语法分析器设计说明

学号：1952650

姓名：陈子翔

目 录

语法分析器题目描述

要求:

1. 对类 C 语法规则，自选语法分析方法，设计语法分析器，输入源程序，分析程序语法是否正确，给出分析过程。
2. 语法分析器在需要的时候调用词法分析器返回单词符号, 供语法分析器使用。
3. 扩充功能：增加一定的出错处理能力。

类 C 词法规则:

1. 关键字：int | void | if | else | while | return
2. 标识符：字母（字母|数字）*（注：不与关键字相同）
3. 数值：数字（数字）*
4. 赋值号：=
5. 算符：+ | - | * | / | = | == | > | >= | < | <= | !=
6. 界符：；
7. 分隔符：，
8. 注释号：/* */ | //

类 C 语法规则:

1. <if 语句> ::= if ‘(‘<表达式>’)’ <语句块> [else <语句块>]（注：[ ]中的项表示可选）
2. <表达式>::=<加法表达式>{ relop <加法表达式>}(注：relop-><|<=|>|>=|==|!=)
3. <加法表达式> ::= <项> {+ <项> | -<项>}
4. <项> ::= <因子> {* <因子> | /<因子>}
5. <因子> ::=ID|num | ‘(’ <表达式>‘)’

1 程序设计说明

1.1 编程语言选择及环境

编程语言：python
配置环境: 处理器: Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz(8 CPUs), ~2.0GHz
内存：8192MB RAM
开发平台：Visual Studio Code（VS code）
Visual Studio Code 是 Microsoft 是针对于编写现代 Web 和云应用的跨平台源代码编辑器，可在桌面上运行，并且可用于 Windows，macOS 和 Linux。它具有对 JavaScript, TypeScript 和 Node.js 的内置支持, 并具有丰富的其他语言（例如 C++, C＃，Java，Python，PHP，Go）。

语言版本：python3.9.2

1.2 输入方式

本程序采用了 Python 作为编程语言，采用了 Python 中的标准 Tk GUI 工具包的接口 tkinter，用户可以在程序运行后通过点击“读入程序”按钮选择相应的要读入的源代码文件，按钮下方的文本输入框会将源文件中的代码呈现出来。同时，用户也可直接在该按钮下方的文本框内直接输入要执行词法分析的代码。

1.3 词法分析具体流程

1.3.1 程序预处理

目的：编译预处理，剔除无用的字符和注释。

1.若为单行注释“//”,则去除注释后面的东西，直至遇到回车换行;
2.若为多行注释“/* 。。。*/”则去除该内容;
3.若出现无用字符，则过滤；否则加载；最终产生净化之后的源程序。

1.3.2 词法分析程序实现

1.读入经程序与处理后的源代码并依次对输入的字符进行扫描;
2.首先判断是否是空格、tab 和换行符：忽略空格和 tab，若遇到换行符，行数加 1;
3.通过符号是否以字母、下划线开头判断是否是标识符。读入后先与已有关键字一一匹配，若匹配成功，则为关键字，写入 table 表中；若匹配不成功，则不是关键字，按照变量标识符处理;
4.若不以字母、下划线为开头，则判断开头是否是数字，若是则表示遇到数字：区分小数和整数，同时判断是否出错;
5.最后来判断是否是运算符，运算符首先判断双目运算符，再行判断单目运算符。

1.3.3 出错处理

具体实现思路在于按照一定的判别顺序判断读入的信息是否是关键字、标识符、数字、符号；若这些均无法识别，则将读入的信息判别为错误。在此大基础上，在各自判断类别内特判一些个别错误（如开头读入数字，判断后续是否还出现字符等）。

1.4 语法分析具体流程

1.4.1 语法预处理

执行语法分析程序前，需要对给定的语法规则进行分析得到后续执行分析的 LR 分析表，具体操作如下：

1. 从文件中读入语法规则;
2. 构造基于该语法的 Clousure（项目集规范族）集合;

3. 基于上一步构造所有规范句型活前缀的 DFA;
4. 根据这个 DFA 来构造 Action 表与 Goto 表（统称为分析表）。

1.4.2 语法分析程序实现

由语法分析预处理程序得到 Action 表与 Goto 表后,开始执行语法分析程序,具体操作如下:

1. 读取由经过词法分析程序产生的 token 三元序列;
2. 对这些三元序列依据 LR1 分析表依次进行规约操作;
3. 若规约成功，则语法分析成功，输出分析栈以及语法分析树。

1.4.3 出错处理

考虑到语法出错的情况，根据语法规则我们对无法规约成功的 token 序列给出语法分析失败的错误提示，同时给出错误定位以及出错信息。

2 程序功能介绍

2.1 词法分析器功能介绍

2.1.1 词法规则

源程序应为类 C 语法程序，其词法规则如下:

① 支持关键字如下

```
keyword = ['int', 'double', 'char', 'float', 'break', 'continue',
                'do', 'while', 'if', 'else', 'for', 'void', 'return']
```

② 标识符应由字母（A-Z,a-z）、数字（0-9）、下划线“_”组成，并且首字符不能是数字，但可以是字母或者下划线_。

③ 字母: a |....| z | A |....| Z |

④ 数字: 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

数值可为无符号整形、科学计数法或复数

⑤ 赋值号可为+= -= *= /= %= ^= &= |= =

算符可为+ - * / % ^ & | = > < <= >= != ==

界符可为 ；

分隔符可为 ，

注释号可为 /* */ 或 //

括号可为 ( ) [ ] { }

单词符号	类型编码	单词符号	类型编码
int	0	<>	21
double	1	==	22
char	2	!=	23
float	3	+	24

break	4	-	25
continue	5	*	26
do	6	/	27
while	7	=	28
if	8	<	29
else	9	>	30
for	10	(	31
goto	11	)	32
for	12	,	33
void	13	;	34
return	14	.	35
标识符	15	[	36
整数	16	]	37
小数	17	:	38
<=	18	{	39
>=	19	}	40
:=	20	‘	41

词法语法规则:

```
start:$ identifier
start:$ integer
start:$ scientific
start:$ complex
start:$ limiter
start:$ operator
operator:+
operator:-
operator:*
operator:/
operator:%
operator:^
operator:&
operator:=
operator:>
operator:<
operator:> equal_tail
operator:< equal_tail
equal_tail:=
limiter:,
```

limiter:;
limiter:[
limiter:]
limiter:(
limiter:)
limiter:{
limiter:}
identifier:_
identifier:alphabet
identifier:_ identifier_tail
identifier:alphabet identifier_tail
identifier_tail:_
identifier_tail:digit
identifier_tail:alphabet
identifier_tail:_ identifier_tail
identifier_tail:digit identifier_tail
identifier_tail:alphabet identifier_tail
integer:digit
integer:digit integer_tail
integer_tail:digit
integer_tail:digit integer_tail
decimal:digit
decimal:digit decimal_tail
decimal_tail:digit
decimal_tail:digit decimal_tail
decimal_tail:e signed_index
signed_index:+ index
signed_index:- index
signed_index:digit
signed_index:digit index_tail
complex:digit
complex:digit complex_first_tail
complex_first_tail:digit complex_first_tail
complex_first_tail:+ complex_second_tail
complex_second_tail:i
complex_second_tail:digit complex_second_tail
index:digit
index:digit index_tail

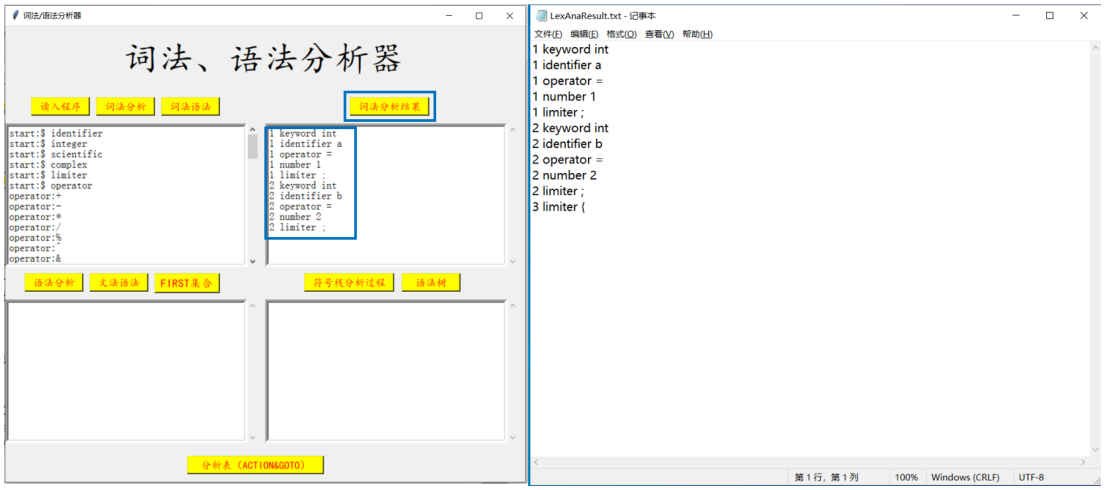
index_tail:digit
index_tail:digit index_tail
scientific:digit
scientific:digit scientific_tail
scientific_tail:digit
scientific_tail:digit scientific_tail
scientific_tail:. decimal
scientific_tail:e signed_index

2.1.2 程序预处理功能

预处理子程序任务:
1.剔除编辑用字符，如 space，tab，CR，LF。
2.剔除注释语句。
3.合并空白字符串为单个 space。
4.有时把独立文件中的多个源程序模块聚合在一起。
5.把宏定义的缩写形式转换为源语句。

2.1.3 词法分析输出内容

词法分析可以在界面文本框中输出词法规则，对输入的源程序执行词法分析程序并在文件中输出词法分析结果的 token 串，同时在界面文本框中输出 token 结果。



2.1.4 错误处理

当出现词法错误，程序无法分析的情况，词法分析程序会在程序终端提示词法分析失败，并输出出错定位及出错信息。


```
Lexical error at line 1, column 5 : 标识符或常量不合法
Lex analyze failed!
```

2.2 语法分析器功能介绍

2.2.1 语法规则

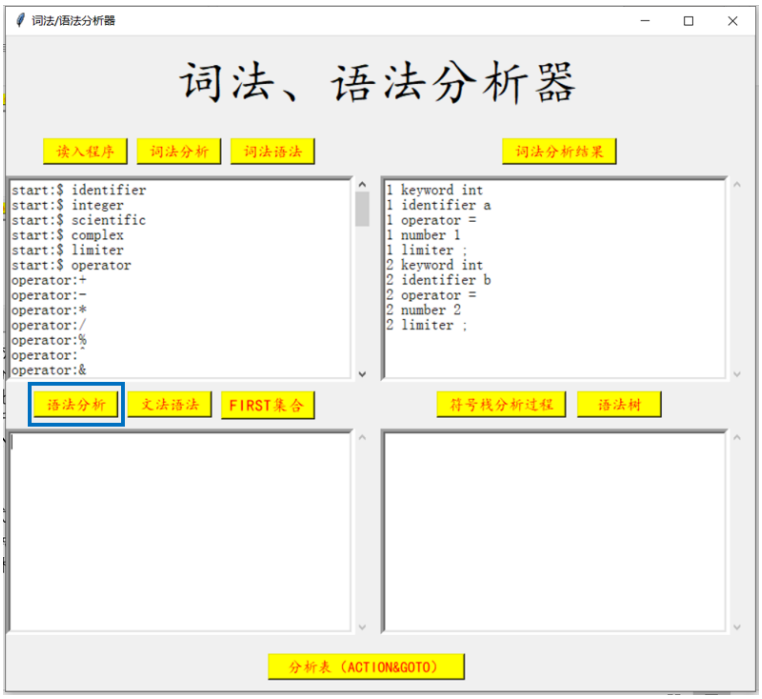
```
sstart:start
start:external_declaration start
start:$
external_declaration:declaration
external_declaration:function_definition
type_specifier:int
type_specifier:float
type_specifier:double
type_specifier:char
declaration:type_specifier declaration_parameter declaration_parameter_suffix ;
declaration_parameter:identifier declaration_parameter_assign
declaration_parameter_assign:= expression
declaration_parameter_assign:$
declaration_parameter_suffix:, declaration_parameter declaration_parameter_suffix
declaration_parameter_suffix:$
primary_expression:identifier
primary_expression:number
primary_expression:( expression )
```

