Compiler Design Lab Project 9
BCSE-III, 2022-23

Group Members:

Roll	Name
002010501035	Kanko Ghosh
002010501036	Mahfujul Haque
002010501041	Rounak Bhattacharjee
002010501068	Soumyajit Rudra Sarma

Jadavpur University, Kolkata

Contents

A. Problem Statement

Language Specification:

In this project, we are trying to parse and validate SQL type DML language supporting

- **Insertion and deletion statements**
    - Support for insertion of data having all attribute values given as well as partial data given (also checking is done if number of attributes is same or not on both sides)
- **Select statements**
    - **where** clause
    - **Group by** - having clause
    - **Order by** clause
- **Conditions and Relation Operators**
    - **AND, OR, NOT**
    - **<=, <, >, >=, =** operation
- **tablename.attribute** notation also permitted

Note: *Nested SQL select statements were not validated (because it was asked so in the question)*

Objectives

- Construction of a CFG for this language.
- Designing a lexical analyser to scan the stream of characters from a SQL query for generating a stream of tokens.
- Designing a top-down parser to detect syntax errors in the SQL queries (modules include FIRST, FOLLOW, parsing table construction and parsing).

Programming Language Used: **C++**

Other Tools: Python (For Graph generation, using igraph library)

B. Modules used in project

- **DFA**
  - <u>**Input**</u>**: Regular expressions** along with **token names** (input_lex.txt)
  - <u>**Output**</u>:
    - **Master Syntax tree** along with annotations (**nullable**, **firstpos**, **lastpos** and **followpos**) corresponding to the DFA (used in intermediate stage to generate the DFA)
    - **DFA Definition** with map of final states to token names (with aliases referring to the tree nodes)

- **Tokenizer**
  - <u>**Input**</u>**: DFA Definitions** along with **input SQL Code** (input_text.txt)
  - <u>**Output**</u>: **Token Stream** (input_tokens.txt) in the form of **token name** and corresponding **lexeme value** (Eg. `<attribute, ID>` and `<\n, SEP>`)

- **LL(1) Parsing Table**
  - <u>**Input**</u>**: Productions** (input_prod.txt)
  - <u>**Output**</u>:
    - First, nullable and follow for each non-terminal symbols (intermediate stage to generate the parsing table)
    - **Augmented LL(1) Parsing Table** (includes **pop** and **scan** operations in empty states of initial LL(1) parsing table)

- **LL(1) Parser as validator**
  - <u>**Input**</u>**:**
    - **Token stream** (input_tokens.txt) generated by the tokenizer
    - Augmented LL(1) parsing table
  - <u>**Output**</u>: Checking if SQL code is **valid** or not, along with **parsing history** (parsing stack, remaining token stream and action)

C. Lex Definition and Grammar

- `Lex Definitions (input_lex.txt):`

Regular Expression	Token
DELETE	`DELETE`
INTO	`INTO`
INSERT	`INSERT`
>=	`>=`
>	`>`
=	`=`
<=	`<=`
<	`<`
\\)	`)`
\\(	`(`
VALUES	`VALUES`
\\.	`.`
;	`;`
BY	`BY`
OR	`OR`
SELECT	`SELECT`
WHERE	`WHERE`
GROUP	`GROUP`
ORDER	`ORDER`
HAVING	`HAVING`
AND	`AND`

FROM	FROM
,	,
NOT	NOT
\n+\t+	SEP
(a+b+c+d+e+f+g+h+i+j+k+l+m+n+o+p+q+r+s+t+u+v+w+x+y+z)(1+2+3+4+5+6+7+8+9+0+a+b+c+d+e+f+g+h+i+j+k+l+m+n+o+p+q+r+s+t+u+v+w+x+y+z)*	ID

- **<u>Productions (input_prod.txt)</u>:**

1	s	->	stmt s; // s is the starting symbol
2	s	->	E
3	stmt	->	selst
4	stmt	->	insst
5	stmt	->	delst
6	ide	->	ID ide'
7	ide'	->	. ID
8	ide'	->	E
9	selst	->	SELECT idlste FROM idlist wherecl grpbycl ordercl
10	idlist	->	ID idlnul
11	idlnul	->	, ID idlnul
12	idlnul	->	E
13	idlste	->	ide idlnle
14	idlnle	->	, ide idlnle
15	idlnle	->	E
16	wherecl	->	WHERE cond
17	wherecl	->	E
18	grpbycl	->	GROUP BY idlste havngcl
19	grpbycl	->	E
20	havngcl	->	HAVING cond
21	havngcl	->	E
22	ordercl	->	ORDER BY idlste
23	ordercl	->	E
24	cond	->	unitcd cond'
25	cond'	->	cdjoin unitcd cond'
26	cond'	->	E
27	cdjoin	->	AND
28	cdjoin	->	OR
29	unitcd	->	NOT (unitcd)

```
30  unitcd     ->    ide compop ide
31  compop     ->    =
32  compop     ->    >
33  compop     ->    <
34  compop     ->    >=
35  compop     ->    <=
36  insst      ->    INSERT INTO ID insst1
37  insst1     ->    VALUES ( idlist )
38  insst1     ->    ( ID insst' ID )
39  insst'     ->    , ID insst' ID ,
40  insst'     ->    ) VALUES (
41  delst      ->    DELETE FROM ID wherecl
```

Note: *Since this project contained completely decoupled modules like DFA generator and LL(1) parsing table creator, any other regex and grammar can be used as input for parsing another language*

D. Input and Output Files

TEST CASE 1 ⇒

- **Input**

```
INSERT INTO mytable(atr1 ,atr2 ,atr3 ,atr4) VALUES ( bren , pro , op , super );
INSERT INTO mytable2 VALUES ( bruh2 , pro2 , op2 , super2 );
INSERT INTO mytable3 (atr1 ,atr2 ,atr3 ,atr4) VALUES ( bruh , pro , op , super );

DELETE FROM mytable;
DELETE FROM mytable WHERE age >= agetemp AND name = myname OR pro < toopro;


SELECT atr, op FROM gtbl;
SELECT atr, op FROM gtbl, btbl;
SELECT atr, gtbl.op FROM gtbl;
SELECT gtbl.atr, btbl.op FROM gtbl, btbl;

SELECT gtbl.atr, btbl.op FROM gtbl, btbl WHERE gtbl.atr >= winatr OR btbl.op = greatop;
SELECT gtbl.atr, btbl.op FROM gtbl, btbl GROUP BY btbl.op;
SELECT gtbl.atr, btbl.op FROM gtbl, btbl GROUP BY btbl.op HAVING btbl.op > opwin;
SELECT gtbl.atr, btbl.op FROM gtbl, btbl WHERE gtbl.atr >= winatr OR btbl.op = greatop
GROUP BY btbl.op;
SELECT gtbl.atr, btbl.op FROM gtbl, btbl WHERE gtbl.atr >= winatr OR btbl.op = greatop
GROUP BY btbl.op HAVING btbl.op > opwin;
SELECT gtbl.atr, btbl.op FROM gtbl, btbl ORDER BY gtbl.atr;
SELECT gtbl.atr, btbl.op FROM gtbl, btbl WHERE gtbl.atr >= winatr OR btbl.op = greatop
ORDER BY gtbl.atr;
SELECT gtbl.atr, btbl.op FROM gtbl, btbl GROUP BY btbl.op ORDER BY gtbl.atr;
SELECT gtbl.atr, btbl.op FROM gtbl, btbl GROUP BY btbl.op HAVING btbl.op > opwin ORDER BY
gtbl.atr;
SELECT gtbl.atr, btbl.op FROM gtbl, btbl WHERE gtbl.atr >= winatr OR btbl.op = greatop
GROUP BY btbl.op ORDER BY gtbl.atr;
SELECT gtbl.atr, btbl.op FROM gtbl, btbl WHERE gtbl.atr >= winatr OR btbl.op = greatop
GROUP BY btbl.op HAVING btbl.op > opwin ORDER BY gtbl.atr;


INSERT INTO dual VALUES(atr); INSERT INTO dual VALUES(atr);INSERT INTO dual VALUES(atr);
INSERT INTO dual VALUES(atr);
```

- **Output**

```
READ TOKEN STREAM -->
[{T(INSERT): INSERT}, {T(INTO): INTO}, {T(ID): mytable}, ... ,  {T(;): ;}, {T($): $}]
___________________________________________________________________________

PARSING RESULT: TRUE
PARSING ERROR INFO: No Error

Parsing Stack: S(s), T($)
Token Stream: {T(INSERT): INSERT}, ..., {T(;): ;}, {T($): $}
Action: A(PRODUCE[ S(s) -> S(stmt) T(;) S(s) ])

Parsing Stack: S(stmt), T(;), S(s), T($)
Token Stream: {T(INSERT): INSERT}, ..., {T($): $}
Action: A(PRODUCE[ S(stmt) -> S(insst) ])

Parsing Stack: S(insst), T(;), S(s), T($)
Token Stream: {T(INSERT): INSERT}, ..., {T($): $}
Action: A(PRODUCE[ S(insst) -> T(INSERT) T(INTO) T(ID) S(insst1) ])

Parsing Stack: T(INSERT), T(INTO), T(ID), S(insst1), T(;), S(s), T($)
Token Stream: {T(INSERT): INSERT}, ..., {T($): $}
Action: A(MATCH_TOKEN)

Parsing Stack: T(INTO), T(ID), S(insst1), T(;), S(s), T($)
Token Stream: {T(INTO): INTO}, ..., {T($): $}
Action: A(MATCH_TOKEN)

...
...
...

Parsing Stack: T(;), S(s), T($)
Token Stream: {T(;): ;}, {T($): $}
Action: A(MATCH_TOKEN)

Parsing Stack: S(s), T($)
Token Stream: {T($): $}
Action: A(PRODUCE[ S(s) -> T(E) ])

Parsing Stack: T($)
Token Stream: {T($): $}
Action: A(MATCH_TOKEN)
```

TEST CASE 2 ⇒

- **Input**

```
INSERT INTO mytable(atr1 ,atr2 ,atr3) VALUES ( bren , pro , op , super );
```

- **Output**

```
READ TOKEN STREAM -->
[{T(INSERT): INSERT}, {T(INTO): INTO}, ..., {T()): )}, {T(;): ;}, {T($): $}]
_____________________________________________________________________________

PARSING RESULT: FALSE
PARSING ERROR INFO: Fatal Error

Parsing Stack: S(s), T($)
Token Stream: {T(INSERT): INSERT}, ..., {T(;): ;}, {T($): $}
Action: A(PRODUCE[ S(s) -> S(stmt) T(;) S(s) ])

Parsing Stack: S(stmt), T(;), S(s), T($)
Token Stream: {T(INSERT): INSERT}, ..., {T(;): ;}, {T($): $}
Action: A(PRODUCE[ S(stmt) -> S(insst) ])

Parsing Stack: S(insst), T(;), S(s), T($)
Token Stream: {T(INSERT): INSERT}, ..., {T(;): ;}, {T($): $}
Action: A(PRODUCE[ S(insst) -> T(INSERT) T(INTO) T(ID) S(insst1) ])

Parsing Stack: T(INSERT), T(INTO), T(ID), S(insst1), T(;), S(s), T($)
Token Stream: {T(INSERT): INSERT}, ..., {T(;): ;}, {T($): $}
Action: A(MATCH_TOKEN)

Parsing Stack: T(INTO), T(ID), S(insst1), T(;), S(s), T($)
Token Stream: {T(INTO): INTO}, ..., {T(;): ;}, {T($): $}
Action: A(MATCH_TOKEN)

...
...
...

Parsing Stack: T(,), T(ID), T()), T(;), S(s), T($)
Token Stream: {T(,): ,}, {T(ID): op}, {T(,): ,}, {T(ID): super}, {T()): )}, {T(;): ;},
{T($): $}
Action: A(MATCH_TOKEN)

Parsing Stack: T(ID), T()), T(;), S(s), T($)
Token Stream: {T(ID): op}, {T(,): ,}, {T(ID): super}, {T()): )}, {T(;): ;}, {T($): $}
Action: A(MATCH_TOKEN)
```

```
Parsing Stack: T()), T(;), S(s), T($)
Token Stream: {T(,): ,}, {T(ID): super}, {T()): )}, {T(;): ;}, {T($): $}
Action: A(INVALID)
```

TEST CASE 3 ⇒

- **Input**

```
SELECT gtbl.atr, btbl.op FROM gtbl, btbl HAVING btbl.op > opwin;
```

