|  |
| --- |
| 说明: 天津大学标准校徽**说明: 校名** |

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 3](#_Toc102935978)

[1. 词法分析器 3](#_Toc102935979)

[2. 语法分析器 4](#_Toc102935980)

[二、概要设计 5](#_Toc102935981)

[三、详细设计 7](#_Toc102935982)

[3.1 词法分析器 7](#_Toc102935983)

[3.1.1 NFA的设计 7](#_Toc102935984)

[3.3.2 DFA的设计 10](#_Toc102935985)

[3.3.2 NFA的确定化 12](#_Toc102935986)

[3.3.3 DFA的最小化 15](#_Toc102935987)

[3.3.4 词法分析 20](#_Toc102935988)

[3.2 语法分析器 24](#_Toc102935989)

[3.2.1 文法预处理阶段 24](#_Toc102935990)

[3.2.2 构造First集合 25](#_Toc102935991)

[3.2.3 构造follow集合 29](#_Toc102935992)

[3.2.4 构造预测分析表 33](#_Toc102935993)

[3.2.5 构造语法分析器 35](#_Toc102935994)

[四、使用说明 37](#_Toc102935995)

[五、任务分工 38](#_Toc102935996)

# 一、需求分析

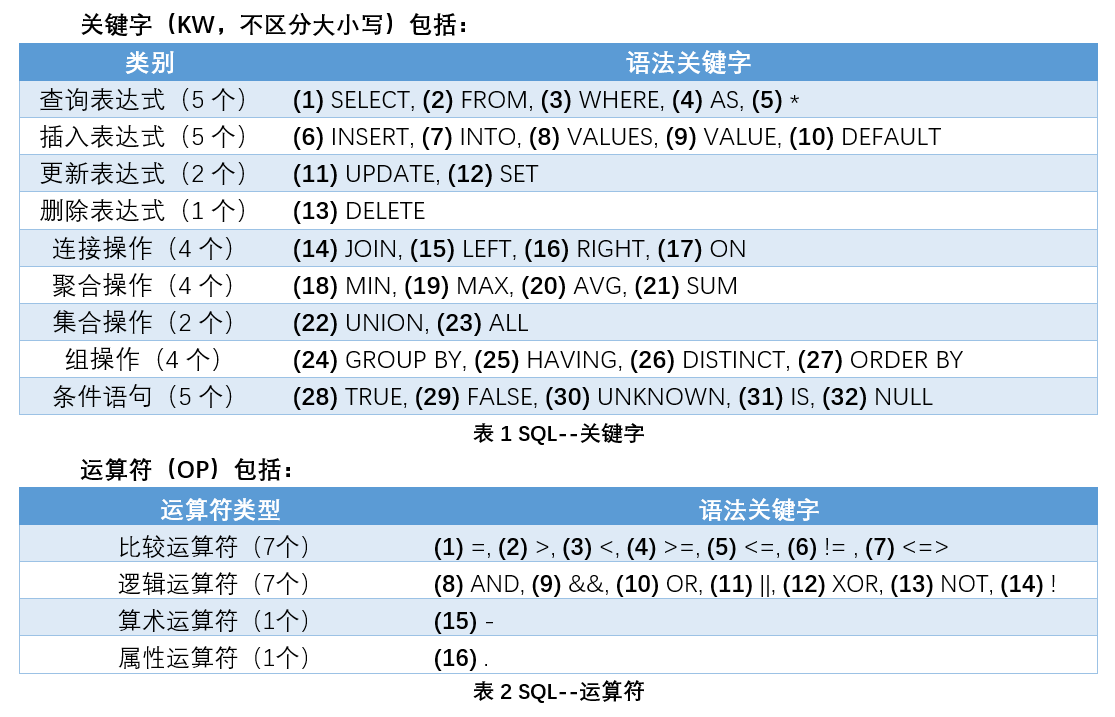
**本次大作业的要求是编写一个SQL--语言的编译器前端（包括词法分析器和语法分析器）：**

