

# 词法分析器开发报告

开发人员：

葛宁凌 学号：3019201264

孙毅宁 学号：3019234250

## 目录文件说明

- 主程序：**lexier\_main\_byfile.py** 用于从文件中读取测试用例，调用其他模块进行词法分析，并输出token序列。
- **RegexToNFA.py, NFAToDFA.py, DFAminimise.py** 依次完成从正则表达式到NFA，NFA到DFA，DFA最小化的转换。
- **Automata.py** 封装了有限自动机类。

其中，实例变量分别为状态集合、初态集合、终态集合、有穷字母表集合和状态转换字典集，组成了确定一个有穷自动机的五元式。方法为对自动机的构建、删减操作等，便于后续搭建NFA和DFA时直接使用。

- **BuildAutomata.py** 定义了由正则表达式构造NFA需要使用的函数，将在后续的 **Regex to NFA** 模块中详细讲解。
- **definitions.py**: 主要定义了sql词法的正则表达式。
- **finalDFA.py**: 将最终得到的最小化DFA封装在finalDFA类中，便于直接调用。
- **utility.py**: 定义了一些实用函数
- **dotfile.dot**: 用于绘制有穷自动机
- **finalDFA.png**: 最终的DFA

## 词法分析器工作流程

1. 写出SQL文法正则表达式
2. 由正则表达式生成NFA
3. 将NFA确定化为DFA
4. 将DFA最小化
5. 使用 **lexier\_main\_byfile.py** 主程序对输入语句进行扫描，通过前四步得到的DFA对词素进行识别，最后输出结果。

以上各部分内容都将在下述模块中详细讲解。

# 自动机相关数据结构与函数说明

以下内容主要是对 `Automata.py` 模块的解释

## 1. 数据结构

根据自动机的五元式表示法，`Automata`中有五个实例变量，依次是 `states`, `startStates`, `finalStates`, `alphabets`, `transitions`，分别是状态集合，初始状态集合，终止状态集合，有穷字母表和状态转换规则。

具体数据结构为：

```
1 states=set()
2 startStates=set()
3 finalStates=set()
4 alphabets=set()
5 transitions=dict()
```

需要说明的是，`transitions` 为字典的嵌套 `{fromstate:{tostate:input}}`，具体表现为：

```
1 transitions = {fromstate_1: tostate_dict_1, ..., fromstate_n: tostate_dict_n}
2 tostate_dict_i = {tostate_1: input_set_1, ..., tostate_m: input_set_m}
3 input_set_i = {input_1, ..., input_k}
```

## 2. 相关函数

外界对一个自动机的改变需要通过 `Automata` 类中相关函数完成，函数主要分为以下几类

- add类——向五元组中添加元素
  - `addStates`: 添加新状态
  - `addStartStates`: 添加新初始状态
  - `addFinalStates`: 添加新结束状态
  - `addAlphabets`: 添加新字母
  - `addTransitions`: 添加新转换规则
- get类——得到五元组的值（函数设置同add类）
- clear类——清空五元组的值（函数设置同add类）
- check类——检查当前自动机是否合法，是否为空等
- display类——通过print或visualize的方式展示自动机，`visualize`的实现将在“visualize”部分详细介绍
- keepOne类
  - `keepOneStartState`: 保持该状态机只有一个初始状态
  - `keepOneFinalState`: 保持该状态机只有一个终止状态

`keepOne`类函数在由正则表达式搭建自动机中使用，详情见“Regex to NFA”部分

# Regex of SQL

regex of SQL 主要在 `definitions.py` 里定义

```
1 INTregex = 0 | [1-9][0-9]*
2 FLOATregex = (0 | [1-9][0-9]*).[0-9]*
3 IDNregex = [A-Za-z_][A-Za-z0-9_]*
4 OPregex = >|=|<|=|<=|=|<=>|&&|||.|!|-|
5 SEregex = ()|,
6 STRINGregex = "C*" //在双引号中的任一字符都会被转换成C
```

KW和OP中只含字母的操作符将由IDNregex识别，在实际识别过程中，对识别得到为IDN的词素进行判断，是否属于KW或OP。若属于，输出token为KW或OP；否则输出IDN

得到综合的regex为

```
1 regexSQL = [
2     SEreg, #SE
3     INTreg, #INT
4     IDNreg, #IDN
5     FLOATreg, #FLOAT
6     OPreg, #OP
7     STRINGreg #STR
8 ]
9 RegexOfSQL = ' | '.join(regexSQL)
```

在实际识别中，还需要支持在初始状态时读入空格" "依然保持在初始状态，因此修改 `RegexOfSQL` 为：

```
RegexOfSQL = " * | (" + RegexOfSQL + ") "
```

此外，由于OP中有`( ) |`， KW中有`*`， 故实际的正则表达式采用了`[]`代替`()`， 采用`+`代替`|`， 采用`-`代替`*`

# Regex to NFA

## 1. 理论依据——正则表达式与有限自动机的等价性

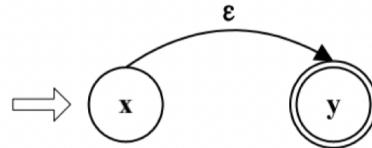
- 对于任何有穷自动机  $M$ ，都存在一个正则表达式  $r$ ，使得  $L(r) = L(M)$
- 对于任何正则表达式  $r$ ，都存在一个有穷自动机  $M$ ，使得  $L(M) = L(r)$

## 2. 构造过程

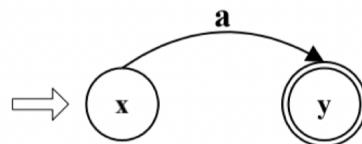
- 对于简单正则表达式：
  - 对于正则表达式  $\Phi$ ，构造NFA为：



- 对于正则表达式 $\epsilon$ , 构造NFA为:



- 对于正则表达式 $a$ ,  $a \in \Sigma$ , 构造的NFA为:

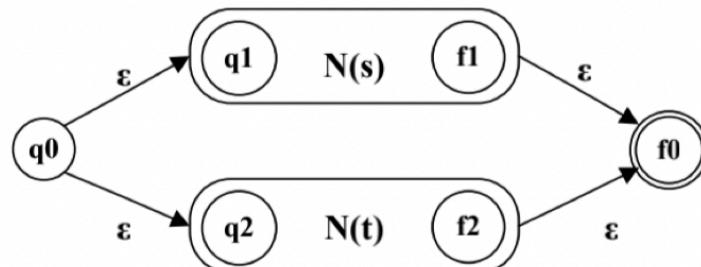


该建立过程主要由 `Automata.py` 中的 `setBasicStructure` 实现。

由于该过程在自动机的建立中使用频率非常高, 建立得到的自动机是复杂自动机的基本组成结构(代码中将其称为 `basicStructure`), 故而将该实现过程封装成 `Automata.py` 模块中的函数 `setBasicStructure`, 专门用于 `basicStructure` 的建立。

