Program : **B.Tech**

Subject Name: **Theory of Computation**

Subject Code: **IT-503**

Semester: **5th**

Department of Information Technology
Subject Notes
IT503 (A) - Theory of Computation
B.Tech, IT-5th Semester

Unit III

Syllabus : Introduction of Context-Free Grammar - derivation trees, ambiguity, simplification of CFGs, normal forms of CFGs- Chomsky Normal Form and Greibach Normal forms, pumping lemma for CFLs, decision algorithms for CFGs, designing CFGs, Closure properties of CFL's.

Unit Objective: Designing of CFG's , Construction of parse trees, finding and removing ambiguity in grammars, simplification of CFG, Conversion of grammar to Chomsky Normal Form , Greibach normal form.

Unit-III: Context Free Grammar

A context-free grammar (CFG) consisting of a finite set of grammar rules is a quadruple (N, T, P, S) where

- N is a set of non-terminal symbols.
- T is a set of terminals where N ∩ T = NULL.
- P is a set of rules, P: N → (N U T)*, i.e., the left-hand side of the production rule. P does have any right context or left context.
- S is the start symbol.

Example:

- The grammar ({A}, {a, b, c}, P, A), P: A → aA, A → abc
- The grammar ({S, a, b}, {a, b}, P, S), P: S → aSa, S → bSb, S → ε
- The grammar ({S, F}, {0, 1}, P, S), P: S → 00S | 11F, F → 00F | ε

Generation of Derivation Tree:

A derivation tree or parse tree is an ordered rooted tree that graphically represents the semantic information a string derived from a context-free grammar.

Representation Technique:

1. Root vertex: Must be labeled by the start symbol.

2. Vertex: Labeled by a non-terminal symbol.

3. Leaves: Labeled by a terminal symbol or ε.

If S → x1x2 …… xn is a production rule in a CFG, then the parse tree / derivation tree will be as follows:

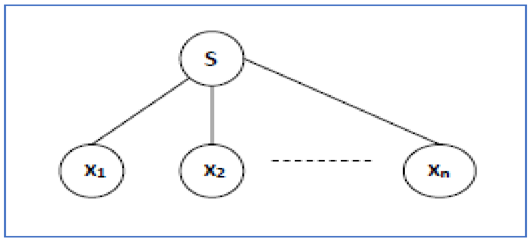

Figure 3.1: Derivation Tree

Top-Down Approach

- Starts with the starting symbol S
- Goes down to tree leaves using productions

Bottom-up Approach

- Starts from tree leaves
- Proceeds upward to the root which is the starting symbol S

Figure 3.2: Approaches of Derivation Tree

Derivation or Yield of a Tree:

The derivation or the yield of a parse tree is the final string obtained by concatenating the labels of the leaves of the tree from left to right, ignoring the Nulls. However, if all the leaves are Null, derivation is Null.

Example:

Let a CFG {N,T,P,S} be

N = {S}, T = {a, b}, Starting symbol = S, P = S → SS | aSb | ε

One derivation from the above CFG is "abaabb"

S → SS → aSbS →abS → abaSb → abaaSbb → abaabb

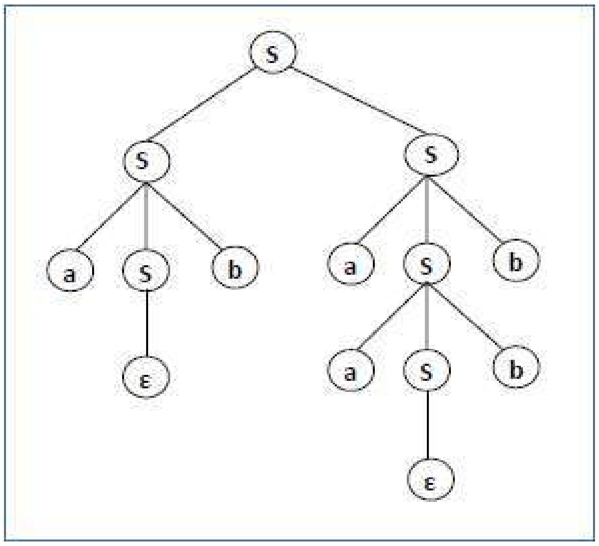

Figure 3.3: Example of Derivation Tree

Sentential Form and Partial Derivation Tree:

A partial derivation tree is a sub-tree of a derivation tree/parse tree such that either all of its children are in the sub-tree or none of them are in the sub-tree.