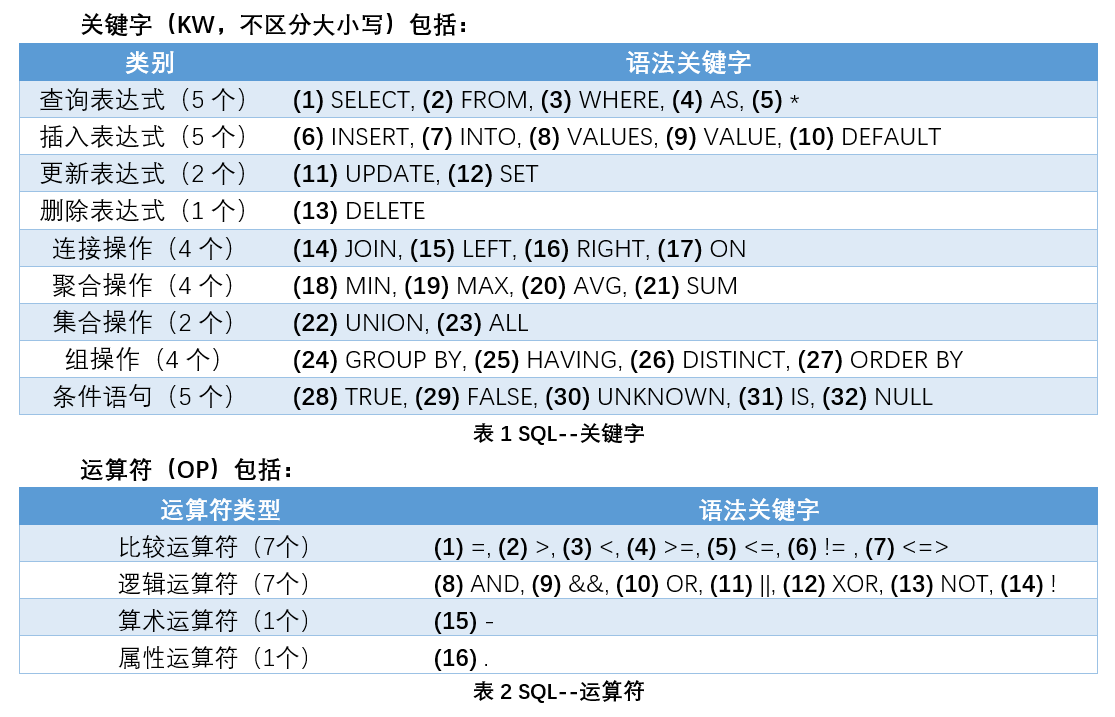
**1）使用自动机理论编写词法分析器**

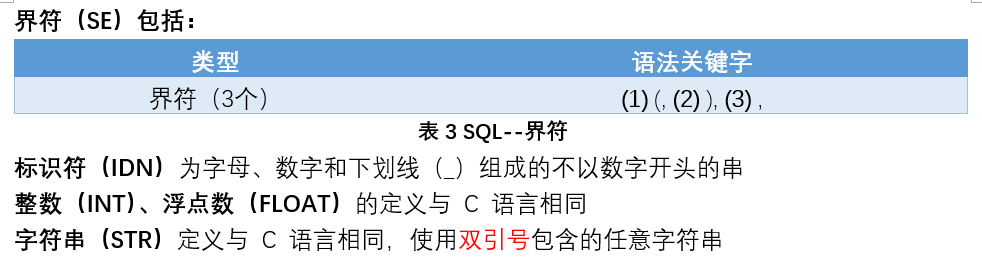
**2）自上而下或者自下而上的语法分析方法编写语法分析器**

## 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：**

****



## 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；**

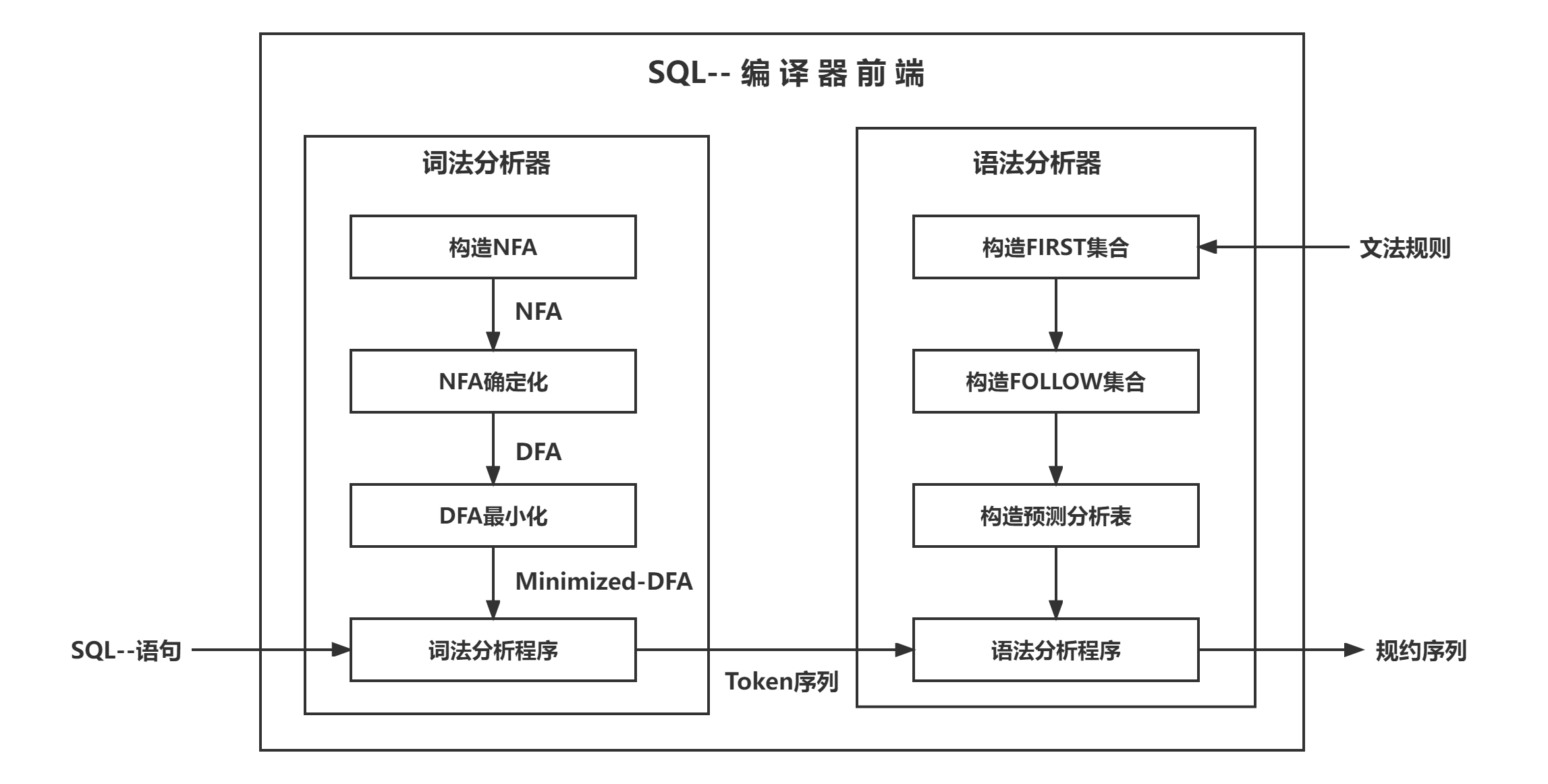
**⑤ 表连接操作（JOIN, LEFT JOIN, RIGHT JOIN）**

