

天津大学

编译原理大作业 开发报告&测试报告



学 院 智能与计算学部

专 业 计算机科学与技术

姓名学号 贾天玉（3019244416）
 刘梦迪（3019244420）
 王志鸣（3019244395）
 李德新（3019244351）

指导教师 陈俊洁、刘爽老师

2022 年 5 月 7 日

SQL_compiler

TJU编译原理大作业——词法分析器与语法分析器设计

目录结构

```
SQL_COMPILER
|-lexical-analysis
  |-input
    |-test0A.sql
    |-test0B.sql
  |-output
    |-13Alex.tsv
    |-13Blex.tsv
  |-NFA.py
  |-NFAtoDFA.py
  |-DFAtoMFA.py
  |-lexer.py
  |-main.py
  |-README.md
|-synatax-annalysis
  |-input
    |-0Alex.tsv
    |-0Blex.tsv
  |-output
    |-13Agra.tsv
    |-13Bgra.tsv
  |-LL.py
  |-LR.py
  |-sql_syntax.txt
  |-README.md
|-README.md
```

开发环境

- windows
- python

一、词法分析器设计

(一) 开发报告

1 实现路径

1.1 实现思路

- 1. 根据实验指导书的单词符号构造出对应的NFA。
- 2. 编写确定化、最小化算法将构造的NFA自动转换成DFA。
- 3. 编写词法分析器的类执行词法分析。

1.2 需要实现的单词符号

在下表中列出需要实现的单词符号

关键字 (KW)

类别	语法关键词
查询关键词	(1) SELECT, (2) FROM, (3) WHERE, (4) AS, (5) *
插入表达式	(6) INSERT, (7) INTO, (8) VALUES, (9) VALUE, (10) DEFAULT
更新表达式	(11) UPDATE, (12) SET
删除表达式	(13) DELETE
连接操作	(14) JOIN, (15) LEFT, (16) RIGHT, (17) ON
聚合操作	(18) MIN, (19) MAX, (20) AVG, (21) SUM
集合操作	(22) UNION, (23) ALL
组操作	(24) GROUP BY, (25) HAVING, (26) DISTINCT, (27) ORDER BY
条件语句	(28) TRUE, (29) FALSE, (30) UNKONWN, (31) IS, (32) NULL

运算符 (OP)

运算符类型	语法关键词
比较运算符	(1) =, (2) >, (3) <, (4) >=, (5) <=, (6) !=, (7) <=>
逻辑运算符	(8) AND, (9) &&, (10) OR, (11) , (12) XOR, (13) NOT, (14) !
算数运算符	(15) -
属性运算符	(16) .

界符 (SE)

类型	语法关键词
界符	(1) (, (2)), (3) ,

标识符 (IDN) 以字母、数字和下划线 (_) 组成的不以数字开头的串

整数 (INT) 、浮点数 (FLAOT) 与C语言相同定义

字符串 (STRING) 使用双引号包含的任意字符串

2 算法描述

2.1 确定化算法

1. 首先从 s_0 出发, 仅经过任意条 ϵ 弧能够到达的状态组成的集合 I 作为 M' 的初态 q_0 .
2. 分别从 q_0 (对应于 M 的状态子集 I)出发, 经过任意 $a \in \Sigma$ 的 a 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。

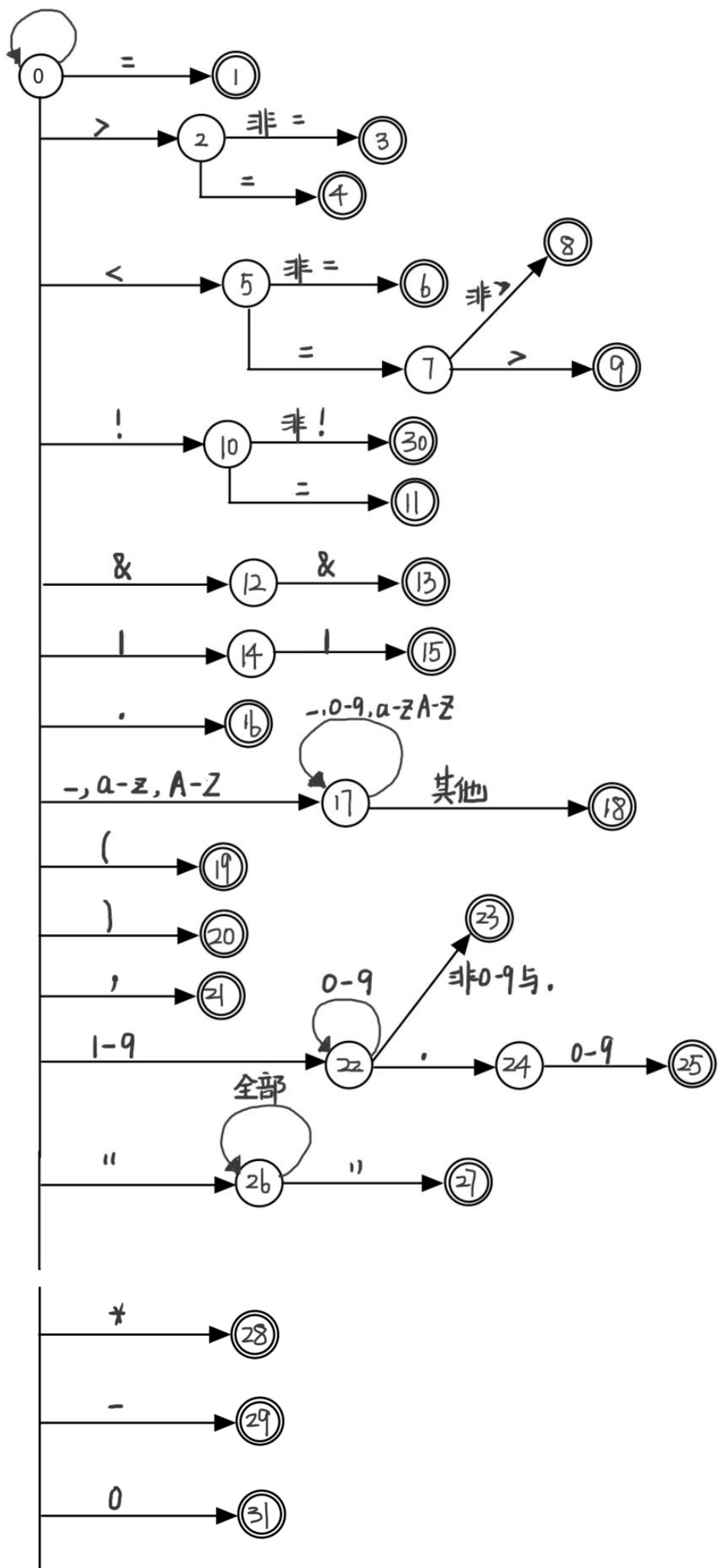
2.2 最小化算法

$DFAM = (S, \Sigma, \sigma, s_0, S_t)$, 最小状态 $DFAM'$

1. 构造状态的初始划分 Π : 终态 S_t 和非终态 $S - S_t$ 两组
2. 对 Π 施用传播性原则构造新划分 Π_{new}
3. 如 $\Pi_{new} = \Pi$, 则令 $\Pi_{final} = \Pi$ 并继续步骤4, 否则 $\Pi := \Pi_{new}$ 重复2
4. 为 Π_{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 $\sigma(s, a) = t$, 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。

2.3 状态转化图

如下图:



3 算法实现

运用**面向对象**的编程思想，将NFA,DFA和词法分析器等抽象成一个个类。

由简到繁，先将类的定义规划好，再逐渐扩充完善细节。

3.1 NFA的定义与实现

根据NFA定义依据五元组 $M = (S, \Sigma, \sigma, S_0, F)$ 构造一个**类**，使用面向对象的编程方式来进行词法分析器的编写。

定义的类：

1. Node

表示NFA中的一个节点，DFA中的节点定义并无区别，所以仍然使用该 `class`

- `id` 是节点的编号；
- `isFinal` 用来判断是否是终止节点；
- `isBackoff` 用来判断是否需要回退；
- `tag` 用于判定最终得到的token类别。

```
class Node:
    def __init__(self, id, is_final, is_back_off, tag):
        # id
        self.id = id
        # 是否是终结节点 1代表是，0代表不是
        self.isFinal = is_final
        # 是否需要回退 1代表需要，0代表不需要
        self.isBackOff = is_back_off
        # 只有终结节点需要tag
        self.tag = tag
```

