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, “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
    • By Email (administrative matters and to fix face-to-face meetings)
      – morzenti@elet.polimi.it
Organization (2/3)

- Exam tests *ability to apply, not to repeat*: mainly written
can consult textbooks and notes
not repetitive, challenging (hopefully)
- Standard written (+ possibly oral) exams on the whole program
  - No midterm exams, forbidden by faculty rules
  - Details will be notified in due time…
Organization (3/3)

• Teaching material:
  – Official text:
    • Ghezzi/Mandrioli: Theoretical Foundations of Computer Science [Wiley]
      – Not re-printed by the publisher: you can photocopy it, possibly in a legal way (ask the teacher how)
    • In Italian: Carlo Ghezzi, Dino Mandrioli, Informatica Teorica, UTET.
  – Previous exam texts with solutions, some in Italian (sorry…) MOST in English
  – Lesson slides (NOT to use as a substitute of the text book!), exercise book, notices and various material on the course available for download at the web page: http://home.dei.polimi.it/morzenti/tcs.html
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 (w.r.t. “time”)

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

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

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

\[ a.x^2 + b.y^2 = c \]
• 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
• 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 *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 \(\varepsilon\): \(|\varepsilon| = 0\)
• \(A^*\) = set of all strings, including \(\varepsilon\), over A.
  \(A = \{0,1\}, \ A^* = \{\varepsilon, 0, 1, 00, 01, 10, \ldots\}\)
• Operations on strings:
  concatenation (product):  \( x \cdot y \)

  \( x = abb, \ y = baba, \ x \cdot y = abbbaba \)
  \( x = \text{Quel ramo}, \ y = \text{del lago di Como}, \)
  \( x \cdot y = \text{Quel ramo del lago di Como} \)

  “.” is an associative, non-commutative operation

• \( A^* \) is called
  \textit{free monoid} over \( A \) built through “.”

• \( \varepsilon \) is the unit w.r.t. “.”
Language

• $L \subseteq A^*$
  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:
  \[ \cup, \cap, L_1 - L_2, \neg L = A^* - L, \quad (\bar{L} = \neg L) \]

• Concatenation (of languages):
  \[ L_1 \cdot L_2 = \{x.y | x \in L_1, y \in L_2\} \]

\[ L_1 = \{0,1\}^*, \quad L_2 = \{a,b\}^* \]

\[ L_1 . L_2 = \{\varepsilon, 0, 1, 0a, 11b, abb, 10ba, \ldots\} \quad NB: \text{ab1 not included!} \]
• \( L^0 = \{ \varepsilon \} \), \( L^i = L^{i-1} . L \)

• \( L^* = \bigcup_{n=0}^{\infty} L^n \)

NB: \( \{ \varepsilon \} \neq \emptyset \)!

\( \{ \varepsilon \} . L = L; \)
\( \emptyset . L = \emptyset \)

• \( L^+ = \bigcup_{n=1}^{\infty} L^n = L^* - L^0 \)
Some practical implications

• let
  – $L_1$: set of “Word/Mac” documents, and
  – $L_2$: set of “Word/PC” documents; then
  – $L_1 \cap L_2$: set of Word documents that are “compatible Mac-PC” (= $\emptyset$?)

• Composition of a message over the net:
  – x.y.z:
    – x = header (address, …)
    – y = text
    – z = “tail”

•
  – $L_1$: set of e-mail messages
  – $L_2$: set of SPAM messages
  – Filter: $L_1 - L_2$ (= $\emptyset$?)
• A language is a means of expression …
• Of a problem
• $x \in \mathcal{L}$?
  – Is a message correct?
  – Is a program correct?
  – $y = x^2$?
  – $z = \text{Det}(A)$? (is $z$ the determinant of matrix $A$)
  – Is the soundtrack of a movie well synchronized with the video?
Translation

• $y = \tau(x)$
  
  $\tau$: translation is a function from $L_1$ to $L_2$
  - $\tau_1$: double the “1” (1 --> 11):
    $\tau_1(0010110) = 0011011110$, …
  - $\tau_2$: change a with b and vice versa (a <---> 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) \((FSA)\):
  - A finite state set:
    - \{on, off\}, ....
    - \{1, 2, 3, 4, ...k\}, \{TV channels\}, \{income classes\}, ...