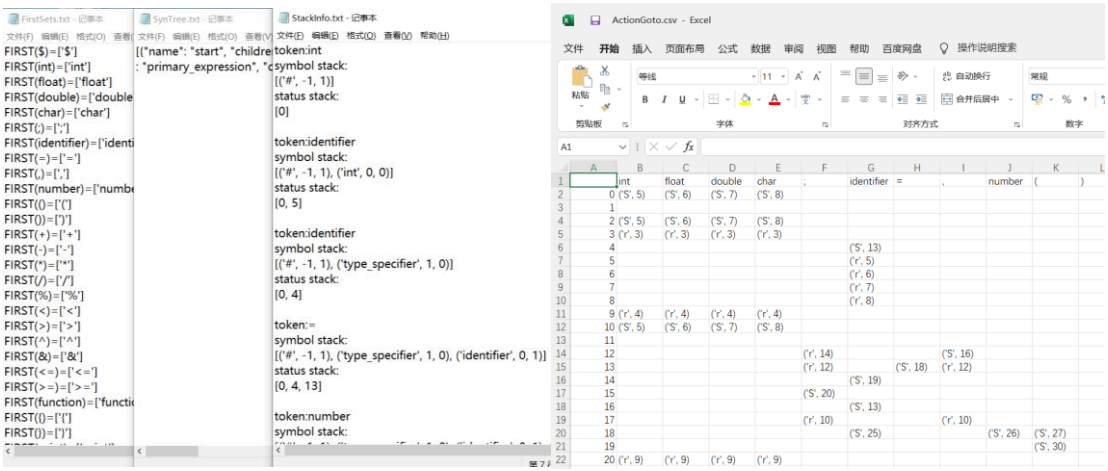
operator:+
operator:-
operator:*
operator:/
operator:%
operator:<
operator:>
operator:^
operator:&
operator:<=
operator:>=
assignment_operator:=
assignment_operator:+ =
assignment_operator:- =
assignment_operator:* =
assignment_operator:/ =
assignment_operator:% =
assignment_operator:^ =
assignment_operator:& =
assignment_expression:identifier assignment_operator expression
assignment_expression_list_suffix:, assignment_expression assignment_expression_list_suffix
assignment_expression_list_suffix:$
assignment_expression_list:assignment_expression assignment_expression_list_suffix
assignment_expression_list:$
expression:constant_expression
expression:function_expression
constant_expression:primary_expression arithmetic_expression
arithmetic_expression:operator
arithmetic_expression:primary_expression arithmetic_expression
arithmetic_expression:operator primary_expression arithmetic_expression
arithmetic_expression:$
function_expression:function identifier (expression_list)
expression_list_suffix:, expression expression_list_suffix
expression_list_suffix:$
expression_list:expression expression_list_suffix
expression_list:$

function_definition:function type_specifier identifier (function_parameter_list) compound_statement
function_parameter_list:function_parameter function_parameter_list_suffix
function_parameter_list:$
function_parameter_list_suffix:, function_parameter function_parameter_list_suffix
function_parameter_list_suffix:$
function_parameter:type_specifier identifier
compound_statement:{ statement_list }
statement_list:statement statement_list
statement_list:$
statement:expression_statement
statement:jump_statement
statement:selection_statement
statement:iteration_statement
statement:compound_statement
statement:declaration
expression_statement:assignment_expression_list ;
expression_statement:print (expression) ;
expression_statement:scanf (identifier) ;
jump_statement:continue ;
jump_statement:break ;
jump_statement:return expression ;
selection_statement:if (expression) statement else statement
iteration_statement:while (expression) statement
iteration_statement:for (declaration expression ; assignment_expression) statement

2.2.2 语法分析输出内容

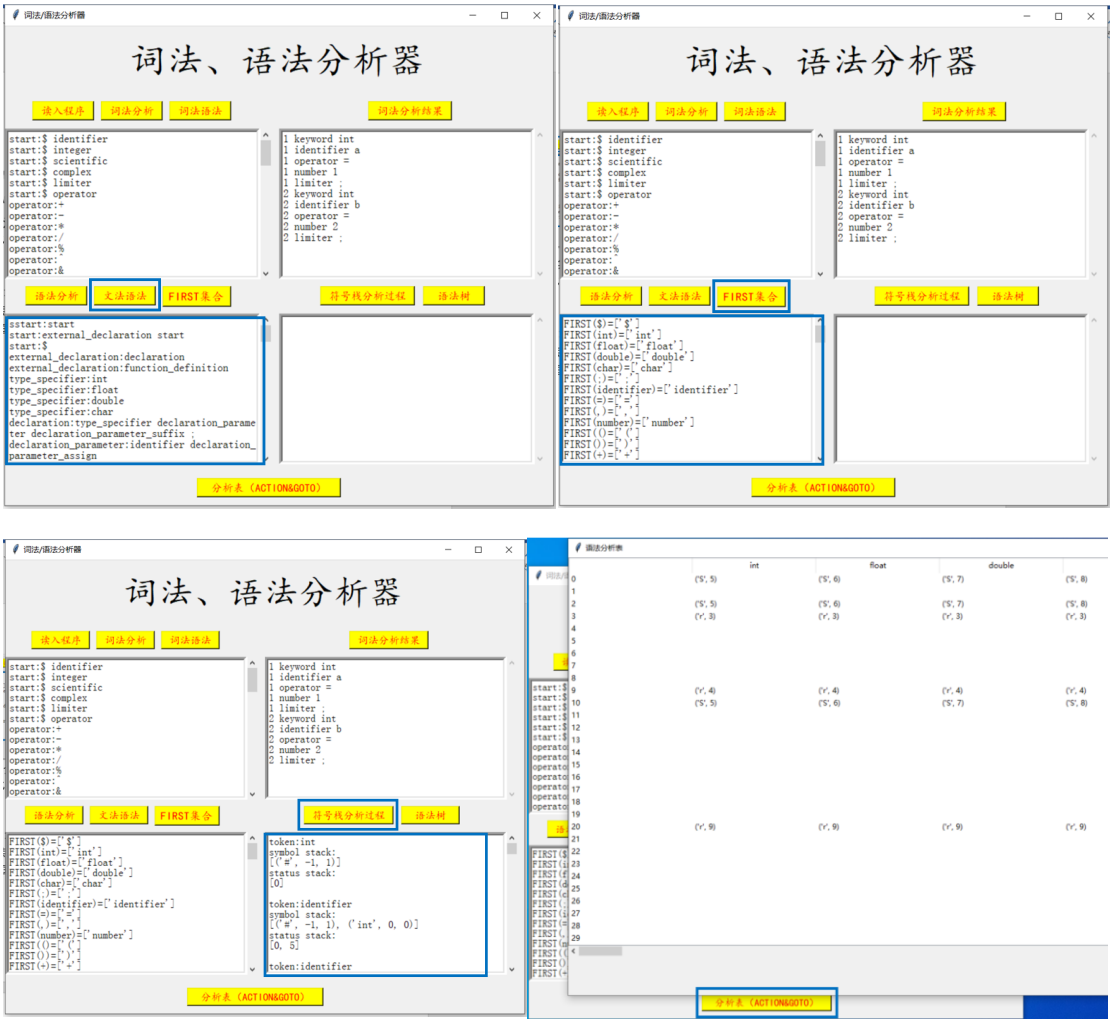
对经过词法分析得到的源程序的 token 表执行语法分析后，会在不同文件中分别输出语法分析的 FIRST 集、语法树内容、语法分析具体符号栈变化流程、语法分析的语法分析表（ACTION&GOTO 表）。同时会在程序终端输出语法分析结果信息，若语法分析成功则输出 Syntax Analyze Successfully!提示成功。





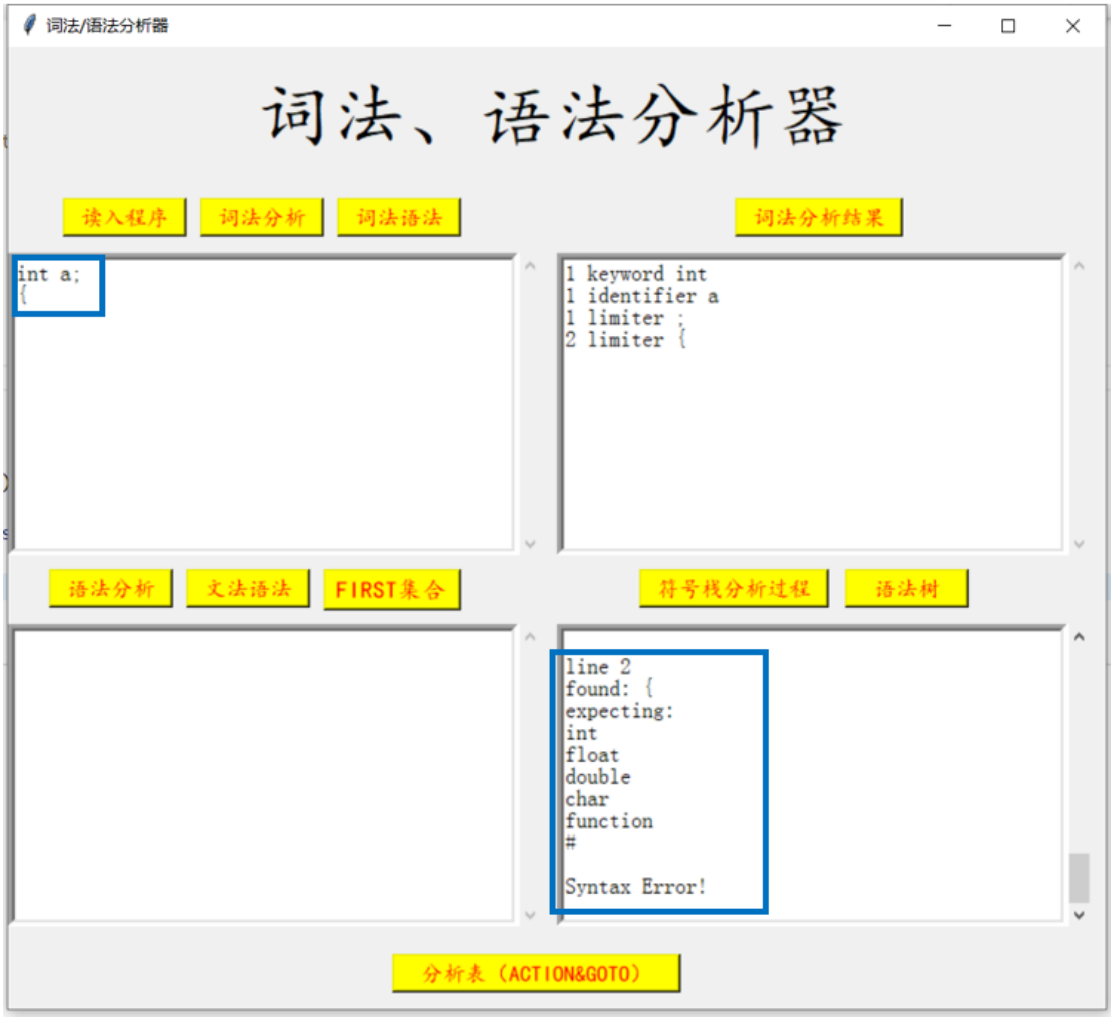

```
Syntax Analyze Successfully!
```

同时，可以在语法分析结束后，通过点击界面的不同按键显示不同信息，包括：文法语法、语法分析的 FIRST 集、符号栈具体分析过程、语法分析表（ACTION&GOTO 表）。



2.2.3 错误处理

当程序出现语法错误时，在执行完语法分析操作后，可以按下“符号栈分析过程”按键显示信息，在文本框底部查看语法错误定位及具体错误信息。同时在程序终端显示语法分析失败，同时提示出错定位与信息。




```
Lex analyze complete!

line 2
found: {
expecting:
int
float
double
char
function
#

Syntax Error!
```

3 详细设计说明

3.1 词法分析器主要函数说明

3.1.1 创建 NFA：LexAnaylze.createNFA

此函数在词法分析过程使用，用于创建词法文法对应的 NFA。

首先给出一个字典用于存储除了起始状态和终结状态之外的所有状态并给出一个函数用于查找状态。

创建 NFA 过程为：

1. 给出起始状态与终结状态
2. 对于词法文法中所有的产生式，将其左部设置为一个节点，若其右部仅有终结符，则将由左部产生的节点与终结状态相连接；否则构造右部的节点，并将左部节点与右部节点连接，连接箭头由终结符决定。
3. 将此字典(即得到的 NFA)存储在类中。

3.1.2 创建 DFA：LexAnaylze.createDFA

此函数在词法分析过程使用，用于创建词法文法对应的 DFA

给出一个字典用于存储所有状态并给出一个函数用于查找状态。

创建 DFA 过程为：

1. 对于从 NFA 的起始状态经过$的所有状态，将其设置为 DFA 的起始状态
2. 不断从某个起始状态开始搜索，根据不同的终结符寻找 NFA 的状态集合，将其作为 DFA 的一个状态，若此集合中存在 NFA 的终态，则将此状态设置为 DFA 的终态。然后若搜寻到的状态此前未被搜寻过则将其放入搜寻队列中，否则从队列的下一个状态开始搜寻
3. 不断重复 2 过程直至将所有起始状态搜寻完成。
4. 将此字典(即得到的 DFA)存储在类中。

3.1.3 利用 DFA 执行词法分析：LexAnalyze.runOnDFA

此函数的输入参数为输入串的某一行行，以及所需分析的标识符的起始位置，用于从输入串中标志位置。

输出是四元组，分别表示(标识符终结位置，标识符类型，标识符名称，分析信号)。

根据标识符的首个终结符判断大致分类，并在不同分类中反复使用 DFA 进行状态转移直至不可接受某个字符，并根据转移结果给出四元组。

3.2 语法分析器主要函数说明

3.2.1 构造 FIRST 集：SynAnalyze.getFirstSetForOne

此函数的输入参数为语法文法中的某个符号以及此前已经查询过的非终结符的集合，输出是这个符号的 FIRST 集。

计算规则为：

1. 如果当前符号的 FIRST 集已经算过(即存在于类中的 firstSet 集)，则直接返回该 FIRST 集；

2. 若此符号是文法中的终结符，则返回它自身；

3. 若是非终结符，则对所有以其为左部的产生式的右部的第一个符号递归求解并求并集，并且$在此 FIRST 集中当且仅当存在一个产生式，它的所有右部符号都能推导致$。

3.2.2 构造闭包：SynAnalyze.getClosure

此函数的输入参数有两个：第一个是一个五元组，表示：(产生式编号，产生式左侧，产生式右侧符号列表，圆点位置，向前搜索符集合)；第二个参数是一个由上述五元组组成的列表。

输出是由上述五元组组成的列表，表示从产生式推导出的闭包。

计算方式为:

