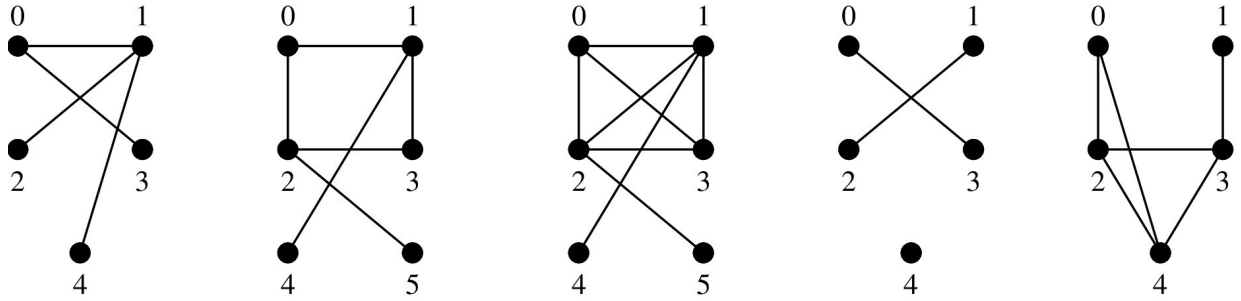
CS39003 Compilers Laboratory
Autumn 2026, Lab Test 1
27-Aug-2026, 07:15pm – 08:45pm
Maximum marks: 75

Roll No: _____

Name: _____

In this exercise, you deal with (undirected) graphs, and write an LL(1) interpreter for graph assignments. A constant graph is specified by its number n of vertices, followed by a list of its edges each in the form $u-v$. We number the vertices of the graph as $0, 1, 2, \ldots, n-1$. Three graph operations are to be supported. Let G be a graph on n_1 vertices and with edge set E_1. Likewise, let H be a graph on n_2 vertices and with edge set E_2. We define $G+H$ as the graph with $\max(n_1, n_2)$ vertices, and with the edge set $E_1 \cup E_2$. We also defined $G-H$ as the graph with n_1 vertices and with the edge set $E_1 - E_2$. Finally, the complement of G is the graph G' with n_1 vertices such that edge (u, v) is an edge of G' if and only if it is not an edge of G. The following figure demonstrates these operations.



An input file consists of a sequence of graph assignments. Constant graphs are presented within square brackets. Expressions involve the operations + , – , and ′. The operators + and – have the same precedence, and are left-to-right associative. The operator ′ has higher precedence than + or –. You can alter the precedence by using parentheses. The complement operation applies to graph variables, to parenthesized graph expressions, and also to bracketed constant graphs. An example input file is given below. Note that white spaces (spaces, tabs, empty lines) are allowed anywhere except within graph names and within numbers.

```
G = [ 5 0-1 1-2 0-3 4-1 ]
H = [6  0 - 1  1 - 3  3 - 2  2 - 0  1 - 4  2 - 5]
K5 = [5]'
G1 = K5 - G
G2 = ((G - H)' - (G + H)')' - (H - G)
```

An LL(1) grammar for each non-empty line of graph assignment is given below.

(1)	ASGN	⟶	**id** **=** RVAL	FIRST(ASGN) = { **id** }	FOLLOW(ASGN) = { **$** }
(2)	RVAL	⟶	**[** CONS **]** COMP	FIRST(RVAL) = { **[** , **id** , **(** }	FOLLOW(RVAL) = { **$** }
(3)			| EXPR		
(4)	CONS	⟶	**num** ELIST	FIRST(CONS) = { **num** }	FOLLOW(CONS) = { **]** }
(5)	ELIST	⟶	**num** **–** **num** ELIST	FIRST(ELIST) = { **num** , ε }	FOLLOW(ELIST) = { **]** }
(6)			| ε		
(7)	EXPR	⟶	TERM EREST	FIRST(EXPR) = { **id** , **(** }	FOLLOW(EXPR) = { **)** , **$** }
(8)	EREST	⟶	**+** TERM EREST	FIRST(EREST) = { **+** , **–** , ε }	FOLLOW(EREST) = { **)** , **$** }
(9)			| **–** TERM EREST		
(10)			| ε		
(11)	TERM	⟶	| CORE COMP	FIRST(TERM) = { **id** , **(** }	FOLLOW(TERM) = { **+** , **–** , **)** , **$** }
(12)	CORE	⟶	**id**	FIRST(CORE) = { **id** , **(** }	FOLLOW(CORE) = { **+** , **–** , **)** , **'** , **$** }
(13)			| **(** EXPR **)**		
(14)	COMP	⟶	**'**	FIRST(COMP) = { **'** , ε }	FOLLOW(COMP) = { **+** , **–** , **)** , **$** }
(15)			| ε		

