层)
├── enter
├── position
├── constdeclaration
├── vardeclaration
├── listcode
└── statement
    ├── expression
    |   ├── term
    |   |   └── factor (最内层，四层嵌套)
    |   └── condition
    └── (各种语句处理)
```

二、SysY 编译器总体设计

1 总体结构

```
├── Compiler.java
├── config.json
├── frontend
|       ├── AST
|       ├── Lexer.java
|       └── Parser.java
├── midend
|       ├── symbol
|       └── IR
└── backend
```

```
        └── ...
```

- **Complier（编译器程序入口）**：负责编译器IO部分，通过修改变量 `mode` 从而调整编译器运行哪一部分，如"lexer"表示运行到词法分析部分，"parser"运行到语法分析部分。
- **ErrorManager（错误管理器）**：维护 `List<Error>` 及 `Set<Integer>` 去重表，负责按行记录词法/语法错误并输出 `行号 错误码`。
- **Lexer（词法分析器）**：输入测试代码串，输出由 `TokenManager` 保存的 `Token` 序列；负责识别关键字、标识符、常量、运算符等，同时剥离注释、维护行号，并在出现 `a` 类错误时调用 `errormanager` 存储错误。
- **TokenManager（词法结果管理）**：持有 `List<Token>`；提供 `addToken`、`getToken(int)`、`size()` 等接口，供语法分析阶段顺序读取 token。
- **Parser（语法分析器）**：从 TokenManager 读取 token，递归下降解析 SysY 文法并构建 AST `CompUnit`，语法错误统一交给 ErrorManager。
- **AST 节点集合**：为语法规则提供结构化表示（`CompUnit`、`FuncDef`、`Stmt`、`AddExp` 等），并通过 `formatOutput()` 生成作业要求的语法树文本。
- **Symbol 符号表**：通过 `visitor` 来遍历语法成分，读取语法分析生成的抽象语法树，建立和维护符号表。
- **IR 中间代码生成**：...
- TODO

---

2 接口设计

- `Complier` 为程序入口，负责编译器IO部分，通过修改变量 `mode` 从而调整编译器运行哪一部分，如"lexer"表示运行到词法分析部分，"parser"运行到语法分析部分。
- 出现错误时统一调用 `errorManager.addError(lineNum, code)`。
- `Lexer` 在扫描过程中调用 `tokenManager.addToken(new Token(lexeme, type, lineNum))`。
- `Parser` 内部维护 `tokenIndex`、`currentToken`，通过 `tokenManager.getToken(tokenIndex)` 获取当前 `token`，通过 `preReadToken(n)` 实现有限预读。
- 语法分析完成后，可通过 `parser.getCompUnit().formatOutput()` 获取AST字符串，通过 `parser.getErrorsOutput()` 调用 `ErrorManager` 获取错误信息。

- TODO

---

3 文件组织

```
Compiler/
├── frontend/
|       ├── Lexer.java
|       ├── Token.java / TokenType.java
|       ├── TokenManager.java
|       ├── Parser.java
|       ├── Error.java / ErrorManager.java
|       └── AST/
|           ├── Node.java
|           ├── CompUnit.java
|           ├── FuncDef.java
|           ├── ...（若干节点类）
|   └── midend
|       ├── symbol
|       └── IR
|   └── backend
├── Compiler.java        # 主程序入口
├── lexer.txt            # 输出结果对照
└── error.txt            # 错误输出
```

三、词法分析

3.1 编码前设计

3.1.1 设计架构

```
Compiler/
├── Compiler.java
└── frontend/
    ├── Lexer.java
    ├── Token.java
    ├── TokenType.java
```