1. 首先将第一个参数加入列表；

（若圆点之后跟着一个非终结符才可进行下述操作）

2. 计算以该非终结符为左侧的产生式的向前搜索符集合；

3. 对以圆点后非终结符为左侧的产生式进行遍历求闭包；

4. 将项目中产生式相同以及圆点位置相同的项目的搜索符做并集。

3.2.3 创建 LR 表：SynAnalyze.createLRTable

输入：指定的 LR(1)表的文档位置。

输出：若整个文法分析成功，则给出成功信号以及 LR(1)表文件；否则给出失败信号。

计算过程为：

1. 根据产生式 sstart:A,设立五元组(产生式编号，产生式左侧，产生式右侧符号列表，圆点位置，向前搜索符集合),根据此五元组求出初始项目集 I_0

2. 将初始项目集加入队列，并进行如下分析直至队列清空：

取出队列中第一个项目集，对其所有项目，若未操作过，则进行判断：若为空产生式或圆点已在产生式最右端，并且对于其他搜索符不存在移进或归约，则将其归约，否则发生冲突，判断此文法不是 LR(1)文法，函数退出;否则将圆点后移，并将同项目中其他所有圆点后跟着的符号与后移过程中跨越的符号相同的项目一并求闭包然后求并集，再将这个并集中产生式相同以及圆点位置相同的项目的搜索符做并集，再判断新求出的项目集是否与已存在项目集中某一项相同，若不相同，则新建项目并加入队列。此后判断原项目是否存在动作冲突，若存在则判断此文法不是 LR(1)文法，函数退出，否则在 LRTable 中写入对应操作。

3. 建立 action 和 goto 表并用 LRTable 填充。

3.2.4 利用 LR 表执行语法分析：SynAnalyze.runOnLRTable

输入：三元组列表和分析文档路径

输出：四元组(成功/失败信号，语法树节点，语法树相连关系，分析结果)

计算过程:

1. 初始化符号栈和状态栈。
2. 直到分析出错或分析动作为 acc 之前，不断进行如下操作:

根据三元组列表中第一个(为方便处理，我们将此列表倒序处理)和 LRtable 以及状态栈决定动作，若不存在动作则报错退出函数；否则为 acc 则退出结束，否则若为 s，则将语法树第 0 层移入另一个三元组(符号，语法树层数，同一层中该节点是第几个节点)，符号栈移入相同三元组，状态栈移入由动作决定的下一个状态;若为 r，则查看归约的产生式，若不为空产生式将语法树中对应节点与某个节点相连，并将此节点置于已在树中节点的最高层的上一层，状态栈和符号栈移除此前节点，移入新增节点，若为空产生式则在第 0 层新移入$节点，上一层，符号栈和状态栈移入新节点。

3. 根据 2 的结果给出对应信息。

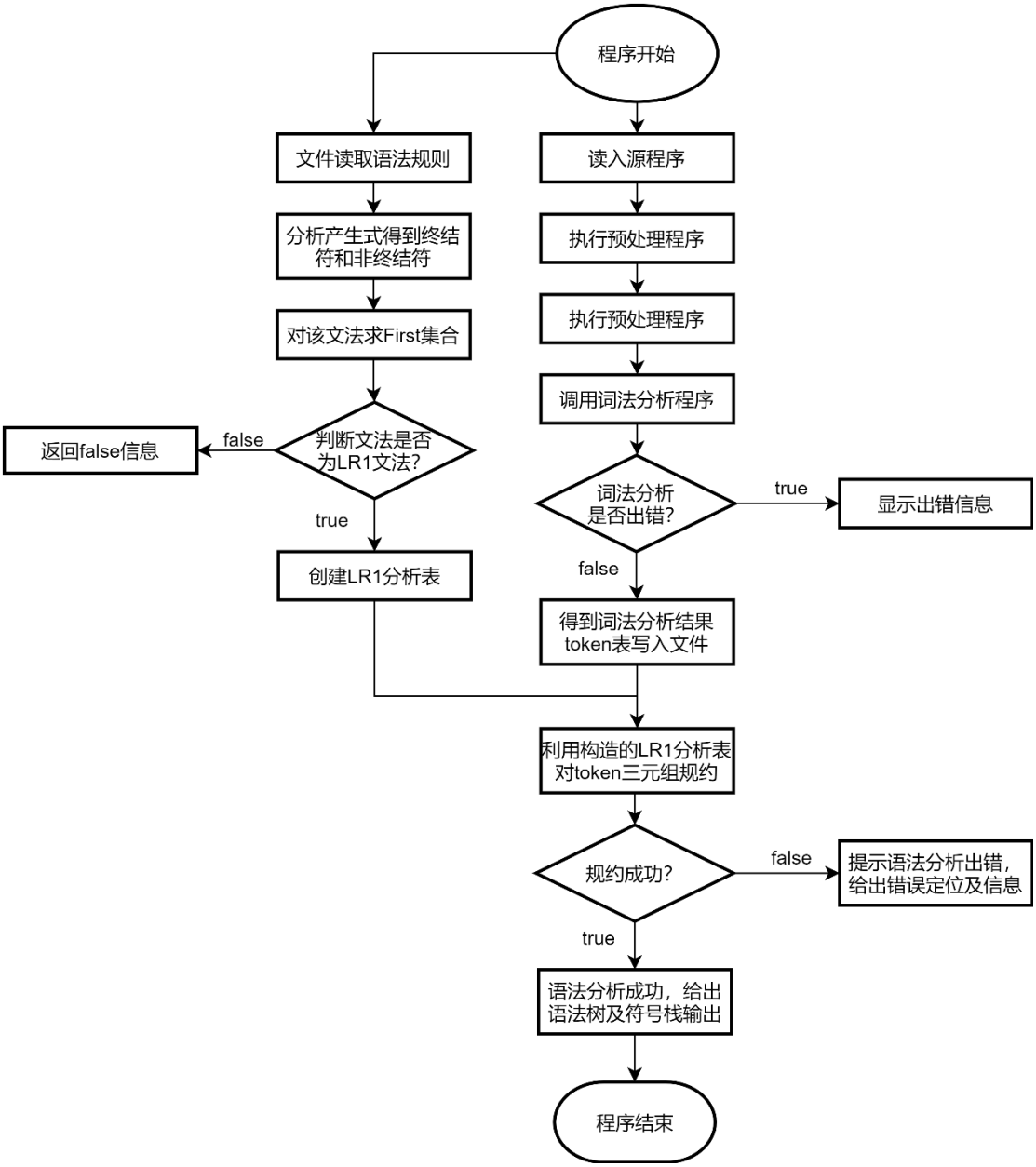
3.2.5 构造语法树：SynAnalyze.get_tree

输入：语法树节点，语法树相连关系，语法树路径。

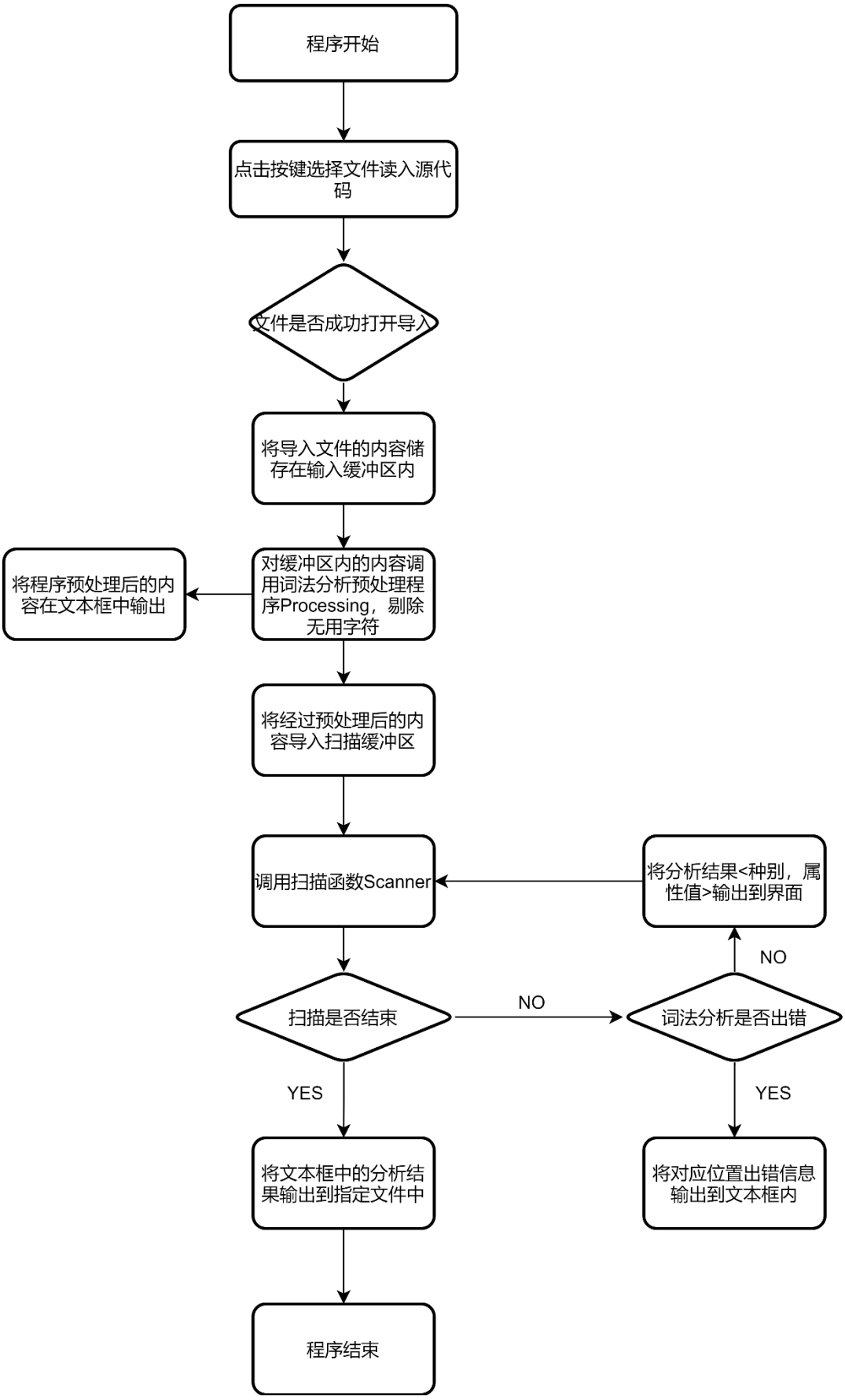
输出：含有语法树节点信息的字典。

4 算法分析框图

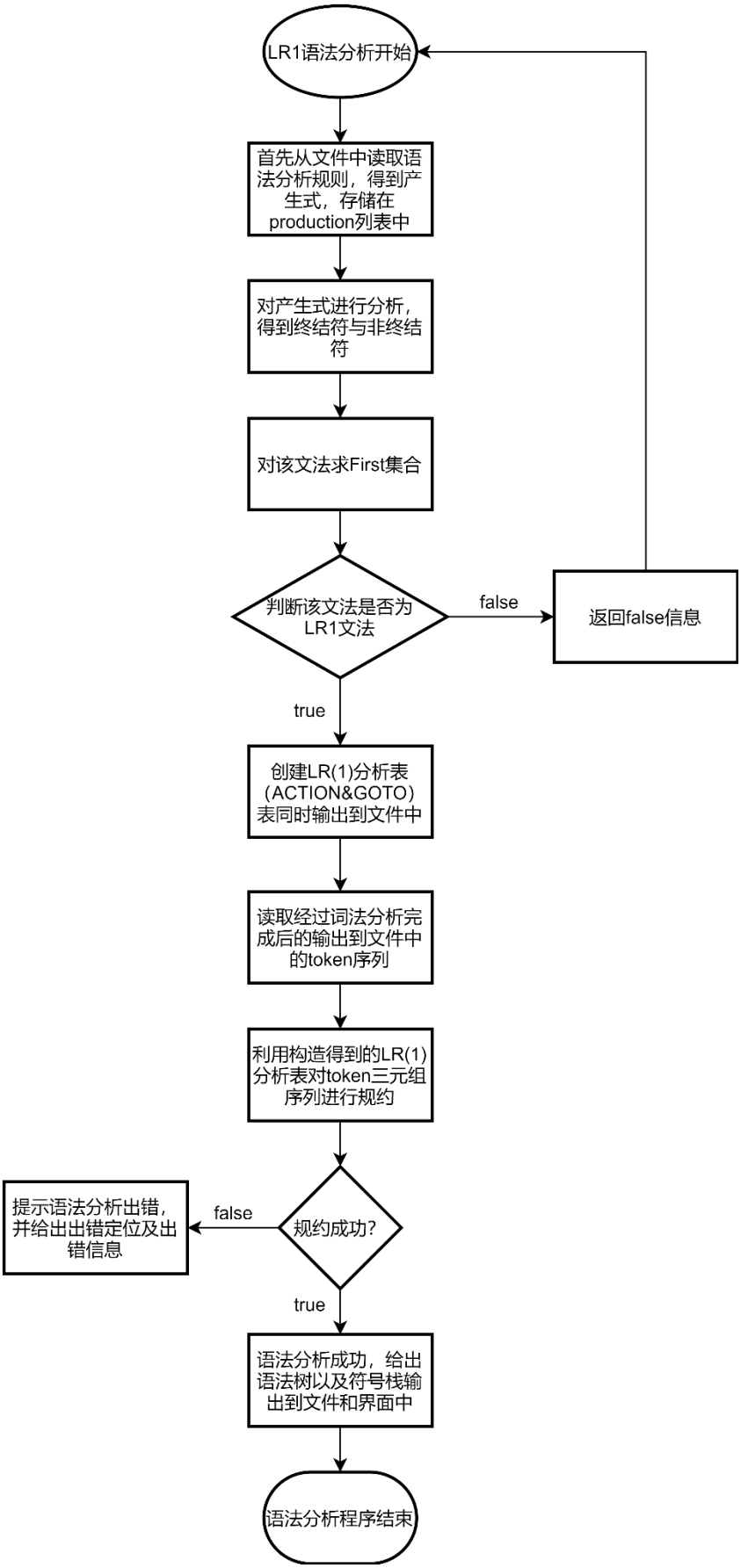
4.1 总程序算法框图



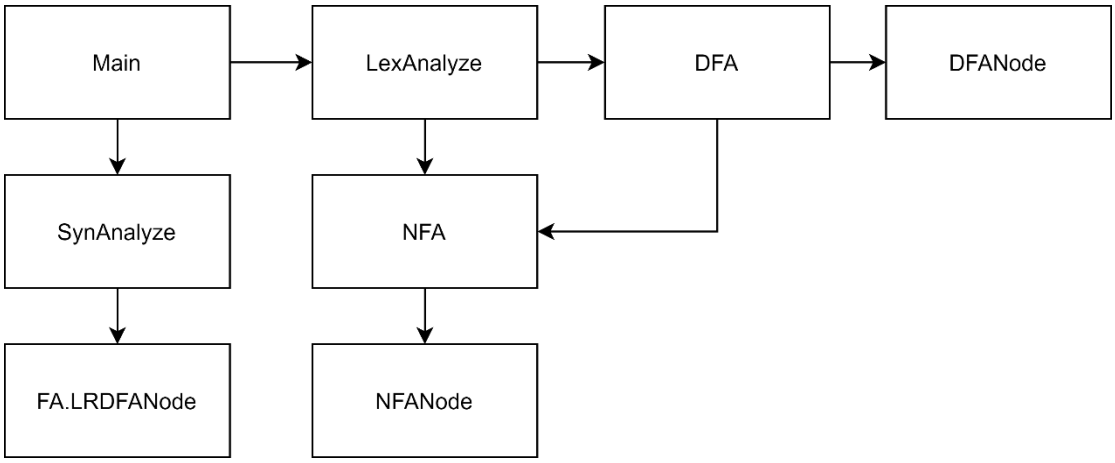
4.2 词法分析算法框图


程序开始
点击按键选择文件读入源代码
文件是否成功打开导入
将导入文件的内容储存在输入缓冲区内
对缓冲区内的内容调用词法分析预处理程序Processing，剔除无用字符
将程序预处理后的内容在文本框中输出
将经过预处理后的内容导入扫描缓冲区
调用扫描函数Scanner
将分析结果<种别，属性值>输出到界面
NO
扫描是否结束
NO
词法分析是否出错
YES
YES
将文本框中的分析结果输出到指定文件中
将对应位置出错信息输出到文本框内
程序结束