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) \((x \in L?)\)

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

\[
L = \{ \text{strings with an even number of "1" any number of "0"} \} 
\]
Formalization of the notion of acceptance of language $L$

- Move sequence:
  - $\delta^*: Q \times I^* \rightarrow Q$
    - $\delta^*$ defined inductively from $\delta$, by induction on the string length
    - $\delta^*(q,\varepsilon) = q$
    - $\delta^*(q,y.i) = \delta(\delta^*(q,y), i)$

- 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

The diagram shows a finite state machine ( FSM ) accepting Pascal identifiers. The states are labeled q₀, q₁, and qₑ. The transitions are marked with symbols: <letter> and <digit>. The transitions are as follows:

- From q₀, a <letter> transition leads to q₁.
- From q₁, <letter> transitions lead back to q₁.
- From qₑ, <digit> transitions lead to qₑ.
- From qₑ, <letter> transitions lead to qₑ.

The symbols represented in the transitions are:
- <letter> for letters of the alphabet
- <digit> for digits 0-9

The possible input symbols are:
- 0
- 1
- 2
- ...
- 9
- a
- b
- c
- ...
- Z
Automata as language translators

\[ y = \tau(x) \]

Transitions with output:

\[ \tau: \text{halves the number of } "0" \text{ and doubles the number of } "1" \text{ (there must be an even number of } "0") \]
Formalization of translating automata (transducers)

- $T = \langle Q, I, \delta, q_0, F, O, \eta \rangle$
  - $\langle Q, I, \delta, q_0, F \rangle$: just like for acceptors
  - $O$: output alphabet
  - $\eta: Q \times I \rightarrow O^*$
- $\eta^*: Q \times I^* \rightarrow O^*$, $\eta^*$ defined inductively as usual
  $\eta^*(q, \varepsilon) = \varepsilon$
  $\eta^*(q, y.i) = \eta^*(q, y).\eta(\delta^* (q, y), i)$
- $\tau(x) [x \in L] = \eta^*(q_0, x) [\delta^* (q_0, x) \in F]$
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 sectors, 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 $q_1 \xrightarrow{aabab} q_1$

If one goes through the cycle once, then one can also go through it 2, 3, ..., n, ... 0 times $\xrightarrow{========}$
Formally:

- If \( x \in L \) and \( |x| > |Q| \) there exists a \( q \in Q \) and a \( w \in I^+ \) such that:
  - \( x = ywz \)
  - \( \delta^* (q, w) = q \)

Then \( yw^nz \in L, \forall n \geq 0 \)

- This is known as the **Pumping Lemma**
Several properties of FSA –both good and bad ones– follow from the pumping lemma and other properties of the graph of $\delta$

- Let $A$ be an FSA, and $L$ the language accepted by $A$
- $L = \emptyset$?
  $\exists x \in L \iff \exists y \in L, |y| < |Q|$: 
  proof: Eliminate all cycles from $A$, then look for a path from initial state $q_0$ to a final state $q \in F$
- $|L| = \infty$? Similar reasoning
- ...
- Notice that being able to answer the question “$x \in L$?” for any string $x$, does not enable us to answer the other questions, such as emptiness of the accepted language !!
A “negative” consequence of the Pumping Lemma (PL)

- Is the language \( L = \{a^n b^n | n > 0\} \) recognized by any FSA?
  - Put it another way: can we count using a FSA?
- Let us assume so (by contradiction):
- Consider \( x = a^m b^m \), with \( m > |Q| \) and apply the PL: then \( x = ywz \) and \( xw^n y \in L \ \forall n \)
- There are 3 possible cases:
  - \( x = ywz \), \( w = a^k \), \( k > 0 \) \( \Rightarrow \) \( a^{m+r+k} b^m \in L \), \( \forall r \) : this cannot be (it contradicts the assumption)
  - \( x = ywz \), \( w = b^k \), \( k > 0 \) \( \Rightarrow \) idem
  - \( x = ywz \), \( w = a^k b^s \), \( k,s > 0 \) \( \Rightarrow \) \( a^{m-k} a^k b^s a^k b^s b^{m-s} \in L \): 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 FSA
  – But this is a wrong abstraction of (way of looking at) a computer!
  – It is important to consider an abstract notion of infinity!

