天津大学

编译器前端大作业开发报告

学院名称 智能与计算学部

专　　业 计算机科学与技术

组　　长 迪丽菲娅

组　　员 格桑曲珍

组　　员 吴柯睿

目录

一、需求分析

本次大作业的要求是编写一个 SQL--语言的编译器前端（包括词法分析器和语法分析器）：

1）使用自动机理论编写词法分析器

2）自上而下或者自下而上的语法分析方法编写语法分析器

1. 词法分析器

使用实现有限自动机确定化，最小化算法类来构造用于识别 token 的 DFA。并用该 DFA 来构造词法分析器。词法分析器的输入为 SQL 语言源代码，输出识别出单词的二元属性。

单词符号的类型包括关键字，标识符，界符，运算符，整数，浮点数，字符串。每种单词符号的具体要求如下：

关键字（KW，不区分大小写）包括：

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

表 1 SQL--关键字

运算符（OP）包括：

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

表 2 SQL--运算符

界符（SE）包括：

类型	语法关键字
界符（3个）	(1) (, (2)), (3) ,

表 3 SQL--界符

标识符（IDN）为字母、数字和下划线（_）组成的不以数字开头的串
整数（INT）、浮点数（FLOAT）的定义与 C 语言相同
字符串（STR）定义与 C 语言相同，使用双引号包含的任意字符串

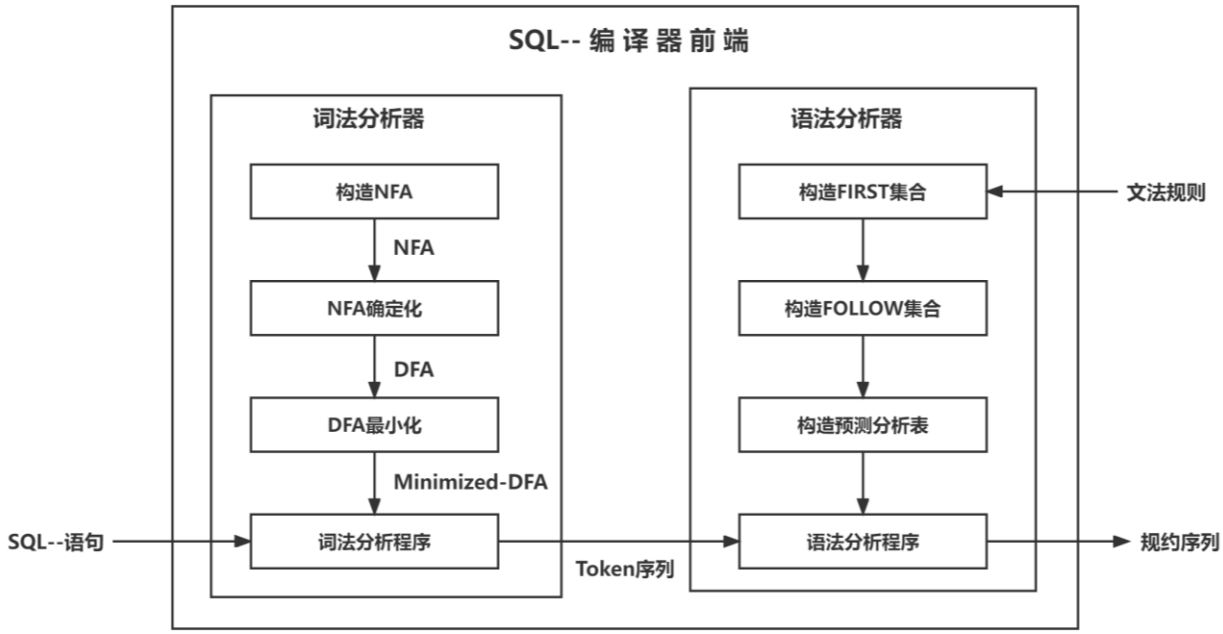
2. 语法分析器

根据给定的文法规则完成 SQL--语言的语法分析器，语法分析器的输入为 SQL--代码的 Token 序列，输出用最左推导或规范规约产生语法树所用的产生式序列。SQL--语言文法须包含以下操作：

① 查询语句；
② 插入语句；
③ 更新语句；
④ 删除语句；
⑤ 表连接操作（JOIN, LEFT JOIN, RIGHT JOIN）
⑥ 聚合操作（MIN，MAX，SUM）
⑦ 组操作（GROUP BY，HAVING，ORDER BY，DISTINCT）
⑧ 集合操作（UNION，UNION ALL）

二、概要设计

在本次实验中我们选择使用 Python 语言来实验 SQL--语言的编译器。通过仔细分析本次实验的需求，我们得到了如下的设计流程图：



整个项目主要由以下几个部分组成，下面给出每个部分的主要任务和工作做流程。

1. 词法分析器(Lexer.py)一共包含四个类，这些类的主要作用如下：

1）Token：存储词法分析生成的每一个 Token

2）NFA：存储 NFA，实现 NFA 的确定化

3）DFA：存储 DFA，实现 DFA 的最小化

4）Lexer：读入输入串和最小化的 DFA ，进行词法分析并输出 Token 序列

2. 语法分析器(Parser.py)包含一个 Parser 类，Parser 类的主要作用如下：

1）读入 LL(1)文法规则，生成 FIRST 和 FOLLOW 集合

2）根据 FIRST 和 FOLLOW 集合构造预测分析表

3）根据预测分析表对词法分析器输出的 Token 序列进行语法分析并输出规约序列

3.编译器前端(main.py) 的主要作用如下：

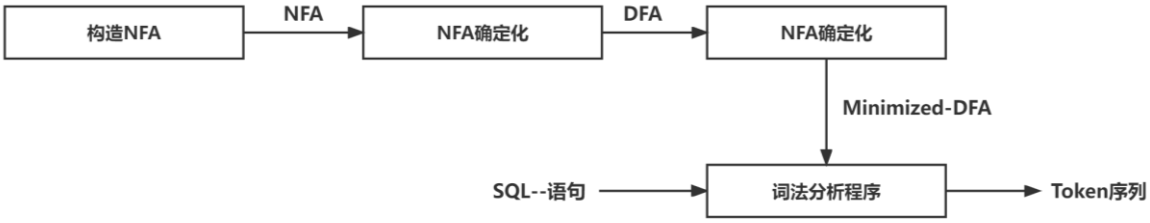
1）调用 Lexer.py 和 Parser.py 中的类来整合实现编译器前端

2）从文本或者终端中获取输入语句后启动词法分析器和语法分析器来完成词法和语法分析工作

三、详细设计

3.1 词法分析器

词法分析器的实现主要包括这几部分工作：NFA 的设计、NFA 的确定化、DFA 的最小化，使用最小化的 DFA 构造词法分析器,使其能读入 SQL--语句并生成对应的 Token 序列。词法分析器的构造流程如下图所示。



