APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

STUDY MATERIALS

KTU ASSIST

a complete app for ktu students

Get it on Google Play

www.ktuassist.in

DOWNLOADED FROM WWW.KTUASSIST.IN

# Module 2

# Syntax Analysis & Top-Down Parsing

## SYNTAX ANALYSIS

**Ques 1) What is syntax analysis?**
**Or**
**What is parsing?**

**Ans: Syntax Analysis /Parsing**
Syntax analysis (also called **parsing**) is the process of determining if a string of tokens can be generated by a grammar. The goals of parsing are to check the validity of a source string, and to determine its syntactic structure.

1) For an **invalid string** the parser issues diagnostic messages reporting the cause and nature of errors in the string.
2) For **valid string** it builds a parse tree to reflect the sequence of derivations or reductions performed during parsing.

Parsing has much in common with scanning. Where lexical analysis splits the input into tokens, the purpose of syntax analysis is to recombine these tokens. Not back into a list of characters, but into something that reflects the structure of the text. This "something" is typically a data structure called **syntax tree** of the text. As the name indicates, this is a tree structure. The leaves of this tree are the tokens found by the lexical analysis, and if the leaves are read from left to right, the sequence is the same as in the input text. Hence, what is important in the syntax tree is how these leaves are combined to form the structure of the tree and how the interior nodes of the tree are labelled.

In addition to finding the structure of the input text, the syntax analysis must also reject invalid texts by reporting syntax errors. As syntax analysis is less local in nature than lexical analysis, more advanced methods are required. We, however, use the same basic strategy – a notation suitable for human understanding is transformed into a machine-like low-level notation suitable for efficient execution. This process is called parser generation. **For example,** let consider the following statement:

c = a + b * 5

**Syntax tree** can be given below:

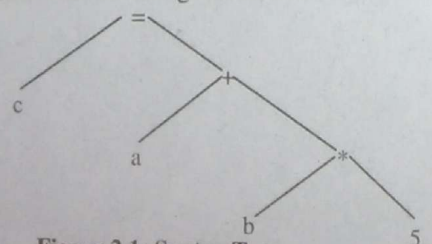



**Figure 2.1: Syntax Tree**

**Ques 2) What is difference between lexical analysis and syntax analysis?**

**Ans: Difference between Lexical Analysis and Syntax Analysis**
**Table below** shows the difference between lexical analysis and syntax analysis:

| Basis | Lexical Analysis | Syntax Analysis |
|---|---|---|
| **Simplicity** | Techniques for lexical analysis are less complex than those required for syntax analysis, so the lexical-analysis process can be simpler if it is separate. | Removing the low-level details of lexical analysis from the syntax analyzer makes the syntax analyzer both smaller and cleaner. |
| **Efficiency** | Optimise the lexical analyzer, because lexical analysis requires a significant portion of total compilation time | It is not fruitful to optimise the syntax analyzer |
| **Portability** | Because the lexical analyzer reads input program files and often includes buffering of that input, it is somewhat platform-dependent. | However, the syntax analyzer can be platform-independent. |

**Ques 3) What is parser? Also discuss its types. What is the role of Parser?**

**Ans: Parser**
Parser is a program that obtains tokens from lexical analyzer and constructs the parse tree which is passed to the next phase of compiler for further processing. Parser implements context free grammar for performing error checks.

A parser is a compiler or interpreter component that breaks data into smaller elements for easy translation into another language. A parser takes input in the form of a sequence of tokens or program instructions and usually builds a data structure in the form of a parse tree or an abstract syntax tree.

DOWNLOADED FROM WWW.KTUASSIST.IN

Scanned by CamScanner

**Types of Parser**

1) **Top Down Parsers:** Top down parsers construct parse tree from root to leaves.
2) **Bottom Up Parsers:** Bottom up parsers construct parse tree from leaves to root.

**Role of Parser**

**Figure 2.2** depicts the role of parser with respect to other phases.

1) Once a token is generated by the lexical analyzer, it is passed to the parser.
2) On receiving a token, the parser verifies the string of token names that can be generated by the grammar of source language.
3) It calls the function getNextToken(), to notify the lexical analyzer to yield another token.
4) It scans the token one at a time from left to right to construct the parse tree.
5) It also checks the syntactic constructs of the grammar.

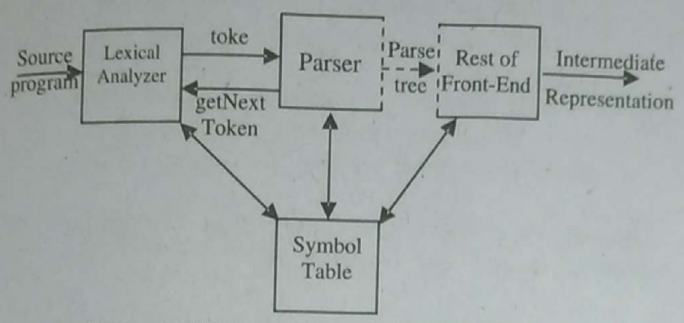



**Figure 2.2: Position of Parser in Compiler Model**

**Ques 4) What is grammar? Explain production rule of grammar.**

**Ans: Grammars**

Grammar is that part of the description of the language which answers the question: which phrases are correct? Once the alphabet of a language has been defined as a first step (in the case of natural language, for example, the Latin alphabet of 22or 26 letters, the Cyrillic alphabet, etc.), the lexical component, which uses this alphabet, identifies the sequence of symbols constituting the words (or **tokens**) of the language defined.

When alphabet and words have been defined, the **syntax** describes which sequences of words constitute legal phrases.

**Syntax** is therefore a relation between signs. Between all possible sequences of words (over a given alphabet), the syntax chooses a subset of sequences to form phrases of the language proper.

**Semantics** is that part of the description of the language which seeks to answer the question "what does a correct phrase mean?"

The lexical and syntactic features of a programming language are specified by its grammar. Language is simply a set of strings involving symbols from some alphabet. A language L can be considered to be a collection of valid sentences. Each sentence can be looked upon as a sequence of words and each word as a sequence of letters or graphic symbols acceptable in L.

A language specified in this manner is known as a formal language. A formal language grammar is a set of rules which precisely specify the sentences of L. It is clear that natural languages are not formal languages due to their rich vocabulary. However, PLs are formal languages.

Any set of strings over an alphabet $\Sigma$ is called a language. The set of all strings, including the empty string over an alphabet $\Sigma$ is denoted as $\Sigma^*$.

**For example,**

1) $L_1 = \{w \in (0, 1)^*$: w has an equal number of 0's and 1's$\}$,
2) $L_2 = \{w \in \Sigma^*: w = w^R\}$ where, $w^R$ is the reverse string of w.

**Production Rule**

A production rule is a statement of programming logic that specifies the execution of one or more actions in the case that its conditions are satisfied.

Production rules therefore have an operational semantic (formalizing state changes, e.g., on the basis of a state transition system formalism).

The effect of executing production rules may depend on the ordering of the rules, irrespective of whether such ordering is defined by the rule execution mechanism or the ordered representation of the rules.

The production rule is typically represented as:

**if [condition] then [action-list]**

Some implementations extend this definition to include an "else" construct as follows:

**if [condition] then [action-list] else [alternative-action-list]**

A grammar is defined by production rules that specify which lexemes may replace which other lexemes; these rules may be used to generate strings, or to parse them. Each such rule has a head, or left-hand side, which consists of the string that may be replaced, and a body, or right-hand side, which consists of a string that may replace it. Rules are often written in the form <head → body>; e.g., the rule z0 → z1 specifies that z0 can be replaced by z1.

A grammar G consists of the following **components**:

1) A finite set N of non-terminal symbols.
2) A finite set $\Sigma$ of terminal symbols that is disjoint from N.

DOWNLOADED FROM WWW.KTUASSIST.IN

Scanned by CamScanner

3) A finite set P of production rules, each rule of the form:

$$(\Sigma \cup N)^* N(\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*$$

where,

$*$ = Kleene star operator
$\cup$ = set union.

That is, each production rule maps from one string of symbols to another, where the first string contains atleast one non-terminal symbol. In the case that the body consists solely of the empty string – i.e., that it contains no symbols at all – it may be denoted with a special notation (often $\Lambda$, e or $\varepsilon$) in order to avoid confusion.

4) A distinguished symbol $S \in N$ that is the start symbol.

**Ques 5) What is context free grammar? Write its definition.**

**Ans: Context Free Grammars (CFG)**
Context-free grammars are used as the basis in the design and implementation of a compiler. The designers of compilers often use such grammars to implement components of a compiler, like parsers, scanners, code generators, etc.

Any implementation of programming language is preceded by a context-free grammar that specifies it.

A context-free grammar (CFG), is a grammar which naturally generates a formal language in which clauses can be nested inside clauses arbitrarily deeply, but where grammatical structures are not allowed to overlap.

Context Free Grammar (CFG) is a set of recursive rewriting rules (or productions) used to generate patterns of strings For the syntactic specification of programming Language, a notation which is used called **Context Free Grammars (CGF)** which is also sometimes called **BNF (Backus-Naur Form)** description.

**Formal Definition of CFG**
A context-free grammar is given as follows:

1) An alphabet $\Sigma$ of **terminal symbols**, also called the object alphabet;
2) An alphabet N of **non-terminal symbols**, the elements of which are also referred to as auxiliary characters, placeholders or, variables, where $N \cap \Sigma = \phi$;
3) A special non-terminal symbol $S \in N$ called the **start symbol**;
4) A finite set of **production rules**, that is strings of the form $R \rightarrow \Gamma$ where, $R \in N$ is a non-terminal symbol and $\Gamma \in (\Sigma \cup N)^*$ is an arbitrary string of terminal and non-terminal symbols, which can be read as 'R can be replaced by $\Gamma$'.

**For example,** let G be the grammar = $(W, \Sigma, R, S)$
where,
$W = \{S, A, N, V, P\} \cup \Sigma$,
$\Sigma = \{$Jim, big, green, cheese, ate$\}$,
$R = \{P \rightarrow N$,
$P \rightarrow AP$,
$S \rightarrow PV\,P$,
$A \rightarrow$ big,
$A \rightarrow$ green,
$N \rightarrow$ cheese,
$N \rightarrow$ Jim,
$V \rightarrow$ ate$\}$

Here, G is designed to be a grammar for a part of English; S stands for sentence, A for adjective, N for noun, V for verb, and P for phrase.

