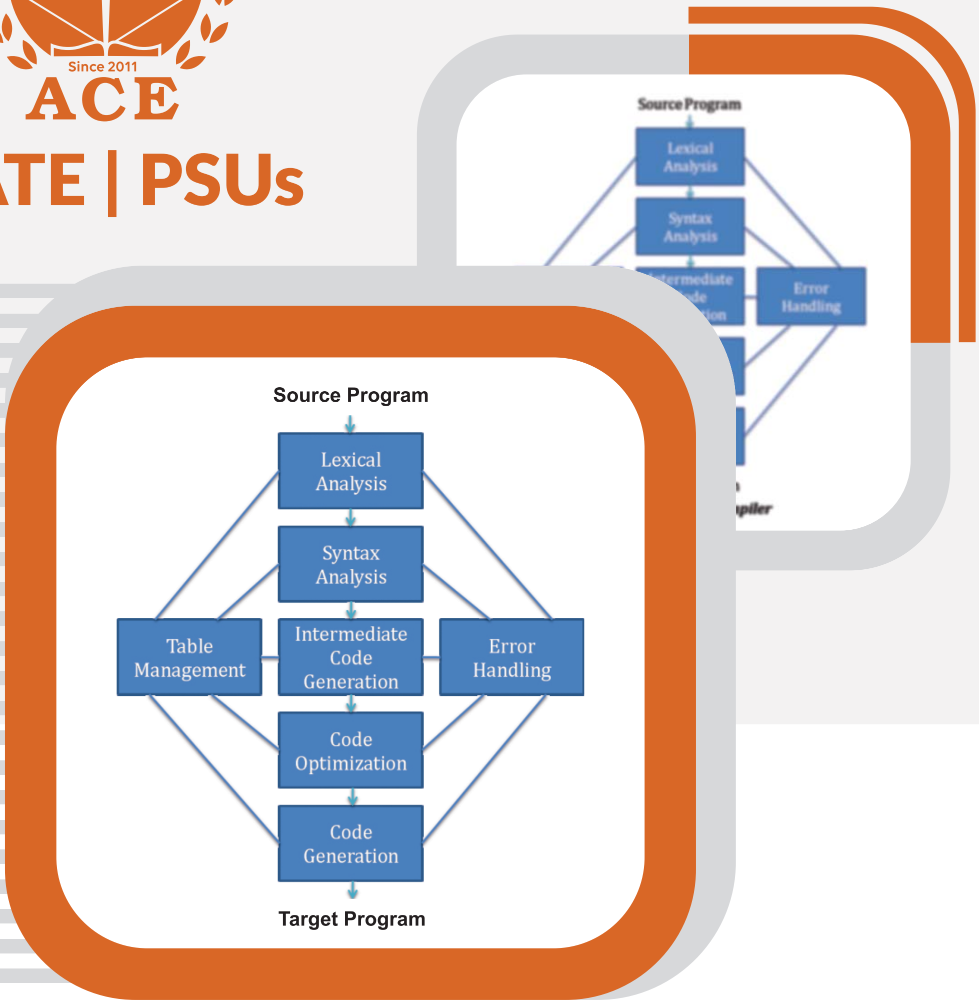
ACE ENGINEERING PUBLICATIONS

Since 2011

ACE

GATE | PSUs

COMPUTER SCIENCE & INFORMATION TECHNOLOGY



COMPUTER SCIENCE & INFORMATION TECHNOLOGY

COMPILER DESIGN

Volume-1 : Study Material with Classroom Practice Questions

Compiler Design

(Solutions for Vol-1_Classroom Practice Questions)

2. Lexical Analysis

01. Ans: (a)

Sol: Comments are deleted during lexical analysis, by ignoring comments.

02. Ans: (a)

Sol: The expansion of macro is done as the input tokens are generated during the lexical analysis phase.

03. Ans: (a)

Sol: As soon as an identifier identifies as lexemes the scanner checks whether it is a reserved word.

04. Ans: (c)

Sol: Type checking is a semantic feature.

05. Ans: (a)

Sol: Compiler identifies only Grammatical errors, but not logical & runtime errors.

06. Ans: (d)

Sol: A compiler that runs on one machine and generates code for another machine is called cross compiler.

07. Ans: (b)

Sol: The object code which is obtained from Assembler is in Hexadecimal, which is not executable, but it is relocated.

08. Ans: (b)

Sol: Syntax analysis can be expanded but the CFG describes the syntax becomes cumbersome.

09. Ans: (a)

Sol: The identifiers are entered into the symbol table during lexical analysis phase.

10. Ans: (a)

Sol: As I/O to an external device is involved most of the time is spent in lexical analysis

11. Ans: (a)

Sol: The lex utility creates a DFA from the regular definition.

