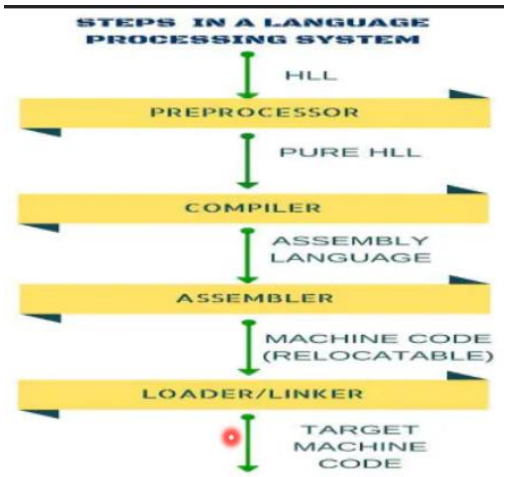
LANGUAGE PROCESSING SYSTEM

Users code in a high-level language while the machine can understand only low-level machine language. To convert HLL to LLL, we need to use a Language Processing system –



Compiler is present in the Language Processing system which will change the pure HLL to Assembly level language. Let us understand this chain of processing –

1. First, we write a code in HLL like C.
2. Next, we send this code to the pre-processor. In this stage, the pre-processor will execute the pre-processor directive (#) and will convert the code to pure HLL. Basically, pre-processor will import, include and execute statements like `#include<stdio.h>`, `#include<stdlib.h>` etc. Pre-processor also will replace short-hand operators like ++, --, += etc. to their standard form.
3. Next, the pure HLL will be sent to the compiler. This is where the code will be converted to Assembly level language. The assembly level language is low level and provides the information as to how is the data stored in which registers.
4. After this comes the assembler that will take the assembly language code and convert it to the lowest level machine code.
5. Finally, the linker will take all the modules of the machine codes and then link them together. Then, this linked code will be loaded and the code will be executed.

COMPILER vs INTERPRETER

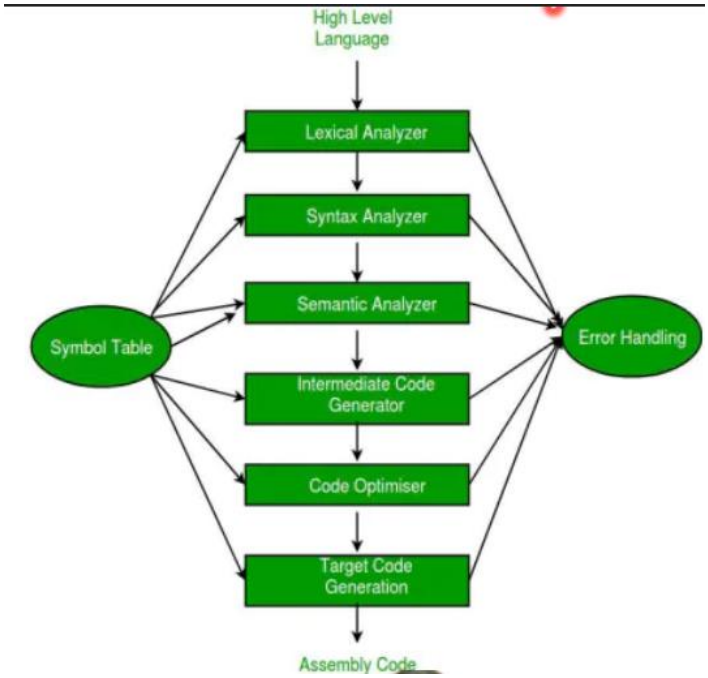
Another alternative to compiler is an interpreter. Interpreter will take the HLL code line by line, processes it to a lower-level language and gets it executed. Then, it moves to the next line. Thus, compared to the compiler an interpreter will not store any intermediate/assembly code. Thus, there is no need for a loader/linker.

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.

COMPILER BREAKDOWN

A compiler consists of the following steps –



Here, the lexical analyzer, syntax analyzer and semantic analyzer are present in the “front-end” while the rest of the blocks are present in the “back-end”.

Lexical Analyzer

This is a part of the compiler that reads a program and converts it into **Lexemes** (words/tokens). It also removes white – spaces and comments. For example, let us assume the following code –

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

In this case, the lexical analyzer will divide the code line into the following tokens –

X → token → identifier
= → token → operator
Y → token → identifier
+ → token → operator
Z → token → identifier
* → token → operator
60 → token → constant

Syntax Analyzer (Parser)

A parser takes the tokens from the lexical analyzer and creates a parse tree. This parse tree is formed with the help of the Context-free grammar that is provided for the system.

Semantic Analyzer

This part checks the parse tree and checks whether it is a meaningful or not. It checks the semantics and grammar to check if it is meaningful or not.

Intermediate Code Generator

This generates an intermediate code which is a form that can be readily understood by the machine. The intermediate code is **the same for all compilers** and only the last two steps are platform dependent. This means that we can take the intermediate code from any compiler and run it on any other compiler.

Code Optimizer

This is used to transform the code to be more optimized such that is consumes lesser resources and produces more speed. The optimization can be either machine dependent or machine independent.

Target Code Generator

This is the final step which returns the final assembly code. The assembly code being generated is dependent on the type of assembler.

LEXICAL ANALYZER

Lexical analysis (also called tokenization or lexing) is a process of converting a sequence of characters into a sequence of tokens. A sequence of characters in the input string that matches the pattern of a token is called a **lexeme.** A Lexical analyzer (Lexer) will take an input string and yield output in the form of tuples –

$$(\boldsymbol{Token}, \boldsymbol{Lexeme})$$

For example, let us take the code line as follows –

```
int x = a*b;
```

The Lexer will return the output as follows –

- (datatype, int)
- (identifier, x)
- (operator, =)
- (identifier, a)
- (operator, *)
- (identifier, b)
- (separator, ;)

Apart from the above function, the lexer also is responsible for –

- Removing comments
- Removing Whitespaces
- Co-relating the errors with the code line number.

Lexer Implementation

To implement a lexer, we need to define the **Lexical Grammar.** The lexer will recognize and work on strings based on the rules of the Lexical Grammar. For example, let us assume that the grammar of a password is given as follows –

$$Regular\ Expression \rightarrow A(A + S + D)^7(A + S + D + \epsilon)^7$$

Where, $A \rightarrow alpha, S \rightarrow special\ symbol, D \rightarrow Digit$. Thus, the rules of the password are as follows –

- The password must start with an alphabet.
- After that, the next characters can be either alphabet, digit or special character
- Minimum length of the password should be 8 (1+7)
- Maximum length of the password should be 15 (1+7+7)

Question

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

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

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

a) $T_1T_2T_3$

b) $T_1T_1T_3$

c) $T_2T_1T_3$

d) T_3T_3

Answer

We can re-write the token expressions as follows –

$$T1 = (a + \epsilon)(b + c)^* a$$

$$T2 = (b + \epsilon)(a + c)^* b$$

$$T3 = (c + \epsilon)(b + a)^* c$$

Now, the string we have is $\boldsymbol{bbaacabc}$**.** If we put this string through the different token expressions, we can see that –

$$T1 \rightarrow bba$$

$$T2 \rightarrow bb$$

$$T3 \rightarrow bbaac$$

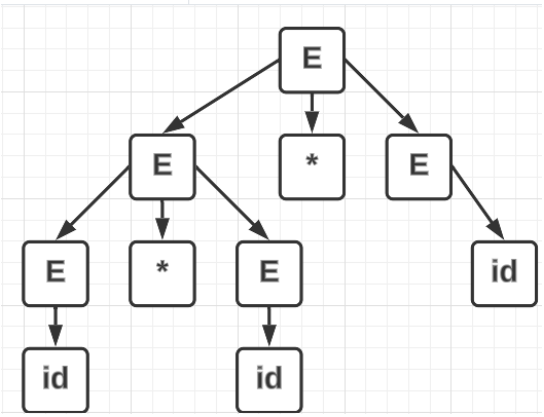
As the question instructs to take the token expression with the longest prefix, we must take T3. Hence, the correct option is **Option D.**

LEXICAL GRAMMAR

Lexical grammar is usually a CFG. CFG is a Type – 2 grammar wherein any production of the form $\alpha \rightarrow \beta$ will have α as a **variable** while β can be anything. The process of deriving a string from the grammar is called **derivation** and the graphical representation of the derivation is called a **Parse (Syntax) Tree.** For example, let us take the grammar as follows –

$$E \rightarrow E + E \mid E * E \mid E = E \mid id$$

If we want to create the string – $E = id + id * id$. Then, the parse tree will be –



If we mention the intermediate steps involved in the derivation, then it is called the **Sentential form.**

Sentential Form
E
E*E
E+E*E
ID+E*E
ID+ID*E
ID+ID*ID

In every step of the sentential form, if the extreme left term is terminated, then it called **Left Most Derivation.** On the other hand, if every step has the extreme right term terminated, then it is called **Right Most Derivation.** The above sentential form is the LMD. To get the RMD for the same tree, we can write –

RMD
E
E+E
E+E*E
E+E*ID
E+ID*ID
ID+ID*ID

NON – DETERMINISTIC GRAMMAR

If a production has **common prefix,** then it is called a Non-Deterministic Grammar as it is not possible to have a deterministic way to derive the string. For example,

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$$

Now, we know that first symbol is α but we don't know which will be the next symbol. To change it to deterministic grammar, we can write it as follows –

$$A \rightarrow \alpha B$$

$$B \rightarrow \beta_1 \mid \beta_2$$

In short, a parser using a Non-deterministic grammar will have to backtrack at some point. To avoid backtracking, we need to convert the Non-Deterministic grammar to a Deterministic Grammar.

NOTE

- Deterministic Grammar is also called **Left Factored Grammar.**

- Non-Deterministic Grammar is also called **Non-Left Factored Grammar**
- The process to convert NDG to DG is called **Left Factoring.**

Question

Convert the following grammar to Left Factored Grammar

$$S \rightarrow aSb \mid abS \mid ab$$

Answer

$$S \rightarrow aA$$

$$A \rightarrow Sb \mid bB$$

$$B \rightarrow S \mid \epsilon$$

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

If the variable on the LHS appears on the RHS as well, it is called **recursive grammar.** In addition to that, if the recursive variable appearing on the RHS is at the extreme left, then it is called **Left recursive grammar.** If the recursive variable appears on the extreme right, then it is called **Right recursive grammar.**

$$S \rightarrow Sa \ (Left\ RG)$$

$$S \rightarrow aS \ (Right\ RG)$$

$$S \rightarrow aSb \ (General\ RG)$$

The expressive power of Left RG and Right RG is the **same.**

NOTE

Normally, a Left RG can result in an infinite loop so it is usually a good thing to first change Left RG to Right RG.

Question

Convert the Left RG to Right RG.

$$A \rightarrow A\alpha \mid \beta_1 \mid \beta_2$$

Answer

$$A \rightarrow \beta_1 B \mid \beta_2 B$$

$$B \rightarrow \alpha B \mid \epsilon$$

Question

Convert the Left RG to Right RG.

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \beta$$

Answer

$$A \rightarrow \beta B$$

$$B \rightarrow \alpha_1 B \mid \alpha_2 B \mid \epsilon$$

Question

Convert the Left RG to Right RG.

$$S \rightarrow SSS \mid 0$$

Answer

$$S \rightarrow 0A$$

$$A \rightarrow SSA \mid \epsilon$$

Question

Convert the Left RG to Right RG.

$$S \rightarrow S1S \mid 0$$

Answer

$$S \rightarrow 0A$$

$$A \rightarrow 1SA \mid \epsilon$$

Question

Convert the Left RG to Right RG.

$$S \rightarrow S12 \mid 0$$

Answer

$$S \rightarrow 0A$$

$$A \rightarrow 12A \mid \epsilon$$

Question

Convert the Left RG to Right RG.

$$S \rightarrow S0S1 \mid 0 \mid 1$$

Answer

$$S \rightarrow 0A \mid 1A$$

$$A \rightarrow 0S1A \mid \epsilon$$

Question

Convert the Left RG to Right RG.

