INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR											
									Stamp / Signature of the Invigilator		
EXAMINATION (Mid Semester)									**SEMESTER (Autumn)**		
Roll Number									**Section**		**Name**
Subject Number	C	S	3	1	0	0	3	**Subject Name**	***Compilers***		
Department / Center of the student									**Additional Sheets**		

Important Instructions and Guidelines for Students

1. You must occupy your seat as per the Examination Schedule/Sitting Plan.
2. Do not keep mobile phones or any similar electronic gadgets with you even in the switched off mode.
3. Loose papers, class notes, books or any such materials must not be in your possession, even if they are irrelevant to the subject you are taking examination.
4. Data book, codes, graph papers, relevant standard tables/charts or any other materials are allowed only when instructed by the paper-setter.
5. Use of instrument box, pencil box and non-programmable calculator is allowed during the examination. However, exchange of these items of any other papers (including question papers) is not permitted.
6. Write on both sides of the answer script and do not tear off any page. **Use last page(s) of the answer script for rough work.** Report to the Invigilator if the answer script has torn or distorted page(s).
7. It is your responsibility to ensure that you have signed the Attendance Sheet. Keep your Admit Card/Identity Card on the desk for checking by the invigilator.
8. You may leave the examination hall for wash room or for drinking water for a very short period. Record your absence from the Examination Hall in the register provided. Smoking and the consumption of any kind of beverages are strictly prohibited inside the Examination Hall.
9. Do not leave the Examination Hall without submitting your answer script to the invigilator. **In any case, you are not allowed to take away the answer script with you.** After the completion of the examination, do not leave the seat until the invigilators collect all the answer scripts.
10. During the examination, either inside or outside the Examination Hall, gathering information from any kind of sources or exchanging Information with others or any such attempt will be treated as **'unfair means'**. Do not adopt unfair means and do not indulge in unseemly behavior.

Violation of any of the above instructions may lead to severe punishment.

Signature of the Student

To be filled in by the examiner

Question Number	1	2	3	4	5	6	7	8	9	10	**Total**
Marks obtained											

Marks Obtained (in words)	**Signature of the Examiner**	**Signature of the Scrutineer**

CS31003 COMPILERS
AUTUMN 2025 – 2026
MID-SEMESTER EXAMINATION
20-SEPTEMBER-2025, 2:00PM – 4:00PM
MAXIMUM MARKS: 75

Instructions to students

- Write your answers in the question paper itself.
- Answer **all** questions.
- Write in the blank spaces provided in the questions. Use the empty pages at the end for rough work. Please avoid supplementary sheets. The answers to all the questions must be written in this question paper only. If you continue some answer to the Rough-Work pages, please supply an appropriate pointer in your answer.
- Do not write anything on this page. Questions start from the next page (Page 3).
- If you need to make any assumptions in some questions, write them clearly in your respective answers.

1. **(a)** Write a regular expression for a date in the *dd*/*mm*/*yy* format. Here, *dd* is an integer in the range 1 – 31, *mm* is an integer in the range 1 – 12, and *yy* is a 2-digit or a 4-digit integer. A one-digit day or month can be specified by a single digit with or without a single leading zero. Your regular expression does not need to check the validity of a date. For example, 29/2/09 and 31/09/2025 are considered valid, whereas 32/8/99 and 12/12/345 are not. **[6]**

In the lex format, the regular expression can be written as follows.

```
DD      (0?[1-9])|[1-2][0-9]|3[0-1])
MM      (0?[1-9]|1[0-2])
YY      ([0-9][0-9])?[0-9][0-9]
DATE    DD/MM/YY
```

(b) Assume that a compiler works on a subset of C, and accepts tokens of the types KWD, ID, NUM, OP, and PUNC only (standing respectively for keywords, identifiers, numbers, operators, and punctuation symbols). Consider the following code snippet in that subset of C.

```
int prod = 1;
while ( prod <= n ) prod *= 2;
```

