CONTEXT FREE GRAMMAR (CFG)

Regular Languages (FSMs, Regular expressions. and Regular grammars) that offer less power and flexibility than a general purpose programming language provides. Because the frameworks were restrictive, we were able to describe a large class of useful operations that could be performed on the languages that we defined.

We will begin our discussion of the context-free languages with another restricted formalism, the **context-free grammar.**

Introduction to Re-write systems and grammars:

Re-write system is a rule based system in which there is a collection of rules and an algorithm for applying them.

Each rule has LHS and RHS

Example:

$\mathbf{S} \rightarrow aS$

$\mathbf{S} \rightarrow bA$

$\mathbf{A} \rightarrow \varepsilon$

Re-write system works on particular set of strings and try to match the LHS of the rule against some part of the string. But the core ideas that we will present can be used to define rewrite systems that operate on richer data structures (programming Languages).

Obtaining the string **w** using the rules in re-writes system is called **Derivation**.

Grammar

What is a grammar?

The re-write system which is used to define a language is called Grammar.

G is a grammar which generates a language L then the language is specified as L(G). Grammar work on set of symbols; can be of two types: **Non-terminal** symbols and **Terminal** symbols.

Non-terminal and Terminal Symbols

What is Non-terminal and Terminal Symbols?

Non-terminal symbols are kind of symbols that act as **working symbols** while the grammar is working on derivation. Non-terminal symbols disappear when the grammar completely derived the string **w** of L(G).

Terminal Symbols are from the input ∑. These symbols generate the string **w** of L(G).

Every grammar need one special symbol called start symbol. It is normally denoted by **S**.

Context-Free Grammars and Languages

We now define a context-free grammar (or CFG) to be a grammar in which each rule must:

- have a left-hand side that is a single **non terminal**, and
- Have any sequence (possibly empty) of symbols (non-terminal and/or terminals) on right-hand side.

For Example: $S \rightarrow aSb$

$S \rightarrow \varepsilon$

$T \rightarrow T$

$S \rightarrow aSbbTT$

The grammar is so called **context Free** because, using these rules, the decision to replace a non terminal by some other sequence is made without looking at the context in which the non terminal occurs. This rule says that **S** can be replaced by **aSb or ε or aSbbTT,** as required.

NOTE:

The rule **aSa → aTa** is not a context free grammar. This rule says that **S** can be replaced by **T** when it is surrounded by **a**'s. This type of grammar rule is called **context-sensitive** because its rules allow context to be considered.

Define Context Free Grammar (CFG).

A context free grammar G is a Quadruple (V, T, P, S) where

V – Set of Non-terminal symbols

T – Set of Terminal symbols

P – Set of production rules, where each production rule is in the form of

$$A \rightarrow \alpha$$

Where α is in $(V \cup T)^*$ and A is non-terminal and there is only one non- terminal on the left hand side of the production rule.

Example: $S \rightarrow AaB \mid Ba \mid abb$

$A \rightarrow bA \mid a$

$B \rightarrow b$

Define the term productions in CFG.

In CFG the rules which are applied to obtain a grammatically correct sentence are called productions.

- Each production starts with non-terminal, followed by an arrow, followed by combinations of non-terminals and/or terminals.

Example: $S \rightarrow AaB \mid Ba \mid abb \mid \varepsilon$

DESIGN OF CONTEXT FREE GRAMMAR

Context free grammar can be generated for different formal languages by constructing the basic building block grammar for the languages like:

$a^n \mid n \geq 0$.

$a^n b^n \mid n \geq 0$.

$a^{n+1} b^n \mid n \geq 0$.

$a^n b^{n+1} \mid n \geq 0$.

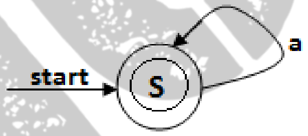
$a^{2n} b^n \mid n \geq 0$.

$a^n b^{2n} \mid n \geq 0$.

Write the CFG for the language $L = \{a^n \mid n \geq 0\}$.

This we can easily write, by constructing DFA and then converting into CFG or we can directly write the grammar.

The DFA for the given language is



The transition function is given $\delta(S, a) = S$.

This transition function can be converted into CFG as follows:

The first symbol (non-terminal) is identified as the state for which transition is defined.

ie: **S** and **,** is replaced by $\rightarrow$ and the input symbol '**a**' and the next state **S** are concatenated and written on RHS of CFG. **ie:** $S \rightarrow aS$.

For final state we have to include '**ε**' on RHS of CFG. ie: $S \rightarrow \varepsilon$.

Therefore the CFG for the language $L = \{a^n \mid n \geq 0\}$ is given by**:**

$$S \rightarrow aS.$$
$$S \rightarrow \varepsilon.$$

Note: The CFG for the language $L = \{a^n \mid n \geq 1\}$ is given by:

$S \rightarrow aS$

$S \rightarrow a$ **;** Where the minimum string is **a** when n =1

Similarly we can write the CFG for other languages as follows:

The CFG for the language $L = \{ a^n b^n \mid n \geq 0\}$ is given by:

$S \rightarrow aSb$

$S \rightarrow \varepsilon$

Here for every '**a**' one '**b**' has to be generated. This is obtained by suffixing '**aS'** with one '**b**'. The minimum string when n= 0 is **ε.**

If $L =\{ a^n b^n \mid n \geq 1\}$, then minimum string is '**ab'** instead of **ε** when n= 1, so the resulting grammar is: **$S \rightarrow aSb$**

$S \rightarrow ab$

The CFG for the language $L =\{ a^{n+1} b^n \mid n \geq 0\}$ is given by**:**

$S \rightarrow aSb$

$S \rightarrow a$

If $n \geq 1$ then CFG becomes:

$S \rightarrow aSb$

$S \rightarrow aab$

The CFG for the language $L =\{ a^n b^{n+1} \mid n \geq 0\}$ is given by**:**

$S \rightarrow aSb$

$S \rightarrow b$

If $n \geq 1$ then CFG becomes:

$S \rightarrow aSb$

$S \rightarrow abb$

The CFG for the language $L = \{ a^{2n} b^n \mid n \geq 0\}$

$S \rightarrow aaSb$

$S \rightarrow \varepsilon$

Here for every two '**a**'s one 'b' has to be generated. This is obtained by suffixing '**aaS'** with one '**b**'. The minimum string is **ε.**

If $L= \{a^{2n} b^n \mid n \geq 1\}$, then minimum string is '**aab'** instead of **ε,** when n= 1, so the resulting grammar is:

$S \rightarrow aaSb$

$S \rightarrow aab$

The CFG for the language L = { $a^n b^{2n}$ | n ≥ 0}

$S \rightarrow aSbb$

$S \rightarrow \varepsilon$

Here for every '**a**' two '**b**'s have been be generated. This is obtained by suffixing '**aS**' with '**bb**'. The minimum string is **ε.**

If n ≥ 1 then CFG becomes:

$S \rightarrow aSbb$

$S \rightarrow abb$

Design of context free grammar for the language represented using regular expression.

i. Obtain CFG for the language L = { $(a, b)^*$}

Language represents any number of a's and b's with ε.

$S \rightarrow aS \mid bS \mid \varepsilon$

ii. Obtain CFG for the language L = { (w ab w| where w € $(a+b)^*$ }

OR

Obtain CFG for the language containing strings of a's and b's with substring 'ab'

L = {w ab w} can be re-written as: $\underset{\mathbf{A}}{w}\ \mathbf{ab}\ \underset{\mathbf{A}}{w}$

Where A production represents any number of a's and b's and is given by:

A → aA | bA | ε

Therefore the resulting grammar is G = (V, T, P, S) where,

V = { S, A }, T = { a, b}, S is the start symbol, and P is the production rule is as shown below:

$S \rightarrow AabA$

$A \rightarrow aA \mid bA \mid \varepsilon$

iii. Obtain CFG for the language L = { $(011 + 1)^* 01$ }

L can be re-written as: $\underbrace{(011 + 1)^*}_{A}\ 01$

A production represents any number of 011's and 1's including ε.

ie: A → 011A | 1A | ε

Therefore the resulting grammar is G = (V, T, P, S) where,

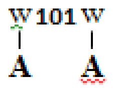
V = { S, A }, T = { 0, 1}, S is the start symbol, and P is the production rule is as shown below:

S → A01

A → 011A | 1A | ε

iv. Obtain CFG for the language L = { w| w € $(0,1)^*$ with at least one occurrence of '101' }.

The regular expression corresponding to the language is L = { w 101 w }



Where A production represents any number of 0's and 1's and is given by:

A → 0A | 1A | ε

Therefore the resulting grammar is G = (V, T, P, S) where,

V = { S, A }, T = { 0, 1}, S is the start symbol, and P is the production rule is as shown below:

S → A101A

A → 0A | 1A | ε

v. Obtain CFG for the language L = { w| wab € $(a,b)^*$ }.

OR

Obtain CFG for the language containing strings of a's and b's ending with'ab'. }.

The resulting grammar is G = (V, T, P, S) where,

V = { S, A }, T = { a, b}, S is the start symbol, and P is the production rule is as shown below:

S → Aab

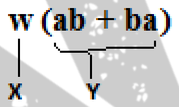
A → aA | bA | ε

vi. Obtain CFG for the language containing strings of a's and b's ending with'ab' or 'ba'. }.

OR

Obtain the context free grammar for the language L = { XY | X € $(a, b)^*$ and Y € (ab or ba)

The regular expression corresponding to the language is **w (ab + ba)** where w is in(a, b)*



X→ aX | bX | ε

Y → ab | ba

The resulting grammar is G = (V, T, P, S) where,

V = { S, X, Y }, T = { a, b}, S is the start symbol, and P is the production rule is as shown below:

S → XY

X→ aX | bX | ε

Y → ab | ba

Obtain the CFG for the language L = { N_a (w) = N_b (w) | w € $(a, b)^*$ }

OR

Obtain the CFG for the language containing strings of a's and b's with equal number of a's and b's.

Answer:

To get equal number of a's and b's, we know that there are 3 cases:

i. An empty string **ε** has equal number of a's and b's
ii. Equal number of a's followed by equal number of b's.
iii. Equal number of b's followed by equal number of a's.

The corresponding productions for these 3 cases can be written as

S→ ε

S→ aSb

S→ bSa

Using these productions the strings of the form ε, ab, ba, ababab….., bababa…. etc can be generated.

But the strings such as abba, baab, etc, where the strings starts and ends with the same symbol, cannot be generated from these productions. So to generate these type of strings, we need to concatenate the above two productions which generates **equal a's and equal b's** and **equal b's and equal a's** or vice versa. The corresponding production is S→ SS.

The resulting grammar corresponding to the language with equal number of a's and equal number of b's is G = (V, T, P, S) where,

V = { S }, T = { a, b}, S is the start symbol, and P is the production rule is as shown below:

S → ε

S→ aSb

S → bSa

S → SS

Obtain the CFG for the language L = { N_a (w) = N_b (w) + 1 | w € $(a, b)^*$ }

The language containing stings of **a**'s and **b**'s with number of **a**'s one more than number of '**b**'s.

Here we should have one more a's either in the beginning or at the end or at the middle.

We can write the **A** production with equal number of a's and equal number of b's as

A→ ε | aAb | bAa |AA

and finally inserting one extra '**a**' between these **A** production. ie:

S→ AaA

The resulting grammar corresponding to the language $N_a(w) = N_b(w) +1$ is G = (V, T, P, S) where, V = { S, A }, T = { a, b}, S is the start symbol, and P is the production rule is as shown below:

$S \rightarrow AaA$

$A \rightarrow aAb \mid bAa \mid AA \mid \varepsilon$

******* c) Obtain the CFG for the language L = { set of all palindromes over {a, b} }

i. ε is a palindrome; **ie: $S \rightarrow \varepsilon$**

ii. a is a palindrome; **ie: $S \rightarrow a$** (odd length string**)**

iii. b is a palindrome ; **ie: $S \rightarrow b$ (** odd length string**)**

iv. If **w** is a palindrome then the string **a**w**a** and the string **b**w**b** are palindromes.

ie: If **S** is palindrome then a**S**a and b**S**b is also palindrome. The corresponding production is

$S \rightarrow aSa \mid bSb$

The resulting grammar corresponding to the language L = { set of all palindromes} is G = (V, T, P, S) where,

V = { S }, T = { a, b}, S is the start symbol, and P is the production rule is as shown below:

$S \rightarrow aSa \mid bSb \mid a \mid b \mid \varepsilon$

Obtain the CFG for the language L = {set of all non-palindromes over {a, b}}

Non-palindrome strings are not having same symbol at the start and ending point.

ie: $A \rightarrow aBb \mid bBa$

Where **B** corresponds to any number of a's and b's; ie: $B \rightarrow aB \mid bB \mid \varepsilon$

Finally non-palindrome strings are generated by inserting **A** production between a palindrome production S; ie **$S \rightarrow aSa \mid bSb \mid A$**

The resulting grammar corresponding to the language L = { set of all non-palindromes} is G = (V, T, P, S) where,

V = { S, A, B }, T = { a, b}, S is the start symbol, and P is the production rule is as shown below:

$S \rightarrow aSa \mid bSb \mid A$

$A \rightarrow aBb \mid bBa$

$B \rightarrow aB \mid bB \mid \varepsilon$

Obtain the CFG for the language L = { ww^R | w€ $(a, b)^*$}

NOTE: ww^R generates palindrome strings of a's and b's of **even length.**

That means we can remove the odd length palindrome strings such as '**a**' and '**b**' from the above palindrome problem

The resulting grammar corresponding to the language L = { ww^R} is G = (V, T, P, S) where,

V = { S }, T = { a, b}, S is the start symbol, and P is the production rule is as shown below:

S → aSa | bSb | ε

Obtain the CFG for the language L = { w = w^R | w is in $(a, b)^*$}

OR

L = {palindrome strings over {a, b}

Note: w = w^R indicates that string w and its reversal w^R is always equal; That means the strings generated from the language is palindrome strings. (either even or odd length palindrome).

Answer is same as that of palindrome problem (c)

The resulting grammar corresponding to the language L = { w= w^R } is G = (V, T, P, S) where,

V = { S }, T = { a, b}, S is the start symbol, and P is the production rule is as shown below:

S → aSa | bSb | a | b | ε

Obtain the CFG for the language containing all positive odd integers up to 999.

The resulting grammar corresponding to the language L = {all positive odd integers up to 999 } is G = (V, T, P, S) where,

V = { S, C, D }, T = { 0,1, 2,3,4,5,6,7,8,9}, S is the start symbol, and P is the production rule is as shown below:

S → C | DC | DDC

C → 1|3|5|7|9

D → 0| 1|2|3|4|5|6|7|8|9

Obtain the context free grammar for the language L = {$0^m 1^m 2^n$ | m, n ≥ 1 }

Answer:

$$\text{Let } L = \underbrace{0^m\,1^m}_{A}\ \underset{B}{2^n}$$

We know that CFG corresponding to the language **$0^m 1^m$ | m ≥ 1,** by referring the basic building block grammar of $a^n b^n$ | n **≥ 1.**

The equivalent **A** production is:

A → 0A1

A → 01

Here **B** represents any number of 2's with at least one 2 (**$n \geq 1$), w**hich is similar to a^n grammar.

