电子科技大学

计算机专业类课程

实验报告

课程名称：编译原理实验

学　　院：计算机科学与工程学院（网络空间安全学院）

专　　业：计算机科学与技术

学生姓名：李坤嵘

学　　号：2021010914005

指导教师：陈昆

日　　期：　　2024 年　　5 月　　7 日

电 子 科 技 大 学

实 验 报 告

实验一

一、实验名称：词法分析

二、实验学时：4

三、实验内容和目的：

设计实现一个词法分析器，使得其能够识别单词，并输出单词和其对应的种别。对于不在保留字里的单词，或者非法形式的保留字，能够输出报错信息。

将输出信息保存在.dyd 文件中。

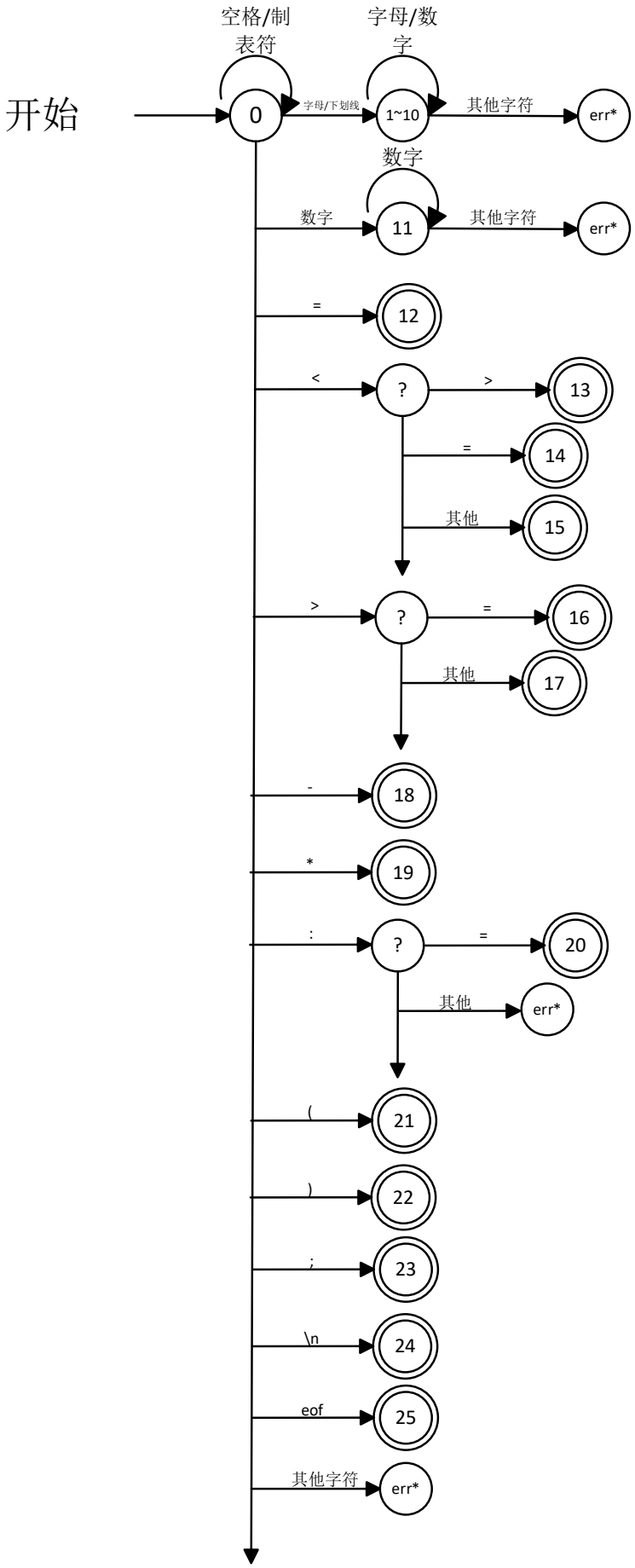
四、实验原理：

词法分析是将字符串序列转换为单词序列的过程，本次实验所需要识别的保留字如下。

种类	内容
关键字	begin、end、integer、if、then、else、function、read、write
标识符	[a-zA-Z_][a-zA-Z0-9_]*
常数	0|[1-9][0-9]*|-[1-9][0-9]*
操作符	=、<>、<=、<、>=、>、-、*、:=、(、)、;

将文件视为一个连续的输入流，不断地从输入流中读取字符。当输入缓冲区中出现保留字时，即记录，并在二元式文件.dyd 中输出一行。之后清空缓存区，继续读取。当读到文件末尾 EOF 时停止。

状态图如下：


开始
空格/制表符
字母/数字
0
字母/下划线
1~10
其他字符
err*
数字
11
数字
其他字符
err*
=
12
<
?
>
13
=
14
其他
15
>
?
=
16
其他
17
-
18
*
19
:
?
=
20
其他
err*
(
21
)
22
;
23
\n
24
eof
25
其他字符
err*

五、实验器材（设备、元器件）

Ubuntu 20.04.3 LTS
G++ version 9.4.0(采用 C++ 17 标准编译，编译时需要指明-std=c++17)
Visual Studio Code

六、实验步骤：

1. 设计实现驱动库 driver.hpp

驱动库中包含需要使用的 C++标准库，例如 iostream，string，vector 等
同时分配命名空间，包括：

控制流命名空间 core，包含三个内联变量使得输入流、输出流、错误流全局化。
词法分析命名空间 lexer，包含入口函数和词法分析的所有函数。
语法分析命名空间 parser，包含入口函数和语法分析的所有函数。

同时，还包括一个 vector<string> keywords，用以指明关键字。
一个 enum Symbols，用以指明每个关键字对应的编号。
enum Errors，用以输出错误码。

2. 设计实现一个主函数 main()

该函数从控制台中获取参数，从而指定要解析的文件，以及根据参数决定是否进行语法分析。

解析完参数后，分配输入控制流，输出控制流，输出控制流，并调用词法分析入口函数。

3. 设计实现输出函数 put_token()

put_token 函数接收两个参数，即当前的 token 以及该 token 对应的 symbol。调用 fprintf 函数格式化输出到输出流之中。

4. 设计实现匹配函数 check_keyword_type()

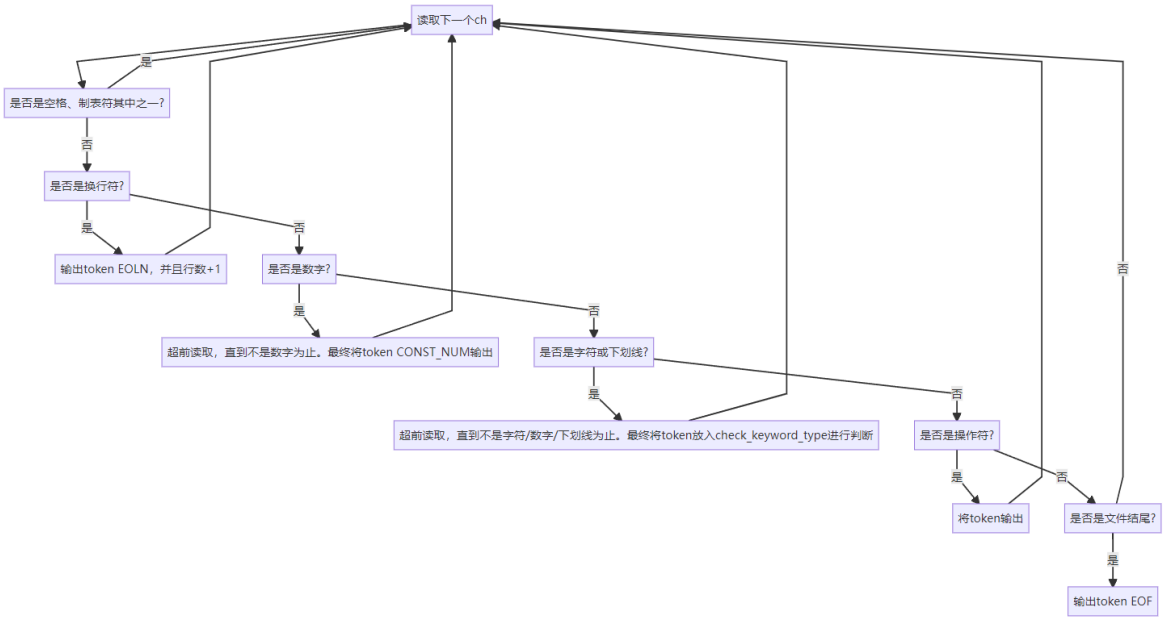
check_keyword_type 函数接收一个字符串，并将其与已经定义好的 keywords 进行匹配。如果匹配,则返回相应的 symbol。如果没有匹配的,则认为该字符串为 identifier，并返回相应的 symbol。

