Foundations of Computer Science

ECE NTUA

Section 1:

Automata, Languages, Grammars

Slide editing: Stathis Zachos, Aris Pagourtzis

Finite State Machines (FSM)

- A way to describe algorithms.
- Describe Finite State Systems:
  - They model a fundamental (apparent) contradiction of computational (and other) systems: finite system size, unlimited input size.
  - They are defined with internal states and a predefined way to transition from one state to another based on the current state and input. They may also have an output.
- Applications in a wide range of scientific fields.

Example: Coffee machine (i)

Specifications

- Two types of coffee: Greek or Freddo.
- Coffee cost: 40 cents.
- 10, 20, or 50 cent coins are allowed.

Design

- How many states do we need;

Example: Coffee machine (ii)

System Design

- Internal states: q_0, q_1, q_2, q_3, q_4
  - q_i : 10*i cents incerted so far
- Possible inputs (actions): C1, C2, C5, B1, B2
  - C1, C2, C5 : insertion of 10, 20, or 50 cent coins
  - B1, B2 : Button 1 for Greek coffee, or Button 2 for Freddo
- Possible outputs: R1, R2, R3, R4, R5, G, F
  - Ri: refund 10**i cents*
  - G: Greek coffee supply
  - F: Freddo supply

▲

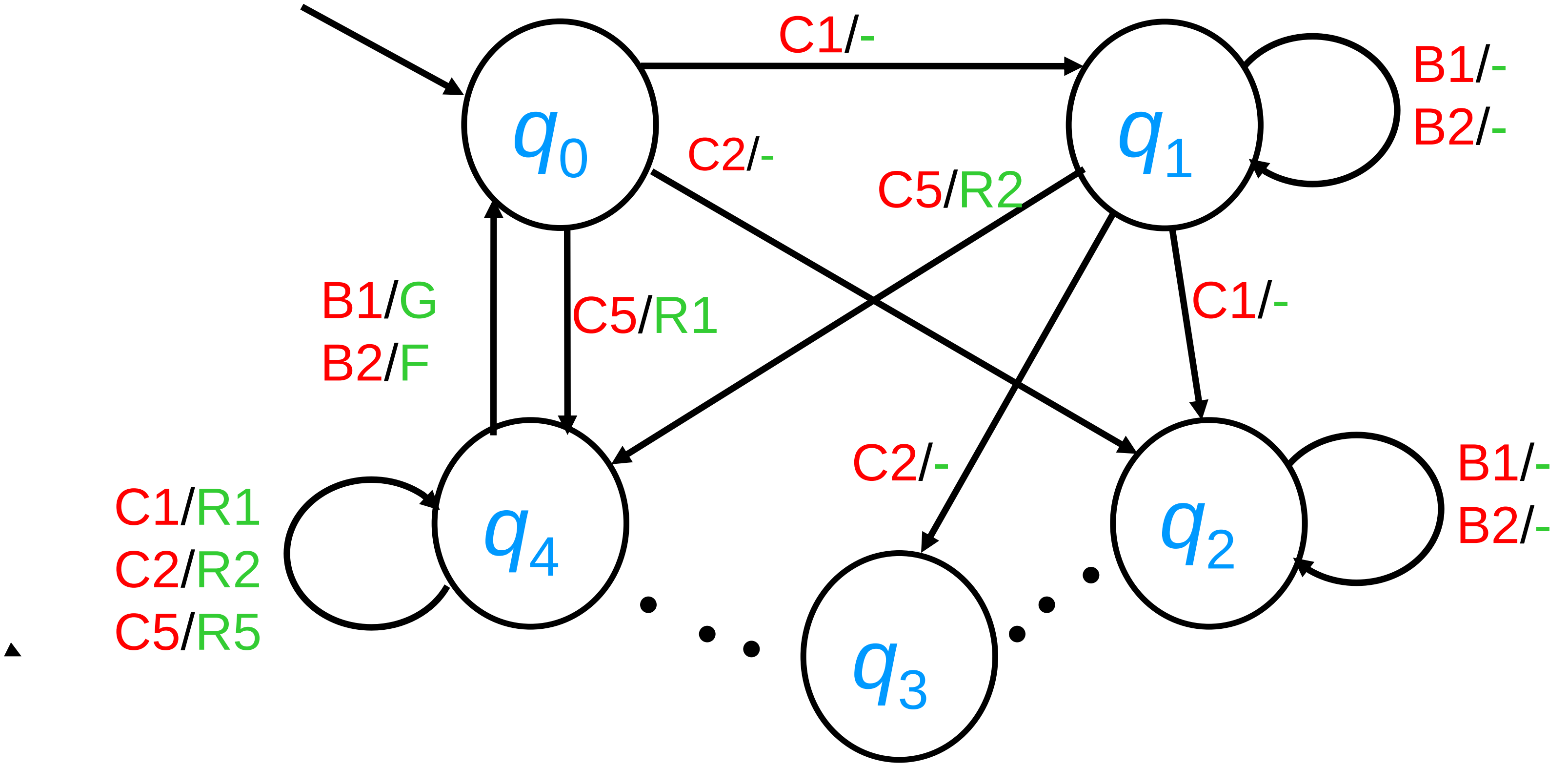
Example: Coffee machine (iii)

- State Table: What the next state and output is for each combination of current state and input. Initial state: q_0.

Input / State	C1	C2	C5	B1	B2
q_0	q_1, -	q_2, -	q_4, R1	q_0, -	q_0, -
q_1	q_2, -	q_3, -	q_4, R2	q_1, -	q_1, -
q_2	q_3, -	q_4, -	q_4, R3	q_2, -	q_2, -
q_3	q_4, -	q_4, R1	q_4, R4	q_3, -	q_3, -
q_4	q_4, R1	q_4, R2	q_4, R5	q_0, G	q_0, F

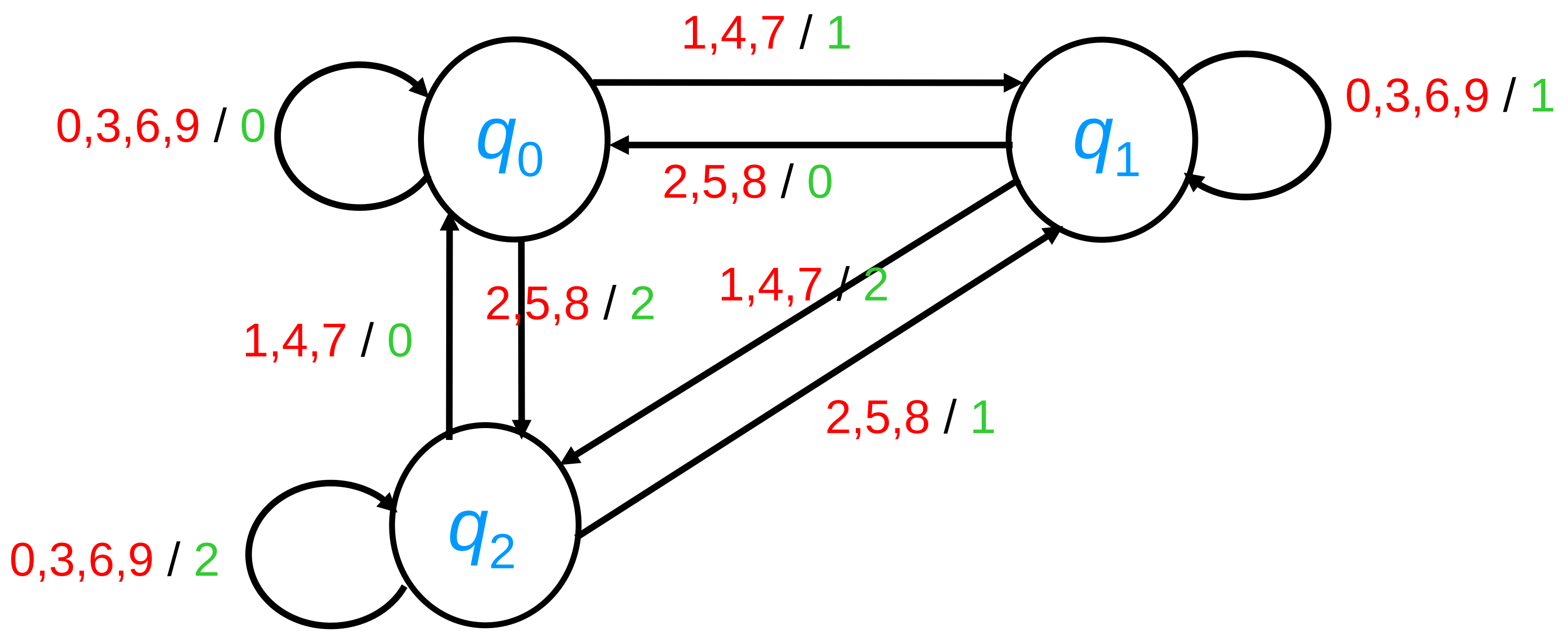
Example: Coffee machine (iv)

- State Diagram: provides the same information as the State Table in a more supervisory way. Initial state: q_0 (marked with an arrow).



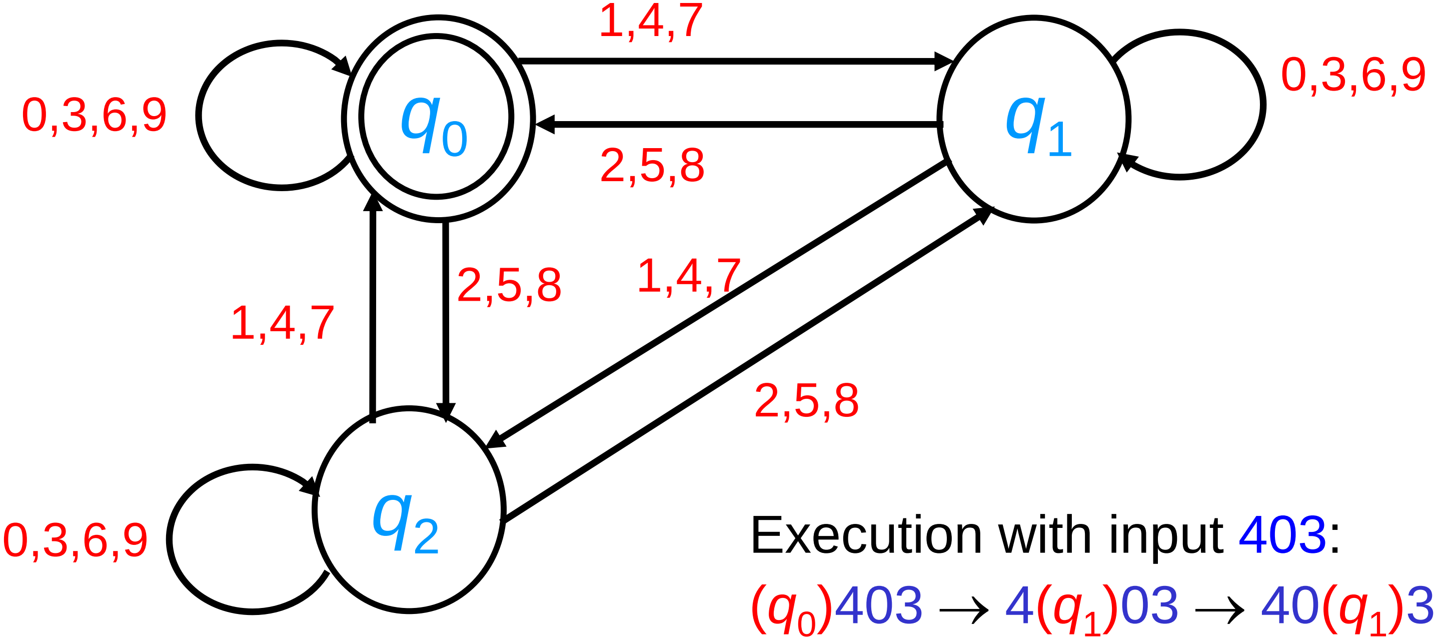
Example II: modulo arithmetic (i)

- Construct a machine that calculates $n \bmod 3$
- How many states are needed?
- Use the property: $n \bmod 3 = (n_1+\ldots+n_k) \bmod 3$, n_i the (base 10) digits of n *Prove it!*



Example II: modulo arithmetic (ii)

- Simplification: If only divisibility by 3 is of interest, no output is needed
- We define acceptance states (double circle)



Execution with input 403:

$(q_0)403 \rightarrow 4(q_1)03 \rightarrow 40(q_1)3 \rightarrow 403(q_1)$

REJECT

Exercises in modulo arithmetic

- Exercise 1: design a machine that determines whether a number is divisible with 5.
- Exercise 2: design a machine that determines whether a number is divisible with 7.

Automata

- Finite State Machines with no output: Some states accept (denoted by an extra circle), while others reject.

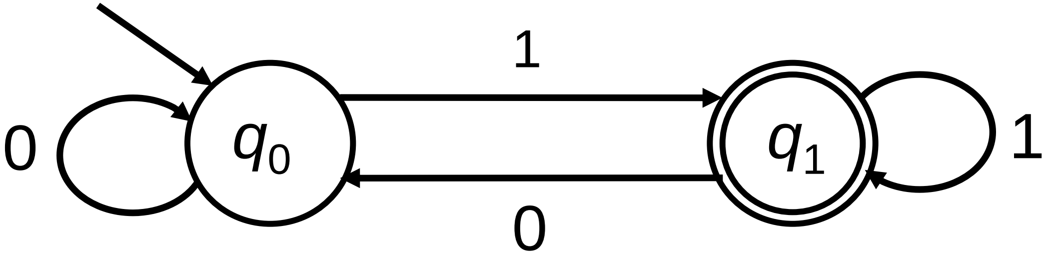
- An automaton has some internal states q_0, q_1, q_7, q_{15}, ... , and a transition function δ that determines the next state of the automaton, based on the current state and the input string. It accepts or rejects the input string.

- Language recognizers (ie. they solve decision problems, properly *described*).

Automata and Formal Languages

- Formal Languages: used to describe computational problems and to define programming languages.
  e.g. $L = \{x \in \{0,1\}^* \mid x$ is a prime number in binary representation$\}$
- Automata: used to identify formal languages and to rank the difficulty of the corresponding problems:
  - Each automaton recognizes a formal language: *the set of strings that lead it to an accept state*.

