Department of Computer Science and Engineering
PES University, Bangalore, India

UE23CS243A: Automata Formal Language and Logic

Prakash C O, Associate Professor, Department of CSE

Regular Grammar (type-3 grammar):

Why do we care about regular grammars?

Programs are composed of tokens:

- Identifiers
- Literals (Integer literals, floating point literals, character literals, string literals, …)
- Keywords
- Operators and
- Special symbols (i.e., Punctuation symbols, …)

Each of these can be defined by regular grammars.

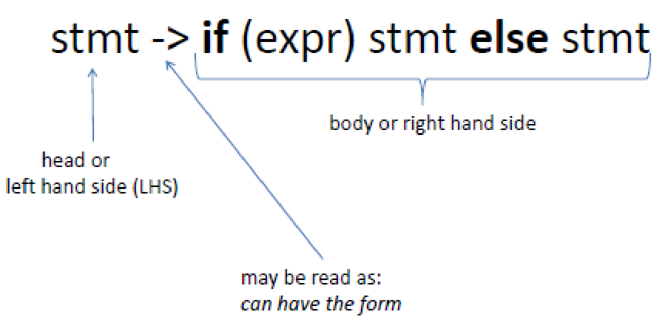
Formal Definition of a Grammar

Formally, a grammar is a 4-tuple G = (V, T, P, S)

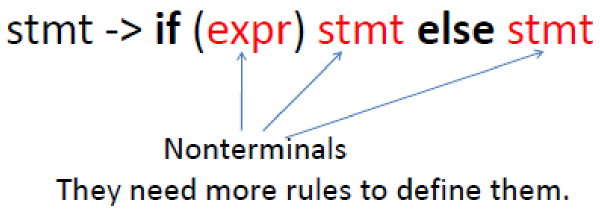
where:

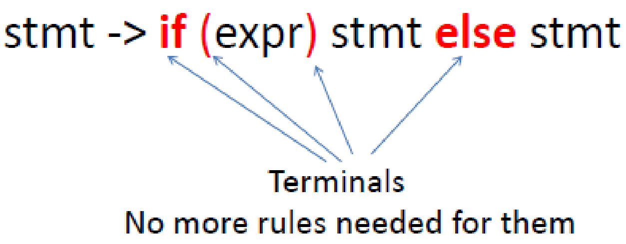
- **V - Set of nonterminals (or variables)**
- **T - Set of terminal symbols**
- **P - Set of productions(or rules)**
  - **The head is nonterminal**
  - **The body is a sequence of teminals and/or nonterminals**
- **S – Start Symbol (Designation of one nonterminal as starting symbol)**

Example:



➢ **Production rules.**





Note: **All the variables in your grammar must be reachable from the start symbol of the grammar and all variables must derive something (or end)**

Derivation:

A derivation in compiler design is **the successive application of production rules to produce the desired input string.**

- Given the grammar (i.e. productions)
- begin with the start symbol
- repeatedly replacing nonterminal by the body
- We obtain the language string/statement defined by the grammar (i.e. group of terminal strings)

There are two types of derivation in compiler design:

1) **Left-most Derivation**
2) **Right-most Derivation**

Left-most Derivation

The left-most derivation is a method of transforming an input string according to the grammar rules of a programming language. **The leftmost non-terminal is selected at each stage of left-most derivation.**

Grammar:

$$\begin{aligned} &\mathbf{S \rightarrow ABC} \\ &\mathbf{A \rightarrow a} \\ &\mathbf{B \rightarrow b} \\ &\mathbf{C \rightarrow c} \end{aligned}$$

Input string: **abc**

Left-most derivation: $\mathbf{S \Rightarrow ABC \Rightarrow aBC \Rightarrow abC \Rightarrow abc}$

Right-most Derivation

The right-most derivation is a method for transforming an input text depending on the grammar rules of a programming language.

The rightmost non-terminal is selected for expansion at each step, which is regulated by the production rule associated with that non-terminal.

Grammar:

S → ABC
A → a
B → b
C → c

Input string: abc

Right-most derivation: **S ⇒ ABC ⇒ ABc ⇒ Abc ⇒ abc**

Parse Tree (or Derivation tree): **Parse Tree is the geometrical or pictorial representation of a derivation.**

- Parse tree is the hierarchical representation of terminals or non-terminals.
- These symbols (terminals or non-terminals) represent the derivation of the grammar to yield input strings.
- In parsing, the string springs using the beginning symbol.
- The starting symbol of the grammar must be used as the root of the Parse Tree.
- Leaves of parse tree represent terminals.
- Each interior node represents non-terminals (or productions) of a grammar.

