**编译原理期中报告**

**类Rust词法和语法分析工具设计与实现**

**徽标

描述已自动生成**

**小组成员：**

[Truman-min-show](https://github.com/Truman-min-show)

[TheTry16](https://github.com/TheTry16)

[limeszstone](https://github.com/limeszstone)

**院 系 同济大学计算机系**

**指导老师 丁志军**

**完成日期 2025.5.6**

**目录**

[一、实验概述 3](#_Toc197594645)

[1、目的与意义 3](#_Toc197594646)

[2、主要任务 3](#_Toc197594647)

[3、需求分析 4](#_Toc197594648)

[二、使用说明 4](#_Toc197594649)

[1、环境配置 4](#_Toc197594650)

[2、整体设计 5](#_Toc197594651)

[3、实现功能 7](#_Toc197594652)

[4、拓展功能 11](#_Toc197594653)

[三、详细设计 12](#_Toc197594654)

[1、词法分析 12](#_Toc197594655)

[2、语法分析 22](#_Toc197594656)

[3、可视化设计 46](#_Toc197594657)

[四、小组分工 56](#_Toc197594658)

[五、总结和展望 56](#_Toc197594659)

[1、总结 56](#_Toc197594660)

[2、展望 57](#_Toc197594661)

[六、参考文献 58](#_Toc197594662)

# 一、实验概述

## 1、目的与意义

编译器是计算机科学领域的基础核心技术之一，深刻理解其工作原理对于理解编程语言的底层机制，以及未来进行系统软件、编程语言设计等领域的研究与工作都具有重要意义。本实验通过实践，将理论知识转化为实际动手能力。实验旨在设计并实现一个针对简化版类Rust语言的编译器。其核心目的在于深入理解和实践编译器设计的基本原理与关键技术，具体包括：

* **掌握词法分析原理与实现:** 理解如何通过有限状态自动机或正则表达式将源代码文本流分解为有意义的词法单元。
* **掌握语法分析原理与实现:** 理解文法的概念，并掌握一种语法分析方法来构建抽象语法树，验证代码的语法结构。
* **理解抽象语法树:** 认识抽象语法树作为编译器内部核心数据结构的重要性，掌握抽象语法树节点的设计与构建方法。

## 2、主要任务

本项目的主要任务是分阶段完成类Rust语言编译器前端的各个核心组件：

* **词法分析器(Lexer):**

1. 定义类Rust语言的词法规则（关键字、标识符、常量、运算符、分隔符等）。
2. 实现一个Lexer，能够读取类Rust源代码文件，并输出Token序列，包含每个Token的类型、字面量、行号和列号信息。
3. 能够正确处理注释和空白字符。

* **语法分析器(Parser):**

1. 定义类Rust语言的文法，覆盖包括基本程序结构、函数声明与调用、变量声明、基本类型、控制流语句、返回语句、基本表达式等语法结构。
2. 设计并实现AST节点类，用于表示类Rust程序的语法结构。
3. 实现一个Parser，接收Lexer输出的Token序列，根据文法规则构建AST。
4. 能够检测并报告基本的语法错误。

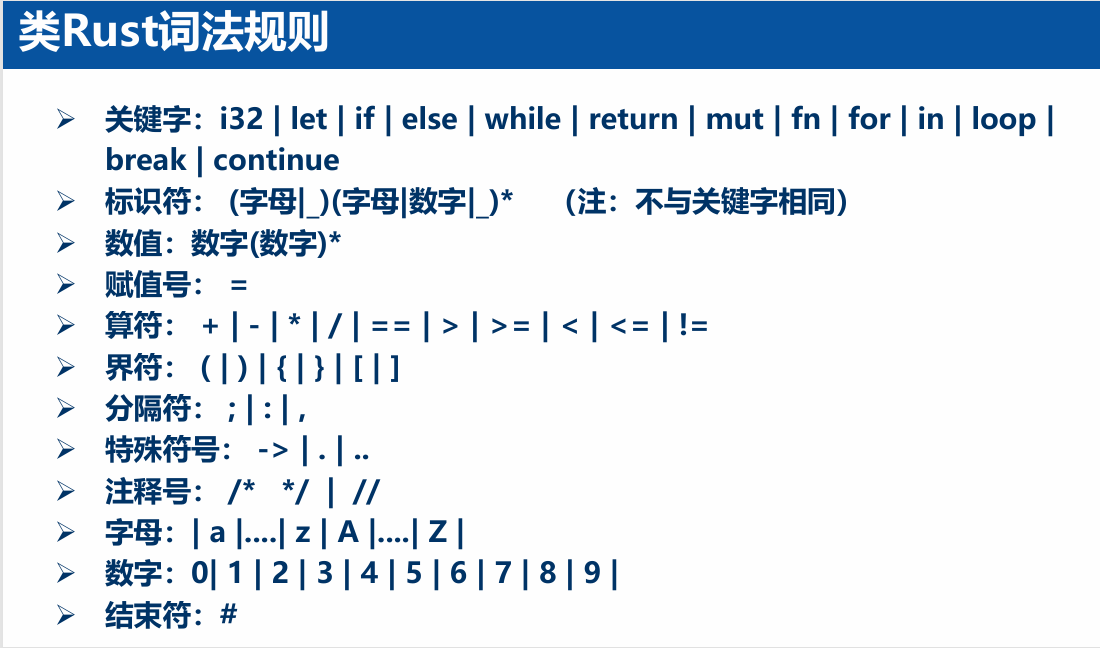
* **抽象语法树(AST)与可视化:**

1. 实现一个AST访问者模式用于遍历并可视化生成的AST结构，方便调试和理解。

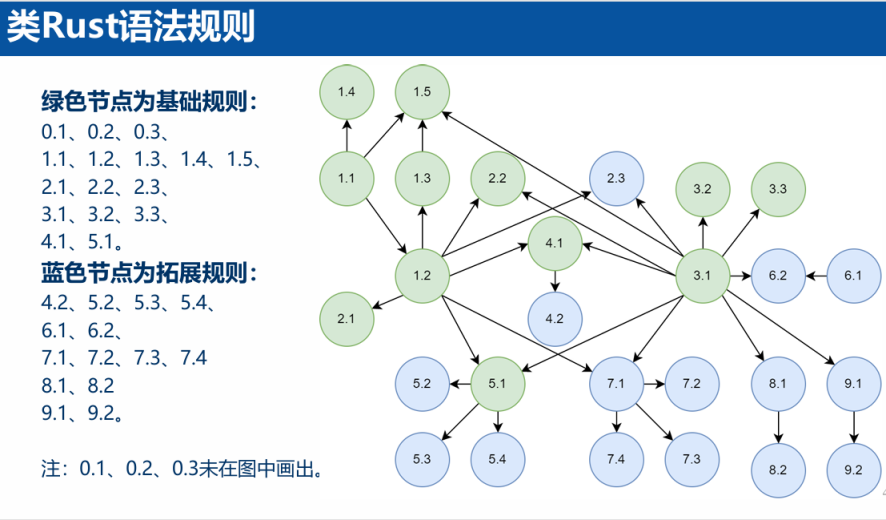
## 3、需求分析

本项目旨在实现的类Rust编译器前端需要满足以下基本需求：

* **输入:** 接收包含类Rust源代码的文本文件。
* **词法分析:** 将源代码文本流分解为一个个有意义的、符合预定规则的Token。



* **语法分析:** 依据语言的语法规则，检查词法分析产生的Token序列是否构成了一个有效的程序结构，并将其组织成一个AST。



* **输出:** 成功的词法分析输出Token序列。成功的语法分析输出AST 并将其可视化表示；分析失败时输出明确的错误信息

# 二、使用说明

## 1、环境配置

以下命令需要在代码目录下的命令行界面中执行，需要先行安装python 3.11、pip等工具，建议使用虚拟环境进行环境配置。

1. 安装依赖

requirements.txt中包含了python运行该项目需要的环境依赖，安装依赖的命令为：

pip install -r requirements.txt

1. 运行

程序main函数在gui.py中，因此运行需要以gui.py为主文件，同时由于生成语法树需要依赖graphviz\_bin中的文件，因此需要将graphviz\_bin放在代码的同一目录下。运行程序的命令为：

python gui.py

1. 打包

打包程序需要用到pyinstaller，打包命令如下：

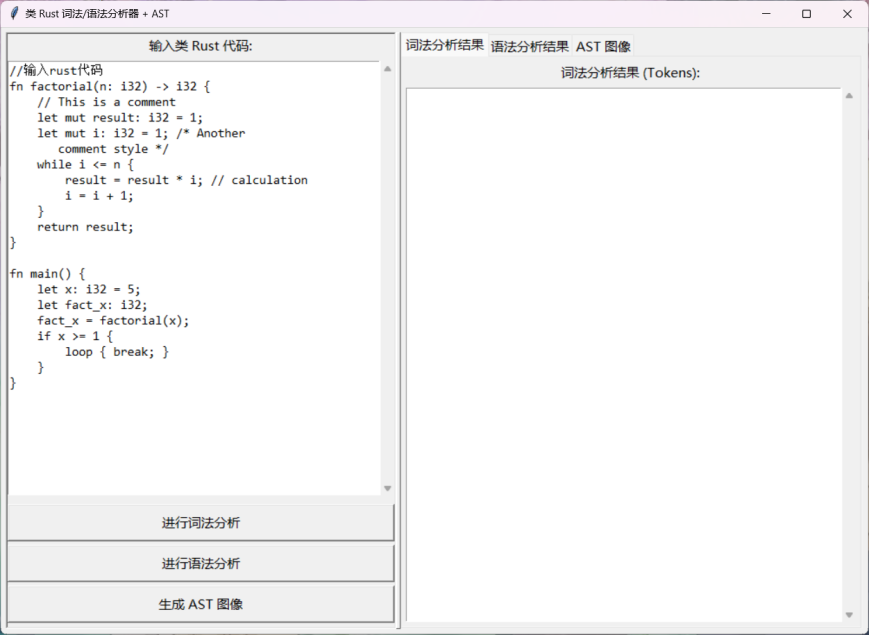
pyinstaller --onefile --noconsole --name=Rust-Compiler --icon=rust.ico gui.py

打包完成后的exe运行也需要graphviz\_bin的文件，因此运行打包后的Rust-Compiler.exe也需要将graphviz\_bin放在同一目录下运行。

## 2、整体设计

1. 初始界面

进入初始界面，左边是输入框，可以输入rust代码，初始展示的代码是写在gui.py内部的。我们的分析器可以通过在左侧编辑器输入代码，从而进行之后的分析过程。输入框下方有“进行词法分析”、“进行语法分析”、“生成AST图像”三个按钮，点击按钮后，可以生成相应的词法分析、语法分析、AST等结果，具体结果查看可以通过点击右半界面上方对应的标签，进入相应的界面查看。



1. 词法分析结果

点击左下方的“进行词法分析”按钮，点击右侧“词法分析结果”，即可在右侧界面生成相关代码的词法分析结果。词法分析结果对代码的每一行每一个token都进行了分析展示。由于词法分析结果较多，可以通过滚轮进行上下滑动查看全部内容。



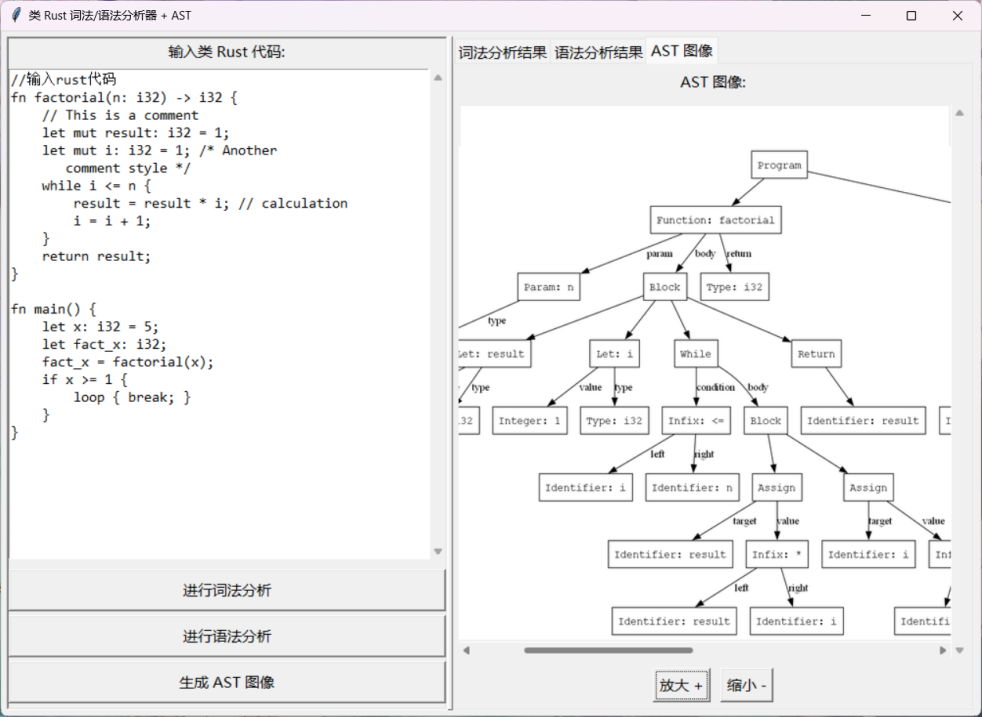
1. 语法分析结果

点击左下方的“进行语法分析”按钮，点击右侧“语法分析结果”，即可在右侧界面生成相关代码的语法分析结果。右侧界面上方输出语法分析是否有错误的结果，下方则是输出左边代码的语法结构。



1. AST图像

点击左下方的“生成AST图像”按钮，点击右侧“AST图像”，即可在右侧界面生成相关代码的语法树图像。语法树生成结果依赖于前面语法分析的语法结构，根据相关结构进行树的划分。同时，语法树图像还可以通过上下左右滚轮进行查看，同时在右侧界面下方提供放大、缩小按钮，可以通过其进行缩放，从而更好的查看图像。

