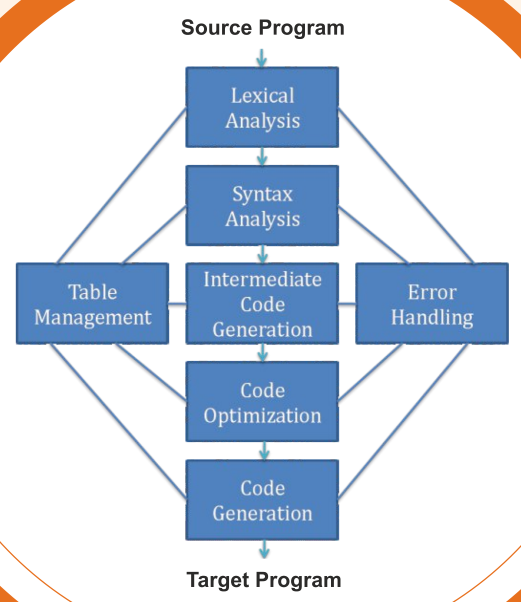
ACE Engineering Publications

GATE | PSUs



Computer Science & Information Technology

COMPILER DESIGN

Text Book:
Theory with worked out Examples and Practice Questions

Compiler Design

(Solutions for Text Book Practice Questions)

Chapter 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: (d)

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

06. Ans: (b)

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

07. Ans: (b) & (c)

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

08. Ans: (b), (c) & (d)

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

09. Ans: (a) & (d)

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

10. Ans: 20

11. Ans: 7

12. Ans: (b)

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

13. Ans: (a), (b) & (c)

Sol: All are tokens only.

14. 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.

15. Ans: (c)

Sol: In $50000; $ is an illegal symbol identified in lexical analysis phase

16. 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.

17. Ans: (a) & (b)

Sol:

abc	abc	aa	bbaac	c	caba	c	cbb
3	3	1	5	2	2	2	2

abc	abc	aa	bbaac	c	ca	bac	cbb
3	3	1	5	2	2	5	2

18. Ans: (b)

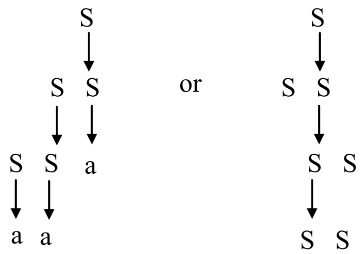
Chapter 3 Parsing Techniques

01. Ans: (b)

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

02. Ans: (d)

Sol:



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

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

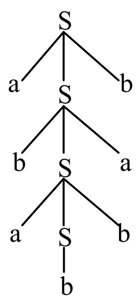
So the sentence has an infinite number of derivations.

03. Ans: (a)

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

04. Ans: (d)

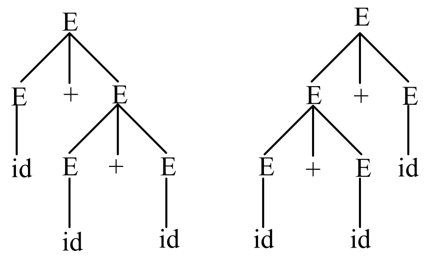
Sol:



Hence option (d) is correct.

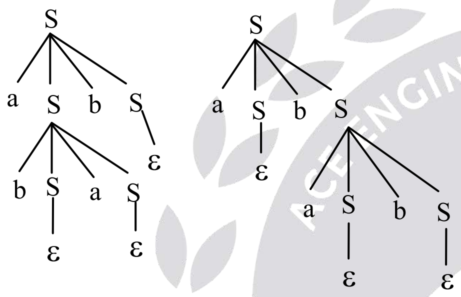
05. Ans: 2

Sol:



06. Ans: (c)

Sol:



07. Ans: (d)

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

08. Ans: (c)

Sol: The production of the form A → A α/β 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: (a) & (c)

11. Ans: (b) & (d)

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: (a), (b) & (c)

