天津大学

编译原理
词法分析器/语法分析器
开发报告

学　　院 智能与计算/求是学部

专　　业 计算机科学与技术

组　　长 杨亦凡 **3019234258**

组　　员 李自安 **3019207257**

组　　员 石昊　**3019208051**

组　　员 华溢　**3019244091**

目录

1 项目简介

本项目针对 SQL - - 语言开发了一个编译器前端，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) 使用自下而上的语法分析方法编写语法分析器

1.1 开发环境

macOS Monterey 12.3.1

1.2 实现语言

Python 3.9

1.3 项目结构

```
.
├── docs                          # 文档
├── main.py                       # 主程序
├── output                        # 输出文件夹
│   ├── ds_details
│   ├── ds_entity
│   ├── input_stack
│   ├── parse_result              # 规约序列
│   └── token_table               # token 序列
├── src                           # 源代码
│   ├── lexer                     # 词法分析器
│   │   ├── ds                    # 词法分析器数据结构
│   │   │   ├── graph.py
│   │   │   └── token.py
│   │   ├── lexical_analyzer.py   # 词法分析器主程序
│   │   ├── rex2nfa.py            # 正则表达式转 NFA
│   │   ├── nfa2dfa.py            # NFA 确定化
│   │   └── dfa2min.py            # DFA 最小化
│   └── LR1_parser                # LR1语法分析器
│       ├── ds                    # 语法分析器数据结构
│       │   ├── analysisTable.py
│       │   ├── FF.py
│       │   ├── grammar.py
│       │   ├── item.py
│       │   └── stack.py
│       └── LR1.py                # 语法分析器主程序
├── tests                         # 测试用例
└── utils                         # 全局工具
    ├── configs.py                # 配置
    ├── regex.json                # 词法
    ├── grammar.txt               # 语法
    └── transformer.py            # 转换器
```

`./src/lexer/lexical_analyzer.py` 是词法分析器主程序

`./src/LR1_parser/LR1.py` 是语法分析器主程序

`./tests` 是测试用例文件夹

`./output/token_table` 是 token 序列输出

`./output/parse_result` 是规约序列输出

2 词法分析器

词法分析器首先用 SQL - - 语言所有单词符号对应的正则表达式通过正则表达式转 NFA 算法构建相应的 NFA，再通过 NFA 确定化算法构建相应的 DFA，然后通过 DFA 最小化算法构建有限自动机，最后根据有限自动机编程实现 SQL - - 语言的词法分析器。

2.1 需求分析

词法分析器的输入为 SQL - - 语言源代码，词法分析器对其预处理去除注释，识别单词的二元属性并生成符号表，将待分析代码转化为语法分析器可接受的序列。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:

TABLE 1: 关键字（KW，规定为大写）

类别	语法关键字
查询表达式	(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	(10) UPDATE, (11) 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) ORFER BY
条件语句	(28) TRUE, (29) FALSE, (30) UNKNOWN, (31) IS, (32) NULL

TABLE 2: 运算符（OP，规定为大写）

类别	语法关键字
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) !
算术运算符	(13) -
属性运算符	(13) .

TABLE 3: 界符（SE）

类别	语法关键字
界符	(1) (, (2)), (3) ,

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**正数**相同，负数通过在正数前面加算术运算符（-）实现;

字符串（STRING）定义与 C 语言相同，使用**双引号**包含的任意字符串。

2.2 数据结构描述

2.2.1 符号表项

ADT 1: TokenLine 抽象数据类型

```
ADT TokenLine {
    数据对象: D = {
        待测代码中的单词符号 word_content,
        单词符号大类 word_type,
        单词符号种别 token_type,
        单词符号内容 token_content
        }
    数据关系: R = {
        <word_content, word_type>,
        <word_type, token_type>,
        <token_type, token_content>
    }
    基本操作:
        input_raw_line(word_content, word_type)
            初始条件: word_content 存在, word_type 合法
            操作结果: 产生 token_type, 产生 token_content
        output_token_line()
            初始条件: word_content 存在, word_type 合法
            操作结果: 产生 token_type, 产生 token_content
        _process_raw_line()
            初始条件: word_content 存在, word_type 合法
            操作结果: 产生 token_type, 产生 token_content
        _get_token_id(token_type)
            初始条件: token_type 存在
```

```
            操作结果：返回 word_content 在 token_type 表中对应的序号
} ADT TokenLine
```

2.2.2 符号表

ADT 2: TokenTable 抽象数据类型

```
ADT TokenTable {
    数据对象：D = {TokenLine i | TokenLine i 是符号表项，i = 0,1,...,n，n >= 0}
    数据关系：R = {<Ti，Ti+1>| Ti，Ti+1 属于D，i = 0,1,...,n-1}
    基本操作：
        add_token_line(toke_line)
            初始条件：toke_line 存在
            操作结果：将 toke_line 加入到 TokenTable 末尾
        print()
            初始条件：TokenTable 存在
            操作结果：格式化打印 TokenTable
        save(test_name)
            初始条件：TokenTable 存在
            操作结果：格式化存储 TokenTable，文件名为 test_name
        reset()
            操作结果：重置 TokenLine
} ADT TokenTable
```

2.2.3 状态

ADT 3: State 抽象数据类型

```
ADT State {
    数据对象：D = {
        状态编号 id,
        结点集合 node_set,
        起始状态标识位 is_begin,
        结束状态标识位 is_end
} ADT State
```

2.2.4 结点

ADT 4: Node 抽象数据类型

```
ADT Node {
    数据对象: 结点编号 num
} ADT Node
```

2.2.5 边

ADT 5: Edge 抽象数据类型

```
ADT Edge {
    数据对象: D = {
        弧尾结点编号 node_tail,
        弧头结点编号 node_head,
        边的类型 node_type
        }
    数据关系: R = {<Ni, Nj> | Ni, Nj 属于 Node, i, j = 0,1,...,n}
} ADT Edge
```

2.2.6 图

ADT 6: Graph 抽象数据类型

```
ADT Graph {
    数据对象: D = {
        源点编号 node_tail,
        汇点编号 node_head,
        源点集合 node_heads = {Ni | Ni 属于 Node, i = 0,1,...,n}
        边集合 edges = {Ei | Ei 属于 Edge, i = 0,1,...,n}
        }
    基本操作:
        add_star(obj)
            初始条件: obj 存在
            操作结果: 构建一个带有 obj 类型的自环的结点,
                      头尾各通过空弧接一个结点, 并将整体加入到图中。
        add_union(obj1, obj2)
            初始条件: obj1, obj2 存在
            操作结果: 构建一对通过两条反向弧连接的结点,
                      弧的类型分别为 obj1, obj2,
                      并将整体加入到图中。
        add_concat(obj1, obj2)
            初始条件: obj1, obj2 存在
```

```
        操作结果：构建一对通过两条同向弧连接的结点，
                  弧的类型分别为 obj1，obj2，
                  并将整体加入到图中。
} ADT Graph
```

