CLASS TEST

S.No. : 01 SK_CS_A+B_130422

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

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

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

COMPILER DESIGN

COMPUTER SCIENCE & IT

Date of Test : 13/04/2022

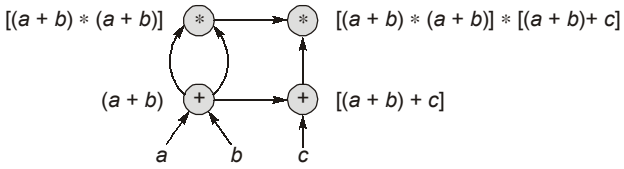
ANSWER KEY ➤

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

1. (d)

- If a grammar is LR (0) then it will always be SLR (1). It implies that if no conflict is present in LR (0) then SLR (1) also don't have any kind of conflict.
- Intermediate code tends to be machine independent code.

2. (c)



Hence, the expression DAG will be :

$[(a + b) * (a + b)] * [(a + b) + c]$

3. (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,

$S \rightarrow AB \mid BA$

$A \rightarrow bA'$

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

$B \rightarrow aB'$

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

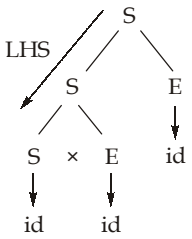
4. (b)

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

$E \rightarrow F + E \mid F$

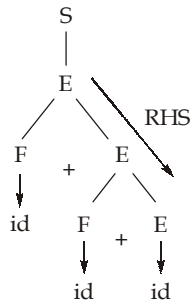
$F \rightarrow id$

1. For expression "id × id × id".



So, '×' is left associative.

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



So, '+' is left associative.

5. (d)

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

6. (d)

String given: "*xxxyxy*"

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

- 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.

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. (b)

A SDT is called S attributed if it has only synthesized attributed. L-attributed definitions contain both synthesized and inherited attributes.

9. (b)

By common sub expression we mean- the expression which is already computed but during compilation it again appears for computation. In constant folding - the expression which is having a constant value at the time of compilation it can be referred to its respective value or we can say that it can be replaced with its similar value. It is constant folding where the replacement is done during compilation time.

10. (a)

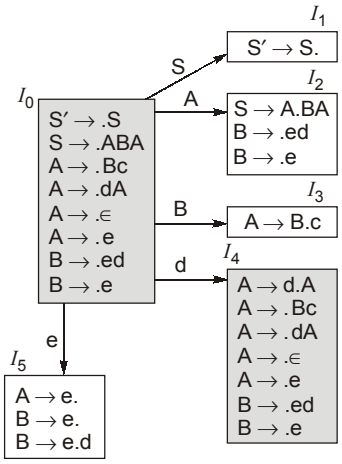
It's constant folding.

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.

11. (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.

12. (c)

From the table, the grammar can be concluded as :

S → P

P → QR | TQR

Q → q | ∈

R → r | ∈

T → x | y

String 'y' and 'xqr' can be parsed successfully with the help of stack and given LL (1) parsing table.

13. (b)

The code is as follows,

$$t_1 = a + b$$
$$t_2 = -c$$
$$t_2 = t_2 + d = [-c + d]$$
$$t_1 = t_1 * t_2 = [a + b] * [-c + d]$$
$$t_1 = t_1 + e = ([a + b] * [-c + d]) + e$$
$$S = t_1$$
$$S = [(a + b) * (-c + d)] + e$$

14. (d)

LL (1) passing table :

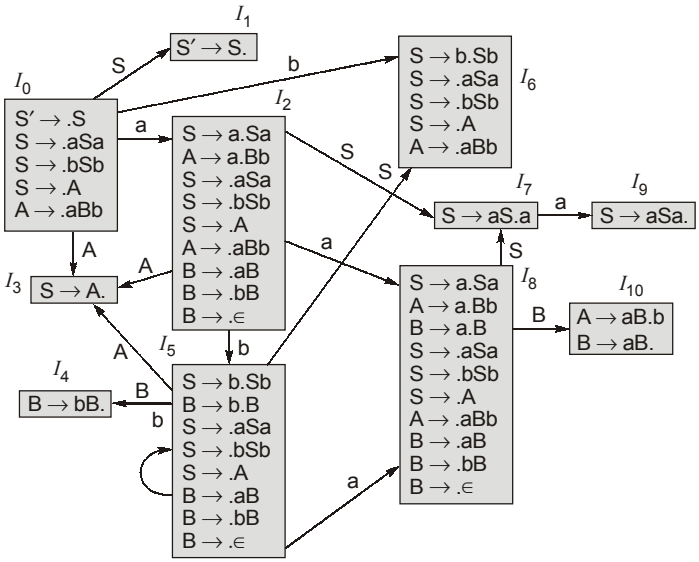
	a	b	$
S	S → aSa | S → A	S → bSb	
A	A → aBb		
B	B → aB	B → bB | B → ∈	