12. Ans: (b)

Sol: The specifications of lexical analysis we write in lex language, when it run through lex compiler it generates an output called lex.yy.c.

13. Ans: (a)

Sol: Parenthesis matching cannot be done at the lexical analysis phase.

14. Ans: 20

15. Ans: 7

16. Ans: (b)

Sol: if , (, x , > =, y,), {, x, =, x, +, y, ; , }, else, {, x, =, x, –, y, ; , }, ; ,

17. Ans: (d)

Sol: All are tokens only.

18. Ans: (c)

Sol: Syntax tree is input to semantic analyzer. Character stream is input to lexical analyzer. Intermediate representation is input to code generation. Token stream is input to syntax analyzer.

19. Ans: 18

20. Ans: (b)

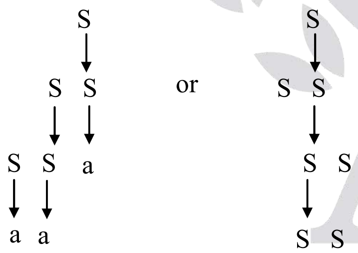
3. Parsing Techniques

01. Ans: (b)

Sol: As + is left associative the left most + should be reduced first

02. Ans: (d)

Sol:



$$S \rightarrow S^{k_1}SS^{k_2}SS^{k_3}....SS^{k_1}$$

$$\rightarrow \varepsilon^{k_1}a\varepsilon^{k_2}a\varepsilon^{k_3}a$$

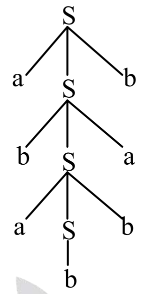
So the sentence has an infinite number of derivations.

03. Ans: (a)

Sol: The grammar which is both left and right recursive is always ambiguous grammar.

04. Ans: (d)

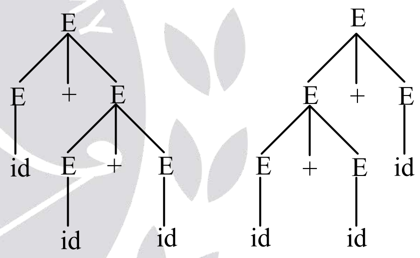
Sol:



Hence the option (d) is correct.

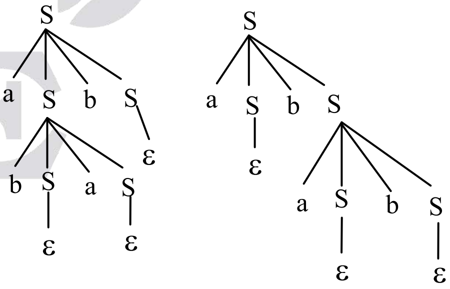
05. Ans: 2

Sol:



06. Ans: (c)

Sol:



07. Ans: (a)

Sol: S →Ad → Sad is indirect left recursion.

08. Ans: (c)

Sol: The production of the form $A \rightarrow A\ \alpha/\beta$ is left recursive, and can be eliminated by replacing with

$A \rightarrow \beta A^1$

$A^1 \rightarrow \alpha\ A^1/\varepsilon$

09. Ans: (d)

Sol: ↑ is least precedence and left associative

+ is higher precedence and right associative

10. Ans: (c)

Sol: Precedence from low to high is ↑, +, id.

11. Ans: (b)

Sol: – > *

12. Ans: 144

Sol: 3–2*4$2*3$2

1*4$2*3$2

1*16*9

16*9

= 144

13. Ans: (b)

Sol: Rule ‘a’ evaluates to 4096

Rule ‘b’ evaluates to 65536

Rule ‘c’ evaluates to 32

14. Ans: (c)

Sol: A bottom up parsing technique builds the derivation tree in bottom up and simulates a rightmost derivation in reverse

15. Ans: (d)

Sol: Operator precedence parser is a shift reduce parser.

16. Ans: (c)

Sol: first(s) = first(A) ∪ first(a) ∪ first (Bb)

= {d} ∪ {f, a} ∪{e, b}={a,b,d,e,f}

17. Ans: (d)

Sol: {$, s} both follow additional.

