1. What is left recursive grammar? Give an example. What are the steps in removing left recursion?

What is left recursion

- A grammar is said to be left –recursive if it has a non-terminal A such that there is a derivation A -> Aα, for some string α
- A Production is immediately left recursive if its LHS and Head of RHS is the same symbol
    - Example B -> Bvt
- Indirect Left Recursion -> A grammar is said to possess indirect left recusion if it derives a string where the head is that symbol
    - Example A -> Br -> Csr -> Atsr
    - A indirectly gives Atsr, which is a left recursion
- Immediate left recursion is a special case of indirect left recursion
    - Derivation involved is a single step

Eliminating Immediate Left Recursion

- Example
    - A -> Aα | β
    - This can be replaced by non recursive productions
        - A -> βA'
        - A' -> αA' | ε
    - Where β does not start with A
- In General
    - If we have productions of the form
        - $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots\ldots\ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots\ldots\ldots \mid \beta_n$
    - Could be replaced with non recursive productions

- $$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots\ldots\ldots \mid \beta_n A'$$
  $$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots\ldots\ldots \mid \alpha_m A' \mid \epsilon$$
- where $\beta_1 \ldots\ldots\ldots \beta_n$ does not start with A

Example Problem

Consider the following left recursive grammar

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \mathbf{id} \end{aligned}$$

By the general rule we followed earlier we can transform it

$$A \rightarrow A\alpha \mid \beta \quad \text{=====>} \quad \begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \mid \epsilon \end{aligned}$$

Here A = E, α = +T, β = T

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid \epsilon \end{aligned}$$

Eliminating Indirect Left Recursion

Algorithm to eliminate Indirect Left Recursion

1. Arrange non-terminals in some order: $A_1, A_2 \ldots A_n$
2. for i = 1 to n do begin

    for j = 1 to i-1 do begin

        replace each production of the form $A_i \rightarrow A_j \gamma$

            by the productions $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \ldots\ldots\ldots\ldots \mid \delta_k \gamma$

            where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots\ldots\ldots\ldots \mid \delta_k$ are all the current A_j productions

