Umm Al-Qura University
Computers
College

Department of
Computer Science & AI 1446
3rd Semester

# Project Report For a course Compilers Construction

## ID for group: G1-05

## Students' names & ID:

445001892 - ولاء موسى حمدي
445005420 - رنا عبد الرحمن النفيعي
445005055 - الشيهانه محمد القثامي
444001392 - مها إبراهيم المطرفي
445004008 - ولاء عبدالكريم الزهراني

## Supervisd by

Dr.Daren Fadolalkarim

# Introduction:

This Compiler Construction project provides students with experience in understanding the components of a compiler using standard compiler tools: LEX and YACC. It focuses on learning how to implement a lexical analyzer and build a syntax analyzer, helping students gain a clearer understanding of how a compiler works.
And Students will gain a deeper comprehension of regular expressions, grammar rules, and lexical and syntactic analysis, all important ideas in compiler construction.
In this project:The members of the group will research YACC and Lex and respond to the questions. After that, they will construct the code for both parts and use test cases to evaluate the results. Lastly, this report will contain all of the work completed on the project.

# Lexical Analyzer(LEX):

## 1) What is LEX?

LEX is a tool used to generate a lexical a
nalyzer, which is the part of a compiler responsible for reading the source code and breaking it into small meaningful units called tokens, such as keywords, numbers, variable names, and symbols.
In simple terms, LEX helps recognize the words in the code before analyzing their structure.

## 2) How does LEX work? LEX works by analyzing the text

character by character and dividing it into tokens using regular expressions.
When it finds a match with a pattern, it executes the appropriate code for that token.
In short: LEX is responsible for recognizing the form of words in the program, such as numbers, names (identifiers), and symbols

## 3) What is the syntax of LEX code?

It consists of three parts: a part for definitions, a part for the token and what is printed when the original code matches the token, and the C code that will execute these commands.
This is the structure:

```
{%
Defines
%}
%%
Pattrens
%%
C code
```

## 4) Can you use an IDE for LEX development?

Yes you can use IDE to write a LEX code, you can use any text editor. Such as: Visual Studio Code, Eclipse, and IntelliJ IDEA. Using IDE to write a LEX code will make the process much faster and easier, With all the efficiency it provides

# The code for Lexical Analyzer(Lex):

lexerR.l

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

    extern int token_count;
    extern int symbol_count;
    extern char* symbol_table[];
    #define MAX_SYMBOLS 100

    void add_symbols(const char* word){
        for (int i =0 ; i < symbol_count; i++){
            if (strcmp(symbol_table[i],word)==0)
                return;
        }
        if (symbol_count < MAX_SYMBOLS){
            symbol_table[symbol_count] = strdup(word);
            symbol_count++;
        }
    }
%}

%%

"and"        { printf("KEYWORD: %s\n", yytext); token_count++; return AND; }
"begin"      { printf("KEYWORD: %s\n", yytext); token_count++; return BEGIN_T; }
"forward"    { printf("KEYWORD: %s\n", yytext); token_count++; return FORWARD; }
"div"        { printf("KEYWORD: %s\n", yytext); token_count++; return DIV; }
"do"         { printf("KEYWORD: %s\n", yytext); token_count++; return DO; }
"else"       { printf("KEYWORD: %s\n", yytext); token_count++; return ELSE; }
"end"        { printf("KEYWORD: %s\n", yytext); token_count++; return END; }
"for"        { printf("KEYWORD: %s\n", yytext); token_count++; return FOR; }
"function"   { printf("KEYWORD: %s\n", yytext); token_count++; return FUNCTION; }
"if"         { printf("KEYWORD: %s\n", yytext); token_count++; return IF; }
"array"      { printf("KEYWORD: %s\n", yytext); token_count++; return ARRAY; }
"mod"        { printf("KEYWORD: %s\n", yytext); token_count++; return MOD; }
"not"        { printf("KEYWORD: %s\n", yytext); token_count++; return NOT; }
"of"         { printf("KEYWORD: %s\n", yytext); token_count++; return OF; }
"or"         { printf("KEYWORD: %s\n", yytext); token_count++; return OR; }
"procedure"  { printf("KEYWORD: %s\n", yytext); token_count++; return PROCEDURE; }
"program"    { printf("KEYWORD: %s\n", yytext); token_count++; return PROGRAM; }
"record"     { printf("KEYWORD: %s\n", yytext); token_count++; return RECORD; }
"then"       { printf("KEYWORD: %s\n", yytext); token_count++; return THEN; }
"to"         { printf("KEYWORD: %s\n", yytext); token_count++; return TO; }
"type"       { printf("KEYWORD: %s\n", yytext); token_count++; return TYPE; }
"var"        { printf("KEYWORD: %s\n", yytext); token_count++; return VAR; }
"while"      { printf("KEYWORD: %s\n", yytext); token_count++; return WHILE; }

