Theoretical Computer Science

Recap on Grammars

Lecture 12 - Manuel Mazzara

Models for languages

Models suitable to **recognize/accept, translate, compute** languages

- They "receive" an input string and process it

$\rightarrow$ **Operational models (Automata)**

Models suitable to **describe how to generate** a language

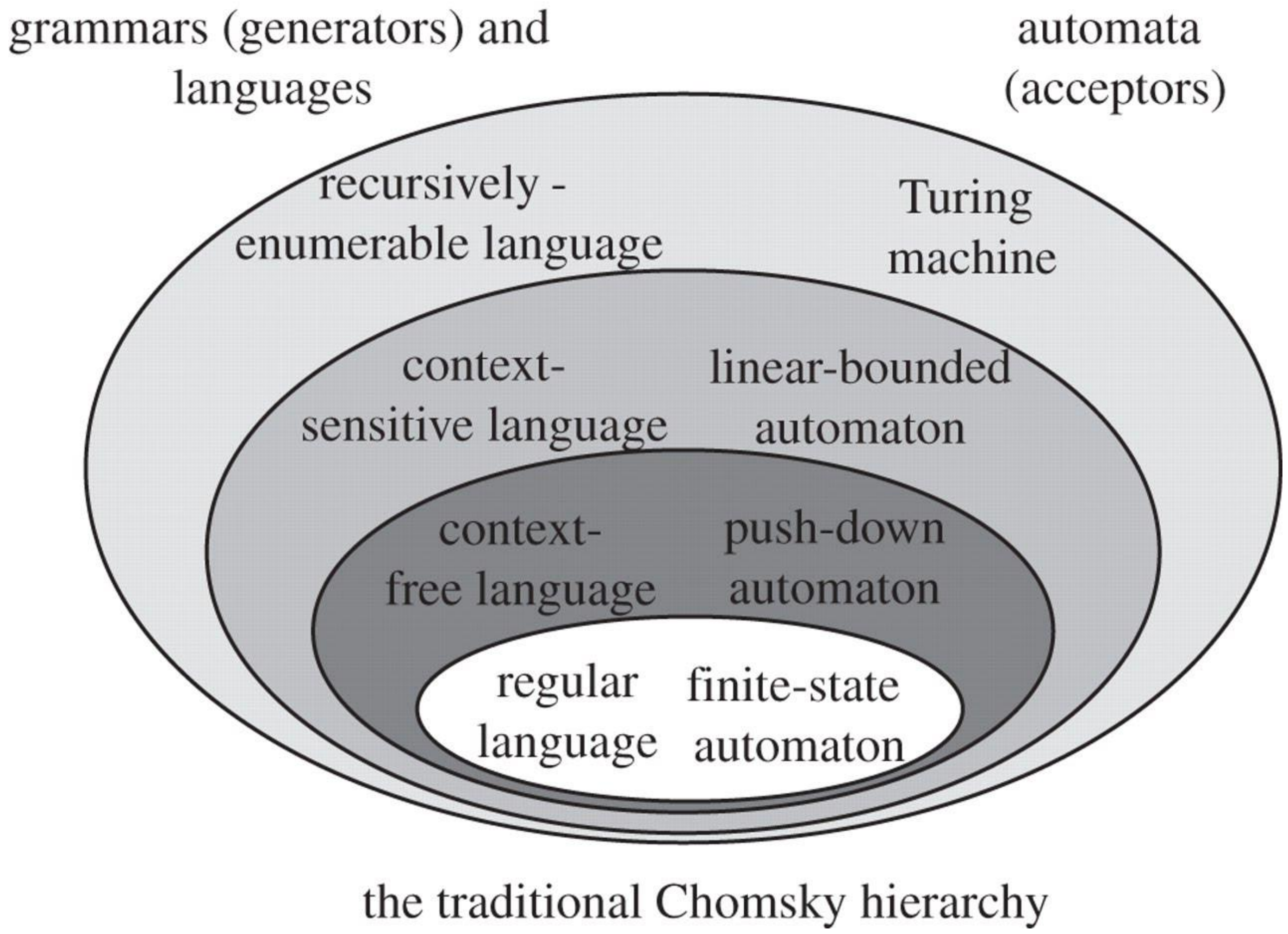
- Sets of rules to build phrases of a language

$\rightarrow$ **Generative models (Grammars)**

Automata, languages, and grammars

Chomsky hierarchy	**Grammars**	**Languages**	**Minimal automaton**
Type-0	Unrestricted	Recursively enumerable	Turing machine
Type-1	Context-sensitive	Context-sensitive	**(Linear bounded automaton)**
Type-2	Context-free	Context-free	NDPDA
Type-3	Regular	Regular	FSA

Generators vs acceptors



Theoretical Computer Science

Chomsky Hierarchy

Lecture 12 - Manuel Mazzara

Noam Chomsky

Avram Noam Chomsky (born December 7, 1928) is an American linguist, philosopher, cognitive scientist, historian, logician, social critic, and political activist. – **Wikipedia**

The "*father of modern linguistics*"

