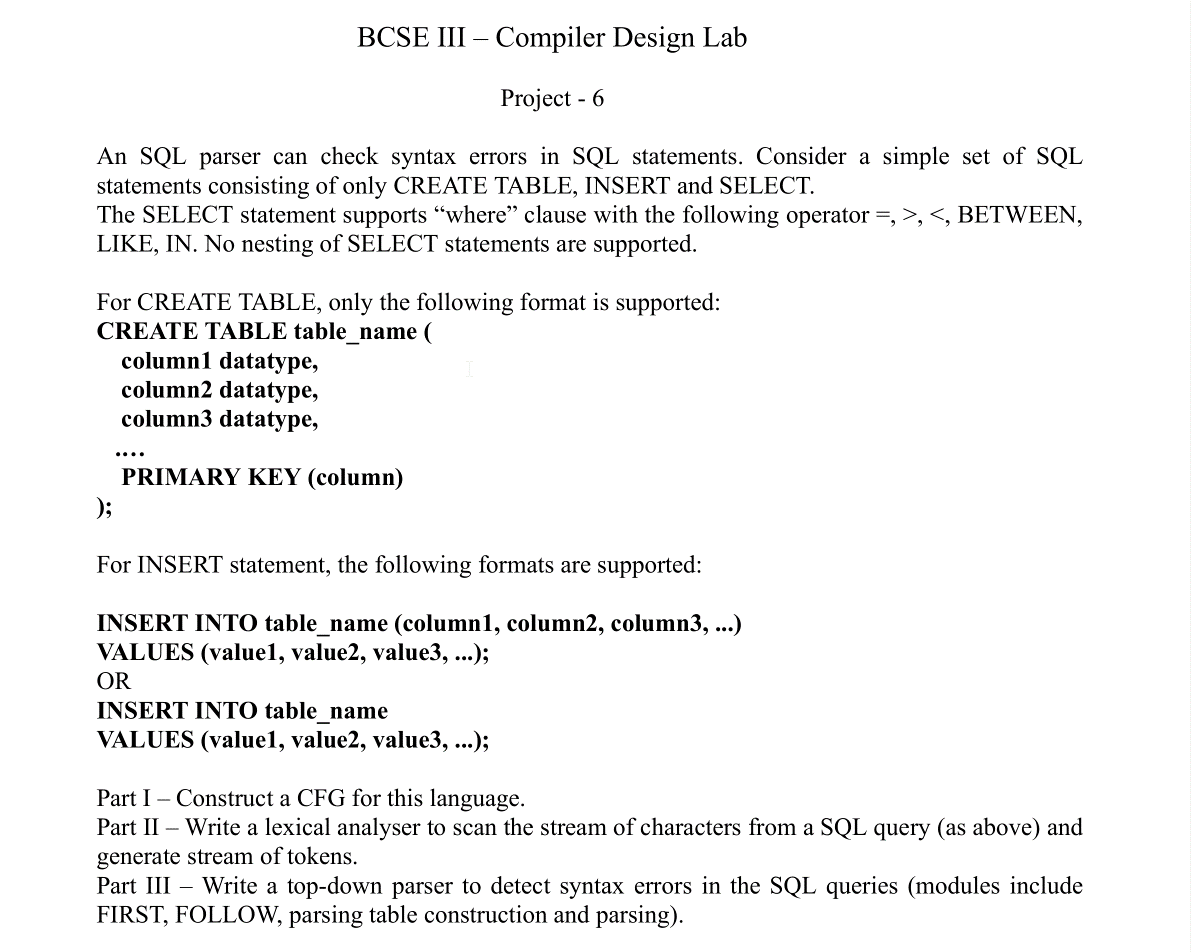
SQL Parser

Nikhil Badyal 001810501069

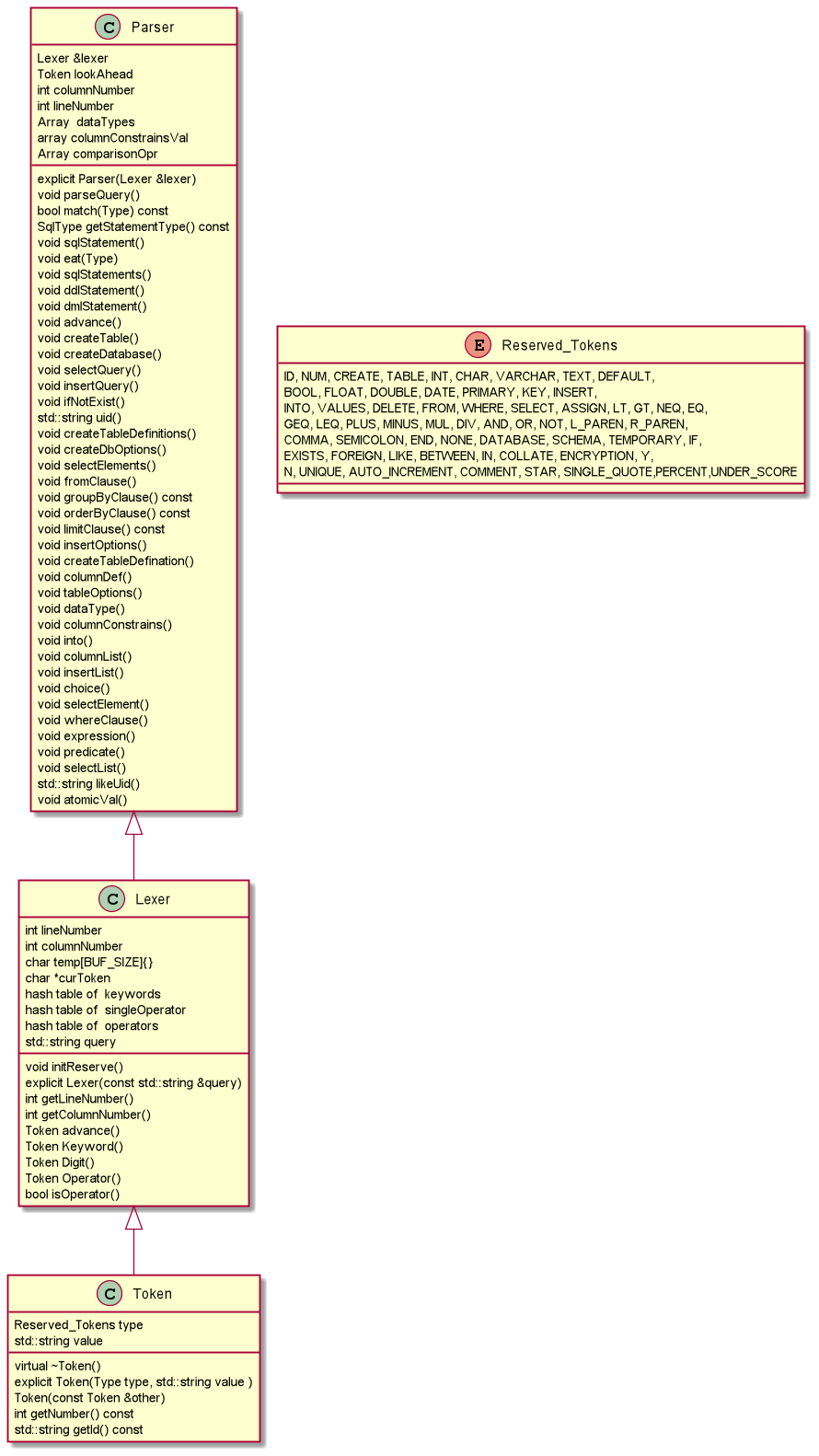
Sangita 001810501050

Kaushal 001810501051

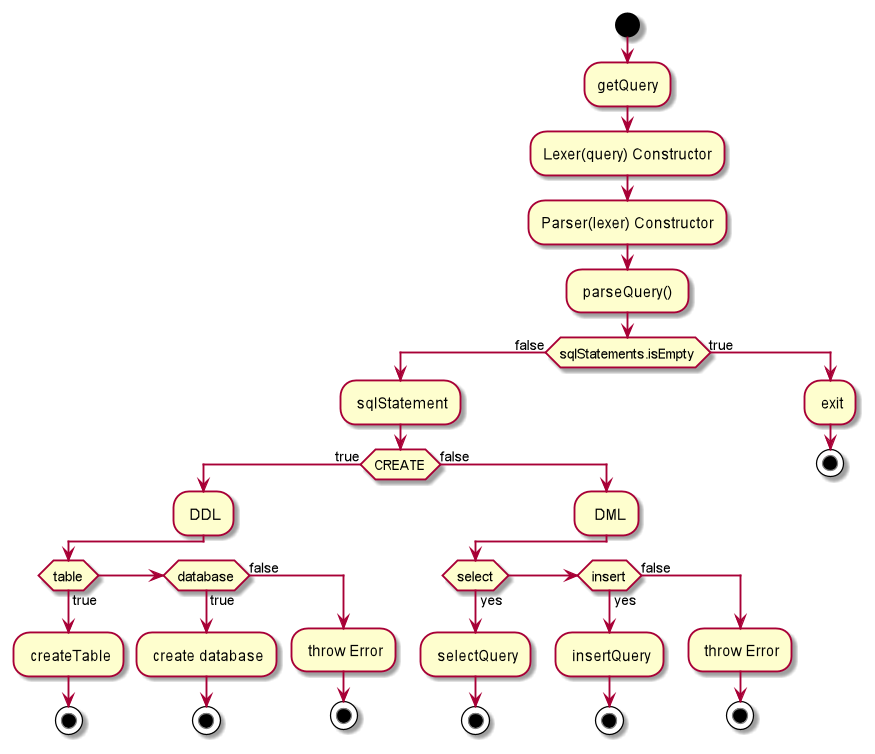
Aman Sharma 001810501068



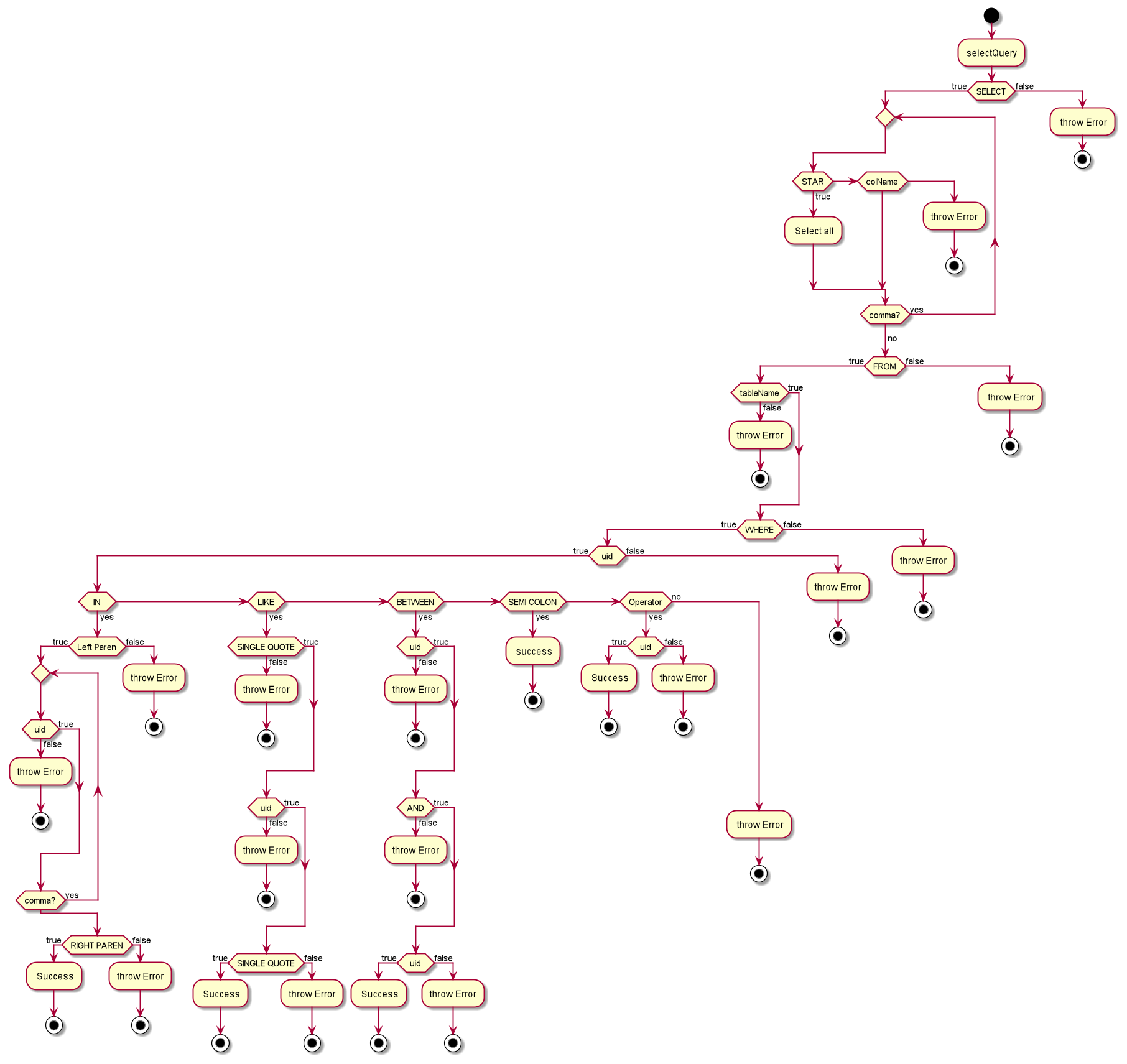
CLASS DIAGRAM



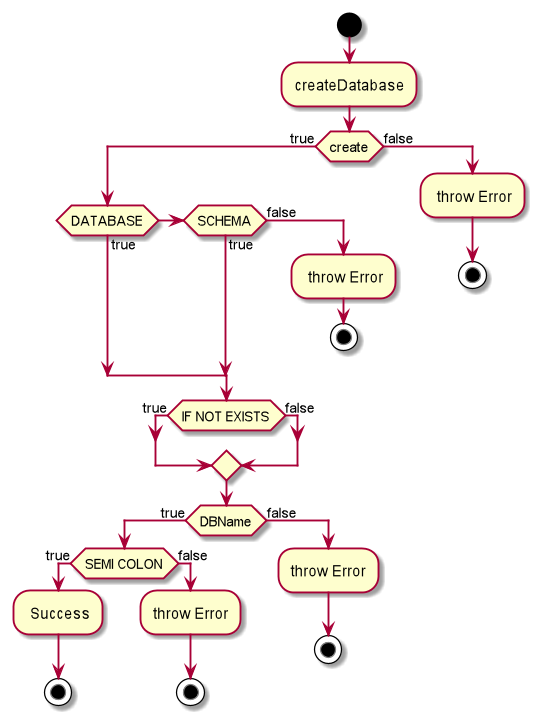
**WORK FLOW**



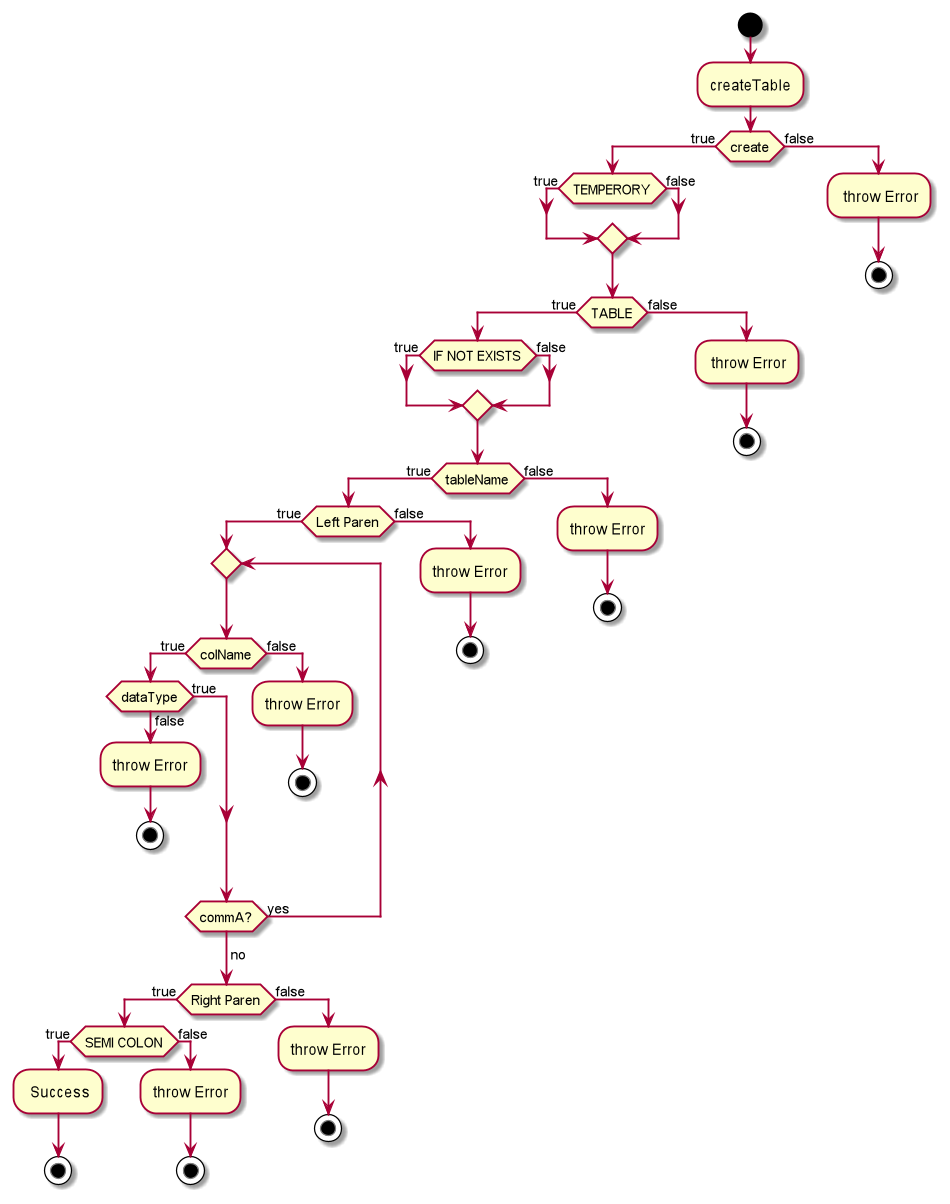
**WORKING OF SELECT CLAUSE**



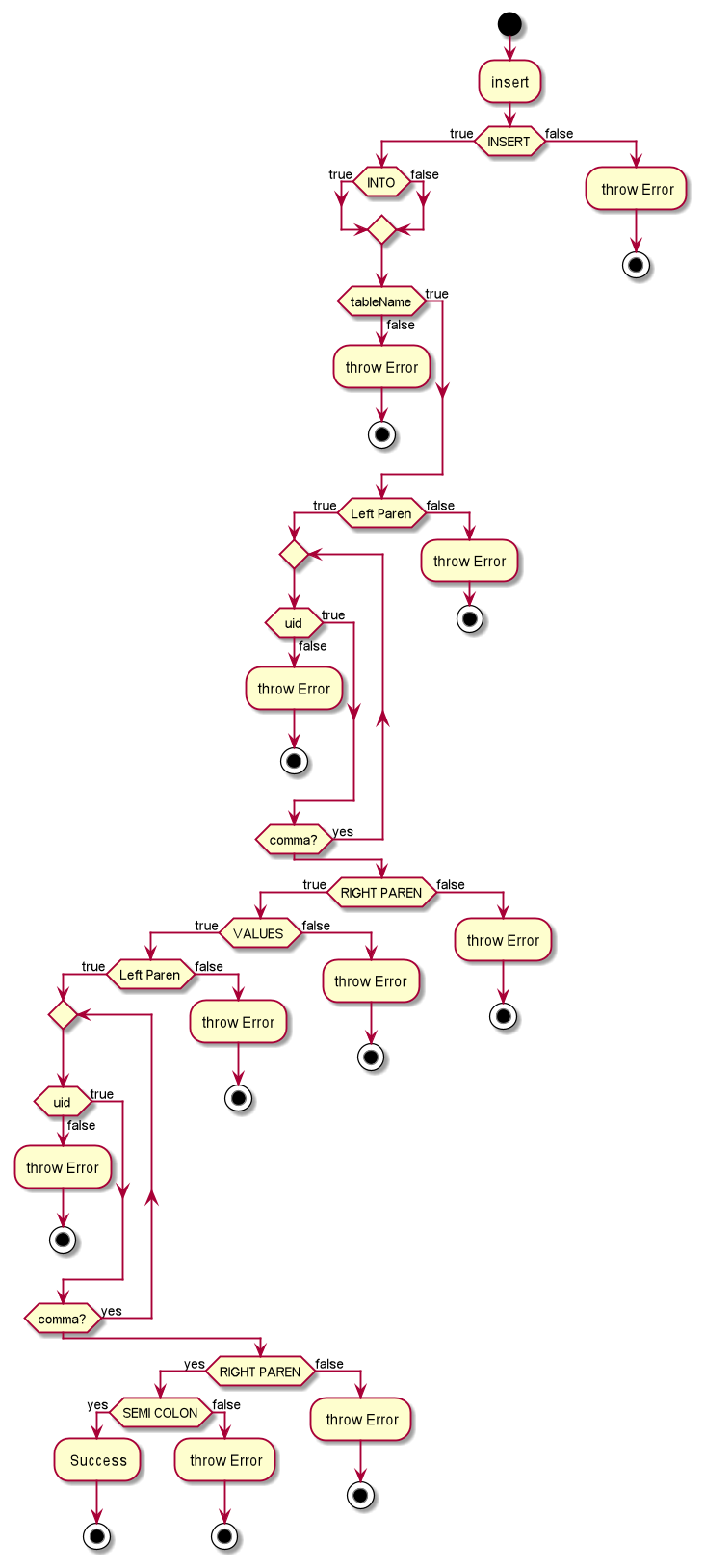
**Working of create DATABASE**



**WOKRING OF CREATE TABLE**



**WORKING OF INSERT CLAUSE**



MODULE 3

To convert the grammar into actual code. Following pattern was used

1. Every production rule corresponds to a new and separate function call.
2. Multiple choices in a production rule are depicted by if else/ switch clause.
3. Star symbol (Regular Expression) or self-production (Recursion) are depicted with a while loop.

parser.parseQuery();

*void* Parser::parseQuery() {  
 sqlStatements();  
}

It starts the parsing process.

sqlStatements();

Is just a dummy function. Which we have used just to tell that we made a SQL parser. It also checks if there is anything user has entered.

*void* Parser::sqlStatements() {  
 *if* (lexer.query.empty() || lexer.query.size() == 1) {  
 *throw* ParserException("Enter Something");  
 } *else* {  
 sqlStatement();

… Some Error handling

sqlStatement()

This starts the actual work and check if the statement is DDL or DML

*void* Parser::sqlStatement() {  
 *auto* statementType = getStatementType();  
 *if* (statementType == DDL) {  
 ddlStatement();  
 } *else* {  
 dmlStatement();  
 }  
}

getStatementType();

This a helper function which uses follow set to identify the type of query

SqlType Parser::getStatementType() *const* {  
 *if* (lookAhead.type == CREATE) {  
 *return* DDL;  
