TONGJI UNIVERSITY

类Rust编译器设计报告

课程名称	编译原理
学生	2254287　侯俊皓
授课教师	丁志军
日　　期	2025.08

1. 实验概述

1.1 实验目的

本次实验的核心目标是在完成词法分析与语法分析的基础上，设计并实现一个功能完善的语义分析器、中间代码生成、目标代码生成编译器，从而构建一个完整的编译器前后端web应用。

1.2 主要任务

本次实验的主要任务可以分解为以下几个部分，涵盖了从基础功能实现到高级特性探索的完整链条。

1. 基本功能

语义分析器设计与实现： 在已有的抽象语法树（AST）基础上，遍历 AST，实现一个功能完备的语义检查器。该检查器必须能够处理：

变量与函数的作用域、声明与使用问题（如未声明、重复定义）。

严格的类型匹配检查，包括算术运算、赋值、函数调用及返回等场景。

中间代码生成，将通过语义检查的 AST，翻译成平台无关的四元式中间代码。要求能够正确处理顺序、选择（if-else）和循环（while, loop）等基本程序结构。

静态错误诊断与报告： 系统必须能够捕获所有在编译前端阶段可发现的静态错误（包括词法、语法、语义错误），并向用户提供清晰、准确的错误信息。

在此前的编译器基础上，更改词法语法分析逻辑，将两次扫描程序转换为一次扫描，具体操作是将词法分析作为语法分析子程序调用。随后添加最终代码生成功能。

```
# 一遍扫描：语法分析驱动词法分析
parser = Parser(code)
ast = parser.parse_program()
```

目标代码（汇编）

```
section .data
a dd 0
program_1_1 dd 0
t0 dd 0
section .text
global _start
_start:
program_1_1:
    mov eax, [a]
    cmp eax, [0]
    setg al
    movzx eax, al
    mov [t0], eax
    mov eax, [t0]
    cmp eax, 0
    je L0
    ret
    ret
    ret
    mov eax, 1
    int 0x80
```

2. 扩展与特色功能

实现类Rust 核心语义：

类型推导： 支持 let x = ...形式的变量声明，并根据初始化表达式自动推断其类型。 借用检查： 模拟Rust 的所有权与借用规则，对不可变引用（&T）和可变引用（&mut T）的创建与共存进行静态检查，对违反规则的代码进行报错。

构建Web 可视化平台：

设计并实现一个基于 Web 的前端用户界面，提供代码输入、分析触发和结果展示功能。利用D3.js 库，将生成的AST 动态渲染成一个可交互（可折叠/展开、可高亮）的树状图，直观展示程序的语法结构。将词法分析、中间代码和错误信息等结果，以友好、易读的方式同步展示在界面上。

1.3 需求分析

源代码输入：用户通过前端界面提供的 Web 文本编辑器，直接输入或粘贴一段符合我们定义的类 Rust 语言规范的源代码。程序应能实时响应用户的编辑操作。

程序输入范例如下：

```
    // 一个包含变量声明、条件判断和函数调用的示例 fn program_1_1() -> i32
{ let mut a: i32 = 1;
if a > 0 {
    a = a + 1;
} else {
    a = 0;
}
return a;
}
```

本项目的语法分析器采用递归下降法实现，文法规则内嵌于 parser.py 的各个 parse_*函数中，无需从外部文件读取。我们为类 Rust 语言设计的核心文法规则（BNF 范式简化表示）如下：

```
Program         ::= { Function }*
Function        ::= 'fn' Identifier '(' [ Params ] ')' [ '->' Type ] Block
Block           ::= '{' { Statement }* [ Expression ] '}'
Statement       ::= LetStmt | AssignStmt | ReturnStmt | IfStmt | LoopStmt | ExprStmt | ';'
LetStmt         ::= 'let' [ 'mut' ] Identifier [ ':' Type ] [ '=' Expression ] ';'
Expression      ::= ... (包含算术运算、比较、函数调用、引用、解引用等)
Type            ::= 'i32' | '&' Type | '&mut' Type | '(' [ Type { ',' Type }* ] ')'
```

当用户点击“分析”按钮后，前端界面应在右侧结果区同步展示以下内容：

词法分析结果： 将源代码分解成的 Token 流，并用不同颜色高亮不同类型的 Token。
语法分析可视化：
一个静态的、基于HTML 嵌套 div 的 AST 结构文本化展示。
一个动态的、由 D3.js 生成的、可交互的 SVG 语法树图。
四元式中间代码： 逐行展示生成的四元式指令序列。
目标代码: 即语义相同的汇编代码
错误/警告信息：
若编译过程无误，此区域为空。
若存在任何错误 (Error)（如语法错误、类型不匹配、借用冲突），则在此区域用红色字体显示详细的错误信息，并终止中间代码的生成。
若存在警告 (Warning)（如未来可能引入的“未使用变量”等），则用黄色字体显示，但不影响编译流程。

