Languages and Computation

Discrete Math, Spring 2026

Konstantin Chukharev

Graph Theory

- Graphs & digraphs
- Paths & connectivity
- Trees & spanning trees
- Bipartite graphs
- Matchings & Hall's theorem
- Planarity & coloring
- Network flows

Languages & Computation

- Alphabets & formal languages
- Regular expressions
- Finite automata (DFA, NFA)
- Pumping lemma
- Context-free grammars
- Pushdown automata
- Turing machines
- Decidability & complexity

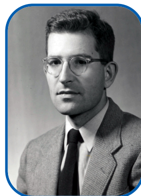
Combinatorics & Recurrences

- Counting principles
- Permutations & combinations
- Inclusion–exclusion
- Partitions & Stirling numbers
- Generating functions
- Recurrence relations
- Asymptotic analysis

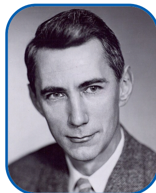
Formal Languages

"The limits of my language mean the limits of my world."

— Ludwig Wittgenstein



Noam Chomsky



Claude Shannon

Why Formal Languages?

Every non-trivial computation involves *processing structured input*:

- Compilers read *programs* (strings over some alphabet).
- Databases evaluate *queries* (strings with specific syntax).
- Network protocols parse *packets* (sequences of bytes).
- Bioinformatics analyzes *DNA* (strings over $\{A, C, G, T\}$).

The theory of formal languages gives us a *mathematical framework* for answering:

- What kinds of patterns can we *describe*?
- What kinds of patterns can we *recognize* (and how efficiently)?
- What are the *fundamental limits* of computation?

Key insight: The study of formal languages is not just about strings — it is the foundation of the *theory of computation*.

Every computational problem can be phrased as: "Given a string w, does w belong to language L?"

Basic Terminology

Definition 1: *Alphabet* Σ is a finite non-empty set of symbols.

Examples: $\Sigma_1 = \{\mathrm{a}, \mathrm{b}, \mathrm{c}\}$, $\Sigma_2 = \{0, 1\}$, $\Sigma_3 = \{$🦀, 🐈, 🦭, 🦁$\}$.

Definition 2: A *word*, or a *string*, over Σ is a *finite* sequence of symbols from Σ.

Examples: "abacaba", "10110001", "i am a word", "" (empty word ε).

Definition 3: The set of *all* finite words over the alphabet Σ is called the *Kleene star*, $\Sigma^* = \bigcup_{k=0}^{\infty} \Sigma^k$.

Definition 4: A *formal language* $L \subseteq \Sigma^*$ is a set of finite words over a finite alphabet.

Examples: $L_1 = \{0, 001, 0001, \ldots\}$, $L_2 = \{\mathrm{a}, \mathrm{aba}, \mathrm{ababa}, \mathrm{abababa}, \ldots\}$, $L_3 = \emptyset$, $L_4 = \{\varepsilon, \mathrm{ricercar}\}$.

Operations on Languages

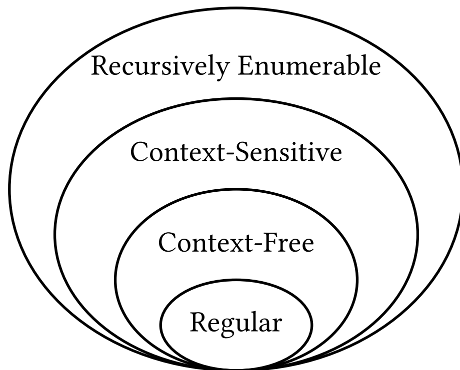
- A formal language, $L \subseteq \Sigma^*$, can be defined by:
  - an *enumeration* of words, *e.g.* $L = \{w_1, w_2, \ldots, w_n\}$
  - a *regular expression*, *e.g.* $L \triangleq \texttt{01*}$
  - a *formal grammar*, *e.g.* $L \cong G$
- *Set-theoretic* operations:
  - $L_1 \cup L_2 = \{w \mid w \in L_1 \lor w \in L_2\}$, the *union* of L_1 and L_2
  - $\overline{L} = \{w \mid w \notin L\} = \Sigma^* \setminus L$, the *complement* of L
  - $|L|$ is the *cardinality* of L
- *Concatenation*:
  - $L_1 \cdot L_2 = \{ab \mid a \in L_1, b \in L_2\}$, where ab is the concatenation of words a and b.
  - $L^k = \underbrace{L \cdot \ldots \cdot L}_{k \text{ times}} = \{w_1 w_2 \ldots w_k \mid w_i \in L\}$
  - $L^0 = \{\varepsilon\}$
- *Kleene star*: $L^* = \bigcup_{k=0}^{\infty} L^k$

Chomsky Hierarchy

Formal languages are classified by *Chomsky hierarchy* — a nested family of increasingly powerful language classes, each recognized by a correspondingly more powerful machine.

Type	Class	Machine	Example
3	Regular	Finite Automata	$\{a^n \mid n \geq 0\}$
2	Context-Free	Pushdown Automata	$\{a^n b^n \mid n \geq 0\}$
1	Context-Sensitive	Linear-Bounded TMs	$\{a^n b^n c^n \mid n \geq 0\}$
0	Recursively Enumerable	Turing Machines	$\{\langle M, w\rangle \mid M \text{ halts on } w\}$

Chomsky Hierarchy [2]



Each level of the hierarchy represents a *trade-off* between *expressive power* and *decidability*.

More expressive languages come at the cost of harder (or impossible) algorithmic questions.

Decision Problems as Languages

Any *decision problem* — a question with a "yes" or "no" answer — can be reformulated as a *language membership test*. The language encodes all inputs for which the answer is "yes".

Definition 5: A *decision problem* is a question with a "yes" or "no" answer depending on the input. Formally, the set of inputs for which the answer is "yes" is a language $L \subseteq \Sigma^*$.

Deciding the problem is equivalent to *recognizing* the language L.

Satisfiability (SAT): Given a Boolean formula φ, is it satisfiable?

$$\text{SAT} = \{\varphi \mid \varphi \text{ is a satisfiable Boolean formula}\}$$

Validity (VALID): Given a Boolean formula φ, is it a tautology?

$$\text{VALID} = \{\varphi \mid \varphi \text{ is a valid (universally true) formula}\}$$

Halting Problem (HALT): Given a TM M and input w, does M halt on w?

$$\text{HALT} = \{\langle M, w \rangle \mid \text{TM } M \text{ halts on input } w\}$$

Decision Problems as Languages [2]

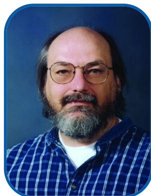
Asking “is w in L?” and asking “does the algorithm say yes on input w?” are *the same question*.

