AUTOMATA FORMAL LANGUAGES AND LOGIC

Lecture notes on Regular grammar and Parsing

Prepared by:

Prof.Sangeeta V I

Assistant Professor

Department of Computer Science & Engineering

PES UNIVERSITY

(Established under Karnataka Act No.16 of 2013)

100-ft Ring Road, BSK III Stage, Bangalore – 560 085

Table of contents

Examples Solved:

Examples Solved:

#	Generating a parse tree for a given String w and Grammar G.	Page number
1	Generate parse tree for the sting w=1010 given the grammar . S→ 0S|1S|0	19
2	Generate parse tree for the sting w=aaabbbb given the grammar . S→aaB B→ aB|bbbE E→ bE |λ	20

Examples Solved:

#	Constructing a left linear grammar from the finite automata.	Page number
1	Constructing a left linear grammar for L={aw,w∈{a,b}*}.	22

Examples Solved:

#	Converting to left linear grammar to finite automata	Page number
1	Convert a given left linear grammar to finite automata Left Linear grammar B→ Ba|Bb|Aa A→ λ	23

1. Introduction to Grammar

Language is a set of strings constructed over an alphabet which satisfies certain properties or follows a certain set of rules. These rules can be encoded using the concept of Grammar. Grammars are used to compactly express and generate languages.

Grammars denote syntactic rules that means it is concerned only with the syntax of a string and not the meaning .

Grammar is another way to represent a set of strings. Rather than define the set through a notion of which strings are accepted or rejected, like we do with a machine model, we define the set by describing a collection of strings that the grammar can generate.

For example:She is a boy, is grammatically correct but semantically wrong.

There are various applications of regular grammars for example in compiler design the concept of grammar is used construct syntax of a programming language

Formal grammar or just grammar is a set of rules that describe how strings that are valid according to the language's syntax. can be generated from the language's alphabet.

2. Grammar definition

A grammar is a 4-tuple **G = (V,T,P,S)**, where

- **V** is a set of variables/non-terminals. We can have

  a. Lowercase names such as expr,var,op,stmt. or

  b. Capital letter near the beginning of the alphabet. A,B,C,D or Uppercase letters near the end of alphabet X,Y,Z. as non-terminals

  c. Usually Variable S is used as the-start symbol.

  d. Lowercase letters near the end of the alphabet. U,v,w,x,y,z are used to represent the entire string.

- **T** is the set terminal symbols

  a. These are basically the symbols from the input alphabet for example could be a Keywords such as :

  b. If ,else,then ,for ,while,do while etc

  c. Or Digits 0 t0 9

  d. Or Lowercase alphabets such as a,b,c,d etc

  e. Or Bold faced letters

  f. Or Symbols such as + , - , * , / , : , ; etc

  We can use Greek letters-α,$\boldsymbol{\beta}$,$\boldsymbol{\gamma}$ to denote a grammar symbol which could either be a either terminal or a non-terminal.

- **P** is a finite set of production rules (also called simply rules or productions) of form $\alpha \rightarrow \boldsymbol{\beta}$ where α is always a non terminal and cannot be $\boldsymbol{\lambda}$. And $\boldsymbol{\beta}$ is a string over $(V \cup T)^*$

- **S** is the start symbol.

- The set of strings that can be generated or derived from Start symbol S of a grammar is called Language of the Grammar denoted by L(G).

3. Sentence and Sentential form

Sentence or the word string are used alternatively and are made up only of terminals.

Example :

For the Grammar S->aS | a |λ

Sentence (or string) is { a or λ or aa or aaaa)

A **sentential form** is made up over set of terminals and non-terminals (VUT)*

Example:

For the Grammar S->aS | a |λ

Sentential form is { a or λ or aS or aa or or aaS aaaa)

4. Chomsky Hierarchy

Noam Chomsky in 1956 described Chomsky hierarchy of grammars

Chomsky Hierarchy represents the class of languages that are accepted by the different machines. Chomsky classified grammars on the basis of the form of their productions. This is a hierarchy. Therefore every language of type 3 is also of type 2, 1 and 0. Similarly, every language of type 2 is also of type 1 and type 0, etc

Type 3 Grammar is known as Regular Grammar.

