GITHUB LINK:
https://github.com/cs-ubbcluj-ro/lab-work-computer-science-2024-Sergiu2404.git

LEX SPEC lang.lxi:

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int currentLine = 1;
%}

%option noyywrap

STRING_CONST       [\"][a-zA-Z0-9 ]+[\"]
CHAR_CONST         [\'][a-zA-Z0-9 ][\']
IDENTIFIER   [a-zA-Z_][a-zA-Z0-9_]*
NUMBER_CONST     0|[+|-]?[1-9][0-9]*([.][0-9]*)?|[+|-]?0[.][0-9]*

%%

"start"|"read"|"write"|"if"|"elif"|"else"|"for"|"while"|"int"|"string"|"char"|"array"|"return"
{printf("<%s> is a reserved word\n", yytext);}
"smallerEq"|"greaterEq"|"is"|"isNot"|"smaller"|"greater"|"takes"|"+"|"-"|"*"|"/"|"%"
{printf("<%s> is an operator\n", yytext);}
"{"|"}"|"("|")"|"["|"]"|":"|";"|","|"'"|"\""                    {printf("<%s> is a separator\n", yytext);}
{IDENTIFIER}            {printf("<%s> is an identifier\n", yytext);}
{NUMBER_CONST}              {printf("<%s> is a number\n", yytext);}
{STRING_CONST}              {printf("<%s> is a string constant\n", yytext);}
{CHAR_CONST}                {printf("<%s> is a char const\n", yytext);}

[ \t]+              {}
[\n]+     {currentLine++;}

[0-9][a-zA-Z0-9_]*              {printf("Illegal identifier at line %d\n", currentLine);}
[+|-]0              {printf("Illegal numeric constant at line %d\n", currentLine);}
[+|-]?[0][0-9]*([.][0-9]*)?                 {printf("Illegal numeric constant at line %d\n",
currentLine);}
[\'][a-zA-Z0-9 ]{2,}[\']|[\'][a-zA-Z0-9 ][a-zA-Z0-9 ][\']              {printf("Illegal character constant
at line %d\n", currentLine);}
[\"][a-zA-Z0-9_]+|[a-zA-Z0-9_]+[\"]             {printf("Illegal string constant at line %d\n",
currentLine);}

%%
```

```
void main(argc, argv)
int argc;
char** argv;
{
if (argc > 1)
{
    FILE *file;
    file = fopen(argv[1], "r");
    if (!file)
    {
        fprintf(stderr, "Could not open %s\n", argv[1]);
        exit(1);
    }
    yyin = file;
}