This lets us use the theory of *formal languages* to study the limits of *computation*.

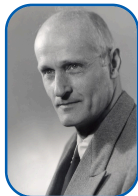
Regular Languages

“A language that doesn’t affect the way you think about programming, is not worth knowing.”

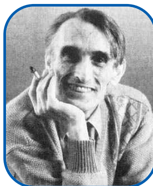
— Alan Perlis



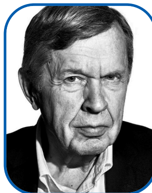
Ken Thompson



Stephen Kleene



Marcel-Paul Schützenberger



Alfred Aho

John Hopcroft

Jeffrey Ullman

Regular Languages

Definition 6: A class of regular languages REG is defined inductively:

- $\text{Reg}_0 = \{\varnothing, \{\varepsilon\}\} \cup \{\{a\} \mid a \in \Sigma\}$, the *empty* and *singleton* languages.
- $\text{Reg}_{i+1} = \text{Reg}_i \cup \{A \cup B \mid A, B \in \text{Reg}_i\} \cup \{A \cdot B \mid A, B \in \text{Reg}_i\} \cup \{A^* \mid A \in \text{Reg}_i\}$, the inductively extended $(i+1)$-th *generation* of regular languages.
- $\text{REG} = \bigcup_{k=0}^{\infty} \text{Reg}_k$, the *class* of all regular languages.

Theorem 1: REG is closed under union, concatenation, and Kleene star operations.

Proof: Let $A \in \text{Reg}_i$, $B \in \text{Reg}_j$.

- $(A \cup B) \in (\text{Reg}_i \cup \text{Reg}_j) \in \text{Reg}_{\max(i,j)+1} \subseteq \text{REG}$
- $(A \cdot B) \in (\text{Reg}_i \cdot \text{Reg}_j) \in \text{Reg}_{\max(i,j)+1} \subseteq \text{REG}$
- $A^* \in \text{Reg}_{i+1} \subseteq \text{REG}$

□

Regular Expressions

Regular languages can be composed from “smaller” regular languages.

- *Atomic* regular expressions:
  - $\varnothing$, an empty language
  - ε, a singleton language consisting of a single ε word
  - $\mathtt{a}$, a singleton language consisting of a single 1-letter word a, for each $a \in \Sigma$
- *Compound* regular expressions:
  - $R_1 R_2$, the concatenation of R_1 and R_2
  - $R_1 \mid R_2$, the union of R_1 and R_2
  - $R^* = RRR...$, the Kleene star of R
  - (R), just a bracketed expression
  - Operator precedence: $\mathtt{ab{*}c|d} \triangleq ((\mathtt{a}\ (\mathtt{b}^*))\ \mathtt{c}) \mid \mathtt{d}$

Regular Expressions: Summary Table

Language	Expression	Description
$\varnothing$		Empty language
$\{\varepsilon\}$	ε	Language with a single empty word
$\{\mathrm{a}\}$	`a`	Singleton language with a literal character "a"
A	α	Language A denoted by regex α
B	β	Language B denoted by regex β
$A \cup B$	$\alpha \mid \beta$	Union of languages A and B
$A \cdot B$	$\alpha\beta$	Concatenation of languages A and B
A^*	α^*	Kleene star of language A
A^+	α^+	Kleene plus of language A

Regular Expressions: Quick Reading

Example: `(a|bc)*` $= \{\varepsilon$, a, aa, aaa, …, bc, bcbc, bcbcbc, …, abc, bca, abca, abcbc, bcabc, …$\}$

Example: `0(10)*1` $= \{01, 0101, 010101, \ldots\}$

See also: PCRE

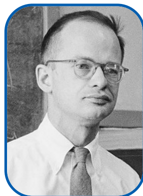
Key insight: Regular expressions describe *exactly* the regular languages. This is not obvious — it takes Kleene's Theorem (stated later) to prove it.

Finite Automata

“We may hope that machines will eventually compete with men in all purely intellectual fields.”

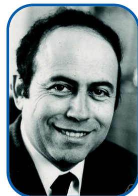
— Alan Turing

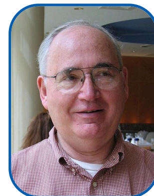
Warren McCulloch


Walter Pitts

Victor Glushkov

Janusz Brzozowski

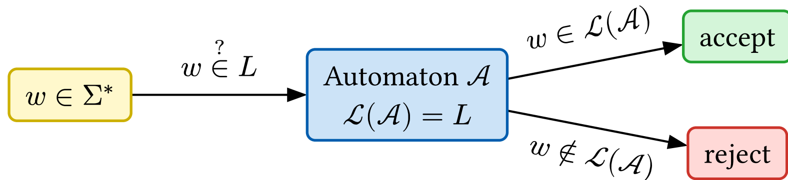

Michael Rabin


Dana Scott

Recognizers vs Transducers

There are two main types of finite-state *machines*:

1. *Acceptors* (or *recognizers*), automata that produce a binary *yes/no answer*, indicating whether or not the received input word $w \in \Sigma^*$ is *accepted*, *i.e.*, belongs to the language L recognized by the automaton.



2. *Transducers*, machines that produce an output action *for each* symbol of an input.
   - Moore machines (1956)
   - Mealy machines (1955)

In this course, we focus on *acceptors*.

Deterministic Finite Automata

Definition 7: A *Deterministic Finite Automaton* (DFA) is a 5-tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ where:

- Q is a *finite* set of states,
- Σ is an *alphabet* (finite set of input symbols),
- $\delta : Q \times \Sigma \to Q$ is a *transition function*,
- $q_0 \in Q$ is the *start* state,
- $F \subseteq Q$ is a set of *accepting* (*final*) states.

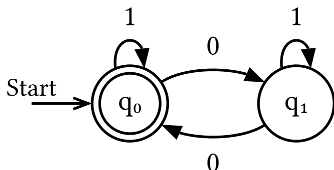
A DFA is a machine that reads an input string *one symbol at a time*, transitioning between states. After reading the entire input, the machine either *accepts* or *rejects* based on whether it ended in an accepting state.

Note: *Deterministic* means: at every state, for every symbol, there is *exactly one* transition.

DFA: Example

Example: Automaton $\mathcal{A}$ recognizing strings with an even number of 0s, $\mathcal{L}(\mathcal{A}) = \{w \in \{0,1\}^* \mid w \text{ has even number of 0s}\}$.