The equivalent B production is:

$B \rightarrow 2B$

$B \rightarrow 2$

So the context free grammar for the language **$L = \{ 0^m 1^m 2^n \mid m, n \geq 1$** } is G = (V, T, P, S) where,

V = { S, A, B}, T = { 0, 1, 2}, S is the start symbol, and P is the production rule is as shown below:

$S \rightarrow AB$

$A \rightarrow 0A1 \mid 01$

$B \rightarrow 2B \mid 2$

Obtain the context free grammar for the language $L = \{a^{2n} b^m \mid m, n \geq 0 \}$

Answer:

Since '**a**' represented in terms of '**n**' and '**b**' represented in terms of '**m**', we can re-write the language as:

$$\text{Let } L = \underset{\mathbf{A}}{\mathbf{a^{2n}}}\ \underset{\mathbf{B}}{\mathbf{b^{m}}}$$

Here **A** represents **2n** number of **a**'s, and its equivalent production is $A \rightarrow aaA \mid \varepsilon$

and **B** represents **m** number of **b**'s, and its equivalent production is $B \rightarrow bB \mid \varepsilon$

So the context free grammar for the language **$L = \{a^{2n} b^m \mid m, n \geq 0$** } is G = (V, T, P, S) where,

V = { S, A, B}, T = { a, b }, S is the start symbol, and P is the production rule is as shown below:

$S \rightarrow AB$

$A \rightarrow aaA \mid \varepsilon$

$B \rightarrow bB \mid \varepsilon$

Obtain the context free grammar for the language $L = \{0^i 1^j 2^k \mid i = j \text{ or } j = k \text{ where } i, j, k \geq 0 \}$

Case 1: when i = j

The given language becomes $L = \underbrace{\mathbf{0^i 1^i}}_{\mathbf{A}}\ \underset{\mathbf{B}}{\mathbf{2^k}}$

The resultant production is given by: $A \rightarrow 0A1 \mid \boldsymbol{\varepsilon}$ and $B \rightarrow 2B \mid \varepsilon$

Therefore case 1 results in productions

$S \rightarrow AB$

$A \rightarrow 0A1 \mid \varepsilon$

$B \rightarrow 2B \mid \varepsilon$

Case 2: when j = k

$$L = \underbrace{0^i}_{C}\ \underbrace{1^j 2^j}_{D}$$

Therefore case 2 results in productions

S→ CD

C→ 0C | ε

D→ 1D2| ε

So the context free grammar for the language **$L = \{ 0^i 1^j 2^k \mid i = j$ or $j = k$ where $i, j, k \geq 0 \}$**

is G = (V, T, P, S) where,

V = {S, A, B, C, D}, T = {0, 1, 2}, S is the start symbol, and P is the production rule is as shown below:

S→ AB| CD

A→ 0A1| ε

B→ 2B| ε

C→ 0C | ε

D→ 1D2| ε

Obtain the context free grammar for the language $L = \{ 0^i 1^j \mid i \neq j$ where $i, j \geq 0 \}$

Answer:

Here we should not have equal number of 0's and 1's. ie: $i \neq j$.

Therefore we have two possible cases:

Case 1: when i > j. ie: Number of 0's greater than number of 1's. That means at least one 1 followed by equal number of 0's and 1's. Therefore the corresponding language is

$L = \{ 0^i 1^j 1^+ \mid i, j \geq 0 \}$

$$= \underbrace{\mathbf{0^+}}_{\mathbf{A}}\ \underbrace{0^i 1^j}_{\mathbf{B}}$$

Where A→ 0A| 0 B→ 0B1| ε

Case 2: when i < j. ie: Number of 0's less than number of 1's. That means at least one 0 preceded by equal number of 0's and 1's. Therefore the corresponding language is

$L = \{ \mathbf{0^+}\ 0^i 1^j \mid i, j \geq 0 \}$

$$= \quad \underbrace{0^i 1^j}_{B} \quad \underset{C}{\underset{|}{1^+}}$$

Where C→ 1C| 1

So the context free grammar for the language **L = { 0^i 1^j | i ≠ j where i, j ≥ 0 }**

is G = (V, T, P, S) where,

V = {S, A, B, C}, T = {0, 1}, S is the start symbol, and P is the production rule is as shown below:

S→ AB |BC

A→ 0A| 0

B→ 0B1| ε

C→ 1C| 1

Obtain the context free grammar for the language L = {a^n b^m | n = 2m where m ≥ 0 }

Answer:

By substituting n = 2m we have

L= { a^{2m} b^m | m ≥ 0 }

Here for every two 'a's one 'b' has to be generated. This is obtained by suffixing 'aaS' with one 'b'. The minimum string is ε.

So the context free grammar for the language **L = {a^n b^m | n = 2m where m ≥ 0** }

is G = (V, T, P, S) where,

V = {S }, T = {a, b}, S is the start symbol, and P is the production rule is as shown below:

S → aaSb

S → ε

Obtain the context free grammar for the language L = {a^n b^m | n ≠ 2m where n, m ≥ 1 }

Answer:

Here **n ≠ 2m** means n > 2m **or** n< 2m, which results in two possible cases of Language L.

Case 1: when n > 2m, we can re-write the language **L** by taking n = 2m + 1

L= { a^{2m+1} b^m | m ≥ 1}; by referring the basic building block grammar example, the resulting production (**a^{2m} b^m**) is given by:

A → **aaAb**

The minimum string when m = 1 is **'aaab'.**

ie : A → **a**

Therefore A $\rightarrow$ **aaAb | aaab**

Case 2: when n < 2m, we can re-write the language **L** by taking n = 2m - 1

L= { a^{2m-1} b^m | m ≥ 1 }; by referring the basic building block grammar example, the resulting production ($\mathbf{a^{2m}\ b^m}$) is given by:

B $\rightarrow$ **aaBb**

The minimum string when m = 1 is **'ab'.**

ie : B$\rightarrow$ **ab**

Therefore ; B $\rightarrow$ **aaBb | ab.**

So the context free grammar for the language **L = {a^n b^m | n ≠ 2m where n, m ≥ 1 }**

is G = (V, T, P, S) where,

V = {S, A, B }, T = {a, b}, S is the start symbol, and P is the production rule is as shown below:

S $\rightarrow$A | B

A $\rightarrow$ aaAb | aaab

B $\rightarrow$ aaBb | ab.

Obtain the context free grammar for the language L = { 0^i 1^j 2^k | i + j = k where i, j ≥ 0 }

Answer:

When i+ j =k, the given language becomes: L = $\mathbf{0^i\ 1^j\ 2^{i+j}}$

L = $\mathbf{0^i\ 1^j 2^i\ 2^j}$ = $\mathbf{0^i\ \underbrace{1^j\ 2^j}_{A}\ 2^i}$ **; minimum value when i=0 is A**

Note: For this type of language we have to select the **middle string** as a substring (A) and we need to insert this substring between the **start** production **ie**: $\mathbf{0^i\ 2^i}$ **(**where middle term A is ignored**)**

The equivalent substring **A** production is given by: A$\rightarrow$ 1A2| ε

The start production S$\rightarrow$ 0S2| A ; here the minimum value when i = 0 is A

So the context free grammar for the language **L = { 0^i 1^j 2^k | i + j = k where i, j ≥ 0 }**

is G = (V, T, P, S) where,

V = {S, A}, T = {0, 1, 2}, S is the start symbol, and P is the production rule is as shown below:

S$\rightarrow$ 0S2| A

A$\rightarrow$ 1A2| ε

Obtain the context free grammar for the language L = { a^n b^m c^k | n+ 2m = k where n, m ≥ 0 }

When n+2m = k, the given language becomes: L = a^n b^m c^{n+2m} = $\mathbf{a^n\ \underbrace{b^m\ c^{2m}}_{A}\ c^n}$

Minimum value when n = 0 is A

So the context free grammar for the language **L = { a^n b^m c^k | n + 2m = k where n, m ≥ 0 }** is G = (V, T, P, S) where,

V = {S, A}, T = {a, b, c}, S is the start symbol, and P is the production rule is as shown below:

$S \rightarrow aSc \mid A$

$A \rightarrow bAcc \mid \varepsilon$

Obtain the context free grammar for the language L = { $w\, a^n b^n w^R$ | W is in $(0, 1)^*$ and n ≥ 0 }

Answer: we can re-write the language L as $W \underbrace{a^n\, b^n}_{A} W^R$

The corresponding A production is given by; $A \rightarrow aAb \mid \boldsymbol{\varepsilon}$ **;** min. value is **ε** when n = 0

We can insert this substring A production between ww^R production represented by S.

The corresponding S production is $S \rightarrow$ **0S0 | 1S1 |A**

Note: In **S** production minimum value is **A**, when ww^R results in ε; **ie**: only the middle substring A appears.

So the context free grammar for the language **L = { $w\, a^n b^n w^R$| w is in $(0, 1)^*$ and n $\geq$ 0 }**

is G = (V, T, P, S) where,

V = {S, A}, T = {a, b, 0, 1}, S is the start symbol, and P is the production rule is as shown below:

$S \rightarrow 0S0 \mid 1S1 \mid A$

$A \rightarrow aAb \mid \varepsilon$

Obtain the context free grammar for the language L = { $a^n ww^R b^n$ | w is in (0, 1)* and n ≥ 2 }

L = $a^n \underbrace{WW^R}_{A} b^n$

Corresponding A production is $A \rightarrow 0A0 \mid 1A1 \mid \boldsymbol{\varepsilon}$

By inserting this substring **A** production between start production S; ie for $a^n b^n \mid n \geq 2$

S production is; $S \rightarrow aSb \mid aaAbb$; minimum string when n =2 is aaAbb

So the context free grammar for the language **L = { $a^n ww^R b^n$ | W is in (0, 1)* and n $\geq$ 2 }**

is G = (V, T, P, S) where,

V = {S, A}, T = {a, b, 0, 1}, S is the start symbol, and P is the production rule is as shown below:

$S \rightarrow aSb \mid aaAbb$

$A \rightarrow 0A0 \mid 1A1 \mid \varepsilon$

Obtain the context free grammar for the language $L = \{ a^n b^n c^i \mid n \geq 0, i \geq 1 \;\cup\; a^n b^n c^m d^m \mid n, m \geq 0 \}$

Answer:

$$L = \underbrace{a^n b^n c^i}_{S_1} \; U \; \underbrace{a^n b^n c^m d^m}_{S_2} \}$$

S_1 production is ; $S_1 \rightarrow AB$

$A \rightarrow aAb \mid \varepsilon$

$B \rightarrow cB \mid c$

S_2 production is ; $S_2 \rightarrow AC$

$C \rightarrow cCd \mid \varepsilon$

So the context free grammar for the language **$L = \{a^n b^n c^i \mid n \geq 0, i \geq 1 \;\; U \; a^n b^n c^m d^m$** $\mid n, m \geq$ **$0 \}$** is G = (V, T, P, S) where,

V = {S, S_1, S_2 A, B, C}, T = {a, b, c, d}, S is the start symbol, and P is the production rule is as shown below:

$S \rightarrow S_1 \mid S_2$

$S_1 \rightarrow AB$

$A \rightarrow aAb \mid \varepsilon$

$B \rightarrow cB \mid c$

$S_2 \rightarrow AC$

$C \rightarrow cCd \mid \varepsilon$

Obtain the context free grammar for the language L_1L_2 where $L_1 = \{ a^n b^n c^i \mid n \geq 0, i \geq 1 \}$ and $L_2 = \{ 0^n 1^{2n} \mid n \geq 0 \}$

Answer:

$$L = \underbrace{a^n b^n c^i}_{S_1} \quad \underbrace{0^n \;\; 1^{2n}}_{S_2}$$

S_1 production is ; $S_1 \rightarrow AB$

$A \rightarrow aAb \mid \varepsilon$

$B \rightarrow cB \mid c$

S_2 production is: $S_2 \rightarrow 0\, S_2 11 \mid \varepsilon$

So the context free grammar for the language **$L_1 = \{ a^n b^n c^i \mid n \geq 0, i \geq 1 \}$ and $L_2 = \{ 0^n 1^{2n} \mid n \geq 0 \}$**

is G = (V, T, P, S) where,

V = {S, S_1, S_2, A, B}, T = {a, b, c, 0, 1}, S is the start symbol, and P is the production rule is as shown below:

$S \rightarrow S_1 S_2$

$S_1 \rightarrow AB$

$A \rightarrow aAb \mid \varepsilon$

$B \rightarrow cB \mid c$

$S_2 \rightarrow 0S_2 11 \mid \varepsilon$

********* Obtain the context free grammar for the language $L = \{ a^{n+2} b^m \mid n \geq 0, m > n \}$**

It is clear from the above language that set of strings that can be generated is represented as:

n = 0				n = 1				n = 2			…….
m = 1	m = 2	m = 3	…	m = 2	m = 3	m = 4	.	m = 3	m = 4	m = 5	……….
aab	aabb	aabbb	…	aaabb	aaabb	aaabb	.	aaaabbb	aaaabb	aaaabbb	
					b	b			bb	b	
$a\ ab\ b^*$				$a\ aabb\ b^*$				$a\ aaabbb\ b^*$			……….

$$\mathbf{a\ a^n b^n\ b^*}$$

We observe that above language consists of strings of **a**'s and **b**'s which starts with one **a** followed by $a^n b^n$ $n \geq 1$, which in-term followed by any number of **b**'s (b^*) .

Therefore the given language L can be re-written as **$L = \{ a\, a^n b^n\, b^* \mid n \geq 1\}$**

$$L = a\ \underbrace{a^n b^n}_{A}\ \underbrace{b^*}_{B}$$

$A \rightarrow aAb \mid ab$

$B \rightarrow bB \mid \varepsilon$; and S production is $S \rightarrow aAB$

So the context free grammar for the language **$L = \{ a^{n+2} b^m \mid n \geq 0, m > n \}$** is G = (V, T, P, S) where,

V = {S, A, B}, T = {a, b}, S is the start symbol, and P is the production rule is as shown below: **$S \rightarrow aAB$**

$A \rightarrow aAb \mid ab$

$B \rightarrow bB \mid \varepsilon$

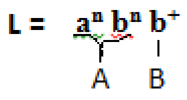
********* Obtain the context free grammar for the language $L = \{ a^n b^m \mid n \geq 0, m > n \}$**

n = 0	n = 1	n = 2	…….
m = 1 m = 2 m = 3 …	m = 2 m = 3 m = 4 .	m = 3 m = 4 m = 5	……….
ε b ε bb ε bbb …	abb abbb abbbb .	aabbb aabbbb aabbbbb	
$\varepsilon\ b^+$	$ab\ b^+$	$aabb\ b^+$	……….

$a^n b^n b^+$ **where** $n \geq 1$

We observe that above language consists of strings of **a**'s and **b**'s with n number of **a**'s followed by n number of **b**'s, which in term followed by any number of **b**'s with at least one **b**

$L = \{ a^n b^n b^+ \mid n \geq 0 \}$



A production is; $A \rightarrow aAb \mid \varepsilon$

B production is; $B \rightarrow bB \mid b$

So the context free grammar for the language $L = \{ a^n b^m \mid n \geq 0, m > n \}$ is G = (V, T, P, S) where,

V = {S, A, B}, T = {a, b}, S is the start symbol, and P is the production rule is as shown below:

$S \rightarrow AB$

$A \rightarrow aAb \mid \varepsilon$

$B \rightarrow bB \mid b$

********* Obtain the context free grammar for the language $L = \{ a^n b^{n-3} \mid n \geq 3 \}$**

Answer:

L = { aaaε, aaaab, aaaaabb, aaaaaabbb,…………………………………….. }

So we can re-write the language as;

$L = aaa\ a^n b^n \mid n \geq 0$

So the context free grammar for the language $L = \{ a^n b^{n-3} \mid n \geq 3 \}$ is G = (V, T, P, S) where,

V = {S, A}, T = {a, b}, S is the start symbol, and P is the production rule is as shown below:

$S \rightarrow aaaA$

$A \rightarrow aAb \mid \varepsilon$

MODULO – K PROBLEMS: WRITING CFG BY CONSTRUCTING DFA:

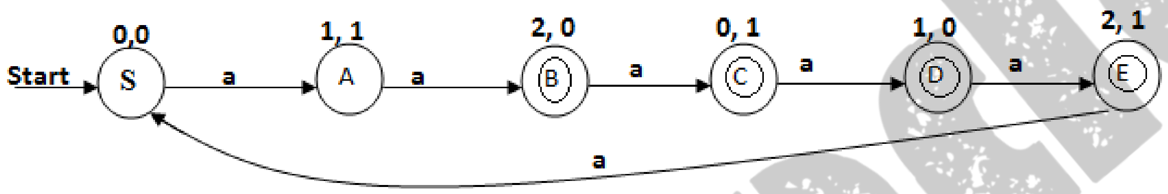
********* Obtain the context free grammar for the language L = { w € (a)* | |w| mod 3 ≠ |w| mod 2 }**

Answer:

Here mod 3 results in 3 remainders such as; **0, 1 and 2** and mod 2 results in 2 remainders such as **0 and 1**:

The possible states are: (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)

The equivalent DFA:



The productions are:

S → aA

A→ aB

B→ aC | ε

C→ aD | ε

D→ aE| ε

E→ aS |ε

So the context free grammar for the language **L = { w € (a)* | |w| mod 3 ≠ |w| mod 2 }**

is G = (V, T, P, S) where,

V = {S, A, B, C, D, E}, T = {a, b}, S is the start symbol, and P is the production rule is as shown below:

S → aA

A→ aB

B→ aC | ε

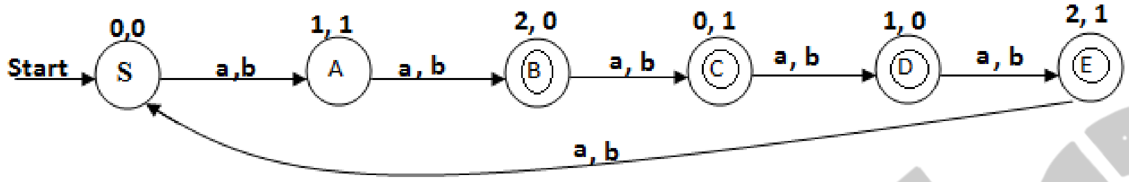
C→ aD | ε

D→ aE| ε

E→ aS |ε

******* Obtain the context free grammar for the language L = { w € (a, b)* | |w| mod 3 ≠ |w| mod 2 }

DFA:



The productions are:

S → aA | bA

A→ aB |bB

B→ aC | bC| ε

C→ aD |bD | ε

D→ aE| bE |ε

E→ aS |bS| ε

So the context free grammar for the language **L = {w € (a, b)* | |w| mod 3 ≠ |w| mod 2 }**

is G = (V, T, P, S) where,

V = {S, A, B, C, D, E}, T = {a, b}, S is the start symbol, and P is the production rule is as shown below:

S → aA | bA

A→ aB |bB

B→ aC | bC| ε

C→ aD |bD | ε

D→ aE| bE |ε

E→ aS |bS| ε

******* **Obtain the context free grammar for the language L = {w: N$_a$(w) mod 2 = 0 where w € (a, b)* }**

Answer:

N$_a$(w) mod 2 = 0 means; the string contains even number of **a**'s and any number of **b's**.

