COMMANDS:

```
flex lang.lxi
bison -d lang.y
gcc lex.yy.c lang.tab.c -o result.exe
result<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 = 0; index < production_string_length; 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 SUB
%token DIV
%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}
IDENIFIER       [a-zA-Z][a-zA-Z0-9_]*

%%

START     {printf( "Reserved word: %s\n", yytext); return START;}
read      {printf( "Reserved word: %s\n", yytext); return READ;}
write     {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;}
break     {printf( "Reserved word: %s\n", yytext); return BREAK;}
int       {printf( "Reserved word: %s\n", yytext); return INT;}
string    {printf( "Reserved word: %s\n", yytext); return STRING;}
char      {printf( "Reserved word: %s\n", yytext); return CHAR;}
list      {printf( "Reserved word: %s\n", yytext); return LIST;}
```

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

{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 SUB;}
"*"             {printf( "Operator: %s\n", yytext ); return MUL;}
"/"             {printf( "Operator: %s\n", yytext ); return DIV;}
"%"             {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;}
"!"             {printf( "Operator: %s\n", yytext ); return NOT;}
"&&"     {printf( "Operator: %s\n", yytext ); return AND;}
"||"      {printf( "Operator: %s\n", yytext ); return OR;}

[ \t]+  {}
[\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

```
START {
   int ab;
   int ba;
   int c;
   int max;
```

```
    read(ab);
    read(ba);
    read(c);

    if(a>b&&a>c){
        max=a;
    }
    else{
        if(b>c&&b>a){
            max=b;
        }
        else{
            max=c;
        }
    }

    write(max);
}
```