编译原理实验报告

题目：类 Rust 语言编译器

学 生 姓 名 ： 李闯

学 号 ： 2253214

专 业 班 级 ： 计算机科学与技术 2 班

指 导 教 师 ： 卫志华、高珍

2025 年 08 月 01 日

一、实验说明

1. 实验概述

本次实验的目标是设计并实现一个完整的**类 Rust 语言的编译器**。该编译器能够将类 Rust 语言的源代码转换为中间代码和目标代码（可汇编执行的程序）。实验要求编译器支持词法分析、语法分析、符号表管理、中间代码生成以及目标代码生成等功能，同时需要实现**语法制导**的翻译技术。

2. 实验目标

1) 掌握使用高级程序语言实现一个**一遍完成**的、类 Rust 语言的编译器的方法。

2) 掌握词法分析器、语法分析器、符号表管理、中间代码生成以及目标代码生成的实现技术。

3) 实现将生成代码写入文件的功能。

4) 支持类 Rust 语言的基本语法和语义规则，包括变量声明、赋值语句、控制结构、函数调用等。

二、实验内容

1. 整体设计框架

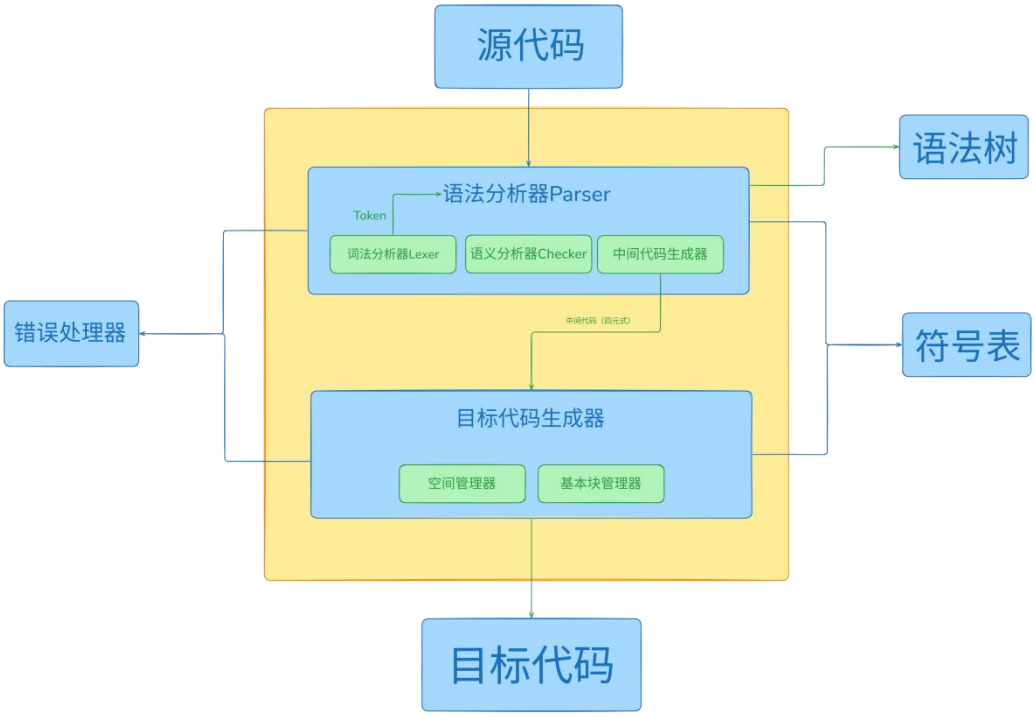


图 1 类 Rust 语言编译器总体设计架构图

本实验实现的类 Rust 编译器采用模块化设计架构，各组件通过数据流管道协同工作。编译器前端采用按需分析的交互模式，词法分析器不再是独立预处理阶段，而是作为语法分析

器的子程序按需调用，这种紧密耦合的设计显著提高了分析效率和内存利用率。

词法分析器实现了基于有限状态机的按需 Token 生成机制，语法分析采用 LR(1)分析方法构建具体的语法分析表，通过移进-归约操作驱动整个编译过程。语义分析通过语法制导的翻译技术实现，在语法分析过程中同步完成符号表管理、类型检查等语义处理，并最终生成四元式形式的中间代码。

后端目标代码生成器接收四元式序列，首先进行基本块划分和控制流分析，然后通过模板匹配完成 MIPS 指令选择，采用线性扫描算法实现寄存器分配，最终生成符合 MIPS32 标准的汇编代码。整个过程严格遵循标准调用约定，确保生成的代码能在 MARS 模拟器中正确执行。