18. Ans: (c)

Sol: first(A) = {a, c}, follow(A) = {b, c}

first(A) ∩ follow(A) = {c}

19. Ans: (d)

Sol: Follow(B) = First(C)∪First(x) ∪ Follow (D)

= {y, m}∪ {x} ∪ Follow(A) ∪First(B)

= {y, m, x} ∪ {$} ∪ {w, x}

= {w, x, y, m, $}

20. Ans: (a)

Sol: Follow (S) = {$}

Consider S → [SX]

Follow (S) = First (X)

= {+,–, b} ∪ {]}

= {+, – , b,]}

Consider X → + SY

Follow(S) = First (Y)

= {–} ∪ Follow(X)

= {–} ∪ {c,]}

= {–, c,]}

Consider Y → – S X c

Follow(S) = First(X)

= {+, –, b} ∪ First(c)

= {+, –, b, c}

∴ Follow(S) = {+, – , b, c,], $}

21. Ans: (c)

Sol: Follow (T) = {+, $}

First (S) = {a, +, ε}

∴ Follow (T) ∩ First (S) = {+}

22. Ans: (d)

Sol: Follow(A)=first(B)∪Follow(S) ∪ Follow(B)

={e} ∪ {f} ∪ {c, d}={c,d,e,$}.

23. Ans: (d)

Sol: Follow (S) = { $, a,d}

Follow (A) = {a}

Follow (B) = {a,d}

Follow (C) = {$, a, d}

24. Ans: (c)

Sol: The predictive parsing table.

	identifier	*	$
<expression>	<expression>→ <factor><rest>		
<rest>		<rest>→ *<expression>	<rest>→ε
<factor>	<factor> → identifier		

25. Ans: (c)

Sol: The grammar is not LL(1), as on input symbol a there is a choice.

The grammar is not LL(2), as input ab there is a choice.

The grammar is LL(3) as on input abc there is not choice.

26. Ans: (c)

Sol: To distinguish between

S → if expr then stmt

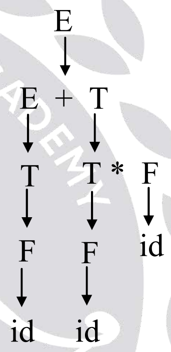
& S → if expr then stmt else stmt

We need a look ahead of 5 symbols.

27. Ans: (c)

Sol: * has a higher precedence than +.

Consider



28. Ans: (a)

Sol: A left recursive grammar cannot be LL(1).

29. Ans: (c)

Sol: A → ε production is added in 'A' row and Follow(A) column.

30. Ans: (d)

Sol: S→aSbs and S→ε both appear in 'S' row and 'a' column.

31. Ans: (b)

Sol: The first 2 symbols of 'S' production is distinct hence the grammar is LL(2).

32. Ans: (d)

Sol: The rightmost derivation is
<accumulated_sum>→<accumulated_sum> *<number>
→<accumulated_sum>+<number>* number
→ number + number * number

33. Ans: (d)

Sol: An operator grammar is ε-free grammar and no two non terminals are adjacent.

34. Ans: (c)

Sol: An operator grammar is 'ε' free grammar and no two non-terminals are adjacent.

35. Ans: (c)

Sol: An operator grammar is 'ε'-free grammar and has no two adjacent non-terminals.

36. Ans: (d)

Sol: As per normal HLL rules exponentiation is right associative where as –, +, * are left associative.

37. Ans: (d)

Sol: Lead (S) = {a} ∪ {c} ∪ Lead (B) ∪ {d}
= { a,c,d,e}

38. Ans: (b)

Sol: Trail (E) = {+} ∪ Trail(T)
= {+, *} ∪ Trail(F)
= {+, *,), id}

39. Ans: (b)

Sol: Lead (E) >+ and lead (E) contains {+, ↑, id}

40. Ans: (d)

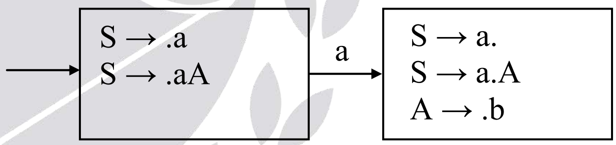
Sol: Possible relations with 'c' are d>c and c>$ only.

41. Ans: (b)

Sol: The grammar E → E + E/a can have an operator precedence parser but not an LR parser.

42. Ans: (a)

Sol: The grammar
E → E + T | T, T → i
is left recursive. So it is not LL(1) but is LR(0). So (a) is true & (b) is false.
The grammar
S → a | aA
A → b
has the LR(0) machine



Hence not LR(1) but is SLR(1).

43. Ans: (d)

Sol: The grammar
E → E +E | E ∗ E | i
Can have a shift reduce parser if we use the precedence and associativity of operations. The operator precedence technique works with some ambiguous grammars.

44. Ans: (d)

Sol: The grammar

S → a | A, A → a

is neither LL(1) nor LR(0) & is ambiguous.

No ambiguous grammar can be LL or LR.

45. Ans: (d)

Sol: No ambiguous grammar can be LR(1).

