Umm Al-Qura University

Computer Science & Artificial Intelligent

College Computer Science Department

COMPILER CONSTRUCTION CS2341 PROJECT

GROUP ID: 11

Appendix

Name	Part	ID
ليان صالح القرشي	Report and research	44511056
وجن العتيبي	Part 1 task ll	445005620
ريتاج خالد القرشي	Part 2 task ll	44512037
ريم الشهراني	Part 1 task lll	445000375
سارة سلطان المطرفي	Report and research	445005027

Introduction

Description:

This project seeks to provide a practical understanding of the main components of compiler constructing by implementing a lexical analyzer (LEX) and a syntax analyzer (YACC). These tools are crucial in the early stages of compilation because they break down source code into understandable tokens and check their syntactic structure by using a specific grammar. In addition, **a symbol table** is used to store and track identifiers and keywords discovered during lexical analysis.

Structure of the Report:

1. **Lexical Analysis**: This part presents research findings on LEX, its syntax, and the implementation of the lexical analyzer, including code, input, and output screenshots.
2. **Parsing**: This section covers YACC research, including its structure and integration with LEX, as well as the syntax analyzer implementation and associated screenshots.
3. **Summary**: Reflections on project problems, and lessons learned throughout the completion of the project
4. **Citations / References** of all resources used to complete the project

Lexical Analysis

1) What is LEX?

Lex is a tool or program that creates a lexical analyzer and helps us perform the task of lexical analysis (It converts characters stream into tokens).

2) How does LEX work?

The working of lex in compiler design as a lexical analysis takes place in multiple steps. Firstly, we create a file that describes the generation of the lex analyzer. This file is written in Lex language and has a. l extension. The lex compiler converts this program into a C file called lex.yy.c. The C compiler then runs this C file, and it is compiled into file. This file is our working Lexical Analyzer which will produce the stream of tokens based on the input text.

3) What is the syntax of LEX code?

Declarations (The declarations include declarations of variables.)

%%

Translation rules (These rules consist of Pattern and Action.)

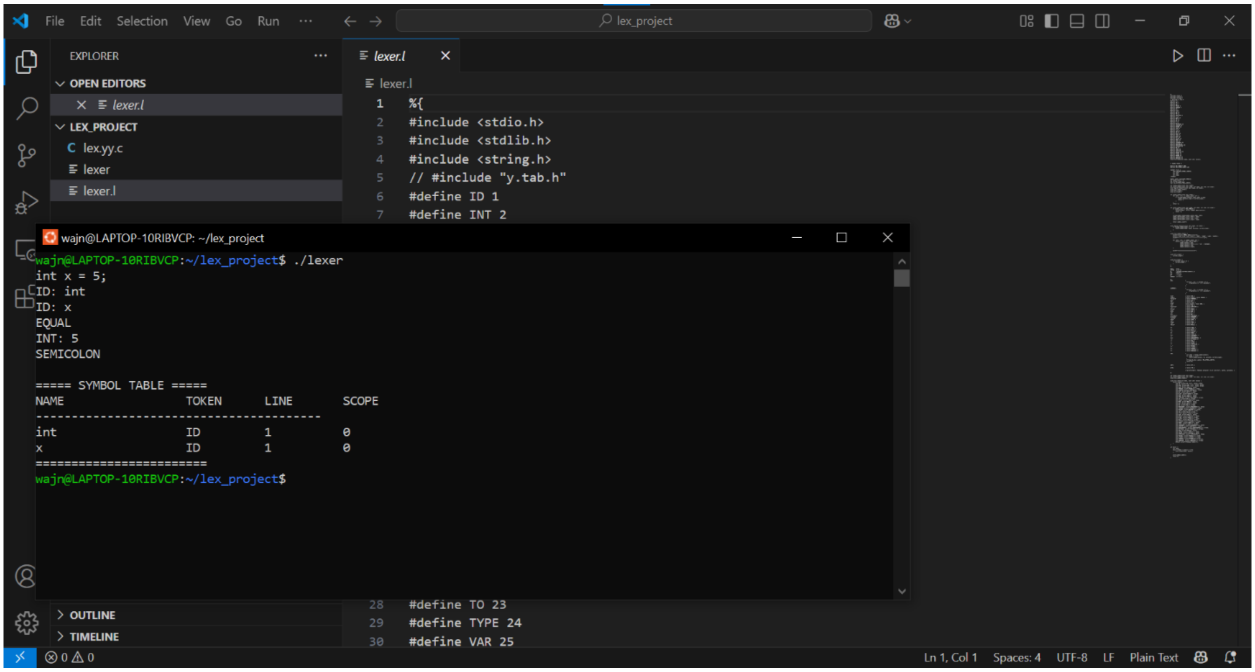
%%

Auxiliary procedures (The Auxilary section holds auxiliary functions used in the actions.)

4) Can you use an IDE for LEX development?

Yes, it’s possible. While LEX is typically used from the command line, it can be integrated with IDEs that support C programming, such as Code::Blocks, Eclipse CDT, or Visual Studio Code. These IDEs help with writing (.l) files, syntax highlighting, and compiling the generated C code. Some configuration may be needed to run LEX and compile the resulting files.