**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**

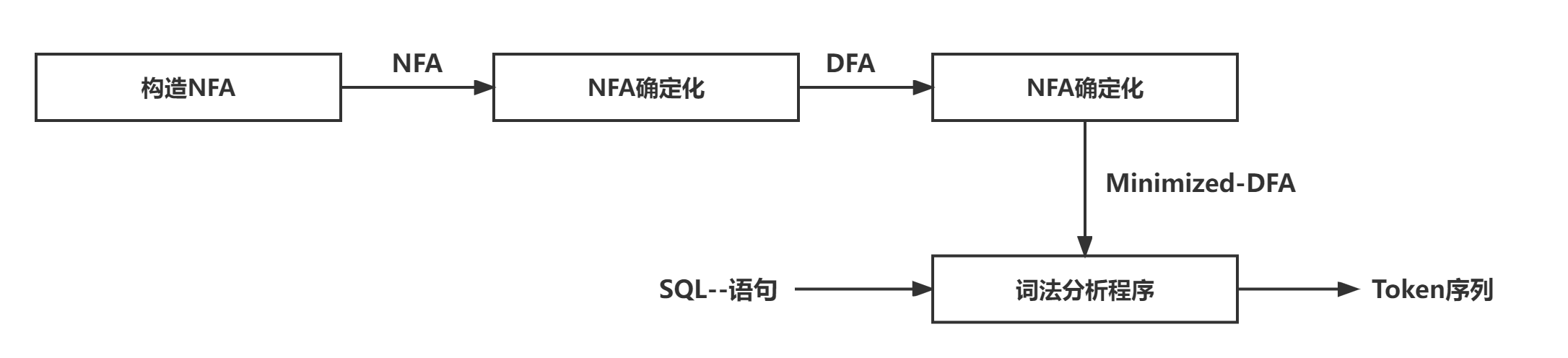
**3.编译器前端(main.py) 的主要作用如下：**

**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。**

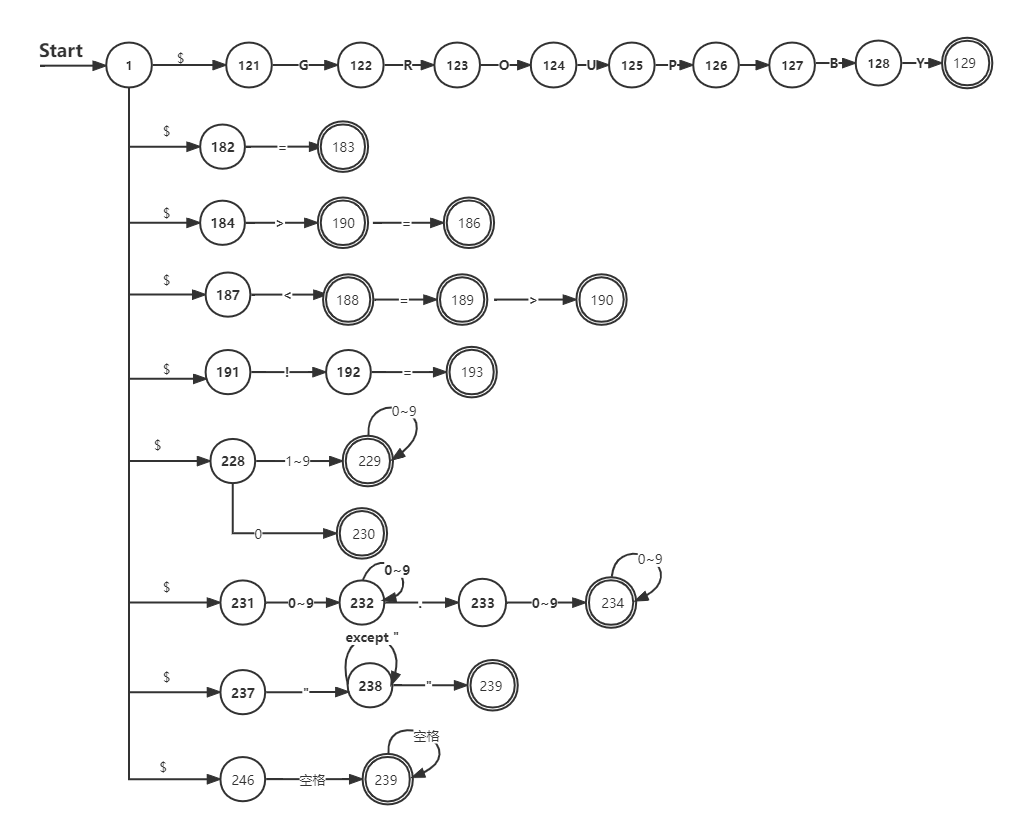
**为了实现上图所示的词法分析器，我一共创建了四个类分别为NFA、DFA、 Token和Lexer。接下来将会给出每个类中的数据结构和函数负责的功能，函数使用到的算法的介绍以及详细的实现过程。**