5. 设计实现错误输出函数 put_error()

put_error 函数接收两个参数，即当前的 token 和错误类型。根据错误类型向错误流中输出相应的错误输出。

6. 设计词法分析主函数 scan()

scan 函数从输入流中获取一个字符,并设 flag=false。对字符的处理遵循以下流程。



在 scan 流程中，如果发现错误，则调用 put_error 函数，并设 flag=true。scan 函数返回一个 bool 值，以告诉主控函数词法分析是否成功。

7. 错误类型以及触发条件

syntax_error：在 scan 函数中没有与任何保留字匹配。
invalid_length_error：判定 token 的类型是 identifier，并且其字符长度超过 16 个。
invalid_assign_error：在 token 中检测到了“:”，但是没有“=”与之匹配。

七、实验数据及结果分析：

1. 标准测试文件

该文件的词法是正确的

```
begin
  integer k;
  integer function F(n);
    begin
      integer n;
      if n<=0 then F:=1
      else F:=n*F(n-1)
    end;
  read(m);
  k:=F(m);
  write(k)
end
```

输出结果

```
```

```
  begin  1  1
    EOLN 24  2
integer  3  2
      k 10  2
      ; 23  2
    EOLN 24  3
integer  3  3
function  7  3
      F 10  3
      ( 21  3
      n 10  3
      ) 22  3
      ; 23  3
    EOLN 24  4
  begin  1  4
    EOLN 24  5
integer  3  5
      n 10  5
      ; 23  5
    EOLN 24  6
     if  4  6
      n 10  6
     <= 14  6
      0 11  6
   then  5  6
      F 10  6
     := 20  6
      1 11  6
    EOLN 24  7
   else  6  7
      F 10  7
     := 20  7
      n 10  7
      * 19  7
      F 10  7
      ( 21  7
      n 10  7
      - 18  7
      1 11  7
      ) 22  7
    EOLN 24  8
    end  2  8
      ; 23  8
    EOLN 24  9
   read  8  9
      ( 21  9
      m 10  9
      ) 22  9
      ; 23  9
    EOLN 24 10
      k 10 10
     := 20 10
      F 10 10
      ( 21 10
      m 10 10
      ) 22 10
      ; 23 10
    EOLN 24 11
  write  9 11
      ( 21 11
      k 10 11
      ) 22 11
    EOLN 24 12
    end  2 12
    EOLN 24 13
     EOF 25 13
```

```
```

结论

注：为了方便语法分析报错，因此第三列为所读取的 token 所在的行数

可以发现词法分析的结果是正确的。

2. 错误测试文件

该文件出现了全部三种错误

```
begin
    %%%
    abcdefghijklmnopqrst:1
end
```

错误输出

```
--lexer
syntax error: '%', at line 2
syntax error: '%', at line 2
syntax error: '%', at line 2
identifier or constant is too long: 'abcdefghij...', at line 3
invalid assignment: missing '=', at line 3

```

结论

可以看出所有错误都可以被正确识别。

八、实验结论、心得体会和改进建议：

本次实验成功实现了一个能够读取输入并正常输出结果和报错信息的词法分析程序，并且经过验证功能完备。

通过本次实验，我了解了词法分析的基本原理，并通过实践了解了在具体工程实践中会遇到的问题，对理论结合实际有了更进一步的了解。

电 子 科 技 大 学

实 验 报 告

实验二

一、实验名称：语法分析

二、实验学时：4

三、实验内容和目的：

实现一个语法分析程序，其能够对词法分析生成的二元式文件进行分析，判断其是否符合语法，并输出相应的错误输出。设计思路采用递归下降分析算法。

四、实验原理：

递归下降分析：指对文法的每一非终结符号，都根据相应产生式各候选式的结构，为其编写一个子程序（或函数），用来识别该非终结符号所表示的语法范畴。

LL(1)文法：
（1）文法不含左递归
（2）对文法中每一个非终结符 A 的各个产生式的 FIRST 集合两两不相交。
（3）对文法中每一个非终结符 A，若存在某个 FIRST 集合包含 ε，则： First(A)∩Follow(A)=∅。

本次实验采用的文法：

```
# 1
program : sub_program

# 2
sub_program : begin declaration_table ; execution_table end

# 3
declaration_table : declaration
                  | declaration_table ; declaration

# 3'
declaration_table : declaration declaration_table_prime
```

```
declaration_table_prime : ; declaration declaration_table_prime
                        | null

FOLLOW = {;}

# 4
declaration : variable_declaration
            | function_declaration
            | null

FOLLOW = {;}

# 5
variable_declaration : integer variable

# 6
variable : identifier

identifier : alpha
           | identifier alpha
           | identifier digit

alpha : ...

digit : ...

# 7
function_declaration : integer function identifier(parameter) ; function_body

# 8
parameter : integer variable

# 9
function_body : begin declaration_table ; execution_table end

# 10
execution_table : execution
                | execution_table ; execution

# 10'
execution_table : execution execution_table_prime
execution_table_prime : ; execution execution_table_prime
                      | null

FOLLOW = {;, end}

# 11
```

```
execution : read_statement
          | write_statement
          | assignment
          | condition
          | null

FOLLOW = {;}

# 12
read_statement : read(variable)

# 13
write_statement : write(variable)

# 14
assignment : variable:=expression

# 15
expression : expression-item
          | item

# 15'
expression : item expression_prime
expression_prime : -item expression_prime | null

FOLLOW = {;}

# 16
item : item*factor
       | factor

# 16'
item : factor item_prime
item_prime : *factor item_prime
              | null


# 17
factor : variable
       | constant
       | function_call

constant : unsigned_int
```

```
unsigned_int : digit
               | unsigned_int digit

# 18
condition : if condition_expression then execution else execution

# 19
condition_expression : expression relational_operator expression

# 20
relational_operator : <
                    | <=
                    | >
                    | >=
                    | =
                    | <>

# 21
function_call : identifier(expression)
               | identifier()
```

注：文法经过改造后为 LL(1)文法。

五、实验器材（设备、元器件）

Ubuntu 20.04.3 LTS
G++ version 9.4.0(采用 C++ 17 标准编译，编译时需要指明-std=c++17)
Visual Studio Code

六、实验步骤：

1. **设计实现驱动库 driver.hpp**

见实验 1。

在语法分析部分，增添了变量表和过程表的结构体定义，表本身使用的数据结构为 vector。

同时增加了变量表和过程表的输出文件.var 和.pro 的定义。

增加一个全局变量 level，用于标记当前变量的作用层级。

全局变量 is_checking_function 来确定当前是否在检测声明是不是一个函数。

全局变量 current_function_index 来确定当前函数体的在函数表中的位置。

2. **设计实现一个主函数 main()**

在主函数 main 中增加一个 parser_flag，只有当读取到参数-p 时才能开始使 parser_flag=true。当进行完词法后，如果 parser_flag=true，则将输入流替换为词法分析产生的二元式文件.dyd。

3. 设计实现入口函数 parser()

parser 调用第一个非终结符对应的函数，当该终结符成功归约时，则返回 true，否则返回 false。

4. 设计实现读取函数 parse_next_token()

parse_next_token 函数从词法分析生成的二元式文件中读取一行，并载入全局变量 token。如果读取到了 EOLN，则跳过并读取下一个 token。

5. 设计实现匹配函数 match()

match 接受一个目标 token 的 symbol，与当前 token 的 symbol 匹配。如果匹配则返回 true，否则返回 false。

当 symbol == BEGIN 时，level += 1，current_function_index += 1（表示进入了新的作用域）

当 symbol == END 时，level -= 1，current_function_index -= 1（表示离开了作用域）

6. 设计实现错误输出函数 put_error()

put_error 函数接受一个错误类型，并将对应的错误和当前的 token 输出到错误流之中。

语法分析总共设计了 15 种错误。

MISSING_BEGIN_ERROR：程序体缺少 begin
MISSING_END_ERROR：程序体缺少 end
MISSING_SEMICOLON_ERROR：语句后缺少分号
MISSING_INTEGER_ERROR：变量声明缺少 integer 标识符
MISSING_IDENTIFIER_ERROR：声明后缺少 identifier
MISSING_FUNCTION_ERROR：函数声明缺少 function
MISSING_ROUND_BRACKET_ERROR：缺少“(”或“)”
MISSING_ELSE_ERROR：if 语句缺少 else
ILLEGAL_PARAMETER_ERROR：函数调用的参数形式不合法
BAD_DECLARATION_ERROR：非法声明语句
BAD_EXECUTION_ERROR：非法执行语句
UNDECLARED_VARIABLE_ERROR：未声明的变量
UNDECLARED_FUNCTION_ERROR：未声明的过程
DUPLICATED_VARIABLE_DECLARATION：重复声明的变量
DUPLICATED_FUNCTION_DECLARATION：重复声明的过程

