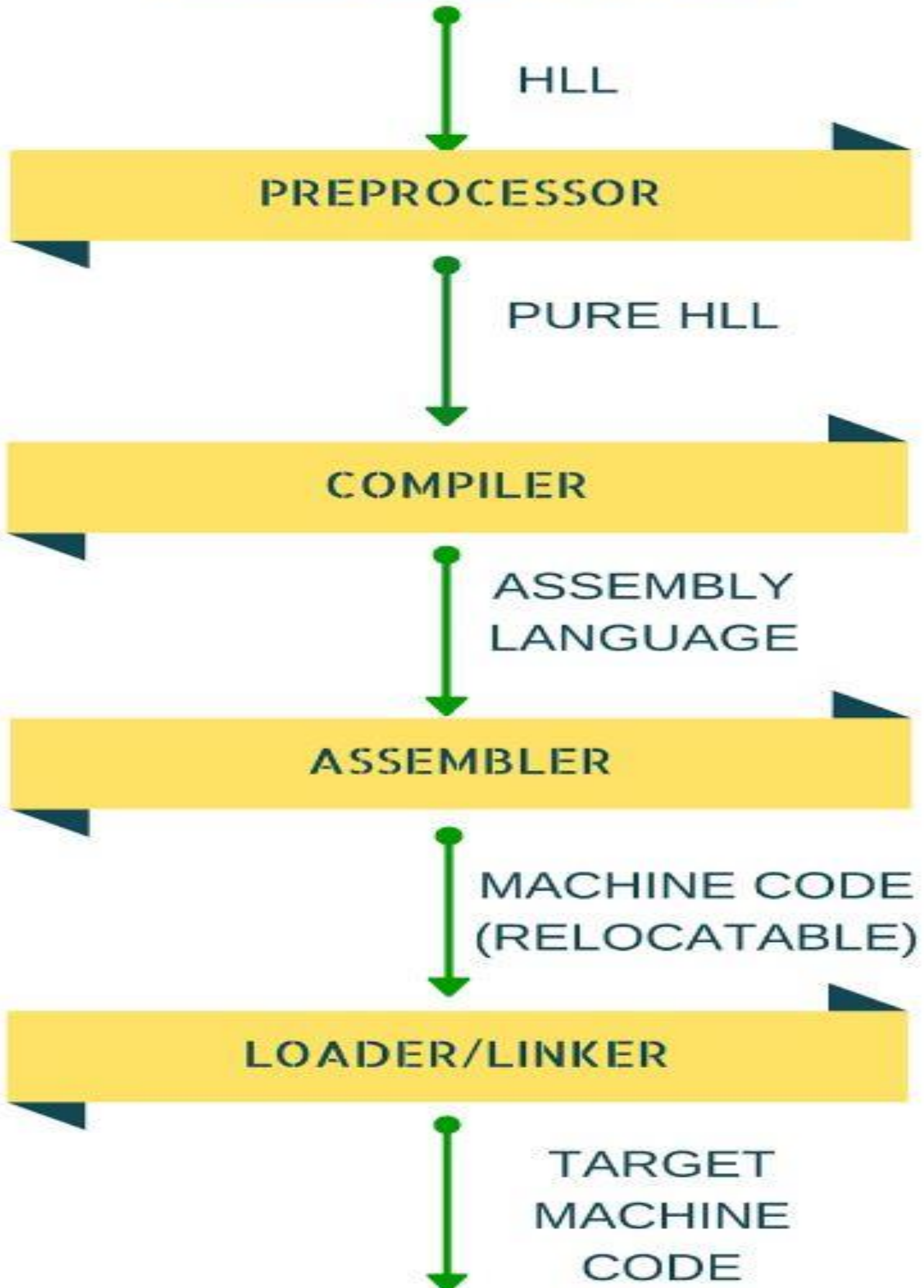
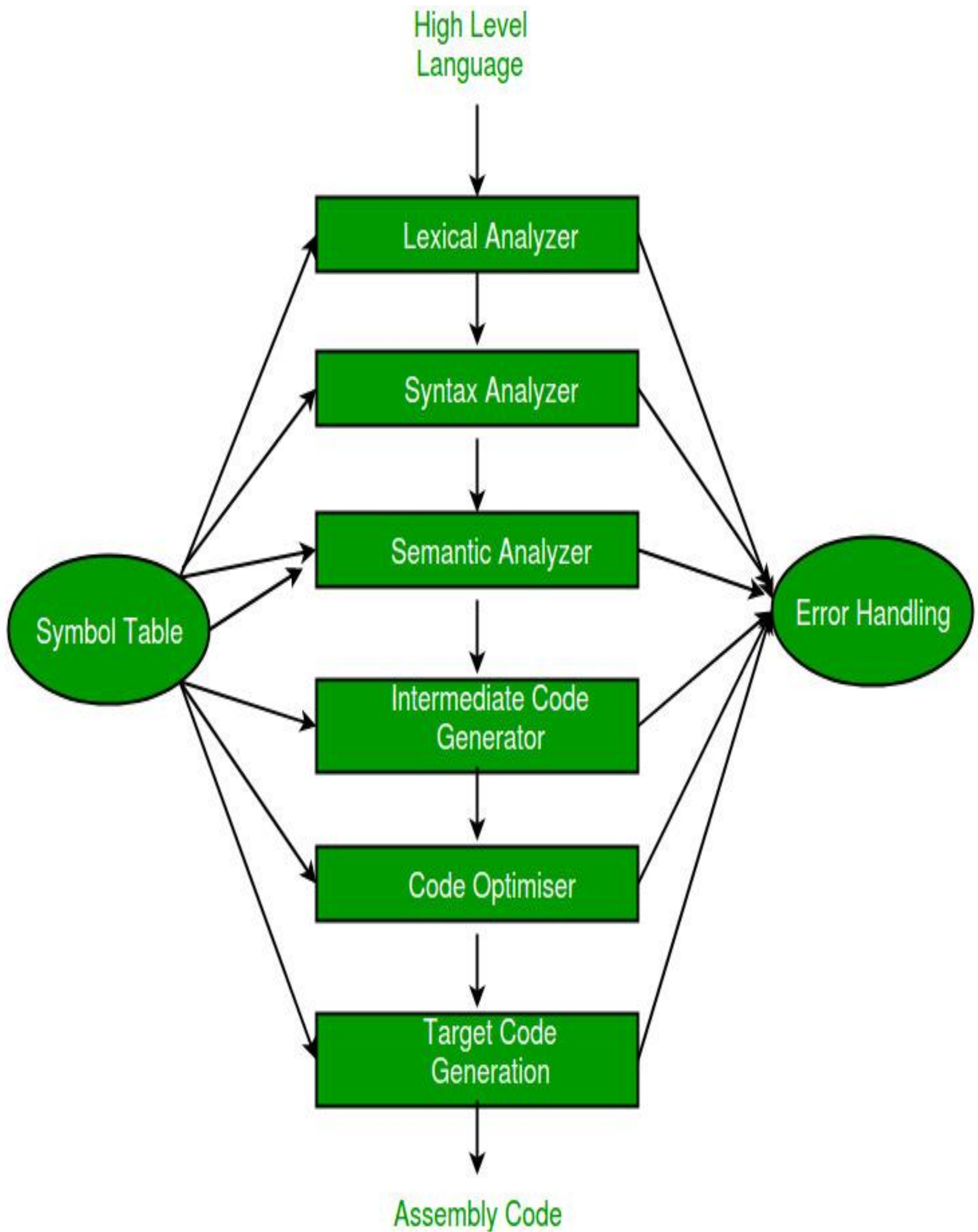
Language Processing System



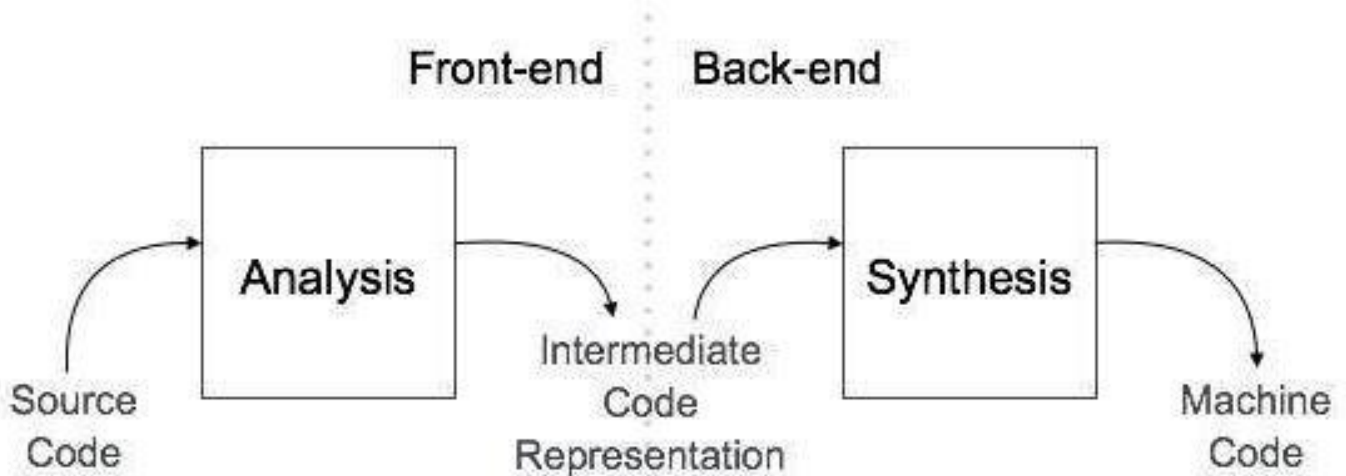
- We know a computer is a logical assembly of Software and Hardware. The hardware knows a language, that is hard for us to grasp, consequently we tend to write programs in high-level language, that is much less complicated for us to comprehend and maintain in thoughts. Now these programs go through a series of transformation so that they can readily be used machines. This is where language procedure systems come handy.
- **High Level Language** – If a program contains #define or #include directives such as #include or #define it is called HLL. They are closer to humans but far from machines. These (#) tags are called pre-processor directives. They direct the pre-processor about what to do.
- **Pre-Processor** – The pre-processor removes all the #include directives by including the files called file inclusion and all the #define directives using macro expansion. It performs file inclusion, macro-processing, short hand operators etc.
- **Pure High-Level Language** – That HLL which can be directly understood by the compiler.
- **Compiler** – A compiler is a computer program that translates computer code written in one programming language (the source language) into another language (the target language). The name compiler is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language) to create an executable program.
    - The reason is we are not comfortable in writing a low-level language there for we write a code which is easy and then convert it into low level language.
- **Assembly Language** – Its neither in binary form nor high level. It is an intermediate state that is a combination of machine instructions and some other useful data needed for execution.
- **Assembler** – For every platform (Hardware + OS) we will have an assembler. They are not universal since for each platform we have one. The output of assembler is called object file. It translates assembly language to machine code.
- **Relocatable Machine Code** – It can be loaded at any point and can be run. The address within the program will be in such a way that it will cooperate for the program movement.
- **Loader/Linker** – It converts the relocatable code into absolute code and tries to run the program resulting in a running program or an error message (or sometimes both can happen). Linker loads a variety of object files into a single file to make it executable. Then loader loads it in memory and executes it.

BASIS FOR COMPARISON	COMPILER	INTERPRETER
Input	It takes an entire program at a time.	It takes a single line of code or instruction at a time.
Output	It generates intermediate object code.	It does not produce any intermediate object code.
Working mechanism	The compilation is done before execution.	Compilation and execution take place simultaneously.
Speed	Comparatively faster	Slower
Memory	Memory requirement is more due to the creation of object code.	It requires less memory as it does not create intermediate object code.
Errors	Display all errors after compilation, all at the same time.	Displays error of each line one by one.
Error detection	Difficult	Easier comparatively
Pertaining Programming languages	C, C++, C#, Scala, typescript uses compiler.	PHP, Perl, Python, Ruby uses an interpreter.

Compiler Design



- We basically have two phases of compilers, namely Analysis phase and Synthesis phase.
    - Analysis phase creates an intermediate representation from the given source code.
    - Synthesis phase creates an equivalent target program from the intermediate representation.



- The compiler has two modules namely front end and back end. Front-end constitutes of the Lexical analyser, semantic analyser, syntax analyser and intermediate code generator. And the rest are assembled to form the back end.
- **Lexical Analyzer** – It reads the program and converts it into tokens. It converts a stream of lexemes into a stream of tokens. Tokens are defined by regular expressions which are understood by the lexical analyser. It also removes white-spaces and comments.
- **Syntax Analyzer** – It is sometimes called as parser. It constructs the parse tree. It takes all the tokens one by one and uses Context Free Grammar to construct the parse tree.
- **Semantic Analyzer** – It verifies the parse tree, whether it's meaningful or not. It furthermore produces a verified parse tree. It also does type checking, Label checking and Flow control checking.
- **Intermediate Code Generator** – It generates intermediate code, that is a form which can be readily executed by machine We have many popular intermediate codes. Example – Three address code etc. Intermediate code is converted to machine language using the last two phases which are platform dependent.
    - Till intermediate code, it is same for every compiler out there, but after that, it depends on the platform. To build a new compiler we don't need to build it from scratch. We can take the intermediate code from the already existing compiler and build the last two parts.
- **Code Optimizer** – It transforms the code so that it consumes fewer resources and produces more speed. The meaning of the code being transformed is not altered. Optimisation can be categorized into two types: machine dependent and machine independent.

- **Target Code Generator** – The main purpose of Target Code generator is to write a code that the machine can understand and also register allocation, instruction selection etc. The output is dependent on the type of assembler. This is the final stage of compilation.

Symbol Table

- It is a data structure being used and maintained by the compiler to store the complete information about the source code. e.g.

(1) int x = 10;

Line No	Keyword	identifier	Constant	Operator
1	int	x	10	;

- Lexical analysis in the first phase to communicate with the symbol and the compiler generate the symbol table during the lexical analysis phase.
- Compiler is responsible to provide the memory for symbol table. at every phase if any new variable occurs, then they will be stored in the symbol table.
- Every phase of the compiler will be interacting with the symbol table.
- In general, during the first two phases, we store the information in the symbol table and in the memory and in the later phases, we make use of the information available in symbol table.
- Information stored in the symbol table about identifier
    - name
    - type
    - scope
    - size
    - offset
- other information in case of array, records and procedures etc.
    - array → size
    - records → column names
    - procedure → i/p parameter
    - functions → i/p, o/p parameter, actual, formal parameter
- Operation on the symbol table: - there are 4 operation function that can be performed on symbol table
    - insert
    - lookup/search
    - Modify
    - delete
- Implementation of symbol tables can be done using anyone of the data structure
    - liner table, tree, list, hash table (most popular)

Error handler

- It is a sub routine to care take care of the continuation of compilation, even any error at any phase, i.e. error handler is responsible to continue the compilation process even any error occurs at phase 1 or phase 2 or phase 3.
- After phase 3, if the error handler object is empty, then the source code is free error and it can be converted into target code. if the error handler object is not empty after phase 3 then there will be some error at phase 1,2 or 3 then the error will be displayed.
- Compiler can handle, there types of errors: -
    - lexical error
    - syntax error
    - semantic error
- The error handles by compiler are known as exception and programmer is responsible to handle the exception.
- At the time of execution also, we can get some error they are called fatal error and system admin is responsible to handle fatal error.

A compiler can have many phases and passes.

- **Pass**: A pass refers to the traversal of a compiler through the entire program. Compiler pass are two types:
  - Single Pass Compiler
    - A one pass/single pass compiler is that type of compiler that passes through the part of each compilation unit exactly once, so going directly from lexical analysis to code generator, and then going back for the next read. e.g. pascal
    - Single pass compiler is faster and smaller than the multi pass compiler.
    - As a disadvantage of single pass compiler is that it is less efficient in comparison with multi pass compiler.
    - As we can't backup and process, it again so grammar should be limited or simplified.
  - Two Pass Compiler *or* Multi Pass Compiler.
    - A Two pass/multi-pass Compiler is a type of compiler that processes the *source code* or abstract syntax tree of a program multiple times. Multi pass compiler is used to process the source code of a program several times.
    - In the first pass, compiler can read the source program, scan it, extract the tokens and store the result in an output file.
    - In the second pass, compiler can read the output file produced by first pass, build the syntactic tree and perform the syntactical analysis. The output of this phase is a file that contains the syntactical tree.
    - In the third pass, compiler can read the output file produced by second pass and check that the tree follows the rules of language or not. The output of semantic analysis phase is the annotated tree syntax.
    - This pass is going on, until the target output is produced.
- **Phase**: A phase of a compiler is a distinguishable stage, which takes input from the previous stage, processes and yields output that can be used as input for the next stage. A pass can have more than one phase.