"+"          { printf("SYM: +\n"); token_count++; return PLUS; }
"-"          { printf("SYM: -\n"); token_count++; return MINUS; }
"*"          { printf("SYM: *\n"); token_count++; return MUL; }
"="          { printf("SYM: =\n"); token_count++; return EQUAL; }
```

```
"+"             { printf("SYM: +\n"); token_count++; return PLUS; }
"-"             { printf("SYM: -\n"); token_count++; return MINUS; }
"*"             { printf("SYM: *\n"); token_count++; return MUL; }
"="             { printf("SYM: =\n"); token_count++; return EQUAL; }
"<="            { printf("SYM: <=\n"); token_count++; return LTE; }
">="            { printf("SYM: >=\n"); token_count++; return GTE; }
"<>"            { printf("SYM: <>\n"); token_count++; return NEQ; }
">"             { printf("SYM: >\n"); token_count++; return GT; }
"<"             { printf("SYM: <\n"); token_count++; return LT; }
"."             { printf("SYM: .\n"); token_count++; return DOT; }
","             { printf("SYM: ,\n"); token_count++; return COMMA; }
":"             { printf("SYM: :\n"); token_count++; return COLON; }
";"             { printf("SYM: ;\n"); token_count++; return SEMICOLON; }
":="            { printf("SYM: :=\n"); token_count++; return ASSIGN; }
".."            { printf("SYM: ..\n"); token_count++; return DOTDOT; }
"("             { printf("SYM: (\n"); token_count++; return LPAREN; }
")"             { printf("SYM: )\n"); token_count++; return RPAREN; }
"["             { printf("SYM: [\n"); token_count++; return LBRACKET; }
"]"             { printf("SYM: ]\n"); token_count++; return RBRACKET; }

[a-zA-Z_][a-zA-Z0-9_]*   { printf("ID: %s\n", yytext); add_symbols(yytext); token_count++; return ID; }
[0-9]+                  { printf("INT: %s\n", yytext); token_count++; return INT; }
\"[^\"]*\"              { printf("STR: %s\n", yytext); token_count++; return STR; }
\{[^}]*\}               {/* ignore comments completely */ }
[ \t\n\r]+    { /* ignore whitespace completely */ }

. {
    if (yytext[0] != '\n' && yytext[0] != '\r' && yytext[0] != '\t' && yytext[0] != ' ')
        printf("UNKNOWN: %s\n", yytext);
    token_count++;
}