2.3 算法描述

2.3.1 正则表达式转 NFA 算法

Algorithm 1 REGEX TO NFA Algorithm

Inputs:

正则表达式 $regex$

Outputs:

NFA $M = \{S, \Sigma, \delta, S_0, S_t\}$

对输入的正则表达式预处理，将正则表达式中隐含的 & 补充
初始化操作符栈 $OPTR$ 和操作数栈 $OPND$
while $OPTR$[-1] == '#' and $regex$[i] == '#' **do**
    扫描处理后的正则式，读入一个字符 ch
    while ch 为操作数 **do**
        将 ch 压入 $OPND$，读入一个字符 ch
    end while
    比较当前输入操作符和操作符栈顶的操作符的优先级
    if 当前输入操作符优先级更高 **then**
        将 ch 压入 $OPTR$，读入一个字符 ch
    end if
    if 两者优先级相同 **then**
        弹出 $OPTR$ 栈顶元素，读入一个字符 ch
    end if
    if 当前输入操作符优先级更低 **then**
        弹出 $OPTR$ 栈顶元素，根据其具体类别弹出 $OPND$ 1 个或 2 个操作数，生成相应的状态图并压入 $OPND$
    end if
end while

2.3.2 NFA 确定化算法

Algorithm 2 NFA TO DFA Algorithm

Inputs:

NFA $M = \{S, \Sigma, \delta, S_0, S_t\}$

Outputs:

DFA $M' = \{S, \Sigma, \delta, S_0, S_t\}$

1: 从 M 的 S_0 出发，将仅经过任意条 ε 弧能到达的状态组成的集合 I 作为 M' 的初态 q_0
2: **while** 不再产生新的状态 **do**
3: 　从 I 中元素出发，将经过任意 $a \in \Sigma$ 的 a 弧转换 I_a 所组成的集合作为 M' 的状态
4: **end while**

2.3.3 DFA 最小化算法

Algorithm 3 DFA Minimization Algorithm

Inputs:

DFA $M = \{S, \Sigma, \delta, S_0, S_t\}$

Outputs:

DFA $M' = \{S, \Sigma, \delta, S_0, S_t\}$

1: 构造状态集的初始划分 Π，包含终态 S_t 和非终态 $S - S_t$ 两组
2: **while** not done **do**
3: 　对 Pi 使用传播性原则构造新划分 Π_{new}
4: 　**if** Π_{new} == Π **then**
5: 　　Π_{final} == Π done
6: 　**else**
7: 　　Π == Π_{new}
8: 　**end if**
9: **end while**
10: **for** all Π_{final} **do**
11: 　选一代表元素 s，加入到 M' 的 S
12: 　**if** 代表 s 满足 $\delta(s, a) == t$ **then**
13: 　　令 r 作为 t 组的代表，将转换 $\delta(s, a) = r$ 加入到 M' 的 δ
14: 　**end if**
15: 　将含有 S_0 的组的代表作为 M' 的开始状态
16: 　将含有 S_t 的组的代表作为 M' 的终态
17: **end for**
18: 去除 M' 的 S 中死状态

2.3.4 去除注释算法

Algorithm 4 Input Preprocessing Algorithm

Inputs:

带注释的 SQL - - 语言源代码 $Text_{raw}$

Outputs:

去除注释的 SQL - - 语言源代码 $Text_{preprocessed}$

1: 当前字符串 $current\_string \leftarrow$ ””

```
当前状态 state ← 代码
for all ch ∈ Text_raw do
    if state == 代码 then
        if ch == '/' then
            state ← 斜杠
        else if ch == "-" then
            state ← 短横线
        else
            current_string += ch
            if ch == '\'' then
                state ← 字符
            else if ch == '\"' then
                state ← 字符串
            end if
        end if
    else if state == 短横线 then
        if ch == '-' then
            state ← 单行注释
        else
            current_string += "-" + ch
            state ← 代码
        end if
    else if state == 斜杠 then
        if ch == '*' then
            state ← 多行注释
        else if ch == '/' then
            state ← 单行注释
        else
            current_string += "/"
            current_string += ch
            state ← 代码
        end if
    else if state == 多行注释 then
        if ch == '*' then
            state ← 多行注释遇到 *
        else
            if ch == '\n' then
                current_string += '\r\n'
                state ← 多行注释
```