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

Answer

$$E \rightarrow TA$$

$$A \rightarrow +TA \mid \epsilon$$

$$T \rightarrow FB$$

$$B \rightarrow *FB \mid \epsilon$$

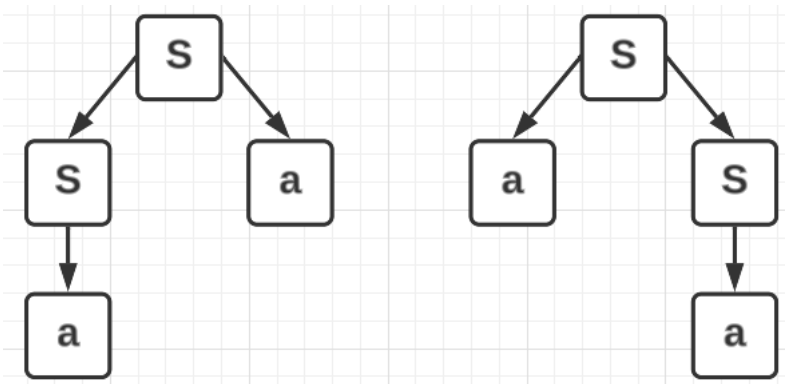
$$F \rightarrow (E) \mid id$$

AMBIGUOUS GRAMMAR

If a CFG exists where any string can have more than one derivation tree then it is said to be **ambiguous.** For example,

$$S \rightarrow aS \mid Sa \mid a$$

In this case, we can draw two derivation graphs as follows –



In this case, both the graphs result in the same string $\boldsymbol{aa}$ and hence, the above grammar is **ambiguous.** We can write the same grammar as –

$$S \rightarrow aS \mid a$$

Now the grammar is **unambiguous.** Here are a few things to note about ambiguous grammar –

- If a grammar has both left and right recursive, then it will be **ambiguous.** However, the converse is not true and an ambiguous grammar need not have both left and right recursive.
- There are some ambiguous grammars that can't be converted to unambiguous grammar. For example, $\{a^n b^m c^m d^n\} \cup \{a^n b^n c^m d^m\}$ is a CFL which doesn't have any unambiguous way to derive.

Question

Is the following grammar ambiguous or unambiguous?

$$S \rightarrow SS \mid 0 \mid 1$$

Answer

Since there are both left and right recursive grammar, the grammar is **ambiguous.**

MINIMIZATION OF CFG

We can follow the given steps to simplify/minimize the given CFG –

- Removal of NULL productions
- Removal of Unit productions
- Removal of Useless symbols

Removal of NULL Productions

A NULL production is of the form $A \rightarrow \epsilon$. During derivation, if there is a null production then it replaces a non-terminal with ϵ and hence we will need to derive the string further to get the required string. In short, a NULL production **doesn't functionally add** to the string derivation. Therefore, it is advisable to remove NULL productions to simplify the CFG. For example, let us take the CFG as follows –

S → AbB
A → a / ∈
B → b / ∈

If $A \rightarrow \epsilon$, then $S \rightarrow bB$. Similarly, if $B \rightarrow \epsilon$, then $S \rightarrow Ab$. Finally, if both $A \rightarrow \epsilon$ and $B \rightarrow \epsilon$, then $S \rightarrow b$. Thus, we get –

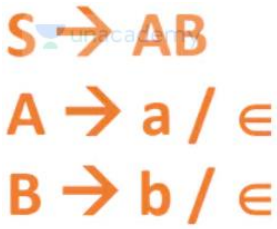
$$S \rightarrow AbB \mid bB \mid Ab \mid b$$

$$A \rightarrow a$$

$$B \rightarrow b$$

Question

For the given CFG, remove the NULL productions.



Answer

$$S' \rightarrow S \cup \{\epsilon\}$$

$$S \rightarrow AB \mid B \mid A$$

$$A \rightarrow a$$

$$B \rightarrow b$$

Removal of Unit Productions

A production where there is a single terminal or variable on the RHS is called a Unit production. A unit production is a waste and instead can be included in any of the previous productions itself. For example –

$$S \rightarrow Aa$$

$$A \rightarrow a \mid B$$

$$B \rightarrow d$$

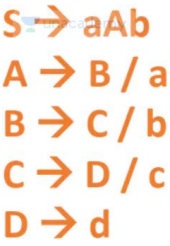
Here, $B \rightarrow d$ is a unit production. Thus, we can remove it as follows –

$$S \rightarrow Aa$$

$$A \rightarrow a \mid d$$

Question

Remove the unit productions in the given CFG –



Answer

$$S \rightarrow aAb$$

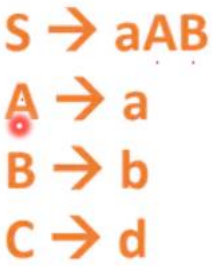
$$A \rightarrow d \mid c \mid b \mid a$$

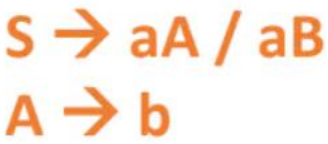
Removal of Useless Symbols

These are analogous to the dead and unreachable states in FA. There can be two types of useless productions –

- Which can't be reached by the start symbol (Unreachable)
- Which don't derive any terminal and hence can't be terminated (Dead)

For example –

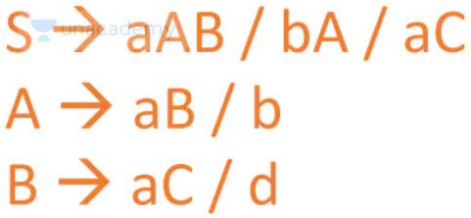




In the first CFG, $C \rightarrow d$ is an unreachable production and hence can be removed. In the second CFG, the variable B can be reached from S but it doesn't derive any strings/terminals. So, that is also useless.

Question

Remove the useless productions –



Answer

$$S \rightarrow aAB \mid bA$$

$$A \rightarrow aB \mid b$$

$$B \rightarrow d$$

NORMALIZATION OF CFG

This is done to transform the CFG to be more compiler friendly. We have 2 normal forms –

- Chomsky Normal Form (CNF)
- Greiback Normal Form (GNF)

Both CNF and GNF have the same expressive power. Also, every CFG can be expressed in the CNF and GNF form. Additionally, to perform normalization we need the grammar to be in its minimized form.

Chomsky Normal Form (CNF)

Every production in CNF must be of the form –

$$\boldsymbol{A \rightarrow BC \mid a}$$

Where A, B, C are variables and a is a terminal. For example, let us take the CFG as follows –

$$S \rightarrow aSb \mid ab$$

For a CNF, we can have either 2 variables or 1 terminal in the RHS. So, we can write the CFG as follows –

$$S \rightarrow AS' \mid AB$$

$$S' \rightarrow SB$$

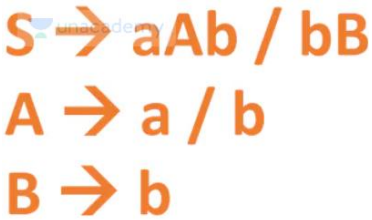
$$A \rightarrow a$$

$$B \rightarrow b$$

We can see that the above grammar is in CNF 😊

Question

Convert to CNF form –



Answer

$$S \rightarrow S'B \mid BB$$

$$S' \rightarrow CA$$

$$A \rightarrow a \mid b$$

$$B \rightarrow b$$

$$C \rightarrow a$$

Suppose we need to create a string $\boldsymbol{aab}$ using the above grammar, then we can write the sentential forms as follows –

- $S \rightarrow S'B$
- $S \rightarrow CAB$
- $S \rightarrow aAB$

- $S \rightarrow aaB$
- $S \rightarrow aab$

Thus, we can see that to generate a string of length **3,** we have **5** sentential forms. Therefore, we can conclude that in general for CNF, if we need to generate a string of length $\boldsymbol{n}$**,** we would have a total of $\boldsymbol{2n-1}$ sentential forms.

Greiback Normal Form

Here, every production needs to be of the form –

$$A \rightarrow a\alpha$$

Where A is a variable, a is a terminal and $\alpha \in V_n^*$. For example, let us take the CFG as follows –

$$S \rightarrow aSb \mid ab$$

Then, we can write GNF as follows –

$$S \rightarrow aSB \mid aB$$

$$B \rightarrow b$$

We can see that the above grammar is in GNF 😊

Question

Convert to CNF form –

S → aAb / bB

A → a / b

B → b

Answer

$$S \rightarrow aAB \mid bB$$

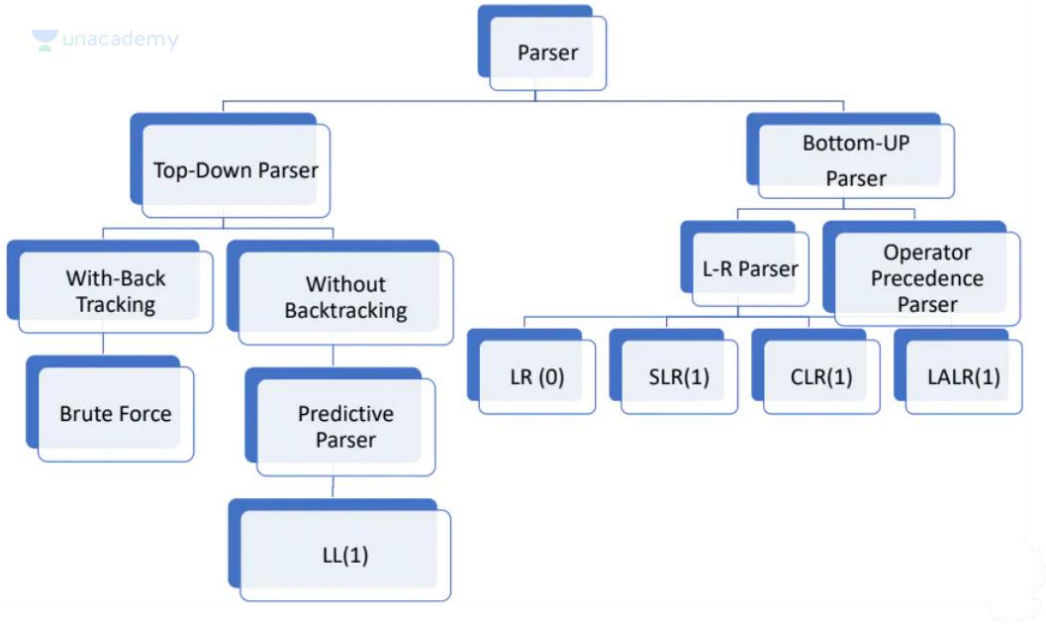
$$A \rightarrow a \mid b$$

$$B \rightarrow b$$

Unlike CNF, if we need to generate a string of length $\boldsymbol{n}$ in a GNF grammar, then we need $\boldsymbol{n}$ sentential forms.

SYNTAX ANALYZER (PARSER)

This is the next step after the Lexical analysis. Parsers can be of various types –



TOP-DOWN PARSERS

Top-down parsers start parsing the input, and start constructing a parse tree from the root node gradually moving down to the leaf nodes. These use **left – most derivations** for construction. For any top-down parser, the CFG from which it is constructed must be –

- Non-left recursive grammar (as the parser might go into an infinite loop)
- Unambiguous

The top-down parsers can be of 2 types –

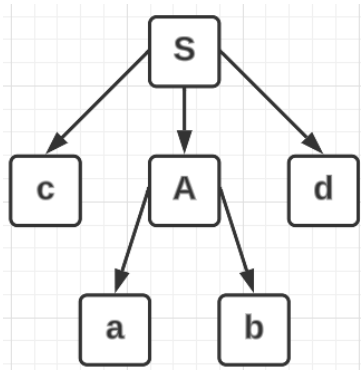
- Brute Force (requires back tracking)
- Predictive Parser (doesn't support back tracking)

Brute Force Parsers

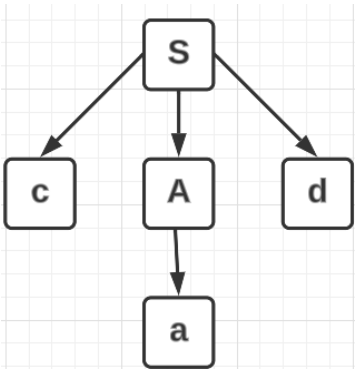
Brute force parsers are simple…you check a string by trial and error. If the string doesn't match in the CFG, then back track and try another method. It is a basic but time consuming and long processes. These can be constructed from **both deterministic and non-deterministic CFG.** For example, let us take the CFG as follows –

$$S \rightarrow cAd$$
$$A \rightarrow ab \mid a$$

Now, suppose we need to construct a parse tree for the word $\boldsymbol{w = cad}$**.** Then, we can start as follows –



We got this using the left – most derivation. However, this is not the word we want. Hence, we now **backtrack** and can re-write the parse tree as follows –



Question

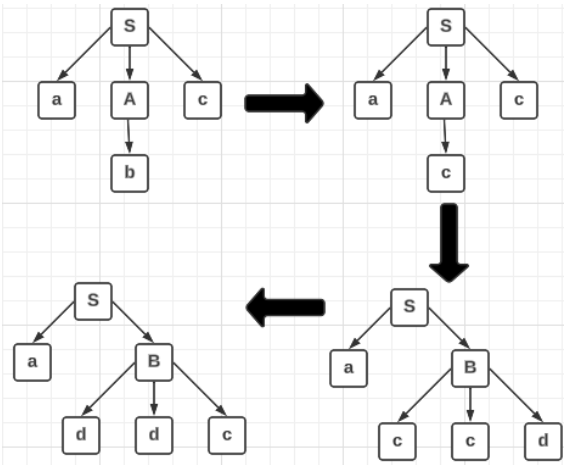
Draw the parse tree from a Brute force parser to select the word $\boldsymbol{w = addc}$ using the following CFG –

