**Alphabets and Languages**

* [**Definitions**](http://faculty.simpson.edu/lydia.sinapova/www/cmsc365/LN365_Lewis/L05-Languages.htm#alphabet)
* [**Operations on strings**](http://faculty.simpson.edu/lydia.sinapova/www/cmsc365/LN365_Lewis/L05-Languages.htm#concat)
* [**Languages**](http://faculty.simpson.edu/lydia.sinapova/www/cmsc365/LN365_Lewis/L05-Languages.htm#language)
* [**Operations on languages**](http://faculty.simpson.edu/lydia.sinapova/www/cmsc365/LN365_Lewis/L05-Languages.htm#oplang)
* [**Problems**](http://faculty.simpson.edu/lydia.sinapova/www/cmsc365/LN365_Lewis/L05-Languages.htm#problems)

[**Learning goals**](http://faculty.simpson.edu/lydia.sinapova/www/cmsc365/LN365_Lewis/L05-Languages.htm#goals)[**Exam-like problems**](http://faculty.simpson.edu/lydia.sinapova/www/cmsc365/LN365_Lewis/L05-Languages.htm#exam)

1. **Some definiions and properties**

**Alphabet**: A finite set of symbols.

E.G {a,b,c,…x.y.z}. {0,1},{0,1,2,3,4,5,6,7,8,9}

**String**over an alphabet: A finite sequence of symbols from the alphabet

E.G.: thisisastring - over {a,b,c,…,z}

01011 - over {0,1}  
3786 - over {0,1,2,3,4,5,6,7,8,9}

A string with one symbol only = symbol itself

**Empty string** - no symbols, notation: *e*

Note: we use the letters a,b,c,…, w,x,y,z both for naming strings and for writing instances of strings.   
Usually for names of strings we use the last letters: w, x,y,z

Thus x = abc means that **abc** is a string and we call it **x**.

**Length of a string** - its length as a sequence (the number of symbols)

if ***w* = abcd, |*w***| = 4  
If ***w*** = classroom, |***w***| = 9

We can match a position in a string with the symbol there:

If ***w*** = classroom, ***w***(3) = a, ***w***(4) = s, and ***w***(5) = s

To be able to distinguish between same symbols, we refer to them as different occurrences of the same symbol.

1. **Operations on strings**

**Concatenation:** combines two strings by putting them one after the other.

E.G ***x*= abc**, ***y* = mnop**, then ***x ◦ y* = abcmnop**, or simply ***xy* = abcmnop**

The concatenation of the empty string with any other string gives the string itself:

***x e = ex = x***

**Substring:** If ***w*** is a string, then ***v*** is a substring of ***w*** if there exist strings ***x***and ***y*** such that ***w = xvy***

***x*** is called **prefix**, and ***y*** is called **suffix** of ***w***

The i-th concatenation of a string with itself is defined in the following way:

***w0 = e***

***w i+1 = w i ◦ w*** for each i ≥ 0.

So ***w1 = w***, bang 2 = bangbang

**Kleene star operation** on strings: Let **w** be a string. **w\*** is the set of strings obtained   
by applying any number of concatenations of **w** with itself, including the empty string.

**Example:** a\* = { e, a, aa, aaa, aaaa, aaaaa, …}

**Reversal**of a string ***w***denoted ***w R*** is the string spelled backwards

**Formal definition:**

If ***w*** is a string of length 0, then ***w R= w = e***

If ***w*** is a string of length n+1 > 0, then ***w = ua*** for some a **є**∑, and ***w R= a u R.***

1. **Languages**

If ∑ is an alphabet, then ∑\* is the set of all strings over ∑.

**Language:** any set of strings over an alphabet ∑, i.e. any subset of ∑\*.

∑\* is a countably infinite set. Its elements can be ordered in the following way:

* + 1. The alphabet ∑ is a finite set, so we can order the symbols in some way.
    2. The set ∑\* can be partitioned into disjoint sets with respect to the length of the strings (there are infinite number of strings, however each string has a finite length)
    3. For each **k** ≥ 0 first we enumerate all strings of length **k** before all strings of length **k+1**. This means that we first order the strings of length 0 (this is the empty string), then strings of length 1, then of length 2, etc.
    4. Strings of length k, denoted as nk are enumerated lexicographically :

**ai1ai2…aik** precedes **aj1aj2…ajk**   
if for some **m**, 0 ≤ m ≤ k-1, we have ih=jh for h = 1,…,m   
and im+1 < jm+1. Note that if ih=jh means that aih is the same as ajh .

1. **Operations on languages**

Languages are sets, so the operations union, intersection and difference are applicable. There are two operations specific for languages:

**Concatenation of languages**

Concatenation of languages is defined in the following way:

If L1 and L2 are languages, then L = L1**◦**L2 (or simply L1L2) is the set:

L = {w **є** ∑\* **:** w = x **◦**y , x **є** L1, y**є** L2}

i.e. L consists of all possible concatenations between strings in L1 and strings in L2.

