CS39003 Compilers Laboratory
Lab Test 2
30-Oct-2025, 7:15pm–8:15pm
Maximum Marks: 50

Roll No: _____

Name: _____

[*Write your answers in the question paper itself. Be brief and precise. Write proper codes* (*not pseudocodes*).]

In this test again, you deal with polynomials. This time, a polynomial

$$f(x) = a_d x^d + a_{d-1} x^{d-1} + a_{d-2} x^{d-2} + \ldots + a_2 x^2 + a_1 x + a_0$$

is represented as a list $[\, a_d, a_{d-1}, a_{d-2}, \ldots, a_2, a_1, a_0 \,]$ from higher to lower degrees. The task is to write an interpreter that can evaluate a polynomial f at a point a, using Horner's formula given below.

$$\text{eval}(a, [\, a_d, a_{d-1}, a_{d-2}, \ldots, a_2, a_1, a_0 \,]) = (\ldots(((a_d \times a + a_{d-1}) \times a + a_{d-2}) \times a + a_{d-3}) \times a + \ldots + a_1) \times a + a_0$$

All the coefficients a_i and the evaluation point a are assumed to be (signed) integers, so the result of a polynomial evaluation is again an integer. We can recursively use polynomial evaluation for each coefficient a_i and/or for the evaluation point a. As an example, consider the following expression.

```
eval (
     eval (1,
          [2, 0, 0, 3, eval (-1, [1, 0, 0, eval(0, [3, 0, 2, 1])]), -7]),
     [ 1 , 2 , eval(2,[1,-2,3]), 4, 5, 6, 7 ]
)
```

First, [3, 0, 2, 1] is evaluated at 0; the result is 1. Second, [1, 0, 0, 1] (whose last coefficient comes from the first evaluation) is evaluated at –1, and the result is 0. The third evaluation is of [2, 0, 0, 3, 0, –7] at 1, and the result is –2. This is used for the last (fifth) evaluation of [1, 2, 3, 4, 5, 6, 7]. Before that, the coefficient 3 requires another (fourth) evaluation of [1, –2, 3] at 2. Your program should produce an output as follows. This printing should be in the same order as the evaluations are completed (illustrated in the example above).

```
eval(0,[3,0,2,1]) = 1
eval(-1,[1,0,0,1]) = 0
eval(1,[2,0,0,3,0,-7]) = -2
eval(2,[1,-2,3]) = 3
eval(-2,[1,2,3,4,5,6,7]) = 31
```

In order to solve this problem, you write a lex file *evalpoly.l* for identifying the tokens from the input file, and a yacc file *evalpoly.y* with `main()` for parsing and interpreting the input. Use the grammar given below. Since Horner's formula processes the coefficients from left to right (higher degrees to lower degrees, the same as in a polynomial list), the coefficient list (non-empty) of a polynomial is generated by CLIST left-recursively from the degree (d) of the polynomial to the lowest degree (0). The start symbol is EXPR (a single polynomial-evaluation expression). A POLY is a polynomial in the […] format. An evaluation point or a coefficient is called an ARG. This may be a (signed) integer constant, or the result of another polynomial evaluation. The terminal symbols are shown in bold face. Here, **eval** is a keyword.

EXPR → **eval (** ARG **,** POLY **)**
POLY → **[** CLIST **]**
CLIST → ARG | CLIST **,** ARG
ARG → **NUM** | EXPR

[Lex file *evalpoly.l*]

Write a complete lex file (with all the three sections) to identify all the tokens, and to do other bookkeeping tasks. Here, **eval** (keyword) and **NUM** (integer constant) are non-letter tokens (to be numbered by yacc). The remaining tokens are the punctuation symbols: **, ( ) [** and **]** . Your lex file must include appropriate header files, and supply a definition of `yywrap()`. Invalid characters must be ignored with a warning message. **[10]**

```
%{
/* The needed system headers are automatically included by lex. Redoing so is not a harm anyway. */
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* The yacc-generated header file must be included by you (because of the tokens EVAL and NUM) */
#include "y.tab.h"
%}

ws      [ \t\n]+
num     [+-]?[0-9]+

%%

{ws}    { }
eval    { return EVAL; }
{num}   { yylval = atoi(yytext); return NUM; }
\(      { return yytext[0]; }
\)      { return yytext[0]; }
\[      { return yytext[0]; }
\]      { return yytext[0]; }
,       { return yytext[0]; }
.       { fprintf(stderr, "*** lex error: Invalid character '%c'\n", yytext[0]); }

%%

int yywrap ( ) { return 1; }
```

