Compiler Design

M[S, id]

m	id
S	

Q-15: Consider the following grammar:

S → FR
R → S | ε
F → id

In the predictive parser table, M, of the grammar the entries M[S, id] and M[R, $] respectively.

(A) {S → FR} and {R → ε }
(B) {S → FR} and { }
(C) {S → FR} and {R → *S}
(D) {F → id} and {R → ε}

GATE 2005 → 98%

~~Q-15~~:The grammar A → AA | (A) | ε is not suitable for predictive-parsing because the grammar is

(A) ambiguous ✓
(B) left-recursive
✗ **(C)** right-recursive
✗ **(D)** an operator-grammar

Compiler Design

Q-16: Consider the following translation scheme.

S → ER

R → *E{print("*");}R | ε

E → F + E {print("+");} | F

F → (S) | id {print(id.value);}

Here id is a token that represents an integer and id.value represents the corresponding integer value. For an input '2 * 3 + 4', this translation scheme prints

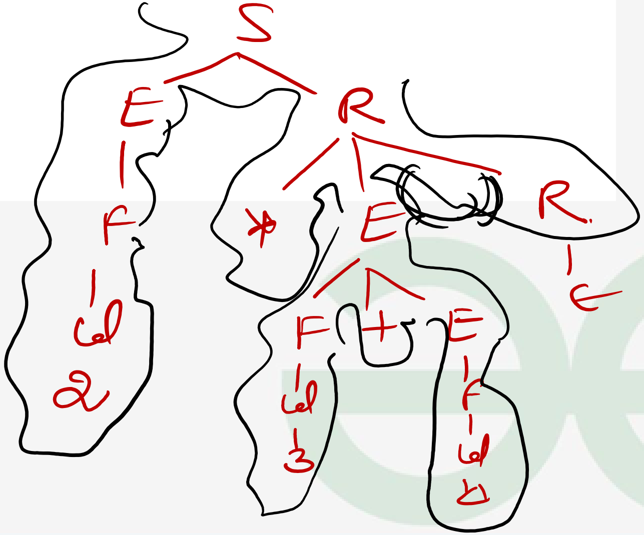
(A) 2 * 3 + 4

(B) 2 * +3 4

(C) 2 3 * 4 +

(D) 2 3 4+*

2 3 4 + *



Compiler Design

Q-17: Consider the grammar

S → (S) | a

n_1 n_2 n_3

Let the number of states in SLR(1), LR(1) and LALR(1) parsers for the grammar be n1, n2 and n3 respectively. The following relationship holds good

