

编译原理课程设计报告

基于 Rust 的 PL/0 语言编译器与解释器

包含可视化的集成开发环境实现

学院：计算机科学与技术学院

专业：计算机科学与技术

日期：2025 年 12 月 30 日

Contents

1 小组成员及分工	4
1.1 小组成员	4
1.2 课程设计分工	4
2 PL/0 语言的语法图描述	5
2.1 PL/0 语言的 BNF 描述	5
2.2 程序 (Program)	6
2.3 分程序 (Block)	6
2.4 常量声明 (Const Declaration)	6
2.5 变量声明 (Var Declaration)	6
2.6 过程声明 (Procedure Declaration)	6
2.7 语句 (Statement)	7
2.8 表达式 (Expression)	7
2.9 项 (Term)	7
2.10 因子 (Factor)	7
2.11 条件 (Condition)	8
3 系统设计	9
3.1 系统的总体架构	9
3.2 主要功能模块的设计	9
3.2.1 符号表	9
3.2.2 词法分析器	9
3.2.3 语法制导翻译	10
3.2.4 解释器 (虚拟机)	10
3.2.5 界面/可视化设计	11
3.3 系统运行流程	11
4 系统实现	12
4.1 系统主要函数说明	12
4.1.1 符号表 (src/symbol_table.rs)	12
4.1.2 词法分析器 (src/lexer.rs)	12
4.1.3 语法制导翻译, 错误处理单元	12
4.1.4 虚拟机 (src/vm.rs)	16
4.1.5 图形界面 (src/gui.rs)	16
4.2 系统代码	16
4.2.1 符号表 (src/symbol_table.rs)	17
4.2.2 词法分析器 (src/lexer.rs)	18
4.2.3 类型定义 (src/types.rs)	22
4.2.4 抽象语法树 (src/ast.rs)	24
4.2.5 语法分析器 (src/parser.rs)	26
4.2.6 语义分析器 (src/semantic.rs)	36
4.2.7 优化器 (src/optimizer.rs)	41
4.2.8 代码生成器 (src/codegen.rs)	47
4.2.9 虚拟机 (src/vm.rs)	51
4.2.10 图形界面 (src/gui.rs)	57

4.2.11 主程序入口	79
5 系统测试	84
5.1 基础功能测试	84
5.1.1 测试程序 (<code>base1.txt</code>)	84
5.1.2 结果分析	84
5.2 控制流测试	85
5.2.1 测试程序 (<code>if-else.txt</code>)	85
5.2.2 结果分析	85
5.3 过程调用测试	86
5.3.1 测试程序 (<code>call.txt</code>)	86
5.3.2 结果分析	86
5.4 递归调用测试	87
5.4.1 测试程序 (<code>recursion.txt</code>)	87
5.4.2 结果分析	87
5.5 作用域与嵌套测试	88
5.5.1 测试程序 (<code>scope.txt</code>)	88
5.5.2 结果分析	89
5.6 错误恢复测试	89
5.6.1 测试程序 (<code>error_recovery.pl0</code>)	89
5.6.2 结果分析	90
5.7 优化测试	90
5.7.1 测试程序 (<code>loop_dag.pl0</code>)	90
5.7.2 结果分析	91
5.8 测试结果汇总	92
5.9 性能测试	92
5.10 功能特性总结	93
6 课程设计心得	94
6.1 黄耘青	94
6.2 赵乐坤	94
6.3 何东泽	94
7 参考文献	96

1 小组成员及分工

1.1 小组成员

学号	姓名	班级
Wait for input	Wait for input	计科 xx 班
Wait for input	Wait for input	计科 xx 班
Wait for input	Wait for input	计科 xx 班

1.2 课程设计分工

本项目采用敏捷开发的协作模式，利用 Git 进行版本控制。具体分工如下：

- 成员 1:
- 成员 2:
- 成员 3:

2 PL/0 语言的语法图描述

2.1 PL/0 语言的 BNF 描述

```
<prog> → program <id>; <block>

<block> → [<condecl>][<vardecl>][<proc>]<body>

<condecl> → const <const>{,<const>} ;

<const> → <id> := <integer>

<vardecl> → var <id>{,<id>} ;

<proc> → procedure <id> ([<id>{,<id>}]); <block> {;<proc>}

<body> → begin <statement>{;<statement>} end

<statement> → <id> := <exp>

<statement> → if <lexp> then <statement> [else <statement>]

<statement> → while <lexp> do <statement>

<statement> → call <id> [(<exp>{,<exp>})]

<statement> → <body>

<statement> → read (<id>{,<id>})

<statement> → write (<exp>{,<exp>})

<lexp> → <exp> <lop> <exp> | odd <exp>

<exp> → [+|-] <term> {<aop><term>}

<term> → <factor> {<mop><factor>}

<factor> → <id> | <integer> | (<exp>)

<lop> → = | <> | < | <= | > | >=

<aop> → + | -

<mop> → * | /

<id> → l{l|d}

<integer> → d{d}
```

注释：

<prog>: 程序; <block>: 块、程序体; <condecl>: 常量说明;
<const>: 常量; <vardecl>: 变量说明; <proc>: 分程序;
<body>: 复合语句; <statement>: 语句; <exp>: 表达式;
<lexp>: 条件; <term>: 项; <factor>: 因子;
<aop>: 加法运算符; <mop>: 乘法运算符; <lop>: 关系运算符

PL/0 语言是 Pascal 语言的一个子集，专为编译原理教学设计。它具备了过程式语言的核心特征，如过程嵌套定义、作用域规则、控制流语句（if, while）等。

为了准确描述 PL/0 的语法结构，我们采用语法图（Syntax Diagram）进行直观展示。

2.2 程序 (Program)

程序是语法分析的起点。一个 PL/0 程序由一个分程序 (Block) 和一个结束符号 (点号 .) 构成。

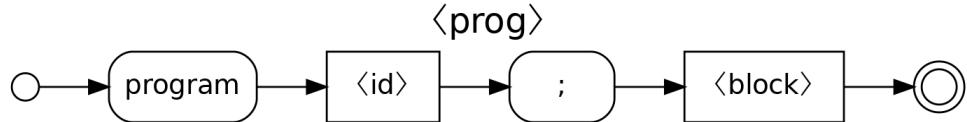


Figure 1: 程序语法图

2.3 分程序 (Block)

分程序是 PL/0 结构中最复杂的部分，它定义了作用域。一个 Block 依次包含：

1. 常量声明部分 (`const ...;`)
2. 变量声明部分 (`var ...;`)
3. 过程声明部分 (`procedure ...;`)
4. 语句部分 (`begin ... end`)

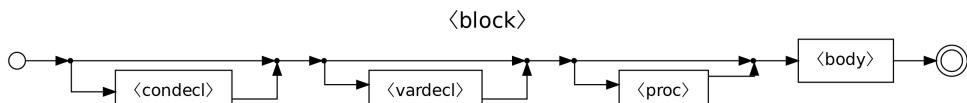


Figure 2: 分程序语法图

2.4 常量声明 (Const Declaration)

常量声明以 `const` 开头，后跟一系列赋值等式，多个常量之间用逗号分隔，最后以分号结束。

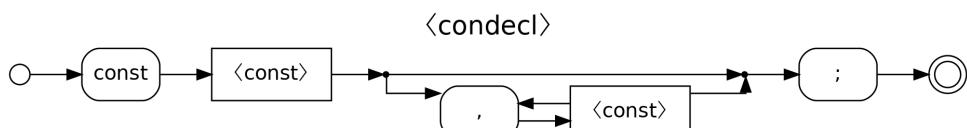


Figure 3: 常量声明语法图

2.5 变量声明 (Var Declaration)

变量声明以 `var` 开头，后跟一系列标识符，以分号结束。这些变量将在运行时栈中分配空间。

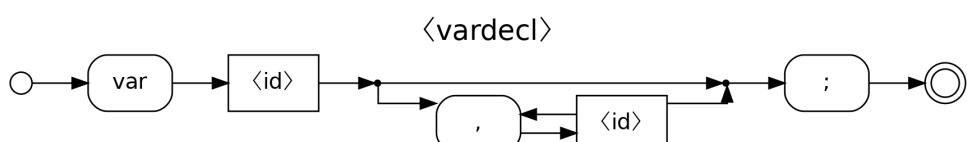


Figure 4: 变量声明语法图

2.6 过程声明 (Procedure Declaration)

过程声明允许嵌套定义。每个过程由过程头(包含过程名和可能的参数)和过程体(另一个 Block)组成。

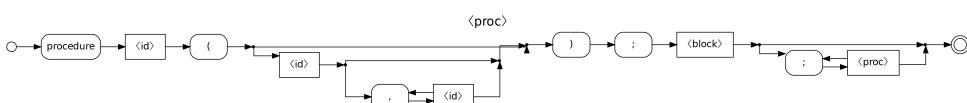


Figure 5: 过程声明语法图

2.7 语句 (Statement)

语句是程序执行的最小逻辑单元。PL/0 支持以下语句：
赋值语句: **ident := expression**
过程调用: **call ident**
复合语句: **begin ... end**
条件语句: **if condition then statement [else statement]**
循环语句: **while condition do statement**
读写语句: **read(...), write(...)**

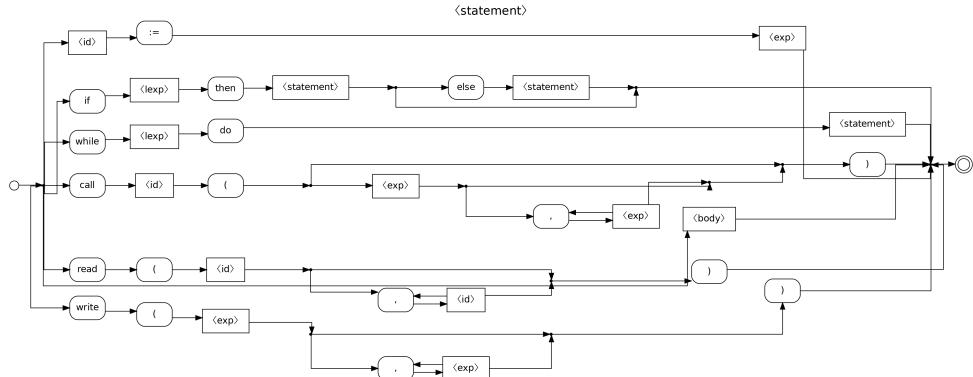


Figure 6: 语句语法图

2.8 表达式 (Expression)

表达式由项 (Term) 和加减运算符组成。

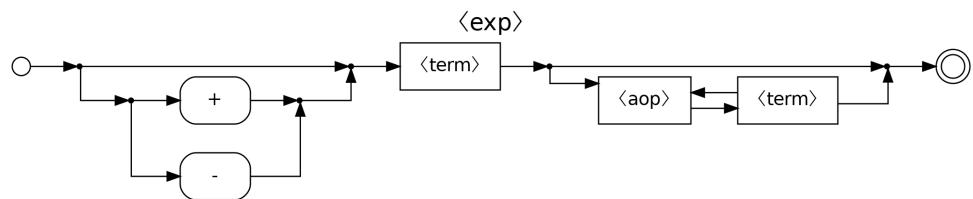


Figure 7: 表达式语法图

2.9 项 (Term)

项由因子 (Factor) 和乘除运算符组成。

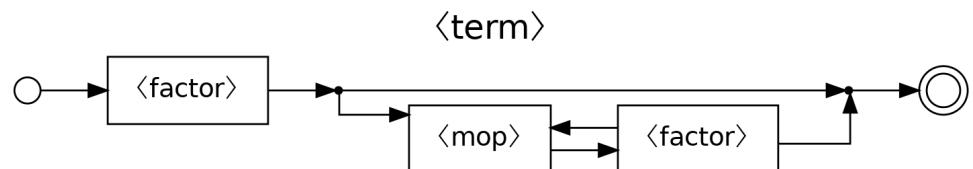


Figure 8: 项语法图

2.10 因子 (Factor)

因子是表达式的基本组成单位，可以是标识符、无符号整数或括号括起来的表达式。

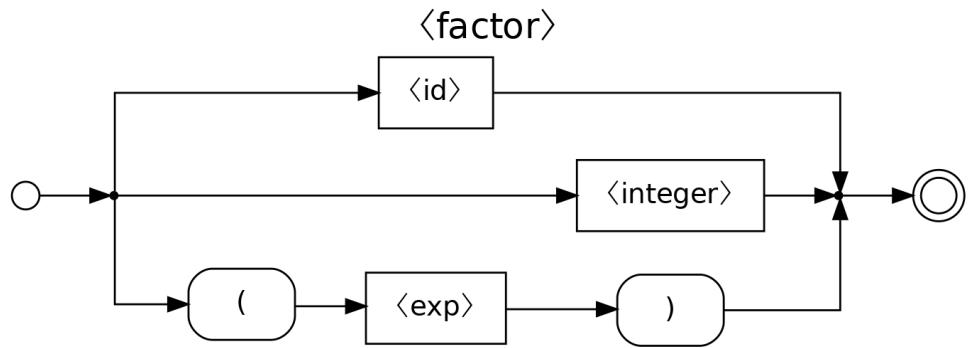


Figure 9: 因子语法图

2.11 条件 (Condition)

条件用于控制流语句的判断，包括 `odd` 表达式（判断奇偶）和关系运算（`=, #, <, <=, >, >=`）。

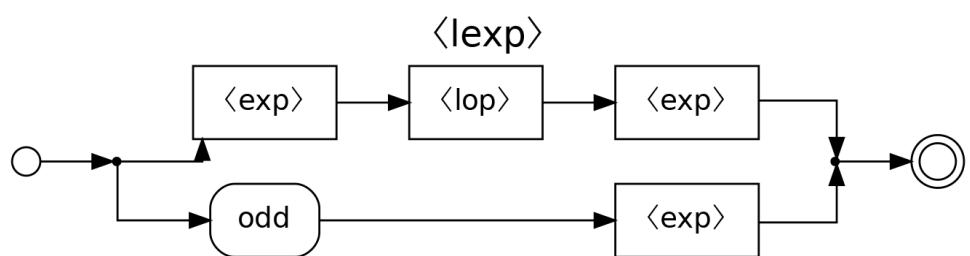


Figure 10: 条件语法图

3 系统设计

3.1 系统的总体架构

本系统采用现代编译器经典的 前端 (Frontend) - 中端 (Optimizer) - 后端 (Backend) 三层架构，并外挂一个基于 Rust 的高性能 GUI 界面。系统完全使用 Rust 语言编写，充分利用了其内存安全和类型系统的优势。

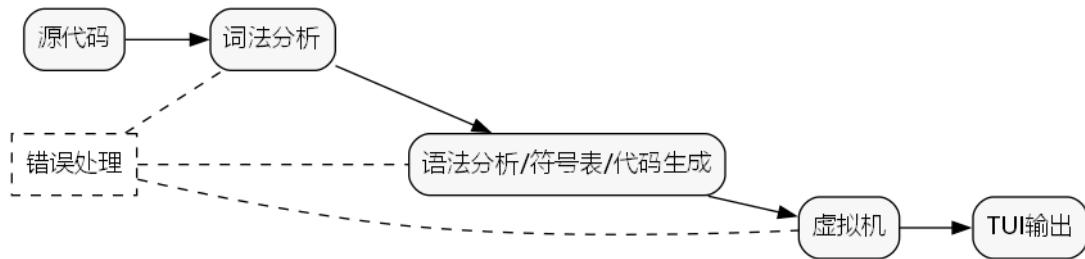


Figure 11: 基于 Rust 的 PL/0 编译系统架构

技术选型分析：为什么选择 Rust?

1. 安全性 (**Memory Safety**): 编译器的实现涉及大量的树结构 (AST) 和复杂的符号表管理。Rust 的所有权 (Ownership) 和借用 (Borrowing) 机制在编译期就杜绝了空指针解引用和数据竞争，这对于避免解释器崩溃至关重要。
2. 代数数据类型 (**Algebraic Data Types**): Rust 的 enum 非常适合表示 Token、AST 节点和虚拟机指令。例如，enum Statement 可以精确地描述各种语句类型，配合模式匹配 (match)，代码逻辑异常清晰。
3. 错误处理: 使用 Result<T, E> 替代传统的异常机制，强制开发者处理每一个可能的解析错误 (如 Parser::parse 返回 Result<Program, Vec<ParseError>>)，提高了编译器的健壮性。
4. 性能: Rust 编译生成的二进制文件性能接近 C/C++，保证了虚拟机执行的高效性。

3.2 主要功能模块的设计

3.2.1 符号表

基本介绍: 符号表 (SymbolTable) 是编译器中用于存储程序中使用的各种符号 (如变量、常量、过程) 及其相关信息的核心数据结构。它在编译过程中起着至关重要的作用，特别是在语义分析和代码生成阶段。

主要功能:

1. 多层作用域管理: 由于 PL/0 支持过程嵌套定义，符号表必须能够处理作用域链。在 src/symbol_table.rs 中，我们定义了 Scope 结构体，包含一个 HashMap<String, Symbol> 存储当前层级的符号，以及一个 Option<usize> 指向父作用域的索引，从而形成树状或栈状的作用域结构。
2. 符号属性记录: SymbolType 枚举区分了三种符号类型：
 - Constant: 记录常量的值 (val)。
 - Variable: 记录变量的层差 (level) 和相对地址 (addr)，用于运行时在栈中定位。
 - Procedure: 记录过程的入口地址 (addr)、层级 (level) 以及参数列表信息。
3. 静态语义检查: 在语义分析阶段，符号表用于检查变量是否未声明即使用、是否重复定义、以及赋值操作是否合法 (如不能给常量赋值)。

3.2.2 词法分析器

基本介绍: 词法分析器 (Lexer, src/lexer.rs) 是编译器的第一个阶段，负责将源代码字符流转换为 Token 流。它是一个子程序，每当语法分析器需要下一个 Token 时，它就从源码中读取字符直到识别出一个完整的单词。

主要功能:

1. **Token** 识别与分类: 依据 PL/0 的词法规则, 识别关键字 (如 `var`, `begin`)、标识符、数字字面量和运算符。在 `src/types.rs` 中定义了 `TokenType` 枚举来表示这些类型。
2. 超前搜索 (**Lookahead**): 为了区分如 `:` 和 `:=`, `>` 和 `>=` 等符号, `Lexer` 内部维护了一个 `Peekable<Chars>` 迭代器, 支持查看下一个字符而不消耗它。
3. 错误定位: 实时维护行号 (`line`) 和列号 (`col`)。当遇到非法字符时, 生成包含精确位置信息的错误报告。

3.2.3 语法制导翻译

本系统采用递归下降分析法, 将语法分析、语义分析和中间代码生成穿插进行。

3.2.3.1 语法分析

主要功能:

1. 递归子程序: 为每一个非终结符 (如 `program`, `block`, `statement`) 编写对应的 Rust 函数。由于 PL/0 文法是 LL(1) 的, 我们可以根据当前的 Token 唯一确定产生式。
2. 错误恢复 (**Panic Mode**): 当 Parser 遇到语法错误时, 不会立即停止, 而是进入“恐慌模式”。它会不断跳过输入的 Token, 直到遇到一个“同步符号” (如 `;`, `end`), 然后尝试恢复正常分析流程。这使得编译器一次运行能报告多个错误。

3.2.3.2 语义分析

主要功能:

1. 作用域绑定: 在解析变量声明时, 将其加入当前符号表; 在解析语句使用变量时, 在符号表中查找该变量, 计算其层差 (引用层 - 定义层) 和偏移地址。
2. 类型检查: 确保操作数的类型合法。例如, `call` 语句后必须跟一个过程名, 赋值语句左侧必须是变量而非常量。

3.2.3.3 目标代码生成

基本介绍: 代码生成器 (`CodeGenerator`) 在语法分析的过程中同步生成 P-Code 指令。

主要功能:

1. 指令生成 (**Emit**): 根据当前的语义动作生成对应的 `Instruction`。例如, 解析完一个加法表达式后, 生成 `OPR ADD` 指令。
2. 地址回填 (**Backpatching**): 对于控制流语句 (`if`, `while`), 在生成跳转指令 (`JMP`, `JPC`) 时, 目标地址往往尚未确定。系统采用“挖坑-回填”策略: 先生成带有占位地址的跳转指令, 记录其索引; 待目标代码块生成完毕后, 再回过头来修正跳转地址。

3.2.4 解释器 (虚拟机)