- **Output**

```
READ TOKEN STREAM -->
[{T(SELECT): SELECT}, {T(ID): gtbl}, {T(.): .}, ..., {T(;): ;}, {T($): $}]
__________________________________________________________________________


PARSING RESULT: FALSE
PARSING ERROR INFO: Recovered Error

Parsing Stack: S(s), T($)
Token Stream: {T(SELECT): SELECT}, ..., {T(;): ;}, {T($): $}
Action: A(PRODUCE[ S(s) -> S(stmt) T(;) S(s) ])

Parsing Stack: S(stmt), T(;), S(s), T($)
Token Stream: {T(SELECT): SELECT}, ..., {T(;): ;}, {T($): $}
Action: A(PRODUCE[ S(stmt) -> S(selst) ])

Parsing Stack: S(selst), T(;), S(s), T($)
Token Stream: {T(SELECT): SELECT}, ..., {T(;): ;}, {T($): $}
Action: A(PRODUCE[ S(selst) -> T(SELECT) S(idlste) T(FROM) S(idlist) S(wherecl) S(grpbycl)
S(ordercl) ])

Parsing Stack: T(SELECT), S(idlste), T(FROM), S(idlist), S(wherecl), S(grpbycl),
S(ordercl), T(;), S(s), T($)
Token Stream: {T(SELECT): SELECT}, ..., {T(;): ;}, {T($): $}
Action: A(MATCH_TOKEN)

Parsing Stack: S(idlste), T(FROM), S(idlist), S(wherecl), S(grpbycl), S(ordercl), T(;),
S(s), T($)
Token Stream: {T(ID): gtbl}, ..., {T(;): ;}, {T($): $}
Action: A(PRODUCE[ S(idlste) -> S(ide) S(idlnle) ])


...
...
...
```

```
Parsing Stack: T(ID), S(idlnul), S(wherecl), S(grpbycl), S(ordercl), T(;), S(s), T($)
Token Stream: {T(ID): btbl}, {T(HAVING): HAVING}, {T(ID): btbl}, {T(.): .}, {T(ID): op},
{T(>): >}, {T(ID): opwin}, {T(;): ;}, {T($): $}
Action: A(MATCH_TOKEN)

Parsing Stack: S(idlnul), S(wherecl), S(grpbycl), S(ordercl), T(;), S(s), T($)
Token Stream: {T(HAVING): HAVING}, {T(ID): btbl}, {T(.): .}, {T(ID): op}, {T(>): >},
{T(ID): opwin}, {T(;): ;}, {T($): $}
Action: A(SCAN)

Parsing Stack: S(idlnul), S(wherecl), S(grpbycl), S(ordercl), T(;), S(s), T($)
Token Stream: {T(ID): btbl}, {T(.): .}, {T(ID): op}, {T(>): >}, {T(ID): opwin}, {T(;): ;},
{T($): $}
Action: A(SCAN)

...
...
...

Parsing Stack: T(;), S(s), T($)
Token Stream: {T(;): ;}, {T($): $}
Action: A(MATCH_TOKEN)

Parsing Stack: S(s), T($)
Token Stream: {T($): $}
Action: A(PRODUCE[ S(s) -> T(E) ])

Parsing Stack: T($)
Token Stream: {T($): $}
Action: A(MATCH_TOKEN)
```

E. Log File

Note: *The Log file corresponds to the DFA creation, LL(1) parser creation, and the first input test case.*

- **Concat Processed Lex Definition:**

```
#Registered Tokens: 26

[ {D , C} ,{. , O} ,{E , C} ,{. , O} ,{L , C} ,{. , O} ,{E , C} ,{. , O} ,{T , C} ,{. , O}
,{E , C} ]  -->  DELETE
[ {I , C} ,{. , O} ,{N , C} ,{. , O} ,{T , C} ,{. , O} ,{O , C} ]  -->  INTO
[ {I , C} ,{. , O} ,{N , C} ,{. , O} ,{S , C} ,{. , O} ,{E , C} ,{. , O} ,{R , C} ,{. , O}
,{T , C} ]  -->  INSERT
[ {> , C} ,{. , O} ,{= , C} ]  -->  >=
[ {> , C} ]  -->  >
[ {= , C} ]  -->  =
[ {< , C} ,{. , O} ,{= , C} ]  -->  <=
[ {< , C} ]  -->  <
[ {) , C} ]  -->  )
[ {( , C} ]  -->  (
[ {V , C} ,{. , O} ,{A , C} ,{. , O} ,{L , C} ,{. , O} ,{U , C} ,{. , O} ,{E , C} ,{. , O}
,{S , C} ]  -->  VALUES
[ {. , C} ]  -->  .
[ {; , C} ]  -->  ;
[ {B , C} ,{. , O} ,{Y , C} ]  -->  BY
[ {O , C} ,{. , O} ,{R , C} ]  -->  OR
[ {S , C} ,{. , O} ,{E , C} ,{. , O} ,{L , C} ,{. , O} ,{E , C} ,{. , O} ,{C , C} ,{. , O}
,{T , C} ]  -->  SELECT
[ {W , C} ,{. , O} ,{H , C} ,{. , O} ,{E , C} ,{. , O} ,{R , C} ,{. , O} ,{E , C} ]  -->
WHERE
[ {G , C} ,{. , O} ,{R , C} ,{. , O} ,{O , C} ,{. , O} ,{U , C} ,{. , O} ,{P , C} ]  -->
GROUP
[ {O , C} ,{. , O} ,{R , C} ,{. , O} ,{D , C} ,{. , O} ,{E , C} ,{. , O} ,{R , C} ]  -->
ORDER
[ {H , C} ,{. , O} ,{A , C} ,{. , O} ,{V , C} ,{. , O} ,{I , C} ,{. , O} ,{N , C} ,{. , O}
,{G , C} ]  -->  HAVING
[ {A , C} ,{. , O} ,{N , C} ,{. , O} ,{D , C} ]  -->  AND
[ {F , C} ,{. , O} ,{R , C} ,{. , O} ,{O , C} ,{. , O} ,{M , C} ]  -->  FROM
[ {, , C} ]  -->  ,
[ {N , C} ,{. , O} ,{O , C} ,{. , O} ,{T , C} ]  -->  NOT
[ {\n , C} ,{+ , O} ,{\t , C} ,{+ , O} ,{  , C} ]  -->  SEP
[ {( , O} ,{a , C} ,{+ , O} ,{b , C} ,{+ , O} ,{c , C} ,{+ , O} ,{d , C} ,{+ , O} ,{e , C}
,{+ , O} ,{f , C} ,{+ , O} ,{g , C} ,{+ , O} ,{h , C} ,{+ , O} ,{i , C} ,{+ , O} ,{j , C}
,{+ , O} ,{k , C} ,{+ , O} ,{l , C} ,{+ , O} ,{m , C} ,{+ , O} ,{n , C} ,{+ , O} ,{o , C}
,{+ , O} ,{p , C} ,{+ , O} ,{q , C} ,{+ , O} ,{r , C} ,{+ , O} ,{s , C} ,{+ , O} ,{t , C}
,{+ , O} ,{u , C} ,{+ , O} ,{v , C} ,{+ , O} ,{w , C} ,{+ , O} ,{x , C} ,{+ , O} ,{y , C}
,{+ , O} ,{z , C} ,{) , O} ,{. , O} ,{( , O} ,{1 , C} ,{+ , O} ,{2 , C} ,{+ , O} ,{3 , C}
,{+ , O} ,{4 , C} ,{+ , O} ,{5 , C} ,{+ , O} ,{6 , C} ,{+ , O} ,{7 , C} ,{+ , O} ,{8 , C}
,{+ , O} ,{9 , C} ,{+ , O} ,{0 , C} ,{+ , O} ,{a , C} ,{+ , O} ,{b , C} ,{+ , O} ,{c , C}
,{+ , O} ,{d , C} ,{+ , O} ,{e , C} ,{+ , O} ,{f , C} ,{+ , O} ,{g , C} ,{+ , O} ,{h , C}
,{+ , O} ,{i , C} ,{+ , O} ,{j , C} ,{+ , O} ,{k , C} ,{+ , O} ,{l , C} ,{+ , O} ,{m , C}
,{+ , O} ,{n , C} ,{+ , O} ,{o , C} ,{+ , O} ,{p , C} ,{+ , O} ,{q , C} ,{+ , O} ,{r , C}
,{+ , O} ,{s , C} ,{+ , O} ,{t , C} ,{+ , O} ,{u , C} ,{+ , O} ,{v , C} ,{+ , O} ,{w , C}
,{+ , O} ,{x , C} ,{+ , O} ,{y , C} ,{+ , O} ,{z , C} ,{) , O} ,{* , O} ]  -->  ID
```

- **Postfix Processed Lex Definition:**

```
#Registered Tokens: 26

[ {D , C} ,{E , C} ,{. , O} ,{L , C} ,{. , O} ,{E , C} ,{. , O} ,{T , C} ,{. , O} ,{E , C}
,{. , O} ]  -->  DELETE
[ {I , C} ,{N , C} ,{. , O} ,{T , C} ,{. , O} ,{O , C} ,{. , O} ]  -->  INTO
[ {I , C} ,{N , C} ,{. , O} ,{S , C} ,{. , O} ,{E , C} ,{. , O} ,{R , C} ,{. , O} ,{T , C}
,{. , O} ]  -->  INSERT
[ {> , C} ,{= , C} ,{. , O} ]  -->  >=
[ {> , C} ]  -->  >
[ {= , C} ]  -->  =
[ {< , C} ,{= , C} ,{. , O} ]  -->  <=
[ {< , C} ]  -->  <
[ {) , C} ]  -->  )
[ {( , C} ]  -->  (
[ {V , C} ,{A , C} ,{. , O} ,{L , C} ,{. , O} ,{U , C} ,{. , O} ,{E , C} ,{. , O} ,{S , C}
,{. , O} ]  -->  VALUES
[ {. , C} ]  -->  .
[ {; , C} ]  -->  ;
[ {B , C} ,{Y , C} ,{. , O} ]  -->  BY
[ {O , C} ,{R , C} ,{. , O} ]  -->  OR
[ {S , C} ,{E , C} ,{. , O} ,{L , C} ,{. , O} ,{E , C} ,{. , O} ,{C , C} ,{. , O} ,{T , C}
,{. , O} ]  -->  SELECT
[ {W , C} ,{H , C} ,{. , O} ,{E , C} ,{. , O} ,{R , C} ,{. , O} ,{E , C} ,{. , O} ]  -->
WHERE
[ {G , C} ,{R , C} ,{. , O} ,{O , C} ,{. , O} ,{U , C} ,{. , O} ,{P , C} ,{. , O} ]  -->
GROUP
[ {O , C} ,{R , C} ,{. , O} ,{D , C} ,{. , O} ,{E , C} ,{. , O} ,{R , C} ,{. , O} ]  -->
ORDER
[ {H , C} ,{A , C} ,{. , O} ,{V , C} ,{. , O} ,{I , C} ,{. , O} ,{N , C} ,{. , O} ,{G , C}
,{. , O} ]  -->  HAVING
[ {A , C} ,{N , C} ,{. , O} ,{D , C} ,{. , O} ]  -->  AND
[ {F , C} ,{R , C} ,{. , O} ,{O , C} ,{. , O} ,{M , C} ,{. , O} ]  -->  FROM
[ {, , C} ]  -->  ,
[ {N , C} ,{O , C} ,{. , O} ,{T , C} ,{. , O} ]  -->  NOT
[ {\n , C} ,{\t , C} ,{+ , O} ,{  , C} ,{+ , O} ]  -->  SEP
[ {a , C} ,{b , C} ,{+ , O} ,{c , C} ,{+ , O} ,{d , C} ,{+ , O} ,{e , C} ,{+ , O} ,{f , C}
,{+ , O} ,{g , C} ,{+ , O} ,{h , C} ,{+ , O} ,{i , C} ,{+ , O} ,{j , C} ,{+ , O} ,{k , C}
,{+ , O} ,{l , C} ,{+ , O} ,{m , C} ,{+ , O} ,{n , C} ,{+ , O} ,{o , C} ,{+ , O} ,{p , C}
,{+ , O} ,{q , C} ,{+ , O} ,{r , C} ,{+ , O} ,{s , C} ,{+ , O} ,{t , C} ,{+ , O} ,{u , C}
,{+ , O} ,{v , C} ,{+ , O} ,{w , C} ,{+ , O} ,{x , C} ,{+ , O} ,{y , C} ,{+ , O} ,{z , C}
,{+ , O} ,{1 , C} ,{2 , C} ,{+ , O} ,{3 , C} ,{+ , O} ,{4 , C} ,{+ , O} ,{5 , C} ,{+ , O}
,{6 , C} ,{+ , O} ,{7 , C} ,{+ , O} ,{8 , C} ,{+ , O} ,{9 , C} ,{+ , O} ,{0 , C} ,{+ , O}
,{a , C} ,{+ , O} ,{b , C} ,{+ , O} ,{c , C} ,{+ , O} ,{d , C} ,{+ , O} ,{e , C} ,{+ , O}
,{f , C} ,{+ , O} ,{g , C} ,{+ , O} ,{h , C} ,{+ , O} ,{i , C} ,{+ , O} ,{j , C} ,{+ , O}
,{k , C} ,{+ , O} ,{l , C} ,{+ , O} ,{m , C} ,{+ , O} ,{n , C} ,{+ , O} ,{o , C} ,{+ , O}
,{p , C} ,{+ , O} ,{q , C} ,{+ , O} ,{r , C} ,{+ , O} ,{s , C} ,{+ , O} ,{t , C} ,{+ , O}
,{u , C} ,{+ , O} ,{v , C} ,{+ , O} ,{w , C} ,{+ , O} ,{x , C} ,{+ , O} ,{y , C} ,{+ , O}
,{z , C} ,{+ , O} ,{* , O} ,{. , O} ]  -->  ID
```