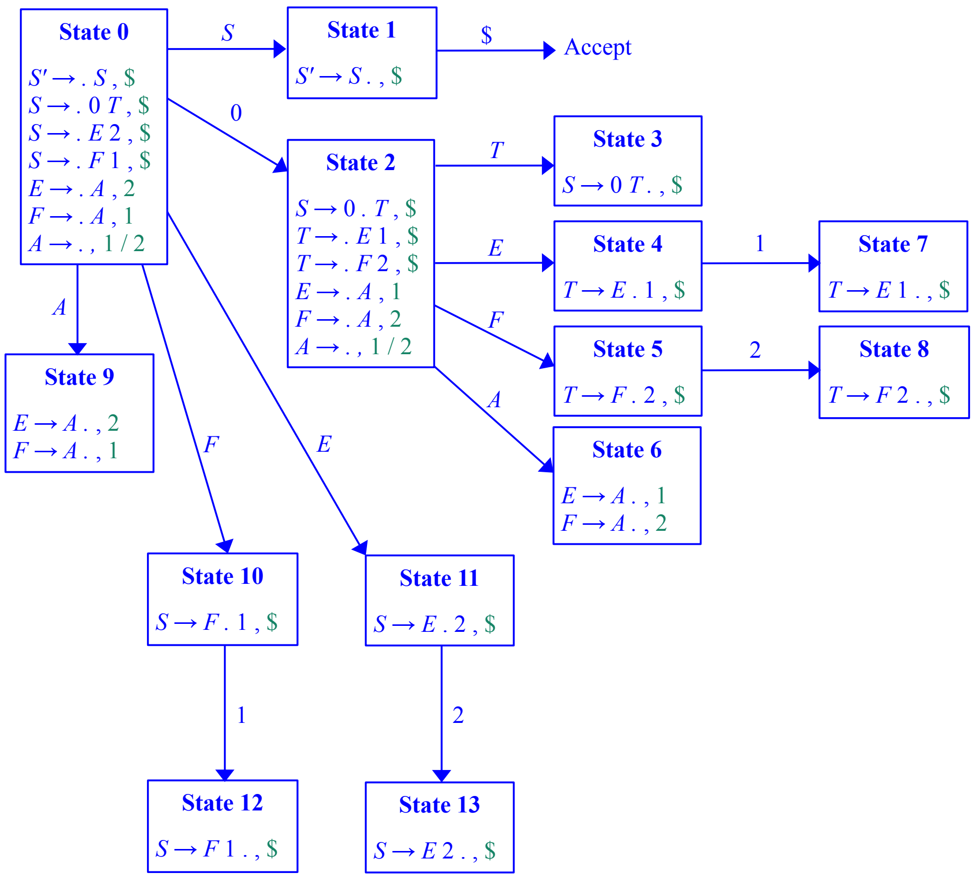
For this code snippet, identify the sequence of lexemes, and the respective token types and token values for the lexemes. Present your answer in the following tabular format. The last token is given below as an example. **[7]**

Lexeme	**Token Type**	**Token Value**
int	KWD	The keyword int (if stored in symbol table, reference to that entry)
prod	ID	Reference to the symbol-table entry for prod
=	OP	The operator ASSIGNMENT
1	NUM	Integer constant 1
;	PUNC	Statement delimiter
while	KWD	The keyword int (if stored in symbol table, reference to that entry)
(	PUNC	Left parenthesis
prod	ID	Reference to the symbol-table entry for prod
<=	OP	Comparison operator less than or equal to
n	ID	Reference to the symbol-table entry for n
)	PUNC	Right parenthesis
prod	ID	Reference to the symbol-table entry for prod
*=	OP	Operator multiply by
2	NUM	Integer constant 2
;	PUNC	Statement delimiter or Semi-colon

2. The following grammar has terminals 0, 1, and 2, and non-terminals *S* (start symbol), *T*, *E*, *F*, and *A*.

(1) $S \rightarrow 0\ T$	(2) $S \rightarrow E\ 2$	(3) $S \rightarrow F\ 1$
(4) $T \rightarrow E\ 1$	(5) $T \rightarrow F\ 2$	
(6) $E \rightarrow A$	(7) $F \rightarrow A$	(8) $A \rightarrow \varepsilon$

(a) Draw the LR(1) automaton for this grammar, in the space below. Number the states as 0 (the start state), 1 (the accept state), 2, 3, … **[9]**



