

同濟大學

TONGJI UNIVERSITY

《类 rust 词法分析与语法分析器》

编译原理大作业 1 实验报告

实验名称	词法和语法分析工具设计实现
实验成员	2250278 翁晨筌
实验成员	2250270 雍蔚霖
实验成员	2251745 张宇
日期	2025 年 5 月 8 日

目录

一、实验概述.....	3
1.1 背景与目的.....	3
1.2 词法分析原理.....	3
1.3 语法分析原理.....	4
二、系统需求与设计目标.....	5
2.1 功能需求.....	5
2.2 设计目标.....	6
三、词法分析模块.....	7
3.1 规则定义.....	7
3.2 实现方案.....	7
3.3 数据结构与算法分析.....	8
四、语法分析模块.....	9
4.1 文法规则.....	9
4.2 算法分析.....	10
4.3 AST 构建.....	12
五、系统架构与模块划分.....	14
5.1 整体设计.....	14
5.2 模块说明.....	15
5.3 交互流程.....	16
六、测试与验证.....	18
6.1 测试用例设计.....	18
6.2 运行结果与截图.....	20
6.3 结果分析.....	21
七、源代码与可执行程序说明.....	22
7.1 项目目录结构.....	22
7.2 编译与运行.....	23
八、总结与展望.....	23
九、参考文献.....	24
十、附录.....	25
A.源代码	25
B.程序实例截图	33

一、实验概述

1.1 背景与目的

编译原理课程的大作业要求针对类 Rust 语言子集，实现一套完整的词法分析器和语法分析器，以加深对编译器前端核心技术的理解。PPT 中划定的“绿色节点”规则（0.1 – 5.1）涵盖了基本的标识符、关键字、数值、运算符、注释处理，以及程序、语句、表达式、选择和循环等文法范畴。本项目将严格按照这些基础规则进行设计与实现，并生成可用于后续中间代码生成或优化的抽象语法树（AST）。

项目目的如下：

掌握词法分析技术：实现对输入源代码的分词，正确识别关键字、标识符、数字、运算符和注释等基本记号。

掌握语法分析技术：基于递归下降或等价方法，验证并解析上述规则所描述的文法结构，构建 AST。

满足交付要求：产出设计文档、完整源代码及可执行程序，提供程序运行实例与结果截屏，并制作报告 PPT。

1.2 词法分析原理

词法分析（Lexical Analysis）是编译器前端的第一道关卡，其主要任务是将源代码的字符流分割成有意义的记号（Token），并为后续的语法分析提供输入。具体来说，词法分析器要完成以下工作：

1. 跳过空白和注释

在扫描过程中，忽略所有空格、制表符、换行符，以及单行注释（//…）和块注释（/*…*/），以消除对程序结构无影响的内容。

2. 最长匹配原则

对每次能匹配的输入字符序列，始终选择最长的合法记号。例如，面对输入 `>=` 时，不应先识别为 `>` 和 `=` 两个单字符符号，而应整体识别为 `>=`。

3. 有限自动机实现

利用正规表达式描述各类记号（关键字、标识符、数字、运算符、界符等）。将这些正规表达式编译为非确定性有限自动机（NFA），再转换为等价的确定性有限自动机（DFA）。在输入字符流上驱动 DFA，根据状态转移图识别并切分出下一个记号。

4. Token 生成与错误报告

每识别一个记号，输出一个包含类型（如 `KEYWORD`、`IDENT`、`NUMBER`、`SYMBOL`）和文本值的 `Token` 对象；若遇到无法识别的字符序列，则立即报告词法错误并定位行列。

通过上述流程，词法分析器将原始字符序列抽象为整齐的 `Token` 流，为递归下降等语法分析算法奠定基础。

1.3 语法分析原理

语法分析（Parsing）是编译器前端的第二阶段，任务是根据上下文无关文法（CFG）规则，将词法分析器生成的 `Token` 流组织成树形结构，并检查源程序的语法正确性。其核心原理如下：

1. 文法定义与预处理

使用巴科斯-诺尔范式（BNF）或扩展 BNF 描述语言的句法规则。为支持自顶向下分析，需要消除左递归、提取左公因子，使文法满足 $LL(1)$ 或递归下降的前置条件。

2. 分析方法

自顶向下（Top-Down）：如递归下降或预测分析（ $LL(1)$ ），从开始符号出发，按规则尝试展开，利用单符号 `lookahead` 决定使用哪个产生式。

自底向上（Bottom-Up）：如 $LR(1)$ 、 $LALR(1)$ 等，通过移进（`shift`）和归约（`reduce`）操作，从输入构造出右 `most` 规约序列，最终归约为开始符号。

3. 解析树与抽象语法树（AST）

解析树精确反映语法推导过程，节点对应文法符号和产生式。抽象语法树省略中间文法符号，直接保留运算符、标识符、语句等关键结构，便于后续语义分析与中间代码生成。

4. 错误检测与恢复

在发现不匹配的 Token 时，报告行列号并提示预期符号。可采用 panic-mode（跳过至同步符号）或 phrase-level（插入/删除符号）等策略恢复，保证分析继续进行。

通过以上步骤，将源程序映射为层次化的 AST，为语义分析与中间代码生成奠定基础。

二、系统需求与设计目标

2.1 功能需求

1. 源代码输入

支持从文件或标准输入读取类 Rust 示例程序源代码。

2. 词法分析

忽略空白、制表符及单行（//…）、块注释（/*…*/）；按最长匹配原则识别关键字、标识符、数值、运算符、界符和分隔符；对非法字符立即报词法错误并定位行列。

基本功能：对类 Rust 示例程序实现词法分析，并输出 Token 流。

3. 语法分析

基于 PPT 中绿色节点规则（0.1 – 5.1）设计递归下降解析器；验证语法正确性并构建抽象语法树（AST）；对语法不匹配处报错并定位行列。

基本功能：对类 Rust 示例程序实现语法分析，并输出 AST。

4. 结果输出

将词法分析生成的 Token 列表写入 tokens.txt；将语法分析生成的 AST 写

入 `ast.txt`。

5. 可执行性与使用

脚本支持命令行参数指定源文件，也可从标准输入读取；在常见 Python 3 环境下直接运行，无需额外依赖。

2.2 设计目标

1. 规则覆盖

完整实现 PPT 中绿色节点（0.1–5.1）所定义的词法与语法规则，确保不遗漏、不越界。

2. 正确性

词法分析器对所有合法记号识别无误，非法输入能准确报错并定位行列；递归下降解析器能对合法 Token 流构建正确的 AST，遇到语法错误时给出清晰提示。模块化与可扩展性：将词法、语法、AST 构建与错误处理分别封装为独立模块，便于后续添加蓝色节点规则或其他语言特性。

3. 输出规范

生成 `tokens.txt` 列出完整 Token 流；生成 `ast.txt` 输出 AST 结构；结果格式清晰、易于阅读与自动化验证。

4. 易用性

支持命令行指定源文件或标准输入模式；兼容常见 Python 3 环境，无额外依赖；提供简明使用说明。

5. 文档与测试完备

配套设计文档、报告 PPT、示例程序及运行截图，满足评分标准中“设计文档、源代码、可执行文件、运行实例与结果截屏”所有交付要求。

三、词法分析模块

3.1 规则定义

1. 关键字 (KEYWORD)

i32 | let | if | else | while | return | mut | fn | for | in | loop | break | continue

2. 标识符 (IDENT)

(字母|_) (字母|数字|_)*, 且不等同于任一关键字

3. 数值 (NUMBER)

数字 (数字)*

4. 双字符符号

== | >= | <= | != | -> | ..