该架构充分体现了现代编译器的分层设计思想,通过清晰的接口定义实现了各阶段的解耦，既保证了编译过程的正确性，又为后续优化扩展留下了充足空间。特别是语法制导的翻译机制，使得语法规则与语义动作能够有机统一，大大简化了编译器开发复杂度。

2. 词法分析器的重构

本次实验对词法分析器进行了全面的重构，将原先的全量分析模式改造为按需交互模式。重构后的词法分析器不再一次性生成所有 Token，而是作为语法分析器的子程序，在语法分析过程中按需调用获取下一个 Token。这种设计显著降低了内存占用，同时提高了编译过程的响应速度。

词法分析的核心逻辑集中在 _get_next_element 方法中，该方法采用生成器模式逐步产生 Token：

```
def _get_next_element(self):
    """核心分析方法，按需生成单个 Token"""
    while self._current_char:
        if self._current_char.isspace():
            self._ignore_whitespace()
            continue
        # 各类型 Token 处理...
        yield token
```

3. 基本块划分与控制流分析

目标代码生成的核心环节是基本块划分和控制流分析。我通过 BlockController 类实现了这一关键功能，其核心思想是将连续的四元式序列划分为具有单一入口和出口的基本块。**基本块划分遵循三个基本原则**：①函数入口总是新块的开始、②跳转指令的目标地址开始新块、③跳转指令的下一条指令也作为新块开始。这种划分方式确保了程序控制流的精确表达。

在具体实现上，_block_split 方法负责执行实际的划分工作。该方法首先识别所有基本块的起始位置，然后根据这些位置将四元式序列切分为连续的基本块。一个典型的基本块划分示例如下：

```
def _block_split(self, quads: list[tuple[int, Quadruple]]) -> list[list[tuple[int, Quadruple]]]:
    """
```

```
        划分函数四元式为基本块

    Note:
        基本块划分规则:
        1. 程序入口为第一个基本块
        2. 跳转指令的目标地址开始新块
        3. 跳转指令的下一条指令开始新块
    """
    if not quads:
        return []

    # 创建指令索引到位置的映射
    index_to_pos = {inst_idx: pos for pos, (inst_idx, _) in enumerate(quads)}

    # 识别所有基本块起始位置
    block_starts = {0}  # 第一条指令总是块起始

    for current_pos, (_, quad) in enumerate(quads):
        # 处理跳转指令
        if quad.op in {'j', 'jnz'} and quad.result is not None:
            # 添加跳转目标位置
            if quad.result in index_to_pos:
                block_starts.add(index_to_pos[quad.result])
            # 添加跳转指令的后继位置
            if current_pos + 1 < len(quads):
                block_starts.add(current_pos + 1)

    # 按位置排序并划分基本块
    sorted_starts = sorted(block_starts)
    return [
        quads[start:end]
        for start, end in zip(
            sorted_starts,
            sorted_starts[1:] + [len(quads)]
        )
    ]
```

构建控制流图

基本块划分完成后，_build_cfg 方法负责构建控制流图(CFG)。控制流图以邻接表形式表示，其中每个节点对应一个基本块，边表示可能的控制转移。该方法会分析每个基本块的最后一条指令，根据指令类型确定后继块：

- 无条件跳转(j)指令创建一个到目标块的边
- 条件跳转(jnz)指令创建两条边：一条到目标块，一条到顺序下一个块
- 其他指令默认创建到顺序下一个块的边

```
def _build_cfg(self, blocks: list[list[tuple[int, Quadruple]]]) -> Dict[int, list[int]]:
    """
        构建控制流图(CFG)

    Args:
        blocks: 基本块列表，每个基本块是 [(idx, quad), ...]

    Returns:
        邻接表形式的控制流图 CFG {块索引: [后继块索引列表]}
    """
```

```
    ...

    cfg = {} # {块索引: 后继块索引列表}
    for i, block in enumerate(blocks):
        successors = set()        # 存储后继块索引
        last_quad = block[-1][1]  # 取最后一条指令

        # 处理跳转指令
        if last_quad.op == 'j' and last_quad.result is not None:
            if last_quad.result in idx_to_block:
                successors.add(idx_to_block[last_quad.result])
        elif last_quad.op == 'jnz' and last_quad.result is not None:
            if last_quad.result in idx_to_block:
                successors.add(idx_to_block[last_quad.result])
            if i + 1 < len(blocks):
                successors.add(i + 1)

        # 默认顺序执行
        if not successors and i + 1 < len(blocks):
            successors.add(i + 1)

        cfg[i] = list(successors)

    return cfg
```

