Biskra University

Computer Science Department

Théorie des langages(TL)

Formal Languages (2025)

By

Pr. CHERIF Foudil

Course Program: Formal Languages

1- Introduction to formal logic
2- Introduction to languages
3- Typology of grammars
4- Regular languages (**Type 3**)
    a- Regular grammars
    b- Finite state automata
    c- Regular expressions
5- Algebraic languages (free context) (**Type 2**)
    a- Transformation of grammars (empty word, recursion, ..)
    b- Chomsky's Grammar
    c- Greibach's Grammar
    d- pushdown automata
6- **Type 1**: Contextual languages and linear terminal automata
7- **Type 0** languages and Turring machines

Bibliography

1.Dexter Kozen. **Automata and Computability** (1997).

2.Michael Sipser. **Introduction to the Theory of Computation, 3rd ed. (2012).**

3.John Hopcroft and Jeffrey Ullman. **Introduction to Automata Theory, Languages, and Computation,** 1st ed. (1979). .

4.Jeffrey Shallit. **A Second Course in Formal Languages and Automata Theory (2008).**

5. P. Wolper. **Introduction à la calculabilité**. 2006, Dunod.

6. P. Séébold. **Théorie des automates**. 2009, Vuibert.

7.J.M. Autebert **Théorie des langages et des automates**. 1994, Masson.

8. J. Hopcroft, J. Ullman. **Introduction to Automata Theory, Languages and Compilation** 1979, Addison- Wesley

Motivation of the course

The objective of this course is to introduce the theory of formal languages.

Languages allow humans to exchange information and ideas and to communicate with machines.

The languages used between humans are called **'natural languages'**, they are usually **informal** and **ambiguous** and require interpretation by a human brain to be interpreted correctly.

The languages created by humans to communicate with the machine are the **formal languages** or **artificial languages**. They must be formalized and unambiguous in order to be interpreted by a machine, this is the goal of this course.

Chapter 1:

Introduction to formal logic

Chapter 1: Introduction to formal logic

1. Definition of formal systems

A formal system is a set of data which makes it possible to manipulate a set of symbols by considering only their syntax (structure) without taking into account their semantics (meaning, interpretation).

A formal system consists of a **syntax:**

1. A finite alphabet of symbols
2. A formula construction process for describing well-formed formulas of this system (language).

Example: An alphabet V= { a, b, c }

The well-formed formulas: sequence of letters of V containing the letter **a only one time**, and the letter **c only one time** and **b is before c**

Chapter 1: Introduction to formal logic

2- Introduction to the theory of languages (formal languages):

The language theory defines programming languages, but compilation transforms programs written in these languages into machine code.

The source program is transformed into:

1. absolute machine language (directly executable)
2. translatable machine language (requires linking)
3. assembly language (requires assembler)
4. high-level language (requires a compiler)

The basic structure for the theory of languages is the **monoid** (is a structure equipped with an operation)

Chapter 1: Introduction to formal logic

A language is defined on a set called vocabulary (characters or symbols) is a subset of finite strings of characters.

A **language** is defined by a **grammar**.

Automata are symbolic machines validating the membership of a given string in the language it describes (all these notions will be studied in the following chapters.

1.finite state automata (type 3)

2.Pushdown automata (type 2)

3.linear terminal automata (type 1)

4.Turing's machines (type 0)

Chapter 1: Introduction to formal logic

3- Monoid structure:

A monoid is a structure with the composition law is **associative.**

Associativity: The operation must be associative, meaning that for any elements a, b, and c in the set, (a * b) * c = a * (b * c).

We call **free monoid** any monoid having an **identity element**.

Identity element: There must exist an element in the set, called the identity element, such that for any element a in the set, a * identity = identity * a = a.

The example that interests us in our course is the set of finite character strings on a finite vocabulary, this set is provided with the operation of **concatenation** which is associative and which has an identity element, **the empty string**

3.1 Vocabulary

A vocabulary V or alphabet is a finite set of letters or symbols called letters (letters, numbers or other symbols)

Examples:

1) V = { a1, a2,…,an) V: the alphabet ai: the letters

2) V= { 1 } one-letter alphabet

3) V= { 0, 1} binary alphabet

4) V ={ ., - , / } Morse code for transmission

5) V = { 0,1,…9,a,b,…z } any alphabet

3.2 Monoide V^+

We call **monoid V^+** the set of all the strings **of non-empty finite lengths** defined on V. These strings are called words and the set V^+ is infinite.

In other words V^+ is the set of words of length greater than or equal to 1 that can be constructed from the alphabet V

Exemple:

$V = \{ a, b \}$

$V^+ = \{ a,b,aa,bb,ab,ba,bb,aaa,\ldots \}$

3.3 Concatenation operation

The concatenation operation consists in juxtaposing two words in order to obtain a new word. It is **associative** but **not commutative** operation.

$x, y \in V^+$ x and y are two words

$x = x_1.x_2 \ldots x_k$ / $x_i \in V$

$y = y_1.y_2 \ldots .y_p$ / $y_i \in V$

$x.y = x_1.x_2 \ldots x_k \; y_1.y_2 \ldots .y_p$

$(x.y).z = x.(y.z)$. Is associative

$x.y \neq y.x$. is not commutative

3.4 Free Monoïde V^*

The concatenation operation admits an identity element which is the empty string(length equal to zero) and denoted by ε, $x.\varepsilon = \varepsilon.x = x$

We can define $V^* = V^+ \cup \{\varepsilon\}$

3.5 Word length

The length of a word x which is generally noted |x| matches each word with the number of symbols it contains.

We define a particular word called empty word, this word is not composed of any character, its length is therefore zero ($|\varepsilon| = 0$).

3.5 Subword

We say that y Є V* is a subword (or factor) of x Є V* if there exist finite words u, v Є V* such that x = u y v and $|y| \leq |x|$

If x = **y** v we say that y is left factor or **prefix** of x

If x= v **y** we say that y is right factor or **suffix** of x

Example

V= { a, b }

x = ab**bbb**aa and y = bbb so x = ab**y**aa subword

x = bbbaa and y = bbb so x = yaa left factor

Chapter 2: Introduction to languages

Chapter 2: Introduction to languages

1. Définition

A language on a vocabulary V is a subset of the words defined over V, in other words a language is a part of the free monoid V* .

$L \subset V^*$

We can differentiate between the empty language (L = Ø) and the language containing the only empty word (L = {ε })

Example :

V= { a, b }

V^* = { ε ,a,b,aa,bb,ab,ba,bb,aaa,… }

L = { aa, bb, ab, ba } the set of words on V* of length equal to 2

2. Syntax of a language

A sentence is well-formed if and only if it belongs to the language.

The syntax of a language is the set of constraints(rules) which make it possible to define the sentences of this language.

Example: A simple measurement language can be defined by the following constraints:

- **measure** followed by an **operator** and a **measure**
- a measure is the number **1** followed by a **unit**
- Units are : **cm , liter, km**
- The opérateur is +

2. Syntax of a language

- **measure followed by an operator and a measure**
- **a measure is the number 1 followed by a unit**
- **Units are : cm , litre, km**
- **The operator is +**

The well-formed sentences are:

1cm + 1cm	1cm + 1liter	1cm + 1km
1liter + 1cm	1liter + 1liter	1liter + 1km
1km + 1cm	1km + 1liter	1km + 1km

3. Sémantic of a language

The semantics of a language is a set of constraints on this language.

Among the well-formed sentences in the measurement example, only a few are semantically correct, those whose units are of the same type (measure or weight). These sentences are:

1cm + 1cm	1cm + 1km	1liter + 1liter
1km + 1cm	1km + 1km	

4. Opérations on languages

As we have seen, languages are sets of words. The usual operations concerning sets such as **union**, **intersection** and **complementation** are applicable to languages.

Consider two languages L_1 and L_2 respectively defined on the two alphabets V_1 and V_2.

a) Union

The union of L_1 and L_2 is the language defined on $V_1 \cup V_2$ containing all the words which are either contained on L_1 or contained on L_2:

$$L_1 \cup L_2 = \{ x / x \in L_1 \text{ or } x \in L_2 \}$$

b) Intersection

The intersection of L_1 and L_2 is the language defined on $V_1 \cap V_2$ containing all the words which contained on L_1 and on L_2:

$$L_1 \cap L_2 = \{ x / x \in L_1 \text{ and } x \in L_2 \}$$

4. Opérations on languages

c) Complementation

The complement of L_1 is the language defined on V_1 containing all the words which are not in L_1.

$$C(L_1) = \{ x / x \notin L_1 \}$$

d) Difference

The difference of L_1 and L_2 is the language defined on L_1 containing all the words of L_1 which are not in L_2.

$$L_1 - L_2 = \{ x / x \in L_1 \text{ and } x \notin L_2 \}$$

4. Opérations on languages

e) Concatenation

The concatenation of L_1 and L_2 is the language defined on $V_1 \cup V_2$ containing all the words made up of a word from L_1 followed by a word from L_2.

$L_1 L_2 = \{ xy / x \in L_1 \text{ and } y \in L_2 \}$

The concatenation operation is not commutative $L_1 L_2 \neq L_2 L_1$

f) Power

The power of a language is defined by:

$L^0 = \{\varepsilon\}$ $\quad\quad L^{n+1} = L^n L$

4. Opérations on languages

g) Kleen closure

The **iterative closure** of L or **Kleen closure** (**iterate of L**, or **Kleen star**) is the set of words formed by a finite concatenation of words of L.

$L^*=\{x \ / \ \exists \ k \geq 0$ and $x_1, x_2,\ldots,x_k \in L$ such as $x= x_1x_2\ldots x_k \}$

$L^* = \{ \varepsilon \} \cup L \cup L^2 \cup L^3 \ldots \cup L^n \cup \ldots$

We can similarly define the **positive Kleene closure** of L by:

$L^+=\bigcup_{i\geq1}L^i = L \cup L^2 \cup L^3 \ldots \cup L^n \cup \ldots$

4. Opérations on languages

h) Reversal (R or ~)

The reversal of a string $w = x_1x_2 \ldots x_k$ the string with the symbols written in reverse order, $w^R = x_k x_{k-1} \ldots . x_2 x_1$

Formally,

$L^R = \{ x^R / x \in L \}$

a) if $w=\varepsilon$, then $\varepsilon^R=\varepsilon$ and

b) if $w= ax$ for a symbol $a \in V$ and a string $x \in V^*$, then $(ax)^R=x^R a$

If $w = w^R$, we say w is a **palindrome**

4. Opérations on languages

i) Remarks:

$L^+ = L\, L^*$

$L^*L^* = L^*$

$(L^R)^R = L$

$(L^*)^R = L^*$

$(L_1 L_2)^R = L_2^R L_1^R$

Chapter 2: Introduction to languages

Examples :

Let L_1, L_2 and L_3 be three languages defined by:

$L_1 = \{\varepsilon, aa\}$, $L_2 = \{a^i b^j / i, j \geq 0\}$ and $L_3 = \{ab, b\}$.

Calcule :

$L_1.L_2$, $L_1.L_3$, $L_1 \cup L_2$, $L_2 \cap L_3$, L_1^{10}, L_1^*, L_2^R

Solutions :

- $L_1.L_2 = L_2$;
- $L_1.L_3 = \{ab, b, aaab, aab\}$;
- $L_1 \cup L_2 = L_2$;
- $L_2 \cap L_3 = L_3$;
- $L_1^{10} = \{a^{2n} / 10 \geq n \geq 0\}$;
- $L_2^R = \{b^i a^j / i, j \geq 0\}$.

Find:

$L_1 = \{w \in \{a,b\}^* / |w|_a = |w|_b\}$

Chapter 3:
Typology of grammars

Chapter 3: Typology of grammars

1- Definition of a grammar

With a grammar, we describe in a generic and productive way the well-formed expressions of a language.

A grammar is a formal system defined by an **axiom** and **rules** called **production rules**.

The sentences are **derived** from the axiom and by successive application of the rules.

The rules of the grammar are constructed with effective symbols called **terminal symbols** and tool symbols called **non-terminal symbols**, which denote pieces of correct strings during language construction.

Axiom , rules, terminal symbols, non-terminal-symbols

Chapter 3: Typology of grammars

Example1: Consider the following sentence: $-19.5\ 10^{-3}$

L: set of numbers of this form (decimal numbers)

ND: A decimal number

We can form a grammar that generates the set L of decimal numbers as follows:

ND → S E P E F

E → C E

E → C

C → 0 / 1 / 2 / 3 / 4 / 5 / 6 / 7 / 8 / 9

P → .

F → 10 S E

S → + / -

Chapter 3: Typology of grammars

Example 2: **English grammar** :

sentence → <subject> <verb-phrase> <object>

subject → This / Computers / I

verb-phrase → <adverb> <verb> / <verb>

adverb → never

verb → is / run / am / eat / tell

object → the <noun> / a <noun> / <noun>

noun → university / world / cheese / mouse / lies

Using these rules, we can derive simple sentences like:

This is the university

Computers run the world

the cheese eat the mouse

I never tell lies.

Chapter 3: Typology of grammars

Example 2: **English grammar :**

Derivation of the first sentence :

<sentence> → <subject> <verb-phrase> <object>

→ This <verb-phrase> <object>

→ This <verb> <object>

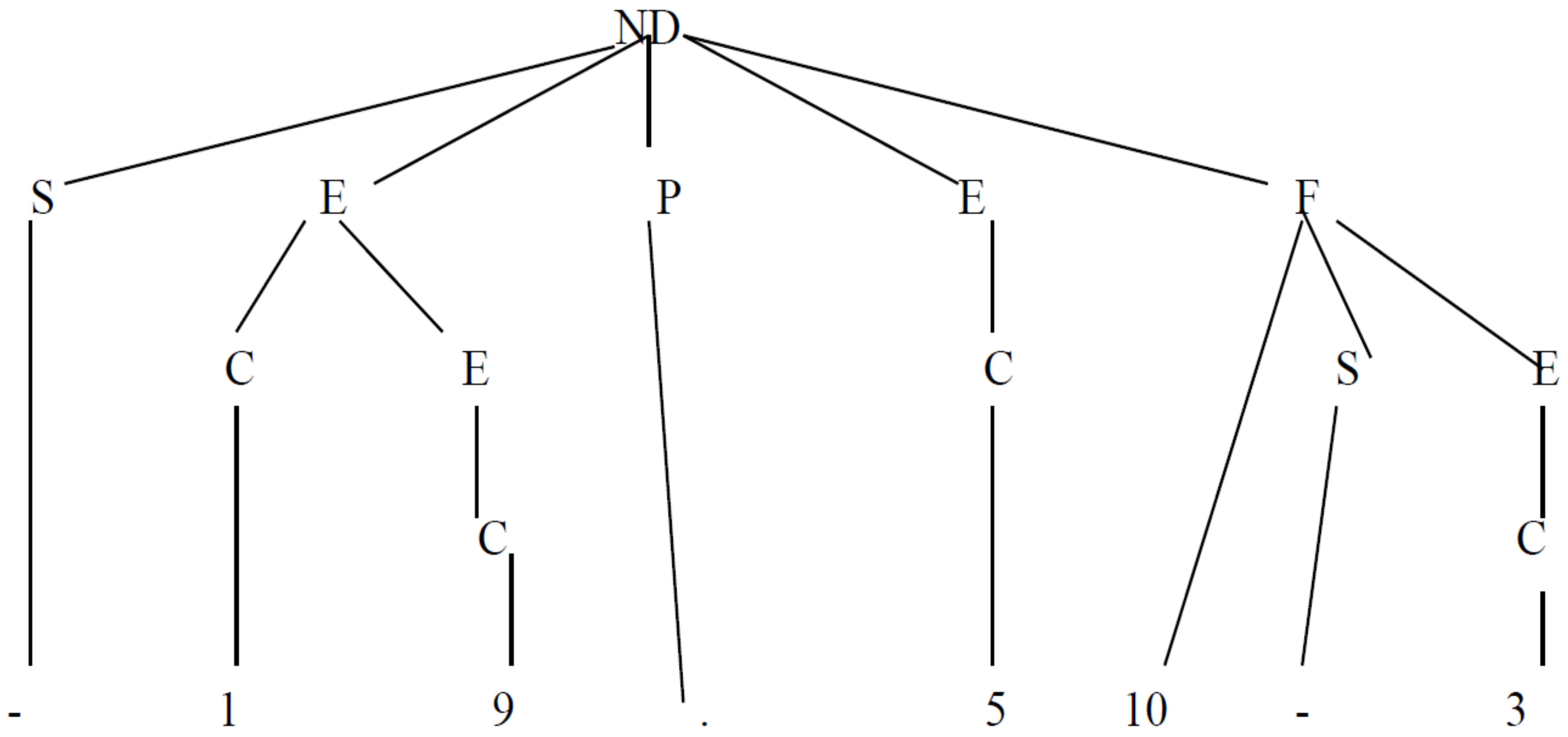
→ This is <object>

→ This is the <noun>

→ This is the university

Chapter 3: Typology of grammars

A decimal number is defined by a **derivation tree**



Chapter 3: Typology of grammars

Concepts to be defined:

1.Decimal number: ND is the **root** of the **derivation tree**

2.Syntactic elements: (ND, S, E, P, F, C) **Non-terminal vocabulary**

3.Initial alphabet: (0,1,2,3,4,5,6,7,8,9,+,-,.,10) **Terminal vocabulary**

4.Set of rules: It is the **articulation** of the different elements between them.

