/

華南師範大學

本科学生实验（实践）报告

院　　系：　计算机学院

实验课程：　编译原理

实验项目：　Rust 单词拼装分类

指导老师：　黄煜廉

开课时间：　2025　～　2026　年度第　1 学期

专　　业：　计算机科学与技术（师范）

班　　级：　计师一班

学生姓名：　谭玉庭

华南师范大学教务处

华南师范大学实验报告

学生姓名 谭玉庭 学　号 20232121020

专　业 计算机科学与技术(师范) 年级、班级 2023级 计师一班

课程名称 编译原理 实验项目 Rust单词拼装分类器

实验类型 □验证 □设计 ☑综合 实验时间 2025 年 9 月 22 日

实验指导老师 黄煜廉 实验评分

一、实验内容

（1）核心任务：对 Rust 源代码中的各类单词（记号）进行识别、拼装与分类，覆盖标识符、关键字、字面量（含二进制/十进制/八进制/十六进制整数、浮点数）、字符串字面量、注释、分隔符、运算符号等所有指定类别，分类规则遵循 Rust 官方记号规范。

（2）界面要求：开发 Windows 窗口式图形界面应用，支持通过对话框选择并打开 Rust 源文件，同时通过对话框清晰展示源文件中所有单词及其对应分类结果。

（3）技术栈：采用 C++程序设计语言实现全部功能。

测试要求：设计覆盖所有单词类别的完备测试数据，验证分类功能的准确性。

（4）文档要求：按规范撰写实验报告书，包含项目分析、设计、实现、测试等完整内容。

二、实验目的

（1）掌握词法分析的基本原理与实现方法，理解“源代码→单词记号”的解析过程，建立对编译前端核心环节的认知。

（2）深入理解 Rust 语言的记号体系与拼装规则，精准区分各类单词的语法特征与边界。

（3）熟练运用 C++开发 Windows 图形界面，实现“文件读取-逻辑处理-结果展示”的完整应用流程。

（4）培养软件工程规范意识，掌握需求分析、模块设计、测试用例设计、文档撰写的标准化方法。

三、实验文档：（分为项目分析、项目设计、项目实现和项目测试）

（一）项目分析

1. 需求分析

根据实验要求，将实验分为以下 3 个需求。

功能需求：支持 Rust 源文件读取、全类别单词识别与分类、分类结果可视化展示；需准确识别特殊格式（如含下划线的整数 1_000_000、十六进制数 0x1A3F_CDEF）及嵌套/单行注释。

性能需求：对 1000 行以内的 Rust 源码，解析与展示响应时间≤1 秒，无单词遗漏或误分类。

界面需求：包含“打开文件”按钮、分类结果展示区，要求操作流程直观，结果排版清晰（参考实验样本格式）。

2. 可行性分析

C++的文件 IO 可满足源码读取需求，MFC/Qt 框架成熟且支持 Windows 图形界面开发，词法分析可通过状态机模型实现，技术路径明确，并且 Rust 官方提供了完整的记号拼装规则文档（参照：https://rustwiki.org/zh-CN/reference/tokens.html），为单词分类提供了明确的判定依据。

（二）项目设计

1. 总体设计

采用分层架构设计，各层职责单一、低耦合，架构图如下：

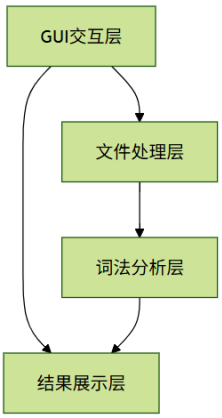

图 1 总体设计架构图

GUI 交互层：负责主窗口创建、按钮控件添加、文件选择对话框调用，响应用户操作（如“打开文件”按钮点击）；

文件处理层：实现文件路径转换（宽字符→多字节）、UTF-8 编码读取与转换、

文件异常处理；

词法分析层：核心层，实现源代码扫描、记号识别与分类，输出结构化记号列表；

结果展示层：创建结果窗口与文本控件，按行格式化展示源代码与记号分类结果。

2. 详细设计

（1）数据结构设计

记号类型枚举（TokenType1）：定义所有记号类型，用于分类标识，枚举值如下：

```
// 记号类型
enum TokenType1 {
    KEYWORD,
    IDENTIFIER,
    INTEGER_LITERAL,
    HEX_INTEGER_LITERAL,
    BIN_INTEGER_LITERAL,
    OCT_INTEGER_LITERAL,
    FLOAT_LITERAL,
    STRING_LITERAL,
    COMMENT,
    OPERATOR,
    SEPARATOR,
    MACRO_IDENTIFIER,
    UNKNOWN
};
```