- **Master Syntax Tree**

```
Pre-Order Traversal ->  [ + [ + [ + [ + [ + [ + [ + [ + [ + [ + [ + [ + [ + [ + [ + [ + [ +
[ + [ + [ + [ + [ + [ + [ + [ + [ . [ . [ . [ . [ . [ . [ D ] [ E ] ] [ L ] ] [ E ] ] [ T ]
] [ E ] ] [ # ] ] [ . [ . [ . [ . [ I ] [ N ] ] [ T ] ] [ O ] ] [ # ] ] ] [ . [ . [ . [ . [
. [ . [ I ] [ N ] ] [ S ] ] [ E ] ] [ R ] ] [ T ] ] [ # ] ] ] [ . [ . [ > ] [ = ] ] [ # ] ]
] [ . [ > ] [ # ] ] ] [ . [ = ] [ # ] ] ] [ . [ . [ < ] [ = ] ] [ # ] ] ] [ . [ < ] [ # ] ]
] [ . [ ) ] [ # ] ] ] [ . [ ( ] [ # ] ] ] [ . [ . [ . [ . [ . [ . [ V ] [ A ] ] [ L ] ] [ U
] ] [ E ] ] [ S ] ] [ # ] ] ] [ . [ . ] [ # ] ] ] [ . [ ; ] [ # ] ] ] [ . [ . [ B ] [ Y ] ]
[ # ] ] ] [ . [ . [ O ] [ R ] ] [ # ] ] ] [ . [ . [ . [ . [ . [ . [ S ] [ E ] ] [ L ] ] [ E
] ] [ C ] ] [ T ] ] [ # ] ] ] [ . [ . [ . [ . [ . [ W ] [ H ] ] [ E ] ] [ R ] ] [ E ] ] [ #
] ] ] [ . [ . [ . [ . [ . [ G ] [ R ] ] [ O ] ] [ U ] ] [ P ] ] [ # ] ] ] [ . [ . [ . [ . [
. [ O ] [ R ] ] [ D ] ] [ E ] ] [ R ] ] [ # ] ] ] [ . [ . [ . [ . [ . [ . [ H ] [ A ] ] [ V
] ] [ I ] ] [ N ] ] [ G ] ] [ # ] ] ] [ . [ . [ . [ A ] [ N ] ] [ D ] ] [ # ] ] ] [ . [ . [
. [ . [ F ] [ R ] ] [ O ] ] [ M ] ] [ # ] ] ] [ . [ , ] [ # ] ] ] [ . [ . [ . [ N ] [ O ] ]
[ T ] ] [ # ] ] ] [ . [ + [ + [ \n ] [ \t ] ] [   ] ] [ # ] ] ] [ . [ . [ + [ + [ + [ + [ +
[ + [ + [ + [ + [ + [ + [ + [ + [ + [ + [ + [ + [ + [ + [ + [ + [ + [ + [ + [ + [ a ] [ b ]
] [ c ] ] [ d ] ] [ e ] ] [ f ] ] [ g ] ] [ h ] ] [ i ] ] [ j ] ] [ k ] ] [ l ] ] [ m ] ] [
n ] ] [ o ] ] [ p ] ] [ q ] ] [ r ] ] [ s ] ] [ t ] ] [ u ] ] [ v ] ] [ w ] ] [ x ] ] [ y ]
] [ z ] ] [ * [ + [ + [ + [ + [ + [ + [ + [ + [ + [ + [ + [ + [ + [ + [ + [ + [ + [ + [ + [
+ [ + [ + [ + [ + [ + [ + [ + [ + [ + [ + [ + [ + [ + [ + [ + [ 1 ] [ 2 ] ] [ 3 ] ] [ 4 ] ]
[ 5 ] ] [ 6 ] ] [ 7 ] ] [ 8 ] ] [ 9 ] ] [ 0 ] ] [ a ] ] [ b ] ] [ c ] ] [ d ] ] [ e ] ] [ f
] ] [ g ] ] [ h ] ] [ i ] ] [ j ] ] [ k ] ] [ l ] ] [ m ] ] [ n ] ] [ o ] ] [ p ] ] [ q ] ]
[ r ] ] [ s ] ] [ t ] ] [ u ] ] [ v ] ] [ w ] ] [ x ] ] [ y ] ] [ z ] ] ] ] [ # ] ] ]
```

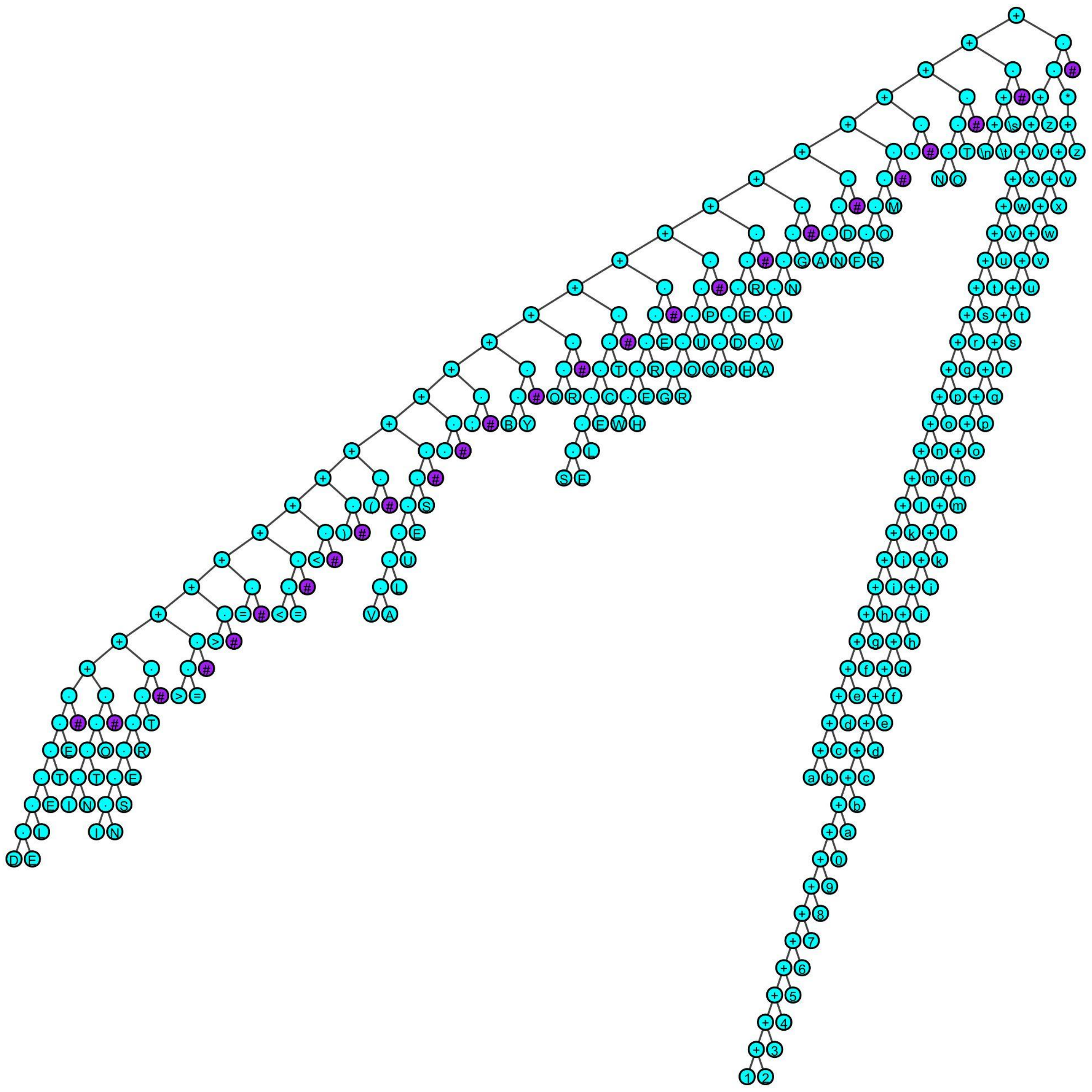

Fig: Visual Representation of Generated Master Syntax Tree

- **Master Syntax Tree Annotation (nullable, firstpos, lastpos, followpos):**

```
#0 -->
Nullable: False,
First Pos: 0,
Last Pos: 0,
Follow Pos: 1,


#1 -->
Nullable: False,
First Pos: 1,
Last Pos: 1,
Follow Pos: 2,
#2 -->
Nullable: False,
First Pos: 2,
Last Pos: 2,
Follow Pos: 3,


#3 -->
Nullable: False,
First Pos: 3,
Last Pos: 3,
Follow Pos: 4,

...
...

#103 -->
Nullable: False,
First Pos: 103,
Last Pos: 103,
Follow Pos: 154, 162, 137, 135, 155, 136, 151, 153, 132, 147, 164, 165, 156, 133, 131, 152,
150, 160, 158, 134, 130, 146, 143, 144, 159, 140, 163, 149, 145, 138, 139, 129, 141, 142,
148, 161, 157,


#104 -->
Nullable: False,
First Pos: 104,
Last Pos: 104,
Follow Pos: 154, 162, 137, 135, 155, 136, 151, 153, 132, 147, 164, 165, 156, 133, 131, 152,
150, 160, 158, 134, 130, 146, 143, 144, 159, 140, 163, 149, 145, 138, 139, 129, 141, 142,
148, 161, 157,


#105 -->
Nullable: False,
First Pos: 105,
Last Pos: 105,
Follow Pos: 154, 162, 137, 135, 155, 136, 151, 153, 132, 147, 164, 165, 156, 133, 131, 152,
150, 160, 158, 134, 130, 146, 143, 144, 159, 140, 163, 149, 145, 138, 139, 129, 141, 142,
148, 161, 157,

...
...

#164 -->
```

```
Nullable: False,
First Pos: 164,
Last Pos: 164,
Follow Pos: 154, 162, 137, 135, 155, 136, 151, 153, 132, 147, 164, 165, 156, 133, 131, 152,
150, 160, 158, 134, 130, 146, 143, 144, 159, 140, 163, 149, 145, 138, 139, 129, 141, 142,
148, 161, 157,


#165 -->
Nullable: False,
First Pos: 165,
Last Pos: 165,
Follow Pos:
```

- **DFA Definition (with Alias):**

```
DFA DEFINATIONS -->
Symbols from RE: 0 , 9 , 8 , 7 , 6 , 5 , 4 , 3 , 2 , 1 , z , y , v , u , t , s , r , q , p
, n , m , l , k , j , i , h , g , f , e , w , ( , M , x , ) , N , < , a , A , R , S , . ,
, > , c , O , V , E , o ,   , L , D , T , I , = , b , U , ; , B ,
 , Y , C , W , H , G , P , F , , , d ,
DFA -->
-> #0 -->  (z , 1)  |  (y , 1)  |  (v , 1)  |  (u , 1)  |  (t , 1)  |  (s , 1)  |  (r , 1)
|  (q , 1)  |  (p , 1)  |  (n , 1)  |  (m , 1)  |  (l , 1)  |  (k , 1)  |  (j , 1)  |  (i ,
1)  |  (h , 1)  |  (g , 1)  |  (f , 1)  |  (e , 1)  |  (w , 1)  |  (( , 2)  |  (x , 1)  |
() , 3)  |  (N , 4)  |  (< , 5)  |  (a , 1)  |  (A , 6)  |  (S , 7)  |  (. , 8)  |  (\t ,
9)  |  (> , 10)  |  (c , 1)  |  (O , 11)  |  (V , 12)  |  (o , 1)  |  (  , 9)  |  (D , 13)
|  (I , 14)  |  (= , 15)  |  (b , 1)  |  (; , 16)  |  (B , 17)  |  (\n , 9)  |  (W , 18)  |
(H , 19)  |  (G , 20)  |  (F , 21)  |  (, , 22)  |  (d , 1)  |   &SubCnt: 49
** (ID) #1 -->  (0 , 1)  |  (9 , 1)  |  (8 , 1)  |  (7 , 1)  |  (6 , 1)  |  (5 , 1)  |  (4
, 1)  |  (3 , 1)  |  (2 , 1)  |  (1 , 1)  |  (z , 1)  |  (y , 1)  |  (v , 1)  |  (u , 1)  |
(t , 1)  |  (s , 1)  |  (r , 1)  |  (q , 1)  |  (p , 1)  |  (n , 1)  |  (m , 1)  |  (l , 1)
|  (k , 1)  |  (j , 1)  |  (i , 1)  |  (h , 1)  |  (g , 1)  |  (f , 1)  |  (e , 1)  |  (w ,
1)  |  (x , 1)  |  (a , 1)  |  (c , 1)  |  (o , 1)  |  (b , 1)  |  (d , 1)  |   &SubCnt: 36
** (() #2 -->   &SubCnt: 0
** ()) #3 -->   &SubCnt: 0
   #4 -->  (O , 23)  |   &SubCnt: 1
** (<) #5 -->  (= , 24)  |   &SubCnt: 1
   #6 -->  (N , 25)  |   &SubCnt: 1
   #7 -->  (E , 26)  |   &SubCnt: 1
** (.) #8 -->   &SubCnt: 0
** (SEP) #9 -->   &SubCnt: 0
** (>) #10 -->  (= , 27)  |   &SubCnt: 1
   #11 -->  (R , 28)  |   &SubCnt: 1
   #12 -->  (A , 29)  |   &SubCnt: 1
   #13 -->  (E , 30)  |   &SubCnt: 1
   #14 -->  (N , 31)  |   &SubCnt: 1
```

```
** (=) #15 -->   &SubCnt: 0
** (;) #16 -->   &SubCnt: 0
   #17 -->  (Y , 32)  |   &SubCnt: 1
   #18 -->  (H , 33)  |   &SubCnt: 1
   #19 -->  (A , 34)  |   &SubCnt: 1
   #20 -->  (R , 35)  |   &SubCnt: 1
   #21 -->  (R , 36)  |   &SubCnt: 1
** (,) #22 -->   &SubCnt: 0
   #23 -->  (T , 37)  |   &SubCnt: 1
** (<=) #24 -->   &SubCnt: 0
   #25 -->  (D , 38)  |   &SubCnt: 1
   #26 -->  (L , 39)  |   &SubCnt: 1
** (>=) #27 -->   &SubCnt: 0
** (OR) #28 -->  (D , 40)  |   &SubCnt: 1
   #29 -->  (L , 41)  |   &SubCnt: 1
   #30 -->  (L , 42)  |   &SubCnt: 1
   #31 -->  (S , 43)  |  (T , 44)  |   &SubCnt: 2
** (BY) #32 -->   &SubCnt: 0
   #33 -->  (E , 45)  |   &SubCnt: 1
   #34 -->  (V , 46)  |   &SubCnt: 1
   #35 -->  (O , 47)  |   &SubCnt: 1
   #36 -->  (O , 48)  |   &SubCnt: 1
** (NOT) #37 -->   &SubCnt: 0
** (AND) #38 -->   &SubCnt: 0
   #39 -->  (E , 49)  |   &SubCnt: 1
   #40 -->  (E , 50)  |   &SubCnt: 1
   #41 -->  (U , 51)  |   &SubCnt: 1
   #42 -->  (E , 52)  |   &SubCnt: 1
   #43 -->  (E , 53)  |   &SubCnt: 1
   #44 -->  (O , 54)  |   &SubCnt: 1
   #45 -->  (R , 55)  |   &SubCnt: 1
   #46 -->  (I , 56)  |   &SubCnt: 1
   #47 -->  (U , 57)  |   &SubCnt: 1
   #48 -->  (M , 58)  |   &SubCnt: 1
   #49 -->  (C , 59)  |   &SubCnt: 1
   #50 -->  (R , 60)  |   &SubCnt: 1
   #51 -->  (E , 61)  |   &SubCnt: 1
   #52 -->  (T , 62)  |   &SubCnt: 1
   #53 -->  (R , 63)  |   &SubCnt: 1
** (INTO) #54 -->   &SubCnt: 0
   #55 -->  (E , 64)  |   &SubCnt: 1
   #56 -->  (N , 65)  |   &SubCnt: 1
   #57 -->  (P , 66)  |   &SubCnt: 1
** (FROM) #58 -->   &SubCnt: 0
   #59 -->  (T , 67)  |   &SubCnt: 1
** (ORDER) #60 -->   &SubCnt: 0
   #61 -->  (S , 68)  |   &SubCnt: 1
   #62 -->  (E , 69)  |   &SubCnt: 1
   #63 -->  (T , 70)  |   &SubCnt: 1
** (WHERE) #64 -->   &SubCnt: 0
   #65 -->  (G , 71)  |   &SubCnt: 1
** (GROUP) #66 -->   &SubCnt: 0
** (SELECT) #67 -->   &SubCnt: 0
** (VALUES) #68 -->   &SubCnt: 0
** (DELETE) #69 -->   &SubCnt: 0
** (INSERT) #70 -->   &SubCnt: 0
** (HAVING) #71 -->   &SubCnt: 0

Size:  72

Alias -->
#0 -> 126 ,108 ,107 ,113 ,123 ,112 ,104 ,110 ,121 ,103 ,116 ,106 ,88 ,93 ,35 ,22 ,125 ,114
,29 ,33 ,105 ,99 ,95 ,44 ,26 ,100 ,101 ,52 ,119 ,12 ,0 ,122 ,120 ,71 ,7 ,65 ,42 ,84 ,59 ,24
,124 ,109 ,31 ,127 ,46 ,115 ,49 ,77 ,117 ,111 ,19 ,128 ,118 ,
#1 -> 157 ,165 ,133 ,156 ,155 ,153 ,132 ,147 ,151 ,135 ,137 ,154 ,162 ,152 ,131 ,136 ,164
,150 ,160 ,158 ,134 ,130 ,146 ,143 ,144 ,159 ,140 ,163 ,149 ,145 ,138 ,139 ,129 ,141 ,142
,148 ,161 ,
#2 -> 34 ,
#3 -> 32 ,
#4 -> 96 ,
#5 -> 27 ,30 ,

...
...

#69 -> 6 ,
#70 -> 18 ,
#71 -> 83 ,
```

