Program : **B.Tech**

Subject Name: **Theory of Computation**

Subject Code: **CS-501**

Semester: **5th**

Unit III: Grammars:

3.1 Grammars:

Grammars denote syntactical rules for conversation in natural languages. Linguistics has attempted to define grammars since the inception of natural languages like English, Sanskrit, Mandarin, etc.
In formal language also grammar defines the rule or syntax or structure of the language.

Example: "Dog runs"

<Sentence> -> <Noun> <Verb>
< Noun > -> < Dog >
<Verb> -> <Run>

Formal Definition of Grammar:

A grammar G can be formally written as a 4-tuple (V, T, S, P) where –

- VN is a set of variables or non-terminal symbols.
- T or ∑ is a set of Terminal symbols.
- S is a special variable called the Start symbol, S ∈ V
- P is Production rules for Terminals and Non-terminals.

A production rule has the form $\alpha \rightarrow \beta$, where α and β are strings on VN ∪ ∑ and least one symbol of α belongs to VN.

Example:

Grammar G1 –
({S, A, B}, {a, b}, S, {S → AB, A → a, B → b})
Here:

S, A, and B are Non-terminal symbols;
a and b are Terminal symbols
S is the Start symbol, S ∈ N
Productions, P : S → AB, A → a, B → b

3.2 Types of Grammar (Chomsky Classification):

Type 0 Unrestricted/ Phase structured/ Recursively Enumerable
Type 1 Context Sensitive Grammar (CSG)
Type 2 Context Free Grammar (CFG)
Type 3 Regular Grammar

Type 0: Unrestricted/ Phase structured/ Recursively Enumerable:

- The productions have no restrictions.
- The productions can be in the form of α → β where α is a string of terminals and non-terminals with at least one non-terminal and α cannot be null.
- β is a string of terminals and non-terminals.

Type 1: Context Sensitive Grammar (CSG):

- First of all Type 1 grammar should be Type 0.
- The productions can be in the form of α → β & with restriction that |α| <= |β | that is count of symbol in Left side of the production:|α| is less than or equal to Right side: |β|

Type 2: Context Free Grammar (CFG):

- First of all it should be Type 1.
- The productions can be in the form of α → β & with restriction that Left hand side of production can have only one variable (Non-Terminal).| α | = 1.
- There is no restriction on β it can be (V U T)*.

Type 3: Regular Grammar:

- First of all it should be Type 2.
- It is most restricted form of grammar. It should be in the given form only :
  V –> VT* / T* or V –> T*V /T*
- It can of two types either left linear or right linear

3.2.1 Chomsky hierarchy and classification:

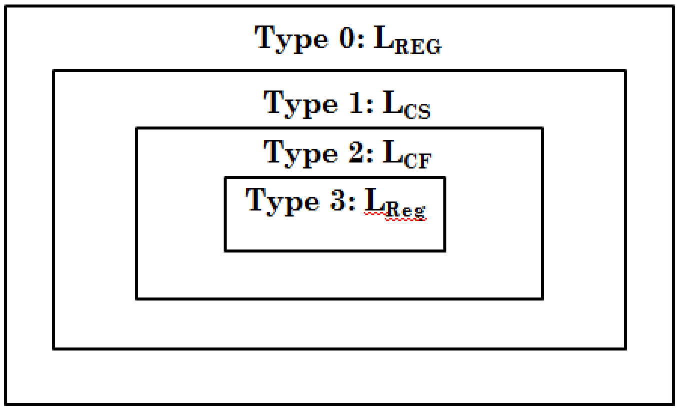


Fig: 3.1 Chomsky hierarchy.