2. 使用说明

本章将详细介绍如何配置运行环境、安装项目依赖、启动应用程序，并对界面的核心功能进行说明，以确保用户能够顺利地部署并使用我们的类 Rust 语言可视化分析平台。

2.1 环境配置说明

为了保证项目能够正确运行，需要在本地计算机上准备好以下软件环境。本项目后端基于 Python，前端依赖于 Node.js 生态系统进行包管理。

1. 基础软件环境**:**
Python: 要求版本: Python 3.8 或更高版本。
验证方法: 在终端（命令行）中输入 python --version 或 python3 --version，检查输出的版本号是否满足要求。
安装建议: 推荐从 Python 官方网站 (python.org) 下载并安装。在安装过程中，请务必勾选 "Add Python to PATH" 选项，以便在任何路径下都能直接调用 python 和pip 命令。

pip (Python 包管理器**):**
pip 通常会随Python 一同安装。
验证方法: 在终端中输入 pip --version 或 pip3 --version。

Node.js 与 **npm:**
要求版本: Node.js 16.x 或更高版本 (推荐使用 LTS 长期支持版)。npm (Node Package Manager) 会随 Node.js 一同安装。
验证方法: 在终端中分别输入 node -v 和 npm -v，检查版本号。
安装建议: 访问 Node.js 官方网站 (nodejs.org) 下载并安装。

Git (版本控制系统**):**
要求: 用于从代码仓库克隆项目。
验证方法: 在终端中输入 git --version。

2.1.1 安装依赖

在成功配置好基础环境并激活虚拟环境后，请按照以下步骤安装项目所需的所有后端和前端依赖包。

1. 安装**Python** 后端依赖**:**
项目的所有 Python 库依赖都记录在 requirements.txt 文件中。确
保已处于项目根目录，并且虚拟环境已激活。
在终端中运行以下命令：

```
pip install -r requirements.txt
```

此命令会自动读取文件内容，并下载安装 Flask 等所有必需的 Python 库。

2. 安装**JavaScript** 前端依赖**:**
本项目虽然没有使用大型前端框架，但为了规范化开发，我们可能使用一些开发工具（例如，如果未来引入代码格式化工具 Prettier 或打包工具）。

说明： 本项目前端界面设计力求简洁，未使用复杂的构建工具链，因此无需安装额外的前端依赖。所有必需的库（如 D3.js）均通过 CDN 链接直接在 frontend.html 中引入，以简化部署流程。

2.2 总体设计

2.2.1 初始界面

当应用程序成功启动后，用户在浏览器中访问指定的地址（默认为 http://127.0.0.1:5000），将会看到如下图所示的初始界面。整个界面采用清晰、专业的多栏布局，主要分为代码编辑区、功能控制区和结果展示区三个部分。

左侧：代码编辑区

功能：这是用户输入和编辑类Rust源代码的核心区域。我们提供了一个宽敞的文本域，支持多行文本编辑、复制、粘贴等标准操作。

预置代码： 为了方便用户快速体验，界面在初次加载时会自动载入一段预置的、功能相对完整的示例代码。这段代码展示了变量声明、条件分支、函数调用等多种语言特性。

自动分析： 界面在加载完成后，会自动对预置代码执行一次分析，让用户可以立刻看到一个完整的分析结果示例，直观了解平台的功能。

上方：功能控制区

“分析”按钮： 这是平台最核心的交互按钮。每当用户修改完代码后，点击此按钮，前端会立即将编辑器中的最新代码发送至后端服务器进行完整的编译前端分析。分析完成后，右侧的所有结果展示区将随之更新。

“复制代码”按钮： 一个便捷的辅助功能，点击后会将当前编辑器内的所有代码复制到系统剪贴板，方便用户保存或分享。

右侧：结果展示区

布局： 此区域从上到下垂直排列了多个信息面板，分别对应编译流程的不同阶段。每个面板都有清晰的标题，如“词法分析结果”、“语法分析可视化”等。

内容： 初次加载时，这里会展示对预置代码的分析结果，包括：

词法分析结果： 以彩色文本形式展示的 Token 流。

语法分析可视化： 简易的 AST 文本结构。

语法树结构图： 可交互的 D3.js 动态 SVG 树图。

四元式中间代码： 生成的中间代码指令列表。

目标代码：生成的目标汇编代码

错误信息区：位于界面最下方，专门用于显示编译过程中可能出现的任何错误或警告。初始状态下，如果预置代码无误，此区域将为空。

2.2.2 基础功能展示

我们的平台旨在将复杂的编译过程透明化。用户通过简单的“编辑-点击分析”操作，即可触发一次完整的分析并观察其结果。下面我们以一个不包含错误的基础代码为例，介绍各结果面板所展示的内容。

输入示例代码：

```
    fn main()
{ let x: i32 =
10; if x > 5 {
        return 1;
} else {
        return 0;
}
}
```

1. 词法分析结果

该面板会显示经过tokenizer.py 处理后生成的 Token 序列。

我们通过为不同类型的 Token（如关键字、标识符、数字、操作符）应用不同的 CSS 类，实现了语法高亮。例如，fn 和 let 会显示为蓝色粗体，标识符 main 和 x 会显示为绿色，数字 10 和 5 会显示为红色。

