CLASS TEST

S.No. : 01 SK_CS_ABCD_240523

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

COMPILER DESIGN

COMPUTER SCIENCE & IT

Date of Test : 24/05/2023

ANSWER KEY ➤

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

DETAILED EXPLANATIONS

1. (a)

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

2. (c)

x is inherited.

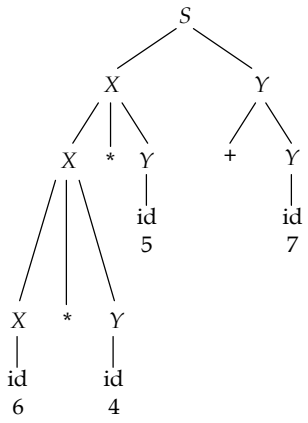
y is synthesized.

3. (d)

4. (c)

Lexical analyzer → Syntax analyzer → Semantic analyzer → Intermediate code → Code optimizer.

5. (c)



Output : 64 * 5 * 7 –

6. (d)

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

7. (d)

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

Hence only statement (d) is false.

8. (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.

9. (d)

A Grammar G is said to be operator grammar if

(a) it does not contain null production.

(b) it does not contain 2 adjacent variable on right hand side.

So, both G_1 and G_2 are not operator grammar.

10. **(c)**

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

11. **(d)**

$$\text{FOLLOW(S)} = \{a, b, c, e, \$\}$$

Total 5 elements are there.

12. **(c)**

Only option (c) is false since the GoTo part remains same.

13. **(a)**

LALR parser is more powerful thus option (b) is false.
CLR parser is more powerful than LALR so option (c) is also false.
Option (d) is also false with same reason which described above.

14. **(a)**

Compiler will remove all a's and replace it with $(x + 1)$ Thus, program would become like given below:

```
#include <stdio.h>
int x = 2;
void b( )
{
    x = (x + 1);
    printf("%d\n", x);
}
    void c( )
{
    int x = 1;
    printf("%d\n", (x + 1));
}
void main( ) {b( ); c( );}
```

Thus, output will be:

3
2

15. **(a)**

It's constant folding.

Constant folding: Replacing the value of expression before compile time.

Here the value of 4 * 2.14 is replaced by 8.56. So it's constant folding.

16. **(d)**

LL(1) grammar are free from left recursion, ambiguity left factoring.
So option (a), (b) and (c) are the pre requisite to be LL(1) grammar.

17. **(d)**

No grammar with empty production can be LR(0). But empty rules are allowed in regular grammar as the only condition for a grammar to be regular is to be either left linear or right linear.

18. (b)

LR(0) $\subset$ LR(1) = LR(2) ..... LR(k) = LR(k + 1); for k >= 1 : LR(k) = LR(k + 1) so, languages of grammars parsed by LR(2) parsers is not a strict super set of the languages of grammars parsed by LR(1) parsers although they both are same. If you compare their grammars then LR(0) $\subset$ LR(1) $\subset$ LR(2) ..... LR(k) $\subset$ LR(k + 1).

19. (b)

It is because we construct LALR parsing table by merging states of CLR(1) which are only separated by look-aheads. In doing so we may merge states which may introduce R-R conflicts.

20. (b)

- Viable prefix is nothing but stack content in LR Parsing. In this question just check option if it is visible in stack while doing parsing or not.
- A short trick to solve such question is to check, if there is already reduce handle in stack then stack can't store at any symbol after that reduce handle symbol.

Option (b) is only viable prefixes.

21. (d)

(a) Grammar is not ambiguous - true
(b) Priority of + is > than *, since + comes lower in parse tree - true.
(c) Right associativity – true.

22. (d)

Left recursion is there in the production Rule 2 and 3.

