Finite State Automata and Regular Languages

Translated and adapted by L. Breveglieri

FINITE STATE AUTOMATA AND HOW TO RECOGNIZE A REGULAR LANGUAGE

PROBLEM: recognizing whether a string belongs to a given language, before elaborating it in some way (i.e., before doing semantic analysis)

In order to describe a string recognition procedure, abstract machines (of various kinds) called AUTOMATA are used

In the following these five issues will be presented and discussed

1. finite state automata and their relationship with regular languages
2. the non-deterministic and deterministic behaviours of the recognizer
3. the construction of the minimum automaton (smallest # of states)
4. the composition of recognizers by means of various operations that preserve the structure of the language family (closure)
5. how to exploit finite state automata to analyze languages

RECOGNITION ALGORITHM

DOMAIN: a set of strings over the alphabet Σ
IMAGE: the answer *yes* or *no* (recognition is a decision problem)

Applying the recognition algorithm α to the string *x* is indicated as α (*x*).
The string *x* is *recognized* (*accepted*) or *unrecognized* (*rejected*)
by α if α (*x*) = yes or α (*x*) = no, respectively

The language L(α) is defined as $\boxed{L(\alpha) = \{ x \mid \quad x \in \Sigma^* \ \wedge \ \alpha(x) = yes \}}$

If the language L is semidecidable (= recursively denumerable, but not recursive), it may happen that for some uncorrect string *x* the algorithm α will not terminate, i.e., α (*x*) is undefined

Here only decidable languages are considered, which have a recognition algorithm that terminates for every string, no matter if accepted or rejected

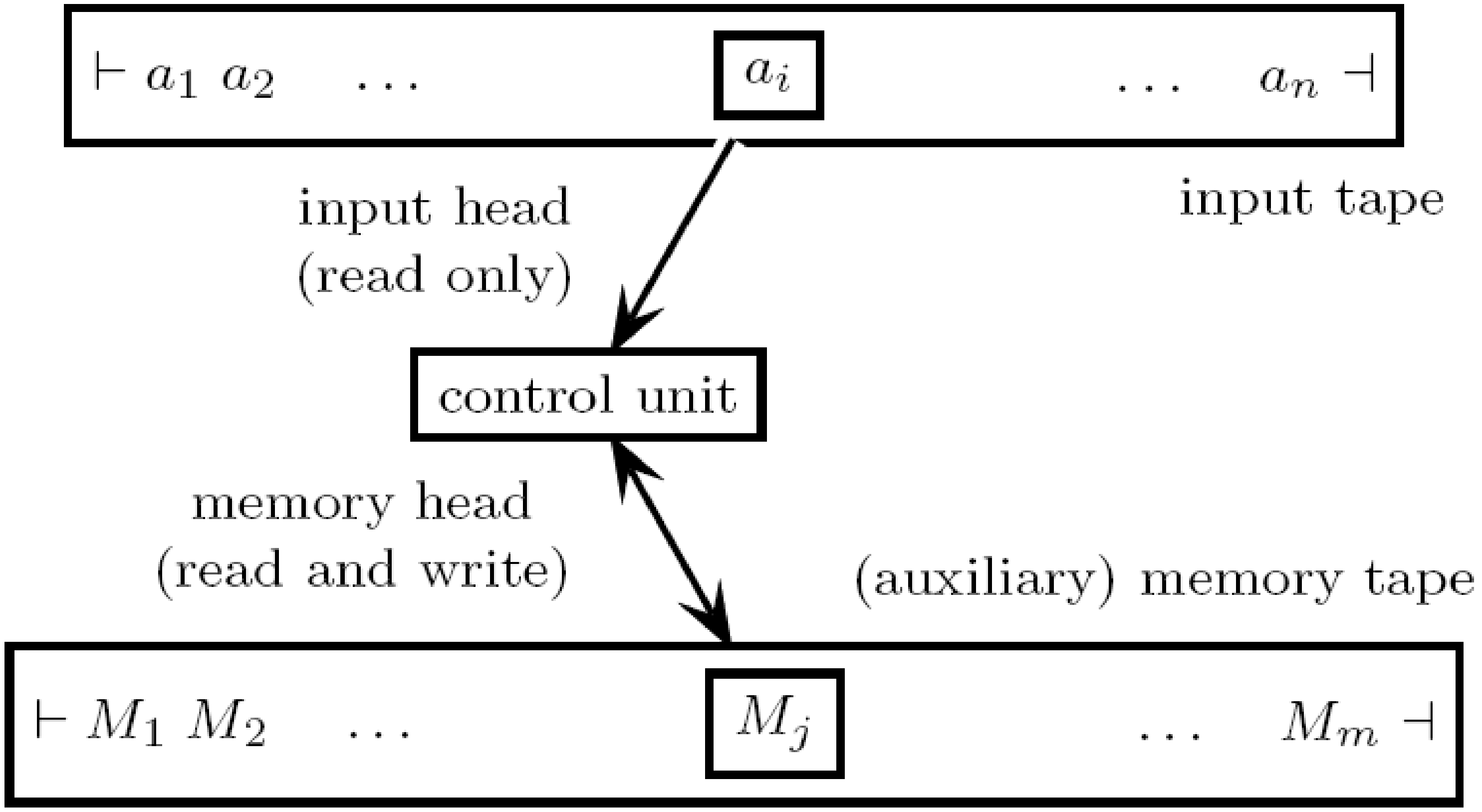
Almost all the problems of some interest here, have solutions with a relatively low time complexity degree (= the number of the computation steps executed by the recognition algorithm), i.e., a linear or at worst a polynomial time complexity, with respect to the size of the problem, which in most cases is the length of the string