```
    ├── TokenManager.java
    └── Error.java
```

`Lexer`类负责管理词法分析主要逻辑，`Token`类负责封装词法单元，枚举类`TokenType`存储Token类型，`Error`类负责封装错误类与错误输出。

3.1.2 设计目标

将 SysY 源程序的字符流转换为 Token 流，采用有限状态自动机（DFA）进行词法识别，得到List<Token>供`TokenManager`管理，后续供语法分析使用

支持的 Token 类别：

- 关键字：const、int、static、void、if、else、for、while、break、continue、return、main、printf 等
- 标识符：以字母或下划线开头，后接字母/数字/下划线
- 整数常量
- 运算符：+ - * / % ! && || < > <= >= == != =
- 分界符：;,(){}[]
- 格式字符串：printf 的字符串常量
- EOF

3.1.3 错误处理

词法分析中仅可能出现a类错误，即非法符号（出现 '&' 和'|'，应当作 '&&' 与'||' 处理，但记录单词名称时仍为 '&' 和'|'，报错行号为符号所在行号）。在`Lexer`类中管理错误。

3.1.4 接口与输出

在`Compiler`中调用`lexer.getOutput()`输出到lexer.txt，调用`lexer.getErrorOutput()`输出到error.txt

3.2 编码完成之后的修改

3.2.1 架构调整

```
Compiler/
├── Compiler.java
└── frontend/
    ├── Lexer.java
    ├── Token.java
    ├── TokenType.java
    ├── TokenManager.java
    ├── Error.java
    └── ErrorManager.java
```

- 新增 `ErrorManager` 类，仿照 `TokenManager` 类进行错误管理，方便后续语法分析部分调用。
- 在 `Compiler` 中调用 `ErrorManager.getOutput()` 输出错误，调用 `TokenManager.getOutput()` 输出Token列表

3.2.2 设计调整

- 效仿pl0编译器，建立保留字表。
- 若读到'\n'，行号+1，当前索引+1
- 若读到空白，当前索引+1，跳过空白
- 若读到字母或下划线，调用 `handleIdentifier()` 处理标识符
- 若读到数字，调用 `handleNumber()` 处理常数
- 若读到'"'，调用 `handleString()` 处理字符串常量
- 否则调用 `handleSymbol()` 处理运算符和分节符或者注释
- 在 `handleIdentifier()` 中若匹配到保留字表中的键值，则识别为保留字Token，否则为标识符保存供语义分析使用

四、语法分析

4.1 编码前的设计

4.1.1 架构设计

```
Compiler/
├── Compiler.java
└── frontend/
    ├── AST
    |   ├── Node.java
    |   └── ...（节点类）
    ├── Lexer.java
    ├── Parser.java
    ├── Token.java
    ├── TokenType.java
    ├── TokenManager.java
    ├── Error.java
    └── ErrorManager.java
```

- 建立抽象语法树AST，由文法中所有产生式左部组成，如CompUnit, Stmt等，继承 `Node` 类，可通过 `node.getFormatOutput()` 输出语法分析要求的语法成分。
- 同时调整 `Compiler` 逻辑，通过修改变量 `mode` 从而调整编译器运行哪一部分，如"lexer"表示运行到词法分析部分，"parser"运行到语法分析部分。

4.1.2 设计目标

- 复用词法分析阶段生成的 `TokenManager` 与 `ErrorManager`，在保持单一源码载入的前提下完成递归下降分析。
- 构建覆盖实验文法的抽象语法树（AST），并保留评测所需的 `<非终结符>` 输出接口，为后续语义与中间代码阶段复用。
- 对 i/j/k 三类语法错误使用 `ErrorManager` 进行管理
- 使程序易于扩展，便于后续语义和中间代码阶段使用

4.1.3 文法改写与递归下降策略

将表达式等左递归规则按照 EBNF 形式改写为循环，并通过预读 `parser.preread(n)` 消除歧义规则。

4.1.4 错误处理方案

语法分析部分新增三种错误i/j/k，仍然在 `ErrorManager` 中进行错误管理

4.1.5 接口设计

- 在 `Parser` 类中调用 `TokenManager` 取出词法分析部分得到的Token列表，使用 `parseStmt` 等方法解析语法成分。
- 使用教程中的抽象语法树思路，将文法左部全部定义为 `Node` 子类，通过构造函数保存产生式右部的 token 或子节点，供编译器后续部分使用。
- 设计所有节点的 `formatOutput()` 方法以遵循评测格式：先递归输出子节点，再输出 `<非终结符>`。最后调用 `CompUnit.getFormatOutput()` 输出语法成分。

4.2 编码完成后的调整

4.2.1 接口调整

- `Compiler` 在 `parser` 模式下新增对 `Parser.init()` 的调用，对 `lexer` 的调用也改为 `Lexer.init()`；词法分析和语法分析共用一个 `ErrorManager` 减少开销。

4.2.2 递归下降子程序实现

- 为每个非终结符编写 `parse*` 函数，在函数内部通过局部变量暂存匹配到的 token，最后统一构造 AST 节点。
- 对含有 EBNF `{}` 结构的产生式（如 `AddExp`、`FuncFParams`），使用 `while` 循环与 `List` 收集重复子项，等价替换掉左递归实现。
- `parseStmt` 解析中使用 `isAssignStmt()` 向前预读扫描 `=` 与 `;`，在不引入回溯的情况下区分赋值与表达式语句；`break`、`continue`、`return`、`printf` 等关键字语句通过首符分类直接调用专用解析函数。

五、语义分析

5.1 编码前设计

5.1.1 设计架构