S → aAc / aB

A → b / c

B → ccd / ddc

Answer



Since Brute force is taking a lot of time, it has a time complexity of $\boldsymbol{O(2^n)}$**.**

Predictive Parsers

A **predictive parser** will use rules to predict the next incoming symbol. This parser doesn't support back tracking. Therefore, it can only be constructed from **deterministic grammar.** Since this parser doesn't have the option of back tracking, we can't have any scope for error. Thus, before beginning we need to understand 2 functions –

- **FIRST (X) –** It is a function that gives the set of terminals that **begin** the strings derived from the production rule. In other words, it returns the set of terminals with which the string $\boldsymbol{X}$ can start with.
- **FOLLOW (X) –** It returns a set of terminals that will be achieved when $\boldsymbol{X = \epsilon}$**.** FOLLOW function can only be applied on variables and **not on terminals.**

For example, suppose we have the grammar –

$$S \rightarrow ABC$$

$$A \rightarrow ab$$

$$B \rightarrow ca$$

$$C \rightarrow b$$

Then, we can write –

$$FIRST(abc) = a$$

$$FIRST(cba) = c$$

$$FIRST(A) = a\ ;\ FOLLOW(A) = \{c\}$$

$$FIRST(B) = c\ ; FOLLOW(B) = b$$

$$FIRST(S) = a\ ; FOLLOW(S) = \{\$\}$$

Question

Find the First and Follow for the following CFG –

$$S \rightarrow a \mid b \mid \epsilon$$

Answer

$$FIRST(S) = \{a, b, \epsilon\}$$

$$FOLLOW(S) = \{\$\}$$

Question

Find the First and Follow for the following CFG –

S → aA / bB

A → ε

B → ε

Answer

$$FIRST(S) = \{a, b\}$$

$$FIRST(A) = \{\epsilon\}$$

$$FIRST(B) = \{\epsilon\}$$

$$FOLLOW(S) = \{\$\}$$

$$FOLLOW(A) = \{\$\}$$

$$FOLLOW(B) = \{\$\}$$

Question

Find the First and Follow for the following CFG –

S → aAb / Ba

A → aA / b

B → c / d

Answer

$$FIRST(S) = \{a, c, d\}$$

$$FIRST(A) = \{a, b\}$$

$$FIRST(B) = \{c, d\}$$

$$FOLLOW(S) = \{\$\}$$

$$FOLLOW(A) = \{b\}$$

$$FOLLOW(B) = \{a\}$$

Question

Find the First and Follow for the following CFG –

S → AaB / BA

A → a / b

B → d / e

Answer

$$FIRST(S) = \{a, b, d, e\}$$

$$FIRST(A) = \{a, b\}$$

$$FIRST(B) = \{d, e\}$$

$$FOLLOW(S) = \{\$\}$$

$$FOLLOW(A) = \{\$, a\}$$

$$FOLLOW(B) = \{\$, a, b\}$$

Question

Find the First and Follow for the following CFG –

S → AaB
A → b / ε
B → c

Answer

$$FIRST(S) = \{b, a\}$$

$$FIRST(A) = \{b, \epsilon\}$$

$$FIRST(B) = \{c\}$$

$$FOLLOW(S) = \{\$\}$$

$$FOLLOW(A) = \{a\}$$

$$FOLLOW(B) = \{\$\}$$

Question

Find the First and Follow for the following CFG –

S → aAB / BA
A → BA / ε
B → AB / a / ε

Answer

$$FIRST(S) = \{a, \epsilon\}$$

$$FIRST(A) = \{a, \epsilon\}$$

$$FIRST(B) = \{a, \epsilon\}$$

$$FOLLOW(S) = \{\$\}$$

$$FOLLOW(A) = \{a, \$\}$$

$$FOLLOW(B) = \{a, \$\}$$

NOTE

We can easily find the first of an element, but FOLLOW seems to be a little weird. So here is a cheat sheet –

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

2) if A → αAβ, β →! ∈
   Follow(A) = First(β)

3) if S → αA
   Follow(A) = Follow(S)

4) S → αAβ, where β → ∈
   Follow(A) = First(β) U Follow(S) - ∈

LL1 PARSER

First and foremost, we need to have a CFG that is non-left recursive, unambiguous and deterministic. Once we have that, we can proceed with the LL1 parser derivation. Let us take an example of a CFG as follows –

$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid \epsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow * FT' \mid \epsilon$$

$$F \rightarrow (E) \mid id$$

This CFG is non-left recursive, unambiguous and deterministic. Now, let us assume that we need to make the parse tree for the string –

$$W = id + id * id$$

From the grammar, we can observe that –

$$Terminals = \{+, *, (,), id\}$$

$$Variables = \{E, E', T, T', F\}$$

We now draw a table with these values as shown below –

	$*$	$+$	$($	$)$	$\boldsymbol{id}$	$
E						
E'						
T						
T'						
F						

From the grammar, we can see that –

$$FIRST(E) = \{(, id\} \qquad FOLLOW(E) = \{\$,)\}$$

$$FIRST(E') = \{+, \epsilon\} \qquad FOLLOW(E') = \{\$,)\}$$

$$FIRST(T) = \{(, id\} \qquad FOLLOW(T) = \{+, \$,)\}$$

$$FIRST(T') = \{*, \epsilon\} \qquad FOLLOW(T') = \{+, \$,)\}$$

$$FIRST(F) = \{(, id\} \qquad FOLLOW(F) = \{*, +, \$,)\}$$

Now, we can start filling the table. To fill the table, we first take the productions one-by-one as follows –

$$\alpha \to \beta \qquad where\ \beta \neq \epsilon$$

For the production, we take the row α and take the columns of the values returned from $FIRST(\alpha)$ and we fill the production in those columns. In our case, the first production is –

$$E \to TE'$$

Thus, we fill the above production in the row E and column will be $\{(, id\}$. Thus, we get –

	$*$	$+$	$($	$)$	$\boldsymbol{id}$	$
E			$E \to TE'$		$E \to TE'$	
E'						
T						
T'						
F						

Similarly, we can do the same for the rest of the productions and fill the table as follows –

	$*$	$+$	$($	$)$	$\boldsymbol{id}$	$
E			$E \to TE'$		$E \to TE'$	
E'		$E' \to +TE'$				
T			$T \to FT'$		$T \to FT'$	
T'	$T' \to * FT'$					
F			$F \to (E)$		$F \to (E)$	

Now, we still need to consider the case where –

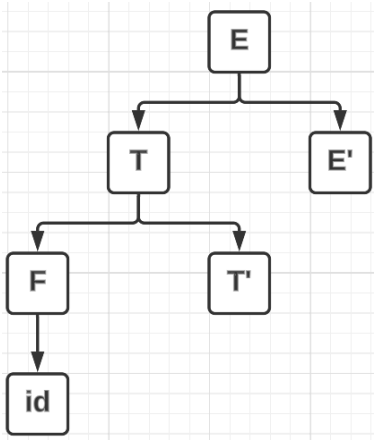
$$\alpha \to \epsilon$$

For the production, we take the row α and take the columns of the values returned from $FOLLOWS(\alpha)$ and we fill the production in those columns. Thus, we get –

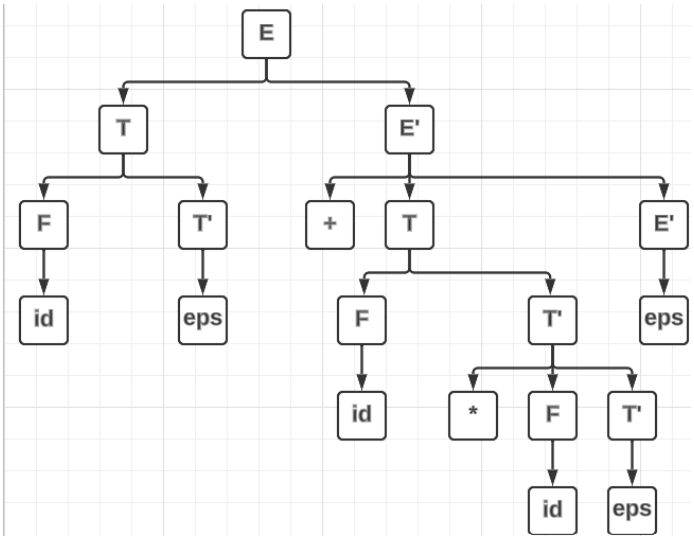
	$*$	$+$	$($	$)$	$\boldsymbol{id}$	$
E			$E \rightarrow TE'$		$E \rightarrow TE'$	
E'		$E' \rightarrow +TE'$		$E' \rightarrow \epsilon$		$E' \rightarrow \epsilon$
T			$T \rightarrow FT'$		$T \rightarrow FT'$	
T'	$T' \rightarrow * FT'$	$T' \rightarrow \epsilon$		$T' \rightarrow \epsilon$		$T' \rightarrow \epsilon$
F			$F \rightarrow (E)$		$F \rightarrow id$	

Now that we have filled this table, we can start creating the parse tree. First, we break the given string into tokens as shown –

Now, we try to use Left – most derivation to get the tokens one-by-one. The first token is $\boldsymbol{id}$**.** So, we can use the productions under the id column to draw the parse tree as follows –



Just like that, we will do the same for the rest of the tokens as well. Thus, we get –



NOTE

There are 3 parts of the name of the parser –

- The first letter represents the direction in which the tape is accessed.
- The second letter represents the derivation direction.
- The number represents the number of tokens it takes at a time.

Since the parser here accesses the tape from left to right, we are using left-most derivation and taking one token at a time. Thus, the name becomes **LL1.**

NOTE

In the LL1 table, each cell can have just one production. If there is a case where the cell will have more than 1 entries, then it is not LL1.

Question

Is the following grammar in LL1?

S → AB
A → bA / ε
B → aB / ε

Answer

We can see that this grammar is non-left recursive, unambiguous and deterministic. Therefore, the initial conditions are satisfied. Next, we can write –

$$FIRST(S) = \{a, b, \epsilon\} \qquad FOLLOW(S) = \{\$\}$$
$$FIRST(A) = \{b, \epsilon\} \qquad FOLLOW(A) = \{a, \$\}$$
$$FIRST(B) = \{a, \epsilon\} \qquad FOLLOW(B) = \{\$\}$$

We can draw the LL1 table as follows –

	a	b	$
S	$S \rightarrow AB$	$S \rightarrow AB$	
A	$A \rightarrow \epsilon$	$A \rightarrow bA$	$A \rightarrow \epsilon$
B	$B \rightarrow aB$		$B \rightarrow \epsilon$

Since every cell has just one production, we can conclude that the grammar is in LL1.

Question

Is the following LL1 grammar?

S → AaAb / BaBb
A → ε
B → ε

Answer

The grammar is not LL1 as the cell (S, a) will have multiple values.

NOTE

- A Grammar without ε is LL(1), if
  - for every production of the A → α_1 / α_2 / α_3 / ----- / α_n, the set First(α_1), First(α_2), First(α_3),....., First(α_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**

Question

Is the following grammar LL1 or not?

$$S \rightarrow aAbB$$

$$A \rightarrow a \mid \epsilon$$

$$B \rightarrow b \mid \epsilon$$

Answer

The grammar is LL1.

Question

Is the following grammar LL1 or not?

S → aA / bB

A → Bb / a

B → bB / c

Answer

The grammar is LL1.

Question

Is the following grammar LL1 or not?

S → (L) / a
L → SL'
L' → ,SL' / ε

Answer

The grammar is LL1.

Question

Is the following grammar LL1 or not?

E → T+E / T
T → id

Answer

The grammar is **NOT** LL1.

BOTTOM – UP PARSER

This is more detailed and has a higher power when compared to Top – down parsers. Most of the high level languages like C, Python, Java etc. use a Bottom – up parser. As the name suggests, the parser creates the parse tree by starting from the children and proceeding towards the root.

To begin with, let us take the grammar as shown –

S → AA
A → aA / b

First step, we need to re-write this grammar as follows –

$$S' \rightarrow .S$$

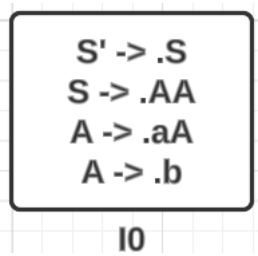
$$S \rightarrow .AA$$

$$A \rightarrow .aA$$

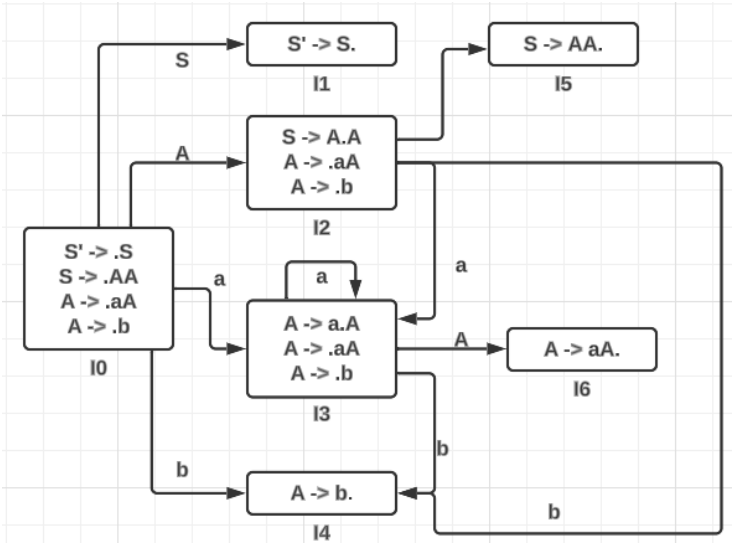
$$A \rightarrow .b$$

Here, the $(.)(\boldsymbol{dot})$ operator represents the position to which we have scanned the production. Since initially we haven't scanned the production yet, the dot operator is in the beginning.

