CS39003 Compilers Laboratory
Autumn 2026
Assignment 4
Date of posting: 11-Aug-2026

LL(1) Parsing

In this assignment, you write an LL(1) parser from the scratch, for a language of assignments of matrices. The matrix operations are similar to as in Lab Assignment 1 (LA1). In particular, we assign constant matrices to variables in the same format as in LA1. For matrix operations, we now introduce two additional constructs. First, we can enclose subexpressions within parentheses, and the level of nesting of the parentheses may be arbitrary. Second, we introduce scalar multiplication by writing a number (a signed integer) before a matrix name or a parenthesized expression. Scalar multiplication will *not* use the product sign. The symbol `*` is reserved for matrix-matrix multiplications only. Here are examples of an assignment of a constant matrix and of an assignment of a matrix expression.

```
MyMat_123  = [ 1, 2, 3, 4;   5, -6, 7, -8;     9, -10, 11, -12]
M          = 3((M1 + 2M2) * (2M3 * -3M4 * (M5 - M6) + M3 * 2(M5 + M6 * (M4 - 2M7)))) - M8 + 10M9
```

Spaces are allowed at any place except inside an identifier or a number. In the expression `M1-2M2`, the token `-2` is treated as a number (lex performs longest matches), so parsing would fail. To disambiguate, write this as `M1 - 2M2`.

The grammar for each (non-empty) line in the input file is given below. This grammar is presented in the LL(1) format. The start symbol is ASGN.

(1)	ASGN	⟶	**ID** **=** EXPR
(2)	EXPR	⟶	LITM |
(3)			MATOP
(4)	MATOP	⟶	TERM TREST
(5)	TREST	⟶	ε |
(6)			**+** TERM TREST |
(7)			**–** TERM TREST
(8)	TERM	⟶	FACTOR FREST
(9)	FREST	⟶	ε |
(10)			***** FACTOR FREST
(11)	FACTOR	⟶	SCALAR MATFAC
(12)	SCALAR	⟶	ε |
(13)			**NUM**
(14)	MATFAC	⟶	**ID** |
(15)			**(** MATOP **)**
(16)	LITM	⟶	**[** RLIST **]**
(17)	RLIST	⟶	ROW RLREST
(18)	RLREST	⟶	ε |
(19)			**;** ROW RLREST
(20)	ROW	⟶	**NUM** RREST
(21)	RREST	⟶	ε |
(22)			**,** **NUM** RREST

Part 1: Write a lex file

The language in question is not regular, and lex cannot handle general context-free productions. This is unlike the two previous assignments where your interpreter performed the work from inside lex's actions. In this (and future) assignments, lex will only decompose the input into tokens. Upon the discovery of each token, lex will *return* the *token type*. The token types may be `#define`'d as integers. For our grammar, the token types are **ID** (matrix names), **NUM** (signed integers), **LP** (left parenthesis), **RP** (right parenthesis), **LB** (left bracket), **RB** (right bracket), **PLUS** (addition operator), **MINUS** (subtraction operator), **STAR** (multiplication operator), **EQUAL** (assignment operator), **SEMICOLON** (row separator), **COMMA** (element separator), and **EOL** ($ or end-of-line). For any other character, a token **ERROR** may be returned. Only the first two token types **ID** and **NUM** have values (a string and an integer, respectively). These may be retrieved from the currently matched `yytext`. Ignore white spaces (except new lines).

Write a lex file that will only return the above token types. No other actions will be taken by the lex file.

Part 2: Write an LL(1) parser

The grammar we are dealing with is LL(1). The FIRST and FOLLOW sets for the non-terminals are given below.

FIRST(ASGN)	=	{ ID }	FOLLOW(ASGN)	=	{ $ }
FIRST(EXPR)	=	{ [, NUM , ID , (}	FOLLOW(EXPR)	=	{ $ }
FIRST(MATOP)	=	{ NUM , ID , (}	FOLLOW(MATOP)	=	{) , $ }
FIRST(TREST)	=	{ ε , + , – }	FOLLOW(TREST)	=	{) , $ }
FIRST(TERM)	=	{ NUM , ID , (}	FOLLOW(TERM)	=	{ + , – ,) , $ }
FIRST(FREST)	=	{ ε , * }	FOLLOW(FREST)	=	{ + , – ,) , $ }
FIRST(FACTOR)	=	{ NUM , ID , (}	FOLLOW(FACTOR)	=	{ + , – ,) , $ }
FIRST(SCALAR)	=	{ ε , NUM }	FOLLOW(SCALAR)	=	{ ID , (}
FIRST(MATFAC)	=	{ ID , (}	FOLLOW(MATFAC)	=	{ + , – ,) , $ }
FIRST(LITM)	=	{ [}	FOLLOW(LITM)	=	{ $ }
FIRST(RLIST)	=	{ NUM }	FOLLOW(RLIST)	=	{] }
FIRST(RLREST)	=	{ ε , ; }	FOLLOW(RLREST)	=	{] }
FIRST(ROW)	=	{ NUM }	FOLLOW(ROW)	=	{ ; ,] }
FIRST(RREST)	=	{ ε , , }	FOLLOW(RREST)	=	{ ; ,] }

From these values, the LL(1) parse table for the given grammar of matrix assignments can be derived as follows.