Chapter 3: Typology of grammars

2- Formal definition of a grammar

A grammar is a quadruple G = (V_t, V_n, S, R) where

- **V_t**: is the terminal vocabulary. Is a non-empty finite set.
- **V_n**: is the non-terminal vocabulary, the set of symbols which do not appear in the generated words, but which are used during the generation. Is a non-empty finite set.
- **S**: is an element of V_n, is the **starting symbol** or **axiom**. It is from this symbol that we will begin the generation of words using the rules of the grammar.

Chapter 3: Typology of grammars

2- Formal definition of a grammar

A grammar is a quadruple $G = (V_t, V_n, S, R)$ where

- R: is a set of so-called rewriting or production rules of the form:
  - $u \rightarrow v$ such as $u \in (V_t \cup V_n)^+$ and $v \in (V_t \cup V_n)^*$
  - and $V_t \cap V_n = \emptyset$

Terminology :

- A sequence of terminal and non-terminal symbols (an element of $(V_t \cup V_n)^*$ is called a **form**.
- A rule $u \rightarrow v$ such that $v \in V_t^*$ is called a **terminal rule**.

Chapter 3: Typology of grammars

<u>3- Grammar derivation</u>

- **Direct derivation : "====> "**

Let a rule of R $u \rightarrow v$ and let x, y two words of $(Vt \cup Vn)^*$

we say that y derives directly from x in G (x ===> y) if and only if

$x = \alpha u \beta$ and $y = \alpha v \beta$ $\alpha, \beta \in (Vt \cup Vn)^*$

- **Indirect derivation: "==*==> "**

We say that y derives indirectly from x in G (x ==*=> y) if and only if

it exists a finite sequence $W_0, W_1, \ldots, W_n$ such as $W_0 = x$ $W_n = y$

and W_i ====> W_{i+1} $0 \leq i \leq n$

x ===> W_1 …. W_{n-1} ====> y

Chapter 3: Typology of grammars

- **example:**

4- Language generated by a grammar

The language defined, or generated, by a grammar is the set of words that can be obtained from the starting symbol(axiom) by applying the rules of the grammar.

More formally is the set of terminal derivations of the axiom.

G = (Vt, Vn, S, R) L(G) = { x / x Є Vt^* and S $==^*==>$ x }

Chapter 3: Typology of grammars

Important remark:

A grammar defines a single language. On the other hand, a language can be generated by several different grammars.

These two **grammars are equivalent**.

We say that G_1 and G_2 are equivalent if and only if $L(G_1) = L(G_2)$

Important remark: equivalent grammars

Let the grammar G1 = ({a,b}, {S,X} , S , {S→aXa , X→bX| ϵ})

➔ L(G1) ={ ab^*a }. Unique language

But the language L ={ ab^*a } can be generated by these 3 different grammars:

G1 = ({a,b}, {S,X} , S , {S→aXa , X→bX| ϵ }) ;

G2 = ({a,b}, {S,X ,Y} , S , {S→aY , Y→Xa , X→bX| V}) ;

G3 = ({a,b}, {S,X} , S , {S→aX , X→bX|a }) ;

L(G1)=L(G2)=L(G3) ➔ G1 , G2, G3 are equivalent,

Chapter 3: Typology of grammars

Examples: languages construction

Find the languages generated by these grammars:

1. G_1=({a,b}, {S}, S, R)

   R = (S → a S b , S → ab)

2. G_2=({_/ , _}, {S, A, U, V}, S, R)

   R = (S → U A V S → U V A → V S U

   A → V U U → _/ V → _)

Chapter 3: Typology of grammars

3. G_3=({a,b}, {S, A, B}, S, R)

R = (S → AS   S → Ab   A → A B   B → a A   )

4. G_4=({a}, {S}, S, R)

R= (S → A S A   S → ϵ   A → S A   A → A S a   )

Examples of languages :

- $L_1 = \{ab, a, ba, bb\}$;
- $L_2 = \{\omega \in \{a, b\}* / |\omega| > 3\}$;
- $L_3 = \{\omega \in \{a, b\}* / |\omega| \equiv 0\ [5]\ \}$;

Chapter 3: Typology of grammars

L_1 = {ab, a, ba, bb}

G=({a,b}, {S, A, B}, S, R)

R = (S → aA   S → b B   A → b   A → ϵ   B → a   B → b)

L_2 = {ω ∈ {a, b}* / |ω| > 3}

G_4=({a,b}, {S,A,B}, S, R)

R= (S → A AAAB   A → a / b   B → AB   B → ϵ)

L_3 = { ω ∈ {a, b}* / |ω| ≡ 0 [5] } or L_3 = { ω ∈ {a, b}* / |ω| ≡ 0 **mod 5** ***}***

R= (S → A AAAAS   A → a / b   S → ϵ)

Chapter 3: Typology of grammars

5- <u>Types of grammars</u>

By introducing more or less restrictive criteria on the production rules, we obtain hierarchical classes of grammars, ordered by inclusion. The classification of grammars, defined in 1957 by **Noam CHOMSKY**, distinguishes the following four classes:

Chomsky in 2017

Born in 1928 (age 97)

Philadelphia, Pennsylvania, U.S.

5.1- Grammar type 0

Grammars without restriction on rules,

so all grammars are type 0.

$u \rightarrow v \qquad u \in (Vn \cup Vt)^{+} \quad \text{and} \quad v \in (Vn \cup Vt)^{*}$

Chapter 3: Typology of grammars

5.2 Grammar type 1:

a) context sensitive grammars

Type 1 grammars are also called **context sensitive** or **context sensitive grammars**.

The grammar rules are of the form:

$$\mathbf{u\ A\ v \rightarrow u\ W\ v}$$

$\mathbf{A \in Vn}$, $W \in (Vn \cup Vt)^{+}$ and $u,v \in (Vn \cup Vt)^{*}$

In other words, the **non-terminal symbol A** is replaced by the **form W** but if we have the **contexts u on the left and v on the right**.

We restrict the rules by forcing the right side to be at least as long as the left side**.**

$$\mathbf{| u\ A\ v | \leq | u\ W\ v |}$$

Chapter 3: Typology of grammars

5.2 Grammar type 1:

a) context sensitive grammars

This forces the **empty word** to be excluded from the grammar.

But we accept the rule **S →** ϵ with this condition : « the non terminal S does not exist in the right of each rule of the grammar

b) Monotone grammar (not-decreasing grammar)

$\alpha \rightarrow \beta$ where $|\alpha| \leq |\beta|$

Remark: ***All context sensitive grammars are monotones but monotone grammars are not necessary context sensitive grammars***

Chapter 3: Typology of grammars

5.2 Grammar type 1:

Grammar 1: Monotone and not **context sensitive**

S → aSBc

S → abc

cB → Bc this rule not **context sensitive**

bB → bb

Grammar 2: Monotone and **context sensitive**

S → aSBC S → aBC CB → HB HB → HC HC→ BC

aB → ab bB → bb bC → bc cC → cc

Grammar 1 and grammar 2 are equivalent have le same language

Chapter 3: Typology of grammars

5.3- Grammar type 2:

Type 2 grammars are also called **context-free**, **algebraic** or **Chomsky grammars**. It is the most widely used grammar in language theory and compilation.

The grammar rules are of the form:

$$\mathbf{A \rightarrow W}$$

$A \in Vn$, $W \in (Vn \cup Vt)^*$

In other words, the left member consists of a single non-terminal symbol.

Chapter 3: Typology of grammars

5.4- Grammar type 3:

Type 3 grammars are also called **regular grammars** on the right (respectively on the left), **linear grammars**.

The grammar rules are of **one** of these 02 forms: A, B ϵ Vn and a ϵ Vt

Form1:	A → **a B**	**Form2:**	**A → B a**
or	**A → a**	**or**	**A → a**
or	**A → ϵ**	**or**	**A → ϵ**

The left member of each rule consists of a single non terminal symbol, and the right member consists of a terminal symbol possibly followed (respectively preceded) by a single non terminal.

Chapter 3: Typology of grammars

Examples:

G=({a,b,c}, {S, A, B, C,D}, S, R)

R1= (S $\rightarrow$ ACaB   AB $\rightarrow$ AbBc   A $\rightarrow$ bcA   B $\rightarrow$ b)

R2= (S $\rightarrow$ ACaB   Bc $\rightarrow$ acB   CB $\rightarrow$ DB   aD $\rightarrow$ Db)

R3= (S $\rightarrow$ aAB   B $\rightarrow$ aAB   aA $\rightarrow$ aaA   bbAa $\rightarrow$ bbBa   A $\rightarrow$ bcA   B $\rightarrow$ ϵ)

Chapter 3: Typology of grammars

6- Language type:

Each type of grammar is associated with a type of language:

Type 3 grammars generate regular languages,

type 2 grammars generate context-free languages

type 1 grammars generate contextual languages

and type 0 grammars generate all "**decidable**" languages, in other words, all languages that can be recognized in **finite time** by a machine.

Languages that cannot be generated by a type 0 grammar are called "**undecidable**".

Chapter 3: Typology of grammars

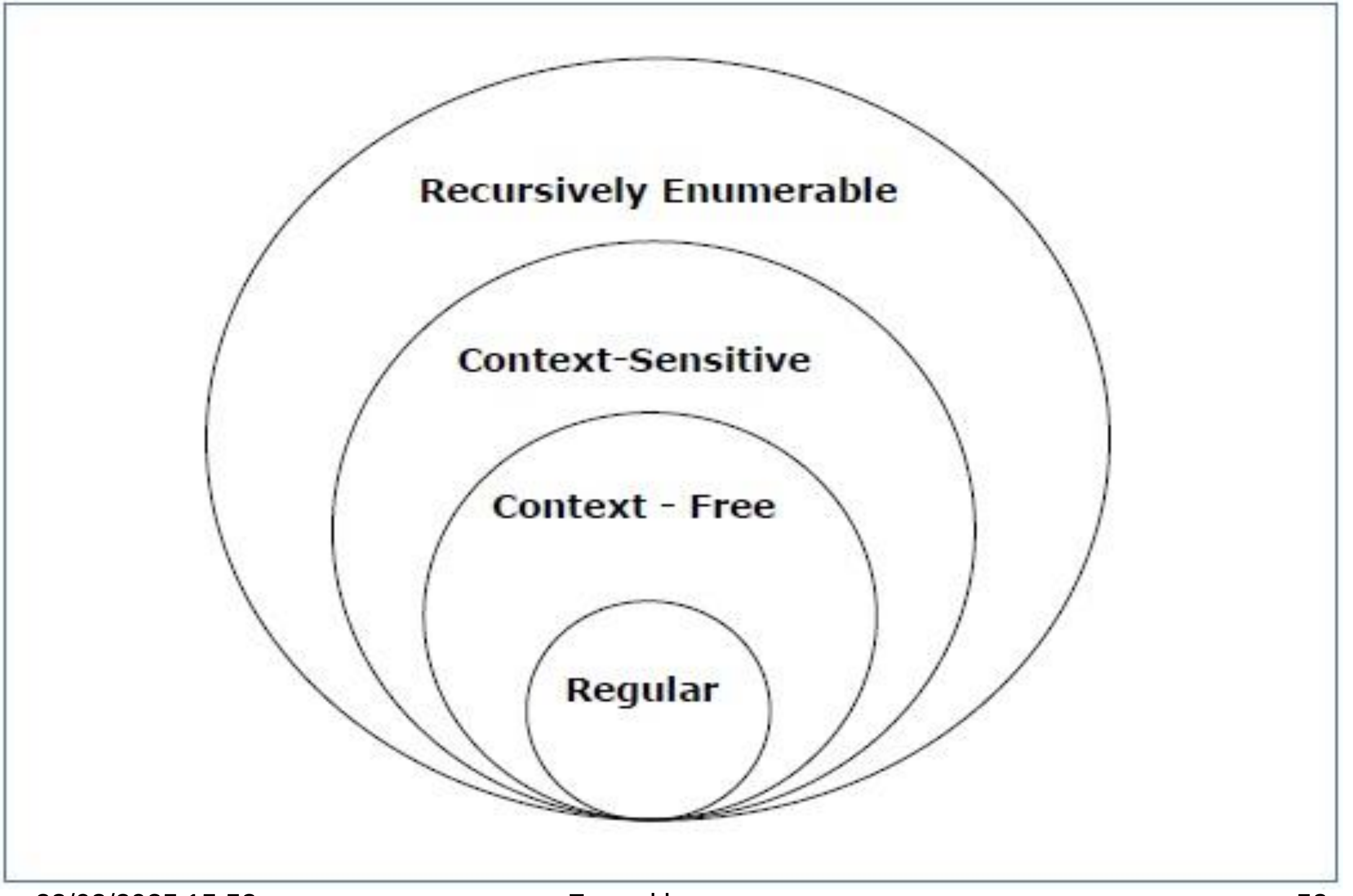
Language type:

These languages are ordered by inclusion: the set of languages generated by type n grammars is strictly included in that of type n-1 grammars (for n = 1,2,3).

Examples :

- a type 3 grammar is also type 2, 1, 0
- a type 2 grammar is also type 1, 0   but not type 3
- a type 1 grammar is also type 0

Chapter 3: Typology of grammars



Chapter 3: Typology of grammars(conclusion)

Each type of grammar is associated with a type of automata which makes it possible to recognize the languages of its class:

- regular languages are recognized by **finite automata**

- context-free languages are recognized by **push down automata**

- Contextual languages are recognized by **linear bounded machines**

- and type 0 languages are recognized by **Turing machines**

The Turing machine is considered the most powerful model, where any language that cannot be processed by one Turing machine, cannot be processed by another machine.

Chapter 4:
Regular languages

Chapter 4: Regular languages

4.1 Regular grammar definition:

Type 3 grammars are also called **regular grammars** on the right (respectively on the left), **linear grammars**.