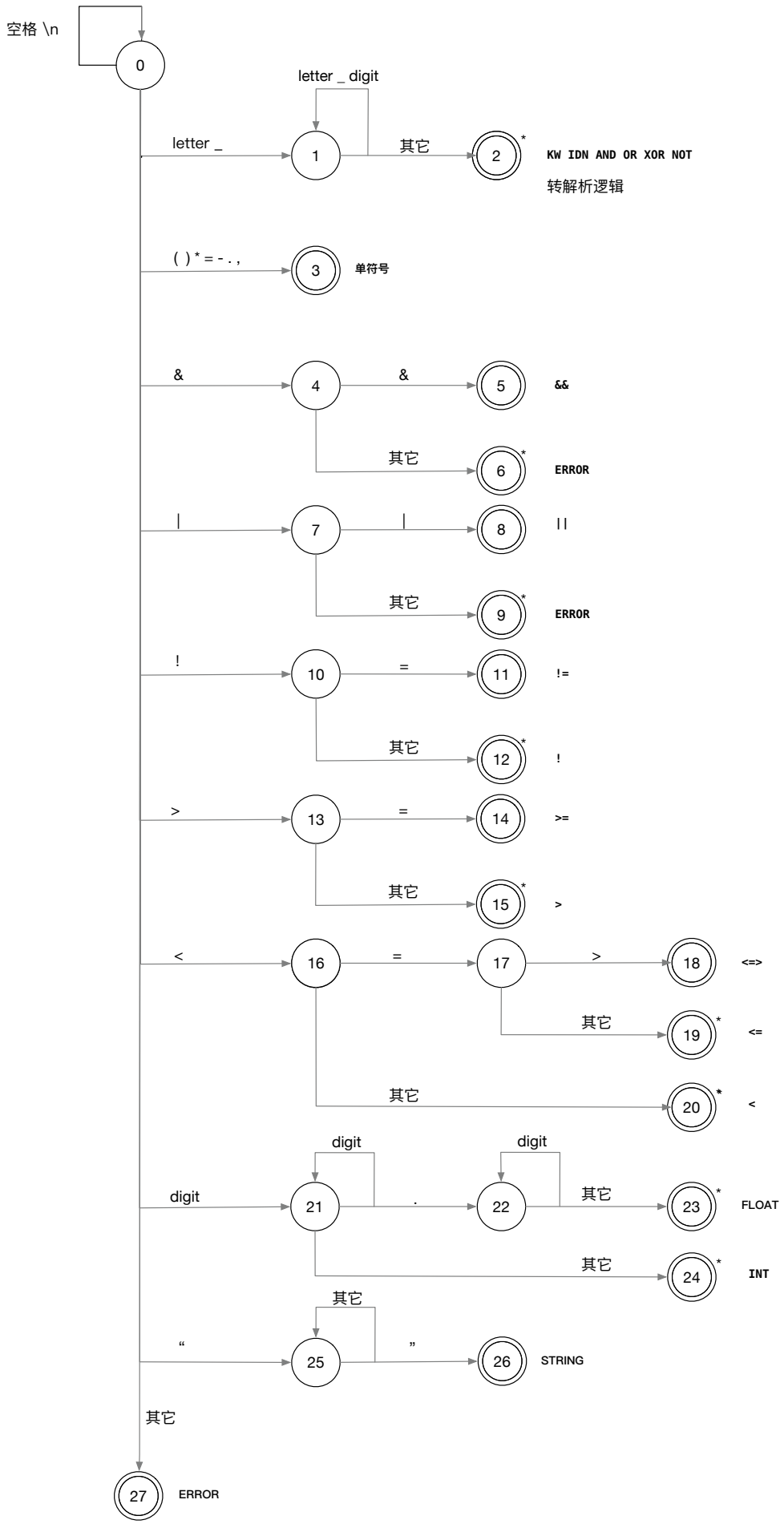
```
        end if
    end if
else if state == 多行注释遇到 then
    if ch == '/' then
        state ← 代码
    else if ch == '*' then
        state ← 多行注释遇到
    else
        state ← 多行注释
    end if
else if state == 单行注释 then
    if ch == '\\' then
        state ← 拆行注释
    else if ch == '\n' then
        current_string += '\r\n'
        state ← 代码
    else
        state ← 单行注释
    end if
else if state == 拆行注释 then
    if ch == '\\' OR ch == '\r' OR ch == '\n' then
        if ch == '\n' then
            current_string += '\r\n'
        end if
        state ← 拆行注释
    else
        state ← 单行注释
    end if
else if state == 字符 then
    current_string += ch
    if ch == '\\' then
        state ← 字符中的转义字符
    else if ch == '\'' then
        state ← 代码
    else
        state ← 字符
    end if
else if state == 字符中的转义字符 then
    current_string += ch
```

```
        state ← 字符
    else if state == 字符串 then
        current_string += ch
        if ch == '\\' then
            state ← 字符串中的转义字符
        else if ch == '\"' then
            state ← 代码
        else
            state ← 字符串
        end if
    else if state == 字符串中的转义字符 then
        current_string += ch
        state ← 字符串
    end if
end for
Text_preprocessed += current_string
```

2.3.5 词法分析器解析

词法分析器将去除注释的 SQL - - 语言源代码逐行输入到有限状态机逻辑2.3.5，初步解析出单词和单词符号大类，再经过符号表项解析逻辑得到准确的单词符号种别和单词符号内容。

值得说明的是，对于复合单词”ORDER BY” 与”GROUP BY”，当词法分析器识别到”ORDER” 或”GROUP” 后超前搜索 4 个字符，如果为” BY ” 则将两者合并输出，否则转到词法检查逻辑，根据具体情况决定是单独输出”ORDER”/”GROUP” 或是抛出异常。


空格 \n
0
letter _ digit
letter _
1
其它
2
KW IDN AND OR XOR NOT
转解析逻辑
( ) * = - . ,
3
单符号
&
4
&
5
&&
其它
6
ERROR
|
7
|
8
| |
其它
9
ERROR
!
10
=
11
!=
其它
12
!
>
13
=
14
>=
其它
15
>
<
16
=
17
>
18
<=>
其它
19
<=
其它
20
<
digit
digit
digit
21
.
22
其它
23
FLOAT
其它
24
INT
其它
“
25
”
26
STRING
其它
27
ERROR

2.4 输出格式说明

2.4.1 正则表达式转 NFA 算法

```
Input: (12|345)*123
================= NFA ==================
start_node:9
node_head:13
Edge 0 -> 1, node_type:1
Edge 1 -> 2, node_type:2
Edge 3 -> 4, node_type:3
Edge 4 -> 5, node_type:4
Edge 5 -> 6, node_type:5
Edge 7 -> 0, node_type:varepsilon
Edge 7 -> 3, node_type:varepsilon
Edge 2 -> 8, node_type:varepsilon
Edge 6 -> 8, node_type:varepsilon
Edge 9 -> 10, node_type:varepsilon
Edge 9 -> 7, node_type:varepsilon
Edge 8 -> 10, node_type:varepsilon
Edge 8 -> 7, node_type:varepsilon
Edge 10 -> 11, node_type:1
Edge 11 -> 12, node_type:2
Edge 12 -> 13, node_type:3
```

2.4.2 NFA 确定化算法

```
Input: (12|345)*123
================= DFA ==================
start_node:0
node_heads:[5]
Edge 0 -> 1, node_type:1
Edge 0 -> 2, node_type:3
Edge 1 -> 3, node_type:2
Edge 2 -> 4, node_type:4
Edge 3 -> 1, node_type:1
Edge 3 -> 5, node_type:3
Edge 4 -> 6, node_type:5
Edge 5 -> 4, node_type:4
Edge 6 -> 1, node_type:1
Edge 6 -> 2, node_type:3
```

2.4.3 DFA 最小化算法

```
Input: (12|345)*123
========== Construct matrix ==========
['1', '2', '3', '4', '5']
[1, -1, 2, -1, -1]
[-1, 3, -1, -1, -1]
[-1, -1, -1, 4, -1]
[1, -1, 5, -1, -1]
[-1, -1, -1, -1, 6]
[-1, -1, -1, 4, -1]
[1, -1, 2, -1, -1]
=======================================
============ Minimized DFA ============
start_node:0
node_heads:[5]
Edge 1 -> 3, node_type:2
Edge 2 -> 4, node_type:4
Edge 3 -> 1, node_type:1
Edge 3 -> 5, node_type:3
Edge 4 -> 0, node_type:5
Edge 5 -> 4, node_type:4
Edge 0 -> 1, node_type:1
Edge 0 -> 2, node_type:3
```

2.4.4 TOKEN 序列

Token 输出格式：

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

其中，单词符号种别为 KW（关键字）、OP（运算符）、SE（界符）、IDN（标识符）、INT（整形数）、FLOAT（浮点数）、STRING（字符串）；单词符号内容 KW、OP、SE 为其编号，KW 见表 2.1，OP 见表 2.1，SE 见表2.1；其余类型为其值。

