Compiler Design Lab
Phase-II Report

***Language*: SQL**

Team Members

K.Pranay - AP19110010349
K.Smaran - AP19110010350
M.Gayathri - AP19110010352
E.Sravanthi - AP19110010354
K.Padmini - AP19110010381
A.Nanda Kishore - AP19110010391
K.Bineeth Kumar - AP19110010396

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.

Lexical Analysis of SQL Language:

Lex Code:

```
#include <stdio.h>
#include <string.h>
#include <regex.h>
#include <stdlib.h>
#include <stdbool.h>
#include "lex.h"


bool is_type(char *);
char* substring(char * , int ,int );
int check_regex(char *);
char* yytext(void);
int yylineno(void);

int flag =1;
```

```
/*char *tokentype[] = {";", ",", "'", "(", ")", "=", ".", "create", "table", "select", "delete", "from", "where", "insert",
"into", "integer", "in", "values", "update", "set", "drop", "column", "add", "not", "NULL", "primary", "key", "*",
"alter"};*/


typedef struct token{
        int token_class;
        char *token_text;
        int line_no;
} token;

int token_ptr = 0;                      // icrement at each insertion for new token in token_list
int current_token_ptr = -1;                     // increment at each call for yylex()
token token_list[1000];


int current_line_no = 0;


int tokenizefull(){
        char *a=(char * )malloc(sizeof(char)*1000);
        while(gets(a)){
                current_line_no++;
                char *tok= strtok(a," \t\n");
                while(tok != NULL){     int itemp=0;
                        char *temp=(char * )malloc(sizeof(char)*100);
                        strcpy(temp ,tok);
                        while(strlen(temp) >0){
                                while(!is_type(temp))       //define this func s.t it adds qualifying tokens
to main token array.
                                        temp[strlen(temp)-1] ='\0';   //NULL char

                                itemp += strlen(temp);
                                strcpy(temp, substring(tok, itemp, strlen(tok)));  //define this func (string,
start index, end index)

                        }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(char *temp){


		if ( strcmp(temp, ";")== 0 ) {
			token_list[token_ptr].token_class = semi_colon;
			token_list[token_ptr].token_text = ";";
			token_list[token_ptr].line_no = current_line_no;
			token_ptr++;
			return true;
			}
		else if ( strcmp(temp, ",")== 0 ) {
			token_list[token_ptr].token_class = comma;
			token_list[token_ptr].token_text = ",";
			token_list[token_ptr].line_no = current_line_no;
			token_ptr++;
			return true;
			}
		else if ( strcmp(temp, ".")== 0 ) {
			token_list[token_ptr].token_class = dot;
			token_list[token_ptr].token_text = ".";
			token_list[token_ptr].line_no = current_line_no;
			token_ptr++;
			return true;
			}
		else if ( strcmp(temp, "'")== 0 ) {
			token_list[token_ptr].token_class = quote;
			token_list[token_ptr].token_text = "'";
			token_list[token_ptr].line_no = current_line_no;
			token_ptr++;
			return true;
			}
		else if ( strcmp(temp, "(")== 0 ) {
			token_list[token_ptr].token_class = left_parenthesis;
			token_list[token_ptr].token_text = "(";
			token_list[token_ptr].line_no = current_line_no;
			token_ptr++;
			return true;
			}
		else if ( strcmp(temp, ")")== 0 ) {
			token_list[token_ptr].token_class = right_parenthesis;
			token_list[token_ptr].token_text = ")";
			token_list[token_ptr].line_no = current_line_no;
```

```
                token_ptr++;
                return true;
                }
        else if ( strcmp(temp, "=")== 0 ) {
                token_list[token_ptr].token_class = equal;
                token_list[token_ptr].token_text = "=";
                token_list[token_ptr].line_no = current_line_no;
                token_ptr++;
                return true;
                }
        else if ( strcmp(temp, "create")== 0 ) {
                token_list[token_ptr].token_class = create;