The grammar rules are of the form:

A → **a B** **(respectivement A → B a)**

or **A → a**

or **A → є**

A, B є Vn et a є Vt

Chapter 4: Regular languages

Regular languages are languages generated by regular grammars.

Regular grammars are used in the lexical analysis phase of compilation.

Example:

G=({a,b}, {S,A}, S, R)

R=(S $\rightarrow$ a S

S$\rightarrow$ bA

A $\rightarrow$ a)

4.2 Automata definition **(automata plural or automaton singular)**

In formal language, we have two systems:

- The generation systems which are the **grammars**

- The recognition systems which are the **automata**

Automata is a virtual machines (**programs**), which takes as input a string of symbols and performs a string recognition algorithm.

If the algorithm terminates under certain conditions, the automata accepts this string.

The language recognized by an automata is the set of strings it accepts.

Fields of application: compilation and real-time applications

4.2 Automata definition (Fields of application)

Finite automata has several applications in many areas such as:

compiler design, special purpose hardware design, real-time applications ,protocol specification,…

a) Compiler Design

Lexical analysis is an important phase of a compiler. A program such as a C program is scanned and the different tokens (constructs such as variables, keywords, numbers) in the program are identified.

4.2 <u>Automata definition (</u>Fields of application)

b) Vending Machines:

A vending machine is an automated machine that dispenses numerous items such as cold drinks, snacks, etc. to sale automatically, after a buyer inserts currency or credit into the machine.

Vending machine works on finite state automata to control the functions process.

c)Text Parsing:

Text parsing is a technique which is used to derive a text string using the production rules of a grammar to check the acceptability of a string

4.2 <u>Automata definition (</u>Fields of application)

d)Traffic Lights:

The optimization of traffic light controllers in a city is a representation of handling the instructions of traffic rules. Its process depends on a set of instruction works in a loop with switching among instruction to control traffic,

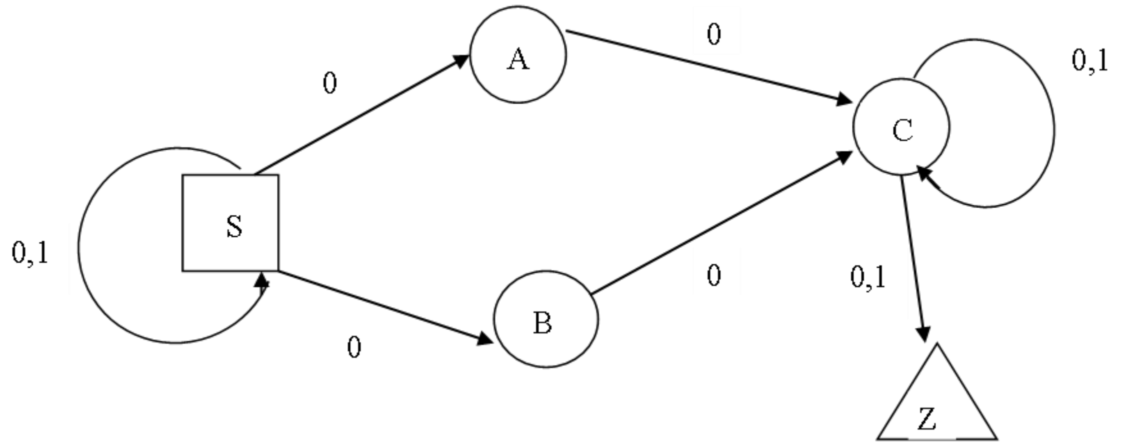
e)Video Games:

Video games levels represent the states of automata. In which a sequence of instructions are followed by the players to accomplish the task

4.3 Finite State Automata (FSA)

4.3.1 Definition

A FSA is composed of a finite set of states (represented graphically by **circles**), a transition function describing the action that allows to pass from one state to another (**these are the arrows**), an initial state (the state denoted by **square**) and one or more final states (denoted by **triangles**).

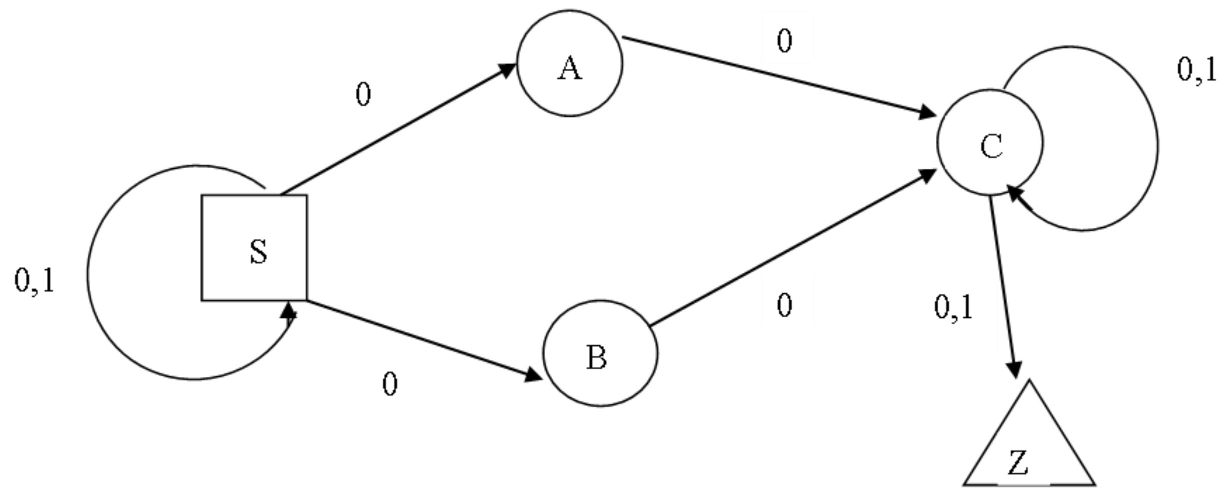


4.3 Finite State Automata (FSA)

4.3.1 Definition

A FSA is therefore a directed and valued graph where the nodes correspond to the states and the values of the arcs to the terminal symbols,

FSA does not use any memory to recognize a string.



4.3.2 Formal definition

A finite state automata is a quintuplet:

$A = (V_t, Q, q_0, f, F)$

- V_t : is the terminal vocabulary, non-empty finite set of symbols
- Q : is the state set of the automata, non-empty finite set
- q_0 : The set Q contains a particular state q_0 called initial state. $q_0 \in Q$
- F : The set Q contains a subset of particular states F called final states. $F \subset Q$
- f : is an application of $Q \times V_t \cup \{ \epsilon \} \rightarrow Q$

The automata stops on a final state or the complete reading of the input string.

Representations of a FSA

There are three main representations for a FSA:

- The matrix representation,
- A directed and weighted graph
- Or transition functions (relations)

a) Transition function

f is the transition function of A $f(q, a) = q_1$

Indicates that if the automata is in state q and it encounters the symbol a, it goes to state q_1.

Moreover for all q of Q $f(q, \epsilon) = q$

Example:

Let A be the FSA defined by the quintuplet (Vt, *Q*, *q0*, *f*, *F*) such that:

Vt =$\{a, b\}$,

Q= $\{q_0, q_1\}$,

q_0 : initial state

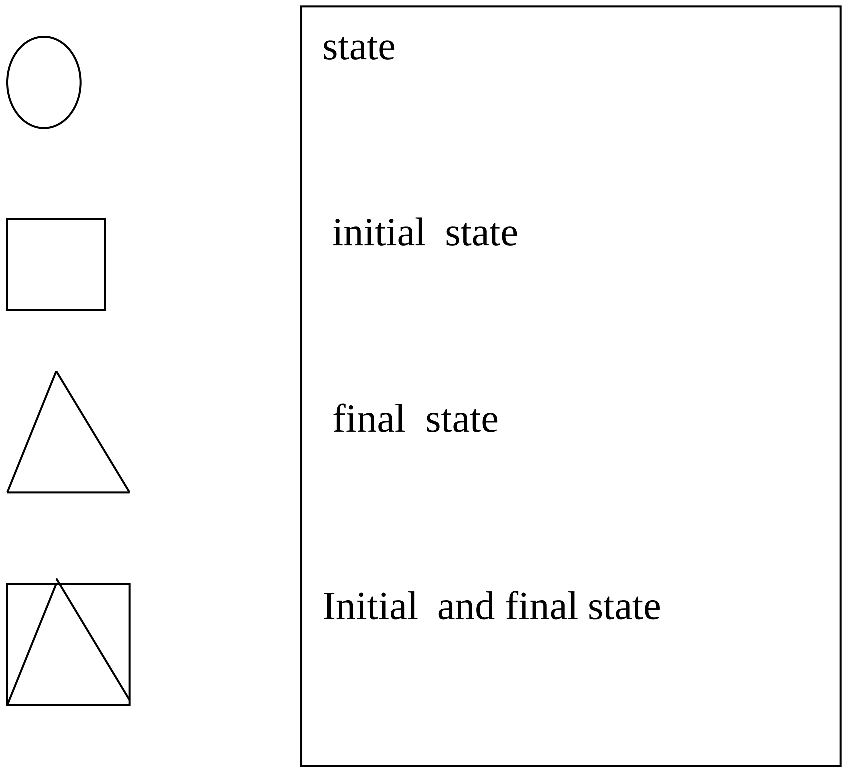
F= $\{q_1\}$

and we define the following transitions:

$f(q_0, a) = q_0$, $f(q_0, b) = q_1$, $f(q_1, b) = q_1$

b) Directed Graph :

We represent a finite state automata by a directed and valued graph, whose arcs carry symbols of V_t and whose nodes carry the states.

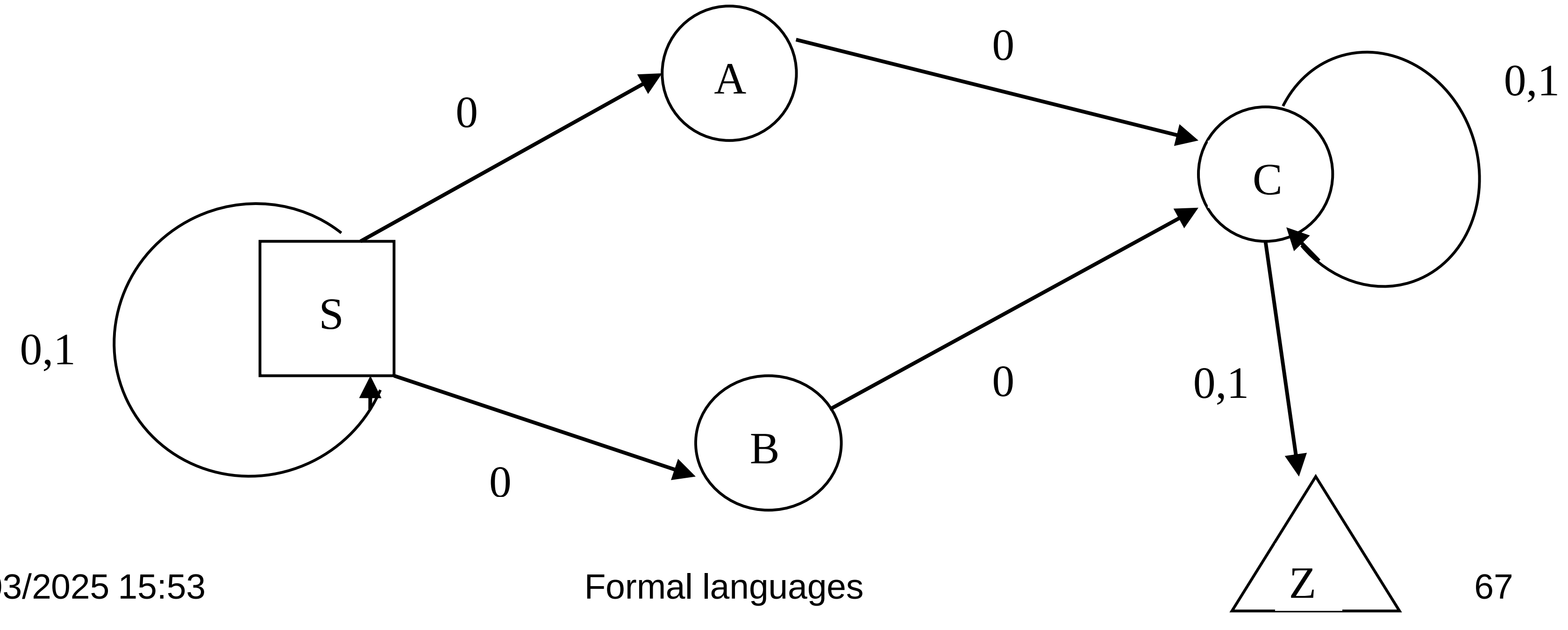


Example: From this graph, we can define the automata **:** (Vt, *Q*, *q0*, *f*, *F)*

Vt={0,1} Q={ S,A,B,C,Z } q0= S F= {Z}

f(S,0)=S f(S,1)=S f(S,0) = A f(S,1)= B f(A,0)=C f(B,1)=C f(C,0)=C

f(C,1)=C f(C,0)=Z f(C,1)=Z



c) <u>Table of transitions (matrix):</u>

Let A be the FSA defined by the quintuplet (Vt, *Q, q0, f, F)*) such that:

Vt =$\{a, b\}$, Q= $\{q_0, q_1\}$, q_0 : initial state F= $\{q_1\}$

and we define the following transitions: $f(q_0, a) = q_0$, $f(q_0, b) = q_1$, $f(q_1, b) = q_1$

The transition function can be represented by this matrix:

Q \ V_t	a	b
q_0	q_0	q_1
q_1		q_1

Language recognized by a finite state automata:

The language recognized by a finite state automata is the set of strings whose symbols lead from the initial state to one of its final states by a succession of transitions using all its symbols in order.

Definition of a configuration

The configuration of the FSA A, at a certain time, is given by the current state of the FSA and of the word which remains to be read:

(current state, word which remains to be read).

The initial configuration is (q_0, ω) where q_0 is the initial state of the FSA and ω the word submitted to A (to be recognized).

The final configuration is given by (q_f, ε) where q_f is a final state and the empty word indicates that there is nothing left to read.

➔ (the word belongs to the language).

Definition of a direct derivation:

We say that a configuration $(q_1, a\omega)$ directly derives the configuration (q_2, ω)

if and only if $f(q_1, a) = q_2$ where f is the transition function, $a \in Vt$ and $\omega \in Vt^*$.

We denote **$(q_1, a\omega) \models (q_2, \omega)$.**

Definition of an indirect derivation:

We say that a configuration (q, ω_1) indirectly derives another configuration (p, ω_2),

if and only if there exist 0, 1 or several direct derivations which, from (q, ω_1), lead

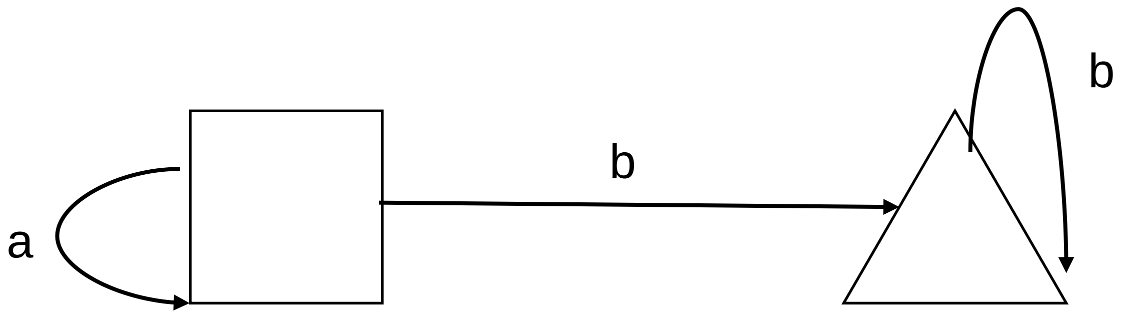
to the configuration (p, ω_2).