Following are some strings in L(G):
Jim ate cheese
big Jim ate green cheese
big cheese ate Jim

Unfortunately, the following are also strings in L(G):
big cheese ate green green big green big cheese
green Jim ate green big Jim

**Ques 6) Construct a context-free grammar G generating all integers (with sign).**

**Ans:** Let $G = (V_N, \Sigma, P, S)$

where,
$V_N = \{S, \langle sign\rangle, \langle digit\rangle, \langle Integer\rangle\}$
$\Sigma = \{0, 1, 2, 3, \ldots, 9, +, -\}$

P consists of $S \rightarrow \langle sign\rangle\ \langle integer\rangle$, $\langle sign\rangle \rightarrow + \mid -$,
$\langle integer\rangle \rightarrow \langle digit\rangle\ \langle integer\rangle \mid \langle digit\rangle$
$\langle digit\rangle \rightarrow 0 \mid 1 \mid 2 \mid \ldots \mid 9$

L(G) = the set of all integers. **For example,** the derivation of –17 can be obtained as follows:
$S \Rightarrow \langle sign\rangle\ \langle integer\rangle \Rightarrow -\langle integer\rangle$
$\Rightarrow -\langle digit\rangle\ \langle integer\rangle \Rightarrow -1\langle integer\rangle \Rightarrow -1\langle digit\rangle$
$\Rightarrow -17$

**Ques 7) Consider the context-free grammar $G = (V, \Sigma, R, S)$, where $V = \{S, a, b\}$, $\Sigma = \{a, b\}$, and R consists of the rules $S \rightarrow aSb$ and $S \rightarrow e$. Derive a string aabb from the given CFG.**

**Ans:** A possible derivation is

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$$

Here the first two steps used the rule $S \rightarrow aSb$, and the last used the rule $S \rightarrow e$. In fact, it is not hard to see that $L(G) = \{a^n b^n : n \geq 0\}$. Hence some context-free languages are not regular.

DOWNLOADED FROM WWW.KTUASSIST.IN

Scanned by CamScanner

**Ques 8) Let G = (V, Σ, R, E) where V, Σ, and R are as follows:**

**V = {+, *, (,), id, T, F, E},**

**Σ = {+, *, (,), id},**

**R = {E → E + T, .....(R1)**
**E → T, ..... (R2)**
**T → T * F ..... (R3)**
**T → F ..... (R4)**
**F → (E), ..... (R5)**
**F → id} ..... (R6)**

**The symbols E, T, and F are abbreviations for expression, term, and factor, respectively.**

**Derive the string (id * id + id) * (id + id) by using the given rules.**

**Ans:** The grammar G generates the string (id * id + id) * (id + id) by the following derivation:

| | |
|---|---|
| E ⇒ T | by Rule R2 |
| ⇒ T * F | by Rule R3 |
| ⇒ T * (E) | by Rule R5 |
| ⇒ T * (E + T) | by Rule R1 |
| ⇒ T * (T + T) | by Rule R2 |
| ⇒ T * (F + T) | by Rule R4 |
| ⇒ T * (id + T) | by Rule R6 |
| ⇒ T * (id + F) | by Rule R4 |
| ⇒ T * (id + id) | by Rule R6 |
| ⇒ F * (id + id) | by Rule R4 |
| ⇒ (E) * (id + id) | by Rule R5 |
| ⇒ (E + T) * (id + id) | by Rule R1 |
| ⇒ (E + F) * (id + id) | by Rule R4 |
| ⇒ (E + id) * (id + id) | by Rule R6 |
| ⇒ (T + id) * (id + id) | by Rule R2 |
| ⇒ (T * F + id) * (id + id) | by Rule R3 |
| ⇒ (F * F + id) * (id + id) | by Rule R4 |
| ⇒ (F * id + id) * (id + id) | by Rule R6 |
| ⇒ (id * id + id) * (id + id) | by Rule R6 |

**Ques 9) Define the terms derivation and reduction.**

**Ans: Derivation, Reduction and Parse Trees**

A grammar G is used for two purposes, to generate valid strings of Lg and to 'recognize' valid strings of lg. The derivation operation helps to generate valid strings while the reduction operation helps to recognize valid strings.

A parse tree is used to depict the syntactic structure of a valid string as it emerges during a sequence of derivations or reductions.

1) **Derivation:** Let production $P_1$ of grammar G be of the form

**$P_1$: A ::= α**

and let β be a string such that β ≡ γAθ, then replacement of A by α in string β constitutes a derivation according to production $P_1$. We use the notation N ⇒ η to denote direct derivation of η from N and N $\overset{*}{\Rightarrow}$ η to denote transitive derivation of η (i.e., derivation in zero or more steps) from N, respectively. Thus, A ⇒ α only if A ::= α is a production of G and A $\overset{*}{\Rightarrow}$ δ if A ⇒ ...⇒ δ. We can use this notation to define a valid string according to a grammar G as follows: δ is a valid string according to G only if S $\overset{*}{\Rightarrow}$ δ, where S is the distinguished symbol of G.

**For example,** Grammar given below defines a language consisting of noun phrases in English

<Noun Phrase> :: = <Article> <Noun>

<Article> :: = a | an | the .....(1)
<Noun> :: = child | mango

<Noun Phrase> is the distinguished symbol of the grammar, the child and a mango are some valid strings in the language.

**For example,** Derivation of the string "the child" according to grammar (1) shown above, can be depicted as:

<Noun Phrase>⇒ <Article> <Noun>
⇒ the <Noun>
⇒ the child

A string α such that S $\overset{*}{\Rightarrow}$ α is a sentential form of $L_G$. The string α is a sentence of $L_G$ if it consists of only Ts.

2) **Reduction:** Let production $P_1$ of grammar G be of the form:

**$P_1$ : A ::= α**

and let σ be a string such that σ ≡ γαθ, then replacement of α by A in string σ constitutes a reduction according to production $P_1$. We use the notations η → N and η $\overset{*}{\rightarrow}$ N to depict direct and transitive reduction, respectively. Thus, α → A only if A ::= α is a production of G and α $\overset{*}{\Rightarrow}$ A if α → ... → A. We define the validity of some string δ according to grammar G as follows: δ is a valid string of $L_G$ if δ $\overset{*}{\Rightarrow}$ S, where S is the distinguished symbol of G.

**For example,** To determine the validity of the string, "the child ate an mango" according to grammar (2) **shown below**, we perform the following reductions:

Grammar G:

<Sentence> :: = <Noun Phrase> <Verb Phrase>
<Noun Phrase> :: = <Article> <Noun>

DOWNLOADED FROM WWW.KTUASSIST.IN

Scanned by CamScanner

<Verb Phrase> :: = <Verb><Noun Phrase> .....(2)
<Article> :: = a | an | the
<Noun> :: = child | mango

| Step | String |
|---|---|
| 0 | the child ate an apple |
| 1 | <Article> child ate an apple |
| 2 | <Article> <Noun> ate an mango |
| 3 | <Article> <Noun> <Verb> an mango |
| 4 | <Article> <Noun> <Verb> <Article> mango |
| 5 | <Article> <Noun> <Verb> <Article> <Noun> |
| 6 | <Noun Phrase> <Verb> <Article> <Noun> |
| 7 | <Noun Phrase> <Verb> <Noun Phrase> |
| 8 | <Noun Phrase> <Verb Phrase> |
| 9 | <Sentence> |

The string is a sentence of $L_G$, since we are able to construct the reduction sequence:

the child ate an mango $\xrightarrow{*}$ <Sentence>.

**Ques 10) What is parse tree or derivation tree?**

**Ans: Parse Trees (Derivation Trees)**
Grammar can be represented using trees. Such trees representing derivations are called **derivation trees**.

A derivation tree (also called a parse tree) for a $G = (V_N, \Sigma, P, S)$ is a tree satisfying the following:

1) Every vertex has a label which is a variable or terminal or $\Lambda$.
2) The root has label S.
3) The label of an internal vertex is a variable.
4) If the vertices $n_1, n_2, \ldots, n_k$ written with labels $X_1, X_2, \ldots, X_k$ are the sons of vertex n with label A, then $A \rightarrow X_1X_2 \ldots X_k$ is a production in P.
5) A vertex n is a leaf if its label is $a \in \Sigma$ or $\Lambda$ ; n is the only son of its father if its label is $\Lambda$.

**For example,** let $G = (\{S, A\}, \{a, b\}, P, S)$, where P consists of $S \rightarrow aAS \mid a \mid SS$, $A \rightarrow SbA \mid ba$.

**Figure 2.3** is an example of a derivation tree.

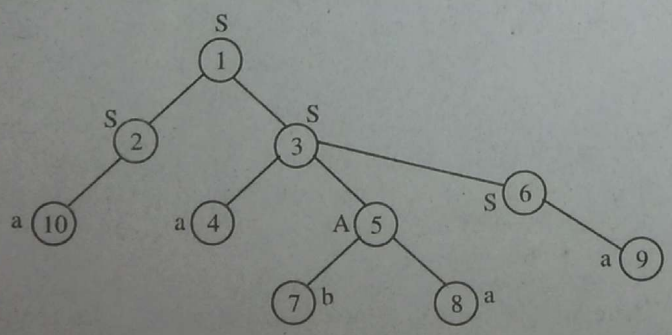


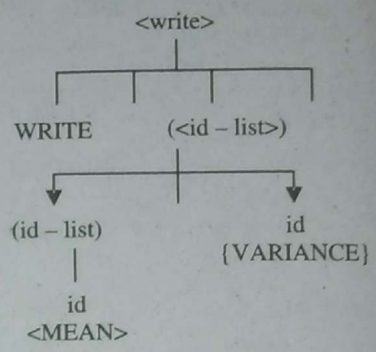

**Figure 2.3: Derivation Tree**

**Note:** Vertices 4-6 are sons of 3 written from the left, and $S \rightarrow aAS$ is in P. Vertices 7 and 8 are sons of 5 written from the left, and $A \rightarrow ba$ is a production in P. Vertex 5 is an internal vertex and its label is A, which is a variable.

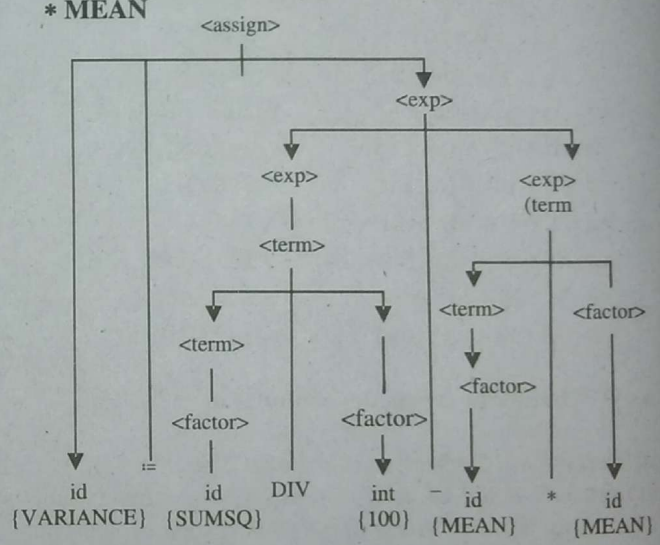
In **figure 2.3**, e.g., the sons of the root are 2 and 3 ordered from the left. So, the son of 2, *viz.*, 10, is to the left of any son of 3. The sons of 3 ordered from the left are 4-5-6. The vertices at level 2 in the left-to-right ordering are 10-4-5-6. 4 is to the left of 5. The sons of 5 ordered from the left are 7-8. So 4 is to the left of 7. Similarly, 8 is to the left of 9. Thus the order of the leaves from the left is 10-4-7-8-9.