Also, the group of these productions is called a **closure** and is represented as follows –



Now, we will start to scan the productions one symbol at a time and create a sort of tree of closures. This can be done as follows –



Let me now deep wtf just happened. First, we write the original closure I0. From I0, we are parsing based on each symbol – both terminals and non-terminals**.** Here is a step – by – step process of the parsing –

1. First, choose a terminal/non-terminal $\boldsymbol{x}$ based on which you are parsing.
2. Find all the productions of the form $\boldsymbol{\alpha \rightarrow .x\beta}$
3. Write those productions in a new closure.
4. If $\boldsymbol{\beta}$ is a non – terminal, then write all the productions of the form $\boldsymbol{\beta \rightarrow y}$ in the closure as well.

Now that a step-by-step procedure has been established, we can start with the process. First, let us parse using $\boldsymbol{S}$**.** We can see that there is only one production to be considered –

$$S' \rightarrow .S$$

Therefore, we write a new closure I1 with the production $\boldsymbol{S' \rightarrow S}$. As we can see, there is no $\boldsymbol{\beta}$**,** so we can skip step 4. Thus, there is only 1 production in closure I1.

Now, we consider a parse using $\boldsymbol{A}$**.** In this case, we again have just one production –

$$S \rightarrow .AA$$

So, we transform the production to $\boldsymbol{S \rightarrow A.A}$ and write it in production I2. However, we can see that $\boldsymbol{\beta = A}$ which is a non-terminal. So, we need to include all such productions with $\boldsymbol{A}$ on the LHS. Thus, closure I2 has a total of **3 productions.**

Following this process, we can get a total of **7 closures (I0 – I6)**

Once we have the closures, we can then proceed to create the **parse table** as shown below –

	ACTION			**GOTO**	
	a	b	\$	S	A
I0					
I1					
I2					
I3					
I4					
I5					
I6					

Basically, we write ACTION and GOTO as the 2 sections in the table wherein the terminals are listed under ACTION and the non-terminals are listed under GOTO. This is because we terminate at the terminals and we usually proceed to the next step for a non-terminal. Each row represents each of the closures we had created.

To fill this table, let us start with I0. We can see from the closure diagram that upon encountering $\boldsymbol{a}$**,** we **shift from I0 to I3**. This is represented as $\boldsymbol{S3}$**.** Similarly, upon encountering $\boldsymbol{b}$**,** we **shift from I0 to I4**. This is represented as $\boldsymbol{S4}$. Now, we fill the table accordingly.

	ACTION			**GOTO**	
	a	b	\$	S	A
I0	S3	S4			
I1					
I2	S3	S4			
I3	S3	S4			
I4					
I5					
I6					

Just like the terminals, we now check for non-terminals as well. We can see that upon S, I0 goes to I1. This is represented simply by $\mathbf{1}$**.** We don't write S1 for the non-terminals in the GOTO section. Thus, the table now becomes –

	ACTION			**GOTO**	
	a	b	\$	S	A
I0	S3	S4		1	2
I1					
I2	S3	S4			5
I3	S3	S4			6
I4					
I5					
I6					

After this, we divert our attention to the original grammar –

From here, we assign each production as a **rule** as follows –

$$\boldsymbol{R1}: S \rightarrow AA$$

$$\boldsymbol{R2}: A \rightarrow aA$$

$$\boldsymbol{R3}: A \rightarrow b$$

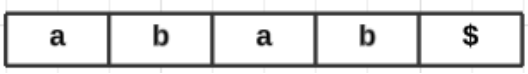
We can see that rules R1, R2 and R3 correspond to closures I5, I6 and I4 respectively. Hence, we will fill those out in the table as follows –

	ACTION			**GOTO**	
	a	b	$\$$	S	A
I0	S3	S4		1	2
I1					
I2	S3	S4			5
I3	S3	S4			6
I4	R3	R3	R3		
I5	R1	R1	R1		
I6	R2	R2	R2		

Finally, we can see that the start symbol ($\boldsymbol{S'}$) is present in closure I1. So, we will fill the row I1 and column $ with the word **ACCEPT.** Basically, if after parsing the string we end up on ACCEPT, then the string is accepted by the parser.

	ACTION			**GOTO**	
	a	b	$\$$	S	A
I0	S3	S4		1	2
I1			**accept**		
I2	S3	S4			5
I3	S3	S4			6
I4	R3	R3	R3		
I5	R1	R1	R1		
I6	R2	R2	R2		

Phew! The table is now complete and now we can proceed with the next step which is to use a string for testing. Let us assume a string as $\boldsymbol{abab}$**.** From intuition, we know that this string is part of the CFG and hence should be accepted by the parser. As usual, we first get the tokens of the string –



To perform the analysis, the parser actually uses a **stack.** Let us assume such a stack with the initial value as **0**

0														

Here, the number 0 denotes the initial closure I0. Similarly, a number 3 will represent closure I3 and so on. Now, we traverse the input string.

The first element is $\boldsymbol{a}$**.** As per the stack, we are currently in closure I0. Thus, we can see that at closure I0, we have encountered $\boldsymbol{a}$**.** Thus, as per the table we need to shift to I3 $(\boldsymbol{S3})$. Thus, the stack now becomes –

0	a	3												

Now, the next symbol coming is $\boldsymbol{b}$**.** Thus, as per the stack we are in closure I3 and now we have encountered a $\boldsymbol{b}$**.** Looking at the table, we need to perform $\boldsymbol{S4}$**.** Hence, the stack becomes –

0	a	3	b	4										

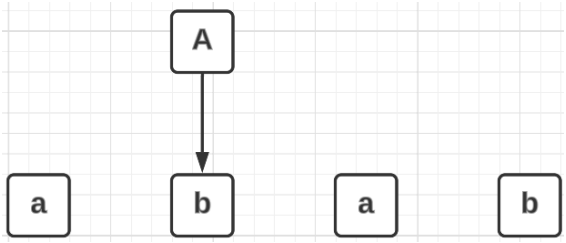
Next, we have the symbol $\boldsymbol{b}$**.** As per the table, when we encounter $\boldsymbol{b}$ in closure I4, the output is Rule 3 $(\boldsymbol{R3})$**.** This is an interesting case. Here, we are not shifting but rather this is where we perform **reduction –** which is where we go up a level in the tree. Here are the rules of the reduction –

- Move up the graph as per the rule.
- If the rule is like $\boldsymbol{\alpha \rightarrow \beta}$**,** then pop $\boldsymbol{2 * len(\beta)}$ elements from the stack
- Push the reduced non-terminal to the stack.

In our case, we have encountered Rule 3 which is –

$$A \rightarrow b$$

Therefore, the $\boldsymbol{b}$ symbol will get reduced to $\boldsymbol{A}$ non-terminal. At the same time, we know that $\boldsymbol{len(b) = 1}$**.** Thus, we pop **2 elements** from the stack and push $\boldsymbol{A}$ into it. So, the tree will now become something as shown below –



At the same time, the stack has now become –

0	a	3	A											

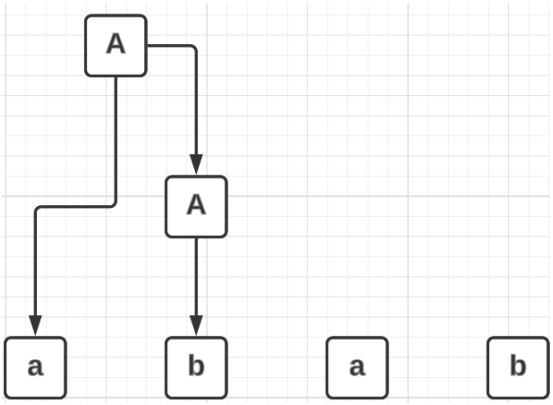
Now, we are at closure I3 and have encountered symbol $\boldsymbol{A}$**.** As per the table, we **GOTO** closure I6.

0	a	3	A	6										

The next symbol is $\boldsymbol{a}$ and we are at closure I6. As per the table, we need to perform reduction via **Rule 2.** As per R2, we have –

$$A \rightarrow aA$$

That means both $\boldsymbol{a}$ and $\boldsymbol{A}$ get reduced to $\boldsymbol{A}$**.** We move up a level in the graph. Also, $\boldsymbol{len(aA) = 2}$**.** Thus, we pop **4 elements** from the stack. The graph and the stack are now modified as follows –



0	A													

At this point, we are at closure I0 and the symbol encountered is $\boldsymbol{A}$**.** Thus, we **GOTO** closure I2.

0	A	2												

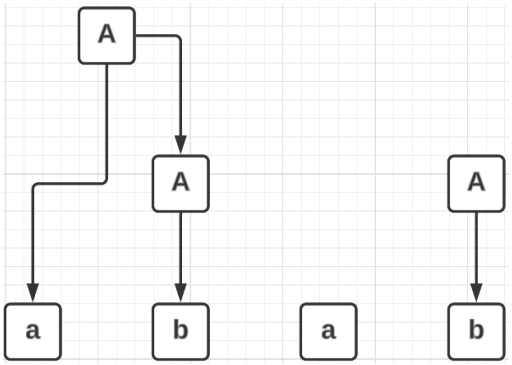
The next symbol is $\boldsymbol{a}$ and at closure I2, we encounter a $\boldsymbol{a}$ and then move on to closure I3.

0	A	2	a	3										

The next symbol to appear is $\boldsymbol{b}$ which upon closure I3 will shift to closure I4.

0	A	2	a	3	b	4								

At this point, all the symbols of the string have been pushed onto the stack at some point or the other. So all that is left now is reduction. Now, the last symbol to be encountered is \$**.** As per the table, the closure I4 on \$ will give reduction via Rule 3. Thus, the tree and stack become –

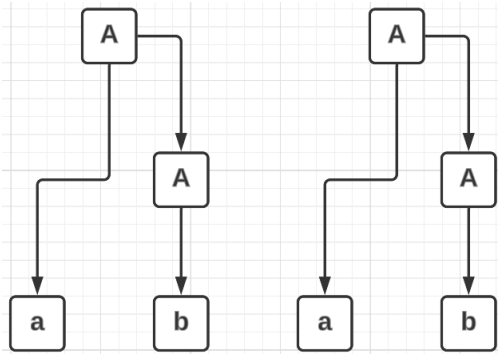


0	A	2	a	3	A									

On closure I3, we encountered $\boldsymbol{A}$ and thus we **GOTO** to closure I6.

0	A	2	a	3	A	6								

Now, in closure 6 if we encounter \$**,** then we need to reduce via Rule 2. Thus, we get –

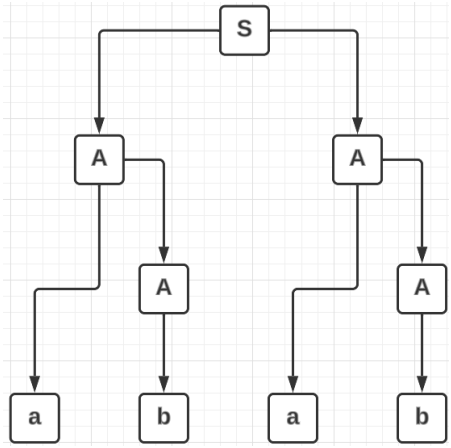


0	A	2	A											

In closure 2, if we encounter $\boldsymbol{A}$**,** we **GOTO** closure I5.

0	A	2	A	5										

Now, if we encounter a $ in closure I5, we need to reduce via Rule 1. Hence, we get –



0	S													

On closure I0, when we encounter $\boldsymbol{S}$**,** we need to **GOTO** closure I1. Now, if we encounter $ at closure I1, we can see that we get **ACCEPT.**

OOOFF!!! We are done finally. We can see that we started with the entire string and built the tree in a bottom – up fashion. At the same time, since we reached **ACCEPT,** it means that the string is a valid string for the given CFG.

BOTTOM – UP PARSER THEORY

We just performed the process of developing a BUP, so we should define a few terms as well.

