**LAB 9**

**LINK TO GIT**: <https://github.com/radutalaviniaelena/FLCD>

**Statement: Use yacc**

You may use any version  (yacc or bison)

1. Write a specification file containing the production rules corresponding to the language specification (use syntax rules from lab1).
2. Then, use the parser generator (no errors)

Deliverables: lang.y (yacc specification file)

**BONUS:**modify lex to return tokens and use yacc to return string of productions

**Content of scanner.lxi**:

%{

#include <stdio.h>

#include <string.h>

#include "parser.tab.h"

int no\_of\_lines = 0;

%}

%option noyywrap

%option caseless

DIGIT [0-9]

NZ\_DIGIT [1-9]

LETTER [a-zA-Z]

INTEGER\_CONSTANT [+-]?{NZ\_DIGIT}{DIGIT}\*|0

STRING\_CONSTANT \"({LETTER}|{DIGIT})\*\"

CHAR\_CONSTANT \'({DIGIT}|{LETTER})\'

IDENTIFIER "\_"{LETTER}({LETTER}|{DIGIT})\*

CONSTANT {INTEGER\_CONSTANT}|{STRING\_CONSTANT}|{CHAR\_CONSTANT}

%%

"read" { printf("%s - reserved word\n", yytext); return READ; }

"write" { printf("%s - reserved word\n", yytext); return WRITE; }

"if" { printf("%s - reserved word\n", yytext); return IF; }

"else" { printf("%s - reserved word\n", yytext); return ELSE; }

"while" { printf("%s - reserved word\n", yytext); return WHILE; }

"for" { printf("%s - reserved word\n", yytext); return FOR; }

"in" { printf("%s - reserved word\n", yytext); return IN; }

"range" { printf("%s - reserved word\n", yytext); return RANGE; }

"Integer" { printf("%s - reserved word\n", yytext); return INTEGER; }

"String" { printf("%s - reserved word\n", yytext); return STRING; }

"Char" { printf("%s - reserved word\n", yytext); return CHAR; }

"main" { printf("%s - reserved word\n", yytext); return MAIN; }

{IDENTIFIER} { printf("%s - identifier\n", yytext); return IDENTIFIER; }

{INTEGER\_CONSTANT} { printf("%s - int\_constant\n", yytext); return INT\_CONSTANT; }

{STRING\_CONSTANT} { printf("%s - string\_constant\n", yytext); return STRING\_CONSTANT; }

{CHAR\_CONSTANT} { printf("%s - char\_constant\n", yytext); return CHAR\_CONSTANT; }

"+" { printf("%s - operator\n", yytext); return PLUS; }

"-" { printf("%s - operator\n", yytext); return MINUS; }

"\*" { printf("%s - operator\n", yytext); return MULTIPLICATION; }

"/" { printf("%s - operator\n", yytext); return DIVISION; }

"%" { printf("%s - operator\n", yytext); return MODULO; }

"=" { printf("%s - operator\n", yytext); return ASSIGNMENT; }

">" { printf("%s - operator\n", yytext); return GT; }

">=" { printf("%s - operator\n", yytext); return GTE; }

"<" { printf("%s - operator\n", yytext); return LT; }

"<=" { printf("%s - operator\n", yytext); return LTE; }

"==" { printf("%s - operator\n", yytext); return EQ; }

"!=" { printf("%s - operator\n", yytext); return NOT\_EQ; }

">>" { printf("%s - separator\n", yytext); return READ\_SYMBOL; }

"<<" { printf("%s - separator\n", yytext); return WRITE\_SYMBOL; }

";" { printf("%s - separator\n", yytext); return SEMICOLON; }

":" { printf("%s - separator\n", yytext); return COLON; }

"(" { printf("%s - separator\n", yytext); return OPEN\_ROUND\_BRACKET; }

")" { printf("%s - separator\n", yytext); return CLOSED\_ROUND\_BRACKET; }

"[" { printf("%s - separator\n", yytext); return OPEN\_SQUARE\_BRACKET; }

"]" { printf("%s - separator\n", yytext); return CLOSED\_SQUARE\_BRACKET; }

"{" { printf("%s - separator\n", yytext); return OPEN\_CURLY\_BRACKET; }

"}" { printf("%s - separator\n", yytext); return CLOSED\_CURLY\_BRACKET; }

"," { printf("%s - separator\n", yytext); return COMMA; }

[ \t]+ {} /\* elimina spatii \*/

\n ++no\_of\_lines;

