编译原理实验报告

题目：类 Rust 语言词法与语法分析器设计（基于 LR(1)分析法）

学 生 姓 名 ： 李闯

学 号 ： 2253214

专 业 班 级 ： 计算机科学与技术 2 班

指 导 教 师 ： 卫志华、高珍

2025 年 04 月 29 日

一、实验说明

1. 实验概述

本次实验旨在设计并实现一个类 Rust 语言的词法与语法分析器，主要包含以下内容：

1) **词法分析器设计：**基于有限状态自动机思想，实现对源代码的扫描和分析，识别保留字、标识符、常量、运算符、界符和注释等词法单元，并生成对应的 Token 序列。

2) **语法分析器设计：**采用 LR(1)分析方法，构建 LR(1)分析表（Action 表和 Goto 表），指导语法分析过程并生成语法分析树。

1) **前端界面实现：**使用 Python 的 Tkinter 库开发交互式界面，展示代码编辑、语法规则、语法分析树及分析过程。语法树的绘制采用 Graphviz 工具，将分析生成的抽象语法树可视化为图像，并嵌入界面中展示。

2. 实验目标

通过本实验，深入理解词法分析和语法分析的核心原理，掌握 LR(1)分析法的实现细节，并能够设计符合 Rust 语言特性的文法规则。最终实现一个功能完整、交互友好的类 Rust 语言分析器，为后续的语义分析和代码生成奠定基础。

3. 实现功能

规则编号	规则类型	语法规则
0.1	变量声明	<变量声明内部> -> mut <ID>
0.2	类型系统	<类型> -> i32
0.3	可赋值元素	<可赋值元素> -> <ID>
1.1	程序结构	Program -> <声明串> <声明串> -> 空 | <声明> <声明串> <声明> -> <函数声明> <函数声明> -> <函数头声明> <语句块> <函数头声明> -> fn <ID> '(' <形参列表> ')' <形参列表>-> 空 <语句块> -> '{' <语句串> '}' <语句串> -> 空
1.2	语句基础	<语句串> -> <语句> <语句串> <语句>-> ';'
1.3	返回语句	<语句>-> <返回语句> <返回语句> -> return ';'
1.4	函数参数	<形参列表>-> <形参> | <形参> ',' <形参列表> <形参> -> <变量声明内部> ':' <类型>
1.5	函数返回值	<函数头声明> -> fn <ID> '(' <形参列表> ')' '->' <类型> <返回语句> -> return <表达式> ';'
2.1	变量声明语句	<语句>-> <变量声明语句> <变量声明语句> -> let <变量声明内部> ':' <类型> ';' <变量声明语句> -> let <变量声明内部> ';'
2.2	赋值语句	<语句>-> <赋值语句>

		<赋值语句> -> <可赋值元素> '='<表达式> ';'
		<语句>-> <变量声明赋值语句>
2.3	声明并赋值	<变量声明赋值语句> -> let <变量声明内部> ':' <类型> '='<表达式> ';'
		<变量声明赋值语句> -> let <变量声明内部> '='<表达式> ';'
		<语句> -> <表达式> ';'
		<表达式> -> <加法表达式>
3.1	基础表达式	<加法表达式> -> <项>
		<项 term> -> <因子>
		<因子 factor> -> <元素 element>
		<元素> -> <NUM> | <可赋值元素> | '(' <表达式> ')'
		<表达式> -> <表达式> <比较运算符> <加法表达式>
		<加法表达式> -> <加法表达式> <加减运算符> <项>
3.2	运算符扩展	<项> -> <项> <乘除运算符> <因子>
		<比较运算符> -> '<' | '<=' | '>' | '>=' | '==' | '!='
		<加减运算符> -> '+' | '-'
		<乘除运算符> -> '*' | '/'
3.3	函数调用	<元素> -> <ID> '(' <实参列表> ')'
		<实参列表>-> 空 | <表达式> | <表达式> ',' <实参列表>
		<语句> -> <if 语句>
4.1	选择结构	<if 语句> -> if <表达式> <语句块> <else 部分>
		<else 部分> -> 空 | else <语句块>
4.2	Else-if 扩展	<else 部分> -> else if <表达式> <语句块> <else 部分>
		<语句> -> <循环语句>
5.1	While 循环	<循环语句> -> <while 语句>
		<while 语句> -> while <表达式> <语句块>
		<循环语句> -> <for 语句>
5.2	For 循环	<for 语句>-> for <变量声明内部> in <可迭代结构> <语句块>
		<可迭代结构> -> <表达式> '..' <表达式>
5.3	Loop 循环	<循环语句> -> <loop 语句>
		<loop 语句>->loop <语句块>
5.4	循环控制	<语句> -> break ';' | continue ';'
6.1	不可变变量	<变量声明内部> -> <ID>
6.2	引用与解引用	<因子> -> '*' <因子> | '&' mut <因子> | '&' <因子>
		<类型>-> '&' mut <类型> | '&' <类型>
		<表达式> -> <函数表达式语句块>
7.1	表达式块	<函数表达式语句块>->'{' <函数表达式语句串> '}'
		<函数表达式语句串>-> <表达式> | <语句> <函数表达式语句串>
7.2	表达式函数体	<函数声明> -> <函数头声明> <函数表达式语句块>
7.3	选择表达式	<表达式> -> <选择表达式>
		<选择表达式> -> if <表达式> <函数表达式语句块> else <函数表达式语句块>
7.4	循环表达式	<表达式> -> <loop 语句>
		<语句> -> break <表达式> ';'
		<类型>-> '[' <类型> ';' <NUM>']'
8.1	数组类型	<因子> -> '[' <数组元素列表> ']' | <数组元素>
		<数组元素列表>-> 空 | <表达式> | <表达式> ',' <数组元素列表>
8.2	数组操作	<可赋值元素> -> <元素> '[' <表达式> ']'
		<可迭代结构> -> <数组元素> -> <可赋值元素>
		<类型> -> '(' <元组类型内部> ')'
9.1	元组类型	<元组类型内部> -> 空 | <类型> ',' <类型列表>
		<类型列表> ->空 | <类型> | <类型> ',' <类型列表>

		<因子> -> '(' <元组赋值内部> ')' <元组赋值内部> -> 空 | <表达式> ',' <元组元素列表> <元组元素列表>->空 | <表达式> | <表达式> ',' <元组元素列表>
9.2	元组访问	<可赋值元素>-><因子> '.' <NUM>

二、实验内容

1. 词法分析器(Lexer)设计

本实验的词法分析器采用手工编码的方式实现，通过有限状态自动机的思想对源代码进行扫描和分析。分析器能够识别类 Rust 语言中的保留字、标识符、常量、运算符、界符和注释等词法单元，并生成对应的 Token 序列输出。词法分析器整体运行流程图如图 1 所示。

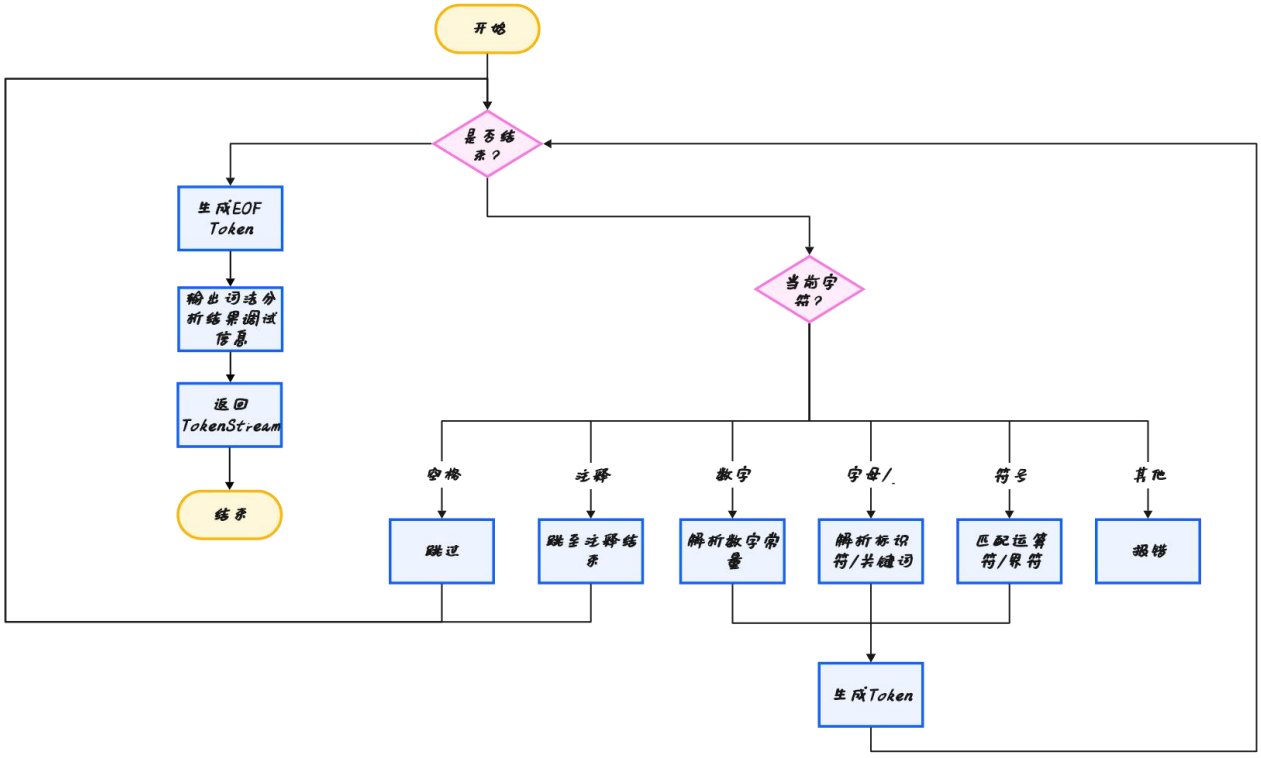


图 1 词法分析器分析过程流程图

1.1. 设计思路：

词法分析器通过三个核心组件实现高效、可靠的词法分析功能。首先，**TokenType 枚举类**明确定义了所有词法单元类型，包括关键字、运算符、界符等，为整个分析过程提供类型规范。其次，**Token 类**作为数据载体，封装了词素类型、原始值以及精确的源码位置信息，形成标准化的词法单元表示。最后，**Lexer 类**作为核心驱动，负责源代码的扫描和分析，通过条件判断实现确定性的状态转移。

TokenType 枚举类

```
class TokenType(Enum):
```

```
    """词法单元类型枚举"""
    # 关键词
    FN = 'fn'                # 函数
    LET = 'let'              # 变量声明
    I32 = 'i32'              # 类型：32 位整数
    # ...

    # 标识符
    IDENTIFIER = 'ID'

    # 常量
    INTEGER = 'NUM' # 整数
    FLOAT = 'NUM' # 浮点数
    STRING = 'STRING' # 字符串

    # 运算符
    PLUS = '+'           # +
    MINUS = '-'          # -
    MULTIPLY = '*'       # *
    DIVIDE = '/'         # /
    MOD = '%'            # %
    # ...

    # 界符和分隔符
    SEMICOLON = ';' # ;
    COMMA = ','     # ,
    # ...

    # 结束符
    EOF = '$'
```

Token 类

```
class Token:
    """
    Token 数据结构
    属性:
    - type: TokenType 枚举值
    - value: 原始词素值（仅标识符/常量需要）
    - line: 所在行号（1-based）
    - column: 所在列号（1-based）
    """
    def __init__(self, type_, value=None, line=None, column=None):
        self.type = type_
        self.value = value
        self.line = line
        self.column = column
```