- **Handle –** Any subset of the RHS of a production is called a **handle.**
- **Pruning –** The process of searching and replacing a handle with its LHS part is called **pruning.**

In the example we had taken previously, there was a production –

$$S \rightarrow AA$$

Thus, $\boldsymbol{AA}$ is a handle while the process in the last step where we replaced AA with $\boldsymbol{S}$ is called pruning.

Now, we can see that we performed basically 2 main operations in the entire process – **Shifting** to a new closure and **Reducing** based on a rule. Therefore, BUPs are also called **Shift-Reduce Parsers.**

The production we have just created parses the string from **left-to-right** and at the same time, it also performs **reverse rightmost derivation.** It also doesn't have any lookahead symbols. Therefore, this parser is called a **LR(0) parser.**

Since we are doing Reverse Rightmost Derivation, we can use this parser on CFG with Left-recursion as well. Additionally, suppose we have a production as follows –

$$S \rightarrow aA \mid aB$$

The CFG with the above production is non-deterministic since when we encounter $\boldsymbol{a}$**,** we don't know if the next symbol will be either $\boldsymbol{A}$ or $\boldsymbol{B}$**.** However, this is the case when we use Leftmost Derivation. In case we are using Rightmost derivation, then we don't have any problems. In conclusion, **LR(0) can work on left-recursive and non-deterministic CFG.** Therefore, a BUP has **higher expressive power** when compared to TDP.

Question

Check if the given CFG is LR(0) or not

E → T + E / T

T → id

Answer

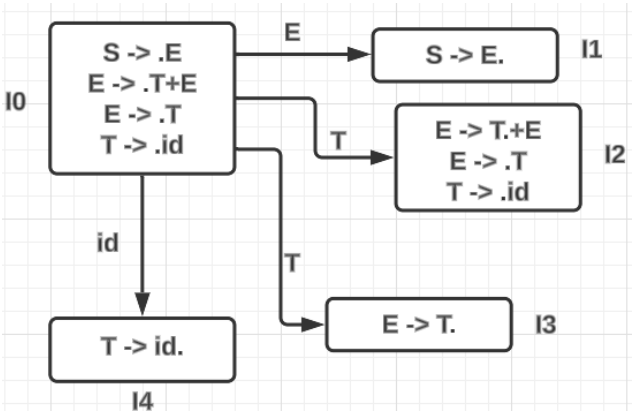
For this case, the canonical closures will be as follows –

$$S \rightarrow .E$$

$$E \rightarrow .T + E$$

$$E \rightarrow .T$$

$$T \rightarrow .id$$



At this point itself, we can see that if we apply T to I0, we can either get a shift to I2 or a reduction. This will result in the parse table having multiple values in a cell and thus, the CFG is **NOT LR(0).**

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 →б. (reduced)

- R-R Conflict: If the same state contains more than one final item, then it is R-R Conflict
  A → α. (x is terminal, shift)
  B →α. (reduced)

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

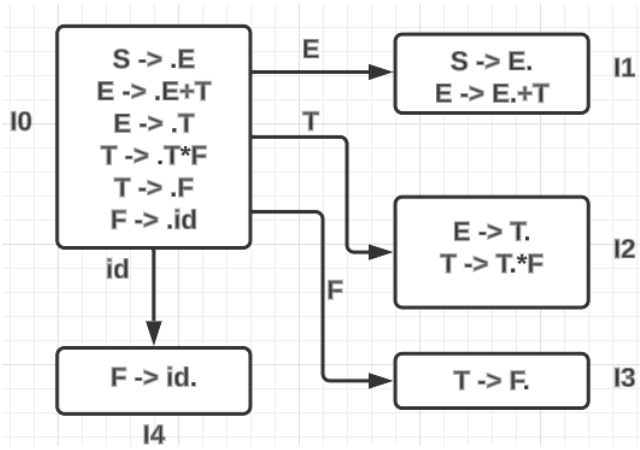
Question

Check if the given CFG is LR(0) or not

E → E + T / T
T → T * F / F
F → id

Answer

For this case, the canonical closures will be as follows –

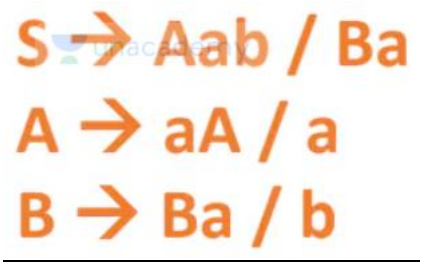


In this case, I1 has a reduction and a shifting. However, the reduction $S \rightarrow E$. Is just the **accept** condition so it will not have any clash. On the other hand, in I2 there is a reduction and a shifting.

Therefore, there is a **SR clash** and hence the grammar is not LR(0).

Question

Check if the given grammar is LR(0) or not.



Answer

This is not a LR(0) grammar.

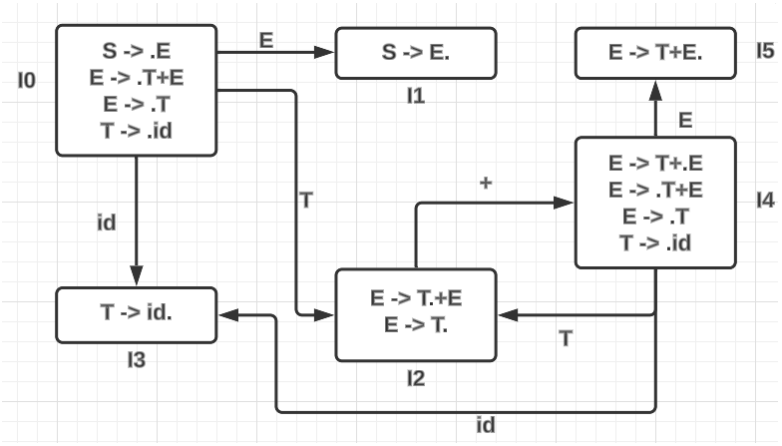
SLR(1) PARSER

Let us take the case of the grammar shown below –

$$E \rightarrow T + E \mid T$$

$$T \rightarrow id$$

Now, for this grammar we can proceed to form the canonical closure –



From this, we can draw the parse table as follows –

	ACTION			GOTO	
	$+$	id	$\$$	E	T
I0		S3		1	2
I1			ACCEPT		
I2	S4/R2	R2	R2		
I3	R3	R3	R3		
I4		S3		5	2
I5	R1	R1	R1		

We can see that the cell in row I2 and column $+$ has 2 values. This is a **SR conflict.** Thus, the given grammar is not in LR(0) form. However, we are now going to learn about a new parser – **SLR(1) parser.**

The process for SLR(1) is the same as LR(0) however as the name suggests, it has a single lookahead symbol. This means that ***"the reduction rules are applied to the Follow of the LHS".***

In our case, we have –

$$FOLLOW(E) = \{\$\}$$

$$FOLLOW(T) = \{\$, +\}$$

We can see that Rules R1 and R2 have $\boldsymbol{E}$ as the LHS while Rule R3 has $\boldsymbol{T}$ as the LHS. Thus, we can say that –

- R1 and R2 will only be filled in the $ column
- R3 will be filled only in $\{\$, +\}$ columns

Thus, the parse table now becomes –

	ACTION			**GOTO**	
	$+$	id	$\$$	E	T
I0		S3		1	2
I1			ACCEPT		
I2	S4		R2		
I3	R3		R3		
I4		S3		5	2
I5			R1		

Now, there are no conflicts and thus, this CFG is accepted under **SLR(1) parser.** Thus, we can say that the expressive power of SLR(1) is **greater than** the expressive power of LR(0).

Question

Check if the given grammar is in LR(0) and SLR(1)

S → Aa / bAc / dc / bda

A → d

Answer

$$S' \rightarrow .S$$

$$S \rightarrow .Aa$$

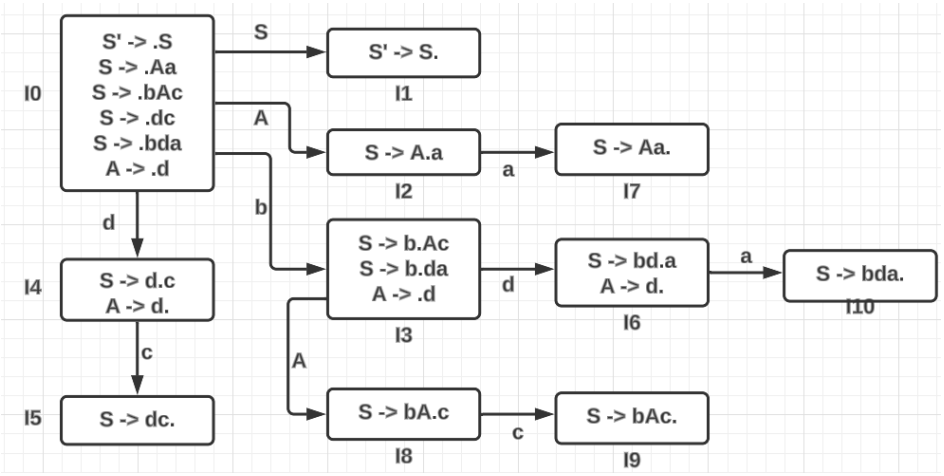
$$S \rightarrow .bAc$$

$$S \rightarrow .dc$$

$$S \rightarrow .bda$$

$$A \rightarrow .d$$

We can draw the closures as follows –



For LR(0), we can write the parse table as follows –

	ACTION					GOTO	
	a	b	c	d	\$	S	A
I0		S3		S4		1	2
I1					ACCEPT		
I2	S7						
I3				S6			8
I4	R5	R5	**S5/R5**	R5	R5		
I5	R3	R3	R3	R3	R3		
I6	**S10/R5**	R5	R5	R5	R5		
I7	R1	R1	R1	R1	R1		
I8			S9				
I9	R2	R2	R2	R2	R2		
I10	R4	R4	R4	R4	R4		

We can see that this grammar is **not in LR(0**). Now, we need to check for SLR(1). To do so, first we need to get the Follow functions as follows –

$$FOLLOW(S) = \{\$\}$$

$$FOLLOW(A) = \{a, c\}$$

Hence, the SLR(1) parse table becomes –

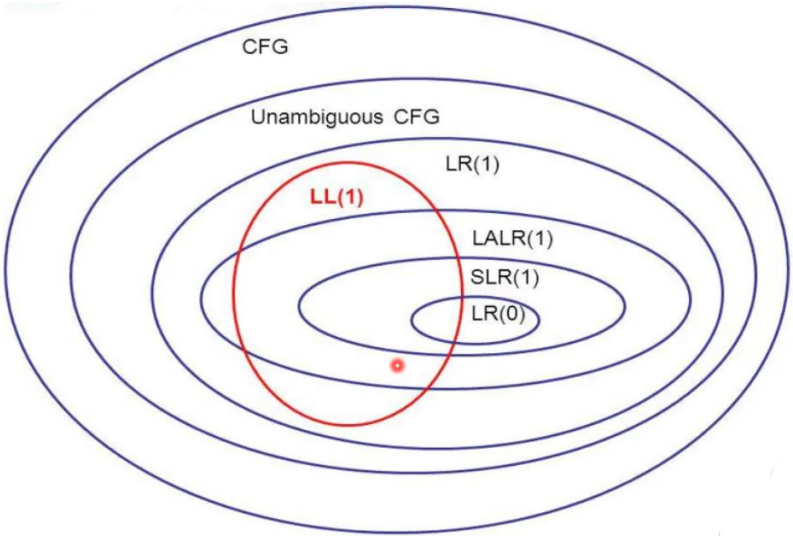
	ACTION					GOTO	
	a	b	c	d	\$	S	A
I0		S3		S4		1	2
I1					ACCEPT		
I2	S7						
I3				S6			8
I4	R5		**S5/R5**				
I5					R3		
I6	**S10/R5**		R5				
I7					R1		
I8			S9				

I9					R2		
I10					R4		

We can see that there are still SR conflicts. Hence, this grammar is **NOT in SLR(1) as well.**

NOTE

In general, the Venn diagram of the different types of parsers is given as follows –



We can see that LL(1) has a lot of power, but every LL(1) grammar is LR(1). Therefore, it just proves that the BUP have a higher power when compared to TDP.

Question

Check if the given grammar is in LR(0) and SLR(1)

S → AaB

A → ab / a

B → b

Answer

The CFG are in SRL(1) but not in LR(0)

Question

Check if the given grammar is in LR(0) and SLR(1)

S → AS / b

A → SA / a

Answer

The CFG is neither in LR(0) and SLR(1).

Question

Check if the given grammar is in LR(0) and SLR(1)

Answer

The CFG is neither in LR(0) and SLR(1).