In the theory and practice of formal languages, one computation step is a single atomic operation of the abstract recognition machine (automaton), which can manipulate only one symbol at a time. Therefore it is customary to present the recognition algorithm by means of an automaton of some kind, no matter if a recognizer or a transducer machine, mainly for the following reasons

1. outlining the correspondence between the various families of languages and the respective generative devices, i.e., their grammars
2. skipping any unnecessary and premature reference to the effective implementation of the algorithm in some programming language

Once the recognizer automaton has been designed, it is relatively easy to write the corresponding program, in C or similar programming languages

GENERAL MODEL OF A RECOGNIZER AUTOMATON


$\vdash a_1\ a_2\ \ldots\ \boxed{a_i}\ \ldots\ a_n \dashv$
input head
(read only)
input tape
control unit
memory head
(read and write)
(auxiliary) memory tape
$\vdash M_1\ M_2\ \ldots\ \boxed{M_j}\ \ldots\ M_m \dashv$

THE AUTOMATON ANALYZES THE INPUT STRING and executes a series of moves. Each move depends on the symbols currently pointed by the heads and also on the current state. The move may have the following effects

1. shifting the input head of one position to the left or right
2. replacing the current memory symbol by a new one and shifting the memory head of one position to the left or right
3. changing the current state

Some of the operations listed above may be absent

ONE-WAY AUTOMATON: the input head can be shifted only to the right. In the following only this case is considered. It corresponds to scanning the input string only once from left to right

NO AUXILIARY MEMORY: this is the celebrated FINITE STATE AUTOMATON. It is the well known machine model that recognizes the regular languages

AUXILIARY MEMORY: structured and handled as a PUSHDOWN STACK. This is the well known machine model that recognizes the free languages

A CONFIGURATION (instantaneous) is the set of the three components that determine the behaviour of the automaton

- the still unread part of the input tape
- the contents and position of the memory tape and head, respectively
- the current state of the control unit

INITIAL CONFIGURATION: the input head is positioned on the symbol immediately following the start-marker, the control unit is in a specific state (initial state), and the memory tape only contains a special initial symbol. The automaton configuration changes through a series of transitions, each of which is driven by a move. The whole series of transitions is the computation of the automaton

DETERMINISTIC BEHAVIOUR: in every instantaneous configuration, at most one move is possible (or none). Otherwise (i.e., there are two or more moves), the automaton is said to be non-deterministic (or indeterministic)

FINAL CONFIGURATION: the control unit is in a special state qualified as final and the input head is positioned on the end-marker of the string to be recognized. Sometimes the final configuration is characterized by a condition for the memory tape: to be empty or to contain only one special final symbol

A SOURCE STRING x IS ACCEPTED BY THE AUTOMATON if it starts from the initial input configuration $\vdash x \dashv$, executes a series of transitions (moves) and reaches a final configuration. If the automaton is non-deterministic, it may reach the same final configuration in two or more different sequences of transitions, or it may even reach two or more different final configurations

THE COMPUTATION terminates either because the automaton has reached a final configuration or because it cannot execute any more transition steps, due to the fact that in the current instantaneous configuration there is not any possible move left.
In the former case the input string is accepted, in the latter case it is rejected

THE SET OF ALL THE STRINGS ACCEPTED BY THE AUTOMATON constitutes the language RECOGNIZED (or ACCEPTED or DEFINED) BY THE AUTOMATON

Two automata that accept the same language are said to be EQUIVALENT.
CAUTION: two equivalent automata may be modeled differently and may not have the same computational complexity

REGULAR LANGUAGES, which are recognized by finite state automata, are a sub-family of the languages recognizable in real time by a TURING MACHINE

FREE LANGUAGES are a subfamily of the languages recognized by a TURING MACHINE that have a polynomial time complexity

$\boxed{\text{FINITE STATE AUTOMATA (or simply FINITE AUTOMATA)}}$

Many applications of computer science and engineering make use of finite state automata: digital design, theory of control, communication protocols, the study of system realiability and security, etc

IN THE FOLLOWING

- state-transition graphs of finite automata
- finite automata as recognizers of regular languages
- deterministic and non-deterministic behaviour
- how to transform a non-deterministic automaton into a deterministic one
- language sub-families characterized by a local recognition test

STATE-TRANSITION GRAPH

A finite state automaton consists of the following three elements

the input tape, which contains the input string $x \in \Sigma^*$

the control unit and its finite memory, which contains the state table

the input head, initially positioned at the start-marker of string *x*, which is shifted to the right at every move, as far as it reaches the end-marker of string *x* or an error happens

After reading an input character, the automaton updates the current state of the control unit

After scanning the input string *x*, the automaton recognizes string *x* or rejects it, depending on the current state

STATE-TRANSITION GRAPH (continued)

STATE-TRANSITION GRAPH: is a directed graph that represents the automaton and consists of the following elements