4.3 语法分析算法框图


LR1语法分析开始
首先从文件中读取语法分析规则，得到产生式，存储在production列表中
对产生式进行分析，得到终结符与非终结符
对该文法求First集合
判断该文法是否为LR1文法
false
返回false信息
true
创建LR(1)分析表(ACTION&GOTO)表同时输出到文件中
读取经过词法分析完成后的输出到文件中的token序列
利用构造得到的LR(1)分析表对token三元组序列进行规约
规约成功?
false
提示语法分析出错，并给出出错定位及出错信息
true
语法分析成功，给出语法树以及符号栈输出到文件和界面中
语法分析程序结束

4.4 程序模块调用关系

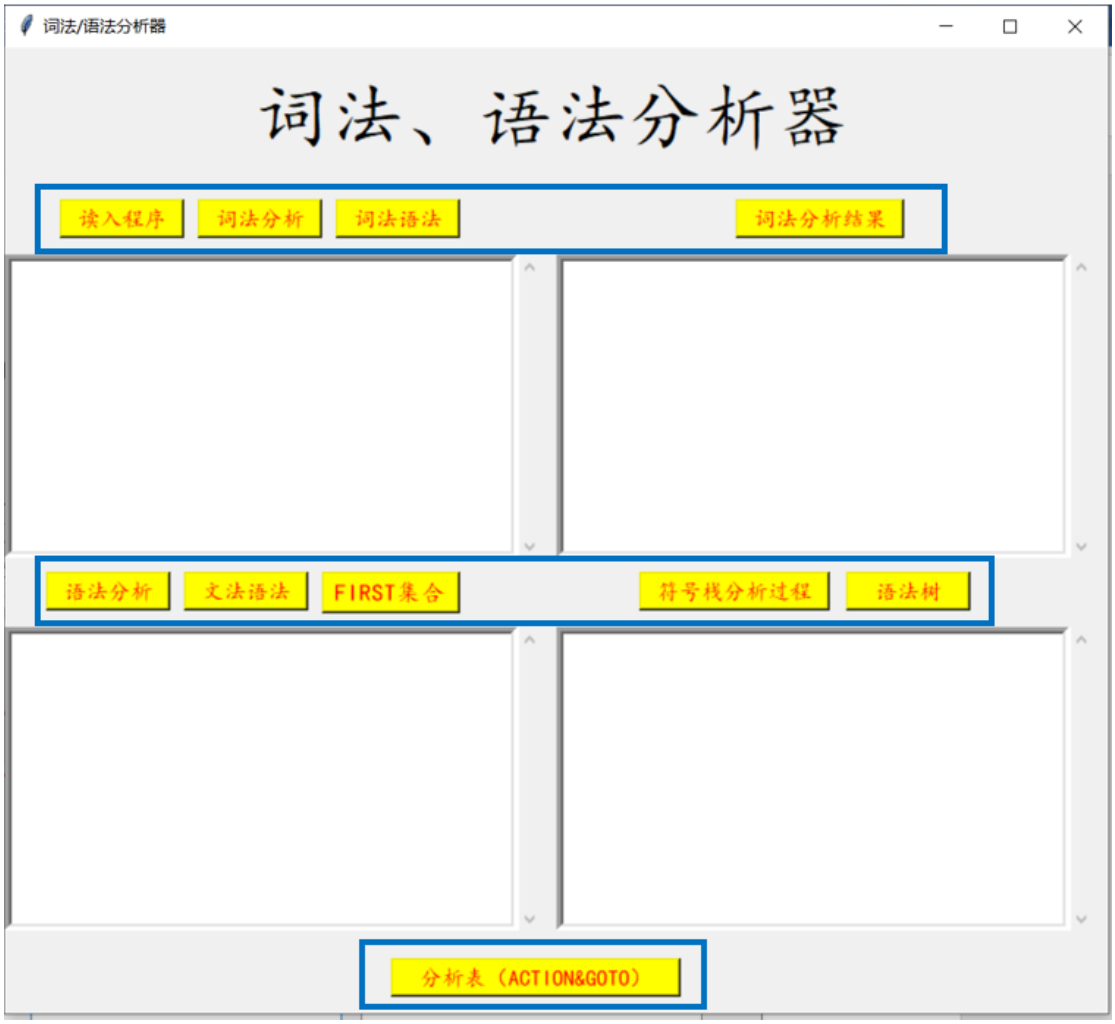


5 程序使用说明

5.1 UI 界面按键功能说明

1. 总界面:

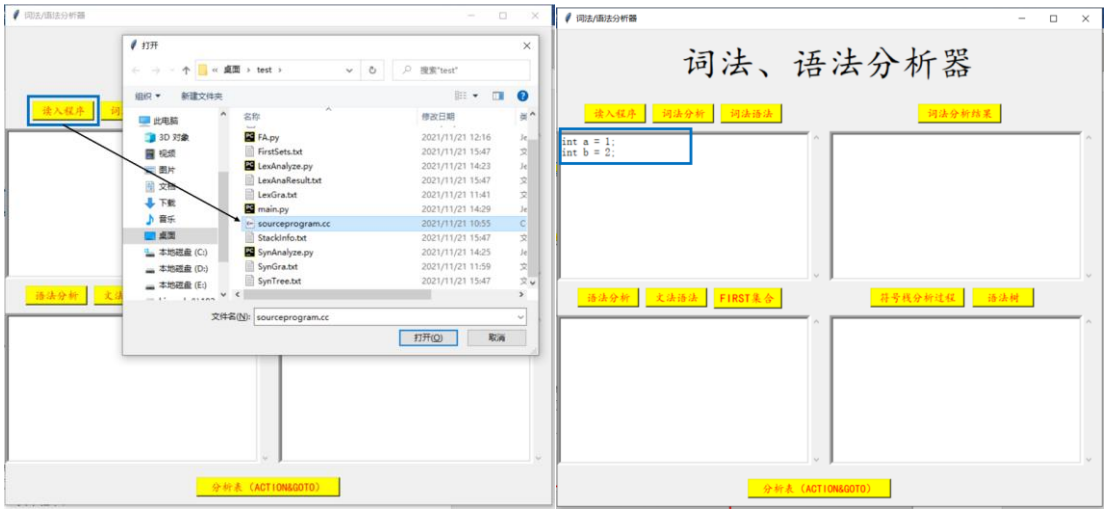
该分析器词法分析部分按键包括:“读入程序”、“词法分析”、“词法语法”、“词法分析结果”;语法分析部分按键包括:“语法分析”、“文法语法”、“FIRST 集合”、“符号栈分析过程”、“语法树”、“分析表（ACTION&GOTO）”。



2. 词法分析按键:

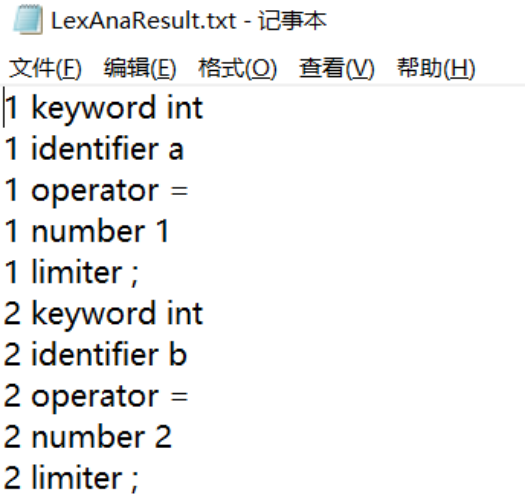
① “读入程序”

按下此按键可以进行文件路径选择，选择需要执行词法分析的文件，并将此文件中的内容显示在按键下方的文本框中。



② “词法分析”

按下此按键执行词法分析程序，对文件中的源程序信息分析生成 token 表输出到文件中，同时在程序终端输出分析结果信息（分析成功、分析错误及出错信息）。



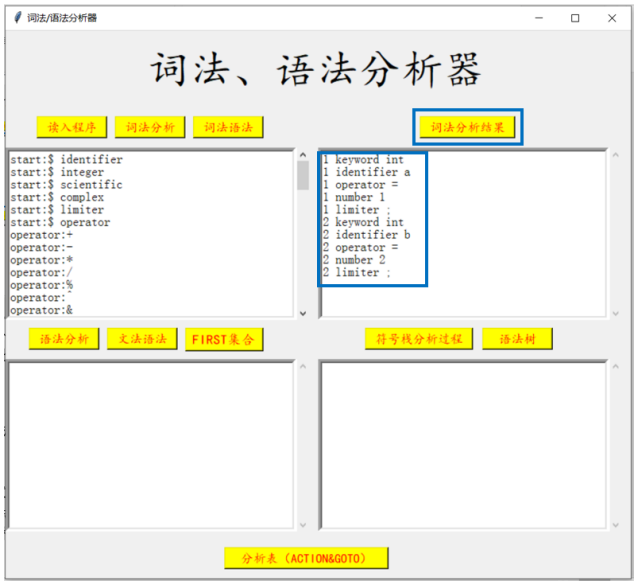

```
Lex analyze complete!
Lexical error at line 3, column 1 : 标识符或常量不合法
Lex analyze failed!
Lex analyze complete!
```

③ “词法语法”

按下此按键显示词法分析语法规则。

④ “词法分析结果”

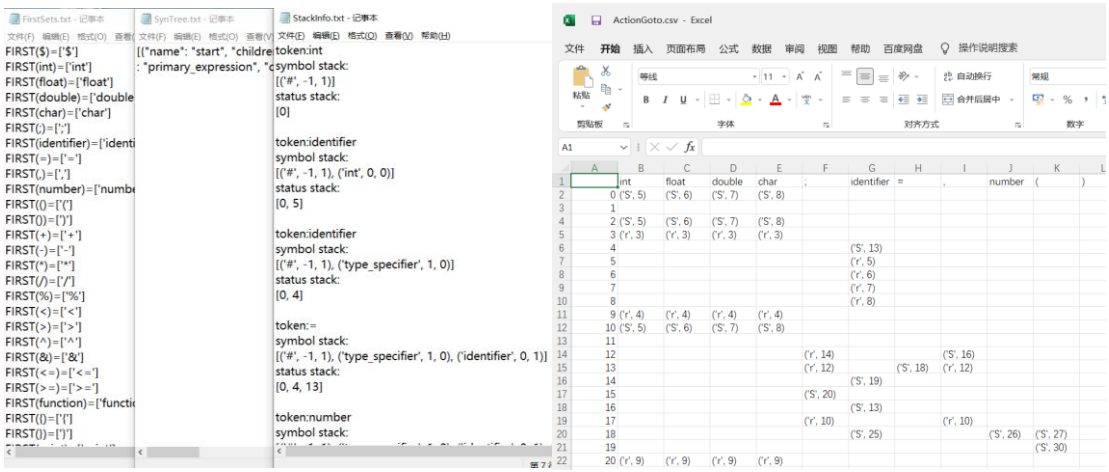
按下此按键将输出到文件中的词法分析结果（token 表）输出到下方文本框中。



2. 语法分析按键:

①“语法分析”

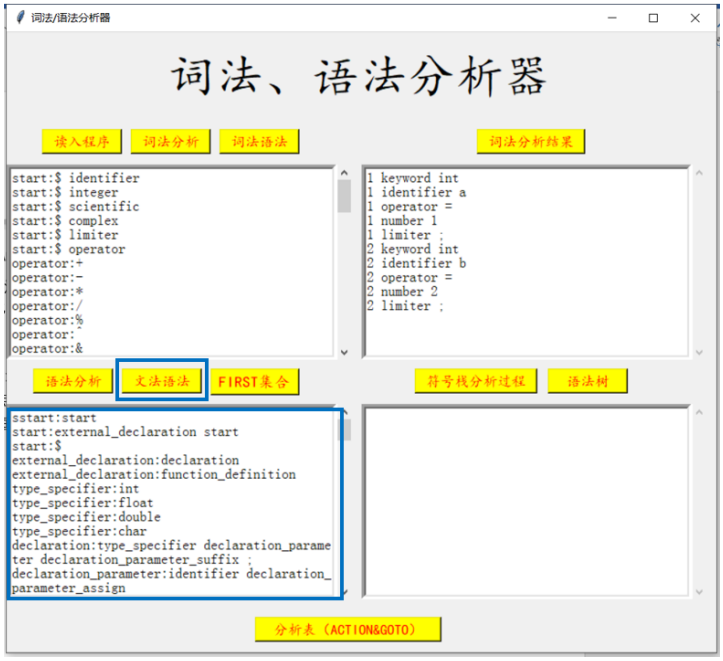
按下此按键执行语法分析程序，对词法分析生成的 token 表按照指定语法规则进行分析，得到 FIRST 集合、符号栈分析过程、语法树输出到文件中，同时生成 ACTION&GOTO 分析表输出到表格文件中。同时在终端输出语法分析结果信息：分析正确、分析错误+出错处信息。



Syntax Analyze Successfully!