Example: Odd number identification



- q_0: last digit other than 1
- q_1: last digit equal to 1
- q_0 is called the initial state while q_1 is called the accept (or *final*) state
- Execution with input 0110:
  $(q_0)0110 \rightarrow 0(q_0)110 \rightarrow 01(q_1)10 \rightarrow 011(q_1)0 \rightarrow 0110(q_0)$ REJECT
- Execution with input 101:
  $(q_0)101 \rightarrow 1(q_1)01 \rightarrow 10(q_0)1 \rightarrow 101(q_1)$ ACCEPT

Other automata

- Mechanisms: without input – output: $\delta(q_i) = q_j$
  execution: q_0 -> q_j -> q_k -> q_m . . .
- Pushdown Automata (PDA): much more capabilities as they can use memory (in stack form).
- Turing Machines (TM): even more capabilities as the use unlimited memory (in tape form, with the ability to return).
- Linearly Bounded Automata (LBA): TM with linear bounded memory (the length of the tape is a linear function of the size of the input).

▲

Other formal languages

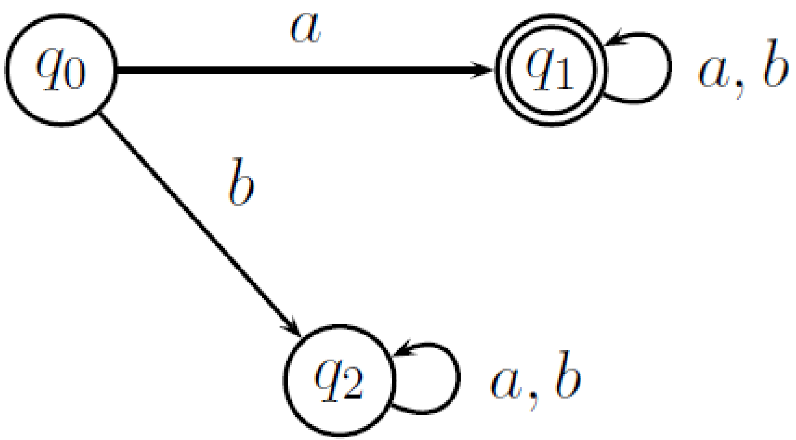
$L_1 = \{w \in \{a,b\}^* \mid w$ starts with “a”$\}$

$L_2 = \{w \in \{a,b\}^* \mid w$ contains an even number of “a”$\}$

$L_3 = \{w \in \{a,b\}^* \mid w$ is a palindrome$\}$

Example: DFA for L_1

$L_1 = \{w \in \{a,b\}^* \mid w \text{ starts with “}a\text{”}\}$



	a	b
q_0	q_1	q_2
q_1	q_1	q_1
q_2	q_2	q_2

Execution with *abba* input:

- $(q_0)abba \rightarrow a(q_1)bba \rightarrow ab(q_1)ba \rightarrow abb(q_1)a \rightarrow abba(q_1)$

ACCEPT

DFA: Formal Definition

- (Deterministic Finite Automaton, DFA): tuple $M = (Q,\Sigma,\delta,q_0,F)$
  - Q : the set of states of M (finite),
    e.g. $Q = \{q_0, q_1, q_2\}$
  - Σ : finite input alphabet ($\Sigma \cap Q = \emptyset$),
    e.g. $\Sigma = \{a,b\}$
  - $\delta : Q \times \Sigma \rightarrow Q$: transition function, e.g. $\delta(q_0,a) = q_1$
  - $q_0 \in Q$: initial (or start) state
  - $F \subseteq Q$: set of accept states,
    e.g. $F = \{q_1\}$

DFA acceptance: formal definitions

- Extension of function *δ*: *Q* x *Σ** → *Q*

  the extended *δ* takes as arguments a state *q* and a string *u* and returns the state where the automaton will reach if it starts from *q* and reads *u*.

- Definitions of extended *δ* (*primitive recursion* scheme):

$$\begin{cases} \delta(q,\varepsilon) = q \\ \delta(q,wa) = \delta(\delta(q,w),a) \end{cases}$$

where *w* is a string of any length, and *a* is an alphabet symbol

DFA acceptance: formal definitions

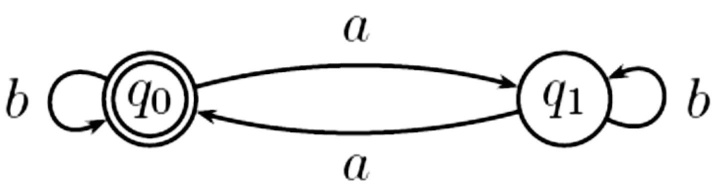
- DFA accepts a string u iff $\delta(q_0,u) \in F$
- DFA M accepts language

$$L(M) = \{w \mid \delta(q_0,w) \in F\}$$

- Languages accepted by a DFA are called

(Kleene) regular

$L_2 = \{w \in \{a,b\}^* \mid w$ contains an even number of "a"$\}$



	a	b
q_0	q_1	q_0
q_1	q_0	q_1

$L_3 = \{w \in \{a,b\}^* \mid w$ is a palindrome$\}$

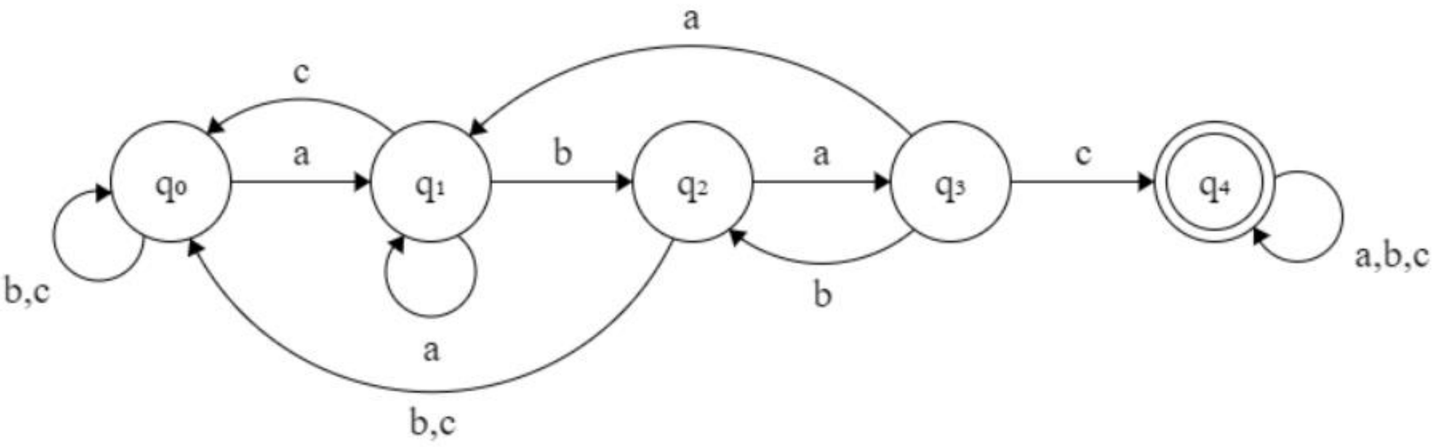
There is no DFA that recognizes L_3

(memory needed with size depending on the input)

Application: String Matching

Problem:

Suppose we are given a text from the alphabet *Σ*={*a*,*b*,*c*}.
How can we check if the string abac is contained in the text?



▲

Non-deterministic automata

- Deterministic automata: for each state / input symbol combination there is a unique next state.
- Non-deterministic automata:
  - For each state/input symbol combinations there is *a number* of possible subsequent states
  - Acceptance if *any* sequence leads to acceptance.

Non-deterministic Finite Automata

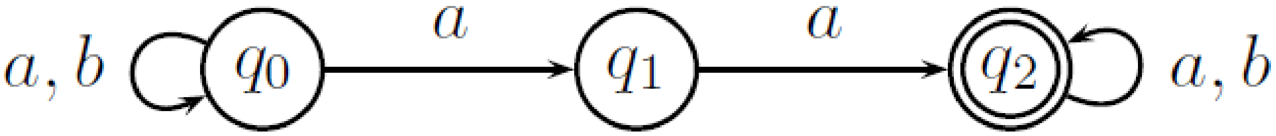
- NFA (Non-deterministic Finite Automaton): for each state and input symbol, one state of a set of possible subsequent states is selected.

- NFAε *or ε-NFA* (NFA with ε-transitions): can change state without reading the next symbol.

▲

Example: NFA

$L_4 = \{w \in \Sigma^* \mid w$ contains two consecutive "a"$\}$

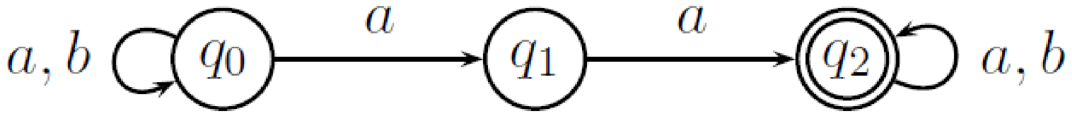


	a	b
q_0	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\{q_2\}$	$\emptyset$
q_2	$\{q_2\}$	$\{q_2\}$

In the transition function, an empty set means that the current execution rejects.

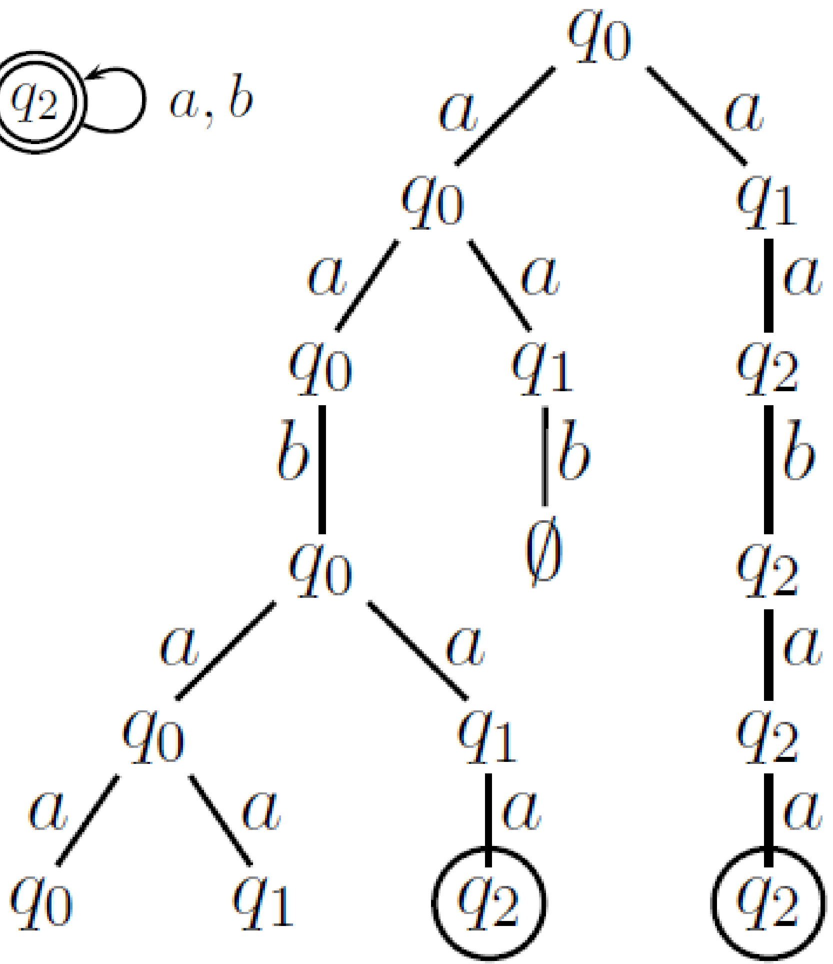
(*another may accept!*).

Example: NFA



Computation tree

for input *aabaa*



ACCEPT

NFA: formal definition

tuple $M = (Q,\Sigma,\delta,q_0,F)$

- Q : the set of states of M (finite)
- Σ : finite input alphabet ($\Sigma \cap Q = \varnothing$)
- $\delta : Q \times \Sigma \rightarrow \mathrm{Pow}(Q)$: transition function,
  e.g. $\delta(q_i, \alpha) = \{ q_j , q_k , q_m \}$
- $q_0 \in Q$: initial state
- $F \subseteq Q$: set of final states (accept)

Reminder: in the transition function, an empty set implies that this execution rejects (*note*: *another one may accept!*).

NFA acceptance: formal definitions

- An NFA accepts string w if $\delta(q_0,w) \cap F \neq \emptyset$
- An NFA accepts the language

$$L(M) = \{w \mid \delta(q_0,w) \cap F \neq \emptyset\}$$

- *Note*: function δ is extended to take as arguments a state q and a string w and return the set of states where the automaton can be found if it starts from q with w *as an input.*
- *Example:* $\delta(q_0, \alpha\alpha) = \{q_0, q_1, q_2\}$, $\delta(q_0, b\alpha) = \{q_0, q_1\}$

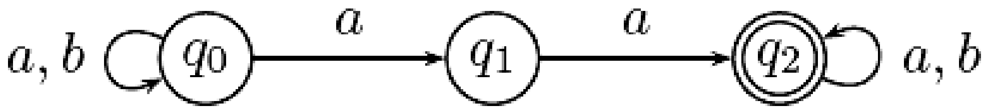
NFA and DFA equivalence

- Rabin-Scott Theorem: for every NFA there exists a DFA that accepts the same language.
- DFA and NFA recognize exactly the same class of languages: the regular languages.
- Regular languages correspond to regular expressions:

e.g. $(a+b)*bbab(a+b)*$

NFA ➔ DFA (i)

NFA for language L_4 ("2 consecutive *a*")



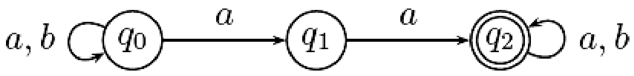
	a	b
q_0	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\{q_2\}$	$\emptyset$
$\textcircled{q_2}$	$\{q_2\}$	$\{q_2\}$

We construct the *powerset* of states.
Initial state: $\{q_0\}$.
Final: those that contain a final.

Hint: We only test state sets that are *accessible from* $\{q_0\}$.

NFA → DFA (ii)

NFA for language L_4



	a	b
q_0	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\{q_2\}$	$\emptyset$
$\textcircled{q_2}$	$\{q_2\}$	$\{q_2\}$

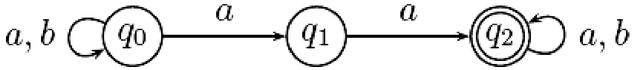
DFA for language L_4

$Q' \backslash \Sigma$	a	b
$\surd$ $\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$

Hint: We only test state sets that are *accessible from* $\{q_0\}$.

NFA ➔ DFA (iii)

NFA for language L_4



	a	b
q_0	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\{q_2\}$	$\emptyset$
q_2	$\{q_2\}$	$\{q_2\}$

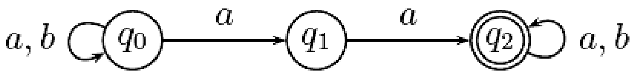
DFA for language L_4

$Q' \backslash \Sigma$	a	b
√ $\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
√ $\{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_0\}$

Hint: We only test state sets that are *accessible from* $\{q_0\}$.