Example 1:

S → sAB

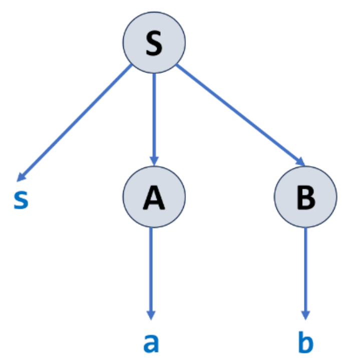
A → a

B → b

The input string is "**sab**",

Derivation: S ⇒ sAB ⇒ saB ⇒ sab

then the Parse Tree is:



Example-2:

S → AB

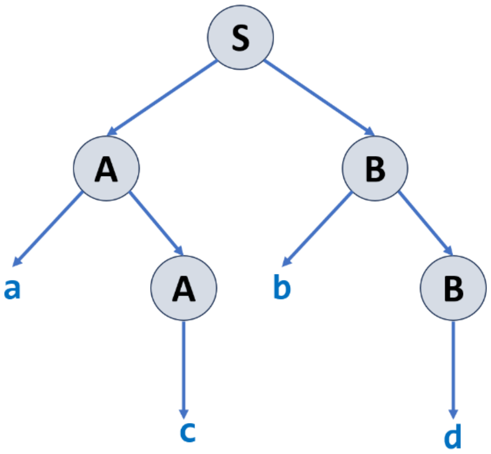
A → c | aA

B → d | bB

The input string is "**acbd**",

Derivation: S ⇒ AB ⇒ aAB ⇒ acB ⇒ acbB ⇒ acbd

then the Parse Tree is as follows:



Sentential Form:

A sentential form is any string consisting of non-terminals and/or terminals that is derived from a start symbol. Therefore, every sentence is a sentential form, but only **sentential forms without non-terminals are called sentences.**

Example: Grammar: **S → aSb | λ**

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$$

Sentential Forms sentence

We write:

$$S \overset{*}{\Rightarrow} aaabbb$$

Instead of:

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$$

Linear Grammar

Grammars with at most one variable (or Nonterminal) at the right hand side of a production.

Example-1:

S → aSb
S → λ

Example-2:

S → Ab
A → aAb
A → λ

Non-Linear Grammar

Grammars with more than one variable (or Nonterminal) at the right hand side of a production.

Example:

S → SS

```
S → λ
S → aSb
S → bSa
```

$L(G) = \{ w \mid n_a(w)=n_b(w) \}$

Right-Linear Grammar

- **Right linear grammar:** The non-terminal symbol should be at the right end of the production body.
  All productions have the form:

S → wS	or	S → wA
S → w		A → w

  where *S* and *A* are non-terminals in *V* and *w* is a string of terminals i.e., $w \in \Sigma^*$
- **Right linear grammar: The grammars, in which all rules are of the form *A* → *wα* where *w* is a string of terminals and *α* is either empty or a single nonterminal.**

Examples:

1)
```
S → abS
S → a
```

2)
```
S → 00B | 11S
B → 0B | 1B | 0 | 1
```

3)
```
S → cS
S → λ
```

Left-Linear Grammar

- **Left linear grammar:** The non-terminal symbol should be at the left end of the production body.
  All productions have the form:

S → Sw	or	S → Aw
S → w		A → w

  where *S* and *A* are non-terminals in *V* and *w* is a string of terminals i.e., $w \in \Sigma^*$
- **Left linear grammar: The grammars, in which all rules are of the form *A* → *αw* where *α* is either empty or a single nonterminal and *w* is a string of terminals.**

Example:

```
S → Aab
A → Aab | B
B → a
```

Regular Grammar

- **A regular language can be described by a special kind of grammar called regular grammar.**
- **A regular grammar is a grammar that is either left-linear or right-linear.**
- **Regular grammars generate regular languages.**

Example: Construct a regular grammar for the language of the regular expression (a|b)* *a*

Regular Grammar is:

$$\left.\begin{array}{l} S \rightarrow aS \\ S \rightarrow bS \\ S \rightarrow a \end{array}\right\} \quad \text{or} \quad S \rightarrow aS \mid bS \mid a$$

Does this sentence (or string) **baaba** conform to the written grammar. YES

Derivation: $S \Rightarrow bS \Rightarrow baS \Rightarrow baaS \Rightarrow baabS \Rightarrow baaba$

Construct a Regular Grammar for the given language description.

Hint: Easy way is to construct NFA(or λ-NFA with state names as uppercase letters) and then writing the regular grammar.

1	**L = {a}**
2	**L = {ab}**
3	**L = { w | w ∈ {a, b}* and |w| = 2 }**
4	**L = { w | w ∈ {a, b}* and |w| ≤ 2 }**
5	**L = {aaaa}**
6	**L = {a, b}**
7	**L = {ab, ba}**
8	**L = { λ, a, aa, aaa, ... }**
9	**L = { a, aa, aaa, ... }**
10	**L = { λ, ab, abab, ... }**
11	**L = { λ, a, b, aa, ab, ba, bb, aaa, bbb, bba, bab, abb, aab, aba, baa, ... }**
12	**L = { a, b, aa, ab, ba, bb, aaa, bbb, bba, bab, abb, aab, aba, baa, ... }**
13	**Strings with zero or more a's only and Strings with zero or more b's only.**
14	**Strings with one or more a's only and Strings with one or more b's only.**
15	**Strings begins with zero or more a's followed by single b.**
16	**Strings begins with a and followed by zero or more b's.**
17	**L={ a^{2n} | n≥0 }**
17a	**L={ a^{2n} | n≥1 }**
18	**L={ a^{2n+1} | n≥0 }**
19	**L={ a^{4n} | n≥0 }**
20	**L = { w_1aw_2 | w_1 ∈ {b}* and w_2 ∈ {a, b}*}**
21	**L={ a^mb^n | m≥0 and n>0 }**
22	**L={ a^mb^n | m>0 and n≥0 }**
23	**L={ a^mb^n | m>0 and n>0 }**
24	**L={ a^mb^n | m≥0 and n≥0}** L = { λ, a, aa, aaa, ... , b, bb, bbb, bbbb, ... , ab, aab, abb, abbb, aabb, aaab, ... }
25	**0 or more a's, followed by 0 or more b's, followed by 0 or more c's. L = {ε, a, b, c, aa, ab, ac,**