2. Edge

表示NFA中的边。

要注意的是由于是 **NFA**，所以同一个 `tag` 可以**去往多个不同的节点**，因此这里 `toNodeIds` 使用的是 `set` 的数据结构，用来存放多个可能的节点。

- `fromNodeId` 表示有向边的出发节点；
- `tag` 表示获得 `tag` 可以从 `fromNodeId` 转化为 `toNodeIds`；
- `toNodeIds` 使用集合表示通过 `tag` 可以抵达的节点集合。

```
class Edge:
    def __init__(self, from_node_id, to_node_id: set, tag):
        # from 节点
        self.fromNodeId = from_node_id
        # to 节点 同一个tag可以去到的所有节点集合
        self.toNodeIds = to_node_id
        # 转化需要的信息，使用正则表达式表示
        self.tag = tag
```

3. NFA

依据绘制的NFA状态转化图进行初始化。

属性：

- `nodes` 数组中元素类型为 `class Node`
- `edges` 数组中元素类型为 `class Edge`
- `nowId` 代表当前状态机指向的位置；
- `startId` 代表初始节点的 `id` (与 `node` 的 `id` 属性对应)

函数：

- `add_node`：添加节点至 `nodes`
- `add_edges`：添加节点至 `edges`
- `get_start`：将指针指向开始节点
- `is_final`：判定当前指针是否在**终止节点**
- `is_back_off`：判定是否需要退出一个字符
- `get_tag`：获得 `node` 的 `tag`

```
class NFA:
    def __init__(self):
        # 存放节点
        self.nodes = []
        self.edges = []
        # 当前状态机所在的位置
        self.nowId = 0
        # 开始节点
        self.startId = 0

        # 初始化nodes和edges
        # OP
        self.add_node(0, 0, 0, "")
        self.add_node(1, 1, 0, "OP")
        self.add_node(2, 0, 0, "")
        self.add_node(3, 1, 1, "OP")
        self.add_node(4, 1, 0, "OP")
        self.add_node(5, 0, 0, "")
        self.add_node(6, 1, 1, "OP")
        self.add_node(7, 0, 0, "")
        self.add_node(8, 1, 1, "OP")
        self.add_node(9, 1, 0, "OP")
        self.add_node(10, 0, 0, "")
        self.add_node(11, 1, 0, "OP")
        # self.add_node(12, 0, 0, "")
        # self.add_node(13, 0, 0, "")
        # self.add_node(14, 1, 0, "AND")
        self.add_node(12, 0, 0, "")
        self.add_node(13, 1, 0, "OP")
        self.add_node(14, 0, 0, "")
        self.add_node(15, 1, 0, "OP")
        self.add_node(16, 1, 0, "OP")
        # 标识符IDN
        self.add_node(17, 0, 0, "")
        self.add_node(18, 1, 1, "IDNorKWorOP")
```

```

# 界符 SE
self.add_node(19, 1, 0, "SE")
self.add_node(20, 1, 0, "SE")
self.add_node(21, 1, 0, "SE")

# 整数、浮点数
self.add_node(22, 0, 0, "")
self.add_node(23, 1, 1, "INT")
self.add_node(24, 0, 0, "")
self.add_node(25, 1, 1, "FLOAT")

# 字符串
self.add_node(26, 0, 0, "")
self.add_node(27, 1, 0, "STRING")

# 后续补充的节点
self.add_node(28, 1, 0, "IDNorKWorOP")
self.add_node(29, 1, 0, "OP")
self.add_node(30, 1, 1, "OP")

# 0
self.add_node(31, 1, 0, "INT")

# 添加边的信息
# 部分OP到 <=>为止
self.add_edges(0, {0}, " ")
self.add_edges(0, {1}, "=")
self.add_edges(0, {2}, ">")
self.add_edges(2, {3}, "[^=]")
self.add_edges(2, {4}, "=")
self.add_edges(0, {5}, "<")
self.add_edges(5, {6}, "[^=]")
self.add_edges(5, {7}, "=")
self.add_edges(7, {8}, "[^>]")
self.add_edges(7, {9}, ">")
self.add_edges(0, {10}, "!")
self.add_edges(10, {11}, "=")

self.add_edges(0, {12}, "&")
self.add_edges(12, {13}, "&")
self.add_edges(0, {14}, "[\\|]")
self.add_edges(14, {15}, "[\\|]")

self.add_edges(0, {16}, "[\\. ]")

# 标识符和keyword
self.add_edges(0, {17}, "[_a-zA-Z]")
self.add_edges(17, {17}, "[_0-9a-zA-Z]")
self.add_edges(17, {18}, "[^_0-9a-zA-Z]")

self.add_edges(0, {19}, "[ (]")
self.add_edges(0, {20}, "[ )]")
self.add_edges(0, {21}, ",")

# int, float
self.add_edges(0, {22}, "[1-9]")
self.add_edges(22, {22}, "[0-9]")
self.add_edges(22, {23}, "[^\\.0-9]")

```



```

self.add_edges(22, {24}, "[\.]")
self.add_edges(24, {24}, "[0-9]")
self.add_edges(24, {25}, "[^0-9]")

# string
self.add_edges(0, {26}, "[\"'\"]")
self.add_edges(26, {26}, ".")
self.add_edges(26, {27}, "[\"'\"]")

# 后续补充的边
self.add_edges(0, {28}, "[\*]")
self.add_edges(0, {29}, "-")
self.add_edges(10, {30}, "[^!]")

# debug发现的
self.add_edges(0, {31}, "[0]")

# 添加节点
def add_node(self, id, is_final, is_back_off, tag):
    new_node = Node(id, is_final, is_back_off, tag)
    self.nodes.append(new_node)

# 添加边
def add_edges(self, from_node_id, to_node_ids: set, tag):
    new_edge = Edge(from_node_id, to_node_ids, tag)
    self.edges.append(new_edge)

# 将指针指向开始节点
def get_start(self):
    self.nowId = self.startId

# 是否结束
def is_final(self, id):
    # 因为是按照顺序添加的节点,所以nodes的下标对应着一样的id
    return self.nodes[id].isFinal

# 是否需要退出一个字符
def is_back_off(self, id):
    return self.nodes[id].isBackOff

# 获得tag
def get_tag(self, id):
    # 可以根据tag返回需要的内容
    return self.nodes[id].tag

```

3.2 DFA的定义与实现

基于已经实现的NFA，通过确定化算法和最小化算法，得到DFA。

定义的类:

1. DFAEdge

与Edge的定义类似, 不同点在于 toNodeIds 的定义。DFA 中通过一个 tag 只能抵达一个确定的 node, 所以在 DFAEdge 中 toNodeIds 是一个 int 类型数据。

- fromNodeId 表示有向边的出发节点;
- tag 表示获得 tag 可以从 fromNodeId 转化为 toNodeIds;
- toNodeIds 使用集合表示通过 tag 可以抵达的节点。

```
class DFAEdge:
    def __init__(self, from_node_id, to_node_id: int, tag):
        # from 节点
        self.fromNodeId = from_node_id
        # to 节点 同一个tag可以去到的所有节点集合
        self.toNodeIds = to_node_id
        # 转化需要的信息
        self.tag = tag
```

2. DFA

通过已有的NFA进行确定化和最小化算法生成DFA。

属性:

- nodes 数组中元素类型为 class Node
- edges 数组中元素类型为 class DFAEdge
- nowId 代表当前状态机指向的位置;
- startId 代表初始节点的 id (与 node 的 id 属性对应)

函数:

- epsilon_closure: 计算闭包
- move: 计算move集合
- determine: 确定化算法
- add_node: 添加节点至 nodes
- add_edges: 添加节点至 edges

```
class DFA:
    def __init__(self):
        # 存放节点
        self.nodes = []
        self.edges = []
        # 当前状态机所在的位置
        self.nowId = 0
        # 开始节点
        self.startId = 0

        # 确定化将NFA转化为DFA, 将nodes和edges填上
        # self.determine(nfa)

    @staticmethod
    # e-closure计算
```

```

def epsilon_closure(self, node_set: set, nfa: NFA):
    # 查找node_set经过任意条epsilon弧能抵达的节点们
    edges = nfa.edges

    # 用于判断是否已经出现过了
    node_id_set = set()
    # 获得所有的node的id
    for node in node_set:
        node_id_set.add(node.id)

    for node in node_set:
        node_id = node.id
        for edge in edges:
            if edge.tag == "epsilon" and edge.fromNodeId == node_id:
                # 如果新的node不在node_set中则加入，否则跳过
                # 遍历所有的可以抵达的node
                for toNodeId in edge.toNodeIds:
                    if toNodeId in node_id_set:
                        continue
                    else:
                        # 将能够抵达的node加入new_node_set
                        node_set.add(nfa.nodes[toNodeId])

    return node_set

@staticmethod
def move(self, node_set: set, nfa: NFA, tag):

    edges = nfa.edges
    # 返回的全新node集合
    new_node_set = set()
    # 用于判断是否已经出现过了
    node_id_set = set()
    # # 获得所有的node的id
    # for node in node_set:
    #     node_id_set.add(node.id)

    # 遍历每一个node
    for node in node_set:
        for edge in edges:
            # 相同tag的匹配
            if edge.fromNodeId == node.id and edge.tag == tag:
                for toNodeId in edge.toNodeIds:
                    if toNodeId in node_id_set:
                        continue
                    else:
                        new_node_set.add(nfa.nodes[toNodeId])
    return new_node_set

# 确定化算法
def determine(self, nfa: NFA):
    self.nodes = []

    # 先计算nfa的起始节点的闭包
    start_node = nfa.nodes[nfa.startId]

```

```

# new_start_node_set = self.epsilon_closure(self, {start_node}, nfa)
new_start_node_set = {start_node}

# 初始化将初始点加入集合中
node_queue = [new_start_node_set]
now_id = 0
self.add_node(now_id, 0, 0, "")

# 因为是按照顺序进入的，所以point和from_node_id是相同的
point = 0
while point < len(node_queue):
    # 取出队列中未计算的最靠前的set
    node_set = node_queue[point]
    # 对每一个tag进行move计算
    for tag in tags:
        move_node_set = self.move(self, node_set, nfa, tag)
        # 如果是空则忽略
        if len(move_node_set) == 0:
            continue
        # 非空且未出现过需要连接edge，并添加node
        elif not (move_node_set in node_queue):
            # 先加入队列，用于继续计算
            node_queue.append(move_node_set)
            # 对DFA处理node和edges

            # 从move_node_set中的第一个节点获得is_final 和 is_back_off
            is_final = 0
            is_back_off = 0
            # 获得一个isFinal, isBackOff
            node_tag = ""
            for one in move_node_set:
                is_final = one.isFinal
                is_back_off = one.isBackOff
                node_tag = one.tag
                break
            now_id += 1
            # self.add_node(now_id, is_final, is_back_off, tag)
            # print(now_id, is_final, is_back_off, node_tag)
            self.add_node(now_id, is_final, is_back_off, node_tag)
            self.add_edges(point, now_id, tag)
        # 非空但出现过只需要连接edge
        else:
            # 计算to_node_id，就是在node_queue中的index
            to_node_id = node_queue.index(move_node_set)
            self.add_edges(point, to_node_id, tag)
    point += 1
# for i in self.edges:
#     print('fromId:' + str(i.fromNodeId) + ' tag:' + str(i.tag) + ' toNodeId:'
#           + str(i.toNodeIds))

# 添加节点
def add_node(self, id, is_final, is_back_off, tag):
    new_node = Node(id, is_final, is_back_off, tag)
    self.nodes.append(new_node)

```

```
# 添加边
def add_edges(self, from_node_id, to_node_id: int, tag):
    new_edge = DFAEdge(from_node_id, to_node_id, tag)
    self.edges.append(new_edge)
```

3.3 DFA的最小化

定义的类:

DFAtoMFA

属性与DFA类似，只是增加了一个dfa的属性，dfa为最小化之前的DFA。

属性:

- dfa 代表最小化之前的DFA
- nodes 数组中元素类型为 class Node
- edges 数组中元素类型为 class DFAEdge
- nowId 代表当前状态机指向的位置;
- startId 代表初始节点的 id (与 node 的 id 属性对应)

函数:

- getToNode: 从 node 经过 tag 转移能到达的所有状态的集合
- build: 构造MFA
- add_node: 添加节点至 nodes
- add_edges: 添加节点至 edges
- get_start: 将指针指向开始节点
- is_final: 判定当前指针是否在**终止节点**
- is_back_off: 判定是否需要退出一个字符
- get_tag: 获得 node 的tag
- next_id: 通过得到的 ch 获得一个 node 的 id
- get_token_type: 根据给出的 token 判断类型，用于输出
- get_token_num: 根据给出的 token 判定编号，用于输出
- getnewDfa: 根据新的节点ID和最小化之前的节点ID对应关系构造新的DFA

```
class DFAtoMFA: #DFA最小化
    def __init__(self,dfa):
        self.dfa = dfa
        # 存放节点
        self.nodes = []
        self.edges = []
        # 当前状态机所在的位置
        self.nowId = 0
        # 开始节点
        self.startId = 0
        self.buildMFA()

    # 从 node 经过 tag 转移能到达的所有状态的集合
    def getToNode(self,nodeId,tag):
        for i in self.dfa.edges:
            if(i.fromNodeId == nodeId and i.tag == tag):
                return i.toNodeIds
```

```

return 100

def buildMFA(self):
    # 如果只有一个节点，那它本身为MFA
    if len(self.dfa.nodes) <= 1:
        self.mfa = self.dfa
        return

    finalNoBackNodesOP = [] # 存储终态并且不会回退的节点的编号
    finalNoBackNodesSE = []
    finalNoBackNodesID = []
    finalNoBackNodesSTR = []

    finalBackNodesOP = [] # 存储终态并且会回退的节点的编号
    finalBackNodesID = []
    finalBackNodesINT = []
    finalBackNodesFL = []

    noFinalNodes = []
    nodeIds = [] # 存储所有节点的Id
    for i in self.dfa.nodes:
        nodeIds.append(i.id)
        if i.isFinal == 1 and i.isBackOff == 1:
            if i.tag == 'OP':
                finalBackNodesOP.append(i.id)
            if i.tag == 'IDNorKWorOP':
                finalBackNodesID.append(i.id)
            if i.tag == 'INT':
                finalBackNodesINT.append(i.id)
            if i.tag == 'FLOAT':
                finalBackNodesFL.append(i.id)

        if i.isFinal == 1 and i.isBackOff == 0:
            if i.tag == 'OP':
                finalNoBackNodesOP.append(i.id)
            if i.tag == 'SE':
                finalNoBackNodesSE.append(i.id)
            if i.tag == 'IDNorKWorOP':
                finalNoBackNodesID.append(i.id)
            if i.tag == 'STRING':
                finalNoBackNodesSTR.append(i.id)

        if i.isFinal == 0:
            noFinalNodes.append(i.id)

    pos = dict(zip(nodeIds, range(len(nodeIds)))) # pos 中存储每一个节点对应处于的集合编号
    pos[100] = 9

    set1 = set(finalBackNodesOP)
    set2 = set(finalBackNodesID)
    set3 = set(finalBackNodesINT)
    set4 = set(finalBackNodesFL)

    set5 = set(finalNoBackNodesID)
    set6 = set(finalNoBackNodesOP)
    set7 = set(finalNoBackNodesSE)