Q Which of the following are the principles tasks of the linker?

I. Resolve external references among separately compiled program units.

II. Translate assembly language to machine code.

III. Relocate code and data relative to the beginning of the program.

IV. Enforce access-control restrictions on system libraries. **(NET-AUG-2016)**

A) I and II **b)** I and III **c)** II and III **d)** I and IV

Ans: b

Q Which of the following statement(s) regarding a linker software is/are true?

I A function of a linker is to combine several object modules into a single load module.

II A function of a linker is to replace absolute references in an object module by symbolic references to locations in other modules. **(NET-AUG-2016)**

A) Only I **b)** Only II **c)** Both I and II **d)** Neither I nor II

Ans: a

Lexical Analyzer

- The implementation of the Lexical Analyser is known as Lexer or Lexical Generator.
- Lexer read the source code, divide them into Lexemes, if these lexemes are accepted by finite automata, then they are called as tokens.
- Actual representation is called of stream of characters is called as Lexemes; Logical meaning of Lexemes is known as Tokens.

Float x, y, z;

X = y+ z*60

X → identifier → token

= → operator → token

Y → identifier → token

+ → operator → token

Z → identifier → token

* → operator → token

60 → constant → token

$id_1 = id_2 + id_3 * 60$

Q In a compiler, keywords of a language are recognized during **(GATE - 2011) (1 Marks)**

(A) parsing of the program

(B) the code generation

(C) the lexical analysis of the program

(D) dataflow analysis

Answer: (C)

- Secondary Function of Lexical Analyser are: -
    - Removal of Comments lines
    - Removal of White space characters
    - Co-relating with Error Messages along with line number.
- For a Lexemes, the input is source code and output are the stream of tokens.

- Design of Lexical Analyser
    - Hand Coding
    - Lex Tool
- Hand Coding
    - Construct the Regular Expression
    - Convert the Regular Expression into Finite Automata

Q Identify the meaning of the rule implemented by Lexer using regular expression for identifying tokens of a password?
Ans: Regular Expression $A(A+S+D)^7 (A+S+D+\in)^7$

- Lex Tools
    - It is an automated tool to implement Lexical Analysis
    - We need to design the rule, to recognise the pattern and rest of the work will be done by Lex tool. i.e. Construction of Regular Expression into Finite automata will be taken care.

Q The lexical analysis for a modern computer language such as Java needs the power of which one of the following machine models in a necessary and sufficient sense? **(GATE - 2011) (1 Marks)**

(A) Finite state automata
(B) Deterministic pushdown automata
(C) Non-Deterministic pushdown automata
(D) Turing Machine

Answer: (A)

Q Match the following according to input (from the left column) to the compiler phase (in the right column) that processes it: **(GATE - 2017) (2 Marks)**

(P) Syntax tree	(i) Code generator
(Q) Character stream	(ii) Syntax analyzer
(R) Intermediate representation	(iii) Semantic analyzer
(S) Token stream	(iv) Lexical analyzer

(A) P→(ii), Q→(iii), R→(iv), S→(i)
(B) P→(ii), Q→(i), R→(iii), S→(iv)
(C) P→(iii), Q→(iv), R→(i), S→(ii)
(D) P→(i), Q→(iv), R→(ii), S→(iii)

ANSWER C

Q Match the following:

(P) Lexical analysis	**(i)** Leftmost derivation
(Q) Top down parsing	**(ii)** Type checking
(R) Semantic analysis	**(iii)** Regular expressions
(S) Runtime environments	**(iv)** Activation records

(GATE - 2016) (2 Marks)

(1) P ↔ i, Q ↔ ii, R ↔ iv, S ↔ iii
(2) P ↔ iii, Q ↔ i, R ↔ ii, S ↔ iv
(3) P ↔ ii, Q ↔ iii, R ↔ i, S ↔ iv
(4) P ↔ iv, Q ↔ i, R ↔ ii, S ↔ iii

ANSWER 2

Match all items in Group 1 with correct options from those given in Group 2.

Group 1		Group 2	
P.	Regular expression	1.	Syntax analysis
Q.	Pushdown automata	2.	Code generation
R.	Dataflow analysis	3.	Lexical analysis
S.	Register allocation	4.	Code optimization

(GATE - 2009) (2 Marks)

a) P-4, Q-1, R-2, S-3
b) P-3, Q-1, R-4, S-2
c) P-3, Q-4, R-1, S-2
d) P-2, Q-1, R-4, S-3

Answer B

Match the following:

(P) Lexical analysis — (1) Graph coloring
(Q) Parsing — (2) DFA minimization
(R) Register allocation — (3) Post-order traversal
(S) Expression evaluation — (4) Production tree

(GATE - 2015) (2 Marks)

(1) P-2, Q-3, R-1, S-4
(2) P-2, Q-1, R-4, S-3
(3) P-2, Q-4, R-1, S-3
(4) P-2, Q-3, R-4, S-1

ANSWER 3

Q A lexical analyser uses the following patterns to recognize three tokens T_1, T_2, and T_3 over the alphabet {a, b, c}.

T_1: a?(b | c)*a
T_2: b?(a | c)*b
T_3: c?(b | a)*c

Note that 'x?' means 0 or 1 occurrence of the symbol x. Note also that the analyser outputs the token that matches the longest possible prefix. If the string bbaacabc is processes by the analyzer, which one of the following is the sequence of tokens it outputs? **(GATE - 2018) (2 Marks)**

a) $T_1T_2T_3$
b) $T_1T_1T_3$
c) $T_2T_1T_3$
d) T_3T_3

(ANSWER-D)

Q The number of tokens in the following C statement is **(GATE - 2000) (1 Marks)**

printf ("i = %d, &i = %x", i, &i);

(A) 3 **(B)** 26 **(C)** 10 **(D)** 21

Answer: (C)

Q The number of tokens in the following C statement is

If (x < y)

    Printf("Hello");

Q Match the description of several parts of a classic optimizing compiler in List - I, with the names of those parts in List - II: **(NET-NOV-2017)**

	List - I		List - II
(a)	A part of a compiler that is responsible for recognizing syntax.	(i)	Optimizer
(b)	A part of a compiler that takes as input a stream of characters and produces as output a stream of words along with their associated syntactic categories.	(ii)	Semantic Analysis
(c)	A part of a compiler that understand the meanings of variable names and other symbols and checks that they are used in ways consistent with their definitions.	(iii)	Parser
(d)	An IR-to-IR transformer that tries to improve the IR program in some way (Intermediate Representation).	(iv)	Scanner

Code :

	(a)	(b)	(c)	(d)
(1)	(iii)	(iv)	(ii)	(i)
(2)	(iv)	(iii)	(ii)	(i)
(3)	(ii)	(iv)	(i)	(iii)
(4)	(ii)	(iv)	(iii)	(i)

Q Which phase of compiler generates stream of atoms? **(NET-June-2015)**

a) Syntax Analysis **b)** Lexical Analysis

c) Code Generation **d)** Code Optimization

Ans: a

Q How many tokens will be generated by the scanner for the following statement? **(NET-DEC-2014)**

x = x ∗ (a + b) – 5;

(A) 12 **(B)** 11 **(C)** 10 **(D)** 07

Ans: a

Q Match the following **(Gate-2015) (2 Marks)**

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

Answer: (C)

Syntax analysis

- The process of construction of the parse tree/ syntax tree/ derivation tree is called as parsing.
- For any input string which is given in the form of stream of tokens for the parser, if the derivation tree exists, then the input string is syntactically or grammatically correct.
- If the parser cannot generate the derivation tree from the i/p string, then there must be some grammatical mistakes in the string.

Introduction

Language usually contains infinite number of strings (string length is finite), we cannot tabulate each and every string to represent the language, therefore like automata, grammar is also a mathematical model of representing a language, using which we can generate the entire language. Therefore, a grammar is usually thought of as a language generator.

Formal Grammar

BASIC DEFINITIONS

- A phrase-structure grammar (or simply a grammar) is a 4-tuple (VN, ∑, P, S), where
- VN is a finite nonempty set whose elements are called variables,
- ∑ is a finite nonempty set whose elements are called terminals, $VN \cap \Sigma = \Phi$.
- S is a special variable (i.e., an element of VN ($S \in Vn$)) called the start symbol. Like every automaton has exactly one initial state, similarly every grammar has exactly one start symbol.
- P is a finite set whose elements are $\alpha \rightarrow \beta$. where α and β are strings on $VN \cup \Sigma$. α has at least one symbol from VN, the element of P are called productions or production rules or rewriting rules. $\{\Sigma \cup Vn\}^*$ some writer refers it as total alphabet

For a formal valid production,

$$\alpha \rightarrow \beta$$

$$\alpha \in \{\Sigma \cup V_n\}^* V_n \{\Sigma \cup V_n\}^*$$

$$\beta \in \{\Sigma \cup V_n\}^*$$

Some points to note about productions

- Reverse substitution is not permitted. For example, if S → AB is a production, then we can replace S by AB but we cannot replace AB by S.
- No inversion operation is permitted. For example, if S → AB is a production, it is not necessary that AB →S is a production.
- To generate a string in the language, one begins with a string consisting of only a single start symbol. The production rules are then applied in any order, until a string that contains neither the start symbol nor designated nonterminal symbols is produced. A sequence of rule applications is called a derivation.
- A production rule is applied to a string by replacing one occurrence of the production rule's left-hand side in the string by that production rule's right-hand side.

Defining a language by grammar

The concept of defining a language using grammar is, starting from a start symbol using the production rules of the grammar any time, deriving the string. Here every time during derivation a production is used as its LHS is replaced by its RHS, all the intermediate stages(strings) are called sentential forms. The language formed by the grammar consists of all distinct strings that can be generated in this manner.

$L(G) = \{w \mid w \in \Sigma^*, S \rightarrow^* W\}$

$\rightarrow^*$(reflexive, transitive closure) means from s we can derive w in zero or more steps

Using same idea, we do process of natural languages in computers, actively used in compliers

L (G) is the set of all terminal strings derived from the start symbol S.

G1 and G2 are equivalent if L (G1) = L (G2).

Let us take some examples to understand how to find L (G).

Example:

If $G = (\{S\}, \{0, 1\}, \{S \rightarrow 0S1, S \rightarrow \Lambda\}, S)$, find $L(G)$.

Ans. $S \rightarrow 0S1 \rightarrow 0^2S1^2 \rightarrow \ldots \rightarrow 0^n1^n$**, The string will end using S→^**

Therefore,

$0^n1^n \in L(G)$

Type 2 Grammar

- Also known as Context Free Grammar, which will generate context free language that will be accepted by push down automata. (NPDA default case)
- if there is a production, from
- $\alpha \rightarrow \beta$
- $\alpha \in Vn$ $|\alpha| = 1$
- $\beta \in \{\Sigma \cup Vn\}^*$
- In other words, the L.H.S. has no left context or right context.
- A grammar is called a type 2 grammar if it contains only type 2 productions.
- Eg ALGOL 60, PASCAL
- CFG possess both left and right associative property.
- While constructing the parse tree for any i/p string, we need to expand the parse in both left and right association

Derivation: - The process of deriving a string is known as derivation.

Derivation/ syntax/ parse tree: - The geometrical representation of derivation is known as derivation tree.

Sentential form: - intermediate step involve in the derivation is known as sentential form

	LMD	RMD
E → E + E	**E**	**E**
E→ E * E	**E*E**	**E+E**
E→ E = E	**E+E*E**	**E+E*E**
E→ id	**ID+E*E**	**E+E*ID**
	ID+ID+E	**E+ID+ID**
	ID+ID+ID	**ID+ID+ID**

left most derivation: - the process of construction of parse tree by expanding the left most non terminal is known as LMD and the graphical representation of LMD id known as LMDT (left most derivation tree)

right most derivation: - the process of construction of parse tree by expanding the right most non terminal is known as RMD and the graphical representation of RMD id known as RMDT (right most derivation tree)

Q Consider the CFG with {S, A, B) as the non-terminal alphabet, {a, b) as the terminal alphabet, S as the start symbol and the following set of production rules **(GATE-2007) (1 Marks)**

S --> aB	S --> bA
B --> b	A --> a
B --> bS	A --> aS
B --> aBB	A --> bAA

Which of the following strings is generated by the grammar?

(A) aaaabb **(B)** aabbbb **(C)** aabbab **(D)** abbbba

Answer: (C)

Q For the correct answer strings to above question, how many derivation trees are there? **(GATE-2007) (1 Marks)**

(A) 1 **(B)** 2 **(C)** 3 **(D)** 4

Answer: (B)

Recursive production: - the production which has same variable both at left- and right-hand side of production is known as recursive production.

S→aSb

S→aS

S→Sa

Recursive grammar: - the grammar which contains at least one recursive production is known as recursive grammar.

S→aS/a

S→Sa/a

S→aSb/ab

Left Recursive Grammar: - The grammar G is said to be left recursive, if the Left most variable of RHS is same as the variable at LHS.

Right Recursive Grammar: - The grammar G is said to be right recursive, if the right most variable of RHS is same as the variable at LHS.

General recursion: - the recursion which is neither left nor right is called as general recursion.

If a CFG generates infinite number of string then it must be a recursive grammar.

Non recursive grammar: - the grammar which is free from recursive production is called as non-recursive grammar.

S→AaB

A→a

B→b

Process of making a CFG Compiler Friendly

- If a CFG contains left recursion then the compiler may go to infinite loop, hence to avoid the looping of the compiler, we need to convert the left recursive grammar into its equivalent right recursive production.

Q Consider the following Left recursive grammar and convert them into Corresponding Right recursive Grammar?

1) $A \rightarrow A\alpha \,/\, \beta$

2) $A \rightarrow A\alpha \,/\, \beta_1 \,/\, \beta_2$

3) $A \rightarrow A\alpha \,/\, \beta_1 \,/\, \beta_2$ -------/ β_n

4) $A \rightarrow A\alpha_1 \,/\, A\alpha_2 \,/\, \beta$

5) $A \rightarrow A\alpha_1 \,/\, A\alpha_2$ /----------$A\alpha_n$/ β

6) $A \rightarrow A\alpha_1 \,/\, A\alpha_2$ /----------$A\alpha_n$ / $\beta_1 \,/\, \beta_2$ -------/ β_m

7) $A \rightarrow A(A) \,/\, \alpha$

8) A → A a A / b

9) S → S a b / c

10) S → S S S / 0

11) S → S 0 S 1 / 0 / 1

12) S → A a B
A → Aa / bB / a
B → bB / c

13) E → E + T
T → T * F
F → (E) / id

14) E → E + E / E + E / (E) / id

15) R → R * / R R / (R) / id

16) S → (L) / a
L → L , S / S

17) S → A a B
A → S A c / a
B → B a / b

Q Consider the following expression grammar G. **(GATE-2017) (2 Marks)**
E -> E - T | T
T -> T + F | F
F -> (E) | id
Which of the following grammars are not left recursive, but equivalent to G.?

A)	B)	C)	D)
E -> E - T | T T -> T + F | F F -> (E) | id	E -> TE' E' -> -TE' | ε T -> T + F | F F -> (E) | id	E -> TX X -> -TX | ε T -> FY Y -> +FY | ε F -> (E) | id	E -> TX | (TX) X -> -TX | +TX | ε T -> id

Q Which one of the following grammars is free from left recursion? **(Gate-2016) (1 Marks)**

(A) $S \rightarrow AB$
$A \rightarrow Aa \mid b$
$B \rightarrow c$

(B) $S \rightarrow Ab \mid Bb \mid c$
$A \rightarrow Bd \mid \varepsilon$
$B \rightarrow e$

(C) $S \rightarrow Aa \mid B$
$A \rightarrow Bb \mid Sc \mid \varepsilon$
$B \rightarrow d$

(D) $S \rightarrow Aa \mid Bb \mid c$
$A \rightarrow Bd \mid \varepsilon$
$B \rightarrow Ae \mid \varepsilon$

Answer: (B)

Ambiguous grammar

Ambiguous grammar: - the grammar G is said to be ambiguous if there more than derivation tree for any input string i.e. if there exist more than one LMDT or RMDT, the grammar is said to be ambiguous.