 } *else if* (lookAhead.type == INSERT || lookAhead.type == SELECT) {  
 *return* DML;  
 } *else* {  
 *throw* ParserException(  
 "Sql statement must be DDL or DML statements Error in line no " + std::to\_string(lineNumber) +  
 " and column number " + std::to\_string(columnNumber));  
 }  
}

ddlStatement();

This function parses DDL statements. It detects the type of create statement whether it is CREATE DATABASE OR CREATE TABLE

*if* (lookAhead.type == DATABASE || lookAhead.type == SCHEMA) {  
 createDatabase();  
} *else if* (lookAhead.type == TABLE || lookAhead.type == TEMPORARY) {  
 *if* (lookAhead.type == TEMPORARY) {  
 eat(TEMPORARY);  
 }  
 createTable();  
} *else* {  
 *throw* ParserException()

….. Some code

dmlStatement()

This function parses DDL statements. It detects the type of create statement whether it is SELECT OR INSERT

*if* (match(SELECT)) {  
 selectQuery();  
} *else if* (match(INSERT)) {  
 insertQuery();  
} *else* {  
 *throw* ParserException()

..SOME Code

selectQuery()

This function handles all select commands. Since select commands start with SELECT it check this first.

*if* (match(SELECT)) {  
 eat(SELECT);  
 selectElements();  
 fromClause();  
 groupByClause();  
 orderByClause();  
 limitClause();  
 match(SEMICOLON);  
} *else* {  
 *throw* ParserException("Syntax Error

… Some Error handlers

insertQuery()

This function handles all insert commands. Since select commands start with INSERT it check this first.

*if* (match(INSERT)) {  
 eat(INSERT);  
 into();  
 std::string tableName = uid();  
 insertOptions();  
} *else* {  
 *throw* ParserException("Syntax Er…….

Some Error handling.

eat(Type type)

This function consumes the current token and ask lexer to give him (Parser) the next token. Just to be safe it first checks whether the expected and current token in the actual input are same or not.