	bb, bc, cc, aaa, ...}
26	L={ $a^m b^n$ | m>0 or n>0 }
27	Strings ending with abb. L = {abb, aabb, babb, aaabb, ababb, baabb, bbabb, ...}
28	Strings starting with ab. L = {ab, aba, abb, abaa, abab, abba, abbb, ...}
29	Strings that contains aa. L = {aa, aaa, baa, aab, ...}
30	1 or more a's, followed by 1 or more b's, followed by 1 or more c's. L = {abc, aabc, abbc, abcc, aabbc, aabcc, abbcc, ...}
31	Strings that end with a or bb. L = {a, bb, aa, abb, ba, bbb, ...}
32	Strings with even number of a's followed by odd number of b's. L = {b, aab, bbb, aabbb, ...}
33	Binary strings ending with 3 0's. L = {000, 0000, 1000, 00000, 01000, 10000, 11000, ...}
34	Strings with even number of 1's. L = {ε, 11, 1111, 111111, ...}

Solutions:

Sl. No.	Language	Regex	Regular Grammar
1	L = {a}	a	S → a
2	L = {ab}	ab	S → aB B → b
3	L = { w | w ∈ {a, b}* and |w| = 2 }	(a|b)(a|b) Or [ab][ab]	S → aA | bA A → a | b
4	L = { w | w ∈ {a, b}* and |w| ≤ 2 }	(a|b)? (a|b)? or (ε|a|b)(ε|a|b)	S → aA | bA | λ A → a | b | λ
5	L = {aaaa}	aaaa	S → aA A → aB B → aC C → a
6	L = {a, b}	a|b	S → a | b
7	L = {ab, ba}	ab|ba	S → ab | ba or S → aB | bA A → a B →b
8	L = { λ, a, aa, aaa, ... }	a*	S → aS | λ
9	L = { a, aa, aaa, ... }	a^+	S → aS | a
10	L = { λ, ab, abab, ... }	(ab)*	S → abS | λ
11	L = { λ, a, b, aa, ab, ba, bb, aaa, bbb, bba, bab, abb, aab, aba, baa, ... }	(a|b)*	S → aS | bS | λ
12	L = { a, b, aa, ab, ba, bb, aaa, bbb, bba, bab, abb, aab, aba, baa, ... }	$(a\|b)^+$	S → aS | bS | a | b
13	Strings with zero or more a's only and Strings	a*|b*	S → A | B

	with zero or more b's only.		A → aA | λ B → bB | λ or S → aA | bB | λ A → aA | λ B → bB | λ
14	Strings with one or more a's only and Strings with one or more b's only.	$a^+ \| b^+$ or aa*|bb*	S → A | B A → aA | a B → bB | b or S → aA | bB A → aA | λ B → bB | λ
15	Strings begins with zero or more a's and ends with single b.	a*b	S → aS | b or S → Ab A → Aa | λ
16	Strings begins with a and followed by zero or more b's.	ab*	S → aB B → bB | λ or S → a | Sb
17	$L=\{ a^{2n} \mid n\geq0 \}$	(aa)*	S → aA | λ A → aS or S → aaS | λ
17a	$L=\{ a^{2n} \mid n\geq1 \}$	(aa)+	S → aA A → aS | a or S → aaS | aa
18	$L=\{ a^{2n+1} \mid n\geq0 \}$	(aa)*a	S → aA | a A → aS or S → aA A → aS | λ or S → aaS | a
19	$L=\{ a^{4n} \mid n\geq0 \}$	(aaaa)*	S → aA | λ A → aB B → aC C → aS or