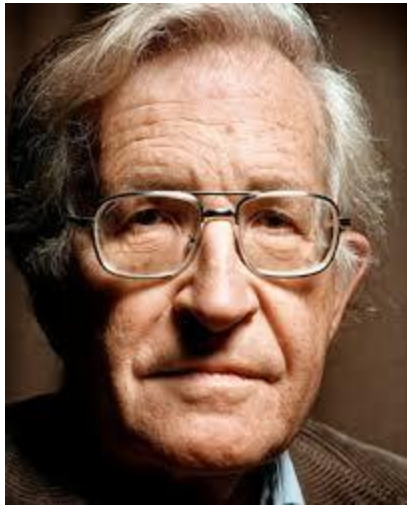
Chomsky and Grammars

- "A grammar can be regarded as **a device that enumerates the sentences** of a language"
- "A grammar of *L* can be regarded as **a function whose range is exactly *L***"

Noam Chomsky

On Certain Formal Properties of Grammars

Information and Control, Vol 2, 1959



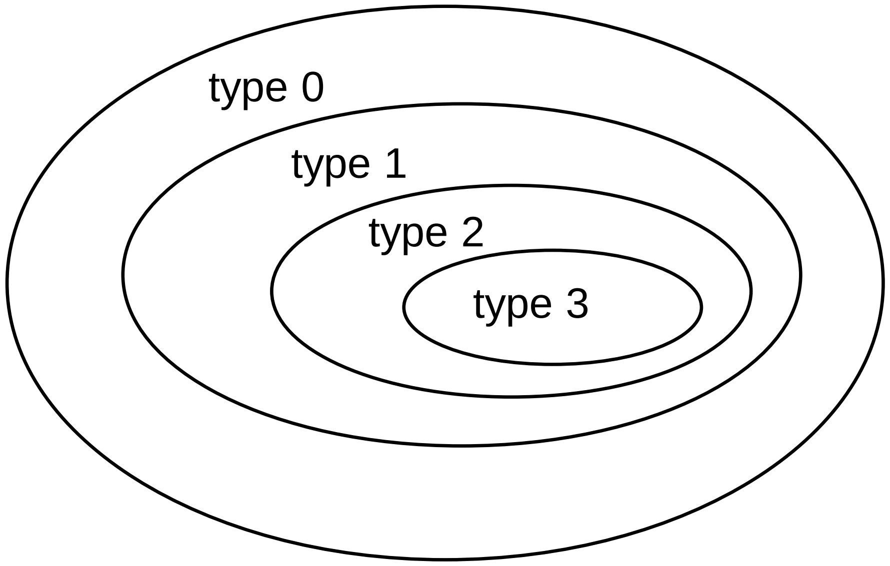
Universal Grammars

- In the 1960s Noam Chomsky proposed a new idea:
  - The **ability to learn grammar is hard-wired into the brain**
  - We are **all born with an innate knowledge of grammar**
  - Language is a **basic instinct** of humans

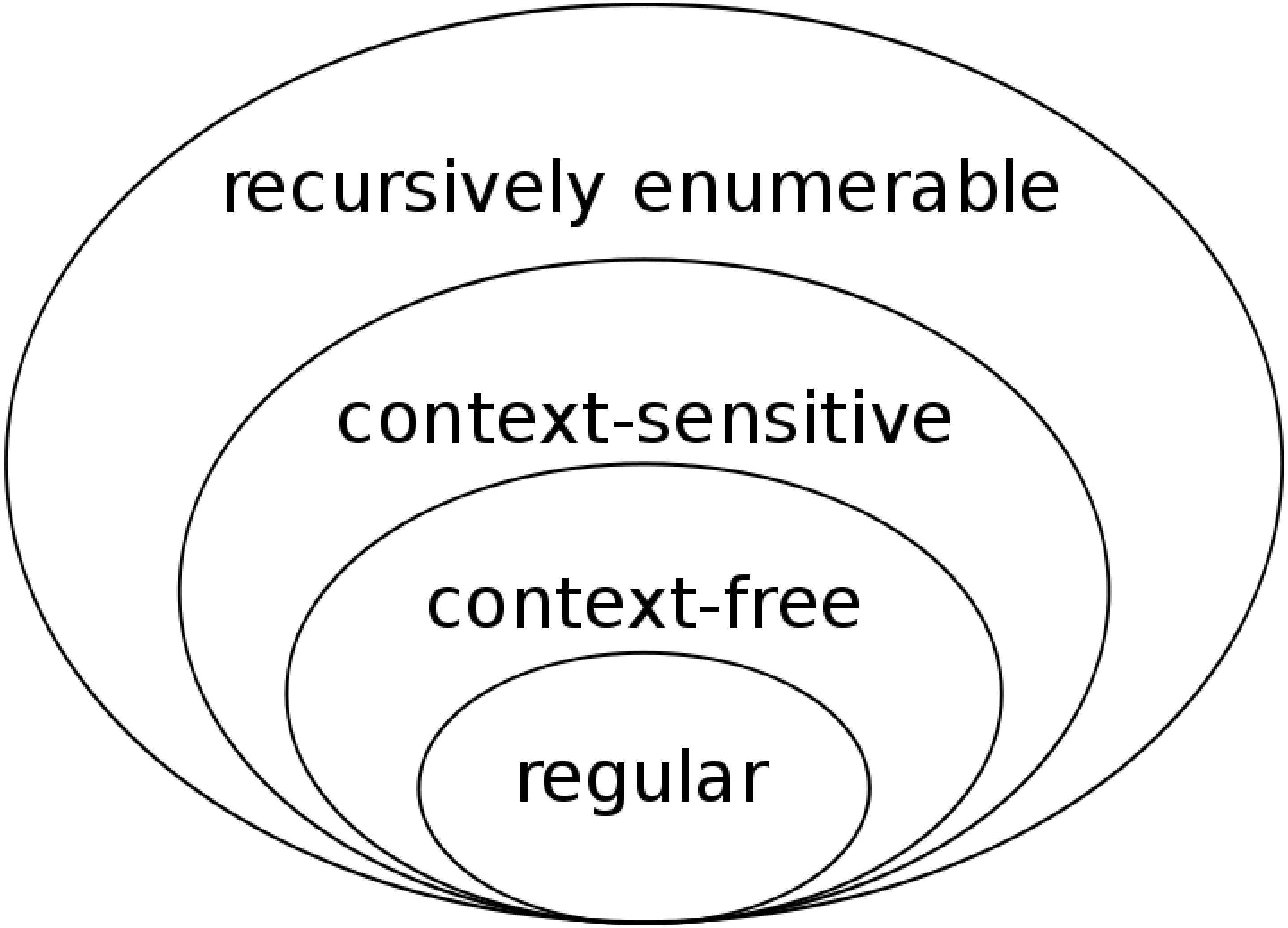
- The theory has always had widespread criticism