Then make a call to advance function, which further calls the lexer’s advance function , which ultimate give a new token back.

It is a Parser’s function

*void* Parser::eat(Type type) {  
 *if* (match(type)) {  
 advance();  
 } *else* {  
 *throw* ParserException("Syntax Error in line no " + std::to\_string(lineNumber) + " and column number " +  
 std::to\_string(columnNumber));  
  
 }  
  
}

match(Type type)

Match is simple function , it Just return a simple TRUE or FALSE. It help us the parser in knowing whether the token which parser is expecting and the token which is actual present in the lookahead are same or not.

*bool* Parser::match(Type expectingType) *const* {  
 *return* lookAhead.type == type;  
}

MODULE 1

**LEXER**

Lexer::advance()

This is the main function in lexer which actually reads the input and make a token out of it and give it back to parser.

Token Lexer::advance() {  
 *while* (isspace(\*curToken)) { *//Space* columnNumber++;  
 curToken++;  
 }  
 *if* (isalpha(\*curToken) || \*curToken == '\_') { *// Keyword or ID  
 return* Keyword();  
 } *else if* (singleOperator.find(\*curToken) != singleOperator.end()) {  
 *return* Token(singleOperator[\*curToken++]);  
 }*else if* (isdigit(\*curToken)) { *//Number  
 return* Digit();  
 } *else if* (isOperator()) { *//Operator  
 return* Operator();  
 } *else* {  
 *throw* LexerException("Invalid input query");  
 }  
}

Parsers advanced method uses this function like this

*void* Parser::advance() {  
 lookAhead = lexer.advance();  
 columnNumber = lexer.getColumnNumber();  
 lineNumber = lexer.getLineNumber();  
}

MODULE 2

(CFG)

ParseQuery

: SQLStatements

| €

;

SQLStatements

: sqlStatement SQLStatements // Recursive call to add support for multiple queries

| €

;

sqlStatement

: ddlStatement // Handles all ddl statement(CREATE)

| dmlStatement // Handles all dml statement(SELET INSERT)

;

ddlStatement

: createDatabase // Handles all create database

| createTable // Handles all create table

;

dmlStatement

: select // Handles all select query

| insert // Handles all insert query

;

createDatabase

: CREATE dbFormat ifNotExist uid SEMI\_COLON

;

dbFormat // Type of db (MYSQL 8.0 syntax)

: DATABASE

| SCHEMA

;

ifNotExist // OPTIONAL CLAUSE

: IF NOT EXISTS

| €

;

createTable // handles create table query

: CREATE temporary TABLE ifNotExist uid createDefinations SEMI\_COLON

;

temporary // OPTIONAL CLAUSE

: TEMPERORY

| €

;

createDefinations // Can consume input like (userid int , salary int , username string)

: (createDefination createDefinations)

| , createDefination createDefinations // Recusrive call , because there can be multiple

| €

;

createDefination // Can consume -🡪 userid string or salary int or PRIMARY KEY(userid)

: uid datatypes

| tableOptions

;

Datatype // Supported data types

: INT

| CHAR

| VARCHAR

| TEXT

| BOOL

;

tableOptions: // Supported table options

: PRIMARY KEY list

| FOREIGN KEY list

;

select

: selectQuery

;

selectQuery // Handles select query

: SELECT selectElements fromClause limitClause SEMI\_COLON

;

selectElements // Can consume select **\*** from or select **userid, salary**

: \*

| list

| €

;

fromClause // Hanldes from clause

: FROM uid whereClause

;

whereClause // Handles where clause and OPTIONAL CLAUSE

: WHERE expression

| €

;

limitClause // hanldes limit clause

: LIMIT uid

| €

;

Insert // Handles insert clause

: INSERT into uid insertOpt SEMI\_COLON

;

into // Optinal into as mysql

: INTO

| €

;

insertOpt // Can consume insert into table\_name **values (userid, salary)** or insert into table\_name **(userid, salary)**

: list insertOpt

| VALUES list

| €

;

list:

: (uid list)

| uid list

| , uid list

| €

;

expression // To represent expression in where clause

: predicate

;

predicate

: predicate IN list // IN

| predicate BETWEEEN predicate AND predicate // Can consume salary BETWEEN 100 AND 2000

| predicate LIKE predicate // like

| predicate comparisonOpr predicate // > 30

| atomicVal

;

atomicVal

: uid

;

Uid // handles all string or number or strings which begun with single quotes

: [a-zA-Z] uid;

| \_ uid

| SINGLE\_QUOTE uid SINGLE\_QUOTE

| €

;

comparisonOpr

: =

| <

| >

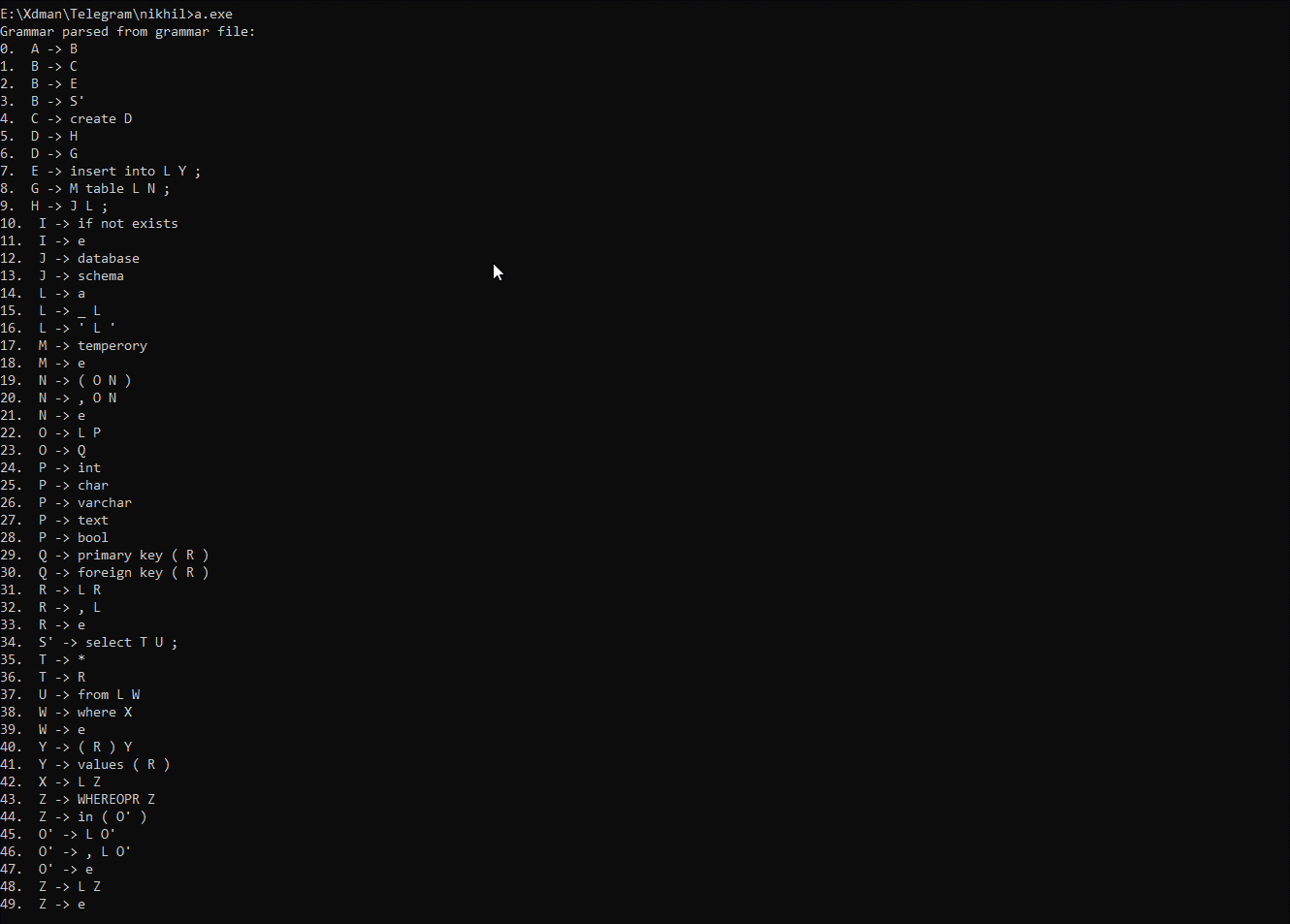
| >=

| <=

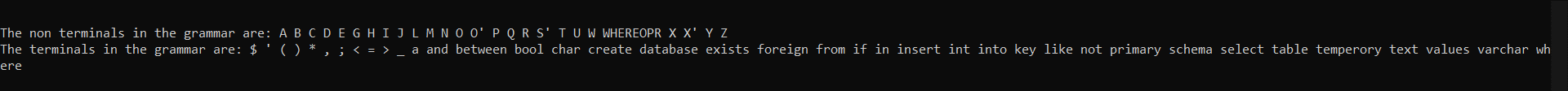
**LL1 Grammar**

A ::= B  
B ::= C  
B ::= E  
B ::= S'  
C ::= create D  
D ::= H  
D ::= G  
E ::= insert into L Y ;  
G ::= M table L N ;  
H ::= J L ;  
I ::= if not exists  
I ::= ''  
J ::= database  
J ::= schema  
L ::= a  
L ::= \_ L  
L ::= ' L '  
M ::= temperory  
M ::= ''  
N ::= ( O N )  
N ::= , O N  
N ::= ''  
O ::= L P  
O ::= Q  
P ::= int  
P ::= char  
P ::= varchar  
P ::= text  
P ::= bool  
Q ::= primary key ( R )  
Q ::= foreign key ( R )  
R ::= L R  
R ::= , L  
R ::= ''  
S' ::= select T U ;  
T ::= \*  
T ::= R  
U ::= from L W  
W ::= where X  
W ::= ''  
Y ::= ( R ) Y  
Y ::= values ( R )  
X ::= L Z  
Z ::= WHEREOPR Z  
Z ::= in ( O' )  
O' ::= L O'  
O' ::= , L O'  
O' ::= ''  
Z ::= L Z  
Z ::= ''  
WHEREOPR ::= between  
WHEREOPR ::= like  
WHEREOPR ::= and  
WHEREOPR ::= >  
WHEREOPR ::= <  
WHEREOPR ::= =  
X' ::= =  
X' ::= <  
X' ::= >