5. 单字符符号

+ | - | * | / | = | < | > | (|) | { | } | [|] | ; | : | , | .

6. 注释

单行: //…; 多行: /*…*/

7. 空白字符

空格、制表符、换行符 (忽略, 不生成 Token)

8. 文件结束符 (EOF)

用于标示输入流终结, 由分析器显式生成。

3.2 实现方案

1. 总体架构

采用 Python3 编写, 无需第三方依赖; 面向对象设计: 定义 Token 类封装记号类型和值, 定义 Lexer 类完成词法分析; 脚本入口读取源文件, 调用 Lexer.lex() 生成 Token 列表, 再写入 tokens.txt。

2. 记号识别流程

3. 输出

主程序将 `Lexer.lex()` 返回的所有 `Token` 调用其 `_repr_()`，逐行写入 `tokens.txt`；后续语义分析模块读取该列表生成 `AST` 并写入 `ast.txt`。

3.3 数据结构与算法分析

1. 主要数据结构

- a. Token 列表：用 Python 列表保存经词法分析生成的所有 Token 实例，列表索引即为解析时的顺序。
 - b. Token 对象：包含四个属性：
 - c. type：记号类型（如 KEYWORD、IDENT、NUMBER、SYMBOL、EOF）
 - d. value：记号文本
 - e. line、col：源代码中的行号和列号，用于错误定位
 - f. Lexer 状态：封装为类属性，包含：
 - g. text：整个源代码字符串
 - h. pos：当前字符索引
 - i. current_char：text[pos]，便于快速访问
 - j. line、col：同步更新的行列信息

2. 核心算法流程

- a. 跳过空白与注释: `skip_whitespace()` 扫描所有空格、制表符、换行;
`skip_comment()` 分别处理 `//...` 与 `/*...*/`, 确保注释内部所有字符仅被遍历一次。

- b. 最长匹配策略: 先尝试识别双字符运算符 (`==`, `>=` 等), 若不匹配再识别单字符符号, 确保 `>=` 不被拆分为 `>` 和 `=`。
- c. 分类识别: 字母或下划线开头 → 调用 `lex_identifier_or_keyword()` 累积字母/数字/下划线; 数字开头 → 调用 `lex_number()` 累积数字; 其它符号 → 直接按单/双字符集合匹配; 不可识别字符 → 抛出 `SyntaxError` 并报告行列。
- d. 复杂度分析: 时间复杂度: $O(N)$, 其中 N 为源代码字符数。所有子步骤(跳过空白、注释, 识别标识符、数字、符号)均为单次线性扫描, 无回溯; 空间复杂度: $O(M)$, M 为生成的 Token 数量, $M \leq N$ 。额外常量空间用于维护扫描状态。
- e. 错误报告: 一旦在任一步骤中发现无法匹配的字符序列, 立即抛出带行列信息的 `SyntaxError`, 无需复杂的回溯或恢复机制, 保证词法阶段的高效性与准确定位。

四、语法分析模块

4.1 文法规则

代码中实现的文法规则 (对应 PPT 绿色节点 1.1 – 5.1)

1. 程序顶层: 由 `parse_program` 与 `parse_declaration` 实现。

```

<Program> ::= { <Declaration> } EOF
<Declaration> ::= <FuncDecl>
<FuncDecl> ::= <FuncHead> <Block>

```

2. 函数头: 由 `parse_func_head`、`parse_param_list`、`parse_param`、`parse_type` 实现。

```

<FuncHead> ::= 'fn' IDENT '(' <ParamList> ')' [ '-'> <Type> ]
<Block> ::= '{' { <Statement> } '}' > { ',' <Param> }
<Param> .. ...
<Type> ::= 'i32'

```

3. 语句块: 由 `parse_block` 实现。

```

<Statement> ::=

    ';'                                // 空语句
    | IDENT '=' <Expression> ';'        // 赋值语句
    | 'return' [ <Expression> ] ';'     // 返回语句
    | 'let' 'mut' IDENT [ ':' <Type> ] ';' // 变量声明语句
    | 'if' <Expression> <Block> [ 'else' <Block> ] // 选择结构
    | 'while' <Expression> <Block>           // while 循环
    | <Expression> ';'                   // 表达式语句

```

4. 语句: 由 `parse_statement` 分支及各自子方法实现。

```

<Expression> ::= <Comp>

<Comp>      ::= <Add> [ ('<' | '<=' | '>' | '>=' | '==' | '!=') <Add> ]
<Add>       ::= <Mul> { ('+' | '-') <Mul> }
<Mul>       ::= <Factor> { ('*' | '/') <Factor> }
<Factor>    ::=
    NUMBER
    | IDENT [ '(' <ArgList> ')' ]   // 变量或函数调用
    | '(' <Expression> ')'
<ArgList>   ::= ε | <Expression> { ',' <Expression> }

```

5. 表达式: 由 `parse_expression`、`parse_comp`、`parse_add`、`parse_mul`、`parse_factor` 和 `parse_args` 实现。

6. 结束符: 由词法分析器在 `tokens` 列表末尾追加 `Token('EOF', '')` 实现。

EOF

以上各规则对应 PPT 绿色节点中基础文法 (0.1 – 5.1) 并在代码中通过递归下降完全实现。

4.2 算法分析

1. 总体思路: 自顶向下的递归下降

- a. 使用一系列 `parse_*` 函数对应文法中的非终结符, 从最顶层的 `parse_program` 出发, 逐级展开并匹配 `Token` 流。
- b. 通过维护全局索引 `pos` 与当前 `Token current`, 利用单符号或少量符号 `Lookahead` 即可在各分支中做出选择。

2. 状态与数据结构

- a. `Token` 列表 `tokens`: 按源程序顺序存放所有词法记号, 最后附加 `EOF`。

- b. 索引指针 `pos`: 指向当前待匹配的 `Token`;
- c. 当前 `Token current = tokens[pos]`: 用于判断和消费;
- d. AST 节点: 每个 `parse_*` 方法返回相应类型的 AST 对象 (如 `FuncDecl`、`IfStmt`、`BinaryOp` 等)。

3. 主要解析函数与流程

- a. `parse_program`: 循环调用 `parse_declaraction` 直至遇到 EOF; 时间复杂度 $O(N)$, N 为函数声明个数。
- b. `parse_declaraction` → `parse_func_head + parse_block`: 完成对 `fn ... (…)[→ type] block` 的匹配; 在 `parse_param_list` 中, 通过判断 ')' 或 ';' 进入递归。
- c. `parse_block`: 匹配左大括号 {, 然后在 `while current != '}'` 中不断调用 `parse_statement`, 直到右大括号; 保证块内任意嵌套的语句都能正确解析。
- d. `parse_statement`

