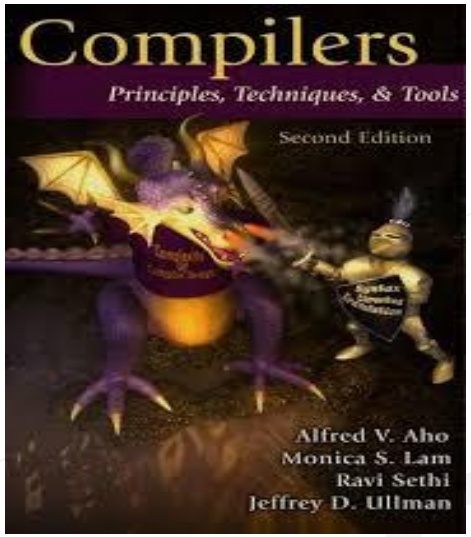
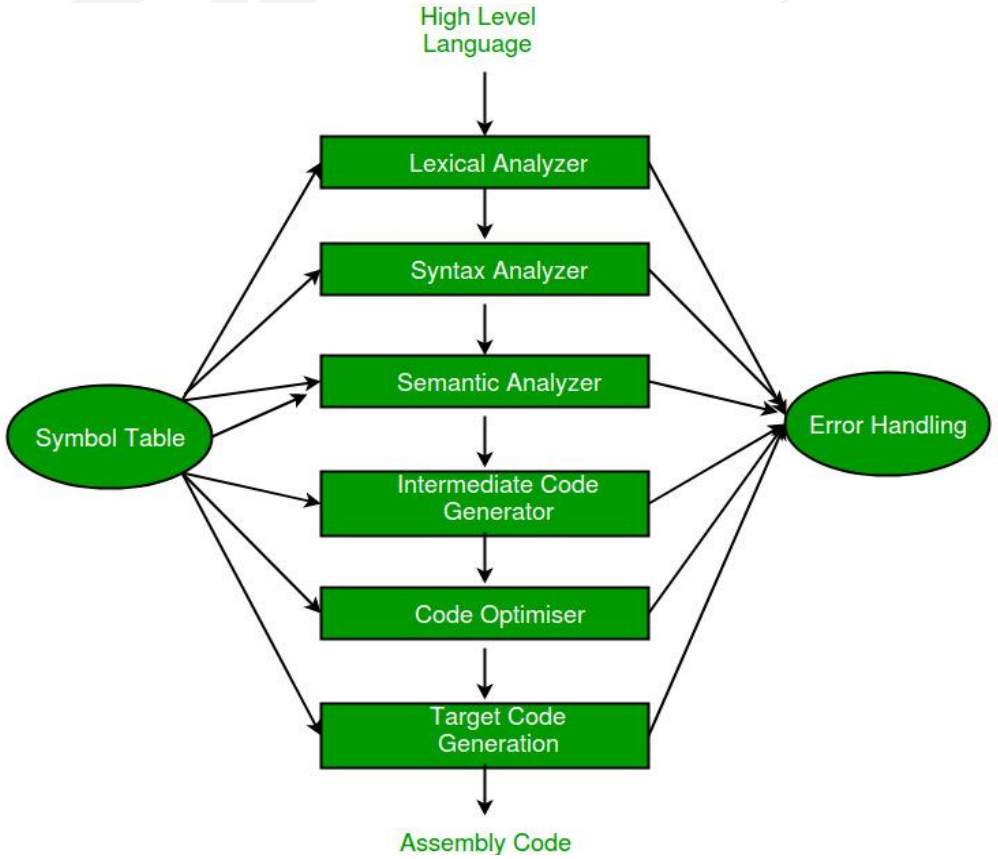
Compiler Design

- Compiler Design is a core subject in university exams and frequently appears in standardized tests such as GATE (4–5 marks) and NET (8–10 marks). Its syllabus is relatively small and heavily math-oriented, comprising about 25% theory and 75% numerical problems. Although it's not commonly emphasized in industry interviews, it remains an important academic subject. A background in Theory of Computation, as well as some familiarity with Operating Systems and Computer Architecture, can be helpful. While time-consuming, the concise syllabus and its focus on applied mathematics make it manageable.





- **Title:** Compilers: Principles, Techniques, and Tools
- **Authors:** Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman
- **Publisher:** Pearson
- **Edition:** 2nd Edition

Syllabus

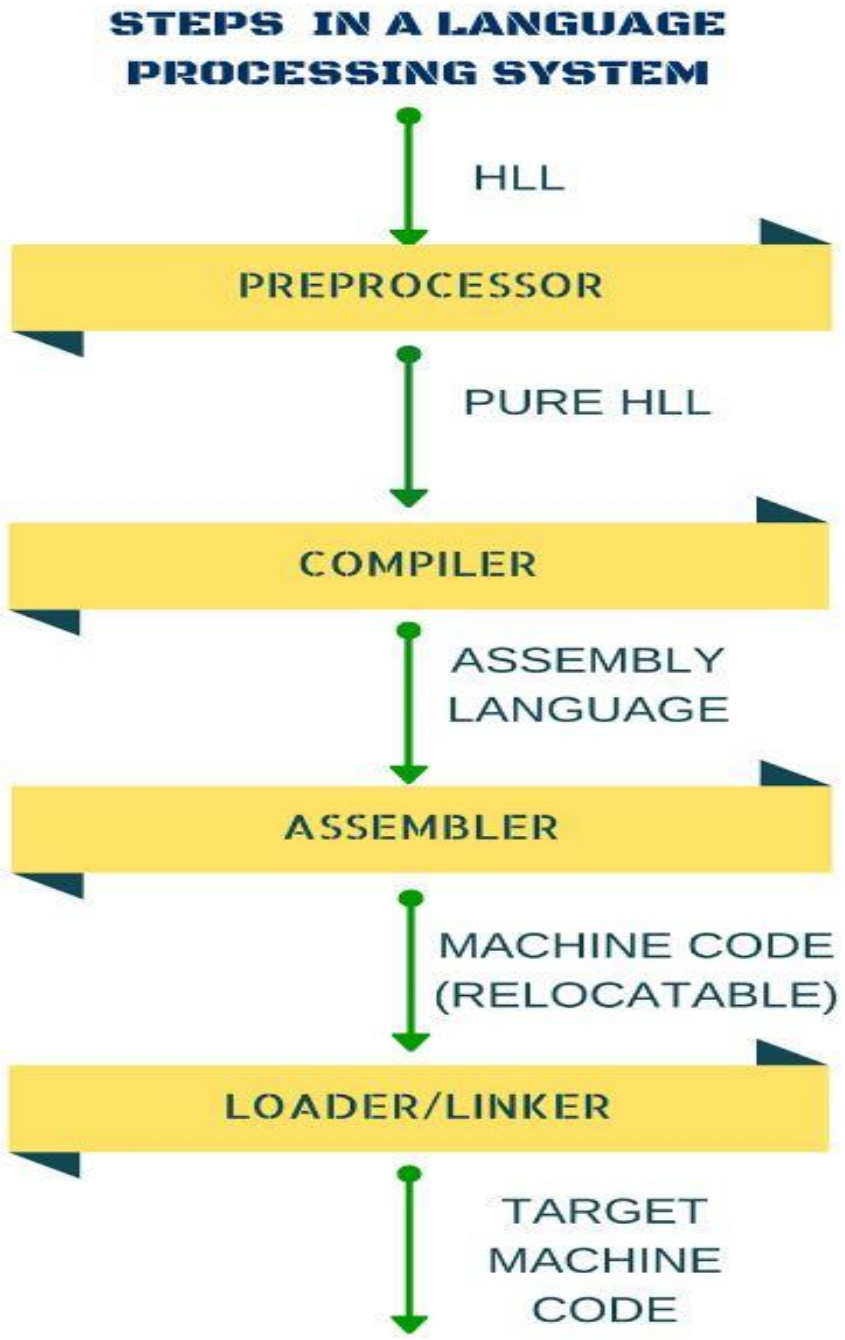
- Lexical analysis
- Parsing
- Syntax-directed translation
- Runtime environments
- Intermediate code generation
- Local optimisation, Data flow analyses: constant propagation, liveness analysis, common subexpression elimination.

Subject blue print

- Basics of compiler (0 marks)
  - Language Process System, pre-processor, compiler, assembler, loader/linker
  - Phases and pass in a compiler
  - Compiler vs interpreter
  - Symbol table and error handler
  - Types of error
- Lexical Analysis (1-2 marks question, most of the time theoretical question)
  - Basics, Lexemes, Tokens
  - Functions of lexical analyser
  - Lexical analyser implementation
- Parsing Techniques (2-3, marks question mostly on numerical one theoretical)
  - Basics of parsing
  - CFG grammar, ambiguity, derivation, determinism
  - Top down parsing, LL (1)
  - Bottom up parsing, LR (0), SLR (1), CLR (1) and LALR (1)
  - OPP
- Syntax Directed Translation (1-2 basic question)
  - Basics
  - Types of attributes
  - DAG
  - Types of SDT, S/L attributes
- Code Generation and Optimization (1-2 marks theory or numerical)
  - Basics
  - 3 address code
  - Code optimization

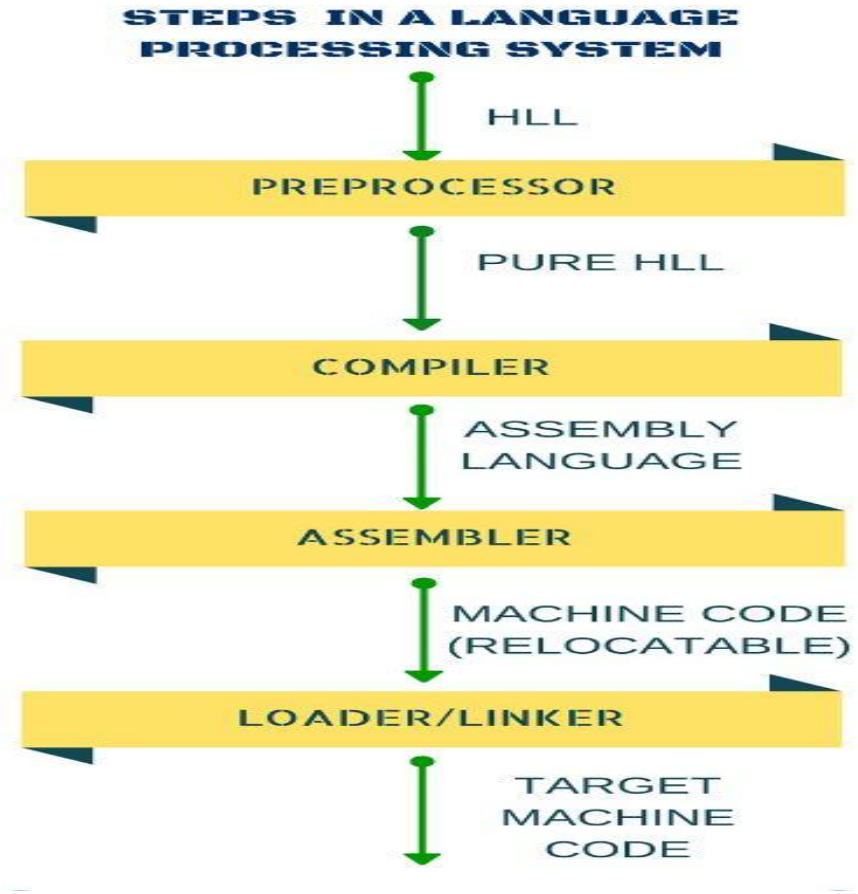
Language Processing System

- Computers operate using low-level machine code, which is difficult for humans to read and write. To overcome this complexity, we use high-level programming languages that are more intuitive and easier to understand.
- However, these human-readable programs must be converted into a form that machines can execute efficiently. This conversion process involves a series of translation and optimization steps, collectively known as language processing.
- By leveraging tools like compilers, interpreters, and assemblers, we transform high-level code into low-level instructions that the hardware can directly execute. In this way, language processing systems bridge the gap between human-friendly languages and machine-readable code, ensuring that our software runs smoothly on hardware.



High Level Language

- High-level languages (HLLs) often include preprocessor directives, such as #include and #define, which simplify coding and make programs more readable. These directives instruct the preprocessor to integrate external files, define constants, and configure the code before it’s compiled. While this approach brings the program closer to human understanding, it remains distant from the machine’s low-level operations.




```c
#include <stdio.h>
#include <unistd.h>

int main ()
{
        printf("Running 'net join' with the following parameters: \n");
        char *domain="mydomain";
        char *user="domainjoinuser";
        char *pass="mypassword";
        char *vastool="/opt/quest/bin/vastool";
        char *ou="OU=test,DC=mtdomian,DC=local";
        char unjoin[512];

        sprintf(unjoin,"/opt/quest/bin/vastool -u %s -w '%s' unjoin -f",user,pass);

        printf("Domain: %s\n",domain);
        printf("User: %s\n",user);
        printf("-----------------\n");

        printf("\nUnjoin..........\n");
        system(unjoin);

        printf("\nJoin.............\n");
        execl("/opt/quest/bin/vastool", "vastool", "-u", user, "-w", pass, "join", "-c", ou,
"-f", domain, (char*)0);

}
```

Macros may be nested

- in definitions, e.g.:

```
#define  Pi          3.1416
#define  Twice_Pi    2*Pi
```

- in uses, e.g.:

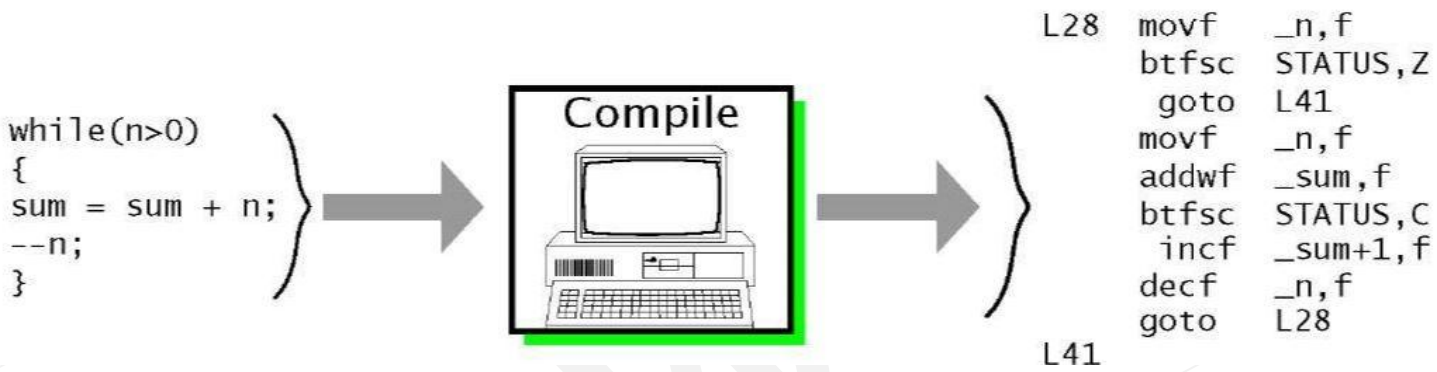
```
#define  double(x)   x+x
#define  Pi  3.1416
...
if ( x > double(Pi) ) ...
```

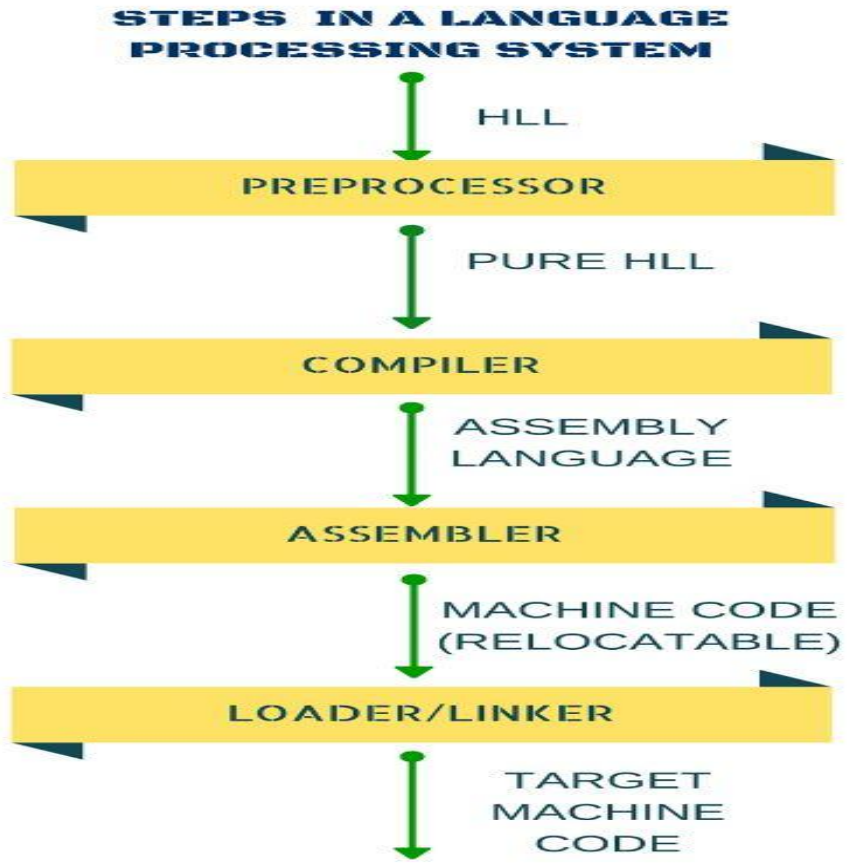
Simple assignment operator	Shorthand operator
a = a+1	a + =1
a = a-1	a - =1
a = a* (m+n)	a * = m+n
a = a / (m+n)	a / = m+n
a = a %b	a %=b

- The preprocessor handles several tasks before compilation, such as including external files in place of #include directives and expanding #define macros. Essentially, it streamlines the source code by performing file inclusion, macro-processing, and other shorthand operations, ensuring that the compiler receives a fully prepared codebase.

Compiler

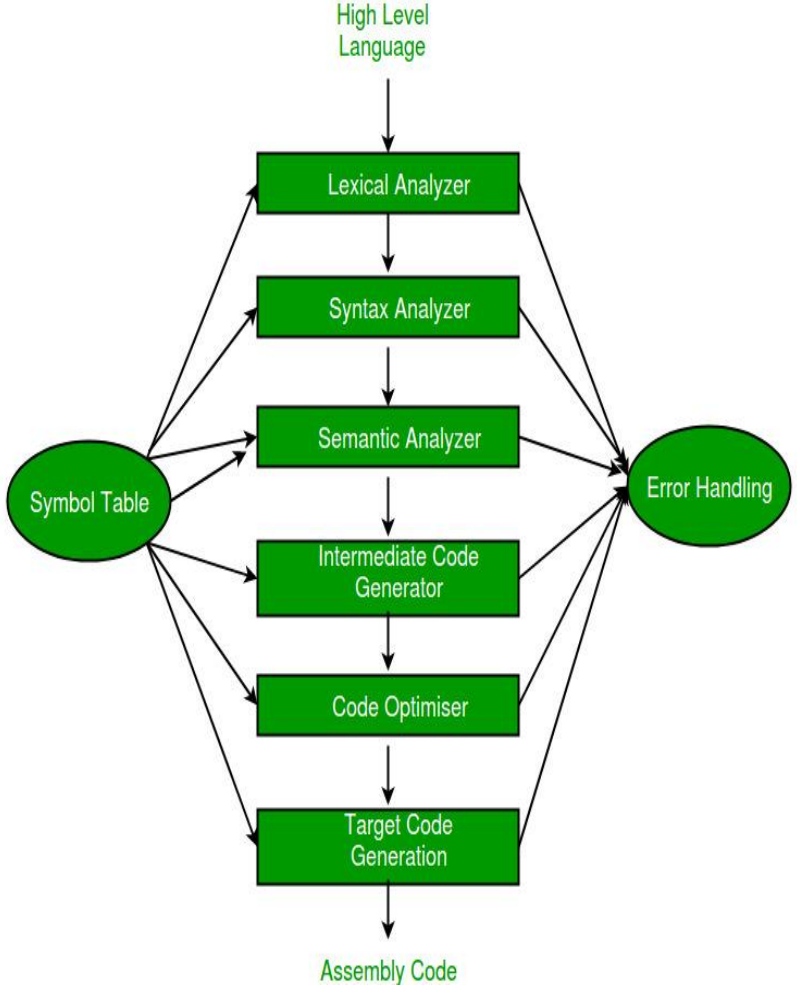
- A compiler is a tool that transforms source code written in a high-level, human-friendly language into a lower-level language, such as assembly, ultimately producing an executable program. This conversion enables developers to write code in a more intuitive, maintainable form without having to deal directly with the complexity of machine-level instructions.

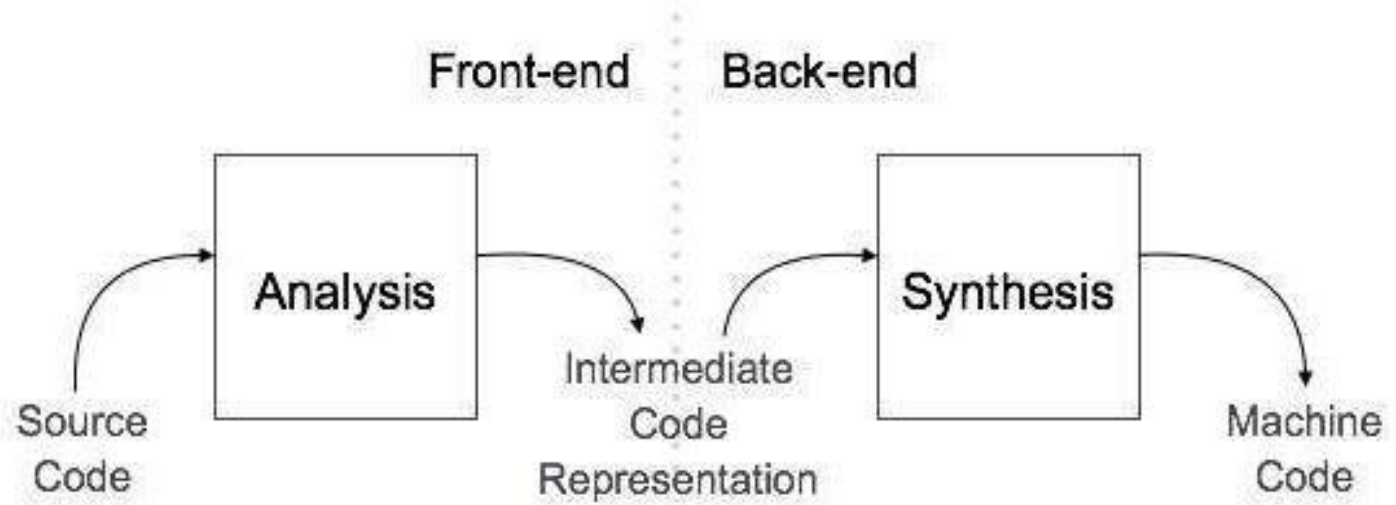




Compiler Design

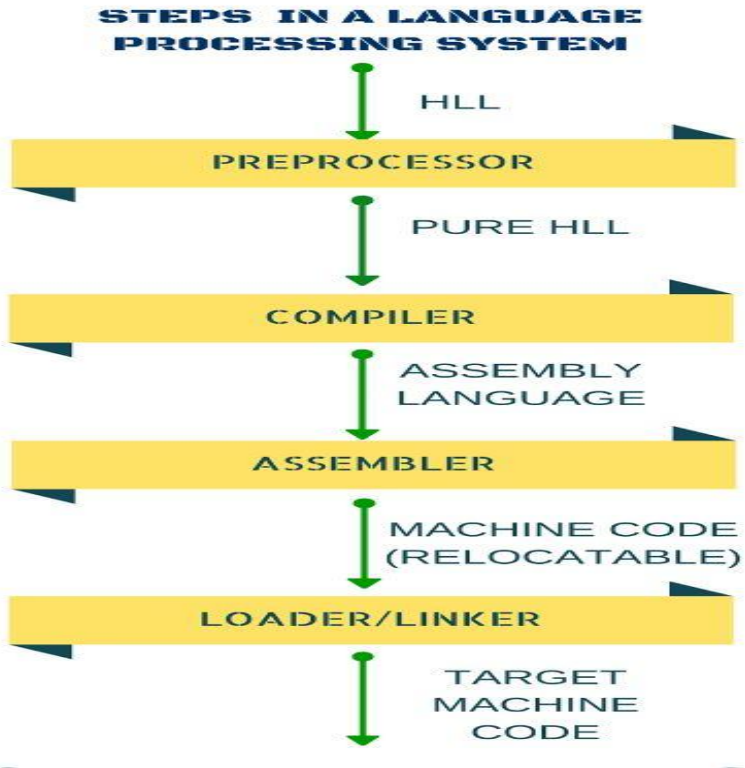
- **Analysis Phase:**
  - Converts source code into an intermediate representation (IR).
  - Focuses on understanding the program's structure and semantics.
- **Synthesis Phase:**
  - Transforms the intermediate representation into the target program (e.g., assembly or machine code).
  - Produces executable code equivalent to the original source.

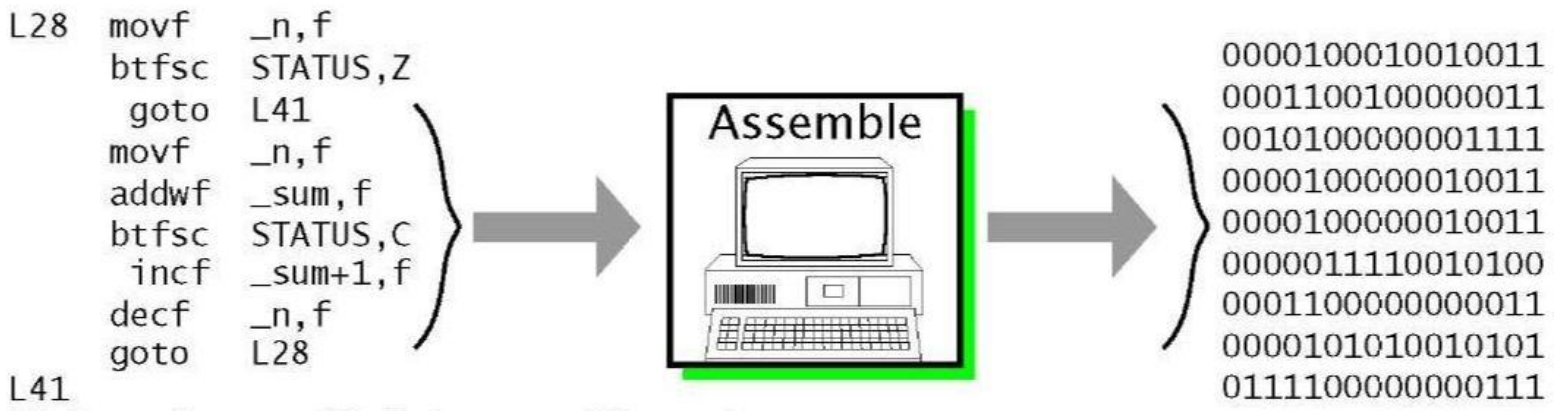


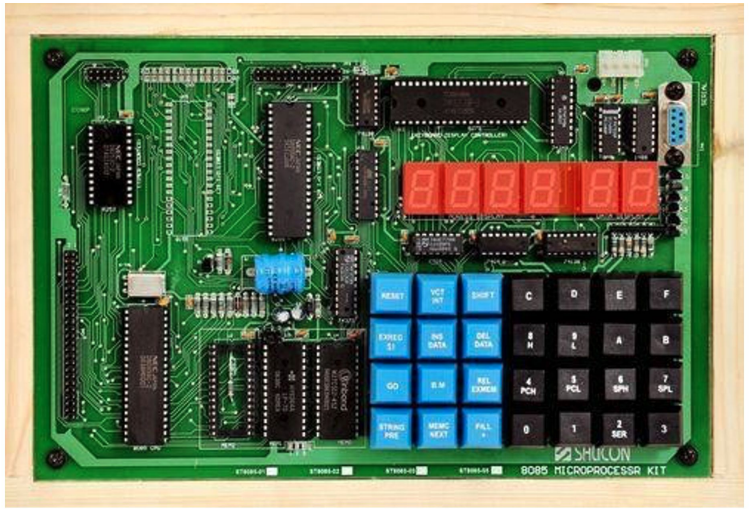


Assembly Language

- **Assembly Language Overview:** It serves as an intermediate representation between high-level languages and binary machine code. Consists of machine instructions and additional useful data required for execution.
- **Platform-Specific Nature:** Each hardware and operating system (platform) has its unique assembler. Assembly languages are not universal and vary based on the platform.
- **Assembler Functionality:** Translates assembly language into machine code. Produces an output called the **object file**, which can be executed on the target platform.