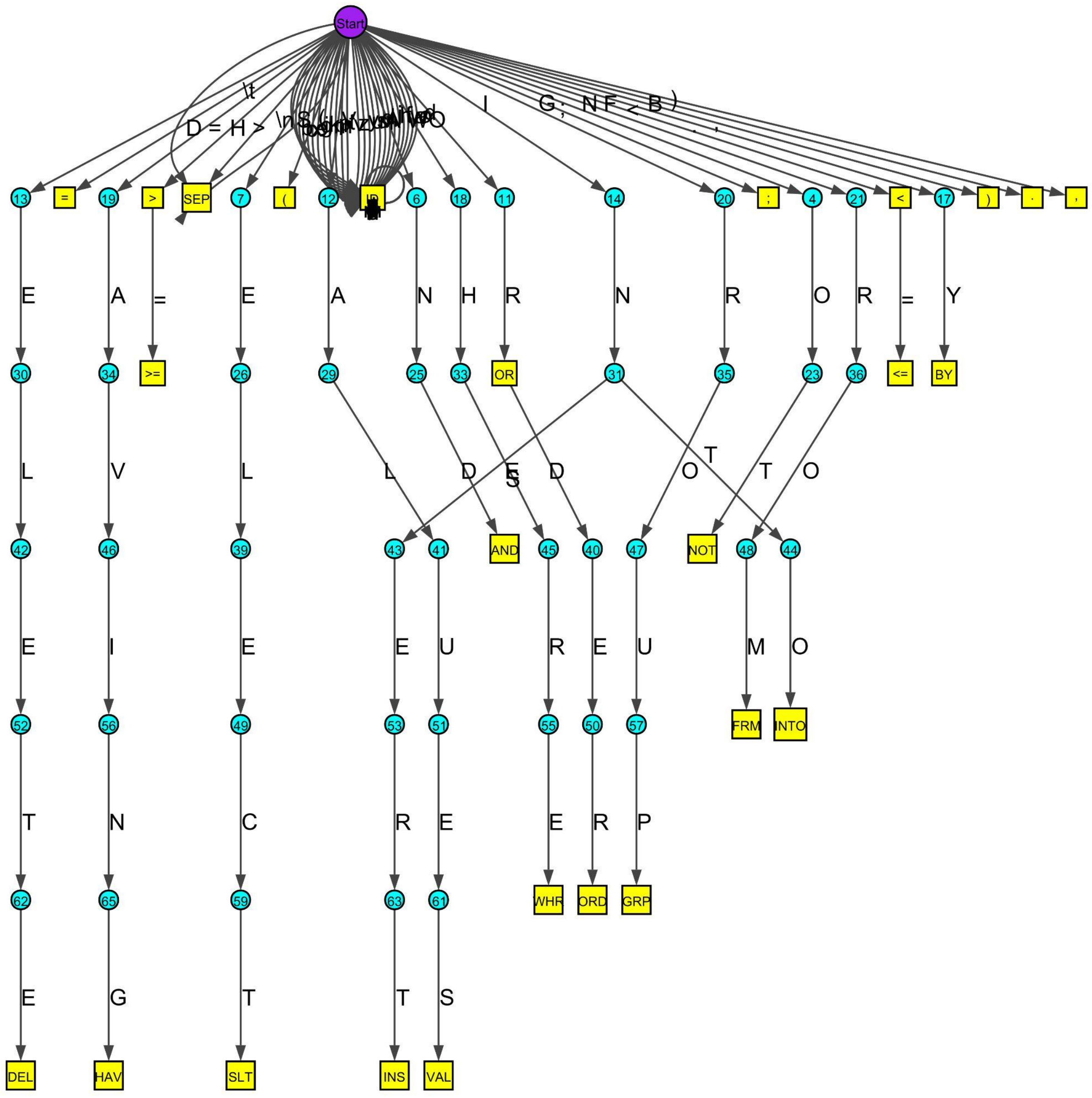

Fig: Visual Representation of Generated DFA

- **Tokenized Stream**

```
Size = 459

INSERT INSERT
INTO INTO
ID mytable
( (
ID atr1
, ,
ID atr2
, ,
ID atr3
, ,
ID atr4
) )
VALUES VALUES
( (
ID bren
, ,
ID pro
, ,
ID op
, ,
ID super
) )
; ;
INSERT INSERT
INTO INTO
ID mytable2

...
...

ID atr
) )
; ;
INSERT INSERT
INTO INTO
ID dual
VALUES VALUES
( (
ID atr
) )
; ;
INSERT INSERT
INTO INTO
ID dual
VALUES VALUES
( (
ID atr
) )
; ;
INSERT INSERT
INTO INTO
ID dual
VALUES VALUES
( (
ID atr
) )
; ;
```

- **Parsed Production**

```
READ PRODUCTIONS [ Start Symbol = S(s) ] -->
S(delst) -> T(DELETE) T(FROM) T(ID) S(wherecl)
S(insst) -> T(INSERT) T(INTO) T(ID) S(insst1)
S(unitcd) -> T(NOT) T(() S(unitcd) T()) | S(ide) S(compop) S(ide)
S(idlist) -> T(ID) S(idlnul)
S(insst') -> T(,) T(ID) S(insst') T(ID) T(,) | T()) T(VALUES) T(()
S(ide') -> T(.) T(ID) | T(E)
S(s) -> S(stmt) T(;) S(s) | T(E)
S(selst) -> T(SELECT) S(idlste) T(FROM) S(idlist) S(wherecl) S(grpbycl) S(ordercl)
S(insst1) -> T(VALUES) T(() S(idlist) T()) | T(() T(ID) S(insst') T(ID) T())
S(cdjoin) -> T(AND) | T(OR)
S(ide) -> T(ID) S(ide')
S(compop) -> T(=) | T(>) | T(<) | T(>=) | T(<=)
S(stmt) -> S(selst) | S(insst) | S(delst)
S(idlnul) -> T(,) T(ID) S(idlnul) | T(E)
S(cond) -> S(unitcd) S(cond')
S(idlnle) -> T(,) S(ide) S(idlnle) | T(E)
S(grpbycl) -> T(GROUP) T(BY) S(idlste) S(havngcl) | T(E)
S(wherecl) -> T(WHERE) S(cond) | T(E)
S(havngcl) -> T(HAVING) S(cond) | T(E)
S(idlste) -> S(ide) S(idlnle)
S(ordercl) -> T(ORDER) T(BY) S(idlste) | T(E)
S(cond') -> S(cdjoin) S(unitcd) S(cond') | T(E)

SYMBOLS:  S(delst) , S(insst) , S(unitcd) , S(idlist) , S(insst') , S(ide') , S(s) ,
S(selst) , S(insst1) , S(cdjoin) , S(ide) , S(compop) , S(stmt) , S(idlnul) , S(cond) ,
S(idlnle) , S(grpbycl) , S(wherecl) , S(havngcl) , S(idlste) , S(ordercl) , S(cond')
TERMINALS:  T($) , T(DELETE) , T(INTO) , T(INSERT) , T(>=) , T(<) , T(>) , T(=) , T()) ,
T(VALUES) , T(E) , T(.) , T(;) , T(<=) , T(BY) , T(OR) , T(SELECT) , T(WHERE) , T(GROUP) ,
T(ORDER) , T(HAVING) , T(AND) , T(FROM) , T(,) , T(NOT) , T(ID) , T(()
```

- **Left Recursion Removal**

This function is available in this project, however the SQL grammar created by us did not contain any left recursion, and hence, the output of this log was same as that of the previous subsection i.e. Parsed Production

- **Nullable, first and follow of Non terminals**

```
NULLABLE MAP -->
S(idlist) -> False
S(cdjoin) -> False
S(insst1) -> False
S(wherecl) -> True
```

```
S(insst) -> False
S(grpbycl) -> True
S(cond) -> False
S(idlnle) -> True
S(idlnul) -> True
S(insst') -> False
S(ide') -> True
S(s) -> True
S(stmt) -> False
S(compop) -> False
S(havngcl) -> True
S(idlste) -> False
S(ordercl) -> True
S(cond') -> True
S(ide) -> False
S(selst) -> False
S(delst) -> False
S(unitcd) -> False


FIRST MAP -->
S(s) -> { T(INSERT) , T(DELETE) , T(SELECT) }
S(cond') -> { T(AND) , T(OR) }
S(idlste) -> { T(ID) }
S(ordercl) -> { T(ORDER) }
S(idlist) -> { T(ID) }
S(ide') -> { T(.) }
S(insst') -> { T()) , T(,) }
S(delst) -> { T(DELETE) }
S(unitcd) -> { T(ID) , T(NOT) }
S(insst) -> { T(INSERT) }
S(wherecl) -> { T(WHERE) }
S(selst) -> { T(SELECT) }
S(insst1) -> { T(() , T(VALUES) }
S(cdjoin) -> { T(OR) , T(AND) }
S(ide) -> { T(ID) }
S(stmt) -> { T(DELETE) , T(SELECT) , T(INSERT) }
S(compop) -> { T(>=) , T(<) , T(=) , T(<=) , T(>) }
S(idlnul) -> { T(,) }
S(cond) -> { T(NOT) , T(ID) }
S(idlnle) -> { T(,) }
S(grpbycl) -> { T(GROUP) }
S(havngcl) -> { T(HAVING) }


FOLLOW MAP -->
S(cond') -> { T(;) , T(ORDER) , T(GROUP) }
S(havngcl) -> { T(;) , T(ORDER) }
```

```
S(idlnle) -> { T(;) , T(HAVING) , T(ORDER) , T(FROM) }
S(cond) -> { T(;) , T(GROUP) , T(ORDER) }
S(delst) -> { T(;) }
S(stmt) -> { T(;) }
S(compop) -> { T(ID) }
S(s) -> { T($) }
S(ide) -> { T(GROUP) , T(HAVING) , T(FROM) , T(ORDER) , T(;) , T(OR) , T(AND) , T(<=) ,
T(>) , T(=) , T()) , T(>=) , T(<) , T(,) }
S(unitcd) -> { T(GROUP) , T(ORDER) , T(AND) , T()) , T(;) , T(OR) }
S(ordercl) -> { T(;) }
S(idlste) -> { T(;) , T(ORDER) , T(FROM) , T(HAVING) }
S(idlnul) -> { T(;) , T(WHERE) , T(ORDER) , T()) , T(GROUP) }
S(idlist) -> { T(;) , T()) , T(GROUP) , T(ORDER) , T(WHERE) }
S(wherecl) -> { T(;) , T(ORDER) , T(GROUP) }
S(insst) -> { T(;) }
S(grpbycl) -> { T(;) , T(ORDER) }
S(insst1) -> { T(;) }
S(cdjoin) -> { T(ID) , T(NOT) }
S(selst) -> { T(;) }
S(insst') -> { T(ID) }
S(ide') -> { T(GROUP) , T(ORDER) , T(FROM) , T(HAVING) , T(,) , T(<) , T(AND) , T(;) ,
T(OR) , T(<=) , T(>) , T(=) , T(>=) , T()) }
```