46. Ans: (c)

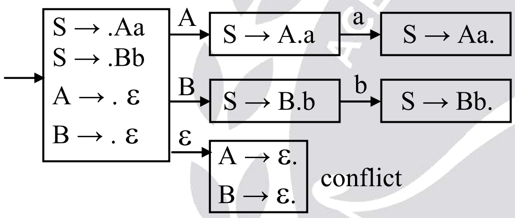
Sol: The grammar

S → Aa | Bb

A → ε

B → ε is LL(1) but not LR(0)

The LR(0) machine has a conflict.



The grammar is

S → a | ab

Is LR(2) & not LR(1).

47. Ans: (d)

Sol: Every LR(0) grammar is SLR(1)

Every SLR(1) grammar is LALR(1)

Every LALR(1) grammar is LR(1)

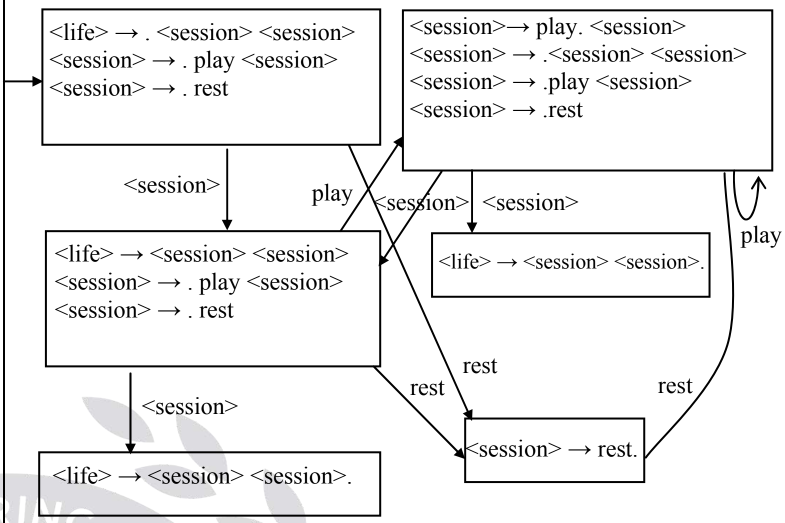
The grammar S → a is both LL(2) & LR(0) trivially.

48. Ans:(b)

Sol: Every LL(1) is LR (1)

49. Ans: (a)

Sol: The LR(0) machine for the grammar



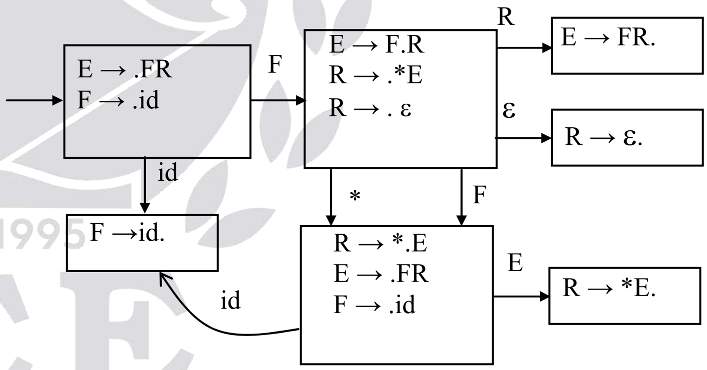
50. Ans: (b)

Sol: The LR(0) machine

E → FR

R → ∗E/ε

F → id



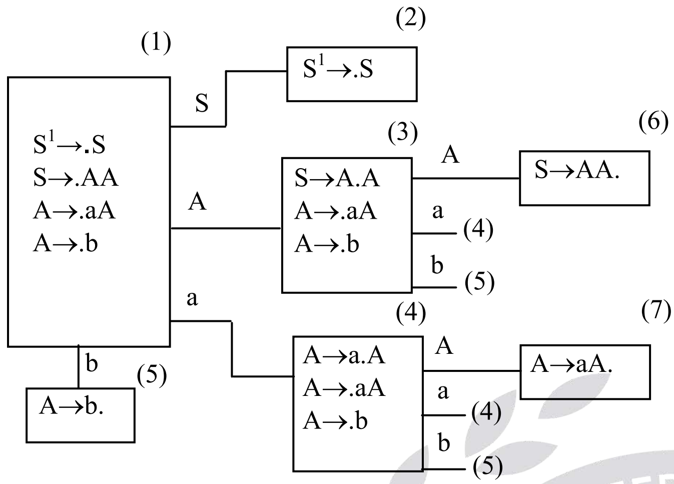
51. Ans: (b)

Sol:

S^1→.S
S→.SB
S→.A
A→.a

52. Ans: 7

Sol:

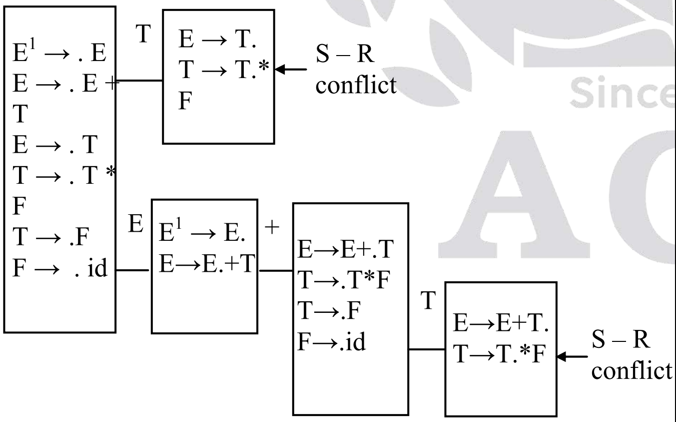


53. Ans: (c)

Sol: The given grammar is LR(0) and every LR(0) is LR(1).

54. Ans: 2

Sol:

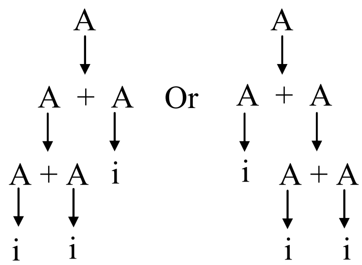


55. Ans: (a)

Sol: The grammar is LL(1), LR(0), SLR(1), LALR(1) & LR(1).

56. Ans: (d)

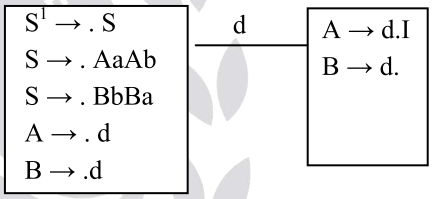
Sol: The grammar is ambiguous.



There are two derivation trees for the sentence i + i + i. As the grammar is ambiguous it cannot be LL or LR. So, (a), (b), (c), are ruled out. The answer is (d).

57. Ans: 2

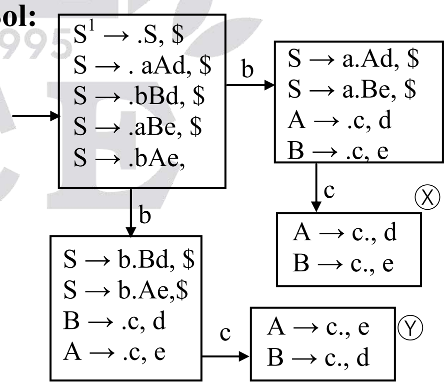
Sol: The LR(0) items of the grammar is



Reduce – Reduce conflict.

58. Ans: (a)

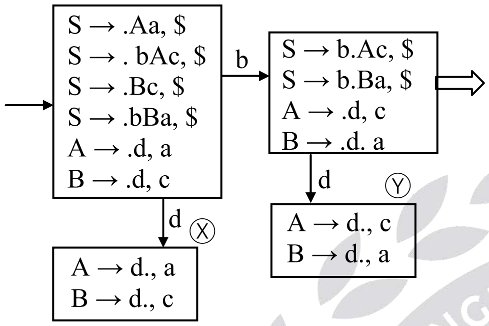
Sol:



Consider the partial LR(1) machine shown above. The states Ⓧ & Ⓨ have a common core. However if we merge the sates to obtain the LALR(1) machine we will end up with conflicts. So the grammar is LR(1) but not LALR(1).

59. Ans: (a)

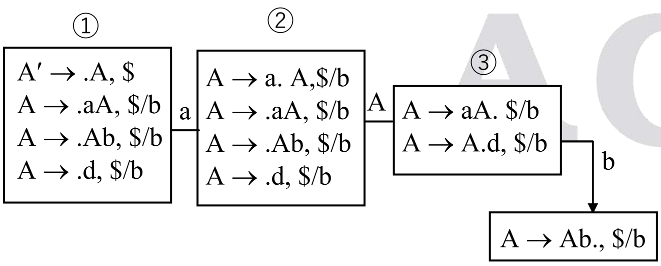
Sol:



Consider the partial LR(1) machine above. The states Ⓧ&Ⓨ have a common core but different look ahead sets. If we merge Ⓧ&Ⓨ So obtain the LALR(1) a conflict arise.