CLR(1) PARSER

This is the **most powerful parser.** In short, if a grammar is not in CLR(1), then it can't be done by any of the parsers. To understand the CLR(1) parser, let us take an example of a CFG as follows –

$$S' \rightarrow .S$$

$$S \rightarrow .CC$$

$$C \rightarrow .cC$$

$$C \rightarrow .d$$

If we were making SLR(1), then we would have found the follow for S and C. However, we can see that $\boldsymbol{C}$ occurs a total of **3 times** in the RHS of the productions. **Each of these occurrences have a different follow.** Let us **label** these occurrences as follows –

$$S' \rightarrow .S$$

$$S \rightarrow .C_1C_2$$

$$C_1 \rightarrow .cC_3$$

$$C_1 \rightarrow .d$$

We can see that even though $C_1 = C_2 = C_3 = C$, their follows will be different –

$$FOLLOW(C_1) = \{c, d\}$$

$$FOLLOW(C_2) = \{\$\}$$

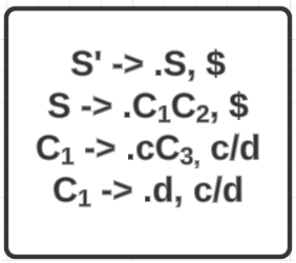
$$FOLLOW(C_3) = \{c, d\}$$

Here is the interesting part. If we were constructing SLR(1), we would be writing the **Reduction rules** in the columns $\boldsymbol{c}, \boldsymbol{d}, \$$ since SLR(1) would have calculated $FOLLOW(C) = \{c, d, \$\}$. On the other hand, for CLR(1), we can see that each occurrence has a different follows and hence, when we fill the table, **CLR(1) will have lesser SR/RR conflicts when compared to SLR(1).** This is the reason CLR(1) is the most accommodating, has the highest expressive powers and also has minimal elements in the parse table.

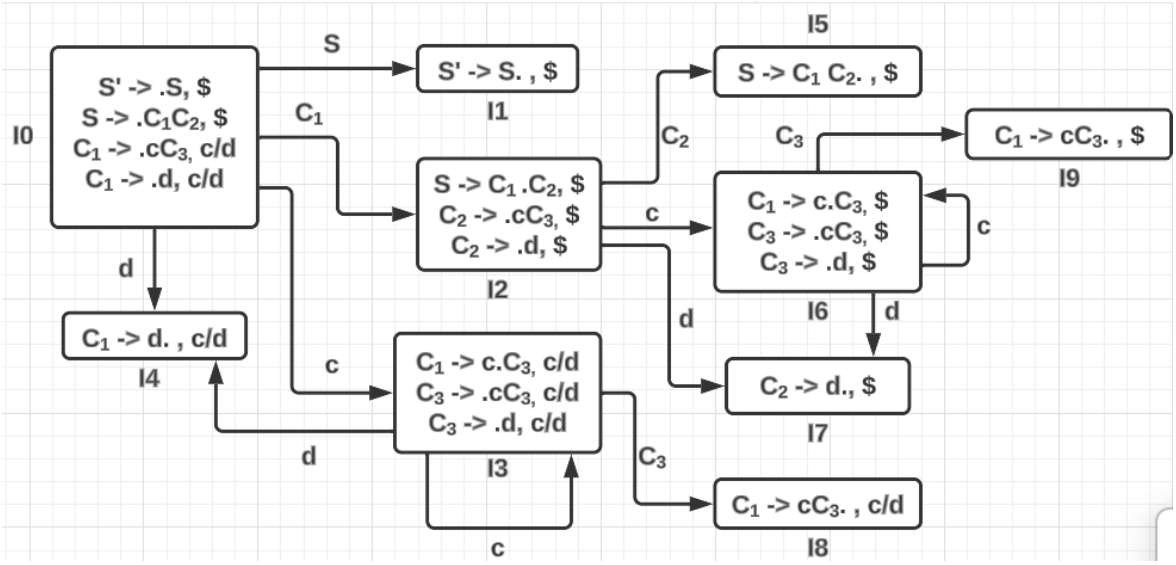
To create CLR(1), we need to **include each productions follow in the closure**. For example, if closure has the production $\boldsymbol{\alpha \rightarrow \beta}$**,** then we write –

$$\boldsymbol{\alpha \rightarrow \beta}\,, \boldsymbol{FOLLOW(\alpha)}$$

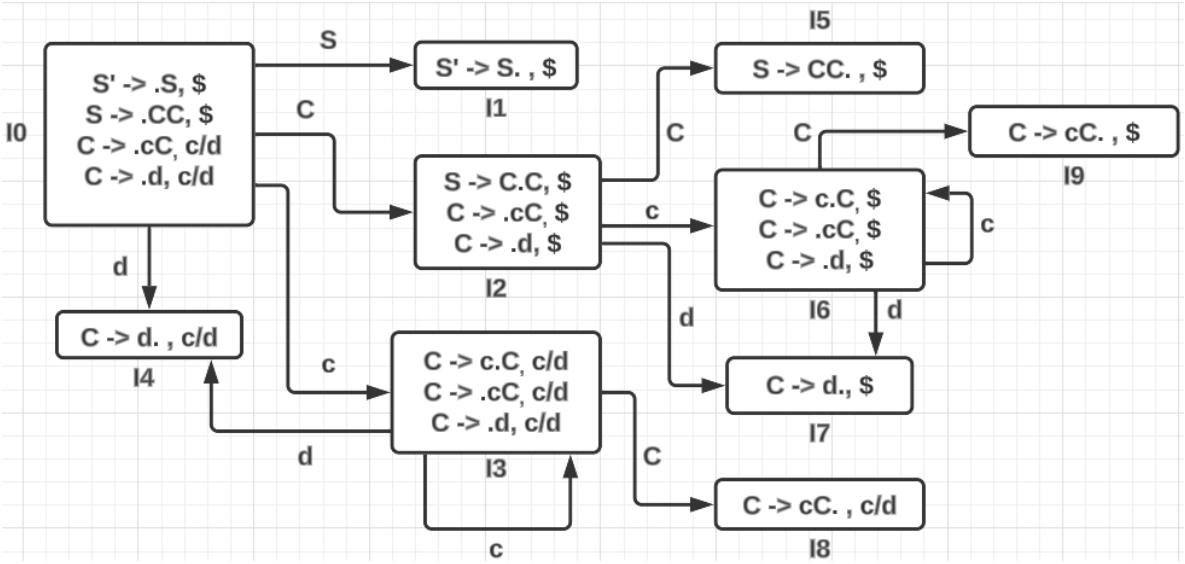
Thus, for the example grammar, the initial closure will be as follows –



One thing to note here is that $C_1, C_2\ and\ C_3$ are basically the same as $\boldsymbol{C}$**.** Since this is the first example, I am differentiating them so that it is easy to understand which occurrence of C we are talking about. After this, we continue drawing the canonical closures as we have –



Since we have $C_1 = C_2 = C_3 = C$, we can modify the canonical closure as follows –



Using the above closure, we can write the parse table as follows –

	ACTION			GOTO	
	c	d	\$	S	C
I0	S3	S4		1	2
I1			ACCEPT		
I2	S6	S7			5
I3	S3	S4			8
I4	R3	R3			
I5			R1		
I6	S6	S7			9
I7			R3		
I8	R2	R2			
I9			R2		

As we can see, there are no SR/RR conflicts thus the **CFG is CLR(1) parser**.

We can see that there are a couple of closures that have the same productions but have different follows. These closures are –

$$I3 - I6$$

$$I4 - I7$$

$$I8 - I9$$

Thus, we can combine these closures into each other. That will give us the following parse table –

	ACTION			GOTO	
	c	d	\$	S	C
I0	S3	S47		1	2
I1			ACCEPT		
I2	S6	S7			5
I36	S36	S47			89
I47	R3	R3	R3		
I5			R1		
I89	R2	R2	R2		

As we can see, we are able to perform the combination without any conflicts. Thus, we can say that this grammar is **also LALR(1).**

NOTE

If a grammar is in CLR(1) and there is no possibility of combination or if there is a possibility of combination and there are no conflicts, then the CFG is also in LALR(1). If there is a possibility of combination but it results in a conflict, then the CFG is not in LALR(1).

Question

Check if the given grammar is CLR(1) and LALR(1) or not.

Answer

This is CLR(1) but not LALR(1).

OPERATOR PRECEDENCE GRAMMAR

These are parsers that are used to manipulate and work on mathematical expressions. It is a less complex grammar that can be made on both **ambiguous and unambiguous grammars.** Every CFG is not OPG. OPG are CFG which have the following properties –

- They don't contain NULL productions
- There are no adjacent non-terminals on the RHS of the productions.

For example –

S → AaB
B → aA / b
A → b
B → a

SEMANTIC ANALYSIS

This is the next step in the compiler process. To perform a Semantic Analysis, we need to first define something called **Syntax Directed Translation (SDT).** We can say that –

$$SDT = Grammar + Semantic\ Rules + Semantic\ Actions$$

Semantic analyzer is quite tough to learn in – depth, so in GATE they usually ask very specific type of questions. We will have a look at these questions along the way 😊. Also note that SDT can be used for functions apart from the semantic analysis.

Question

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 $id_{.val}$ 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 $S_{.val}$ for root of the parse tree for the expression 20 #10%5 #8%2%2 is _____. **(GATE 2022) (2 MARKS)**

Answer

First off, let us notice that against each of the productions, we have some expressions inside curly braces. These are the **semantic statements.** These statements are usually of 2 types –

- **Semantic Rules –** If there is a $\boldsymbol{if}$ statement in the Semantic statement, then it is called a semantic rule.
- **Semantic Action –** If there is no $\boldsymbol{if}$ statement, then it is called a semantic action

In the question, we don't have any semantic rules, so there is no condition check here. Now, let us learn how to interpret the semantic action. For example, let us take the production –

$$S \rightarrow S_1 \# T \qquad \{S.val = S.val * T.val\}$$

As per the semantic action, we can say that the production will multiply S and T and store the result back in S. One small thing to note is that as per the question, sub-scripted non-terminals are the same non-terminals in a time instant. So basically $S = S_1$.

Thus, we can write –

$$x \# y \quad \rightarrow \quad x = x * y$$

Similarly, we can see that the $\%$ (mod) sign represents division. Thus, the expression becomes –

$$20 * 10 \div 5 * 8 \div 2 \div 2$$

Here comes the important part. Now that we have the operator definitions, we need the **OPERATOR PRECEDENCE AND ASSOCIATIVITY.**

To get the precedence, we use common sense. When we make a tree, we go from root node to the leaf nodes. However, when we calculate values, we go from leaf nodes to the root nodes. Hence, **the operator that comes first when we go from bottom to top has higher precedence.** In our case,

$$\Pr(\%) > \Pr(\#)$$

Once we have the precedence, we need associativity. To get associativity, check the production recursion which has the operator. In our case, we have –

$$S \rightarrow S_1 \# T$$

$$T \rightarrow T_1 \% R$$

Since these both productions are left recursive, we can say that **both operators have a left-to-right associativity.**

Now, we can solve the expression –

$$20 * (10 \div 5) * \big((8 \div 2) \div 2\big)$$

The final value will be **80.**

Question

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 **(C)** 160

(B) 180 **(D)** 40

Answer

We can see from the grammar that –

- # is multiplication and is left-to-right associative
- & is addition and is left-to-right associative

We can also say that since & comes first when going from bottom-to-top, & has higher priority when compared to #.

$$2 * (3 + 5) * (6 + 4)$$

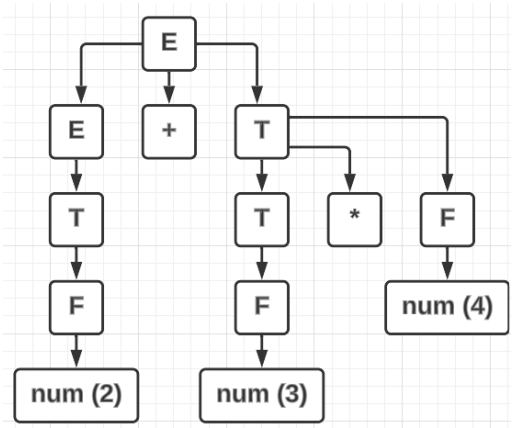
The final answer will be **160.**

Question

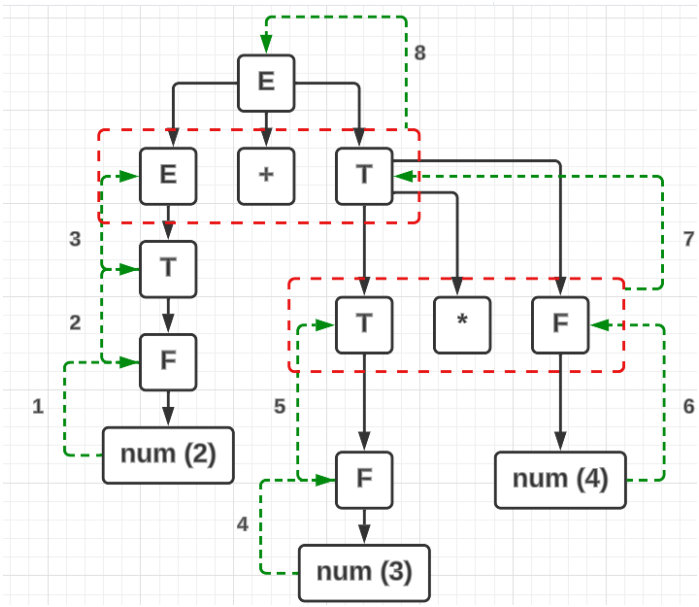
Q Consider the grammar with the following translation rules and E as the start symbol.
$E \rightarrow E_1$ + T {print ('+');}
E → T
$T \rightarrow T_1$ * F {print ('*');}
T → F
F → num {print ('num.val');}
Construct the parse tree for the string 2 + 3 * 4, and find what will be printed.