7. 设计实现放入变量表/函数表的函数 put_variable/put_function()

读取当前的 token，并遍历相应的变量表/函数表，如果没有重复，则将当前 token 对应的 identifier 初始化并放入相应表中。

如果成功，则返回 1，否则返回相应的错误码。

8. 设计实现检查变量是否在当前作用域声明过的函数 check_variable()

依据输入的变量名检查给定的变量名是否在当前作用域作用域内有声明。(不同作用

域可以声明自己的变量名，并覆盖更高级作用域的作用名，但不能在同一个作用域重复声明。）

9. 设计实现检查变量是否在表中的函数 is_variable_declared()

依据输入的变量名检查给定的变量名是否在当前作用域作用域或更高级的作用域有声明。

从变量表底部开始遍历，并且要求 level 是递增的或不变时才能比较。（有可能之前的变量是在不同的低级作用域里声明的，用递增来略过这些变量。）

之后，变量名必须相同，其次这个变量在当前作用域或更高级的作用域有声明，才能判断其已经声明。

分成 check_variable 和 is_variable_declared 的原因是，变量声明和使用时的逻辑是不同的。

10. 设计实现检查函数是否在表中的函数 check_function()

依据输入的函数名检查给定的函数名是否在当前作用域作用域或更高级的作用域有声明。

可以在函数内嵌套函数，前提是不能与更高级作用域的函数重名，否则会不知道选择那个函数进行调用。

11. 设计实现非终结符相关的函数

由于这部分逻辑较为复杂，故不一一展开，仅就实现中的要点来讲。

总体逻辑：

遍历非终结符所对应的产生式，试验 token 是否匹配当前产生式。如果完成后没有任何一种产生式与其匹配，则产生了语法错误。

如何处理递归产生式：

例如 declaration 和 execution，这两种非终结符可能是递归的。针对这种产生式，我们需要先将产生式替换为右递归的，并且在这类产生式的判断末尾超前读取一个 token。如果该 token 匹配这类非终结符的开头，那么则继续匹配。如果不匹配，那么在结束时不解析下一个 token，留给其余的非终结符进行判断。

如何处理有相同开头 token 的产生式：

例如在 declaration 中，变量和函数的声明都以 integer 开头，如果直接读取下一个 token，那么会导致其中一个非终结符的判断出现错 误 。 因 此 在 declaration 中，先匹配这两者的共同开头 token integer，再调用二者的产生式判断函数。

如何处理空语句：

有时候 declaration 和 execution 语句为空，这样的程序仍然是合法的。原语法并不支持判空，因此引入一类变量来判断上述类型的语句是否为空。即 is_declaration/execution_blank。当 declaration/execution 没有在开头匹配到任何一个非空产生式，又在最后匹配到了 FOLLOW 集中的终结符(;和 end)时，则说明当前语句为空，这样的语句也是能通过语法检测的。然而没有匹配任何一个非空产生式，也可能是因为在匹配过程中产生了错误，这样的话语句就是不合法的。因此加入一类变量 is_declaration/execution_illegal 来辅助判断。如果该类变量为真，就说明语句不合法。

七、实验数据及结果分析：

1. 测试标准程序

测试程序

```
begin
  integer k;
  integer function F(integer n);
    begin
      if n<=0 then F:=1
      else F:=n*F(n-1)
    end;
  read(m);
  k:=F(m);
  write(k)
end
```

var 表

name	proc	is_parameter	type	level	index
k	main	0	integer	0	0
n	F	1	integer	1	1

pro 表

name	proc	type	level	first_index	last_index
main	main	integer	0	0	0
F	main	integer	1	1	1

测试结果

```
--lexer
--parser
missing ';': getting 'end', at line 7
undeclared variable:'m', at line 8
undeclared variable:'m', at line 9
missing ';': getting 'end', at line 11
```

在函数声明的函数体中，if 语句缺少分号，对应第一条错误信息。
除此之外，变量 m 未声明，对应第二、三条错误信息。
write 语句没有写分号，对应第四条错误信息。

2. 测试修改后的标准程序

我们将第一次测试中的程序的报错改正

测试程序

```
begin
  integer k;
  integer m;
  integer function F(integer n);
    begin
      if n<=0 then F:=1
      else F:=n*F(n-1);
    end;
  read(m);
  k:=F(m);
  write(k);
end
```

var 表

name	proc	is_parameter	type	level	index
k	main	0	integer	0	0
m	main	0	integer	0	1
n	F	1	integer	1	2

pro 表

name	proc	type	level	first_index	last_index
main	main	integer	0	0	1
F	main	integer	1	2	2

测试结果

```
--lexer
--parser
```

完全正确。

3. 测试空语句

测试程序

```
begin
  ;
  ;
  integer function F(integer n);
    begin
    end;
end

```

var 表

name	proc	is_parameter	type	level	index
n	F	1	integer	1	0

pro 表

name	proc	type	level	first_index	last_index
main	main	integer	0	-1	-1
F	main	integer	1	0	0

测试结果

```
--lexer
--parser

```

能够通过语法分析。

5. 测试非法声明

测试程序 1

```
begin
  integer k;
  intger m;
  integer function F(integer n);
    begin
      if n<=0 then F:=1
      else F:=n*F(n-1);
    end;
  read(m);
  k:=F(m);
  write(k);
end

```

var 表

```
name        proc        is_parameter    type        level       index
k           main        0               integer     0           0

```

pro 表

```
name        proc        type        level       first_index last_index
main        main        integer     0           0           0

```

测试结果

```
--lexer
--parser
undeclared variable:'intger', at line 3
illegal execution statements: at line 3

```

可以检测出是非法语句。
注：因为标识符错误，所以没有被认为是一个声明语句，而在执行语句执行报错。

测试程序 2

```
begin
  integer k;
  integer funtion F(integer n);
    begin
      if n<=0 then F:=1
      else F:=n*F(n-1);
    end;
  read(m);
  k:=F(m);
  write(k);
end

```

var 表

```
name        proc        is_parameter    type        level       index
k           main        0               integer     0           0

```

pro 表

```
name        proc        type        level       first_index last_index
main        main        integer     0           0           0

```

测试结果

```
--lexer
--parser
illegal declaration statements: at line 3

```

可以检测出是非法语句。

6. 测试缺少字符时的报错

测试程序 1

```
begin
  integer k;
  integer function F(integer n);
    begin
      if n<=0 then F:=1
      else F:=n*F(n-1);
    end;
  readm);
  k:=F(m);
  write(k);
end

```

var 表

```
name        proc        is_parameter    type        level       index
k           main        0               integer     0           0
n           F           1               integer     1           1

```

pro 表

```
name        proc        type        level       first_index last_index
main        main        integer     0           0           0
F           main        integer     1           1           1

```

测试结果

```
--lexer
--parser
undeclared variable:'readm', at line 8
illegal execution statements: at line 8

```

由于缺少左括号，因此被 readm 认为是一个 identifier，所以报错。
同时这是一个非法的执行语句，因此产生第二行错误。

测试程序 2

```
begin
  integer k;
  integer m;
  integer function F(integer n);
    begin
      if n<=0 then F:=1
      else F:=n*F(n-1);
    end;
  read(m;
  k:=F(m);
  write(k);
end

```

var 表

name	proc	is_parameter	type	level	index
k	main	0	integer	0	0
m	main	0	integer	0	1
n	F	1	integer	1	2

pro 表

name	proc	type	level	first_index	last_index
main	main	integer	0	0	1
F	main	integer	1	2	2