Example:

If in any CFG the productions are:

S → AB, A → aaA | ε, B →Bb| ε

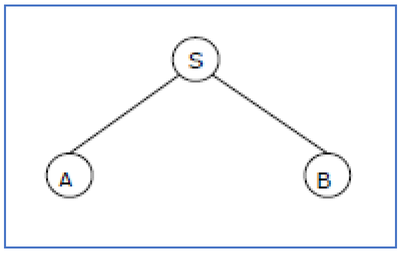

Figure 3.4: Example of Partial Derivation Tree

If a partial derivation tree contains the root S, it is called a sentential form. The above sub-tree is also in sentential form.

Leftmost and Rightmost Derivation of a String:

Leftmost Derivation: A leftmost derivation is obtained by applying production to the leftmost variable in each step.

Rightmost Derivation: A rightmost derivation is obtained by applying production to the rightmost variable in each step.

Ambiguity in Context Free Grammar:

If a context free grammar G has more than one derivation tree for some string w ∈ L(G), it is called an ambiguous grammar. There exist multiple right-most or left-most derivations for some string generated from that grammar.

Problem:

Check whether the grammar G with production rules:

X → X+X | X*X |X| a is ambiguous or not.

Solution:

Let's find out the derivation tree for the string "a+a*a". It has two leftmost derivations.

Derivation 1: X → X+X→ a +X→ a+ X*X →a+a*X→ a+a*a
Derivation 2: X → X*X→X+X*X→ a+ X*X →a+a*X→ a+a*a

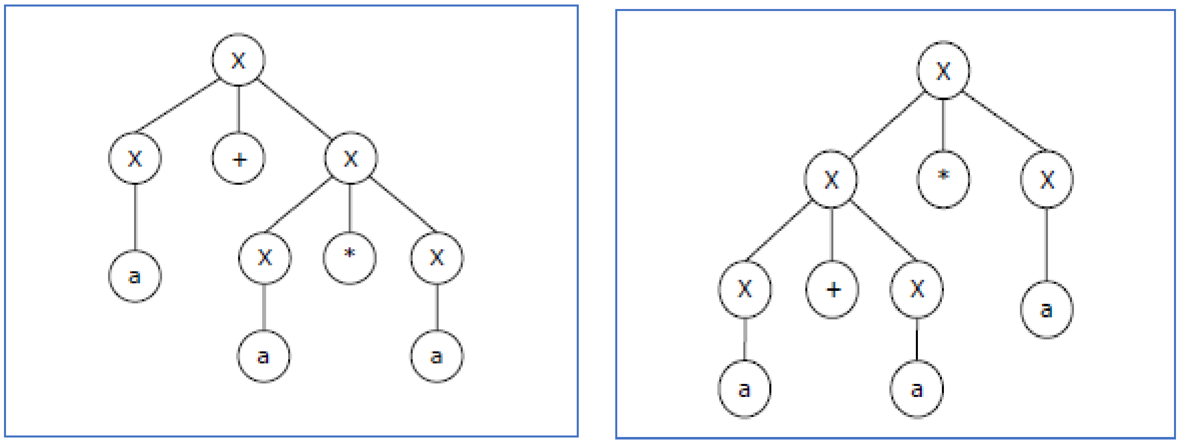

Figure 3.5: (a) Derivation 1 (b) Derivation 2

Since there are two parse trees for a single string "a+a*a", the grammar G is ambiguous.

Simplification of Context free Grammar:
In a CFG, it may happen that all the production rules and symbols are not needed for the derivation of strings. Besides, there may be some null productions and unit productions.

Elimination of these productions and symbols is called simplification of CFGs.