```

```

set8 = set(finalNoBackNodesSTR)

set9 = set(noFinalNodes)

set1 = list(set1)
set2 = list(set2)
set3 = list(set3)
set4 = list(set4)
set5 = list(set5)
set6 = list(set6)
set7 = list(set7)
set8 = list(set8)
set9 = list(set9)

for i in set1:
    pos[i] = 0
for i in set2:
    pos[i] = 1
for i in set3:
    pos[i] = 2
for i in set4:
    pos[i] = 3
for i in set5:
    pos[i] = 4
for i in set6:
    pos[i] = 5
for i in set7:
    pos[i] = 6
for i in set8:
    pos[i] = 7
for i in set9:
    pos[i] = 8

allsets = [set1, set2, set3, set4, set5, set6, set7, set8, set9]
counts = 10
flag = True
while flag:
    flag = False
    for char in tags:
        for sub_set in allsets:
            dic = dict() # 存储节点和新对应的编号
            lists = []
            # 找出某个set中通过某个tag能够到达的所有节点
            for oneNode in sub_set:
                num = self.getToNode(oneNode, char)
                num = pos[num] # 获取转移状态对应的集合编号
                # print(num)

                if num not in dic.keys(): # 如果没有建立 该状态对应的集合编号
                    # 字典关系 新建
                    dic[num] = counts
                    counts += 1
                    lists.append(dic[num])
            if len(lists) == 0:
                continue

```

循环

合的编号

```
# print(lists)

if len(dic) > 1: #证明该集合中状态转移 不是转移到同一处 拆分元素 跳出
    flag = True
    tmp_set=dict() #新编号与新数值相对应 加入不同的新list
    for i1 in range(len(sub_set)):
        if lists[i1] not in tmp_set.keys():
            tmp_set[lists[i1]]=list()
            tmp_set[ lists[i1] ].append(sub_set[i1])
            pos[sub_set[i1]] = lists[i1] #更新pos 更新状态 所在的集
        else:
            continue

    allsets.remove(sub_set) #将旧的list移除
    for i1 in tmp_set.values(): #将新的list 加入
        allsets.append(i1)
    break

if flag == True:
    break

for i in range(len(allsets)): #计算出每个数值对应的新的数值
    for j in allsets[i]:
        pos[j] =i+1

# print(pos)
# print(set1)
pos.pop(100)
# print(pos)
# for i in self.dfa.nodes:
#     print('id: ' + str(i.id) + 'isBackOff: ' + str(i.isBackOff))

self.getnewDfa(pos,self.dfa)

# 添加节点
def add_node(self, id, is_final, is_back_off, tag):
    new_node = Node(id, is_final, is_back_off, tag)
    self.nodes.append(new_node)

# 添加边
def add_edges(self, from_node_id, to_node_id: int, tag):
    new_edge = DFAEdge(from_node_id, to_node_id, tag)
    self.edges.append(new_edge)

# 将指针指向开始节点
def get_start(self):
    self.nowId = self.startId

# 获得下一个ID
def next_id(self, tag):
    for edge in self.edges:
```



```

        if edge.fromNodeId == self.nowId and re.match(edge.tag, tag):
            # 并将nowId指向新的位置
            self.nowId = edge.toNodeIds
            # 说明成功找到下一个节点
            return True
        return False

def is_final(self, id):
    # 因为是按照顺序添加的节点,所以nodes的下标对应着一样的id
    for i in self.nodes:
        if i.id == id:
            return i.isFinal

# 是否需要退出一个字符
def is_back_off(self, id):
    for i in self.nodes:
        if i.id == id:
            return i.isBackOff

# 获得tag
def get_tag(self, id):
    # 可以根据tag返回需要的内容
    for i in self.nodes:
        if i.id == id:
            return i.tag

def get_token_type(self, token, node_tag):
    # KW, OP, SE, IDN, INT, FLOAT, STR

    # OP, SE, INT, FLOAT,STR都可以直接判断
    if node_tag == "OP" or node_tag == "SE" or node_tag == "INT" or node_tag
    == "FLOAT" or node_tag == "STRING":
        return node_tag
    elif node_tag == "IDNorKWorOP":
        keywords = TYPE_TO_CONTENT_DICT_KW.keys()
        ops = TYPE_TO_CONTENT_DICT_OP.keys()
        if token in keywords:
            return "KW"
        elif token in ops:
            return "OP"
        else:
            return "IDN"

# 判断编号
def get_token_num(self, token, token_type):
    if token_type == "IDN" or token_type == "INT" or token_type == "FLOAT"
    or token_type == "STRING":
        return token
    elif token_type == "KW":
        return TYPE_TO_CONTENT_DICT_KW[token]
    elif token_type == "OP":
        return TYPE_TO_CONTENT_DICT_OP[token]
    elif token_type == "SE":
        return TYPE_TO_CONTENT_DICT_SE[token]

def getnewDfa(self,dicts,dfa):

```

```

self.startId = dicts[dfa.startId]
# print(self.startId)

addSets = [] # 存储加入新的DFA的节点
# 在新的DFA加入节点
for key,value in dicts.items():
    if value not in addSets:
        addSets.append(value)

self.add_node(value,dfa.nodes[key].isFinal,dfa.nodes[key].isBackOff,dfa.nodes[key].tag)
    addEdges = []
    for edge in dfa.edges:
        tup = (dicts[edge.fromNodeId],dicts[edge.toNodeIds],edge.tag)
        if tup not in addEdges:

self.add_edges(dict[edge.fromNodeId],dict[edge.toNodeIds],edge.tag)
        addEdges.append(tup)
# print('addSets: ' + str(addSets))

```

3.4 词法分析器Lexer

Lexer 类的主要功能是通过读取 sql-- 文件生成 token 序列，并进行输出。

支持打印在控制台和输出到文件中两种输出方式。

定义的类：

1. Token

依据本次实验要求输出的格式进行定义。包含三个部分：

- lexeme 待测代码中的单词符号
- tokenType 单词符号种别
- tokenNum 单词符号内容

```

class Token:
    def __init__(self, lexeme: str, token_type: str, token_num: str):
        self.lexeme = lexeme
        self.tokenType = token_type
        # 说是num，其实不是全是num，所以还是用str类型
        self.tokenNum = token_num

```

2. TokenTable

定义了Token的输出方式。

属性：

- tokens 一个保存 token 的数组

方法：

- print_token_table 在控制台按照格式输出token列表
- save_token_table 保存输出到文件中。

```

class TokenTable:

```

```

def __init__(self):
    self.tokens = []

def print_token_table(self):
    for token in self.tokens:
        print("{} <{},{}>".format(token.lexeme, token.tokenType,
token.tokenNum))

def push_token(self, token: Token):
    self.tokens.append(token)

def save_token_table(self, path):
    f = open(path, "w+")
    for token in self.tokens:
        f.write("{} <{},{}>\n".format(token.lexeme, token.tokenType,
token.tokenNum))
    f.close()

```

3. Lexer

执行词法分析的主程序，需要提供一个 `DFA` 作为输出参数。会生成一个 `tokenTable` 类，保存输出的数据。

方法：

- `run` 执行词法分析

```

class Lexer:
    def __init__(self, path: str, token_table: TokenTable, dfa: DFA):
        self.source = open(path, 'r').read()
        self.source += " "
        self.tokenTable = token_table
        self.dfa = dfa

    # 执行词法分析
    def run(self):
        # 流程:
        # token_now = ""
        # 1. 读取字符
        # 2. 查找对应的状态转换
        #     2.1 如果找不到则说明错误的词法，报错
        #     2.2 如果找到了则状态转换到新的状态
        #         2.2.1 非终结态则继续读取
        #         2.2.2 终结态判断is_back_off
        #             2.2.2.1 true: 从token_now退出一个ch,将生成的token_now加入
        # tokenTable
        #             2.2.2.2 false: 将token_now加入token list中
        #         2.2.3 修改DFA的指针指向初始位置, token_now = ""

        # 初始化
        text = self.source
        token_now = ""
        self.dfa.get_start()
        ID = 0
        i = 0