We denote **$(p, \omega_1) \models^* (q, \omega_2)$.**

Definition of the language recognized by a FSA :

The language recognized by the FSA A denoted L(A) is defined by:

$$L(A) = \{ \ \omega \in Vt^* \ / \ (q_0, \omega) \mid=^* \ (q_f, \varepsilon) \ \} \quad \text{with} \quad q_f \in F$$



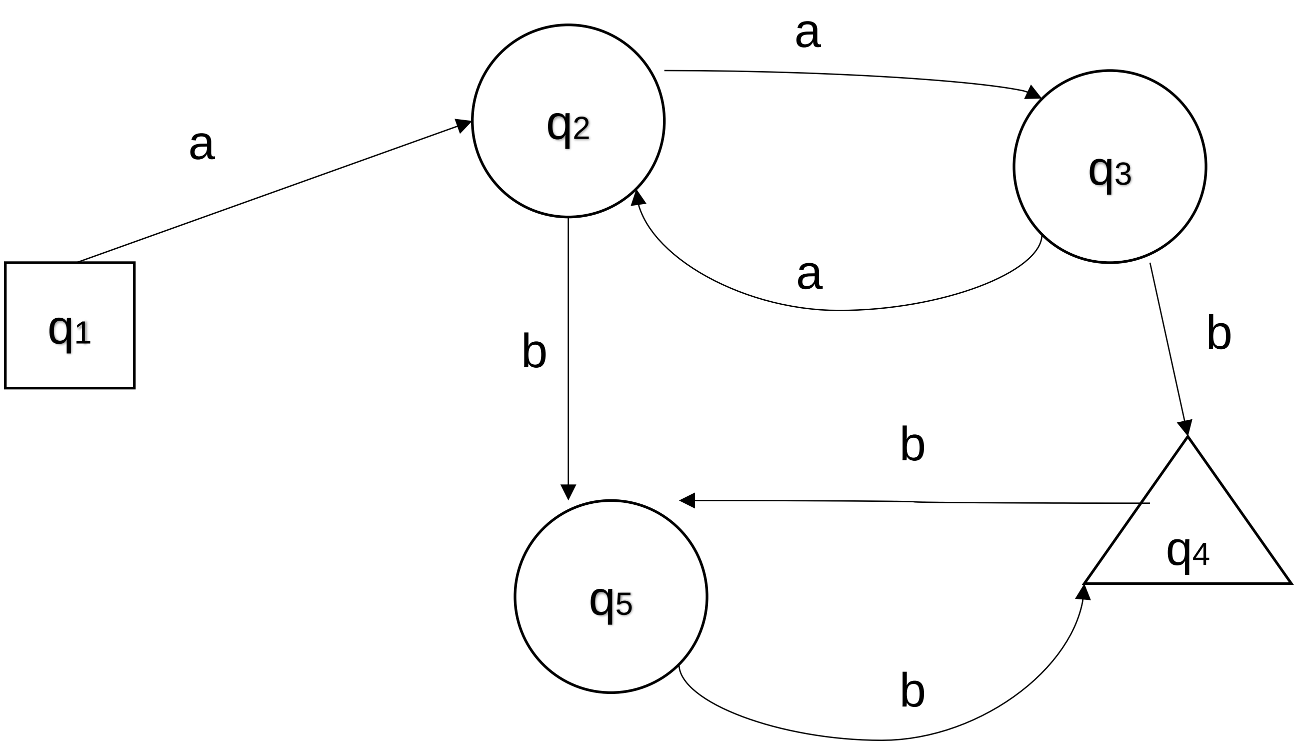
Simple example :

The minimal string is : b

The language recognized by this FSA is:

$L(A) = \{ a^n \ b \ b^m \ / \ n \geq 0 \ ; \ m \geq 0 \}$

Example 2: Find the language recognized by this Automata



Note: the state 4 is a final state and at the same time is an internal state

The possible paths:

- $q_1 \; q_2 \; q_5 \; q_4$
- $q_1 \; q_2 \; q_3 \; q_4$
- $q_1 \; q_2 \; q_3 \; q_2 \; q_5 \; q_4$
- $q_1 \; q_2 \; q_3 \; q_4 \; q_5 \; q_4$

The minimum strings:

- aab et abb

The recognized strings:

- $a\,(aa)^*\,b\;\;b\,(bb)^*$ - $a\,a\,(aa)^*\,b\,(bb)^*$

$L(A) = \{ a\,(aa)^*\,b\;\;b\,(bb)^* \} \cup \{ a\,a\,(aa)^*\,b\,(bb)^* \}$

Chapter 4: Regular languages

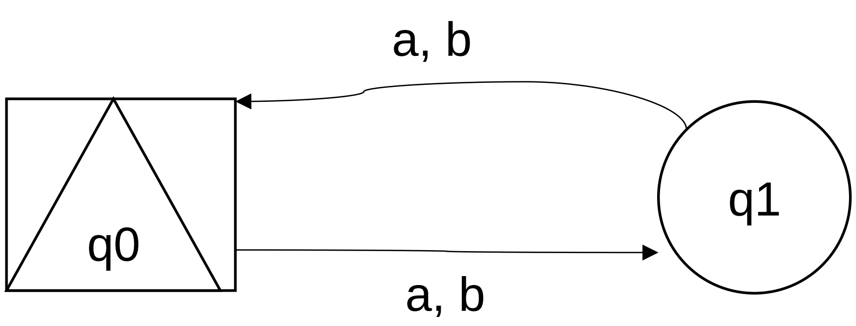
Remarks:

1- A finite state automata recognizes a single language, but the same language can be recognized by several automata,

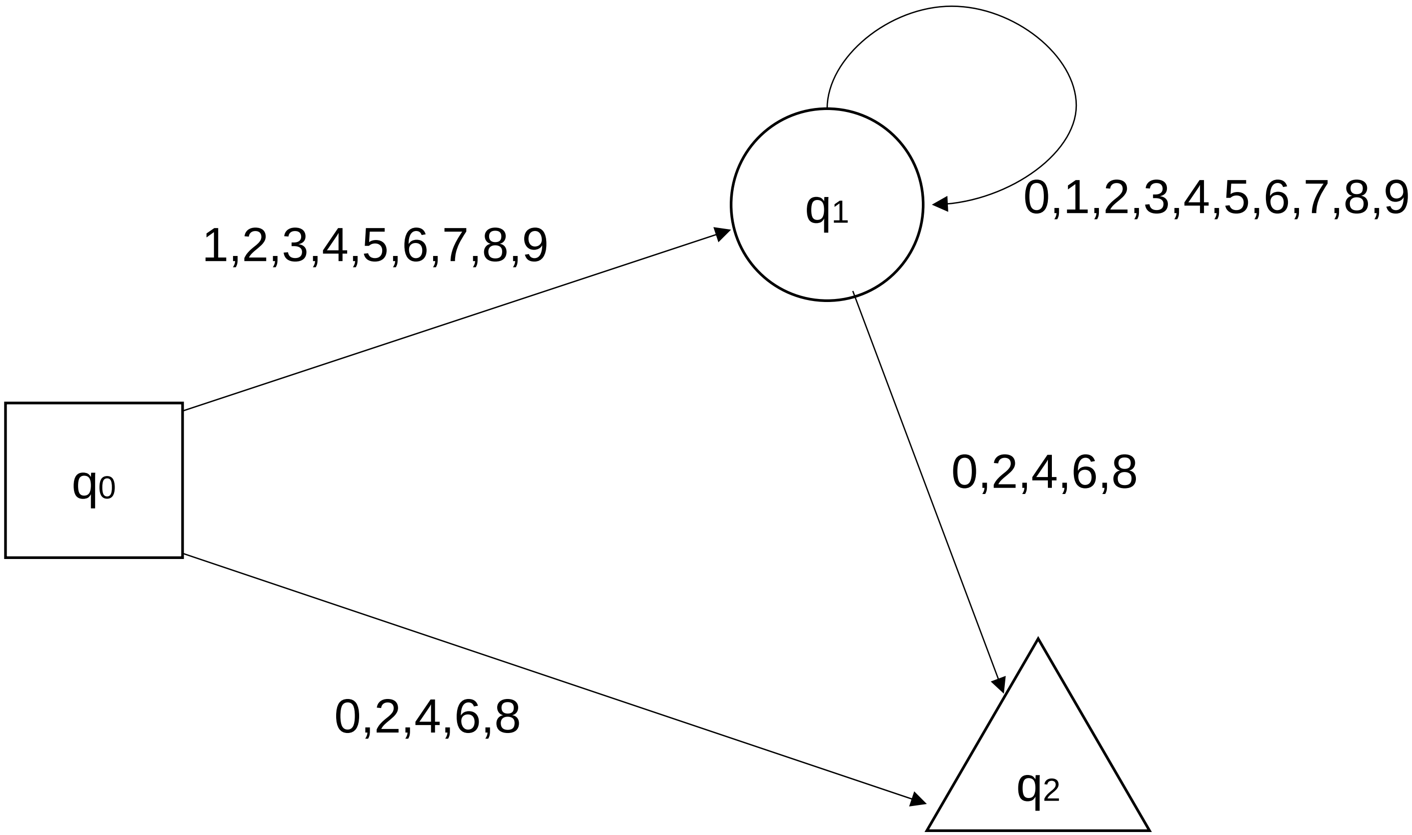
2- We say that two finite state automata A_1 and A_2 are equivalent if and only if these two automata recognize the same language. $L(A_1)= L(A_2)$

Example of automata construction:

L1= { w/ w ∈ {a,b}* et |w| ≡ 0 [2] }

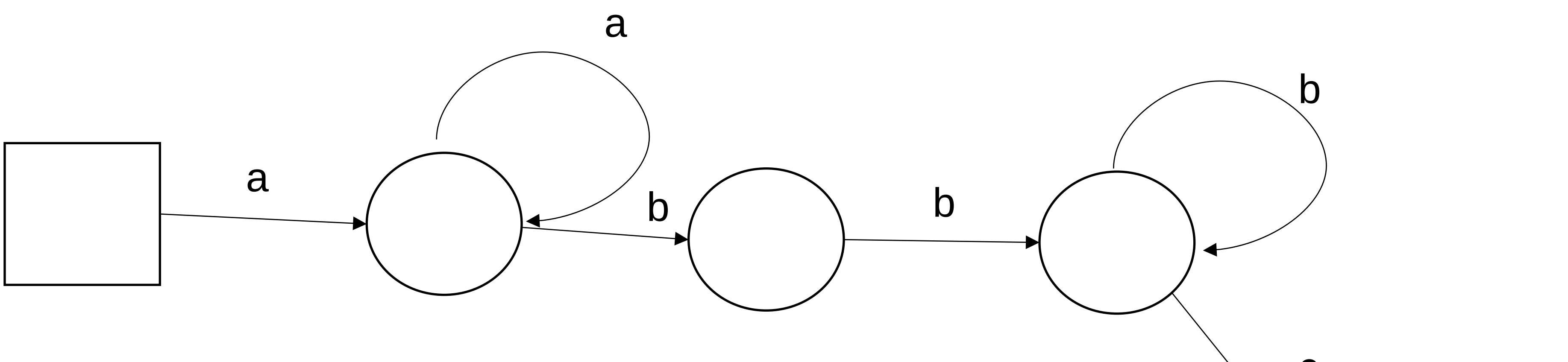


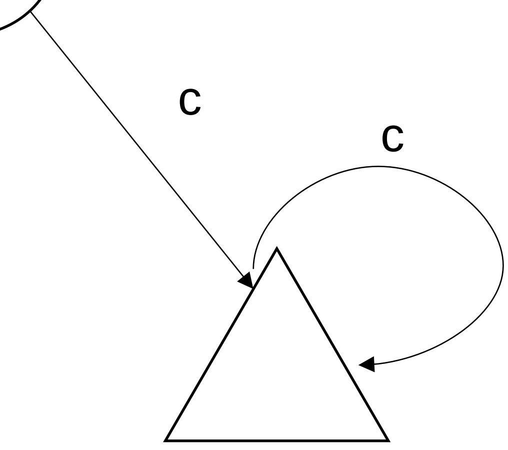
L2= { w/ w $\in$ *N* *and* *w* $\equiv$ *0* *[2]* }



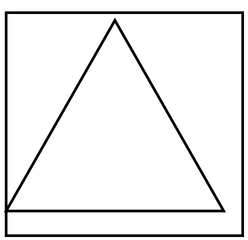
Chapter 4: Regular languages

L3= { $a^i b^j c^k$ / $i \geq 1$, $k \geq 1$ and $j > 1$ }

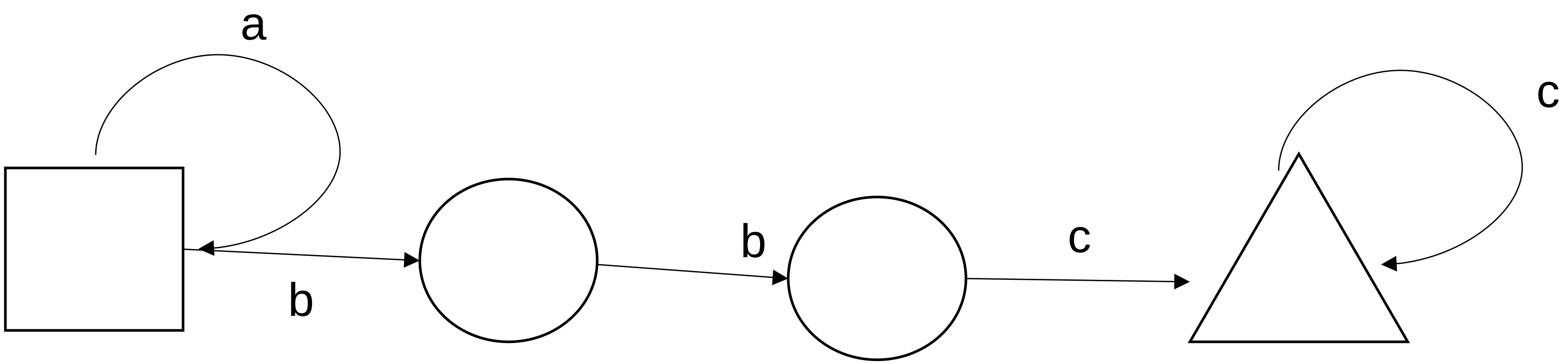




L4= { ε }



$L5 = \{ a^i b^2 c^k \ / \ i \geq 0 \text{ and } k > 0 \}$



4.4 Variants of finite state automata

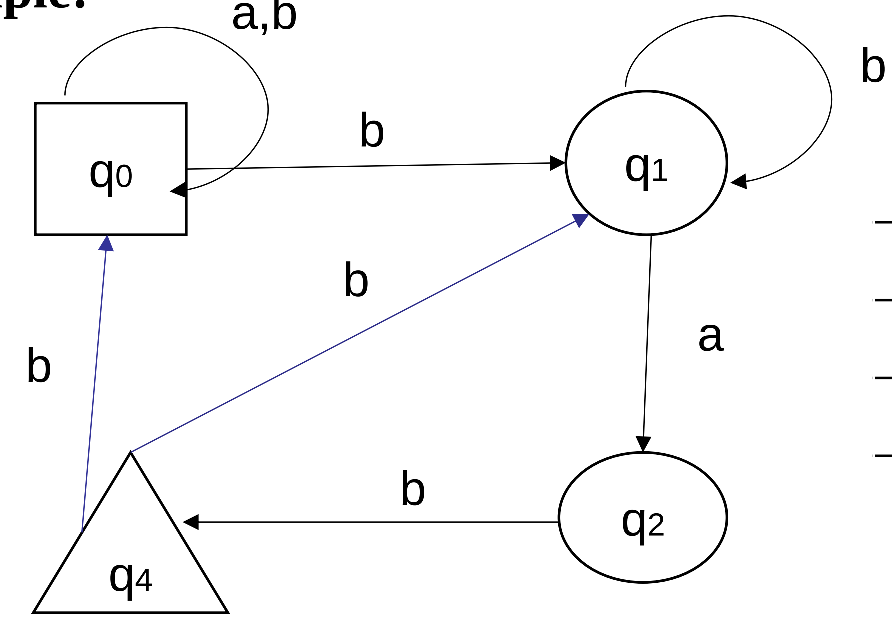
a) Deterministic Finite State Automata (DFSA)

A deterministic finite state automata is an automata such that its transition function is a function. For any state and for any symbol, there exists at most one state in which the automata can pass.

b) Nondeterministic finite state automata (NDFSA)

A nondeterministic finite state automata is an automata such that there exists at least one pair formed by a state and a symbol, which admits more than one image by the transition function. The automata must make choices to progress.