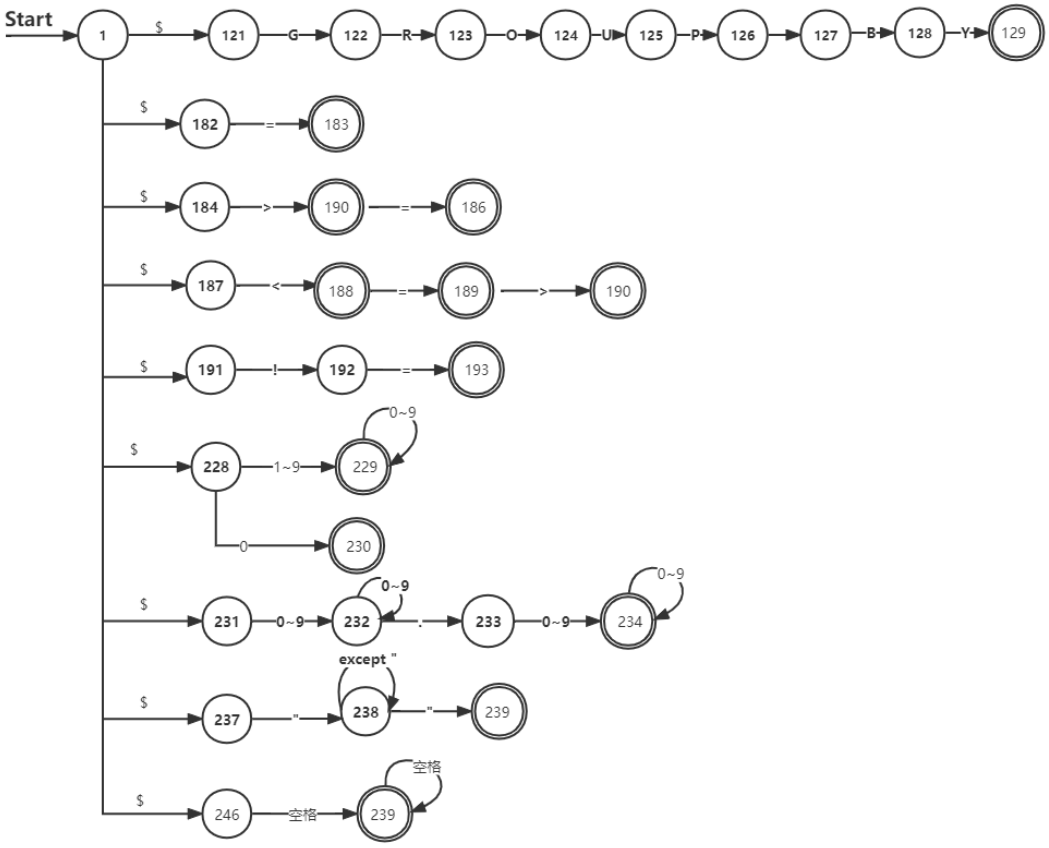
为了实现上图所示的词法分析器，我一共创建了四个类分别为 NFA、DFA、Token 和 Lexer。接下来将会给出每个类中的数据结构和函数负责的功能，函数使用到的算法的介绍以及详细的实现过程。

3.1.1 NFA 的设计

1.构造 NFA 状态转移图

NFA 的构造和设计是词法分析器构建的第一步。首先，我们根据大作业文档中每种单词符号的具体要求，画了出识别这些单词符号的 NFA 的状态转移图。图中给出了我们所构造的 NFA 中识别具有特点的几个单词符号的部分。

下图中给出了 NFA 中识别 GROUP BY 关键字(KW)、部分运算符(OP)、标识符(SE)、整数(INT)、浮点数(FLOAT)及字符串(STRING)的部分，完整的 NFA 的状态转移图在文件中给出。



2. 设计 NFA 类

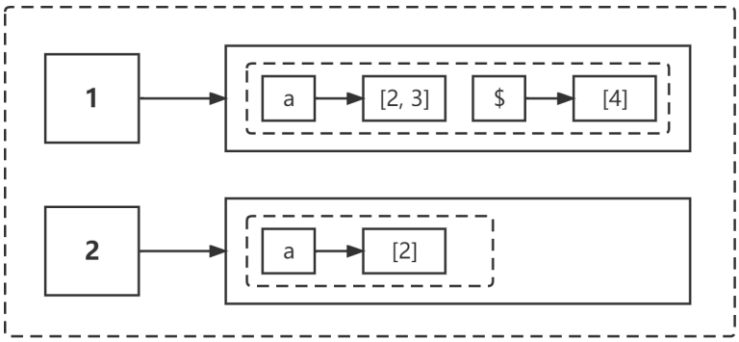
根据 NFA 状态转移图构造的 NFA 类包含以下几个属性：

self.start_state	保存 NFA 的起始状态
self.accept_states	保存 NFA 的接受状态
self.trans_function	保存 NFA 的状态转移函数
self.non_accept_states	保存 NFA 的非接受状态
self.state_num	保存 NFA 的状态数

接下来，我将详细介绍最为核心的两个属性：self.trans_function 和 self.accept_states。

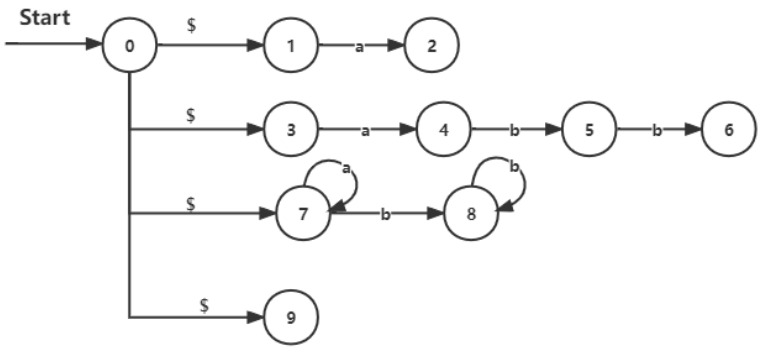
1) self.trans_function

在 NFA 类的所有属性中状态转移函数的存储最为重要。我选择了使用字典和列表构建的数据结构来存储 NFA 的状态转移函数，数据结构如下图所示。字典中的键是结点的序号，而每个键对应的值又为一个字典(在下文中成为子字典)，用来保存该结点在读入不同字符时的转移状态。子字典的键是输入符号，键对应的值是列表类型，用于存储在读入某个字符时能到的所有的状态。



如图中给出了某个 NFA 的 trans_function 的一部分。从图中可以看出，结点 1 在读入字符 a 时可以到达状态 2 和 3，在读入空字符时会到达结点 4；结点 2 在读入字符 B 时会到达本身。

下面给出了如图所示的 NFA 的 trans_function：

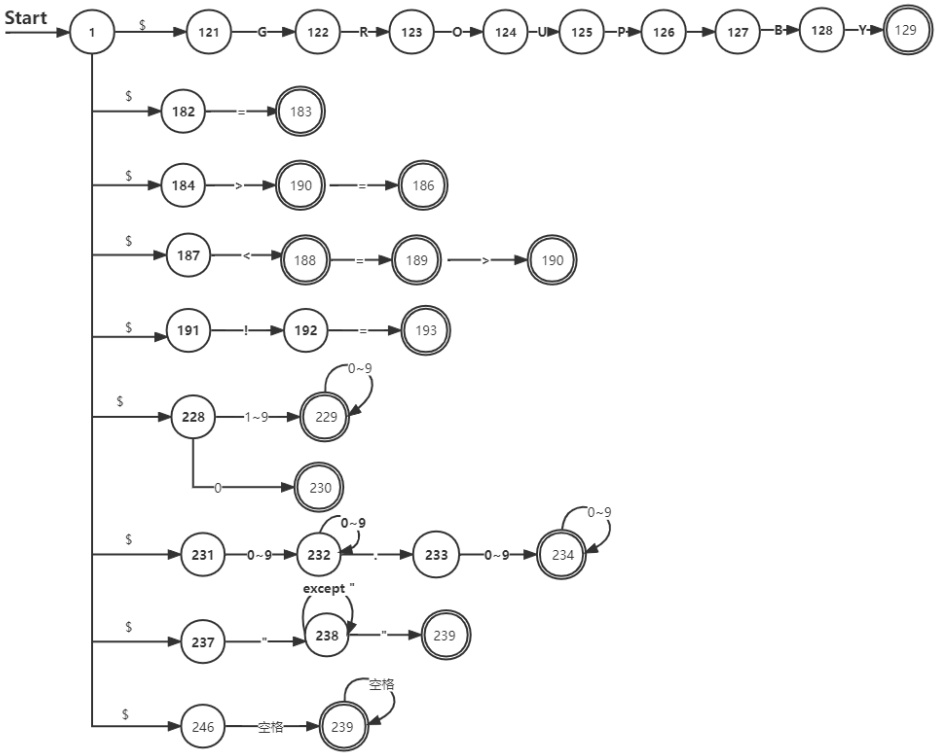



```
fun = {0: {'$': [1, 3, 7, 9]},
       1: {'a': [2]},
       3: {'a': [4]},
       4: {'b': [5]},
       5: {'b': [6]},
       7: {'a': [7], 'b': [8]},
       8: {'b': [8]},
       }
```