%%
int yywrap() {

    return 1;
}
```

# Syntax Analyzer(YACC):

## 1) What is YACC?

YACC is a tool used to generate a syntax parser, which takes the tokens provided by LEX and analyzes their structure according to the grammar rules of the programming language.
In simple terms, YACC checks whether the tokens are arranged correctly to form valid programming statements.

### 2) How does YACC work?

YACC is a tool used to build a syntax parser.
It operates using a set of rules, and each rule represents a certain structure in the program — such as an if statement or an arithmetic operation.
If the tokens received from LEX match a specific rule,
YACC executes the code associated with that rule.
In short: YACC ensures that the code is written in the correct syntactic structure or grammar

## 3) How does it integrate with LEX?

YACC was originally designed for being complemented by LEX. And it's commonly used with it. Since LEX is a compiler tool by itself, you can use it and YACC to create a parsing engine that processes text according to specific rules

## 4) What is the structure of a YACC program?

A YACC program consists of three sections: Declarations, Rules and Auxiliary functions. (Note the similarity with the structure of LEX programs).

DECLARATIONS

%%

RULES

%%

AUXILIARY FUNCTIONS

- Declaration section handles control information for yacc generated parser and generally sets up the execution environment in which the parser will operate.
- Rules section contains the rules(Grammar) for the parser.
- Auxiliary functions section contains a c-code copied as it is into generated c-program

# The code for Syntax Analyzer (YACC):

parserR.y

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

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

int token_count = 0;
int symbol_count = 0;
char* symbol_table[100];
%}

%token ID INT STR
%token PROGRAM VAR IF WHILE FOR BEGIN_T END THEN ELSE DO
%token FUNCTION PROCEDURE RECORD ARRAY TYPE OF TO NOT MOD DIV
%token FORWARD
%token AND OR
%token PLUS MINUS MUL EQUAL LT LTE GT GTE NEQ
%token DOT COMMA COLON SEMICOLON ASSIGN DOTDOT
%token LPAREN RPAREN LBRACKET RBRACKET

%left OR
%left AND
%left NOT
%left EQUAL NEQ LT LTE GT GTE
%left PLUS MINUS
%left MUL DIV MOD

%%

program
    : PROGRAM ID SEMICOLON type_definitions var_declarations subprog_declarations compound_stmt DOT
    { printf("program\n"); }
    ;

type_definitions
    :
    { printf("type_definitions\n"); }
    | TYPE type_definition SEMICOLON
    { printf("type_definitions\n"); }
    | type_definitions type_definition SEMICOLON
    { printf("type_definitions\n"); }
    ;

type_definition
    : ID EQUAL type
    { printf("type_definition\n"); }
    ;

type
```

```
type
    : standard_type
    { printf("type\n"); }
    | ARRAY LBRACKET INT DOTDOT INT RBRACKET OF standard_type
    { printf("type\n"); }
    | RECORD var_declarations END
    { printf("type\n"); }
    ;

standard_type
    : ID
    { printf("standard_type\n"); }
    ;

var_declarations
    :
    { printf("var_declarations\n"); }
    | VAR var_declaration SEMICOLON
    { printf("var_declarations\n"); }
    | var_declarations var_declaration SEMICOLON
    { printf("var_declarations\n"); }
    ;

var_declaration
    : identifier_list COLON type
    { printf("var_declaration\n"); }
    ;

identifier_list
    : ID
    { printf("identifier_list\n"); }
    | identifier_list COMMA ID
    { printf("identifier_list\n"); }
    ;

subprog_declarations
    :
    { printf("subprog_declarations\n"); }
    | subprog_declarations subprog_declaration SEMICOLON
    { printf("subprog_declarations\n"); }
    ;

subprog_declaration
    : procedure_decl
    { printf("subprog_declaration\n"); }
    | function_decl
    { printf("subprog_declaration\n"); }
```

```
subprog_declaration
    : procedure_decl
    { printf("subprog_declaration\n"); }
    | function_decl
    { printf("subprog_declaration\n"); }
    ;

procedure_decl
    : PROCEDURE ID LPAREN formal_params RPAREN SEMICOLON proc_body