    end

    eliminate the immediate left-recursions among the A_i productions

end

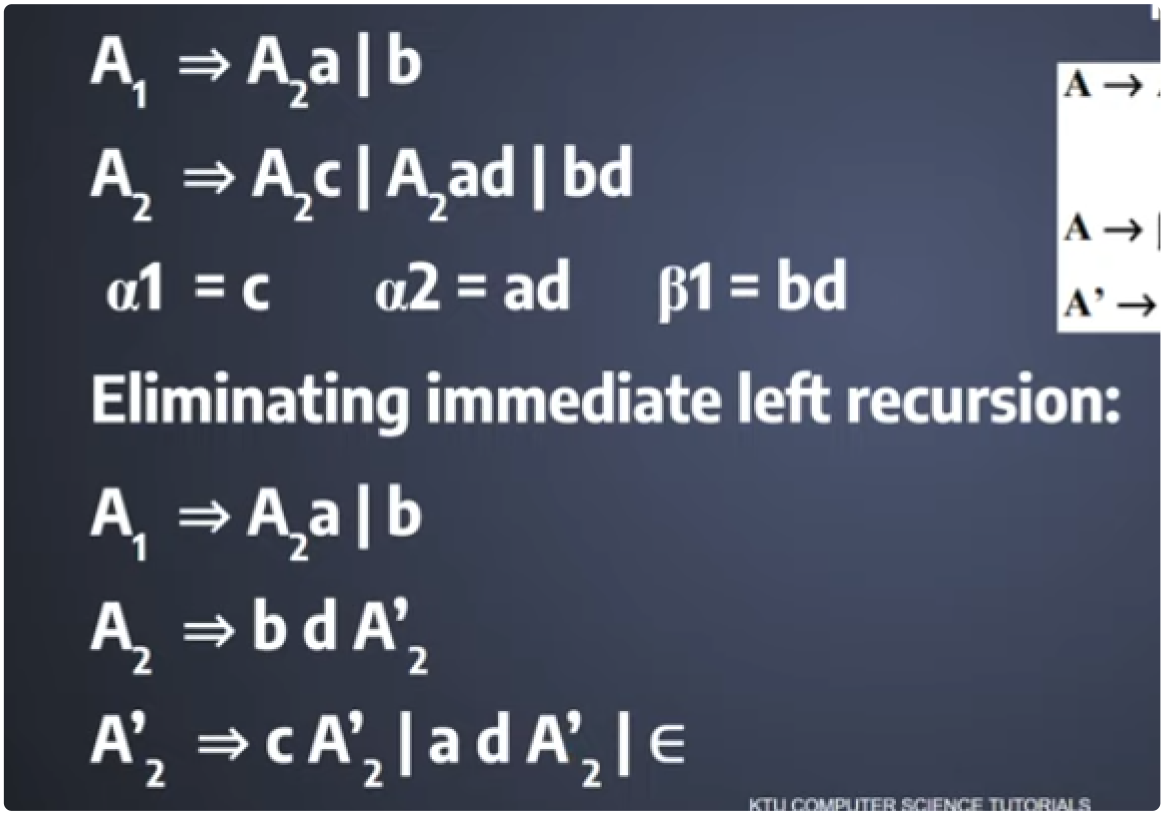
Consider this example

$$S \Rightarrow Aa \mid b$$

$$A \Rightarrow Ac \mid Sd$$

- First we Arrange the non terminals in some order
    - The non terminals here are S and A
    - S => A1, A => A2
    - There are 2 non terminals, set n=2
- $A_1 \Rightarrow A_2 a \mid b \quad A_2 \Rightarrow A_2 c \mid A_1 d$
- Loop 1
    - Loop from i=1 to 2 (2 is taken because n=2)
        - For j=1 to n i -1
            - Loop goes from 1 to 0, so nothing happens
- Loop 2
    - Case i=2 to 2
        - J=1 to 1

        - Replacing each production of form Ai -> Aj γ
            - Here i = 2, and j = 1
                - A2 -> A1d
                    - See what all alternatives does A1
                    - $A_1 \Rightarrow A_2a \mid b$
                    - The alternatives are
                        - $\delta_1 = A_2a$
$\delta_2 = b$
                - A2 -> A2ad | bd
                - So basically
                    - $A_2 \Rightarrow A_1d$
becomes
$A_2 \rightarrow A_2ad \mid bd$
                - We now get the Immediate Left Recursion
- Eliminating Immediate Left Recursion

- 

2. Compute First and follow for the following grammar

S -> ABCDE

A -> a | ε

C -> c

D -> d | ε

E -> e | ε

First

S -> ABCDE	FIRST(S) = {a,b,c}	FOLLOW(S) = {$}
A -> a | ε	FIRST(A) = {a,ε}	FOLLOW(A) = {b,c}
B-> b | ε	FIRST(B) = {b,ε}	FOLLOW(B) = {c}
C -> c	FIRST(C) = {c}	FOLLOW(C) = {d,e,$}
D -> d | ε	FIRST(D) = {d,ε}	FOLLOW(D) = {e,$}
E -> e | ε	FIRST(E) = {e,ε}	FOLLOW(E) = {$}

Explanation for first

- A -> a | ε
    - First we take the production
        - A -> a
        - a
    - Next we take production
        - A -> ε
        - ε
- S -> ABCDE
    - First we consider A
        - A derives a
            - Adding a to First(s)
        - A also derives ε
            - So we should consider next letter B
    - Considering B
        - B derives b
            - Adding b to First(s)
        - B derives ε
    - C
        - C derives c
            - Adding c to First(s)
-

Follow

Theres are 3 rules

- FOLLOW (S) = {$}, S is the start symbol
- If A -> αBβ then (if B is followed by beta then)
    - FOLLOW(B) = FIRST (β)
        - Include all symbols in the FIRST(beta) except epsilon
- If A -> αB or A -> αBβ where FIRST(β) contains ε
    - FOLLOW(B) = FOLLOW(A)

