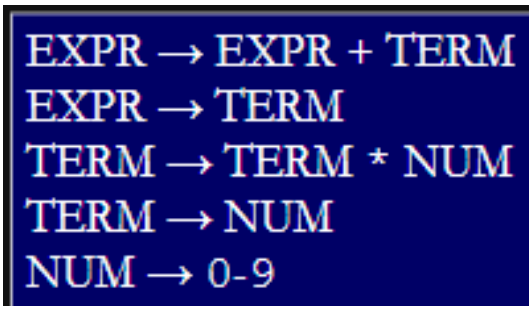
Assignment-9

TOPIC : Context-Free-Language

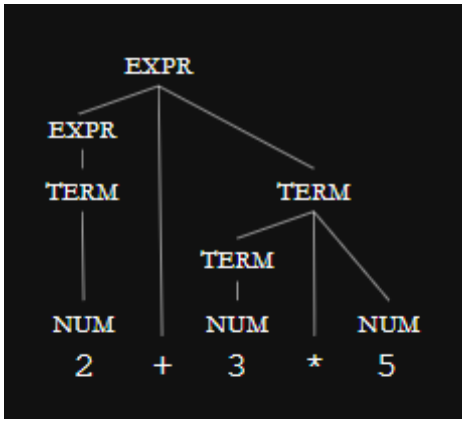
So far we have learnt that from regular expressions we generate strings and we can check from DFA and NFA. However, there are some restrictions that cannot be solved by regular expressions thus here comes CFG. In RE,DFA we cannot determine the starting point however, using CFG we can.

Suppose, we have 1^n0^n, if we write this in RE form we get 1*0*
10, 110,100,1100,1110,... so on. But is this what that notation means? No. We need to have an equal number of 1s and 0s in a string thus we can see some strings can not be generated by RE.

2+3*5 = 17 however why can't we write 25? That's because in maths there are certain precedents we follow for arithmetic calculations similarly in automata we have context free grammar which are rules that are used to define a language.

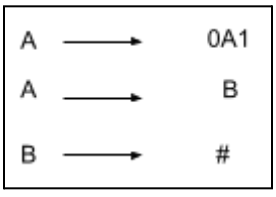


Here are 5 rules. We will apply these rules on the arithmetic problem that we just saw above and draw a parsing tree.



As you can see we started with rule 1 because rule-1 has a "+" sign and if we see the arithmetic problem it matches. Then we keep on drawing and using the rules whatever seems appropriate for each step to receive the final step.

English grammars are similar and using the grammar we get natural language. However at times people tend to not use the grammar but we still understand what it means. CFG is somewhat similar and easier to understand which creates Context free language.

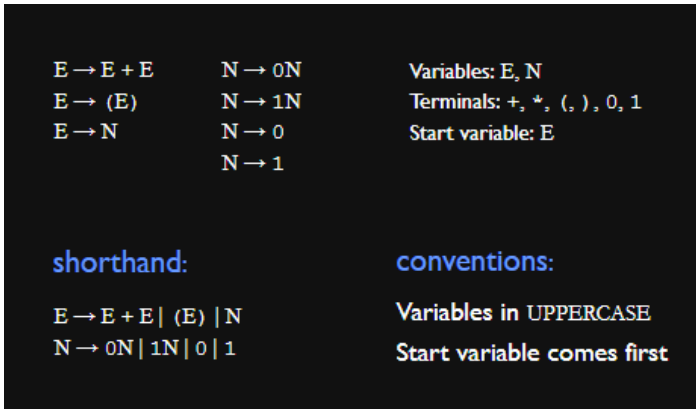


These are 3 rules that we need to follow in order to achieve the language. A and B are called variables as these are on the left side.

0,1 and # are called terminals because they are only found on the left side. Why don't we call 0A1 and B as terminals? That's because both have variables in it (Both contain components which are present on the left side) thus those cannot be terminals.