- **Augmented LL(1) parsing table (with pop and scan)**

	T($)	T(DELETE)	T(INTO)	T(INSERT)	T(>=)	T(<)	T(>)	T(=)	T())	T(VALUES)	T(E)	T(.)	T(;)	T(<=)	T(BY)	T(OR)	T(SELECT)	T(WHERE)	T(GROUP)	T(ORDER)	T(HAVING)	T(AND)	T(FROM)	T(,)	T(NOT)	T(ID)	T(()
S(delst)	POP	41	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	POP	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN
S(insst)	POP	SCAN	SCAN	36	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	POP	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN
S(unitcd)	POP	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	POP	SCAN	SCAN	SCAN	POP	SCAN	SCAN	POP	SCAN	SCAN	POP	POP	SCAN	POP	SCAN	SCAN	29	30	SCAN
S(idlist)	POP	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	POP	SCAN	SCAN	SCAN	POP	SCAN	SCAN	SCAN	SCAN	POP	POP	POP	SCAN	SCAN	SCAN	SCAN	SCAN	10	SCAN
S(insst')	POP	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	40	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	39	SCAN	POP	SCAN
S(ide')	POP	SCAN	SCAN	SCAN	8	8	8	8	8	SCAN	SCAN	7	8	8	SCAN	8	SCAN	SCAN	8	8	8	8	8	8	SCAN	SCAN	SCAN
S(s)	2	1	SCAN	1	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	1	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN
S(selst)	POP	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	POP	SCAN	SCAN	SCAN	9	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN
S(insst1)	POP	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	37	SCAN	SCAN	POP	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	38
S(cdjoin)	POP	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	28	SCAN	SCAN	SCAN	SCAN	SCAN	27	SCAN	SCAN	POP	POP	SCAN
S(ide)	POP	SCAN	SCAN	SCAN	POP	POP	POP	POP	POP	SCAN	SCAN	SCAN	POP	POP	SCAN	POP	SCAN	SCAN	POP	POP	POP	POP	POP	POP	SCAN	6	SCAN
S(compop)	POP	SCAN	SCAN	SCAN	34	33	32	31	SCAN	SCAN	SCAN	SCAN	SCAN	35	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	POP	SCAN
S(stmt)	POP	5	SCAN	4	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	POP	SCAN	SCAN	SCAN	3	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN
S(idlnul)	POP	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	12	SCAN	SCAN	SCAN	12	SCAN	SCAN	SCAN	SCAN	12	12	12	SCAN	SCAN	SCAN	11	SCAN	SCAN	SCAN
S(cond)	POP	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	POP	SCAN	SCAN	SCAN	SCAN	SCAN	POP	POP	SCAN	SCAN	SCAN	SCAN	24	24	SCAN
S(idlnle)	POP	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	15	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	15	15	SCAN	15	14	SCAN	SCAN	SCAN
S(grpbycl)	POP	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	19	SCAN	SCAN	SCAN	SCAN	SCAN	18	19	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN
S(wherecl)	POP	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	17	SCAN	SCAN	SCAN	SCAN	16	17	17	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN
S(havngcl)	POP	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	21	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	21	20	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN
S(idlste)	POP	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	POP	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	POP	POP	SCAN	POP	SCAN	SCAN	13	SCAN
S(ordercl)	POP	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	23	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	22	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN
S(cond')	POP	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	SCAN	26	SCAN	SCAN	25	SCAN	SCAN	26	26	SCAN	25	SCAN	SCAN	SCAN	SCAN	SCAN

F. Class Description and Class Diagram

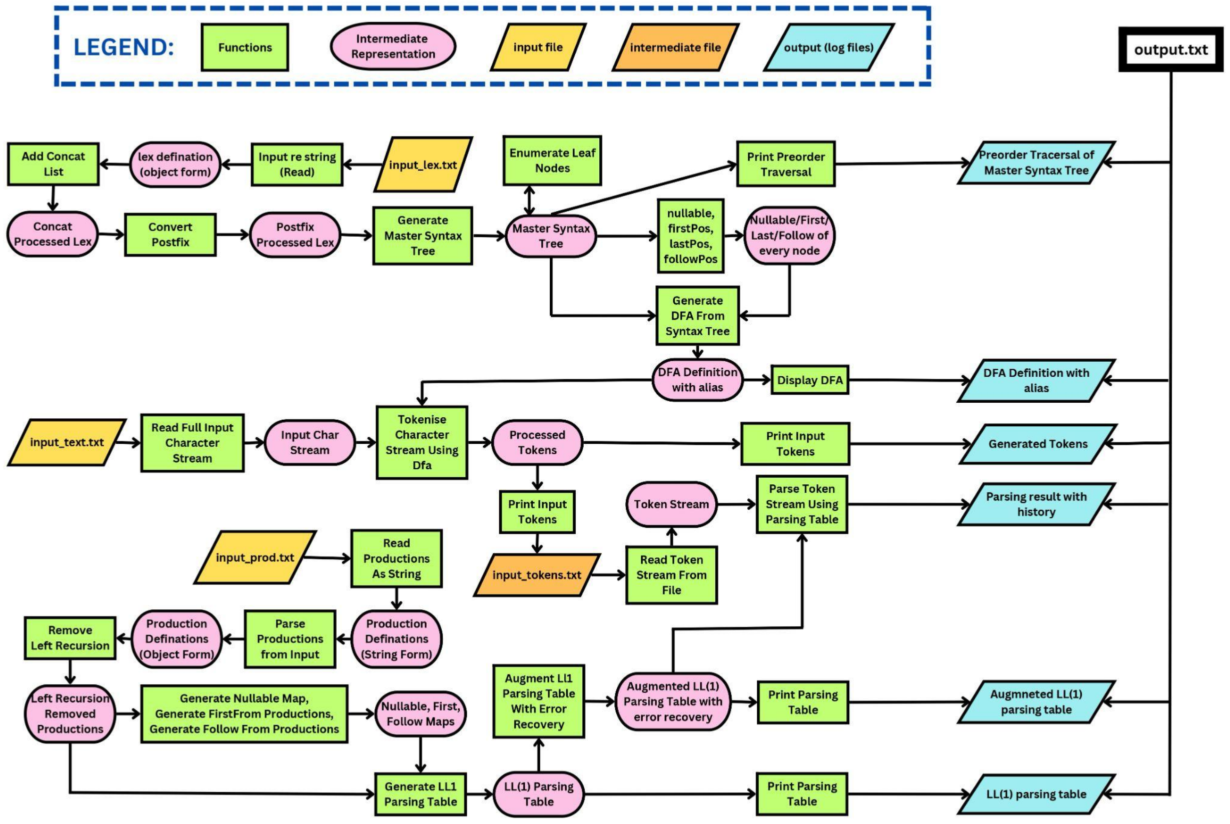
LEGEND:
Functions
Intermediate Representation
input file
intermediate file
output (log files)
output.txt
Add Concat List
lex defination (object form)
Input re string (Read)
input_lex.txt
Enumerate Leaf Nodes
Print Preorder Traversal
Preorder Tracersal of Master Syntax Tree
Concat Processed Lex
Convert Postfix
Postfix Processed Lex
Generate Master Syntax Tree
Master Syntax Tree
nullable, firstPos, lastPos, followPos
Nullable/First/Last/Follow of every node
Generate DFA From Syntax Tree
DFA Definition with alias
Display DFA
DFA Definition with alias
input_text.txt
Read Full Input Character Stream
Input Char Stream
Tokenise Character Stream Using Dfa
Processed Tokens
Print Input Tokens
Generated Tokens
Print Input Tokens
Token Stream
Parse Token Stream Using Parsing Table
Parsing result with history
input_prod.txt
Read Productions As String
input_tokens.txt
Read Token Stream From File
Remove Left Recursion
Production Definations (Object Form)
Parse Productions from Input
Production Definations (String Form)
Augment LL1 Parsing Table With Error Recovery
Augmented LL(1) Parsing Table with error recovery
Print Parsing Table
Augmneted LL(1) parsing table
Left Recursion Removed Productions
Generate Nullable Map, Generate FirstFrom Productions, Generate Follow From Productions
Nullable, First, Follow Maps
Generate LL1 Parsing Table
LL(1) Parsing Table
Print Parsing Table
LL(1) parsing table

Fig: Visual Representation of the architecture and functional interfaces used in the SQL DML parsing system

The following classes were used as intermediate representations in the Parser →

1. **Class Name:** Node
   Used In: Tokenizer (to implement to master syntax tree nodes)
   Class Structure (Member functions and Variables):

```
struct Node
{
    char data;
    Node *left, *right;

    Node(char data = '\0', Node *left = nullptr, Node *right = nullptr){ this->data
= data; this->left = left; this->right = right; }
};
```

Component Description:

- **Data [`char`]:** The operator or character value of the current node
- **Left [`Node *`]:** Pointer to left child of the current node
- **Right [`Node *`]:** Pointer to left child of the current node

2. **Class Name:** Character
   Used In: Tokenizer (to differentiate Character & Operator in regular expression)
   Class Structure (Member functions and Variables):

```
typedef pair<char, bool> Character;  // <value of char , isOperator>
```

Component Description:

- **First `[char]`:** Which character or operator is used
- **Second `[bool]`:** It is operator if true else Character

3. **Class Name:** Nullable, FirstPos, LastPos, FollowPos
   Used In: Tokenizer (to store the evaluated annotations in memory)
   Class Structure (Member functions and Variables):

```
unordered_map<Node *, bool> dpNullable;
unordered_map<Node *, unordered_set<Node *>> dpFirstPos, dpLastPos, dpFollowPos;
```

Component Description:

- **dpNullable:** Mapping of each Non terminal to a boolean representing if the non terminal is nullable or not
- **dpFirstPos, dpLastPos, dpFollowPos:** Mapping of each Non terminal to a set of non terminals (represented in the Node form) which are present in the corresponding First-pos, Last-pos and Follow-pos set respectively

4. **Class Name:** DFA
   Used In: Tokenizer (to store the DFA transition function)
   Class Structure (Member functions and Variables):

```
typedef vector<vector<pair<char, int>>> Dfa;
```

Component Description:

- **`pair<char, int>`:** Mapping to which state it must transition for a given character
- **`vector<pair<char, int>>`:** For this current state, set of all possible transitions for possible characters
- **`vector<vector<pair<char, int>>>`:** For every state , what are its possible transitions

5. **Class Name:** Item
   Used In: Parser (to represent a single symbol or terminal in a production)
   Class Structure (Member functions and Variables):

```
class Item
{
public:
    string val;
    bool isSymbol;

    Item(){ this->val = ""; this->isSymbol = false; }
    Item(string val, bool isSymbol){ this->val = val; this->isSymbol = isSymbol; }
    bool operator == (const Item &i) const{ return this->val == i.val &&
this->isSymbol == i.isSymbol; }
    bool operator != (const Item &i) const{ return !(this->val == i.val &&
this->isSymbol == i.isSymbol); }
    bool isEpsilon(){ return !isSymbol && val.size() == 1 && val[0] == EPSILON; }
    bool isInvalid(){ return val.empty(); }
    string toString(){ if(isInvalid()) return "I(INVALID)"; string ans; if(isSymbol)
ans = "S("; else ans =  "T("; ans += val + ")"; return ans; }

    friend ostream & operator << (ostream &os, Item &i);
};

ostream & operator << (ostream &os, Item &i)
{
    if(i.isInvalid()){ os << "I(INVALID)"; return os; }

    if(i.isSymbol) os << "S(";
    else os << "T(";
    os << i.val << ")";
    return os;
}
```

   Component Description:
   - **Val `[string]`:** Name of the symbol or terminal
   - **isSymbol `[bool]`:** represents if the val is a symbol (returns true) or a terminal (false)
   - Functions:
     - **isEpsilon**: if the item is an epsilon symbol
     - **isInvalid**: if the item is empty (has no characters in it)
     - **Operator overloading** (==, !=): for checking equality of two items

Note: *Item is not to be confused with the production Item, here we used Item to represent a particular symbol or terminal in a production …*

6. **Class Name:** MapFirst, MapFollow, MapNullable
   Used In: Parser (to store the nullable, first and follow mappings in memory)
   Class Structure (Member functions and Variables):

```
struct ItemHasher
{
public:
    size_t operator () (const Item &i) const
    {
        return hash<string>()(i.val) ^ hash<bool>()(i.isSymbol);
    }
};

typedef unordered_map<Item, vector<vector<Item>>, ItemHasher> Map;
typedef unordered_set<Item, ItemHasher> Set;
typedef unordered_map<Item, Set, ItemHasher> MapFirst;
typedef unordered_map<Item, Set, ItemHasher> MapFollow;
typedef unordered_map<Item, bool, ItemHasher> MapNullable;
```