S→aS/Sa/a

Unambiguous grammar: - The grammar G is said to be unambiguous if there exist only one parse tree for every i/p string i.e. if there exist only one LMDT or RMDT, then the grammar is unambiguous e.g.

S→aSb/ab

- Some cfl are called inherently ambiguous means there exist no unambiguous CFG to generate the corresponding CFL, proved by Rohit Parikh 1961, $\{a^n b^m c^m d^m\} \cup \{a^n b^n c^m d^m\}$
- Grammar which is both left and right recursive is always ambiguous, but the ambiguous grammar need not be both left and right recursive.
- DCFL takes linear time, FL takes polynomial time

1) S → Sa / aS / ∈

2) S → SS / 0 / 1

3) S → SaS / b

4) S → SaSbS / ∈

5) S → SAB
S → AaB / a
S → AS / b

6) S → aSa / bSb / ∈

7) S → aSb / bSa / ∈

8) S → aAB
A → aA / BaA / a
B → Aa / b

9) A → A(A) / a

10) S → (L) / a
L → L , S / S

11) S → AA
A → aA / b

Q Which one of the following statements is FALSE? **(GATE-2004) (1 Marks)**
(A) There exist context-free languages such that all the context-free grammars generating them are ambiguous
(B) An unambiguous context free grammar always has a unique parse tree for each string of the language generated by it.
(C) Both deterministic and non-deterministic pushdown automata always accept the same set of languages
(D) A finite set of string from one alphabet is always a regular language.
Answer: (C)

Q A grammar that is both left and right recursive for a non – terminal, is
a) Ambiguous
b) Unambiguous
c) information is not sufficient to decide
d) None of these

Q The grammar **(NET-JUNE-2012)**
S → aB / bA
A → a / aS / bAA
B → b / bS / aBB
a) Generates an equal number of a's and b's
b) Generates an inherently ambiguous language
c) Generates more a's than b's
d) Generates an unequal number of a's and b's
Ans: a

Q Consider the following statements about the context free grammar **(GATE-2006) (1 Marks)**
G = {S → SS, S → ab, S → ba, S → E}
I. G is ambiguous
II. G produces all strings with equal number of a's and b's
III. G can be accepted by a deterministic PDA.

Which combination below expresses all the true statements about G?

(A) I only

(B) I and III only

(C) II and III only

(D) I, II and III

Answer: (B)

Q Let G = ({S}, {a, b} R, S) be a context free grammar where the rule set R is **(GATE-2003) (1 Marks)**

S → a S b | SS | ε

Which of the following statements is true?

(A) G is not ambiguous

(B) There exist x, y, ∈ L (G) such that xy ∉ L(G)

(C) There is a deterministic pushdown automaton that accepts L(G)

(D) We can find a deterministic finite state automaton that accepts L(G)

Answer: (C)

Q Which of the following statements is/are TRUE?

(i) The grammar S → SS | a is ambiguous (where S is the start symbol).

(ii) The grammar S → 0S1 | 01S | e is ambiguous (the special symbol e represents the empty string and S is the start symbol).

(iii) The grammar (where S is the start symbol). **(NET-NOV-2017)**

S → T/U

T → x S y / xy / e

U → yT

generates a language consisting of the string yxxyy.

a) Only (i) and (ii) are TRUE

b) Only (i) and (iii) are TRUE

c) Only (ii) and (iii) are TRUE

d) All of (i), (ii) and (iii) are TRUE

Ans: d

Q Given the following grammars: G1: S → AB | aaB, A → aA | ∈, B → bB | ∈ G2: S → A|B, A → aAb | ab, B → abB | ∈ Which of the following is correct? **(NET-JUNE-2015)**

a) G 1 is ambiguous and G 2 is unambiguous grammars

b) G 1 is unambiguous and G 2 is ambiguous grammars

c) both G 1 and G 2 are ambiguous grammars

d) both G 1 and G 2 are unambiguous grammars

Ans: c

Q Given the following two grammars: **(NET-JUNE-2014)**
G1: S → AB | aaB A → a | Aa B → b
G2: S → a S b S | b S a S | λ
Which statement is correct?
(A) G1 is unambiguous and G2 is unambiguous.
(B) G1 is unambiguous and G2 is ambiguous.
(C) G1 is ambiguous and G2 is unambiguous.
(D) G1 is ambiguous and G2 is ambiguous.
Ans: d

Q Given the production rules of a grammar G1 as
S1 → AB | aaB
A → a | Aa
B → b
and the production rules of a grammar G2 as S2 → aS2bS2 | bS2aS2 | λ Which of the following is correct statement? **(NET-JUNE-2013)**
(A) G1 is ambiguous and G2 is not ambiguous.
(B) G1 is ambiguous and G2 is ambiguous.
(C) G1 is not ambiguous and G2 is ambiguous.
(D) G1 is not ambiguous and G2 is not ambiguous.
Ans: b

Q The grammar 'G1' S→ OSO| ISI | 0|1| and the grammar 'G2' is S→ as |asb| X, X → Xa | a. Which is the correct statement? **(NET-DEC-2012)**
(A) G1 is ambiguous, G2 is unambiguous
(B) G1 is unambiguous, G2 is ambiguous
(C) Both G1 and G2 are ambiguous
(D) Both G1 and G2 are unambiguous
Ans: b

Q Consider the following two Grammars: **(NET-JULY-2018)**
G1: S → SbS | a
G2: S → aB | ab, A → GAB | a, B → Abb | b
Which of the following option is correct?
(1) Only G1 is ambiguous
(2) Only G2 is ambiguous
(3) Both G1 and G2 are ambiguous
(4) Both G1 and G2 are not ambiguous
Ans: c

Non-Deterministic Grammar

- The grammar with common prefix is known as Non-Deterministic Grammar.
- $A \rightarrow \alpha\beta 1 / \alpha\beta 2$

- The grammar with common prefixes requires a lot of Back-tracking, back-tracking is very time consuming.
- To avoid the back-tracking, we need to remove the common prefixes, i.e. we need to convert the non-deterministic Grammar into Deterministic.
- Left Factoring: - The process of conversion of Non-Deterministic grammar into deterministic grammar is known as Left-Factoring.
- $A \rightarrow \alpha\beta 1 / \alpha\beta 2$

- $A \rightarrow \alpha\beta$
- $A \rightarrow \beta 1 / \beta 2$

- Consider the following non-deterministic grammar and convert them into deterministic grammar by the process of left factoring.

1) S → aSb / abS / ab

2) S → ab / abc / abcd / b

3) S → aAb / aABc / aABcd / aA / a

Backus-Naur Form (BNF)

$A \rightarrow \alpha 1$
$A \rightarrow \alpha 2$
$A \rightarrow \alpha 3$

$A \rightarrow \alpha 1 / \alpha 2 / \alpha 3$

Simplification or Minimization of CFG

- The process of deleting and eliminating of useless symbols, unit production and null production is known as simplification of CFG.

Removal of Null or Empty productions

the production of the form A → ∈ is known as null production of empty production, here we try to remove them by replacing equivalent derivation.

1) S → AbB
   A → a / ∈
   B → b / ∈

2) S → AB
   A → a / ∈
   B → b / ∈

3) S → aSb / ∈

Removal of unit productions

the production of the form A → B where A, B ∈ Vn, |A| = |B| = 1, is known as unit production.

1) S → Aa
   A → a / B
   B → d

2) S → aAb
   A → B / a
   B → C / b
   C → D / c
   D → d

3) S → aSb / ∈

Removal of useless symbols

The variables which are not involved in the derivation of nay string is known as useless symbol. Select the variable that cannot be reached from the start symbol of the grammar and remove them along with their all production.

Select variable that are reachable from the start symbol but which does not derive any terminal, remove them along with their productions

1) $S \rightarrow aAB$
   $A \rightarrow a$
   $B \rightarrow b$
   $C \rightarrow d$

2) $S \rightarrow aA$ / aB
   $A \rightarrow b$

3) $S \rightarrow aAB$ / bA / aC
   $A \rightarrow aB$ / b
   $B \rightarrow aC$ / d

Q Consider the grammar consisting of 7 productions
$S \rightarrow aA \mid aBB$
$A \rightarrow aaA \mid \lambda$
$B \rightarrow bB \mid bbC$
$C \rightarrow B$
After elimination of Unit, useless and λ – productions, how may production remain in the resulting grammar?

a) 2 **b)** 3 **c)** 4 **d)** 5

Q Consider the CFG G (V, T, P, S) with the following production
$S \rightarrow AB \mid a$
$A \rightarrow a$
Let CFG G' is an equivalent CFG with no useless symbols. How many minimum productions will be there in G'?

a) 1 **b)** 2 **c)** 3 **d)** 5

Q $S \rightarrow XY$ $X \rightarrow ZB$ $Y \rightarrow BW$ $Z \rightarrow AB$ $W \rightarrow Z$ $A \rightarrow aA/Ba/\epsilon$ $B \rightarrow Ba/Bb/\epsilon$
After elimination of null productions from the CFG, the output is _______________

Q For the grammar give below, what is the equivalent CFG without useless symbols S→AB/a, S →a

a) S →A, A → a　　**b)** S → a　　**c)** A→ a　　**d)** S → A/a, A → a

Q For the given grammar below, what is the equivalent CFG without ϵ - productions?

S → AB, A → ϵ | a | aS, B → b | bS

a) S → AB, A → a | aS, B → b | bS

b) S → A|B|AB, A → a | aS, B → b | bS

c) S → B|AB, A → a | aS, B → b | bS

d) Not possible to remove ϵ production

Q For the simplification of a given CFG, the sequence of steps to be followed is

a) Remove ∈-productions, unit productions and useless productions in that order

b) Remove unit productions, ∈-productions, and useless productions in that order

c) Remove unit productions, useless productions and ∈-productions in that order

d) Remove ∈-productions, useless productions and unit productions in that order

Normalization of CFG

- The Grammar G is said to be in Chomsky Normal Form, if every production is in the form

A → BC / a
BC ∈ Vn
a ∈ Σ

1) S → aSb / ab
A → B / a
B → C / b

2) S → aAb / bB
A → a / b
B → b

- if CFG is in CNF, then for a derivation of string w, with length we need exactly 2n -1 production. |w| = n, number of sentential forms will be 2n -1

Q To obtain a string of n Terminals from a given Chomsky normal form grammar, the number of productions to be used is: **(NET-JULY-2018)**
(1) 2n−1 **(2)** 2n **(3)** n+1 **(4)** n^2

Q If G is a context – free grammar and W is a string of length l in L(G), how long is a derivation of W in G, if G is Chomsky normal form? **(GATE – 1992) (1 Marks)**
a) 2l **b)** 2l + 1 **c)** 2l – 1 **d)** l

Q If the parse tree of a word w generated by a Chomsky normal form grammar has no path of length greater than i, then the word w is of length **(NET-DEC-2012)**
(A) no greater than 2i+1 **(B)** no greater than 2i
(C) no greater than 2i–1 **(D)** no greater than i

Q Consider the grammar S → PQ |SQ | PS, P → x, Q → y. To get string of n terminals, the of productions to be used is
a) n^2 **b)** 2n **c)** 2^{n+1} **d)** 2n – 1

Which of the following statements are true?

I. Every left-recursive grammar can be converted to a right-recursive grammar and vice-versa

II. All ϵ- productions can be removed from any context-free grammar by suitable transformations

III. The language generated by a context-free grammar all of whose productions are of the form $X \rightarrow w$ or $X \rightarrow wY$ (where, w is a string of terminals and Y is a non-terminal), is always regular

IV. The derivation trees of strings generated by a context-free grammar in Chomsky Normal Form are always binary trees

1.I, II, III and IV **2.II, III and IV only**
3.I, III and IV only **4.I, II and IV only**
ANSWER C

- The Grammar G is said to be in Greibeck Normal Form, if every production is in the form

$A \rightarrow a\alpha$
$A \in Vn$
$a \in \Sigma$
$\alpha \in Vn^*$

1) S → aSb / ab
A → B / a
B → C / b

2) S → aAb / bB
A → a / b
B → b

- if CFG is in CNF, then for a derivation of string w, with length we need exactly n production. |w| = n, number of sentential forms will be n

Q A CFG is said to be in Griebach Normal Form (GNF), if all the productions are of the form A → aX where X is a sequence of any number of variables. Let G be a CFG in GNF. To derive a string of terminals of length x, the number of productions to be used is

a) 2x – 1 **b)** x **c)** 2x + 1 **d)** 2^x

Q The Greibach normal form grammar for the language L = $\{a^n b^{n+1} \mid n \geq 0\}$ is **(NET-DEC-2013)**

(A) S → a SB, B → bB | λ **(B)** S → a SB, B → bB | b

(C) S → a SB | b, B → b **(D)** S → a Sb | b

Ans: c

Q Which one of the following is not a Greibach Normal form grammar? **(NET-JUNE-2012)**

(i) S → a | bA | aA | bB, A → a, B → b

(ii) S → a | aA | AB, A → a, B → b

(iii) S → a | A | aA, A → a

(A) (i) and (ii) **(B)** (i) and (iii)

(C) (ii) and (iii) **(D)** (i), (ii) and (iii)

Ans: c

Q Let G = (V, T, S, P) be a context-free grammar such that every one of its productions is of the form A → v, with |v| = K > 1. The derivation tree for any W ∈ L(G) has a height h such that (NET-AUG-2016)

a) logK|W| ≤ h ≤ logK((|W|-1)/k-1) **b)** logK|W| ≤ h ≤ logK(K|W|)

c) logK|W| ≤ h ≤ K logK|W| **d)** logK|W| ≤ h ≤ ((|W|-1)/k-1)

Decidable Properties of CFG

- The following properties are decidable for CFG Grammar G.
  - Emptiness
  - Non-emptiness
  - Finiteness
  - Infiniteness
  - Membership

Q consider the following CFG and identify which of the following CFG generate Empty language?

1) S → aAB / Aa
   A → a

2) S → aAB
   A → a / b

3) S → aAB / aB
   A → aBb
   B → aA

Q consider the following CFG and identify which of the following CFG generate Finite language?

1) S → SS
   S → AB
   A → BC / a
   B → CC / b

2) S → AB
   A → B / a

3) S → AB
   A → BC / a
   B → CC / b
   C → AB

Q consider the following CFG and check out the membership properties?

1) S → AB / BB
   A → BA / AS / b

B → AA / SB / a
B → CC / b

w = aba
w = abaab
w = abababba

Q For every context free grammar (G) there exists an algorithm that passes any $w \in L(G)$ in number of steps proportional to **(NET-JUNE-2013)**

(A) $\ln|w|$ **(B)** $|w|$ **(C)** $|w|^2$ **(D)** $|w|^3$

Ans: d

Q Consider the following expression grammar G.

E -> E - T | T

T -> T + F | F

F -> (E) | id

Which of the following grammars are not left recursive, but equivalent to G.