```

```

while i < len(text):
    # 需要跳过的情况
    ch = text[i]
    if token_now == "" and (ch == "\n" or ch == ' '):
        i += 1
        continue

    # GROUP BY和ORDER BY特殊处理
    if token_now == "GROUP" or token_now == "ORDER":
        i += 1
        token_now += " "
        continue

    token_now += ch
    # 匹配成功到下一个节点
    if self.dfa.next_id(ch):
        ID = self.dfa.nowId
        # 判断is_final
        if self.dfa.is_final(ID):
            # 判断is_back_off
            if self.dfa.is_back_off(ID):
                # 指针回退一个
                token_now = token_now[0:-1]
                i -= 1

            # 获得最终节点的tag
            node_tag = self.dfa.get_tag(ID)
            # 这个判断应该是dfa提供的
            token_type = self.dfa.get_token_type(token_now,
node_tag)

            token_num = self.dfa.get_token_num(token_now,
token_type)

            self.tokenTable.push_token(Token(token_now, token_type,
token_num))

            token_now = ""
            self.dfa.get_start()
            i += 1
        # 匹配失败，则抛出异常
    else:
        print("Lexical error: 不符合sql词法!")
        return

    # 如果最后一个词是属于IDNorKWorOP那么也要加入token_list中

if not self.dfa.is_final(ID):
    print("Lexical error: 最终一个词不是完整的token")
    return

```

4 输出格式说明

[待测代码中的单词符号] [TAB] <[单词符号种别],[单词符号内容]>

4.1 控制台中的打印格式

按照要求进行打印：

```
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 #><==>
```

4.2 文件中的保存格式

按照要求进行保存：

```

1  SELECT    <KW,1>
2  from_     <IDN,from_>
3  .         <OP,16>
4  _1_       <IDN,_1_>
5  ,         <SE,3>
6  SUM       <KW,21>
7  (         <SE,1>
8  from_     <IDN,from_>
9  .         <OP,16>
10 _2_       <IDN,_2_>
11 )         <SE,2>
12 FROM      <KW,2>
13 from_     <IDN,from_>
14 JOIN      <KW,14>
15 _1A       <IDN,_1A>
16 ON        <KW,17>
17 from_     <IDN,from_>
18 .         <OP,16>
19 _1_       <IDN,_1_>
20 =         <OP,1>
21 _1A       <IDN,_1A>
22 .         <OP,16>

```

5 源程序编译步骤

本词法分析器使用python实现，因此无需编译。

5.1 引入外部包

需要的外部包：

- `re`：正则表达式相关的包

5.2 根据需要修改main.py文件

- 修改 `path` 为 `sql--` 所在路径。
- 修改 `main()` 函数中的保存路径。

```

# 输出文件的保存路径
lexer.tokenTable.save_token_table("./test.txt")

```

- 直接执行 `main.py` 代码即可。

(二) 测试报告

测试1

测试用例

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

词法分析结果

```
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>
```

测试2

测试用例

```
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 #>=<="
```

词法分析结果

```

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 #><==>
```

测试3

测试用例

```
SELECT websites.name, access_log.count, access_log.date  
FROM websites  
JOIN access_log  
ON websites.id=access_log.site_id  
ORDER BY access_log.count
```

词法分析结果

```

SELECT      <KW,1>
Websites    <IDN,Websites>
.           <OP,16>
name        <IDN,name>
,           <SE,3>
access_log  <IDN,access_log>
.           <OP,16>
count       <IDN,count>
,           <SE,3>
access_log  <IDN,access_log>
.           <OP,16>
date        <IDN,date>
FROM        <KW,2>
Websites    <IDN,Websites>
JOIN        <KW,14>
access_log  <IDN,access_log>
ON          <KW,17>
Websites    <IDN,Websites>
.           <OP,16>
id          <IDN,id>
=           <OP,1>
access_log  <IDN,access_log>
.           <OP,16>
site_id     <IDN,site_id>
ORDER BY    <KW,27>
access_log  <IDN,access_log>
.           <OP,16>
count       <IDN,count>

```

测试4

测试用例

```
SELECT AVG(count) AS CountAverage FROM access_log
```

词法分析结果

```

SELECT    <KW,1>
AVG      <KW,20>
(        <SE,1>
count    <IDN,count>
)        <SE,2>
AS       <KW,4>
CountAverage    <IDN,CountAverage>
FROM      <KW,2>
access_log    <IDN,access_log>

```

测试5

测试用例

```

SELECT LastName,FirstName,Address FROM Persons
WHERE Address IS NOT NULL

```

```

SELECT    <KW,1>
LastName  <IDN,LastName>
,         <SE,3>
FirstName <IDN,FirstName>
,         <SE,3>
Address   <IDN,Address>
FROM      <KW,2>
Persons   <IDN,Persons>
WHERE     <KW,3>
Address   <IDN,Address>
IS        <KW,31>
NOT       <OP,13>
NULL      <KW,32>

```

二、语法分析器设计

(一) 开发报告

实现LL(1)和LR(1)两种语法分析方法

1 LL(1)

1.1 实现思路

1. 构造**first**集。
2. 构造**follow**集。
3. 根据first集和follow集构造**分析表table**。
4. 构造**符号栈v_stack**和**状态栈state_stack**根据分析表对输入串进行语法分析。

1.2 具体实现

1.2.1 数据结构定义

```
VN = [] # 非终结符集
VT = [] # 终结符集
V = [] # 符号集

rules = [] # 文法规则

first = [] # first集
follow = [] # follow集
table = [] # 分析表

test_str = [] # 输入串
state_stack = [] # 状态栈
v_stack = [] # 符号栈
```

1.2.2 函数定义

```
# 读取sql语法文件
def read_sql_syntax()
# 划分终结符和非终结符
def get_v()

# 检测规则是否为A->$的形式
def in_empty_rule(this_vn)

# 初始化、构造及输出first集
def init_first()
def get_first()
def print_first()
# 初始化、构造及输出follow集
def init_follow()
def get_follow()
def print_follow()
```

```
# 初始化、构造及输出table分析表
def init_table()
def get_table()
def print_table(max_vn_index, max_vt_index)
# 计算符号在符号集中的索引(FLAG = 0 为总符号集; FLAG = 1 为终结符集; FLAG = 2 为非终结符集)

def get_v_index(this_v, FLAG)
#计算语法规则的索引
def get_rule_index(this_rule)
```

1.2.3 main函数

1. 依次调用函数 read_sql_syntax() 读取sql语法、get_v() 划分终结符集和非终结符集、get_first() 构造first集、get_follow() 构造follow集、get_table 构造table分析表。
2. 输入分析串，例如：

```
SELECT IDN . IDN FROM IDN WHERE IDN . IDN > INT
```

3. 初始化 test_str 分析串（在输入串后加#并进行预处理）。初始化 v_stack 符号栈，即将 # root 进栈。初始化 state_stack 状态栈，即将 0 进栈。
4. 从左依次遍历 test_str 分析串，每次取 a = test_str[0] 直至 test_str == None:
 - a 不是终极符，终止循环，输入串不符合语法规则，输出 error。
 - a 是终结符，a 等于 v_stack[-1] 且 a 为 #，则输入串符合语法规则，输出 accept。
 - a 是终结符，a 等于 v_stack[-1] 且 a 不为 #，则对 a 进行移入，输出 move。

```
v_stack.pop()
test_str.pop(0)
```

- a 是终结符，a 不等于 v_stack[-1]，则查找分析表 table 进行规约，输出 reduction。注意：所用规约的语法规则 rule 中的 \$ 不需 push 符号栈。

```
rule = []
vt_index = get_v_index(a, 1)
vn_index = get_v_index(v_stack[-1], 2)
rule_index = table[vn_index][vt_index]
rule = rules[rule_index - 1][:]

if a != '#':
    print("{}\t{}\t{}\t{}\t{}\treduction".format(step, rule_index, v_stack[-1],
a), file = result)
else:
    print("{}\t{}\t{}\t{}\t{}\treduction".format(step, rule_index, v_stack[-1]),
file = result)
    step = step + 1

v_stack.pop()
for i in range(len(rule) - 2):
    tmp = rule[len(rule) - i - 1][:]
    if tmp != '$':
        v_stack.append(tmp)
```



```
        如果vn != S break
    如果$在vn的first集中且vn的follow集不为空
        遍历vn的follow集
            对于vn的follow集每个终结符vt, 如果存在规则 vn->$, 则
                table[vn_index][follow_vt_index] = rule_index
```