2) self.non_accept_states

在考虑 NFA 接受状态的表示时，不仅要考虑接受状态结点号的存储，还要考虑如何记录该接受状态接收的字符。因此，我选择了用字典来存储 NFA 的接

收状态，字典中的键为接受状态的结点号，对应的值为该接受状态接收的字符串的类型和序号，如下图所示。



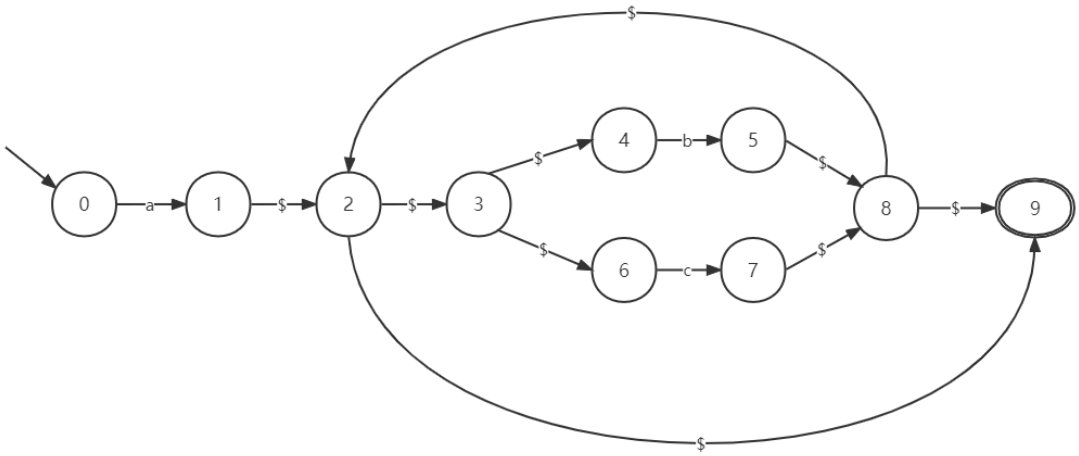
在保存关键字(KW)、运算符(OP)和标识符(SE)时需要记录他们的类型和该类型中此关键字的序号；对于整数(INT)、浮点数(FLOAT)及字符串(STRING)则只需保存它们的类型即可。如，上图所示的 NFA 的接受状态用如下字典保存。

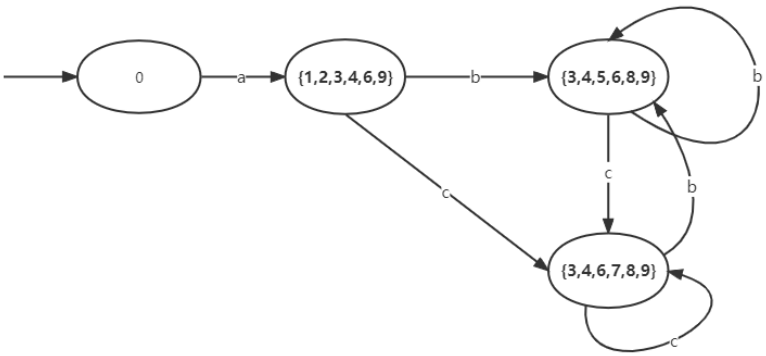
```
nfa_accept = {8:   ['KW', 1],   183: ['OP', 1], 185: ['OP', 2],
              186: ['OP', 4],   188: ['OP', 3], 189: ['OP', 5],
              190: ['OP', 7],   193: ['OP', 6], 204: ['OP', 8],
              223: ['OP', 14],  229: ['INT'],   230: ['INT'],
              234: ['FLOAT'],   236: ['IDN'],   239: ['STRING'],
              }
```

3.3.2 DFA 的设计

DFA 类的属性与 NFA 的类似，其中保存用于保存接收状态和状态转移函数的字典数据结构与 NFA 中定义的一样，唯一不同的是，在 DFA 的这些字典中字

典键不是单个整数，而是包含一系列整数的元组。这样设计的原因是因为 NFA 确定化生成的 DFA 中的每个结点是 NFA 结点的集合。如：





上图中，上面的是 ε-NFA 的状态转移图，下面的是确定化以后的 DFA 的状态转移图，该 DFA 的状态转移函数在类中的保存如下所示：

```
DFA = {(0)                : {'a': (1, 2, 3, 4, 6, 9)},
       (1, 2, 3, 4, 6, 9): {'b': (3, 4, 5, 6, 8, 9)},
       (3, 4, 5, 6, 8, 9): {'c': (3, 4, 6, 7, 8, 9)},
       (1, 2, 3, 4, 6, 9): {'c': (3, 4, 6, 7, 8, 9)},
       (3, 4, 6, 7, 8, 9): {'c': (3, 4, 6, 7, 8, 9),
                            'b': (3, 4, 5, 6, 8, 9)},
       (3, 4, 5, 6, 8, 9): {'b': (3, 4, 5, 6, 8, 9)},
       }
```

3.3.2 NFA 的确定化

1.算法介绍

1）定理

对于字母表 S 上任何一个具有ε-转移的 NFA M, 一定存在一个的 DFA M', 使得：L(M') = L(M)。

2）基本思想

NFA M =(S, S, f, S0, Z') 用构造ε-closure(T) 的方法构造 DFA M'=（S', S, f ', q0 , Z') :

① 首先从 S0 出发，仅经过任意条ε箭弧所能到达的状态所组成的集合 I 作为 M' 的初态 q0.

② 分别把从 q0（对应于 M 的状态子集 I）出发，经过任意 a∈ S 的 a 弧转换 I_a 所组成的集合作为 M' 的状态，其中 I_a = ε-closure(move(I,a)) = ε-closure(J)。如此继续，直到不再有新的状态为止。

2.算法实现

在实现 NFA 的最小化算法时需要计算某些状态集合的 ε 闭包，因此，我编写了 closure(self, states)函数来计算 ε 闭包，并在 NFA 确定化函数 nfa2dfa(self) 中使调用了 closure 函数来实现了ε-NFA 到等价的 DFA 的转换。接下来将详细介绍这两个函数涉及到的数据结构和算法的实现流程。

1) closure(self, states)

该函数的作用是计算出作为参数传入的状态集合 states 的 ε 闭包并返回。该函数中的 states 参数为列表类型，该列表中保存了需要计算 ε 闭包的状态集合的结点号，函数返回的也是存有状态结点的列表 result。函数的实现流程如下：

创建一个空队列 queue 来保存新加入到 result 中的状态

创建一个 result 列表来保存结果，将 states 中的所有状态都加到 result 中

对于 states 中的每一状态 s:

如果在 trans_function 中有该状态读入 ε 后到达另一个节点的记录：

则将 trans_function[s]['$']加入到 result 中和队列去（加到队列中是因为在后续计算中还需要将此新状态读入 ε 后到达的其他状态也加到 result 中）

如果没有：

Continue

当队列不为空时：

如果队头状态已经在 result 中了：

队头元素出队

如果队头状态不在 result 中：

如果在 trans_function 中有该状态读入 ε 后到达另一个节点的记录：

则将 trans_function[s]['$']加入到 result 中和队列去

如果没有：

队头元素出队

返回 result 列表

2）nfa2dfa

该函数的作用是将ε-NFA 转换为等价的 DFA，并返回一个 DFA 类的实例。DFA 类的详细介绍在已在 3.3.2 中给出，该函数的详细介绍如下。

（1）数据结构：

dfa_start_state	保存 DFA 起始状态的元组
dfa_trans_func	保存 DFA 状态转移函数的字典
dfa_accept_state	保存 DFA 接受状态的字典
queue	每当由 NFA 产生一个新的 DFA 状态时，加入到该队列中
visited_states	保存已经构造过的 DFA 结点的集合