	0	**1**
q0	q1	q0
q1	q0	q1



Here, q_0 is the *start* (denoted by an arrow) and also the *accepting* (denoted by double circle) state.

On input 1, the machine stays.
On input 0, the machine toggles.

DFA: Computation

Definition 8: A process of *computation* by a finite-state machine $\mathcal{A}$ is a finite sequence of *configurations*, or *snapshots*. A set of all possible configurations is denoted $\mathrm{SNAP} = Q \times \Sigma^*$.

Definition 9: A *reachability relation* $\vdash$ is a binary relation over configurations:

$$\langle q, \alpha \rangle \vdash \langle r, \beta \rangle \quad \text{iff} \quad \begin{cases} \alpha = c\beta & \text{where } c \in \Sigma \\ r = \delta(q, c) \end{cases}$$

- $c_1 \vdash c_2$ means "configuration c_2 is reachable in *one step* from c_1".
- $\vdash^*$, the reflexive-transitive closure of $\vdash$, denotes "reachable in *any* number of steps".

Note: A configuration captures *everything* the machine "knows" at a given moment: its current state and the remaining input.

Automata Languages

Definition 10: A word $w \in \Sigma^*$ is *accepted* by an automaton $\mathcal{A}$ if the computation, starting in the initial configuration at state q_0 with input w, *can reach the final configuration $\langle f, \varepsilon \rangle$*, where $f \in F$ is any accepting state, and ε denotes that the input has been fully consumed.

Formally, $\mathcal{A}$ *accepts* $w \in \Sigma^*$ if $\langle q_0, w \rangle \vdash^* \langle f, \varepsilon \rangle$ for some $f \in F$.

Definition 11: The language *recognized* by an automaton $\mathcal{A}$ is a set of all words accepted by $\mathcal{A}$.

$$\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^* \mid \langle q_0, w \rangle \vdash^* \langle f, \varepsilon \rangle \text{ where } f \in F\}$$

Definition 12: The class of *automaton languages* recognized by DFAs is denoted AUT.

$$\mathrm{AUT} = \{X \mid \exists \mathcal{A} \text{ such that } \mathcal{L}(\mathcal{A}) = X\}$$

DFA Exercises

For each language below (over the alphabet $\Sigma = \{0, 1\}$), draw a DFA recognizing it:

1. $L_1 = \{101, 110\}$
2. $L_2 = \Sigma^* \setminus \{101, 110\}$
3. $L_3 = \{w \mid w \text{ starts and ends with the same bit}\}$
4. $L_4 = \{110\}^* = \{\varepsilon, 110, 110110, 110110110, \ldots\}$
5. $L_5 = \{w \mid w \text{ contains } 110 \text{ as a substring}\}$

Non-determinism

Non-deterministic Finite Automata

Definition 13: A *Non-deterministic Finite Automaton* (NFA) is a 5-tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, where
- Q is a *finite* set of states,
- Σ is an *alphabet* (finite set of input symbols),
- $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ is a *transition function*,
- $q_0 \in Q$ is an *initial* (*start*) state,
- $F \subseteq Q$ is a set of *accepting* (*final*) states.

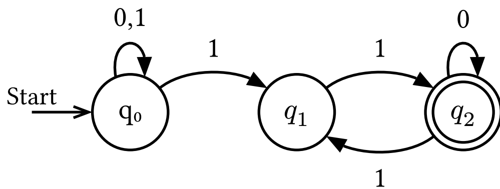
The key difference from a DFA: the transition function returns a *set* of possible next states.

Note: $\delta : (q, c) \mapsto \underbrace{\{q^{(1)}, ..., q^{(n)}\}}_{\text{non-determinism}}$

NFA: Example

The following NFA recognizes $\mathcal{L}(\mathcal{A}) = \Sigma^*(110^*)^+$ — strings containing at least one `11` followed by any number of `0`s:

	0	**1**
q0	q0	q0, q1
q1		q2
q2	q2	q1



This NFA has *two* transitions from q_0 by the symbol 1: it can go to q_0 or to q_1.

If an NFA needs to make a non-existent transition (*e.g.*, at q_1 by 0), it *dies* and that particular path rejects.

Determinism vs Non-determinism

Definition 14: A model of computation is *deterministic* if at every point in the computation, there is exactly *one choice* that can be made.

Note: The machine accepts if *that* series of choices leads to an accepting state.

Definition 15: A model of computation is *non-deterministic* if the computing machine may have *multiple decisions* that it can make at one point.

Note: The machine accepts if *any* series of choices leads to an accepting state.

Key insight: Non-determinism is not about randomness. A non-deterministic machine accepts if *there exists* at least one accepting path — it doesn't matter how unlikely that path might be.

Intuitions on Non-determinism

There are three useful ways to think about non-determinism:

1. Tree Computation

At each *decision point*, the automaton *clones* itself for each possible decision. The series of choices forms a directed, rooted *tree*. If *any* leaf is accepting, the machine accepts.

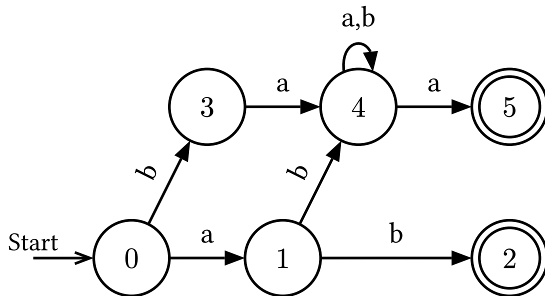
2. Perfect Guessing

A non-deterministic machine has *magic superpowers*: it can always *guess* the right sequence of choices (if one exists). No physical implementation is known.

3. Massive Parallelism

An NFA can be thought of as a machine that tries *all possibilities in parallel*, using an unlimited number of "processors". Each symbol read causes a transition on every currently active state.

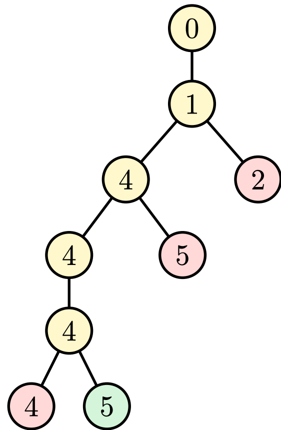
Tree Computation: Example



Input:
$w = \text{a b a b a}$

How to read this: each time there are multiple outgoing transitions, the NFA branches into multiple active computations.

Tree Computation: Accepting Branch



- At each *decision point*, the automaton *clones* itself for each possible decision.
- At the end, if *any* active accepting (green) states remain, we *accept*.