**Ques 11) Write parse tree for the following statements:**
**1) WRITE (MEAN, VARIANCE)**
**2) VARIANCE : = SUM DIV 100 – MEAN * MEAN**

**Ans: Parse Tree for WRITE (MEAN, VARIANCE)**




**Parse Tree for VARIANCE:= SUM DIV 100 – MEAN * MEAN**




**Ques 12) Consider G whose productions are $S \rightarrow aAS \mid a$, $A \rightarrow SbA \mid SS \mid ba$. Show that $S \overset{*}{\Rightarrow} aabbaa$ and construct a derivation tree whose yield is aabbaa.**

**Ans:** $S \Rightarrow aAS \Rightarrow aSbAS \Rightarrow aabAS \Rightarrow a^2bbaS \Rightarrow a^2b^2a^2$ .....(1)

Hence, $S \overset{*}{\Rightarrow} a^2b^2a^2$. The derivation tree is given in **figure 2.4**.

DOWNLOADED FROM WWW.KTUASSIST.IN

Scanned by CamScanner

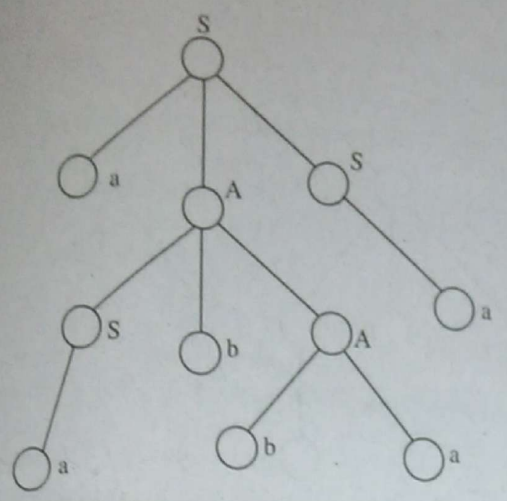



**Figure 2.4: Derivation Tree with Yield aabbaa**

Another derivation of $a^2b^2a^2$ is

$S \Rightarrow aAS \Rightarrow aAa \Rightarrow aSbAa \Rightarrow aSbbaa \Rightarrow aabbaa$ ..... (2)

Yet another derivation of $a^2b^2a^2$ is

$S \Rightarrow aAS \Rightarrow aSbAS \Rightarrow aSbAa \Rightarrow aabAa \Rightarrow aabbaa$ ..... (3)

In derivation (1), whenever we replace a variable X using a production, there are no variables to the left of X. In derivation (2), there are no variables to the right of X. But in (3), no such conditions are satisfied.

**Ques 13) Let G be the grammar S → 0B | 1A, A→ 0| 0S| 1AA, B → 1| 1S| 0BB. For the string 00110101, find:**

1) **Leftmost derivation,**
2) **Rightmost derivation, and**
3) **Derivation tree.**

**Ans:**

1) $S \Rightarrow 0B \Rightarrow 00BB \Rightarrow 001B \Rightarrow 0011S$
$\Rightarrow 0^21^20B \Rightarrow 0^21^201S \Rightarrow 0^21^2010B \Rightarrow 0^21^20101$

2) $S \Rightarrow 0B \Rightarrow 00BB \Rightarrow 00B1S \Rightarrow 00B10B$
$\Rightarrow 0^2B101S \Rightarrow 0^2B1010B \Rightarrow 0^2B10101$
$\Rightarrow 0^2110101.$

3) The derivation tree is given in **figure 2.5**.

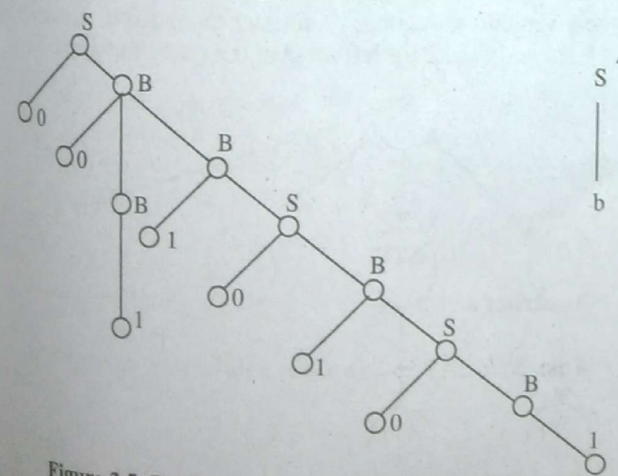



**Figure 2.5: Derivation Tree with Yield 00110101**

**Ques 14) What is ambiguity?**

**Ans: Ambiguity**

A grammar that produces more than one parse tree for same string is said to be **ambiguous.** In other words an ambiguous grammar is one that produces more than one left most or more than one right most derivation tree for same string.

For some context free grammars, it is possible to find a terminal string with more than one parse tree or equivalently, more than one left most derivation or more than one right most derivation. Such a grammar is called ambiguous.

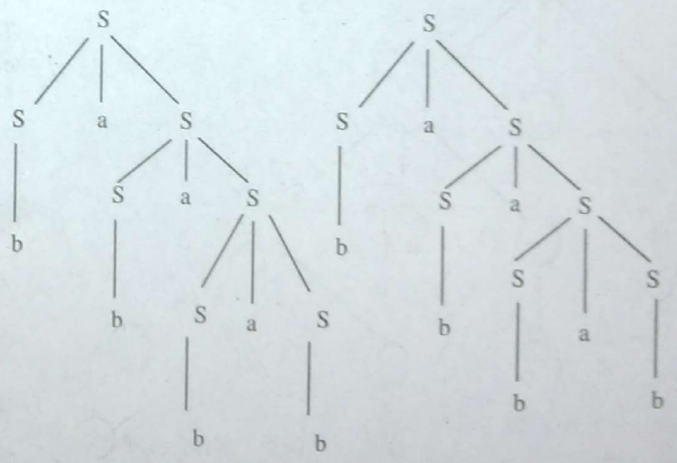
For many useful grammars, such as those that describe the structure of programs in a typical programming language, it is possible to find an unambiguous grammar that generates the same language.

Unfortunately, the unambiguous grammar is frequently more complex than the simplest ambiguous grammar for the languages. There are also some context free languages that are inherently ambiguous, meaning that every grammar for that language is ambiguous.

**Ques 15) Consider a grammar G, S → SaS | b, show that G is ambiguous for string 'bababab'.**

**Ans:** To prove that G is ambiguous, we have to find W∈ L(G) which is ambiguous.

| Apply left most derivation | Right most derivation |
|---|---|
| S → SaS | S → SaS |
| → baS | → SaSaS |
| → baSaS | → baSaS |
| → babaS | → babaS |
| → babaSaS | → babaSaS |
| → bababaS | → bababaS |
| → bababab | → bababab |




There are two derivation tree for the string w = 'bababab'. So given CFG is ambiguous.

DOWNLOADED FROM WWW.KTUASSIST.IN

Scanned by CamScanner

**Ques 16) Consider G = ({S}, {a, b, +, *}, P, S), where P consists of S → S + S | S * S | a | b. Draw the possible ambiguous derivation trees.**

**Ans:** We have two derivation trees for a + a * b given in **figure 2.6**.

The leftmost derivations of a + a * b induced by the two derivation trees are,

$S \Rightarrow S+S \Rightarrow a+S \Rightarrow a+S*S \Rightarrow a+a*S \Rightarrow a+a*b$

$S \Rightarrow S*S \Rightarrow S+S*S \Rightarrow a+S*S \Rightarrow a+a*S \Rightarrow a+a*b$

Therefore, a + a * b is ambiguous.

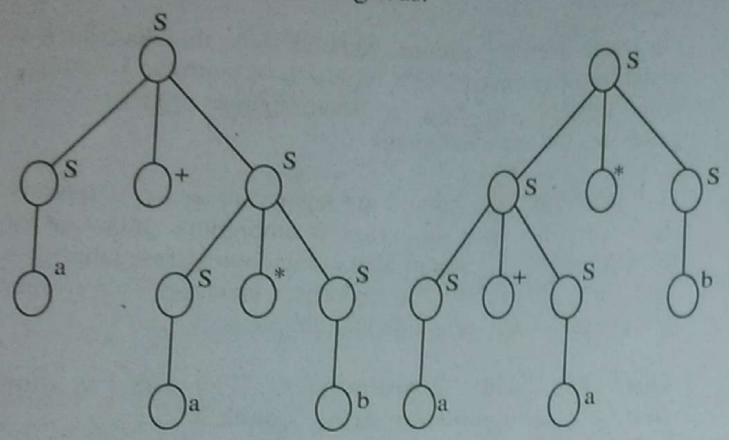
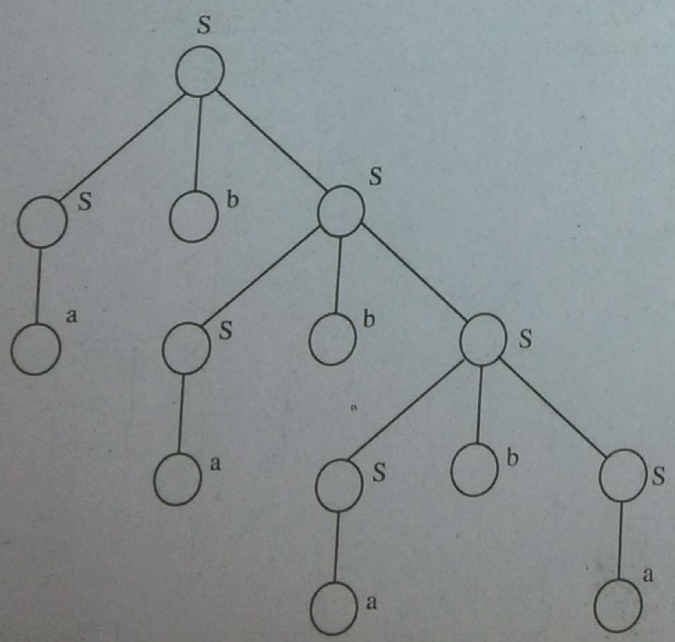



**Figure 2.6: Derivation Trees for a + a * b**

**Ques 17) If G is the grammar S → SbS | a, show that G is ambiguous.**

**Ans:** To prove that G is ambiguous, we have to find a w ∈ L (G), which is ambiguous. Consider w = abababa ∈ L(G).

Then we get two derivation trees for w (**figure 2.7**). Thus, G is ambiguous.