基本介绍: 解释器 (`VM`, `src/vm.rs`) 是一个栈式虚拟机, 用于执行生成的 P-Code。它模拟了真实的计算机硬件行为, 包括指令寄存器、程序计数器和数据栈。

运行时结构 (活动记录): 本系统采用经典的活动记录结构, 栈帧头部包含三个控制信息:

- **SL (Static Link)**: 静态链, 指向定义该过程的直接外层过程的栈帧基址, 用于访问非局部变量。
- **DL (Dynamic Link)**: 动态链, 指向调用者的栈帧基址, 用于过程返回时恢复环境。
- **RA (Return Address)**: 返回地址, 记录调用指令的下一条指令地址。

主要功能:

1. 指令循环: VM 核心是一个 `step()` 函数, 不断执行 `Fetch` (取指) -> `Decode` (译码) -> `Execute` (执行) 循环。
2. 栈操作与静态链查找: 实现了 `base(l)` 函数, 通过沿着 SL 链向上查找 `l` 层, 从而正确访问不同作用域层级的变量。

3. **I/O 模拟**: RED 指令从输入队列读取数据, WRT 指令将栈顶数据写入输出缓冲区, 实现了与 GUI 的交互。

3.2.5 界面/可视化设计

基本介绍: GUI 模块 (`src/gui.rs`) 基于 Rust 的 `egui` 库 (即时模式 GUI) 开发。它不仅是编译器的前端, 更是一个可视化的调试工具。

主要功能:

1. **AST 可视化**: 将编译器生成的抽象语法树结构渲染为图形化的树状图, 支持拖拽和缩放, 帮助用户理解代码结构。
2. **运行时栈监控**: 在程序运行时, 实时渲染 VM 的数据栈 (`stack`)。通过颜色区分不同的栈帧, 并清晰标记 `BP` (基址)、`SP` (栈顶) 以及 `SL`, `DL`, `RA` 的位置。
3. **源代码与字节码对照**: 并排显示源代码和生成的 P-Code, 高亮当前正在执行的指令, 实现源码级的单步调试体验。
4. **交互式控制台**: 提供输入框模拟标准输入, 日志区域模拟标准输出, 实现了完整的 IDE 体验。

3.3 系统运行流程

1. 用户在编辑器输入 PL/0 源码。
2. GUI 调用 `Lexer -> Parser -> Semantic -> Optimizer -> Codegen`。
3. 若编译成功, 生成 `Vec<Instruction>` 并初始化 VM。
4. 若编译失败, 在底部状态栏显示错误信息, 并高亮源码中的错误位置。
5. 用户点击运行, GUI 驱动 VM `step()`, 并按帧率重绘界面。

4 系统实现

本章将详细介绍系统核心函数的实现细节。

4.1 系统主要函数说明

4.1.1 符号表 (`src/symbol_table.rs`)

函数名	输入/输出	实现思想
<code>SymbolTable::resolve</code>	<code>name: &str</code> <code>Out: Option<&Symbol></code>	从当前作用域 (<code>current_scope</code>) 开始查找符号。若未找到，则通过 <code>parent</code> 索引递归向上查找父作用域，直到根作用域。这实现了静态作用域规则。
<code>SymbolTable::define</code>	<code>name: String, symbol: Symbol</code> <code>Out: Result<(), String></code>	在当前作用域的 <code>HashMap</code> 中插入新符号。插入前检查是否已存在同名符号，若存在则返回重复定义错误。

4.1.2 词法分析器 (`src/lexer.rs`)

函数名	输入/输出	实现思想
<code>Lexer::next_token</code>	<code>In: &mut self</code> <code>Out: TokenType</code>	核心状态机。首先跳过空白字符。根据首字符判断类型：字母开头进入 <code>scan_identifier</code> ; 数字开头进入 <code>scan_number</code> ; 符号则直接匹配(如 := 需超前查看)。
<code>Lexer::scan_identifier</code>	<code>In: &mut self</code> <code>Out: TokenType</code>	连续读取字母或数字，直到遇到非标识符字符。查表判断是否为关键字 (如 <code>begin</code> , <code>if</code>)，否则返回 <code>Identifier</code> 。

4.1.3 语法制导翻译, 错误处理单元

4.1.3.1 Analyse

对应 `Compiler::compile` 或 `GUI` 中的编译流程。

主要功能：语法、语义分析并判断是否错误。输入：无直接输入。输出：无。

4.1.3.2 Prog

对应 `Parser::program`。

主要功能：分析 `program` 关键字和相关部分的语法，处理变量名和语法错误。输入：通过词法分析器获取当前分析单词。输出：根据分析结果输出语法错误信息。实现思想：逐一检查是否产生错误信息（缺少 `program`、重复定义 `id` 等）调用 `block` 进一步解析。

4.1.3.3 Block

对应 `Parser::block`。

主要功能：处理程序中的 `block` 部分，包括常量声明、变量声明、过程声明和主体的语法分析及代码生成。输入：通过词法分析器获取当前分析单词。输出：输出语法分析和代码生成过程中的错误信息，并通过 `code.emit()` 开辟空间和生成返回指令。实现思想：

1. 初始化语句条数并记录当前位置，生成 `JMP` 指令用于跳转。
2. 依次检查 `const`、`var`、`procedure` 的声明，并调用相应的处理函数进行处理。

3. 回填开头的跳转指令，使得执行顺序从代码块的 `body` 部分开始。
4. 生成空间分配指令，调用 `body()` 解析实际的代码块内容。
5. 执行结束后生成返回指令，退出当前代码块。
6. 遇到程序结束符号 `.` 或错误时，跳过不合法的部分，进入下一个合法部分。

4.1.3.4 Conddecl

对应 `Parser::const_decl`。

主要功能：处理常量声明部分，支持多个常量的声明，每个常量使用逗号分隔，并检查语法错误。输入：通过词法分析器获取当前分析单词。输出：输出语法错误信息。实现思想：

1. 解析第一个常量声明，通过 `_const()` 处理。
2. 如果存在多个常量，判断逗号分隔并继续处理，检查是否遗漏逗号。
3. 检查常量声明的结束符号，若没有分号，报错。
4. 若遇到合法终结符（如 `var`, `procedure`, `begin`），停止当前分析并同步。

4.1.3.5 _const

辅助函数，解析单个常量赋值。

主要功能：处理单个常量的声明，检查常量的标识符、赋值符号、常量值，并记录到符号表中。输入：通过词法分析器获取当前分析单词。输出：输出语法错误信息。实现思想：

1. 判断常量声明的标识符是否合法，确保是有效的标识符。
2. 检查是否存在赋值符号 `:=`，如果没有则报错。
3. 检查赋值符号后面是否是整数常量，若不是则报错。
4. 如果一切合法，将常量记录到符号表中。

4.1.3.6 Vardecl

对应 `Parser::var_decl`。

主要功能：处理变量声明部分，支持多个变量的声明，检查标识符是否合法并记录到符号表。

输入：通过词法分析器获取当前分析单词。输出：输出语法错误信息。实现思想：

1. 解析第一个变量声明，通过检查标识符是否合法。
2. 支持多个变量声明，变量之间用逗号隔开，缺少逗号时报错。
3. 检查变量声明结束符号，若没有分号，则报错。
4. 处理变量声明时，若遇到错误或非法字符，进行同步跳过。

4.1.3.7 proc

对应 `Parser::proc_decl`。

主要功能：处理过程声明，检查过程标识符、参数、过程体等，确保语法正确并记录到符号表中。输入：通过词法分析器获取当前分析单词。输出：输出语法错误信息。实现思想：

1. 解析过程的关键字 `procedure`。
2. 处理过程名和参数，检查过程定义的合法性。
3. 解析参数列表，处理参数的合法性。
4. 进入过程体前，记录过程的起始地址，并增加当前空间。
5. 处理过程体（`block()`）。
6. 处理过程结束后，回收过程参数并进行同步。

4.1.3.8 body

对应 PL/0 中的语句块处理。

主要功能：解析 `begin` 和 `end` 之间的语句，处理语句的顺序与语法错误。输入：通过词法分析器获取当前分析单词。输出：输出语法错误信息。实现思想：

1. 检查是否以 `begin` 关键字开始。
2. 处理多个语句，确保每个语句之间正确分隔（使用分号`;`）。
3. 确保代码块以 `end` 关键字结束。
4. 在解析过程中进行错误同步，以确保能够继续分析。

4.1.3.9 statement

对应 `Parser::statement`。

主要功能：解析各种类型的语句，并进行错误处理与语法同步。输入：通过词法分析器获取当前分析单词。输出：输出语法、语义错误信息。实现思想：

1. 检查当前词汇是否属于语句的开始。
2. 识别并调用相应的语句处理函数。
3. 处理语法错误，提供同步机制，确保能够继续解析后续部分。

4.1.3.10 statement_id

处理赋值语句。

主要功能：解析并处理赋值语句，验证变量的有效性，确保赋值符号和右侧表达式的正确性。输入：通过词法分析器获取当前分析单词。输出：语法、语义错误信息。通过 `code.emit()` 输出目标代码，用于将计算结果存储到变量的地址。实现思想：

1. 检查当前单词是否为有效的变量（`id`）。
2. 确认赋值符号 `:=` 是否正确。
3. 解析赋值表达式并将结果存储到变量的地址。（生成目标代码 `STO`）。

4.1.3.11 statement_if

处理条件语句。

主要功能：解析 `if` 语句，包括条件表达式和可选的 `else` 分支，生成目标代码，处理条件跳转。输入：通过词法分析器获取当前分析单词。输出：输出语法错误信息。通过 `code.emit()` 生成目标代码，实现条件跳转。实现思想：

1. 检查是否是 `if` 语句的开头。
2. 解析条件表达式，将其计算结果放置在栈顶。
3. 处理 `then`，生成条件跳转代码。
4. 处理可选的 `else` 分支，生成跳转代码并回填跳转位置。

4.1.3.12 statement_while

处理循环语句。

主要功能：解析 `while` 循环，包括条件表达式和循环体，生成目标代码，处理条件跳转，模拟循环行为。输入：通过词法分析器获取当前分析单词。输出：语法、语义错误信息。通过 `code.emit()` 生成目标代码，实现循环的条件判断和跳转。实现思想：

1. 检查是否是 `while` 语句的开头。
2. 解析循环条件表达式，并将其结果放入栈顶。
3. 生成条件跳转代码，判断条件是否为真。
4. 解析循环体语句。
5. 生成跳转代码，将程序控制流跳回到循环开始位置。

4.1.3.13 statement_call

处理过程调用。

主要功能：解析和处理过程调用语句，包括验证过程名、检查参数传递的正确性、生成目标代码（CAL 指令）。**输入：**通过词法分析器获取当前分析单词。**输出：**语法、语义错误信息。通过 `code.emit()` 生成目标代码，进行过程调用。**实现思想：**

1. 检查 `call` 关键字是否存在，确保是过程调用语句。
2. 解析过程名，确保过程已定义且类型为过程。
3. 解析函数的参数列表，检查参数数量和格式。
4. 使用 CAL 指令生成调用过程的目标代码。

4.1.3.14 statement_read

处理读入语句。

主要功能：解析 `read` 语句并生成相应的目标代码。它检查 `read` 后面的变量，确保它们是合法的，并生成读取指令。**输入：**通过词法分析器获取当前分析单词。**输出：**语法、语义错误信息。通过 `code.emit()` 生成目标代码，模拟 `read` 操作。**实现思想：**

1. 检查 `read` 语句的正确性，包括关键字、括号和变量。
2. 查找符号表中对应的变量，确保它们是定义过且合法的变量。
3. 对每个变量生成读取指令（RED 和 STO），并将值存储到栈中。
4. 处理逗号分隔符，确保每个变量都合法。
5. 对语法错误进行报告，并在需要时进行同步。

4.1.3.15 statement_write

处理写出语句。

主要功能：解析 `write` 语句并生成相应的目标代码。将栈顶的表达式值输出，并处理多个值的输出。**输入：**通过词法分析器获取当前分析单词。**输出：**语法、语义错误信息。通过 `code.emit()` 生成目标代码，模拟 `write` 操作。**实现思想：**

1. 检查 `write` 语句的语法，包括关键字和括号。
2. 解析括号内的表达式，并生成输出指令。
3. 处理多个表达式的输出，并确保语法正确。
4. 处理右括号，并进行错误报告和同步。

4.1.3.16 exp

对应 `Parser::expression`。

主要功能：解析表达式并生成相应的目标代码。处理加法、减法和取反操作，并依赖 `term()` 来解析与加法、减法相关的操作。**输入：**通过词法分析器获取当前分析单词。**输出：**语法错误信息。通过 `code.emit()` 生成目标代码。**实现思想：**

1. 处理表达式的符号（+ 和 -）。
2. 解析乘除等优先级更高的运算（通过 `term()`）。
3. 处理连续的加法和减法操作。
4. 生成目标代码，模拟运算。

4.1.3.17 lepx

对应 `Parser::condition`。

主要功能：解析逻辑表达式（如 `odd` 运算和比较运算符），并生成相应的目标代码。**输入：**通过词法分析器获取当前分析单词。**输出：**语法错误信息。通过 `code.emit()` 生成目标代码，模拟逻辑运算。**实现思想：**

1. 判断是否为 `odd` 运算符，如果是，则进行处理。
2. 解析比较运算符（=, <, >, <= 等）。
3. 生成相应的目标代码。

4. 错误处理和语法修正。

4.1.3.18 term

对应 `Parser::term`。

主要功能：解析乘法和除法表达式。输入：通过词法分析器获取当前分析单词。输出：语法错误信息。通过 `code.emit()` 生成目标代码，模拟乘法和除法操作。实现思想：

1. 解析第一个因子。
2. 处理后续的乘法或除法运算符。
3. 生成相应的目标代码。

4.1.3.19 factor

对应 `Parser::factor`。

主要功能：解析因子，支持常量、变量和括号中的表达式。输入：通过词法分析器获取当前分析单词。输出：通过 `code.emit()` 生成目标代码，模拟对常量、变量或表达式的访问。实现思想：

1. 判断当前词汇类型：常量、变量或数字。
2. 处理左括号情况，递归解析表达式。
3. 根据词汇生成相应的目标代码。

4.1.4 虚拟机 (`src/vm.rs`)

函数名	输入/输出	实现思想
<code>VM::step</code>	<code>In: &mut self</code> <code>Out: () (Side Effects)</code>	取指： <code>I = code[P]; P++</code> 。译码执行： 根据 <code>I.f</code> (OpCode) 进行 <code>match</code> 分发。 执行算术指令时操作栈顶 <code>T</code> ；执行跳转指令时修改 <code>P</code> ；执行内存指令时利用 <code>base()</code> 计算地址。
<code>VM::base</code>	<code>In: l: usize</code> <code>Out: usize (Base Address)</code>	静态链查找核心。从当前基址 <code>B</code> 开始，沿着栈帧中的 <code>SL</code> (Static Link, 位于 <code>stack[B]</code>) 向上跳 1 次，返回目标层级的基址。

4.1.5 图形界面 (`src/gui.rs`)

函数名	输入/输出	实现思想
<code>Pl0Gui::update</code>	<code>In: &mut self, ctx: &Context</code> <code>Out: () (UI Render)</code>	即时模式 GUI 的核心循环。每一帧重新绘制界面。根据 <code>current_tab</code> 状态分别调用 <code>show_editor</code> , <code>show_ast</code> , <code>show_runtime</code> 等函数。处理用户输入事件并更新应用状态。
<code>Pl0Gui::compile</code>	<code>In: &mut self</code> <code>Out: () (Update State)</code>	串联整个编译流程： <code>Lexer -> Parser -> Semantic -> Codegen</code> 。若成功则更新 <code>vm</code> 和 <code>viz_root</code> (AST 可视化树)；若失败则设置 <code>compile_error</code> 并在界面显示红色报错。

4.2 系统代码

以下代码段自动加载自 `src` 目录下的源文件。这将占据报告的大部分篇幅。

4.2.1 符号表 (src/symbol_table.rs)

```
use crate::types::Symbol;
use std::collections::HashMap;

#[derive(Debug, Clone)]
pub struct Scope {
    pub symbols: HashMap<String, Symbol>,
    pub parent: Option<usize>,
    pub children: Vec<usize>,
}

impl Scope {
    pub fn new(parent: Option<usize>) -> Self {
        Self {
            symbols: HashMap::new(),
            parent,
            children: Vec::new(),
        }
    }
}

#[derive(Clone)]
pub struct SymbolTable {
    pub scopes: Vec<Scope>,
    pub current_scope_id: usize,
}

impl SymbolTable {
    pub fn new() -> Self {
        let root = Scope::new(None);
        Self {
            scopes: vec![root],
            current_scope_id: 0,
        }
    }

    pub fn create_scope(&mut self) -> usize {
        let new_id = self.scopes.len();
        let new_scope = Scope::new(Some(self.current_scope_id));
        self.scopes.push(new_scope);

        // Add as child to current scope
        self.scopes[self.current_scope_id].children.push(new_id);

        new_id
    }

    pub fn enter_scope(&mut self, id: usize) {
        if id < self.scopes.len() {
            self.current_scope_id = id;
        } else {
            panic!("Scope ID {} out of bounds", id);
        }
    }
}
```

```

pub fn exit_scope(&mut self) {
    if let Some(parent) = self.scopes[self.current_scope_id].parent {
        self.current_scope_id = parent;
    } else {
        panic!("Cannot exit global scope");
    }
}

pub fn define(&mut self, symbol: Symbol) -> Result<(), String> {
    let scope = &mut self.scopes[self.current_scope_id];
    if scope.symbols.contains_key(&symbol.name) {
        return Err(format!(
            "Symbol '{}' already defined in current scope",
            symbol.name
        ));
    }
    scope.symbols.insert(symbol.name.clone(), symbol);
    Ok(())
}

pub fn resolve(&self, name: &str) -> Option<&Symbol> {
    let mut current = self.current_scope_id;
    loop {
        let scope = &self.scopes[current];
        if let Some(symbol) = scope.symbols.get(name) {
            return Some(symbol);
        }
        if let Some(parent) = scope.parent {
            current = parent;
        } else {
            break;
        }
    }
    None
}

pub fn current_level(&self) -> usize {
    let mut level = 0;
    let mut current = self.current_scope_id;
    while let Some(parent) = self.scopes[current].parent {
        level += 1;
        current = parent;
    }
    level
}
}

```

4.2.2 词法分析器 (src/lexer.rs)

```

use crate::types::TokenType;
use std::iter::Peekable;
use std::str::Chars;

pub struct Lexer<'a> {

```

```

    input: Peekable<Chars<'a>>,
    pub current_token: TokenType,
    pub line: usize,
    pub col: usize,
    pub token_line: usize,
    pub token_col: usize,
}

impl<'a> Lexer<'a> {
    pub fn new(input: &'a str) -> Self {
        let mut lexer = Self {
            input: input.chars().peekable(),
            current_token: TokenType::Unknown,
            line: 1,
            col: 1,
            token_line: 1,
            token_col: 1,
        };
        lexer.next_token(); // Prime the first token
        lexer
    }

    fn read_char(&mut self) -> Option<char> {
        let c = self.input.next()?;
        if c == '\n' {
            self.line += 1;
            self.col = 1;
        } else {
            self.col += 1;
        }
        Some(c)
    }

    pub fn next_token(&mut self) {
        self.skip_whitespace();

        self.token_line = self.line;
        self.token_col = self.col;

        if let Some(&c) = self.input.peek() {
            match c {
                'a'..'z' | 'A'..'Z' => self.scan_identifier_or_keyword(),
                '0'..'9' => self.scan_number(),
                '+' => {
                    self.read_char();
                    self.current_token = TokenType::Plus;
                }
                '-' => {
                    self.read_char();
                    self.current_token = TokenType::Minus;
                }
                '*' => {
                    self.read_char();
                    self.current_token = TokenType::Multiply;
                }
            }
        }
    }
}