			S → aaaaS | λ
20	L = { w_1aw_2 | $w_1 \in \{b\}^*$ and $w_2 \in \{a, b\}^*$}	b*a(a|b)*	S → bS | aA A → aA | bA | λ
21	L={ a^mb^n | m≥0 and n>0 } L = { b, bb, ab, aab, abb, bbb, ...}	a*bb* or a^*b^+	S → aS | bB B → bB | λ
22	L={ a^mb^n | m>0 and n≥0 } L = { a, aa, ab, aab, abb, aaa, ...}	aa*b* or a^+b^*	S → aA A → aA | λ | bB B → bB | λ
23	L={ a^mb^n | m>0 and n>0 }	aa*bb* or a^+b^+	S → aA A → aA | bB B → λ | bB
24	L={ a^mb^n | m≥0 and n≥0} L = { λ, a, aa, aaa, ... , b, bb, bbb, bbbb, ... , ab, aab, abb, abbb, aabb, aaab, ... }	a*b*	S → aS | bB | λ B → bB | λ
25	0 or more a's, followed by 0 or more b's, followed by 0 or more c's. L = {ε, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, ...}	a*b*c*	S → aS | bB | cC | ε B → bB | cC | ε C → cC | ε
26	L={ a^mb^n | m>0 or n>0 }	aa*b*| a*b*b or a^+b^*| a^*b^+	S → aS | aA | bA A → bA| λ
27	Strings ending with abb. L = {abb, aabb, babb, aaabb, ababb, baabb, bbabb, ...}	(a|b)*abb	S → aS | bS | aB B → bA A → b
28	Strings starting with ab. L = {ab, aba, abb, abaa, abab, abba, abbb, ...}	ab(a|b)*	S → aB B → bA A → aA | bA | ε
29	Strings that contains aa. L = {aa, aaa, baa, aab, ...}	(a|b)*aa(a|b)*	S → aS | bS | aA A → aB B → aB | bB | ε
30	1 or more a's, followed by 1 or more b's, followed by 1 or more c's. L = {abc, aabc, abbc, abcc, aabbc, aabcc, abbcc, ...}	$a^+b^+c^+$ or aa*bb*cc*	S → aA A → aA | bB B → bB | cC C → cC | ε
31	Strings that end with a or bb. L = {a, bb, aa, abb, ba, bbb, ...}	(a|b)*(a|bb)	S → aS | bS | a | bB B → b
32	Strings with even number of a's followed by odd number of b's. L = {b, aab, bbb, aabbb, ...}	(aa)*(bb)*b	S → aA | bB | b A → aC C → aA | bB | ε B → bD | ε D → bB
33	Binary strings ending with 3 0's. L = {000, 0000, 1000, 00000, 01000, 10000, 11000, ...}	(0+1)*000	S → 0S | 1S | 0A A → 0B

			B → 0
34	Strings with even number of 1's. L = {ε, 11, 1111, 111111, ...}	(11)*	S → 1A | ε A → 1S
35	Strings with atmost 3 a's and Σ ∈ {a, b}	b*a?b*| b*ab*ab*| b*ab*ab*ab*	S → bS | aA | λ A → bA | aB | λ B → bB | aC| λ C → bC | λ

Note:

Every regular language can be specified by a finite state automaton, a regular expression, or a regular grammar. Furthermore, all of these specifications are equivalent in the sense that they all capture the class of regular languages.

Finite State Automaton	**Recognizer**	Determines whether a particular input string in the regular language is recognized by the FSA or not.
Regular Expression	**Expresser**	Expresses a regular language as a pattern to which every string in the regular language conforms.
Regular Grammar	**Generator**	Provides grammar rules for generating strings in the regular language

Note: Regular grammar is a formal specification of a regular language by way of grammar rules.

Exercises:

Construct a Regular Grammar for the given language description and Regular expression.

1. Strings of a's and b's of length 2. Regex = aa | ab | ba | bb OR (a|b)(a|b)

2. Strings of a's and b's of length ≤ 2. Regex = ε|a|b|aa|ab|ba|bb OR (ε | a | b)(ε | a | b) OR (a|b)? (a|b)?

3. Strings of a's and b's of length ≤ 5. Regex = $(ε | a | b)^5$

4. Even-lengthed strings of a's and b's. Regex = (aa | ab | ba | bb)* OR ((a|b)(a|b))*

5. Odd-lengthed strings of a's and b's. Regex = (a|b) ((a|b)(a|b))*

6. L(R) = { w : w ∈ {0,1}* with at least three consecutive 0's } Regex = (0|1)* 000 (0|1)*

7. Strings of 0's and 1's with no two consecutive 0's. Regex = (1 | 01)* (0 | ε)

8. Strings of a's and b's starting with a and ending with b. Regex = a (a|b)* b

9. Strings of a's and b's whose second last symbol is a. Regex = (a|b)* a (a|b)

10. Strings of a's and b's whose third last symbol is a and fourth last symbol is b.

Regex = (a|b)* b a (a|b) (a|b)

11. Strings of a's and b's whose first and last symbols are the same. Regex = (a (a|b)* a) | (b (a|b)* a)

12. Strings of a's and b's whose first and last symbols are different. Regex = (a (a|b)* b) | (b (a|b)* a)

13. **Strings of a's and b's whose last and second last symbols are same.** Regex = **(a|b)* (aa | bb)**

14. **Strings of a's and b's whose length is even or a multiple of 3 or both.**

    Regex = **R1 | R2** where **R1 = ((a|b)(a|b))*** and **R2 = ((a|b)(a|b)(a|b))***

15. **Strings of a's and b's such that every block of 4 consecutive symbols has at least 2 a's.**