Explanation for Follow

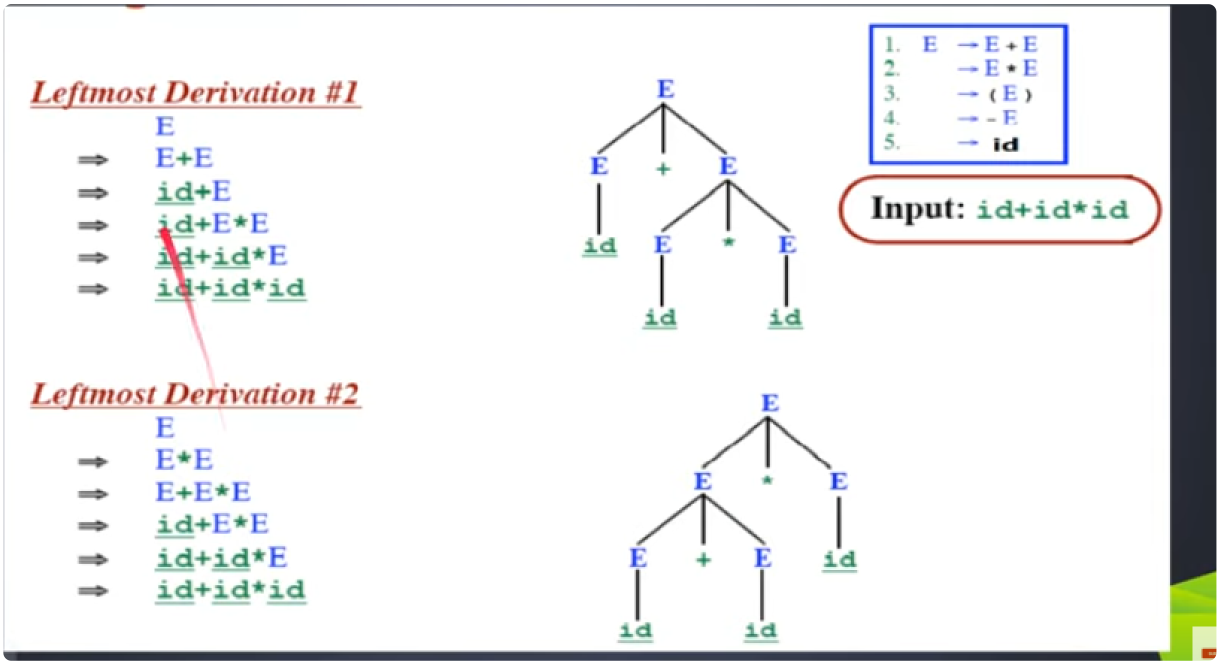
- FOLLOW(S)

    - As per the rule the follow is $
- FOLLOW(A)
    - What are the productions having A in the RHS, only one is there
        - S -> ABCDE
    - Now we need to find FOLLOW(A)
        - As per the rule
            - Set BCDE as β
        - Apply the rule
            - FOLLOW(B) = FIRST (β)
        - FOLLOW(A) = FIRST(BCDE)
            - FIRST(B) = b,ε
                - ε is there, so we need to find the FIRST of next letter, that is C
            - FIRST(C) = c
                - No ε, so stopping here
            - FOLLOW(A) = {b,c}
    - FOLLOW(B)
        - FOLLOW(B) = FIRST(C)
            - c
            - There is no epsilon, so stopping
        - FOLLOW(B) = {c}
    - FOLLOW(C)
        - FOLLOW(C) = FIRST(D)
            - d and epsilon
            - FIRST(D) = e,epsilon
            - SInce the production ends with epsilon, we need to include FOLLOW(S)=$
        - FOLLOW(C) = {d,e,$}
- SImilar steps for others

---

3. Check whether given grammar is ambigous or not

- A grammar is ambiguous if there are multiple parse trees for the same sentence
- Example

- 

---

4. Error recovery strategies in parsing (1.Panic mode, 2.Phrase level recovery, 3.Error production, 4. Global generation

Panic Mode

- Detects the error and discard until the synchronizing token is seen
- The Panic mode correction skips a lot of input without checking for additional errors

Phrase Level Recovery

- On discovering an error , a parser may perform local correction on the remaining input, it may replace a prefix of the remaining input by some string that allow the parser to continue
- Example
    - Replace , by a ;
    - Delete an extra ;
    - Insert a missing ;

