S.No.:01SKCS_ABCDEF_070624

MADE EASY
India's Best Institute for IES, GATE & PSUs

Delhi | Bhopal | Hyderabad | Jaipur | Pune | Bhubaneswar | Kolkata

Web: www.madeeasy.in | **E-mail:** info@madeeasy.in | **Ph:** 011-45124612

COMPLIER DESIGN

COMPUTER SCIENCE & IT

Date of Test : 07/06/2024

ANSWER KEY ➤

1.	(b)	7.	(d)	13.	(d)	19.	(d)	25.	(a)
2.	(b)	8.	(a)	14.	(b)	20.	(b)	26.	(d)
3.	(d)	9.	(c)	15.	(b)	21.	(d)	27.	(d)
4.	(d)	10.	(d)	16.	(b)	22.	(c)	28.	(d)
5.	(c)	11.	(d)	17.	(d)	23.	(c)	29.	(c)
6.	(a)	12.	(a)	18.	(c)	24.	(c)	30.	(a)

DETAILED EXPLANATIONS

1. (b)

- **Lexical analyzer** scan the source code as a stream of characters and counts it into meaning full lexemes.
- **Syntax analyzer** checks the token arrangement against the source code grammar.
- **Semantic analyzer** check whether the parse tree constructed follows the rules of language.
- **Code optimizer** do code optimization of the intermediate code.

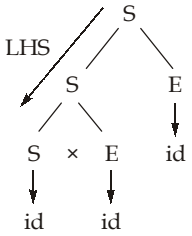
2. (b)

$S \rightarrow S \times E \mid E$

$E \rightarrow F + E \mid F$

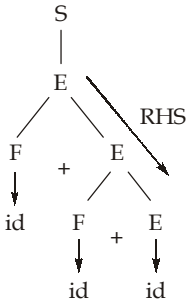
$F \rightarrow id$

1. For expression "id × id × id".



So, '×' is left associative.

2. For expression "id + id + id".



So, '+' is left associative.

3. (d)

- Live variable analysis needed in register allocation and deallocation.
- Basic block does not contain jump into middle of the block i.e. sequence of instruction where control enter the sequence at begin and exist at end.
- Three address code is linear representation of syntax tree.
- With triple, optimization cannot change the execution order but with indirect triple we can.

4. (d)

String given: "$xxxyxy$"

$$S \rightarrow ZZ \rightarrow \overset{\text{Handle } \{Z \rightarrow xZ\}}{\overbrace{ZxZ}} \rightarrow Zxy \rightarrow xZxy \rightarrow xxZxy \rightarrow xxxZxy \rightarrow xxxyxy$$

- ZxZ is not handle i.e. cannot reduce to any variable.
- Zxy is not handle i.e. cannot reduce to any variable.
- $xZxy$ is not handle i.e. cannot reduce to any variable.
- xZ is handle since xZ reduce to Z in next step.

5. (c)

```
int main ( )
①  ②  ③④
{
⑤
    int m = 10 ;
    ⑥ ⑦⑧⑨⑩
    int n , n1 ;
    ⑪⑫⑬⑭⑮
    n = ++ m ;
    ⑯⑰⑱⑲⑳
    n1 = m ++ ;
    ㉑㉒㉓㉔㉕
    n -- ;
    ㉖㉗㉘
    -- n1 ;
    ㉙㉚㉛
    n -= n1 ;
    ㉜㉝㉞㉟
    printf ( "%d" , n ) ;
      ㊱ ㊲ ㊳ ㊴㊵㊶㊷
    return 0 ;
      ㊸ ㊹㊺
}
㊻
```

Number of tokens are 46.

6. (a)

aaab → 2

cc → 3

abbb → 2

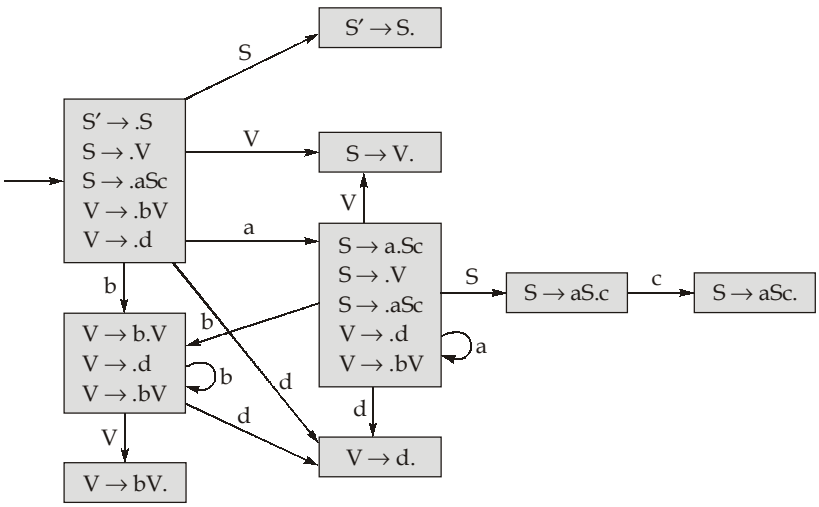
7. (d)