The Language can be re-written as: $b^n a^{2m} b^n$ | n ≥ 0, m ≥ 0

The S production is given by: S → ABA where **A** represents b^n | n ≥ 0 and **B** represents a^{2m} | m ≥ 0.

A production is given by: A → bA | ε

B production is given by: B$\rightarrow$ aaB | ε

So the context free grammar for the language **L = {w: $N_a(w)$ mod 2 = 0 where w € (a, b)* }**

is G = (V, T, P, S) where,

V = {S, A, B }, T = {a, b}, S is the start symbol, and P is the production rule is as shown below:

S → ABA

A → bA | ε

B→ aaB | ε

Write a CFG for the language **L** defines balanced parentheses.

OR

L = { { (,) }*| parentheses are balanced }

So the context free grammar G = (V, T, P, S) where,

V = {S}, T = {(,)}, S is the start symbol, and P

S → (S)

S → SS

S→ ε

DERIVATION

Define the following terms:

i. Derivation

ii. Left Most Derivation

iii. Right Most Derivation.

iv. Sentential Form

v. Left Sentential Form

Derivation: The process of obtaining string of terminals and/or non-terminals from the start symbol by applying some or all production rules is called derivation.

If a string is obtained by applying only one production, then it is called one step derivation.

Example: Consider the Productions: S →AB, A→ aAb|ε, B →bB|ε

S => AB

⇨ aAbB

⇨ abB

⇨ abbB

⇨ abb

Note: The derivation process may end whenever one of the following things happens.

i. The working string no longer contains any non terminal symbols (including, as a special case when the working string is **ε**). Ie: working string is generated.

ii. There are non terminal symbols in the working string but there is no match with the left-hand side of any rule in the grammar. For example, if the working string were **AaBb**, this would happen if the only left-hand side were **C**

Left Most Derivation (LMD): In derivation process, if a leftmost variable is replaced at every step, then the derivation is said to be leftmost.

Example: E → E+E | E*E | a | b

Let us derive a string **a+b*a** by applying LMD.

E => **E***E

⇨ **E**+E*E

⇨ a +**E***E

⇨ a+b***E**

⇨ a+b*a

Right Most Derivation (RMD): In the derivation process, if a rightmost variable is replaced at every step, then the derivation is said to be rightmost.

Example: E → E+E | E*E | a | b

Let us derive a string **a+b*a** by applying RMD.

E => E+**E**

⇨ E+E***E**

⇨ E +**E***a

⇨ **E**+b*a

⇨ a+b*a

Sentential form: For a context free grammar G, any string '**w**' in $(V \cup T)^*$ which appears in every derivation step is called a sentence or sentential form.

Two ways we can generate sentence:

i. Left sentential form

ii. Right sentential form

Example: S => AB

⇨ aAbB

⇨ abB

⇨ abbB

⇨ abb

Here {S, AB, aAbB, abB, abbB, abb } can be obtained from start symbol S, Each string in the set is called sentential form.

Left Sentential form**:** For a context free grammar G, any string '**w**' in $(V \cup T)^*$ which appears in every Left Most Derivation step is called a Left sentential form.

Example: E => **E***E

⇨ **E**+E*E

⇨ a +**E***E

⇨ a+b***E**

⇨ a+b*a

Left sentential form = {E, **E***E, **E**+E*E, a +**E***E, a+b***E,** a+b*a }

Right Sentential form**:** For a context free grammar G, any string '**w**' in $(V \cup T)^*$ which appears in every Right Most Derivation step is called a Left sentential form.

Example: E => E+**E**

⇨ E+E***E**

⇨ E +**E***a

⇨ **E** + b*a

⇨ a + b*a

Right sentential form = {E, **E**+E, **E**+E*E, E +**E***a, E+ b*a**,** a + b * a }

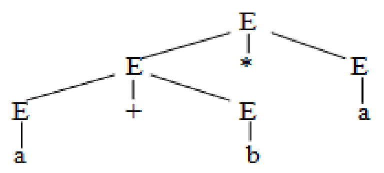
PARSE TREE: (DERIVATION TREE)

What is parse tree?

The derivation process can be shown in the form of a tree. Such trees are called derivation trees or Parse trees.

Example: E → E+E | E*E | a | b

The Parse tree for the LMD of the string **a+b*a** is as shown below:

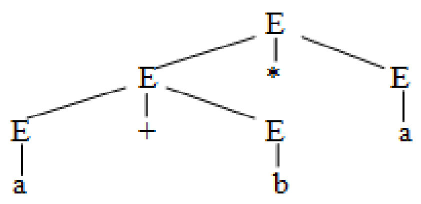


YIELD OF A TREE

What is Yield of a tree?

The yield of a tree is the string of terminal symbols obtained by only reading the leaves of the tree from left to right without considering the **ε** symbols.

Example:



For the above parse tree, the yield of a tree is **a + b * a**

Branching factor

Define the branching factor of a CFG.

The branching factor of a grammar G is the length (the number of symbols) of the longest right-hand side of any rule in G.

Then the branching factor of any parse tree generated by G is less than or equal to the branching factor of G.

NOTE:

1. Every leaf node is labelled with terminal symbols including ε.
2. The root node is labelled with start symbol S
3. Every interior- node is labelled with some element of V.

Problem 1:

Consider the following grammar G:

S → aAS |a

A→ SbA |SS |ba

Obtain: i) LMD; ii. RMD iii. Parse tree for LMD iv. Parse tree for RMD for the string 'aabbaa'

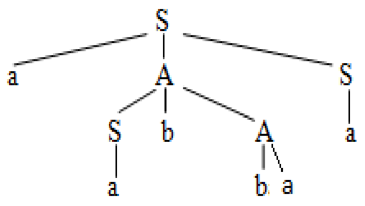
RMD:

S => aAS
⇨ aAa
⇨ aSbAa
⇨ aSbbaa
⇨ **aabbaa**

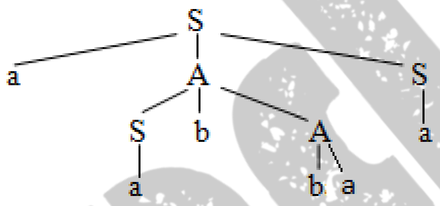
LMD:

S => aAS
⇨ aSbAS
⇨ aabAS
⇨ aabbaS
⇨ **aabbaa**

Parse tree for RMD:



Parse tree for LMD:



Problem 2:

Design a grammar for valid expressions over operator – and /. The arguments of expressions are valid identifier over symbols a, b, 0 and 1. Derive LMD and RMD for string w = (a11 – b0) / (b00 – a01). Write parse tree for LMD

Answer:

Grammar for valid expression:

E → E – E | E / E | (E) |I

I → a | b | Ia |Ib | I0 |I1

LMD for string (a11 – b0) / (b00 – a01):

E => E / E

⇨ (E) / E

⇨ (E - E) / E

⇨ (I – E) / E

⇨ (I1 – E) / E

⇨ (I11 – E) / E

⇨ (a11 – E) / E

⇨ (a11 – I) / E

⇨ (a11 – I0) / E

⇨ (a11 – b0) / E

⇨ (a11 – b0) / (E)

⇨ (a11 – b0) / (E - E)

⇨ (a11 – b0) / (I - E)

⇨ (a11 – b0) / (I0 - E)

⇨ (a11 – b0) / (I00 - E)

⇨ (a11 – b0) / (b00 - E)

⇨ (a11 – b0) / (b00 - I)

⇨ (a11 – b0) / (b00 – I1)

⇨ (a11 – b0) / (b00 – I01)

⇨ **(a11 – b0) / (b00 – a01)**

RMD : E => E / E

⇨ E / (E)

⇨ E / (E-E)

⇨ E / (E-I)

⇨ E / (E-I1)

⇨ E / (E-I01)

⇨ E / (E-a01)

⇨ E / (I-a01)

⇨ E / (I0-a01)

⇨ E / (I00-a01)

⇨ E / (b00-a01)

⇨ (E) / (b00-a01)

⇨ (E-E) / (b00-a01)

⇨ (E-I) / (b00-a01)

⇨ (E-I0) / (b00-a01)

⇨ (E-b0) / (b00-a01)

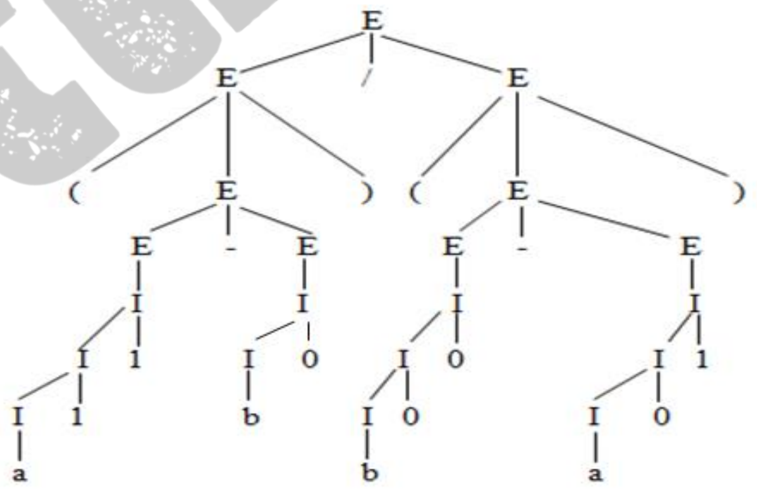
⇨ (I-b0) / (b00-a01)

⇨ (I1-b0) / (b00-a01)

⇨ (I11-b0) / (b00-a01)

⇨ **(a11-b0) / (b00-a01)**

Parse Tree for LMD:



Problem 3:

Consider the following grammar G:

E → + EE | * EE | - EE | x | y

Find the: i) LMD; ii. RMD iii. Parse tree for the string '+*-xyxy'

Answer:

E → + EE | * EE | - EE | x | y

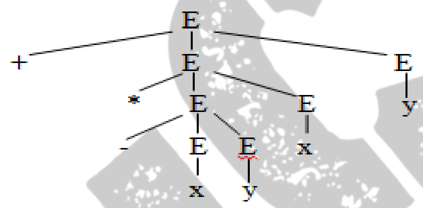
LMD:

E => + EE
⇨ +*EEE
⇨ +*-EEEE
⇨ +*-xEEE
⇨ +*-xyEE
⇨ +*-xyxE
⇨ +*-xyxy

RMD:

E => + EE
⇨ +Ey
⇨ +*EEy
⇨ +*Exy
⇨ +*-EExy
⇨ +*-Eyxy
⇨ +*-xyxy

Parse tree for LMD:



Problem 4:

Show the derivation tree for the string 'aabbbb' with grammar:

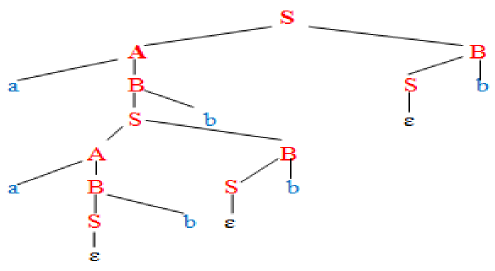
S → AB |ε

A → aB

B → Sb

Give a verbal description of the language generated by this grammar.

Answer: **Derivation tree:**



Verbal Description of the language generated by this grammar:

From the above grammar we can generate strings of ε, abb, aabbbb, aaabbbbbb…….etc.

Therefore the language corresponding to the above grammar is L = { $a^n b^{2n}$ | n ≥ 0 }.

That means for 'n' number of a's '2n' number of b's should be generated.

Problem 5:

Consider the following grammar:

E → E+E | E-E

E →E*E | E/E

E → (E)

E → a | b | c

i. Obtain the LMD for the string (a + b * c)

ii. Obtain the RMD for the string (a + b)* c**)**

Answer:

LMD for the string (a+ b * c)

E => (E)

⇨ (**E** * E)

⇨ (**E**+E * E)

⇨ (a +**E** * E)

⇨ (a + b * **E**)

⇨ (a +b * c)

RMD for the string (a+ b) * c

E => E * **E**

⇨ **E** * c

⇨ (**E**) * c

⇨ (E + **E**) * c

⇨ (**E** + b) * c

⇨ (a + b) * c

Problem 6:

Consider the following grammar:

S → AbB

A →aA |ε

B → aB | bB |ε

Give LMD, RMD and parse tree for the string aaabab

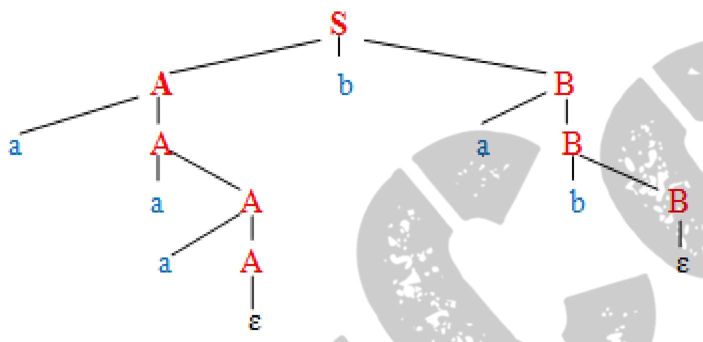
LMD:

S => AbB
⇨ aAbB
⇨ aaAbB
⇨ aaaAbB
⇨ aaabB
⇨ aaabaB
⇨ aaababB
⇨ aaabab

RMD:

S => AbB
⇨ AbaB
⇨ AbabB
⇨ Abab
⇨ aAbab
⇨ aaAbab
⇨ aaaAbab
⇨ aaabab

Parse tree for LMD:



Obtain the context free grammar for generating integers and derive the integer 1278 by applying LMD.