### 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。**

****

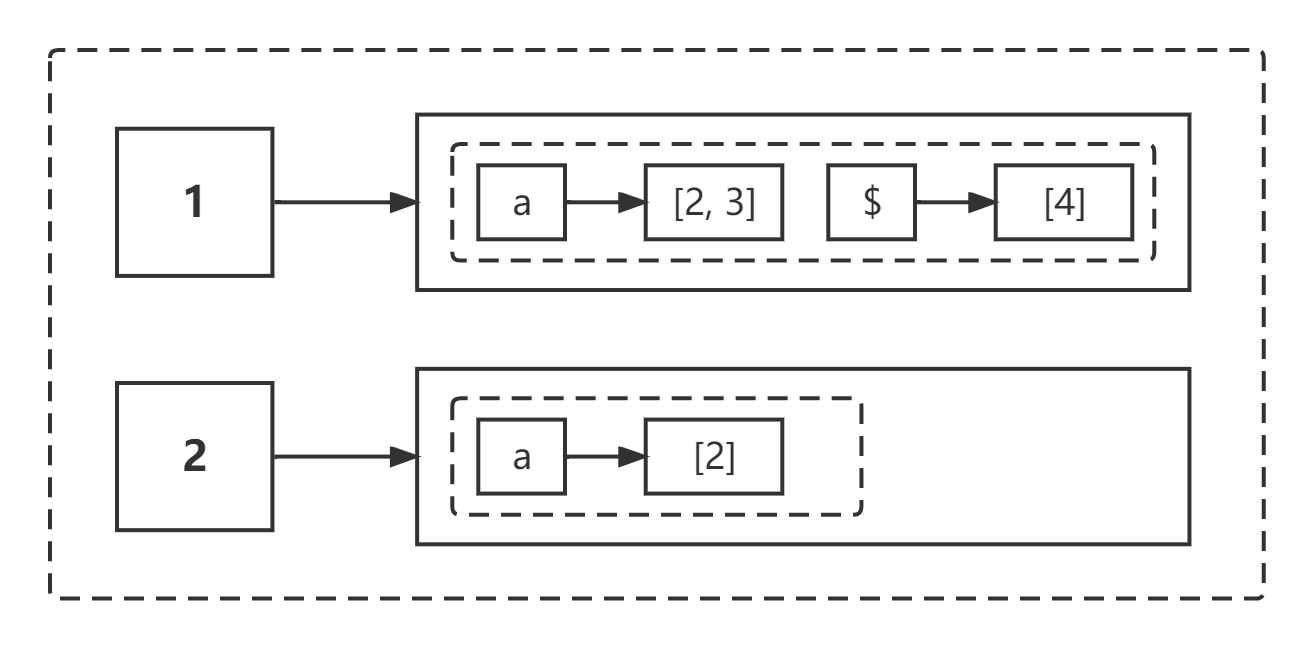
#### 2. 设计NFA类

**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** |

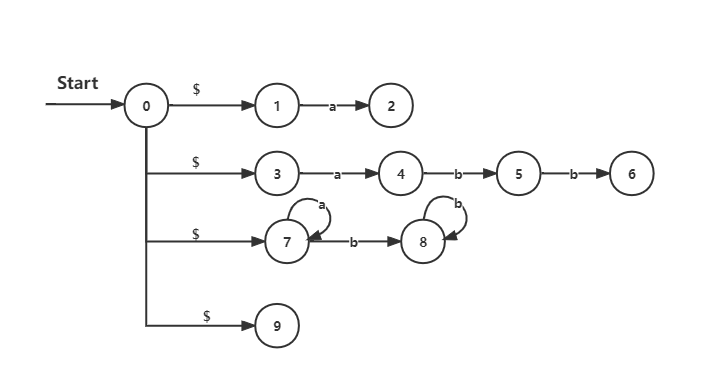
**接下来，我将详细介绍最为核心的两个属性：self.trans\_function和self.accept\_states。**