In case of y, the translation on RHS of production is defined in terms of translation of nonterminal on the left. So, y is inherited.

In case of x, translation of nonterminal on the left side of production is defined as function of translation of non-terminals on right hand side. So, x is synthesized

8. (a)

SLR Parser:



Zero inadequate states since no SR conflict or RR conflict is present.

9. **(c)**

Static scoping means that x refers to the x declared innermost scope of declaration. Since 'h' is declared inside the global scope, the innermost x is the one in the global scope (it has no access to the x's in 'f' and 'g', since it was not declared inside them), so the program prints 23 twice.

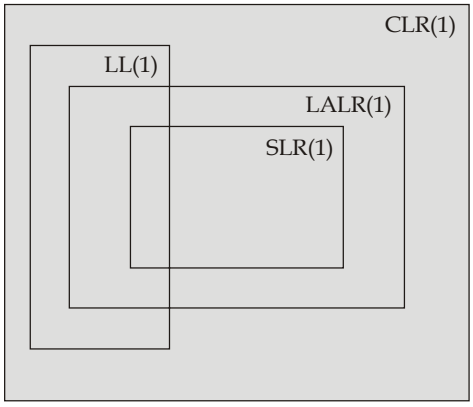
Dynamic scoping means that x refers to the x declared in the most recent frame of the call stack. 'h' will use the x from either 'f' or 'g', whichever one that called it so the program would print 22 and 45.

10. **(d)**

Analysis phase {lexical analysis, syntax analysis, semantic analysis} is followed by synthesis phase {intermediate code generation, code optimizer, machine code generation}.

11. **(d)**

Relation between LL(1), SLR(1) and CLR(1) and LALR(1) given below:



S_1 is false, S_2 is true and S_3 is false.

12. **(a)**

Given expression: ((3 × 2) – (10 + (5 – ((7 × 6) / 3))))

= (6 – (10 + (5 – (42/3))))

= (6 – (10 + (5 –14)))

= (6 – (10 – 9))

= (6 – (1))

= 5

13. **(d)**

Given grammar:

$A \to B \mid a \mid CBD$

$B \to C \mid b$

$C \to A \mid c$

$D \to d$

$\equiv$

$A \to A \mid a \mid ABD \mid c \mid b \mid cBD$

$B \to C \mid b$

$C \to A \mid c$

$D \to d$

Removing left recursion from $A \to A \mid a \mid b \mid c \mid ABD \mid cBD$

$A \to aA' \mid bA' \mid cA' \mid cBDA'$

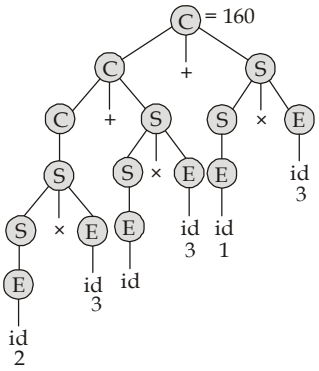
$A' \to BDA' \mid \in$

$B \to C \mid b$

$C \to A \mid c$

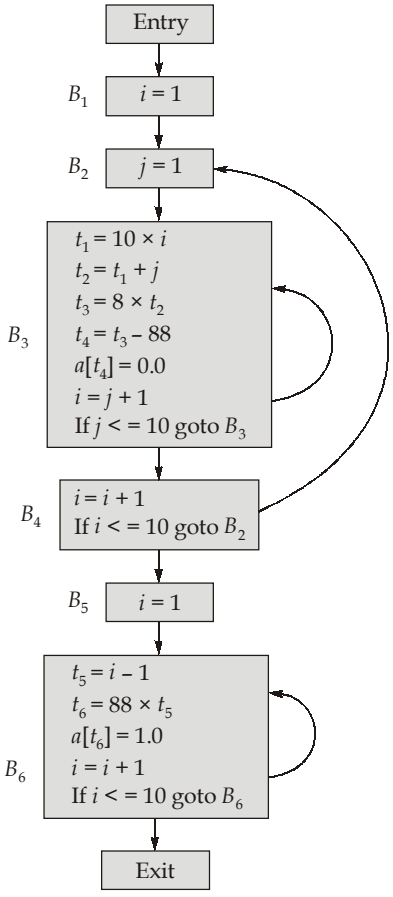
$D \to d$

14. (b)



