Assignment-10

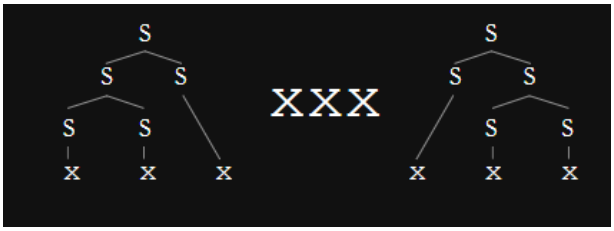
TOPIC : Ambiguity Parsing Algorithm

Previously we have learnt about how some expressions cannot be written by regular expression thus CFG was introduced which is much more powerful than regular expression. We have also seen how a single parse tree may represent many different types of derivation **but** only one single parse tree.
However, if such a case arrives in which a single derivation forms more than one parse tree then which one will be correct? This is called **Ambiguity.**
A CFG is called ambiguous when a derived string has more than one parse tree.
This creates a confusion in deciding which parse tree is correct for the given string.

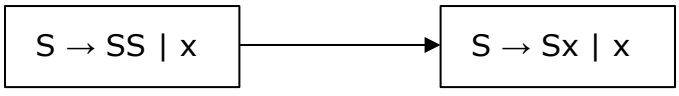
We will look further into how a CFG can be ambiguous and how to prevent it.

EXAMPLE-1

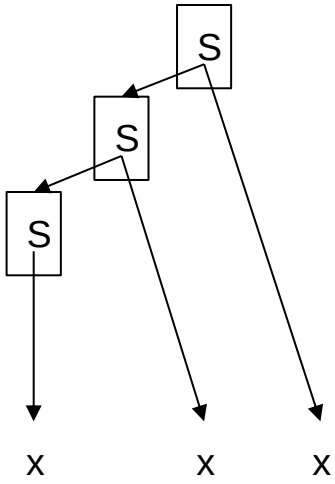
Given a grammar, S → SS | x and we check for a string "xxx"



If we draw the parse tree we get something like the figure on the left. Thus we can say that the given CFG is ambiguous. To remove the ambiguity we can rewrite the CFG in such a way.
For this case, we can change it just like given below.



Now if we draw the parse tree we check that the new parse tree matches with the left parse tree.



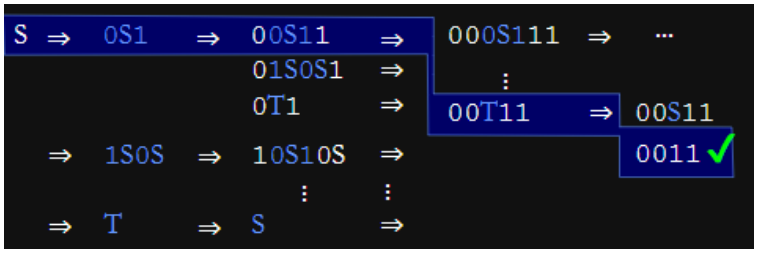
Parsing

One way of parsing is to try all possible derivations from each CFG. Suppose for the given CFGs

$$S \rightarrow 0S1 \mid 1S0S \mid T$$

$$T \rightarrow S \mid \varepsilon$$

We will now check all possible derivations from each CFG for the string 0011. We will get something like the following below.



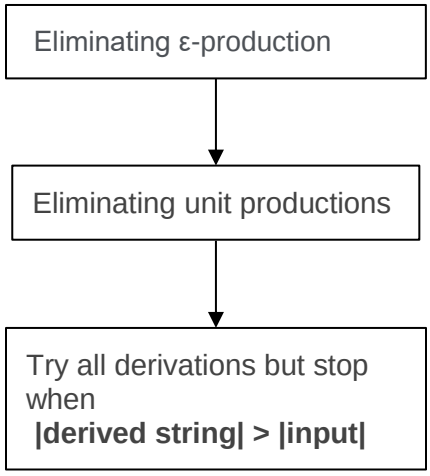
Here when a string is found after a following derivation, it will stop right there. However, if the string is not found therefore the parsing will not be stopped. It will continue like a loop. Just as T-> S then S->T again S->T it will go on and on until that string is found or else it will continue. Also, it will be too long for the process.

Idea

Stop parsing when **|derived string| > |input|**

Using the CFG as before, if we use the string 000111

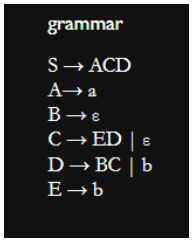
S-> 0S1->00S11->000S111 Here the length of the derived string till here is > than the input string. So we will stop.
However, the problem with this Idea is that ε- productions can **shrink** the strings. So this idea solely is not appropriate however we can use it with some removals in the process.



Eliminating ε- production
Step-1 : Find all null variables which means find the variables from which empty string can be derived.

Step-2: Remove the null variables one at a time such that other productions are not harmed in that way

Let us see an example of :



Here we have a grammar and we can identify that B,C are null variables. However D is also because if B and C derives an empty string then D will also. Thus, **null variables = B,C,D**.

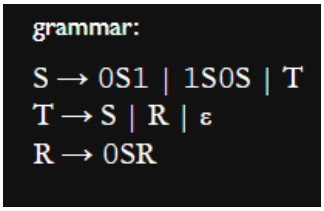
Now eliminate each of them.

Eliminating Unit production

A unit production is when a variable produces a single variable. Such as A-> B->C and so on.

If given A->B->C then we can directly write it like A->C.

Here we have a grammar but with unit productions. Let us see how to remove it.



We can see that from S we can derive T and from T->R. We can change it and directly write S->R. This is the same as with S. From S we can derive S itself by going through T. Thus,we can rewrite it to S->S.
Therefore, we can show it as,

$$S \rightarrow 0S1 \mid 1S0S$$
$$S \rightarrow R \mid \varepsilon$$
$$R \rightarrow 0SR$$

We still have unit production which is S->R now we can replace R with 0SR.

$$S \rightarrow 0S1 \mid 1S0S$$
$$\rightarrow 0SR \mid \varepsilon$$
$$R \rightarrow 0SR$$

We keep the last grammar as we would need later on for deriving the R itself but we eliminate **ε** so that the length of the strings does not shrink. Then we get our final grammar.

Example

We will now look at a problem and see how using the above processes we get the perfect grammar. Given,

$$S \rightarrow 0S1 \mid 0S0S \mid T$$

$$T \rightarrow S \mid 0$$

There is no need to eliminate ε. However we need to remove unit productions.

After eliminating,

$$S \rightarrow 0S1 \mid 0S0S \mid 0$$

Now we check for the input 00011,

S-> 0 X

S → 0S1
-> 00S11-> 00011
-> 001 X
-> 00S0S1 Too long

S->0S0S->00S0S0S Too long
->00S100S1 Too long
>0000 X

Here we can say that the string belongs to the language. And the checking stops when we find our desired derived string. If by any chance, this string was not found after derivation, then it would also have stopped because either it wouldn't match after reaching derivation or else the length of the derived string would be longer than the input string.