Github repo: https://github.com/915-Petruta-Razvan/LFTC_Labs

Documentation

lang.lxi

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

    int currentLine = 1;
%}

%option noyywrap
%option case-insensitive

DIGIT [0-9]
NON_ZERO_DIGIT [1-9]
UNSIGNED_INTEGER {NON_ZERO_DIGIT}{DIGIT}*
INT_CONSTANT [+-]?{UNSIGNED_INTEGER}|0
LETTER [a-zA-Z_]
OTHER_CHAR [ ?!,.]
STRING_CONSTANT (\"({LETTER}|{DIGIT}|{OTHER_CHAR})*\")
IDENTIFIER {LETTER}({LETTER}|{DIGIT})*
WRONG_IDENTIFIER {DIGIT}({LETTER}|{DIGIT})*{LETTER}

%%

"int" {return INT;}
"char" {return CHAR;}
"string" {return STRING;}
"vector" {return VECTOR;}
"if" {return IF;}
"else" {return ELSE;}
"while" {return WHILE;}
"read" {return READ;}
"write" {return WRITE;}

"+" {return PLUS;}
"-" {return MINUS;}
"*" {return TIMES;}
"/" {return DIV;}
"%" {return MOD;}

"==" {return EQUAL;}
```

```
"!=" {return NOT_EQUAL;}
"<" {return LESS_THAN;}
"<=" {return LESS_THAN_OR_EQUAL;}
">" {return GREATER_THAN;}
">=" {return GREATER_THAN_OR_EQUAL;}

"=" {return ASSIGN;}

"(" {return LEFT_PARENTHESIS;}
")" {return RIGHT_PARENTHESIS;}
"[" {return LEFT_BRACKET;}
"]" {return RIGHT_BRACKET;}
"{" {return LEFT_BRACE;}
"}" {return RIGHT_BRACE;}
";" {return SEMICOLON;}

{IDENTIFIER} {return IDENTIFIER;}

{WRONG_IDENTIFIER} {return -1;}

{INT_CONSTANT} {return INT_CONSTANT;}

{STRING_CONSTANT} {return STRING_CONSTANT;}

[ \t]+ {}

[\n]+ {currentLine++;}

. {printf("Unknown token: %s at line %d\n", yytext,
currentLine); exit(1);}

%%
```

About

Modified the lang.lxi program to return tokens instead of just printing messages. These tokens will be used when defining the production rules in the parser code.

lang.y

```
%{
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    int yylex(void);
    int yyerror(char *s);
%}

%token INT;
%token CHAR;
%token STRING;
%token VECTOR;
%token IF;
%token ELSE;
%token WHILE;
%token READ;
%token WRITE;

%token PLUS;
%token MINUS;
%token TIMES;
%token DIV;
%token MOD;

%token EQUAL;
%token NOT_EQUAL;
%token LESS_THAN;
%token LESS_THAN_OR_EQUAL;
%token GREATER_THAN;
%token GREATER_THAN_OR_EQUAL;

%token ASSIGN;

%token LEFT_PARENTHESIS;
%token RIGHT_PARENTHESIS;
%token LEFT_BRACKET;
%token RIGHT_BRACKET;
%token LEFT_BRACE;
%token RIGHT_BRACE;
%token SEMICOLON;

%token IDENTIFIER;
%token INT_CONSTANT;
%token STRING_CONSTANT;

%%
program : statement_list {printf("program\n");}

statement_list : statement statement_list {printf("statement
statement_list\n");}
```

```
               | statement {printf("statement\n");}

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

simple_statement : declaration_statement
{printf("declaration_statement\n");}
                 | assignment_statement
{printf("assignment_statement\n");}
                 | io_statement {printf("io_statement\n");}

struct_statement : if_statement {printf("if_statement\n");}
                 | while_statement
{printf("while_statement\n");}

declaration_statement : type IDENTIFIER
{printf("declaration_statement\n");}

type : INT {printf("int\n");}
     | CHAR {printf("char\n");}
     | STRING {printf("string\n");}
     | VECTOR LEFT_BRACKET INT_CONSTANT RIGHT_BRACKET
{printf("vector[?]\n");}

assignment_statement : IDENTIFIER ASSIGN expression
{printf("assignment_statement\n");}
                     | IDENTIFIER LEFT_BRACKET IDENTIFIER
RIGHT_BRACKET ASSIGN expression {printf("vector[?]\n");}