Grammar



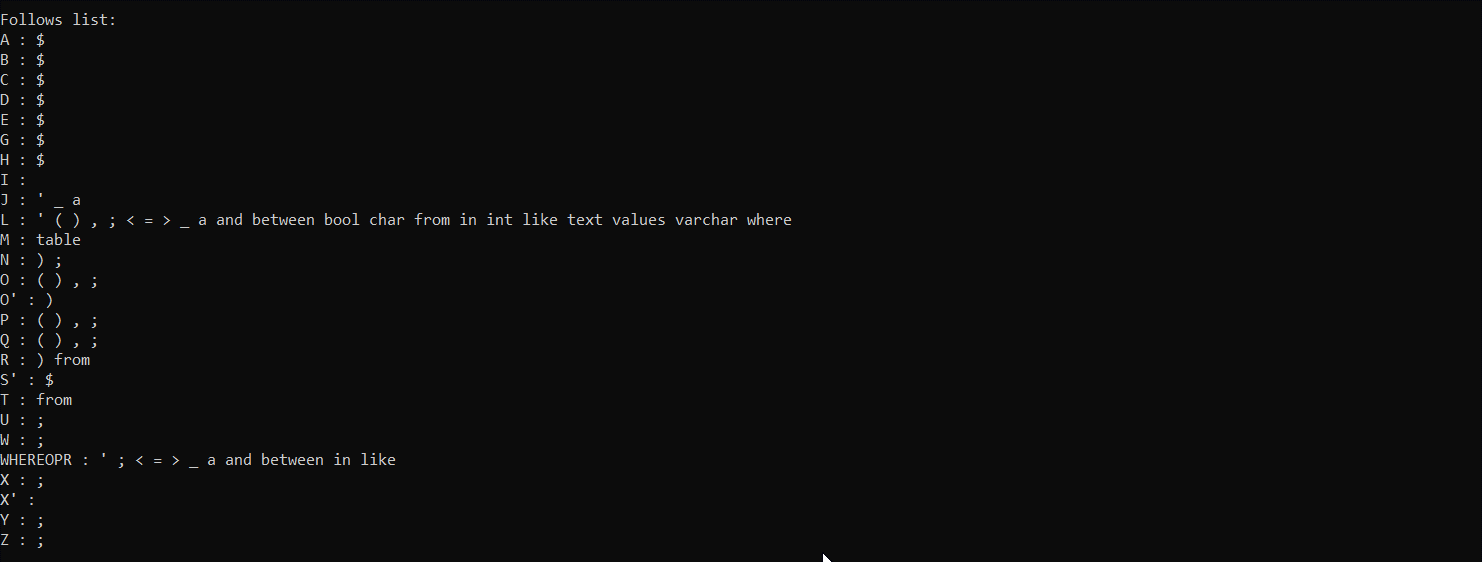
**Terminal / NT**



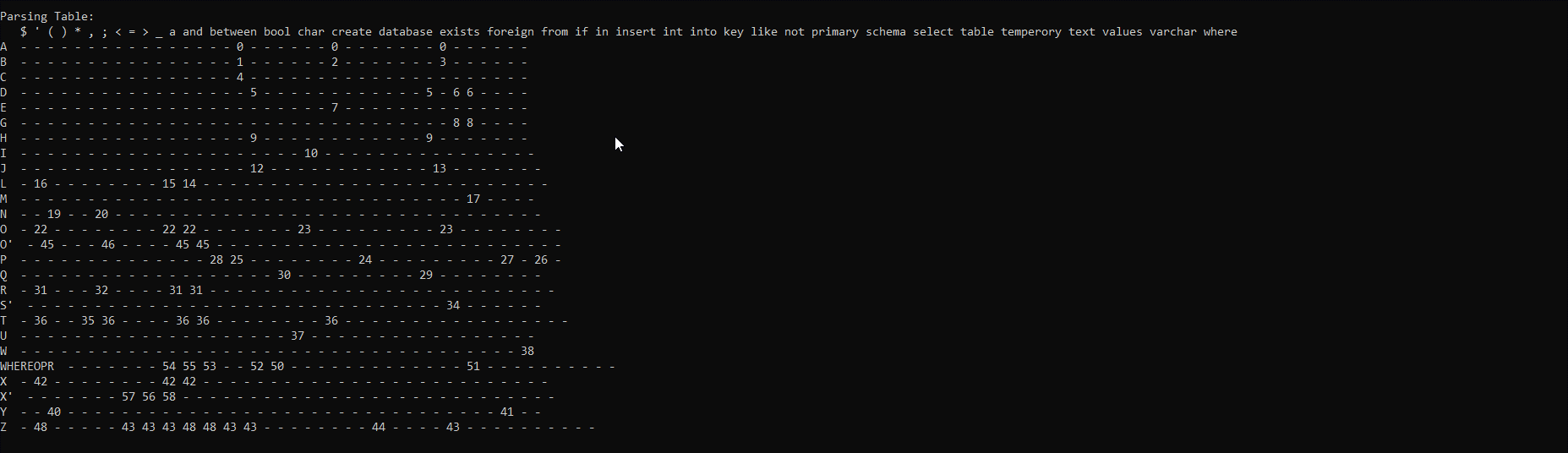
**First List**

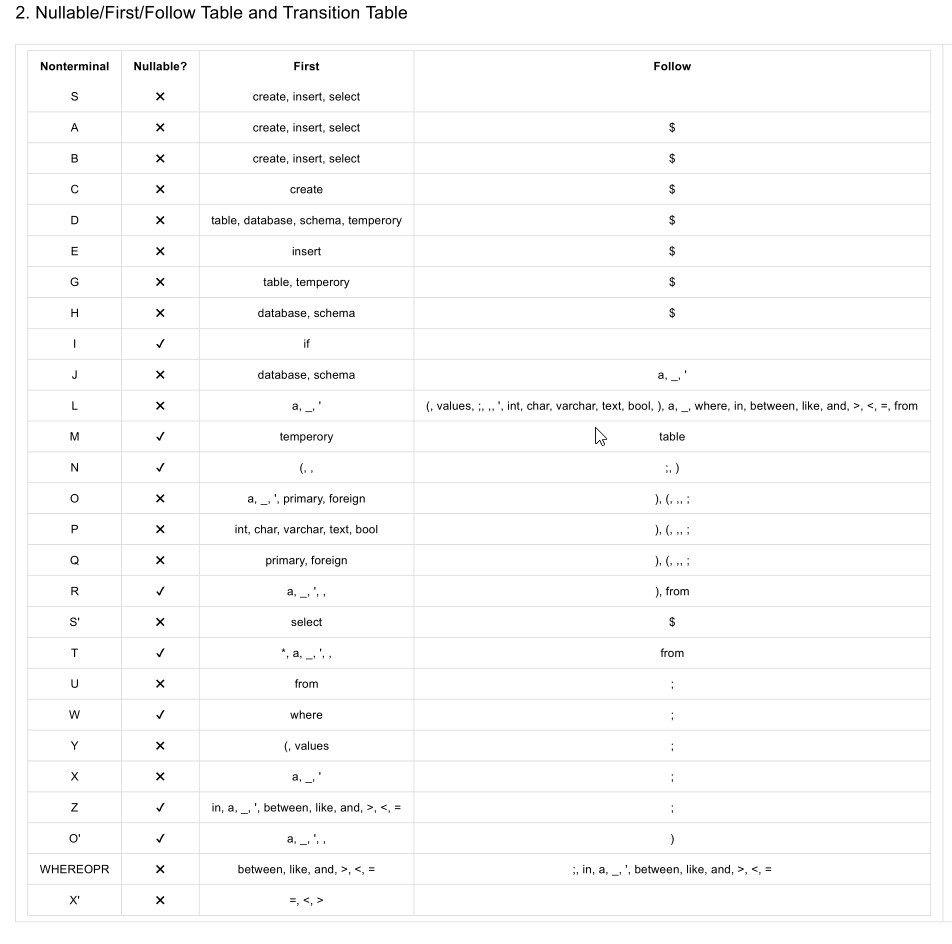


**Follow**

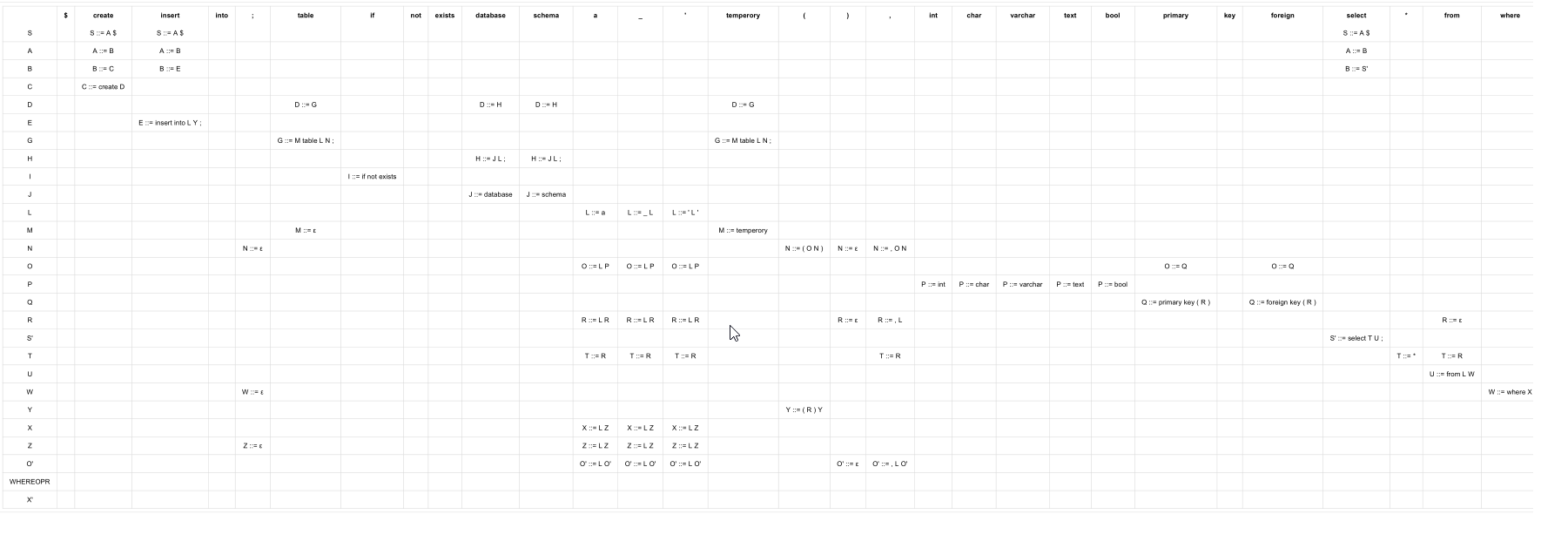


**Parsing Table**

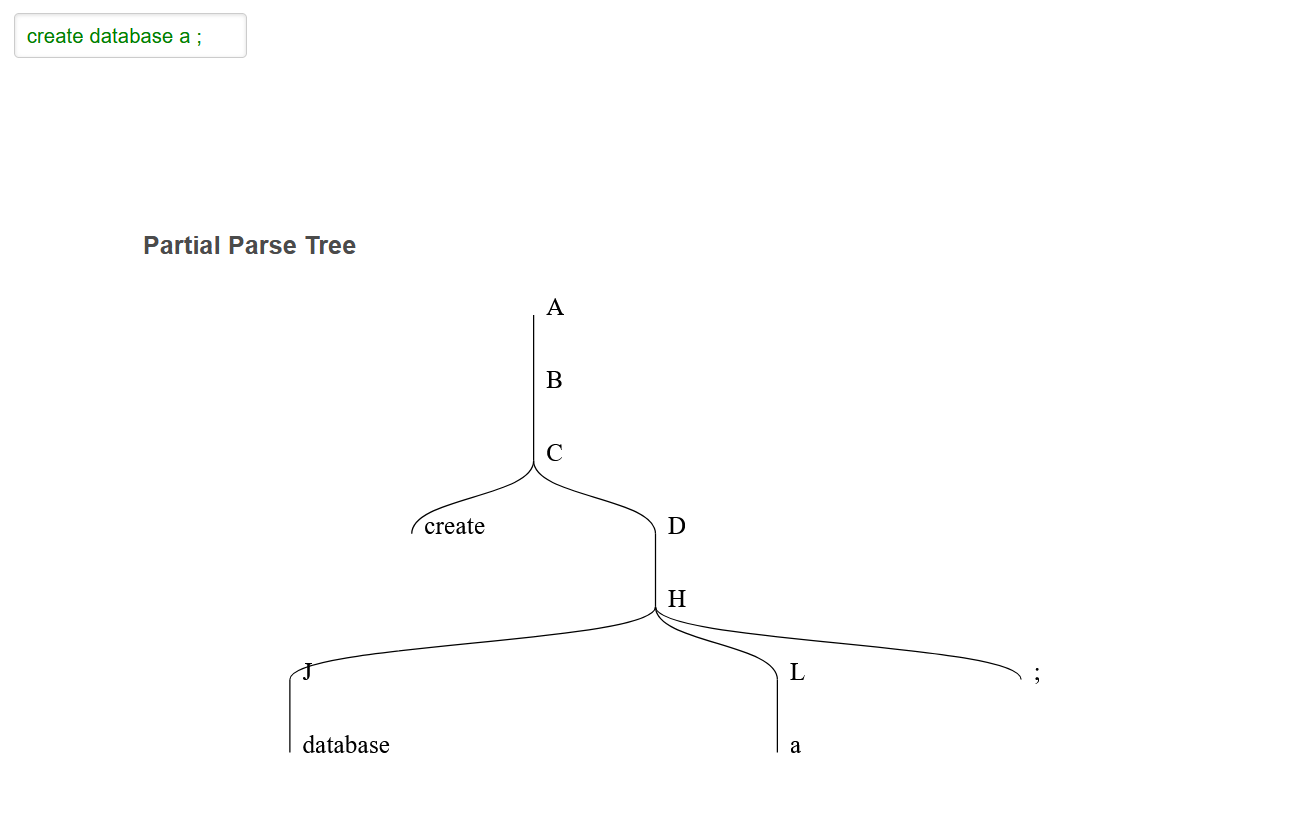


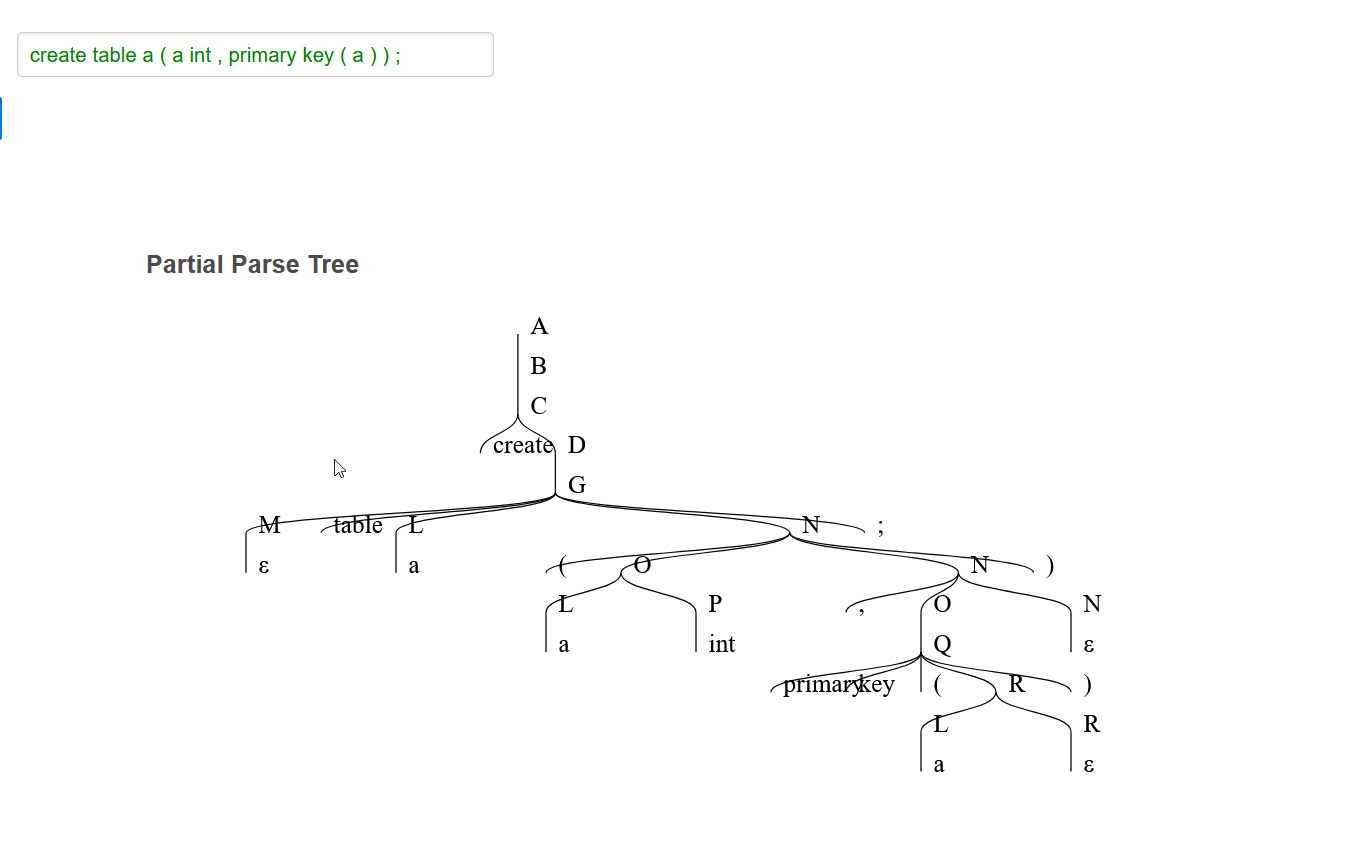


**LL1 table**

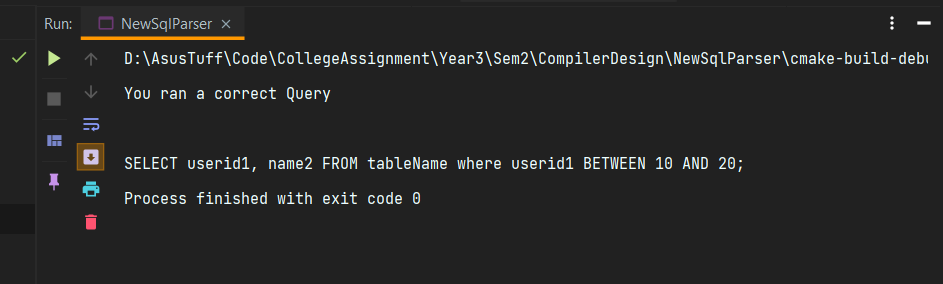


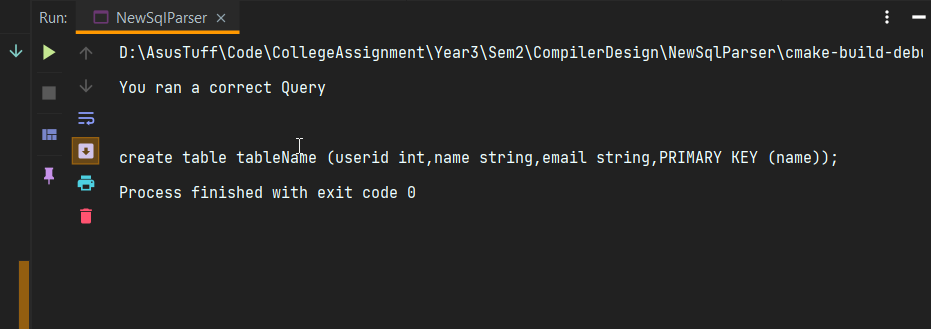
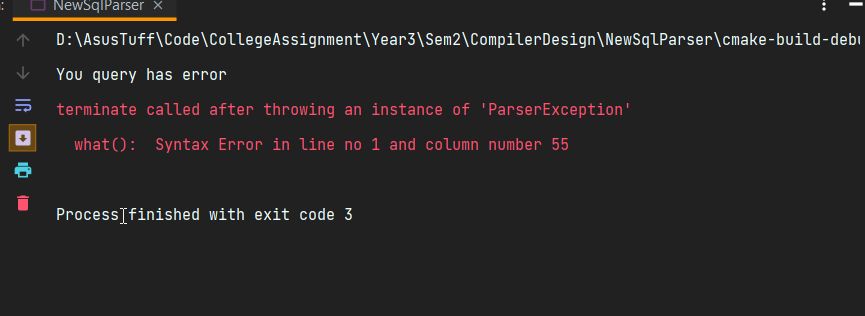
Some examples

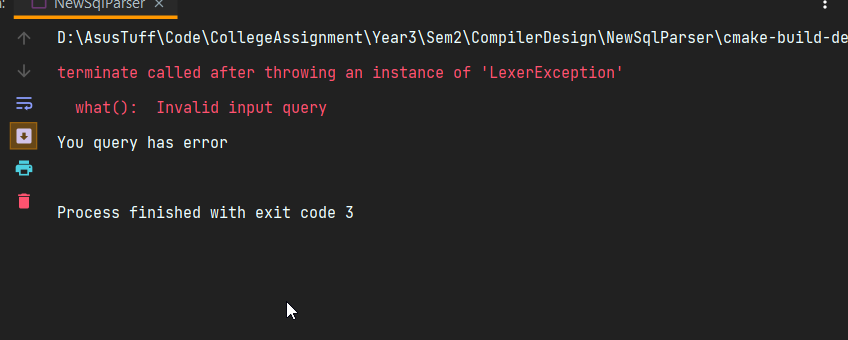


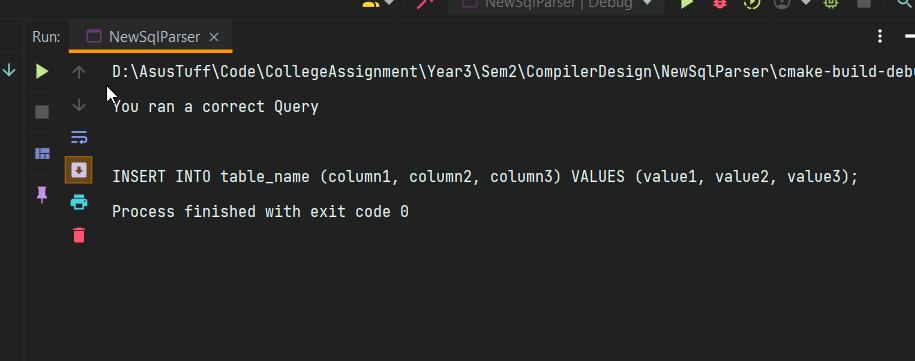


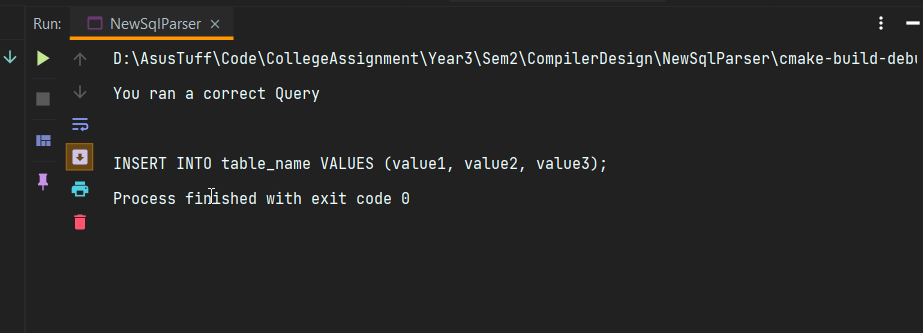
**SCREEN SHOTS**

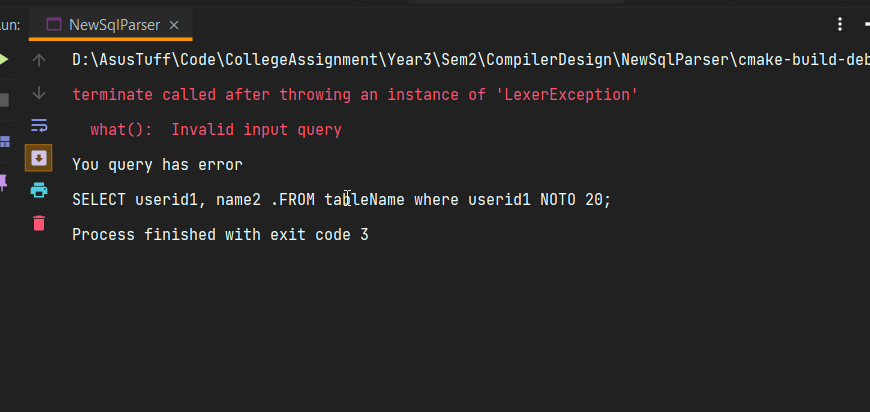