- 对于复杂的正则表达式: 若 $s, t$ 为 $\Sigma$ 上的正则表达式, 相应的NFA分别为 $N(s)$ 和 $N(t)$ , 则有:

- 对于正则表达式 $R = s|t$ , 所构造的NFA(R)为:

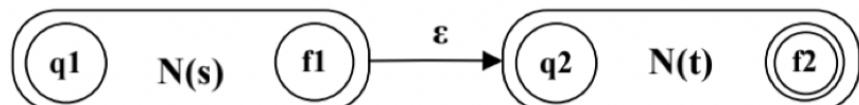


该过程由 `BuildAutomata.py` 中的 `orConcat` 函数实现。

建立过程: 向自动机 $N(S)$ 中添加 $N(t)$ 的状态、字母表、转关规则等, 同时更改两个自动机中状态的序号。在合并的最后增加唯一的初始状态 $q_0$ 和唯一的终止状态 $f_0$ , 并增加相应的转换规则。

调用了 `Automata.py` 中的 `add-` 类函数和 `keepOneStartState`、`keepOneFinalState` 等。

- 对于正则表达式 $R = st$ , 构造的NFA(R)为:

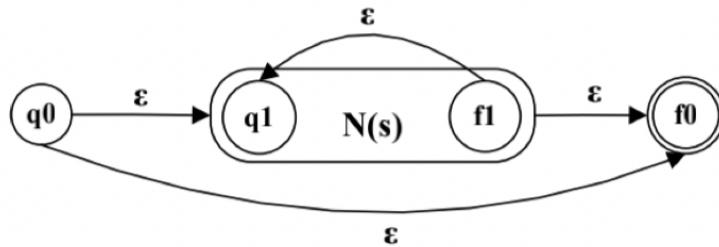


该过程由 `BuildAutomata.py` 中的 `dotConcat` 函数实现。

建立过程：需要先对两个输入的自动机做预处理，保证二者分别只有一个初始状态和一个终止状态。之后进行连接，连接过程同 `orConcat`。向自动机  $N(S)$  中添加  $N(t)$  的状态、字母表、转关规则等，同时更改两个自动机中状态的序号。

调用了 `Automata.py` 中的 `add-` 类函数和 `keepOneStartState`、`keepOneFinalState` 等。

- 对于正则表达式  $R = s*$ ，构造的NFA(R)为：



该过程由 `BuildAutomata.py` 中的 `starConstruct` 函数实现。

建立过程：需要先对输入的自动机做预处理，保证只有一个初始状态和一个终止状态。之后按照上述规则进行建立，主要为增加由  $f_1$  到  $q_1$  的转换规则，增加唯一的初始状态  $q_0$  和唯一的终止状态  $f_0$  以及相应的转换规则。

调用了 `Automata.py` 中的 `add-` 类函数和 `keepOneStartState`、`keepOneFinalState` 等。

### 3. Regex to NFA实现思路

将正则表达式视为由操作数和操作符组成，操作符为 `(` `)` `|` `*`，操作数为正则表达式中非操作符的有穷字母表中的元素。

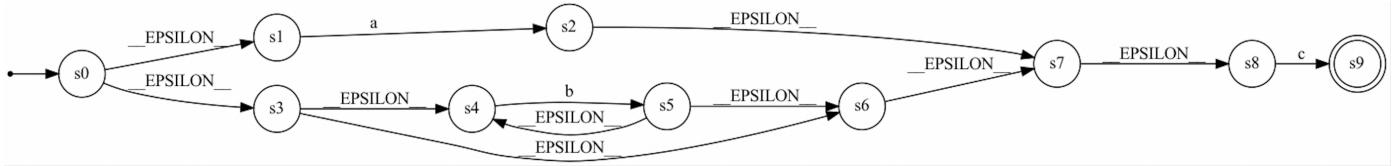
- 算法过程：

- 对正则表达式做预处理，添加“乘法”操作符，本实验中设置为反引号`
- 检查正则表达式输入是否合法
- 初始化两个栈：操作数栈和操作符栈
- 从左到右扫描正则表达式：
  - 若读入操作数，按照2中描述的情况“对于正则表达式  $a$ ,  $a \in \Sigma$ ”搭建状态机，并将状态机压入操作数栈中。
  - 若读入操作符，比较当前操作符和栈顶操作符的优先级大小：
    - 若当前操作符优先级高，则压入操作符栈。
    - 若当前操作符优先级相同，则弹出操作符栈顶元素。
    - 若当前操作符优先级低，弹出操作符栈顶元素，根据其具体类别弹出1或者2个操作数，生成相应状态图并压入操作数栈中。

### 5. 运行实例

如输入正则表达式 `(a|b*)c`

得到运行结果如下：



图中 EPSILON 即  $\epsilon$

图使用graphviz生成，具体过程见"visualize"

## NFA to DFA

### 1. 算法思路

- 得到状态集合的 $\epsilon$ -闭包
- 得到状态集合的 $\alpha$ -弧转换
- 从NFA初态 $S_0$ 出发，将该状态对应的 $\epsilon$ -闭包作为 $M'$ 的初态 $q_0$
- 分别把从 $q_0$ (对应于 $M$ 的状态子集 $I$ )出发，经过任意 $a \in S$ 的 $\alpha$ 弧转换 $I_a$ 所组成的集合作为 $M'$ 的状态，如此继续，直到不再有新的状态为止。

### 2. $\epsilon$ -闭包的实现

- 得到状态集合 $I$ 的 $\epsilon$ -闭包，伪代码描述如下：

```

1  for state_i in I do
2      e_CLOSURE[state_i] = {}
3      add state_i into e_CLOSURE[state_i]
4      queue.push(state_i)
5      while queue is not empty do
6          current_state = queue.pop()
7          for next_state in I do
8              if next_state can be reached from current_state by epsilon and is not in
9                  e_CLOSURE[state_i] do
10                 add next_state into e_CLOSURE[state_i]
11                 queue.push(next_state)
12             end
13         end
14     end