NODES: represent the states of the control unit

ARCS: represent the moves of the automaton

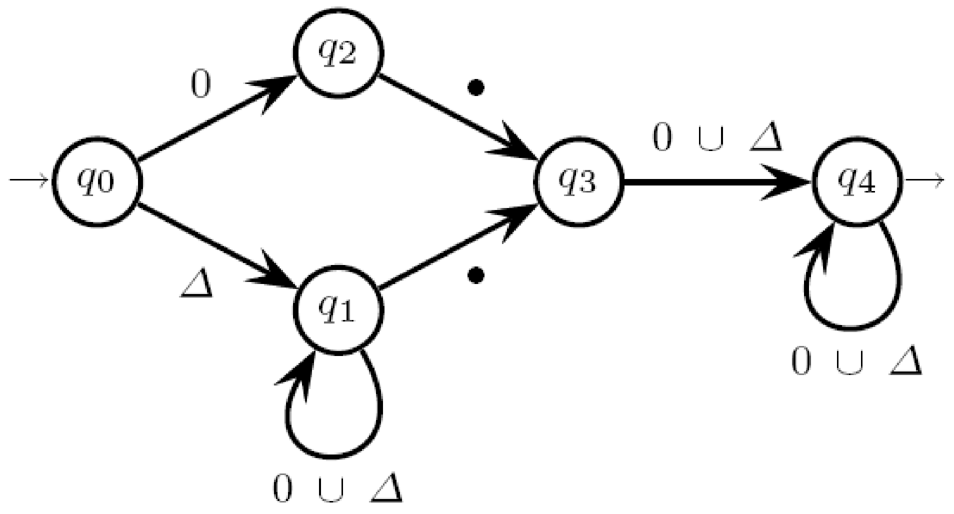
Each arc is labeled by an input symbol, and it represents the move enabled when the current state matches the source state of the arc and the current input symbol matches the arc label

The state-transition graph has a unique INITIAL STATE (if it had two or more, then it would be non-deterministic), but it may have none, one or more FINAL STATES (if it does not have any final states, then it recognizes the empty set)

The graph can be represented in the form of an INCIDENCE MATRIX. Each matrix entry is indexed by the current state and by the input symbol, and contains the next state. Such an incidence matrix is often called STATE TABLE

EXAMPLE – decimal constants in numerical form

state-transition diagram



state-transition table

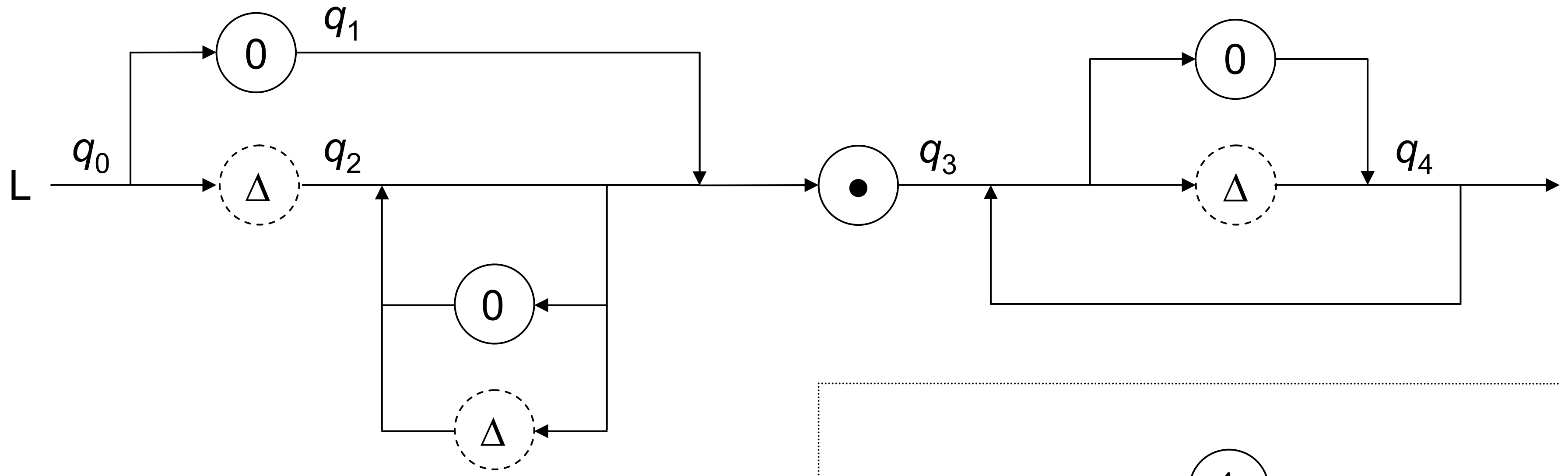
current	*current character*				
state	0	1	...	9	•
$\rightarrow q_0$	q_2	q_1	...	q_1	–
q_1	q_1	q_1	...	q_1	q_3
q_2	–	–	...	–	q_3
q_3	q_4	q_4	...	q_4	–
$q_4 \rightarrow$	q_4	q_4	...	q_4	–

$$\Sigma = \Delta \cup \{ 0, \bullet \} \qquad \Delta = \{ 1, 2, 3, 4, 5, 6, 7, 8, 9 \}$$

$$e = \left(0 \cup \Delta \left(0 \cup \Delta \right)^* \right) \bullet \left(0 \cup \Delta \right)^+$$

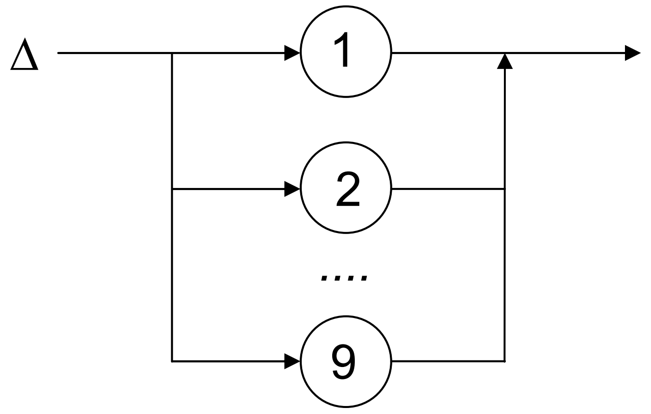
NOTE: in the drawing, a few topologically identical arcs are grouped into one (in fact from q_0 to q_1 nine arcs should be drawn)