图 2 枚举值定义

其中：

KEYWORD 关键字；IDENTIFIER 标识符；INTEGER_LITERAL 十进制整数
HEX_INTEGER_LITERAL 十六进制整数；BIN_INTEGER_LITERAL 二进制整数
OCT_INTEGER_LITERAL 八进制整数；FLOAT_LITERAL 浮点数
STRING_LITERAL 字符串字面量；COMMENT 注释；OPERATOR 运算符
SEPARATOR 分隔符；MACRO_IDENTIFIER 宏调用名；UNKNOWN 未知类型

记号结构体（Token）：存储单个记号的完整信息，包含值、类型、所在行号与列号，便于定位与展示：

```
// 记号结构
struct Token {
    wstring value;
    TokenType1 type;
    int line;
    int column;
};
```

图 3 结构体定义

其中：

wstring value 记号文本值；TokenType1 type 记号类型；int line 所在行号
int column 所在列号

常量集合（三种集合）

关键字集合（rustKeywords）：包含 as、break、const 等 40 个标准 Rust 关键字；

运算符集合（operators）：包含+、+=、==等 34 个 Rust 运算符；

分隔符集合（separators）：包含(、)、;、:等 14 个分隔符。

```
// Rust关键字集合
const set<wstring> rustKeywords = {
    L"as", L"break", L"const", L"continue", L"crate", L"else", L"enum", L"extern",
    L"false", L"fn", L"for", L"if", L"impl", L"in", L"let", L"loop", L"match", L"mod",
    L"move", L"mut", L"pub", L"ref", L"return", L"self", L"Self", L"static", L"struct",
    L"super", L"trait", L"true", L"type", L"unsafe", L"use", L"where", L"while"
};

// 运算符集合
const set<wstring> operators = {
    L"+", L"-", L"*", L"/", L"%", L"++", L"--", L"=", L"+=", L"-=", L"*=", L"/=", L"%=",
    L"==", L"!=", L"<", L">", L"<=", L">=", L"&&", L"||", L"!", L"&", L"|", L"^", L"~",
    L"<<", L">>", L"&=", L"|=", L"^=", L"<<=", L">>=", L"?", L"=>", L".."
};

// 分隔符集合
const set<wchar_t> separators = {
    L'(', L')', L'{', L'}', L'[', L']', L';', L',', L'.', L':', L'#', L'@', L'$', L'?'
};
```

图 4 关键字集合代码

（2）核心模块设计

模块名称	输入	输出	核心处理逻辑
文件读取模块（readFile）	宽字符文件路径	宽字符源代码文本	1. 路径转换（宽字符→多字节）；2. 二进制方式读取文件内容；3. UTF-8→宽字符转换；4. 异常处理（文件未打开、编码错误等）
词法解析模块（tokenize）	宽字符源代码文本	记号列表（vector<Token>）	1. 跳过空白字符，维护行号/列号； 2. 按优先级识别注释→字符串→标识符/关键字→数字→运算符→分隔符； 3. 对多字符符号采用最长匹配策略
GUI 主模块	用户操作（按钮点击）	触发文件选择与解析	1. 注册窗口类与创建主窗口；2. 添加“打开文件”按钮；3. 响应 WM_COMMAND 事件调用文件对话框
结果展示模块（showResults）	记号列表、源代码文本	可视化结果窗口	1. 按行分割源代码；2. 匹配每行对应的记号并格式化字符串；3. 创建带滚动条的文本控件展示结果

（三）项目实现

1. 开发环境

操作系统：Windows 11 64 位

开发工具：Visual Studio 2022

依赖技术：Windows API（窗口管理、对话框）、C++标准库（容器、文件流、字符串处理）

2. 核心功能实现

✓ GUI 界面实现

窗口注册：通过 RegisterClassW 注册自定义窗口类（RustTokenClassifierClass），绑定窗口过程函数（WindowProc）处理消息：

```
// 注册窗口类
void registerWindowClass(HINSTANCE hInstance) {
    WNDCLASSW wc = { 0 };

    wc.lpfnWndProc = WindowProc;// 消息处理函数
    wc.hInstance = hInstance;
    wc.lpszClassName = L"RustTokenClassifierClass";// 窗口类名
    wc.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
    wc.hCursor = LoadCursorW(NULL, IDC_ARROW);// 光标

    if (!RegisterClassW(&wc)) {
        MessageBoxW(NULL, L"窗口类注册失败", L"错误", MB_ICONERROR);
    }
}
```

