Compiler Design Project

Team Members:
Kanko Ghosh (002010501035)
Mahfujul Haque (002010501036)
Rounak Bhattacharjee (002010501041)
Soumyajit Rudra Sarma (002010501068)

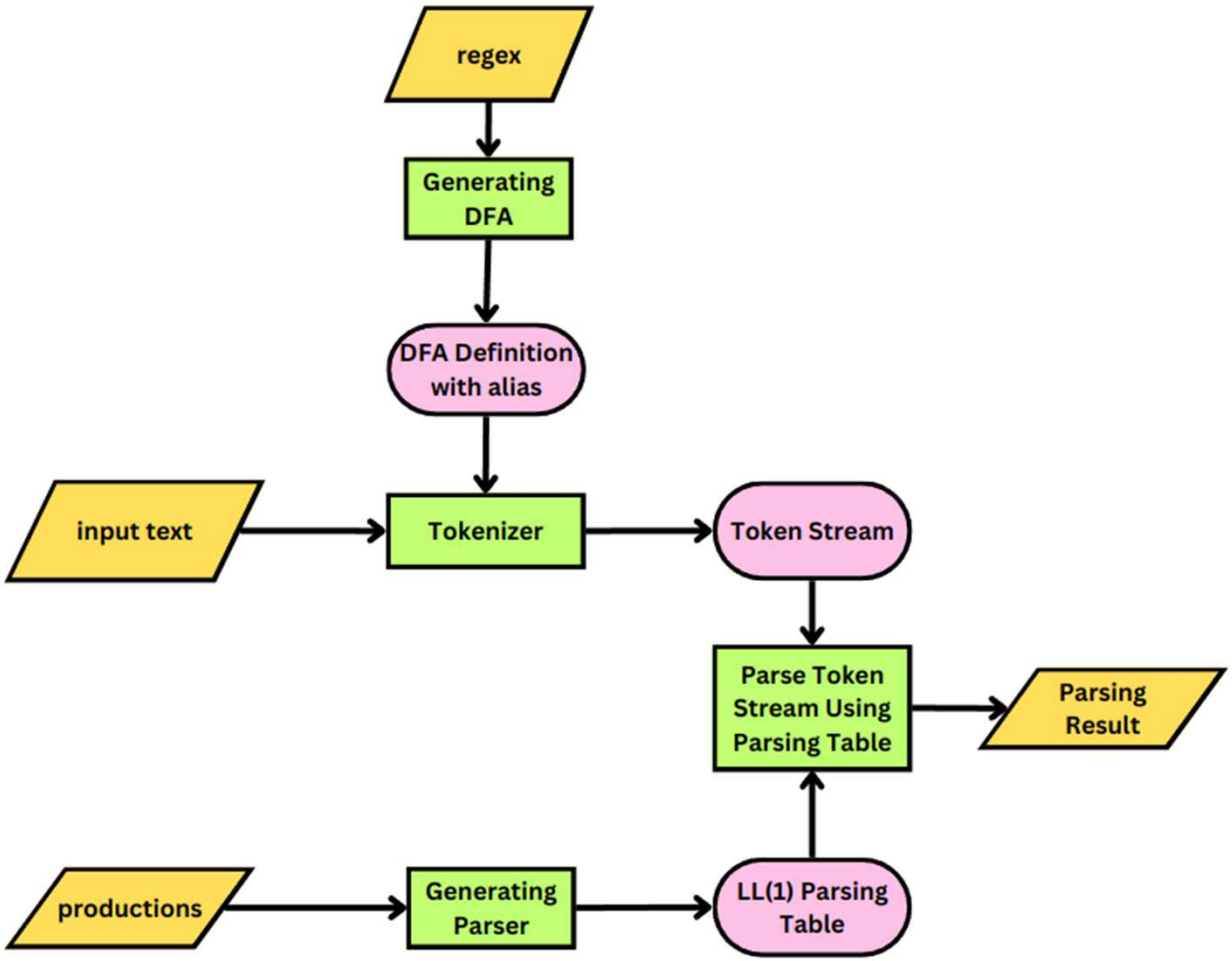
Problem Statement

- Parsing DML of SQL language using LL(1) grammar
- Validates Insertion, Deletion and Select statements
- Other Features
  - Support for where, group by and order by clauses
  - Conditional and Relational operators
  - Support for tablename.attribute notation