NFA → DFA (iv)

NFA for language L_4



	a	b
q_0	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\{q_2\}$	$\emptyset$
q_2	$\{q_2\}$	$\{q_2\}$

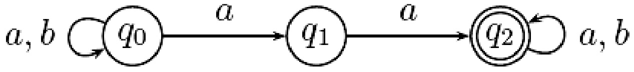
DFA for language L_4

$Q' \backslash \Sigma$	a	b
√ $\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
√ $\{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_0\}$
√ $\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_2\}$

Hint: We only test state sets that are *accessible from* $\{q_0\}$.

NFA → DFA (v)

NFA for language L_4



	a	b
q_0	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\{q_2\}$	$\emptyset$
q_2	$\{q_2\}$	$\{q_2\}$

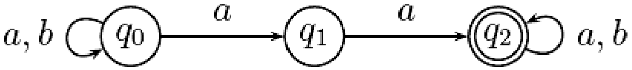
DFA for language L_4

$Q' \backslash \Sigma$	a	b
√ $\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
√ $\{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_0\}$
√ $\{q_0, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_2\}$
√ $\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_2\}$

Hint: We only test state sets that are *accessible from* $\{q_0\}$.

NFA ➔ DFA (vi)

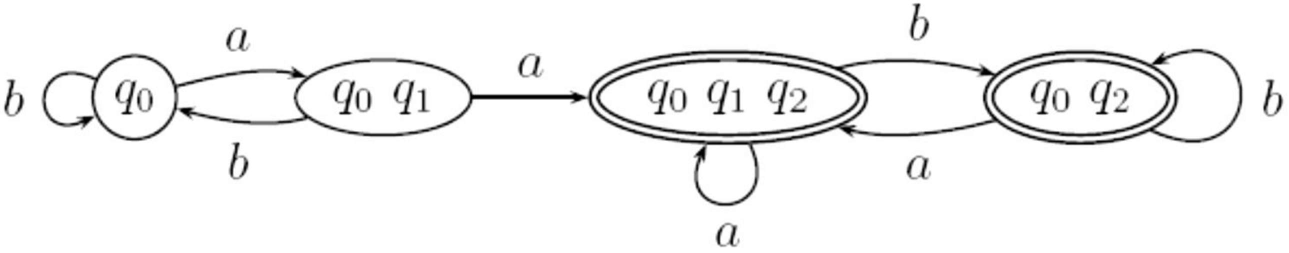
NFA for language L_4



	a	b
q_0	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\{q_2\}$	$\emptyset$
q_2	$\{q_2\}$	$\{q_2\}$

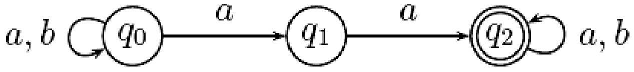
DFA for language L_4

	$Q'\backslash\Sigma$	a	b
√	$\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
√	$\{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_0\}$
√	$\{q_0, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_2\}$
√	$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_2\}$



NFA ➔ DFA (vii)

NFA for language L_4

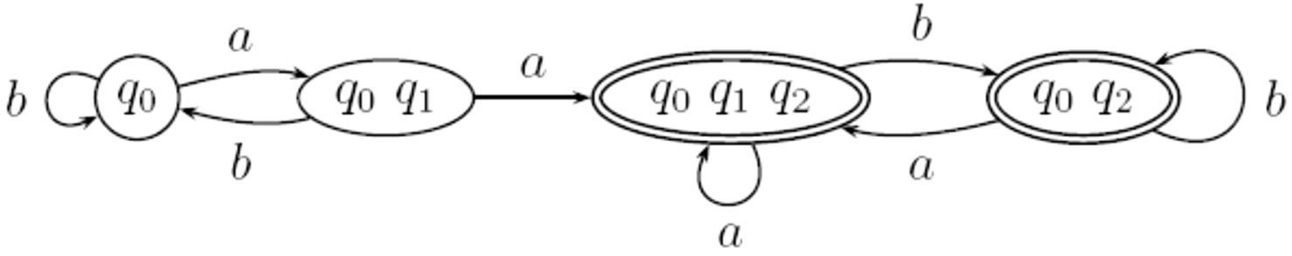


	a	b
q_0	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\{q_2\}$	$\emptyset$
q_2	$\{q_2\}$	$\{q_2\}$

DFA for language L_4

$Q' \backslash \Sigma$	a	b
~~$\emptyset$~~	~~$\emptyset$~~	~~$\emptyset$~~
√ $\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
~~$\{q_1\}$~~	~~$\{q_2\}$~~	~~$\emptyset$~~
~~$\{q_2\}$~~	~~$\{q_2\}$~~	~~$\{q_2\}$~~
√ $\{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_0\}$
√ $\{q_0, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_2\}$
~~$\{q_1, q_2\}$~~	~~$\{q_2\}$~~	~~$\{q_2\}$~~
√ $\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_2\}$

Inaccessible states are of no interest!



NFA ➔ DFA: the method, formally

Suppose NFA $M = (Q,\Sigma,q_0,F,\delta)$.

An equivalent DFA $M' = (Q',\Sigma,q'_0,F',\delta')$, is defined as follows:

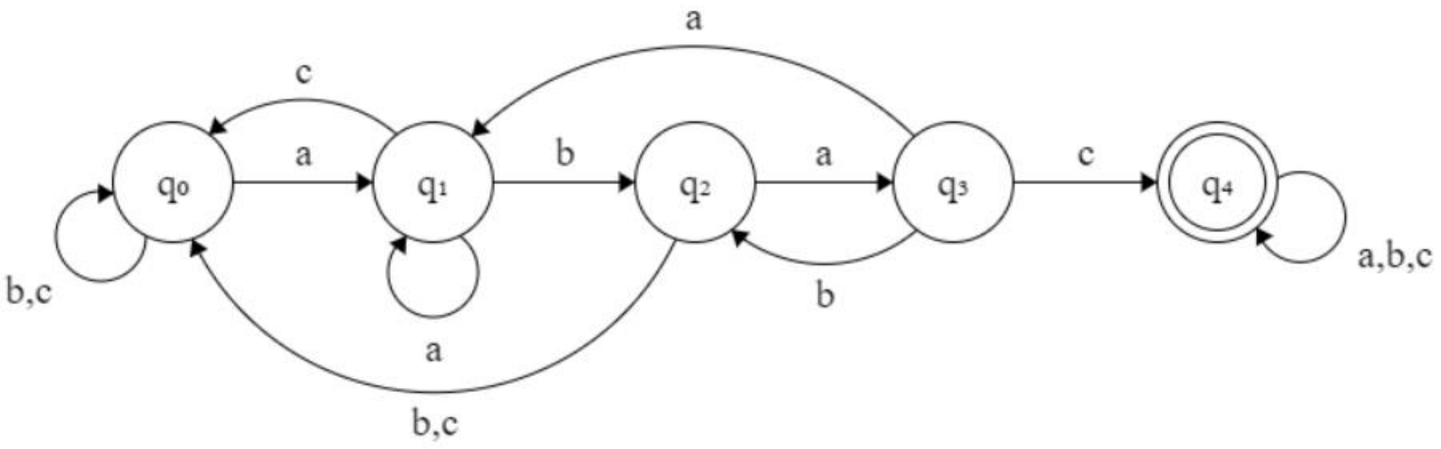
- $Q' = \mathrm{Pow}(Q)$, i.e. The states of *M'* are all subsets of states of *M*.
- $q'_0 = \{q_0\}$,
- $F' = \{R \in Q' \mid R \cap F \neq \emptyset\}$, i.e. a state of *M'* is final if it contains a final state of *M*.
- $\delta'(R, \alpha) = \{q \in Q \mid q \in \delta(r, \alpha) \text{ for } r \in R\}$, i.e. it is the set of possible *M* states starting from any state of the set *R* and reading the symbol *α* (*α*-transition).

Application: String Matching

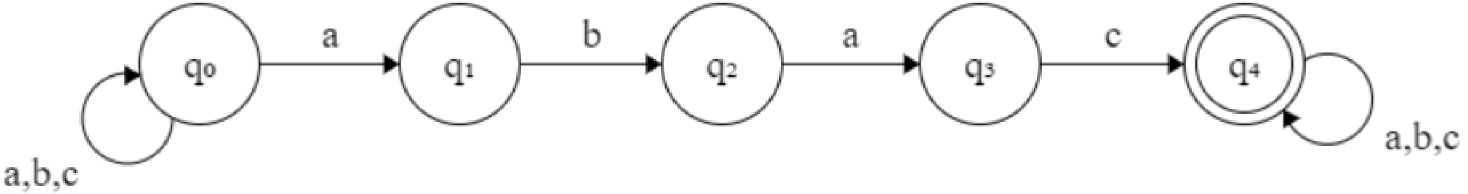
Problem:

Suppose we are given a text from the alphabet $\Sigma=\{a,b,c\}$.
How can we check if the string abac is contained in the text?

DFA



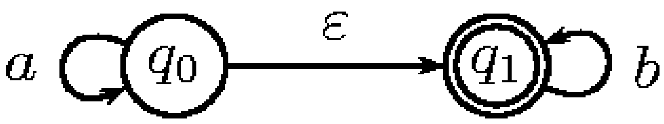
NFA



Automata with ε-transitions: NFA$_\varepsilon$

- They allow transitions without reading a symbol. (equivalent: with input the empty string ε).
- They accept strings that can reach a final state using (potentially) ε-transitions.

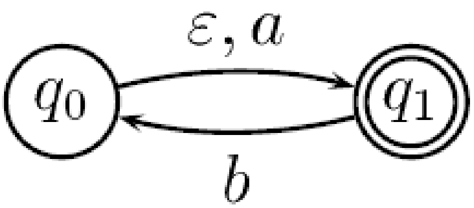
Example: NFA$_\varepsilon$ for $L_5 := \{a^*b^*\} = \{a^n b^m | n, m \in \mathbb{N}\}$



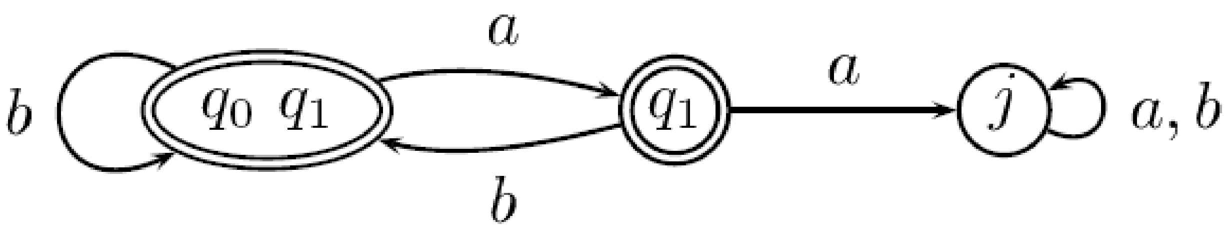
	a	b	ε
q_0	$\{q_0\}$	$\emptyset$	$\{q_0, q_1\}$
q_1	$\emptyset$	$\{q_1\}$	$\{q_1\}$

NFA$_\varepsilon$ and DFA equivalence: example

NFA$_\varepsilon$ for $\overline{L_4}$ (not two consecutive “a”):



DFA for $\overline{L_4}$:



NFA_ε ➔ DFA: the method, formally

Let NFA_ε $M = (Q,\Sigma,q_0,F,\delta)$.

An equivalent DFA $M' = (Q',\Sigma,q'_0,F',\delta')$, is defined as follows:

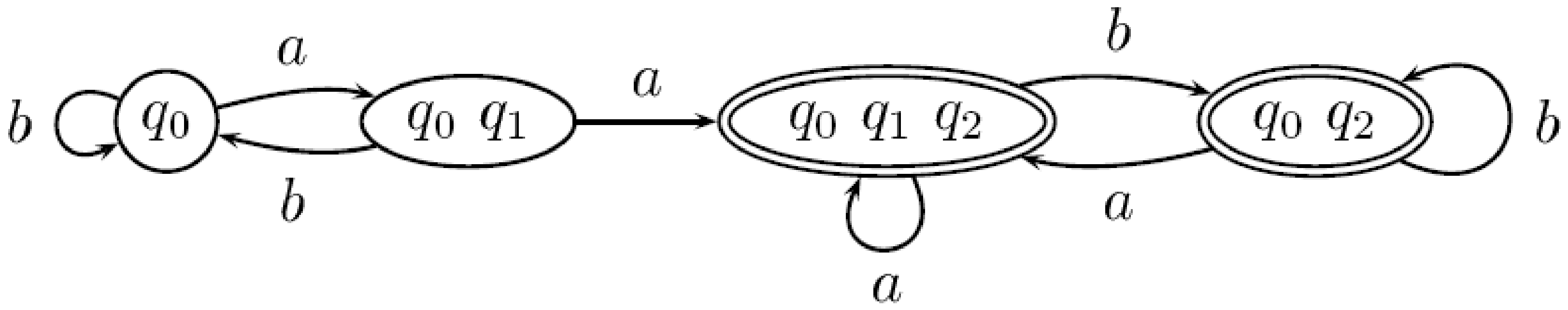
- $Q' = Pow(Q)$, i.e. the states of M' are all subsets of states of M.
- $q'_0 = \varepsilon\text{-closure}(q_0) = \{p \mid p$ accessible from q_0 only with ε-transitions$\}$,
- $F' = \{R \in Q' \mid R \cap F \neq \varnothing\}$, i.e. a state of M' is final if it contains a final state of M.
- $\delta'(R,a) = \{q \in Q \mid q \in \varepsilon\text{-closure}(\delta(r,\alpha))$ for $r \in R\}$, i.e. $\delta'(R,\alpha)$ is the set of states that *M can reach* starting from any state of R , making an α-transition and then using any ε-transitions.