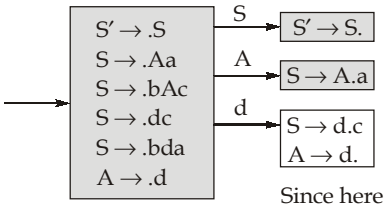
15. (b)

Control flow graph will be:



16. (b)

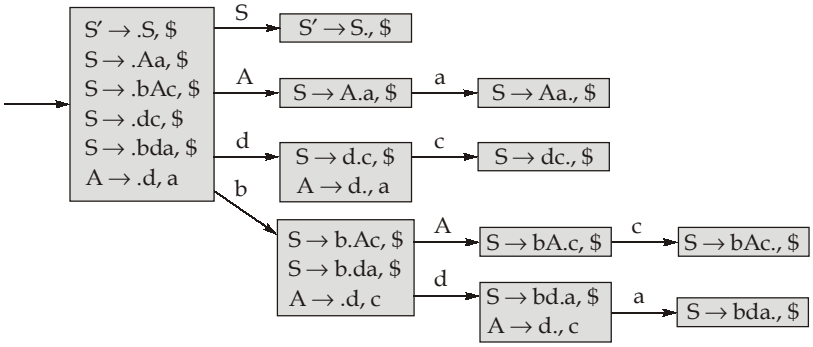
Check for SLR(1):



Since here shift-reduce present so check
shift(c) ∩ follow(A) = ϕ or not i.e. {c} ∩ {a, c} ≠ ϕ
Hence, cannot be SLR(1)

No need to design full DFA, check on each state.

Check for LALR:



Since no state present, which only differs is look ahead symbols, hence grammar has LALR(1) and LR(1) DFA with same state. So grammar is LR(1), LALR(1) but not SLR(1).

17. (d)

First(S) = {(, a, d} Follow(S) = {), b, $}
First(U) = {(, a, d} Follow(U) = {a, c,), b, $}
First(V) = {a, ∈} Follow(V) = {c,), b, $}
First(W) = {c, ∈} Follow(W) = {), b, $}

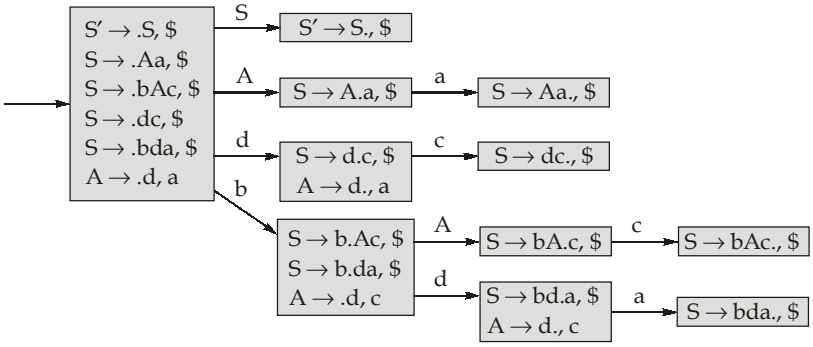
LL(1) Table:

M[T, t]	a	b	c	d	(	)	$
S	S → UVW			S → UVW	S → UVW		
U	U → aSb			U → d	U → (S)		
V	V → aV	V → ∈	V → ∈			V → ∈	V → ∈
W		W → ∈	W → cW			W → ∈	W → ∈

Six entries are missing.