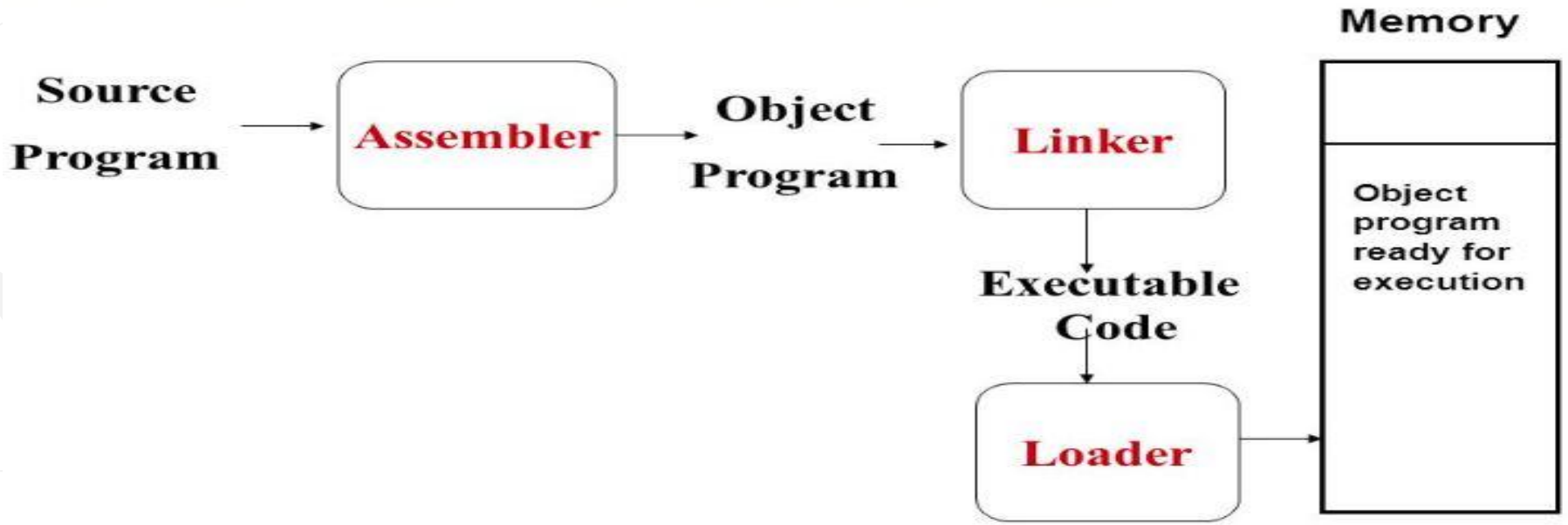




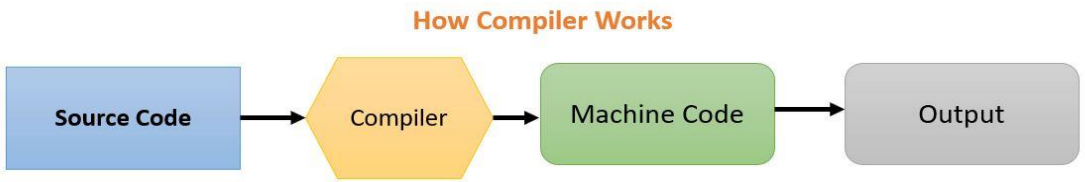
8085 Microprocessor

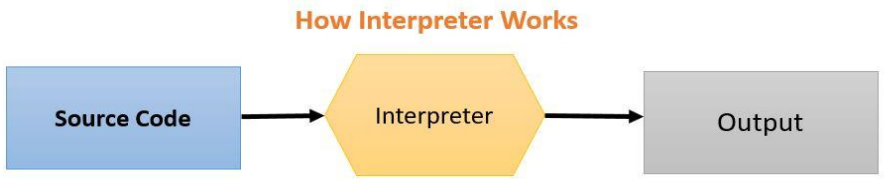
Linker Loader

- **Linker:**
  - Combines multiple object files into a single executable file.
  - Converts relocatable code into absolute code, ready to run.
  - Produces an output that either runs successfully or generates an error message.
- **Loader:**
  - Loads the executable code into memory.
  - Executes the program directly from memory.
- **Workflow:**
  - The source program is converted into an object program by the assembler.
  - The linker processes object programs to create executable code.
  - The loader moves the executable code into memory and executes it.

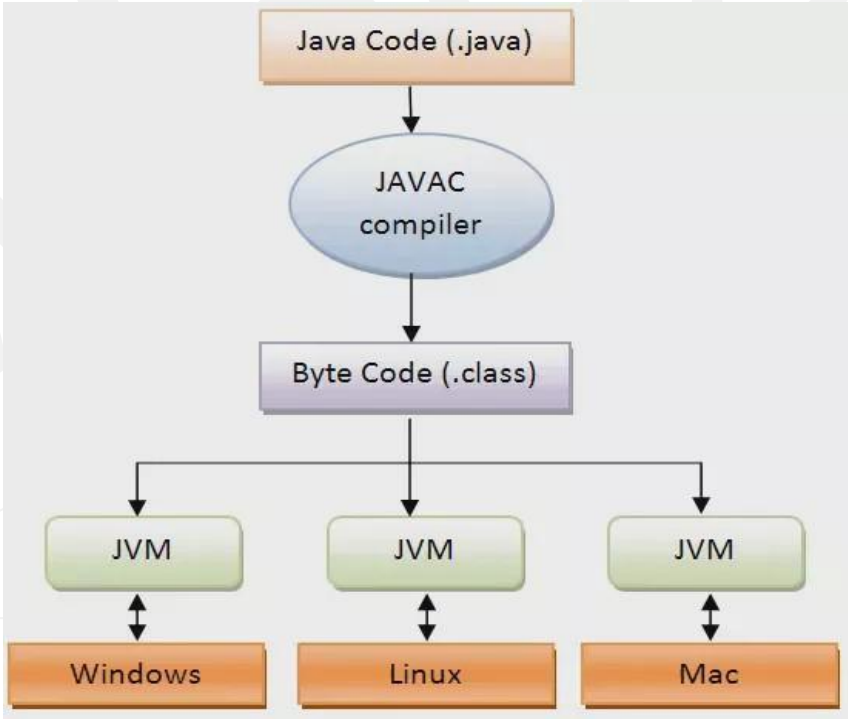


BASIS FOR COMPARISON	COMPILER	INTERPRETER
Input	It takes an entire program at a time.	It takes a single line of code or instruction at a time.
Output	It generates intermediate object code.	It does not produce any intermediate object code.
Working mechanism	The compilation is done before execution.	Compilation 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
Code Optimization	Code Optimization is possible to a much greater extent	Code Optimization is not very effective
Pertaining Programming languages	C, C++, C#, Scala, typescript uses compiler.	PHP, Perl, Python, Ruby uses an interpreter.





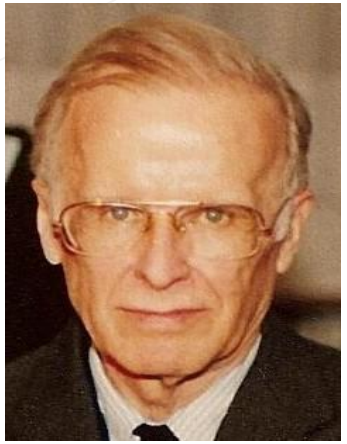

JAVA
Compiler
JAVA
Program.java
CLASS
Program.class
JVM
10101100
Program


Java Code (.java)
JAVAC
compiler
Byte Code (.class)
JVM
JVM
JVM
Windows
Linux
Mac

- **1951**: The first practical compiler was written by **Corrado Böhm** for his PhD thesis.

- **1952**: The first modern compiler and **Autocode** were developed by **Alick Glennie** at the University of Manchester for the Mark 1 computer.

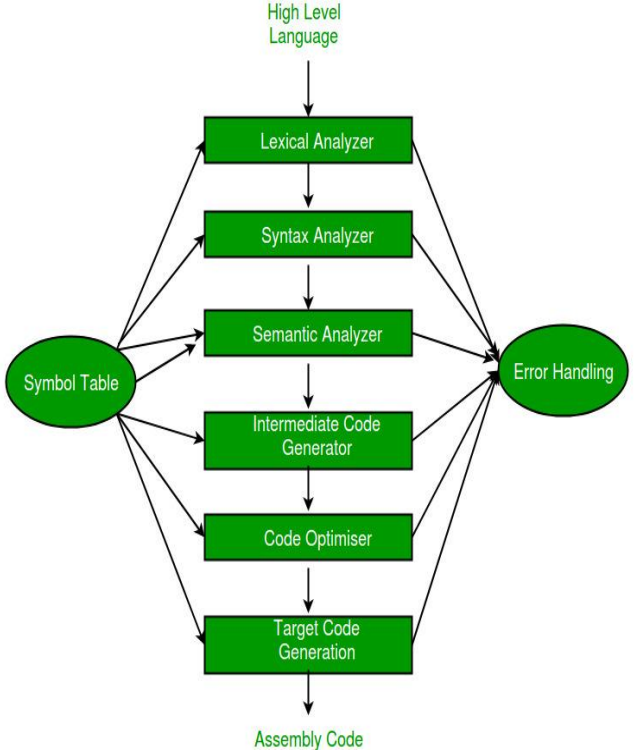
- **1957**: The first commercially available compiler was introduced by IBM's **FORTRAN team**, led by **John W. Backus**. It took 18 person-years to develop.



- **Grace Hopper** coined the term "compiler" and developed the **A-0 system**, which functioned as a loader or linker rather than a compiler in the modern sense.

Lexical Analyzer

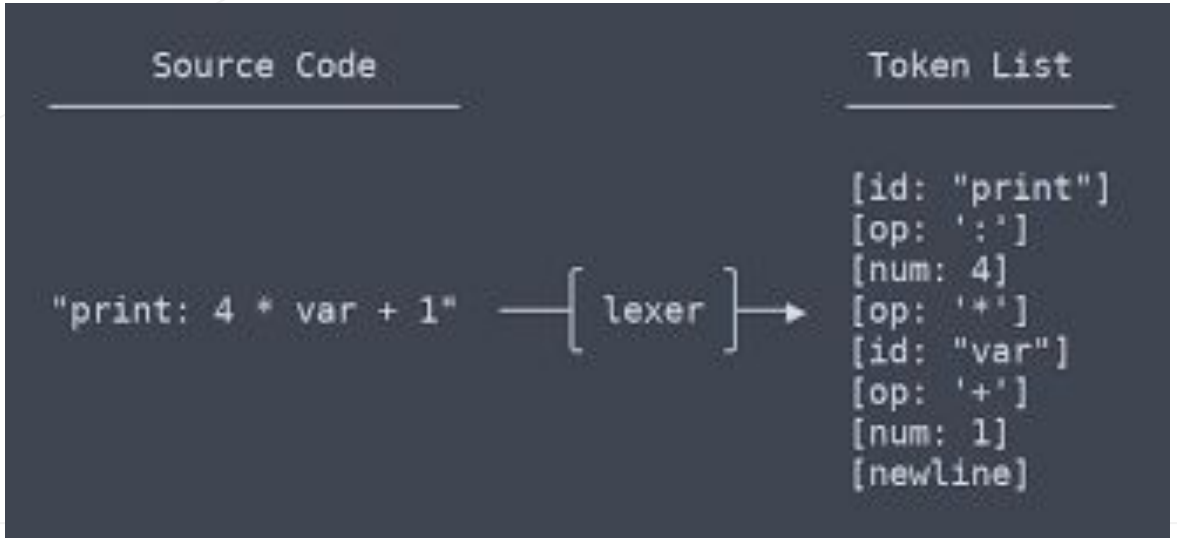
- **Purpose of Lexical Analysis:** Reads the program and breaks it into **lexemes** (smallest units of meaningful text). Converts lexemes into **tokens**, which are predefined structures recognized by the lexical analyzer.
- **Tokenization:** Tokens are categorized using **regular expressions**. The lexical analyzer identifies and assigns token types like identifiers, operators, and constants.



Float x, y, z;
x = y+ z*60

X -> token -> identifier
= -> token -> operator
Y -> token -> identifier
+ -> token -> operator
Z -> token -> identifier
* -> token -> operator
60 -> token -> constant

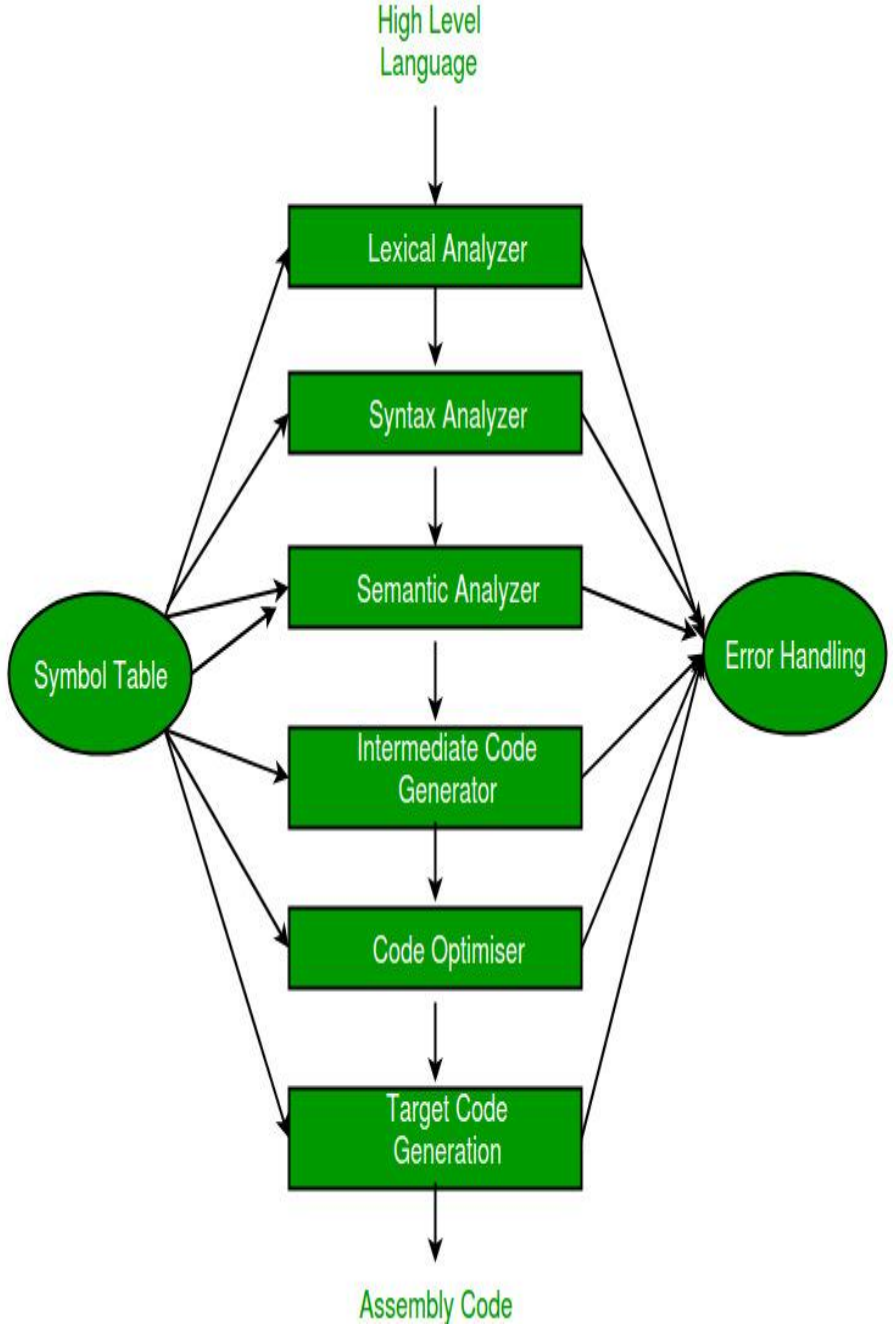
$id_1 = id_2 + id_3 * 60$



Syntax Analyzer

- **Role:** Responsible for analyzing the structure of the code. Ensures the code adheres to the grammatical rules of the programming language.
- **Key Features:** Constructs a **Parse Tree** or **Syntax Tree** to represent the program's grammatical structure. Utilizes **Context-Free Grammar (CFG)** rules to build the parse tree.
- **Process:** Reads tokens generated by the lexical analyzer, one by one. Verifies the syntax based on predefined production rules.

S -> id = E
E -> E + T / T
T -> T * F / F
F -> id / num
$id_1 = id_2 + id_3 * 60$



1. *program* → *declarationList*
2. *declarationList* → *declarationList declaration* | *declaration*
3. *declaration* → *varDeclaration* | *funDeclaration*

---

4. *varDeclaration* → *typeSpecifier varDeclList* ;
5. *scopedVarDeclaration* → *scopedTypeSpecifier varDeclList* ;
6. *varDeclList* → *varDeclList* , *varDeclInitialize* | *varDeclInitialize*
7. *varDeclInitialize* → *varDeclId* | *varDeclId* : *simpleExpression*
8. *varDeclId* → **ID** | **ID** [**NUMCONST**]
9. *scopedTypeSpecifier* → **static** *typeSpecifier* | *typeSpecifier*
10. *typeSpecifier* → **int** | **bool** | **char**

---

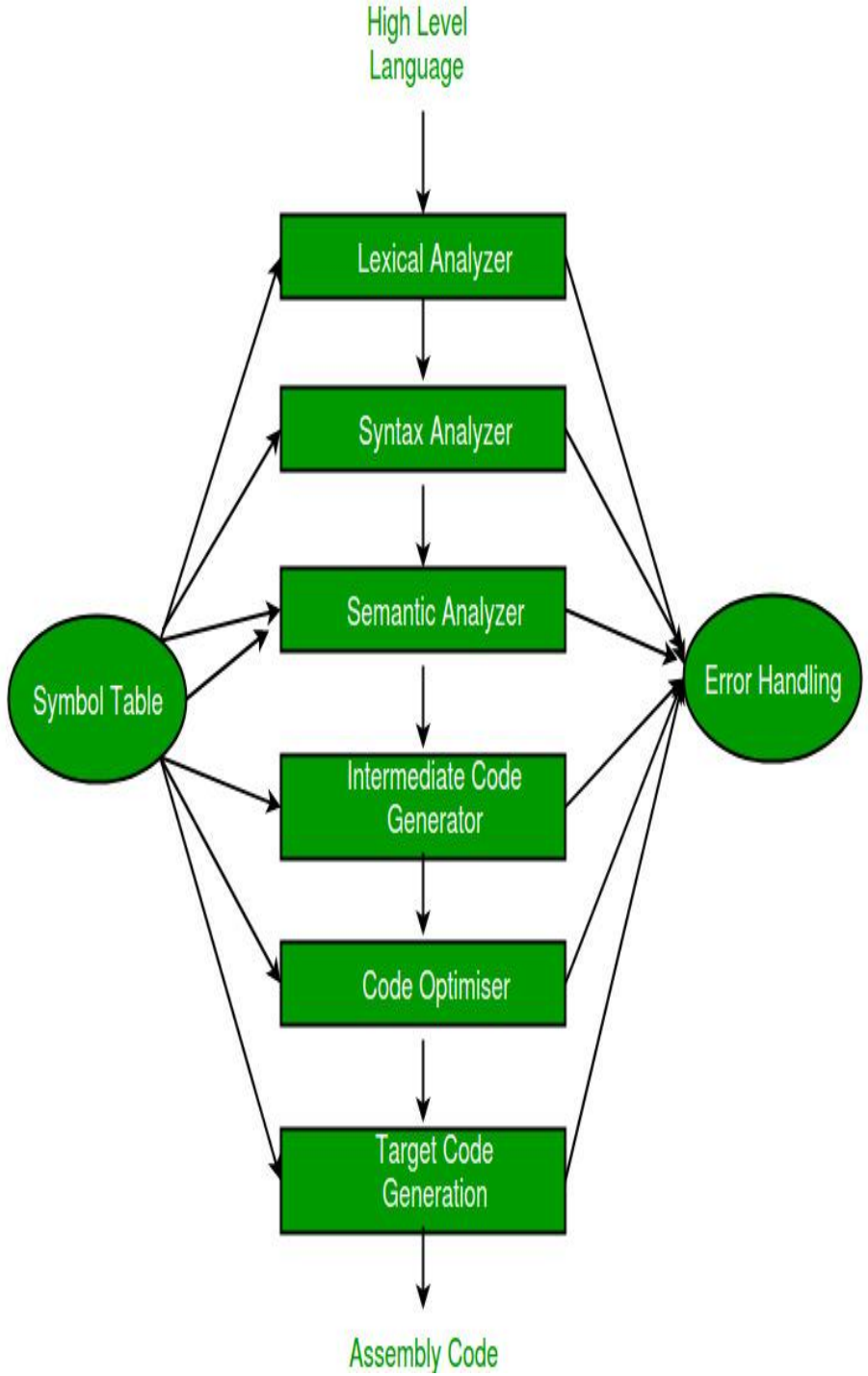
11. *funDeclaration* → *typeSpecifier* **ID** (*params*) *statement* | **ID** (*params*) *statement*
12. *params* → *paramList* | ϵ
13. *paramList* → *paramList* ; *paramTypeList* | *paramTypeList*
14. *paramTypeList* → *typeSpecifier paramIdList*
15. *paramIdList* → *paramIdList* , *paramId* | *paramId*
16. *paramId* → **ID** | **ID** [ ]

---

17. *statement* → *expressionStmt* | *compoundStmt* | *selectionStmt* | *iterationStmt* | *returnStmt* | *breakStmt*
18. *expressionStmt* → *expression* ; | ;
19. *compoundStmt* → { *localDeclarations statementList* }
20. *localDeclarations* → *localDeclarations scopedVarDeclaration* | ϵ
21. *statementList* → *statementList statement* | ϵ
22. *elsifList* → *elsifList* **elsif** *simpleExpression* **then** *statement* | ϵ
23. *selectionStmt* → **if** *simpleExpression* **then** *statement elsifList* | **if** *simpleExpression* **then** *statement elsifList* **else** *statement*
24. *iterationRange* → **ID** = *simpleExpression* .. *simpleExpression* | **ID** = *simpleExpression* .. *simpleExpression* : *simpleExpression*
25. *iterationStmt* → **while** *simpleExpression* **do** *statement* | **loop forever** *statement* | **loop** *iterationRange* **do** *statement*
26. *returnStmt* → **return** ; | **return** *expression* ;
27. *breakStmt* → **break** ;
28. *expression* → *mutable* = *expression* | *mutable* += *expression* | *mutable* −= *expression* | *mutable* *= *expression* | *mutable* /= *expression* | *mutable* ++ | *mutable* −− | *simpleExpression*
29. *simpleExpression* → *simpleExpression* **or** *andExpression* | *andExpression*
30. *andExpression* → *andExpression* **and** *unaryRelExpression* | *unaryRelExpression*
31. *unaryRelExpression* → **not** *unaryRelExpression* | *relExpression*
32. *relExpression* → *sumExpression relop sumExpression* | *sumExpression*
33. *relop* → <= | < | > | >= | == | !=
34. *sumExpression* → *sumExpression sumop mulExpression* | *mulExpression*
35. *sumop* → + | −
36. *mulExpression* → *mulExpression mulop unaryExpression* | *unaryExpression*
37. *mulop* → * | / | %
38. *unaryExpression* → *unaryop unaryExpression* | *factor*
39. *unaryop* → − | * | ?
40. *factor* → *immutable* | *mutable*
41. *mutable* → **ID** | *mutable* [*expression*]
42. *immutable* → (*expression*) | *call* | *constant*
43. *call* → **ID** (*args*)
44. *args* → *argList* | ϵ
45. *argList* → *argList* , *expression* | *expression*
46. *constant* → **NUMCONST** | **CHARCONST** | **STRINGCONST** | **true** | **false**

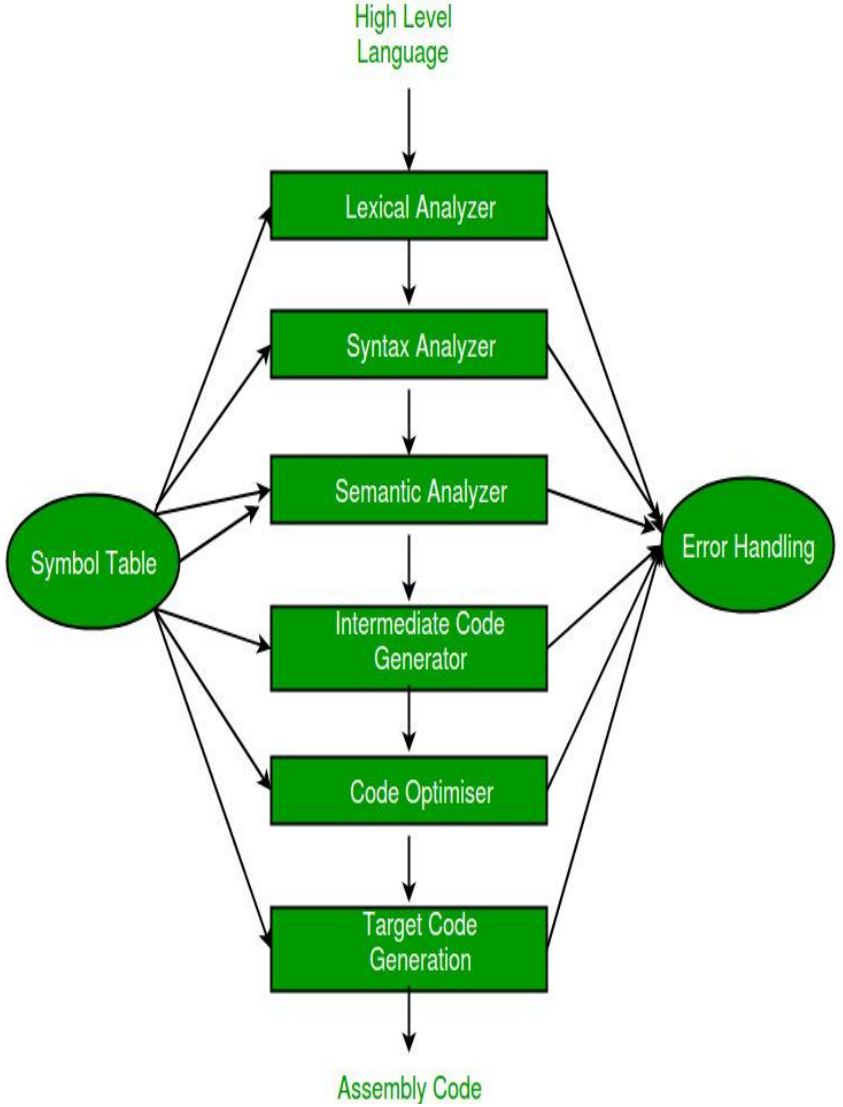
Semantic Analyzer

- **Role:** Ensures that the parse tree constructed by the syntax analyzer is meaningful and adheres to semantic rules.
- **Key Features:** Verifies the logical correctness of the parse tree. Checks for semantic errors like type mismatches, undeclared variables, or invalid operations.
- **Output:** Produces a **verified** or **annotated parse tree** that includes additional information, such as data types and scope rules, for further processing.



Intermediate Code Generator

- **Role:** Produces an **intermediate code** that is simpler than source code and easier for machines to interpret. Acts as a bridge between the front-end (language-dependent) and back-end (platform-dependent) of the compiler.
- **Characteristics:** Intermediate code is **platform-independent**. Common formats include **three-address code (TAC)**, among others.
- **Process Example:** For the expression
  - x = y + z * 60:
  - t_1 = z * 60
  - t_2 = y + t_1
  - x = t_2
- **Advantages:** Simplifies the development of new compilers by reusing intermediate code from existing compilers. Final translation to machine code depends on the platform and occurs in the last two compiler phases, making it platform-dependent.

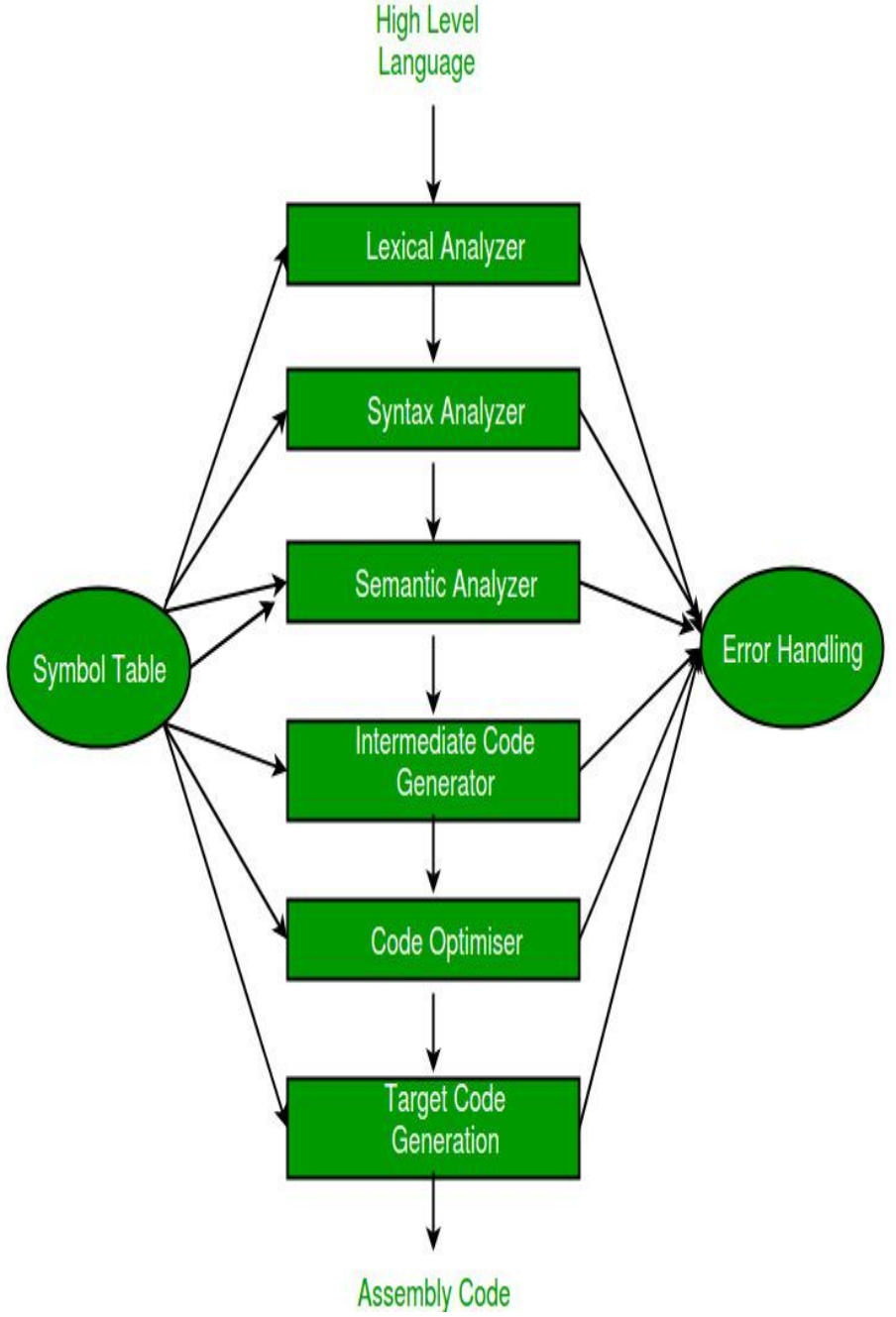


Code Optimizer

- **Purpose:** Improves the **efficiency** of the code by reducing resource consumption and enhancing execution speed. Ensures that the **meaning** of the original code remains unchanged.
- **Types of Optimization:**
  - **Machine-Independent Optimization:** Focuses on general improvements applicable across platforms (e.g., eliminating redundant calculations).
  - **Machine-Dependent Optimization:** Tailored for specific hardware to utilize its features efficiently (e.g., using specific instructions or registers).

