# 电子科技大学

## 计算机专业类课程

# 实验报告

课程名称：**编译原理**

学　　院：**计算机科学与工程学院**

学院专业：**计算机科学与技术**

学　　号：**2022010910017**

学生姓名：**谢卿云**

指导教师：**陈昆，陈文宇**

日　　期：**2025 年 5 月 10 日**

# 电 子 科 技 大 学

# 实 验 报 告

**学生姓名：谢卿云　　学号：2022010910017　指导教师：陈昆，陈文宇**

**实验地点：实验室　　实验时间：2025 年 5 月 10 日**

## 一、 实验室名称：

电子科技大学清水河校区基础实验大楼 A-504

## 二、 实验项目名称：

词法分析器 + 语法分析器 + 符号表

## 三、 实验原理：

### （一）编译程序的流程

编译器将源程序映射为语义等价的目标程序，由两部分构成：

1） 前端/分析: 将源程序分解，加上先验的语法结构创建源程序的中间表示；创建符号表; 报语法错误；

2） 后端/综合: 输入中间表示和符号表，输出期待的目标程序；

图 1 展示了编译程序处理源程序的一种流程图；

### （二）词法分析器的工作原理

词法分析器的主要任务是读取源代码字符流，并将其分解成一系列有意义的单元，称为词法单元；这些词法单元是语法分析阶段的输入。

### （三）有限自动机

有限自动机是一种计算模型，它根据输入符号序列在有限数量的状态之间转换。有两种主要的类型：非确定性有限自动机 (NFA) 和确定性有限自动机 (DFA)。DFA 可以识别输入字符流中是否包含符合该正则表达式模式的子串。整个词法分析器可以被看作是一个大型的有限自动机，它同时识别所有可能的词法单元类型。理论上，可以将每个词法单元的正则表达式转换为一个 NFA，然后将这些 NFA 合

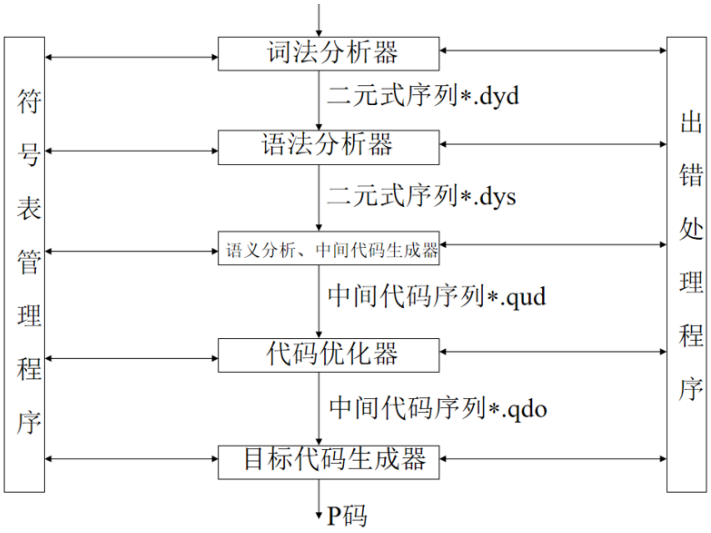


**图** 1: 编译程序处理源程序的流程图

并成一个大的 NFA。DFA 的优势在于在识别过程中不需要回溯，对于每个输入字符，都只有唯一确定的下一个状态，因此非常高效。

**（四）递归下降法**

语法分析的任务是根据语言的语法规则，检查词法分析器生成的词法单元序列是否构成一个合法的程序结构，并通常会构建一个抽象语法树的中间结构，这一结构并不是直接的数据结构；递归下降是一种自顶向下的语法分析方法。它的核心思想是将文法中的每个非终结符对应实现一个解析函数。每个解析函数负责识别和处理输入流中对应非终结符所能推导出的语言结构。当解析函数遇到文法产生式中的终结符时，它会检查当前的输入词法单元是否匹配，如果匹配则消耗该词法单元并前进；如果不匹配则报告语法错误。当解析函数遇到文法产生式中的非终结符时，它会递归地调用对应非终结符的解析函数来处理子结构。整个解析过程从文法开始符号对应的函数调用开始。

**（五）LL(1) 文法**

LL(1) 文法代表一种从左到右（Left-to-right）扫描输入生成最左（Leftmost）推导，只需向前看 1 个输入词法单元来决定使用哪条产生式进行推导的文法，这是一种无二义性的，无公共左因子和左递归的文法，因此在递归下降时不需要回溯；可以通过计算文法符号的 FIRST 集（非终结符可以推导出的第一个终结符的集合）和 FOLLOW 集（在推导过程中可能出现在某个非终结符后面的终结符的集合）来构建预测分析表或直接指导解析过程。如果文法是 LL(1) 的，那么对于任意非终结

符 A 和输入符号 a，预测分析表中最多只有一条 $A \rightarrow$ 的产生式。

**（六）预测分析器的工作原理**

预测分析法是递归下降分析法的一种特殊形式，它不使用回溯，利用向前看 lookahead 的输入词法单元来确定当前非终结符应该使用哪条产生式进行推导。对于 LL(1) 文法，只需向前看一个词法单元即可做出决定。

**（七）符号表和变量作用域**

符号表一个重要的数据结构，它在编译过程的各个阶段被用来记录和管理程序中使用的各种名字及其相关信息。符号表将程序中的名字或标识符映射到它们所代表的实体的相关信息。这些信息通常包括名字本身，实体的数据类型，函数返回类型，种类/类别，这决定了如何解释和使用这个名字。作用域用于指示该名字在程序的哪个区域是可见和有效的，它定义了一个名字在程序文本中的可见性区域或有效范围。同一个名字可以在程序的不同部分代表不同的实体，而作用域规则 precisely 规定了在程序的给定点，哪个声明与某个名字关联。处理块结构语言中的静态嵌套作用域的标准方法就是使用一个符号表栈

## 四、 实验目的：

1） 掌握词法分析和语法分析的基本原理, 学会改写符合 LL1 范式的文法；
2） 理解符号表在编译程序前端的作用，理解变量在静态作用域中的出入规则；
3） 理解 token，自动机，递归下降法，预测分析法在编译过程中应用；

## 五、 实验内容：

1） 实现预处理器: 负责读取源程序文件内容
2） 实现词法分析器: 定义目标语言的词法规则，识别各类词法单元
3） 实现语法分析器：实现基于该文法的递归下降预测分析器
4） 实现符号表管理器：实现基于栈的符号表结构，用于管理程序中的作用域嵌套。
5） 实现错误处理器：对词法分析和语法分析中的错误及时抛出

## 六、 实验器材（设备、元器件）：

Cargo 等 rust 语言包，Linux 服务器等

## 七、 实验步骤：

### （一）改写为 LL1 文法

我们需要对目标程序设计一个符合 LL1 范式的文法规则，设计的规则如表 1 所示。

### （二）设计自动机

该实验的词法分析器遵循以下状态转化图，这是一个有限自动机，如图 2 所示：

### （三）约定种别表

我们事先约定按照如下种别表，打印二元式，二元式的格式为：单词符号 + 种别，其中，单词符号应该为右对齐的长度为 16 的字符串，中间有一个空格，种别为长度为 2 的数字种别对照表的内容如图 3 所示，二元式文件的后缀为 *.dyd。