测试结果

```
--lexer
--parser
missing '(' or ')': getting ';', at line 9
illegal execution statements: at line 9

```

第一个错误信息对应了 read 缺少右括号，并且总体而言是一条非法声明语句。

测试程序 3

```
begin
  integer k;
  integer m;
  integer function F(integer n);
    begin
      if n<=0 then F:=1
      else F:=n*F(n-1);
    end;
  read();
  k:=F(m);
  write(k);
end
```

var 表

name	proc	is_parameter	type	level	index
k	main	0	integer	0	0
m	main	0	integer	0	1
n	F	1	integer	1	2

pro 表

name	proc	type	level	first_index	last_index
main	main	integer	0	0	1
F	main	integer	1	2	2

测试结果

```
--lexer
--parser
missing the name of variable or function: getting ')', at line 9
illegal execution statements: at line 9
```

第一个错误信息表明 read 缺少参数。

7. 测试重复声明变量或函数

测试程序 1

```
begin
  integer k;
  integer k;
  integer m;
  integer function F(integer n);
    begin
      if n<=0 then F:=1
      else F:=n*F(n-1);
    end;
  read(m);
  k:=F(m);
  write(k);
end
```

var 表

name	proc	is_parameter	type	level	index
k	main	0	integer	0	0
m	main	0	integer	0	1
n	F	1	integer	1	2

pro 表

name	proc	type	level	first_index	last_index
main	main	integer	0	0	1
F	main	integer	1	2	2

测试结果

```
--lexer
--parser
variable has been declared:'k', at line 3
```

第一个错误信息表明 k 被重复声明。

测试程序 2

```
begin
  integer k;
  integer m;
  integer function F(integer n);
    begin
      if n<=0 then F:=1
      else F:=n*F(n-1);
    end;
  integer function F(integer m);
    begin
    end;
  read(m);
  k:=F(m);
  write(k);
end
```

var 表

name	proc	is_parameter	type	level	index
k	main	0	integer	0	0
m	main	0	integer	0	1
n	F	1	integer	1	2

pro 表

name	proc	type	level	first_index	last_index
main	main	integer	0	0	1
F	main	integer	1	2	2

测试结果

```
--lexer
--parser
function has been declared:'F', at line 9
illegal declaration statements: at line 9
```

第一个错误信息表明函数 F 被重复声明。

8. 测试使用未声明的函数

注：关于未声明的变量见测试 1

测试程序

```
begin
  integer k;
  integer m;
  read(m);
  k:=F(m);
  write(k);
end
```

var 表

name	proc	is_parameter	type	level	index
k	main	0	integer	0	0
m	main	0	integer	0	1

pro 表

name	proc	type	level	first_index	last_index
main	main	integer	0	0	1

测试结果

```
--lexer
--parser
undeclared function:'F', at line 5
```

第一个错误信息表明函数 F 未被声明。

八、实验结论、心得体会和改进建议：

本次实验成功的实现了语法分析程序，其能够对程序是否合法进行判断，并生成相应的错误报告。

通过本次实验，我了解了递归下降分析算法的基本原理，了解了工程实现中各种需要注意的事项，认识了递归下降法分析法具有较多的局限性。例如编写困难，效率较低等。

九、代码

driver.hpp

```
#ifndef DRIVER_HPP
#define DRIVER_HPP

#include <ctype.h>
#include <iostream>
#include <regex>
```

```cpp
#include <string>
#include <vector>

namespace core {
inline FILE* in;
inline FILE* out;
inline FILE* err;
inline FILE* var;
inline FILE* pro;

inline int lines = 1;
}  // namespace core

namespace lexer {
bool scan();
void put_token(std::string s, int symbol);
int check_keyword_type(std::string s);
void put_error(std::string s, int type);
}  // namespace lexer

namespace parser {
/* tools */
bool parser();
void parse_next_token();
bool match(int symbol);
void put_error(int type);

int put_variable(std::string name, bool kind);
bool check_variable(std::string name);       // declaration
bool is_variable_declared(std::string name);  // using a variable
int put_function(std::string name);
bool check_function(std::string name);  // declaration & use
void print_all_table();

/* states */
bool program();
bool sub_program();

bool declaration_table();
bool declaration_table_prime();
bool declaration();

bool variable_declaration();
bool variable();
```

```
bool function_declaration();
bool parameter();
bool function_body();

bool execution_table();
bool execution_table_prime();
bool execution();

bool read_statement();
bool write_statement();
bool assignment();

bool expression();
bool expression_prime();

bool item();
bool item_prime();

bool factor();
bool condition();
bool condition_expression();
bool relational_operator();
bool function_call();
}  // namespace parser

struct variable_table_type {
    std::string name;
    std::string proc;
    bool is_parameter;  // false stands for variable, true stands for parameter
    std::string type;
    int level;
    int index;  // index of the vector
};

struct function_table_type {
    std::string name;
    std::string proc;
    std::string type;  // return type
    int level;
    int first_index;  // first variable index in variable_table
    int last_index;   // last variable index in variable_table
};

inline std::vector<std::string> keywords = {"begin",    "end",  "integer",
                                            "if",       "then", "else",
                                            "function", "read", "write"};
```

```
inline std::vector<struct variable_table_type> variable_table;
inline std::vector<struct function_table_type> function_table;

enum Symbols {
    /* keywords */
    BEGIN = 1,
    END = 2,
    INTEGER = 3,
    IF = 4,
    THEN = 5,
    ELSE = 6,
    FUNCTION = 7,
    READ = 8,
    WRITE = 9,

    /* others */
    IDENTIFIER = 10,
    CONST_NUM = 11,

    /* symbols */
    EQUAL = 12,                 // =
    NOT_EQUAL = 13,             // <>
    LESS_EQUAL = 14,            // <=
    LESS_THAN = 15,             // <
    GREATER_EQUAL = 16,         // >=
    GREATER_THAN = 17,          // >
    MINUS = 18,                 // -
    TIMES = 19,                 // *
    ASSIGN = 20,                // :=
    LEFT_ROUND_BRACKET = 21,    // (
    RIGHT_ROUND_BRACKET = 22,   // )
    SEMICOLON = 23,             // ;
    EOLN = 24,                  // \n
    EOF_ = 25,                  // Avoid confliction
};

enum Errors {
    /* lexer error */
    SYNTAX_ERROR = 255,
    INVALID_LENGTH_ERROR = 256,
    INVALID_ASSIGN_ERROR = 257,

    /* parser error */
```

```
    /* missing keyword */
    MISSING_BEGIN_ERROR = 258,
    MISSING_END_ERROR = 259,
    MISSING_SEMICOLON_ERROR = 260,
    MISSING_INTEGER_ERROR = 261,
    MISSING_IDENTIFIER_ERROR = 262,
    MISSING_FUNCTION_ERROR = 263,
    MISSING_ROUND_BRACKET_ERROR = 264,
    MISSING_ELSE_ERROR = 265,

    ILLEGAL_PARAMETER_ERROR = 266,
    BAD_DECLARATION_ERROR = 267,  // debug
    BAD_EXECUTION_ERROR = 268,    // debug0

    /* variable-table & function-table */
    UNDECLARED_VARIABLE_ERROR = 269,
    UNDECLARED_FUNCTION_ERROR = 270,
    DUPLICATED_VARIABLE_DECLARATION = 271,
    DUPLICATED_FUNCTION_DECLARATION = 272,
};

#endif
```

main.cpp

```
#include "driver.hpp"

// FILE* in = nullptr;
// FILE* out = nullptr;
// FILE* err = nullptr;

int main(int argc, char* argv[]) {
    --argc, ++argv;  // Remove program name

    bool parser_flag = false;

    /* release */
    if (argc == 0) {
        core::in = stdin;  // no parameter
    } else if (argc == 1) {
        if (0 == strcmp(argv[0], "-p")) {  // 1 parameter
            core::in = stdin;
            printf("can not do parse when using stdin\n");
        } else
            core::in = fopen(argv[0], "r");
    } else if (argc == 2) {
        for (int i = 0; i < argc; i++) {  // 2 parameter
            std::string temp = argv[i];
            if ("-p" == temp)
```

```
            parser_flag = true;
        else {
            core::in = fopen(argv[i], "r");
        }
    }
} else {
    printf("too many parameter\n");
    return 0;
}

/* debug */
// core::in = fopen("../test/test1.pas", "r");
// parser_flag = true;

core::out = fopen("middle.dyd", "w");

core::err = fopen("middle.err", "w");  // Clean middle.err
core::err = fopen("middle.err", "a");

core::var = fopen("middle.var", "w");
core::pro = fopen("middle.pro", "w");

if (lexer::scan()) {
    if (parser_flag) {
        fclose(core::in);
        core::in = fopen("middle.dyd", "r");
        fclose(core::out);
        core::out = fopen("middle.dys", "w");

        if (parser::parser())
            printf("parse success\n");
        else
            printf("parse failed\n");
    }

    printf("compile done\n");
} else {
    printf("lexical analysis failed\n");
}

fclose(core::in);
fclose(core::out);
fclose(core::err);
fclose(core::var);
```

```
    fclose(core::pro);

    return 0;
}
```