Here, ASGN (assignment) is the start symbol (for each non-empty input line). Other non-terminals are RVAL (r-value of an assignment), CONS (a constant graph), ELIST (list of edges), EXPR (expression involving graph operations), EREST (rest of an expression), TERM (a term in an expression), CORE (a graph name or a parenthesized expression), and COMP (the optional quote symbol). Terminal symbols are shown in bold face (and in lower case).

Part 1: Write a lex file

Your lex file must *return* an integer for each token type. The only valid token types for this language are **id** (a non-empty string denoting the name of a graph), **num** (an integer; since numbers of vertices and vertex numbers cannot be negative, *only unsigned integers* are allowed), the operators **=**, **+**, **−**, and **'**, and the punctuation symbols **(**, **)**, **[**, and **]**, and **\n** (playing the role of the input-end marker **$**). For any other character, an **error** token should be returned. Note that constructs like **id =** (as in ASGN) and **num − num** (as in ELIST) are *not* considered as single tokens. White spaces (spaces, tabs, and empty lines) should be ignored. Only the token types **id** and **num** have values which can be retrieved from the currently matched `yytext`. Write below your complete lex file (all three sections, *no* `main()` function). **[15]**

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

char *graphname;
int vertexnum;

#define ID 1000
#define NUM 1001
#define EQ 1002
#define LB 1003
#define RB 1004
#define LP 1005
#define RP 1006
#define PLUS 1007
#define MINUS 1008
#define QUOTE 1009
#define EOL 1010
#define ERR 2000
%}

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

%%

{ws}    { }
{id}    { graphname = strdup(yytext); return ID; }
{num}   { vertexnum = atoi(yytext); return NUM; }
=       { return EQ; }
\[      { return LB; }
\]      { return RB; }
\(      { return LP; }
\)      { return RP; }
\+      { return PLUS; }
-       { return MINUS; }
'       { return QUOTE; }
\n      { return EOL; }
.       { fprintf(stderr, "*** Bad character: '%s'\n", yytext); return ERR; }

%%

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

Part 2: Write an LL(1) parser

The given grammar is LL(1), and has the following LL(1) parsing table.

	id	num	[	]	(	)	+	–	'	$
ASGN	(1)									
RVAL	(3)		(2)		(3)					
CONS		(4)								
ELIST		(5)		(6)						
EXPR	(7)				(7)					
EREST						(10)	(8)	(9)		(10)
TERM	(11)				(11)					
CORE	(12)				(13)					
COMP						(15)	(15)	(15)	(14)	(15)

Your task is to write a *recursive descent parser* (exactly as given in the Dragon Book) based on the above parsing table. For each non-terminal symbol, you should write a function (of the same name as the non-terminal). These non-terminal functions will be mutually recursive as required by the productions of the grammar.

Before writing these functions, note that you need functions for graphs and for symbol-table management. In the parser code, you must *not* implement these functions. Use the following ADT for graphs. Assume that there is a data type called `graph`. The following ADT operations (and nothing else) on this data type are to be used.

- `newgraph(n)` creates and returns a `graph` with `n` vertices and no edges.
- `addedge(G,u,v)` adds the (undirected) edge `(u,v)` to the graph `G`, and returns the modified graph.
- `gadd(G1,G2)` returns the graph `G1 + G2` (as defined on the first page).
- `gsub(G1,G2)` returns the graph `G1 – G2` (as defined on the first page).
- `gcomp(G)` returns the complement graph `G′` (as defined on the first page).
- `gprn(G)` prints a graph `G`.

Assume that the symbol table `ST` is maintained globally. In this part, you do not need to bother about how `ST` is defined and implemented. Use only the following interface to query and update `ST`. A reference to an entry in `ST` will be of the data type `stref`. Since `ST` is stored globally, you do not need to pass it to the functions.