$t_1 = z * 60$

$x = y + t_1$

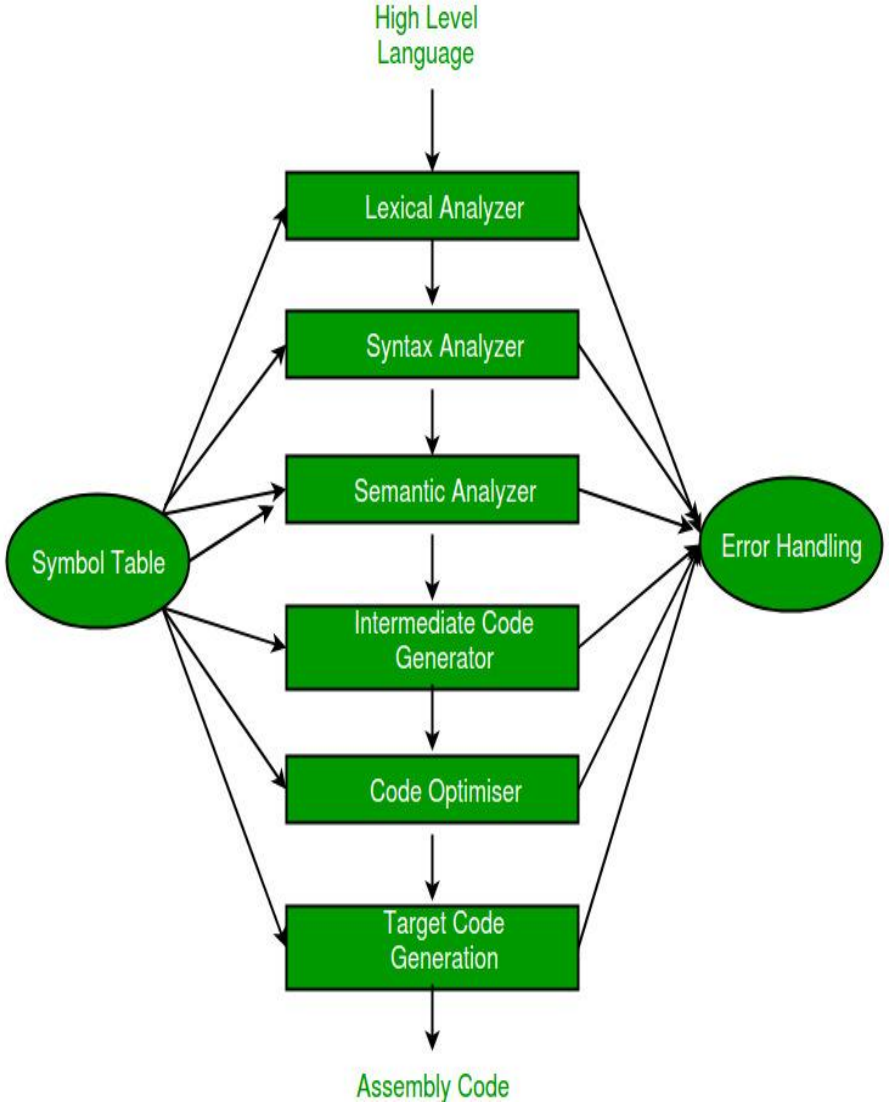


Target Code Generator

- **Purpose:** Converts intermediate code into **machine-understandable code**. Handles **register allocation** and **instruction selection** to optimize the code for execution.

- **Characteristics:** The output is dependent on the **type of assembler** and the target machine architecture. This is the **final phase** of the compilation process.

MOV R_1, Z
MUL R_1, 60
ADD R_1, Y
STORE X, R_1

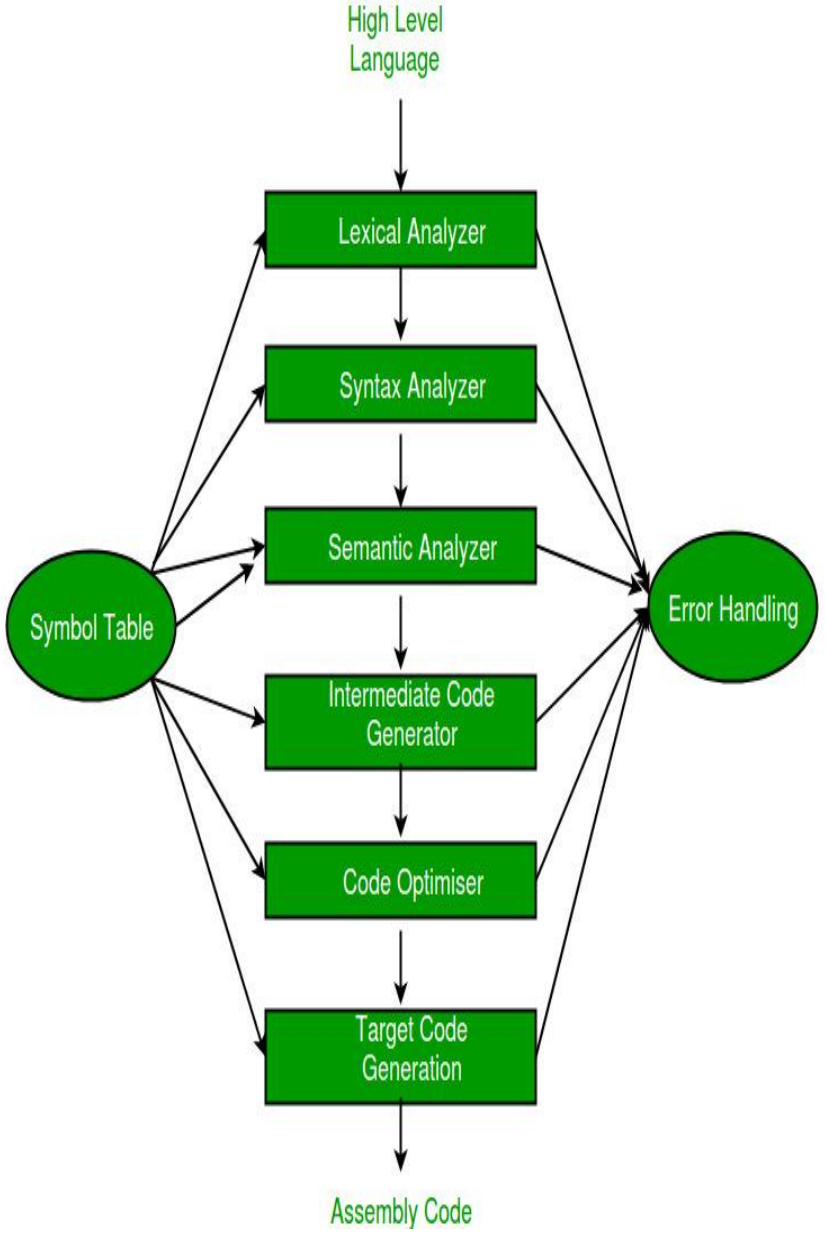


Symbol Table

- A **data structure** used and maintained by the compiler to store all identifiers' names along with their types, scope, and other information.
- Facilitates **smooth compiler operations** by allowing quick lookups for identifiers during compilation.
- Tracks variables, arrays, records, procedures, and functions across the program.

1. int x = 10;

Line No	Keyword	identifier	Constant	Operator
1	int	x	10	;



- **Information Stored:**
  - **General:** Name, type, scope, size, and offset.
  - **Specific:**
    - Arrays: Size.
    - Records: Column names.
    - Procedures/Functions: Input/output parameters, actual/formal parameters.

- **Compiler Interaction:**
  - **Front-End (Lexical, Syntax, and Semantic Analyzers):** Fills the symbol table with information.
  - **Back-End (Optimization, Code Generation):** Uses the symbol table for efficient code production.
  - The **symbol table** is populated during the **lexical analysis phase**, and each phase of the compiler interacts with it.

- **Operations on Symbol Table:**
  - **Insert:** Add new identifiers.
  - **Lookup/Search:** Retrieve information about an identifier.
  - **Modify:** Update existing information.
  - **Delete:** Remove an identifier.

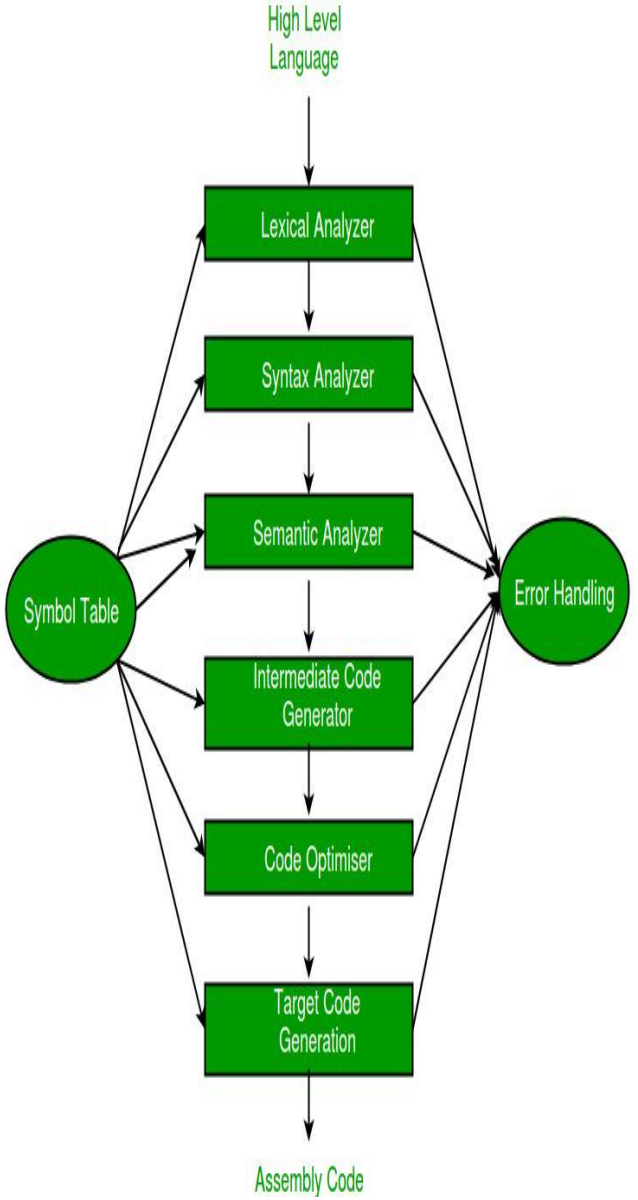
- **Implementation Techniques:**
  - Linear Table.
  - Binary Search Tree.
  - Linked List.
  - **Hash Table (Most Popular):** Offers efficient lookup and insertion.

Q. Which ONE of the following statements is FALSE regarding the symbol table? **(Gate 2025)**

A) Symbol table is responsible for keeping track of the scope of variables.
B) Symbol table can be implemented using a binary search tree.
C) Symbol table is not required after the parsing phase.
D) Symbol table is created during the lexical analysis phase.

Error handler

- **Purpose:** Ensures the compilation process continues even if errors occur during any phase of compilation. Collects and reports errors, allowing the programmer to correct them.
- **Process:** Errors detected during **Phase 1, 2, or 3** are recorded in the **error handler object**. After Phase 3: If the error handler is **empty**, the code is error-free and ready for conversion to target code. If the error handler contains errors, they are displayed for correction.
- **Types of Errors Handled:**
  - **Lexical Errors:** Detected during lexical analysis (e.g., invalid tokens).
  - **Syntax Errors:** Detected during syntax analysis (e.g., missing parentheses).
  - **Semantic Errors:** Detected during semantic analysis (e.g., type mismatches).
- **Error Types Beyond Compilation:**
  - **Exceptions:** Errors during compilation handled by the compiler (e.g., division by zero).
  - **Fatal Errors:** Runtime errors requiring system administrator intervention (e.g., system crashes).
- **Key Responsibility:**
  - The **compiler** handles exceptions during the compilation phase.
  - The **programmer** is responsible for handling exceptions within the code.
  - **System administrators** manage fatal errors at runtime.



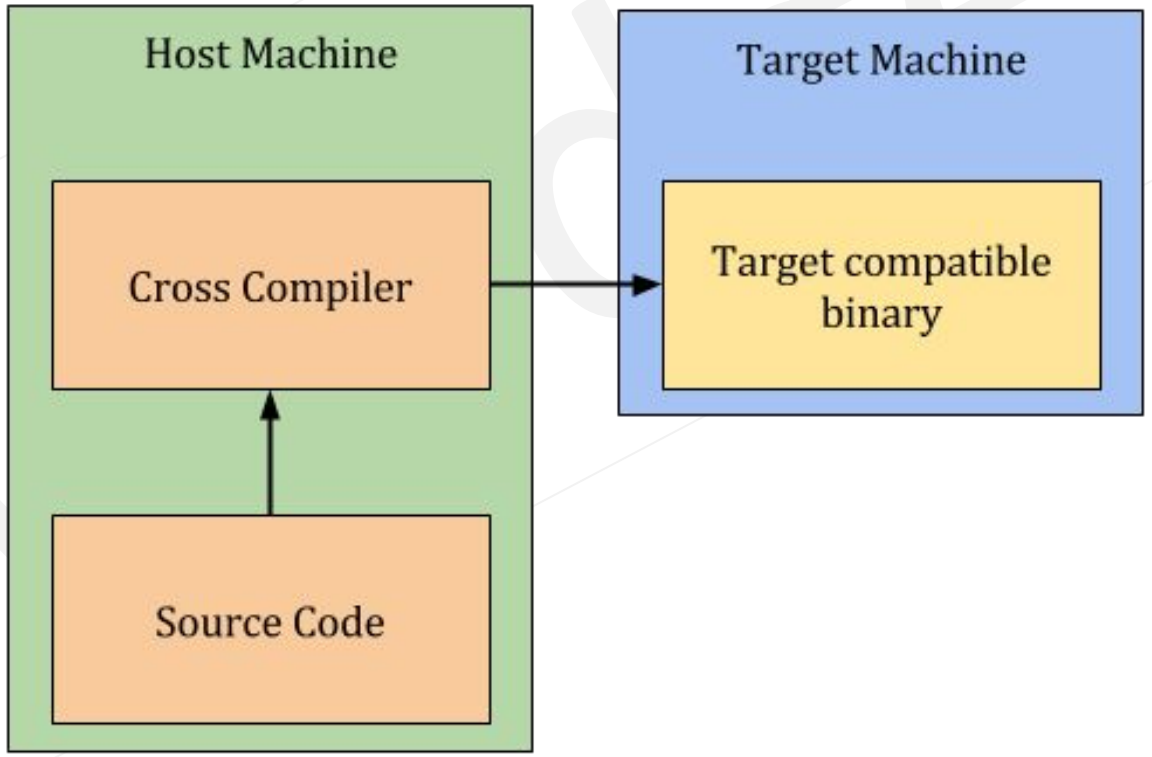
- **Phases and Passes in a Compiler**
- **Pass:**
  - A **pass** refers to a full traversal of the compiler through the program or part of it. A compiler may require single or multiple passes to process the source code.
- **Phase:**
  - A **phase** is a distinct stage in the compilation process. Each phase takes input from the previous stage, processes it, and produces output for the next phase. Multiple phases can occur in a single pass.

- **Single Pass Compiler: Definition:** Processes each part of the program exactly once, moving directly from lexical analysis to code generation.
  - **Advantages:** Faster and smaller in size.
  - **Disadvantages:** Less efficient compared to multi-pass compilers. Cannot revisit previous phases; hence, the grammar must be simple or limited.
  - **Example Language:** Pascal.

- **Multi-Pass Compiler: Definition:** Processes the source code multiple times, with each pass focusing on a specific aspect.
- **Workflow:**
  - **First Pass:** Reads source code, extracts tokens, and stores the result.
  - **Second Pass:** Reads the stored tokens, builds the syntax tree, and performs syntactic analysis.
  - **Third Pass (Optional):** Verifies that the syntax tree adheres to the rules of the language (semantic analysis). Continues until the target code is generated.
- **Advantages:** More efficient and flexible; handles complex grammars and optimizations.
- **Disadvantages:** Slower than single-pass compilers due to multiple traversals.

Cross Compiler

- **Definition:** A compiler that generates executable code for a platform different from the one it is running on (e.g., compiling on Windows for Android).
- **Purpose:** Enables code to be compiled for multiple platforms from a single development host.
- **Historical Use:** Allowed one computer to generate executables for multiple operating systems.



Cross Compiler operation

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)

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)

(a) P ↔ i, Q ↔ ii, R ↔ iv, S ↔ iii

(b) P ↔ iii, Q ↔ i, R ↔ ii, S ↔ iv

(c) P ↔ ii, Q ↔ iii, R ↔ i, S ↔ iv

(d) P ↔ iv, Q ↔ i, R ↔ ii, S ↔ iii

Match the following:

(GATE - 2015) (2 Marks)

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

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

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

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

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

Q. Consider the following two sets: **(Gate 2024 CS) (1 Marks) (MCQ)**

Set X	Set Y
P. Lexical Analyzer	1. Abstract Syntax Tree
Q. Syntax Analyzer	2. Token
R. Intermediate Code Generator	3. Parse Tree
S. Code Optimizer	4. Constant Folding

Which one of the following options is the CORRECT match from Set X to Set Y ?

(a) P – 4; Q – 1; R – 3; S – 2

(b) P – 2; Q – 3; R – 1; S – 4

(c) P – 2; Q – 1; R – 3; S – 4

(d) P – 4; Q – 3; R – 2; S – 1

Q Consider the following ANSI C program:

```
int main () {
   Integer x;
   return 0;
}
```

Which one of the following phases in a seven-phase C compiler will throw an error?**(GATE 2021)**
(1 MARKS)

(A) Lexical analyzer

(B) Syntax analyzer

(C) Semantic analyzer

(D) Machine dependent optimizer

Q Consider the following statements regarding the front-end and back-end of a compiler.

S_1: The front-end includes phases that are independent of the target hardware.

S_2: The back-end includes phases that are specific to the target hardware.

S_3: The back-end includes phases that are specific to the programming language used in the source code.

Identify the CORRECT option. **(Gate-2023) (1 Marks)**

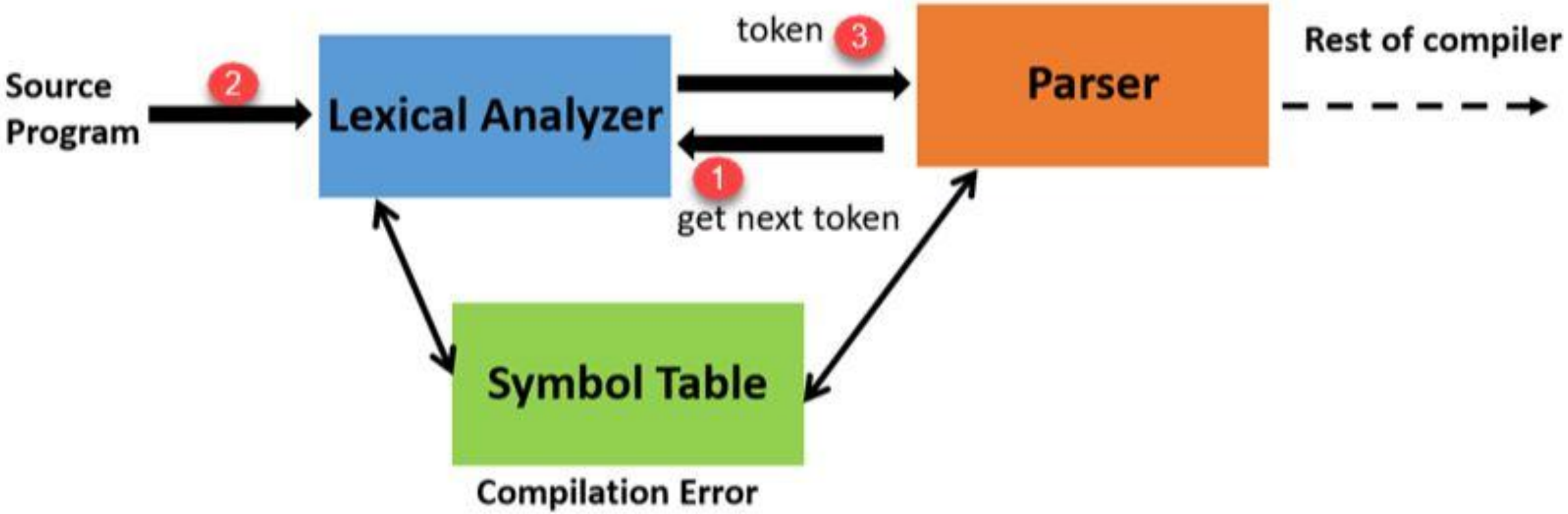
(a) Only S_1 is TRUE.

(b) Only S_1 and S_2 are TRUE.

(c) S_1, S_2, and S_3 are all TRUE.

(d) Only S_1 and S_3 are TRUE.

Lexical Analyzer



Lexical Analyzer

- In computer science, **lexical analysis**, **lexing** or **tokenization** is the process of converting a sequence of characters (such as in a computer program or web page) into a sequence of tokens.

- **Key Concepts:**
  - **Lexeme:** Actual sequence of characters that matches the pattern for a token.
  - **Token:** Logical meaning assigned to a lexeme, such as identifier, operator, or keyword.

- **Stages in Lexing:**
  - **Scanning:** Segments the input into syntactic units (lexemes) and categorizes them into token classes.
  - **Evaluating:** Converts lexemes into processed values.

- Consider this expression in the C programming language:
  - x = a + b * 2;

- The lexical analysis of this expression yields the following sequence of tokens:
  - [(identifier, x), (operator, =), (identifier, a), (operator, +), (identifier, b), (operator, *), (literal, 2), (separator, ;)]

- **Implementation:**
  - Performed by a **Lexer** or **Lexical Generator**.
- **Application:**
  - Used in programming language compilers and natural language processing for segmenting text into units like words or tokens.

```
letter      = [a-zA-Z]
digit       = [0-9]
digits      = digit digit*
identifier  = letter | (letter | digit | _)*
fraction    = . digits | ε
exponent    = ((E | e) (+ | - | ε)) digits) | ε
number      = digits fraction exponent
operator    = + | - | * | / | > | >= | < | <= | = | ==
parenthesis = ( | )
```

- Secondary Function of Lexical Analyser are: -
  - Removal of Comments lines

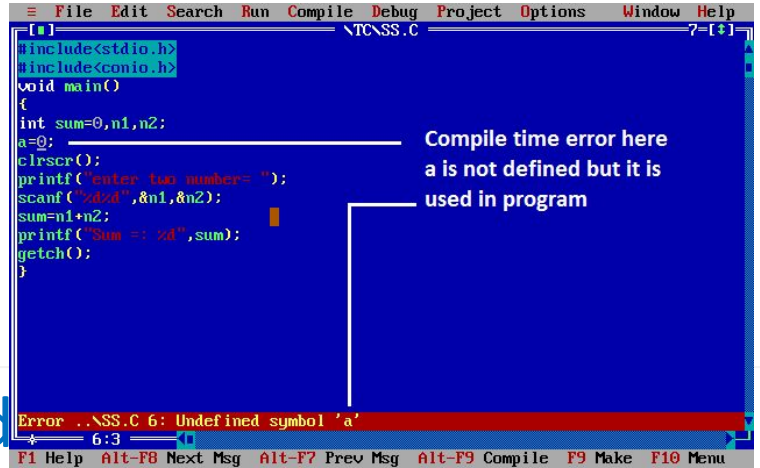
```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int a; // Declare two variables
    printf("Enter any number\n\n");
    scanf("%d",&a);
/* After declaration, we print our message on the screen*/
/* and in the 3rd step, save the address of a */
   return 0;
}
```

  - Removal of White space characters

WHITE SPACE CHARACTERS

\b	blank space	\t	horizontal tab	\v	vertical tab
\\	Back slash	\'	Single quote	\"	Double quote
\r	carriage return	\f	form feed	\n	new line
\?	Question mark	\0	Null	\a	Alarm (bell)

  - Co-relating with Error Messages along with line number.



Implementation

- **Lexical Grammar:**
  - Defines the lexical syntax of a programming language using **regular expressions** that specify possible character sequences (**lexemes**) for a token.
- **Lexer Role:**
  - Recognizes strings and converts them into tokens by matching them with predefined patterns.

- **Design of Lexical Analyzer:**
  - **Hand Coding:**
    - Construct regular expressions for token patterns.
    - Convert regular expressions into finite automata for efficient string matching.
  - **Lex Tool:**
    - Automates lexer creation by generating code based on regular expression rules.

Q Identify the meaning of the rule implemented by Lexer using regular expression for identifying tokens of a password?

Regular Expression -> $A(A+S+D)^7 (A+S+D+\in)^7$

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

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

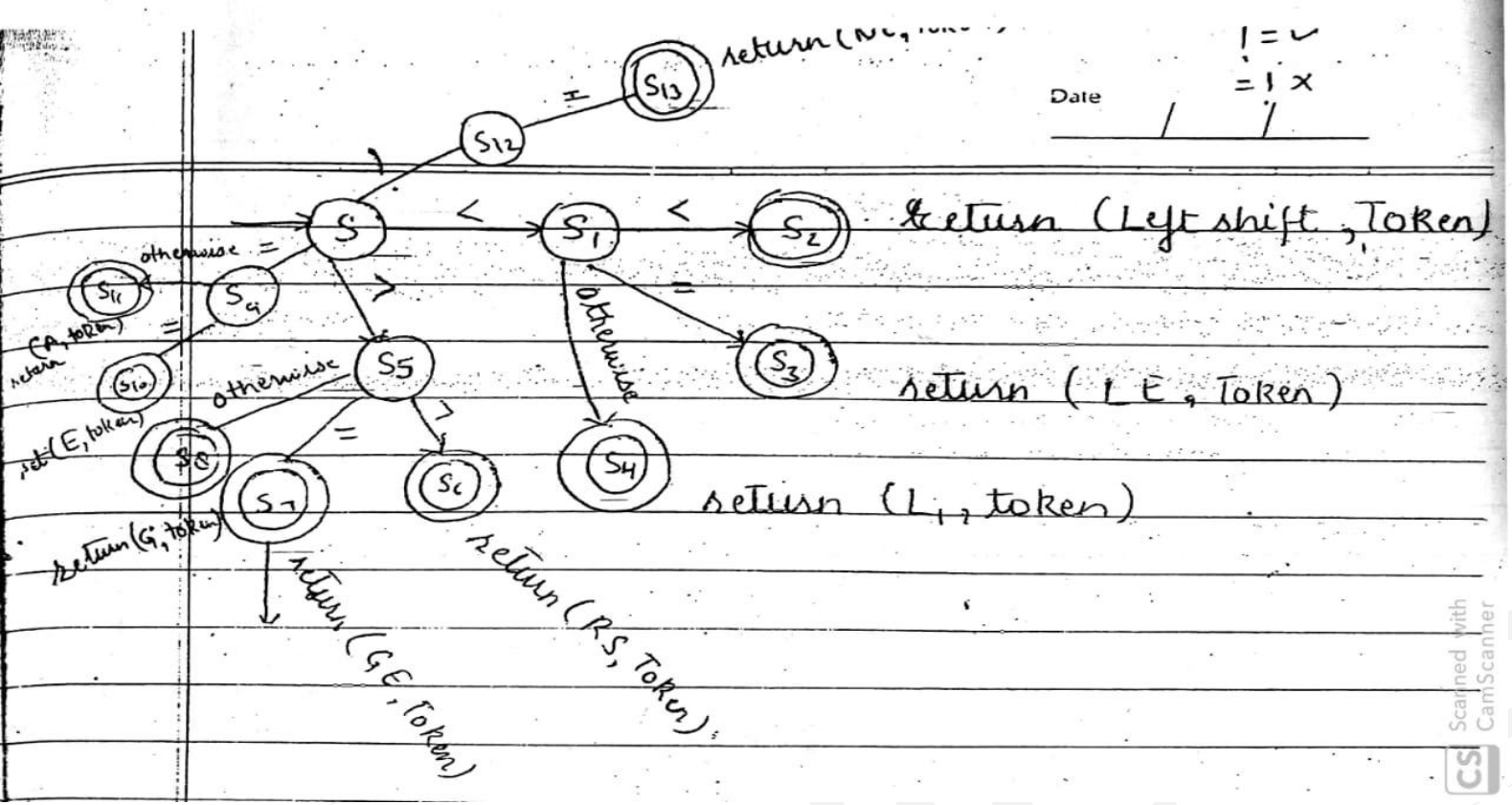
```
printf("i = %d, &i = %x", i, &i);
```

(A) 3

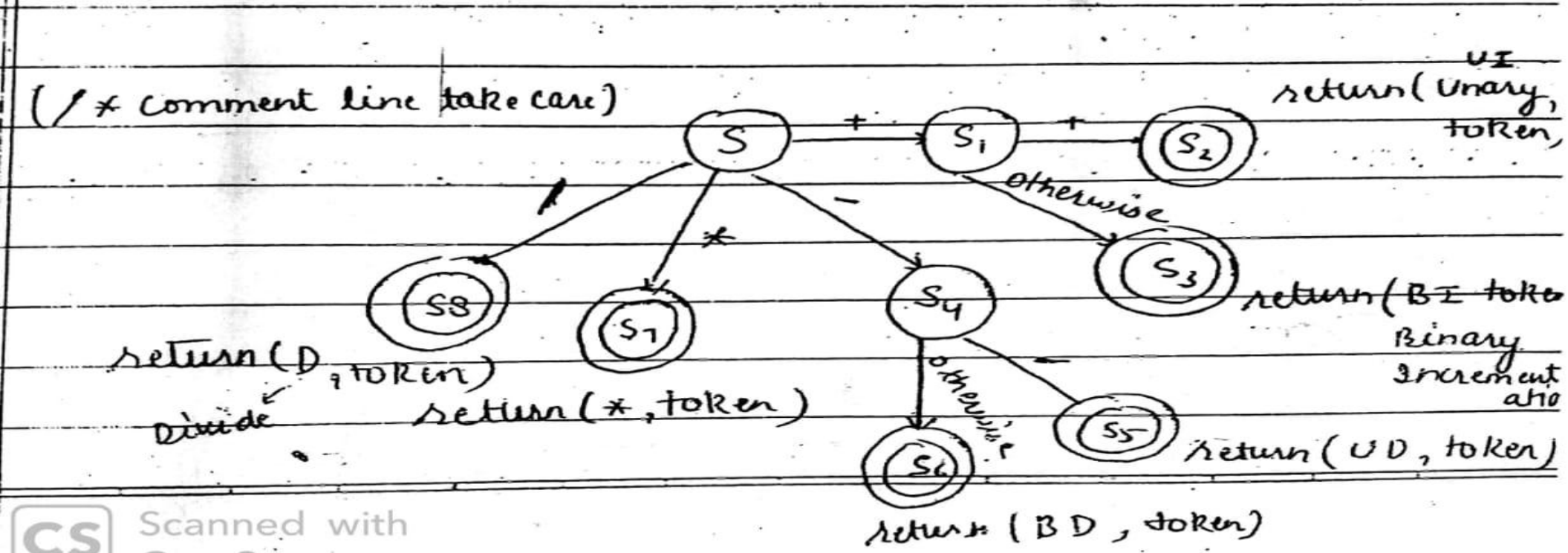
(B) 26

(C) 10

(D) 21



<>==<=>=!=<==>



++-+-+/*-+//*

Introduction

- Languages typically consist of an infinite number of strings, making it impractical to list all strings explicitly. Similar to automata, grammar provides a mathematical model to represent a language. It serves as a generator for all valid strings within a language. E.g. L = {a, aa, ab, aaa, aab, aba,…….}

Grammar

- **Start with the Start Symbol**: Begin with a string that contains only the designated start symbol.
- **Apply Production Rules**: Use the grammar's production rules in any order to replace symbols.
- **Reach Terminal Form**: Continue applying rules until a string is formed that contains no start or non-terminal symbols. Such strings are part of the language.
- The intermediate stages during this process are called **sentential forms**. The collection of all distinct strings generated by this process defines the language of the grammar.
  - $L(G) = \{w \mid w \in \sum^*, S \rightarrow^* W\}$
  - $\rightarrow^*$(reflexive, transitive closure)

Formal Grammar

- A Formal Grammar is defined by 4-tuple (V_N, $\sum$, P, S), where
- V_N is a finite nonempty set whose elements are called variables,
- $\sum$ is a finite nonempty set whose elements are called terminals, $V_N \cap \sum = \Phi$.
- S is a special variable (i.e., an element of V_N ($S \in V_n$)) 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 $V_N \cup \sum$.
  - α has at least one symbol from V_N, the element of P are called productions or production rules or rewriting rules.
  - For a formal valid production,

α -> β

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

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

V_N	$\sum$	P	S
S,A	a,b	S -> aA	S
		A -> aA / bA / $\in$	

Some points to note about productions

- **No Reverse Substitution**:
  - If there is a production rule like S⟶AB, we can replace S with AB, but the reverse—replacing AB with S—is not allowed.