这种可视化的 Token 流，是理解编译器如何“阅读”源代码的第一步。

2. 语法树结构图

这是最具特色的展示区。它将 parser.py 生成的层级式 AST 数据，通过 D3.js 库渲染成一个动态的、可交互的 SVG 树状图。

结构： 树的根节点是 Program，其下逐级展开为 Function、Block、If、BinaryOp 等节点，完整地再现了代码的语法结构。

交互性：

折叠/展开： 用户可以用鼠标点击任何一个非叶子节点，其子树会平滑地收起或展开。这对于分析复杂的、嵌套层次很深的代码尤为有用。

高亮： 当鼠标悬停在某个节点上时，该节点及其所有子孙节点都会被高亮显示，帮助用户聚焦于特定的语法子结构。

3. 四元式中间代码

该面板展示了 irgen.py 将 AST 翻译后的结果。

每一行都是一条格式化的四元式指令，例如 (>, x, 5, t0) 或 (IFZ, t0, , L1)。

通过观察这个指令序列，用户可以清晰地看到高级的 if-else 结构是如何被分解为基本的条件判断和跳转指令，从而理解程序在更低层次上的控制流。

通过这一整套的可视化设计，用户不再是将编译器视为一个“黑盒”，而是能够清晰、直观地追踪源代码在编译前端的每一步“变形”过程，极大地增强了对编译原理的学习和理解。

4. 目标代码

展示了根据四元式翻译为汇编语言的结果。

2.3 错误处理

2.3.1 函数输入输出错误

1. 函数的输入输出类型需保持一致。

类Rust词法/语法分析可视化

```
fn program_1_5__2() -> i32 {
   return ;
}
```

分析 复制代码

词法分析结果

```
fn program_1_5__2 ( ) -> i32 { return ; }
```

语法分析可视化

```
函数program_1_5__2声明返回类型为i32，但return无值
```

2.3.2 变量声明错误

1. 后续无语句，无法推断 **b** 的类型

类Rust词法/语法分析可视化

```
fn program_2_1__2() {
   let mut b;
}
```

分析 复制代码

词法分析结果

```
fn program_2_1__2 ( ) { let mut b ; }
```

语法分析可视化

```
变量b类型无法推导
```

2. 变量未声明

2.3.3 函数调用错误

1. 实参数量与形参数量不一致

2. 实参类型与形参类型不一致

类Rust词法/语法分析可视化

```
fn program_3_3__4__a(mut a:i32) {
}

fn program_3_3__4__b() {
   program_3_3__4__a(program_3_3__4__a);
}
```

分析 复制代码

词法分析结果

```
fn program_3_3__4__a ( mut a : i32 ) { } fn program_3_3__4__b ( ) { program_3_3__4__a ( program_3_3__4__a ) ; }
```

语法分析可视化

变量program_3_3__4__a未声明

3. 无返回值函数不能作为右值

类Rust词法/语法分析可视化

```
fn program_3_3__5__a() {
}

fn program_3_3__5__b() {
   let mut a=program_3_3__5__a();
}
```

分析 复制代码

词法分析结果

```
fn program_3_3__5__a ( ) { } fn program_3_3__5__b ( ) { let mut a = program_3_3__5__a ( ) ; }
```

语法分析可视化

变量a类型无法推导

2.3.4 循环结构错误

1. **break;** 必须出现在循环体内。

2.3.5 不可变变量赋值错误

1. 不可变变量不可二次赋值。

2.3.6 引用错误

1. 不允许对非引用类型进行解引用

类Rust词法/语法分析可视化

```
fn program_6_2__3() {
  let mut a:i32=1;
  let mut b=*a;
}
```

分析 复制代码

词法分析结果

```
fn program_6_2__3 ( ) { let mut a : i32 = 1 ; let mut b = * a ; }
```

语法分析可视化

解引用操作符 * 只能用于引用类型

2. 可变引用不能和其他的引用共存

类Rust词法/语法分析可视化

```
fn program_6_2__4() {
  let mut a:i32=1;
  let b=&a;
  let mut c=&mut a;
}
```

分析 复制代码

词法分析结果

```
fn program_6_2__4 ( ) { let mut a : i32 = 1 ; let b = & a ; let mut c = &mut a ; }
```

语法分析可视化

不能创建对a的可变引用，因为已存在其他引用

3. 仅支持从可变变量创建可变引用

2.3.7 数组与元组错误

1. 变量赋值的类型与声明的类型不一致

2. 初始化时的元素数量与数组长度不一致

类Rust词法/语法分析可视化

```
fn program_8_1__3(mut a:i32) {
  let mut a:[i32;2];
  a=[1,2,3];
}
```

分析 复制代码

词法分析结果

```
fn program_8_1__3 ( mut a : i32 ) { let mut a : [ i32 ; 2 ] ; a = [ 1 , 2 , 3 ] ; }
```

语法分析可视化

```
变量a类型为{'array': 'i32', 'size': 2}，赋值类型为{'array': 'i32', 'size': 3}，类型不一致
```

3. 初始化时元素的类型与数组元素类型不一致

类Rust词法/语法分析可视化

```
fn program_8_1__4() {
  let mut a=[[i32;1];1];
  a=[1];
}
```

分析 复制代码

词法分析结果

```
fn program_8_1__4 ( ) { let mut a = [ [ i32 ; 1 ] ; 1 ] ; a = [ 1 ] ; }
```