Chomsky hierarchy and productions form

- Grammars are classified according **to the form of their productions**
- Chomsky classified grammars in four types



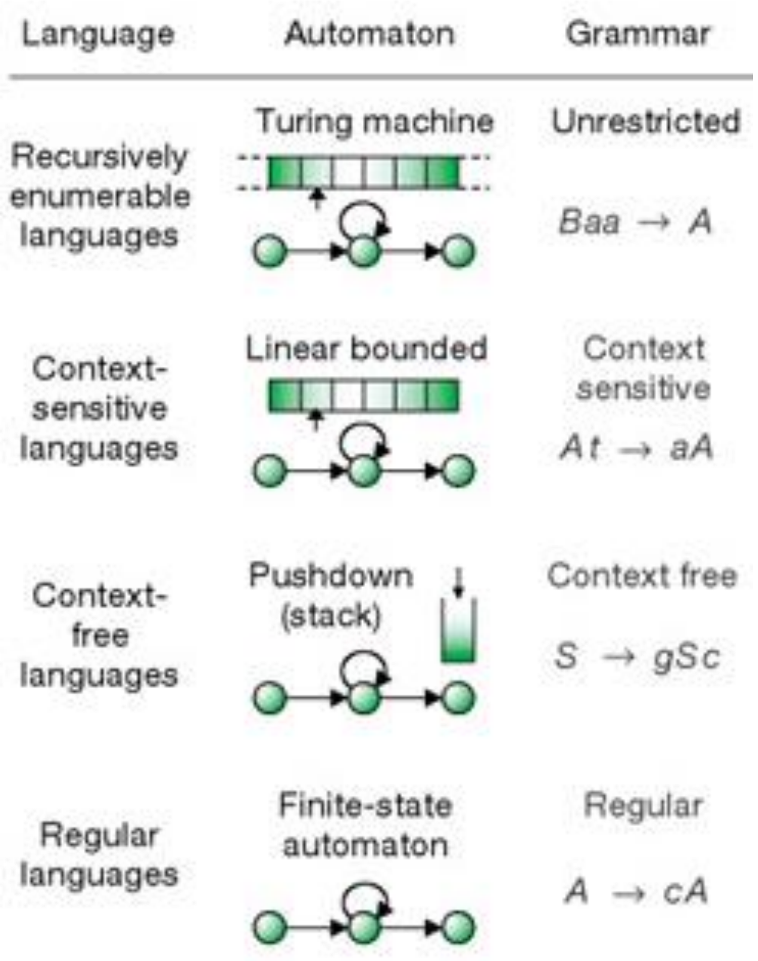
Chomsky hierarchy (named)



Automata, languages, and grammars

Chomsky hierarchy	**Grammars**	**Languages**	**Minimal automaton**
Type-0	Unrestricted	Recursively enumerable	Turing machine
Type-1	Context-sensitive	Context-sensitive	**(Linear bounded automaton)**
Type-2	Context-free	Context-free	NDPDA
Type-3	Regular	Regular	FSA

A glance forward



Productions have no restrictions

Rewrite a nonterminal according to the context

Context does not count

Rewrite a nonterminal as a terminal followed by at most one nonterminal

Unrestricted grammars (type 0)