### （四）词法分析器的实现

在本项目中，src/lex.rs 文件实现了词法分析器，主要通过 Lexer 结构体来完成。

#### 7.4.1. 属性

1） source: 存储整个源程序的字符串内容。
2） pos 和 cha: pos 记录当前正在处理字符在 source 中的位置，cha 存储当前字符。
3） nxt 和 peek: nxt 记录下一个字符的位置，peek 存储下一个字符，用作缓冲区。
4） line: 记录当前处理的行号，用于错误报告。
5） token: 暂存正在构建的词法单元的字符串表示。
6） stream: 存储分析完成后生成的词法单元序列。
7） reserve_table: 存储关键字及其对应的 Token 类型，用于快速查找识别关键字。
8） word_table: 存储已识别的标识符。

| 非终结符 | 产生式 |
| --- | --- |
| < 程序 > | < 分程序 > |
| < 分程序 > | begin < 说明语句表 >< 执行语句表 >end |
| < 说明语句表 > | { < 说明语句 >; } |
| < 说明语句 > | integer < 说明语句'> |
| < 说明语句'> | function < 标识符 >( < 参数 >) ; < 函数体 > |
| | \| < 变量 > |
| < 函数体 > | begin < 说明语句表 >< 执行语句表 >end |
| < 参数 > | < 算术表达式 > |
| < 执行语句表 > | { < 执行语句 >; } |
| < 执行语句 > | < 读语句 > |
| | \| < 写语句 > |
| | \| < 赋值语句 > |
| | \| < 条件语句 > |
| < 赋值语句 > | < 变量 >:= < 算术表达式 > |
| < 条件语句 > | if < 条件表达式 >then < 执行语句 >else < 执行语句 > |
| < 读语句 > | read ( < 变量 >) |
| < 写语句 > | write ( < 变量 >) |
| < 条件表达式 > | < 算术表达式 >< 关系运算符 >< 算术表达式 > |
| < 算术表达式 > | < 项 >< 算术表达式'> |
| < 算术表达式'> | - < 项 >< 算术表达式'> |
| | \| $\varepsilon$ |
| < 项 > | < 因子 >< 项'> |
| < 项'> | * < 因子 >< 项'> |
| | \| $\varepsilon$ |
| < 因子 > | < 标识符 >< 因子后缀 > |
| | \| < 常数 > |
| | \| ( < 算术表达式 >) |
| < 因子后缀 > | ( < 参数 >) |
| | \| $\varepsilon$ |
| < 关系运算符 > | <\| <=\| >\| >=\| =\| <> |
| < 变量 > | < 标识符 > |
| < 常数 > | < 整数 > |



表 1: LL1 文法规则

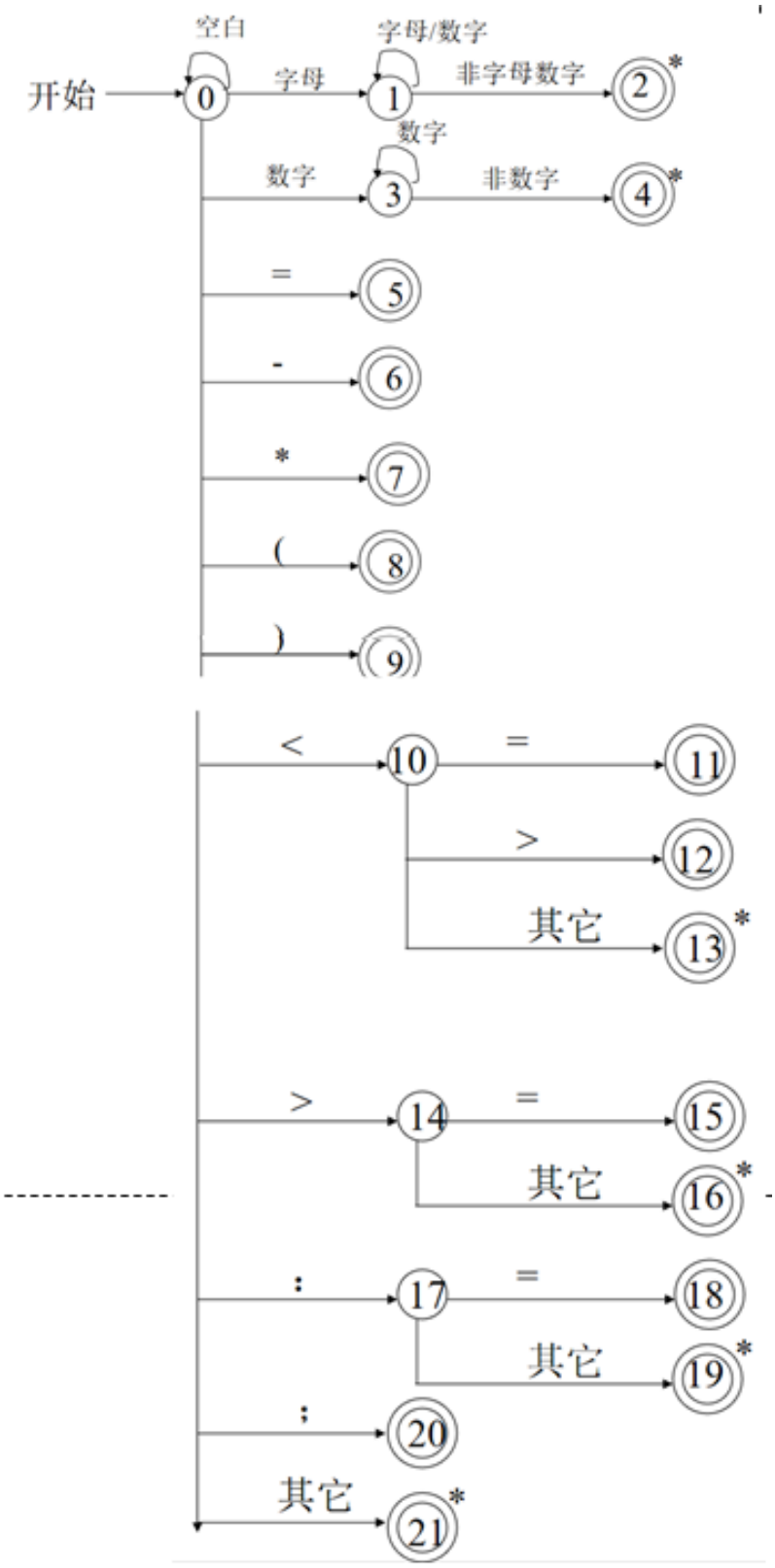


图 2: 词法分析器提取 token 运行的自动机模型

单词符号与种别对照表

| 单词符号 | 种别 | 单词符号 | 种别 | 单词符号 | 种别 |
|---|---|---|---|---|---|
| begin | 1 | end | 2 | integer | 3 |
| if | 4 | then | 5 | else | 6 |
| function | 7 | read | 8 | write | 9 |
| 标识符 | 10 | 常数 | 11 | = | 12 |
| <> | 13 | <= | 14 | < | 15 |
| >= | 16 | > | 17 | - | 18 |
| * | 19 | := | 20 | ( | 21 |
| ) | 22 | ; | 23 | | |

图 3: dyd 文件的输出内容需要遵循实现约定的种别对照表

9） literal_table: 存储已识别的整数常量。

10） max_len: 标识符的最大允许长度限制。

11） mode: 控制错误信息的输出方式（控制台或文件）。

#### 7.4.2. 方法

1） analyse(): 是词法分析的入口点。它会循环执行以下步骤直到文件末尾:

2） getnbc(): 跳过当前位置的空白字符（空格、制表符等，但不包括换行符）。

3） current_token(): 这是核心方法，负责识别并返回当前的词法单元。这是自动机实现的核心方法。将识别到的词法单元添加到 stream 中并清空 token 缓冲区，为下一个词法单元的识别做准备。

4） getchar(): 从 source 中读取下一个字符，更新 pos 和 cha，并根据需要增加 line。

5） get_peek(): 读取 cha 后面的一个字符并存入 peek，用于向前看一个字符，但不移动 pos。

6） getnbc(): 循环调用 getchar 跳过空白字符，直到遇到非空白字符或文件末尾。

#### 7.4.3. 标识符和常量的处理

1） lex_indentifier(): 当识别到以字母或下划线开头的字符序列时调用。它会一直读取后续的字母、数字或下划线 (is_dlu 判断)，构建完整的字符串到 token 中。然后调用 reserve() 检查 token 是否是关键字。如果是关键字，返

回对应的关键字 Token；否则，将其视为标识符，调用 word() 将其加入 word_table 并返回 Token::Identifier。

2） lex_digits_str(): 当识别到以数字开头的字符序列时调用。它会一直读取后续的数字，将字符串转换为整数值。然后调用 literal() 将其加入 literal_table 并返回 Token::IntegerLiteral。

3） reserve(): 检查 token 是否存在于 reserve_table (关键字表) 中。

4） word(): 将新的标识符添加到 word_table 中。

5） literal(): 将新的整数常量添加到 literal_table 中。

#### 7.4.4. 错误处理

当识别到非法字符或格式不正确的词法单元时，current_token 或 lex_indentifier/lex_digits_str 会调用 error 方法报告错误，并可能调用 skip_bad_line 跳过当前行的剩余部分，尝试从下一行继续分析，以实现错误恢复。