Concatenation of languages corresponds to the Cartesian product of sets.

**Kleene star** of a language L: the set of all strings obtained by concatenating zero or more strings from L. It is denoted by L\*.

If we consider ∑ as a language, then ∑\* would be the Kleene star of that language.

1. **Problems**
   1. Is the set of all possible meaningful English sentences countable?
   2. Is the set of all possible meaningless English sentences countable?
   3. Define the relation **<** between words so that it describes the ordering of words in a dictionary.

**Solution**

The word ***wi*** precedes the word ***wj*** in the dictionary iff one of the following is true:

* + 1. There is a nonempty string ***x,*** such that ***wj = wix***.   
       This means that ***wi***is a prefix of ***wj***, e.g. **class** and **classroom**.
    2. If ***wi = x*ai*yi, wj = x*aj*yj***, where ***x, yi*** and ***yj*** are strings (may be empty)  
       and **ai** and **aj** are letters in the alphabet such that **ai < aj**.

Examples:

car and cat: ***x***= ca, **ai** = r,**aj** = t, r < t, ***yi*** and ***yj***are empty.   
result and theory: ***x*** is empty, **ai** = r,**aj** = t, r < t, ***yi***=esult, ***yj*** = heory

stack and string: ***x*** = st, **ai** = a,**aj** = i, a < i, ***yi*** = ck, ***yj*** = ing

* 1. If the alphabet is {0,1} and L is the language containing strings of the type

0, 1, 01,001,0001,011,00011,00001111111, i.e. zeros to the left and 1s to the right,

how this language can be formally defined?

**Solution**

We can describe the language L as the following set:

L = {0n1m, n ≥ 0, m ≥ 0}

Note that this definition assumes that the empty string is a member of the language, while the problem did not say this explicitly. The convention is to assume the empty string to be in the language, and only if we want to consider a language without the empty string, to say this explicitly.

**Learning goals**

* Know the definitions of the concepts described above
* Know the operations on strings and languages

**Exam-like problems**

1. Concatenation of languages corresponds to Cartesian products of sets. Explain why.
2. Give an example of a string **w** such that **w3 = w4**
3. Give an example of a string **w** such that **wi = wi+1**, i is nonnegative.
4. Let L1 = {a}\*, L2 = {b}\*. Give the set representation for L1 È L2, L1 Ç L2

**Finite Representation of Languages  
Regular Expressions and Regular Languages**

* [**Introduction**](http://faculty.simpson.edu/lydia.sinapova/www/cmsc365/LN365_Lewis/L06-RegLang.htm#Introduction)
* [**Regular expressions**](http://faculty.simpson.edu/lydia.sinapova/www/cmsc365/LN365_Lewis/L06-RegLang.htm#expressions)
* [**Regular languages**](http://faculty.simpson.edu/lydia.sinapova/www/cmsc365/LN365_Lewis/L06-RegLang.htm#languages)
* [**Examples**](http://faculty.simpson.edu/lydia.sinapova/www/cmsc365/LN365_Lewis/L06-RegLang.htm#Examples)

[**Learning goals**](http://faculty.simpson.edu/lydia.sinapova/www/cmsc365/LN365_Lewis/L06-RegLang.htm#goals)[**Exam-like problems**](http://faculty.simpson.edu/lydia.sinapova/www/cmsc365/LN365_Lewis/L06-RegLang.htm#exam)

1. **Introduction**

Language: any set of strings over an alphabet ∑, i.e. any subset of ∑\*.

**Grammars**- finite sets of rules used to describe languages.

Grammars are finite representations. Each grammar is in fact a finite sequence of symbols,   
i.e. a string over some alphabet. We know that given an alphabet ∑, the set of all possible strings ∑\* is countably infinite. Hence the set of all possible grammars is countably infinite.

Since each language is a subset of ∑\*, the set of all possible languages is 2∑\* the power set of ∑\*.   
The power set of ∑\* is not countable (the power set of any countably infinite set is not countable).   
This means that the set of all possible languages is not countable, though each language is countable.

Thus on one hand we have the set of all possible grammars to be countably infinite   
and on the other hand we have the set of all possible languages to be uncountable.   
Hence we cannot have a representation for every language.

Within the set of all possible languages we will consider the subset of those languages   
that can be represented by a grammar description. We will see that these languages and   
their grammars can be grouped into four types. We start with the simplest representation   
called **regular expressions**(regular grammars) and the corresponding languages - **regular languages**.

1. **Regular expressions**

Expressions consisting of string concatenations combined with the symbols U and \*, possibly using '(' and ')' are called regular expressions.

**Example:**

If **∑ =**{0,1}, regular expressions are**:**

0,1, 010101, any combination of 0s and 1s

0 U 1, (0 U 1)1\*

(0 U 1)\*01

(0\* U 1\*)

Before discussing the meaning of a regular expression, we'll give the formal definition:

**Formal definition:**

1. *Ø* and each member of **∑** is a regular expression.