（2）实现流程：

令 dfa_start_state 等于利用 closure 算出的 NFA 起始状态的 ε 闭包，并将算出来的闭包加入到队列中

当队列不为空时：

取出队头元素 dfa_state

计算能使队头 DFA 状态中的每一个 NFA 状态发生转移的终结符（除 ε 以外）并保存在 terminal_set 集合中

对于 terminal_set 中的每一个终结符 t：

计算出读入该终结符时 DFA 状态中的每一个 NFA 状态能到达的状态的集合，并调用 closure 函数来计算 ε 闭包后并保存在 trans_state 中

将 dfa_stated 读入 t 时转移到 trans_state 的情况存入 dfa_trans_func 中

如果 trans_state 不在 visited_states 中：

将 trans_state 加入到 queue 和 visited_states 中

// 判断 trans_state 状态是否为 DFA 的接收状态

对于 trans_state 中每一个 NFA 状态 s：

如果 s 是 NFA 的接收状态：

将键值对 trans_state：nfa.accept_states[s]存入 dfa_accept_state

// 如果某一 DFA 状态可能是 IDN，KW 中的一个，应该将

// IDN 的优先级看为最低，即 SELECT 可以是 KW 也可以是 IDN

// 这种情况下应该先将其视为 KW

如果 nfa.accept_states[s][0] != IDN：

Break

3.运行结果

确定化以后的 DFA 的状态转移函数和接收状态的一部分和如下图所示：

```
(68, 236)   {'0': [236], 'p': [236], 'z': [236], 'E': [236], '4': [236], 'e': [236], 'V': [236], 'm': [236],
 'J': [236], 'l': [236], 'F': [236], '8': [236], 'S': [236], 'j': [236], 'H': [236], 'q': [236], 'A': [236],
 'n': [236], 'h': [236], 'f': [236], 'c': [236], 'L': [236], '3': [236], '9': [236], 'K': [236], 'u': [236],
 's': [236], 'G': [236], 'i': [236], 'o': [236], 'M': [236], 'k': [236], 'Y': [236], 'Q': [236], 'X': [236],
 'Z': [236], '2': [236], 'R': [236], 'r': [236], '7': [236], 'C': [236], 'P': [236], 'U': [236], 'x': [236],
 'I': [236], 'b': [236], 'N': [236], 'v': [236], 'd': [236], 'y': [236], '5': [236], '6': [236], 'D': [236],
 '1': [236], 'a': [236], 'w': [236], 'T': [236], '0': [236], '_': [236], 'W': [236], 't': [236], 'B': [236],
 'g': [236]}
(110, 236)  {'0': [236], 'p': [236], 'z': [236], 'E': [236], '4': [236], 'e': [236], 'V': [236], 'm': [236],
 'J': [236], 'l': [236], 'F': [236], '8': [236], 'S': [236], 'j': [236], 'H': [236], 'q': [236], 'A': [236],
 'n': [236], 'h': [236], 'f': [236], 'c': [236], 'L': [236], '3': [236], '9': [236], 'K': [236], 'u': [236],
 's': [236], 'G': [236], 'i': [236], 'o': [236], 'M': [236], 'k': [236], 'Y': [236], 'Q': [236], 'X': [236],
 'Z': [236], '2': [236], 'R': [236], 'r': [236], '7': [236], 'C': [236], 'P': [236], 'U': [236], 'x': [236],
 'I': [236], 'b': [236], 'N': [236], 'v': [236], 'd': [236], 'y': [236], '5': [236], '6': [236], 'D': [236],
 '1': [236], 'a': [236], 'w': [236], 'T': [236], '0': [236], '_': [236], 'W': [236], 't': [236], 'B': [236],
 'g': [236]}
```

```
DFA accept states
(236,)  ['IDN']
(229, 232)  ['INT']
(3, 66, 108, 236)   ['IDN']
(131, 236)  ['IDN']
(183, 186, 189) ['OP', 1]
(188,)  ['OP', 3]
(122, 236)  ['IDN']
(87, 236)   ['IDN']
(178, 219, 236) ['IDN']
(156, 236)  ['IDN']
(93, 147, 209, 236) ['IDN']
(185, 190)  ['OP', 2]
(21, 104, 118, 202, 236)    ['IDN']
(96, 100, 236)  ['IDN']
(215, 236)  ['IDN']
(59, 112, 167, 236) ['IDN']
```

3.3.3 DFA 的最小化

1. 算法介绍

DFA M = （S,∑, δ, s0, St), 最小状态DFA M'

1）构造状态的初始划分∏：终态St 和非终态$S - S_t$ 两组

2）对∏施用传播性原则构造新划分∏ new

3）如∏new == ∏,则令 ∏ $final$ = ∏ 并继续步骤

4)否则∏: = ∏ new重复 2 4.为∏ $final$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 $s0$的那组的代表, M'的终态是含有st 的那组的代表

5）去掉M'中的死状

- 对∏施用传播性原则构造新划分∏new步骤：
- 假设∏被分成 m 个子集∏ = $\{S1, S2, \ldots, Sm\}$, 且属于不同子集的状态是可区别

的，检查∏的每一个子集Si，看是否能够进一步划分。

• 对于某个Si，令$Si = \{s1, s2, \ldots, sk\}$, 若存在数据字符 a 使得$I_a$不全包含在现行∏ 的某一子集$Sj$ 中，则将Si 一分为二：即假定状态$s1, s2$，经过 a 弧分别达到 状态$t1, t2$, 且$t1, t2$属于现行∏的两个不同子集，那么将Si 分成两半，一半含有$s1$: $Si_1 = \{s \mid s \in Si$ 且 s 经 a 弧到达 t1 所在子集中的某状态$\}$；另一半含有$s2$: $Si_2 = Si - Si_1$

• 由于$t1, t2$是可区别的，即存在 w, 使得 t1 读出 w 而停于终态，t2 读不出 w 或 读出 w 却未停于终态。因而 aw 可以将 s1, s2 区分开。也就是说$Si1$和$Si2$中的 状态时可区别的。

• $\prod new = \{S1, S2, \ldots Si_1, Si_2, \ldots, Sm\}$

2. 算法实现

在构造最小化 DFA 时需要使用的算法在算法介绍中给出，但是，在最小化用于识别 token 的 DFA 时的方法会有些许不同。在前面提到的算法中，状态集合的初始化分为终态St 和非终态$S - S_t$ 两组，如果这样划分用于识别 token 的 DFA 会出现 DFA 中用于识别不同 token 的接受状态合并为同一个接受状态，导致当 DFA 读入输入串到达某个接收状态时，只能知道该输入串是合法的 token，而无法判定是哪一种类型的 token。为了解决这个问题，就需要修改 DFA 最小化算法中初始状态的划分，将非接收状态划分为一个集合然后将每一个不同的接受状态划分为一个集合。

为了简化 DFA 最小化计算的过程，我根据由 NFA 确定化得到的 DFA 的一些特性，对状态进行了如下划分：非接受状态集合、标识符接收状态集合和整数接收状态集合。因为，在我们构造的 DFA 中对于其他 token 的接收状态都只有一个，而接收 INT 和 IDN 的状态有多个。将状态划分为以上三个集合以后，就可以对划分好的集合使用前面中提到的算法来进行进一步划分，直到集合不再能划分为止。划分完成后，只需在集合里面加入其余的接收状态集合（加入的集合中

每个集合只包含一个状态)；最后，使用划分好的状态集合和原 DFA 的状态转移函数来构造最小化的 DFA。

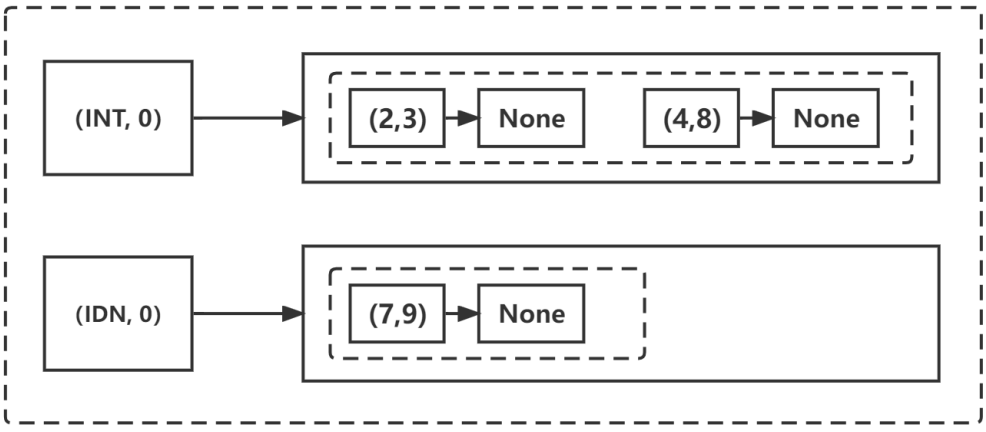
为了完成以上描述的过程，我在 DFA 类中添加了一些新的属性和函数，接下来将详细介绍其中核心的数据结构和函数。

1）initial_set(self, category)

