Theoretical Computer Science

Course Presentation

Objectives and motivations

Why Theoretical Computer Science in an Engineering Curriculum?

Theory is stimulated by practice,

practice is helped by theory:

generality, rigor, insight, “control”

- Engineer is a person who:
  - Applies theory to practice
  - Extracts theory from practice

- Deeply and critically understand the principles of Computer Science (careful re-view of basic CS notions)
- Build solid ground to understand and master innovation (e.g.: multimedia, modeling of concurrent and wireless computation)
- Theory as an antidote to overspecialization and as a support to interdisciplinary studies

- Throw a bridge between basic, applicative courses and more advanced ones (SW engineering, Hardware and computer architecture, distributed systems…)
- Direct application of some notions of TCS to practical cases: in follow-up courses such as Formal Languages and Compilers, Formal Methods, and thesis work

The program (1/2)

- Modeling in Computer Science
  (How to describe a problem and its solution):
  Do not go deep into specific models, rather
  provide ability to understand models and invent new ones
- Theory of Computation:
  what can I do by means of a computer
  (which problems can I solve)?

The program (2/2)

- Complexity theory:
  how much does it cost to solve a problem through a computer?
- Only the basics: further developments in follow-up courses

Organization (1/3)

- Requirements :
  - Basics of Computer science (CS 1)
  - Elements of discrete mathematics (Algebra and Logic)
- Lesson and practice classes (all rather classical style…)
  - Student-teacher interaction is quite appreciated:
    - In classroom
    - In my office (when possible)
    - By Email (administrative matters and to fix face-to-face meetings)
      - angelo.morzenti@polimi.it

Organization (2/3)

- Exam tests *ability to apply, not to repeat*:
  mainly written
  open book when in presence (you can consult textbooks and notes)
  not repetitive, challenging (hopefully)
- Standard written (+ possibly oral) exams on the whole program

Organization (3/3)

- Teaching material:
  - Official text :
    - Ghezzi/Mandrioli: Theoretical Foundations of Computer Science [Wiley]
      - Not re-printed by the publisher: you get it, possibly, in a legal way, see the WeBeep course website
    - In Italian: Dino Mandrioli, Paola Spoletini, Informatica Teorica, 2nd Ed. Città Studi, 2011.
  - Dino Mandrioli, Paola Spoletini, Mathematical logic for computer science: an Introduction, Esculapio Società Editrice, 2010.
  - D.Mandrioli, D.Martinenghi, A.Morzenti, M.Pradella, M.Rossi: Lecture Notes on Monadic First- and Second-Order Logic on Strings, lecture notes, 2019, on the WeBeep course website
  - English translation of the Italian Exercise book: Mandrioli, Lavazza, Morzenti, San Pietro, Spoletini, Esercizi di informatica teorica, 3rd edition, Esculapio, on the WeBeep course website
  - Previous exam texts with solutions, on the WeBeep course website
  - Lesson slides (NOT to use as a substitute of the text book!), on the WeBeep course website

Models in Computer Science

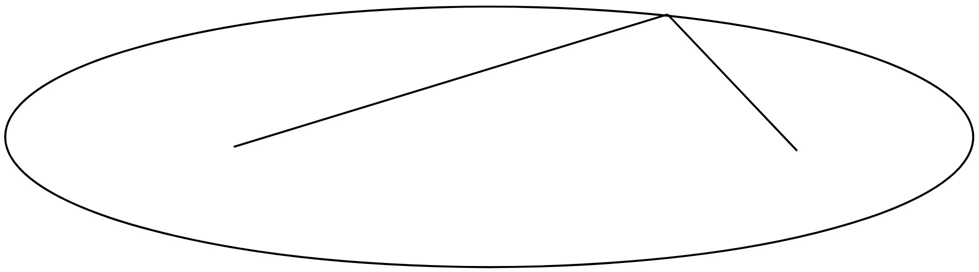
- Not only discrete vs. continuous
  (bits & bytes vs. real numbers and continuous functions)

- Operational models
  (abstract machines, dynamic systems, …)
  based on the concept of a **state** and of means (operations) to represent its evolution (chronologically, i.e., w.r.t. "time")

- Descriptive models
  aimed at expressing properties (desired or feared) of the modeled system, rather than its functioning as an evolution in time through states

Examples

- Operational model of an ellipsis (how you can draw it):



- descriptive model of it (***property*** of the coordinates of its points):

$$\boldsymbol{a \cdot x^{2} + b \cdot y^{2} = c}$$

- Operational definition of sorting:
  - Find minimum element and put it in first position;
  - Find minimum element among the remaining ones and put it in second position;
  - …
- Descriptive definition of sorting:
  - A permutation of the sequence such that

$$\forall i,\ a[i] \leq a[i+1]$$

- In fact, differences between operational and descriptive models are not so sharp
  - (grammars are somewhere in between)
- It is however a very useful classification of models
  - Corresponds closely to different “attitudes” that people adopt when reasoning on systems

A first, fundamental, “meta”model: the notion of *language*

- Italian, French, English, …
- C, Pascal, Ada, …

  but also:
- Graphics
- Music
- Multimedia, ...

Elements of a language

- Alphabet or vocabulary
  (from a mathematical viewpoint these are synonyms):
  Finite set of basic symbols
  {a,b,c, …z}
  {0,1}
  {Do, Re, Mi, …}
  ASCII char set
  ...

- String (over an alphabet A):
  ordered finite sequence of elements of A, possibly with repetitions
  a, b, aa, alpha, john, leaves of grass, …
- Length of a string:
  $|a| = 1$, $|ab| = 2$
- Null string, or empty string, ε: $|\varepsilon| = 0$
- A* = set of all strings, including ε, over A
  $A = \{0,1\}$, $A^* = \{\varepsilon, 0, 1, 00, 01, 10, \ldots\}$

- Operations on strings:
  concatenation (product): x · y

  x = abb, y = baba, x·y = abbbaba
  x = Quel ramo, y = del lago di Como,
  x · y = Quel ramo del lago di Como
  "·" is an associative, non-commutative operation

- A* is called
  free monoid over A built through "·"
- ε is the unit w.r.t. "·" : for any *x*, $\varepsilon \cdot x = x \cdot \varepsilon = x$

Language

- L is any set of strings, i.e. subset of A*: ($L \subseteq A^*$)
  Italian, C, Pascal, … but also:
  sequences of 0 and 1 with an even number of 1
  the set of musical scores in F minor
  symmetric square matrices
  …
- It is a very broad notion, somehow *universal*

Operations on languages

- Set theoretic operations:
  $\cup$, $\cap$, L_1 - L_2, $\neg L = A^* - L$, $(\overline{L} = \neg L)$

- Concatenation (of languages):
  $\boldsymbol{L_1 \cdot L_2 = \{\ x \cdot y \mid x \in L_1,\ y \in L_2\ \}}$

  $\boldsymbol{L_1 = \{0, 1\}^*, \quad L_2 = \{a, b\}^*}$ **;**
  $\boldsymbol{L_1 \cdot L_2 = \{\varepsilon, 0, 1, 0a, 11b, abb, 10ba, \ldots.\}}$ **NB: ab1 not included!**

Language power, inductive definition

- $L^0 = \{\varepsilon\}; \quad \forall i > 0 \;\; L^i = L^{i\text{-}1} \cdot L$
- $L^* = \bigcup_{n=0}^{\infty} L^n$

  NB: $\{\varepsilon\} \neq \emptyset$!
  $\{\varepsilon\} \cdot L = L$;
  $\emptyset \cdot L = \emptyset$

- $L^+ = \quad \bigcup_{n=1}^{\infty} L^n \qquad = \text{ (if } \;\varepsilon \notin L \text{) } = L^* \text{-} \{\varepsilon\}$

Some practical examples

- let
  - L_1 : set of “MS Office” documents, and
  - L_2 : set of “LibreOffice” documents; then
  - $L_1 \cap L_2$: set of documents that are “compatible MSOffice - LibreOffice”
- Composition of a message over the net:
  - $x \cdot y \cdot z$:
  - x = header (address, …)
  - y = text
  - z = “tail”
-
  - L_1 : set of e-mail messages
  - L_2 : set of SPAM messages
  - Filter: $L_1 - L_2$

- A language can be a means of expression …Of a ***problem***
- $x \in L$?
  - Is a message correct? Is a program correct?
  - $y = x^2$?
  - $z = Det(A)$? (is z the determinant of matrix A?)
- Problems of the kind «computing a function» can always be recast (by means of a suitable encoding) into problems of the kind «language string inclusion»
  - E.g., compute square of a number: computation of function $\boldsymbol{y=x^2}$
  - Language $L_{sq} = \{(x, y) \mid y = x^2\}$
    - To compute a^2 try repeatedly to solve $(a, b) \in L_{sq}$ until you find the right b
- Every string can be encoded in binary
  - every binary string can be interpreted as a natural number ...
  - … hence every problem can be recast (by suitable encoding) into an inclusion problem for a set of natural numbers

Translation

- $y = \tau(x)$
  τ : translation is a function from L_1 to L_2
  - τ_1 : double the “1” ($1 \Rightarrow 11$):
    $\tau_1(0010110) = 0011011110$, …
  - τ_2 : change *a* with *b* and viceversa ($a \Leftrightarrow b$):
    $\tau_2(abbbaa) = baaabb$, …
  - Other examples:
    - File compression
    - Self-correcting protocols
    - Computer language compilation
    - translation Italian ⇒ English

Conclusion

- The notion of language and the basic associated operations provide a very general and expressive means to describe any type of systems, their properties and related problems:
  - Compute the determinant of a matrix;
  - Determine if a bridge will collapse under a certain load;
  - ….
- Notice that, after all, in a computer any information is a string of bits (hence a string of some language…)

Operational Models

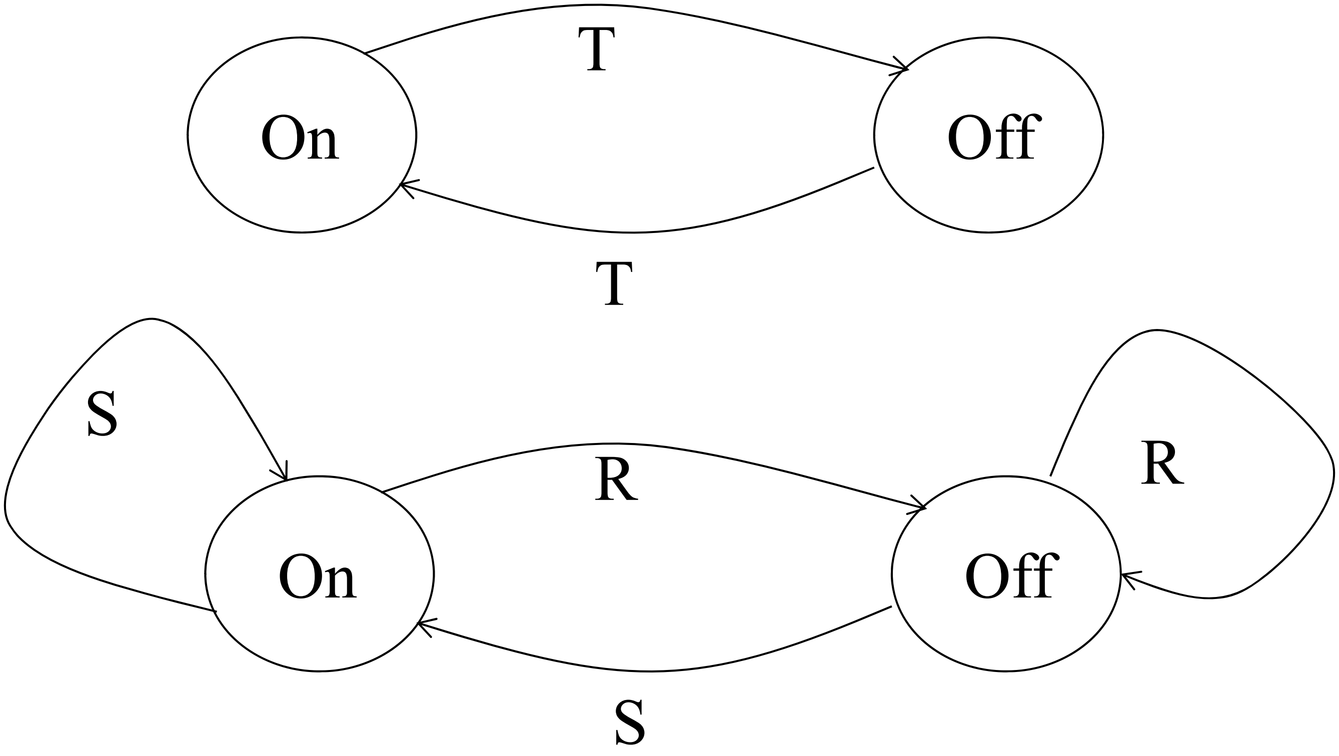
(state machines, dynamical systems)

- Finite State Machines (or *automata*) (*FSA*, or *FA*):
  - A finite state set:
    { on, off }, ….
    {1, 2, 3, 4, …k}, {TV channels}, …

    Graphic representation:

Commands (input) and state transitions

- Two very simple flip-flops:



Turning a light on and off, ...

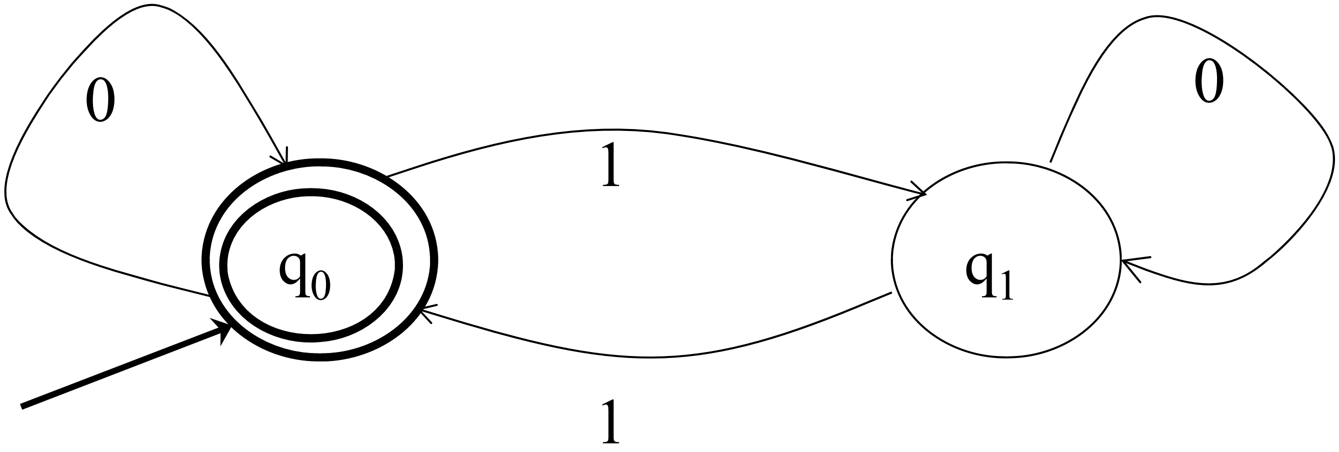
A first formalization

- A finite state automaton is (made of):
  - A finite state set: Q
  - A finite input alphabet: I
  - A transition function (***partial***, in general):
    $\delta: Q \times I \rightarrow Q$

Automata as language recognizers (or acceptors) (*x* ∈ *L*?)

- A *move sequence* starts from an ***initial state*** and it is *accepting* if it reaches a ***final*** (or *accepting*) *state*.

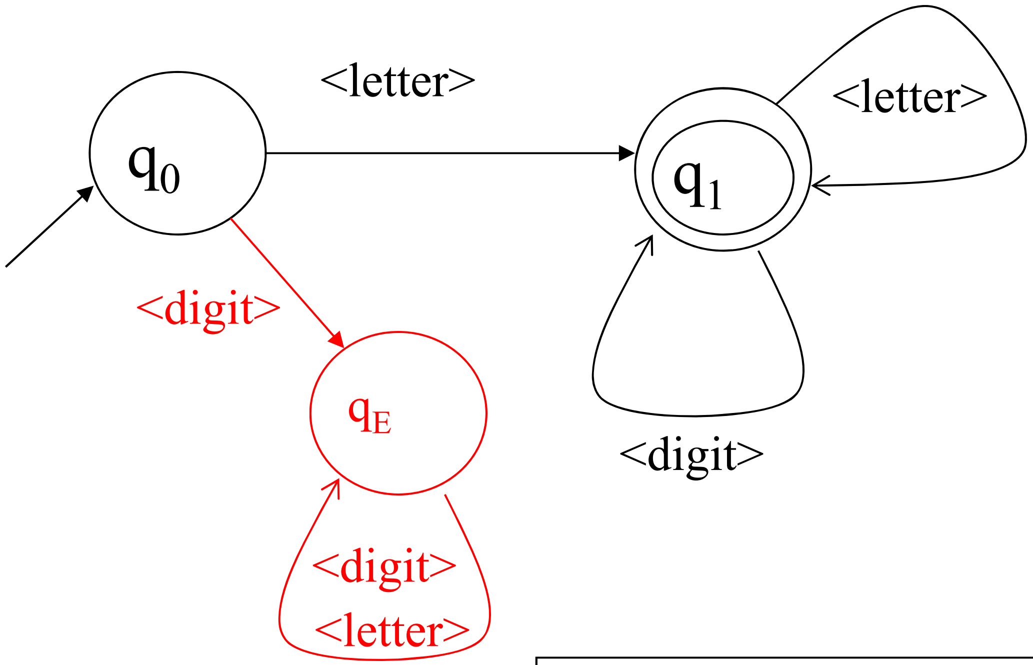
L = {strings with an even number of “1” any number of “0”}

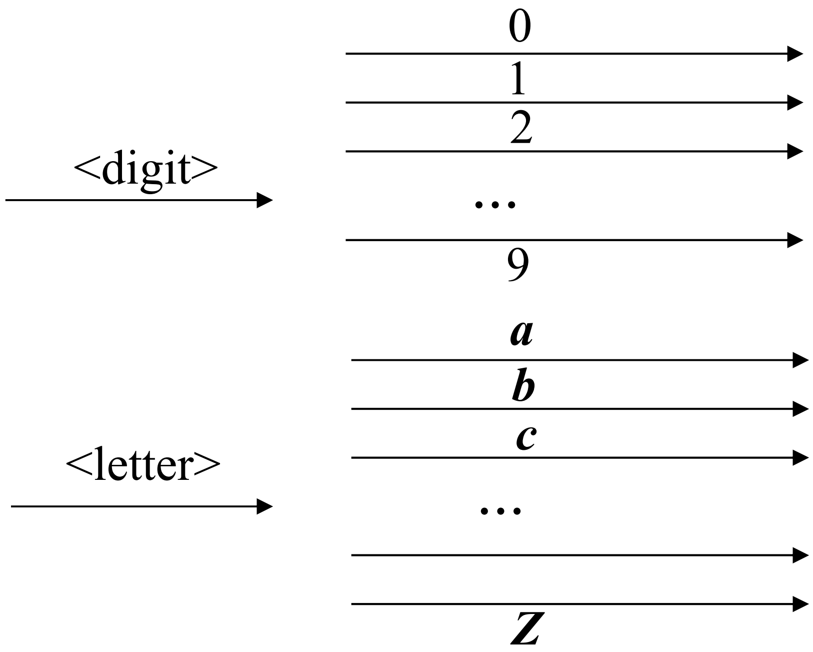


Formalization of the notion of acceptance of language *L*

- Move sequence for an input ***string*** (not just one symbol):
  - $\delta^*: Q \times I^* \rightarrow Q$
    δ^* defined inductively from δ, by induction on the string length
    $\delta^* (q, \varepsilon) = q$
    $\delta^* (q, y \cdot i) = \delta(\delta^* (q,y), i)$
- Initial state: $q_0 \in Q$
- Set of final, or accepting states: $F \subseteq Q$
- Acceptance: $x \in L \leftrightarrow \delta^* (q_0, x) \in F$