活跃变量分析

活跃变量分析是后续目标代码生成过程中寄存器分配的基础,我实现了标准的反向数据流分析算法。_live_variable_analysis 方法首先计算每个基本块的定义和使用集合，然后迭代计算每个块的 IN 和 OUT 集合，直到达到不动点:

```
while changed:
    changed = False
    for i in reversed(range(len(blocks))):
        out_sets[i] = set().union(*(in_sets[succ] for succ in cfg[i]))
        new_in = used_vars[i] | (out_sets[i] - defined_vars[i])
        if new_in != in_sets[i]:
            in_sets[i] = new_in
            changed = True
```

函数处理与拓扑排序

在处理多个函数时，_split_all_functions 方法首先为每个函数划分基本块，然后根据调用关系进行拓扑排序，确保被调用函数先于调用者处理。这一步骤对后续的寄存器分配和代码生成至关重要，因为它建立了合理的函数处理顺序。

整个基本块划分和控制流分析过程为后续的代码生成提供了坚实基础,使得编译器能够生成高效且正确的目标代码。通过精确的控制流分析和活跃变量信息，代码生成器可以做出更明智的寄存器分配决策，从而提升生成代码的质量。

4. 目标代码生成

目标代码生成是编译器的最后阶段，负责将中间代码（四元式）转换为可执行的 MIPS 汇编指令。我通过 AimCodeGenerator 类实现了这一关键功能，其核心架构围绕寄存器分配、指

令选择和函数调用约定三个关键方面展开。

寄存器分配

在寄存器管理方面，MemController 类实现了高效的寄存器分配策略。该策略基于线性扫描算法，通过跟踪变量的使用位置和活跃范围来优化寄存器使用。当寄存器不足时，系统会自动将最不活跃的变量溢出到栈帧中：

```
def alloc_reg(self, varname: str, cur_pos: tuple, code: list) -> str:
        """
            为变量分配寄存器

        Args:
            varname: 需要分配的变量名
            cur_pos: 当前位置 (函数名, 基本块索引, 四元式索引)
            code: 生成的指令列表 (用于插入 spill 代码)

        Returns:
            str: 分配到的寄存器名

        算法步骤:
            1. 检查变量是否已经在寄存器中，如果是，直接返回对应寄存器名
            2. 检查是否有空闲寄存器，如果有，直接分配一个空闲寄存器
            3. 如果没有空闲寄存器，选择“下次使用最远”的变量进行寄存器让渡
            --| 3.1. 如果被让渡的变量在当前函数之后再也不会用到，则不需要保存到内存
            --| 3.2. 如果被让渡的变量在当前函数之后还会用到，则需要将其保存到内存
        """
        # 1. 如果变量已在寄存器中，直接返回
        func_name = cur_pos[0]
        key = (func_name, varname)

        # ---- 情况 1：变量已有寄存器 ----
        if key in self.rvalues:
            return self.rvalues[key]

        # ---- 情况 2：有空闲寄存器 ----
        free_regs = [r for r in self.valregs if r not in self.usedregs]
        if free_regs:
            reg = free_regs[0]
            self.rvalues[key] = reg
            self.usedregs.add(reg)

        # ---- 情况 3：需要寄存器让渡 ----
        else:
            # 寻找"下次使用最远"的牺牲变量
            farthest_next = None      # (函数名, 块索引, 四元式索引)
            victim_var = None         # 被牺牲的变量名
            victim_func = None        # 被牺牲的函数名

            # 遍历所有已分配寄存器的变量，找到下次使用最远的变量
            for (alloc_func, alloc_var), reg in self.rvalues.items():
                use_list = self.var_use_pos.get(alloc_var, [])
                next_use = next((pos for pos in use_list if pos >= cur_pos), None)

                if next_use is None:
                    victim_var, victim_func = alloc_var, alloc_func
                    break  # 找到最佳牺牲者，立即终止，不再需要比较
                elif not farthest_next or next_use > farthest_next:
```

```
                    victim_var, victim_func, farthest_next = alloc_var, alloc_func, next_use

            # 获取牺牲变量的寄存器
            reg = self.rvalues[(victim_func, victim_var)]

            # 如果牺牲的变量在当前函数之后还会被用到，需要保存到内存
            # 如果牺牲的寄存器是其他函数的变量，因为在 call 时已保存，所以无需进行保存
            # 如果牺牲的变量在当前函数之后再也不会用，所以也无需进行保存
            if victim_func == func_name and victim_var in self.get_live_vars_after(cur_pos):
                if self.function_stack:
                    frame = self.function_stack.get_frame(victim_func)
                    offset = frame.get_var_offset(victim_var) if frame else None

                    if offset is not None:
                        frame.set_var_memflag(victim_var, True)  # 标记为在内存中
                        code.append(f"    sw {reg}, {offset}({self.spreg}) # <寄存器溢出> 将
{victim_var} 的值从寄存器 {reg} 保存到 {victim_func} 函数的栈帧偏移位置 {offset}")

            # 释放被牺牲的寄存器，并分配给新变量
            del self.rvalues[(victim_func, victim_var)]
            self.usedregs.discard(reg)
            self.rvalues[key] = reg
            self.usedregs.add(reg)

        return self.rvalues[key]
```