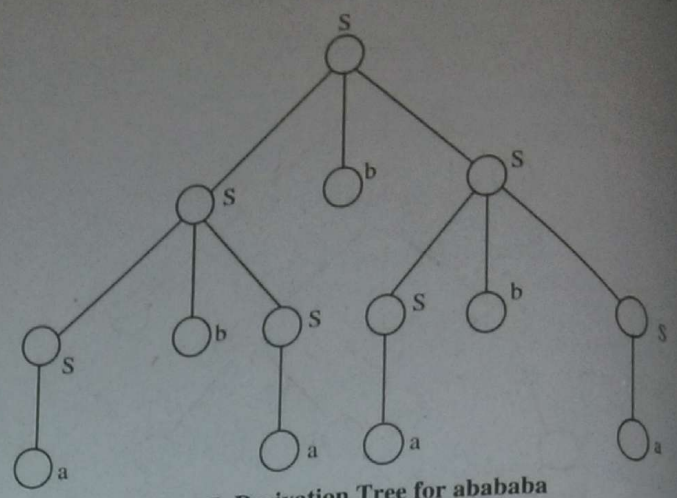




**Figure 2.7: Derivation Tree for abababa**

## TOP-DOWN PARSING

**Ques 18) Explain top-down parsing. List out the different forms of top-down parsers. Also list out the techniques of top-down parsing.**

**Ans: Top - Down Parsing**

Top-down parsing can be viewed as the problem of constructing a parse tree for the input string, starting from the root and creating the nodes of the parse tree in preorder.

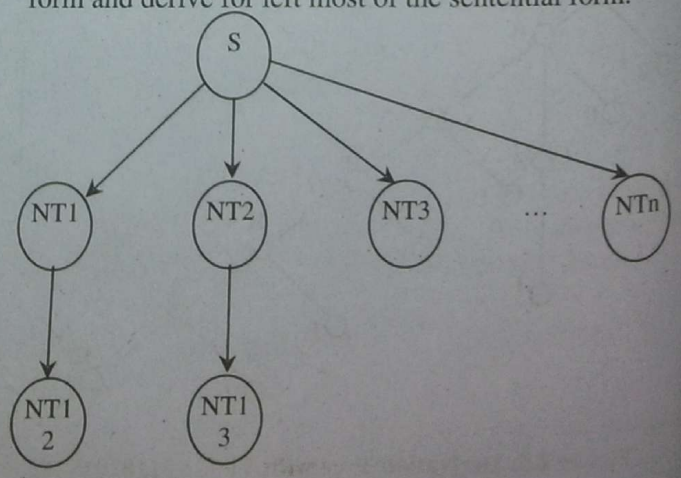
Equivalently, top-down parsing cab be viewed as finding a leftmost derivation for an input string.

The operations of construction of parse tree starting from the root node and proceeding toward leaves is called **top down parsing**, i.e., the top down parser attempts to derive a tree.

Give an input string by successive applications of grammar to the grammar's distinguished symbol. If **a** is an input string then

S → .. → .. → ..... → a

This is a top down loop approach. It is easy and convenient to choose a non-terminal from a sentential form and derive for left most of the sentential form.




DOWNLOADED FROM WWW.KTUASSIST.IN
Scanned by CamScanner

Hence Top Down parsing is called as **LL (left-to-left parsing)**. The input string is scanned by the parser (left to right one symbol token at a time) and leftmost derivation generates the leaves of the parse tree in left to right order, which matches against order of scanning of the input.

**Forms of Top-Down Parser**

Top-down parsers come in two forms:

1) Backtracking Parsers
2) Predicative Parsers

**Techniques of Top-Down Parsing**

There are two kinds of top-down parsing algorithms that are as below:

1) Recursive - Descent Parsing
2) LL(1) Parsing

**Ques 19) Discuss the drawbacks of top-down parsing.**

**Ans: Drawbacks of Top- Down Parsing**

1) **Infinite Looping:** Consider the grammar

   $X \xrightarrow{+} Xa$ for same a

   When we expand X, we may be just going on expanding X without having consumed any input.

   ($X \rightarrow Xa \rightarrow Xaa \rightarrow Xaaa$ like this)

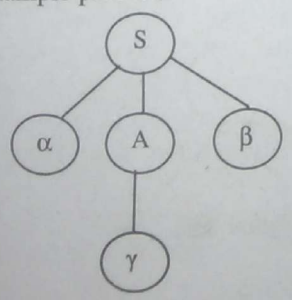
   This will surely result in erroneous input. Such types of grammars are called **Left Recursive** grammars.

2) **Left Recursion:** A grammar G is said to be left Recursive if the non-terminal X of G is such that there exists a derivation of the form $X \rightarrow X\alpha$ for some $\alpha$.

   Hence, by left recursion, the top down parse may go into infinite loop. Hence, left recursion must be eliminated.

3) **Back Tracking Problem:** If we repeatedly scan the erroneous inputs and discover a mismatch then the entries that are entered in the symbol table will be affected (add an element and delete an element). First they must be entered when scanned, after a mismatch these entries are to be removed. Such type of actions takes more work overhead and hence back tracking must be avoided. Recursive descent parser and predictive parsers are the types of top down parsers that avoids back tracking.

4) **Order of Alternatives:** Whatever alternatives we tried, may possibly affect the language accepted. Consider the same grammar that we did in the back tracking example problem.




   After we found $\alpha\ \gamma$, if next match symbol $\beta$ is not found, then the match for alternate $\alpha A \beta$ is wrong.

5) **No idea about errors:** We are really not in position to check where error has occurred while implementing the Top- down Parsing.

**Ques 20) Explain the backtracking parser.**

**Ans: Backtracking Parsers**

The process of repeated scans of the input string is called **back tracking**. A backtracking parser will go for different possibilities for a parse of the input.

When the parser generates the parse tree and the leaves, it may repeatedly scan the input several times to obtain the leaves which are leftmost derivate. If a non-terminal A is to derive a next left most derivation then there may be multiple productions as

$A \rightarrow \alpha 1 \mid \alpha 2 \mid \ldots\ldots \mid \alpha n$

Now this is a situation to decide the production says

$A \rightarrow \alpha 1$ or $A \rightarrow \alpha 2$ ..... Like this and so on.

Using the above productions and leftmost derivations the parser should finally lead to the derivation of a given string a, then it announces successful completion of parsing. Else the parser resets the pointer and tries for other productions and continues leftmost derivations until successful completion or failure comes. Hence a Top down parser may require to do back tracking.

**For example,** consider the grammar below:

$S \rightarrow \alpha A \beta$

$A \rightarrow \gamma \delta / \gamma$

Recall that a top down parser is viewed as an attempt to find the leftmost derivations from an input string. Now from the above grammar let us derive the string w.

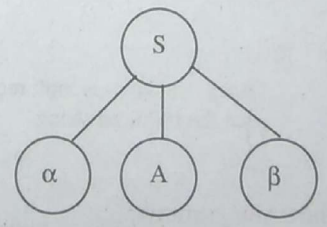
$w = \alpha \gamma \beta$

The leftmost derivation is:

$S \rightarrow \alpha \underline{A} \beta$

$\rightarrow \alpha \gamma \beta$

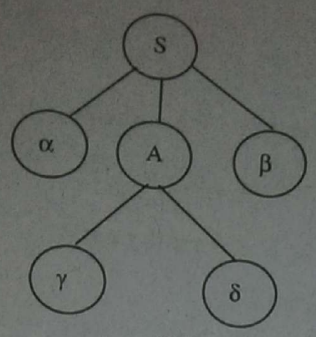
Similarly, to represent the above leftmost representation as a parse tree,




Leftmost leaf $\alpha$ is the terminal and is not a non-terminal. It matches with the first symbol of our desired input string $\alpha\ \gamma\ \beta$. The second symbol is A in the parse tree. Since A is a non-terminal we try to expand A as:

DOWNLOADED FROM WWW.KTUASSIST.IN

Scanned by CamScanner

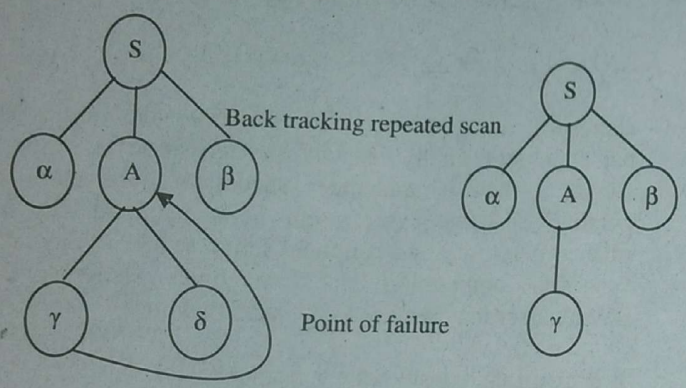



(Point of failure)

If we observe the parse tree, the last symbol after A is β, which is terminal and hence need not be derived.

But when we try to match our derived input string α γ β with the derived parse tree string α γ δ β, we report failure and go back to see the alternate choice for A (Alternate A production).

Hence, we reset the input pointer at the second symbol at A and use another production A → γ.

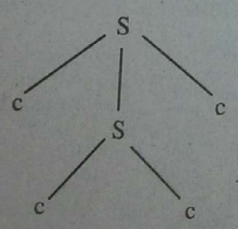



Since we have produced a parser tree for the string α γ β we halt and announce successful completion of parsing.

**Ques 21) Consider the grammar**
**S → cc/cSc**

**Parse the string cccc also explain the back tracking process.**

**Ans:** S → cc / cSc




Back tracking is not required because we got the required string.

Again consider the same grammar
S → cc / cSc

The parser that succeeds on two c's, four c's but not on six c's.

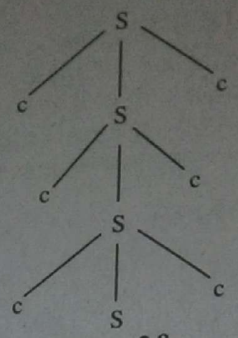



**Figure 2.8**

Now to get six c's, it is found that there is no match, and therefore it will back track to S as shown in the **figure 2.8**.