• 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 FSA

• The mathematical notion of closure:
  – Natural numbers are closed w.r.t. the sum operation
  – But not w.r.t. subtraction
  – Integer numbers are closed w.r.t. sum, subtraction, multiplication, but not …
  – Rational numbers …
  – Real numbers …
  – Closure (w.r.t. operations and relations) is a very important notion
In the case of Languages:

- $L = \{L_i\}$: a family of languages

- $L$ is closed w.r.t. OP if and only if (iff)
  for every $L_1, L_2 \in L$, $L_1 \text{ OP } L_2 \in L$.

- $R$: regular languages, those accepted by an FSA

- $R$ is closed w.r.t. set theoretic operations, concatenation, “*”, ... and virtually “all” others.
I can simulate the “parallel run” of A and B by simply “coupling them”:
Formally:

- Given \( A^1 \) and \( A^2 \)

- The automaton \( A^3 \) is defined as:

- One can show by a simple induction that

\[
L(<A^1, A^2>) = L(A^1) \cap L(A^2)
\]
Union

• A similar construction …
  otherwise ... exploit identity $A \cup B = \neg(\neg A \cap \neg B)$
  $\Rightarrow$ need a FSA for the complement language

Complement:

0
\[ q_0 \]
1
0
\[ q_1 \]

An idea: $F^\land = Q - F$:

Yes, it works for the automaton above, but …. 
If I only turn $F$ into $Q - F$ the "complement $F$ construction" doesn't work. The problem is caused by the fact that $\delta$ is partial:

$L$: strings with exactly one '1'

$\neg L$: strings with no '1' or with more than one '1'
Some general remarks on the complement

• If I can examine all the string then it suffices the “turn a yes into a no” (F into Q-F)
• If, for some string, I do not reach the end of the string (I get blocked or …) then turn F into Q-F does not work
• With FSA the problem is easily solved …
• In general one must pay attention in considering a negative answer to a problem (e.g., $x \in L$) as the positive answer for the complement problem (e.g., $x \in \neg L$)!
Let us increase the power of the FSA by increasing its memory

- A more “mechanical” view of the FSA:
• Now let us “enrich it”:

Input tape

Control device (finite state)

Output tape

“stack” memory
The move of the stack automaton:

• Depending on:
  – the symbol read on the input tape (but it could also not read anything …)
  – the symbol read on top of the stack
  – the state of the control device:

• the stack automaton
  – changes its state
  – moves ahead the scanning head
  – changes the symbol A read on top of the stack with a string \( \alpha \) of symbols (possibly empty: this amounts to a pop of A)
  – (if translator) it writes a string (possibly empty) on the output tape (moving the writing head consequently)
• The input string $x$ is recognized (accepted) if
  – The automaton scans it completely (the scanning head
    reaches the end of $x$)
  – Upon reaching the end of $x$ it is in an acceptance state (just
    like for the FSA)

• If the automaton is also a translator, then
  $\tau(x)$ is the string on the output tape after $x$ has been
  completely scanned (if $x$ is accepted, otherwise $\tau(x)$ is
  undefined: $\tau(x) = \bot$

  ($\bot$ is the “undefined” symbol)
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:

\[ q_0 \xrightarrow{a,Z_0/Z_0A} q_1 \xrightarrow{b,A/\varepsilon} q_2 \xrightarrow{\varepsilon, Z_0/\varepsilon} q_3 \]

\( \varepsilon \)-move (AKA *spontaneous* move)
A (classical) stack automaton-translator

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

- Stack automaton [transducer]: $<Q,I,\Gamma,\delta, q_0, Z_0, F, O, \eta>$$

- $Q, I, q_0, F [O]$ just like FSA [FST]

- $\Gamma$ stack alphabet (disjoint from other ones for ease of definition)

- $Z_0$ : initial stack symbol (not essential, useful to simplify definitions)

- $\delta: Q \times (I \cup \{\epsilon\}) \times \Gamma \rightarrow Q \times \Gamma^*$ $\delta$ : partial!