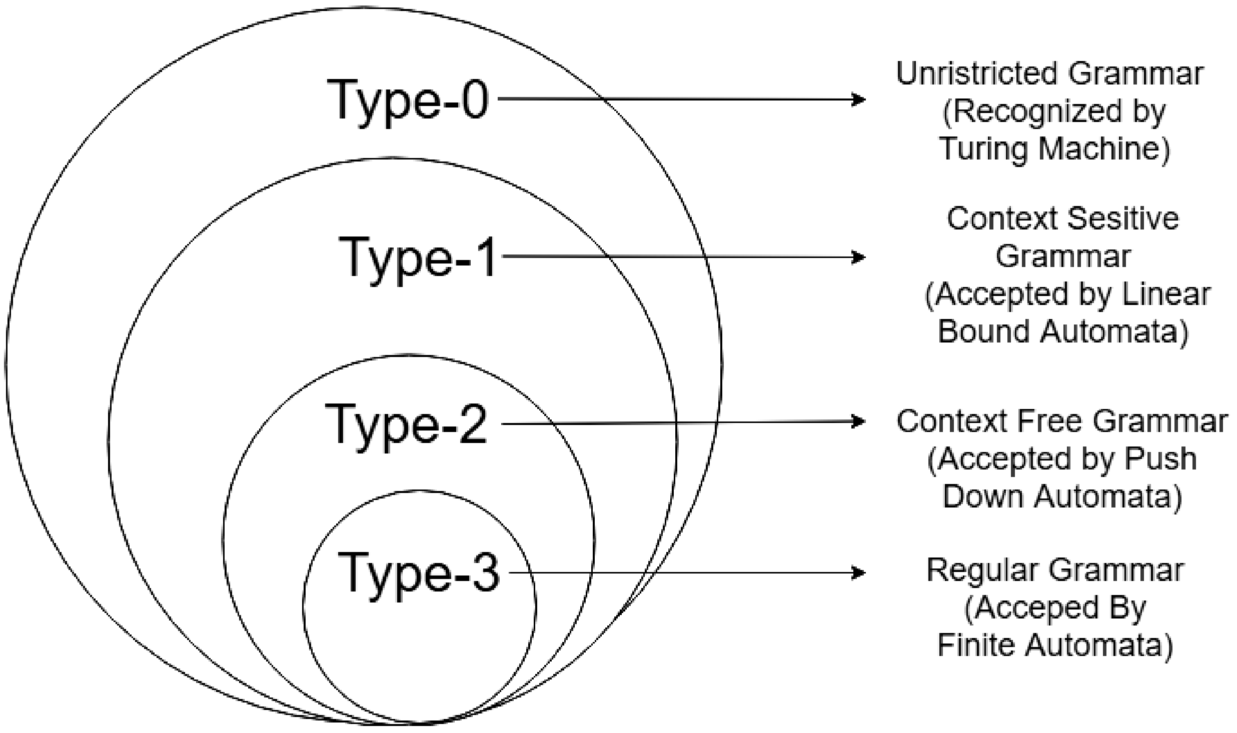

Type-0
Type-1
Type-2
Type-3
Unristricted Grammar (Recognized by Turing Machine)
Context Sesitive Grammar (Accepted by Linear Bound Automata)
Context Free Grammar (Accepted by Push Down Automata)
Regular Grammar (Acceped By Finite Automata)

Fig: 3.2 Chomsky Classifications

3.3 Derivation

To infer any string by replacing variable using productions of given CFG is known as derivation. Consider CFG G = (V, T, P , S) and αAβ be a string of terminals and variables with A is variable from V and α, β are string of terminals and variables. Lets there is production A → γ in grammar G then there will be derivation αAβ => α γ β.

When derivation begins from start symbol S and end in string of terminals T then these inferred string of terminal is language of Grammars G.

Example: Grammar: Arith. Exp
E ----> E + E
E ----> E * E
E ----> (E)
E ----> a | b | c
Here is a sequence of replacements that leads to a sequence of terminal symbols.