   Component Description:
   - **First [`Item`]:** Mapping for which symbol
   - **Second [`bool`]:** If true, then nullable (first set contains `EPSILON`)
   - **Second [`Set`]:** Set of Terminals that are in First or Follow Set

7. **Class Name:** Action
   Used In: Parser (to represent a `PRODUCE`, `POP`, `SCAN` or `MATCH_TOKEN` action in the Parsing Table)
   Class Structure (Member functions and Variables):

```
class Action
{
public:
    const static int PRODUCE = 1, POP = 2, SCAN = 3, MATCH_TOKEN = 4;

    static Map *prodMap;
    int type;
    Item prodLhs;
    int prodRhsIndex;

    Action(){ type = 0; }
    Action(int type){ this->type = type; }
    Action(int type, Item prodLhs, int prodRhsIndex){ this->type = type;
this->prodLhs = prodLhs; this->prodRhsIndex = prodRhsIndex; }
    Action(Item prodLhs, int prodRhsIndex){ this->type = PRODUCE; this->prodLhs =
prodLhs; this->prodRhsIndex = prodRhsIndex; }
    static void setProdMap(Map *prodMapActual){ Action::prodMap = prodMapActual; }
    bool operator == (const Action &a) const{ return ((this->type == POP ||
this->type == SCAN || this->type == MATCH_TOKEN) && this->type == a.type) ||
(this->type == a.type && this->prodLhs == a.prodLhs && this->prodRhsIndex ==
a.prodRhsIndex); }
    bool operator != (const Action &a) const{ return !(*(this) == a); }
    bool isInvalid(){ return type < 1 || type > 4; }

    string toString()
    {
        if(isInvalid()) return "A(INVALID)";

        string ans("A(");
        if(type == Action::POP) ans += "POP";
        else if(type == Action::SCAN) ans += "SCAN";
        else if(type == Action::MATCH_TOKEN) ans += "MATCH_TOKEN";
        else {
            ans += "PRODUCE[";
            if(!Action::prodMap) ans += prodLhs.toString() + "," +
to_string(prodRhsIndex);
            else{ ans += " " + prodLhs.toString() + " -> "; for(Item i:
(*Action::prodMap)[prodLhs][prodRhsIndex]) ans += i.toString() + " "; }
            ans += "]";
        }
        ans += ")";
        return ans;
    }

    friend ostream & operator << (ostream &os, Action &a);
};

Map *Action::prodMap = nullptr;

ostream & operator << (ostream &os, Action &a)
{
    if(a.isInvalid()){ os << "A(INVALID)"; return os; }

    os << "A(";
    if(a.type == Action::POP) os << "POP";
    else if(a.type == Action::SCAN) os << "SCAN";
    else if(a.type == Action::MATCH_TOKEN) os << "MATCH_TOKEN";
    else {
        os << "PRODUCE[";
        if(!Action::prodMap) os << a.prodLhs << "," << a.prodRhsIndex;
        else{ os << " " << a.prodLhs << " -> "; for(Item i:
(*Action::prodMap)[a.prodLhs][a.prodRhsIndex]) os << i << " "; }
        os << "]";
    }
    os << ")";
    return os;
}
```

Component Description:

- **prodMap (static) [`Map *`]:** A reference to the Parsing Table, so that we can identify which production to produce from during a `PRODUCE` action
- **Type [`int`]:** Type of production – `PRODUCE`, `POP`, `SCAN` or `MATCH_TOKEN`
- **prodLhs [`Item`]:** The symbol from which production is done
- **prodRhsIndex [`int`]:** Which production to do from that symbol [`Multiple productions are indexed symbol-wise`]

8. **Class Name:** Parsing Table

Used In: In Parser (to store the Parsing Table in memory)

Class Structure (Member functions and Variables):

```
typedef unordered_map<Item, unordered_map<Item, Action, ItemHasher>, ItemHasher>
ParsingTable;
```

Component Description:

- **First [`Item`]:** The symbol for which action is to be taken
- **Second [`unordered_map<Item, Action, ItemHasher>`]:**
  - **`Item`:** The terminal for which action is to be taken
  - **`Action`:** The action that is to be taken

9. **Class Name:** Token

Used In: Both Tokenizer and Parsing (to represent the generated tokens from the input SQL File)

Class Structure (Member functions and Variables):

```
class Token
{
public:
    Item tokenType;
    string tokenVal;

    Token(){ tokenType = Item(); }
    Token(Item tokenType, string tokenVal){ this->tokenType = tokenType;
this->tokenVal = tokenVal; }

    friend ostream & operator << (ostream &os, Token &t);
};

ostream & operator << (ostream &os, Token &t)
{
    os << "{" << t.tokenType << ": " << t.tokenVal << "}";
    return os;
}
```

Component Description:

- **tokenType [`Item`]:** The token class it belongs to (represented by Item class as a terminal)
- **tokenVal [`string`]:** The value of the lexeme of this current token

10. **Class Name:** Parsing History

Used In: Parser (to store the history of parsing stack, remaining input stream and next action while using augmented LL(1) Parsing Table)

Class Structure (Member functions and Variables):

```
class ParsingHistoryEntry
{
public:
    vector<Item> curParsingStack;
    vector<Token> curTokenStream;
    Action a;

    ParsingHistoryEntry(vector<Item> &curParsingStack, vector<Token>
&curTokenStream, Action a){ this->curParsingStack = curParsingStack;
this->curTokenStream = curTokenStream; this->a = a; }

    friend ostream & operator << (ostream &os, ParsingHistoryEntry &e);
};

ostream & operator << (ostream &os, ParsingHistoryEntry &e)
{
    os << "Parsing Stack: "; bool lol = false; for(int i = e.curParsingStack.size()
- 1; i >= 0; i--){ if(lol) os << ", "; lol = true; os << e.curParsingStack[i]; } os
<< endl;
    os << "Token Stream: "; lol = false; for(Token t: e.curTokenStream){ if(lol) os
<< ", "; lol = true; os << t; } os << endl;
    os << "Action: " << e.a;
    return os;
}

typedef vector<ParsingHistoryEntry> ParsingHistory;
```

Component Description:

- **curParsingStack [`vector<Item>`]:** Contents of the Parsing Stack (having both symbols and terminals)
- **curTokenStream [`vector<Token>`]:** Remaining tokens (terminals) in the input stream
- **a [`Action`]:** Current action to be taken based on the top / front of the above 2 parameters (referring from the Augmented LL(1) Parsing Table)

G. Procedures and Explanation

(I) Creation of DFA ⇒

- **Create Concat processed Regular expression**: First, we have to insert the Concatenation Operators in the correct places to allow the program to properly understand the regex and build the syntax tree correctly. For example, `ab(c+d)efg` should be processed as `a.b(c+d)e.f.g`, and similarly SQL keyword `DELETE` can be processed to `D.E.L.E.T.E`.

```
vector<Character> addConcat(string &re)
{
    vector<Character> ans;
    bool escapeOn = false;
    for(int i = 0; i < re.size(); i++)
    {
        if(escapeOn)
        {
            if(re[i] == 'n') ans.push_back({'\n', false});  // newline
            else if(re[i] == 't') ans.push_back({'\t', false});  // tab
            else ans.push_back({re[i], false});

            if(i != re.size() - 1 && re[i + 1] != '+' && re[i + 1] != '.' && re[i + 1] !=
'*' && re[i + 1] != ')') ans.push_back({'.', true});
            escapeOn = false;
        }
        else
        {
            if(re[i] == '\\') escapeOn = true;
            else
            {
                ans.push_back({re[i], isOperatorAugmented(re[i])});
                if(i != re.size() - 1 && re[i] != '(' && re[i] != '+' && re[i] != '.' &&
                   re[i + 1] != '+' && re[i + 1] != '.' && re[i + 1] != '*' && re[i + 1] !=
')') ans.push_back({'.', true});
            }
        }
    }

    return ans;
}
```

- **Create Postfix processed Regular Expression**: Now, we have converted every regular expression to its postfix form with the operator precedences in the order `‘*’ > ‘.’ > ‘+’`. This is done because creating the syntax tree is easier using postfix evaluation of the regular expression rather than from its infix evaluation.

```
int precedenceOperator(char op){
    if(op == '+') return 1;
    if(op == '.') return 2;
    if(op == '*') return 3;
    else return -1;
}

vector<Character> convertPostfix(vector<Character> &re)
{
    stack<char> st;
    vector<Character> ans;

    for(Character ch: re)
    {
        if(ch.first == '(' && ch.second) st.push(ch.first);
        else if(ch.first == ')' && ch.second){ while(st.top() != '('){
ans.push_back({st.top(), true}); st.pop(); } st.pop(); }
        else if(ch.second){ while(!st.empty() && precedenceOperator(ch.first) <=
precedenceOperator(st.top())){ ans.push_back({st.top(), true}); st.pop(); }
st.push(ch.first); }  // Operator
        else ans.push_back({ch.first, false});
    }
    while(!st.empty()){ ans.push_back({st.top(), true}); st.pop(); }
    return ans;
}
```

- **Generating syntax tree from a regular expression:** Here, we use the postfix processed form of the regular expression to generate the corresponding syntax tree. A stack is used to keep track of the character nodes and when we encounter an operator, we assign the required number of nodes from the top of the stack to the operator node and push the newly generated node back into the stack. A check is kept that only 1 node must remain in the stack at the end of the operation.

```
Node *generateSyntaxTree(vector<Character> &postRe)
{
    stack<Node *> st;
    for(Character ch: postRe)
    {
        if(ch.second)
        {
            if(ch.first == '*')   // Unary Operator
            {
                if(st.size() < 1) return nullptr;  // Error
                Node *n = st.top(); st.pop();
                st.push(new Node(ch.first, n));
            }
            else   // Binary Operator
            {
                if(st.size() < 2) return nullptr;  // Error
                Node *nRight = st.top(); st.pop();
                Node *nLeft = st.top(); st.pop();
                st.push(new Node(ch.first, nLeft, nRight));
            }
        }
        else st.push(new Node(ch.first));
    }

    if(st.size() != 1) return nullptr;  // Error
    else return new Node('.', st.top(), new Node(DELIM));
}
```

- **Generating master syntax tree from all regular expressions:** We have multiple regular expressions for different token classes. So, using a single syntax tree with single ending node (`'#'`) won't suffice. This is why we generate the syntax tree for every regular expression and keep combining them with an 'OR' operator (`'+'`) node. This gives our master syntax tree with a mapping of the token class to their respective final nodes, instead of having a single final node.

  Note: *Conventionally, in case of regular expression, we use `'|'` symbol, but in the project, we used `'+'` as the symbol or 'OR' operation.*

```
pair<Node *, unordered_map<Node *, string>>
generateCombinedSyntaxTree(vector<pair<vector<Character>, string>> &reList)
{
    Node *masterTree = nullptr;
    unordered_map<Node *, string> finalNodeToTokenTypeMap;
    for(pair<vector<Character>, string> &reTokenPair: reList)
    {
        Node *syntaxTree = generateSyntaxTree(reTokenPair.first);
        string tokenType = reTokenPair.second;

        if(masterTree == nullptr) masterTree = syntaxTree;
        else masterTree = new Node('+', masterTree, syntaxTree);

        finalNodeToTokenTypeMap[syntaxTree->right] = tokenType;
    }

    return {masterTree, finalNodeToTokenTypeMap};
}


void printPreorderTraversal(Node *root, ostream &o)
{
    if(!root) return;

    o << " [ " << ((root->data == '\n')? "\\n": (root->data == '\t')? "\\t": string(1,
root->data));
    printPreorderTraversal(root->left, o);
    printPreorderTraversal(root->right, o); o << " ]";
}
```

- **Generating Nullable, First-pos, Last-pos and Follow-pos for each Node:** Now, we will generate the nullable, first-pos, last-pos and follow-pos values for every node (both operator and character nodes) of the syntax tree. The first 3 have been computed by a recurrence relation and hence the already computed results were stored in memory and are to be referenced for further computation (Dynamic Programming) . The follow-pos was computed using a pre-order traversal of the master syntax tree. Nullable can have values - `True` or `False` , while the other 3 have a set of nodes as their values.

```
unordered_map<Node *, bool> dpNullable;
unordered_map<Node *, unordered_set<Node *>> dpFirstPos, dpLastPos, dpFollowPos;

bool nullable(Node *n)
{
    if(!n->left && !n->right) return n->data == EPSILON_DFA;
    else if(dpNullable.find(n) != dpNullable.end()) return dpNullable[n];

    if(n->data == '+') return dpNullable[n] = nullable(n->left) || nullable(n->right);
    else if(n->data == '.') return dpNullable[n] = nullable(n->left) && nullable(n->right);
    else if(n->data == '*') return dpNullable[n] = true;
    else return false;
}
```

```
unordered_set<Node *> unionSet(unordered_set<Node *> s1, unordered_set<Node *> s2)
{
    unordered_set<Node *> ans;
    for(Node *n: s1) ans.insert(n);
    for(Node *n: s2) ans.insert(n);
    return ans;
}


unordered_set<Node *> firstPos(Node *n)
{
    if(!n->left && !n->right){ if(n->data == EPSILON_DFA) return unordered_set<Node *>();
                               else return unordered_set<Node *>({n}); }
    else if(dpFirstPos.find(n) != dpFirstPos.end()) return dpFirstPos[n];

    if(n->data == '+') return dpFirstPos[n] = unionSet(firstPos(n->left),
firstPos(n->right));
    else if(n->data == '.'){ if(nullable(n->left)) return dpFirstPos[n] =
unionSet(firstPos(n->left), firstPos(n->right));
                             else return dpFirstPos[n] = firstPos(n->left); }
    else if(n->data == '*') return dpFirstPos[n] = firstPos(n->left);
    else return unordered_set<Node *>();
}


unordered_set<Node *> lastPos(Node *n)
{
    if(!n->left && !n->right){ if(n->data == EPSILON_DFA) return unordered_set<Node *>();
                               else return unordered_set<Node *>({n}); }
    else if(dpLastPos.find(n) != dpLastPos.end()) return dpLastPos[n];

    if(n->data == '+') return dpLastPos[n] = unionSet(lastPos(n->left), lastPos(n->right));
    else if(n->data == '.'){ if(nullable(n->right)) return dpLastPos[n] =
unionSet(lastPos(n->left), lastPos(n->right));
                             else return dpLastPos[n] = lastPos(n->right); }
    else if(n->data == '*') return dpLastPos[n] = lastPos(n->left);
    else return unordered_set<Node *>();
}


void dfsComputeFollowPos(Node *root)
{
    if(!root) return;

    // Since terminal nodes can have . and * , its better to check for non-Leaf nodes ...
    if(root->data == '.' && root->left && root->right)
    {
        for(Node *l: lastPos(root->left))
            for(Node *f: firstPos(root->right))
                dpFollowPos[l].insert(f);
    }
    else if(root->data == '*' && root->left)
    {
        for(Node *l: lastPos(root->left))
            for(Node *f: firstPos(root->left))
                dpFollowPos[l].insert(f);
    }

    dfsComputeFollowPos(root->left);
    dfsComputeFollowPos(root->right);
}
```