2. If α and β are regular expressions, then (α β) is a regular expression

3. If α and β are regular expressions, then α U β is a regular expression.

4. If α is a regular expression, then α\* is a regular expression.

5. Nothing else is a regular expression.

1. **Regular languages**

Each regular expression can be mapped to a language, using a function L

1. L (*Ø* ) = Ø (the empty set) and L ( a ) = {a}, for each **a** in the alphabet

2. If **α** and **β** are regular expressions, then L (α β) = L (α ) L (β)

3**.**If **α** and **β**are regular expressions, then L ((α U β)) = L (α ) È L (β)

4. If **α** is a regular expression, then L (α\*) = (L (α ))\*

5. Nothing else is a regular expression.

**Note:** **U** is used in regular expressions, È is used to denote set union

Applying this function to a regular expression, we obtain strings in a language.   
A language, whose strings can be obtained from a regular expression is called   
a **regular language**.

1. **Examples**

More about the correspondence between a regular expression and the language   
that it describes (details about the function L )

* 1. Rule 1 above says that L ( a ) = {a}. This means that the regular expression **a**corresponds   
     to a language that consists only of one letter - the letter **a**
  2. Rule 2 says that L (α β) = L (α) L (β), where α and β are regular expressions.   
     Let α = a, β = b. We have already seen that the regular expression **a** corresponds to L1 = {a},   
     similarly the regular expression **b** will correspond to the set of one letter L2 = {b}.   
     The regular expression **ab** will correspond to the concatenation of L1 and L2, i.e. to L3 = {ab}

Another example: Let α = ab, β = bc. L (α) L (β) = {ab}{bc} = {abbc}, a language consisting   
of one string only - the string **abbc**.

* 1. Rule 3 says that L ((α U β)) = L (α ) È L (β). Let α = a, β = b.   
     The regular expression **a U b** will describe the set {a} È {b} = {a,b}.

More examples:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Expression:** | **Language:** | | ab U b | L = {ab,b} | | ab U ac | L = {ab,ac} | | a U ab U b | L = {a,b,ab} | |  |
|  |  |
|  |  |
|  |  |

* 1. Rule 4 says that L (α\*) = L (α )\*. Let α = a .   
     The expression **a\*** corresponds to {a}\*, i.e. all possible strings that consist only of **a**.

More examples that use all rules

|  |  |
| --- | --- |
| **Expression** | **Language** |
| **(a U b)\*** | **L = {a,b}\*** |
| **(ab U bc)\*** | **L = {ab,bc}\*** |
| **(a\* U b)** | **L = {a}\* È {b}, i.e. the language consists of the empty string,  all possible strings of a, and the letter b** |
| **(a\* U b\*)** | **L = {a}\* È {b}\*,  the language consists of the empty string,  all possible strings of a and all possible strings of b.** |
| **a(a\* U b)** | **L = { aw : w Î {a}\* È {b}}  L consist of strings that start with an a,  followed by zero or more a's, or by one b.** |
| **a(a U b)\*a** | **L = (awa : w Î {a,b}\*}  L consists of all string that start with an a, followed by any possiblestring consisting of a's and b's, including the empty string,  and end with an a.** |
| **b\*ab\*** | **L = {xay : x,y Î {b}\*}  Note that while the regular expression has b\* both to the left and to the right of a, when representing L as a set of strings we use x and y- different letters for the left and the right parts. This is so because b\* corresponds to any string of b's, and the left part of a string may be different from the right part.** |
| **(a\*ba\*) U bc\*** | **L = { xby : x,y Î {a}\*}È {bz, z Î {c}\*** |
| **ab\*(b U bc\*)** | **L = {axy : x Î {b}\*, y Î {b} È {b}{c}\* }** |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

**Learning goals**

<="" li="">

* Be able to manipulate regular expressions and describe the corresponding regular languages verbally and/or by set representation.
* Given a language described verbally and/or by set representation, be abble to derive the corresponding regular expression.

**Exam-like problems**

1. Problems concerning transformations of regular expressions
2. Given a regular language described verbally / as a set / by a regular expression,   
   derive the other two types of language description.

Study the [**examples**](http://faculty.simpson.edu/lydia.sinapova/www/cmsc365/LN365_Lewis/L06-RegLang.htm#examples) above and see some [**solved problems**](http://faculty.simpson.edu/lydia.sinapova/www/cmsc365/LN365_Lewis/L06-RegLangSol.htm) from the textbook.