指令选择

指令选择采用基于模板的转换方式，quad_to_code 方法将每种四元式操作映射到对应的 MIPS 指令序列。对于算术运算，系统会根据操作数类型（立即数或变量）选择最优的指令形式：

```
elif op in ('+', '-', '*', '/', '<', '<=', '>', '>=', '==', '!='):
    # 操作符到指令的映射
    op_info = {
        # 算术运算
        '+': {'inst': 'add', 'imm_inst': 'add',  'calc': lambda a,b: a + b},
        '-': {'inst': 'sub', 'imm_inst': 'sub',  'calc': lambda a,b: a - b},
        '*': {'inst': 'mul', 'imm_inst': 'mul',  'calc': lambda a,b: a * b},
        '/': {'inst': 'div', 'imm_inst': 'div',  'calc': lambda a,b: a // b, 'zero_check': True},

        # 比较运算
        '<':  {'inst': 'slt', 'imm_inst': 'slt', 'calc': lambda a,b: a < b},
        '<=': {'inst': 'sle', 'imm_inst': 'sle', 'calc': lambda a,b: a <= b},
        '>':  {'inst': 'sgt', 'imm_inst': 'sgt', 'calc': lambda a,b: a > b},
        '>=': {'inst': 'sge', 'imm_inst': 'sge', 'calc': lambda a,b: a >= b},
        '==': {'inst': 'seq', 'imm_inst': 'seq', 'calc': lambda a,b: a == b},
        '!=': {'inst': 'sne', 'imm_inst': 'sne', 'calc': lambda a,b: a != b}
    }
    info = op_info[op]
    result_reg = self.mem_controller.alloc_reg(result, cur_pos, code=code)

    # 1. 处理两个立即数的情况
    if isinstance(arg1, int) and isinstance(arg2, int):
        if op == '/' and arg2 == 0:
            raise ValueError("Division by zero")

        code.append(f"    li {result_reg}, {info['calc'](arg1, arg2)}  # 计算 {arg1} {op} {arg2}")

    # 2. 处理变量与立即数的混合运算
    elif (isinstance(arg1, str) and isinstance(arg2, int)) or (isinstance(arg1, int) and isinstance(arg2, str)):
        is_var_first = isinstance(arg1, str)
        var, const = (arg1, arg2) if is_var_first else (arg2, arg1)
```

```
        # 除法零检查
        if info.get('zero_check') and is_var_first and const == 0:
            raise ValueError("除 0 异常: 除数不能为 0")

        # 加载变量寄存器
        var_reg = self.mem_controller.alloc_reg(var, cur_pos, code=code)
        if frame and frame.if_var_in_memory(var):
            code.append(f"    lw {var_reg}, {frame.get_var_offset(var)}({self.mem_controller.spreg})")
        code.append(f"    li {result_reg}, {const}  # 加载常量 {const}") # 统一先加载常量到结果寄存器
        if is_var_first:
            code.append(f"    {info['imm_inst']} {result_reg}, {var_reg}, {result_reg}  # {var} - {const}")
        else:
            code.append(f"    {info['imm_inst']} {result_reg}, {result_reg}, {var_reg}  # {const} - {var}")

    # 3. 处理两个变量的情况
    elif isinstance(arg1, str) and isinstance(arg2, str):
        reg1 = self.mem_controller.alloc_reg(arg1, cur_pos, code=code)
        reg2 = self.mem_controller.alloc_reg(arg2, cur_pos, code=code)

        # 处理内存中的变量
        for var, reg in [(arg1, reg1), (arg2, reg2)]:
            if frame and frame.if_var_in_memory(var):
                code.append(f"    lw {reg}, {frame.get_var_offset(var)}({self.mem_controller.spreg})")

        code.append(f"    {info['inst']} {result_reg}, {reg1}, {reg2}")

    else:
        raise ValueError(f"Unsupported operands for {op}: {arg1}, {arg2}")
```