Error Production

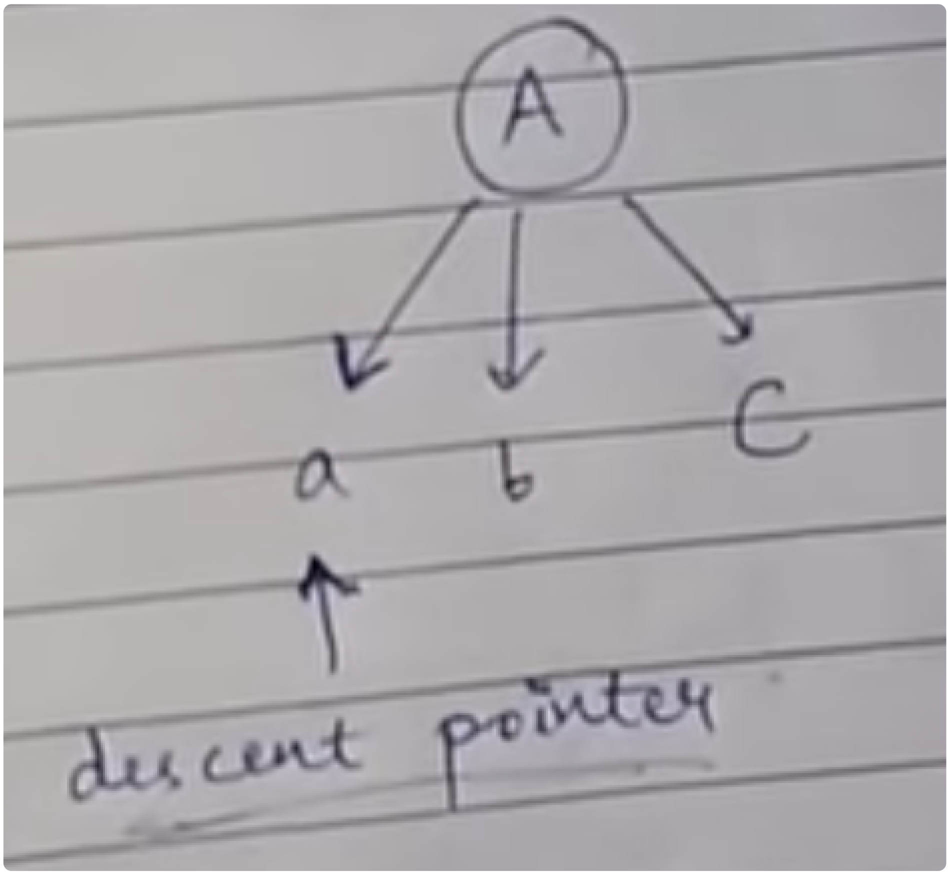
- If an error production is used by the parser, we can generate appropriate error diagnosis to indicate the rrror. Construct that has been recognized in the input

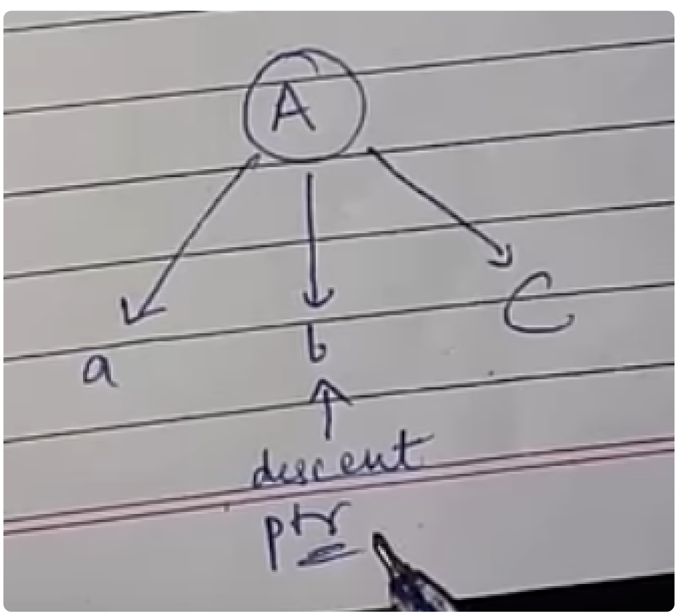
Global Correction

- In this method, the compiler itself makes a few changes as possible in processing an incorrect input string