Since the production [S → aSa and S → A] are in the same tupple [S, a] of the parsing table. Hence

grammar isn't LL (1).

Similarly, the productions [B → bB and B → ∈] 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.

15. (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.

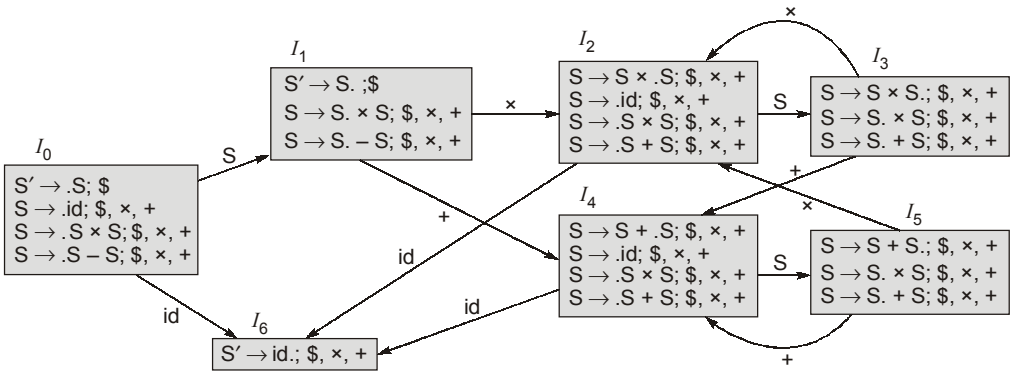
16. (b)

$r_1 = 6$
$r_2 = 7$
$r_3 = r_1 * r_2$
$r_4 = r_3 + r_1$
$r_5 = 8$
$r_4 = r_4 * r_1$
$r_4 = r_1 + r_4$
$r_4 = r_4 * r_2$
$r_3 = r_3 + r_1$
return r_3

So, total 5 registers are required to execute this program without spilling.

17. (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.

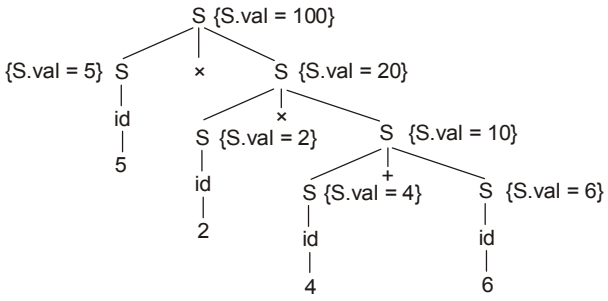
Expression : $5 \times 2 \times 4 + 6$

$\Rightarrow 5 \times 2 \times 10$

$\Rightarrow 5 \times 20$

$\Rightarrow 100$

Annotated Parse Tree :



Hence, the expression evaluates to 100.

18. (c)