5. Linear and Non-Linear grammar

We can broadly categorize grammars into two categories as being linear and non-linear

Linear grammar is the one which has at most one non terminal in the RHS of the production.

For Examples S -> aSa | aS | Sa | ϵ

here we have only one nonterminal/variable in the right hand side of each of its productions.

In non linear grammar there is no restriction on the number non terminals that can appear on the RHS of a production for example : S -> SS | aSS | a | ϵ

Here we have two non terminals in the RHS .

A regular grammar has more restriction and is a subset of linear grammar.

5.1 Right Linear grammar and Left Linear Grammar

A regular grammar can either be right linear or left linear.

A grammar is **left linear** iff the non-terminal on RHS of a production appears on the leftmost side. That means the non-terminal is the first grammar symbol on the RHS of a production

For example

S -> Sa | ϵ (left linear)

A grammar is **right linear** iff the non-terminal on RHS of a production appears on the rightmost end. That means the non-terminal is the last grammar symbol on the RHS of a production

For example

S -> aS | ϵ (right linear)

Since both the forms are linear, we can have only one non terminal on RHS of a production.

The right- and left-linear grammars are equivalent. That means for a given language we can have both right and left linear grammar constructed. We could also convert one form to another.

6. Constructing regular grammar for given language descriptions.

Example 1:

Construct a regular grammar for the language L = {a}.

Language	Regular Expression	Regular Grammar
L = {a}	a	S→a

We start the regular grammar with a start symbol "S".

We can convert regular expressions to regular grammar as the regular expressions are a form of representing regular languages.

Here ,since 'a' is the only symbol in the language ,the regular grammar from S generates only 'a', S->a (production rule)

Example 2:

Construct a regular grammar for the language L = {a, b}.

Language	Regular Expression	Regular Grammar
L = {a, b}	a+b	S→a | b

Here ,since 'a' or 'b' is a symbol in the language ,the regular grammar from S generates either 'a' or 'b' , S->a or S->b.

Example 3:

.Construct a regular grammar for the language L = {ab}.

Language	Regular Expression	Regular Grammar
L = {ab}	a.b	S→aA A→b

Here ,since ‘a.b’ is symbol in the language ,the regular grammar produces S->ab

Example 4:.

Construct a regular grammar for the language L = {ab,ba}.

Language	Regular Expression	Regular Grammar
L = {ab}	a.b	S→ab|ba

Example 5:

Construct a regular grammar for the language L={a^n|n>=0}

Language	Regular Expression	Regular Grammar
L = {**λ**,a,aa,aaa,...}	a*	S→aS|**λ**

The language with any number of a’s including empty string .

The regular grammar from S generates any number of a’s with the production S->aS or empty string with production S-> **λ**

We combine the productions,

S-aS| **λ**

Example 6:

Construct a regular grammar for the language L={a^n|n>=1}

Language	Regular Expression	Regular Grammar
L = {a,aa,aaa,...}	a^+	S→aS|a

The language with any number of a's but not an empty string ,minimum string is one 'a'.

The regular grammar from S generates any number of a's with the production S→ aS or single 'a' with production S→ a

We combine the productions,

S→ aS| a

Example 7:

Construct a regular grammar for the regular expression (a+b)*

Language	Regular Expression	Regular Grammar
L = {**λ**,a,b,abb,baaaab,......} L={any number of 'as and b's}	(a+b)*	S→aS|bS|**λ**

The language with any number of a's and any number of b's including an empty string.

The regular grammar from S generates any number of a's with the production S→ aS and any number of b's with the production S->bS and empty string with production

S→ **λ**.

