Compiler Design Lab

**Final Report**



***Language*: SQL**

**Team Members**

Ganesh sesha sai Akhil . K – AP21110010931

Lokesh . T -- AP21110010956

Sai Amruth . D -- AP21110010962

Sai Dheeraj . J -- AP21110011244

Lexical Analysis:

Lexical analysis is the first phase of a compiler. It takes modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.

# Tokens

Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions.

In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuation symbols can be considered as tokens.

CFG FOR AT LEAST FIVE CONSTRUCTS IN YOUR LANGUAGE:

1.Create Table :

database\_name → (table name) | (table name)\*

database\_statement → CREATE DATABASE ;

data\_type → int | char | float | bit | float | double | date | timestamp | varchar(255),

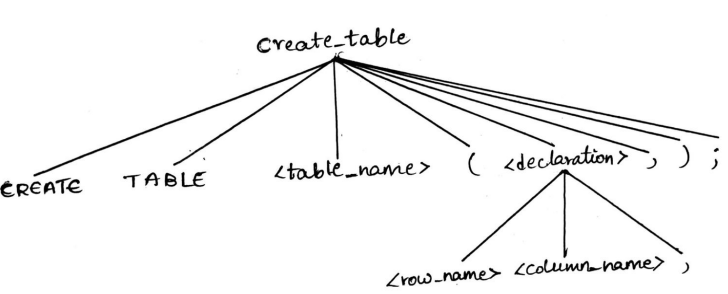
table\_name → (table name) | (table name)\* ,

variable\_name → (variable name) | (variable name)\*

declaration → | ( )\*

create\_table → CREATE TABLE ( table\_name);

Parse tree :



2.Drop / Remove table :

table\_name → (table name) | (table name)\*,

drop\_statement → DROP TABLE ;

Parse Tree :

A diagram of a table

Description automatically generated

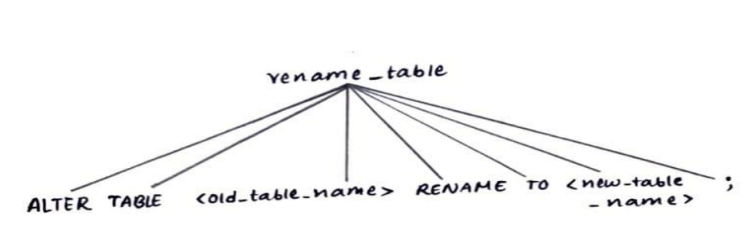
3.Rename :

old\_table\_name → (old table name) | (old table name)\*

new\_table\_name → (new table name) | (new table name)\*

rename\_table → ALTER TABLE\* RENAME TO\* ;

parse tree :



4.Select :

table\_name → (table name) | (table name)\*

column\_name → (column name) | (column name)\*

type\_1 → \* |

type\_2 → UNIQUE | DISTINCT | FIRST | LAST | COUNT

types → | select\_statement →

SELECT from ;

Parse Tree :

A diagram of a function

Description automatically generated with medium confidence

5.Insert :

table\_name → (table name) | (table name)\*

column\_name → (column name) | (column name)\*

columns → | ( , ,)\*

value → (a-z)\* | (A-Z)\* | (A-Z)(a-z)\*

values → | (,,)\*

Insert\_statement → INSERT INTO () VALUES();

Parse Tree :

A close-up of a diagram

Description automatically generated

AT LEAST TWO ITERATIVE STATEMENTS:

1.Using While Loop :

Example:

DECLARE @COUNT INT;

SET @Count = 1;

WHILE @count < = 5;

BEGIN

PRINT @count

SET @count = @count + 1;

END;

A diagram of a process

Description automatically generated

2.While loop with continue condition:

Example:

DECLARE @count INT;

DECLARE @mod INT;

SET @count = 1;

WHILE @count<= 10

BEGIN

set @mod = @count % 2

IF @mod = 1

BEGIN

SET @count = @count + 1;

CONTINUE

END

PRINT @count

SET @count = @count + 1;

END;

A diagram of a condition

Description automatically generated

Lexical Analysis of SQL Language:

Lex Code:

#include <stdio.h>

#include <string.h>

#include <regex.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX\_TOKENS 1000

typedef enum {

semi\_colon, comma, dot, quote, left\_parenthesis, right\_parenthesis,

equal, create, table, select, delete, from, where, insert, into, values,

update, set, drop, column, add, not, null\_token, primary, key, identifier,

data\_type, alter

} token\_class;

typedef struct {

int token\_class;

char \*token\_text;

int line\_no;

} token;

int token\_ptr = 0;

int current\_token\_ptr = -1;

token token\_list[MAX\_TOKENS];

int current\_line\_no = 0;

int flag = 1;

void insert\_token(token\_class class, const char \*text) {

token\_list[token\_ptr].token\_class = class;

token\_list[token\_ptr].token\_text = strdup(text);

token\_list[token\_ptr].line\_no = current\_line\_no;

token\_ptr++;

}

void tokenizefull() {

char \*a = (char \*)malloc(sizeof(char) \* 1000);

while (fgets(a, 1000, stdin)) {

current\_line\_no++;

char \*tok = strtok(a, " \t\n");

while (tok != NULL) {

int itemp = 0;

char \*temp = strdup(tok);

while (strlen(temp) > 0) {

while (!is\_type(temp))

temp[strlen(temp) - 1] = '\0';

itemp += strlen(temp);

strcpy(temp, substring(tok, itemp, strlen(tok)));

}

free(temp);

tok = strtok(NULL, " \t\n");

}

}

}

int yylex() {

if (flag == 1) {

tokenizefull();

flag = 0;

}

current\_token\_ptr++;

return token\_list[current\_token\_ptr].token\_class;

}

bool is\_type(const char \*temp) {

if (strcmp(temp, ";") == 0) {

insert\_token(semi\_colon, ";");

return true;

} else if (strcmp(temp, ",") == 0) {

insert\_token(comma, ",");

return true;

} else if (strcmp(temp, ".") == 0) {

insert\_token(dot, ".");

return true;

} else if (strcmp(temp, "'") == 0) {

insert\_token(quote, "'");

return true;

} else if (strcmp(temp, "(") == 0) {

insert\_token(left\_parenthesis, "(");

return true;

} else if (strcmp(temp, ")") == 0) {

insert\_token(right\_parenthesis, ")");

return true;

} else if (strcmp(temp, "=") == 0) {

insert\_token(equal, "=");

return true;

} else if (strcmp(temp, "create") == 0) {

insert\_token(create, "create");

return true;

} else if (strcmp(temp, "table") == 0) {

insert\_token(table, "table");

return true;

} else if (strcmp(temp, "select") == 0) {

insert\_token(select, "select");

return true;

} else if (strcmp(temp, "delete") == 0) {

insert\_token(delete, "delete");

return true;

} else if (strcmp(temp, "from") == 0) {

insert\_token(from, "from");

return true;

} else if (strcmp(temp, "where") == 0) {

insert\_token(where, "where");

return true;

} else if (strcmp(temp, "insert") == 0) {

insert\_token(insert, "insert");

return true;

} else if (strcmp(temp, "into") == 0) {

insert\_token(into, "into");

return true;

} else if (strcmp(temp, "values") == 0) {

insert\_token(values, "values");

return true;

} else if (strcmp(temp, "update") == 0) {

insert\_token(update, "update");

return true;

} else if (strcmp(temp, "set") == 0) {

insert\_token(set, "set");

return true;

} else if (strcmp(temp, "drop") == 0) {

insert\_token(drop, "drop");

return true;

} else if (strcmp(temp, "column") == 0) {

insert\_token(column, "column");

return true;

} else if (strcmp(temp, "add") == 0) {

insert\_token(add, "add");

return true;

} else if (strcmp(temp, "not") == 0) {

insert\_token(not, "not");

return true;

} else if (strcmp(temp, "null") == 0) {

insert\_token(null\_token, "null");

return true;

} else if (strcmp(temp, "primary") == 0) {

insert\_token(primary, "primary");

return true;

} else if (strcmp(temp, "key") == 0) {

insert\_token(key, "key");

return true;

} else if (strcmp(temp, "\*") == 0) {

insert\_token(identifier, "\*");

return true;

} else if (strcmp(temp, "integer") == 0) {

insert\_token(data\_type, "integer");

return true;

} else if (strcmp(temp, "alter") == 0) {

insert\_token(alter, "alter");

return true;