▲

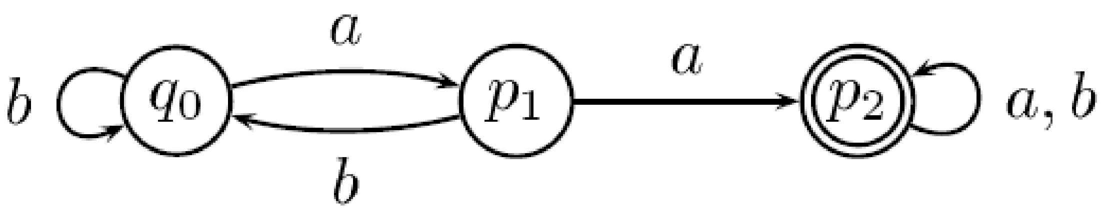
DFA minimization: example

$L_4 = \{ w \in \{a,b\}^* \mid w$ contains 2 consecutive "a" $\}$:

Initial DFA

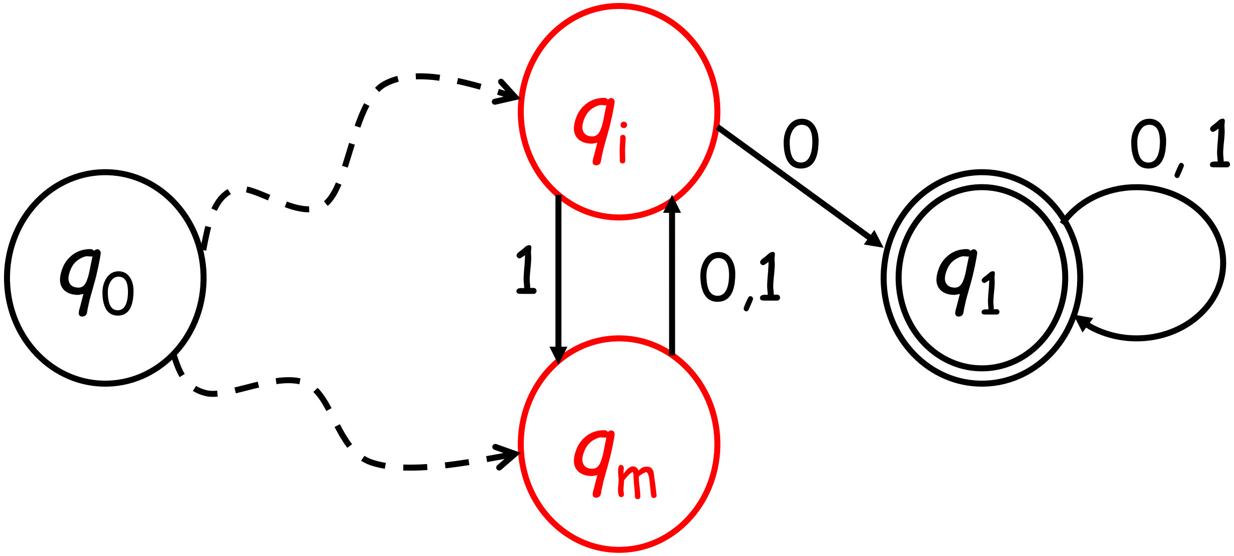


Minimal DFA



DFA minimization (i)

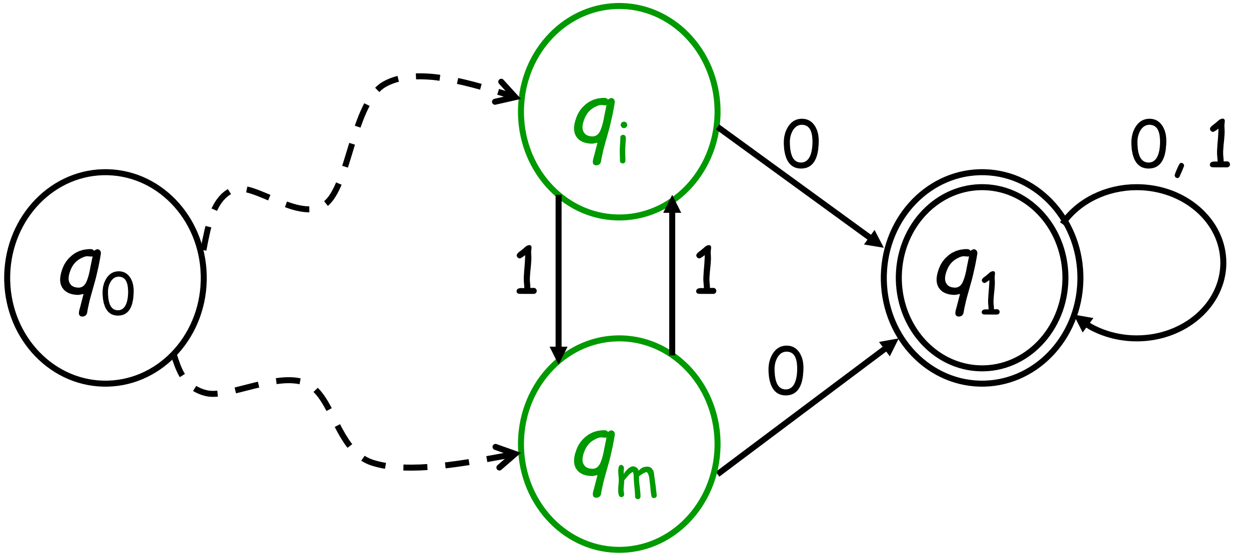
Two DFA states are said to be *non-equivalent*, that is, *distinguishable*, if there is a string that leads one of them to a final state and not the other.



DFA minimization (ii)

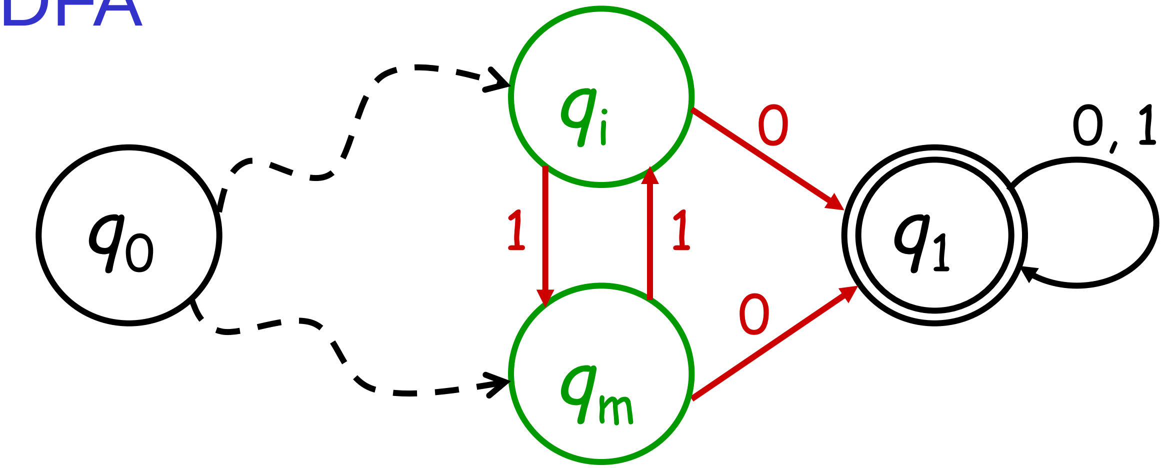
Two states can be merged into one (they are *equivalent*) if:

They lead to the same result with the same strings

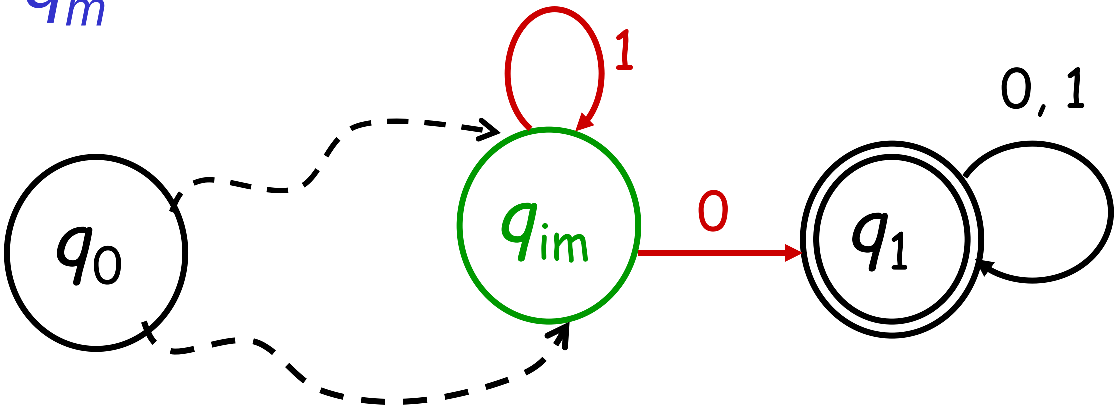


DFA minimization (iii)

Initial DFA

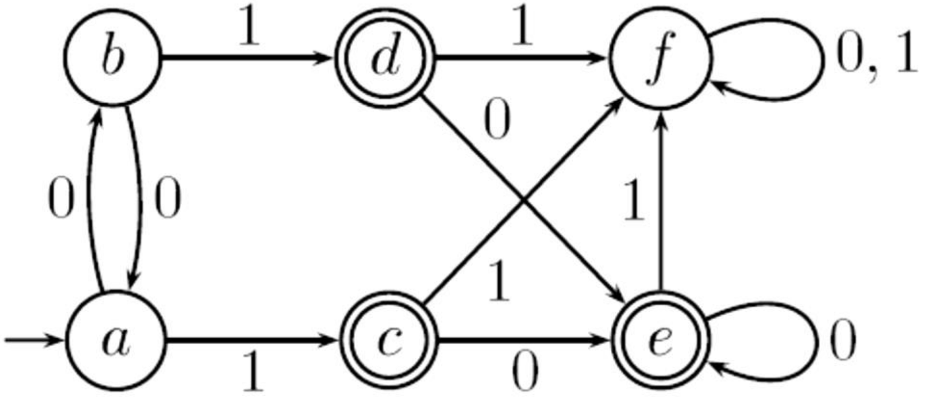


Merging q_i, q_m

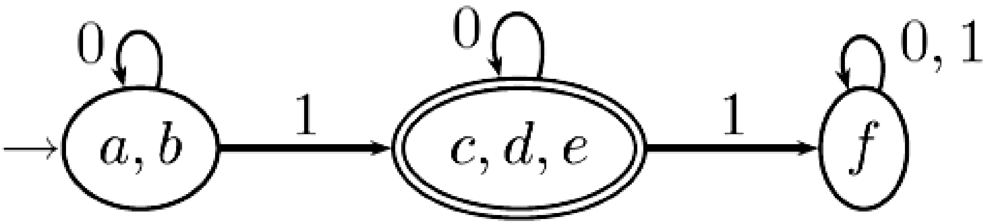


DFA minimization : 2nd example

Initial DFA

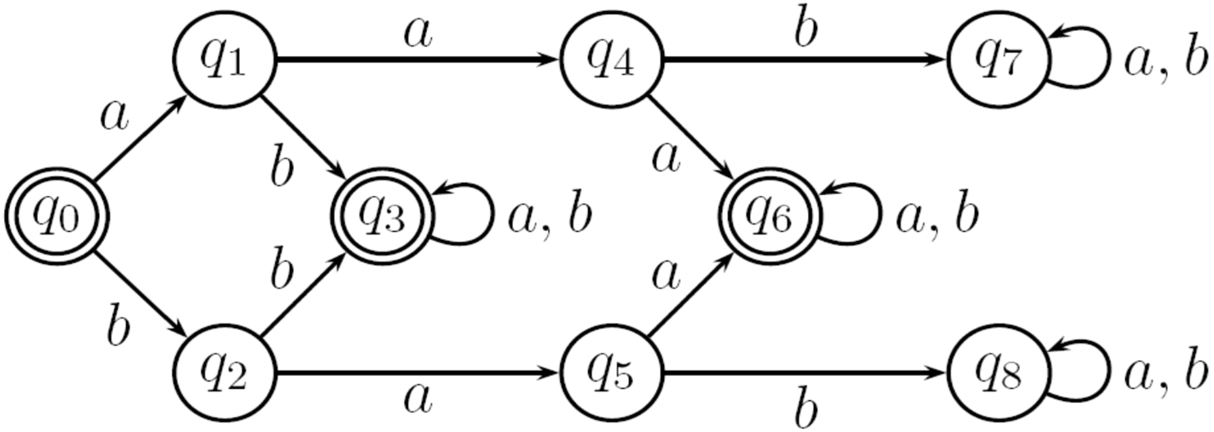


Minimal DFA

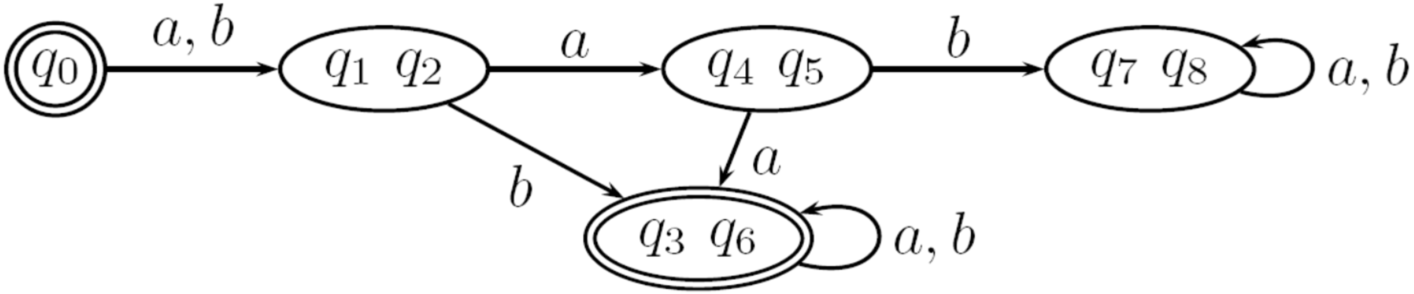


DFA minimization: 3rd example

DFA



Minimal DFA



DFA minimisation method

- Two states are ***k*-distinguishable** if with *a string* of length exactly *k* they lead to a different result (and they are not *i*-distinguishable for any $i<k$). Thus, two states are:
  - 0-distinguishable iff one is final and the other is not
  - (*i*+1)-distinguishable iff with *a symbol they* lead to *i*-distinguishable states.
- Two states are **equivalent** if they are not *k*-distinguishable for *any k*.

▲

DFA minimisation method

- Idea: for all *i* = 0, 1, 2, ... We identify the *i-distinguishable* pair of states until no more occur. The rest of the pairs are equivalent.