(b) Using the LR(1) automaton of Part (a), construct the (canonical) LR(1) parsing table for the grammar (number the productions as given). From the table, justify that the given grammar is (canonical) LR(1). **[9]**

STATE	**ACTION**				**GOTO**				
	0	1	2	$	S	T	E	F	A
0	s2				1		11	10	9
1				Accept					
2						3	4	5	6
3				r1					
4		s7							
5			s8						
6		r6	r7						
7				r4					
8				r5					
9		r7	r6						
10		s12							
11			s13						
12				r3					
13				r2					

No multiple entries in any cell ⇒ No conflicts ⇒ The given grammar is LR(1)

(c) Mention clearly which of the states in the LR(1) automaton of Part (a) can be merged to form the LALR(1) automaton. You do not have to draw the LALR(1) automaton. From the merged states, conclude (with justification) whether the given grammar is LALR(1). **[4]**

The only merging happens between the states 6 and 9, giving the merged state 69 as follows. This state has a reduce-reduce conflict. So the given grammar is not LALR(1).

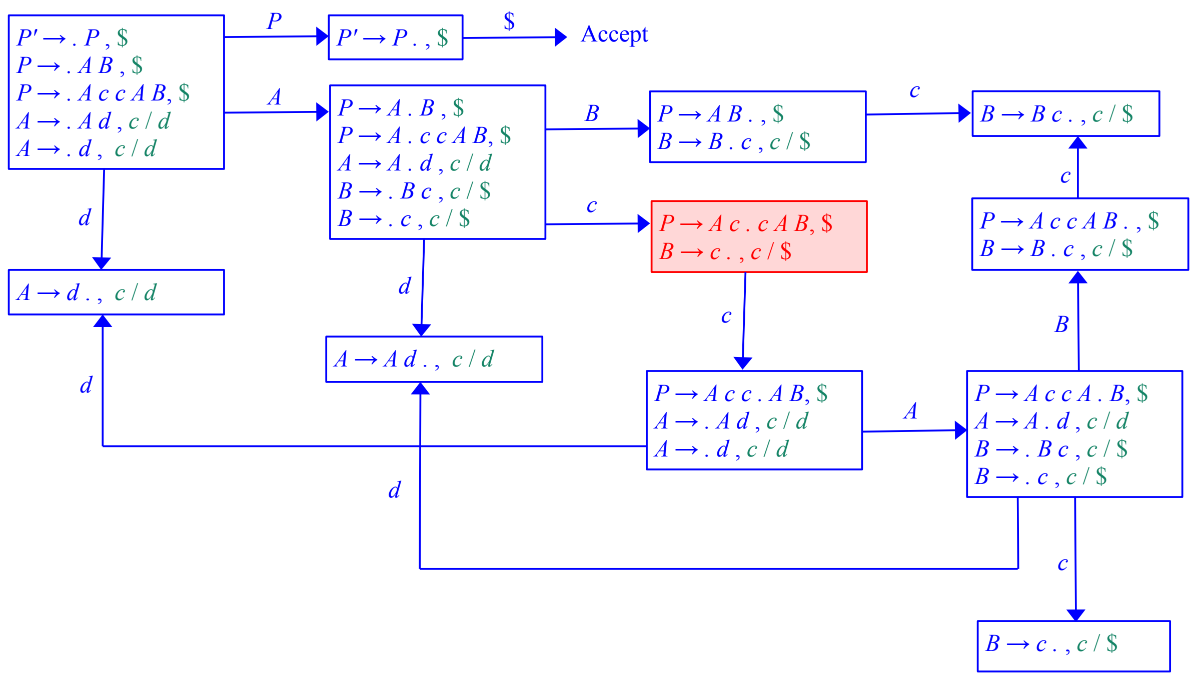
State 69

$E \rightarrow A\,.\,,\ 1\,/\,2$
$F \rightarrow A\,.\,,\ 1\,/\,2$

3. Think of a program that starts with a sequence A of declarations, and that ends with a sequence B of computations. Between these two sequences, there is an optional part. The programmer may write two steps of computation (user input, and computing sizes of dynamic arrays), and a set of subsequent declarations (of dynamic arrays). Let us denote each declaration step by d, and each computation step by c. In this exercise, we treat d and c as terminal symbols. Since the declaration steps and the computation steps should be carried out in the given sequence, we define the grammar of a program (or program pattern) P (the start symbol) as follows. In the grammar, the non-terminal symbols A and B generate non-empty sequences of declaration steps and computation steps, respectively.