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: (c)

Sol: Follow(A) = first (C)

= {f, ∈} – {∈} ∪ follow (S)

= {f} ∪ {$}

= {f, $}

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)$\cup$First(x)$\cup$Follow (D)

= {y, m}$\cup$ {x} $\cup$ Follow(A) $\cup$First(B)

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

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

20. Ans: (a)

Sol: Follow(S) = {$}

Consider S $\rightarrow$ [SX]

Follow(S) = First (X)

= {+,–, b} $\cup$ {]}

= {+, – , b,]}

Consider X $\rightarrow$ + SY

Follow(S) = First (Y)

= {–} $\cup$ Follow(X)

= {–} $\cup$ {c,]}

= {–, c,]}

Consider Y $\rightarrow$ – S X c

Follow(S) = First(X)

= {+, –, b} $\cup$ First(c)

= {+, –, b, c}

$\therefore$ Follow(S) = {+, – , b, c,], $}

21. Ans: (c)

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

First(S) = {a, +, ε}

$\therefore$ Follow(T) $\cap$ First (S) = {+}

22. Ans: (d)

Sol: Follow(A)=first(B)$\cup$Follow(S)$\cup$Follow(B)

={e} $\cup$ {f} $\cup$ {c, d}={c,d,e,$}.

23. Ans: (a)

Sol: First(A)= {*, +, id, $\in$}

Follow(A) = {d, $}

24. Ans: (a)

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

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 no choice.

26. Ans: (c)

Sol: To distinguish between

S $\rightarrow$ if expr then stmt

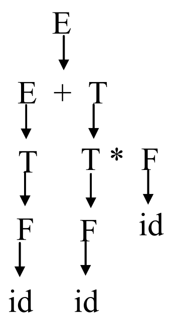
& S $\rightarrow$ if expr then stmt else stmt

We need a look ahead of 5 symbols.

27. Ans: (c)

Sol: * has a higher precedence than +.

Consider



28. Ans: (c)

Sol: M[B,y] contains both B$\rightarrow$yA and B$\rightarrow$ ε

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: 0

Sol: The grammar is LL(1) Since the parse table is free from multiple entries

32. Ans: (c)

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

Follow(A) = {a}

S→∈ is entered into M[S, follow(S)] = S→∈

A→S is entered into M[S, follow(A)] = A→S

33. Ans: (a) & (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: (b)

Sol: The precedence relation between two adjacent terminals is =.

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

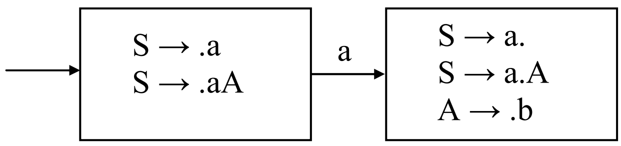
is left recursive. So it is not LL(1) but is LR(0). So (a) is true & (b) is false.

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

$E \to E + E \mid E * E \mid i$

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

$S \to a \mid A, \;\; A \to a$

is neither LL(1) nor LR(0) & is ambiguous. No ambiguous grammar can be LL or LR.

45. Ans: (a), (b) & (c)

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

46. Ans: (c)

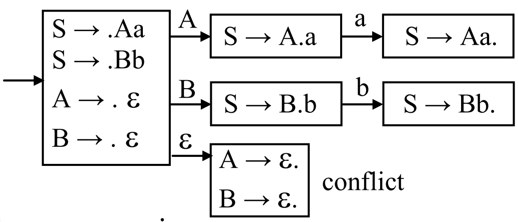
Sol: The grammar

$S \to Aa \mid Bb$

$A \to \varepsilon$

$B \to \varepsilon$ is LL(1) but not LR(0)

The LR(0) machine has a conflict.