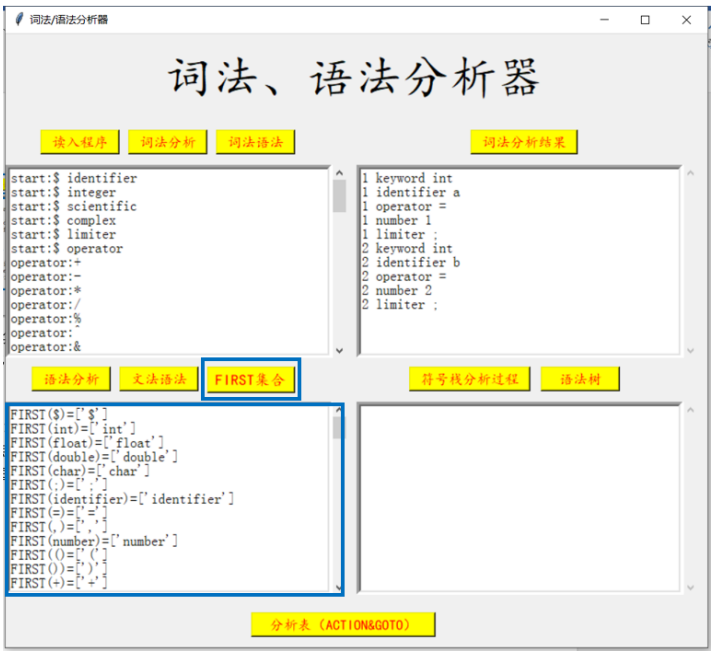
② “文法语法”

按下此按键显示语法分析文法规则。



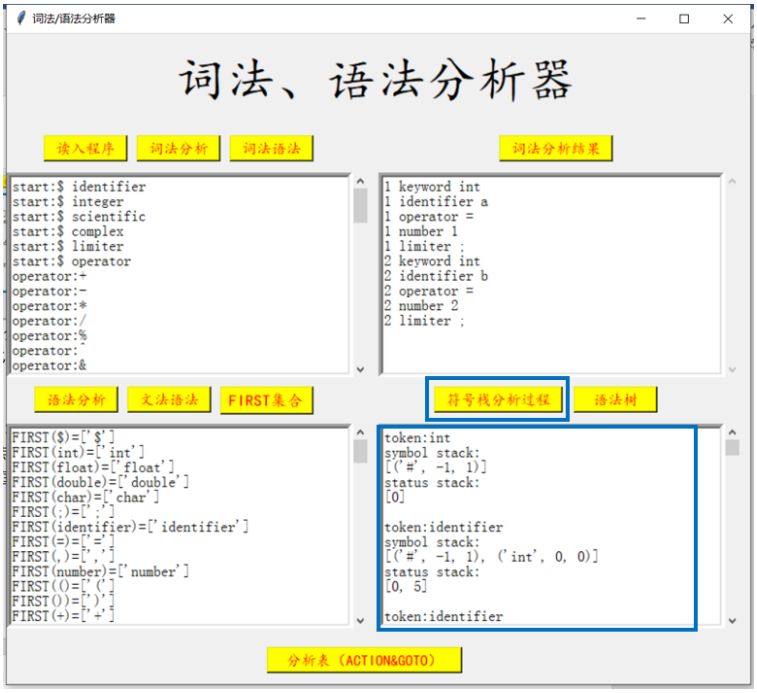
② “FIRST 集合”

按下此按键显示对 token 表按指定语法分析文法规则得到的 FIRST 集合。



② “符号栈分析过程”

按下此按键显示执行语法分析的符号栈分析过程。

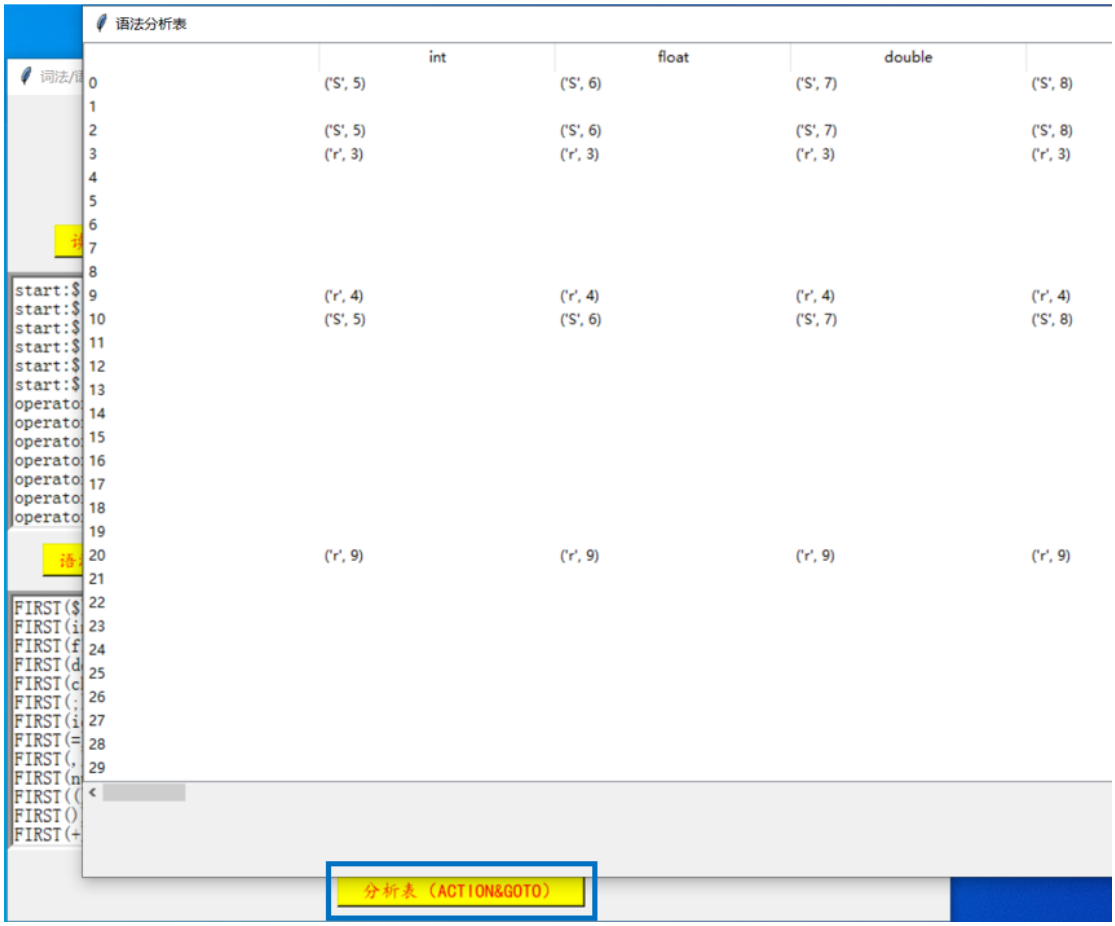


③ “语法树”

按下此按键显示执行语法分析所创建的语法树。

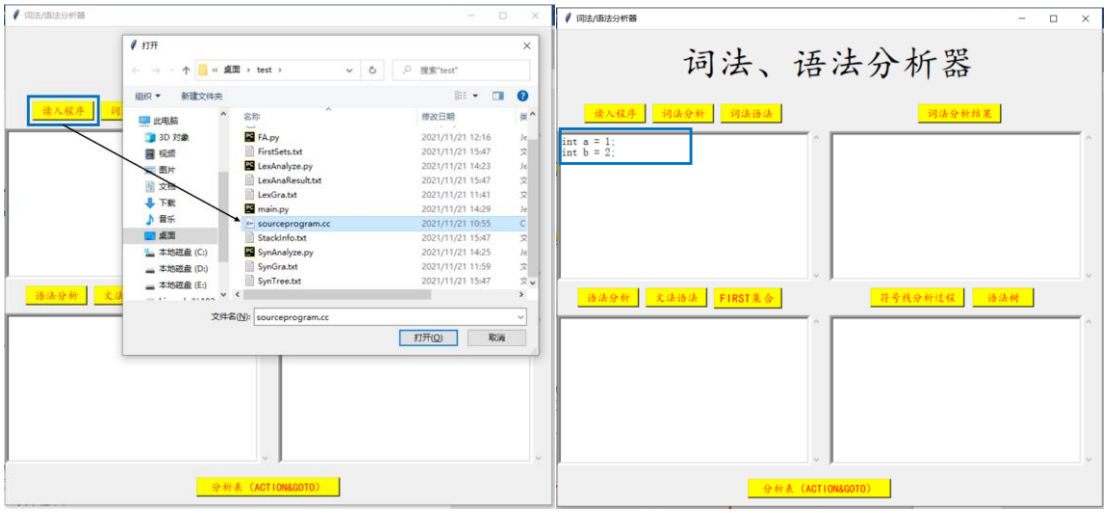
④ “分析表（ACTION&GOTO）”