(A) n1 < n2 < n3

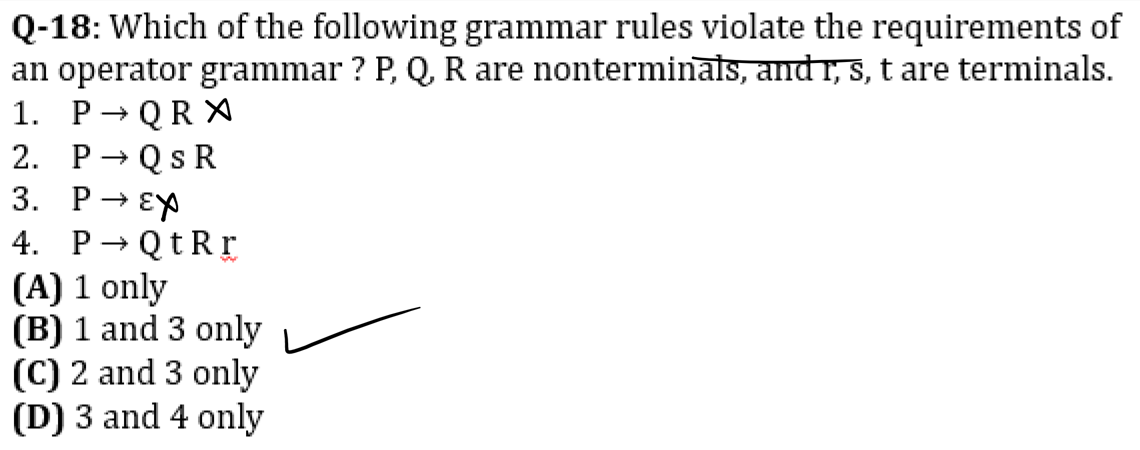
(B) n1 = n3 < n2

(C) n1 = n2 = n3

(D) n1 ≥ n3 ≥ n2

$$\boxed{n_1 = n_3 \le n_2}$$

Compiler Design



Q-18: Which of the following grammar rules violate the requirements of an operator grammar ? P, Q, R are nonterminals, and r, s, t are terminals.

1. P → Q R ✗
2. P → Q s R
3. P → ε ✗
4. P → Q t R r

(A) 1 only

[✓ **(B)** 1 and 3 only]

(C) 2 and 3 only

(D) 3 and 4 only

Compiler Design

Q-19: Consider the grammar with the following translation rules and E as the start symbol.

E → E1 # T { E.value = E1.value * T.value }
      | T{E.value = T.value }
T → T1 & F { T.value = T1.value + F.value }
      | F{T.value = F.value }
F → num{ F.value = num.value }

Compute E.value for the root of the parse tree for the expression: 2 ~~#~~ * (3 ~~&~~ + 5) ~~#~~ * (6 ~~&~~ + 4.)

(A) 200
(B) 180
(C) 160 ✓
(D) 40

+ → #

2 * 8 * 10 = 160

Compiler Design

Q-20: Which of the following suffices to convert an arbitrary CFG to an LL(1) grammar?

(A) Removing left recursion alone ✗

(B) Factoring the grammar alone ✗

(C) Removing left recursion and factoring the grammar ✗

[✓ **(D)** None of these]

Q-21: Assume that the SLR parser for a grammar G has n1 states and the LALR parser for G has n2 states. The relationship between n1 and n2 is:

(A) n1 is necessarily less than n2

[✓ **(B)** n1 is necessarily equal to n2]

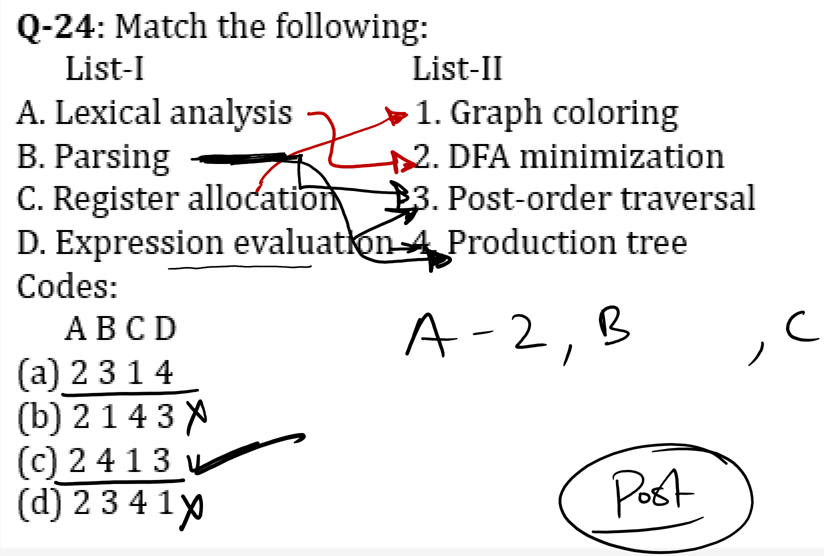
(C) n1 is necessarily greater than n2

(D) none of these

SLR   LALR

$n_1 = n_2$

Compiler Design



Q-24: Match the following:

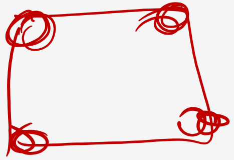
List-I	List-II
A. Lexical analysis	1. Graph coloring
B. Parsing	2. DFA minimization
C. Register allocation	3. Post-order traversal
D. Expression evaluation	4. Production tree