$$\begin{array}{lll} P & \rightarrow & A\,B \mid A\,c\,c\,A\,B \\ A & \rightarrow & A\,d \mid d \\ B & \rightarrow & B\,c \mid c \end{array}$$

(a) Draw the LR(1) automaton for this grammar, in the space below. **[9]**



(b) Form the automaton of Part (a), conclude whether the given grammar is canonical LR(1) (do not to prepare the parsing table). Is the grammar LALR(1)? Justify. **[2]**

Since the LR(1) automaton contains one shift-reduce conflict (marked by red), the grammar in question is not LR(1).

There is no merging possibility in the LR(1) automaton, that is, the LR(1) automaton is the LALR(1) automaton too. The same shift-reduce conflict stays in the LALR(1) automaton, causing the grammar to be not LALR(1).

(c) Since the above grammar uses left recursion, this grammar is obviously not LL(1). Propose an LL(1) grammar for the given language. (**Warning:** Removal of left recursion alone will not be sufficient.) **[4]**

Let P first generate the sequence of initial non-empty sequence of declarations followed by only one c, in a non-left-recursive manner.

$$\begin{aligned} P &\to d\,P' \\ P' &\to d\,P' \mid c\,Q \end{aligned}$$

Q first checks if any other symbol is present in the input. If not, we have an input of the form d^+c. Otherwise, the next symbol must be c.

$$Q \to \varepsilon \mid c\,R$$

Now, R can be either the second set of declarations followed by the computations, or the remaining part of the computation (may be empty).

$$R \to A\,c\,B \mid B$$

A generates one or more declarations, whereas B generates zero or more computations.

$$\begin{aligned} A &\to d\,A' \\ A' &\to \varepsilon \mid d\,A' \end{aligned}$$

$$B \to \varepsilon \mid c\,B$$

(d) Prepare the LL(1) parsing table for your grammar of Part (c). From the table, argue that your grammar of Part (c) is actually LL(1). **[6]**

We start by computing the FIRST and FOLLOW values of the non-terminals.

$P \to d\,P'$	FIRST(P) = { d }	FOLLOW(P) = { \$ }
$P' \to d\,P' \mid c\,Q$	FIRST(P') = { d , c }	FOLLOW(P') = { \$ }
$Q \to \varepsilon \mid c\,R$	FIRST(Q) = { ε , c }	FOLLOW(Q) = { \$ }
$R \to A\,c\,B \mid B$	FIRST(R) = { ε , d , c }	FOLLOW(R) = { \$ }
$A \to d\,A'$	FIRST(A) = { d }	FOLLOW(A) = { c }
$A' \to \varepsilon \mid d\,A'$	FIRST(A') = { ε , d }	FOLLOW(A') = { c }
$B \to \varepsilon \mid c\,B$	FIRST(B) = { ε , c }	FOLLOW(B) = { \$ }

From these, the LL(1) parsing table can be derived as follows.

	c	d	\$
P		$P \to d\,P'$	
P'	$P' \to c\,Q$	$P' \to d\,P'$	
Q	$Q \to c\,R$		$Q \to \varepsilon$
R	$R \to B$	$R \to A\,c\,B$	
A		$A \to d\,A'$	
A'	$A' \to \varepsilon$	$A' \to d\,A'$	
B	$B \to c\,B$		$B \to \varepsilon$

Since no cells in the LL(1) parsing table contains multiple entries, the grammar of Part (c) is LL(1).