```
Compiler/
├── Compiler.java
└── midend/
    ├── symbol/
    |   ├── Symbol.java
    |   ├── ValueSymbol.java
    |   ├── FuncSymbol.java
    |   ├── SymbolType.java
    |   ├── Scope.java
    |   └── SymbolManager.java
    └── visitor/
        └── Visitor.java
```

- `Compiler` 在 `semantic` 模式下串联词法、语法和语义阶段，复用已有的 `TokenManager`、`ErrorManager` 以及 AST 根节点 `CompUnit`。
- 仿照 `TokenManager` 完成 `SymbolManager` 实现对符号表的管理和维护
- `midend.symbol` 负责符号表层次关系、符号实体以及作用域编号的分配。
- 采用 `Visitor` 类统一处理所有语义分析任务，在遍历 AST 的过程中同步完成符号收集、错误检查和符号表构建。

5.1.2 设计目标

- 遍历语法分析阶段生成的 AST，完成符号收集、类型检查、语义错误检测，并在正确程序下生成符号表输出。
- 统一管理 b/c/d/e/f/g/h/l/m 等语义错误，复用编译器前端的 `ErrorManager` 错误处理类

5.1.3 符号表与作用域

- 符号表条目需记录作用域 ID、名称、类型、定义行号、维度、是否静态、是否常量，符号表输出按作用域序号和声明顺序排序。
- 使用enterScope()和exitScope()维护作用域。作用域序号按照进入顺序递增，全局作用域为1；
- main函数不作为符号登记；

5.1.4 语义检查与错误处理

- 对 const/var/func Def执行重名检测（b 错误），对未定义的标识符报告 c 错误。
- 函数调用参数检查：参数个数不匹配报 d 错误，参数类型不匹配报 e 错误（在midend.semantic.TypeResolver中实现类型推导与检测）。
- void 函数有返回值报 f 错误（有多少错报多少错）；
- 有返回值函数缺少 return 报 g 错误（只检查函数体最后一句是否为Return语句且不检查Exp）；
- 常量赋值报 h 错误；
- printf 格式符数量不匹配报 l 错误。
- 非循环中使用 break/continue 报 m 错误；

5.1.5 接口设计

- 在 `Compiler` 中调用 `Visitor.init()` 启动语义分析，遍历完成后通过 `Visitor.getSymbolManager().getSymbolTableOutput()` 输出符号表。
- `SymbolManager` 提供 `enterScope()`、`exitScope()`、`addSymbol()`、`findSymbol()`、`findSymbolLocal()` 等接口供 Visitor 调用。
- 在 `Vistor` 类内部使用 `visit*` 方法遍历AST节点成分，实现语义检测

5.2 编码完成后的调整

5.2.1 架构调整

重构后新增 `midend.semantic` 包，将原本混杂在 `Visitor` 中的类型推导和错误检查逻辑分离到独立模块：

```
Compiler/
├── Compiler.java
└── midend/
    ├── symbol/
    |   ├── Symbol.java
    |   ├── ValueSymbol.java
    |   ├── FuncSymbol.java
    |   ├── SymbolType.java
    |   ├── Scope.java
    |   └── SymbolManager.java
    ├── semantic/
    |   ├── TypeResolver.java       # 新增：类型推导器
    |   └── SemanticChecker.java    # 新增：语义检查器
    └── visitor/
        └── Visitor.java
```

5.2.2 设计调整

- 符号表初始化时加入SysY标准库函数如 `getint()`，并在输出时判断 `Symbol` 类中新增的 `builtin` 标志位是否为真来决定是否输出符号。
- 创建 `TypeResolver` 类，专门负责计算表达式维度（0 表示 int，1 表示 int[]），
- `Visitor` 在需要类型信息时调用 `TypeResolver` 类方法。
- 创建 `SemanticChecker` 类，集中管理所有语义错误检查和处理。
- 每种错误类型对应一个方法，清晰的接口设计便于理解和修改。