- **Equivalence of Grammars**
  - Two grammars, G_1 and G_2, are considered equivalent if they generate the same language. That is, $L(G_1)=L(G_2)$.

- **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

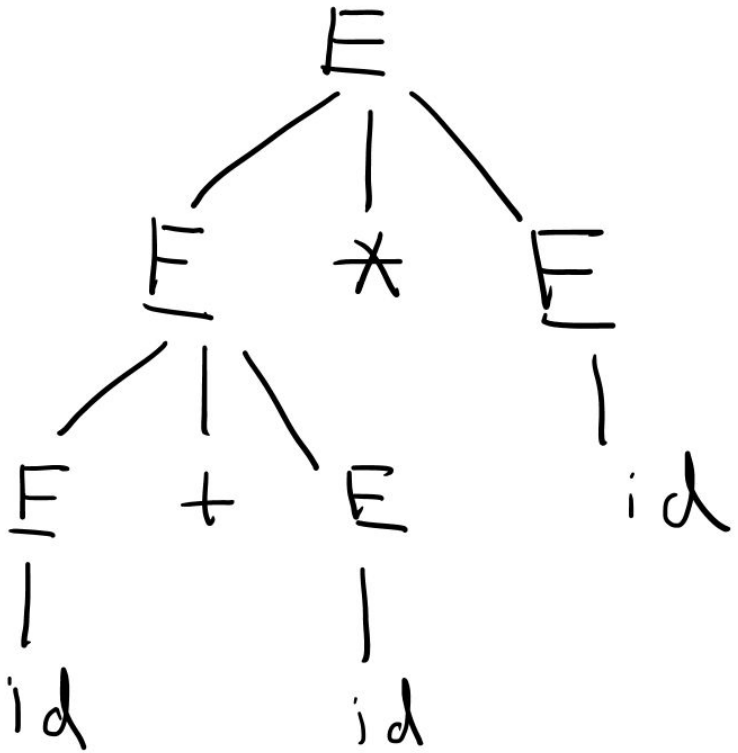
$$\alpha \rightarrow \beta$$
$$\alpha \in V_n \qquad |\alpha| = 1$$
$$\beta \in \{\Sigma \cup V_n\}^*$$

- A grammar is called a type 2 grammar if it contains only type 2 productions.
- Eg ALGOL 60, PASCAL

Chomsky Classification of Grammar
Type 0 Grammar
Type 1 Grammar
Type 2 Grammar
Type 3 Grammar

- **Derivation:** - The process of deriving a string is known as derivation.
- **Derivation/ Syntax/ Parse Tree**: - The graphical representation of derivation is known as derivation tree.
- **<u>Sentential form</u>:** - Intermediate step involve in the derivation is known as sentential form.

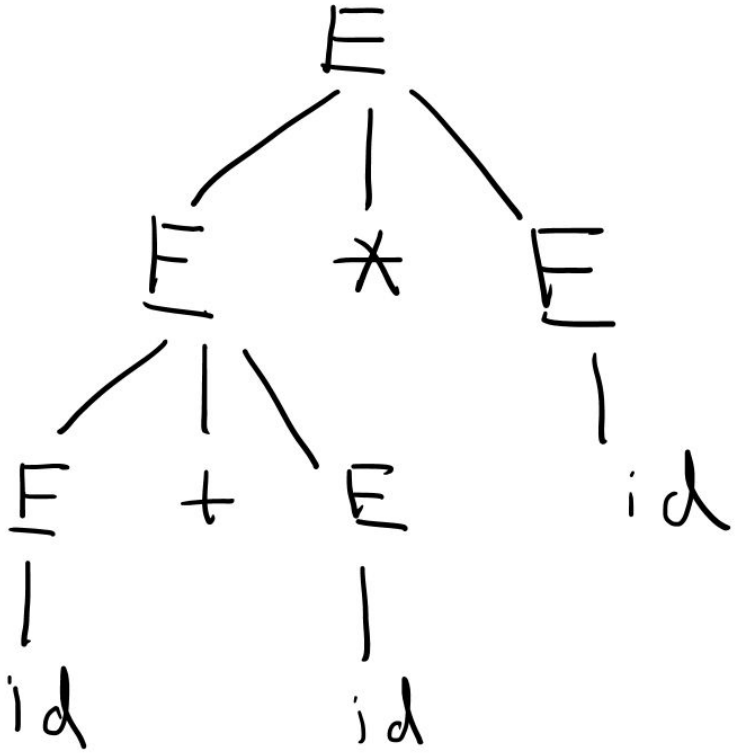
E -> E + E / E * E / E = E / id



Sentential Form
E
E*E
E+E*E
ID+E*E
ID+ID*E
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 is 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 is known as RMDT (right most derivation tree)

	LMD
E -> E + E	E
E-> E * E	E*E
E-> E = E	E+E*E
E-> id	ID+E*E
E-> id	ID+ID*E
E-> id	ID+ID*ID



	RMD
E -> E + E	E
E-> E * E	E+E
E-> E = E	E+E*E
E -> id	E+E*ID
E-> id	E+ID*ID
E-> id	ID+ID*ID

Q. Consider the following context-free grammar G, where S, A, and B are the variables (non-terminals), a and b are the terminal symbols, S is the start variable, and the rules of G are described as:

$$S \rightarrow aaB \mid Abb$$

$$A \rightarrow a \mid aA$$

$$B \rightarrow b \mid bB$$

Which ONE of the languages $L(G)$ is accepted by G? **(Gate 2025)**

A) $L(G) = \{a^2b^n \mid n \geq 1\} \cup \{a^nb^2 \mid n \geq 1\}$

B) $L(G) = \{a^nb^{2n} \mid n \geq 1\} \cup \{a^{2n}b^n \mid n \geq 1\}$

C) $L(G) = \{a^nb^n \mid n \geq 1\}$

D) $L(G) = \{a^{2n}b^{2n} \mid n \geq 1\}$

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?

$A \rightarrow A\alpha / \beta$ **$A \rightarrow A\alpha / \beta_1 / \beta_2$** **$A \rightarrow A\alpha / \beta_1 / \beta_2 \text{-------} / \beta_n$**

$A \rightarrow A\alpha_1 / A\alpha_2 / \beta$

$A \rightarrow A\alpha_1 / A\alpha_2 /\text{---------}A\alpha_n / \beta$

$$A \rightarrow A\alpha_1 / A\alpha_2 / \text{----------} A\alpha_n / \beta_1 / \beta_2 \text{ -------} / \beta_m$$

A -> A(A) / α

A -> A a A / b

S -> S a b / c

S -> S S S / 0

S -> S 0 S 1 / 0 / 1

S -> A a B
A -> Aa / bB / a
B -> bB / c

E -> E + T / T
T -> T * F / F
F -> (E) / id

E -> E + E / E * E / (E) / id

R -> R * / R R / (R) / id

S -> (L) / a
L -> L , S / S

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

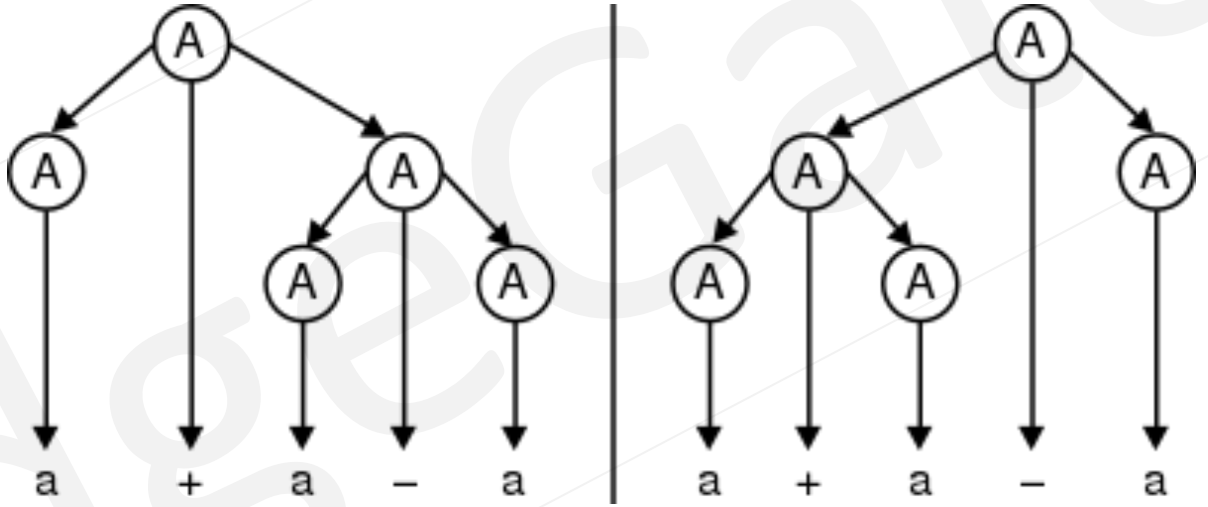
(B)
$$S \rightarrow Ab \mid Bb \mid c$$
$$A \rightarrow Bd \mid \varepsilon$$
$$B \rightarrow e$$

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

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

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

A -> A+A / A-A/ a



Unambiguous grammar: - The CFG is said to be unambiguous if there exist only one parse tree for every 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,
- $\{a^n b^m c^m d^n\} U \{a^n b^n c^m d^m\}$
- Grammar which is both left and right recursive is always ambiguous, but the ambiguous grammar need not be both left and right recursive.

S -> Sa / aS / ∈

S -> SS / 0 / 1

S -> SaS / b

S -> (L) / a
L -> L , S / S

S -> SAB / ∈

A -> AaB / a

B -> AS / b

S -> SaSbS / ∈

S -> aSa / bSb / ∈

S -> aSb / bSa / ∈

S -> aAB

A -> aA / BaA / a

B -> Aa / b

A -> A(A) / a

S -> AA

A -> aA / b

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 Consider the following statements about the context free grammar **(GATE-2006) (1 Marks)**

$$G = \{S \rightarrow SS, S \rightarrow ab, S \rightarrow ba, S \rightarrow \in\}$$

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

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.

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

$$S \longrightarrow a\,S\,b \,/\, SS \,/\, \varepsilon$$

Which of the following statements is true?

(A) G is not ambiguous

(B) There exist x, y, $\in$ L (G) such that xy $\notin$ 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)

Non-Deterministic Grammar

- A grammar with a common prefix in its production rules is called **Non-Deterministic Grammar**.
  - A □ $\alpha\beta_1 / \alpha\beta_2$

- **Problem with Backtracking**:
  - Non-deterministic grammar requires extensive **backtracking** to explore different possibilities, making the process slow and time-consuming.

- **Solution**:
  - To eliminate backtracking, **common prefixes** must be removed by converting the non-deterministic grammar into **deterministic grammar**.

- **<u>Left Factoring</u>**: - The process of conversion of Non-Deterministic grammar into deterministic grammar is known as Left-Factoring.

- $A \rightarrow \alpha\beta_1 / \alpha\beta_2$
  - $A \rightarrow \alpha\beta$
  - $A \rightarrow \beta_1 / \beta_2$

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

S -> aSb / abS / ab

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

S -> ab / abc / abcd / b

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

S -> aAb / aABc / aABcd / aA / a

Simplification or Minimization of CFG

- The reason we simplify CFG to make it more efficient and compiler friendly.
- 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.

S -> AbB

A -> a / ∈

B -> b / ∈

S -> AB

A -> a / ∈

B -> b / ∈

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.

S -> Aa

A -> a / B

B -> d

S -> aAb

A -> B / a

B -> C / b

C -> D / c

D -> d

S -> aSb / ∈

Removal of useless symbols

- The variables which are not involved in the derivation of any 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.

S -> aAB
A -> a
B -> b
C -> d

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

S -> aA / aB
A -> b

S -> aAB / bA / aC

A -> aB / b

B -> 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 →AB | a

A → 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

Chomsky Normal Form

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

$$A \rightarrow BC / a$$

$$B, C \in V_n$$

$$a \in \Sigma$$

S -> aSb / ab

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 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. Let G=(V,∑,S,P) be a context-free grammar in chomsky Normal Form with ∑={a,b,c} and v containing 10 variable symbols including the start symbol S. The string w= $a^{30}b^{30}c^{30}$ is derivable from S. The number of steps (application of rules)in the derivation S⟶*w is_____**(Gate 2024,CS) (2 Marks) (NAT)**

Greiback Normal Form

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

$$A \rightarrow a\alpha$$

$$A \in V_n$$

$$a \in \Sigma$$

$$\alpha \in V_n^*$$

- **Sheila Adele Greibach** (born 6 October 1939 in New York City) is a researcher in formal languages in computing, automata, compiler theory and computer science.

S -> aSb / ab

S -> aAb / bB

A -> a / b

B -> b

- if CFG is in GNF, 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 $A \rightarrow aX$ 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

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?

S □ aAB / Aa
A □ a

S □ aAB
A □ a / b

S □ aAB / aB
A □ aBb
B □ aA

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

S □ SS / AB
A □ BC / a
B □ CC / b

S □ AB
A □ B / a

S □ AB
A □ BC / a
B □ CC / b
C □ AB

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

S □ AB / BB

A □ BA / AS / b

B □ AA / SB / a

w_1 = aba

w_2 = abaab

w_3 = abababba

CYK algorithm

- In computer science, the **Cocke–Younger–Kasami algorithm** (alternatively called **CYK**, or **CKY**) is a parsing algorithm for context-free grammars, named after its inventors, John Cocke, Daniel Younger and Tadao Kasami. It employs bottom-up parsing and dynamic programming.

- The standard version of CYK operates only on context-free grammars given in Chomsky normal form (CNF). However any context-free grammar may be transformed to a CNF grammar expressing the same language (Sipser 1997).

- The importance of the CYK algorithm stems from its high efficiency in certain situations. Using Big O notation, the worst case running time of CYK is $O(n^3 . |G|)$, Where n is the length of the parsed string and |G| is the size of the CNF grammar G.

- This makes it one of the most efficient parsing algorithms in terms of worst-case asymptotic complexity, although other algorithms exist with better average running time in many practical scenarios.

Backus-Naur Form (BNF)

$A \rightarrow \alpha_1$

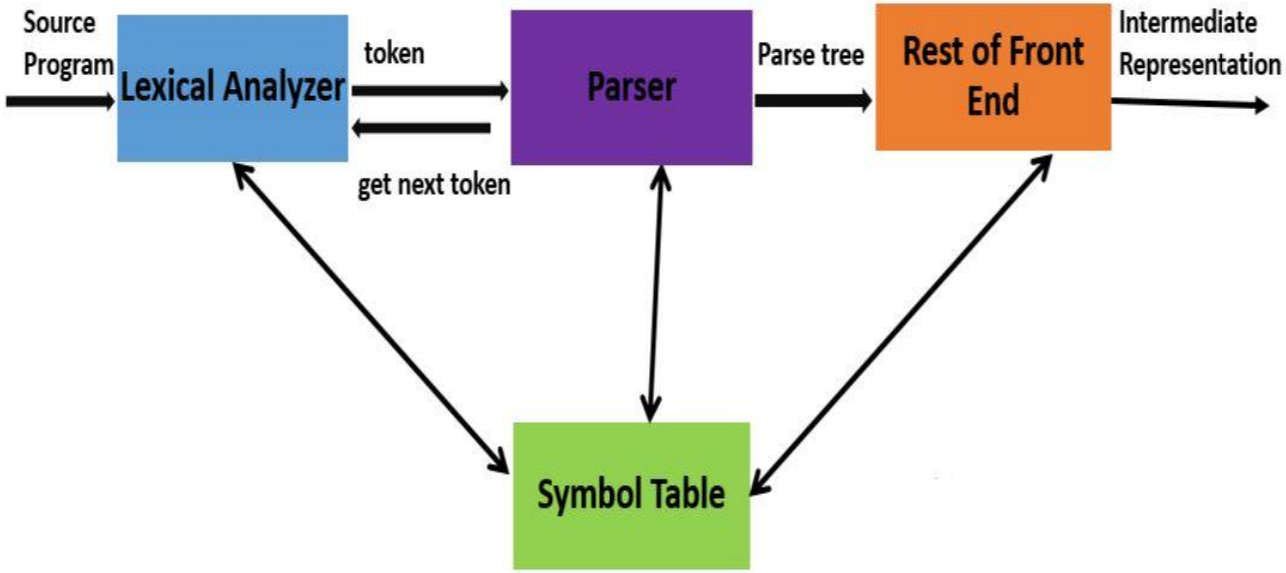
$A \rightarrow \alpha_2$

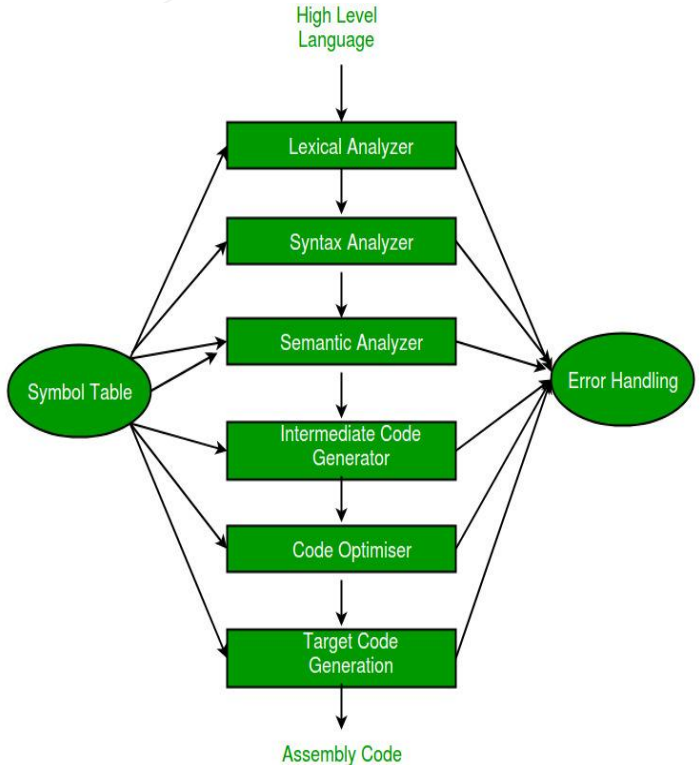
$A \rightarrow \alpha_3$

$A \rightarrow \alpha_1 / \alpha_2 / \alpha_3$

Syntax analysis

- **Input and Output**:
  - **Input**: A stream of tokens.
  - **Output**: A syntax tree (also called a parse tree or derivation tree).
- **Process**:
  - The process of constructing a syntax tree is known as **parsing**.
- **Validation of Input Strings**:
  - If a derivation tree can be constructed for the input string, it is considered **syntactically correct**.
  - If no derivation tree can be generated, it indicates **grammatical errors** in the input string.





S □ id = E

E □ E + T / T

T □ T * F / F

F □ id / num

$id_1 = id_2 + id_3 * 60$

1. *program* → *declarationList*
2. *declarationList* → *declarationList declaration* | *declaration*
3. *declaration* → *varDeclaration* | *funDeclaration*

---

4. *varDeclaration* → *typeSpecifier varDeclList* ;
5. *scopedVarDeclaration* → *scopedTypeSpecifier varDeclList* ;
6. *varDeclList* → *varDeclList* , *varDeclInitialize* | *varDeclInitialize*
7. *varDeclInitialize* → *varDeclId* | *varDeclId* : *simpleExpression*
8. *varDeclId* → **ID** | **ID** [**NUMCONST**]
9. *scopedTypeSpecifier* → **static** *typeSpecifier* | *typeSpecifier*
10. *typeSpecifier* → **int** | **bool** | **char**

---

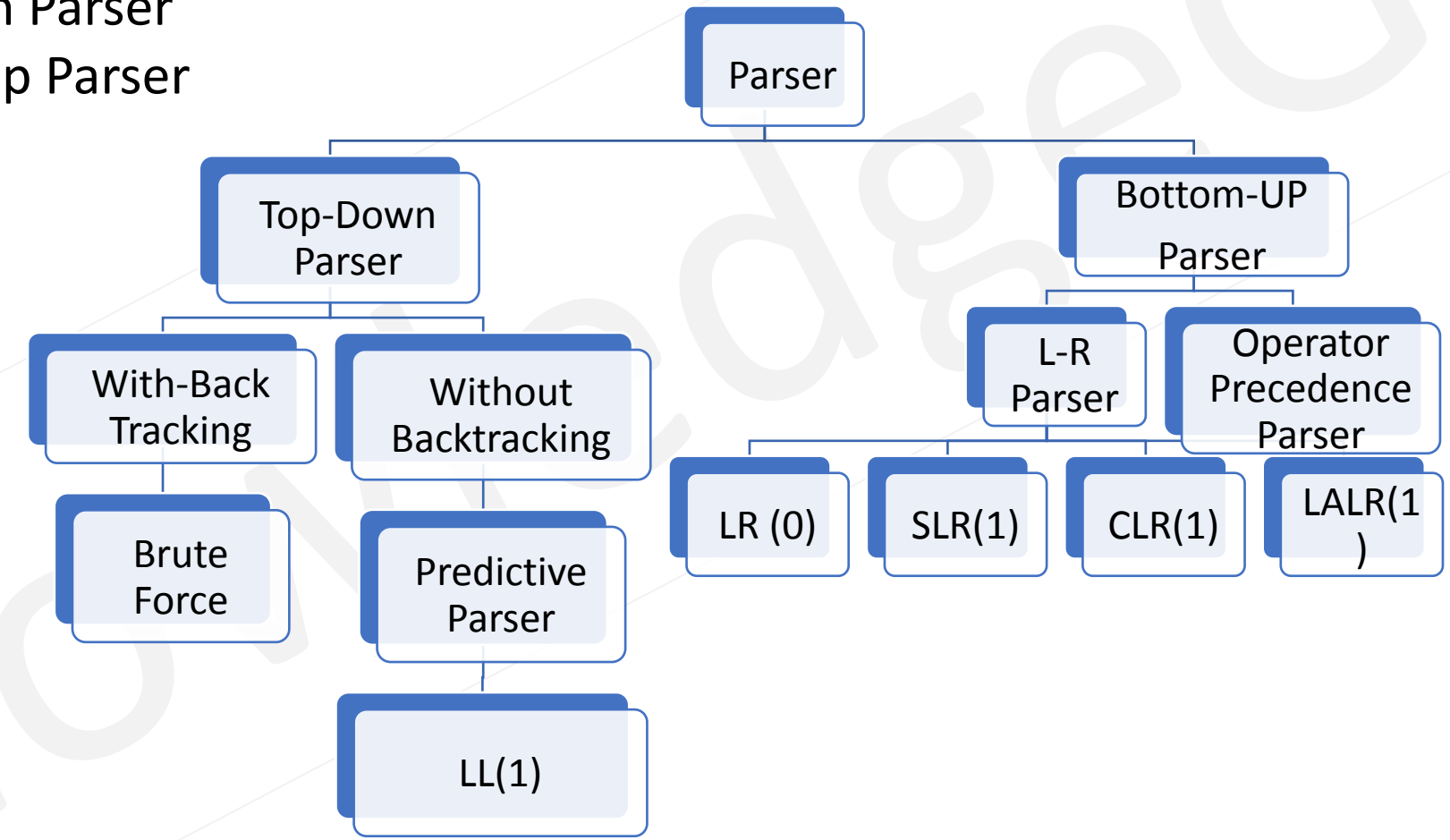
11. *funDeclaration* → *typeSpecifier* **ID** (*params*) *statement* | **ID** (*params*) *statement*
12. *params* → *paramList* | ϵ
13. *paramList* → *paramList* ; *paramTypeList* | *paramTypeList*
14. *paramTypeList* → *typeSpecifier paramIdList*
15. *paramIdList* → *paramIdList* , *paramId* | *paramId*
16. *paramId* → **ID** | **ID** [ ]

---

17. *statement* → *expressionStmt* | *compoundStmt* | *selectionStmt* | *iterationStmt* | *returnStmt* | *breakStmt*
18. *expressionStmt* → *expression* ; | ;
19. *compoundStmt* → { *localDeclarations statementList* }
20. *localDeclarations* → *localDeclarations scopedVarDeclaration* | ϵ
21. *statementList* → *statementList statement* | ϵ
22. *elsifList* → *elsifList* **elsif** *simpleExpression* **then** *statement* | ϵ
23. *selectionStmt* → **if** *simpleExpression* **then** *statement elsifList* | **if** *simpleExpression* **then** *statement elsifList* **else** *statement*
24. *iterationRange* → **ID** = *simpleExpression* .. *simpleExpression* | **ID** = *simpleExpression* .. *simpleExpression* : *simpleExpression*
25. *iterationStmt* → **while** *simpleExpression* **do** *statement* | **loop forever** *statement* | **loop** *iterationRange* **do** *statement*
26. *returnStmt* → **return** ; | **return** *expression* ;
27. *breakStmt* → **break** ;
28. *expression* → *mutable* = *expression* | *mutable* += *expression* | *mutable* −= *expression* | *mutable* *= *expression* | *mutable* /= *expression* | *mutable* ++ | *mutable* −− | *simpleExpression*
29. *simpleExpression* → *simpleExpression* **or** *andExpression* | *andExpression*
30. *andExpression* → *andExpression* **and** *unaryRelExpression* | *unaryRelExpression*
31. *unaryRelExpression* → **not** *unaryRelExpression* | *relExpression*
32. *relExpression* → *sumExpression relop sumExpression* | *sumExpression*
33. *relop* → <= | < | > | >= | == | !=
34. *sumExpression* → *sumExpression sumop mulExpression* | *mulExpression*
35. *sumop* → + | −
36. *mulExpression* → *mulExpression mulop unaryExpression* | *unaryExpression*
37. *mulop* → * | / | %
38. *unaryExpression* → *unaryop unaryExpression* | *factor*
39. *unaryop* → − | * | ?
40. *factor* → *immutable* | *mutable*
41. *mutable* → **ID** | *mutable* [*expression*]
42. *immutable* → (*expression*) | *call* | *constant*
43. *call* → **ID** (*args*)
44. *args* → *argList* | ϵ
45. *argList* → *argList* , *expression* | *expression*
46. *constant* → **NUMCONST** | **CHARCONST** | **STRINGCONST** | **true** | **false**

Classification of parser

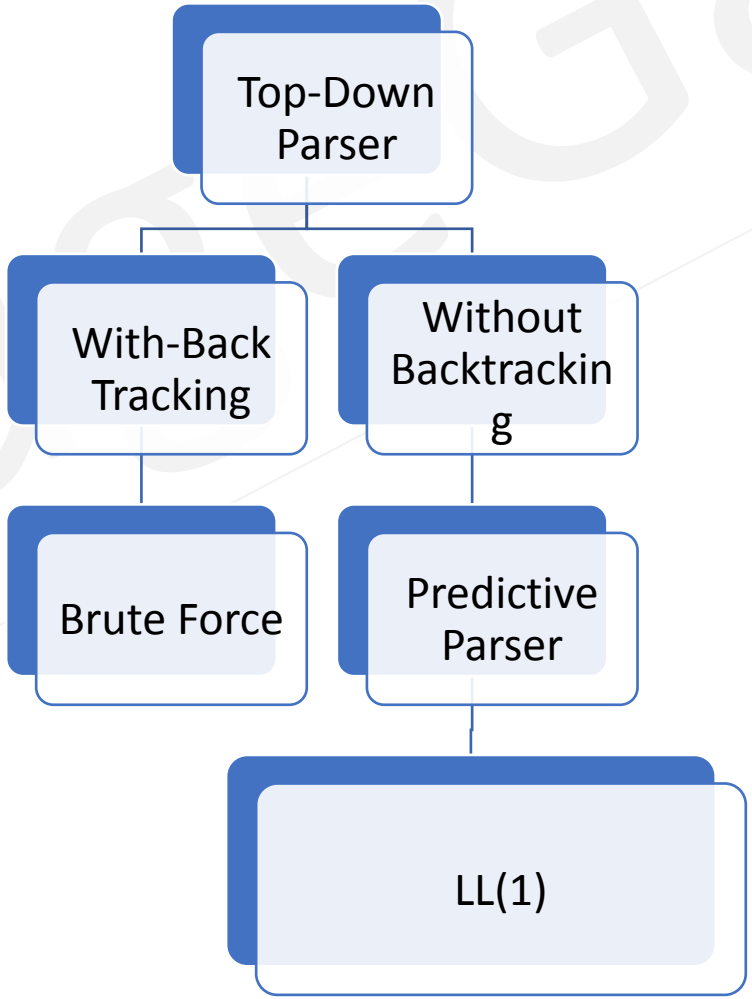
- The program which perform parsing is known as parser or syntax analyzer.
- There are two types of parser
  - Top-Down Parser
  - Bottom Up Parser



Top down parsing

- 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.

A -> aA / ∈



- **Key Features**:
  - Uses **left-most derivation**.
  - Can only be constructed if the grammar is:
    - Free from **left recursion**.
    - Free from **ambiguity**.
  - Can handle both **left factored** and **non-left factored** grammars.
- **Handling Grammar Types**:
  - For **non-deterministic grammar** (common prefixes), brute-force techniques are used.
  - For **deterministic grammar**, a **predictive parser** is applied for efficient parsing.
- **Performance Considerations**:
  - Parsing is slow and performance is low if the grammar has high complexity.
  - **Worst-case time complexity** of TDP can reach O(n4)O(n^4)O(n4), except when using brute-force methods.

