**COMMANDS:**

flex lang.lxi

bison -d lang.y

gcc lex.yy.c lang.tab.c -o parser.exe

./parser.exe<p1.txt

**Lang.y**

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define YYDEBUG 1

int production\_string[300];

int production\_string\_length = 0;

void addToProductionString(int production\_number) {

production\_string[production\_string\_length++] = production\_number;

}

void printProductionString() {

int index;

for(index = production\_string\_length - 1; index >= 0; index--){

printf("P%d ---> ", production\_string[index]);

}

printf("\n");

}

%}

%token READ

%token START

%token WRITE

%token IF

%token ELSE

%token FOR

%token WHILE

%token BREAK

%token INT

%token STRING

%token CHAR

%token LIST

%token RETURN

%token IDENTIFIER

%token CONSTANT

%token ATRIB

%token EQ

%token NE

%token LT

%token LE

%token GT

%token GE

%token ASIGN

%left ADD SUB

%left DIV MOD MUL

%left OR

%left AND

%left NOT

%token ADD

%token ADDEQ

%token SUB

%token SUBEQ

%token DIV

%token DIVEQ

%token MOD

%token MUL

%token OPEN\_CURLY\_BRACKET

%token CLOSED\_CURLY\_BRACKET

%token OPEN\_ROUND\_BRACKET

%token CLOSED\_ROUND\_BRACKET

%token OPEN\_RIGHT\_BRACKET

%token CLOSED\_RIGHT\_BRACKET

%token COMMA

%token SEMI\_COLON

%start program

%%

program : START compound\_statement {addToProductionString(1);}

;

compound\_statement : OPEN\_CURLY\_BRACKET statement\_list CLOSED\_CURLY\_BRACKET {addToProductionString(2);}

;

statement\_list : statement {addToProductionString(3);}

| statement statement\_list {addToProductionString(4);}

;

statement : simple\_statement {addToProductionString(5);}

| struct\_statement {addToProductionString(6);}

;

simple\_statement : assign\_statement {addToProductionString(7);}

| io\_statement {addToProductionString(8);}

| declaration {addToProductionString(9);}

;

struct\_statement : compound\_statement {addToProductionString(10);}

| if\_statement {addToProductionString(11);}

| while\_statement {addToProductionString(12);}

| for\_statement {addToProductionString(13);}

;

assign\_statement : IDENTIFIER ASIGN expression SEMI\_COLON {addToProductionString(14);}

| indexed\_identifier ASIGN expression SEMI\_COLON {addToProductionString(15);}

;

io\_statement : read\_statement {addToProductionString(16);}

| write\_statement {addToProductionString(17);}

;

read\_statement : READ OPEN\_ROUND\_BRACKET IDENTIFIER CLOSED\_ROUND\_BRACKET SEMI\_COLON {addToProductionString(18);}

| READ OPEN\_ROUND\_BRACKET indexed\_identifier CLOSED\_ROUND\_BRACKET SEMI\_COLON {addToProductionString(19);}

;

write\_statement : WRITE OPEN\_ROUND\_BRACKET id CLOSED\_ROUND\_BRACKET SEMI\_COLON {addToProductionString(20);}

;

if\_statement : IF OPEN\_ROUND\_BRACKET condition\_statement CLOSED\_ROUND\_BRACKET compound\_statement {addToProductionString(21);}

| IF OPEN\_ROUND\_BRACKET condition\_statement CLOSED\_ROUND\_BRACKET compound\_statement ELSE compound\_statement {addToProductionString(22);}

;

for\_statement : FOR OPEN\_ROUND\_BRACKET assign\_statement condition SEMI\_COLON assign\_statement CLOSED\_ROUND\_BRACKET compound\_statement {addToProductionString(23);}

;

while\_statement : WHILE OPEN\_ROUND\_BRACKET condition\_statement CLOSED\_ROUND\_BRACKET compound\_statement {addToProductionString(24);}

;

condition\_statement : condition {addToProductionString(25);}

| condition logical condition {addToProductionString(26);}

;

expression : CONSTANT {addToProductionString(27);}

| number\_expression {addToProductionString(28);}

;

number\_expression : CONSTANT {addToProductionString(29);}

| CONSTANT operator number\_expression {addToProductionString(30);}

| IDENTIFIER {addToProductionString(31);}

| IDENTIFIER operator number\_expression {addToProductionString(32);}

;

id : IDENTIFIER {addToProductionString(33);}

| CONSTANT {addToProductionString(34);}

| indexed\_identifier {addToProductionString(35);}

;

indexed\_identifier : IDENTIFIER OPEN\_RIGHT\_BRACKET INT CLOSED\_RIGHT\_BRACKET {addToProductionString(36);}

