Cairo University
Faculty of Computers & Artificial Intelligence
Theory of Computations

Lab#7

Context-free Grammar

Context-Free Grammar(CFG) :

A context-free grammar (CFG) is a set of recursive rewriting rules (or productions) used to generate patterns of strings.

Why Context-Free Grammar (CFG)?

Context-free grammars are strictly more powerful than regular expressions.

- Any language that can be generated using regular expressions and finite automata can be generated by a context-free grammar.
- There are languages that can be generated by a context-free grammar that cannot be generated by any regular expression and finite automata.

Example: Using regular expression or finite automata to express this language

$$\{0^n\ 1^n \text{ where } n>=1\}$$

Neither finite automata nor regular expression would successed in describing this language as These methods cannot count.

CFG

A context-free grammar (CFG) is a set of recursive rewriting rules (or productions) used to generate patterns of strings.
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 $N \cap T = NULL$.
- P is a set of rules, P: $N \rightarrow (N \cup T)^*$, 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:
Grammar G1:

$$A \rightarrow 0A1$$
$$A \rightarrow B$$
$$B \rightarrow \#$$

P: Grammar G1 contains three rules.
N: G1's non-terminal symbols are A and B,
S: A is the start variable.
T: Its terminals are 0, 1, and #.

The grammar uses a **parse tree** for generating pattern strings.

Practice:

1) A CFG describing strings of letters with the word "dog" somewhere in the string:

<program> → <word> dog <word>
<word> → <letter> <word> | epsilon
<letter> → A | B | ... | Z | a | b ... | z

2) A CFG describing strings of letters that the number of a's is equal to number of b's and, presence of a's followed by b's and the length of the string is greater than or equal 2.

S → a S b
S → ab

3) Construct a CFG for the regular expression (0+1)*

S → 0S | 1S
S → ε

4) Construct a CFG for a language L = {wcw^R | where w € (a, b)*}.
S → aSa
S → bSb
S → c

5) { w ∈ (0, 1)* | w contains at least three 1s }
S → X 1 X 1 X 1 X
X → 0 X | 1 X | ε

Generation of Derivation Tree or Parse tree

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

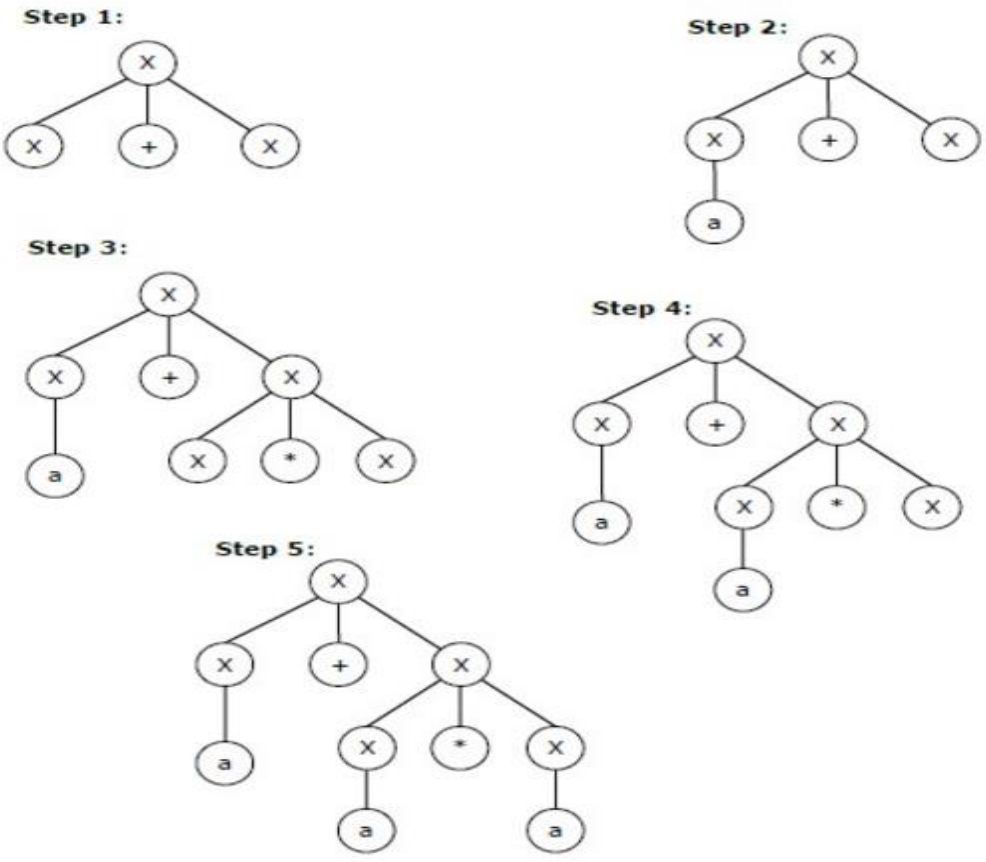
Tree nodes represent symbols of the grammar (nonterminals or terminals) and tree edges represent derivation steps.