[+-]0 { printf("Illegal integer constant at line %d: a number cannot start with 0.\n", no\_of\_lines); return -1; }

0{DIGIT}\* { printf("Illegal integer constant at line %d: a number cannot start with 0.\n", no\_of\_lines); return -1; }

\'[^({DIGIT}|{LETTER})]\' { printf("Illegal char constant at line %d: a character should be a digit or a letter.\n", no\_of\_lines); return -1; }

\'({DIGIT}|{LETTER}) { printf("Illegal char constant at line %d: unclosed quotes.\n", no\_of\_lines); return -1; }

\"(({LETTER}|{DIGIT})\*[^({LETTER}|{DIGIT})]({LETTER}|{DIGIT})\*)\*\" { printf("Illegal string constant at line %d: a string should contain only digits and letters.\n", no\_of\_lines); return -1; }

\"({LETTER}|{DIGIT})\* { printf("Illegal string constant at line %d: unclosed quotes.\n", no\_of\_lines); return -1; }

. { printf("Illegal token at line %d.\n", no\_of\_lines); return -1; }

%%

**Content of parser.y**:

%{

#include <stdio.h>

#include <stdlib.h>

#define \_XOPEN\_SOURCE\_EXTENDED 1

#include <strings.h>

#define YYDEBUG 1

%}

%token MAIN

%token READ

%token WRITE

%token IF

%token ELSE

%token WHILE

%token FOR

%token IN

%token RANGE

%token INTEGER

%token STRING

%token CHAR

%token READ\_SYMBOL

%token WRITE\_SYMBOL

%token SEMICOLON

%token COLON

%token COMMA

%token OPEN\_ROUND\_BRACKET

%token CLOSED\_ROUND\_BRACKET

%token OPEN\_SQUARE\_BRACKET

%token CLOSED\_SQUARE\_BRACKET

%token OPEN\_CURLY\_BRACKET

%token CLOSED\_CURLY\_BRACKET

%token PLUS

%token MINUS

%token MULTIPLICATION

%token DIVISION

%token MODULO

%token ASSIGNMENT

%token GT

%token GTE

%token LT

%token LTE

%token EQ

%token NOT\_EQ

%token INT\_CONSTANT

%token STRING\_CONSTANT

%token CHAR\_CONSTANT

%token IDENTIFIER

%start program

%%

program : MAIN OPEN\_ROUND\_BRACKET CLOSED\_ROUND\_BRACKET OPEN\_CURLY\_BRACKET statement CLOSED\_CURLY\_BRACKET {printf("program -> main ( ) { statement }\n");}

;

statement : declaration\_statement {printf("statement -> declaration\_statement\n");}

| assignment\_statement {printf("statement -> assignment\_statement\n");}

| if\_statement {printf("statement -> if\_statement\n");}

| for\_statement {printf("statement -> for\_statement\n");}

| while\_statement {printf("statement -> while\_statement\n");}

| read\_statement {printf("statement -> read\_statement\n");}

| write\_statement {printf("statement -> write\_statement\n");}

| declaration\_statement statement {printf("statement -> declaration\_statement statement\n");}

| assignment\_statement statement {printf("statement -> assignment\_statement statement\n");}

| if\_statement statement {printf("statement -> if\_statement statement\n");}

| for\_statement statement {printf("statement -> for\_statement statement\n");}

| while\_statement statement {printf("statement -> while\_statement statement\n");}

| read\_statement statement {printf("statement -> read\_statement statement\n");}

| write\_statement statement {printf("statement -> write\_statement statement\n");}