;

declaration : type IDENTIFIER SEMI\_COLON {addToProductionString(37);}

;

type : simple\_type {addToProductionString(38);}

| array\_declaration {addToProductionString(39);}

simple\_type : INT {addToProductionString(40);}

| STRING {addToProductionString(41);}

| CHAR {addToProductionString(42);}

;

array\_declaration : LIST LT simple\_type GT {addToProductionString(43);}

condition : expression relation expression {addToProductionString(44);}

;

relation : LT {addToProductionString(45);}

| LE {addToProductionString(46);}

| EQ {addToProductionString(47);}

| NE {addToProductionString(48);}

| GT {addToProductionString(49);}

| GE {addToProductionString(50);}

;

logical : AND {addToProductionString(51);}

| OR {addToProductionString(52);}

operator : ADD {addToProductionString(53);}

| MUL {addToProductionString(54);}

| MOD {addToProductionString(55);}

| SUB {addToProductionString(56);}

| DIV {addToProductionString(57);}

%%

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()) printProductionString();

}

**Lang.lxi**

%option noyywrap

%{

#include <stdio.h>

#include <string.h>

#include "lang.tab.h"

int lines = 0;

%}

DIGIT [0-9]

WORD \"[a-zA-Z0-9\_]\*\"

INTEGER [+-]?[1-9][0-9]\*

CHARACTER \'[a-zA-Z0-9\_]\'

CONSTANT {WORD}|{INTEGER}|{CHARACTER}|{DIGIT}

IDENIFIER [a-zA-Z][a-zA-Z0-9\_]\*

%%

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

print {printf( "Reserved word: %s\n", yytext); return WRITE;}

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

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

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

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

int {printf( "Reserved word: %s\n", yytext); return INT;}

char {printf( "Reserved word: %s\n", yytext); return CHAR;}

MAIN {printf( "Reserved word: %s\n", yytext); return START;}

{IDENIFIER} {printf( "Identifier: %s\n", yytext); return IDENTIFIER;}

{CONSTANT} {printf( "Constant: %s\n", yytext ); return CONSTANT;}

";" {printf( "Separator: %s\n", yytext ); return SEMI\_COLON;}

"," {printf( "Separator: %s\n", yytext ); return COMMA;}

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

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

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

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

"[" {printf( "Separator: %s\n", yytext ); return OPEN\_RIGHT\_BRACKET;}

"]" {printf( "Separator: %s\n", yytext ); return CLOSED\_RIGHT\_BRACKET;}

"+" {printf( "Operator: %s\n", yytext ); return ADD;}

"+=" {printf( "Operator: %s\n", yytext ); return ADDEQ;}

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

"-=" {printf( "Operator: %s\n", yytext ); return SUBEQ;}

"\*" {printf( "Operator: %s\n", yytext ); return MUL;}

"/" {printf( "Operator: %s\n", yytext ); return DIV;}

"/=" {printf( "Operator: %s\n", yytext ); return DIVEQ;}

"%" {printf( "Operator: %s\n", yytext ); return MOD;}

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

"<=" {printf( "Operator: %s\n", yytext ); return LE;}

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

">=" {printf( "Operator: %s\n", yytext ); return GE;}

"!=" {printf( "Operator: %s\n", yytext ); return NE;}

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

"=" {printf( "Separator: %s\n", yytext ); return ASIGN;}

[ \t]+ {}

[\r\n]+ {lines++;}

[+-]?0[0-9]\* {printf("Illegal integer at line %d\n", lines); return -1;}

[0-9]+[a-zA-Z\_]+[a-zA-Z0-9\_]\* {printf("Illegal identifier %d\n", lines); return -1;}

\'[a-zA-Z0-9]{2,}\' {printf("Character of length >=2 at line %d\n", lines); return -1;}

. {printf("Lexical error\n"); return -1;}

**p1.txt**

MAIN {  
int a;  
int b;  
int r;  
read(a);  
read(b);  
r = 20;  
while(r > 0) {  
 a = a + b;  
 b = b + r;  
 r = r - 1;  
}  
print(a);  
}