A) E -> E - T | T

T -> T + F | F

F -> (E) | id

B) E -> TE'

E' -> -TE' | ε

T -> T + F | F

F -> (E) | id

C) E -> TX

X -> -TX | ε

T -> FY

Y -> +FY | ε

F -> (E) | id

D) E -> TX | (TX)

X -> -TX | +TX | ε

T -> id

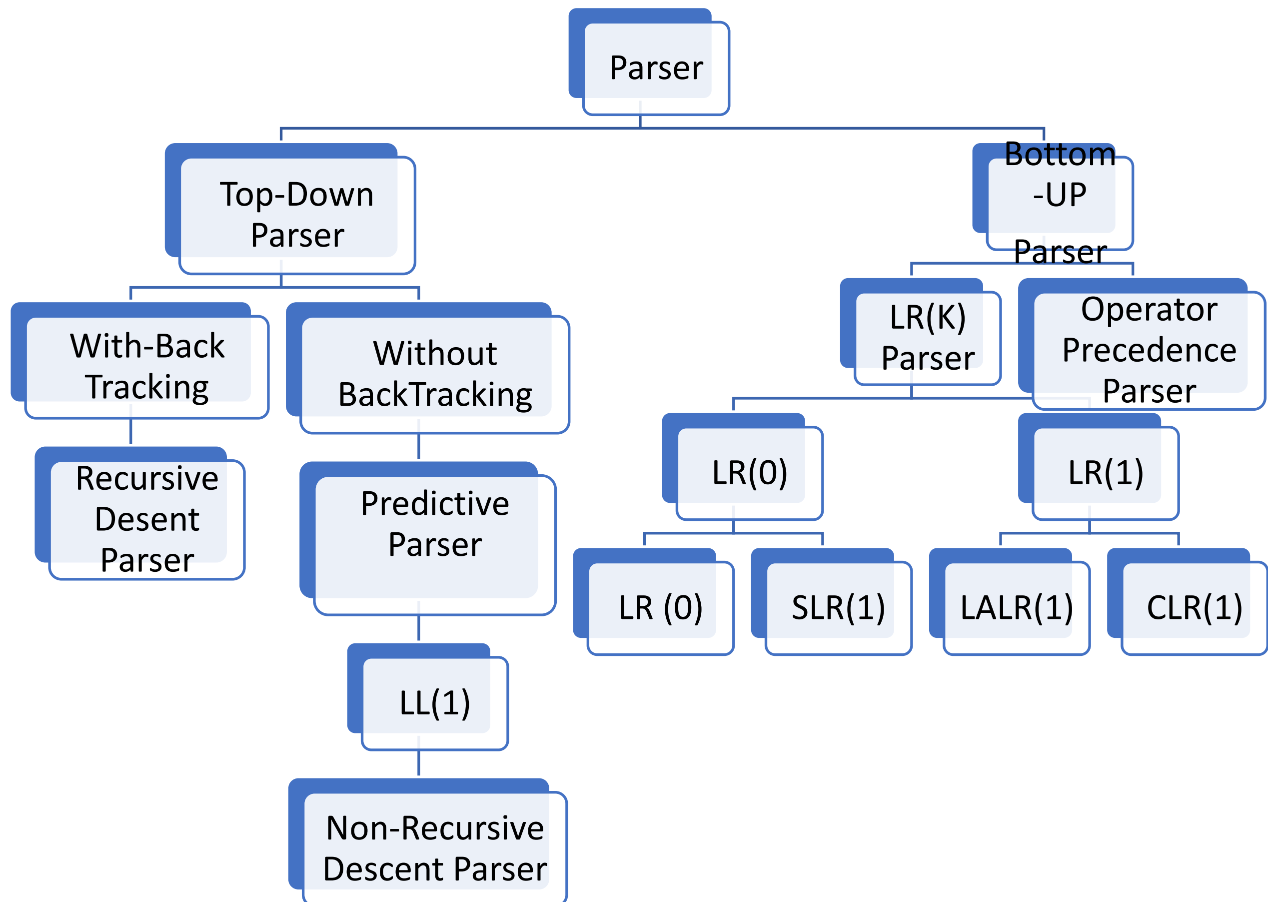
ANSWER C

Parsing basic theory

- The process of deriving the string from a given grammar using the productions of the grammar is known as parsing.
- Here the input will be stream of tokens and output will be Parse tree.
- If we get a success in generating the syntax tree then it means there is no grammatical mistakes in the string, otherwise it means a grammatical error.

Classification of parser

- The program which perform parsing is known as parser or syntax analyzer.
- There are two types of parser
    - Top-Down Parser
    - Bottom Up Parser



Top down parser

- The process of construction of parse tree, starting from root and process to children, is known as TOP down parsing, i.e. getting the i/p string by starting with a start symbol of the grammar is top down parsing.
- Top down parser uses, left most derivation.
- The TDP is constructed for the grammar, if it is free from left recursion and ambiguity.
- TDP may constructed for both left factor and non-left factor grammar.
- If the grammar is non- deterministic, then we use brute technique and if the grammar is deterministic, then we go with the predictive parser.
- The TDP is constructed for a grammar, if there is less complexity, i.e. if the complexity of grammar is more than the parsing mechanism is very slow and performance is low.
- Avg time complexity is $O(n^4)$

Q Which of the following derivations does a top-down parser use while parsing an input string? The input is assumed to be scanned in left to right order. **(GATE-2000) (2 Marks)**
(A) Leftmost derivation
(B) Leftmost derivation traced out in reverse
(C) Rightmost derivation
(D) Rightmost derivation traced out in reverse
Answer: (A)

Brute force technique or Back Tracking or Recursive Descent Parser

- Whenever a non- terminal is expanding first time, then go with the first alternative and compare with the i/p string. if does not matches, go for the second alternative and compare with i/p string, if does not matches go with the 3rd alternative and continue with each and every alternative.
- if the matching occurs for at least one alternative, then the parsing is successful, otherwise parsing is failed.

S → cAd

A → ab / a

w = cad

S → Sa / b

w = baaa

S → aAc / aB

A → b / c

B → ccd / ddc

w = addc

- Brute force requires lot of back-tracking, takes $O(2^n)$
- Back Tracking is very costly & reduces the performance of parser
- Debugging is very difficult.

```
Void A ()
{
        choose a production A → X1, X2, X3, X4, ----, XK
        for (i=1 to k)
        {
                if (X1 is non-terminal)

                        X1()

                Else

                        if (Xi == lookahead symbol)
                                increment i/p

                        Else
                                error(backtracking)
        }

}
```

S → ABCDE / BCDE / CDE / DE

A → a

B → b

C → c

D → d

E → e

First

Q Consider the following Grammar find the First and follow for each of them?

FIRST(α) is calculated for both Terminals and Non-Terminals
FOLLOW(X) is calculated for non-Terminals

First(α) is a set of all terminals that may be in beginning in any sentential form, derived from α

Rules for finding First(α)

1) if α is a terminal, then

First(α) = {α}

2) if α is a non - terminal, defined by α → ε, then

First(α) = {ε}

3) if α is a non - terminal, defined by α → β, β ∈ T

First(α) = {β}

4) if α is a non - terminal, defined by α → $X_1 X_2 X_3$, then

First(α) = First(X_1)	**iff** X_1 →! ∈
First(α) = First(X_1) U First(X_2)	**iff** X_1 → ∈ && **iff** X_2 →! ∈
First(α) = First(X_1) U First(X_2) U First(X_2)	**iff** X_1 → ∈ && X_2 → ∈ && **iff** X_3 →! ∈
First(α) = First(X_1) U First(X_2) U First(X_2) U ∈	**iff** X_1 → ∈ && X_2 → ∈ && **iff** X_3 → ∈

S → a / b / ε

S → aA / bB

A → ε

B → ε

S → aAb / Ba

A → aA / b

B → c / d

S → AaB / BA

A → a / b

B → d / e

S → AaB

A → b / ε

B → c

S → AaB / Bb

A → bA / ε

B → cB / ε

S → AB

A → a / ε

B → b / ε

S → ABCDE

A → a / ε

B → b / ε

C → c / ε

D → d

E → e / ε

S → aAB / BA

A → BA / ε

B → AB / a / ε

S → AA

A → aA

A → b

S → (L) / a

L → SL’

L’ → ,SL’ / ε

E → TE’

E’ → +TE’ / ε

T’ → FT’

T’ → *FT’ / ε

F → (E) / id

A → aA’

A’ → (A)A’ / ε

A → (A)A’ / aA’

A’ → AA’ / ε

S → aBDh

B → cC

C → bC / ε

D → EF

E → g / ε

F → f / ε

S → 0S’ / 1S’

S’ → 0S1S’ / ε

R → (R)R’ / aR’ / bR’

R’ → +RR’ / RR’ / *R’ / ε

Follow

Follow(A) is the set of all terminals that may follow to the right of (A) in any form of sentential Grammar.

Rules:

1) if A is the start symbol then Follow(A) = {$}

2) if $A \rightarrow \alpha A\beta$, $\beta \rightarrow ! \in$

Follow(A) = First(β)

3) if $S \rightarrow \alpha A$

Follow(A) = Follow(S)

4) $S \rightarrow \alpha A\beta$, where $\beta \rightarrow \in$

Follow(A) = First(β) U Follow(S) - ∈

$S \rightarrow \varepsilon$

$S \rightarrow aA$
$A \rightarrow bA / \varepsilon$

$S \rightarrow AB / BA$
$A \rightarrow a / \varepsilon$
$B \rightarrow b / \varepsilon$

S → AaBb / BbAa

A → ε

B → ε

S → ABCDE

A → a / ε

B → b / ε

C → c / ε

D → d / ε

E → e

E → TE’

E’ → +TE’ / ε

T → FT’

T’ → *FT’ / ε

F → (E) / id

E → E+T / T

T → T*F / F

F → (E) / id

S → aAb

A → Ba / b

B → d

S → S0S1 / ε

S → AaBb / BaBb

A → ε

B → ε

S → aBDh

B → cC

C → bC / ε

D → EF

E → g / ε

F → f / ε

S → aBCbD

B → bBh / BD / d

C → CE / a

D → EC / b / ε

E → CDb / ε

Q Consider the following grammar

p --> xQRS

Q --> yz|z

R --> w|∈

S -> y

Which is FOLLOW(Q)? (GATE-2017) (1 Marks)

a) {R} **b)** {w} **c)** {w, y} **d)** {w, ∉}

ANSWER C

Q Consider the following given grammar:

S → Aa

A → BD

B → b | ε

D → d | ε

Let a, b, d and $ be indexed as follows:

a	b	d	$
3	2	1	0

Compute the FOLLOW set of the non-terminal B and write the index values for the symbols in the FOLLOW set in the descending order. (For example, if the FOLLOW set is {a, b, d, $}, then the answer should be 3210). **(Gate - 2019) (2 Marks)**

Ans: 31

LL(1) or Table Driven Parser

- LL(1) is constructed for the Grammar, if
  - free from left recursion
  - free from ambiguity
  - left factored

Q find which of the following grammar satisfies conditions of LL(1) Grammar ?

A → AA / a

A → aA / Ab / c

S → aA / abB / c
A → d
B → f

- LL(1)
  - First L means Left to right scanning
  - Second L means Left most derivation
  - 1 means no of look ahead symbol
- To predict the required production, to extend the parse tree, LL(1) parser depends on current processing symbol.
- The current processing symbol is called as Look-ahead-symbol.

Q Which of the following suffices to convert an arbitrary CFG to an LL(1) grammar? **(GATE-2003) (1 Marks)**
(A) Removing left recursion alone
(B) Factoring the grammar alone
(C) Removing left recursion and factoring the grammar
(D) None of these
Answer: (D)

Q Which of the following suffices to convert an arbitrary CFG to an LL(1) grammar ? **(NET-JUNE-2014)**

(A) Removing left recursion alone

(B) Removing the grammar alone

(C) Removing left recursion and factoring the grammar

(D) None of the above

Ans: d

Q Which of the following is true while converting CFG to LL(I) grammar? **(NET-DEC-2012)**

(A) Remove left recursion alone

(B) Factoring grammar alone

(C) Both of the above

(D) None of the above

Ans: d

The grammar $S \rightarrow (S) \mid SS \mid \varepsilon$ is ***not*** suitable for predictive parsing because the grammar is **(NET-DEC-2018)**

a) Right Recursive Grammar

b) Left Recursive Grammar

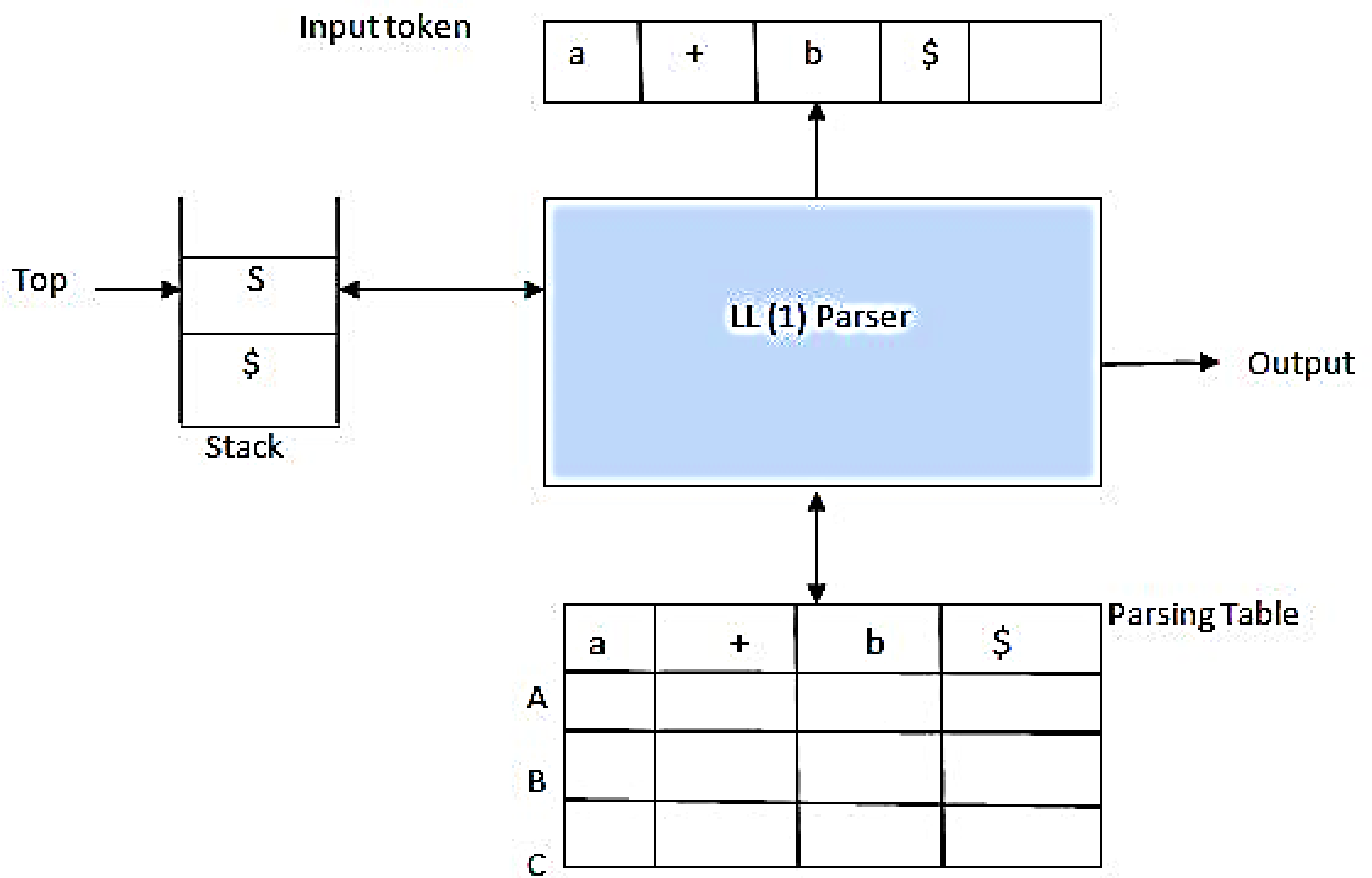
c) Ambiguous Grammar

d) Operator Grammar

Ans: c

Block-Diagram of LL(1) Parser

- LL(1) parser consist of 3 components
  - Input Buffer
  - Parse Stack
  - Parse Table



Fig(a):- Model for LL(1) Parser

Q Consider a given Grammar LL(1) grammar design Parsing table and perform complete parsing table?

A → (A) / a
w = ((a))

Stack	i/p	Action
$	((a))$	PUSH A
$ A	((a))$	use production A → (A), POP A and PUSH)A(
$)A(	((a))$	match (
$)A	(a))$	use production A → (A), POP A and PUSH)A(
$))A(	(a))$	match (
$))A	a))$	use production A → a, POP A and PUSH a
$))a	a))$	match a
$))	))$	match)
$)	)$	match)
$	$	Accepted

- Input Buffer
    - it is divided into finite no of cells and each cell is capable of holding only one i/p symbol.
    - input buffer contains only i/p string at any point of time.
    - the tape header is always pointing only one look ahead symbol and after parsing the current look ahead symbol, the header moves to next cell towards right side.
- Parse Stack
    - it contains the grammar symbol, the grammar symbol is pushed into stack or POP from the stack based on the occurrence of matching. if the topmost symbol of the stack is matching with look ahead symbol, then the grammar symbol is POP out from the stack.
    - if the TOP most symbol of the stack is not matching with the look ahead symbol, then the grammar symbol is Pushed into stack.
- Parse table
    - it is a two-dimensional array of order m*n where m = no of non-terminal and n = no of terminals + 1
    - parse table contains all the production which are used to contain the parse tree for that i/p string.
- pda alone can not simulate the LL(1) parser, since the start which is there in PDA can be used to store the Grammar symbol and can not be stored the production which are used to construct the parse tree, hence if we attach the parse table for pda, then it will stimulate the LL(1) parser.