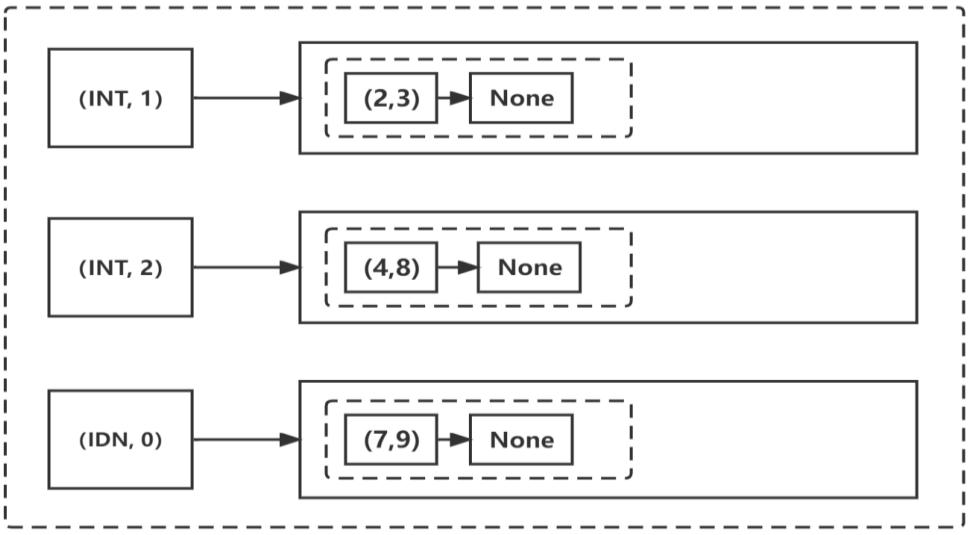
在前面提到了 DFA 最小化算法中的初始划分为：非接受状态集合（用 NON 表示)、标识符接收状态集合（IDN）和整数接收状态集合(INT)。该函数的作用就是根据 category 参数来返回 INT 或者 IDN 的接收状态的集合。只需遍历 DFA 的 accept_states 并从中返回接受状态的动作为 category 的状态集合、返回结果保存在字典中，字典的键为表示状态标号的元组，对应的值为 None。这样设计的原因会在下面其它函数的介绍中给到。

2）self.state_sets

该字典类型的数据结构的作用是保存 DFA 最小化过程中会用到集合划分，数据结构如下图所示。该字典的键为包含状态结点类型和集合序号的二元组，对应的值为保存了该集合中所有状态的字典（下面将成为子字典)。子字典的键为 DFA 状态的元组标号，对应的值为 None。



如图中表示的划分中展示了一个集合划分，集合(INT, 0)包含了两个状态。如果对该集合（使用 split_set 函数）进行划分，处理后的新的集合划分的数据结构如下图所示：



3）split_set(self, set_key)

该函数的作用是对集合名称为 set_key 的状态集合进行划分。如果对指定的集合进行了划分则返回 True，如果指定的集合无需划分则返回 False。函数的实现流程大体如下：

对 state_sets [set_key]中的每一个结点：

根据每个结点对应的转移后状态的类型，对集合中的状态分类并保存在 new_set 中

如果 new_set 中的集合个数大于 1：

说明发生了划分，self.state_sets 中的原集合删去，加入划分后新集合

返回 True

其它：

说明没有发生划分，返回 False

4）dfa_minimization(self)

该函数的作用是使用 initial_set、split_set 和 generate_minimized_dfa 这三个函数来实现 DFA 最小化算法。该函数的大体工作流程为：先调用 initial_set 函数和使用 self.non_accept_state 数据结构来构造最初始的状态集合划分，再使用 split_set 函数来对集合进行再划分直到所有的集合都不能再划分为止，最后

调用 generate_minimized_dfa 函数来根据最终划分好的状态集合 self.state_sets 构造最小化的 DFA。具体实现如下：

对状态进行划分后保存在 self.state 中

令 pre_state_sets_num 为 0，cur_state_sets_num 为集合个数

当 cur_state_sets_num != pre_state_sets_num 时:

对于划分中的每一个集合 set_key：

在 terminal_set 列表中保存该集合中每个状态读入后可以发生转移的所有字符

对于 terminal_set 中的每一个字符 t：

为 state_sets[set_key]中的每个状态保存读入 t 后回到达的状态的类型

调用 split_set 函数，根据当前集合读入 t 时的情况进行集合划分

如果当前集合产生了划分：

回到最开始，重新开始划分

如果当前集合没有划分：

Continue

令 pre_state_sets_num = cur_state_sets_num

令 cur_state_sets_num 为集合个数

将其余的接收状态单状态集合加入到 state_sets 中去

返回 generate_minimized_dfa()函数的结果

5）generate_minimized_dfa(self)

该函数的作用是根据最终的状态划分集合构造新的 DFA（即，最小化后的 DFA），并返回。在 dfa_minimization 函数中求出的 self.state_sets 中的数据会类似如下：

```
{('INT', 1): {(1, 7): None},
 ('IND', 2): {(2, 6): None, (4, 3): None},
 ('NON', 2): {(2,)  : None, (1, 3): None, (2, 3): None}}
```

如图中可以看出一共有三个集合，其中集合('NON', 2)中含有两个状态，每

一个集合中的所有状态都是等价的，此函数的任务就是利用这样的数据结构来构造最小化的 DFA，实现流程如下：

在 state_mapping 字典中保存集合名称（即，最小化 DFA 的状态名称）和原 DFA 状态之间的映射关系

根据该映射关系构造新的起始状态、接收状态和状态转移函数

返回 使用 DFA 类构造的最小化的 DFA

3.3.4 词法分析

1. Token 类的构造

词法分析的运行结果为一连串的 token 序列。我为该大作业设计的 Token 类只包含__init__方法，在__init__含有的属性如下：

lexeme	保存分析到的词素
category	保存单词符号的类型
seq	保存单词符号的序号
keyword	保存单词符号的语法关键字
pos	保存单词符号在整个输入中的位置

如，对于以下输入字符串 SELCET table (只用于展示 token 类，语法并不准确)，词法分析后的产生的 token 为：

Token(SELCET, KW, 1, SELCET, 1)

Token(table, IDN, table, IDN, 8)

2. Lexer 类的构造

Lexer 类的主要作用是读入用户给定的输入串，将其在最小化后的 DFA 上运行，根据 DFA 的读入输入串后的状态类生成 token。为此，Lexer 类包含以下属性：

dfa	保存最小化的 DFA
input_buffer	字符串的输入缓冲区
output_buffer	词素的输出缓冲区
tokens	保存词法分析后产生的 Token 序列

3. 词法分析过程

词法分析的过程可以简单的描述为如下几部：先将输入字符串存入输入缓冲区中，对输入缓冲区里的每一个字符调用 run_on_dfa 函数在 DFA 上运用它，根据 DFA 读入字符串后到达的状态来确定当前输入是否为合法的 token。这些过程在 lexical_analysis 函数中实现，接下来就详细介绍上述提到的两个函数。

1）run_on_dfa(self, char)

该函数是 DFA 类中的一个方法，该函数的作用是对于读入的字符 char 将其在 DFA 上运行，并返回读入后到达的结点和读入前的结点。为了实现该函数，我在 DFA 类中新添加了两个属性 self.cur_state 和 self.pre_state 来保存读入字符前的状态和读入字符后的状态。初始值：self.pre_state 为 None，self.cur_state 为起始状态。该函数需根据输入的字符 char、self.cur_state 和 DFA 的状态转移函数来确定前面提到的两个属性并返回即可，如果读入 char 后到达死结点，用 None 来表示死结点。

2）lexical_analysis(self, text, file_path)

该函数的作用是读入给定的字符串 text，对 text 字符串进行词法分析，把分析后产生的 token 存入 self.tokens 列表中，并将分析后得到的 token 序列按照要求输出到指定文件中去。实现过程如下：

当输入缓冲区不为空的时候：

一个一个的读入输入缓冲区内的字符：

如果 DFA 读入字符后到达的状态为 None：

Break

将读入字符存进输出缓冲区中

如果 cur_state 为 None：

如果 pre_state 为接收状态：

boundary_terminal = [' ', '.', '*', '>', '<', '=', '&', '(', ')', ',', '|', '-']