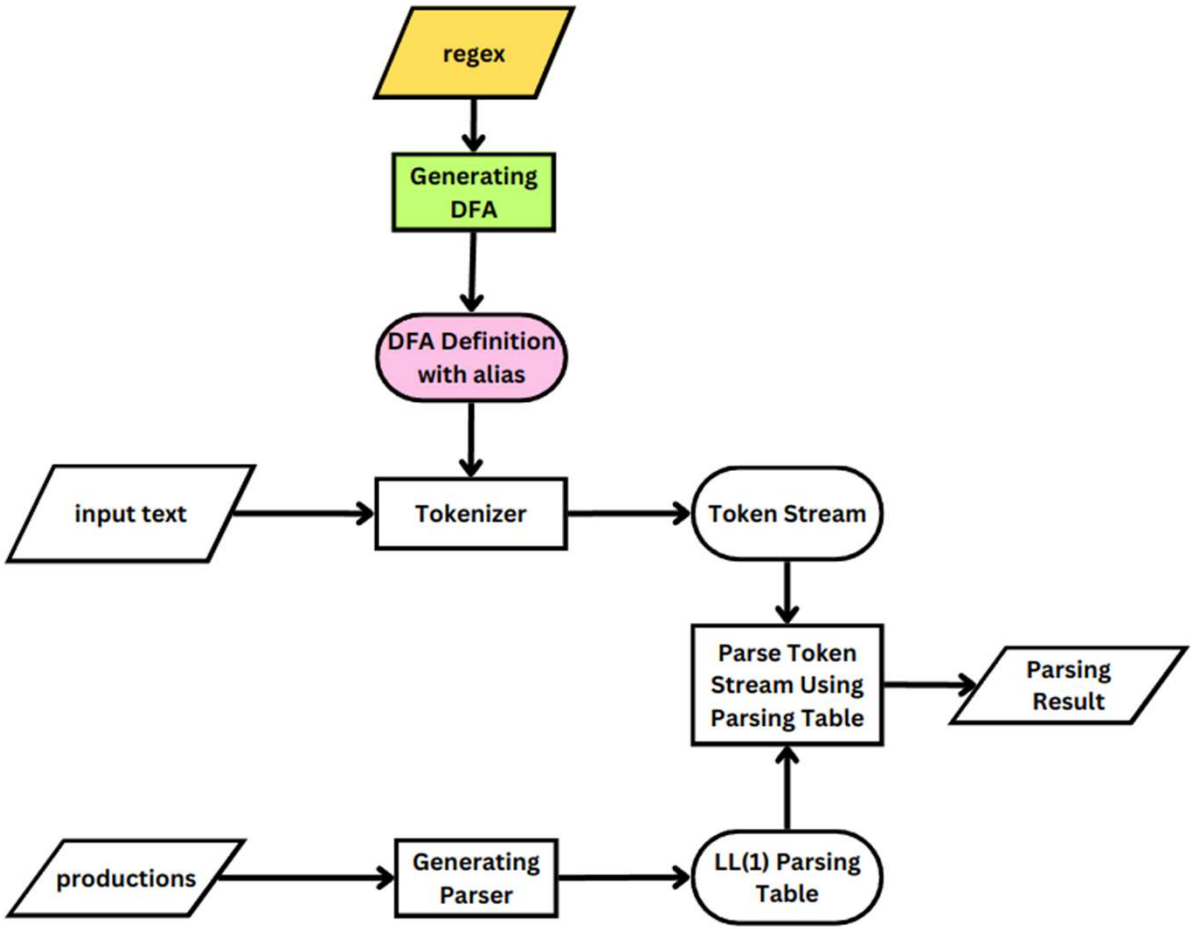
Technologies Used

- Major programming language: **C++**
  - Creation of Lexical analyser
  - Creation of LL(1) parsing table
  - Validating SQL statements for syntax errors
- Visualisation: **Igraph** library of **python**

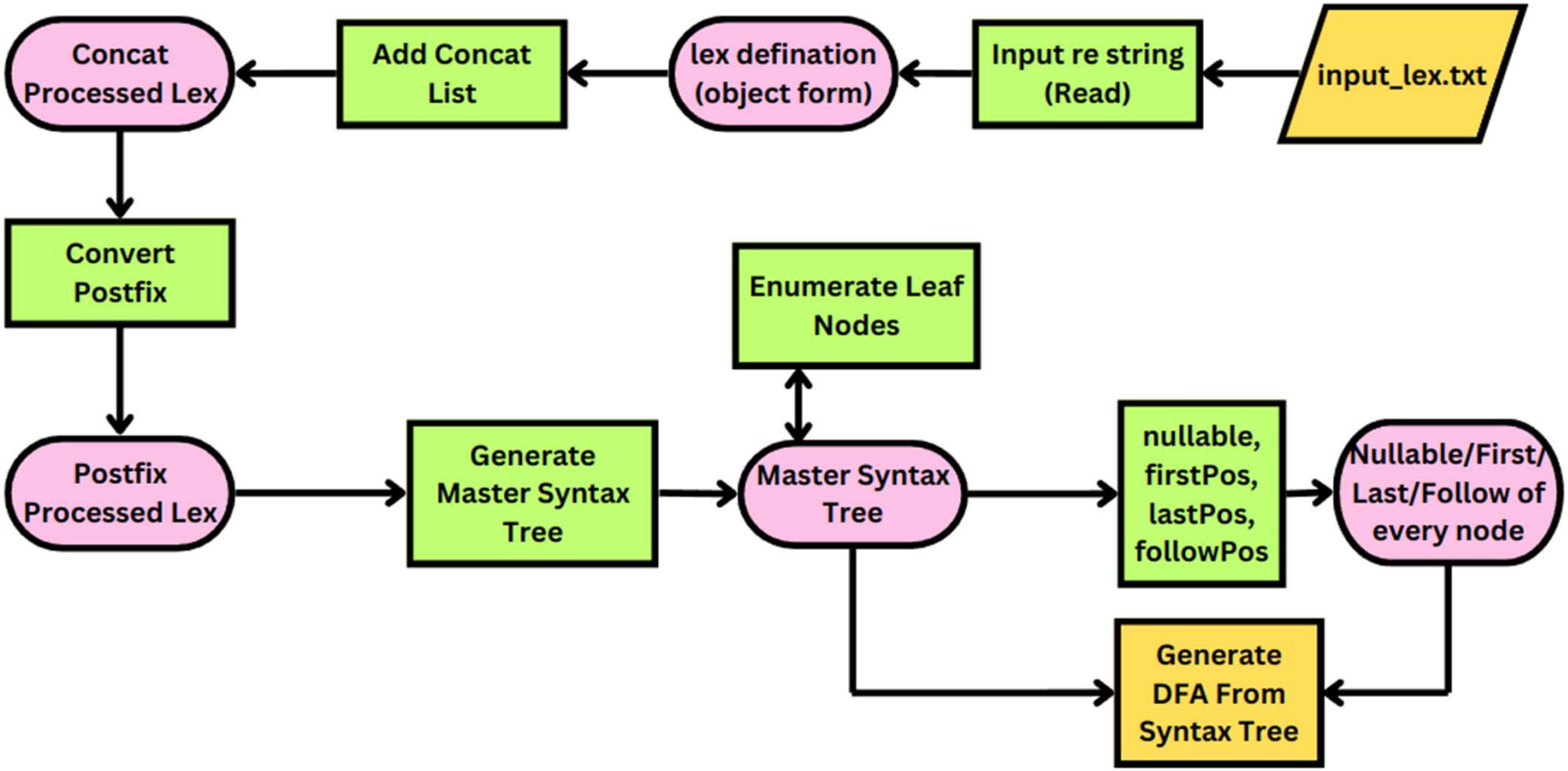
Overall Flow Diagram



Generating DFA



Classes and functions for DFA creation



Generating DFA for Lexical Analysis

- It is a multi step procedure which consists of
  - Creation of the Regular expressions
    - Concat processed lex definition
    - Postfix processed lex definition
  - Creating Master Syntax tree for the regular expressions
    - Creating syntax tree for each regular expression
  - Evaluation of nullable, first-pos, last-pos and follow-pos of every node, followed by annotation
  - Creating DFA from syntax tree (with alias)

Regular Expressions

```
1 DELETE  DELETE
2 INTO    INTO
3 INSERT  INSERT
4 >=      >=
5 >       >
6 =       =
7 <=      <=
8 <       <
9 \)      )
10 \(     (
11 VALUES VALUES
12 \.     .
13 ;      ;
14 BY     BY
15 OR     OR
16 SELECT SELECT
17 WHERE  WHERE
18 GROUP  GROUP
19 ORDER  ORDER
20 HAVING HAVING
21 AND    AND
22 FROM   FROM
23 ,      ,
24 NOT    NOT
25 \n+\t+ SEP
26 (a+b+....+y+z)(1+2+..+8+9+0+a+b+...+y+z)*
27 ID
```

- There are 26 Regular expressions
- For each line (except the last two), the two values separated by tab indicate the **regular expression** and the **token type** returned by that regular expression

Concat Processed Lex Definition

- Inserting Concatenation operator appropriately
- Example:
  - $\text{DELETE} \xrightarrow{\text{Concatenation}} \text{D.E.L.E.T.E}$
  - $\text{a+bcd+ef(gf(e+fg)*hi)jkl} \xrightarrow{\text{Concatenation}} \text{a+b.c.d+e.f.(g.f.(e+f.g)*.h.i).j.k.l}$