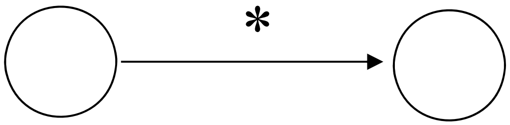
Accepting Pascal identifiers



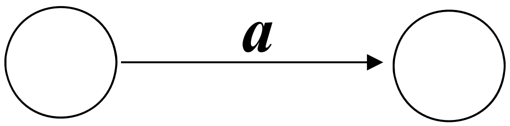


NB: the red part might be omitted

a useful notation/shorthand that is occasionally adopted



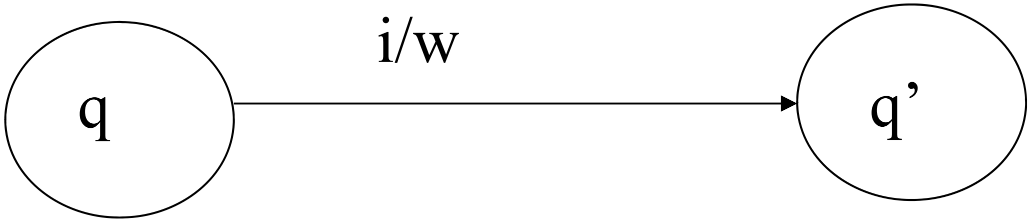
means



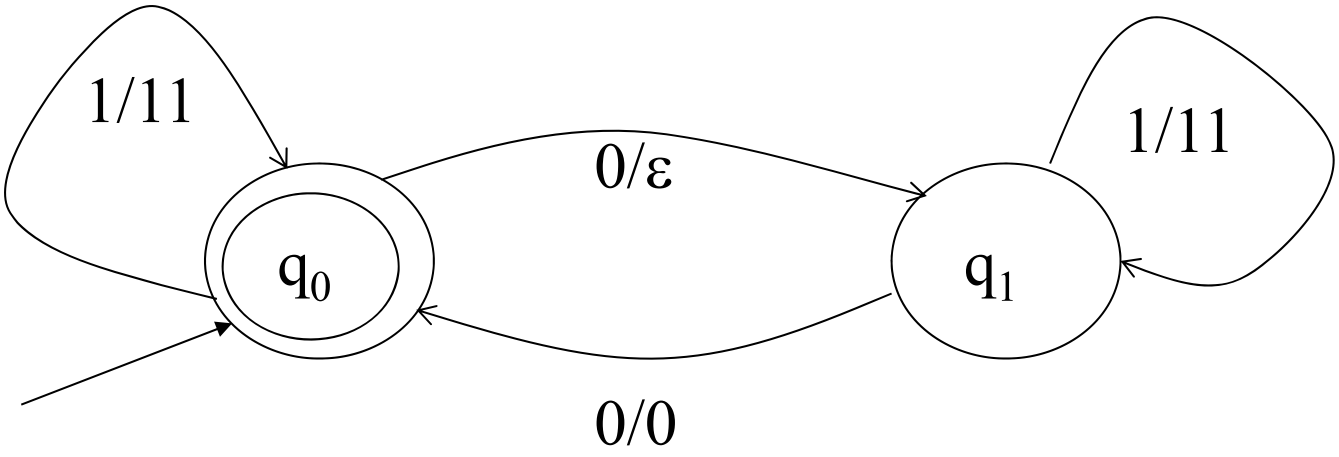
for all $\boldsymbol{a} \in$ **alphabet** $\boldsymbol{I}$

Automata as language translators
$y = \tau(x)$

Transitions with output: notice that w is a ***string*** (possibly ε)



τ: doubles the number of “1” and halves the “0” (the input must have an even number of “0”)



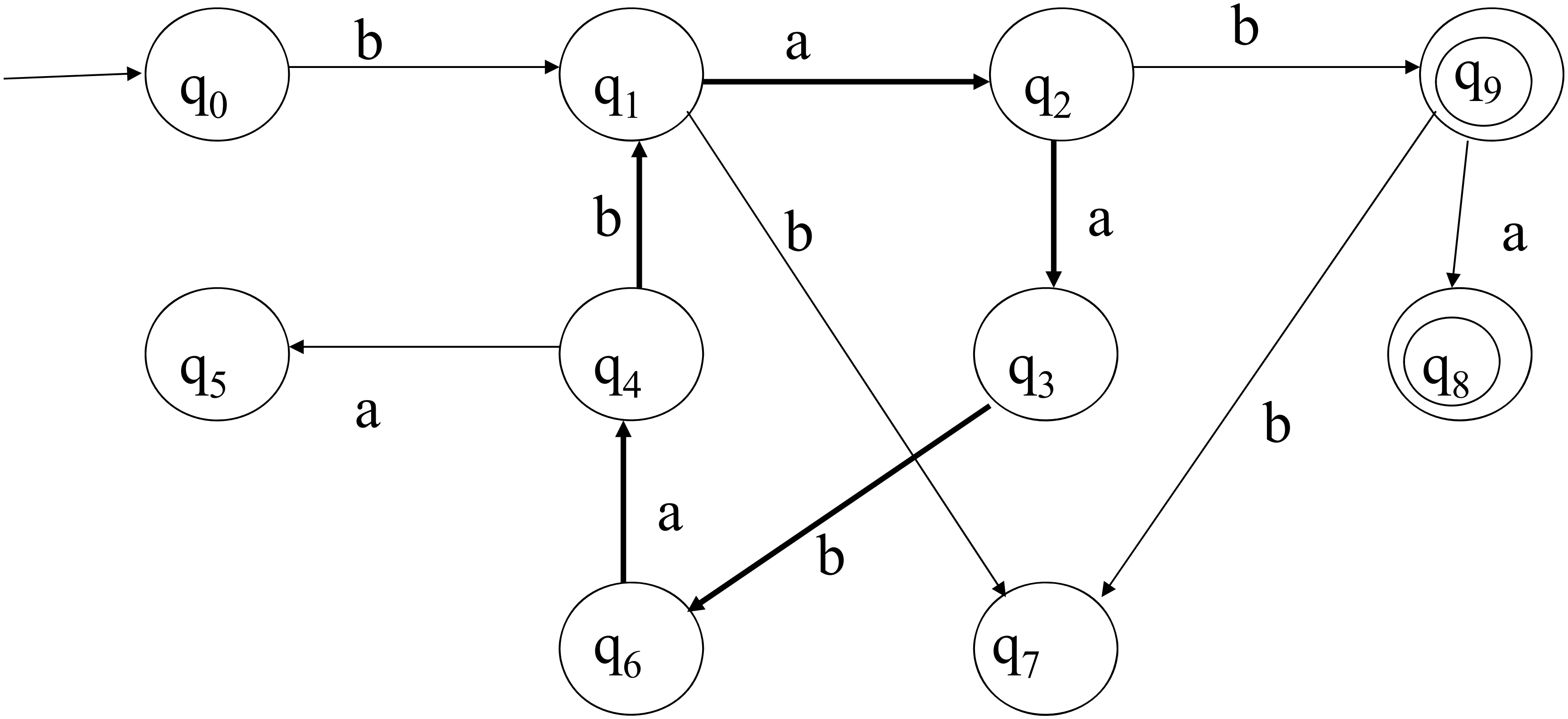
Formalization of translating automata (transducers)

- $T = <Q, I, \delta, q_0, F, O, \eta>$
  - $<Q, I, \delta, q_0, F>$: just like for acceptors
  - O: output alphabet
  - $\eta : Q \times I \rightarrow O^*$ (NB: the output is a string)
- $\eta^* : Q \times I^* \rightarrow O^*$, η^* defined inductively as usual
  $\eta^*(q, \varepsilon) = \varepsilon$
  $\eta^*(q, y \cdot i) = \eta^*(q, y) \cdot \eta(\delta^* (q,y), i)$
- definition of the translation [provided that…]

$$\tau(x)\ {\scriptstyle [x \in L]} = \eta^*(q_0, x)\ [\delta^* (q_0, x) \in F]$$

Analysis of the finite state model
(for the synthesis refer to other courses - e.g. on digital circuits)

- Very simple, intuitive model, applied in various applicative domains, also outside computer science
- Is there a price for this simplicity?
- …
- A first, fundamental property: the ***cyclic behavior*** of finite automata



There is a cycle q_1 ----***aabab***---> q_1

If, when reading a string, one goes through the cycle once, then one can also go through it 2, 3, …, n, … times

Formally:

- If $x \in L$ and $|x| > |Q|$ then there exists a $\boldsymbol{q \in Q}$ and a $\boldsymbol{w \in I^{+}}$ such that:

  $$\boldsymbol{x = ywz}$$

  $$\boldsymbol{\delta^{*}(q, w) = q}$$

  Then $\boldsymbol{y\, w^{n}\, z \in L, \forall\, n \geq 0}$:

- This is known as the **Pumping Lemma**

Several properties of FA –both good and bad ones– follow from the pumping lemma and other properties of the graph of δ

- Let A be an FA, and L=L(A) the language accepted by A
- $|L| = \infty$? $|L| = \infty$ iff there exists a cycle on the graph of δ with a node on a path from the initial state q_0 to a final state $q \in F$
- $L = \emptyset$? $\exists\ x \in L \leftrightarrow \exists\ y \in L, |y| < |Q|$:
  proof: Eliminate all cycles from A, then look for a path from initial state q_0 to a final state $q \in F$
- …
- Notice that being able to answer the question "***x* ∈ *L*** ?" for every possible given string ***x*** (i.e., for a ***generic*** string) does not enable us to answer other questions, such as the emptiness of the accepted language

A “negative” consequence of the Pumping Lemma (PL)

- Is the language $L = \{a^n b^n \mid n > 0\}$ recognized by any FA?
  - Put another way: can we count using a FA?
- Let us assume so (by contradiction):
- Consider $x = a^m b^m$, with $m > |Q|$ and apply the PL: then $x = ywz$ and $yw^n z \in L\ \forall n$
- There are 3 possible cases:
  - $x = ywz$, $w = a^k$, $k > 0$ ====> $a^{m+r.k} b^m \in L, \forall r$: this cannot be (it contradicts the assumption)
  - $x = ywz$, $w = b^k$, $k > 0$ ====> idem
  - $x = ywz$, $w = a^k b^s$, $k,s > 0$ ====> $a^{m-k} a^k b^s a^k b^s b^{m-s} \in L$: this cannot be (it contradicts the assumption)

- Intuitive conclusion: to "count" any ***n*** one needs an infinite memory!
- NB: strictly speaking any computer is a FA: so a computer cannot count?
  - But this is a wrong abstraction of (way of looking at) a computer!
  - It is important to consider a different, less immediate notion of infinity!
  - We consider the memory of a computer infinite as long as all the strings that it manipulates (confortably) fit into its (finite) memory.
- Going from the toy example $\{a^n b^n\}$ to more concrete ones:
  - Recognizing parenthetical structures like those of the programming languages cannot be done with a finite memory
- Therefore we need "more powerful" models

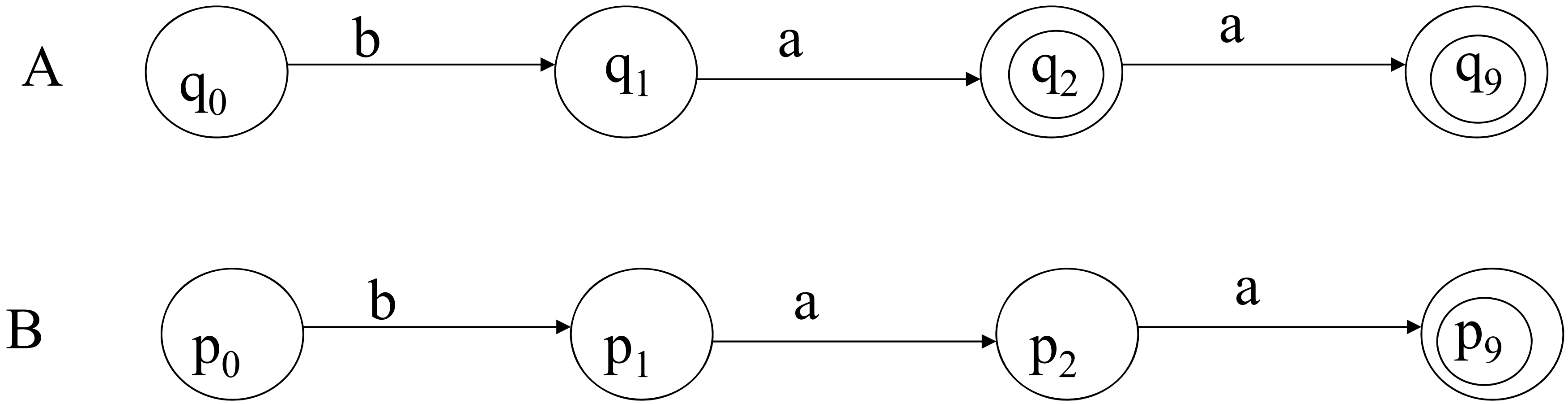
Closure properties of FA

- The mathematical notion of closure:
  - Natural numbers are closed under the sum operation
  - But not under subtraction
  - Integer numbers are closed under sum, subtraction, multiplication, but not …
  - Rational numbers …
  - Real numbers …
  - Closure (under operations and relations) is a very important notion

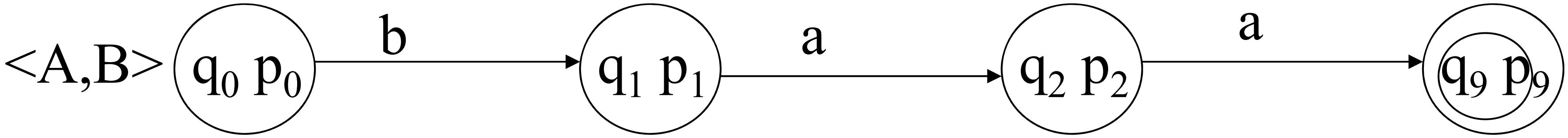
In the case of Languages:

- $\mathcal{L} = \{L_i\}$: a *family* of languages
- $\mathcal{L}$ is closed under OP if and only if (iff) for every $L_1, L_2 \in \mathcal{L}$, L_1 OP $L_2 \in \mathcal{L}$
- $\mathcal{R}$: *regular languages*, those accepted by an FA
- $\mathcal{R}$ is closed under set theoretic operations, concatenation, "*", … and virtually "all others"

Intersection



One can simulate the “parallel run” of A and B by simply “coupling them”:



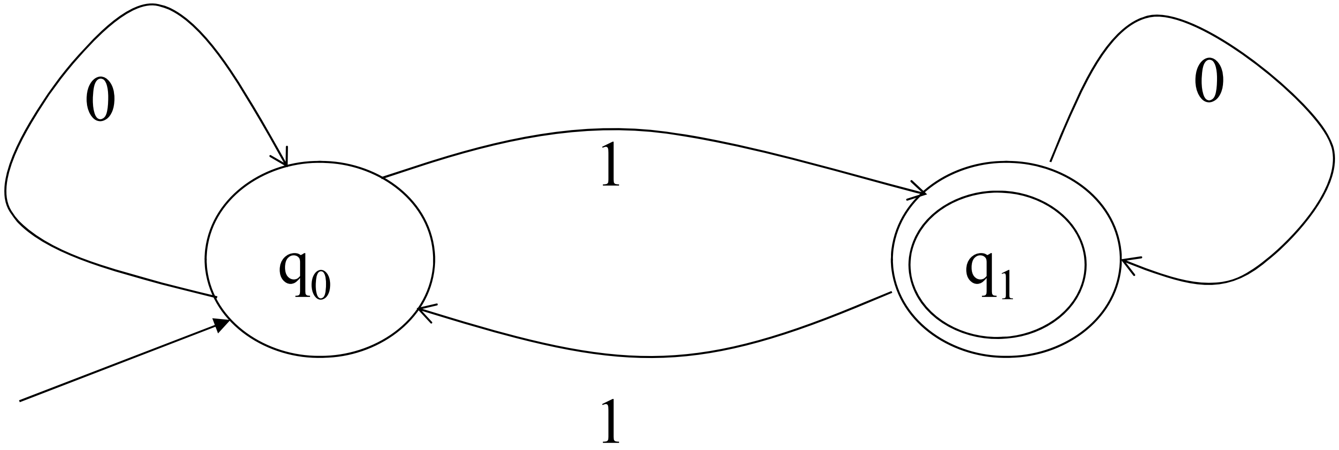
Formally:

- Given $A^1 = \langle Q^1, I, \delta^1, q_0{}^1, F^1\rangle$ and
  $A^2 = \langle Q^2, I, \delta^2, q_0{}^2, F^2\rangle$
- The automaton $\langle A^1, A^2 \rangle$ is defined as:
  $\langle Q^1 \times Q^2, I, \delta, \langle q_0{}^1 , q_0{}^2 \rangle, F^1 \times F^2 \rangle$
  $\delta(\langle q^1 , q^2 \rangle , i) = \langle \delta^1(q^1, i), \delta^2(q^2,i)\rangle$
- One can show by a simple induction that
  $L(\langle A^1, A^2 \rangle) = L(A^1) \cap L(A^2)$

Union

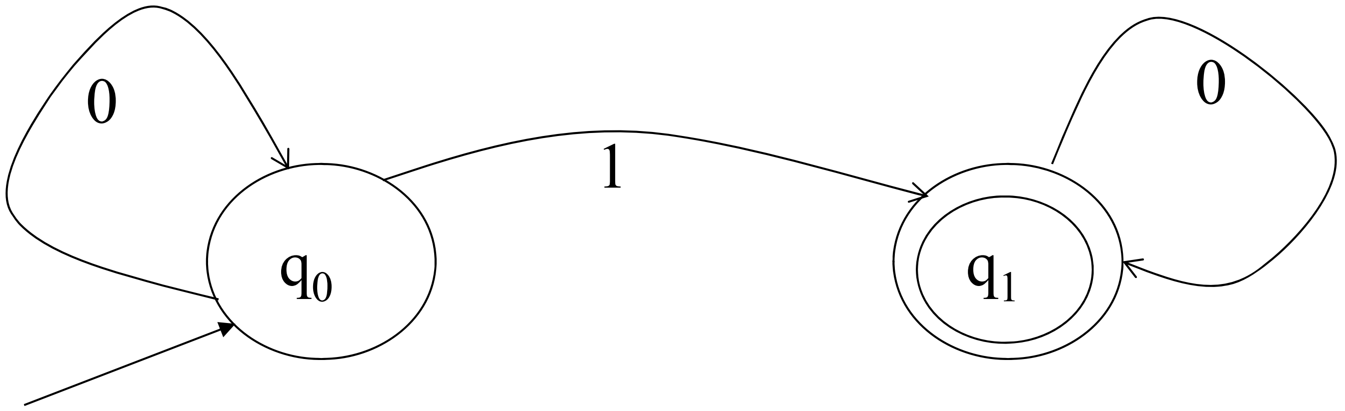
- A similar construction …
  otherwise ... exploit identity A ∪ B = ¬(¬A ∩ ¬B)
  ⇒ need a FA for the complement language

Complement: example automaton accepting
binary strings x with odd number of 1's



An idea: F^ = Q - F:

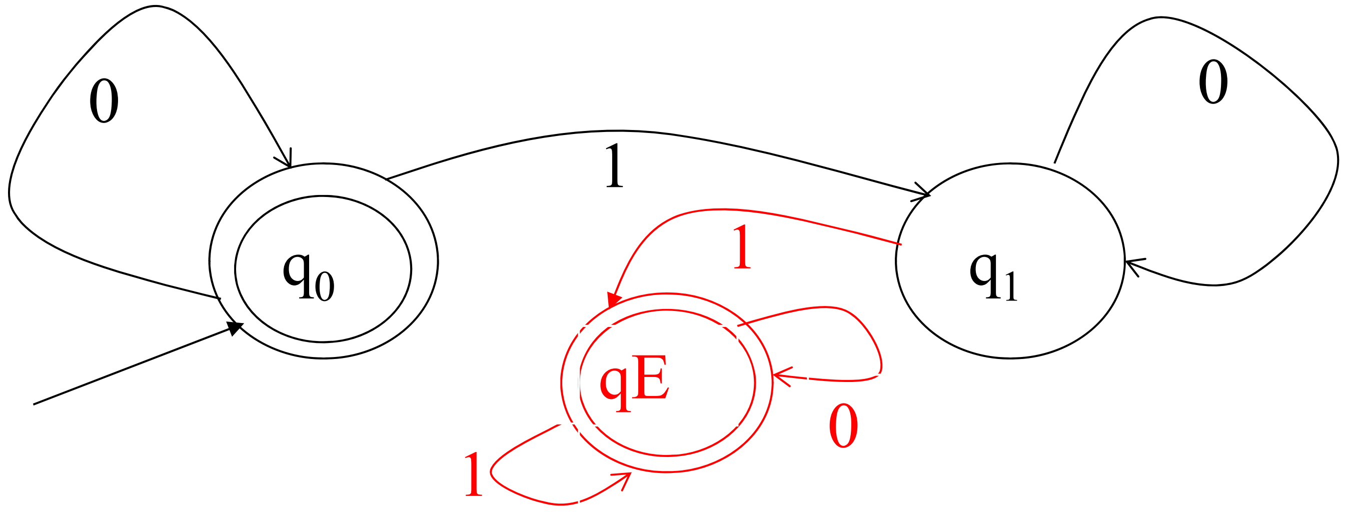
Yes, it works for the automaton above, but ….