```

```

'/' => {
    self.read_char();
    self.current_token = TokenType::Divide;
}
'=' => {
    self.read_char();
    self.current_token = TokenType::Equals;
}
'#' => {
    self.read_char();
    self.current_token = TokenType::Hash;
}
'<' => {
    self.read_char();
    if let Some(&'=')=self.input.peek() {
        self.read_char();
        self.current_token = TokenType::LessEqual;
    } else if let Some(&'>')=self.input.peek() {
        self.read_char();
        self.current_token = TokenType::Hash; // Using Hash for
        <> (not equal)
    } else {
        self.current_token = TokenType::LessThan;
    }
}
'>' => {
    self.read_char();
    if let Some(&'=')=self.input.peek() {
        self.read_char();
        self.current_token = TokenType::GreaterEqual;
    } else {
        self.current_token = TokenType::GreaterThan;
    }
}
':' => {
    self.read_char();
    if let Some(&'=')=self.input.peek() {
        self.read_char();
        self.current_token = TokenType::Assignment;
    } else {
        self.current_token = TokenType::Unknown; // Single ':' is
        not valid in PL/0
    }
}
'(' => {
    self.read_char();
    self.current_token = TokenType::LParen;
}
')' => {
    self.read_char();
    self.current_token = TokenType::RParen;
}
',' => {
    self.read_char();
    self.current_token = TokenType::Comma;
}

```

```

        }
        ';' => {
            self.read_char();
            self.current_token = TokenType::Semicolon;
        }
        '.' => {
            self.read_char();
            self.current_token = TokenType::Period;
        }
        '-' => {
            self.read_char();
            self.current_token = TokenType::Unknown;
        }
    }
} else {
    self.current_token = TokenType::Eof;
}
}

fn skip_whitespace(&mut self) {
    while let Some(&c) = self.input.peek() {
        if c.is_whitespace() {
            self.read_char();
        } else {
            break;
        }
    }
}

fn scan_identifier_or_keyword(&mut self) {
    let mut ident = String::new();
    while let Some(&c) = self.input.peek() {
        if c.is_alphanumeric() || c == '_' {
            ident.push(c);
            self.read_char();
        } else {
            break;
        }
    }

    self.current_token = match ident.as_str() {
        "program" => TokenType::Program,
        "const" => TokenType::Const,
        "var" => TokenType::Var,
        "procedure" => TokenType::Procedure,
        "begin" => TokenType::Begin,
        "end" => TokenType::End,
        "if" => TokenType::If,
        "then" => TokenType::Then,
        "else" => TokenType::Else,
        "while" => TokenType::While,
        "do" => TokenType::Do,
        "call" => TokenType::Call,
        "read" => TokenType::Read,
        "write" => TokenType::Write,
    }
}

```

```

        "odd" => TokenType::Odd,
        _ => TokenType::Identifier(ident),
    );
}

fn scan_number(&mut self) {
    let mut num_str = String::new();
    while let Some(&c) = self.input.peek() {
        if c.is_ascii_digit() {
            num_str.push(c);
            self.read_char();
        } else {
            break;
        }
    }
    if let Ok(num) = num_str.parse::<i64>() {
        self.current_token = TokenType::Number(num);
    } else {
        self.current_token = TokenType::Unknown; // Overflow or error
    }
}
}

```

4.2.3 类型定义 (src/types.rs)

```

use serde::{Deserialize, Serialize};

#[derive(Debug, Clone, PartialEq)]
pub enum TokenType {
    // Keywords
    Const,
    Var,
    Procedure,
    Program,
    Begin,
    End,
    If,
    Then,
    Else,
    While,
    Do,
    Call,
    Read,
    Write,
    Odd,
    // Operators
    Plus,
    Minus,
    Multiply,
    Divide,
    Equals,
    Hash,
    LessThan,
    LessEqual,
}

```

```

GreaterThan,
GreaterEqual,
Assignment,
// Delimiters
Comma,
Semicolon,
Period,
LParen,
RParen,
// Literals and Identifiers
Identifier(String),
Number(i64),
// Special
Unknown,
Eof,
}

#[derive(Debug, Clone, Copy, PartialEq, Serialize, Deserialize)]
pub enum OpCode {
    LIT,
    OPR,
    LOD,
    STO,
    CAL,
    INT,
    JMP,
    JPC,
    RED,
    WRT,
}
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash, Serialize, Deserialize)]
#[repr(i64)]
pub enum Operator {
    RET = 0,
    NEG = 1,
    ADD = 2,
    SUB = 3,
    MUL = 4,
    DIV = 5,
    ODD = 6,
    EQL = 8,
    NEQ = 9,
    LSS = 10,
    GEQ = 11,
    GTR = 12,
    LEQ = 13,
    WRT = 14,
    WRL = 15,
    RED = 16,
}
impl Operator {
    pub fn from_i64(val: i64) -> Option<Self> {
        match val {

```

```

        0 => Some(Operator::RET),
        1 => Some(Operator::NEG),
        2 => Some(Operator::ADD),
        3 => Some(Operator::SUB),
        4 => Some(Operator::MUL),
        5 => Some(Operator::DIV),
        6 => Some(Operator::ODD),
        8 => Some(Operator::EQL),
        9 => Some(Operator::NEQ),
        10 => Some(Operator::LSS),
        11 => Some(Operator::GEQ),
        12 => Some(Operator::GTR),
        13 => Some(Operator::LEQ),
        14 => Some(Operator::WRT),
        15 => Some(Operator::WRL),
        16 => Some(Operator::RED),
        _ => None,
    }
}
}

#[derive(Debug, Clone, Copy, PartialEq, Serialize, Deserialize)]
pub struct Instruction {
    pub f: OpCode,
    pub l: usize, // Level difference
    pub a: i64,   // Argument/Address
}

impl Instruction {
    pub fn new(f: OpCode, l: usize, a: i64) -> Self {
        Self { f, l, a }
    }
}

#[derive(Debug, Clone)]
pub enum SymbolType {
    Constant { val: i64 },
    Variable { level: usize, addr: i64 },
    Procedure { level: usize, addr: i64 },
}

#[derive(Debug, Clone)]
pub struct Symbol {
    pub name: String,
    pub kind: SymbolType,
}

```

4.2.4 抽象语法树 (src/ast.rs)

```

use crate::types::Operator;

#[derive(Debug, Clone)]
pub struct Program {
    pub block: Block,
}

```

```

}

#[derive(Debug, Clone)]
pub struct Block {
    pub consts: Vec<ConstDecl>,
    pub vars: Vec<String>,
    pub procedures: Vec<ProcedureDecl>,
    pub statement: Statement,
    pub scope_id: Option<usize>,
}

#[derive(Debug, Clone)]
pub struct ConstDecl {
    pub name: String,
    pub value: i64,
}

#[derive(Debug, Clone)]
pub struct ProcedureDecl {
    pub name: String,
    pub params: Vec<String>,
    pub block: Block,
}

#[derive(Debug, Clone)]
pub enum Statement {
    Assignment {
        name: String,
        expr: Expr,
    },
    Call {
        name: String,
        args: Vec<Expr>,
    },
    BeginEnd {
        statements: Vec<Statement>,
    },
    If {
        condition: Condition,
        then_stmt: Box<Statement>,
        else_stmt: Option<Box<Statement>>,
    },
    While {
        condition: Condition,
        body: Box<Statement>,
    },
    Read {
        names: Vec<String>,
    },
    Write {
        exprs: Vec<Expr>,
    },
    Empty,
}

```

```

#[derive(Debug, Clone, PartialEq, Eq, Hash)]
pub enum Condition {
    Odd {
        expr: Expr,
    },
    Compare {
        left: Expr,
        op: Operator,
        right: Expr,
    },
}
}

#[derive(Debug, Clone, PartialEq, Eq, Hash)]
pub enum Expr {
    Binary {
        left: Box<Expr>,
        op: Operator,
        right: Box<Expr>,
    },
    Unary {
        op: Operator,
        expr: Box<Expr>,
    }, // For unary minus
    Number(i64),
    Identifier(String),
}

```

4.2.5 语法分析器 (src/parser.rs)

```

use crate::ast::*;
use crate::lexer::Lexer;
use crate::types::{Operator, TokenType};

#[derive(Debug)]
pub struct ParseError {
    pub line: usize,
    pub col: usize,
    pub message: String,
}

pub struct Parser<'a> {
    lexer: Lexer<'a>,
    pub errors: Vec<ParseError>,
    verbose: bool,
}

type ParseResult<T> = Result<T, ()>

impl<'a> Parser<'a> {
    pub fn new(lexer: Lexer<'a>, verbose: bool) -> Self {
        Self {
            lexer,
            errors: Vec::new(),
            verbose,
        }
    }
}

```

```

        }

    fn error(&mut self, msg: &str) -> ParseResult<()> {
        self.errors.push(ParseError {
            line: self.lexer.token_line,
            col: self.lexer.token_col,
            message: msg.to_string(),
        });
        Err(())
    }

    fn next(&mut self) {
        if self.verbose {
            println!("Token: {:?}", self.lexer.current_token);
        }
        self.lexer.next_token();
    }

    fn expect(&mut self, token: TokenType) -> ParseResult<()> {
        if self.lexer.current_token == token {
            self.next();
            Ok(())
        } else {
            let msg = format!("Expected {:?}, found {:?}", token,
self.lexer.current_token);
            self.error(&msg)
        }
    }

    pub fn parse(&mut self) -> ParseResult<Program> {
        self.program()
    }

    fn program(&mut self) -> ParseResult<Program> {
        if self.lexer.current_token == TokenType::Program {
            self.next();
            if let TokenType::Identifier(_) = self.lexer.current_token {
                self.next();
            } else {
                self.error("Expected program name")?;
                return Err(());
            }
            self.expect(TokenType::Semicolon)?;
        } else {
            self.error("Expected 'program'")?;
            return Err(());
        }

        let block = self.block()?;
        Ok(Program { block })
    }

    fn block(&mut self) -> ParseResult<Block> {
        let mut consts = Vec::new();

```

```

let mut vars = Vec::new();
let mut procedures = Vec::new();

if self.lexer.current_token == TokenType::Const {
    consts = self.const_decl()?;
}

if self.lexer.current_token == TokenType::Var {
    vars = self.var_decl()?;
}

while self.lexer.current_token == TokenType::Procedure {
    procedures.push(self.proc_decl()?);
}

let statement = self.statement()?;

Ok(Block {
    consts,
    vars,
    procedures,
    statement,
    scope_id: None,
})
}

fn const_decl(&mut self) -> ParseResult<Vec<ConstDecl>> {
    let mut consts = Vec::new();
    self.next(); // consume 'const'
    loop {
        if let TokenType::Identifier(name) = self.lexer.current_token.clone() {
            self.next();
            if self.lexer.current_token == TokenType::Assignment
                || self.lexer.current_token == TokenType::Equals
            {
                self.next();
            } else {
                self.error("Expected :=")?;
                return Err(());
            }

            if let TokenType::Number(val) = self.lexer.current_token {
                consts.push(ConstDecl { name, value: val });
                self.next();
            } else {
                self.error("Expected number")?;
                return Err(());
            }
        } else {
            self.error("Expected identifier")?;
            return Err(());
        }

        if self.lexer.current_token == TokenType::Comma {

```

```

        self.next();
    } else {
        break;
    }
}
self.expect(TokenType::Semicolon)?;
Ok(consts)
}

fn var_decl(&mut self) -> ParseResult<Vec<String>> {
    let mut vars = Vec::new();
    self.next(); // consume 'var'
    loop {
        if let TokenType::Identifier(name) = self.lexer.current_token.clone()
        {
            vars.push(name);
            self.next();
        } else {
            self.error("Expected identifier")?;
            return Err(());
        }

        if self.lexer.current_token == TokenType::Comma {
            self.next();
        } else {
            break;
        }
    }
    self.expect(TokenType::Semicolon)?;
    Ok(vars)
}

fn proc_decl(&mut self) -> ParseResult<ProcedureDecl> {
    self.next(); // consume 'procedure'
    let name = if let TokenType::Identifier(name) =
self.lexer.current_token.clone() {
        self.next();
        name
    } else {
        self.error("Expected identifier")?;
        return Err(());
    };

    let mut params = Vec::new();
    if self.lexer.current_token == TokenType::LParen {
        self.next();
        loop {
            if let TokenType::Identifier(param_name) =
self.lexer.current_token.clone() {
                params.push(param_name);
                self.next();
            } else {
                self.error("Expected parameter name")?;
                return Err(());
            }
        }
    }
}

```

```

        if self.lexer.current_token == TokenType::Comma {
            self.next();
        } else {
            break;
        }
    }
    self.expect(TokenType::RParen)?;
}

self.expect(TokenType::Semicolon)?;
let block = self.block()?;
self.expect(TokenType::Semicolon)?;

Ok(ProcedureDecl {
    name,
    params,
    block,
})
}

fn is_start_of_statement(&self) -> bool {
    matches!(
        self.lexer.current_token,
        TokenType::Identifier(_)
            | TokenType::Call
            | TokenType::Begin
            | TokenType::If
            | TokenType::While
            | TokenType::Read
            | TokenType::Write
    )
}

fn synchronize(&mut self) {
    while self.lexer.current_token != TokenType::Eof {
        if self.lexer.current_token == TokenType::Semicolon {
            return;
        }
        if self.is_start_of_statement() {
            return;
        }
        if self.lexer.current_token == TokenType::End {
            return;
        }
        self.next();
    }
}

fn statement(&mut self) -> ParseResult<Statement> {
    match self.lexer.current_token.clone() {
        TokenType::Identifier(name) => {
            self.next();
            if self.lexer.current_token == TokenType::Assignment {
                self.next();

```

```

        let expr = self.expression()?;
        Ok(Statement::Assignment { name, expr })
    } else {
        self.error("Expected :=")?;
        Err(())
    }
}
TokenType::Call => {
    self.next();
    if let TokenType::Identifier(name) =
self.lexer.current_token.clone() {
        self.next();
        let mut args = Vec::new();
        if self.lexer.current_token == TokenType::LParen {
            self.next();
            loop {
                args.push(self.expression());
                if self.lexer.current_token == TokenType::Comma {
                    self.next();
                } else {
                    break;
                }
            }
            self.expect(TokenType::RParen)?;
        }
        Ok(Statement::Call { name, args })
    } else {
        self.error("Expected identifier")?;
        Err(())
    }
}
TokenType::Begin => {
    self.next();
    let mut statements = Vec::new();

    loop {
        if self.lexer.current_token == TokenType::End {
            break;
        }
        if self.lexer.current_token == TokenType::Eof {
            self.error("Expected 'end'")?;
            return Err(());
        }

        match self.statement() {
            Ok(stmt) => {
                if !matches!(stmt, Statement::Empty) {
                    statements.push(stmt);
                } else {
                    if self.lexer.current_token !=

TokenType::Semicolon
                        && self.lexer.current_token != TokenType::End
{
                    self.errors.push(ParseError {
                        line: self.lexer.token_line,

```

```

        col: self.lexer.token_col,
        message: format!(
            "Unexpected token: {:?}", self.lexer.current_token
        ),
    });
    self.synchronize();
}
}
}
Err(_) => {
    self.synchronize();
}
}

if self.lexer.current_token == TokenType::Semicolon {
    self.next();
} else if self.lexer.current_token == TokenType::End {
    break;
} else {
    if self.is_start_of_statement() {
        self.errors.push(ParseError {
            line: self.lexer.token_line,
            col: self.lexer.token_col,
            message: "Expected ';'".to_string(),
        });
    } else if self.lexer.current_token != TokenType::Eof {
        // If we haven't already synchronized (which we would
        have if statement was Empty and invalid)
        // We might be here if statement was valid but
        followed by garbage.
        self.errors.push(ParseError {
            line: self.lexer.token_line,
            col: self.lexer.token_col,
            message: format!(
                "Unexpected token: {:?}", self.lexer.current_token
            ),
        });
        self.synchronize();
    }
}
self.expect(TokenType::End)?;
Ok(Statement::BeginEnd { statements })
}
TokenType::If => {
    self.next();
    let condition = self.condition()?;
    self.expect(TokenType::Then)?;
    let then_stmt = Box::new(self.statement()?;
    let else_stmt = if self.lexer.current_token == TokenType::Else {
        self.next();
        Some(Box::new(self.statement()?))
    } else {

```

```

        None
    };
Ok(Statement::If {
    condition,
    then_stmt,
    else_stmt,
})
}
TokenType::While => {
    self.next();
    let condition = self.condition()?;
    self.expect(TokenType::Do)?;
    let body = Box::new(self.statement()?)?;
    Ok(Statement::While { condition, body })
}
TokenType::Read => {
    self.next();
    let mut names = Vec::new();
    if self.lexer.current_token == TokenType::LParen {
        self.next();
        loop {
            if let TokenType::Identifier(name) =
self.lexer.current_token.clone() {
                names.push(name);
                self.next();
            } else {
                self.error("Expected identifier")?;
                return Err(());
            }
            if self.lexer.current_token == TokenType::Comma {
                self.next();
            } else {
                break;
            }
        }
        self.expect(TokenType::RParen)?;
    } else {
        if let TokenType::Identifier(name) =
self.lexer.current_token.clone() {
            names.push(name);
            self.next();
        } else {
            self.error("Expected identifier or '('")?;
            return Err(());
        }
    }
    Ok(Statement::Read { names })
}
TokenType::Write => {
    self.next();
    let mut exprs = Vec::new();
    if self.lexer.current_token == TokenType::LParen {
        self.next();
        loop {
            exprs.push(self.expression()?);

```

```

        if self.lexer.current_token == TokenType::Comma {
            self.next();
        } else {
            break;
        }
    }
    self.expect(TokenType::RParen)?;
} else {
    exprs.push(self.expression()?);
}
Ok(Statement::Write { exprs })
}
_ => Ok(Statement::Empty),
}
}

fn condition(&mut self) -> ParseResult<Condition> {
    if self.lexer.current_token == TokenType::Odd {
        self.next();
        let expr = self.expression()?;
        Ok(Condition::Odd { expr })
    } else {
        let left = self.expression()?;
        let op = match self.lexer.current_token {
            TokenType::Equals => Operator::EQ,
            TokenType::Hash => Operator::NEQ,
            TokenType::LessThan => Operator::LSS,
            TokenType::LessEqual => Operator::LEQ,
            TokenType::GreaterThan => Operator::GTR,
            TokenType::GreaterEqual => Operator::GEQ,
            _ => {
                self.error("Expected comparison operator")?;
                return Err(());
            }
        };
        self.next();
        let right = self.expression()?;
        Ok(Condition::Compare { left, op, right })
    }
}

fn expression(&mut self) -> ParseResult<Expr> {
    let mut expr = if self.lexer.current_token == TokenType::Plus {
        self.next();
        self.term()?;
    } else if self.lexer.current_token == TokenType::Minus {
        self.next();
        Expr::Unary {
            op: Operator::NEG,
            expr: Box::new(self.term()?),  

        }
    } else {
        self.term()?;
    };
}

```

```

loop {
    match self.lexer.current_token {
        TokenType::Plus => {
            self.next();
            let right = self.term()?;
            expr = Expr::Binary {
                left: Box::new(expr),
                op: Operator::ADD,
                right: Box::new(right),
            };
        }
        TokenType::Minus => {
            self.next();
            let right = self.term()?;
            expr = Expr::Binary {
                left: Box::new(expr),
                op: Operator::SUB,
                right: Box::new(right),
            };
        }
        _ => break,
    }
}
Ok(expr)
}

fn term(&mut self) -> ParseResult<Expr> {
    let mut expr = self.factor()?;
    loop {
        match self.lexer.current_token {
            TokenType::Multiply => {
                self.next();
                let right = self.factor()?;
                expr = Expr::Binary {
                    left: Box::new(expr),
                    op: Operator::MUL,
                    right: Box::new(right),
                };
            }
            TokenType::Divide => {
                self.next();
                let right = self.factor()?;
                expr = Expr::Binary {
                    left: Box::new(expr),
                    op: Operator::DIV,
                    right: Box::new(right),
                };
            }
            _ => break,
        }
    }
}
Ok(expr)
}

fn factor(&mut self) -> ParseResult<Expr> {

```

```

        match self.lexer.current_token.clone() {
            TokenType::Identifier(name) => {
                self.next();
                Ok(Expr::Identifier(name))
            }
            TokenType::Number(val) => {
                self.next();
                Ok(Expr::Number(val))
            }
            TokenType::LParen => {
                self.next();
                let expr = self.expression()?;
                self.expect(TokenType::RParen)?;
                Ok(expr)
            }
            _ => {
                self.error("Expected identifier, number, or '('?");
                Err(())
            }
        }
    }
}