NFA Computation Model

Reachability relation for NFA is very similar to DFA's:

$$\langle q, x\rangle \vdash_{\text{DFA}} \langle r, y\rangle \quad \text{iff} \quad \begin{cases} x = cy & \text{where } c \in \Sigma \\ r = \delta(q, c) \end{cases}$$

$$\langle q, x\rangle \vdash_{\text{NFA}} \langle r, y\rangle \quad \text{iff} \quad \begin{cases} x = cy & \text{where } c \in \Sigma \\ r \in \delta(q, c) \end{cases}$$

Definition 16: An NFA *accepts* a word $w \in \Sigma^*$ iff $\langle q_0, w\rangle \vdash^* \langle f, \varepsilon\rangle$ for some $f \in F$.

Definition 17: A language *recognized* by an NFA is a set of all words accepted by the NFA.

$$\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^* \mid \langle q_0, w\rangle \vdash^* \langle f, \varepsilon\rangle, f \in F\}$$

DFAs vs NFAs

Deterministic (DFA)	**Non-Deterministic (NFA)**
Single transition per symbol	Multiple possible transitions
No ε-transitions	May have ε-transitions
Unique computation path	Multiple parallel paths
$\delta : Q \times \Sigma \to Q$	$\delta : Q \times \Sigma \to \mathcal{P}(Q)$
Easy to simulate	Hard to simulate directly
May need exponentially many states	Can be exponentially more concise

Why NFAs?

Despite having the same expressive power as DFAs, NFAs are often *exponentially more concise*.

They are also the natural output of many constructions (*e.g.*, Thompson's construction).

ε-NFA

Definition 18: An *ε-NFA* is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ with the same components as an NFA, but with a modified transition function:

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$$

From any state the machine may follow *ε-transitions* without consuming any input symbol.

Definition 19: *Epsilon closure* of a state q, denoted $E(q)$ or $\varepsilon\text{-clo}(q)$, is a set of states reachable from q by ε-transitions.

$$E(q) = \varepsilon\text{-clo}(q) = \left\{ r \in Q \;\middle|\; q \xrightarrow{\varepsilon} r \right\}$$

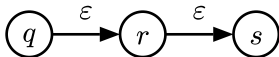
This definition can be extended to the *sets of states*. For $P \subseteq Q$:

$$E(P) = \bigcup_{q \in P} E(q)$$

ε-NFA [2]

Note: $q \in \varepsilon\text{-clo}(q)$ since each state has an *implicit* ε-loop.

Example: For the following NFA, epsilon closure of q is $\varepsilon\text{-clo}(q) = \{q, r, s\}$.



From ε-NFA to NFA

To construct an NFA from an ε-NFA:

1. Perform a *transitive closure* of ε-transitions.
   - After that, accepted words contain *no two consecutive* ε-transitions.
2. *Back-propagate* accepting states over ε-transitions.
   - After that, accepted words *do not end* with ε.
3. Perform *symbol-transition back-closure* over ε-transitions.
   - After that, accepted words *do not contain* ε-transitions.
4. *Remove* ε-transitions.
   - After that, you get a plain NFA.

Rabin–Scott Powerset Construction

Any NFA can be converted to a DFA using *Rabin–Scott subset construction*.

$\mathcal{A}_{\mathrm{N}} = (Q_{\mathrm{N}}, \Sigma, \delta_{\mathrm{N}}, q_0, F_{\mathrm{N}})$

- $Q_{\mathrm{N}} = \{q_1, q_2, \ldots, q_n\}$
- $\delta_{\mathrm{N}} : Q_{\mathrm{N}} \times \Sigma \to \mathcal{P}(Q_{\mathrm{N}})$

$\mathcal{A}_{\mathrm{D}} = (Q_{\mathrm{D}}, \Sigma, \delta_{\mathrm{D}}, \{q_0\}, F_{\mathrm{D}})$

- $Q_{\mathrm{D}} = \mathcal{P}(Q_{\mathrm{N}}) = \{\varnothing, \{q_1\}, \ldots, \{q_2, q_4, q_5\}, \ldots, Q_{\mathrm{N}}\}$
- $\delta_{\mathrm{D}} : Q_{\mathrm{D}} \times \Sigma \to Q_{\mathrm{D}}$
- $\delta_{\mathrm{D}} : (A, c) \mapsto \{r \mid \exists q \in A.\, r \in \delta_{\mathrm{N}}(q, c)\}$
- $F_{\mathrm{D}} = \{A \mid A \cap F_{\mathrm{N}} \neq \varnothing\}$

Warning: The powerset construction can produce *exponentially many* states.

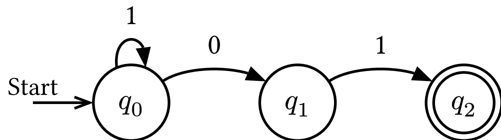
An NFA with n states can require a DFA with up to 2^n states.

In practice, many of these states are unreachable and can be pruned.

NFA to DFA: Worked Example

Let's walk through a complete example of converting an NFA to a DFA using the powerset construction.

Example: Consider the NFA $\mathcal{N}$ over alphabet $\Sigma = \{0, 1\}$ that accepts strings ending with 01:



Formally, $\mathcal{N} = (Q, \Sigma, \delta, q_0, F)$ where:

- $Q = \{q_0, q_1, q_2\}$
- $\delta(q_0, 0) = \{q_0, q_1\}$, $\delta(q_0, 1) = \{q_0\}$, $\delta(q_1, 1) = \{q_2\}$, all other transitions go to $\emptyset$
- q_0 is start state
- $F = \{q_2\}$

Step 1: Determine Reachable States

We begin with the start state of the DFA: $\{q_0\}$.

Compute transitions from $\{q_0\}$:

- On 0: $\delta_{D(\{q_0\},0)} = \delta_{N(q_0,0)} = \{q_0, q_1\}$
- On 1: $\delta_{D(\{q_0\},1)} = \delta_{N(q_0,1)} = \{q_0\}$

Now we have new states $\{q_0, q_1\}$ and $\{q_0\}$. $\{q_0\}$ is already known.

Step 2: Compute Transitions for New States

From $\{q_0, q_1\}$:

- On 0: $\delta_{N(q_0,0)} \cup \delta_{N(q_1,0)} = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$
- On 1: $\delta_{N(q_0,1)} \cup \delta_{N(q_1,1)} = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$

New state: $\{q_0, q_2\}$

From $\{q_0, q_2\}$:

- On 0: $\delta_{N(q_0,0)} \cup \delta_{N(q_2,0)} = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$
- On 1: $\delta_{N(q_0,1)} \cup \delta_{N(q_2,1)} = \{q_0\} \cup \emptyset = \{q_0\}$

No new states.