                token_list[token_ptr].token_text = "create";
                token_list[token_ptr].line_no = current_line_no;
                token_ptr++;
                return true;
                }
        else if ( strcmp(temp, "table")== 0 ) {
                token_list[token_ptr].token_class = table;
                token_list[token_ptr].token_text = "table";
                token_list[token_ptr].line_no = current_line_no;
                token_ptr++;
                return true;
                }
        else if ( strcmp(temp, "select")== 0 ) {
                token_list[token_ptr].token_class = select;
                token_list[token_ptr].token_text = "select";
                token_list[token_ptr].line_no = current_line_no;
                token_ptr++;
                return true;
                }
        else if ( strcmp(temp, "delete")== 0 ) {
                token_list[token_ptr].token_class = delete;
                token_list[token_ptr].token_text = "delete";
                token_list[token_ptr].line_no = current_line_no;
                token_ptr++;
                return true;
                }
        else if ( strcmp(temp, "from")== 0 ) {
                token_list[token_ptr].token_class = from;
                token_list[token_ptr].token_text = "from";
                token_list[token_ptr].line_no = current_line_no;
                token_ptr++;
                return true;
                }
        else if ( strcmp(temp, "where")== 0 ) {
                token_list[token_ptr].token_class = where;
                token_list[token_ptr].token_text = "where";
                token_list[token_ptr].line_no = current_line_no;
```

```
		token_ptr++;
		return true;
		}
	else if ( strcmp(temp, "insert")== 0 ) {
		token_list[token_ptr].token_class = insert;
		token_list[token_ptr].token_text = "insert";
		token_list[token_ptr].line_no = current_line_no;
		token_ptr++;
		return true;
		}
	else if ( strcmp(temp, "into")== 0 ) {
		token_list[token_ptr].token_class = into;
		token_list[token_ptr].token_text = "into";
		token_list[token_ptr].line_no = current_line_no;
		token_ptr++;
		return true;
		}
	else if ( strcmp(temp, "values")== 0 ) {
		token_list[token_ptr].token_class = values;
		token_list[token_ptr].token_text = "values";
		token_list[token_ptr].line_no = current_line_no;
		token_ptr++;
		return true;
		}
	else if ( strcmp(temp, "update")== 0 ) {
		token_list[token_ptr].token_class = update;
		token_list[token_ptr].token_text = "update";
		token_list[token_ptr].line_no = current_line_no;
		token_ptr++;
		return true;
		}
	else if ( strcmp(temp, "set")== 0 ) {
		token_list[token_ptr].token_class = set;
		token_list[token_ptr].token_text = "set";
		token_list[token_ptr].line_no = current_line_no;
		token_ptr++;
		return true;
		}
	else if ( strcmp(temp, "drop")== 0 ) {
		token_list[token_ptr].token_class = drop;
		token_list[token_ptr].token_text = "drop";
		token_list[token_ptr].line_no = current_line_no;
		token_ptr++;
		return true;
		}
	else if ( strcmp(temp, "column")== 0 ) {
		token_list[token_ptr].token_class = column;
		token_list[token_ptr].token_text = "column";
		token_list[token_ptr].line_no = current_line_no;
```

```
            token_ptr++;
            return true;
            }
    else if ( strcmp(temp, "add")== 0 ) {
            token_list[token_ptr].token_class = add;
            token_list[token_ptr].token_text = "add";
            token_list[token_ptr].line_no = current_line_no;
            token_ptr++;
            return true;
            }
    else if ( strcmp(temp, "not")== 0 ) {
            token_list[token_ptr].token_class = not;
            token_list[token_ptr].token_text = "not";
            token_list[token_ptr].line_no = current_line_no;
            token_ptr++;
            return true;
            }
    else if ( strcmp(temp, "null")== 0 ) {
            token_list[token_ptr].token_class = null;