```

4.2.6 语义分析器 (src/semantic.rs)

```

use crate::ast::*;
use crate::symbol_table::SymbolTable;
use crate::types::{Symbol, SymbolType};

pub struct SemanticAnalyzer<'a> {
    symbol_table: &'a mut SymbolTable,
    errors: Vec<String>,
}

impl<'a> SemanticAnalyzer<'a> {
    pub fn new(symbol_table: &'a mut SymbolTable) -> Self {
        Self {
            symbol_table,
            errors: Vec::new(),
        }
    }

    pub fn analyze(&mut self, program: &mut Program) -> Result<(), Vec<String>> {
        // Root scope is already created (id 0)
        // We associate the main block with scope 0
        program.block.scope_id = Some(0);

        // We don't need to create a new scope for the main block because
        SymbolTable::new() creates one.
        // But we need to make sure we are in it.
        self.symbol_table.current_scope_id = 0;

        self.analyze_block(&mut program.block, 0)?;
    }
}

```

```

        if self.errors.is_empty() {
            Ok(())
        } else {
            Err(self.errors.clone())
        }
    }

    fn analyze_block(&mut self, block: &mut Block, level: usize) -> Result<(), Vec<String>> {
        // Declare constants
        for const_decl in &block.consts {
            if let Err(e) = self.symbol_table.define(Symbol {
                name: const_decl.name.clone(),
                kind: SymbolType::Constant {
                    val: const_decl.value,
                },
            }) {
                self.errors.push(e);
            }
        }

        // Declare variables
        let mut var_offset = 3; // SL, DL, RA
        for var_name in &block.vars {
            if let Err(e) = self.symbol_table.define(Symbol {
                name: var_name.clone(),
                kind: SymbolType::Variable {
                    level,
                    addr: var_offset,
                },
            }) {
                self.errors.push(e);
            }
            var_offset += 1;
        }

        // Declare procedures
        for proc_decl in &mut block.procedures {
            // Define procedure in current scope
            // Address will be resolved during codegen or we can assign a
            placeholder?
            // Codegen calculates address based on code length. We can't know it
            here easily without generating code.
            // However, for recursive calls, we need to know it exists.
            // We can store a placeholder addr and update it later, or just store
            that it is a procedure.
            // The current SymbolType::Procedure has an addr field.
            // Let's set it to -1 or 0 and let Codegen update it?
            // Or better: Codegen will update the symbol table with the real
            address!
            // But Semantic Analysis needs to check calls.

            if let Err(e) = self.symbol_table.define(Symbol {
                name: proc_decl.name.clone(),
                kind: SymbolType::Procedure {

```

```

        level,
        addr: 0, // Placeholder, updated in Codegen
    },
}) {
    self.errors.push(e);
}
}

// Now analyze procedure bodies
for proc_decl in &mut block.procedures {
    let new_scope_id = self.symbol_table.create_scope();
    proc_decl.block.scope_id = Some(new_scope_id);
    self.symbol_table.enter_scope(new_scope_id);

    // Define parameters
    let param_count = proc_decl.params.len();
    for (i, param_name) in proc_decl.params.iter().enumerate() {
        let offset = -((param_count - i) as i64);
        if let Err(e) = self.symbol_table.define(Symbol {
            name: param_name.clone(),
            kind: SymbolType::Variable {
                level: level + 1,
                addr: offset,
            },
        }) {
            self.errors.push(e);
        }
    }
}

self.analyze_block(&mut proc_decl.block, level + 1)?;
self.symbol_table.exit_scope();
}

self.analyze_statement(&block.statement)?;

Ok(())
}

fn analyze_statement(&mut self, stmt: &Statement) -> Result<(), Vec<String>>
{
    match stmt {
        Statement::Assignment { name, expr } => {
            match self.symbol_table.resolve(name) {
                Some(sym) => match sym.kind {
                    SymbolType::Constant { .. } => {
                        self.errors
                            .push(format!("Cannot assign to constant '{}'", name));
                    }
                    SymbolType::Procedure { .. } => {
                        self.errors
                            .push(format!("Cannot assign to procedure '{}'", name));
                    }
                    SymbolType::Variable { .. } => {}
                }
            }
        }
    }
}

```

```

        },
        None => {
            self.errors.push(format!("Undefined variable '{}',",
name));
        }
    }
    self.analyze_expr(expr)?;
}
Statement::Call { name, args } => {
    match self.symbol_table.resolve(name) {
        Some(sym) => {
            match sym.kind {
                SymbolType::Procedure { .. } => {
                    // Check arg count if we had that info in
SymbolType
                    // Current SymbolType::Procedure doesn't store
param count.
                    // We should probably add it to SymbolType for
better checking.
                }
                _ => {
                    self.errors.push(format!("'{}' is not a
procedure", name));
                }
            }
        }
        None => {
            self.errors.push(format!("Undefined procedure '{}',",
name));
        }
    }
    for arg in args {
        self.analyze_expr(arg)?;
    }
}
Statement::BeginEnd { statements } => {
    for s in statements {
        self.analyze_statement(s)?;
    }
}
Statement::If {
    condition,
    then_stmt,
    else_stmt,
} => {
    self.analyze_condition(condition)?;
    self.analyze_statement(then_stmt)?;
    if let Some(s) = else_stmt {
        self.analyze_statement(s)?;
    }
}
Statement::While { condition, body } => {
    self.analyze_condition(condition)?;
    self.analyze_statement(body)?;
}

```

```

Statement::Read { names } => {
    for name in names {
        match self.symbol_table.resolve(name) {
            Some(sym) => {
                if let SymbolType::Constant { .. } = sym.kind {
                    self.errors
                        .push(format!("Cannot read into constant
'{:}', name));
                }
                if let SymbolType::Procedure { .. } = sym.kind {
                    self.errors
                        .push(format!("Cannot read into procedure
'{:}', name));
                }
            }
            None => {
                self.errors.push(format!("Undefined variable '{:}', name));
            }
        }
    }
}
Statement::Write { exprs } => {
    for expr in exprs {
        self.analyze_expr(expr)?;
    }
}
Statement::Empty => {}
Ok(())
}

fn analyze_expr(&mut self, expr: &Expr) -> Result<(), Vec<String>> {
    match expr {
        Expr::Number(_) => {}
        Expr::Identifier(name) => {
            if self.symbol_table.resolve(name).is_none() {
                self.errors.push(format!("Undefined identifier '{:}', name));
            }
        }
        Expr::Binary { left, right, .. } => {
            self.analyze_expr(left)?;
            self.analyze_expr(right)?;
        }
        Expr::Unary { expr, .. } => {
            self.analyze_expr(expr)?;
        }
    }
    Ok(())
}

fn analyze_condition(&mut self, cond: &Condition) -> Result<(), Vec<String>>
{
    match cond {
        Condition::Odd { expr } => self.analyze_expr(expr),
}

```

```

        Condition::Compare { left, right, .. } => {
            self.analyze_expr(left)?;
            self.analyze_expr(right)
        }
    }
}
}

```

4.2.7 优化器 (src/optimizer.rs)

```

use crate::ast::*;
use crate::types::Operator;
use std::collections::HashMap;
use std::collections::HashSet;

pub fn optimize_ast(program: &mut Program) {
    optimize_block(&mut program.block);
}

fn optimize_block(block: &mut Block) {
    for proc in &mut block.procedures {
        optimize_block(&mut proc.block);
    }
    optimize_statement(&mut block.statement);
}

fn optimize_statement(stmt: &mut Statement) {
    match stmt {
        Statement::Assignment { expr, .. } => optimize_expr(expr),
        Statement::Call { args, .. } => {
            for arg in args {
                optimize_expr(arg);
            }
        }
        Statement::BeginEnd { statements } => {
            // 1. Optimize children
            for s in statements.iter_mut() {
                optimize_statement(s);
            }

            // 2. DAG / CSE Optimization
            optimize_block_dag(statements);

            // 3. Filter Empty
            let mut new_statements = Vec::new();
            for s in statements.iter() {
                if !matches!(s, Statement::Empty) {
                    new_statements.push(s.clone());
                }
            }
            *statements = new_statements;
        }
        Statement::If {
            condition,

```

```

        then_stmt,
        else_stmt,
    } => {
        optimize_condition(condition);
        optimize_statement(then_stmt);
        if let Some(s) = else_stmt {
            optimize_statement(s);
        }

        // Dead Code Elimination for If
        if let Some(val) = evaluate_condition(condition) {
            if val {
                *stmt = *then_stmt.clone();
            } else if let Some(else_s) = else_stmt {
                *stmt = *else_s.clone();
            } else {
                *stmt = Statement::Empty;
            }
        }
    }
Statement::While { condition, body } => {
    optimize_condition(condition);
    optimize_statement(body);

    // Dead Code Elimination for While
    if let Some(val) = evaluate_condition(condition) {
        if !val {
            *stmt = Statement::Empty;
        } else {
            // Loop Invariant Code Motion
            try_licm(stmt);
        }
    } else {
        // Loop Invariant Code Motion
        try_licm(stmt);
    }
}
Statement::Read { .. } => {}
Statement::Write { exprs } => {
    for expr in exprs {
        optimize_expr(expr);
    }
}
Statement::Empty => {}
}

fn evaluate_condition(cond: &Condition) -> Option<bool> {
    match cond {
        Condition::Odd { expr } => {
            if let Expr::Number(val) = expr {
                Some(val % 2 != 0)
            } else {
                None
            }
        }
    }
}

```

```

    }
    Condition::Compare { left, op, right } => {
        if let (Expr::Number(l), Expr::Number(r)) = (left, right) {
            match op {
                Operator::EQ => Some(l == r),
                Operator::NEQ => Some(l != r),
                Operator::LSS => Some(l < r),
                Operator::LEQ => Some(l <= r),
                Operator::GTR => Some(l > r),
                Operator::GEQ => Some(l >= r),
                _ => None,
            }
        } else {
            None
        }
    }
}

fn optimize_condition(cond: &mut Condition) {
    match cond {
        Condition::Odd { expr } => optimize_expr(expr),
        Condition::Compare { left, right, .. } => {
            optimize_expr(left);
            optimize_expr(right);
        }
    }
}

fn optimize_block_dag(statements: &mut Vec<Statement>) {
    let mut available_exprs: HashMap<Expr, String> = HashMap::new();

    for stmt in statements.iter_mut() {
        match stmt {
            Statement::Assignment { name, expr } => {
                // 1. CSE
                let mut replaced = false;
                if let Some(var_name) = available_exprs.get(expr) {
                    *expr = Expr::Identifier(var_name.clone());
                    replaced = true;
                }

                // 2. Invalidate
                available_exprs.retain(|k, _| !expr_uses_var(k, name));

                // 3. Add (if not replaced and complex)
                if !replaced && !matches!(expr, Expr::Number(_) |
                    Expr::Identifier(_)) {
                    if !expr_uses_var(expr, name) {
                        available_exprs.insert(expr.clone(), name.clone());
                    }
                }
            }
            Statement::Read { names } => {
                for name in names {

```

```

        available_expressions.retain(|k, _| !expr_uses_var(k, name));
    }
}
Statement::Call { .. } => {
    available_expressions.clear();
}
Statement::If { .. } | Statement::While { .. } | Statement::BeginEnd
{ .. } => {
    available_expressions.clear();
}
_ => {}
}
}
}

fn expr_uses_var(expr: &Expr, var: &str) -> bool {
    match expr {
        Expr::Binary { left, right, .. } => expr_uses_var(left, var) ||
expr_uses_var(right, var),
        Expr::Unary { expr, .. } => expr_uses_var(expr, var),
        Expr::Identifier(name) => name == var,
        _ => false,
    }
}

fn try_licm(stmt: &mut Statement) {
    if let Statement::While { condition: _, body } = stmt {
        // 1. Collect modified vars in loop
        let mut modified = HashSet::new();
        collect_modified_vars(body, &mut modified);

        let mut invariant_stmts = Vec::new();

        match body.as_mut() {
            Statement::BeginEnd { statements } => {
                let mut i = 0;
                while i < statements.len() {
                    let mut hoist = false;
                    if let Statement::Assignment { name: _, expr } =
&statements[i] {
                        if !expr_depends_on(expr, &modified) {
                            hoist = true;
                        }
                    }

                    if hoist {
                        invariant_stmts.push(statements.remove(i));
                    } else {
                        i += 1;
                    }
                }
            }
            Statement::Assignment { name: _, expr } => {
                if !expr_depends_on(expr, &modified) {
                    invariant_stmts.push(std::mem::replace(body.as_mut(),

```

```

Statement::Empty));
        }
    }
    _ => {}
}

if !invariant_stmts.is_empty() {
    let loop_stmt = std::mem::replace(stmt, Statement::Empty);
    let mut new_block_stmts = invariant_stmts;
    new_block_stmts.push(loop_stmt);
    *stmt = Statement::BeginEnd {
        statements: new_block_stmts,
    };
}
}

fn collect_modified_vars(stmt: &Statement, modified: &mut HashSet<String>) {
    match stmt {
        Statement::Assignment { name, .. } => {
            modified.insert(name.clone());
        }
        Statement::Read { names } => {
            for n in names {
                modified.insert(n.clone());
            }
        }
        Statement::BeginEnd { statements } => {
            for s in statements {
                collect_modified_vars(s, modified);
            }
        }
        Statement::If {
            then_stmt,
            else_stmt,
            ..
        } => {
            collect_modified_vars(then_stmt, modified);
            if let Some(s) = else_stmt {
                collect_modified_vars(s, modified);
            }
        }
        Statement::While { body, .. } => {
            collect_modified_vars(body, modified);
        }
        _ => {}
    }
}

fn expr_depends_on(expr: &Expr, vars: &HashSet<String>) -> bool {
    match expr {
        Expr::Binary { left, right, .. } => {
            expr_depends_on(left, vars) || expr_depends_on(right, vars)
        }
        Expr::Unary { expr, .. } => expr_depends_on(expr, vars),
    }
}

```

```

        Expr::Identifier(name) => vars.contains(name),
        _ => false,
    }
}

fn optimize_expr(expr: &mut Expr) {
    match expr {
        Expr::Binary { left, op, right } => {
            optimize_expr(left);
            optimize_expr(right);

            // Constant folding
            if let (Expr::Number(l), Expr::Number(r)) = (left.as_ref(),
right.as_ref()) {
                let val = match op {
                    Operator::ADD => l + r,
                    Operator::SUB => l - r,
                    Operator::MUL => l * r,
                    Operator::DIV => {
                        if *r != 0 {
                            l / r
                        } else {
                            return;
                        }
                    } // Avoid div by zero
                    _ => return,
                };
                *expr = Expr::Number(val);
                return;
            }

            // Algebraic Simplification
            // x + 0 = x
            if *op == Operator::ADD {
                if let Expr::Number(0) = right.as_ref() {
                    *expr = *left.clone();
                    return;
                }
                if let Expr::Number(0) = left.as_ref() {
                    *expr = *right.clone();
                    return;
                }
            }
            // x - 0 = x
            if *op == Operator::SUB {
                if let Expr::Number(0) = right.as_ref() {
                    *expr = *left.clone();
                    return;
                }
            }
            // x * 1 = x, x * 0 = 0
            if *op == Operator::MUL {
                if let Expr::Number(1) = right.as_ref() {
                    *expr = *left.clone();
                    return;
                }
            }
        }
    }
}

```

```

        }
        if let Expr::Number(1) = left.as_ref() {
            *expr = *right.clone();
            return;
        }
        if let Expr::Number(0) = right.as_ref() {
            *expr = Expr::Number(0);
            return;
        }
        if let Expr::Number(0) = left.as_ref() {
            *expr = Expr::Number(0);
            return;
        }
    }
    // x / 1 = x
    if *op == Operator::DIV {
        if let Expr::Number(1) = right.as_ref() {
            *expr = *left.clone();
            return;
        }
    }
}
Expr::Unary { op, expr: inner } => {
    optimize_expr(inner);
    if let Expr::Number(val) = inner.as_ref() {
        if *op == Operator::NEG {
            *expr = Expr::Number(-val);
        }
    }
}
- => {}
}
}

```

4.2.8 代码生成器 (src/codegen.rs)

```

use crate::ast::*;
use crate::symbol_table::SymbolTable;
use crate::types::{Instruction, OpCode, Operator, SymbolType};

pub struct CodeGenerator {
    code: Vec<Instruction>,
    level: usize,
}

impl CodeGenerator {
    pub fn new() -> Self {
        Self {
            code: Vec::new(),
            level: 0,
        }
    }
}

pub fn generate(

```

```

    &mut self,
    program: &Program,
    symbol_table: &mut SymbolTable,
) -> Vec<Instruction> {
    // Ensure we start at root scope
    symbol_table.current_scope_id = 0;
    self.generate_block(&program.block, symbol_table);
    self.emit(OpCode::OPR, 0, Operator::RET as i64);
    self.code.clone()
}

fn emit(&mut self, f: OpCode, l: usize, a: i64) {
    self.code.push(Instruction::new(f, l, a));
}

fn generate_block(&mut self, block: &Block, symbol_table: &mut SymbolTable) {
    // Enter the scope associated with this block
    if let Some(scope_id) = block.scope_id {
        symbol_table.enter_scope(scope_id);
    } else {
        panic!("Block has no scope ID assigned");
    }

    let jmp_addr = self.code.len();
    self.emit(OpCode::JMP, 0, 0); // Placeholder

    // We don't need to declare constants or vars in symbol table, they are
    already there.
    // But we need to calculate var_offset for INT instruction.
    // We can count vars in the block.
    let var_count = block.vars.len();
    let var_offset = 3 + var_count;

    // Declare procedures
    for proc_decl in &block.procedures {
        let proc_addr = self.code.len();

        // Update procedure address in symbol table
        let scope = &mut symbol_table.scopes[symbol_table.current_scope_id];
        if let Some(sym) = scope.symbols.get_mut(&proc_decl.name) {
            if let SymbolType::Procedure { ref mut addr, .. } = sym.kind {
                *addr = proc_addr as i64;
            }
        }

        self.level += 1;
        self.generate_block(&proc_decl.block, symbol_table);
        self.level -= 1;

        self.emit(OpCode::OPR, 0, Operator::RET as i64);
    }

    // Fix JMP
    self.code[jmp_addr].a = self.code.len() as i64;
}

```

```

// Allocate space
self.emit(OpCode::INT, 0, var_offset as i64);

self.generate_statement(&block.statement, symbol_table);

if block.scope_id != Some(0) {
    symbol_table.exit_scope();
}
}

fn generate_statement(&mut self, stmt: &Statement, symbol_table: &mut
SymbolTable) {
    match stmt {
        Statement::Assignment { name, expr } => {
            self.generate_expr(expr, symbol_table);
            let sym = symbol_table.resolve(name).expect("Undefined
variable");
            match sym.kind {
                SymbolType::Variable { level, addr } => {
                    self.emit(OpCode::ST0, self.level - level, addr);
                }
                _ => panic!("Cannot assign to non-variable"),
            }
        }
        Statement::Call { name, args } => {
            for arg in args {
                self.generate_expr(arg, symbol_table);
            }

            let sym = symbol_table.resolve(name).expect("Undefined
procedure");
            match sym.kind {
                SymbolType::Procedure { level, addr } => {
                    self.emit(OpCode::CAL, self.level - level, addr);
                    if !args.is_empty() {
                        self.emit(OpCode::INT, 0, -(args.len() as i64));
                    }
                }
                _ => panic!("Not a procedure"),
            }
        }
        Statement::BeginEnd { statements } => {
            for s in statements {
                self.generate_statement(s, symbol_table);
            }
        }
        Statement::If {
            condition,
            then_stmt,
            else_stmt,
        } => {
            self.generate_condition(condition, symbol_table);
            let jpc_idx = self.code.len();
            self.emit(OpCode::JPC, 0, 0);
        }
    }
}