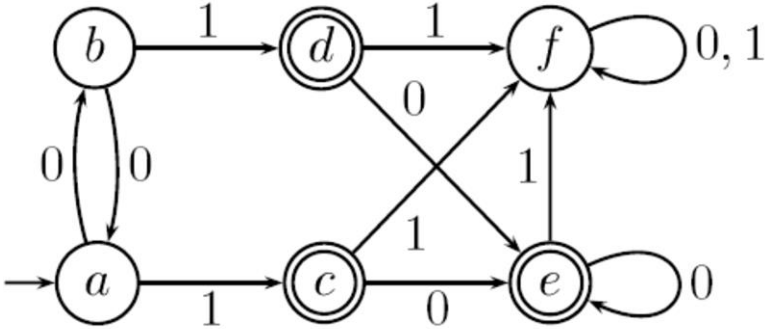
There are no (i+1)-distinguishable states if there are no (i)-distinguishable states

The method:

We construct a triangular table to compare each pair of states. We write X_k in the corresponding position in the table, the first time we find that two states are *k*-distinguishable, as follows:

- We write X_0 to all pairs of states that are *0*-distinguishable because one is final and the other is not.
- In each “round" *i+1*, we examine all unmarked pairs and write X_{i+1} to a pair, if out of its two states with *a symbol, the* DFA reaches *i* -distinguishable states (already marked with X_i).
- Repeat until a round *k* where there is no pair marked with X_k .
- Unmarked pairs correspond to equivalent states (which are therefore merged).

Applying the method: Example

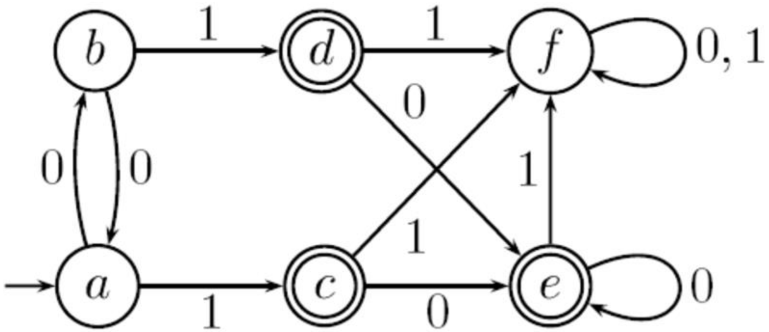


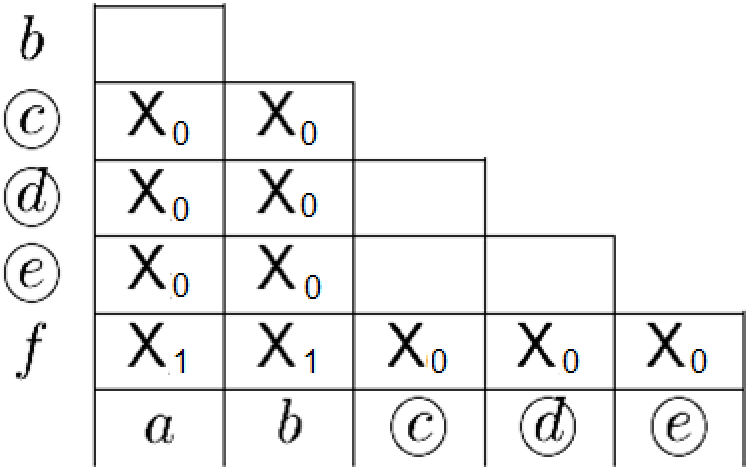
b					
ⓒ	X_0	X_0			
ⓓ	X_0	X_0			
ⓔ	X_0	X_0			
f			X_0	X_0	X_0
	a	b	ⓒ	ⓓ	ⓔ

Round 0:

nine pairs
0-distinguishable
states

Applying the method: Example



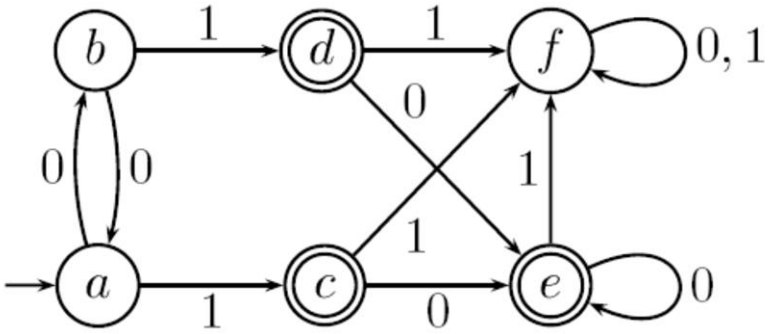


b					
ⓒ	X_0	X_0			
ⓓ	X_0	X_0			
ⓔ	X_0	X_0			
f	X_1	X_1	X_0	X_0	X_0
	a	b	ⓒ	ⓓ	ⓔ

Round 1:

Two pairs
1-distinguishable
states

Applying the method: Example

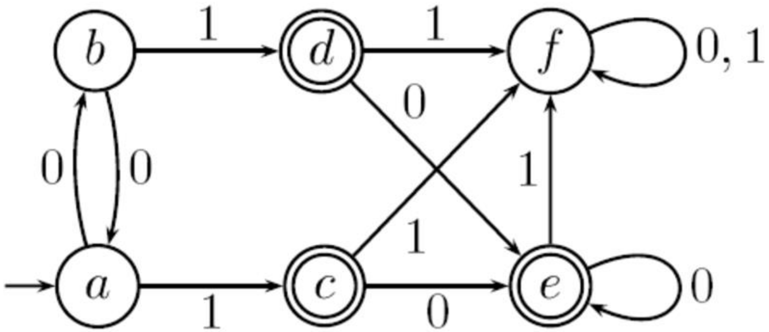


b					
ⓒ	X_0	X_0			
ⓓ	X_0	X_0			
ⓔ	X_0	X_0			
f	X_1	X_1	X_0	X_0	X_0
	a	b	ⓒ	ⓓ	ⓔ

Round 2:

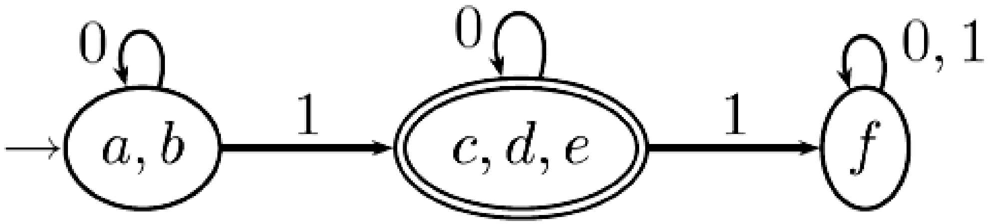
No pair
2-distinguishable
states

Applying the method: Example



b					
ⓒ	X_0	X_0			
ⓓ	X_0	X_0			
ⓔ	X_0	X_0			
f	X_1	X_1	X_0	X_0	X_0
	a	b	ⓒ	ⓓ	ⓔ

$$a \equiv b, c \equiv d \equiv e$$



Languages, Automata, Grammars

- Formal Languages: used to describe computational problems and programming languages.
- Automata: used to recognise formal languages and rank the difficulty of the corresponding problems.
- Formal Grammars: another way of describing formal languages. Every formal grammar *produces* a formal language.

▲

Language and Grammar Theory

Applications:

- Digital Design,
- Programming Languages,
- Compilers,
- Artificial Intelligence,
- Complexity Theory.

Important researchers:

- Chomsky, Backus, Rabin, Scott, Kleene, Greibach, …

Formal languages

- Primary concepts: symbols, concatenation.
- Alphabet: a finite set of symbols. e.g. {0,1}, {x,y,z}, {a,b}.
- Word (or string, or sentence) of an alphabet: a finite-length sequence of symbols of the alphabet. e.g. 011001, abbbab.
- $|w|$: length of word w.
- ε : empty word, $|\varepsilon| = 0$.
- prefix, suffix, substring, reversal, palindrome.

▲

Formal languages (cont.)

- *vw* = concatenation of words *v and w*.
- *εx* = *xε* = *x*, for every string *x*.
- define x^n with primitive recursion:

$$\begin{cases} x^0 = \varepsilon \\ x^{k+1} = x^k x \end{cases}$$

- Σ*: the set of all the words of alphabet Σ.
- Language from alphabet Σ: any set of strings $L \subseteq \Sigma^*$.

▲

Formal Grammars

- A systematic way to transform strings through production rules.
- Alphabet: terminal and non-terminal symbols and a start symbol (non-terminal).
- Finite set of rules of the form $\alpha \rightarrow \beta$: define the capability of replacing string α with string β.
- Every formal grammar produces a formal language: the set of strings (with only *termination symbols*) that are produced from the start symbol.
- Also known as rewriting systems or phase structure grammars.

Example: Grammar for the language of odd numbers

$S \rightarrow A\ 1$
$A \rightarrow A\ 0$
$A \rightarrow A\ 1$
$A \rightarrow \varepsilon$

S: start symbol

A: non-terminal symbol

0,1: terminal symbols

ε: the empty string

- *S* and *A* are replaced according to the rules.
- Every odd is produced from S with some sequence of valid substitutions.
- regular expression: (0+1)*1

Formal grammars: definitions (i)

A **formal grammar** ***G*** consists of:

- An alphabet *V* of *non-terminal* symbols (variables),
- An alphabet *T* of *terminal* symbols (constants), s.t. $V \cap T = \emptyset$,
- A finite set *P* of *production rules*, i.e. ordered pairs (α,β), where $\alpha,\beta \in (V \cup T)^*$ and $\alpha \neq \varepsilon$ (convention: we write $\alpha \rightarrow \beta$ instead of (α,β)),
- a *start symbol* (or axiom) $S \in V$.

▲

Formal grammars: definitions(ii)

Convention for the use of letters:

- $a, b, c, d, \ldots \in T$: lowercase Latin, the initials of the alphabet, represent terminals
- $A, B, C, D, \ldots \in V$: capital Latin, represent non-terminals
- $u, v, w, x, y, z \ldots \in T^*$: lowercase Latin, the last of the alphabet, represent terminal strings
- $\alpha, \beta, \gamma, \delta, \ldots \in (V \cup T)^*$: Greek represent any strings (terminal and non-terminal)

Formal grammars: definitions(iii)

Production definitions:

- We say that $\gamma_1\alpha\gamma_2$ *produces* $\gamma_1\beta\gamma_2$, and we denote it by $\gamma_1\alpha\gamma_2 \Rightarrow \gamma_1\beta\gamma_2$, if $\alpha \rightarrow \beta$ is a production rule (i.e. $(\alpha,\beta) \in P$).
- We denote by $\overset{*}{\Rightarrow}$ the reflexive transitive closure of $\Rightarrow$, (*α derives β in zero or more steps*), $\alpha \overset{*}{\Rightarrow} \beta$ means that there is a sequence $\alpha \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \alpha_k \Rightarrow \beta$.
- *Language generated* by grammar G:

$$L(G) := \{w \in T^* \mid S \overset{*}{\Rightarrow} w\}$$

- grammars G_1, G_2 *equivalent* if $L(G_1) = L(G_2)$.

Formal grammars: Example

$$\boxed{G\colon\ V = \{S\},\ T = \{a, b\},\ P = \{S \rightarrow \varepsilon | aSb\}}$$

$S \rightarrow \varepsilon \mid \alpha Sb$: abbreviation of $S \rightarrow \varepsilon$ and $S \rightarrow \alpha Sb$

A possible production sequence:

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

Produced language:

$$\boxed{L(G) = \{a^n b^n \mid n \in N\}}$$

Chomsky Hierarchy

Type 0: (general grammars, unrestricted, phrase structure, semi-Thue)

$\alpha \rightarrow \beta, \alpha \neq \varepsilon$

Type 1: (context sensitive grammars, monotonic)

$\alpha \rightarrow \beta, |\alpha| \leq |\beta|$ $(S \rightarrow \varepsilon, \text{allowed})$

Type 2: (context free grammars)

$A \rightarrow \alpha$ $(A \in V)$

Type 3: (regular grammars)

Right linear: $A \rightarrow w, A \rightarrow wB$ $(w \in T^*, A, B \in V)$ or,

Left linear: $A \rightarrow w, A \rightarrow Bw$ $(w \in T^*, A, B \in V)$

Type 3 ⊂ Type 2 ⊂ Type 1 ⊂ Type 0

Chomsky Hierarchy

- **Type 0** ↔ Turing Machines
- **Type 1** ↔ Linear Bounded Automata
- **Type 2** ↔ PushDown Automata
- **Type 3** ↔ DFA (and NFA)

Regular Grammars

- **Regular grammars** are grammars where all the rules are of the form:
  - Right linear

$$A \to wB \text{ or } A \to w$$

  - Left linear

$$A \to Bw \text{ or } A \to w$$

(w is a string of terminal language symbols)

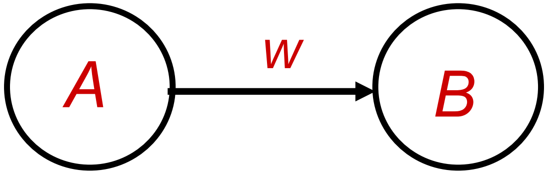
Theorem:

▲

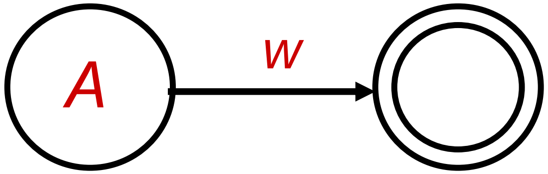
Regular languages *are equivalent to languages produced by* regular grammars.

Equivalence of regular grammars and DFA

- Using the right-linear format:
  - $A \rightarrow wB$ corresponds to



  - $A \rightarrow w$ corresponds to



  - S corresponds to q_0

Another equivalence!

Theorem:

Regular languages *are equivalent to languages described by* regular expressions.

Regular expressions

Languages L, L_1, L_2 Same alphabet Σ.

- $L_1 L_2 := \{uv \mid u \in L_1 \wedge v \in L_2\}$: *concatenation*
- $L_1 \cup L_2 := \{w \mid w \in L_1 \vee w \in L_2\}$: *union*
- $L_1 \cap L_2 := \{w \mid w \in L_1 \wedge w \in L_2\}$: *intersection*
- $L^0 := \{\varepsilon\}$, $L^{n+1} := LL^n$
- $L^* := \bigcup_{n=0}^{\infty} L^n$: *Kleene star*
- $L^+ := \bigcup_{n=1}^{\infty} L^n$

Regular expressions (definitions)

Regular expressions: represent languages derived from symbols of an alphabet using the operations of concatenation, union, and Kleene star.

- $\varnothing$: represents the empty language
- ε : represents $\{\varepsilon\}$
- α : represents $\{\alpha\}$, $\alpha \in \Sigma$
- $(r+s)$: represents $R \cup S$, $R = L(r)$, $S = L(s)$
- (rs) : represents RS, $R = L(r)$, $S = L(s)$
- (r^*) : represents R^*, $R = L(r)$

$L(t)$ the language represented by reg.ex. t

Regular expressions (examples)

$$L_1 = a(a+b)^*$$

$$L_2 = (b^*ab^*a)^*b^* = (b+ab^*a)^*$$

$$L_3$$

$$L_4 = (a+b)^*aa(a+b)^*$$

$$\overline{L_4} = (a+\varepsilon)(ba+b)^*$$

$$L_5 = a^*b^*$$

operator priority:

- Kleene star
- concatenation
- union

Equivalence of Regular Expressions and Finite Automata

Theorem. A language L can be described with a *regular expression* iff it is *regular* (i.e. *L*=*L*(*M*) for a finite automaton *M*).