Lexer 类处理源代码函数

```
def tokenize(self, text):
    """词法分析"""
    self.reset(text) # 先初始化
    tokens = []
    while((token := self.next_token()).type != TokenType.EOF):
        tokens.append(token)
    tokens.append(token)  # 加入 EOF
    self._log_lexer_result(tokens)
    return tokens

def next_token(self):
    while self.current_char is not None:
        # 跳过空格
        if self.current_char.isspace():
            self._skip_whitespace()
```

```
            continue
        # 注释
        if self.current_char == '/' and (self.peek() == '/' or self.peek() == '*'):
            self._skip_comment()
            continue
        # 数字
        if self.current_char.isdigit():
            return self._handle_number()
        # 字符串
        if self.current_char == '"':
            return self._handle_string()
        # 标识符和关键字
        if self.current_char.isalpha() or self.current_char == '_':
            return self._handle_identifier()
        # 运算符
        if operator_token := self._handle_operator():
            return operator_token
        # 界符和分隔符
        if delimiter_token := self._handle_delimiters():
            return delimiter_token
        # 其他字符
        self.error(f"Unknown character: {self.current_char}")

    return Token(TokenType.EOF, None, self.line, self.column)
```

在字符处理方面，分析器实现了精细的位置跟踪机制。**advance()方法**负责基础字符推进，自动处理换行符和制表符等特殊情况，确保行列计数准确无误。配合 **peek()方法**提供的单字符前瞻能力，系统能够准确识别多字符运算符（如==、->等）和复杂词法结构。

1.2. 核心处理功能实现：

词法分析的核心处理功能是通过一系列 _handle_* 方法实现的（具体实现见源代码 compiler_lexer.py 文件），每种方法负责处理特定类型的词法单元。具体功能划分如下表所示：

处理方法	功能描述	识别示例
_handle_number()	识别整数和浮点数，支持连续数字和小数点	42, 3.14
_handle_string()	处理字符串字面量，支持转义字符	"hello\n"
_handle_identifier()	识别标识符和关键字，通过预定义映射区分二者	foo, fn, mut
_handle_operator()	解析运算符，优先匹配多字符运算符	+=, ==, ->
_handle_delimiters()	识别界符，包括单字符和双字符形式	;, ::, ..
_skip_comment()	跳过单行和多行注释，不影响最终 Token 生成	// comment, /* comment */

对于关键字的识别，我采用了字典查询的方式，将预定义的关键字字符串映射到对应的 TokenType。这种方法相比硬编码的条件判断更加清晰和易于维护：

```
keyword_map = {
    'fn': TokenType.FN,
    'let': TokenType.LET,
    'mut': TokenType.MUT,
    # ...其他关键字
}
```

1.3. 类 Rust 特色处理：

① **类型系统支持**：将 i32 等类型标识符作为特殊关键字处理，与普通标识符区分开来。这为后续的类型检查奠定了基础。

② **复合运算符解析**：采用最长匹配原则，优先识别多字符运算符。例如："=="被识别为 EQUAL 运算符而非两个 ASSIGN。

③ **嵌套注释处理**：通过状态跟踪支持多行注释的嵌套，确保“*/* 外层注释 /* 内层注释 */ 继续外层 */*”形式的注释能被正确跳过。

④ **位置信息记录**：每个 Token 都记录了其出现的行号和列号，这不仅有助于调试，也为后续的语法错误提示提供了精准定位。

1.4. 词法分析结果调试验证：

词法分析器的分析结果通过_log_lexer_result()方法输出到日志文件中，该方法将源代码与生成的 Token 序列并排显示，方便验证分析结果。

以下是一个测试用例的运行示例：

```
fn program_2_1() {
    let mut a:i32;
    let mut b;
}
```

输出结果：

```
[INFO] [compiler_lexer.py:450] ====== LEXER RESULT START ======
[INFO] [compiler_lexer.py:456]     1 | fn program_2_1() {
[INFO] [compiler_lexer.py:452]           -> [FN] [IDENTIFIER:program_2_1] [LPAREN:(] [RPAREN:)] [LBRACE:{]
[INFO] [compiler_lexer.py:456]     2 |         let mut a:i32;
[INFO] [compiler_lexer.py:452]           -> [LET] [MUT] [IDENTIFIER:a] [COLON::] [I32] [SEMICOLON:;]
[INFO] [compiler_lexer.py:456]     3 |         let mut b;
[INFO] [compiler_lexer.py:452]           -> [LET] [MUT] [IDENTIFIER:b] [SEMICOLON:;]
[INFO] [compiler_lexer.py:456]     4 | }
[INFO] [compiler_lexer.py:452]           -> [RBRACE:}] [EOF]
[INFO] [compiler_lexer.py:457]======= LEXER RESULT END =======
```

通过这样的设计，我的词法分析器不仅能够准确识别类 Rust 语言的各类词法单元，还具备了良好的可扩展性和可维护性，为后续的语法分析阶段打下了坚实基础。

2. 语法分析器设计（LR(1)核心）

本实验的语法分析器采用 **LR(1)分析方法**实现，能够处理类 Rust 语言的语法结构。分析器通过构建 **LR(1)分析表（Action 表和 Goto 表）**来指导语法分析过程，最终生成语法分析树。下面从核心数据结构、分析表构建和语法分析与语法树构建流程三个方面进行详细说明。

2.1. 核心数据结构：

① **LR(1)项目：**

在本次实验中，我使用 namedtuple 定义 LR1Item 结构来表示 **LR(1) 项目**，其包含四个字

段：prod_lhs（产生式左部）、prod_rhs（产生式右部）、dot_pos（点的位置）、lookahead（向前看符号）。

选择将其定义为不可变的元组结构，主要是为了保证其**可哈希性**，从而可以在后续计算闭包（closure）时，能够高效地使用字典存储和检索状态。因为 Python 的集合和字典要求键值必须是可哈希对象，因此采用元组形式存储 LR(1) 项目能够确保它们可以作为字典的键或集合的元素，进而优化状态管理和查重效率。

```
LR1Item = namedtuple('LR1Item', ['prod_lhs', 'prod_rhs', 'dot_pos', 'lookahead'])
```

② **语法树节点：**

在语法分析阶段需要可视化一棵语法分析树（Parse Tree）来展示语法分析的结果，为此定义了 ParseNode 类来存储语法树的节点信息，

每个节点通过 symbol 字段标识其语法类型，可能是关键字、运算符等终结符，也可能是表达式、语句等非终结符。对于终结符节点，token 字段关联了词法分析阶段生成的词法单元，保存了原始字符串值及其在源代码中的位置信息，而非终结符节点则通过 children 字段维护其子节点列表，形成层次化的树状结构。

这种数据结构使得语法树既能准确反映程序的语法结构，又能保留必要的词法信息，为后续的阶段奠定了坚实基础。无论是进行语义检查、生成中间代码，还是最终的目标代码生成，都可以通过遍历这棵语法树来获取所需的信息。同时，清晰的节点结构也使得语法树的可视化和调试变得更加方便。

```
class ParseNode:
    def __init__(self, symbol, children=None, token=None):
        """语法分析树节点"""
        self.symbol = symbol  # 非终结符名或终结符类型
        self.children = children or []
        self.token = token
        self.value = token.value if token else None
        self.line = token.line if token else -1
        self.column = token.column if token else -1
```

③ **文法存储：**

在语法分析器的实现中，文法的存储方式直接影响语法分析的效率和可维护性。我采用结构化的字典来存储文法的各个组成部分，这种设计使得文法的各个要素能够被清晰地组织并快速访问。字典中包含四个关键字段：terminals 存储所有终结符的集合，如运算符和标识符；non_terminals 记录所有非终结符，代表语法中的抽象结构；productions 以列表形式保存所有产生式规则，每条规则包含左部符号和右部符号序列；start_symbol 则指明文法的起始符号，作为语法分析的入口点。

这种结构使得代码具有可读性和可维护性，当需要修改或扩展文法时，只需在字典结构中增减相应内容即可，无需改动核心分析算法。

```
{
```

```
    'terminals': {'+', '*', '(', ')', 'id'},
    'non_terminals': {'E', 'T', 'F'},
    'productions': [
        {'prod_lhs': 'E', 'prod_rhs': ['E', '+', 'T']},
        # 其他产生式...
    ],
    'start_symbol': 'E'
}
```

2.2. LR(1)分析表构建：

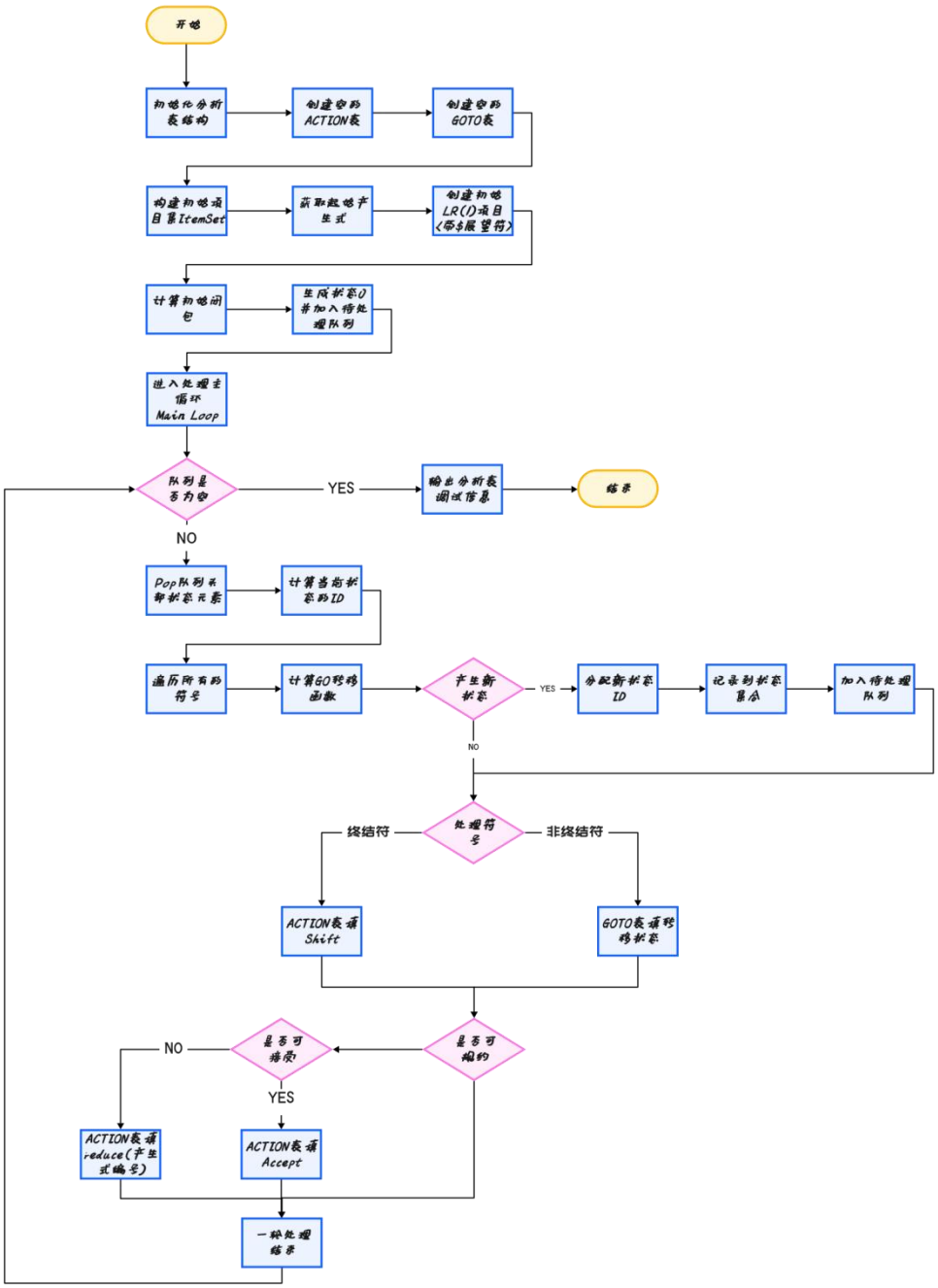


图 2 LR(1)分析表构建流程图

LR(1)分析表的构建是语法分析器的核心环节，其目标是通过计算项目集闭包（closure）和状态转移关系，生成确定性的 **ACTION 表和 GOTO 表**，以指导后续的移入-规约分析过程。

如图 2 所示，算法从文法的起始符号出发，构造初始 LR(1)项目集，并通过计算闭包生成第一个分析状态。随后，采用广度优先的策略遍历所有可能的符号转移，对每个状态和符号组合计算 GO(I,X)函数，生成新的项目集闭包作为后续状态。在此过程中，移入动作会被记录到 ACTION 表，而非终结符的转移目标则填入 GOTO 表。同时，算法会检查每个状态中的完成项目（即点号位于产生式末尾的项目），在 ACTION 表中填入规约或接受动作。最终，当所有状态处理完毕时，分析表构建完成，可以用于驱动语法分析过程。