The grammar is

$S \to a \mid ab$

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

47. Ans: (a), (b) & (c)

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

Every SLR(1) grammar is LALR(1)

Every LALR(1) grammar is LR(1)

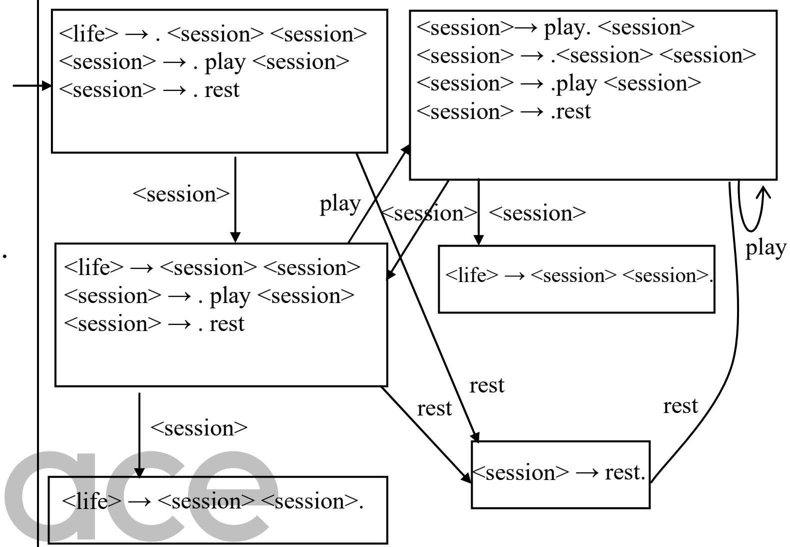
The grammar $S \to a$ is both LL(1) & LR(0) trivially.

48. Ans: (b)

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

49. Ans: (a)

Sol: The LR(0) machine for the grammar



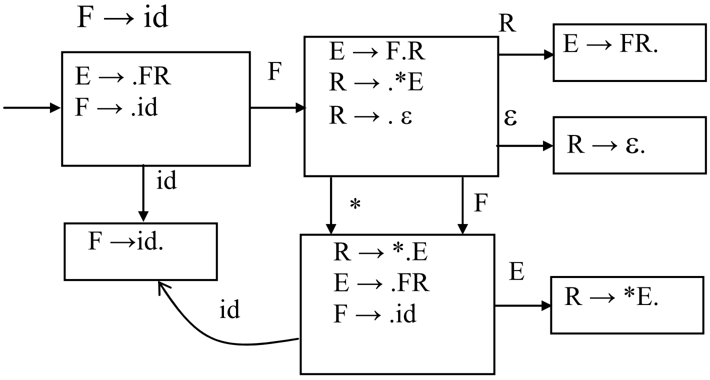
50. Ans: (b)

Sol: The LR(0) machine

$E \to FR$

$R \to *E/\varepsilon$

$F \to id$



51. Ans: (b)

Sol:

$S^1 \to .S$

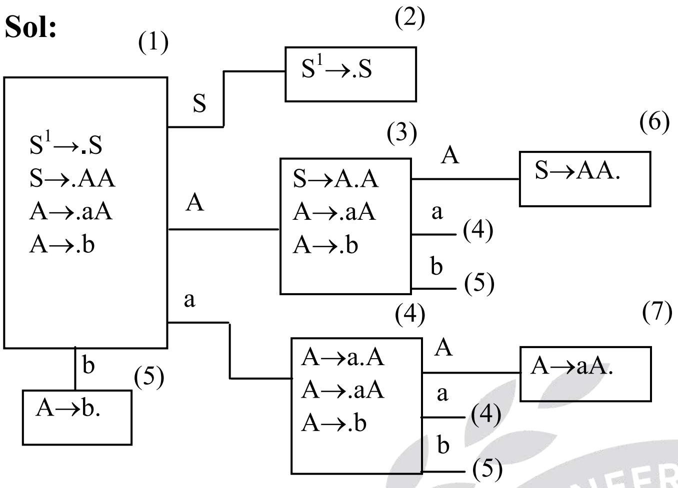
$S \to .SB$

$S \to .A$

$A \to .a$

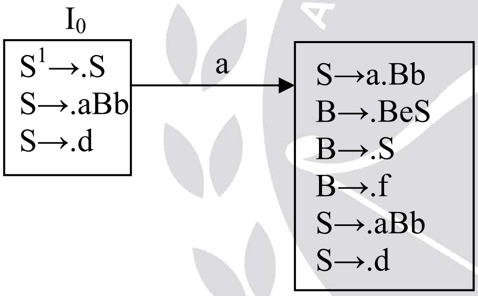
52. Ans: 7

Sol:



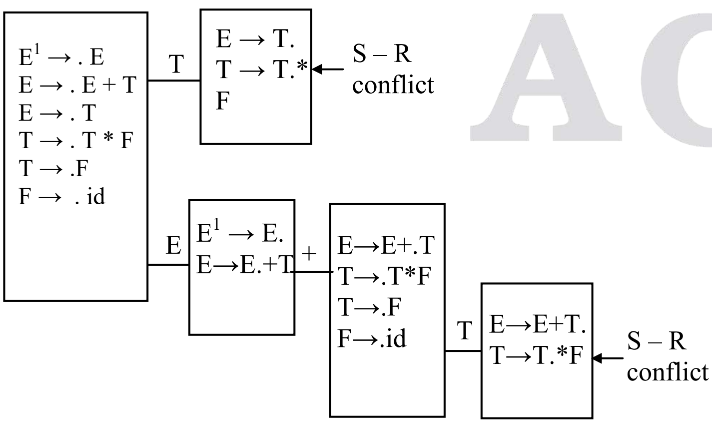
53. Ans: 6

Sol:



54. Ans: 2

Sol:

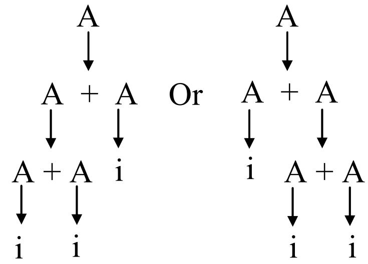


55. Ans: (b)

Sol: An input 'a' the set has A→Ba.c, B→a. C→.

56. Ans: (d)

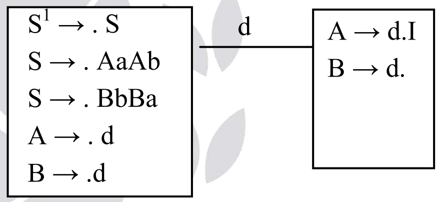
Sol: The grammar is ambiguous.



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

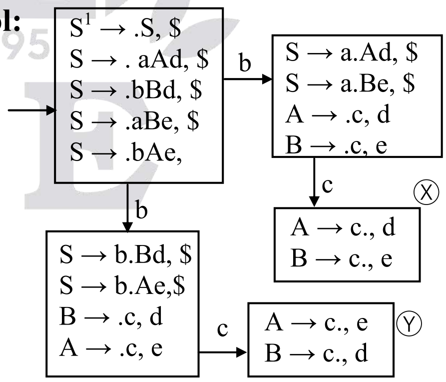
Sol: The LR(0) items of the grammar is



Reduce – Reduce conflict.

58. Ans: (a)

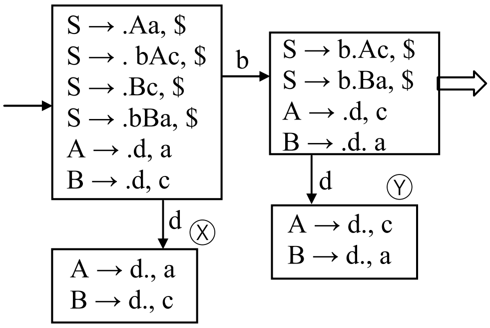
Sol:



Consider the partial LR(1) machine shown above. The states Ⓧ & Ⓨ have a common core. However if we merge the states 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)