- Parsing Process
    - push the start symbol into stack
    - compare the top most symbol of the stack with the look ahead symbol
    - if matching occurs, then pop of the grammar symbol from the stack and increment the i/p pointer
    - o/p the production which is used for expanding a non- terminal, i.e. the result is a production which is used for push operation.
- LL(1) parsing algorithm
    - let x is a grammar symbol (stack symbol) and a is the look ahead symbol
    - if x == a == $, then the parsing is successful
    - if x==a! = $, then pop of and increment the i/p pointer
    - if x!= a!=$ and m[x, a] contain the production, x → uvw, then replace a by uvw in the reverse order and continue the process.
    - out of the production which is used for expanding the non-terminal, i.e. the production which is used for PUSH operation

Q Which of the following derivations does a top-down parser use while parsing an input string? The input is scanned from left to right. **(NET-DEC-2013)**

(A) Leftmost derivation

(B) Leftmost derivation traced out in reverse

(C) Rightmost derivation traced out in reverse

(D) Rightmost derivation

Ans: a

LL(1) Grammar: the grammar for which LL(1) parser can be constructed is known as LL(1) Grammar or the Grammar whose LL(1) parse table does not contains multiple entries, then the grammar is LL(1)

- Procedure to constructed LL(1) parser:
  - for every production A → α, repeat the following steps
  - add A → α in M[A, α] for every terminal 'α' in first(A).
  - if First(α) contains ε, then add A → ε in M[A, b] for every symbol, b in Follow[A]

Q find which of the following grammar is LL(1) ?

E → E+T / T
T → T*F / F
F → (E) / id

S → AB
A → bA / ε
B → aB / ε

S → AaAb / BaBb
A → ε
B → ε

S → AaAb / BbBa
A → ε
B → ε

E → T+E / T
T → id

S → (L) / a
L → SL'
L' → ,SL' / ε

Short Cut Techniques for LL(1)

- A Grammar without ε is LL(1), if
  - for every production of the $A \rightarrow \alpha_1 / \alpha_2 / \alpha_3 / ----- / \alpha_n$, the set First($\alpha_1$), First($\alpha_1$), First($\alpha_1$),…………., First($\alpha_1$) are mutually disjoint. First(α_1) ⋂ First(α_1) ⋂ First(α_1) ⋂………… ⋂ First(α_1) = ϕ
- A Grammar with is LL(1), if
  - for every production of the $A \rightarrow \alpha / \varepsilon$
  - First(α) ⋂ First(A) = ϕ

Q find which of the following grammar is LL(1) ?

S → AB

A → a

B → b

S → aA / bB

A → Bb / a

B → bB / c

S → aAbB

A → a / ε

B → b / ε

S → AaBb / BaBb

A → ε

B → ε

S → AaBb / BbBa

A → ε

B → ε

S → aAbB

A → c / ε

B → d / ε

S → ACB / CbB / Ba

A → da / BC

B → g / ε

C → h / ε

Q A grammar G is LL(1) if and only if the following conditions hold for two distinct productions A → α | β

I. First (α) ∩ First (β) ≠ {a} where a is some terminal symbol of the grammar.

II. First (α) ∩ First (β) ≠ λ

III. First (α) ∩ Follow (A) = ϕ if λ ∈ First (β) **(NET-JUNE-2014)**

(A) I and II **(B)** I and III **(C)** II and III **(D)** I, II and III

Ans: d

Q For the grammar below, a partial LL(1) parsing table is also presented along with the grammar. Entries that need to be filled are indicated as E1, E2, and E3.   is the empty string, $ indicates end of input, and, | separates alternate right hand sides of productions**? (GATE-2012) (2 Marks)**

S → a A b B | b A a B | ε

A → S

B → S

	a	b	$
S	**E1**	**E2**	S → ε
A	A → S	A → S	error
B	B → S	B → S	**E3**

(A) FIRST(A) = {a, b, ε} = FIRST(B)
FOLLOW(A) = {a, b}
FOLLOW(B) = {a, b, $}

(B) FIRST(A) = {a, b, $}
FIRST(B) = {a, b, ε}
FOLLOW(A) = {a, b}
FOLLOW(B) = {$}

(C) FIRST(A) = {a, b, ε} = FIRST(B)
FOLLOW(A) = {a, b}
FOLLOW(B) = ∅

(D) FIRST(A) = {a, b} = FIRST(B)
FOLLOW(A) = {a, b}
FOLLOW(B) = {a, b}

Answer: (A)

Q Consider the date same as above question. The appropriate entries for E1, E2, and E3 are **(GATE-2012) (2 Marks)**

(A) E1: S → aAbB, A → S
E2: S → bAaB, B → S
E3: B → S

(B) E1: S → aAbB, S → ε
E2: S → bAaB, S → ε
E3: S → ε

(C) E1: S → aAbB, S → ε
E2: S → bAaB, S → ε
E3: B → S

(D) E1: A → S, S → ε
E2: B → S, S → ε
E3: B → S

Answer: (C)

Q 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. **(GATE-2006) (2 Marks)**

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

Answer: (A)

Q Which of the following is FALSE? **(NET-AUG-2016)**

a) The grammar S → aS | aSbS |∈, where S is the only non-terminal symbol, and ∈ is the null string, is ambiguous.

b) An unambiguous grammar has same left most and right most derivation.

c) An ambiguous grammar can never be LR(k) for any k.

d) Recursive descent parser is a top-down parser.

Ans: b

Q Which is the correct statement(s) for Non-Recursive predictive parser? **(NET-JUNE-2013)**

S1 : First(α) = {t| α ⇒ * tβ for some string β} ⇒ * tβ

S2 : Follow(X)={ a| S⇒ * αXaβ for some strings α and β}

(A) Both statements S1 and S2 are incorrect.

(B) S1 is incorrect and S2 is correct.

(C) S1 is correct and S2 is incorrect.

(D) Both statements S1 and S2 are correct.

Ans: D

Q Consider the grammar with non-terminals N = {S, C, S1}, terminals T = {a, b, i, t, e}, with S as the start symbol, and the following set of rules (Gate-2007) (2 Marks)

S --> iCtSS1|a

S1 --> eS|ϵ

C --> b

The grammar is NOT LL(1) because:

(A) it is left recursive

(B) it is right recursive

(C) it is ambiguous

(D) It is not context-free.

Answer: (C)

Bottom Up parser

- The process of constructing the parse tree in the Bottom-Up manner, i.e. starting from the children & proceeding towards root.

S → aABc
A → b / bc
B → d

w = abcde

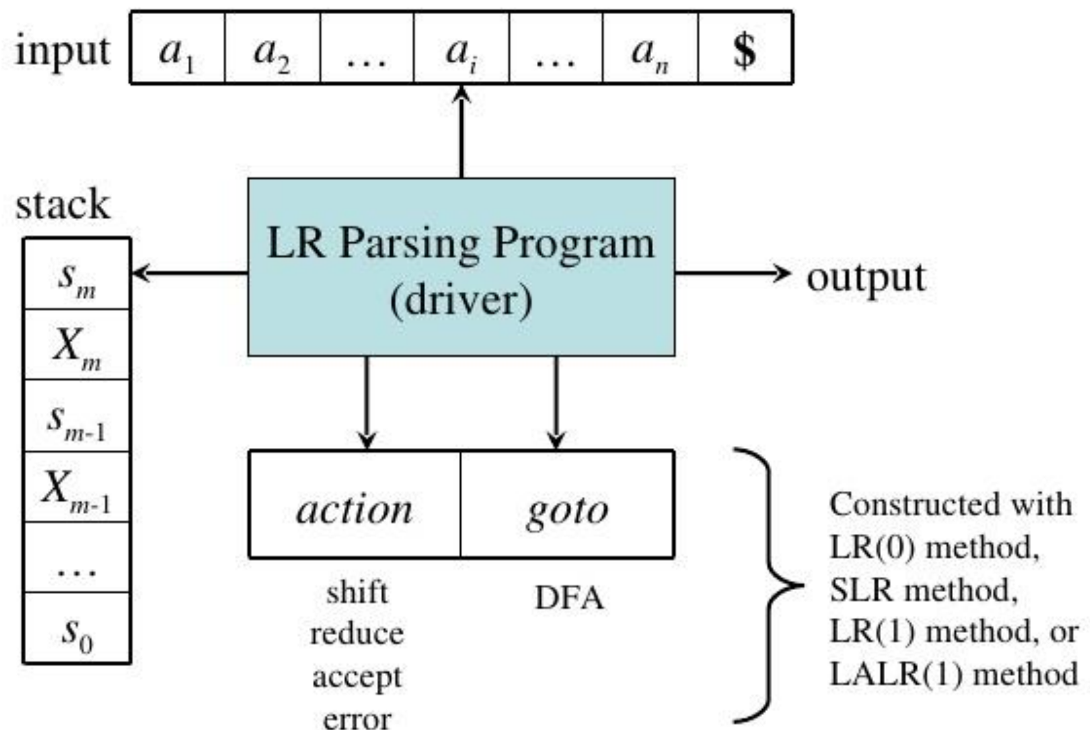
- Handle: - Substring of the i/p string that matches with RHS of any production, is called as Handle.
- The process of finding the handle & replacing that handle by it's LHS variable is called Handle Pruning.
- Bottom-Up-Parser is also known as Shit-Reduced Parser.
- BUP can be constructed for both Ambiguous & Unambiguous grammar
    - Ambiguous →OPP
    - Unambiguous → LR(k)
- LR(K) parser can be constructed for Unambiguous Parser.
- BUP Simulates the Reverse of Right Most Derivation.
- BUP can be constructed for the grammar which has more complexity.
- Bottom-up parsing is faster than Top-Up-Parsing, such that BUP is more efficient than TDP
- The performance of BUP is very high. avg time complexity is $O(n^3)$
- Handle pruning is the overhead for bottom-UP-parsing.

S → AA

A → aA / b

Stack	i/p	Action
$	abab$	Shift
$a	bab$	Shift
$ab	ab$	Reduce (A → b)
$aA	ab$	Reduce (A →aA)
$A	ab$	Shift
$Aa	b$	Shift
$Aab	$	Reduce (A →b)
$AaA	$	Reduce (A → aA)
$ AA	$	Reduce (A → AA)
$A	$	

- Bottom Up Parser consist of three components
    - i/p buffer
    - Parse stack
    - Parse table



- Input buffer: divide into cells & each cell contains only one i/p symbol.
- Stack: stack contains the grammar symbol, the grammar symbol are push into stack or pop from the stack, using shift & reduced operation. if handle occurs from the topmost symbol of the stack, then apply the reduced operation & if handle does not occurs in the topmost symbol of the stack, then apply the shift operation.
- Parse table: parse table is constructed using terminals, non-terminals & LR(0) items. this parse table consist of two parts:
    - Action
    - Goto

- Action part contains shift & reduced operation over the terminals
- Goto part consists of only Shift operation over the Non-terminals.

- Operation in shift/reduced passer
  - shift
  - reduced
  - accept
  - error

	Action	Goto
I0	Terminals	Non-Terminals
In-1	Shift / Reduce	Shift

- Shift: shift operation can be used when handle does not occur from the topmost symbol of the stack. using shift operation, will moving a look ahead symbol in stack.
- Reduce: reduce operation can be whenever handle occurs from the Topmost symbol from the stack. using reduced operation, we rename the topmost symbol of the stack that matches with look-ahead symbol.
- Accept: after scanning the complete i/p string from the i/p buffer, if the stack contains only the start symbol of the grammar as topmost symbol, then the i/p string is accepted and the parsing is successful.
- Error: after the complete i/p string, if the attack contains any symbol which is different from start symbol as a topmost symbol, then the parsing is unsuccessful and hence error

Q A → aA / b

I_0: Closure (A' → .A)
A' → .A
A → .aA
A → .b

I_1: Goto (I_0, A)
A' → .A

I_2: Goto (I_0, a)
A → a.A
A → .aA
A → .b

I_3: Goto (I_0, b)
A → b.

I_4: Goto (I_2, A)
A → aA.

I_5: Goto (I_2, a)
A → a.A
A → .aA
A → .b

I_6: Goto (I_2, b)
A → b.

	a	b	$	A
I_0	s2	s3		1
I_1			accept	
I_2	s2	s3		4
I_3	r2	r2	r2	
I_4	r1	r1	r1	

LR Parser:

- LR(K)
    - First L stand for left to right scanning
    - Second R stand for reverse of right most derivation
    - K is Look-Ahead symbol

- Procedure for the construction of LR Parser table:
    - Obtain the augmented grammar for the given grammar
    - Create the canonical collection of LR items or compiler items.
    - Draw the DFA & prepare the table based on LR items.

- Augmented grammar
    - The grammar which is obtained by addition one more production that start symbol of the grammar, is known as Augmented grammar.
    - S → AB A → a B→ b
    - S' → S S → AB A → a B→ b

LR(0)

- LR(0) or Compiler item
  - The production, which has dot(.) anywhere on RHS is known as LR(0) items.
  - A → abc
  - LR(0) items:
    - A → .abc
    - A → a.bc
    - A → ab.c
    - A → abc.   Final / Completed items

- Canonical Collection:
  - The set $C = \{I_0, I_1, I_2, I_3, \ldots\ldots\ldots I_N\}$ is known as canonical collection of LR(0) items.

- Function used to generate LR(0) item's:
  - Closure: i/p set of items & o/p also set of items
  - GOTO

- Add everything from i/p to o/p
- If A → α.Bβ is in closure(I) & β → б is in the grammar G
  - Then add β → .б to the closure(I)
  - A → α.Bβ
  - β → .б
- Repeat the previous step for every newly added item

- Goto(I, X)
  - Goto(I, X)
  - Goto(A → α.Xβ, X) = A → αX.β

- Procedure to construct LR parse Table:
- LR parse table consist of two parts
    - Action
    - Goto

- Action
    - Action part consists of both shift & reduced operation that are performed on terminals.
- Goto
    - Goto parts consists of shift operation performed on Non-terminals

Goto(I_i, X) = I_j (X is terminal)

	X
$\mathbf{I_i}$	S_j (I_j)

Goto(I_i, X) = I_j (X is non-terminal)

	X
$\mathbf{I_i}$	j (I_j)

If I_i is any final item & represent the production R_i, then place R_i under all the terminal symbols in the action part of the table.

	t_1	t_2	t_3					t_n	$
I	r_i	r_i	r_i					r_i	r_i

E → T + E / T

T → id

E → E + T / T

T → T * F / F

F → id

S → AA

A → aA / b

S → (L) / a

L → L,S / S

S → AaAb / BbBa

A → ε

B → ε

- Conflicts in LR(0) parser
  - SR Conflicts (Sift Reduce)
  - RR Conflicts (Reduce Reduce)

If the state of DFA contains both final & non-final items, then it is S-R conflicts

A → α.xβ (x is terminal, shift)

B →б. (reduced)

R-R Conflict: If the same state contains more than one final item, then it is R-R Conflict

A → α. (x is terminal, shift)

B →α. (reduced)

LR(0) Grammar: An unambiguous grammar LR(0) parse table is free from multiple entries, i.e. free both SR & RR conflicts, is LR(0) grammar.

Q Consider the following grammar. **(GATE-2006) (2 Marks)**

S -> S * E

S -> E

E -> F + E

E -> F

F -> id

Consider the following LR(0) items corresponding to the grammar above.

(i) S -> S * .E

(ii) S -> F. + E

(iii) S -> F + .E

Given the items above, which two of them will appear in the same set in the canonical sets-of-items for the grammar?