Step 3: Identify Accepting States

A DFA state is accepting if it contains any NFA accepting state:

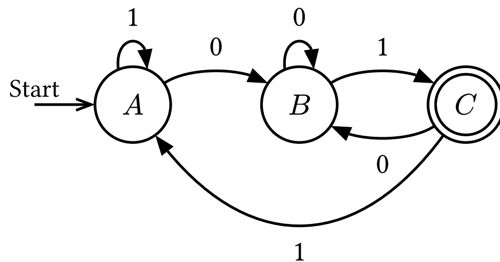
- $\{q_0\}$: contains q_0? No ($q_0 \notin F$). Not accepting.
- $\{q_0, q_1\}$: contains q_2? No. Not accepting.
- $\{q_0, q_2\}$: contains q_2? Yes ($q_2 \in F$). Accepting.

Step 4: Construct the DFA

We have three reachable states: $A = \{q_0\}$, $B = \{q_0, q_1\}$, $C = \{q_0, q_2\}$.

Transition table:

State	0	1	Accepting?
$A = \{q_0\}$	B	A	✗
$B = \{q_0, q_1\}$	B	C	✗
$C = \{q_0, q_2\}$	B	A	✓



Step 5: Verify Equivalence

Both automata accept the same language: strings ending with 01.

- The NFA has 3 states, the DFA has 3 reachable states (out of possible $2^3 = 8$).
- Unreachable states like $\{q_1\}$, $\{q_2\}$, $\{q_1, q_2\}$, $\{q_0, q_1, q_2\}$, $\varnothing$ were never generated.

Key observation: Although the powerset construction can produce exponentially many states, in practice many states are unreachable. This example shows that sometimes the resulting DFA can be as small as the original NFA.

So far, we have studied three different *representations* of languages: regular expressions, DFAs, and NFAs. A natural question arises: Are these representations equivalent? Can they express the same set of languages? Kleene's theorem answers this definitively.

Kleene's Theorem

Kleene's Theorem

Theorem 2 (Kleene): $\mathrm{REG} = \mathrm{AUT}$.

That is, a language is *regular* (definable by a regular expression) if and only if it is recognized by a *finite automaton*.

Proof *(*$\mathrm{REG} \subseteq \mathrm{AUT}$*)*: *For every regular language, there is a DFA that recognizes it.*

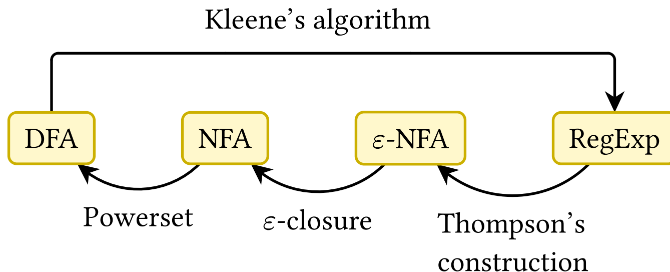
Use *Thompson's construction* to build an ε-NFA from a regular expression, and then convert it to a DFA. □

Proof *(*$\mathrm{AUT} \subseteq \mathrm{REG}$*)*: *The language recognized by a DFA is regular.*

Use *Kleene's algorithm* to construct a regular expression from an automaton. □

The Equivalence Cycle

The following diagram summarizes the conversions between the four representations. Each arrow represents a *constructive* conversion algorithm.



All four representations — DFA, NFA, ε-NFA, and regular expressions — are *equally powerful*.

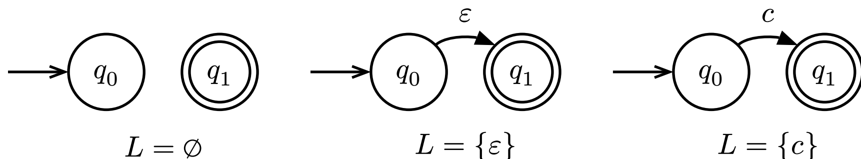
They describe exactly the same class of languages: the *regular languages*.

Thompson's Construction

Thompson's construction is a method of constructing an ε-NFA from a regular expression.

Prove REG $\subseteq$ AUT by induction over the *generation index k*. Show that $\forall k.\, \mathrm{Reg}_k \subseteq \mathrm{AUT}$.

Base: $k = 0$, construct automata for $\mathrm{Reg}_0 = \{\emptyset, \{\varepsilon\}, \{c\}$ for $c \in \Sigma\}$.



Induction step: $k > 0$, already have automata $\mathcal{A}_1, \mathcal{A}_2$ for languages $L_1, L_2 \in \mathrm{Reg}_{k-1}$.

Construct automata for:

- $L_1 \cup L_2$ — add new start state with ε-transitions to both $\mathcal{A}_1$ and $\mathcal{A}_2$
- $L_1 \cdot L_2$ — connect accepting states of $\mathcal{A}_1$ via ε-transitions to start of $\mathcal{A}_2$
- L_1^* — add new start/accept state with ε-transitions forming a loop through $\mathcal{A}_1$

Kleene's Algorithm

Definition 20: *Kleene's algorithm* is a method of constructing a regular expression from a DFA.

Let R_{ij}^k be a set of all words that take $\mathcal{A}$ from state q_i to q_j without going *through* (both entering and leaving) any state numbered higher than k. Note that i and j *can* be higher than k. Since all states are numbered 1 to n, R_{ij}^n denotes the set of all words that take q_i to q_j. We can define R_{ij}^k recursively:

$$R_{ij}^k = R_{ij}^{k-1} \cup R_{ik}^{k-1} \left(R_{kk}^{k-1}\right)^* R_{kj}^{k-1}$$

$$R_{ij}^0 = \begin{cases} \{a \mid \delta(q_i, a) = q_j\} & \text{if } i \neq j \\ \{a \mid \delta(q_i, a) = q_j\} \cup \{\varepsilon\} & \text{if } i = j \end{cases}$$

Kleene's algorithm is structurally identical to the *Floyd–Warshall* algorithm for shortest paths in graphs. Both use dynamic programming with "allowed intermediate nodes up to k".

Limits of Finite Automata

"There are more things in heaven and earth, Horatio,
than are dreamt of in your philosophy."

— William Shakespeare

The Memory Bottleneck of Finite Automata

When does a language fail to be regular? Finite automata (DFAs and NFAs) possess exactly one form of memory: **their current state**.