L: strings with exactly one ‘1’

The “complement F construction”, i.e., turning F into Q – F, doesn’t work

Why? because δ is partial



¬L: strings with no ‘1’ or with more than one ‘1’

Some general remarks on the complement

- If the analysis of the string terminates for all possible strings then it suffices the “turn a yes into a no” (F into Q-F)
- If, for some string, the analysis of the string does not terminate (it gets blocked or continues forever) then turning F into Q-F does not work
- With FA the problem is easily solved …
- In general if we are unable to provide a positive answer to a problem (e.g., $\boldsymbol{x \in L}$), this does not necessarily mean we can provide a positive answer for the complement problem (e.g., $\boldsymbol{x \in \neg L}$)

Myhill – Nerode Theorem

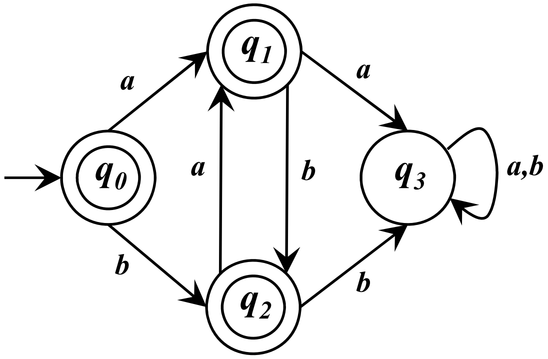
- What do the states a of a DFA $A = < Q, I, \delta, q_0, F >$ represent?
  - every state represents the **set of strings** that, input to A, lead to it
  - these strings “share a common destiny”:

    notice that, for every $\boldsymbol{x}, \boldsymbol{y} \in I^*$, if $\delta(q_0, x) = \delta(q_0, y)$ then

    $\forall \boldsymbol{z} \in \mathrm{I}^*$ $\delta(q_0, x\cdot z) = \delta(q_0, y\cdot z)$ hence $x\cdot z\in L(A)$ iff $y\cdot z\in L(A)$

    i.e., either both or none of $\boldsymbol{x\cdot z}$ and $\boldsymbol{y\cdot z}$ $\in L(A)$

- Example: automaton accepting
  - strings ∈{a, b}* s.t. no two adjacent symbols are equal



  - q_0 associated with {ε}, q_1 associated with {strings ending with a},
  - q_2 with {strings ending with b}, q_3 with {strings with two equal adjacent symbols}

- Apply this idea to a language $L \subseteq I^*$: define a relation $\approx_L$ among strings
  - $\boldsymbol{x}, \boldsymbol{y} \in \boldsymbol{L}$ are ***indistinguishable*** $\quad \boldsymbol{x} \approx_L \boldsymbol{y}$ iff

    $\forall\, \boldsymbol{z} \in \boldsymbol{I^*} \qquad \boldsymbol{x \cdot z} \in \boldsymbol{L} \quad \Leftrightarrow \quad \boldsymbol{y \cdot z} \in \boldsymbol{L}$
  - note that $\boldsymbol{\approx_L}$ is an equivalence relation (‘$\Leftrightarrow$’ is a sort of equality)
  - hence $\boldsymbol{I^*}$ is *partitioned* by $\approx_L$ into equivalence classes
- the cardinality of the equiv. classes of $\approx_L$ may be
  - **finite**. Example: $L = \{\, x \in \{a, b\}^* \mid$ no two adjacent symbols of $\boldsymbol{x}$ are equal $\}$
    - {ε}, {strings ending with *a*}, {strings ending with *b*}, {strings with two equal adjacent symbols}
  - **infinite**. Example: $L = \{\, a^n\, b^n \mid n \geq 0 \,\}$
    - {ε}, {*a*}, {*aa*}, {*aaa*}, {*aaaa*}, … {a^n}, … and others for strings containing *b*’s

- Myhill-Nerode theorem:

  a language *L* is accepted by a DFA

  iff $\approx_L$ has a finite number of equivalence classes

- Proof (sketch) of Myhill-Nerode theorem:

  L accepted by a DFA iff $\approx_L$ has a finite number of equivalence classes

- (1): $L=L(A)$ for a DFA $A \Rightarrow \approx_L$ has a finite number of equiv. classes

  immediate: make ***A*** complete if necessary (by adding an error sink state)

  then every $x \in I^*$ is in the equiv. class of some state of ***A***

- (2): $\approx_L$ finite equiv. classes $\Rightarrow L$ accepted by the DFA $A = (Q, I, \delta, q_0, F)$:
  - $q_0 = [\varepsilon]$ (NB: $[x]$ denotes the equiv. class that includes x)
  - $Q = \{ [x] \mid x \in I^* \}$
  - for every equiv. class $[x]$, $\forall a \in I \quad \delta([x], a) = [x \cdot a]$
    - NB: δ is *well defined*: $\delta([x], a)$ does not depend on the x chosen as a representative of the class, by the definition of $\approx_L$
  - $F = \{ [x] \mid x \in L \}$

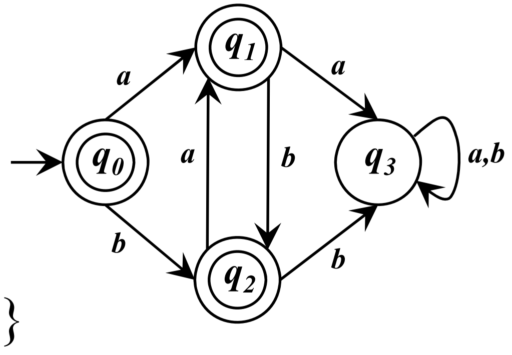
- Another (simplest) example: $L = \{ x \in \{a\}^* \mid |x| \text{ MOD } 3 = 1 \}$
  - Exercise: Identify the equiv. classes of $\approx_L$; Design the accepting automaton

“Applications” of the Myhill-Nerode theorem

- Prove (Disprove) that a language is accepted by a DFA
  - Based on the fact the $\approx_L$ has (does not have) a finite number of equiv. classes
- DFA minimization (find the minimal equivalent DFA)
  - …introducing the notion of ***indistinguishable states***

- “Annotate” the states of a DFA by the associated $\approx_L$ equiv. classes

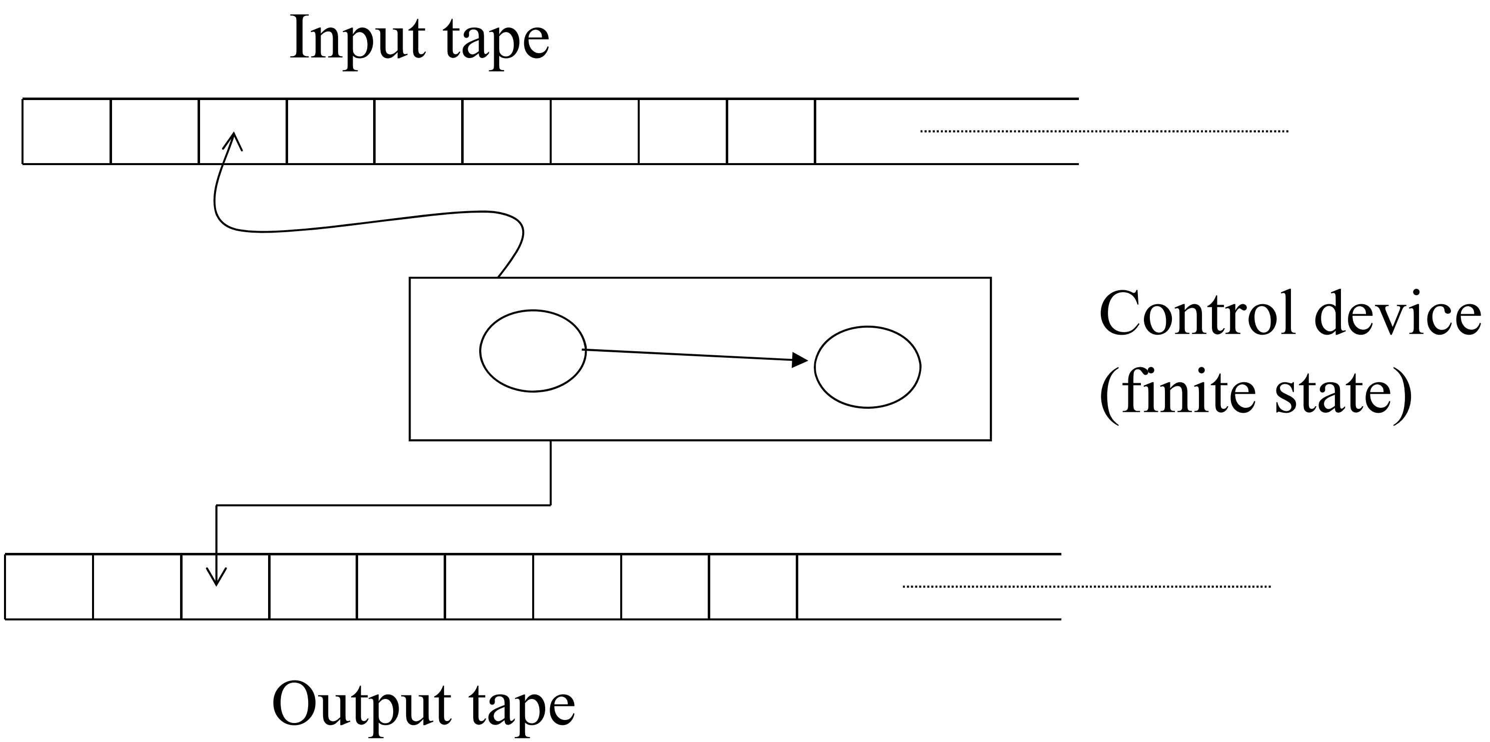
- Example $L = \{ x \in \{a, b\}^* \mid$ no two adjacent symbols of $\boldsymbol{x}$ are equal $\}$



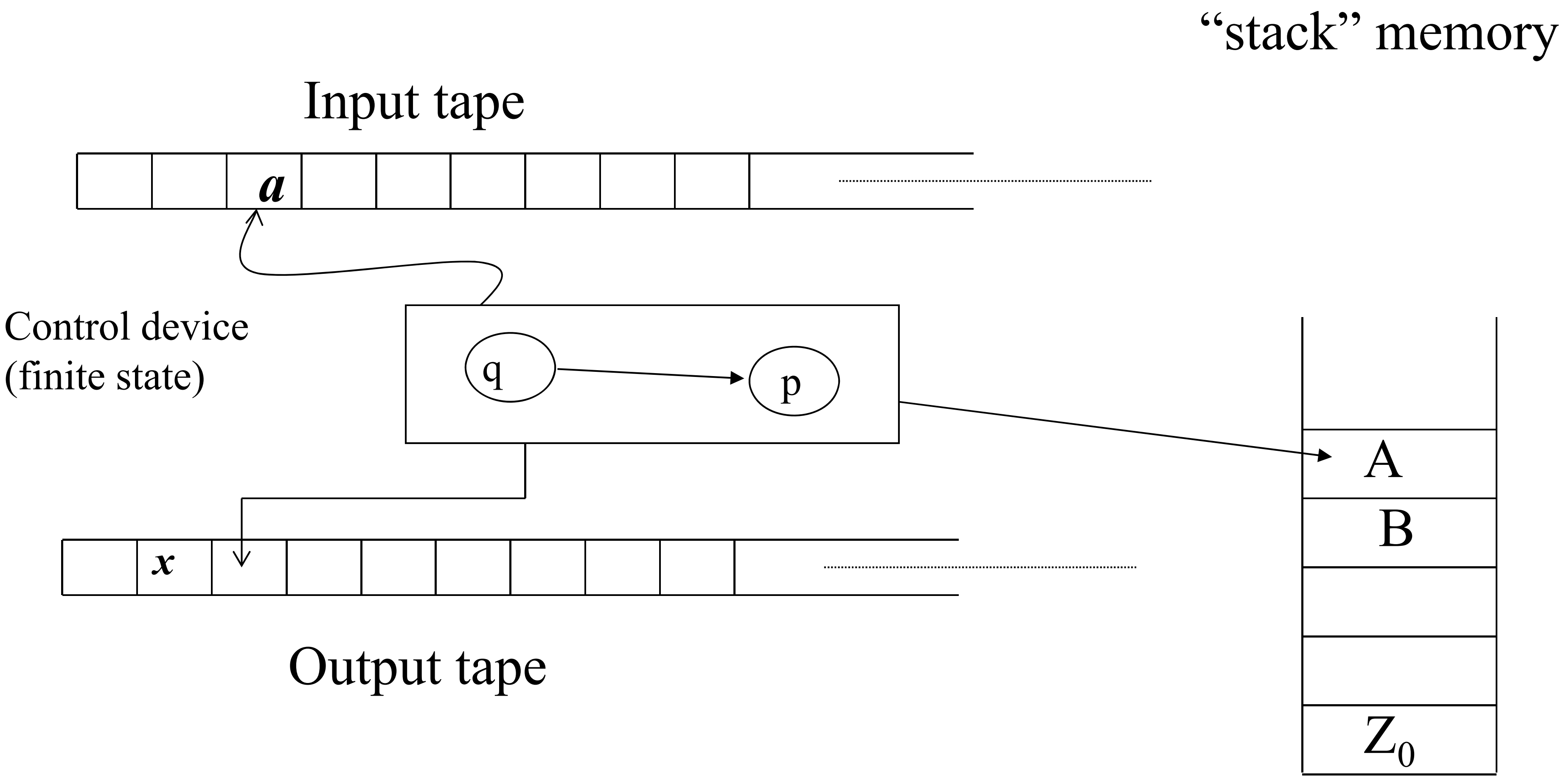
  - $q_0 \leftrightarrow [\varepsilon] = \{\varepsilon\}$,
  - $q_1 \leftrightarrow [a] = \{ x \mid \exists y \ \ x = y{\cdot}a \wedge$ “no aa nor bb in x” $\}$,
  - $q_2 \leftrightarrow [b] = \{ x \mid \exists y \ \ (x = y{\cdot}b) \wedge$ “no aa nor bb in x” $\}$,
  - $q_3 \leftrightarrow [aa] = \{ x \mid \exists y, z \in I^* (x = y{\cdot}aa{\cdot}z \ \vee \ x = y{\cdot}bb{\cdot}z) \}$

Let us increase the power of the FA by increasing its memory

- Consider a more "mechanical" view of the FA:



- Now let us “enrich it” :



The move of the pushdown automaton (with a **stack** memory):

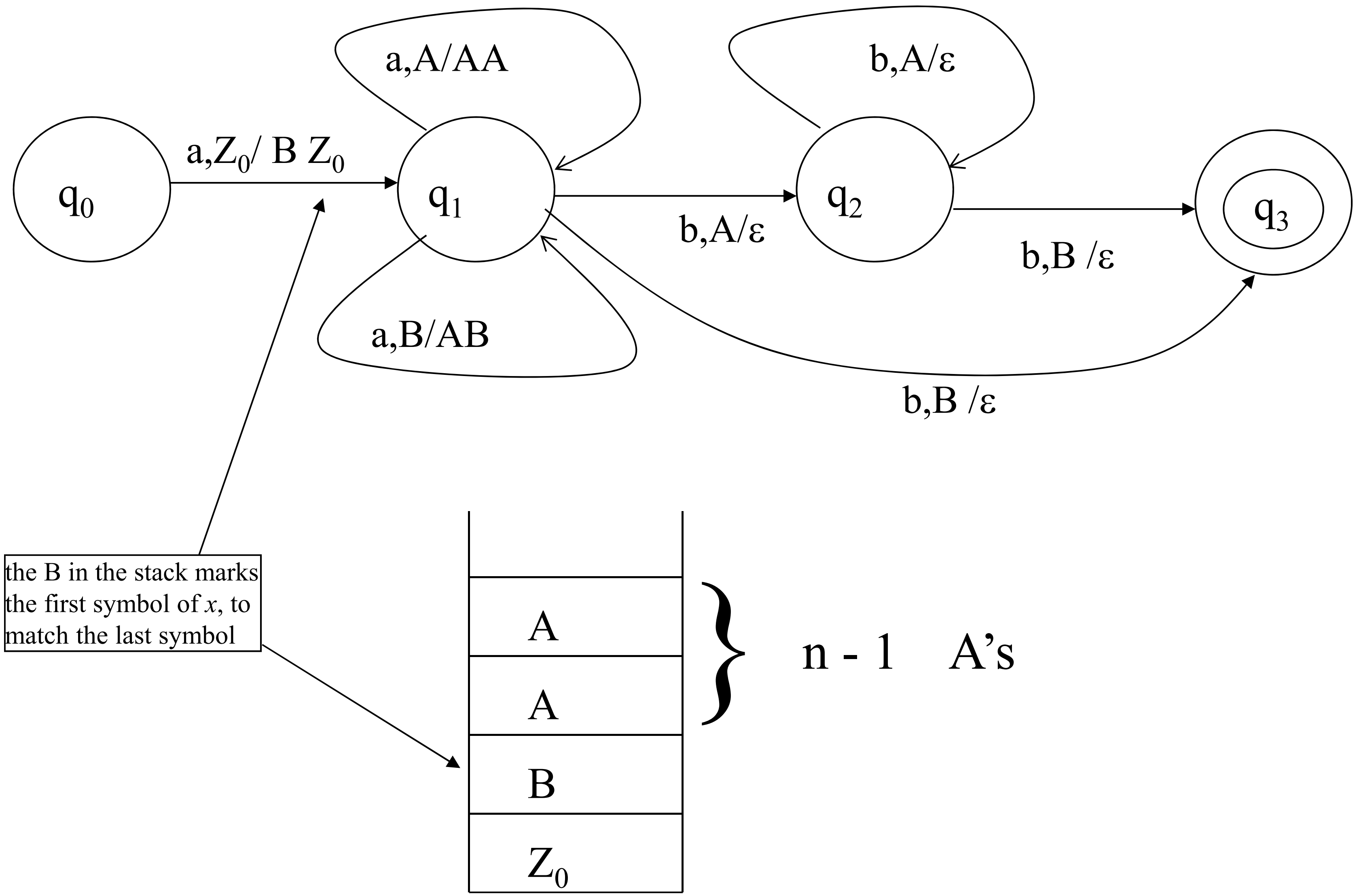
- Depending on :
  - the symbol read on the input tape (but it could also ignore the input …)
  - the symbol read on top of the stack
  - the state of the control device:
- the pushdown automaton
  - changes its state
  - moves ahead the scanning head (or it does not if input was ignored)
  - changes the symbol *A* read on top of the stack with a ***string*** α of symbols (α may be empty: this amounts to a ***pop*** of *A*)
  - (if translator) it writes a string (possibly empty) on the output tape (advancing the writing head consequently)

- The input string ***x*** is recognized (accepted) if
  - The automaton scans it completely (the scanning head reaches the end of ***x***), and
  - Upon reaching the end of ***x*** it is in an acceptance state (just like for the FA)
- If the automaton is also a translator, then **τ(*x*)** is the string on the output tape after ***x*** has been completely scanned (if x is accepted, otherwise $\tau(x)$ is undefined: **τ(*x*) =** $\perp$)

NB: $\perp$ is the "undefined" symbol:

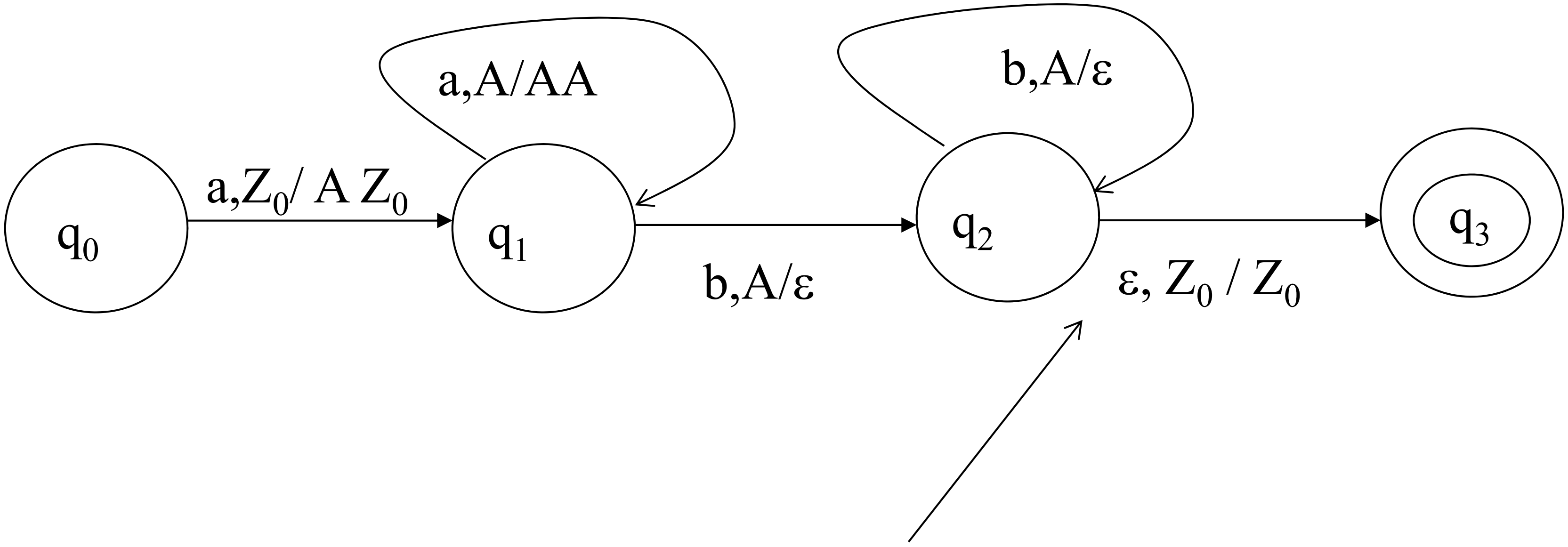
τ(*x*) = $\perp$ is just a shorthand for **$\neg\exists$*y* (τ(*x*) = *y*)**

A first example: accepting $\{a^n b^n \mid n > 0\}$



the B in the stack marks the first symbol of *x*, to match the last symbol

Another one:

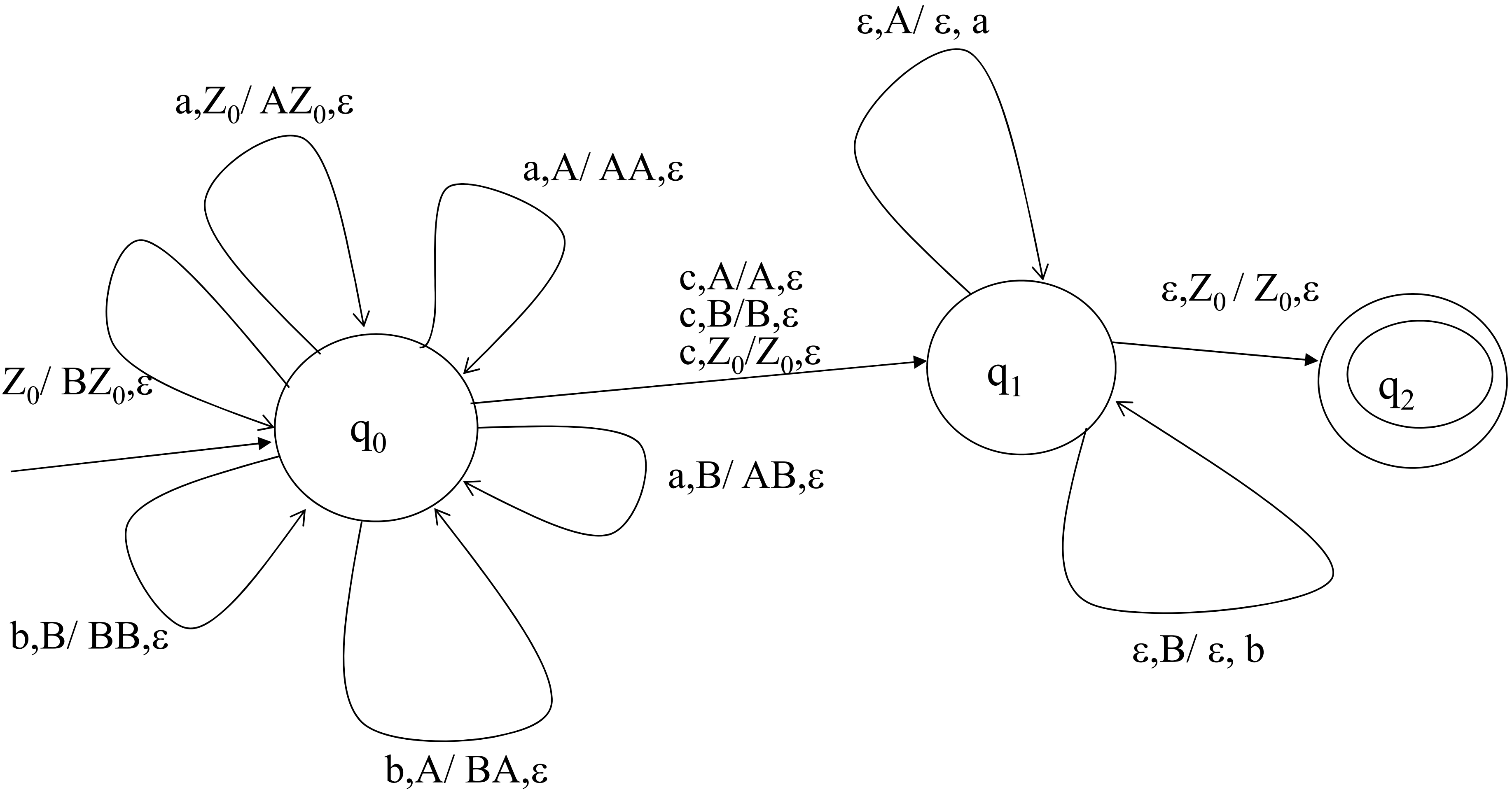


ε-move (AKA ***spontaneous*** move)

the ‘ε’ symbol is «overloaded»: used as an ***additional input symbol***

A (classical) pushdown automaton-translator

It reverses a string: $\tau(wc)=w^R$, $\forall w \in \{a,b\}^+$

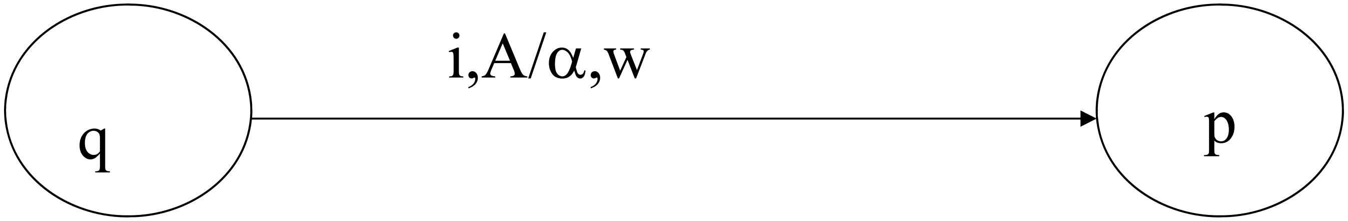


Now we formalize ...

- Pushdown automaton [transducer]: $<Q, I, \Gamma, \delta, q_0, Z_0, F [, O, \eta]>$
- Q, I, q_0, F [O] just like FA [FST]
- Γ stack alphabet (disjointed from other ones for ease of definition)
- Z_0 : initial stack symbol (not essential, useful to simplify definitions)
- $\delta: Q \times (I \cup \{\varepsilon\}) \times \Gamma \rightarrow Q \times \Gamma^*$ δ is partial just as in FSA
- $\eta: Q \times (I \cup \{\varepsilon\}) \times \Gamma \rightarrow O^*$ (η defined where δ is)

Graphical notation:

$<p,\alpha> = \delta(q,i, A)$
$w \qquad = \eta(q,i, A)$



- Configuration (a generalization of the notion of state): c = <q, x, γ, [z]>:
  - q: state of the control device
  - x: ***unread*** portion of the input string (the head is positioned on the first symbol of x)
  - γ : string of symbols in the stack (convention: <top-left, bottom-right>)
  - z: string written (up to now) on the output tape

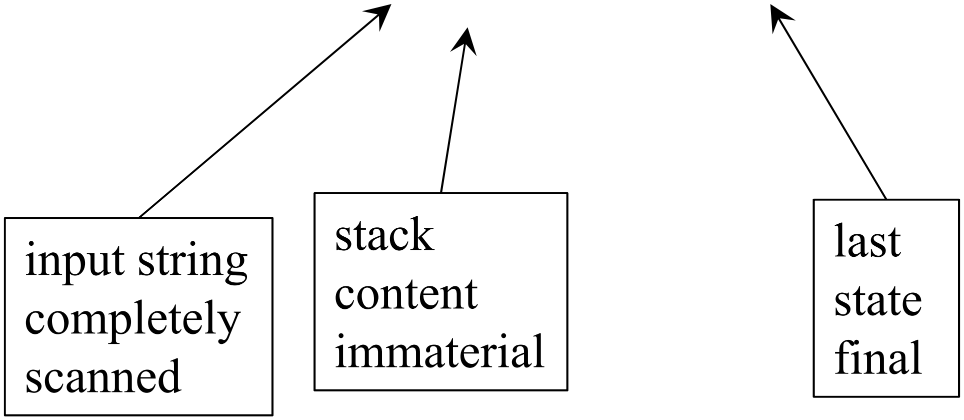
- ***Transition relation* '|--'** among configurations:

  c = <q, x, γ, [z]> |-- c' = <q', x', γ', [z']>

  - γ = Aβ
  - **Case 1**: x = i.y and δ(q, i, A) = <q', α> (**ordinary, "input-consuming" move**) [η(q,i, A) = w]
  - x' = y
  - γ'= αβ
  - [z' = z.w]
  - **Case 2**: δ(q, ε, A) = <q', α> (**spontaneous, ε move**) [η(q, ε, A) = w]
  - x' = x
  - γ'= αβ
  - [z' = z.w]
- NB: ∀q,A,i (δ(q, i, A) ≠ ⊥ ⇒ δ(q, ε, A) = ⊥) (hence δ(q, ε, A) ≠ ⊥ ⇒ δ(q, i, A) = ⊥)
  - i.e., ε-moves are alternative to all input-consuming ones
- Otherwise … nondeterminism!

- Acceptance [and translation] of a string
- |-*- : reflexive, transitive closure of the relation |--
  - i.e., |-*- denotes a number ≥0 of “steps” of the relation |--
- $x \in L\ [z = \tau(x)] \leftrightarrow$

$$c_0 = <q_0, x, Z_0, [\varepsilon]> \ |\text{-*-}\ c_F = <q, \varepsilon, \gamma, [z]>, q \in F$$



input string completely scanned

stack content immaterial

last state final

Pay attention to ε-moves, especially at the end of the string!

Pushdown automata in practice

- They are the heart of compilers
- Stack memory (LIFO) suitable to analyze nested syntactic structures (arithmetical expressions, compound instructions, …)
- Abstract run-time machine for programming languages with recursion
- ….

  Occur very frequently in the course of Languages and translators

Properties of pushdown automata (especially as acceptors)

- $\{a^nb^n \mid n > 0\}$ is accepted by a pushdown automaton (not by a FA)
  - However $\{a^nb^nc^n \mid n > 0\}$ ….
  - NOT: after counting –using the stack- *n a*'s and "de-counting" *n b*'s how can we remember *n* to count the *c*'s?
    The stack is ***not*** a read-only memory: to read it, one must destroy it!
    This limitation of the pushdown automaton can be proved formally through a generalization of the pumping lemma.
- $\{a^nb^n \mid n > 0\}$ accepted by a pushdown automaton;
  $\{a^nb^{2n} \mid n > 0\}$ accepted by a pushdown automaton
- However $\{a^nb^n \mid n > 0\} \cup \{a^nb^{2n} \mid n > 0\}$ …
  - Reasoning intuitively similar to the previous one:
  - If I empty all the stack with *n b*'s then I am unable to count other *b*'s
  - If I empty only half the stack and I do not find any more *b* I cannot know if I am halfway in the stack
  - The formalization of this reasoning is however not trivial ….

Some consequences

- $\mathcal{LP}$ = class languages accepted by pushdown automata
- $\mathcal{LP}$ is not closed under union nor intersection
- Why?
  - [consider the languages $\{a^n b^n c^n \mid n > 0\}$ and $\{a^n b^n \mid n > 0\} \cup \{a^n b^{2n} \mid n > 0\}$]
- Considering the complement …
  The same principle as with FA: change the accepting states into non accepting states.
  There are however new difficulties

- Function δ must be made complete (as with FA) with an error state. Pay attention to the nondeterminism caused by ε-moves!
- The ε-moves can cause cycles ⇒ never reach the end of the string ⇒ the string is not accepted, but it is not accepted either by the automaton with F^ = Q-F.
- There exists however a construction that associates to every automaton an equivalent loop-free automaton
- Not finished yet: what if there is a sequence of ε-moves at the end of the scanning with some states in F and other ones not in F?

- $<q_1, \varepsilon, \gamma_1>$ |-- $<q_2, \varepsilon, \gamma_2>$ |-- $<q_3, \varepsilon, \gamma_3>$ |-- …
  $q_1 \in F$, $q_2 \notin F$, …. ?
- Then we must "force" the automaton to accept only at the end of a (necessarily finite, as the automaton is loop-free) sequence of ε-moves.
- This is also possible through a suitable construction.

Once more, rather than the technicalities of the construction/proof we are interested in the general mechanism to accept the complement of a language: sometimes the same machine that solves the "positive instance" of the problem can be adapted to solve the "negative instance": this can be trivial or difficult: we must be sure to be able to complete the construction

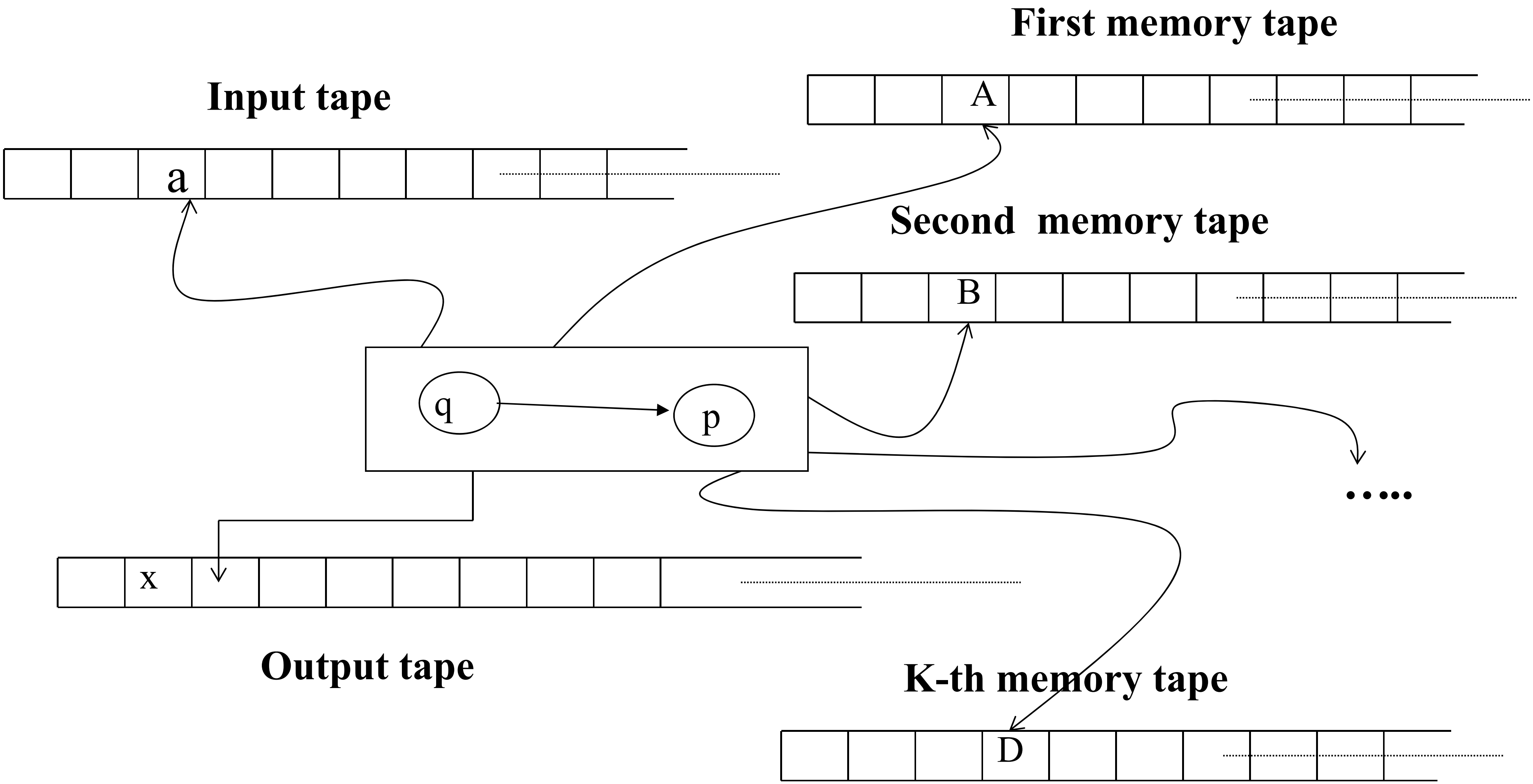
Pushdown automata [as acceptors (PDA) or translators (PDT)] are more powerful than finite state ones (a FA is a trivial special case of a PDA; more, PDA have an unlimited counting ability that FA lack)
However also PDA/PDT have their limitations …
… a new and "last" (for us) automaton:

the ***Turing Machine*** (TM)
Historical model of a "computer", simple and conceptually important under many aspects.

We consider it as an automaton; then we will derive from it some important, universal properties of automatic computation.

For now we consider the "K-tape" version, slightly different from the (even simpler) original model. This choice will be explained later.

k-tape TM


First memory tape
Input tape
A
a
Second memory tape
B
q
p
…..
x
Output tape
K-th memory tape
D

Informal Description and Partial Formalization of the TM

- States and alphabets as with other automata (input, output, control device, memory alphabet)
- For historical reasons and due to some “mathematical technicalities” the tapes are represented as *infinite* cell sequences [0,1,2, …] rather than finite strings. There exists however a special symbol “blank” (“ ”, or “barred b” or “_”) and it is assumed that every tape contains only a finite number of non-blank cells.
  - The equivalence of the two ways of representing the tape content is obvious.
- Scanning and output heads are also as in previous models

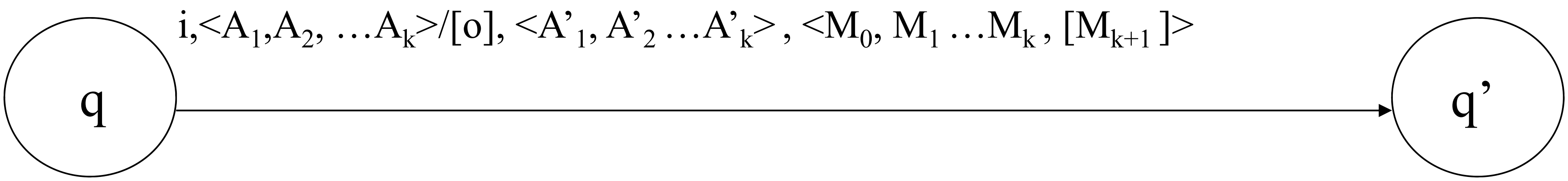
- The move of the TM:
- Reading:
  - one symbol on the input tape
  - k symbols on the k memory tapes (one for each tape)
  - state of the control device
- Action:
  - State change: q ----> q’
  - Write a symbol in place of the one read on each of the k memory tapes: A_i ----> A_i’, 1 <= i <= k
  - [Write a symbol on the output tape]
  - Move of the k + 2 heads:
    - memory and scanning heads can move one position right (R) or left (L) or stand still (S)
    - The output head can move one position right (R) or stand still (S)

As a consequence:

$$<\delta,[\eta]>: Q \times I \times \Gamma^k \to Q \times \Gamma^k \times \{R,L,S\}^{k+1}\ [\times O \times \{R,S\}]$$

(partial!)