Codes:

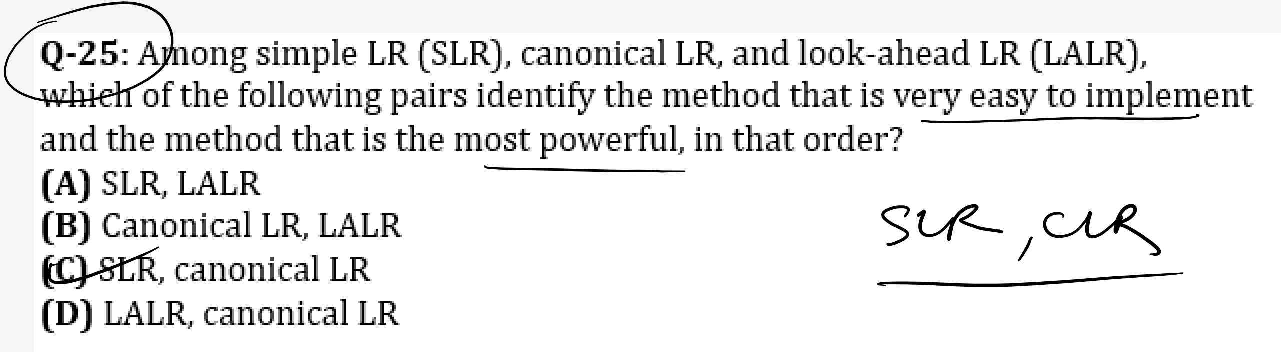
	A	B	C	D
(a)	2	3	1	4
(b)	2	1	4	3 ✗
(c)	2	4	1	3 ✓
(d)	2	3	4	1 ✗

A - 2, B , C - 1, D —

Post



Compiler Design



Q-25: Among simple LR (SLR), canonical LR, and look-ahead LR (LALR), which of the following pairs identify the method that is very easy to implement and the method that is the most powerful, in that order?

(A) SLR, LALR

(B) Canonical LR, LALR

[✓ **(C)** SLR, canonical LR]

(D) LALR, canonical LR

SLR , CLR

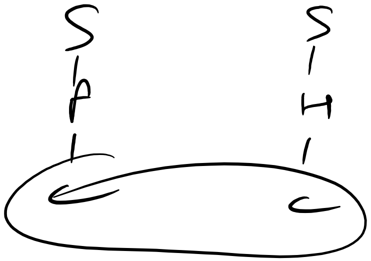
Compiler Design

Q-26: Consider the following grammar G.

S → F | H

F → p | c

H → d | c



Where S, F and H are non-terminal symbols, p, d and c are terminal symbols.
Which of the following statement(s) is/are correct?

S1: LL(1) can parse all strings that are generated using grammar G.

S2: LR(1) can parse all strings that are generated using grammar G.

(A) Only S1

(B) Only S2

(C) Both S1 and S2

[✓ **(D)** Neither S1 and S2]

Compiler Design

Q-27: Consider the grammar shown below.

S → C C

C → c C | d

The grammar is

[✓ **(A)** LL(1)]

(B) SLR(1) but not LL(1)

(C) LALR(1) but not SLR(1)

(D) LR(1) but not LALR(1)

S → c c ×

C → c C / d

⇒ LL(1) ⇒ SLR(1) ⇒ LALR(1)

⇒ LR(1)