按下此按键将语法分析的 ACTION&GOTO 分析表在界面可视化，展示具体表的内容。



5.2 程序输入方式

① 通过点击“读入程序”按键选择文件路径导入文件中的源代码，程序会将文件中需要执行词法分析的代码显示在文本框中。

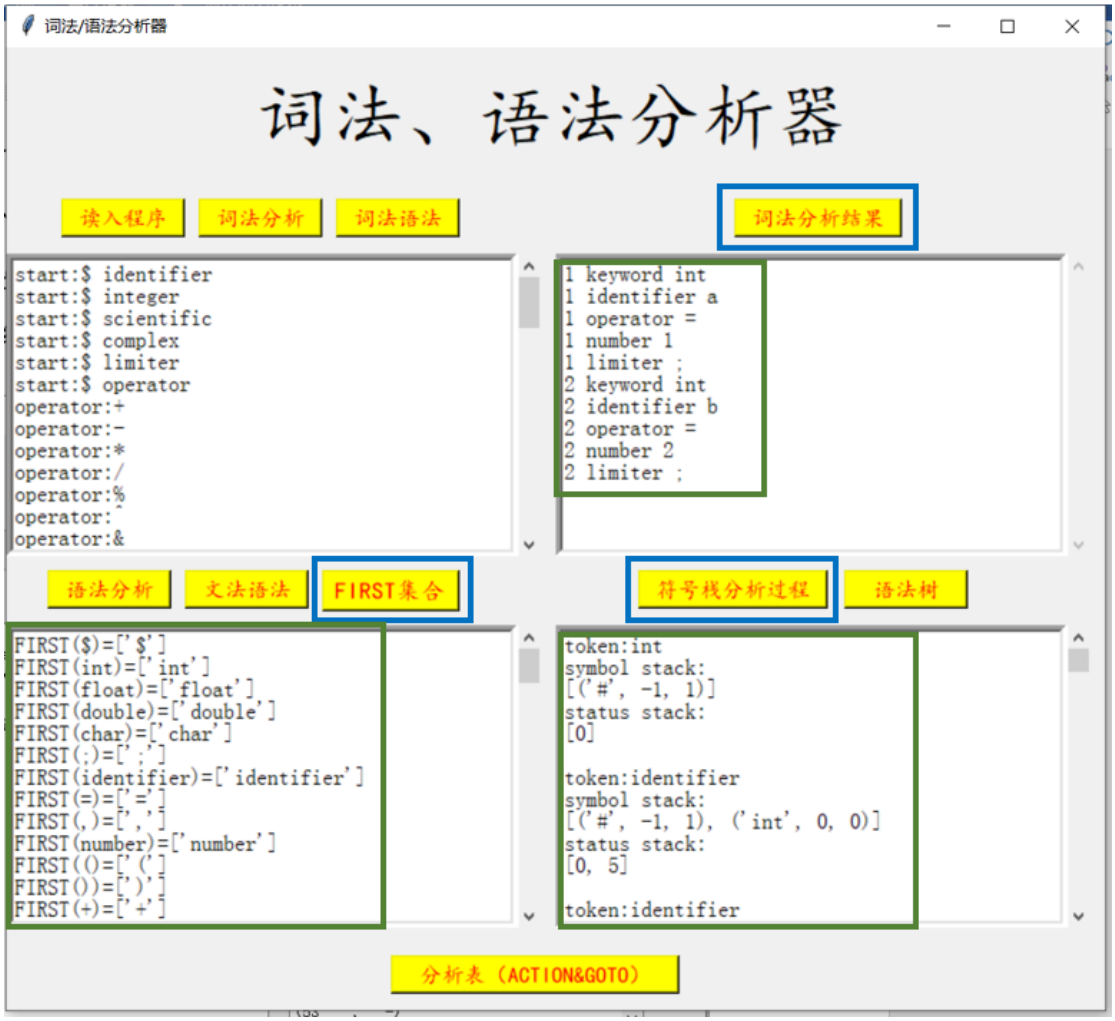


② 还可通过直接在文本框中输入要进行词法分析的源代码进行分析。

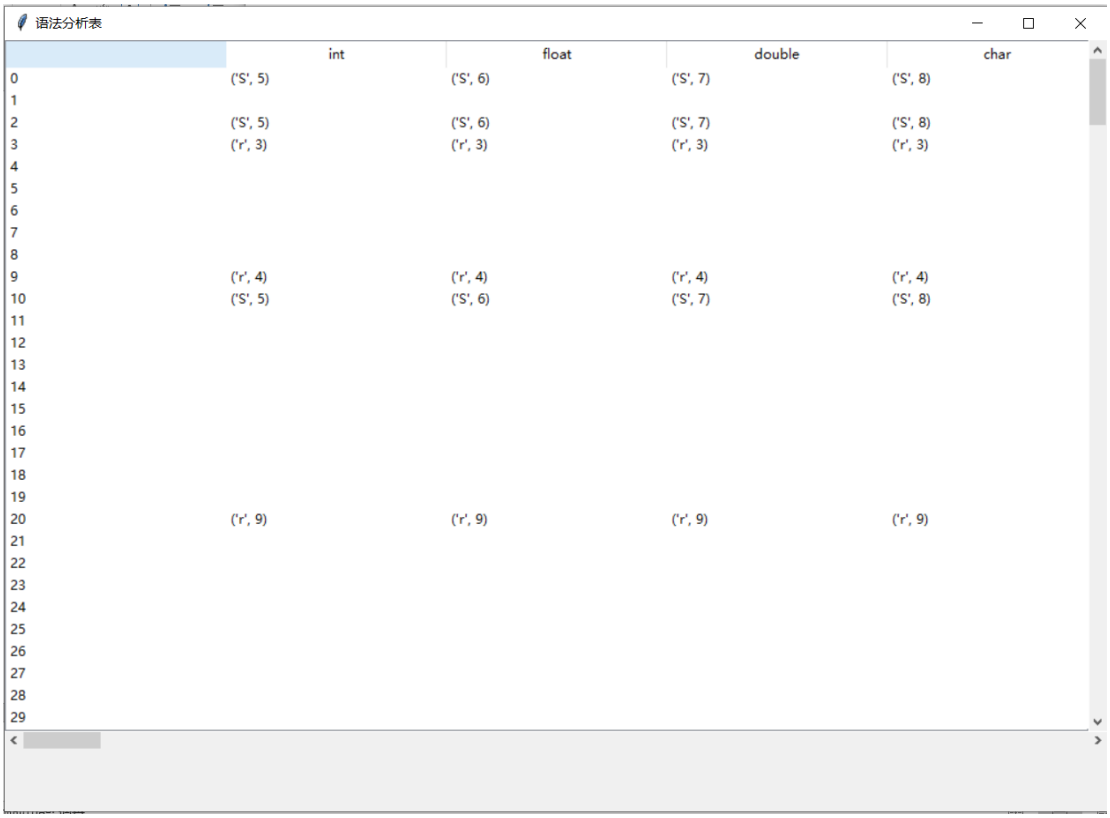
5.3 结果输出方式

① 程序将经过词法/语法分析所得的结果在界面中的文本框中显示出来。

如图在不同的文本框中分别输出词法分析结果 token 表、语法分析 FIRST 集合、语法分析符号栈分析过程以及语法分析语法树。



② 程序界面显示语法分析 ACTION&GOTO 分析表



③ 在程序终端输出词法/语法分析结果信息，若分析成功输出成功信息，若分析失败则定位出错位置并给出出错原因提示。

词法分析成功：

```
Lex analyze complete!
```

词法分析失败，并给出出错定位及出错信息：

```
Lexical error at line 3, column 1 : 标识符或常量不合法
Lex analyze failed!
```

语法分析成功：

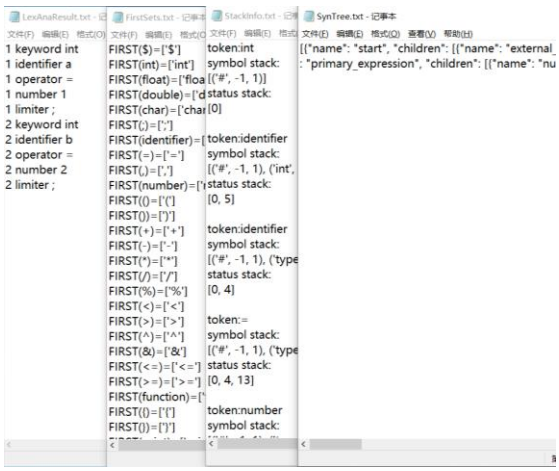
```
Syntax Analyze Successfully!
```

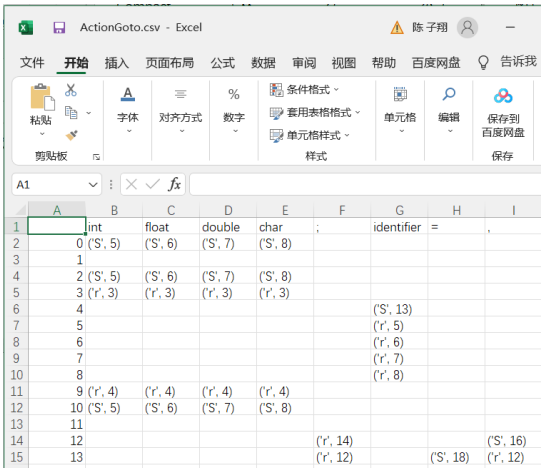
语法分析失败，并给出出错定位及出错信息：

```
line 3
found: {
expecting:
int
float
double
char
function
#

Syntax Error!
```

④ 同时程序将词法分析的结果（程序 token 表）、语法分析的结果（FIRST 集合、符号栈分析过程、语法树、语法分析表）分别输出到项目目录下的 LexAnaResult.txt、FirstSets.txt、StackInfo.txt、SynTree.txt、ActionGoto.csv 中。





6 执行实例

6.1 用户友好界面

本次词法分析结果主体采用 UI 界面输出。运行程序后，用户可以通过点击“读入程序”按钮选择要读入源文件的路径并将程序内容输出到左侧文本框中，直观显示。读入程序内容结束后，点击“词法分析”按钮后，表示词法分析结束，具体结束信息在控制台呈现；点击“词法语法”可以查看词法规则；点击“词法分析结果”在下方文本框中显示词法分析结果的 token 序列。同理，点击语法相关按键执行语法分析操作，显示语法分析过程的信息（FIRST 集合、符号栈内容、语法树）。点击“分析表”后将 LR1 分析表在界面可视化呈现。总体交互体验不错。

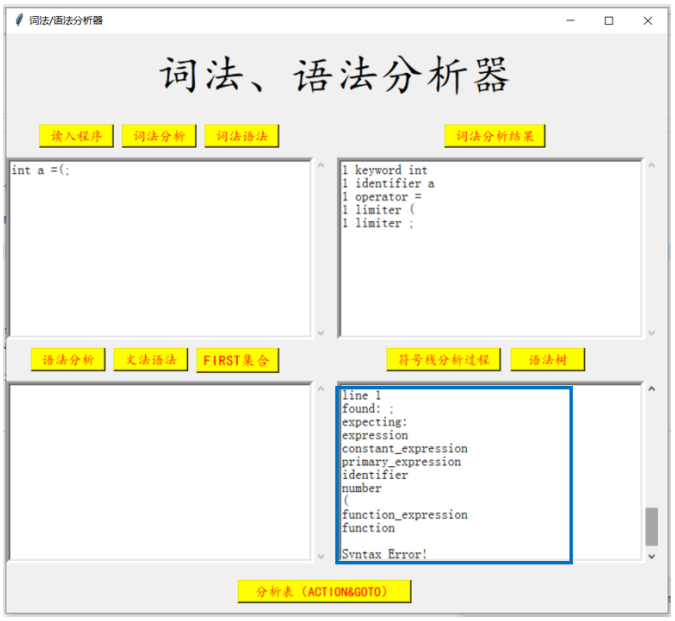
6.2 运行结果截图

① 词法分析错误示例：


```
PS C:\Users\C.X\Desktop\test>  c:; cd 'c:\Users\C.X\Desktop\test'; & 'E:\python3.9.2\python.exe'
69775\pythonFiles\lib\python\debugpy\launcher' '50277' '--' 'c:\Users\C.X\Desktop\test\main.py'
Lexical error at line 2, column 5 : 标识符或常量不合法
Lex analyze failed!
```

源程序中出现了错误标识 2a，在终端提示“Lex analyze failed!”提示词法分析出错，同时给出错误定位“Lexical error at line 2, column 5”，以及错误信息：“标识符或常量不合法”。

② 词法分析成功、语法分析失败示例：



```
69775\pythonFiles\lib\python\debugpy\launcher' '65198' '--' 'c:\Users\C.X\Desktop\test\main.py'
Lex analyze complete!

line 1
found: ;
expecting:
expression
constant_expression
primary_expression
identifier
number
(
function_expression
function

Syntax Error!
```

源程序末尾缺少右括号“)”，其中内容均符合词法分析过程，因此词法分析成功，程序终端输出提示“Lex analyze complete!”表示词法分析成功。

而程序末尾缺少右括号，不符合语法分析规则，无法通过语法分析程序，界面显示符号栈分析过程时输出出错位置：“line 1”；以及后续出错信息，输出“Syntax Error！”表示语法分析失败。

同时在终端也输出相应语法分析失败的信息。

③ 词法分析成功、语法分析成功示例 1（PPT 类 C 源程序示例）：

```
PS C:\Users\C.X\Desktop\test> c:; cd 'c:\Users\C.X\Desktop\test'; & 'E:
69775\pythonFiles\lib\python\debugpy\launcher' '50894' '--' 'c:\Users\C.
Lex analyze complete!

Syntax Analyze Successfully!
```

使用 PPT 中的类 C 源代码，均能通过词法分析以及语法分析，并提示：词法分析成功：“Lex analyze complete!”，语法分析成功：“Syntax Analyze Successfully!”。

④ 词法分析成功、语法分析成功示例 2：


```
69775\pythonFiles\lib\python\debugpy\launcher' '50936' '--' 'c:\Use
Lex analyze complete!

Syntax Analyze Successfully!
```

语法分析成功、词法分析成功!

7 程序源代码

7.1 main.py

```
from SynAnalyze import SynAnalyze
from LexAnalyze import LexAnalyze
import tkinter as tk
from tkinter.filedialog import askdirectory
from tkinter import StringVar
from tkinter import *
from tkinter import scrolledtext
from pandas import read_csv
from tkinter import ttk
from tkinter import messagebox
import csv
from tkinter import filedialog

#向分析表中插入数据
def insert(table, result):
    # 插入数据
    for index, data in enumerate(result):
        if index != 0:
            table.insert('', END, values=data)  # 添加数据到末尾

def selectPath():
    var=StringVar()
    scr = scrolledtext.ScrolledText(window, width=30, height=10, font=("
隶书",18), bd =5)
    scr.place(x=0,y=50)

    path_ = askdirectory()
    print(path_)
    path_=path_.replace('\\','/')+'/LexAnaResult.txt'
    f = open(path_,'r',encoding = 'utf-8')
    for line in f:
        scr.insert('end',line)
    print(f.read())

    with open('ActionGoto.csv', 'r') as f:
        reader = csv.reader(f)
        result = list(reader)
        print(result[0])
```

```
    columns = result[0]

    screenwidth = window.winfo_screenwidth()  # 屏幕宽度
    screenheight = window.winfo_screenheight()  # 屏幕高度
    width = 1000
    height = 500
    x = int((screenwidth - width) / 2)
    y = int((screenheight - height) / 2)
    window.geometry('{}x{}+{}+{}'.format(width, height, x, y))  # 大小以
及位置

    tabel_frame = tk.Frame(window)
    tabel_frame.pack()

    xscroll = Scrollbar(tabel_frame, orient=HORIZONTAL)
    yscroll = Scrollbar(tabel_frame, orient=VERTICAL)

    #columns = ['学号', '姓名', '性别', '出生年月', '籍贯', '班级', '班级',
'班级', '班级', '班级', '班级']
    table = ttk.Treeview(
            master=tabel_frame,  # 父容器
            height=10,  # 表格显示的行数,height 行
            columns=columns,  # 显示的列
            show='headings',  # 隐藏首列
            xscrollcommand=xscroll.set,  # x 轴滚动条
            yscrollcommand=yscroll.set,  # y 轴滚动条
            )
    for column in columns:
        table.heading(column=column, text=column, anchor=CENTER,
                      command=lambda name=column:
                      messagebox.showinfo('', '{}描述信息
~~~'.format(name)))  # 定义表头
    table.column(column=column, width=100, minwidth=100,
anchor=CENTER, )  # 定义列
    xscroll.config(command=table.xview)
    xscroll.pack(side=BOTTOM, fill=X)
    yscroll.config(command=table.yview)
    yscroll.pack(side=RIGHT, fill=Y)
    table.pack(fill=BOTH, expand=True)

    insert(table, result)
```

```
    btn_frame = Frame()
    btn_frame.pack()
    Button(btn_frame, text='添加', bg='yellow', width=20,
command=insert).pack()

# 界面显示源程序
def SourceProGet():
    LexScr1.delete(1.0, END)
    source_path = filedialog.askopenfilename()
    sourcefile = open(source_path,'r',encoding = 'utf-8')
    for line in sourcefile:
        LexScr1.insert('end',line)

# 执行词法分析程序
def LexAnalyzeFunc():
    LexGrammar_path = './LexGra.txt'  # 词法规则文件相对路径
    TokenTable_path = './LexAnaResult.txt'  # 存储 TOKEN 表的相对路径

    lex_ana = LexAnalyze()
    lex_ana.readLexGrammar(LexGrammar_path)
    lex_ana.createNFA()
    lex_ana.createDFA()

    codelist = lex_ana.Preprocessing(LexScr1.get(1.0,'end'))
    lex_ana.analyze(codelist, TokenTable_path)