### （五）词法分析器的实现

该项目中，src/parse.rs 实现了语法分析器，采用递归下降和预测分析方法，用于处理一个 LL(1) 文法。文法中的每一个非终结符都对应于 Parser 结构体中的一个独立的 parse_node_* 方法。解析过程从代表文法开始符号的方法调用开始，通过函数间的相互调用模拟文法产生式的应用，实现自顶向下的推导过程。在每个解析函数中，解析器利用向前看一个词法单元来决定当前非终结符应该使用哪一条产生式进行推导。当期望匹配文法产生式中的终结符时，解析器检查当前输入词法单元是否匹配。若匹配，则消耗该词法单元并前进到下一个输入；若不匹配，则报告语法错误。

#### 7.5.1. 属性

1） stream: 存储 Lexer 生成的词法单元序列。这是语法分析器的输入。

2） pos: 记录当前正在处理的词法单元在 stream 向量中的索引位置。

3） line: 记录当前正在处理的词法单元所在的源代码行号（1-based）。

4） mode: 指定错误信息的输出模式。

5） name: 存储源文件的名称，用于在文件模式下生成错误文件名。

#### 7.5.2. 核心分析方法

1） analyse() 启动语法分析过程。这是语法分析的入口点，它调用 parse_node_-program 方法来解析整个程序结构。它接受一个可变的 Env 引用，以便在解析过程中管理符号表。

2） 词法单元操作方法:

3） current_token() 获取当前 pos 指向的词法单元，但不移动 pos。这是预测分析的关键，用于向前看一个符号。如果 pos 超出 stream 范围，返回 Token::Eof。

4） advance(): 将 pos 移动到下一个词法单元模拟消耗当前词法单元。它还会检查移动后的当前词法单元是否是换行并相应地更新当前行号；

5） match_token() 检查当前词法单元是否与给定的 tk 相等，用于匹配文法产生式中的终结符。

6） 解析非终结符方法:

7） parse_node_*(...) 对应文法中的每一个非终结符（如 < 程序 >, < 分程序 >, < 说明语句表 >, < 执行语句 >, < 算术表达式 > 等），负责解析和验证输入流中与该非终结符对应的语言结构。这些方法是递归下降的核心。

#### 7.5.3. 符号表交互方法

1） parse_node_block() 解析一个分程序（块），并在进入和退出块时管理符号作用域。

2） parse_node_declaration_statement_prime(): 处理变量或函数声明，并将其添加到当前作用域的符号表中。根据当前符号决定解析变量还是函数。成功解析出标识符后，检查重复声明，将其添加到符号表中。

#### 7.5.4. 特殊处理

1） 处理重复结构：循环的继续或终止依赖于对当前词法单元的检查。如果当前词法单元是该结构可以开始的符号，则解析对应的语句。如果当前词法单元属于该结构非终结符的 FOLLOW 集，则退出循环。否则，视为语法错误。

2） 因子后缀：用于处理函数调用（标识符后的 (< 参数 >)）。通过检查当前符号是否为左括号来决定是解析函数调用还是空推导。

3） 区分声明语句分支: 在解析 integer 后，通过检查下一个词法单元是否是 Token::Function 来区分是变量声明还是函数声明。

#### 7.5.5. 错误处理

1） 错误报告方法: 根据 mode 属性选择错误输出方式，调用 console_error（将格式化的错误信息打印到标准输出）或 file_error（将格式化的错误信息写入以.err 为后缀的文件）
2） 错误恢复方法: 在检测到错误后，跳过当前行的剩余词法单元；
3） 集成的错误处理流程：handle_error（）是解析方法中调用来处理错误的中心函数。
4） 错误传播: 子解析方法的调用使用了 match 语句来检查返回的 Result。如果子调用返回错误，则错误会被自动地向上传播，

## 八、 实验数据及结果分析：

#### （一）测试程序（输入）

本实验采取的正确输入程序如图 4 所示，这是一个用于计算 n 的阶乘的嵌套过程体。本实验一共设计了 6 个可能的输入错误程序。如图 6-11 所示

具体来说，每个程序直观看上去，分别包含一个错误，分别是

1） 非法标识符：2interger n;
2） 标识符长度溢出：interger k1145141919180aaaaaaaaa;
3） 冒号不匹配：k:114514=F(m);
4） 关键字出错：intgr k;
5） 关键字出错：funion;
6） 缺少左括号：readm);

#### （二）出错信息

正确的测试程序完整地通过了前端的编译，输出二元式文件的部分展示如图 12 所示，而剩下的错误程序也相应地返回了.err 文件，内容如图 12-17 所示：

```
1   begin
2     integer k ;
3     integer function F(n) ;
4       begin
5         integer n;
6         if n<=0
7         then F:=1
8         else F:=n*F(n-1) ;
9       end;
10    read(m) ;
11    k:=F(m) ;
12    write(k) ;
13  end
```

图 4: 正确的 Pascal 输入程序

```
1   begin
2     integer k;
3     integer function F(n) ;
4       begin
5         2integer n;
6         if n<=0
7         then F:=1
8         else F:=n*F(n-1) ;
9       end;
10    read(m) ;
11    k:=F(m) ;
12    write(k) ;
13  end
```

图 5: 非法标识符：2interger n;

```
1   begin
2     integer k1145141919810aaaaaaaaa ;
3     integer function F(n) ;
4       begin
5         integer n;
6         if n<=0
7         then F:=1
8         else F:=n*F(n-1) ;
9       end;
10    read(m) ;
11    k:=F(m) ;
12    write(k) ;
13  end
14
```

图 6: 标识符长度溢出：interger k1145141919180aaaaaaaaa;

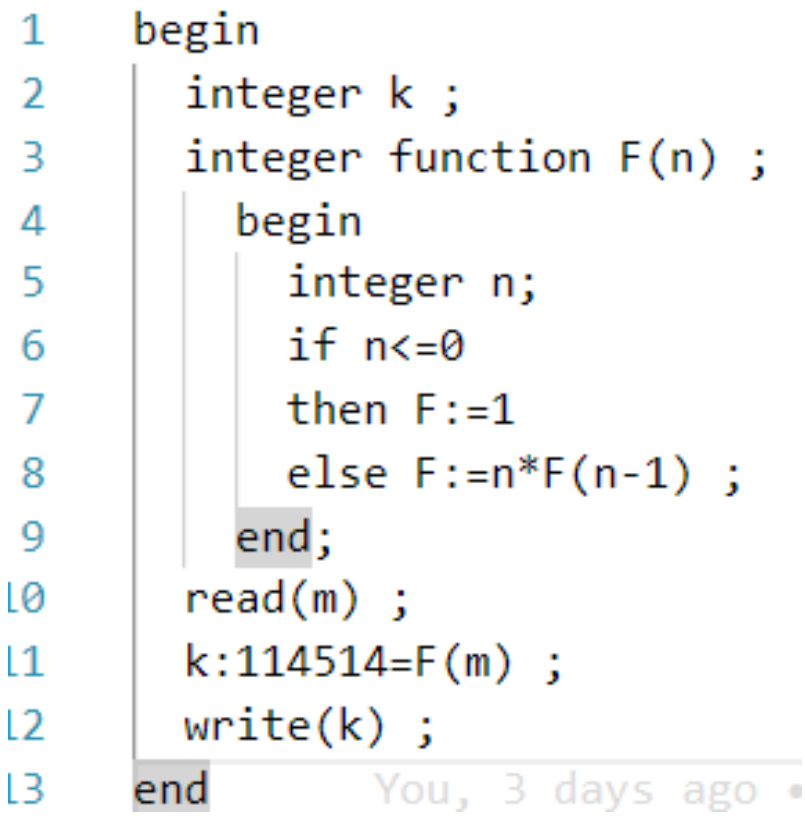


图 7: 冒号不匹配：k:114514=F(m);