```

### 3. $\alpha$ -弧转换的实现

- 得到状态集合 $I$ 的 $\alpha$ -弧转换，伪代码描述如下：

```

1  for state_i in I do
2      alpha_transition[next_state] = {}
3      for next_state in transition[state_i] do
4          if next_state can be reached by alpha from state_i do
5              add e_CLOSURE[next_state] into alpha_transition[next_state]
6          end
7      end
8  end

```

## 4. DFA构造过程

- 数据结构说明
  - newState: 作为 $M'$ 新状态的原 $M$ 中的状态集合
  - statesQueue: 用于存储新状态newState的队列
  - newState\_list: 存放当前得到的 $M'$ 新状态的列表
- 伪代码描述

```

1  old_start_state = nfa.start_state
2  newState = e_CLOSURE[old_start_state]
3  newState_list.append(newState)
4  statesQueue.push(newState)
5  while statesQueue is not empty do
6      cur_state = stateQueue.pop()
7      for ch in nfa.alphabets do
8          newState = {}
9          for state in cur_state do
10             add alpha_transition[state] into newState
11         end
12         if newState not in newState_list do
13             newState_list.append(newState)
14             statesQueue.push(newState)
15         end
16         from_state = get_index_of(newState_list, cur_state)
17         to_state = get_index_of(newState_list, newState)
18         add transition(from_state, to_state, ch) into dfa.transitions
19     end
20  end
21  // set final states of dfa
22  for state in nfa.final_states do
23      new_final_state = get_state_index_of_dfa(state)
24      add new_final_state into dfa.final_states
25  end
26  set dfa.start_state = {0}
27  set dfa.state = {0, 1, 2, ..., length(newState_list)-1}
28  set dfa.alphabets = nfa.alphabets

```

- 上述代码中涉及的函数说明

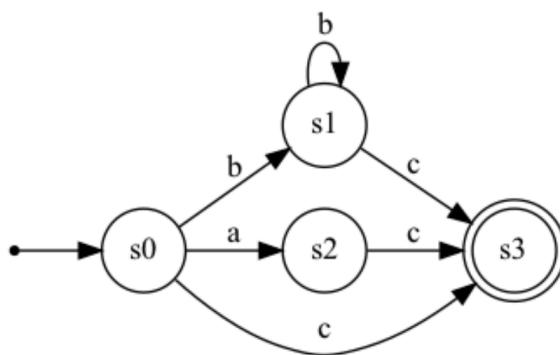
```

1  function get_index_of(newState_list(newState){
2      for i, state_set enumerate(in newState_list) do
3          if state_set == newState do
4              return i
5          end
6      end
7  }
8
9  function get_state_index_of_dfa(state){
10     //if state is in newState_list[i], then return i
11     for i, state_set enumerate(in newState_list) do
12         if state in state_set do
13             return i
14         end
15     end
16 }
```

## 5. 运行实例

输入正则表达式依然为  $(a|b^*)c$

经过 `NFAToDFA.py` 得到 DFA 如下图：



## DFA minimise

### 1. 算法思路

算法核心：将M的状态集分成不相交的子集

- DFA  $M = (S, \Sigma, \delta, s_0, S_t)$ , 最小状态 DFA  $M'$ 
  1. 构造状态的初始划分  $\prod$ : 终态  $S_t$  和非终态  $S - S_t$  两组
  2. 对  $\prod$  施用传播性原则构造新划分  $\prod_{new}$
  3. 如  $\prod_{new} = \prod$ , 则令  $\prod_{final} = \prod$  并继续步骤4, 否则  $\prod := \prod_{new}$  重复2
  4. 为  $\prod_{final}$  中的每一组选一代表, 这些代表构成  $M'$  的状态。若  $s$  是一代表, 且  $\delta(s, a) = t$ , 令  $r$  是  $t$  组的代表, 则  $M'$  中有一转换  $\delta'(s, a) = r$ 。  
 $M'$  的开始状态是含有  $s_0$  的那组的代表,  $M'$  的终态是含有  $s_t$  的那组的代表
  5. 去掉  $M'$  中的死状态

## 2. 实现过程

- 变量说明
  - newStates: 用于存放新划分的列表
  - minimised\_dfa: 所求的最小化DFA
  - dfa: 输入的DFA
- 总体过程伪代码描述

```
1 newStates = []
2 // 使用Automata类中的get类函数得到初始划分: 终态集和非终态集
3 newStates.append(dfa.getFinalStates())
4 newStates.append(dfa.getNonFinalStates())
5
6 // 得到新划分
7 newStates = createNewDivision(newStates)
8
9 // 设置minimisedDFA的五元组
10 minimised_dfa.startStates = {0}
11 minimised_dfa.states = {0,1, ..., length(newStates)-1}
12 minimised_dfa.alphabets = dfa.alphabets
13 // get new final states
14 for state in dfa.finalStates do
15     new_final_state = get_div_idx(state) //得到state所在新划分的编号
16     add new_final_state into minimised_dfa.finalStates
17 end
18 // get new transitions
19 for transition in dfa.transitions do
20     extract fromstate, tostate, input from transition
21     new_fromstate = get_div_idx(get_div_idx)
22     new_tostate = get_div_idx(tostate)
23     add (new_fromstate, new_tostate, input) into minimised_dfa.transitions
24 end
```

- createNewDivision()函数
  - 核心: 实现“传播性原则”
  - 思路: 检测preStates中每个集合是否能继续划分, 如果能, 将新划分加入。反复扫描, 直至划分在一趟扫描中不会发生改变。
  - 数据结构说明
    - state\_division\_map: 用于记录每一个状态当前属于的划分的序号
  - 伪代码描述

```
1 function createNewDivision(newStates){
2     //initialize state_division_map
3     for idx, state_set in enumerate(preStates) do
4         for state in state_set do
5             state_division_map[state] = idx
6     end
```