Graphical notation:



i,<A_1,A_2, …A_k>/[o], <A'_1, A'_2 …A'_k> , <M_0, M_1 …M_k , [M_{k+1}]>

$\forall i \in [0..k]$ $M_i \in \{R,L,S\}$ $M_{k+1} \in \{R,S\}$

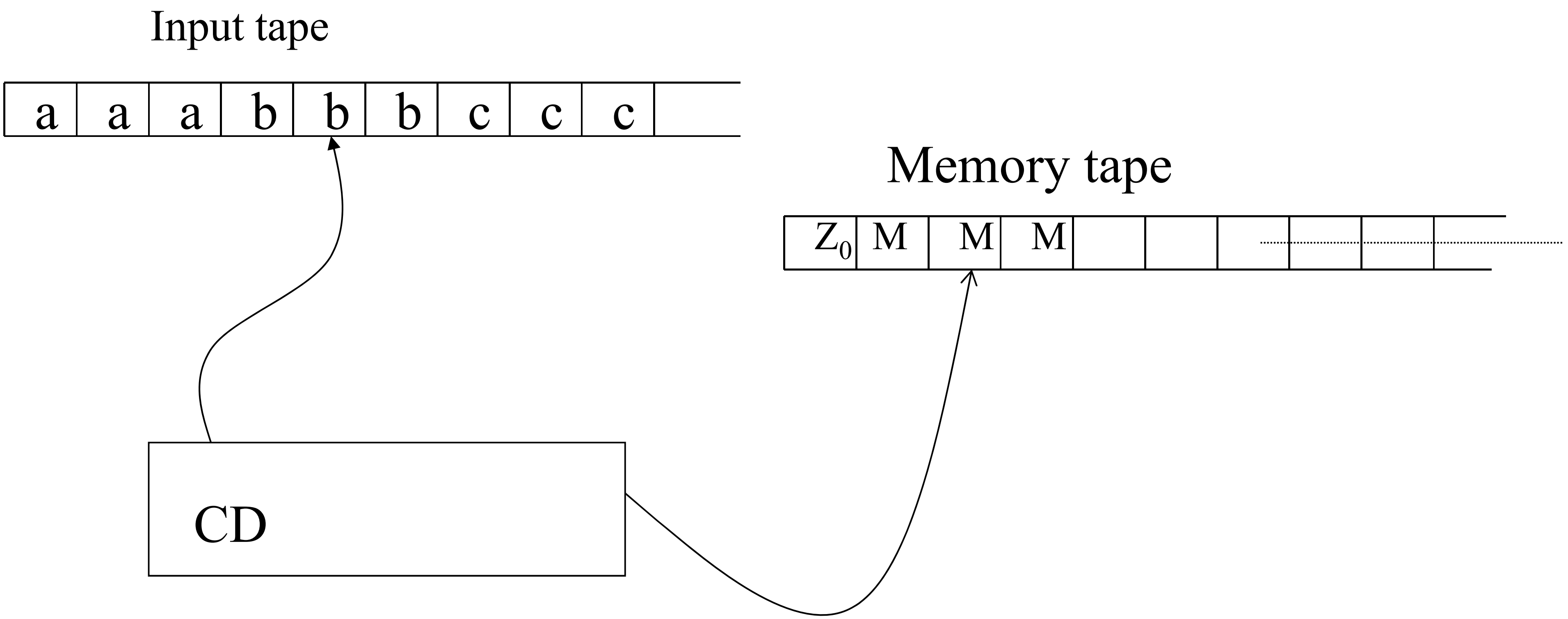
Why do we not loose generality having O rather than O^* for the output?

- Initial configuration:
  - Z_0 followed by all blanks in the memory tapes
  - [output tape all blank]
  - Heads in the 0-th position on every tape
  - Initial state of the control device q_0
  - Input string x starting from the 0-th cell of the input tape, followed by all blanks

- Final configurations:
  - Accepting states $F \subseteq Q$
  - For ease of notation, convention:
    $<\delta,[\eta]> (q, \ldots) = \perp \ \forall\, q \in F$: no further action from a final state
  - The TM stops when $<\delta,[\eta]> (q, \ldots) = \perp$
  - Input string x is accepted iff:
    - After a ***finite*** number of moves the TM stops (hence it is in a configuration where $<\delta,[\eta]> (q, \ldots) = \perp$)
    - When it stops, its state $q \in F$
- NB: as a consequence
  - x is *not* accepted if:
    - The TM stops in a state $q \notin F$; or
    - The TM never stops (NB: this case is very important)
  - There is a similarity with the PDA (a non-loop-free PDA might also not accept because of a “non stopping run”), however … does there exist a loop-free TM?

Some examples

- A TM accepting $\{a^n b^n c^n \mid n > 0\}$



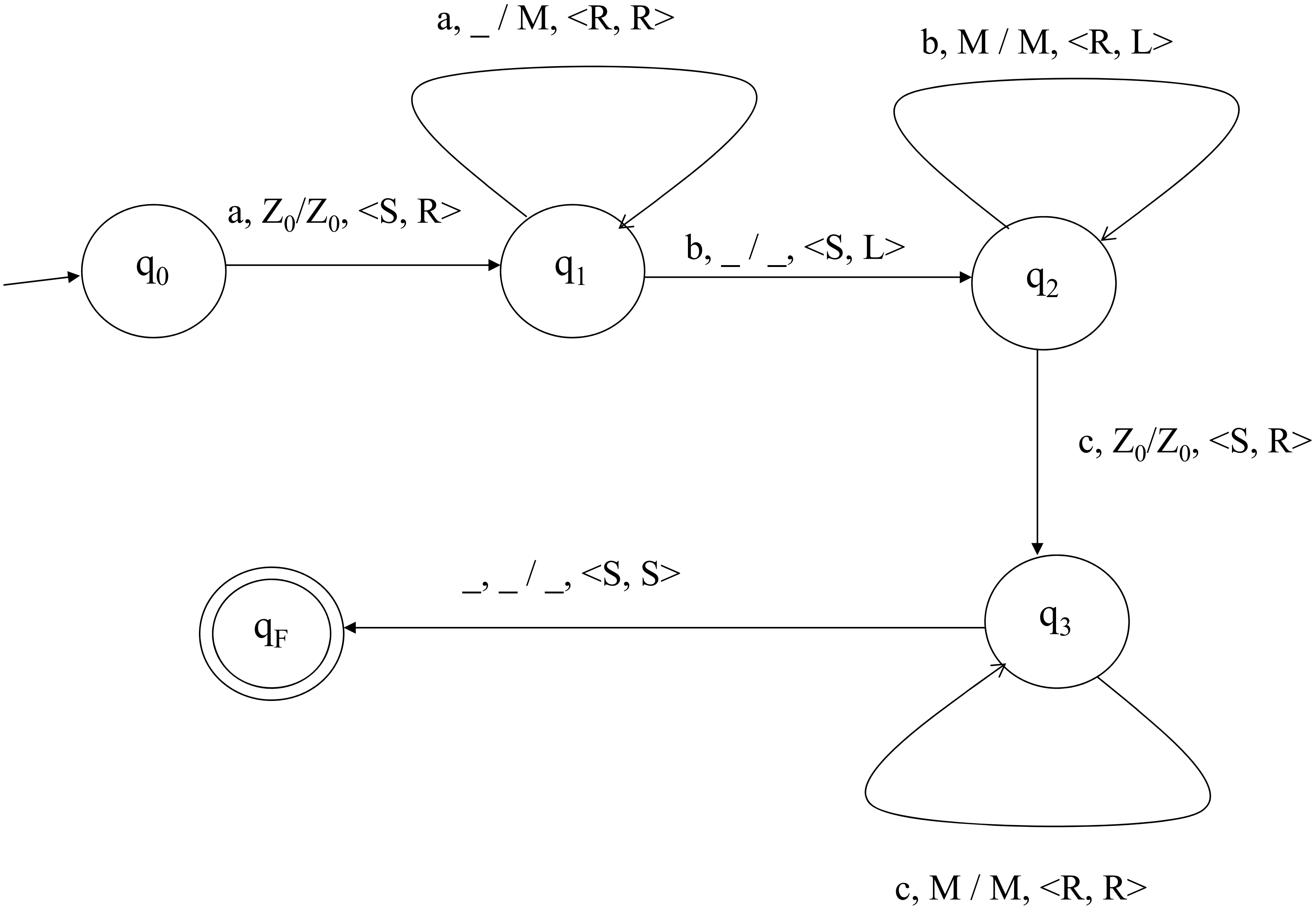
Input tape

a a a b b b c c c

Memory tape

Z_0 M M M

CD

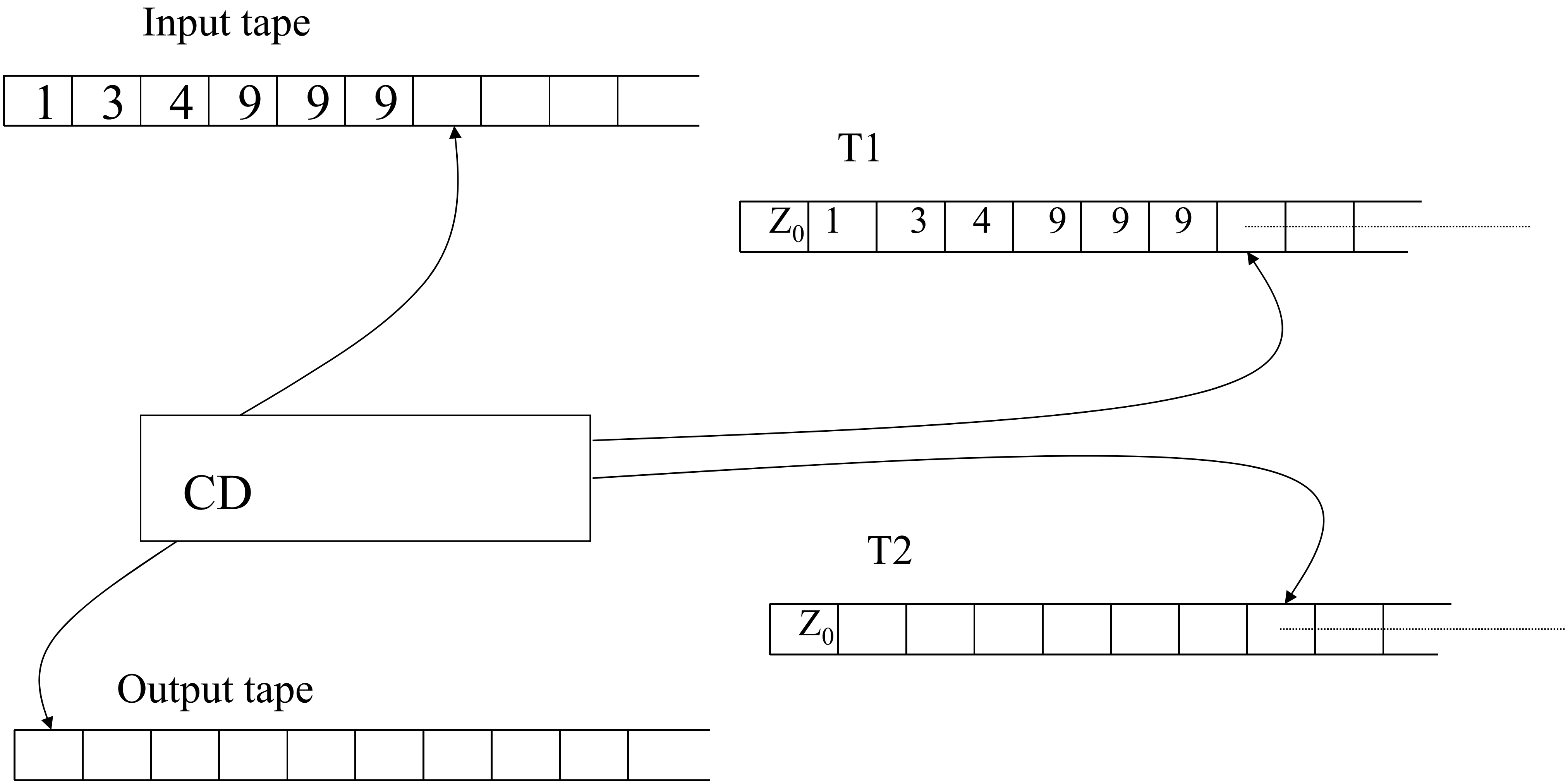

a, _ / M, <R, R>
b, M / M, <R, L>
a, Z_0/Z_0, <S, R>
q_0
q_1
b, _ / _, <S, L>
q_2
c, Z_0/Z_0, <S, R>
_, _ / _, <S, S>
q_F
q_3
c, M / M, <R, R>

Computing the successor of a number *n* coded with decimal digits two memory tapes T_1 and T_2

- M copies all digits of *n* on T_1, to the right of Z_0, while it moves head T_2 by the same number of positions.
- M scans the digits of T_1 from right to left. It writes on T_2 from right to left changing the digits as needed (9's become 0's, first digit≠9 becomes the successive digit, then all other ones are unchanged, …)
- M copies T_2 on the output tape.

- Notation: ***d** : any decimal digit*
- **_ : *blank***
- **# : *any digit*** ≠ 9
- **^ : *successor of a digit denoted as # (in the same transition)***

Purpose of the example: to show that the TM can compute ***any function***



Input tape

1 3 4 9 9 9

T1

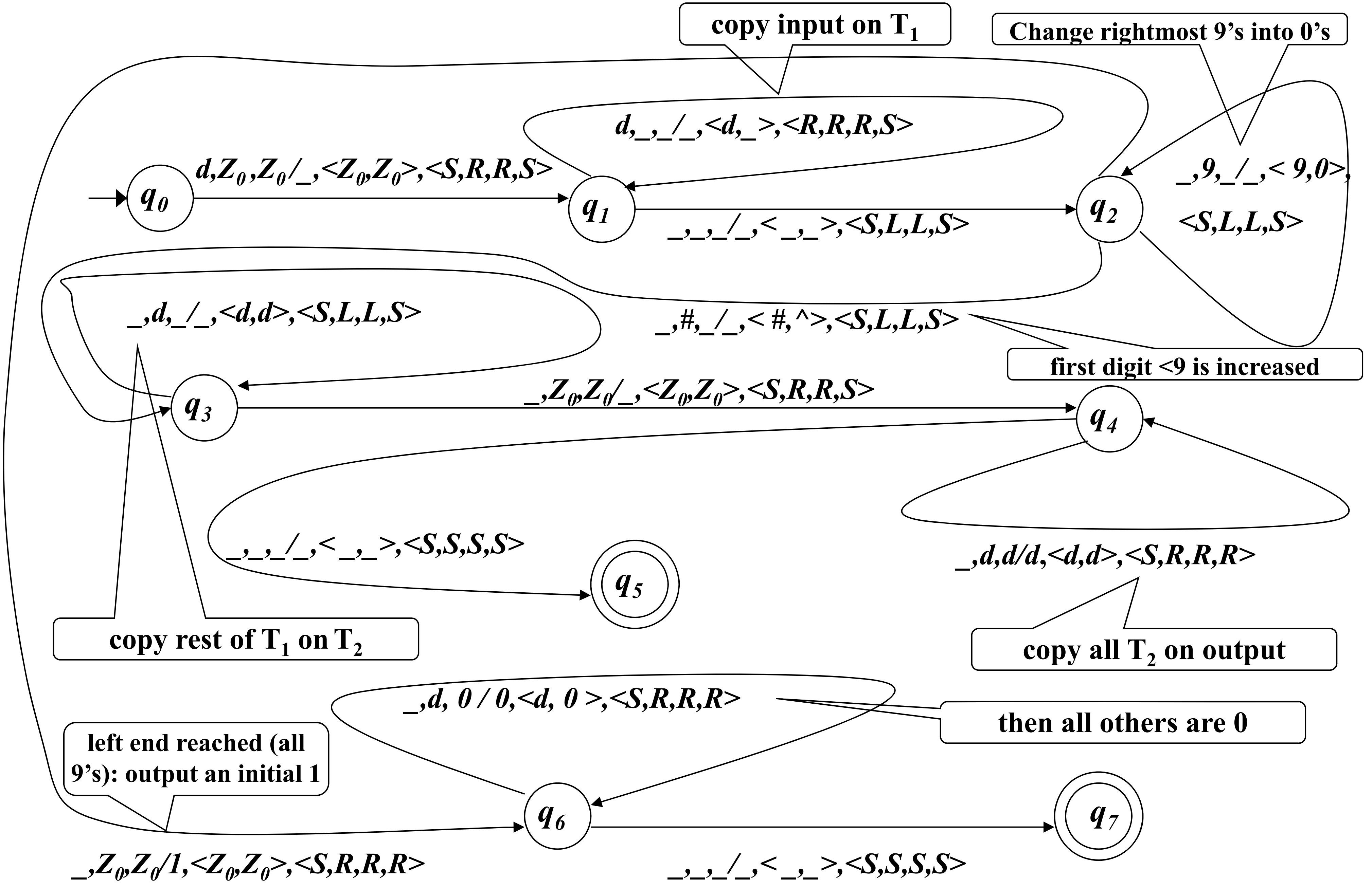
Z_0 1 3 4 9 9 9

CD

T2

Z_0

Output tape


q_0
q_1
q_2
q_3
q_4
q_5
q_6
q_7
$d,Z_0,Z_0/\_,<Z_0,Z_0>,<S,R,R,S>$
$d,\_,\_/\_,<d,\_>,<R,R,R,S>$
$\_,\_,\_/\_,<\_,\_>,<S,L,L,S>$
$\_,9,\_/\_,<9,0>,$
$<S,L,L,S>$
$\_,\#,\_/\_,<\#,\hat{}>,<S,L,L,S>$
$\_,d,\_/\_,<d,d>,<S,L,L,S>$
$\_,Z_0,Z_0/\_,<Z_0,Z_0>,<S,R,R,S>$
$\_,\_,\_/\_,<\_,\_>,<S,S,S,S>$
$\_,d,d/d,<d,d>,<S,R,R,R>$
$\_,d,0/0,<d,0>,<S,R,R,R>$
$\_,Z_0,Z_0/1,<Z_0,Z_0>,<S,R,R,R>$
$\_,\_,\_/\_,<\_,\_>,<S,S,S,S>$

copy input on T_1

Change rightmost 9’s into 0’s

first digit <9 is increased

copy rest of T_1 on T_2

copy all T_2 on output

then all others are 0

left end reached (all 9’s): output an initial 1

Closure properties of TM

- $\cap$: OK (a TM can easily simulate two other ones, both “in series” and “in parallel”)
- $\cup$: OK (idem)
- Idem for other operations (concatenation, *, ….)

- What about the complement?

  Negative answer! (Proof later on)
  If there existed loop-free TM’s, it would be easy: it would suffice to define the set of halting states (it is easy to make it disjoint from the set of non-halting states) and partition it into accepting and non-accepting states.