Postfix Processed Lex

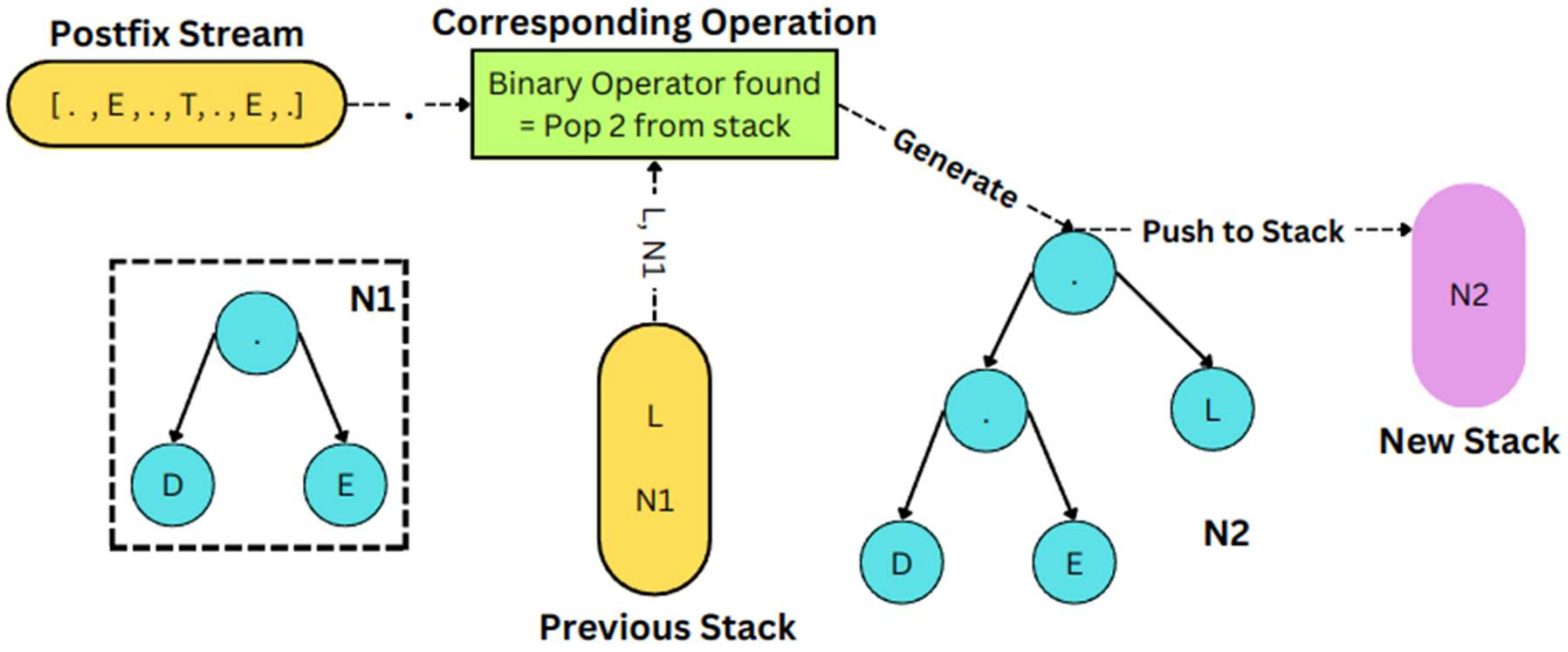
- Converting Infix regular expression to postfix form
- Purpose: Creating syntax tree from postfix form is easier (using Stack)

$$D.E.L.E.T.E \xrightarrow{\text{Postfix}} DE.L.E.T.E.$$

$$a+b.c.d+e.f.(g.f.(e+f.g)^*.h.i).j.k.l \xrightarrow{\text{Postfix}} abc.d.+ef.gf.efg.+^*.h.i..j.k.l.+$$

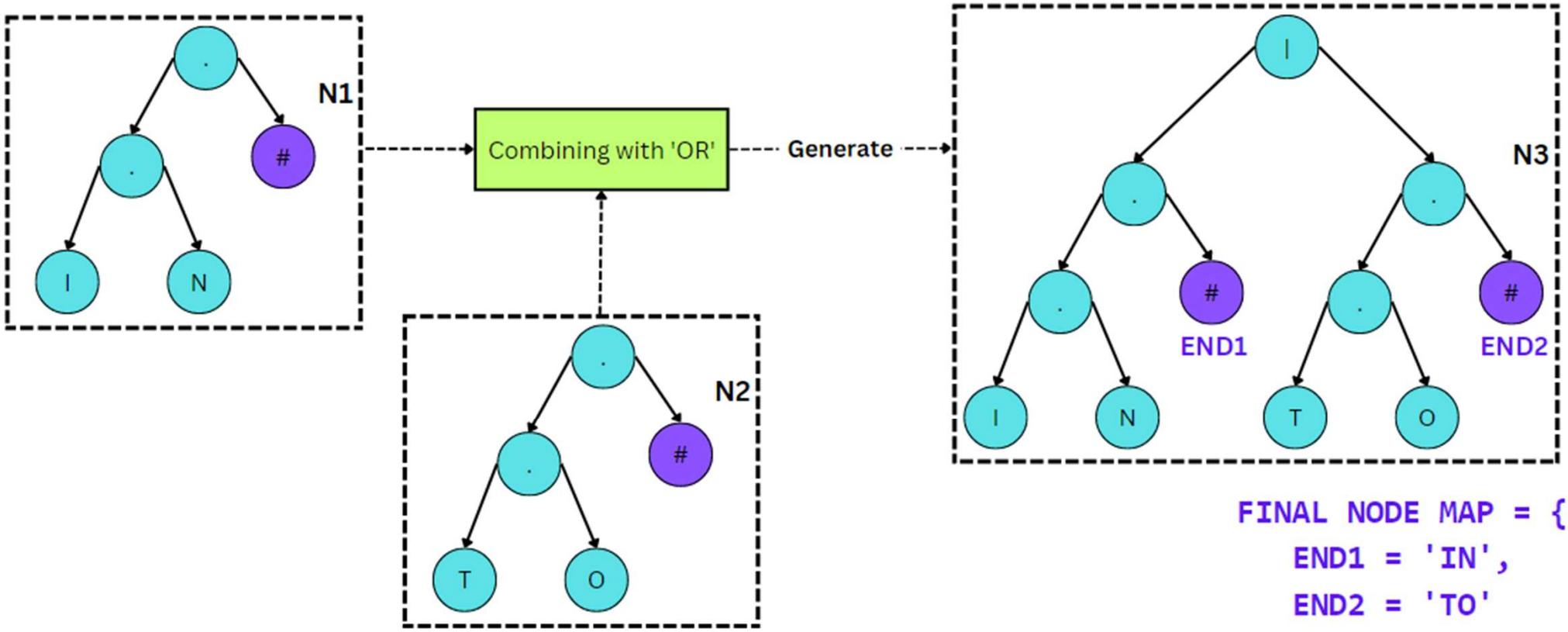
Syntax Tree from Postfix R.E.

