ToyC 编译器实现实践报告

编译系统实践小组

2025 年 8 月 7 日

1 项目概述

1.1 项目背景

本项目是武汉大学计算机学院编译系统实践课程的作业,要求四人为一组实现 ToyC 语言的编译器。ToyC 是 C 语言的一个简化子集，包含基本的数据类型、控制结构和函数定义等核心功能。

1.2 项目目标

- 实现完整的 ToyC 语言编译器
- 支持从 ToyC 源代码到 RISC-V32 汇编代码的编译
- 通过所有 15 个官方测试用例
- 实现基本的代码优化算法

1.3 技术选型

- **编程语言**: C++20
- **构建系统**: Make/CMake
- **词法/语法分析**: 手写递归下降分析器
- **目标架构**: RISC-V32

2 ToyC 语言规范

2.1 语法特性

ToyC 语言支持以下核心语法特性：

2.1.1 数据类型

- `int`: 32 位有符号整数
- `void`: 空类型（仅用于函数返回类型）

2.1.2 语句类型

- 变量声明: `int x = 10;`
- 赋值语句: `x = y + 5;`
- 表达式语句: `func();`
- 控制流: `if-else`, `while`, `break`, `continue`, `return`
- 语句块: `{ ... }`

2.1.3 表达式

- 算术运算: `+`, `-`, `*`, `/`, `%`
- 关系运算: `<`, `>`, `<=`, `>=`, `==`, `!=`
- 逻辑运算: `&&`, `||`, `!`
- 函数调用: `func(arg1, arg2)`
- 括号表达式: `(expr)`

2.2 语义约束

- 程序必须包含 `int main()` 函数作为入口点
- 函数只能声明在全局作用域中
- 变量必须先声明后使用
- 支持函数递归调用
- 遵循 C 语言的作用域规则

3 编译器架构设计

3.1 整体架构

编译器采用经典的多阶段设计，包含以下主要模块：

词法分析	语法分析	语义分析	代码生成	优化
ManualLexer	ManualParser	SemanticAnalyzer	CodeGenerator	Optimizer
Token 流	AST	符号表	RISC-V 汇编	优化代码

图 1: 编译器架构流程图

3.2 模块设计

3.2.1 抽象语法树 (AST)

设计了完整的 AST 节点层次结构：

- **基类**: `ASTNode`, `Expr`, `Stmt`
- **表达式节点**: `BinaryExpr`, `UnaryExpr`, `NumberExpr`, `VarExpr`, `CallExpr`
- **语句节点**: `Block`, `IfStmt`, `WhileStmt`, `ReturnStmt`, `VarDecl`, `AssignStmt`
- **顶层节点**: `CompUnit`, `FuncDef`

3.2.2 符号表管理

实现了层次化的符号表系统：

- 支持嵌套作用域管理
- 变量和函数的符号信息存储
- 栈空间分配管理
- 类型检查支持

4 实现细节

4.1 词法分析器

手写实现的词法分析器 `ManualLexer` 包含以下功能：

```
class ManualLexer {
public:
    Token next_token();              // 获取下一个token
    std::vector<Token> tokenize();   // 完整词法分析

private:
    Token read_number();             // 识别数字
    Token read_identifier();         // 识别标识符和关键字
```

```
    void skip_whitespace();          // 跳过空白字符
    void skip_comment();             // 跳过注释
};
```

Listing 1: 词法分析器核心方法

关键特性：

- 支持 C 风格单行 (//) 和多行 (/* */) 注释
- 正确识别所有 ToyC 运算符和关键字
- 提供详细的行号和列号信息用于错误报告

4.2 语法分析器

采用递归下降的语法分析方法 `ManualParser`：

```
class ManualParser {
private:
    // 语法分析方法对应文法规则
    std::unique_ptr<CompUnit> parse_comp_unit();
    std::unique_ptr<FuncDef> parse_func_def();
    std::unique_ptr<Block> parse_block();
    std::unique_ptr<Stmt> parse_stmt();

    // 表达式解析按优先级组织
    std::unique_ptr<Expr> parse_lor_expr();      // 逻辑或
    std::unique_ptr<Expr> parse_land_expr();     // 逻辑与
    std::unique_ptr<Expr> parse_rel_expr();      // 关系运算
    std::unique_ptr<Expr> parse_add_expr();      // 加减运算
    std::unique_ptr<Expr> parse_mul_expr();      // 乘除模运算
    std::unique_ptr<Expr> parse_unary_expr();    // 一元运算
    std::unique_ptr<Expr> parse_primary_expr(); // 基本表达式
};
```

Listing 2: 语法分析器结构

设计亮点：

- 按运算符优先级组织表达式解析
- 完整的错误恢复机制
- 支持 ToyC 语言的所有语法结构

4.3 语义分析器

SemanticAnalyzer 实现了完整的语义检查：

```
class SemanticAnalyzer : public ASTVisitor {
public:
    bool analyze(CompUnit& root);        // 分析入口
    const std::vector<SemanticError>& get_errors() const;

private:
    // 类型检查
    DataType analyze_expression(Expr& expr);
    bool check_type_compatibility(DataType expected, DataType actual);

    // 作用域管理
    void enter_function_scope(const std::string& func_name, DataType
        return_type);
    void exit_function_scope();

    // 控制流检查
    void enter_loop();
    void exit_loop();
    bool is_in_loop() const;
};
```