Rule 2 :

$A \rightarrow Aba \mid b$

After removing left recursion :

$A \rightarrow bA'$
$A' \rightarrow baA' \mid \in$

Rule 3 :

$B \rightarrow BaA \mid a$

After removing left recursion :

$B \rightarrow aB'$
$B' \rightarrow aAB' \mid \in$

Hence, the grammar after resolving left recursion is,

$S \rightarrow AB \mid BA$
$A \rightarrow bA'$
$A' \rightarrow baA' \mid \in$
$B \rightarrow aB'$
$B' \rightarrow aAB' \mid \in$

23. (d)

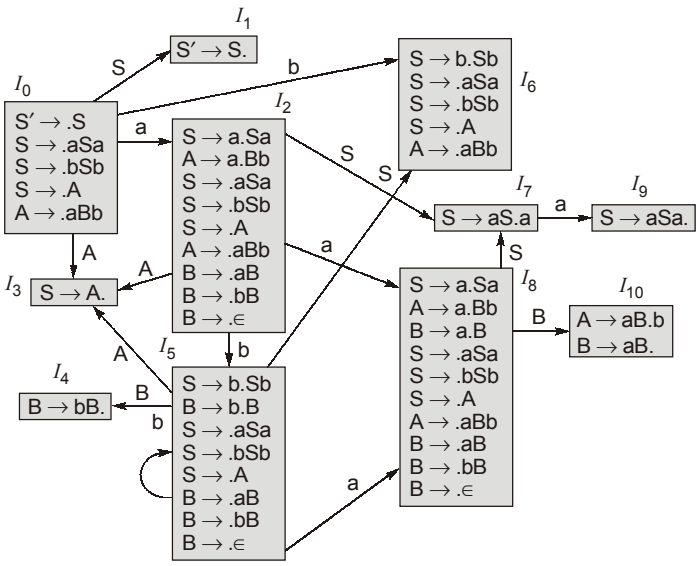
LL (1) passing table :

	a	b	$
S	$S \to aSa \mid S \to A$	$S \to bSb$	
A	$A \to aBb$		
B	$B \to aB$	$B \to bB \mid B \to \in$	

Since the production [$S \to aSa$ and $S \to A$] are in the same tupple [S, a] of the parsing table. Hence grammar isn't LL (1).

Similarly, the productions [$B \to bB$ and $B \to \in$] are in the same tupple [B, b] of the parsing table.

LR (0) parsing table :



This grammar is not LR (0) as well because there is shift reduce conflict is canonical item I_{10}.

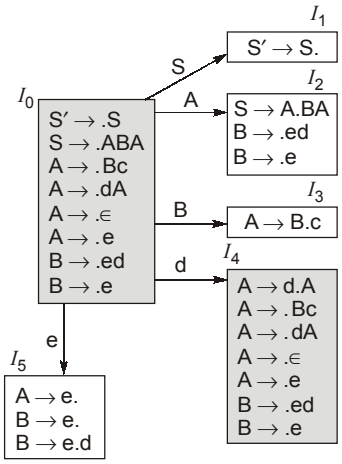
Note : LL(1) and LR(0) grammar can't contain null production.

24. (b)

- A token is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit e.g. a particular keyword or a sequence of input characters denoting an identifier.
- A pattern is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword.
- A lexeme is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

25. (c)

Intermediate states of SLR (1) parser :



Considering the state I_5, it can be said that an intermediate state has both shift reduce as will as reduce-reduce conflict

I_5 : A → e., B → e., B → e.d

[I_5, d] - shift entry for B → e.d

[I_5, e, d, $] - reduce entry for B → e.

[I_5, e, $] - reduce entry for A → e.

26. (c)

Consider each statement :

S_1 : Consider the grammar S → P*y* P, P → g. This grammar is both left as well as right recursive but it's not ambiguous.

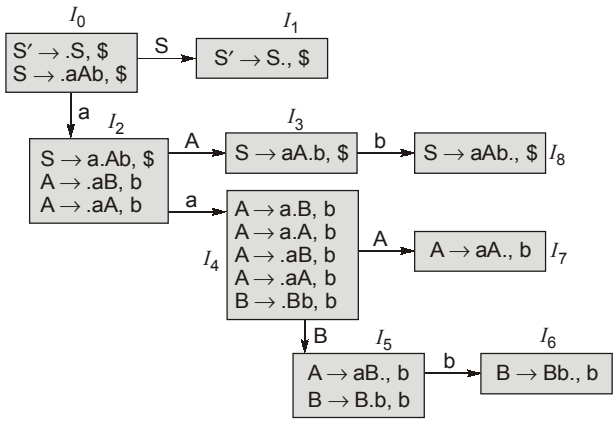
S_2 : Every regular set can be converted to right linear grammar. All right linear grammar are LL (1). A grammar which is LL (1) will also be LR (1).

