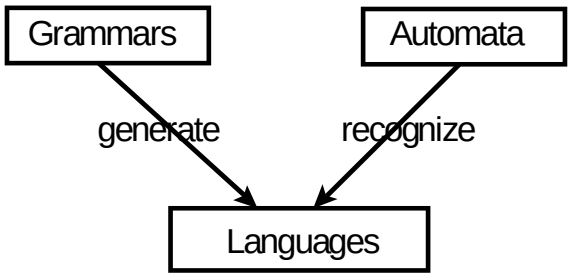
Languages, Automata, Grammars, and Regular Expressions

Note: some of this is material you should have seen in CSc 28 (or its equivalent).

Languages, automata and grammars (regular expressions although not technically grammars function are substitutes for simple languages) are three related concepts. These concepts are particularly useful in:

- Understanding Compilers
- Software Specification
- Language Definition (especially programming languages)



Basic Definitions

Alphabet: a finite collection of atomic symbols ("atomic" means indivisible).

Examples: A = { a, b, c, ... , y, z} B = { 0, 1} C = { a, b}

D = {the, elephants, like, peanuts}

String: a sequence of atomic symbols. We say that a string is a string "over" (or "on") the alphabet.

Examples: *abbcehcs* is a string over A

0011010001 is a string over B

aabba is a string over C

elephants peanuts the like is a string over D (ignore the spaces)

λ denotes the null string

Concatenation: the basic operation on strings. This operation is denoted by the operator symbol "dot" = • often simply denoted by juxtaposition, i.e., a•b = ab.

Language: a specified set of strings over some specified alphabet.

Imagine all possible strings over a given alphabet A.

(How many are there if A = {0, 1}?)

Regular Expressions

In practical applications regular expressions are often used to define search patterns. A regular expression is a "meta-language" that is a language that describes another language (i.e., a set of strings). In the present context, a regular expression can be viewed as a **pattern** to generate strings of a desired language (i.e. a set of strings over some alphabet A), or alternatively, as a pattern to match given strings to. It is made up of the letters of the described language alphabet, as well as the following special characters:

( ) • * + λ

They are used as follows:

( )	grouping
*	repetition
•	concatenation (often omitted)
+	a choice ("or")
λ	a special symbol denoting the null string

[Precedence from highest to lowest: ( ) • * +]

Examples: (A = {*a, b*} unless otherwise stated)

a • b • a (or just aba)	matched only by the string *aba*
ab + ba	matched by exactly two strings: *ab* and *ba*
b*	matched by { λ, b, bb, bbb, ...}
b(a + ba*)*a (b + λ)	matched by bbaaab

Formal (recursive) Definition of a Regular Expression

1. if **a** ∈ A, then **a** is a regular expression
2. λ is a regular expression
3. if r and s are regular expressions, then the following are also regular expressions: r*, r • s = rs, r + s, (r)

Some convenient extensions to the regular expression notations:

$aa = a^2$, $bbbb = b^4$, etc.

$a^+ = a \bullet a^*$ = { any string of a's of positive length, i.e. excludes λ }

Hence, given the alphabet {a, b}:

$(ab)^2 = abab \neq a^2 b^2$ so don't use "algebra" (although we still have $a^2 . a^3 = a^5$, etc.)

$(a+b)^2 = (a+b)(a+b)$ *aa* or *ab* or *ba* or *bb,*

(a+b)* **any string! this is quite important!!**

Sample problems:

Construct a regular expression (over {a, b}) that exactly describes:

- all strings that begin with a and end with b
- all non-empty strings of even length
- all strings with at least one b
- all strings with at least two a's
- all strings of one or more b's with an optional single leading a
- the language { ab, ba, abaa, bbb }

Tips:

Check the simplest cases
Check for “sins of omission” (forgot some strings)
Check for “sins of commission” (included some unwanted strings)

Your turn...

Find a regular expression for the following sets of strings on {a, b}:

All strings with exactly two b’s.

All strings with no more than two b’s.

All strings with at least one a, and at least one b.

All strings which end in a double letter.