∅

Q-28: Shift reduce parsing belongs to a class of

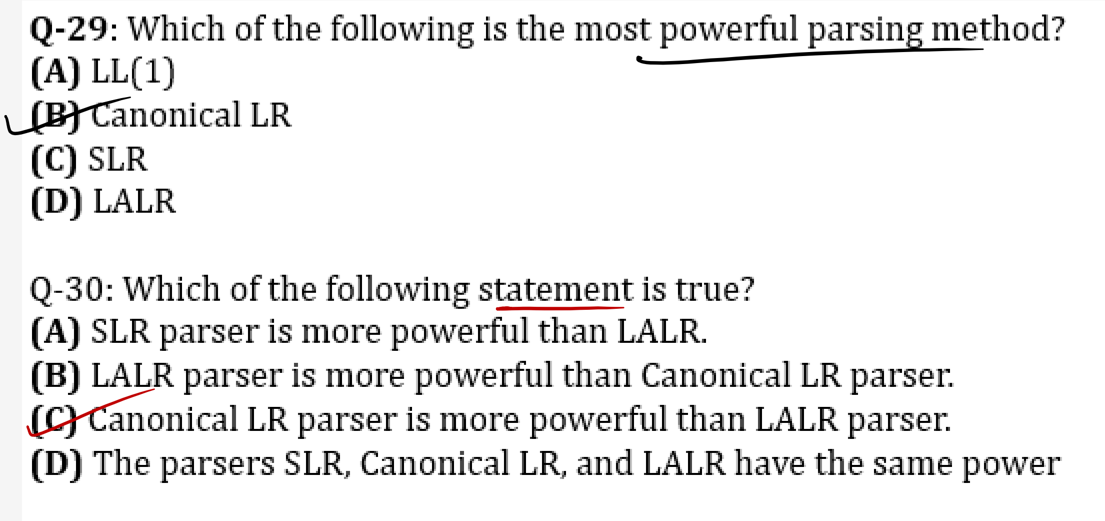
[✓ **(A)** bottom up parsing]

(B) top down parsing

(C) recursive parsing

(D) predictive parsing

Compiler Design



Q-29: Which of the following is the most powerful parsing method?
(A) LL(1)
[✓ **(B)** Canonical LR]
(C) SLR
(D) LALR

Q-30: Which of the following statement is true?
(A) SLR parser is more powerful than LALR.
(B) LALR parser is more powerful than Canonical LR parser.
[✓ **(C)** Canonical LR parser is more powerful than LALR parser.]
(D) The parsers SLR, Canonical LR, and LALR have the same power

Compiler Design

Q-31: Type checking is normally done during

(A) Lexical analysis

(B) Syntax analysis

[✓ **(C)**] Syntax directed translation (SDT) — semantic Analysis

(D) Code optimization

Q-32: Which of the following is essential for converting an infix expression to the postfix from efficiently ?

(A) An operator stack

(B) An operand stack

(C) An operand stack and an operator stack

(D) A parse tree

Compiler Design

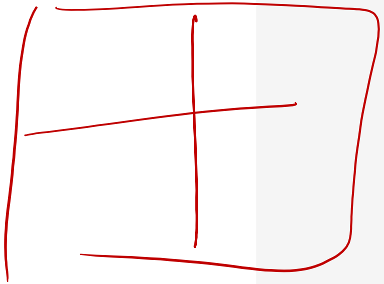
Q-33: Which is True about SR and RR-conflict:

(A) If there is no SR-conflict in CLR(1) then definitely there will be no SR-conflict in LALR(1). T

(B) RR-conflict might occur if lookahead for final items(reduce-moves) is same. T

(C) Known that CLR(1) has no RR-conflict, still RR-conflict might occur in LALR(1). T

[✓ **(D)**] All of the above.



Q-34: Which one of the following statements is FALSE?

(A) Context-free grammar can be used to specify both lexical and syntax rules.

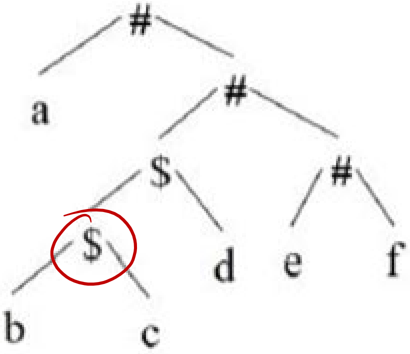
[✓ **(B)**] Type checking is done before parsing.