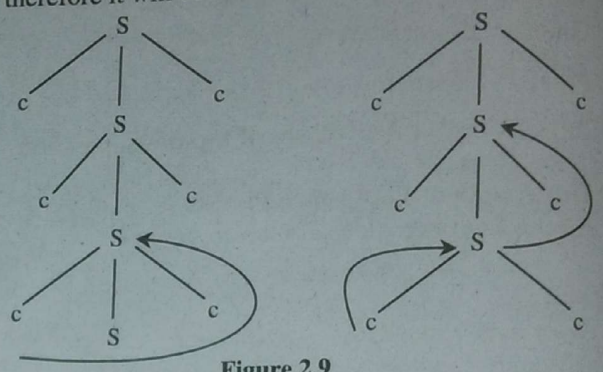



**Figure 2.9**

**Hint:** But since there is no match it again backtracks. Since, there is no production for S that can be tried.

It will backtrack one more step as shown in **figure 2.9**.

The parser then tries the alternate cc as **shown 2.10**:

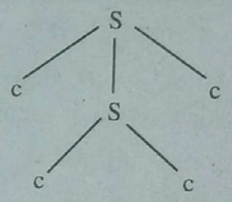



**Figure 2.10**

Since the parser again finds mismatch, hence it will backtrack as shown below:

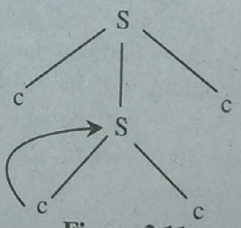



**Figure 2.11**

Since there is no alternate of production S to be tried, it will backtrack one more step as **shown 2.12**:

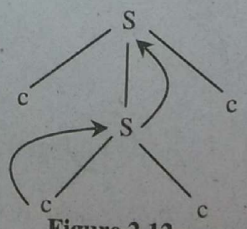



**Figure 2.12**

DOWNLOADED FROM WWW.KTUASSIST.IN

Scanned by CamScanner

Hence the required **parse tree** is:

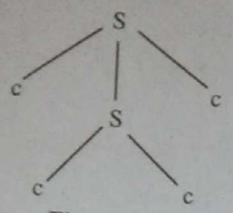



**Figure 2.13**

**Ques 22) Explain the concept of predictive parser. How one can construct the predictive parser table?**

**Or**

**Define predictive parsing. Also discuss the first and follow method to construct the predictive parser table.**

**Ans: Predictive Parsing**

It is a top-down parsing method, which consists of a set of mutually procedures to process the input and handles a stack of activation records explicitly.

Predictive parsing does not require backtracking in order to derive the input string. Predictive parsing is possible only for the class of LL(k) grammar(context-free grammar).

**Predictive parser** is the tabular representation of recursive descent parser. A predictive parser predicts the next construction in the input string by using one or more look-ahead token, i.e., these are table driven top down parsers used to eliminate backtracking.

Given a non-terminal the production should be made on the terminal that the non-terminal produces, along with the building a parse tree (which indicates that right hand side choices to be made for production).

Stacks are used in predictive parsing to avoid recursive procedures to store non-terminals in the present sequential form.

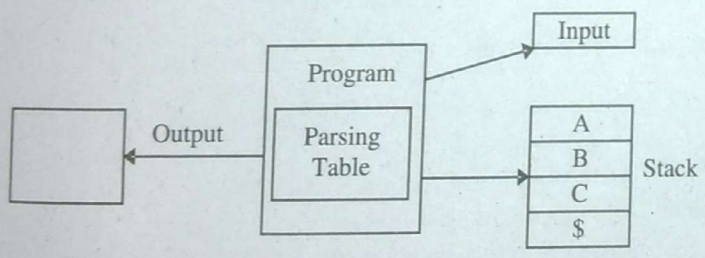



**Figure 2.14: Predictive Parser**

The predictive parser is the non-deterministic recognizer of the language generated by the grammar. If the top-down parser is made of predicting or detecting the alternative choices for expanding the non-terminal during the parsing, then the parser can be made deterministic. Hence,

1) First we have to write the grammar very carefully.
2) Eliminate the left recursion from the grammar.

Then we obtain the grammar that can be parsed by a top down parser, which will be able to predict, which is the right alternative for the expression of the non-terminal during the parsing process and hence needs no back-tracking.

The predictive parser has a stack input and output. The $ is placed at the end of the input string that is to be parsed, to denoted the end of the string. The stack initially does not contain any symbol of the input symbol except $.

**Construction of Predictive Parsing Tables**

Algorithm for construction of a predictive parsing table:

**Input:** Grammar G,

**Output**: Parsing Table M,

**Method:**

1) Arrange table M as two-dimensional array such that the terminals are on the columns and Non-Terminals are put in rows.
2) For each production of the form A→ α, 'do steps 3 and 4'.
3) For each terminal a in FIRST (α), add A→ α to M [A, a].
4) If ε is in FIRST (α), add A→ α to M [A, b] for each terminal b in FOLLOW (A). If ε is in FIRST (α) and $ is in FOLLOW (A), add A→ α to M [A, $].
5) Make each undefined entry of M as "ERROR".

**First and Follow**

First and Follow techniques is used to generating parsing table. The construction of both top-down and bottom-up parsers is aided by two functions, FIRST and FOLLOW, associated with a grammar G. For construction of predictive parsing table these two functions are needed.

1) **Computation of FIRST:** Let X is a terminal or non-terminal, then the set FIRST(X) consists of terminals (possibly ε) is defined as follows:

   **Rule 1)** If X is a terminal then FIRST (X) = {X}
   **Hint:** If there is a production of the form X → abc
   Then FIRST (X) = {a}

   **Rule 2)** If X is ε , i.e., X → ε then FIRST (X) = {ε}

   **Rule 3)** If α = XYZ then FIRST (α) = FIRST (XYZ) , if FIRST (X) not contains ε then Rule 2 else
   Now, if FIRST (X) contains ε then,

   FIRST (XYZ) = FIRST (X) – {ε} U FIRST (YZ)

i.e., except ε whatever in FIRST (X) and FIRST (YZ) will be in FIRST (XYZ). Now to compute first (YZ) same Rule 3 is applied.

FIRST (YZ) = FIRST (Y) – { ε} U FIRST (Z) if FIRST (Y) contains ε

DOWNLOADED FROM WWW.KTUASSIST.IN

Scanned by CamScanner

2) **Computation of FOLLOW:** The set FOLLOW (X) is defined as follows:

**Rule 1)** If X is a start symbol (distinguished symbol) then place $ in FOLLOW (X).

**Rule 2)** If there is a production of the form X → α B β then FOLLOW (B) = FIRST (β) – {ε} i.e., everything of FIRST (β) except ε.

**Rule 3)** If there is a production of the form X → α B β and FIRST (β) contains ε then everything in FOLLOW (X) is in FOLLOW (B).

If there is a production of the form X → α B then everything in FOLLOW (X) is in FOLLOW (B).

**Ques 23) Consider the grammar**
**S → ACB /CbB / Ba**
**A → da/ BC**
**B → g/ ε**
**C → h/ ε**

**Calculate FIRST and FOLLOW.**

**Ans:** FIRST (S) = FIRST (ACB) = FIRST (CbB) = FIRST (Ba)

From a productions,
Now, FIRST (A) = FIRST (da) and
FIRST (A) = FIRST (BC)
FIRST (B) = {g, ε}
FIRST (C) = {h, ε}

By rule 3 in first definition.

FIRST (A) = {da, g, h, ε}

d is terminal
Hence include
[FIRST (da) by
applying rule 1]

FIRST (BC) = FIRST (B) – {ε} U
FIRST (C) as FIRST (B) contains ε,
That is, {g, ε} U {h, ε} = {g, h, ε}
From rule 3 definitions

Now, FIRST (S) = FIRST (ACB) = FIRST (CbB) = FIRST (Ba)

FIRST (ACB) = FIRST (A) – {∈} U FIRST (CB)
= {d, g, h} U FIRST (CB) since FIRST (A) contains ε as per rule 3

FIRST (CB) = FIRST (C) – {ε} U FIRST (B) as per rule 3 as FIRST (C) contains ε
= {h} U {g, ε} = { h, g, ε}

FIRST (ACB) = {d, g, h, ε}

Now, FIRST (CbB) = FIRST (C) – {∈} U FIRST (bB) as per rule 3
= {h, ε} – {ε} U {b} = {h, b, ε}

Again, FIRST (Ba) = FIRST (B) – {ε} U FIRST (a)

From rule 3 as FIRST (B) contains ε
= {g} U {a} = {g, a}

Hence,
FIRST (S) = {a, b, d, g, h, ε}
FIRST (A) = {d, g, h, ε}
FIRST (B) = {g, ε} FIRST (C) = {h, ε}

FOLLOW (S) = {$} FOLLOW (A) = {g, b} FOLLOW (B)
= {a, h, g, b, $}
FOLLOW (C) = {g, b}

DOWNLOADED FROM WWW.KTUASSIST.IN

Scanned by CamScanner

**Ques 24) Consider the grammar**

$E \rightarrow TE^1$
$E^1 \rightarrow +E / \varepsilon$
$T \rightarrow FT^1$
$T^1 \rightarrow T / \varepsilon$
$F \rightarrow PF^1$
$F^1 \rightarrow *F^1 / \varepsilon$
$P \rightarrow (E)/a/b/ \varepsilon$

**Compute FIRST function.**

**Ans:**

FIRST (E) = {(, a, b, *, +, ε}
FIRST ($E^1$) = {+, ε}
FIRST (T) = {(, a, b, *, ε}
FIRST ($T^1$) = {(, a, b, *, ε}
FIRST (F) = {(, a, b, *, ε}
FIRST ($F^1$) = {*, ε}
FIRST (P) = {(, a, b, ε}

**Ques 25) Consider the grammar :**

$E \rightarrow E+T / T$
$T \rightarrow T*F / F$
$F \rightarrow (E) / id$

**Construct the predictive parsing table.**

$E \rightarrow TE^1$
$E^1 \rightarrow +TE^1 / \varepsilon$
$T \rightarrow F\,T^1$
$T^1 \rightarrow *\,FT^1 / \varepsilon$
$F \rightarrow (E) / id$

**Ans:**

FIRST (E) = FIRST (T) = FIRST (F) = { (, id}
FIRST ($E^1$) = { +, ε}
FIRST ($T^1$) = { *. ε}
FOLLOW (E) = FOLLOW ($E^1$) = { ), $}
FOLLOW (T) = FOLLOW ($T^1$)= {+, ), $}
FOLLOW (F) = { +, *, ), $}

Let's construct the predictive parsing table M in which there are 6 terminals and 5 non-terminals.