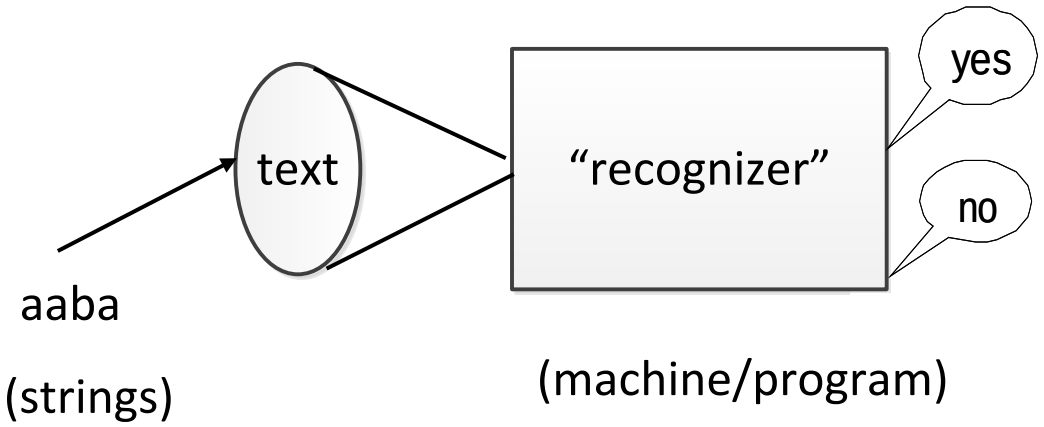
All strings of even length (includes 0 length).

All strings of odd length.

All strings not ending in aa.

All strings where no b is followed immediately by another b.

Finite State Automata



A finite automaton consists of five pieces:

1. Q = a finite set of (internal) states
2. Σ = a set of input symbol (input alphabet)
3. δ = a function δ: Q x Σ → Q, describing the "transitions" between states according to fixed rules, in response to one of finitely many different possible inputs. It looks like:
   $\delta(q_x, c) = q_y$ if you are in q_x, reading c, move to q_y
   This transition function F can be presented in the form of a table (called a transition table), or of a graph (called a transition diagram).
4. q_0 = a designated "start" or "initial" state from Q
5. F = a subset of Q, designated as "accepting" (or "yes", or "final") states

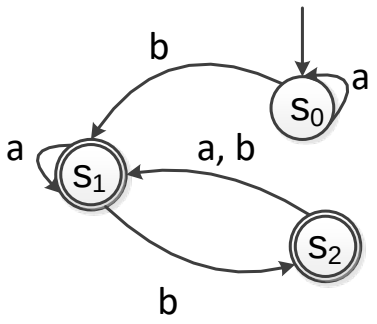
Example: Σ= {*a, b* }, Q = { s_0 , s_1 , s_2 }, F = { s_1 , s_2}, initial is s_0

current input ⇒	Q	a	b
⇑	s_0	s_0	s_1
current state	s_1	s_1	s_2
⇓	s_2	s_1	s_1

Transition Table

Gives the next state

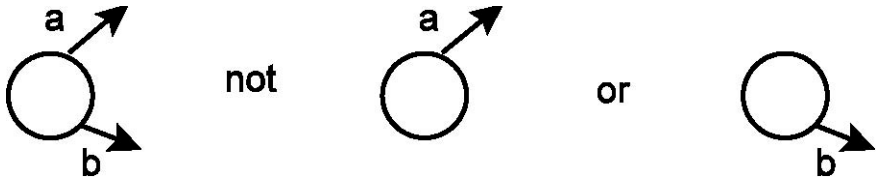
More common and useful:



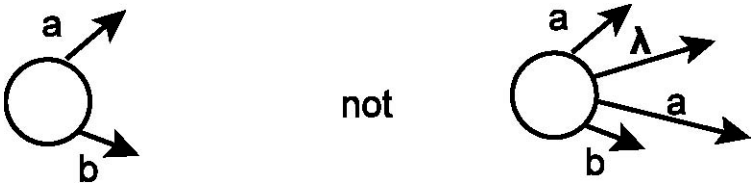
Transition diagram (technically this is a "network": a directed graph with "weighted /annotated" edges).

Note two *important characteristics* of this FSA. It is:

1. *Complete*: No undefined transitions



2. *Deterministic*: No choices

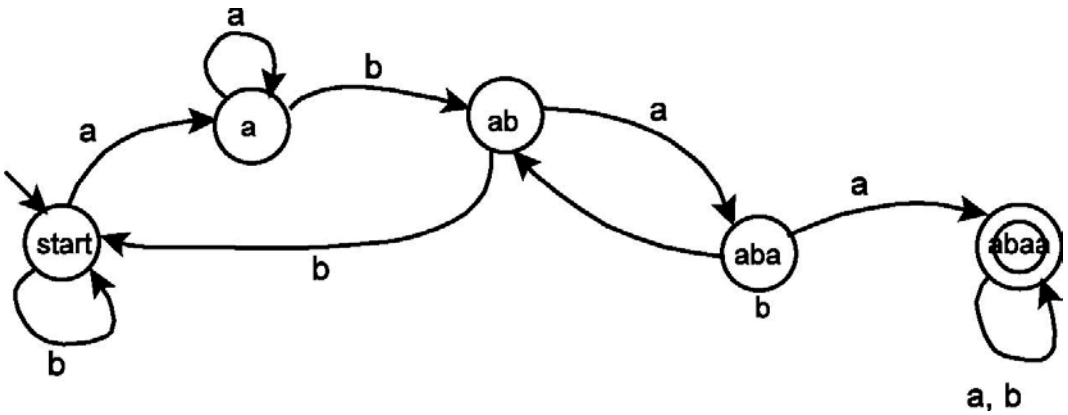


The usual problem format is: given some verbal description of a language, construct a (complete, deterministic) FSA accepting **exactly** that language.

Note: **exactly** is usually the difficult part!!!
"Skeleton Method": A useful solution technique.
The "skeleton" is a sequence of states assuming legal input.

First, construct a skeleton then complete it by presuming that no additional states will be needed.
The completion is guided by the need to have the FSA **complete and deterministic:** for A= { a,b }, every state has exactly two arcs leaving it, one labeled "a" and one labeled "b".

Example (skeleton): All strings containing *abaa.*



Your turn . . .

Assume A= { a, b}. Construct the following automata which:

1. Accepts any string.

2. Accepts λ only.

3. Accepts strings which begin with a

4. Accepts strings of the form (a+b)*

5. Accepts strings containing 'aa' (skeleton method)

6. All words containing at least two a's

7. All words containing exactly two a's

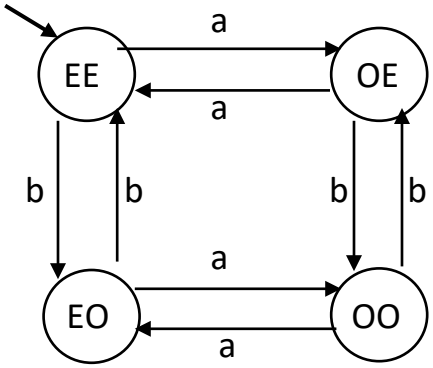
8. All words containing aa, or bba, or both

9. All words containing 11 but not 111.

10. All strings not finishing in aa

Another example:

Let us construct an FSA which counts the number of a's and b's in strings on { a, b} to determine there are an odd or even number of each. Note that there are only four possible cases: even/even, even/odd, odd/even, and odd/odd. if we denote by X the status (E, or O) for the a's, and Y the status for the b's and XY to denote the combination there are four possible values for XY: EE, EO, OE, and OO. Therefore the corresponding FSA needs only four states labelled EE, EO, OE, OO. The FSA (without final state(s) will look like:

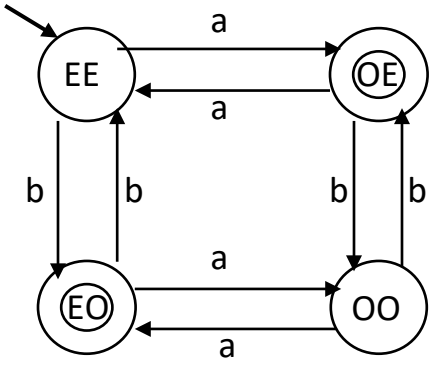


The location of the final state(s) determines the language accepted.

Let us look at the language:

L = { strings on {a,b}, of odd length}

The corresponding FSA will be:



What would be the final state(s) for the following language:

L = { strings on {a,b}, where the number of a's is even}

Relation between Regular Expressions and Automata

Note: although regular expressions generate languages, the way grammars do, they are not regarded as grammars and they pertain to a very specific group of languages, the regular languages. There is no equivalent for the more complex classes of languages.

Some Facts to Remember

1 For every regular expression r, defining a language L, there is a FSA M recognizing exactly L.

2 For every FSA M, recognizing a language L, there is a regular expression r matching all the strings of L **and no others**. (we will prove this later)

Question: is there a FSA recognizing the language { λ, ab, aabb, aaabbb, . . . }, in other words the language { $w = a^n b^n \mid n \geq 0$}?

Answer: No, there is not because we need to "remember" how many a's have been seen before we can verify that there are as many b's. Since an FSA can only count a's via its states and it has only a finite number of states, it cannot count infinitely many a's. Consequently there is no regular expression capable of describing that language either. We need a more powerful kind of "recognizer" and a more powerful kind of generator as well. We will see both later.