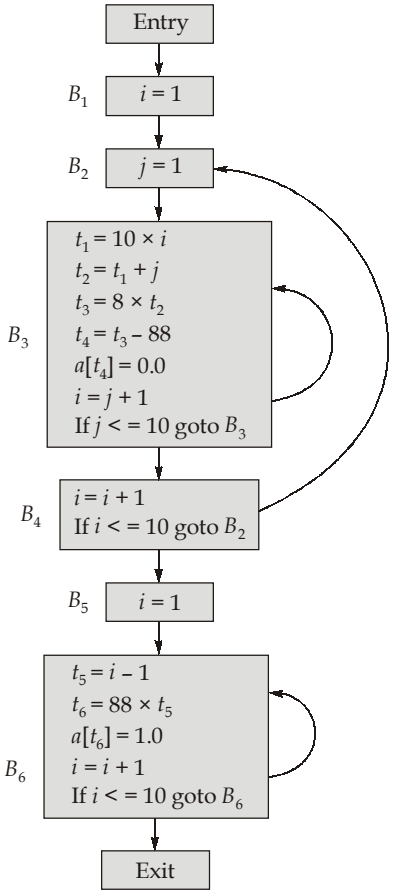
$$\begin{aligned}
\text{FIRST (C)} &= \text{FIRST (PF class id XY)} \\
&= \{\text{public}\} \cup \text{FIRST (F class id XY)} \\
&= \{\text{public}\} \cup \{\text{final}\} \cup \text{FIRST (class id XY)} \\
&= \{\text{public}\} \cup \{\text{final}\} \cup \{\text{class}\} \\
&= \{\text{public, final, class}\} \\
\text{FIRST (X)} &= \text{FIRST (Y)} \\
&= \{\text{implements}\} \cup \text{FOLLOW (C)} \\
&= \{\text{implements}\} \cup \{\$\} \\
&= \{\text{implements}, \$\} \\
\text{FIRST (Y)} &= \text{FIRST (implements I)} \cup \text{FIRST } (\in) \\
&= \{\text{implements}, \in\} \\
\text{FOLLOW (P)} &= \text{FIRST (F)} \\
&= \{\text{final}\} \cup \text{FIRST (class)} = \{\text{final, class}\}
\end{aligned}$$

19. (d)

Since for every 1 after fractional point represented by $1/2^i$. So, {B.val = 1/2}, then, for B_0=1B_1 lower bit from fractional side added to B.val i.e., {B_0.val = B_1.val/2+1/2}.
Finally $B_0 \rightarrow 0B_1$, old value divide by 2 i.e., {B_0.val = B_1.val/2}

20. (b)

Control flow graph will be:



21. (c)

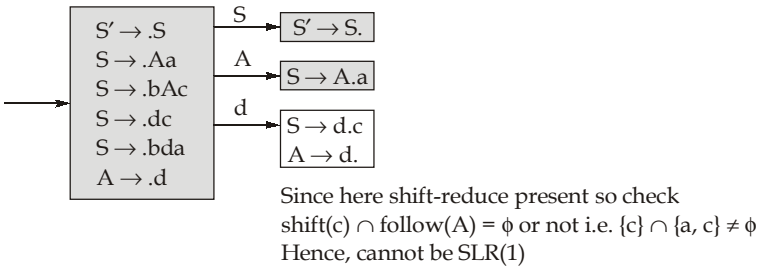
S_1 : Stack and heap is used for dynamic memory allocation.
S_2 : Heap and stack both are present in main memory.
S_3 : Access to heap memory variables is slower as compared to accessing variables allocated on stack. Because to access a heap memory variable we need access pointer variable first.
S_4 : In a multithreaded situation, each thread has its own stack and share a common heap memory.

22. (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.

23. (b)

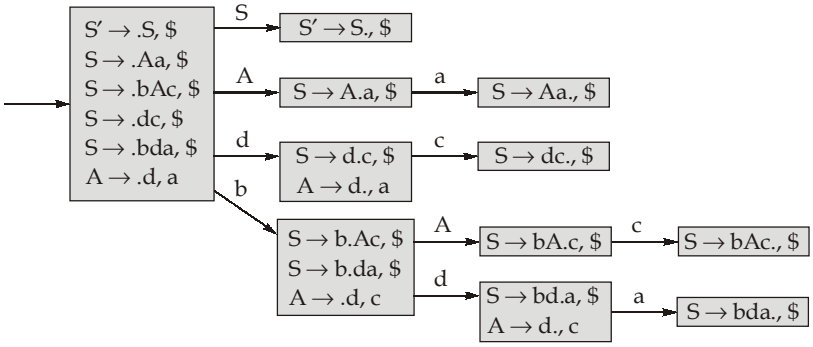
Check for SLR(1):



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).

24. (b)

Viable prefixes: Combination of non-terminal and terminal which can be in the stack during parsing is called viable prefix.

A handle X→ b is available. So whenever terminal b is encounter it is popped out before inserting new element into the stack.

Option (a) is wrong because before popping 'b' element 'aab' is inserted.

Option (b) is correct after 'b' no element has inserted.

So, option (c) and (d) also incorrect.

25. (c)

Any symbols $, ′, @ like that gives lexical error in C if put any where outside of string and the comment lines. They can not pass the lexer because lexer can not recognize them as a valid token.