} else if (check\_regex(temp)) {

return true;

} else {

// printf("%s:unexpected character\n", temp);

return false;

}

}

int check\_regex(const char \*temp) {

regex\_t regex1, regex2, regex3;

int reti1, reti2, reti3;

char \*pattern1 = "^varchar([0-9][0-9]\*)$";

char \*pattern3 = "^[\_a-zA-Z][\_a-zA-Z0-9]\*$";

char \*pattern2 = "^[0-9][0-9]\*$";

reti1 = regcomp(&regex1, pattern1, 0);

reti2 = regcomp(&regex2, pattern2, 0);

reti3 = regcomp(&regex3, pattern3, 0);

if (!regexec(&regex1, temp, 0, NULL, 0)) {

insert\_token(data\_type, temp);

} else if (!regexec(&regex2, temp, 0, NULL, 0)) {

insert\_token(integer, temp);

} else if (!regexec(&regex3, temp, 0, NULL, 0)) {

insert\_token(identifier, temp);

} else {

return 0;

}

return 1;

}

char \*substring(const char \*input, int start, int end) {

return strndup(input + start, end - start);

}

const char \*yytext() {

return token\_list[current\_token\_ptr].token\_text;

}

int yylineno() {

return token\_list[current\_token\_ptr].line\_no;

}

int main(void) {

int ntoken = yylex();

while (ntoken) {

printf("%d:%s:%d\n", ntoken, yytext(), yylineno());

ntoken = yylex();

}

return 0;

}

lex.h:

#define add 1

#define alter 2

#define column 3 #define comma 4

#define create 5

#define data\_type 6

#define delete 7

#define dot 8

#define drop 9

#define equal 10

#define from 11

#define identifier 12

#define in 13

#define insert 14

#define integer 15

#define into 16 #define key 17

#define left\_parenthesis 18

#define not 19

#define null 20

#define primary 21

#define quote 22

#define right\_parenthesis 23

#define select 24

#define semi\_colon 25

#define set 26

#define table 27

#define update 28

#define values 29

#define where 30 input.txt:

create table student( id integer not null primary key, name varchar(10) not null, hostel varchar(10)

);

select name from student;

select name from student where data1 = (select id from student where hostel = umiam

);

Output:

A screen shot of a computer

Description automatically generated

A black background with white text

Description automatically generated

Parser:

Parser is that phase of the compiler which takes token string as input and with the help of existing grammar, converts it into the corresponding parse tree. Parser is also known as Syntax Analyzer.

Lex Code:

%{

#include <stdio.h> #include "y.tab.h" extern int yylval;

%}

%%

select return SELECT; from return FROM; where return WHERE;

and return AND;

[\*] return \*yytext;

[,] return \*yytext;

[=] return \*yytext;

[a-zA-Z][a-zA-Z0-9]+ return IDENTIFIER;

\n return \*yytext;

[ \t]+ /\* ignore whitespace \*/;

%%

YACC Code:

%{

#include <stdio.h>

void yyerror (const char \*str) { fprintf(stderr, "error: %s\n", str);

}

int yywrap() {

return 1;

}

main() { yyparse();

}

%}

%%

%token SELECT FROM IDENTIFIER WHERE AND;

line: select items using condition '\n' { printf("Syntax Correct\n"); }; select: SELECT;

items: '\*' | identifiers; identifiers: IDENTIFIER | IDENTIFIER ',' identifiers;

using: FROM IDENTIFIER WHERE; condition: IDENTIFIER '=' IDENTIFIER | IDENTIFIER '=' IDENTIFIER AND condition;

%%

Parser Output:

A screenshot of a computer program

Description automatically generated

Commands User for Syntax Correct:

1. flex lex.l

2. bison -d yacc.y

3. gcc lex.yy.c y.tab.c -w -o lexyacc

4.lexyacc

**Insights of Building SQL Compiler:**

* We have chosen Ubuntu as our Linux environment. We have used packages such as Yacc and lex to build this project.
* We faced many issues while building a Lexical Analyzer for this language.
* We faced many issues while generating tokens accordingly.
* Most of our time was spent building this lexical analyzer.
* We developed this parsing code in short lines covering a few CFG’s.
* We couldn't upgrade the parser as we are facing many bugs later.