Example:



	a	b
q_0	q_0	q_0, q_1
q_1	q_2	q_1
q_2		q_4
q_4	q_0, q_1	

This automata is non-deterministic: for the same state and the same symbol we have two states, we find more than one value in a cell of the transition table

$f(q_0,b) = q0$ et $f(q_0,b) = q_1$

$f(q_4,b) = q_0$ et $f(q_4,b) = q_1$

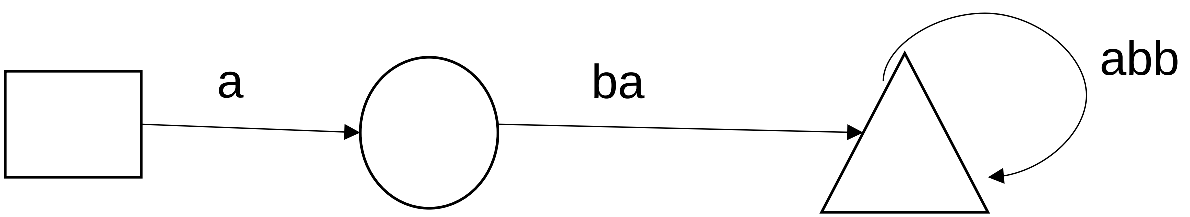
The problem of the NDFSA :

These automata analyze the strings more slowly than deterministic automata, we must make choices to progress and make feedback afterwards in the case of failure

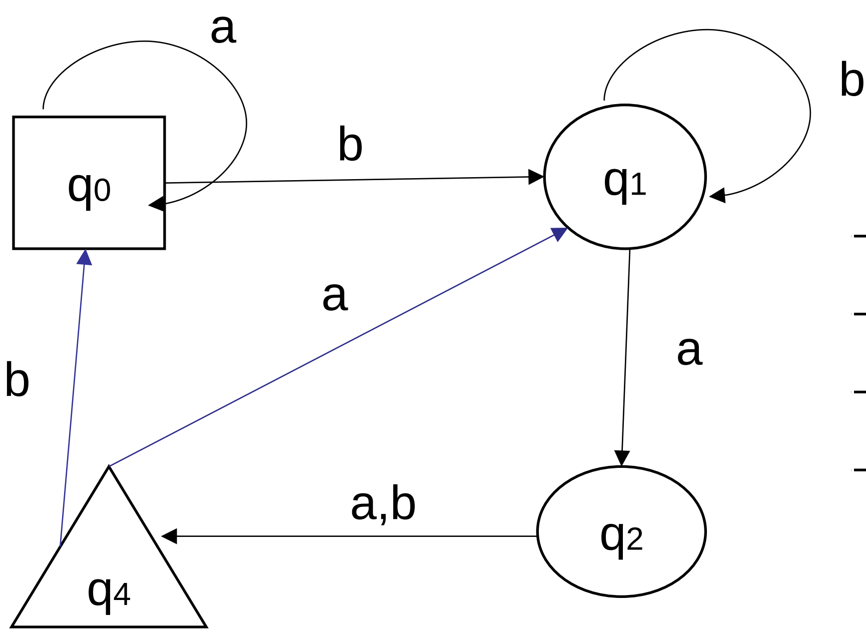
c) Generalized finite state automata GFSA)

- Transitions can be generated by words instead of symbols.

- The transitions caused by the empty word (ε) are called spontaneous or empty transitions (ε-transition). It is a change of state without reading.



Chapter 4: Regular languages



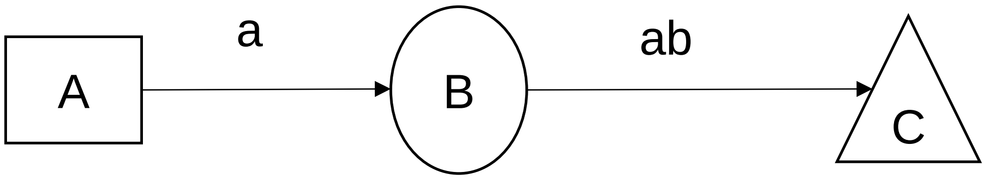
	a	b
q_0	q_0	q_1
q_1	q_2	q_1
q_2	q_4	q_4
q_4	q_1	q_0

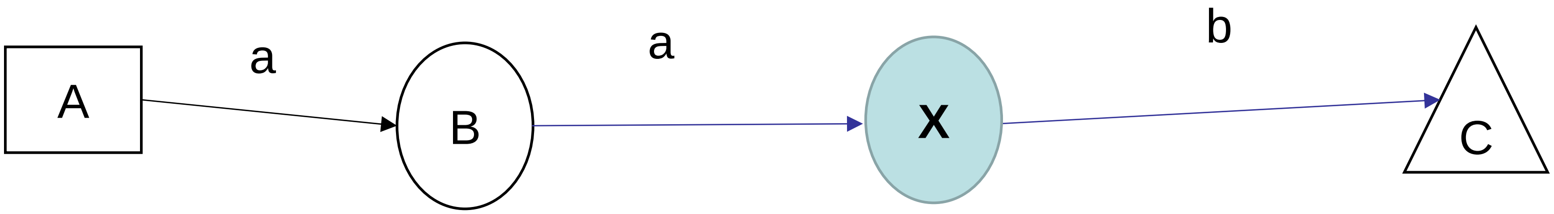
d) Complete state machine(CSM)

An automata is called complete if it is **deterministic** and for each state and for each symbol there is exactly one transition..

4.5 Transformation of a generalized automata into a simple automata

Any generalized finite state automata admits an equivalent simple automata by adding additional transitions.





4.6 Transformation from a non-deterministic FSA to a deterministic automata

To any non-deterministic finite state automata corresponds an equivalent deterministic finite state automata and vice versa.

The transition from a non-deterministic automata to a deterministic automata is done according to the following algorithm:

The goal is to find the elements of the deterministic automata.

$A=(V_t, Q, q_0, f, F)$ non-déterministic automata given

$A'=(V'_t, Q', q'_0, f', F')$ déterministic automata accepting the same language

1) $V'_t = V_t$

2) $q'_0 = \{q_0\}$

3) Q' is included in the set of combinations of Q, P(Q)

4) $F' = B \in Q'$ si $B \cap F \neq \theta$ ==> $B \in F'$

5) f' : $f'(B, x) = M$

let $B = \{ q_1, q_2, \ldots, q_i, \ldots, q_k \}$

$f(q_1, x) = M_1$ ….. $f(q_i, x) = M_i$, …., $f(q_k, x) = M_k$

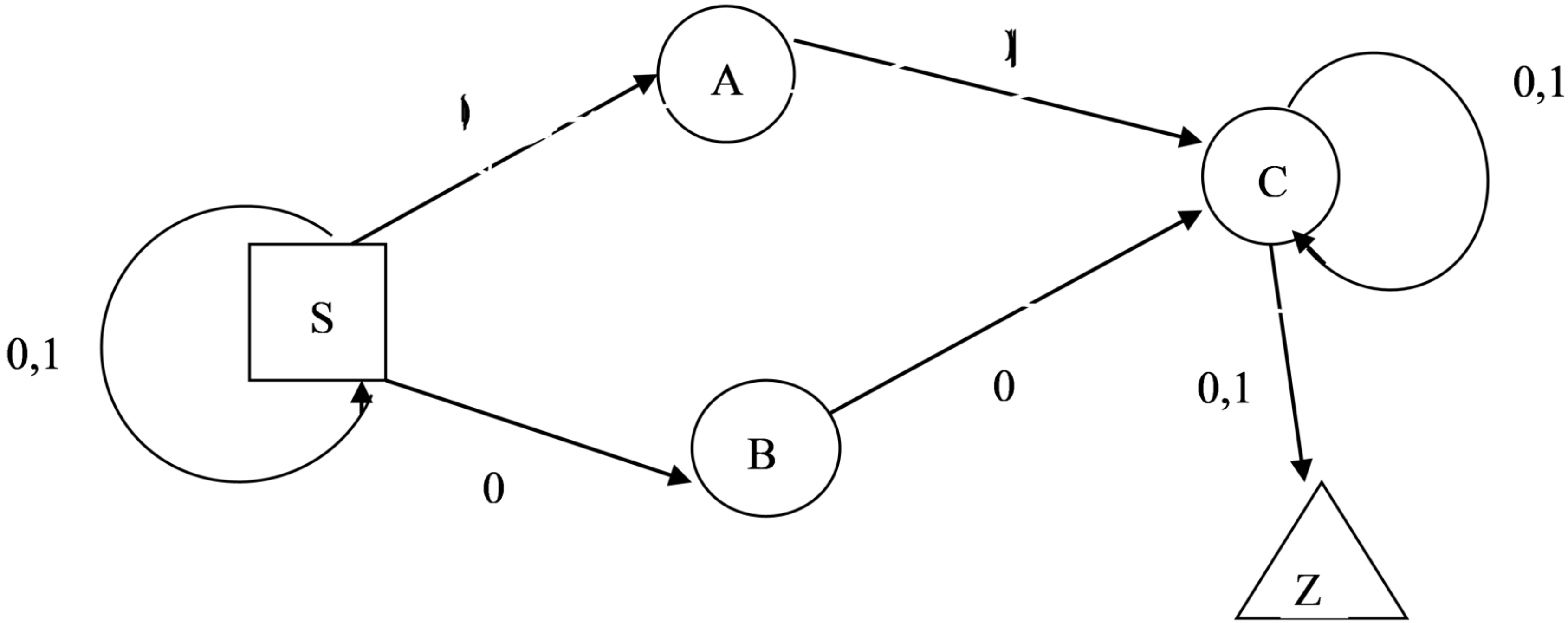
$M_i \in Q'$ M_i included in P(Q)

$M = \cup M_i \quad 1 \leq i \leq k$

6) Remove empty transitions.

Chapter 4: Regular languages

Example: find the deterministic automata



The construction of the new transitions table is done as follows :

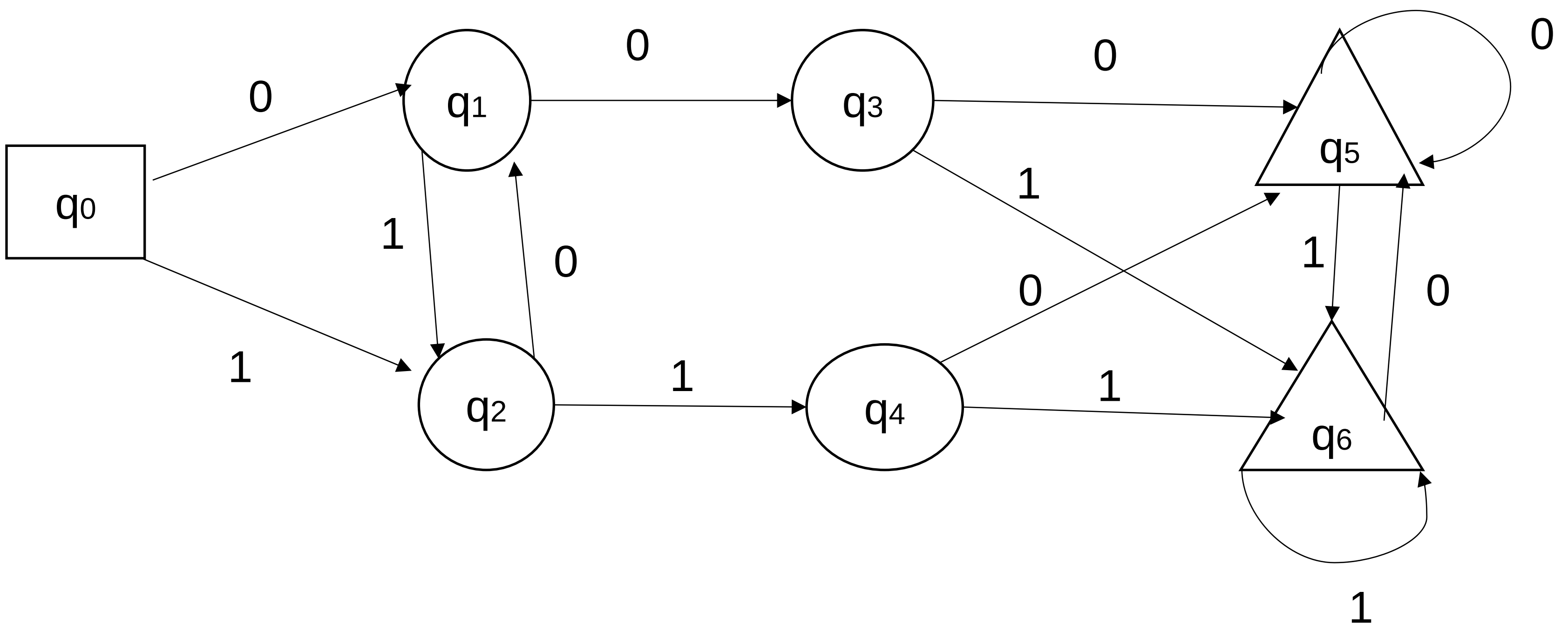
	0	1
{S } = q0	{S,B} q1	{S,A} q2
{S,B} = q1	{S,B,C} q3	{S,A} q2
{S,A} = q2	{S,B} q1	{S,A,C} q4
{S,B,C} = q3	{S,B,C,Z} q5	{S,A,C,Z} q6
{S,A,C}= q4	{S,B,C,Z} q5	{S,A,C,Z} q6
{S,B,C,Z} = q5	{S,B,C,Z} q5	{S,A,C,Z} q6
{S,A,C,Z} = q6	{S,B,C,Z} q5	{S,A,C,Z} q6