A-> 0A1 -> 00A11 ->000A111 we can break it down as long as we can, however it will not be accepted as a string of this language as it contains variable "A". So as per rule 2, we can change A->B so, 000B111 but can we call this as string? Not again as well. Because variable "B" is present and using rule 3 we can write it as 000#111. Now this is a string of the language because everything in this string are terminals. CFL1 = { 0,1,#}

- A context-free grammar is given by (V, Σ, R, S) where
  - V is a finite set of variables or non-terminals
  - Σ is a finite set of terminals
  - R is a set of productions or substitution rules of the form
  - A is a variable and α is a string of variables and terminals
  - S is a variable called the start variable

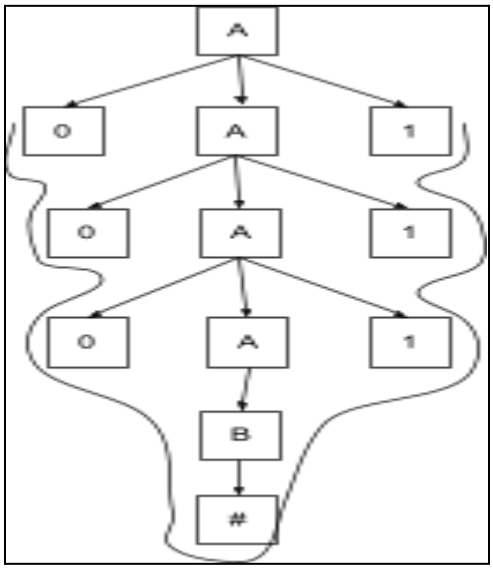


As given in the figure on the left, we can write the rules in two ways: left top and left bottom.

There are some conventions mentioned in the figure which we will need to follow at all times.

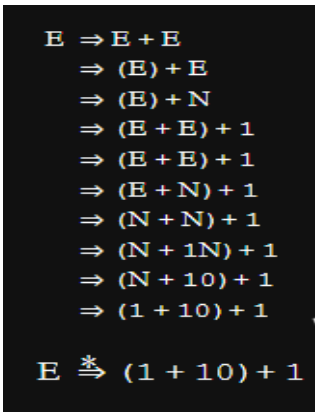
By seeing the rules we can identify the variables, terminals and start variables just as shown.

The **derivations** that we have seen so far suppose such as A-> 0A1 -> 00A11 ->000A111->000B111->000#111 is a form of derivation so we can say that it is a sequential application of productions.



This is a parse tree that we get from 000#111 from the derivation that we performed earlier. The leafs on the trees will be the **final string** we achieve after the derivation using the context grammar. The leaves of a tree are "**terminals**".

Here we are given a string that we have to reach from the grammars given to us.



First we will identify our start variable, terminals and variables. By using the grammar, we start with rule 1 and so on until we reach our string requirement which is (1+01)+1. Now how do we understand which rule to start with? The first variable is usually our starting variable. Moreover, it makes more sense to use rule 1, E = E + E because ultimately we need a "+" in our string which will only be possible using this rule. The figure on the left is a derivation by which we can obtain our final string. Now the machine will use ALL possible grammars until it finds a way to obtain what is asked for.

CONTEXT FREE LANGUAGE:
It is a set of all strings formed using context free grammar after each derivation.

Suppose we will be given some grammar and based on that we have to decide whether the strings given are acceptable or not.

For example- $A \rightarrow 0A1 \mid B$ $B \rightarrow \#$

Strings:
00#11 -> we can get this string by A->0A1->00A11->00B11->00#11 [Accepted]
-> Accepted because A->B->#.

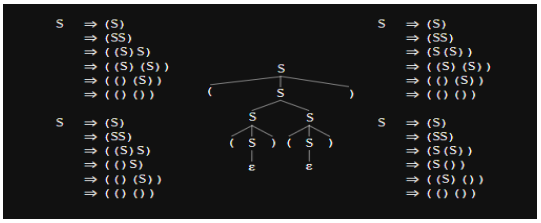
00#111 -> Not accepted because of uneven numbers of 0s and 1s. There should be an equal number of both 0&1.

After looking at the these strings we get an overview of what the language can be and that is 0^n # 1^n.

PARSE TREE

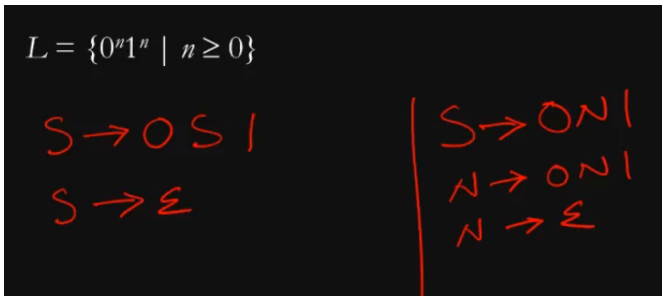
So far we have looked at how the parse tree looks and how using the context free grammar we make it up.