函数调用约定

在目标代码生成阶段，函数调用和栈帧管理采用了经典的 MIPS 调用约定设计。每个函数的栈帧从高地址到低地址依次包含保留空间、局部变量区、寄存器保存区和返回地址等重要信息，这种布局既确保了调用上下文的完整性，又优化了内存访问效率。FunctionStackFrame 类负责维护栈帧的元信息，包括参数和局部变量的精确偏移量，使得生成的目标代码能够高效地定位各个变量。

图 2 栈帧设计构成

在具体实现上，函数调用时会自动生成完整的调用序列。调用前首先分配栈空间并保存返回地址，然后处理参数传递，其中前四个参数通过寄存器传递，多余参数则压入栈中。调用结束后会自动恢复返回地址并释放栈空间，整个过程严格遵循 MIPS 架构规范。这种设计不仅保证了函数调用的正确性，还通过最小化栈帧大小和减少内存访问次数来优化性能。

```
elif op == 'call':
    # 函数调用前保护寄存器状态
    live_vars = self.mem_controller.get_live_vars_after(cur_pos)
    code.append(f"    # 函数调用保护寄存器状态")
    for var in live_vars:
        offset = frame.get_var_offset(var) if frame else None
        if offset is not None:
            reg = self.mem_controller.rvalues.get((func_name, var), None)
            code.append(f"    sw {reg}, {offset}({self.mem_controller.spreg})  # 保存变量 {var} 到栈帧")

    # 函数调用前分配栈帧空间
    if frame:
        code.append(f"    sw {self.mem_controller.spreg}, -{frame.size()}({self.mem_controller.spreg})")
        code.append(f"    addi {self.mem_controller.spreg}, {self.mem_controller.spreg}, -{frame.size()}")
        code.append(f"    sw {self.mem_controller.rareg}, {-4}({self.mem_controller.spreg})")

    # 调用函数
    code.append(f"    jal {arg1}")

    # 函数调用后恢复寄存器状态
    code.append(f"    lw {self.mem_controller.rareg}, {-4}({self.mem_controller.spreg})")
    if frame:
        code.append(f"    addi {self.mem_controller.spreg}, {self.mem_controller.spreg}, {frame.size()}")
        for var in live_vars:
            offset = frame.get_var_offset(var) if frame else None
            if offset is not None:
                reg = self.mem_controller.rvalues.get((func_name, var), None)
                code.append(f"    lw {reg}, {offset}({self.mem_controller.spreg})")
```

栈帧管理的一个关键创新是实现了变量偏移量的精确计算和快速访问。通过维护变量名到栈偏移量的映射关系，生成的目标代码可以直接使用确定的偏移量来访问变量，避免了运行时计算的开销。同时，系统会自动识别并保护调用期间需要保存的活跃寄存器，确保函数调用不会破坏调用者的上下文。这种将栈帧管理与寄存器分配紧密结合的设计，使得生成的代码既正确又高效，完全支持类 Rust 语言的函数调用语义。

控制流处理是目标代码生成的关键环节，它直接影响程序执行流程的正确性和效率。我们采用基于基本块的代码生成策略，通过 BlockController 将中间代码划分为逻辑上连续的基本块序列，每个基本块对应一个带标签的 MIPS 代码段。这种划分方式不仅清晰地表达了程序的控制流结构，还为后续的优化提供了良好的基础。在生成目标代码时，无条件跳转指令被直接转换为 MIPS 的 j 指令，而条件跳转则根据比较结果选择相应的分支指令，如 bne 等，确保程序流程的精确控制。