The context free grammar corresponding to the language containing set of integers is G = (V, T, P, S) where, V = { I, N, D }, T = { 0, 1}, **I** is the start symbol, and P is the production rule is as shown below:

I → N | SN

S → + | - | ε

N → D | DN | ND

D → 0 | 1 | 2 | 3 | ……….| 9

LMD for the integer 1278:

I => N
⇨ ND
⇨ NDD
⇨ NDDD

⇨ DDDD

⇨ 1DDD

⇨ 12DD

⇨ 127D

⇨ **1278**

AMBIGUOUS GRAMMAR

Sometimes a Context Free Grammar may produce more than one parse tree for some (or all) of the strings it generates. When this happens, we say that the grammar is ambiguous. More precisely. a grammar G is ambiguous if there is at least one string in L(G) for which G produces more than one parse tree.

***What is an ambiguous grammar?

A context free grammar **G** is an ambiguous grammar if and only if there exists at least one string '**w**' is in L(G) for which grammar G produces **two or more** different parse trees by applying either LMD or RMD.

Show how ambiguity in grammars are verified with an example.

Testing of ambiguity in a CFG by the following rules:

i. Obtain the string '**w**' in L(G) by applying LMD twice and construct the parse tree. If the two parse trees are different, then the grammar is ambiguous.

ii. Obtain the string '**w**' in L(G) by applying RMD twice and construct the parse tree. If the two parse trees are different, then the grammar is ambiguous.

iii. Obtain the LMD and get a string '**w**'. Obtain the RMD and get the same string '**w**' for both the derivations construct the parse tree. If there are two different parse trees then the grammar is ambiguous.

Show that the following grammar is ambiguous:

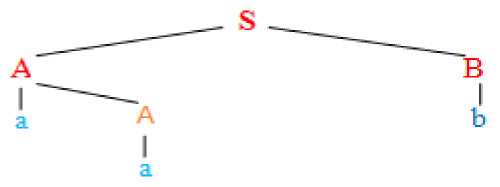
$S \rightarrow AB \mid aaB$

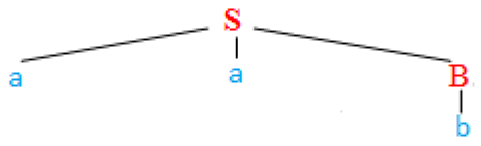
$A \rightarrow a \mid Aa$

$B \rightarrow b$

Let us take the string w= **aab**

This string has two parse trees by applying LMD twice so the grammar is ambiguous;





Show that the following grammar is ambiguous:

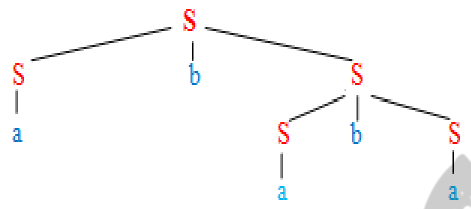
S → SbS

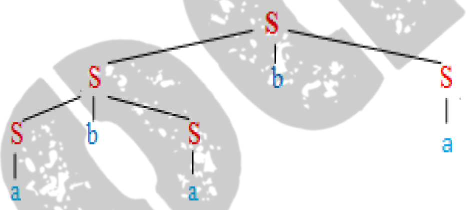
S→ a

Answer:

Take string like w = **ababa**

This string has two parse trees by applying LMD twice so the grammar is ambiguous;





Show that the following CFG is an ambiguous grammar:

E → E+E

E →E*E

E → a | b | c

Let us consider the string : **a + b + c**

*First LMD for the string **a + b + c** is:*

E => **E**+E

⇨ **E**+ E +E

⇨ a + **E** +E

⇨ a +b +**E**

⇨ **a + b + c**

*Second LMD for the string **a + b + c** is:*

E => **E**+E

⇨ **a** + **E**

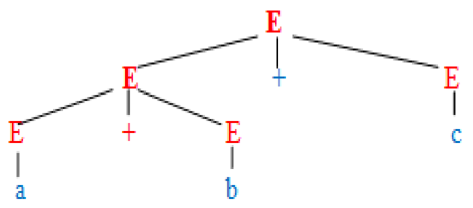
⇨ a + **E** +E

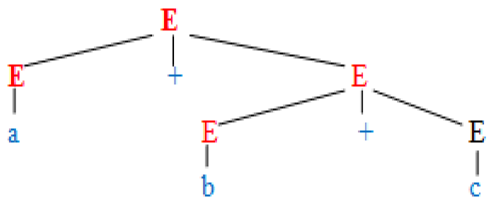
⇨ a + b +**E**

⇨ **a + b + c**

Parse Tree for LMD1:

Parse Tree for LMD2:





The grammar is ambiguous, because we are getting two different parse trees for the same string by applying LMD twice.

Associativity and Precedence Priority in CFG:

Example:

E → E+E| E-E

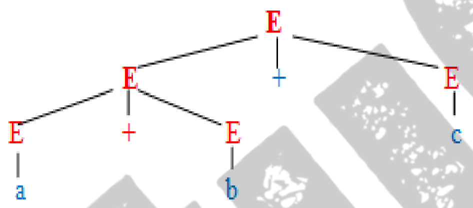
E →E*E

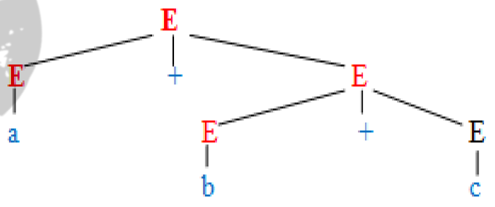
E → a | b | c

Associativity:

Let us consider the string : **a + b + c**

Parse Tree for LMD1: **Parse Tree for LMD2:**





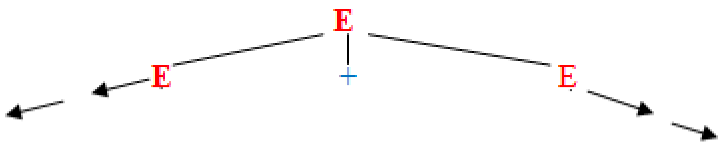
The two different parse trees exist because of the associativity rules fails. That means for the given string a **+ b +** c; on either side of the operand '**b'**, there exist two operators. Which operator should I associate with operand **b**? This ambiguity results in either I should consider the operand '**b**' with left side operator (Left associative) or right side (Right associative) operator. So the first parse tree is correct, where the left most '**+**' is evaluated first.

How to resolve the associtivity rules:

E → E + E

E → a | b | c

Here the grammar is not defined in the proper order, **ie:** the growth of the tree is in either left direction or right direction.



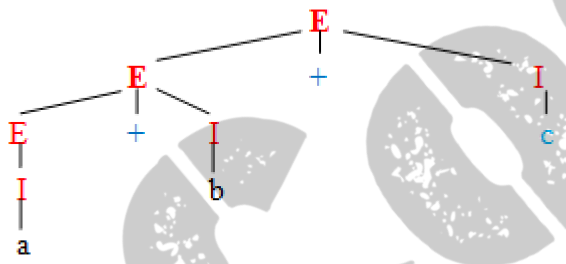
The growth of the first parse tree is in left direction. That means it is left associative. The growth second parse tree is in right direction, ie: right associative.

For normal associative rule is left associative, so we have to restrict the growth of parse tree in right direction by modifying the above grammar as:

E → E + I | I

I→ a | b | c

The parse tree corresponding to the string: a+b+c:



The growth of the parse tree is in left direction since the grammar is left recursive, therefore it is left associative. There is only one parse tree exists for the given string. So the grammar is ambiguous.

Note: For the operators to be left associative, grammar should be left recursive. Also for the operators to be right associative, grammar should be right recursive.

Left Recursive grammar: A production in which the leftmost symbol of the body is same as the non-terminal at the head of the production is called a left recursive production.

Example: E → E + T

Right Recursive grammar: A production in which the rightmost symbol of the body is same as the non-terminal at the head of the production is called a right recursive production.

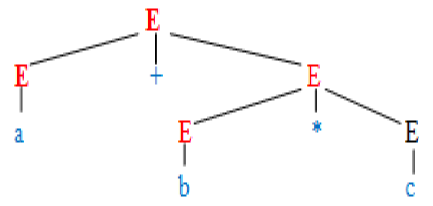
Example: E → T + E

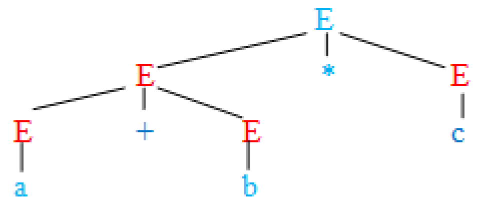
Precedence of operators in CFG:

Let us consider the string: **a + b * c**

LMD 1 for the string: a+b*c

LMD 2 for the string: a+b*c





The first parse tree is valid, because the highest precedence operator '*' is evaluated first compared to '+'. (See the lower level of parse tree, where '*' is evaluated first). The second parse tree is not valid, since the expression containing '+' is evaluated first. So here we got two parse trees because of the precedence is not taken care.

So if we take care of associativity and precedence of operators in CFG, then the grammar is un-ambiguous.

NOTE:

Normal precedence rule: If we have the operators such as +, -, *, /, ↑, then the highest precedence operator ↑ is evaluated first.

Next highest precedence operator * and / is evaluated. Finally the least precedence operator + and – is evaluated.

Normal Associativity rule: Grammar should be left associative.

How to resolve the operator precedence rules:

E → E+E| E *E

E → a | b | c

If we have two operators of different precedence, then the highest precedence operator (*) is evaluated first. This can be done by re- writing the grammar with least precedence operator at the first level and highest precedence operator at the next level.

E → E + T ; + is left associative, because the non-terminal symbol to the left of + is same as that of non- terminal at the LHS.

At the first level expression containing all '+'s are generated.

Suppose if no '+'s are included in the grammar, then we have to bypass the grammar E + T.

This can be done by:

E → T

Finally the first level grammar is

E → E + T | T

Similarly at the second level, we have to generate all '*'s.

T → T * F ; * is left associative.

If the expression does not contain any '*'s, then we have to bypass the grammar T → T * F

T → F

Finally the second level grammar is

T → T * F | F

Third level:

F → a | b | c

So the resultant un-ambiguous grammar is:

E → E + T | T

T → T * F | F

F → a | b | c

So the operator which is closest to the start symbol has least precedence and the operator which is farthest away from start symbol has the highest precedence.

Un-Ambiguous Grammar:

For a grammar to be un-ambiguous we have to resolve the two properties such as:

i. Associativity of operators: This can be resolved by writing the grammar recursion.

ii. Precedence of operators: can be resolved by writing the grammar in different levels.

Is the following grammar is ambiguous?

If the grammar is ambiguous, obtain the un-ambiguous grammar assuming normal precedence and associativity.

E → E + E

E → E * E

E → E / E

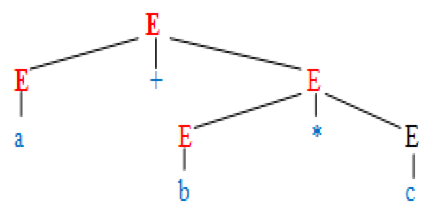
E → E - E

E → (E) | a | b| c **10**

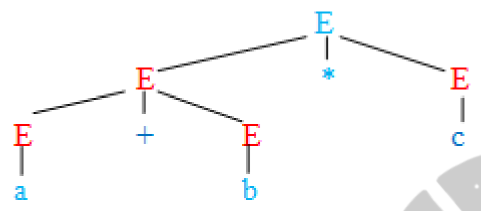
Answer:

Let us consider the string: a + b * c

LMD 1 for the string: a+b*c



LMD 2 for the string: a+b*c



For the given string there exists two different parse trees, by applying LMD twice. So the above grammar is ambiguous.

The equivalent un-ambiguous grammar is obtained by writing all the operators as left associative and writing the operators +, – at the first level and *, / at the next level.

Equivalent un-ambiguous grammar:

E → E + T | E – T | T

T → T * F | T / F | F

F → (E) | a | b | c

Is the following grammar is ambiguous?

If the grammar is ambiguous, obtain the un-ambiguous grammar assuming the operators + and – are left associative and * and / are right associative with normal precedence .

E → E + E

E → E * E

E → E / E

E → E - E

E → (E) | a | b| c

Ambiguous grammar------- see the previous answer.

Equivalent un-ambiguous grammar:

E → E + T | E – T | T

T → F * T | F / T | F

F → (E) | a | b | c

Is the following grammar is ambiguous?

If the grammar is ambiguous, obtain the un-ambiguous grammar assuming the operators + and / are left associative and * and - are right associative with + and / has the highest precedence and * and – has the least precedence**.**

E → E + E

E → E * E

E → E / E

E → E - E

E → (E) | a | b| c

Equivalent un-ambiguous grammar:

E → T - E | T * E | T

T → T + F | T / F | F

F → (E) | a | b | c

Consider the grammar:

S → aS | aSbS | ε

Is the above grammar ambiguous? Show in particular that the string ‘aab’ has two:

i. Parse trees

ii. Left Most Derivations

iii. Right Most Derivations.

OR

Define ambiguous grammar. Prove that the following grammar is ambiguous.

S → aS | aSbS| **ε**

<u>LMD 1 for the string ‘**aab**’:</u>

S => aS

⇨ aaSbS

⇨ aabS

⇨ aab

<u>LMD 2 for the string ‘**aab**’:</u>

S => aSbS

⇨ aaSbS

⇨ aabS

⇨ aab

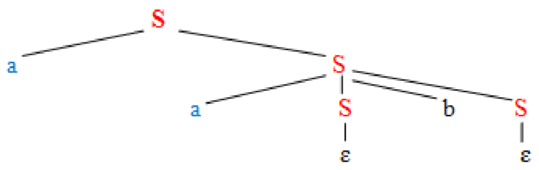
RMD 1 for the string '**aab**':

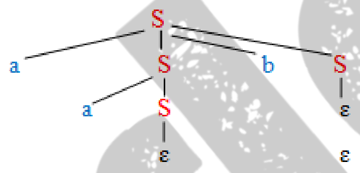
S => aS
⇨ aaSbS
⇨ aaSb
⇨ aab

RMD 2 for the string '**aab**':

S => aSbS
⇨ aSb
⇨ aaSb
⇨ aab

Two Parse trees for LMD:





The above grammar is ambiguous, since we are getting two parse trees for the same string '**aab**' by applying LMD twice.

Consider the grammar:

S → S +S | S * S | (S) | a

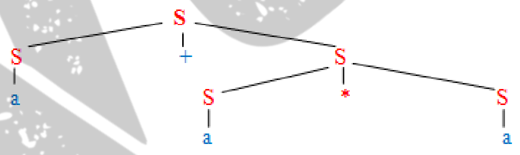
Show that string a + a * a has two

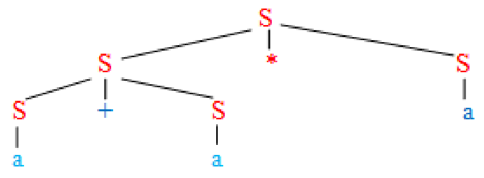
i. Parse trees

ii. Left Most Derivations

Find an un-ambiguous grammar G' equivalent to G and show that L (G) = L (G') and G' is un-ambiguous.

Two different parse trees for the string a+a*a :





Two LMDs:

S => S +S
⇨ a + S
⇨ a + S * S
⇨ a + a * S
⇨ a + a * a

S => S * S
⇨ S + S * S
⇨ a + S * S
⇨ a + a * S
⇨ a + a * a

Un-ambiguous grammar Corresponding to the above grammar (G) is:

G' = (V', T', P', S) where V' = { S, T, F }, T' = { a }, S is the start symbol and P' is the production given by:

$S \rightarrow S + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (S) \mid a$

Consider the string **a + a * a (Generate this string using G' using LMD)**

$S \Rightarrow S + T$
$\Rightarrow T + T$
$\Rightarrow F + T$
$\Rightarrow a + T$
$\Rightarrow a + T * F$
$\Rightarrow a + F * F$
$\Rightarrow a + a * F$
$\Rightarrow a + a * a$

So we proved that same language can be generated using G'

NOTE: Suppose if we have an exponential operator ($\uparrow$) in an expression such as;

$2^{3^{2}}$; This can be represented as $2 \uparrow 3 \uparrow 2$ where 3^2 is evaluated first as 9, then 2^9 is evaluated. That means the evaluation starts from right side; therefore the operator $\uparrow$ is right associative.