Simplification essentially comprises of the following steps:

- Removal of Epsilon/Null Productions
- Removal of Unit Productions
- Removal of Useless Productions

Removal of Epsilon/Null Productions:
Example: Remove the null productions from the following grammar
S -> ABAC
A -> aA / ϵ
B -> bB / ϵ
C -> c
Solution:
We have two null productions in the grammar A -> ϵ and B -> ϵ. To eliminate A -> ϵ we have to change the productions containing A in the right side. Those productions are S -> ABAC and A -> aA.
So replacing each occurrence of A by ϵ, we get four new productions
S -> ABC / BAC / BC
A -> a
Add these productions to the grammar and eliminate A -> ϵ.
S -> ABAC / ABC / BAC / BC
A -> aA / a
B -> bB / ϵ
C -> c
To eliminate B -> ϵ we have to change the productions containing B on the right side. Doing that we generate these new productions:
S -> AAC / AC / C
B -> b
Add these productions to the grammar and remove the production B -> ϵ from the grammar. The new grammar after removal of ϵ – productions is:
S -> ABAC / ABC / BAC / BC / AAC / AC / C
A -> aA / a
B -> bB / b
C -> c

Removal of Unit Production:
Example: Remove the unit productions from the following grammar
S -> AB
A -> a
B -> C / b
C -> D
D -> E
E -> a

Solution:
There are 3 unit production in the grammar
B -> C
C -> D
D -> E
For production D -> E there is E -> a so we add D -> a to the grammar and add D -> E from the grammar. Now we have C -> D so we add a production C -> a to the grammar and delete C -> D from the grammar. Similarly we have B -> C by adding B -> a and removing B -> C we get the final grammar free of unit production as:
S -> AB
A -> a
B -> a / b
C -> a
D -> a
E -> a
We can see that C, D and E are unreachable symbols so to get a completely reduced grammar we remove them from the CFG. The final CFG is :
S -> AB
A -> a
B -> a / b

Removal of Useless Production:
Example: Remove the useless symbol from the given context free grammar:
S -> aB / bX
A -> Bad / bSX / a
B -> aSB / bBX
X -> SBD / aBx / ad
Solution:
A and X directly derive string of terminals a and ad, hence they are useful. Since X is a useful symbol so S is also a useful symbol as S -> bX. But B does not derive any string w in V^*_t so clearly B is a non-generating symbol. So eliminating those productions with B in them we get
S -> bX
A -> bSX / a
X -> ad
In the reduced grammar A is a non-reachable symbol so we remove it and the final grammar after elimination of the useless symbols is
S -> bX
X -> ad

Normal Forms:
There are many normal forms for CFGs. They are the forms of context free grammar which are broad enough so that any grammar has an equivalent normal form version. You may think of it as cleaning up the grammar.

Chomsky Normal Form (CNF):
A CFG is in Chomsky Normal Form if the Productions are in the following forms:

A → a

A → BC

S → ϵ

where A, B, and C are non-terminals and a is terminal.

Algorithm to Convert into Chomsky Normal Form:

Step 1: If the start symbol S occurs on some right side, create a new start symbol S' and a new production S' → S.

Step 2: Remove Null productions.

Step 3: Remove unit productions.

Step 4: Replace each production A → B1…Bn where n > 2 with A → B1C where C → B2 …Bn. Repeat this step for all productions having two or more symbols in the right side.

Step 5: If the right side of any production is in the form A→ aB where a is a terminal and A, B are non-terminal, then the production is replaced by A→ XB and X→ a. Repeat this step for every production which is in the form A→ aB.

Example: Consider the CFG & convert it to CNF

S→ ASA | aB
A → B | S
B → b | ϵ

Solution:

(1) Since S appears in R.H.S, we add a new state S0 and S0→S is added to the production set and it becomes:

S0→S
S→ ASA | aB
A → B | S
B → b | ϵ

(2) Now we will remove the null productions:

S0→S
S→ ASA | a
A → S
B → b

(3) Now we will remove the unit productions.

S0→ASA | a
S→ ASA | SSA | AaA | SSa | Aaa |a
A → S | ASA | SSA | AaA | SSa | Aaa |a
B → b

(4) Now we will find out more than two variables in the R.H.S

Here, S0→ ASA, S → ASA, A→ ASA violates two Non-terminals in R.H.S.

Hence we will apply step 4 and step 5 to get the following final production set which is in CNF:

S0→AX | a
S→ AX | SX | AaA | SSa | Aaa |a
A → S | AX | SX | AaA | SSa | Aaa |a
B → b
X→ SA

Greibach Normal forms (GNF):
A CFG is in Greibach Normal Form if the Productions are in the following forms:

A → b

A → bD1…Dn

S → ϵ

where A, D1,....,Dn are non-terminals and b is a terminal.