;

declaration\_statement : variable\_declaration\_statement {printf("declaration\_statement -> variable\_declaration\_statement\n");}

| array\_declaration\_statement {printf("declaration\_statement -> array\_declaration\_statement\n");}

;

variable\_declaration\_statement : identifier\_list COLON type SEMICOLON {printf("variable\_declaration\_statement -> identifier\_list : type ;\n");}

| identifier\_list COLON type ASSIGNMENT expression SEMICOLON {printf("variable\_declaration\_statement -> identifier\_list : type = expression ;\n");}

;

array\_declaration\_statement : identifier\_list COLON type OPEN\_SQUARE\_BRACKET CLOSED\_SQUARE\_BRACKET SEMICOLON {printf("array\_declaration\_statement -> identifier\_list : type [ ] ;\n");}

;

identifier\_list : IDENTIFIER {printf("identifier\_list -> identifier\n");}

| IDENTIFIER COMMA identifier\_list {printf("identifier\_list -> identifier , identifier\_list\n");}

;

type : INTEGER {printf("type -> Integer\n");}

| STRING {printf("type -> String\n");}

| CHAR {printf("type -> Char\n");}

;

expression : int\_expression {printf("expression -> int\_expression\n");}

| string\_expression {printf("expression -> string\_expression\n");}

| char\_expression {printf("expression -> char\_expression\n");}

;

int\_expression : INT\_CONSTANT {printf("int\_expression -> constant\n");}

| INT\_CONSTANT PLUS int\_expression {printf("int\_expression -> constant + int\_expression\n");}

| INT\_CONSTANT MINUS int\_expression {printf("int\_expression -> constant - int\_expression\n");}

| INT\_CONSTANT MULTIPLICATION int\_expression {printf("int\_expression -> constant \* int\_expression\n");}

| INT\_CONSTANT DIVISION int\_expression {printf("int\_expression -> constant / int\_expression\n");}

| INT\_CONSTANT MODULO int\_expression {printf("int\_expression -> constant % int\_expression\n");}

| IDENTIFIER {printf("int\_expression -> identifier\n");}

| IDENTIFIER PLUS int\_expression {printf("int\_expression -> identifier + int\_expression\n");}

| IDENTIFIER MINUS int\_expression {printf("int\_expression -> identifier - int\_expression\n");}

| IDENTIFIER MULTIPLICATION int\_expression {printf("int\_expression -> identifier \* int\_expression\n");}

| IDENTIFIER DIVISION int\_expression {printf("int\_expression -> identifier / int\_expression\n");}

| IDENTIFIER MODULO int\_expression {printf("int\_expression -> identifier % int\_expression\n");}

;

string\_expression : STRING\_CONSTANT {printf("string\_expression -> constant\n");}

| IDENTIFIER {printf("string\_expression -> identifier\n");}

;

char\_expression : CHAR\_CONSTANT {printf("char\_expression -> constant\n");}

| IDENTIFIER {printf("char\_expression -> identifier\n");}

;

assignment\_statement : IDENTIFIER ASSIGNMENT IDENTIFIER SEMICOLON {printf("assignment\_statement -> identifier = identifier ;\n");}

| IDENTIFIER ASSIGNMENT expression SEMICOLON {printf("assignment\_statement -> identifier = expression ;\n");}

;

if\_statement : IF OPEN\_ROUND\_BRACKET condition CLOSED\_ROUND\_BRACKET OPEN\_CURLY\_BRACKET statement CLOSED\_CURLY\_BRACKET {printf("if\_statement -> if ( condition ) { statement }\n");}

| IF OPEN\_ROUND\_BRACKET condition CLOSED\_ROUND\_BRACKET OPEN\_CURLY\_BRACKET statement CLOSED\_CURLY\_BRACKET ELSE OPEN\_CURLY\_BRACKET statement CLOSED\_CURLY\_BRACKET {printf("if\_statement -> if ( condition ) { statement } else { statement }\n");}