Any expression containing the operators such as: +, -, *, / and $\uparrow$

$\uparrow$ highest precedence (Farthest away from start symbol)

*, / next highest precedence. (next least level)

+, - least precedence (closest to the start symbol)

Show that the following grammar is ambiguous. Also find the un-ambiguous grammar equivalent to the grammar by normal precedence and associative rules.

$E \rightarrow E + E \mid E - E$

$E \rightarrow E * E \mid E / E$

$E \rightarrow E \uparrow E$

$E \rightarrow (E) \mid a \mid b$

Answer:

We already proved that the above grammar is ambiguous

Equivalent Un-ambiguous grammar:

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

$T \rightarrow T * F \mid T / F \mid F$

$F \rightarrow G \uparrow F \mid G$

$G \rightarrow (E) \mid a \mid b$

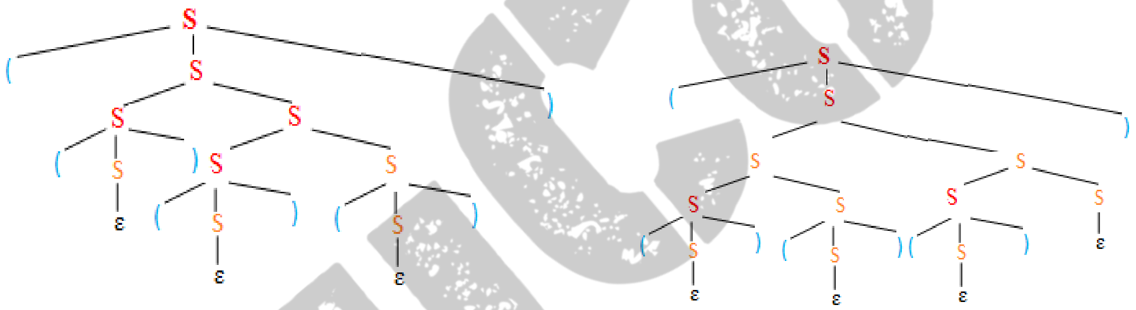
Here the operator $\uparrow$ is right associative.

Show that the following grammar is ambiguous: **4**

$S \rightarrow SS$

$S \rightarrow (S) \mid \varepsilon$ over string w = (( ) ( ) ( ))

The given string has two parse trees by applying LMD twice so the grammar is ambiguous;



Show that the following grammar is ambiguous using the string " ibtibtaea"

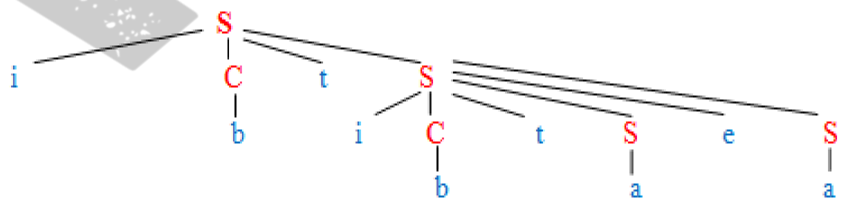
$S \rightarrow iCtS \mid iCtSeS \mid a$

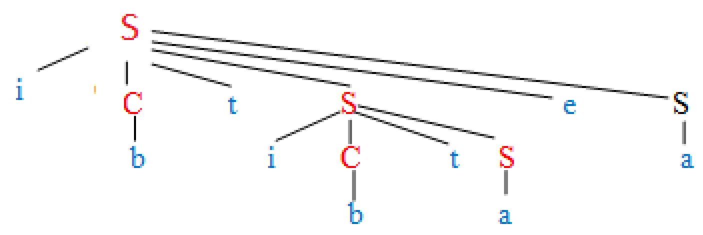
$C \rightarrow b$

Answer:

String w = ibtibtaea

The given string has two parse trees by applying LMD twice:





For the grammar

S → SS + | SS * | a Give the left most and right most derivation for the string **aa+a***

Leftmost derivation of the string aa+a*:

S => SS*

=> SS+S*

=>aS+S*

=>aa+S*

=>aa+a*

Rightmost derivation

S => SS*

=> Sa*

=> SS+a*

=> Sa+a*

=> aa+a*

Dangling- else grammar:

stmt → **if** expr **then** stmt | **if** expr **then** stmt **else** stmt | **other**

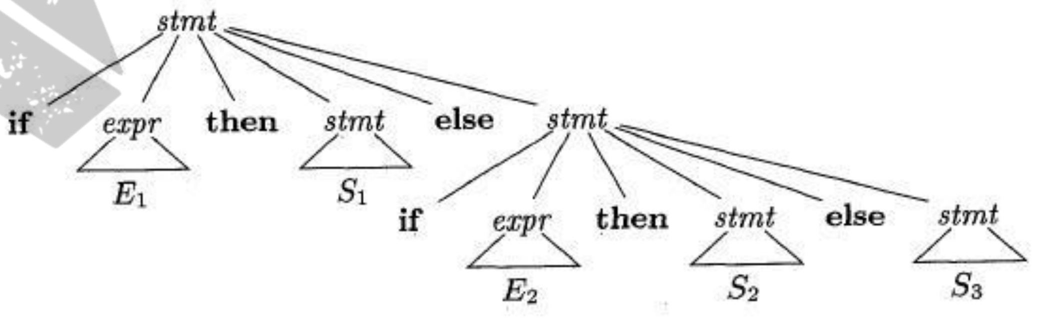
Terminals are keywords **if, then** and **else.**

Non terminals are **expr** and stmt.

Here "**other**" stands for any other statement. According to this grammar one of the compound conditional statement can be written as

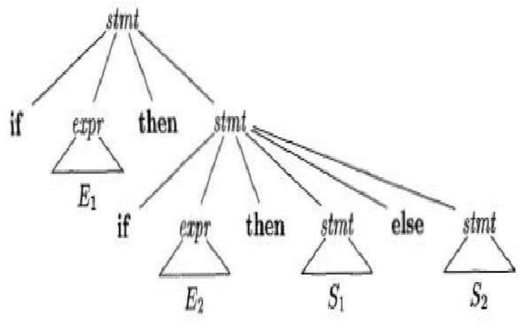
if E_1 **then** S_1 **else if** E_2 **then** S_2 **else** S_3

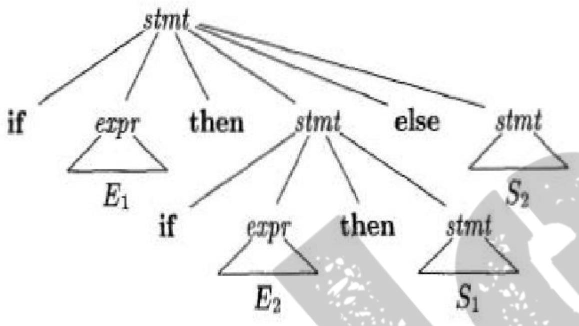
It has the parse tree as shown below:



Example: Let us consider the string **if** E_1 **then if** E_2 **then** S_1 **else** S_2

It has two parse trees, so the above grammar is an ambiguous grammar.





In all programming languages with conditional statements of this form, the first parse tree is preferred. The general rule is match each **else** with the closest unmatched **then**.

Unambiguous grammar for this if else statements:

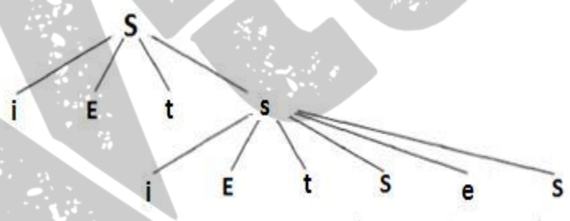
stmt	→	matched_stmt | open_stmt
matched_stmt	→	**if** expr **then** matched_stmt **else** matched_stmt | **other**
ope_stmt	→	**if** expr **then** stmt | **if** expr **then** matched_stmt **else** open_stmt

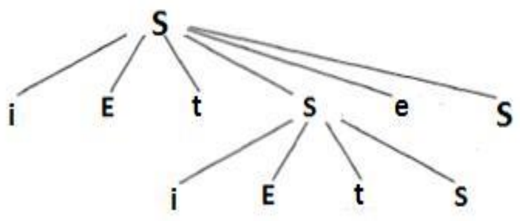
Show that the following grammar is ambiguous:

S	→	iEtS | iEtSeS | a
E	→	b

Also write an equivalent unambiguous grammar for the same.

Let us consider the string **iEtiEtSeS,** which has two parse trees.





Unambiguous grammar :

S	→	M |U
M	→	iEtMeM |a
U	→	iEtS | iEtMeU
E	→	b

LEFT RECURSION

A production in which the leftmost symbol of the body is same as the non-terminal at the head of the production is called a left recursive production.

Example: $E \rightarrow E + T$

Immediate Left recursive production:

A production of the form $A \rightarrow A\alpha$ is called an immediate left recursive production. Consider a non-terminal A with two productions

$$A \rightarrow A\alpha \mid \beta$$

Where α and β are sequence of terminals and non-terminals that do not start with A.

Repeated application of this production results in sequence of α's to the right of A. When A is finally replaced by β, we have β followed by a sequence of zero or more α^s.

Therefore a non-left recursive production for $A \rightarrow A\alpha \mid \beta$ is given by

$$\mathbf{A \rightarrow \beta A'}$$
$$\mathbf{A' \rightarrow \alpha A' \mid \varepsilon}$$

Note: In general we can eliminate any immediate left recursive production of the form

$$\mathbf{A \rightarrow A\alpha_1 \mid A\,\alpha_2 \mid A\alpha_3 \ldots\ldots\ldots\ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots\ldots\ldots\ldots \mid \beta_n}$$

By replacing **A** production by

$$\mathbf{A \rightarrow \beta 1 A' \mid \beta 2\, A' \mid \beta 3\, A' \mid \ldots\ldots\ldots\ldots \mid \beta n\, A'}$$
$$\mathbf{A' \rightarrow \alpha 1\, A' \mid \alpha 2\, A' \mid \alpha 3\, A' \mid \ldots\ldots\ldots\ldots\ldots \mid \alpha m\, A' \mid \varepsilon}$$

no β_i begins with A

What is left recursion?

A grammar is left recursive if it has a non-terminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string α.

Top down parsing methods cannot handle left recursive grammars, so a transformation is needed to eliminate left recursion.

A grammar containing productions results in left recursive productions, after applying two or more steps of derivations can be eliminated using the following algorithm.

Algorithm to eliminate left recursion from a grammar having no ε production:

Write an algorithm to eliminate left recursion from a grammar.

1. Arrange the non-terminals in some order A1, A2, . . . , A_n

2. **for** (each **i** from **1** to **n**)

    {

```
3.    for ( each j from 1 to i - 1 )
      {
4.        replace each production of the form $A_i \rightarrow A_j \Gamma$ by the
          productions $A_i \rightarrow \partial_1 \Gamma \mid \partial_2 \Gamma \mid \ldots\ldots \mid \partial_k \Gamma$, where $A_j \rightarrow \partial_1 \mid \partial_2 \mid \ldots\mid \partial_k$ are all current
          $A_j$ - productions.
5.    }
6.        eliminate the immediate left recursion among the $A_i$-productions.
7.    }
```

Eliminate left recursion from the following grammar:

E → E + T | T

T → T * F | F

F → (E) | id

The above grammar contains two immediate left recursive productions **E → E + T | T** and

T → T * F | F

The equivalent non-left recursive grammar is given by:

E → TE'

E' → +TE' | ε

T → FT'

T' → *FT' | ε

F → (E) | id

Eliminate left recursion from the following grammar:

E → E + T | T

T → id | id [ ] | id [X]

X → E , E | E

The given grammar has one immediate left recursive productions **E → E + T | T**

The equivalent non-left recursive grammar is given by:

E → TE'

E' → +TE' | ε

T → id | id [ ] | id [X]

X → E , E | E

Eliminate left recursion from the following grammar:

S → Aa | b

A → Ac| Sd | a

By applying elimination algorithm,

Arrange the non-terminals as A_1 = S and A_2 = A

Since there is no immediate left recursion among S production, so nothing happens during the outer loop for i =1.

For i =2, we substitute for S in A → Sd to obtain the following A productions.

A → Ac| Aad | bd | a

Eliminating the immediate left recursion among these A- productions yields the following grammar

S → Aa | b

A → bdA' | aA'

A' → cA' | adA' | ε

Eliminate left recursion from the following grammar:

A → BC | a

B → CA| Ab

C → AB| CC | a

Arrange the non-terminals as A_1 = A , A_2 = B and A_3 = C

Since there is no immediate left recursion among A production, so nothing happens during the outer loop for i =1. ie **A → BC | a**

For i =2, we substitute for A in B → Ab to obtain the following A productions.

B → CA| BCb | ab

Eliminating the immediate left recursion among these B- productions results in a new B productions as **B → CAB' | abB'**

B' → CbB' |ε

For i =3, we substitute for A in C → AB to obtain the following C productions

C → BCB| CC | aB |a

Again substitute for B in C → BCB production to obtain the C productions as

$C \rightarrow CAB'CB \mid abB'CB \mid CC \mid aB \mid a$

Eliminating the immediate left recursion among these C- productions results in new C productions as

$\mathbf{C \rightarrow abB'CBC' \mid aBC' \mid aC'}$

$\mathbf{C' \rightarrow AB'CB\,C' \mid CC' \mid \varepsilon}$

The equivalent non- left recursive grammar is given by:

$A \rightarrow BC \mid a$

$B \rightarrow CAB' \mid abB'$

$B' \rightarrow CbB' \mid \varepsilon$

$C \rightarrow abB'CBC' \mid aBC' \mid aC'$

$C' \rightarrow AB'CB\,C' \mid CC' \mid \varepsilon$

Eliminate left recursion from the following grammar.

$Lp \rightarrow no \mid Op\ Ls$

$Op \rightarrow +\ 1 - 1\ *$

$Ls \rightarrow Ls\ Lp \mid Lp$

For i = 1 and 2 nothing happens to the production L_p and O_p.

For i= 3

By removing immediate left recursion,

$Ls \rightarrow L_p\ Ls'$

$Ls' \rightarrow Lp\ Ls' \mid \varepsilon$

The equivalent non- left recursive grammar is given by:

$Lp \rightarrow no \mid Op\ Ls$

$Op \rightarrow +\ 1 - 1\ *$

$Ls \rightarrow L_p\ Ls'$

$Ls' \rightarrow Lp\ Ls' \mid \varepsilon$

Eliminate left recursion from the following grammar:

$S \rightarrow aB \mid aC \mid Sd \mid Se$

$B \rightarrow bBc \mid f$

$C \rightarrow g$

For i =1 , results in a new S productions as

$S \rightarrow aB S' \mid aC S'$

$S' \rightarrow d S' \mid eS' \mid \varepsilon$

For i =2 nothing happens to B productions, $B \rightarrow bBc \mid f$

For i =3 nothing happens to C productions $C \rightarrow g$

The equivalent non- left recursive grammar is given by:

$S \rightarrow aB S' \mid aC S'$

$S' \rightarrow d S' \mid eS' \mid \varepsilon$

$B \rightarrow bBc \mid f$

$C \rightarrow g$

LEFT FACTORING (Non-deterministic to Deterministic CFG conversion)

It is a grammar transformation method used in parser. When the choice between two alternative **A** productions is not clear, we can rewrite the productions so to make the right choice.

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots\ldots\ldots \mid \alpha\beta_n \mid \Gamma$

By left factoring this grammar, we get

$A \rightarrow \alpha A'' \mid \Gamma$

$A'' \rightarrow \beta_1 \mid \beta_2 \ldots\ldots\ldots\ldots\ldots \mid \beta_n$

Γ is other alternatives that do not begin with α.

A predictive parser (a top-down parser without backtracking) insists that the grammar must be left-factored.

What is left factoring?

Left factoring is removing the common left factor that appears in two or more productions of the Same non-terminal.

Example: $\mathbf{S} \rightarrow i\ EtSeS \mid iEtS \mid a$

$E \rightarrow b$

By left factoring the above grammar we get,

$$\mathbf{S} \rightarrow \text{i EtS}\mathbf{S}' \mid \text{a}$$
$$\mathbf{S}' \rightarrow \mathbf{eS} \mid \boldsymbol{\varepsilon}$$
$$E \rightarrow b$$

Perform left factoring for the grammar.