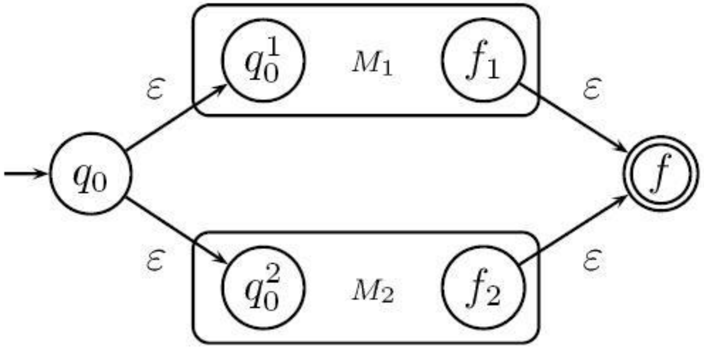
Proof (idea):

‘=>’ :   Induction on the structure of regular expression *r* :

1. Induction base case:

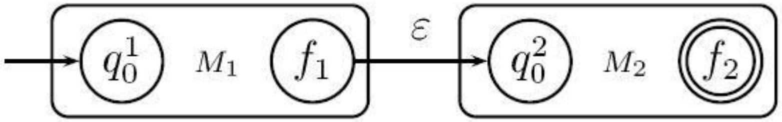
2. Induction step. Assume for regex r_1, r_2 automata M_1, M_2, with final states f_1, f_2:

α: $r = r_1 + r_2$



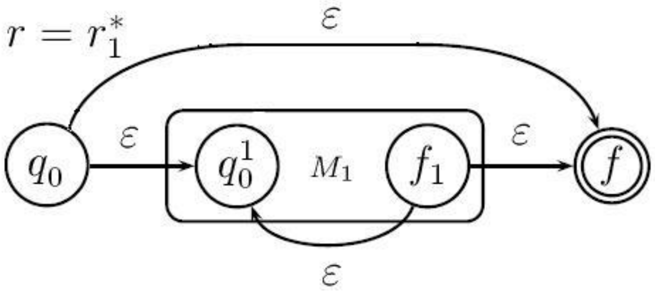
$$L(M) = L(M_1) \cup L(M_2)$$

β: $r = r_1 r_2$



$$L(M) = L(M_1)L(M_2)$$

γ: $r = r_1^*$

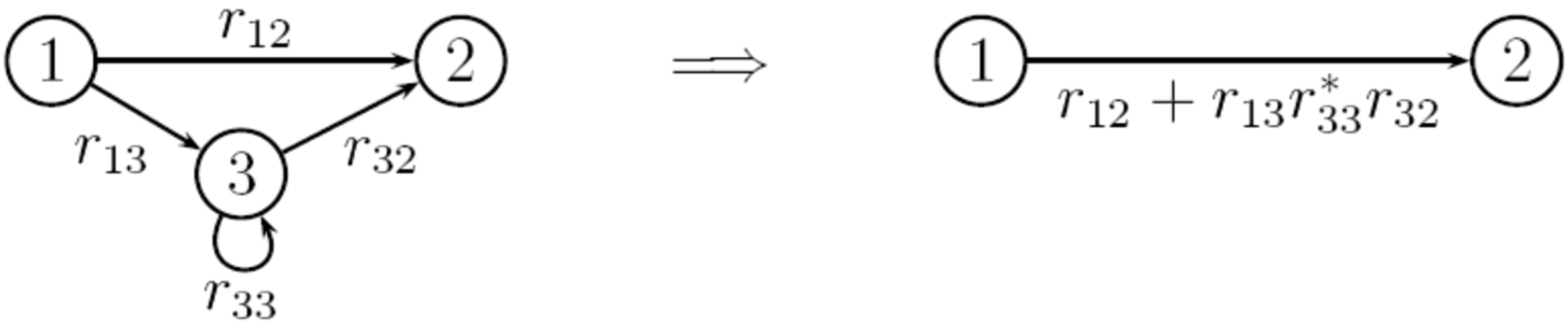


$$L(M) = L(M_1)^*$$

Equivalence of Regular Expressions and Finite Automata

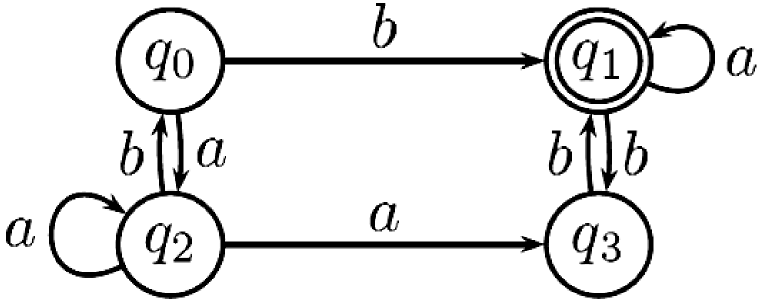
‘<=’ : Regular expression construction from a FA (GNFA).

Eliminate intermediate states:

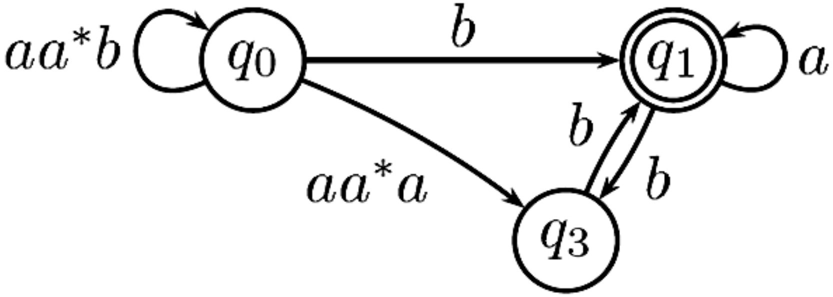


FA → Regular Expression (example)

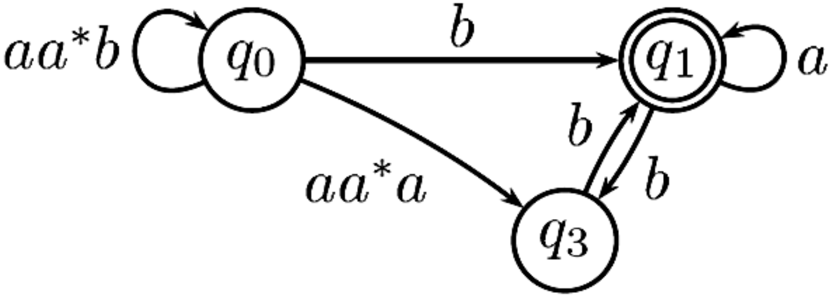
Initial NFA



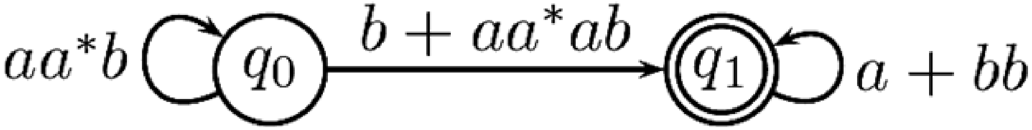
Delete q_2



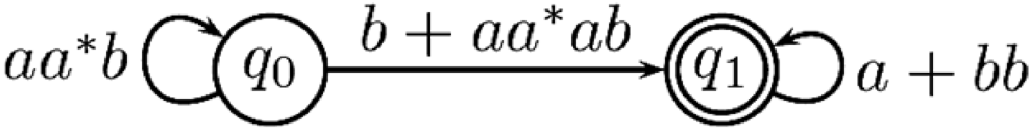
FA → Regular Expression (example)



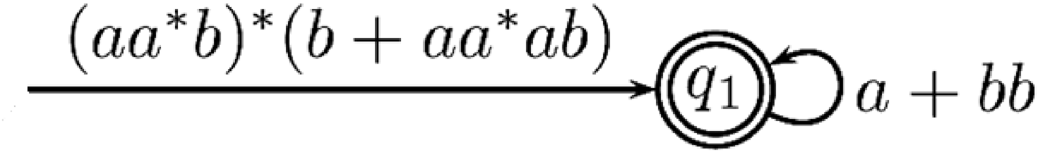
Delete q_3



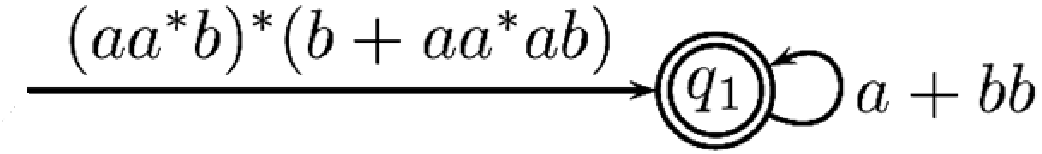
FA → Regular Expression (example)



Delete q_0



FA → Regular Expression (example)



Final expression:

$$\boxed{(aa^*b)^*(b+aa^*ab)(a+bb)^*}$$

Which languages are regular?

- All finite.
- All produced from regular languages using the operations: concatenation, union, Kleene star
- Also complementation, intersection, reverse, etc.
- Product of Automata: a way to construct DFA για intersection (and union) of regular languages.

Product of Automata: DFA

- Assume DFA M_1, M_2 with states *n*, *m* respectively ($Q_1=\{q_0, ..., q_{n-1}\}$, $Q_2=\{p_0, ..., p_{m-1}\}$) and a common alphabet, that recognize languages L_1, L_2 respectively.
- The **product of M_1, M_2** is a DFA with $m \cdot n$ states, one for every pair of states of the initial automaton (state set $Q = Q_1 \times Q_2$), the same alphabet and initial state (q_0,p_0).
- Transition function: $\delta'((q_i,p_j),\sigma)=(q_{i'}, p_{k'}) \Leftrightarrow \delta(q_i,\sigma)=q_{i'} \wedge \delta(p_k,\sigma)=p_{k'}$
- Final states: dependent on the operation between L_1, L_2 . For the intersection we set as finals, pairs that both are final states for M_1, M_2, for union pairs that include at least one final state.
- Note: You can easily implement other operations between L_1, L_2 (difference, symmetric difference) by appropriately defining the final states.

Product of Automata: NFA

- Defined in a similar way.

- Consider also ε-transitions and junk states

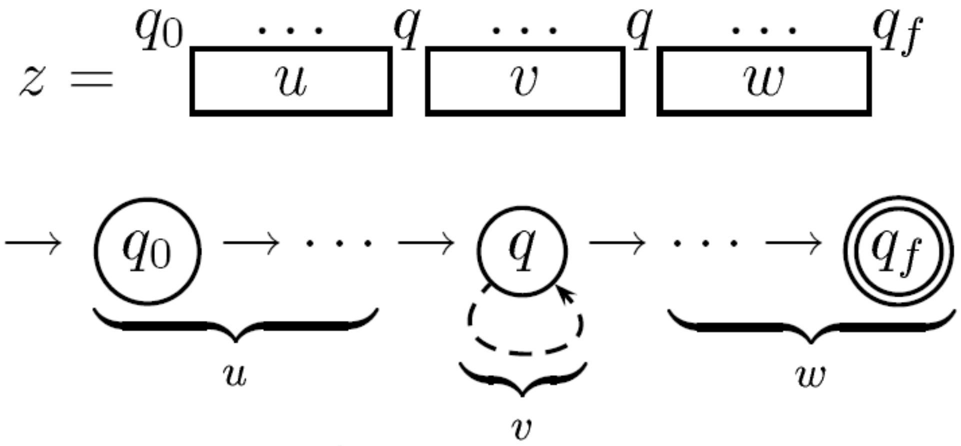
Are all languages regular?

- «No»

- To prove that we will use Pumping Lemma (or closure properties)

Pumping Lemma (intuition)

- If a language *L* is regular, then it is accepted by a DFA with a finite number of states, *n*.
- Let *z* be a word, $|z|>=n$ that belongs to the language, so it is accepted by the automaton.
- As we process *z*, the automaton has to go through a state again (pigeonhole principle):



- *Since* $z = uvw \in L$, $uv^iw \in L$, for all $i \in$ N

Pumping Lemma

Let a regular language L. Then:

- There exists a natural number n (= the number of states of DFA) such that:
- For all $z \in L$ with length $|z| \geq n$
- There exists a «split» of z *into substrings* u, v, w, i.e. $z = uvw$, $|uv| \leq n$ and $|v| > 0$
- So that for all $i = 0, 1, 2, \ldots$:

$$uv^iw \in L$$

▲

Use of the Lemma to prove non-regularity

Use of Pumping Lemma to prove that a (non-finite) language L *is not regular*:

Let L a regular language. Then:

- By the PL, there exists n. We for all n
- We choose a suitable $z \in L$ with length $|z| \geq n$
- By the PL, there exists «split» $z = uvw$, $|uv| \leq n$, $|v| > 0$. We for all «split» $z = uvw$, $|uv| \leq n$, $|v| > 0$
- We choose i so that the word uv^iw is not in the language L, *a CONTRADICTION!*

(*adversary argument*)

Using Pumping Lemma – Example (i)

- **Theorem.** *Language* $L = \{z \mid z$ has the same number of 0 and 1$\}$ *is not regular*.
- *Proof*: Suppose L regular. Then:
  - By the PL, there exists n. We for all n
  - We chose $z = 0^n 1^n \in L$ with length $|z| = 2n > n$

$$z = \underbrace{000000000...0}_{n}\underbrace{111111111...1}_{n}$$

  - By the PL, there exists a «split» $z = uvw$, $|uv| \leq n$, $|v| > 0$. We for all «splits» $z = uvw$, $|uv| \leq n$, $|v| > 0$

Using Pumping Lemma – Example (i)

- We observe that necessarily $v = 0^k$ for some k:
  - $w = \underbrace{0...0}_{u}\underbrace{0...0}_{v}\underbrace{0...0111111111...1}_{w}$
- And we choose $i = 2$, finding that $uv^iw = uv^2w$ is not a string of L, a contradiction.
- Therefore, L is not regular.

Using Pumping Lemma – Example (ii)

Theorem. *Language* $L = \{z \mid z=0^i1^j, i > j\}$ *is not regular*.