- $\eta: Q \times (I \cup \{\epsilon\}) \times \Gamma \rightarrow O^*$ ($\eta$ defined where $\delta$ is)

Graphical notation:

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

$w = \eta(q,i, A)$
• Configuration (a generalization of the notion of state):
  \[ c = <q, x, \gamma, [z]>: \]
  
  – \( q \): state of the control device
  
  – \( x \): \textit{unread} portion of the input string (the head is positioned on the first symbol of \( x \))
  
  – \( \gamma \): string of symbols in the stack
    (convention: \( \text{<high-right, low-left>} \))
  
  – \( z \): string written (up to now) on the output tape
• Transition among configurations:
\[ c = <q, x, \gamma, [z]> \mid-- c' = <q', x', \gamma', [z']> \]
  - \( \gamma = \beta A \)

  - **Case 1:** \( x = i.y \) and \( \delta(q, i, A) = <q', \alpha> \) (is defined)
    \[ [\eta(q,i, A) = w] \]
    - \( x' = y \)
    - \( \gamma' = \beta \alpha \)
    - \( [z' = z.w] \)

  - **Case 2:** \( \delta(q, \epsilon, A) = <q', \alpha> \) (is defined)
    \[ [\eta(q, \epsilon, A) = w] \]
    - \( x' = x \)
    - \( \gamma' = \beta \alpha \)
    - \( [z' = z.w] \)

• **NB:** \( \forall q,A,i \ (\delta(q, \epsilon, A) \neq \bot \Rightarrow \delta(q, i, A) = \bot) \) (hence \( \delta(q, i, A) \neq \bot \Rightarrow \delta(q, \epsilon, A) = \bot \))
  - i.e., \( \epsilon \)-moves are alternative/exclusive to input-consuming ones
• Otherwise … nondeterminism!
• Acceptance [and translation] of a string
• |-* - : reflexive, transitive closure of the relation |-–
  – i.e., |-* - denotes a number $\geq 0$ of “steps” of the relation |-–
• $x \in L \ [z = \tau(x)] \iff$
  $$c_0 = <q_0, x, Z_0, [\varepsilon]> \ |-* - c_F = <q, \varepsilon, \gamma, [z]>, q \in F$$

Pay attention to $\varepsilon$-moves, especially at the end of the string!
Stack 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
- ….
  Will occur very frequently in the course of Languages and translators
Properties of stack automata
(especially as acceptors)

• $\{a^n b^n \mid n > 0\}$ is accepted by a stack automaton (not by a FSA)
  – However $\{a^n b^n c^n \mid n > 0\}$ ….
  – 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 a destructive memory: to read it, one must destroy it!
    This limitation of the stack automaton can be proved formally through a generalization of the pumping lemma.

• $\{a^n b^n \mid n > 0\}$ accepted by a stack automaton;
  $\{a^n b^{2n} \mid n > 0\}$ accepted by a stack automaton

• However $\{a^n b^n \mid n > 0\} \cup \{a^n b^{2n} \mid n > 0\}$ …
  – Reasoning -intuitively- similar to the previous one:
    – If I empty all the stack with $n$ b’s then I forget if there are 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

- \( L_P \) = class languages accepted by stack automata
- \( L_P \) is not closed under union nor intersection
- Why?
  - [consider the languages \( \{a^n b^n c^n | n > 0\} \) and \( \{a^n b^n | n > 0\} \cup \{a^n b^{2n} | n > 0\} \) ]
- Considering the complement …
  The same principle as with FSA: change the accepting states into non accepting states.
  There are however new difficulties
• Function $\delta$ must be completed (as with FSA) with an error state. Pay attention to the nondeterminism caused by $\varepsilon$-moves!
• The $\varepsilon$-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 $F^\uparrow = Q - F$.
• 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 $\varepsilon$-moves at the end of the scanning with some states in $F$ and other ones not in $F$?
• \(<q_1, \varepsilon, \gamma_1> \longrightarrow <q_2, \varepsilon, \gamma_2> \longrightarrow <q_3, \varepsilon, \gamma_3> \longrightarrow \ldots\)
\(q_1 \in F, q_2 \notin F, \ldots\)

• Then we must “force” the automaton to accept only at the end of a (necessarily finite) sequence of \(\varepsilon\)-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.
Stack automata [as acceptors (SA) or translators (ST)] are more powerful than finite state ones (a FSA is a trivial special case of a SA; more, SA have an unlimited counting ability that FSA lack) However also SA/ST have their limitations … … a new and “last” (for us) automaton:

the *Turing Machine* (TM)
Historical model of “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 than the (even simpler) original model. This choice will be explained later.
k-tape TM

Input tape

Output tape

First memory tape

Second memory tape

K-th memory tape
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 $[0,1,2,\ldots]$ 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
  – 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) (if it has written it moves; if it moves without writing it leaves a blank on the tape cell)
As a consequence:

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

(partial!)