60. Ans: (b)

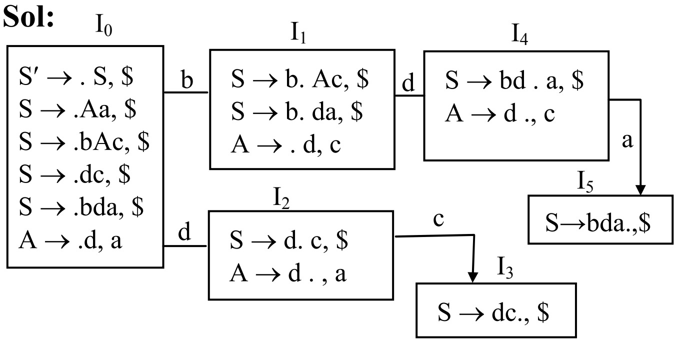
Sol: LR(1) items of the grammar is



Item 3 has Shift-Reduce conflict.

61. Ans:(d)

Sol:



As there is no conflicts the grammar is in LALR(1).

62. Ans: (c)

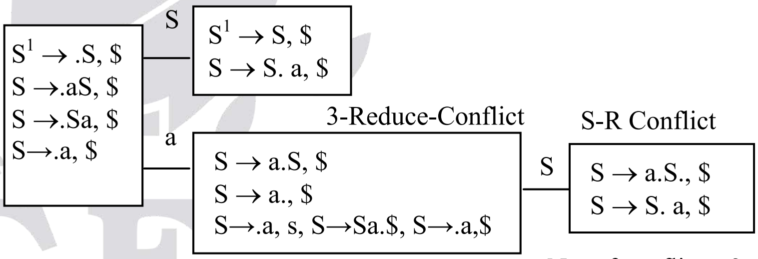
Sol: S→.A,$ S→. A, $

A→. AB, $/ Follow (A) ⇒ A→. AB, $/b

A→. , $/ Follow (A) A→., $/b

63. Ans: (d)

Sol:



No. of conflicts=2

64. Ans: (c)

Sol:

S1 → .S. $
S →. L=R, $
S →. R, $
L→. *R,=/ $
L→.id, = /$
R→.L, $

65. Ans : (c)

Sol: The grammar is only LR(1)

66. Ans: (d)

Sol: The grammar is LL(1)

$S' \rightarrow .S$
$S \rightarrow .(S)$
$S \rightarrow .$

Every LL(1) is LR (1)

67. Ans: (b)

68. Ans: (b)

Sol: SLR(1) & LALR(1) have the same number of states. LR(1) may have more.

69. Ans: 10

Sol: The number of states in both SLR(1) and LALR(1) are same.

70. Ans: (c)

Sol: YACC uses LALR (1) parse table as it uses less number of states requires less space and takes less time for the construction of parse tree.

4. Syntax Directed Translation Schema

01. Ans: (c)

Sol: SDT is part of semantic Analysis

02. Ans: (c)

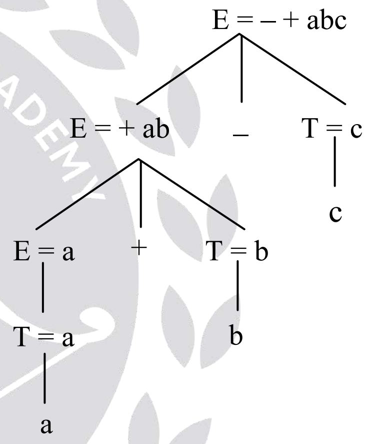
Sol: The attribute ‘val’ is synthesized and the SDD is S-attributed and every ‘S’-attributed is L-attributed definition

03. Ans: (c)

Sol: Given SDT counting the number of a’s and b’s in a given string.

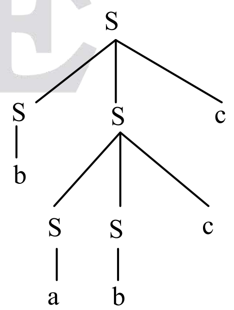
04. Ans: (c)

Sol: For input: a + b – c



05. Ans: (c)

Sol:



Bottom up traversal of the parse tree results the output: 10.