- **Generating DFA from Master Syntax Tree, Nullable, First-Pos, Last-Pos and Follow-Pos:** Now, the main algorithm is used to generate a DFA using the Master Syntax Tree and its computed annotations. Here, are the rules →
  - ❖ The first-pos of the root node is the starting DFA state (#0). Here, a state of DFA is a set of nodes from the master syntax tree.
  - ❖ We check for every character used in the regular expressions, if a new DFA state can be obtained by transitioning from the previous state using this character. If so, then the same process is done for this new state till no new states emerge.
  - ❖ A DFA state with with empty set of nodes means that transition cannot occur (goes to dead state).
  - ❖ The new state consists of union of follow-pos of every tree node in the current DFA state for which the character of the tree node is same as that of the transitioning character.

```
pair<pair<vector<vector<pair<char, int>>>, unordered_map<int, string>>,
vector<unordered_set<Node *>>> generateDFAFromSyntaxTree(Node *root, unordered_set<char>
symbols, unordered_map<Node *, string> &endNodeToTokenTypeMap, bool log = true)
{
    // returns (graph, final_states) first , then returns actual alias of enumeration to
unordered_set of nodes ...
    // 0th state is always starting state ...
    vector<unordered_set<Node *>> dStates;  // used to keep track of marked dStates ...
    unordered_map<unordered_set<Node *>, int, MySetHasher> indexOfState;
    unordered_map<unordered_set<Node *>, vector<pair<char, unordered_set<Node *>>>,
MySetHasher> dTrans;  // Weighted Graph (char, newState [as a set]) ...
    unordered_set<Node *> curState;
    int cur = 0;
    dStates.push_back({firstPos(root)});
    dTrans[{firstPos(root)}];
    indexOfState[{firstPos(root)}] = 0;

    while(cur < dStates.size())
    {
        curState = dStates[cur++];

        for(char symbol: symbols)
        {
            unordered_set<Node *> nextState;

            for(Node *n: curState)
                if(n->data == symbol)
                    for(Node *f: followPos(n))
                        nextState.insert(f);

            if(!nextState.empty())
            {
                // if nextState is empty , means dead state ...
                if(dTrans.find(nextState) == dTrans.end()) dTrans[nextState],
dStates.push_back(nextState), indexOfState[nextState] = dStates.size() - 1;
                dTrans[curState].push_back({symbol, nextState});
            }
        }
    }

    vector<vector<pair<char, int>>> dfa(dStates.size());
    unordered_map<int, string> finalStates;
    vector<string> multipleDefinationLogs;
    for(int i = 0; i < dStates.size(); i++)
    {
        for(Node *n: dStates[i]) if(endNodeToTokenTypeMap.find(n) !=
endNodeToTokenTypeMap.end())
        {
            if(finalStates.find(i) != finalStates.end())
multipleDefinationLogs.push_back("Node #" + to_string(i) + " -> " +
endNodeToTokenTypeMap[n]);
            else finalStates[i] = endNodeToTokenTypeMap[n];
        }

        for(pair<char, unordered_set<Node *>> &edge: dTrans[dStates[i]])
            dfa[i].push_back({edge.first, indexOfState[edge.second]});
    }

    if(log && !multipleDefinationLogs.empty()){ cout << "MULTIPLE DEFINATIONS OF DFA TOKENS
FOUND -->" << endl; for(string st: multipleDefinationLogs) cout << "  " << st << endl; cout
<< endl; }

    return {{dfa, finalStates}, dStates};
}


void displayDFA(vector<vector<pair<char, int>>> &dfa, unordered_map<int, string>
&finalStates, ostream &o)
{
    for(int i = 0; i < dfa.size(); i++)
    {
        if(i == 0) o << "-> "; else if(finalStates.find(i) != finalStates.end()) o << "**
(" << finalStates[i] << ") "; else o << "   ";
        o << "#" << i << " -->  ";
        for(pair<char, int> edge: dfa[i]) o << "(" << ((edge.first == '\n')? "\\n":
(edge.first == '\t')? "\\t": string(1, edge.first)) << " , " << edge.second << ")  |  ";
        o << " &SubCnt:  " << dfa[i].size() << endl;
    }

    o << endl << "Size:  " << dfa.size() << endl;
}
```

- **Tokenizing the character stream using DFA:** Now, we use our generated DFA to generate the tokens based on the following rules →
    - ❖ Till, we can extend the lexeme using the next character in the stream (i.e., transition using the character does not go to dead state and "either final state has not been reached or next state is also a final state") , pop the character and extend the lexeme.
    - ❖ If current lexeme cannot be extended, return the current token type (`INVALID` token if not in a final state)
    - ❖ If different characters are present, which are not defined in the Regular Expressions, every character is an `INVALID` token

```
vector<Token> tokeniseCharacterStreamUsingDfa(string &stream, vector<unordered_map<char,
int>> &dfaMap, unordered_map<int, string> finalStates)
{
    vector<Token> ans;
    string lexeme;
    int ptr = 0, curState = 0;  // Starting state is always 0 ...
    char ch;
    while(ptr < stream.size())
    {
        ch = stream[ptr++];
        if(dfaMap[curState].find(ch) != dfaMap[curState].end()) curState =
dfaMap[curState][ch], lexeme.push_back(ch);
```

```
        else
        {
            if(curState == 0) ans.push_back({makeTerminal("INVALID"), string(1, ch)});
            else ans.push_back({makeTerminal(((finalStates.find(curState) !=
finalStates.end())? finalStates[curState]: "INVALID")), lexeme}), ptr--;

            lexeme.clear();
            curState = 0;
        }
    }

    if(!lexeme.empty()) ans.push_back({makeTerminal(((finalStates.find(curState) !=
finalStates.end())? finalStates[curState]: "INVALID")), lexeme});

    return ans;
}
```

(II) __Creation of LL(1) Parser__ ⇒

- **Reading and validating productions:** A production should have 1 symbol on LHS and multiple symbols or terminals (or epsilon) on the right, separated by an arrow (`‘->’`). Every term in the production should be separated by spaces. A new production should begin in a new line. In first pass, all the terms on the LHS are recorded in a set as the symbols. Then, rest terms are identified as terminals. This would help us later in the program in various phases of the parser.

  __**Note**__: *`‘E’` has been reserved and used for denoting `epsilon`*

```
vector<string> myTokenizer(string &s, char del = ' ')
{
    vector<string> ans;
    stringstream ss(s);
    string word;
    while (!ss.eof()){ getline(ss, word, del); if(!word.empty()) ans.push_back(word);
word.clear(); }

    return ans;
}

pair<Map, pair<Set, Set>> parseProductionsfromInput(vector<string> &productions)
{
    Map prodMap;
    vector<vector<string>> curItemList;
    vector<string> curItems;
    unordered_set<string> symbols;
    Set symbolItems;
    Set terminalItems;

    for(string production: productions)
    {
        curItemList.push_back(myTokenizer(production));
        symbols.insert(curItemList.back()[0]);
        symbolItems.insert(makeSymbol(curItemList.back()[0]));
    }

    int j = 0;
    for(string production: productions)
    {
        curItems = curItemList[j++];
        if(curItems[1] != string("->")){ cout << "Invalid Production: " << production <<
endl; exit(0); }

        vector<Item> prodRhs;
        for(int i = 2; i < curItems.size(); i++) if(symbols.find(curItems[i]) !=
symbols.end()) prodRhs.push_back(makeSymbol(curItems[i])); else
prodRhs.push_back(makeTerminal(curItems[i])),
terminalItems.insert(makeTerminal(curItems[i]));
        prodMap[makeSymbol(curItems[0])].push_back(prodRhs);
    }

    terminalItems.insert(END_TERMINAL);

    return {prodMap, {symbolItems, terminalItems}};
}
```

- **Left Recursion Removal:** Both immediate and non-immediate left recursion have been removed by making sure that only symbols which have already been processed (including itself) can appear in the RHS of a symbol's production, else the symbol is replaced with its productions and immediate left recursion removal is applied. This is how non-immediate left recursion removal works (which also removes immediate left recursion)

```
Map removeImmediateLeftRecursion(Map &prodMap)
{
    // A → Aα1|Aα2| … . |Aαm|β1|β2| … . . βn
    // can be replaced by -->
    // A → β1A'|β2A'| … . . | … . . |βnA'
    // A → α1A'|α2A'| … . . |αmA'|ε
    Map prodMapAns;
    for(auto prod : prodMap)
    {
        Item leftSymbol = prod.first;
        vector<vector<Item>> rightList = prod.second;

        vector<vector<Item>> hasLr, noLr;
        for(vector<Item> iList: rightList) if(iList.front() == leftSymbol)
hasLr.push_back(iList); else noLr.push_back(iList);

        if(hasLr.empty()) prodMapAns[leftSymbol] = rightList;
        else
        {
            // S -> S alpha | beta

            // S -> beta S'
            // S' -> alpha S' | E
            Item augmentedLeftSymbol = makeSymbol(leftSymbol.val + "'");
            vector<vector<Item>> alphaList, betaList;

            for(vector<Item> lol: hasLr)
            {
                vector<Item> v;
                for(int i = 1; i < lol.size(); i++) v.push_back(lol[i]);
                alphaList.push_back(v);
            }

            for(vector<Item> lol: noLr)
            {
                vector<Item> v;
                for(int i = 0; i < lol.size(); i++) v.push_back(lol[i]);
                betaList.push_back(v);
            }

            for(vector<Item> lol: betaList)
            {
                vector<Item> v;
                if(!(lol[0] == makeTerminal("E"))) for(int i = 0; i < lol.size(); i++)
v.push_back(lol[i]);
                v.push_back(augmentedLeftSymbol);
                prodMapAns[leftSymbol].push_back(v);
            }

            for(vector<Item> lol: alphaList)
            {
                vector<Item> v;
                // alphaList is empty here (if a -> a) ...
                if(!lol.empty())
                {
                    for(int i = 0; i < lol.size(); i++) v.push_back(lol[i]);
                    v.push_back(augmentedLeftSymbol);
                    prodMapAns[augmentedLeftSymbol].push_back(v);
                }
            }

            prodMapAns[augmentedLeftSymbol].push_back({makeTerminal("E")});
        }
    }

    return prodMapAns;
}




Map removeNonImmediateLeftRecursion(Map &prodMap)
{
    // ALGORITHM -->
    //
    // for i = 1 to n
    // {
    //     for j = 1 to i - 1
    //     {
    //         replace each production Ai → Ajγ
    //         by productions Ai → δ1γ| δ2γ| … … . . |δKγ
    //         where Aj → δ1|δ2| … … . |δK
    //     }
    //     Remove immediate Left Recursion among Ai productions.
    // }

    Map prodMapAns;
    Set visSymbols;

    for(auto &p: prodMap)
    {
        Item symbol = p.first;
        SetProd prodRhsCur, prodRhsCurProc;
        for(auto lol: p.second) prodRhsCur.insert(lol);


        bool changed = true;
        while(changed)
        {
            changed = false;
            for(const vector<Item> &itemList: prodRhsCur)
            {
                if(visSymbols.find(itemList[0]) != visSymbols.end())
                {
                    Item recursedSymbol = itemList[0];
                    vector<Item> procItemList(itemList.begin() + 1, itemList.end());
                    changed = true;

                    for(vector<Item> &itemListRecursed: prodMapAns[recursedSymbol])
                    {
                        if(itemListRecursed[0] == makeTerminal("E"))
prodRhsCurProc.insert(procItemList);
                        else
                        {
                            vector<Item> newProd;
                            for(Item i: itemListRecursed) newProd.push_back(i);
                            for(Item i: procItemList) newProd.push_back(i);
                            prodRhsCurProc.insert(newProd);
                        }
                    }
                }
                else prodRhsCurProc.insert(itemList);
            }

            prodRhsCur = prodRhsCurProc;
            prodRhsCurProc.clear();
        }

        vector<vector<Item>> prodTemp(prodRhsCur.begin(), prodRhsCur.end());
        Map prodCur; prodCur[symbol] = prodTemp;
        visSymbols.insert(symbol);
        prodCur = removeImmediateLeftRecursion(prodCur);

        for(auto &p: prodCur) prodMapAns[p.first] = p.second;
    }

    return prodMapAns;
}
```

- **Generating Nullable, First and Follow Mappings for Symbols:** The nullable, first and follow mappings have to be generated which are needed for the generation of LL(1) Parser. For computing all the 3 maps, we keep updating the maps (using their inference rules) till no more updates occur on the maps. The rules are →

- ❖ **Nullable:** If a symbol S directly derives `EPSILON (S -> E)`, then it is nullable. Also, if in a production, `S -> `S_1 S_2 … S_i … S_n, all of S_i (`i = 1 to n`) are nullable , then S is also nullable.
- ❖ **First Map:** First set for any symbol in the grammar contains the terminal symbol which can be found at the beginning of any string which can be generated from the symbol.