|  |
| --- |
| **CmSc 365 Theory of Computation** |
| **Finite State Automata (Finite State Machines) Deterministic Finite State Automata (FSA)**   * [**Introduction**](http://faculty.simpson.edu/lydia.sinapova/www/cmsc365/LN365_Lewis/L07-FSA.htm#intro) * [**Formal Definitions**](http://faculty.simpson.edu/lydia.sinapova/www/cmsc365/LN365_Lewis/L07-FSA.htm#def) * [**Computation and configurations**](http://faculty.simpson.edu/lydia.sinapova/www/cmsc365/LN365_Lewis/L07-FSA.htm#config) * [**Examples**](http://faculty.simpson.edu/lydia.sinapova/www/cmsc365/LN365_Lewis/L07-FSA.htm#examples)   [**Problems with solutions**](http://faculty.simpson.edu/lydia.sinapova/www/cmsc365/LN365_Lewis/L07-Problems.htm)[**Learning goals**](http://faculty.simpson.edu/lydia.sinapova/www/cmsc365/LN365_Lewis/L07-FSA.htm#goals)   1. **Introduction**   FSA (Finite State Automata) are the simplest model of a computing device -  *a language recognition device.*The input to FSAs is strings of symbols, delivered on a tape.  The operation of an FSA proceeds at discrete steps.  At each step of its operation FSA can read one symbol from the tape and can move ahead to read the next symbol. The purpose of FSA is to determine whether a given string written on the tape belongs to a particular language or not. Thus the set of inputs recognized at each particular step is fixed in the description of the particular FSA.  **Informally**, FSA is described by   * 1. A set of **internal states.** At each step of its operation FSA is at some internal state.  Whether the current symbol on the tape is recognized or not depends on  its current internal state. Thus we have the next part of the FSA description:   2. A set of symbols that can be recognized at each internal state of the FSA.  This description is given by a **transition function** - telling the FSA what would be  its next state when a given symbol is recognized.   3. The set of all symbols that can be recognized constitute the **alphabet**.   4. There should be **one initial state** to start from.   5. There should be also **one or more final states** to end the recognition process.   Here is **an example** of the laughing FSA - it recognizes the strings  ha! ha ha ! ha ha ha…ha!    q0, q1, q2, q3 - states of the FSA  FSA can be represented by graphs (drawing the picture of the graph), or by tables:  h a !  --------------------------  q0 q1 - -  q1 - q2 -  q2 q1 - q3  q3 - - -  --------------------------  **Initial state: q0 Final state: q3**  At state q0 FSA accepts **h** and enters q1 At state q1 FSA accepts **a** and enters q2 At state q2 FSA can accept **h** and in that case it enters q1,  or it can accept **!**' and in that case it enters q3, which is the final state.  ♣ FSA recognizes a string (accepts a string) if after the string has been read the FSA is at a final state. Otherwise the string is not recognized by the FSA.  ♣ A language is recognized (accepted) by an FSA if all its strings are recognized by the FSA  ♣ Two FSAs are equivalent if they recognize one and the same language   1. **Formal Definitions**   **Definition:** A deterministic finite automaton is a quintuple **M = (K, ∑, δ, s, F),** where  **K** is a finite set of **states** ∑ is an **alphabet s є K** is the **initial state F  K** is the set of **final states δ** is the **transition function** from **K x ∑** to **K**  If **M** is in state **q є K** and the symbol read is **a є∑,** then **δ (q,a) є K** is the uniquely determined next state of **M** on reading **a**.  **Important:**   * 1. Why deterministic automaton?   Because for a given state and a given input symbol there is not more than one state to enter next. If there were two or more possible next states, the automaton is called **nondeterministic**.   * 1. What if **a є ∑**, however for some **q** the function **δ(q,a)** is not given?   In that case the FSA is called *incompletely defined*,  i.e. the domain of the transition function is a subset of **K x ∑ : D(δ)  K x ∑.**  An incompletely defined FSA can be extended to a completely defined FSA  by introducing a new state **q'**, which is not final, and adding  **δ(q,a)= q'** for each pair **(q,a)** for which **δ(q,a)**has not been initially defined, i.e.  **δ(q',a)= q'** for each **a є∑**.  The new FSA will recognize all strings recognized by the incompletely defined FSA.  If a symbol is read for which the transition function of the initial FSA was not specified,  the new FSA will enter the state **q'**and will stay there till the end of the string.  Since the new state **q'**is not a final state, at the end the FSA will not be at a final state,  which means that the string is not recognized.  **Example:**  Making the laughing machine a completely defined FSA:  h a !  ---------------------------  q0 q1 q4 q4  q1 q4 q2 q4  q2 q1 q4 q3  q3 q4 q4 q4  q4 q4 q4 q4  ----------------------------  Initial state: q0,  Final state q3.  **q4:**Additional state to make the FSA completely defined:   1. **Computation and configurations**   **Definition:***Computation*by an FSA on an input string is a **sequence of configurations**  that represents the status of the machine at successive moments.  A configuration is determined by the current state and the unread part of the string. Hence a **configuration is any element (q,*w*) in** **K x ∑\***  **Definition:** If **(q,*w*)** and **(q',*w*')** are two configurations,  then **(q,*w*) yields** **(q',*w*')** **in one** **step** if and only if ***w* = a*w*'** for some **a є∑,** and **δ(q,a)= q'**  This is written as:  **(q,*w*)** ├ M **(q',*w*')**  The transitive closure of ├ Mis denoted by ├ M\*.  **(q,*w*)** ├ M\* **(q',*w*')**means that **(q,*w*) yields (q',*w*')** after some number of steps**.**    **Definition:**A string w є ∑\* is said to be accepted by M  iff there is a state q є F such that **(s,w) ├ M\* (q,e)**  **Definition:**The language accepted by **M**, L(M), is the set of all strings accepted by **M.**   1. **Examples**    1. An FSA that accepts even number of **b**'s   http://faculty.simpson.edu/lydia.sinapova/www/cmsc365/LN365_Lewis/L07-Fig03.jpg  Initial state: q0 Final state: q0  Input **aaabba**  (q0, aaabba) ├ M(q0, aabba)  ├ M(q0, abba)  ├ M(q0, bba)  ├ M(q1, ba)  ├ M(q0, a)  ├ M(q0, e)  Since q0 is a final state, the string was accepted  Input **aaabbba**  (q0, aaabbba) ├ M(q0, aabbba)  ├ M(q0, abbba)  ├ M(q0, bbba)  ├ M(q1, bba)  ├ M(q0, ba)  ├ M(q1, a)  ├ M(q1, e)  q1 is not a final state, so the string is not accepted   * 1. An FSA that accepts strings not containing three consecutive b's   Initial state: q0 Final states: q0,q1,q2  Input**: baabb**  (q0, baabb) ├ M(q1, aabb)  ├ M(q0, abb)  ├ M(q1, b)  ├ M(q2, e)  Since q0 is a final state, the string was accepted  Input**: abbaabbba**  (q0, abbaabbba) ├ M(q0, bbaabbba)  ├ M(q1, baabbba)  ├ M(q2, aabbba)  ├ M(q0, abbba)  ├ M(q0, bbba)  ├ M(q1, bba)  ├ M(q2, ba)  ├ M(q3, ba)  ├ M(q3, a)  ├ M(q3, e)  Since q3 is not a final state, the string was not accepted  **Learning Goals**   1. Know the formal definition of a Finite State Automaton 2. Distinguish between incompletely defined and completely defined FSA 3. Be able to extend an incompletely defined FSA to a completely defined one 4. Be able to determine the set of strings accepted by an FSA 5. Be able to construct an FSA that recognizes regular expressions of the following type:    * having a repeated pattern, e.g. **(aab)\*** as in **aabaabaabaab**    * ending with a given pattern, e.g. **(a U b)\*bab** as in **aaaaaabab, aabbab**    * starting with a given pattern, e.g. **bba(a U b)\*** as in **bbababbaaaabababaaa, bbabba, bbaababa**    * having no more than **n**consecutive occurrences of a given symbol    * having at least **n**consecutive occurrences of a given symbol 6. Be able to solve problems similar to the presented examples and [**problems**](http://faculty.simpson.edu/lydia.sinapova/www/cmsc365/LN365_Lewis/L07-Problems.htm)   **Exam-like problems**   * Study the presented examples and [**solved problems**](http://faculty.simpson.edu/lydia.sinapova/www/cmsc365/LN365_Lewis/L07-Problems.htm) |
| [Back to Contents page](http://faculty.simpson.edu/lydia.sinapova/www/cmsc365/LN365_Lewis/Contents.htm)  *Created by*[*Lydia Sinapova*](http://faculty.simpson.edu/lydia.sinapova/www/index.htm) |