<u>Type-0 grammars include all formal grammars</u>

$$\alpha \rightarrow \beta$$

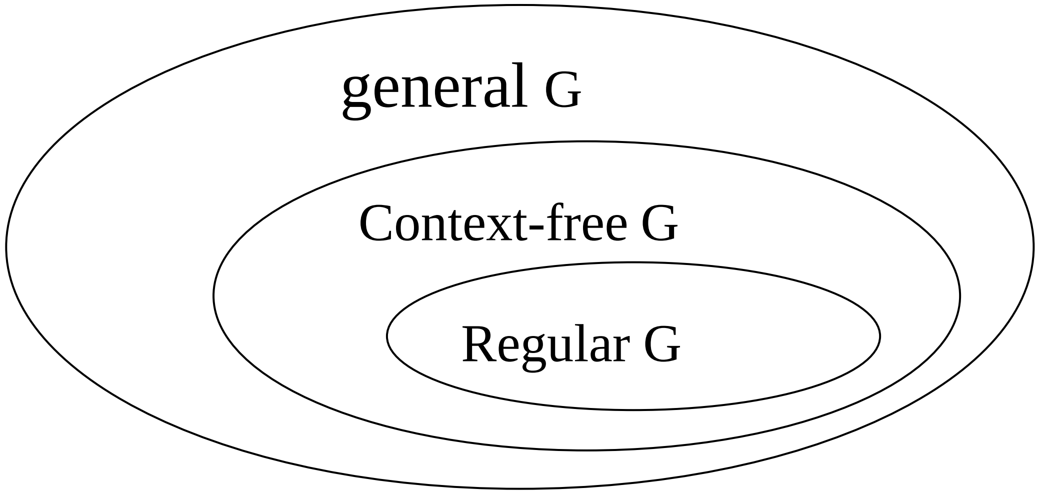
String of nonterminals and terminals

String of nonterminals and terminals

The only rrestriction on rules is **left-hand side cannot be the empty string (you cannot generate symbols out of nothing)**

Definition

- **General** (also called unrestricted) grammars are grammars without any limitation on productions
  - They correspond to type 0 in the Chomsky hierarchy
- Both context-free grammars and regular grammars are unrestricted



Example (type 0)

VN = {S, T , C, P}

VT = {a, b}

P = {S → T E

T → aTa | bTb | C

C → CP

Paa → aPa

Pab → bPa

Pba → aPb

Pbb → bPb

PaE → Ea

PbE → Eb

CE → ε

}

Context-Sensitive grammars

- **Type-1 grammars** have rules of the form $\boldsymbol{\alpha A \beta \rightarrow \alpha\ \gamma\ \beta}$, where **A** is a nonterminal and α, β and γ are strings of terminals and nonterminals.
- γ must be non-empty
- **Why are they called context-sensitive?**

Example (type 1)

- $V_N = \{S, A, B\}$
- $V_T = \{a,b,c\}$
- $P = \{S \rightarrow abc \mid aAbc$,

  $Ab \rightarrow bA$

  $Ac \rightarrow Bbcc$

  $bB \rightarrow Bb$

  $aB \rightarrow aa$

  $aB \rightarrow aaA$

  $\}$

$L=\{a^n b^n c^n \mid n \geq 1\}$

Context-free grammars

- **Type-2 grammars** are defined by rules of the form $A \rightarrow \gamma$ where **A is a nonterminal** and γ **is a string of terminals and nonterminals**

- **Why are they called context-free?**

Definition

- A grammar is called **context-free** (CFG) if
  - for each $\alpha \to \beta \in P$, we have $|\alpha| = 1$ and β is an element of $V=V_N \cup V_T$
- They are called **context-free** because the rewriting of α **does not depend on its context**
  - context = part of the string surrounding it

Example (type 2)

- $V_N = \{S\}$
- $V_T = \{a,b\}$
- $P = \{S \rightarrow aSb \mid \varepsilon \}$

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

Context-free grammars

- **CFGs are the same as the BNFs used for defining the syntax of programming languages**
  - they are well fit to define typical features of programming and natural languages
  - Regular grammars are also context-free grammars
  - But not vice versa

Right-Linear Grammars

- All productions have form:
  - A → x**B**
  - or A → x
  - x is a string of terminals

- Example:
  - S → abS
  - S → a

Left-Linear Grammars

- All productions have form:
  - A → **B**x
  - or A → *x*
  - *x* is a string of terminals
- Example:
  - S → Aab
  - A → Aab |B
  - B → A

Regular grammars

- **Type 3 grammars** restrict productions to
  - **a single nonterminal on the left-hand side**
  - **right-hand side consisting of**
    - **a string of terminals**
    - possibly followed by a single nonterminal
    - or preceded, but **not both** in the same grammar

- **right-linear XOR left-linear**

Example (type 3)

- $V_N = \{S\}$
- $V_T = \{a\}$
- $P = \{S \rightarrow aS \mid \varepsilon\}$ (right linear)

$$L=\{a^n \mid n \geq 0\}$$

Theoretical Computer Science

More on Grammars

Lecture 12 - Manuel Mazzara

Some natural questions

- What is the **practical use** of grammars?
- **What languages** can be obtained through grammars?
- What is the **relationship between automata and grammars**?
  - And between languages generated by grammars and languages accepted by automata?
  - And the Chomsky hierarchy?