06. Ans: (b)

Sol: $S \rightarrow S_1\ S_2\ c\{\ S.val = S_1.val * S_2.val - 4\}$

$S \rightarrow a\ \{S.val = 6\}$

$S \rightarrow b\ \{S.val = 2\}$

The rightmost derivation of 'abc' is

$S \Rightarrow\ SSc$

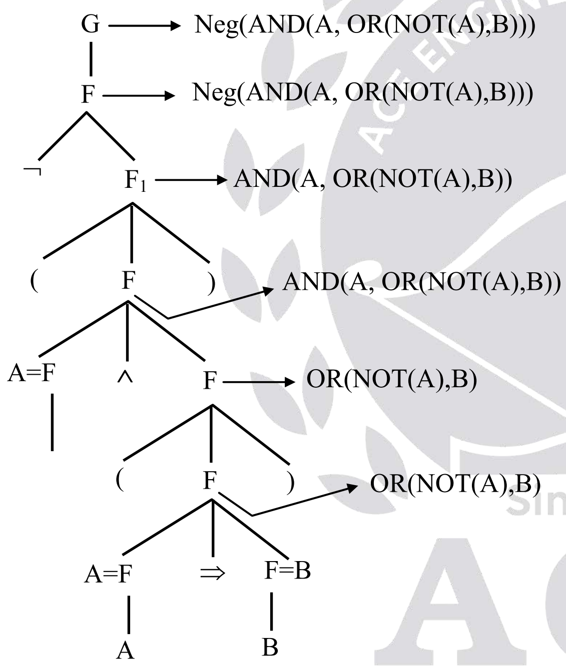
$\Rightarrow S\ bc$

$\Rightarrow a\ b\ c$

In $S_1\ S_2$ c, $S_1.val = 6$, $S_2.val = 2$. So answer is "8".

07. Ans: (c)

Sol: $\neg(A \wedge (A \Rightarrow B))$



08. Ans: (c)

Sol: The rightmost derivation is

$E \rightarrow E + E \rightarrow E + E + E$

$\rightarrow E + E + E + E$

$\rightarrow E + E + E + E + E$

$\equiv a + b + c + d + e$

09. Ans: (a)

Sol: The leftmost derivation for aaaa is

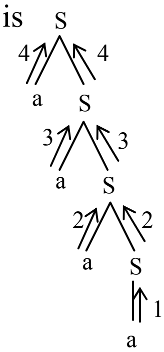
$S \rightarrow aS$

$\rightarrow aaS$

$\rightarrow aaaS$

$\rightarrow aaaa$

The dependency graph



10. Ans: (a)

Sol: The rightmost derivation is

$S \rightarrow aB \rightarrow aa\ BB \rightarrow aa\ Bb \rightarrow aa\ bb$

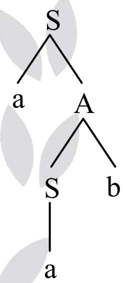
11. Ans: (c)

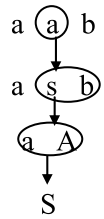
Sol: $S \rightarrow aA$ {print 1}

$S \rightarrow a$ {print 2}

$A \rightarrow Sb$ {print 3}

Input: aab





231

12. Ans: (c)

Sol: $a_1b_1a_2b_2b_3$

$S \Rightarrow a_1S$	$S \rightarrow a_1S$
$\Rightarrow a_1b_1S$	$S \rightarrow b_1S$
$\Rightarrow a_1b_1a_2S$	$S \rightarrow a_2S$
$\Rightarrow a_1b_1a_2b_2S$	$S \rightarrow b_2S$
$\Rightarrow a_1b_1a_2b_2b_3$	$S \rightarrow b_3$

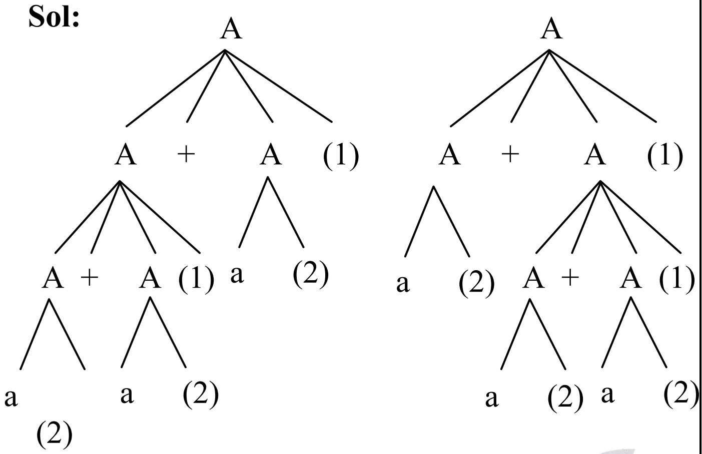
Above is rightmost derivation

$S \xrightarrow{①} b_3$ z $S \xrightarrow{②} b_2S$ zy $S \xrightarrow{③} a_2S$ zyx

$S \xrightarrow{⑤} b_1S$ zyxy $S \xrightarrow{④} a_1S$ zyxyx

13. Ans: (a)

Sol:

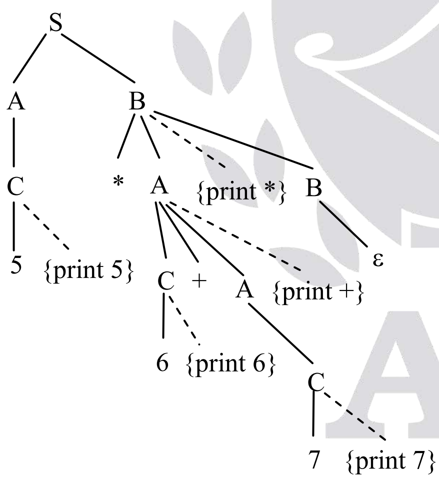


14. Ans: (c)

Sol: As the grammar is ambiguous & we do not specify the precedence of operators either postfix form may result depending on the parser implementation.