We combine the productions,

S→aS|bS|**λ**

Example 8 :

Construct a regular grammar for the regular expression $(a+b)^+$

Language	**Regular Expression**	**Regular Grammar**
L = {λ,a,b,abb,baaaab,......} **L={any number of 'a's and b's}**	**$(a+b)^+$**	**S→aS|bS|a|b**

The language with any number of a's and any number of b's but not an empty string.

The regular grammar from S generates any number of a's with the production S->aS and any number of b's with the production S->bS ,single 'a' with production S-> a and single 'b' with production S-> b.

We combine the productions,

S→aS|bS|a|b

Example 9:

Construct a regular grammar for the regular expression (ab)*

Language	Regular Expression	Regular Grammar
L = {λ,,ab,ababababab,......} L={sequences of ab's}	(ab)*	S -> abS | λ

The language with sequences of ab's including empty string.

Example 10:

Construct a regular grammar for the regular expression (ab+ba)*

Language	Regular Expression	Regular Grammar
L = {λ,,ab,ababababab,......} L={sequences of ab's}	(ab)*	S -> abS | baS | λ

The language with sequences of ab's or ba's including empty string.

Example 11:

Construct a regular grammar for the language L={a^m b^n |n,m>=0}

Language	Regular Expression	Regular Grammar
L={λ,a,b,bb,abb,,,......} L={any number of a's followed by any number of b's}	a*b*	S -> aS|A A->bA |λ

aS generates as many a's as we want and when we go ahead in the pattern we introduce a new non-terminal **A** which generates only b's.

Example 12:

Construct a regular grammar for the given language with an even number of a's. L={a^{2n} |n>=0}

Language	Regular Expression	Regular Grammar
L={λ, aa, aaaa, aaaaaa ….	(aa)*	S -> aaS | λ

Example 13:

Construct a regular grammar for the given language with an odd number of a's. L={a^{2n+1} |n>=0}.

Language	Regular Expression	Regular Grammar
L={a, aaa, aaaaa, aaaaaaa ….	(aa)*a	S -> aaS | a

Example 14:

Construct a regular grammar for the given language with a number of a's as multiples of 4. L={a^{4n} |n>=0}

Language	Regular Expression	Regular Grammar
L={**λ**, aaaa, aaaaaaaa, aaaaaaaaaaaa, ….	(aaaa)*	S -> aaaaS | **λ**

Example 15:

Convert finite automata to regular grammar for the language to accept at least one 'a' over the alphabet Σ={a,b}

Language	Regular Expression	Regular Grammar
L={a, bb... a, ba any # of a's & b's	b* a (a+b)*	S -> bS | aA A -> aA | bA | **λ**

7. Aspects of a Grammar

There are two Aspects of a Grammar.

First is the Generative(derivation) aspect that means given a Grammar G describing language we can generate all strings that belong to the language of the Grammar.

Second is theAnalytical(parsing) aspect where given a string w we can make a check whether w belongs to the language of the Grammar or not.

8. Parse tree/ derivation tree

We all know what a tree looks like. It has leaves, branches and a root.

Technically we specify the tree upside down. That is we specify the root first and then leaves where branches will help connect root to each of the leaves.

Parse/Derivation tree is a graphical representation for the derivation of the given production rules .