关键函数实现：

- **闭包（closure）计算：**

闭包计算的作用是将当前项目集扩展为完整的等价项目集。该函数的实现的步骤为：

1) 初始化闭包集合，将输入项目转换为可哈希的 LR1Item 对象
2) 通过迭代方式不断扩展项目集，直到没有新项目可以添加为止
3) 对于每个项目[A→α · Bβ,a]，如果 B 是非终结符，则将所有 B 的产生式[B→ · γ]加入闭包，其中展望符为 FIRST(βa)
4) 最终对项目集进行排序以保证确定性

```
def closure(self, items):
    """
    计算项目集的闭包(扩展当前项目集)
    核心思想：将所有可能通过 非终结符转移 触发的子项目纳入当前状态。
    :param items: 项目集 [{'production': 产生式, 'dot_pos': 位置, 'lookahead': 向前看符号}]
    :return: 闭包后的项目集 (LR1Item)
    """
    # 将项目闭包转换为集合以便于快速查看生成的新项目是否已经存在
    closure_set = set()

    for item in items:
        if isinstance(item, dict):
            closure_set.add(self.dict_to_lr1_item(item))
        else:
            closure_set.add(item)
    changed = True
    while changed:
        changed = False
        for item in list(closure_set):
            # [A -> α.Bβ, a]
            # 如果点在产生式的右侧，且点后还有符号
            if item.dot_pos < len(item.prod_rhs):
                B = item.prod_rhs[item.dot_pos]
                # 对于每个产生式 B -> γ，添加到闭包中
                if B in self.non_terminals:
                    beta = item.prod_rhs[item.dot_pos+1:] # β
                    # 计算 FIRST(βa)
                    lookaheads = self.first(beta + (item.lookahead,)) if beta else {item.lookahead}
                    # 添加新项目
                    for prod in self.productions[B]:
```

```
                        for la in lookaheads:
                            new_item = LR1Item(
                                prod_lhs=B,
                                prod_rhs=tuple(prod['rhs']),
                                dot_pos=0,
                                lookahead=la
                            )
                            if new_item not in closure_set:
                                closure_set.add(new_item)
                                changed = True
    # 排序逻辑：按 prod_lhs → prod_rhs → dot_pos → lookahead 的优先级
    sorted_items = sorted(
        closure_set,
        key=lambda x: (x.prod_lhs, x.prod_rhs, x.dot_pos, x.lookahead)
    )
    return tuple(sorted_items)
```

- **状态转移函数（go）：**

状态转移函数定义了从一个项目集通过某个符号转移到另一个项目集的规则：

1) 遍历当前项目集中所有项目，找出点号后紧跟给定符号的项目

2) 将这些项目的点号位置后移一位，形成新的基础项目集

3) 对新生成的项目集计算闭包，得到完整的状态

```
def go(self, items, symbol):
    """
    状态转移函数
    :param items: 项目集
    :param symbol: 符号
    :return: 计算转移后的项目集(状态)
    """
    new_items = set()
    for item in items:
        # 当前的项目 Item
        current  =self.dict_to_lr1_item(item) if isinstance(item, dict) else item
        if current.dot_pos < len(current.prod_rhs) and current.prod_rhs[current.dot_pos] == symbol:
            new_item = LR1Item(
                prod_lhs=current.prod_lhs,
                prod_rhs=current.prod_rhs,
                dot_pos=current.dot_pos + 1,  # 移动点位置
                lookahead=current.lookahead
            )
            new_items.add(new_item)
    return self.closure(new_items) if new_items else None
```

- **First 集计算：**

First 集计算是语法分析的基础性工作，其计算过程如下所示：

1) 采用递归方式计算符号串的 First 集合

2) 对终结符直接返回其本身

3) 对非终结符需要考虑其所有产生式右部的 First 集合

4) 实现中加入了缓存机制和循环依赖检测来优化性能

```python
def first(self, symbols, visited=None):
    """
    计算符号串的 FIRST 集合
    :param symbols: 符号串
    :return: FIRST 集合
    """
    if visited is None:
        visited = set()
    cache_key = tuple(symbols)
    if cache_key in self._first_cache:
        return self._first_cache[cache_key].copy()

    # 检测循环依赖
    if cache_key in visited:
        return set()
    visited.add(cache_key)
    first_set = set()
    # 处理空串''
    if not symbols:
        first_set.add('')
        return first_set

    for symbol in symbols:
        # 处理终结符
        if symbol in self.terminals or symbol == '$':
            first_set.add(symbol)
            first_set.discard('') # 终结符会阻塞ε传播
            break

        # 处理非终结符
        elif symbol in self.non_terminals:
            # ['A']
            has_epsilon = False # 假设 FIRST(A)不存在ε 用于拦截ε
            for prod in self.productions.get(symbol, []):
                # A -> ε
                if not prod['rhs']:
                    first_set.add('')
                    has_epsilon = True
                    continue

                # 递归计算产生式右部的 FIRST 集
                # A -> B C D
                sub_first = self.first(prod['rhs'], visited.copy())
                first_set.update(sub_first - {''})
                if '' in sub_first:
                    has_epsilon = True

            # 若 FIRST(A)的不包含ε 需要将ε移除并且需要跳出循环
            if not has_epsilon:
                first_set.discard('')
                break

        # 未知类型
        else :
            raise ValueError(f"未知符号类型: {symbol}")

    self._first_cache[cache_key] = first_set
    return first_set
```

2.3. 语法分析与语法树构建过程

LR(1)语法分析器的核心工作流程通过 parse 函数实现，该函数严格遵循 LR(1)自动机的状

态转移规则。分析过程维护两个核心数据结构：**状态栈**记录自动机的状态转移路径，**节点栈**存储已构建的语法树节点。算法通过查 ACTION 表决定当前应采取的动作，遇到终结符时执行移入操作并创建叶子节点，当识别到完整产生式时执行规约操作并构建语法树中间节点，最终在遇到接受动作时返回完整的语法分析树。

语法树的构建与分析过程同步进行，体现了自底向上的构建策略。移入操作（shift）会为终结符创建 ParseNode 叶子节点，保留原始 Token 信息（包括行列位置和字面值）；规约操作（reduce）则会从节点栈中弹出右部符号对应的子节点，组合成新的非终结符节点。

语法分析与语法树构建过程

```
def parse(self, tokens):
    """
    执行 LR(1)语法分析
    :param tokens: Token 对象列表
    :return: 语法分析树根节点
    """
    stack = [0]                     # 状态栈，初始状态为 0
    node_stack = []                 # 节点栈
    idx = 0                         # 当前 token 指针
    # eof_token = Token(type_='$')  # 结束标记
    token_stream = list(tokens)

    while True:
        state = stack[-1]
        current_token = token_stream[idx]
        # 1. 查 ACTION 表
        # logger.info(current_token.type)
        action = self.action_table[state].get(current_token.type.value)
        if not action:
            self._log_state_items(state, self.states[state])
            expected = sorted(self.action_table[state].keys())
            context = token_stream[max(0, idx-2):idx+1]
            raise SyntaxError(
                f"语法错误（第{current_token.line}行，第{current_token.column}列）\n"
                f"意外 Token: {current_token}\n"
                f"期望: {expected}\n"
                f"上下文: {context}"
            )

        # 2. 执行动作
        if action[0] == 'shift':
            # 移入：压入新状态和当前符号
            stack.append(action[1])
            # 终结符
            node_stack.append(ParseNode(
                symbol=current_token.type.name,
                token=current_token,  # 存储完整 Token 对象
                children=None
            ))
            idx += 1
            logger.info(f"移入: {current_token} -> 状态{action[1]}")
        elif action[0] == 'reduce':
            # 规约：应用产生式
            prod_index = action[1] # 产生式编号
            prod = self.prod_by_index[prod_index]
            lhs = prod['lhs'] # 字符串
            rhs_len = len(prod['rhs'])
            # 弹出子节点
```

```
            children = []
            if rhs_len > 0:
                stack = stack[:-rhs_len]
                children = node_stack[-rhs_len:]
                node_stack = node_stack[:-rhs_len]
            # 创建新节点
            # 非终结符
            new_node = ParseNode(
                symbol=lhs,
                children=children
            )
            node_stack.append(new_node)
            # GOTO 转移
            goto_state = self.goto_table[stack[-1]].get(lhs)
            if goto_state is None:
                raise SyntaxError(f"无效 GOTO：状态{stack[-1]}遇到{lhs}")
            stack.append(goto_state)

            logger.info(f"规约: {lhs} → {' '.join(prod['rhs']) if prod['rhs'] else 'ε'}")
        elif action[0] == 'accept':
            logger.success("✓ 语法分析完成")
            return node_stack[0]  # 返回语法树根节点

        else:
            raise SyntaxError(f"Invalid action: {action}")
```

2.4. 文法（Grammar）设计：

在本次实验的文法设计中，我依据自身的理解将表达式系统划分为两大类：**位置表达式（Place Expression）和值表达式（Value Expression）**，也可以将其称为**左值（Left Value）和右值（Right Value）**。这种分类方式源于对程序语义的理解，位置表达式代表内存中可被赋值的具体位置，即 PPT 中的<可赋值元素>，包括变量标识符、指针解引用、数组索引等左值；而值表达式代表具体的计算结果，涵盖各种运算符组合形成的右值。

在值表达式的设计过程中，我特别注重运算符优先级的层次化处理，并且在参考标准课件时，我发现诸如“元素”、“因子”等属于容易造成概念混淆，为此，我采用了更直观的命名方式，直接以运算类型作为非终结符的名称，这使得文法规则既易于理解，又可以准确反映了运算的优先级关系，比如乘法运算的结果作为因子（multiplicative_expr）参与到加法运算，这种层级关系自然地体现了“先乘除后加减”的运算规则。依照这种层次化处理的思想可以加入其他各种不同层级的运算，因此我在 PPT 给出的文法基础之上简单地加入了逻辑或（||）和逻辑与（&&）。