New transition table of the deterministic automata

Initial state : q_0

F= { q_5, q_6 } $Z \in q_5$ and $Z \in q_6$

The FSA deterministic equivalent



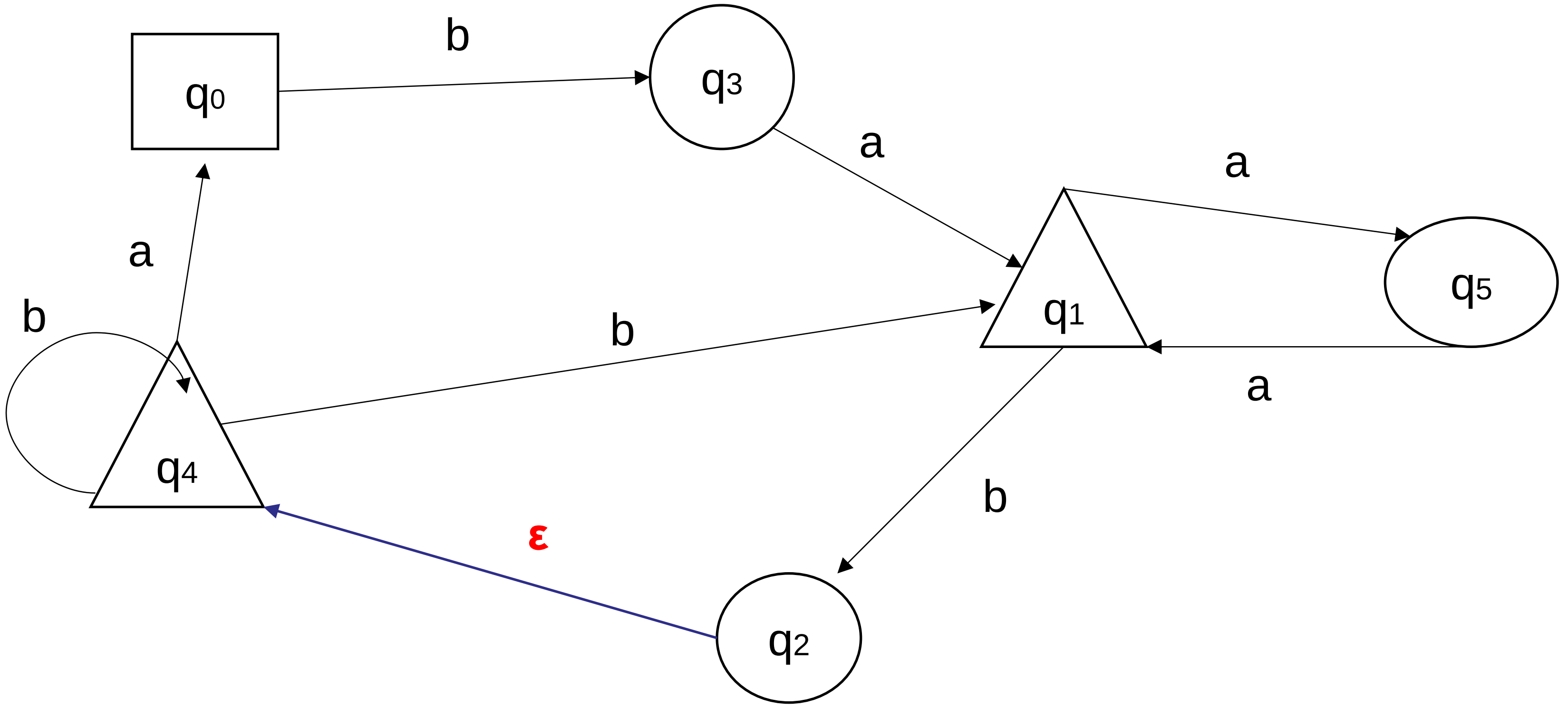
4.6 Elimination of empty transitions

Removing these transitions gives us a simple FSA , to do this we must first remove the transitions by ε:

$$\delta(q_i, \varepsilon) = q_j \quad \begin{cases} if \;\; q_j \in F \quad then \quad q_i \in F \\ and \\ \forall a \in Vt \; : if \;\; \delta(q_j, a) = q_k \quad then \quad \delta(q_i, a) = q_k \end{cases}$$

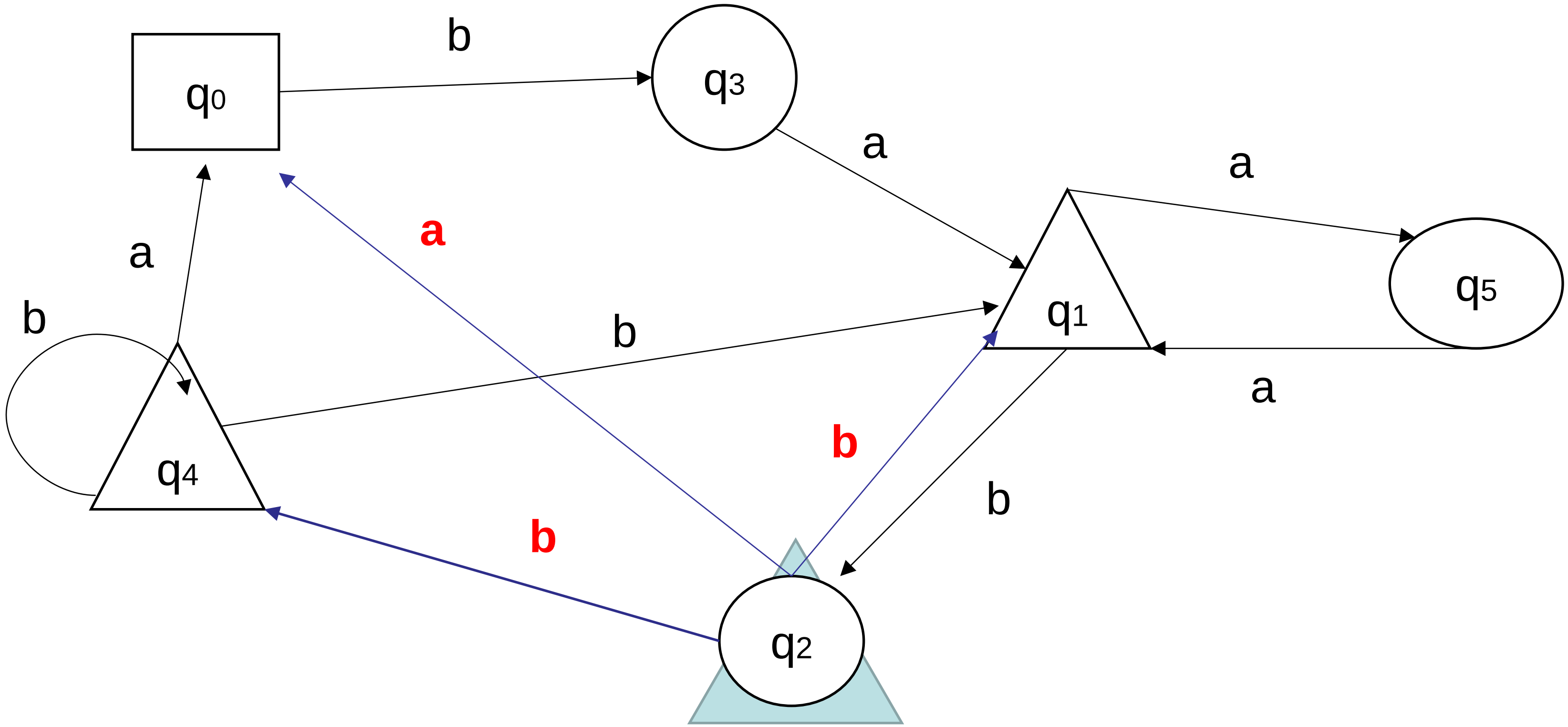
An example of this transformation is shown on the following FSA:

We have a single ε transition f(q2, ε) = q4 , q4 final state so q2 becomes final state and we have 3 transitions f(q4,a) = q0 , f(q4,b)= q1 and f(q4,b) = q4, so we add 3 transitions replacing q4 by q2



Chapter 4: Regular languages

We remove the empty transition and replace it with these transitions: f(q2,a) = q0 , f(q2,b)= q1 and f(q2,b) = q4



4.7 Transition from regular grammar to FSA

For any regular grammar G = (Vt, Vn, S, R), there exists a FSA

A = (Vt, Q, q_0,f, F) such that L(G)=L(A).

The passage is done by associating a transition to each rule of the grammar.

The built automata is not automatically deterministic.

Let G = (Vt, Vn, S, R), a regular grammar, the question is how to find a FSA

A = (Vt', Q, q_0,f, F) such that L(G)=L(A)

4.7 Transition from regular grammar to FSA

A = (Vt', Q, q_0,f, F) such that   L(G)=L(A)

1) $Vt' = Vt$

2) $Q = Vn \cup q_f$ such that $q_f \in F$

3) $q_0 = S$

4) $F = \{q_f\}$.

5) For each rule: $A \rightarrow a\ B$,   we associate the transition   f(A, a) = B.

For each rule of the form $A \rightarrow a$,   we associate the transition f(A, a) = q_f.

6) If the grammar has the rule $S \rightarrow \varepsilon$ then S becomes a final state F= { q_f, S}

4.7 Example

Let's find the FSA equivalent to the following grammar:

G=({a,b,c}, {S,A,B}, S, R)

R = ($S \rightarrow aS \mid bA$

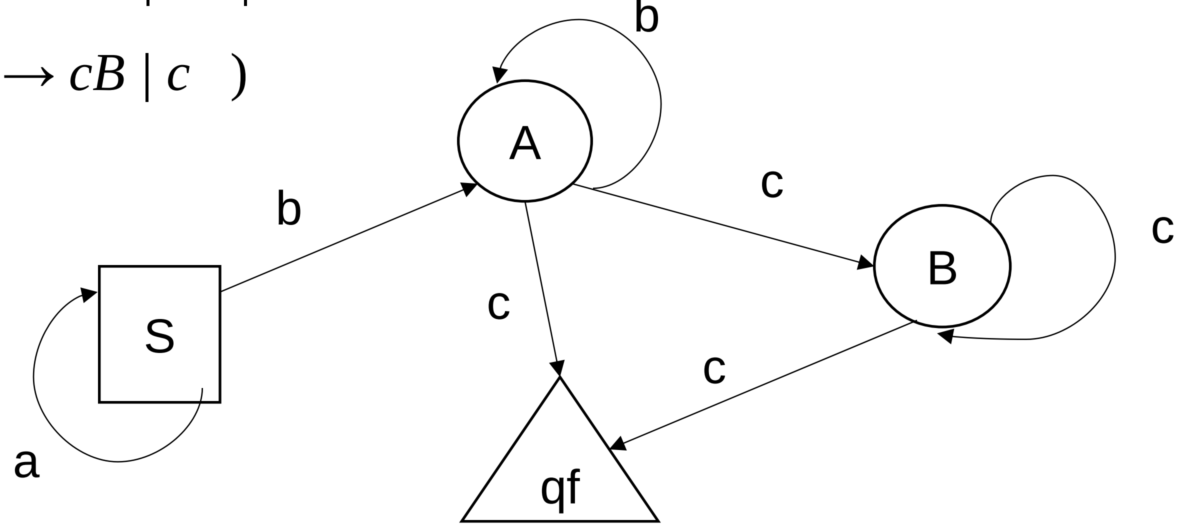
$A \rightarrow bA \mid cB \mid c$

$B \rightarrow cB \mid c$)

Q= { S,A,B, qf}

F={qf}

initial state : S



4.8 Transition from FSA to regular grammar:

For any FSA A = (Vt, Q, q_0,f, F) there exists an equivalent regular grammar

G = (Vt', Vn, S, R) such that L(G)=L(A).

The transition is done as follows:

1) Vt' = Vt 2) Vn = Q 3) $q_0 = S$

4)

if	$f(q, a) = p \in A$	then	$q \rightarrow a\, p$	$\in R$
If	$f(q, a) = q_f \in A$	then	$q \rightarrow a$	$\in R$
if	$q_0 \in F$	then	$q_0 \rightarrow \varepsilon$	$\in R$

<u>4.8 Example :</u> Find the grammar equivalent to this automata:

G=({a,b},{S,A,B, C,D}, S, R)

R = f(S,a) = A become **S → a A**

f(A,a) = B become **A → a B**

f(A,b) = C become **A → b C**

f(S,a) = A become **S → a A**

f(B,a) = A become **B → a A**

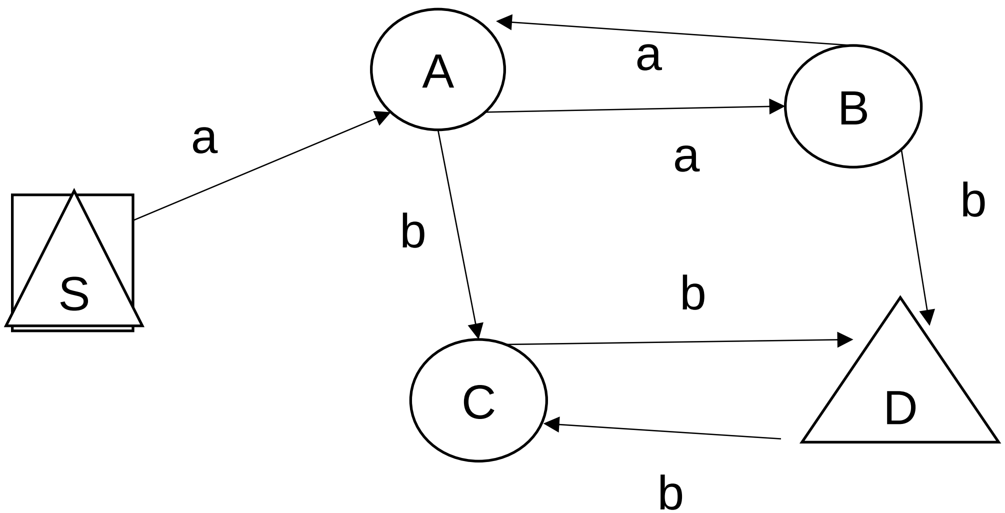
f(B,b) = D (D **internal**) become **B → b D**

f(B,b) = D (D final) become **B → b**

f(D,b) = C become **D → b C**

f(C,b) = D (D **internal** become **C → b D**

f(C,b) = D (D final) become **C → b**



The State **S** is an initial and final state at the same time, so we add the rule:

S → ε

4.9 Regular expressions

a) Definition :

Regular expressions (RE) provide another method of defining regular languages. They are more practical than the other two systems (regular grammars and automata).

Each regular expression describes a set of terminal strings.

The regular expression formalism uses 03 operations:

1. Concatenation
2. Closing noted * (power)
3. The alternative noted + or / (choice between two expressions)

4.9 Regular expressions

b) Formal definition:

- ϕ is a regular expression which denotes (represents) the empty language
- ε is a regular expression which denotes the language {ε}
- a (where a∈ Vt) is a regular expression which denotes the language {a}.

Induction:

ε and **a** are regular expressions;

If e, e' are regular expressions then e+e' , e.e', e* are regular expressions.

Remarks :

- The exponent has a higher priority than the concatenation which has a higher priority than the sum.

- Two regular expressions are ε-equivalent if and only they denote the same language.