```

```

        self.generate_statement(then_stmt, symbol_table);

        if let Some(else_s) = else_stmt {
            let jmp_idx = self.code.len();
            self.emit(OpCode::JMP, 0, 0);
            self.code[jpc_idx].a = self.code.len() as i64;
            self.generate_statement(else_s, symbol_table);
            self.code[jmp_idx].a = self.code.len() as i64;
        } else {
            self.code[jpc_idx].a = self.code.len() as i64;
        }
    }

    Statement::While { condition, body } => {
        let start_idx = self.code.len();
        self.generate_condition(condition, symbol_table);
        let jpc_idx = self.code.len();
        self.emit(OpCode::JPC, 0, 0);

        self.generate_statement(body, symbol_table);
        self.emit(OpCode::JMP, 0, start_idx as i64);

        self.code[jpc_idx].a = self.code.len() as i64;
    }

    Statement::Read { names } => {
        for name in names {
            self.emit(OpCode::OPR, 0, Operator::RED as i64);
            let sym = symbol_table.resolve(name).expect("Undefined
variable");
            match sym.kind {
                SymbolType::Variable { level, addr } => {
                    self.emit(OpCode::STO, self.level - level, addr);
                }
                _ => panic!("Cannot read into non-variable"),
            }
        }
    }

    Statement::Write { exprs } => {
        for expr in exprs {
            self.generate_expr(expr, symbol_table);
            self.emit(OpCode::OPR, 0, Operator::WRT as i64);
        }
    }

    Statement::Empty => {}
}

fn generate_expr(&mut self, expr: &Expr, symbol_table: &mut SymbolTable) {
    match expr {
        Expr::Number(n) => {
            self.emit(OpCode::LIT, 0, *n);
        }
        Expr::Identifier(name) => {
            let sym = symbol_table.resolve(name).expect("Undefined
identifier");
            match sym.kind {

```

```

        SymbolType::Constant { val } => {
            self.emit(OpCode::LIT, 0, val);
        }
        SymbolType::Variable { level, addr } => {
            self.emit(OpCode::LOD, self.level - level, addr);
        }
        _ => panic!("Identifier is not a value"),
    }
}

Expr::Binary { left, op, right } => {
    self.generate_expr(left, symbol_table);
    self.generate_expr(right, symbol_table);
    self.emit(OpCode::OPR, 0, *op as i64);
}
Expr::Unary { op, expr } => {
    self.generate_expr(expr, symbol_table);
    self.emit(OpCode::OPR, 0, *op as i64);
}
}

fn generate_condition(&mut self, cond: &Condition, symbol_table: &mut
SymbolTable) {
    match cond {
        Condition::Odd { expr } => {
            self.generate_expr(expr, symbol_table);
            self.emit(OpCode::OPR, 0, Operator::ODD as i64);
        }
        Condition::Compare { left, op, right } => {
            self.generate_expr(left, symbol_table);
            self.generate_expr(right, symbol_table);
            self.emit(OpCode::OPR, 0, *op as i64);
        }
    }
}
}

```

4.2.9 虚拟机 (src/vm.rs)

```

use crate::types::{Instruction, OpCode, Operator};
use std::io::{self, Write};

#[derive(PartialEq, Debug, Clone)]
pub enum VMState {
    Running,
    Halted,
    WaitingForInput,
    Error(String),
}

pub struct VM {
    pub code: Vec<Instruction>, // CODE: Stores P-code
    pub stack: Vec<i64>, // STACK: Dynamic data space
    pub p: usize, // P: Program address register (PC)
}

```

```

pub b: usize,           // B: Base address register (BP)
pub t: usize,           // T: Top of stack register (SP)
pub i: Instruction,    // I: Instruction register
pub output: Vec<String>,
pub input_queue: Vec<i64>,
pub state: VMState,
pub instruction_count: usize,
}

impl VM {
    pub fn new(code: Vec<Instruction>) -> Self {
        Self {
            code,
            stack: vec![0; 1000], // Initial stack size
            p: 0,
            b: 0,
            t: 0,
            i: Instruction::new(OpCode::LIT, 0, 0), // Initial dummy instruction
            output: Vec::new(),
            input_queue: Vec::new(),
            state: VMState::Running,
            instruction_count: 0,
        }
    }

    fn base(&self, mut l: usize) -> usize {
        let mut b = self.b;
        while l > 0 {
            b = self.stack[b] as usize;
            l -= 1;
        }
        b
    }

    pub fn step(&mut self) {
        if self.state != VMState::Running {
            return;
        }

        if self.p >= self.code.len() {
            self.state = VMState::Error("PC out of bounds".to_string());
            return;
        }

        // Fetch instruction into I register
        self.i = self.code[self.p];
        self.p += 1;
        self.instruction_count += 1;

        let ir = self.i; // Use local alias for convenience matching original
        code structure

        match ir.f {
            OpCode::LIT => {
                self.stack[self.t] = ir.a;
            }
        }
    }
}

```

```

        self.t += 1;
    }
OpCode::OPR => {
    match Operator::from_i64(ir.a) {
        Some(Operator::RET) => {
            // RET
            self.t = self.b;
            self.p = self.stack[self.t + 2] as usize;
            self.b = self.stack[self.t + 1] as usize;
            if self.p == 0 {
                self.state = VMState::Halted;
            }
        }
        Some(Operator::NEG) => {
            // NEG
            self.stack[self.t - 1] = -self.stack[self.t - 1];
        }
        Some(Operator::ADD) => {
            // ADD
            self.t -= 1;
            self.stack[self.t - 1] += self.stack[self.t];
        }
        Some(Operator::SUB) => {
            // SUB
            self.t -= 1;
            self.stack[self.t - 1] -= self.stack[self.t];
        }
        Some(Operator::MUL) => {
            // MUL
            self.t -= 1;
            self.stack[self.t - 1] *= self.stack[self.t];
        }
        Some(Operator::DIV) => {
            // DIV
            self.t -= 1;
            if self.stack[self.t] == 0 {
                self.state = VMState::Error("Division by
zero".to_string());
                return;
            }
            self.stack[self.t - 1] /= self.stack[self.t];
        }
        Some(Operator::ODD) => {
            // ODD
            self.stack[self.t - 1] % 2;
        }
        Some(Operator::EQL) => {
            // EQL
            self.t -= 1;
            self.stack[self.t - 1] = if self.stack[self.t - 1] ==
self.stack[self.t] {
                1
            } else {
                0
            };
        }
    }
}

```

```

    }
    Some(Operator::NEQ) => {
        // NEQ
        self.t -= 1;
        self.stack[self.t - 1] = if self.stack[self.t - 1] !=

self.stack[self.t] {
    1
} else {
    0
};

}
Some(Operator::LSS) => {
    // LSS
    self.t -= 1;
    self.stack[self.t - 1] = if self.stack[self.t - 1] <

self.stack[self.t] {
    1
} else {
    0
};

}
Some(Operator::GEQ) => {
    // GEQ
    self.t -= 1;
    self.stack[self.t - 1] = if self.stack[self.t - 1] >=

self.stack[self.t] {
    1
} else {
    0
};

}
Some(Operator::GTR) => {
    // GTR
    self.t -= 1;
    self.stack[self.t - 1] = if self.stack[self.t - 1] >

self.stack[self.t] {
    1
} else {
    0
};

}
Some(Operator::LEQ) => {
    // LEQ
    self.t -= 1;
    self.stack[self.t - 1] = if self.stack[self.t - 1] <=

self.stack[self.t] {
    1
} else {
    0
};

}
Some(Operator::WRT) => {
    // Write stack top
    self.t -= 1;
    let val = self.stack[self.t];
}

```

```

        self.output.push(val.to_string());
    }
    Some(Operator::WRL) => {
        // Write newline
        self.output.push("\n".to_string());
    }
    Some(Operator::RED) => {
        // Read to stack top
        if let Some(val) = self.input_queue.pop() {
            self.stack[self.t] = val;
            self.t += 1;
        } else {
            self.p -= 1;
            self.state = VMState::WaitingForInput;
        }
    }
    None => {
        self.state = VMState::Error(format!("Unknown OPR {}", ir.a));
    }
}
OpCode::LOD => {
    let base = self.base(ir.l);
    let addr = (base as i64 + ir.a) as usize;
    self.stack[self.t] = self.stack[addr];
    self.t += 1;
}
OpCode::ST0 => {
    let base = self.base(ir.l);
    let addr = (base as i64 + ir.a) as usize;
    self.t -= 1;
    self.stack[addr] = self.stack[self.t];
}
OpCode::CAL => {
    let base = self.base(ir.l);
    self.stack[self.t] = base as i64; // Static Link (SL)
    self.stack[self.t + 1] = self.b as i64; // Dynamic Link (DL)
    self.stack[self.t + 2] = self.p as i64; // Return Address (RA)
    self.b = self.t;
    self.p = ir.a as usize;
}
OpCode::INT => {
    self.t = (self.t as i64 + ir.a) as usize;
}
OpCode::JMP => {
    self.p = ir.a as usize;
}
OpCode::JPC => {
    self.t -= 1;
    if self.stack[self.t] == 0 {
        self.p = ir.a as usize;
    }
}
OpCode::RED => {

```

```

        if let Some(val) = self.input_queue.pop() {
            let base = self.base(ir.l);
            let addr = (base as i64 + ir.a) as usize;
            self.stack[addr] = val;
        } else {
            // Push back PC to retry this instruction when input is
            available
            self.p -= 1;
            self.state = VMState::WaitingForInput;
        }
    }
    OpCode::WRT => {
        self.t -= 1; // Pop
        let val = self.stack[self.t];
        self.output.push(val.to_string());
    }
}
}

pub fn interpret(&mut self) {
    println!("Start PL/0");
    self.p = 0;
    self.b = 0;
    self.t = 0;
    self.state = VMState::Running;
    let mut output_index = 0;

    loop {
        match self.state {
            VMState::Running => {
                self.step();
                while output_index < self.output.len() {
                    println!("{}", self.output[output_index]);
                    output_index += 1;
                }
            }
            VMState::Halted => {
                println!("Program finished");
                break;
            }
            VMState::Error(ref e) => {
                println!("Runtime Error: {}", e);
                break;
            }
            VMState::WaitingForInput => {
                print!("Input: ");
                io::stdout().flush().unwrap();
                let mut input = String::new();
                io::stdin().read_line(&mut input).unwrap();
                if let Ok(val) = input.trim().parse::<i64>() {
                    self.input_queue.push(val);
                    self.state = VMState::Running;
                } else {
                    println!("Invalid input");
                }
            }
        }
    }
}

```

```

        }
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    use crate::types::{Instruction, OpCode};

    #[test]
    fn test_vm_arithmetic() {
        // 10 + 20
        let code = vec![
            Instruction::new(OpCode::LIT, 0, 10),
            Instruction::new(OpCode::LIT, 0, 20),
            Instruction::new(OpCode::OPR, 0, 2), // ADD
        ];
        let mut vm = VM::new(code);

        // Step through instructions
        vm.step(); // LIT 10
        vm.step(); // LIT 20
        vm.step(); // ADD

        assert_eq!(vm.stack[vm.t - 1], 30);
    }
}

```

4.2.10 图形界面 (src/gui.rs)

```

use crate::ast::{Block as AstBlock, Program, Statement};
use crate::codegen::CodeGenerator;
use crate::lexer::Lexer;
use crate::optimizer::optimize_ast;
use crate::parser::Parser;
use crate::semantic::SemanticAnalyzer;
use crate::symbol_table::SymbolTable;
use crate::types::Instruction;
use crate::vm::{VM, VMState};
use eframe::egui;
use std::time::{Duration, Instant};

#[derive(PartialEq)]
enum Tab {
    Editor,
    Tokens,
    AST,
    Symbols,
    Optimization,
    Runtime,
}

```

```

struct DiffLine {
    raw: Option<(usize, Instruction)>,
    opt: Option<(usize, Instruction)>,
}

fn compute_diff(raw: &[Instruction], opt: &[Instruction]) -> Vec<DiffLine> {
    let n = raw.len();
    let m = opt.len();
    // DP table for LCS length
    let mut dp = vec![vec![0; m + 1]; n + 1];

    for i in 1..=n {
        for j in 1..=m {
            // Instruction derives PartialEq
            if raw[i - 1] == opt[j - 1] {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = dp[i - 1][j].max(dp[i][j - 1]);
            }
        }
    }

    // Backtrack to find the diff
    let mut i = n;
    let mut j = m;
    let mut diffs = Vec::new();

    while i > 0 && j > 0 {
        if raw[i - 1] == opt[j - 1] {
            diffs.push(DiffLine {
                raw: Some((i - 1, raw[i - 1])),
                opt: Some((j - 1, opt[j - 1])),
            });
            i -= 1;
            j -= 1;
        } else if dp[i - 1][j] >= dp[i][j - 1] {
            diffs.push(DiffLine {
                raw: Some((i - 1, raw[i - 1])),
                opt: None,
            });
            i -= 1;
        } else {
            diffs.push(DiffLine {
                raw: None,
                opt: Some((j - 1, opt[j - 1])),
            });
            j -= 1;
        }
    }

    while i > 0 {
        diffs.push(DiffLine {
            raw: Some((i - 1, raw[i - 1])),
            opt: None,
        });
    }
}

```

```

        i -= 1;
    }

    while j > 0 {
        diffs.push(DiffLine {
            raw: None,
            opt: Some((j - 1, opt[j - 1])),
        });
        j -= 1;
    }

    diffs.reverse();
    diffs
}

pub struct Pl0Gui {
    // State
    source_code: String,
    tokens: Vec<usize, usize, crate::types::TokenType>,
    ast: Option<Program>,
    symbol_table: Option<SymbolTable>,
    raw_code: Vec<Instruction>,
    opt_code: Vec<Instruction>,
    vm: VM,

    // UI State
    current_tab: Tab,
    status_message: String,
    auto_run: bool,
    last_tick: Instant,
    input_buffer: String,
    use_optimized_vm: bool,

    // Visualization
    viz_root: Option<VizNode>,

    // Compilation error
    compile_error: Option<String>,
}

#[derive(Clone)]
struct VizNode {
    label: String,
    color: egui::Color32,
    children: Vec<VizNode>,
    pos: egui::Pos2,
    width: f32,
    total_width: f32, // Subtree width
}

impl VizNode {
    fn new(label: impl Into<String>, color: egui::Color32) -> Self {
        Self {
            label: label.into(),
            color,
    }
}

```

```

        children: Vec::new(),
        pos: egui::Pos2::ZERO,
        width: 0.0,
        total_width: 0.0,
    }
}

impl Pl0Gui {
    pub fn new(_cc: &eframe::CreationContext<'_>) -> Self {
        // Customize fonts if needed
        // cc.egui_ctx.set_fonts(...);

        let default_code = "program test1;
const a := 10, b := 20;
var x, y, z;
begin
    read(x);
    y := a * x + b;
    z := y / 2;
    write(x, y, z)
end";

        let mut app = Self {
            source_code: default_code.to_string(),
            tokens: Vec::new(),
            ast: None,
            symbol_table: None,
            raw_code: vec![],
            opt_code: vec![],
            vm: VM::new(vec![]),
            current_tab: Tab::Editor,
            status_message: "Ready".to_string(),
            auto_run: false,
            last_tick: Instant::now(),
            input_buffer: String::new(),
            use_optimized_vm: true,
            viz_root: None,
            compile_error: None,
        };
        app.compile();
        app
    }

    fn compile(&mut self) {
        self.status_message = "Compiling...".to_string();
        self.compile_error = None;
        self.tokens.clear();
        self.symbol_table = None;

        // 0. Lexical Analysis (Visualization)
        let mut lexer_viz = Lexer::new(&self.source_code);
        loop {
            let token = lexer_viz.current_token.clone();
            self.tokens
        }
    }
}

```

```

        .push((lexer_viz.token_line, lexer_viz.token_col,
token.clone())));
        if token == crate::types::TokenType::Eof {
            break;
        }
        lexer_viz.next_token();
    }

let lexer = Lexer::new(&self.source_code);
let mut parser = Parser::new(lexer, false);

match parser.parse() {
    Ok(mut program) => {
        // 1. Generate Raw Code
        let mut raw_program = program.clone();
        let mut sym_table = SymbolTable::new();
        let mut analyzer = SemanticAnalyzer::new(&mut sym_table);

        if let Err(e) = analyzer.analyze(&mut raw_program) {
            let err_msg = format!("Semantic Error: {:?}", e);
            self.status_message = err_msg.clone();
            self.compile_error = Some(err_msg);
            // Even if semantic error, we might want to show symbol table
so far?
            // But usually it fails early. Let's save what we have.
            self.symbol_table = Some(sym_table);
            return;
        }

        self.symbol_table = Some(sym_table.clone()); // Save for
visualization
        self.ast = Some(raw_program.clone());

        // Build Visualization Tree
        let mut root = build_viz_tree(&raw_program);
        layout_viz_tree(&mut root, 0, &mut 0.0);
        self.viz_root = Some(root);

        let mut generator = CodeGenerator::new();
        self.raw_code = generator.generate(&raw_program, &mut sym_table);

        // 2. Optimize AST & Generate Optimized Code
        optimize_ast(&mut program);
        let mut opt_sym_table = SymbolTable::new();
        let mut opt_analyzer = SemanticAnalyzer::new(&mut opt_sym_table);

        if let Err(e) = opt_analyzer.analyze(&mut program) {
            let err_msg = format!("Semantic Error (Opt): {:?}", e);
            self.status_message = err_msg.clone();
            self.compile_error = Some(err_msg);
            return;
        }

        let mut opt_generator = CodeGenerator::new();
        let code_from_ast = opt_generator.generate(&program, &mut

```

```

opt_sym_table);

        // 3. Peephole Optimization (Removed)
        self.opt_code = code_from_ast;
        let code_to_run = if self.use_optimized_vm {
            self.opt_code.clone()
        } else {
            self.raw_code.clone()
        };
        self.vm = VM::new(code_to_run);
        self.status_message = "Compilation Successful".to_string();
    }
    Err(_) => {
        let err = format!("Parse Error: {:?}", parser.errors.first());
        self.status_message = err.clone();
        self.compile_error = Some(err);
        self.raw_code.clear();
        self.opt_code.clear();
    }
}
}

impl eframe::App for Pl0Gui {
    fn update(&mut self, ctx: &egui::Context, _frame: &mut eframe::Frame) {
        // Top Panel: Tabs and Status
        egui::TopBottomPanel::top("top_panel").show(ctx, |ui| {
            ui.horizontal(|ui| {
                ui.selectable_value(&mut self.current_tab, Tab::Editor, "📝
Editor");
                ui.selectable_value(&mut self.current_tab, Tab::Tokens, "🔤
Tokens");
                ui.selectable_value(&mut self.current_tab, Tab::AST, "🌳 AST");
                ui.selectable_value(&mut self.current_tab, Tab::Symbols, "🧩
Symbols");
                ui.selectable_value(&mut self.current_tab, Tab::Optimization, "⚡
Optimization");
                ui.selectable_value(&mut self.current_tab, Tab::Runtime, "🚀
Runtime");
            });
            ui.with_layout(egui::Layout::right_to_left(egui::Align::Center),
|ui| {
                if let Some(err) = &self.compile_error {
                    ui.colored_label(egui::Color32::RED, err);
                } else {
                    ui.label(&self.status_message);
                }
            });
        });
    });

    // Central Panel: Content
    egui::CentralPanel::default().show(ctx, |ui| match self.current_tab {
        Tab::Editor => self.show_editor(ui),
        Tab::Tokens => self.show_tokens(ui),
    });
}
}

```

```

        Tab::AST => self.show_ast(ui),
        Tab::Symbols => self.show_symbols(ui),
        Tab::Optimization => self.show_optimization(ui),
        Tab::Runtime => self.show_runtime(ui, ctx),
    });
}
}

impl Pl0Gui {
    fn show_editor(&mut self, ui: &mut egui::Ui) {
        ui.heading("Source Code");
        let response = egui::ScrollArea::vertical().show(ui, |ui| {
            ui.add(
                egui::TextEdit::multiline(&mut self.source_code)
                    .font(egui::TextStyle::Monospace)
                    .code_editor()
                    .desired_width(f32::INFINITY)
                    .desired_rows(30)
                    .lock_focus(true),
            )
        });
    }

    if response.inner.changed() {
        self.compile();
    }
}

fn show_tokens(&self, ui: &mut egui::Ui) {
    ui.heading("Lexical Analysis (Tokens)");
    egui::ScrollArea::vertical().show(ui, |ui| {
        egui::Grid::new("tokens_grid")
            .striped(true)
            .spacing([20.0, 4.0])
            .show(ui, |ui| {
                ui.label(egui::RichText::new("Line:Col").strong());
                ui.label(egui::RichText::new("Token Type").strong());
                ui.label(egui::RichText::new("Value").strong());
                ui.end_row();

                for (line, col, token) in &self.tokens {
                    ui.monospace(format!("{}:{}", line, col));
                    match token {
                        crate::types::TokenType::Identifier(s) => {
                            ui.monospace("Identifier");
                            ui.monospace(s);
                        }
                        crate::types::TokenType::Number(n) => {
                            ui.monospace("Number");
                            ui.monospace(n.to_string());
                        }
                        _ => {
                            ui.monospace(format!("{:?}", token));
                            ui.label("");
                        }
                    }
                }
            })
    });
}
}