lexer. cpp

```
#include "driver.hpp"

/**
 * @brief A main function to scan a character everytime from input file.
 * Check if the character forms a valid token, and do output.
 *
 */
bool lexer::scan() {
    printf("\nlexer begin\n");
    fprintf(core::err, "--lexer\n");

    std::string token = "";
    bool flag = false;

    char ch = fgetc(core::in);
    while (!feof(core::in)) {
        if (' ' == ch || '\t' == ch || '\r' == ch) {
            ch = fgetc(core::in);
        } else if ('\n' == ch) {
            core::lines++;
            put_token("EOLN", EOLN);
            ch = fgetc(core::in);
        } else if (isdigit(ch)) {
            // It may be a constant
            // leave it to parser to judge if it is legal
            token.clear();
            while (isdigit(ch)) {
                token += ch;
                ch = fgetc(core::in);
            }

            put_token(token, CONST_NUM);
        } else if (isalpha(ch) || '_' == ch) {
            // It may be a identifier or a keyword
            token.clear();
            while (isalpha(ch) || isdigit(ch) || '_' == ch) {
                token += ch;
                ch = fgetc(core::in);
            }

            int symbol = check_keyword_type(token);
```

```
            if (token.length() > 16) {
                put_error(token, INVALID_LENGTH_ERROR);
                flag = true;
            } else
                put_token(token, symbol);
        } else {
            token.clear();
            token += ch;

            int symbol = SYNTAX_ERROR;

            switch (ch) {
                case '=':
                    ch = fgetc(core::in);
                    symbol = EQUAL;
                    break;
                case '<':
                    ch = fgetc(core::in);
                    if ('>' == ch) {
                        token += ch;
                        symbol = NOT_EQUAL;
                        ch = fgetc(core::in);
                    } else if ('=' == ch) {
                        token += ch;
                        symbol = LESS_EQUAL;
                        ch = fgetc(core::in);
                    } else {
                        symbol = LESS_THAN;
                    }

                    break;
                case '>':
                    ch = fgetc(core::in);
                    if ('=' == ch) {
                        token += ch;
                        symbol = GREATER_EQUAL;
                        ch = fgetc(core::in);
                    } else {
                        symbol = GREATER_THAN;
                    }

                    break;
                case '-':
                    ch = fgetc(core::in);
```

```
                symbol = MINUS;
                break;
            case '*':
                ch = fgetc(core::in);
                symbol = TIMES;
                break;
            case ':':
                ch = fgetc(core::in);
                if ('=' == ch) {
                    token += ch;
                    symbol = ASSIGN;
                    ch = fgetc(core::in);
                } else {
                    symbol = INVALID_ASSIGN_ERROR;
                }
                break;
            case '(':
                ch = fgetc(core::in);
                symbol = LEFT_ROUND_BRACKET;
                break;
            case ')':
                ch = fgetc(core::in);
                symbol = RIGHT_ROUND_BRACKET;
                break;
            case ';':
                ch = fgetc(core::in);
                symbol = SEMICOLON;
                break;
            default:
                ch = fgetc(core::in);
                break;
        }

        if (symbol >= 255) {
            put_error(token, symbol);
            flag = true;
        } else
            put_token(token, symbol);
    }
}

put_token("EOF", EOF_);

if (flag)
    return false;
else
    return true;
```

```
}

/**
 * @brief A function to put a formatted output into output file.
 *
 * @param s The name of the token
 * @param symbol The number of that symbol, check driver.hpp for more
 *
 */
void lexer::put_token(std::string s, int symbol) {
    printf("putting a token %s:%d\n", s.c_str(), symbol);

    fprintf(core::out, "%16s %2d %2d\n", s.c_str(), symbol, core::lines);
}

/**
 * @brief A function to check the token type.
 *
 * @param s The token
 *
 * @return The corresponding number of the token, -1 as error
 *
 */
int lexer::check_keyword_type(std::string s) {
    for (auto i = 0; i < keywords.size(); ++i) {
        if (s == keywords[i]) {
            return i + 1;
        }
    }

    return IDENTIFIER;
}

/**
 * @brief A function to put a formatted output into error file.
 *
 * @param s The word that caused an error
 * @param type The error type
 *
 */
void lexer::put_error(std::string s, int type) {
    printf("putting an error %s:%d\n", s.c_str(), type);

    switch (type) {
```

```
        case SYNTAX_ERROR:
            fprintf(core::err, "syntax error: '%s', at line %d\n", s.c_str(),
                    core::lines);
            break;
        case INVALID_LENGTH_ERROR:
            fprintf(
                core::err,
                "identifier or constant is too long: '%.10s...', at line %d\n",
                s.c_str(), core::lines);
            break;
        case INVALID_ASSIGN_ERROR:
            fprintf(core::err, "invalid assignment: missing '=', at line %d\n",
                    core::lines);
            break;
    }
}
```