(A) (i) and (ii) **(B)** (ii) and (iii) **(C)** (i) and (iii) **(D)** None of the above

Answer: (d)

S → aAB / Ba / Ba

A → c

B → c

S → Aab / Ba
A → aA / a
B → Ba / b

E → E + T / T
T → T * F / F
F → (E) / id

E → E + T / T
T → EF / Ea / b
F → (E) / a

S → AB / BA
A → Aab / b
B → BaA / a

S → A#a / @aB
A → $A / #

A → A(A) / bA / a

S → Aab / bab / bac / acb
A → aBA / b
B → b

E → E F / e
F → FT / f
T → ET / g

S → aAb / bB / ac
A → bA / c
B → bB / c

Q Consider the grammar **(Gate-2005) (2 Marks)**
E → E + n | E × n | n
For a sentence n + n × n, the handles in the right-sentential form of the reduction are
(A) n, E + n and E + n × n
(B) n, E + n and E + E × n
(C) n, n + n and n + n × n
(D) n, E + n and E × n
Answer: (D)
Explanation:

E → E + n {Applying E → E + n }

→ E + E * n {Applying E → E * n }

→ E + n * n {Applying E → n }

→ n + n * n {Applying E → n }

Q Which one of the following is True at any valid state in shift-reduce parsing? **(Gate - 2015) (1 Marks)**
(A) Viable prefixes appear only at the bottom of the stack and not inside
(B) Viable prefixes appear only at the top of the stack and not inside
(C) The stack contains only a set of viable prefixes
(D) The stack never contains viable prefixes
Answer: (C)

SLR(1)

- The procedure for constructing the parse table is similar to LR(0) parse, but there is a restriction in the reducing the entries.
- whenever there is a final item, then placed the reduced entries under the follow symbol of LHS symbol.
- if the SLR(1) parse table is free from multiple entries than the grammar is SLR(1) grammar.

Conflicts in SLR (1):

- Conflicts in LR(0) parser
  - SR Conflicts (Sift Reduce)
  - RR Conflicts (Reduce Reduce)

If the state of DFA contains both final & non-final items, then it is S-R conflicts

$A \rightarrow \alpha.x\beta$ (x is terminal, shift)

$B \rightarrow б.$ (reduced)

if Follow(B) $\cap$ {x} != ɸ

R-R Conflict: If the same state contains more than one final item, then it is R-R Conflict

$A \rightarrow \alpha.$ (x is terminal, shift)

$B \rightarrow \alpha.$ (reduced)

if Follow(A) $\cap$ Follow(B) != ɸ

every LR(0) grammar is SLR(1), but every SLR(1) grammar need not be LR(0)

SLR(1) parser is more powerful then LR(0) parser

The no of entries in SLR(1) parser table <= no of entries in LR(0) parse table

$E \rightarrow T + E / T$

$T \rightarrow id$

S → AaB

A → ab / a

B → b

S → Aa / bAc / dc / bda

A → ε

S → Aa / bAc / dc / bda

A → d

S → AS / b

A → SA / a

S → AaAb / BbBa

A → ε

B → ε

S → dA / aB

A → bA / c

B → bB / c

S → A / a

A → a

S → (L) / a

L → L,S / S

E → E + T / T

T → id

E → E + T / T

T → TF / F

F → F* / a / b

CLR(1)

- LR(0) does not depends on look ahead symbol
- LR(1) depends on one look Ahead symbol

- Closure(I):
  - Add everything from i/p to o/p
  - A → α.Bβ, $ is in closure I and β → б is in the grammar G, then add β → .б, first(β, $), to the closure I
  - repeat previous step for every newly added items

- Goto(I, x):
  - there will not be any change in the goto part while finding the transition.
  - these many be change in the follow or look Ahead part while finding the closure.

- Conflicts of LR(1)
  - there are also two conflicts, SR & RR
  - SR Conflicts
    - A → α.aβ, b
    - β → б., a
  - RR Conflicts
    - A → .б, a
    - B → .б, a

S → CC

C → cC

C → d

- LR(1) grammar: the grammar for, which LR(1) is constructed is known as LR(1) or CLR(1).
- the grammar whose LR(1) parse is free from multiple entries or conflicts, then it is LR(1) grammar.
- Every SLR(1) grammar is CLR(1). but every CLR(1) grammar need not be SLR(1)
- CLR(1) is more powerful than SLR(1)
- the no of entries in SLR(1) parse table <= no of entries in CLR(1) parse table.

A → aA / a

E → T + E / T

T → id

S → AA

A → aA

A → b

A → (A) / a

S → AaAb / BbBa

A → ε

B → ε

S → Aa / bAc / dc / bda

A → d

S → Aa / bAc / Bc / bBa

A → d

B → d

S → A

A → AB / ε

B → aB / b

E → E + T / T

T → T * F / F

F → id

- In CLR(1) parser, we can find some of the state contains more than one production whose production part is same but follow part is different.
- because of this reason, the CLR(1) parse contains more no of entries & hence CLR(1) parser become most costly.

Q A canonical set of items is given below **(Gate - 2014) (2 Marks)**

S --> L. > R

Q --> R.

On input symbol < the set has

(A) a shift-reduce conflict and a reduce-reduce conflict.

(B) a shift-reduce conflict but not a reduce-reduce conflict.

(C) a reduce-reduce conflict but not a shift-reduce conflict.

(D) neither a shift-reduce nor a reduce-reduce conflict.

Answer: (D)

LALR(1)

- In CLR(1) parser, there can ne more than one state, having same production part and different follow part. Now combine those state whose production part is common and follow part is different, it is a single state and then construct the parse table, if the parse table is free from multiple entries, then the grammar is LALR(1).
- Conflict in LALR(1) parser
  - If there is no RR conflict in CLR(1), then there is no SR conflict in LALR(1) also.
  - If there is no RR conflicts in CLR(1), these may be RR conflict in LALR(1).

S → Aa / bAc /dc /bda

A → d

if in CLR(1), if there are no states having some production, but different follow part, the grammar is CLR(1) and LALR(1)

S → Aa

S → bAc

S → Bc

S → bBa

A → d

B → d

- every LALR(1) grammar is CLR(1) but every CLR(1) grammar need not br LALR(1)
- no of entries in LALR(1) parse table is <= no of entries in CLR(1) parse table
- every SLR(1) grammar is LALR(1) but converse need not be true.
- LALR(1) is powerful than SLR(1)

Q The grammar S → aSa | bS | c is **(Gate - 2010) (1 Marks)**

(A) LL(1) but not LR(1)

(B) LR(1)but not LR(1)

(C) Both LL(1)and LR(1)

(D) Neither LL(1)nor LR(1)

Answer: (C)

Q Consider the grammar shown below. **(Gate - 2003) (2 Marks)**

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)

Answer: (a)

Q 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? **(Gate-2015) (1 Marks)**

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

Answer: (D)

Q Consider the augmented grammar given below: **(Gate-2019) (2 Marks)**

S′ → S

S → (L) / id

L → L, S / S

Let I_0 = CLOSURE ({[S′ → S]}). The number of items in the set GOTO (I_0, ( ) is __________.

Ans: 5

Q Which of the following is the most powerful parring method? **(NET-DEC-2012)**

(A) LL(I)

(B) Canonical LR

(C) SLR

(D) LALR

Q Which of the following statements about the parser is/are correct? **(Gate - 2017) (1 Marks)**

I. Canonical LR is more powerful than SLR.

II. SLR is more powerful than LALR.

III. SLR is more powerful than canonical LR.

a) I only **b)** II only **c)** III only **d)** II and III only

Ans: a

Consider the following grammar G :

$S \rightarrow A \mid B$

$A \rightarrow a \mid c$

$B \rightarrow b \mid c$

where {S, A, B} is the set of non-terminals, {a, b, c} is the set of terminals.

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

S_1 : LR(1) can parse all strings that are generated using grammar G.

S_2 : LL(1) can parse all strings that are generated using grammar G.

Choose the correct answer from the code given below : (**NET-DEC-2018)**

a) Only S1 **b)** Only S2 **c)** Both S1 and S2 **d)** Neither S1 Nor S2

Q Which of the following is true? **(NET-DEC-2014)**

(A) Canonical LR 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

Ans: c

Q A shift reduce parser suffers from **(NET-JUNE-2014)**

(A) shift reduce conflict only

(B) reduce reduce conflict only

(C) both shift reduce conflict and reduce reduce conflict

(D) shift handle and reduce handle conflicts

Ans: c

Q Which of the following statements is false?**(Gate-2001) (2 Marks)**

(A) An unambiguous grammar has same leftmost and rightmost derivation

(B) An LL(1) parser is a top-down parser

(C) LALR is more powerful than SLR

(D) An ambiguous grammar can never be LR(k) for any k

Answer: (A)

Q 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? **(Gate - 2015) (2 Marks)**

(A) SLR, LALR

(B) Canonical LR, LALR

(C) SLR, canonical LR

(D) LALR, canonical LR

Answer: (C)

Q Consider the grammar **(Gate - 2005) (2 Marks)**

S → (S) | a

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

Answer: (B)

Q 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 **(Gate - 2003) (2 Marks)**

(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

Answer: (B)

Q Consider the following two sets of LR(1) items of an LR(1) grammar. **(Gate - 2013) (2 Marks)**

X -> c.X, c/d

X -> .cX, c/d

X -> .d, c/d

X -> c.X, $

X -> .cX, $

X -> .d, $

Which of the following statements related to merging of the two sets in the corresponding LALR parser is/are FALSE?

1. Cannot be merged since look aheads are different.

2. Can be merged but will result in S-R conflict.

3. Can be merged but will result in R-R conflict.

4. Cannot be merged since goto on c will lead to two different sets.

(A) 1 only **(B)** 2 only **(C)** 1 and 4 only **(D)** 1, 2, 3, and 4

Answer: (D)

Q An LALR(1) parser for a grammar G can have shift-reduce (S-R) conflicts if and only if **(Gate - 2008) (2 Marks)**

(A) the SLR(1) parser for G has S-R conflicts

(B) the LR(1) parser for G has S-R conflicts

(C) the LR(0) parser for G has S-R conflicts

(D) the LALR(1) parser for G has reduce-reduce conflicts

Answer: (B)

Q Which of the following describes a handle (as applicable to LR-parsing) appropriately? **(GATE - 2008) (1 Marks)**

(A) It is the position in a sentential form where the next shift or reduce operation will occur

(B) It is non-terminal whose production will be used for reduction in the next step

(C) It is a production that may be used for reduction in a future step along with a position in the sentential form where the next shift or reduce operation will occur

(D) It is the production p that will be used for reduction in the next step along with a position in the sentential form where the right hand side of the production may be found

Answer: (D)

Q Given the following statements: **(NET-DEC-2013)**

S1: SLR uses follow information to guide reductions. In case of LR and LALR parsers, the look-ahead are associated with the items and they make use of the left context available to the parser.

S2 : LR grammar is a larger sub- class of context free grammar as compared to that SLR and LALR grammars.
Which of the following is true ?

(A) S1 is not correct and S2 is not correct. **(B)** S1 is not correct and S2 is correct.
(C) S1 is correct and S2 is not correct. **(D)** S1 is correct and S2 is correct.

Ans: d

Q What is the maximum number of reduce moves that can be taken by a bottom-up parser for a grammar with no epsilon- and unit-production (i.e., of type A -> є and A -> a) to parse a string with n tokens? **(Gate-2013) (1 Marks)**

(A) n/2 **(B)** n-1 **(C)** 2n-1 **(D)** 2n

Answer: (B)

Q Given a grammar: S1 → Sc, S → SA | A, A → aSb | ab, there is a rightmost derivation S1 ⇒ Sc ⇒ SAC ⇒ SaSbc Thus, SaSbc is a right sentential form, and its handle is **(NET-JUNE-2013)**

(A) SaS **(B)** bc **(C)** Sbc **(D)** aSb

Ans: D

Q Shift-Reduce parsers perform the following: **(NET-DEC-2014)**

(A) Shift step that advances in the input stream by K(K > 1) symbols and Reduce step that applies a completed grammar rule to some recent parse trees, joining them together as one tree with a new root symbol.
(B) Shift step that advances in the input stream by one symbol and Reduce step that applies a completed grammar rule to some recent parse trees, joining them together as one tree with a new root symbol.
(C) Shift step that advances in the input stream by K(K = 2) symbols and Reduce step that applies a completed grammar rule to form a single tree.
(D) Shift step that does not advance in the input stream and Reduce step that applies a completed grammar rule to form a single tree.

Ans: b

Q Which of the following is FALSE? **(NET-JULY-2016)**

a) The grammar S → aSb | bSa | SS | ∈, where S is the only non-terminal symbol and ∈ is the null string, is ambiguous.
b) SLR is powerful than LALR.

c) An LL(1) parser is a top-down parser.
d) YACC tool is an LALR(1) parser generator

Q Which one from the following is false? **(NET-JUNE-2015)**
a) LALR parser is Bottom - Up parser
b) A parsing algorithm which performs a left to right scanning and a right most deviation is RL (1)
c) LR parser is Bottom - Up parser.
d) In LL(1), the 1 indicates that there is a one - symbol look - ahead.
Ans: b

Q Which of the following statements is false? **(NET-DEC-2015)**
a) Top-down parsers are LL parsers where first L stands for left - to - right scan and second L stands for a leftmost derivation.
b) (000)* is a regular expression that matches only strings containing an odd number of zeroes, including the empty string.
c) Bottom-up parsers are in the LR family, where L stands for left - to - right scan and R stands for rightmost derivation.
d) The class of context - free languages is closed under reversal. That is, if L is any context - free language, then the language LR = $\{w^R : w \in L\}$ is context - free.
Ans: b

Q Which one of the following is a top-down parser? **(Gate - 2007) (2 Marks)**
(A) Recursive descent parser.
(B) Operator precedence parser.
(C) An LR(k) parser.
(D) An LALR(k) parser
Answer: (A)

Operator precedence grammar

- Operator precedence parser can be constructed for both ambiguous and unambiguous grammar.
- In general operator precedence grammar have less complexity
- Every CFG is not operator precedence grammar
- Generally used for languages which are useful in scientific application.
- Operator Grammar
    - The grammar that does not contain ε production & adjacent non-terminals on RHS of any rule is called operator Grammar.

S → AB
A → a
B → b

S → AaB
A → a / ε
B → b

S → AaB
B → aA / b
B → a

S → AOB / int
O → + / * / -

Operator precedence parsing algorithm
let a be the TOS & b will be the lookahead symbol than
if a , b or a = = b, then shift b & increment i/p pointer
if a>b, than pop once and repeat pop operation until current TOS is < previous TOS
if a == b == $ then successful completion.

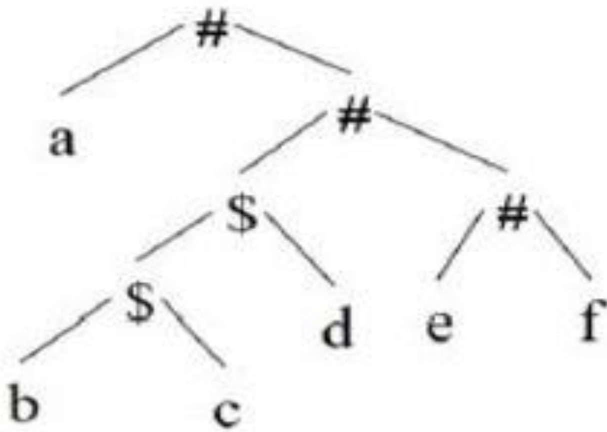
Q Which of the following grammar rules violate the requirements of an operator grammar? P, Q, R are nonterminal, and r, s, t are terminals. **(Gate-2004) (2 Marks)**

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

Answer: (B)

Q Consider the following parse tree for the expression a#bcd#e#f, involving two binary operators $ and # **(Gate - 2018) (2 Marks)**



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

(ANSWER- A)

Q Consider the grammar defined by the following production rules, with two operators ∗ and +

S --> T * P

T --> U | T * U

P --> Q + P | Q

Q --> Id

U --> Id

Which one of the following is TRUE? **(Gate-2014) (2 Marks)**

(A) + is left associative, while ∗ is right associative

(B) + is right associative, while ∗ is left associative

(C) Both + and ∗ are right associative

(D) Both + and ∗ are left associative

Answer: (B)

Q The attributes of three arithmetic operators in some programming language are given below. **(Gate-2016) (1 Marks)**

Operator	Precedence	Associativity	Arity
+	High	Left	Binary
–	Medium	Right	Binary
*	Low	Left	Binary

The value of the expression 2 – 5 + 1 – 7 * 3 in this language is __________.