**Finite State Automata (Finite State Machines)  
NonDeterministic Finite State Automata**

* [**Description of a non-deterministic FSA**](http://faculty.simpson.edu/lydia.sinapova/www/cmsc365/LN365_Lewis/L08-NondetFSA.htm#Intro)
* [**Learning goals**](http://faculty.simpson.edu/lydia.sinapova/www/cmsc365/LN365_Lewis/L08-NondetFSA.htm#goals)
* [**Exam-like questions and problems**](http://faculty.simpson.edu/lydia.sinapova/www/cmsc365/LN365_Lewis/L08-NondetFSA.htm#exam)

**Description of a non-deterministic FSA**

An **incompletely specified FSA** is one whose transition function is not defined on all pairs (q,a),   
where **q** is a state, **a** is a symbol from the alphabet.

A **nondeterministic FSA** is one, where for some state **q**and a symbol **a** we may have more than one transition. Here we do not have a transition function, as all functions have to specify unique images.   
Here we have a **transition relation**  K x { ∑  e } x K.

Note that the empty symbol is included in the transition relation.

**Example:**

This FSA accepts ( ab U aba)\*

In state q1 there is a choice. We can either read a and go to q0 or simply “jump” to q0 without reading a symbol. Note that this FSA is incomplete. Depending on the choice the final state may not be reached for a string, that is intended to be accepted by the FSA. However, there is a possible sequence of choices, such that the final state is reached for that string.