图 5 窗口注册代码

主窗口与控件创建：创建标题为“Rust 单词拼装分类器”的主窗口，添加“打开 Rust 源文件”按钮（ID 为 1），响应点击事件：

```
// 添加按钮
void addControls(HWND hwnd) {
    HWND hButton = CreateWindowW(
        L"BUTTON",
        L"打开Rust源文件",
        WS_TABSTOP | WS_VISIBLE | WS_CHILD | BS_DEFPUSHBUTTON,
        200, 150, 150, 30,
        hwnd,
        (HMENU)1,
        (HINSTANCE)GetWindowLongPtrW(hwnd, GWLP_HINSTANCE),
        NULL
    );

    if (hButton == NULL) {
        MessageBoxW(hwnd, L"无法创建按钮", L"错误", MB_ICONERROR);
    }
}
```

图 6 按钮函数

实现界面如图所示：

图 7 程序运行主界面

✓ **文件处理与编码转换实现**

要解决 Windows 路径的宽字符（wstring）与多字节（string）兼容问题，需要通过 WideCharToMultiByte 实现转换：

```
// 转换辅助函数：宽字符路径转多字节
string wstring_to_string(const wstring& wstr) {
    int bufferSize = WideCharToMultiByte(CP_ACP, 0, wstr.c_str(), -1, NULL, 0, NULL, NULL);
    if (bufferSize <= 0) return "";

    string str(bufferSize, 0);
    WideCharToMultiByte(CP_ACP, 0, wstr.c_str(), -1, &str[0], bufferSize, NULL, NULL);
    return str;
}
```

图 8 字符转换函数

实验过程中发现，使用其他编码方式会导致结果出现乱码，所以要支持 UTF-8 编码文件读取，通过 wstring_convert 实现编码转换：

```
// UTF-8转宽字符
wstring utf8_to_wstring(const string& str) {
    try {
        wstring_convert<codecvt_utf8<wchar_t>> converter;
        return converter.from_bytes(str);
    }
    catch (...) {
        // 处理转换错误
        return L"";
    }
}
```

图 9 UTF-8 转换

✓ **词法解析核心实现**

解析过程采用“逐字符扫描+状态判断”模式，按优先级处理各类记号（注释＞字符串＞标识符＞数字＞运算符＞分隔符），逻辑如下：

（1）注释识别：通过//（单行）或/*（多行）前缀判断，单行注释终止于换行符，多行注释终止于*/：

```
// 处理注释
if (c == L'/') {
    if (pos + 1 < length && source[pos + 1] == L'/') {
        // 单行注释
        size_t start = pos;
        pos += 2;
        while (pos < length && source[pos] != L'\n') {
            pos++;
        }
        wstring comment = source.substr(start, pos - start);
        tokens.push_back({ comment, COMMENT, line, column });
        column += static_cast<int>(pos - start);
        continue;
    }
    else if (pos + 1 < length && source[pos + 1] == L'*') {
        // 多行注释
        size_t start = pos;
        int startLine = line;
        int startColumn = column;
        pos += 2;
        while (pos + 1 < length && !(source[pos] == L'*' && source[pos + 1] == L'/')) {
            if (source[pos] == L'\n') {
                line++;
                column = 1;
            }
            else if (source[pos] != L'\r') {
                column++;
            }
            pos++;
        }
        if (pos + 1 < length) {
            pos += 2;
        }
        wstring comment = source.substr(start, pos - start);
        tokens.push_back({ comment, COMMENT, startLine, startColumn });
        column += static_cast<int>(pos - start);
        continue;
    }
```