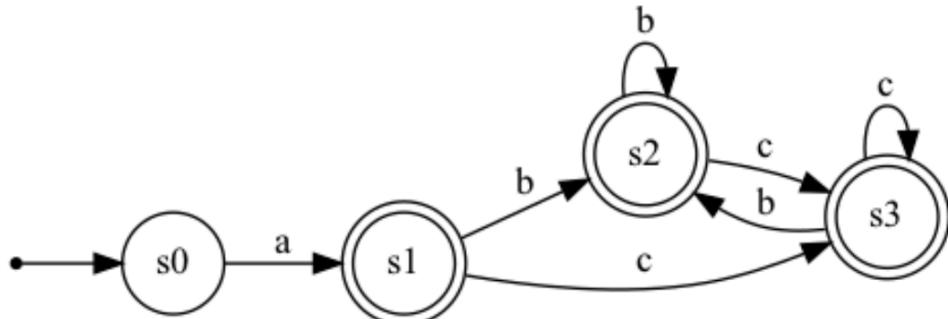
```

7     end
8     flag = true
9     while flag do
10    flag = false
11    for state_set in preStates do
12      for ch in dfa.alphabets
13        div = getDivision(state_set, ch)
14        if len(div) > 1 do //说明产生了新划分
15          flag = true
16          delete state_set from preStates
17          add div into preStates
18          update state_division_map
19          break
20        end
21      end
22    end
23  end
24 }
25
26 function getDivision(state_set, ch){
27   toState_dict = dict()
28   for state in state_set do
29     get toState reached by ch from state
30     // 得到toState对应的划分的编号
31     div_idx = state_division_map[toState]
32     if div_idx not in toState_idx do
33       toState_dict[div_idx] = set()
34     end
35     // 将state加入对应编号的集合中
36     add state into toState_dict[div_idx]
37   end
38   return toState_dict.values()
39 }

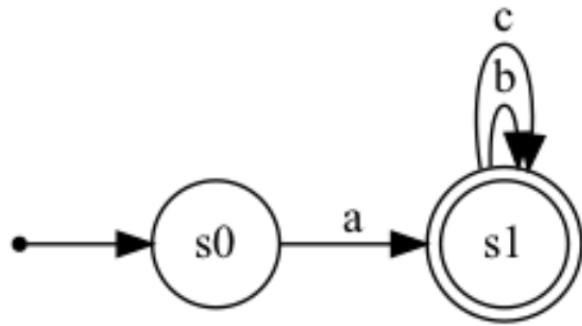
```

### 3. 运行实例

- 输入正则表达式  $a(b|c)^*$
- 得到DFA如下：



- 得到最小化DFA如下：



## Visualize

Visualize 主要在 `Automata.py` 中实现

- .dot文件生成，代码如下：

```

def getDotFile(self):
    dotFile = "digraph DFA {\nrankdir=LR\n"
    if len(self.states) != 0:
        if len(self.startStates) != 1:
            raise Exception("Automata must has only one start state!")
        dotFile += "root=s1\nstart [shape=point]\nstart->s%d\n" % next(iter(self.startStates))
        for state in self.states:
            if state in self.finalStates:
                dotFile += "s%d [shape=doublecircle]\n" % state
            else:
                dotFile += "s%d [shape=circle]\n" % state
        for fromstate, tostates in self.transitions.items():
            for state in tostates:
                for char in tostates[state]:
                    if char == '':
                        char = '\\\\'
                    dotFile += 's%d->s%d [label="%s"]\n' % (fromstate, state, char)
    dotFile += "}"
    with open("dotfile.dot", "w") as f:
        f.write(dotFile)
    return dotFile

```

- 使用graphviz生成自动机图片

```

def drawAutomata(self, file = ""):
    f = popen(r"dot -Tpng -o %s.png" % file, 'w')
    try:
        f.write(self.getDotFile())
    except:
        raise BaseException("Something wrong when drawing graph")
    finally:
        f.close()

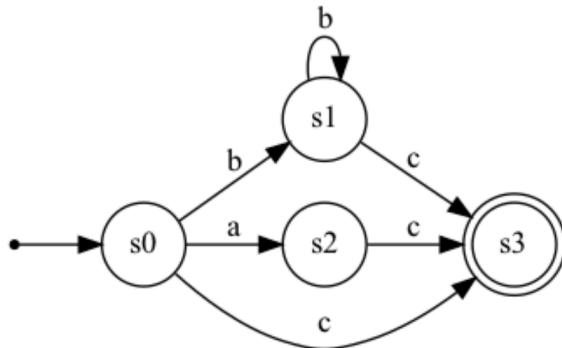
```

- 效果展示

- 输入正则表达式依然为  $(a|b^*)c$
- 生成.dot文件为：

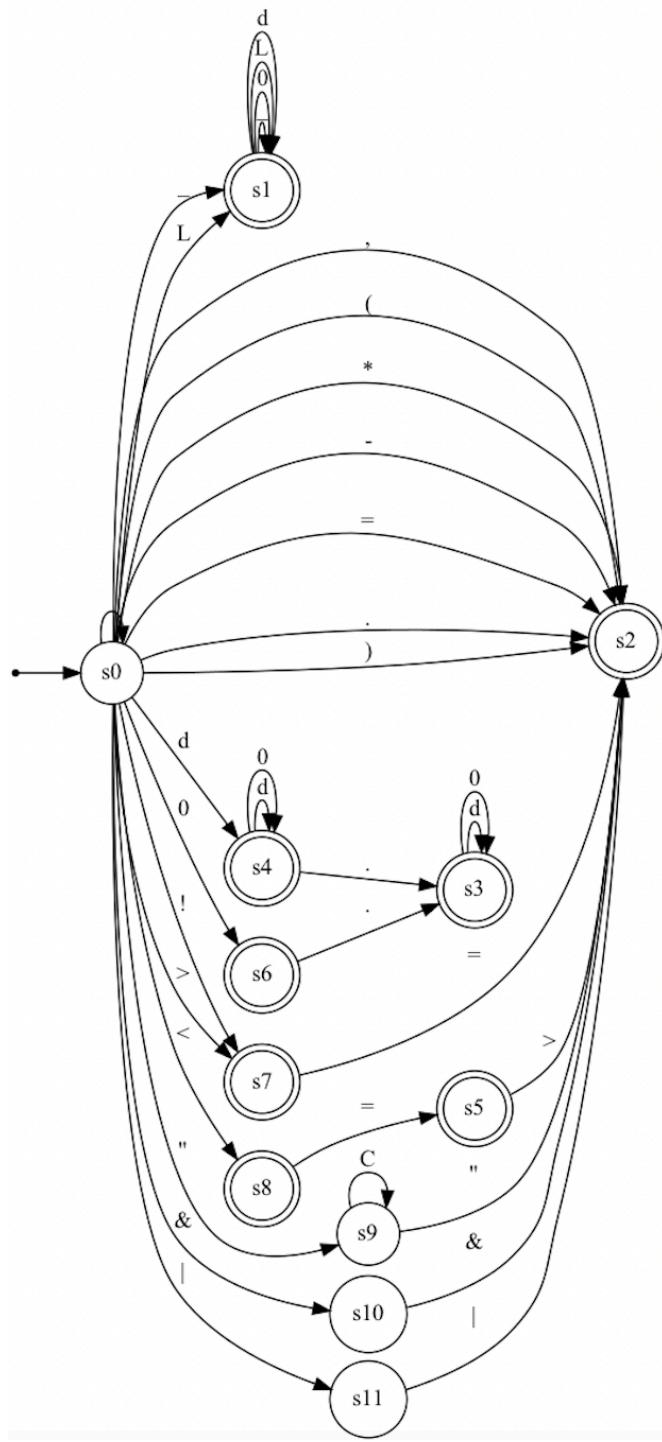
```
1 digraph DFA {  
2     rankdir=LR  
3     root=s1  
4     start [shape=point]  
5     start->s0  
6     s0 [shape=circle]  
7     s1 [shape=circle]  
8     s2 [shape=circle]  
9     s3 [shape=doublecircle]  
10    s0->s1 [label="b"]  
11    s0->s2 [label="a"]  
12    s0->s3 [label="c"]  
13    s1->s1 [label="b"]  
14    s1->s3 [label="c"]  
15    s2->s3 [label="c"]  
16 }
```