	ID	NUM	[	]	(	)	+	–	*	;	,	$
ASGN	(1)											
EXPR	(3)	(3)	(2)		(3)							
MATOP	(4)	(4)			(4)							
TREST						(5)	(6)	(7)				(5)
TERM	(8)	(8)			(8)							
FREST						(9)	(9)	(9)	(10)			(9)
FACTOR	(11)	(11)			(11)							
SCALAR	(12)	(13)			(12)							
MATFAC	(14)				(15)							
LITM			(16)									
RLIST		(17)										
RLREST				(18)						(19)		
ROW		(20)										
RREST				(21)						(21)	(22)	

Write a *recursive* LL(1) parser based on the above table. Follow the algorithm exactly as given in the dragon book (see the algorithm of Figure 4.13 in Section 4.4.1 on Recursive-Descent Parsing). Write mutually recursive functions, one for each non-terminal symbol (there are 14 of them). The functions will keep on asking lex to supply the next token(s) from the input. For each next token a, and for each non-terminal N, an appropriate action is taken as dictated by the LL(1) parse table. At the beginning of each non-empty line, the function `ASGN()` corresponding to the start symbol ASGN will be called. The end-of-line character (`\n`) will play the role of **$**.

Maintain a symbol table. Each table should be able to store the name of a matrix, and the matrix itself (or a pointer to that matrix). The organizations (like data types) of matrices and of the symbol table are left to you. After every assignment, the matrix is to be printed.

You may break your input into multiple C/C++ source files. A header file may contain the data types to be used (like for matrices and symbol tables) and declare all global variables and function prototypes. Another file may contain only the functions for symbol-table management. Yet another file will define the matrix-related functions. Finally, a source file will include the parsing functions (one for each non-terminal) and a `main()` function. This organization of your entire code into functional modules will make your code readable, so we recommend doing this.

Part 3: Write a makefile

The makefile should have compile, run, and clean targets. Compilation involves calling lex to generate `lex.yy.c` from your lex file, and the compilation of your parser program to the final interpreter `a.out`. Include the other source files (and `lex.yy.c`) from the parser program (or mention the source files in appropriate order during the invocation of `gcc`/`g++`). The run target may assume that the interpreter will run on the file `input.txt`. The clean target will remove all non-source files (`lex.yy.c`, `a.out`, and any `.o` files that you may have generated).

What to submit

Submit in a single tar/tgz/zip archive your lex file, your C/C++ source file(s), and the makefile (after cleaning the directory).

Sample Output

Input

```
M1 = [ 1,2,3,4; 5,6,7,8; 9,10,11,12 ]

M2 = [ -1,2,-3,4; 5,-6,7,-8; -9,10,-11,12 ]

M3 = [ 1,2,3; 4,5,6; 7,8,9; 10,11,12 ]

M4 = M1 * M3
M5 = 2M2 * 3M3

M6 = M3 * (M1 + 2M2)

M7 = -2M3 * (M1 - 2M2) + M6
M7 = -2M3 * -6(M1 - 2M2) + 2(2M6 + 5M7) * 2M6

M8 = 2 ( M1 + M2 - 2 M1 ) * M6 * M3 + 100 M5 + 1000 M4

M9 = ( (M1 - 2M2) * M3 * 2(M1 - M2) - M1 ) * (M5 - M4)
M9 = (M5 - M4) * ( (M1 - 2M2) * M3 * 2(M1 - M2) - M1 )
```

Output

```
M1  =          1         2         3         4
               5         6         7         8
               9        10        11        12

M2  =         -1         2        -3         4
               5        -6         7        -8
              -9        10       -11        12

M3  =          1         2         3
               4         5         6
               7         8         9
              10        11        12

M4  =         70        80        90
             158       184       210
             246       288       330

M5  =        156       168       180
            -300      -312      -324
             444       456       468

M6  =          2        84         6       104
              17       174        27       224
              32       264        48       344
              47       354        69       464

M7  =       -146        76      -182        88
            -281       130      -371       160
            -416       184      -560       232
            -551       238      -749       304

M7  =      39224    106800     49320    156128
           62780     99528     75924    162368
           86336     92256    102528    168608
          109892     84984    129132    174848

M8  =      21552     23712     25872
         -158160   -173856   -189552
           25264     31136     37008
*** Incompatible matrices for multiplication

M9  =   EMPTY

M9  =    1795016    949952   2428856   1266608
       -10264256  -5425904 -13884704  -7234640
         3240984   1722912   4390440   2297136
```