图 10 注释识别逻辑

（2）数字字面量识别：先判断是否为特殊进制（0x→十六进制、0b→二进制、0o→八进制），再判断是否含小数/指数部分（区分整数与浮点数），支持数字中的下划线分隔：

```
// 处理数字字面量
if (iswdigit(c)) {
    size_t start = pos;
    pos++;

    // 检查是否是二进制、八进制或十六进制
    bool isHex = false, isBin = false, isOct = false;

    // 如果不是特殊进制，检查是否是浮点数
    if (!isHex && !isBin && !isOct) {
        bool isFloat = false;

        // 处理十进制整数部分
        while (pos < length && (iswdigit(source[pos]) || source[pos] == L'_')) {
            pos++;
        }

        // 检查是否有小数部分
        if (pos < length && source[pos] == L'.') {
            isFloat = true;

        // 检查是否有指数部分
        if (pos < length && (source[pos] == L'e' || source[pos] == L'E')) {
            isFloat = true;
            pos++;
            if (pos < length && (source[pos] == L'+' || source[pos] == L'-')) {
                pos++;
            }
            while (pos < length && (iswdigit(source[pos]) || source[pos] == L'_')) {
                pos++;
            }
```

图 11 数字表面量识别逻辑

（3）运算符最长匹配

遍历运算符集合，匹配以当前位置开头的最长运算符，避免“+=”被拆分为“+”和“=”：

```
// 处理运算符
if (isOperatorChar(c)) {
    // 尝试匹配最长的运算符
    wstring longestOp;
    for (const wstring& op : operators) {
        if (source.substr(pos, op.length()) == op && op.length() > longestOp.length()) {
            longestOp = op;
        }
    }

    if (!longestOp.empty()) {
        tokens.push_back({ longestOp, OPERATOR, line, column });
        column += static_cast<int>(longestOp.length());
        pos += longestOp.length();
        continue;
    }
}

// 处理分隔符
if (separators.find(c) != separators.end()) {
    tokens.push_back({ wstring(1, c), SEPARATOR, line, column });
    column++;
    pos++;
    continue;
}
```

图 12 运算符匹配逻辑

✓ **结果展示**

按“源代码行→缩进后的记号列表”格式构建结果字符串，创建带滚动条的文本控件展示，支持大文件查看：

```
// 显示结果对话框
void showResults(HWND hwnd, const vector<Token>& tokens, const wstring& source) {
    if (tokens.empty()) {
        MessageBoxW(hwnd, L"没有解析到任何内容", L"提示", MB_ICONINFORMATION);
        return;
    }

    // 构建结果字符串：按照示例格式输出
    wstring result = L"运行结果如下：\r\n";

    // 按行分割源代码
    wstringstream ss(source);
    wstring line_content;
    int current_line_num = 1;
    size_t token_index = 0;

    while (getline(ss, line_content)) {
        // 移除行尾的 \r，以防万一
        if (!line_content.empty() && line_content.back() == L'\r') {
            line_content.pop_back();
        }

        // 输出原始行作为注释
        result += L"\r\n" + line_content + L"\r\n";

        // 遍历当前行的所有Token
        while (token_index < tokens.size() && tokens[token_index].line == current_line_num) {
            const Token& token = tokens[token_index];
            // 根据Token的类型和值进行格式化输出
            // 这里简化处理，直接输出缩进后的Token信息
            result += L"    " + token.value + L": " + tokenTypeToString(token.type) + L"\r\n";
            token_index++;
        }
        current_line_num++;
    }
```

