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");
}
```