$E \rightarrow E + T \mid T$

$T \rightarrow id \mid id\ [\ ] \mid id\ [\ X\ ]$

$X \rightarrow E\ ,\ E \mid E$

The equivalent non-left recursive grammar is given by:

$E \rightarrow TE'$

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

$T \rightarrow id \mid id\ [\ ] \mid id\ [\ X\ ]$

$X \rightarrow E\ ,\ E \mid E$

After left factoring the grammar, we get

$E \rightarrow TE'$

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

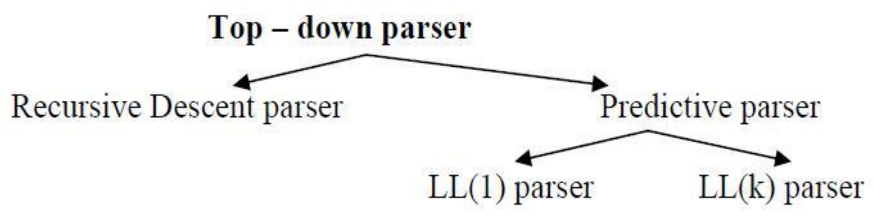
$T \rightarrow id\ T'$

$T' \rightarrow \varepsilon \mid [\ ] \mid [\ X] \mid$

$X \rightarrow E\ X'$

$X' \rightarrow ,\ E \mid \varepsilon$

TOP DOWN PARSER



- Top-down parsing can be viewed as the problem of constructing a parse tree for the input string, starting from the root (Top) and working up towards the leaves (Down).
- Equivalently, top-down parsing can be viewed as finding a leftmost derivation for an input string.
- At each step of a top-down parse, the key problem is that of determining the production to be applied for a non-terminal, say A.
- Once an A-production is chosen, the rest of the parsing process consists of "matching" the terminal symbols in the production body with the input string.

RECURSIVE-DESCENT PARSING

- Backtracking is needed (If a choice of a production rule does not work, we backtrack to try other alternatives.)
- It is a general parsing technique, but not widely used.
- Not an efficient parsing method.

A left-recursive grammar can cause a recursive-descent parser to go into an infinite loop, so we have to eliminate left recursion from a grammar

Recursive-Descent Parsing Algorithm:

Explain Recursive-Descent Parsing Algorithm.

```
void A ( )
    {
1.      Choose an A-production, A → $X_1$ $X_2$ $X_3$ ................................ $X_k$ ;
2.      for (i = 1 to k)
    {
3.  if ( $X_i$ is a non-terminal )
4.  call procedure Xi ( ) ;
5.  else if (Xi equals the current input symbol a)
6.  advance the input to the next symbol;
7.  else /* an error has occurred */;
    }
    }
```

A recursive-descent parsing program consists of a set of procedures, one for each non-terminal. Execution begins with the procedure for the start symbol, which halts and announces success if its procedure body scans the entire input string.

General recursive-descent may require backtracking; that is, it may require repeated scans over the input. However, backtracking is rarely needed to parse programming language constructs, so Back-tracking parsers are not seen frequently.

What are the key problems with top down parser? Write a recursive descent parser for the grammar:

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

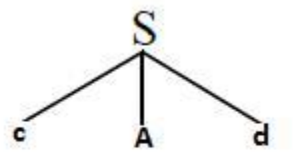
Key problem:

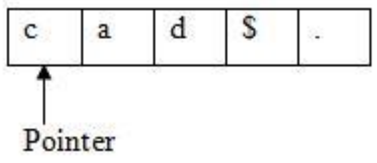
1. Ambiguous grammar
2. Left recursive grammar
3. Left factoring
4. Backtracking

At each step of a top-down parse, the key problem is that of determining the production to be applied for a non-terminal.

CONSTRUCTION OF RECURSIVE DESCENT PARSER:

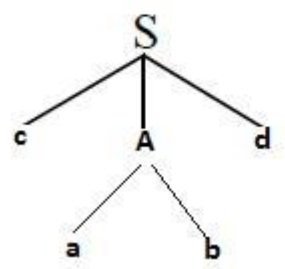
Let us consider the input string **w** = cad, begin with a tree consisting of a single node labeled S. Input pointer points to **c**, the first symbol of w. S has only one production, so we use it to expand S and obtain the tree as shown below:

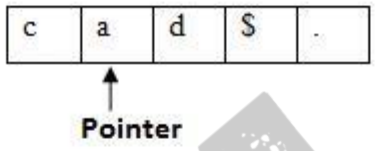




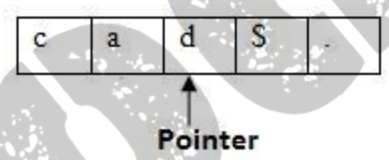
The leftmost leaf labeled **c**, matches the first symbol of input **w**, so we advance the input pointer to **a**, the second symbol of **w**, and consider the next leftmost leaf labeled A.

Expand A using the first alternative **A** → a b to obtain the following tree:

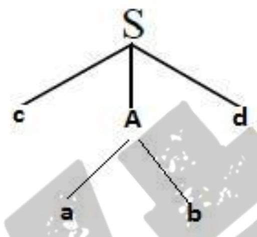




Now we have a match for the second input symbol **a,** with the leftmost leaf labeled **a**, so we advance the input pointer to **d**, third input symbol of **w**.



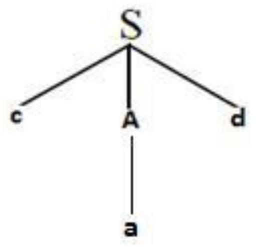
Now compare the current input symbol **d** against the next leaf labeled **b**. Since **b** does not match **d** ,we report failure and go back to **A** (Back tracking) to see whether there is another alternative for **A** that has not been tried, but that might produce a match.

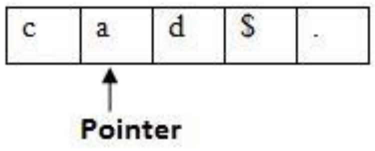


Failure and Backtrack

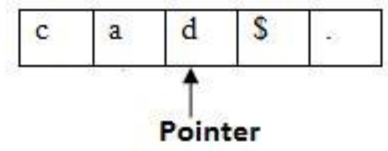
In going back to A, we must reset the input pointer to position 2, the position it had when we first came to A. (procedure must store the pointer in a local variable).

Now the second alternative for A produces the tree as,

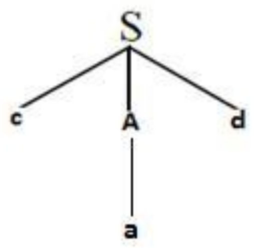


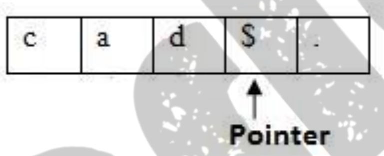


Now the leftmost leaf labeled **a** matches the current input symbol **a**, ie: the second symbol of **w**, then advance the pointer to the next input symbol **d**.



Now the next leaf **d** matches the third input symbol **d**, later when it finds **$** nothing is left out to be read in the tree. Since it produces a parse tree for the string **w**, it halts and announce successful completion of parsing.





return success

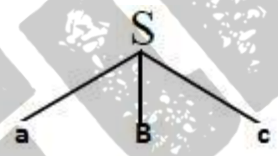
Write a recursive descent parser for the grammar:

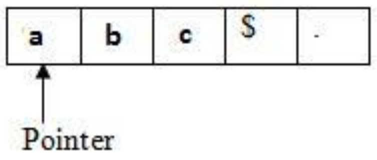
S → aBc

B → bc | b

Input: **abc**

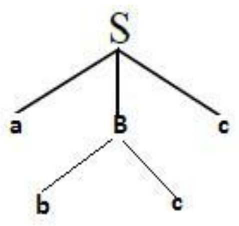
Begin with a tree consisting of a single node labeled **S** with input pointer pointing to first input symbol **a**.

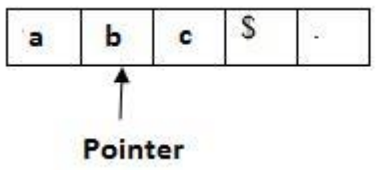




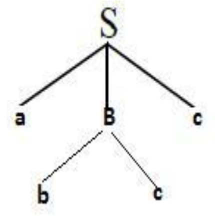
Since the input a matches with leftmost leaf labeled **a**, advance the pointer to next input symbol **b**.

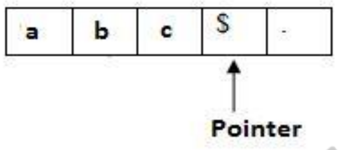
Expand **B** using the alternative **B** → bc





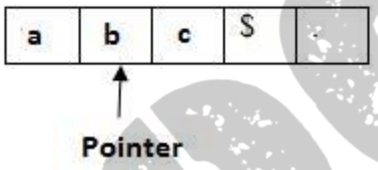
We have a match for second input symbol **b**. Move the pointer again it finds the match for third symbol c. Now the pointer is pointing to **$**, indicating the end of string, but in the tree we find one more symbol **c** to be read, thus it fails



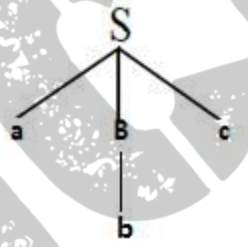


Failure and Backtrack

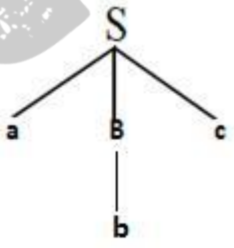
Now it goes back to B and reset the input pointer to position 2.



When the pointer is set to position 2, it checks the second alternative and generates the tree ;



Now the pointer moves to the 2nd symbol finds a match, then advances to the 3rd symbol finds a match, later when it encounters „$" nothing is left out to be read in the tree. Thus it halts and announce successful completion of parsing.



return success

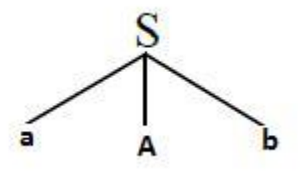
Show that recursive descent parsing fails for the input string 'acdb' for the grammar.

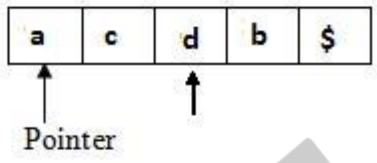
S → aAb

A → cd | c

Input string: acdb

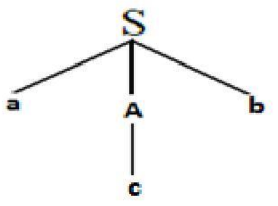
Begin with a tree consisting of a single node labeled **S** with input pointer pointing to first input symbol **a .**

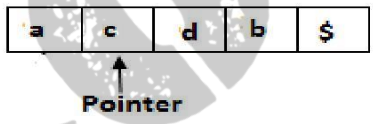




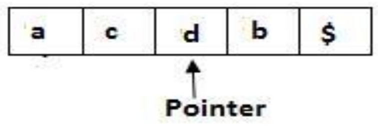
The first input symbol a matches with left most leaf a and advance the pointer to next input symbol c.

Now expand **A** using the second alternative **A** → c

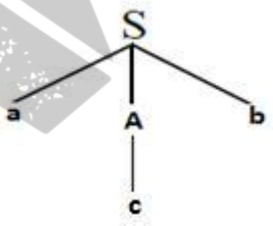


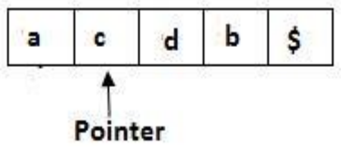


We have a match for second input symbol **c** with left leaf node **c**. Advance the pointer to the next input symbol d.



Now compare the input symbol **d** against the next leaf, labeled **b**. Since **b** does not match **d**, we report failure and go back to A to see another alternative for A and reset the pointer to position 2.





Failure and Backtrack

PREDICTIVE LL(1) PARSER (NON RECURSIVE PREDICTIVE PARSER)

Working of Predictive top down parser (LL(1) parser)

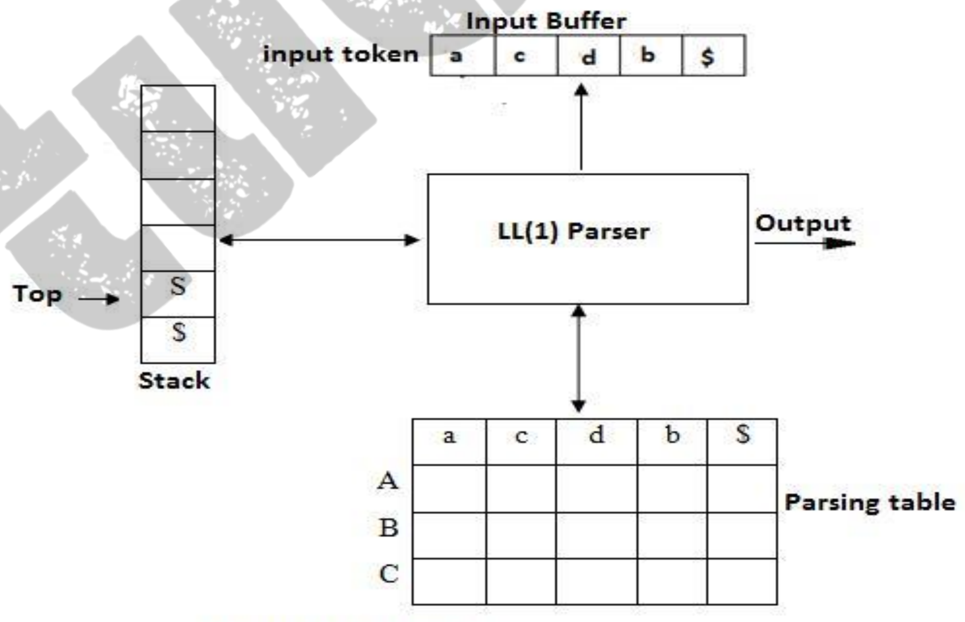
Back tracking problem of recursive descent parser can be solved in predictive parsing. This top down parsing algorithm is a non-recursive type of parsing. Predictive parser is capable to predict or detect, which alternatives are right choices for the expansion of non-terminals during the parsing of input string **w. In** order to parse the grammar by predictive top down parser, we perform the following operations:

- Writing a grammar for program statement
- Eliminate left recursion
- Left factoring the resultant grammar
- Apply the grammar to the predictive top down parser.

Explain with a neat diagram, the model of a table driven predictive parser.

LL(1) parser is a table driven parser, where a parsing table is built for LL(1). Here the first L stands for the input is scanned from left to right. The second L means it uses leftmost derivation for input string and the number 1 in the input symbol means it uses only one input symbol to predict the parsing process.

Model of LL(1) parser:



Model of LL(1) Parser

The data structures used by LL(1) parsers are;

1. **Input buffer**: used to store the input tokens.
2. **Stack** is used to hold the left sentential form, the symbols in RHS of the production rule are pushed into the stack in reverse order, ie: ***from right to left.***
3. A **parsing table** is a two dimensional array, which has row for non terminal and column for terminals.

The construction of predictive LL(1) parser is based on two very important functions and those functions are: FIRST( ) and FOLLOW( ). These two functions allow us to choose which production to apply, based on the next input symbol.

Steps involved in construction of LL(1) predictive parser:

i. Computation of FIRST ( ) and FOLLOW ( ) function.
ii. Construct the predictive parsing table using FIRST and FOLLOW functions
iii. Parse the input string with the help of predictive parsing table

FIRST() FUNCTION:

Define FIRST ( )

FIRST (α) is a set of terminal symbol, that are first symbol appearing at RHS in derivation of α.

If α => ε then ε is also in FIRST (α)

Give the rules for constructing the FIRST and FOLLOW sets.