语法分析可视化

```
未知表达式: Token(type='LET', value='let', pos=24)
```

4. 数组的索引的类型必须是整数类型

类Rust词法/语法分析可视化

```
fn program_8_2__2(mut a:i32) {
  let mut a=[1,2,3];
  let mut b=a[a];
}
```

分析 复制代码

词法分析结果

```
fn program_8_2__2 ( mut a : i32 ) { let mut a = [ 1 , 2 , 3 ] ; let mut b = a [ a ] ; }
```

语法分析可视化

```
数组或元组索引类型必须为i32
```

5. 数组索引越界

类Rust词法/语法分析可视化

```
fn program_8_2__3() {
  let mut a=[1,2,3];
  let mut b=a[3];
}
```

分析 复制代码

词法分析结果

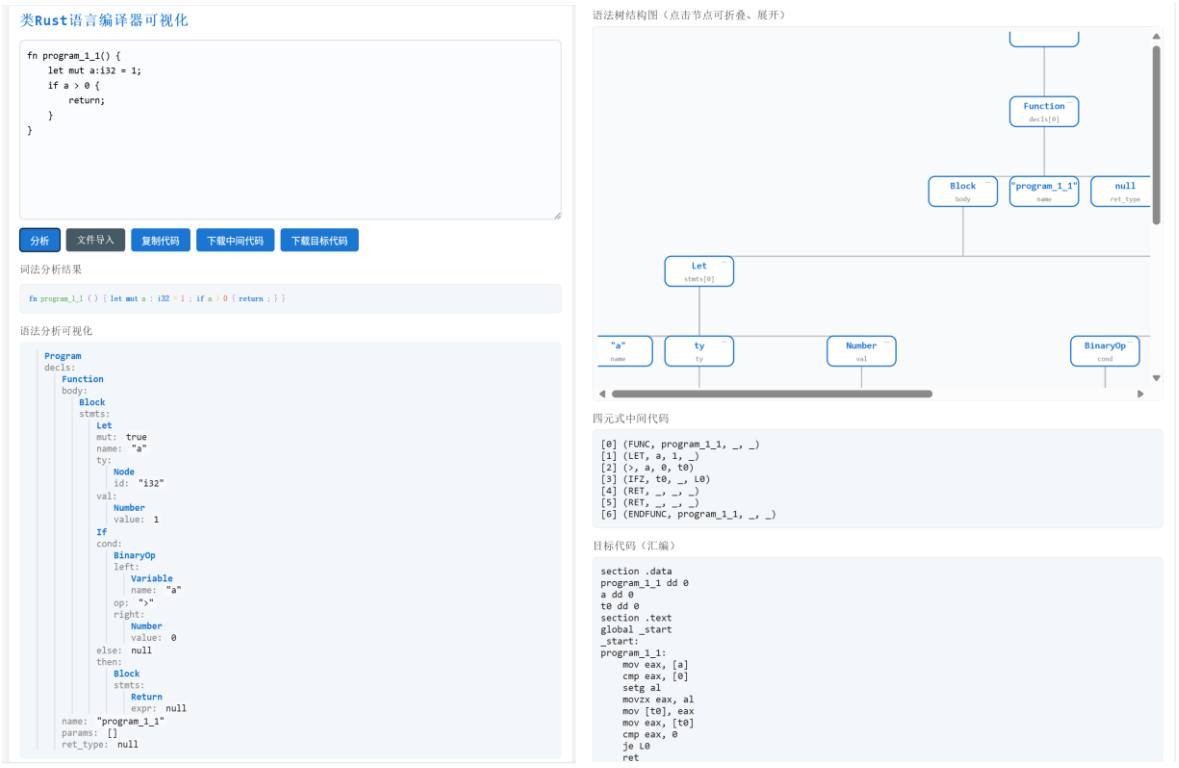
```
fn program_8_2__3 ( ) { let mut a = [ 1 , 2 , 3 ] ; let mut b = a [ 3 ] ; }
```

语法分析可视化

```
数组索引3超出范围[0,2]
```

6. 不可变数组的元素也不可变，不能作为左值

正确样例：



3. 详细设计

3.1 语义分析与中间代码生成

3.1.1 核心数据结构设计的深度解析

为支撑复杂的语义检查和IR 生成，我们设计了一套相互关联、职责分明的数据结构。

1. 符号表 **(Symbol Table)** 与作用域的精细化管理

符号表是进行上下文相关分析（如变量查找、类型检查）的核心。我们设计的分级作用域栈 (scopes) 是一种动态数据结构，完美契合了类Rust 语言的词法作用域（Lexical Scoping）规则。

结构与行为：scopes 是一个list，其每个元素都是一个dict，该字典的键是变量名（str），值是对应的 Symbol 对象。

当分析器进入一个新的作用域时（例如，开始分析一个函数体或一个由{}包裹的代码块），会执行 scopes.append({})操作。这相当于在栈顶压入一个新的、空的符号表，用于存放该作用域内定义的局部变量。

退出作用域： 当分析完一个作用域时，执行 scopes.pop()。这会销毁栈顶的符号表，其中包含的所有局部变量也随之“消亡”，其生命周期结束。这种“后进先出”的机制确保了变量的可见性严格遵守其定义所在的块结构。