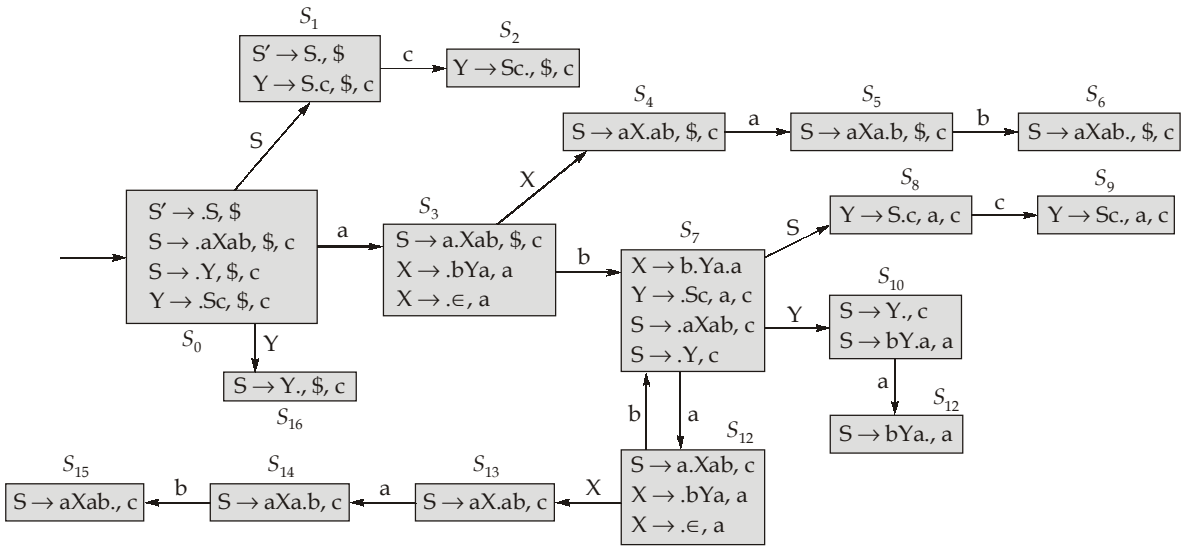
18. (c)

LR(1):



Since their is not state which only differs in look ahead symbol, hence given LR(1) DFA is LALR(1) DFA. Hence 11 states are needed.

19. (d)



So total 17 states are needed for LR(1) DFA.

20. (b)

21. (d)

- $S \rightarrow aabc \mid ab$
  There is left factoring in LL(1). Hence, not LL(1), but it is LL(2).
- Every regular language is LL(1) is true. There exist a regular grammar which is LL(1).
- Every regular grammar is LL(1) is false, because regular grammar may contain left recursion, left factoring, ambiguity.

22. **(c)**

$$\text{Follow}(S) = \{\$, a, b\}$$
$$\text{Follow}(A) = \{a, b\}$$
$$\text{First}(B) = \{a, b\}$$
$$\text{First}(S) = \{a, b, \in\} = \text{First}(A) = \text{First}(B)$$

23. **(c)**

To implement recursion, activation record should be implemented by providing dynamic memory allocation. This dynamic allocation is done from run-time stack. Heap is essential to allocate memory for data structures at run-time, not for recursion.

So, statement (a) and (c)are correct.

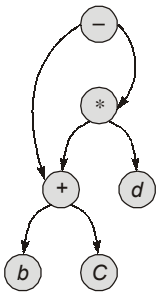
24. **(c)**

$$b = (b + c) \qquad d = (b + c) * d$$
$$d = b * d \qquad b = ((b + c) * d) - (b + c)$$

Final expression is

$$\Rightarrow \quad b = ((b + c) * d) - (b + c)$$

So, DAG representation for above expression is:



Number of nodes = 6

Number of edges = 6

25. **(a)**

- S_1 is correct.
- With triple, optimization cannot change the execution order but with indirect triple we can.

26. **(d)**

27. **(d)**

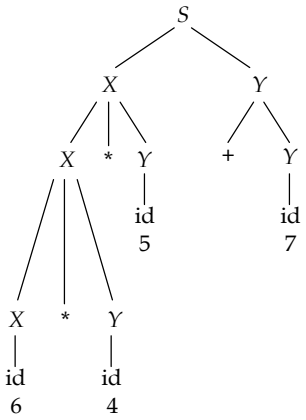
Lexemes are identified by the lexical analyzer as an instance of that token.

Hence only statement (d) is false.

28. **(d)**

- Statement S_1 and S_2 are correct.
- Statement S_3 is incorrect. Heap and stack both are present in main memory.

29. (c)



Output : 64 * 5 * 7 –

30. (a)

	FIRST	FOLLOW
S	$\{a, b, \in\}$	$\{a, b, \$\}$
A	$\{a, b, \in\}$	$\{a, b\}$
B	$\{a, b, \in\}$	$\{a, b, \$\}$

LL(1) Parsing table:

	a	***b***	**$**
S	$S \to aAbB$ $S \to \in$	$S \to bAbB$ $S \to \in$	$S \to \in$
A	$A \to S$	$A \to S$	
B	$B \to S$	$B \to S$	$B \to S$

■■■■