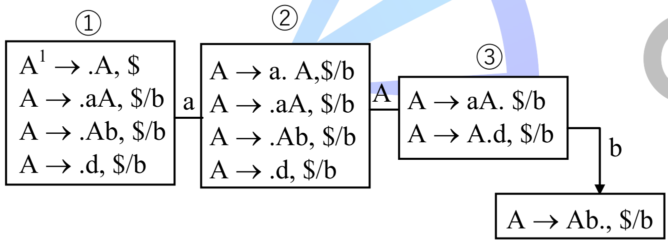
Sol:



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)

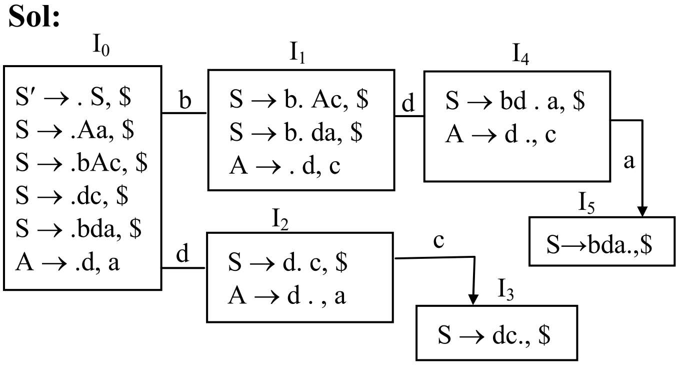
Sol: LR(1) items of the grammar is



Item 3 has Shift-Reduce conflict.

61. Ans:(d)

Sol:



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

62. Ans: (c)

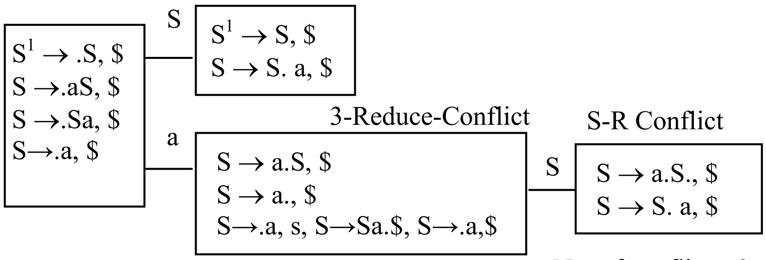
Sol: S⟶.A,$ S⟶. A, $

A⟶. AB, $/ Follow (A) ⇒ A⟶. AB, $/b

A⟶. , $/ Follow (A) A⟶., $/b

63. Ans: (d)

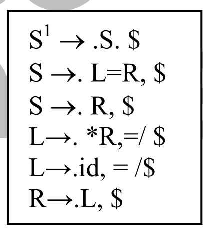
Sol:



No. of conflicts=2

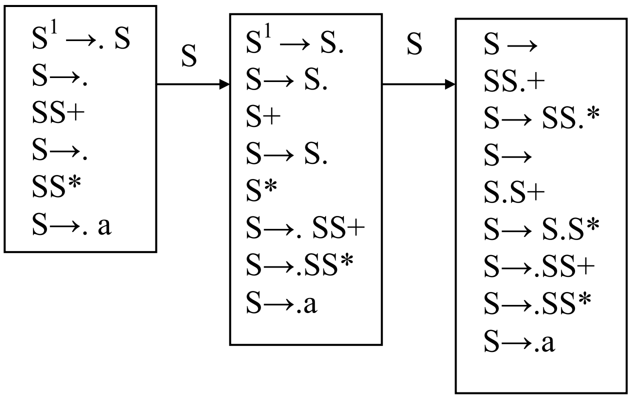
64. Ans: (c)

Sol:



65. Ans: (b)

Sol:



The given grammar is LR(0) as there are no conflicts. Every LR(0) grammar is SLR(1), LALR(1) and LR(1). Given grammar is left recursive and it is not LL(1).