4.9 Regular expressions

c) **Examples :**

(a + b)* = { ε, a, b, aa, bb, ab, ba, aaa, bbb, aab, aba, …….. } infinite language

a*b + b*a = { b, ab, aab, aaab, aa….ab, a, ba, bba, bbba, bb…ba, …. }

ab*(c+ a) = ab*c + ab*a = { ac, abc, abbc, abbbc, ….., aa, aba, abba, abbba, ….}

Chapter 4: Regular languages

Regular Expressions	Regular Set
(0 + 10*)	L = { 0, 1, 10, 100, 1000, 10000, … }
(0*10*)	L = {1, 01, 10, 010, 0010, …}
(0 + ε)(1 + ε)	L = {ε, 0, 1, 01}
(a+b)*	Set of strings of a's and b's of any length including the null string. So L = { ε, a, b, aa , ab , bb , ba, aaa…….}
(a+b)*abb	Set of strings of a's and b's ending with the string abb. So L = {abb, aabb, babb, aaabb, ababb, …………..}
(11)*	Set consisting of even number of 1's including empty string, So L= {ε, 11, 1111, 111111, ……….}
(aa)*(bb)*b	Set of strings consisting of even number of a's followed by odd number of b's , so L = {b, aab, aabbb, aabbbbb, aaaab, aaaabbb, …………..}
(aa + ab + ba + bb)*	String of a's and b's of even length can be obtained by concatenating any combination of the strings aa, ab, ba and bb including null, so L = {aa, ab, ba, bb, aaab, aaba, …………..}

4.10 Transition from regular expression to regular grammar

0 (0 + 1)* 0

G=({0 ,1},{S,A}, S, R)

R = (S → 0 A   A → 0 A   A → 1 A   A → 0   )

(0 + 1)* 0 (0 + 1) (0 + 1)

G=({a,b},{S,A, B }, S, R)

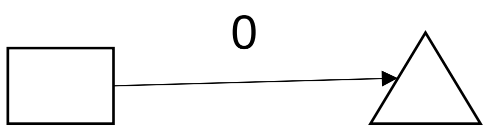
R = (S → 0S / 1S   S → 0 A   A → 0 B / 1B   B → 0 / 1   )

4.11 Transition from a regular expression to a FSA

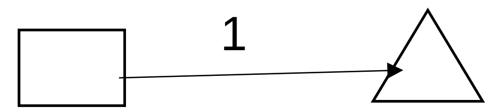
There are automatas for each regular expression

Construction example: **(0 + 1)* 0 1 0**

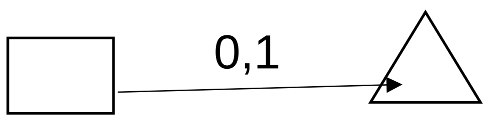
0 :



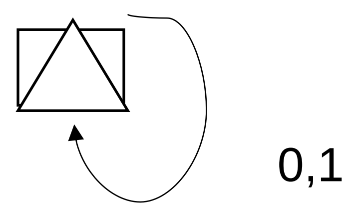
1 :



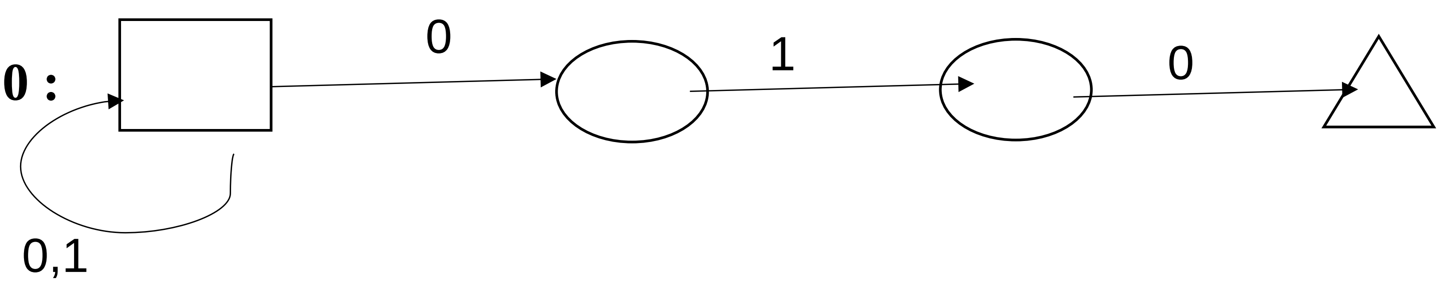
(0 + 1) :



(0 + 1)* :



(0 + 1)* 0 1 0 :



4.11 Methods to show that a language is regular

We can show the regularity of a language L, by one of the following methods:

- All finite languages are regular
- If we find a FSA which recognizes a language L, then L is regular
- If we find a regular grammar generating L, then this language is regular
- We can exploit closure properties to show that a language is regular. (properties of regular expressions)

4.13 Properties of regular languages

- The union of two regular languages is a regular language.
- The concatenation of two regular languages is a regular language
- The iteration of a regular language is a regular language

4.14 Arden's Theorem

In order to find out a regular expression of a FSA , we use Arden's Theorem along with the properties of regular expressions.

Statement

Let P and Q be two regular expressions.

If P does not contain null string, then $R = Q + RP$ has a unique solution that is $R = QP^*$

Proof

$R = Q + (Q + RP)P$ [After putting the value $R = Q + RP$]

$= Q + QP + RPP$

4.14 Arden's Theorem

Proof

$R = Q + (Q + RP)P$ [After putting the value $R = Q + RP$]

$= Q + QP + RPP$

When we put the value of R recursively again and again, we get the following equation:

$R = Q + QP + QP^2 + QP^3 \ldots\ldots$ $R = Q (\varepsilon + P + P^2 + P^3 + \ldots\ldots)$

$R = QP^*$ [As P^* represents $(\varepsilon + P + P^2 + P^3 + \ldots\ldots)$]

Assumptions for Applying Arden's Theorem

- The transition diagram must not have NULL transitions

- It must have only one initial state

4.14 Arden's Theorem

Method

Step 1 − Create equations as the following form for all the states of the FSA having n states with initial state q_1.

$q_1 = q_1R_{11} + q_2R_{21} + \dots + q_nR_{n1} + \boldsymbol{\varepsilon}$

$q_2 = q_1R_{12} + q_2R_{22} + \dots + q_nR_{n2}$

……………………………

……………………………

……………………………

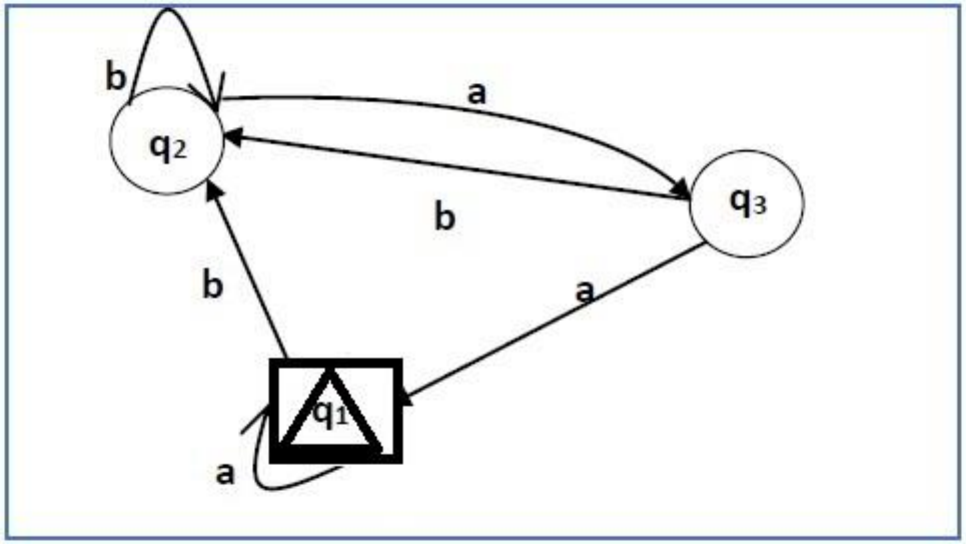
$q_n = q_1R_{1n} + q_2R_{2n} + \dots + q_nR_{nn}$

R_{ij} represents the set of labels of edges from q_i to q_j, if no such edge exists, then $R_{ij} = \emptyset$

Step 2 − Solve these equations to get the equation for **the final state** in terms of R_{ij}

4.14 Arden's Theorem

Problem: Construct a regular expression corresponding to the automata given below:



Solution:

Here the initial state and final state is $\mathbf{q_1}$.

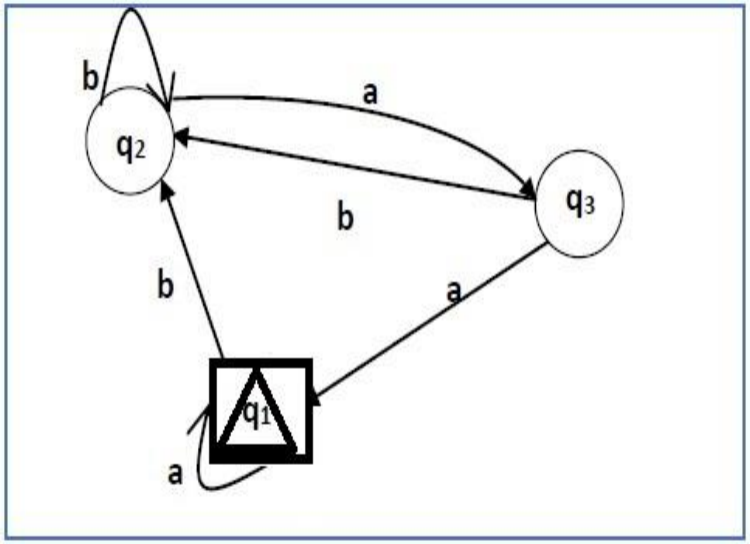
The equations for the three states q1, q2, and q3 are as follows:

$q_1 = q_1a + q_3a + \varepsilon$ (ε move is because q1 is the initial state)

$q_2 = q_1b + q_2b + q_3b$

$q_3 = q_2a$

4.14 Arden's Theorem



Now, we will solve these three equations:

$q_2 = q_1b + q_2b + q_3b$

$= q_1b + q_2b + (q_2a)b$ (Substituting value of q_3)

$= q_1b + q_2(b + ab)$

$= q_1b (b + ab)^*$ (Applying Arden's Theorem)

$q_1 = q_1a + q_3a + \varepsilon$

$= q_1a + q_2aa + \varepsilon$ (Substituting value of q_3)

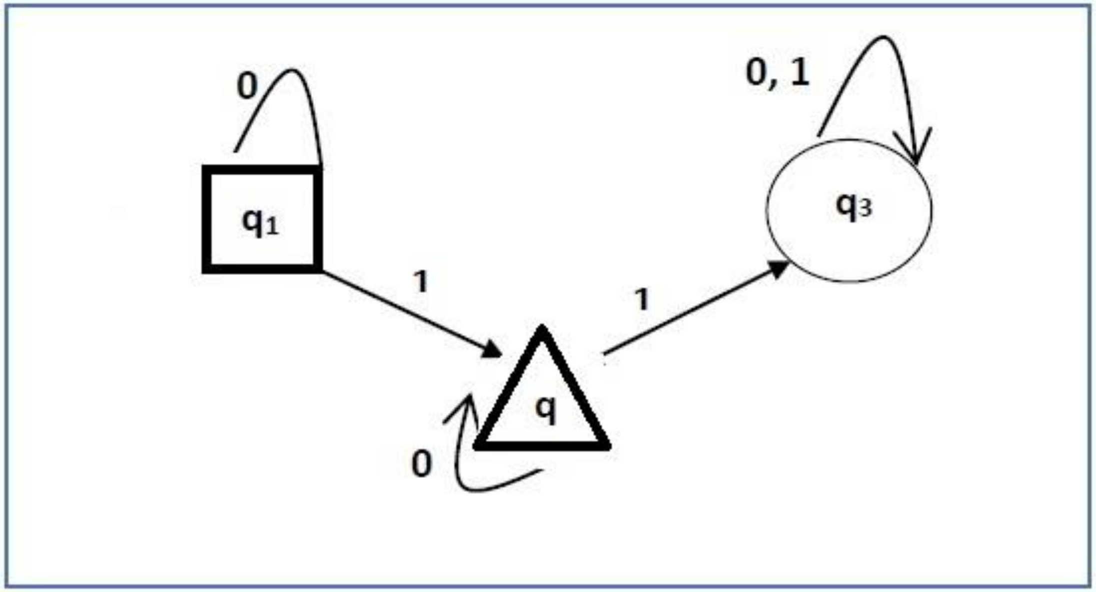
$= q_1a + q_1b(b + ab^*)aa + \varepsilon$ (Substituting value of q_2)

$= q_1(a + b(b + ab)^*aa) + \varepsilon$

$= \varepsilon (a+ b(b + ab)^*aa)^*$

$= (a + b(b + ab)^*aa)^*$

Hence, the regular expression is $(a + b(b + ab)^*aa)^*$.



4.14 Arden's Theorem

Problem

Construct a regular expression corresponding to this automata:

Solution:

Here the initial state is q_1 and the final state is q_2

Now we write down the equations:

$q_1 = q_1 0 + \varepsilon$

$q_2 = q_1 1 + q_2 0$

$q_3 = q_2 1 + q_3 0 + q_3 1$

Now, we will solve these three equations;

$q_1 = \varepsilon 0^*$ [R = Q + RP] So, $q_1 = 0^*$

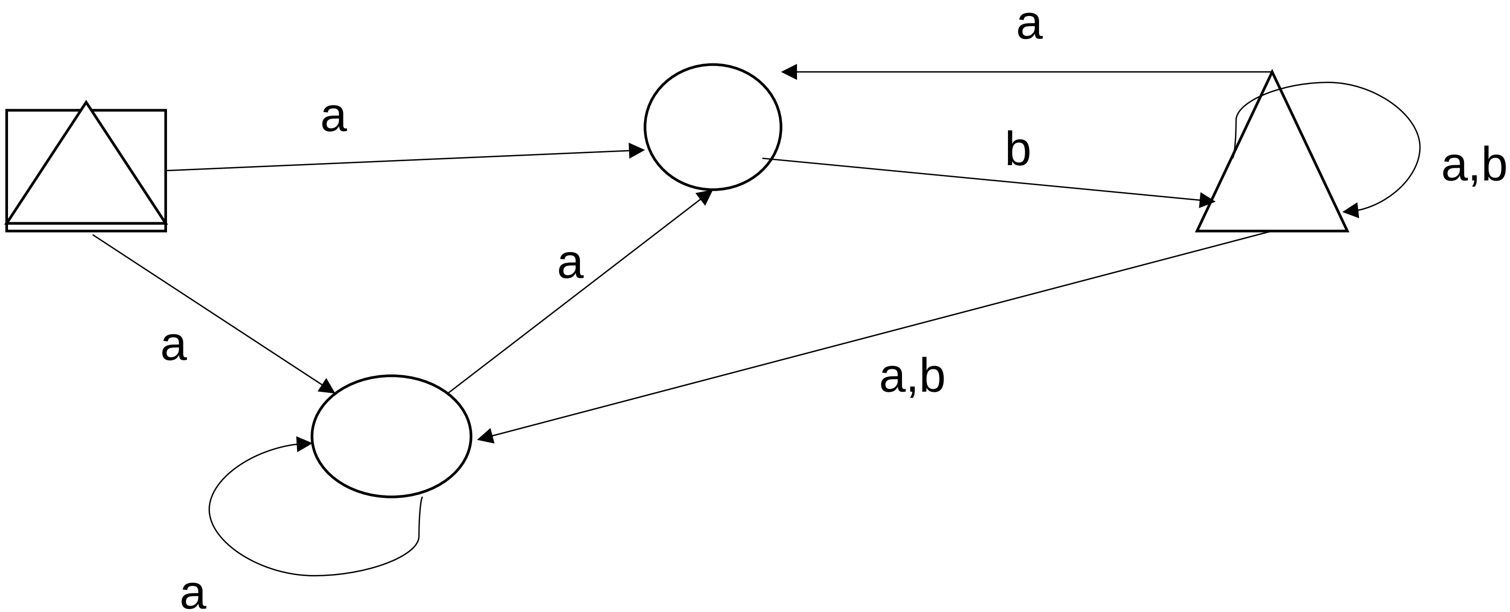
$q_2 = 0^*1 + q_2 0$ So, $q_2 = 0^*1(0)^*$ [By Arden’s theorem]

Hence, the regular expression is **0*10***.

4.14 important Example

- Build a FSA equivalent to this RE :

- **(a* a b (a + b)*)***



Remark : Avoid the loop in the initial state.

4.15 Minimization of DFSA

DFSA minimization stands for converting a given DFSA to its equivalent DFSA with minimum number of states.

There are many methods to minimize DFSA. The most used is equivalence method.