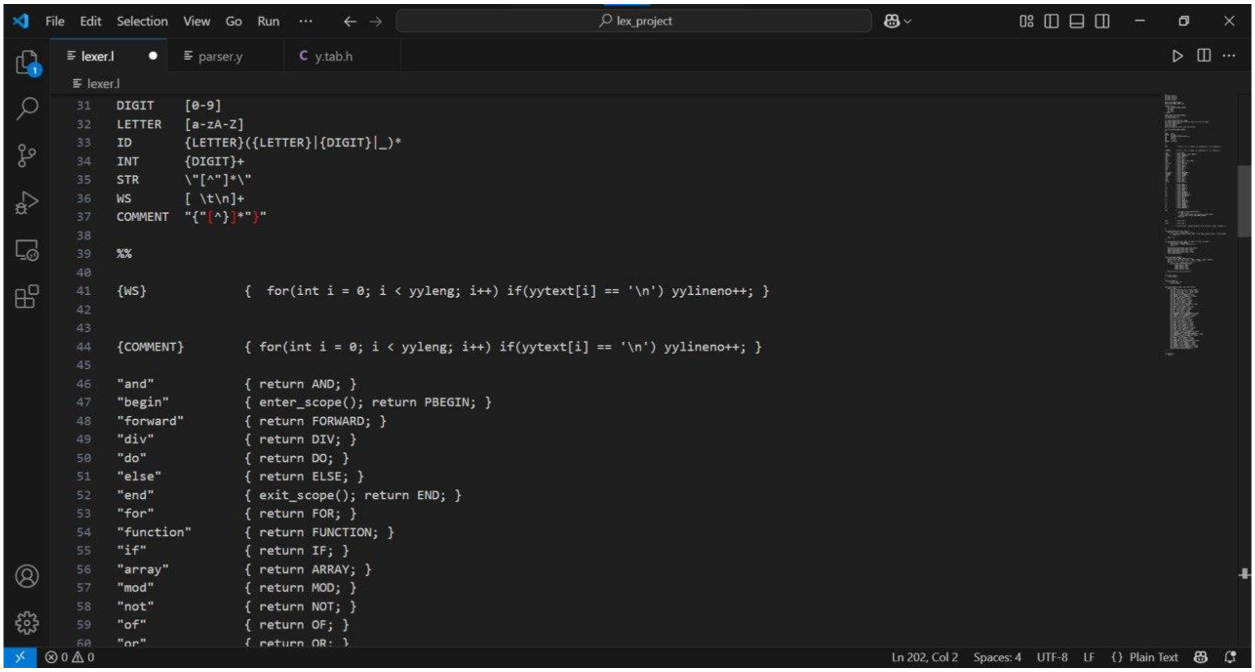
The output / input

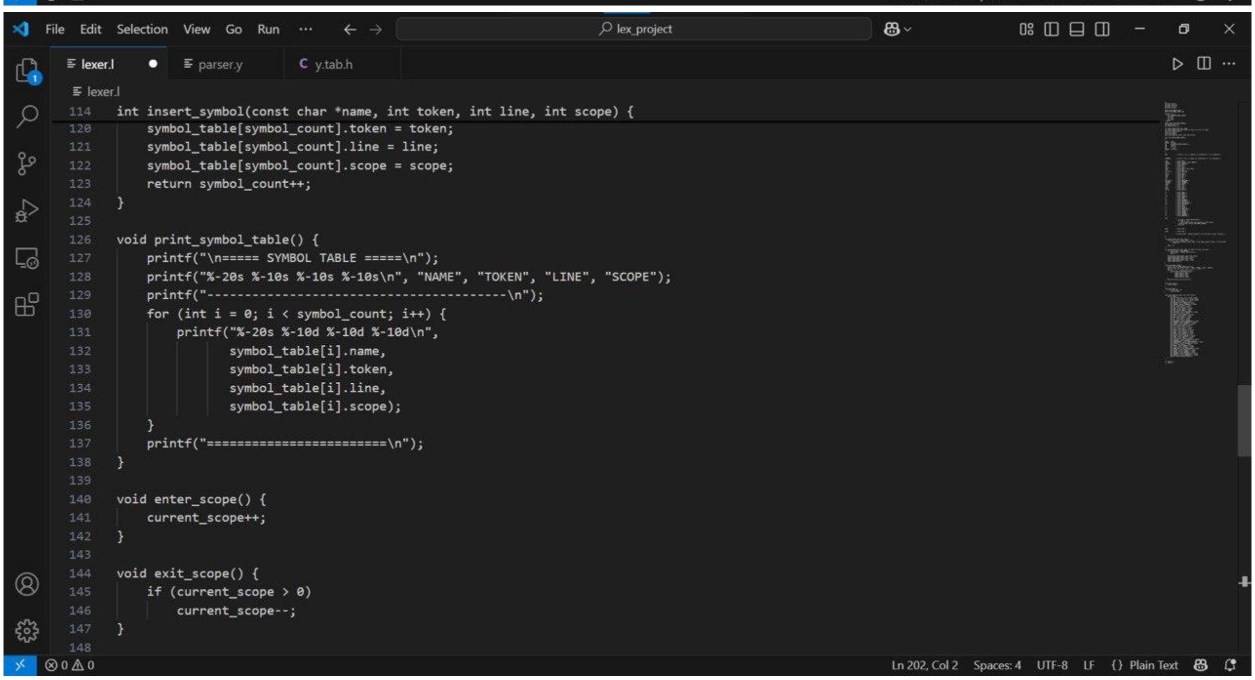


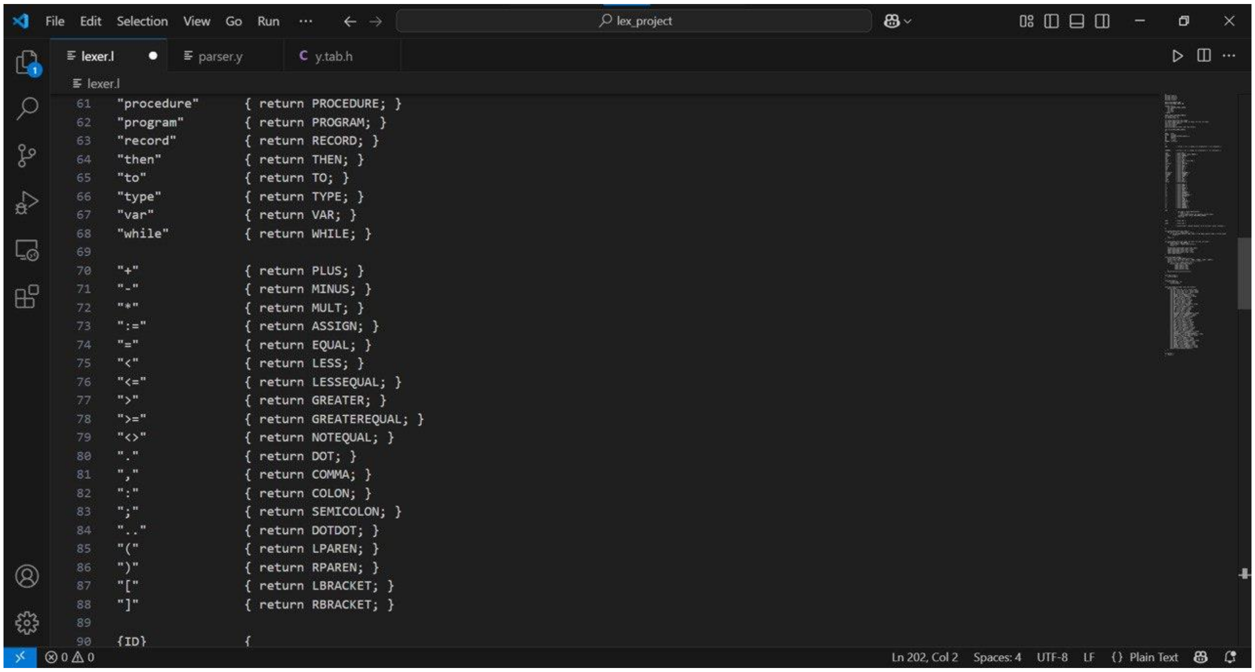
The code

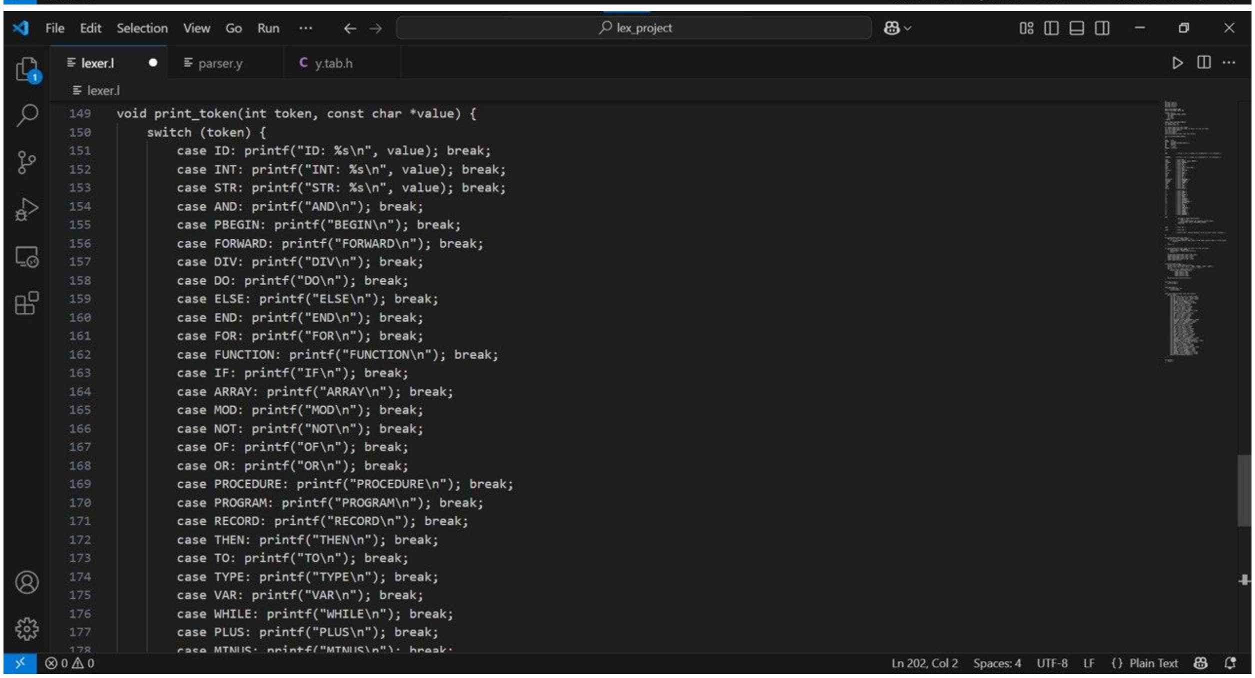
```
void print_token(int token, const char *value) {
    switch (token) {
        case PLUS: printf("PLUS\n"); break;
        case MINUS: printf("MINUS\n"); break;
        case MULT: printf("MULT\n"); break;
        case ASSIGN: printf("ASSIGN\n"); break;
        case EQUAL: printf("EQUAL\n"); break;
        case LESS: printf("LESS\n"); break;
        case LESSEQUAL: printf("LESSEQUAL\n"); break;
        case GREATER: printf("GREATER\n"); break;
        case GREATEREQUAL: printf("GREATEREQUAL\n"); break;
        case NOTEQUAL: printf("NOTEQUAL\n"); break;
        case DOT: printf("DOT\n"); break;
        case COMMA: printf("COMMA\n"); break;
        case COLON: printf("COLON\n"); break;
        case SEMICOLON: printf("SEMICOLON\n"); break;
        case DOTDOT: printf("DOTDOT\n"); break;
        case LPAREN: printf("LPAREN\n"); break;
        case RPAREN: printf("RPAREN\n"); break;
        case LBRACKET: printf("LBRACKET\n"); break;
        case RBRACKET: printf("RBRACKET\n"); break;
        default: printf("Unknown token\n");
    }
}

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