parser.cpp

```
#include "driver.hpp"

/*
 * token[0]: the token name
 * token[1]: the type number
 * token[2]: token location
 */
std::vector<std::string> token;

/*
 * save the parsed info
 * like variable name or function name
 */
std::string context;

/*
 * to mark the scope of a variable;
 * when match a begin, it plus 1 (entering the scope)
 * when match a end, it minus 1 (leaving the scope)
 */
int level = -1;

/*
 * sometimes declaration or execution can be an empty statement
 * but next parser cant check that only through tokens, so we need
 * a new variable
 */
/*
 * and sometime the declaration/execution are not blank
```

```
 * but they are illegal
 */
bool is_declaration_illegal = false;
bool is_declaration_blank = false;
bool is_execution_illegal = false;
bool is_execution_blank = false;

/*
 * matching integer identifier doesn't mean that
 * it is a variable declaration, it might be a wrong function
 * declaration.
 *
 * if we match variable declaration, we need to parse next
 * token to check if it is a wrong function declaration,
 * which breaks the statement scope.
 *
 * so, we need a bool to mark this situation
 */
bool is_checking_function = false;

/*
 * when generating a variable, it needs to know
 * which proc it is in
 */

int current_function_index = -1;

/*
 * when on condition, execution doesn't need a ';'
 * which will course a disruption
 */
// bool on_condition = false;

bool parser::parser() {
    printf("\nparser begin\n");
    fprintf(core::err, "--parser\n");

    if (program()) {
        print_all_table();
        return true;
    } else {
        print_all_table();
        return false;
    }
```

```
}

void parser::parse_next_token() {
loop:
    char* temp;
    size_t len = 0;

    token.clear();

    getline(&temp, &len, core::in);

    int i = 0;
    for (auto index = 0; index < 3; ++index) {
        std::string temp_string = "";
        while (temp[i] == ' ')
            ++i;
        while (temp[i] != ' ' && temp[i] != '\n') {
            temp_string += temp[i];
            ++i;
        }
        token.push_back(temp_string);
    }

    if (atoi(token[1].c_str()) == EOLN) {  // skip '\n'
        free(temp);
        goto loop;
    }
}

bool parser::match(int symbol) {
    if (atoi(token[1].c_str()) == symbol) {
        if (symbol == BEGIN) {
            level++;
        } else if (symbol == END) {
            if (function_table[current_function_index].level == 1) {
                current_function_index = 0;  // jump back to main function
            } else {
                current_function_index--;
            }
            level--;
        }
        return true;
    } else
        return false;
}

void parser::put_error(int type) {
```

```
    printf("putting an error :%d\n\n", type);

    switch (type) {
        case MISSING_BEGIN_ERROR:
            fprintf(core::err, "missing begin: getting '%s', at line %s\n",
                    token[0].c_str(), token[2].c_str());
            break;
        case MISSING_END_ERROR:
            fprintf(core::err, "missing end: getting '%s', at line %s\n",
                    token[0].c_str(), token[2].c_str());
            break;
        case MISSING_SEMICOLON_ERROR:
            fprintf(core::err, "missing ';': getting '%s', at line %s\n",
                    token[0].c_str(), token[2].c_str());
            break;
        case MISSING_INTEGER_ERROR:
            fprintf(core::err,
                    "missing the type of variable: getting '%s', at line %s\n",
                    token[0].c_str(), token[2].c_str());
            break;
        case MISSING_IDENTIFIER_ERROR:
            fprintf(
                core::err,
                "missing the name of variable or function: getting '%s', at "
                "line %s\n",
                token[0].c_str(), token[2].c_str());
            break;
        case MISSING_FUNCTION_ERROR:
            fprintf(core::err, "missing 'function': at line %s\n",
                    token[2].c_str());
            break;
        case MISSING_ROUND_BRACKET_ERROR:
            fprintf(core::err, "missing '(' or ')': getting '%s', at line %s\n",
                    token[0].c_str(), token[2].c_str());
            break;
        case MISSING_ELSE_ERROR:
            fprintf(core::err, "missing 'else': getting '%s', at line %s\n",
                    token[0].c_str(), token[2].c_str());
            break;
        case ILLEGAL_PARAMETER_ERROR:
            fprintf(core::err, "illegal parameter statements: at line %s\n",
                    token[2].c_str());
            break;
        case BAD_DECLARATION_ERROR:
```

```
            fprintf(core::err, "illegal declaration statements: at line %s\n",
                    token[2].c_str());
            break;
        case BAD_EXECUTION_ERROR:
            fprintf(core::err, "illegal execution statements: at line %s\n",
                    token[2].c_str());
            break;
        case UNDECLARED_VARIABLE_ERROR:
            fprintf(core::err, "undeclared variable:'%s', at line %s\n",
                    context.c_str(), token[2].c_str());
            break;
        case UNDECLARED_FUNCTION_ERROR:
            fprintf(core::err, "undeclared function:'%s', at line %s\n",
                    context.c_str(), token[2].c_str());
            break;
        case DUPLICATED_VARIABLE_DECLARATION:
            fprintf(core::err, "variable has been declared:'%s', at line %s\n",
                    context.c_str(), token[2].c_str());
            break;
        case DUPLICATED_FUNCTION_DECLARATION:
            fprintf(core::err, "function has been declared:'%s', at line %s\n",
                    context.c_str(), token[2].c_str());
            break;
    }
}

int parser::put_variable(std::string name, bool kind) {
    struct variable_table_type temp;
    temp.name = name;
    temp.proc = function_table[current_function_index].name;
    if (kind) {
        temp.level = level + 1;
    } else {
        temp.level = level;
    }
    temp.is_parameter = kind;
    temp.type = "integer";
    temp.index = variable_table.size();

    if (kind) {
        variable_table.push_back(temp);
        return 1;
    }

    if (!check_variable(name)) {
        variable_table.push_back(temp);
        return 1;
```

```
    } else {
        return DUPLICATED_VARIABLE_DECLARATION;
    }
}

bool parser::check_variable(std::string name) {
    for (int i = variable_table.size() - 1; i >= 0; i--) {
        struct variable_table_type temp = variable_table[i];
        if (name == temp.name) {
            if (function_table[current_function_index].name == temp.proc &&
                level == temp.level) {
                // current function has declared this variable(including
                // parameter)
                return true;
            }
        }
    }

    return false;
}

bool parser::is_variable_declared(std::string name) {
    int last_level = 2147483647;

    for (int i = variable_table.size() - 1; i >= 0; i--) {
        struct variable_table_type temp = variable_table[i];
        if (temp.level > last_level ||
            temp.level > function_table[current_function_index].level) {
            continue;
        }

        if (name == temp.name) {
            if (temp.proc == function_table[current_function_index].name) {
                return true;
            } else if (temp.level <
                       function_table[current_function_index].level) {
                return true;
            }
        }

        last_level = temp.level;
    }

    return false;
```

```
}

int parser::put_function(std::string name) {
    struct function_table_type temp;
    temp.name = name;
    if (function_table.size() == 0) {
        temp.proc = "main";
    } else {
        temp.proc = function_table[current_function_index].name;
    }
    temp.type = "integer";
    temp.level = level + 1;
    temp.first_index = -1;
    temp.last_index = -1;

    if (!check_function(name)) {
        function_table.push_back(temp);
        return 1;
    } else {
        return DUPLICATED_FUNCTION_DECLARATION;
    }
}
bool parser::check_function(std::string name) {
    int last_level = 2147483647;

    for (int i = function_table.size() - 1; i >= 0; i--) {
        struct function_table_type temp = function_table[i];

        /*
         * function's level is 1 higher than its direct outer function
         * but it must be seen as a function declared in 1 level lower than its
         * level
         *
         * for example
         *
         * function main
         *  begin
         *      function p
         *  end
         *
         * main at level 0, p at level 1
         * however, p belongs to main(level 0)
         */
        int true_level = temp.level - 1;

        if (true_level > last_level ||
            true_level > function_table[current_function_index].level) {
```

```
            continue;
        }

        if (name == temp.name) {
            if (function_table[current_function_index].name == name) {
                // function identifier used as a return value
                // or a function trying to redeclare itself
                return true;
            } else if (true_level < level) {
                // higher level has declared this function
                return true;
            } else if (function_table[current_function_index].name ==
                        temp.proc &&
                    level == true_level) {
                // current function has been declared under same scope
                return true;
            }
        }

        last_level = true_level;
    }

    return false;
}

void parser::print_all_table() {
    fprintf(core::var, "%-10s\t%-10s\t%-15s\t%-10s\t%-10s\t%-10s\n", "name",
            "proc", "is_parameter", "type", "level", "index");
    for (int i = 0; i < variable_table.size(); i++) {
        struct variable_table_type temp = variable_table[i];
        fprintf(core::var, "%-10s\t%-10s\t%-15d\t%-10s\t%-10d\t%-10d\n",
                temp.name.c_str(), temp.proc.c_str(), temp.is_parameter,
                temp.type.c_str(), temp.level, temp.index);
    }

    fprintf(core::pro, "%-10s\t%-10s\t%-10s\t%-10s\t%-10s\t%-10s\n", "name",
            "proc", "type", "level", "first_index", "last_index");
    for (int i = 0; i < function_table.size(); i++) {
        struct function_table_type temp = function_table[i];
        fprintf(core::pro, "%-10s\t%-10s\t%-10s\t%-10d\t%-10d\t%-10d\n",
                temp.name.c_str(), temp.proc.c_str(), temp.type.c_str(),
                temp.level, temp.first_index, temp.last_index);
    }
}
```

```
bool parser::program() {
    printf("program\n\n");

    put_function("main");  // generate main function
    parse_next_token();