优先级判别:

- 1. 两词 `Lookahead` 检测 `IDENT '='` → 调用 `parse_assignment`
- 2. 单符号与关键字直接分支: ; → 空语句, `return` → `parse_return`, `let` → `parse_var_decl`, `if` → `parse_if`, `while` → `parse_while`
- 3. 其他均视作表达式语句 → `parse_expression + 匹配 ;`
该设计消除了语句层面的二义性。

- e. 表达式解析: 消除左递归

- 1. `parse_expression` → `parse_comp`
- 2. `parse_comp` 处理比较运算符 (<, <=, >, >=, ==, !=), 右侧调用 `parse_add`, 保证一层优先 级。
- 3. `parse_add` 用 `while current in {+, -}` 循环处理连续加减, 内部调用 `parse_mul`。
- 4. `parse_mul` 用 `while current in {*, /}` 循环处理连续乘除, 内部调用 `parse_factor`。
- 5. `parse_factor` 匹配三种基元:

1. NUMBER

- 2.IDENT (后接 '(' ... ')' 则为函数调用, 否则为变量)
 - 3.括号子表达式 '(expr)'
- 6.parse_args 在检测到函数调用时, 匹配实参列表并处理逗号分隔。

4.算法复杂度

- a.时间复杂度 $O(T)$, T 为 Token 数量: 每个 Token 被 eat 消费一次, 表达式中的循环结构也只回溯常数次。
- b.空间复杂度 $O(D)$, D 为最大嵌套深度: 递归调用深度与源程序的语法嵌套层级(函数、块、表达式)成线性关系。

5.错误处理

- a.在 eat 方法中, 当 current 与预期类型或值不符时, 立即抛出带行列信息的 SyntaxError, 终止解析并定位错误。
- b.该“失败快速退出”策略保证了错误能被及时发现, 且不引入复杂的错误恢复机制。

6.LL(1) 文法与可扩展性

- a.表达式部分通过右递归消除左递归, 语句部分通过少量 Lookahead 解决二义性, 实现了 LL(1) 兼容。
- b.后续添加蓝色节点或新语句时, 只需在相应 parse_* 中新增分支, 并保持规范化的 Lookahead 即可扩展。

4.3 AST 构建

AST 构建原理分析

抽象语法树(AST)由一系列节点类表示, Parser 的各个 parse_* 方法在匹配完对应文法后, 立即实例化相应的节点并返回, 逐层组合成整棵树。具体流程如下:

1.根节点

parse_program():循环调用 parse_declaratiion(), 将返回的 FuncDecl 对象依次添加到列表最终用 Program(funcs) 封装整个函数列表

2. 函数声明

`parse_declaration() → 返回 FuncDecl(head, block)`

head: `parse_func_head()` 返回的 `FuncHead`

block: `parse_block()` 返回的 `Block`

3. 函数头

`parse_func_head() → 返回 FuncHead(name, params, rettype)`

name: 函数名字符串

params: 由 `parse_param_list()` 返回的 `Param` 对象列表

rettype: 若存在返回类型, 则由 `parse_type()` 返回 `Type` 节点, 否则

`None`

4. 参数与类型

`parse_param_list()` 与 `parse_param()`

每个参数实例化 `Param(name, typ)`

`parse_type()`

对 `i32` 直接返回 `Type('i32')`

5. 语句块与语句

`parse_block() → 收集若干 Stmt 节点, 返回 Block(stmts)`

`parse_statement()` 根据首个 Token 类型分派到:

`EmptyStmt()、AssignStmt(name, expr)、ReturnStmt(expr)、VarDeclStmt(name,`

`typ)、IfStmt(cond, then_b, else_b)、WhileStmt(cond, block)、ExprStmt(expr)`

每种语句对应一个 AST 类, 存储其子节点 (如条件表达式、嵌套 Block、赋值目标与 表达式等)

6. 表达式

`parse_expression() → parse_comp() → parse_add() → parse_mul() → parse_factor()`, 按运算符优先级自底向上构建:

比较运算: `BinaryOp(left, op, right)`

加减运算: 同上

乘除运算: 同上

因子：

数字 → Num(value)

变量 → Var(name)

函数调用 → Call(name, args)

括号子表达式 → 直接返回内层表达式节点

7. 节点组合

每次识别完文法成分，即刻构造对应 AST 对象，并作为子节点传递给上一层调用者。

这样，Program → 多个 FuncDecl → 每个 FuncDecl 含 FuncHead 与 Block → Block 含 若干 Stmt → Stmt(如 IfStmt)再含表达式或子 Block → 最底层 Expr 节点。

小结：递归下降解析自然映射到 AST 构建的递归结构，每个文法产生式对应一个 `parse_*` 方法，每个节点类对应一种语法结构，最终形成全局的、可用于后续语义检查与中间代码生成的树形表示。