依据自身理解重新命名的文法

```
# 依据自身理解 重新命名的文法
RUST_GRAMMAR = {
    # 终结符需要自行定义 出现在左侧的符号加入到非终结符中
    'terminals' : {
        # 关键字
        'fn', 'mut', 'return', '->', 'let', 'if', 'else', 'while', 'for', 'loop', 'break', 'continue',
'in',
        # 类型
        'i32',
        # 标识符和字面量
        'ID', 'NUM',
```

```
        # 运算符
        '+', '-', '*', '/', '%', '&',
        '==', '!=', '<', '<=', '>', '>=',
        '||', '&&',
        # 界符
        '(', ')', '[', ']', '{', '}', ';', ',', ':', '=', '.', '..',
    },
    # 可以通过产生式自动生成
    'non_terminals' : {
    },
    # 每一项是一个产生式 是一推一的关系
    'productions' : [
        # 0. 基础结构(Basic Construct)
        {'prod_lhs': 'Begin', 'prod_rhs': ['program']},
        {'prod_lhs': 'program', 'prod_rhs': ['declaration_list']},
        {'prod_lhs': 'declaration_list', 'prod_rhs': ['declaration', 'declaration_list']},
        {'prod_lhs': 'declaration_list', 'prod_rhs': []},
        {'prod_lhs': 'declaration', 'prod_rhs': ['function_declaration']},

        # 1. 函数声明(Function Declaration)
        {'prod_lhs': 'function_declaration', 'prod_rhs': ['function_header',
'block']},               # 块
        {'prod_lhs': 'function_declaration', 'prod_rhs': ['function_header',
'expr_block']},          # 表达式块
        {'prod_lhs':'function_header','prod_rhs':['fn','ID','(','param_list',')','return_type']},
        {'prod_lhs': 'return_type', 'prod_rhs': ['->', 'type']},
        {'prod_lhs': 'return_type', 'prod_rhs': []},
        {'prod_lhs': 'param_list', 'prod_rhs': ['param']},
        {'prod_lhs': 'param_list', 'prod_rhs': ['param', ',', 'param_list']},
        {'prod_lhs': 'param_list', 'prod_rhs': []},
        {'prod_lhs': 'param', 'prod_rhs': ['variable_declaration', ':', 'type']},

        # 2. 块(Block) & 表达式块(Expression_Block)
        {'prod_lhs': 'block', 'prod_rhs': ['{', 'statement_list', '}']},
        {'prod_lhs': 'statement_list', 'prod_rhs': []},
        {'prod_lhs': 'statement_list', 'prod_rhs': ['statement_with_semi', 'statement_list']},
        {'prod_lhs': 'expr_block', 'prod_rhs': ['{', 'statement_list_expression', '}']},
        {'prod_lhs': 'statement_list_expression', 'prod_rhs': ['bare_expression_statement']},
        {'prod_lhs': 'statement_list_expression', 'prod_rhs': ['statement_with_semi',
'statement_list_expression']},
        {'prod_lhs':'loop_expr_block','prod_rhs':['{','statement_list','break_statement_with_expr',
'}']},

        # 3. 变量和类型
        {'prod_lhs': 'variable_declaration', 'prod_rhs': ['mut', 'ID']}, # 可变变量声明
        {'prod_lhs': 'variable_declaration', 'prod_rhs': ['ID']},        # 不可变变量声明 -- 需要修改
        {'prod_lhs': 'type', 'prod_rhs': ['i32']},
        {'prod_lhs': 'type', 'prod_rhs': ['[', 'type', ';', 'NUM', ']']},
        {'prod_lhs': 'type', 'prod_rhs': ['(', 'tuple_type_inner', ')']},
        {'prod_lhs': 'type', 'prod_rhs': ['&', 'mut', 'type']}, # 可变引用
        {'prod_lhs': 'type', 'prod_rhs': ['&', 'type']},        # 不可变引用

        {'prod_lhs': 'tuple_type_inner', 'prod_rhs': []},
        {'prod_lhs': 'tuple_type_inner', 'prod_rhs': ['type', ',', 'tuple_type_list']},
        {'prod_lhs': 'tuple_type_list', 'prod_rhs': []},
        {'prod_lhs': 'tuple_type_list', 'prod_rhs': ['type']},
        {'prod_lhs': 'tuple_type_list', 'prod_rhs': ['type', ',', 'tuple_type_list']},

        # 4. 语句(Statement)
        {'prod_lhs': 'statement', 'prod_rhs': ['statement_with_semi']},        # 普通语句（带分号）
        {'prod_lhs': 'statement', 'prod_rhs': ['bare_expression_statement']}, # 表达式语句（不带分号）
```

```
        {'prod_lhs': 'statement_with_semi', 'prod_rhs': ['variable_declaration_stmt']},                #
变量声明语句
        {'prod_lhs': 'statement_with_semi', 'prod_rhs': ['variable_declaration_assignment_stmt']},   #
变量声明赋值语句
        {'prod_lhs': 'statement_with_semi', 'prod_rhs': ['assignment_stmt']},                          #
赋值语句
        {'prod_lhs': 'statement_with_semi', 'prod_rhs': ['return_statement']},                         #
返回语句
        {'prod_lhs': 'statement_with_semi', 'prod_rhs': ['if_stmt']},                                  #
if 语句
        {'prod_lhs': 'statement_with_semi', 'prod_rhs': ['loop_stmt']},                                #
循环语句
        {'prod_lhs': 'statement_with_semi', 'prod_rhs': ['break_statement']},                          #
break 语句
        {'prod_lhs': 'statement_with_semi', 'prod_rhs': ['continue_statement']},                       #
continue 语句
        {'prod_lhs': 'statement_with_semi', 'prod_rhs': [';']},                                        #
空语句（分号）

        # 4.1. 表达式语句
        {'prod_lhs': 'statement_with_semi', 'prod_rhs': ['bare_expression_statement', ';']},  # 表达式
语句(有分号)
        {'prod_lhs': 'bare_expression_statement', 'prod_rhs': ['value_expr']},                # 表达式
语句(无分号)
        # 4.2. 变量声明语句
        {'prod_lhs': 'variable_declaration_stmt', 'prod_rhs': ['let', 'variable_declaration', ':',
'type', ';']},
        {'prod_lhs': 'variable_declaration_stmt', 'prod_rhs': ['let', 'variable_declaration', ';']},
        {'prod_lhs': 'variable_declaration_assignment_stmt', 'prod_rhs': ['let',
'variable_declaration', '=', 'value_expr', ';']},
        {'prod_lhs': 'variable_declaration_assignment_stmt', 'prod_rhs': ['let',
'variable_declaration', ':', 'type', '=', 'value_expr', ';']},
        # 4.3. 赋值语句
        {'prod_lhs': 'assignment_stmt', 'prod_rhs': ['place_expr', '=', 'value_expr', ';']},
        # 4.4. 返回语句
        {'prod_lhs': 'return_statement', 'prod_rhs': ['return', ';']},
        {'prod_lhs': 'return_statement', 'prod_rhs': ['return', 'value_expr', ';']},
        # 4.5. if 语句
        {'prod_lhs': 'if_stmt', 'prod_rhs': ['if', 'value_expr', 'block', 'else_part']},
        {'prod_lhs': 'else_part', 'prod_rhs': []},
        {'prod_lhs': 'else_part', 'prod_rhs': ['else', 'block']},
        {'prod_lhs': 'else_part', 'prod_rhs': ['else', 'if_stmt']},
        # 4.6. 循环语句
        {'prod_lhs': 'loop_stmt', 'prod_rhs': ['while', 'value_expr', 'block']},
        {'prod_lhs': 'loop_stmt', 'prod_rhs': ['for', 'variable_declaration', 'in',
'iterable_structure', 'block']},
        {'prod_lhs': 'loop_stmt', 'prod_rhs': ['loop', 'block']},
        # 4.7. break 语句
        {'prod_lhs': 'break_statement', 'prod_rhs': ['break_statement_with_expr']},
        {'prod_lhs': 'break_statement', 'prod_rhs': ['break_statement_without_expr']},
        {'prod_lhs': 'break_statement_with_expr', 'prod_rhs': ['break', 'value_expr', ';']},
        {'prod_lhs': 'break_statement_without_expr', 'prod_rhs': ['break', ';']},
        # 4.8. continue 语句
        {'prod_lhs': 'continue_statement', 'prod_rhs': ['continue', ';']},

        # 可迭代结构
        {'prod_lhs': 'iterable_structure', 'prod_rhs': ['value_expr', '..', 'value_expr']},
        {'prod_lhs': 'iterable_structure', 'prod_rhs': ['value_expr']},

        # 5. 表达式(Expression)
        # Expressions are divided into two main categories: place expressions and value expressions
        # - A place expression is an expression that represents a memory location.
        # - A value expression is an expression that represents an actual value.
```

```
        # Note: Historically, place expressions were called lvalues and value expressions were called
rvalues.
        # 5.1 Place Expression (左值表达式)
        {'prod_lhs': 'place_expr', 'prod_rhs': ['place_expr_base']},
        {'prod_lhs': 'place_expr', 'prod_rhs': ['*', 'place_expr']},  # 指针解引用
        {'prod_lhs': 'place_expr_base', 'prod_rhs': ['ID']},
        {'prod_lhs': 'place_expr_base', 'prod_rhs': ['(', 'place_expr', ')']},
        {'prod_lhs': 'place_expr_base', 'prod_rhs': ['place_expr_base', '[', 'value_expr', ']']},  # 数
组索引
        {'prod_lhs': 'place_expr_base', 'prod_rhs': ['place_expr_base', '.', 'NUM']},              # 字
段访问
        # 5.2 Value Expression (右值表达式)
        # 表达式层次：条件 → 逻辑或 → 逻辑与 → 关系 → 加减 → 乘除 → 一元 → 后缀 → 基本
        {'prod_lhs': 'value_expr', 'prod_rhs': ['[', 'array_element_list', ']']},   # 数组字面量
        {'prod_lhs': 'value_expr', 'prod_rhs': ['(', 'tuple_element_inner', ')']},  # 元组字面量
        {'prod_lhs': 'value_expr', 'prod_rhs': ['logical_or_expr']},
        {'prod_lhs': 'value_expr', 'prod_rhs': ['conditional_expr']},
        # 条件表达式 if condition { branch_1 } else { branch_2 }
        {'prod_lhs': 'conditional_expr', 'prod_rhs': ['logical_or_expr']},
        {'prod_lhs': 'conditional_expr', 'prod_rhs': ['if', 'logical_or_expr', 'expr_block', 'else',
'expr_block']},
        # 逻辑或
        {'prod_lhs': 'logical_or_expr', 'prod_rhs': ['logical_or_expr', 'logic_or_op',
'logical_and_expr']},
        {'prod_lhs': 'logical_or_expr', 'prod_rhs': ['logical_and_expr']},
        # 逻辑与
        {'prod_lhs': 'logical_and_expr', 'prod_rhs': ['logical_and_expr', 'logic_and_op',
'relational_expr']},
        {'prod_lhs': 'logical_and_expr', 'prod_rhs': ['relational_expr']},
        # 关系表达式
        {'prod_lhs': 'relational_expr', 'prod_rhs': ['relational_expr', 'relational_op',
'additive_expr']},
        {'prod_lhs': 'relational_expr', 'prod_rhs': ['additive_expr']},
        # 加减表达式
        {'prod_lhs': 'additive_expr', 'prod_rhs': ['additive_expr', 'additive_op',
'multiplicative_expr']},
        {'prod_lhs': 'additive_expr', 'prod_rhs': ['multiplicative_expr']},
        # 乘除表达式
        {'prod_lhs': 'multiplicative_expr', 'prod_rhs': ['multiplicative_expr', 'multiplicative_op',
'unary_expr']},
        {'prod_lhs': 'multiplicative_expr', 'prod_rhs': ['unary_expr']},
        # 一元表达式
        {'prod_lhs': 'unary_expr', 'prod_rhs': ['unary_op', 'unary_expr']},
        {'prod_lhs': 'unary_expr', 'prod_rhs': ['postfix_expr']},
        {'prod_lhs': 'unary_expr', 'prod_rhs': ['NUM']},
        # 后缀表达式
        {'prod_lhs': 'postfix_expr', 'prod_rhs': ['postfix_expr', '(', 'argument_list', ')']},  # 函数
调用
        {'prod_lhs': 'postfix_expr', 'prod_rhs': ['primary_expr']},
        # 基本表达式
        {'prod_lhs': 'primary_expr', 'prod_rhs': ['place_expr']},
        {'prod_lhs': 'primary_expr', 'prod_rhs': ['(', 'value_expr', ')']}, # 加括号()
        {'prod_lhs': 'primary_expr', 'prod_rhs': ['expr_block']},           # 表达式块{}
        # 循环表达式
        {'prod_lhs': 'primary_expr', 'prod_rhs': ['loop_expr']},
        {'prod_lhs': 'loop_expr', 'prod_rhs': ['loop', 'loop_expr_block']},

        # 数组元素列表
        {'prod_lhs': 'array_element_list', 'prod_rhs': []},
        {'prod_lhs': 'array_element_list', 'prod_rhs': ['value_expr']},
        {'prod_lhs': 'array_element_list', 'prod_rhs': ['value_expr', ',', 'array_element_list']},
        # 元组元素列表
        {'prod_lhs': 'tuple_element_inner', 'prod_rhs': []},
```

```
        {'prod_lhs': 'tuple_element_inner', 'prod_rhs': ['value_expr', ',', 'tuple_element_list']},
        {'prod_lhs': 'tuple_element_list', 'prod_rhs': []},
        {'prod_lhs': 'tuple_element_list', 'prod_rhs': ['value_expr']},
        {'prod_lhs': 'tuple_element_list', 'prod_rhs': ['value_expr', ',', 'tuple_element_list']},
        # 实参列表
        {'prod_lhs': 'argument_list', 'prod_rhs': []},
        {'prod_lhs': 'argument_list', 'prod_rhs': ['value_expr']},
        {'prod_lhs': 'argument_list', 'prod_rhs': ['value_expr', ',', 'argument_list']},

        # 6. 运算符
        # 6.1. 关系运算符
        {'prod_lhs': 'relational_op', 'prod_rhs': ['==']},
        {'prod_lhs': 'relational_op', 'prod_rhs': ['!=']},
        {'prod_lhs': 'relational_op', 'prod_rhs': ['<']},
        {'prod_lhs': 'relational_op', 'prod_rhs': ['<=']},
        {'prod_lhs': 'relational_op', 'prod_rhs': ['>']},
        {'prod_lhs': 'relational_op', 'prod_rhs': ['>=']},
        # 6.2. 加减运算符
        {'prod_lhs': 'additive_op', 'prod_rhs': ['+']},
        {'prod_lhs': 'additive_op', 'prod_rhs': ['-']},
        # 6.3. 乘除运算符
        {'prod_lhs': 'multiplicative_op', 'prod_rhs': ['*']}, # 乘号
        {'prod_lhs': 'multiplicative_op', 'prod_rhs': ['/']},
        {'prod_lhs': 'multiplicative_op', 'prod_rhs': ['%']},
        # 6.4. 一元运算符
        {'prod_lhs': 'unary_op', 'prod_rhs': ['&']},          # 引用
        {'prod_lhs': 'unary_op', 'prod_rhs': ['&', 'mut']},
        # 6.5. 逻辑运算符
        {'prod_lhs': 'logic_or_op', 'prod_rhs': ['||']},
        {'prod_lhs': 'logic_and_op', 'prod_rhs': ['&&']},
    ],
    'start_symbol' : 'Begin'
}
```