- **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.

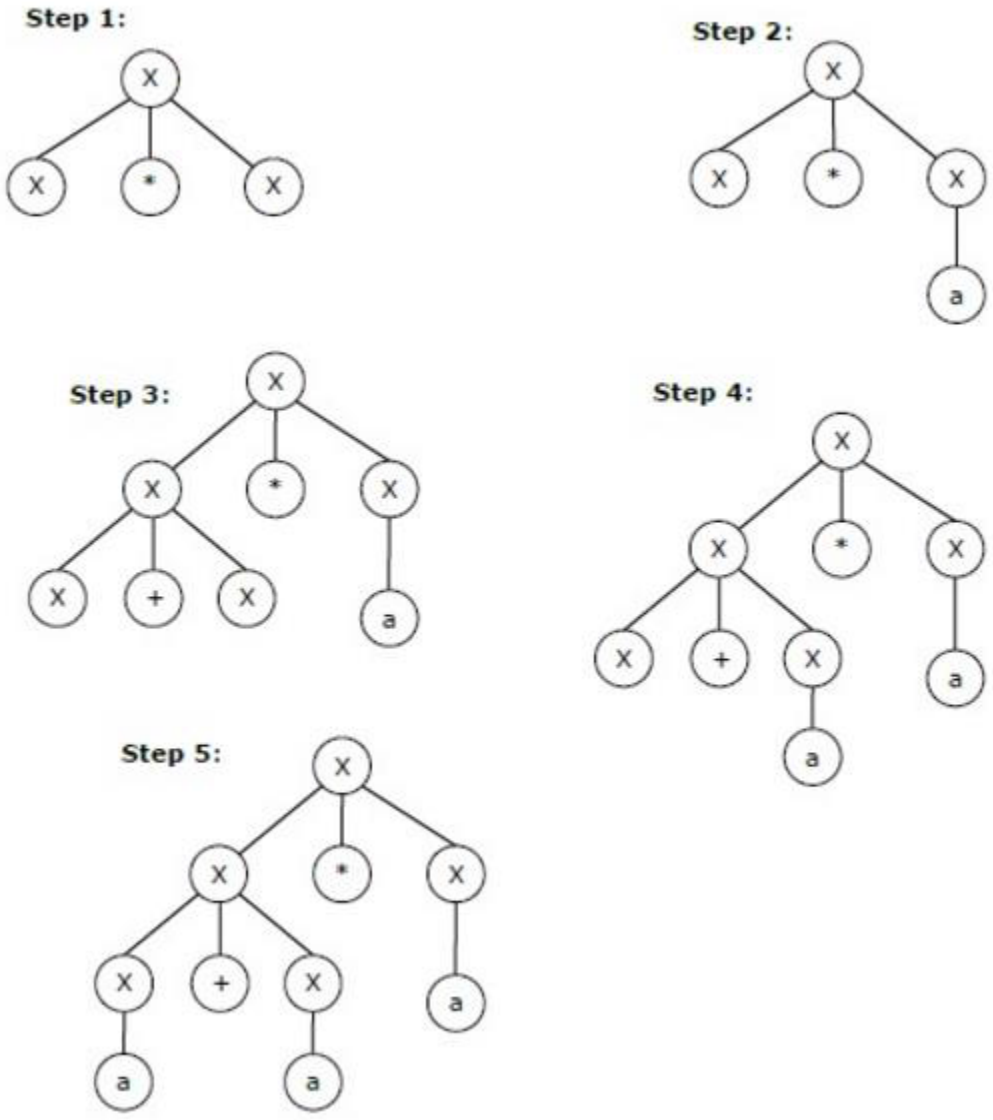
Example:

- Let any set of production rules in a CFG be **X → X+X | X * X | X | a** over an alphabet {a}. What is derivation tree over this string "**a+a*a**" ?