- Nodes generated by postfix evaluation using stack
- At the end of reading Postfix Expression, only one node should remain in stack, else error.
- Root Node = concatenation operator node with left child (from stack) and right child (#)

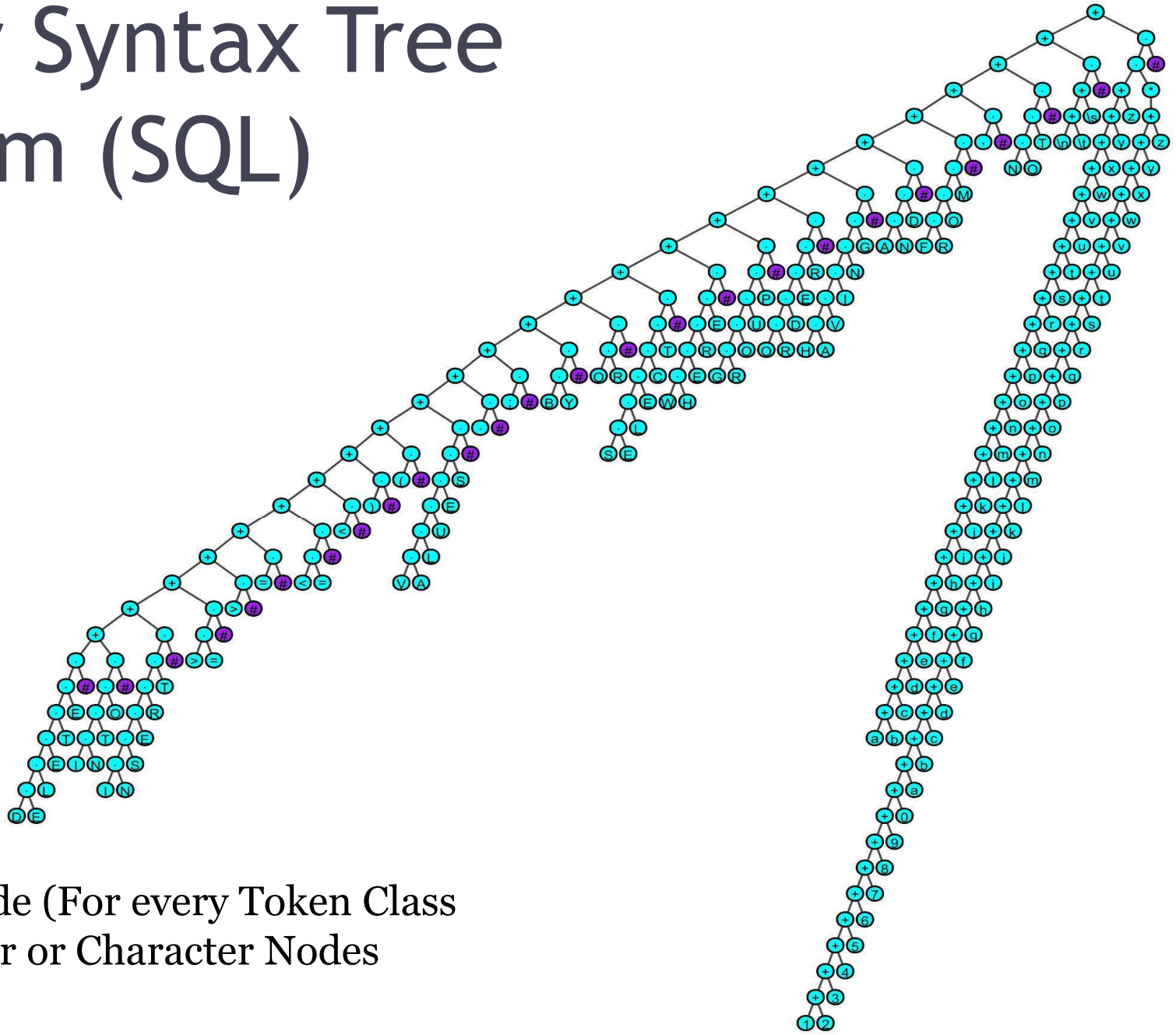


Master Syntax Tree from Individual Trees

- All the individual syntax trees are Merged together using OR ('|') operator node.

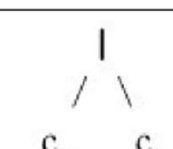
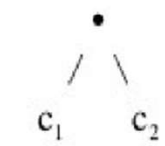
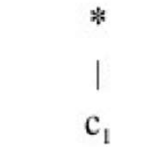


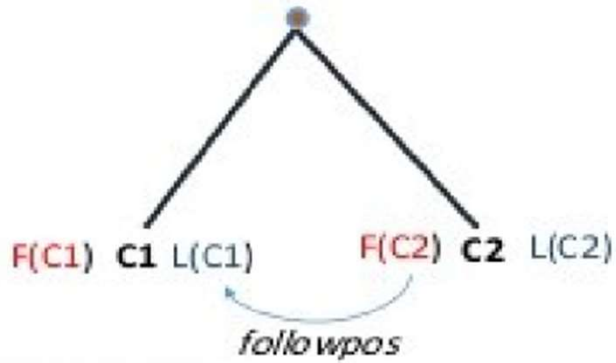
Master Syntax Tree Diagram (SQL)



Purple: # Node (For every Token Class
Blue: Operator or Character Nodes

Nullable, First-pos, Last-pos, Follow-pos,

Node n	$nullable(n)$	$firstpos(n)$	$lastpos(n)$
Leaf ε	true	∅	∅
Leaf i	false	$\{i\}$	$\{i\}$
	$nullable(c_1)$ or $nullable(c_2)$	$firstpos(c_1)$ ∪ $firstpos(c_2)$	$lastpos(c_1)$ ∪ $lastpos(c_2)$
	$nullable(c_1)$ and $nullable(c_2)$	**if** $nullable(c_1)$ **then** $firstpos(c_1) \cup firstpos(c_2)$ **else** $firstpos(c_1)$	**if** $nullable(c_2)$ **then** $lastpos(c_1) \cup lastpos(c_2)$ **else** $lastpos(c_2)$
	true	$firstpos(c_1)$	$lastpos(c_1)$