Rules used in computation of FIRST function:

1. If there is a terminal symbol **a**, then the FIRST (a) = { a }.
2. If there is a production rule X → ε , then FIRST (X) = { ε }
3. If there is a production rule $X \rightarrow Y_1, Y_2, Y_3, \ldots\ldots Y_k$ then

   FIRST (X) = { **a**} if for some i , **a** is in FIRST (Y_i) and

   FIRST (X) = **ε**, if ε is in FIRST (Y_1), FIRST (Y_2), FIRST (Y_3), ……. FIRST (Y_k) and

   FIRST (X) = FIRST (Y_1) if Y_1 does not derive ε, but if $Y_1 \overset{*}{=>} \varepsilon$ then we add FIRST (Y_2) and so on.

   ie: FIRST (X) = FIRST (Y_1) – {ε} U FIRST (Y_2) – {ε} U …….. FIRST (Y_{k-1}) - {ε} U FIRST (Y_k)

   If there is a production rule $X \rightarrow Y_1 \mid Y_2 \mid Y_3 \mid \ldots\ldots \mid Y_k$ then FIRST (X) = FIRST (Y_1) U FIRST (Y_2) …………………… FIRST (Y_k)

FOLLOW ( ) FUNCTION:

FOLLOW (A) is defined as the set of terminal symbols that appear immediately to the right of A. ie : FOLLOW (A) = { **a** | S $\overset{*}{=>}$ αAaβ where α and β are some grammar symbols, may be terminal or non terminal symbols.

Rules used in computation of FOLLOW function:

1. For the start symbol **S** place '$' in FOLLOW (S).
2. If there is a production A→ αBβ, then everything in FIRST (β) except ε is in FOLLOW (B).
3. If there is a production A→ αBβ and FIRST(β) derives ε, then

FOLLOW(B) = FIRST (β) – {ε} U FOLLOW (A) and

if the production is A→ αB then FOLLOW (B) = FOLLOW (A**)**

Find FIRST and FOLLOW for the following grammar by eliminating left recursive productions.

E → E + T | T

T → T * F | F

F → (E) | id

The above grammar contains left recursive productions, so by eliminating left recursive, grammar G becomes:

E → TE'

E' → +TE' | ε

T → FT'

T' → *FT' | ε

F → (E) | id

Computation of FIRST set:

FIRST (E) = FIRST(T) ie: from production E → TE'

FIRST (T) = FIRST(F) ie: from production T → FT'

FIRST (F) = { **(**, id } ie: from production F → (E) | id

Therefore FIRST (E) = FIRST (T) = { **(, id** }

FIRST (E'') = { **+, ε** } ie: from production E' → +TE' | ε

FIRST (T'') = { ***, ε** } ie: from production T' → *FT' | ε

Computation of FOLLOW set:

FOLLW (E) = {), $ }

From F → (E), we have F is followed by a terminal symbol '('. Also

E is the start symbol, add $ in FOLLOW (E).

From E → TE'

FOLLOW (E'') = FOLLOW (E) = { **), $** }

From E' → +TE' | ε

FOLLOW (E'') = FOLLOW (E'')

Therefore **FOLLOW (E') = {), $ }**

From rule E → TE'

FOLLW (T) = FIRST (E'') – { ε } U FOLLOW (E) = { **+,), $** } ie: by applying 3rd rule, as β tends to ε when E'' derives ε

From rule E' → +TE' | ε

FOLLW (T) = FIRST (E'') – { ε } U FOLLOW (E'') = { **+,), $** }

Therefore **FOLLOW (T) = { +,), $ }**

From T → FT'

FOLLW (T'') = FOLLOW (T) = { **+,), $** }

From T' → *FT' | ε

FOLLW (T'') = FOLLOW (T'')

Therefore **FOLLOW (T') = { +,), $ }**

From T →FT'

FOLLOW (F) = FIRST (T'') – { ε } U FOLLOW (T) = **{ *, +,), $}** ie: by applying 3rd rule, as β tends to ε when T'' derives ε

From T' →*FT' | ε

FOLLOW (F) = FIRST (T'') – { ε } U FOLLOW (T'')

Therefore **FOLLOW (F) = { *, +,**

NOTE:

For any non-terminal, FOLLOW set is computed by selecting the productions in which, that non-terminal appears on RHS of production.

Non-terminal symbol	FIRST	FOLLOW
E	{ **(, id** }	{ **),** **$** }
T	{ **(, id** }	{ **+,** **), $** }
F	{ **(, id** }	{ ***, +,** **), $** }
E'	{ **+, ε** }	{ **),$** }
T''	{ ***, ε** }	{ **+,), $** }

Find FIRST and FOLLOW for the following grammar.

E → E + T | T

T → T * F | F

F → (E) | id

Non-terminal symbol	FIRST	FOLLOW
E	{ **(, id** }	{ **+,), $** }
T	{ **(, id** }	{ ***** , **+,** **), $** }
F	{ **(, id** }	{ ***, +,), $** }

FOLLOW (E) = {) } from F → (E) and FOLLOW (E) = { + } from E → E + T .

FOLLOW (T) = {*} from T → T * F and FOLLOW (T) = FOLLOW{E} from E → E + T | T

FOLLOW (F) = FOLLOW (T) from T → T * F | F

Find FIRST and FOLLOW for the following grammar.

Stmt_sequence → **S**tmt **S**tmt_seq'

Stmt_seq' → **;** **S**tmt_sequence | ε

Stmt → s

Non-terminal symbol	**FIRST**	**FOLLOW**
Stmt_sequence	{ **s** }	{ **$** }
Stmt_seq'	{ **;** **ε** }	{ **$** }
Stmt	{ **s** }	{ **;** **$** }

Find FIRST and FOLLOW for the following grammar.

S → ,GH;

G → aF

F → bF | ε

H → KL

K → m | ε

L → n | ε

Non-terminal symbol	**FIRST**	**FOLLOW**
S	{ , }	{ $ }
G	{ a }	{ m n **;** }
F	{ b ε }	{ m n **;** }
H	{ m n ε }	{ **;** }
K	{ m ε }	{ n ; }
L	{ **n** ε }	{ **;** }

FOLLOW (G) = FIRST(H**;**) – {ε} = { m n} and when FIRST(H) = **ε**, FIRST(H;) = **;**

That makes FOLLOW (G) = { m n ; }

FOLLOW (K) = FIRST(L) - {ε} U FOLLOW(H) = { n ; }since β tends to ε, so apply rule3.

Find FIRST and FOLLOW sets for the following grammar.

S → aABb

A → c | ε

B → d | ε

Non-terminal symbol	FIRST	FOLLOW
S	{ a }	$ }
A	{ c , ε }	{ d, b }
B	{ d, ε }	b }

FOLLOW(A) = FIRST(Bb) – {ε} **U** FIRST(b) ----------- when B derives ε, FIRST(Bb) = b

Find FIRST and FOLLOW sets for the following grammar.

S → AbS | e | ε

A → a | cAd

Non-terminal symbol	FIRST	FOLLOW
S	{ a, c, e, ε }	{ $ }
A	{ a, c }	{ b, d }

Find FIRST and FOLLOW sets for the following grammar.

Exp → Exp **addop** term | term

addop → + | -

term → term **mulop** factor | factor

mulop → *

factor → (Exp)

factor → number

OR

E → EAT | T

A → + | -

T → TMF | F

M → *

F → (E) | num

Non-terminal symbols are : EXP, addop, mulop, term, factor

Terminal symbols are : +, -, *, „(„„ „)‟ and „number‟

Non-terminal symbol	FIRST	FOLLOW
Exp	{ (, number }	{ +, -,), $ }
addop	{ +, - }	{ (, number }
term	{ (, number }	{ +, -,), $ }
mulop	{ * }	{ (, number }
factor	{ (, number }	{ +, -,), $ }

LL (1) GRAMMAR:

What is LL(1) grammar?

A grammar G is said to be LL(1) if and only if whenever a production $A \rightarrow \alpha \mid \beta$ are two distinct productions of G, with the following conditions hold:

Enlist the conditions required for a grammar to be LL(1).

- **i.** For no terminal symbol "**a**" do both α and β derive strings beginning with **a**
- **ii.** At most one of **α** and **β** can derive the empty string.
- **iii.** If $\beta \overset{*}{\Rightarrow} \varepsilon$ then α does not derive any string beginning with a terminal in FOLLOW(A). Likewise if $\alpha \overset{*}{\Rightarrow} \varepsilon$ then β does not derive any string beginning with a terminal in FOLLOW(A).

CONSTRUCTION OF PREDICTIVE PARSING TABLE (LL(1) Parser)

Input: Context free grammar G

Output: Predictive parsing table M.

Algorithm for predictive parsing table: (steps involved in construction of predictive parser):

For the production rule $A \rightarrow \alpha$ of grammar G

1. For each terminal symbol „**a**' in FIRST (α) create an entry in parsing table as M [A, a] = $A \rightarrow \alpha$.
2. For „**ε** „ in FIRST (α) create an entry in parsing table as M [A, b] = $A \rightarrow \alpha$. Where **'b'** is the terminal symbols from FOLLOW(A).
3. If „**ε** „ in FIRST (α) and „**$**" is in FOLLOW(A) then create an entry in the table M [A, $] = $A \rightarrow \alpha$.
4. All the remaining entries in the parsing table **M** are marked as „**SYNTAX ERROR'**.

NOTE:

For any grammar to be LL(1), each parsing table entry uniquely identifies a production or signals an error. That means there should not be any multiple entries in the parsing table**.**

Construct the predictive parsing table by making necessary changes to the grammar given below:

E → E + T | T

T → T * F | F

F → (E) | id

Also check whether the modified grammar is LL(1) grammar or not.

The above grammar contains left recursive productions, so we eliminate left recursive productions.

After eliminate left recursive productions, grammar G becomes:

E → TE’

E’ → +TE’ | ε

T → FT’

T’ → *FT’ | ε

F → (E) | id

By computing FIRST and FOLLOW sets of the above grammar:

Non-terminal symbol	FIRST	FOLLOW
E	{ **(, id** }	{ **), $** }
T	{ **(, id** }	{ **+,), $** }
F	{ **(, id** }	{ ***, +,), $** }
E’	{ **+, ε** }	{ **), $** }
T“	{ ***, ε** }	{ **+,), $** }

Construction of predictive parsing table:

Non-terminal		Input Symbol				
	id	+	*	(	)	$
E	E → TE''			E → TE''		
E’		E' →+TE'			E' → ε	E' → ε
T	T→ FT''			T→ FT''		
T’		T' → ε	T' →*FT'		T' → ε	T' → ε
F	F→ id			F→ (E)		

The above modified grammar is LL(1) grammar, since the parsing table entry uniquely identifies a production or signals an error

Construct the LL(1) parsing table for the grammar given below:

E → E * T | T

T → id + T | id

After eliminating left recursive production,

E → T E’

E’ → *T E’

T → id + T | id

After left factoring:

E → T E’

E’ → *T E’ |ε

T → id T’

T’ → +T | ε

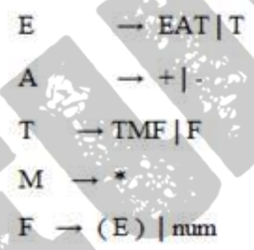
By computing FIRST and FOLLOW sets:

Non-terminal symbol	FIRST	FOLLOW
E	{ **id** }	{ **$** }
E“	{ ***, ε** }	{ **$** }
T	{ **id** }	{ ***, $** }
T“	{ **+, ε** }	{ ***, $** }

Construction of predictive parsing table:

Non-terminal	id	+	*	$
E	E → TE“			
E’			E’ → *T E’	E’ → ε
T	T → id T’			
T’		T’→ +T	T’ → ε	T’ → ε

Consider the grammar given below:


E → EAT | T
A → + | -
T → TMF | F
M → *
F → (E) | num

Do necessary modifications and Construct the LL(1) parsing table for the resultant grammar .

By eliminating left recursive productions:

E → TE’

E’ → ATE’ | ε

A → + | -

T → FT’

T’ → **M**FT’ | ε

M → *

F → (E) | num

By computing FIRST and FOLLOW sets:

Non-terminal symbol	FIRST	FOLLOW
E	{ (, num }	{), $ }
E‟	{ +, -, ε }	{), $ }
T	{ (, num }	{ +, - ,), $ }
T‟	{ *, ε }	{ +, - ,), $ }
A	{ +, - }	{ (, num }
M	{ * }	{ (, num }
F	{ (, num }	{ *, +,- ,), $ }

Construction of predictive parsing table:

Non-terminal	Input Symbol						
	num	**+**	**-**	*****	**(**	**)**	**$**
E	E → TE‟				E → TE‟		
E’		E‟ →ATE‟	E‟ →ATE‟			E‟ → ε	E‟ → ε
T	T→ FT‟				T→ FT‟		
T’		T‟ → ε	T‟ → ε	T‟ →MFT‟		T‟ → ε	T‟ → ε
A		A → +	A → -				
M				M → *			
F	F→ num				F→ (E)		

Construct the LL(1) parsing table for the grammar given below:

S → AaAb | BbBa

A → ε

B → ε

Answer:

Non-terminal symbol	FIRST	FOLLOW
S	{ a, b }	{ **$** }
A	{ ε }	{ **a, b** }
B	{ ε }	{ **a, b** }

Parsing Table:

Non-terminal	a	b	$
E	S → AaAb	S → BbBa	
E’	A → ε	A → ε	
T	B→ ε	B → ε	

Construct the LL(1) parsing table for the grammar given below:

S → A

A → aB

B → bBC | f

C → g

Non-terminal symbol	FIRST
S	{ a }
A	{ a }
B	{ b, f }
C	{ g }

Note: Since the grammar is **ε**- free, FOLLOW sets are not required to be computed in order to enter the productions into the parsing table.

Parsing Table:

Non-terminal	a	b	f	g	d
S	S → A				
A	A → aB			A → d	
B		B → bBC	B → f		
C				C→ g	

Construct the LL(1) parsing table for the grammar given below:

S → aBDh

B → cC

C → bC | ε

D → EF

E → g | ε

F → f | ε

Non-terminal symbol	FIRST	FOLLOW
S	{ a }	{ $ }
B	{ c }	{ g, f, h }
C	{ b, ε }	{ g, f, h }
D	{ g, f, ε }	{ h }
E	{ g, ε }	{ f, h }
F	{ f, ε }	{ h }

Parsing Table:

NT	a	b	c	g	f	h	$
S	S → aBDh						
B			B → cC				
C		C→ bC		C→ ε	C→ ε	C→ ε	
D				D→ EF	D→ EF	D→ EF	
E				E→ g	E → ε	E→ ε	
F					F→ f	F→ ε	

NON RECURSIVE PREDICTIVE PARSING ALGORITHM:

Initially the parser is in a configuration with input string (token) "**w$**" in the input buffer and the start symbol "**S**" of grammar **G**, on the top of the stack , **ie**: above the "$" symbol.

Steps involved in parsing the input token (string):

Let "X" be the symbol on the top of the stack, and "a", be the next symbol of the input string.

i. If X = a = $, then parser announces the successful completion of the parsing and halts.

ii. If X = a ≠ $, then parser pops the X off the stack and advances the input pointer to the next input symbol.

iii. If X is a non-terminal, then the program consults the parsing table entry TABLE [X, a]. If TABLE[X, a] = X→ UVW, then the parser replaces X on the top of the stack by UVW in such a way that U will come on the top. If TABLE [X, a] = error, then the parser calls the error recovery routine.

Given the grammar:

$S \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

i. Make necessary changes to make it suitable for LL(1) parsing.

ii. Construct FIRST and FOLLOW sets.

iii. Construct the predictive parsing table.

iv. Check whether the resultant grammar is LL(1) or not.

v. Show the moves made by the predictive parser on the input id + id * id.