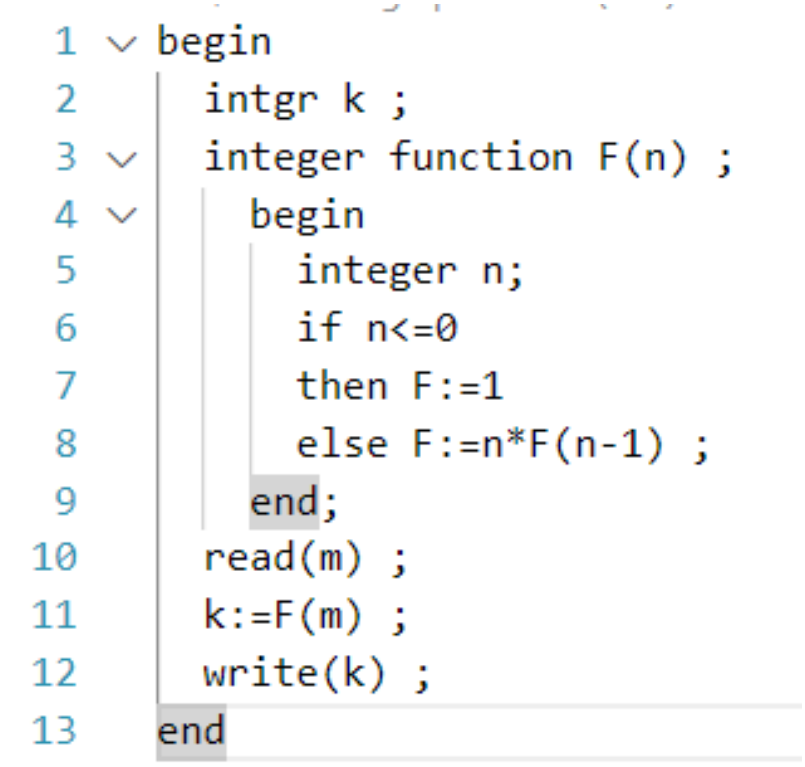


图 8: 关键字出错：intgr k;

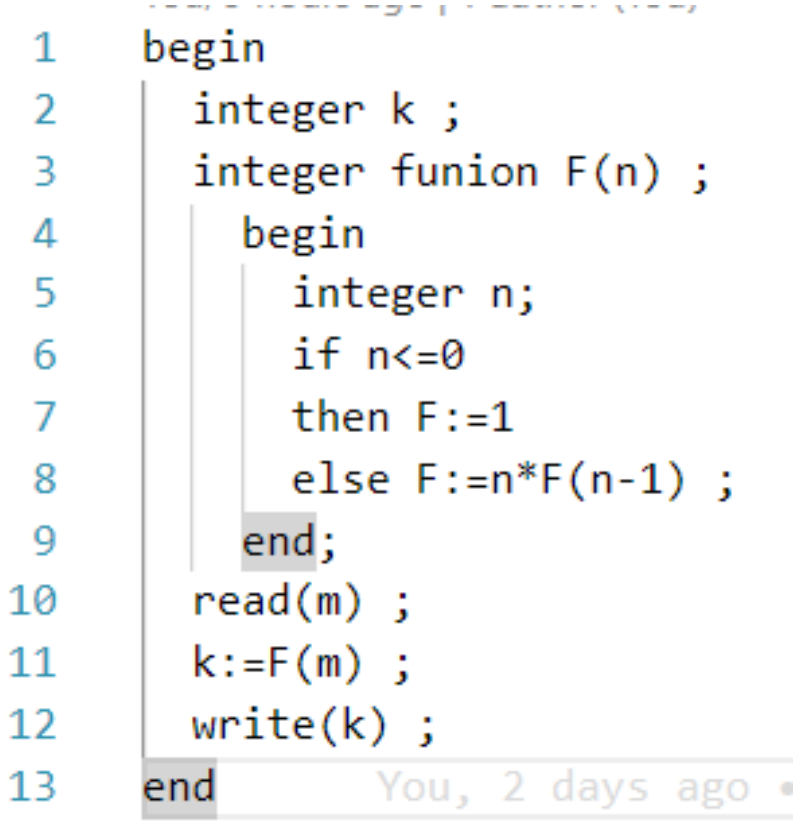


图 9: 关键字出错：funion;

```
1   begin
2     integer k ;
3     integer function F(n) ;
4       begin
5         integer n;
6         if n<=0
7         then F:=1
8         else F:=n*F(n-1) ;
9       end;
10    readm) ;
11    k:=F(m) ;
12    write(k) ;
13  end
```

图 10: 缺少左括号：readm);

```
begin 01
 \EOL 24
integer 03
    k 10
    ; 23
 \EOL 24
integer 03
function 07
    F 10
    ( 21
    n 10
    ) 22
    ; 23
 \EOL 24
 begin 01
 \EOL 24
integer 03
    n 10
    ; 23
 \EOL 24
  if 04
   n 10
  <= 14
   0 11
 \EOL 24
 then 05
    F 10
  := 20
   1 11
 \EOL 24
 else 06
    F 10
  := 20
   n 10
   * 19
   F 10
   ( 21
   n 10
   - 18
   1 11
   ) 22
   ; 23
 \EOL 24
  end 02
   ; 23
 \EOL 24
 read 08
    ( 21
    m 10
    ) 22
    ; 23
 \EOL 24
    k 10
```


图 12

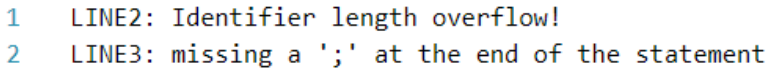


图 13

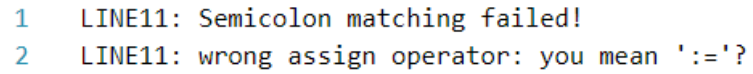


图 14

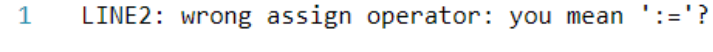


图 15

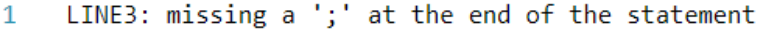


图 16

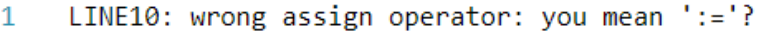


图 17

## 九、 实验结论：

从实验结果可以看到，编译器的报错信息和我们的预期不总是一致的。但是编译器的报错是根据语法分析的结果，也是合理的。事实上，对于实现完整符号表的前端，第 4 个错误程序报错信息应该是类似于未定义标识符，因为该变量没有加入到符号表中，符号表无法查询到相应的标识符；而对于第 6 个报错信息也应该是而定义标识符，因为此时处于对执行语句表解析的阶段，应该分析到一个赋值语句，但是 readm 也是一个没有加入到符号表的标识符，因此无法识别；

对于我们这个没有实现完整符号表的编译器前端，我们的报错结果是符合没有符号表的语法分析的结果的，并不完全是错误的；

代码的展示如附录一所示；

## 十、 总结及心得体会：

尽管实验成功验证了核心原理并构建了基本框架，但也认识到当前实现作为生产级编译器前端存在诸多不足（如错误恢复的健壮性、符号表中信息的完整性、对复杂文法结构的处理能力、性能优化以及与现代编译器架构的差距等）。

总而言之，本次实验通过从零开始实现一个简单的编译器前端，不仅成功地将编译原理的理论知识转化为实际代码。通过构建了一个可工作的简单 demo，我加深了对词法分析、语法分析、作用域管理和符号表工作原理的理解。实验结果证明了所采用方法的有效性。

## 十一、 对本实验过程及方法、手段的改进建议及展望：

错误恢复机制：本项目的词法分析器在遇到非法字符或错误时，会跳过当前行的剩余部分并返回错误。这是一种简单的恐慌模式错误恢复策略，即在检测到错误后，跳过一部分输入直到找到一个看似可以继续解析的同步点。可以设计更加用户友好的错误报告，比如“在 X 处期望得到 Y，但找到了 Z”，并给出可能的修正建议。还比如不仅仅跳到行尾，而是跳到更合适的同步词法单元，例如语句结束符;、块的开始/结束符号 (begin, end) 或文件末尾。这需要在文法中定义哪些词法单元可以作为同步点。

目前，对于变量声明的情况，无法维护其所属的过程名。在支持嵌套函数或过程的语言中，一个变量的完全限定名或用于查找作用域链的信息通常需要包含其所在的函数/过程信息。在解析函数体或任何可能包含变量声明的块时，可以将当

前正在解析的函数/过程的名称作为参数传递给其他方法，限于时间有限无法完成。

在文法中没有区分实参和形参，导致进一步语义分析可能出现了意料不到的错误。函数定义时的参数列表（形参）应该在函数体的局部作用域内进行声明，并且这些形参应该在函数体内部可见。这可能需要修改文法来明确区分函数定义和函数调用，并为函数定义引入形参列表的文法规则，例如 < 形参列表 > → < 变量 > , < 变量 > | ε

我们只对文件流进行了简化实现，在 prep.rs 中的文件不足可能在于错误处理比较简单，以及没有考虑字符编码问题。lex.rs 和 parse.rs 中的文件错误输出每次都会新建并覆盖文件，没有追加功能。而且错误应该输出到标准错误流而不是标准输出，这在命令行工具中是更好的实践。应该采用更安全的 IO 方式；

lex.rs 中的 current_token 方法是手写的状态转换逻辑，模拟了有限自动机。手写 DFA 对于简单文法是可行的, 但对于复杂语言，状态和转换会变得非常多且难以管理，容易出错。之后可以尝试手动实现一个表驱动的 DFA，根据当前状态和输入字符查找下一个状态。