(C) High-level language programs can be translated to different Intermediate Representations.

(D) Arguments to a function can be passed using the program stack.

Compiler Design

Q-35: Consider the following parse tree for the expression a#bcd#e#f, involving two binary operators $ and #.



→ a# / $#

$ → #d / bc

$ > #

Which one of the following is correct for the given parse tree?

[✓ (A)] $ has higher precedence and is left associative; # is right associative

(B) # has higher precedence and is left associative; $ is right associative

(C) $ has higher precedence and is left associative; # is left associative

(D) # has higher precedence and is right associative; $ is left associative

Compiler Design

Q-36: Consider the following statements related to compiler construction :

I. True Lexical Analysis is specified by context-free grammars and implemented by pushdown automata.

II. False Syntax Analysis is specified by regular expressions and implemented by finite-state machine. ✗

Which of the above statement(s) is/are correct ?

(A) Only I ✓

(B) Only II

(C) Both I and II

(D) Neither I nor II

→ [+ = *]

Q-40: Given the following expression grammar:

E → E ∗ F | F + E | F

F → F – F | id

Which of the following is true?

(A) * has higher precedence than + ✗

(B) – has higher precedence than * ✓

(C) + and – have same precedence ✗

(D) + has higher precedence than * ✗

Compiler Design

Q-37: Yacc stands for

(A) yet accept compiler constructs

(B) yet accept compiler compiler

(C) yet another compiler construct

[✓ **(D)**] yet another compiler compiler

Q-38: Which one from the following is false? ✓

(A) LALR parser is Bottom – Up parser T

[✓ **(B)**] A parsing algorithm which performs a left to right scanning and a right most deviation is RL (1) LR(1)

(C) LR parser is Bottom – Up parser. T

(D) In LL(1), the 1 indicates that there is a one – symbol look – ahead

Compiler Design

Q-39: Which phase of compiler generates stream of atoms?

Atomic

(A) Syntax Analysis
[✓ **(B)** Lexical Analysis]
(C) Code Generation
(D) Code Optimization

Q-40: Which of the following is true?

(A) Canonical LR(1) parser is LR (1) parser with single look ahead terminal
(B) All LR(K) parsers with K > 1 can be transformed into LR(1) parsers.
[✓ **(C)** Both (A) and (B)]
(D) None of the above

Compiler Design

Q-41: In a two-pass assembler, symbol table is

[✓ **(A)** Generated in first pass]

(B) Generated in second pass

(C) Not generated at all

(D) Generated and used only in second pass

Q-42: How many tokens will be generated by the scanner for the following statement ?

x = x ∗ (a + b) – 5;

(A) 12

(B) 11

(C) 10

(D) 07

x = x * (a+b) -5 ;

Compiler Design

Q-43: Consider the following code segment.

```
x = u - t;
y = x * v;
x = y + w;
y = t - z;
y = x * y;
```

The minimum number of total variables required to convert the above code segment to static single assignment form is

(A) 6
(B) 8
(C) 9
[✓ **(D)** 10]

Compiler Design

Q-45: Consider the following intermediate program in three address code

$p_3 = a - b$

$q_4 = p_3 * c$

$p_4 = u * v$

$q_5 = p_4 + q_4$

Which one of the following corresponds to a static single assignment from the above code

A)

$p_1 = a – b$

$q_1 = p_1 * c$

$p_1 = u * v$

$q_1 = p_1 + q_1$

B)

$p_3 = a - b$

$q_4 = p_3 * c$

$p_4 = u * v$

$q_5 = p_4 + q_4$

Ans

C)

$p_1 = a - b$

$q_1 = p_2 * c$

$p_3 = u * v$

$q_2 = p_4 + q_3$

D)

$p_1 = a - b$

$q_1 = p * c$

$p_2 = u * v$

$q_2 = p + q$

Thank You !