2.5 源程序编译步骤

本项目的词法分析器采用 python 编写，无须编译，可以直接运行。其中，代码对环境的要求如下：

```
python3
```

在工程根目录终端下，键入如下命令即可快速运行：

```
python main.py
```

其中，main.py 文件的输入为./tests 目录下的”testcase-*.sql” 文件，通过词法分析与语法分析分别输出 token 序列和规约序列。

若要单独查看与调试词法分析器本身的功能，可以在工程根目录下键入以下命令：

```
python ./src/lexer/lexical_analyzer.py
```

若要查看与调试词法分析器子模块 X(可选项为 nfa2dfa，rex2nfa，dfa2min) 的功能，可以在工程根目录下键入以下命令：

```
python ./src/lexer/X.py
```

3 语法分析器

3.1 数据结构描述

3.1.1 GRAMMAR 相关数据结构

语法分析前需要首先对文法进行预处理，即将以文本文件形式存储的语法转换为 $[V_N, V_T, P, S]$ 的形式，其中 V_N 为非终结符集，V_T 为终结符集，P 为产生式集，S 为开始符号。因此分别设计 production（产生式）与 grammar（文法）两个抽象数据类型，来实现相应类型数据的存储与计算，分别如下：

ADT 7: production 抽象数据类型

```
ADT production {
    数据对象：D = {
        左部符号 left,
        右部符号 right,
        产生式编号 index
    }
    基本操作：
        init(left, right, index)
            初始条件：left存在，right与index合法
            操作结果：初始化production
} ADT production
```

ADT 8: grammar 抽象数据类型

```
ADT grammar {
    数据对象：D = {
        非终结符集 non_terminal,
        终结符集 terminal,
        产生式集 productions,
        初始符号 start
        grammar文本内容 grammar_context
```

```
    }
    基本操作:
        init(grammar_file_path)
            初始条件: 路径存在
            操作结果: 初始化grammar各变量
        load_grammar(grammar_file_path)
            初始条件: 路径存在
            操作结果: 根据grammar文本文件, 计算得到non_terminal, terminal,
            productions, start等成员对象
        get_augumented_grammar()
            初始条件: grammar加载完成
            操作结果: 将grammar转换为其对应的增广文法
} ADT grammar
```

3.1.2 FIRST & FOLLOW 集合

在计算项目集规范族之前，需要首先根据文法计算出该文法对应的 FIRST 集与 FOLLOW 集（事实上，在计算项目集规范族时 FOLLOW 集并不是必要的，但是为了完整，我们仍旧设计该数据类型与算法）。为了能够规范地存储与计算 FIRST 集与 FOLLOW 集，设计 FIRST 与 FOLLOW 两个抽象数据类型，分别如下：

ADT 9: FIRST 抽象数据类型

```
ADT FIRST {
    数据对象: D = {
        FIRST集合 first_sets,
        文法对象 grammar_obj
    }
    基本操作:
        init(grammar_obj)
            初始条件: grammar_obj存在
            操作结果: 初始化FIRST
        calculate_first_sets()
            初始条件: FIRST已初始化
            操作结果: 根据文法计算该文法中各个符号的FIRST集合, 并存入first_sets中
        calculate_follow_set(symbols)
            初始条件: FIRST已初始化、first_sets已计算完成、symbols合法
            操作结果: 返回符号串symbols的FIRST集合
        dump_first_sets_into_file(file_path)
            初始条件: FIRST已初始化、first_sets已计算完成、file_path合法
```

```
            操作结果：将FIRST按照规定格式写入文件file_path中
} ADT FIRST
```

ADT 10: FOLLOW 抽象数据类型

```
ADT FOLLOW {
    数据对象：D = {
        FOLLOW集合 follow_sets
    }
    基本操作：
        init(grammar_obj, FIRST_obj)
            初始条件：grammar_obj存在，FIRST_obj存在，并且FIRST_obj
            与grammar_obj相对应
            操作结果：初始化FOLLOW
        calculate_follow()
            初始条件：FOLLOW已初始化
            操作结果：根据文法grammar_obj与其对应的FIRST_obj计算该文法中
            各个符号（终结符）的FOLLOW集合，并存入follow_sets中
        dump_follow_sets_into_file(file_path)
            初始条件：FOLLOW已初始化、follow_sets已计算完成、file_path合法
            操作结果：将FOLLOW按照规定格式写入文件file_path中
} ADT FOLLOW
```

3.1.3 项目集规范族相关数据类型

计算文法的项目集规范族时，涉及到了较多的算法和表示形式，因此分别设计如下三个抽象数据类型：

ADT 11: item 抽象数据类型

```
ADT item {
    数据对象：D = {
        项目产生式左端符号 left,
        项目产生式右端符号 right,
        项目dot位置 dot_pos,
        项目对应的终结符号集合 terminals,
        项目对应的产生式的编号 index
    }
    基本操作：
        init(left, right, dot_pos, terminals, index)
```

```
            初始条件: left, right, dot_pos, terminals, index合法
            操作结果: 初始化item
        go(symbol)
            初始条件: item已初始化、symbol合法
            操作结果: 计算move(item, symbol), 即根据输入的终结符得到下一个项目。

} ADT item
```

ADT: 12 itemSet 抽象数据类型

```
ADT itemSet {
    数据对象: D = {
        项目集合 item_set,
        项目集合的编号 index
    }
    基本操作:
        init(item_set, index)
            初始条件: item_set合法、index合法
            操作结果: 初始化itemSet
        calculate_closure(grammar_obj, FIRST_obj)
            初始条件: itemSet已初始化、grammar_obj存在、FIRST_obj存在,
            并且FIRST_obj
            与grammar_obj相对应
            操作结果: 计算项目集合的闭包, 项目增加入item_set中


} ADT itemSet
```

ADT 13: itemSets 抽象数据类型

```
ADT itemSets {
    数据对象: D = {
        项目集规范族 item_sets
        go关系 go
    }
    基本操作:
        init()
            初始条件: 无
            操作结果: 初始化itemSets
        calculate_itemSets(grammar_obj, FIRST_obj)
```

```
            初始条件：itemSets已初始化、grammar_obj存在、FIRST_obj存在，
            并且FIRST_obj与grammar_obj相对应
            操作结果：根据语法计算其对应的项目集规范族，并存入item_sets中
        calculate_go(itemSet_obj, grammar_obj)
            初始条件：itemSet_obj合法、grammar_obj存在
            操作结果：计算itemSet_obj在grammar_obj的文法下可以跳转到的项目集，
            并将相应的关系存入go中
        convert_go()
            初始条件：itemSets已初始化
            操作结果：将成员变量go从原来的列表类型转换为字典类型
        merge_item_sets()
            初始条件：itemSets各成员变量已经计算完成
            操作结果：将成员变量item_sets中各个项目集进行merge操作，将项目集
            内具有相同产生式和dot位置的项目对应的终结符号集合合并起来，作为一个项目
        dump_into_file():
            初始条件：itemSets已计算完毕
            操作结果：将itemSets中的数据按照规定格式写入文件
} ADT itemSets
```