Symbol 类——变量信息的全息记录**:**

Symbol 对象是变量所有静态属性的载体，其设计细节直接决定了语义检查的深度。

```
   class Symbol:
def __init__(self, name, typ=None, mut=False, inited=False):
    self.name = name        # 变量的唯一标识符
    self.typ = typ          # 变量的类型。它可以是一个简单的字符串如'i32'，
                            # 也可以是一个嵌套的字典来表示复杂类型，
                            # 如 {'array': 'i32', 'size': 5} 或 {'ref': 'mut', 'to': 'i32'}。
    self.mut = mut          # 标记该绑定是否为可变绑定 (let mut)
    self.inited = inited # 跟踪变量是否已被赋值。对未初始化变量的使用会被静态检查捕获。
    self.refs = []          # [核心] 借用跟踪列表。这是实现借用检查的基石。
                            # 它是一个列表，存储了当前所有有效引用的类型：
                            # 'imm' 代表一个不可变引用 (&T)
                            # 'mut' 代表一个可变引用 (&mut T)
```

2. 四元式 **(Quadruple)**

我们选择四元式作为IR，因为它结构统一、语义明确，非常适合进行后续的流图分析和优化。

统一结构**:** {'op': str, 'arg1': str, 'arg2': str, 'res': str}

op: 操作码，如 +, -, *, /, =, GOTO, IFZ (If Zero Jump) 等。

arg1, arg2: 操作数。可以是变量名、常量，或是临时变量名。

res: 结果的存放位置。通常是变量名或临时变量名。对于跳转指令，它可以是目标标

签。

函数调用 **z = sum(a, b):**

('PARAM', 'a', '', '0') # 传递第 0 个参数
('PARAM', 'b', '', '1') # 传递第 1 个参数
('CALL', 'sum', '2', 't1') # 调用 sum 函数，有 2 个参数，返回值存入 t1 ('=',
't1', '', 'z')

3. 辅助生成器：临时变量与标签

为了保证 IR 生成过程中产生的临时实体不与用户定义的变量冲突，并且每个跳转都有唯一的目标，我们实现了两个简单的生成器。

new_temp(): 内部维护一个静态计数器，每次调用返回一个新的临时变量名，如 t0, t1, t2...。

new_label(): 同样基于计数器，每次调用返回一个新的标签名，如 L0, L1, L2...。

3.1.2 语义检查关键流程的深度剖析

Checker 类通过对 AST 进行一次深度优先遍历，完成了所有静态语义的检查。

1. 类型检查与推导 (check_expr & types_equal)

check_expr(node):

递归基: 当节点是叶子节点时（如Number、Variable），直接返回其类型。对于Variable，需要调用resolve_var 在符号表中查找。

递归步骤**:** 当节点是内部节点时（如BinaryOp），它会：递归调用 check_expr 获取其左右子节点的类型，如 left_type 和 right_type。调用 types_equal(left_type, right_type)检查类型是否匹配。如果不匹配，则生成一条详细的类型错误信息。根据操作符的语义，返回该表达式的结果类型（例如，对于算术运算符，结果类型与操作数类型相同；对于比较运算符，结果类型为布尔型）。

类型推导**:** 在 check_let 中，如果 let 语句没有显式类型注解但有初始化值（如 let x = 1 + 2;），check_expr 会被调用来计算初始化表达式的类型，这个类型随后被赋给新创建的 Symbol 的 typ 字段，从而完成了类型推导。

2. 作用域与变量生命周期

变量声明 **(check_let):**从 AST 节点中提取变量名。在当前作用域（scopes[-1]）中检查该变量名是否已存在。如果存在，报告“变量重定义”错误。创建一个新的 Symbol 实例，并根据let 语句中的 mut 关键字和类型注解来设置其属性。将新的 Symbol 添加到当前作用域。

变量查找 **(resolve_var):**从 scopes 列表的末尾（当前作用域）开始向前遍历。在每个作用域（字典）中查找变量名。一旦找到，立即返回对应的 Symbol 对象。如果遍历完整个 scopes栈仍未找到，返回None，调用者会据此报告“变量未声明”的错误。

3. 借用检查 **(Borrow Checking) -** 详细机制

借用检查是本项目的技术亮点，其实现紧密耦合在 check_expr 对 AddrOf (&) 和

AddrOfMut (&mut) 节点的处理中。

当分析 **&x (**创建不可变引用**)** 时:resolve_var('x') 找到 x 的 Symbol 对象，记为 sym_x。检查 sym_x.refs 列表。遍历此列表，如果发现任何一个元素是'mut'，意味着此时已存在一个活跃的可变引用。

规则应用**:** 根据 Rust 的规则“当存在可变引用时，不能再创建任何其他引用”，此操作非法。立即生成错误：“无法创建对 x 的不可变引用，因为已存在一个可变引用”。若检查通过，执行 sym_x.refs.append('imm')，记录下这个新的不可变引用。

当分析 **&mut x (**创建可变引用**)** 时**:**resolve_var('x') 找到 sym_x。前置检查: 首先检查 sym_x.mut 是否为 True。如果为 False，说明 x 是一个不可变变量，无法创建其可变引用。生成错误：“无法从不可变变量 x 创建可变引用”。检查 sym_x.refs 列表的长度。