            token_list[token_ptr].token_text = "null";
            token_list[token_ptr].line_no = current_line_no;
            token_ptr++;
            return true;
            }
    else if ( strcmp(temp, "primary")== 0 ) {
            token_list[token_ptr].token_class = primary;
            token_list[token_ptr].token_text = "primary";
            token_list[token_ptr].line_no = current_line_no;
            token_ptr++;
            return true;
            }
    else if ( strcmp(temp, "key")== 0 ) {
            token_list[token_ptr].token_class = key;
            token_list[token_ptr].token_text = "key";
            token_list[token_ptr].line_no = current_line_no;
            token_ptr++;
            return true;
            }
    else if ( strcmp(temp, "*")== 0 ) {
            token_list[token_ptr].token_class = identifier;
            token_list[token_ptr].token_text = "*";
            token_list[token_ptr].line_no = current_line_no;
            token_ptr++;
            return true;
            }
    else if ( strcmp(temp, "integer")== 0 ) {
            token_list[token_ptr].token_class = data_type;
            token_list[token_ptr].token_text = "integer";
            token_list[token_ptr].line_no = current_line_no;
```

```
                token_ptr++;
                return true;
                }
        else if ( strcmp(temp, "alter")== 0 ) {
                token_list[token_ptr].token_class = alter;
                token_list[token_ptr].token_text = "alter";
                token_list[token_ptr].line_no = current_line_no;
                token_ptr++;
                return true;
                }
        else if ( strcmp(temp, "alter")== 0 ) {
                token_list[token_ptr].token_class = alter;
                token_list[token_ptr].token_text = "alter";
                token_list[token_ptr].line_no = current_line_no;
                token_ptr++;
                return true;
                }
        else{
                if(check_regex(temp)){                  //if regex then return identifier else erro
                        return true;
                }
                else
                        return false;

        }


}


int check_regex(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)){            // if match then 0 else 1
                token_list[token_ptr].token_class = data_type;
                token_list[token_ptr].token_text=(char *)malloc(strlen(temp)*sizeof(char));
                strcpy(token_list[token_ptr].token_text, temp);
                token_list[token_ptr].line_no = current_line_no;
                token_ptr++;
        }
        else if(!regexec(&regex2, temp, 0, NULL, 0)){               // if match then 0 else 1
```

```
			token_list[token_ptr].token_class = integer;
			token_list[token_ptr].token_text=(char *)malloc(strlen(temp)*sizeof(char));
			strcpy(token_list[token_ptr].token_text , temp);
			token_list[token_ptr].line_no = current_line_no;
			token_ptr++;
		}
		else if(!regexec(&regex3, temp, 0, NULL, 0)){			// if match then 0 else 1
			token_list[token_ptr].token_class = identifier;
			token_list[token_ptr].token_text=(char *)malloc(strlen(temp)*sizeof(char));
			strcpy(token_list[token_ptr].token_text ,temp);
			token_list[token_ptr].line_no = current_line_no;
			token_ptr++;
		}
		else{
//			printf("%s:unexpected character\n",temp);
			return 0;
		}

		return 1;

}


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

		char *temp;
		temp = memcpy(temp,&input[start],end-start);
		temp[end-start] = '\0';
		return temp;
}


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, vtoken;
		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:

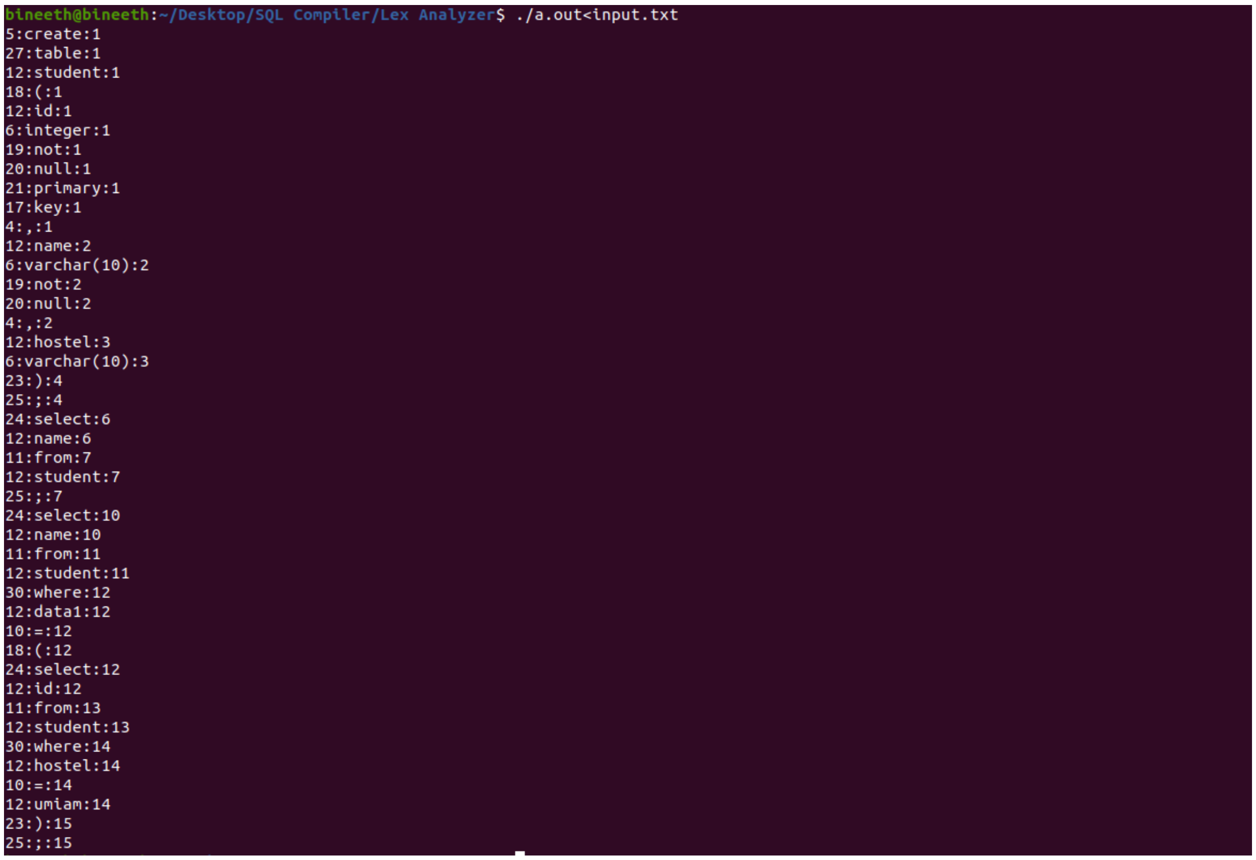
```
create table student( id integer not null primary key,
                      name varchar(10) not null,
                      hostel varchar(10)
```