- Program 1 gives lexical error because "@_" written outside comment.
- Program 2 having no lexical error but producing semantic error by syntax analyzer.
- Program 3 gives lexical error closing comment lines */ is missing.
- Program 4 gives lexical error because integer has been assigned an invalid octal number "09".

26. (c)

In static single assignment, each assignment to variable should be specified with distinct names.

$$t_1 = t_2 / t_3$$
$$t_4 = t_5 + t_6$$
$$t_7 = t_4 - t_1$$
$$t_8 = t_6 + t_7$$

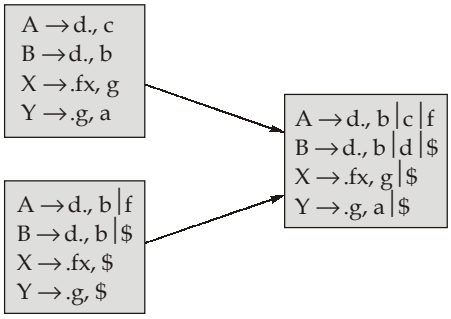
$$t_9 = t_{10} + t_7$$
$$t_{11} = t_9 + t_2$$

So total 11 variables.

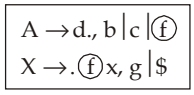
27. (b)

- Type checking is done during semantic analysis phase.
- Target code generation is machine specific.
- SDT with only synthesized attribute does not have any cyclic dependency. So always have order of evaluation.
- Symbol table is constructed during the lexical, syntax and semantic analysis phase.

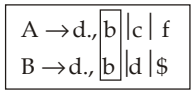
28. (c)



I. Can be merged but will results in S-R conflict is true because of productions.



II. Can be merged but will results in R-R conflict is true because of productions.



2-different production will be in the same entry.

III. Goto of f leading to SR conflict is true.

IV. Cannot be merged since look-a-heads are different false, because merging does not depend on look-a-head.

29. (b)

As we know,

$A \rightarrow A\alpha \mid B$

After eliminating LR

$A \rightarrow BR$

$R \rightarrow \alpha R \mid \in$

In the given case

$$\alpha = Y \{X.x = f(X.x, Y.y)\}$$

Final production after eliminating will be

$X \rightarrow Z \{X.x = g(Z.z)\}\ R$

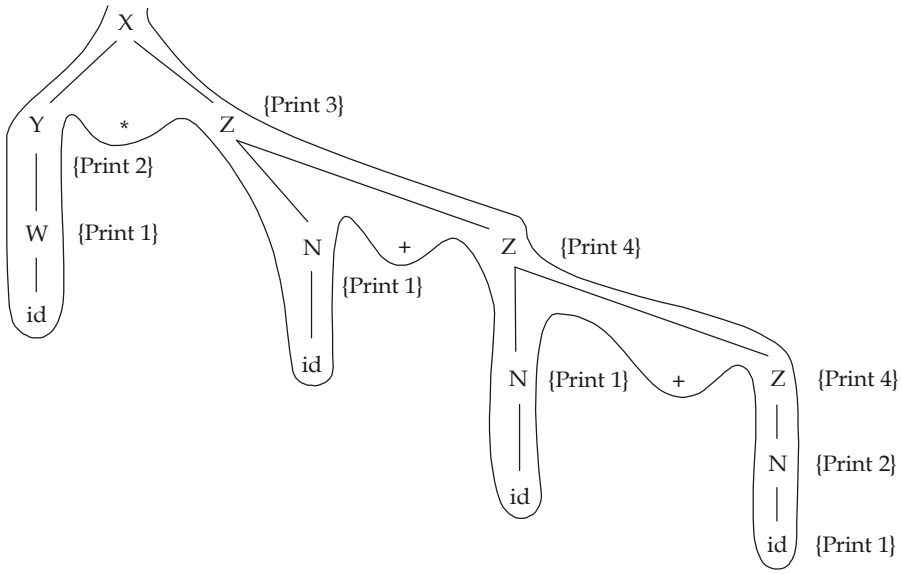
$R \rightarrow Y \{R.x = f(X.x, Y.y)\}\ R$

$R \rightarrow \in$

So, option (b) is correct.

30. (a)

Given input id * id + id + id



Output → 121112443

■■■■