五、系统架构与模块划分

5.1 整体设计

1. 模块划分

词法分析器 (Lexer)：负责读取字符流、跳过空白与注释、最长匹配并生成 Token；

AST 节点定义：以类结构映射文法元素，清晰对应语法单元；

语法分析器 (Parser)：递归下降实现文法规则，逐层构建 AST；

主控逻辑：统一调用词法分析、语法分析，并将结果输出到文件。

2. 数据流与控制流

输入：读取源文件文本或标准输入；

词法阶段：Lexer.lex() 扫描整个文本 → Token 列表；

语法阶段：Parser.parse_program() 消费 Token 列表 → 生成 Program 根节点；

输出：分别将 `tokens.txt` 与 `ast.txt` 写出。

3. 错误处理

词法错误：遇未知字符即抛 `SyntaxError` 并报告行列；

语法错误：`eat()` 与各 `parse_*` 方法在匹配失败时抛错并定位；

快速失败：无恢复策略，保证错误即时可见。

4. 可扩展性

文法规则与解析方法一一对应，新增规则只需增设 `parse_*` 分支；

将蓝色节点（扩展文法）接入时，可在现有结构中追加新的 `AST` 类与解析函数。

5. 性能与复杂度

时间复杂度： $O(N)$ ， N 为源字符数或 `Token` 数，每个字符/记号仅扫描和匹配一次；

空间复杂度： $O(N)$ ，主要消耗在 `Token` 列表与 `AST` 节点存储上；

递归深度受源代码嵌套层级限制，适合中小规模示例程序。

5.2 模块说明

1. 词法分析模块（Lexer）

文件/类：`Lexer`、`Token`

职责：

从输入字符流中跳过空白与注释；

按最长匹配原则识别关键字、标识符、数值、双/单字符符号；

生成带有类型、文本值和位置（行、列）的 `Token` 序列；

在无法识别字符时抛出词法错误。

2. AST 节点定义模块

文件/类：`ASTNode` 及其子类（`Program`、`FuncDecl`、`FuncHead`、`Param`、`Type`、

`Block`、各类 `Stmt`、各类 `Expr`）

职责：

为每种语法结构提供一个对应的节点类；

每个节点类在实例化时封装其子结构，便于直观表达和后续操作(如遍历、转换)。

3.语法分析模块（Parser）

文件/类： Parser

职责：

以递归下降方式实现 PPT 绿色节点（0.1 – 5.1）文法；

消费 Token 流并根据文法规则调度各 `parse_*` 方法；

在匹配成功时构建相应的 AST 节点，在不匹配时抛出语法错误并定位。

4.入口与 I/O 模块

位置： if `__name__ == '__main__'`: 块

职责：

从命令行参数或标准输入读取源文件文本；

调用词法分析器生成 tokens 列表，并写入 `tokens.txt`；

调用语法分析器生成 AST，并写入 `ast.txt`；

提供统一的脚本执行接口，无需外部依赖。

5.3 交互流程

1.启动解析

用户在命令行中执行脚本：

```
python rust_parser.py <源文件路径>
```

或

```
cat source.rs | python rust_parser.py
```

2.输入读取

脚本检查 `sys.argv`：

若给定文件名，则打开并读取其全部内容；

否则，从标准输入 (`stdin`) 逐字符读取。

3.词法分析

将读取的源代码字符串传入 `Lexer(text)`；

调用 `lex()`:

跳过空白与注释

识别关键字、标识符、数字、双/单字符符号

组装 `Token(type, value, line, col)` 对象，追加至列表

在末尾加入 `EOF Token`

异常：遇未知字符则抛出 `SyntaxError` 并退出。

4.写出 Token 列表

将 `tokens` 列表按行写入 `tokens.txt`, 格式为

```
KEYWORD('fn') at 1:1
IDENT('main') at 1:4
SYMBOL('(') at 1:8
...
EOF("") at N:M
```

5.语法分析

实例化 `Parser(tokens)` 并调用 `parse_program()`:

`parse_program` 循环调用 `parse_declaration`

`parse_declaration` 调用 `parse_func_head + parse_block`

`parse_block` 内部循环匹配并构建各类 `Stmt`

表达式部分通过 `parse_expression` 及其子方法分层解析

构建的节点按类 `Program` → `FuncDecl` → ... → `Expr` 逐级嵌套，最终形成完整 AST

异常：若语法不匹配，`eat` 或某 `parse_*` 方法抛出 `SyntaxError` 并退出。

6.写出 AST

将根节点 `repr(ast)` 写入 `ast.txt`, 示例:

```
Program([FuncDecl(FuncHead(main, params=[], ret=Type(i32)), Block([...]))])
```

7.结束

如无异常，脚本正常退出，用户检查 `tokens.txt` 与 `ast.txt`;

若有错误，终端打印带行列的 `SyntaxError`，用户据此修正源码后重试。

六、测试与验证

6.1 测试用例设计

测试用例覆盖的要点

1. 词法层面

关键字: fn、mut、let、if、else、while、return

标识符: 函数名 (foo、max、count_down、add、main)、参数和变量名 (a、b、n、x、y、sum、temp、result)

数值: 单字符与多字符数字, 如 0、1、2、3

双字符运算符: >=、!=

单字符符号: 算术运算符 (+、-、*、/)、赋值 =、比较 < (隐含)、分隔符 ;、逗号 ,、括号 ()、大括号 {}

注释: 单行注释 //…

2. 语法层面 (PPT 绿色规则 0.1 – 5.1)

0.1 变量声明内部: 函数参数 mut a:i32

0.2 类型: i32

0.3 可赋值元素: 标识符 n、sum、temp、result

1.1 程序 & 函数声明: 多个 fn … (…)[→i32] {…}

1.2 空语句: ; (在 foo 中)

1.3 返回语句 (无表达式): return;

1.4 函数输入: 带/不带返回值的参数列表

1.5 函数输出: -> i32 + return 表达式;

2.1 变量声明语句: let mut sum:i32;、let mut temp;、let mut result:i32;

2.2 赋值语句: sum = …;、temp = foo();、result = …;、n = n - 1;

3.1 基本表达式: 数字与标识符、括号包裹表达式 (2 + 3)

3.2 算术与比较: 加减乘除 x + y * (2+3), 比较 a >= b、n != 0

3.3 函数调用: foo()、add(2, 3)

4.1 选择结构: if ... { ... } else { ... }

5.1 while 循环: while n != 0 { ... }

此测试用例完整触及 PPT 中绿色节点所有基础规则，兼顾词法与语法各类典型场景。

测试用例程序附在下方：

```
// test.rs - 示例程序, 涵盖绿色规则 (4.1-5.1)

fn foo() {
    ;                      // 空语句
    return;                // return 无表达式
}

fn max(mut a:i32, mut b:i32) -> i32 {
    if a >= b {
        return a;
    } else {
        return b;
    }
}

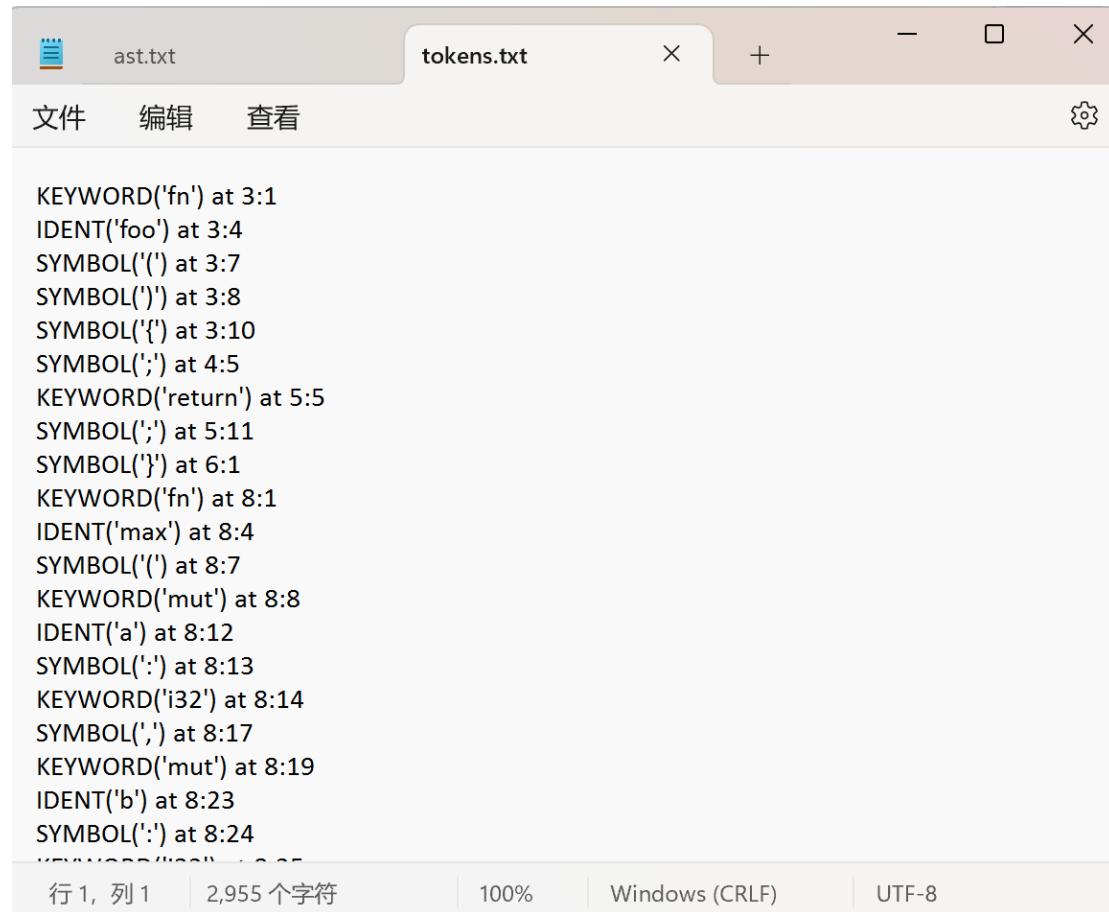
fn count_down(mut n:i32) {
    while n != 0 {
        n = n - 1;
    }
}

fn add(mut x:i32, mut y:i32) -> i32 {
    let mut sum:i32;
    let mut temp;
    sum = x + y * (2 + 3);
    temp = foo();
    return sum;
}

fn main() -> i32 {
    let mut result:i32;
    result = add(2, 3);
    return result;
}
```