1.3 分析结果

1.3.1 运行语法分析器

1. 输入为 input 下的词法分析器的输出文件
2. 输出保存在 output 文件夹下, 命名方式为 13Agra.tsv。
3. 运行语法分析器LL1命令:

```
cd syntax-analysis
python LL.py
```

1.3.2 运行结果展示

4. 构造 first 集 (展示部分)

```
-----first-----
root ['UPDATE', 'DELETE', 'SELECT', '(', 'INSERT']
dm1Statement ['UPDATE', 'DELETE', 'SELECT', '(', 'INSERT']
selectStatement ['SELECT', '(']
unionStatements ['$ ', 'UNION']
unionStatement ['UNION']
unionStatementKey ['UNION']
unionStatementQuery ['SELECT', '(']
unionType ['ALL', 'DISTINCT', '$']
querySpecification ['SELECT', '(']
selectClause ['$ ', 'FROM', 'GROUP BY', 'HAVING', 'ORDER BY']
fromClause ['FROM', '$']
groupByClause ['GROUP BY', '$']
havingClause ['HAVING', '$']
orderByClause ['ORDER BY', '$']
selectElements ['*', 'IDN', 'AVG', 'MAX', 'MIN', 'SUM']
selectElementHead ['*', 'IDN', 'AVG', 'MAX', 'MIN', 'SUM']
selectElementListRec [' ', '$']
selectElement ['IDN', 'AVG', 'MAX', 'MIN', 'SUM']
elementNameAlias ['AS', '$ ', 'IDN']
tableSources ['(', 'IDN']
tableSourceListRec [' ', '$']
tableSource ['(', 'IDN']
joinParts ['$ ', 'JOIN', 'LEFT', 'RIGHT']
tableSourceItem ['(', 'IDN']
tableName ['IDN']
uidList ['IDN']
uidListRec [' ', '$']
uid ['IDN']
fullColumnName ['IDN']
dottedId ['.', '$']
```

5. 构造 follow 集 (展示部分)

```

-----follow-----
root ['#']
dmStatement ['#']
selectStatement ['#']
unionStatements ['#']
unionStatement ['$','UNION','#']
unionStatementKey ['SELECT','(']
unionStatementQuery ['$','UNION','#']
unionType ['SELECT','(','*','IDN','AVG','MAX','MIN','SUM']
querySpecification ['$','UNION','#',')']
selectClause ['$','UNION','#',')']
fromClause ['GROUP BY','$','HAVING','ORDER BY','UNION','#',')']
groupByClause ['HAVING','$','ORDER BY','UNION','#',')']
havingClause ['ORDER BY','$','UNION','#',')']
orderByClause ['$','UNION','#',')']
selectElements ['$','FROM','GROUP BY','HAVING','ORDER BY']
selectElementHead [',','$','FROM','GROUP BY','HAVING','ORDER BY']
selectElementListRec ['$','FROM','GROUP BY','HAVING','ORDER BY']
selectElement [',','$','FROM','GROUP BY','HAVING','ORDER BY']
elementNameAlias [',','$','FROM','GROUP BY','HAVING','ORDER BY','JOIN','LEFT','RIGHT','WHERE','UNION','#','SET','ON',')']
tableSources ['WHERE','$','GROUP BY','HAVING','ORDER BY','UNION','#',')']
tableSourceListRec ['WHERE','$','GROUP BY','HAVING','ORDER BY','UNION','#',')']
tableSource [',','$','WHERE','GROUP BY','HAVING','ORDER BY','UNION','#',')']
joinParts [',','$','WHERE','GROUP BY','HAVING','ORDER BY','UNION','#',')']
tableSourceItem ['$','JOIN','LEFT','RIGHT',' ','WHERE','GROUP BY','HAVING','ORDER BY','UNION','#','ON',')']
tableName ['AS','$','IDN','JOIN','LEFT','RIGHT',' ','WHERE','GROUP BY','HAVING','ORDER BY','UNION','#','(','VALUES','V
ALUE','SET','ON',')']
uidList [')']
uidListRec [')']
uid [',','$','FROM','GROUP BY','HAVING','ORDER BY','AS','IDN','JOIN','LEFT','RIGHT','WHERE','UNION','#',' ','SET','(','
VALUES','VALUE','ON',')','=','>','<','<=','>=','!=','<=>','IS','AND','&&','XOR','OR','||']
fullColumnName ['AS','$','IDN',' ','FROM','GROUP BY','HAVING','ORDER BY','=','>','<','<=','>=','!=','<=>','IS','AND',
'&&','XOR','OR','||','UNION','#',')','JOIN','LEFT','RIGHT','WHERE']

```

6. 构造 table 分析表 (展示部分)

TABLE											
VN	\$	UNION	ALL	DISTINCT	SELECT	()	FROM	GROUP BY	HAVING	
root					1	1					
dmlStatement					2	2					
selectStatement					6	6					
unionStatements	8	7									
unionStatement		9									
unionStatementKey		10									
unionStatementQuery					11	11					
unionType	14		12	13	14	14					
querySpecification					15	16					
selectClause	17							17			

7. 输出结果 13Agra.tsv (展示部分)

```
1 1 root#INSERT reduction
2 3 dmlStatement#INSERT reduction
3 112 insertStatement#INSERT reduction
4 115 insertKeyword#INSERT reduction
5 / INSERT#INSERT move
6 116 into#INTO reduction
7 / INTO#INTO move
8 44 tableName#IDN reduction
9 48 uid#IDN reduction
10 / IDN#IDN move
11 113 insertStatementRight#VALUES reduction
12 118 insertStatementValue#VALUES reduction
13 119 insertFormat#VALUES reduction
14 / VALUES#VALUES move
15 / (#( move
16 123 expressionsWithDefaults#INT reduction
17 127 expressionOrDefault#INT reduction
18 58 expression#INT reduction
19 72 predicate#INT reduction
20 75 expressionAtom#INT reduction
21 80 constant#INT reduction
22 84 decimalLiteral#INT reduction
23 / INT#INT move
24 74 predicateRight#, reduction
25 61 expressionRight#, reduction
26 125 expressionOrDefaultListRec#, reduction
27 / ,#, move
28 127 expressionOrDefault#FLOAT reduction
29 58 expression#FLOAT reduction
30 72 predicate#FLOAT reduction
31 75 expressionAtom#FLOAT reduction
32 80 constant#FLOAT reduction
33 83 decimalLiteral#FLOAT reduction
34 / FLOAT#FLOAT move
35 74 predicateRight#, reduction
36 61 expressionRight#, reduction
37 125 expressionOrDefaultListRec#, reduction
38 / ,#, move
39 127 expressionOrDefault#STRING reduction
40 58 expression#STRING reduction
41 72 predicate#STRING reduction
```

```

42 75 expressionAtom#STRING reduction
43 79 constant#STRING reduction
44 97 stringLiteral#STRING reduction
45 / STRING#STRING move
46 74 predicateRight#) reduction
47 61 expressionRight#) reduction
48 126 expressionOrElseDefaultListRec#) reduction
49 / )#) move
50 122 expressionsWithDefaultsListRec# reduction
51 / # accept

```

与提供的0Agra.tsv进行比对，程序运行结果正确！

2 LR(1)

2.1 实现思路

1. 构造**first**集
2. 构造**项目集**items
3. 根据项目集通过闭包运算构造**项目集规范族**
4. 根据项目集规范族构造**action**表和**goto**表
5. 构造**符号栈**v_stack和**状态栈**state_stack根据action表和goto表对输入串进行语法分析。

2.2 具体实现

2.2.1 数据结构定义

```

# 规范项目
class Standard_item:
    def __init__(self, left, right):
        self.left = left # 项目
        self.right = right # 向前搜索字符串

VN = [] # 非终结符集
VT = [] # 终结符集
V = [] # 符号集

rules = [] # 语法规则
items = [] # 项目集
standard_items = [] # 规范项目集

first = [] # first集

action = [] # action表
goto = [] # goto表

test_str = [] # 输入串
state_stack = [] # 状态栈
v_stack = [] # 符号栈