;

condition : expression relation expression {printf("condition -> expression relation expression\n");}

;

relation : GT {printf("relation -> >\n");}

| GTE {printf("relation -> >=\n");}

| LT {printf("relation -> <\n");}

| LTE {printf("relation -> <=\n");}

| EQ {printf("relation -> ==\n");}

| NOT\_EQ {printf("relation -> !=\n");}

;

while\_statement : WHILE OPEN\_ROUND\_BRACKET condition CLOSED\_ROUND\_BRACKET OPEN\_CURLY\_BRACKET statement CLOSED\_CURLY\_BRACKET {printf("while\_statement -> while ( condition ) { statement }\n");}

;

for\_statement : FOR IDENTIFIER IN IDENTIFIER OPEN\_CURLY\_BRACKET statement CLOSED\_CURLY\_BRACKET {printf("for\_statement -> for identifier in identifier { statement }\n");}

| FOR IDENTIFIER IN RANGE OPEN\_ROUND\_BRACKET range\_list CLOSED\_ROUND\_BRACKET OPEN\_CURLY\_BRACKET statement CLOSED\_CURLY\_BRACKET {printf("for\_statement -> for identifier in range ( range\_list ) { statement }\n");}

;

range\_list : INT\_CONSTANT {printf("range\_list -> constant\n");}

| IDENTIFIER {printf("range\_list -> identifier\n");}

| INT\_CONSTANT COMMA INT\_CONSTANT {printf("range\_list -> constant , constant\n");}

| INT\_CONSTANT COMMA INT\_CONSTANT COMMA INT\_CONSTANT {printf("range\_list -> constant , constant , constant\n");}

;

read\_statement : READ read\_helper SEMICOLON {printf("read\_statement -> read read\_helper ;\n");}

;

read\_helper : READ\_SYMBOL IDENTIFIER {printf("read\_helper -> >> identifier\n");}

| READ\_SYMBOL IDENTIFIER OPEN\_SQUARE\_BRACKET IDENTIFIER CLOSED\_SQUARE\_BRACKET {printf("read\_helper -> >> identifier [ identifier ]\n");}

| READ\_SYMBOL IDENTIFIER read\_helper {printf("read\_helper -> >> identifier read\_helper\n");}

;

write\_statement : WRITE write\_helper SEMICOLON {printf("write\_statement -> write write\_helper ;\n");}

;

write\_helper : WRITE\_SYMBOL IDENTIFIER {printf("write\_helper -> << identifier\n");}

| WRITE\_SYMBOL IDENTIFIER write\_helper {printf("write\_helper -> << identifier write\_helper\n");}

| WRITE\_SYMBOL INT\_CONSTANT {printf("write\_helper -> << constant\n");}

| WRITE\_SYMBOL INT\_CONSTANT write\_helper {printf("write\_helper -> << constant write\_helper\n");}

| WRITE\_SYMBOL STRING\_CONSTANT {printf("write\_helper -> << constant\n");}

| WRITE\_SYMBOL STRING\_CONSTANT write\_helper {printf("write\_helper -> << constant write\_helper\n");}

| WRITE\_SYMBOL CHAR\_CONSTANT {printf("write\_helper -> << constant\n");}

| WRITE\_SYMBOL CHAR\_CONSTANT write\_helper {printf("write\_helper -> << constant write\_helper\n");}

;

%%

yyerror(char \*s)

{

printf("%s\n",s);

}

extern FILE \*yyin;

main(int argc, char \*\*argv)

{

if(argc>1) yyin : fopen(argv[1],"r");

if(argc>2 && !strcmp(argv[2],"-d")) yydebug: 1;

if(!yyparse()) fprintf(stderr, "\tO.K.\n");

}