Graphical notation:

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

$$M_i \in \{R, L, S\} \quad M_{k+1} \in \{R, S\}$$

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:
    $<\delta, [\eta]> (q, \ldots) = \bot \ \forall \ q \in F$:
  - The TM stops when $<\delta, [\eta]> (q, \ldots) = \bot$
  - Input string $x$ is accepted iff:
    • After a **finite** number of moves the TM stops (hence it is in a configuration
      where $<\delta, [\eta]> (q, \ldots) = \bot$)
    • When it stops, its state $q \in F$

• **NB:**
  - $x$ is *not* accepted if:
    • The TM stops in a state $\not\in F$; or
    • The TM never stops
  - There is a similarity with SA (also a non loop-free SA might not accept because
    of a “non stopping run”), however … does there exist a loop-free TM?
Some examples

- A TM accepting \( \{a^n b^n c^n \mid n > 0\} \)
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\( \neq 9 \) becomes the successive digit, then all other ones are unchanged, …)
- \( M \) copies \( T_2 \) on the output tape.

- Notation:  
  - \( \square \) : any decimal digit  
  - \( _\_ \) : blank  
  - \( \# \) : any digit \( \neq 9 \)  
  - \( ^\wedge \) : successor of a digit denoted as \( \# \) (in the same transition)
Copy input on $T_1$

Change rightmost 9's into 0's

Copy rest of $T_1$ on $T_2$

Copy all $T_2$ on output

Left end reached (all 9's): add an initial 1

Then all others are 0
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.

$\Rightarrow$

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

---

X

CD
• 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:
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 problem it can solve)
• There can be a (profound) impact for what concerns the complexity of the computations
• We will consider the implications in both cases
Nondeterministic (operational) models

• Usually one thinks of an algorithm as a determined sequence of operations: in a certain state and with certain input there is no doubt on the next “step”
• Are we sure that this is desirable?

Let us compare

\[
\text{if } x > y \text{ then } \max := x \text{ else } \max := y
\]

with

\[
\text{if } x \geq y \text{ then } \max := x \\
\quad \text{y } \geq x \text{ then } \max := y \\
\text{fi}
\]
• 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 to 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 the nondeterminism)
Among the numerous nondeterministic (ND) models:
ND version of known models

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

Formally:

\[ \delta(q_1, a) = \{q_2, q_3\} \]

\[ \delta : Q \times I \rightarrow \mathcal{P}(Q) \]
$\delta^*$: 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 FSA accept?

\[ x \in L \iff \delta^*(q_0, x) \cap F \neq \emptyset \]

Among the various possible runs (with the same input) of the ND FSA it suffices that one of them (there exists one that) succeeds to accept the input string

Another interpretation of nondeterminism:

**universal** nondeterminism (the previous was **existential**):

**all** runs of the automaton accept

\[ (\delta^*(q_0, x) \subseteq F) \]
nondeterministic SA (NDSA)

- In fact they are “natural born” ND:

\[ q_1, i, A/\alpha \rightarrow q_2 \]
\[ \varepsilon, A/\beta \rightarrow q_3 \]
• 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 subset)
• As usual, the NDSA accepts x if \textit{there exists a sequence}
• \[ c_0 \vdash^* \langle q, \varepsilon, \gamma \rangle, \quad q \in F \]
• \( \vdash \) is not unique (i.e., \textit{functional}) any more!
A “trivial” example: accepting \( \{a^n b^n \mid n>0\} \cup \{a^n b^{2n} \mid n>0\} \)
Some immediate significant consequences

- NDSA can accept a language that is not accepted by deterministic SA ----> they are more powerful
- The previous construction can be easily generalized to obtain a constructive proof of closure w.r.t. union of the NDSA -a property that deterministic SA do not enjoy
- The closure w.r.t. intersection still does not hold ($\{a^nb^nc^n\} = \{a^nb^nc^*\} \cap \{a^*b^n c^n\}$ cannot be accepted by a SA, 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 w.r.t. union and not w.r.t. intersection it cannot be closed w.r.t. complement (why?)
• Hence languages NDSA are not closed w.r.t. 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 NDSA, though it is possible, like for DSA, 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 other kinds of automata?
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 NDFSA can be during a run.

Formally ...
• Given a ND FSA an equivalent deterministic one can be *automatically* computed

• ND FSA are not more powerful than their deterministic relatives (this is different than with SA)
  (so what is their use?)