Algorithm to Convert a CFG into Greibach Normal Form:

Step 1: If the start symbol S occurs on some right side, create a new start symbol S' and a new production S' → S.

Step 2: Remove Null productions.

Step 3: Remove unit productions.

Step 4: Remove all direct and indirect left-recursion.

Step 5: Do proper substitutions of productions to convert it into the proper form of GNF.

Example:

Convert the following CFG into CNF

S→ XY | Xo | p

X → mX | m

Y → Xn | o

Solution:

Here, S does not appear on the right side of any production and there are no unit or null productions in the production rule set. So, we can skip Step 1 to Step 3.

Step 4:

Now after replacing X in S → XY | Xo | p With X → mX | m

we obtain

S → mXY | mY | mXo | mo | p

And after replacing X in Y→ Xn | o with the right side of X → mX | m

we obtain

Y→ mXn | mn | o

Two new productions O→ o and N → n are added to the production set and then we came to the final GNF as the following:

$S \rightarrow mXY \mid mY \mid mXO \mid mO \mid p$

$X \rightarrow mX \mid m$

$Y \rightarrow mXN \mid mN \mid o$

$O \rightarrow o$

$N \rightarrow n$

Pumping Lemma for CGF:

Lemma:

If L is a context-free language, there is a pumping length p such that any string w ∈ L of length ≥ p can be written as w = uvxyz, where vy ≠ ε, |vxy| ≤ p, and for all i ≥ 0, uv^ixy^iz ∈ L.

Applications of Pumping Lemma:

Pumping lemma is used to check whether a grammar is context free or not. Let us take an example and show how it is checked.

Problem:

Find out whether the language L= $\{x^ny^nz^n \mid n \geq 1\}$ is context free or not.

Solution:

Let L is context free. Then, L must satisfy pumping lemma.

At first, choose a number n of the pumping lemma. Then, take z as $0^n1^n2^n$.

Break z into uvwxy, where

|vwx| ≤ n and vx ≠ ε.

Hence vwx cannot involve both 0s and 2s, since the last 0 and the first 2 are at least (n+1) positions apart. There are two cases:

Case 1: vwx has no 2s. Then vx has only 0s and 1s. Then uwy, which would have to be in L, has n 2s, but fewer than n 0s or 1s.

Case 2: vwx has no 0s.

Here contradiction occurs.

Hence, L is not a context-free language.

Closure properties of CFL's

Context free languages are accepted by Pushdown Automata but not by finite automata. Context free languages can be generated by context free grammar which has the form:

A -> ρ (where A ∈ N and ρ ∈ (T ∪ N)* and N is a non-terminal and T is a terminal)

Properties of Context Free Languages

Union: If L1 and If L2 are two context free languages, their union L1 ∪ L2 will also be context free. For example, L1 = { $a^n b^n c^m$ | m >= 0 and n >= 0 } and L2 = { $a^n b^m c^m$ | n >= 0 and m >= 0 }

L3 = L1 ∪ L2 = { $a^n b^n c^m \cup a^n b^m c^m$ | n >= 0, m >= 0 } is also context free.

L1 says number of a's should be equal to number of b's and L2 says number of b's should be equal to number of c's. Their union says either of two conditions to be true. So it is also context free language.

Concatenation: If L1 and If L2 are two context free languages, their concatenation L1.L2 will also be context free. For example,

L1 = { $a^n b^n$ | n >= 0 } and L2 = { $c^m d^m$ | m >= 0 }

L3 = L1.L2 = { $a^n b^n c^m d^m$ | m >= 0 and n >= 0} is also context free.

L1 says number of a's should be equal to number of b's and L2 says number of c's should be equal to number of d's. Their concatenation says first number of a's should be equal to number of b's, then number of c's should be equal to number of d's. So, we can create a PDA which will first push for a's, pop for b's, push for c's then pop for d's. So it can be accepted by pushdown automata, hence context free.

Kleene Closure: If L1 is context free, its Kleene closure L1* will also be context free. For example,

L1 = { $a^n b^n$ | n >= 0 }

L1* = { $a^n b^n$ | n >= 0 }* is also context free.

Intersection and complementation: If L1 and If L2 are two contexts free languages, their intersection L1 ∩ L2 need not be context free. For example,

L1 = { $a^n b^n c^m$ | n >= 0 and m >= 0 } and L2 = ($a^m b^n c^n$ | n >= 0 and m >= 0 }

L3 = L1 ∩ L2 = { $a^n b^n c^n$ | n >= 0 } need not be context free.