1) The leftmost derivation for the string "a+a*a" may be
   X → X+X → a+X → a + X*X → a+a*X → a+a*a



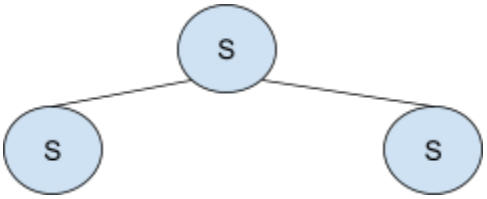
2) The rightmost derivation for the above string "a+a*a" may be
X → X*X → X*a → X+X*a → X+a*a → a+a*a



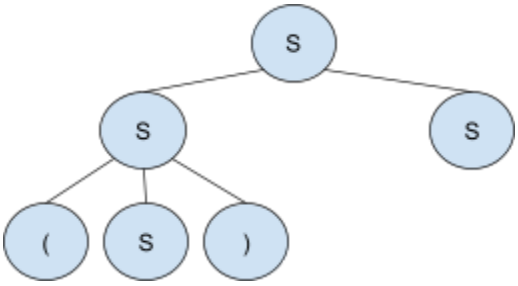
Practice:

- Let any set of production rules in a CFG be S → SS | (S) | ( )
  What is the derivation tree over this string "(())()" ?
  - ➢ Leftmost derivation: S → SS → (S)S → (())S→ (())()

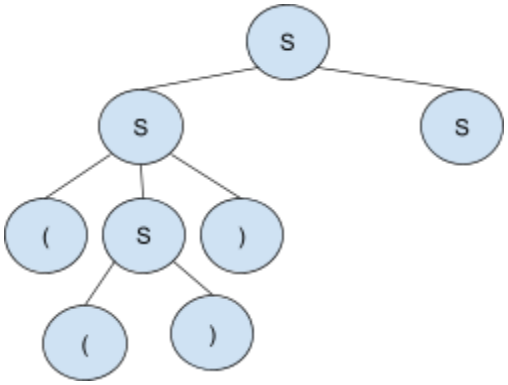
Step 1:



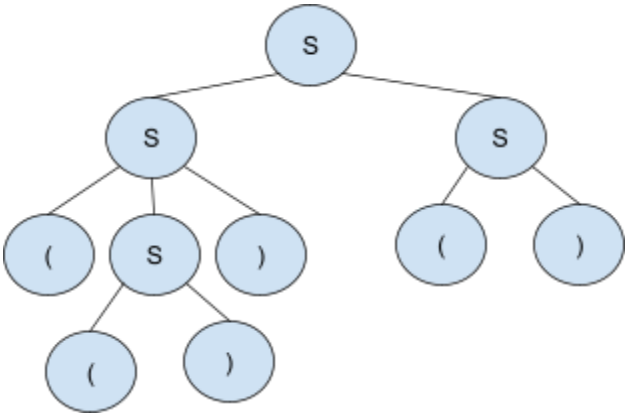
Step 2:



Step 3:



Step 4:



Ambiguous Grammar

A CFG is said to be **ambiguous** if there exists more than one derivation tree for the given input string i.e., more than one LeftMost Derivation Tree or RightMost Derivation Tree.