3.1.4 LR1 分析表相关数据类型

构建 LR1 分析表时，我们需要根据文法分别计算 action_table 以及 goto_table，因此分别设计 action 与 goto 相关的两个抽象数据类型，如下：

ADT 14: actionTable 抽象数据类型

```
ADT actionTable {
    数据对象：D = {
        动作表 action_table
    }
    基本操作：
        init()
            初始条件：无
            操作结果：初始化actionTable
        calculate_action_Table(grammar_obj, itemSets_obj)
            初始条件：actionTable已初始化、grammar_obj存在、itemSets_obj存在，
            并且itemSets_obj与grammar_obj相对应
            操作结果：根据语法规则和项目集规范族计算其对应的动作表，并存入
            action_table中
        dump_table_into_file(file_path)
```

```
            初始条件: action_table已计算完毕、file_path合法
            操作结果: 将action_table中的数据按照规定格式写入文件
} ADT actionTable
```

ADT 15: gotoTable 抽象数据类型

```
ADT gotoTable {
    数据对象: D = {
        跳转表 goto_table
    }
    基本操作:
        init()
            初始条件: 无
            操作结果: 初始化gotoTable
        calculate_goto_Table(grammar_obj, itemSets_obj)
            初始条件: gotoTable已初始化、grammar_obj存在、itemSets_obj存在，并且
            itemSets_obj与grammar_obj相对应
            操作结果: 根据语法规则和项目集规范族计算其对应的跳转表，并存入
            goto_table中
        dump_table_into_file(file_path)
            初始条件: goto_table已计算完毕、file_path合法
} ADT gotoTable
```

3.1.5 LR1 语法分析器

计算出前面所有的数据类型后，我们可以进行语法分析器的进一步设计。我们将语法分析器封装成一个类，以方便调用与计算，其对应的抽象数据类型如下：

ADT 16: LR1_parser 抽象数据类型

```
ADT LR1_parser {
    数据对象: D = {
        语法分析器对应文法 grammar_obj,
        语法分析器对应FIRST集 FIRST_obj,
        语法分析器对应FOLLOW集 FOLLOW_obj,
        语法分析器对应项目集规范族 itemSets_obj,
        语法分析器对应分析表 analysisTable_obj,
        语法分析器基本配置项 Configs
    }
    基本操作:
        init(Configs)
```

初始条件：Configs合法
操作结果：根据Configs初始化LR1_parser
LR1_init(grammar_path)
初始条件：grammar_path合法
操作结果：根据grammar_path计算LR1_parser所有成员变量
parse(input_stack, REDIRECT_STDOUT_TO_FILE, save_path)
初始条件：input_stack合法、REDIRECT_STDOUT_TO_FILE合法、save_path合法
操作结果：对input_stack进行语法分析。若REDIRECT_STDOUT_TO_FILE为真，则将语法分析过程输出到文件save_path中；
} ADT LR1_parser

3.2 算法描述

3.2.1 计算语法四元组

Algorithm 5 Load grammar

Inputs:

grammar.txt

初始化: $non\_terminals \leftarrow [\,], terminals \leftarrow [\,], productions \leftarrow [\,], start \leftarrow None$
$all\_symbols \leftarrow [\,], cnt \leftarrow 1$
for grammar.txt 中的每一个产生式 **do**
    将产生式左端符号 $append$ 到 $non\_terminals$ 中
    将产生式左端符号 $append$ 到 $all\_symbols$ 中
    initialize $production$
    $production.left \leftarrow$ 产生式左端符号
    $production.right \leftarrow$ 产生式右端符号集
    $production.idx \leftarrow cnt$
    $cnt \leftarrow cnt + 1$
    for 产生式右端符号 i **do**
        将产生式右端符号 i $append$ 到 $all\_symbols$ 中
    end for
end for
$Unique(non\_terminals, terminals)$
$terminals = all\_symbols - non\_terminals$
$start \leftarrow productions[0].left$
初始化 $grammar$
$grammar.non\_terminals \leftarrow non\_terminals$
$grammar.terminals \leftarrow terminals$
$grammar.productions \leftarrow productions$

22: $grammar.start \leftarrow start$
23: **return** $grammar$

3.2.2 求 FIRST 集

Algorithm 6 Calculate First Set

Inputs:

文法 grammar

while FIRST 仍在增大 **do**
  for N in $grammar.non\_terminals$ **do**
    $First[N] \leftarrow [\,]$
  end for
  for T in $grammar.terminals$ **do**
    $First[T] \leftarrow [T]$
  end for
  for P in $grammar.productions$ **do**
    if $P.right[0]$ in $grammar.erminals$ **then**
      $First[P.left].append(P.right[0])$
    end if
    if $P.right = [\,]$ **then**
      $First[P.left].append(\varepsilon)$
    end if
    if $P.right$ 的左侧含有连续 i 个符号位于 $grammar.non\_terminals$ 中，且对任意 i, $First[i]$ 均含 ε **then**
      $First[P.left].extend(FIRST[i+1] - [\varepsilon])$
    end if
  end for
end while
return $First$

3.2.3 求 FOLLOW 集

Algorithm 7 Calculate Follow Set

Inputs:

文法 grammar, FIRST 集

while FOLLOW 仍在增大 **do**
  for N in $grammar.non\_terminals$ **do**
    $Follow[N] \leftarrow [\,]$
  end for
  $Follow[grammar.start] \leftarrow [\#]$

for P in $grammar.productions$ **do**
  if P 的形式为 $A \to \alpha B \beta$, 其中 A, B 位于 $grammar.non\_terminals$ 中，α, β 为任意符号串 **then**
    $Follow[B].extend(FIRST[\beta] - [\varepsilon])$
  end if
  if P 的形式为 $A \to \alpha B$, 其中 A, B 位于 $grammar.non\_terminals$ 中，α 为任意符号串 **then**
    $Follow[B].extend(Follow[A])$
  end if
  if P 的形式为 $A \to \alpha B \beta$, 其中 A 位于 $grammar.non\_terminals$ 中，α, β 为任意符号串，且 ε 含在 $First[\beta]$ 中 **then**
    $Follow[B].extend(Follow[A])$
  end if
 end for
end while
return $Follow$

3.2.4 求闭包

Algorithm 8 Calculate Closure

Inputs:

文法 grammar, FIRST 集, FOLLOW 集, 项目集 I

$closure[I] \leftarrow [\,]$
for i in I **do**
  $closure[I].append(i)$
end for
while $closure[I]$ 仍在增大 **do**
  for $item$ in $closure[I]$ **do**
    if $item$ 的形式为 $A \to \alpha \cdot B \beta, a$, 其中 A, B 位于 $grammar.non\_terminals$ 中，α, β 为任意符号串,a 位于 $grammar.terminals$ 中 **then**
      for $grammar.productions$ 中所有满足 $P.left = B$ 的 P **do**
        for 所有 b in $grammar.terminals$, 并且 b 位于 $First[\beta a]$ 中 **do**
          $closure[I].append(B \to \cdot\gamma, b)$
        end for
      end for
    end if
  end for
end while
return $closure[I]$

3.2.5 求项目集规范族与 GO

Algorithm 9 Calculate Item Set and GO

Inputs:

文法 grammar, FIRST 集, FOLLOW 集

初始化：$itemset \leftarrow [\,]$
$itemset.append(grammar.start \rightarrow \cdot grammar.right, \#)$
$itemset[0] \leftarrow closure[itemset[0]]$
计算 $itemset[0]$ 可以 GO 到项目集 Is，其中 $GO(itemset[0], A) = closure[J]$, 其中 J =｛任何形如 $[A \rightarrow \alpha B \cdot \beta, a]$ 的项目 $|[A \rightarrow \alpha \cdot B\beta, a]$ 在 $itemset[0]$ 中 ｝. 同时将相应的 GO 关系加入 GO 中。
for I in Is **do**
    $itemset.append(I)$
end for
while itemset 仍在增大 **do**
    for I in $itemset$ **do**
        计算 I 可以 GO 到项目集 Is，其中 $GO(itemset[0], A) = closure[J]$, 其中 J =｛任何形如 $[A \rightarrow \alpha B \cdot \beta, a]$ 的项目 $|[A \rightarrow \alpha \cdot B\beta, a]$ 在 $itemset[0]$ 中 ｝. 同时将相应的 GO 关系加入 GO 中。
        for I in Is AND I 不在 itemset 中 **do**
            $itemset.append(I)$
        end for
    end for
end while
return $itemset$, GO

3.2.6 求 ACTION 表与 GOTO 表

Algorithm 10 Calculate Analysis Table

Inputs:

项目集规范族 itemsets, GO 关系, 文法 grammar

for $itemset \in itemsets$ **do**
    for $I \in itemset$ **do**
        if I 的形式为 $[A->\alpha \cdot a\beta, b]$，且 $Go[I, a] = I_j$，a 在 $grammar.terminal$ 中 **then**
            $action[itemset.idx, a] \leftarrow s_j$
        end if
        if I 的形式为 $[A->\alpha \cdot, a]$ **then**
            $action[itemset.idx, a] \leftarrow r_j$，其中 r_j 为 I 的编号
        end if
        if I 的形式为 $[grammar.start_p-> grammar.start \cdot, \#]$ **then**

$action[itemset.idx, \#] \leftarrow acc$
end if
if 不满足以上所有情况 **then**
抛出错误
end if
end for
end for
for go in GO **do**
$goto[go.idx, go.symbol] \leftarrow go.next$
end for
return $action, goto$

3.2.7 LR1 PARSE 解析

Algorithm 11 Parse

Inputs:

待解析的 $input\_stack$($token$ 序列) 以及 $action, goto$, 文法 $grammar$

初始化：$state\_stack = Stack()$, $symbol\_stack = Stack()$
初始化：$state\_stack.push(0)$, $symbol\_stack.push(\#)$
while $input\_stack$ 不空 **do**
$top\_state = state\_stack.top()$, $top\_symbol = symbol\_stack.top()$
if $top\_symbol = \#$ *or* top_symbol in $grammar.terminal$ **then**
$action\_now \leftarrow action[top\_state, top\_symbol]$
if $action\_now = acc$ **then**
将 top_symbol, top_state 与 acc 等信息打印
解析成功
return
end if
if $action\_now = None$ **then**
将 top_symbol, top_state 与 err 等信息打印
语法错误
return
end if
if $action\_now[0] = s$ **then**
将 top_symbol, top_state 与 $move$ 等信息打印
将 $action\_now.state$ 压入 $state\_stack$, $input\_stack.peak()$ 压入 $symbol\_stack$, 将 $input\_stack$ 栈顶元素弹出
end if
if $action\_now[0] = r$ **then**

```
        将 top_symbol, top_state 与 redution 等信息打印
        按照 action_now.state 对应的 grammar 将 symbol_stack 栈顶几个元素进行规约，并将规约结果压入 symbol_stack
        根据 symbol_stack.top() 与当前状态 top_state 查找 goto 表，将 goto 表中的结果压入 state_stack
    end if
  end if
  if top_symbol in grammar.non_terminals then
    goto_now ← goto[top_state, top_symbol]
    if goto_now = None then
      将 top_symbol, top_state 与 err 等信息打印
      语法错误
      return
    end if
    if goto_now[0] = s then
      将 top_symbol, top_state 与 move 等信息打印
      将 goto_now.state 压入 state_stack，input_stack.peak() 压入 symbol_stack，将 input_stack 栈顶元素弹出
    end if
  end if
end while
```

3.3 LR1 分析表描述

LR1 分析表分为 action 表与 goto 表，其对应的抽象数据类型详见3.1.4，其对应的构造算法详见3.2.6，输出示例详见3.4.3。下面对该分析表结构及其功能进行简要描述：
LR1 分析表主要由两部分组成，分别为 action_table(动作表) 与 goto_table(状态转移表)。

在实现上，action_table 是一个字典，其中 key 为二元元组，第一个元素为分析栈当前状态，第二元素为当前面临输入栈的栈顶符号（该符号的合法值为 grammar 的终结符或代表输入结束的"#" 符号）；value 为此时应该采取的动作。动作类型包括有"s*","r*","acc","err"，其中"*" 表示任意数字。分别对应的动作为：

- "s*"：将当前输入栈栈顶符号放到分析栈（符号栈）栈顶，弹出输入栈栈顶符号，同时进行状态转移，即将"*" 对应的状态放到状态栈栈顶；
- "r*"：利用序号为 * 的产生式对分析栈（符号栈）进行规约操作，具体为：将序号为 * 的产生式右端符号依次从分析栈（符号栈）栈顶弹出，同时在状态栈中弹出其对应的状态；最后将序号为 * 的产生式左端符号放到分析栈（符号栈）栈顶，同时将 goto 表中对应的状态放置到状态栈顶。
- "acc"：规约成功，程序退出。