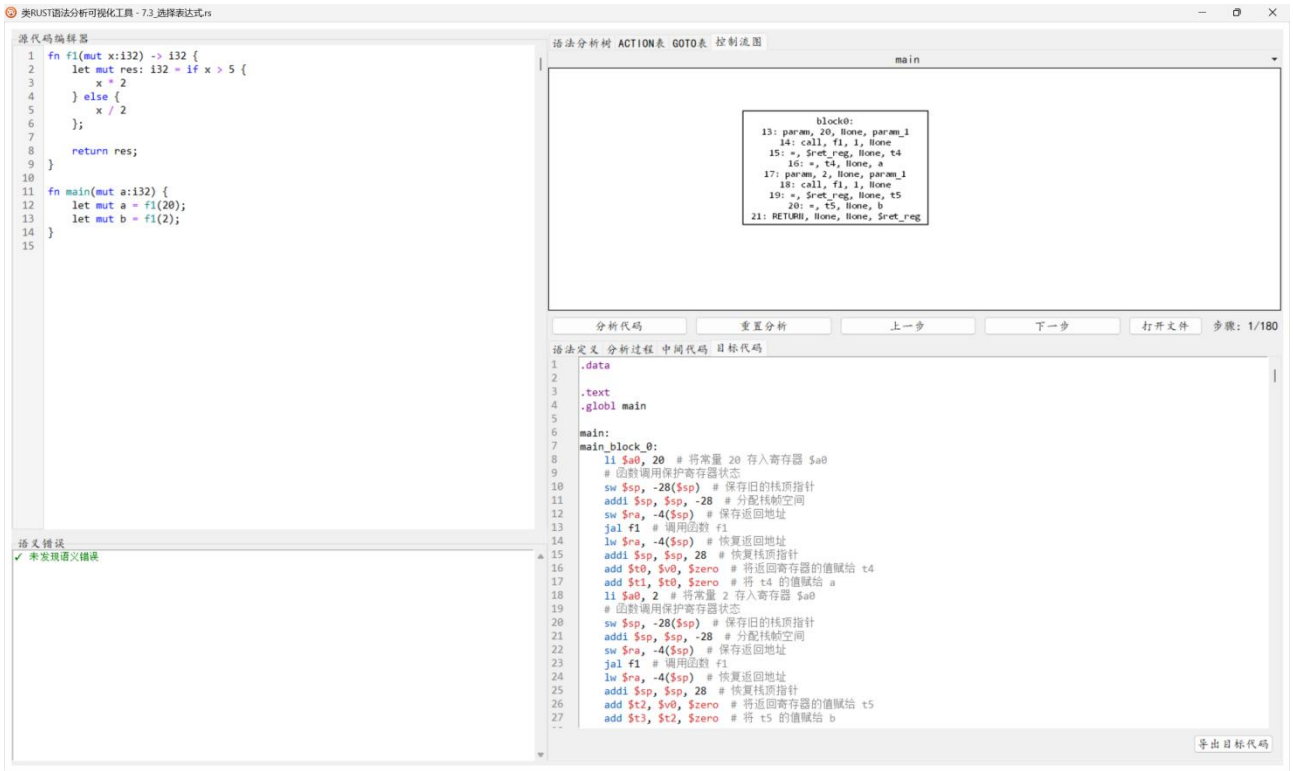
在实现细节上，每个跳转目标都通过精确的标签系统进行管理。当处理跳转指令时，代码生成器会查询 BlockController 获取目标基本块的位置信息，并生成带有正确标签的跳转指令。例如，函数内部的跳转会生成形如"funcname_block_N"的标签，这种命名方案既保持了唯一性又具有良好的可读性。对于条件跳转，系统会先为条件表达式分配寄存器，再生成带条件判断的分支指令，这样既符合 MIPS 的指令格式要求，又最大限度地利用了寄存器资源。