规则应用**:** 如果 len(sym_x.refs) > 0，意味着此时已存在其他任何类型的引用（无论是可变还是不可变）。根据Rust的规则“可变引用是独占的”，此操作非法。生成错误：“无法创建对 x 的可变引用，因为已存在其他引用”。若检查通过，执行 sym_x.refs.append('mut')，记录下这个独占的可变引用。

3.1.3 中间代码生成关键流程深度剖析

IRGen 类负责将已经过完整语义验证的 AST，系统性地翻译为线性的四元式序列。

1. 表达式生成 **(gen_expr)** 的递归逻辑

gen_expr 的实现是一个经典的后序遍历应用。它总是先处理子节点，再处理根节点。

例：**a + b * c:**

gen_expr 被调用处理 + 节点。

它首先递归调用 gen_expr 处理左子节点 a。a 是叶子节点，直接返回其名字'a'。然后递归调用 gen_expr 处理右子节点*。

a. gen_expr 在*节点上，再次递归调用处理其左子节点 b，返回'b'。

b. 再次递归调用处理其右子节点 c，返回'c'。

c. 现在*节点拥有了两个操作数'b'和'c'。它调用new_temp()得到't0'，然后emit('*', 'b', 'c', 't0')。最后，它返回结果的存放位置't0'。

现在+节点拥有了两个操作数'a'和't0'。它调用new_temp()得到't1'，然后emit('+', 'a', 't0', 't1')。最后返回't1'。

整个过程生成的 IR 序列为:

(*, b, c, t0)

(+, a, t0, t1)

2. 控制流语句生成的详细步骤

if-else 语句 **(gen_if_stmt):**

准备: gen_expr(cond) -> t_cond。L_else = new_label()，L_end = new_label()。

条件跳转: emit('IFZ', t_cond, '', L_else)。IFZ 意为“如果t_cond 为零（假）则跳转”。

then 块: gen_block(then_block)。此块的IR 被顺序放置。

无条件跳转: emit('GOTO', L_end, '', '')。这是至关重要的一步，它防止了在then 块执行完毕后，程序“掉入”else 块。

else 标签: emit('LABEL', L_else, '', '')。为IFZ 指令提供了跳转目标。
else 块: gen_block(else_block)。
结束标签: emit('LABEL', L_end, '', '')。为GOTO 指令提供跳转目标，并标记整个语句的结束。
while 循环 (gen_while_stmt)
准备: L_start = new_label()， L_end = new_label()。
循环开始标签: emit('LABEL', L_start, '', '')。这是循环的入口，每次循环都会跳回这里。
条件判断: gen_expr(cond) -> t_cond。
条件退出: emit('IFZ', t_cond, '', L_end)。如果条件为假，则跳出循环，执行循环之后的代码。
循环体: gen_block(body_block)。
返回循环头: emit('GOTO', L_start, '', '')。循环体执行完毕，无条件跳回 L_start 进行下一次条件判断。
循环结束标签: emit('LABEL', L_end, '', '')。为IFZ 指令提供退出循环的目标点。

3.2 前端设计与可视化实现

前端是用户与编译器进行交互的直接窗口，其设计的优劣直接影响到用户对编译过程的理解效率和使用体验。为此，我们投入了大量精力来设计一个信息丰富、交互性强且美观友好的 Web 界面。前端的核心任务是将后端分析模块产出的抽象数据转化为直观、易懂的图形化表示。

3.2.1 界面总体布局与设计理念

我们的前端界面采用了一个经典的多面板布局，旨在将编译流程的各个阶段清晰地并列展示，方便用户进行对照分析。

布局结构：

左侧面板（代码编辑区）：

提供一个大尺寸的<textarea>作为代码编辑器，支持多行输入和垂直缩放。

我们没有使用现成的代码编辑器插件（如Monaco Editor 或CodeMirror），而是选择手动实现语法高亮功能，以更紧密地结合我们自己的词法分析结果。

右侧面板（结果展示区）：

这是一个多功能的区域，通过多个独立的 容器来展示不同阶段的分析结果。

词法分析结果 **(#lex)**： 以流式布局展示所有词法单元，并根据 Token 类型赋予不同的 CSS 类，实现彩色高亮。

语法分析可视化 **(#ast)**： 以嵌套的 div 结构，直观地展示 AST 的层级关系，作为一种简易的树状图预览。

语法树结构图 **(#ast-tree)**： 这是可视化的核心区域，用于承载由 D3.js 动态生成的、可交互的 SVG 语法树。

四元式中间代码 **(#ir-code)**：

以等宽字体的<pre>标签，逐行展示生成的四元式列表，保持格式的整洁。

错误信息区 (#err)： 用于显示后端返回的任何语法或语义错误，采用醒目的红色字体。

设计理念：

信息密度与可读性平衡： 我们选用了 Consolas、JetBrains Mono 等高质量的等宽字体，并精心调整了字号、行高和颜色方案，确保在展示大量信息的同时，用户依然能轻松阅读。