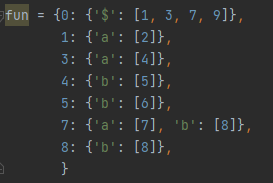
**1) self.trans\_function**

** 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。**

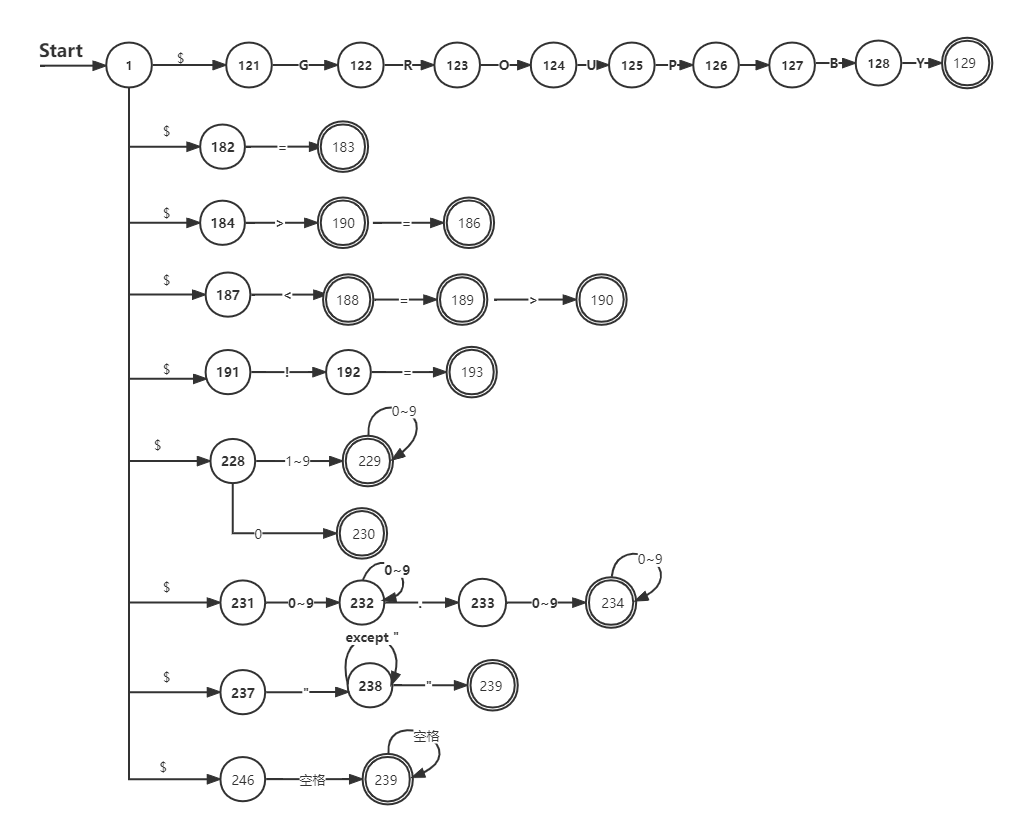
**下面给出了如图所示的NFA的trans\_function：**

****

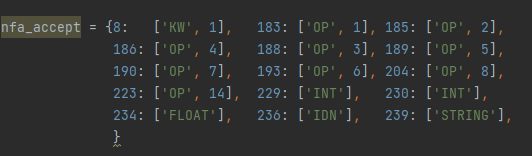
****

**2) self.non\_accept\_states**

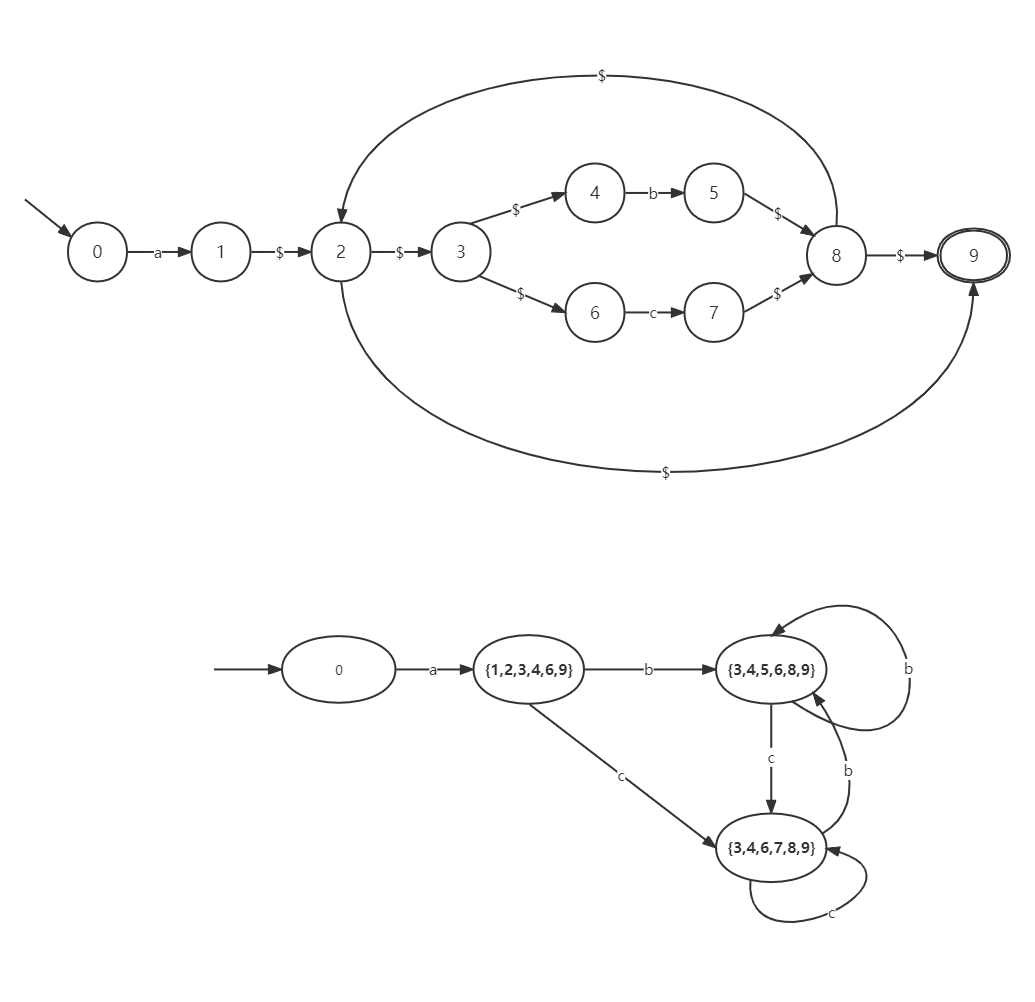
**在考虑NFA接受状态的表示时，不仅要考虑接受状态结点号的存储，还要考虑如何记录该接受状态接收的字符。因此，我选择了用字典来存储NFA的接收状态，字典中的键为接受状态的结点号，对应的值为该接受状态接收的字符串**

**的类型和序号，如下图所示。**

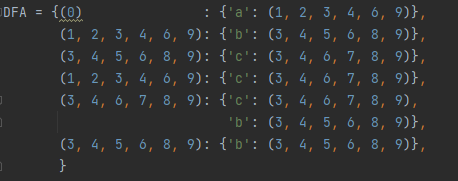
**在保存关键字(KW)、运算符(OP)和标识符(SE)时需要记录他们的类型和该类型中此关键字的序号；对于整数(INT)、浮点数(FLOAT)及字符串(STRING)则只需保存它们的类型即可。如，上图所示的NFA的接受状态用如下字典保存。**

****

### 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：**

****

### 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-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-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 所组成的集合作为M’ 的状态，其中 = -closure(move(I,a)) = -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**

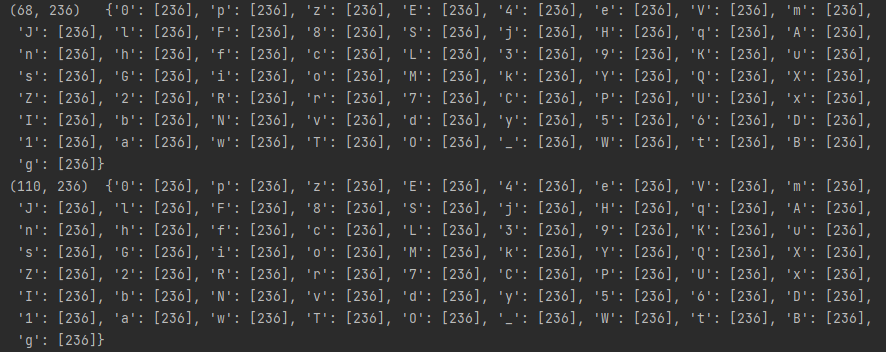
**// 这种情况下应该先将其视为KW**

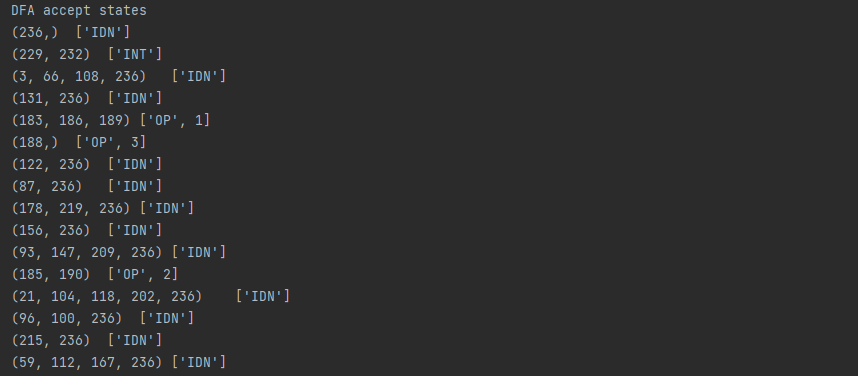
**如果nfa.accept\_states[s][0] != IDN：**

**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：**

****

****

### 3.3.3 DFA的最小化

#### 1. 算法介绍

#### 最小状态

**1）构造状态的初始划分：终态 和非终态两组**

**2）对施用传播性原则构造新划分**

**3）如则令 并继续步骤**

**4）否则重复2 4.为中的每一组选一代表，这些代表构成的状态。若s是一代表,且 , 令r是t组的代表，则中有一转换。 的开始状态是含有 的那组的代表, 的终态是含有 的那组的代表**

**5）去掉中的死状**

**• 对施用传播性原则构造新划分步骤：**

**• 假设被分成m个子集, 且属于不同子集的状态是可区别 的，检查的每一个子集, 看是否能够进一步划分。**

**• 对于某个, 令 若存在数据字符a使得不全包含在现行 的某一子集中，则将一分为二：即假定状态，经过a弧分别达到 状态 且属于现行的两个不同子集，那么将 分成两半，一半 含有且s经a弧到达t1所在子集中的某状态； 另一半含有**

**• 由于是可区别的，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。也就是说和中的 状态时可区别的。**

**•**

#### 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 和非终态两组，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。**