```

2.2.2 函数定义

```
# 读取sql语法文件
def read_sql_syntax()
# 划分终结符和非终结符
def get_v()

# 检测规则是否为A->$的形式
def in_empty_rule(this_vn)
# 初始化、构造及输出first集
def init_first()
def get_first()
def print_first()

# 构造文法项目
def get_items()
# 闭包运算
def get_closure(set)
# 检测规范项目it是否在闭包中
def in_closure(it, closure)
# 得到新的规范项目集
def get_new_item(standard_item, v)
# 构造项目集规范族standard_items
def get_standard_items()
# 判断规范项目集new_item是否在项目集规范族standard_items中
def in_standard_items(new_item)

# 初始化、构造及输出action集
def init_action()
def get_action()
def print_action()
# 初始化、构造及输出goto集
def init_goto()
def get_goto()
def print_goto()

# 计算规范项目集在项目集规范族中的索引
def get_standard_item_index(this_item)
# 计算符号在符号集中的索引 (FLAG = 0 为总符号集; FLAG = 1 为终结符集; FLAG = 2 为非终结符集)
def get_v_index(this_v, FLAG)
# 计算语法规则的索引
def get_rule_index(this_rule)
```

2.2.3 main函数

1. 依次调用函数 `read_sql_syntax()` 读取sql语法、`get_v()` 划分终结符集和非终结符集、`get_items()` 构造文法项目、`get_first()` 构造first集、`get_standard_items()` 构造项目集规范族、`get_action` 构造action分析表、`get_goto` 构造goto分析表。
2. 输入分析串，例如：

```
SELECT IDN . IDN FROM IDN WHERE IDN . IDN > INT
```

3. 初始化 `test_str` 分析串（在输入串后加#并进行预处理）。初始化 `v_stack` 符号栈，即将 `#` `root` 进栈。初始化 `state_stack` 状态栈，即将 `0` 进栈。
4. 从左依次遍历 `test_str` 分析串，每次取 `a = test_str[0]` 直至 `test_str == None`：

- a 不是终结符，终止循环，输入串不符合语法规则，输出 error。
- a 是终结符，获取 a 的终结符索引 a_index，通过 action 分析表得到该执行的动作 act。

```
a_index = get_v_index(a, 1)
act = action[state_stack[-1]][a_index]
```

- 如果 `act == 'acc'`，即则输入串符合语法规则，输出 `accept`。
- 如果 `act[0] == 's'`，即为移进动作，将当前符号 `a` 进符号栈，将 `int(act[1:])` 进状态栈，输出 `move`。

```
test_str.pop(0)
state_stack.append(int(act[1:]))
print("{}\t/\t{}\n{}\tmmove".format(step, v_stack[-1], a), file = resultf)
v_stack.append(a)
step = step + 1
```

- 如果 `act[0] == 'r'`，即为规约动作，先将状态栈前规约后符号数 `len(rules[int(act[1:]) - 1]) - 2` 个状态移出，`$` 不需将状态移出。将符号栈栈顶移出，将规约后的符号 `next_v` 进栈。将 `next_state = goto[state_stack[-1]][v_index]` 进状态栈。

```

if rules[act[1:]) - 1][-1] != '$' or len(rules[int(act[1:]) - 1]) != 3:
    for j in range(len(rules[int(act[1:]) - 1]) - 2):
        state_stack.pop()

v_stack.pop()
next_v = rules[int(act[1:]) - 1][0]
v_stack.append(next_v)
v_index = get_v_index(v_stack[-1], 2)
next_state = goto[state_stack[-1]][v_index]
state_stack.append(next_state)

if a != '#':
    print("{}\t{}\t{}\t{}\tredirection".format(step, int(act[1:]), v_stack[-1],
a), file = result)
else:
    print("{}\t{}\t{}\t{}\tredirection".format(step, int(act[1:]), v_stack[-1]),
file = result)
step = step + 1

```

- 注意通过A->\$规约时, 不需要pop状态栈顶
- 通过规则规约时, pop状态栈顶数 = 规约符号数

2.2.4 get_standard_items()函数

函数调用关系

```
get_standard_items()
  |> get_closure()
  |> in_standard_items()
  |> get_new_item()
      |> get_closure()
          |> in_closure()
```

1. 通过调用 `get_closure()` 函数构造项目集规范族的第一项

```
set = []
set.append(Standard_item(items[0], ['#']))
standard_items.append(get_closure(set))
```

2. 遍历项目集规范族

对于项目集规范族中的每个项目集 `standard_item`，遍历符号集 `v`，调用 `get_new_item(standard_item, v)` 函数获得新的项目集，如果新的项目集不为空，且通过 `in_standard_items(new_item)` 函数判断新的项目集不在项目集规范族中，则将该项目集加入项目集规范族中。

2.2.5 get_action()函数

1. 首先通过 `init_action` 函数初始化action表
2. 遍历项目集规范族，对于项目集规范族中的每个项目集 `standard_item`，遍历 `standard_item` 中的所有项目 `it`。定义 `it_left` 为项目，`it_right` 为展望符号集，`dot_index` 为 `it_left.index('.')`。
 - `it_left[dot_index + 1:] == []`，即项目为“规约”项目
 1. 如果 `$` 在 `it_right` 中，则遍历终结符集 `VT`，`action[item_index][vt_index] == ''` 执行
`action[item_index][vt_index] = 'r' + str(rule_index)`
 2. 如果 `$` 在 `it_right` 中，则只需将 `it_right` 中的终结符，执行 `action[item_index][vt_index] = 'r' + str(rule_index)`
 - `it_left[dot_index + 1:] != []`，即项目为“移进”项目。遍历终结符集 `VT`，如果 `vt == it_left[dot_index + 1]`，则执行 `action[item_index][vt_index] = 's' + str(next_item_index)`
3. `action[1][len(VT)-1] = 'acc'`，即“接受”，`len(VT)-1` 为 `#` 的索引。

2.2.6 get_goto()函数

1. 首先通过 `init_goto` 函数初始化goto表
2. 遍历项目集规范族，对于项目集规范族中的每个项目集 `standard_item`，遍历非终结符集 `VN`，调用 `get_new_item(standard_item, vn)` 函数获得新的项目集，如果新的项目集不为空，且通过 `in_standard_items(new_item)` 函数判断新的项目集不在项目集规范族中，则
`goto[item_index][vn_index] = next_item_index`。

2.3 分析结果

2.3.1 运行语法分析器

1. 输入为 `input` 下的词法分析器的输出文件
2. 输出保存在 `output` 文件夹下，命名方式为 `13Agra.tsv`。
3. 运行语法分析器LL1命令：

```
cd syntax-analysis
python LL.py
```

2.3.2 运行结果展示

1. 构造 action 集 (展示部分)

-ACTION-										
state	\$	UNION	ALL	DISTINCT	SELECT	()	FROM	GROUP BY	HAVING
0					s9	s10				
1										
2										
3										
4		s17								
5										
6										
7										
8										
9			s22	s23						

2. 构造 goto 集 (展示部分)

-GOTO-						
state	root	dmlStatement	selectStatement	unionStatements	unionStatement	unionStatementKey
0	1	2	3			
1						
2						
3						
4				14	15	16
5						
6						
7						
8						
9						