LMD (Left Most Derivative):

E ===> E * E ===> (E) * E ===> (E + E) * E ===> (a + E) * E ===> (a + b) * E ===> (a + b) * c

In this case, we obtained the final sentence (a + b) * c starting from E. One says that "E derives (a + b) * c" or (to use passive voice) "(a + b) * c is derived from E". The sequence is called a derivation sequence. In this case it is a leftmost derivation sequence, because at each stage the leftmost non-terminal was replaced. Each non-terminal that is replaced is colored red above. Sometimes there was only one non-terminal, so it is leftmost on an honorary basis.)

RMD (Right Most Derivative):

E ===> E * E ===> E * c ===> (E) * c ===> (E + E) * c ===> (E + b) * c ===> (a + b) * c
This was a rightmost derivation sequence, because at each stage the rightmost non-terminal was replaced.

Parse tree:

The sentence (a + b) * c has a unique leftmost derivation, a unique (different) rightmost derivation and the unique parse tree shown below:

```
Parse Tree: ( a + b ) * c
-----------------------------------
   E        |         E
  /|\       |        /|\
  E * E     |       / | \
 /|\  \     |      /  |  \
( E )  c    |     /   |   \
 /|\        |    /    |    \
E + E       |   E     |     E
|   |       |  /|\    |     |
a   b       | / | \   |     |
            |/  E  \  |     |
            / /|\ \ |       |
            | / | \ ||      |
            | E | E ||      |
            | | | | ||      |
            ( a + b )*      c
```

Fig 3.3 Parse Tree

3.4 Ambiguity:

There are other sentences derived from E above that have more than one parse tree, and corresponding left- and rightmost derivations. Example: the very simple sentence a + b * c. The table looks at leftmost derivations and parse trees:

```
1st Leftmost Der.   2nd Leftmost Der.

E ===> E + E        E ===> E * E
 ===> a + E          ===> E + E * E
 ===> a + E * E      ===> a + E * E
 ===> a + b * E      ===> a + b * E
 ===> a + b * c      ===> a + b * c

1st Parse Tree      2nd Parse Tree

    E                     E
   /|\                   /|\
  / | \                 / | \
  E + E                 E * E
  |  /|\               /|\  |
  | / | \             / | \ |
  a E * E             E + E c
    |   |             |   |
    b   c             a   b
```

Fig 3.4 Parse Tree showing Ambiguity

- Even if some parse trees are unique, if there are multiple parse trees for any sentence, then the grammar is called ambiguous.
- In a programming language it is not acceptable to have more than one possible reading of a construct. We can't flip a coin to decide which parse tree to use. There are several ways around this problem:
    1. Rewrite the grammar so that it is no longer ambiguous yet still accepts exactly the same language. This is not always possible.
    2. Introduce extra rules that allow the program to decide which of multiple parse trees to use. These are called disambiguating rules.
- An ambiguous grammar may signal problems with language design, and the programming language itself might be changed.

3.5 Simplification of CFG:

- As we have seen, various languages can efficiently be represented by a context-free grammar. All the grammars are not always optimized that means the grammar may consist of some extra symbols (non-terminal).
- Grammars having extra symbols, unnecessary increase the length of grammar.
- Simplification of grammar means reduction of grammar by removing useless symbols.

The properties of reduced grammar are given below:

- Each variable (i.e. non-terminal) and each terminal of G appears in the derivation of some word in L.
- There should not be any production as X → Y where X and Y are non-terminal.
- If ε is not in the language L then there need not to be the production X → ε.