Some answers

- Chomsky hierarchy can be “renamed”
  - Type 3 grammars: regular
  - Type 2 grammars: context-free
  - Type 1 grammars: context-sensitive
  - Type 0 grammars: unrestricted

- Correlations
  - **Regular grammars – regular languages – FSAs –regular expressions**
  - **Context-free grammars – context-free languages –NDPDAs (BNF)**
  - **Unrestricted grammars – recursively enumerable languages - TMs**

Automata, languages, and grammars

Chomsky hierarchy	**Grammars**	**Languages**	**Minimal automaton**
Type-0	Unrestricted	Recursively enumerable	Turing machine
Type-1	Context-sensitive	Context-sensitive	**(Linear bounded automaton)**
Type-2	Context-free	Context-free	NDPDA
Type-3	Regular	Regular	FSA

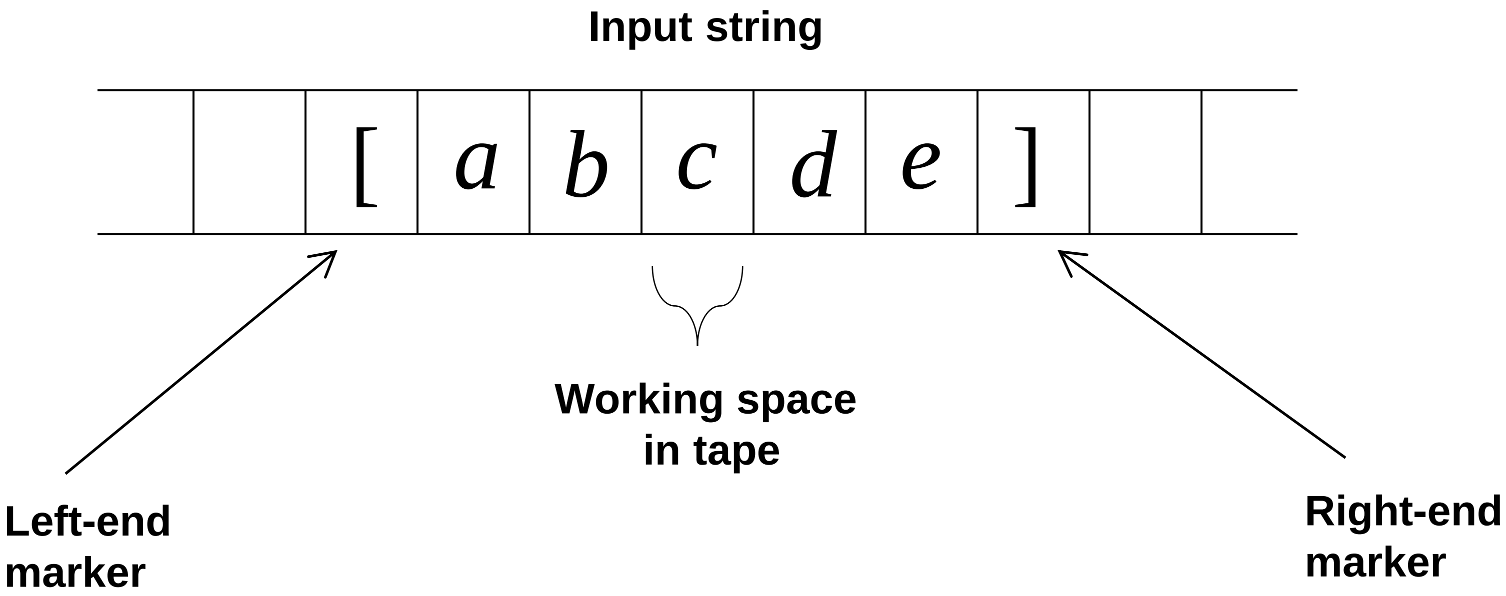
Linear bounded automaton

- A restricted form of Turing machine
- The same as Turing Machines **with one difference:**
  - **The input string tape space is the only tape space allowed to use**
- Computation is restricted to the portion of the tape containing the input (no infinite tape)

Why linear?

- Alternative definition:
  - “An LBA differs from a Turing machine in that while the tape is initially considered to have unbounded length, only a finite contiguous portion of the tape, whose **length is a linear function of the length of the initial input**, can be accessed by the read/write head; hence the name *linear bounded automaton*.”

Example



All computation happens between end markers

Automata, languages, and grammars

Chomsky hierarchy	**Grammars**	**Languages**	**Minimal automaton**
Type-0	Unrestricted	Recursively enumerable	Turing machine
Type-1	Context-sensitive	Context-sensitive	(Linear bounded automaton)
Type-2	Context-free	Context-free	NDPDA
Type-3	**Regular**	**Regular**	**FSA**

RGs and FSAs

- **Let *A* be a FSA. An equivalent RG *G* can be found constructively**.
- Equivalent means that **G** generates exactly the same language that is recognized by **A** (and vice versa)
- **Regular grammars, finite state automata and regular expressions are different models to describe the same class of languages**

Building a RG from a FSA

- If **$A = \langle Q, I, \delta, q_0, F \rangle$** then it is possible to build:
- **$G = \langle V_N, V_T, S, P \rangle$** such that
  – $V_N = Q$,
  – $V_T = I$,
  – $S = \langle q_0 \rangle$
  – For all $\delta(q, i) = q'$
    - $\langle q \rangle \rightarrow i \langle q' \rangle \in P$
    - If $q' \in F$ then $\langle q' \rangle \rightarrow \varepsilon \in P$
- **$\delta^*(q, x) = q'$ if and only if $\langle q \rangle \Rightarrow^* x \langle q' \rangle$**