输入优化：Lexer 在 getchar 中使用下标索引的来获取字符。这种方式在 pos 较大时效率较低，因为它需要从字符串的开头开始遍历字符直到 pos 位置. 更好的方式是使用缓冲读取，对于非常大的文件，可以考虑实现或使用带缓冲的文件读取，而不是一次性将整个文件读入内存。

**报告评分：**

**指导教师签字：**

# 附录一 代码示例

示例代码如代码 **??** 所示。

代码 1: 总控程序 main.rs

```rust
pub mod prep;
pub mod lex;
pub mod env;
pub mod parse;

use prep::Preprocessor;
use lex::Lexer;
use parse::Parser;
use env::Env;

fn main() {
    let path = "test/6";
    let mode = "file";
    let preprocessor = Preprocessor::new(path);
    println!("----------------------------------");

    let mut lexer = Lexer::new(preprocessor, path, mode);
    lexer.analyse();
    lexer.save();
    println!("----------------------------------");

    let mut env = Env::new();
    let s = lexer.get_stream();

    let mut parser = Parser::new(s, mode, path.to_string());
    match parser.analyse(&mut env) {
        Ok(()) => println!("compilered!"),
        _ => println!("syntax error!")
    }
}
```

代码 2: 预处理器 prep.rs

```rust
use std::fs::File;
use std::io::{Read};

pub struct Preprocessor {
    pub path: String, // 源程序名
    pub content: String, // 源程序字符流
}

impl Preprocessor {
    pub fn new(name: &str) -> Self{
        let mut p = Preprocessor{
            path: name.to_string() + ".pas",
            content: String::new(),
        };
        let mut input = File::open(&p.path).unwrap();
        input.read_to_string(&mut p.content).unwrap();
        println!("{}", p.content);
        p
    }
}
```