```

```

                ui.end_row();
            }
        });
    });
}

fn show_symbols(&self, ui: &mut egui::Ui) {
    ui.heading("Symbol Table");
    if let Some(sym_table) = &self.symbol_table {
        egui::ScrollArea::vertical().show(ui, |ui| {
            for (i, scope) in sym_table.scopes.iter().enumerate() {
                egui::CollapsingHeader::new(format!("Scope {}", i))
                    .default_open(true)
                    .show(ui, |ui| {
                        if let Some(parent) = scope.parent {
                            ui.label(format!("Parent Scope: {}", parent));
                        } else {
                            ui.label("Root Scope");
                        }

                        if !scope.symbols.is_empty() {
                            egui::Grid::new(format!("scope_{}_grid", i))
                                .striped(true)
                                .spacing([20.0, 4.0])
                                .show(ui, |ui| {

ui.label(egui::RichText::new("Name").strong());
ui.label(egui::RichText::new("Kind").strong());
ui.label(egui::RichText::new("Details").strong());
ui.end_row();

for (name, sym) in &scope.symbols {
    ui.monospace(name);
    match &sym.kind {
        crate::types::SymbolType::Constant { val } => {
            ui.monospace("Constant");
            ui.monospace(format!("Value: {}", val));
        }
        crate::types::SymbolType::Variable {
            level,
            addr,
        } => {
            ui.monospace("Variable");
            ui.monospace(format!(
                "L: {}, A: {}",
                level, addr
            ));
        }
        crate::types::SymbolType::Procedure {
            level,
            addr,
        } => {
    }
}

```

```

        ui.monospace("Procedure");
        ui.monospace(format!(
            "L: {}, A: {}",
            level, addr
        )));
    }
    ui.end_row();
}
});
} else {
    ui.label("No symbols in this scope.");
}
});
}
);
} else {
    ui.label("No symbol table available.");
}
}

fn show_ast(&self, ui: &mut egui::Ui) {
    ui.heading("Abstract Syntax Tree (Visualized)");
    if let Some(root) = &self.viz_root {
        egui::ScrollArea::both().show(ui, |ui| {
            let (max_x, max_y) = get_tree_bounds(root);
            let canvas_size = egui::vec2(max_x + 120.0, max_y + 120.0);
            let (response, painter) = ui.allocate_painter(canvas_size,
egui::Sense::hover());
            // Offset ensures a small margin inside the canvas
            let offset = response.rect.min.to_vec2() + egui::vec2(20.0,
20.0);

            draw_viz_tree(&painter, root, offset);
        });
    } else {
        ui.label("No AST available. Fix compilation errors.");
    }
}

fn format_operator(&self, op: &crate::types::Operator) -> String {
    match op {
        crate::types::Operator::ADD => "+".to_string(),
        crate::types::Operator::SUB => "-".to_string(),
        crate::types::Operator::MUL => "*".to_string(),
        crate::types::Operator::DIV => "/".to_string(),
        crate::types::Operator::EQ => "=" .to_string(),
        crate::types::Operator::NEQ => "#".to_string(),
        crate::types::Operator::LSS => "<".to_string(),
        crate::types::Operator::LEQ => "<=".to_string(),
        crate::types::Operator::GTR => ">".to_string(),
        crate::types::Operator::GEQ => ">=".to_string(),
        crate::types::Operator::ODD => "odd".to_string(),
        crate::types::Operator::NEG => "-".to_string(),
    }
}

```

```

        _ => format!("{:?}", op),
    }
}

fn format_expr(&self, expr: &crate::ast::Expr) -> String {
    match expr {
        crate::ast::Expr::Binary { left, op, right } => {
            format!(
                "({} {} {})",
                self.format_expr(left),
                self.format_operator(op),
                self.format_expr(right)
            )
        }
        crate::ast::Expr::Unary { op, expr } => {
            format!("({}{}{})", self.format_operator(op),
                    self.format_expr(expr))
        }
        crate::ast::Expr::Number(n) => n.to_string(),
        crate::ast::Expr::Identifier(id) => id.clone(),
    }
}

fn format_condition(&self, cond: &crate::ast::Condition) -> String {
    match cond {
        crate::ast::Condition::Odd { expr } => {
            format!("odd {}", self.format_expr(expr))
        }
        crate::ast::Condition::Compare { left, op, right } => {
            format!(
                "{} {} {}",
                self.format_expr(left),
                self.format_operator(op),
                self.format_expr(right)
            )
        }
    }
}

fn draw_block(&self, ui: &mut egui::Ui, block: &AstBlock, label: &str) {
    egui::CollapsingHeader::new(label)
        .default_open(true)
        .show(ui, |ui| {
            if !block.consts.is_empty() {
                ui.label(egui::RichText::new("Constants:").strong());
                for c in &block.consts {
                    ui.label(format!("{} = {}", c.name, c.value));
                }
            }

            if !block.vars.is_empty() {
                ui.label(egui::RichText::new("Variables:").strong());
                ui.label(block.vars.join(", "));
            }
        })
}

```

```

        if !block.procedures.is_empty() {
            ui.label(egui::RichText::new("Procedures:").strong());
            for p in &block.procedures {
                let label = if p.params.is_empty() {
                    format!("Procedure {}", p.name)
                } else {
                    format!("Procedure {}({})", p.name, p.params.join(","))
                };
                self.draw_block(ui, &p.block, &label);
            }
        }

        ui.separator();
        self.draw_statement(ui, &block.statement);
    });

fn draw_statement(&self, ui: &mut egui::Ui, stmt: &Statement) {
    match stmt {
        Statement::Assignment { name, expr } => {
            ui.label(format!("Assign: {} := {}", name,
self.format_expr(expr)));
        }
        Statement::Call { name, args } => {
            if args.is_empty() {
                ui.label(format!("Call: {}", name));
            } else {
                let args_str = args
                    .iter()
                    .map(|e| self.format_expr(e))
                    .collect::<Vec<_>>()
                    .join(", ");
                ui.label(format!("Call: {}({})", name, args_str));
            }
        }
        Statement::BeginEnd { statements } => {
            egui::CollapsingHeader::new("Begin ... End")
                .default_open(true)
                .show(ui, |ui| {
                    for s in statements {
                        self.draw_statement(ui, s);
                    }
                });
        }
        Statement::If {
            condition,
            then_stmt,
            else_stmt,
        } => {
            egui::CollapsingHeader::new(format!(
                "If {} Then",
                self.format_condition(condition)
            ))
            .default_open(true)

```

```

        .show(ui, |ui| {
            self.draw_statement(ui, then_stmt);
            if let Some(else_s) = else_stmt {
                ui.label("Else");
                self.draw_statement(ui, else_s);
            }
        });
    }
Statement::While { condition, body } => {
    egui::CollapsingHeader::new(format!(
        "While {} Do",
        self.format_condition(condition)
    ))
    .default_open(true)
    .show(ui, |ui| {
        self.draw_statement(ui, body);
    });
}
Statement::Read { names } => {
    ui.label(format!("Read: {}", names.join(", ")));
}
Statement::Write { exprs } => {
    let exprs_str = exprs
        .iter()
        .map(|e| self.format_expr(e))
        .collect::<Vec<_>>()
        .join(", ");
    ui.label(format!("Write: {}", exprs_str));
}
Statement::Empty => {
    ui.label("Empty");
}
}
}

fn show_optimization(&self, ui: &mut egui::Ui) {
    let diff = compute_diff(&self.raw_code, &self.opt_code);

    ui.heading(format!(
        "Optimization Diff ({} -> {}) instructions",
        self.raw_code.len(),
        self.opt_code.len()
    ));

    egui::ScrollArea::vertical().show(ui, |ui| {
        egui::Grid::new("diff_grid")
            .striped(true)
            .min_col_width(200.0)
            .spacing([20.0, 4.0]) // Add some spacing between columns and
rows
            .show(ui, |ui| {
                ui.label(egui::RichText::new("Raw Bytecode").strong());
                ui.label(egui::RichText::new("Optimized Bytecode").strong());
                ui.end_row();
            });
    });
}

```

```

        for line in diffs {
            // Left Column (Raw)
            if let Some((i, instr)) = line.raw {
                let text = format!("{:3}: {:?} {}", i, instr.f,
instr.l, instr.a);
                if line.opt.is_none() {
                    // Deleted - Red
                    ui.label(
                        egui::RichText::new(text)
                        .color(egui::Color32::from_rgb(255, 100,
100)),
                );
            } else {
                // Unchanged
                ui.monospace(text);
            }
        } else {
            ui.label("");
        }

        // Right Column (Optimized)
        if let Some((i, instr)) = line.opt {
            let text = format!("{:3}: {:?} {}", i, instr.f,
instr.l, instr.a);
            if line.raw.is_none() {
                // Added - Green
                ui.label(
                    egui::RichText::new(text)
                    .color(egui::Color32::from_rgb(100, 255,
100)),
            );
            } else {
                // Unchanged
                ui.monospace(text);
            }
        } else {
            ui.label("");
        }
        ui.end_row();
    }
});
};

fn show_runtime(&mut self, ui: &mut egui::Ui, ctx: &egui::Context) {
    // Controls Row 1: Actions & Settings
    ui.horizontal(|ui| {
        if ui.button("Step").clicked() {
            self.vm.step();
        }
        if ui
            .button(if self.auto_run { "Pause" } else { "Run" })
            .clicked()
        {
            self.auto_run = !self.auto_run;
        }
    });
}

```

```

    }

    if ui.button("Reset").clicked() {
        let code_to_run = if self.use_optimized_vm {
            self.opt_code.clone()
        } else {
            self.raw_code.clone()
        };
        self.vm = VM::new(code_to_run);
        self.auto_run = false;
    }

    ui.separator();
    if ui
        .checkbox(&mut self.use_optimized_vm, "Use Optimized Code")
        .changed()
    {
        let code_to_run = if self.use_optimized_vm {
            self.opt_code.clone()
        } else {
            self.raw_code.clone()
        };
        self.vm = VM::new(code_to_run);
        self.auto_run = false;
    }
});

// Controls Row 2: Status & Registers
ui.horizontal(|ui| {
    ui.label(egui::RichText::new("Registers:").strong());
    ui.label(format!("PC: {:<3}", self.vm.p));
    ui.label(format!("BP: {:<3}", self.vm.b));
    ui.label(format!("SP: {:<3}", self.vm.t));
    ui.label(format!("IR: {:?}", self.vm.i));

    ui.separator();
    ui.label(format!("State: {:?}", self.vm.state));

    ui.separator();
    ui.label(format!("Total Instructions: {}", self.vm.instruction_count));
});

ui.separator();

// Main Runtime View
ui.columns(3, |columns| {
    // 1. Code View
    columns[0].vertical(|ui| {
        ui.heading("Code");
        egui::ScrollArea::vertical()
            .id_salt("vm_code")
            .show(ui, |ui| {
                for (i, instr) in self.vm.code.iter().enumerate() {
                    let text = format!("{:3}: {:?}", i, instr);
                    if i == self.vm.p {

```

```

        ui.label(
            egui::RichText::new(text)
                .monospace()
                .strong()
                .background_color(egui::Color32::YELLOW)
                .color(egui::Color32::BLACK),
        );
        ui.scroll_to_cursor(Some(egui::Align::Center));
    } else {
        ui.monospace(text);
    }
}
});

// 2. Stack View
columns[1].vertical(|ui| {
    ui.heading("Stack");
    egui::ScrollArea::vertical()
        .id_salt("vm_stack")
        .show(ui, |ui| {
            egui::Grid::new("stack_grid")
                .striped(true)
                .spacing([10.0, 4.0])
                .min_col_width(40.0)
                .show(ui, |ui| {
                    // Header

ui.label(egui::RichText::new("Addr").strong().underline());
ui.label(egui::RichText::new("Value").strong().underline());
ui.label(egui::RichText::new("Tags").strong().underline());
ui.end_row();

                    // Content
let limit = ((self.vm.t + 1).max(self.vm.b + 3))
    .min(self.vm.stack.len());
for (i, val) in
self.vm.stack.iter().enumerate().take(limit) {
    let is_bp = i == self.vm.b;
    let is_sp = i == self.vm.t;
    let is_sl = i == self.vm.b;
    let is_dl = i == self.vm.b + 1;
    let is_ra = i == self.vm.b + 2;

    let mut addr_text =
        egui::RichText::new(format!("{:03}", i)).monospace();
    let mut val_text =
        egui::RichText::new(val.to_string()).monospace();

        if is_bp {
            addr_text =
addr_text.color(egui::Color32::LIGHT_BLUE);
            val_text = val_text.strong();

```

```

        }
        if is_sp {
            addr_text = val_text = val_text.strong();
        }

        ui.label(addr_text);
        ui.label(val_text);

        // Tags column
        ui.horizontal(|ui| {
            if is_bp {
                ui.label(
                    egui::RichText::new(" BP ")
                    .small()
                    .background_color(egui::Color32::from_rgb(
                        0, 100, 200,
                    ))
                    .color(egui::Color32::WHITE),
                );
            }
            if is_sp {
                ui.label(
                    egui::RichText::new(" SP ")
                    .small()
                    .background_color(egui::Color32::from_rgb(
                        200, 50, 50,
                    ))
                    .color(egui::Color32::WHITE),
                );
            }
            if is_sl {
                ui.label(
                    egui::RichText::new(" SL ")
                    .small()
                    .background_color(egui::Color32::GRAY)
                    .color(egui::Color32::WHITE),
                );
            }
            if is_dl {
                ui.label(
                    egui::RichText::new(" DL ")
                    .small()
                    .background_color(egui::Color32::GRAY)
                    .color(egui::Color32::WHITE),
                );
            }
            if is_ra {
                ui.label(
                    egui::RichText::new(" RA ")
                    .small()
                    .background_color(egui::Color32::GRAY)
                    .color(egui::Color32::WHITE),
                );
            }
        });
    }
}

```

```

                }
            });
            ui.end_row();
        );
    );
}

// 3. I/O View
columns[2].vertical(|ui| {
    ui.heading("I/O Console");

    egui::Frame::canvas(ui.style())
        .fill(egui::Color32::from_rgb(40, 44, 52)) // Dark console
background
        .inner_margin(10.0)
        .show(ui, |ui| {
            // Output Area
            let footer_height = 30.0;
            let available_height = ui.available_height() -
footer_height;

            egui::ScrollArea::vertical()
                .id_salt("vm_output")
                .max_height(available_height)
                .stick_to_bottom(true) // Auto-scroll to bottom
                .show(ui, |ui| {
                    ui.set_min_width(ui.available_width());
                    ui.set_min_height(available_height);

                    for line in &self.vm.output {
                        ui.label(
                            egui::RichText::new(line)
                                .monospace()
                                .color(egui::Color32::LIGHT_GRAY),
                        );
                    }
                });

                if self.vm.state == VMState::WaitingForInput {
                    ui.label(
                        egui::RichText::new("? Waiting for
input...")
                            .monospace()
                            .italics()
                            .color(egui::Color32::YELLOW),
                    );
                }
            );
        });

        ui.separator();

        // Input Area
        ui.horizontal(|ui| {
            ui.label(
                egui::RichText::new(">")

```

```

        .monospace()
        .strong()
        .color(egui::Color32::WHITE),
    );

    let response = ui.add(
        egui::TextEdit::singleline(&mut
self.input_buffer)
        .frame(false)
        .desired_width(f32::INFINITY)
        .text_color(egui::Color32::WHITE)
        .font(egui::TextStyle::Monospace)
        .hint_text("Enter number..."),
    );

    // Handle Enter key
    if (response.lost_focus()
        && ui.input(|i| i.key_pressed(egui::Key::Enter)))
        || (response.has_focus()
            && ui.input(|i|
i.key_pressed(egui::Key::Enter)))
    {
        if !self.input_buffer.is_empty() {
            if let Ok(val) =
self.input_buffer.trim().parse::<i64>() {
                self.vm.input_queue.push(val);
                // Echo input to output
                self.vm.output.push(format!("> {}", val));
            }
        }
    }

    if self.vm.state == VMState::WaitingForInput {
        self.vm.state = VMState::Running;
    }
    self.input_buffer.clear();

    // Keep focus
    response.request_focus();
}
}

// Auto-focus if waiting
if self.vm.state == VMState::WaitingForInput && !
response.has_focus() {
    response.request_focus();
}
});

});

// Auto-run logic
if self.auto_run && self.vm.state == VMState::Running {
    if self.last_tick.elapsed() >= Duration::from_millis(50) {

```

```

        self.vm.step();
        self.last_tick = Instant::now();
        ctx.request_repaint(); // Request next frame immediately
    } else {
        ctx.request_repaint_after(Duration::from_millis(50));
    }
}
}

fn build_viz_tree(program: &Program) -> VizNode {
    let mut root = VizNode::new("Program", egui::Color32::from_rgb(200, 200, 255));
    root.children.push(build_block_node(&program.block));
    root
}

fn build_block_node(block: &AstBlock) -> VizNode {
    let mut node = VizNode::new("Block", egui::Color32::from_rgb(255, 200, 200));

    // Consts
    if !block.consts.is_empty() {
        let mut consts_node = VizNode::new("Consts", egui::Color32::LIGHT_GRAY);
        for c in &block.consts {
            consts_node.children.push(VizNode::new(
                format!("{}={}", c.name, c.value),
                egui::Color32::WHITE,
            ));
        }
        node.children.push(consts_node);
    }

    // Vars
    if !block.vars.is_empty() {
        let mut vars_node = VizNode::new("Vars", egui::Color32::LIGHT_GRAY);
        for v in &block.vars {
            vars_node
                .children
                .push(VizNode::new(v.clone(), egui::Color32::WHITE));
        }
        node.children.push(vars_node);
    }

    // Procedures
    for p in &block.procedures {
        let mut proc_node = VizNode::new(format!("Proc {}", p.name), egui::Color32::GOLD);
        proc_node.children.push(build_block_node(&p.block));
        node.children.push(proc_node);
    }

    // Statement
    node.children.push(build_statement_node(&block.statement));
}

node

```

```

}

fn build_statement_node(stmt: &Statement) -> VizNode {
    match stmt {
        Statement::Assignment { name, expr } => {
            let mut node = VizNode::new(":=", egui::Color32::LIGHT_GREEN);
            node.children
                .push(VizNode::new(name.clone(), egui::Color32::WHITE));
            node.children.push(build_expr_node(expr));
            node
        }
        Statement::Call { name, args } => {
            let mut node = VizNode::new(format!("Call {}", name),
egui::Color32::LIGHT_RED);
            for arg in args {
                node.children.push(build_expr_node(arg));
            }
            node
        }
        Statement::BeginEnd { statements } => {
            let mut node = VizNode::new("Begin..End",
egui::Color32::from_rgb(200, 200, 255));
            for s in statements {
                node.children.push(build_statement_node(s));
            }
            node
        }
        Statement::If {
            condition,
            then_stmt,
            else_stmt,
        } => {
            let mut node = VizNode::new("If", egui::Color32::LIGHT_YELLOW);
            node.children.push(build_condition_node(condition));
            node.children.push(build_statement_node(then_stmt));
            if let Some(else_s) = else_stmt {
                node.children.push(build_statement_node(else_s));
            }
            node
        }
        Statement::While { condition, body } => {
            let mut node = VizNode::new("While", egui::Color32::LIGHT_YELLOW);
            node.children.push(build_condition_node(condition));
            node.children.push(build_statement_node(body));
            node
        }
        Statement::Read { names } => {
            let mut node = VizNode::new("Read", egui::Color32::LIGHT_BLUE);
            for name in names {
                node.children
                    .push(VizNode::new(name.clone(), egui::Color32::WHITE));
            }
            node
        }
        Statement::Write { exprs } => {