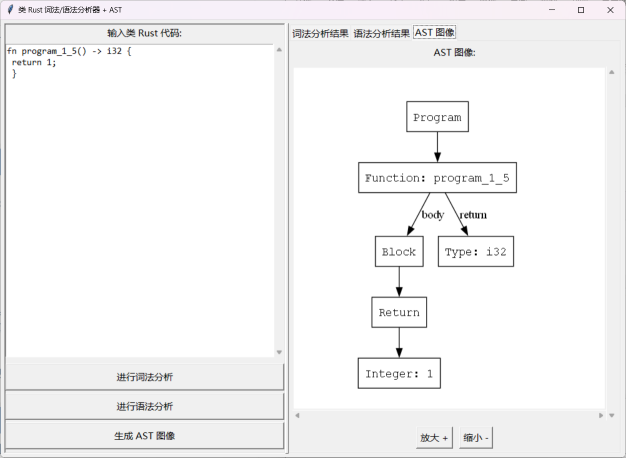


## 3、实现功能

1. 基础程序和函数输出

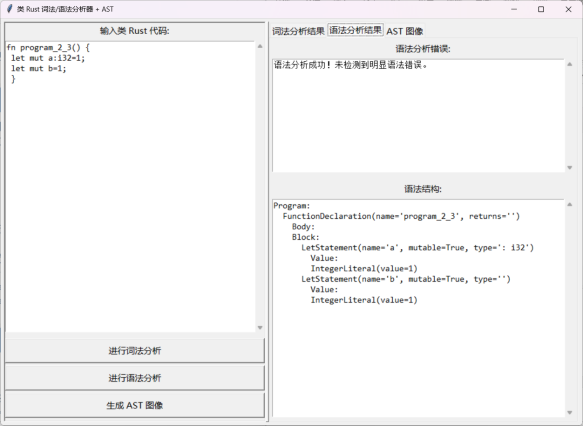
在这里，我们实现了0.1~1.5的基本功能，包括基础程序、语句、返回语句、函数输入、函数输出。具体测试代码见下图左侧的输入代码。相关运行结果如下：

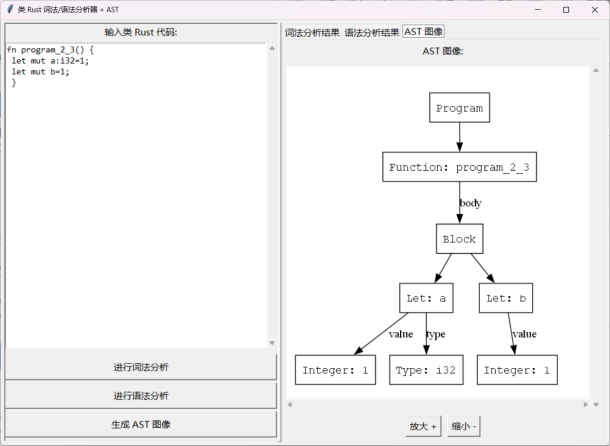




1. 变量声明和赋值语句

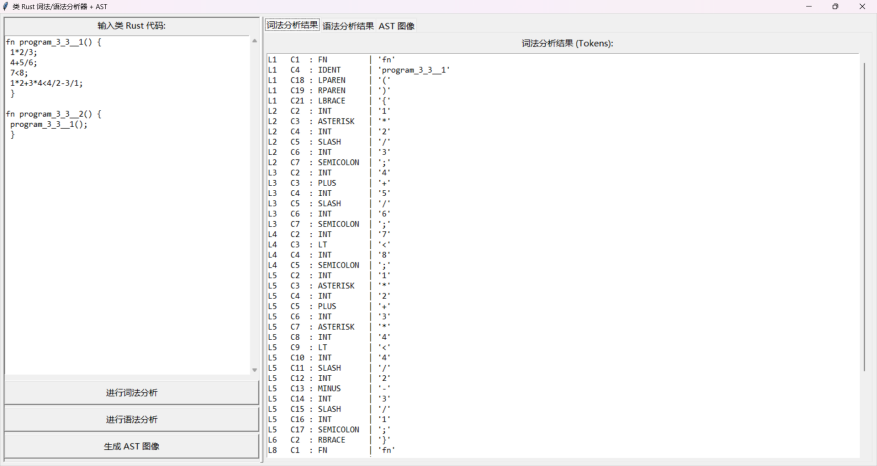
在这里，我们实现了2.1~2.3的功能，包括变量声明语句、赋值语句、变量声明赋值语句。具体测试代码见下图左侧的输入代码。相关运行结果如下：

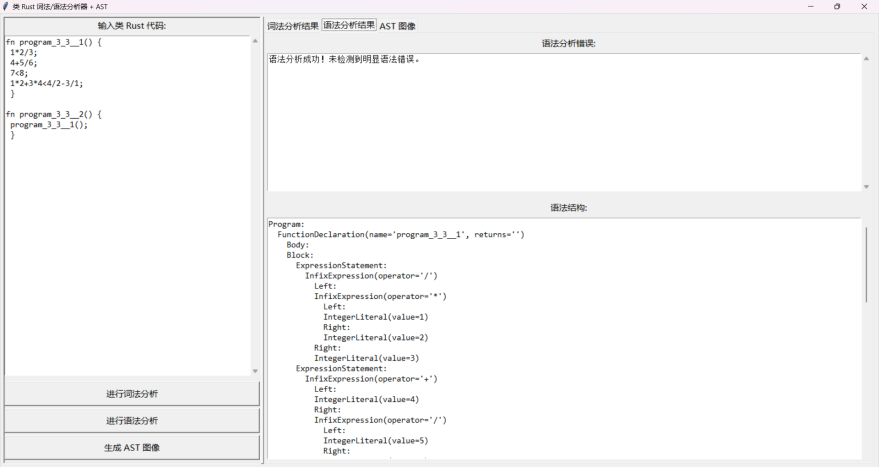


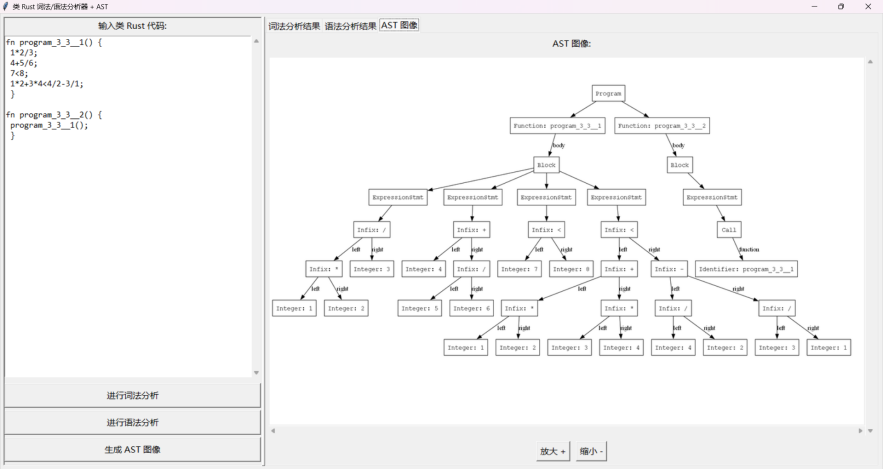


1. 基本表达式和函数调用

在这里，我们实现了3.1~3.3的功能，包括基本表达式、表达式增加计算和比较、函数调用。具体测试代码见下图左侧的输入代码。相关运行结果如下：



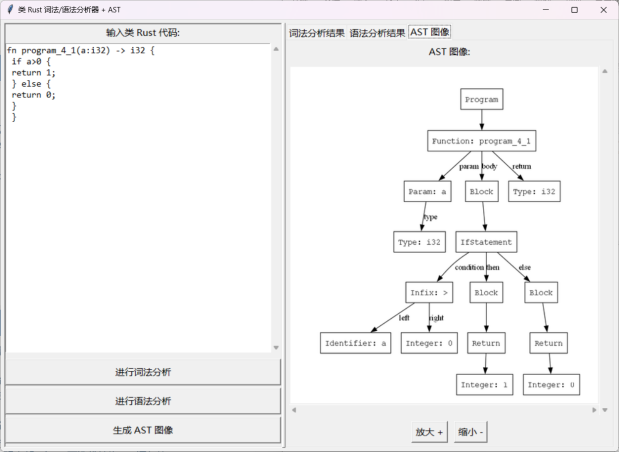




1. 选择结构

在这里，我们实现了4.1的功能，即选择结构。具体测试代码见下图左侧的输入代码。相关运行结果如下：

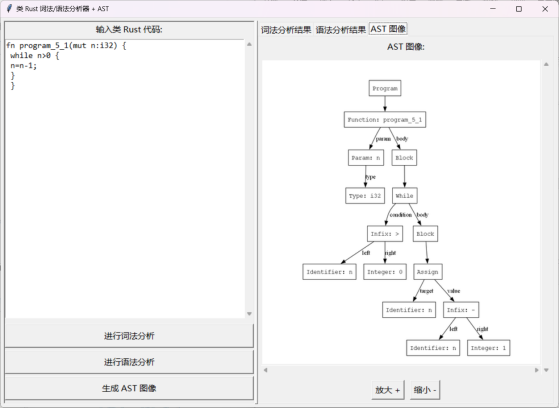




1. 循环结构

在这里，我们实现了4.1的功能，即选择结构。具体测试代码见下图左侧的输入代码。相关运行结果如下：



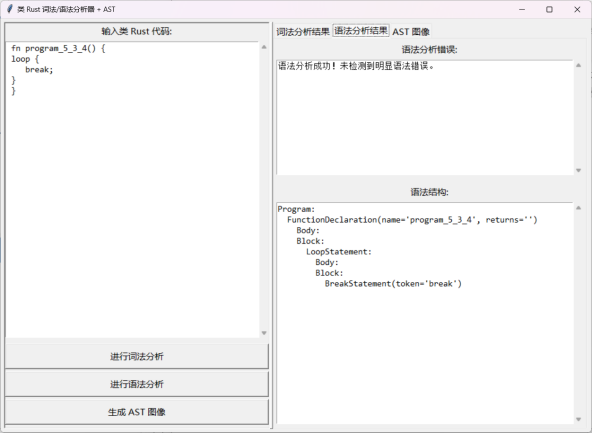


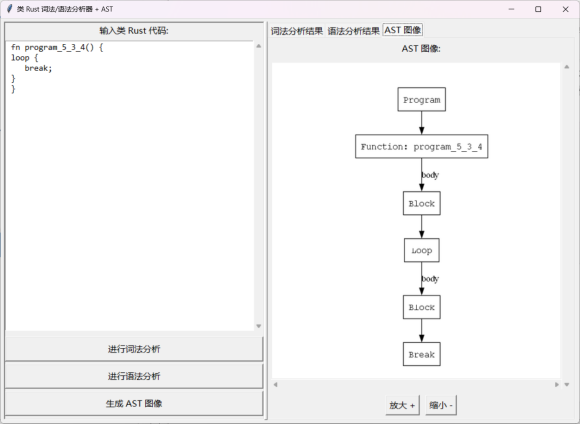
1. 结果分析

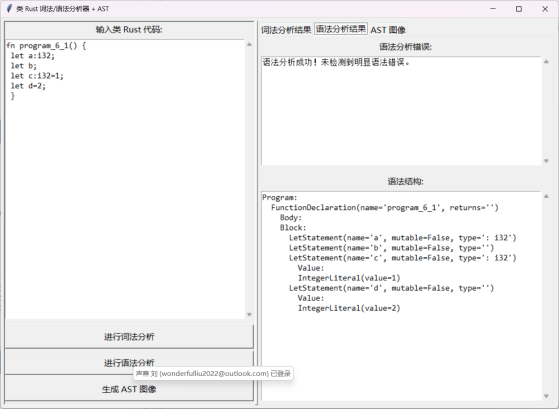
可以看到，程序均可以运行词法分析、语法分析以及生成语法树，相应的词法分析结果、语法分析结果、语法树图像无误，说明对测试程序的运行结果成功，相关功能实现成功。

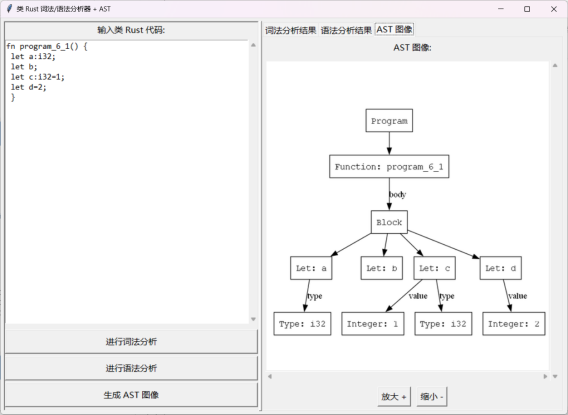
## 4、拓展功能

除了上述基本功能之外，我们还实现了5.3、5.4和6.1的功能内容，包括loop循环结构、增加break和continue、声明不可变变量。具体测试代码见下图左侧的输入代码。相关运行结果如下：









可以看到，以上程序均可以运行词法分析、语法分析以及生成语法树，相应的词法分析结果、语法分析结果、语法树图像无误，说明对测试程序的运行结果成功，相关功能实现成功。

# 三、详细设计

## 1、词法分析

1. 核心数据结构和思想

核心数据结构:

1. TokenType

这是词法分析的基础。它枚举了语言中所有可能的、有意义的最小单元类型（关键字、标识符、运算符、分隔符、字面量、EOF、非法字符等）。使用枚举提供了类型安全和清晰的代码。

class TokenType(enum.Enum):

    # Keywords

    LET = 'let'

    MUT = 'mut'

    FN = 'fn'

    RETURN = 'return'

    IF = 'if'

    ELSE = 'else'

    WHILE = 'while'

    FOR = 'for'

    IN = 'in'

    LOOP = 'loop'

    BREAK = 'break'

    CONTINUE = 'continue'

    I32 = 'i32' # Type keyword

    # Identifiers & Literals

    IDENT = 'IDENT' # Identifier

    INT = 'INT'     # Integer literal

    # Operators

    ASSIGN = '='

    PLUS = '+'

    MINUS = '-'

    ASTERISK = '\*'

    SLASH = '/'

    EQ = '=='      # Equal

    NOT\_EQ = '!='   # Not equal

    LT = '<'       # Less than

    LTE = '<='      # Less than or equal

    GT = '>'       # Greater than

    GTE = '>='      # Greater than or equal

    BANG = '!'      # Used in != (potentially future NOT)

    AMPERSAND = '&' # Reference/Borrow

    DOT = '.'       # Field access

    DOTDOT = '..'   # Range

    ARROW = '->'    # Function return type

    # Delimiters

    LPAREN = '('

    RPAREN = ')'

    LBRACE = '{'

    RBRACE = '}'

    LBRACKET = '['

    RBRACKET = ']'

    # Separators

    COMMA = ','

    COLON = ':'

    SEMICOLON = ';'

    # End of File

    EOF = 'EOF'

    # Illegal token (for errors)

    ILLEGAL = 'ILLEGAL'