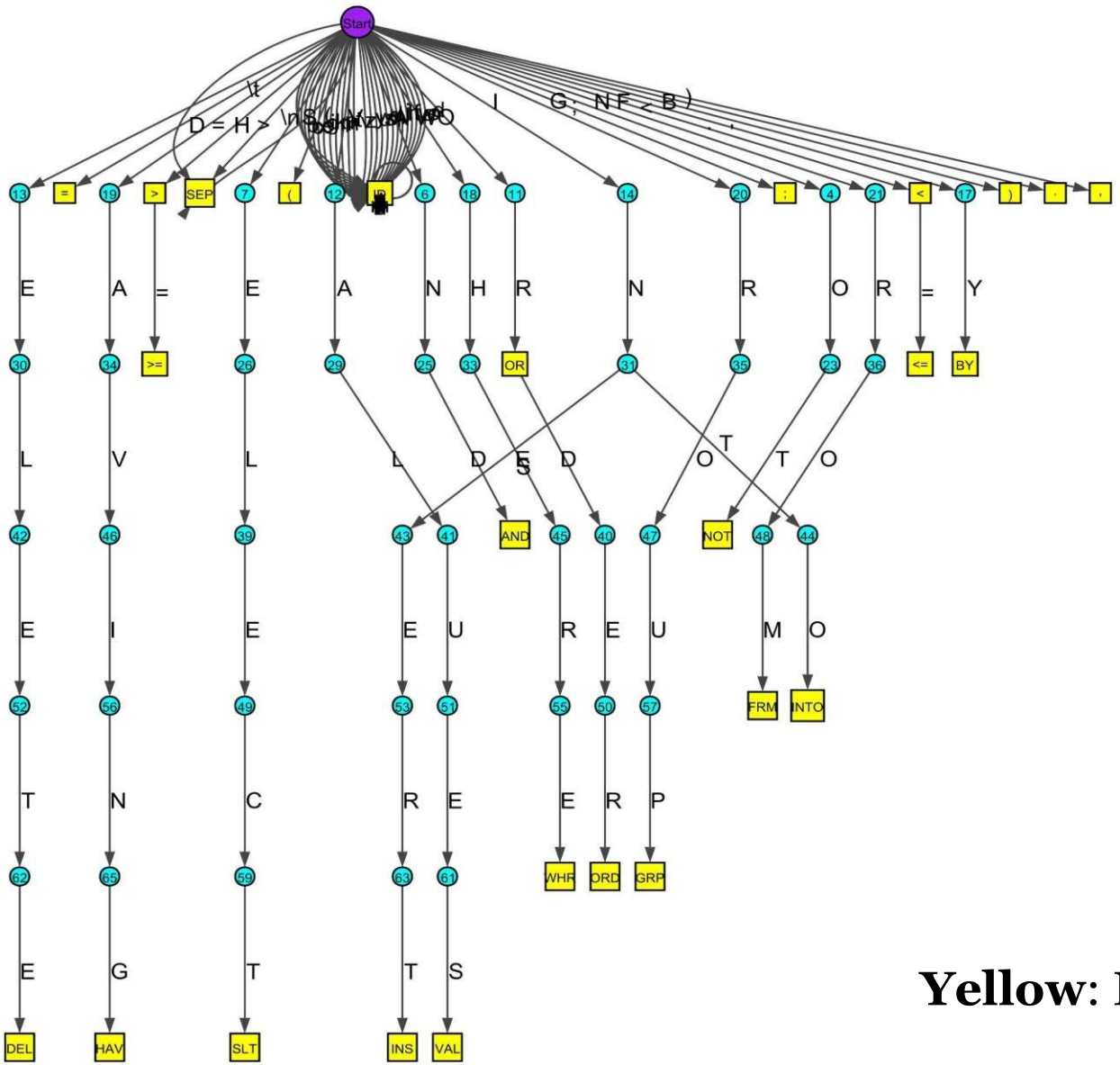

```
#104 -->
Nullable: False,
First Pos: 104,
Last Pos: 104,
Follow Pos: 154, 162, 137, 135, 155, 136, 151, 153, 132, 147, 164, 165, 156, 133, 131, 152,
150, 160, 158, 134, 130, 146, 143, 144, 159, 140, 163, 149, 145, 138, 139, 129, 141, 142,
148, 161, 157,
```

DFA from Master Syntax Tree

- Initialize Dstates to contain only the unmarked state firstpos(n_o), where n_o is the root of syntax tree T for (r) #
- while (there is an unmarked state S in Dstates) {

```
    mark  S;
    for ( each input symbol a ) {
        let  U be the union of followpos(p)  for all p in S
        that correspond to a;
        if ( U is not in Dstates )
            add U as an unmarked state to Dstates; Dtran[S, a] = U;
        }
}
```

DFA Diagram

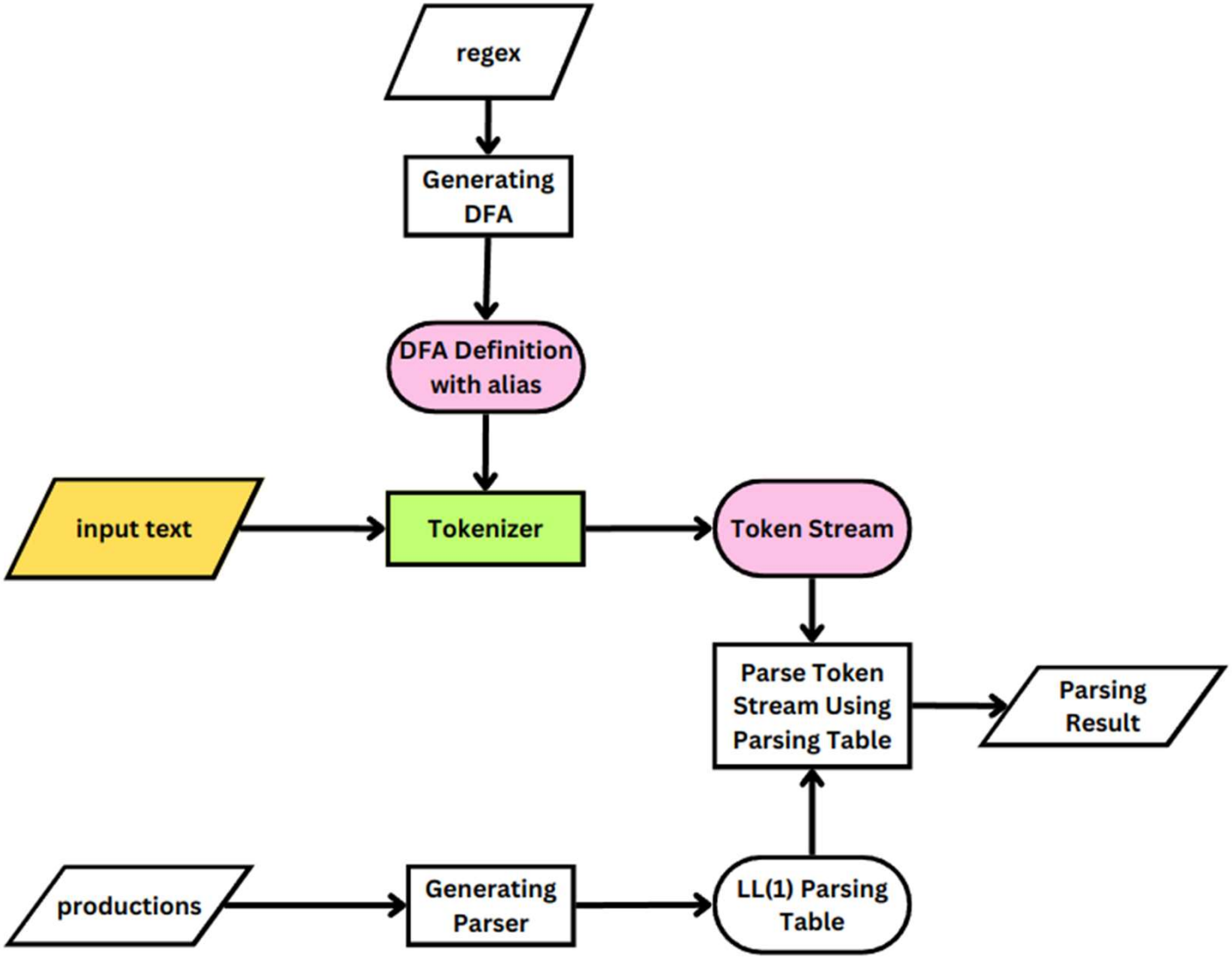



```
Size: 72

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 ,
```