```
{ID}                {
                        int index = lookup_symbol(yytext);
                        if (index == -1)
                            insert_symbol(yytext, ID, yylineno, current_scope);
                        strncpy(id_text, yytext, MAX_SYMBOL_LENGTH);
                        return ID;
                    }

{INT}               { return INT; }

{STR}               { return STR; }

.                   { fprintf(stderr, "Unknown character: %s at line %d\n", yytext, yylineno); }

%%

int lookup_symbol(const char *name) {
    for (int i = 0; i < symbol_count; i++) {
        if (strcmp(symbol_table[i].name, name) == 0 && symbol_table[i].scope <= current_scope)
            return i;
    }
    return -1;
}

int insert_symbol(const char *name, int token, int line, int scope) {
    if (symbol_count >= MAX_SYMBOLS) {
        fprintf(stderr, "Symbol table full\n");
        return -1;
    }
    strcpy(symbol_table[symbol_count].name, name);
```


lexer.l
parser.y
y.tab.h
DIGIT   [0-9]
LETTER   [a-zA-Z]
ID   {LETTER}({LETTER}|{DIGIT}|_)*
INT   {DIGIT}+
STR   \"[^"]*\"
WS   [\t\n]+
COMMENT "{"[^}]*"}"
%%
{WS}   { for(int i = 0; i < yyleng; i++) if(yytext[i] == '\n') yylineno++; }
{COMMENT}   { for(int i = 0; i < yyleng; i++) if(yytext[i] == '\n') yylineno++; }
"and"   { return AND; }
"begin"   { enter_scope(); return PBEGIN; }
"forward"   { return FORWARD; }
"div"   { return DIV; }
"do"   { return DO; }
"else"   { return ELSE; }
"end"   { exit_scope(); return END; }
"for"   { return FOR; }
"function"   { return FUNCTION; }
"if"   { return IF; }
"array"   { return ARRAY; }
"mod"   { return MOD; }
"not"   { return NOT; }
"of"   { return OF; }
"or"   { return OR; }
Ln 202, Col 2 Spaces: 4 UTF-8 LF Plain Text


lexer.l
parser.y
y.tab.h
int insert_symbol(const char *name, int token, int line, int scope) {
    symbol_table[symbol_count].token = token;
    symbol_table[symbol_count].line = line;
    symbol_table[symbol_count].scope = scope;
    return symbol_count++;
}
void print_symbol_table() {
    printf("\n===== SYMBOL TABLE =====\n");
    printf("%-20s %-10s %-10s %-10s\n", "NAME", "TOKEN", "LINE", "SCOPE");
    printf("--\n");
    for (int i = 0; i < symbol_count; i++) {
        printf("%-20s %-10d %-10d %-10d\n",
               symbol_table[i].name,
               symbol_table[i].token,
               symbol_table[i].line,
               symbol_table[i].scope);
    }
    printf("======================\n");
}
void enter_scope() {
    current_scope++;
}
void exit_scope() {
    if (current_scope > 0)
        current_scope--;
}
Ln 202, Col 2 Spaces: 4 UTF-8 LF Plain Text


File Edit Selection View Go Run
lex_project
lexer.l
parser.y
y.tab.h
"procedure"   { return PROCEDURE; }
"program"   { return PROGRAM; }
"record"   { return RECORD; }
"then"   { return THEN; }
"to"   { return TO; }
"type"   { return TYPE; }
"var"   { return VAR; }
"while"   { return WHILE; }
"+"   { return PLUS; }
"-"   { return MINUS; }
"*"   { return MULT; }
":="   { return ASSIGN; }
"="   { return EQUAL; }
"<"   { return LESS; }
"<="   { return LESSEQUAL; }
">"   { return GREATER; }
">="   { return GREATEREQUAL; }
"<>"   { return NOTEQUAL; }
"."   { return DOT; }
","   { return COMMA; }
":"   { return COLON; }
";"   { return SEMICOLON; }
".."   { return DOTDOT; }
"("   { return LPAREN; }
")"   { return RPAREN; }
"["   { return LBRACKET; }
"]"   { return RBRACKET; }
{ID}   {
Ln 202, Col 2 Spaces: 4 UTF-8 LF Plain Text


File Edit Selection View Go Run
lex_project
lexer.l
parser.y
y.tab.h
void print_token(int token, const char *value) {
switch (token) {
case ID: printf("ID: %s\n", value); break;
case INT: printf("INT: %s\n", value); break;
case STR: printf("STR: %s\n", value); break;
case AND: printf("AND\n"); break;
case PBEGIN: printf("BEGIN\n"); break;
case FORWARD: printf("FORWARD\n"); break;
case DIV: printf("DIV\n"); break;
case DO: printf("DO\n"); break;
case ELSE: printf("ELSE\n"); break;
case END: printf("END\n"); break;
case FOR: printf("FOR\n"); break;
case FUNCTION: printf("FUNCTION\n"); break;
case IF: printf("IF\n"); break;
case ARRAY: printf("ARRAY\n"); break;
case MOD: printf("MOD\n"); break;
case NOT: printf("NOT\n"); break;
case OF: printf("OF\n"); break;
case OR: printf("OR\n"); break;
case PROCEDURE: printf("PROCEDURE\n"); break;
case PROGRAM: printf("PROGRAM\n"); break;
case RECORD: printf("RECORD\n"); break;
case THEN: printf("THEN\n"); break;
case TO: printf("TO\n"); break;
case TYPE: printf("TYPE\n"); break;
case VAR: printf("VAR\n"); break;
case WHILE: printf("WHILE\n"); break;
case PLUS: printf("PLUS\n"); break;
Ln 202, Col 2 Spaces: 4 UTF-8 LF Plain Text

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

#define MAX_SYMBOLS 1000
#define MAX_SYMBOL_LENGTH 100

typedef struct {
    char name[MAX_SYMBOL_LENGTH];
    int token;
    int line;
    int scope;
} Symbol;

Symbol symbol_table[MAX_SYMBOLS];
int symbol_count = 0;
int current_scope = 0;

int lookup_symbol(const char *name);
int insert_symbol(const char *name, int token, int line, int scope);
void print_symbol_table();
void enter_scope();
void exit_scope();
void print_token(int token, const char *value);

char id_text[MAX_SYMBOL_LENGTH];
%}
```

C: > Users > sarah > Downloads > Telegram Desktop > test.pas

```
program TestProgram(input, output);
begin
  x := 10;
  y := 20;
  if x + y > 25 then
    z := x + y;
end.
```

Syntax Analyzer

1) What is YACC?

It automatically generates the LALR(1) parsers from formal grammar specifications. YACC plays an important role in compiler and interpreter development since it provides a means to specify the grammar of a language and to produce parsers that either interpret or compile code written in that language.

2) How does YACC work?

YACC takes grammar rules written in a specific syntax and uses them to generate C source code for a parser. This parser can then be compiled into an executable program that reads input, checks if it follows the grammar, and performs actions based on the structure of the input.