```
public boolean checkRedefinition(String name, int lineNum)
  // b 错误
public Symbol checkUndefined(String name, int lineNum)
// c 错误
public boolean checkParamCount(int actual, int expected, int ln)
// d 错误
public boolean checkParamTypes(FuncRParams, FuncSymbol, int ln)
// e 错误
public void checkVoidReturn(boolean hasValue, int lineNum)
  // f 错误
public void checkMissingReturn(Block block, int lineNum)
// g 错误
public void checkConstAssign(Symbol symbol, int lineNum)
// h 错误
public void checkPrintfFormat(String fmt, int count, int lineNum)
 // l 错误
public void checkBreakContinue(int loopDepth, int lineNum)
  // m 错误
```

5.2.3 接口调整

- 支持命令行参数指定输入文件：`java Compiler <mode> <testfile>`，便于批量测试。

六、代码生成（MIPS）

6.1 编码前设计

6.1.1 设计架构

```
Compiler/
├── Compiler.java                # 入口，根据 mode 驱动后端
├── midend/
|   └── codegen/
|   └── llvm/          # LLVM IR 构造
└── backend/
    ├── BackEnd.java           # 接收 IR，调度目标生成
```

```
└── mips/
    ├── MipsGenerator.java     # 选择指令、布局栈帧、寄存器分配
    └── assembly/              # MARS 可运行汇编输出
```

- 输入：语义正确的 AST 与符号信息，可通过模式开关切换生成 LLVM IR（llvm_ir.txt），或者 MIPS（mips.txt）。
- 目标：32 位MIPS汇编代码，可直接在课程组mars上运行。或LLVM IR中间代码。

6.1.2 设计目标与策略

- IR 层：用简单 SSA 风格的 LLVM IR 表达控制流、算术和内存（alloca/load/store/getelementptr/call/ret）。
- 指令选择：对标 MIPS 基础指令（add/sub/mul/div/rem/and/or/sll/addu 等）与 syscall 入口。
- 寄存器策略：采用图着色算法，当寄存器不足时自动溢出到栈。
- 栈帧布局：统一按“局部/溢出/保存寄存器/$ra/对齐”顺序分配，所有偏移由`MipsGenerator`集中计算。
- 调用规约：前 4 个参数用 $a0-$a3，额外参数压栈，返回值在 $v0，调用点保存/恢复易失寄存器。

6.1.3 接口与运行方式

- 入口：`java Compiler [mips | llvm] [input]`，默认输入 testfile.txt，输出 mips.txt | llvm_ir.txt。
- 后端入口：BackEnd 接收模块与函数列表，逐函数调用 MipsGenerator.render() 生成汇编文本。
- Syscall 映射：putint(1)、getint(5)、putch(11)、putstr(4)。

6.2 编码完成后的调整

6.2.1 栈帧与寄存器保存

- 栈帧结构（低址→高址）：alloca 区 → spill 区 → 保存 $t0-$t7、$s2-$s7 → $ra → 对齐填充。

- 进入函数：`addiu $sp, $sp, -frameSize`；保存 $ra 与需保护的临时/保存寄存器。
- 退出函数：恢复寄存器，`addiu $sp, $sp, frameSize`，`jr $ra`。

6.2.2 参数传递与调用

- 前四参：生成器先加载到临时 $s2-$s5，再写入 $a0-$a3，避免互相覆盖。
- 函数额外参数：按从右到左压栈，调用前统一分配额外栈空间，调用后归还。
- 指针/数组参数：调用前保存调用点的 $sp 到 $s7，指针实参在栈上的用保存的基址+偏移形成地址传递，避免后续压栈移动导致地址失效。
- 调用点会保存/恢复 $t0-$t7、$s2-$s7，确保被调函数不需额外约定即可安全使用。

6.2.3 内存与地址计算

- alloca：为每个栈上变量分配字对齐空间，记录在 stackOffsets。
- load/store：优先使用寄存器分配结果，缺失则回退到栈偏移。
- getelementptr：支持全局与栈上数组，常量索引用立即数偏移，变量索引用 `sll` 乘 4 后相加。

6.2.4 指令选择与优化

- 算术/逻辑：直接映射 add/sub/mul/div/rem/and/or，比较用 seq/sne/sgt/sge/slt/sle 辅助伪指令。
- 分支：根据 icmp 结果生成 beq/bne/sgt 等条件跳转，块标签加函数名前缀避免冲突。
- 轻量优化：寄存器分配器尝试为高频值分配 $t/$s 寄存器；溢出统一下沉到栈。

6.2.5 运行时支持

- 全局区：.data 段放字符串常量和全局数组，使用 `la`+`lw/sw` 访问。
- IO：按 syscall 号封装 putint/putch/putstr/getint，调用端只需生成对应 call。

6.2.6 Bug修正

- 修复指针参数传递：指针实参改为传地址而非值，且地址基于调用前保存的 $sp，防止压栈后偏移漂移。
- 扩展保存集：调用点新增保存 $s2-$s7，避免在多数组参数场景覆盖指针基址。
- 栈帧尺寸同步：savedRegsOffset/frameSize 计算与新增保存寄存器保持一致，防止覆盖局部或参数区。

七、代码优化

7.1 编码前设计

7.1.1 设计架构