1. Token

用于封装词法分析的结果。每个Token包含了四个关键信息：

type: 识别出的TokenType。

literal: 该 Token 在源代码中的原始字符串，这对于标识符、字面量和区分不同的运算符很重要。

line: Token在源代码中的起始行号。

col: Token在源代码中的起始列号。

# 使用 namedtuple 存储 Token 信息

# 参数: type (TokenType), literal (原始字符串), line (行号), col (列号)

Token = namedtuple('Token', ['type', 'literal', 'line', 'col'])

1. KEYWORDS

一个从关键字字符串到对应TokenType的映射，用于快速区分一个看起来像标识符的字符串到底是用户定义的名称还是语言的保留字。

KEYWORDS = {

    "let": TokenType.LET,

    "mut": TokenType.MUT,

    "fn": TokenType.FN,

    "return": TokenType.RETURN,

    "if": TokenType.IF,

    "else": TokenType.ELSE,

    "while": TokenType.WHILE,

    "for": TokenType.FOR,

    "in": TokenType.IN,

    "loop": TokenType.LOOP,

    "break": TokenType.BREAK,

    "continue": TokenType.CONTINUE,

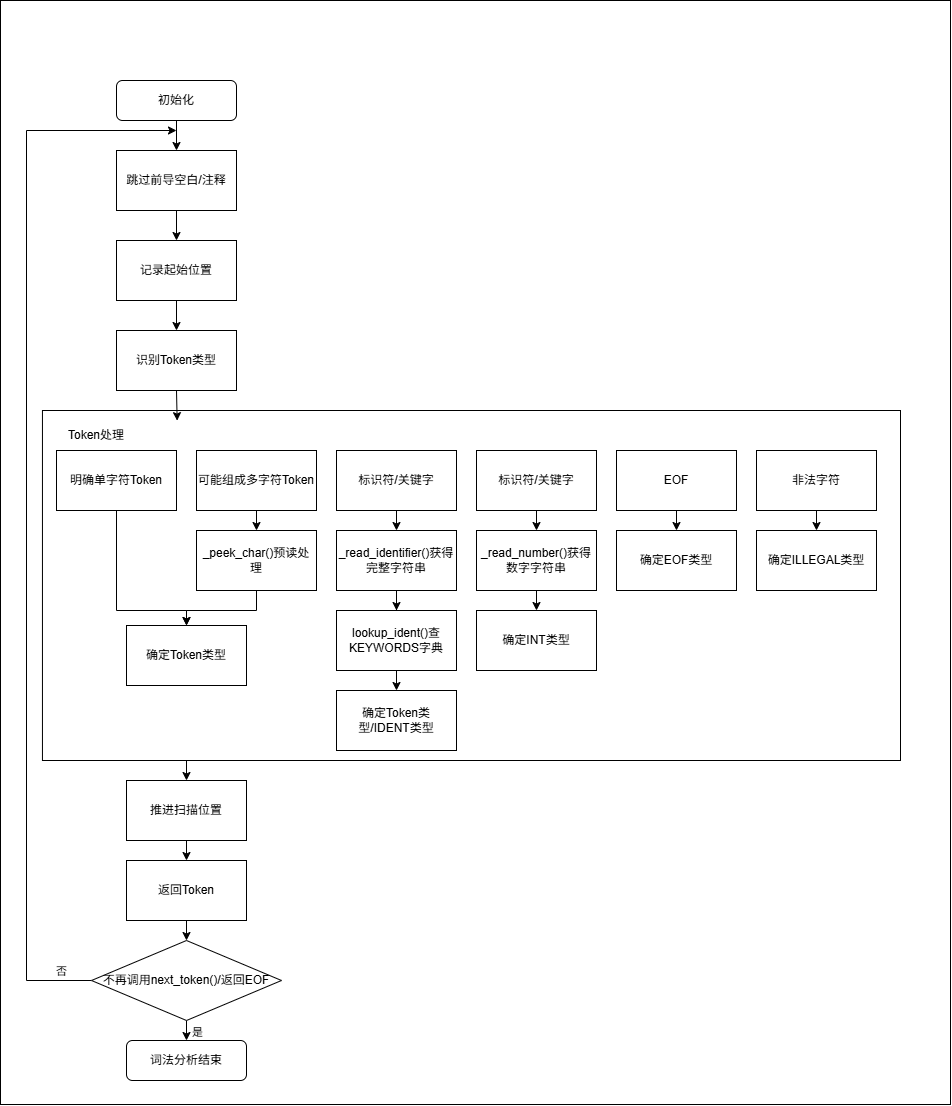
    "i32": TokenType.I32,

}

核心思想：

词法分析器的核心思想是通过逐个扫描源代码字符，并结合预读一个字符的能力，来匹配预定义的词法规则，从而将无结构的文本流切分成一系列带有类型、内容和位置信息的规范化Token单元。

1. 词法分析流程



1. 重点功能实现函数解析
2. **\_\_init\_\_(self, input\_code):**

功能: 初始化Lexer状态。

实现: 保存输入代码input\_code，初始化位置指针(position,read\_position)为0，行号line为1，列号col为0。最关键的是调用\_read\_char()来读取第一个字符，填充self.ch并将指针向前移动一位，为next\_token()的首次调用做好准备。

def \_\_init\_\_(self, input\_code: str):

        self.input = input\_code

        self.position = 0          # 当前字符在 input 中的索引

        self.read\_position = 0     # 下一个要读取的字符的索引 (position + 1)

        self.ch = ''               # 当前正在检查的字符

        self.line = 1              # 当前行号

        self.col = 0               # 当前列号 (相对于当前行)

        self.\_read\_char()          # 初始化 self.ch, self.position, self.read\_position

1. **\_read\_char(self):**

功能: 读取下一个字符，更新状态，处理EOF和换行。

实现: 检查read\_position是否超出输入长度。如果超出，将self.ch设为空字符串''表示EOF；否则，读取input[read\_position]到self.ch。然后，将position更新为当前的read\_position，并将read\_position加1，最后将col加1。

    def \_read\_char(self):

        """读取输入中的下一个字符，并更新位置指针"""

        if self.read\_position >= len(self.input):

            self.ch = ''  # 表示文件结束 (EOF)

        else:

            self.ch = self.input[self.read\_position]

        self.position = self.read\_position

        self.read\_position += 1

        self.col += 1

1. **\_peek\_char(self):**

功能: 查看下一个字符，但不移动指针。

实现: 类似\_read\_char，检查read\_position是否越界，但不更新self.position,self.read\_position,self.ch,line,col。只返回input[read\_position]处的字符或空字符串。

    def \_peek\_char(self) -> str:

        """查看下一个字符，但不移动指针 (用于预读)"""

        if self.read\_position >= len(self.input):

            return ''

        else:

            return self.input[self.read\_position]

1. **\_skip\_whitespace\_and\_comments(self):**

功能: 跳过所有无意义的字符。

实现: 使用whileTrue:循环。在循环内部，检查当前self.ch：

如果是空白(isspace())，如果是换行符则更新line和col，然后调用\_read\_char()读取下一个。

如果是/，则调用\_peek\_char()判断是//还是/\*。

如果是//，则内层循环调用\_read\_char()直到遇到\n或EOF。结束后continue外层循环。

如果是/\*，则消耗掉/\*，然后内层循环调用\_read\_char()直到找到\*/。需要处理嵌套（如果支持）和未闭合的情况。结束后continue外层循环。

如果既不是空白也不是注释开头，则break退出循环。

    def \_skip\_whitespace\_and\_comments(self):

        """跳过空白字符和注释 (包括换行符和不同类型的注释)"""

        while True:

            if self.ch.isspace():

                if self.ch == '\n':

                    self.line += 1

                    self.col = 0 # 换行后重置列号

                self.\_read\_char()

            elif self.ch == '/' and self.\_peek\_char() == '/': # 处理单行注释 //

                # 一直读到行尾或文件尾

                while self.ch != '\n' and self.ch != '':

                    self.\_read\_char()

                # \_read\_char 会停在换行符或EOF，继续循环以跳过换行符或处理下一个空白/注释

                continue # 非常重要，防止跳过换行符后直接退出循环

            elif self.ch == '/' and self.\_peek\_char() == '\*': # 处理多行注释 /\* ... \*/

                self.\_read\_char() # 消耗 /

                self.\_read\_char() # 消耗 \*

                start\_line, start\_col = self.line, self.col # 记录注释开始位置以防未闭合

                while not (self.ch == '\*' and self.\_peek\_char() == '/'):

                    if self.ch == '\n':

                        self.line += 1

                        self.col = 0

                    if self.ch == '': # 检测到未闭合的注释

                        print(f"警告: 在 {start\_line}:{start\_col} 开始的多行注释未闭合")

                        break

                    self.\_read\_char()

                if self.ch == '\*': # 消耗 \*

                    self.\_read\_char()

                if self.ch == '/': # 消耗 /

                     self.\_read\_char()

                continue # 继续检查注释或空白后的内容

            else:

                # 如果不是空白或注释的开始，则停止跳过

                break

1. **\_read\_identifier(self)和\_read\_number(self):**

功能: 读取连续的字符序列，组成标识符或数字。

实现: 记录起始位置start\_pos。使用while循环，只要当前self.ch满足标识符或数字的条件，就调用\_read\_char()读取下一个字符。循环结束后，self.position指向了标识符或数字之后的第一个字符。使用字符串切片self.input[start\_pos:self.position]获取完整的字面量字符串并返回。

    def \_read\_identifier(self) -> str:

        """读取一个完整的标识符"""

        start\_pos = self.position

        # 第一个字符必须是 is\_letter (包括 '\_')

        if not self.is\_letter(self.ch):

             return "" # Should not happen if called correctly

        # 后续字符可以是 is\_letter 或数字

        while self.is\_letter(self.ch) or self.ch.isdigit():

            self.\_read\_char()

        return self.input[start\_pos:self.position]

    def \_read\_number(self) -> str:

        """读取一个完整的整数"""

        start\_pos = self.position

        while self.ch.isdigit():

            self.\_read\_char()

        return self.input[start\_pos:self.position]

1. **lookup\_ident(ident):**

功能: 区分关键字和普通标识符。

实现: 在KEYWORDS字典中查找输入的ident字符串。如果找到，返回对应的关键字TokenType；如果找不到，返回TokenType.IDENT。

def lookup\_ident(ident: str) -> TokenType:

    """检查标识符是否是关键字"""

    return KEYWORDS.get(ident, TokenType.IDENT)

1. **next\_token(self):**

功能: 驱动整个词法分析过程的核心，生成下一个Token。

实现: 如“词法分析流程”中所述，先跳过空白和注释，然后根据self.ch和可能的\_peek\_char()结果，匹配不同的Token规则，调用相应的读取函数（如\_read\_identifier,\_read\_number）或直接创建Token对象，最后推进字符指针\_read\_char()。

    def next\_token(self) -> Token:

        """获取并返回下一个 Token"""

        self.\_skip\_whitespace\_and\_comments()

        tok = None

        start\_line, start\_col = self.line, self.col # 记录 Token 的起始位置

        # --- 根据当前字符 self.ch 确定 Token 类型 ---

        match self.ch:

            case '=':

                if self.\_peek\_char() == '=': # '=='

                    ch = self.ch

                    self.\_read\_char() # 消耗第二个 '='

                    literal = ch + self.ch

                    tok = Token(TokenType.EQ, literal, start\_line, start\_col)

                else: # '='

                    tok = Token(TokenType.ASSIGN, self.ch, start\_line, start\_col)

            case '+':

                tok = Token(TokenType.PLUS, self.ch, start\_line, start\_col)

            case '-':

                 if self.\_peek\_char() == '>': # '->'

                     ch = self.ch

                     self.\_read\_char() # 消耗 '>'

                     literal = ch + self.ch

                     tok = Token(TokenType.ARROW, literal, start\_line, start\_col)

                 else: # '-'

                     tok = Token(TokenType.MINUS, self.ch, start\_line, start\_col)

            case '\*':

                tok = Token(TokenType.ASTERISK, self.ch, start\_line, start\_col)

            case '/':

                # 注释已被 \_skip\_whitespace\_and\_comments 处理

                 tok = Token(TokenType.SLASH, self.ch, start\_line, start\_col)

            case '<':

                 if self.\_peek\_char() == '=': # '<='

                     ch = self.ch

                     self.\_read\_char() # 消耗 '='

                     literal = ch + self.ch

                     tok = Token(TokenType.LTE, literal, start\_line, start\_col)

                 else: # '<'

                    tok = Token(TokenType.LT, self.ch, start\_line, start\_col)

            case '>':

                 if self.\_peek\_char() == '=': # '>='

                     ch = self.ch

                     self.\_read\_char() # 消耗 '='

                     literal = ch + self.ch

                     tok = Token(TokenType.GTE, literal, start\_line, start\_col)

                 else: # '>'

                     tok = Token(TokenType.GT, self.ch, start\_line, start\_col)

            case '!':

                 if self.\_peek\_char() == '=': # '!='

                     ch = self.ch

                     self.\_read\_char() # 消耗 '='

                     literal = ch + self.ch

                     tok = Token(TokenType.NOT\_EQ, literal, start\_line, start\_col)

                 else: # '!' 单独出现暂时视为非法，或定义为 BANG

                     tok = Token(TokenType.ILLEGAL, self.ch, start\_line, start\_col)

                     # tok = Token(TokenType.BANG, self.ch, start\_line, start\_col)

            case '&':

                tok = Token(TokenType.AMPERSAND, self.ch, start\_line, start\_col)

            case '.':

                 if self.\_peek\_char() == '.': # '..'

                     ch = self.ch

                     self.\_read\_char() # 消耗第二个 '.'

                     literal = ch + self.ch

                     tok = Token(TokenType.DOTDOT, literal, start\_line, start\_col)

                 else: # '.'

                     tok = Token(TokenType.DOT, self.ch, start\_line, start\_col)

            case '(':

                tok = Token(TokenType.LPAREN, self.ch, start\_line, start\_col)

            case ')':

                tok = Token(TokenType.RPAREN, self.ch, start\_line, start\_col)

            case '{':

                tok = Token(TokenType.LBRACE, self.ch, start\_line, start\_col)

            case '}':

                tok = Token(TokenType.RBRACE, self.ch, start\_line, start\_col)

            case '[':

                tok = Token(TokenType.LBRACKET, self.ch, start\_line, start\_col)

            case ']':

                tok = Token(TokenType.RBRACKET, self.ch, start\_line, start\_col)

            case ',':

                tok = Token(TokenType.COMMA, self.ch, start\_line, start\_col)

            case ':':

                tok = Token(TokenType.COLON, self.ch, start\_line, start\_col)

            case ';':

                tok = Token(TokenType.SEMICOLON, self.ch, start\_line, start\_col)