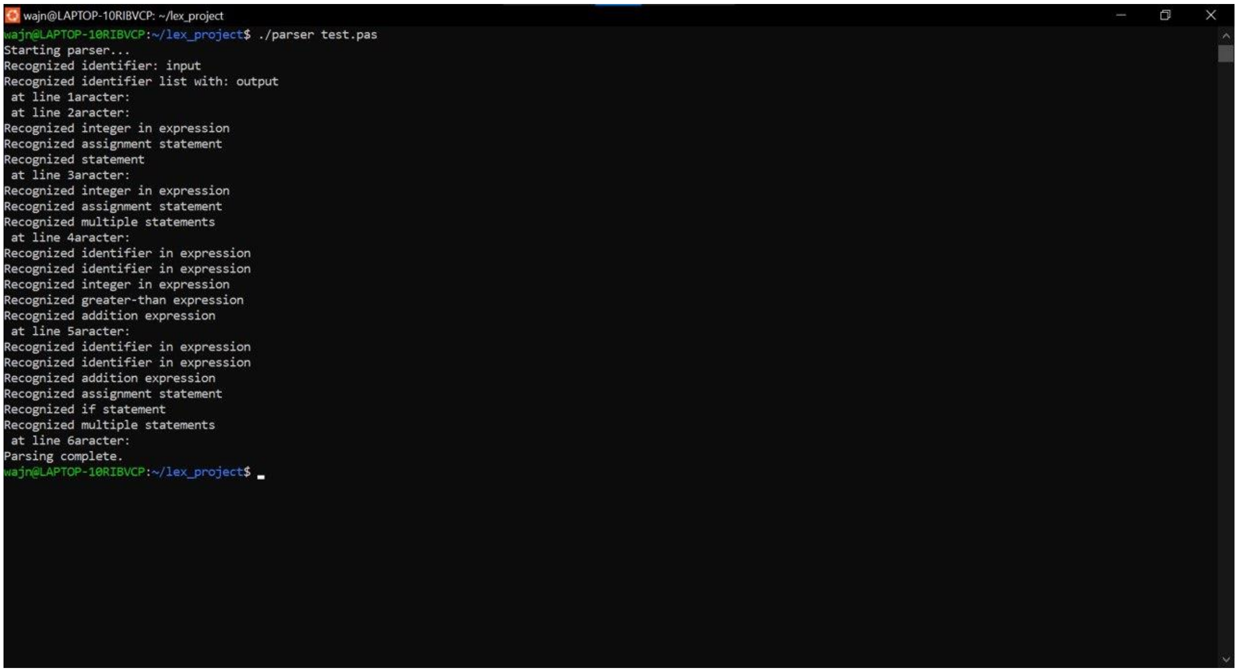
3) How does it integrate with LEX?

YACC works together with LEX by receiving the stream of tokens that LEX generates during lexical analysis. LEX scans the input text, recognizes patterns, and sends tokens to YACC through the yylex() function. After that, YACC uses its grammar rules to analyze the structure of these tokens and performs the appropriate actions. This collaboration allows both tools to function as the front-end of a compiler, where LEX focuses on tokenizing the input and YACC focuses on parsing it according to the language grammar.

4) What is the structure of a YACC program?

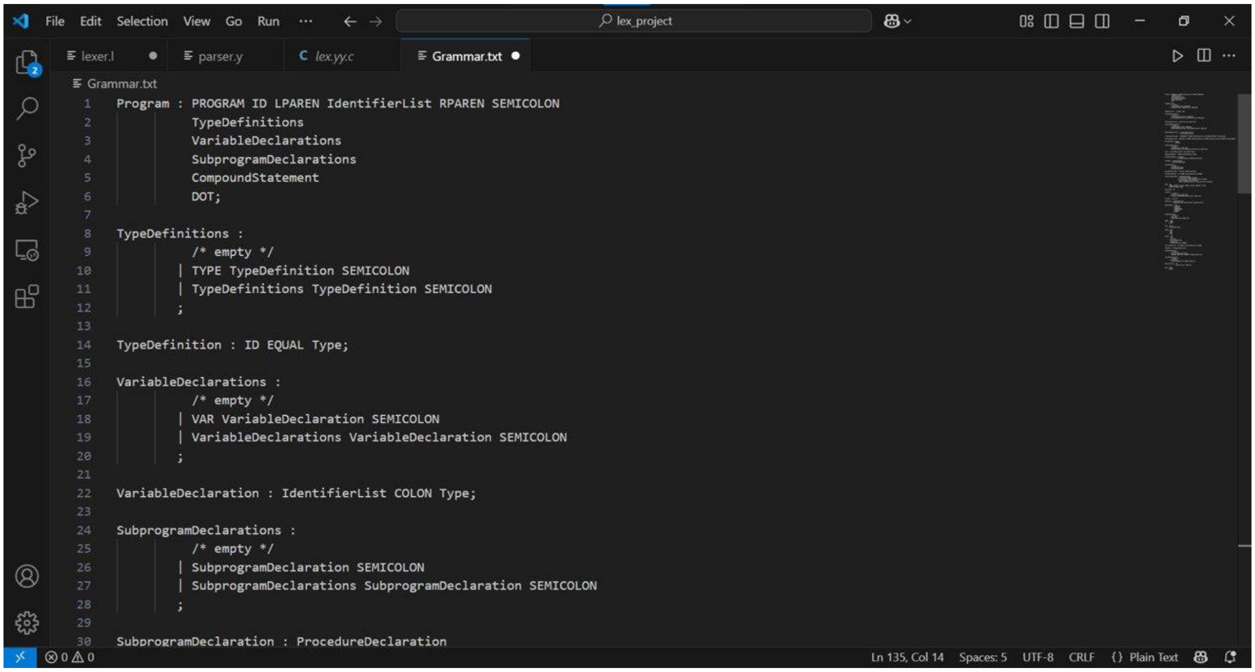
A YACC program is divided into three main sections, separated by %%. The first section is the **declarations section**, where tokens, data types, and header files are defined. The second section contains the **grammar rules**, where each rule defines a production and its associated C action code. The third section includes **auxiliary functions**, such as main() ,which support the parser's functionality. This structure allows YACC to generate a syntax analyzer based on the defined grammar.

The output\ Input

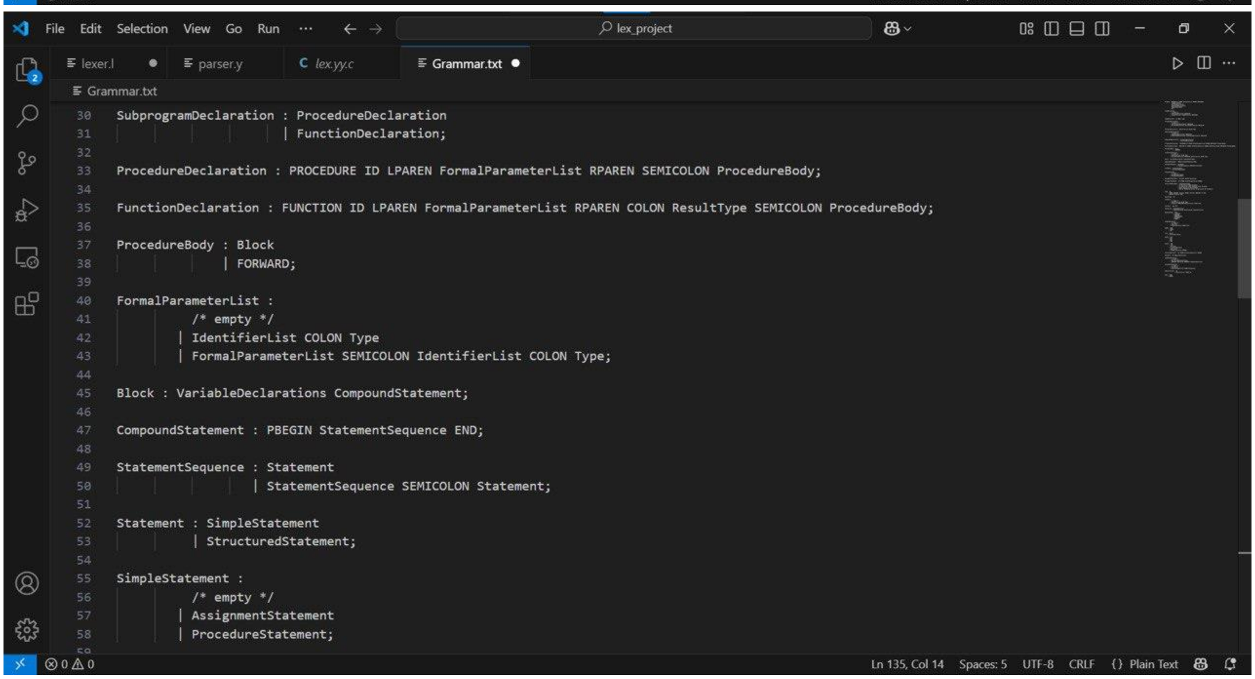


```
wajn@LAPTOP-10RIBVCP: ~/lex_project
wajn@LAPTOP-10RIBVCP:~/lex_project$ ./parser test.pas
Starting parser...
Recognized identifier: input
Recognized identifier list with: output
 at line 1aracter:
 at line 2aracter:
Recognized integer in expression
Recognized assignment statement
Recognized statement
 at line 3aracter:
Recognized integer in expression
Recognized assignment statement
Recognized multiple statements
 at line 4aracter:
Recognized identifier in expression
Recognized identifier in expression
Recognized integer in expression
Recognized greater-than expression
Recognized addition expression
 at line 5aracter:
Recognized identifier in expression
Recognized identifier in expression
Recognized addition expression
Recognized assignment statement
Recognized if statement
Recognized multiple statements
 at line 6aracter:
Parsing complete.
wajn@LAPTOP-10RIBVCP:~/lex_project$
```