    { printf("procedure_decl\n"); }
    ;

function_decl
    : FUNCTION ID LPAREN formal_params RPAREN COLON ID SEMICOLON proc_body
    { printf("function_decl\n"); }
    ;

formal_params
    :
    { printf("formal_params\n"); }
    | param_list
    { printf("formal_params\n"); }
    ;

param_list
    : param
    { printf("param_list\n"); }
    | param_list SEMICOLON param
    { printf("param_list\n"); }
    ;

param
    : identifier_list COLON type
    { printf("param\n"); }
    ;

proc_body
    : FORWARD
    { printf("proc_body\n"); }
    | block
    { printf("proc_body\n"); }
    ;

block
    : var_declarations compound_stmt
    { printf("block\n"); }
    ;

compound_stmt
    : BEGIN_T stmt_seq END
```

```
compound_stmt
    : BEGIN_T stmt_seq END
    { printf("compound_stmt\n"); }
    ;

stmt_seq
    : statement
    { printf("stmt_seq\n"); }
    | stmt_seq SEMICOLON statement
    { printf("stmt_seq\n"); }
    ;

statement
    : simple_stmt
    { printf("statement\n"); }
    | structured_stmt
    { printf("statement\n"); }
    ;

simple_stmt
    :
    { printf("simple_stmt\n"); }
    | assignment_stmt
    { printf("simple_stmt\n"); }
    | proc_stmt
    { printf("simple_stmt\n"); }
    ;

assignment_stmt
    : variable ASSIGN expression
    { printf("assignment_stmt\n"); }
    ;

proc_stmt
    : ID LPAREN actual_params RPAREN
    { printf("proc_stmt\n"); }
    ;

actual_params
    :
    { printf("actual_params\n"); }
    | expression
    { printf("actual_params\n"); }
    | actual_params COMMA expression
    { printf("actual_params\n"); }
    ;

structured_stmt
    : compound_stmt
    { printf("structured_stmt\n"); }
    | IF expression THEN statement opt_else
    { printf("structured_stmt\n"); }
```

```
        { printf("structured_stmt\n"); }
        | WHILE expression DO statement
        { printf("structured_stmt\n"); }
        | FOR ID ASSIGN expression TO expression DO statement
        { printf("structured_stmt\n"); }
        ;

opt_else
        :
        { printf("opt_else\n"); }
        | ELSE statement
        { printf("opt_else\n"); }
        ;

expression
        : simple_expr
        { printf("expression\n"); }
        | simple_expr relop simple_expr
        { printf("expression\n"); }
        ;

relop
        : LT  { printf("relop\n"); }
        | LTE { printf("relop\n"); }
        | GT  { printf("relop\n"); }
        | GTE { printf("relop\n"); }
        | EQUAL { printf("relop\n"); }
        | NEQ { printf("relop\n"); }
        ;

simple_expr
        : term
        { printf("simple_expr\n"); }
        | simple_expr addop term
        { printf("simple_expr\n"); }
        ;

addop
        : PLUS { printf("addop\n"); }
        | MINUS { printf("addop\n"); }
        | OR { printf("addop\n"); }
        ;
```

```
term
    : factor
    { printf("term\n"); }
    | term mulop factor
    { printf("term\n"); }
    ;

mulop
    : MUL { printf("mulop\n"); }
    | DIV { printf("mulop\n"); }
    | MOD { printf("mulop\n"); }
    | AND { printf("mulop\n"); }
    ;

factor
    : INT { printf("factor\n"); }
    | STR { printf("factor\n"); }
    | variable { printf("factor\n"); }
    | func_ref { printf("factor\n"); }
    | NOT factor { printf("factor\n"); }
    | LPAREN expression RPAREN { printf("factor\n"); }
    ;

variable
    : ID { printf("variable\n"); }
    ;

func_ref
    : ID LPAREN actual_params RPAREN { printf("func_ref\n"); }
    ;

%%

void yyerror(const char *s) {
    printf("Parse error: %s\n", s);
}
```

```c
    printf("Parse error: %s\n", s);
}

int main(int argc, char *argv[]) {
    if (argc > 1) {
        yyin = fopen(argv[1], "r");
        if (!yyin) {
            perror("Cannot open input file");
            return 1;
        }
    }

    printf("Parsing...\n\n");
    yyparse();
    printf("\n----- symbol_table -----\n");
    for (int i = 0 ; i < symbol_count; i++){
        printf("%d: %s\n", i + 1, symbol_table[i]);
    }
    printf("\nParsing complete.\n");
    printf("Total tokens: %d\n", token_count);
    return 0;
}
```