    Regex = **(aaxx | axax | axxa | xaax | xaxa | xxaa)* where x = (a|b)**

16. **L = $\{a^n b^m : n \geq 0, m \geq 0\}$** Regex = **a* b***

17. **L = $\{a^n b^m : n > 0, m > 0\}$** Regex = **aa* bb*** **OR** **a^+b^+**

18. **L = $\{a^n b^m : n \mid m \text{ is even}\}$** Regex = **aa* bb* | a(aa)* b(bb)***

19. **L = $\{a^{2n} b^{2m} : n \geq 0, m \geq 0\}$** Regex = **(aa)* (bb)***

20. **Strings of a's and b's containing not more than three a's.** Regex = **b* (ε | a) b* (ε | a) b* (ε | a) b***

21. **L = $\{a^n b^m : n \geq 3, m \leq 3\}$** Regex = **aaa a* (ε | b) (ε | b) (ε | b)**

22. **L = $\{ w : |w| \bmod 3 = 0 \text{ and } w \in \{a,b\}^* \}$** Regex = **((a|b)(a|b)(a|b))***

23. **L = $\{ w : n_a(w) \bmod 3 = 0 \text{ and } w \in \{a,b\}^* \}$** Regex = **b* a b* a b* a b***

24. **Strings of 0's and 1's that do not end with 01.** Regex = **(0|1)* (00 | 10 | 11)**

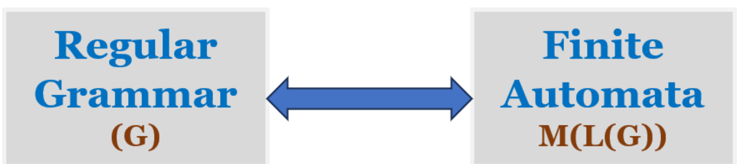
25. **L = $\{ vuv : u, v \in \{a,b\}^* \text{ and } |v| = 2\}$** Regex = **(aa | ab| ba | bb) (a|b)* (aa | ab | ba | bb)**

26. **Strings of a's and b's that end with ab or ba.** Regex = **(a|b)* (ab | ba)**

27. **L = $\{a^n b^m : m,n \geq 1 \text{ and } m,n \geq 3\}$** Regex = **a bbb b* | aaa a* b | aa a* bb b***

Equivalence of Regular Grammar and Finite Automata

The relationship of regular grammar and finite automata is shown below:



If G is a regular grammar, then L(G) is a regular language accepted by Finite automata.

Converting Finite Automata (DFA/NFA) to Regular Grammars

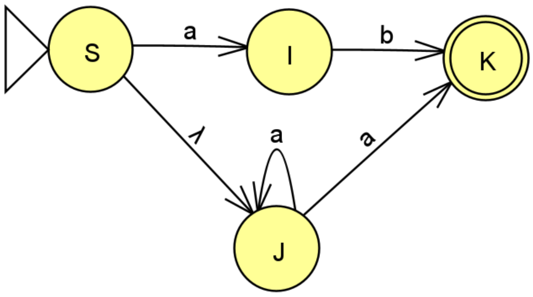
Algorithm: Finite Automata to Regular Grammar

Perform the following steps to construct a regular grammar that generates the language of a given Finite automata:

1. Rename the states to a set of uppercase letters (if the state names are named with q_0, q_1, q_2 ... or 1,2,3,...)
2. **The start symbol of the grammar is the FSM start state.** Each state in the FA becomes a non-terminal symbol in the grammar.
3. **Create Production Rules:**
    a. For each state transition from state A to state B labelled with a, create the production $\mathbf{A \rightarrow aB}$
    b. For each state transition from state A to state B labelled with λ, create the production $\mathbf{A \rightarrow B}$
4. **Handle Final States:** For each final state F, create a null production $\mathbf{F \rightarrow \lambda}$

Note: Every transition in a finite automaton corresponds to a production rule in a right-linear grammar. The grammar generates the same language as the automaton accepts.

Example: **Convert the following finite automata to regular grammar.**

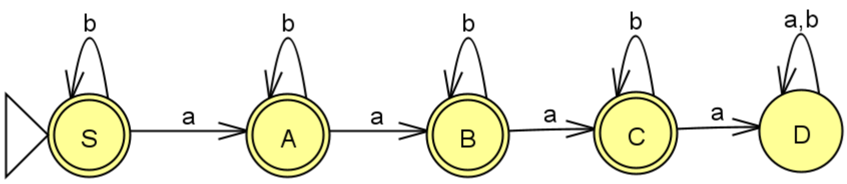


Regular Grammar:

- $S \rightarrow aI$
- $S \rightarrow J$
- $I \rightarrow bK$
- $J \rightarrow aJ$
- $J \rightarrow aK$
- $K \rightarrow \lambda$

Exercise:

1. Convert the following finite automata to regular grammar for the language to accept at most 3 'a's over the alphabet Σ={a, b}.