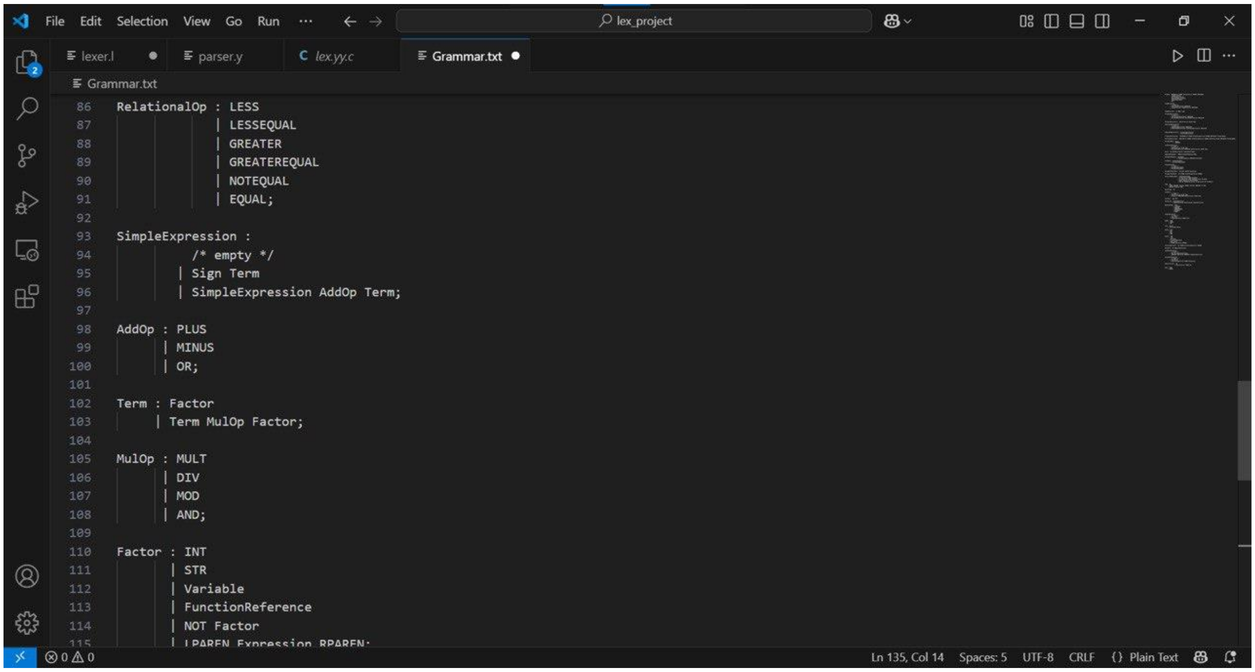
The Grammer code



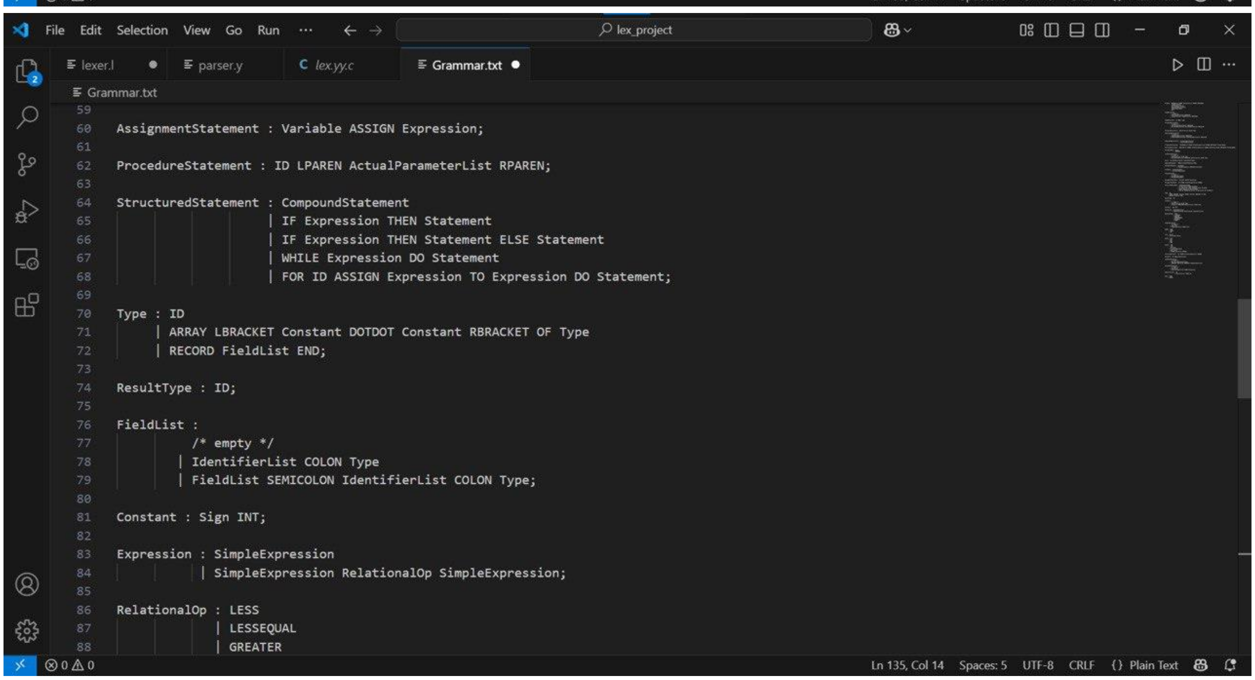
```
Program : PROGRAM ID LPAREN IdentifierList RPAREN SEMICOLON
          TypeDefinitions
          VariableDeclarations
          SubprogramDeclarations
          CompoundStatement
          DOT;

TypeDefinitions :
          /* empty */
        | TYPE TypeDefinition SEMICOLON
        | TypeDefinitions TypeDefinition SEMICOLON
        ;

TypeDefinition : ID EQUAL Type;

VariableDeclarations :
          /* empty */
        | VAR VariableDeclaration SEMICOLON
        | VariableDeclarations VariableDeclaration SEMICOLON
        ;

VariableDeclaration : IdentifierList COLON Type;

SubprogramDeclarations :
          /* empty */
        | SubprogramDeclaration SEMICOLON
        | SubprogramDeclarations SubprogramDeclaration SEMICOLON
        ;

SubprogramDeclaration : ProcedureDeclaration
```