• Let $A_{ND} = <Q_N, I, \delta_N, q_{0N}, F_N>$ the NDFSA from which we build a FSA

• Let $A_D = <Q_D, I, \delta_D, q_{0D}, F_D>$ the FSA we intend to build
  
  $- Q_D = \mathcal{P}(Q_N)$

  $- \delta_D(q_D, i) = \bigcup_{q_N \in q_D} \delta(q_N, i)$

  $- q_{0D} = \{q_{0N}\}$

  $- F_D = \{Q \subseteq Q_N \mid \overline{Q} \cap F_N \neq \emptyset\}$
• Though it is true that for all NDFSA one can find (and build) an equivalent deterministic one

• This does not mean that using NDFSA is useless:
  – It can be easier to “design” a NDFSA 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 NDFSA with 5 states one can obtain, in the worst case, one with $2^5$ states!

• Consider NFA and DFA 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

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

• Is the index F necessary?

• 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 computations

$\text{c}_0$

$\text{c}_{11}$

$\text{c}_{12}$

$\text{c}_{13}$

$\text{c}_{21}$

$\text{c}_{22}$

$\text{c}_{23}$

$\text{c}_{24}$

$\text{c}_{25}$

$\text{c}_{26}$

$\text{c}_{31}$

$\text{c}_{32}$

$\text{c}_{kj}$

$\text{c}_{im}$
• 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 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 the acceptance.
• The problem can however be easily overcome by adopting, for instance, 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; 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 stack 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 (mathematically, 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 an 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 symbol*
  - $P \subseteq V_N^+ \times V^*$: set of rewriting rules, or productions.

$$P = \{<\alpha, \beta>\} \text{ for ease of notation write } P = \{\alpha \rightarrow \beta\}$$
Example

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

\[ \alpha \Rightarrow \beta, \alpha \in V^+, \beta \in V^* \]

\[ \leftrightarrow \]

\[ a = \alpha_1 \alpha_2 \alpha_3, \beta = \alpha_1 \beta_2 \alpha_3 \land \alpha_2 \rightarrow \beta_2 \in P \]

\( \alpha_2 \) is rewritten as \( \beta_2 \) in the context of \( < \alpha_1, \alpha_3 > \)

With reference to the previous grammar:

\[ aaBAS \Rightarrow aacCDS \]

As usual, define the reflexive and transitive closure of \( \Rightarrow \):

\[ \Rightarrow^* \]
Language generated by a grammar

\[ L(G) = \{ x \mid x \in V_T^* \land S \Rightarrow x \} \]

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

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

some may “get stuck”
A first example

\[ G_1 = \langle \{S, A, B\}, \{a, b, 0\}, P, S \rangle \]
\[ P = \{ S \to aA, A \to aS, S \to bB, B \to bS, S \to 0 \} \]

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

\[ G_2 = \langle \{S\}, \{a, b\}, P, S \rangle \]
\[ P = \{ S \to aSb \mid ab \} \quad \text{(abbreviation for } S \to aSb, S \to ab) \]

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 \( S \to ab \) with \( S \to \varepsilon \) we obtain

\[ L(G_2) = \{ a^n b^n \mid n \geq 0 \} \]
Third example: $G_3$

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

\[S \Rightarrow aACD \Rightarrow aCD \Rightarrow aBDc \Rightarrow abc\]

\[S \Rightarrow aACD \Rightarrow aaACCD \Rightarrow aaCCD \Rightarrow aaCC \downarrow\]

\[S \Rightarrow aACD \Rightarrow aaACCD \Rightarrow aaCCD \Rightarrow aaCBDc \Rightarrow\]

\[\quad aaBCDc \Rightarrow aabCDc \Rightarrow aabBDcc \Rightarrow aabbDcc \Rightarrow aabbc\]

\[S \Rightarrow aaaACCCCD \Rightarrow aaaCCCD \Rightarrow aaaCCBDc \Rightarrow aaaCCbDc \Rightarrow aaaCCbc \downarrow\]

... 