            # --- 处理标识符和数字 ---

            case \_ if self.is\_letter(self.ch): # 检查是否是标识符的开头

                literal = self.\_read\_identifier() # 读取完整的标识符

                token\_type = lookup\_ident(literal) # 判断是关键字还是普通标识符

                # 注意: \_read\_identifier 已经移动了指针，不需要再调用 \_read\_char()

                return Token(token\_type, literal, start\_line, start\_col)

            case \_ if self.ch.isdigit(): # 检查是否是数字开头

                literal = self.\_read\_number() # 读取完整的数字

                # 注意: \_read\_number 已经移动了指针

                return Token(TokenType.INT, literal, start\_line, start\_col)

            # --- 处理文件结束 ---

            case '': # 文件结束符

                tok = Token(TokenType.EOF, "", start\_line, start\_col)

            # --- 处理无法识别的字符 ---

            case \_:

                tok = Token(TokenType.ILLEGAL, self.ch, start\_line, start\_col)

        # 对于上面 case 中匹配到的单个或两个字符的 token，需要移动到下一个字符

        self.\_read\_char()

        return tok

## 2、语法分析

1. 核心数据结构和思想

核心数据结构：

1. 抽象语法树 (AST) 节点
2. 基类Node

所有AST节点的抽象基类。它通常包含一个 token 属性，用于存储与该节点相关的第一个词法单元，这对于错误报告和定位非常有用。所有节点都必须实现 accept(self, visitor) 方法，以支持访问者设计模式，方便后续对AST进行遍历和操作（如AST打印、语义分析、代码生成等）。

class Node(ABC):

    def \_\_init\_\_(self, token: Token):

        self.token = token Store first relevant token for location info

    @abstractmethod

    def accept(self, visitor):

        pass

    def token\_literal(self):

        return self.token.literal if self.token else ""

1. 语句节点

继承自 Node，代表程序中的一个执行单元，通常不直接返回值。具体的各部件statement定义如下：

* Program(statements: list[Statement]): AST的根节点，包含一系列顶层语句。
* LetStatement(name: Identifier, value: Expression, type\_info: TypeNode, mutable: bool): 表示 let 声明。
* ReturnStatement(return\_value: Expression): 表示 return 语句。
* ExpressionStatement(expression: Expression): 表示一个表达式后跟分号构成的语句。
* BlockStatement(statements: list[Statement]): 表示一个由 {} 包围的语句块。
* IfStatement(condition: Expression, consequence: BlockStatement, alternative: BlockStatement): 表示 if-else 结构。
* WhileStatement(condition: Expression, body: BlockStatement): 表示 while 循环。
* LoopStatement(body: BlockStatement): 表示 loop 无限循环。
* BreakStatement(): 表示 break 语句。
* ContinueStatement(): 表示 continue 语句。
* AssignmentStatement(target: Expression, value: Expression): 表示赋值语句。
* FunctionDeclarationStatement(name: Identifier, params: list[ParameterNode], body: BlockStatement, return\_type: TypeNode): 表示函数声明。

1. 表达式节点

继承自 Node，代表程序中能够计算出一个值的结构。具体的各部件expression定义如下：

* Identifier(value: str): 表示标识符（如变量名、函数名）。
* IntegerLiteral(value: int): 表示整数字面量。
* PrefixExpression(operator: str, right: Expression): 表示前缀表达式（如 -x, !flag）。
* InfixExpression(left: Expression, operator: str, right: Expression): 表示中缀表达式（如 x + y, a == b）。
* BooleanLiteral(value: bool):表示布尔字面量 true 或 false。
* CallExpression(function: Expression, arguments: list[Expression]): 表示函数调用。

class Statement(Node):

    pass

class Expression(Node):

    pass

class Program(Node):

    def \_\_init\_\_(self, statements: list[Statement]):

        super().\_\_init\_\_(statements[0].token if statements else None)

        self.statements = statements

    def accept(self, visitor):

        return visitor.visit\_program(self)

1. 解析器 (Parser) 内部数据结构
2. 解析过程相关

* lexer: Lexer: 词法分析器的实例，用于逐个获取词法单元。
* current\_token: Token: 当前正在处理的词法单元。
* peek\_token: Token: 预读的下一个词法单元。这种“向前看一个词法单元”的机制对于许多语法解析决策至关重要。
* errors: list[str]: 用于存储在解析过程中遇到的语法错误信息。

1. Pratt 解析器相关

* prefix\_parse\_fns: dict[TokenType, Callable[[], Expression | None]]: 一个字典，键是词法单元类型 (TokenType)，值是对应的前缀解析函数。当解析表达式时，如果 current\_token 的类型在此字典中，则调用相应的函数来处理以该词法单元开头的表达式部分。
* infix\_parse\_fns: dict[TokenType, Callable[[Expression], Expression | None]]: 一个字典，键是词法单元类型，值是对应的中缀解析函数。当解析表达式时，如果 peek\_token 的类型是一个中缀操作符且在此字典中，则调用相应的函数来处理该操作符及其右操作数，并将已解析的左操作数作为参数传入。
* PRECEDENCES: dict[TokenType, int]: 一个字典，定义了不同操作符的优先级。这是Pratt解析器正确处理表达式运算顺序的关键。优先级值越小，优先级越低。

LOWEST = 0

EQUALS = 1 ==, !=

LESSGREATER = 2 <, >, <=, >=

SUM = 3 +, -

PRODUCT = 4 \*, /

PREFIX = 5 -X or !X

CALL = 6myFunction(X)

... 其他优先级

PRECEDENCES = {

TokenType.EQ: EQUALS,

TokenType.NOT\_EQ: EQUALS,

TokenType.PLUS: SUM,

... 其他操作符

TokenType.LPAREN: CALL,

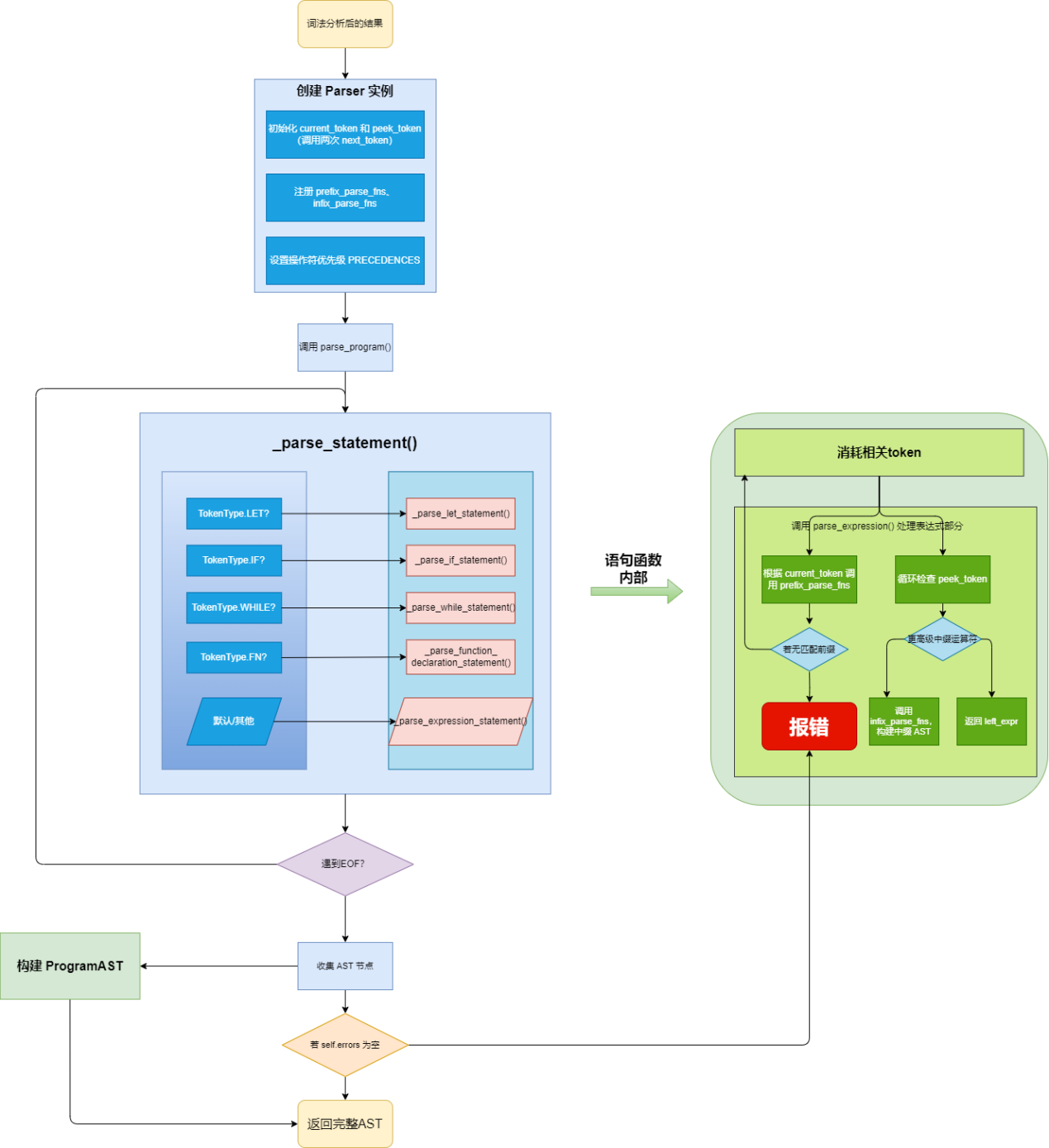
}

核心思想：

本项目的语法分析器采用了混合策略，以发挥不同解析技术的优势：

* 递归下降：对于语句（如let,return,if,while,fn）和程序顶层结构，采用递归下降的方法。这意味着每种主要的语法结构（或非终结符）都有一个对应的解析函数。例如，\_parse\_let\_statement()负责解析let语句，\_parse\_if\_statement()负责解析if语句。这些函数会根据文法规则消耗词法单元，并递归地调用其他解析函数来处理子结构（例如，\_parse\_let\_statement会调用parse\_expression来解析赋值表达式的右侧）。这种方法直观且易于实现，因为代码结构直接映射文法规则。
* Pratt解析器：对于表达式的解析，尤其是包含多种操作符且涉及不同优先级和结合性的情况，Pratt解析器是一种非常优雅和高效的解决方案。它通过prefix\_parse\_fns和infix\_parse\_fns以及操作符优先级表PRECEDENCES来驱动。parse\_expression(precedence) 函数是Pratt解析器的核心，它根据当前词法单元调用前缀解析函数，然后在一个循环中查找并调用优先级更高的中缀解析函数，直到表达式解析完毕。这种方法避免了传统递归下降解析表达式时需要为每个优先级水平编写单独函数的复杂性。

1. 语法分析流程



1. 重点功能函数实现解析
2. **基础程序结构与函数 (规则0.x, 1.1, 1.4, 1.5)**

Program -> <声明串>(1.1)

对应函数:parse\_program()

运作描述: 这是解析器的入口。它循环调用\_parse\_statement()来解析一系列声明（主要是函数声明），直到遇到文件结束符。所有解析到的顶层语句（StatementAST节点）被收集并存入一个ProgramAST节点的statements列表中。

def parse\_program(self) -> Program | None:

        statements = []

        while not self.\_current\_token\_is(TokenType.EOF):

            # Keep original top-level dispatch logic

            stmt = self.\_parse\_statement()

            if stmt:

                statements.append(stmt)

            elif self.errors:

                return None # Error occurred

        if self.errors: return None

        return Program(statements=statements)

<声明> -> <函数声明>(1.1)

<函数声明> -> <函数头声明> <语句块>(1.1)

<函数头声明> -> fn <ID> '('<形参列表>')' ['->' <类型>](1.1, 1.5)

对应函数:\_parse\_function\_declaration\_statement()

运作描述:

1. 当\_parse\_statement()检测到fn关键字时调用此函数。

2. 消耗fn关键字。

3. 期望并消耗一个标识符（函数名），创建Identifier节点。

4. 期望并消耗左括号(.

5. 调用\_parse\_function\_parameters()解析参数列表。

6. 期望并消耗右括号).

7. 检查是否有返回类型指示符->。如果有，则消耗->，并调用\_parse\_type()解析返回类型，创建TypeNode。

8. 期望一个左花括号{，表明函数体的开始。

9. 调用\_parse\_block\_statement()解析函数体（语句块）。

10. 函数体解析完毕后，\_parse\_block\_statement会将current\_token停留在右花括号}上，此函数会消耗它。

11. 构建并返回FunctionDeclarationStatementAST节点。

def \_parse\_function\_declaration\_statement(self) -> FunctionDeclarationStatement | None:

        # Keep original token consumption carefully

        fn\_token = self.current\_token

        if not self.\_expect\_peek(TokenType.IDENT): return None

        name\_node = Identifier(token=self.current\_token, value=self.current\_token.literal)

        if not self.\_expect\_peek(TokenType.LPAREN): return None

        params = self.\_parse\_function\_parameters() # Returns list[ParameterNode]

        if params is None: return None

        # Assume original \_parse\_function\_parameters left current on ')'

        if not self.\_current\_token\_is(TokenType.RPAREN):

             self.errors.append(f"内部错误: \_parse\_function\_parameters 未停留在 ')' L{self.current\_token.line}")

             return None

        self.\_next\_token() # Consume ')'

        return\_type\_node = None

        if self.\_current\_token\_is(TokenType.ARROW):

             self.\_next\_token()

             return\_type\_node = self.\_parse\_type() # Returns TypeNode

             if return\_type\_node is None: return None

             self.\_next\_token() # Consume type's last token

        if not self.\_current\_token\_is(TokenType.LBRACE):

             msg = f"语法错误 (L{self.current\_token.line} C{self.current\_token.col}): 函数声明期望左花括号开始函数体, 得到 {self.current\_token.type.name} ('{self.current\_token.literal}')";

             self.errors.append(msg); return None

        body\_node = self.\_parse\_block\_statement() # Returns BlockStatement

        if body\_node is None: return None

        # Assume original consumed '}' after block here

        self.\_next\_token() # Consume '}'

        return FunctionDeclarationStatement(token=fn\_token, name=name\_node, params=params, body=body\_node, return\_type=return\_type\_node)

<形参列表> -> 空 | <形参> | <形参> ',' <形参列表>(1.1, 1.4)

<形参> -> [mut] <ID> ':' <类型>(1.4, 结合0.1, 0.2)

对应函数:\_parse\_function\_parameters()

运作描述:

1. 如果预读到右括号)，则为空参数列表，直接返回空列表。

2. 否则，进入循环解析参数：

检查是否有mut关键字，记录是否可变。

期望并消耗标识符（参数名）。

期望并消耗冒号:.

