CS39003 Compilers Laboratory
Lab Test 1
28-Aug-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, you deal with polynomials. A polynomial $f(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \ldots + a_dx^d$ is stored in an array as (a_0, a_1, a_2, a_3, … , a_d). Assume that each coefficient a_i is a (signed) integer. Let us store a polynomial in a static array with MAXTERMS entries. Zero terms are stored as 0. For example, if MAXTERMS = 16, the polynomial $10x - 4x^2 + 5x^6 - x^7$ is stored as (0, 10, – 4, 0, 0, 0, 5, – 1, 0, 0, 0, 0, 0, 0, 0, 0). We will support named and constant polynomials with the operations + (addition), – (subtraction), and ′ (derivative). For simplicity, our polynomial expressions will contain no parentheses. Here is an example of a polynomial assignment: $p = f - g' + (0, 1, 2, 3) - (5, 4, 3, 2, 1)' - h$. Here, p, f, g, and h are named polynomials, and the parenthesized lists are constant polynomials. If less than MAXTERMS entries are supplied in a polynomial constant, the unspecified higher-degree coefficients are stored as 0. You are required to write a recursive LL(1) parser to process a sequence of polynomial assignments. The grammar for a polynomial assignment (a line in the input file) is given below. The terminal symbols (tokens) are written in bold face. Here, L, E, T, C, and D stand for a line of input, a polynomial expression, a term in an expression, list of coefficients, and derivative operator (if any).

L	→	**id =** E	FIRST(L) = { **id** }	FOLLOW(L) = { **\n** }
E	→	T D E′	FIRST(E) = { **(** , **id** }	FOLLOW(E) = { **\n** }
E′	→	ε | **+** E | **–** E	FIRST(E′) = { ε , **+** , **–** }	FOLLOW(E′) = { **\n** }
T	→	**(** C **)** | **id**	FIRST(T) = { **(** , **id** }	FOLLOW(T) = { **′** , **+** , **–** , **\n** }
C	→	**num** C′	FIRST(C) = { **num** }	FOLLOW(C) = { **)** }
C′	→	ε | **,** C	FIRST(C′) = { ε , **,** }	FOLLOW(C′) = { **)** }
D	→	ε | **′**	FIRST(D) = { ε , **′** }	FOLLOW(D) = { **+** , **–** , **\n** }

[Lex file]

First, write a lex file `poly.l` to identify the patterns in the input. The first section (Definitions Section) has the definitions of the token types (already provided) and of some regular expressions (you write these).

```
%{
/* Token types to be returned by lex */
#define ID 1                /* Valid name */
#define NUM 2               /* Signed integer */
#define ASGN 3              /* The assignment operator */
#define PLUS 4              /* The addition operator */
#define MINUS 5             /* The subtraction operator */
#define DERIV 6             /* The derivative operator (single forward quote) */
#define LPAREN 7            /* Left parenthesis */
#define RPAREN 8            /* Right parenthesis */
#define COMMA 9             /* Comma used as separator between coefficients of polynomial constants */
#define EOL 10              /* End of line */
%}

/* Define regular expressions for white space, C-type ID's, and signed integers */
```

[5]

```
ws      [ \t]+
id      [a-zA-Z_][a-zA-Z0-9_]*
num     [+-]?[0-9]+


%%
```

In the second section (Rules Section), write the actions for white space and the ten token types defined above. Each action for a token should only return the appropriate token type (and do nothing else). Also add an action to discard invalid characters, with a warning message. **[12]**

```
{ws}    { }
{id}    { return ID; }
=       { return ASGN; }
\(      { return LPAREN; }
{num}   { return NUM; }
,       { return COMMA; }
\)      { return RPAREN; }
\+      { return PLUS; }
-       { return MINUS; }
'       { return DERIV; }
\n      { return EOL; }
.       { fprintf(stderr, "+++ Invalid character '%c'\n", yytext[0]); }
```

%%

In the third section (User Codes Section), write only the relevant function for lex to compile `poly.l`. Do not write any `main()` function here. **[2]**

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

[Parser code]

The LL(1) parsing table for our grammar is given below. Your task is to write a **recursive descent (predictive) parser** based on this table, in a C file `llparser.c`. Use C constructs only. You do not need to use STL data types.

	id	num	=	+	–	′	(	)	,	\n
L	id = E									
E	T D E′						T D E′			
E′				+ E	– E					ε
T	id						(C)			
C		num C′								
C′								ε	, C	
D				ε	ε	′				ε

Your parsing code defines the `poly` data type as follows. It uses two global polynomials `rval` and `term` needed during parsing. The next token waiting at the input is stored in the global `int` variable `token`. A function `llerror()` needs to be invoked in case of a parsing error (accessing an empty entry in the above parsing table).

```
#define MAXTERMS 32
typedef int poly[MAXTERMS];  /* Polynomial data type */
poly rval, term;             /* To be used by the parsing algorithm */
int token;                   /* The next token from the input */

void llerror ( char *msg ) { fprintf(stdrr, "*** Parse error: %s\n", msg); exit(1); }
```

The following functions are to be used in the parsing code. You do not have to implement them. In your code, follow the prototypes as explained below.