Here follow the SYNTAX DIAGRAMS of the automaton, similar to those that can be used to represent the *EBNF* rules. They are the DUAL FORM of the state-transition graph of the automaton



syntax diagram

DUAL FORM: transform nodes into vertices and vertices into nodes



syntax diagram

DETERMINISTIC FINITE STATE AUTOMATON

A DETERMINISTIC FINITE AUTOMATON M consists of five elements

Q	the *set of states* (finite and not empty)
Σ	the *input alphabet* (or terminal alphabet)
$\delta: (Q \times \Sigma) \rightarrow Q$	the *transition function*
$q_0 \in Q$	the *initial state*
$F \subseteq Q$	the *set of final states* (it may be empty)

The transition function encodes the automaton moves: $\boxed{\delta(q_i, a) = q_j}$

$$\boxed{\begin{array}{l} x = ab \quad \delta(q_0, a) = q_1 \quad \delta(q_1, b) = q_2 \\ \delta(q_0, ab) = q_2 \\ \\ \forall q \in Q : \quad \delta(q, \varepsilon) = q \end{array}}$$

When M is in the current state q_i and reads the input symbol *a*, it switches the current state to q_j

If δ (q_i, a) is undefined, then M stops (it enters an error state and rejects the input string) the domain is $Q \times \Sigma^*$ and the transition function is

$$\boxed{\delta(q, y\,a) = \delta(\delta(q, y), a) \qquad \text{where } a \in \Sigma \text{ and } y \in \Sigma^*}$$

EXECUTION OF A COMPUTATION (= sequence of TRANSITIONS)

The transition function δ is extended by posing as aside if, and only if, there exists a path from state *q* to state *q*', labeled by the input string *y*. If the automaton moves through such a path, it recognizes the string *y*. The automaton executes a sequence of transitions, i.e., a computation

$$\boxed{\delta(q, y) = q'}$$

RECOGNITION OF A STRING
A string *x* is recognized (or accepted) if, and only if, when the automaton moves through a path labeled by *x*, it starts from the initial state and ends at one of the final states

$$\boxed{\delta(q_0, x) \in F}$$

The empty string is accepted if, and only if, the initial state is final as well

THE LANGUAGE RECOGNIZED BY THE AUTOMATON M

$$\boxed{L(M) = \{\ x \in \Sigma^* \mid \quad x \text{ is recognized by } M\ \}}$$

The family of the languages recognized by finite stata automata is named family of *finite state languages*

Two finite automata are equivalent if they recognize the same language

The time complexity of finite state automata is optimal: the input string *x* is accepted or rejected in real-time, i.e., in a number of moves equal to the string length. Since it takes exactly as many steps to scan the string from left to right, the recognition time complexity could not be lower than this

EXAMPLE – decimal constants in numercal form (continued)

The automaton M is defined as follows

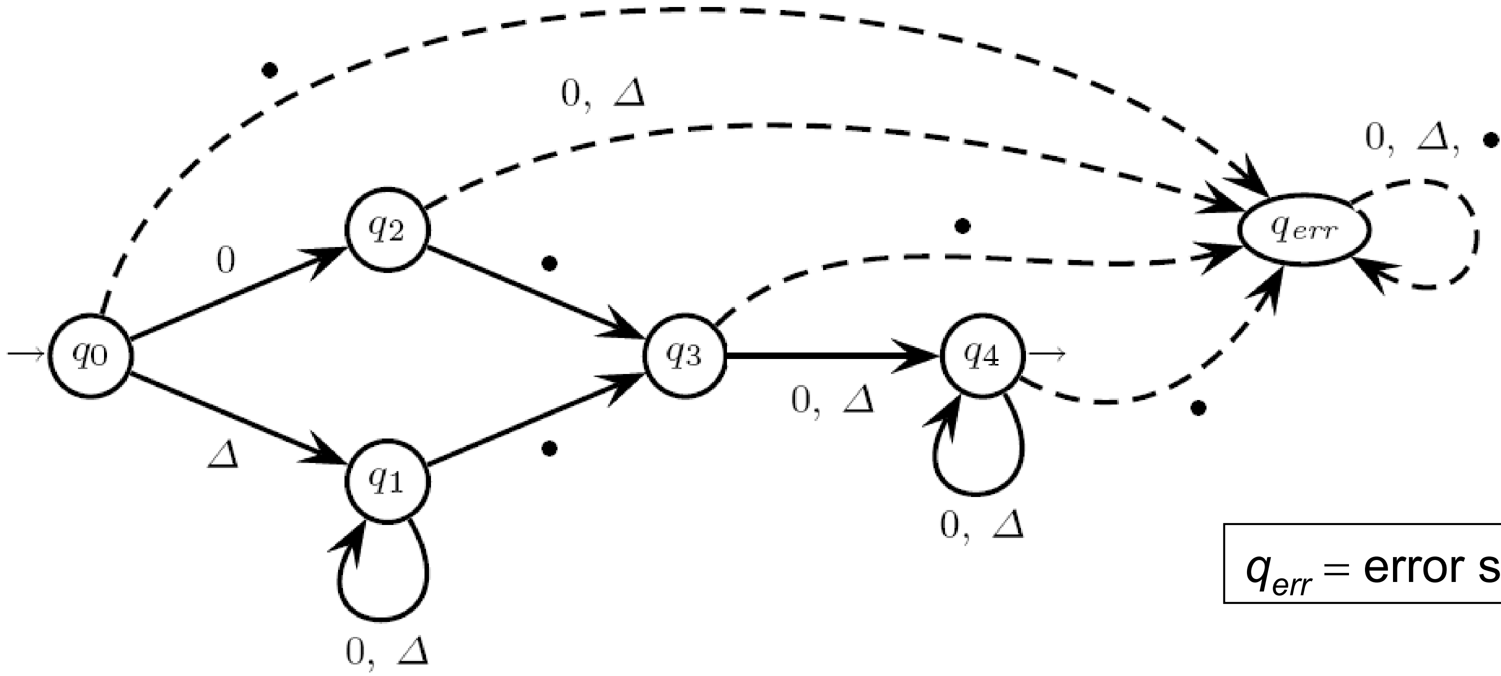
$Q = \{ q_0, q_1, q_2, q_3, q_4 \}$	state set
$\Sigma = \{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, \bullet \}$	alphabet
$q_0 = q_0$	initial state
$F = \{ q_4 \}$	final state set