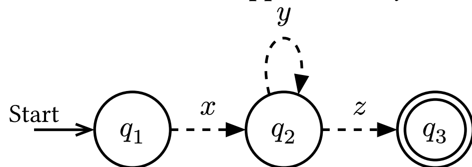
- A DFA with n states can distinguish at most n different "situations" at any point while reading a string.
- It **cannot count** indefinitely.
  (*e.g.*, verifying if there are exactly 1,000,000 zeros vs 1,000,001 zeros might require 1M states).
- It **cannot match** unbounded nested structures.
  (*e.g.*, balancing parentheses in code, checking `<html>` tags).

Key Insight: If a language requires unbounded memory to recognize (*e.g.* keeping track of an arbitrarily large number), it **cannot be regular**.

To formalize this, we exploit the simplest consequence of finite state spaces: the **Pigeonhole Principle**.

Re-visiting States

- Let D be a DFA with n states.
- Any string w accepted by D that has length at least n will force the DFA to take at least n transitions.
- Taking n transitions means visiting $n + 1$ states.
- By the *Pigeonhole Principle*, because the DFA only has n distinct states, **at least one state must be visited twice**.
- Thus, the execution path contains a **loop**. The substring of w that drives the DFA around this loop can be repeated (pumped) as many times as we want, or skipped entirely!



Informally:

- Let L be a regular language.
- If we have a string $w \in L$ that is "sufficiently long", then we can *split* the string into *three pieces* and *"pump"* the middle.
- We can write $w = xyz$ such that xy^0z, xy^1z, xy^2z, ..., xy^nz, ... are all in L.

Weak Pumping Lemma

Theorem 3 (Weak Pumping Lemma for Regular Languages): Let L be regular. Then there exists a purely structural constant $n \in \mathbb{N}$ (the **pumping length**), $n > 0$, such that for every "sufficiently long" $w \in L$ with $|w| \geq n$, there are strings x, y, z where $w = xyz$, and:

1. $y \neq \varepsilon$ (the loop can't be empty)
2. For every $i \geq 0$, $xy^iz \in L$ (the loop can be skipped or repeated)

$$\underbrace{q_0 \cdots}_{x} \underbrace{q_k \cdots}_{y} \underbrace{q_k \cdots}_{z} q_f$$

Here, x is the path to the loop, y is the loop itself, z is the path from the loop to the final state.

Example: Let $\Sigma = \{0, 1\}$ and $L = \{w \in \Sigma^* \mid w \text{ contains } 00 \text{ as a substring}\}$. Given any string of length ≥ 3 in L, we can find a substring to pump. For instance, in $w = 100$, we let $x = 1, y = 0, z = 0$. The pumped strings 10^i0 still contain "00" safely!

Example: Let $\Sigma = \{0, 1\}$ and $L = \{\varepsilon, 0, 1, 00, 01, 10, 11\}$. The weak pumping lemma trivially holds with pumping length $n = 3$. There are **no** strings in L of length ≥ 3, so the condition "for every $w \in L$ with $|w| \geq 3$..." is vacuously true!

Non-regularity: The Equality Language

Definition 21: The *equality problem* is, given two strings x and y, to decide whether $x = y$.

Example: Let $\Sigma = \{0, 1, \#\}$. We can *encode* the equality problem as a string of the form $x\#y$.

- "Is *001* equal to *110* ?" would be $001\#110$.
- "Is *11* equal to *11* ?" would be $11\#11$.

Let $\text{EQUAL} = \{w\#w \mid w \in \{0,1\}^*\}$.

Question: Is EQUAL a *regular* language?

Strategy: Assume regularity, choose $w = 0^n\#0^n$, then pump to destroy equality.

EQUAL is Not Regular

Theorem 4: EQUAL is not a regular language.

Proof: By contradiction. Assume EQUAL is regular, and let n be the pumping length.
Take $w = 0^n \# 0^n \in$ EQUAL. Write $w = xyz$ as in the weak pumping lemma.

The block y cannot contain $\#$, because pumping with $i = 0$ would remove $\#$ and produce a string outside EQUAL. Hence y lies entirely to the left or entirely to the right of $\#$, so $y = 0^k$ for some $k > 0$.

We now pump with $i = 2$:
Case 1: y is to the left of $\#$. Then $xy^2z = 0^{n+k} \# 0^n \notin$ EQUAL.
Case 2: y is to the right of $\#$. Then $xy^2z = 0^n \# 0^{n+k} \notin$ EQUAL.

In both cases we contradict the pumping lemma. Therefore EQUAL is not regular. □

A finite automaton cannot compare two unbounded strings for equality, because it has only finitely many states, *i.e.* it lacks “memory”.

The Classic Non-regular Language

Example: Consider the language $L = \{0^n 1^n \mid n \in \mathbb{N}\}$.

- $L = \{\varepsilon, 01, 0011, 000111, 00001111, \ldots\}$
- L is a classic example of a non-regular language.
- **Intuitively:** if you have *only finitely many states* in a DFA, you cannot *"remember"* an arbitrary number of 0s to match *the same* number of 1s.

How would we prove that L is non-regular?

Use the Pumping Lemma to show that L *cannot* be regular.

Pumping Lemma as a Game

The Pumping Lemma contains alternating quantifiers ($\exists$, $\forall$, $\exists$, $\forall$).

Any such statement can be framed naturally as a **two-player game**.

Think of it as a game between **You** and an **Adversary**.

- **You win** if you can break the pumping lemma conditions (proving it is **not regular**).
- **The Adversary wins** if they satisfy the conditions.

The game is played in 4 steps:

1. The adversary chooses a pumping length n.
2. You cleverly pick a long string w in the language ($|w| \geq n$).
3. The adversary maliciously splits your string $w = xyz$ such that $y \neq \varepsilon$.
4. You cleverly choose a pump count i such that $xy^iz \notin L$. (If you can't, you lose!)

Let's Play the Game: $L = \{0^n 1^n \mid n \in \mathbb{N}\}$

Adversary	You
Maliciously selects "pumping length" n (unknown)	...Wait for adversary to play...
...Wait for you to respond...	Choose a test string $w = 0^n 1^n \in L$. Its length is $2n \geq n$.
Maliciously splits $w = xyz$ such that $y \neq \varepsilon$	...Wait for adversary's split...
...Wait for you to respond...	Cleverly choose a pump multiplier i. If y has only zeros, choose $i = 0$ to delete y. If y has only ones, choose $i = 0$ to delete y. If y mixes 0 & 1, choose $i = 2$ to copy y. **In all cases**, the string is destroyed and $xy^i z \notin L$
Lose	Win

w cannot be pumped uniformly. $0^n 1^n$ is **not** regular

Formal Proof: $0^n 1^n$ is Not Regular

Theorem 5: $L = \{0^n 1^n \mid n \in \mathbb{N}\}$ is not regular.