66. Ans: (d)

Sol: The grammar is LL(1)

$$\begin{array}{l} S^1 \rightarrow .S \\ S \rightarrow .(S) \\ S \rightarrow . \end{array}$$

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.

Chapter 4 Syntax Directed Translation Schema

01. Ans: (c)

Sol: SDT is part of Semantic Analysis

02. Ans: (a) & (b)

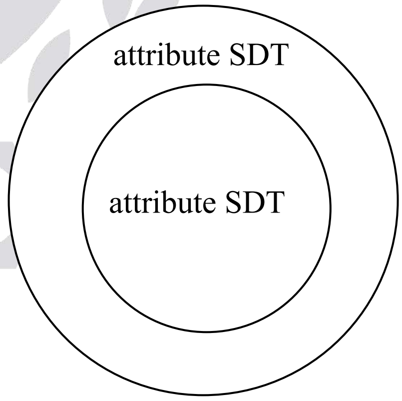
Sol: The attribute 'val' is synthesized and the SDT is S-attributed and every 'S'-attributed is L-attributed definition

03. Ans: (a) & (c)

Sol: $P \rightarrow YQ\{Q.q = g(P.p, Y.y)\}$

Q is taking values from parents and Left siblings. → L-attributed

Since Left siblings are involved not S-attributed.

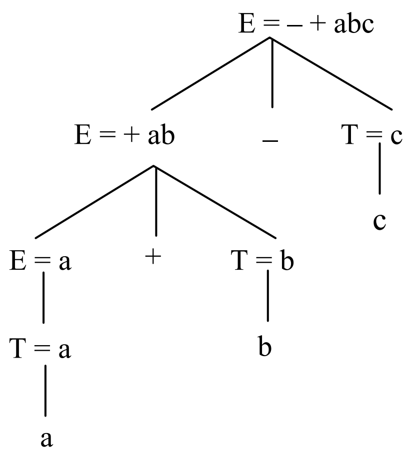


04. Ans: (c)

Sol: The SDD is used to convert the given binary number to decimal number and the answer is 5.625

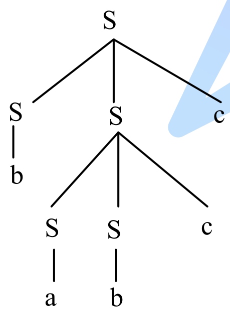
05. Ans: (c)

Sol: For input: a + b – c



06. Ans: (c)

Sol:



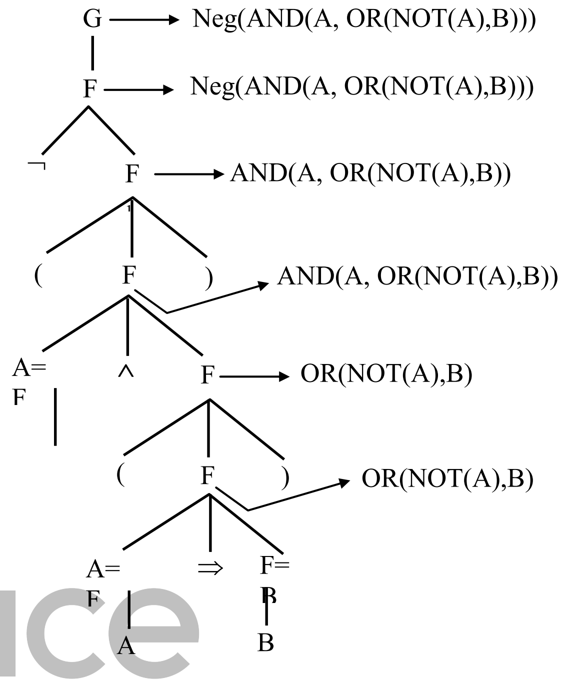
Bottom up traversal of the parse tree results the output: 10.

07. Ans: (b)

Sol: counts the pairs of matching parenthesis.