代码 3: 词法分析器 lex.rs

代码 4: 语法分析器 parse.rs

```rust
use crate::env::{Token, ErrorMessage, Env};
use std::fs;
use std::io::{Write};

pub struct Parser {
    // LL1语法分析器,基于递归下降办法
    pub stream: Vec<Token>, // 输入的token流
    pub pos: usize, //当前token所在位置
    pub line: usize, // 当前token所在行数
    mode: &'static str, // 错误的打印模式
    name: String,
}

impl Parser {
    pub fn new(s: Vec<Token>, mode: &'static str, name: String) -> Self {
        let p = Parser {
            stream: s,
            pos: 0,
            line: 1,
            mode: mode,
            name: name,
        };
        p
    }
    pub fn analyse(&mut self, env: &mut Env) -> Result<(), ErrorMessage> {
        self.parse_node_program(env)
    }

    fn debug(&self) {
        println!("---------------------------");
        println!("debug point");
        println!("pos:{:?}", self.pos);
        println!("line:{:?}", self.line);
        println!("token:{:?}", self.current_token());
    }
    fn current_token(&self) -> Token {
        if self.pos >= self.stream.len() {
            Token::Eof
        } else {
            self.stream[self.pos].clone()
        }
    }
    fn advance(&mut self) {
        if self.pos >= self.stream.len() {
            return;
        }

        self.pos += 1;
        let mut tk = self.current_token();

        // 处理换行符
        while tk == Token::Eol && self.pos < self.stream.len() {
            self.line += 1;
            self.pos += 1;
            tk = self.current_token();
        }
    }
    fn match_token(&self, tk: Token) -> bool {
        self.current_token() == tk
    }
    fn error(&self, errmsg: ErrorMessage) {
        match self.mode {
            "console" => self.console_error(errmsg),
            "file" => self.file_error(&errmsg),
            _ => println!("invalid mode!"),
        };
    }
    fn file_error(&self, errmsg:&ErrorMessage) {
```

```
69              let path = format!("{}.err", self.name);
70              let mut file = fs::OpenOptions::new()
71                  .write(true)
72                  .append(true)
73                  .create(true)
74                  .open(&path)
75                  .expect("Failed to create error file");
76              let err_msg = match errmsg {
77                  ErrorMessage::SyntaxError => format!("LINE{:?}: unknown token!\n
                        ", self.line),
78                  ErrorMessage::WrongReserveYouMeanFunction => format!("LINE{:?}:
                        wrong reserve: you mean 'function'?\n", self.line),
79                  ErrorMessage::WrongReserveYouMeanRead => format!("LINE{:?}:
                        wrong reserve: you mean 'read'?\n", self.line),
80                  ErrorMessage::WrongReserveYouMeanWrite => format!("LINE{:?}:
                        wrong reserve: you mean 'write'?\n", self.line),
81                  ErrorMessage::WrongAssignToken => format!("LINE{:?}: wrong
                        assign operator: you mean ':='?\n", self.line),
82                  ErrorMessage::InvalidTypeExpectedInterger => format!("LINE{:?}:
                        invalid type: expected INTEGER\n", self.line),
83                  ErrorMessage::InvalidNumber => format!("LINE{:?}: Invalid number
                        !\n", self.line),
84                  ErrorMessage::OverflowIdentifier => format!("LINE{:?}:
                        Identifier length overflow!\n", self.line),
85                  ErrorMessage::FailMatchingSemicolon => format!("LINE{:?}:
                        Semicolon matching failed!\n", self.line),
86                  ErrorMessage::MissingSemicolon => format!("LINE{:?}: missing a
                        ';' at the end of the statement\n", self.line),
87                  ErrorMessage::MissingLeftParenthesis => format!("LINE{:?}:
                        expected '(' following the function statement\n", self.line),
88                  ErrorMessage::MissingRightParenthesis => format!("LINE{:?}:
                        expected ')' to cover the block\n", self.line),
89                  ErrorMessage::MissingIf => format!("LINE{:?}: expected 'if' \n",
                         self.line),
90                  ErrorMessage::MissingThen => format!("LINE{:?}: expected 'then'
                        \n", self.line),
91                  ErrorMessage::MissingElse => format!("LINE{:?}: expected 'else'
                        \n", self.line),
92                  ErrorMessage::MissingMultiply => format!("LINE{:?}: expected '*'
                         \n", self.line),
93                  ErrorMessage::SyntaxErrorExpectedABlock => format!("LINE{:?}:
                        syntax error, expected a block\n", self.line),
94                  ErrorMessage::FailMatching => format!("LINE{:?}: Symbol matching
                         error!\n", self.line),
95                  ErrorMessage::MissingEnd => format!("LINE{:?}: missing END: this
                         block is not covered\n", self.line),
96                  ErrorMessage::ExpectedIdentifier => format!("LINE{:?}: Expected
                        identifier in this field\n", self.line),
97                  ErrorMessage::FoundRepeatDeclarationInThisField => format!("LINE
                        {:?}: this symbol's declaration repeated in this field\n",
                        self.line),
98              };
99              file.write_all(err_msg.as_bytes()).expect("Failed to write error
                    file");
100         }
101         fn console_error(&self, errmsg: ErrorMessage) {
102             // 抛出错误
103             // 这里还是简化实现
104             // 应该扔到标准错误流中，写入文件
105             // 不能和标准输出流混合
106             // self.debug();
107             match errmsg {
108                 ErrorMessage::SyntaxError => {
109                     println!("LINE{:?}: syntax error, exited", self.line);
110                 }
111                 ErrorMessage::WrongReserveYouMeanFunction => {
112                     println!("LINE{:?}: wrong reserve: you mean 'function'?",
                            self.line);
113                 }
```

```
114                 ErrorMessage::WrongReserveYouMeanRead => {
115                     println!("LINE{:?}: wrong reserve: you mean 'read'?", self.
                            line);
116                 }
117                 ErrorMessage::WrongReserveYouMeanWrite => {
118                     println!("LINE{:?}: wrong reserve: you mean 'write'?", self.
                            line);
119                 }
120                 ErrorMessage::WrongAssignToken => {
121                     println!("LINE{:?}: wrong assign operator: you mean ':='?",
                            self.line);
122                 }
123                 ErrorMessage::InvalidTypeExpectedInterger => {
124                     println!("LINE{:?}: invalid type here, expected integer",
                            self.line);
125                 }
126                 ErrorMessage::InvalidNumber => {
127                     println!("LINE{:?}: invalid number", self.line);
128                 }
129                 ErrorMessage::OverflowIdentifier => {
130                     println!("LINE{:?}: the length of identifier overflows",
                            self.line);
131                 }
132                 ErrorMessage::FailMatchingSemicolon => {
133                     println!("LINE{:?}: fail in matching semicolon", self.line);
134                 }
135                 ErrorMessage::MissingSemicolon => {
136                     println!("LINE{:?}: missing a ';' at the end of the
                            statement", self.line);
137                 }
138                 ErrorMessage::MissingLeftParenthesis => {
139                     println!("LINE{:?}: missing a '(' following the function
                            statement", self.line);
140                 }
141                 ErrorMessage::MissingRightParenthesis => {
142                     println!("LINE{:?}: missing a ')' to cover the block", self.
                            line);
143                 }
144                 ErrorMessage::MissingMultiply => {
145                     println!("LINE{:?}: expected '*' ", self.line);
146                 }
147                 ErrorMessage::MissingIf => {
148                     println!("LINE{:?}: expected 'if' ", self.line);
149                 }
150                 ErrorMessage::MissingThen => {
151                     println!("LINE{:?}: expected 'then' ", self.line);
152                 }
153                 ErrorMessage::MissingElse => {
154                     println!("LINE{:?}: expected 'else' ", self.line);
155                 }
156                 ErrorMessage::SyntaxErrorExpectedABlock => {
157                     println!("LINE{:?}: syntax error, expected a block", self.
                            line);
158                 }
159                 ErrorMessage::FailMatching => {
160                     println!("LINE{:?}: 符号匹配错误!", self.line);
161                 }
162                 ErrorMessage::MissingEnd => {
163                     println!("LINE{:?}: missing END: this block is not covered",
                             self.line);
164                 }
165                 ErrorMessage::ExpectedIdentifier => {
166                     println!("LINE{:?}: expected indentifier", self.line);
167                 }
168                 ErrorMessage::FoundRepeatDeclarationInThisField => {
169                     println!("LINE{:?}: the declaration of this indentifier
                            repeated in this scope", self.line);
170                 }
171             }
172             println!("--------------------------");
```

```
173         }
174         fn skip_bad_line(&mut self) {
175             let mut tk = self.current_token();
176             while tk != Token::Eol && tk != Token::Eof {
177                 self.advance();
178                 tk = self.current_token();
179             }
180         }
181         fn handle_error(&mut self, errmsg: ErrorMessage) -> Result<(),
                ErrorMessage>{
182             let res = Err(errmsg.clone());
183             self.debug();
184             self.error(errmsg);
185             self.skip_bad_line();
186             res
187         }
188 
189         fn parse_node_program(&mut self, env: &mut Env) -> Result<(),
                ErrorMessage>{
190             // <程序> → <分程序>
191             self.parse_node_block(env)
192         }
193         fn parse_node_block(&mut self, env: &mut Env) -> Result<(), ErrorMessage
                >{
194             // <分程序> → begin <说明语句表><执行语句表> end
195             match self.match_token(Token::Begin) {
196                 true => self.advance(),
197                 false => return self.handle_error(ErrorMessage::
                        SyntaxErrorExpectedABlock)
198             }
199             env.enter_scope();
200             match self.parse_node_declaration_statement_table(env) {
201                 Ok(_) => (),
202                 Err(e) => return self.handle_error(e),
203             }
204             // match self.match_token(Token::Semicolon) {
205             //     true => self.advance(),
206             //     false => return self.handle_error(ErrorMessage::
                    MissingSemicolon),
207             // }
208             match self.parse_node_execution_statement_table() {
209                 Ok(_) => (),
210                 Err(e) => return self.handle_error(e),
211             }
212             match self.match_token(Token::End) {
213                 true => self.advance(),
214                 false => return self.handle_error(ErrorMessage::MissingEnd)
215             }
216             env.exit_scope();
217             Ok(())
218         }
219         fn parse_node_declaration_statement_table(&mut self, env:&mut Env) ->
                Result<(), ErrorMessage>{
220             // <说明语句表> → {<说明语句> ;}
221             loop {
222                 if !self.match_token(Token::Integer) {
223                     // 检查FOLLOW 集
224                     match self.current_token() {
225                         Token::Read | Token::Write | Token::If | Token::
                            Identifier(_) | Token::End | Token::Eof => {
226                             return Ok(());
227                         },
228                         _ => return self.handle_error(ErrorMessage::SyntaxError)
                            ,
229                     }
230                 }
231 
232                 // 匹配说明语句
233                 match self.parse_node_declaration_statement(env) {
```

```
                    Ok(_) => (),
                    Err(e) => return self.handle_error(e),
                }

                // 匹配分号
                match self.match_token(Token::Semicolon) {
                    true => self.advance(),
                    false => return self.handle_error(ErrorMessage::
                        MissingSemicolon),
                }
            }
        }
        fn parse_node_declaration_statement(&mut self, env: &mut Env) -> Result
            <(), ErrorMessage>{
            // <说明语句> → integer <说明语句'>
            match self.match_token(Token::Integer) {
                true => self.advance(),
                false => return self.handle_error(ErrorMessage::
                    InvalidTypeExpectedInterger)
            }
            self.parse_node_declaration_statement_prime(env)
        }
        fn parse_node_declaration_statement_prime(&mut self, env: &mut Env) ->
            Result<(), ErrorMessage>{
            // <说明语句'> → <变量> | function <标识符>(<参数>)<函数体> ;
            if self.match_token(Token::Function) {
                // 函数说明分支
                self.advance();

                // 获取函数标识符名称
                let pname = match self.parse_node_identifier() {
                    Ok(pname) => pname,
                    Err(e) => return self.handle_error(e),
                };
                // 检查是否重复声明，若没有则添加声明
                if env.check_repeat(pname.clone()){
                    return self.handle_error(ErrorMessage::
                        FoundRepeatDeclarationInThisField);
                } else{
                    env.add_procedure(pname.clone());
                }

                match self.match_token(Token::LeftParenthesis) {
                    true => self.advance(),
                    false => return self.handle_error(ErrorMessage::
                        MissingLeftParenthesis)
                }
                match self.parse_node_parameter() {
                    Ok(_) => (),
                    Err(e) => return self.handle_error(e),
                }
                match self.match_token(Token::RightParenthesis) {
                    true => self.advance(),
                    false => return self.handle_error(ErrorMessage::