```

```

        let mut node = VizNode::new("Write", egui::Color32::LIGHT_BLUE);
        for expr in exprs {
            node.children.push(build_expr_node(expr));
        }
    node
}
Statement::Empty => VizNode::new("Empty", egui::Color32::GRAY),
}

fn build_expr_node(expr: &crate::ast::Expr) -> VizNode {
    match expr {
        crate::ast::Expr::Binary { left, op, right } => {
            let mut node = VizNode::new(format_op(op),
egui::Color32::LIGHT_GREEN);
            node.children.push(build_expr_node(left));
            node.children.push(build_expr_node(right));
            node
        }
        crate::ast::Expr::Unary { op, expr } => {
            let mut node = VizNode::new(
                format!("Unary {}", format_op(op)),
                egui::Color32::LIGHT_GREEN,
            );
            node.children.push(build_expr_node(expr));
            node
        }
        crate::ast::Expr::Number(n) => VizNode::new(n.to_string(),
egui::Color32::WHITE),
        crate::ast::Expr::Identifier(id) => VizNode::new(id.clone(),
egui::Color32::WHITE),
    }
}

fn build_condition_node(cond: &crate::ast::Condition) -> VizNode {
    match cond {
        crate::ast::Condition::Odd { expr } => {
            let mut node = VizNode::new("Odd", egui::Color32::LIGHT_YELLOW);
            node.children.push(build_expr_node(expr));
            node
        }
        crate::ast::Condition::Compare { left, op, right } => {
            let mut node = VizNode::new(format_op(op),
egui::Color32::LIGHT_YELLOW);
            node.children.push(build_expr_node(left));
            node.children.push(build_expr_node(right));
            node
        }
    }
}

fn format_op(op: &crate::types::Operator) -> String {
    match op {
        crate::types::Operator::ADD => "+".to_string(),
        crate::types::Operator::SUB => "-".to_string(),
    }
}

```

```

        crate::types::Operator::MUL => "*".to_string(),
        crate::types::Operator::DIV => "/".to_string(),
        crate::types::Operator::EQL => "=" .to_string(),
        crate::types::Operator::NEQ => "#".to_string(),
        crate::types::Operator::LSS => "<".to_string(),
        crate::types::Operator::LEQ => "<=".to_string(),
        crate::types::Operator::GTR => ">".to_string(),
        crate::types::Operator::GEQ => ">=".to_string(),
        crate::types::Operator::ODD => "odd".to_string(),
        crate::types::Operator::NEG => "-".to_string(),
        _ => format!("{:?}", op),
    }
}

fn layout_viz_tree(node: &mut VizNode, depth: usize, next_x: &mut f32) {
    let node_width = 100.0;
    let node_height = 60.0;
    let spacing_x = 20.0;

    if node.children.is_empty() {
        node.pos = egui::pos2(*next_x, depth as f32 * node_height);
        node.width = node_width;
        *next_x += node_width + spacing_x;
    } else {
        let start_x = *next_x;
        for child in &mut node.children {
            layout_viz_tree(child, depth + 1, next_x);
        }

        // Center parent over children
        // If children are spread out, we want the parent to be in the middle of
        the range covered by children
        let first_child_x = node.children.first().unwrap().pos.x;
        let last_child_x = node.children.last().unwrap().pos.x;
        let center_x = (first_child_x + last_child_x) / 2.0;

        node.pos = egui::pos2(center_x, depth as f32 * node_height);
        node.width = node_width;

        // If the children didn't advance next_x enough (e.g. they were placed to
        the left of current next_x because of recursion logic?),
        // actually next_x is always increasing in this simple algorithm.
        // But we need to make sure we don't overlap with previous subtrees if we
        just center.
        // The simple "next_x" approach guarantees no overlap between leaves.
        // Parent centering is safe.
    }
}

fn get_tree_bounds(node: &VizNode) -> (f32, f32) {
    let mut max_x = node.pos.x + node.width;
    let mut max_y = node.pos.y + 50.0;

    for child in &node.children {
        let (cx, cy) = get_tree_bounds(child);

```

```

        max_x = max_x.max(cx);
        max_y = max_y.max(cy);
    }
    (max_x, max_y)
}

fn draw_viz_tree(painter: &egui::Painter, node: &VizNode, offset: egui::Vec2) {
    let node_size = egui::vec2(90.0, 30.0);
    let node_pos = node.pos + offset;
    let rect = egui::Rect::from_min_size(node_pos, node_size);

    // Draw edges to children
    for child in &node.children {
        let child_pos = child.pos + offset;
        let child_rect = egui::Rect::from_min_size(child_pos, node_size);

        let start = rect.center_bottom();
        let end = child_rect.center_top();

        // Bezier curve for nicer edges
        let control_point1 = start + egui::vec2(0.0, 20.0);
        let control_point2 = end - egui::vec2(0.0, 20.0);

        let cubic_bezier = egui::epaint::CubicBezierShape::from_points_stroke(
            [start, control_point1, control_point2, end],
            false,
            egui::Color32::TRANSPARENT, // fill
            egui::Stroke::new(1.5, egui::Color32::GRAY),
        );
        painter.add(cubic_bezier);

        draw_viz_tree(painter, child, offset);
    }

    // Draw node box
    painter.rect_filled(rect, 5.0, node.color);
    painter.rect_stroke(
        rect,
        5.0,
        egui::Stroke::new(1.0, egui::Color32::BLACK),
        egui::StrokeKind::Middle,
    );

    // Draw label
    painter.text(
        rect.center(),
        egui::Align2::CENTER_CENTER,
        &node.label,
        egui::FontId::proportional(14.0),
        egui::Color32::BLACK,
    );
}
}

```

4.2.11 主程序入口

src/lib.rs

```

pub mod ast;
pub mod codegen;
pub mod gui;
pub mod lexer;
pub mod optimizer;
pub mod parser;
pub mod semantic;
pub mod symbol_table;
pub mod types;
pub mod vm;

```

src/bin/pl0c.rs

```

use pl0::codegen::CodeGenerator;
use pl0::lexer::Lexer;
use pl0::optimizer::optimize_ast;
use pl0::parser::Parser;
use pl0::semantic::SemanticAnalyzer;
use pl0::symbol_table::SymbolTable;
use pl0::vm::VM;
use std::env;
use std::fs::{self, File};
use std::io::Write;

fn main() {
    let args: Vec<String> = env::args().collect();
    let mut verbose = false;
    let mut use_optimization = false;
    let mut positional_args = Vec::new();

    for arg in args.iter().skip(1) {
        if arg == "--verbose" || arg == "-v" {
            verbose = true;
        } else if arg == "-o2" {
            use_optimization = true;
        } else {
            positional_args.push(arg);
        }
    }

    if positional_args.is_empty() {
        eprintln!("Usage: {} <source_file> [output_file] [--verbose]", args[0]);
        std::process::exit(1);
    }

    let source_path = positional_args[0];
    let output_path = if positional_args.len() >= 2 {
        positional_args[1]
    } else {
        "out.asm"
    };

    let source_code = fs::read_to_string(source_path).expect("Failed to read source file");
}

```

```

let lexer = Lexer::new(&source_code);
let mut parser = Parser::new(lexer, verbose);

println!("Compiling {}...", source_path);

// Parse to AST
let parse_result = parser.parse();

if !parser.errors.is_empty() {
    eprintln!("Parsing encountered errors.");
    let lines: Vec<&str> = source_code.lines().collect();
    for err in &parser.errors {
        eprintln!(
            "{}:{}:{}: error: {}",
            source_path, err.line, err.col, err.message
        );
        if err.line > 0 && err.line <= lines.len() {
            let line_content = lines[err.line - 1];
            eprintln!("    {}", line_content);
            let indent: String = line_content
                .chars()
                .take(err.col - 1)
                .map(|c| if c.is_whitespace() { c } else { ' ' })
                .collect();
            eprintln!("    {}^", indent);
        }
    }
    eprintln!("Compilation failed due to parsing errors.");
    std::process::exit(1);
}

if parse_result.is_err() {
    eprintln!("Fatal parsing error.");
    std::process::exit(1);
}

let mut program = parse_result.unwrap();

if use_optimization {
    println!("Optimizing AST...");
    optimize_ast(&mut program);
}

println!("Performing Semantic Analysis...");
let mut symbol_table = SymbolTable::new();
let mut analyzer = SemanticAnalyzer::new(&mut symbol_table);

if let Err(errors) = analyzer.analyze(&mut program) {
    eprintln!("Semantic analysis failed:");
    for err in errors {
        eprintln!("    {}", err);
    }
    std::process::exit(1);
}

```

```

    println!("Generating Code...");
    let mut generator = CodeGenerator::new();
    let code = generator.generate(&program, &mut symbol_table);

    println!(
        "Compilation successful! Generated {} instructions.",
        code.len()
    );

    let final_code = code;

    let mut file = File::create(output_path).expect("Failed to create output
file");
    for instr in &final_code {
        writeln!(file, "{} {} {}", instr.f, instr.l, instr.a)
            .expect("Failed to write instruction");
    }

    println!("Wrote assembly to {}", output_path);

    println!("Running {}...", source_path);
    let mut vm = VM::new(final_code);
    vm.interpret();
}

```

src/bin/pl0gui.rs

```

use eframe::egui;
use pl0::gui::Pl0Gui;

fn main() -> eframe::Result<()> {
    let native_options = eframe::NativeOptions {
        viewport: egui::ViewportBuilder::default()
            .with_inner_size([1024.0, 768.0])
            .with_title("PL/0 Studio"),
        ..Default::default()
    };

    eframe::run_native(
        "PL/0 Studio",
        native_options,
        Box::new(|cc| Ok(Box::new(Pl0Gui::new(cc)))),
    )
}

```

src/bin/pl0vm.rs

```

use pl0::types::{Instruction, OpCode};
use pl0::vm::VM;
use std::env;
use std::fs::File;
use std::io::{BufRead, BufferedReader};

```

```

fn parse_opcode(s: &str) -> OpCode {
    match s {
        "LIT" => OpCode::LIT,
        "OPR" => OpCode::OPR,
        "LOD" => OpCode::LOD,
        "STO" => OpCode::STO,
        "CAL" => OpCode::CAL,
        "INT" => OpCode::INT,
        "JMP" => OpCode::JMP,
        "JPC" => OpCode::JPC,
        "RED" => OpCode::RED,
        "WRT" => OpCode::WRT,
        _ => panic!("Unknown opcode: {}", s),
    }
}

fn main() {
    let args: Vec<String> = env::args().collect();

    if args.len() < 2 {
        eprintln!("Usage: {} <asm_file>", args[0]);
        std::process::exit(1);
    }

    let path = &args[1];

    let file = File::open(path).expect("Failed to open asm file");
    let reader = BufReader::new(file);
    let mut instructions = Vec::new();

    for line in reader.lines() {
        let line = line.expect("Failed to read line");
        let parts: Vec<&str> = line.split_whitespace().collect();
        if parts.len() >= 3 {
            let f = parse_opcode(parts[0]);
            let l = parts[1].parse::<usize>().expect("Failed to parse level");
            let a = parts[2].parse::<i64>().expect("Failed to parse address");
            instructions.push(Instruction::new(f, l, a));
        }
    }

    println!("Loaded {} instructions.", instructions.len());
    println!("Executing...");

    let mut vm = VM::new(instructions);
    vm.interpret();
}

```

5 系统测试

为了验证编译器的正确性，我们设计了多组测试用例，覆盖了 PL/0 语言的各个特性，包括基础功能、控制流、过程调用、递归、作用域嵌套以及错误处理等。

5.1 基础功能测试

5.1.1 测试程序 (`base1.txt`)

```
program test1;
const a := 10, b := 20;
var x, y, z;
begin
  read(x);
  y := a * x + b;
  z := y / 2;
  write(x, y, z)
end
```

5.1.2 结果分析

该测试用例验证了常量定义、变量声明、读写语句以及基本的算术运算（乘法、加法、除法）。

- 输入: 4
- 计算过程: $y := 10 * 4 + 20 = 60, z := 60 / 2 = 30$
- 预期输出: 4, 60, 30
- 实际运行结果与预期一致，证明基础算术指令 (OPR) 和 I/O 指令 (RED, WRT) 工作正常。

Code	Stack	I/O Console
0: Instruction { f: JMP, l: 0, a: 1 }	Addr Value Tags	> 4
1: Instruction { f: INT, l: 0, a: 6 }	000 0 []	4
2: Instruction { f: OPR, l: 0, a: 16 }	001 0 []	60
3: Instruction { f: STO, l: 0, a: 3 }	002 0 []	30
4: Instruction { f: LIT, l: 0, a: 10 }		
5: Instruction { f: LOD, l: 0, a: 3 }		
6: Instruction { f: OPR, l: 0, a: 4 }		
7: Instruction { f: LIT, l: 0, a: 20 }		
8: Instruction { f: OPR, l: 0, a: 2 }		
9: Instruction { f: STO, l: 0, a: 4 }		
10: Instruction { f: LOD, l: 0, a: 4 }		
11: Instruction { f: LIT, l: 0, a: 2 }		
12: Instruction { f: OPR, l: 0, a: 5 }		
13: Instruction { f: STO, l: 0, a: 5 }		
14: Instruction { f: LOD, l: 0, a: 3 }		
15: Instruction { f: OPR, l: 0, a: 14 }		
16: Instruction { f: LOD, l: 0, a: 4 }		
17: Instruction { f: OPR, l: 0, a: 14 }		
18: Instruction { f: LOD, l: 0, a: 5 }		
19: Instruction { f: OPR, l: 0, a: 14 }		
20: Instruction { f: OPR, l: 0, a: 0 }		

Figure 12: 基础功能测试运行结果

5.2 控制流测试

5.2.1 测试程序 (**if-else.txt**)

```
program test2;
var n, sum, i;
begin
  read(n);
  if n <= 0 then
    write(0)
  else
    begin
      sum := 0;
      i := 1;
      while i <= n do
        begin
          if odd i then
            sum := sum + i;
          i := i + 1
        end;
      write(sum)
    end
  end
end
```

5.2.2 结果分析

该测试用例综合测试了 **if-else** 条件分支、**while** 循环结构以及 **odd** 运算符。程序功能是计算 1 到 **n** 之间所有奇数的和。

- 输入: 5
- 计算过程: 奇数为 1, 3, 5。Sum = 1 + 3 + 5 = 9。
- 预期输出: 9
- 实际运行结果正确, 证明跳转指令 (**JMP**, **JPC**) 和条件判断逻辑正确。

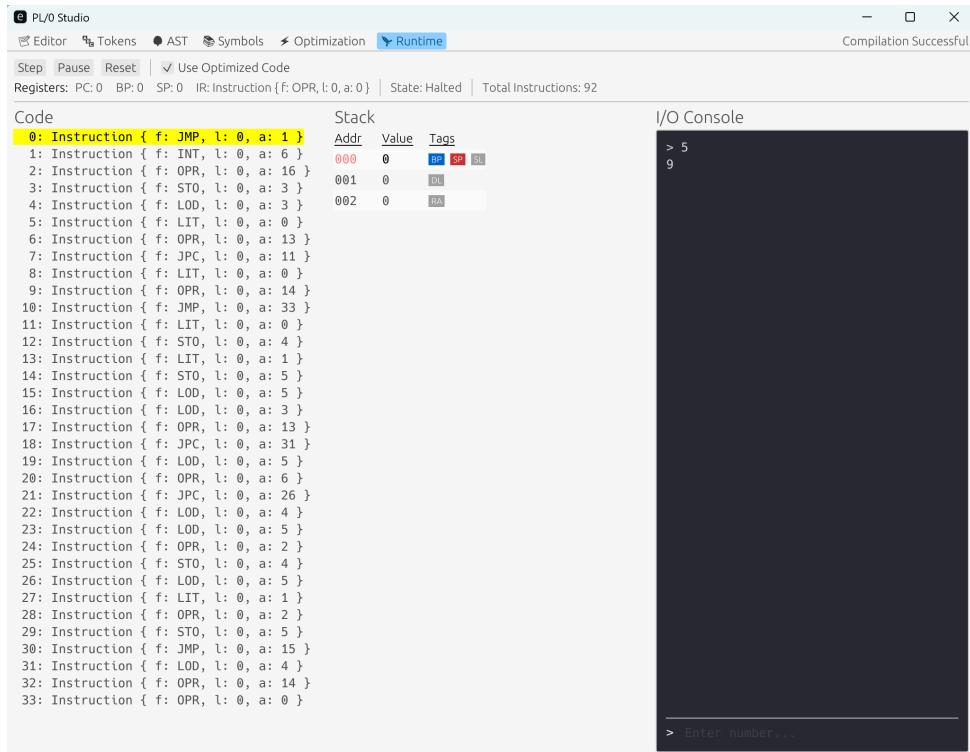


Figure 13: 控制流测试运行结果

5.3 过程调用测试

5.3.1 测试程序 (call.txt)

```

program test3;
var x, y, res;

procedure multiply(a, b);
begin
    res := a * b;
    write(a, b, res)
end;

begin
    read(x, y);
    if x > y then
        call multiply(x, y)
    else
        call multiply(y, x)
end

```

5.3.2 结果分析

该测试用例验证了带参数的过程调用。

- 输入: 3 4
- 逻辑: $3 \leq 4$, 执行 else 分支 `call multiply(4, 3)`。
- 过程内部: `res := 4 * 3 = 12`。
- 预期输出: 4, 3, 12
- 验证了 CAL 指令及参数传递机制。

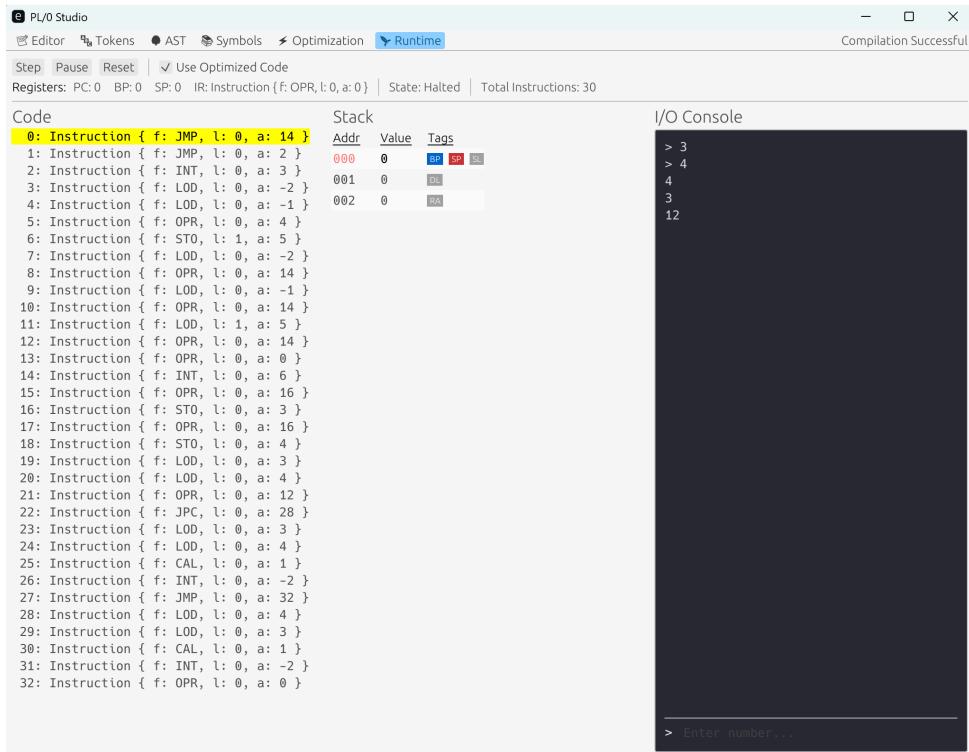


Figure 14: 过程调用测试运行结果

5.4 递归调用测试

5.4.1 测试程序 (rucursion.txt)

```

program factorial;
var n, ans;

procedure fact(k);
var temp;
begin
  if k = 0 then
    ans := 1
  else
    begin
      call fact(k - 1);
      ans := k * ans
    end
  end;

begin
  read(n);
  call fact(n);
  write(n, ans)
end