Answer

In this case, the best thing will be to make the parse tree as follows –



Now, we go from bottom to top to get the value of the expression. Shown below is the path we take for valuation –



In Paths 1, 4, 6, 7 and 8 will have printing as per the semantic actions mentioned in the grammar. Thus, we get –

$$\boldsymbol{Final\ Expression} = \mathbf{234} * +$$

If we note carefully, this grammar is used to perform **infix to postfix conversions.**

Question

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

$E \rightarrow E_1 * T \{ E.value = E_1.value * T.value \}$
$E \rightarrow T\{ E.value = T.value \}$
$T \rightarrow F - T \{ T.value = F.value - T_1.value \}$
$T \rightarrow F\{ T.value = F.value \}$
$F \rightarrow num \{ F.value = num.value \}$

Compute E.value for the root of the parse tree for the expression: 4 – 2 – 4 * 2.

Answer

Here,

- – means subtraction and ∗ means multiplication.
- – is right associative while ∗ is left associative.
- – has higher precedence when compared to ∗

With this information, the expression becomes –

$$E = \big(4 - (2 - 4)\big) * 2 = \mathbf{12}$$

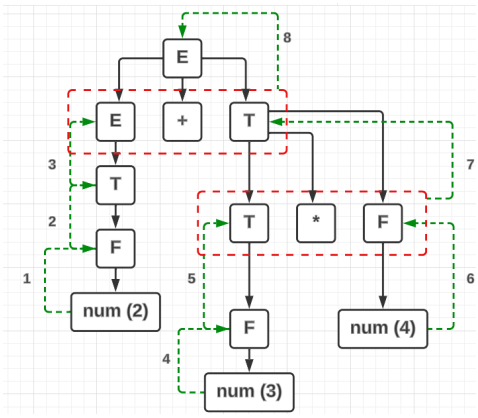
Question

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

Answer

If we notice, the semantic actions mentioned here are not for evaluation but rather for **making a node.** So, this grammar is used to **create a tree.** To start, first let us draw a rough tree and parse sequence as follows –

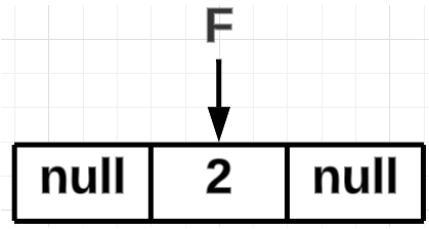


Let us now go step-by-step.

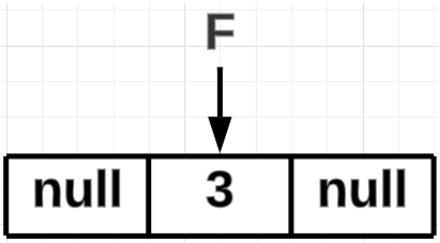
Step 1 – We go from $num(2)$ to F. As per the grammar, the action here is as follows –

{ F.nptr = mknode(null, id name, null);}

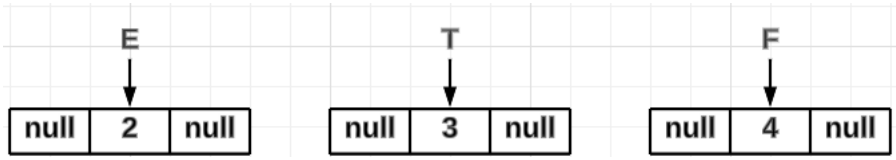
This means that we need to make a node with **left and right pointers as NULL, the value as $id(2)$ and its parent pointer will be F.**



In steps 2 and 3, the parent pointer first changes to T and then to E. In step 4, just like Step 1 another node is created –



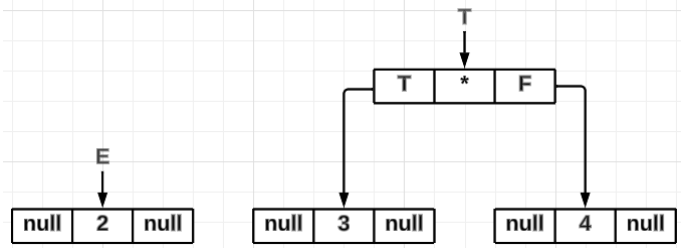
Then in Step 5 the parent pointer for this node is changed to T and then in Step 6, another node for value 4 is created. So now, we have 3 nodes is total –



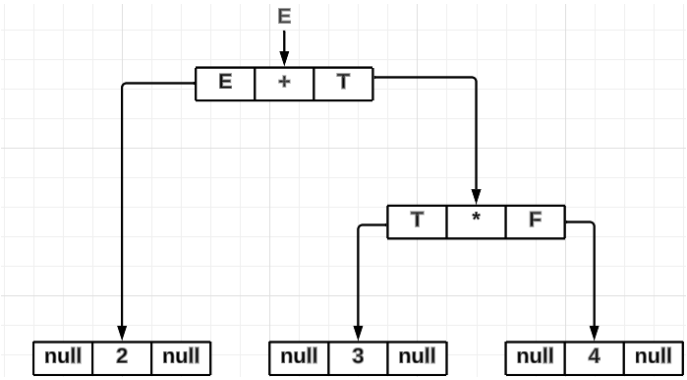
In step 6, we are going to merge using the production $T \rightarrow T * F$. The semantic action in this case will be –

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

Hence, we need to create a node with the left pointer as the T pointer, the right pointer as the F pointer and the value as $*$. The parent pointer will be the T pointer.



Similarly, we do the final step and get the final tree as follows –



CLASSIFICATION OF ATTRIBUTES

Based on the process of evaluation, attributes are classified into two types –

- **Synthesized –** The attributes whose values are calculated based on their children values.
- **Inherited –** The attributes whose values are calculated based on their parent or **left sibling.**

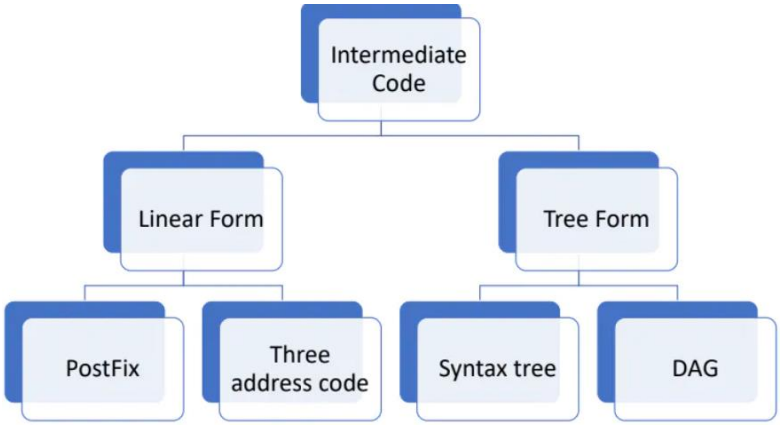
For example, let us take the following production –

$$A \rightarrow XYZ$$

If the value of $\boldsymbol{A}$ is calculated by using the values of X, Y or Z, then X is a **synthesized** attribute. If the value of $\boldsymbol{Y}$ is calculated by using the values of A or X, then it will be termed as an **inherited** attribute.

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

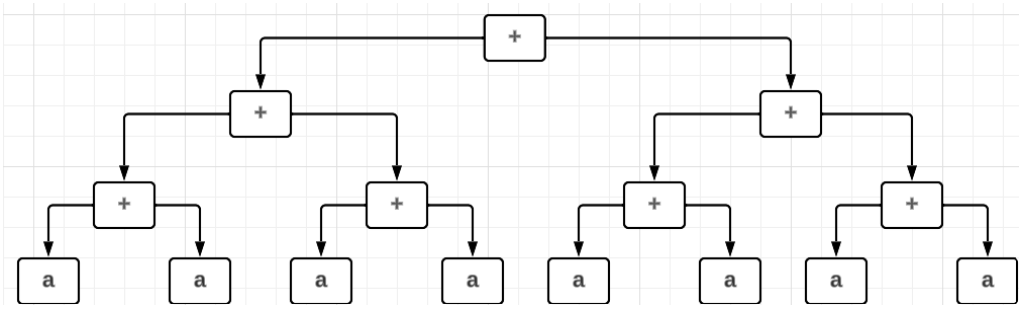
INTERMEDIATE CODE GENERATION



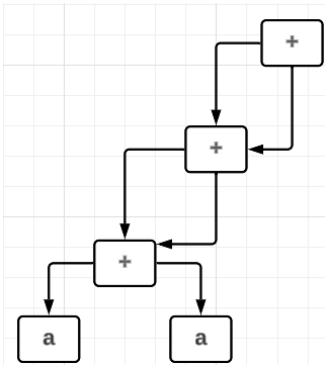
Let us take one expression as follows –

$$\big((a+a)+(a+a)\big)+\big((a+a)+(a+a)\big)$$

We can convert this to postfix using a stack as we had seen in DS. Using **syntax tree,** we can represent this expression as follows –



As we can see, the syntax tree has a lot of repetition. In short, we don't need to create new nodes for each of the $a+a$ operations. Hence, we reduce this to get the **Directed Acyclic Graph (DAG).**



Now, let us take another expression as follows –

$$F=-(a+b)*(c+d)+(a+b+c)$$

Here, we can also express using the **Three Adress Code.** In a three address code, each expression can have only 3 address aka variables. Hence, for this case the TAC will be –

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) $t_7 = t_4 + t_6$

The TAC is also represented in the form of **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

Here, we utilize a lot of space but at the same time we have the flexibility to move the instructions up and down. To save space, we can represent using triplets –

	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

Here, the statement numbers are used as operands. Here, even though the space used is less, there is no flexibility to change the statement order.

Static Single Assignment Form

This is a property of the intermediate code where each variable is assigned **only once.** So, existing variables are split into versions. For example, let us say we have the following code –

$$a = b + c$$

$$a = 2 * a$$

For SSA form, we want the assignment to a variable be done just once. Therefore, we split a into two versions as follows –

$$a = b + c$$

$$a_1 = 2 * a$$

Now this is in SSA.

Constant Folding

In an expression, there is a chance that the constants in the expression can be simplified to reduce run time. This is called constant folding. Suppose we have the expression –

$$a = b + c + 2 + 3 * 4$$

Using constant folding, we can simplify the expression as follows –

$$a = b + c + 14$$

Constant Propagation

Suppose we have expressions as follows –

$$pi = 3.1415$$

$$c = 2 * pi * r$$

We know that the value of **PI** will not change as it is a constant. So, instead of declaring it as a variable, we can directly plug it in the expression. This is called constant propagation. Combining with constant folding, we can improve the expression as follows –

$$c = 6.283 * r$$

Strength Reduction

In this case, we can implement the same expression in more than 1 way. Thus, it would make sense to have the compiler use the expression that has the lowest cost. This is called strength reducing. For example, let us take the 2 expressions –

$$y = 2 * x$$

$$y = x + x$$

In both cases, y will have the same value. However, we know that since multiplication is repetitive addition, it will be more costly. Thus, the compiler would just compile the second expression to save time and resources.

Redundant Code Elimination

As we have seen that there is more than one way to execute an expression, hence causing a case where there can be redundant code lines. Thus, it is better to remove those expressions. This is called Redundant Code Elimination. For example, let us look at the expression below –

$$x = a + b$$

$$y = b + a$$

Instead of performing addition twice, we can simply write –

$$x = a + b$$

$$y = x$$

Algebraic Simplification

Use the basic laws of math to simplify the expressions. For example,

$$x = a * 1$$

$$y = b + 0$$

We can save out on both the operations and simply write –

$$x = a$$

$$y = b$$

Loop Optimization

As the name suggests, we need to optimize the loops in the program. To do so, we first need to perform the following steps –

- First, we convert the High level code to 3-address code
- Then, we get the **leader statements** in the 3-address code
  - The first statement is always the leader
  - The statement right after jump statement is a leader
  - The target statements in the jump statements are also leaders
- Now, we divide the 3-address code lines into **blocks** where the statements between 2 leaders is called a block.
- Finally, we draw the Control Flow Graph to see the relation between the blocks

For example, let us take the code –

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

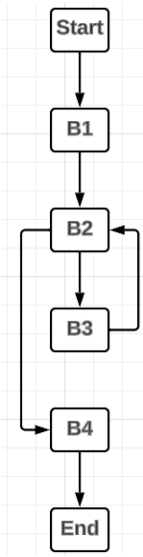
First, we convert the HLL to 3-address codes –

```
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
```

In this case, Lines **1,3,4,9** will be the **leaders.** Hence, we can make the blocks as follows –

```
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
```

Now, we can draw the CFG as follows –



Loop Jamming

This is the process where we combine 2 loops into one loop if they share the same index and no of variables. For example, let us take the case –

```
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"
```

We can see that the second loop and the outer first loop have the same indexing. Thus, we can perform loop jamming here as shown below –

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

Loop Unrolling

This is the case where we reduce the number of iterations in a loop. For example, let us consider the code as follows –

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

In this case, we need to perform 100 iterations. Instead, we can perform Loop unrolling and get the following code –

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

In this case, we will be performing just 50 iterations and getting the same output.