                        MissingRightParenthesis)
                }
                match self.match_token(Token::Semicolon) {
                    true => {
                        self.advance();
                        match self.parse_node_function_body(env) {
                            Ok(_) => Ok(()),
                            Err(e) => return self.handle_error(e),
                        }
                    },
                    false => self.handle_error(ErrorMessage::MissingSemicolon),
                }
            } else {
                // 变量说明分支
                // 获取变量名字
```

```
296                 let vname = match self.parse_node_variable() {
297                     Ok(vname) => vname,
298                     Err(e) => return self.handle_error(e),
299                 };
300                 // 检查是否重复声明，若没有则添加声明
301                 if env.check_repeat(vname.clone()){
302                     return self.handle_error(ErrorMessage::
                        FoundRepeatDeclarationInThisField);
303                 } else{
304                     env.add_variable(vname.clone(), "F".to_string(),0);
305                     Ok(())
306                 }
307             }
308         }
309         fn parse_node_function_body(&mut self, env:&mut Env) -> Result<(),
                ErrorMessage>{
310             // <函数体> → begin <说明语句表><执行语句表> end
311             match self.match_token(Token::Begin) {
312                 true => self.advance(),
313                 false => return self.handle_error(ErrorMessage::
                        SyntaxErrorExpectedABlock)
314             }
315             match self.parse_node_declaration_statement_table(env) {
316                 Ok(_) => (),
317                 Err(e) => return self.handle_error(e),
318             }
319             // match self.match_token(Token::Semicolon) {
320             //     true => self.advance(),
321             //     false => return self.handle_error(ErrorMessage::
                    MissingSemicolon)
322             // }
323             match self.parse_node_execution_statement_table() {
324                 Ok(_) => (),
325                 Err(e) => return self.handle_error(e),
326             }
327             match self.match_token(Token::End) {
328                 true => {
329                     self.advance();
330                     Ok(())
331                 },
332                 false => return self.handle_error(ErrorMessage::MissingEnd)
333             }
334         }
335         fn parse_node_parameter(&mut self) -> Result<(), ErrorMessage>{
336             // <参数> → <算术表达式>
337             match self.parse_node_expression() {
338                 Ok(_) => Ok(()),
339                 Err(e) => return self.handle_error(e),
340             }
341         }
342         fn parse_node_execution_statement_table(&mut self) -> Result<(),
                ErrorMessage>{
343             // <执行语句表> → {<执行语句> ;}
344             // FOLLOW(<执行语句表>) 包含 'end' 和 '$'
345             loop {
346                 // 检查当前token是否可以开始一个执行语句
347                 match self.current_token() {
348                     Token::Read | Token::Write | Token::If | Token::Identifier(_
                            ) => {
349                         // 可以开始执行语句，继续解析
350                     },
351                     Token::End | Token::Eof => {
352                         // 否则，检查是否在 FOLLOW 集里 (end 或 EOF)
353                         // 如果在，说明执行语句表结束
354                         return Ok(());
355                     },
356                     _ => return self.handle_error(ErrorMessage::SyntaxError),
357                 }
```

```
                // 解析一个执行语句
                match self.parse_node_execution_statement() {
                    Ok(_) => (),
                    Err(e) => return self.handle_error(e),
                }

                // 期望匹配分号
                match self.match_token(Token::Semicolon) {
                    true => self.advance(),
                    false => return self.handle_error(ErrorMessage::
                        MissingSemicolon),
                }
            }
        }
        fn parse_node_execution_statement(&mut self) -> Result<(), ErrorMessage
            >{
            // <执行语句> → <读语句>|<写语句>|<赋值语句>|<条件语句>
            match self.current_token() {
                Token::Read => self.parse_node_read_statement(),
                Token::Write => self.parse_node_write_statement(),
                Token::If => self.parse_node_conditional_statement(),
                Token::Identifier(_) => self.parse_node_assignment_statement(),
                _ => self.handle_error(ErrorMessage::SyntaxError)
            }
        }
        fn parse_node_assignment_statement(&mut self) -> Result<(), ErrorMessage
            >{
            // <赋值语句> → <变量> := <算术表达式>
            match self.parse_node_variable() {
                Ok(_) => (),
                Err(e) => return self.handle_error(e),
            }
            match self.match_token(Token::Assign) {
                true => self.advance(),
                false => return self.handle_error(ErrorMessage::WrongAssignToken
                    )
            }
            match self.parse_node_expression() {
                Ok(_) => Ok(()),
                Err(e) => return self.handle_error(e),
            }
        }
        fn parse_node_conditional_statement(&mut self) -> Result<(),
            ErrorMessage>{
            // <条件语句> → if<条件表达式>then<执行语句>else <执行语句>
            match self.match_token(Token::If) {
                true => self.advance(),
                false => return self.handle_error(ErrorMessage::MissingIf)
            }
            match self.parse_node_condition() {
                Ok(_) => (),
                Err(e) => return self.handle_error(e),
            }
            match self.match_token(Token::Then) {
                true => self.advance(),
                false => return self.handle_error(ErrorMessage::MissingThen)
            }
            match self.parse_node_execution_statement() {
                Ok(_) => (),
                Err(e) => return self.handle_error(e),
            }
            match self.match_token(Token::Else) {
                true => self.advance(),
                false => return self.handle_error(ErrorMessage::MissingElse)
            }
            match self.parse_node_execution_statement() {
                Ok(_) => Ok(()),
                Err(e) => return self.handle_error(e),
```

```
422             }
423         }
424         fn parse_node_read_statement(&mut self) -> Result<(), ErrorMessage>{
425             // <读语句> → read(<变量>)
426             match self.match_token(Token::Read) {
427                 true => self.advance(),
428                 false => return self.handle_error(ErrorMessage::
                        WrongReserveYouMeanRead)
429             }
430             match self.match_token(Token::LeftParenthesis) {
431                 true => self.advance(),
432                 false => return self.handle_error(ErrorMessage::
                        MissingLeftParenthesis)
433             }
434             match self.parse_node_variable() {
435                 Ok(_) => (),
436                 Err(e) => return self.handle_error(e),
437             }
438             match self.match_token(Token::RightParenthesis) {
439                 true => {
440                     self.advance();
441                     Ok(())
442                 },
443                 false => return self.handle_error(ErrorMessage::
                        MissingRightParenthesis)
444             }
445         }
446         fn parse_node_write_statement(&mut self) -> Result<(), ErrorMessage>{
447             // <写语句> → write(<变量>)
448             match self.match_token(Token::Write) {
449                 true => self.advance(),
450                 false => return self.handle_error(ErrorMessage::
                        WrongReserveYouMeanWrite)
451             }
452             match self.match_token(Token::LeftParenthesis) {
453                 true => self.advance(),
454                 false => return self.handle_error(ErrorMessage::
                        MissingLeftParenthesis)
455             }
456             match self.parse_node_variable() {
457                 Ok(_) => (),
458                 Err(e) => return self.handle_error(e),
459             }
460             match self.match_token(Token::RightParenthesis) {
461                 true => {
462                     self.advance();
463                     Ok(())
464                 },
465                 false => return self.handle_error(ErrorMessage::
                        MissingRightParenthesis)
466             }
467         }
468         fn parse_node_condition(&mut self) -> Result<(), ErrorMessage>{
469             // <条件表达式> → <算术表达式><关系运算符><算术表达式>
470             match self.parse_node_expression() {
471                 Ok(_) => (),
472                 Err(e) => return self.handle_error(e),
473             }
474             match self.parse_node_relational_operator() {
475                 Ok(_) => self.advance(),
476                 Err(e) => return self.handle_error(e),
477             }
478             match self.parse_node_expression() {
479                 Ok(_) => (),
480                 Err(e) => return self.handle_error(e),
481             }
482             Ok(())
483         }
484         fn parse_node_expression(&mut self) -> Result<(), ErrorMessage>{
```

```
            // <算术表达式> → <项> <算术表达式'>
            match self.parse_node_term() {
                Ok(_) => (),
                Err(e) => return self.handle_error(e),
            }
            match self.parse_node_expression_prime() {
                Ok(_) => (),
                Err(e) => return self.handle_error(e),
            }
            Ok(())
        }
        fn parse_node_expression_prime(&mut self) -> Result<(), ErrorMessage>{
            // <算术表达式'> → -<项> <算术表达式'> | ε
            if self.match_token(Token::Minus) {
                self.advance();
                match self.parse_node_term() {
                    Ok(_) => (),
                    Err(e) => return self.handle_error(e),
                }
                match self.parse_node_expression_prime() {
                    Ok(_) => (),
                    Err(e) => return self.handle_error(e),
                }
                Ok(())
            }else {
                Ok(())
            }
        }
        fn parse_node_term(&mut self) -> Result<(), ErrorMessage>{
            // <项> → <因子> <项'>
            match self.parse_node_factor() {
                Ok(_) => (),
                Err(e) => return self.handle_error(e),
            }
            match self.parse_node_term_prime() {
                Ok(_) => (),
                Err(e) => return self.handle_error(e),
            }
            Ok(())
        }
        fn parse_node_term_prime(&mut self) -> Result<(), ErrorMessage>{
            // <项'> → *<因子> <项'> | ε
            if self.match_token(Token::Multiply) {
                self.advance();
                match self.parse_node_factor() {
                    Ok(_) => (),
                    Err(e) => return self.handle_error(e),
                }
                match self.parse_node_term_prime() {
                    Ok(_) => (),
                    Err(e) => return self.handle_error(e),
                }
                Ok(())
            }else {
                Ok(())
            }
        }
        fn parse_node_factor(&mut self) -> Result<(), ErrorMessage>{
            // <因子> → <标识符> <因子后缀> | <常数> | (<算术表达式>)
            match self.current_token() {
                Token::LeftParenthesis => {
                    self.advance();
                    match self.parse_node_expression() {
                        Ok(_) => (),
                        Err(e) => return self.handle_error(e),
                    }
                    match self.match_token(Token::RightParenthesis) {
                        true => {
                            self.advance();
```

```
                        Ok(())
                    },
                    false => self.handle_error(ErrorMessage::
                        MissingRightParenthesis)
                }
            },
            Token::IntegerLiteral(_) => self.parse_node_constant(),
            Token::Identifier(_) => {
                match self.parse_node_identifier() {
                    Ok(_) => (),
                    Err(e) => return self.handle_error(e),
                }
                match self.parse_node_factor_suffix() {
                    Ok(_) => (),
                    Err(e) => return self.handle_error(e),
                }
                Ok(())
            },
            _ => self.handle_error(ErrorMessage::SyntaxError)
        }
    }
    fn parse_node_factor_suffix(&mut self) -> Result<(), ErrorMessage>{
        // <因子后缀> → (<参数>)| ε
        match self.match_token(Token::LeftParenthesis) {
            true => self.advance(),
            false => return Ok(()),
        }
        match self.parse_node_parameter() {
            Ok(_) => (),
            Err(e) => return self.handle_error(e),
        }
        match self.match_token(Token::RightParenthesis) {
            true => self.advance(),
            false => return self.handle_error(ErrorMessage::
                MissingRightParenthesis)
        }
        Ok(())
    }
    fn parse_node_relational_operator(&mut self) -> Result<(), ErrorMessage
        >{
        // <关系运算符> → <|<=|>|>=|=|<>
        match self.current_token() {
            Token::Equal => Ok(()),
            Token::NotEqual => Ok(()),
            Token::Less => Ok(()),
            Token::LessEqual => Ok(()),
            Token::Greater => Ok(()),
            Token::GreaterEqual => Ok(()),
            _ => self.handle_error(ErrorMessage::SyntaxError)
        }
    }
    fn parse_node_variable(&mut self) -> Result<String, ErrorMessage>{
        // <变量> → <标识符>
        match self.parse_node_identifier() {
            Ok(name) => Ok(name),
            Err(e) => return Err(e),
        }
    }
    fn parse_node_constant(&mut self) -> Result<(), ErrorMessage>{
        // <常量> → <整数>
        match self.current_token() {
            Token::IntegerLiteral(_) => {
                self.advance();
                Ok(())
            },
            _ => self.handle_error(ErrorMessage::InvalidNumber)
        }
    }
    fn parse_node_identifier(&mut self) -> Result<String, ErrorMessage>{
```

```
            // <标识符>
            match self.current_token() {
                Token::Identifier(name) => {
                    self.advance();
                    Ok(name.clone())
                },
                _ => Err(ErrorMessage::ExpectedIdentifier),
            }
        }
    }
```