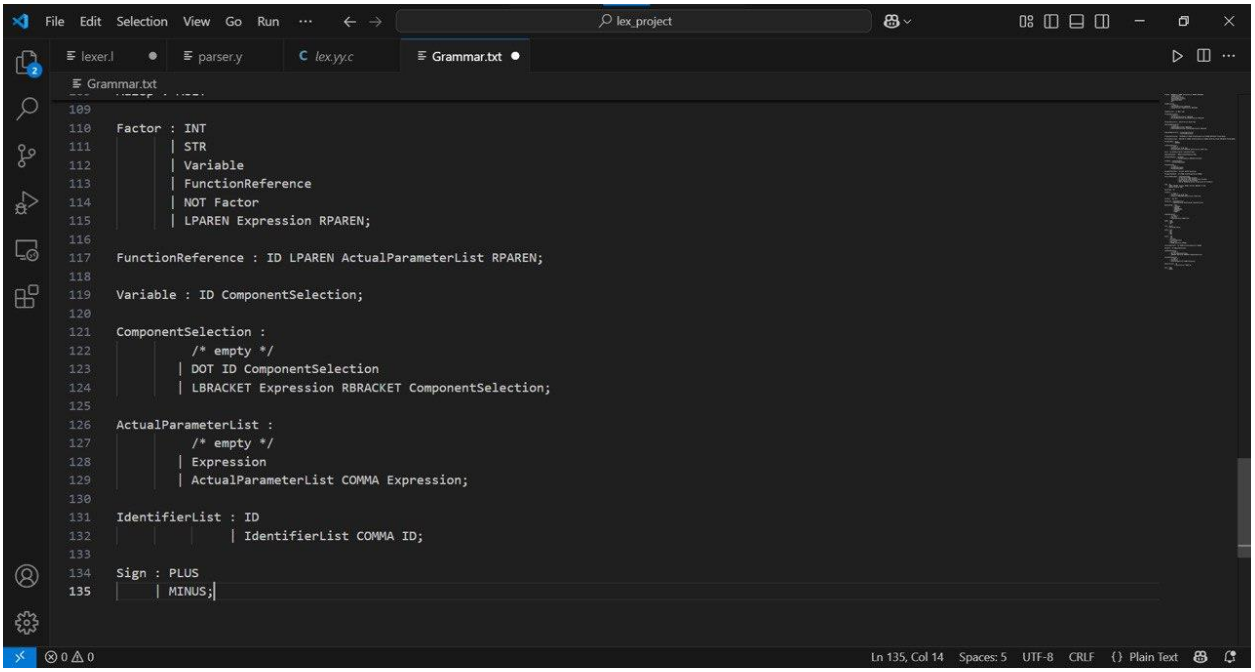
```
SubprogramDeclaration : ProcedureDeclaration
                      | FunctionDeclaration;

ProcedureDeclaration : PROCEDURE ID LPAREN FormalParameterList RPAREN SEMICOLON ProcedureBody;

FunctionDeclaration : FUNCTION ID LPAREN FormalParameterList RPAREN COLON ResultType SEMICOLON ProcedureBody;

ProcedureBody : Block
              | FORWARD;

FormalParameterList :
          /* empty */
        | IdentifierList COLON Type
        | FormalParameterList SEMICOLON IdentifierList COLON Type;

Block : VariableDeclarations CompoundStatement;

CompoundStatement : PBEGIN StatementSequence END;

StatementSequence : Statement
                  | StatementSequence SEMICOLON Statement;

Statement : SimpleStatement
          | StructuredStatement;

SimpleStatement :
          /* empty */
        | AssignmentStatement
        | ProcedureStatement;
```



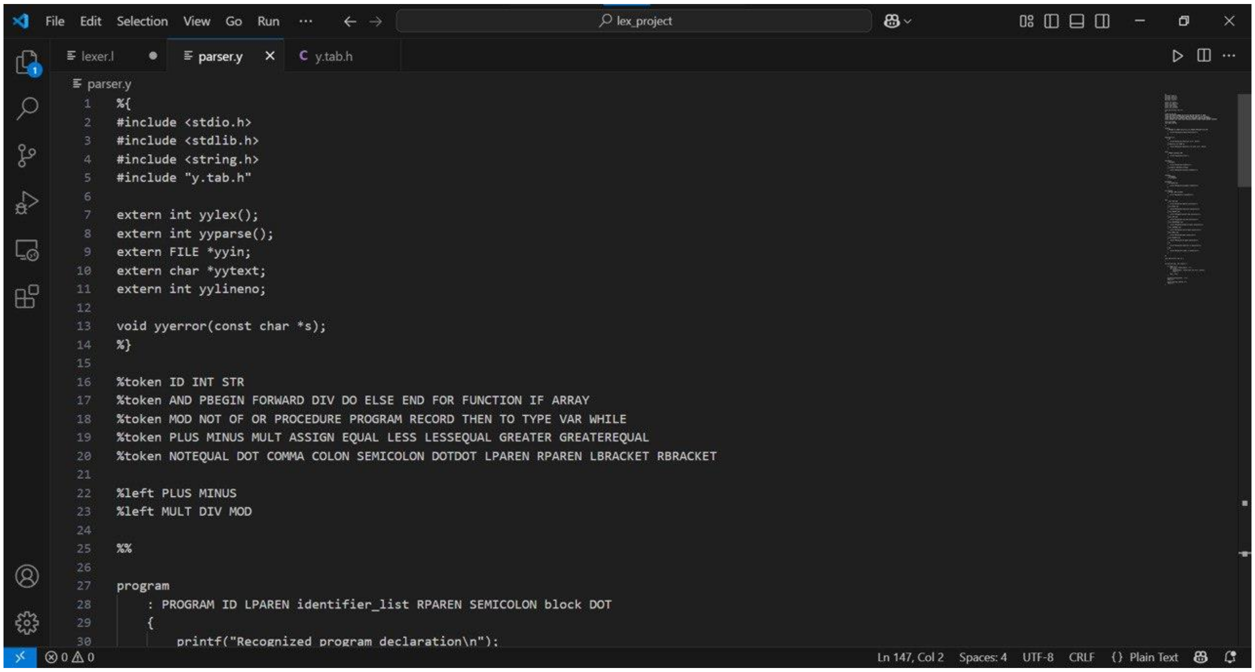
```
RelationalOp : LESS
             | LESSEQUAL
             | GREATER
             | GREATEREQUAL
             | NOTEQUAL
             | EQUAL;

SimpleExpression :
          /* empty */
        | Sign Term
        | SimpleExpression AddOp Term;

AddOp : PLUS
      | MINUS
      | OR;

Term : Factor
     | Term MulOp Factor;

MulOp : MULT
      | DIV
      | MOD
      | AND;

Factor : INT
       | STR
       | Variable
       | FunctionReference
       | NOT Factor
       | LPAREN Expression RPAREN;
```



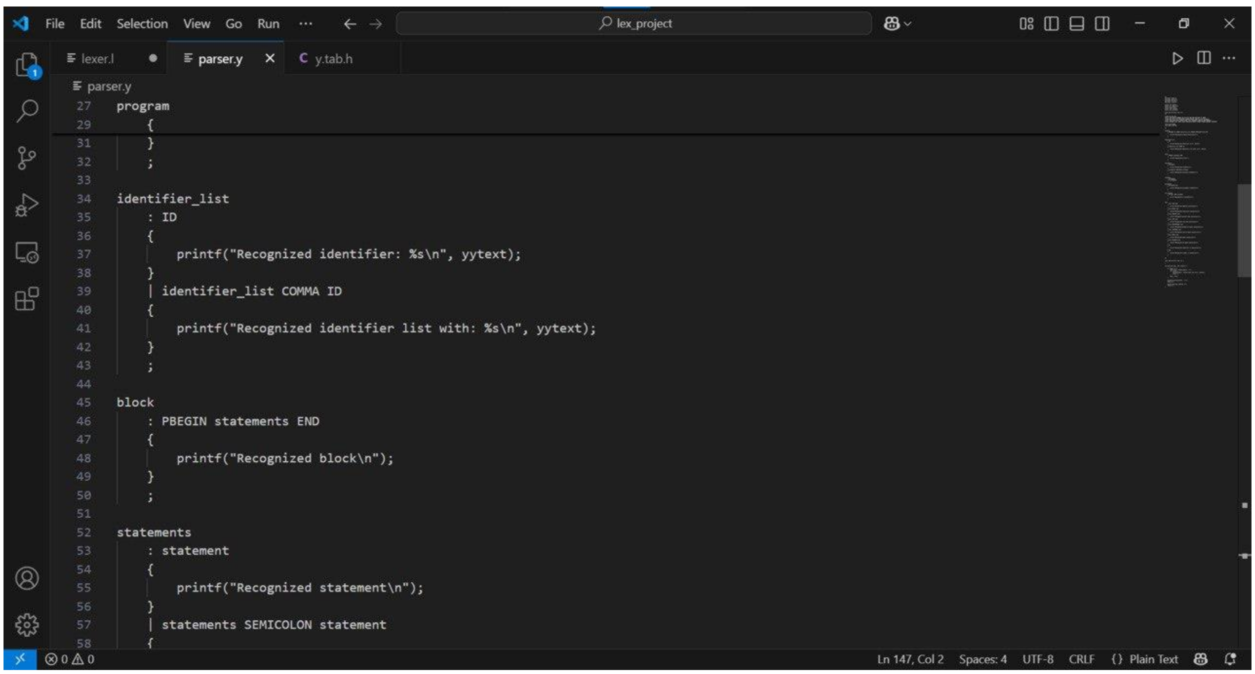
```

AssignmentStatement : Variable ASSIGN Expression;

ProcedureStatement : ID LPAREN ActualParameterList RPAREN;

StructuredStatement : CompoundStatement
                    | IF Expression THEN Statement
                    | IF Expression THEN Statement ELSE Statement
                    | WHILE Expression DO Statement
                    | FOR ID ASSIGN Expression TO Expression DO Statement;

Type : ID
     | ARRAY LBRACKET Constant DOTDOT Constant RBRACKET OF Type
     | RECORD FieldList END;

ResultType : ID;

FieldList :
          /* empty */
        | IdentifierList COLON Type
        | FieldList SEMICOLON IdentifierList COLON Type;

Constant : Sign INT;

Expression : SimpleExpression
           | SimpleExpression RelationalOp SimpleExpression;

RelationalOp : LESS
             | LESSEQUAL
             | GREATER
```