Brute force technique

- **Process**:
  - When expanding a **non-terminal** for the first time, choose the **first alternative** and compare it with the input string.
  - If it doesn't match, try the **second alternative**, then the **third**, and so on, checking each alternative against the input string.
  - Continue this process until a match is found or all alternatives are exhausted.
- **Outcome**:
  - If at least one alternative matches, **parsing is successful**.
  - If no match occurs for any alternative, **parsing fails**.

S -> cAd

A -> ab / a

w_1 = cad **w_2 = cada**

S -> aAc / aB

A -> b / c

B -> ccd / ddc

w = addc

- **Challenges with Brute Force**:
  - Requires extensive **backtracking**, making it highly inefficient with a time complexity of $O(2^n)$.
  - Backtracking is computationally **costly** and significantly **reduces performance**.
  - Debugging issues in such parsers is extremely **difficult**.

- First(α) is a set of all terminals that may be in beginning in any sentential form, derived from α
- FIRST(α) is calculated for both Terminals and Non-Terminals
- **if** α is a terminal, then
  - **First(α) = {α}**
- **if** α is a string of terminal e.g. abc
  - **First(**abc**) = {a}**
- **if** α is a non - terminal, defined by α □ ε, then
  - **First(α) = {ε}**
- **if** α is a non - terminal, defined by α □ β, β ∈ T
  - **First(α) = {β}**
- **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_3)] -** ∈ **iff** X_1 □ ∈ && X_2 □ ∈ && **iff** X_3 □! ∈
  - **First(α) = First(X_1) U First(X_2) U First(X_2)** **iff** X_1 □ ∈ && X_2 □ ∈ && **iff** X_3 □ ∈

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

S -> a / b / ε

First(S) =

Follow(S) =

S -> aA / bB

A -> ε

B -> ε

First(S) =

First(A) =

First(B) =

Follow(S) =

Follow(A) =

Follow(B) =

S -> aAb / Ba
A -> aA / b
B -> c / d

First(S) =
First(A) =
First(B) =
Follow(S) =
Follow(A) =
Follow(B) =

S -> AaB / BA
A -> a / b
B -> d / e

First(S) =
First(A) =
First(B) =
Follow(S) =
Follow(A) =
Follow(B) =

S -> AB

A -> a / ε

B -> b / ε

First(S) =

First(A) =

First(B) =

Follow(S) =

Follow(A) =

Follow(B) =

S -> AaB / Bb

A -> bA / ε

B -> cB / ε

First(S) =

First(A) =

First(B) =

Follow(S) =

Follow(A) =

Follow(B) =

S -> AaB
A -> b / ε
B -> c

First(S) =
First(A) =
First(B) =
Follow(S) =
Follow(A) =
Follow(B) =

S -> aAB / BA
A -> BA / ε
B -> AB / a / ε

First(S) =
First(A) =
First(B) =
Follow(S) =
Follow(A) =
Follow(B) =

S -> ABCDE

A -> a / ε

B -> b / ε

C -> c / ε

D -> d

E -> e / ε

First(S) =

First(A) =

First(B) =

First(C) =

First(D) =

First(E) =

Follow(S) =

Follow(A) =

Follow(B) =

Follow(C) =

Follow(D) =

Follow(E) =

S -> AA

A -> aA

A -> b

First(S) =

First(A) =

Follow(S) =

Follow(A) =

S -> (L) / a

L -> SL’

L’ -> ,SL’ / ε

First(S) =

First(L) =

First(L’) =

Follow(S) =

Follow(L) =

Follow(L’) =

E -> TE’

E’ -> +TE’ / ε

T -> FT’

T’ -> *FT’ / ε

F -> (E) / id

First(E) =

First(E’) =

First(T) =

First(T’) =

First(F) =

Follow(E) =

Follow(E’) =

Follow(T) =

Follow(T’) =

Follow(F) =

A -> aA’

A’ -> (A)A’ / ε

First(A) =

First(A’) =

Follow(A) =

Follow(A’) =

A -> (A)A’ / aA’

A’ -> AA’ / ε

First(A) =

First(A’) =

Follow(A) =

Follow(A’) =

S -> aBDh

B -> cC

C -> bC / ε

D -> EF

E -> g / ε

F -> f / ε

First(S) =

First(B) =

First(C) =

First(D) =

First(E) =

First(F) =

Follow(S) =

Follow(B) =

Follow(C) =

Follow(D) =

Follow(E) =

Follow(F) =

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) - $\in$

S -> ε

First(S) =

Follow(S) =

S -> aA

A -> bA / ε

First(S) =

First(A) =

Follow(S) =

Follow(A) =

S → AaBb / BbAa

A → ε

B → ε

First(S) =

First(A) =

First(B) =

Follow(S) =

Follow(A) =

Follow(B) =

E → E+T / T

T → T*F / F

F → (E) / id

First(E) =

First(T) =

First(F) =

Follow(E) =

Follow(T) =

Follow(F) =

$S \rightarrow aAb$

$A \rightarrow Ba \mid b$

$B \rightarrow d$

First(S) =

First(A) =

First(B) =

Follow(S) =

Follow(A) =

Follow(B) =

$S \rightarrow S0S1 \mid \varepsilon$

First(S) =

Follow(S) =

S → AaAb / BaBb

A → ε

B → ε

First(S) =

First(A) =

First(B) =

Follow(S) =

Follow(A) =

Follow(B) =

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)**

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, ∉}

Q. Which of the following statement(s) is/are TRUE while computing First and Follow during top down parsing by a compiler? **(Gate 2025)**

A) For a production $A \rightarrow \epsilon$, ϵ will be added to $First(A)$.

B) If there is any input right end marker, it will be added to $First(S)$, where S is the start symbol.

C) For a production $A \rightarrow \epsilon$, ϵ will be added to $Follow(A)$.

D) If there is any input right end marker, it will be added to $Follow(S)$, where S is the start symbol.

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

E □ TE'

E' □ +TE' / ∈

T □ FT'

T' □ *FT' / ∈

F □ (E) / id

w = id+id*id

Stack	i/p	Action
$	id+id*id$	Push E
$ E	id+id*id$	Use Production E□TE' POP E and Push E'T
$ E'T	id+id*id$	Use Production T□FT' POP T and Push T'F
$ E'T'F	id+id*id$	Use Production F□id POP F and Push id
$ E'T'id	id+id*id$	Match Pop id and Increment Look Ahead Pointer
-	-	-
-	-	-
-	-	-
-	-	-
$	$	Accepted

- 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 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

E -> TE’

E’ -> +TE’ / ∈

T -> FT’

T’ -> *FT’ / ∈

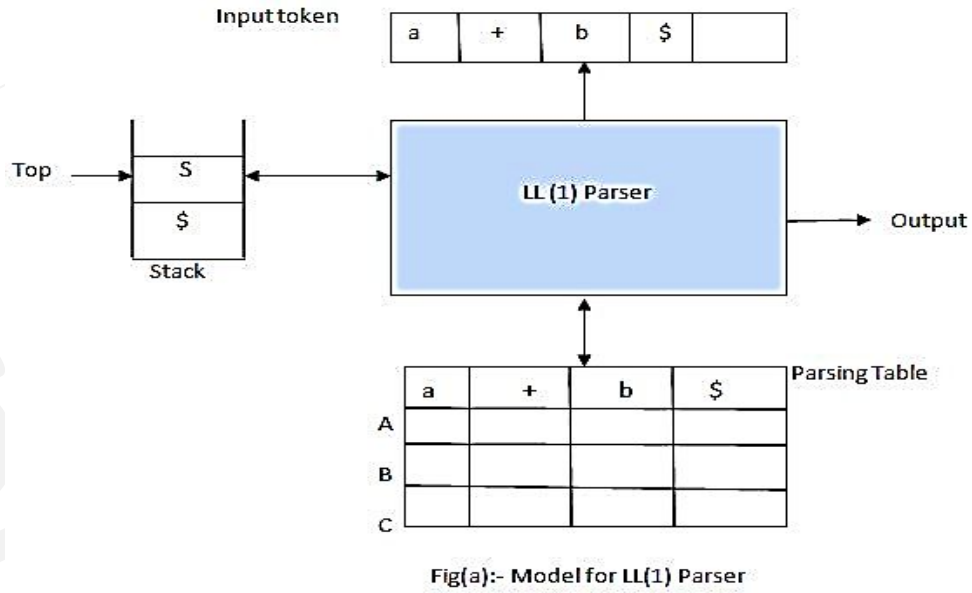
F -> (E) / id

id	+	id	*	id	$

	+	*	(	)	id	$
E						
E’						
T						
T’						
F						

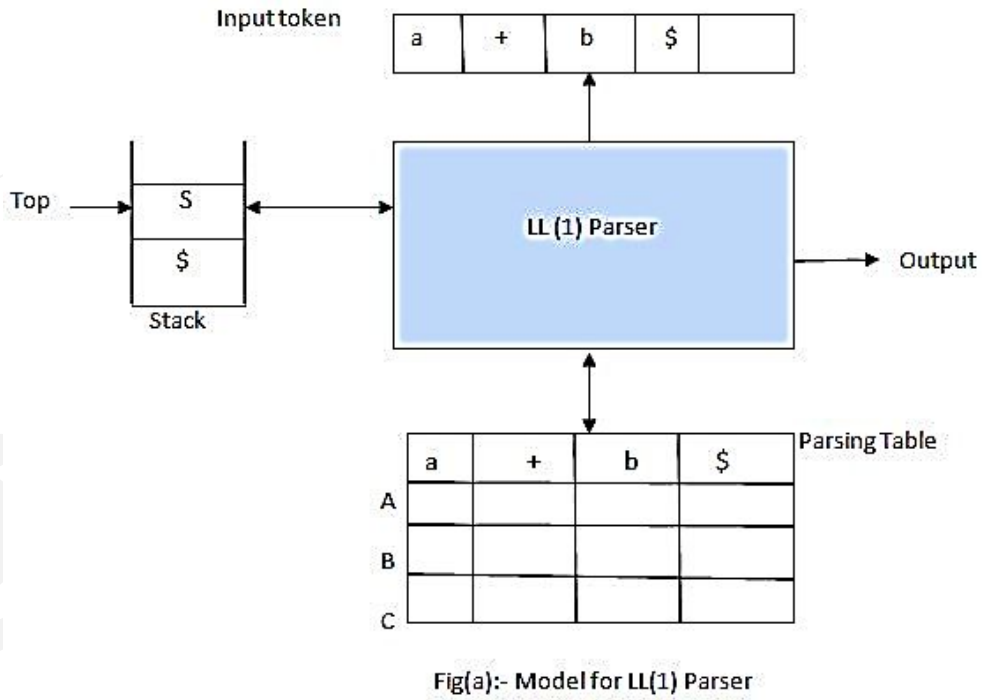
Block-Diagram of LL(1) Parser

- An LL(1) parser consists of three main components:
- **Input Buffer**:
  - Divided into a finite number of cells, each holding one input symbol.
  - Holds the input string at any given time.
  - The **tape header** points to one **lookahead symbol** at a time and moves to the next cell to the right after parsing the current symbol.
  - The end of the string is marked with **$**.
- **Parse Stack**:
  - Stores **grammar symbols**.
  - Symbols are either **pushed** into or **popped** out of the stack based on matches with the lookahead symbol:
    - If the **top stack symbol matches** the lookahead symbol, it is **popped**.
    - If it does not match, the symbol is **pushed**.
- **Parse Table**:
  - A two-dimensional array of size m×nm \times nm×n, where:
    - m = number of non-terminals.
    - n= number of terminals +1 (for $).
  - Contains the **production rules** used to construct the parse tree for the input string.



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

- A **Pushdown Automaton (PDA)** alone cannot simulate an **LL(1) parser** because:
- The PDA stack can only store **grammar symbols**, not the **production rules** required to construct the parse tree.
- To enable a PDA to simulate an LL(1) parser, a **parse table** must be attached to the PDA.



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

- **LL(1) Parser Table Construction**
- To construct an LL(1) parsing table M[A,a], follow these steps for every production A→αA :
  - Add A→αA to M[A,a] for every terminal a in **First(α)**.
  - If **First(α)** contains ε, then add A→ε to M[A,b] for every symbol b in **Follow(A)**.

- **LL(1) Parsing Process**
- **Initialize**:
  - Push the start symbol into the stack.
- **Comparison**:
  - Compare the topmost stack symbol x with the lookahead symbol a.
    - If x=a=$: Parsing is **successful**.
    - If x=a≠$: Pop the grammar symbol and increment the input pointer.
    - If x≠a≠$ and M[x,a] contains a production x→abc:
      - Replace x with abc in reverse order and continue.
- **Output**:
  - Output the production used to expand non-terminals during the parsing process.

- **Definition**:
  - A grammar is considered **LL(1)** if:
    - It is **free from left recursion**.
    - It is **free from ambiguity**.
    - It is **left-factored**.
- **LL(1) Parse Table**:
  - If the parse table does **not contain multiple entries in the same cell**, the grammar qualifies as an **LL(1) Grammar**.

Short Cut Techniques for LL(1)

- A Grammar without ε is LL(1), if
  - for every production of the A → $\alpha_1 / \alpha_2 / \alpha_3 / \text{-----} / \alpha_n$, the set First($\alpha_1$), First($\alpha_2$), First($\alpha_3$),……., First($\alpha_n$) are mutually disjoint.
  - **First(α_1) ∩ First(α_2) ∩ First(α_3) ∩……∩ First(α_n) = ϕ**

- A Grammar with is LL(1), if
  - for every production of the A → α / ε
  - **First(α) ∩ Follow(A) = ϕ**

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

S □ AB
A □ a
B □ b

S □ aA / bB
A □ Bb / a
B □ bB / c

S □ aAbB
A □ a / ε
B □ b / ε

S □ ACB / CbB / Ba

A □ da / BC

B □ g / ε

C □ h / ε

- Avg time complexity is $O(n^4)$ for Left-recursive grammar and $O(n^3)$ for non Left-recursive grammar

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

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 E_1, E_2, and E_3. $ indicates end of input, and, | separates alternate right hand sides of productions? **(GATE-2012) (2 Marks)**

$S \rightarrow a\,A\,b\,B \mid b\,A\,a\,B \mid \varepsilon$
$A \rightarrow S$
$B \rightarrow S$

	a	b	$
S	**E1**	**E2**	$S \rightarrow \varepsilon$
A	$A \rightarrow S$	$A \rightarrow S$	error
B	$B \rightarrow S$	$B \rightarrow S$	**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}

Q Consider the date same as above question. The appropriate entries for E_1, E_2, and E_3 are **(GATE-2012) (2 Marks)**

$S \rightarrow a\,A\,b\,B \mid b\,A\,a\,B \mid \varepsilon$
$A \rightarrow S$
$B \rightarrow S$

	a	b	$
S	**E1**	**E2**	$S \rightarrow \varepsilon$
A	$A \rightarrow S$	$A \rightarrow S$	error
B	$B \rightarrow S$	$B \rightarrow S$	**E3**

(A) E1: $S \rightarrow aAbB$, $A \rightarrow S$
E2: $S \rightarrow bAaB$, $B \rightarrow S$
E3: $B \rightarrow S$

(B) E1: $S \rightarrow aAbB$, $S \rightarrow \varepsilon$
E2: $S \rightarrow bAaB$, $S \rightarrow \varepsilon$
E3: $S \rightarrow \varepsilon$

(C) E1: $S \rightarrow aAbB$, $S \rightarrow \varepsilon$
E2: $S \rightarrow bAaB$, $S \rightarrow \varepsilon$
E3: $B \rightarrow S$

(D) E1: $A \rightarrow S$, $S \rightarrow \varepsilon$
E2: $B \rightarrow S$, $S \rightarrow \varepsilon$
E3: $B \rightarrow S$

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)**

$S \rightarrow iCtSS_1 | a$

$S_1 \rightarrow eS | \epsilon$

$C \rightarrow b$

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.

Q Consider the following grammar:

$S \rightarrow FR$

$R \rightarrow *S \mid \varepsilon$

$F \rightarrow id$

In the predictive parser table, M, of the grammar the entries M[S, id] and M[R, $] respectively. **(GATE-2006) (2 Marks)**

(A) {$S \rightarrow FR$} and {$R \rightarrow \varepsilon$ }

(B) {$S \rightarrow FR$} and { }

(C) {$S \rightarrow FR$} and {$R \rightarrow *S$}

(D) {$F \rightarrow id$} and {$R \rightarrow \varepsilon$}

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

Q Consider the following context-free grammar where the set of terminals is {a,b,c,d,f}.

$$S \rightarrow daT \mid Rf$$
$$T \rightarrow aS \mid baT \mid \epsilon$$
$$R \rightarrow caTR \mid \epsilon$$

The following is a partially-filled LL(1) parsing table.

	a	b	c	d	f	$\$$
S			①	$S \rightarrow daT$	②	
T	$T \rightarrow aS$	$T \rightarrow baT$	③		$T \rightarrow \epsilon$	④
R			$R \rightarrow caTR$		$R \rightarrow \epsilon$	

Which one of the following choices represents the correct combination for the numbered cells in the parsing table ("blank" denotes that the corresponding cell is empty)? **(GATE 2021) (2 MARKS)**

A. ① $S \rightarrow Rf$ ② $S \rightarrow Rf$ ③ $T \rightarrow \epsilon$ ④ $T \rightarrow \epsilon$

B. ① blank ② $S \rightarrow Rf$ ③ $T \rightarrow \epsilon$ ④ $T \rightarrow \epsilon$

C. ① $S \rightarrow Rf$ ② blank ③ blank ④ $T \rightarrow \epsilon$

D. ① blank ② $S \rightarrow Rf$ ③ blank ④ blank

Q.Consider the following grammar G, with S as the start symbol, The grammar G has three incomplete production devoted by (1),(2), and (3): The set of terminals is {a,b,c,d,f}.The FIRST and FOLLOW sets of the different non-terminals are as follows. **(Gate 2024,CS) (2 Marks) (MCQ)**

FIRST(S)= {c,d,f}, FIRST(T)={a,b,∈}, FIRST(R)={c, ∈}

FOLLOW(S) = FOLLOW(T) = {c,f,$}, FOLLOW(R)={f}

Which one of the following options CORRECTLY fills in the incomplete production?

$$S \rightarrow daT \mid \underline{\quad (1) \quad}$$
$$T \rightarrow aS \mid bT \mid \underline{\quad (2) \quad}$$
$$R \rightarrow \underline{\quad (3) \quad} \mid \epsilon$$

(a) (1) S⟶Rf (2)T⟶∈ (3)R⟶cTR

(b) (1) S⟶fR (2)T⟶∈ (3)R⟶cTR

(c) (1) S⟶fR (2)T⟶cT (3)R⟶cR

(d) (1) S⟶Rf (2)T⟶cT (3)R⟶cR

Q. Consider the following context-free grammar where the start symbol is S and the set of terminals is $\{a,b,c,d\}$.

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow cS \mid \epsilon$

$B \rightarrow dS \mid \epsilon$

	a	b	c	d	$\$$
S	$S \rightarrow AaAb$	$S \rightarrow BbBa$	(1)	(2)	
A	$A \rightarrow \epsilon$	(3)	$A \rightarrow cS$		
B	(4)	$B \rightarrow \epsilon$		$B \rightarrow dS$	

The following is a partially-filled LL(1) parsing table.

Which one of the following options represents the CORRECT combination for the numbered cells in the parsing table? Note: In the options, "blank" denotes that the corresponding cell is empty. **(Gate 2024 CS)(2 Marks)(MCQ)**

(a) (1) $S \rightarrow AaAb$ (2) $S \rightarrow BbBa$ (3) $A \rightarrow \epsilon$ (4) $B \rightarrow \epsilon$

(b) (1) $S \rightarrow BbBa$ (2) $S \rightarrow AaAb$ (3) $A \rightarrow \epsilon$ (4) $B \rightarrow \epsilon$

(c) (1) $S \rightarrow AaAb$ (2) $S \rightarrow BbBa$ (3) blank (4) blank

(d) (1) $S \rightarrow BbBa$ (2) $S \rightarrow AaAb$ (3) blank (4) blank

Q. Consider two grammars G_1 and G_2 with the production rules given below:

$$G_1:\ S \rightarrow if\ E\ then\ S \mid if\ E\ then\ S\ else\ S \mid a$$

$$E \rightarrow b$$

$$G_2: S \rightarrow if\ E\ then\ S \mid M\ M \rightarrow if\ E\ then\ M\ else\ S \mid c$$

$$E \rightarrow b$$

where if,$then$,$else$, a, b, c are the terminals.

Which of the following option(s) is/are CORRECT? **(Gate 2025)**

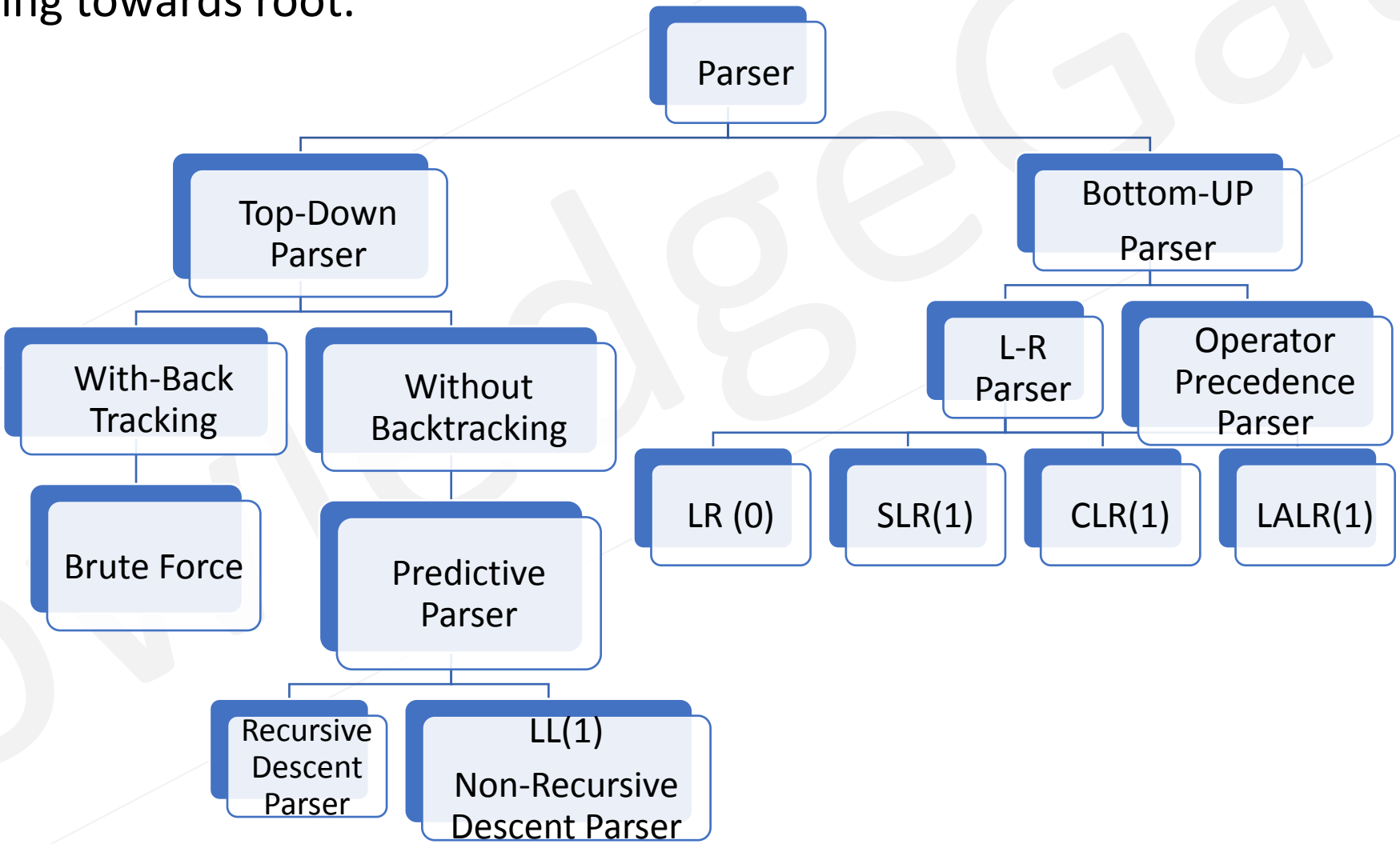
A) G_1 is not $LL(1)$ and G_2 is $LL(1)$.
B) G_1 is $LL(1)$ and G_2 is not $LL(1)$.
C) G_1 and G_2 are not $LL(1)$.
D) G_1 and G_2 are ambiguous.

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 = abcdc



- LR parsers were invented by Donald Knuth in 1965 as an efficient generalization of precedence parsers. Knuth proved that LR parsers were the most general-purpose parsers possible.

- **Donald Ervin Knuth** (born January 10, 1938) is an American computer scientist, mathematician, and professor emeritus at Stanford University. He is the 1974 recipient of the ACM Turing Award, informally considered the Nobel Prize of computer science. Knuth has been called the "father of the analysis of algorithms".

Q S -> AA

A -> aA / b

a	a	a	b	$

	Action			Goto

A -> aA / b

Stack	i/p	Action
$	aaab$	Shift
$a	aab$	Shift
$aa	ab$	Shift
$aaa	b$	Shift
$aaab	$	Reduce(A□b)
$aaaA	$	Reduce(A□aA)
$aaA	$	Reduce(A□aA)
$aA	$	Reduce(A□aA)
$A	$	accept

Q. Given the following syntax directed translation rules:

Rule 1: $R \rightarrow AB \{B.i = R.i - 1; A.i = B.i; R.i = A.i + 1;\}$

Rule 2: $P \rightarrow CD \{P.i = C.i + D.i; D.i = C.i + 2;\}$

Rule 3: $Q \rightarrow EF \{Q.i = E.i + F.i;\}$

Which ONE is the CORRECT option among the following? **(Gate 2025)**

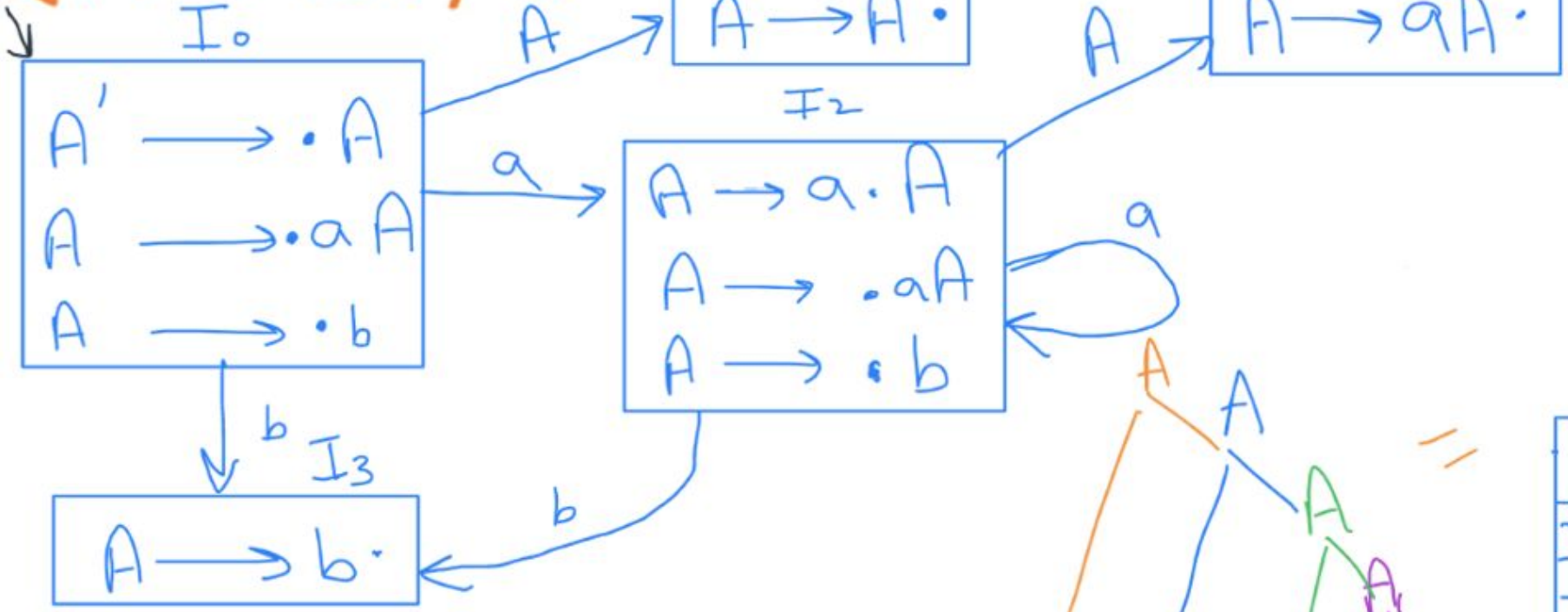
A) Rule 1 is S-attributed and L-attributed; Rule 2 is S-attributed and not L-attributed; Rule 3 is neither S-attributed nor L-attributed

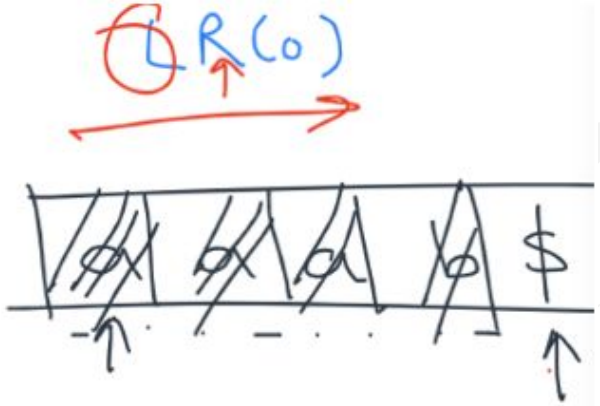
B) Rule 1 is neither S-attributed nor L-attributed; Rule 2 is S-attributed and L-attributed; Rule 3 is S-attributed and L-attributed

C) Rule 1 is neither S-attributed nor L-attributed; Rule 2 is not S-attributed and is L-attributed; Rule 3 is S-attributed and L-attributed

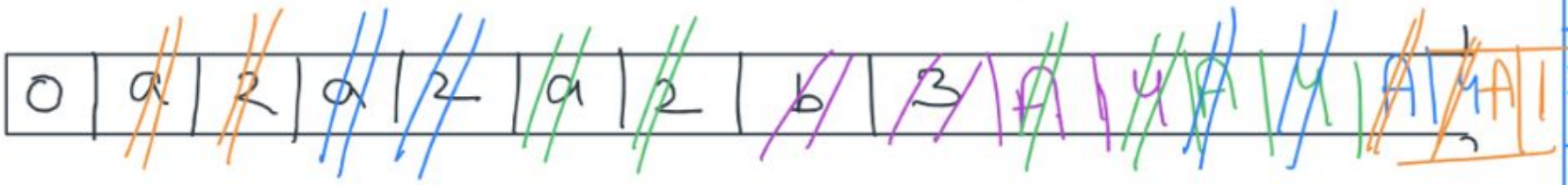
D) Rule 1 is S-attributed and not L-attributed; Rule 2 is not S-attributed and is L-attributed; Rule 3 is S-attributed and L-attributed

Q $A \rightarrow aA / b$ (r_1, r_2)



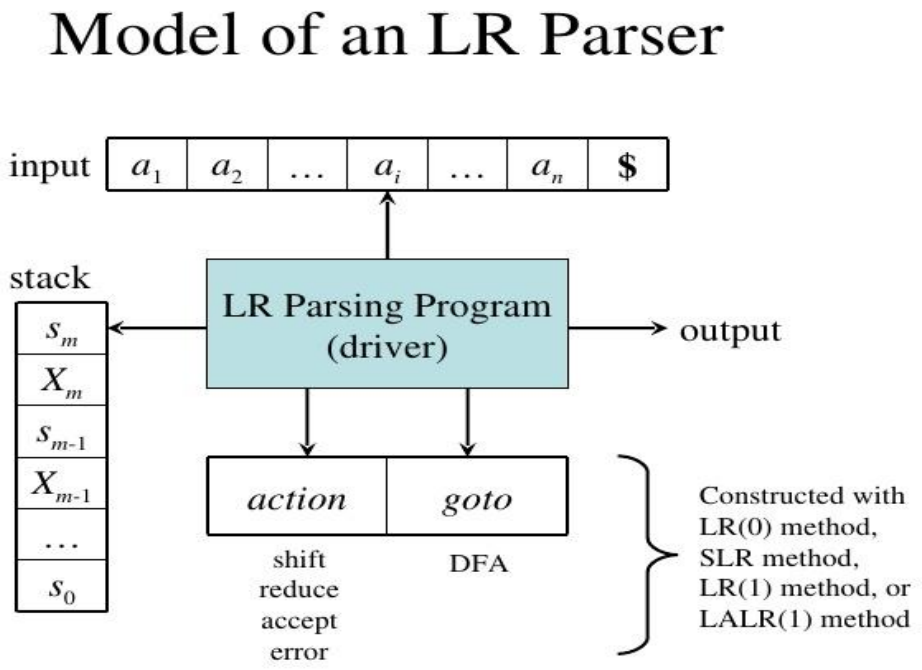


	Action			Goto
	a	b	$	A
I_0	S_2	S_3		S_1
I_1			accept	
I_2	S_2	S_3		S_4
I_3	r_2	r_2	r_2	
I_4	r_1	r_1	r_1	



- **Bottom-Up Parsing (BUP)**
  - **Key Concepts**:
    - **Handle**: A substring of the input string that matches the RHS of any production.
    - **Handle Pruning**: The process of finding a handle and replacing it with its LHS variable.
  - **Characteristics**:
    - Also known as a **Shift-Reduce Parser**.
    - Simulates the **reverse of the rightmost derivation**.
    - Can be constructed for:
      - **Ambiguous grammars**: Operator Precedence Parsing (OPP).
      - **Unambiguous grammars**: LR(k) Parsing.
  - **Advantages**:
    - Faster than Top-Down Parsing (TDP).
    - More efficient for grammars with higher complexity.
    - Average time complexity: O(n3)O(n^3)O(n3).
    - Performance is generally **higher** than TDP.
  - **Disadvantages**:
    - **Handle pruning** introduces computational overhead.