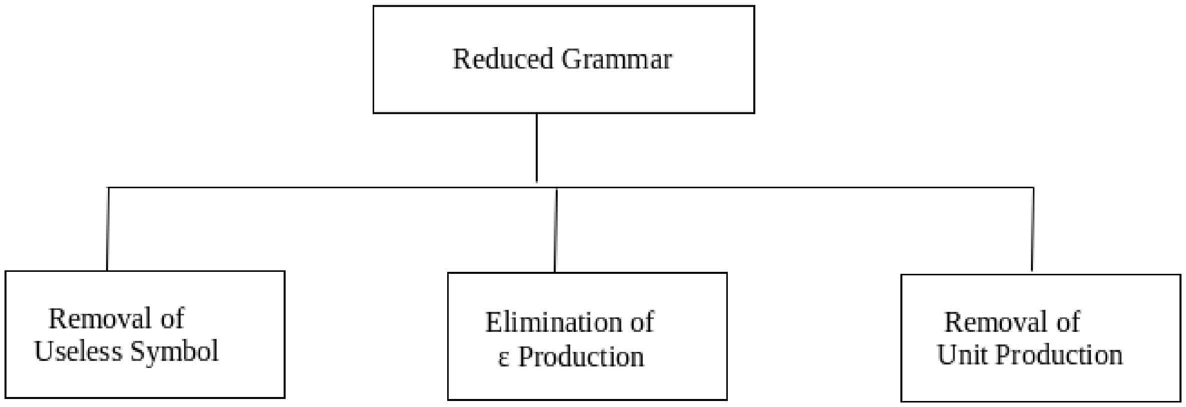

Reduced Grammar
Removal of Useless Symbol
Elimination of ε Production
Removal of Unit Production

Fig 3.5 Simplification (Reduction) of Grammar

3.5.1 Elimination of Useless production/symbols from context free grammar:
Conditions of Useless Symbol:
1) We will entitle any variable useful only when it is deriving any terminal.
2) Also if a symbol is deriving a terminal but not reachable from Start state.
3) All terminals will be useful symbols

Example: Remove Useless Productions/Symbols :

S -> AB/a
A -> BC/b
B -> aB/C
C -> aC/B

Solution:

- Useful Symbols: {a, b, S, A}

And any combination of useful symbols will also make LHS a useful symbol.
So we could see that Symbol B and C are useless symbol, remove them (whole production in which it contains):

S -> a
A -> b

Since A is not reachable so we will remove A -> b as well:

S -> a

One more example to remove useless:

S -> AB/AC
A -> aAb/bAa/a
B -> bbA/aaB/AB
C -> abCA/aDb
D -> bD/aC

Solution:

First find out useful Symbols: {a, b, A, B, S}
And useless symbols are: {C, D}
So remove them and write the whole grammar again:
S -> AB
A -> aAb/bAa/a
B -> bbA/aaB/AB

3.5.2 Elimination of null production from context free grammar

If ε belongs to the language then we are supposed to generate it and thus we will not remove it. Using below example we will understand the whole concept.

Example 1

S -> aSb/aAb/ab/a
A -> ε

- To know whether ε is generated in the CFG or not, we find all variable which are generating ε.
- So only A is genrating ε Thus ε does not belong to the language.

Now we will proceed with elimination of NULL production:

- Replace NULL producing symbol with and without in R.H.S. of remaining states and drop the productions which have ε directly. eg. A -> ε

S -> aSb/aAb/ab/ab/a   But we no need to write "ab" twice

So,

S -> aSb/aAb/ab/a

Example 2 Remove Null productions

S -> AB
A -> aAA/ε
B -> bBB/ε

Solution: Nullale Variables are {A, B, S}
Because start state also a Nullable symbol so ε belongs to given CFG
We will proceed with the method:

S -> AB/A/B/ε
A -> aAA/aA/a
B -> bAA/bA/b

3.5.3 Elimination of Unit production from context free grammar:

A Unit production is like below:

S -> B means V -> V that is a single Variable (non-terminal) producing another single Variable (non-terminal).