调用\_parse\_type()解析参数类型。

创建ParameterNode并添加到参数列表中。

如果预读到逗号,，则消耗它并继续循环；如果预读到右括号)，则结束循环。否则报错。

3. 返回参数节点列表。

    def \_parse\_function\_parameters(self) -> list[ParameterNode] | None:

        # Keep original token consumption carefully, ensure it leaves current on ')' if successful

        params = []

        if self.\_peek\_token\_is(TokenType.RPAREN):

            # Original working version \*must\* have advanced here to make caller work

            self.\_next\_token() # Consume '(', current becomes ')'

            return params

        self.\_next\_token() # Consume '('

        while True:

            param\_token = self.current\_token; is\_mut = False; name\_token = None

            if self.\_current\_token\_is(TokenType.MUT):

                is\_mut = True

                if not self.\_expect\_peek(TokenType.IDENT): return None

                name\_token = self.current\_token

            elif self.\_current\_token\_is(TokenType.IDENT): name\_token = self.current\_token

            else: msg = f"语法错误 (L{self.current\_token.line} C{self.current\_token.col}): 期望参数名 (IDENT) 或 'mut', 得到 {self.current\_token.type.name} ('{self.current\_token.literal}')"; self.errors.append(msg); return None

            name\_node = Identifier(token=name\_token, value=name\_token.literal)

            if not self.\_expect\_peek(TokenType.COLON): return None

            self.\_next\_token() # Consume ':'

            type\_node = self.\_parse\_type() # Returns TypeNode

            if type\_node is None: return None

            params.append(ParameterNode(token=param\_token, name=name\_node, type\_info=type\_node, mutable=is\_mut))

            # Keep original check logic for ',' or ')'

            if self.\_peek\_token\_is(TokenType.COMMA):

                 self.\_next\_token(); self.\_next\_token() # Consume type, consume ','

                 if self.\_current\_token\_is(TokenType.RPAREN): msg = f"语法错误 (L{self.current\_token.line} C{self.current\_token.col}): 参数列表中逗号后不能直接跟 ')', 期望参数"; self.errors.append(msg); return None

            elif self.\_peek\_token\_is(TokenType.RPAREN):

                 self.\_next\_token() # Consume type's last token

                 break # current is now ')'

            else: self.\_peek\_error(TokenType.RPAREN); return None

        return params

<语句块> -> '{' <语句串> '}'(1.1)

<语句串> -> 空 | <语句> <语句串>(1.1, 1.2)

对应函数:\_parse\_block\_statement()

运作描述:

1. 期望当前词法单元是左花括号{，并消耗它。

2. 进入循环，反复调用\_parse\_statement()解析块内的每一条语句，直到遇到右花括号}或文件结束符。

3. 解析到的语句AST节点被收集起来。

4. 当遇到右花括号}时，循环结束。此函数不会消耗右花括号，而是将其保留为current\_token，由调用者（如\_parse\_function\_declaration\_statement或\_parse\_if\_statement）来消耗。

5. 构建并返回BlockStatementAST节点。

def \_parse\_block\_statement(self) -> BlockStatement | None:

        block\_token = self.current\_token  # '{' token

        if not self.\_current\_token\_is(TokenType.LBRACE):

            msg = f"内部错误: \_parse\_block\_statement 在非 '{{' Token 调用 (L{self.current\_token.line})"

            self.errors.append(msg)

            return None

        self.\_next\_token()  # 消耗 '{', current\_token 现在是块内第一个 token (或 '}')

        statements = []

        # 循环直到遇到 '}' 或 EOF

        while not self.\_current\_token\_is(TokenType.RBRACE) and not self.\_current\_token\_is(TokenType.EOF):

            # 在这里直接调用 \_parse\_statement

            # \_parse\_statement 会解析一个完整语句，并消耗包括分号在内的所有相关 token

            # 成功时，\_parse\_statement 会将 current\_token 定位到下一个语句的开始 token

            # 如果发生错误，\_parse\_statement 会添加错误并返回 None

            stmt\_node = self.\_parse\_statement()

            if stmt\_node:

                statements.append(stmt\_node)

            elif self.errors:

                # 如果 \_parse\_statement 内部记录了错误，则传播错误

                return None

        # 循环结束时，current\_token 应该是 '}' 或 EOF

        if not self.\_current\_token\_is(TokenType.RBRACE):

            # 如果不是 '}'，说明块没有正确闭合

            self.\_peek\_error(TokenType.RBRACE)  # 报告期望 '}'

            return None

        # \_parse\_block\_statement 在成功时，将 current\_token 停留在 '}' 上。

        # 调用者 (\_parse\_while\_statement, \_parse\_function\_declaration\_statement etc.)

        # 负责在调用 \_parse\_block\_statement 之后检查并消耗这个 '}' Token。

        return BlockStatement(token=block\_token, statements=statements)

<类型> -> i32(0.2)

对应函数:\_parse\_type()

运作描述: 当前只支持i32。检查current\_token是否为TokenType.I32。如果是，则创建并返回一个表示i32的TypeNode。此函数不消耗该类型词法单元，而是让调用者消耗。

def \_parse\_type(self) -> TypeNode | None:

        type\_token = self.current\_token

        if self.\_current\_token\_is(TokenType.I32):

            # Return node, DO NOT advance token here.

            return TypeNode(token=type\_token, name="i32")

        else:

            msg = f"语法错误 (L{type\_token.line} C{type\_token.col}): 期望类型名 (如 i32), 得到 {type\_token.type.name} ('{type\_token.literal}')"

            self.errors.append(msg)

            return None

1. **语句与返回语句 (规则1.2, 1.3)**

<语句> -> ';'(空语句) (1.2)

对应函数:\_parse\_statement()内的一个分支。

运作描述: 如果current\_token是分号;，则消耗它。通常返回None或一个特殊的空语句节点（如果定义了的话），表示一个有效的空结构。

elif self.\_current\_token\_is(TokenType.SEMICOLON):

            # 处理空语句 ;

            self.\_next\_token()  # 消耗 ';'

            return None  # 返回 None 表示一个有效的空结构

<语句> -> <返回语句>(1.3)

<返回语句> -> return ';' | return <表达式> ';'(1.3, 1.5)

对应函数:\_parse\_return\_statement()

运作描述:

1. 当\_parse\_statement()检测到return关键字时调用。

2. 记录return词法单元，并消耗它。

3. 检查当前词法单元是否为分号;。

如果不是分号，则调用self.parse\_expression(LOWEST)解析返回值表达式。

解析表达式后，current\_token会停在表达式的最后一个词法单元上。

4. 期望下一个词法单元（如果前面有表达式）或当前词法单元（如果前面无表达式）是分号;。消耗分号。

5. 构建并返回ReturnStatementAST节点，其return\_value可能是表达式AST节点或None。

def \_parse\_return\_statement(self) -> ReturnStatement | None:

        # Keep original token consumption

        return\_token = self.current\_token

        self.\_next\_token()

        return\_value\_node = None

        if not self.\_current\_token\_is(TokenType.SEMICOLON):

            return\_value\_node = self.parse\_expression(LOWEST) # Returns Node

            if return\_value\_node is None: return None

        if not self.\_current\_token\_is(TokenType.SEMICOLON):

             if not self.\_expect\_peek(TokenType.SEMICOLON): return None

        else: self.\_next\_token()

        return ReturnStatement(token=return\_token, return\_value=return\_value\_node)

1. **变量声明与赋值 (规则2.1, 2.2, 2.3, 结合0.1, 0.3)**

<语句> -> <变量声明语句> | <赋值语句> | <变量声明赋值语句>(2.1, 2.2, 2.3)

这些语句的解析由\_parse\_statement()分发。

<变量声明语句> -> let [mut] <ID> [':' <类型>] ';'(2.1, 结合0.1, 0.2)

对应函数:\_parse\_let\_statement()(处理不带赋值的部分)

运作描述:

1. 消耗let。

2. 可选地消耗mut并记录。

3. 期望并消耗标识符<ID>。

4. 可选地：消耗:，调用\_parse\_type()解析类型。

5. 期望并消耗分号;。

6. 构建LetStatementAST节点，此时value成员为None。

def \_parse\_let\_statement(self) -> LetStatement | None:

        print(f"DEBUG: \_parse\_let\_statement: 开始, current={self.current\_token}")

        let\_token = self.current\_token  # 'let'

        is\_mut = False

        # 1. 处理 'mut' (如果存在)

        if self.\_peek\_token\_is(TokenType.MUT):

            is\_mut = True

            self.\_next\_token()  # 消耗 let, current\_token 现在是 'mut'

            print(f"DEBUG: \_parse\_let\_statement: 'mut' 之前, current={self.current\_token}")

            if not self.\_expect\_peek(TokenType.IDENT): return None  # 消耗 'mut', current\_token 现在是 IDENT

            print(f"DEBUG: \_parse\_let\_statement: 'mut' 之后, current={self.current\_token} (应为 IDENT)")

        # 2. 处理 IDENT (如果 'mut' 不存在或已处理)

        elif self.\_peek\_token\_is(TokenType.IDENT):

            self.\_next\_token()  # 消耗 'let', current\_token 现在是 IDENT

            print(f"DEBUG: \_parse\_let\_statement: 'let' 之后, current={self.current\_token} (应为 IDENT)")

        else:

            self.\_peek\_error(TokenType.IDENT)

            return None

        # current\_token 现在是 IDENT

        name\_ident\_token = self.current\_token

        name\_node = Identifier(token=name\_ident\_token, value=name\_ident\_token.literal)

        print(f"DEBUG: \_parse\_let\_statement: 获得 IDENT '{name\_node.value}', current={self.current\_token}")

        type\_node = None

        # 3. 处理可选的类型注解 ': Type'

        if self.\_peek\_token\_is(TokenType.COLON):

            self.\_next\_token()  # 消耗 IDENT, current\_token 现在是 ':'

            print(f"DEBUG: \_parse\_let\_statement: 类型 ':' 之前, current={self.current\_token}")

            self.\_next\_token()  # 消耗 ':', current\_token 现在是 Type 的开始

            print(f"DEBUG: \_parse\_let\_statement: 类型开始, current={self.current\_token}")

            type\_node = self.\_parse\_type()

            if type\_node is None: return None

            print(f"DEBUG: \_parse\_let\_statement: \_parse\_type 后, current={self.current\_token} (应为类型最后Token)")

            self.\_next\_token()  # 消耗 Type 的最后一个 token

            print(f"DEBUG: \_parse\_let\_statement: 消耗类型后, current={self.current\_token}")

        else:

            self.\_next\_token()  # 消耗 IDENT (如果没有类型注解)

            print(f"DEBUG: \_parse\_let\_statement: 无类型, 消耗 IDENT 后, current={self.current\_token}")

        # 4. 处理可选的初始化器 '= Expression'

        value\_node = None

        print(f"DEBUG: \_parse\_let\_statement: 检查赋值, current={self.current\_token}")

        if self.\_current\_token\_is(TokenType.ASSIGN):

            assign\_token = self.current\_token

            self.\_next\_token()  # 消耗 '=', current\_token 现在是 Expression 的开始

            print(f"DEBUG: \_parse\_let\_statement: 赋值表达式开始, current={self.current\_token}")

            value\_node = self.parse\_expression(LOWEST)

            if value\_node is None: return None

            print(

                f"DEBUG: \_parse\_let\_statement: parse\_expression 后, current={self.current\_token} (应为表达式最后Token)")

            self.\_next\_token()  # 消耗 Expression 的最后一个 token

            print(f"DEBUG: \_parse\_let\_statement: 消耗表达式后, current={self.current\_token}")

        # 5. 期望并消耗分号 ';'

        print(f"DEBUG: \_parse\_let\_statement: 检查分号, current={self.current\_token}")

        if not self.\_current\_token\_is(TokenType.SEMICOLON):

            msg = f"语法错误 (L{self.current\_token.line} C{self.current\_token.col}): 'let' 语句期望以 ';' 结束, 得到 {self.current\_token.type.name} ('{self.current\_token.literal}')"

            self.errors.append(msg)

            print(f"DEBUG: \_parse\_let\_statement: 分号错误!")

            return None

        self.\_next\_token()  # 消耗 ';'

        print(f"DEBUG: \_parse\_let\_statement: 消耗分号后, current={self.current\_token}")

        print(f"DEBUG: \_parse\_let\_statement: 结束")

        return LetStatement(token=let\_token, name=name\_node, value=value\_node, type\_info=type\_node, mutable=is\_mut)

<赋值语句> -> <可赋值元素> '=' <表达式> ';'(2.2, 结合0.3, 3.1)

<可赋值元素> -> <ID>(0.3)

对应函数:\_parse\_assignment\_statement()

运作描述:

1. 当\_parse\_statement()检测到IDENT后跟ASSIGN(=) 时调用。

2. current\_token已经是作为左值的标识符。调用\_parse\_identifier()创建Identifier节点作为target\_node。（注意：\_parse\_identifier通常不消耗token，只创建节点。此函数后续会消耗它）。

3. 消耗标识符，使current\_token变为=。

4. 消耗=。

5. 调用self.parse\_expression(LOWEST)解析右侧的表达式 (value\_node)。

6. 消耗表达式的最后一个词法单元。

7. 期望并消耗分号;。

8. 构建并返回AssignmentStatementAST节点。

    def \_parse\_assignment\_statement(self) -> Statement | None:

        # 假设调用时 current\_token 是 IDENT (Assignable 的开头)

        # 并且我们已经预读到下一个 Token 是 ASSIGN

        # 解析可赋值的目标 (目前只处理 Identifier)

        # current\_token 已经是 IDENT

        target\_node = self.\_parse\_identifier()  # 返回 Identifier 节点

        if target\_node is None:

            # 如果 \_parse\_identifier 返回 None，说明有错误

            return None

        # 期望并消耗 '='

        # \_parse\_identifier 应该没有消耗 IDENT，所以 current\_token 还是 IDENT

        # 我们需要消耗 IDENT，然后期望 ASSIGN

        self.\_next\_token()  # 消耗 IDENT，current\_token 现在是 ASSIGN

        if not self.\_current\_token\_is(TokenType.ASSIGN):

            # 这个检查理论上不应该失败，因为 \_parse\_statement 已经预读过了

            msg = f"内部错误: \_parse\_assignment\_statement 在非 '=' Token 调用 (L{self.current\_token.line})"

            self.errors.append(msg)

            return None

        assign\_token = self.current\_token  # 保存 '=' token

        self.\_next\_token()  # 消耗 '=', current\_token 现在是 Expression 的开始

        # 解析赋值号右侧的表达式

        value\_node = self.parse\_expression(LOWEST)  # 解析 Expression，current\_token 停在 Expression 最后

        if value\_node is None: return None

        # 消耗表达式的最后一个 token

        self.\_next\_token()  # current\_token 现在是表达式之后

        # 期望并消耗分号 ';'

        if not self.\_current\_token\_is(TokenType.SEMICOLON):

            self.\_peek\_error(TokenType.SEMICOLON)  # 报告期望分号

            return None

        self.\_next\_token()  # 消耗 ';'