# The Output:

## Commands used for operation:

To set up and run the compiler, we first installed WSL (Windows Subsystem for Linux) by running wsl --install in the Windows Command Prompt. This installed the Ubuntu Linux environment on our system. Next, we installed the WSL extension in Visual Studio Code, which allowed us to access the Ubuntu terminal directly from the editor.

In the Ubuntu (WSL) terminal, we used the following commands:

- bison -d parserR.y: generates parser source code.
- flex lexerR.l: generates lexer source code.
- gcc lex.yy.c parserR.tab.c -o parser: compiles both files into an executable file called parser.
- ./parser < testfile1.txt: runs the parser on a test file that we generated.

# 1- test

testfile3.txt

```
program test1;
begin
 x := 5;
end.
```

# Output:

```
Parsing...

KEYWORD: program
ID: test1
SYM: ;
KEYWORD: begin
type_definitions
var_declarations
subprog_declarations
ID: x
SYM: :=
variable
INT: 5
factor
term
SYM: ;
simple_expr
expression
assignment_stmt
simple_stmt
statement
stmt_seq
KEYWORD: end
simple_stmt
statement
stmt_seq
compound_stmt
SYM: .
program

----- symbol_table -----
1: test1
2: x

Parsing complete.
Total tokens: 10
```

# 2- test

testfile3.txt

```
program test2;
var x, y: integer;
function add(a,b: integer): integer; forward;
begin
end.
```

# Output:

```
KEYWORD: program
ID: test2
SYM: ;
KEYWORD: var
type_definitions
ID: x
identifier_list
SYM: ,
ID: y
identifier_list
SYM: :
ID: integer
standard_type
type
var_declaration
SYM: ;
var_declarations
KEYWORD: function
subprog_declarations
ID: add
SYM: (
ID: a
identifier_list
SYM: ,
ID: b
identifier_list
SYM: :
ID: integer
standard_type
type
param
param_list
SYM: )
formal_params
SYM: :
ID: integer
SYM: ;
KEYWORD: forward
proc_body
function_decl
subprog_declaration
SYM: ;
subprog_declarations
KEYWORD: begin
KEYWORD: end
simple_stmt
statement
stmt_seq
compound_stmt
SYM: .
program

----- symbol_table -----
1: test2
2: x
3: y
```

```
----- symbol_table -----
1: test2
2: x
3: y
4: integer
5: add
6: a
7: b

Parsing complete.
Total tokens: 27
```

# 3- test

```
program test3;
procedure printSum (a, b: integer);
begin
  result := a+b;
end;

begin
  printSum(5, 6);
end.
```

# Output:

```
Parsing...

KEYWORD: program
ID: test3
SYM: ;
KEYWORD: procedure
type_definitions
var_declarations
subprog_declarations
ID: printSum
SYM: (
ID: a
identifier_list
SYM: ,
ID: b
identifier_list
SYM: :
ID: integer
standard_type
type
param
param_list
SYM: )
formal_params
SYM: ;
KEYWORD: begin
var_declarations
ID: result
SYM: :=
variable
ID: a
SYM: +
variable
factor
term
simple_expr
addop
ID: b
SYM: ;
variable
factor
term
simple_expr
expression
assignment_stmt
simple_stmt
statement
stmt_seq
KEYWORD: end
simple_stmt
statement
stmt_seq
```

```
expression
assignment_stmt
simple_stmt
statement
stmt_seq
KEYWORD: end
simple_stmt
statement
stmt_seq
compound_stmt
block
proc_body
procedure_decl
subprog_declaration
SYM: ;
subprog_declarations
KEYWORD: begin
ID: printSum
SYM: (
INT: 5
factor
term
SYM: ,
simple_expr
expression
actual_params
INT: 6
factor
term
SYM: )
simple_expr
expression
actual_params
proc_stmt
simple_stmt
statement
stmt_seq
SYM: ;
KEYWORD: end
simple_stmt
statement
stmt_seq
compound_stmt
compound_stmt
compound_stmt
compound_stmt
SYM: .
compound_stmt
compound_stmt
SYM: .
program
```

```
----- symbol_table -----
1: test3
2: printSum
3: a
4: b
5: integer
6: result

Parsing complete.
Total tokens: 32
```