依据 PPT 内容书写的文法

```
# 依据 PPT 内容书写的文法
RUST_GRAMMAR_PPT = {
    # 终结符需要自行定义 出现在左侧的符号加入到非终结符中
    'terminals' : {
        # 关键字
        'fn', 'mut', 'return', '->', 'let', 'if', 'else', 'while', 'for', 'loop', 'break', 'continue',
'in',
        # 类型
        'i32',
        # 标识符和字面量
        'ID', 'NUM',
        # 运算符
        '+', '-', '*', '/', '%', '&',
        '==', '!=', '<', '<=', '>', '>=',
        # 界符
        '(', ')', '[', ']', '{', '}', ';', ',', ':', '=', '.', '..',
    },
    # 可以通过产生式自动生成
    'non_terminals' : {
    },
    # 每一项是一个产生式 是一推一的关系
    'productions' : [
        # Program structure
        {'prod_lhs': 'Begin', 'prod_rhs': ['Program']},
        {'prod_lhs': 'Program', 'prod_rhs': ['DeclarationString']},
        {'prod_lhs': 'DeclarationString', 'prod_rhs': ['Declaration', 'DeclarationString']},
        {'prod_lhs': 'DeclarationString', 'prod_rhs': ['Declaration']},
```

```
        {'prod_lhs': 'Declaration', 'prod_rhs': ['FunctionDeclaration']},

        # Function declarations
        {'prod_lhs': 'FunctionDeclaration', 'prod_rhs': ['FunctionHeaderDeclaration',
'FunctionExpressionBlock']}, # 函数表达式块
        {'prod_lhs': 'FunctionDeclaration', 'prod_rhs': ['FunctionHeaderDeclaration',
'Block']},                  # 函数体
        {'prod_lhs': 'FunctionHeaderDeclaration', 'prod_rhs': ['fn', 'ID', '(', 'Parameters', ')']},
        {'prod_lhs': 'FunctionHeaderDeclaration', 'prod_rhs': ['fn', 'ID', '(', ')']},
        {'prod_lhs': 'FunctionHeaderDeclaration', 'prod_rhs': ['fn', 'ID', '(', 'Parameters', ')', '->',
'Type']},
        {'prod_lhs': 'FunctionHeaderDeclaration', 'prod_rhs': ['fn', 'ID', '(', ')', '->', 'Type']},

        # Blocks and parameters
        {'prod_lhs': 'Block', 'prod_rhs': ['{', 'StatementString', '}']},
        {'prod_lhs': 'Block', 'prod_rhs': ['{', '}']},
        {'prod_lhs': 'Parameters', 'prod_rhs': ['ParamVar']},
        {'prod_lhs': 'Parameters', 'prod_rhs': ['ParamVar', ',']},
        {'prod_lhs': 'Parameters', 'prod_rhs': ['ParamVar', ',', 'Parameters']},
        {'prod_lhs': 'ParamVar', 'prod_rhs': ['VarDeclaration', ':', 'Type']},

        # Variable declarations
        {'prod_lhs': 'VarDeclaration', 'prod_rhs': ['mut', 'ID']},
        {'prod_lhs': 'VarDeclaration', 'prod_rhs': ['ID']},

        # Types
        {'prod_lhs': 'Type', 'prod_rhs': ['i32']},
        {'prod_lhs': 'Type', 'prod_rhs': ['[', 'Type', ';', 'NUM', ']']},
        {'prod_lhs': 'Type', 'prod_rhs': ['(', 'TupleTypeInner', ')']},
        {'prod_lhs': 'Type', 'prod_rhs': ['(', ')']},
        {'prod_lhs': 'Type', 'prod_rhs': ['&', 'mut', 'Type']},
        {'prod_lhs': 'Type', 'prod_rhs': ['&', 'Type']},
        {'prod_lhs': 'TupleTypeInner', 'prod_rhs': ['Type', ',', 'TypeList']},
        {'prod_lhs': 'TupleTypeInner', 'prod_rhs': ['Type', ',']},
        {'prod_lhs': 'TypeList', 'prod_rhs': ['Type']},
        {'prod_lhs': 'TypeList', 'prod_rhs': ['Type', ',']},
        {'prod_lhs': 'TypeList', 'prod_rhs': ['Type', ',', 'TypeList']},

        # Statements
        {'prod_lhs': 'StatementString', 'prod_rhs': ['Statement']},
        {'prod_lhs': 'StatementString', 'prod_rhs': ['Statement', 'StatementString']},
        {'prod_lhs': 'Statement', 'prod_rhs': [';']},
        {'prod_lhs': 'Statement', 'prod_rhs': ['ReturnStatement']},
        {'prod_lhs': 'Statement', 'prod_rhs': ['VarDeclarationStatement']},
        {'prod_lhs': 'Statement', 'prod_rhs': ['AssignStatement']},
        {'prod_lhs': 'Statement', 'prod_rhs': ['Expression', ';']},
        {'prod_lhs': 'Statement', 'prod_rhs': ['IfStatement']},
        {'prod_lhs': 'Statement', 'prod_rhs': ['CirculateStatement']},
        {'prod_lhs': 'Statement', 'prod_rhs': ['VarDeclarationAssignStatement']},
        {'prod_lhs': 'Statement', 'prod_rhs': ['break', ';']},
        {'prod_lhs': 'Statement', 'prod_rhs': ['continue', ';']},
        {'prod_lhs': 'Statement', 'prod_rhs': ['break', 'Expression', ';']},

        {'prod_lhs': 'ReturnStatement', 'prod_rhs': ['return', 'Expression', ';']},
        {'prod_lhs': 'ReturnStatement', 'prod_rhs': ['return', ';']},

        {'prod_lhs': 'VarDeclarationStatement', 'prod_rhs': ['let', 'VarDeclaration', ':', 'Type',
';']},
        {'prod_lhs': 'VarDeclarationStatement', 'prod_rhs': ['let', 'VarDeclaration', ';']},

        {'prod_lhs': 'AssignStatement', 'prod_rhs': ['Assignableidentifier', '=', 'Expression', ';']},
```

```
        {'prod_lhs': 'VarDeclarationAssignStatement', 'prod_rhs': ['let', 'VarDeclaration', ':', 'Type',
'=', 'Expression', ';']},
        {'prod_lhs': 'VarDeclarationAssignStatement', 'prod_rhs': ['let', 'VarDeclaration', '=',
'Expression', ';']},

        # Control flow
        {'prod_lhs': 'IfStatement', 'prod_rhs': ['if', 'Expression', 'Block', 'ElseStatement']},
        {'prod_lhs': 'IfStatement', 'prod_rhs': ['if', 'Expression', 'Block']},
        {'prod_lhs': 'ElseStatement', 'prod_rhs': ['else', 'Block']},
        {'prod_lhs': 'ElseStatement', 'prod_rhs': ['else', 'if', 'Expression', 'Block',
'ElseStatement']},
        {'prod_lhs': 'ElseStatement', 'prod_rhs': ['else', 'if', 'Expression', 'Block']},

        {'prod_lhs': 'CirculateStatement', 'prod_rhs': ['WhileStatement']},
        {'prod_lhs': 'CirculateStatement', 'prod_rhs': ['ForStatement']},
        {'prod_lhs': 'CirculateStatement', 'prod_rhs': ['LoopStatement']},
        {'prod_lhs': 'WhileStatement', 'prod_rhs': ['while', 'Expression', 'Block']},
        {'prod_lhs': 'ForStatement', 'prod_rhs': ['for', 'VarDeclaration', 'in', 'IterableStructure',
'Block']},
        {'prod_lhs': 'LoopStatement', 'prod_rhs': ['loop', 'Block']},

        {'prod_lhs': 'IterableStructure', 'prod_rhs': ['Expression', '..', 'Expression']},
        {'prod_lhs': 'IterableStructure', 'prod_rhs': ['Element']},

        # Expressions
        {'prod_lhs': 'Expression', 'prod_rhs': ['AddExpression']},
        {'prod_lhs': 'Expression', 'prod_rhs': ['Expression', 'Relop', 'AddExpression']},
        {'prod_lhs': 'Expression', 'prod_rhs': ['FunctionExpressionBlock']},
        {'prod_lhs': 'Expression', 'prod_rhs': ['SelectExpression']},
        {'prod_lhs': 'Expression', 'prod_rhs': ['LoopStatement']},

        {'prod_lhs': 'FunctionExpressionBlock', 'prod_rhs': ['{', 'FunctionExpressionString', '}']},
        {'prod_lhs': 'FunctionExpressionString', 'prod_rhs': ['Expression']},
        {'prod_lhs': 'FunctionExpressionString', 'prod_rhs': ['Statement',
'FunctionExpressionString']},

        {'prod_lhs': 'SelectExpression', 'prod_rhs': ['if', 'Expression', 'FunctionExpressionBlock',
'else', 'FunctionExpressionBlock']},

        {'prod_lhs': 'AddExpression', 'prod_rhs': ['Item']},
        {'prod_lhs': 'AddExpression', 'prod_rhs': ['AddExpression', 'AddOp', 'Item']},

        {'prod_lhs': 'Item', 'prod_rhs': ['Factor']},
        {'prod_lhs': 'Item', 'prod_rhs': ['Item', 'MulOp', 'Factor']},

        {'prod_lhs': 'Factor', 'prod_rhs': ['Element']},
        {'prod_lhs': 'Factor', 'prod_rhs': ['[', 'ArrayElementList', ']']},
        {'prod_lhs': 'Factor', 'prod_rhs': ['[', ']']},
        {'prod_lhs': 'Factor', 'prod_rhs': ['(', 'TupleAssignInner', ')']},
        {'prod_lhs': 'Factor', 'prod_rhs': ['(', ')']},
        {'prod_lhs': 'Factor', 'prod_rhs': ['*', 'Factor']},
        {'prod_lhs': 'Factor', 'prod_rhs': ['&', 'mut', 'Factor']},
        {'prod_lhs': 'Factor', 'prod_rhs': ['&', 'Factor']},

        {'prod_lhs': 'ArrayElementList', 'prod_rhs': ['Expression']},
        {'prod_lhs': 'ArrayElementList', 'prod_rhs': ['Expression', ',']},
        {'prod_lhs': 'ArrayElementList', 'prod_rhs': ['Expression', ',', 'ArrayElementList']},

        {'prod_lhs': 'TupleAssignInner', 'prod_rhs': ['Expression', ',', 'TupleElementList']},
        {'prod_lhs': 'TupleAssignInner', 'prod_rhs': ['Expression', ',']},

        {'prod_lhs': 'TupleElementList', 'prod_rhs': ['Expression']},
        {'prod_lhs': 'TupleElementList', 'prod_rhs': ['Expression', ',']},
```

```
        {'prod_lhs': 'TupleElementList', 'prod_rhs': ['Expression', ',', 'TupleElementList']},

        {'prod_lhs': 'Assignableidentifier', 'prod_rhs': ['*', 'Assignableidentifier']},
        {'prod_lhs': 'Assignableidentifier', 'prod_rhs': ['AssignableidentifierInner']},

        {'prod_lhs': 'AssignableidentifierInner', 'prod_rhs': ['Element', '[', 'Expression', ']']},
        {'prod_lhs': 'AssignableidentifierInner', 'prod_rhs': ['Factor', '.', 'NUM']},
        {'prod_lhs': 'AssignableidentifierInner', 'prod_rhs': ['ID']},

        {'prod_lhs': 'Element', 'prod_rhs': ['NUM']},
        {'prod_lhs': 'Element', 'prod_rhs': ['Assignableidentifier']},
        {'prod_lhs': 'Element', 'prod_rhs': ['(', 'Expression', ')']},
        {'prod_lhs': 'Element', 'prod_rhs': ['ID', '(', 'Arguments', ')']},
        {'prod_lhs': 'Element', 'prod_rhs': ['ID', '(', ')']},

        {'prod_lhs': 'Arguments', 'prod_rhs': ['Expression']},
        {'prod_lhs': 'Arguments', 'prod_rhs': ['Expression', ',']},
        {'prod_lhs': 'Arguments', 'prod_rhs': ['Expression', ',', 'Arguments']},

        # Operators
        {'prod_lhs': 'Relop', 'prod_rhs': ['<']},
        {'prod_lhs': 'Relop', 'prod_rhs': ['<=']},
        {'prod_lhs': 'Relop', 'prod_rhs': ['>']},
        {'prod_lhs': 'Relop', 'prod_rhs': ['>=']},
        {'prod_lhs': 'Relop', 'prod_rhs': ['==']},
        {'prod_lhs': 'Relop', 'prod_rhs': ['!=']},

        {'prod_lhs': 'AddOp', 'prod_rhs': ['+']},
        {'prod_lhs': 'AddOp', 'prod_rhs': ['-']},

        {'prod_lhs': 'MulOp', 'prod_rhs': ['*']},
        {'prod_lhs': 'MulOp', 'prod_rhs': ['/']}
    ],
    'start_symbol' : 'Begin'
}
```