```
Compiler/
├── Compiler.java
└── src/
    ├── optimize/
    |   ├── Optimize.java                    # 优化基类
    |   ├── OptimizeManager.java             # 优化管理器
    |   ├── ConstantFoldingOptimize.java     # 常量折叠
    |   ├── CopyPropagationOptimize.java     # 拷贝传播
    |   ├── StoreForwardingOptimize.java     # 存储转发
    |   ├── DeadCodeEliminationOptimize.java # 死代码消除
    |   ├── MultiplyDivideOptimizer.java     # 乘除法优化
    |   └── PeepholeOptimizer.java           # 窥孔优化
    ├── backend/mips/
    |   ├── MipsBuilder.java                 # MIPS指令构建（含后端优
化）
    |   └── assembly/
    |       ├── MipsAlu2.java                # 两操作数ALU指令
    |       └── MipsMfhilo.java              # mfhi/mflo指令
    └── utils/
        └── Setting.java                     # 优化开关配置
```

7.1.2 设计目标

针对评测公式 `FinalCycle = DIV×15 + MULT×5 + MEM×3 + JUMP×2 + OTHER×1` 进行针对性优化：

1. **消除 DIV 指令**（15周期）：使用魔数乘法替代常数除法
2. **减少 MULT 指令**（5周期）：使用移位加减组合替代特定乘法
3. **减少冗余计算**：常量折叠、拷贝传播、死代码消除
4. **减少内存访问**：存储转发、寄存器分配优化

7.1.3 优化层次划分

- **中端优化**：在生成MIPS之前对IR进行变换
    - 常量折叠：编译期计算常量表达式
    - 拷贝传播：消除冗余的变量拷贝
    - 存储转发：将常量store的值传播到load
    - 死代码消除：删除未使用的指令和变量
- **后端优化**：在指令选择阶段进行
    - 乘除法强度削弱：用移位替代2的幂乘除
    - 魔数除法：用乘法+移位替代一般常数除法
    - 窥孔优化：消除冗余的move指令

7.1.4 优化执行顺序