Terminals in the Column side

| | | + | * | ( | ) | id | $ |
|---|---|---|---|---|---|---|---|
| Non- terminals in the row side | E | | | $E \rightarrow T\,E^1$ | | $E \rightarrow TE^1$ | |
| | $E^1$ | $E^1 \rightarrow TE^1$ | | | $E^1 \rightarrow \varepsilon$ | | $E^1 \rightarrow \varepsilon$ |
| | T | | | $T \rightarrow F\,T^1$ | | $T \rightarrow F\,T^1$ | |
| | $T^1$ | $T^1 \rightarrow \varepsilon$ | $T^1 \rightarrow *F\,T^1$ | | $T^1 \rightarrow \varepsilon$ | | $T^1 \rightarrow \varepsilon$ |
| | F | | | $F \rightarrow (E)$ | | $F \rightarrow id$ | |

**Explanation:** FIRST (T), FIRST (F), FIRST (E) contains { (, id}, hence place T, E, F productions in the respective terminal columns. In **id** column $F \rightarrow id$ is placed, in ( column $F \rightarrow (E)$ production is placed. First of $E^1$ and $T^1$ are having ε hence, $E^1 \rightarrow \varepsilon$ and $T^1 \rightarrow \varepsilon$ productions should be placed in follow of $E^1$ and $T^1$.

**Ques 26) Consider the grammar below.**

$S \rightarrow iEtSS_1 / a$
$S_1 \rightarrow eS / \varepsilon$
$E \rightarrow b$

**Construct the predictive parsing table.**

**Ans:**

FIRST (S) = { i, a}
FIRST ($S_1$) = {e, ε}
FIRST (E) = {b}

DOWNLOADED FROM WWW.KTUASSIST.IN

Scanned by CamScanner

FOLLOW (S) = { e, $}
FOLLOW ($S_1$) = {e, $}
FOLLOW (E) = {t}

Terminals in the Column side

| Non-terminals in the row side | | i | a | b | e | t | $ |
|---|---|---|---|---|---|---|---|
| | S | $S \rightarrow iEtSS_1$ | $S \rightarrow a$ | | | | |
| | $S_1$ | | | | $S_1 \rightarrow eS/\varepsilon$ | | $S_1 \rightarrow \varepsilon$ |
| | E | | | $E \rightarrow b$ | | | |

**Ques 27) Consider the following grammar.**
**$E \rightarrow TE^1$**
**$E^1 \rightarrow + TE^1 / \varepsilon$**
**$T \rightarrow FT^1$**
**$T^1 \rightarrow *F T^1 / \varepsilon$**
**$F \rightarrow (E) / id$**

**Let us take the string id +id *id and parse it using the predictive parsing algorithm.**

**Ans:** Initially the stack contains the starting symbol E along with $.

| Stack | Input | Explanation |
|---|---|---|
| $E | id + id * id$ | Stack initially contains the start symbol, hence according to the algorithm, T [A, s] = T [E, id] = **$E \rightarrow TE^1$** so, pop E from the stack, push the productions on to stack in reverse order, i.e., $E^1T$ (not $TE^1$) |
| $E^1 T | id + id *id $ | T on id, in the parsing table it is **$T \rightarrow FT^1$** then pop the symbol T and push $T^1F$ not ($FT^1$) |
| $E^1T^1F | id + id *id $ | The top symbol on the stack is again a non-terminal. Hence from the parsing table the production of F on id is **F → id**. Pop F, push id on the stack |
| $E^1T^1 id | id + id * id$ | Now, the top of the stack is a terminal and the next input symbol is also id, both are matching then according to the algorithm, pop id from the stack remove id from the Input symbol. |
| $E^1T^1 | + id * id $ | Now $T^1$ on + from the table it is ∈ hence pop $T^1$ we need not push ε as it is a null string) |
| $E^1 | + id * id $ | $E^1$, the top symbol on the stack on +, from the parsing table the corresponding production is **$E^1 \rightarrow +TE^1$** hence pop $E^1$ and push in reverse order |
| $E^1 T+ | + id * id $ | The top symbol on the stack is + and the next input symbol to be parsed is also + hence, pop + from the stack and remove + from the input symbol. |
| $E^1T | id * id $ | T on id from the parsing table it is **$T \rightarrow FT^1$** hence push in reverse order. |
| $E^1T^1F | id * id $ | F on id from the parsing table **F → id**, hence pop F, push id. |
| $E^1T^1id | id * id $ | The top symbol and the next input symbol both coincide hence pop id off from the stack, id from the input string. |
| $E^1T^1 | *id$ | $T^1$ on * from the table the production is **$T^1 \rightarrow *FT^1$** hence pop the top symbol and push the production contents in reverse order. |
| $E^1T^1F* | *id$ | The top symbol and the next input symbol both coincide hence pop * off from the stack, * from the input string. |
| $E^1T^1F | id$ | F on id from the parsing table **F → id**, hence pop F, push id. |
| $E^1T^1id | id$ | The top symbol and the next input symbol both coincide hence pop id off from the stack, id from the input string. |
| $E^1T^1 | $ | Now $T^1$ on $ from the table it is ε hence pop $T^1$ (we need not push ε as it is a null string) |
| $E^1 | $ | Now $E^1$ on $ from the table it is ε hence pop $E^1$ (we need not push ε as it is a null string). |
| $ | $ | The stack is empty and all the input symbols are parsed, hence the input symbol is parsed. |

**Ques 28) Parse the string babab using predictive parsing algorithm**
**E → FA**
**A → aE/ ε**
**F → b**

**Ans:**
First (E) = {b}
First (A) = { a, ε}

DOWNLOADED FROM WWW.KTUASSIST.IN

Scanned by CamScanner

First (F) = {b}
Follow (E) = {$}
Follow (A) = {$}
Follow (F) = {a, $}

**Predictive Parsing Table**

| | a | b | $ |
|---|---|---|---|
| E | | E → FA | |
| A | A → a E | | A → ε |
| F | | F → b | |

| Stack | Input | Explanation |
|---|---|---|
| $ E | babab$ | Stack initially contains the start symbol. Hence according to the algorithm, E on b **E → FA** |
| $ AF | babab$ | F on b, in the parsing table it is **F → b** then pop the symbol F and push b |
| $Ab | babab$ | Now, the top of the stack is a terminal, and the next input symbol is also b, both are matching then according to the algorithm, pop b from the stack remove b from the input symbol. |
| $A | abab$ | A on a is **A → a E** hence push in reverse order |
| $Ea | abab$ | Now, the top of the stack is a terminal, and the next input symbol is also a, both are matching then according to the algorithm, pop a from the stack remove a from the input symbol. |
| $E | bab$ | E on b is **E → FA** hence push in reverse order |
| $AF | bab$ | F on b, in the parsing table it is **F → b** then pop the symbol F and push b |
| $Ab | bab$ | Now, the top of the stack is a terminal, and the next input symbol is also b, both are matching then according to the algorithm, pop b from the stack remove b from the input symbol. |
| $A | ab$ | A on a is **A → aE** hence push in reverse order |
| $Ea | ab$ | Now, the top of the stack is a terminal, and the next input symbol is also a, both are matching then according to the algorithm, pop a from the stack remove a from the input symbol. |
| $E | b$ | E on b is **E → FA** hence push in reverse order |
| $AF | b$ | F on b, in the parsing table it is **F → b** then pop the symbol F and push b |
| $Ab | b$ | Now, the top of the stack is a terminal, and the next input symbol is also b, both are matching then according to the algorithm, pop b from the stack remove b from the input symbol. |
| $A | $ | A on $ it is ∈ since it is a null string f we need not push onto the stack; just pop A off the stack. |
| $ | $ | Hence the string has been parsed as the stack is empty and input symbols are all scanned. |

**Ques 29) Parse the string acbgfh for the following grammar using predictive parsing algorithm.**

**S→ aBDh**
**B → cC**
**C → bC/ ε**
**D → EF**
**E → g/ ε**
**F → f/ ε**

**Ans:**

FIRST (S) = {a}
FIRST (B) = {c}
FIRST (C) = {b, ε}
FIRST (D) = {g, f, ε}
FIRST (E) = {g, ε}
FIRST (F) = {f, ε}
FOLLOW (S) = {$}
FOLLOW (B) = {g, f, h}
FOLLOW (C) = {g, f, h}
FOLLOW (D) = {h}
FOLLOW (E) = {f, h}
FOLLOW (F) = {h}

DOWNLOADED FROM WWW.KTUASSIST.IN

Scanned by CamScanner

Terminals in the Column side

| Non-Terminals in the row side | a | b | c | g | f | h | $ |
|---|---|---|---|---|---|---|---|
| S | S→aBDh | | | | | | |
| B | | | B→ cC° | | | | |
| C | | C → bC | | C → ε | C → ε | | |
| D | | | | D → EF | D → EF | D → EF | |
| E | | | | E → g | E → ε | E → ε | |
| F | | | | | F → f | F → ε | |

| Stack | Input | Explanation |
|---|---|---|
| $S | acbgfh$ | Stack initially contains the start symbol. Hence according to the algorithm, S on a **S → aBDh** |
| $hDBa | acbgfh$ | Now, the top of the stack is a terminal, and the next input symbol is also b, both are matching, then according to the algorithm, pop a from the stack remove a from the input symbol. |
| $hDB | bgfh$ | B on c is **B → cC**, hence push in reverse order |
| $hDCc | cbgfh$ | Now, the top of the stack is at terminals, and the next input symbol is also b, both are matching then according to the algorithm, pop c from the stack remove c from the input symbol. |
| $hDC | bgfh$ | C on b is **C → bC** hence push in reverse order |
| $hDCb | gfh$ | Now, the top of the stack is a terminal, and the next input symbol is also b, both are matching then according to the algorithm, pop b from the stack remove b from the input symbol. |
| $hDC | gfh$ | C on g is ∈ hence we need not push it so just pop C off the stack. |
| $hD | gfh$ | D on g it is **D → EF** |
| $hFE | gfh$ | E on g is **E → g** |
| $h Fg | gfh$ | Now, the top of the stack is a terminal, and the next input symbol is also a, both are matching then according to the algorithm, pop g from the stack remove g from the input symbol. |
| $hF | fh$ | F on f is **F → f** hence push in reverse order. |
| $hf | fh$ | Now, the top of the stack is a terminal, and the next input symbol is also b, both are matching then according to the algorithm, pop f from the stack remove f from the input symbol. |
| $h | h$ | Now, the top of the stack is a terminal, and the next input symbol is also b, both are matching then according to the algorithm, pop h from the stack remove h from the input symbol. |
| $ | $ | Hence the string has been parsed as the stack is empty and input symbols are all scanned. |