4. Consider the C-style declaration of a structure along with variables of that structure type, as illustrated below.

```
struct xyz123 {
    int a, b, c;
    float d, e[20], f;
    char g[15];
} r[8], s, t;
```

A grammar for this construct is given below. For simplicity, the grammar deals only with the basic data types `char`, `int`, and `float`, and with variables and one-dimensional arrays of basic and structure types. In our grammar, S is the start symbol, D stands for a set of declarations, E generates one line of declaration, and L derives a comma-separated list of variables and one-dimensional arrays. The terminal symbols are the keywords **struct**, **char**, **int**, and **float**, valid names **id**, positive integers **num**, and the punctuation symbols opening and closing braces, opening and closing square brackets, comma, and semicolon.

$$
\begin{array}{lcl}
S & \rightarrow & \textbf{struct id } \{ \, D \, \} \, L \, \textbf{;} \\
D & \rightarrow & D\,E \mid E \\
E & \rightarrow & B\,L\,\textbf{;} \\
B & \rightarrow & \textbf{char} \mid \textbf{int} \mid \textbf{float} \\
L & \rightarrow & L\,\textbf{,}\,V \mid V \\
V & \rightarrow & \textbf{id} \mid \textbf{id [num]}
\end{array}
$$

This exercise deals with a syntax-directed-definition (SDD) scheme for computing the total memory requirement (also called the size or width) of a single declaration generated by S. To that effect, the total memory requirement for each line generated by E is computed. Subsequently, the total memory requirement for D is computed as the sum of the memory requirements of the components E. This sum is rounded up to the nearest multiple of eight, and is taken as the final size of the structure. The list L in the production for S uses this structure size for computing its own total memory requirement. Finally, the size of this L will be the size of S.

For example, assume that the size of each `int` is 4 bytes, of each `float` is 4 bytes, and of each `char` is 1 byte. Then, in the above example, the total size of `a`, `b`, and `c` is 4 + 4 + 4 = 12, the total size of `d`, `e`, and `f` is 4 + 80 + 4 = 88, and the size of `g` is 15. The sum is 12 + 88 + 15 = 115. Round this up to the nearest multiple of 8, which is 120. So the size of the structure is 120. Finally, the size for the entire declaration is the memory requirement for `r`, `s`, and `t`, and is equal to 120 × 8 + 120 + 120 = 1200.

(a) In order to perform the above size calculations, the non-terminals use one **inherited** attribute *basesize* (size of a single variable of basic or structure type) and one **synthesized** attribute *memreq* (the total memory requirement). For each variable or array generated by V, *memreq* is the size of that variable or array (use the `sizeof` operator to compute this). For each line (E), *memreq* is the total size of all the variables and arrays appearing in that line. Finally, *memreq* for D is the sum of the total memory requirements of all the lines it generates. These attributes are to be computed in a bottom-up manner. When the D in the body of the production for S is processed, *memreq* for this D is rounded up to the nearest multiple of eight, and passed as *basesize* to the following L. This L computes its *memreq* as in each line generated by E. Finally, the *memreq* for S will be the *memreq* of this L.

Complete the SDD against the productions in the respective spaces below. Do **not** use any global variable. **[10]**

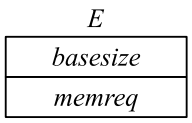
Production	Semantic Rule
$S \rightarrow$ **struct id** { D } L **;**	$L.basesize$ = roundup($D.memreq$) ; [You may use roundup(x) = (x + 7) / 8 × 8 (`int` calculations)] $S.memreq$ = $L.memreq$;
$D \rightarrow D_1\, E$	$D.memreq = D_1.memreq + E.memreq$;