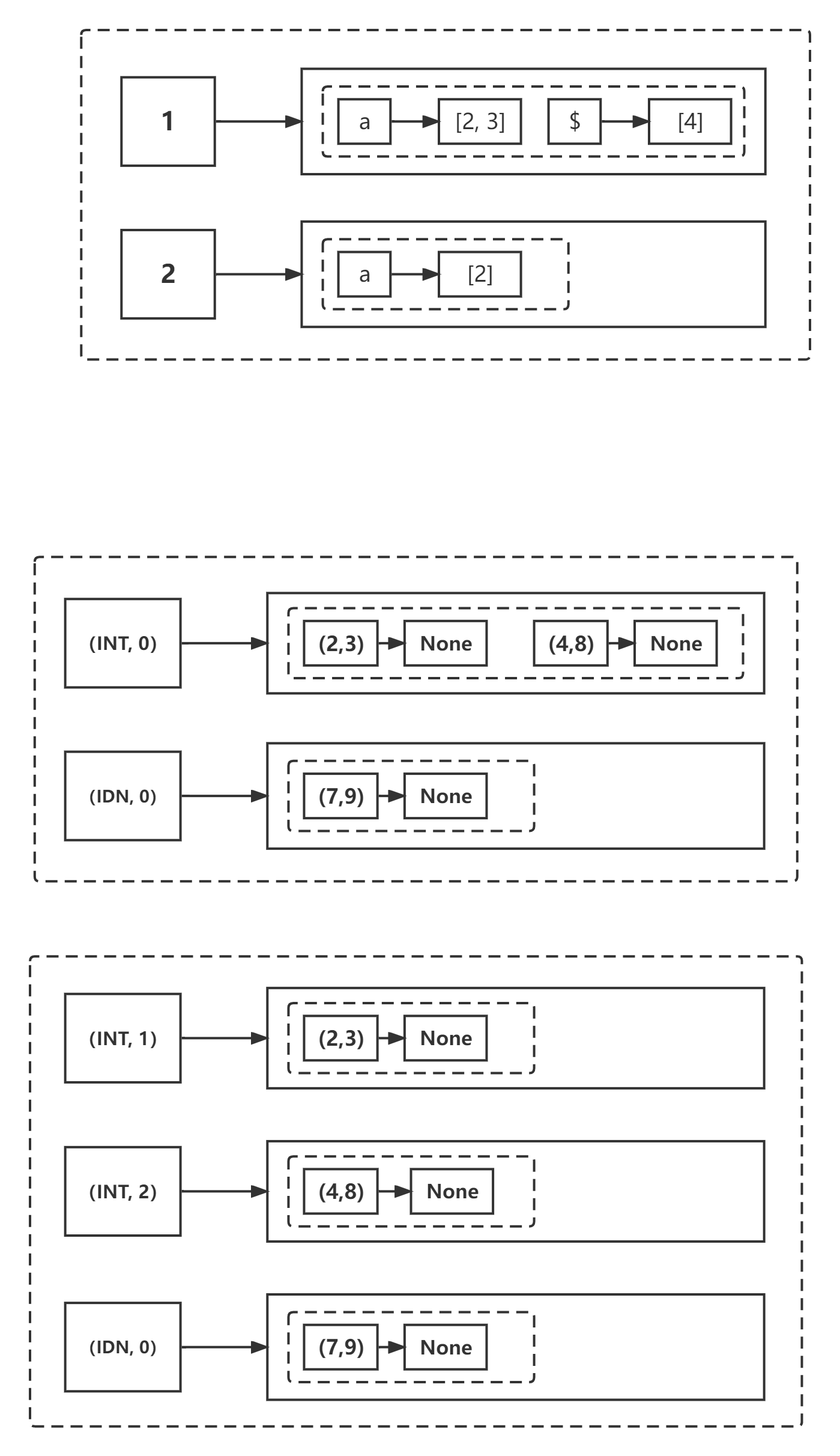
**为了完成以上描述的过程，我在DFA类中添加了一些新的属性和函数，接下来将详细介绍其中核心的数据结构和函数。**