3. 输出结果 13Agra.tsv (展示部分)

```
1  /  ##SELECT  move
2  14 unionType#IDN  reduction
3  /  unionType#IDN  move
4  48 uid#.  reduction
5  /  uid#.  move
6  /  .#IDN  move
7  48 uid#,  reduction
8  52 dottedIdOrStar#,  reduction
9  50 dottedId#,  reduction
10 49 fullColumnName#,  reduction
11 35 elementNameAlias#,  reduction
12 31 selectElement#,  reduction
13 28 selectElementHead#,  reduction
14 /  selectElementHead#,  move
15 /  ,#SUM  move
16 105 function#(  reduction
17 /  function#(  move
18 14 unionType#IDN  reduction
19 /  unionType#IDN  move
20 48 uid#.  reduction
21 /  uid#.  move
22 /  .#IDN  move
23 48 uid#)  reduction
24 52 dottedIdOrStar#)  reduction
25 50 dottedId#)  reduction
26 49 fullColumnName#)  reduction
27 /  fullColumnName#)  move
28 101 aggregateWindowedFunction#FROM  reduction
29 100 functionCall#FROM  reduction
```

```
164 / uid#. move
165 / .#IDN move
166 48 uid#= reduction
167 52 dottedIdOrStar#= reduction
168 50 dottedId#= reduction
169 49 fullColumnName#= reduction
170 76 expressionAtom#= reduction
171 / expressionAtom#= move
172 85 comparisonOperator#STRING reduction
173 / comparisonOperator#STRING move
174 97 stringLiteral# reduction
175 79 constant# reduction
176 75 expressionAtom# reduction
177 74 predicateRight# reduction
178 72 predicate# reduction
179 73 predicateRight# reduction
180 72 predicate# reduction
181 61 expressionRight# reduction
182 58 expression# reduction
183 22 havingClause# reduction
184 25 orderByClause# reduction
185 17 selectClause# reduction
186 15 querySpecification# reduction
187 8 unionStatements# reduction
188 6 selectStatement# reduction
189 2 dmlStatement# reduction
190 1 root# reduction
191 1 root# accept
```

(二) 测试报告

测试1

1.1 输入

词法分析器给出的测试用例为此代码：

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

经过词法分析器的分析后，我们得到了以下Token序列作为语法分析器的输入用例，

```
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>
```

输入文法为：sql_syntax.txt文件

1.2 输出

输出结果为

```
1 1 root#INSERT reduction
2 3 dmlStatement#INSERT reduction
3 112 insertStatement#INSERT reduction
4 115 insertKeyword#INSERT reduction
5 / INSERT#INSERT move
6 116 into#INTO reduction
7 / INTO#INTO move
8 44 tableName#IDN reduction
9 48 uid#IDN reduction
10 / IDN#IDN move
11 113 insertStatementRight#VALUES reduction
12 118 insertStatementValue#VALUES reduction
13 119 insertFormat#VALUES reduction
14 / VALUES#VALUES move
15 / (#( move
16 123 expressionsWithDefaults#INT reduction
17 127 expressionOrDefault#INT reduction
18 58 expression#INT reduction
19 72 predicate#INT reduction
20 75 expressionAtom#INT reduction
21 80 constant#INT reduction
22 84 decimalLiteral#INT reduction
23 / INT#INT move
24 74 predicateRight#, reduction
25 61 expressionRight#, reduction
26 125 expressionOrDefaultListRec#, reduction
27 / ,#, move
28 127 expressionOrDefault#FLOAT reduction
```

```
29 58 expression#FLOAT    reduction
30 72 predicate#FLOAT reduction
31 75 expressionAtom#FLOAT reduction
32 80 constant#FLOAT  reduction
33 83 decimalLiteral#FLOAT reduction
34 /  FLOAT#FLOAT move
35 74 predicateRight#,    reduction
36 61 expressionRight#,    reduction
37 125 expressionOrElseDefaultListRec#,    reduction
38 /  ,#, move
39 127 expressionOrElseDefault#STRING reduction
40 58 expression#STRING    reduction
41 72 predicate#STRING    reduction
42 75 expressionAtom#STRING reduction
43 79 constant#STRING reduction
44 97 stringLiteral#STRING    reduction
45 /  STRING#STRING    move
46 74 predicateRight#)    reduction
47 61 expressionRight#)    reduction
48 126 expressionOrElseDefaultListRec#)    reduction
49 /  )#) move
50 122 expressionsWithDefaultsListRec# reduction
51 /  #    accept
```

展示截图如下：

```

1 1 root#INSERT reduction
2 3 dmlStatement#INSERT reduction
3 112 insertStatement#INSERT reduction
4 115 insertKeyword#INSERT reduction
5 / INSERT#INSERT move
6 116 into#INTO reduction
7 / INTO#INTO move
8 44 tableName#IDN reduction
9 48 uid#IDN reduction
10 / IDN#IDN move
11 113 insertStatementRight#VALUES reduction
12 118 insertStatementValue#VALUES reduction
13 119 insertFormat#VALUES reduction
14 / VALUES#VALUES move
15 / (#( move
16 123 expressionsWithDefaults#INT reduction
17 127 expressionOrDefault#INT reduction
18 58 expression#INT reduction
19 72 predicate#INT reduction
20 75 expressionAtom#INT reduction
21 80 constant#INT reduction
22 84 decimalLiteral#INT reduction
23 / INT#INT move
24 74 predicateRight#, reduction
25 61 expressionRight#, reduction
26 125 expressionOrDefaultListRec#, reduction
27 / ,#, move
28 127 expressionOrDefault#FLOAT reduction
29 58 expression#FLOAT reduction
30 72 predicate#FLOAT reduction
31 75 expressionAtom#FLOAT reduction
32 80 constant#FLOAT reduction

```

测试2

2.1 输入

词法分析器给出的测试用例为此代码：

```

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 #><=="

```

经过词法分析器的分析后，我们得到了以下Token序列作为语法分析器的输入用例，

```

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 #><==>

```

2.2 输出

由于此部分所占篇幅较大，无法在代码块中展示，故用截图代替（展示部分）

```

1 / ##SELECT move
2 14 unionType#IDN reduction
3 / unionType#IDN move
4 48 uid#. reduction
5 / uid#. move
6 / .#IDN move
7 48 uid#, reduction
8 52 dottedIdOrStar#, reduction
9 50 dottedId#, reduction
10 49 fullColumnName#, reduction
11 35 elementNameAlias#, reduction
12 31 selectElement#, reduction
13 28 selectElementHead#, reduction
14 / selectElementHead#, move
15 / ,#SUM move
16 105 function#( reduction
17 / function#( move
18 14 unionType#IDN reduction
19 / unionType#IDN move
20 48 uid#. reduction
21 / uid#. move
22 / .#IDN move
23 48 uid#) reduction
24 52 dottedIdOrStar#) reduction
25 50 dottedId#) reduction
26 49 fullColumnName#) reduction
27 / fullColumnName#) move
28 101 aggregateWindowedFunction#FROM reduction
29 100 functionCall#FROM reduction
30 35 elementNameAlias#FROM reduction
31 32 selectElement#FROM reduction
32 30 selectElementListRec#FROM reduction
33 29 selectElementListRec#FROM reduction
34 26 selectElements#FROM reduction
35 / selectElements#FROM move
36 / FROM#IDN move
37 48 uid#JOIN reduction
38 44 tableName#JOIN reduction
39 35 elementNameAlias#JOIN reduction
40 42 tableSourceItem#JOIN reduction
41 / tableSourceItem#JOIN move
42 / JOIN#IDN move
43 48 uid#ON reduction
44 44 tableName#ON reduction
45 35 elementNameAlias#ON reduction
46 42 tableSourceItem#ON reduction
47 / tableSourceItem#ON move
48 / ON#IDN move
49 48 uid#. reduction
50 / uid#. move

```

```
50 /      uid#.      move
51 /      .#IDN      move
52 48 uid#=      reduction
53 52 dottedIdOrStar#=      reduction
54 50 dottedId#=      reduction
55 49 fullColumnName#=      reduction
56 76 expressionAtom#=      reduction
57 /      expressionAtom#=      move
58 85 comparisonOperator#IDN      reduction
59 /      comparisonOperator#IDN      move
60 48 uid#.      reduction
61 /      uid#.      move
62 /      .#IDN      move
63 48 uid#WHERE      reduction
64 52 dottedIdOrStar#WHERE      reduction
65 50 dottedId#WHERE      reduction
66 49 fullColumnName#WHERE      reduction
67 76 expressionAtom#WHERE      reduction
68 74 predicateRight#WHERE      reduction
69 72 predicate#WHERE      reduction
70 73 predicateRight#WHERE      reduction
71 72 predicate#WHERE      reduction
72 61 expressionRight#WHERE      reduction
73 58 expression#WHERE      reduction
74 108 joinRightPart#WHERE      reduction
75 106 joinPart#WHERE      reduction
76 41 joinParts#WHERE      reduction
77 40 joinParts#WHERE      reduction
78 39 tableSource#WHERE      reduction
```