- **Components of a Bottom-Up Parser**
  - **Input Buffer**:
    - Divided into cells, each containing one input symbol.
  - **Parse Stack**:
    - Stores **grammar symbols**.
    - Operations:
      - **Shift**: Pushes symbols into the stack if a handle is not found.
      - **Reduce**: Replaces a handle (if found at the top of the stack) with its corresponding LHS.
  - **Parse Table**:
    - Constructed using **terminals**, **non-terminals**, and **LR(0) items**.
    - Divided into two parts:
      - **Action**: Handles **shift** and **reduce** operations for terminals.
      - **Goto**: Handles **shift** operations for non-terminals.



- **Operations in Shift-Reduce Parsing**
  - **Shift**:
    - Used when a handle does **not** occur at the top of the stack.
    - Moves the **lookahead symbol** from the input buffer into the stack.
  - **Reduce**:
    - Used when a handle **does** occur at the top of the stack.
    - Replaces the handle (topmost stack symbol) with the corresponding LHS of the production.
  - **Accept**:
    - Parsing is successful if:
      - The entire input string has been scanned, and
      - The stack contains only the **start symbol** as the topmost symbol.
  - **Error**:
    - Parsing is unsuccessful if:
      - After scanning the entire input string, the stack contains a symbol **different** from the start symbol as the topmost symbol.

	Action	Goto
I_0	Terminals	Non-Terminals
I_{n-1}	Shift / Reduce	Shift

- **LR(k) Parsers**
  - **L**: Scans the input **Left-to-right**.
  - **R**: Simulates the **reverse of rightmost derivation**.
  - **k**: Refers to the number of **lookahead symbols** used during parsing.

- **Procedure to Construct an LR Parser Table**
  - **Augmented Grammar**:
    - Derive the **augmented grammar** from the given grammar.
  - **Canonical Collection of LR Items**:
    - Create a collection of **LR items** (also known as compiler items).
  - **DFA Construction**:
    - Build the **Deterministic Finite Automaton (DFA)** using the LR items.
  - **Prepare the Table**:
    - Use the DFA states to construct the **LR parsing table**.

- **Augmented grammar**
  - The grammar which is obtained by addition one more production that generate the 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) 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,…….I_N} is known as canonical collection of LR(0) items.

- **Functions Used to Generate LR(0) Items**
- **Closure**:
  - **Input**: A set of items.
  - **Output**: An expanded set of items.
  - Procedure:
    - Add all items from the input to the output.
    - If an item $A \rightarrow \alpha.\beta BA$ is in **Closure(I)** and $\beta \rightarrow \sigma$ is in grammar G:
      - Add $\beta \rightarrow .\sigma$ to **Closure(I)**.
    - Repeat the step for every newly added item.
- **Goto**:
  - **Input**: A set of items I and a symbol X.
  - **Output**: Transition state after processing X.
  - Procedure:
    - For $A \rightarrow \alpha.X\beta A$ in I:
      - Transition to $A \rightarrow \alpha X.\beta$ when X is processed.

Procedure to construct LR parse Table

- The LR parse table has two parts:
- **Action**:
  - Contains **shift** and **reduce** operations performed on **terminals**.
  - If Ii is a final item representing production Ri, place Ri under all the terminal symbols in the **Action** part.
- **Goto**:
  - Contains **shift** operations performed on **non-terminals**.
  - Transition definitions:
    - Goto(Ii,X)=Ij, where X is a **terminal**.
    - Goto(Ii,X)=Ij, where X is a **non-terminal**.

	X
I_i	S_j

	X
I_i	j

	t_1	t_2	t_3					t_n	$
I	r_i	r_i	r_i					r_i	r_i

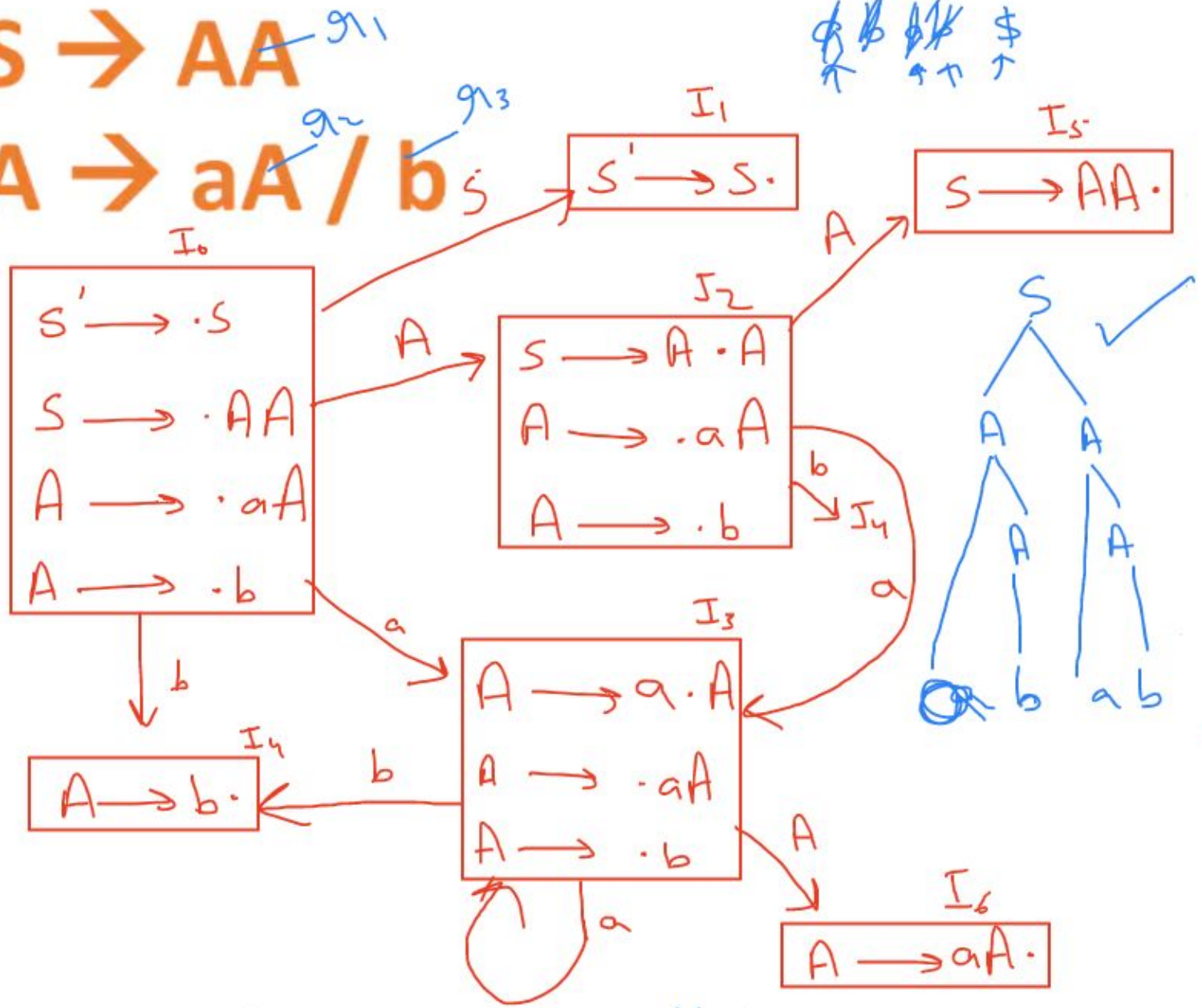
S -> AA

A -> aA / b

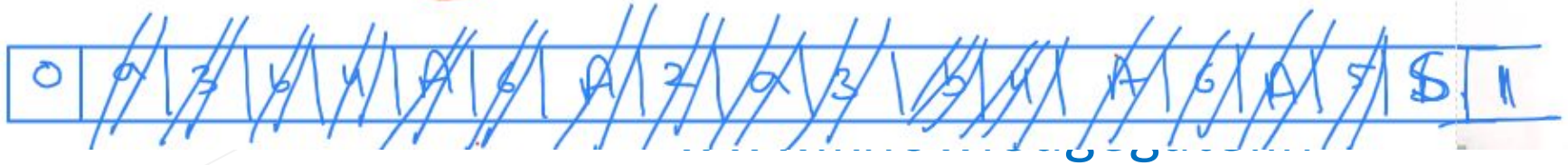
	Action			Goto	

$S \rightarrow AA$ g_1

$A \rightarrow aA$ g_2 / b g_3



	Action			Goto	
	a	b	$	S	A
I_0	S_3	S_4		1	2
I_1			accept		
I_2	S_3	S_4			5
I_3	S_3	S_4			6
I_4	g_3	g_3	g_3		
I_5	g_1	g_1	g_1		
I_6	g_2	g_2	g_2		



E □ T + E / T

T □ id

- **Why is it called LR(0)**
  - The term **"0"** refers to the fact that the parser does not consider the **lookahead symbol** when performing reductions.
  - The parser focuses solely on the current state and the items in the parse stack.
- **Reason for its Limitations:**
  - Since it ignores the lookahead symbol, the parser cannot distinguish between conflicting scenarios that require lookahead to resolve.
  - This simplicity makes LR(0) the **least powerful** among LR parsers, as it cannot handle certain grammars that more advanced parsers (e.g., SLR(1), LALR(1)) can manage.

Conflicts in LR(0) parser

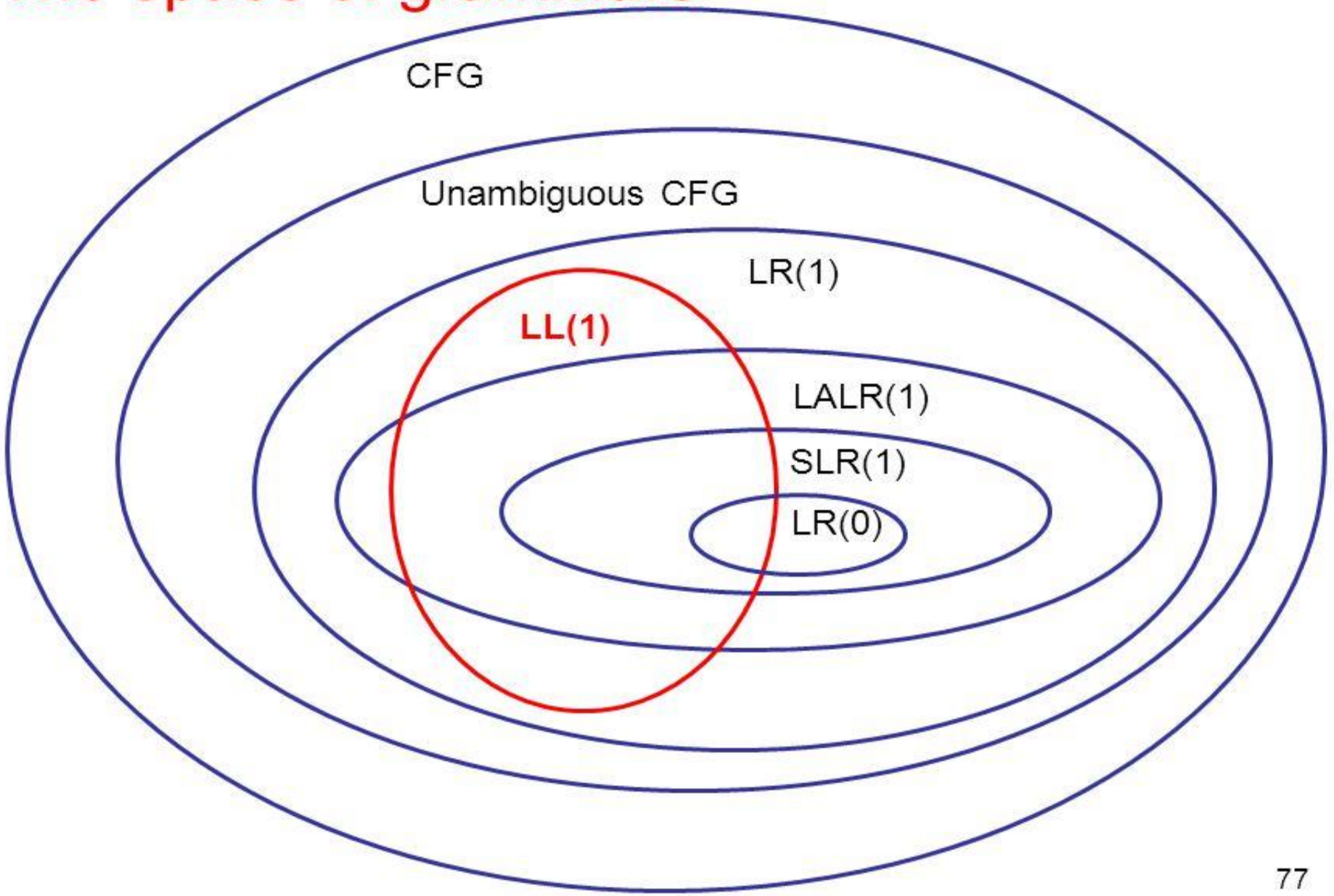
- **SR Conflicts (Shift Reduce)**
- **RR Conflicts (Reduce Reduce)**

- 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 □ɞ. (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.

The space of grammars



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) E -> F. + E

(iii) E -> 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

E → E + T / T
T → T * F / F
F → id

S → (L) / a
L → L,S / S

S → AaAb / BbBa
A → ε
B → ε

S → aAB / 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
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 augmented grammar with {+, *, (,), id} as the set of terminals.

$S' \rightarrow S$

$S \rightarrow S + R \mid R$

$R \rightarrow R^* P \mid P$

$P \rightarrow (S) \mid id$

If I_0 is the set of two LR(0) items {[S' → S.], [S → S. + R]}, then goto(closure(I_0),+) contains exactly _____ items. **(GATE 2022) (1 MARKS)**

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

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

$E \longrightarrow E + n \mid E \times n \mid n$

For a sentence $n + n \times n$, 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

Q. Consider the following augmented grammar, which is to be parsed with a SLR parser. The set of terminals is $\{a,b,c,d,\#,@\}$ **(Gate 2024 CS) (2 Marks) (NAT)**

$$S' \rightarrow S$$

$$S \rightarrow SS \mid Aa \mid bAc \mid Bc \mid bBa$$

$$A \rightarrow d\#$$

$$B \rightarrow @$$

Let $I_0 = CLOSURE(\{S' \rightarrow \bullet S\})$. The number of items in the set $GOTO\ (I_0, S)$ is

_____

Q.Which of the following fields is/are Bottom-Up Parser? **(Gate 2024,CS) (1 Marks) (MSQ)**

(a) Shift-reduce Parser

(b) Predictive Parser

(c) LL(1) Parser

(d) LR Parser

SLR(1)

- **Characteristics:**
  - The **procedure** for constructing the SLR(1) parse table is similar to the **LR(0) parse table**, but with additional restrictions for reducing entries.
- **Steps for SLR(1) Parse Table Construction:**
  - **Reduction Rule**:
    - If there is a **final item**, place the **reduce entries** under the **follow set of the LHS symbol** of the production.
    - This ensures that reduction is only performed when it is valid based on the input string.
  - **Conflict Resolution**:
    - If the resulting SLR(1) parse table contains **no multiple entries**, then the grammar is classified as **SLR(1)**.
- **Key Difference from LR(0):**
  - LR(0) allows reductions to be placed in all cells of the row corresponding to a final item.
  - SLR(1) restricts reductions to only those cells corresponding to the **follow set** of the LHS, thus avoiding conflicts in some cases.
  - SLR(1) parsers are a step forward from LR(0) in handling grammars with fewer conflicts, but they are still less powerful compared to LALR(1) or LR(1). Let me know if you'd like a detailed example!

E -> T + E / T

T -> id

	Action			Goto	

S □ Aa / bAc / dc / bda

A □ d

	Action					Goto	

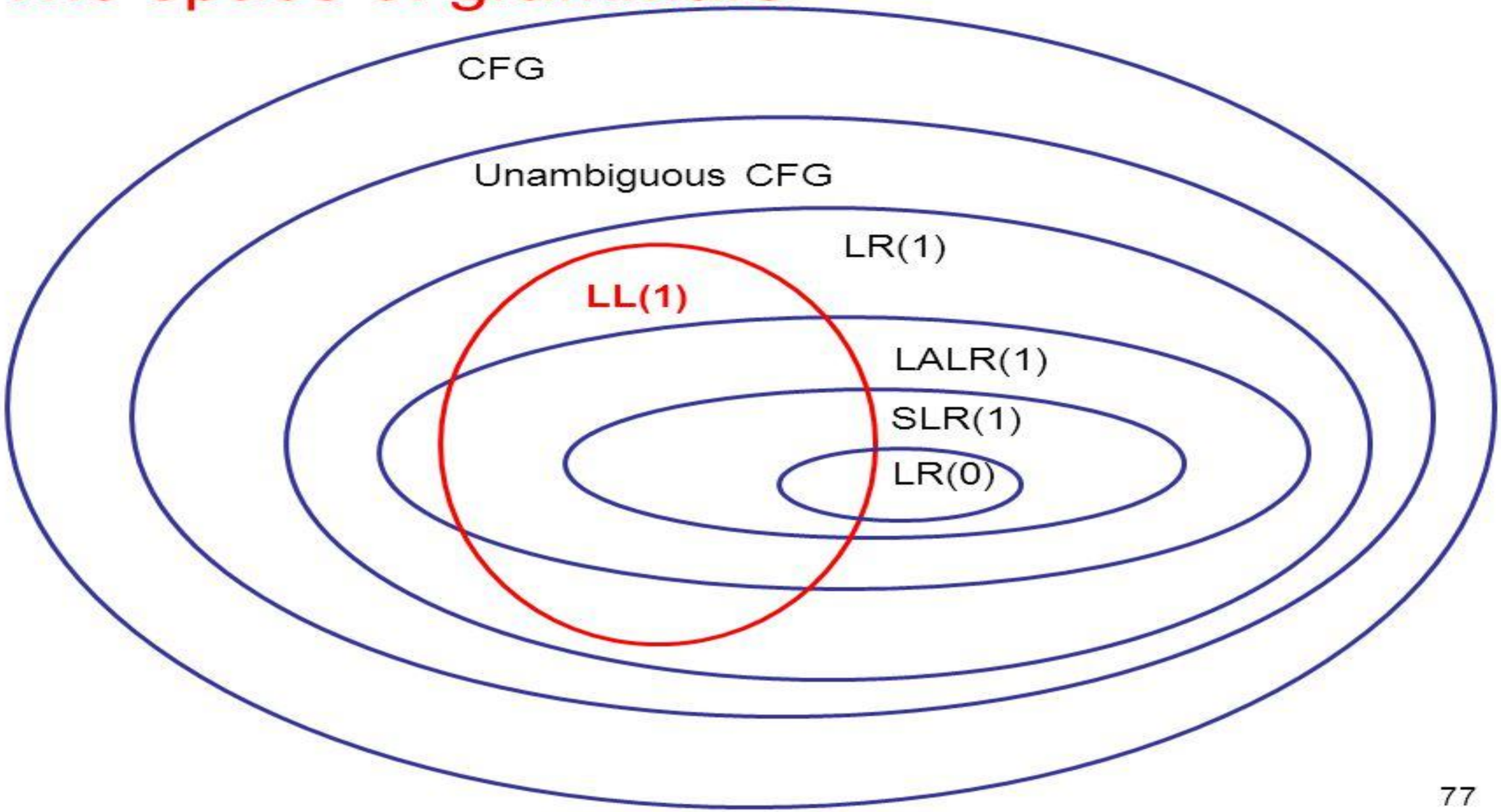
- Conflicts in SLR (1):
  - SR Conflicts (Shift Reduce)
  - RR Conflicts (Reduce Reduce)

- 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 □ɞ. (reduced)
  if Follow(B) ∩ {x} != ϕ

- R-R Conflict: If the same state contains more than one final item, then it is R-R Conflict
  A □ α. (reduced)
  B □α. (reduced)
  if Follow(A) ∩ Follow(B) != ϕ

- **Relationship**:
  - Every **LR(0)** grammar is an **SLR(1)** grammar.
  - However, not every **SLR(1)** grammar is an **LR(0)** grammar.
- **Power**:
  - **SLR(1) parsers** are **more powerful** than **LR(0) parsers** because SLR(1) uses the **follow set** to resolve reduce operations, leading to fewer conflicts.
- **Parse Table**:
  - The number of entries in an **SLR(1) parse table** is **less than or equal to** the number of entries in an **LR(0) parse table**. This is due to the additional restrictions imposed by the SLR(1) mechanism, which reduces redundancy in the parse table.

The space of grammars



S □ AaB
A □ ab / a
B □ b

S □ Aa / bAc / dc / bda
A □ ε

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

Q Consider the following augmented grammar with {#,@,<,>,a,b,c} as the set of terminals.

$S' \rightarrow S$

$S \rightarrow S\#cS$

$S \rightarrow SS$

$S \rightarrow S@$

$S \rightarrow <S>$

$S \rightarrow a$

$S \rightarrow b$

$S \rightarrow c$

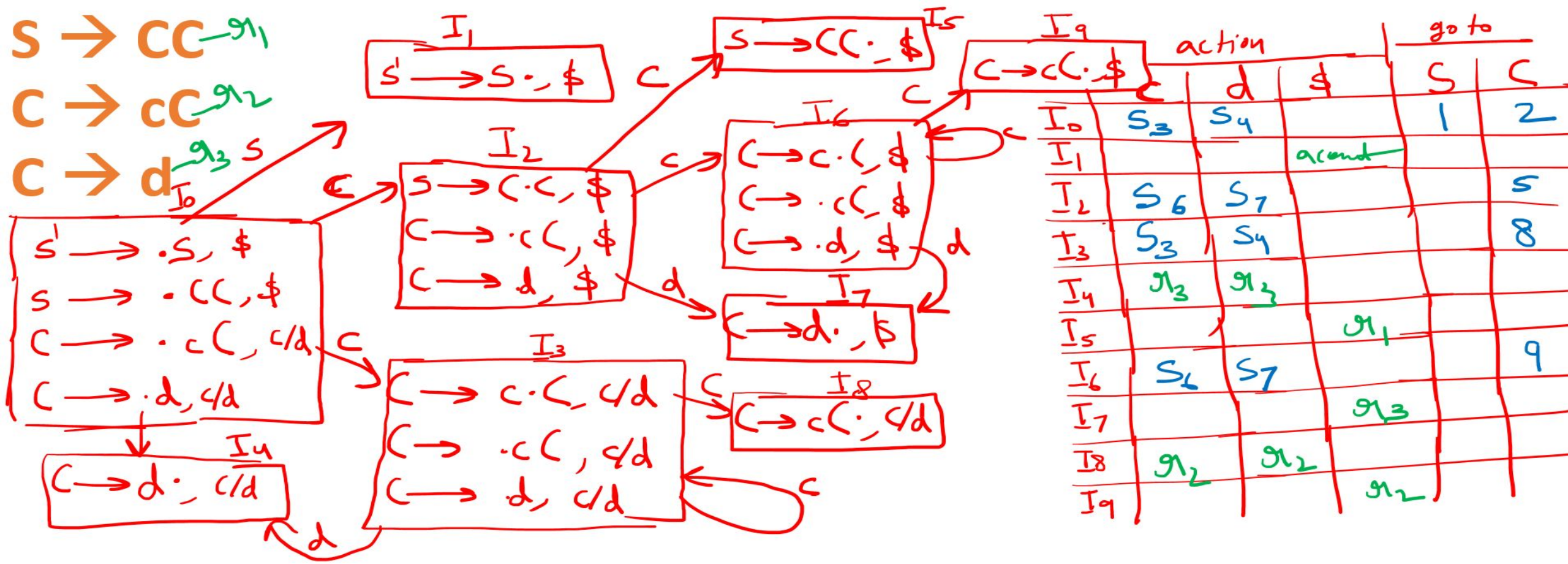
Let I0=CLOSURE({$S' \rightarrow \cdot S$}). The number of items in the set GOTO(GOTO(I0 <), <) is _____ .**(GATE 2021) (2 MARKS)**

S -> CC

C -> cC

C -> d

	Action			Goto	



CLR(1)

- 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 □ ɞ is in the grammar G, then add β □ .ɞ, 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.