ANSWER 9

Q Given the following expression grammar **(Gate-2000) (2 Marks)**

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 *

Answer: (B)

Q Given the following expressions of a grammar **(NET-DEC-2012)**

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 *

Ans: b

Q Consider two binary operators '↑' and '↓' with the precedence of operator ↓ being lower than that of the operator ↑. Operator ↑ is right associative while operator ↓, is left associative. Which one of the following represents the parse tree for expression (7↓3↑4↑3↓2)? **(Gate-2011) (2 Marks)**

a	b	c	d

ANSWER B

Q The grammar A → AA | (A) | e is not suitable for predictive-parsing because the grammar is **(Gate-2005) (2 Marks)**

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

Answer: (a)

Q Consider the following statements related to compiler construction:

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

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

Which of the above statement(s) is/are correct? **(NET-NOV-2017)**

A) Only I
b) Only II
c) Both I and II
d) Neither I nor II

Ans: d

Q Which one of the following statements is FALSE? **(Gate - 2018) (1 Marks)**

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.
(ANSWER-B)

Q A student wrote two context-free grammars G1 and G2 for generating a single C-like array declaration. The dimension of the array is at least one. For example,
int a[10][3];
The grammars use D as the start symbol, and use six terminal symbols int ; id [ ] num.

Grammar G1
D → int L;
L → id [E
E → num]
E → num] [E

Grammar G2
D → int L;
L → id E
E → E[num]
E → [num]

Which of the grammars correctly generate the declaration mentioned above? **(Gate - 2016) (1 Marks)**
a) Both G1 and G2 **b)** Only G1 **c)** Only G2 **d)** Neither G1 nor G2

Semantic Analysis

- Grammar + Semantic Rule + Semantic Actions = Syntax Directed Translation
- With grammar we give meaningful rules, and apart from semantic analysis SDT can also be used to perform things like
    - Code generation
    - Intermediate code generation
    - Value in the symbol table
    - Expression evaluation
    - Converting infix to post fix
    - Store information in the symbol table
    - To construct symbol table
    - To check variable declaration.
    - Equivalence between formal and actual parameter.
- Things can be done in parallel to parsing…so with semantic action and rule parsers become much powerful

Q Consider the grammar with the following translation rules and E as the start symbol.
E → E_1 # T { E.value = E1.value * T.value }
E → T{ E.value = T.value }
T → T_1 & F { T.value = T1.value + F.value }
T → 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. **(Gate-2004) (2 Marks)**
(A) 200 **(B)** 180 **(C)** 160 **(D)** 40
Answer: (C)

Q Consider the grammar with the following translation rules and E as the start symbol.
E → E_1 + T {print ('+');}
E → T
T → T_1 * F {print ('*');}
T → F
F → num {print ('num.val');}
Construct the parse tree for the string 2 + 3 * 4, and find what will be printed.

Q Consider the translation scheme shown below **(Gate-2003) (2 Marks)**
S → T R
R → + T {print ('+');} R | ε
T → num {print (num.val);}
Here num is a token that represents an integer and num.val represents the corresponding integer value. For an input string '9 + 5 + 2', this translation scheme will print
(A) 9 + 5 + 2 **(B)** 9 5 + 2 + **(C)** 9 5 2 + + **(D)** + + 9 5 2
Answer: (B)

Q Consider the following translation scheme.**(Gate-2006) (2 Marks)**
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+*
Answer: (D)

Q Consider the following Syntax Directed Translation Scheme (SDTS), with non-terminals {S, A} and terminals {a, b}}.

S ⟶ **a**A { print 1 }
S ⟶ **a** { print 2 }
A ⟶ S**b** { print 3 }

Using the above SDTS, the output printed by a bottom-up parser, for the input 'aab' is **(GATE-2016) (2 Marks)**

a) 1 3 2 **b)** 2 2 3 **c)** 2 3 1 **d)** Syntax Error

ANSWER C

Q Consider the following Syntax Directed Translation Scheme (SDTS), with non-terminals {S, W} and terminals {x, y, z}}. Using the above SDTS, the output printed by a bottom-up parser, for the input 'xxxxyzz' is

Q Consider the grammar with the following translation rules and E as the start symbol.
E → E_1 * T { E.value = E1.value * T.value }
E → T{ E.value = T.value }
T → F - T { T.value = F.value - T1.value }
T → F{ T.value = F.value }
F → num { F.value = num.value }
Compute E.value for the root of the parse tree for the expression: 4 – 2 – 4 * 2.

Q Consider the grammar with the following translation rules and E as the start symbol.
E → E_1 + T { E.nptr = mknode(E1.nptr, +, T.ptr);}
E → T{ E.nptr = T.nptr }
T → T1 * F { T.nptr = mknode(T1.nptr, *, F.ptr);}
T → F{ T.nptr = F.nptr }
F → id { F.nptr = mknode(null, id name, null);}
Construct the parse tree for the expression: 2+3*4

Q Consider the syntax directed definition shown below. **(Gate-2003) (2 Marks)**
S → id : = E {gen (id.place = E.place;);}
E → E1 + E2 {t = newtemp ( ); gen (t = El.place + E2.place;); E.place = t}
E → id {E.place = id.place;}
Here, gen is a function that generates the output code, and newtemp is a function that returns the name of a new temporary variable on every call. Assume that ti's are the temporary variable names generated by newtemp.
For the statement 'X: = Y + Z', the 3-address code sequence generated by this definition is

(A) X = Y + Z
(B) t1 = Y + Z; X t1
(C) t1 =Y; t2 = t1 + Z; X = t2
(D) t1 = Y; t2 = Z; t3 = t1 + t2; X = t3

Answer: (C)

Q Consider the syntax directed definition shown below. **(Gate-2003) (2 Marks)**

N → L {N.dval = L.dval}

L → L1 B {L.dval = L1.dval*2 + B.dval}

L → B {L.dval = B.dval}

B → 0 {B.dval = 0}

B → 1 {B.dval = 1}

Q Consider the syntax directed definition shown below. **(Gate-2003) (2 Marks)**

S → id = E {gen(id.name = E.place)}

E → E1 + T {E.place = newtemp(); gen(E.place = E1.place + T.place);}

E → T {E.place = T.place;}

T → T1 * F {T.place = newtemp(); gen(T.place = T1.place * F.place);}

T → F {T.place = F.place;}

F → id {F.place = id.name;}

Classification of Attributes

Based on the process of Evaluation of the values, attributes are classified into two types:

- Synthesised Attributes
- Inherited Attributes

- **Synthesised Attributes**: The attributes Whose value is Evaluated in terms of attribute value of its children is known as Synthesised attributes.
  - A → XYZ {A.S = f(X.S / Y.S / Z.S)}
- **Inherited Attributes**: The attribute whose values are evaluated in terms of attribute value of parents & Left siblings is known as inherited attributes.
  - Inherited attributes are convenient for expressing dependence of a programming language construct on the context in which it appears.
  - T → int {T.type = integer}
  - T → double {T.type = double}

S-Attributed	L-Attributes SDT
Uses only Synthesized attributes	Uses both inherited and synthesised attributes. Each inherited attribute is restricted to inherit either form parent or left sibling only.
Semantic actions are placed at extreme right on right end of production	Semantic actions are placed anywhere on right hand side of the production.
Attributes are evaluated during BUP	Attributes are evaluated by traversing parse tree depth first left to right.

S-Attributed SDT

Q Which of the following statements are TRUE? **(Gate-2009) (1 Marks)**

I. There exist parsing algorithms for some programming languages whose complexities are less than $O(n^3)$.

II. A programming language which allows recursion can be implemented with static storage allocation.

III. No L-attributed definition can be evaluated in The framework of bottom-up parsing.

IV. Code improving transformations can be performed at both source language and intermediate code level.

(A) I and II **(B)** I and IV **(C)** III and IV **(D)** I, III and IV

Answer: (B)

Q In a bottom-up evaluation of a syntax directed definition, inherited attributes can **(Gate-2003) (2 Marks)**
(A) always be evaluated
(B) be evaluated only if the definition is L-attributed
(C) be evaluated only if the definition has synthesized attributes
(D) never be evaluated
Answer: (B)

Q Consider the following grammar and the semantic actions to support the inherited type declaration attributes. Let X1, X2, X3, X4, X5 and X6 be the placeholders for the non-terminals D, T, L or L1 in the following table:

Production rule	Semantic action
D → TL	X1. type = X2. type
T → int	T. type = int
T → float	T.type = float
L → L1, id	X3.type = X4.type addType(id.entry, X5type)
L → id	addType(id.entry, X6type)

Which one of the following are the appropriate choices for X1, X2, X3 and X4? **(Gate-2019) (2 Marks)**
a) X1 = L, X2 = T, X3 = L1, X4 = L
b) X1 = L, X2 = L, X3 = L1, X4 = T
b) X1 = T, X2 = L, X3 = L1, X4 = T
d) X1 = T, X2 = L, X3 = T, X4 = L1
Ans: a

Q Match the following according to input (from the left column) to the compiler phase (in the right column) that process it: **(Gate-2017) (2 Marks)**

(P) Syntax tree	(i) Code generator
(Q) Character stream	(ii) Syntax analyzer
(R) Intermediate representation	(iii) Semantic analyzer
(S) Token stream	(iv) Lexical analyzer

a) P -> (ii), Q -> (iii), R -> (iv), S -> (i)
b) P -> (ii), Q -> (i), R -> (iii), S -> (iv)
c) P -> (iii), Q -> (iv), R -> (i), S -> (ii)
d) P -> (i), Q -> (iv), R -> (ii), S -> (iii)

Ans: c

Q Match the following: **(NET-JUNE-2014)**

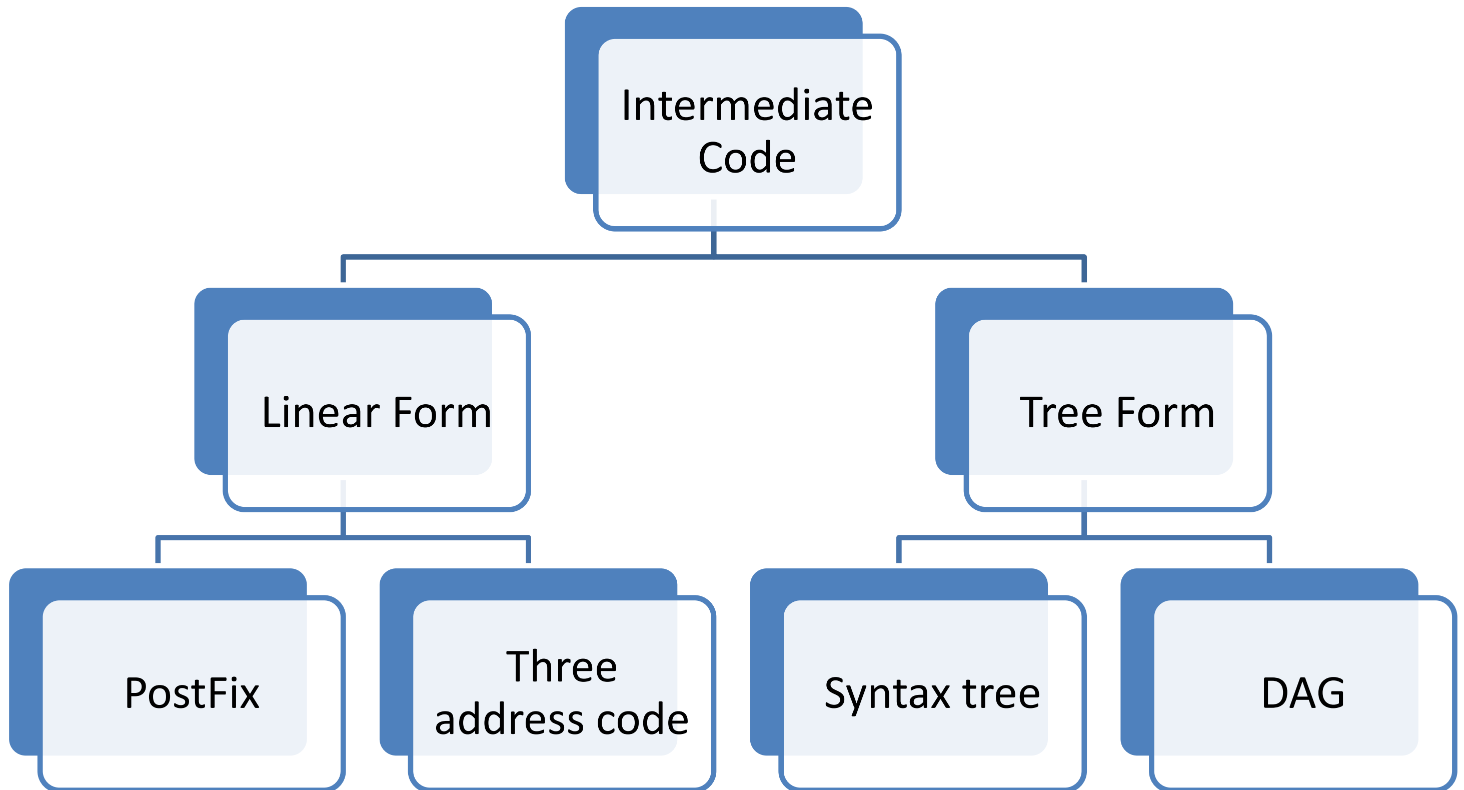
List – I	**List – II**
a. Chomsky Normal form	**i.** S → b S S | a S | c
b. Greibach Normal form	**ii.** S → a S b | ab
c. S-grammar	**iii.** S → AS | a, A → SA | b
d. LL grammar	**iv.** S → a B S, B B → b

Codes:

	a	b	c	d
a)	iv	iii	i	ii
b)	iv	iii	ii	i
c)	iii	iv	i	ii
d)	iii	iv	ii	i

Ans: c

Intermediate Code Generation

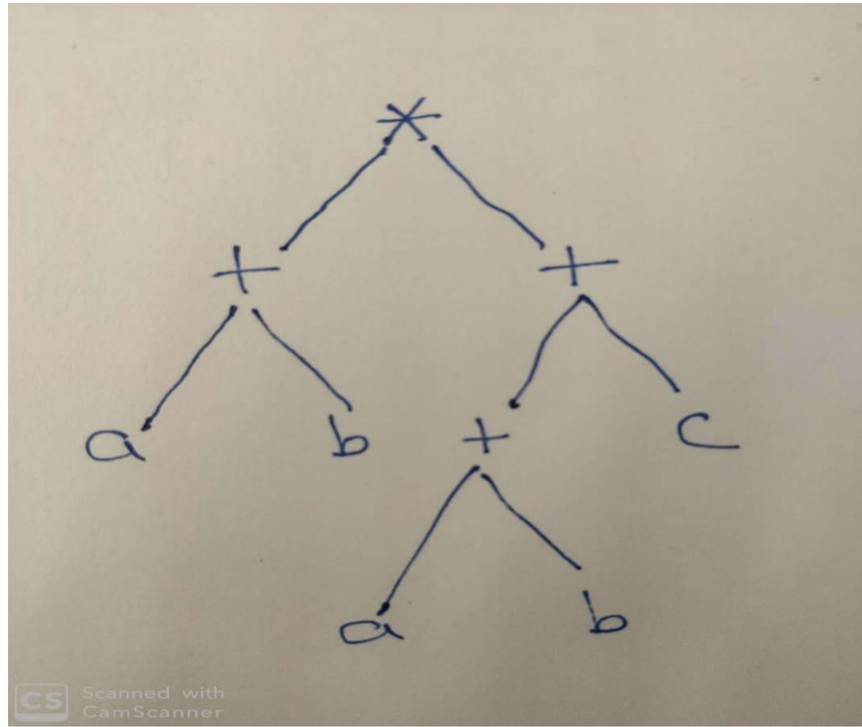


(a+b) * (a + b + c)