5. Recursive descent parser with a suitable grammar. (with function)

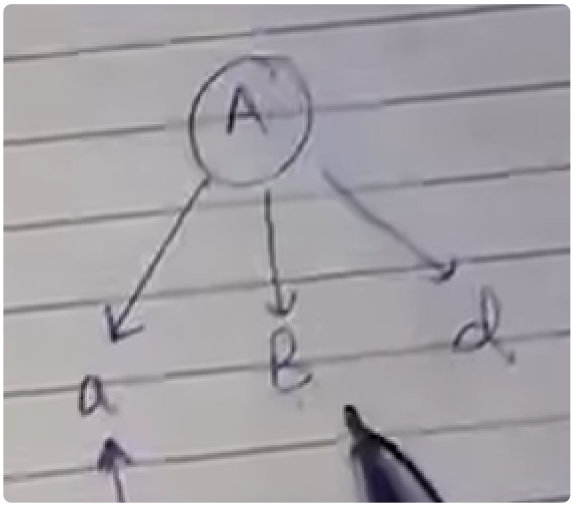
- Method used to find out whether given string can be formed with the help of given grammar
- A -> abC | aBd | aAD
- B -> bB | ε
- C -> d | ε
- D -> a | b | ε
- Taking the first production A -> abC
  - The Input given is a a b a
  - There will be 2 pointers
    - input pointer
    - descent pointer
  - Input
    - **a** a b a
  - Descent pointer

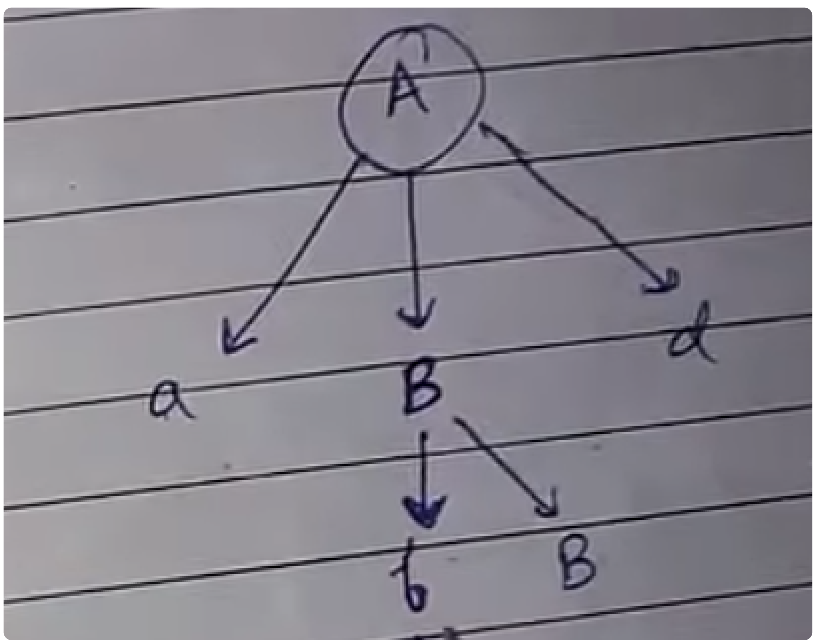
- 

- Value at input pointer and descent pointer are same
- Update input and descent pointer
- Input
    - a **a** b a
- Descent pointer
    - 