S_3 : Intermediate codes are generated by the compiler to enhance the portability of the front end of the compiler.

S_4 : Left recursive grammars can not be used by Recursive Descent Parsers.

27. (d)

Constructing CLR (1) parser :



In canonical item, I_5, the production A → aB. Will have its parsing table entry at [I_5, b]. Similarly, for production, B → B.b, the entry will be at [I_5, *b*], which shows a shift-reduce conflict.

Hence grammar is not CLR (1).

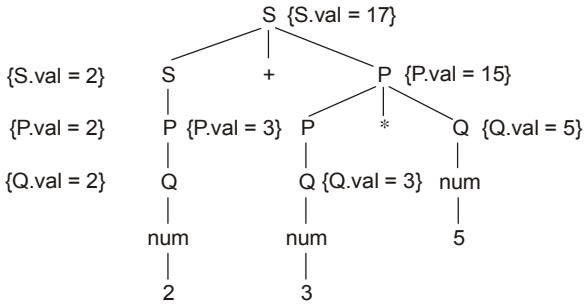
28. (b)

The code is as follows,

$$\begin{aligned} t_1 &= a + b \\ t_2 &= -c \\ t_2 &= t_2 + d = [-c + d] \\ t_1 &= t_1 * t_2 = [a + b] * [-c + d] \\ t_1 &= t_1 + e = ([a + b] * [-c + d]) + e \\ S &= t_1 \\ S &= [(a + b) * (-c + d)] + e \end{aligned}$$

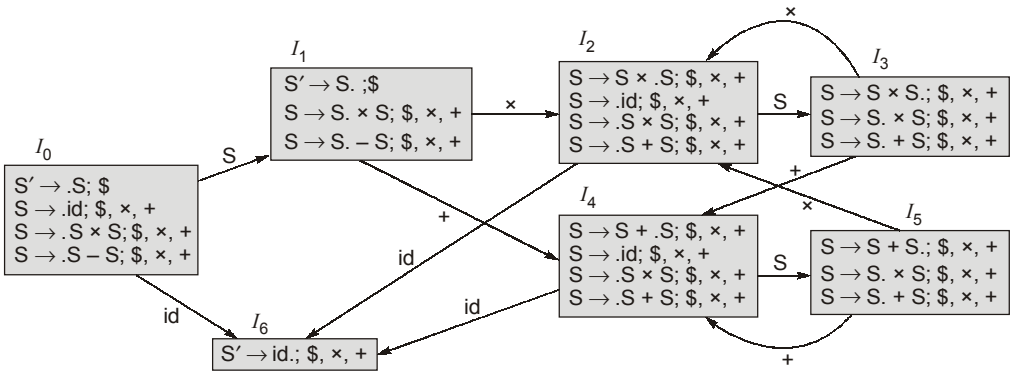
29. (a)

Given SDT is S-attributed and hence L-attributed too. Since all translations are appended at end and attributes are synthesis, hence both L-attributed and S-attributed approach evaluates to same value.



30. (a)

Construction a LALR (1) parser :



As per the DFA, it can be observed that states I_3 and I_5 have shift reduce conflicts. Since, they are fed to YACC, hence it will be resolved in the favor of shift.

So, it can be concluded that both '×' and '+' have same precedence are right associative since we are reducing in spite of shifting.

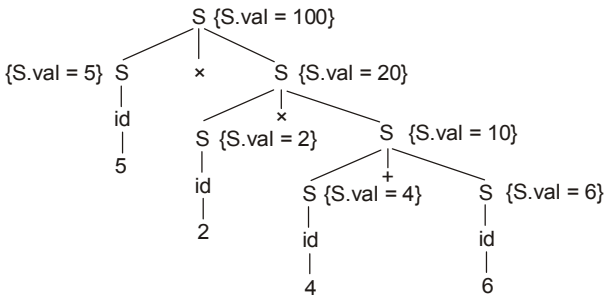
Expression : $5 \times 2 \times 4 + 6$

$\Rightarrow 5 \times 2 \times 10$

$\Rightarrow 5 \times 20$

$\Rightarrow 100$

Annotated Parse Tree :



Hence, the expression evaluates to 100.

■■■■