- 得到状态机为



## Final DFA

根据以上4个步骤( $regex \rightarrow NFA \rightarrow DFA \rightarrow minimised\_DFA$ )得到的最终状态机转换图如下图所示：



# Lexier Main

lexier main 主要在 `lexier_main_bufFile.py` 中实现

## 1. 算法思路

lexier main词法分析器，它接受两个参数，`input_file_path` 和 `output_file_path`，分别指定测试用例所在文件夹和输出结果存储位置。

```

1 if __name__ == '__main__':
2     INPUT_FILE = 'test_input'
3     OUTPUT_FILE = 'test_output'
4     LEXICAL_ANLYLIZER = Lexiyer(input_file_path=INPUT_FILE,
5         output_file_path=OUTPUT_FILE)
6     LEXICAL_ANLYLIZER.read_file(input_file='input.txt')

```

- 初始化词法单元token序列还有词素lexeme序列。
- 调用read\_file函数，它接受一个可以为None的参数input\_file。
- 对于指定的input\_file，process\_line函数会对该目标文件进行词法分析。
- token\_sequence\_save函数将生成的token序列存储进文件中。

## 2. 实现过程

- read\_file函数：
  - 词法分析器逐行读入并处理输入文件中的文本信息，扫描完毕后，存储每一行所对应的token序列。

```

1         with open(self.input_file, 'r') as read_file:
2             for line_index, line in enumerate(read_file.readlines()):
3                 line_index += 1
4                 line = line.strip()
5                 line += ' '
6                 if line:
7                     self.line_index = line_index
8                     self.process_line(line)
9                     self.save_token_for_input_infile(file)
10                    self.save_token_infile(file)

```

- 在处理输入文件中的每一行时，首先去掉每行中的换行符"\n"，并在每行的末尾加上“空格”。原因如下：代码逻辑中，判断当前lexeme的类型，是需要超前读入一个字符，这部分在后面有详细的说明。如果行尾没有“空格”，则会直接跳出循环，进入到下一行输入的处理中，那么，将无法判断并输出该行行尾lexeme的token序列。因此，在每行的末尾多加一个空格，是为了使状态机在读入每行最后一个字符后，进入到分支判断条件：判断状态机前一个状态是否为终态。
- process\_line函数：

- 变量说明：
  - ptr指向当前要读入的字符
  - name存储单个的词素
- 对于注释行的处理：
  - 单行注释：如果遇到行以"--"或者"#"开头，则表示该行为注释行，直接跳过；

```

1         elif line.strip().startswith("--"):
2             return
3         elif line.strip().startswith("#"):
4             return

```

- 多行注释：为了简化处理，注释的符号只出现在行的开头或者结尾。

```

1      if self.proc_line[self.ptr] == "/" and
2          self.proc_line[self.ptr+1] == "*":
3              self.line_flag = 1
4
5      if self.line_flag == 1:
6          while self.ptr < len(self.proc_line):
7              if self.proc_line[self.ptr] == "*" and
8                  self.proc_line[self.ptr + 1] == "/":
9                  self.line_flag = 0
10                 # print(self.proc_line[self.ptr])
11                 self.ptr += 1
12
13         return

```

代码的逻辑如下：

- 在行的开头遇到连续的'/'和'\*'；

将line\_flag置为1；

- 当line\_flag==1时；

在读入的每一行中搜索：

如果遇到连续的'/'和'\*'：

将line\_flag置为0+

直接返回；

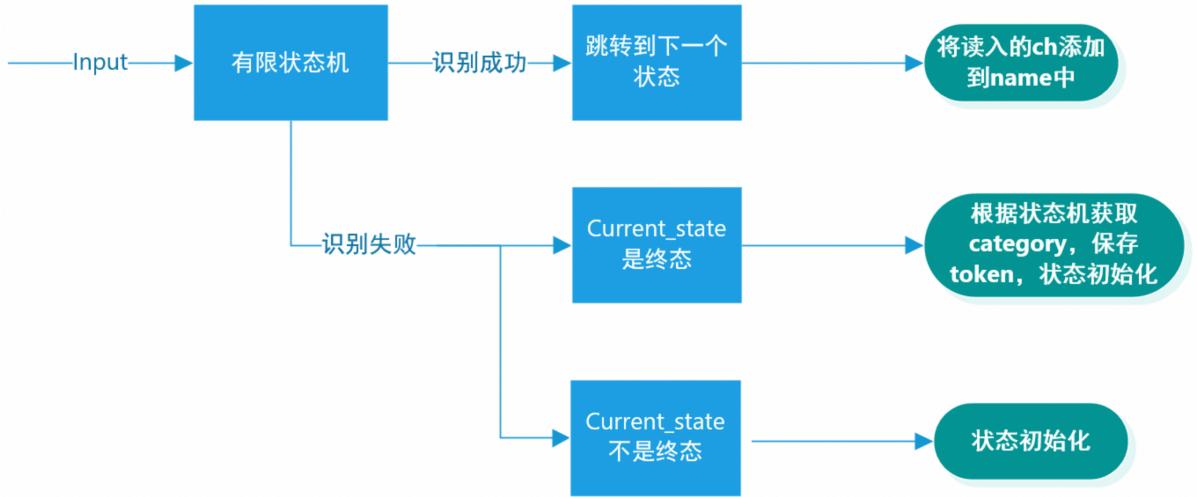
- 函数的处理流程如下

- 使用ptr来作为索引，对源程序进行逐行扫描。读入新的一行时，ptr重置为0，从该行的第一个字符开始读取，直至行尾。
- 将name以及str\_flag初始化，开始按照ptr所在的位置，逐个读入字符。
- 将读入的字符经过相应的处理后，输入到状态机中。状态机根据当前读入的字符input\_ch以及读入该字符前所在的状态current\_state来进行判断。将状态机中的节点与边转化为transitionTable（见图）。