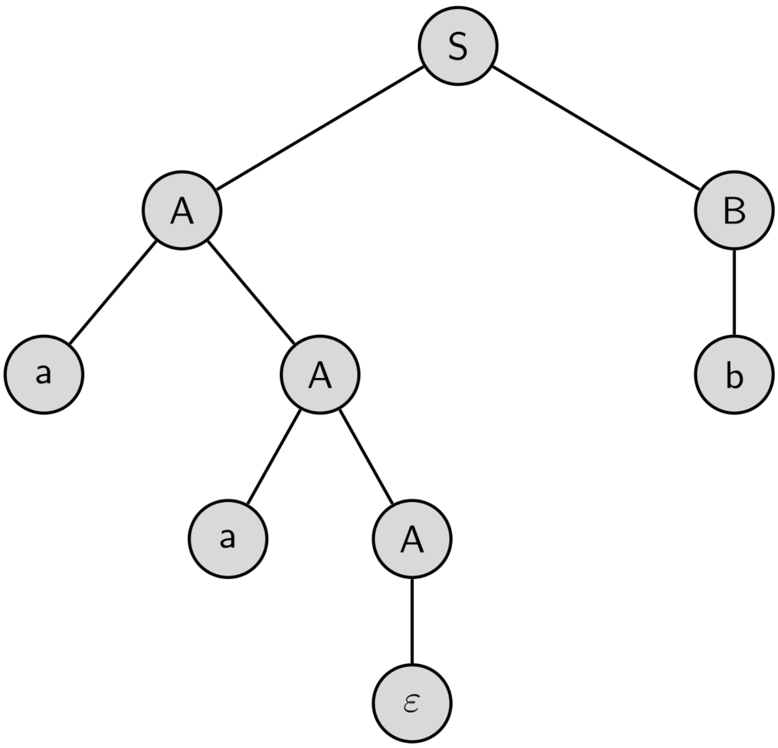
A parse tree/derivation tree contains the following properties:

- The root node is always a node indicating start symbols.
- The interior nodes are always non-terminal.
- The leaf node is always represented by a terminal.
- The derivation is read from left to right. Yield or output of the parse tree is the string derived.

9.1 Tree and its representation

- A tree is nothing but a graph.
- It is made up of a finite set of nodes.
- Nodes are usually denoted by circles or ovals .
- We use the word Edge instead of Branch.
- Edges are the connections between the nodes. It connects two nodes.
- Edges are usually represented by lines, or lines with arrows.
- Tree is a graph without cycles. That is When following the graph from node to node, we will never visit the same node twice.
- From the root node we can reach every other node. So, tree is completely connected.
- Hence we can say a tree is a connected acyclic graph.

Representation of a tree using data



S is the root note .

V is the set of non -terminals which are the interior nodes.

T is the set of terminal symbols ,which are the leaf nodes.

Yield of the parse tree is abb.

9.2 Parsing

Parsing is the process of determining whether a String w belongs to the language of the Grammar or not. Stated another way, we can say whether the String w can be derived from the Grammar or not.

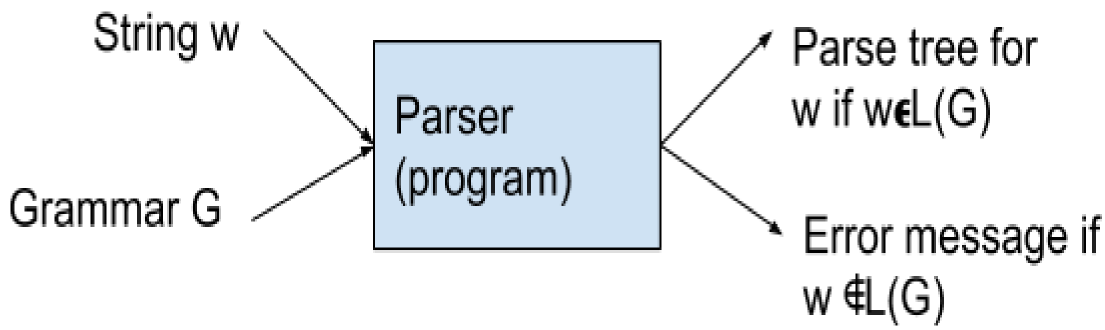
There are two ways in which parsing can be performed:

a) Top Down Parsing
b) Bottom up Parsing

9.2.1 Top-down Parsing and bottom up Parsing

Top-down Parsing is a parsing technique that first looks at the highest level of the parse tree which is the Start Symbol and works down the parse tree by using the rules of grammar while Bottom-up Parsing is a parsing technique that first looks at the lowest level of the parse tree that means the string w and works up the parse tree till the Start symbol by using the rules of grammar

Parser is a program which takes as input the string w and grammar g and produces as output parse tree for w if w belongs to lang of the grammar otherwise outputs an error message.