**Ques 30) Construct predictive parsing table for the string (a^a)**
**The given grammar is**
**S → a**
**S→ ^**
**S → (T)**
**T→ S $T^1$**
**$T^1$ → ^S $T^1$/ ε**

**Ans:**
FIRST (S) = {a, ^, ( }
FIRST (T) = {a, ^, ( }
FIRST ($T^1$) = {^, ε}
FOLLOW (S) = { $, ) }
FOLLOW (T) = { ) }
FOLLOW ($T^1$) = { ) }

**Predictive Parsing Table**

| | a | ^ | ( | ) | $ |
|---|---|---|---|---|---|
| S | S→ a | S → ^ | S→ (T) | | |
| T | T → $ST^1$ | T → $ST^1$ | T → $ST^1$ | | |
| $T^1$ | | $T^1$ → ^$ST^1$ | | $T^1$ → ε | |

DOWNLOADED FROM WWW.KTUASSIST.IN

Scanned by CamScanner

**Ques 31) Explain recursive-descent parsing.**

**Ans: Recursive - Descent Parsing**

The process of repeated scans of the input string is called "back tracking". Usually a top down parser does not require backtracking. Consider a production of the form

$A \rightarrow a_1 / a_2 / \ldots\ldots\ldots a_n$

We have to choose a unique production that derives a string beginning with say b (the current input symbol). The right hand side of a grammar rule for A specifies the structure of the code for this procedure: the sequence of terminals and non-terminals in a choice correspond to matches of the input and calls to other procedures, with choices correspond to alternatives with in the code.

In recursive descent method of a parsing, a sequence of production applications is realized in a sequence of function calls. In particular, functions are written for each non-terminal. Each function returns a value of true or false depending on whether or not it recognizes a sub string, which is the expansion of that non-terminal.

All recursive techniques will not work on all context free grammars. A parser that uses a set of recursive procedures to recognize its input without any backtracking is called recursive descent parser. A recursive descent procedure that recognizes a non-terminal on the right hand side of a production can be written in pseudo code.

A parser that uses a set of recursive procedures to recognize its input with no backtracking is called "**Recursive Descent Parser**".

**For example,** consider the grammar

$E \rightarrow E + T / T$
$T \rightarrow T * F / F$
$F \rightarrow (E) / id$

After left recursion we obtain the list of productions as:

$E \rightarrow TE^1$
$E^1 \rightarrow + TE^1 / \varepsilon$
$T \rightarrow FT^1$
$T^1 \rightarrow *FT^1 / \varepsilon$
$F \rightarrow (E) / id$

The table representation of recursive descent parser is called as **predictive parser**.

**Ques 32) What is LL(1) Grammar? Explain the properties of LL(1) grammars.**

**Ans: LL(1) Grammar**

An LL (1) parser is a table driven parser for left-to-left parsing. The '1' in LL(1) indicates that the grammar uses a look-ahead of one source symbol- that is, the prediction to be made is determined by the next source symbol. A major advantage of LL (1) parsing is its amenability to automatic construction by a parser generator.

For certain grammars, the entries in the predictive parsing table may contain more than a single entry (multiple entries) that is, multiple productions in the parsing table for a terminal.

A Grammar whose parsing table has no multiply-defined entries is said to LL(1). That means, scanning the input from left to right for producing the leftmost derivations and also by using one input symbol of look ahead at each step to make the parsing action.

**Properties of LL(1) Grammars**

1) LL(1) grammars are not ambiguous.
2) LL (1) grammars are not left recursive.
3) If a grammar is LL (1) then a deterministic top down table drier recognizer can be constructed to recognize the L(G).
4) If $A \rightarrow \alpha / \beta$ are any two productions of any grammar G then the corresponding grammar is said to be LL (1) if FIRST $(\alpha) \cap$ FIRST $(\beta) = \phi$ where $\alpha \neq \beta$ and for every non terminal X such that first (X) contains $\varepsilon$ first $(\alpha) \cap$ follow $(X) = \phi$.

DOWNLOADED FROM WWW.KTUASSIST.IN

Scanned by CamScanner

**For example,** consider, e.g., the simplified grammar of if-statements as **shown below**:

statement → if-stmt | other
if-stmt → if (exp) statement else-part
else-part → else statement | ε
exp → 0 | 1

Constructing the LL(1) parsing table gives the result shown in **table below**:

**LL(1) Parsing Table for (Ambiguous) if-Statements**

| M[N, T] | if | other | else | 0 | 1 | $ |
|---|---|---|---|---|---|---|
| statement | statement → if-stmt | Statement → other | | | | |
| if-stmt | if-stmt → if( exp ) statement else-part | | | | | |
| else-part | | | else-part → else statement<br>else-part → ε | | | else-part → ε |
| exp | | | | exp → 0 | exp → 1 | |

In **table above**, the entry M[else-part, else] contains two entries, corresponding to the dangling else ambiguity. As in recursive-descent, when constructing this table, we could apply a disambiguating rule that would always prefer the rule that generates the current lookahead token over any other, and thus the production

else-part → else statement

would be preferred over the production else-part → ε. This corresponds in fact to most closely nested disambiguating rule. With this modification, **table above** becomes unambiguous, and the grammar can be parsed as if it were an LL(1) grammar. **For example, table below** shows the parsing actions of the LL(1) parsing algorithm, given the string

if(0) if(1) other else other

(For conciseness, we use the following abbreviations: statement = S, if-stmt = I, else-part = L, exp = E, if = i, else = e, other = o.)

**LL(1) Parsing Actions for if-statements using the Most Closely Nested Disambiguating Rule**

| Parsing Stack | Input | Action |
|---|---|---|
| $S | 1(0)i(1)oeo$ | S → I |
| $I | i(0)i(1)oeo$ | I → i(E)SL |
| $LS)E(i | i(0)i(1)oeo$ | match |
| $LS)E( | (0)i(1)oeo$ | match |
| $LS)E | 0)i(1)oeo$ | E → 0 |
| $LS)0 | 0)i(1)oeo$ | match |
| $LS) | )i(1)oeo$ | match |
| $LS | i(1)oeo$ | S → I |
| $LI | i(1)oeo$ | I → i(E)SL |
| $LLS)E(i | i(1)oeo$ | match |
| $LLS)E( | (1)oeo$ | match |
| $LLS)E | 1)oeo$ | E → 1 |
| $LLS)1 | 1)oeo$ | match |
| $LLS) | )oeo$ | match |
| $LLS | oeo$ | S → o |
| $LLo | oeo$ | match |
| $LL | eo$ | L → eS |
| $LSe | eo$ | match |
| $LS | o$ | S → o |
| $Lo | o$ | match |
| $L | $ | L → ε |
| $ | $ | accept |

DOWNLOADED FROM WWW.KTUASSIST.IN

Scanned by CamScanner

**Ques 33) Consider the Grammar:**
**$S \rightarrow iEtSS_1 / a$**
**$S_1 \rightarrow eS / \varepsilon$**
**$E \rightarrow b$**

**Show that the grammar is not LL(1).**

**Ans:**
FIRST (S) = { i, a}
FIRST ($S_1$) = { e, ε}
FIRST (E) = {b}
FOLLOW (S) = {e, $}
FOLLOW ($S_1$) = {e, $}
FOLLOW (E) = {t}

**Predictive parsing table**

| | i | t | a | e | b | $ |
|---|---|---|---|---|---|---|
| S | $S \rightarrow iEtSS_1$ | | $S \rightarrow a$ | | | |
| $S_1$ | | | | $S_1 \rightarrow eS$<br>$S_1 \rightarrow \varepsilon$ | | $S_1 \rightarrow \varepsilon$ |
| E | | | | | $E \rightarrow b$ | |

Since the grammar has got multiply defined entries (shaded area) the said grammar is said to be Non-LL(1).

**Ques 34) Consider the Grammar:**
**$S \rightarrow aSA / \varepsilon$**
**$A \rightarrow bB / cc$**
**$B \rightarrow bd / \varepsilon$**

**Show that the grammar is not LL(1).**

**Ans:**
FIRST (S) = { a, ε}
FIRST (A) = { b, cc}
FIRST (B) = {b, ε}
FOLLOW (S) = {$, b, cc}
FOLLOW (A) = {$, b, cc}
FOLLOW (B) = {$, b, cc}

**Predictive Parsing table**

| | a | b | d | cc | $ |
|---|---|---|---|---|---|
| S | $S \rightarrow aSA$ | $S \rightarrow \varepsilon$ | | $S \rightarrow \varepsilon$ | $S \rightarrow \varepsilon$ |
| A | | $A \rightarrow bB$ | | $A \rightarrow cc$ | |
| B | | $B \rightarrow bd$<br>$B \rightarrow \varepsilon$ | | $B \rightarrow \varepsilon$ | $B \rightarrow \varepsilon$ |

Since the parsing table has multiply defined entries it is not said to be LL(1).

**Ques 35) Discuss the techniques used to remove the drawbacks of top-down parsing.**

**Or**

**Write as note on**
1) **Elimination of left recursion**
2) **Left factoring**

**Ans: Techniques to Remove Drawbacks of Top-Down Parsing**

Two techniques, useful for rewriting grammars so they become suitable for top-down parsing are:

1) **Elimination of Left Recursion:** A grammar is left recursive if it has a nonterminal A such that there is a derivation $A \overset{+}{\Rightarrow} Aa$ for some string a .Top-down parsing methods cannot handle left-recursive grammars, so a transformation that eliminates left recursion is needed.

Consider the following production:
$A \rightarrow Aa / b$ where b does not begin with A

If any production is of the above form, we can eliminate left recursion by replacing the part of productions with:
$A \rightarrow bA^1$
$A^1 \rightarrow a A^1 / \varepsilon$

without changing the set of strings derivable from A.

DOWNLOADED FROM WWW.KTUASSIST.IN

Scanned by CamScanner

In the below transformation (**figure 2.15**), it is shown that the set of strings that are derivable from A are not changed.

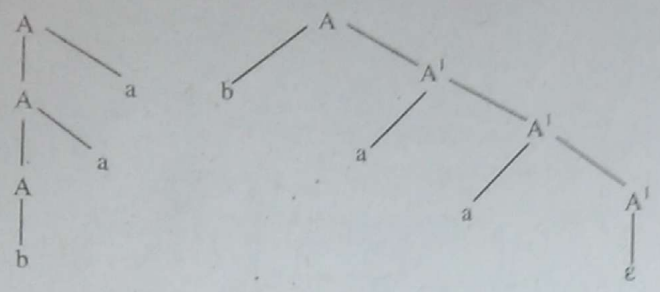



**Figure 2.15**

Now let us generalize the formula for the elimination of left recursion.

