Experiment No. 09

Aim- Implementation of parser

Requirement- Java and printout pages

Theory- A parser is a compiler or interpreter component that breaks data into smaller elements for easy translation into another language. A parser takes input in the form of a sequence of tokens, interactive commands, or program instructions and breaks them up into parts that can be used by other components in programming.

The overall process of parsing involves three stages:

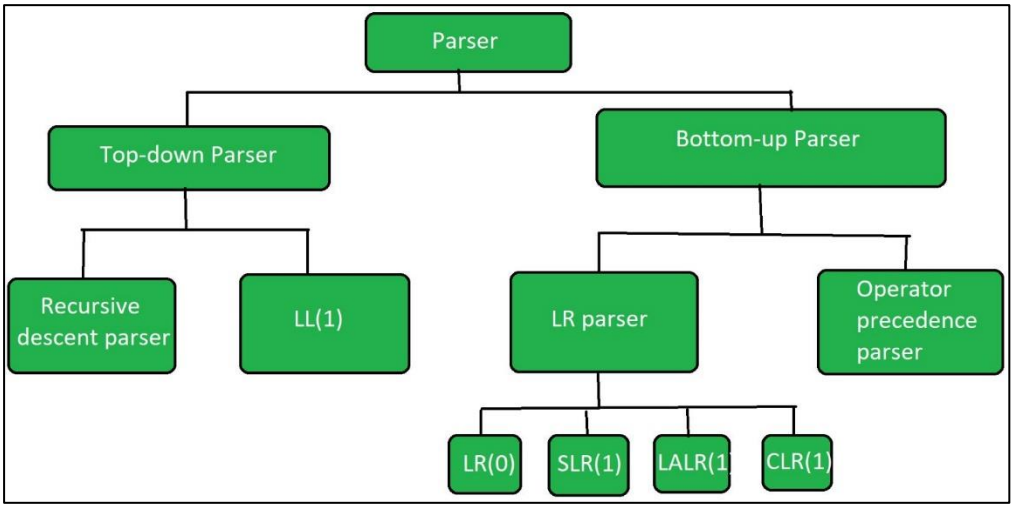
1. **Lexical Analysis:** A lexical analyzer is used to produce tokens from a stream of input string characters, which are broken into small components to form meaningful expressions. A token is the smallest unit in a programming language that possesses some meaning (such as +, -, *, “function”, or “new” in JavaScript).
2. **Syntactic Analysis:** Checks whether the generated tokens form a meaningful expression. This makes use of a context-free grammar that defines algorithmic procedures for components. These work to form an expression and define the particular order in which tokens must be placed.
3. **Semantic Parsing:** The final parsing stage in which the meaning and implications of the validated expression are determined and necessary actions are taken.

A parser's main purpose is to determine if input data may be derived from the start symbol of the grammar. This is achieved as follows:

- **Top-Down Parsing**: Involves searching a parse tree to find the left-most derivations of an input stream by using a top-down expansion. Parsing begins with the start symbol which is transformed into the input symbol until all symbols are translated and a parse

tree for an input string is constructed. Examples include LL parsers and recursive-descent parsers. Top-down parsing is also called predictive parsing or recursive parsing.

- **Bottom-Up Parsing:** Involves rewriting the input back to the start symbol. It acts in reverse by tracing out the rightmost derivation of a string until the parse tree is constructed up to the start symbol This type of parsing is also known as shift-reduce parsing. One example is an LR parser.



Code-

Lex Code:

```
%{
#include "y.tab.h"
%}
%%
[ /t/n]                 { ; }
"select"                 { return Select; }
"from"                   { return from; }
"distinct"               { return distinct; }
"where"                   { return where; }
"like"                  { return like; }
"and"                    { return and; }
```

```
"or"                    { return or; }
[0-9]+                  { return number; }
[A-Za-z]([A-Za-z]|[0-9])*       { return id; }
"<="                    { return le; }
">="                    { return ge; }
"=="                    { return eq; }
"!="                    { return ne; }
.                       { return yytext[0]; }

%%

int yywrap(void) { return 1; }
```

Yacc Code:

```
%{

void yyerror (char *s);
int yylex(void);
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

%}

%start start

%token Select from distinct where like and or number id le ge eq ne

%left and or
%left '<' '>' le ge eq ne

%right '='

%%

start   :  sql_a ';'            { printf("Valid SQL statement"); }
        |
        ;

sql_a   :  Select attributes from tables sql_b      { ; }
        | Select distinct attributes from tables sql_b      { ; }
        | Select distinct attributes from tables      { ; }
        | Select attributes from tables      { ; }
        ;

sql_b   :  where condition    { ; }
        ;

attributes :   id ',' attributes
```

```
            | '*'
            | id
            ;

tables    :    id ',' tables
            | id
            ;

condition  :   condition or condition
            | condition and condition
            | E
            ;

E     :       F '=' F
            | F '<' F
            | F '>' F
            | F le F
            | F ge F
            | F eq F
            | F ne F
            | F or F
            | F and F
            | F like F
            ;

F   :         id
            | number
            ;

%%

int main(void)
{
   printf("\n\n*****SQL Parser*****\n\n");
   return yyparse();
}

void yyerror (char *s) { fprintf (stderr, "%s\n", s); }
```

Output-