如果当前读入的字符或者前一个字符在 boundary_terminal 中：

根据 DFA 的状态和输出缓冲区里的内容构造 token，加入 self.tokens 中

删去输入缓冲区中已被分析过的内容

其它：

提示错误信息，返回

如果 pre_state 不是接收状态且前一个读入的字符为空格：

删去输入缓冲区中已被分析过的内容

其它：

提示错误信息，返回

如果 cur_state 为接收状态：

根据 DFA 的状态和输出缓冲区里的内容构造 token，加入 self.tokens 中

删去输入缓冲区中已被分析过的内容

其它：

提示错误信息，返回

将分析后得到的 tokens 按照要求输出到指定的文件中

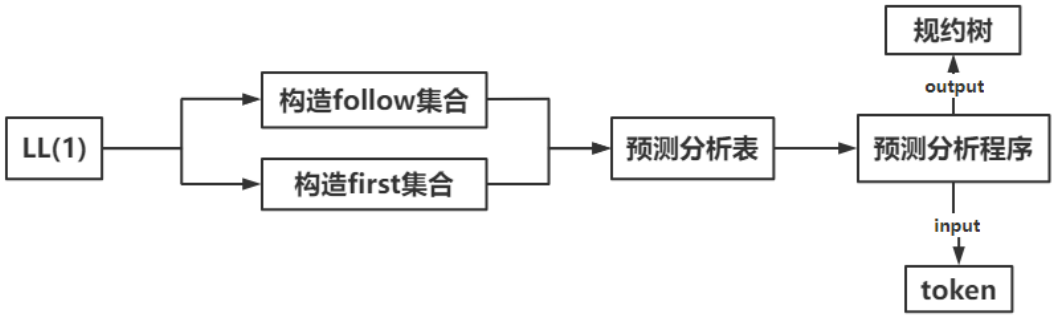
4. 运行结果

图中给出了对于 sql 语句 `SELECT * FROM t WHERE t.a = -1.5` 对应的 Token 序列，由于篇幅原因这里仅给出了一个测试用例，在测试文档中给出了更多且全面的测试实例。

```
SELECT    <KW,1>
*         <KW,5>
FROM      <KW,2>
t         <IDN,t>
WHERE     <KW,3>
t         <IDN,t>
.         <OP,16>
a         <IDN,a>
=         <OP,1>
-         <OP,15>
1.5       <FLOAT,1.5>
```

3.2 语法分析器

语法分析器主要有以下几部分工作：根据在 grammar.txt 中给定的 LL(1)文法来构造 First 和 Follow 集合，利用 First 和 Follow 集合来构造预测分析表(Parsing Table)，最后利用建立好的预测分析表来分析 Lexer 生成的 Token 序列，并输出结果。Parser 的工作流程如下图所示：



为了实现该部分的功能我创建了一个 Parser 类，在接下来的内容里将会详细介绍 Parser 类中每一个部分的设计和实现过程。

3.2.1 文法预处理阶段

在语法分析器构建过程使用到的文法在 grammar.txt 文件中给出，为了在后续的部分中更好地使用这些文法规则，我将它们保存在了列表 rules_table 中，列表结构如下表所示。

joinPart	[JOIN, tableSourceItem , joinRightPart]
joinPart	[joinDirection, JOIN, tableSourceItem, ON, expressionON, expression]
joinRightPart	[ON, expression]

其中每一行代表一个文法规则，对于文法标号为 rule_id 的文法，

rules_table[rule_id][0]表示文法中生成式的左端，rules_table[rule_id][1]表示生成式的右端。该部分的工作由 generate_rules_table 函数来完成。

3.2.2 构造 First 集合

1.算法介绍

对每一个文法符号$X \in \mathbf{V_T} \cup \mathbf{V_N}$构造FIRXT(X): 应用下列规则,直到每个集合$FIRST$不再增大为止.

(1)如果$X \in \mathbf{V_T}$,则$FIRST(X) = \{X\}$

(2)如果$X \in \mathbf{V_N}$,且有产生式$X \to a$ ...,则把 a 加入到$FIRST(X)$中;若$X \to \varepsilon$也是一个 产生式, 则把ε加入到$FIRST(X)$中.

(3)如果$X \to Y$ ...是一个产生式且$Y \in \mathbf{V_N}$, 则把$FIRST(Y)\backslash\{\varepsilon\}$加到$FIRST(X)$中; 如果$X \to \mathbf{Y_1Y_2}$ ... $\mathbf{Y_K}$是一个产生式, $\mathbf{Y_1}, \ldots, Y_{i-1} \in N_N$,而且对任何$j, j \in [1, i-1] \varepsilon \in FIRST$ (Y_j), (即$Y_1Y_2 \ldots Y_{i-1} => \varepsilon$), 则把$FIRST(Y_i)\backslash\{\varepsilon\}$加到$FIRST(X)$中; 特别是, 若所有的$FIRST(Y_j)$均含有$\varepsilon$, $j = 1, 2 \cdots k$, 则把ε加到$FIRST(X)$中

2.算法实现

为了实现上述描述的算法我在 Parser 类中添加了三个数据结构和三个函数，每个数据结构的和函数的简要介绍如下。

（1）数据结构：

self.terminals	集合	保存文法中的终结符
self.pre_first_dict	字典	保存用于计算 FIRST 集合的文法

self.first_set	字典	保存 FIRST 集合计算的结果

（2）函数：

① generate_pre_first_dict(self)

② first(self, non_terminal)

③ generate_first_set(self)

接下来，将会详细给出每个函数与数据结构的作用和实现方式等。

1）generate_pre_first_dict 函数

在构造某个非终结符的 FIRST 集合时，首先会需要用到该非终结符在出现在生成式左端的所有生成式。但是，目前用于保存文法规则的数据结构并不能很好的应用与这种情况，因此，我重新构建了一个新的数据结构来保存文法从而提高 FIRST 集合计算的效率。

该函数的作用是使用 self.rules_table 保存的内容来构造一个便利与计算 FIRST 集合的新的数据结构 pre_first_dict 来保存文法，该字典的结构如下图所示。

```
dict = {'root'              : [0],
        'dmlStatement'      : [1, 2, 3, 4],
        'selectStatement'   : [5],
        'unionStatements'   : [6, 7]}
```

该字典中的键为非终结符，每一个键对应一个 list 数据结构，其中保存了该非终结符出现在生成式左端的生成式在 rules_table 中的下标。如：dmlStatement 对应的列表中保存了 1、2、3、4，表明了 dmlStatement 作为生成式左端的生成式在 rules_table 中的下标为 1、2、3、4。

2）first(self, non_terminal)函数

该函数的作用是计算传入的单个非终结符 non_terminal 的 FIRST 集合并保存在 self.first_set 中，self.first_set 字典的结构如下图：

```
first = {'querySpecification'   : ['SELECT', '('],
         'selectStatement'      : ['SELECT', '('],
         'dmlStatement'         : ['SELECT', '(', 'INSERT', 'UPDATE', 'DELETE']}
```

self.first_set 的键为非终结符，每一个键对应一个 list，该列表中就保存了 FIRST 集合中的所有终结符。如图中给出了 querySpecification、selectStatement 和 amlStatement 的 FIRST 集合。

计算 FIRST 集合的过程如下，即为算法的代码实现：

对于 pre_first_dict[non_terminal]中保存的每个生成式：
    如果该生成式右端的第一个符号为非终结符：
        将该终结符添加到 self.first_set[non_terminal]中
    如果该生成式右端不以非终结符开头：
        对于生成式右端的每一个符号 symbol：
            如果当前符号 symbol 与 non_terminal 相同：
                则跳过该符号
            如果当前符号 symbol 为终结符：
                将该终结符添加到 self.first_set[non_terminal]中并退出
            如果当前符号为非终结符且该符号的 FIRST 集合还没有求得：
                调用 first(symbol)函数计算该符号的 FIRST 集合（递归求解）
            如果当前符号为非终结符且该符号的 FIRST 集合已经求得：
                如果 FIRST 集合中包含ε或者该非终结符为生成式右端的最后一个符号：
                    将 self.first_set[symbol]添加到 self.first_set[non_terminal]中后返回
                如果 FIRST 集合中不含ε或且该非终结符不是生成式右端的最后一个符号：
                    将 self.first_set[symbol]去掉ε后添加到 self.first_set[non_terminal]中，并返回

3）generate_first_set(self)