        # 返回 AssignmentStatement 节点

        # 你可能需要在 ast\_nodes.py 中添加 AssignmentStatement 节点定义

        # class AssignmentStatement(Statement): ...

        # 并在 ast\_printer.py 中添加对应的 visit 方法

        return AssignmentStatement(token=target\_node.token, target=target\_node, value=value\_node)  # 返回节点

<变量声明赋值语句> -> let [mut] <ID> [':' <类型>] '=' <表达式> ';'(2.3)

对应函数:\_parse\_let\_statement()(处理带赋值的部分)

运作描述: 结合了上述变量声明和赋值的逻辑。在解析完可选的类型后，如果遇到=，则继续解析赋值表达式，并将结果存入LetStatement的value成员。

1. **表达式 (规则3.1, 3.2, 3.3)**

<语句> -> <表达式> ';'(表达式语句) (3.1)

对应函数:\_parse\_expression\_statement()

运作描述:

1. 调用self.parse\_expression(LOWEST)解析表达式。

2. parse\_expression返回后，current\_token在表达式的最后一个词法单元。

3. 期望peek\_token是分号;。报错或继续。

4. 消耗表达式的最后一个词法单元，然后消耗分号。

5. 构建并返回ExpressionStatementAST节点。

def \_parse\_expression\_statement(self) -> ExpressionStatement | None:

        start\_token = self.current\_token

        expression\_node = self.parse\_expression(LOWEST)

        if expression\_node is None:

            # 错误已由 parse\_expression 或其子调用记录

            return None

        # parse\_expression 将 current\_token 停在表达式的最后一个 token。

        # 我们需要检查 \*下一个\* token 是否是分号。

        if not self.\_peek\_token\_is(TokenType.SEMICOLON):

            self.\_peek\_error(TokenType.SEMICOLON)  # 记录错误：期望分号

            return None

        self.\_next\_token()  # 消耗表达式的最后一个 token

        self.\_next\_token()  # 消耗 ';' (现在 current\_token 指向 ';' 之后)

        return ExpressionStatement(token=start\_token, expression=expression\_node)

基本表达式、算术与比较运算 (3.1, 3.2)

<表达式> -> <加法表达式>,<加法表达式> -> <项>,<项> -> <因子>,<因子> -> <因子>,<元素> -> <NUM> | <可赋值元素> | '('<表达式>')'

增加运算：<表达式> -> <表达式><比较运算符><加法表达式>, etc.

对应函数:parse\_expression(precedence)及其相关的前缀解析函数 (如\_parse\_identifier,\_parse\_integer\_literal,\_parse\_prefix\_expression,\_parse\_grouped\_expression\_or\_tuple) 和中缀解析函数 (\_parse\_infix\_expression)。

运作描述:

\_parse\_identifier(): 当表达式以标识符开头时被调用，创建Identifier节点。

\_parse\_integer\_literal(): 当表达式以数字开头时被调用，创建IntegerLiteral节点。

\_parse\_prefix\_expression(): 当遇到前缀操作符（如-）时，消耗操作符，递归调用parse\_expression(PREFIX)解析右操作数，并创建PrefixExpression节点。

\_parse\_grouped\_expression\_or\_tuple(): 当遇到左括号(时，消耗(，递归调用parse\_expression(LOWEST)解析括号内的表达式，然后期望并消耗右括号).

\_parse\_infix\_expression(left\_expr): 当遇到中缀操作符时，left\_expr是已解析的左操作数。此函数消耗操作符，记录操作符和其优先级，递归调用parse\_expression(operator\_precedence)解析右操作数，然后创建InfixExpression节点。

parse\_expression通过调度这些前缀和中缀函数，并利用PRECEDENCES表来正确处理操作符优先级和结合性。

def parse\_expression(self, precedence: int) -> Expression | None:

        prefix\_fn = self.prefix\_parse\_fns.get(self.current\_token.type)

        if prefix\_fn is None: self.\_no\_prefix\_parse\_fn\_error(self.current\_token.type); return None

        left\_expr = prefix\_fn() # Returns Node

        if left\_expr is None: return None

        while not self.\_peek\_token\_is(TokenType.SEMICOLON) and precedence < self.\_peek\_precedence(): # Review terminators if needed

            infix = self.infix\_parse\_fns.get(self.peek\_token.type)

            if infix is None: return left\_expr

            self.\_next\_token() # Assume original loop advanced operator here

            left\_expr = infix(left\_expr) # infix returns updated node

            if left\_expr is None: return None

        return left\_expr

    def \_parse\_identifier(self) -> Identifier | None:

        return Identifier(token=self.current\_token, value=self.current\_token.literal)

    def \_parse\_integer\_literal(self) -> IntegerLiteral | None:

        try: value = int(self.current\_token.literal)

        except ValueError: msg = f"语法错误 (L{self.current\_token.line} C{self.current\_token.col}): 无法将 '{self.current\_token.literal}' 转换为整数"; self.errors.append(msg); return None

        return IntegerLiteral(token=self.current\_token, value=value)

    def \_parse\_prefix\_expression(self) -> PrefixExpression | None:

        operator\_token = self.current\_token; operator = operator\_token.literal

        self.\_next\_token()

        right\_node = self.parse\_expression(PREFIX) # Returns Node

        if right\_node is None: return None

        return PrefixExpression(token=operator\_token, operator=operator, right=right\_node)

    def \_parse\_grouped\_expression\_or\_tuple(self) -> Expression | None:

         self.\_next\_token() # Assume consumes '('

         # TODO: Add tuple parsing if original had it

         expr\_node = self.parse\_expression(LOWEST) # Returns Node

         if expr\_node is None: return None

         if not self.\_expect\_peek(TokenType.RPAREN): return None # Assume consumes ')'

         return expr\_node

    def \_parse\_if\_expression(self) -> IfStatement | None: # Treat 'if' expr like statement for now

         return self.\_parse\_if\_statement()

    def \_parse\_infix\_expression(self, left\_expr: Expression) -> InfixExpression | None:

        # left\_expr is node, current\_token is operator

        infix\_token = self.current\_token; operator = infix\_token.literal

        precedence = self.\_current\_precedence()

        self.\_next\_token() # Assume original consumed operator here

        right\_expr = self.parse\_expression(precedence) # Returns Node

        if right\_expr is None: return None

        return InfixExpression(token=infix\_token, left=left\_expr, operator=operator, right=right\_expr)

函数调用<元素> -> <ID> '(' <实参列表> ')'(3.3)

<实参列表> -> 空 | <表达式> | <表达式> ',' <实参列表>(3.3)

对应函数:

中缀解析函数\_parse\_call\_expression(function\_expr)(当LPAREN作为中缀操作符时被parse\_expression调用)。function\_expr是代表函数名的Identifier或其他可调用表达式。

辅助函数\_parse\_call\_arguments()。

运作描述:

1. \_parse\_call\_expression被调用时，current\_token是左括号(.

2. 它调用\_parse\_call\_arguments()来解析括号内的参数。

3. \_parse\_call\_arguments()：

如果预读到右括号)，则为空参数列表。

否则，循环解析参数：调用self.parse\_expression(LOWEST)解析每个参数表达式。如果预读到逗号,，则消耗并继续；如果预读到)，则结束。

此函数结束时，current\_token会停留在右括号)上。

4. \_parse\_call\_expression确认current\_token是)(由\_parse\_call\_arguments保证)，然后构建并返回CallExpressionAST节点。Pratt解析器的主循环之后会处理这个).

def \_parse\_call\_expression(self, function\_expr: Expression) -> CallExpression | None:

        # function\_expr 是函数名/表达式节点

        # current\_token 是 '('

        call\_lparen\_token = self.current\_token  # 保存 '(' token

        arguments = self.\_parse\_call\_arguments() # 这个方法现在负责消耗到 ')' (并停在 ')' 上)

        if arguments is None: return None

        # \_parse\_call\_arguments 结束后，current\_token 应该已经是 ')' 了

        if not self.\_current\_token\_is(TokenType.RPAREN):

            # 内部错误

            # ...

            return None

        # 返回 CallExpression 节点。current\_token 保持在 ')' 上。

        # Pratt 解析器的主循环会在之后处理这个 ')' (比如，如果后面没有更高优先级的操作符，

        # parse\_expression 就会返回，此时 current\_token 依然是 ')')

        return CallExpression(token=call\_lparen\_token, function=function\_expr, arguments=arguments)

    def \_parse\_call\_arguments(self) -> list[Expression] | None:

        args = []

        # current\_token 应该是 '('

        if not self.\_current\_token\_is(TokenType.LPAREN):

             # ... 错误处理 ...

             return None

        # 处理空参数列表: call()

        if self.\_peek\_token\_is(TokenType.RPAREN):

            self.\_next\_token()  # 消耗 '(', current\_token 现在是 ')'

            return args # 成功, current\_token 是 ')'

        self.\_next\_token()  # 消耗 '(', current\_token 现在是第一个参数的开始

        # 解析第一个参数

        first\_arg\_node = self.parse\_expression(LOWEST)

        if first\_arg\_node is None: return None

        args.append(first\_arg\_node)

        # current\_token 现在是第一个参数的最后一个 token

        # 循环处理后续参数

        while self.\_peek\_token\_is(TokenType.COMMA):

            self.\_next\_token()  # 消耗上一个参数的最后一个 token

            self.\_next\_token()  # 消耗 ','

            # ... 检查逗号后不能直接跟 ')' ...

            arg\_node = self.parse\_expression(LOWEST)

            if arg\_node is None: return None

            args.append(arg\_node)

            # current\_token 现在是当前参数的最后一个 token

        # 循环结束后，current\_token 停留在最后一个参数的最后一个 token

        # 期望下一个 token 是 ')'

        if not self.\_peek\_token\_is(TokenType.RPAREN):

            self.\_peek\_error(TokenType.RPAREN)

            return None

        self.\_next\_token()  # 消耗最后一个参数的最后一个 token，使得 current\_token 现在是 ')'

        return args # 成功, current\_token 是 ')'

1. **选择结构 (规则4.1)**

<语句> -> <if语句>(4.1)

<if语句> -> if <表达式> <语句块> [<else部分>](4.1)

<else部分> -> 空 | else <语句块> | else if ...(4.1, 4.2 - 但基础只含else <语句块>)

对应函数:\_parse\_if\_statement()

运作描述:

1. 当\_parse\_statement()检测到if关键字时调用。

2. 消耗if。

3. 调用self.parse\_expression(LOWEST)解析条件表达式。

4. 期望peek\_token是左花括号{，消耗条件表达式的最后一个token，使{成为current\_token。

5. 调用\_parse\_block\_statement()解析then语句块（consequence）。

6. 消耗then块后的}。

7. 检查current\_token是否为else。

如果是else：消耗else。

如果接下来是if(构成else if)，则递归调用\_parse\_if\_statement()解析alternative。

如果接下来是{，则调用\_parse\_block\_statement()解析else语句块作为alternative，然后消耗其后的}。

否则报错。

如果没有else，则alternative为None。

8. 构建并返回IfStatementAST节点。

def \_parse\_if\_statement(self) -> IfStatement | None:

        if\_token = self.current\_token

        self.\_next\_token()

        condition = self.parse\_expression(LOWEST) # Returns Node

        if condition is None: return None

        # Assume original logic advanced correctly before checking '{'

        if not self.\_peek\_token\_is(TokenType.LBRACE): self.\_peek\_error(TokenType.LBRACE); return None

        self.\_next\_token() # Consume condition last token

        # Current is '{'

        consequence = self.\_parse\_block\_statement() # Returns Node, leaves current on '}'

        if consequence is None: return None

        self.\_next\_token() # Consume '}'

        alternative = None

        if self.\_current\_token\_is(TokenType.ELSE):

            self.\_next\_token()

            if self.\_current\_token\_is(TokenType.IF): alternative = self.\_parse\_if\_statement() # Returns Node

            elif self.\_current\_token\_is(TokenType.LBRACE):

                 alternative = self.\_parse\_block\_statement() # Returns Node, leaves current on '}'

                 if alternative is None: return None

                 self.\_next\_token() # Consume '}'

            else: msg = f"语法错误 (L{self.current\_token.line} C{self.current\_token.col}): 'else' 后期望 'if' 或 '{{', 得到 {self.current\_token.type.name} ('{self.current\_token.literal}')"; self.errors.append(msg); return None

            if alternative is None and self.errors: return None

        return IfStatement(token=if\_token, condition=condition, consequence=consequence, alternative=alternative)

1. **循环结构 (规则5.1)**

<语句> -> <循环语句>(5.1)

<循环语句> -> <while语句>(5.1)

<while语句> -> while <表达式> <语句块>(5.1)

对应函数:\_parse\_while\_statement()

运作描述:

1. 当\_parse\_statement()检测到while关键字时调用。

2. 消耗while。

3. 调用self.parse\_expression(LOWEST)解析条件表达式。

4. 期望peek\_token是左花括号{，消耗条件表达式的最后一个token，使{成为current\_token。

5. 调用\_parse\_block\_statement()解析循环体。

6. 消耗循环体后的}。

7. 构建并返回WhileStatementAST节点。

def \_parse\_while\_statement(self) -> WhileStatement | None:

        """解析 while 语句: while Expression BlockStatement"""

        while\_token = self.current\_token  # 'while'

        self.\_next\_token()  # 消耗 'while'

        # 解析条件表达式

        condition\_node = self.parse\_expression(LOWEST)

        if condition\_node is None: return None

        # parse\_expression 之后，current\_token 应该是表达式的最后一个 token

        if not self.\_peek\_token\_is(TokenType.LBRACE):

            self.\_peek\_error(TokenType.LBRACE)

            return None

        self.\_next\_token()  # 消耗表达式的最后一个 token，现在 current\_token 是 '{'

        # 解析循环体

        body\_node = self.\_parse\_block\_statement()  # 这个方法内部会消耗 '{' 和 '}'

        if body\_node is None: return None

        # \_parse\_block\_statement 应该将 current\_token 停留在 '}' 上

        if not self.\_current\_token\_is(TokenType.RBRACE):