$$\begin{aligned}
\delta(q_0, 3 \bullet 1) &= \delta(\delta(q_0, 3 \bullet), 1) \\
&= \delta(\delta(\delta(q_0, 3), \bullet), 1) \\
&= \delta(\delta(q_1, \bullet), 1) \\
&= \delta(q_3, 1) = q_4
\end{aligned}$$

acceptance

Since it holds $q_4 \in F$, string $3 \bullet 1$ is accepted. On the contrary, since state $\delta(q_0, 3 \bullet) = q_3$ is not final, string $3 \bullet$ is rejected, as well as string $0\,2$ because function $\delta(q_0, 0\,2) = \delta(\delta(q_0, 0), 2) = \delta(q_2, 2)$ is undefined.

ERROR STATE AND NATURAL COMPLETION OF THE AUTOMATON



q_{err} = error state

$$\forall\ q \in Q\ \forall\ a \in \Sigma \text{ if } \delta(q,\ a) \text{ is undefined then set } \delta(q,\ a) = q_{err}$$

$$\forall\ a \in \Sigma \text{ set } \delta(q_{err},\ a) = q_{err}$$

It is always possible to complete the deterministic automaton by adding the error state, without changing the accepted language

AUTOMATON IN CLEAN FORM (REDUCED FORM)

An automaton may contain useless parts, i.e., states, which do not give any contribution to the recognition process and which usually can be eliminated

A state *q* is said to be ACCESSIBLE (or REACHABLE) FROM STATE *p* if there is a computation that moves the automaton from state *p* to state *q*

A state *q* is said to be ACCESSIBLE (or REACHABLE) if it can be reached from the initial state

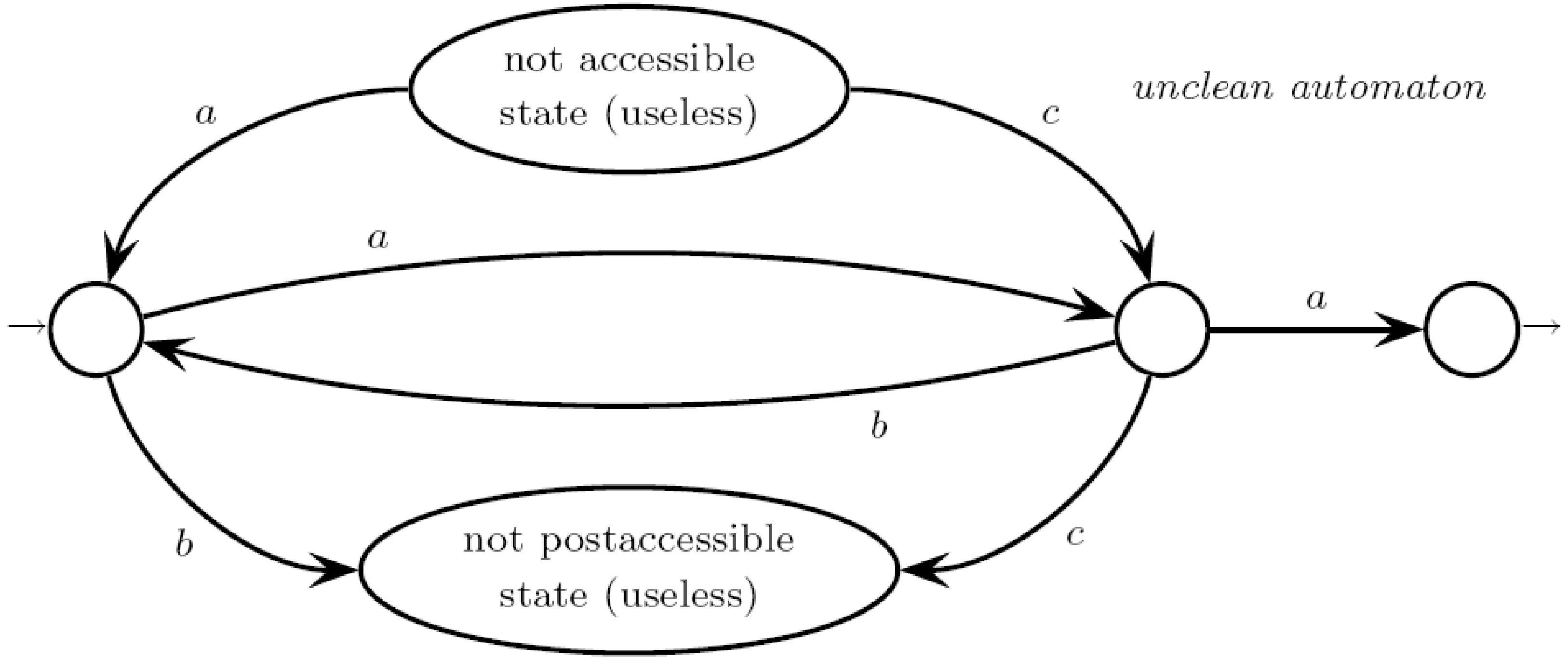
A state *q* said to be POSTACCESSIBLE (or DEFINED) if some final state can be reached from state *q*