Building a FSA from a RG

- If **$G = \langle V_N, V_T, S, P \rangle$** then it is possible to build:
- **$A = \langle Q, I, \delta, q_0, F \rangle$** such that
  - $Q = V_N \cup \{q_F\}$
  - $I = V_T$,
  - $\langle q_0 \rangle = S$,
  - $F = \{q_F\}$
  - For all $A \to bC$, $\delta(A,b) = C$
  - For all $A \to b$, $\delta(A,b) = q_F$

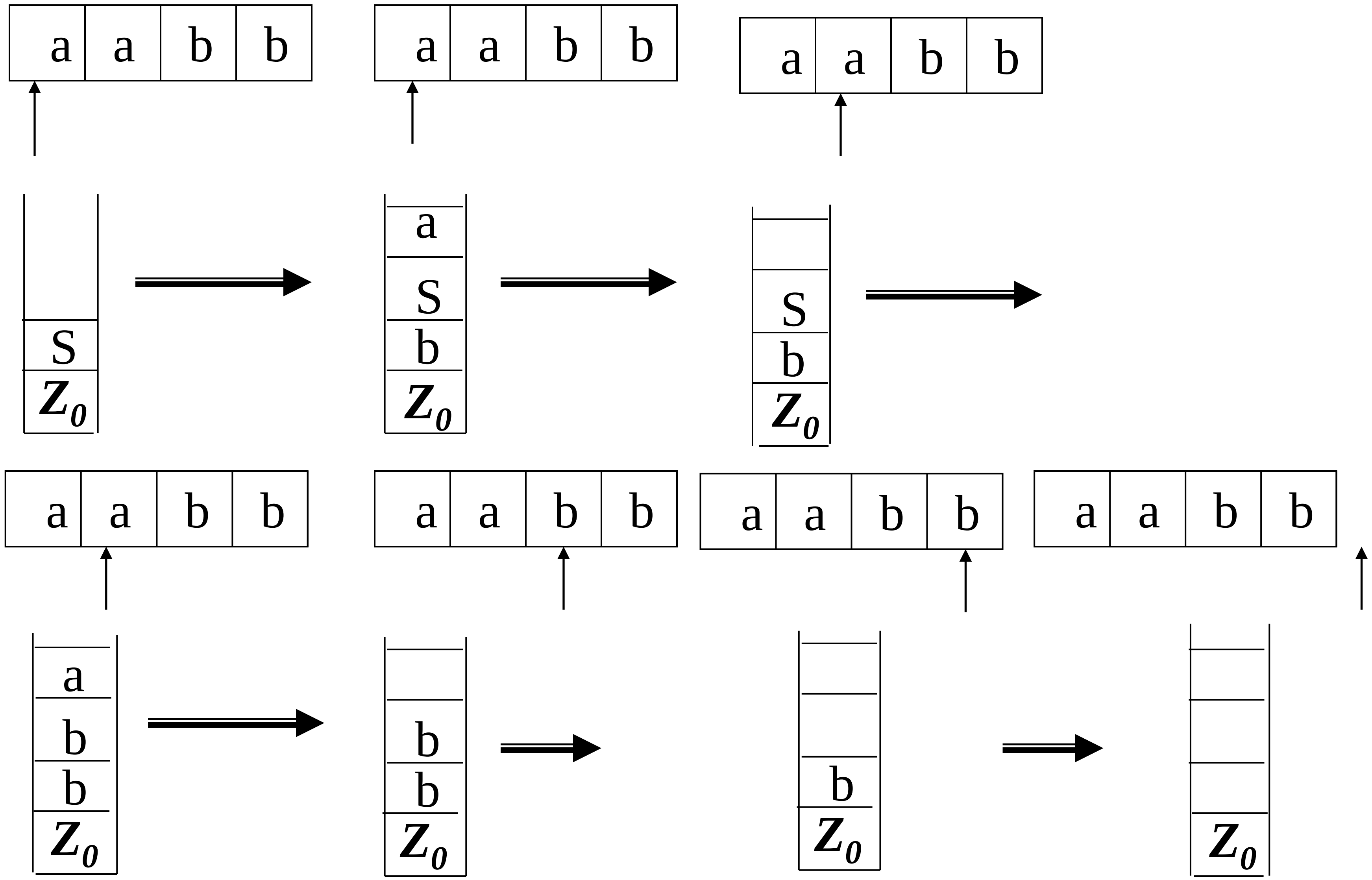
Automata, languages, and grammars

Chomsky hierarchy	Grammars	Languages	Minimal automaton
Type-0	Unrestricted	Recursively enumerable	Turing machine
Type-1	Context-sensitive	Context-sensitive	(Linear bounded automaton)
Type-2	**Context-free**	**Context-free**	**NDPDA**
Type-3	Regular	Regular	FSA

CFGs and NDPDAs

- **Context-free grammars are equivalent to nondeterministic PDAs**
- We show an **intuitive justification**
- The proof is the "core" of compiler construction

S→ aSb | ab $S \Rightarrow aSb \Rightarrow aabb$



Automata, languages, and grammars

Chomsky hierarchy	Grammars	Languages	Minimal automaton
Type-0	**Unrestricted**	**Recursively enumerable**	**Turing machine**
Type-1	Context-sensitive	Context-sensitive	(Linear bounded automaton)
Type-2	Context-free	Context-free	NDPDA
Type-3	Regular	Regular	FSA