File Edit Selection View Go Run ···
lex_project
lexer.l
parser.y
lex.yy.c
Grammar.txt
Grammar.txt
Factor : INT
| STR
| Variable
| FunctionReference
| NOT Factor
| LPAREN Expression RPAREN;
FunctionReference : ID LPAREN ActualParameterList RPAREN;
Variable : ID ComponentSelection;
ComponentSelection :
/* empty */
| DOT ID ComponentSelection
| LBRACKET Expression RBRACKET ComponentSelection;
ActualParameterList :
/* empty */
| Expression
| ActualParameterList COMMA Expression;
IdentifierList : ID
| IdentifierList COMMA ID;
Sign : PLUS
| MINUS;
Ln 135, Col 14 Spaces: 5 UTF-8 CRLF {} Plain Text

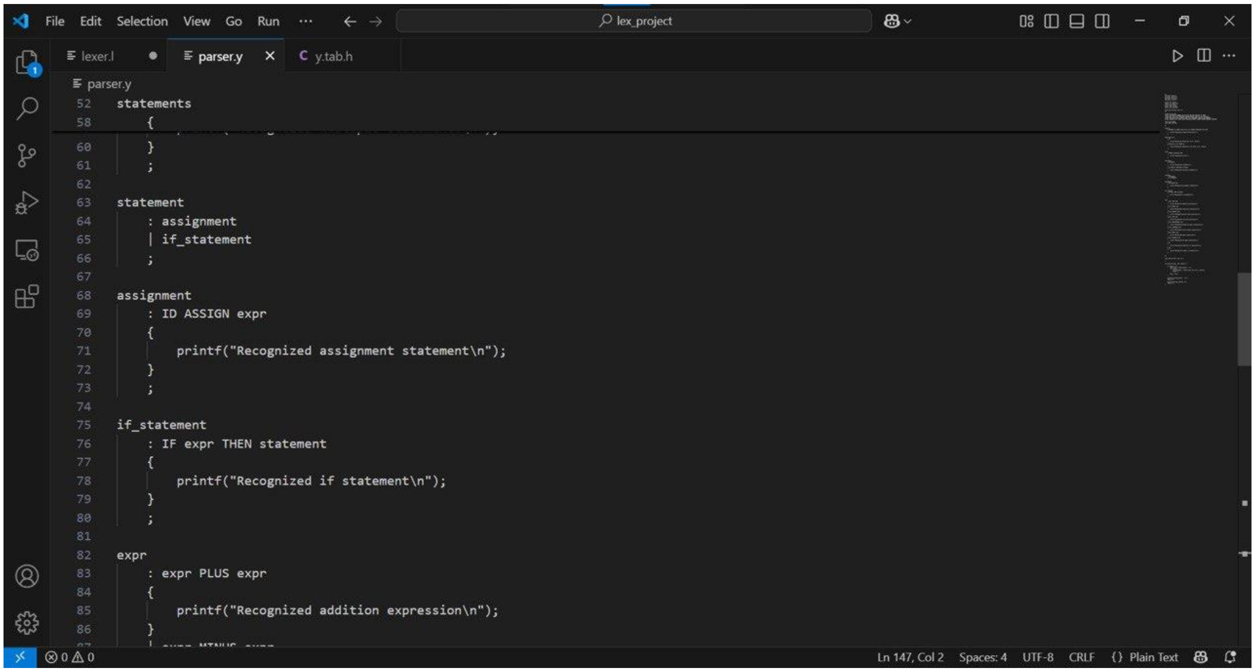
The Parser code

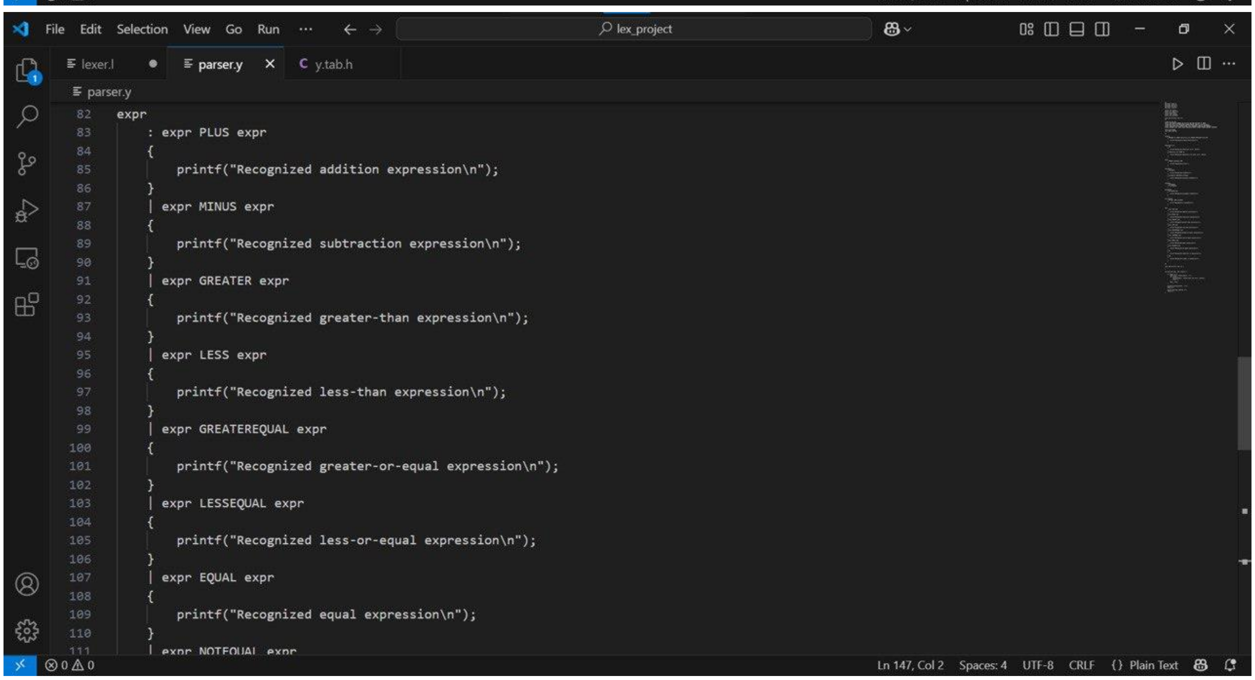


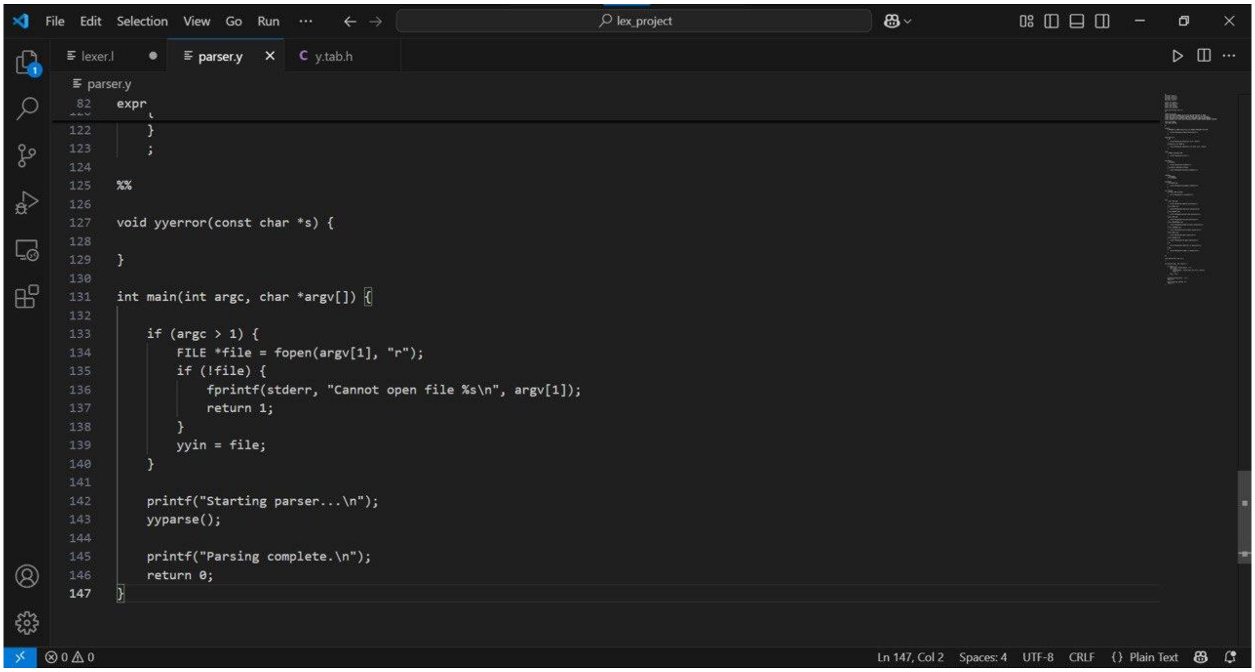
```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "y.tab.h"

extern int yylex();
extern int yyparse();
extern FILE *yyin;
extern char *yytext;
extern int yylineno;

void yyerror(const char *s);
%}

%token ID INT STR
%token AND PBEGIN FORWARD DIV DO ELSE END FOR FUNCTION IF ARRAY
%token MOD NOT OF OR PROCEDURE PROGRAM RECORD THEN TO TYPE VAR WHILE
%token PLUS MINUS MULT ASSIGN EQUAL LESS LESSEQUAL GREATER GREATEREQUAL
%token NOTEQUAL DOT COMMA COLON SEMICOLON DOTDOT LPAREN RPAREN LBRACKET RBRACKET

%left PLUS MINUS
%left MULT DIV MOD

%%

program
    : PROGRAM ID LPAREN identifier_list RPAREN SEMICOLON block DOT
    {
        printf("Recognized program declaration\n");
```



```
program
    {
    }
    ;

identifier_list
    : ID
    {
        printf("Recognized identifier: %s\n", yytext);
    }
    | identifier_list COMMA ID
    {
        printf("Recognized identifier list with: %s\n", yytext);
    }
    ;

block
    : PBEGIN statements END
    {
        printf("Recognized block\n");
    }
    ;

statements
    : statement
    {
        printf("Recognized statement\n");
    }
    | statements SEMICOLON statement
    {
```


File Edit Selection View Go Run
lex_project
lexer.l
parser.y
y.tab.h
parser.y
statements
{
}
;
statement
: assignment
| if_statement
;
assignment
: ID ASSIGN expr
{
printf("Recognized assignment statement\n");
}
;
if_statement
: IF expr THEN statement
{
printf("Recognized if statement\n");
}
;
expr
: expr PLUS expr
{
printf("Recognized addition expression\n");
}
Ln 147, Col 2 Spaces: 4 UTF-8 CRLF {} Plain Text


File Edit Selection View Go Run
lex_project
lexer.l
parser.y
y.tab.h
parser.y
expr
: expr PLUS expr
{
printf("Recognized addition expression\n");
}
| expr MINUS expr
{
printf("Recognized subtraction expression\n");
}
| expr GREATER expr
{
printf("Recognized greater-than expression\n");
}
| expr LESS expr
{
printf("Recognized less-than expression\n");
}
| expr GREATEREQUAL expr
{
printf("Recognized greater-or-equal expression\n");
}
| expr LESSEQUAL expr
{
printf("Recognized less-or-equal expression\n");
}
| expr EQUAL expr
{
printf("Recognized equal expression\n");
}
Ln 147, Col 2 Spaces: 4 UTF-8 CRLF {} Plain Text


File Edit Selection View Go Run
lex_project
lexer.l
parser.y
y.tab.h
parser.y
expr
}
;
%%
void yyerror(const char *s) {
}
int main(int argc, char *argv[]) {
if (argc > 1) {
FILE *file = fopen(argv[1], "r");
if (!file) {
fprintf(stderr, "Cannot open file %s\n", argv[1]);
return 1;
}
yyin = file;
}
printf("Starting parser...\n");