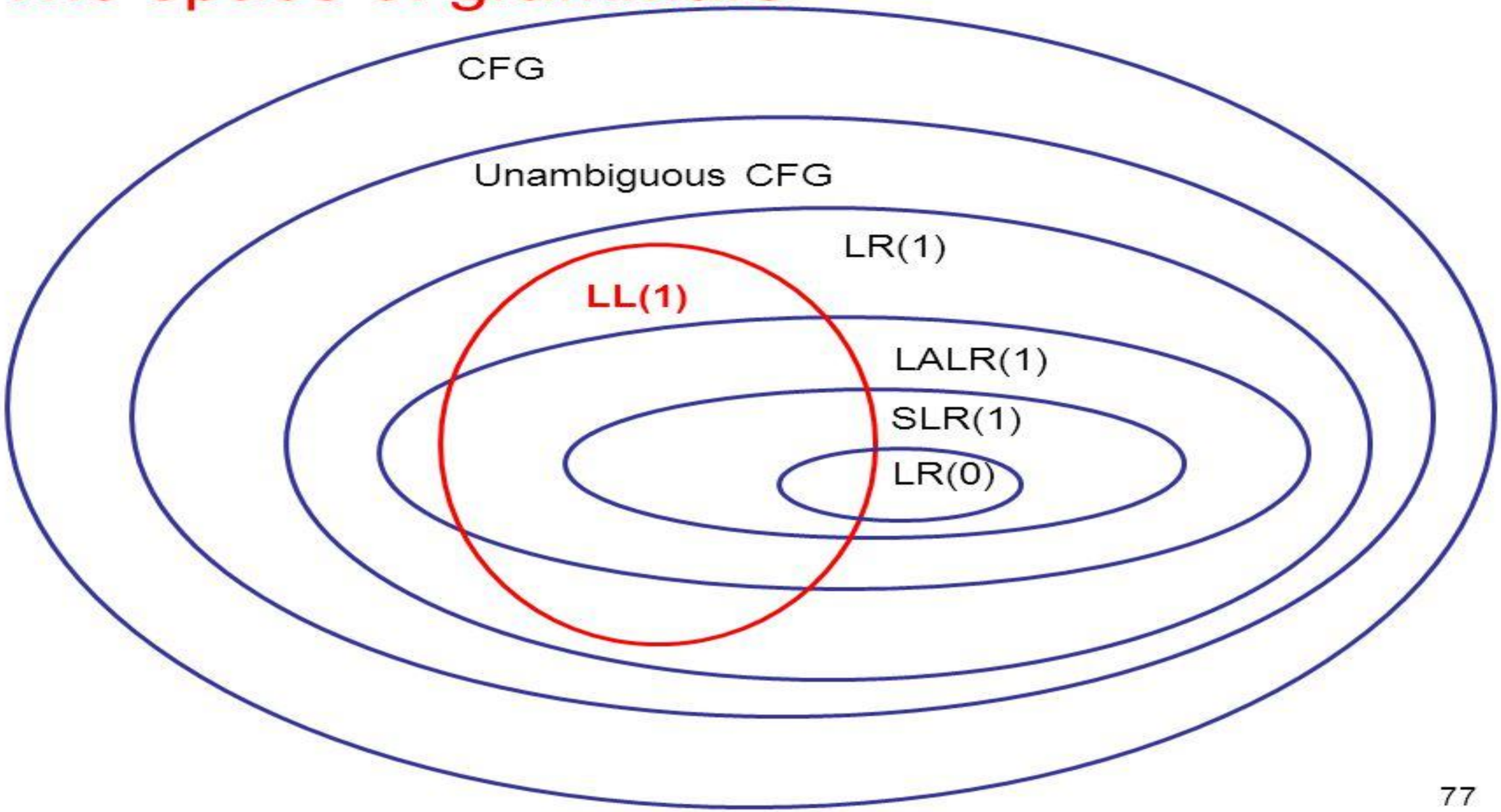
- Conflicts of LR(1)
  - there are also two conflicts, SR & RR

  - SR Conflicts
    - A □ α.aβ, b
    - β □ ɞ., a

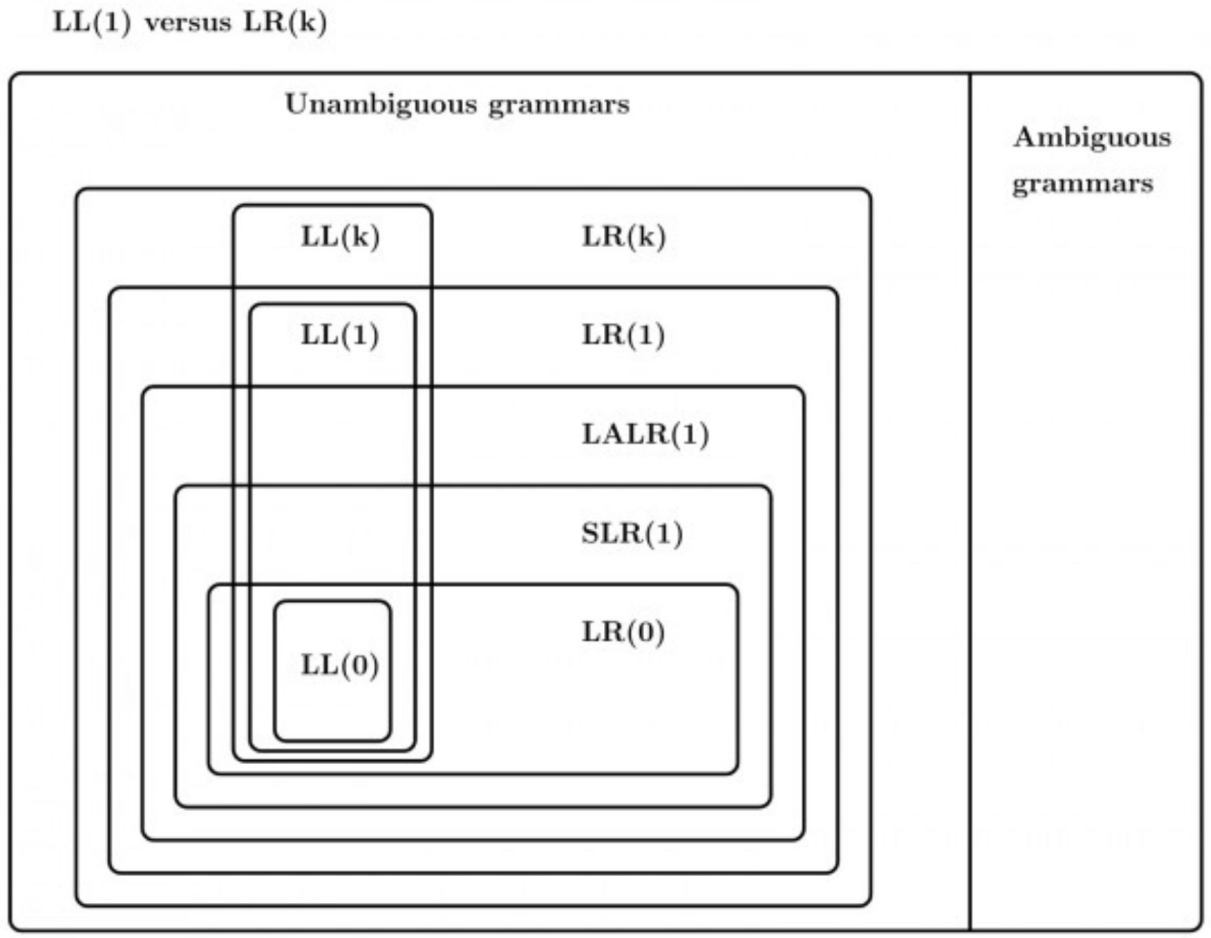
  - RR Conflicts
    - A □ .ɞ, a
    - B □ .ɞ, a

- **Key Points on LR(1) or CLR(1) Grammar:**
- **Definition**:
  - A grammar is said to be **LR(1)** or **CLR(1)** if its **LR(1) parse table** is free from **multiple entries** or **conflicts** (shift-reduce or reduce-reduce conflicts).
- **Relationship**:
  - Every **SLR(1) grammar** is also a **CLR(1) grammar**.
  - However, not every **CLR(1) grammar** is an **SLR(1) grammar**. This is because CLR(1) parsers consider **lookahead symbols** explicitly, making them more powerful.
- **Power**:
  - **CLR(1)** parsers are more powerful than **SLR(1)** parsers because they include **lookahead information** in the parsing process, which helps resolve conflicts more effectively.
- **Parse Table**:
  - The number of rows in the **SLR(1) parse table** is always **less than or equal to** the number of rows in the **CLR(1) parse table**. This is because CLR(1) uses additional information, resulting in more states.

The space of grammars



- **LR(k) Grammars and Parsers**
  - A language can be generated by an **LR(k) grammar** for any **k ≥ 1** if and only if it is **deterministic and context-free**, and can also be generated by an **LR(1) grammar**.
  - The unique characteristic of this parser is that **any LR(k) grammar with k > 1 can be transformed into an equivalent LR(1) grammar**.
  - LR(k) parsers are capable of handling **all deterministic context-free languages**, making them highly versatile and powerful in parsing deterministic grammars.



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

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.

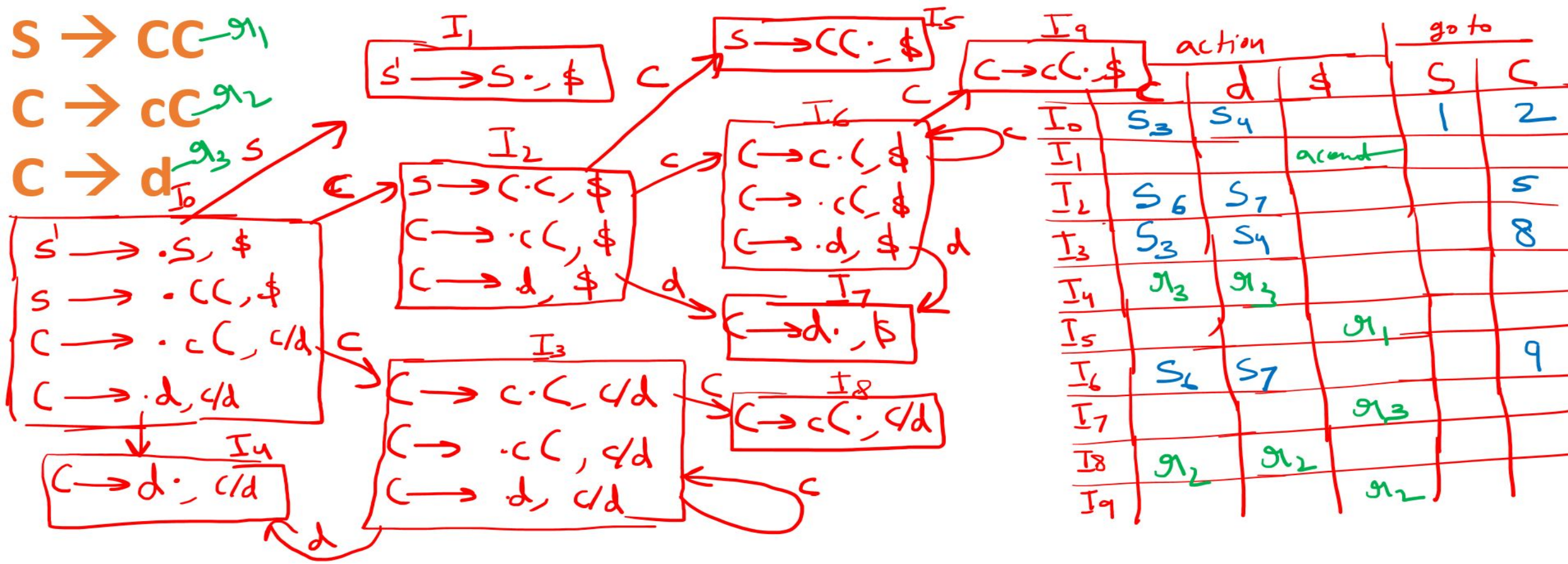
(C) a reduce-reduce conflict but not a shift-reduce conflict.

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

- **CLR(1) Parser Characteristics**
  - In CLR(1) parsers, certain states may contain multiple productions where the production body is identical, but the follow sets differ.
  - This leads to a higher number of table entries in CLR(1) parsers, making them more complex and costly compared to other parsers.

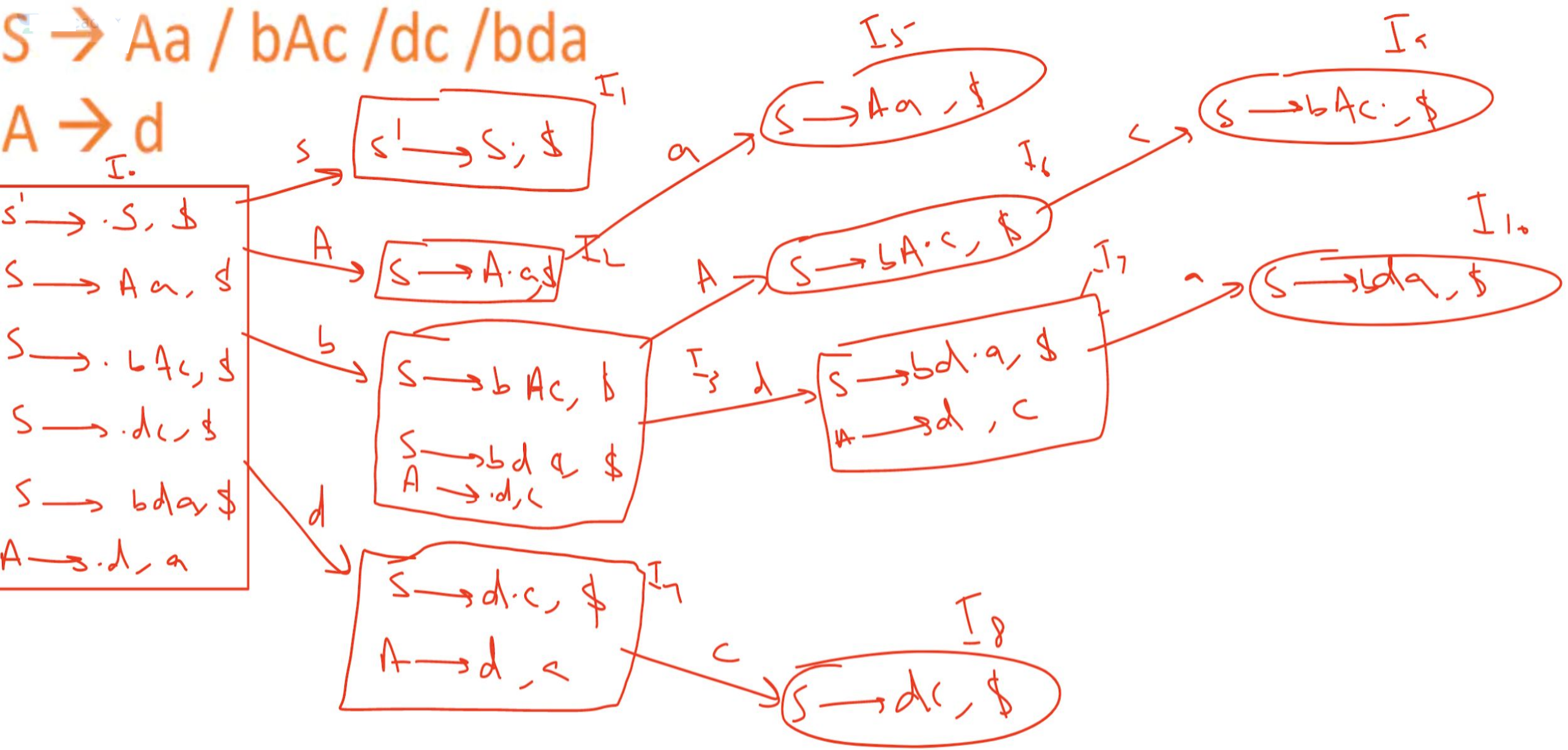
- **LALR(1) Parser**
  - In a CLR(1) parser, multiple states may have the same production body but different follow sets.
  - To create an LALR(1) parser, combine such states into a single state and construct the parse table. If this table is free from multiple entries, the grammar is LALR(1).

- **Conflicts in LALR(1) Parser:**
  - **Shift-Reduce (SR) Conflicts**:
    If there are no SR conflicts in CLR(1), there will be no SR conflicts in LALR(1) either.
  - **Reduce-Reduce (RR) Conflicts**:
    While there may be no RR conflicts in CLR(1), they can occur 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

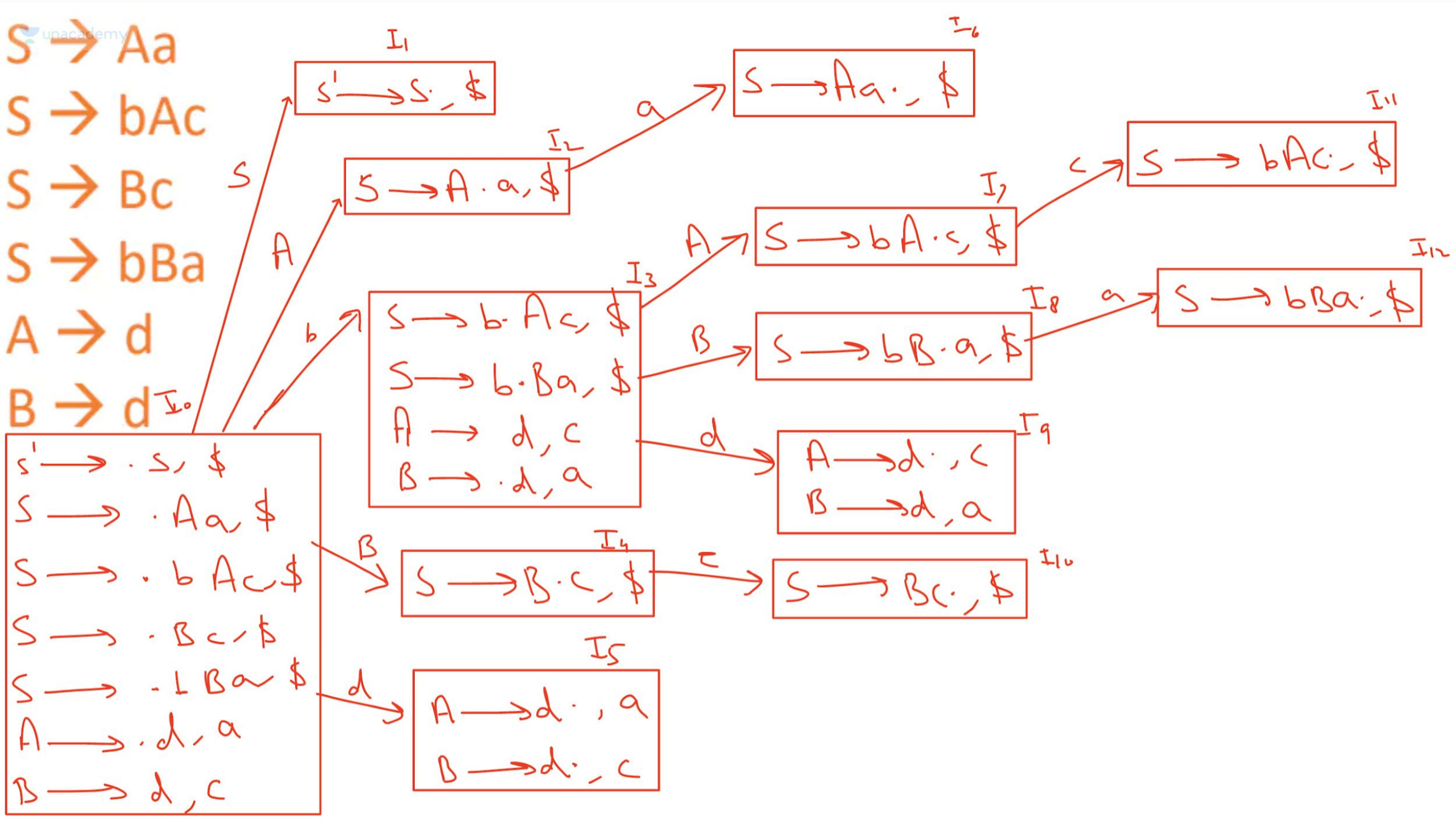
S -> bAc

S -> Bc

S -> bBa

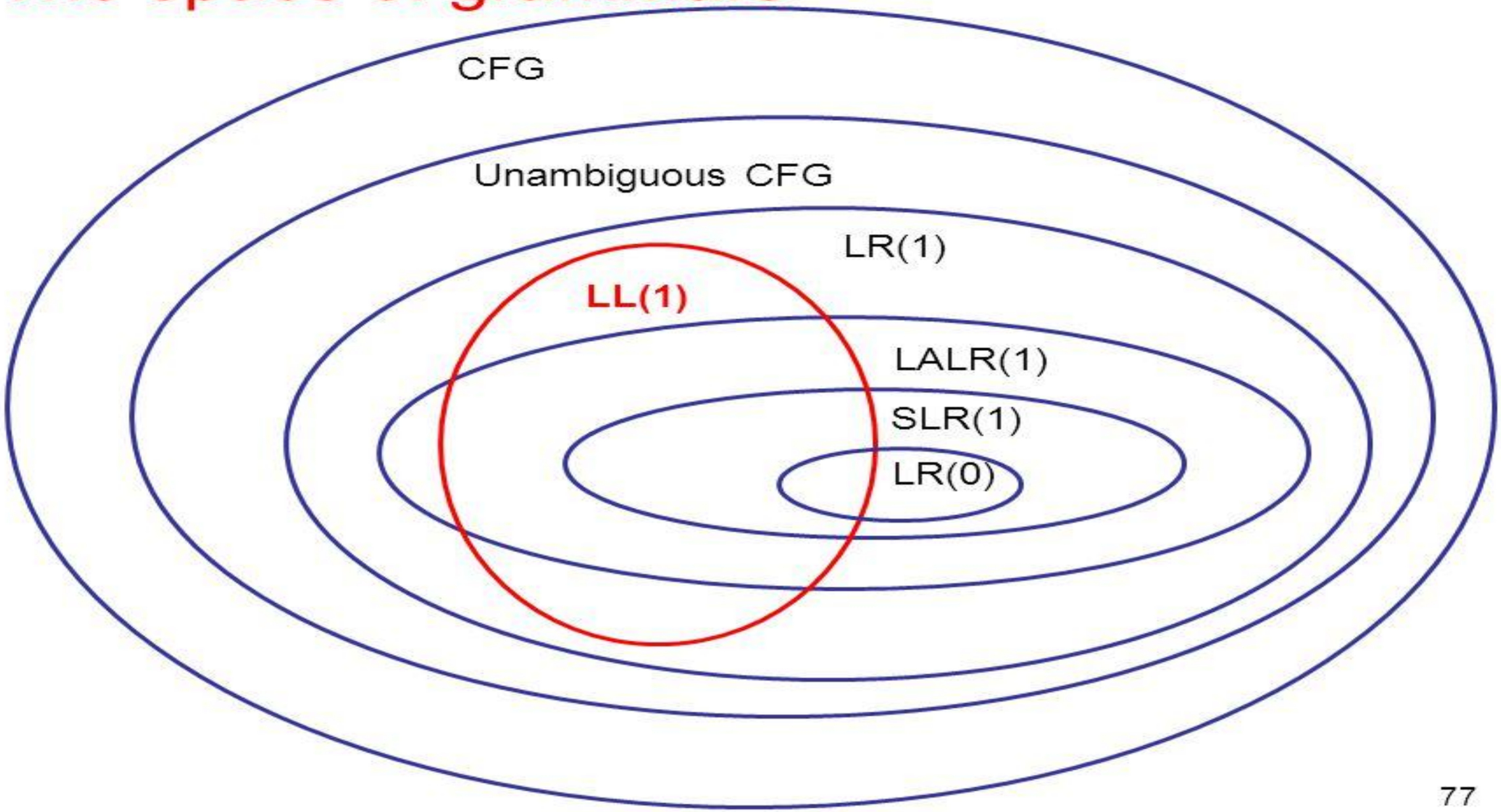
A -> d

B -> d


S → Aa
S → bAc
S → Bc
S → bBa
A → d
B → d
I0
s' → . S, $
S → . Aa, $
S → . bAc, $
S → . Bc, $
S → . bBa, $
A → . d, a
B → d, c
S
A
B
b
d
I1
s' → S . , $
I2
S → A . a, $
a
I6
S → Aa . , $
I3
S → b . Ac, $
S → b . Ba, $
A → d, c
B → . d, a
A
B
d
I7
S → bA . c, $
c
I11
S → bAc . , $
I8
S → bB . a, $
a
I12
S → bBa . , $
I9
A → d . , c
B → d, a
I4
S → B . c, $
c
I10
S → Bc . , $
I5
A → d . , a
B → d . , c

- **Relationship Between CLR(1), LALR(1), and SLR(1)**
  - Every **LALR(1)** grammar is a **CLR(1)** grammar, but the reverse is not always true.
  - The number of entries in an **LALR(1)** parse table is less than or equal to those in a **CLR(1)** parse table.
  - Every **SLR(1)** grammar is a **LALR(1)** grammar, but the reverse is not always true.
  - **LALR(1)** is more powerful than **SLR(1)**.

The space of grammars



Q. Given a Context-Free Grammar G as follows:

$$S \rightarrow Aa \mid bAc \mid dc \mid bda$$

$$A \rightarrow d$$

Which ONE of the following statements is TRUE? **(Gate 2025)**

A) G is neither LALR(1) nor SLR(1)

B) G is CLR(1), not LALR(1)

C) G is LALR(1), not SLR(1)

D) G is LALR(1), also SLR(1)

Operator Precedence Grammar

- An **operator precedence parser** can handle both ambiguous and unambiguous grammars.
- These grammars typically have **lower complexity** compared to other context-free grammars.
- **Not all CFGs** (Context-Free Grammars) qualify as operator precedence grammars.
- Operator precedence grammars are commonly used in languages designed for **scientific applications**.
- An **Operator Grammar** is a type of **context-free grammar** with the following characteristics:
  - **No ε-productions**: It does not allow empty (ε) productions.
  - **No adjacent non-terminals**: The right-hand side (RHS) of any production does not have two consecutive non-terminal symbols.

$S \rightarrow AB$

$A \rightarrow a$

$B \rightarrow b$

$S \rightarrow AaB$

$B \rightarrow aA \mid b$

$A \rightarrow b$

$B \rightarrow a$

$S \rightarrow AaB$

$A \rightarrow a \mid \varepsilon$

$B \rightarrow b$

$S \rightarrow AOB \mid int$

$O \rightarrow + \mid * \mid -$

$A \rightarrow b$

$B \rightarrow a$

- **Operator Precedence Parsing Algorithm**
  - Let **a** be the top of the stack (TOS) and **b** be the lookahead symbol.
  - **Shift Operation**:
    - If **a , b** (precedence of a is less than b) or **a <= b**, then:
      - Shift **b** onto the stack.
      - Increment the input pointer.
  - **Reduce Operation**:
    - If **a > b** (precedence of a is greater than b):
      - Pop symbols from the stack.
      - Continue popping until the current TOS has precedence < the previous TOS.
  - **Success**:
    - If **a == b == $** (end of input and stack), the parsing completes successfully.

Q Which one of the following statements is TRUE? **(GATE 2022) (1 MARKS)**

(A) The LALR(1) parser for a grammar G cannot have reduce-reduce conflict if the LR(1) parser for G does not have reduce-reduce conflict.

(B) Symbol table is accessed only during the lexical analysis phase.

(C) Data flow analysis is necessary for run-time memory management.

(D) LR(1) parsing is sufficient for deterministic context-free languages.

Q Consider the following statements.

- **S_1:** Every SLR(1) grammar is unambiguous but there are certain unambiguous grammars that are not SLR(1).
- **S_2:** For any context-free grammar, there is a parser that takes at most $O(n^3)$ time to parse a string of length n.

Which one of the following options is correct? **(GATE 2021)**

(a) S_1 is true and S_2 is false

(b) S_1 is false and S_2 is true

(c) S_1 is true and S_2 is true

(d) S_1 is false and S_2 is false

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

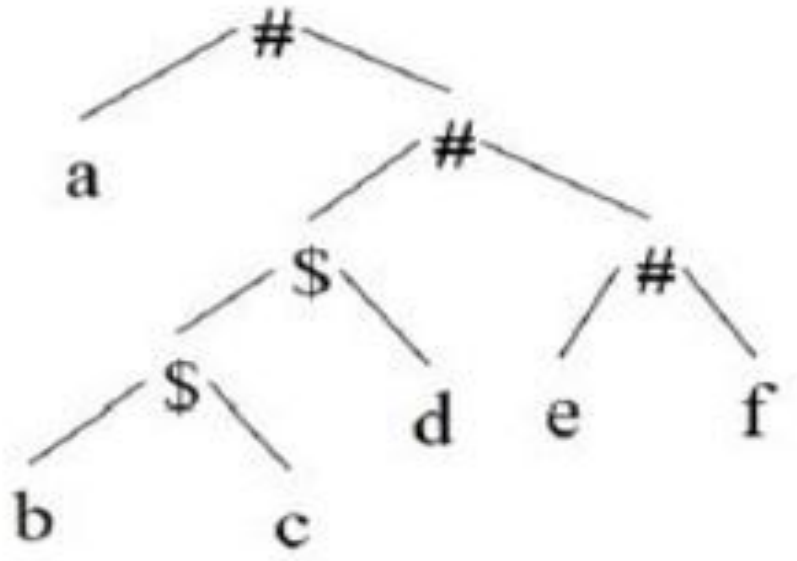
$S' \rightarrow S$

$S \rightarrow (L) / id$

$L \rightarrow L, S / S$

Let I_0 = CLOSURE ({[$S' \rightarrow S$]}). The number of items in the set GOTO (I_0, ( ) is _____.

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

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.

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

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 _____.

Q A student wrote two context-free grammars G_1 and G_2 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 G_1

D → int L;

L → id [E

E → num]

E → num] [E

Grammar G_2

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 G_1 and G_2 **b)** Only G_1

c) Only G_2 **d)** Neither G_1 nor G_2

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

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

Q Consider the following grammar G.

$S \rightarrow F / H$

$F \rightarrow p / c$

$H \rightarrow d / c$

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)**

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

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

(A) Only S_1

(B) Only S_2

(C) Both S_1 and S_2

(D) Neither S_1 and S_2

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

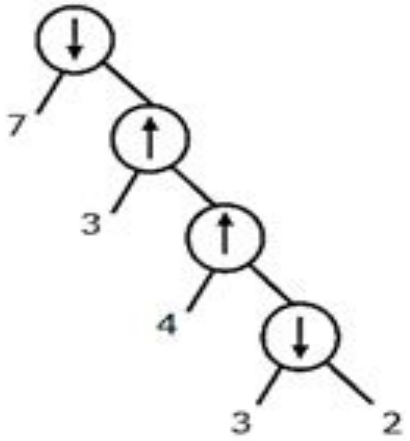
X -> c.X, $
X -> .cX, $
X -> .d, $

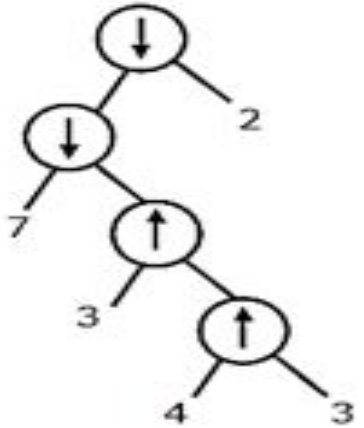
Which of the following statements related to merging of the two sets in the corresponding LALR parser is/are FALSE?

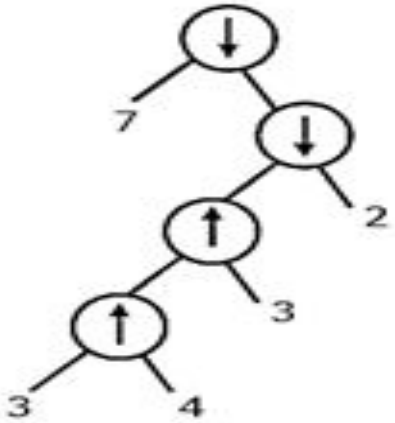
1. Cannot be merged since look aheads are different.
2. Can be merged but will result in S-R conflict.
3. Can be merged but will result in R-R conflict.
4. Cannot be merged since goto on c will lead to two different sets.

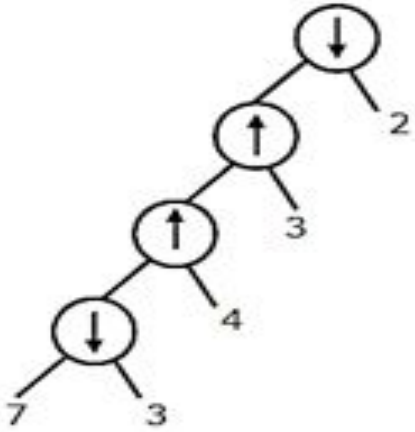
(A) 1 only
(B) 2 only
(C) 1 and 4 only
(D) 1, 2, 3, and 4

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)**









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

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

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

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

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

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

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

Q Which of the following statements are true? **(GATE-2008) (1 Marks)**

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 → w or X → 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

a) I, II, III and IV

b) II, III and IV only

c) I, III and IV only

d) I, II and IV only

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

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

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. $P \rightarrow Q\ R$
2. $P \rightarrow Q\ s\ R$
3. $P \rightarrow \varepsilon$
4. $P \rightarrow Q\ t\ R\ r$

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

Q Assume that the SLR parser for a grammar G has n_1 states and the LALR parser for G has n_2 states. The relationship between n_1 and n_2 is **(Gate - 2003) (2 Marks)**

(A) n_1 is necessarily less than n_2

(B) n_1 is necessarily equal to n_2

(C) n_1 is necessarily greater than n_2

(D) none of these

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

$S \rightarrow C\ C$

$C \rightarrow c\ C \mid d$

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)

Q Which of the following statements is false? **(CS-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

Q Which of the following features cannot be captured by context free grammars? **(GATE – 1994) (2 Marks)**

a) Syntax of if – then – else statements

b) Syntax of recursive producers

c) Whether a variable has been declared before its use

d) Variable names of arbitrary length

Semantic Analysis

- Semantic analysis combines **Grammar**, **Semantic Rules**, and **Semantic Actions** to create **Syntax Directed Translation (SDT)**. This approach ensures that parsing includes meaningful actions, making parsers more powerful and versatile.