Solution:

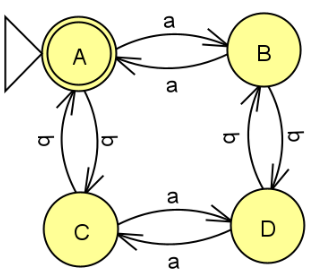
$S \rightarrow bS \mid aA \mid \lambda$
$A \rightarrow bA \mid aB \mid \lambda$
$B \rightarrow bB \mid aC \mid \lambda$
$C \rightarrow bC \mid a\mathbf{D} \mid \lambda$

State D is a trap (or dead) state and its rules are

$\mathbf{D \rightarrow aD \mid bD}$ (Not used in any string derivation, it's an useless production)

Note: The start symbol of the grammar is S, because the start state is S.

2. Convert the following finite automata to regular grammar.
Where `L = { nₐ(w) mod 2=0 and n_b(w) mod 2=0}`



Solution:

A → aB | bC | λ

B → aA | bD

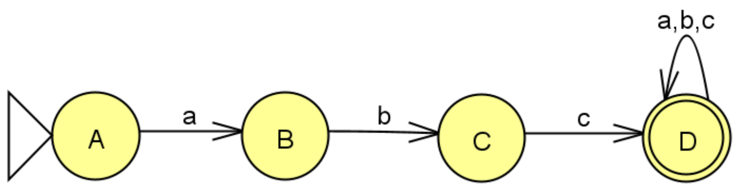
C → aD | bA

D → aC | bB

Note: The start symbol of the grammar is A, because the start state is A.

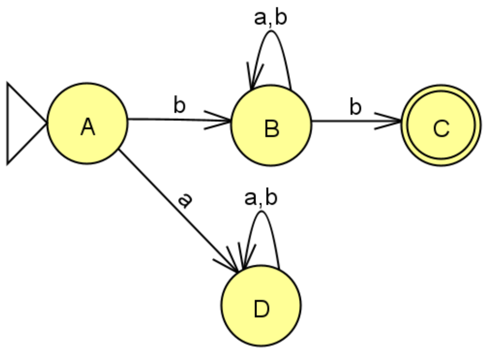
3. Convert the following finite automata to regular grammar.

Where `L = { abcw | where w∈{a,b,c}*}`



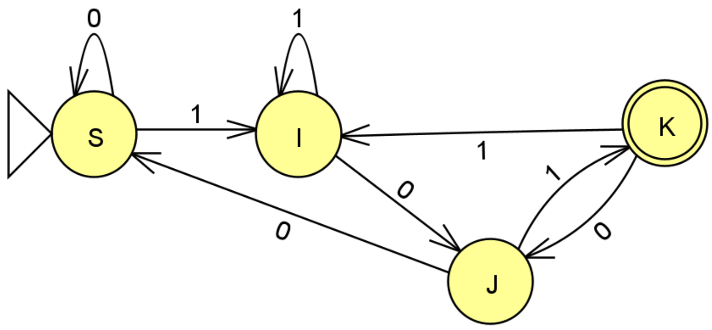
4. Convert the following finite automata to regular grammar.

Where L = { bwb | where w∈{a,b}*}



5. Convert the following finite automata to regular grammar.

Where language contains binary strings ends with 101.



Converting Regular Grammars to Finite Automata

Algorithm-01: Regular Grammar to Finite Automata

Perform the following steps to construct an Automata that accepts the language of a given regular grammar:

1. If necessary, transform the grammar so that all productions have the form A→x or A → xB, where x is either a single letter (i.e., terminal symbol) or λ
2. **Start State**: The start state of the Automata is the grammar's start symbol.
3. **Add Transitions**:
    a. For each production of the form A → aB, add a transition: δ(A, a) = B.
    b. For each production of the form A → B, add a transition: δ(A, λ) = B.
    c. If there are productions of the form A → a for some letter a, then create a single new final state with symbol F.
    For each production A → a, add a transition: δ(A, a) = F
    d. If there is a production of the form A → λ or A → ε (i.e., Null production), then mark A as an Accepting state.

Examples:

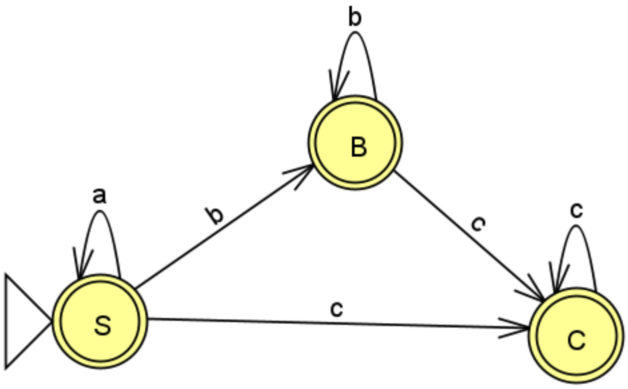
1) Convert the following regular grammar to finite automata.