```
                              );

select name
from student;


select name
from student
where data1 = (select id
                      from student
                      where hostel = umiam
                     );
```

Output:



```
bineeth@bineeth:~/Desktop/SQL Compiler/Lex Analyzer$ ./a.out<input.txt
5:create:1
27:table:1
12:student:1
18:(:1
12:id:1
6:integer:1
19:not:1
20:null:1
21:primary:1
17:key:1
4:,:1
12:name:2
6:varchar(10):2
19:not:2
20:null:2
4:,:2
12:hostel:3
6:varchar(10):3
23:):4
25:;:4
24:select:6
12:name:6
11:from:7
12:student:7
25:;:7
24:select:10
12:name:10
11:from:11
12:student:11
30:where:12
12:data1:12
10:=:12
18:(:12
24:select:12
12:id:12
11:from:13
12:student:13
30:where:14
12:hostel:14
10:=:14
12:umiam:14
23:):15
25:;:15
```

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;

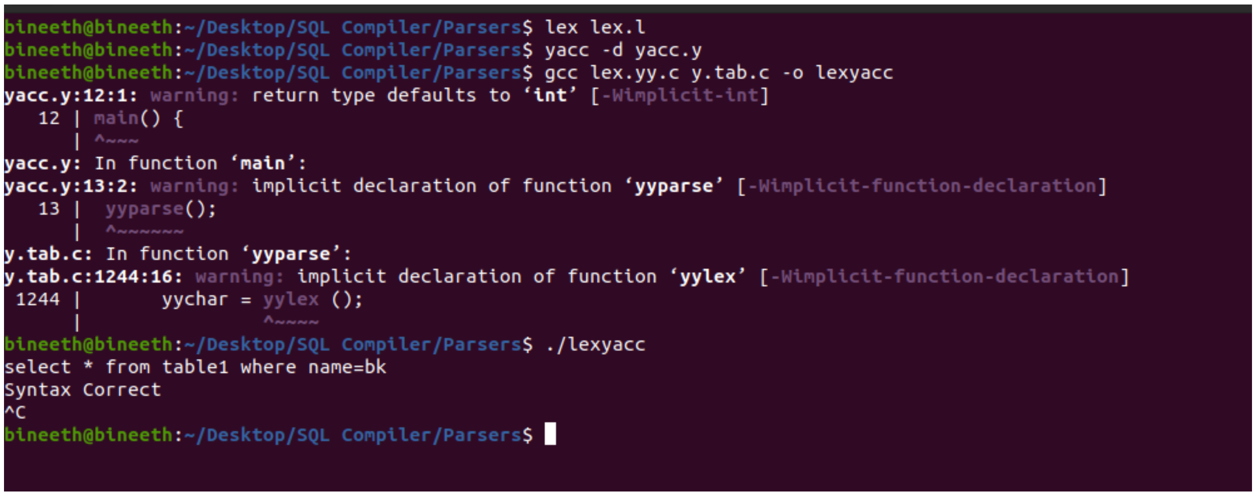
identifiers: IDENTIFIER | IDENTIFIER ',' identifiers;

using: FROM IDENTIFIER WHERE;

condition: IDENTIFIER '=' IDENTIFIER | IDENTIFIER '=' IDENTIFIER AND condition;

%%

Output:



```
bineeth@bineeth:~/Desktop/SQL Compiler/Parsers$ lex lex.l
bineeth@bineeth:~/Desktop/SQL Compiler/Parsers$ yacc -d yacc.y
bineeth@bineeth:~/Desktop/SQL Compiler/Parsers$ gcc lex.yy.c y.tab.c -o lexyacc
yacc.y:12:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   12 | main() {
      | ^~~~
yacc.y: In function ‘main’:
yacc.y:13:2: warning: implicit declaration of function ‘yyparse’ [-Wimplicit-function-declaration]
   13 |  yyparse();
      |  ^~~~~~~
y.tab.c: In function ‘yyparse’:
y.tab.c:1244:16: warning: implicit declaration of function ‘yylex’ [-Wimplicit-function-declaration]
 1244 |       yychar = yylex ();
      |                ^~~~~
bineeth@bineeth:~/Desktop/SQL Compiler/Parsers$ ./lexyacc
select * from table1 where name=bk
Syntax Correct
^C
bineeth@bineeth:~/Desktop/SQL Compiler/Parsers$
```

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.