**1）initial\_set(self, category)**

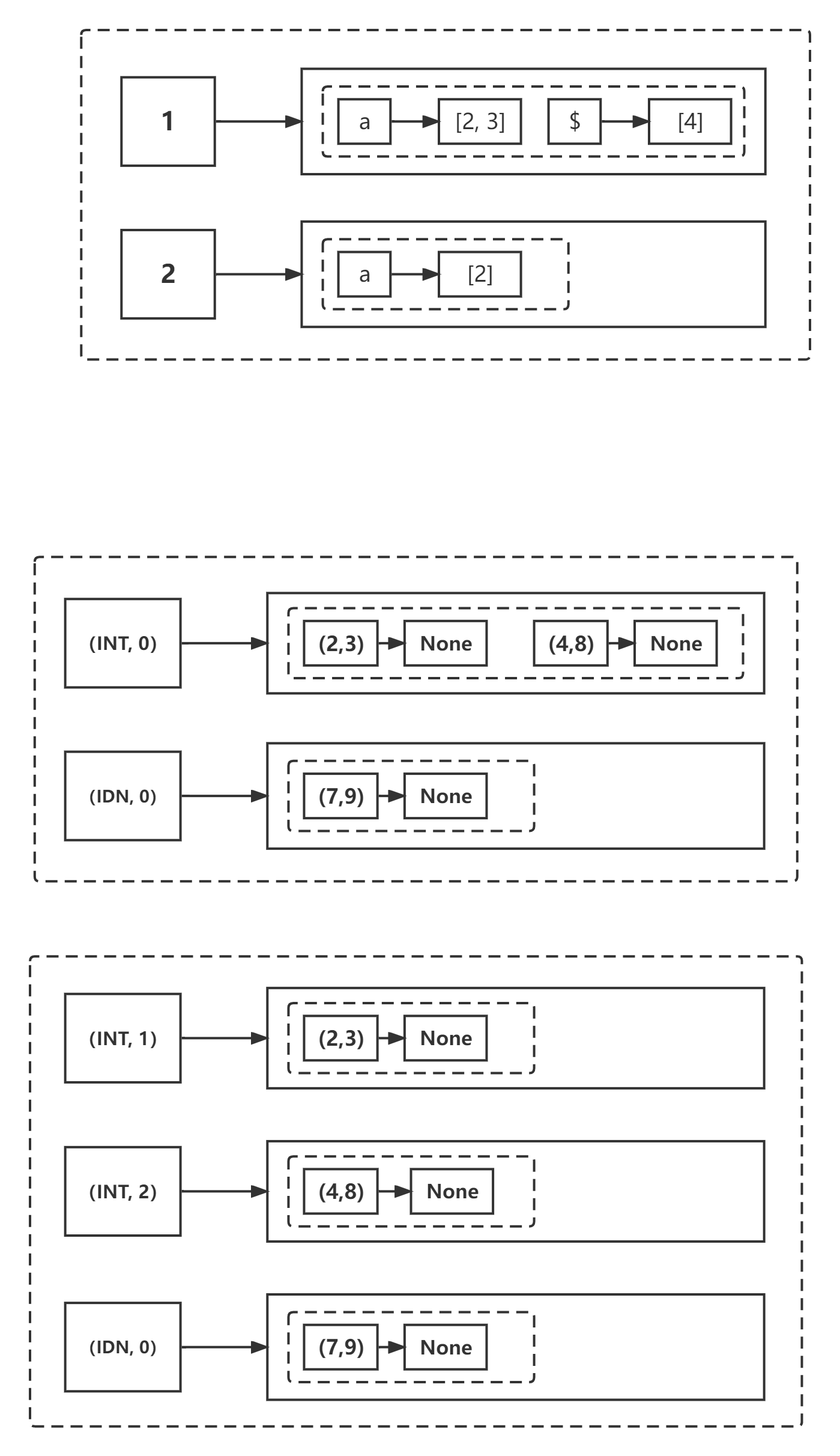
**在前面提到了DFA最小化算法中的初始划分为：非接受状态集合（用NON表示）、标识符接收状态集合（IDN）和整数接收状态集合(INT)。该函数的作用就是根据category参数来返回INT或者IDN的接收状态的集合。只需遍历DFA的accept\_states并从中返回接受状态的动作为category的状态集合、返回结果保存在字典中，字典的键为表示状态标号的元组，对应的值为None。这样设计的原因会在下面其它函数的介绍中给到。**

**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。具体实现如下所示：**

**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**

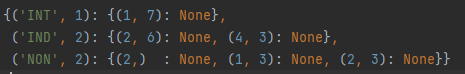
**令cur\_state\_sets\_num为集合个数**

**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：**

****

**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：**

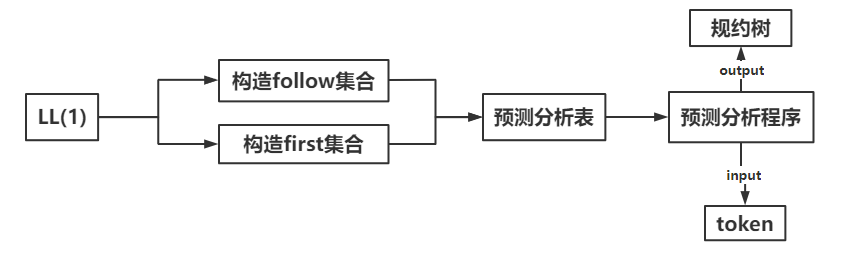
**提示错误信息，返回**

**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。**

## 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

**对每一个文法符号构造: 应用下列规则,直到每个集合不再增大为止.**

**(1)如果,则**

**(2)如果且有产生式则把a加入到中;若也是一个 产生式, 则把加入到中.**

**(3)如果是一个产生式且, 则把加到中; 如果是一个产生式, ,而且对任何****Y, (即) , 则把加到中; 特别是, 若所有的均含有 则把加到中**

#### 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)**

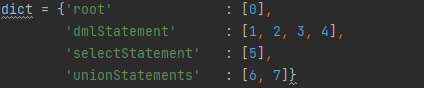
**③ generate\_first\_set(self)**

**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。**

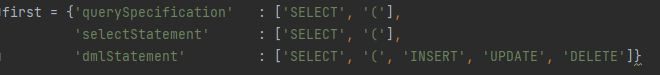
**该函数的作用是使用self.rules\_table保存的内容来构造一个便利与计算FIRST集合的新的数据结构pre\_first\_dict来保存文法，该字典的结构如下图所示。**

****

**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：**

****

**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：**

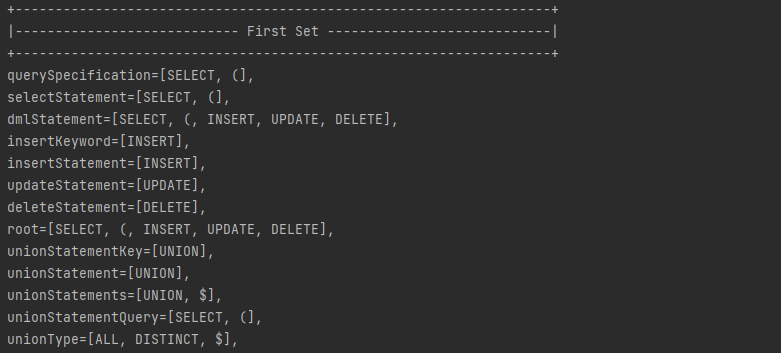
**则调用first(nt)来求FIRST集合**

**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。**

****

### 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 若是一个产生式, 则把FIRST()\{}加至FOLLOW(B)中;**

**③ 若是一个产生式, 或是一个产生式而 (即∈FIRST() ), 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：**

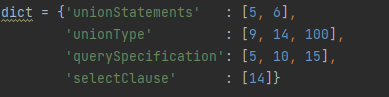
**① generate\_pre\_follow\_dict (self)**