响应式设计： 通过CSS 媒体查询 (@media)，对小屏幕设备（如手机）进行了布局优化，例如减小边距、调整字体大小，保证了在移动端的可用性。

用户引导： 界面元素（如标题、按钮）命名清晰，用户无需学习即可理解各部分的功能。

3.2.2 前后端交互逻辑

前端的交互核心由 JavaScript 驱动，主要围绕 runAnalysis()函数展开。

异步通信 **(fetch API):**

当用户点击“分析”按钮时，runAnalysis()函数被触发。

它首先从<textarea>获取用户输入的源代码字符串。

然后，使用现代的 fetch API 向后端的/analyze 端点发起一个 POST 请求。请求头中指定 Content-Type: application/json，请求体则是包含代码的 JSON 对象：{ "code": "..." }。

fetch 返回一个 Promise。我们使用.then()链式调用来处理异步响应：

第一个.then(resp => resp.json())将收到的 HTTP 响应体解析为 JSON 对象。

第二个.then(data => { ... })接收解析后的data 对象，并根据其内容进行后续的渲染操作。

数据驱动的动态渲染：

在第二个.then()回调中，我们首先检查 data.success 字段。

如果为true（分析成功）：

调用renderTokens(data.tokens)更新词法分析区。

调用 renderAST(data.ast)更新简易 AST 预览区。

调用 renderIR(data.ir)更新四元式区。

最后，也是最关键的，调用 renderAstTreeByD3(data.ast)来生成动态 SVG 语法树。

如果为**false**（分析失败）：

我们设计了优雅的降级处理。即使分析失败，后端仍可能返回部分结果（如 tokens）。我们会先渲染这些有效结果。

然后，将 data.error 中详细的错误信息展示在指定的错误区域。一个重要的设计是，我们将错误信息显示在原“语法分析可视化”区域，因为大多数错误发生于此阶段，这样布局更为直观。

错误处理：

使用.catch(e => { ... })来捕获网络错误或服务器内部错误，并在界面上向用户显示“分析失败”的提示，增强了应用的鲁棒性。

3.2.3 D3.js 动态语法树可视化——核心技术实现

这是前端工作的技术核心和最大亮点。我们没有使用现成的图表库，而是直接利用 D3.js 的底层能力，从零开始构建了一个高度定制化的、可交互的语法树。

1. 数据转换 **(astToTree** 函数**)**

挑战： 后端返回的 AST 是一个通用的 JSON（嵌套字典）结构，而 D3.js 的 tree 布局需要一个特定的层级数据格式，即每个节点都包含 name（用于显示）和 children（一个包含子节点的数组）属性。

解决方案： 我们编写了一个递归函数 astToTree。该函数接收一个 AST 节点，将其转换为 D3 所需的格式。它将 AST 节点的 type 字段作为新节点的 name。它会遍历原 AST 节点的所有其他属性。如果属性值是一个列表，它会递归地转换列表中的每个元素；如果属性值是其他对象或值，它也会递归地进行转换。所有转换后的子节点都被收集到 children 数组中。

2. D3 树布局与 **SVG** 坐标计算 **(d3.tree)**

自动化布局： 我们使用 d3.tree() 来创建一个树布局生成器。const treeLayout = d3.tree().nodeSize([nodeWidth, nodeHeight])我们只需将经过 astToTree 和 d3.hierarchy 转换后的根节点数据传递给 treeLayout()，D3 就会自动为树中的每一个节点计算出理想的 x 和 y 坐标，从而形成一个整齐、美观的树状结构。这极大地简化了布局的复杂性。

3. SVG 元素的数据绑定与渲染

D3.js 的精髓在于其“数据绑定”能力。我们使用它将计算好的节点和连线数据“绑定”到 SVG 图形元素上。

渲染连线 **(Links):**svg.selectAll('path').data(root.links()).join('path')我们将 root.links()（一个包含所有父子连接信息的数组）绑定到<path>元素上。对于每个<path>，我们使用一个路径生成器函数来动态计算其 d 属性（路径描述），从而画出从父节点到子节点的直角连接线。

渲染节点 **(Nodes):**svg.selectAll('g.node-group').data(root.descendants()).join('g') 我们将 root.descendants()（一个包含所有节点信息的数组）绑定到<g>（组合）元素上。每个<g>代表一个完整的节点。在每个<g>内部，我们再追加：一个<rect>作为节点的背景框。一个或多个<text>元素，用于显示节点的 name（类型）和 key（它在父节点中的属性名），并分别设置不同的样式。

4. 核心交互功能的实现

点击折叠/展开：

为每个节点<g>元素添加一个 click 事件监听器。D3 的层级数据结构中，可见的子节点存储在 node.children 属性中，而被折叠的子节点可以临时存放在一个自定义属性，如 node._children 中。

点击时的逻辑：

如果 node.children 存在（即节点是展开的），则将其内容移动到 node._children，并将 node.children 设为 null。如果 node._children 存在（即节点是折叠的），则将其内容移回 node.children，并将 node._children 设为 null。

关键步骤： 每次点击并修改数据后，必须重新调用 update()函数，该函数会清空并根据新的数据结构（某些节点的 children 属性已改变）重新计算布局和渲染整个 SVG 树。这就实现了动态折叠/展开的流畅动画效果。