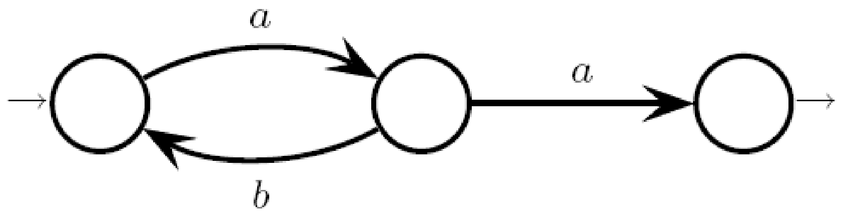
A state *q* is said to be USEFUL if it is both accessible and postaccessible, i.e., if it is placed on a path that connects the initial state to a final state

An automaton A is said to be in CLEAN FORM (or in REDUCED FORM) if every state of A is useful, i.e., both accessible and postaccessible

PROPERTY – Every finite state automaton has an equivalent clean form.
To reduce an automaton: first identify all the useless states, then strip them off the automaton along with all their incoming and outgoing arcs

EXAMPLE – elimination of the useless states


not accessible state (useless)
unclean automaton
a
c
a
a
b
b
c
not postaccessible state (useless)



clean automaton

CAUTION: it may not be minimal yet !

MINIMAL AUTOMATON

PROPERTY – For every finite state language there exists one, and only one, deterministic finite state recognizer that has the smallest possible number of states, which is called the *minimal automaton*

UNDISTINGUISHABLE STATE – A state *p* is UNDISTINGUISHABLE FROM A STATE *q* if, and only if, for every input string *x* either both the next states δ (*p* ,x) and δ (*q*, *x*) are final, or neither one is; equivalently, if one starts from *p* and *q*, scans the string *x*, but does not reach a final state in either case

Two undistinguishable states can be MERGED and thus the number of states of the automaton can be reduced, without changing the recognized language

UNDISTINGUISHABILITY is a binary relation, and is reflexive, symmetric and transitive, therefore it is an *equivalence* relation

p is DISTINGUISHABLE from *q* if and only if (1) or (2) hold	1) *p* is final and *q* is not (or viceversa) 2) δ (*p*, *a*) is distinguishable from δ (*q*, *a*)

EXAMPLE

deterministic automaton

$$\begin{tikzcd}
{} \arrow[r] & q_0 \arrow[loop above, "b"] \arrow[r, "a"] & q_1 \arrow[loop below, "a"] \arrow[r, "b"] & q_2 \arrow[loop above, "b"] \arrow[d] \arrow[r, bend left, "a"] & q_3 \arrow[loop below, "a"] \arrow[l, bend left, "b"] \arrow[r] & {} \\
& & & {} & &
\end{tikzcd}$$

Undistinguishability table

q_1	$(1,1)$ $(0,2)$		
q_2	$\times$	$\times$	
q_3	$\times$	$\times$	$(3,3)$ $(2,2)$
	q_0	q_1	q_2

undistinguishable state pair: [q_2, q_3]

q_1	$\times$		
q_2	$\times$	$\times$	
q_3	$\times$	$\times$	$(3,3)$ $(2,2)$
	q_0	q_1	q_2

distinguishable state pair: [q_0, q_1]

Equivalence classes of the undistinguishability relation: [q_0], [q_1] and [q_2, q_3]

REDUCTION OF THE NUMBER OF STATES (MINIMIZATION)

The equivalence classes of the undistinguishability relation of the original automaton M are those of the minimal automaton M’ equivalent to M.

To define the transition function of M’, it suffices to state that there is an arc from class $C_1 = [\dots, p_r, \dots]$ to class $C_2 = [\dots, q_s, \dots]$ if and only if in M there is an arc from state p_r to state q_s (with the same label)

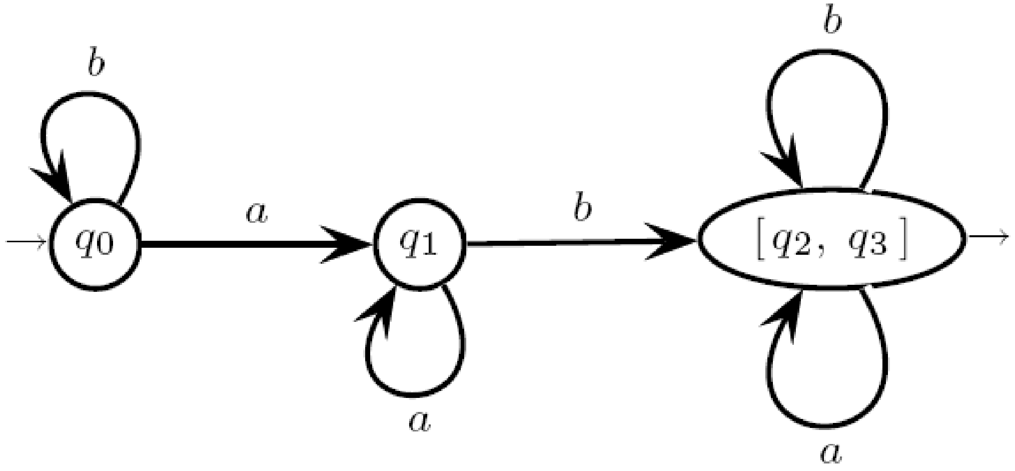
$$\boxed{p_r \xrightarrow{b} q_s}$$

$$\boxed{\overbrace{[\ \dots\ p_r\ \dots\ ]}^{C_1} \xrightarrow{b} \overbrace{[\ \dots\ q_s\ \dots\ ]}^{C_2}}$$