    if (sub_program()) {
        return true;
    } else {
        return false;
    }
}

bool parser::sub_program() {
    printf("sub_program\n");
    printf("parsing '%s'\n\n", token[0].c_str());

    if (match(BEGIN)) {
        current_function_index = 0;  // point to main function
        parse_next_token();
        if (declaration_table()) {
            // to check declaration is ended, we have move to next token
            // behind
            // ';'
            // so we dont check ';'
            if (execution_table()) {
                if (match(END)) {
                    return true;
                } else {
                    put_error(MISSING_END_ERROR);
                    goto bad;
                }
            } else {
                goto bad;
            }
        } else {
            goto bad;
        }
    } else {
        put_error(MISSING_BEGIN_ERROR);
        goto bad;
    }

bad:
    return false;
}
```

```
bool parser::declaration_table() {
    printf("declaration_table\n");
    printf("parsing '%s'\n\n", token[0].c_str());

    if (declaration()) {
        if (is_declaration_blank || is_checking_function) {
            // declaration is blank
            // then token stops at ';' or 'end' or something else
            // no need to parse next token
            // remain is_declaration_blank true
            is_checking_function = false;
        } else {
            // declaration is not blank and legal
            // so go next token to check if it is a loop
            parse_next_token();
        }
        if (declaration_table_prime()) {
            is_declaration_blank = false;
            is_declaration_illegal = false;
            return true;
        } else {
            goto bad;
        }
    } else {
        put_error(BAD_DECLARATION_ERROR);
        goto bad;
    }

bad:
    is_declaration_blank = false;
    is_declaration_illegal = false;
    return false;
}

bool parser::declaration_table_prime() {
    printf("declaration_table_prime\n");
    printf("parsing '%s'\n\n", token[0].c_str());

    if (match(SEMICOLON)) {
        parse_next_token();
        if (declaration()) {
            if (is_declaration_blank || is_checking_function) {
                is_checking_function = false;
            } else {
```

```
            parse_next_token();
        }
        if (declaration_table_prime()) {
            is_declaration_blank = false;
            is_declaration_illegal = false;
            return true;
        } else {
            goto bad;
        }
    } else if (is_declaration_illegal) {
        put_error(BAD_DECLARATION_ERROR);
        is_declaration_illegal = false;
        return false;
    }
    // matching the ';' or nothing already prove legal
    // if there is something behind the ';' and it's not declaration
    // then we have left the scope, leave the token to next checker
} else if (is_declaration_blank) {
    // declaration is blank, and no more declaration
    is_declaration_blank = false;
    return true;
} else {
    // declaration is not blank, but it doesn't end with ';'
    put_error(MISSING_SEMICOLON_ERROR);
    return true;  // continue
}

bad:
    is_declaration_blank = false;
    is_declaration_illegal = false;
    return false;
}

bool parser::declaration() {
    printf("declaration\n");
    printf("parsing '%s'\n\n", token[0].c_str());

    if (match(INTEGER)) {
        is_declaration_blank = false;
        is_declaration_illegal = false;
        parse_next_token();
        // variable_declaration and function_declaration
        // both begin with integer
        if (variable_declaration()) {
            is_checking_function = true;
            // check if it is a function declaration
            context = token[0];  // save variable name
```

```
            parse_next_token();
            if (match(LEFT_ROUND_BRACKET)) {
                is_declaration_illegal = true;
                put_error(MISSING_FUNCTION_ERROR);  // lack 'function' as a
                                                    // declaration
                goto bad;
            } else if (match(IDENTIFIER)) {
                is_declaration_illegal = true;  // program try to declare a
                                                // function with a wrong spell
                goto bad;
            }

            if (put_variable(context, 0) == DUPLICATED_VARIABLE_DECLARATION) {
                put_error(DUPLICATED_VARIABLE_DECLARATION);
            } else {
                if (function_table[current_function_index].first_index == -1)
{
                    // first variable declaration in this function
                    function_table[current_function_index].first_index =
                        variable_table.size() - 1;
                }
                function_table[current_function_index].last_index =
                    variable_table.size() - 1;
            }  // add as a variable

            return true;
        } else if (function_declaration()) {
            return true;
        } else {
            // illegal declaration
            is_declaration_illegal = true;
            goto bad;
        }
    } else {
        // match nothing
        is_declaration_blank = true;
        return true;
    }

bad:
    return false;
}

bool parser::variable_declaration() {
```

```
    printf("variable_declaration\n");
    printf("parsing '%s'\n\n", token[0].c_str());

    if (variable()) {
        // parse_next_token();
        // if(match(LEFT_ROUND_BRACKET)){
        //     // it might be a function
        // }
        return true;
    } else {
        goto bad;
    }

bad:
    return false;
}

bool parser::variable() {
    printf("variable\n");
    printf("parsing '%s'\n\n", token[0].c_str());

    if (match(IDENTIFIER)) {
        return true;
    } else {
        goto bad;
    }

bad:
    return false;
}

bool parser::function_declaration() {
    printf("function_declaration\n");
    printf("parsing '%s'\n\n", token[0].c_str());

    if (match(FUNCTION)) {
        parse_next_token();
        if (match(IDENTIFIER)) {
            context = token[0];
            if (put_function(token[0]) == DUPLICATED_FUNCTION_DECLARATION) {
                put_error(DUPLICATED_FUNCTION_DECLARATION);
                goto bad;
            }
            current_function_index =
                function_table.size() - 1;  // enter a new function
            parse_next_token();
            if (match(LEFT_ROUND_BRACKET)) {
```

```
                parse_next_token();
                if (parameter()) {  // parameter can be nothing
                    parse_next_token();
                    if (match(RIGHT_ROUND_BRACKET)) {
                        parse_next_token();
                        if (match(SEMICOLON)) {
                            parse_next_token();
                            if (function_body()) {
                                return true;
                            } else {
                                goto bad;
                            }
                        } else {
                            put_error(MISSING_SEMICOLON_ERROR);
                            goto bad;
                        }
                    } else {
                        put_error(MISSING_ROUND_BRACKET_ERROR);
                        goto bad;
                    }
                }
            } else {
                put_error(MISSING_ROUND_BRACKET_ERROR);
                goto bad;
            }
        } else {
            put_error(MISSING_IDENTIFIER_ERROR);
            goto bad;
        }
    } else {
        put_error(MISSING_IDENTIFIER_ERROR);
        return false;
    }

bad:
    is_declaration_illegal = true;
    return false;
}

bool parser::parameter() {
    printf("parameter\n");
    printf("parsing '%s'\n\n", token[0].c_str());

    if (match(INTEGER)) {
```

```
        parse_next_token();
        if (variable()) {
            put_variable(token[0], 1);  // add as a parameter
            if (function_table[current_function_index].first_index == -1) {
                // fisrt parameter
                function_table[current_function_index].first_index =
                    variable_table.size() - 1;
            }
            function_table[current_function_index].last_index =
                variable_table.size() - 1;

            return true;
        } else {
            goto bad;
        }
    } else {
        put_error(ILLEGAL_PARAMETER_ERROR);
        goto bad;
    }

bad:
    return false;
}

bool parser::function_body() {
    printf("function_body\n");
    printf("parsing '%s'\n\n", token[0].c_str());

    if (match(BEGIN)) {
        parse_next_token();
        if (declaration_table()) {
            if (execution_table()) {
                if (match(END)) {
                    return true;
                } else {
                    put_error(MISSING_END_ERROR);
                    goto bad;
                }
            } else {
                goto bad;
            }
        } else {
            goto bad;
        }
    } else {
        put_error(MISSING_BEGIN_ERROR);
        goto bad;
```

```
    }
bad:
    return false;
}

bool parser::execution_table() {
    printf("execution_table\n");
    printf("parsing '%s'\n\n", token[0].c_str());

    if (execution()) {
        // execution need to look forward to check
        // if the loop is ended
        // so execution() makes the decisions about
        // how to parse next token
        if (execution_table_prime()) {
            is_execution_blank = false;
            is_execution_illegal = false;
            return true;
        } else {
            goto bad;
        }
    } else {
        put_error(BAD_EXECUTION_ERROR);
        goto bad;
    }

bad:
    is_execution_blank = false;
    is_declaration_illegal = false;
    return false;
}

bool parser::execution_table_prime() {
    printf("execution_table_prime\n");
    printf("parsing '%s'\n\n", token[0].c_str());

    if (match(SEMICOLON)) {
        parse_next_token();
        if (execution()) {
            // execution need to look forward to check
            // if the loop is ended
            // so execution() makes the decisions about
            // how to parse next token
            if (execution_table_prime()) {
```

```
                is_execution_blank = false;
                is_execution_illegal = false;
                return true;
            } else {
                goto bad;
            }
        } else if (is_execution_illegal) {
            put_error(BAD_EXECUTION_ERROR);
            is_execution_illegal = false;
            return false;
        }
        // matching the ';' already prove legal
        // if there is something behind the ';' and it's not execution