- **Applications of SDT:**
  - **Code Generation**: Producing machine-level or intermediate code.
  - **Intermediate Code Generation**: Creating an abstraction for optimization.
  - **Updating Symbol Table**: Storing and accessing variable information.
  - **Expression Evaluation**: Calculating results during parsing.
  - **Infix to Postfix Conversion**: Simplifying mathematical expressions.

- **Advantages:**
  - Tasks can be performed **parallel to parsing**, enhancing efficiency.
  - Integrates parsing with semantic rules to perform complex operations seamlessly.

Q Consider the following grammar (that admits a series of declarations, followed by expressions) and the associated syntax directed translation (SDT) actions, given as pseudo-code

$$
\begin{aligned}
P &\rightarrow D^* E^* \\
D &\rightarrow \text{int ID}\{\text{record that ID.lexeme is of type int}\} \\
D &\rightarrow \text{bool ID}\{\text{record that ID.lexeme is of type bool}\} \\
E &\rightarrow E_1 + E_2\{\text{check that } E_1.\text{type} = E_2.\text{type} = \text{int}; \text{set } E.\text{type} := \text{int}\} \\
E &\rightarrow !E_1\{\text{check that } E_1.\text{type} = \text{bool}; \text{ set } E.\text{type} := \text{bool}\} \\
E &\rightarrow \text{ID}\{\text{set } E.\text{type} := \text{int}\}
\end{aligned}
$$

With respect to the above grammar, which one of the following choices is correct?. **(GATE 2021) (2 MARKS)**

(a) The actions can be used to correctly type-check any syntactically correct program

(b) The actions can be used to type-check syntactically correct integer variable declarations and integer expressions

(c) The actions can be used to type-check syntactically correct boolean variable declarations and boolean expressions.

(d) The actions will lead to an infinite loop

Q Consider the following grammar along with translation rules.

$S \rightarrow S_1 \# T \quad \{S_{.val} = S_{1.val} * T_{.val}\}$

$S \rightarrow T \quad \{S_{.val} = T_{.val}\}$

$T \rightarrow T_1 \% R \quad \{T_{.val} = T_{1.val} \div R_{.val}\}$

$T \rightarrow R \quad \{T_{.val} = R_{.val}\}$

$R \rightarrow id \quad \{R_{.val} = id_{.val}\}$

Here # and % are operators and id is a token that represents an integer and $id_{\bullet val}$ represents the corresponding integer value. The set of non-terminals is {S,T, R, P} and a subscripted non-terminal indicates an instance of the non-terminal.Using this translation scheme, the computed value of $S_{\bullet val}$ for root of the parse tree for the expression 20 #10%5 #8%2%2 is _____. **(GATE 2022) (2 MARKS)**

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

$E \rightarrow E_1$ # T { E.value = E_1.value * T.value }

E → T{ E.value = T.value }

$T \rightarrow T_1$ & F { T.value = T_1.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

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

$E \rightarrow E_1 + T$ {print ('+');}

$E \rightarrow T$

$T \rightarrow T_1 * F$ {print ('*');}

$T \rightarrow F$

$F \rightarrow num$ {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)**

$S \rightarrow T\ R$

$R \rightarrow + T$ {print ('+');} R / ε

$T \rightarrow$ 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

Q Consider the following translation scheme. **(Gate-2006) (2 Marks)**

$S \rightarrow ER$

$R \rightarrow *E\{print(“*”);\}R \mid \varepsilon$

$E \rightarrow F + E \{print(“+”);\} \mid F$

$F \rightarrow (S) \mid id \{print(id.value);\}$

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+*

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

$$S \longrightarrow aA \quad \{ \text{print } 1 \}$$
$$S \longrightarrow a \quad \{ \text{print } 2 \}$$
$$A \longrightarrow Sb \quad \{ \text{print } 3 \}$$

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

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

$E \longrightarrow E_1 * T$ { E.value = E_1.value * T.value }

$E \longrightarrow T$ { E.value = T.value }

$T \longrightarrow F - T$ { T.value = F.value - T_1.value }

$T \longrightarrow F$ { T.value = F.value }

$F \longrightarrow num$ { 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.

$E \rightarrow E_1 + T$ { E.nptr = mknode(E_1.nptr, +, T.ptr);}

$E \rightarrow T$ { E.nptr = T.nptr }

$T \rightarrow T_1 * F$ { T.nptr = mknode(T_1.nptr, *, F.ptr);}

$T \rightarrow F$ { T.nptr = F.nptr }

$F \rightarrow$ 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.

N -> L {N.dval = L.dval}

L -> L_1 B {L.dval = L_1.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)**

$S \rightarrow id := E$ {gen (id.place = E.place;);}

$E \rightarrow E_1 + E_2$ {t = newtemp ( ); gen (t = El.place + E2.place;); E.place = t}

$E \rightarrow id$ {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) $t_1 = Y + Z; X = t_1$

(C) $t_1 = Y; t_2 = t_1 + Z; X = t_2$

(D) $t_1 = Y; t2 = Z; t_3 = t_1 + t_2; X = t_3$

Classification of Attributes

- Attributes in semantic analysis are classified based on the evaluation process into two categories:
- **Synthesized Attributes**:
  - These attributes derive their values from the attribute values of their **children** in the parse tree.
  - Commonly used to propagate information up the parse tree.
    - $A \rightarrow XYZ$ {A.S = f(X.S, Y.S, Z.S)}
  - Here, A.S (the synthesized attribute of A) is calculated based on the synthesized attributes of X, Y, and Z.
- **Inherited Attributes**:
  - These attributes derive their values from the attribute values of their **parents** or **left siblings** in the parse tree.
  - Useful for expressing dependencies on the context in which a construct appears, such as types or scopes.
    - $T \rightarrow$ int {T.type = integer}
    - $T \rightarrow$ double {T.type = double}
  - Here, T.type (an inherited attribute) is determined based on the semantic rule specified in the production.

S-Attributed SDT	L-Attributes SDT
Uses only Synthesized attributes	**Uses both inherited and synthesized attributes.** Inherited attributes can only inherit from the parent or left sibling.
Semantic actions are placed at the extreme right end of the production.	**Semantic actions can be placed anywhere** on the right-hand side of the production.
Attributes are evaluated during Bottom-Up Parsing (BUP).	**Attributes are evaluated during Depth-First Traversal** of the parse tree, from left to right.

- **S-Attributed SDT**:
  - Suitable for simpler grammar, where all attributes can be synthesized (i.e., derived from child nodes).
  - Works well with bottom-up parsers like **LR parsers**.
- **L-Attributed SDT**:
  - Handles both context-sensitive and context-free grammars due to its use of inherited attributes.
  - Can be used with **top-down parsers** like LL parsers.

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

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

Q Consider the following grammar and the semantic actions to support the inherited type declaration attributes. Let X_1, X_2, X_3, X_4, X_5 and X_6 be the placeholders for the non-terminals D, T, L or L_1 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 X_1, X_2, X_3 and X_4? **(Gate-2019) (2 Marks)**

a) $X_1 = L, X_2 = T, X_3 = L_1, X_4 = L$

b) $X_1 = L, X_2 = L, X_3 = L_1, X_4 = T$

c) $X_1 = T, X_2 = L, X_3 = L_1, X_4 = T$

d) $X_1 = T, X_2 = L, X_3 = T, X_4 = L_1$

Q. Consider the following Syntax-directed definition (SDD). Given "MMLK" as the input, which one of the following options is the CORRECT value computed by the SDD (in the attribute S.val)? **(Gate 2024,CS) (2 Marks) (MCQ)**

(a) 45

(b) 50

(c) 55

(d) 65

$S \rightarrow DHTU$	$\{ S.val = D.val + H.val + T.val + U.val; \}$
$D \rightarrow \text{"M"} D_1$	$\{ D.val = 5 + D_1.val; \}$
$D \rightarrow \epsilon$	$\{ D.val = -5; \}$
$H \rightarrow \text{"L"} H_1$	$\{ H.val = 5 * 10 + H_1.val; \}$
$H \rightarrow \epsilon$	$\{ H.val = -10; \}$
$T \rightarrow \text{"C"} T_1$	$\{ T.val = 5 * 100 + T_1.val; \}$
$T \rightarrow \epsilon$	$\{ T.val = -5; \}$
$U \rightarrow \text{"K"}$	$\{ U.val = 5; \}$

Q. Which of the following statements is/are FALSE? **(Gate 2024 CS) (1 Mark) (MSQ)**

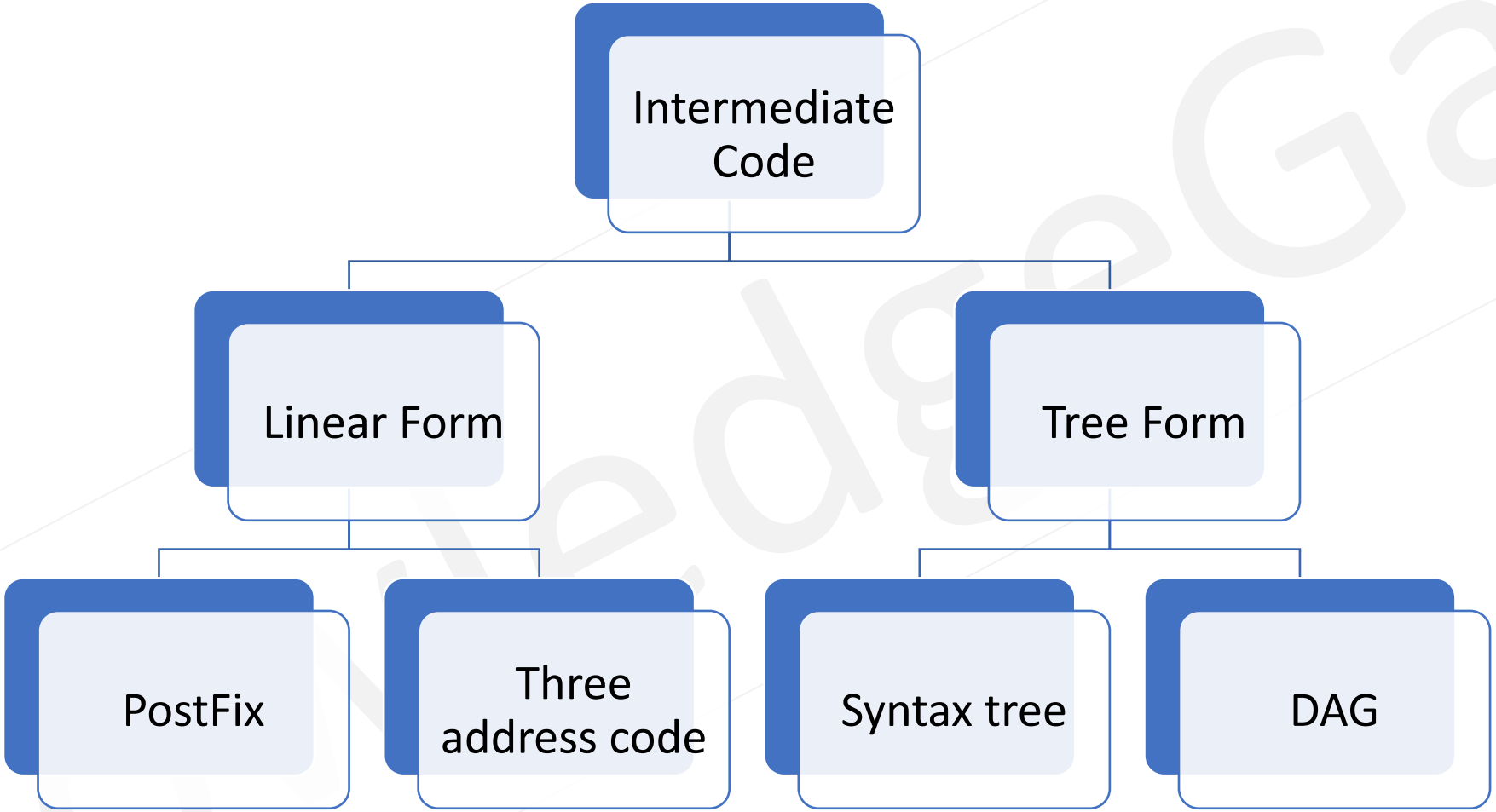
(a) An attribute grammar is a syntax-directed definition (SDD) in which the functions in the semantic rules have no side effects

(b) The attributes in a L-attributed definition cannot always be evaluated in a depth first order

(c) Synthesized attributes can be evaluated by a bottom-up parser as the input is parsed

(d) All L-attributed definitions based on LR(1) grammar can be evaluated using a bottom-up parsing strategy

Intermediate Code Generation



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

Post fix

ab+ab+c+*

Three address code

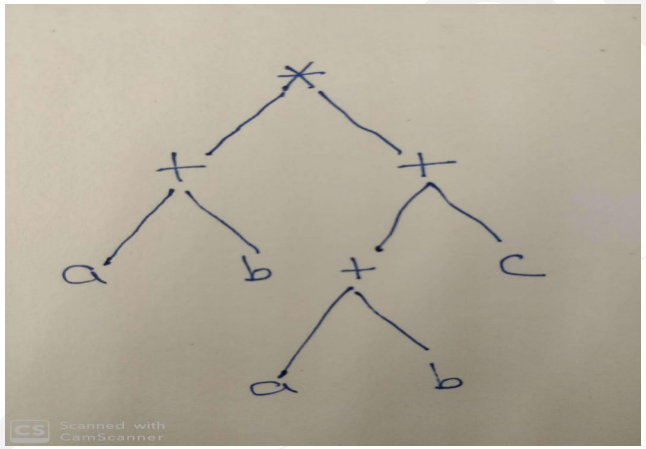
$t_1 = a+b$

$t_2 = a+b$

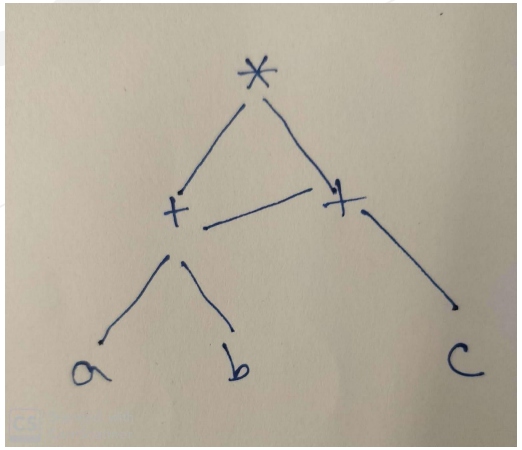
$t_3 = t_2 + c$

$t_4 = t_1 * t_3$

Syntax Tree



Direct Acyclic Graph



((a+a) + (a+a)) + ((a+a) + (a+a))

Q Consider the following C code segment:

```
a = b + c;
e = a + 1;
d = b + c;
f = d + 1;
g = e + f
```

In a compiler, this code segment is represented internally as a directed acyclic graph (DAG). The number of nodes in the DAG is _____. **(GATE 2021) (2 MARKS)**

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) $t_7 = t_4 + t_6$

Q. Refer to the given 3-address code sequence. This code sequence is split into basic blocks. The number of basic blocks is _____. (Answer in integer) **(Gate 2025)**

```
1001: i = 1
1002: j = 1
1003: t1 = 10 * i
1004: t2 = t1 + j
1005: t3 = 8 * t2
1006: t4 = t3 – 88
1007: a[t4] = 0.0
1008: j = j + 1
1009: if j <= 10 goto 1003
1010: i = i + 1
1011: if i <= 10 goto 1002
1012: i = 1
1013: t5 = i – 1
1014: t6 = 88 * t5
1015: a[t6] = 1.0
1016: i = i + 1
1017: if i <= 10 goto 1013
```

Quadruples

	Operator	$Operand_1$	$Operand_2$	Result
1)				
2)				
3)				
4)				
5)				
6)				
7)				

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) $t_7 = t_4 + t_6$

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

Quadruples

	Operator	$Operand_1$	$Operand_2$	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

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) $t_7 = t_4 + t_6$

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

Triplet

	Operator	$Operand_1$	$Operand_2$
1)			
2)			
3)			
4)			
5)			
6)			
7)			

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) $t_7 = t_4 + t_6$

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

Triplet

	Operator	$Operand_1$	$Operand_2$
1)	+	a	b
2)	-	1	
3)	+	c	d
4)	*	2	3
5)	+	a	b
6)	+	5	c
7)	+	4	6

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) $t_7 = t_4 + t_6$

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

Indirect Triplet

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+1) t=0
i+2) goto (____)
i+3) t=1
i+4) exit

while(C) do S

i) if (C) goto ____
i+1) goto ____
i+2) S
i+3) goto ____
i+4) exit

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

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

Q In the context of compilers, which of the following is/are NOT an intermediate representation of the source program?**(GATE 2021) (1 MARKS)**

(a) Three address code

(b) Abstract Syntax Tree (AST)

(c) Control Flow Graph (CFG)

(d) Symbol table

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.

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

Static single assignment form

- In compiler design, the **Static Single Assignment (SSA) form** is a property of an intermediate representation (IR) where:
  - Each variable is **assigned exactly once**.
  - Every variable must be **defined before it is used**.
- **Key Features:**
  - Variables in the original IR are split into **versions**.
  - These versions are typically denoted by subscripts (e.g., x1, x2) to ensure each definition gets its unique version.
- **Advantages of SSA Form:**
  - Simplifies data flow analysis.
  - Enhances compiler optimizations such as **constant propagation** and **dead code elimination**.
  - Makes dependency tracking between variables easier.

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)
p_1 = a - b	p_3 = a - b	p_1 = a - b	p_1 = a - b
q_1 = p1 * c	q_4 = p3 * c	q_1 = p2 * c	q_1 = p * c
p_1 = u * v	p_4 = u * v	p_3 = u * v	p_2 = u * v
q_1 = p1 + q1	q_5 = p4 + q4	q_2 = p4 + q3	q_2 = p + q

Q Consider the following code segment.

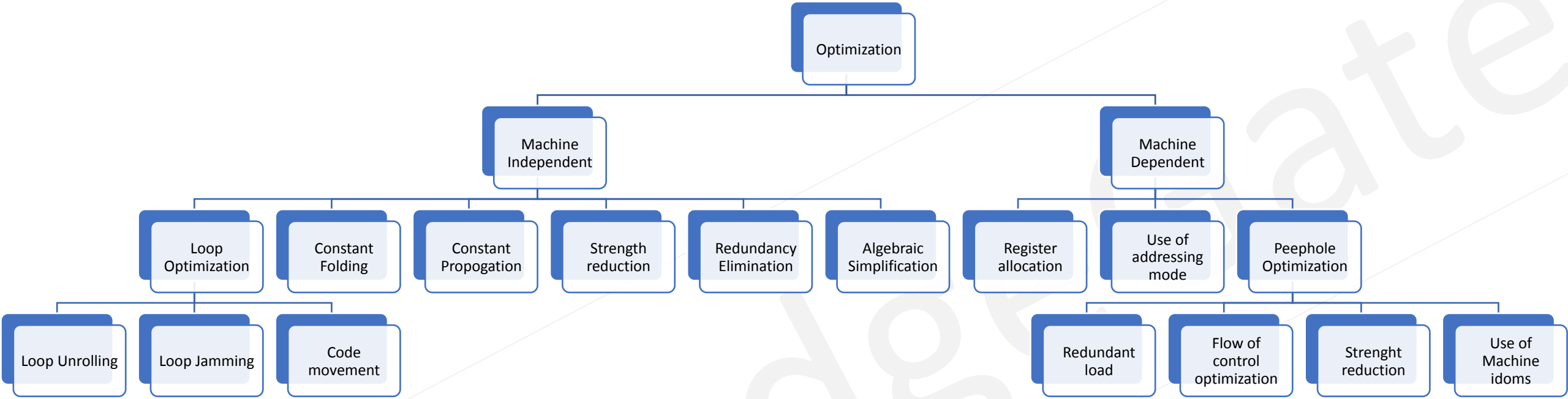
```
x = u - t;
y = x * v;
x = y + w;
y = t - z;
y = x * y;
```

The minimum number of variables required to convert the above code segment to static single assignment form is _____. **(Gate - 2016) (2 Marks)**

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)**

Optimization

- Optimization in compiler design is the process of **reducing the execution time** of a code **without altering the functionality** or outcome of the source program.
- **Key Objective:**
  - Improve program performance by minimizing computational resources such as **time** and **memory usage** while ensuring the program's semantics remain unchanged.



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

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

a = b*1

a=b

a=b+0

a=b

Q. Which ONE of the following techniques used in compiler code optimization uses live variable analysis? **(Gate 2025)**

A) Run-time function call management
B) Register assignment to variables
C) Strength reduction
D) Constant folding

Redundant code Elimination / Common subexpression 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
```

- **Loop Optimization:** Loop optimization is a critical step in improving program performance by enhancing execution speed and minimizing computational effort. The process involves several steps, starting with **loop detection**.
- **Steps for Loop Optimization:**
  - **Detecting Loops**:
    - Loops are identified using **Control Flow Analysis (CFA)**, which is performed using a **Program Flow Graph (PFG)**.
    - The PFG is constructed by first identifying **basic blocks** in the program.
  - **Basic Block**:
    - A **basic block** is a sequence of three-address statements where control enters only at the beginning and exits only at the end (without any jumps or halts in between).
  - **Identifying Basic Blocks**: To find basic blocks, **leaders** in the program are identified. A **basic block** starts from one leader and extends up to (but does not include) the next leader.
  - **Identifying Leaders**: A leader is defined as:
    - The **first statement** in the program.
    - A statement that is the **target of a conditional or unconditional jump**.
    - A statement that **immediately follows a conditional or unconditional jump**.
  - **Constructing the Program Flow Graph**:
    - Once the basic blocks are identified, a PFG is created to visualize the control flow between these blocks.

- **Benefits of Loop Optimization:**
  - Improves execution time by reducing redundant computations.
  - Streamlines control flow, resulting in faster and more efficient code.

```
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

Q Consider the intermediate code given below:

1. i = 1
2. j = 1
3. $t_1 = 5 * i$
4. $t_2 = t_1 + j$
5. $t_3 = 4 * t_2$
6. $t_4 = t_3$
7. $a[t_4] = -1$
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

Q. Consider the following Pseudo-code. Which one of the following options CORRECTLY specifies the number of basic blocks and the number of instructions in the largest basic block, respectively? **(Gate 2024,CS) (2 Marks) (MCQ)**

(a) 6 and 6

(b) 6 and 7

(c) 7 and 7

(d) 7 and 6

$$
\begin{aligned}
L1:&\quad t1 = -1\\
L2:&\quad t2 = 0\\
L3:&\quad t3 = 0\\
L4:&\quad t4 = 4 * t3\\
L5:&\quad t5 = 4 * t2\\
L6:&\quad t6 = t5 * M\\
L7:&\quad t7 = t4 + t6\\
L8:&\quad t8 = a[t7]\\
L9:&\quad \text{if } t8 <= max \text{ goto } L11\\
L10:&\quad t1 = t8\\
L11:&\quad t3 = t3 + 1\\
L12:&\quad \text{if } t3 < M \text{ goto } L4\\
L13:&\quad t2 = t2 + 1\\
L14:&\quad \text{if } t2 < N \text{ goto } L3\\
L15:&\quad max = t1
\end{aligned}
$$

Q. Consider the following statements about the use of backpatching in a compiler for intermediate code generation:

(I) Backpatching can be used to generate code for Boolean expression in one pass.

(II) Backpatching can be used to generate code for flow-of-control statements in one pass.

Which ONE of the following options is CORRECT? **(Gate 2025)**

A) Only (I) is correct
B) Only (II) is correct.
C) Both (I) and (II) are correct.
D) Neither (I) nor (II) is correct.

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

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

Liveness Analysis

- **Definition**: Liveness analysis is a type of **data-flow analysis** used in compilers to determine which variables are "live" at any point in a program.
- **Key Concept**:
  - A variable is considered **live** at a specific point if:
    - It holds a value that may be required later.
    - Its value may be **read** before it is **overwritten** with a new value.
- **Purpose**:
  - To optimize the program by identifying variables that need to be kept in memory and those that can be safely discarded or overwritten.
- **Applications**:
  - **Register allocation**: Helps decide which variables should stay in registers.
  - **Code optimization**: Eliminates unnecessary computations or assignments.
  - **Dead code elimination**: Identifies and removes variables or operations that do not affect the program's output.

	D	E	F	A	B	C
1						
2						
3						
4						
5						
6						
7						
8						

- B = 1
- F = B + 1
- D = F * B
- E = D * 2
- C = D + F * E
- A = F + 2
- B = A + C + E
- Return(B + A)

Q For a statement S in a program, in the context of liveness analysis, the following sets are defined:

- **USE(S) :** the set of variables used in S
- **IN(S) :** the set of variables that are live at the entry of S
- **OUT(S) :** the set of variables that are live at the exit of S

Consider a basic block that consists of two statements, S_1 followed by S_2. Which one of the following statements is correct? **(GATE 2021) (2 MARKS)**

(a) $OUT(S_1) = IN(S_2)$

(b) $OUT(S_1) = IN(S_1) \cup USE(S_1)$

(c) $OUT(S_1) = IN(S_2) \cup OUT(S_2)$

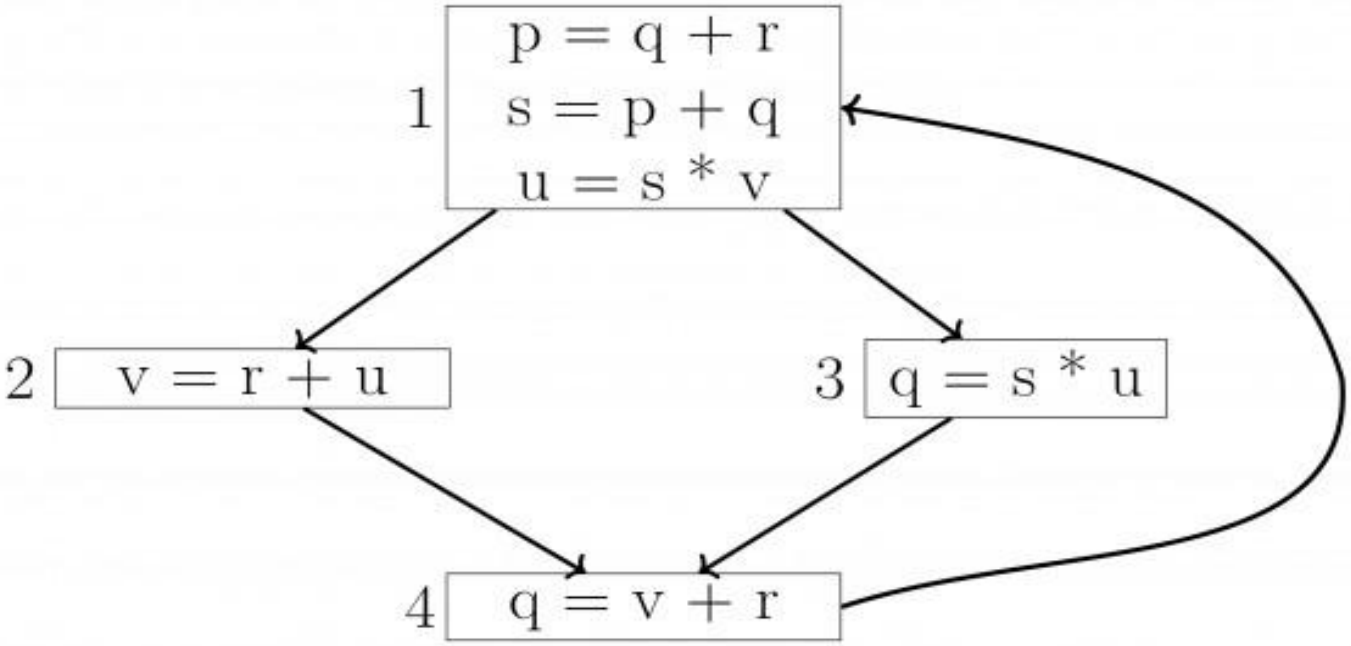
(d) $OUT(S_1) = USE(S_1) \cup IN(S_2)$

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

i. There exists a statement Sj that uses x

ii. There is a path from Si to Sj in the flow graph corresponding to the program

iii. The path has no intervening assignment to x including at Si and Sj



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 **(GATE-2015) (2 Marks)**

a) p, s, u **b)** r, s, u

c) r, u **d)** q, v

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

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

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

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

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

Q Consider the grammar rule $E \rightarrow E_1 - E_2$ 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) E_1 should be evaluated first

(B) E_2 should be evaluated first

(C) Evaluation of E_1 and E_2 should necessarily be interleaved

(D) Order of evaluation of E_1 and E_2 is of no consequence

Q. Consider the following expression: $x[i] = (p + r) * -s[i] + u/w$. The following sequence shows the list of triples representing the given expression, with entries missing for triples (1), (3), and (6). **(Gate 2024 CS) (2 Marks) (MCQ)**

Which one of the following options fills in the missing entries CORRECTLY?

(a) (1) =[] s i (3) * (0) (2) (6) []= x i

(b) (1) []= s i (3) – (0) (2) (6) =[] x (5)

(c) (1) =[] s i (3) * (0) (2) (6) []= x (5)

(d) (1) []= s i (3) – (0) (2) (6) =[] x i

(0)	+	p	r
(1)			
(2)	uminus	(1)	
(3)			
(4)	/	u	w
(5)	+	(3)	(4)
(6)			
(7)	=	(6)	(5)

Q Consider the following ANSI C code segment:

```
z=x + 3 + y□f1 + y □ f2;
for (i = 0; i < 200; i = i + 2)
{
        if (z > i)
        {
                p = p + x + 3;
                q = q + y □ f1;
        }
        else
        {
                p = p + y □ f2;
                Q= q + x + 3;
}
```

} Assume that the variable y points to a struct (allocated on the heap) containing two fields f_1 and f_2, and the local variables x, y, z, p, q, and i are allotted registers. Common sub-expression elimination (CSE) optimization is applied on the code. The number of addition and the dereference operations (of the form y □ f_1 or y □ f_2) in the optimized code, respectively, are:**(GATE 2021) (2 MARKS)**

(a) 403 and 102

(b) 203 and 2

(c) 303 and 102

(d) 303 and 2