If the productions of the form
$A \rightarrow Aa_1 / Aa_2 / Aa_3 / \ldots\ldots\ldots\ldots / Aa_m / b_1 / b_2 / b_3 / \ldots\ldots\ldots\ldots /bi$

where no $b_i$ begin with an A, then replace the A - productions by
$A \rightarrow b_1A^1 / b_2A^1 / b_3A^1 / \ldots\ldots\ldots\ldots / b_nA^1$
$A \rightarrow a_1A^1 / a_2A^1 / a_3A^1 / \ldots\ldots\ldots\ldots / a_mA^1 / \varepsilon$

**Under the Condition that $a_i \rightarrow \varepsilon$**
The above process will remove immediate left recursion. But it does not eliminate left recursion involving derivations of two or more steps.

**Example**
$S \rightarrow A\alpha / \beta$
$A \rightarrow A\gamma / S\delta / e$

The nonterminal S is left recursive because
$S \rightarrow A\ \alpha$
$\underline{S} \rightarrow \underline{S}\ \delta\ \alpha$

But it is not immediately left recursive

**Algorithm to Eliminate Left Recursion from a Grammar**
Consider a grammar G:
Let G be in order $A_1, A_2, A_3 \ldots\ldots\ldots A_n$
**for** i: = 1 **to** n **do begin**
**for** j:= 1 **to** i – 1 **do begin**
replace each production of the form $A_i \rightarrow A_jb$
by the production $A_i \rightarrow a_1b / a_2b / \ldots\ldots\ldots a_ib$
where $A_j \rightarrow a_1 / a_2 / \ldots\ldots\ldots a_i$ are all the current $A_j$-productions;
**end**

eliminate the immediate left recursion among the $A_i$-productions
**end**

2) **Left Factoring:** Left parsing is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.

If there is a production of the form
$A \rightarrow \alpha\beta / \alpha\gamma$
then it can be written as
$A \rightarrow \alpha A^1$
$A^1 \rightarrow \beta / \gamma$

**Algorithm:** Left factoring a grammar.

**Input:** Grammar G.

**Output:** An equivalent left-factored grammar.

**Method:** For each nonterminal A find the longest prefix $\alpha$ common to two or more of its alternatives. If $\alpha \neq \in$, i.e., there is a nontrivial common prefix; replace all the A productions $A \rightarrow \alpha\beta_1 | \alpha\beta_2 \ldots. \alpha\beta_n | \gamma$ where $\gamma$ represents all alternatives that do not begin with $\alpha$ by
$A \rightarrow \alpha A' | \gamma$
$A' \rightarrow \beta_1 | \beta_2 \ldots. | \beta_n$

Here A' is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix.

**Example:** Consider the grammar
$S \rightarrow iCtS / iCtSeS/a$
$C \rightarrow b$
After left factoring the resultant grammar is
$S \rightarrow iC\ tSS^1 /a$
$S^1 \rightarrow eS / \varepsilon$
$C \rightarrow b$

**Example:**
$S \rightarrow Aab/ AabBb$

After Left factoring the above grammar is
$S \rightarrow AS'$
$S' \rightarrow ab / abBb / \varepsilon$

**Example:**
$E \rightarrow T + E /T$

After left factoring the grammar is
$E \rightarrow T\ E'$
$E' \rightarrow +E / \varepsilon$

**Ques 36) Consider the grammar**
$S \rightarrow A\ \alpha / \beta$ ..... (1)
$A \rightarrow A\ \gamma / S\delta / e$ ..... (2)

**Remove left recursion.**

**Ans:** Production 1 does not require left recursion, as the non-terminal A does not begin with S.
Now consider production 2.

It is written as:
$A \rightarrow A\ \gamma$
$A \rightarrow S\delta$
$A \rightarrow e$
$A \rightarrow S\delta$

DOWNLOADED FROM WWW.KTUASSIST.IN

Scanned by CamScanner

$\rightarrow \beta\delta$ (Using the production $S \rightarrow \beta$)

$A \rightarrow S\delta$
$\rightarrow A\,\alpha\,\delta$ (Using the production $S \rightarrow A\,\alpha$)

Hence,
$A \rightarrow A\,\gamma / A\,\alpha\,\delta / \beta\,\delta / e$

Now let us apply recursion formula for the above production as the Non Terminal A yields to another non-terminal A in the left side.
$AA \rightarrow A\,\gamma / A\,\alpha\,\delta / \beta\,\delta / e$

Using left recursion,
$A \rightarrow \beta\,\delta\,A^1 / e\,A^1$
$A^1 \rightarrow \gamma\,A^1 / \alpha\,\delta\,A^1 / \varepsilon$

**Ques 37) For the grammar**
**Exp → exp + term**
**Exp → exp – term**
**Exp → term**

**Remove left recursion.**

**Ans:**
$Exp \rightarrow term\ exp^1$
$Exp^1 \rightarrow +\ term\ exp^1 / -term\ exp^1 / \varepsilon$

**Ques 38) For the grammar**
**E → E + T/ T**
**T → T * F / F**
**F → (E) / id**

**Remove left recursion in the above-mentioned grammar.**

**Ans:** $E \rightarrow TE^1$
$E^1 \rightarrow +\ TE^1 / \varepsilon$
$T \rightarrow FT^1$
$T^1 \rightarrow *F\,T^1 / \varepsilon$

$F \rightarrow (E) / id$

**Ques 39) Consider the following grammar**
**A → Ba / Aa /c**
**B → Bb / Ab/ d**

**Remove the left recursion.**

**Ans:** First consider the first production.
$A \rightarrow BaA^1 / cA^1$
$A^1 \rightarrow aA^1 / \varepsilon$
$B \rightarrow Bb / BaA^1\,b / cA^1\,b/d$

Again we should remove the left recursion for the above grammar.
$A \rightarrow BaA^1 / c\,A^1$
$A^1 \rightarrow a\,A^1 / \varepsilon$
$B \rightarrow c\,A^1bB^1 / dB^1$
$B^1 \rightarrow b\,B^1 / aA^1bB^1 / \varepsilon$

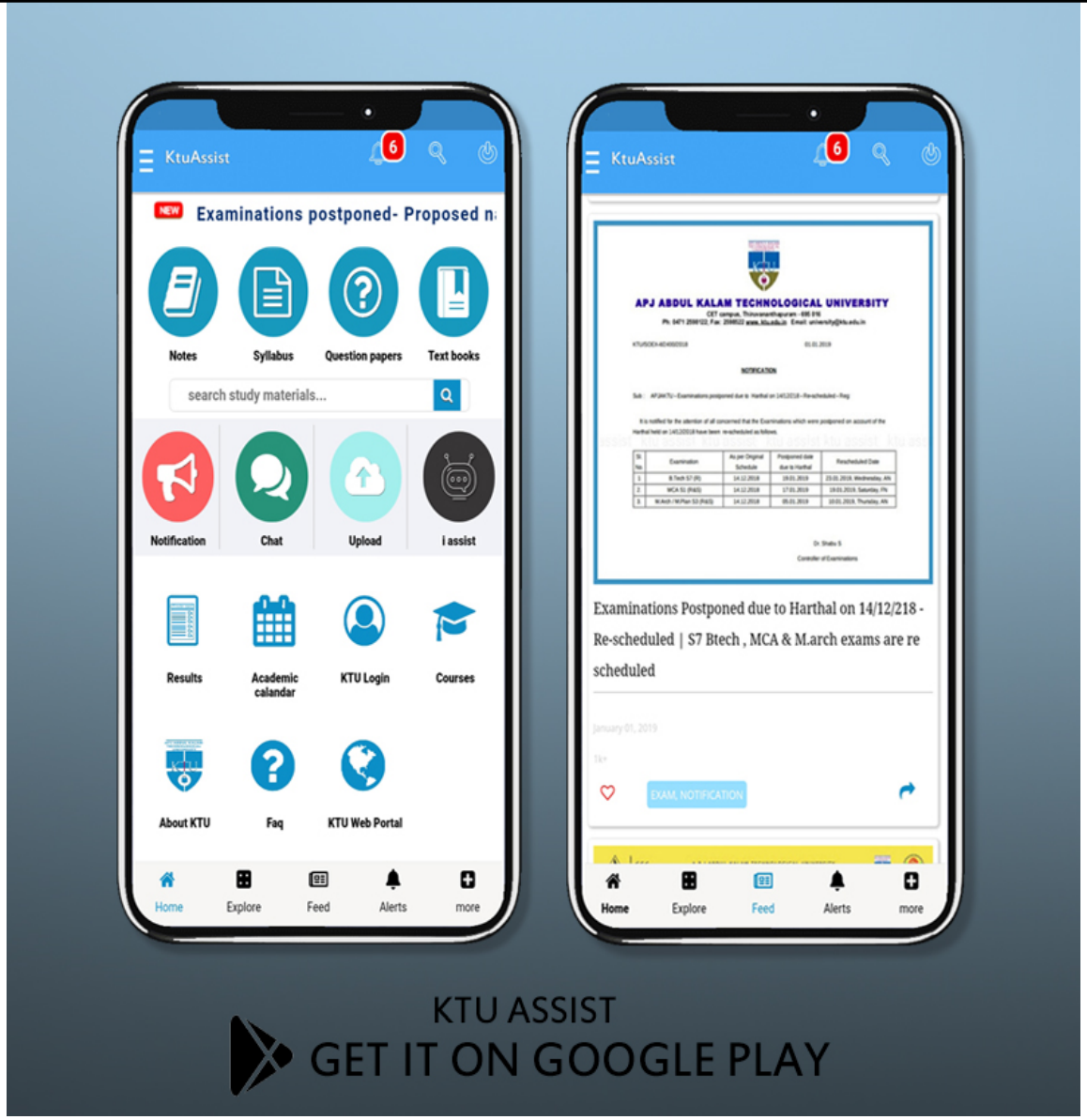
**Ques 40) Consider the grammar**
**S → a/ ^ / (T)**
**T → T, S / S**

**Remove left recursion for the above grammar.**

**Ans:**
S → a / ^ / (T) (no changes here as RHS does not start with S)
$T \rightarrow ST^1$
$T^1 \rightarrow {}^\wedge S\,T^1 / \varepsilon$

DOWNLOADED FROM WWW.KTUASSIST.IN

Scanned by CamScanner




# END

facebook.com/ktuassist
instagram.com/ktu_assist

DOWNLOADED FROM WWW.KTUASSIST.IN