Parse trees are very much similar/same to the derivations that we write. However, there is only a difference between both. Let's have a look at it with a figure.



Here there are many different derivations which bring out the same final string; however, these all different derivations can be solely presented by one single parse tree.

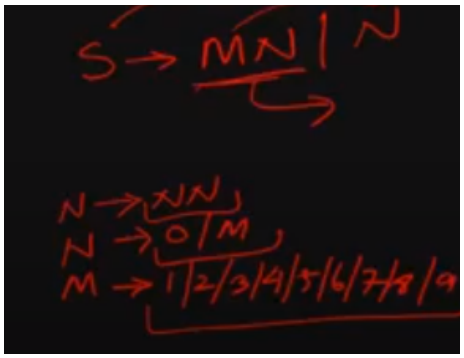
DESIGN EXAMPLE



Here we have drawn two different types of grammar. Now which one is the most appropriate? Now if we see this language has a condition that is n>= 0. Now for left hand side the smallest possible derivation will be Є and for the left side we get 01. Thus, if we follow the right hand side, and if n = 0, this is not possible to find the string because the smallest/minimum string that it can make is 01. Therefore, the left side will be the best option/correct answer. If n>0 then right side grammar would have been the best choice.

Example-2

Given, L= numbers without the leading 0s.
First we can use our previous knowledge and start with S-> SS.
S can be any number however we will find that this is a higher chance of getting 0s in the leading part. That's why we introduce S->MN. Where M is a 1 digit number except for 0. And N is any digit. However, S can also be two digits or single digit numbers. Therefore, S-> MN | N.
After much consideration we get context free grammar just like below.



Example-3

We will have a look into another example,
Suppose given a language, $L = \{0^n1^n0^m1^m \mid n \geq 0, m \geq 0\}$
Now if we divide 0^n1^n = N and 0^m1^m = M then we can start off with S->NM.
N->0N1 | Є , Why Є you might ask? Because as as defined n>= 0 and the smallest string we will find is Є. Same goes for M->0M1.
So finally we get

S->NM
N->0N1 | Є
M->0M1 | Є

Now we change the conditions to n>0, $m \geq 0$, here N must be present no matter what so,

S->0N1M
N->0N1 | Є

M->0M1 | Є

we can further break it up,

S->XY
X-> 0N1
N->0N1 | Є
Y-> 0M1
M->0M1 | Є

Example-4

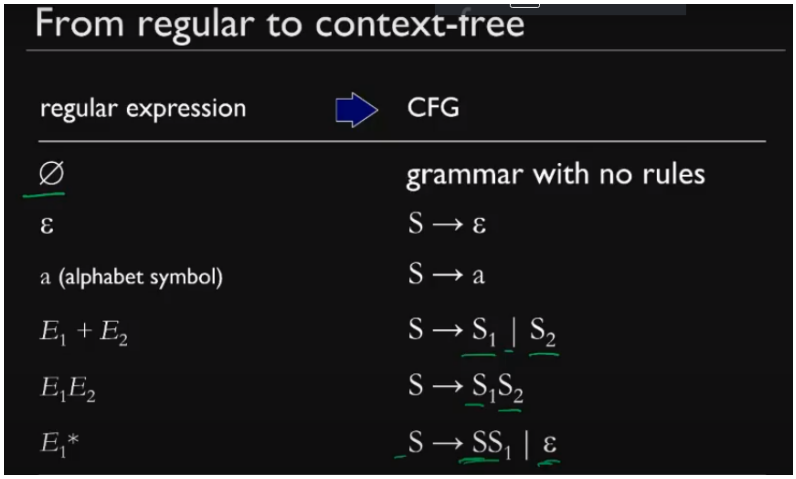
$L = \{0^n1^m0^m1^n \mid n \geq 0, m \geq 0\}$

We will also divide it here but in a different way. The outer 0^n and 1^n will be N and the inner 1^m and 0^m will be labelled as M.

S->N
N-> 0N1
N->M
M->1M0
M->Ɛ

Context free grammar and regular expressions

From regular expression we get regular language and from CFG we get CFL. We can also convert from regular expressions to CFG. Below are some of the rules that are used in converting from regular expression to CFG.



We have further looked into a simple example of converting a given regular expression into CFG.