代码 5: 符号表控制器 env.rs

```
use std::collections::HashMap;

#[derive(Clone, PartialEq, Debug)]
pub enum Token {
    // 所有的记号
    // 标识符
    Identifier(String),

    // 字面量
    IntegerLiteral(i64),

    // 算术运算符
    Minus,
    Multiply,
    Assign,

    // 关系运算符
    Equal,
    NotEqual,
    Less,
    LessEqual,
    Greater,
    GreaterEqual,

    // 分界符
    Begin,
    End,
    LeftParenthesis,
    RightParenthesis,
    Semicolon,

    // 行末提示符
    Eol,

    // 文件末提示符
    Eof,

    // 关键字
    Integer,
    Function,
    If,
    Then,
    Else,
    Read,
    Write,

    // 非法字符
    Illegal(char),
}

#[derive(Clone)]
pub enum ErrorMessage {
    // 所有的报错信息
    SyntaxError,// 语法错误
    WrongReserveYouMeanFunction, // wrong reserve: you mean 'function'?
```

```
56          WrongReserveYouMeanRead, // wrong reserve: you mean 'read'?
57          WrongReserveYouMeanWrite, // wrong reserve: you mean 'write'?
58          WrongAssignToken, // wrong assign operator: you mean ':='?
59          InvalidTypeExpectedInterger, // 非法的类型, expected integer
60          InvalidNumber, // 非法数字串
61          OverflowIdentifier,// 标识符长度溢出
62          FailMatchingSemicolon, // 冒号不匹配
63          MissingSemicolon, // 缺一个分号,
64          MissingLeftParenthesis, // expected '(' following the function statement
65          MissingRightParenthesis, // expected ')' to cover the block
66          MissingIf, // expected 'if'
67          MissingThen, // expected 'then'
68          MissingElse, // expected 'else'
69          MissingMultiply, // expected 'multiply'
70          SyntaxErrorExpectedABlock, // expected a block
71          FailMatching, // 符号匹配错误
72          MissingEnd, // begin没有匹配的end
73          ExpectedIdentifier, // 符号无声明
74          FoundRepeatDeclarationInThisField, //符号重复声明
75      }
76
77      #[derive(Clone)]
78      pub struct VariableItem {
79          // 变量表项
80          pub vname: String, // 变量名
81          pub vproc: String, // 所属过程
82          pub vkind: i32, // 0-变量, 1-形参
83          pub vlev: i32, // 变量所在层次
84          // pub vadr: i32, // 相对于第一个变量在变量表中的位置
85          pub vtype: Vec<i32>, // 变量类型
86      }
87      impl VariableItem {
88          pub fn new(vname: String, vproc: String, vkind: i32, vlev: i32)-> Self{
89              let v = VariableItem {
90                  vname: vname,
91                  vproc: vproc,
92                  vkind: vkind,
93                  vlev: vlev,
94                  vtype: Vec::new(),
95              };
96              v
97          }
98      }
99
100     #[derive(Clone)]
101     pub struct ProcedureItem {
102         // 过程表项
103         pub pname: String, // 过程名
104         pub ptype: Vec<i32>, //过程类型
105         pub plev: i32, // 过程所在层次
106         // pub fadr: i32, // 第一个变量在变量表里的位置
107         // pub ladr: i32, // 最后一个变量在变量表中的位置
108     }
109     impl ProcedureItem {
110         pub fn new(pname: String, plev: i32) -> Self {
111             let p = ProcedureItem {
112                 pname: pname,
113                 ptype: Vec::new(),
114                 plev: plev
115             };
116             p
117         }
118     }
119
120     #[derive(Clone)]
121     pub struct SymbolTable {
122         // 符号表,每个作用域都应该对应一个符号表
123         pub variables: HashMap<String, VariableItem>, // 变量表
```

```
124         pub procedures: HashMap<String, ProcedureItem>, // 过程表
125         pub level: i32, // 当前作用域层级
126     }
127     impl SymbolTable {
128         pub fn new(level: i32) -> Self {
129             let st = SymbolTable {
130                 variables: HashMap::new(),
131                 procedures: HashMap::new(),
132                 level: level,
133             };
134             st
135         }
136         pub fn get_level(&self) -> i32{
137             self.level
138         }
139     }
140
141     #[derive(Clone)]
142     pub struct Env {
143         // 符号表栈，管理顶层符号表随作用域变化
144         pub stack: Vec<SymbolTable>,
145     }
146     impl Env {
147         pub fn new() -> Self {
148             let e = Env {
149                 stack: Vec::new(),
150             };
151             e
152         }
153         pub fn enter_scope(&mut self){
154             // 进入作用域，移入一个空符号表
155             let t = SymbolTable::new(self.stack.len() as i32);
156             self.stack.push(t);
157         }
158         pub fn exit_scope(&mut self){
159             // 退出作用域，移出栈顶符号表
160             self.stack.pop();
161         }
162         pub fn add_variable(&mut self, vname: String, vproc: String, vkind: i32)
                {
163             // 声明一个变量
164             let t: &mut SymbolTable = self.stack.last_mut().unwrap();
165             let item = VariableItem::new(vname.clone(), vproc, vkind, t.
                    get_level());
166             t.variables.insert(vname, item);
167         }
168         pub fn delete_cariable(&mut self, vname: String){
169             // 析构一个变量
170             let t: &mut SymbolTable = self.stack.last_mut().unwrap();
171             t.variables.remove_entry(&vname);
172         }
173         pub fn add_procedure(&mut self, pname: String) {
174             // 声明一个过程
175             let t: &mut SymbolTable = self.stack.last_mut().unwrap();
176             let item = ProcedureItem::new(pname.clone(), t.get_level());
177             t.procedures.insert(pname, item);
178         }
179         pub fn delete_procedure(&mut self, pname: String) {
180             // 析构一个过程
181             let t: &mut SymbolTable = self.stack.last_mut().unwrap();
182             t.procedures.remove_entry(&pname);
183         }
184         pub fn find_symbol(&self, name: String) -> bool{
185             // 自顶向下查找一个符号
186             for s in self.stack.iter().rev() {
187                 if s.variables.get(&name).is_some()||s.procedures.get(&name).
                        is_some() {
188                     return true;
189                 }
```

```
            }
            false
        }
        pub fn check_repeat(&self, name: String) -> bool {
            // 检查当前作用域是否重复声明某符号
            let t = self.stack.last().unwrap();
            t.variables.contains_key(&name) || t.procedures.contains_key(&name)
        }
        pub fn save(&self){
            // 保存在.var文件
        }
    }
```