**Definition:** A deterministic finite automaton is a quintuple **M = (K, ∑, Δ, s, F),** where

**K** is a finite set of **states**∑ is an **alphabet  
s є K** is the **initial state  
F  K** is the set of **final states  
Δ** is the **transition relation**, a subset of **K x (∑  e) x K**

A transition is a triple (q, u, p), where q and p are in K, and u is in ∑  e.  
If a transition (q, e, p) is followed, no input symbol is read.

A **configuration** is any element (q,*w*) in **K x ∑\***

**Definition:** If **(q,*w*)** and **(q',*w*')** are two configurations,   
then **(q,*w*) yields** **(q',*w*')** **in one** **step** if and only if ***w* = u*w*'**, where u є ∑  e, and (q, u, q**'** ) є Δ.

This is written as:

**(q,*w*)** ├ M **(q',*w*')**

**(q,*w*)** ├ M\* **(q',*w*')**means that **(q,*w*) yields (q',*w*')** after some number of steps. ├ M\* denotes the transitive closure of ├.

**Definition:**A string w є ∑\* is said to be accepted by M   
iff there is a state q є F such that **(s,w) ├ M\* (q,e)**

**Definition:**The language accepted by **M**, L(M), is the set of all strings accepted by **M.**

So we say that **a non-deterministic FSA accepts a string, if there is a possible path of transitions from the initial state to a final state for that string**.

In summary there are two essential differences between deterministic and non-deterministic automata:

1. Deterministic FSAs are defined with a transition function, while nondeterministic FSAs are defined with transition relation
2. Each transition in a deterministic FSA is upon reading a symbol from ∑, while in a non-deterministic FSA there may be transition upon the empty symbol (such transitions are usually called “jumps”)

There is a theorem stating that **for each nondeterministic FSA there is an equivalent deterministic FSA**.

**Learning Goals**

* Be able to distinguish between a deterministic and a nondeterministic FSA

**Exam-like questions and problems**

1. Where is the difference between a deterministic FSA and a non-deterministic FSA?
2. When do we say that a non-deterministic FSA accepts a string?
3. Construct an FSA that accepts each of the following sentences:
   1. Tom is a banker.
   2. Tom is a rich banker
   3. Tom is happy
   4. Tom is very happy
   5. Tom is a rich banker and is very very happy.

Consider each word as a symbol in an alphabet.   
What other sentences would be accepted by the FSA?

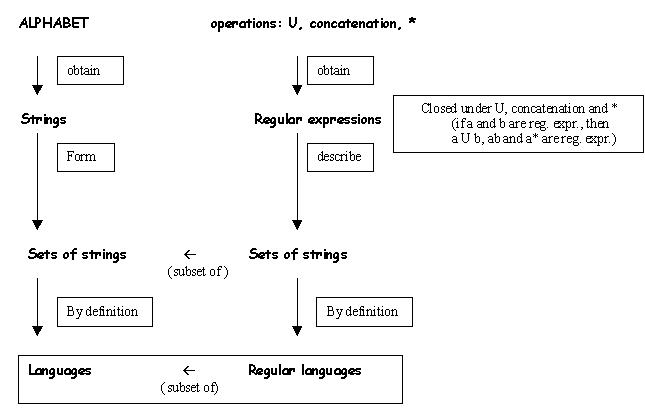
1. Problems: 2.2.1, 2.2.2, 2.2.3 in the textbook.

**Finite State Automata and Regular Expressions**

* [**Concepts' chart**](http://faculty.simpson.edu/lydia.sinapova/www/cmsc365/LN365_Lewis/L09-FSAReg.htm#chart)
* [**Theorem**](http://faculty.simpson.edu/lydia.sinapova/www/cmsc365/LN365_Lewis/L09-FSAReg.htm#theorem)
* [**How to construct FSAs**](http://faculty.simpson.edu/lydia.sinapova/www/cmsc365/LN365_Lewis/L09-FSAReg.htm#construct)
* [**Example**](http://faculty.simpson.edu/lydia.sinapova/www/cmsc365/LN365_Lewis/L09-FSAReg.htm#ex)

[**Learning goals**](http://faculty.simpson.edu/lydia.sinapova/www/cmsc365/LN365_Lewis/L09-FSAReg.htm#goals)[**Exam-like questions**](http://faculty.simpson.edu/lydia.sinapova/www/cmsc365/LN365_Lewis/L09-FSAReg.htm#exam)

1. **Concepts' chart**



1. **Theorem**

**The class of languages accepted by finite automata is closed under**

* 1. union
  2. concatenation
  3. Kleene star
  4. complementation
  5. intersection

**Proof**

* 1. **union**

Let M1 = (K1, ∑, Δ1, s1, F1), M2 = (K2, ∑, Δ2, s2, F2)  
We construct a non-deterministic automaton M = (K, ∑, Δ, s, F), where