General grammars and TMs

- General grammars (GGs) and TMs are **equivalent formalisms** (constructive proof)
  - Given a GG it is possible to build a TM that recognizes the language generated by the grammar
  - Given a TM it is possible to define a GG that generates the language accepted by the TM

  - This is left as exercise

Theory of Computation

Computability Theory

Lecture 12 - Manuel Mazzara

Post Correspondence Problem

- Introduced by **Emil Post** in 1946
- Given two lists A and B:

$$\mathbf{A = w_1, w_2, \ldots, w_k \qquad B = x_1, x_2, \ldots, x_k}$$

- Determine whether there is a sequence of one or more integers

$$\mathbf{i_1, i_2, \ldots, i_m}$$

such that:

$$\mathbf{w_1 w_{i_1} w_{i_2} \ldots w_{i_m} = x_1 x_{i_1} x_{i_2} \ldots x_{i_m}}$$

- (w_i, x_i) is called a corresponding pair.

Example

	A	B
i	w_i	x_i
1	11	1
2	1	111
3	0111	10
4	10	0

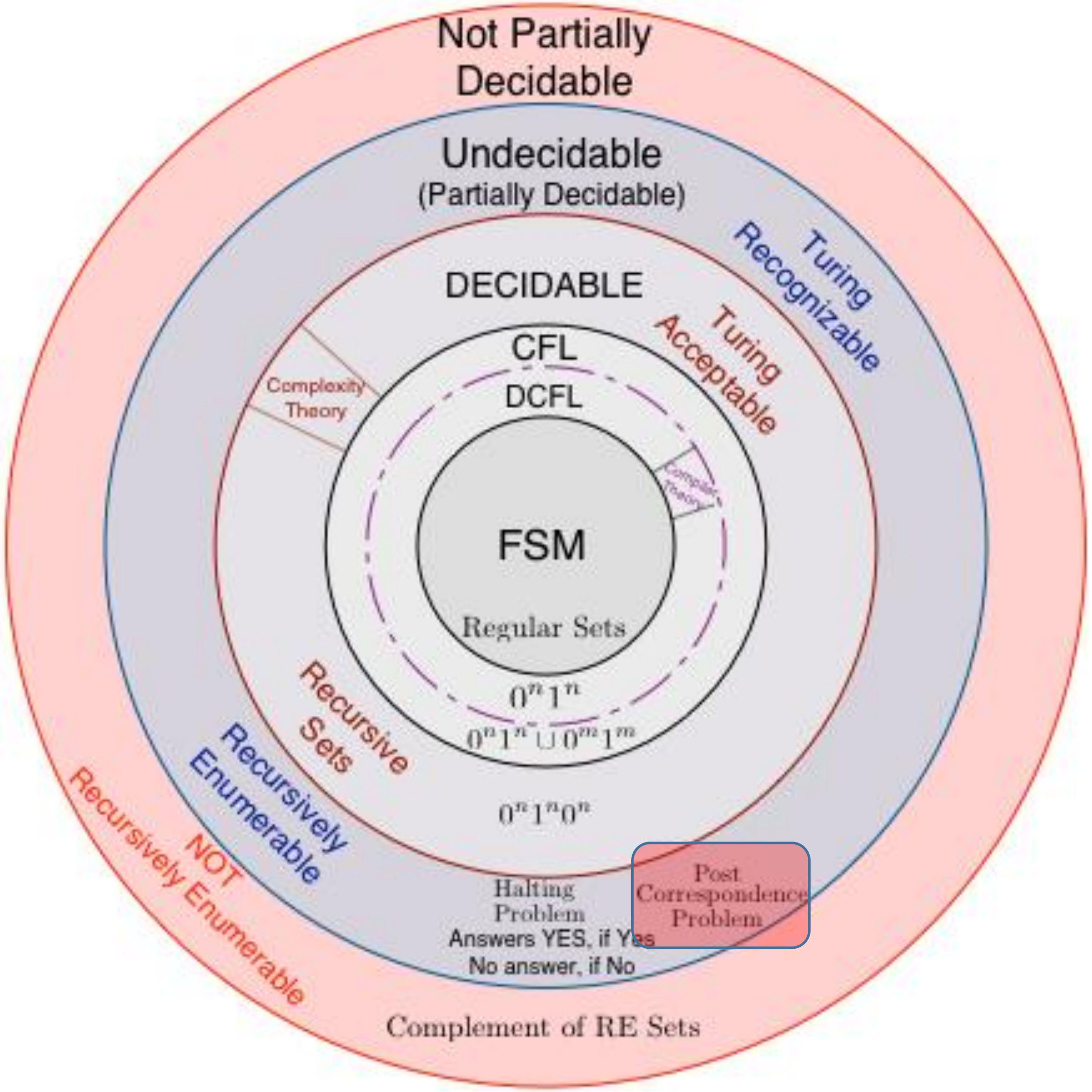
This PCP instance has a solution: 3, 2, 2, 4:

$\mathbf{w_1 w_3 w_2 w_2 w_4 = x_1 x_3 x_2 x_2 x_4 = 1101111110}$

Would you be able to write an algorithm to compute the solution?

The algorithm does not exist!