该函数的作用是求出文法中包含的所有非终结符的 FIRST 集合。因此，该函数会对于 pre_first_dict 中的每个键 nt（即文法中的所有非终结符）进行如下操

作：

如果该终结符不在 self.first_set 中：

则调用 first(nt)来求 FIRST 集合

如果该终结符在 self.first_set 中：

跳过该非终结符

3.运行结果

求得的 FRIST 集合如下图所示，由于篇幅原因这里只给出了一部分结果，完整的结果在文件中给出。

```
+---------------------------------------------------------------------+
|--------------------------- First Set ---------------------------|
+---------------------------------------------------------------------+
querySpecification=[SELECT, (],
selectStatement=[SELECT, (],
dmlStatement=[SELECT, (, INSERT, UPDATE, DELETE],
insertKeyword=[INSERT],
insertStatement=[INSERT],
updateStatement=[UPDATE],
deleteStatement=[DELETE],
root=[SELECT, (, INSERT, UPDATE, DELETE],
unionStatementKey=[UNION],
unionStatement=[UNION],
unionStatements=[UNION, $],
unionStatementQuery=[SELECT, (],
unionType=[ALL, DISTINCT, $],
```

3.2.3 构造 follow 集合

1.算法介绍

对文法 G 的每个非终结符 A 构造 FOLLOW(A)的办法是：连续应用下列规则，直到每个后随符号集 FOLLOW 不再增大为止.

① 对于文法的开始符号 S，置#于 FOLLOW(S)中;

② 若$A \to \alpha B \beta$是一个产生式，则把 FIRST(β)\{ε}加至 FOLLOW(B)中;

③ 若$A \to \alpha B$是一个产生式，或$A \to \alpha B \beta$是一个产生式而$\beta => \varepsilon$（即$\varepsilon \in$ FIRST(β))，则把 FOLLOW(A)加至 FOLLOW(B)中

2.算法实现

为了实现上述描述的算法我在 Parser 类中添加了两个数据结构和三个函数，每个数据结构的和函数的简要介绍如下。

（1）数据结构：

self. pre_follow_dict	字典	保存用于计算 FOLLOW 集合的文法
self.first_set	字典	保存 FOLLOW 集合计算的结果

（2）函数：

① generate_pre_follow_dict (self)

② generate_follow_set (self)

③ first_of_symbols (self, rhs_list)

接下来，将会详细给出每个函数与数据结构的作用和实现方式等。

1）generate_pre_follow_dict (self)

在求某个非终结符的 FOLLOW 集合时，首先会需要用到该非终结符在出现在生成式右端的所有生成式。因此，该函数的作用是使用 self.rules_table 来构造一个便利与计算 FOLLOW 集合的新的数据结构 pre_follow_dict 来保存文法，该字典的结构如下图所示。

```
dict = {'unionStatements'    : [5, 6],
        'unionType'          : [9, 14, 100],
        'querySpecification': [5, 10, 15],
        'selectClause'       : [14]}
```

该字典中的键为非终结符，每一个键对应一个 list 数据结构，其中保存了该非终结符出现在生成式右端的生成式在 rules_table 中的下标。如：unionType 对应的列表中保存了 9、14、100，表明了 unionType 出现在生成式右端的生成式在 rules_table 中的下标为 9、14、100。

2）first_of_rhs(self, rhs_list)

从算法描述中可以看出，在求某一个非终结符的 FOLLOW 集合时往往需要求得在产生式右端紧挨着该非终结符的其它终结符的 FIRST 集合。如对于如下文法，如果想要求得 A 的 FOLLOW 集合，则需要求出 FRIST(CB)。

S -> A C B	B -> g
S -> C b B	A -> B C
S -> B a	A -> d a
B -> $	C -> h
C -> $	

而在 3.2.1 中求得的 self.first_set 只包含了单个符号的 FIRST 集合，因此需要构造另外一个函数来求得一连串符号的 FIRST 集合，first_of_symbols 函数的作用就是求指定的符号序列的 FIRST 集合并返回结果，结果保存在 result 列表中。算法描述如下：

对文法 G 的任何符号串$\alpha = X_1X_2 \dots X_n$构造集合$FIRST(\alpha)$

(1)首先置对文法$FIRST(\alpha) = FIRST(X_1)\backslash\{\varepsilon\}$.

(2)如果对任何$j, j \in [1, i-1] \varepsilon \in FIRST(X_j)$，则把$FIRST(X_i)\backslash\{\varepsilon\}$加入到$FIRST(\alpha)$中.特别是,若所有的$FIRST(X_i)$均含有$\varepsilon, j = 1, 2, \ldots, n$，则把$\varepsilon$加到$FIRST(\alpha)$中

该函数的参数 rhs_list 是含有符号的列表，函数的算法的实现流程如下。对于 rhs_list 中的每一个符号 s 都有：

如果 s 是终结符：

将终结符添加到 result 列表中，并返回 result。

如果 s 是为非终结符且 FIRST(s)中不含有 ε：

将 FIRST(s)的值添加到 result 列表中，并返回 result。

如果 s 是为非终结符并且 FIRST(s)中含有 ε 且该非终结符不是 rhs_list 中的最后一个：

将 FIRST(s)的去掉 ε 后的值添加到 result 列表中，并返回 result。

其余情况：

将 FIRST(s)的值添加到 result 列表中。

3）generate_follow_set(self)

该函数的目的是求得文法中所包含的所有非终结符的 FOLLOW 集合，计算后得到的 FOLLOW 集合保存在 self.follow_set 中。self. follow_set 字典的结构如下图：

```
dict = {'root'              : ['#'],
        'dmlStatement'      : ['#'],
        'selectStatement'   : ['#'],
        'unionStatements'   : ['#'],
        'unionStatement'    :['UNION', '#'],
        'unionStatementKey' :['SELECT', '('],
        'unionType'         :['*', 'IDN', 'AVG', 'MAX', 'MIN', 'SUM', 'SELECT', '(']}
```

self.follow_set 的键为非终结符，每一个键对应一个 list，该列表中就保存了该非终结符的 FOLLOW 集合。如图中给出了 root 和 dmlStatement 等非终结符的 FOLLOW 集合。根据算法描述求得 self.follow_set 过程如下。

令起始节点的 self.follow_set[self.start_symbol] 设为 #

对 self.pre_follow_dict 中的每个符号 symbol 进行如下操作：

对 self.pre_follow_dict[symbol]中的每一个生成式：

调用 first_of_symbols 求得生成式右端 symbol 之后符号的 FRIST 集合 first_of_rhs:

如果 first_of_rhs 中没有 ε 且 symbol 不是生成式右端的最后一个符号：

self.follow_set[symbol]中添加 first_of_rhs 中的元素

如果 first_of_rhs 中有 ε 或者 symbol 是生成式右端的最后一个符号：

// 说明需要将 symbol 右端的符号的 FIRST 集合以外产生式左端的非终

// 结符的 FOLLOW 也加到 self.follow_set[symbol]中去

self.follow_set[symbol]中添加 first_of_rhs 去掉 ε 后的元素

// 此时左端的非终符的 FOLLOW 还没有求出来，因此将 symbol 以及左端的

// 非终结符保存下来，最后再重新计算

用字典 temp 将 symbol 和产生式左端的非终结符记录下来

对 temp 中的每一个 symbol 进行如下操作：

对 temp [symbol]中的每一个非终结符 nt：

将 nt 的 FOLLOW 集合加到 symbol 的 FOLLOW 集合中去

3.运行结果

求得的 FOLLOW 集合如下图所示，由于篇幅原因这里只给出了一部分结果，完整的结果在文件中给出。

```
+-------------------------------------------------------------------------+
|--------------------------- Follow Set ----------------------------|
+-------------------------------------------------------------------------+
root=[#],
dmlStatement=[#],
selectStatement=[#],
unionStatements=[#],
unionStatement=[UNION, #],
unionStatementKey=[SELECT, (],
unionStatementQuery=[UNION, #],
unionType=[*, IDN, AVG, MAX, MIN, SUM, SELECT, (],
querySpecification=[UNION, ), #],
selectClause=[UNION, ), #],
fromClause=[GROUP BY, HAVING, ORDER BY, UNION, ), #],
groupByClause=[HAVING, ORDER BY, UNION, ), #],
havingClause=[ORDER BY, UNION, ), #],
orderByClause=[UNION, ), #],
```