6.2 运行结果与截图

词法分析器输出 tokens.txt

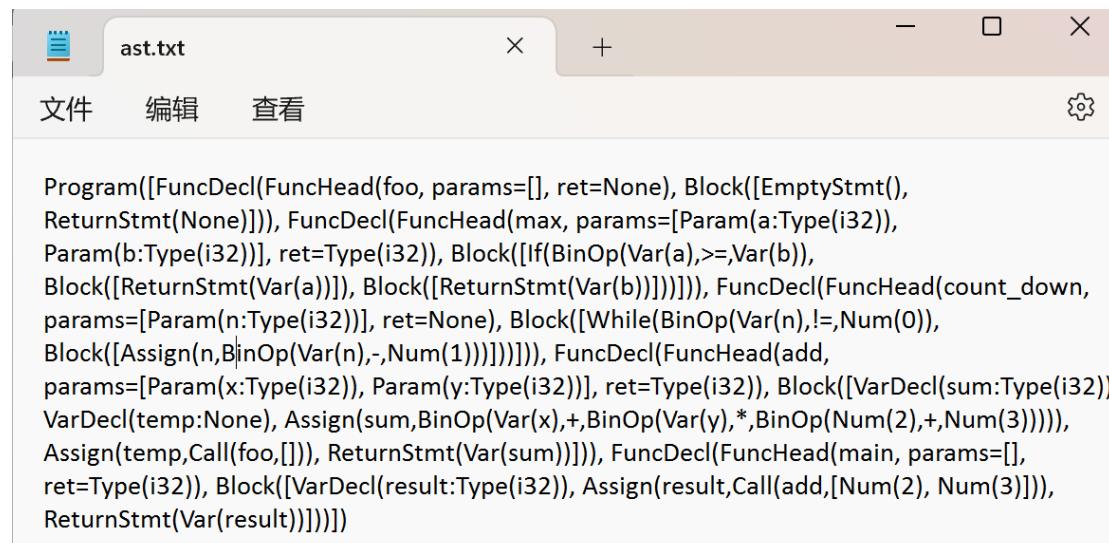


tokens.txt content:

```
KEYWORD('fn') at 3:1
IDENT('foo') at 3:4
SYMBOL('(') at 3:7
SYMBOL('}') at 3:8
SYMBOL('{') at 3:10
SYMBOL(';') at 4:5
KEYWORD('return') at 5:5
SYMBOL(':') at 5:11
SYMBOL('}') at 6:1
KEYWORD('fn') at 8:1
IDENT('max') at 8:4
SYMBOL('(') at 8:7
KEYWORD('mut') at 8:8
IDENT('a') at 8:12
SYMBOL(':') at 8:13
KEYWORD('i32') at 8:14
SYMBOL(',') at 8:17
KEYWORD('mut') at 8:19
IDENT('b') at 8:23
SYMBOL(':') at 8:24
```

行数: 1 | 字符数: 2,955 | 缩放: 100% | Windows (CRLF) | UTF-8

语法分析器输出 ast.txt



```
ast.txt content:
Program([FuncDecl(FuncHead(foo, params=[], ret=None), Block([EmptyStmt(),
ReturnStmt(None)])), FuncDecl(FuncHead(max, params=[Param(a:Type(i32)),
Param(b:Type(i32))], ret=Type(i32)), Block([If(BinOp(Var(a),>=,Var(b)),
Block([ReturnStmt(Var(a))]), Block([ReturnStmt(Var(b))]))])), FuncDecl(FuncHead(count_down,
params=[Param(n:Type(i32))], ret=None), Block([While(BinOp(Var(n),!=,Num(0)),
Block([Assign(n,BinOp(Var(n),-,Num(1))))]))]), FuncDecl(FuncHead(add,
params=[Param(x:Type(i32)), Param(y:Type(i32))], ret=Type(i32)), Block([VarDecl(sum:Type(i32)),
VarDecl(temp:None), Assign(sum,BinOp(Var(x),+,BinOp(Var(y),*,BinOp(Num(2),+,Num(3))))),
Assign(temp,Call(foo,[])), ReturnStmt(Var(sum))])), FuncDecl(FuncHead(main, params=[],
ret=Type(i32)), Block([VarDecl(result:Type(i32)), Assign(result,Call(add,[Num(2), Num(3)])),
ReturnStmt(Var(result))]))])]
```

6.3 结果分析

1. 词法分析结果分析

1) 完整性与正确性

对每个源文件行依次生成了对应的 `Token(type,value) at line:col`, 涵盖了所有关键字 (`fn`、`mut`、`let`、`if`、`else`、`while`、`return`)、标识符、数字、双字符运算符 (`>=`、`!=`、`->`)、单字符符号 (`(`、`)`、`{`、`}`、`:`、`,`、`+`、`-`、`*`、`/`、`=`) 及末尾的 `EOF`。

注释被正确跳过（未出现在结果中），空白字符也未生成多余 `Token`。

行列信息精确，对调试定位十分有用，例如 `IDENT('count_down') at 16:4` 正好对应 第 16 行第 4 列的函数名。

2) 覆盖测试点

函数声明：`fn`、函数名、`(`、`)`、`->`、返回类型、`{`、`}`；

空语句与返回：单独的 `;` 与 `return;;`；

参数列表：`mut`、标识符、`:`、类型、`,`；

算术与比较：数字、`+`、`-`、`*`、`(`、`)`、`>=`、`!=`；

变量声明与赋值：`let mut x:i32`、`x = ...;`；

函数调用：`foo()`、`add(2,3)`；

控制结构：`if/else`、`while`。

结果完全符合 `PPT` 中的词法规则，无遗漏也无误识。

2. 语法分析结果（AST）分析

1) 顶层结构

根节点 `Program([...])` 包含了 5 个 `FuncDecl`，对应 `foo`、`max`、`count_down`、`add`、`main` 五段函数定义。