15. Ans: (d)

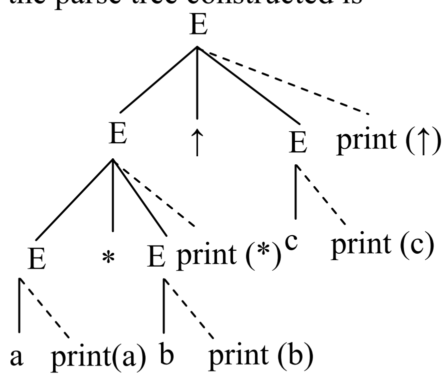
Sol:



The depth first traversal of above tree prints 5 6 7 + *

16. Ans: (a)

Sol: According to the action of shift reduce parser, the parse tree constructed is



The Depth First Traversal of the above parse tree is a b * c ↑

5. Intermediate Code Generation

01. Ans: (c)

Sol: The purpose of using intermediate codes in compilers is to reuse machine independent code for other compilers.

02. Ans: (d)

Sol: The final result is the machine language code. The others are all standard intermediate forms.

03. Ans: (d)

Sol: TAC is a statement that contains atmost three memory references.

04. Ans: (d)

Sol: TAC can be implemented as a record structure with fields for operator, and arguments as Quadruples, triples and indirect triples.

05. Ans: (b)

Sol: The Quadruples is record structure with four fields.

1. (*, b, c, T_1)
2. (+, a, T_1, T_2)
3. (–, T_2, d, T_3)

06. Ans: (c)

Sol: (1) (and, b, c, T_1)

(2) (or, a, T_1, T_2, c, T_3)

(3) (or, T_2, c, T_3)

07. Ans: (a)

Sol: 1. (+, b, c)

2. (NEG, (1))

3. (*, a, (2))

08. Ans: 10

Sol: Rewriting the given assignments

$x_1 = u_1 - t_1$; → needs two new variables

$y_2 = x_1 * v_1$; → needs three new variables

$x_3 = y_2 + w_1$; → needs four new variables

$y_4 = t_2 - z_1$; → needs five new variables

$y_5 = y_2 + w_1 + y_4$; → needs 10 new variables atmost

09. Ans: (b)

Sol: All assignments in SSA are to variables with distinct names

$p_3 = a - b$

$q_4 = P_3 * c$

$p_4 = u * v$

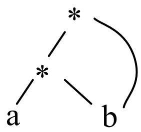
$q_5 = P_4 + q_4$

10. Ans: (d)

Sol: Peephole optimization expression is the final code.

11. Ans: (d)

Sol: DAG for the expression a*b*b is

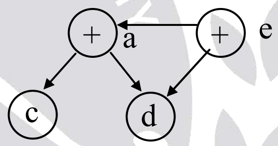


12. Ans: (b)

Sol: DAG is constructed based on precedence and associativity of operators, and option (b) is the correct representation.

13. Ans: 4

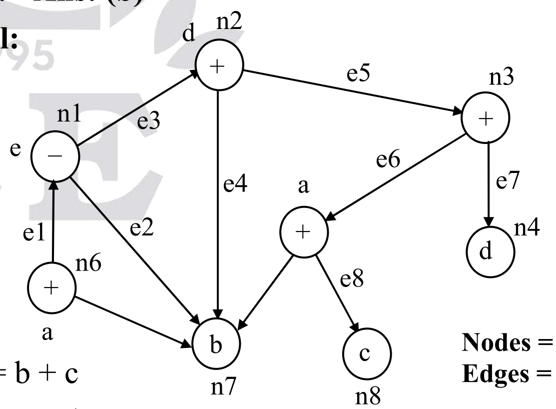
Sol:



Number of nodes = 4

14. Ans: (b)

Sol:



Nodes = 8
Edges = 10

a = b + c

c = a + d

d = b + c

e = d – b

a = e + d

Number of nodes = 8
Number of edges = 10

15. Ans: (a)

Sol: In C the storage for array is row major order. Between X[l] [32] [8] & X [l+1] [32] [8] there must be 32 × 8 integer of type int i.e 32 × 8 × 4 = 1024 bytes. So in X[i] [j] [k] for a variation of index i by 1, 1024 bytes must be skipped. So the answer must be (a)

16. Ans: (b)

Sol: (1) (+, c, d)
(2) (–, b, (1))
(3) (*, e, f)
(4) (+, (2), (3))
(5) (=, a, (4))