            # 如果 \_parse\_block\_statement 实现正确，这里不应该发生

            msg = f"内部错误: \_parse\_block\_statement 未在 '}}' 处停止 (L{self.current\_token.line})"

            self.errors.append(msg)

            return None

        self.\_next\_token()  # 消耗 '}'

        return WhileStatement(token=while\_token, condition=condition\_node, body=body\_node)

1. **拓展功能实现**

拓展功能 5.3:loop循环结构

<循环语句> -> <loop语句>

<loop语句> -> loop <语句块>

对应函数:\_parse\_loop\_statement()

运作描述:

1. 当\_parse\_statement()检测到loop关键字时调用。

2. 消耗loop。

3. 期望current\_token是左花括号{。

4. 调用\_parse\_block\_statement()解析循环体。

5. 消耗循环体后的}。

6. 构建并返回LoopStatementAST节点。

def \_parse\_loop\_statement(self) -> LoopStatement | None:

        loop\_token = self.current\_token  # 'loop' token

        self.\_next\_token() # Consume 'loop'. Current token should now be '{'.

        if not self.\_current\_token\_is(TokenType.LBRACE):

            msg = f"语法错误 (L{self.current\_token.line} C{self.current\_token.col}): 'loop' 语句体期望以 '{{' 开始, 得到 {self.current\_token.type.name} ('{self.current\_token.literal}')"

            self.errors.append(msg)

            return None

        # current\_token is LBRACE, \_parse\_block\_statement expects this

        body\_node = self.\_parse\_block\_statement()

        if body\_node is None:

            return None # Error already reported by \_parse\_block\_statement or its children

        # \_parse\_block\_statement leaves current\_token on RBRACE

        self.\_next\_token() # Consume RBRACE

        return LoopStatement(token=loop\_token, body=body\_node)

拓展功能 5.4:break和continue语句

<语句> -> break ';' | continue ';'

对应函数:\_parse\_break\_statement()和\_parse\_continue\_statement()

运作描述 (\_parse\_break\_statement):

1. 当\_parse\_statement()检测到break关键字时调用。

2. 记录并消耗break词法单元。

3. 期望current\_token是分号;，并消耗它。

4. 构建并返回BreakStatementAST节点。

\_parse\_continue\_statement的逻辑与此完全相同，只是关键字和返回的AST节点类型为ContinueStatement。

def \_parse\_break\_statement(self) -> BreakStatement | None:

        break\_token = self.current\_token  # 'break' token

        self.\_next\_token() # Consume 'break'

        if not self.\_current\_token\_is(TokenType.SEMICOLON):

            msg = f"语法错误 (L{self.current\_token.line} C{self.current\_token.col}): 'break' 语句期望以 ';' 结束, 得到 {self.current\_token.type.name} ('{self.current\_token.literal}')"

            self.errors.append(msg)

            return None

        self.\_next\_token() # Consume ';'

        return BreakStatement(token=break\_token)

    def \_parse\_continue\_statement(self) -> ContinueStatement | None:

        continue\_token = self.current\_token # 'continue' token

        self.\_next\_token() # Consume 'continue'

        if not self.\_current\_token\_is(TokenType.SEMICOLON):

            msg = f"语法错误 (L{self.current\_token.line} C{self.current\_token.col}): 'continue' 语句期望以 ';' 结束, 得到 {self.current\_token.type.name} ('{self.current\_token.literal}')"

            self.errors.append(msg)

            return None

        self.\_next\_token() # Consume ';'

        return ContinueStatement(token=continue\_token)

拓展功能 6.1: 声明不可变变量

<变量声明内部> -> <ID>(相较于基础的mut <ID>)

对应函数:\_parse\_let\_statement()

运作描述: 此功能体现在\_parse\_let\_statement解析let语句时对mut关键字的处理。在消耗let关键字后，函数会检查peek\_token是否为mut。如果是mut，则消耗mut，并将传递给LetStatement构造函数的mutable参数设为True；如果不是mut（而是直接跟着标识符），则不消耗mut（因为它不存在），并将传递给LetStatement构造函数的mutable参数设为False（或默认为False）。后续解析标识符、可选类型、可选赋值和分号的逻辑不变。这样，LetStatement节点就携带了变量是否可变的信息。词法分析阶段IDENT已经是基础词法单元，语法分析在此基础上判断上下文。

# 2. 处理 IDENT (如果 'mut' 不存在或已处理)

        elif self.\_peek\_token\_is(TokenType.IDENT):

            self.\_next\_token()  # 消耗 'let', current\_token 现在是 IDENT

            print(f"DEBUG: \_parse\_let\_statement: 'let' 之后, current={self.current\_token} (应为 IDENT)")

        else:

            self.\_peek\_error(TokenType.IDENT)

            return None

## 3、可视化设计

在gui.py中，我们使用tkinter作为GUI的框架，对rust代码提供词法分析、语法分析以语法树的可视化展示。

1. 整体页面设计



界面使用 PanedWindow 分为左右两部分：

1. 左侧：代码输入区与操作按钮。

左侧面板的输入区域为一个可滚动的文本框 (ScrolledText)，用于输入 Rust 代码。同时还有操作按钮：

* "进行词法分析"：触发词法分析，显示 token 列表。
* "进行语法分析"：触发语法分析，显示错误信息和语法结构。
* "生成 AST 图像"：生成并显示 AST 的图像。

1. 右侧：结果展示区，包含词法分析、语法分析和 AST 图像三个标签页。

右侧面板为结果显示，包括：

* 词法分析结果：显示 token 列表，包括行号、列号、token 类型和字面值。
* 语法分析结果：
  + 错误信息：如果有语法错误，显示错误列表。
  + 语法结构：如果分析成功，显示 AST 的文本表示。
* AST 图像：显示 AST 的图像，支持缩放功能。

1. 重点功能实现函数解析
2. **主界面初始化**

功能：LexerApp.\_\_init\_\_方法负责初始化整个 GUI 的布局、控件和初始状态，确保程序启动时界面结构完整且功能可用，为后续交互功能奠定了基础。预置代码和清晰的标签页设计降低了交互难度，而滚动条和缩放控件的预配置确保了复杂输出内容的可视化支持。

实现：

* 窗口设置：设置主窗口标题为“类 Rust 词法/语法分析器 + AST”，尺寸为1000x700像素，提供足够空间展示输入和输出区域。
* 分栏布局：使用PanedWindow创建水平分栏，左侧宽度初始为450像素，右侧为 550 像素，支持用户拖动调整。
* 左侧输入区：
  + 包含标题标签（“输入类 Rust 代码:”）和ScrolledText输入框，支持代码输入、滚动和撤销操作。
  + 预置示例rust代码，便于用户快速测试。
  + 添加三个按钮（“进行词法分析”、“进行语法分析”、“生成 AST 图像”），分别绑定 analyze\_code、parse\_syntax和show\_ast方法，布局紧凑且功能明确。
* 右侧结果区：使用ttk.Notebook创建三个标签页：
  + 词法分析结果：包含标签和ScrolledText，用于显示token列表，初始禁用以防止编辑。
  + 语法分析结果：包含错误信息和语法结构两个ScrolledText区域，分别显示解析错误和语法结构文本表示。
* AST图像：包含Canvas用于显示AST图像，配有垂直和水平滚动条（Scrollbar）支持大图像浏览。添加“放大 +”和“缩小 -”按钮，绑定zoom\_in\_ast和zoom\_out\_ast方法。
* 图像区域初始化：为AST图像设置Canvas和Label，通过create\_window居中显示图像，并绑定<Configure>事件动态更新滚动区域。

class LexerApp:

    def \_\_init\_\_(self, master):

        self.master = master

        master.title("类 Rust 词法/语法分析器 + AST")

        master.geometry("1000x700") # Increased size

        default\_font = tkFont.nametofont("TkDefaultFont")

        default\_font.configure(size=11)

        text\_font = tkFont.Font(family="Consolas", size=11)

        # --- PanedWindow (Unchanged) ---

        self.m = PanedWindow(master, orient=tk.HORIZONTAL, sashrelief=tk.RAISED, sashwidth=5)

        self.m.pack(fill=tk.BOTH, expand=True, padx=5, pady=5)

        # --- Left Frame (Input & Buttons) (Unchanged structure) ---

        left\_frame = tk.Frame(self.m, bd=2, relief=tk.SUNKEN)

        self.m.add(left\_frame, width=450) # Adjust initial width maybe

        input\_label = tk.Label(left\_frame, text="输入类 Rust 代码:", font=default\_font)

        input\_label.pack(pady=(0, 5), fill=tk.X)

        self.input\_text = scrolledtext.ScrolledText(left\_frame, wrap=tk.WORD, undo=True, font=text\_font, height=20)

        self.input\_text.pack(fill=tk.BOTH, expand=True)

        # ... (insert initial code) ...

        self.input\_text.insert(tk.END,

        """//输入rust代码

fn factorial(n: i32) -> i32 {

    // This is a comment

    let mut result: i32 = 1;

    let mut i: i32 = 1; /\* Another

       comment style \*/

    while i <= n {

        result = result \* i; // calculation

        i = i + 1;

    }

    return result;

}

fn main() {

    let x: i32 = 5;

    let fact\_x: i32;

    fact\_x = factorial(x);

    if x >= 1 {

        loop { break; }

    }

}

    """)

        analyze\_button = tk.Button(left\_frame, text="进行词法分析", command=self.analyze\_code, font=default\_font, pady=5)

        analyze\_button.pack(pady=(10,2), fill=tk.X)

        parse\_button = tk.Button(left\_frame, text="进行语法分析", command=self.parse\_syntax, font=default\_font, pady=5)

        parse\_button.pack(pady=(2, 2), fill=tk.X)

        ast\_button = tk.Button(left\_frame, text="生成 AST 图像", command=self.show\_ast, font=default\_font, pady=5)

        ast\_button.pack(pady=(2, 5), fill=tk.X)

        # --- Right Frame (Outputs - Use Notebook for Tabs) ---

        right\_notebook = ttk.Notebook(self.m)

        self.m.add(right\_notebook, width=550) # Adjust initial width

        # -- Lexer Output Tab --

        lexer\_tab = tk.Frame(right\_notebook)

        right\_notebook.add(lexer\_tab, text='词法分析结果')

        lexer\_output\_label = tk.Label(lexer\_tab, text="词法分析结果 (Tokens):", font=default\_font)

        lexer\_output\_label.pack(pady=(5, 5), fill=tk.X, padx=5)

        self.lexer\_output\_text = scrolledtext.ScrolledText(lexer\_tab, wrap=tk.NONE, font=text\_font, state=tk.DISABLED, height=20)

        self.lexer\_output\_text.pack(fill=tk.BOTH, expand=True, padx=5, pady=(0,5))

        # -- Parser Output Tab --

        parser\_tab = tk.Frame(right\_notebook)

        right\_notebook.add(parser\_tab, text='语法分析结果')

        parser\_error\_label = tk.Label(parser\_tab, text="语法分析错误:", font=default\_font)

        parser\_error\_label.pack(pady=(5, 5), fill=tk.X, padx=5)

        self.parser\_error\_text = scrolledtext.ScrolledText(parser\_tab, wrap=tk.NONE, font=text\_font, state=tk.DISABLED, height=10)

        self.parser\_error\_text.pack(fill=tk.BOTH, expand=True, padx=5, pady=(0,5))

        parser\_struct\_label = tk.Label(parser\_tab, text="语法结构:", font=default\_font)

        parser\_struct\_label.pack(pady=(10, 5), fill=tk.X, padx=5)

        self.parser\_struct\_text = scrolledtext.ScrolledText(parser\_tab, wrap=tk.NONE, font=text\_font, state=tk.DISABLED, height=20)

        self.parser\_struct\_text.pack(fill=tk.BOTH, expand=True, padx=5, pady=(0,5))

        # -- AST Output Tab --

        ast\_tab = tk.Frame(right\_notebook)

        right\_notebook.add(ast\_tab, text='AST 图像')

        ast\_output\_label = tk.Label(ast\_tab, text="AST 图像:", font=default\_font)

        ast\_output\_label.pack(pady=(5, 5), fill=tk.X, padx=5)

        # --- AST 图片区域（带滚动条）---

        ast\_frame = tk.Frame(ast\_tab)

        ast\_frame.pack(fill=tk.BOTH, expand=True, padx=5, pady=5)

        self.ast\_canvas = tk.Canvas(ast\_frame, bg="white")

        self.ast\_scrollbar\_y = tk.Scrollbar(ast\_frame, orient=tk.VERTICAL, command=self.ast\_canvas.yview)

        self.ast\_scrollbar\_x = tk.Scrollbar(ast\_frame, orient=tk.HORIZONTAL, command=self.ast\_canvas.xview)

        self.ast\_canvas.configure(yscrollcommand=self.ast\_scrollbar\_y.set, xscrollcommand=self.ast\_scrollbar\_x.set)

        self.ast\_scrollbar\_y.pack(side=tk.RIGHT, fill=tk.Y)

        self.ast\_scrollbar\_x.pack(side=tk.BOTTOM, fill=tk.X)

        self.ast\_canvas.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)

        self.ast\_image\_label = tk.Label(self.ast\_canvas, bg="white")

        self.ast\_canvas.create\_window((0, 0), window=self.ast\_image\_label, anchor='center')

        self.ast\_image\_label.bind('<Configure>',

                                  lambda e: self.ast\_canvas.configure(scrollregion=self.ast\_canvas.bbox("all")))

        # --- AST 图片缩放控制 ---

        zoom\_frame = tk.Frame(ast\_tab)

        zoom\_frame.pack(pady=(5, 5))

        self.zoom\_in\_button = tk.Button(zoom\_frame, text="放大 +", command=self.zoom\_in\_ast)

        self.zoom\_in\_button.pack(side=tk.LEFT, padx=5)

        self.zoom\_out\_button = tk.Button(zoom\_frame, text="缩小 -", command=self.zoom\_out\_ast)

        self.zoom\_out\_button.pack(side=tk.LEFT, padx=5)

        self.ast\_image\_original = None  # 保存原图

        self.ast\_zoom\_ratio = 1.0  # 当前缩放比例

1. **词法分析结果展示**

功能：从输入框获取代码，调用lexer,py中的Lexer类进行词法分析，并将生成的 token 列表显示在右侧的“词法分析结果”标签页中，可以直观呈现代码的词法分解过程。

实现：

* 使用 ScrolledText 控件以可滚动文本形式展示 token，格式为 行号 | 列号 | 类型 | 字面值。
* 若遇到非法字符或异常，添加错误提示（如 \*\*\* 错误: 非法字符 'x' \*\*\*），通过红色高亮或文本标注增强可读性。

def analyze\_code(self):