yylex();
}
```

YACC SPEC parser.y:

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

extern FILE* yyin;
extern int currentLine;
extern int yylex();
void yyerror(const char* s);
%}

%union {
    char* str;
}

%token <str> IDENTIFIER STRING_CONST CHAR_CONST NUMBER_CONST
%token INT FLOAT STRING CHAR READ WRITE IF ELIF ELSE FOR WHILE RETURN
START END ARRAY
%token TAKES SMALLER GREATER SMALLER_EQ GREATER_EQ IS IS_NOT
%token PLUS MINUS MUL DIV MOD LEFT_CURLY_BRACKET
RIGHT_CURLY_BRACKET
%token LEFT_ROUND_BRACKET RIGHT_ROUND_BRACKET LEFT_BRACKET
RIGHT_BRACKET
```

```
%token COLON SEMICOLON COMMA SQRT

%type <str> simple_type constant expression term factor relation function_call

%start program

%%
program:
    START statement_list END { printf("Program parsed successfully.\n"); }
;

statement_list:
    statement
    | statement statement_list
;

statement:
    simple_statement SEMICOLON
    | compound_statement
    | if_statement SEMICOLON
    | loop_statement SEMICOLON
;

simple_statement:
    declaration
    | assignment_statement
    | io_statement
    | return_statement
;

declaration:
    simple_type IDENTIFIER { printf("Declaration: %s of type %s\n", $2, $1); }
    | ARRAY simple_type IDENTIFIER LEFT_BRACKET RIGHT_BRACKET { printf("Array
declaration: %s of type %s\n", $3, $2); }
;

simple_type:
    INT { $$ = strdup("int"); }
    | FLOAT { $$ = strdup("float"); }
    | STRING { $$ = strdup("string"); }
    | CHAR { $$ = strdup("char"); }
;

assignment_statement:
    IDENTIFIER TAKES expression { printf("Assignment: %s takes expression\n", $1); }
;

io_statement:
```

```
    READ LEFT_ROUND_BRACKET IDENTIFIER RIGHT_ROUND_BRACKET { printf("Read
input into: %s\n", $3); }
    | WRITE LEFT_ROUND_BRACKET expression RIGHT_ROUND_BRACKET {
printf("Write output expression\n"); }
;

if_statement:
    if_part elif_parts else_part
    | if_part elif_parts
    | if_part else_part
    | if_part
;

if_part:
    IF LEFT_ROUND_BRACKET condition RIGHT_ROUND_BRACKET
compound_statement
;

elif_parts:
    elif_part
    | elif_part elif_parts
;

elif_part:
    ELIF LEFT_ROUND_BRACKET condition RIGHT_ROUND_BRACKET
compound_statement
;

else_part:
    ELSE compound_statement
;

compound_statement:
    LEFT_CURLY_BRACKET statement_list RIGHT_CURLY_BRACKET
;

loop_statement:
    WHILE LEFT_ROUND_BRACKET condition RIGHT_ROUND_BRACKET
compound_statement
    | FOR LEFT_ROUND_BRACKET assignment_statement SEMICOLON condition
SEMICOLON assignment_statement RIGHT_ROUND_BRACKET compound_statement
;

return_statement:
    RETURN expression { printf("Return value.\n"); }
;

condition:
```

```
    expression relation expression
;

relation:
    SMALLER { $$ = strdup("<"); }
    | GREATER { $$ = strdup(">"); }
    | SMALLER_EQ { $$ = strdup("<="); }
    | GREATER_EQ { $$ = strdup(">="); }
    | IS { $$ = strdup("=="); }
    | IS_NOT { $$ = strdup("!="); }
;

expression:
    expression PLUS term { $$ = strdup("addition"); }
    | expression MINUS term { $$ = strdup("subtraction"); }
    | term { $$ = $1; }
;

term:
    term MUL factor { $$ = strdup("multiplication"); }
    | term DIV factor { $$ = strdup("division"); }
    | term MOD factor { $$ = strdup("modulus"); }
    | factor { $$ = $1; }
;

factor:
    LEFT_ROUND_BRACKET expression RIGHT_ROUND_BRACKET { $$ = $2; }
    | function_call { $$ = $1; }
    | IDENTIFIER { $$ = $1; }
    | constant { $$ = $1; }
;

function_call:
    SQRT LEFT_ROUND_BRACKET expression RIGHT_ROUND_BRACKET { $$ =
strdup("sqrt_function"); }
;

constant:
    NUMBER_CONST { $$ = $1; }
    | STRING_CONST { $$ = $1; }
    | CHAR_CONST { $$ = $1; }
;
%%

void yyerror(const char* s) {
    fprintf(stderr, "Error at line %d: %s\n", currentLine, s);
}
```

```
int main(int argc, char** argv) {
    if (argc > 1) {
        FILE* file = fopen(argv[1], "r");
        if (!file) {
            fprintf(stderr, "Could not open file %s\n", argv[1]);
            return 1;
        }
        yyin = file;
    }

    int result = yyparse();
    if (result == 0) {
        printf("Parsing completed successfully.\n");
    }

    return result;
}
```

INSTRUCTIONS:
bison -d parser.y (generates files parser.tab.c and parser.tab.h)
flex lang.lxi
gcc lex.yy.c parser.tab.c
now an exe file was created (a.exe most of the times), so for running one of our examples (p1.txt, ...):
a.exe p1.txt
The output represents all of the tokens classified and steps inside the program