S → aS | bB | cC | ε

B → bB | cC | ε

C → cC | ε

Solution:



L = { $a^n b^m c^k$ | n,m,k≥0}

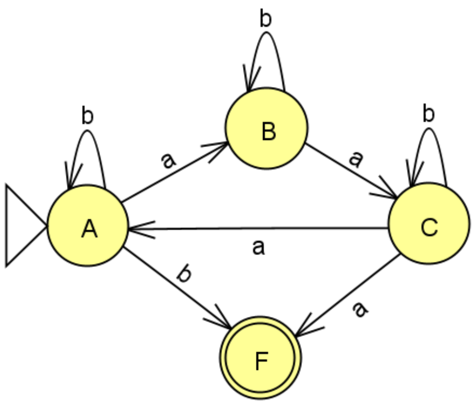
2) Convert the following regular grammar to finite automata.

A → aB | bA | b

B → aC | bB

C → aA | bC | a

Solution:



3) Convert the following regular grammar to finite automata.

S → bS | aA

A → aA | aB |bA

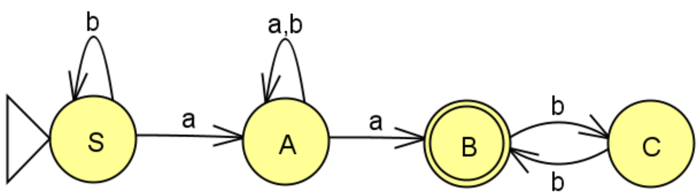
B → bbB

B → λ

Solution:

Transform the given grammar so that all productions have the form A→x or A → xB, where x is either a single letter (i.e., terminal symbol) or λ

S→bS | aA
A→aA | aB | bA
B→bC
C→bB
B→λ



Algorithm-02: Regular Grammar to Automata

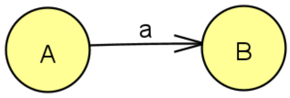
Assume that a regular grammar is given in its right-linear form, this grammar may be easily converted to a Automata. A right-linear grammar, defined by G = (V, T, P, S), may be converted to a automata, defined by M = (Q, Σ, δ, q0, F) by:

1. Create a state for each variable.

2. Convert each production rule into a transition.

a) If the production rule is of the form $\mathbf{V_i \rightarrow aV_j}$, where a ∈ T, add the transition $\boldsymbol{\delta(V_i, a) = V_j}$ to automata M.

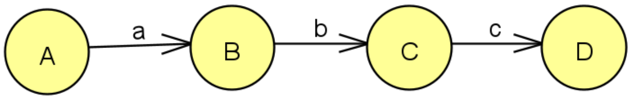
i) For example, **A → aB** becomes:



ii) For example, **A → aA** becomes:

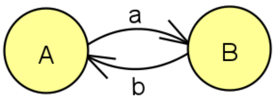
b) If the production rule is of the form $\mathbf{V_i \rightarrow wV_j}$, where w ∈ T*, create a series of states which derive w and end in $\mathbf{V_j}$. Add the states in between to set Q.

i) For example, **A → abcD** becomes:



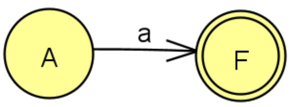
Note: For intermediate states, give new names (i.e., names that are not there in the Non-terminals list V of the given grammar).

ii) For example, **A → abA** becomes:

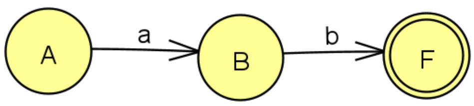


c) If the production rule is of the form $\mathbf{V_i \to w}$, where w ∈ T*, create a series of states which derive w and end in a final state.

i) For example, **A → a** becomes:



ii) For example, **A → ab** becomes:



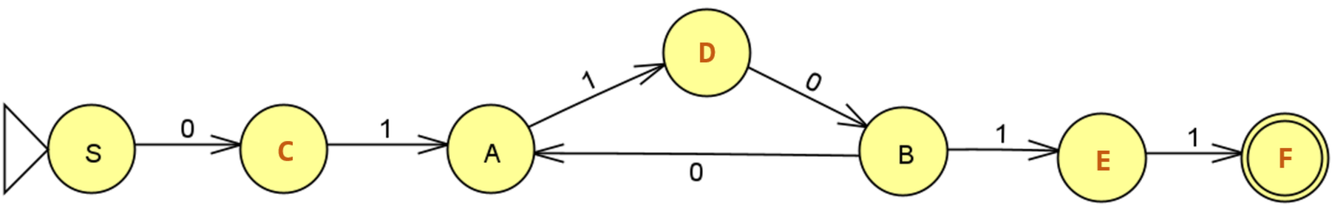
d) If the production rule is of the form $\mathbf{V_i \to \lambda}$ **or** $\mathbf{V_i \to \varepsilon}$ (i.e., Null production), then **state $\mathbf{V_i}$ is a final state.**

Examples:

1) Convert the following regular grammar to Automata.

S→01A
A→10B
B→0A|11

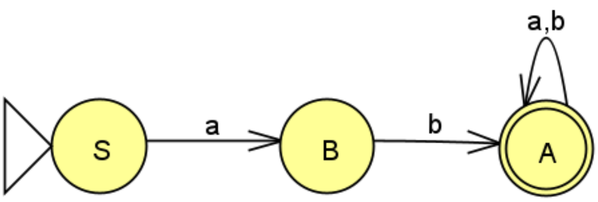
Solution:



2) Convert the following regular grammar to Automata.