Proof: Suppose *L* regular. Then:

- By the PL there exists *n*. We for all *n*
- We chose $z = 0^{n+1}1^n \in L$ with length $|z| = 2n+1 > n$
  - $z = \underbrace{000000000...0}_{n+1}\underbrace{1111111...1}_{n}$
- By the PL there exists a «split» $z = uvw$, $|uv| \leq n$, $|v| > 0$. We for all «splits» $z = uvw$, $|uv| \leq n$, $|v| > 0$

Using Pumping Lemma – Example (ii)

- We observe that ότι necessarily $v = 0^k$ for some k:

$$z = \underbrace{0\ldots0}_{u}\underbrace{0\ldots0}_{v}\underbrace{0\ldots0111111111\ldots1}_{w}$$

  - But repeating v gives strings of the language
  - At first glance this seems problematic…
  - But PL states that for all $i \geq 0$: $uv^iw \in L$
- We choose $i = 0$: sting uv^0w is not a string of L, a contradiction.
- Therefore, L is not regular.

Attention when using PL!

- Pumping Lemma is a necessary but not sufficient condition for a language to be regular.
- There are non-regular languages that satisfy its conditions!
- It is therefore only useful for proving non-regularity.
- Another way of proving language non-regularity: closure of regular languages operations.

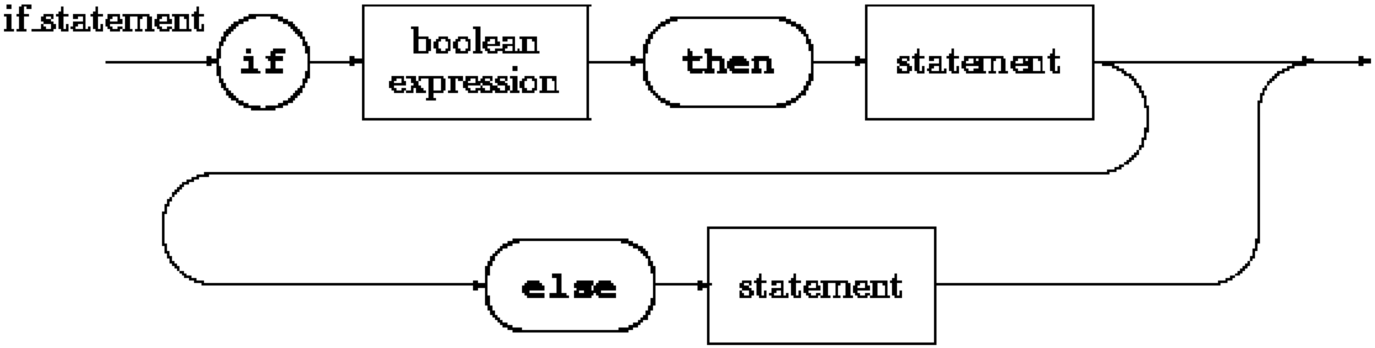
Grammars for non-regular languages

- Context-free (CF): type 2, corresponding to pushdown automata (PDA)
- Context-sensitive (CS): type 1, corresponding to Linear Bounded Automata (LBA)
- General: type 0, corresponding to Turing Machines (TM)

Context-Free Grammars (i)

Applications:

- Programming languages syntax (Pascal, C, C++, Java)



- Web pages description languages syntax (HTML, XML), editors, ...

Context-Free Grammars (ii)

- Rule form: $A \rightarrow \alpha$, A non-terminal
- Example:

$$G_1: \quad V = \{S\},\ T = \{a, b\},\ P = \{S \rightarrow \varepsilon, S \rightarrow aSb\}$$

Possible production sequence:

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

Produced Language:

$$\boxed{L(G_1) = \{a^n b^n | n \in \mathbb{N}^*\}}$$

Context-Free Grammars (iii)

- 2nd example:

G_2: $T = \{0,1,2,3,4,5,6,7,8,9,+,*\}$ $V = \{S\}$

$P: \; S \rightarrow S+S, \quad S \rightarrow S*S,$

$S \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Possible production sequences:

$S \Rightarrow 3, \quad S \Rightarrow S+S \Rightarrow 3+S \Rightarrow 3+S*S \Rightarrow 3+4*7$

Context-Free Grammars (iv)

- 3rd example:

G_3: $V = \{S, A, B\}$, $T = \{a, b\}$, and P includes:

$$S \to aB \mid bA, \quad A \to a \mid aS \mid bAA, \quad B \to b \mid bS \mid aBB$$

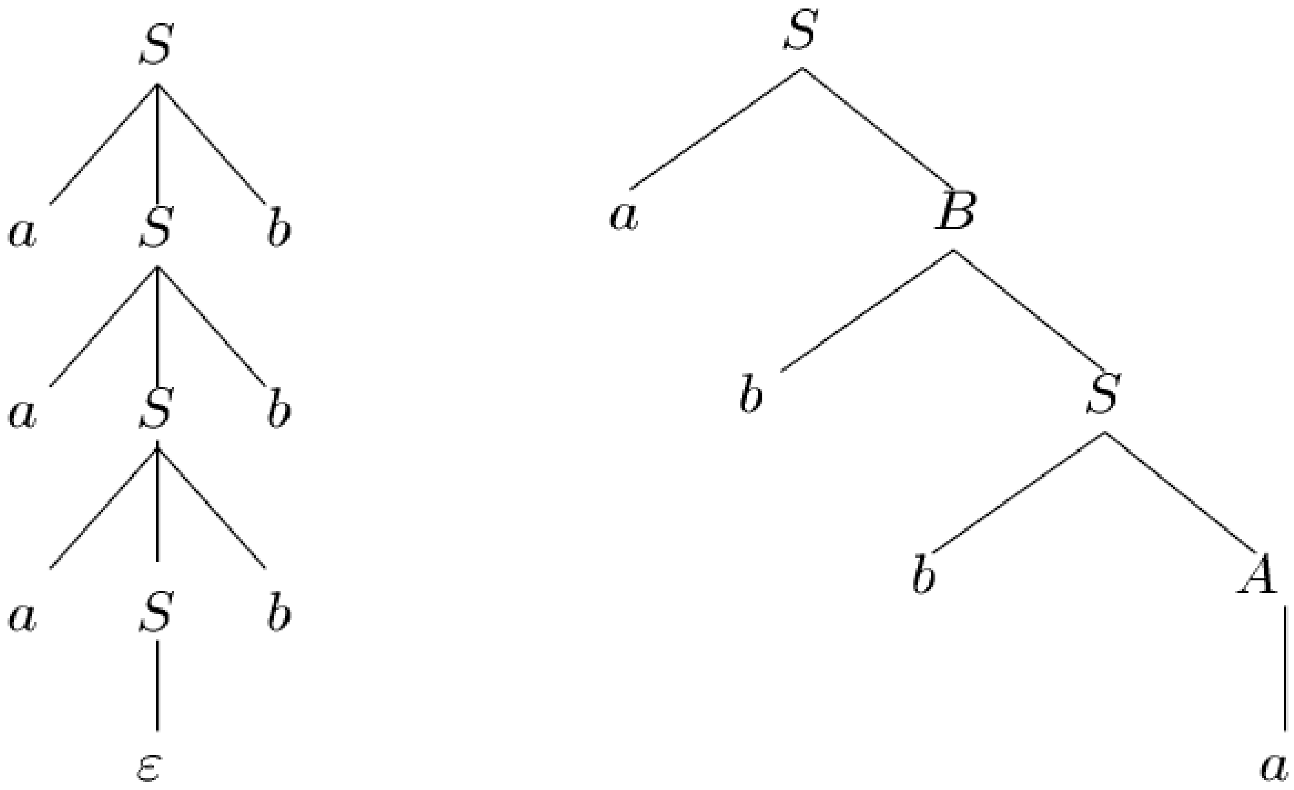
Possible production sequence:

$$S \Rightarrow aB \Rightarrow abS \Rightarrow abbA \Rightarrow abba$$

Produced Language (that is not obvious):

$$L(G_3) = \{w \in T^+ | w\ has\ an\ equal\ number\ of\ "a"\ and\ "b"\}$$

Parse Trees (i)



Leafstring: *aaabbb* and *abba* respectively.

Parse Trees (ii)

Let $G=\{V,T,P,S\}$ a context-free grammar. A tree is a parse tree of G if:

- Each node in the tree has a label, which is a symbol (terminal, or non-terminal, or ε).
- The label of the root is S.
- If an internal node is labeled A, then A is a non terminal symbol. If its children, from left to right, have labels $X_1,X_2,...,X_k$ then $A \rightarrow X_1,X_2,...,X_k$ is a production rule.
- If a node is labeled ε, then it is a leaf and is the only child of its parent.

Parse Trees (iii)

Theorem. *Let* $G=\{V,T,P,S\}$ *be a context-free grammar. Then* $S \overset{*}{\Rightarrow} \alpha$ *iff there exists a parse tree of G with leafstring* α.

▲

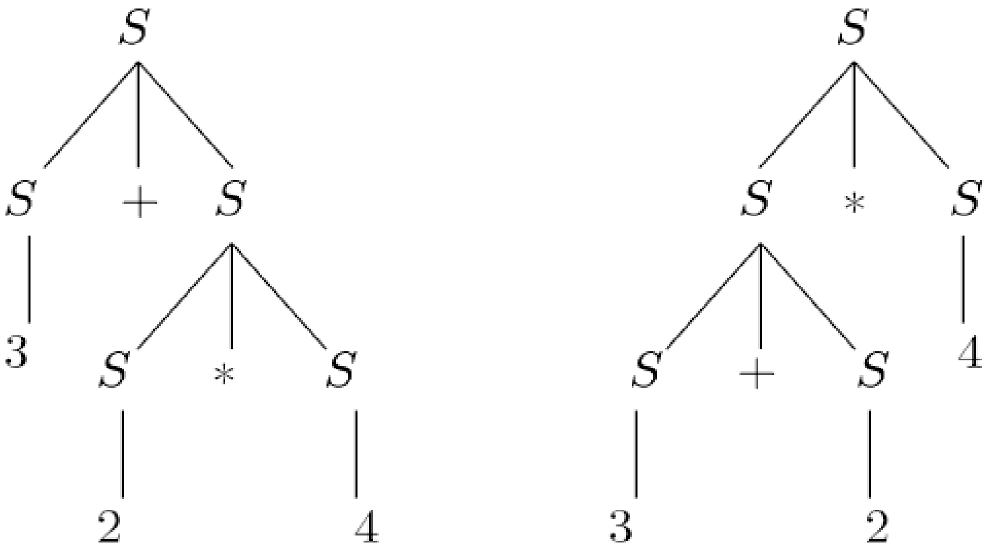
Ambiguous grammars

A grammar G is called ambiguous if two parse trees exist with the same leafstring $w \in L(G)$

Example:

G_2: $T = \{0,1,2,3,4,5,6,7,8,9,+,*\}$ $V = \{S\}$

$P: S \rightarrow S+S \mid S*S \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$



CF grammar recognition algorithm: CYK

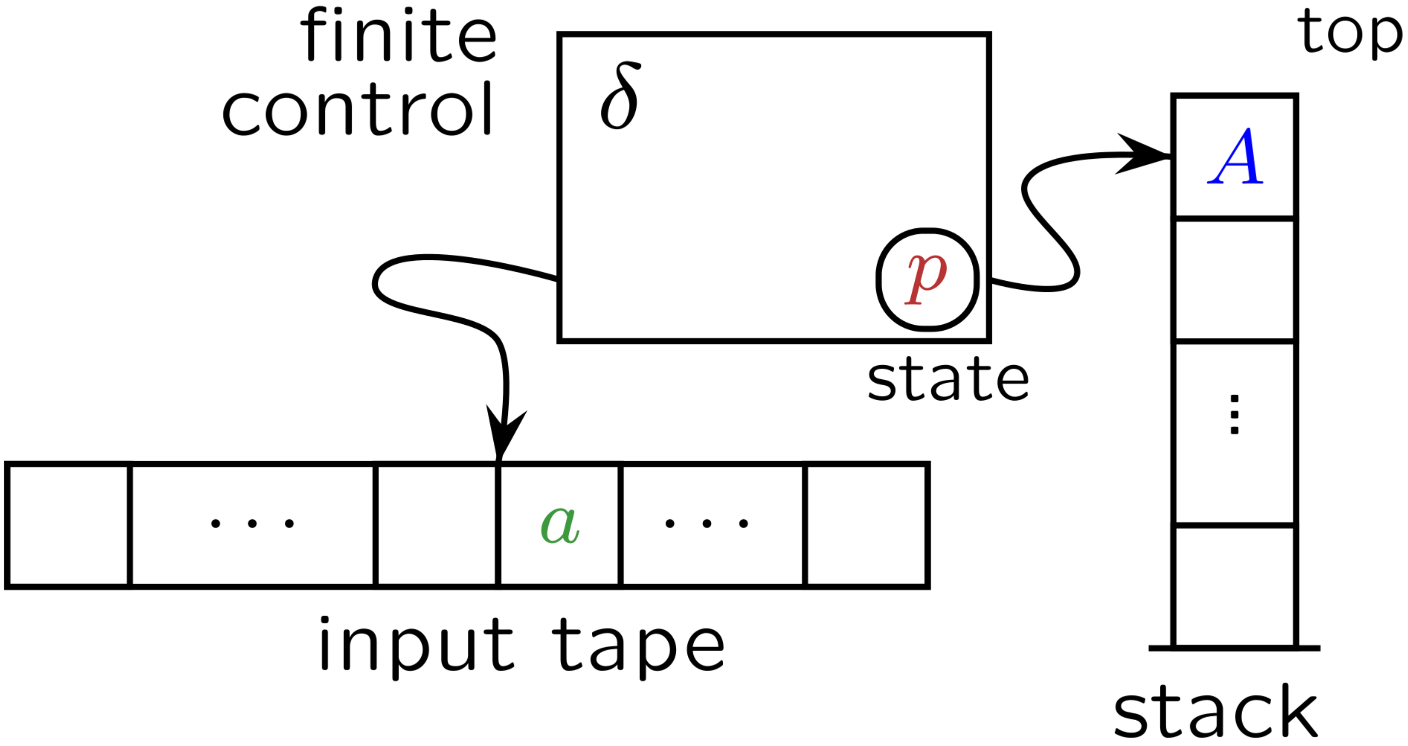
- In an exhaustive way we can decide whether a string x is generated by a CF grammar in exponential time.
- The properties of the Chomsky Normal Form allow for a faster recognition of a string.
- CYK algorithm (Cocke, Younger, Kasami): decides whether a string x is generated from a grammar in time $O(|x|^3)$, as long as the grammar is given in Chomsky Normal Form.