- a and b wont match, so we will backtrack

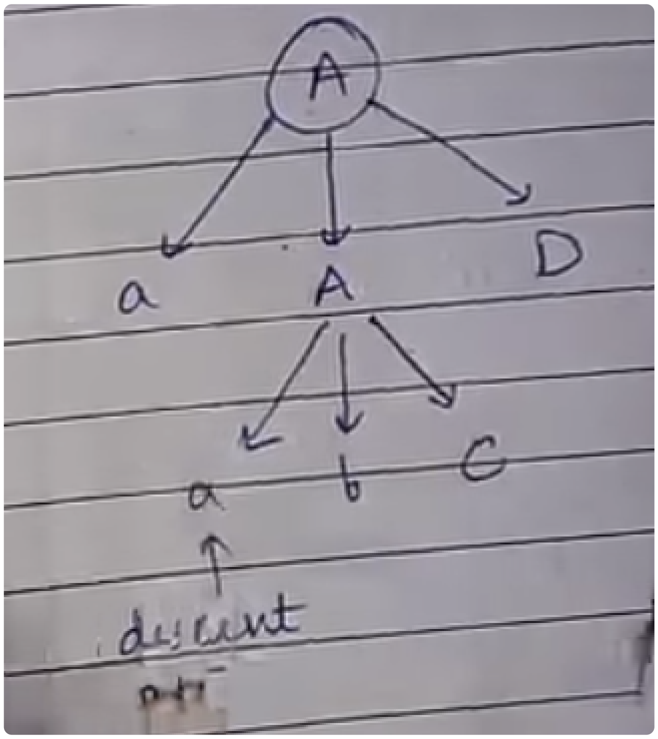
- Taking next production A -> aBd

- a **a** b a
- 

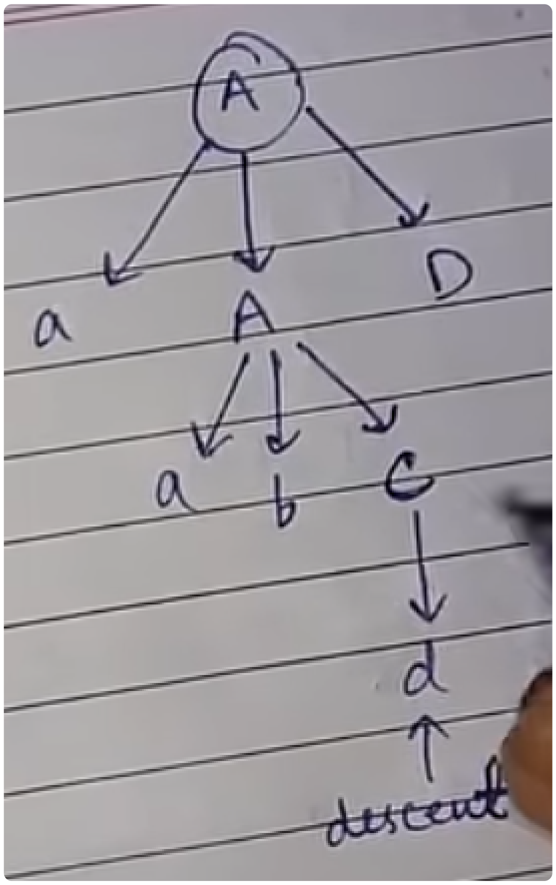
-
    - Value at input pointer and descent pointer are same
- Update input and descent pointer
- Descent pointer is pointing to a non terminal B
- 

- Derive the value of B
    - B -> bB
        - Value at input pointer (a) and descent pointer (b) are not same
    - B -> ε
        - This will lead the descent pointer to d
        - value at input pointer (a) and descent pointer (d) are not same
- Backtracking again

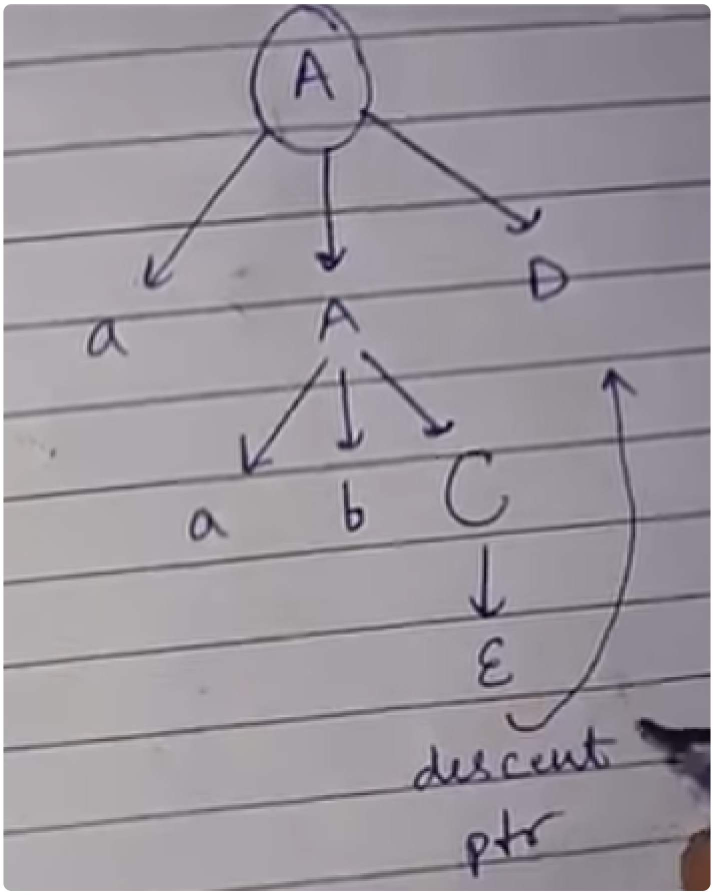
- Taking next production A -> aAD
    - **a** a b a