[Yacc file *evalpoly.y*]

The yacc file uses some global arrays to store polynomials. These polynomials are needed not for evaluating POLY using Horner's formula, but for printing. Since coefficients of a polynomial may be obtained from other evaluations, these sub-evaluations are done first, and stored as integers. Each polynomial is stored in an `int` array of size 1000 (assume that the input program does not deal with larger polynomials). Also, there is a provision of storing a maximum of 1000 polynomials (assume that this is sufficient for the input program). The two-dimensional array `poly[1000][1000]` stores the polynomials located in the input. The number of polynomials is stored in a global variable `npoly`. Polynomials are identified by their row indices in the `poly` array. A global array `nterms[1000]` stores the numbers of terms in the input polynomials. The function `printpoly(p)` (given below) prints the `p`-th polynomial (the `p`-th row of the two-dimensional array `poly`).

```
%{
int poly[1000][1000];
int nterms[1000];
int npoly = 0;

void printpoly ( int p )
{
   printf("[%d", poly[p][0]);
   for (int i = 1; i < nterms[p]; ++i) printf(",%d", poly[p][i]);
   printf("]");
}

/* Write other declarations (extern) needed for compilation */
```

[4]

```
extern FILE *yyin;
extern int yylex () ;
void yyerror ( char * ) ;

%}
```

Define the tokens and the start symbol. All grammar symbols are now `int`-valued, so you do not need a union declaration. **[4]**

```
%token EVAL NUM
%start EXPR

%%
```

The production for EXPR involves a marker non-terminal P to record the number of the polynomial (row index in the `poly` array) for storing its coefficients. P is pushed to the parse stack just before POLY is going to be read. At the time of reduction by this production, a line like `eval(1,[2,0,0,3,0,-7]) = -2` is printed, and the value of POLY (result of evaluation of POLY at ARG) is pushed to the parse stack (as EXPR). **[6]**

```
EXPR   :   EVAL   '('   ARG   ','   P   POLY   ')'   {

            printf("eval(%d,",$3);
            printpoly($5);
            printf(") = %d\n", $6);
            $$ = $6;

       } ;
```

The marker non-terminal P → ε is used to record the next polynomial number (row index in `poly`), and to initialize the number of terms of that polynomial to 0. **[4]**

```
P   :   {

            nterms[npoly] = 0;
            $$ = npoly;
            ++npoly;

       } ;
```

POLY copies the value of CLIST (the evaluated polynomial). **[2]**

```
POLY   : '[' CLIST ']'  {

              $$ = $2;

       } ;
```

Now, write the actions for the productions of CLIST. Each of these actions serves two purposes. First, one step in Horner's formula is calculated (eval(a,[a_d, … , a_i]) = eval(a,[a_d, … , a_{i+1}]) × a + a_i for $i < d$), and pushed to the parse stack. Second, the new coefficient (a_i) used in the formula is appended to the current polynomial. These computations involve two values available in the parse stack. The evaluation point is sitting as an (evaluated) ARG, and the current polynomial number (row index) is stored in (the last) P. Both these values are available at fixed depths relative to the top of the stack (beneath CLIST at the time of reduction). **Do not use other marker non-terminals.** Note also that **Horner's formula is applied on the fly** (the incremental formula is mentioned above), and **not on the stored polynomial**.

```
CLIST  :  ARG {
```
[6]

```
              $$ = $1;
              poly[$-1][0] = $1;          /* You can also write: poly[$-1][nterms[$-1]] = $1; */
              nterms[$-1] = 1;            /* Or you can do this: ++nterms[$-1]; */


       }
       |  CLIST , ARG    {
```
[6]

```
             $$ = $1 * $-3 + $3;
             poly[$-1][nterms[$-1]] = $3;
             ++nterms[$-1];


       } ;
```

Finally, write the actions against the productions for ARG. **[2 + 2]**

```
ARG    :   NUM {

              $$ = $1;

       }
       |   EXPR {

             $$ = $1;

       } ;

%%
```

Write the `main()` function in the third section. Use (conditional) redirection of a file name to lex's input. **[4]**

```
int main ( int argc, char *argv[] )
{
   if (argc > 1) yyin = (FILE *)fopen(argv[1],"r");
   yyparse();
   fclose(yyin);
   exit(0);
}
```