        // then we have left the scope, leave the token to next checker
    } else if (is_execution_blank) {
        // execution is blank, and no more execution
        is_execution_blank = false;
        return true;
        // } else if (on_condition && match(END)){
        //     on_condition = false;
        //     return true;
    } else {
        // execution is not blank, but it doesn't end with ';'
        put_error(MISSING_SEMICOLON_ERROR);
        return true;
    }

bad:
    is_execution_blank = false;
    is_execution_illegal = false;
    return false;
}

bool parser::execution() {
    printf("execution\n");
    printf("parsing '%s'\n\n", token[0].c_str());

    if (read_statement()) {
        is_execution_blank = false;
        parse_next_token();
        return true;
    } else if (write_statement()) {
        is_execution_blank = false;
        parse_next_token();
        return true;
    } else if (assignment()) {
        is_execution_blank = false;
```

```
        // assignment need to look forward to check
        // if it is ended
        return true;
    } else if (condition()) {
        is_execution_blank = false;
        return true;
    } else if (!is_execution_illegal) {
        // no match and no illegal
        // so execution is blank
        is_execution_blank = true;
        return true;
    } else {
        // illegal
        goto bad;
    }

bad:
    return false;
}

bool parser::read_statement() {
    printf("read_statement\n");
    printf("parsing '%s'\n\n", token[0].c_str());

    if (match(READ)) {
        parse_next_token();
        if (match(LEFT_ROUND_BRACKET)) {
            parse_next_token();
            if (variable()) {
                if (!is_variable_declared(token[0])) {
                    context = token[0];
                    put_error(UNDECLARED_VARIABLE_ERROR);
                }
                parse_next_token();
                if (match(RIGHT_ROUND_BRACKET)) {
                    return true;
                } else {
                    put_error(MISSING_ROUND_BRACKET_ERROR);
                    is_execution_illegal = true;
                    goto bad;
                }
            } else {
                put_error(MISSING_IDENTIFIER_ERROR);
                is_execution_illegal = true;
```

```
                goto bad;
            }
        } else {
            put_error(MISSING_ROUND_BRACKET_ERROR);
            is_execution_illegal = true;
            goto bad;
        }
    } else {
        goto bad;
    }

bad:
    return false;
}

bool parser::write_statement() {
    printf("write_statement\n");
    printf("parsing '%s'\n\n", token[0].c_str());

    if (match(WRITE)) {
        parse_next_token();
        if (match(LEFT_ROUND_BRACKET)) {
            parse_next_token();
            if (variable()) {
                if (!is_variable_declared(token[0])) {
                    context = token[0];
                    put_error(UNDECLARED_VARIABLE_ERROR);
                }
                parse_next_token();
                if (match(RIGHT_ROUND_BRACKET)) {
                    return true;
                } else {
                    put_error(MISSING_ROUND_BRACKET_ERROR);
                    is_execution_illegal = true;
                    goto bad;
                }
            } else {
                put_error(MISSING_IDENTIFIER_ERROR);
                is_execution_illegal = true;
                goto bad;
            }
        } else {
            put_error(MISSING_ROUND_BRACKET_ERROR);
            is_execution_illegal = true;
            goto bad;
        }
    } else {
```

```
        goto bad;
    }

bad:
    return false;
}

bool parser::assignment() {
    printf("assignment\n");
    printf("parsing '%s'\n\n", token[0].c_str());

    if (variable()) {
        if (!is_variable_declared(token[0]) &&
            token[0] != function_table[current_function_index].name) {
            // if it is not a variable
            // it could be the function tries to define its return value
            // if not, cause an error
            context = token[0];
            put_error(UNDECLARED_VARIABLE_ERROR);
        }
        parse_next_token();
        if (match(ASSIGN)) {
            parse_next_token();
            if (expression()) {
                return true;
            } else {
                is_execution_illegal = true;
                goto bad;
            }
        } else {
            is_execution_illegal = true;
            goto bad;
        }
    } else {
        goto bad;
    }

bad:
    return false;
}

bool parser::expression() {
    printf("expression\n");
    printf("parsing '%s'\n\n", token[0].c_str());
```

```
    if (item()) {
        if (expression_prime()) {
            return true;
        } else {
            // expression is a combinations of items
            // even it is followed by a illegal token
            // the scanned part is legal
            // leave the illegal part to upper class
            return true;
        }
    } else {
        goto bad;
    }

bad:
    return false;
}

bool parser::expression_prime() {
    printf("expression_prime\n");
    printf("parsing '%s'\n\n", token[0].c_str());

    if (match(MINUS)) {
        parse_next_token();
        if (item()) {
            if (expression_prime()) {
                return true;
            } else {
                // expression is a combinations of items
                // even it is followed by a illegal token
                // the scanned part is legal
                // leave the illegal part to upper class
                return true;
            }
        } else {
            goto bad;
        }
    } else if (match(SEMICOLON)) {
        // no more item
        return true;
    } else {
        goto bad;
    }

bad:
    return false;
```

```
}

bool parser::item() {
    printf("item\n");
    printf("parsing '%s'\n\n", token[0].c_str());

    if (factor()) {
        if (item_prime()) {
            return true;
        } else {
            // item is a combinations of factors
            // even it is followed by a illegal token
            // the scanned part is legal
            // leave the illegal part to upper class
            return true;
        }
    } else {
        goto bad;
    }

bad:
    return false;
}

bool parser::item_prime() {
    printf("item_prime\n");
    printf("parsing '%s'\n\n", token[0].c_str());

    if (match(TIMES)) {
        parse_next_token();
        if (factor()) {
            if (item_prime()) {
                return true;
            } else {
                goto bad;
            }
        } else {
            goto bad;
        }
    } else if (match(MINUS)) {
        // no more factor
        return true;
    } else if (match(SEMICOLON)) {
        // no more factor
```

```
        return true;
    } else {
        goto bad;
    }

bad:
    return false;
}

bool parser::factor() {
    printf("factor\n");
    printf("parsing '%s'\n\n", token[0].c_str());

    if (variable()) {
        // it may be a function_call
        std::string temp = token[0];

        parse_next_token();
        if (match(LEFT_ROUND_BRACKET)) {
            if (function_call()) {
                if (!check_function(temp)) {
                    // function undeclared
                    context = temp;
                    put_error(UNDECLARED_FUNCTION_ERROR);
                }
                parse_next_token();
                return true;
            } else {
                goto bad;
            }
        } else {
            if (!is_variable_declared(temp)) {
                context = temp;
                put_error(UNDECLARED_VARIABLE_ERROR);
            }
            // not a function call
            // return anyway
            return true;
        }
    } else if (match(CONST_NUM)) {
        parse_next_token();
        return true;
    } else {
        goto bad;
    }

bad:
```

```
    return false;
}

bool parser::condition() {
    printf("condition\n");
    printf("parsing '%s'\n\n", token[0].c_str());

    if (match(IF)) {
        parse_next_token();
        if (condition_expression()) {
            if (match(THEN)) {
                parse_next_token();
                if (execution()) {
                    if (match(ELSE)) {
                        parse_next_token();
                        if (execution()) {
                            return true;
                        } else {
                            is_execution_illegal = true;
                            goto bad;
                        }
                    } else {
                        put_error(MISSING_ELSE_ERROR);
                        is_execution_illegal = true;
                        goto bad;
                    }
                } else {
                    is_execution_illegal = true;
                    goto bad;
                }
            } else {
                is_execution_illegal = true;
                goto bad;
            }
        } else {
            is_execution_illegal = true;
            goto bad;
        }
    } else {
        goto bad;
    }

bad:
    return false;
```

```
}

bool parser::condition_expression() {
    printf("condition_expression\n");
    printf("parsing '%s'\n\n", token[0].c_str());

    if (expression()) {
        if (relational_operator()) {
            parse_next_token();
            if (expression()) {
                return true;
            } else {
                goto bad;
            }
        } else {
            goto bad;
        }
    } else {
        goto bad;
    }
bad:
    return false;
}

bool parser::relational_operator() {
    printf("relational_operator\n");
    printf("parsing '%s'\n\n", token[0].c_str());

    if (match(LESS_THAN)) {
        return true;
    } else if (match(LESS_EQUAL)) {
        return true;
    } else if (match(GREATER_THAN)) {
        return true;
    } else if (match(GREATER_EQUAL)) {
        return true;
    } else if (match(EQUAL)) {
        return true;
    } else if (match(NOT_EQUAL)) {
        return true;
    } else {
        goto bad;
    }
bad:
    return false;
}
```

```
bool parser::function_call() {
    printf("function_call\n");
    printf("parsing '%s'\n\n", token[0].c_str());

    if (match(LEFT_ROUND_BRACKET)) {
        parse_next_token();
        if (expression()) {
            if (match(RIGHT_ROUND_BRACKET)) {
                return true;
            } else {
                put_error(MISSING_ROUND_BRACKET_ERROR);
                goto bad;
            }
        } else if (match(RIGHT_ROUND_BRACKET)) {
            // there is no parameter
            return true;
        } else {
            goto bad;
        }
    } else {
        goto bad;
    }

bad:
    return false;
}
```