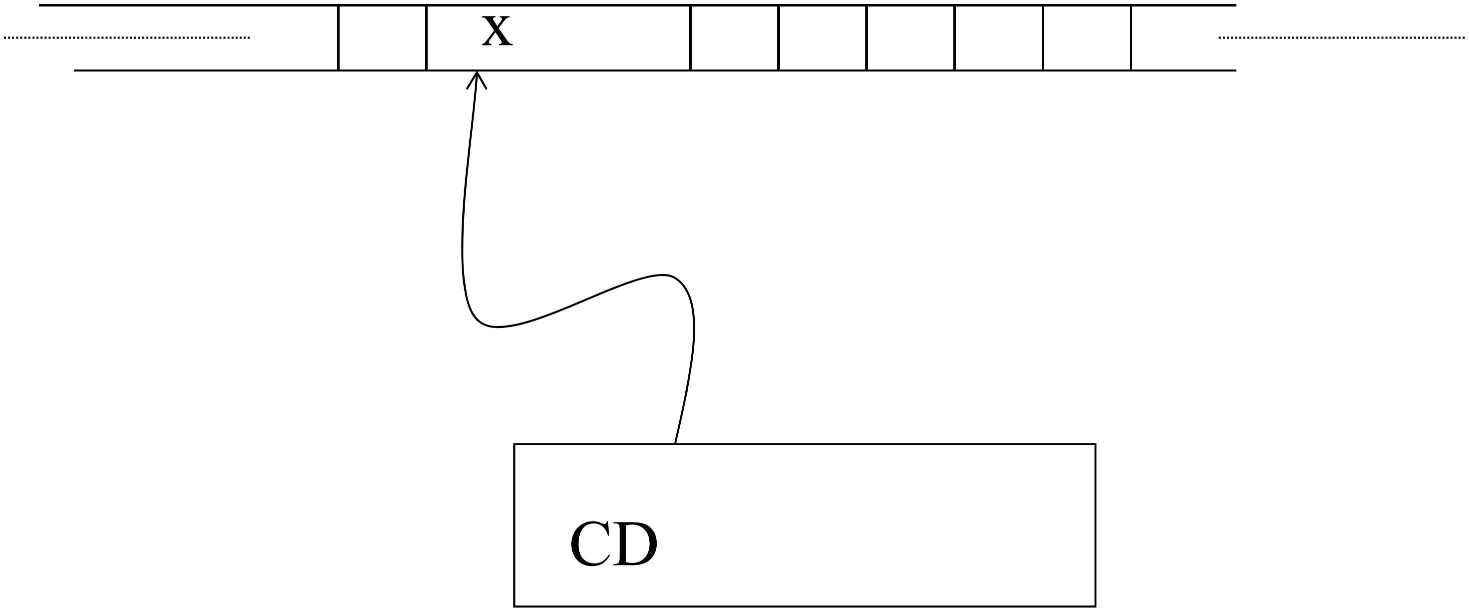
  ===========>

  It is therefore apparent that the problem arises from ***nonterminating computations***

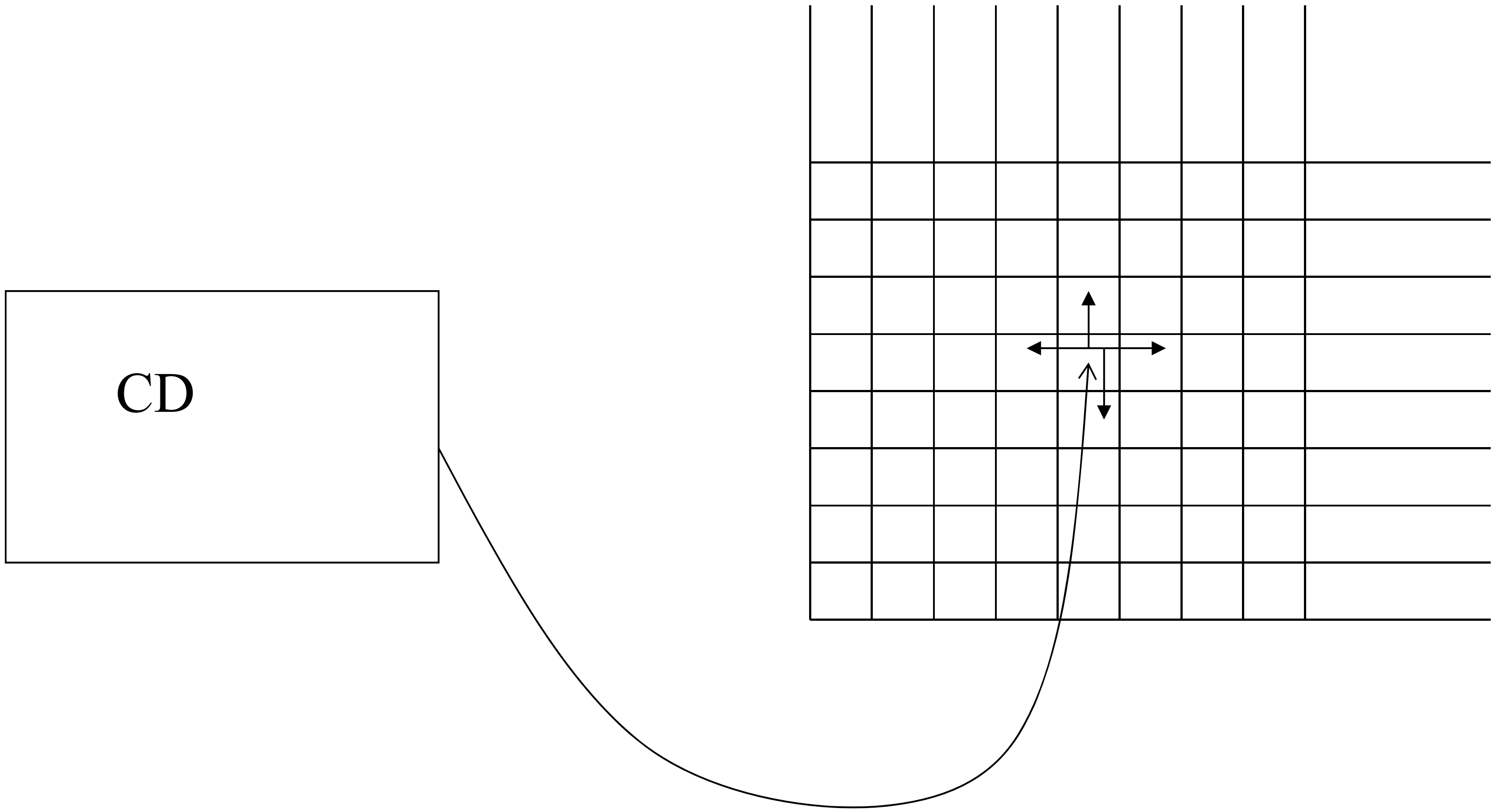
Equivalent TM models

- Single tape TM (NB: ≠ TM with 1 memory tape)

A single tape (usually unlimited in both directions): serves as input, memory, and output

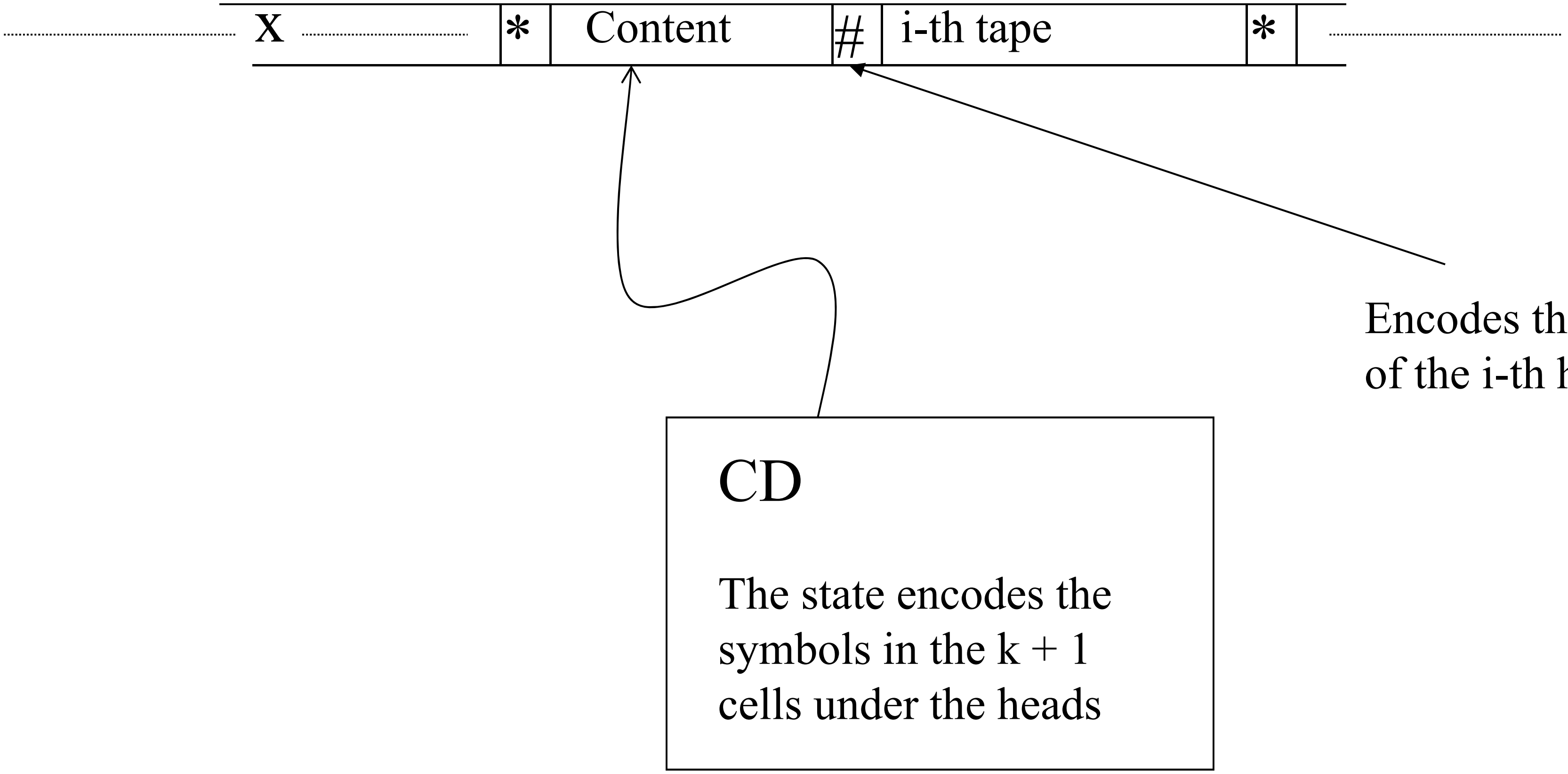


- Bidimensional tape TM



- TM with *k* heads for each tape
- …..

All versions of the TM are equivalent, w.r.t. their accepting or translating ability, for instance:



Encodes the position of the i-th head

CD

The state encodes the symbols in the k + 1 cells under the heads

What relations exist among the various automata (TM in particular) and more traditional/realistic computing models?

- The TM can simulate a Von Neumann machine (which is also "abstract")
- The main difference is in the way the memory is accessed: sequential rather than "random" (direct)
- This does not influence the machine for what concerns computational power (i.e., the class of problems it can solve)
- There can be a (profound) impact for what concerns the *complexity* of the computations, i.e., the time and memory needed
- We will consider the implications in both cases

Nondeterministic (operational) models

- Usually one thinks of an algorithm as a *uniquely determined* sequence of operations: in a certain configuration there is no doubt on what the next “step” will be
- Are we sure that this is desirable?

Let us compare

```
if x > y then max := x else max := y
```

with

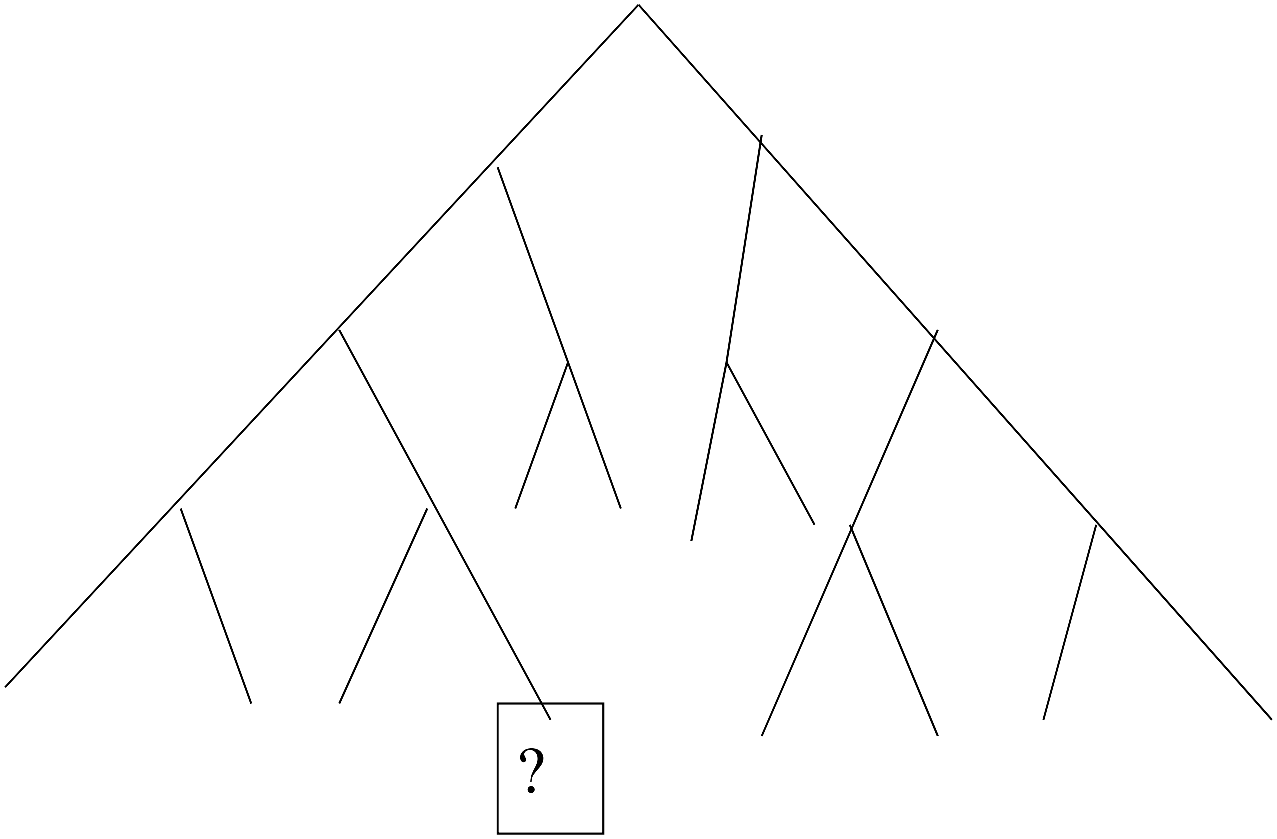
```
if      x >= y then max := x
        y >= x then max := y
fi
```

- Is it only a matter of elegance?

- Let us consider the **case** construct of Pascal & others:
  why not having something like the following?

```
case
  –  x = y        then S1
  –  z > y +3     then S2
  –  ….           then …
endcase
```

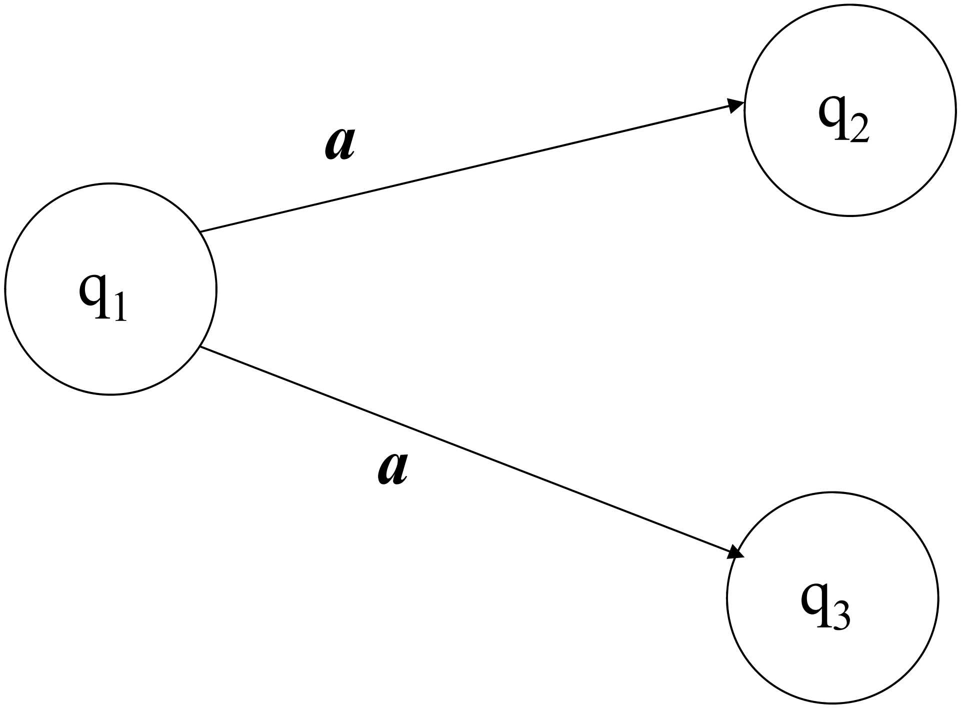
Another form of nondeterminism which is usually "hidden": blind search



- In fact, the search algorithms are a “simulation” of “basically nondeterministic” algorithms:
- Is the searched element in the root of the tree?
- If yes, OK. Otherwise
  - Search the left subtree
    or
  - Search the right subtree
- Choice of priority among various paths is often arbitrary
- If we were able to assign two tasks in parallel to two distinct machines ---->
- Nondeterminism as a model of computation or at least a model of design of parallel computing
  (For instance Ada and other concurrent languages exploit nondeterminism)

Among the numerous nondeterministic (ND) models: ND version of known models

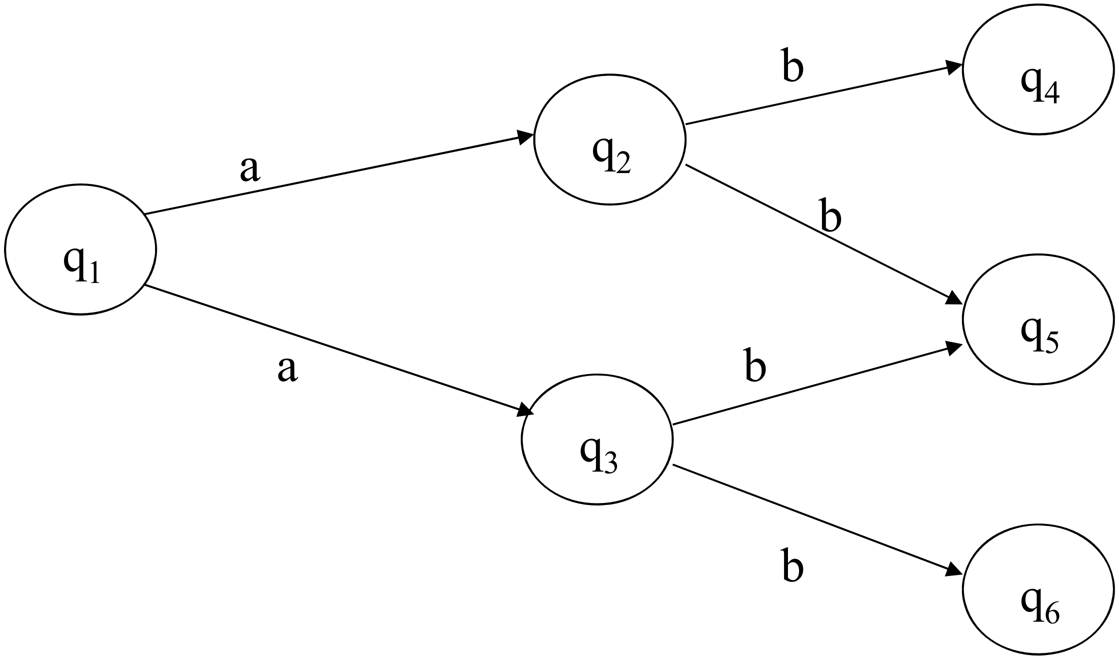
- ND FA (we will soon see how handy it is)



Formally: $\boldsymbol{\delta(q_1,a) = \{q_2, q_3\}}$

$\delta : Q \times I \rightarrow \mathcal{P}(Q)$

δ^* : formalization of a move sequence



$\delta(q_1,a) = \{q_2, q_3\}$, $\delta(q_2,b) = \{q_4, q_5\}$, $\delta(q_3,b) = \{q_6, q_5\}$

$\delta^*(q_1,ab) = \{q_4, q_5 , q_6\}$

$$\delta^*(q,\varepsilon) = \{q\}$$

$$\delta^*(q, y.i) = \bigcup_{q' \in \delta^*(q,y)} \delta(q', i)$$

How does a ND FA accept?

$$x \in L \quad \leftrightarrow \quad \delta^*(q_0, x) \cap F \neq \varnothing$$

Among the various possible runs (with the same input) of the ND FA ***it suffices that one of them*** (that is, ***there exists one that***) succeeds and accepts the input string

Another, alternative interpretation of nondeterminism:

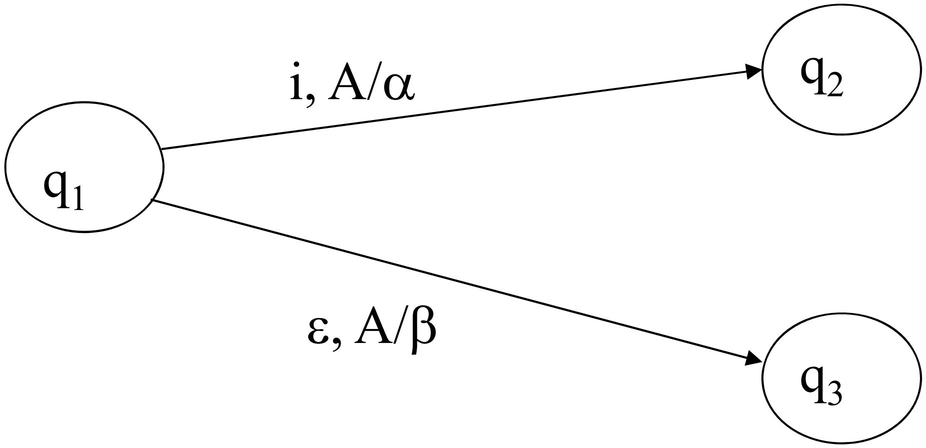
universal nondeterminism (the previous one is ***existential***):

all runs of the automaton accept

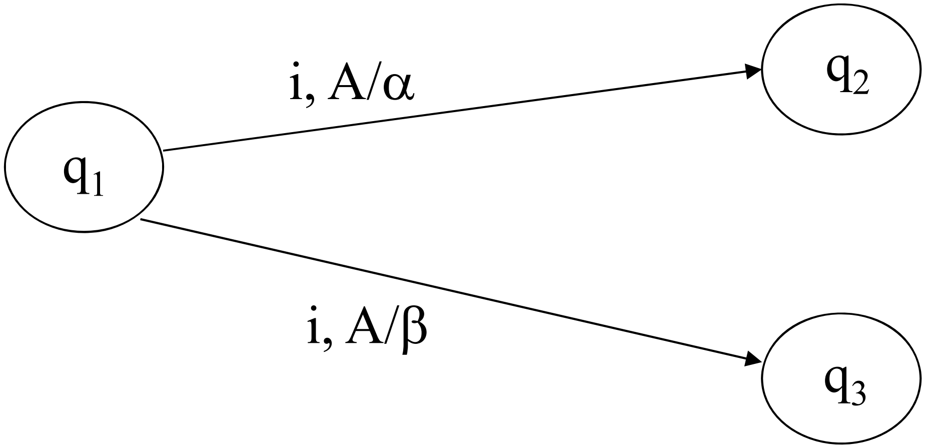
$$(\delta^*(q_0, x) \subseteq F)$$

nondeterministic PDA (NPDA)

- In fact PDA are “natural born” ND :



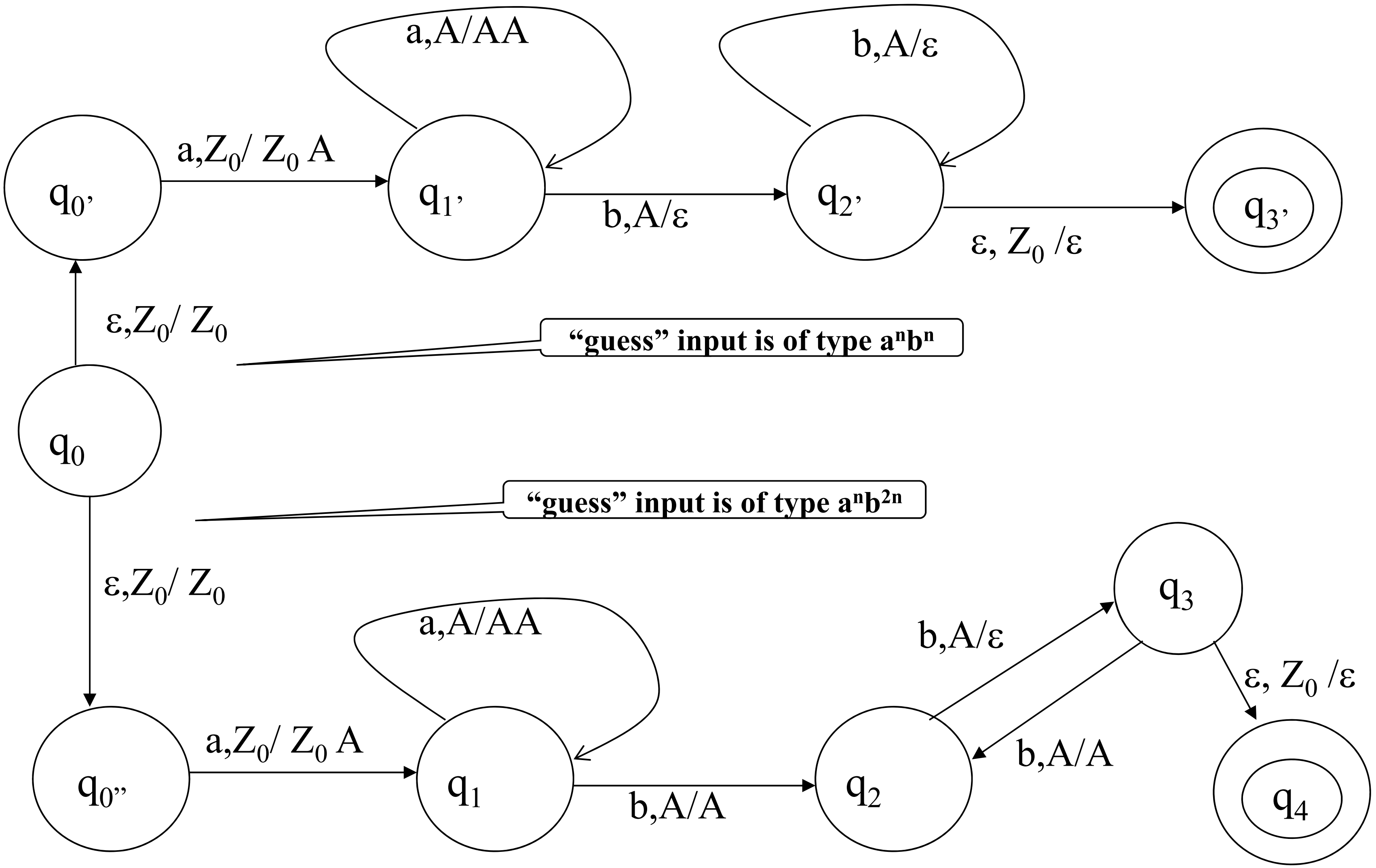
- We might as well remove the deterministic constraint and generalize:



$$\delta : Q \times (I \cup \{\varepsilon\}) \times \Gamma \rightarrow \wp_F(Q \times \Gamma^*)$$

- Why index F? (finite subsets, we do not want infinite ones)
- As usual, the NPDA accepts ***x*** if *there exists a sequence*
- c_0 |-*- <q, ε, γ>, q ∈ F
- in case of nondeterminism, the relation '|--' is not unique (i.e., ***functional***) any more

A “trivial” example: accepting $\{a^nb^n \mid n>0\} \cup \{a^nb^{2n} \mid n>0\}$



Some immediate significant consequences

- NPDA can accept a language that is not accepted by deterministic PDA ----> they are more powerful
- The previous construction can be easily generalized to obtain a *constructive* proof of closure under union of the NPDA
  -a property that deterministic PDA do not enjoy
- The closure under intersection still does not hold ($\{a^nb^nc^n\} = \{a^nb^nc^*\} \cap \{a^*b^nc^n\}$ cannot be accepted by a PDA, not even ND)
  -the two cited examples, $\{a^nb^nc^n\}$ and $\{a^nb^n\} \cup \{a^nb^{2n}\}$, are in fact not so similar…

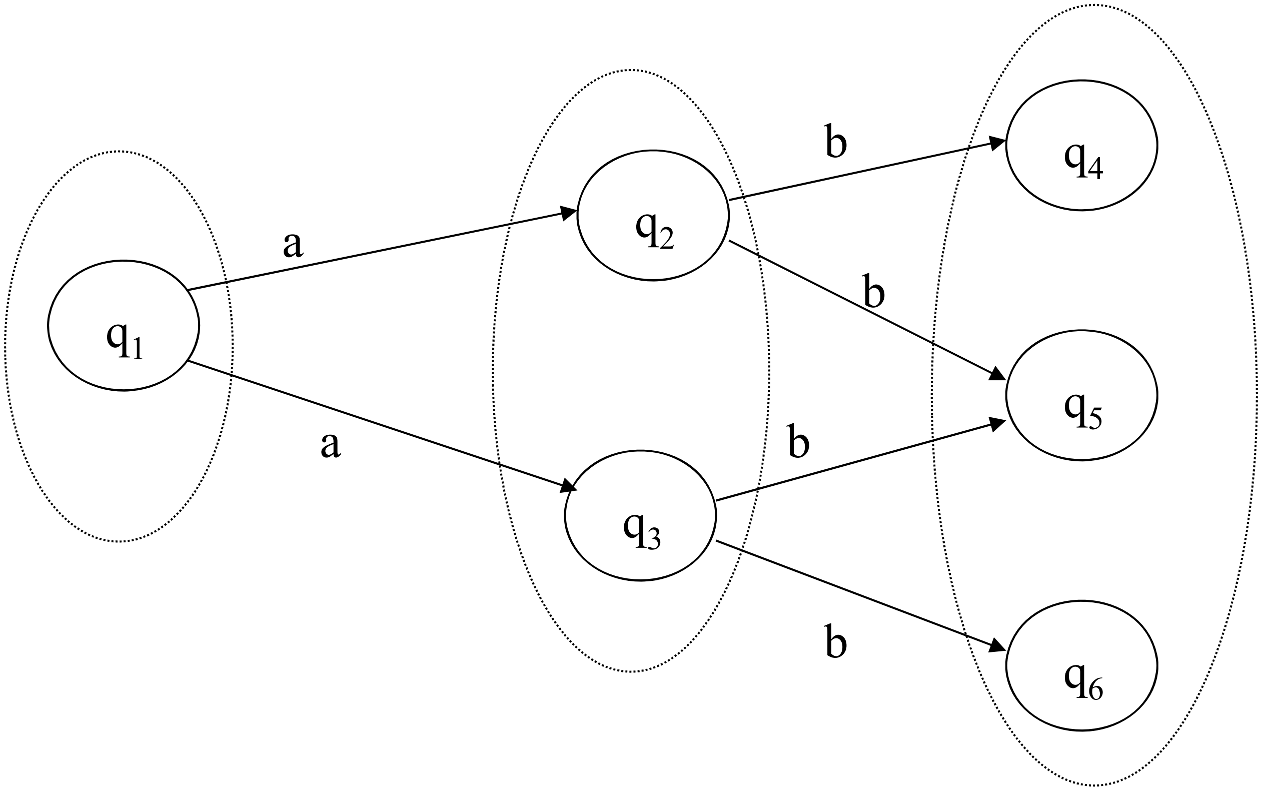
- If a language family is closed under union and not w.r.t. intersection
  it cannot be closed under complement (why?)
- Hence the family of lang. accepted by NPDA is ***not*** closed under complement
- This highlights a deep change caused by nondeterminism concerning the complement of a problem -in general-:
  if the way of operating of a machine is deterministic and its computation finishes it suffices to change the positive answer into a negative one to obtain the solution of the “complement problem” (for instance, *presence* rather than *absence* of errors in a program)

- In the case of NPDA, though it is possible, like for PDA, to make a computation always finish, there can be two computations
  - c_o |-*- $<q_1, \varepsilon, \gamma_1>$
  - c_o |-*- $<q_2, \varepsilon, \gamma_2>$
  - $q_1 \in F$, $q_2 \notin F$
- In this case ***x*** is accepted
- However, if F turned into Q-F, ***x*** is still accepted:
  with nondeterminism changing a yes into a no does not work!

- …and for other kinds of automata?

  does nondeterminism increase the power of the model?

Nondeterministic Finite-state Automata (NFA)



Starting from q_1 and reading ***ab*** the automaton reaches a state that belongs to the set $\{q_4,q_5,q_6\}$

Let us call again "state" the set of possible states in which the NFA can be during a run.

Formally ...

- Given a NFA an equivalent deterministic one can be *automatically* computed $\Rightarrow$
- NFA are ***not*** more powerful than their deterministic relatives (this is different than with PDA)
  (so what is their use?)
- Let A_{ND} = <Q_N,I, δ_N, q_{0N}, F_N> the NFA from which we build a FA
- Let A_D = <Q_D,I, δ_D, q_{0D}, F_D> the FA we intend to build
  - $\mathbf{Q_D = \wp(Q_N)}$
  - $$\delta_D(q_D, i) = \bigcup_{q_N \in q_D} \delta_N(q_N, i)$$
  - $q_{0D} = \{q_{0N}\}$
  - $$F_D = \{\overline{Q} \subseteq Q_N \mid \overline{Q} \cap F_N \neq \varnothing\}$$

- Though it is true that for all NFA one can find (and *build*) an equivalent deterministic one
- This does not mean that using NFA is useless:
  - It can be easier to “design” a NFA and then obtain from it automatically an equivalent deterministic one, just to skip the (painful) job of build it ourselves deterministic from the beginning (we will soon see an application of this idea)
  - For instance, from a NFA with 5 states one can obtain, in the worst case, one with 2^5 states!
- Consider NFA and FA for languages $L_1=(a,b)^*a(a,b)$ (i.e., strings over $\{a,b\}$ with ‘a’ as the symbol before the last one) and $L_2=(a,b)^*a(a,b)^4$ (i.e., ‘a’ as the fourth symbol before the last...)

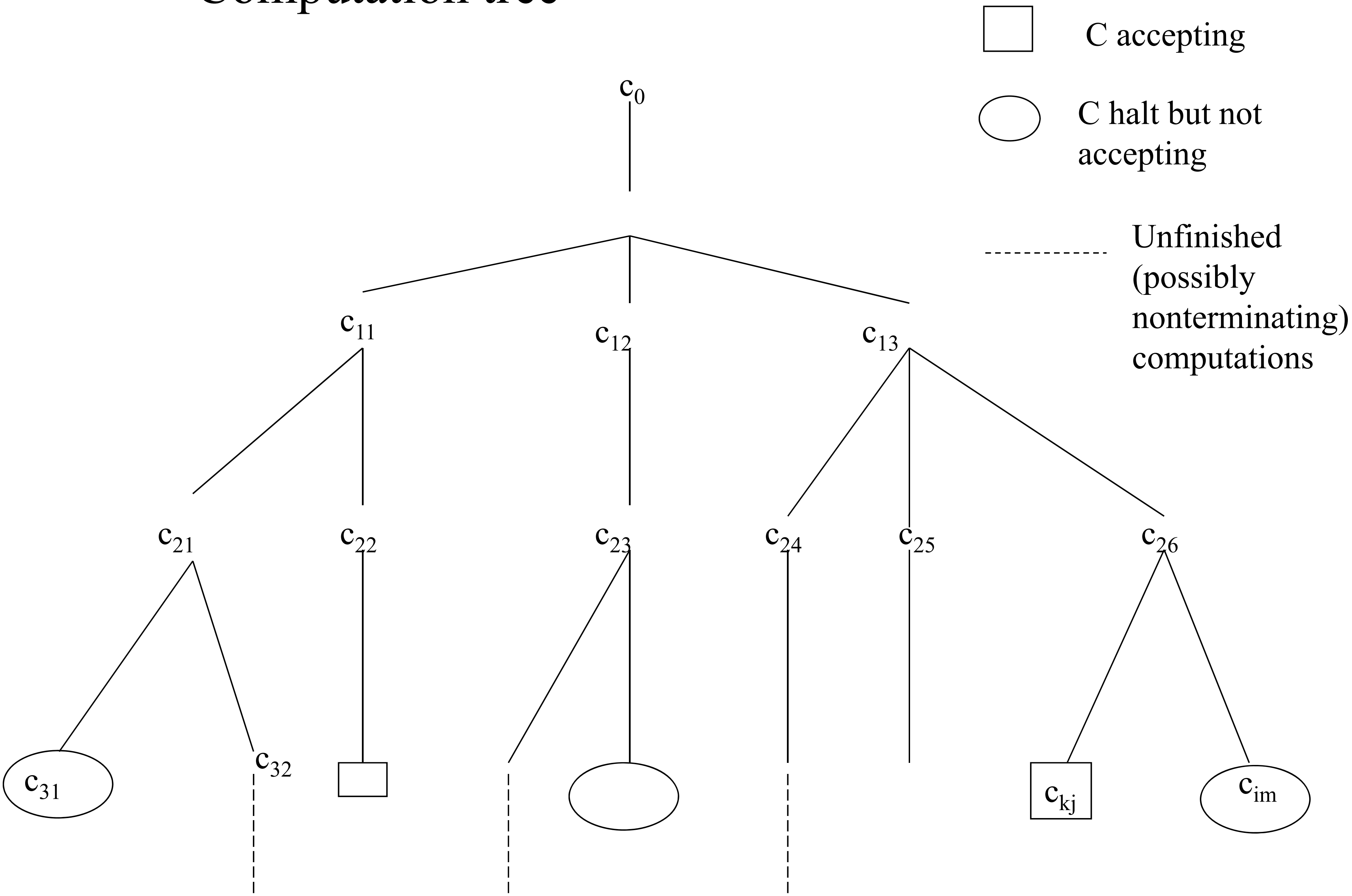
- We still have to consider the TM ...

Nondeterministic TM

$$<\delta,[\eta]>: Q \times I \times \Gamma^k \rightarrow \wp(Q \times \Gamma^k \times \{R,L,S\}^{k+1}\,[\times O \times \{R,S\}\,])$$

- •Configurations, transitions, transition sequences and acceptance are defined as usual
- •Does nondeterminism increment the power of TM's?

Computation tree



□ C accepting

○ C halt but not accepting

- - - - - Unfinished (possibly nonterminating) computations

- ***x*** is accepted by a ND TM iff ***there exists*** a computation that terminates in an accepting state
- Can a deterministic TM establish whether a “sister” ND TM accepts ***x***, that is, accept ***x*** if and only if the ND TM accepts?
- This amounts to “visit” the computation tree of the NDTM to establish whether it contains a path that finishes in an accepting state
- This is a (***almost***) trivial, well known problem of tree visit, for which there are classical algorithms
- The problem is therefore reduced to implementing an algorithm for visiting trees through TM’s: a boring, but certainly feasible exercise … but beware the above “almost” ...

- Everything is easy if the computation tree is finite
- But it could be that some paths of the tree are infinite (they describe nonterminating –hence non-accepting– computations)
- In this case a depth-first visit algorithm (for instance leftmost preorder) might "get stuck in an infinite path" without ever discovering that another branch is finite and leads to acceptance.
- The problem can however be easily overcome by adopting a breadth-first visit algorithm (it uses a queue data structure rather than a stack to manage the nodes still to be visited).
- Hence nondeterminism does ***not*** increase the power of the TM

Conclusions

- Nondeterminism: a useful abstraction to describe search problems and algorithms; or situations where there are no elements of choice or they are equivalent; or parallel computations
- In general it does not increase the computing power, at least in the case of TM's (which are the most powerful automaton seen so far) but it can provide more compact descriptions
- It increases the power of pushdown automata
- It can be applied to various computational models (to every one, in practice); in some cases "intrinsically nondeterministic" models were invented to describe nondeterministic phenomena
- For simplicity we focused only on (D and ND) acceptors but the notion applies also to translator automata
- NB: the notion of ND must not be confused with that of *stochastic* (there exist stochastic models -e.g. Markov chains- that are completely different from the nondeterministic ones)

Grammars

- Automata are a model suitable to recognize/accept, translate, compute (languages): they “receive” an input string and process it in various ways
- Let us now consider a ***generative model***:
  a grammar produces, or *generates*, strings (of a language)
- General notion of a *grammar* or *syntax* (alphabet and vocabulary, and grammar and syntax are synonymous):
  set of ***rules*** to build phrases of a language (strings): it applies to any notion of a language in the widest possible sense.
- In a way similar to normal linguistic mechanisms, a formal grammar generates strings of a language through a process of ***rewriting***:

- “A phrase is made of a subject followed by a predicate”
  “A subject can be a noun or a pronoun, or …”
  “A predicate can be a verb followed by a complement…”
- A program consists of a declarative part and an executable part
  The declarative part …
  The executable part consists of a statement sequence
  A statement can be simple or compound
  ….

- An email message consists of a header and a body
  The header contains an address, ….
- …

- In general this kind of linguistic rules describes a "main object" (a book, a program, a message, a protocol, ….) as a sequence of "composing objects" (subject, header, declarative part, …). Each of these is then "refined" by replacing it with more detailed objects and so on, until a sequence of base elements is obtained (bits, characters, …)
  The various rewriting operations can be alternative: a subject can be a noun or a pronoun or something else; a statement can be an assignment, or I/O, ...