Proof: By contradiction. Assume that L is regular.

Let n be the pumping length guaranteed by the weak pumping lemma ("there exists n…"). Consider the string $w = 0^n 1^n$. Then $|w| = 2n \geq n$ and $w \in L$, so we can write (split) $w = xyz$ such that $y \neq \varepsilon$ and for any $i \in \mathbb{N}$, we have $xy^i z \in L$.

Case 1: y consists solely of 0s. Then $xy^0 z = xz = 0^{n-|y|} 1^n$.
Since $|y| > 0$, the number of 0s is less than n, so $xz \notin L$.

Case 2: y consists solely of 1s. Then $xy^0 z = xz = 0^n 1^{n-|y|}$.
Since $|y| > 0$, the number of 1s is less than n, so $xz \notin L$.

Case 3: y consists of some 0s followed by some 1s. If we pump y with $i = 2$, then $y^2 = yy$, which places 1s from the first copy of y **before** the 0s of the second copy.
Thus the resulting string $xy^2 z$ is not even of the form $0^* 1^*$, so it cannot be in L.

Formal Proof: $0^n 1^n$ is Not Regular [2]

In all three possible ways the adversary splits the string, we found a choice of i giving a string outside L. This contradicts the weak pumping lemma. Thus L is **not regular**. □

A Word of Caution

- The pumping lemma describes a *necessary* condition for regularity.
  - Regularity $\rightarrow$ Pumping Lemma holds.
  - Pumping Lemma FAILS $\rightarrow$ Not Regular (by contrapositive).
- It is *not a sufficient* condition.
  - Pumping Lemma holds $\not\rightarrow$ Regularity.
  - There exist non-regular languages that still satisfy the pumping lemma.

Warning: "The language satisfies the pumping lemma, therefore it is regular."

This is a critical logic error! The theorem is a **one-way street** acting only as a filter for non-regularity. To **prove** that a language is regular, your only option is to construct a DFA, NFA, or Regular Expression for it!

The Full Pumping Lemma

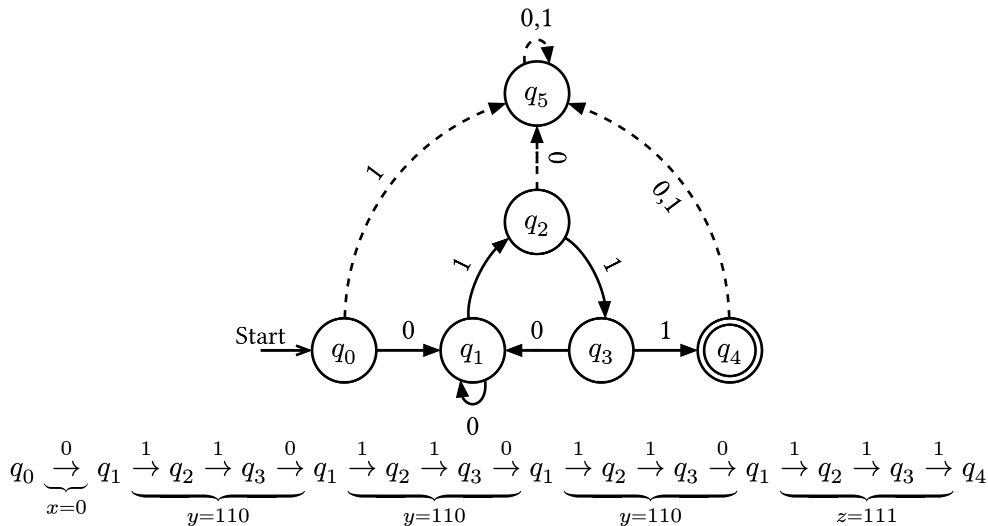
By the pigeonhole principle: any accepting path visiting $n + 1$ states must revisit some state — and this state repeat occurs *within the first n steps*.

Theorem 6 (Full Pumping Lemma for Regular Languages): Let L be a regular language. Then there exists $n \in \mathbb{N}$, $n > 0$, such that for every $w \in L$ with $|w| \geq n$, there exist strings x, y, z with:

- $w = xyz$,
- $y \neq \varepsilon$,
- $|xy| \leq n$, and
- for every $i \in \mathbb{N}$, $xy^i z \in L$.

Upgrade from the weak version: the additional constraint $|xy| \leq n$ forces the pump y to lie within the first n characters of w. This closes a loophole: some non-regular languages (such as $\{w \mid w \text{ has equal 0s and 1s}\}$) pass the *weak* lemma, but are caught by the *full* version.

Pumping Lemma: DFA Revisit



Note: The first repeated state is q_1, revisited on step 4, so $y = 110$ lies within the first $n = 6$ symbols. Here $x = 0$, and pumping y preserves acceptance until the suffix z reaches q_4.

Formal Proof: Equal 0s and 1s

Consider the language L over $\Sigma = \{0, 1\}$ of strings $w \in \Sigma^*$ that contain *an equal number* of 0s and 1s.

For example, `01` in L, `11011` not in L, `110010` in L.

Theorem 7: $L = \{w \in \{0, 1\}^* \mid w \text{ has an equal number of 0s and 1s}\}$ is *not regular*.

Proof: By contradiction. Assume that L is regular.

Let n be the pumping length guaranteed by the full pumping lemma. Consider the string $w = 0^n 1^n$. Then $|w| = 2n \geq n$ and $w \in L$. Therefore, there exist strings x, y, and z such that $w = xyz$, $|xy| \leq n$, $y \neq \varepsilon$, and for any $i \in \mathbb{N}$, we have $xy^i z \in L$.

Since $|xy| \leq n$, y must consist solely of 0s. But then $xy^2 z = 0^{n+|y|} 1^n$, and since $|y| > 0$, $xy^2 z \notin L$.

We have reached a contradiction, so our assumption was wrong and L is not regular. □

Note: This language *cannot* be shown to be non-regular using the *weak* pumping lemma alone (it passes the weak version!). The constraint $|xy| \leq n$ from the full version is essential.

Palindromes Over $\{0, 1\}$

Definition 22: A *palindrome* is a string that reads the same forwards and backwards.