- `stsearch(var)` returns a reference (an `stref`) to the symbol-table entry storing the graph named `var` (a string). If the name is not defined, then `INVALID` is to be returned.
- `stget(s)` returns the graph stored at the (valid) `stref s`.
- `stsetlval(var)` first checks whether a graph named `var` is already stored in `ST`. If that is the case, a reference to that entry (an `stref`) is returned. Otherwise, a new entry in `ST` is created to store the graph named `var`, and a reference (an `stref`) to this new entry in `ST` is returned.
- `stsetrval(s,G)` sets to `G` the `graph` associated with the `ST` entry reference `s` (an `stref`).
- `stprn(s)` prints the name and the graph for the `stref s`.

Now, implement the nine non-terminal functions and the `main()` function. These should be C functions that use the above ADT calls for handling graphs and the symbol table `ST` (there is no need to write C++ code here). *Attempts to implement the ADT functions will be penalized.* Whenever needed, appropriate arguments should be passed to the functions, and appropriate data items (like a `graph`) should be returned. The `main()` function may take a single optional command-line argument specifying the name of the input file. This file name (if supplied) should be used to set lex's input `yyin`. Assume that whenever a syntax error is found, the program exits with an error code (no need to print any error messages). However, when an undefined graph name is found in an expression, do not exit. Instead print an error message, and use the empty graph (the graph with zero vertices). For your ready reference, portions of the grammar and the parsing table are reproduced on some of the following pages. **[5 × 9]**

(1)	ASGN	⟶	**id** = RVAL	FIRST(ASGN) = { **id** }	FOLLOW(ASGN) = { **$** }
(2)	RVAL	⟶	**[** CONS **]** COMP	FIRST(RVAL) = { **[** , **id** , **(** }	FOLLOW(RVAL) = { **$** }
(3)			| EXPR		
(4)	CONS	⟶	**num** ELIST	FIRST(CONS) = { **num** }	FOLLOW(CONS) = { **]** }
(5)	ELIST	⟶	**num – num** ELIST	FIRST(ELIST) = { **num** , ε }	FOLLOW(ELIST) = { **]** }
(6)			| ε		
(7)	EXPR	⟶	TERM EREST	FIRST(EXPR) = { **id** , **(** }	FOLLOW(EXPR) = { **)** , **$** }

	id	**num**	**[**	**]**	**(**	**)**	**+**	**–**	**'**	**$**
ASGN	(1)									
RVAL	(3)		(2)		(3)					
CONS		(4)								
ELIST		(5)		(6)						
EXPR	(7)				(7)					

For your convenience, the function for the start symbol ASGN is given below. Write the other functions in this format. Assume that `nexttoken` is a global `int` variable storing the next unprocessed token returned by lex.

```
void ASGN ( )
{
     char *lval;
     graph G;
     stref i;

     if (nexttoken != ID) exit(1);
     lval = strdup(yytext);
     nexttoken = yylex();
     if (nexttoken != EQ) exit(2);
     nexttoken = yylex();
     G = RVAL();
     i = stsetlval(lval);
     stsetrval(i,G);
     stprn(i);
}

graph RVAL ( )
{
     graph G;

     if (nexttoken == LB) {
        nexttoken = yylex();
        G = CONS();
        if (nexttoken != RB) exit(3);
        nexttoken = yylex();
        if (COMP()) G = gcomp(G);
     } else if ((nexttoken == ID) || (nexttoken == LP)) {
        G = EXPR();
     } else {
        exit(4);
     }

     return G;

}
```

```
graph CONS ( )
{



     graph G;

     if (nexttoken != NUM) exit(5);

     G = newgraph(vertexnum);
     nexttoken = yylex();
     G = ELIST(G);

     return G;




}

graph ELIST ( graph G )
{







    int src, dst;

    if (nexttoken == RB) return G;

    if (nexttoken != NUM) exit(6);

    src = vertexnum;
    nexttoken = yylex();
    if (nexttoken != MINUS) exit(7);
    nexttoken = yylex();
    if (nexttoken != NUM) exit(8);
    dst = vertexnum;
    G = addedge(G,src,dst);
    nexttoken = yylex();

    G = ELIST(G);

    return G;









}

graph EXPR ( )
{



     graph G;

     if ((nexttoken != ID) && (nexttoken != LP)) exit(9);
     G = EREST(TERM());
     return G;




}
```