1. $S \rightarrow aACD$ and $A \rightarrow aAC$ generate as many a’s as C’s 
2. any $x \in L$ includes only terminal symbols, hence nonterminal symbols must disappear 
3. C disappears only when it “crosses” the D and then it generates a ‘B’ and a ‘c’ 
4. C’s and B’s can switch to permit the D to reach all the C’s 
5. 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 $\{a^n b^n\}$?)
- 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
• Simplest example: language *compilation*: the grammar defines the language, the automaton accepts and translates it.
• More systematically:
Classes of grammars

• Context-free grammars:
  – \( \forall \alpha \rightarrow \beta \in P, |\alpha| = 1 \), i.e., \( \alpha \) is an element of \( V_N \).
  – Context free because the rewriting of \( \alpha \) does not depend on its context (part of the string surrounding it).
  – 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)
  – \( G_1 \) e \( G_2 \) above are context-free not so for \( G_3 \).
• Regular Grammars:
  – $\forall \alpha \to \beta \in P, |\alpha| = 1, \beta \in V_T \cdot V_N \cup V_T$.
  – 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

- general $G$ (GenG & GenL)
- Context-free $G$ (CFG & CFL)
- Regular $G$ (RG & RL)
It immediately follows that:

\[ RL \subseteq CFL \subseteq GenL \]

But, are inclusions strict?

The answer comes from the comparison with automata
Relations between grammars and automata  
(with few surprises)

- Define “equivalence” between RG and FSA  
  (i.e., a FSA accepts same language that RG generates)
  - Given a FSA $A$, let $V_N = Q$, $V_T = I$, $S = <q_0>$, and,
    for each $\delta(q, i) = q'$ let
    $<q> \rightarrow i <q'>$
    furthermore 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:
    - Given a RG, let $Q = V_N \cup \{q_F\}$, $I = V_T$, $<q_0> = S$, $F = \{q_F\}$ and,
      for each $A \rightarrow bC$ let $\delta(A,b) = C$
      for each $A \rightarrow b$ let $\delta(A,b) = q_F$

NB: The FSA thus obtained is nondeterministic: much easier!
• CFG equivalent to SA (ND!)

intuitive justification (no proof: the proof is the “hart” of compiler construction)
$S \Rightarrow aSb \Rightarrow aabb$
genG equivalent to TM

• Given G let us construct (in broad lines) a ND TM, M, accepting L(G):
  – M has one memory tape
  – The input string x is on the input tape
  – The memory tape is initialized with S (better: $Z_0S$)
  – The memory tape (which, in general, will contain a string $\alpha$, $\alpha \in V^*$) 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:
\[ \alpha \Rightarrow \beta \quad \iff \quad c = < q_s, Z_0 \alpha > \uparrow \downarrow \cdots \downarrow < q_s, Z_0 \beta > \]

*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 move sequence 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*. In fact the definition of acceptance requires that \( M \) reaches an accepting configuration if and only if \( x \in L \), but does not requires that \( M \) terminates its computation without accepting (i.e., in a “rejecting state”) if \( x \notin L \)

• We are again dealing with the complement-problem and the asymmetry between solving a problem in the positive or negative sense.
• Given M (single tape, for ease of reasoning and without loss of generality) let us build (in broad lines) a G, that generates 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 \( x\$X \Rightarrow^* x \) if and only if \( x \) is accepted by M.

  – The base idea is to simulate each move of M by an immediate derivation of G:
– We represent the configuration

\[
\begin{array}{cccc}
\alpha & B & A & C & \beta \\
\end{array}
\]

\[q\]

– through the string (special cases are left as an exercise): $\alpha BqAC\beta$
– G has therefore derivations of the kind $x$\$X \Rightarrow x$\$q_0X$ (initial configuration of M)
– If, for M it is defined :
  – $\delta(q,A) = \langle q', A', R \rangle$ G includes the production $qA \rightarrow A'q'$
  – $\delta(q,A) = \langle q', A', S \rangle$ G includes the production $qA \rightarrow q'A'$
  – $\delta(q,A) = \langle q', A', L \rangle$ G includes the production $BqA \rightarrow q'BA'$

$\forall B$ in the alphabet of M (recall that M is single tape, hence it has a unique alphabet for input, memory, and output)
– This way:

- If and only if: $x\alpha BqAC\beta \Rightarrow x\alpha BA'q'C\beta$, etc.
- We finally add productions allowing $G$ to derive from $x\alpha Bq_{F}AC\beta$ a unique $x$ if—and only if—$M$ reaches an accepting configuration $(\alpha Bq_{F}AC\beta)$, by deleting whatever is at the right of $\$, $\$ included.