为了验证我依据自身理解所写文法的合理性和有效性，我进行了文法对比实验，本实验将分别使用我自主设计的文法（以下简称为“修改文法”）和依据 PPT 编写的文法（以下简称为“参考文法”）对相同的代码片段进行解析，通过对比生成的语法树结构，重点分析一下两个方面的差异：

1) 在表达式处理方面，修改文法采用的双轨制（Place Expression/Value Expression）分类方式将如何影响语法树的组织结构

2) 运算符优先级的显示层级设计会带来怎样的解析差异

测试用例 1：

```
fn program_3_2() {
    1*2/3;
    4+5/6;
    7<8;
    1*2+3*4<4/2-3/1;
}
```

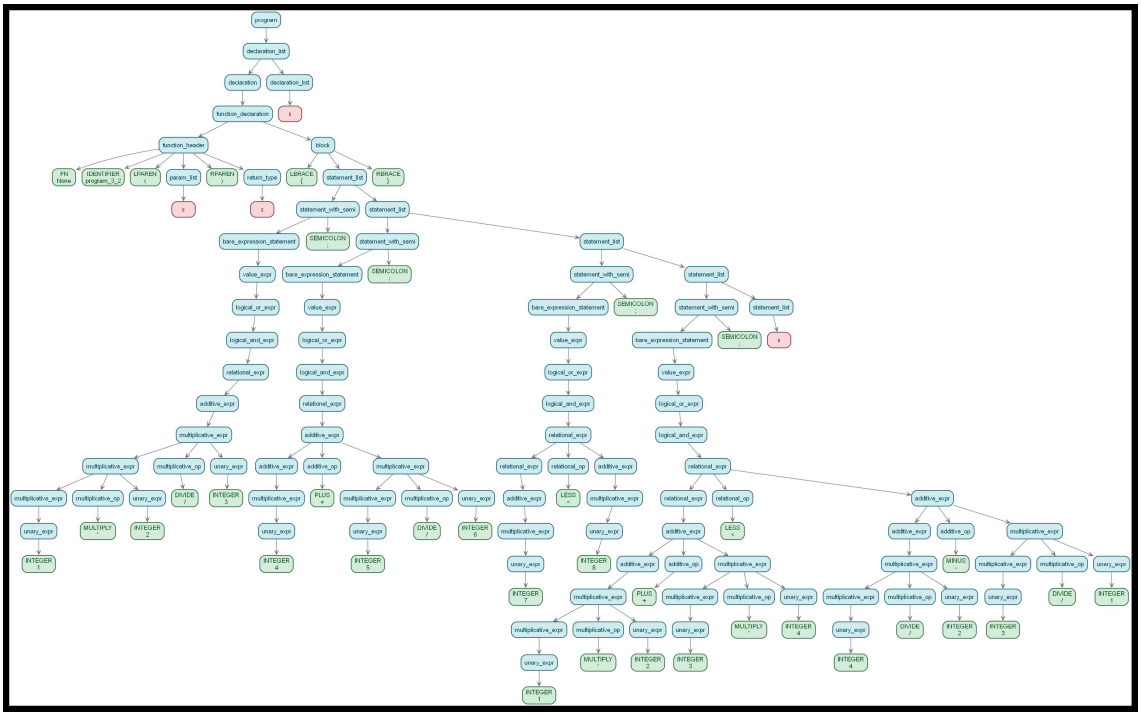


图 3 修改文法语法树（测试用例 1）

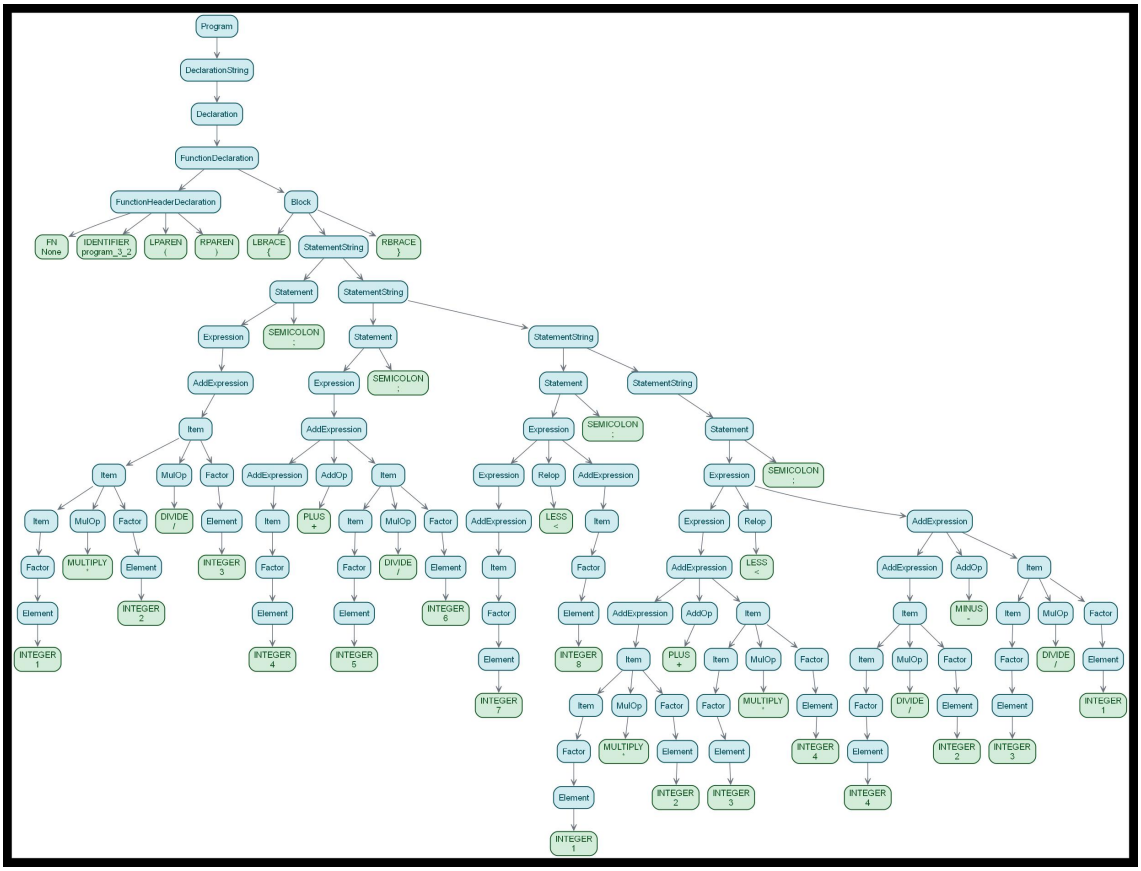


图 4 参考文法语法树（测试用例 1）

测试用例 2:

```
fn program_2_2(mut a:i32) {
    a=32;
}
```

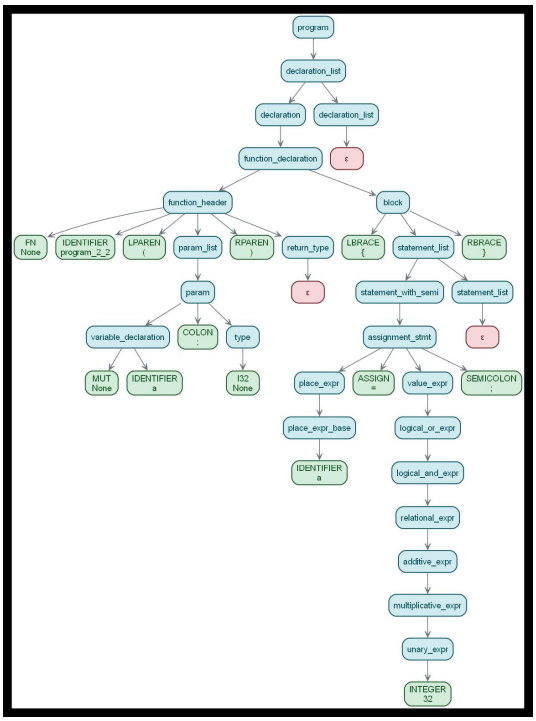


图 5 修改文法语法树（测试用例 2）

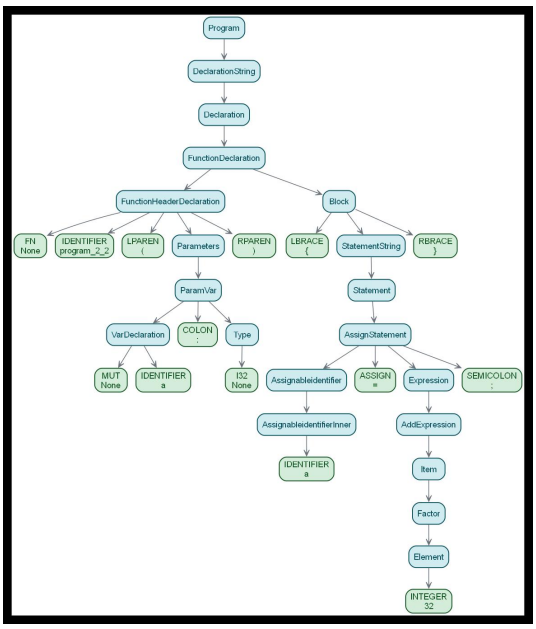


图 6 参考文法语法树（测试用例 2）

测试用例 3:

```
fn program_8_2(mut a:[i32;3]) {
    let mut b:i32=a[0];
    a[0]=1;
}
```