yyparse();
printf("Parsing complete.\n");
return 0;
}
Ln 147, Col 2 Spaces: 4 UTF-8 CRLF {} Plain Text

Test Cases Output

Test case 1:

```
1) Production rules heads:
Program
CompoundStatement
StatementSequence
AssignmentStatement

2) Symbol Table:
Name                Type
------------------ --------------------
x                   variable
--------------------------------------------------
```

Test case 2 :

```
1) Production rules heads:
Program
VariableDeclaration
FunctionDeclaration
CompoundStatement

2) Symbol Table:
Name                Type
------------------ --------------------
x                   variable
y                   variable
add                 function
--------------------------------------------------
```

Test case 3 :

```
1) Production rules heads:
Program
SubprogramDeclarations
ProcedureDeclaration
Block
CompoundStatement
StatementSequence
AssignmentStatement
CompoundStatement
StatementSequence
ProcedureStatement

2) Symbol Table:
Name                 Type
------------------ --------------------
test3                program
printSum             procedure
a                    parameter
b                    parameter
result               variable
--------------------------------------------------
```

Test case 4 :

```
1) Production rules heads:
Program
SubprogramDeclarations
FunctionDeclaration
Block
CompoundStatement
StatementSequence
AssignmentStatement
CompoundStatement
StatementSequence
AssignmentStatement

2) Symbol Table:
Name                 Type
------------------ --------------------
test4                program
square               function
n                    parameter
result               variable
--------------------------------------------------
```

How To Run The Code

LEXER:

mkdir lex_project

cd lex_project

code.

flex lexer.l

gcc lex.yy.c -o lexer.exe -lfl

gcc lex.yy.c -o lexer -lfl

./ lexer

——> input

Symbol Table:

cd lex_project

ls

(File name)

flex lexer.l

gcc lex.yy.c -o lexer -lfl

./lexer

touch test.pas

./ lexer < test.pas

——> input

Parser:

cd lex_project

ls

(File name)

yacc -d parser.y

flex lexer.l

gcc -o parser y.tab.c lex.yy.c -lfl

./parser test.pas

——> input

Summary

- Challenges:
- **General Debugging** : Recompiling frequently, tracing down ambiguous error s, and determining which part (lexer vs parser) was broken required a great deal of mental effort.
- **Disk Space Issues**: Errors like no space left on device wasted time debugging non-code-related problems.
- **Token Mismatch**: Token names in the lexer had to *exactly* match those declared in the parser. Which toke long to deal with.

References

Lex & Yacc book :https://www.naukri.com/code360/library/lexical-analysis-in-compiler-design

https://www.geeksforgeeks.org/introduction-to-yacc/https://www.geeksforgeeks.org/introduction-to-yacc/

https://norasandler.com/2017/11/29/Write-a-Compiler.html

https://austinhenley.com/blog/teenytinycompiler1.html

https://cse.iitkgp.ac.in/~bivasm/notes/LexAndYaccTutorial.pdf

https://www.ibm.com/sa-ar

note: Throughout the research and writing process, we would like to appreciate the usage of OpenAI's ChatGPT as an additional tool for **understanding** course material, and analyzing difficult subjects.