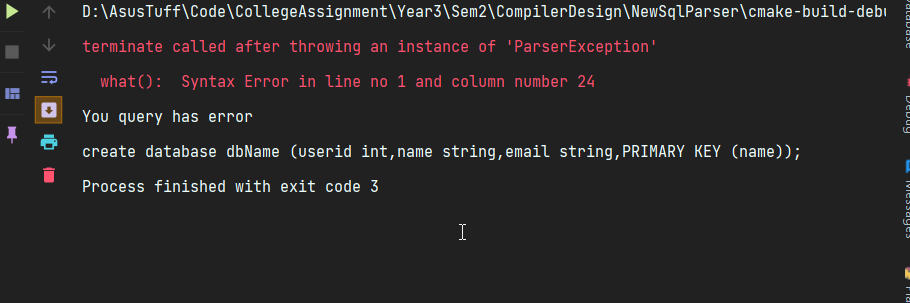












**CODE**

**Main.cpp**

#include <iostream>  
#include "parser/Parser.h"  
  
*int* main() {  
  
*// std::string query = "create database cp\_project ;";  
// std::string query = "create database cp\_project";  
// std::string query = "create schema cp\_project;";  
// std::string query = "create database cp\_project;";  
// std::string query = "create database IF NOT EXISTS cp\_project;";  
// std::string query = "create schema IF NOT EXISTS cp\_project;";  
// std::string query = "create schema cp\_project ;";* std::string query = "create table a ("  
 "a int,"  
 "PRIMARY KEY ( a )"  
 ");";  
  