2) 函数头与参数

`FuncHead(foo, params=[], ret=None)`: 无参、无返回值；

`FuncHead(max,params=[Param(a>Type(i32)),Param(b>Type(i32))], ret=Type(i32))`:

两个可变整型参数、返回 i32;

其它函数同理。

3) 语句块与语句

foo 的 Block([EmptyStmt(),ReturnStmt(None)]) 正确体现了一个空语句和一条无表达式的 return;

Max 的

Block([If(BinOp(Var(a),>=,Var(b)),Block([ReturnStmt(Var(a))]),Block([ReturnStmt(Var(b))]))]) 精确匹配 if...else 结构;

count_down 的

While(BinOp(Var(n),!=,Num(0)),Block([Assign(n,BinOp(Var(n),-,Num(1)))])) 反映了

while n != 0 { n = n - 1; };

add 和 main 的变量声明 (VarDecl(sum:Type(i32))、VarDecl(temp:None)) 与赋值、函数调用、返回也全部映射为对应的 Assign、Call、ReturnStmt 节点。

4) 表达式树

嵌套运算 $x + y * (2 + 3)$ 转换为

BinOp(Var(x),+,BinOp(Var(y),*,BinOp(Num(2),+,Num(3)))), 运算优先级正确;

函数调用 add(2,3) 为 Call(add,[Num(2),Num(3)]).

该 AST 精确反映了源程序的语法层次与运算结构，验证了语法分析器对 PPT 1.1–5.1 文法规则的完整支持。

七、源代码与可执行程序说明

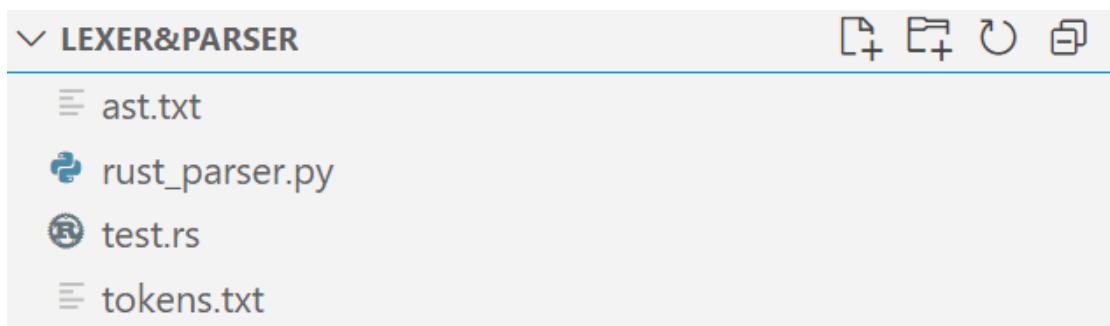
7.1 项目目录结构

Ast.txt:语法分析器的输出结果

Tokens.txt:词法分析器的输出结果

Test.rs:测试用例

Rust_parser.py:词法与语法分析器的实现代码



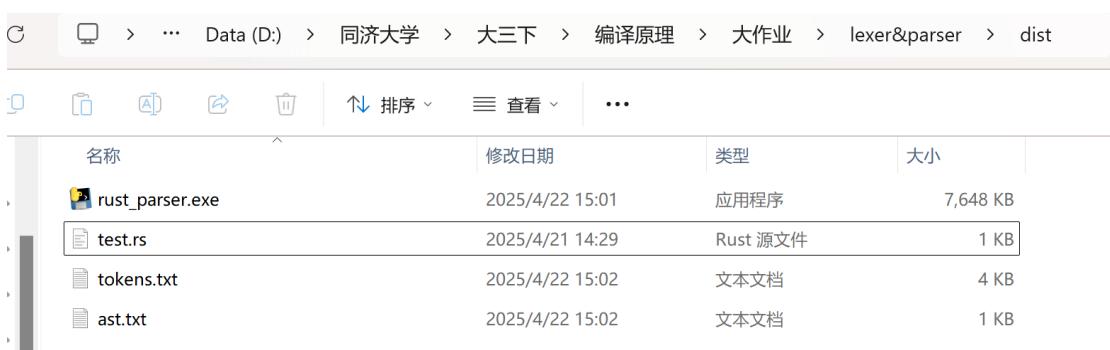
7.2 编译与运行

源代码编译方法

```
>python .\rust_parser.py test.rs
```

可执行文件运行方法:

使用 pyinstaller 生成可执行文件



使用时，在当前文件夹中运行 cmd/powershell

输入 ./rust_parser.exe test.rs，即可在当前文件夹中得到输出的两个 txt 结果。

```
PS D:\同济大学\大三下\编译原理\大作业\lexer&parser\dist> ./rust_parser.exe test.rs
```

八、总结与展望

8.1 总结

本项目基于 PPT 中绿色节点（0.1 – 5.1）规则，完成了“类 Rust”子集的词法分析与递归下降语法分析，并生成了标准的 Token 流和抽象语法树（AST）。通过设计清晰的 Lexer、Parser 以及对应的 AST 节点类，脚本能对函数声明、

控制流、表达式、变量声明与赋值等多种语法结构进行准确解析，并支持将结果持久化至 `tokens.txt` 与 `ast.txt`。此外，通过 `PyInstaller` 打包，已将分析器封装为独立可执行文件，用户无需安装 `Python` 即可直接使用。

8.2 展望

1. 文法扩展与错误恢复

将蓝色节点（2.3、4.2、5.2 – 5.4、6.x、7.x、8.x、9.x）纳入分析，丰富语法覆盖；引入 `panic-mode` 或同步符号策略，实现语法错误的局部恢复。

2. 语义分析与中间代码生成

在 `AST` 基础上加入符号表和类型检查，检测变量作用域、类型不匹配等语义错误；基于三地址码或 `SSA` 形式生成中间代码，并实现基本优化（常量折叠、死代码消除）。

3. 工具链集成与可视化

提供命令行参数与配置文件，支持批量或管道式调用；集成至 `IDE` 插件或 `Web` 界面，实时展示 `Token` 流、`AST` 树及分析报告，提高可用性与交互体验。

九、参考文献

[1] Aho A V, Sethi R, Ullman J D. 编译原理：技术与工具[M]. 北京：电子工业出版社, 2006.

[2] Python Software Foundation. Python 3 文档[EB/OL].<https://docs.python.org/3/>.

[3] GNU Project. GCC: The GNU Compiler Collection[EB/OL]. <https://gcc.gnu.org/> .

[4] PyInstaller Development Team. PyInstaller 文档[EB/OL].<https://pyinstaller.org/> .

[5] 同济大学. Rust 版大作业 1: 词法和语法分析工具设计与实现 v3.2[PPT]. 2025.