9.2.2 Generating a parse tree for a given String w and

Grammar G.

Example 1:

Generate parse tree for the sting w=1010 given the grammar .

S→ 0S|1S|0

Derivation: w=1010

S⇒ 1S (using S→ 1S)

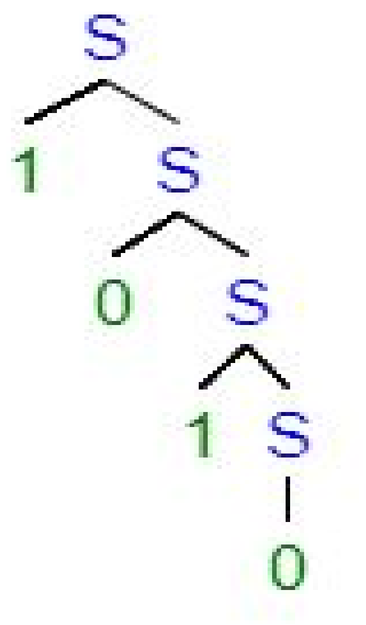
 ⇒ 10S (using S→ 0S)

 ⇒101S (using S→ 1S)

 ⇒ 1010 (using S→ 0)

Sentence or string =1010

Parse Tree: w=1010



Yield of the tree=1010

Example 2:

Generate parse tree for the sting w=aaabbbb given the grammar .

S→aaB

B→ aB|bbbE

E→ bE |λ

Derivation: w=aaabbbb

S⇒ aaB (using S→ aaB)

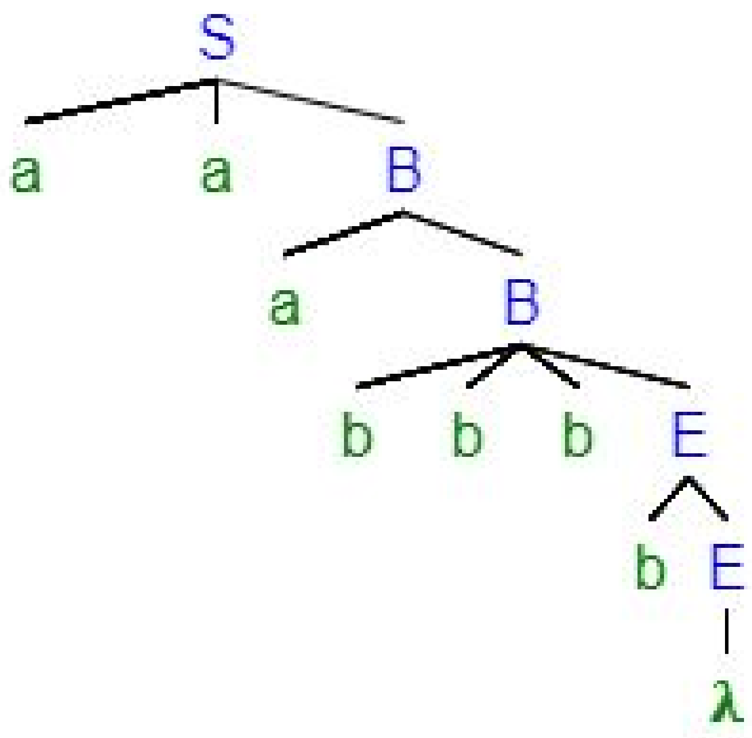
⇒ aaaB (using S→ aB)

⇒ aaabbbE (using B→ bbbE)

⇒ aaabbbbE (using E→ bE)

⇒ aaabbbb (using E→ λ)

Parse Tree: w=aaabbbb



Yield of the tree:aaabbbb

9.3 Constructing a left linear Grammar

There is another form in which regular grammars could possibly be written and it is known as the Left linear form. That means the non-terminal on RHS of a production rule must be present at the leftmost side or should be the first symbol on RHS of the production rule.