 */\* std::string query = "SELECT userid1, name2 ."  
 "FROM tableName where userid1 NOTO 20;";\*/  
 /\* std::string query = "SELECT userid1, name2 "  
 "FROM tableName where userid1 LIKE 10 AND 20;";\*/  
 /\* std::string query = "SELECT userid1, name2 "  
 "FROM tableName where userid1 BETWEEN 10 AND 20;";\*/  
 /\* std::string query = "SELECT userid1, name2 "  
 "FROM tableName where userid1 BETWEEN 'Nikhil' AND 'Aikhil';";\*/  
 /\* std::string query = "SELECT userid1, name2 "  
 "FROM tableName where userid1 BETWEENO 'Nikhil' AND 'Aikhil';";\*/  
 /\* std::string query = "SELECT userid1, name2 "  
 "FROM tableName where userid1 "  
 "AND 23;";\*/  
 /\* std::string query = "SELECT userid1, name2 "  
 "FROM tableName where userid1 "  
 "AND 23;";\*/  
 /\* std::string query = "SELECT userid1, name2 "  
 "FROM tableName where userid1 > 23;";\*/  
 /\* std::string query = "SELECT userid1, name2 "  
 "FROM tableName where userid1 >= 23;";\*/  
 /\*std::string query = "SELECT userid1, name2 "  
 "FROM tableName where userid1 <= 23;";\*/  
 /\* std::string query = "SELECT \* from table\_name;";\*/  
 /\*std::string query = "SELECT userid1, name2 "  
 "FROM tableName where userid1 NOT 23;";\*/  
 /\* std::string query = "SELECT userid1, name2 "  
 "FROM tableName where userid1 <= 23;";\*/  
 /\* std::string query = "SELECT userid1, name2,"  
 "FROM tableName where userid1 between 23 and 24;";\*/  
 /\* std::string query = "SELECT userid1, name2 "  
 "FROM tableName where userid1 in ('1','2','3');";\*/  
 /\* std::string query = "SELECT userid1, name2 "  
 "FROM tableName where userid1 LIKE '%\_\_nikhil%%\_\_\_';";\*/  
  
 /\* std::string query = "INSERT INTO table\_name (column1, column2, column3) "  
 "VALUES (value1, value2, value3);";\*/  
  
 /\* std::string query = "INSERT INTO table\_name "  
 "VALUES (value1, value2, value3);";\*/* std::cout<<"You query has error \n"+ query;  
 *auto* lexer = Lexer(query);  
 Parser parser = Parser(lexer);  
 parser.parseQuery();  
*// std::cout << "You ran a correct Query\n\n"+query;  
  
 return* 0;  
}

Token.h

#ifndef **NEWSQLPARSER\_TOKEN\_H**#define **NEWSQLPARSER\_TOKEN\_H**#include <string>  
#include <utility>  
#include <vector>  
  
*enum* Type {  
 ID, NUM, CREATE, TABLE, INT, CHAR, VARCHAR, TEXT, DEFAULT,  
 BOOL, FLOAT, DOUBLE, DATE, PRIMARY, KEY, INSERT,  
 INTO, VALUES, DELETE, FROM, WHERE, SELECT, ASSIGN, LT, GT, NEQ, EQ,  
 GEQ, LEQ, PLUS, MINUS, MUL, DIV, AND, OR, NOT, L\_PAREN, R\_PAREN,  
 COMMA, SEMICOLON, END, NONE, DATABASE, SCHEMA, TEMPORARY, IF,  
 EXISTS, FOREIGN, LIKE, BETWEEN, IN, COLLATE, ENCRYPTION, Y,  
 N, UNIQUE, AUTO\_INCREMENT, COMMENT, STAR, SINGLE\_QUOTE,PERCENT,UNDER\_SCORE  
};  
  
  
*class* Token {  
*public*:  
 *virtual* ~Token();  
  
 *explicit* Token(Type type, std::string value = "") : type(type), value(std::move(value)) {}  
  
 Token(*const* Token &other);  
  
 Type type;  
 std::string value;  
  
 *int* getNumber() *const* ;  
  
 std::string getId() *const* ;  
  
  
};  
  
#endif *//NEWSQLPARSER\_TOKEN\_H*

Token.cpp

#include "Token.h"  
#include <string>  
  
Token::Token(*const* Token &other) {  
 type = other.type;  
 value = other.value;  
}  
  
Token::~Token() {  
  
}  
  
*int* Token::getNumber() *const* {  
 *return* 0;  
}  
  
std::string Token::getId() *const* {  
 *return nullptr*;  
}

Lexer.h

#pragma clang diagnostic push  
#pragma ide diagnostic ignored "modernize-use-nodiscard"  
  
  
#ifndef **NEWSQLPARSER\_LEXER\_H**#define **NEWSQLPARSER\_LEXER\_H**#include <string>  
#include <utility>  
#include <map>  
#include "Token.h"  
  
*constexpr int* BUF\_SIZE = 256;  
  
*class* Lexer {  
 *int* lineNumber = 1; *// To tell the number line while error  
 int* columnNumber = 1; *//To tell the column number while error  
 char* temp[BUF\_SIZE]{}; *// To analyze token/ID/Identifier  
 char* \*curToken;  
 std::map<std::string, Type> keywords;  
 std::map<*char*, Type> singleOperator;  
 std::map<std::string, Type> operators;  
  
  
*public*:  
 std::string query;  
  
 *void* initReserve();  
  
 *explicit* Lexer(*const* std::string &query) : curToken(*const\_cast*<*char* \*>( query.c\_str())), query(query) {  
 initReserve();  
 }  
  
 *int* getLineNumber() *const* {  
 *return* lineNumber;  
 }  
  
 *int* getColumnNumber() *const* {  
 *return* columnNumber;  
 }  
  
 Token advance();  
  
  
 Token Keyword();  
  
 Token Digit();  
  
 Token Operator();  
  
 *bool* isOperator() *const*;  
  
};  
  
#endif *//NEWSQLPARSER\_LEXER\_H*

Lexer.cpp

#include <stdexcept>  
#include <algorithm>  
#include <iostream>  
#include "Lexer.h"  
  
  
*class* LexerException : *public* std::exception {  
 std::string errorMsg;  
*public*:  
 *explicit* LexerException(std::string errorMsg) : errorMsg(std::move(errorMsg)) {}  
  
 [[nodiscard]] *const char* \*what() *const* **\_GLIBCXX\_TXN\_SAFE\_DYN \_GLIBCXX\_NOTHROW** *override* {  
 *return* errorMsg.c\_str();  
 }  
};  
  
Token Lexer::advance() {  
 *while* (isspace(\*curToken)) { *//Space* columnNumber++;  
 curToken++;  
 }  
 *if* (isalpha(\*curToken) || \*curToken == '\_') { *// Keyword or ID  
 return* Keyword();  
 } *else if* (singleOperator.find(\*curToken) != singleOperator.end()) {  
 *return* Token(singleOperator[\*curToken++]);  
 }*else if* (isdigit(\*curToken)) { *//Number  
 return* Digit();  
 } *else if* (isOperator()) { *//Operator  
 return* Operator();  
 } *else* {  
 *throw* LexerException("Invalid input query");  
 }  
}  
  
  
Token Lexer::Keyword() {  
 *int* i;  
 *for* (i = 0; isalnum(\*curToken) || \*curToken == '\_'; ++curToken) {  
 columnNumber++;  
 temp[i++] = \*curToken;  
 }  
 temp[i] = '\0';  
 std::string s = std::string(temp);  
 std::transform(s.begin(), s.end(), s.begin(), ::tolower);  
 *if* (*auto* itr = keywords.find(s);itr != keywords.end()) {  
 *return* Token(itr->second, itr->first);  
 } *else* {  
 *return* Token(ID, s);  
 }  
}  
  
Token Lexer::Digit() {  
 *int* i;  
 *int* flag = 1;  
 *for* (i = 0; isdigit(\*curToken) || \*curToken == '.'; ++curToken) {  
 columnNumber++;  
  
 *if* (\*curToken == '.') {  
 *if* (flag) {  
 flag = 0;  
 } *else* {  
 *break*;  
 *//TODO Error digit has 2 decimals* }  
 }  
 temp[i++] = \*curToken;  
  
 }  
 temp[i] = '\0';  
 *return* Token(NUM, temp);  
}  
  
Token Lexer::Operator() {  
 *int* i = 0;  
 *do* {  
 columnNumber++;  
  
 temp[i++] = \*curToken;  
 curToken++;  
 } *while* (isOperator() && i != 2);  
 std::string str(temp);  
 *return* Token(operators[str]);  
}  
  
*void* Lexer::initReserve() {  
 keywords["num"]= NUM;  
 keywords["create"] = CREATE;  
 keywords["table"] = TABLE;  
 keywords["default"] = DEFAULT;  
 keywords["primary"] = PRIMARY;  
 keywords["key"] = KEY;  
 keywords["insert"] = INSERT;  
 keywords["into"] = INTO;  
 keywords["values"] = VALUES;  
 keywords["select"] = SELECT;  
 keywords["assign"] = ASSIGN;  
 keywords["lt"] = LT;  
 keywords["gt"] = GT;  
 keywords["neq"] = NEQ;  
 keywords["eq"] = EQ;  
 keywords["geq"] = GEQ;  
 keywords["leq"] = LEQ;  
 keywords["plus"] = PLUS;  
 keywords["minus"] = MINUS;  
 keywords["mul"] = MUL;  
 keywords["div"] = DIV;  
 keywords["and"] = AND;  
 keywords["or"] = OR;  
 keywords["not"] = NOT;  
 keywords["l\_paren"] = L\_PAREN;  
 keywords["r\_paren"] =R\_PAREN;  
 keywords["comma"] = COMMA;  
 keywords["semicolon"] = SEMICOLON;  
 keywords["database"] = DATABASE;  
 keywords["schema"] = SCHEMA;  
 keywords["temporary"] = TEMPORARY;  
 keywords["if"] = IF;  
 keywords["exists"] = EXISTS;  
 keywords["foreign"] = FOREIGN;  
 keywords["collate"] = COLLATE;  
 keywords["encryption"] = ENCRYPTION;  
 keywords["y"] = Y;  
 keywords["n"] = N;  
 keywords["unique"] = UNIQUE;  
 keywords["auto\_increment"] = AUTO\_INCREMENT;  
 keywords["comment"] = COMMENT;  
 keywords["star"] = STAR;  
 keywords["from"] = FROM;  
 keywords["where"] = WHERE;  
 keywords["between"] = BETWEEN;  
 keywords["in"] = IN;  
 keywords["like"] = LIKE;  
  
 */\* keywords["table"] = TABLE;  
 keywords["table"] = TABLE;  
 keywords["table"] = TABLE;  
 keywords["table"] = TABLE;  
 keywords["table"] = TABLE;  
 keywords["table"] = TABLE;\*/* singleOperator['+'] = PLUS;  
 singleOperator['-'] = MINUS;  
 singleOperator['\*'] = MUL;  
 singleOperator['/'] = DIV;  
 singleOperator['('] = L\_PAREN;  
 singleOperator[')'] = R\_PAREN;  
 singleOperator[','] = COMMA;  
 singleOperator[';'] = SEMICOLON;  
 singleOperator['='] = EQ;  
 singleOperator['<'] = LT;  
 singleOperator['>'] = GT;  
 singleOperator['\''] = SINGLE\_QUOTE;  
 singleOperator['%'] = PERCENT;  
 singleOperator['\_'] = UNDER\_SCORE;  
  
 operators["&&"] = AND;  
 operators["||"] = OR;  
 operators["!"] = NOT;  
 operators["<"] = LT;  
 operators[">"] = GT;  
 operators["<>"] = NEQ;  
 operators["="] = ASSIGN;  
 operators["=="] = EQ;  
 operators[">="] = GEQ;  
 operators["<="] = LEQ;  
 operators["+"] = PLUS;  
 operators["-"] = MINUS;  
 operators["\*"] = MUL;  
 operators["/"] = DIV;  
 operators["("] = L\_PAREN;  
 operators[")"] = R\_PAREN;  
 operators[","] = COMMA;  
 operators[";"] = SEMICOLON;  
  
}  
  
*bool* Lexer::isOperator() *const* {  
 *return* \*curToken == '&' || \*curToken == '|' || \*curToken == '!' || \*curToken == '<'  
 || \*curToken == '>' || \*curToken == '=' || \*curToken == '+' || \*curToken == '-'  
 || \*curToken == '\*' || \*curToken == '/' || \*curToken == '(' || \*curToken == ')'  
 || \*curToken == ',' || \*curToken == ';';  
}

Parser.h

#pragma clang diagnostic push  
#pragma ide diagnostic ignored "modernize-use-nodiscard"  
  
  
#ifndef **NEWSQLPARSER\_PARSER\_H**#define **NEWSQLPARSER\_PARSER\_H**#include <utility>  
  
#include "../lexer/Lexer.h"  
#include "../lexer/Token.h"  
  
*enum* SqlType {  
 DML,  
 DDL  
};  
  
*class* Parser {  
  
*public*:  
 *explicit* Parser(Lexer &lexer) : lexer(lexer), lookAhead(lexer.advance()),  
 columnNumber(lexer.getColumnNumber()), lineNumber(lexer.getLineNumber()) {  
 }  
  
 *void* parseQuery();  
*private*:  
  