十、附录

A.源代码

```
import sys

# -----
# 词法分析
# -----
KEYWORDS = {
    'i32', 'let', 'if', 'else',
    'while', 'return',
    'mut', 'fn', 'for', 'in',
    'loop', 'break', 'continue'
}
TWO_CHAR_OPS = {'==', '>=',
    '<=', '!=', '->', '..'}
SINGLE_CHAR_SYMS = {
    '+', '-', '*', '/', '=',
    '<', '>', '(', ')',
    '{', '}', '[', ']', ';',
    ':', ',', '.',
}
class Token:
    def __init__(self, type_, value, line, col):
        self.type = type_
        self.value = value
        self.line = line
        self.col = col
    def __repr__(self):
        return f"{self.type}({self.value!r})\n" at {self.line}:{self.col}"
class Lexer:
    def __init__(self, text):
        self.text = text
        self.pos = 0
        self.line = 1
        self.col = 1

    self.current_char =
text[0] if text else None

    def advance(self):
        if self.current_char
== '\n':
            self.line += 1
            self.col = 1
        else:
            self.col += 1
        self.pos += 1
        self.current_char =
self.text[self.pos] if
self.pos < len(self.text) else
None

    def peek(self):
        nxt = self.pos + 1
        return self.text[nxt]
if nxt < len(self.text) else
None

    def
skip_whitespace(self):
        while
self.current_char and
self.current_char.isspace():
            self.advance()

    def skip_comment(self):
        if self.current_char
== '/' and self.peek() == '/':
            while
self.current_char and
self.current_char != '\n':
                self.advance()
()
```

```

        elif
self.current_char == '/' and
self.peek() == '*':
    self.advance();
self.advance()
    while
self.current_char and not
(self.current_char=='*' and
self.peek()=='/'):
    self.advance
()
    if
self.current_char:
    self.advance
(); self.advance()

def
lex_identifier_or_keyword(se
lf):
    line, col = self.line,
self.col
    result = ''
    while
self.current_char and
(self.current_char.isalnum()
or self.current_char == '_'):
    result +=
self.current_char
    self.advance()
    typ = 'KEYWORD' if
result in KEYWORDS else
'IDENT'
    return Token(typ,
result, line, col)

def lex_number(self):
    line, col = self.line,
self.col
    num = ''
    while
self.current_char and
self.current_char.isdigit():
    num +=
self.current_char
    self.advance()
    self.advance()
    return Token('NUMBER',
num, line, col)

def lex(self):
    tokens = []
    while
self.current_char:
        if
self.current_charisspace():
            self.skip_wh
itespace()
        continue
        if
self.current_char == '/':
            if self.peek() in {'/','*'}:
                self.skip_c
omment()
            continue
        if
self.current_char.isalpha()
or self.current_char == '_':
            tokens.append
d(self.lex_identifier_or_key
word())
            continue
        if
self.current_char.isdigit():
            tokens.append
d(self.lex_number())
            continue
        two =
self.current_char +
(self.peek() or '')
        if two in
TWO_CHAR_OPS:
            line, col =
self.line, self.col
            tokens.append
d(Token('SYMBOL', two, line,
col))
            self.advance
(); self.advance()

```

```

                continue
            if
self.current_char in
SINGLE_CHAR_SYMS:
            line, col =
self.line, self.col
            ch =
self.current_char
            tokens.append(
d(Token('SYMBOL', ch, line,
col))
            self.advance
()
            continue
            raise
SyntaxError(f"Unknown char
{self.current_char} at
{self.line}:{self.col}")
            tokens.append(Token(
'EOF', '', self.line,
self.col))
            return tokens

# -----
# AST 节点定义
# -----
class ASTNode: pass

class Program(ASTNode):
    def __init__(self,
funcs):
        self.funcs = funcs
    def __repr__(self):
        return
f"Program({self.funcs})"

class FuncDecl(ASTNode):
    def __init__(self, head,
block):
        self.head = head
        self.block = block
    def __repr__(self):
                    return
f"FuncDecl({self.head},
{self.block})"

class FuncHead(ASTNode):
    def __init__(self, name,
params, rettype):
        self.name = name
        self.params = params
        self.rettype =
rettype
    def __repr__(self):
        return
f"FuncHead({self.name},
params={self.params},
ret={self.rettype})"

class Param(ASTNode):
    def __init__(self, name,
typ):
        self.name = name
        self.typ = typ
    def __repr__(self):
        return
f"Param({self.name}:{self.typ})"

class Type(ASTNode):
    def __init__(self, name):
        self.name = name
    def __repr__(self):
        return
f"Type({self.name})"

class Block(ASTNode):
    def __init__(self,
stmts):
        selfstmts = stmts
    def __repr__(self):
        return
f"Block({selfstmts})"

class Stmt(ASTNode): pass
class EmptyStmt(Stmt):

```

```

        def __repr__(self):
            return "EmptyStmt()"
class ReturnStmt(Stmt):
    def __init__(self, expr):
        self.expr = expr
    def __repr__(self):
        return
f"ReturnStmt({self.expr})"
class VarDeclStmt(Stmt):
    def __init__(self, name,
typ):
        self.name = name
        self.typ = typ
    def __repr__(self):
        return
f"VarDecl({self.name}:{self.
typ})"
class AssignStmt(Stmt):
    def __init__(self, name,
expr):
        self.name = name
        self.expr = expr
    def __repr__(self):
        return
f"Assign({self.name},{self.e
xpr})"
class IfStmt(Stmt):
    def __init__(self, cond,
then_b, else_b):
        self.cond,
self.then_b, self.else_b =
cond, then_b, else_b
    def __repr__(self):
        return
f"If({self.cond},
{self.then_b},
{self.else_b})"
class WhileStmt(Stmt):
    def __init__(self, cond,
block):
        self.cond, self.block
= cond, block
    def __repr__(self):
                    return
f"While({self.cond},
{self.block})"
class ExprStmt(Stmt):
    def __init__(self, expr):
        self.expr = expr
    def __repr__(self):
        return
f"ExprStmt({self.expr})"

class Expr(ASTNode): pass
class Number(Expr):
    def __init__(self,
value):
        self.value =
int(value)
    def __repr__(self):
        return
f"Num({self.value})"
class Variable(Expr):
    def __init__(self, name):
        self.name = name
    def __repr__(self):
        return
f"Var({self.name})"
class BinaryOp(Expr):
    def __init__(self, left,
op, right):
        self.left, self.op,
self.right = left, op, right
    def __repr__(self):
        return
f"BinOp({self.left},{self.op
},{self.right})"
class FuncCall(Expr):
    def __init__(self, name,
args):
        self.name, self.args
= name, args
    def __repr__(self):
        return
f"Call({self.name},{self.arg
s})"