Step-01:

- Eliminate all the dead states and inaccessible states from the given DFSA (if any).

Step-02:

- Draw a state transition table for the given DFSA.
- Transition table shows the transition of all states on all input symbols

Step-03:

Now, start applying equivalence theorem.

- Take a counter variable k and initialize it with value 0. $k \leftarrow 0$
- Divide Q (set of states) into two sets such that one set contains all the non-final states and other set contains all the final states.
- This partition is called P_0. 0 equivalence

Step-04:

Increment k by 1.

- Find P_k by partitioning the different sets of P_{k-1} .
- In each set of P_{k-1} , consider all the possible pair of states within each set and if the two states are distinguishable, partition the set into different sets in P_k.

Two states q_1 and q_2 are distinguishable in partition P_k for any input symbol 'a', if f (q_1, a) and f (q_2, a) are in different sets in partition P_{k-1}.

Step-05:

- Repeat step-04 until no change in partition occurs.
- In other words, when you find $P_k = P_{k-1}$, stop.

Step-06:

- All those states which belong to the same set are equivalent.
- The equivalent states are merged to form a single state in the minimal DFSA.

4.15 Minimization of DFSA : Example

Minimize this automata:

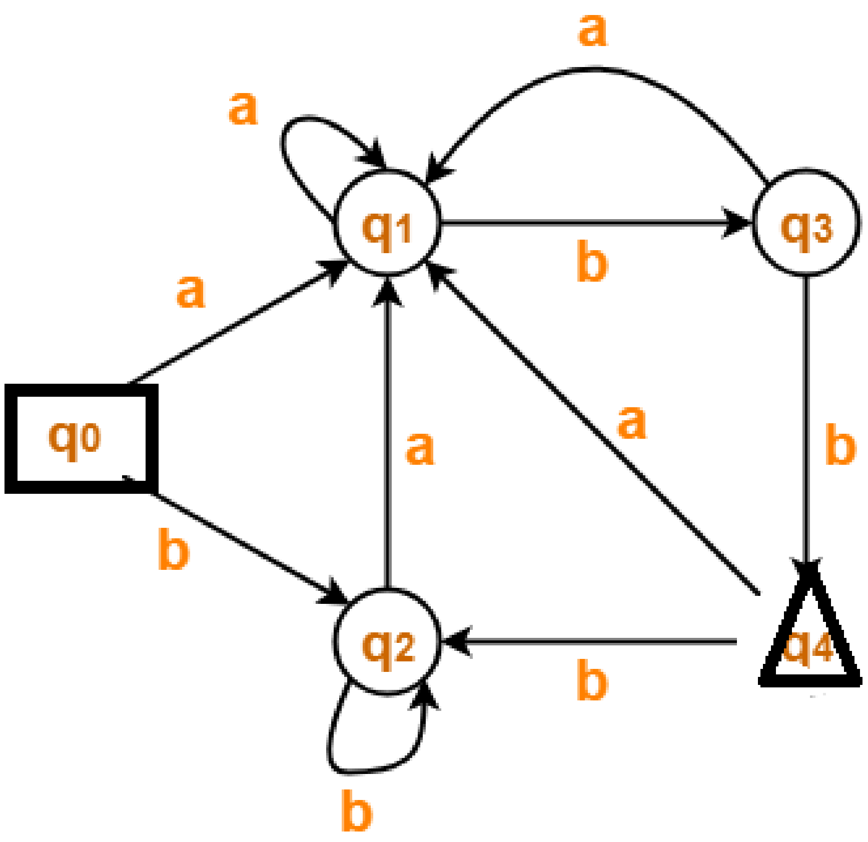
Step-01:

The given DFSA contains no dead states and inaccessible states.

Step-02:

Draw a state transition table-

	a	b
→ q0	q1	q2
q1	q1	q3
q2	q1	q2
q3	q1	q4
← q4	q1	q2



4.15 Minimization of DFSA : Example

Step-03:

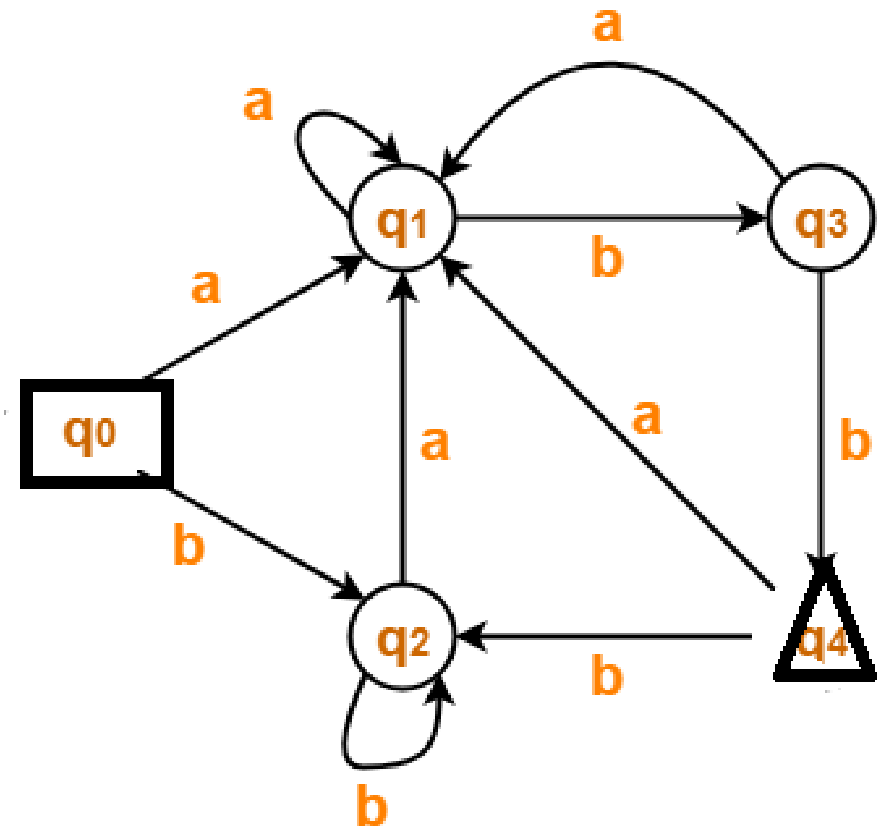
Now using Equivalence Theorem, we have:

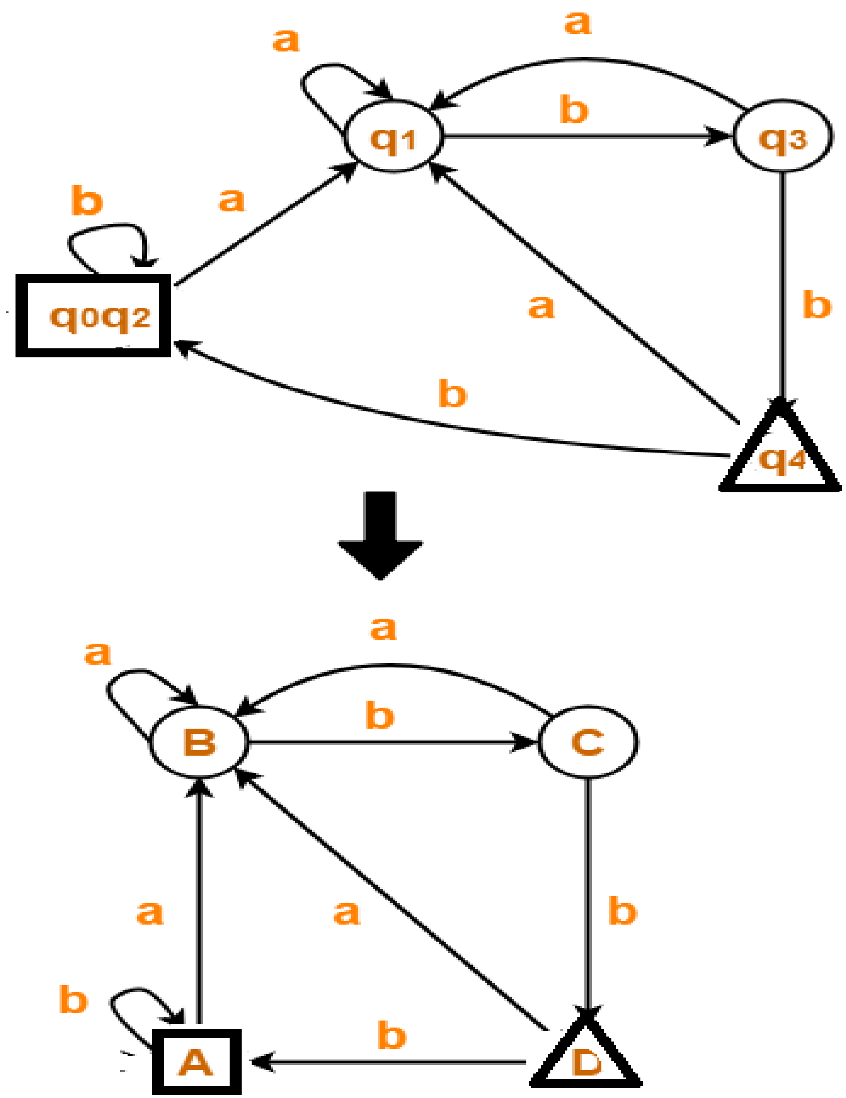
$P_0 = \{ q_0 , q_1 , q_2 , q_3 \} \{ q_4 \}$
$P_1 = \{ q_0 , q_1 , q_2 \} \{ q_3 \} \{ q_4 \}$
$P_2 = \{ q_0 , q_2 \} \{ q_1 \} \{ q_3 \} \{ q_4 \}$
$P_3 = \{ q_0 , q_2 \} \{ q_1 \} \{ q_3 \} \{ q_4 \}$

Since $P_3 = P_2$, so we stop.

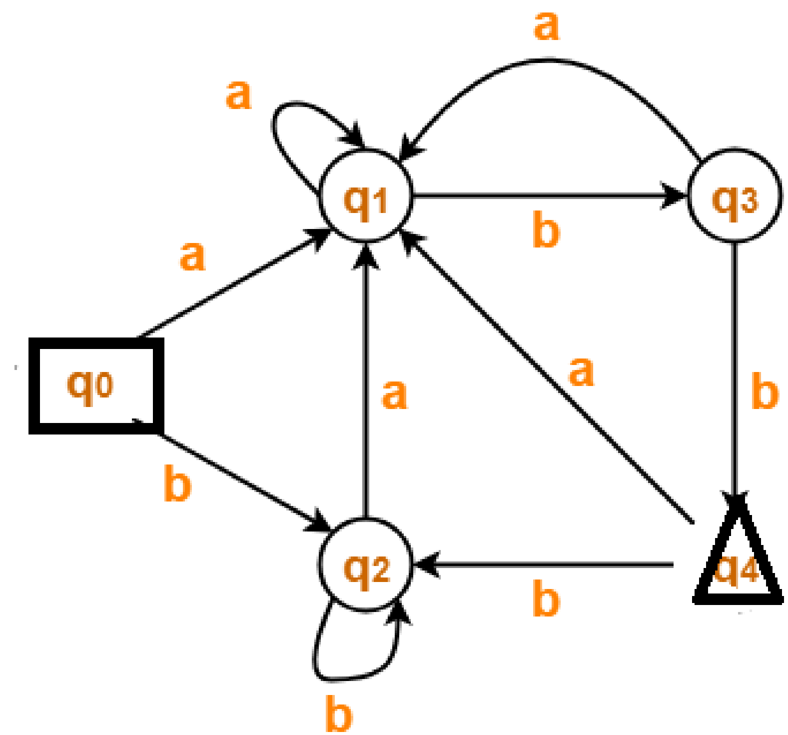
From P_3, we infer that states **q_0 and q_2** are equivalent and can be merged together.

So, Our minimal DFSA is:

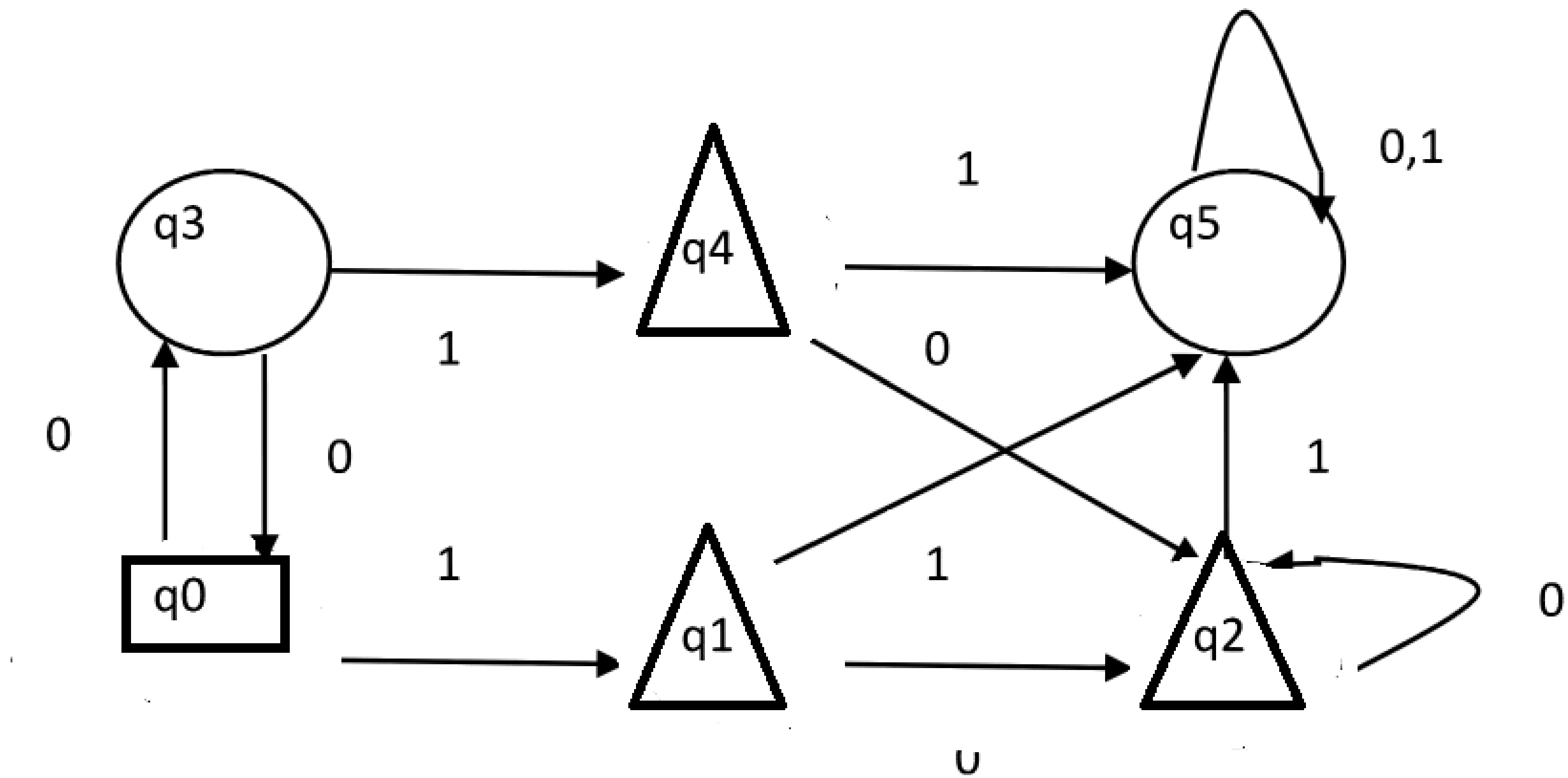




DFSA minimal



DFSA non minimal



DFSA non minimal