- “err”：规约出错，程序退出。

goto_table 也是以字典的方式实现，字典的 key 与 value 和 action_table 的设定相同，只不过 key 的第二个元素限定为非终结符。goto 表的动作类型包括有”s*”,”err”，其中“*”表示任意数字。分别对应的动作为：

- “s*”：将当前输入栈栈顶符号放到分析栈（符号栈）栈顶，弹出输入栈栈顶符号，同时进行状态转移，即将”*” 对应的状态放到状态栈栈顶；
- “err”：规约出错，程序退出。

进行语法分析时，首先根据3.2中的算法计算语法对应的分析表，然后分别对状态栈、分析栈与输入栈进行初始化操作（细节详见3.2.7），根据当前栈的状态查询 LR1 分析表进行对应的动作即可。

3.4 输出格式说明

3.4.1 FIRST 集与 FOLLOW 集输出

FIRST 集每一行的输出格式：

[符号（终结符或非终结符）] [=] [符号对应的FIRST集]

输出示例：

```
FIRST:
  !=['!']
  !==['!=']
  #=['#']
  &&=['&&']
  (=['(']
  )=[')']
  *=['*']
  ,=[',']
  -=['-']
  .=['.']
  <=['<']
  <==['<=']
  <=>=['<=>']
  ==['=']
  >=['>']
  >==['>=']
  ALL=['ALL']
  AND=['AND']
  AS=['AS']
  AVG=['AVG']
```

FOLLOW 集每一行的输出格式：

[符号（终结符或非终结符）] [=] [符号对应的FOLLOW集]

输出示例：

```
FOLLOW:
  expressionOrDefault=[')', 'WHERE', ',', '#']
  expressionOrDefaultListRec=[')']
  expressionRec=['#', 'HAVING', 'ORDER_BY', 'UNION', ')']
  expressionRight=['WHERE', '#', 'HAVING', 'ORDER_BY', 'UNION', 'GROUP_BY', 'LEFT', 'RIGHT', ')', 'JOIN', ',']
  expressions=['#', 'HAVING', 'ORDER_BY', 'UNION', ')']
  expressionsWithDefaults=[')']
  expressionsWithDefaultsListRec=['#']
```

3.4.2 项目集规范族输出

项目集规范族中每一个项目集的输出格式：

[itemset] [项目集在项目集规范族中的序号] [:] [\n]
[项目1] [\n]
[项目2] [\n]
.....
[项目n] [\n][\n]

其中，项目的输出格式如下：

[左侧符号] [,] [右侧符号集] [,] [dot位置] [,] [终结符号集] [,] [产生式序号]

输出示例：

```
item_set0:
rootp, ['root'], 0, ['#'], 0,
root, ['dmlStatement'], 0, ['#'], 1,
dmlStatement, ['selectStatement'], 0, ['#'], 2,
dmlStatement, ['insertStatement'], 0, ['#'], 3,
dmlStatement, ['updateStatement'], 0, ['#'], 4,
dmlStatement, ['deleteStatement'], 0, ['#'], 5,
selectStatement, ['querySpecification', 'unionStatements'], 0, ['#'], 6,
insertStatement, ['insertKeyword', 'tableName', 'insertStatementRight'], 0, ['#'], 112,
updateStatement, ['UPDATE', 'tableName', 'elementNameAlias', 'SET', 'updatedElement', 'updatedElementL
deleteStatement, ['DELETE', 'FROM', 'tableName', 'deleteStatementRight'], 0, ['#'], 135,
querySpecification, ['SELECT', 'unionType', 'selectElements', 'selectClause'], 0, ['#', 'UNION'], 15,
querySpecification, ['(', 'querySpecification', ')'], 0, ['#', 'UNION'], 16,
insertKeyword, ['INSERT', 'into'], 0, ['IDN'], 115,

item_set1:
querySpecification, ['(', 'querySpecification', ')'], 1, ['#', 'UNION'], 16,
querySpecification, ['SELECT', 'unionType', 'selectElements', 'selectClause'], 0, [')'], 15,
querySpecification, ['(', 'querySpecification', ')'], 0, [')'], 16,
```

项目集规范族各项目集之间的转移关系 go 的输出格式：

[(][(][当前项目集序号][,][面临符号][)][,][转移后的项目集序号][)]

输出示例：

```
((0, '('), 1)
((0, 'DELETE'), 2)
((0, 'INSERT'), 3)
((0, 'SELECT'), 4)
((0, 'UPDATE'), 5)
((0, 'deleteStatement'), 6)
((0, 'dmlStatement'), 7)
((0, 'insertKeyword'), 8)
((0, 'insertStatement'), 9)
((0, 'querySpecification'), 10)
((0, 'root'), 11)
((0, 'selectStatement'), 12)
((0, 'updateStatement'), 13)
((1, '('), 14)
((1, 'SELECT'), 15)
((1, 'querySpecification'), 16)
```

3.4.3 LR1 分析表输出

LR1 分析表包括 action 表和 goto 表两个部分，二者分别输出。

其中：

action 表输出格式：

```
[(] [当前状态] [,] [面临输入符号] [)] [  ----->  ] [动作类型] [动作值]
```

在 action 表中，面临输入符号限定为非终结符，动作类型的可选值为“s”、“r”、“acc”、“None”（表示不存在该动作，与“err”等价）。

输出示例：

```
action table:
  (0, '!')  ----->  None
  (0, '!=')  ----->  None
  (0, '#')  ----->  None
  (0, '&&')  ----->  None
  (0, '(')  ----->  s1
  (0, ')')  ----->  None
  (0, '*')  ----->  None
  (0, ',')  ----->  None
  (0, '-')  ----->  None
  (0, '.')  ----->  None
  (0, '<')  ----->  None
  (0, '<=')  ----->  None
  (0, '<=>')  ----->  None
  (0, '=')  ----->  None
  (0, '>')  ----->  None
  (0, '>=')  ----->  None
  (0, 'ALL')  ----->  None
  (0, 'AND')  ----->  None
  (0, 'AS')  ----->  None
  (0, 'AVG')  ----->  None
  (0, 'DEFAULT')  ----->  None
  (0, 'DELETE')  ----->  s2
  (0, 'DISTINCT')  ----->  None
```

goto 表输出格式：

[(] [当前状态] [,] [面临输入符号] [)] [---->] [动作类型] [动作值]

在 goto 表中，面临输入符号限定为终结符，动作类型的可选值：“s”、“None”（表示不存在该动作，与“err”等价）。输出示例：

```
goto table:
  (0, 'aggregateWindowedFunction') ----> None
  (0, 'booleanLiteral') ----> None
  (0, 'comparisonOperator') ----> None
  (0, 'constant') ----> None
  (0, 'decimalLiteral') ----> None
  (0, 'deleteStatement') ----> s6
  (0, 'deleteStatementRight') ----> None
  (0, 'dmlStatement') ----> s7
```