# 4-test

```
program test4;
function square (n: integer): integer;
begin
  square := n * n;
end;
begin
  result := square (4);
end.
```

# Output:

```
Parsing...

KEYWORD: program
ID: test4
SYM: ;
KEYWORD: function
type_definitions
var_declarations
subprog_declarations
ID: square
SYM: (
ID: n
identifier_list
SYM: :
ID: integer
standard_type
type
param
param_list
SYM: )
formal_params
SYM: :
ID: integer
SYM: ;
KEYWORD: begin
var_declarations
ID: square
SYM: :=
variable
ID: n
SYM: *
variable
factor
term
mulop
ID: n
SYM: ;
variable
factor
term
simple_expr
expression
assignment_stmt
simple_stmt
statement
stmt_seq
KEYWORD: end
simple_stmt
statement
stmt_seq
compound_stmt
block
proc_body
```

```
subprog_declaration
SYM: ;
subprog_declarations
KEYWORD: begin
ID: result
SYM: :=
variable
ID: square
SYM: (
INT: 4
factor
term
SYM: )
simple_expr
expression
actual_params
func_ref
factor
term
SYM: ;
simple_expr
expression
assignment_stmt
simple_stmt
statement
stmt_seq
KEYWORD: end
simple_stmt
statement
stmt_seq
compound_stmt
SYM: .
program

----- symbol_table -----
1: test4
2: square
3: n
4: integer
5: result

Parsing complete.
Total tokens: 32
```

# Challenges:

- Several times, we encountered an issue where the grammar did not match the parser code. Although the final file made things simpler, it was still challenging to ensure that every word in the lexer and the parser matched precisely.

- Fixing errors took too long

- When we tried to run the program, the terminal was very heavy It required downloading many tools like gcc & flex,which caused the code not to run properly and caused many problems

- The variables are duplicated in both files but we used <extern>

- We faced several difficulties in find a simple and effective operating environment, in the end we found terminal ” WSL”

# Summary:

In this project, we used Lex and YACC to build a compiler that parses Pascal code. We tested the implementation using multiple Pascal test cases and ran it via the Ubuntu terminal (WSL). Although we encountered several challenges during development, we were able to resolve them. This project greatly improved our understanding of how compilers work.

# Member's contributions

| | |
|---|---|
| ولاء موسى حمدي | Symbol table & summary for the project & Search for Lex structure |
| الشيهانه محمد القثامي | Parser code implementation & Introduction to the project |
| رنا عبد الرحمن النفيعي | Lex code implemntation & run the code, test it & how we run the code. |
| ولاء الزهراني | Parser code implemntation& Search What mean each (Lex & YACC) |
| مها المطرفي | Parser code implemntation& Find out how each works (Lex & YACC) |

# Reference


- https://brainly.com/question/33173343
https://www.geeksforgeeks.org/introduction-to-yacc/
- Chat gpt, Help me understand Lex more, Solve the error problems while running the program, how to use the commands properly.
- https://youtu.be/Q_41sx5dcIo?si=AZ7W6-S61CrxQY_d
- https://youtu.be/54bo1qaHAfk?si=VCY4TWbrHSYaMja2
- Chat gpt "Help me understand the parser and the lexer and what their definitions mean" , Lectures of the Compiler Construction Course
- Structure of yacc program with example program
- YACC
- Lex program
- Lex program
- Lex program