===== transition table =====

(0, 'L')	:	1
(0, '_')	:	1
(0, '&')	:	11
(0, ' ')	:	10
(0, '.')	:	2
(0, '(')	:	2
(0, ',')	:	2
(0, '*')	:	2
(0, '=')	:	2
(0, ')')	:	2
(0, '-')	:	2
(0, 'd')	:	4
(0, ' ')	:	0
(0, '>')	:	5
(0, '!!')	:	5
(0, '0')	:	6
(0, '<')	:	8
(0, '...')	:	9
(1, 'd')	:	1
(1, 'L')	:	1
(1, '_')	:	1
(1, '0')	:	1
(11, '&')	:	2
(10, ' ')	:	2
(4, '0')	:	4
(4, 'd')	:	4
(4, '.')	:	3
(5, '=')	:	2
(6, '.')	:	3
(8, '=')	:	7
(9, 'C')	:	9
(9, '...')	:	2
(3, '0')	:	3
(3, 'd')	:	3
(7, '>')	:	2

等号'='左侧为：当前读入的字符input\_ch以及读入该字符前所在的状态current\_state，右侧为：将要跳转到的状态，具体的流程见图。



识别的结果有以下两种：

1. 识别成功：当前读入的字符input\_ch以及读入该字符前所在的状态current\_state在transitionTable的等号左侧有相对应的值，表示能够跳转到下一个状态。此时将ch添加到name中，ptr加1。
2. 识别失败：当前读入的字符input\_ch以及读入该字符前所在的状态current\_state在transitionTable的等号左侧没有相对应的值。此时有两种可能：
  - 情况1：读入该字符前所在的状态current\_state是终态；
  - 情况2：读入的字符为非法字符

```

1  if ((self.current_state, input_ch) in self.transitionTable):
2      self.current_state = self.transitionTable[(self.current_state,
3          input_ch)]
4      if input_ch != ' ':
5          self.name += ch
6          self.ptr += 1
7      else:
8          if self.current_state in self.finalStates:
9              upperName = self.name.upper()
10             if upperName in kwWithWhiteSpace:
11                 if self.proc_line[self.ptr:self.ptr + 3].upper() in
12                     kwAfterWhiteSpace:
13                     self.name += self.proc_line[self.ptr:self.ptr + 3]
14                     self.token_sequence_save(name=self.name, category='KW')
15                     self.ptr += 3
16                     self.current_state = 0
17                     self.name = ''
18                     str_flag = 0
19                     continue
20             else:
21                 self.token_sequence_save(name=self.name, category='IDN')
22                 self.now_state = self.current_state
23                 self.category_in = self.finalStateToType[self.now_state]
24                 if self.category_in == IDN:
25                     upperName = self.name.upper()

```

```

24         if upperName in keyWordList:
25             print("the category is KW")
26             self.token_sequence_save(name=self.name, category='KW')
27         elif upperName in opInIDNList:
28             print("the category is OP")
29             self.token_sequence_save(name=self.name, category='OP')
30         else:
31             print("the category is IDN")
32             self.token_sequence_save(name=self.name, category='IDN')
33         elif self.category_in == INT:
34             self.token_sequence_save(name=self.name, category='INT')
35         elif self.category_in == FLT:
36             self.token_sequence_save(name=self.name, category='FLOAT')
37         elif self.category_in == OP:
38             self.token_sequence_save(name=self.name, category='OP')
39         elif self.category_in == UNKNOWN:
40             back_ch = self.back_char()
41             if back_ch in [')', ')', ',', '']:
42                 self.token_sequence_save(name=self.name, category='SE')
43             elif back_ch == '':
44                 self.token_sequence_save(name=self.name,
45                                         category='STRING')
46             elif back_ch == '*':
47                 self.token_sequence_save(name=self.name, category='KW')
48             else:
49                 self.token_sequence_save(name=self.name, category='OP')
50             else:
51                 print("the DFA fails to identify the type.")
52
52         self.current_state = 0
53         self.name = ''
54         str_flag = 0
55     else:
56         self.current_state = 0
57         self.name = ''
58         str_flag = 0
59         print("the input is wrong.")
60         exit()

```

代码逻辑如下：

情况1：

- 将当前name中的内容大写成upperName；
- 如果upperName为“GROUP”或“ORDER”：
  1. 往后读三个字符，并将其大写；
  2. 如果是“BY”：
    - 将ch及其后两个字符添加到name中，调用token\_sequence\_save 函数（category

- 为"KW") , ptr+3, 初始化, continue;
3. 否则:
    - 调用token\_sequence\_save函数 (category为"IDN")
    - 否则:
      1. 调用finalStateToType函数, 输入当前的状态, 来获取对应的category。
      2. 如果category为"INT", "FLT", "OP", 则直接调用category为对应值的token\_sequence\_save函数;
      3. 如果category为"IDN":
        1. 如果upperName在keyWordList或者opInIDNList中:
          - 则直接调用category为对应值的token\_sequence\_save函数;
        2. 否则:
          - 直接调用category为"IDN"的token\_sequence\_save函数;
      4. 如果category为"UNKNOWN":
        1. 往后读取一个字符back\_char;
        2. 如果back\_char在seList中:
          - 则直接调用category为"SE"的token\_sequence\_save函数;
        3. 如果back\_char为双引号:
          - 则直接调用category为"SE"的token\_sequence\_save函数;
        4. 如果back\_char为'\*':
          - 则直接调用category为"SE"的token\_sequence\_save函数;
        5. 否则:
          - 直接调用category为"OP"的token\_sequence\_save函数;

情况2:

- 状态初始化, 退出程序。

- Token\_sequence\_save函数:

在/entity\_regex.py中重写了str方法

```
1 def __str__(self):
2     return '%s\t<%s,%s>' % (self.token_name, self.token_type, self.token_value)
```

修改了其输出格式为token序列的输出格式:

```
lexeme <token_type,token_value>.
```

接下来判断token\_value的值, 将name均变为大写的upper\_name:

1. 如果upper\_name属于key\_words:
  - token\_value为该upper\_name在kwdict中对应的编号;
2. 如果upper\_name属于operations:

- token\_value为该upper\_name在opdict中对应的编号;
- 如果upper\_name属于se:
    - token\_value为该upper\_name在sedict中对应的编号;
  - 否则:
    - token\_value为name中的内容;
  - 将该token添加到token\_sequence中

- save\_token\_infile函数:
    - 在/output/token\_sequence/保存token序列文件。
- 输出格式示例:

```

1  select          <KW,1>
2  *               <KW,5>
3  from           <KW,2>
4  employees      <IDN,employees>
5  select          <KW,1>
6  num            <IDN,num>
7  ,               <SE,3>
8  birth_date     <IDN,birth_date>
9  ,               <SE,3>
10 gender         <IDN,gender>
11 ,               <SE,3>
12 hire_date      <IDN,hire_date>
13 from           <KW,2>
14 employees      <IDN,employees>
15 where          <KW,3>
16 first_name     <IDN,first_name>
17 =              <OP,1>
18 "Paternela"   <STRING,"Paternela">
19 and            <OP,8>
20 last_name      <IDN,last_name>
21 =              <OP,1>
22 "Anick"        <STRING,"Anick">

```

- save\_token\_for\_input\_infile函数:
    - 在/output/token\_sequence/保存转化成语法分析所需的str序列文件。
- 输出格式示例:

```

1  SELECT
2  *
3  FROM
4  IDN
5  #

```

```
6  SELECT
7  IDN
8  ,
9  IDN
10 ,
11 IDN
12 ,
13 IDN
14 FROM
15 IDN
16 WHERE
17 IDN
18 =
19 STRING
20 AND
21 IDN
22 =
23 STRING
24 #
```

## 测试报告

基于老师给的测试用例进行测试，本组额外编写的测试用例见附件。

部分输出过长，截图时有所压缩，造成分辨率较低，会有点模糊，还请老师见谅。

### 老师的测试用例

#### test 1

```
INSERT INTO _tb1 VALUES (1,1.78,"SELECT")
```

得到token序列如下：

```
INSERT <KW,6>
INTO <KW,7>
_tb1 <IDN,_tb1>
VALUES <KW,8>
( <SE,1>
1 <INT,1>
, <SE,3>
1.78 <FLOAT,1.78>
, <SE,3>
"SELECT" <STRING,SELECT>
) <SE,2>
```

#### test 2

```
SELECT from_._1_,SUM(from_._2_) FROM from_ JOIN _1A ON from_._1_=_1A.cr7 WHERE from_._2_>1  
AND from_._3_<3.1415926 OR 1.25 IS NOT NULL GROUP BY from_._2_ HAVING from_._3_="ORDER BY  
#><=="
```

得到token序列如下：

```
SELECT <KW,1>  
from_ <IDN,from_>  
. <OP,16>  
_1_ <IDN,_1_>  
, <SE,3>  
SUM <KW,21>  
( <SE,1>  
from_ <IDN,from_>  
. <OP,16>  
_2_ <IDN,_2_>  
) <SE,2>  
FROM <KW,2>  
from_ <IDN,from_>  
JOIN <KW,14>  
_1A <IDN,_1A>  
ON <KW,17>  
from_ <IDN,from_>  
. <OP,16>  
_1_ <IDN,_1_>  
= <OP,1>  
_1A <IDN,_1A>  
. <OP,16>  
cr7 <IDN,cr7>  
WHERE <KW,3>  
from_ <IDN,from_>  
. <OP,16>  
_2_ <IDN,_2_>  
> <OP,2>  
1 <INT,1>  
AND <OP,8>  
from_ <IDN,from_>  
. <OP,16>  
_3_ <IDN,_3_>  
< <OP,3>  
3.1415926 <FLOAT,3.1415926>  
OR <OP,10>  
1.25 <FLOAT,1.25>  
IS <KW,31>  
NOT <OP,13>  
NULL <KW,32>  
GROUP BY <KW,24>  
from_ <IDN,from_>  
. <OP,16>  
_2_ <IDN,_2_>  
HAVING <KW,25>  
from_ <IDN,from_>  
. <OP,16>  
_3_ <IDN,_3_>  
= <OP,1>  
"ORDER BY #><==" <STRING,ORDER BY #><==>
```

## 自行编写的测试用例

### SELECT

```
select num from employees where first_name <= "123_hjs[]\" and last_name  
=" 'kokoi*%$"  
  
      select <KW,1>  
      num   <IDN,num>  
      from  <KW,2>  
      employees <IDN,employees>  
      where   <KW,3>  
      first_name <IDN,first_name>  
      <=    <OP,5>  
      "123_hjs[]\" <STRING,123_hjs[]\>  
      and    <OP,8>  
      last_name <IDN,last_name>  
      =    <OP,1>  
      "kokoi*%$" <STRING,'kokoi*%$\>
```

### INSERT

```
INSERT INTO ORDER(movie_t1,movie_y1,starName)VALUES( "Gone with  
wind_version(2.0)",1942,-12.125)  
  
      INSERT <KW,6>  
      INTO  <KW,7>  
      ORDER <IDN,ORDER>  
      (    <SE,1>  
      movie_t1 <IDN,movie_t1>  
      ,       <SE,3>  
      movie_y1 <IDN,movie_y1>  
      ,       <SE,3>  
      starName <IDN,starName>  
      )    <SE,2>  
      VALUES <KW,8>  
      (    <SE,1>  
      "Gone with wind_version(2.0)" <STRING,Gone with wind_version(2.0)>  
      ,       <SE,3>  
      1942  <INT,1942>  
      ,       <SE,3>  
      -     <OP,15>  
      12.125 <FLOAT,12.125>  
      )    <SE,2>
```

### DELETE

```
DELETE FROM group
```

```
      DELETE <KW,13>  
      FROM   <KW,2>  
      group  <IDN,group>
```

### GROUP BY

```
SELECT name,AVG(movie_1.length) FROM movie_1,exec_1 WHERE movie_1._2_ = exec_1.we_10 GROUP  
BY name
```

```
SELECT <KW,1>
name <IDN,name>
,<SE,3>
AVG <KW,20>
(<SE,1>
movie_1 <IDN,movie_1>
.<OP,16>
length <IDN,length>
)<SE,2>
FROM <KW,2>
movie_1 <IDN,movie_1>
,<SE,3>
exec_1 <IDN,exec_1>
WHERE <KW,3>
movie_1 <IDN,movie_1>
.<OP,16>
_2_ <IDN,_2_>
=<OP,1>
exec_1 <IDN,exec_1>
.<OP,16>
we_10 <IDN,we_10>
GROUP BY <KW,24>
name <IDN,name>
```