08. Ans: (c)

Sol: ¬(A ∧(A⇒B))



09. Ans: (c)

Sol: The rightmost derivation is

E → E +E → E + E + E

→ E + E + E + E

→ E + E + E + E + E

≡ a + b + c + d + e

10. Ans: (c)

Sol: The leftmost derivation for aaaa is

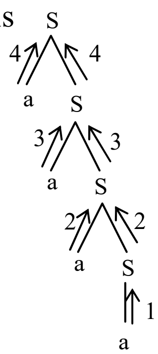
S → aS

→ aaS

→ aaaS

→ aaaa

The dependency graph



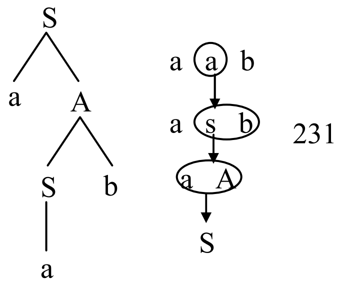
11. Ans: (a)

Sol: The rightmost derivation is

S → aB → aa BB →aa Bb → aa bb

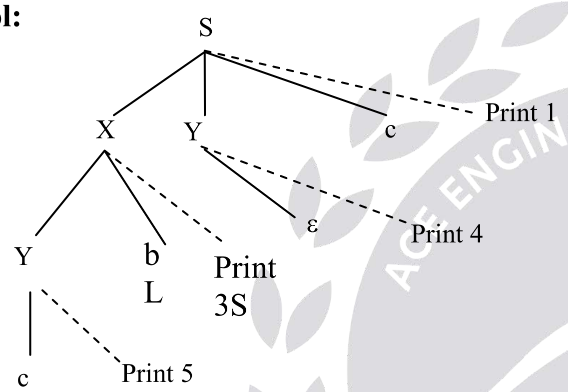
12. Ans: (c)

Sol: S → aA {print 1}
S → a {print 2}
A → Sb {print 3}
Input: aab



13. Ans: (b)

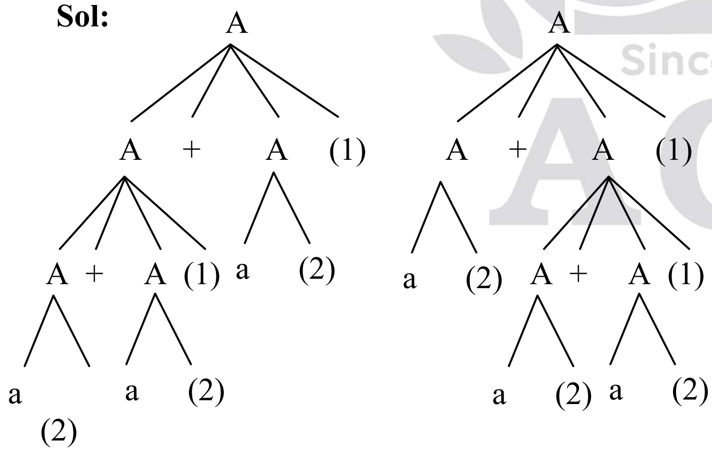
Sol:



The depth first traversal of a parse tree generates an output 5, 3, 4, 1.

14. Ans: (a) & (c)

Sol:

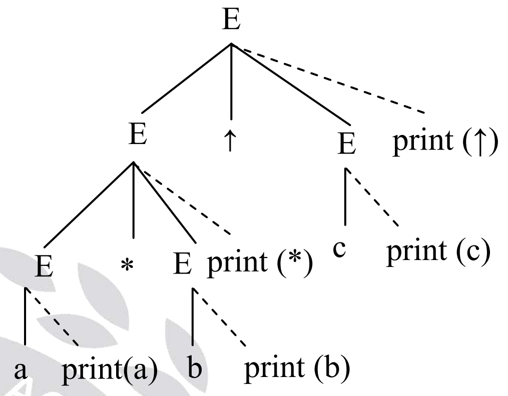


15. Ans: (b)

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

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 ↑

Chapter 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: (a), (b) & (c)