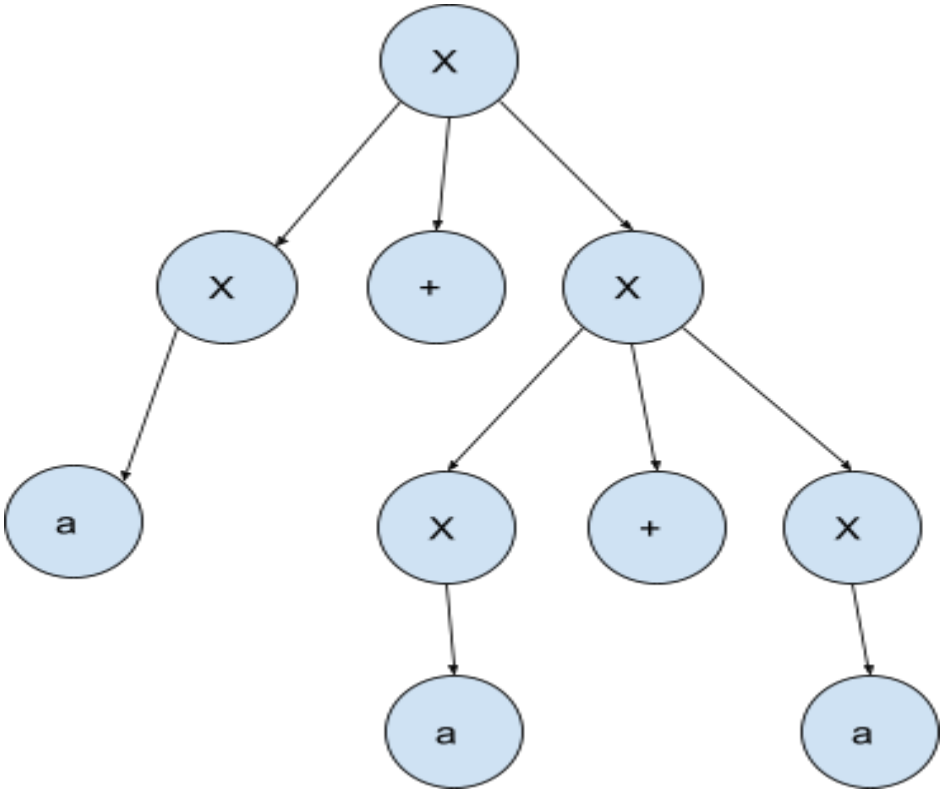
Example:

X → X+X | X*X | X | a

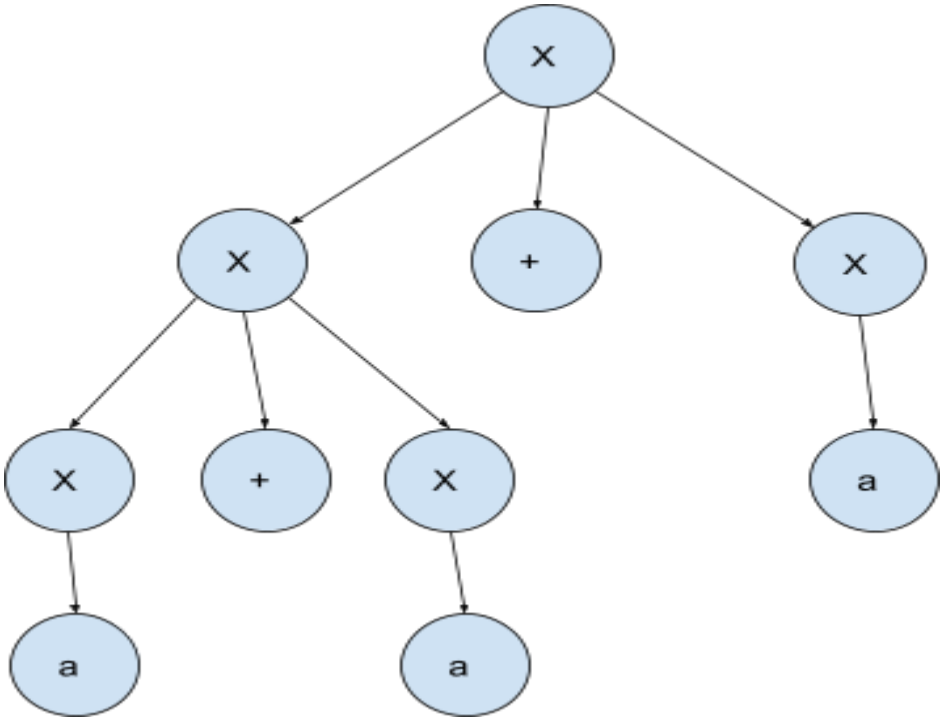
What is the derivation tree over this string "a+a+a" ?

- There is more than one derivation for it:

1. X → X+X → a+X → a+X+X → a+a+X→ a+a+a

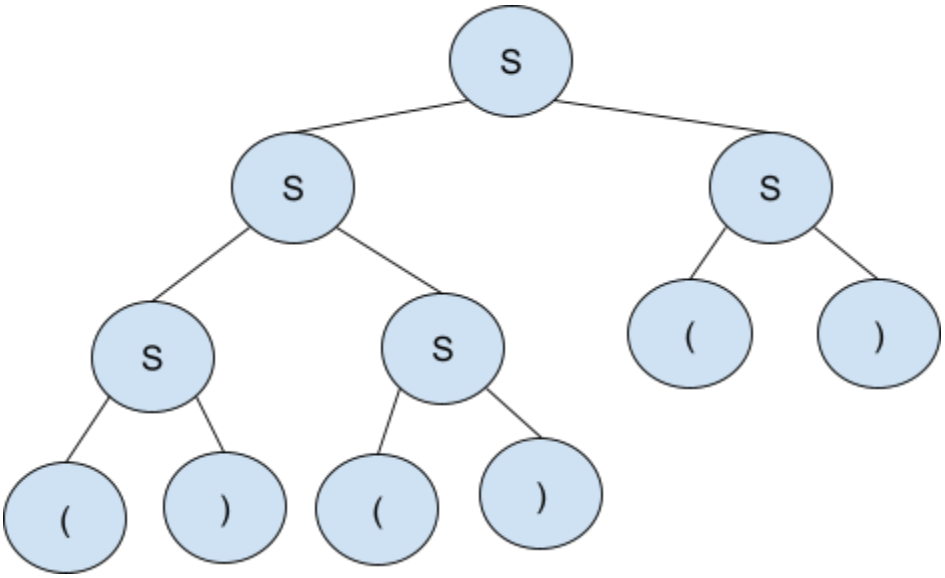


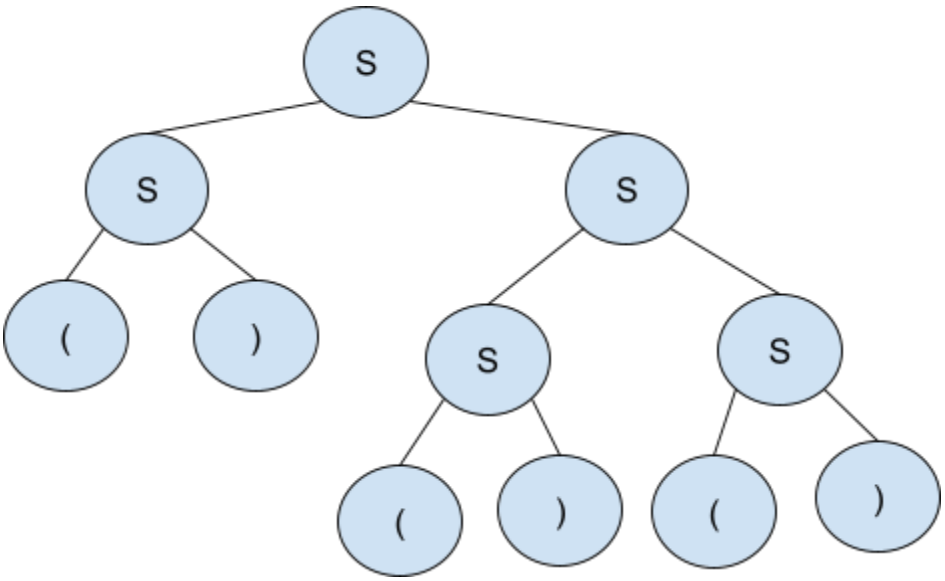
2. X → **X**+X → **X+X**+X → a+X+X → a+a+X → a+a+a



Practice:

1. Let any set of production rules in a CFG be S → SS | (S) | ()
   Show the 2 parse trees over string “()()()” that causes ambiguity.