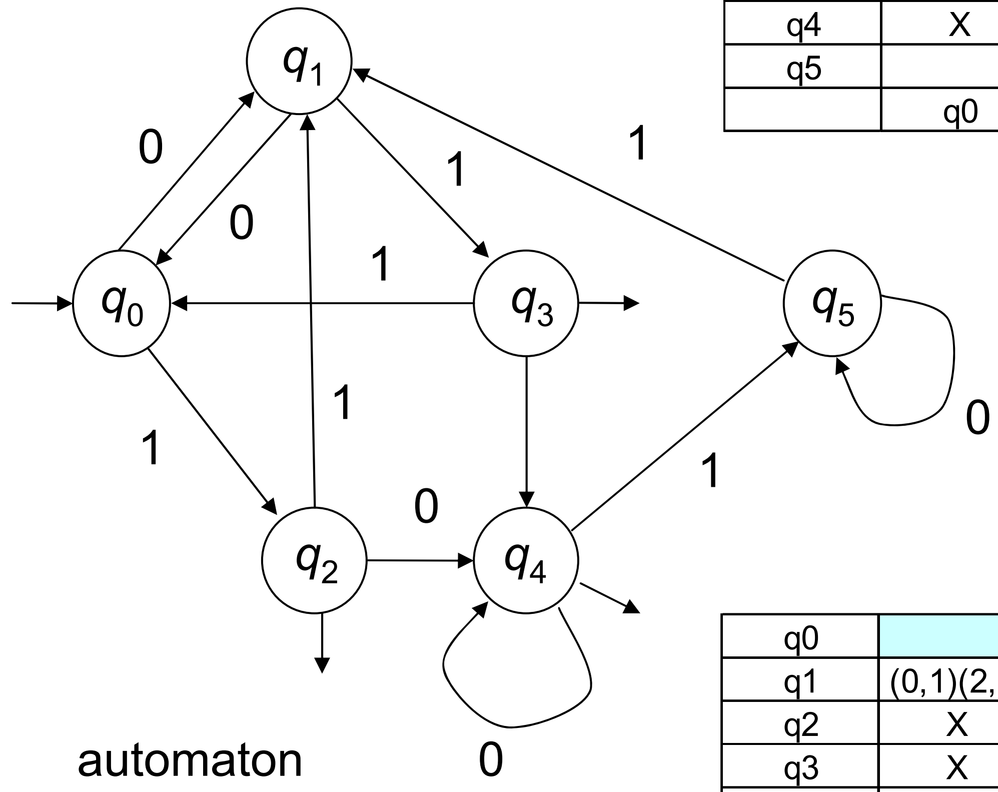
That is, there is an arc between two states belonging to the two classes.
CAUTION: the same arc of M’ may originate from two or more arcs of M

EXAMPLE (continued)

minimal automaton



EXAMPLE (with six states)



automaton

q0						
q1						
q2	X	X				
q3	X	X				
q4	X	X				
q5			X	X	X	
	q0	q1	q2	q3	q4	q5

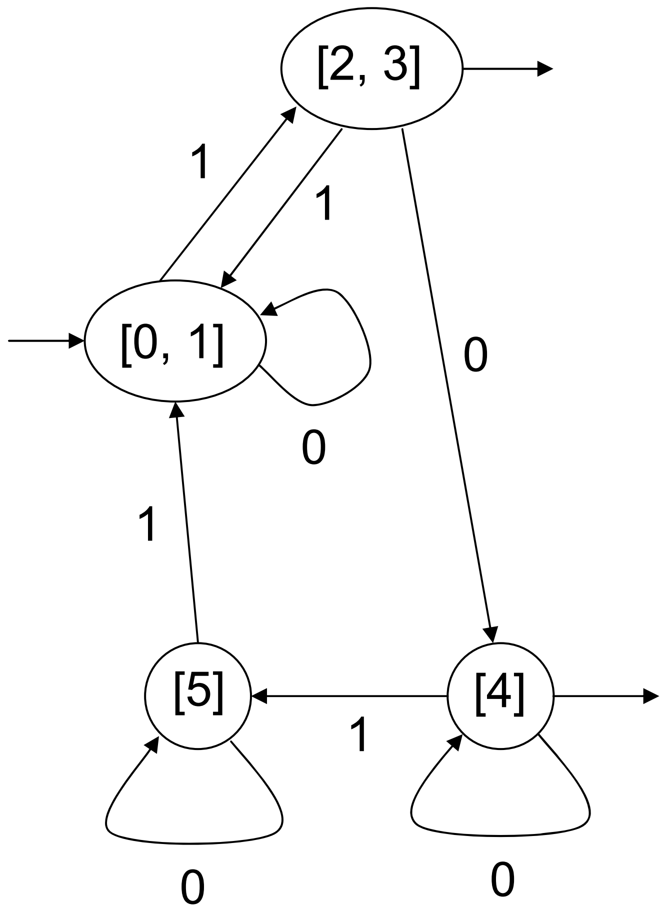
undistinguishability table

undistinguishability table

q0						
q1	(0,1)(2,3)					
q2	X	X				
q3	X	X	(4,4)(0,1)			
q4	X	X	(4,4)(1,5)	(4,4)(0,5)		
q5	(1,5)(2,1)	(0,5)(3,1)	X	X	X	
	q0	q1	q2	q3	q4	q5