S → aB
B → bA
A → aA | bA | ε

Solution:



Reverse of a Regular Language

If L is a regular language, then L^R is also regular

- Let L be a regular language, then there exists an automata M such that L = L(M).
  M = (Q, ∑, δ, q_0, F) (where machine M has a single final state)
- Construct a new machine M^R that accepts L^R (i.e., L^R=L(M^R)) by toggling initial and final state and by reversing the arrows (i.e., swapping initial and final state and changing the directions of the edges).
- The reverse of a regular language i.e., L^R is the language accepted by automata M^R.

Converting Right Linear Grammar to Left Linear Grammar

If G is a Right Linear Grammar, then there is a Left Linear Grammar G'' such that L(G) = L(G'')

Conversion outline:

a) From given Right-Linear Grammar G, construct Finite automata M that accepts L(G).
b) From Finite automata M construct M^R that accepts L^R
c) Generate G' a Right Linear Grammar from Finite automata M^R that accepts L^R
d) Generate G'' a Left Linear Grammar from G' for $(L^R)^R$ =L (i.e., getting G'' from G' by reversing all symbols on RHS of productions)

Example:

1) Convert the following Right Linear Grammar to Left Linear Grammar

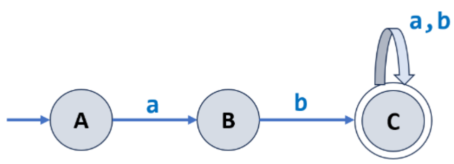
The Right Linear Grammar G is:

A → aB
B → bC
C → aC | bC | ε

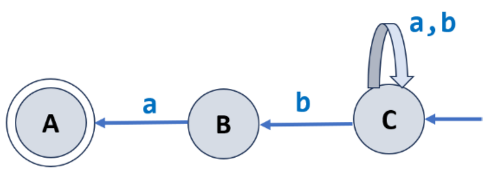
Where L(G) = { abw | w ∈ {a,b}*}

Solution:

a) From G, construct automata M that accepts L(G)



b) From Finite automata M construct M^R that accepts L^R



c) Generate G' a Right Linear Grammar from M^R that accepts L^R

C → aC | bC | bB
B → aA
A → ε

d) Generate G'' a Left Linear Grammar from G' for $(L^R)^R = L$ (i.e., getting G'' from G' by reversing all symbols on RHS of productions)

C → Ca | Cb | Bb
B → Aa
A → ε

Note: **L(G) = L(G'')**

Converting Left Linear Grammar to Right Linear Grammar

If G is a Left Linear Grammar, then there is a Right Linear Grammar G'' such that L(G) = L(G'')

Conversion outline:

a) From given Left-Linear Grammar G, Generate G' a Right-Linear Grammar (i.e., getting G' from G by reversing all symbols on RHS of productions)
b) From Right-Linear Grammar G' construct Finite automata M.
c) Construct Finite automata M^R from M.
d) Generate G'' a right Linear Grammar from Finite automata M^R

Example:

1) Convert the following Left Linear Grammar to Right Linear Grammar

Left Linear Grammar G is:

C → Ca | Cb | Bb
B → Aa
A → ε

Where L(G) = { abw | w ∈ {a,b}*}

Solution:

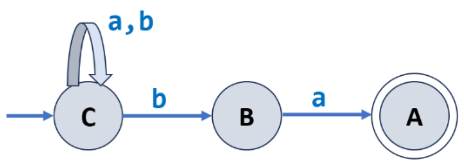
a) From given Left-Linear Grammar G, Generate G' a Right-Linear Grammar (i.e., getting G' from G by reversing all symbols on RHS of productions)

G':

C → aC | bC | bB
B → aA
A → ε

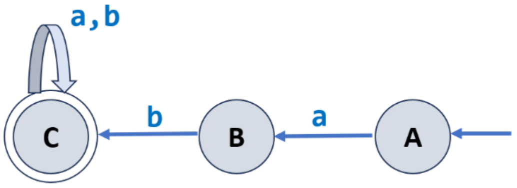
b) From Right-Linear Grammar G' construct Finite automata M.

M:



c) Construct Finite automata M^R from M.

M^R :



d) Generate G'' a right Linear Grammar from Finite automata M^R

A → aB
B → bC
C → aC | bC | ε

-------***-------

Note:

- Regular languages can be generated by non-linear grammars.
    - Regular language `L = {aa, ab, ba, bb}`
    - Non-linear Grammar that generates the above regular language `L is`
        - S → AA
        - A → a|b

      Regular languages can be generated by regular grammars, which are even more restricted (e.g., only one non-terminal and terminal per rule, in a specific order). **But using a non-linear grammar doesn't prevent it from generating a regular language—it just means the grammar itself isn't regular.**

- But not all languages generated by non-linear grammars are regular.