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

03. Ans: (a), (b) & (c)

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

04. Ans: (a), (b) & (c)

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: 3

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 3 new variables atmost

09. Ans: (b)

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

$p_3 = a - b$

$q_4 = P_3 * c$

$p_4 = u * v$

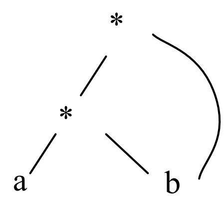
$q_5 = P_4 + q_4$

10. Ans: (d)

Sol: Peephole optimization expression is the final code.

11. Ans: (d)

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

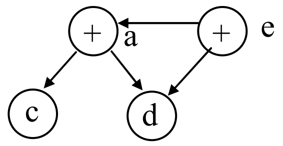


12. Ans: (b)

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

13. Ans: 4

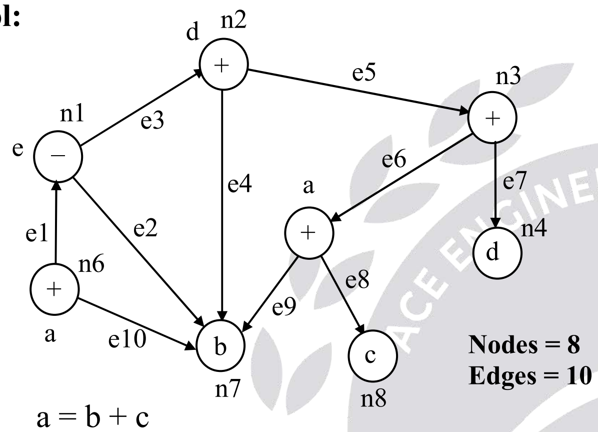
Sol:



Number of nodes = 4

14. Ans: (b)

Sol:



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))

Chapter 6 Code optimization

01. Ans: (a)

Sol: It is called reduction in strength example: replace * by +

02. Ans: (c)

Sol: It is classical example of reduction in strength

03. Ans: (c)

Sol: Machine dependent optimization based on the machine properties and machine dependent optimization is one of it.

04. Ans: (a) & (b)

Sol: Copy propagation generally creates dead code that can then be eliminated. Eliminating dead code improves efficiency of the program by avoiding the execution of unnecessary statements at run time. If one variable is assigned to another, replace uses of the assigned variable with the copied variable.

05. Ans: (c)

Sol: A fragment of code that resides in the loop and computes the same value at each iteration is called loop-invariant code.

06. Ans: (a)

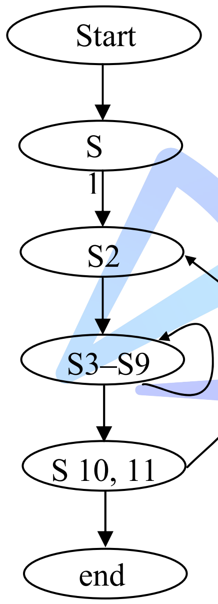
Sol: Eliminating dead code improves efficiency of the program by avoiding the execution of unnecessary statements at run time

07. Ans: (c)

Sol: Before compilation a = b + 2*2.5 after compilation a = b + 5

08. Ans: (b)

Sol: Control flow graph of the above code is



09. Ans: (b)

Sol: b + c is not common sub expression as the value of b changed between 1st and 3rd statements.

10. Ans: (b)

Sol: It has many advantages like optimization and Program analysis is more accurate on intermediate code than on machine code.

11. Ans: (d)

Sol: x = 4 * 5 ⇒ x = 20 is called constant folding.

12. Ans: (d)

Sol: Two *for loops* can be optimized here as code contains loop-invariant computation.

4*j can be evaluated once so there is scope of common sub expression elimination in this code.

The operator * can be replaced by + so there is scope of strength reduction in this code.

No dead code in this program segment.