L1 says number of a's should be equal to number of b's and L2 says number of b's should be equal to number of c's. Their intersection says both conditions need to be true, but push down automata can compare only two. So it cannot be accepted by pushdown automata, hence not context free.

Similarly, complementation of context free language L1 which is ∑* – L1, need not be context free.

Deterministic Context-free Languages:

Deterministic CFL are subset of CFL which can be recognized by Deterministic PDA. Deterministic PDA has only one move from a given state and input symbol. For example, L1= { $a^n b^n c^m$ | m >= 0 and n >= 0} is a DCFL because for a's, we can push on stack and for b's we can pop. It can be recognized by Deterministic PDA. On the other hand, L3 = { $a^n b^n c^m \cup a^n b^m c^m$ | n >= 0, m >= 0 } cannot be recognized by DPDA because either number of a's and b's can be equal or either number of b's and c's can be equal. So, it can only be implemented by NPDA. Thus, it is CFL but not DCFL.

Note: Out of union, concatenation, complementation, intersection and kleene-closure, DCFL are closed only under complementation.

Example:

Consider the language L1,L2,L3 as given below.

L1 = { $a^m b^n$ | m, n >= 0 }

L2 = { $a^n b^n$ | n >= 0 }

L3 = { $a^n b^n c^n$ | n >= 0 }

Which of the following statements is NOT TRUE?

A. Push Down Automata (PDA) can be used to recognize L1 and L2

B. L1 is a regular language

C. All the three languages are context free

D. Turing machine can be used to recognize all the three languages

Solution:

Option (A) says PDA can be used to recognize L1 and L2. L1 contains all strings with any no. of a followed by any no. of b. So, it can be accepted by PDA. L2 contains strings with n no. of a's followed by n no. of b's. It can also be accepted by PDA. So, option (A) is correct.

Option (B) says that L1 is regular. It is true as regular expression for L1 is a*b*.

Option (C) says L1, L2 and L3 are context free. L3 languages contains all strings with n no. of a's followed by n no. of b's followed by n no. of c's. But it can't be accepted by PDA. So option (C) is not correct.

Option (D) is correct as Turing machine can be used to recognize all the three languages

Decision problems involving CFL's:

Based on the Pumping Lemma, we can state some decidability results for context-free languages.

Lemma: It is decidable whether or not a given string belongs to a context-free language.

Proof:

If we eliminate all ε-productions, and all unit productions from the context-free grammar for the language, then each time we apply a production rule we either:

- Make the sentential form grow longer or
- Increase the number of terminal symbols in the sentential form

Therefore the number of production rules applied (and thus the height of the parse tree) will be bounded by the length of the input string. Since this is finite, an effective (though inefficient) decision algorithm need only enumerate all such parse trees, and check the leaves to see if they correspond to the string.

Without examining the issue in detail, we also note the following:

- The constraints placed on the rule-format of **context-sensitive** grammars allow us to use the same argument as above to assert that language membership is decidable.
- In order to decide if a string is a member of a **free** language, we run the corresponding Turing Machine, and see if it halts on ``yes''. Clearly this is the **Halting Problem**, which is undecidable. Thus it is not possible in general to construct a parser for a free language.

Lemma: It is decidable whether or not a context-free language is empty.

Proof:

We need to show that the corresponding context-free grammar can generate a string.

If a CFG can generate a string, then it should be able to do so without using recursion (since we could just skip the recursion and generate a shorter string). If there are m non-terminals altogether, then some tree of height less than m must have leaves which are only terminals.

Thus we need only examine all such trees to see if the CFG generates a sentence.

Some decision problems for CFGs (and accordingly for push-down automata) are not solvable.

We state, without proof, that the following problems are undecidable in general:

- Given two context-free languages L_1 and L_2, is $L_1 \cap L_2 \equiv \emptyset$? i. e. do two context-free grammars generate the same string?
- Given some alphabet of terminal symbols Σ, does a grammar over this alphabet generate *all* the strings of Σ^*?
- Is the context-free grammar for a language ambiguous?

- Given two context-free languages L_1 and L_2, is $L_1 \subset L_2$?
- Given two context-free languages L_1 and L_2, is $L_1 = L_2$?