Steps to remove Unit production:

- Write production without Unit production
- Check what we are missing because of Step 1

Example:

Given grammar is :

S -> Aa/B/c
B -> A/bb
A -> a/bc/B

Remove unit Productions

Solution:

Now we will apply step 1:

S -> Aa/c
B -> bb
A -> a/bc

Now check what we are missing after applying Step 1:

First : S -> B -> bb
And : S -> B -> A -> a
And : S -> B -> A -> bc

So add these in the production list of "S"

S -> Aa/c/bb/a/bc
B -> bb
A -> a/bc

Second : B -> A -> a
And : B -> A -> bc

So add this in the prodcution list of "B"

S -> Aa/c/bb/a/bc
B -> bb/a/bc
A -> a/bc

Third: A-> B -> bb

So add this in the prodcution list of "A"

S -> Aa/c/bb/a/bc
B -> bb/a/bc
A -> a/bc/bb

One more example with 1 variation:

S -> AC
A -> a
C -> B/d
B -> D
D -> E
E -> b

Solution:

Now apply step 1:
S -> AC
A -> a
C -> b
Now check what are we missing after applying Step 1:
For C: C -> B -> D -> E -> b
So add this in the production of "C"
S -> AC
A -> a
C -> d/b
For B: B -> D -> E -> b
So add this in the production of "B"
S -> AC
A -> a
C -> d/b
B -> b
Similarily for D and E also add:
S -> AC
A -> a
C -> d/b
B -> b
D -> b
E -> b

But if we see that B -> b, D -> b and E -> b
Productions are useless as they can't be reached, Remove them:

S -> AC
A -> a

C -> d/b

3.6 Conversion of Grammar to Finite Automata:

Steps for converting grammar to Finite Automata are:

- Start from the first production
- And then for every left alphabet go to SYMBOL followed by it
- Start State: It will be the first production's state
- Final State: Take those states which end up with input alphabets. eg. State A and C are below CFG

Example: Here we are giving one right linear grammar

A -> aB/bA/b
B -> aC/bB
C -> aA/bC/a

now see the output and you will understand what just happened.

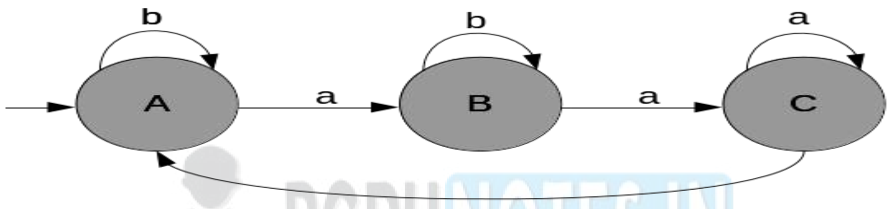


Fig 3.5 Grammar to automata

Explanation:

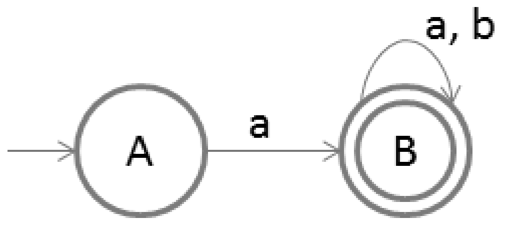
- As you can see for state A, transition on 'a' is going on state B and on 'b' it is going on state A itself. This method will apply to all the states.
- Note: Final states are A and C, cause they are having terminal production

3.7 Conversion of Finite Automata to Grammar:

Steps for converting finite automata to grammar:

- Repeat the process for every state
- Begin the process from start state
- Write the production as the output followed by the state on which the transition is going
- And at the last add ε because that's is required to end the derivation
- Note: We have added ε because either you could continue the derivation or would like to stop it. So to stop the derivation we have written ε

Example:



Solution: So we are applying the above procedure
Pick start state and output is on symbol 'a' we are going on state B
So we will write as :

A -> aB

And then we will pick state B and then we will go for each output.
so we will get the below production.

B -> aB/bB/ε

So final we got right linear grammar as:

A -> aB
B -> aB/bB/ε

3.8 Chomsky's Normal Form (CNF):

CNF stands for Chomsky normal form. A CFG(context free grammar) is in CNF(Chomsky normal form) if all production rules satisfy one of the following conditions:

- Start symbol generating ε. For example, A → ε.
- A non-terminal generating two non-terminals. For example, S → AB.
- A non-terminal generating a terminal. For example, S → a.