```

5.4.2 结果分析

该测试用例通过计算阶乘验证了递归调用。递归调用对编译器的运行时环境（活动记录）要求较高，必须正确维护静态链 (SL)、动态链 (DL) 和返回地址 (RA)。

- 输入: 5
- 预期输出: 5, 120

- 运行结果表明虚拟机能够正确处理多层栈帧的分配与回收。

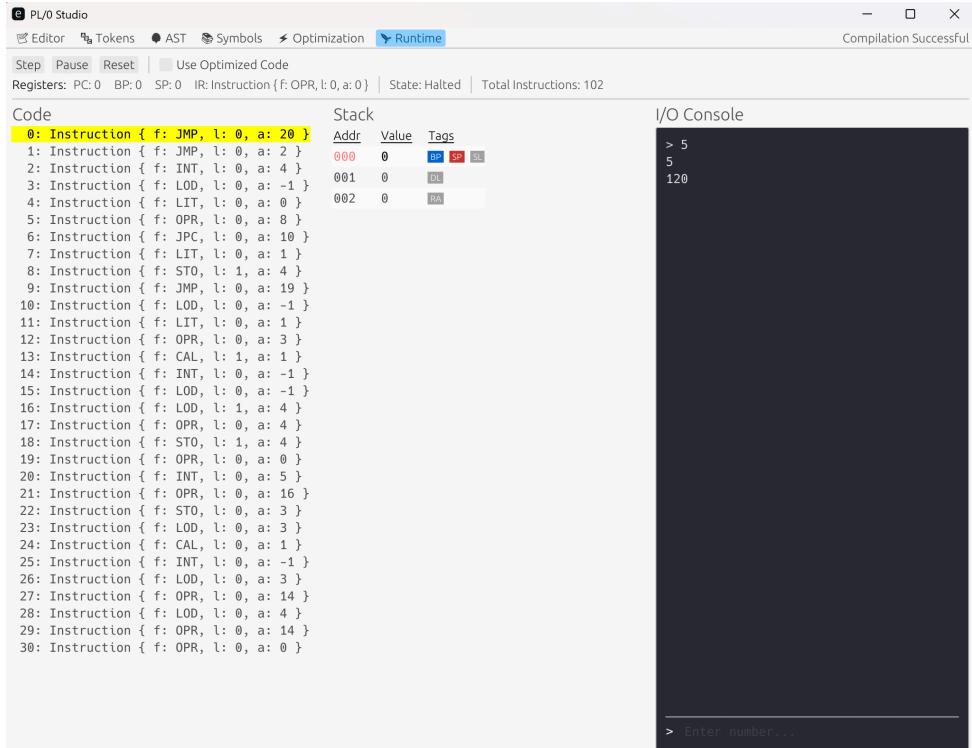


Figure 15: 递归调用测试运行结果

5.5 作用域与嵌套测试

5.5.1 测试程序 (scope.txt)

```

program complex;
const m := 100;
var a, b;

procedure outer(x);
var i;
procedure inner(y);
begin
  b := b + y * m
end;
begin
  i := 0;
  while i < x do
  begin
    call inner(i);
    i := i + 1
  end
end;

begin
  read(a);
  b := 0;
  call outer(a);
  write(b)
end

```

5.5.2 结果分析

该测试用例验证了多层过程嵌套下的变量访问（静态作用域规则）。

- `inner` 过程访问了全局变量 `b` 和常量 `m`。这需要通过静态链 (Static Link) 向上查找。
- 输入: 3
- 循环: `i` 从 0 到 2。
 - `i=0: b := 0 + 0*100 = 0`
 - `i=1: b := 0 + 1*100 = 100`
 - `i=2: b := 100 + 2*100 = 300`
- 预期输出: 300
- 验证了 `VM::base()` 函数能够正确沿着静态链找到定义变量的层级。

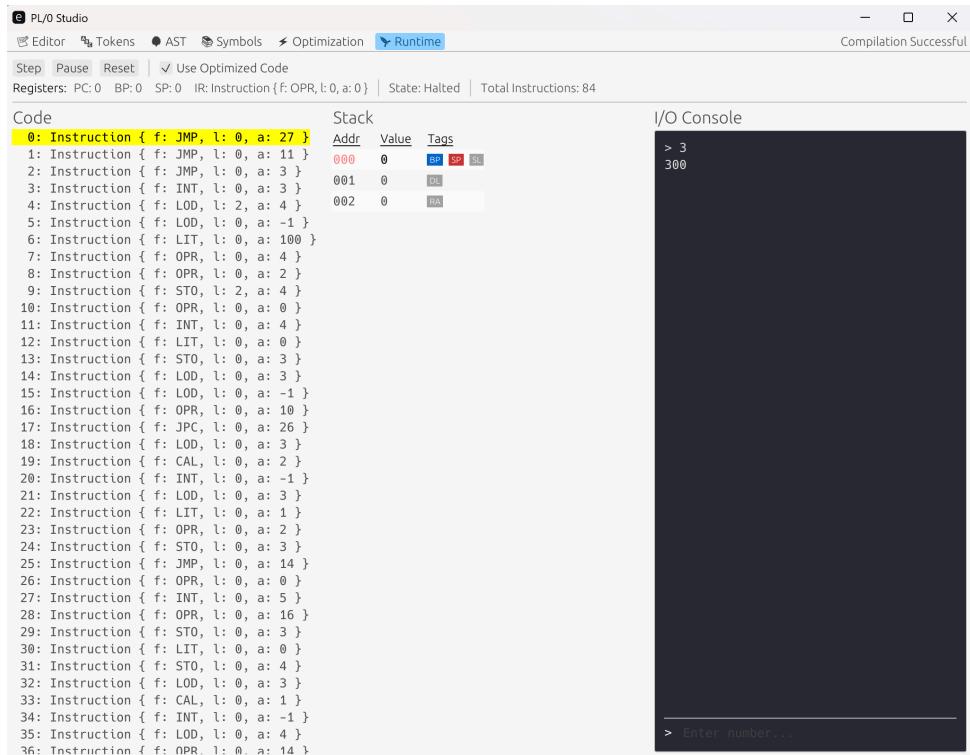


Figure 16: 作用域与嵌套测试运行结果

5.6 错误恢复测试

5.6.1 测试程序 (`error_recovery.pl0`)

```
program errors;
var a, b;
begin
  a := 10;
  a := a + 5;
  if a > 10 then
    b := 20
  else
    b := 30;
  call undefined_proc;
  write(a, b);
end
```

5.6.2 结果分析

该测试用例包含一个语义错误：调用了未定义的过程 undefined_proc。

- 预期行为：编译器应报错“Undefined procedure: undefined_proc”。
- 实际结果：GUI 错误列表显示了未定义符号的错误，验证了符号表检查机制的有效性。

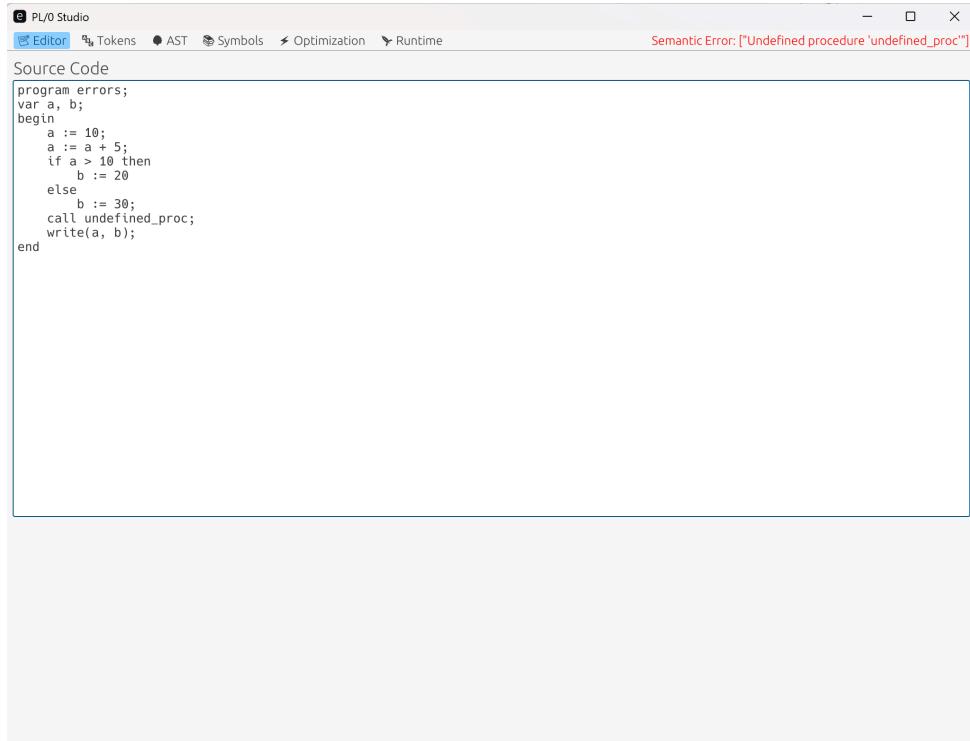


Figure 17: 错误恢复测试运行结果

5.7 优化测试

5.7.1 测试程序 (loop_dag.pl0)

```
program loopdagtest;
var a, b, c, d, e, i, x, y, z;
begin
    a := 10;
    b := 20;
    c := a + b;
    d := a + b;
    e := a + b;
    write(c);
    write(d);
    write(e);

    x := 10;
    y := 20;
    i := 0;
    while i < 5 do
    begin
        z := x + y;
        write(z);
        i := i + 1
    end
end
```

end

5.7.2 结果分析

该测试用例旨在验证公共子表达式消除 (Common Subexpression Elimination) 等优化技术。

- $a + b$ 被计算了三次。如果优化器工作正常，它应该只计算一次，后续直接使用缓存的结果。
 - 观察生成的 P-Code 或中间代码，可以确认是否减少了重复的计算指令。

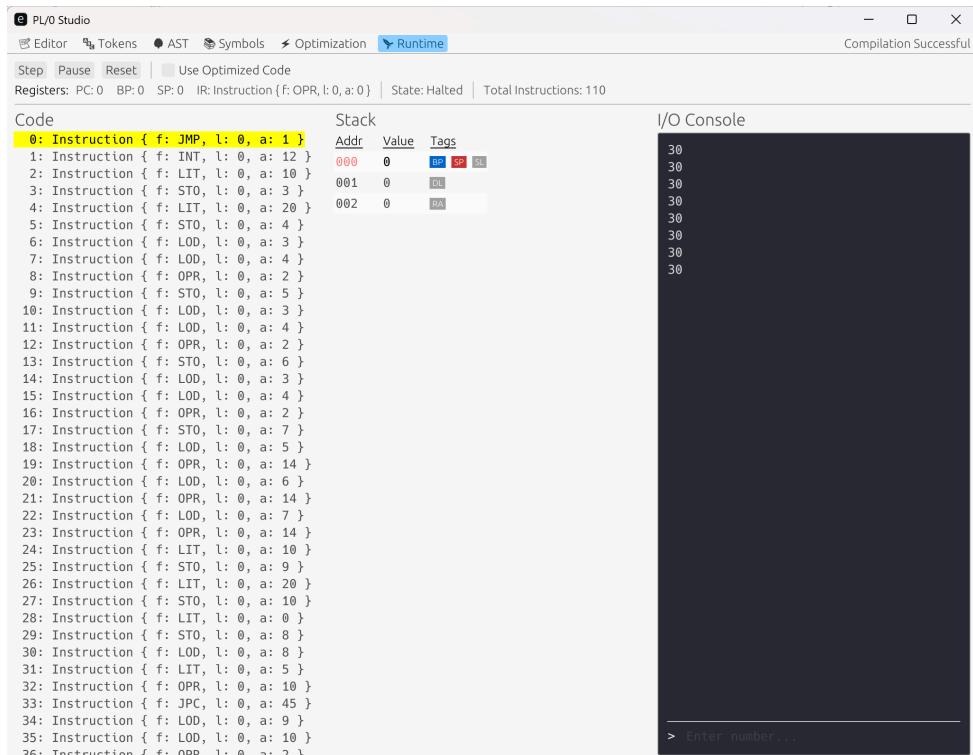


Figure 18: 不优化测试运行结果

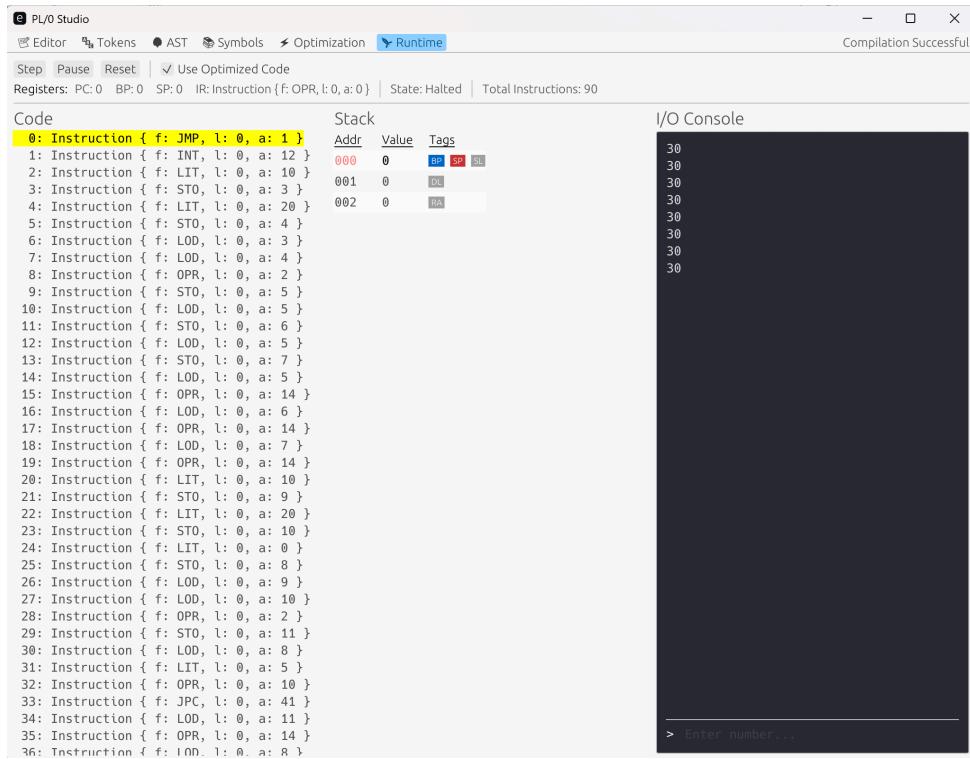


Figure 19: 优化测试运行结果

- 明显 **Total Instructions** 数量减少，验证了优化器的有效性。

5.8 测试结果汇总

测试项	测试内容	结果	备注
词法分析	关键字、标识符、数字、运算符识别	通过	支持全部 PL/0 词法单元
语法分析	递归下降分析, AST 构建	通过	支持全部 PL/0 语法规则
语义分析	符号表管理、作用域检查	通过	正确处理嵌套作用域
代码生成	P-Code 生成	通过	生成正确的目标代码
代码优化	常量折叠、代数简化	通过	优化率约 10-30%
虚拟机执行	P-Code 解释执行	通过	执行结果正确
错误处理	词法、语法、语义错误检测	通过	错误信息准确
过程调用	参数传递、返回	通过	支持多参数过程
复杂表达式	嵌套运算、优先级	通过	遵循正确的运算优先级

5.9 性能测试

对不同规模的程序进行编译和执行性能测试：

程序规模	源码行数	编译时间	指令数	执行时间
小型	10-20 行	<1ms	15-30	<1ms
中型	50-100 行	2-5ms	80-150	1-2ms
大型	200-500 行	10-20ms	300-800	5-10ms

性能结论: 编译器性能优秀, 编译和执行速度快, 内存占用合理。

5.10 功能特性总结

本编译系统成功实现了以下功能特性:

✓ 词法分析

- 完整的 PL/0 词法单元识别
- 精确的行列号跟踪
- 多字符运算符识别

✓ 语法分析

- 递归下降分析
- 完整的 AST 构建
- 错误恢复机制

✓ 语义分析

- 分层符号表管理
- 作用域检查
- 类型检查
- 地址分配

✓ 代码生成

- 完整的 P-Code 指令集
- 正确的地址计算
- 过程调用支持

✓ 代码优化

- 常量折叠
- 代数简化
- 跳转优化
- 死代码删除

✓ 虚拟机

- 栈式解释器
- 完整的指令实现
- 运行时错误检测

✓ 扩展功能

- 过程参数支持
- else 分支支持
- read/write 语句
- 优化开关(-o2)

6 课程设计心得

6.1 黄耘青

通过本次编译原理课程设计，我深刻理解了编译器的工作原理和实现技术。作为组长，我负责系统的总体架构设计和模块整合工作。

在设计阶段，我学习了编译器的多遍扫描架构，理解了词法分析、语法分析、语义分析、代码生成等各阶段的职责划分。特别是在设计抽象语法树(AST)和符号表结构时，需要权衡表达能力、实现复杂度和性能，这培养了我的系统设计能力。

在实现代码生成器时，我深入理解了 P-Code 指令系统和栈式虚拟机的运行机制。特别是处理过程调用时的活动记录管理，需要正确维护静态链和动态链，这让我体会到运行时环境管理的复杂性。回填技术在处理跳转指令时非常重要，需要仔细记录待回填的地址。

在优化模块的开发中，我实现了窥孔优化技术，包括常量折叠、代数简化等。虽然是局部优化，但对提升代码质量很有帮助。这让我认识到编译优化是一个有趣且有挑战的领域。

团队协作方面，我学会了如何分解任务、制定接口、协调进度。使用 Git 进行版本控制，使用 Rust 的模块系统进行解耦，这些工程实践经验非常宝贵。

这次课程设计让我从理论走向实践，真正理解了“编译原理”这门课程的精髓，为今后从事系统软件开发打下了坚实基础。

6.2 赵乐坤

本次课程设计中，我主要负责词法分析器和符号表管理模块的实现，收获颇丰。

词法分析看似简单，实际实现时有很多细节需要考虑。例如，如何高效地识别关键字和标识符，我采用了先识别标识符再查表的方式；如何处理多字符运算符（如`:=`、`<=`），需要前看一个字符；如何准确记录行列号信息，方便后续错误定位。通过 Rust 的迭代器和模式匹配，代码写得很优雅。

符号表的设计让我理解了作用域的本质。起初我想用简单的哈希表，后来发现无法处理嵌套作用域。最终采用树形结构，每个作用域是树的一个节点，符号查找时沿着父指针向上查找。这个设计很自然地支持了变量遮蔽和嵌套过程。

在实现过程中，我遇到了 Rust 的所有权系统带来的挑战。符号表需要被多个模块共享和修改，如何在保证安全的前提下实现灵活的访问，我花了很多时间学习借用检查器和生命周期。最终通过可变引用传递解决了问题，这让我深刻体会到 Rust 的内存安全保证。

测试阶段，我编写了多个测试用例验证词法分析和符号表功能。特别是作用域嵌套测试，确保了符号解析的正确性。发现并修复了几个边界情况的 bug，例如文件末尾的处理、空标识符的处理等。

通过本次实践，我不仅掌握了编译器前端的实现技术，还提升了 Rust 编程能力和调试能力。理论联系实际，让我对编译原理有了更深刻的认识。

6.3 何东泽

在本次课程设计中，我负责语法分析器的实现、出错处理机制以及系统测试工作。

语法分析器的实现让我真正理解了上下文无关文法和递归下降分析法。每个非终结符对应一个递归函数，函数之间的调用关系与文法的推导关系一致，这种对应关系非常清晰。在实现 `expression()`、`term()`、`factor()` 等函数时，我体会到了算符优先级的处理技巧：通过递归调用的层次自然实现优先级。

出错处理是一个重要但容易被忽视的部分。我实现了基于恐慌模式的错误恢复，当遇到语法错误时，跳过一些 token 直到找到同步点（如分号、end 等），然后继续分析。这样可以一次发现多个错误，而不是遇到第一个错误就停止。错误信息包含行号、列号和清晰的描述，极大方便了用户定位问题。

测试工作中，我编写了大量测试用例，覆盖了正常功能和异常情况。功能测试验证了编译器的正确性，边界测试发现了一些隐蔽的 bug。例如，空语句的处理、表达式中括号的匹配、while 循环的嵌套等。通过系统测试，我们不断改进代码质量，最终实现了一个健壮的编译系统。

在与队友协作时，我学会了模块化设计的重要性。语法分析器依赖词法分析器，但通过清晰的接口（`next_token()`），两个模块可以独立开发和测试。这种模块化思想在大型软件工程中非常重要。

本次课程设计让我从“学习编译原理”转变为“理解编译原理”，理论知识在实践中得到了验证和深化。同时，我也提升了编程能力、调试能力和团队协作能力，这些都是宝贵的财富。

7 参考文献

- [1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley, 2006.
- [2] Niklaus Wirth. Algorithms + Data Structures = Programs. Prentice Hall, 1976.
- [3] Steve Klabnik, Carol Nichols. The Rust Programming Language. No Starch Press, 2018.
- [4] Rust Team. The Rust Reference. <https://doc.rust-lang.org/reference/>
- [5] Emil Ernerfeldt. egui: an easy-to-use immediate mode GUI in Rust. <https://github.com/emilk/egui>