3.2.4 构造预测分析表

1.算法介绍

构造分析表 M 的算法如下：

① 对文法 G 的每个产生式$A \to \alpha$，执行第 ② 和 ③ 步；

② 对每个终结符$a \in FIRST(\alpha)$，把$A \to \alpha$加入$M[A, a]$中；

③ 若$\varepsilon \in FIRST(\alpha)$，则对任何$b \in FOLLOW(A)$，把$A \to \varepsilon$加入$M[A, b]$中；

④ 把所有无定义的$M[A, a]$标上“出错标志”

2.算法实现

为了构造预测分析表我在 Lexer 类中添加了 generate_parsing_table 函数和 parsing_table 字典，分别用来构造预测分析表和保存预测分析表，接下里将会详细介绍预测分析表的数据结构和构造方式。

1）self.parsing_table

self.parsing_table 为字典类型的数据结构，该字典的键为二元元组，其中元组的一个元素表示非终结符，第二元素表示读入的终结符，对于这样的二元组字典中保存的值为该情况需要使用的产生式在 self.rules_table 中的下标，如下图所示。

```
dict = {('root', 'SELECT')          : 0,
        ('root', '(')               : 0,
        ('root', 'INSERT')          : 0,
        ('root', 'UPDATE')          : 0,
        ('root', 'DELETE')          : 0,
        ('dmlStatement', 'SELECT')  : 1,
        ('dmlStatement', '(')       : 1,
        ('dmlStatement', 'INSERT')  : 2}
```

如图中展示的 self.rules_table 中表示在栈顶符号为 dmlStatement 时读入 SELECT 则使用 1 号文法进行解析。

2）generate_parsing_table(self)

预测分析表的构造过程如下：

对于 rules_table 中的每一个产生式$A \to \alpha$（序号为 seq_num）：

对于产生式右端 FIRST 集合中(即调用 first_of_symbols(α))的每一个终结符 t

如果该终结符为 ε ：

对于 FOLLOW(A)中的每一个元素 s：

在 parsing_table 中添加键值对 (A ,s) : seq_num

如果该终结符不为 ε ：

在 parsing_table 中添加键值对 (A ,t) : seq_num

3.运行结果

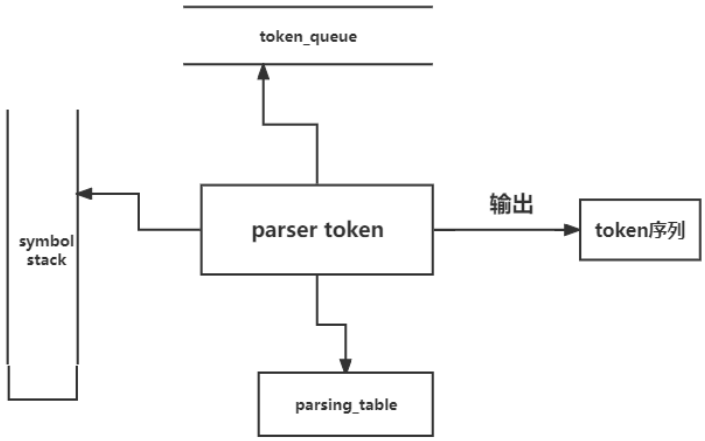
调用 generate_parsing_table 后生成的 parsing_table 的输出如下，为了更直观的观察到要使用哪个产生式，我在输出添加了下标对应的产生式，如下图所示：

```
('root', 'SELECT')  0 root -> dmlStatement
('root', '(')   0 root -> dmlStatement
('root', 'INSERT')  0 root -> dmlStatement
('root', 'UPDATE')  0 root -> dmlStatement
('root', 'DELETE')  0 root -> dmlStatement
('dmlStatement', 'SELECT')  1 dmlStatement -> selectStatement
('dmlStatement', '(')   1 dmlStatement -> selectStatement
('dmlStatement', 'INSERT')  2 dmlStatement -> insertStatement
('dmlStatement', 'UPDATE')  3 dmlStatement -> updateStatement
('dmlStatement', 'DELETE')  4 dmlStatement -> deleteStatement
('selectStatement', 'SELECT')   5 selectStatement -> querySpecification unionStatements
('selectStatement', '(')    5 selectStatement -> querySpecification unionStatements
('unionStatements', 'UNION')    6 unionStatements -> unionStatement unionStatements
('unionStatements', '#')    7 unionStatements -> $
('unionStatement', 'UNION') 8 unionStatement -> unionStatementKey unionStatementQuery
('unionStatementKey', 'UNION')  9 unionStatementKey -> UNION unionType
('unionStatementQuery', 'SELECT')   10 unionStatementQuery -> querySpecification
```

3.2.5 构造语法分析器

1.算法介绍

语法分析的核心为预测分析函数，该程函数会读入 Lexer 产生的 Token 序列，并将它存入输入队列中。预测分析函数 parse_token 会根据当前的 Token 队列的队头符号和符号栈的栈顶符号，通过查询预测分析表来确定下一步使用的产生式，工作过程如下图所示。



图中给出的 parsing_table 是在 3.2.3 中构造的预测分析表，而 token_queue 和 symbol_stack 则是为了构造预测分析器而添加的新的数据结构。

2.算法实现

图中的 parse_token 函数的工作流程如下：

将 Lexer 给出的 token 序列存入 self. token_queue 中，并在最后追加 # 符号

将 # 符号和起始符号压栈

当 self. token_queue 不为空时：

如果栈顶符号与队头符号相同：

self. token_queue 的队头符号出队，self. symbol_stack 栈顶元素弹栈

如果元组(栈顶符号，队头符号)在 parsing_table 中：

对应产生式右端如果为 ε：

只需栈顶元素弹栈

对应产生式右端如果不为 ε：

栈顶元素弹栈，再将产生式右端的符号倒序压栈

如果元组(栈顶符号，队头符号)不在 parsing_table 中：

输出错误提示信息，并返回

将规约序列存入指定的文件中

3.运行结果

图中给出了对于 sql 语句 `SELECT *` 对应的 Token 序列 `SELECT *` 的规约序列，由于篇幅原因这里仅给出了一个测试用例。在测试文档中给出了更多且全面的测试实例。

```
SELECT *
1   1   root#SELECT reduction
2   2   dmlStatement#SELECT reduction
3   6   selectStatement#SELECT  reduction
4   15  querySpecification#SELECT   reduction
5   /   SELECT#SELECT   move
6   14  unionType#* reduction
7   26  selectElements#*    reduction
8   27  selectElementHead#* reduction
9   /   *#* move
10  30  selectElementListRec##  reduction
11  17  selectClause##  reduction
12  19  fromClause##    reduction
13  21  groupByClause## reduction
14  23  havingClause##  reduction
15  25  orderByClause## reduction
16  8   unionStatements##   reduction
17  /   ### move
Accept!
```

四、使用说明

下载代码后运行 main.py 函数，此编译器前端提供了两种输入 SQL 语句的方法，一种是从文件中读取，另一种是从终端读取。在运行 main.py 文件后输出以下内容：

```
Welcome to SQL-- Compiler Front-end!
    1. Read the SQL statement from the file
    2. Input SQL statement from terminal
>
```

根据提示输入 1 或 2 来选择对应的输入方式。

1）从文件中读取

在测试前需要将待测的文件存入 Compiler_Front-end_Project\Test\Input 文件夹中，根据提示输入文件名后就可以进行词法和语法分析。分析后的结果会存入 Compiler_Front-end_Project\Test\Output 文件夹中。

2）从终端读取

选择 2 后，只需在终端输入 SQL 语句就可以进行词法和语法分析。分析后的结果会存入 Compiler_Front-end_Project\Test\Output 文件夹中。

```
Welcome to SQL-- Compiler Front-end!
    1. Read the SQL statement from the file
    2. Input SQL statement from terminal
> 2

sql > SELECT * FROM t WHERE t.a = -1.5
```

五、任务分工

学号	姓名	任务
3019244005	迪丽菲娅	词法分析器和语法分析器的实现、开发报告的撰写
3019244018	格桑曲珍	NFA 状态转移图的设计、测试报告撰写
3019244365	吴柯睿	NFA 状态转移图的设计、测试报告撰写