 Lexer &lexer;  
 Token lookAhead;  
 *int* columnNumber;  
 *int* lineNumber;  
  
 std::vector<std::string> dataTypes{"int", "char", "varchar", "text", "default",  
 " bool", "float", "double"," date","string"};  
  
  
 std::vector<std::string> columnConstrainsVal{"null", "not", "primary", "unique", "auto\_increment",  
 "default", "comments"};  
  
 std::vector<Type> comparisonOpr {  
 EQ,GT,LT,  
 };  
  
 std::vector<std::string> createTableColumnList ;  
  
 *bool* match(Type) *const*;  
  
 SqlType getStatementType() *const*;  
  
 *void* sqlStatement();  
  
 *void* eat(Type);  
  
  
 *void* sqlStatements();  
  
 *void* ddlStatement();  
  
 *void* dmlStatement();  
  
 *void* advance();  
  
 *void* createTable();  
  
 *void* createDatabase();  
  
 *void* selectQuery();  
  
 *void* insertQuery();  
  
 *void* ifNotExist();  
  
 std::string uid();  
  
 *void* createTableDefinitions();  
  
 *void* createDbOptions();  
  
 *void* selectElements();  
  
 *void* fromClause() ;  
  
 *void* groupByClause() *const* ;  
  
 *void* orderByClause() *const* ;  
  
 *void* limitClause() *const* ;  
  
 *void* insertOptions();  
  
 *void* createTableDefination();  
  
 *void* columnDef();  
  
 *void* tableOptions();  
  
 *void* dataType();  
  
 *void* columnConstrains();  
  
 *void* into();  
  
 *void* columnList();  
  
 *void* insertList();  
  
 *void* choice();  
  
 *void* selectElement();  
  
 *void* whereClause();  
  
 *void* expression();  
  
 *void* predicate();  
  
 *void* selectList();  
  
 std::string likeUid();  
  
 *void* atomicVal();  
};  
  
#endif *//NEWSQLPARSER\_PARSER\_H*

Parser.cpp

#include "Parser.h"  
  
#include <utility>  
#include <iostream>  
#include <algorithm>  
  
*class* ParserException : *public* std::exception {  
 std::string errorMsg;  
*public*:  
 *explicit* ParserException(std::string errorMsg) : errorMsg(std::move(errorMsg)) {}  
  
 [[nodiscard]] *const char* \*what() *const* **\_GLIBCXX\_TXN\_SAFE\_DYN \_GLIBCXX\_NOTHROW** *override* {  
 *return* errorMsg.c\_str();  
 }  
};  
  
  
*void* Parser::sqlStatement() {  
 *auto* statementType = getStatementType();  
 *if* (statementType == DDL) {  
 ddlStatement();  
 } *else* {  
 dmlStatement();  
 }  
}  
  
*void* Parser::ddlStatement() {  
 *if* (match(CREATE)) {  
 eat(CREATE);  
 *if* (lookAhead.type == DATABASE || lookAhead.type == SCHEMA) {  
 createDatabase();  
 } *else if* (lookAhead.type == TABLE || lookAhead.type == TEMPORARY) {  
 *if* (lookAhead.type == TEMPORARY) {  
 eat(TEMPORARY);  
 }  
 createTable();  
 } *else* {  
 *throw* ParserException(  
 "Only CREATE TABLE AND CREATE DATABASE are valid DDL statements. Error in line no " +  
 std::to\_string(lineNumber) + " and column number " + std::to\_string(columnNumber));  
 }  
 } *else* {  
 *throw* ParserException("Only CREATE DDL Statement is supported. Error in line no " + std::to\_string(lineNumber) +  
 " and column number " + std::to\_string(columnNumber));  
 }  
}  
  
*void* Parser::dmlStatement() {  
 *if* (match(SELECT)) {  
 selectQuery();  
 } *else if* (match(INSERT)) {  
 insertQuery();  
 } *else* {  
 *throw* ParserException(  
 "Only SELECT & INSERT DML Statements are supported in line no " + std::to\_string(lineNumber) +  
 " and column number " + std::to\_string(columnNumber));  
 }  
  
}  
  
*void* Parser::createTable() {  
 *if* (match(TABLE)) {  
 eat(TABLE);  
 ifNotExist();  
 std::string tableName = uid();  
 createTableDefinitions();  
 } *else* {  
 *throw* ParserException("Syntax Error in line no " + std::to\_string(lineNumber) + " and column number " +  
 std::to\_string(columnNumber));  
 }  
}  
  
*void* Parser::createDatabase() {  
 *if* (match(DATABASE)) {  
 eat(DATABASE);  
 } *else if* (match(SCHEMA)) {  
 eat(SCHEMA);  
 } *else* {  
 *throw* ParserException("Syntax Error in line no " + std::to\_string(lineNumber) + " and column number " +  
 std::to\_string(columnNumber));  
 }  
 ifNotExist();  
 std::string dbName = uid();  
 createDbOptions();  
 eat(SEMICOLON);  
}  
  
*void* Parser::selectQuery() {  
 *if* (match(SELECT)) {  
 eat(SELECT);  
 selectElements();  
 fromClause();  
 groupByClause();  
 orderByClause();  
 limitClause();  
 match(SEMICOLON);  
 } *else* {  
 *throw* ParserException("Syntax Error in line no " + std::to\_string(lineNumber) + " and column number " +  
 std::to\_string(columnNumber));  
  
 }  
}  
  
*void* Parser::insertQuery() {  
 *if* (match(INSERT)) {  
 eat(INSERT);  
 into();  
 std::string tableName = uid();  
 insertOptions();  
 } *else* {  
 *throw* ParserException("Syntax Error in line no " + std::to\_string(lineNumber) + " and column number " +  
 std::to\_string(columnNumber));  
 }  
}  
  
*void* Parser::eat(Type type) {  
 *if* (match(type)) {  
 advance();  
 } *else* {  
 *throw* ParserException("Syntax Error in line no " + std::to\_string(lineNumber) + " and column number " +  
 std::to\_string(columnNumber));  
  
 }  
  
}  
  
*void* Parser::ifNotExist() {  
 *if* (match(IF)) {  
 eat(IF);  
 *if* (match(NOT)) {  
 eat(NOT);  
 *if* (match(EXISTS)) {  
 eat(EXISTS);  
 } *else* {  
 *throw* ParserException("Syntax Error in line no " + std::to\_string(lineNumber) + " and column number " +  
 std::to\_string(columnNumber));  
 }  
 } *else* {  
 *throw* ParserException("Syntax Error in line no " + std::to\_string(lineNumber) + " and column number " +  
 std::to\_string(columnNumber));  
 }  
 }  
}  
  
std::string Parser::uid() {  
 *auto* flag = match(SINGLE\_QUOTE); *// Nikhil 'Nikhil' 25  
 if* (flag) {  
 eat(SINGLE\_QUOTE);  
 }  
 *if* (match(ID) || match(NUM)) {  
 *auto* str = lookAhead.value;  
  
 eat(lookAhead.type);  
 *if* (flag) {  
 eat(SINGLE\_QUOTE);  
 }  
 *return* str;  
 } *else* {  
 *throw* ParserException("Syntax Error in line no " + std::to\_string(lineNumber) + " and column number " +  
 std::to\_string(columnNumber));  
 }  
}  
  
*void* Parser::createTableDefinitions() {  
 eat(L\_PAREN);  
 createTableDefination();  
 *while* (match(COMMA)) {  
 eat(COMMA);  
 createTableDefination();  
 }  
 eat(R\_PAREN);  
  
}  
  
*void* Parser::createDbOptions() {  
 *if* (match(COLLATE)) {  
 eat(COLLATE);  
 std::string col\_name = uid();  
  
 } *else if* (match(ENCRYPTION)) {  
 eat(ENCRYPTION);  
 choice();  
  
 }  
}  
  
  
*void* Parser::selectElements() {  
 *if* (match(MUL)) {  
 eat(MUL);  
 } *else if* (match(ID)) {  
 selectElement();  
 *while* (match(COMMA)) {  
 eat(COMMA);  
 selectElement();  
 }  
  
 } *else* {  
 *throw* ParserException("Syntax Error in line no " + std::to\_string(lineNumber) + " and column number " +  
 std::to\_string(columnNumber));  
 }  
  
}  
  
*void* Parser::fromClause() {  
 *if* (match(FROM)) {  
 eat(FROM);  
 std::string tableName = uid();  
 whereClause();  
 } *else* {  
 *throw* ParserException("Syntax Error in line no " + std::to\_string(lineNumber) + " and column number " +  
 std::to\_string(columnNumber));  
 }  
  
}  
  
*void* Parser::groupByClause() *const* {  
 *// Implement this*}  
  
*void* Parser::orderByClause() *const* {  
 *// Implement this*}  
  
*void* Parser::limitClause() *const* {  
 *// Implement this*}  
  
*void* Parser::insertOptions() {  
 columnList();  
 *if* (match(VALUES)) {  
 eat(VALUES);  
 insertList();  
 } *else* {  
 *throw* ParserException("Syntax Error in line no " + std::to\_string(lineNumber) + " and column number " +  
 std::to\_string(columnNumber));  
 }  
  
}  
  
SqlType Parser::getStatementType() *const* {  
 *if* (lookAhead.type == CREATE) {  
 *return* DDL;  
 } *else if* (lookAhead.type == INSERT || lookAhead.type == SELECT) {  
 *return* DML;  
 } *else* {  
 *throw* ParserException(  
 "Sql statement must be DDL or DML statements Error in line no " + std::to\_string(lineNumber) +  
 " and column number " + std::to\_string(columnNumber));  
 }  
}  
  
*void* Parser::sqlStatements() {  
 *if* (lexer.query.empty() || lexer.query.size() == 1) {  
 *throw* ParserException("Enter Something");  
 } *else* {  
 sqlStatement();  
 }  
}  
  
*void* Parser::parseQuery() {  
 sqlStatements();  
}  
  
*void* Parser::createTableDefination() {  
 *if* (match(ID)) {  
 createTableColumnList.push\_back(uid());  
 columnDef(); *//Data type* } *else if* (match(PRIMARY) || match(FOREIGN)) {  
 tableOptions();  
 } *else* {  
 *// TODO throw error* }  
  
}  
  
*void* Parser::columnDef() {  
 *if* (*auto* itr = std::find(dataTypes.begin(), dataTypes.end(), lookAhead.value); itr != dataTypes.end()) {  
 eat(lookAhead.type);  
 *auto* it = std::find(columnConstrainsVal.begin(), columnConstrainsVal.end(), lookAhead.value);  
 *while* (it != columnConstrainsVal.end()) {  
 *switch* (lookAhead.type) {  
 *case* NOT: {  
 eat(NOT);  
*// eat(NULL); TODO handle not null cases  
 break*;  
 }  
 *case* PRIMARY: {  
 eat(PRIMARY);  
 eat(KEY);  
 *break*;  
 }  
 *case* UNIQUE: {  
 eat(UNIQUE);  
 *if* (lookAhead.type == KEY) {  
 eat(KEY);  
 }  
 *break*;  
 }  
 *case* KEY: {  
 eat(KEY);  
 *break*;  
 }  
 *case* AUTO\_INCREMENT: {  
 eat(AUTO\_INCREMENT);  
 *break*;  
 }  
 *case* DEFAULT: {  
 eat(DEFAULT);  
 *// TODO handle default value  
 break*;  
 }  
 *case* COMMENT: {  
 eat(COMMENT);  
 std::string comment = uid();  
 *break*;  
 }  
 *default* :  
 *throw* ParserException(  
 "Syntax Error in line no " + std::to\_string(lineNumber) + " and column number " +  
 std::to\_string(columnNumber));  
  
 }  
 it = std::find(columnConstrainsVal.begin(), columnConstrainsVal.end(), lookAhead.value);  
  
 }  
 } *else* {  
 *throw* ParserException("Syntax Error in line no " + std::to\_string(lineNumber) + " and column number " +  
 std::to\_string(columnNumber));  
 }  
}  
  
*void* Parser::tableOptions() {  
 *if* (match(PRIMARY)) {  
 eat(PRIMARY);  
 eat(KEY);  
 eat(L\_PAREN);  
 std::string columnName = uid();  
 *if* (*auto* itr = std::find(createTableColumnList.begin(), createTableColumnList.end(), columnName);  
 itr == createTableColumnList.end()) {  
 *throw* ParserException("Unknown variable " + columnName);  
 }  
 eat(R\_PAREN);  
  
 } *else if* (match(FOREIGN)) {  
 eat(FOREIGN);  
 eat(KEY);  
 eat(L\_PAREN);  
 std::string columnName = uid();  
  
 eat(R\_PAREN);  
  