# 界面显示词法分析规则
def LexRuleDisplay():
    LexScr1.delete(1.0, END)
    LexGrammar_path = './LexGra.txt'  # 词法规则文件相对路径
    sourcefile = open(LexGrammar_path,'r',encoding = 'utf-8')
    for line in sourcefile:
        LexScr1.insert('end',line)

# 界面显示词法分析结果
def LexResult():
    LexScr2.delete(1.0, END)
    TokenTable_path = './LexAnaResult.txt'  # 存储 TOKEN 表的相对路径
    sourcefile = open(TokenTable_path,'r',encoding = 'utf-8')
    for line in sourcefile:
        LexScr2.insert('end',line)
```

```
# 执行语法分析程序
def SynAnalyzeFunc():
    SynGrammar_path = './SynGra.txt'  # 语法规则文件相对路径
    TokenTable_path = './LexAnaResult.txt'  # 存储 TOKEN 表的相对路径
    LRTable_path = './ActionGoto.csv'  # 存储 LR 表的相对路径
    FirstSets_path = './FirstSets.txt'

    syn_ana = SynAnalyze()
    syn_ana.readSynGrammar(SynGrammar_path)
    syn_ana.getTerminatorsAndNon()
    syn_ana.getFirstSets()
    sourcefile = open(FirstSets_path,'w',encoding = 'utf-8')
    result = ''
    for key,value in syn_ana.firstSet.items():
        result = result + 'FIRST(' + key + ')' + '=' + str(value) + '\n'
    sourcefile.write(result)
    syn_ana.createLRTable(LRTable_path)
    sourcefile.close()
    syn_ana.analyze(TokenTable_path,SynAnalyzeProcess_path="./StackInfo.
txt")

# 界面显示语法分析规则
def SynRuleDisplay():
    SynScr1.delete(1.0, END)
    SynGrammar_path = './SynGra.txt'  # 词法规则文件相对路径
    sourcefile = open(SynGrammar_path,'r',encoding = 'utf-8')
    for line in sourcefile:
        SynScr1.insert('end',line)

# 界面显示语法分析 FIRST 集合
def SynFIRSTDisplay():
    SynScr1.delete(1.0, END)
    FirstSets_path = './FirstSets.txt'
    sourcefile = open(FirstSets_path,'r',encoding = 'utf-8')
    for line in sourcefile:
        SynScr1.insert('end',line)

# 界面显示语法分析界 csv 分析表
def csvdisplay():
    windowson = tk.Tk()
    windowson.title('语法分析表')
    windowson.geometry('1000x700')
```

```
    LRTable_path = './ActionGoto.csv'
    sourcefile = open(LRTable_path,'r',encoding = 'utf-8')
    reader = csv.reader(sourcefile)
    result = list(reader)
    columns = result[0]
    tabel_frame = tk.Frame(windowson)
    tabel_frame.pack()

    xscroll = Scrollbar(tabel_frame, orient=HORIZONTAL)
    yscroll = Scrollbar(tabel_frame, orient=VERTICAL)

    table = ttk.Treeview(
            master=tabel_frame,  # 父容器
            height=30,  # 表格显示的行数,height 行
            columns=columns,  # 显示的列
            show='headings',  # 隐藏首列
            xscrollcommand=xscroll.set,  # x 轴滚动条
            yscrollcommand=yscroll.set,  # y 轴滚动条
            )
    for column in columns:
        table.heading(column=column, text=column, anchor=CENTER,
                      command=lambda name=column:
                      messagebox.showinfo('', '{}描述信息
~~~'.format(name)))  # 定义表头
    table.column(column=column, width=100, minwidth=100,
anchor=CENTER, )  # 定义列
    xscroll.config(command=table.xview)
    xscroll.pack(side=BOTTOM, fill=X)
    yscroll.config(command=table.yview)
    yscroll.pack(side=RIGHT, fill=Y)
    table.pack(side=BOTTOM, fill=BOTH, expand=True)

    insert(table, result)

# 界面显示语法分析分析栈
def Synstackdisplay():
    SynScr2.delete(1.0, END)
    FirstSets_path = './StackInfo.txt'
    sourcefile = open(FirstSets_path,'r',encoding = 'utf-8')
    for line in sourcefile:
        SynScr2.insert('end',line)
```

```
# 界面显示语法分析语法树
def SynTreedisplay():
    SynScr2.delete(1.0, END)
    FirstSets_path = './SynTree.txt'
    sourcefile = open(FirstSets_path,'r',encoding = 'utf-8')
    for line in sourcefile:
        SynScr2.insert('end',line)

if __name__ == '__main__':
    # 第 1 步，实例化 object，建立窗口 window
    window = tk.Tk()
    # 第 2 步，给窗口的可视化起名字
    window.title('词法/语法分析器')
    # 第 3 步，设定窗口的大小(长 * 宽)
    window.geometry('800x700')  # 这里的乘是小 x

    #标题
    Title = tk.Label(window, text = '词法、语法分析器', font=('楷体',
40), width=20, height=1)
    #Title.pack()
    Title.place(x=125, y=20)

    #词法分析
    button1 = tk.Button(window, text = "读入程序", font=('楷体', 12),
fg="red", bg="yellow", width=10, height=1, command=SourceProGet)
    button1.place(x=40, y=110)
    LexScr1 = scrolledtext.ScrolledText(window, width=45, height=13,
font=("隶书",12), bd =5)
    LexScr1.place(x=0,y=150)
    button2 = tk.Button(window, text = "词法分析", font=('楷体', 12),
fg="red", bg="yellow", width=10, height=1, command=LexAnalyzeFunc)
    button2.place(x=140, y=110)
    button3 = tk.Button(window, text = "词法语法", font=('楷体', 12),
fg="red", bg="yellow", width=10, height=1, command=LexRuleDisplay)
    button3.place(x=240, y=110)
    LexScr2 = scrolledtext.ScrolledText(window, width=45, height=13,
font=("隶书",12), bd =5)
    LexScr2.place(x=400,y=150)
    button4 = tk.Button(window, text = "词法分析结果", font=('楷体', 12),
fg="red", bg="yellow", width=14, height=1, command=LexResult)
    button4.place(x=530, y=110)
```

```
    #语法分析
    buttonSyn = tk.Button(window, text = "语法分析", font=('楷体', 12),
fg="red", bg="yellow", width=10, height=1, command=SynAnalyzeFunc)
    buttonSyn.place(x=30, y=380)

    button5 = tk.Button(window, text = "文法语法", font=('楷体', 12),
fg="red", bg="yellow", width=10, height=1, command=SynRuleDisplay)
    button5.place(x=130, y=380)
    button6 = tk.Button(window, text = "FIRST 集合", font=('楷体', 13),
fg="red", bg="yellow", width=10, height=1, command=SynFIRSTDisplay)
    button6.place(x=230, y=380)
    SynScr1 = scrolledtext.ScrolledText(window, width=45, height=13,
font=("隶书",12), bd =5)
    SynScr1.place(x=0,y=420)

    button7 = tk.Button(window, text = "符号栈分析过程", font=('楷体',
12), fg="red", bg="yellow", width=16, height=1,
command=Synstackdisplay)
    button7.place(x=460, y=380)
    button8 = tk.Button(window, text = "语法树", font=('楷体', 12),
fg="red", bg="yellow", width=10, height=1, command=SynTreedisplay)
    button8.place(x=610, y=380)
    SynScr2 = scrolledtext.ScrolledText(window, width=45, height=13,
font=("隶书",12), bd =5)
    SynScr2.place(x=400,y=420)

    #显示分析表
    button9 = tk.Button(window, text = "分析表（ACTION&GOTO）", font=('楷
体', 12), fg="red", bg="yellow", width=25, height=1, command=csvdisplay)
    button9.place(x=280, y=660)

        window.mainloop()
```