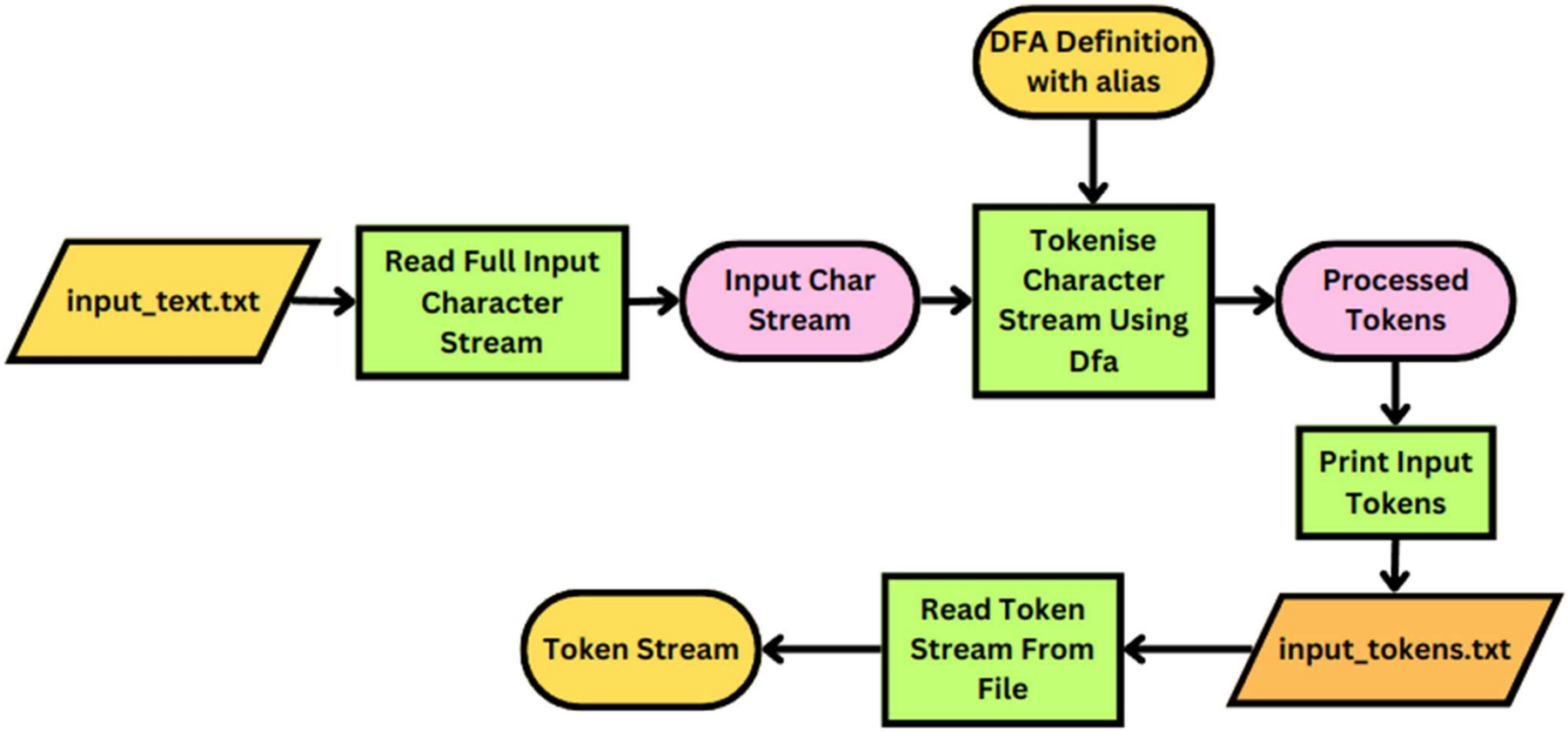
Purple: Start State
Blue: Intermediate State
Yellow: Final State (For a token class)