K = K1  K2  {s}  
F = F1  F2  
Δ = Δ1  Δ2  {(s, e, s1), (s, e, s2)}

We introduce a new start state **s**, and M “guesses” whether the input is for M1 or M2.  
Formally,

if w є ∑\* then   
(s,w) ├ \*M (q, e) for some q є F iff

either (s1, w) ├ \*M1 (q, e) for some q є F1   
or      (s2, w) ├ \*M2 (q, e) for some q є F2

Hence M accepts w iff M1 accepts w or M2 accepts w, thus L(M) = L(M1)  L(M2)

* 1. **concatenation**

We will construct an automaton M that accepts L(M) = L(M1)  L(M2)  
Let M1 = (K1, ∑, Δ1, s1, F1), M2 = (K2, ∑, Δ2, s2, F2)  
We construct a non-deterministic automaton M = (K, ∑, Δ, s, F), where

K = K1  K2   
s = s1  
F = F2  
Δ = Δ1  Δ2  {(q, e, s2)| q є F1}

Here the idea is the connect all final states of M1 with empty links to the start state of M2.

Formally,

if w є ∑\* then   
(s,w) ├ \*M (q, e) for some q є F iff

w = uv, and

(s1, u) ├ \*M1 (q, e) for some q є F1, i.e. u є L(M1)  
(s2, v) ├ \*M2 (q, e) for some q є F2, i.e. v є L(M2)

Hence M accepts w iff M1 accepts part of w and then M2 accepts the remaining part. Thus L(M) = L(M1) L(M2)

* 1. **Kleene star**

Let M1 = (K1, ∑, Δ1, s1, F1)  
We construct a non-deterministic automaton M that accepts L(M) = L(M)\* in the following way:  
M = (K, ∑, Δ, s, F), where

K = K1  {s}   
F = F1  {s}  
Δ = Δ1  {(s, e, s1) } { (q, e, s1) | q є F1}

The idea here is to have a new start state connected to s1 with an empty link, so that M accepts the empty string, and to have e-transitions from all final states of M1 to s1, so that once a string has been read, the computation can resume from the initial state.

* 1. **complementation**

Let M = (K, ∑, δ, s, F), accepting L(M). The complement of L(M) is ∑\* - L(M). It is accepted by a deterministic FSA constructed in the following way:

M' = (K, ∑, δ, s, K - F)

M' is identical to M except that the final and non-final states are interchanged.  
Thus a string that is accepted by M will not be accepted by M’ and vice versa.

* 1. **intersection**

Here we apply the equality:

L1  L2 = ∑\* - ((∑\* - L1)  (∑\* - L2))

The closedness under intersection follows from the closedness under union and complementation.

How to construct M (we consider deterministic complete automata):  
Let M1 = (K1, ∑, δ1, s1, F1), M2 = (K2, ∑, δ2, s2, F2)

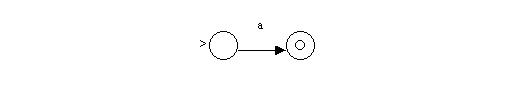
We construct a deterministic automaton M = (K, ∑, δ, s, F), where

K = K1 x K2  
s = (s1,s2)  
F = {(q, p) | q є F1, p є F2}  
δ = { (q1, p1), a, (q2,p2)) | (q1,a, q2) є δ1,(p1,a,p2) є δ2 }

Let M1 and M2 are accepting L(M1) and L(M2) respectively.  
The new automaton M has to accept strings { w | w L(M1)  L(M2)}, i.e. both M1 and M2 on w have to stop in final states.

1. **How to construct FSAs**

**Rule 1:** An FSA that accepts a string of one letter or the empty string:



When applying Rule 2, Rule 3, and Rule 4 below, assume that an FSA has one final state only.   
If there are more final states, we can replace them by internal states and link them   
to a new final state by the empty string.

**Rule 2: Sequential linking** of two FSAs.

We link the final state of the first FSA with the starting state of the second FSA by an empty link.   
The final state of the first FSA is not a final state in the new FSA.   
The starting state of the second FSA is not anymore a starting state in the new FSA.   
Thus we obtain a new FSA. If the first FSA accepts L1, and the second FSA accepts L2,   
then the new FSA will accept L1L2

**Rule 3: Parallel linking** of two FSAs

* 1. Introduce a new starting state.
  2. Link the new starting state with the starting states of each FSA by an empty link   
     (the starting states of the two FSAs are no more starting states)
  3. Introduce a new final state
  4. Link the final states of each FSA to the new final state by an empty link.   
     The final states of the two FSAs are no more final states.

If the first FSA accepts L1, and the second FSA accepts L2, the new FSA will accept L1  L2

**Rule 4: Kleene star** on a FSA.

Given an FSA that accepts the language L, how do we build an FSA hat accepts L\*?

We link its starting state to its final state by an empty link,   
and its final state to its starting state by an empty link.

**Rule 5: Complementation**

Here we simple interchange the final and the non-final states.