```
存储转发 → 常量折叠 → 拷贝传播 → 死代码消除 → [迭代直到不动点] → MIPS生成（含后端优化）
```

7.1.5 接口设计

- `OptimizeManager`：管理所有中端优化，支持迭代执行直到不动点
- `Optimize`：优化基类，提供 `runOnModule()`、`runOnFunction()` 接口
- `Setting`：通过静态常量控制各优化开关，便于调试
- `MultiplyDivideOptimizer`：静态工具类，返回 `OptimizeResult` 描述优化策略

7.2 编码完成后的调整

7.2.1 中端优化实现

7.2.1.1 常量折叠（ConstantFoldingOptimize）

功能：在编译期计算常量表达式的值

支持的优化：

- 二元运算折叠：`7*5923` → `41461`
- 代数恒等式：`x+0=x`、`x*1=x`、`x*0=0`、`x/1=x`
- 比较运算折叠：`icmp eq i32 5, 5` → `1`

实现要点：

- 使用正则表达式匹配 LLVM IR 指令格式
- 维护 `Map<String, Integer>` 记录已知常量的SSA变量
- 折叠结果统一输出为 `add i32 0, <const>` 格式，便于后续拷贝传播处理

7.2.1.2 拷贝传播（CopyPropagationOptimize）

功能：将 `%t = add i32 0, %s` 形式的拷贝指令中 `%t` 的所有使用替换为 `%s`

实现要点：

- 第一遍：收集所有拷贝指令，建立 `dest → src` 映射
- 传播拷贝链：若 `%a→%b` 且 `%b→%c`，则更新为 `%a→%c`
- 第二遍：替换所有使用处的变量名

7.2.1.3 存储转发（StoreForwardingOptimize）

功能：对于只被存储一次常量值的局部变量，将后续的load替换为该常量

适用场景：`const int a1 = 1;` 这类常量变量

实现要点：

- 统计每个 alloca 变量的 store 次数

- 仅处理 `storeCount == 1` 且存储值为常量的情况
- 将 `load i32, i32* %ptr` 替换为 `add i32 0, <const>`

7.2.1.4 死代码消除（DeadCodeEliminationOptimize）

功能：删除未被使用的指令

实现要点：

- 标记阶段：从"根"指令（call、store到非死变量、ret、br、终结指令）出发，递归标记所有被使用的指令
- 清除阶段：删除未被标记的指令
- 死变量检测：识别只有alloca+store但无load的局部变量，将其标记为死变量

7.2.2 后端优化实现

7.2.2.1 乘法优化

支持的模式：

模式	示例	实现
2^n	x*8	`sll x, 3`
2^n+1	x*9	`(x<<3)+x`
2^n-1	x*7	`(x<<3)-x`
2^i+2^j	x*10	`(x<<3)+(x<<1)`
2^i-2^j	x*6	`(x<<3)-(x<<1)`
2^i+2^j+1	x*11	`(x<<3)+(x<<1)+x`

7.2.2.2 除法优化——魔数除法

原理：将 `x/d` 转换为 `(x * magic) >> (32 + shift)`

计算魔数：

```
// x / d ≈ mulhi(x, magic) >> shift
// 若 magic >= 2^31，需要额外加上被除数
```

收益：DIV(15周期) → MULT(5周期) + 少量移位，节省约10周期

实现要点：

- 使用 `mult rs, rt` 指令（两操作数形式）
- 使用 `mfhi rd` 获取乘法结果的高32位
- 需要处理负数情况：结果需加上 `(n >>> 31)`

7.2.2.3 取模优化

2的幂取模：`x % 2^n` 使用位运算实现（需处理负数符号）

一般取模：`x % d = x - (x/d)*d`，复用除法优化结果

7.2.2.4 窥孔优化

功能：扫描生成的MIPS指令序列，消除局部冗余

当前支持：

- 删除 `move $t, $t` 形式的自拷贝指令

7.2.3 优化效果验证

样例优化效果（testfile7）：

源代码中的复杂表达式：

```
j = 7*5923%56*57 - fib(fib(5)+2) + (a1+a2-(89/2*36-53)/1*6-2*
(45*56/85-56+35*56/4-9));
```

优化后的关键IR：

```
; 整个括号表达式被折叠为常量 -10091
%t45 = add i32 %t26, -10091
```

7.2.4 遇到的问题与解决

1. **魔数除法溢出问题**：当魔数 >= 2^31 时，有符号乘法会产生错误结果
    - 解决：设置 `addFlag`，在乘法后额外加上被除数
2. **取模优化寄存器冲突**：`x % d = x - (x/d)*d` 实现中，若 `rd == rs` 会导致原始值被覆盖
    - 解决：使用临时寄存器保存原始值
3. **拷贝传播的定义覆盖**：替换变量时不能替换定义本身
    - 解决：跳过以 `%dest =` 开头的指令