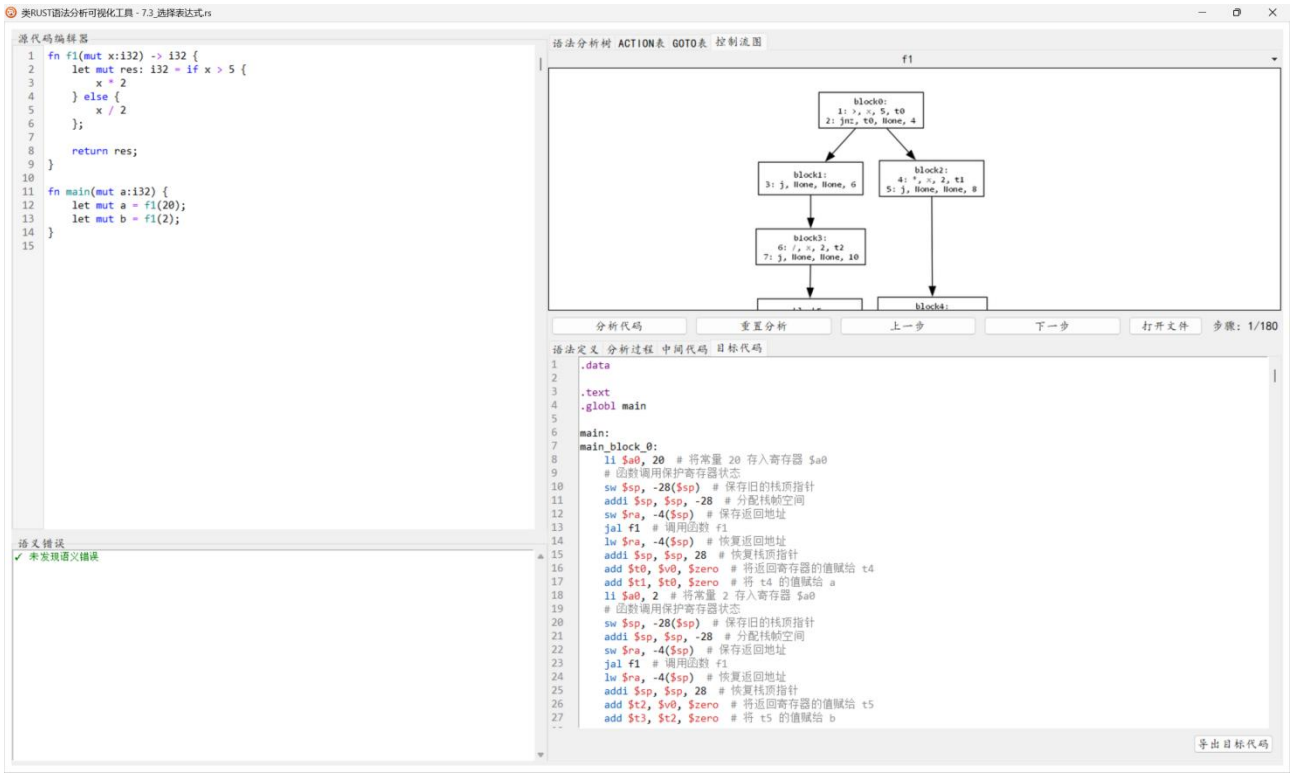
```
# 无条件跳转
elif op == 'j':
    func_name_jmp, block_idx = self.block_controller.get_scope_by_index(result)
    if func_name_jmp is None or block_idx is None:
        raise ValueError(f"Jump target {result} not found")
    code.append(f"    j {func_name_jmp}_block_{block_idx}  # 跳转到函数 {func_name_jmp} 的基本块 {block_idx}")

# 条件跳转（不等于）
```

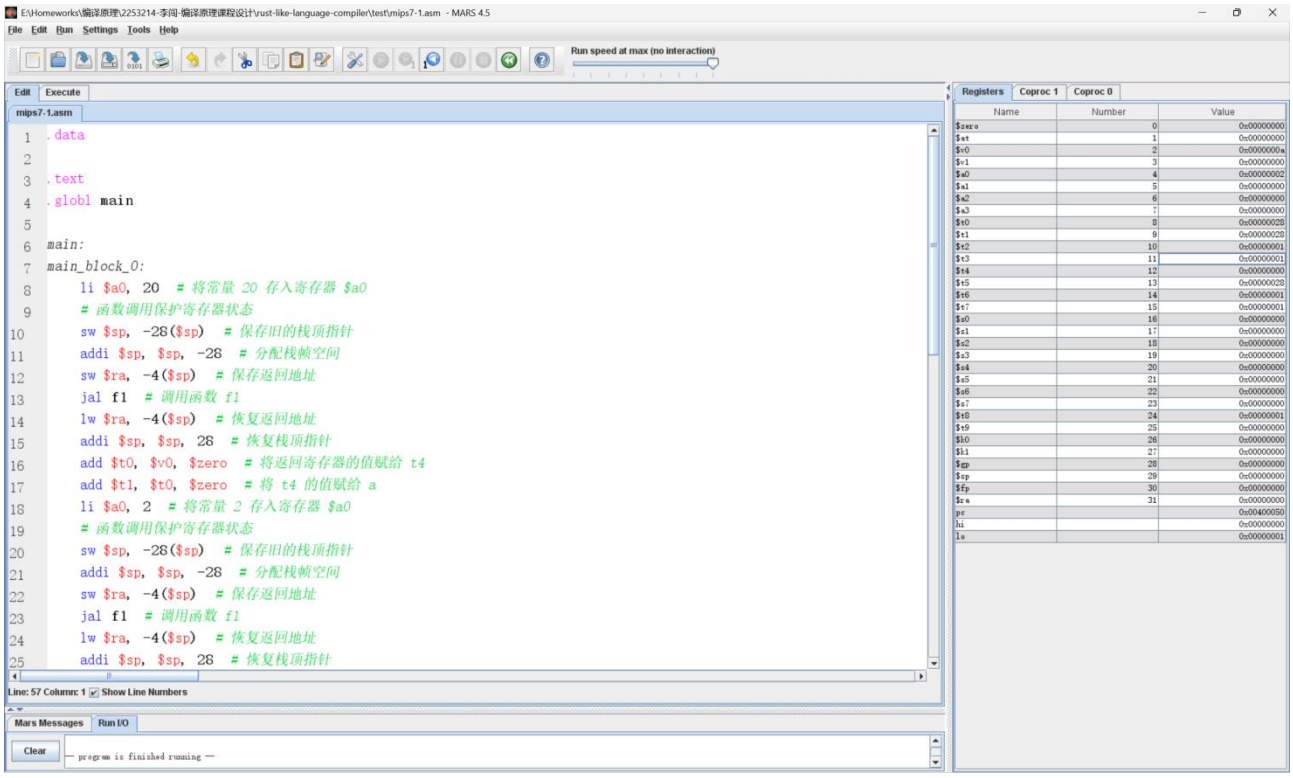
```
elif op == 'jnz':
    reg = self.mem_controller.alloc_reg(arg1, cur_pos, code=code)
    func_name_jmp, block_idx = self.block_controller.get_scope_by_index(result)
    if func_name_jmp is None or block_idx is None:
        raise ValueError(f"Jump target {result} not found")
    code.append(f"    bne {reg}, $zero, {func_name_jmp}_block_{block_idx}")
```