 } *else* {  
 *// TODO throw error* }  
  
}  
  
*void* Parser::dataType() {  
  
}  
  
*void* Parser::columnConstrains() {  
  
}  
  
*void* Parser::into() {  
 *if* (match(INTO)) {  
 eat(INTO);  
 }  
}  
  
*void* Parser::columnList() {  
 *if* (match(VALUES)) {  
 *return*;  
 }  
 eat(L\_PAREN);  
 std::vector<std::string> columnName;  
 columnName.push\_back(uid());  
 *while* (match(COMMA)) {  
 eat(COMMA);  
 columnName.push\_back(uid());  
 }  
 eat(R\_PAREN);  
}  
  
*void* Parser::insertList() {  
 eat(L\_PAREN);  
  
 std::vector<std::string> columnName;  
 columnName.push\_back(uid());  
 *while* (match(COMMA)) {  
 eat(COMMA);  
 columnName.push\_back(uid());  
 }  
 eat(R\_PAREN);  
  
}  
  
*bool* Parser::match(Type type) *const* {  
 *return* lookAhead.type == type;  
}  
  
*void* Parser::advance() {  
 lookAhead = lexer.advance();  
 columnNumber = lexer.getColumnNumber();  
 lineNumber = lexer.getLineNumber();  
}  
  
*void* Parser::choice() {  
 *if* (match(Y)) {  
 eat(Y);  
  
 } *else if* (match(N)) {  
 eat(N);  
 } *else* {  
 *//TODO throw error* }  
  
}  
  
*void* Parser::selectElement() {  
 *if* (match(ID)) {  
 std::string columnName = uid();  
 } *else* {  
 *throw* ParserException("Syntax Error in line no " + std::to\_string(lineNumber) + " and column number " +  
 std::to\_string(columnNumber));  
 }  
  
}  
  
*void* Parser::whereClause() {  
 *if* (match(WHERE)) {  
 eat(WHERE);  
 expression();  
 }  
  
}  
  
*void* Parser::expression() {  
 *if* (match(NOT)) {  
 eat(NOT);  
 expression();  
 } *else if* (match(ID) || match(SINGLE\_QUOTE) || match(NUM)) {  
 predicate();  
 } *else* {  
 *throw* ParserException("Syntax Error in line no " + std::to\_string(lineNumber) + " and column number " +  
 std::to\_string(columnNumber));  
 }  
  
}  
  
*void* Parser::predicate() {  
 *if* (match(ID) || match(NUM) || match(SINGLE\_QUOTE)) {  
 atomicVal();  
 }  
 *if* (lookAhead.type == IN) {  
 eat(IN);  
 eat(L\_PAREN);  
 *if* (match(SELECT)) {  
 selectQuery();  
 } *else* {  
 selectList();  
 }  
 eat(R\_PAREN);  
 } *else if* (lookAhead.type == LIKE) {  
 eat(LIKE);  
 eat(SINGLE\_QUOTE);  
 std::string matchSTr = likeUid();  
 eat(SINGLE\_QUOTE);  
  
 } *else if* (lookAhead.type == BETWEEN) {  
 eat(BETWEEN);  
 uid();  
 eat(AND);  
 uid();  
 } *else if* (*auto* itr = std::find(comparisonOpr.begin(), comparisonOpr.end(), lookAhead.type); itr !=  
 comparisonOpr.end()) {  
 eat(lookAhead.type);  
 predicate();  
 } *else if* (match(SEMICOLON)) {  
 *return*;  
 } *else* {  
 *throw* ParserException("Syntax Error in line no " + std::to\_string(lineNumber) + " and column number " +  
 std::to\_string(columnNumber));  
 }  
}  
  
*void* Parser::atomicVal() {  
 *if* (match(NUM)) {  
 *try* {  
 *auto* val = std::stoi(lookAhead.value);  
 eat(lookAhead.type);  
*// predicate();* } *catch* (ParserException e) {  
 *throw* ParserException("Syntax Error in line no " + std::to\_string(lineNumber) + " and column number " +  
 std::to\_string(columnNumber));  
 }  
 } *else if* (match(ID) || match(SINGLE\_QUOTE)) {  
 std::string columnName = uid();  
*// predicate();* }  
}  
  
*void* Parser::selectList() {  
 *if* (match(NUM) || match(SINGLE\_QUOTE)) {  
 std::string value = uid();  
 *while* (match(COMMA)) {  
 eat(COMMA);  
 value = uid();  
 }  
 } *else* {  
 *throw* ParserException("Parsing Error in line no " + std::to\_string(lineNumber) + " and column number " +  
 std::to\_string(columnNumber));  
 }  
  
}  
  
std::string Parser::likeUid() {  
 *while* (match(PERCENT) || match(UNDER\_SCORE)) {  
 eat(lookAhead.type);  
 }  
 *if* (match(ID) || match(NUM)) {  
 *auto* str = lookAhead.value;  
 eat(lookAhead.type);  
 *if* (match(SINGLE\_QUOTE)) {  
 *return* str;  
 }  
 likeUid();  
 *return* str;  
 } *else* {  
 *throw* ParserException("Syntax Error in line no " + std::to\_string(lineNumber) + " and column number " +  
 std::to\_string(columnNumber));  
 }  
}

**Follow**

*void* find\_follow(vector< pair<string , vector<string>> >& gram, map< string, set<string> > &follows, map< string, set<string> > firsts, string non\_term, set<string>& terms) {  
 *for*(*auto* it = gram.begin(); it != gram.end(); ++it) {  
 *bool* finished = *true*;  
 *auto* ch = it->second.begin();  
 *for*(;ch != it->second.end() ; ++ch) {  
 *if*(\*ch == non\_term) {  
 finished = *false*;  
 *break*;  
 }  
 }  
 ++ch;  
  
 *for*(;ch != it->second.end() && !finished; ++ch) {  
 *if*(terms.find(\*ch) != terms.end()) {  
 follows[non\_term].insert(\*ch);  
 finished = *true*;  
 *break*;  
 }  
  
 set<string> firsts\_copy(firsts[\*ch]);  
 *if*(firsts\_copy.find("e") == firsts\_copy.end()) {  
 follows[non\_term].insert(firsts\_copy.begin(), firsts\_copy.end());  
 finished = *true*;  
 *break*;  
 }  
 firsts\_copy.erase("e");  
 follows[non\_term].insert(firsts\_copy.begin(), firsts\_copy.end());  
 }  
 *if*(ch == it->second.end() && !finished) {  
 *if*(follows[it->first].empty()) {  
 find\_follow(gram, follows, firsts, it->first, terms);  
 }  
 follows[non\_term].insert(follows[it->first].begin(), follows[it->first].end());  
 }  
  
 }  
  
}

**First**

*void* find\_first(vector< pair<string, vector<string>> >& gram, map< string, set<string> > &firsts, string curr\_non\_term, set<string>& terms) {  
 *for*(*auto* it = gram.begin(); it != gram.end(); ++it) {  
 *if*(it->first != curr\_non\_term) {  
 *continue*;  
 }  
 vector<string> rhs = it->second;  
 *for*(*auto* ch = rhs.begin(); ch != rhs.end(); ++ch) {  
 *if*(terms.find(\*ch) != terms.end() || ( (\*ch) == "e")) {  
 firsts[curr\_non\_term].insert(\*ch);  
 *break*;  
 }  
 *else* {  
 *if*(firsts[\*ch].empty()) {  
 find\_first(gram, firsts, \*ch, terms);  
 }  
 *if*(firsts[\*ch].find("e") == firsts[\*ch].end()) {  
 firsts[curr\_non\_term].insert(firsts[\*ch].begin(), firsts[\*ch].end());  
 *break*;  
 }  
  
 set<string> firsts\_copy(firsts[\*ch].begin(), firsts[\*ch].end());  
 *if*(ch + 1 != rhs.end()) {  
 firsts\_copy.erase("e");  
 }  
 firsts[curr\_non\_term].insert(firsts\_copy.begin(), firsts\_copy.end());  
 }  
 }  
 }  
}

**Parsing Table Constructor**

*int* parsingTable(vector< pair<string, vector<string>> >& gram, set<string> non\_terms, set<string>& terms, map< string, set<string> > firsts , map< string, set<string> > follows){  
  
 *int* parse\_table[non\_terms.size()][terms.size()];  
 fill(&parse\_table[0][0], &parse\_table[0][0] + *sizeof*(parse\_table)/*sizeof*(parse\_table[0][0]), -1);  
  
 *for*(*auto* prod = gram.begin(); prod != gram.end(); ++prod) {  
 vector<string> rhs = prod->second;  
  
 set<string> next\_list;  
 *bool* finished = *false*;  
 *for*(*auto* ch = rhs.begin(); ch != rhs.end(); ++ch) {  
 *if*(terms.find(\*ch) != terms.end()) {  
 *if*(\*ch != "e") {  
 next\_list.insert(\*ch);  
 finished = *true*;  
 *break*;  
 }  
 *continue*;  
 }  
  
 set<string> firsts\_copy(firsts[\*ch].begin(), firsts[\*ch].end());  
 *if*(firsts\_copy.find("e") == firsts\_copy.end()) {  
 next\_list.insert(firsts\_copy.begin(), firsts\_copy.end());  
 finished = *true*;  
 *break*;  
 }  
 firsts\_copy.erase("e");  
 next\_list.insert(firsts\_copy.begin(), firsts\_copy.end());  
 }  
 *// If the whole rhs can be skipped through epsilon or reaching the end  
 // Add follow to next list  
 if*(!finished) {  
 next\_list.insert(follows[prod->first].begin(), follows[prod->first].end());  
 }  
  
  
 *for*(*auto* ch = next\_list.begin(); ch != next\_list.end(); ++ch) {  
 *int* row = distance(non\_terms.begin(), non\_terms.find(prod->first));  
 *int* col = distance(terms.begin(), terms.find(\*ch));  
 *int* prod\_num = distance(gram.begin(), prod);  
 *if*(parse\_table[row][col] != -1) {  
 cout<<"Collision at ["<<row<<"]["<<col<<"] for production "<<prod\_num<<"\n";  
 *continue*;  
 }  
 parse\_table[row][col] = prod\_num;  
 }  
  
 }  
 *// Print parse table* cout<<"Parsing Table: \n";  
 cout<<" ";  
 *for*(*auto* i = terms.begin(); i != terms.end(); ++i) {  
 cout<<\*i<<" ";  
 }  
 cout<<"\n";  
 *for*(*auto* row = non\_terms.begin(); row != non\_terms.end(); ++row) {  
 cout<<\*row<<" ";  
 *for*(*int* col = 0; col < terms.size(); ++col) {  
 *int* row\_num = distance(non\_terms.begin(), row);  
 *if*(parse\_table[row\_num][col] == -1) {  
 cout<<"- ";  
 *continue*;  
 }  
 cout<<parse\_table[row\_num][col]<<" ";  
 }  
 cout<<"\n";  
 }  
 cout<<"\n";  
  
  
 vector<string> input = {"select", "\*", "from", "a", ";"};  
 vector<string> input\_string = input;  
 input\_string.push\_back("$");  
 stack<string> st;  
 st.push("$");  
 st.push("S");  
  
 *// Check if input string is valid  
 for*(*auto* ch = input\_string.begin(); ch != input\_string.end(); ++ch) {  
 *if*(terms.find(\*ch) == terms.end()) {  
 cout<<"Input string is invalid\n";  
 *return* 2;  
 }  
 }  
  
 *// cout<<"Processing input string\n";  
 bool* accepted = *true*;  
 *while*(!st.empty() && !input\_string.empty()) {  
 *// If stack top same as input string char remove it  
  
 if*(input\_string[0] == st.top()) {  
 st.pop();  
 input\_string.erase(input\_string.begin());  
 }  
 *//if non terminal found but lookhead doesn't match  
 else if*(terms.find(st.top()) != terms.end()) {  
 cout<<"Unmatched terminal found\n";  
 accepted = *false*;  
 *break*;  
 }  
 *else* {  
 string stack\_top = st.top();  
 *int* row = distance(non\_terms.begin(), non\_terms.find(stack\_top));  
 *int* col = distance(terms.begin(), terms.find(input\_string[0]));  
 *int* prod\_num = parse\_table[row][col];  
  
 *if*(prod\_num == -1) {  
 cout<<"No production found in parse table\n";  
 accepted = *false*;  
 *break*;  
 }  
  
Lodu Chand, [18.05.21 23:40]  
st.pop();  
 vector<string> rhs = gram[prod\_num].second;  
 *if*(rhs[0] == "e") {  
 *continue*;  
 }  
 *for*(*auto* ch = rhs.rbegin(); ch != rhs.rend(); ++ch) {  
 st.push(\*ch);  
 }  
 }  
 }  
  
 *if*(accepted) {  
 cout<<"Input string is accepted\n";  
 }  
 *else* {  
 cout<<"Input string is rejected\n";  
 }  
}