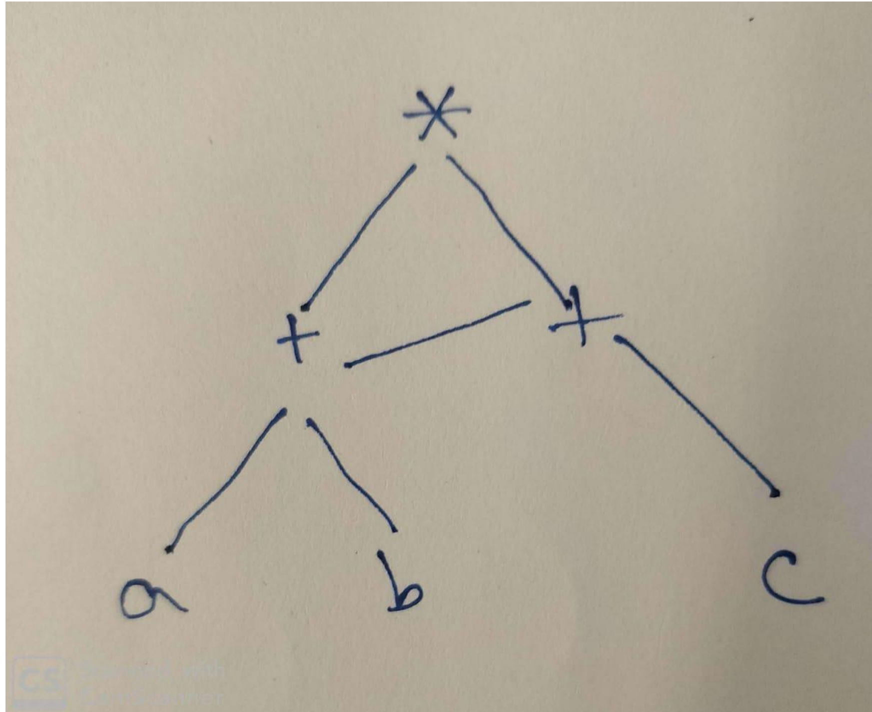
- **Post fix**
  - ab+ab+c+*

- **Three address code**
  - t_1 = a+b
  - t_2 = a+b
  - t_3 = t_2 + c
  - t_4 = t_1 * t3

- **Syntax Tree**



- **Direct Acyclic Graph**



3 Address Code

Types of 3 address codes

1) x = y operator z
2) x = operator z
3) x = y
4) goto L
5) A[i] = x
   y = A[i]
6) x = *p
   y = &x

- 3 address codes can be implemented in a number of ways

(a + b) * (c + d) + (a + b + c)

1) t_1 = a+b
2) t_2 = -t_1
3) t_3 = c+d
4) t_4 = t_2 * t_3
5) t_5 = a+b
6) t_6 = t_5 + c
7) t_7 = t_4 + t_6

Quadruples

	Operator	Operand1	Operand2	Result
1)	+	a	b	t_1
2)	-	t_1		t_2
3)	+	c	d	t_3
4)	*	t_2	t_3	t_4
5)	+	a	b	t_5
6)	+	t_5	c	t_6
7)	+	t_4	t_6	t_7

- **Advantage**
  - statement can be moved around
- **Disadvantage**
  - too much of space is wasted

Triplet

	Operator	Operand1	Operand2
1)	+	a	b
2)	-	1	
3)	+	c	d
4)	*	2	3
5)	+	a	b
6)	+	5	c
7)	+	4	6

- **Advantage**
  - Space is not wasted
- **Disadvantage**
  - Statement cannot be moved

Indirect triple

triple can be separated by order of execution and uses the pointers concepts

- **Advantage**
  - Statement can be moved

- **Disadvantage**
  - two memory access

```
if(a, b) then t=1
else t = 0

i)      if (a<b) goto (i+3)
i+1)    t=0
i+2)    goto (i+4)
i+3)    t=1
i+4)    exit
```

```
while(C) do S

i)      if (E) goto i+2
i+1)    goto i+4
i+2)    S
i+3)    goto i
i+4)    exit
```

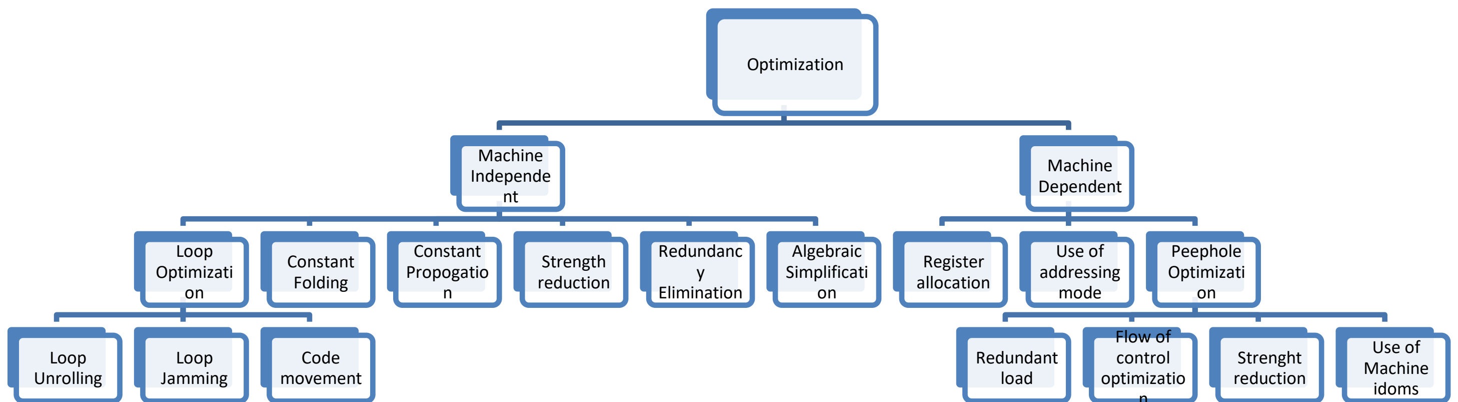
```
for(1=0; i<10 ; i++)
        s

i)      i = 0
i+1)    if(i<10) goto i+3
i+2)    goto i+6
i+3)    S
i+4)    i = i + 1
i+5)    goto i+1
i+6)    exit
```

Q One of the purposes of using intermediate code in compilers is to **(Gate-2014) (1 Marks)**
(a) make parsing and semantic analysis simpler
(b) improve error recovery and error reporting.
(c) increase the chances of reusing the machine-independent code optimizer in other compilers
(d) improve the register allocation.
ANSWER -C

Optimization

- Process of reducing the execution time of a code without effecting the outcome of the source program, is called as optimization.



Constant Folding: Replacing the value of expression before compilation is called as constant folding

x = a + b + 2 * 3 + 4

x = a + b + 10

Constant Propagation: replacing the value of constant before compile time, is called as constant propagation.

pi = 3.1415

x = 360 / pi

x = 360/3.1415

Strength reduction: replacing the costly operator by cheaper operator, this process is called strength reduction.

y = 2 * x

y = x + x

Redundant code Elimination: avoiding the evaluation of any expression more than once is redundant code elimination.

x = a + b

y = b + a

x = a + b

y = x

Algebraic Simplification: Basic laws of math's which can be solved directly.

a = b*1

a=b

a=b+0

a=b

- Loop Optimization
    - To apply loop optimization, we must first detect loops.
    - For detecting loops, we use control flow analysis (CFA) using program flow graph (PFG)
    - To find PFG, we need to find basic blocks.
    - A Basic block is a sequence of 3-adress statements where control enters at the beginning and leaves only at the end without any jumps or halts

- The block can be identified with the help of leader
    - Finding the leader
    - Finding the bocks
    - Construct PFG

- In order to find the basic blocks, we need to finds the leader in the program then a basic block will start from one leader to the next leader but not including next leader.
- identifying leaders in a basic block
    - First statement is a leader
    - Statement that is the target of conditional or unconditional statement is a leader
    - Statement that follow immediately a conditional or unconditional statement is a leader

```
Fact(x)
{
        int f=1
        for(i=2 ; i<=x ; i++)
                f = f*i;
        return f;
}
```

1) f=1;
2) i=2
3) if(i>x), goto 9
4) t1=f*i;
5) f=t1;
6) t2=i+1;
7) i=t2;
8) goto(3)
9) goto calling program

1) f=1;
2) i=2

3) if(i>x), goto 9

4) t1=f*i;
5) f=t1;
6) t2=i+1;
7) i=t2;
8) goto(3)

9) goto calling program

Loop Jamming: combining the bodies of two loops, whenever they share the same index and same no o variables

```
for (int i=0; i<=10; i++)
        for (int j=0; j<=10; j++)
                x[i, j]="TOC"
for (int j=0; j<=10; j++)
                y[i]="CD"
```

```
for (int i=0; i<=10; i++)
{
        for (int j=0; j<=10; j++)
        {
                x[i, j]="TOC"
        }
        y[i]="CD"
}
```

Loop Unrolling: getting the same output with less no of iteration is called loop unrolling

```
int i=1;
while(i<=100)
{
        print(i)
        i++
}
```

```
int i=1;
while(i<=100)
{
        print(i)
        i++
        print(i)
        i++
}
```

Code movement: removing those code out from the loop which is not related to loop.

```
int i=1;
while(i<=100)
{
        a =b+c
        print(i)
        i++
}
```

Q In compiler design 'reducing the strength' refers to **(NET-DEC-2012)**

(A) reducing the range of values of input variables.

(B) code optimization using cheaper machine instructions.

(C) reducing efficiency of program.

(D) None of the above

Ans: B

Q Loop unrolling is a code optimization technique: **(NET-DEC-2015)**

a) that avoids tests at every iteration of the loop.

b) that improves performance by decreasing the number of instructions in a basic block.

c) that exchanges inner loops with outer loops

d) that reorders operations to allow multiple computations to happen in parallel

Ans: a

Q In compiler optimization, operator strength reduction uses mathematical identities to replace slow math operations with faster operations. Which of the following code replacements is an illustration of operator strength reduction? **(NET-AUG-2016)**

a) Replace P + P by 2 * P or Replace 3 + 4 by 7.

b) Replace P * 32 by P < < 5

c) Replace P * 0 by 0

d) Replace (P < <4) – P by P * 15

Q Consider the following intermediate program in three address code

p = a - b

q = p * c

p = u * v

q = p + q

Which one of the following corresponds to a static single assignment from the above code **(Gate - 2017) (2 Marks)**?

A)	B)	C)	D)
p1 = a - b	p3 = a - b	p 1 = a - b	p1 = a - b

q 1 = p1 * c p1 = u * v q1 = p1 + q1	q4 = p3 * c p4 = u * v q5 = p4 + q4	q1 = p2 * c p3 = u * v q2 = p4 + q3	q1 = p * c p2 = u * v q2 = p + q

ANSWER B

Q Consider the following code segment.

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

The minimum number of variables required to convert the above code segment to static single assignment form is ________. **(Gate - 2017) (2 Marks)**

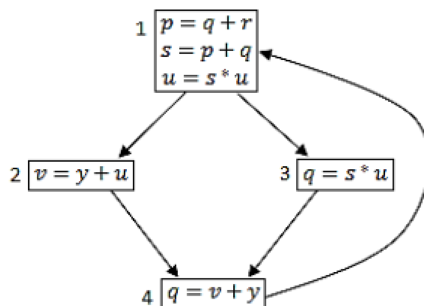
ANSWER 10

Q The least number of temporary variables required to create a three-address code in static single assignment form for the expression *q + r/3 + s – t * 5 + u * v/w* is __________. (Gate - 2015) (1 Marks)

ANSWER 8

A variable x is said to be live at a statement S_i in a program if the following three conditions hold simultaneously:

i. There exists a statement S_j that uses x
ii. There is a path from S_i to S_j in the flow graph corresponding to the program
iii. The path has no intervening assignment to x including at S_i and S_j



The variables which are live both at the statement in basic block 2 and at the statement in basic block 3 of the above control flow graph are

(a) p, s, u **(b) r, s, u** **(c) r, u** **(d) q, v**

Answer 3

Q Consider the intermediate code given below:

1. i = 1
2. j = 1
3. t1 = 5 * i
4. t2 = t1 + j
5. t3 = 4 * t2
6. t4 = t3
7. a[t4] = –1
8. j = j + 1
9. if j <= 5 goto(3)
10. i = i + 1
11. if i < 5 goto(2)

The number of nodes and edges in the control-flow-graph constructed for the above code, respectively, are **(Gate - 2015) (2 Marks)**

a) 5 and 7 **b)** 6 and 7 **c)** 5 and 5 **d)** 7 and 8

ANSWER B

Q Which one of the following is FALSE? **(Gate - 2014) (1 Marks)**

(1) A basic block is a sequence of instructions where control enters the sequence at the beginning and exits at the end.

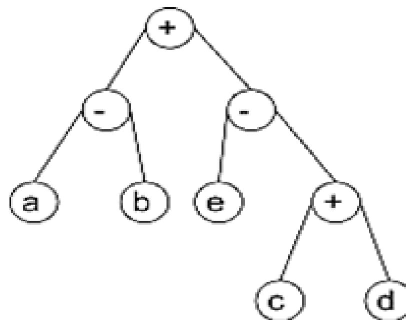
(2) Available expression analysis can be used for common subexpression elimination

(3) Live variable analysis can be used for dead code elimination

(4) x=4*5⇒x=20 is an example of common subexpression elimination

ANSWER D

Consider evaluating the following expression tree on a machine with load-store architecture in which memory can be accessed only through load and store instructions. The variables a, b, c, d and e are initially stored in memory. The binary operators used in this expression tree can be evaluated by the machine only when the operands are in registers. The instructions produce result only in a register. If no intermediate results can be stored in memory, what is the minimum number of registers needed to evaluate this expression?



(GATE - 2011) (2 Marks)

ANSWER 3

The program below uses six temporary variables a, b, c, d, e, f.

```
a = 1
b = 10
c = 20
d = a+b
e = c+d
f = c+e
b = c+e
e = b+f
d = 5+e
return d+f
```

Assuming that all operations take their operands from registers, what is the minimum number of registers needed to execute this program without spilling?

(GATE - 2010) (2 Marks)

ANSWER 3

Q Some code optimizations are carried out on the intermediate code because **(GATE - 2008) (1 Marks)**

a) They enhance the portability of the compiler to other target processors

b) Program analysis is more accurate on intermediate code than on machine code

c) The information from dataflow analysis cannot otherwise be used for optimization

d) The information from the front end cannot otherwise be used for optimization

ANSWER a

Q A simplified computer the instructions are **(GATE-2007) (2 Marks)**

OP R_j, R_i	- Performs R_j OP R_i and stores the result in register R_i.
OP m, R_i	- Performs *val* OP R_i and stores the result in R_i. *val* denotes the content of memory location *m*.
MOV m, R_i	- Moves the content of memory location *m* to register R_i.
MOV R_i, m	- Moves the content of register R_i to memory location *m*.

The computer has only to registers, and OP is either ADD or SUB. Consider the following basic block:

$$t_1 = a + b$$
$$t_2 = c + d$$
$$t_3 = e - t_2$$
$$t_4 = t_1 - t_3$$

Assume that all operands are initially in memory. The final value of the computation should be in memory. What is the minimum number of MOV instructions in the code generated for this basic block?

(A) 2 **(B)** 3 **(C)** 5 **(D)** 6

Answer: (B)

Q Consider the following C code segment. **(Gate-2006) (1 Marks)**

```
for (i = 0, i<n; i++)
{
   for (j=0; j<n; j++)
   {
      if (i%2)
      {
         x += (4*j + 5*i);
         y += (7 + 4*j);
      }
   }
}
```

Which one of the following is false?

(A) The code contains loop invariant computation

(B) There is scope of common sub-expression elimination in this code

(C) There is scope of strength reduction in this code

(D) There is scope of dead code elimination in this code

Answer: (B)

Q Consider the grammar rule E → E1 – E2 for arithmetic expressions. The code generated is targeted to a CPU having a single user register. The subtraction operation requires the first operand to be in the register. If E1 and E2 do not have any common sub expression, in order to get the shortest possible code **(Gate-2014) (2 Marks)**

(A) E1 should be evaluated first
(B) E2 should be evaluated first
(C) Evaluation of E1 and E2 should necessarily be interleaved
(D) Order of evaluation of E1 and E2 is of no consequence
Answer: (B)

Q 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 **(Gate-2016) (2 Marks)**

Note: This question was asked as Numerical Answer Type.

(A) 6 **(B)** 8 **(C)** 9 **(D)** 10

Answer: (D)

Q Match all items in Group 1 with correct options from those given in Group 2. **(Gate - 2009) (2 Marks)**

Group 1	Group 2
P. Regular expression	1. Syntax analysis
Q. Pushdown automata	2. Code generation
R. Dataflow analysis	3. Lexical analysis
S. Register allocation	4. Code optimization

(A) P-4. Q-1, R-2, S-3
(B) P-3, Q-1, R-4, S-2
(C) P-3, Q-4, R-1, S-2
(D) P-2, Q-1, R-4, S-3

Answer: (B)