Let us consider the right linear grammar for the lang where all the strings must start with an 'a'.

Right Linear Grammar

$A \rightarrow aB$

$B \rightarrow aB|bB|\lambda$

L={starting with 'a'}

You might think that reversing all the symbols on RHS of every production rule will get us an equivalent Left linear grammar for the language L.

But surprisingly, the language represented by this grammar is the reverse of the language represented by the Right linear form.

When we reverse all the symbols in RHS of every production rule what we get is a language where every string ends with an a.

So, the conversion or construction of LLG is not as straightforward. We need to work our way via finite automata.

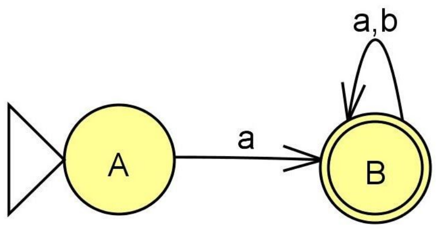
9.4 Constructing a left linear grammar from the finite automata.

- Step 1: Consider the Finite automata for language "L".
- Step 2: Rever the finite automata L .
    - Steps to reverse:
        - Change initial to final state.
        - Final state to initial state.
        - Reverse the directions of the transitions.
        - We get L^R.
- Step 3: Construct a right linear grammar for this language which is reverse of L L^R.
- Step 4: Now Reverse the symbols in RHS of every production rule.
    - Now with this step we have got a left linear grammar for a language which is a reverse of the reversed language, thereby constructing a left linear grammar for L.

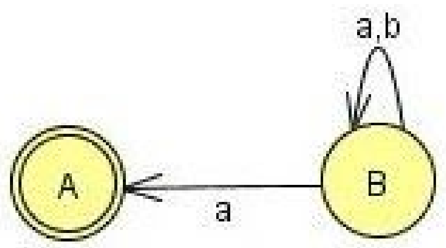
Example 1:

Constructing a left linear grammar for L={aw,w∈{a,b}*}.

Step 1: Finite automata for the language where the strings must start with an a,L={aw,w∈{a,b}*}



Step 2: Reversing directions in this automata represents automata for the reverse of the language L which will be a set of strings ending with an a.



Here, B is the start state.

Step 3: We will now generate the right linear grammar for this reversed language. We very well know how to convert finite automata to regular grammar.

$B \rightarrow aB|bB|aA$
$A \rightarrow \lambda$

Step 4: In order to get LLG for the language L is to reverse the symbols in RHS of every production rule of right linear grammar.

$B \rightarrow Ba|Bb|Aa$
$A \rightarrow \lambda$

9.5 Converting to left linear grammar to finite automata

This is exactly a reverse process of conversion from FA to Left linear grammar.

- Step 1: Given the Left linear grammar for Language "L",reverse the symbols on RHS of each production rule in order to get a right linear grammar for reversal of the language.

- Step 2: Construct finite automata for reverse of the language using right linear grammar.
- Step 3: Reverse this finite automata. This automata will now accept the language L.

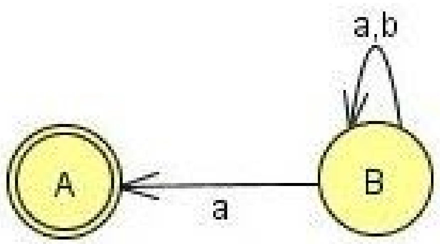
Example 1:

Convert a given left linear grammar to finite automata
Left Linear grammar
B→ Ba|Bb|Aa
A→ λ

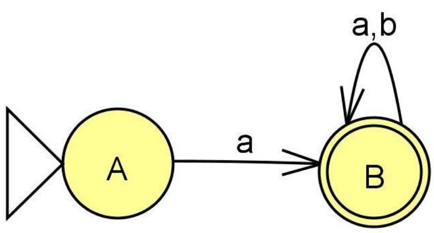
- Step 1: Reverse the symbols on RHS of each production rule in order to get a right linear grammar for reversal of the language.