Alias: Set of tree nodes representing the DFA state

Tokenizing Input SQL Statements



Flow diagram of Tokenizing



Tokenized Stream SQL Preview

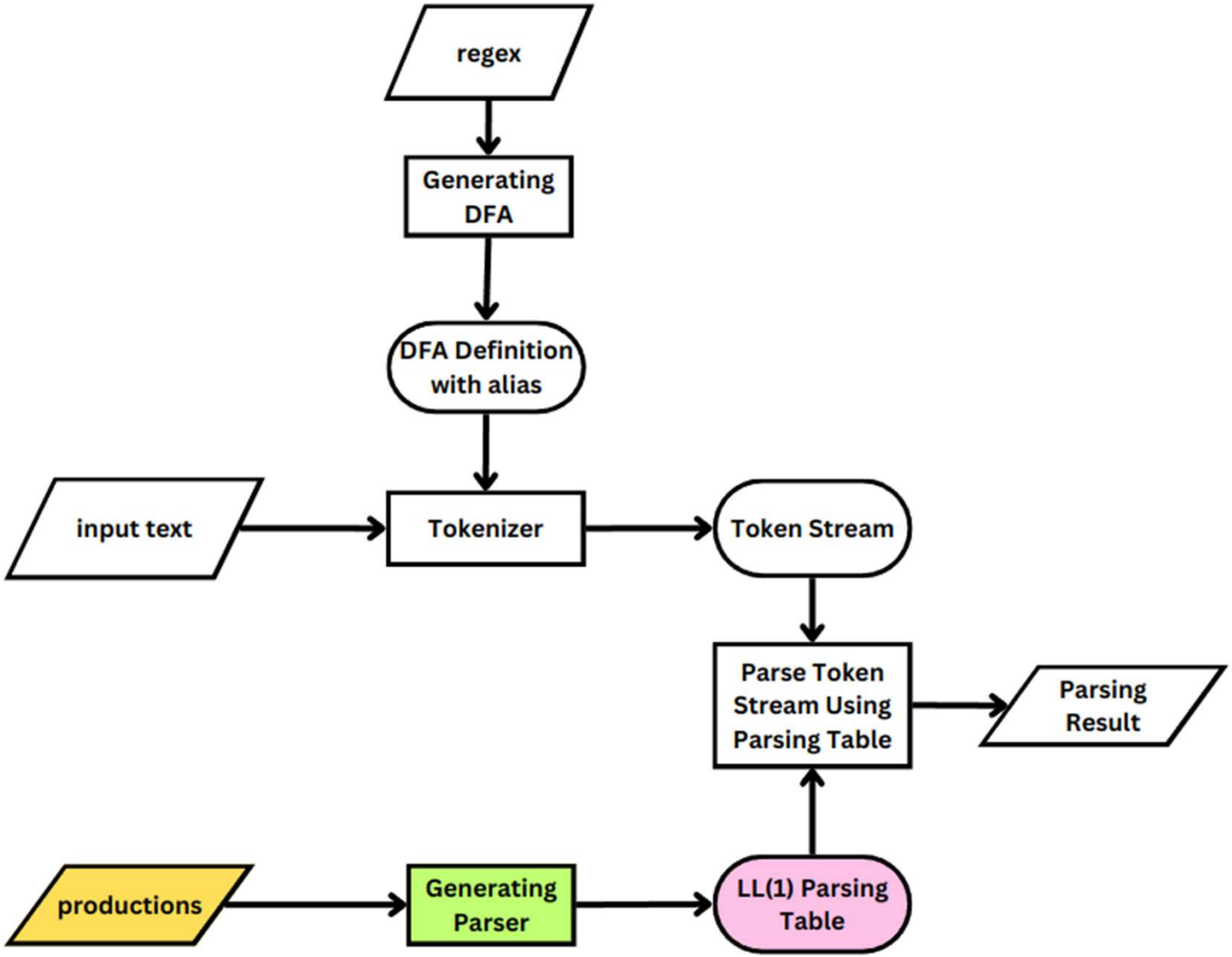
- **Input**

```
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;
```

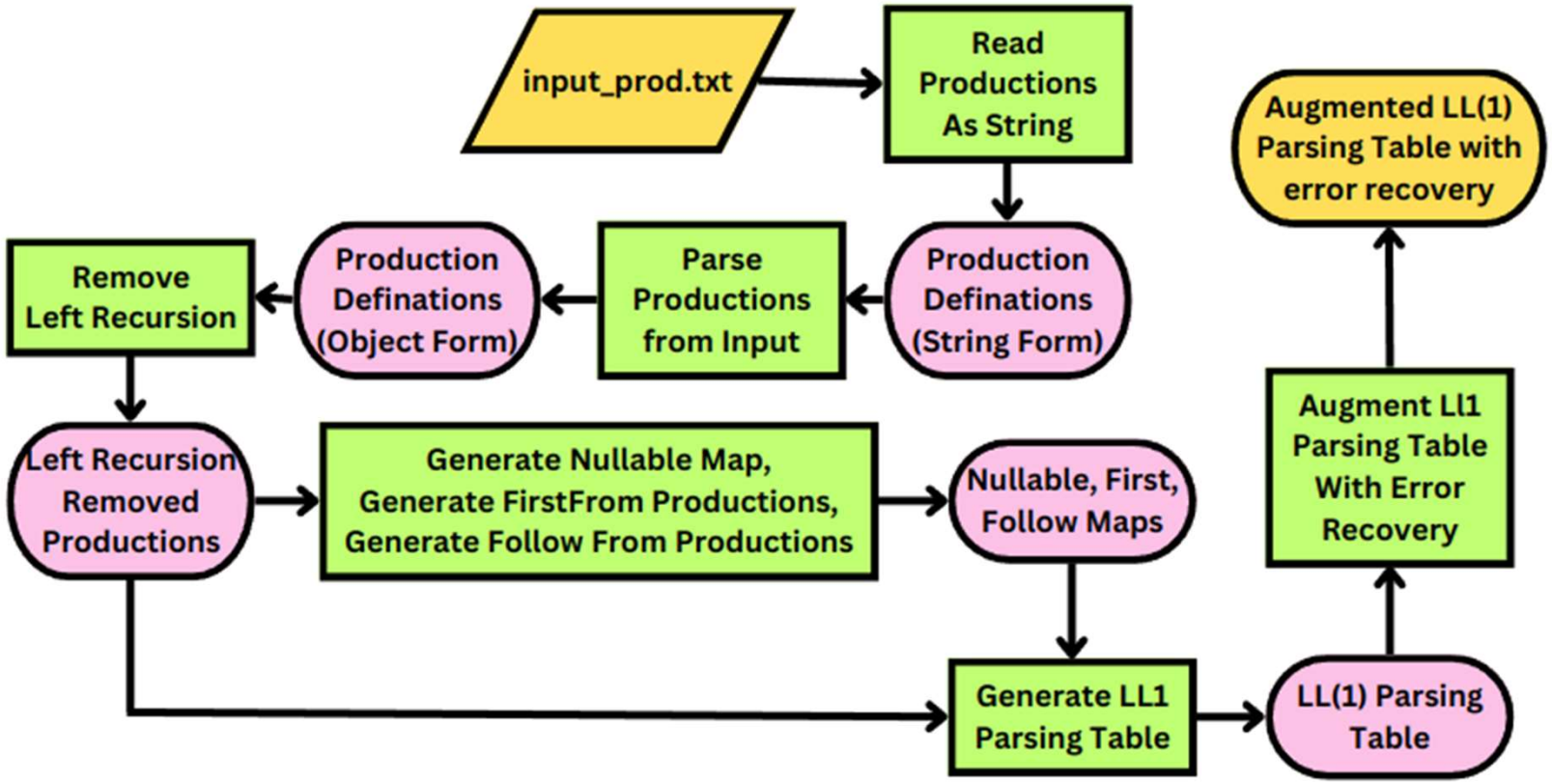
- **Tokenized Stream**

```
Size = 459

INSERT INSERT
INTO INTO
ID mytable
( (
ID atr1
, ,
ID atr2
, ,
ID atr3
, ,
```

Generating LL(1) Parser



Classes and Functions for LL(1) generation



Generating LL(1) Grammar for Parsing

- Pre Processing Productions for recognizing symbols
- Left Recursion Removal
- Left Factoring
- Generating Nullable, First and Follow of each Non terminals
- Generating LL(1) Parsing Table
- Augmenting LL(1) Parsing Table (pop and scan)