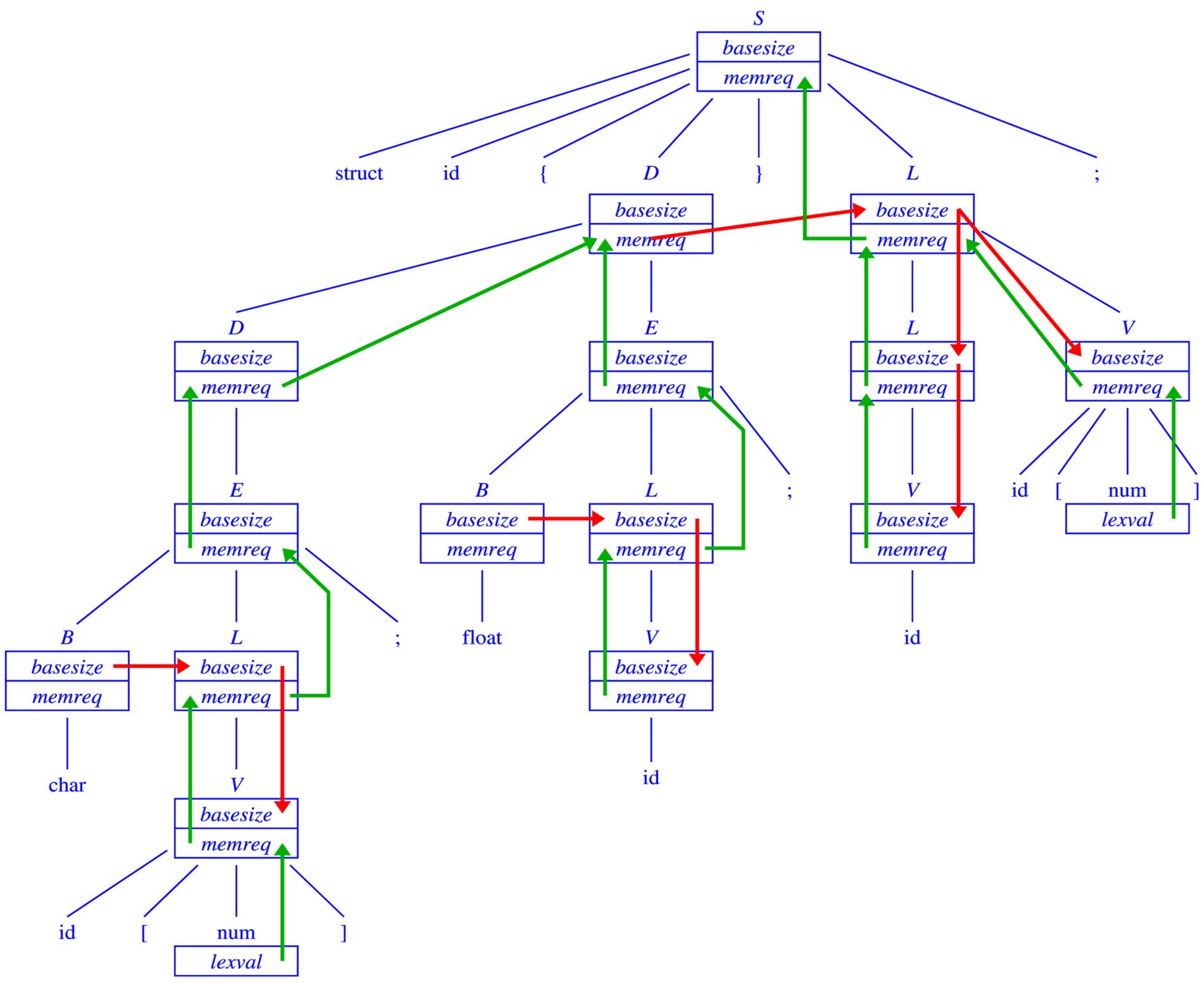
$D \rightarrow E$	$D.memreq = E.memreq$;
$E \rightarrow B\ L$ **;**	$L.basesize = B.basesize$; $E.memreq = L.memreq$;
$B \rightarrow$ **char**	$B.basesize =$ sizeof(**char**) ;
$B \rightarrow$ **int**	$B.basesize =$ sizeof(**int**) ;
$B \rightarrow$ **float**	$B.basesize =$ sizeof(**float**) ;
$L \rightarrow L_1$ **,** V	$L_1.basesize = L.basesize$; $V.basesize = L.basesize$; $L.memreq = L_1.memreq + V.memreq$;
$L \rightarrow V$	$V.basesize = L.basesize$; $L.memreq = V.memreq$;
$V \rightarrow$ **id**	$V.memreq = V.basesize$;
$V \rightarrow$ **id [num]**	$V.memreq = V.basesize \times$ **num**.$lexval$;

(b) Draw the dependency graph (along with the parse tree) for the following declaration.

```
struct student {
    char name[60];
    float cgpa;
} AbhijitHazra, CS[160];
```

Do **not** annotate the parse tree (by computing the attribute values). Only show the vertices (embedded on the parse tree) and the directed edges of the dependency graph. Clearly show the source and the destination of each edge. You may draw a non-terminal node as illustrated below (right). **[9]**





Rough Work

Rough Work

Rough Work