NOTE:

In order to construct any predictive parser, First thing we have to

a. Eliminate left recursive productions if any

b. Left factoring, if possible.

i. After eliminating left recursive productions:

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \quad id$

i. FIRST & FOLLOW sets

Non-terminal symbol	FIRST	FOLLOW
E	{ (, id }	{), $ }
T	{ (, id }	{ +,), $ }
F	{ (, id }	{ *, +,), $ }
E'	{ +, ε }	{), $ }
T"	{ *, ε }	{ +,), $ }

iii. Construction of predictive parsing table:

Non-terminal	Input Symbol					
	id	+	*	(	)	$
E	E → TE"			E → TE"		
E'		E' →+TE'			E' → ε	E' → ε
T	T→ FT"			T→ FT"		
T'		T' → ε	T' →*FT'		T' → ε	T' → ε
F	F→ id			F→ (E)		

iv. The above modified grammar is LL(1) grammar, since the parsing table entry uniquely Identifies a production or signals an error.

v. Moves made by predictive parser on input id + id * id

MATCHED	STACK	INPUT	ACTION
	E$	id+ id * id$	
	TE"$	id+ id * id$	Output E → TE"
	FT"E"$	id+ id * id$	Output T → FT"
	idT"E"$	id+ id * id$	Output F → id
id	T"E"$	+ id * id $	match **id**
id	E"$	+ id * id $	Output T" → ε

id	+TE‟$	+ id * id $	Output E‟ →+TE‟
id +	TE‟$	id * id $	match +
id +	FT‟E‟$	id * id $	Output T→FT‟
id +	id T‟E‟$	id * id $	Output F → id
id + id	T‟E‟$	* id$	match **id**
id + id	*FT‟E‟$	* id$	Output T‟ → *FT‟
id + id*	FT‟E‟$	id$	match *
id + id*	id T‟E‟$	id$	Output F → id
id + id*id	T‟E‟$	$	match **id**
id + id*id	E‟$	$	Output T‟ → ε
id + id*id	$	$	Output E‟ → ε

Given the grammar:

S → (L) | a

L → L, S | S

i. Make necessary changes to make it suitable for LL(1) parsing.

ii. Construct FIRST and FOLLOW sets.

iii. Construct the predictive parsing table.

iv. Check whether the resultant grammar is LL(1) or not.

v. Show the moves made by the predictive parser on the input (a, (a, a))

Answer:

i. After eliminating left recursive productions, the grammar G becomes:

S → (L) | a

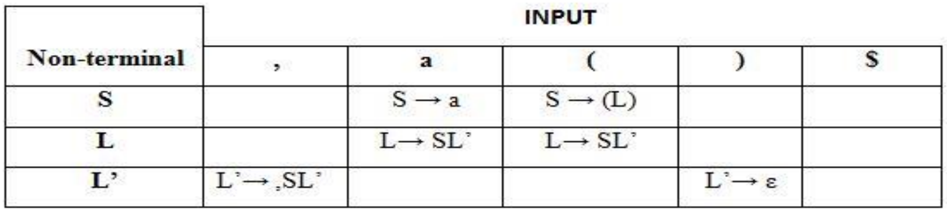
L → SL’

L’ → , SL’ | ε

ii. FIRST & FOLLOW sets

Non-terminal symbol	**FIRST**	**FOLLOW**
S	{ **(** **a** }	{ **,** **)** **$** }
L	{ **(** **a** }	{ **)** }
L’	{ **,** **ε** }	{ **)** }

iii. Predictive parsing Table:



Non-terminal	INPUT				
	,	a	(	)	$
S		S → a	S → (L)		
L		L→ SL'	L→ SL'		
L'	L'→ ,SL'			L'→ ε	

iv. The above modified grammar is LL(1) grammar, since the parsing table entry uniquely identifies a production or signals an error.

v. Moves made by predictive parser on input (a, (a,a))

MATCHED	STACK	INPUT	ACTION
	S$	(a,(a, a))$	
	(L)$	(a,(a, a))$	Output S → (L)
(	L)$	a,(a, a))$	match **(**
(	SL’)$	a,(a, a))$	Output L →SL’
(	aL’)$	a,(a, a))$	Output S→a
(a	L’)$	,(a, a))$	match **a**
(a	,SL’)$	,(a, a))$	Output L’→,SL’
(a,	SL’)$	(a, a))$	match **,**
(a,	(L)L“)$	(a, a))$	Output S → (L)
(a,(	L)L’)$	a, a))$	match **(**
(a,(	SL’)L’)$	a, a))$	Output L →SL’
(a,(	a L’)L’)$	a, a))$	Output S→a
(a,(a	L’)L’)$	, a))$	match **a**
(a,(a	,SL’)L’)$	, a))$	Output L’→,SL’
(a,(a,	SL’)L’)$	a))$	match **,**
(a,(a,	aL’)L’)$	a))$	Output S→a
(a,(a,a	L’)L’)$	))$	match **a**
(a,(a,a	)L’)$	))$	Output L’→ε
(a,(a,a)	L’)$	)$	match **)**
(a,(a,a)	)$	)$	Output L’→ε
(a,(a,a))	$	$	match **)**

Given the grammar:

S → aABb

A → c |ε

B → d |ε

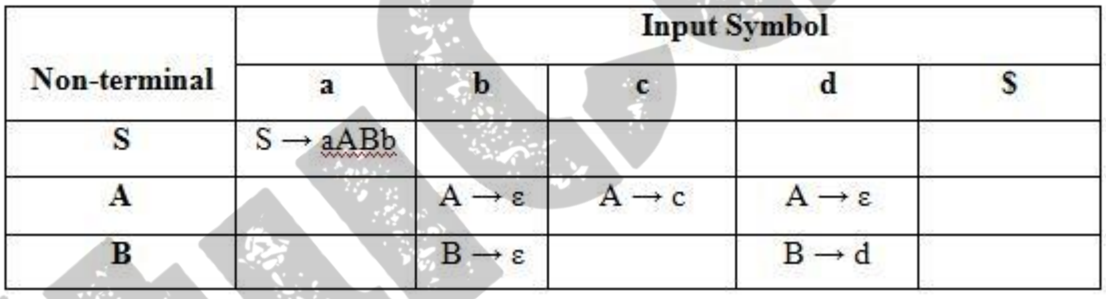
i. Construct FIRST and FOLLOW sets.

ii. Construct the predictive parsing table.

iii. Show the moves made by the predictive parser on the input acdb.

Non-terminal symbol	FIRST	FOLLOW
S	{ **a** }	{ **$** }
A	{ **c**, **ε** }	{ **d, b** }
B	{ **d**, ε}	{ b }

Parsing Table:



Non-terminal	Input Symbol				
	a	b	c	d	$
S	S → aABb				
A		A → ε	A → c	A → ε	
B		B → ε		B → d	

Moves made by predictive parser on input acdb

MATCHED	STACK	INPUT	ACTION
	S$	acdb$	
	aABb$	acdb$	Output S →aABb
a	ABb$	cdb$	match **a**
a	cBb$	cdb$	Output A→ c
ac	Bb$	db$	match **c**
ac	db$	db$	Output B→ d
acd	b$	b$	match **d**
acdb	$	$	match **b**

Write the parsing table for the grammar shown below:

S→ iEtSS' |a

S' → eS |ε

E→ b

Is this grammar LL(1) ? Justify your answer.

Non-terminal symbol	FIRST	FOLLOW
S	{ **i, a** }	{ **e, $** }
S'	{ **e, ε** }	{ **e, $** }
E	{ **b** }	{ t }

Non-terminal	**a**	**b**	**e**	**i**	**t**	**$**
S	**S → a**			**S → iEtSS'**		
S'			**S' → eS** **S' → ε**			**S' → ε**
E		**E→ b**				

The above parsing table contains two production rules for **M [S', e].** So the given grammar is not LL(1) grammar.

Here the grammar is ambiguous, and the ambiguity is manifested by a choice in what production to use when an **e** (else) is seen. We can resolve this ambiguity by choosing **S' → eS.**

SYNTAX ERROR HANDLING

Common Programming errors can occur at many different levels.

1. **Lexical errors**: include misspelling of identifiers, keywords, or operators.

2. Syntactic errors : include misplaced semicolons or extra or missing braces.

3. **Semantic errors**: include type mismatches between operators and operands.

4. **Logical errors:** can be anything from incorrect reasoning on the part of the programmer.

What should the parser do in an error case?

- The parser should be able to give an error message (as much as possible meaningful error message).
- It should be recovered from that error case, and it should be able to continue the parsing with rest of the input.

Error Recovery in Predictive Parsing

An error may occur in the predictive parsing (LL(1) parsing)

1. If the terminal symbol on the top of stack does not match with the current input symbol.
2. If the top of stack is a non-terminal A, the current input symbol is **a**, and the parsing table entry M [A, a] is empty.

ERROR-RECOVERY STRATEGIES

Explain error recovery strategies used during syntax analysis or by predictive parser.

The various error recovery strategies used during syntax analysis phase of a compiler (By predictive parser):

1. Panic-Mode Recovery.
2. Phrase-Level Recovery.
3. Error Productions.
4. Global Correction.

Panic-Mode Recovery

On discovering an error, the parser discards input symbols one at a time until one of a designated set of Synchronizing tokens is found. Synchronizing tokens are usually delimiters.

Example: semicolon or **}** whose role in the source program is clear and unambiguous.

It often skips a considerable amount of input without checking it for additional errors.

Advantage: Simplicity and it is guaranteed not to go into an infinite loop .

Phrase-Level Error Recovery

A parser may perform local correction on the remaining input. **i.e:** it may replace a prefix of the remaining input by some string that allows the parser to continue.

Each empty entry in the parsing table is filled with a pointer to a specific error routine to take care that error case.

Example: replace a comma by a semicolon, insert a missing semicolon etc.

- Local correction is left to the compiler designer.

- It is used in several error-repairing compliers, as it can correct any input string.
- Difficulty in coping with the situations in which the actual error has occurred before the point of detection.

Error Productions

- If we have a good idea of the common errors that might be encountered, we can augment the grammar with productions that generate erroneous constructs.
- When an error production is used by the parser, we can generate appropriate error diagnostics.
- Since it is almost impossible to know all the errors that can be made by the programmers, this method is not practical.

Global Correction

- Ideally, we would like a compiler to make as few changes as possible in processing incorrect inputs and we have to globally analyze the input to find the error.
- This is an expensive method, and it is not in practice.
- We use algorithms that perform minimal sequence of changes to obtain a globally least cost correction.
- It is too costly to implement in terms of time space, so these techniques only of theoretical interest.

PANIC-MODE ERROR RECOVERY IN LL (1) PARSING

In panic-mode error recovery, we skip all the input symbols until a synchronizing token is found.

What is the synchronizing token?

All the terminal-symbols in the follow set of a non-terminal can be used as a synchronizing token set for that non-terminal. So, a simple panic-mode error recovery for the LL(1) parsing:

- Place all symbols in FOLLOW (A) and FIRST(A) into the synchronizing set for non terminal A as **sync.**
- If the parser looks up entry is **synch**, then the non terminal on top of the stack is popped (except for **start** symbol) in an attempt to resume parsing.
- If the parser looks up entry M [A, a] and finds that it is blank, then the input symbol „**a**" is skipped.
- To handle unmatched terminal symbols, the parser pops that unmatched terminal symbol from the stack and it issues an error message saying that unmatched terminal was inserted

and continue parsing.

Panic-Mode Error Recovery – Example:

Explain panic mode error recovery techniques used for the following grammar:

S → AbS |e | ε

A→ a|cAd

Non-terminal symbol	FIRST	FOLLOW
S	{ **a, c, e, ε** }	{ **$** }
A	{ **a, c** }	{ **b, d** }

Construction of Predictive parsing table:

Non-terminal	Input Symbol					
	a	**b**	**c**	**d**	**e**	**$**
S	S →AbS		S →AbS		S → e	S → ε
A	A → a	**Sync**	A → cAd	**Sync**		

Moves made by predictive parser on input: ceadb

STACK	INPUT	REMARK
S$	ceadb$	
AbS$	ceadb$	
cAdbS$	ceadb$	
AdbS$	eadb$	
AdbS$	adb$	**Error**, Skip **e**
adbS$	adb$	
dbS$	db$	
bS$	b$	
S$	$	
$	$	

Explain how panic mode error recovery techniques used for the following grammar:

E → TE'

E' → +TE' | ε

T → FT'

T' → *FT' | ε

F → (E) | id

Consider the string:)id*+id

Non-terminal symbol	FIRST	FOLLOW
E	{ (, id }	{), $ }
T	{ (, id }	{ +,), $ }
F	{ (, id }	{ *, +,), $ }
E'	{ +, ε }	{), $ }
T''	{ *, ε }	{ +,), $ }

Construction of predictive parsing table:

Non-terminal	Input Symbol					
	id	+	*	(	)	$
E	E → TE''			E → TE''	Sync	Sync
E'		E' →+TE'			E' → ε	E' → ε
T	T→ FT''	Sync		T→ FT''	Sync	Sync
T'		T' → ε	T' →*FT'		T' → ε	T' → ε
F	F→ id	Sync	Sync	F→ (E)	Sync	Sync

Moves made by the predictive parser on the input-)id*+id

STACK	INPUT	REMARK
E$	)id*+id$	**Error**, skip **)**
E$	id*+id$	
TE‟$	id*+id$	
FT‟E‟$	id*+id$	
id T‟E‟$	id*+id$	
T‟E‟$	*+id$	
*FT’ E‟$	*+id$	
FT’ E‟$	+id$	**Error**, M [F, +] = **synch**
T’ E‟$	+id$	So **F** has been popped
E‟$	+id$	
+TE’$	+id$	
TE’$	id$	
FT‟E’$	id$	
idT‟E’$	id$	
T‟E’$	$	
E’$	$	
$	$	

Phrase-Level Error Recovery

- Each empty entry in the parsing table is filled with a pointer to a special error routine which will take cares that error case. These error routines may: Change, insert, or delete input symbols. It may issue appropriate error messages and pop items from the stack.

Note: We should be careful when we design these error routines, because we may put the parser into an infinite loop.

NOTE:

How to determine a Context free grammar is LL(1) or Not? without constructing parsing Table

1. For any CFG of the form:

$$A \rightarrow \boldsymbol{\alpha_1} \mid \boldsymbol{\alpha_2} \mid \boldsymbol{\alpha_3} \mid \ldots\ldots\ldots$$

If there is no **ε** in any of these **rules**, then find FIRST(**α_1**), FIRST(**α_2**), FIRST(**α_3**) and so on.

Take the intersection of these FIRST()^s pair-wise.

FIRST (α_1) ∩ FIRST (α_2) ∩ FIRST (α_3) ……………….. = Ø (No common terms)

Then the grammar is LL(1) grammar otherwise it is not LL(1)

[Find the pair-wise intersection of FIRST()]

2. For any CFG of the form:

$$A \rightarrow \boldsymbol{\alpha_1} \mid \boldsymbol{\alpha_2} \mid \boldsymbol{\alpha_3} \mid \ldots\ldots\ldots \mid \varepsilon$$

Then find the FIRST(**α_1**), FIRST(**α_2**), FIRST(**α_3**) and so on and also find the FOLLOW(A), for the rule ε.

Take the intersection of these FIRST()^s and FOLLOW(A) pair-wise.

FIRST (α_1) ∩ FIRST (α_2) ∩ FIRST (α_3) ……… ∩ FOLLOW(A) ……….. = Ø

Then the grammar is LL(1) grammar otherwise it is not LL(1).

Example:

Check whether the following grammar is LL(1) or not without constructing parsing table.

1. S → aSa | bS | c

Answer:

FIRST(**α_1**) = FIRST(aSa) = {a}

FIRST(**α_2**) = FIRST(bS) = {b}

FIRST(**α_3**) = FIRST(c) = {c}

FIRST (α_1) ∩ FIRST (α_2) ∩ FIRST (α_3) = {a}∩ {b}∩ {c} = **Ø**

Therefore the given grammar is LL(1) grammar

Check whether the following grammar is LL(1) or not without constructing parsing table

$S \rightarrow iCtSS_1 \mid bS \mid a$

$S_1 \rightarrow eS \mid \varepsilon$

$C \rightarrow b$

For **S** production rule:

FIRST(**α_1**) = FIRST ($iCtSS_1$) = {i}

FIRST(**α_2**) = FIRST(bS) = {b}

FIRST(**α_3**) = FIRST(a) = {a}

FIRST (α_1) ∩ FIRST (α_2) ∩ FIRST (α_3) = {i} ∩ {b}∩ {a} = **Ø**

For **S_1** production rule:

FIRST(**α_1**) = FIRST (eS) = {e}

Since the rule contains ε term we have to find FOLLOW(S_1)

FOLLOW(S_1) = FOLLOW(S) = FIRST(S_1) = {e}

FIRST (α_1) ∩ FOLLOW(S_1) = {e} which is not equal to Ø

Therefore the given grammar is not LL(1) grammar

(No need to check for C production rule)