Code Movement

Basically, move all the code that is not dependent on the loop outside the loop.

Live and Dead variables

The basic idea is that there can be a case where a variable will finish its execution before the end of the program and thus, it can relinquish control of the memory location. When a variable is being used, it is said to be live. When a variable has finished its role in the program, it is termed as dead. For example,

1. $p = q + r$
2. $s = p + q$
3. $u = s * v$
4. $v = r + u$
5. $q = v + r$

For these 5 statements, we can use a table to get the live and dead variables as follows –

	p	q	r	u	v	s
1.	Dead	Live	Live	Dead	Live	Dead
2.	Live	Live	Live	Dead	Live	Dead
3.	Dead	Dead	Live	Dead	Live	Live
4.	Dead	Dead	Live	Live	Live	Dead
5.	Dead	Dead	Live	Dead	Live	Dead

QUESTION BANK

Question 1

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)

Question 2

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

Question 3

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

Question 4

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

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

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

Question 5

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

Question 6

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

Question 7

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

x = x * (a + b) – 5;

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

Question 8

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

---

Question 9

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

Question 10

Q Consider the following two Grammars: **(NET-JULY-2018)**

G_1	G_2
S → SbS / a	S → aB / ab A → AB / a B → Abb / b

Which of the following option is correct?

(a) Only G_1 is ambiguous

(b) Only G_2 is ambiguous

(c) Both G_1 and G_2 are ambiguous

(d) Both G_1 and G_2 are not ambiguous

Question 11

Q Given the following two grammars: **(NET-JUNE-2014)**

G_1	G_2
S → AB / aaB A → a / Aa B → b	S → a S b S / b S a S / λ

Which statement is correct?

(A) G_1 is unambiguous and G_2 is unambiguous.

(B) G_1 is unambiguous and G_2 is ambiguous.

(C) G_1 is ambiguous and G_2 is unambiguous.

(D) G_1 is ambiguous and G_2 is ambiguous.

Question 12

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

Question 13

Q Which of the following statements is/are TRUE?
(i) The grammar **S → SS / a** is ambiguous (where S is the start symbol).
(ii) The grammar **S → 0S1 / 01S / ∈** is ambiguous (the special symbol e represents the empty string and S is the start symbol).
(iii) The grammar (where S is the start symbol).

S → T/U
T → x S y / xy / e
U → yT

generates a language consisting of the string yxxyy. **(NET-NOV-2017)**

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

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

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

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

Question 14

Q Given the following grammars:

G_1	G_2
S → AB / aaB A → aA / ∈ B → bB / ∈	S → A / B A → aAb / ab B → abB / ∈

Which of the following is correct? **(NET-JUNE-2015)**
a) G_1 is ambiguous and G_2 is unambiguous grammars

b) G_1 is unambiguous and G_2 is ambiguous grammars

c) both G_1 and G_2 are ambiguous grammars

d) both G_1 and G_2 are unambiguous grammars

Question 15

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.

Question 16

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

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

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)

Question 17

Q The grammar 'G_1' and the grammar 'G_2'. Which is the correct statement? **(NET-DEC-2012)**

G_1	G_2
S→ 0S0 / 1S1 / 0 / 1	S→ aS / aSb / X X → Xa / a

(A) G_1 is ambiguous, G_2 is unambiguous

(B) G_1 is unambiguous, G_2 is ambiguous

(C) Both G_1 and G_2 are ambiguous

(D) Both G_1 and G_2 are unambiguous

Question 18

Q The grammar **(NET-JUNE-2012)**

$S \rightarrow aB / bA$

$A \rightarrow a / aS / bAA$

$B \rightarrow b / bS / aBB$

a) Generates an equal number of a's and b's

b) Generates an inherently ambiguous language

c) Generates more a's than b's

d) Generates an unequal number of a's and b's

Question 19

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

Question 20

Q Consider the CFG G (V, T, P, S) with the following production

$S \rightarrow AB \mid a$

$A \rightarrow a$

Let CFG G' is an equivalent CFG with no useless symbols. How many minimum productions will be there in G'?

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

Question 21

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

S → AB
A → ϵ / a / aS
B → b / bS

a) S → AB A → a / aS B → b / bS	**b)** S → A / B / AB A → a / aS B → b / bS
c) S → B / AB A → a / aS B → b / bS	**d)** Not possible to remove ϵ production

---

Question 22

Q. To obtain a string of n Terminals from a given Chomsky normal form grammar, the number of productions to be used is: **(NET-JULY-2018)**

(a) $2n-1$

(b) $2n$

(c) $n+1$

(d) n^2

Question 23

Q. If the parse tree of a word w generated by a Chomsky normal form grammar has no path of length greater than i, then the word w is of length **(NET-DEC-2012)**

(A) no greater than 2^{i+1}

(B) no greater than 2^{i}

(C) no greater than 2^{i-1}

(D) no greater than i

Question 24

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

(A) S → aSB B → bB / λ	**(B)** S → aSB B → bB / b
(C) S → aSB / b B → b	**(D)** S → aSb / b

Question 25

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

(A) (i) and (ii)

(B) (i) and (iii)

(C) (ii) and (iii)

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

(i)	(ii)	(iii)
S → a / bA / aA / bB A → a B → b	S → a / aA / AB A → a B → b	S → a / A / aA A → a

Question 26

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

(A) Removing left recursion alone

(B) Removing the grammar alone

(C) Removing left recursion and factoring the grammar

(D) None of the above

Question 27

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)**

Question 28

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, ∉}

Question 29

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

Question 30

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

Question 31

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 → a A b B | b A a B | ε
A → S
B → S

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

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

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

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

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

Question 32

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 → a A b B | b A a B | ε
A → S
B → S

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

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

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

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

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

Question 33

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)**

$S \rightarrow iCtSS_1 \mid a$

$S_1 \rightarrow eS \mid \epsilon$

$C \rightarrow b$

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.

Question 34

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 → FR} and {R → ε }

(B) {S → FR} and { }

(C) {S → FR} and {R → *S}

(D) {F → id} and {R → ε}

Question 35

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

Question 36

Q Consider the augmented grammar with {+, *, (,), id} as the set of terminals.
S' → S
S → S + R | R
R → R* P | P
P → (S) | 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)**

Question 37

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

Question 38

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

(B) n, E + n and E + E × n

(C) n, n + n and n + n × n

(D) n, E + n and E × n

Question 39

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

Question 40

Q Consider the following augmented grammar with {#,@,<,>,a,b,c} as the set of terminals.
S' → S
S → S#cS
S → SS
S → S@
S → <S>
S → a
S → b
S → c
Let I0=CLOSURE({S'→·S}). The number of items in
the set GOTO(GOTO(I0 <), <) is _____ .**(GATE 2021) (2 MARKS)**

Question 41

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.

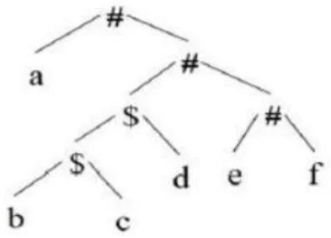
Question 42

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

1.	P → Q R	**(A)** 1 only
2.	P → Q s R	**(B)** 1 and 3 only
3.	P → ε	**(C)** 2 and 3 only
4.	P → Q t R r	**(D)** 3 and 4 only

Question 43

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

Question 44

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 _____.

Question 45

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

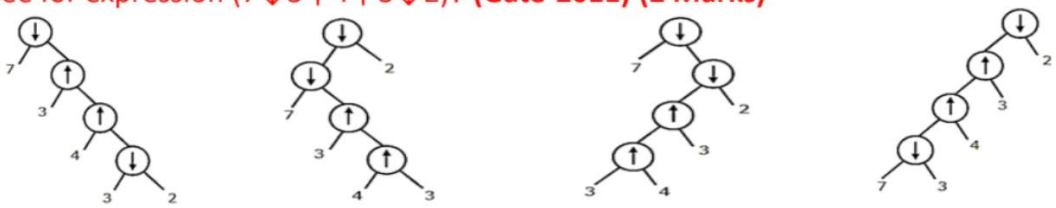
S --> T * P
T --> U | T * U
P --> Q + P | Q
Q --> Id
U --> Id

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

Question 46

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)**



Question 47

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' → S]}). The number of items in the set GOTO $(I_0, ( )$ is _____.

Question 48

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 **c)** III only

b) II only **d)** II and III only

Question 49

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.

Question 50

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	Grammar G_2
D → int L;	D → int L;
L → id [E	L → id E
E → num]	E → E[num]
E → num] [E	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

Question 51

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.

Question 52

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 **(c)** S_1 is true and S_2 is true

(b) S_1 is false and S_2 is true **(d)** S_1 is false and S_2 is false

Question 53

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

Question 54

Q Consider the following grammar G.

S → F / H
F → p / c
H → d / c

Where S, F and H are non-terminal symbols, p, d and c are terminal symbols.
Which of the following statement(s) is/are correct? **(Gate-2015) (1 Marks)**

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

Question 55

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

Question 56

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)

Question 57

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

Question 58

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

S → C C

C → c C | d

The grammar is

(A) LL(1)

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

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

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

Question 59

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

Question 60

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

Question 61

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

Question 62

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 $id.val$ 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 $S.val$ for root of the parse tree for the expression 20 #10%5 #8%2%2 is _____. **(GATE 2022) (2 MARKS)**

Question 63

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 **(C)** 160

(B) 180 **(D)** 40

Question 64

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

(A) 9 + 5 + 2
(B) 9 5 + 2 +
(C) 9 5 2 + +
(D) + + 9 5 2

Question 65

Q Consider the following translation scheme. **(Gate-2006) (2 Marks)**
S → ER
R → *E{print("*");}R | ε
E → F + E {print("+");} | F
F → (S) | id {print(id.value);}
Here id is a token that represents an integer and id.value represents the corresponding integer value. For an input '2 * 3 + 4', this translation scheme prints

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

Question 66

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

Question 67

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

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

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

Question 68

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

Question 69

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

Question 70

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

Question 71

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

Question 72

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)**

Question 73

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.

Question 74

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

Question 75

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)**

Question 76

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)**

Question 77

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

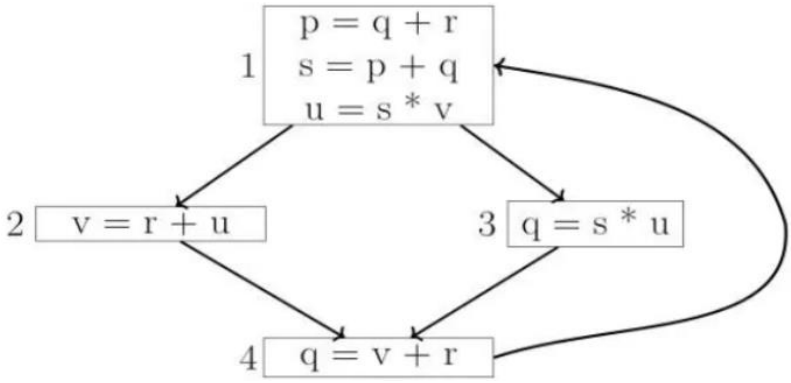
Question 78

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

ANSWER KEY

1	C	**17**	B	**33**	C	**49**	B	**65**	D
2	B	**18**	A	**34**	A	**50**	A	**66**	C
3	C	**19**	C	**35**	A	**51**	D	**67**	B
4	A	**20**	A	**36**	5	**52**	C	**68**	B
5	A	**21**	C	**37**	D	**53**	C	**69**	B
6	C	**22**	A	**38**	D	**54**	D	**70**	B
7	A	**23**	C	**39**	C	**55**	D	**71**	A
8	C	**24**	C	**40**	8	**56**	C	**72**	6
9		**25**	C	**41**	D	**57**	A	**73**	C
10	C	**26**	D	**42**	B	**58**	A	**74**	B
11	D	**27**	31	**43**	A	**59**	A	**75**	10
12	B	**28**	C	**44**	9	**60**	B	**76**	8
13	D	**29**	A	**45**	B	**61**	B	**77**	B
14	C	**30**	D	**46**	B	**62**	80	**78**	C
15	C	**31**	A	**47**	5	**63**	C		
16	C	**32**	C	**48**	A	**64**	B		