Steps for converting CFG into CNF:

Step 1: In the grammar, remove the null, unit and useless productions. You can refer to the Simplification of CFG.
Step 2: Eliminate terminals from the RHS of the production if they exist with other non-terminals or terminals. For example, production S → aA can be decomposed as:
S → RA
R → a
Step 3: Eliminate RHS with more than two non-terminals. For example, S → ASB can be

decomposed as:

S → RS

R → AS

Example: Convert given grammar into CNF

S -> bA/aB
A -> bAA/aS/a

Solution:

Now we have to add new production:
S -> NA/MB
A -> NAA/MS/a
N -> a
M -> b
Now combine NAA with either NA or AA, So
S -> NA/MB
A -> NO/MS/a
N -> a
M -> b
O -> AA

3.9 Greibach Normal Form (GNF):

GNF stands for Greibach normal form. A CFG(context free grammar) is in GNF (Greibach normal form) if all the production rules satisfy one of the following conditions:

- A start symbol generating ε. For example, S → ε.
- A non-terminal generating a terminal. For example, A → a.
- A non-terminal generating a terminal which is followed by any number of non-terminals. For example, S → aASB.

For example:

G1 = {S → aAB | aB, A → aA| a, B → bB | b}

G2 = {S → aAB | aB, A → aA | ε, B → bB | ε}

- The production rules of Grammar G1 satisfy the rules specified for GNF, so the grammar G1 is in GNF.
- However, the production rule of Grammar G2 does not satisfy the rules specified for GNF as A → ε and B → ε contains ε(only start symbol can generate ε). So the grammar G2 is not in GNF.

Steps for converting CFG into GNF:

Step 1: Convert the grammar into CNF.

If the given grammar is not in CNF, convert it into CNF. You can refer the following topic to convert the CFG into CNF: Chomsky normal form

Step 2: If the grammar exists left recursion, eliminate it.

If the context free grammar contains left recursion, eliminate it. You can refer the following topic to eliminate left recursion: Left Recursion

Step 3: In the grammar, convert the given production rule into GNF form.

If any production rule in the grammar is not in GNF form, convert it.

Example: Convert CFG to GNF

S → XB | AA
A → a | SA
B → b
X → a

Solution:

As the given grammar G is already in CNF and there is no left recursion, so we can skip step 1 and step 2 and directly go to step 3.

The production rule A → SA is not in GNF,
so we substitute S → XB | AA in the production rule A → SA as:

S → XB | AA
A → a | XBA | AAA
B → b
X → a

The production rule S → XB and B → XBA is not in GNF, so we substitute X → a in the production rule S → XB and B → XBA as:

S → aB | AA
A → a | aBA | AAA
B → b
X → a

Now we will remove left recursion (A → AAA), we get:

S → aB | AA
A → aC | aBAC
C → AAC | ε
B → b
X → a

Now we will remove null production C → ε, we get:

S → aB | AA
A → aC | aBAC | a | aBA
C → AAC | AA

B → b

X → a

The production rule S → AA is not in GNF, so we substitute A → aC | aBAC | a | aBA in production rule S → AA as:

S → aB | aCA | aBACA | aA | aBAA

A → aC | aBAC | a | aBA

C → AAC

C → aCA | aBACA | aA | aBAA

B → b

X → a

The production rule C → AAC is not in GNF, so we substitute A → aC | aBAC | a | aBA in production rule C → AAC as:

S → aB | aCA | aBACA | aA | aBAA

A → aC | aBAC | a | aBA

C → aCAC | aBACAC | aAC | aBAAC

C → aCA | aBACA | aA | aBAA

B → b

X → a

Hence, this is the GNF form for the grammar G.