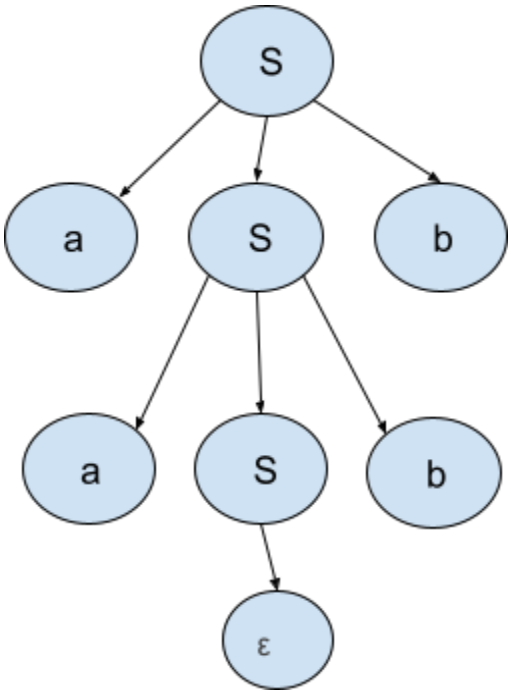


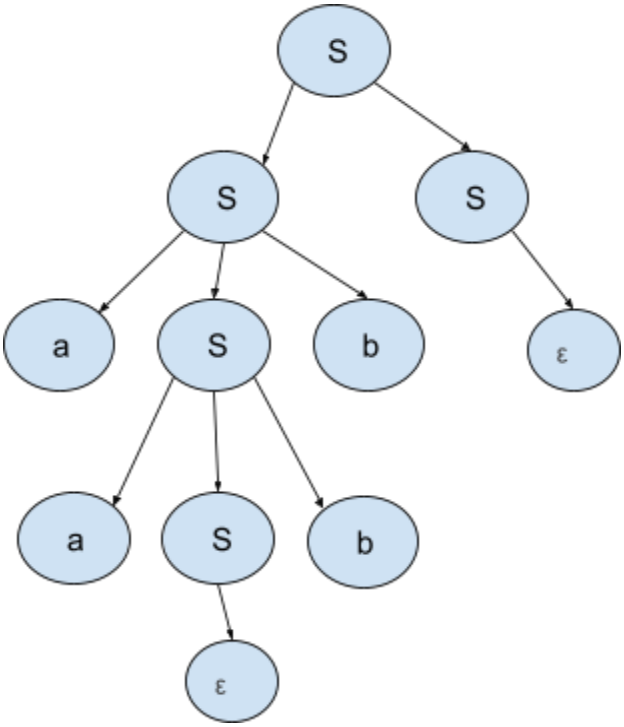
Practice:

Check whether the given grammar G is ambiguous or not:
S → aSb | SS | ε

Solution:
- The grammar is ambiguous as for the string "aabb" the above grammar can generate two parse trees:

1) S → aSb → aaSbb → aabb
2) S → SS → aSbS → aaSbbS → aabbS → aabb





Removing grammar ambiguity

1. Left recursion.
2. Left factoring.

Both problems prevent any LL parser from deciding deterministically which rule should be fired.

Left Recursion

We have to eliminate left recursion because top down parsing methods cannot handle left recursive grammars.

- A grammar is left recursive if it has a nonterminal A such that there is a derivation $A \rightarrow A\alpha$ for some string α.
- Consider the left recursive grammar: $A \rightarrow A\ \alpha \mid \beta$

  $A \rightarrow \beta\ A'$

  $A' \rightarrow \alpha A' \mid \varepsilon$

Practice:

Remove the left recursion from the following CFGs:

1) $A \rightarrow A\ a \mid A\ b \mid c \mid d$

2) $E \rightarrow E + T \mid T$

   $T \rightarrow T * F \mid F$

3) $S \rightarrow (L) \mid a$

   $L \rightarrow L\ ,\ S \mid S$

4) $N \rightarrow N\ 0\ N\ 1\ N$

   $N \rightarrow 0\ 1$

Solution:

1) $A \rightarrow c\ A' \mid d\ A'$
$A' \rightarrow a\ A' \mid b\ A' \mid \varepsilon$

2) $E \rightarrow TE'$
$E' \rightarrow +TE'$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$

3) $S \rightarrow (L) \mid a$
$L \rightarrow S\ L'$
$L' \rightarrow , S\ L' \mid \varepsilon$

4) $N \rightarrow 0\ 1\ N'$
$N' \rightarrow 0\ N\ 1\ N\ N' \mid \varepsilon$

Left Factoring

- In left factoring it is not clear which of two alternative productions to use to expand a nonterminal A.
  A → α β1 | α β2
- We don't know whether to expand A to α β1 or to α β2
- To remove left factoring for this grammar replace all A productions containing α as prefix by:
  A → α A′
  A′ → β1 | β2

Practice:

Remove the left factoring from the following CFGs:
1) A → aAB | aBc | aAc
2) S → a | ab | abc | abcd

1. Step 1: A → a A'
   A' → AB | Bc | Ac
   Step 2: A → a A'
   A' → AS | Bc
   S → B | c

2. Step 1: S → aA
   A → b | bc | bcd | ε
   Step 2: S → aA
   A → b B
   B → c | cd | ε
   Step 3: S → aA
   A → b B | ε
   B → c C | ε
   C → d | ε

Task

1) **Find a context-free grammar** for the language that contains palindromes of even length over the alphabet (a, b).

2) **Remove left recursion** for the the following grammar then **write the derivation and draw the derivation tree** for the string “aaacbb”:
   S → AcB
   A → Aa | a
   B → Bb | b