Formal Grammars

A more powerful kind of generators. They are much more general than regular expressions as they can generate many different types of languages (Also called: phrase-structured grammars, or just plain "grammars").

Context-Free Grammar (CFG)

A **"context-free" grammar** G consists of four elements:

1. a finite set Σ of "terminals" (usually denoted by lower-case letters).
2. a finite set N of "non-terminals" (usually denoted by upper-case letters), $\Sigma \cap N = \varnothing$ (i.e. no symbol can be both a terminal and a non-terminal)
3. a unique "start symbol" $S \in N$
4. a finite set P of "productions" of the form $A \rightarrow \omega$, where $A \in N$ and $\omega \in (\Sigma + N)^*$ (this is the set of all strings on both Σ and N). These are understood to be replacement rules."

In other words, a grammar G is defined as $G = (\Sigma, N, S, P)$

Note that sometimes $N \cup \Sigma$ is denoted V and called the vocabulary.

Notes:

- Two rules with the same left-hand side may be combined using an OR ("|").

  For example, the two rules:

  A ➜ aBb

  A ➜ Aa

  can be combined into:

  A ➜ aBb | Aa

  Note that even if combined it is still two rules
- The language generated by the grammar G is denoted by L(G).
- Sometimes "::=" is used instead of ➜. This is within a notation called Backus-Naur Form (BNF). Also called Backus Normal Form. We will see that in more details shortly.
- Usually lower case letters at the end of the alphabet, such as w, denote strings of terminals only. Greek letters usually denote strings of terminals and non-terminals.
- A grammar describes the syntax rules for forming strings (also called sentences). It also allows us to tell if a particular string is "well-formed," or "legal."
- No appropriate grammar has been developed so far for natural languages such as English or French. This is due to the fact that such languages are inherently ambiguous and their interpretation is dependent on context.

An Example inspired by the English language:

The grammar:

<sentence>	➜	<noun-phrase> <predicate>
<noun-phrase>	➜	<article> <noun>
<predicate>	➜	<verb>
<article>	➜	the | a | an
<noun>	➜	cat | flower
<verb>	➜	jumps | blooms

Would generate the sentences "*the cat jumps*." and "*a flower blooms.*"

Note that a grammar only describes syntactically correct sentences, not necessarily meaningful ones. For example the above grammar will generate sentences such as "*the cat blooms*," or " a flower jumps" which of course do not make sense.

A grammar describes the *syntax* (the form) and not the *semantics* (the meaning). However, the syntax may, indirectly, affect the semantics (more later on this topic).

Two ways that grammars are used in relation to computer programs:

1. Given a grammar G, and a string ω, determine whether or not ω $\in$ L(G). This is referred to as "parsing" the string ω. This is a task that the front-end of a compiler takes care of (the "scanner" and the "parser").
2. Given a verbal description of a language L, construct a grammar G which *(precisely & concisely*) generates L. This is an important Computer Science skill. It allows the use of Unix tools such as *yacc and bison.*

Example for #1:

Consider the grammar G with the following rules:

S ➜ Aa
S ➜ AS
A ➜ b
A ➜ λ

Is the string **ba** $\in$ L(G)? That is, is **ba** "legal" according to grammar G?

Answer: The string **ba** is legal only if it can be generated by G. We can demonstrate this by producing a "derivation" which produces **ba**. A Derivations consist of sequences of sentential forms separated by the symbol "⇒". We distinguish leftmost/ rightmost derivations where the non-terminals being replaced are systematically the leftmost/rightmost non-terminals.

To produce a derivation: start with S. Apply one rule at a time to one nonterminal replacing it with the right side of an applicable rule. Don't quit until only terminals are left in the string being generated. This string of terminals is said to be an element of L(G) = "the language generated by the grammar G."

In the present case here is a derivation producing **ba**.

S ⇒ AS ⇒ AaA ⇒ baA ⇒ ba

Example for # 2:

Build a grammar for all strings on {a, b} that begin with **a** and end with **b**.

Solution 1:

S	➜	aAb	start with an **a** and end with a **b**
A	➜	aA | bA | λ	produces **a** or **b**, repeatedly, quit with λ

Solution 2:

S	➜	aA	start with an **a**
A	➜	aA | bA | B	produces **a** or **b**, repeatedly
B	➜	b	produces a final **b** and quit

These grammars fall in the category of regular grammars which is a subset of context-free grammars.