鼠标悬停高亮：

为每个节点<g>元素添加 mouseover 和 mouseout 事件监听器。mouseover 时： 获取当前悬停节点及其所有后代节点（d_hovered.descendants()）。然后使用 D3 的选择器，选中所有这些对应的 SVG 节点，并改变它们的样式（如边框颜色、背景填充色）。mouseout 时： 同样获取节点及其后代，然后将样式恢复为默认状态。

我们还添加了 CSS 过渡（transition）效果，使得高亮和恢复的过程有一个平滑的动画，提升了视觉体验。

3.3 目标代码生成

在编译器的目标代码生成阶段，codegen.py 文件扮演着将中间代码（IR，通常是四元式序列）转化为底层汇编代码的关键角色。该阶段的核心是 CodeGen 类，它以 IR 为输入，通过分析和处理每个四元式，生成对应的 x86 汇编指令序列。首先，CodeGen 会遍历整个 IR，收集所有出现的变量、临时变量、标签和函数名等符号，并分别归类，为后续汇编代码的组织打好基础。这些符号会被用于数据段和代码段的声明，确保每个变量和临时量都有独立的内存空间。

在具体的代码生成过程中，CodeGen 会根据 IR 中每个操作符的语义，动态拼接汇编指令。例如，赋值语句会被翻译为寄存器值的传递和内存存储，算术运算则对应寄存器之间的加减乘除指令，比较操作映射为条件设置指令和零扩展等。更复杂的语法，如一元运算、

取地址与指针解引用、数组/元组访问等，也都通过适当的 x86 指令实现，从而支持多样的语言结构。跳转、条件分支和标签的处理，则确保了程序的流程控制和函数调用机制的正确实现。

此外，目标代码生成阶段不仅仅是简单的逐条翻译，还需关注符号管理、存储分配和指令顺序，以保证生成代码的正确性和可执行性。整个过程结束后，CodeGen 会把数据段和代码段拼接成完整的汇编代码文本，既包括变量空间声明，也包括主程序入口和所有逻辑实现。这个阶段的实现为编译器架构的后端提供了坚实基础，使得高级语言的抽象语法结构最终能够落地为可运行的底层机器指令。

3.4 一次扫描逻辑

在这个编译器的设计中，一次扫描意味着整个源代码只需要线性遍历一遍，即可完成从字符流到抽象语法树（AST）的构建。这里的核心思想是让语法分析过程动态地驱动词法分析。具体而言，语法分析器在解析每个语法规则时，并不会预先读取或缓存所有的词法单元，而是按需调用词法分析器的接口，每次只请求下一个token或者进行前瞻。这样，词法分析器就以子程序的形式嵌入到语法分析器内部，负责把原始字符流切分成语法分析所需的token序列。整个过程由语法分析主导，词法分析则“服务”于语法分析的需要。

这种实现方式带来了极强的灵活性和高效率。语法分析器在遇到需要判断或分支的地方，可以随时请求词法分析器进行前瞻（peek），甚至可以做标记和回溯（mark/reset），以支持更复杂的语法结构，比如尝试-回退式的递归下降解析。这种“增量”式的词法分析不仅减少了内存占用，也使得错误定位更加精确。当语法分析器遇到无法匹配的产生式或语法错误时，可以立刻报告，并准确指出出错的字符位置和token类型。这对于编译器的交互性和用户体验极为重要。

此外，把词法分析作为语法分析的子程序，还有助于编译器整体架构的模块化和可维护性。词法分析器专注于token的切分和分类，语法分析器则负责高层结构的识别和语法树的构建，两者通过明确的接口协同工作。最终，这种一次扫描的机制使得编译过程自然而高效地贯穿始终，无需多余的预处理或回退，大大提升了编译速度和响应能力。这种方法尤其适用于语法不太复杂、结构清晰的语言，是现代轻量级编译器实现的主流方案。

项目总结：

在此次课设中，我实现了一个基于递归下降法的编译器前端与后端一体化系统，涵盖了从源代码输入到最终目标代码生成的完整流程。用户通过前端页面 frontend.html 提交源代码，后端由 Flask 框架驱动，app.py 作为主入口，负责服务启动与请求分发。路由逻辑在 routes.py 中实现，确保前端与后端的数据交互顺畅，用户体验友好。

在编译流程中， parser.py 采用递归下降法进行语法分析，依据语言文法规则将记号序列构建为语法树。值得一提的是，项目采用“一次扫描”逻辑，即在语法分析阶段同步完成语义检查与符号表维护，避免多次遍历源代码，提高了编译效率。checker.py 在语法分析过程中嵌入语义检查，及时发现变量未声明、类型不匹配等错误，保证代码正确性。

完成语法和语义分析后，irgen.py 负责生成中间代码，为后端优化和目标代码生成提供桥梁。最终，codegen.py 将中间代码转换为目标平台可执行的代码，实现了编译器的后端功能。整个流程自前端输入到目标代码输出，均在一次扫描中高效完成，体现了递归下降法与一体化设计的优势。项目结构清晰，模块分工明确，既便于维护扩展，也为后续优化和功能增强提供了良好基础。

本次课设不仅加深了我对编译原理的理解，也丰富了开发经验，收获很多。