(8)	EREST ⟶	**+** TERM EREST	FIRST(EREST) = { **+** , **–** , ε }	FOLLOW(EREST) = { **)** , **$** }
(9)		| **–** TERM EREST		
(10)		| ε		
(11)	TERM ⟶	| CORE COMP	FIRST(TERM) = { **id** , **(** }	FOLLOW(TERM) = { **+** , **–** , **)** , **$** }
(12)	CORE ⟶	**id**	FIRST(CORE) = { **id** , **(** }	FOLLOW(CORE) = { **+** , **–** , **)** , **'** , **$** }
(13)		| **(** EXPR **)**		
(14)	COMP ⟶	**'**	FIRST(COMP) = { **'** , ε }	FOLLOW(COMP) = { **+** , **–** , **)** , **$** }
(15)		| ε		

	id	**num**	**[**	**]**	**(**	**)**	**+**	**–**	**'**	**$**
EREST						(10)	(8)	(9)		(10)
TERM	(11)				(11)					
CORE	(12)				(13)					
COMP						(15)	(15)	(15)	(14)	(15)

Note: The graph operations + and – are left-to-right associative, and an LL(1) grammar cannot be left-recursive. You therefore need to pass the partially computed graph for the next operation in the rest of the expression.

```
graph EREST ( graph G )
{

    if ((nexttoken == RP) || (nexttoken == EOL)) return G;
    if (nexttoken == PLUS) { nexttoken = yylex(); return EREST(gadd(G,TERM())); }
    if (nexttoken == MINUS) { nexttoken = yylex(); return EREST(gsub(G,TERM())); }
    exit(10);

}

graph TERM ( )
{

    graph G;

    if ((nexttoken != ID) && (nexttoken != LP)) exit(11);

    G = CORE();
    if (COMP()) G = gcomp(G);

    return G;

}
```

```
graph CORE ( )
{




     graph G;

     if (nexttoken == ID) {
        stref i = stsearch(graphname);
        nexttoken = yylex();
        if (i < 0) {
           fprintf(stderr, "*** Error in CORE(): Invalid graph name %s in rval\n", graphname);
           G = newgraph(0);
        } else {
           G = stget(i);
        }
        return G;
     }

     if (nexttoken == LP) {
        nexttoken = yylex();
        G = EXPR();
        if (nexttoken != RP) exit(12);
        nexttoken = yylex();
        return G;
     }

     exit(13);









}

int COMP ( ) /* Returns a flag (1/0) indicating whether complement is to be computed or not */
{



     if (nexttoken == QUOTE) { nexttoken = yylex(); return 1; }
     if ((nexttoken == PLUS) || (nexttoken == MINUS) || (nexttoken == RP) || (nexttoken == EOL)) { return 0; }
     exit(14);



}

int main ( int argc, char *argv[] )
{



     if (argc > 1) yyin = (FILE *)fopen(argv[1], "r");

     while (1) {
        nexttoken = yylex();
        if (nexttoken == 0) break;
        if (nexttoken == ID) ASGN();
        if (nexttoken == EOL) continue;
        if (nexttoken == ERR) continue;
     }

     if (argc > 1) fclose(yyin);

     exit(0);




}
```

Part 3: Implement your symbol table ST

Declare a suitable data type for storing the symbol table `ST` and the corresponding `stref` type. Then, implement the functions `stsearch()`, `stget()`, `stsetlval()`, `stsetrval()`, and `stprn()` with the exact prototypes as given on Page 3. Here too, you use the `graph` type and the associated ADT calls without bothering about their implementations. **[15]**

```
typedef struct _stentry {
   char name[16];
   graph G;
} stentry;

typedef int stref;

int nstentry = 0;
stentry ST[MAXENT];

stref stsearch ( char *varname )
{
   stref i;

   for (i=0; i<nstentry; ++i)
      if (!strcmp(varname, ST[i].name)) return i;
   return -1;
}

graph stget ( stref i )
{
   return ST[i].G;
}

stref stsetlval ( char *varname )
{
   stref i;

   for (i=0; i<nstentry; ++i)
      if (!strcmp(varname, ST[i].name)) return i;
   strcpy(ST[nstentry].name, varname);
   ++nstentry;
   return i;
}

void stsetrval ( stref i, graph G )
{
   ST[i].G = G;
}

void stprn ( stref i )
{
   printf("\n+++ ST[%d] stores the graph %s\n", i, ST[i].name);
   gprn(ST[i].G);
}
```