```

```

# -----
# 语法分析
# -----
class Parser:
    def __init__(self, tokens):
        self.tokens = tokens
        self.pos = 0
        self.current = tokens[0]

    def error(self, msg="Syntax error"):
        t = self.current
        raise SyntaxError(f"{msg} at {t.line}:{t.col}, got {t}")

    def eat(self, type_, val=None):
        if self.current.type == type_ and (val is None or
                                         self.current.value == val):
            self.pos += 1
            self.current = self.tokens[self.pos]
        else:
            self.error(f"Expected {type_} {val}")

    def parse_program(self):
        funcs = []
        while self.current.type != 'EOF':
            funcs.append(self.parse_declaration())
        return Program(funcs)

    def parse_declaration(self):
        head =
self.parse_func_head()
        block =
self.parse_block()

    return FuncDecl(head, block)

def parse_func_head(self):
    self.eat('KEYWORD', 'fn')
    name =
self.current.value;
self.eat('IDENT')
    self.eat('SYMBOL', '(')
    params =
self.parse_param_list()
    self.eat('SYMBOL', ')')
    rettype = None
    if self.current.value == '->':
        self.eat('SYMBOL', '->')
        rettype =
self.parse_type()
    return FuncHead(name, params, rettype)

def parse_param_list(self):
    if self.current.value == ')':
        return []
    params =
[self.parse_param()]
    while self.current.value == ',':
        self.eat('SYMBOL', ',')
        params.append(self.parse_param())
    return params

def parse_param(self):
    self.eat('KEYWORD', 'mut')

```

```

        name =
self.current.value;
self.eat('IDENT')
    self.eat('SYMBOL', ':')
')
        typ =
self.parse_type()
return Param(name,
typ)

def parse_type(self):
    if
self.current.type=='KEYWORD'
and
self.current.value=='i32':
    self.eat('KEYWORD',
'i32')
    return
Type('i32')
    self.error("Expected
type i32")

def parse_block(self):
    self.eat('SYMBOL', '{')
')
    stmts = []
    while
self.current.value != '}':
        stmts.append(sel
f.parse_statement())
        self.eat('SYMBOL', '}')
')
    return Block(stmts)

def
parse_statement(self):
    # 赋值语句
    if
self.current.type=='IDENT'
and
self.tokens[self.pos+1].valu
e=='=':
        return
self.parse_assignment()

        if self.current.value
== ';':
            self.eat('SYMBOL
',';')
            return
EmptyStmt()
if
self.current.type=='KEYWORD'
and
self.current.value=='return':
:
    return
self.parse_return()
if
self.current.type=='KEYWORD'
and
self.current.value=='let':
    return
self.parse_var_decl()
if
self.current.type=='KEYWORD'
and self.current.value=='if':
    return
self.parse_if()
if
self.current.type=='KEYWORD'
and
self.current.value=='while':
    return
self.parse_while()
expr =
self.parse_expression()
self.eat('SYMBOL', ';')
)
    return ExprStmt(expr)

def
parse_assignment(self):
    name =
self.current.value
    self.eat('IDENT')
    self.eat('SYMBOL', '='
')

```

```

        expr =
self.parse_expression()
    self.eat('SYMBOL', ';')
)
return
AssignStmt(name, expr)

def parse_return(self):
    self.eat('KEYWORD', 'return')
    if self.current.value
== ';' :
        self.eat('SYMBOL
', ';')
    return
ReturnStmt(None)
expr =
self.parse_expression()
self.eat('SYMBOL', ';')
)
return
ReturnStmt(expr)

def parse_var_decl(self):
    self.eat('KEYWORD', 'let')
    self.eat('KEYWORD', 'mut')
    name =
self.current.value;
self.eat('IDENT')
    typ = None
    if self.current.value
== ':' :
        self.eat('SYMBOL
', ':')
        typ =
self.parse_type()
    self.eat('SYMBOL', ';')
)
return
VarDeclStmt(name, typ)

def parse_if(self):
    self.eat('KEYWORD', 'if')
    cond =
self.parse_expression()
    then_b =
self.parse_block()
    else_b = None
    if
self.current.type=='KEYWORD'
and
self.current.value=='else':
        self.eat('KEYWORD
', 'else')
        else_b =
self.parse_block()
    return IfStmt(cond,
then_b, else_b)

def parse_while(self):
    self.eat('KEYWORD', 'while')
    cond =
self.parse_expression()
    block =
self.parse_block()
    return WhileStmt(cond,
block)

def
parse_expression(self):
    return
self.parse_comp()

def parse_comp(self):
    left =
self.parse_add()
    if self.current.value
in
('<', '<=', '>', '>=',
'==', '!='):
        op =
self.current.value;
self.eat('SYMBOL', op)

```

```

        right =
self.parse_add()
        return
BinaryOp(left, op, right)
        return left

    def parse_add(self):
        left =
self.parse_mul()
        while
self.current.value in
('+', '-'):
            op =
self.current.value;
self.eat('SYMBOL',op)
            right =
self.parse_mul()
            left =
BinaryOp(left, op, right)
            return left

    def parse_mul(self):
        left =
self.parse_factor()
        while
self.current.value in
('*','/'):
            op =
self.current.value;
self.eat('SYMBOL',op)
            right =
self.parse_factor()
            left =
BinaryOp(left, op, right)
            return left

    def parse_factor(self):
        tok = self.current
        if
tok.type=='NUMBER':
            self.eat('NUMBER')
        return
Number(tok.value)

        if tok.type=='IDENT':
            name = tok.value;
self.eat('IDENT')
            if
self.current.value == '(':
                args =
self.parse_args()
                return
FuncCall(name, args)
                return
Variable(name)
            if tok.value == '(':
                self.eat('SYMBOL
', '(')
                expr =
self.parse_expression()
                self.eat('SYMBOL
', ')')
                return expr
            self.error("Invalid
factor")

    def parse_args(self):
        self.eat('SYMBOL', '('
)
        args = []
        if
self.current.value != ')':
            args.append(self
.parse_expression())
            while
self.current.value == ',':
                self.eat('SY
MBOL', ',')
                args.append(
self.parse_expression())
                self.eat('SYMBOL', ')
')
        return args

if __name__ == '__main__':
    if len(sys.argv) >= 2:

```

```

data =
open(sys.argv[1],
encoding='utf-8').read()
else:
    data =
sys.stdin.read()
tokens =
Lexer(data).lex()
ast =
Parser(tokens).parse_program
()
with open('tokens.txt',
'w', encoding='utf-8') as f:
    for t in tokens:
        f.write(f"{t}\n")
with open('ast.txt', 'w',
encoding='utf-8') as f:
    f.write(repr(ast))

```

B.程序实例截图

The screenshot displays a development environment with the following components:

- Resource Manager:** Shows project structure with files like `test.rs`, `tokens.txt`, and `rust_parser.py`.
- Code Editors:**
 - rust_parser.py:** Contains Python code for a lexer and parser. It defines tokens like `i32`, `let`, `if`, `else`, `while`, `return`, `mut`, `fn`, `for`, `in`, `loop`, `break`, `continue`. It also defines two-character operators and single-character symbols.
 - test.rs:** Contains Rust code for a function `ret` that takes tokens and returns an AST.
 - tokens.txt:** Shows the tokens extracted from the input file, including keywords, identifiers, symbols, and numbers.
- Terminal:** Shows the command `python .\rust_parser.py test.rs` being run in a PowerShell-like terminal.

 i32 { if a >= b { return a; } else { return b; } } fn count_down(mut n:i32) { while n != 0 { n = n - 1; } }</div>