**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。**

**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的任何符号串构造集合**

**(1)首先置对文法**

**(2)如果对任何, 则把加入到 中.特别是,若所有的均含有 则把加到中**

**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。**

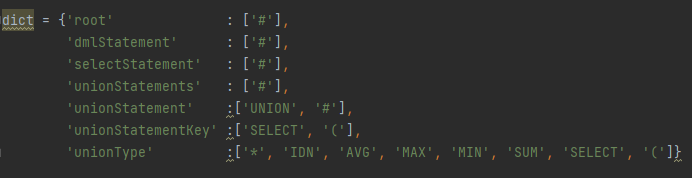
**如果s是为非终结符并且FIRST(s)中含有 ε 且该非终结符不是rhs\_list中的最后一个：**

**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：**

**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**

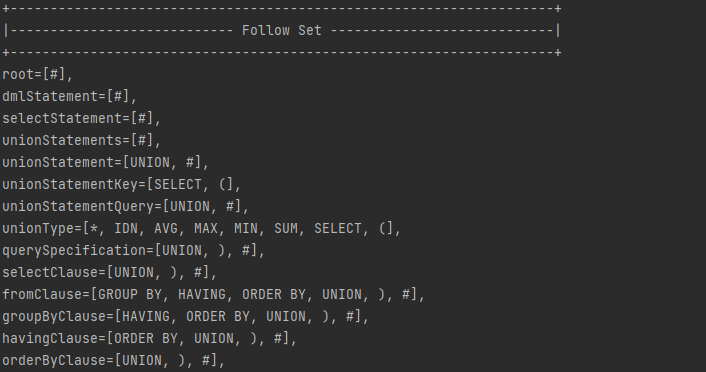
**用字典temp将symbol和产生式左端的非终结符记录下来**

**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。**

### 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 执行第 ② 和 ③ 步；**

**② 对每个终结符, 把加入中；**

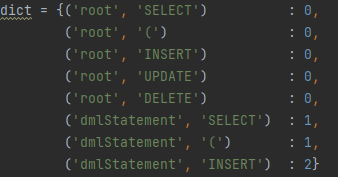
**③ 若 则对任何, 把加入中；**

**④ 把所有无定义的标上“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。**

**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（序号为seq\_num）：**

**对于产生式右端FIRST集合中(即调用first\_of\_symbols())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：**

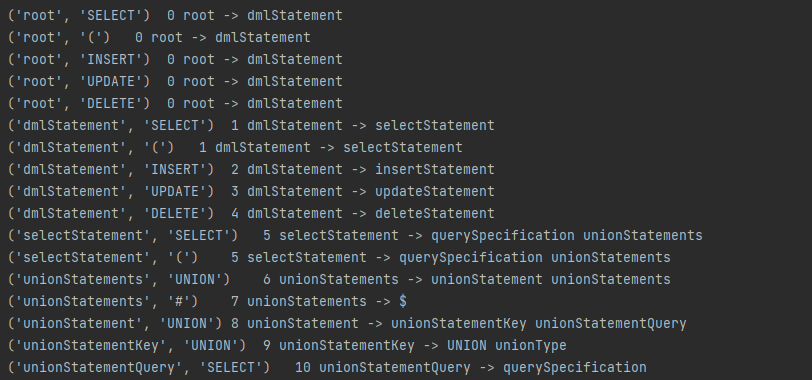
**在parsing\_table中添加键值对 (A ,s) : seq\_num**

**如果该终结符不为 ε ：**

**在parsing\_table中添加键值对 (A ,t) : seq\_num**

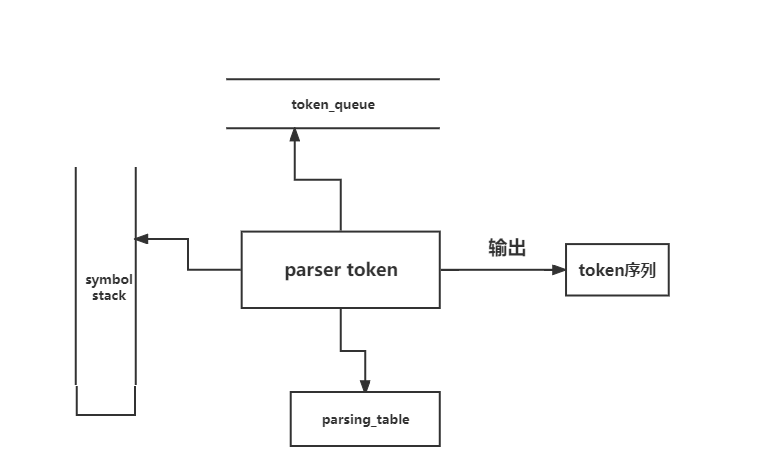
#### 3.运行结果

**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：**

****

### 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：**

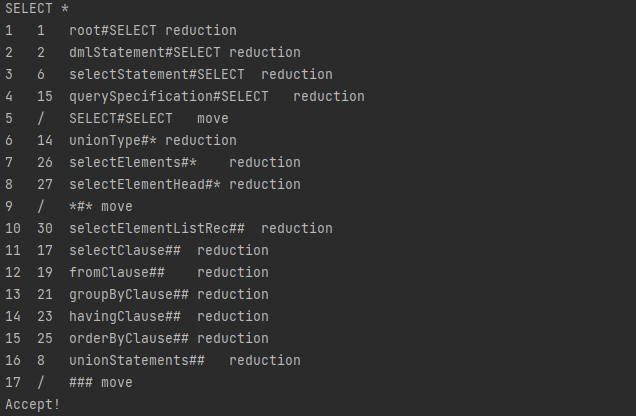
**栈顶元素弹栈，再将产生式右端的符号倒序压栈**

**如果元组(栈顶符号，队头符号)不在parsing\_table中：**

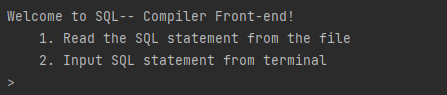
**输出错误提示信息，并返回**

**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。**

# 四、使用说明

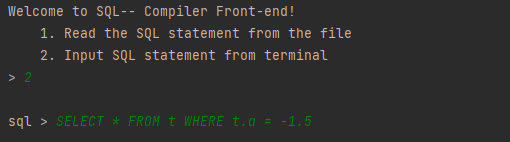
** 下载代码后运行main.py函数，此编译器前端提供了两种输入SQL语句的方法，一种是从文件中读取，另一种是从终端读取。在运行main.py文件后输出以下内容：**

**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。**

# 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** |