`polyinit(f)`	Initialize `f` to the zero polynomial
`polyadd(f, g)`	Set `f` to `f` + `g` (you may have `f` = `g`)
`polysub(f, g)`	Set `f` to `f` – `g` (you may have `f` = `g`)
`polydrv(f, g)`	Store in `f` the derivative of `g` (you may have `f` = `g`)
`polyprn(f)`	Print the polynomial `f`
`STget(f, name)`	Read in `f` the polynomial of the given `name` from the symbol table (if undefined, set `f` to 0)
`STstr(name,f)`	Store the polynomial `f` in the symbol table with the given `name` (overwrite if already present)

The parser code uses the following mutually recursive functions for the non-terminals in the grammar.

```
void L(), E(int), E1(), T(), C(int), C1(int), D();      /* E′() and C′() are renamed as E1() and C1() */
```

We do not use parentheses to group expressions, so the above functions do not need to return anything. We use two global polynomials `rval` and `term` to compute respectively the R-value in an assignment (a polynomial expression), and each term in the expression. `L()` initializes `rval` to 0. Each term is a named polynomial (use symbol-table lookup) or a literal polynomial (enclosed by parentheses), followed optionally by the derivative operator (single forward quote). `E()` initializes `term` to 0, and `T()`, `C()`, `C1()`, and `D()` prepare this polynomial in `term`. `E()` then adds/subtracts (as needed) `term` to/from `rval`. Finally, `L()` stores `rval` in the symbol table. Write below the bodies of the non-terminal functions. Implement recursive descent (predictive) parsing using the LL(1) parsing table given earlier. **[4 × 7]**

```
void L ( )
{
            char *name;

            if (token != ID) llerror("ID expected");
            name = strdup(yytext);
            token = yylex();
            if (token != ASGN) llerror("= expected");
            token = yylex();
            polyinit(rval);
            E(PLUS);
            STstr(name,rval);
            printf("+++ Stored %s = ", name); polyprn(rval);

}
void E ( int op )
/* op is PLUS or MINUS, needed to know whether the next term is to be added to or subtracted from rval */
{
            polyinit(term);
            if ((token != ID) && (token != LPAREN)) llerror("id or ( expected");
            T();
            D();
            if (op == PLUS) polyadd(rval,term); else if (op == MINUS) polysub(rval,term);
            E1();

}
void E1 ( )
{
            if (token == EOL) return;
            if ((token == PLUS) || (token == MINUS)) {
               op = token;
               token = yylex();
               E(op);
               return;
            }
            llerror("+, - or \n expected");

}
```

```
void T ( )
{

          if (token == ID) { STget(term,yytext); token = yylex(); }
          if (token == LPAREN) {
             token = yylex();
             C(0);
             if (token != RPAREN) llerror(") expected");
             token = yylex();
          }

}
```

The non-terminal functions `C()` and `C1()` read a polynomial in `term` coefficient-by-coefficient. It is therefore necessary to send to the functions the index at which the next coefficient is to be stored.

```
void C ( int i )
{

          if (token != NUM) llerror("num expected");
          term[i] = atoi(yytext);
          token = yylex();
          C1(i);

}
void C1 ( int i )
{

          if (token == RPAREN) return;
          if (token == COMMA) {
             token = yylex();
             C(i+1);
          }

}
void D ( )
{

          if (token == DERIV) { polydrv(term,term); token = yylex(); }
          if ((token != PLUS) && (token != MINUS) && (token != EOL))
             llerror("+ or - or \\n expected");

}
int main ( int argc, char *argv[] )
{
   /* Let lex read from a file (instead of stdin) if a command-line argument is supplied */          [2]

          if (argc > 1) yyin = (FILE *)fopen(argv[1],"r");
   ____________________________________________________________________________________________________
   while (1) {
      token = yylex();

      if ( ____________________ ) break;                                                             [1]
                token == 0
      L();
   }
   exit(0);
}
```