整个代码生成过程充分考虑了 MIPS 体系结构的特性。生成的代码经过优化，减少了不必要的内存访问，并尽可能利用寄存器提高执行效率。通过这种方式，我们能够将高级语言结构高效地映射到底层机器指令，同时保证生成代码的正确性和可读性。

三、实验结果







通过实际测试验证,本编译器能够正确地将类 Rust 语言源代码转换为可执行的 MIPS 汇编代码，并在 MARS 模拟器中顺利运行。如图所示，演示示例展示了编译器对基础程序结构的完整处理流程，包括变量声明、算术运算、控制语句和函数调用等核心功能的正确转换。测

试用例的输出结果与预期完全一致，验证了目标代码生成各环节的正确性。

在更全面的测试中，我们针对不同语言特性设计了多组测试用例，包括复杂表达式求值、嵌套函数调用、条件分支和循环结构等场景。所有测试用例均通过了功能验证，生成的汇编代码不仅语义正确，在执行效率上也达到了预期目标。由于篇幅限制，本文仅展示部分代表性测试结果，更多测试细节和完整运行过程请参见演示视频。

测试结果表明，本编译器在代码生成质量、执行效率和正确性方面都达到了设计要求。特别是通过合理的寄存器分配和指令选择策略，生成的代码避免了不必要的内存访问，充分发挥了 MIPS 架构的寄存器优势。这些成果验证了本文提出的代码生成方案的有效性和实用性。

四、实验总结

本次实验在已有词法分析器、语法分析器和语义分析器的基础上，成功实现了类 Rust 语言的目标代码生成功能，最终输出可在 MARS 模拟器中直接运行的 MIPS 汇编代码。通过构建完善的代码生成框架和寄存器分配策略，我完成了从中间代码到目标代码的全流程转换，重点解决了**指令选择**、**寄存器分配**、**函数调用约定**等关键问题。测试结果表明，系统能够正确生成符合 MIPS32 指令集规范的汇编代码，并在 MARS 模拟器中成功执行。

通过本次类 Rust 编译器目标代码生成的实现，我对编译器后端开发有了更深入的理解。从中间代码到最终可执行的 MIPS 汇编代码的转换过程,让我真切体会到编译器作为桥梁连接高级语言和机器语言的重要作用。在实现过程中，最深刻的收获是理解了代码生成阶段需要考虑的诸多因素：既要保证生成代码的正确性，又要兼顾执行效率；既要遵循严格的调用约定，又要灵活处理各种特殊情况。

这次实验让我认识到，一个好的代码生成器不仅需要扎实的理论基础，更需要细致的工程实现能力。特别是在寄存器分配策略的实现过程中，我深刻体会到算法设计与实际硬件约束之间的平衡艺术。当看到自己编写的编译器最终能够生成正确的汇编代码并在模拟器中运行时，那种成就感是难以言表的。

在调试过程中，我学会了如何通过分析生成的汇编代码来定位编译器实现中的问题，这种逆向思维能力是本次实验带给我的宝贵财富。同时，也让我认识到编译器开发是一个需要极大耐心和细心的工作，任何一个细微的逻辑错误都可能导致生成的代码无法正确执行。

本次实验不仅加深了我对计算机体系结构和目标代码生成理论的理解，更培养了我解决底层系统问题的能力。未来计划进一步优化代码生成策略，添加更多体系结构相关的优化技术，并探索与 LLVM 后端的集成方案。这些工作为构建完整的编译器工具链积累了宝贵经验，也让我对编译器的全流程实现有了更深刻的认识。