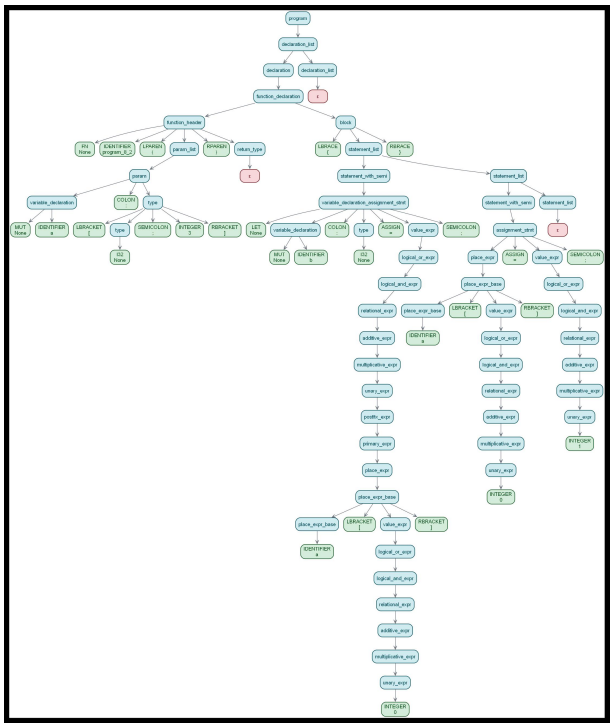


图 7 修改文法语法树（测试用例 3）

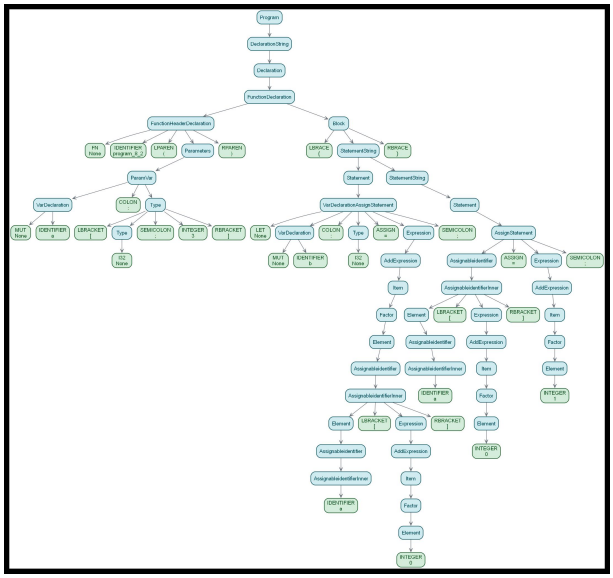


图 8 参考文法语法树（测试用例 3）

通过对改进文法和参考文法的测试对比，发现两者在解析上呈现高度一致性。在基础表达式（如算术运算、逻辑判断）的解析过程中，两种文法生成的语法树在节点结构和层次关系上基本吻合，都能正确体现运算符的优先级和结合性。

3. 前端展示界面设计

本系统的前端界面采用 Python 的 **Tkinter 库**实现，如图 3 所示，整体采用经典的左右分栏布局，左侧为代码编辑区和语法规则展示区，右侧为语法分析树可视化区和分析过程展示区。

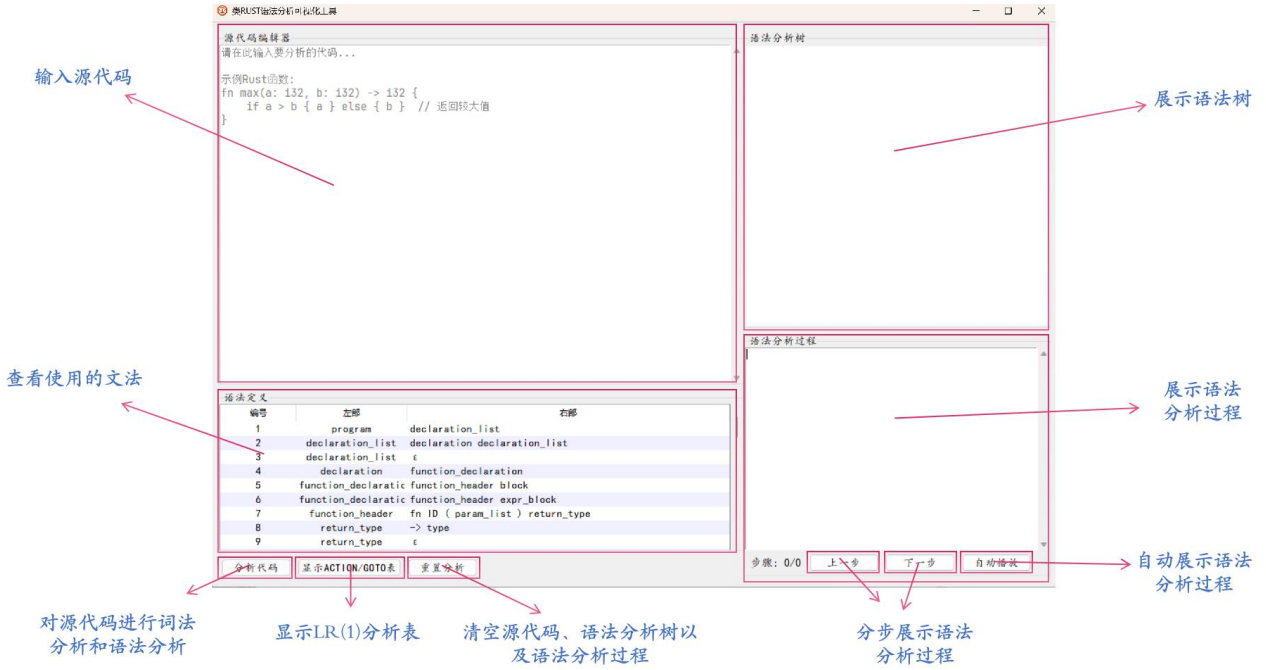


图 9 前端展示界面（1）

如图 4 所示，在视觉风格方面，界面采用了蓝绿色系作为主色调，非终结符节点使用浅蓝色背景，终结符节点使用浅绿色背景，ε节点使用浅红色背景，通过颜色编码帮助用户快速区分不同类型的语法元素。所有控件都采用了圆角矩形和适当的间距，营造出友好的视觉效果。代码编辑器使用等宽字体保证代码对齐，而说明文字使用楷体增强中文可读性。

交互设计方面实现了多项增强功能：语法分析树支持鼠标滚轮缩放和拖拽浏览；分析过程支持单步前进/后退和自动播放；所有表格都内置滚动条以适应大数据量展示。特别值得一提的是加载界面使用了动画 GIF，在耗时操作（如分析表构建）期间提供视觉反馈，避免用户误以为程序卡死。这些交互细节显著提升了系统的易用性。

响应式布局设计确保界面在不同分辨率下都能正常显示。主窗口默认大小为 1200x800 像素，所有面板都采用 pack 布局管理器实现自动扩展和填充。语法分析树可视化区域会根据画布大小自动调整图像尺寸，保证完整显示的同时最大化利用可用空间。