        # --- Lexer analysis logic (Unchanged) ---

        source\_code = self.input\_text.get("1.0", tk.END)

        self.lexer\_output\_text.config(state=tk.NORMAL)

        self.lexer\_output\_text.delete("1.0", tk.END)

        if not source\_code.strip():

            self.lexer\_output\_text.insert(tk.END, "请输入代码后再进行分析。\n")

            self.lexer\_output\_text.config(state=tk.DISABLED)

            return

        lexer = Lexer(source\_code)

        output\_lines = []

        error\_found = False

        try:

            while True:

                tok = lexer.next\_token()

                line\_str = f"L{tok.line:<3} C{tok.col:<3}: {tok.type.name:<10s} | '{tok.literal}'"

                output\_lines.append(line\_str)

                if tok.type == TokenType.EOF: break

                if tok.type == TokenType.ILLEGAL:

                    output\_lines.append(f"\*\*\* 错误: 非法字符 '{tok.literal}' \*\*\*"); error\_found = True

        except Exception as e:

             output\_lines.append(f"\n\*\*\* 词法分析时发生意外错误: {e} \*\*\*"); error\_found = True

        self.lexer\_output\_text.insert(tk.END, "\n".join(output\_lines))

        if error\_found: self.lexer\_output\_text.insert(tk.END, "\n\n分析过程中检测到错误。")

        self.lexer\_output\_text.config(state=tk.DISABLED)

1. **语法分析与结构展示**

功能：基于词法分析结果，调用parser.py中的Parser类进行语法分析，再调用Parser.parse\_program()生成语法结构，并在右侧“语法分析结果”标签页中显示错误信息和 语法结构，从而将复杂的语法分析过程分为错误提示和结构展示两部分。

实现：

* 错误信息：通过 parser.errors 检查语法错误，若存在，则在 parser\_error\_text 中列出详细错误（如语法错误 (L15 C2): 期望 RBRACE, 得到 EOF ('')）。
* 语法结构：若分析成功，使用ASTPrinter遍历AST，将层次化的语法结构文本表示输出到parser\_struct\_text。
* 两个输出区域均使用 ScrolledText，支持滚动查看长内容。

def parse\_syntax(self):

        """获取输入代码，执行语法分析，生成并显示 AST"""

        source\_code = self.input\_text.get("1.0", tk.END)

        # Clear previous outputs

        self.parser\_error\_text.config(state=tk.NORMAL)

        self.parser\_error\_text.delete("1.0", tk.END)

        self.parser\_struct\_text.config(state=tk.NORMAL)

        self.parser\_struct\_text.delete("1.0", tk.END)

        if not source\_code.strip():

            self.parser\_error\_text.insert(tk.END, "请输入代码后再进行分析。\n")

            self.parser\_error\_text.config(state=tk.DISABLED)

            self.parser\_struct\_text.config(state=tk.DISABLED)

            return

        # Perform lexing first

        lexer = Lexer(source\_code)

        parser = Parser(lexer) # Create the parser

        # --- NEW: Call parse\_program to get AST root ---

        ast\_root = parser.parse\_program() # Returns Program node or None

        # --- Display Errors ---

        if parser.errors:

            self.parser\_error\_text.insert(tk.END, "语法分析失败！检测到以下错误:\n")

            self.parser\_error\_text.insert(tk.END, "-----------------------------------\n")

            for error in parser.errors:

                self.parser\_error\_text.insert(tk.END, f"- {error}\n")

            self.parser\_error\_text.insert(tk.END, "-----------------------------------\n")

            self.parser\_struct\_text.insert(tk.END, "由于语法错误，无法生成语法结构。\n")

        elif ast\_root is None:

             self.parser\_error\_text.insert(tk.END, "语法分析未能生成有效的程序结构 (可能是空输入或内部错误)。\n")

             self.parser\_struct\_text.insert(tk.END, "无法生成语法结构。\n")

        else:

             # --- Parsing successful ---

             self.parser\_error\_text.insert(tk.END, "语法分析成功！未检测到明显语法错误。\n")

             # --- NEW: Print AST using Visitor ---

             try:

                 printer = ASTPrinter()

                 ast\_root.accept(printer) # Traverse AST and collect output

                 ast\_string = printer.get\_output() # Get collected output

                 self.parser\_struct\_text.insert(tk.END, ast\_string)

             except Exception as e:

                  self.parser\_struct\_text.insert(tk.END, f"生成语法结构可视化时发生错误: {e}\n")

                  import traceback

                  self.parser\_struct\_text.insert(tk.END, traceback.format\_exc())

        self.parser\_error\_text.config(state=tk.DISABLED)

        self.parser\_struct\_text.config(state=tk.DISABLED)

1. **AST图像生成**

功能：调用ASTGraphvizDrawer类生成AST的图像表示，并通过render\_to\_memory 将其转换为 PNG 格式，显示在“AST 图像”标签页，将抽象的AST以图形化形式呈现。

实现：

* 使用 graphviz.Digraph 构建树状图，每个节点标注类型和值（如 Let: result），边标注关系（如 value、condition）。
* 若语法分析失败，弹出 messagebox.showerror 提示用户（如 “由于语法错误，无法生成 AST”）。

def display\_ast\_image(self):

        if self.ast\_image\_original is None:

            return

        # 计算新的尺寸

        width, height = self.ast\_image\_original.size

        new\_size = (int(width \* self.ast\_zoom\_ratio), int(height \* self.ast\_zoom\_ratio))

        resized\_image = self.ast\_image\_original.resize(new\_size, Image.Resampling.LANCZOS)

        self.ast\_image = ImageTk.PhotoImage(resized\_image)

        self.ast\_image\_label.config(image=self.ast\_image)

        self.ast\_image\_label.image = self.ast\_image

    def zoom\_in\_ast(self):

        self.ast\_zoom\_ratio \*= 1.2  # 每次放大20%

        self.display\_ast\_image()

    def zoom\_out\_ast(self):

        self.ast\_zoom\_ratio /= 1.2  # 每次缩小20%

        self.display\_ast\_image()

    def show\_ast(self):

        source\_code = self.input\_text.get("1.0", tk.END)

        # 清空 AST 图片

        self.ast\_image\_label.config(image='')  # 去掉原来的图

        self.ast\_image\_label.image = None  # 防止引用残留

        if not source\_code.strip():

            messagebox.showinfo("AST生成失败", "请输入代码后再进行分析。\n")

            return

        # 词法分析

        lexer = Lexer(source\_code)

        parser = Parser(lexer)

        # --- 获取 AST 根节点 ---

        ast\_root = parser.parse\_program()

        # --- 如果有语法错误 ---

        if parser.errors:

            messagebox.showerror("AST生成失败","由于语法错误，无法生成 AST。\n")

        elif ast\_root is None:

            messagebox.showerror("AST生成失败","无法生成 AST。\n")

        else:

            # --- 成功 ---

            # --- 新增：生成 AST 树状图 ---

            try:

                drawer = ASTGraphvizDrawer()

                ast\_root.accept(drawer)

                img\_bytes = drawer.render\_to\_memory()

                # 把 bytes 转成 Image

                image = Image.open(io.BytesIO(img\_bytes))

                self.ast\_image\_original = image  # 保存原始图，供缩放用

                self.ast\_zoom\_ratio = 1.0  # 每次分析完，重置缩放比例

                # 初次显示，1.0倍

                self.display\_ast\_image()

            except Exception as e:

                messagebox.showerror("AST生成失败", f"无法生成AST图形，错误信息:\n{e}")

# 四、小组分工

本次报告与PPT由小组成员共同完成，类Rust词法和语法分析工具设计实现分工如下：

[Truman-min-show](https://github.com/Truman-min-show)：主要负责语法分析器设计编写，实现基础语法分析功能。

[TheTry16](https://github.com/TheTry16)：主要负责可视化设计，展示分析结果。

[limeszstone](https://github.com/limeszstone)：主要负责词法分析器设计编写，实现拓展语法分析功能。

# 五、总结和展望

## 1、总结

本次编译原理期中项目，我们小组成功设计并实现了一个针对类Rust语言的编译器，主要包括词法分析器、语法分析器以及AST的可视化展示。通过本次实践，我们不仅将课堂上学习的编译原理理论知识应用于实际，更深刻地理解了编译器设计的基本原理、核心数据结构和关键算法。

语法分析方面，本次实验成功设计并实现了一个针对类 Rust 语言的词法分析(lexer.py)。核心工作围绕以下几点展开：

* **明确词法单元:** 通过 token\_defs.py 文件，使用 Python 的 Enum 清晰地定义了语言的所有词法单元类型 (TokenType)，包括关键字、标识符、整型字面量、各类运算符和分隔符，以及特殊的 EOF 和 ILLEGAL 标记。同时，利用 namedtuple 定义了 Token 结构，用于封装每个词法单元的类型、原始字面量 (literal) 和其在源代码中的精确位置（行号 line、列号 col）。
* **扫描与状态机思想:** lexer.py 中的 Lexer 类实现了核心的扫描逻辑。它通过维护当前字符 (ch)、位置指针 (position, read\_position) 和行列号 (line, col) 的状态，逐个字符地处理输入。其行为可以看作一个隐式的有限状态机，根据当前字符和预读字符 (\_peek\_char()) 决定识别哪种 Token 或进行状态转移（消耗字符）。
* **输出与接口:** Lexer 最终向语法分析器提供了一个清晰、规范的 Token 序列（流），每个 Token 都包含了足够的信息供后续阶段使用，包括错误报告。

在语法分析方面，我们构建了一个能够有效处理类Rust语言核心文法的解析器。该解析器基于token\_defs.py中定义的词法单元类型和ast\_nodes.py中定义的AST节点结构，采用递归下降与Pratt解析相结合的混合策略：

* **核心数据结构：**我们精心设计了AST节点（如Program, LetStatement, FunctionDeclarationStatement, IfStatement, WhileStatement, LoopStatement, InfixExpression等），使其能够清晰表示源代码的层级结构。Parser内部通过维护current\_token、peek\_token、错误列表errors，以及用于Pratt解析的prefix\_parse\_fns、infix\_parse\_fns和PRECEDENCES表，有效地驱动解析过程。
* **解析流程：**从parse\_program()入口开始，通过\_parse\_statement()分发至各具体语句解析函数，这些函数负责消耗词法单元并构建语句级AST节点。当遇到表达式时，则调用parse\_expression()，利用Pratt解析器的能力处理操作符优先级和结合性，构建表达式AST子树。
* **功能覆盖：**我们成功实现了PDF中定义的所有基础语法规则（绿色节点0.1-5.1），并完成了部分拓展规则（蓝色节点），具体包括：

拓展5.3：loop循环结构，通过\_parse\_loop\_statement()实现，支持无条件无限循环。

拓展5.4：break和continue语句，通过\_parse\_break\_statement()和\_parse\_continue\_statement()实现，为循环控制提供了必要的手段。

拓展6.1：声明不可变变量，在\_parse\_let\_statement()中，通过判断mut关键字是否存在来确定变量的可变性，符合Rust默认不可变的特性。

通过本次期中项目，我们不仅巩固了理论知识，锻炼了工程实践能力，也对编译器这一复杂系统有了更深的敬畏和探索的欲望。我们相信，在下次的大作业中，这些经验将为我们理解和构建更复杂的软件系统打下坚实的基础。我们期待有机会能够继续完善这个项目，向着一个功能更完整的编译器迈进。

## 2、展望

尽管本项目已达成预定的目标，并实现了一个功能较为完备的类Rust编译器，但从构建一个更成熟编译器的角度来看，仍有广阔的拓展空间和深入研究的方向：

1. 错误处理与恢复的增强

目前我们的错误报告机制相对简单，主要是在遇到第一个语法错误时停止或记录。未来可以引入更高级的错误恢复策略，使解析器在遇到错误后能尝试同步到下一个有效的语句或声明开头，从而一次性报告多个错误，提升用户体验。

2. 支持更丰富的Rust语法特性

当前拓展功能还有部分未实现，未来可以考虑加入以下功能：

* 完整数组与元组支持：全面实现PDF中规则8（数组）和规则9（元组）的定义、字面量、元素访问和类型解析。
* For循环结构（5.2）：实现for <var> in <iterable> { ... }的语法解析，并考虑后续如何处理可迭代对象。
* 更完善的引用与解引用（6.2）：深化对&、&mut、\*在复杂表达式和类型声明中的解析。

3. 后端开发与代码生成：

在AST和语义分析之后，将AST转换为某种中间表示（IR），如三地址码或LLVM IR和四元式。进一步地，可以基于IR进行代码优化，并最终生成目标机器码或WebAssembly，从而构建一个完整的编译流程。

# 六、参考文献

1. 知乎专栏. "编译原理中是如何进行「词法分析」的" [EB/OL]. 知乎技术专栏, 2021. https://zhuanlan.zhihu.com/p/363589423
2. CSDN 博客. "编译原理：词法分析器设计与实现(C语言)" [EB/OL]. CSDN 技术博客, 2025. https://wenku.csdn.net/column/4amenkbjmk
3. CSDN 博客. "源代码解析艺术：词法分析器设计原理详解" [EB/OL]. CSDN 技术博客, 2020. https://blog.csdn.net/hpu2022/article/details/106530171
4. PingCode 文章. "如何用python写词法分析器" [EB/OL]. PingCode 百科文章, 2025. https://docs.pingcode.com/ask/1091084.html
5. 知乎专栏. "编译原理(6) 自顶向下的分析方法" [EB/OL]. 知乎技术专栏, 2023. https://zhuanlan.zhihu.com/p/654321098
6. 稀土掘金. "计算机编程语言原理与源码实例讲解：递归下降解析器的构建" [EB/OL]. 稀土掘金技术博客, 2023. https://juejin.cn/post/7245678901234567890
7. CSDN 博客. "手写一个Pratt Parser基本运算解析器" [EB/OL]. CSDN 技术博客, 2023. https://blog.csdn.net/qq\_45678901/article/details/136589823
8. 知乎专栏. "编程实践｜如何使用MoonBit编写Pratt解析器？" [EB/OL]. 知乎技术专栏, 2024. https://zhuanlan.zhihu.com/p/789012345
9. pandolia. 自己动手写编译器 [EB/OL]. (2019-01-01) [2025-05-08]. https://pandolia.net/tinyc/index.html
10. Steve Klabnik, Carol Nichols, Rust Community. The Rust Programming Language [EB/OL]. (2023-05-20)[2025-05-08]. https://doc.rust-lang.org/book/
11. Overview of the compiler - Rust Compiler Development Guide [EB/OL]. (2023-05-20) [2025-05-08]. https://rustc-dev-guide.rust-lang.org/overview.html
12. Logos Handbook. Logos Handbook [EB/OL]. [2025-05-08]. https://logos.maciej.codes/