- The **Post correspondence problem** is an **undecidable problem**
- Like the HP or the decision problem, but simpler to express and often used as an example
- **What does it mean that the algorithm does not exist?**
- **Computability theory is the field of study answering such questions**


Not Partially Decidable
Undecidable
(Partially Decidable)
Turing Recognizable
DECIDABLE
Turing Acceptable
CFL
DCFL
Complexity Theory
FSM
Regular Sets
0^n1^n
$0^n1^n \cup 0^m1^m$
Recursive Sets
Recursively Enumerable
$0^n1^n0^n$
NOT Recursively Enumerable
Halting Problem
Answers YES, if Yes
No answer, if No
Post Correspondence Problem
Complement of RE Sets

Two questions of CS

- **Mathematics:** What can be computed?
- **Engineering:** How can we build computers (and then software etc.…)?

- During your studies you will cover both!
  - **Computability theory** is about the mathematical aspect

The math side

1. **Do there exist computing formalisms more powerful than TMs?**
   – Some (future) super computer?
   – Quantum computing?
2. **Can we always solve problems by means of some mechanical device?**

$\rightarrow$Do we have the answers?

$\rightarrow$Yes, we do

Computability theory (1)

- **Computability Theory** deals with the foundational mathematical basis of Computer Science

- It has major practical applications
  - **software verification**

- What can be computed? What cannot be computed? Where is the line between the two?

Computability theory (2)

- **“Speed of light of CS”**
- We study the **limits of mechanical problem solving**
- **Can we mechanically solve any definable problem?**

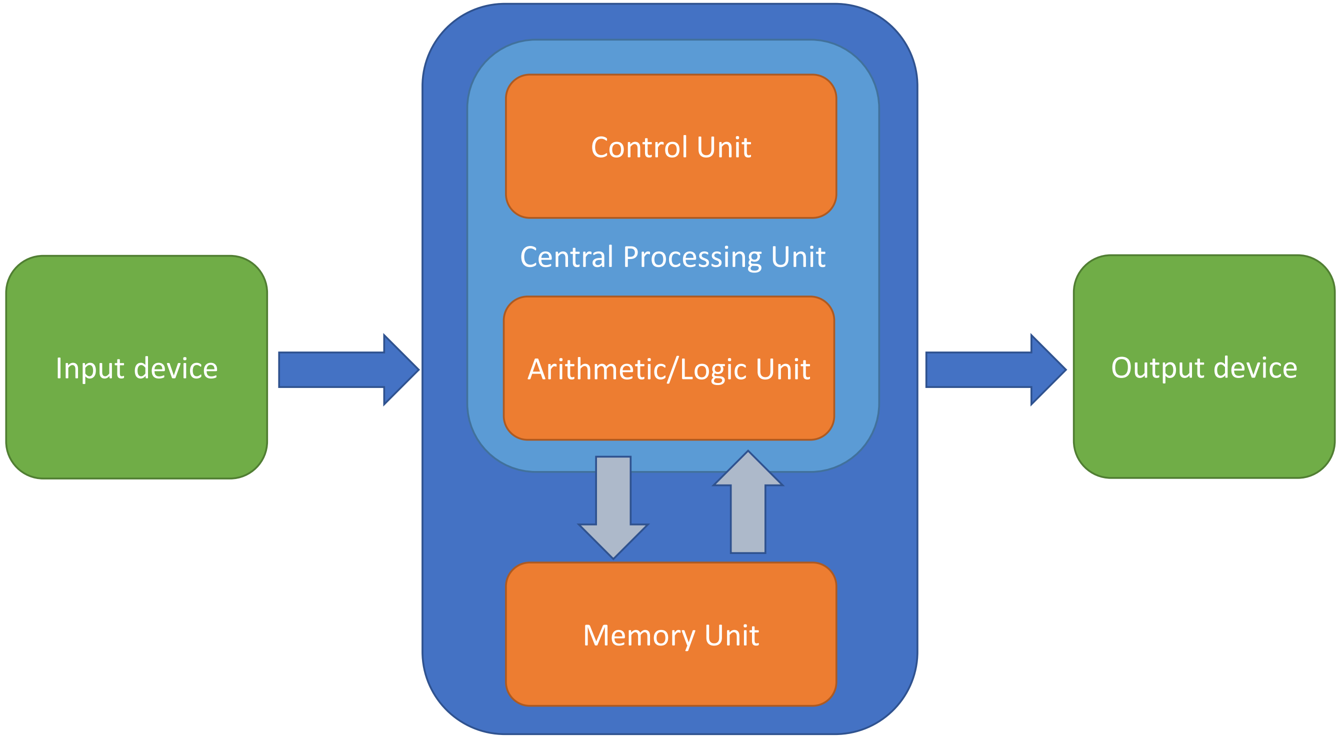
Computers are physical systems: what they can and cannot do is ultimately dictated by the **laws of physics**

What do we mean by
mechanical computation?

Turing Machine

- It is intended to emulate the human behavior when computing
- Limits of mechanical computation are in common with “human computation”
- Performance is another issue

Von Neumann Architecture



Why TMs?

TMs have the **same expressive power** as high-level programming languages

TMs are **theoretical models** not really meant for programming but for proofs and understanding

The two questions we will explore

Do there exist computing formalisms more powerful than TMs?

- **Church-Turing thesis**

Can we always solve problems by means of some mechanical device?

- **Halting problem and undecidability**