Formal definition of a grammar

- $G = <V_N, V_T, P, S>$
  - V_N : ***nonterminal*** alphabet or vocabulary
  - V_T : ***terminal*** alphabet or vocabulary
  - $V = V_N \cup V_T$
  - $S \in V_N$: a particular element of V_N called ***axiom*** or ***initial* (Start) *symbol***
  - $P \subseteq V_N^+ \times V^*$: set of ***rewriting rules***, or ***productions***

    $P = \{<\alpha, \beta>|\ \alpha \in V_N^+ \wedge\ \beta \in V^*\}$

    for ease of notation, write $<\alpha, \beta>$ as $\alpha \rightarrow \beta$

    to emphasize the action of ***rewriting***

Example

- $V_N = \{S, A, B, C, D\}$
- $V_T = \{a,b,c\}$
- S
- $P = \{S \rightarrow AB,\ BA \rightarrow cCD,\ CBS \rightarrow ab,\ A \rightarrow \varepsilon\}$

Relation of Immediate Derivation "⇒"

$\alpha \Rightarrow \beta, \alpha \in V^{+}, \ \beta \in V^{*}$

if and only if

$\alpha = \alpha_1\alpha_2\alpha_3 \quad , \quad \beta = \alpha_1\beta_2\alpha_3 \quad \wedge \quad \alpha_2 \rightarrow \beta_2 \in P$

α_2 is rewritten as β_2 in the context of α_1 and α_3

With reference to the previous grammar: applying rule **BA → cCD**

$\mathrm{aa\underline{BA}S} \Rightarrow \mathrm{aa\underline{cCD}S}$

As usual, define the reflexive and transitive closure of ⇒

$$\overset{*}{\Rightarrow}$$

it means: "zero or more rewriting steps"

Language generated by a grammar

$$L(G) = \{x \mid x \in V_T^* \wedge S \overset{*}{\Rightarrow} x\}$$

It consists of all strings, containing ***only terminal*** symbols, that can be derived (in any number of steps) from ***S***

NB: not necessarily all derivations lead to a string of terminal symbols

some may “get stuck” (string is not terminal but no rule can be applied)

some may be “never ending”

A first example

$G_1 =< \{S, A, B\}, \{a, b, 0\}, P, S >$

$P = \{S \to aA, A \to aS, S \to bB, B \to bS, S \to 0\}$

Some derivations

$S \Rightarrow 0$

$S \Rightarrow aA \Rightarrow aaS \Rightarrow aa0$

$S \Rightarrow bB \Rightarrow bbS \Rightarrow bb0$

$S \Rightarrow aA \Rightarrow aaS \Rightarrow aabB \Rightarrow aabbS \Rightarrow aabb0$

Through an easy generalization :

$L(G_1) = \{aa, bb\}^*.0$

Second example

$G_2 =< \{S\}, \{a,b\}, P, S >$

$P = \{S \rightarrow aSb \mid ab\}$ (abbreviation for $S \rightarrow aSb$, $S \rightarrow ab$)

Some derivations

$S \Rightarrow ab$

$S \Rightarrow aSb \Rightarrow aabb$

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

Through an easy generalization :

$L(G_2) = \{a^n b^n \mid n \geq 1\}$

By substituting production $S \rightarrow ab$ with $S \rightarrow \varepsilon$ we obtain

$L(G_2) = \{a^n b^n \mid n \geq 0\}$

Third example: G_3

$$\{S \to aACD, A \to aAC, A \to \varepsilon, B \to b, \underline{CD \to BDc,} \underline{CB \to BC,} D \to \varepsilon\}$$

$$S \Rightarrow a\boldsymbol{A}CD \Rightarrow a\boldsymbol{CD} \Rightarrow a\boldsymbol{BD}c \overset{*}{\Rightarrow} abc$$

$$S \Rightarrow a\boldsymbol{A}CD \Rightarrow aa\boldsymbol{A}CCD \Rightarrow aaCC\boldsymbol{D} \Rightarrow aaCC\downarrow$$

$$S \Rightarrow a\boldsymbol{A}CD \Rightarrow aa\boldsymbol{A}CCD \Rightarrow aaC\boldsymbol{CD} \Rightarrow aa\boldsymbol{CB}Dc \Rightarrow$$

$$aa\boldsymbol{B}CDc \Rightarrow aab\boldsymbol{CD}c \Rightarrow aab\boldsymbol{B}Dcc \Rightarrow aabb\boldsymbol{D}cc \Rightarrow aabbcc$$

$$S \overset{*}{\Rightarrow} aaa\boldsymbol{A}CCCD \Rightarrow aaaCC\boldsymbol{CD} \Rightarrow aaaCC\boldsymbol{B}Dc \Rightarrow aaaCCb\boldsymbol{D}c \Rightarrow aaaCCbc\downarrow$$

…

1. **$S \to aACD$, $A \to aAC$ and $A \to \varepsilon$ generate as many *a*'s as *C*'s, and a final *D***

2. any $x \in L$ includes only terminal symbols, hence nonterminal symbols must disappear

3. ***C* disappears only when it "hits" the *D* and then it generates a '*B*' and a '*c*'**

4. ***C*'s and *B*'s must switch to permit all the *C*'s to reach the *D***

5. ***Hence $C^n D \Rightarrow^* b^n c^n$***

6. Hence $L = \{a^n b^n c^n \mid n > 0\}$

Some “natural” questions

- What is the practical use of grammars (beyond funny “tricks” like $\{a^nb^n\}$?)
- What languages can be obtained through grammars?
- What relations exist among grammars and automata (better: among languages *generated* by grammars and languages *accepted* by automata?

Some answers

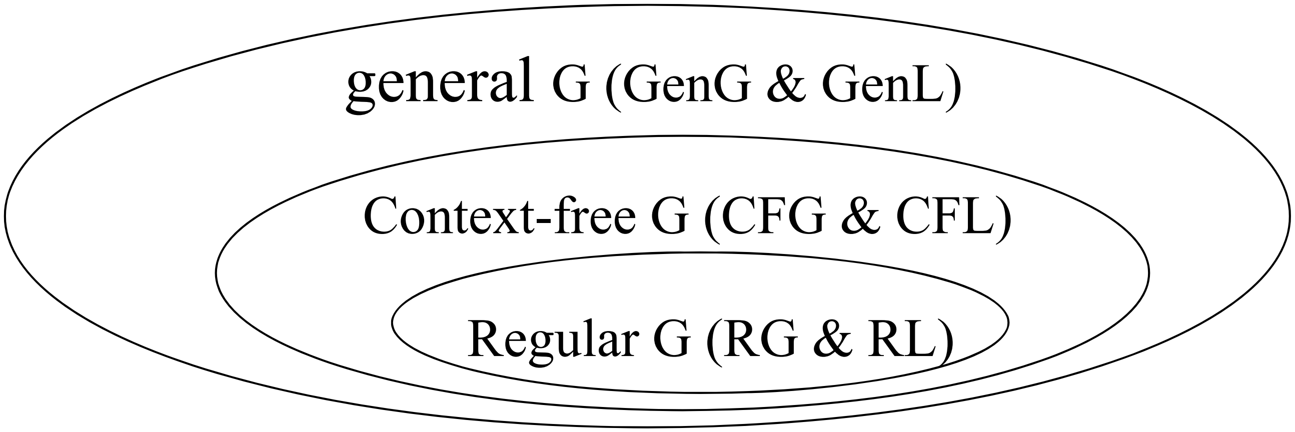
- Definition of the syntax of the programming languages
- Applications are “dual” w.r.t. automata: grammars ***generate*** languages, whereas automata ***recognize*** them
- Simplest example: language *compilation*: the grammar ***defines*** the language, the automaton ***accepts*** and ***translates*** it

Classes of grammars

- Context-free grammars:
  - $\forall$ rule $(\alpha \rightarrow \beta) \in P$, $|\alpha| = 1$, i.e., α is an element $\boldsymbol{A}$ of $\boldsymbol{V_N}$.
  - Context free because the rewriting of α (i.e., of $\boldsymbol{A} \in \boldsymbol{V_N}$) does not depend on its context (the string parts surrounding it do not appear in the left-hand side of the rule)
  - These are in fact the same as the BNF used for defining the syntax of programming languages (so they are well fit to define typical features of programming and natural languages, … but not all)
  - G_1 and G_2 above are context-free not so for G_3

- Regular Grammars:
  - $\forall$ rule $(\alpha \to \beta) \in P$, $|\alpha| = 1$, $\beta \in ((V_T . V_N) \cup V_T \cup \{\varepsilon\})$
  - Regular grammars are also context free, but not vice versa
  - G_1 above is regular, not so G_2.

Inclusion relations among grammars and corresponding languages



NB: we call Regular Languages those generated by a regular grammar

this appears to be an abuse (a name clash with languages accepted by FA's), but it is not ...

It immediately follows that :

$\mathcal{RL} \subseteq \mathcal{CFL} \subseteq \mathcal{GenL}$

But, are these inclusions strict?

The answer comes from the comparison with automata

Relations between grammars and automata (with few surprises)

- Define “equivalence” between RG and FA

  (i.e., the FA accepts same language that the RG generates)

  - **From FA to RG**: given a FA $A=(Q, I, q_0, F)$, let $V_N = Q$, $V_T = I$, $S = <q_0>$, and, for each $\delta(q, i) = q'$ let $<q> \rightarrow i<q'>$ and, if $q' \in F$, add $<q> \rightarrow i$
  - It is an easy intuition (proved by induction) that $\delta^*(q, x) = q'$ iff $<q> \Rightarrow^* x<q'>$, and hence, if $q' \in F$, $<q> \Rightarrow^* x$

- Vice versa, **from RG to FA**:
  - Given a RG, let $Q = V_N \cup \{q_F\}$, $I = V_T$, $<q_0> = S$, $q_F \in F$ and,

    for each $A \rightarrow bC$ let $\delta(A,b) = C$

    for each $A \rightarrow b$ let $\delta(A,b) = q_F$
  - for each $A \rightarrow \varepsilon$ let $A \in F$

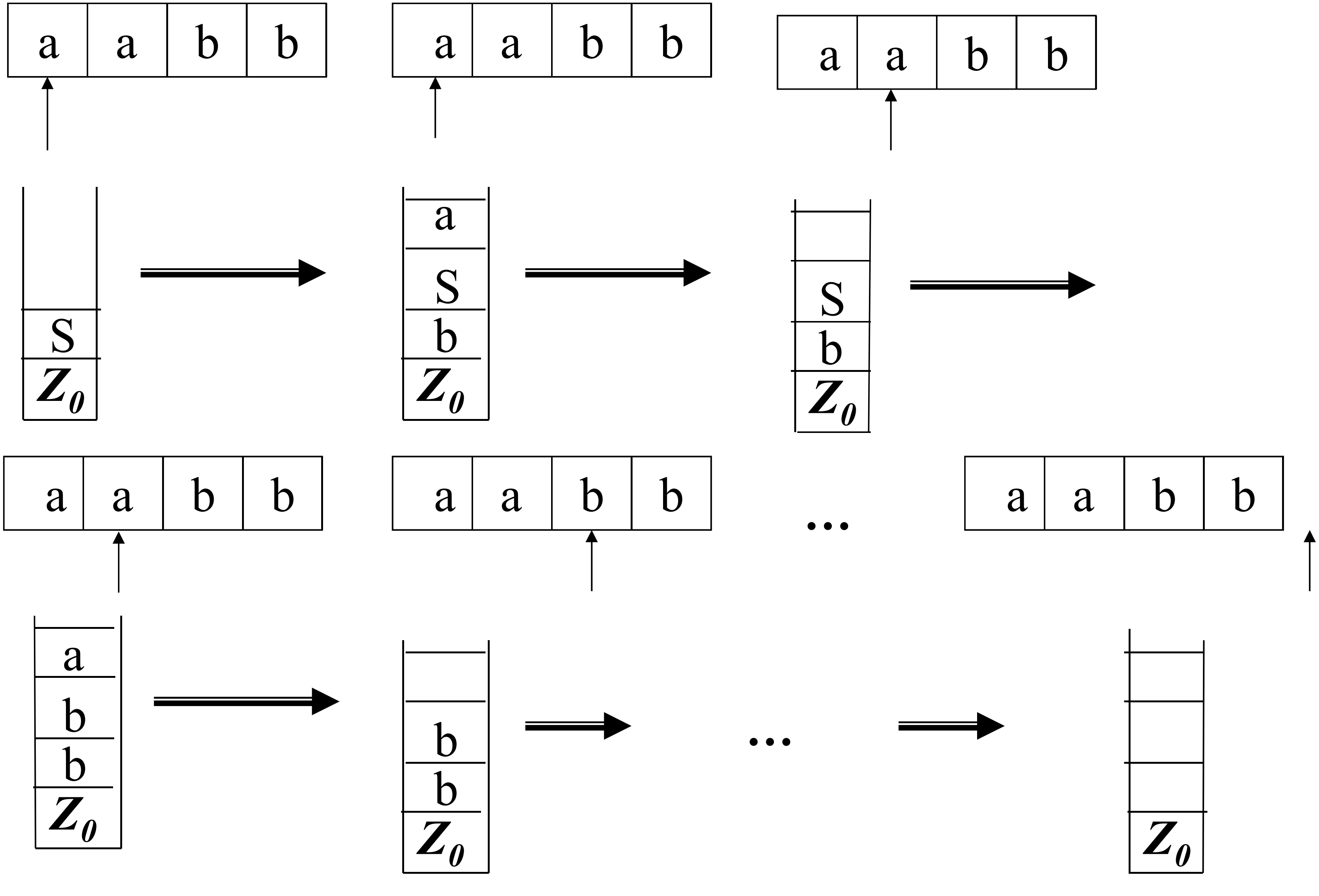
NB: The FA thus obtained is nondeterministic (why?): much easier!

- CFG equivalent to PDA (ND!)

intuitive justification (no proof:
the proof is the “hart” of compiler construction)

S → aSb | ab

$$S \Rightarrow aSb \Rightarrow aabb$$



genG equivalent to TM

- Given G let us construct (in broad lines) a ***ND*** TM, M, accepting L(G):
  - The input string *x* is on the input tape
  - M has one memory tape: it tries in all possible ways to derive *x* on it, and accepts *x* iff it finds a derivation for it
  - The memory tape is initialized with *S* (better: Z_0S)
  - The memory tape (which, in general, will contain a string α, $\alpha \in V^*$) is scanned searching the left part of some production of *P*
  - When one is found -*not necessarily the first one, M operates a ND choice*- it is substituted by the corresponding right part (if there are many right parts again *M* operates nondeterministically)

– This way:

$$\alpha \Rightarrow \beta \quad \leftrightarrow \quad c = < x, q_s, Z_0\alpha > |-*- < x, q_s, Z_0\beta >$$

If and when the tape holds a string $y \in V_T^*$, it is compared with x. If they coincide, x is accepted, otherwise this particular computation of the nondeterministic TM does not lead to acceptance.

Notice that:

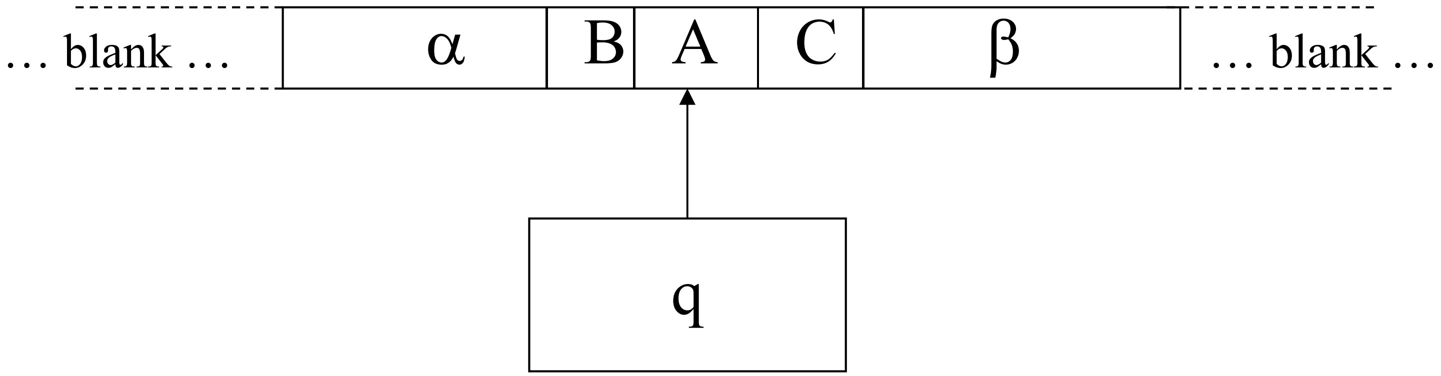
•Using a ND TM facilitates the construction but is immaterial (not necessary)

•It is instead necessary -and, we will see, unavoidable- that, if $x \notin L(G)$, M might "try an infinite number of ways", some of which might never terminate, without being able (rightly) to conclude that $x \in L(G)$, but *not even the opposite*.

This is consistent with the definition of acceptance, which requires M to reach an accepting configuration if and only if $x \in L$, but does not requires M to terminate its computation without accepting (i.e., in a "rejecting state") if $x \notin L$

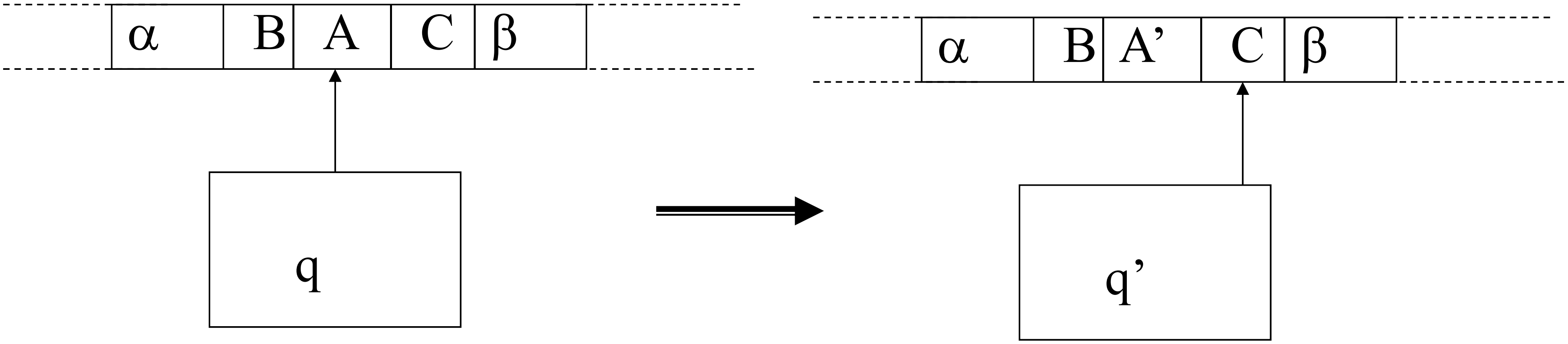
- Given M (***single tape***, for ease of reasoning and without loss of generality) we define (in broad lines) a G generating L(M):
  - First, G generates *all* strings of the type x\$X, $x \in V_T^*$, X being a “copy of x” composed of nonterminal symbols (e.g., for x = aba, x\$X = aba\$ABA)
  - G simulates the successive configurations of M using the string on the right of \$
  - G is defined in a way such that it has a derivation $x\$X \Rightarrow^* x$, with $x \in V_T^*$, if and only if x is accepted by M.
  - The idea: simulate each move of M by an immediate derivation of G

- We represent the configuration



- through the string (special cases are left as an exercise): $\$\alpha BqAC\beta$
- to start, G has therefore a set of derivations of the kind $x\$X \Rightarrow x\$q_oX$ (where q_oX encodes the initial configuration of M)
- for each value of the transition function δ of M, a rule of G is defined :
  - $\delta(q,A) = <q', A', R>$ G includes the production $qA \rightarrow A'q'$
  - $\delta(q,A) = <q', A', S>$ G includes the production $qA \rightarrow q'A'$
  - $\delta(q,A) = <q', A', L>$ G includes the productions $BqA \rightarrow q'BA'$ $\forall$ B in the alphabet of M (recall that M is single tape, hence it has a unique alphabet for input, memory, and output)

– This way, for instance:



– If and only if: x$αBqACβ ⇒ x$αBA’q’Cβ,

– etc. …

– We finally add productions allowing G to derive from x$αBq$_F$ACβ a unique x if –and only if– M reaches an accepting configuration (αBq$_F$ACβ), by deleting whatever is to the right of $, and also $