## MIN\_MAX\_AVG\_SUM

```
SELECT MIN(t.a_1),MAX(t.b),AVG(t._2a_),SUM(t.a_) FROM t WHERE t.a_1!=30.9 OR t.b>=0.2AND
t._2a_="string1.0**"XOR t.a_<=-20.1
```

```

SELECT <KW,1>
MIN <KW,18>
(
<SE,1>
t <IDN,t>
. <OP,16>
a_1 <IDN,a_1>
) <SE,2>
. <SE,3>
MAX <KW,19>
(
<SE,1>
t <IDN,t>
. <OP,16>
b <IDN,b>
) <SE,2>
. <SE,3>
AVG <KW,20>
(
<SE,1>
t <IDN,t>
. <OP,16>
_2a_ <IDN,_2a_>
) <SE,2>
. <SE,3>
SUM <KW,21>
(
<SE,1>
t <IDN,t>
. <OP,16>
a_ <IDN,a_>
) <SE,2>
FROM <KW,2>
t <IDN,t>
WHERE <KW,3>
t <IDN,t>
. <OP,16>
a_1 <IDN,a_1>
!= <OP,6>
30.9 <FLOAT,30.9>
OR <OP,10>
t <IDN,t>
. <OP,16>
b <IDN,b>
>= <OP,4>
0.2 <FLOAT,0.2>
AND <OP,8>
t <IDN,t>
. <OP,16>
_2a_ <IDN,_2a_>
= <OP,1>
"string1.0***" <STRING,string1.0**>
XOR <OP,12>
t <IDN,t>
. <OP,16>
a_ <IDN,a_>
<= <OP,5>
- <OP,15>
20.1 <FLOAT,20.1>

```

## HAVING

```

SELECT t.GROUP_BY_HAHA,MIN(t.order) AS ORDER FROM t,_t_ WHERE NOT BY_HAHA IS NULL GROUP
BY ORDER HAVING SUM(t.order)<-12.340

```

```
SELECT  <KW,1>
t       <IDN,t>
.
<OP,16>
GROUP  <IDN,GROUP>
BY_HAHA <IDN,BY_HAHA>
,
<SE,3>
MIN    <KW,18>
(
<SE,1>
t       <IDN,t>
.
<OP,16>
order   <IDN,order>
)
<SE,2>
AS     <KW,4>
ORDER   <IDN,ORDER>
FROM    <KW,2>
t       <IDN,t>
,
<SE,3>
_t     <IDN,_t_>
WHERE   <KW,3>
NOT    <OP,13>
BY_HAHA <IDN,BY_HAHA>
IS     <KW,31>
NULL   <KW,32>
GROUP BY      <KW,24>
ORDER   <IDN,ORDER>
HAVING <KW,25>
SUM    <KW,21>
(
<SE,1>
t       <IDN,t>
.
<OP,16>
order   <IDN,order>
)
<SE,2>
< <OP,3>
- <OP,15>
12.340  <FLOAT,12.340>
```

## UNION

```
SELECT supplier_id, supplier_name FROM suppliers WHERE supplier_id > 2000 UNION ALL SELECT
company_id, company_name FROM companies WHERE company_id > 1000 ORDER BY 1
```

```
SELECT <KW,1>
supplier_id <IDN,supplier_id>
,<SE,3>
supplier_name <IDN,supplier_name>
FROM <KW,2>
suppliers <IDN,suppliers>
WHERE <KW,3>
supplier_id <IDN,supplier_id>
> <OP,2>
2000 <INT,2000>
UNION <KW,22>
ALL <KW,23>
SELECT <KW,1>
company_id <IDN,company_id>
,<SE,3>
company_name <IDN,company_name>
FROM <KW,2>
companies <IDN,companies>
WHERE <KW,3>
company_id <IDN,company_id>
> <OP,2>
1000 <INT,1000>
ORDER BY <KW,27>
1 <INT,1>
```

## UPDATE

```
UPDATE departments SET dept_name = " __Dept" WHERE dept_no = "d005"
```

```
UPDATE <KW,11>
departments <IDN,departments>
SET <KW,12>
dept_name <IDN,dept_name>
= <OP,1>
" __Dept" <STRING,__Dept>
WHERE <KW,3>
dept_no <IDN,dept_no>
= <OP,1>
"d005" <STRING,d005>
```

## MIX

```
SELECT t.a,t.b,SUM(t.c),MAX(tt.d) FROM t,tt WHERE t.a>=t.d&&t.c!=t.b GROUP BY t.a HAVING
MIN(t.a)>-10.0 AND SUM(t.d)<=0 ORDER BY t.b
```

```
SELECT <KW,1>
t <IDN,t>
-<OP,16>
a <IDN,a>
,<SE,3>
t <IDN,t>
-<OP,16>
b <IDN,b>
-<SE,3>
```

'  
SUM <KW,21>  
( <SE,1>  
t <IDN,t>  
. <OP,16>  
c <IDN,c>  
) <SE,2>  
. <SE,3>  
MAX <KW,19>  
( <SE,1>  
tt <IDN,tt>  
. <OP,16>  
d <IDN,d>  
) <SE,2>  
FROM <KW,2>  
t <IDN,t>  
. <SE,3>  
tt <IDN,tt>  
WHERE <KW,3>  
t <IDN,t>  
. <OP,16>  
a <IDN,a>  
>= <OP,4>  
t <IDN,t>  
. <OP,16>  
d <IDN,d>  
&& <OP,9>  
t <IDN,t>  
. <OP,16>  
c <IDN,c>  
!= <OP,6>  
t <IDN,t>  
. <OP,16>  
b <IDN,b>  
GROUP BY <KW,24>  
t <IDN,t>  
. <OP,16>  
a <IDN,a>  
HAVING <KW,25>  
MIN <KW,18>  
( <SE,1>  
t <IDN,t>  
. <OP,16>  
a <IDN,a>  
) <SE,2>  
. <OP,16>

```
>      <OP,<>>
-      <OP,15>
10.0   <FLOAT,10.0>
AND    <OP,8>
SUM    <KW,21>
(      <SE,1>
t      <IDN,t>
.      <OP,16>
d      <IDN,d>
)      <SE,2>
<=     <OP,5>
0      <INT,0>
ORDER BY <KW,27>
t      <IDN,t>
.      <OP,16>
b      <IDN,b>
```

根据上述测试结果可得，词法分析器符合要求。