  B→ aB|bB|aA
  A→ λ

- Step 2: Construct finite automata for reverse of the language using right linear grammar.



- Step 3: Reverse this finite automata. This automata will now accept the language L.

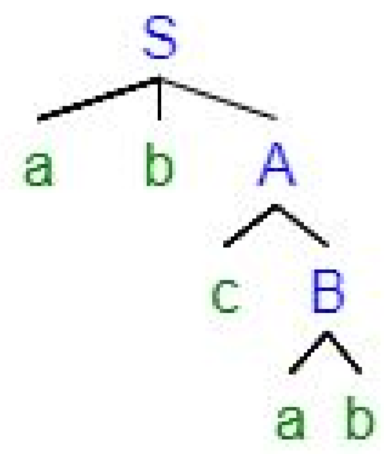


Here ,B is the start state.

9.6 Which one is easier left linear or right linear?

Given the right linear grammar,
S→ abA
A→ cB|aC
B→ ab
C→ b

Parse tree for the string abcab



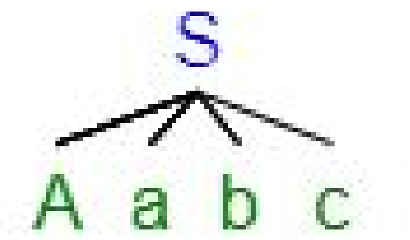
We start with the Start Symbol S. Since there is only one alternative and the first symbol in our string is also a, we will use the production rule S→ abA and expand our parse tree.
So we have successfully got a match for the first two symbols.
The next symbol to be matched is c and we have the non-terminal A.
Therefore we will expand the parse tree using the alternative A → cB.
Next we must match a and the non-terminal to be expanded is B. We use B→ab to expand the tree which will also match the last symbol in our string which is b.

We see the process of derivation of string is easy. We just need to see which input symbol is next and pick the alternative for non-terminal which will match that symbol. It is quite easy as we scan the string left to right and also derive the string left to right.

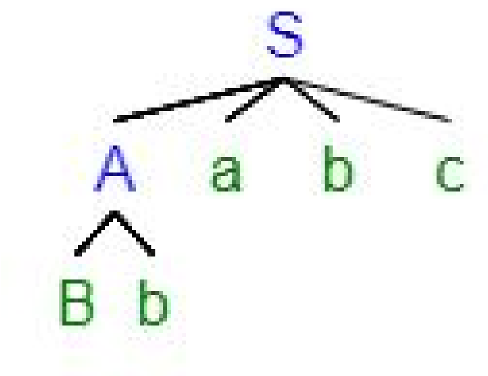
Consider this Left linear grammar
S→ Aabc
A→ Bb|C
B→ a
C→ b

Parse tree for the string ababc

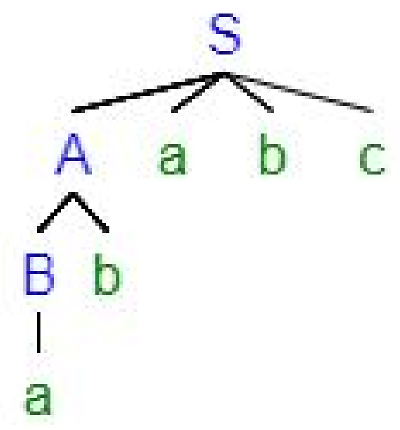
Can the rule S→Aabc recognize the string ababc?
We see that the string is ending with abc and it matches with the last part of the rule.



Now,which rule do we choose to A→Bb or A->C
We choose A->Bb as we see from the input string that the next symbol to be parsed is b
.



Then we choose B->a which completes parsing the string.



This way of parsing although possible is not very intuitive as we need to match the symbols in the string backwards.

Hence right linear form is always preferred over left linear form.