io_statement : READ LEFT_PARENTHESIS IDENTIFIER
RIGHT_PARENTHESIS {printf("read(?)\n");}
             | WRITE LEFT_PARENTHESIS IDENTIFIER
RIGHT_PARENTHESIS {printf("write(?)\n");}
             | WRITE LEFT_PARENTHESIS STRING_CONSTANT
RIGHT_PARENTHESIS {printf("write(?)\n");}
             | WRITE LEFT_PARENTHESIS INT_CONSTANT
RIGHT_PARENTHESIS {printf("write(?)\n");}

if_statement : IF LEFT_PARENTHESIS condition
RIGHT_PARENTHESIS LEFT_BRACE statement_list RIGHT_BRACE
{printf("if_statement\n");}
             | IF LEFT_PARENTHESIS condition
RIGHT_PARENTHESIS LEFT_BRACE statement_list RIGHT_BRACE ELSE
LEFT_BRACE statement_list RIGHT_BRACE {printf("if_statement
else\n");}
```

```
while_statement : WHILE LEFT_PARENTHESIS condition
RIGHT_PARENTHESIS LEFT_BRACE statement_list RIGHT_BRACE
{printf("while_statement\n");}

expression : expression PLUS term {printf("expression +
term\n");}
           | expression MINUS term {printf("expression -
term\n");}
           | term {printf("term\n");}

term : term TIMES factor {printf("term * factor\n");}
     | term DIV factor {printf("term / factor\n");}
     | term MOD factor {printf("term % factor\n");}
     | factor {printf("factor\n");}

factor : LEFT_PARENTHESIS expression RIGHT_PARENTHESIS
{printf("(expression)\n");}
       | INT_CONSTANT {printf("int_constant\n");}
       | STRING_CONSTANT {printf("string_constant\n");}
       | IDENTIFIER {printf("identifier\n");}
       | IDENTIFIER LEFT_BRACKET IDENTIFIER RIGHT_BRACKET
{printf("vector[?]\n");}

condition : expression EQUAL expression {printf("expression
== expression\n");}
          | expression NOT_EQUAL expression
{printf("expression != expression\n");}
          | expression LESS_THAN expression
{printf("expression < expression\n");}
          | expression LESS_THAN_OR_EQUAL expression
{printf("expression <= expression\n");}
          | expression GREATER_THAN expression
{printf("expression > expression\n");}
          | expression GREATER_THAN_OR_EQUAL expression
{printf("expression >= expression\n");}
%%

extern FILE *yyin;

yyerror(char *s)
{
    printf("%s\n",s);
}

int main(int argc, char **argv)
{
    if (argc != 2)
    {
        printf("Usage: ./result.exe <input file>\n");
        exit(1);
    }

    FILE *fp = fopen(argv[1], "r");
    if (fp == NULL)
    {
        printf("Cannot open file %s\n", argv[1]);
        exit(1);
    }

    yyin = fp;
    yyparse();
    fclose(fp);

    return 0;
}
```

About

The parser is designed to recognize a variety of constructs typical in programming languages, such as variable declarations, assignments, control flow statements (if and while), and input/output operations.
Defined tokens include various data types (INT, CHAR, STRING, VECTOR), control flow keywords (IF, ELSE, WHILE), I/O keywords (READ, WRITE), operators (PLUS, MINUS, etc.), separators (parentheses, brackets, braces, semicolon), and value types (IDENTIFIER, INT_CONSTANT, STRING_CONSTANT).

Grammar rules

- **program**: The root rule, representing the entire program.
- **statement_list**: Handles sequences of statements.
- **statement**: Distinguishes between simple and structured statements.
- **simple_statement**: Covers declarations, assignments, and I/O operations.
- **struct_statement**: Includes control flow constructs like if and while statements.
- **declaration_statement**: For variable declarations.
- **type**: Defines types for variables, including support for vector types.
- **assignment_statement**: For assigning values to variables, including vector elements.
- **io_statement**: Handles read and write operations.
- **if_statement** and **while_statement**: Control flow constructs.
- **expression**, **term**, **factor**: For arithmetic and logical expressions.

- **condition**: Defines conditions for if and while statements.

Usage

– Generate the C code from the lex file:

```
lex lang.lxi
```

– Generate the C code from the yacc file:

```
yacc -d lang.y
```

– Get an executable from those 2 generated files:

```
gcc lex.yy.c y.tab.c -o result.exe
```

– Run the executable with a file:

```
./result.exe p1.txt
```