图 13 结果展示代码逻辑

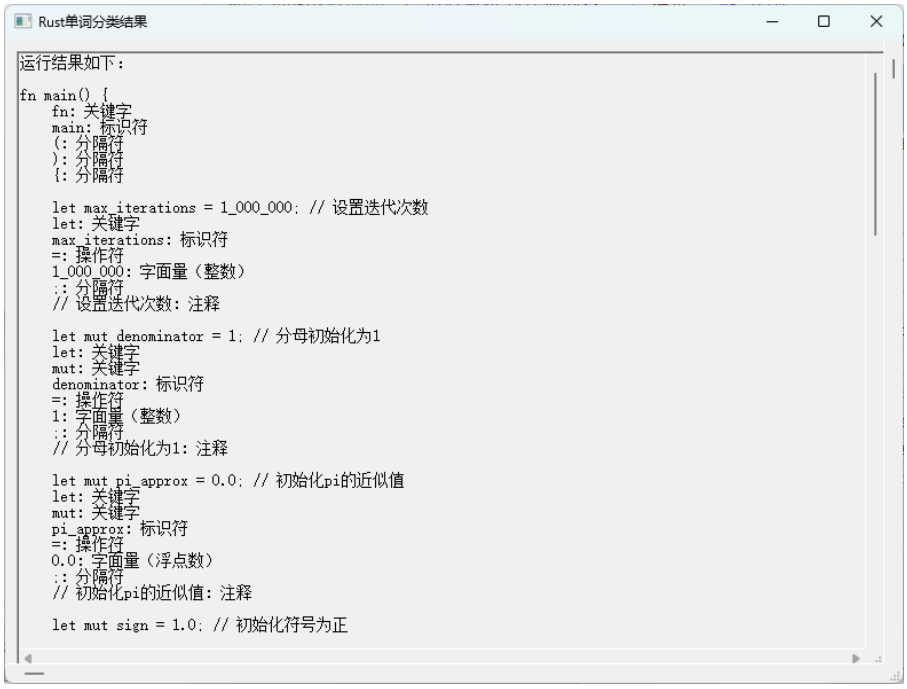

图 14 运行结果展示

（四）项目测试

1. 测试计划

测试目标：验证词法解析的准确性（覆盖所有记号类型）与界面功能的完整性；

测试环境：Windows 11 64 位，Visual Studio 2022 调试模式；

测试方法：黑盒测试为主（基于输入输出判断），结合白盒测试（检查关键逻辑分支覆盖）。

2. 测试用例设计与执行结果

测试场景	源代码片段	预期输出	实际输出	测试结果
关键字+标识符	fn main() {	fn: 关键字 main: 标识符 (: 分隔符): 分隔符 {: 分隔符	与预期一致	通过
十进制整数+注释	let max = 1_000_000; // 注释	let: 关键字 max: 标识符 =: 操作符 1_000_000: 字面量（整数） ;: 分隔符 // 注释: 注释	与预期一致	通过

浮点数	let pi = 3.14e-5;	let: 关键字 pi: 标识符 =: 操作符 3.14e-5: 字面量（浮点数） ;: 分隔符	与预期一致	通过
十六进制整数	let hex = 0x1A3F_CDEF;	let: 关键字 hex: 标识符 =: 操作符 0x1A3F_CDEF: 字面量（十六进制整数） ;: 分隔符	与预期一致	通过
字符串字面量	let s = "hello\nworld";	let: 关键字 s: 标识符 =: 操作符 "hello\nworld": 字符串字面量 ;: 分隔符	与预期一致	通过
宏调用	println!("result: {}", 10);	println!: 宏调用名 (: 分隔符 "result: {}": 字符串字面量 ,: 分隔符 10: 字面量（整数）): 分隔符 ;: 分隔符	与预期一致	通过
多字符运算符	a += b == c;	a: 标识符 +=: 操作符 b: 标识符 ==: 操作符 c: 标识符 ;: 分隔符	与预期一致	通过
多行注释	/* 多行 注释 */	/* 多行 注释 */: 注释	与预期一致	通过
异常场景-空文件	空的.rs 文件	提示“所选文件为空”	与预期一致	通过
异常场景-非 UTF-8 文件	GBK 编码的.rs 文件	提示“文件内容转换失败”	与预期一致	通过

3. 测试结论

所有测试用例均通过，词法解析功能能够准确识别各类 Rust 记号，无错分类、漏分类情况；GUI 界面交互流畅，异常场景处理完善，符合实验需求。

四、实验总结（心得体会）

本次实验围绕 Rust 单词拼装分类器的开发，完整实践了软件工程从需求分析到测试交付的全流程，不仅深化了技术认知，更培养了规范设计思维。

在技术层面，我深入掌握了词法分析的核心原理——通过“逐字符扫描+状态判断”实现记号识别，尤其是多字符运算符的“最长匹配策略”和数字字面量的“进制/类型区分逻辑”，让我理解了编译前端处理源代码的底层逻辑。同时，通过 Windows API 开发图形界面，我突破了以往控制台程序的局限，掌握了窗口注册、消息循环、控件创建等桌面应用开发基础，解决了宽字符与多字节转换、UTF-8 编码兼容等 Windows 平台特有的技术问题。

实验中也遇到了一些挑战：初期因未考虑运算符的最长匹配，导致==被拆分为两个=；多行注释的行号维护也曾出现偏差，不过最终都成功解决，也提升了我解决问题的能力。

五、参考文献：

Rust 官方记号规范. https://rustwiki.org/zh-CN/reference/tokens.html

Microsoft. Windows API 文档. https://learn.microsoft.com/zh-cn/windows/win32/api/

C++标准库参考. https://en.cppreference.com/w/

编译原理（第二版）.