SQL Grammar

- Number of productions: 41
- Starting symbol: s

```
1 s -> stmt ; s
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
```

Pre-processing Productions

- Reading the productions from file and identification of Terminals and Non terminals
- Achieved in 2 passes
  - **Pass 1**: Recognizing Non Terminals from LHS of Productions (stored in a set)
  - **Pass 2**: Annotating every element in RHS as terminal or non terminal

```
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)
```

Left Recursion Removal

- Immediate Left Recursion Removal

$$S \rightarrow S\,a_1 \mid \ldots \mid S\,a_n \mid b_1 \mid \ldots \mid b_m \xrightarrow{\textit{left recurssion removal}} S \rightarrow b_1\,S' \mid \ldots \mid b_m\,S', \qquad S' \rightarrow a_1\,S' \mid \ldots \mid a_n\,S' \mid \varepsilon$$

- Non Immediate Left Recursion Removal

```
1 FOR i = 1 to n:
2     FOR j in 1 to i-1:
3         replace each production Ai -> Ajy by
4         productions Ai -> Ai → δ1γ| δ2γ| … … . . |δKγ
5         where Aj → δ1|δ2| … … . |δK
6     END FOR
7     remove immediate left recursion among Ai productions
8 END FOR
9
```

Left Factoring Removal

- It is done to avoid ambiguity while parsing based on one terminal occuring in first set of multiple productions

$$S \rightarrow aB \mid aC \xrightarrow{left\ factoring} S \rightarrow aD\ , D \rightarrow B|C$$

Generating Nullable, First and Follow of each Non terminals

For production **`S -> S1 S2 … Si … Sn`**

- Nullable:
  - all of S_i (i = 1 to n) are nullable
  - If a symbol S directly derives EPSILON (S -> E).
- First Map:
  - first(S) = first(S_1) ∪ (if nullable(S_1) then first(S_2)) ∪ … ∪ (if nullable(S_1) && nullable(S_2) && … && nullable(S_{n-1}) then first(S_n))
- Follow Map:
  - follow(S_i) = first(S_{i+1}) ∪ (if nullable(S_{i+1}) then first(S_{i+2})) ∪ … ∪ (if nullable(S_{i+1}) && nullable(S_{i+2}) && … && nullable(S_n) then first(S)).

Generating LL(1) parsing Table

- Row Head: **Non Terminals**
- Column Head: **Terminals**

- A production of the form **`S->S1 S2 … Si … Sn`**, 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))

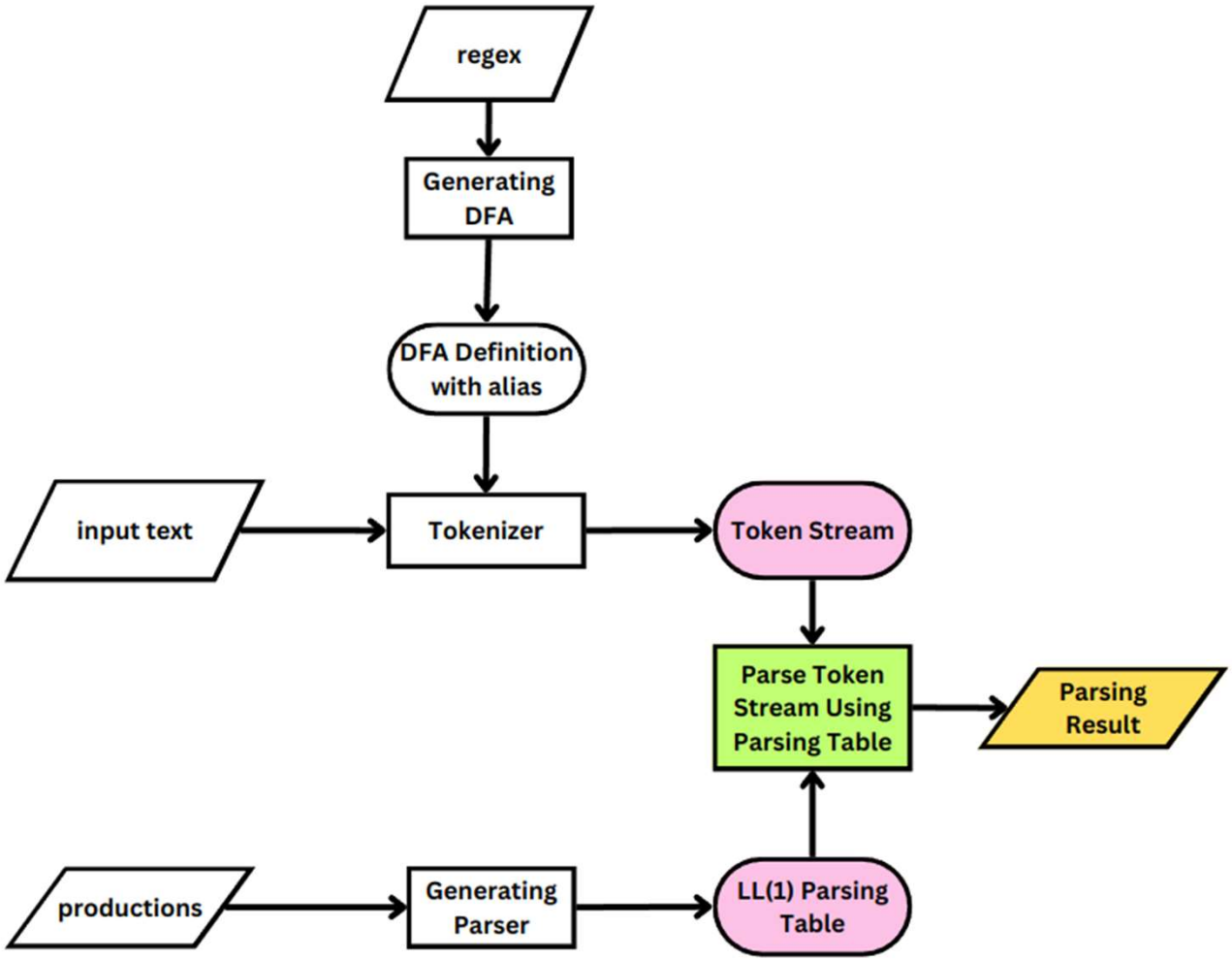
Augmenting LL(1) Parsing Table

- Pop and Scan are inserted in empty spaces
- **POP**:
  - terminal of the index of the table is end terminal (‘$’)
  - the terminal is present in the follow set of the symbol
- **SCAN**:
  - Everywhere else

Augmented LL(1) parsing table

	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

Validating SQL statements



Validating Using LL(1) Parser

- Components:
  - Parsing Table
  - Token Stream (containing tokens ,i.e., terminals)
  - Parsing Stack (containing symbols and terminals)
- 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
- Parsing History was also saved

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: 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)
```

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;
```

- **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(;), 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)
```

Thank You

Any Questions?