3.4.4 规约序列输出

规约序列每一行的输出格式：

[序号] [TAB] [选用规则序号] [TAB] [栈顶符号]#[面临输入符号] [TAB] [执行动作]

其中，选用规则序号见附件文法规则；执行动作为“reduction”（归约），“move”（LR 分析的移进），“accept”（接受）或“error”（错误）输出示例：

```
1  1   /   ##INSERT     move
2  2   /   INSERT#INTO  move
3  3   116 INTO#IDN     reduction
4  4   115 into#IDN     reduction
5  5   /   insertKeyword#IDN   move
6  6   48  IDN#(    reduction
7  7   44  uid#(    reduction
8  8   /   tableName#( move
9  9   /   (#IDN     move
10 10  48  IDN#,    reduction
11 11  /   uid#,    move
12 12  /   ,#IDN    move
13 13  48  IDN#,    reduction
14 14  /   uid#,    move
15 15  /   ,#IDN    move
16 16  48  IDN#)    reduction
17 17  47  uid#)    reduction
18 18  46  uidListRec#)     reduction
19 19  46  uidListRec#)     reduction
20 20  45  uidListRec#)     reduction
21 21  /   uidList#)   move
22 22  /   )#VALUES    move
23 23  119 VALUES#(    reduction
24 24  /   insertFormat#(  move
25 25  /   (#STRING    move
```

3.5 源程序编译步骤

本项目的 LR1 语法分析器采用 python 编写，无须编译，可以直接运行。其中，代码对环境的要求如下：

```
python3
matplotlib==3.4.3
pickle>=0.75
```

在工程根目录终端下，键入如下命令即可快速运行：

```
python main.py
```

其中，main.py 文件的输入为./tests 目录下的”testcase-*.sql” 文件，通过词法分析与语法分析分别输出 token 序列和规约序列。

若要单独查看与调试 LR1 语法分析器本身的功能，可以在工程根目录下键入以下命令：

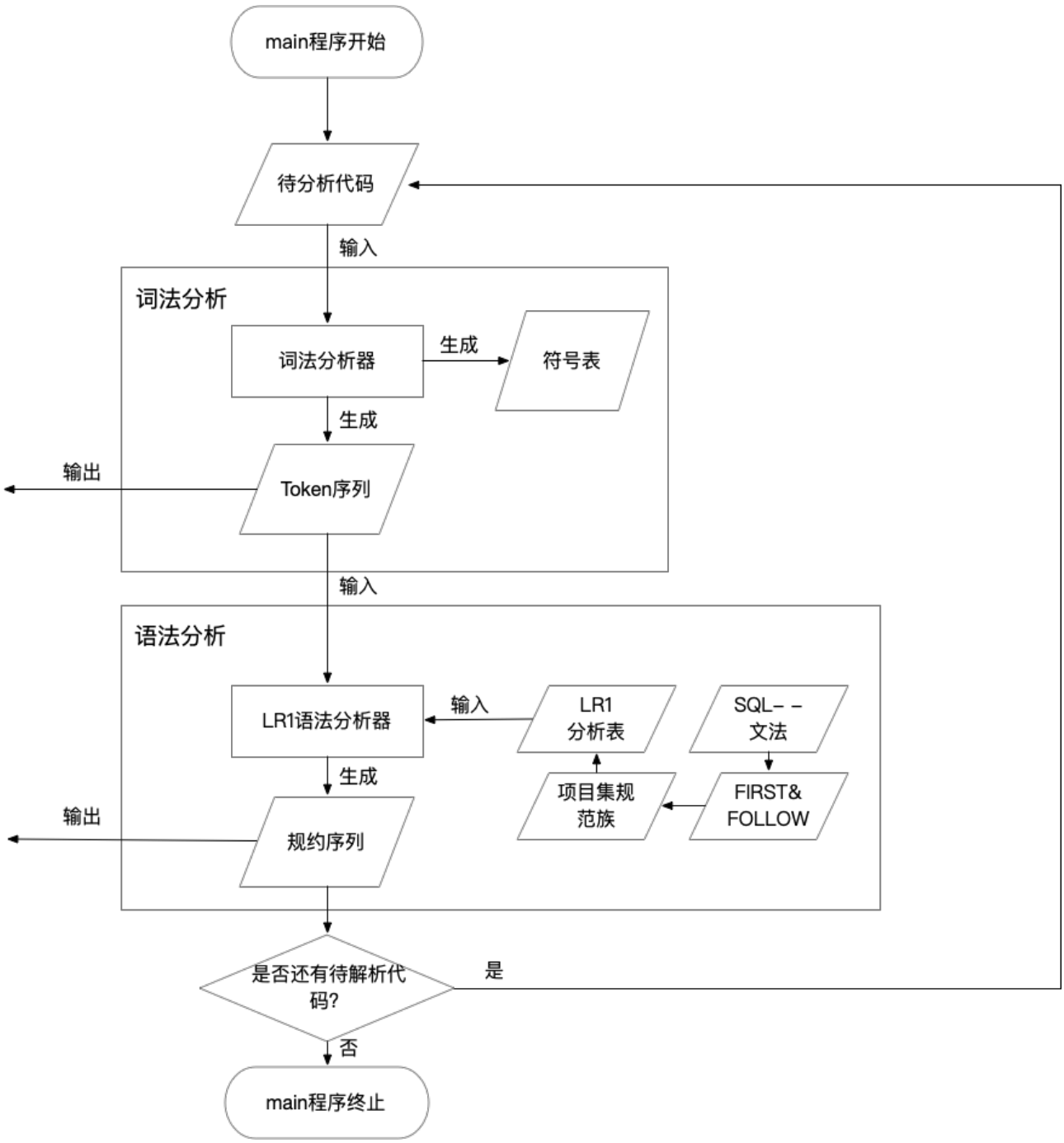
```
python ./src/LR1_parser/LR1.py
```

若要查看与调试 LR1 语法分析器子模块 X (可选项为 grammar，FF，item，analysisTable) 的功能，可以在工程根目录下键入以下命令：

```
python ./src/LR1_parser/ds/X.py
```

4 主程序分析流程

main 程序运行后，首先读取位于./tests 文件夹中的测试样例，然后依次调用词法分析器与语法分析器对测试样例进行分析。采用多遍编译的方法。支持连续测试多个测试样例并依次输出。相应流程图如下：



在使用前，通过修改 Configs 文件中的相应配置项来设置主程序的输入文法、输入测试样例、输出位置等参数，尔后在工程根目录终端下，键入如下命令即可快速运行：

```
python main.py
```