  For example, if we consider the production `S -> `S_1 S_2 … S_i … S_n, then $\text{first}(S) = \text{first}(S_1) \cup$
  $(\text{if nullable}(S_1) \text{ then first}(S_2)) \cup \ldots \cup$
  $(\text{if nullable}(S_1) \,\&\&\, \text{nullable}(S_2) \,\&\&\, \ldots \,\&\&\, \text{nullable}(S_{n-1})$
  $\text{then first}(S_n))$
- ❖ **Follow Map:** Follow of a symbol, say A is the set of all terminals that can appear immediately to the right of A in some sentential form.

  For example, if we consider the non terminal S_i, for all productions of the form `S -> `S_1 S_2 … S_i … S_n, containing S_i
  $\text{then follow}(S_i) = \text{first}(S_{1+1}) \cup$
  $(\text{if nullable}(S_{i+1}) \text{ then first}(S_{i+2})) \cup \ldots \cup$
  $(\text{if nullable}(S_{i+1}) \,\&\&\, \text{nullable}(S_{i+2}) \,\&\&\, \ldots \,\&\&$
  $\text{nullable}(S_n) \text{ then first}(S)).$

```
typedef unordered_map<Item, vector<vector<Item>>, ItemHasher> Map;
typedef unordered_set<Item, ItemHasher> Set;
typedef unordered_map<Item, Set, ItemHasher> MapFirst;
typedef unordered_map<Item, Set, ItemHasher> MapFollow;
typedef unordered_map<Item, bool, ItemHasher> MapNullable;

MapNullable generateNullableMap(Map &prodMap, Set &symbols)
{
    MapNullable mapNullable;

    Set remainingSymbols;
    for(Item i: symbols) if(find(prodMap[i].begin(), prodMap[i].end(),
vector<Item>({EPSILON_TERMINAL})) != prodMap[i].end()) mapNullable[i] = true;
                        else remainingSymbols.insert(i);

    bool updated = true;
    while(updated && !remainingSymbols.empty())
    {
        updated = false;

        for(Item symbol: remainingSymbols)
        {
            bool nullableFound = false;
            for(vector<Item> ItemList: prodMap[symbol])
            {
                bool notNullableItemFoundInProd = false;
                for(Item i: ItemList) if((i.isSymbol && !mapNullable[i]) || !i.isSymbol){
notNullableItemFoundInProd = true; break; }

                if(!notNullableItemFoundInProd){ nullableFound = true; break; }
            }

            if(nullableFound) updated = true, mapNullable[symbol] = true,
remainingSymbols.erase(symbol);
        }
    }

    for(Item symbol: remainingSymbols) mapNullable[symbol] = false;

    return mapNullable;
}

bool tryInsertHelper1(Item i, Set &s)
{
    if(s.find(i) != s.end()) return false;
    else s.insert(i); return true;
}

MapFirst generateFirstFromProductions(Map &prodMap, Set &symbols, MapNullable &mapNullable)
{
    MapFirst mapFirst;
    Item firstElem;

    // Flooding with immediate non-terminals ...
    for(Item symbol: symbols)
    {
        for(vector<Item> itemList: prodMap[symbol])
        {
            firstElem = itemList.front();
            if(!firstElem.isSymbol && firstElem != EPSILON_TERMINAL)
mapFirst[symbol].insert(firstElem);
        }
    }

    // Solving left recursive cases with updates done ...
    // We know immediate and non-immediate left recursions have been removed here ...
    bool updated = true;
    while(updated)
    {
        updated = false;

        for(Item symbol: symbols)
        {
            for(vector<Item> itemList: prodMap[symbol])
            {
                if(itemList.front() == EPSILON_TERMINAL) continue;

                for(Item i: itemList)
                {
                    if(i.isSymbol){ for(Item fi: mapFirst[i]) if(tryInsertHelper1(fi, 
mapFirst[symbol])) updated = true; if(!mapNullable[i]) break; }
                    else{ if(tryInsertHelper1(i, mapFirst[symbol])) updated = true; break; 
/* non-terminal is non-nullable ... */  }
                }
            }
        }
    }

    return mapFirst;
}




Set deduceFirstHelper(int pos, vector<Item> &prod, MapNullable &mapNullable, MapFirst 
&mapFirst)
{
    Set curFirstSet;
    Item i;

    for(; pos < prod.size(); pos++)
    {
        i = prod[pos];

        if(i.isSymbol){ for(Item fi: mapFirst[i]) curFirstSet.insert(fi); 
if(!mapNullable[i]) break; }
        else{ curFirstSet.insert(i); break;  /* non-terminal is non-nullable ... */  }
    }

    return curFirstSet;
}

MapFollow generateFollowFromProductions(Map &prodMap, Set &symbols, MapNullable 
&mapNullable, MapFirst &mapFirst, Item startSymbol)
{
    MapFollow mapFollow;
    Item i;

    // First step ...
    mapFollow[startSymbol].insert(END_TERMINAL);
    // Two rules -->
    // (1) For each production of the form A -> αBβ, add the first set of β to the follow
set of B, except for ε.
    // (2) For each production of the form A -> αB, add the follow set of A to the follow
set of B.

    for(Item symbol: symbols)
    {
        for(vector<Item> itemList: prodMap[symbol])
        {
            if(itemList.front() == EPSILON_TERMINAL) continue;

            for(int j = 0; j < itemList.size(); j++)
            {
                i = itemList[j];
                if(!i.isSymbol) continue;

                // First Type ...
                if(j != itemList.size() - 1){ for(Item f: deduceFirstHelper(j + 1,
itemList, mapNullable, mapFirst)) tryInsertHelper1(f, mapFollow[i]);
                }
            }
        }
    }

    bool updated = true; int cnt = 1;
    while(updated)
    {
        updated = false;

        for(Item symbol: symbols)
        {
            for(vector<Item> itemList: prodMap[symbol])
            {
                if(itemList.front() == EPSILON_TERMINAL) continue;
                for(int j = itemList.size() - 1; j >= 0; j--)
                {
                    // Second Type ...
                    i = itemList[j];
                    if(i.isSymbol){ for(Item f: mapFollow[symbol]) if(tryInsertHelper1(f,
mapFollow[i])) updated = true; if(!mapNullable[i]) break; }
                    else break;  /* non-terminal is non-nullable ... */
                }
            }
        }
    }
    return mapFollow;
}
```

- **Generating Initial LL(1) Parsing Table:** Now, the initial Parsing Table is generated using the following rules →

  In the LL(1) parsing table, the row-headings are symbols and the column-headings are terminal. Based on the mentioned symbol and the terminal, the productions are placed in the table. A production of the form `S -> S`$_1$ `S`$_2$ `… S`$_i$ `… S`$_n$`,` is placed in the position `(S, First(S`$_1$ `S`$_2$ `… S`$_i$ `… S`$_n$`))` in the LL(1) parsing table. If S is nullable, then the production is also placed in the position `(S, Follow(S))`

```
bool deduceNullableHelper(vector<Item> &prod, MapNullable &mapNullable)
{
    for(Item i: prod)
        if(i.isSymbol){ if(!mapNullable[i]) return false; }
        else return false;  /* non-terminal is non-nullable ... */

    return true;
}

bool tryInsertHelper2(Item symbol, Item terminal, Action a, ParsingTable &parsingTable)
{
    if(parsingTable[symbol][terminal].isInvalid()){ parsingTable[symbol][terminal] = a;
return true; }
    else return parsingTable[symbol][terminal] == a;  // Because of multiple reasons , same
production can be inserted ...
}

void tryInsertHelper3(Item symbol, Item terminal, int index, ParsingTable &parsingTable)
{
    Action a = makeProduceAction(symbol, index);
    if(!tryInsertHelper2(symbol, terminal, a, parsingTable)) {
        cout << "AMBIGUOUS GRAMMAR FOUND !!!" << endl;
        cout << "parsingTable[" << symbol << "][" << terminal << "]  -->  " <<
parsingTable[symbol][terminal] << "  and  " << a << endl;
        exit(0);
    }
}

ParsingTable generateLl1ParsingTable(Map &prodMap, Set &symbols, MapNullable &mapNullable,
MapFirst &mapFirst, MapFollow &mapFollow)
{
    // parsingTable[Symbol][Terminal] -> Action ...
    ParsingTable parsingTable;

    for(Item symbol: symbols)
    {
        int index = -1;
        for(vector<Item> itemList: prodMap[symbol])
        {
            index++;
            if(itemList.front() == EPSILON_TERMINAL)
            {
                for(Item terminal: mapFollow[symbol]) tryInsertHelper3(symbol, terminal, 
index, parsingTable);
                continue;
            }
            for(Item i: itemList)
            {
                for(Item terminal: deduceFirstHelper(0, itemList, mapNullable, mapFirst)) 
tryInsertHelper3(symbol, terminal, index, parsingTable);
                if(deduceNullableHelper(itemList, mapNullable)) for(Item terminal: 
mapFollow[symbol]) tryInsertHelper3(symbol, terminal, index, parsingTable);
            }
        }
    }

    return parsingTable;
}
```

- **Augmenting LL(1) Parsing Table with POP & SCAN:** Now, every `INVALID_ACTION` entry is replaced by a `POP` or `SCAN` action using the following rules →
    - ❖ If the terminal of the index of the table is end terminal (`'$'`) or the terminal is present in the follow set of the symbol , then action is `'POP'`
    - ❖ Else , the action is `'SCAN'`

```
void augmentLl1ParsingTableWithErrorRecovery(Map &prodMap, Set &symbols, Set &terminals, 
MapNullable &mapNullable, MapFirst &mapFirst, MapFollow &mapFollow, ParsingTable 
&parsingTable)
{
    for(Item symbol: symbols)
    {
        for(Item terminal: terminals)
        {
            if(!parsingTable[symbol][terminal].isInvalid()) continue;
            if(terminal == END_TERMINAL || mapFollow[symbol].find(terminal) != 
mapFollow[symbol].end()) parsingTable[symbol][terminal] = POP_ACTION;
            else parsingTable[symbol][terminal] = SCAN_ACTION;
        }
    }
}
```

- **Parsing Token Stream using LL(1) Parsing Table and Saving Parsing History:** We use the Parsing Table, keep a current Token Stream (containing tokens ,i.e., terminals) and Parsing Stack (containing symbols and terminals) and parse according to the following rules →
    - Parsing Stack starts with the value “`S $`” (where S is starting symbol)
    - If both terminals at top of stack and token stream match, they are popped , else throw fatal error
    - Else if the top of stack has a symbol, decide the action by looking from the corresponding Parsing Table entry

```
pair<pair<bool, string>, ParsingHistory> parseTokenStreamUsingParsingTable(vector<Token>
&curTokenStream, Set &symbols, Set &terminals, Item startSymbol, ParsingTable
&parsingTable, Map &prodMap)
{
    ParsingHistory parsingHistory;
    vector<Item> curParsingStack({END_TERMINAL, startSymbol});
    Action a;
    Item i;
    Token t;
    bool errorFound = false;
    int tokenPos = 0;

    while(tokenPos < curTokenStream.size() || !curParsingStack.empty())
    {
        // Decide Action ...
        i = curParsingStack.back();
        t = curTokenStream[tokenPos];
        if(i.isSymbol) a = parsingTable[i][t.tokenType];
        else
        {
            if(i == t.tokenType) a = MATCH_TOKEN_ACTION;
            else a = INVALID_ACTION;
        }

        // Save History ...
        vector<Token> tempTokenStream({curTokenStream.begin() + tokenPos,
curTokenStream.end()});
        parsingHistory.push_back({curParsingStack, tempTokenStream, a});

        // Perform Action ...
        // curParsingStack.pop_back();
        if(a == MATCH_TOKEN_ACTION) tokenPos++, curParsingStack.pop_back();
        else if(a == POP_ACTION) curParsingStack.pop_back(), errorFound = true;
        else if(a == SCAN_ACTION) tokenPos++, errorFound = true;
        else if(a.isInvalid()) return {{true, "Fatal Error"}, parsingHistory};
        else{ curParsingStack.pop_back(); for(int j =
prodMap[a.prodLhs][a.prodRhsIndex].size() - 1; j >= 0; j--)
if(prodMap[a.prodLhs][a.prodRhsIndex][j] != EPSILON_TERMINAL)
curParsingStack.push_back(prodMap[a.prodLhs][a.prodRhsIndex][j]); }
    }

    return {{errorFound, ((errorFound)? "Recovered Error": "No Error")}, parsingHistory};
}
```

H. Conclusion

In this project, an attempt was made to parse and validate `SQL DML` statements involving `select`, `insert` and `delete` operation including `where`, `group by` and `having` clauses. Since our project work was based on the validity of statements using lexical analysis and syntax analysis, the designed architecture can only check lexical and syntax errors.

However, the program is incapable of performing semantic analysis, hence errors involving semantics (e.g. validating if attribute is present in a table in `table.attribute` format or in `SELECT attribute FROM table` format) is not considered. Also the nested statements were not considered as valid statements in this project because it was the requirement in the question, else it could have been easily done by changing the productions using our current program.

[End of Report]