Listing 3: 语义分析器核心功能

检查内容：

- 变量先声明后使用
- 函数先定义后调用
- 类型匹配检查
- 返回语句有效性
- break/continue 语句只能在循环中使用

4.4 代码生成器

CodeGenerator 负责生成 RISC-V32 汇编代码：

```
class CodeGenerator : public ASTVisitor {
private:
    std::ostringstream output;           // 汇编代码输出
    SymbolTable* symbol_table;          // 符号表引用

```

```
    // 寄存器管理
    std::vector<std::string> temp_registers;
    std::vector<bool> register_used;

    // 栈管理
    int current_function_stack_size;

    // 标签管理
    std::stack<std::string> break_labels;
    std::stack<std::string> continue_labels;

public:
    std::string generate_expr(Expr& expr);   // 表达式代码生成
    void setup_function_prologue(...);       // 函数序言
    void setup_function_epilogue();          // 函数尾声
};
```

Listing 4: 代码生成器架构

生成特性：

- 完整的 RISC-V32 指令集支持
- 寄存器分配和管理
- 函数调用约定实现
- 短路求值优化
- 控制流代码生成

4.5 优化器

Optimizer 实现了基本的编译时优化：

```
class Optimizer : public ASTVisitor {
private:
    bool optimization_enabled;
    int optimizations_applied;

    // 常量折叠
    std::unique_ptr<Expr> constant_folding(BinaryExpr& expr);
    std::unique_ptr<Expr> constant_folding(UnaryExpr& expr);

    // 表达式简化
    std::unique_ptr<Expr> simplify_expression(std::unique_ptr<Expr> expr);

```

```
    // 死代码消除
    void eliminate_dead_code(Block& block);

    // 控制流优化
    std::unique_ptr<Stmt> optimize_if_statement(IfStmt& if_stmt);
    std::unique_ptr<Stmt> optimize_while_statement(WhileStmt& while_stmt);
};
```

Listing 5: 优化算法实现

优化技术：

- 常量折叠：编译时计算常量表达式
- 代数简化：`x + 0 = x`, `x * 1 = x`, `x * 0 = 0`
- 死代码消除：移除不可达代码
- 控制流优化：常量条件的 if/while 语句简化

5 测试与验证

5.1 测试用例覆盖

编译器通过了所有 15 个官方测试用例：

5.2 测试结果

使用自动化测试脚本进行批量测试：

```
=== ToyC编译器批量测试 ===
测试 01_minimal.tc:    编译成功
测试 02_assignment.tc:    编译成功
测试 03_if_else.tc:    编译成功
...（省略中间结果）
测试 15_multiple_return_paths.tc:    编译成功

=== 测试结果 ===
总计: 15 个测试用例
通过: 15 个
失败: 0 个
  所有测试用例通过!
```

Listing 6: 批量测试结果

测试覆盖率: 100% (15/15)

编号	测试文件	测试功能
01	minimal.tc	最基本的 main 函数
02	assignment.tc	变量赋值
03	if_else.tc	条件分支
04	while_break.tc	循环和 break
05	function_call.tc	函数调用
06	continue.tc	continue 语句
07	scope_shadow.tc	作用域和变量覆盖
08	short_circuit.tc	短路求值
09	recursion.tc	递归函数
10	void_fn.tc	void 函数
11	precedence.tc	运算符优先级
12	division_check.tc	除法运算
13	scope_block.tc	语句块作用域
14	nested_if_while.tc	嵌套结构
15	multiple_return_paths.tc	多返回路径

表 1: 测试用例汇总

5.3 生成代码示例

以阶乘函数为例展示生成的 RISC-V 汇编代码质量：

```
int fact(int n) {
    if (n <= 1) {
        return 1;
    } else {
        return n * fact(n - 1);
    }
}

int main() {
    return fact(5);
}
```

Listing 7: ToyC 源代码示例

生成的汇编代码正确实现了：

- 函数调用约定
- 栈帧管理
- 条件分支

- 递归调用
- 寄存器分配

6 性能优化

6.1 编译时优化

实现的优化算法在测试中应用了多项优化：

- **常量折叠**: 将 `3 + 4` 直接计算为 `7`
- **代数简化**: 将 `x + 0` 简化为 `x`
- **死代码消除**: 移除 return 语句后的不可达代码
- **控制流优化**: 优化常量条件的分支语句

6.2 代码质量

生成的 RISC-V 汇编代码特点：

- 正确的函数调用约定
- 高效的寄存器使用
- 合理的栈空间管理
- 支持标准的 RISC-V32 指令集

7 项目总结

7.1 完成情况

- 完整实现 ToyC 语言编译器
- 所有 15 个测试用例 100% 通过
- 支持 RISC-V32 汇编代码生成
- 实现基本的编译时优化
- 完善的错误检测和报告

7.2 技术亮点

1. **手写实现**: 完全手写的词法和语法分析器，深入理解编译原理
2. **模块化设计**: 清晰的模块划分和接口设计
3. **访问者模式**: 使用访问者模式实现 AST 遍历，便于扩展
4. **错误处理**: 完整的错误检测、定位和报告机制
5. **优化算法**: 实现了多种经典的编译优化技术

7.3 学习收获

通过本项目实践，团队成员深入理解了：

- 编译器的完整工作流程
- 词法分析和语法分析的实现原理
- 语义分析和符号表管理
- 代码生成和目标机器指令
- 编译优化的基本技术
- 大型软件项目的协作开发

7.4 项目特色

- **完全原创**: 所有核心组件均为原创实现，未使用任何现成框架
- **高度完整**: 涵盖编译器的所有核心阶段
- **测试驱动**: 通过完整的测试用例验证功能正确性
- **文档完善**: 提供详细的设计文档和使用说明