Pushdown Automata (PDA) (i)

- They have a one-way input tape (like FA) but have additional memory in the form of a stack.
- Access only to the top of the stack using functions:
    - push(*x*): places element *x* at the top of the stack
    - pop: reads and removes element from the top of the stack

▲

Pushdown Automata (PDA) (i)



Pushdown Automata (PDA) (ii)

Example: PDA for language recognition of

$$\boxed{L = \{wcw^R \mid w \in (0+1)^*\}}$$

Automaton description

- push(*a*) onto the stack for every 0 in the input, push(*b*) onto the stack for every 1 in the input, continue until *c* is read
- After that, pop: if the top stack element matches the input (*a* with 0, *b* with 1) continue
- Acceptance with an *empty stack*

PDA: formal definition

Pushdown Automaton, PDA:

tuple $M = (Q,\Sigma,\Gamma,\delta,q_0,Z_0,F)$

- Q : the set of states of M (finite)
- Σ : input alphabet
- Γ : stack alphabet
- $\delta : Q \text{ x } \Sigma \cup \{\varepsilon\} \text{ x } \Gamma \rightarrow \text{Pow}(Q \text{ x } \Gamma^*)$: transition function (non-determinism, ε-transitions)
- $q_0 \in Q$: initial state
- $Z_0 \in \Gamma$: initial stack symbol
- $F \subseteq Q$: set of final states

Pushdown Automata (PDA) (iv)

PDA acceptance

- If it reaches a final state (i.e. accepted) once the entire input has been read, regardless of stack content
- If the PDA has an empty stack once the entire input has been read, regardless of the state

Languages are defined accordingly:

- $L_f(M)$: acceptance with final state
- $L_e(M)$: acceptance with empty stack

Pushdown Automata (PDA) (v)

- For the following language to be accepted

$$\boxed{L_1 = \{ww^R \mid w \in (0+1)^*\}}$$

i.e. without the middle symbol *c* we necessarily need a non- deterministic PDA.

- Non-deterministic PDAs are more powerful than the deterministic ones.
- By PDA we usually refer to non-deterministic PDAs.

▲

CF grammars and PDA equivalence

Theorem. The following are equivalent for a language L:

- $L = L_f(M)$, M is PDA.
- $L = L_e(M')$, M' is PDA.
- L is context-free language

▲

Which languages are Context Free?

- All regular.
- Those formed from CF languages using the operations: concatenation, union, Kleene star.
- But not necessarily with the operations intersection, complement:

  e.g. language $\{a^n b^n c^n \mid n \in \mathbf{N}\}$ is not CF, while being an intersection of two CF languages:

  $$\{a^n b^n c^n \mid n \in \mathbf{N}\} = \{a^n b^n c^m \mid n,m \in \mathbf{N}\} \cap \{a^k b^n c^n \mid k,n \in \mathbf{N}\}$$

Are all languages Context Free?

- «No».
- To prove that we use another pumping lemma, the Pumping Lemma for context-free languages.
- It is based on the syntax tree (more in the course «Computability and Complexity»).

Context Sensitive Grammars (i)

Type 1: Context sensitive or monotonic

$$\alpha \to \beta, \quad |\alpha| \leq |\beta| \qquad S \to \varepsilon, \ \alpha \neq \varepsilon$$

«context sensitive» because they can be put in the following normal form:

$$\begin{array}{l} \alpha_1 A \alpha_2 \to \alpha_1 \beta \alpha_2, \\ \ \nwarrow \quad \nearrow \\ \mathit{context} \end{array}$$

Context Sensitive Grammars (ii)

CS grammar for the language $1^n 0^n 1^n$:

$S \to 1Z1$

$Z \to 0 \mid 1Z0A$

$A0 \to 0A$

$A1 \to 11$

Conversion to normal form
(1st attempt)

$A0 \to H0$ $\quad H0 \to HA$ $\quad HA \to 0A$

Not yet regular (why?)

Other examples: $\{1^i\, 0^j\, 1^k : i <= j <= k\}$,

$\{ ww \mid w \in \Sigma^* \}$, $\{a^n\, b^n\, a^n\, b^n \mid n \in \mathbf{N}\}$

Context Sensitive Grammars (ii)

CS grammar for the language $1^n 0^n 1^n$:

$S \rightarrow 1Z1$

$Z \rightarrow U \mid 1ZUA$

$AU \rightarrow UA$

$A1 \rightarrow 11$

$U \rightarrow 0$

Conversion to normal form

$AU \rightarrow HU \qquad HU \rightarrow HA \qquad HA \rightarrow UA$

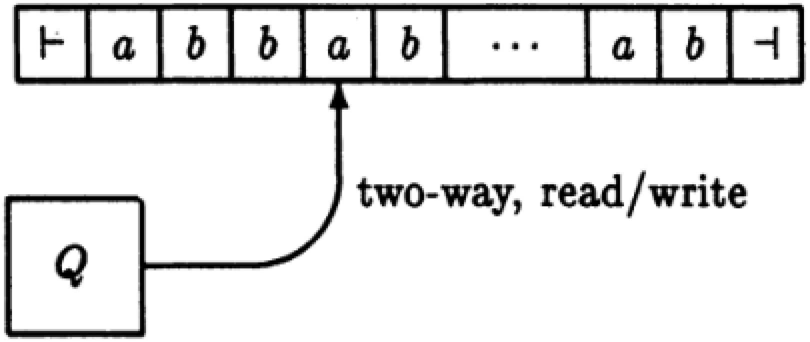
Other examples: $\{1^i\, 0^j\, 1^k : i <= j <= k\}$,

$\{ ww \mid w \in \Sigma^* \}$, $\{a^n\, b^n\, a^n\, b^n \mid n \in \mathbf{N}\}$

CS grammars and LBA equivalence

Linear Bounded Automaton (LBA):

Is a non-deterministic Turing Machine that its head is constrained to move only in the part containing the initial input.



Equivalent form:

PDA with 2 stacks, linearly bounded.

Theorem. The following are equivalent (*L* without ε):

1. Language *L* is accepted by LBA.
2. Language *L* is context sensitive.

General Grammars (i)

Type 0: general, unrestricted

$$\alpha \longrightarrow \beta, \alpha \neq \varepsilon$$

Example: $\{a^{2^n} \mid n \in \mathbf{N}\}$

$$\begin{aligned}
&S \rightarrow AaCB\\
&CB \rightarrow E \mid DB\\
&aE \rightarrow Ea\\
&AE \rightarrow \varepsilon\\
&aD \rightarrow Da\\
&AD \rightarrow AC\\
&Ca \rightarrow aaC
\end{aligned}$$

General Grammars (ii)

Theorem. The following are equivalent:

1. Language *L* is accepted by a Turing Machine
2. *L*=*L*(*G*), where *G* is a general grammar

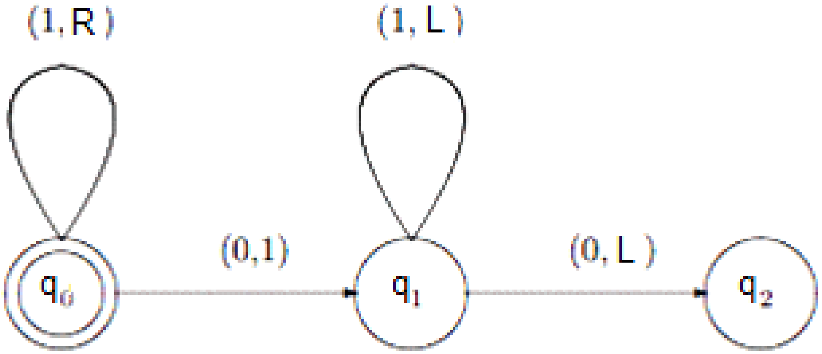
Such a language is also called *recursively enumerable*.

▲

Turing Machines

Automata with indefinitely long tape. Input is initially written in the tape, the head can move left-right and write symbols on the tape.

Transition function example:



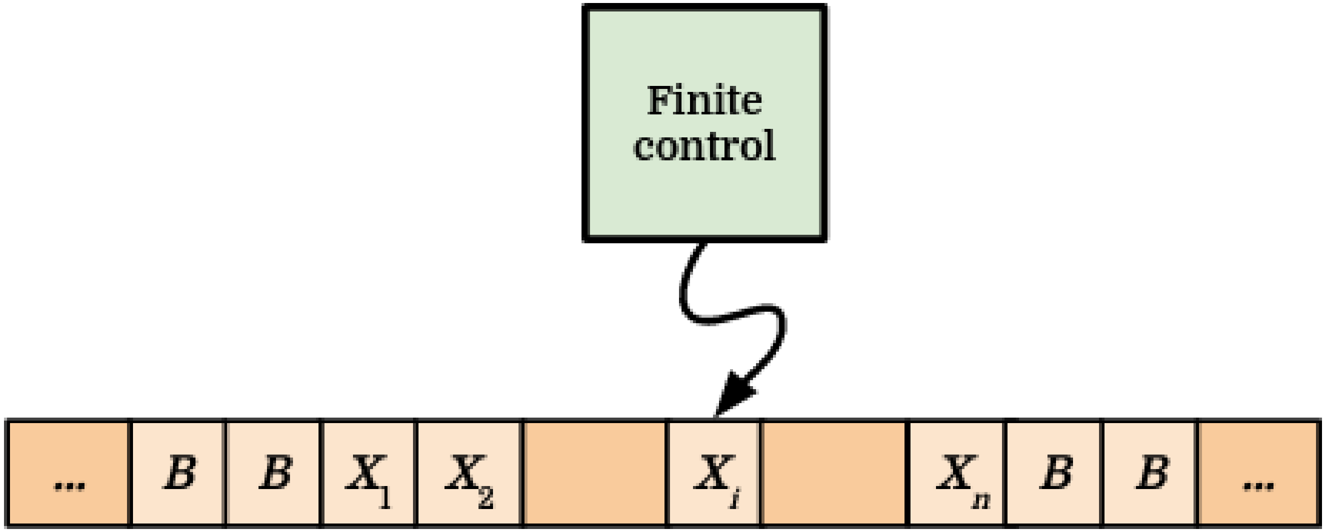
$< q_0, 1, q_0, R >$

$< q_0, 0, q_1, 1 >$

$< q_1, 1, q_1, L >$

$< q_1, 0, q_2, R >$

Turing Machines



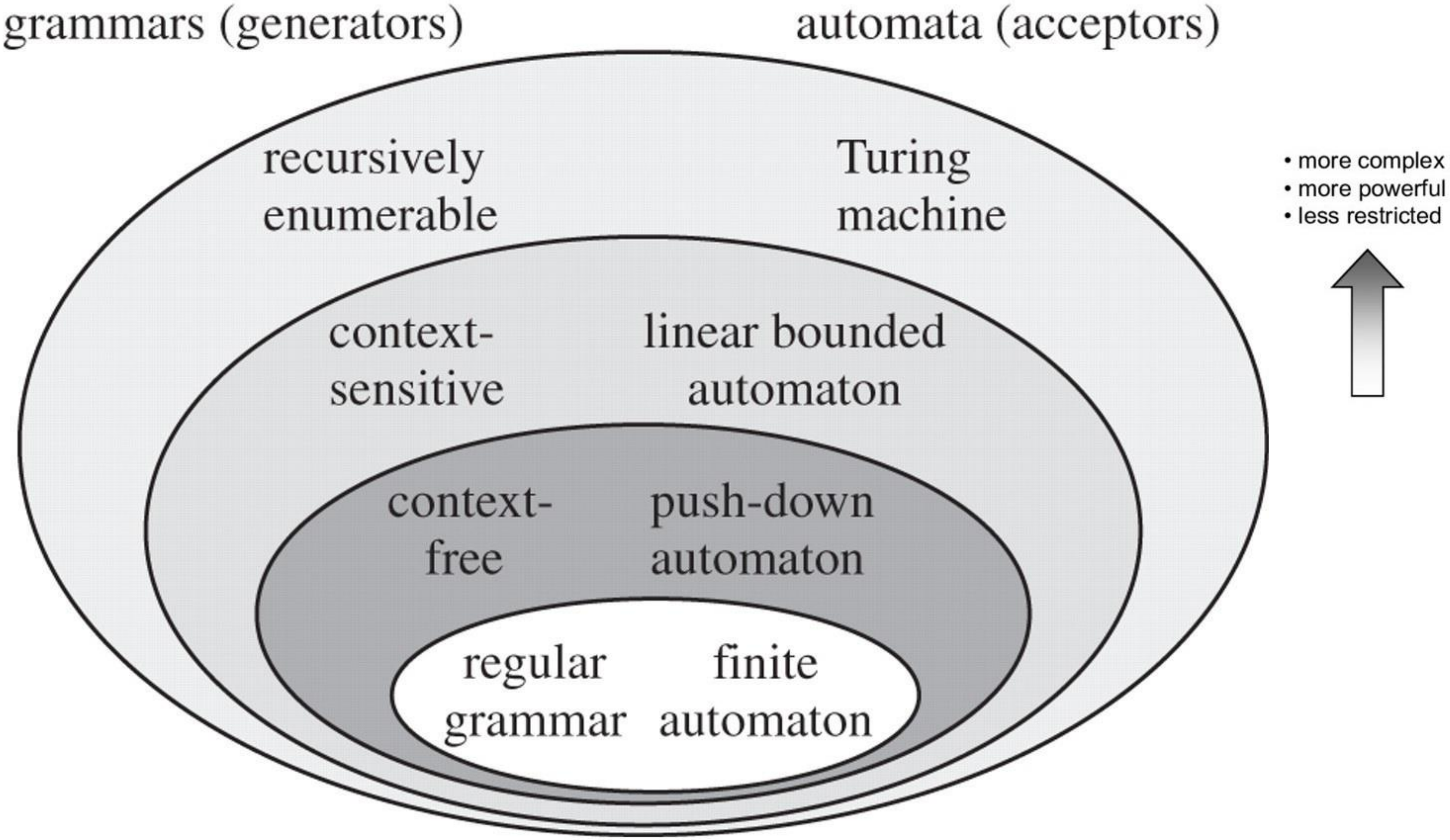
Language class hierarchy

Hierarchy Theorem.

regular $\subsetneq$ context free $\subsetneq$ context sensitive $\subsetneq$ recursiverly enumerable

- Type 0 ↔ TM (Turing Machines)
- Type 1 ↔ LBA (Linear Bounded Automata)
- Type 2 ↔ PDA (Pushdown Automata)
- Type 3 ↔ DFA (and NFA)

Language class hierarchy



- more complex
- more powerful
- less restricted

Chomsky Hierarchy Levels. Source: W. Tecumseh Fitch, Toward a computational framework for cognitive biology: Unifying approaches from cognitive neuroscience and comparative cognition, Physics of Life Reviews, Volume 11, Issue 3, 2014, Pages 329-364, ISSN 1571-0645