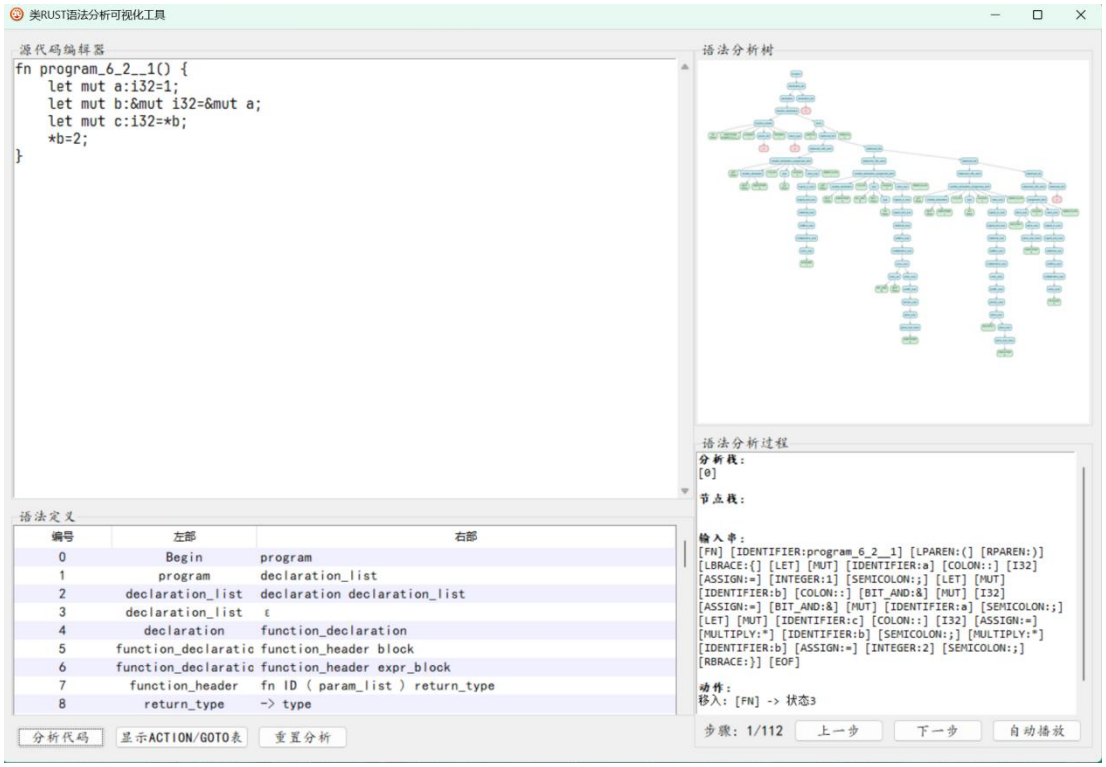


图 10 前端展示界面（2）

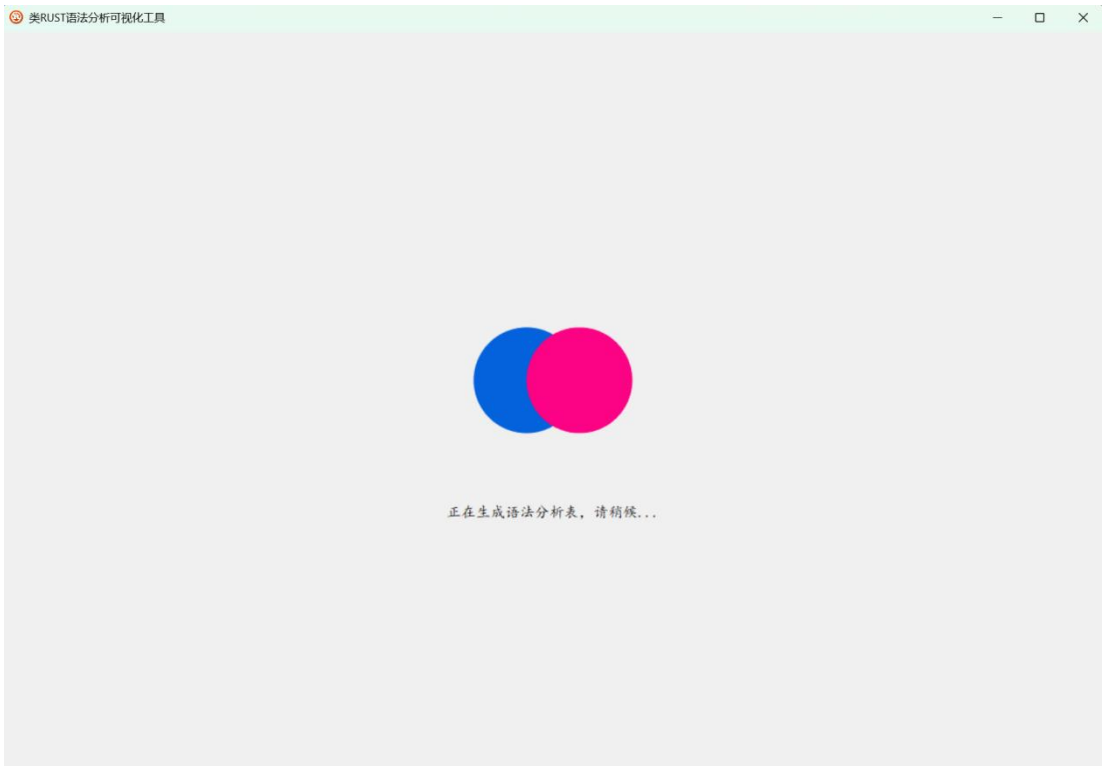
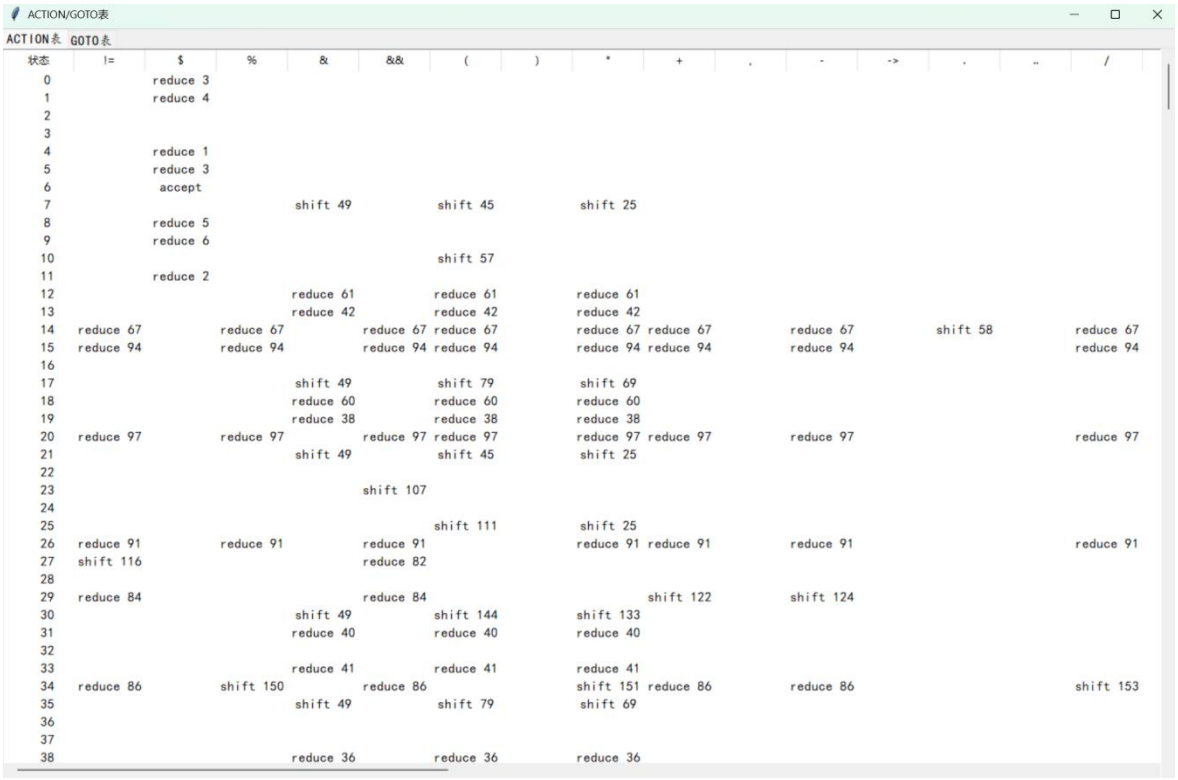
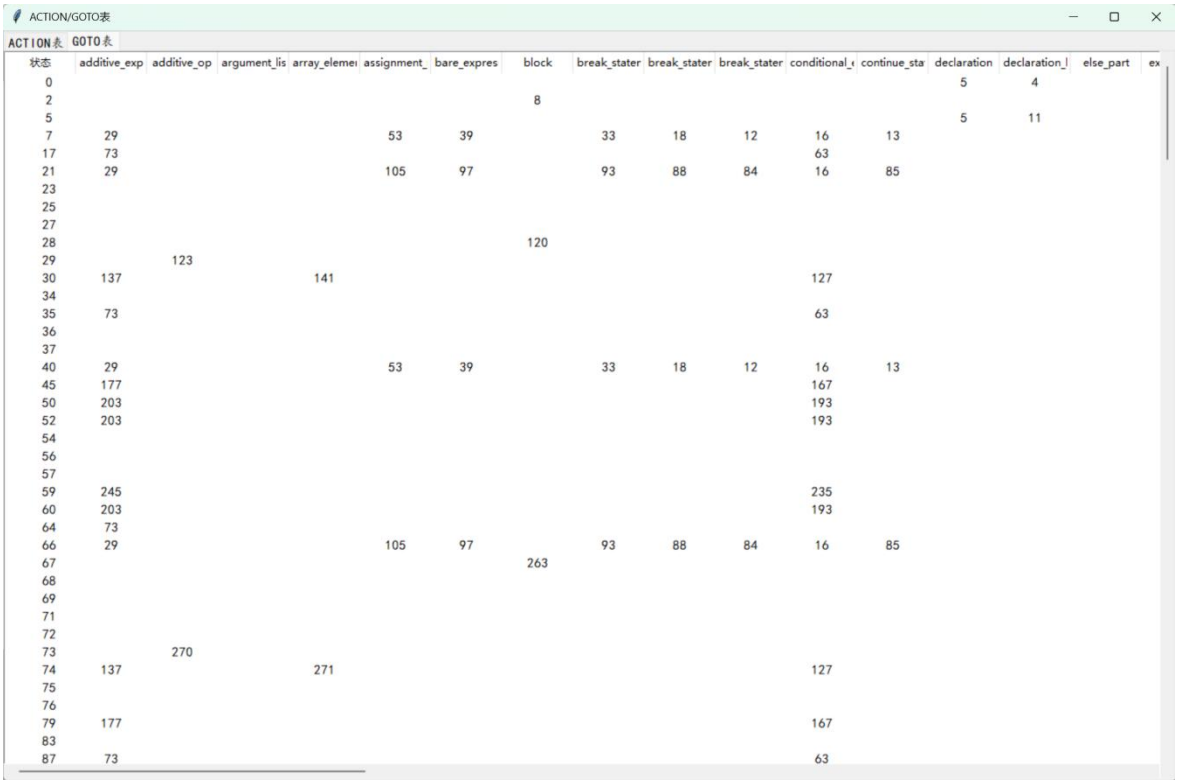


图 11 前端加载界面



状态	!=	$	%	&	&&	(	)	*	+	,	-	->	.	..	/
0		reduce 3													
1		reduce 4													
2															
3															
4		reduce 1													
5		reduce 3													
6		accept													
7				shift 49		shift 45		shift 25							
8		reduce 5													
9		reduce 6													
10						shift 57									
11		reduce 2													
12				reduce 61		reduce 61		reduce 61							
13				reduce 42		reduce 42		reduce 42							
14	reduce 67		reduce 67		reduce 67	reduce 67		reduce 67	reduce 67		reduce 67		shift 58		reduce 67
15	reduce 94		reduce 94		reduce 94	reduce 94		reduce 94	reduce 94		reduce 94				reduce 94
16															
17				shift 49		shift 79		shift 69							
18				reduce 60		reduce 60		reduce 60							
19				reduce 38		reduce 38		reduce 38							
20	reduce 97		reduce 97		reduce 97	reduce 97		reduce 97	reduce 97		reduce 97				reduce 97
21				shift 49		shift 45		shift 25							
22															
23					shift 107										
24															
25						shift 111		shift 25							
26	reduce 91		reduce 91		reduce 91			reduce 91	reduce 91		reduce 91				reduce 91
27	shift 116				reduce 82										
28															
29	reduce 84				reduce 84				shift 122		shift 124				
30				shift 49		shift 144		shift 133							
31				reduce 40		reduce 40		reduce 40							
32															
33				reduce 41		reduce 41		reduce 41							
34	reduce 86		shift 150		reduce 86			shift 151	reduce 86		reduce 86				shift 153
35				shift 49		shift 79		shift 69							
36															
37															
38				reduce 36		reduce 36		reduce 36							

图 12 ACTION 表展示



状态	additive_exp	additive_op	argument_lis	array_elemei	assignment_	bare_expres	block	break_stater	break_stater	break_stater	conditional_(	continue_sta	declaration	declaration_l	else_part	ex
0													5	4		
2							8									
5													5	11		
7	29				53	39		33	18	12	16	13				
17	73										63					
21	29				105	97		93	88	84	16	85				
23																
25																
27																
28							120									
29		123														
30	137			141							127					
34																
35	73										63					
36																
37																
40	29				53	39		33	18	12	16	13				
45	177										167					
50	203										193					
52	203										193					
54																
56																
57																
59	245										235					
60	203										193					
64	73															
66	29				105	97		93	88	84	16	85				
67							263									
68																
69																
71																
72																
73		270														
74	137			271							127					
75																
76																
79	177										167					
83																
87	73										63					

图 13 GOTO 表展示

三、思考

（一）1.4 函数输入中的形参列表可以识别怎样的语言？

- **1.4 函数输入（前置规则0.1、0.2、1.1）**
  - <形参列表>-> <形参> | <形参> ',' <形参列表>
  - <形参> -> <变量声明内部> ':' <类型>

特性	示例	说明
基本参数声明	(X: i32)	标准变量名+类型标注格式
多个参数	(X: i32, y: i32)	逗号分隔的参数列表
可变参数	(mut x: i32)	通过 mut 关键字标记
数组类型	(arr: [i32; 256])	支持定长数组类型声明
元组类型	(point: (i32, i32))	支持任意长度元组
引用类型	(s: &str)	支持不可变引用
可变引用	(s: &mut str)	支持可变引用
嵌套类型	(m: &mut [(i32, i32)])	支持任类型的任意嵌套组合

（二）9.1 元组 比 8.1 数组 多了几条产生式，为什么？

- **8.1 数组（前置规则0.2、3.1）**
  - <类型>-> '[' <类型> ';' <NUM>']'
  - <因子> -> '[' <数组元素列表> ']' | <数组元素>
  - <数组元素列表>-> 空 | <表达式> | <表达式> ',' <数组元素列表>

- **9.1 元组（前置规则0.2、3.1）**
  - <类型> -> '(' <元组类型内部> ')'
  - <元组类型内部> -> 空 | <类型> ',' <类型列表>
  - <类型列表> ->空 | <类型> | <类型> ',' <类型列表>
  - <因子> -> '(' <元组赋值内部> ')'
  - <元组赋值内部> -> 空 | <表达式> ',' <元组元素列表>
  - <元组元素列表>->空 | <表达式> | <表达式> ',' <元组元素列表>

元组比数组多出几条产生式的根本原因在于 Rust 语法中元组**需要特殊处理单元素情况以避免歧义**。由于括号在 Rust 中同时用于运算优先级控制和元组定义，单元素元组必须通过额外的逗号来区分（如(1,)），这就需要在文法中专门为单元素元组添加额外的产生式规则。相比之下，数组的语法形式始终明确（如[1]和[1,2,3]都是合法的数组表示法），不需要处理这种特殊情况。

四、实验总结

本次实验基于 LR(1)语法分析方法，成功实现了类 Rust 语言的词法与语法分析工具。在实验过程中，我完成了从文法设计到语法树可视化的全流程开发，重点解决了表达式优先级处理、语法冲突消解等问题。通过构建 LR(1)分析表（Action 表与 Goto 表），系统能够精准识别变量声明、函数调用、条件分支等复杂语法结构，并生成层次清晰的语法树。实验结果表明，该工具能够正确解析测试用例中的各类语法规则，验证了 LR(1)方法在复杂语言分析中的有效性。

通过本次实验，我不仅更加深入理解了 LR(1)闭包计算和状态转移等核心算法，还掌握了模块化系统设计的工程实践能力。通过分离词法分析、语法分析和可视化模块，显著提升了代码的可维护性和扩展性。此外，借助 Graphviz 实现的语法树可视化技术，让我对抽象语法结构的理解更加直观，为后续可能的语义分析阶段奠定了良好基础。

未来，我将继续优化系统的错误恢复能力和性能表现，并探索如何将现有成果扩展至更完整的编译器前端。本次实验不仅巩固了编译原理的理论知识，也极大地提升了我的工程实践能力和问题解决能力。

参考资料

[1] Introduction - The Rust Reference. https://doc.rust-lang.org/reference/

[2] 编译原理学习笔记（十）~LR(1)分析. https://blog.csdn.net/weixin_44225182/article/details/105597613

[3] 自底向上文法分析：LR(1) 分析. https://zhuanlan.zhihu.com/p/551222757

[4] 编译原理：第六章 LR 分析. https://cloud.tencent.com/developer/article/2069227