EXAMPLE (continued)

minimal automaton



undistinguishability table

q0						
q1						
q2	X	X				
q3	X	X				
q4	X	X	X	X		
q5	X	X	X	X	X	
	q0	q1	q2	q3	q4	q5

equivalence classes of M
$[q_0, q_1]$, $[q_2, q_3]$, $[q_4]$ and $[q_5]$

How does the minimization algorithm behave when the transition function δ is not total ? Suppose to modify the automaton M by removing the transition $\delta(q_3, a) = q_3$. To do so, set $\delta(q_3, a) = q_{err}$. Thus the states q_2 and q_3 are distinguishable, because $\delta(q_2, a) = q_3$ and $\delta(q_3, a) = q_{err}$, and q_3 is distinguishable from q_{err}. Therefore the automaton M is minimal

It is possible to prove that the minimal deterministic automaton is unique. This property does not hold for the minimal non-deterministic automaton, which usually is not unique (but in some special cases)

The uniqueness of the minimal automaton offers a way to check whether two deterministic finite state automata are equivalent. First minimize the number of states of both and obtain the corresponding minimal automata; then check if the two minimal state-transition graphs are topologically indentical (i.e., if nodes and arcs can be exactly overlapped). The two original automata are equivalent if, and only if, the two minimal automata are topologically identical.
CAUTION: there exist different state minimization algorithms as well
CAUTION: it does not work for non-det. automata (non-unique min. form !)

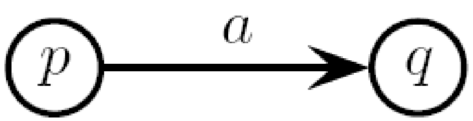
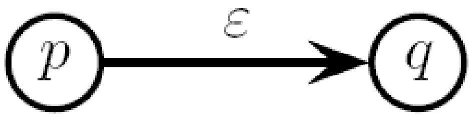
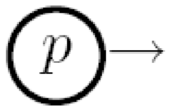
FROM THE AUTOMATON TO THE GRAMMAR

FINITE STATE AUTOMATA AND UNI-LINEAR GRAMMARS (or type 3 Chomsky grammars) deal with the same language family

Procedure to construct a finite automaton corresponding to a right linear grammar.

First it is necessary to modify the definition of finite automaton, in order to include explicitly the notion of indeterminism

The states of the automaton Q are the non-terminal symbols of the grammar G, and the initial state is the axiom

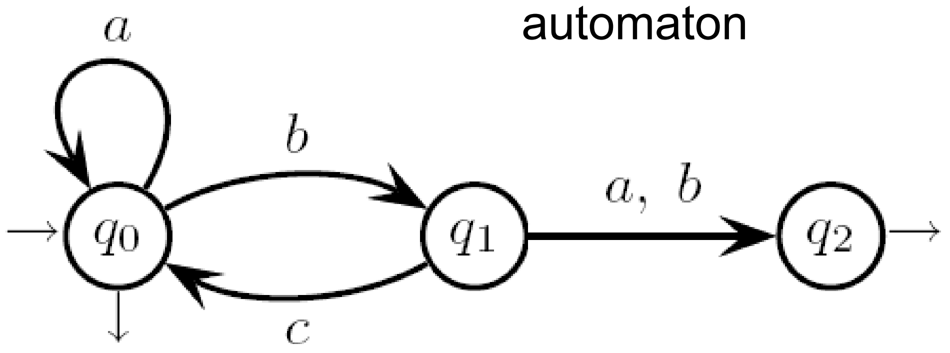
$p \rightarrow a\,q$ where $a \in \Sigma$ and $p,\ q \in V$	
$p \rightarrow q$ where $p,\ q \in V$	
$p \rightarrow \varepsilon$	final state 

There is a one-to-one (or bijective) correspondence between the computations of the automaton and the derivations of the right uni-linear grammar

$$\boxed{q_0 \overset{+}{\Rightarrow} x}$$

EXAMPLE

automaton



right uni-linear grammar

$$\begin{cases} q_0 \rightarrow a\ q_0 \ \mid\ b\ q_1 \ \mid\ \varepsilon \\ q_1 \rightarrow c\ q_0 \ \mid\ a\ q_2 \ \mid\ b\ q_2 \\ q_2 \rightarrow \varepsilon \end{cases}$$

recognition of string *bca*

$$\boxed{q_0 \Rightarrow b\ q_1 \Rightarrow b\ c\ q_0 \Rightarrow b\ c\ a\ q_0 \Rightarrow b\ c\ a\ \varepsilon}$$

<u>Non-nullable form of the grammar</u>

In this version of the uni-linear grammar, an automaton move to a final state is mapped onto two grammar rules

$$q \xrightarrow{a} r \rightarrow \qquad \boxed{q \rightarrow ar \mid a}$$

$$\boxed{\begin{array}{l} Null = \{q_0, q_2\} \\ q_0 \rightarrow aq_0 \mid bq_1 \mid a \mid \varepsilon \\ q_1 \rightarrow cq_0 \mid aq_2 \mid bq_2 \mid a \mid b \mid c \end{array}}$$