- 

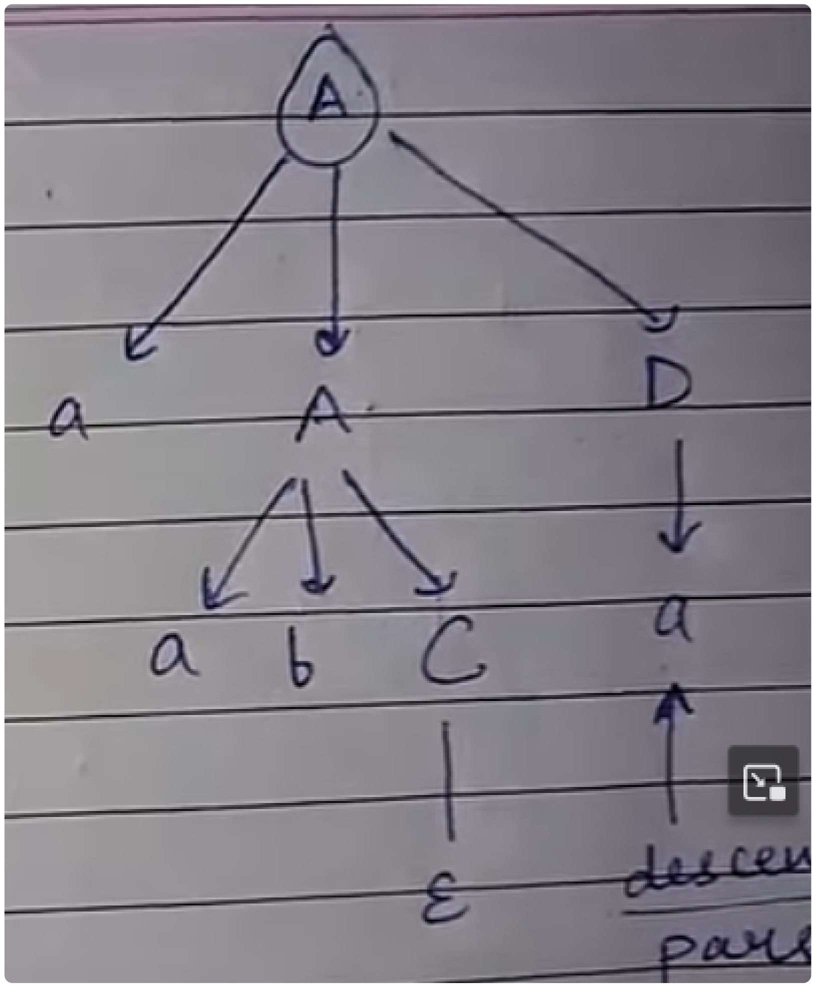
- Value at input pointer (a) and descent pointer (a) are same
- Incrementing the pointers
- a **a** b a
- Value at input pointer (a) and descent pointer (a) are same
- Incrementing the pointers
- ```
  Value at input pointer (a) and descent pointer (a) are same
  ```
- Value at input pointer (b) and descent pointer (b) are same
- Incrementing pointers
- a a b **a**
  ```

- 

- C is a non terminal, expanding, we get C -> d or C -> ε
- C ->d doesnt work
  - Input(a) and descent(d) doesnt match
- C -> ε

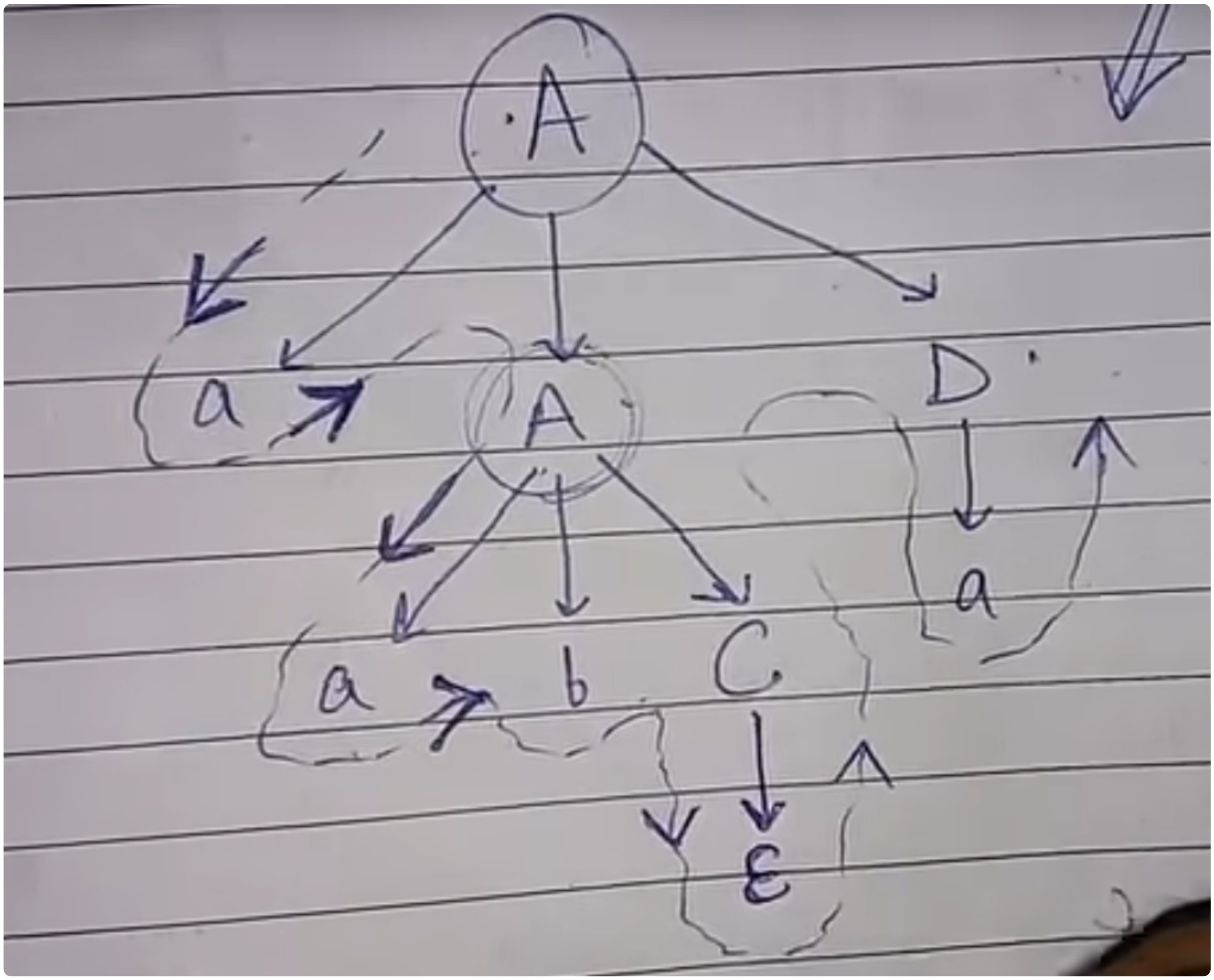
- 

- We come across another non terminal D
- Expanding it
  - D -> a | b | ε
  - Using D -> a

- 

- input and descent matches, since they are both a's

- a a b a string can be found using the parse tree as shown below



6. Non-Recursive Predictive parsering algorithm.

7. What is left factoring? Left factor the following grammar

8. Top-Down parser.

9. parsing table

- Non-recursive predictive parsing table (algorithm)
- LL(1) parsing table
- Predictive parsing table(¬ Eliminate left recursion, and ¬ Perform left factoring.)

10. Prove that the grammar is LL(1) or not.