**Rule 6: Intersection**

See part e) in the theorem above.

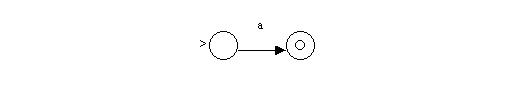
1. **EXAMPLES**

Rule1, Rule2 Rule3 and Rule4 give the method to build an FSA that accepts a language represented by some regular expression.

**Example 1**

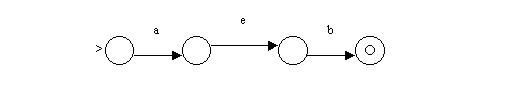
Build an FSA that accepts **(ab U b\*)\***

* 1. We build an FSA that accepts 'a' - FSA1 (Rule 1)

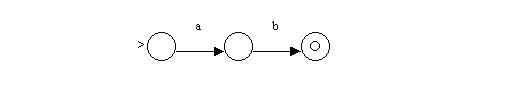


* 1. We build an FSA that accepts 'b' - FSA2 (Rule 1)

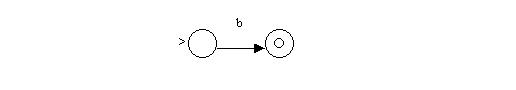
* 1. We link these two FSA sequentially - FSA3 (Rule 2)



Here we can simplify the FSA:



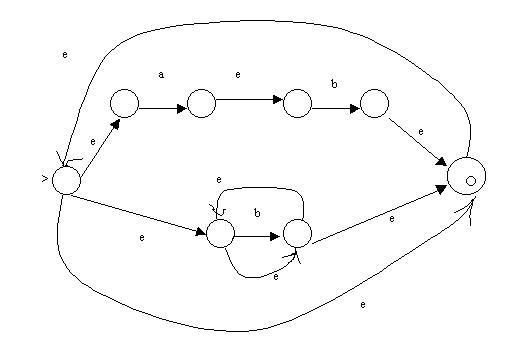
* 1. Next we build an FSA that accepts b (same as in 2.): - FSA4 (Rule 1)



* 1. The FSA that accepts b\* will be FSA5 (Rule 4)

* 1. Now we link in parallel FSA3 and FSA5. The new FSA is FSA6 (Rule 3)

* 1. Last we build FSA7 = FSA6\* (Rule 4)



**Example 2**

Construct automaton that accepts L1  L2, where

L1 = {w є ∑\* | w contains even number of a’s}  
L2 = {w є ∑\* | w contains odd number of b’s}

Let M1 accept L1, and M2 accept L2. We will describe M1 and M2 by transition tables:

M1:

K1 = {q1, q2}

S1 = q1

F1 = {q1}

transition table:

a b

q1 q2 q1

q2 q1 q2

M2:

K2 = {p1, p2}

S2= p1

F2 = {p2}

transition table:

a b

p1 p1 p2

p2 p2 p1

The new automaton M will have the following states:

K = {(q1, p1), (q1, p2), (q2, p1), (q2, p2)}  
S = (q1, p1)  
F = {(q1, p2)}

The transition table is:

a b

(q1, p1) ( q2, p1) (q1, p2)

(q1, p2) ( q2, p2) (q1, p1)

(q2, p1) ( q1, p1) (q2, p2)

(q2, p2) ( q1, p2) (q2, p1)

We will show now that the string **aabaabb** is accepted, while the strings **aabb**and **abbb** will not be accepted

((q1, p1), aabaabb) ├ ((q2, p1), abaabb) ├ ((q1, p1), baabb) ├ ((q1, p2), aabb) ├ ((q2, p2), abb) ├ ((q1, p2), bb) ├   
((q1, p1), b) ├((q1, p2), e)

(q1, p2) is a final state in M, so the string **aabaabb** is accepted

((q1, p1), aabb) ├ ((q2, p1), abb) ├ ((q1, p1), bb) ├ ((q1, p2), b) ├ ((q1, p1), e)

(q1,p1) is not a final state, so **aabb** is not accepted

((q1, p1), abbb) ├ ((q2, p1), bbb) ├ ((q2, p2), bb) ├ ((q2, p1), b) ├ ((q2, p2), e)

(q2, p2) is not a final state, so **abbb** is not accepted

**Learning goals**

* Know how the basic concepts concerning languages and FSA are related.
* Know the method to construct an FSA that accepts a represented by a regular expression.

**Exam-like questions**

* Give the definition of a regular language
* What is the class of languages accepted by FSAs?
* Construct FSAs to accept the following languages:
  + a\*(ab U ba U e)b\*
  + ((a U b)\*(e U c)\*)\*
  + ((ab)\*U(bc)\*)ab
  + ((ab U aba)\*a)\*
* Problem 2.3.7 in the textbook
* Other [**problems with solutions**](http://faculty.simpson.edu/lydia.sinapova/www/cmsc365/LN365_Lewis/L09-Problems.htm)