7.2 SynAnalyze.py

```
from FA import LRDFANode
import pandas as pd
import numpy as np
import json
import sys

class SynAnalyze(object):
```

```
    "语法分析类"
    def __init__(self):
        self.firstSet = dict()              # 终结符和非终结符的 first 集
        self.productions = list()           # 产生式列表
        self.terminators = list()           # 终结符集合
        self.nonTerminators = list()        # 非终结符集合
        self.productionsDict = dict()       # 将产生式集合按照左侧的非终结
符归类
        self.LRTable = dict()               # LR(1)分析表

    def readSynGrammar(self, filename):
        "读语法分析规则，得到产生式，储存在 production 中"
        for line in open(filename, 'r'):
            line = line.strip()
            cur_left = line.split(':')[0]
            cur_right = line.split(':')[1]
            right_list = list()
            # 产生式右侧不止一个元素
            if cur_right.find(' ') != -1:
                right_list = cur_right.split(' ')
            # 产生式右侧就一个元素
            else:
                right_list.append(cur_right)
            production = {cur_left: right_list}
            self.productions.append(production)

    def getTerminatorsAndNon(self):
        "得到终结符和非终结符"
        all_elem = list()
        for production in self.productions:
            for left in production.keys():
                if left not in self.productionsDict:
                    self.productionsDict[left] = list()
                self.productionsDict[left].append((
                    tuple(production[left]),
                    self.productions.index(production)))  # 待思考，或许只
要存 index
                if left not in all_elem:
                    all_elem.append(left)
                if left not in self.nonTerminators:
                    self.nonTerminators.append(left)
                for right in production[left]:
```

```
                    if right not in all_elem:
                        all_elem.append(right)

        # 终结符集合 = 所有元素 - 非终结符集合
        # self.terminators = all_elem - self.nonterminators
        for i in all_elem:
            if i not in self.nonTerminators:
                self.terminators.append(i)

    def getFirstSetForOne(self, cur_status, is_visit):
        "求单个符号的 First 集"
        # 如果当前符号的 first 集已经算过，则直接返回
        if cur_status in self.firstSet.keys():
            return self.firstSet[cur_status]
        cur_first_set = list()
        # 如果当前符号是终结符，则返回该终结符
        if cur_status in self.terminators:
            if cur_status not in cur_first_set:
                cur_first_set.append(cur_status)
            return cur_first_set
        # 如果当前符号是非终结符，则继续计算
        if cur_status not in is_visit:
            is_visit.append(cur_status)
        # 对左边为该非终结符的所有产生式进行计算
        for right_list in self.productionsDict[cur_status]:
            flag = True
            for right in right_list[0]:
                if right == '$':
                    cur_first_set.append('$')
                    break
                if right in is_visit:  # 如果该元素在递归过程中已经访问过，
则不再计算
                    continue
                cur_set = self.getFirstSetForOne(right, is_visit)  # 递归
求解
                if '$' in cur_set:
                    for i in cur_set:
                        if i not in cur_first_set:
                            cur_first_set.append(i)
                    cur_first_set.remove('$')
                else:
                    for i in cur_set:
```

```
                        if i not in cur_first_set:
                            cur_first_set.append(i)
                    flag = False
                    break
                if '$' not in cur_set:  # 如果当前符号不能推出空，则不在向后计算
                    break
            if flag:
                cur_first_set.append('$')

        return cur_first_set

    def getFirstSets(self):
        "求 First 集"
        for terminator in self.terminators:
            self.firstSet[terminator] = self.getFirstSetForOne(
                terminator, list())
        for nonterminator in self.nonTerminators:
            self.firstSet[nonterminator] = self.getFirstSetForOne(
                nonterminator, list())

    # cur_item: (产生式编号，产生式左侧，产生式右侧符号列表，圆点位置，向前搜索符集合)
    def getClosure(self, cur_item, item_set):
        "求闭包"
        item_set.append(cur_item)
        right_list = cur_item[2]
        point_index = cur_item[3]
        tail_set = cur_item[4]
        # 圆点后为非终结符才继续增大项目集
        if point_index < len(right_list) and (right_list[point_index] in self.nonTerminators):
            # 计算以该非终结符为左侧的产生式的向前搜索符集合
            new_tail_set = list()
            flag = True
            for i in range(point_index + 1, len(right_list)):
                cur_first_set = self.firstSet[right_list[i]]
                if '$' in cur_first_set:
                    for j in cur_first_set:
                        if j != '$':
                            new_tail_set.append(j)
                else:
```

```
                    flag = False
                    for j in cur_first_set:
                        if j not in new_tail_set:
                            new_tail_set.append(j)
                    break
            if flag:
                for j in tail_set:
                    if j not in new_tail_set:
                        new_tail_set.append(j)
            # 对以圆点后非终结符为左侧的产生式进行遍历
            for pro_list in
self.productionsDict[right_list[point_index]]:
                new_item = (pro_list[1], right_list[point_index],
                            pro_list[0], 0, tuple(new_tail_set))
                if new_item not in item_set:  # 递归计算
                    for i in self.getClosure(new_item, item_set):
                        if i not in item_set:
                            item_set.append(i)
            # 合并项目集中产生式相同的项目
            new_item_set = dict()
            for item in item_set:
                pro_key = (item[0], item[1], item[2], item[3])
                if tuple(pro_key) not in new_item_set.keys():
                    new_item_set[tuple(pro_key)] = list()
                for i in item[4]:
                    if i not in new_item_set[tuple(pro_key)]:
                        new_item_set[tuple(pro_key)].append(i)
            item_set = list()
            for key in new_item_set.keys():
                final_item = (key[0], key[1], key[2], key[3],
                              tuple(new_item_set[key]))
                if tuple(final_item) not in item_set:
                    item_set.append(tuple(final_item))
        return item_set

    def createLRTable(self, LRTable_path):
        "创建 LR 分析表,若文法不是 LR1 的 返回 False"
        all_status = dict()
        all_item_set = dict()

        def getLRDFANode(id):
            if id in all_status:
```

```
                return all_status[id]
            else:
                node = LRDFANode(id)
            return node
        # 建立开始项目集 I0
        start_id = 0
        start_node = getLRDFANode(start_id)
        start_item_set = self.getClosure(
            (0, 'sstart', ('start',), 0, ('#',)), list())
        start_node.addItemSetBySet(start_item_set)
        all_item_set[tuple(start_item_set)] = start_id
        all_status[start_id] = start_node
        # BFS
        queue = list()
        queue.append(start_node)
        moving_id = 0  # 动态生成项目集编号
        while queue:
            now_node = queue.pop(0)
            now_item_set = now_node.itemSet
            now_node_id = now_node.id
            is_visit = list()
            is_visit.clear()
            for item in now_item_set:
                if item in is_visit:
                    continue
                is_visit.append(item)
                pro_id = item[0]
                left = item[1]
                right_list = item[2]
                point_index = item[3]
                tail_set = item[4]
                if point_index >= len(right_list) or '$' in
right_list:  # 归约
                    if now_node_id not in self.LRTable.keys():  # 为当前
项目集在 LR 分析表中创立表项
                        self.LRTable[now_node_id] = dict()
                    for tail in tail_set:
                        if tail in self.LRTable[now_node_id].keys():
                            print('当前文法不属于 LR(1)文法！！！')
                            return False
                        if pro_id:
                            self.LRTable[now_node_id][tail] = (
```

```
                                'r', pro_id)  # 用第 pro_id 个产生式进行归
约
                        else:
                            self.LRTable[now_node_id][tail] = ('acc',)
                else:
                    next_item = (pro_id, left, right_list,
                                 point_index + 1, tail_set)
                    next_item_set = self.getClosure(next_item, list())
                    next_c = right_list[point_index]
                    for ex_item in now_item_set:  # 对后一个字符相同的其他
产生式也要求闭包，并取并
                        if ex_item in is_visit:
                            continue
                        ex_right_list = ex_item[2]
                        ex_point_index = ex_item[3]
                        if ex_point_index < len(ex_right_list) and
ex_right_list[ex_point_index] == next_c:
                            if ex_item not in is_visit:
                                is_visit.append(ex_item)
                            ex_next_item = (
                                ex_item[0], ex_item[1], ex_item[2],
ex_point_index + 1, ex_item[4])
                            for i in self.getClosure(ex_next_item,
list()):
                                if i not in next_item_set:
                                    next_item_set.append(i)
                    # 合并项目集中产生式相同的项目
                    new_item_set = dict()
                    for iitem in next_item_set:
                        pro_key = (iitem[0], iitem[1], iitem[2],
iitem[3])
                        if tuple(pro_key) not in new_item_set.keys():
                            new_item_set[tuple(pro_key)] = list()
                        for i in iitem[4]:
                            if i not in new_item_set[tuple(pro_key)]:
                                new_item_set[tuple(pro_key)].append(i)
                    next_item_set = list()
                    for key in new_item_set.keys():
                        final_item = (key[0], key[1], key[2],
                                      key[3], tuple(new_item_set[key]))
                        if tuple(final_item) not in next_item_set:
                            next_item_set.append(tuple(final_item))
```

```
                    if tuple(next_item_set) in all_item_set.keys():  # 该
项目集已经处理过
                        next_node_id = all_item_set[tuple(next_item_set)]
                    else:  # 新建一个项目集
                        moving_id += 1
                        next_node_id = moving_id
                        all_item_set[tuple(next_item_set)] = next_node_id
                        next_node = getLRDFANode(next_node_id)
                        next_node.addItemSetBySet(next_item_set)
                        all_status[next_node_id] = next_node
                        queue.append(next_node)
                    if now_node_id not in self.LRTable.keys():  # 为当前
项目集在 LR 分析表中创立表项
                        self.LRTable[now_node_id] = dict()
                    if right_list[point_index] in
self.LRTable[now_node_id].keys():
                        print('当前文法不属于 LR(1)文法！！！')
                        return False
                    if right_list[point_index] in self.terminators:
                        self.LRTable[now_node_id][right_list[point_index]
] = (
                            'S', next_node_id)
                    else:
                        self.LRTable[now_node_id][right_list[point_index]
] = (
                            'G', next_node_id)
        actColumns = self.terminators
        actColumns.append("#")
        action = pd.DataFrame(index=list(
            self.LRTable.keys()), columns=actColumns)
        goto = pd.DataFrame(index=list(self.LRTable.keys()),
                            columns=self.nonTerminators)
        for i in self.LRTable.keys():
            for j in self.LRTable[i].keys():
                if self.LRTable[i][j][0] == 'G':
                    goto[j][i] = self.LRTable[i][j]
                else:
                    action[j][i] = self.LRTable[i][j]
        action[np.nan] = np.nan
        table = action.join(goto)
        table = table.drop(['$'], axis=1)
        table.to_csv(LRTable_path)
```

```
        return True

    def runOnLRTable(self, tokens,SynAnalyzeProcess_path):
        "开始分析"
        status_stack = [0]  # 状态栈
        symbol_stack = [('#', -1, 1)]  # 符号栈
        tree_layer = list()
        tree_layer_num = list()
        tree_line = list()
        tokens.reverse()
        isSuccess = False
        step = 0
        fp=open(SynAnalyzeProcess_path,'w')#分析过程存在这里
        message=''#报错信息/成功信息
        while True:
            step += 1
            top_status = status_stack[-1]
            now_line_num, now_token = tokens[-1]
            if step != 1:
                fp.write('\ntoken:%s'%now_token)
            else:
                fp.write('token:%s'%now_token)
            fp.write('\nsymbol stack:\n')
            fp.write(str(symbol_stack))
            fp.write('\nstatus stack:\n')
            fp.write(str(status_stack))
            fp.write('\n')

            if now_token in self.LRTable[top_status].keys():  # 进行状态
转移
                action = self.LRTable[top_status][now_token]
                if action[0] == 'acc':
                    isSuccess = True
                    break
                elif action[0] == 'S':
                    if len(tree_layer_num) == 0:
                        tree_layer_num.append(0)
                    else:
                        tree_layer_num[0] += 1
                    status_stack.append(action[1])
                    symbol_stack.append((now_token, 0,
tree_layer_num[0]))
```

```
                    tree_layer.append((now_token, 0, tree_layer_num[0]))
                    tokens = tokens[:-1]
                elif action[0] == 'r':
                    production = self.productions[action[1]]
                    left = list(production.keys())[0]
                    next_line = 0
                    if production[left] != ['$']:  # 不需修改两个栈
                        right_length = len(production[left])
                        status_stack = status_stack[:-right_length]
                        #symbol_stack = symbol_stack[:-right_length]
                        for i in range(len(symbol_stack) - 1,
len(symbol_stack) - right_length - 1, -1):
                            next_line = max(next_line,
symbol_stack[i][1])
                            tree_line.append(
                                [symbol_stack[i][1], symbol_stack[i][2],
0, 0])
                            symbol_stack.pop(i)
                        next_line += 1
                    else:
                        next_line = 1
                        right_length = 1
                        if len(tree_layer_num) == 0:
                            tree_layer_num.append(0)
                        else:
                            tree_layer_num[0] += 1
                        tree_layer.append(('$', 0, tree_layer_num[0]))
                        tree_line.append([0, tree_layer_num[0], 0, 0])
                    go = self.LRTable[status_stack[-1]][left]  # 归约时判
断接下来的状态
                    if next_line == len(tree_layer_num):
                        tree_layer_num.append(0)
                    else:
                        tree_layer_num[next_line] += 1
                    for i in tree_line[-right_length:]:
                        i[2], i[3] = next_line, tree_layer_num[next_line]
                    status_stack.append(go[1])
                    symbol_stack.append(
                        (left, next_line, tree_layer_num[next_line]))
                    tree_layer.append(
                        (left, next_line, tree_layer_num[next_line]))
                else:  # 无法进行状态转移，报错
```

```
                #print('line %s' % now_line_num)
                #print('found: %s' % now_token)
                #print('expecting:')
                message+='\nline %s\n' % now_line_num+'found: %s\n' %
now_token+'expecting:\n'
                for exp in self.LRTable[top_status].keys():
                    #print(exp)
                    message+=exp+'\n'
                break
        if isSuccess==True:
            message+= '\nSyntax Analyze Successfully!\n'
        else:
            message+= '\nSyntax Error!\n'
        fp.write(message)
        fp.close()
        print(message)
        return isSuccess, tree_layer, tree_line,message

    def get_tree(self, tree_layer, tree_line):
        "获取画语法树所需信息"
        pre_data = dict()
        for i in tree_layer:
            if i[1] not in pre_data:
                pre_data[i[1]] = list()
            pre_data[i[1]].append({'name': i[0]})
        for i in tree_line:
            if 'children' not in pre_data[i[2]][i[3]]:
                pre_data[i[2]][i[3]]['children'] = list()
            pre_data[i[2]][i[3]]['children'].insert(
                0, pre_data[i[0]][i[1]])
        data = pre_data[max(pre_data.keys())]
        #print(json.dumps(data))
        file = open('SynTree.txt', 'w')
        file.write(json.dumps(data))
        # for i in range(len(data)):
        #     s = str(data[i]).replace('{', '').replace('}',
'').replace("'", '').replace(':', ',') + '\n'
        #     file.write(s)

        #json.load(open('2.txt','w'),data)
        # Syn_tree = Tree().add("", data, orient="TB").set_global_opts(
        #     title_opts=opts.TitleOpts(title="Syn_Tree"))
```

```
        # Syn_tree.render(path=tree_path)

    def analyze(self, token_table_path,SynAnalyzeProcess_path):
        "语法分析，顶层函数"
        token_table = open(token_table_path, 'r')#读 token 表并处理
        tokens = list()
        for line in token_table:
            line = line[:-1]
            next_token_type = line.split(' ')[1]
            if next_token_type == 'identifier' or next_token_type ==
'number':
                tokens.append((line.split(' ')[0], next_token_type))
            else:
                next_token = line.split(' ')[2]
                tokens.append((line.split(' ')[0], next_token))
        tokens.append((str(0), '#'))
        token_table.close()
        isSuccess, tree_layer, tree_line,message =
self.runOnLRTable(tokens,SynAnalyzeProcess_path)#分析
        if isSuccess:#成功
            self.get_tree(tree_layer, tree_line)
            return True,message
        else:
            return False,message

#构造树的函数
def pretty_dict(obj, indent=' '):
    def _pretty(obj, indent):
        for i, tup in enumerate(obj.items()):
            k, v = tup
            # 如果是字符串则拼上""
            if isinstance(k,  str):
                k = '"%s"' % k
            if isinstance(v,  str):
                v = '"%s"' % v
            # 如果是字典则递归
            if isinstance(v, dict):
                v = ''.join(_pretty(v, indent + ' ' * len(str(k) + ':
{')))  # 计算下一层的 indent
            # case,根据(k,v)对在哪个位置确定拼接什么
            if i == 0:  # 开头,拼左花括号
                if len(obj) == 1:
```

```
                    yield '{%s: %s}' % (k, v)
                else:
                    yield '{%s: %s,\n' % (k, v)
            elif i == len(obj) - 1:  # 结尾,拼右花括号
                yield '%s%s: %s}' % (indent, k, v)
            else:  # 中间
                yield '%s%s: %s,\n' % (indent, k, v)

    #print(''.join(_pretty(obj, indent)))/home/yandy/公共的
/zhangbo/python/product/log
    logfile = open(r'/home/yandy/公共的
/zhangbo/python/product/log/tree.log', 'w')
    # logfile = open(r'F:\A-我的文档\C-工作区间\生产车间\Tsinghua Work
File\文档\TSMCmodule\python\log\log.log', 'w')

    print(''.join(_pretty(obj, indent)),file = logfile)
    logfile.close()

class c_exprer:
    left      = ''
    right     = []
    rightNum  = 0
    currIndex = 0

    # generate expression
    def __init__(self, str, currIndex = 0):
        self.left = str[0:str.find('~')]
        tmp = str[str.find('~')+1:len(str)]
        self.right = tmp.split(' ')
        self.currIndex = currIndex
        self.rightNum = len(self.right)

        if(self.right == ['^']): # END
            currIndex = 100

    # E->a.b, nextChar() is b
    def nextChar(self):
        if(self.currIndex >= self.rightNum):
            return None
        else:
            return self.right[self.currIndex]
```

```
# E->a.b, moveNext() get E->ab.
def moveNext(self):
    self.currIndex += 1
    if(self.currIndex >= self.rightNum):
        return 0
    else:
        return 1

# show myself, print expression
def show(self):
    str = ('%s ->' % self.left)
    i = 0

    if(self.right == ['^']): # END
        str += '^'
    else:
        for r in self.right:
            if(i == self.currIndex):
                str += ' . '
            str += r
            str += ' '
            i += 1
        if(self.currIndex >= self.rightNum):
            str += ' . '
    str += '\n'
    while(str.find('  ') >= 0):
        str = str.replace('  ', ' ')
    sys.stdout.write(str)

#
# expression to string
#
def showToString(self):
    str = ('%s ->' % self.left)
    i = 0

    if(self.right == ['^']): # END
        str += '^'
    else:
        for r in self.right:
            if(i == self.currIndex):
                str += ' . '
```

```
                str += r
                str += ' '
                i += 1
            if(self.currIndex >= self.rightNum):
                str += ' . '
        str += '\n'
        while(str.find('  ') >= 0):
            str = str.replace('  ', ' ')
        return str


if __name__ == '__main__':
    SynGrammar_path = './SynGra.txt'  # 语法规则文件相对路径
    TokenTable_path = './LexAnaResult.txt'  # 存储 TOKEN 表的相对路径
    LRTable_path = './ActionGoto.csv'  # 存储 LR 表的相对路径

    syn_ana = SynAnalyze()
    syn_ana.readSynGrammar(SynGrammar_path)
    syn_ana.getTerminatorsAndNon()
    syn_ana.getFirstSets()
    syn_ana.createLRTable(LRTable_path)
    # print(syn_ana.firstSet)
        syn_ana.analyze(TokenTable_path,SynAnalyzeProcess_path="./StackI
nfo.txt")
```