Let $\text{PAL} = \{w \in \{0,1\}^* \mid w = w^R\}$ where w^R is the reversal of w.

Theorem 8: PAL is not regular.

Proof: Assume PAL is regular, with pumping length n. Choose the string $w = 0^n 1 0^n$. Then $w \in \text{PAL}$ and $|w| = 2n + 1 \geq n$.

By the full pumping lemma, there exist strings x, y, z such that $w = xyz$ and:

- $y \neq \varepsilon$ (the loop is real)
- $|xy| \leq n$ (the loop happens entirely within the first n characters)

Since the first n characters of w are all 0s, the string y must be 0^k for some $k > 0$. Choosing $i = 2$ gives $xy^2z = 0^{n+k}10^n$, which is not a palindrome. This contradicts the lemma, so PAL is not regular. □

Palindromes Over $\{0, 1\}$ [2]

Example: For $n = 3$: the string 000100 is a palindrome. The pumping lemma gives us xyz with $|xy| \leq 3$. The y part is some prefix of 000, say $y = 00$. Then:

- xyz (original): 000100 ✓ (palindrome)
- xy^2z (pumped): 00000100 ✗ (not a palindrome: 5 leading zeros but only 2 trailing zeros)

Strings with More 0s than 1s

Definition 23: Let $\mathrm{MORE0} = \{w \in \{0,1\}^* \mid w \text{ has strictly more 0s than 1s}\}$.

Theorem 9: MORE0 is not regular.

Proof: Assume regularity, and let n be the pumping length. Choose the string $w = 0^{n+1}1^n$.
- It clearly has $n + 1 > n$ zeros, so $w \in \mathrm{MORE0}$.
- Its length $2n + 1 \geq n$, so it works.

The adversary splits $w = xyz$ such that $|xy| \leq n$. The first n characters of w are all 0s. Thus, y captures exactly k zeros (where $k > 0$).

If we choose $i = 2$, the number of 0s increases, so that does not help. Instead, choose $i = 0$ and delete the block y. Our modified string is $xz = 0^{n+1-k}1^n$. Since $k \geq 1$, the remaining zeros are at most n.

The new string has at most n zeros, but it still has exactly n ones. Therefore it does not have strictly more 0 s than 1s, so it is not in MORE0. This is a contradiction. □

Strings with Unequal Counts

Definition 24: Let $\text{UNEQUAL} = \{w \in \{0,1\}^* \mid w \text{ has different number of 0s and 1s}\}$.

Theorem 10: UNEQUAL is not regular.

Proof: Note that $\text{UNEQUAL} = \overline{\text{EQUAL}}$, the complement of the language of strings with equal 0s and 1s. Since EQUAL is not regular (proved earlier), and regular languages are closed under complement, UNEQUAL cannot be regular either. (If UNEQUAL were regular, its complement EQUAL would also be regular — contradiction.) □

Contrast: A Regular Language That Seems Complex

Definition 25: Let $\text{DIV7} = \{w \in \{0,1\}^* \mid w \text{ interpreted as binary number is divisible by } 7\}$.

Theorem 11: DIV7 is regular.

Proof: Construct a DFA with 7 states representing the remainder modulo 7. As we read each bit $b \in \{0,1\}$, update the remainder: $r_{\text{new}} = (2 \cdot r_{\text{old}} + b) \bmod 7$. Accept if final remainder is 0. □

Key insight: The pumping lemma proves non-regularity. Some languages that seem complex are actually regular because they require only *bounded memory*. Here, the 7 states encode the remainder — that's enough. EQUAL requires *unbounded* memory (count of 0s), so it cannot be regular.

Myhill-Nerode Theorem: Intuition

The pumping lemma studies *loops* in long computations. The Myhill-Nerode theorem studies something even more basic: when two prefixes are *indistinguishable* to a finite automaton.

Suppose a DFA has read two different prefixes x and y and, after each of them, ends in the same state. From that moment on, the machine has exactly the same future behavior on both inputs. No suffix can reveal which prefix was read earlier, because the automaton remembers only its current state.

Key insight: If a language forces the machine to keep infinitely many different prefixes apart, then no finite automaton can recognize it.

The Equivalence Relation

Definition 26: Let $L \subseteq \Sigma^*$. Define a relation $\equiv_L$ on Σ^* by

$$x \equiv_L y \quad \text{iff} \quad \forall z \in \Sigma^*. \, (xz \in L \Longleftrightarrow yz \in L).$$

Thus $x \equiv_L y$ means that *every* suffix z has the same effect on x and on y. In other words, x and y are indistinguishable with respect to membership in L.

Example: Let $L = \{w \in \{0,1\}^* \mid w \text{ ends with } 01\}$.

- The strings 001 and 101 are equivalent: both already end with 01, so after appending any suffix z, the membership of $001z$ and $101z$ depends only on z, not on the earlier prefix.
- The strings 0 and 01 are distinguishable: take $z = \varepsilon$. Then $0z = 0 \notin L$, but $01z = 01 \in L$.

The Myhill-Nerode Theorem

Theorem 12 (Myhill-Nerode Theorem): A language L is regular if and only if the relation $\equiv_L$ has finite index, that is, only finitely many equivalence classes.

Moreover, the number of equivalence classes is exactly the number of states in the minimal DFA for L.

Proof *(idea)*: *($\Rightarrow$)* Assume L is regular, and let a DFA A with state set Q recognize it. Every string $w \in \Sigma^*$ sends the start state to one state of Q. If two strings x and y lead to the same state, then for every suffix z the automaton behaves identically on xz and yz. Hence $x \equiv_L y$. Therefore there can be at most $|Q|$ equivalence classes.

($\Leftarrow$) Now assume $\equiv_L$ has only finitely many equivalence classes. Build a DFA whose states are those classes. The start state is the class of ε. On symbol a, move from the class of x to the class of xa. Accept exactly those classes containing strings from L. This automaton recognizes L, so L is regular. □

Connection to Minimal DFAs

The theorem does more than characterize regular languages: it explains what the states of the minimal DFA *mean*. Each state corresponds to one equivalence class of prefixes.

Definition 27: To construct the minimal DFA for L:

1. Compute the equivalence classes of $\equiv_L$.
2. Use these classes as the states.
3. Take the class of ε as the start state.
4. Mark a class as accepting iff it contains a string from L.
5. On input a, send the class of x to the class of xa.

Example: Let $L = \{w \mid w \text{ has an odd number of 1s}\}$ over $\{0, 1\}$. There are exactly two equivalence classes:

- prefixes with an even number of 1s;
- prefixes with an odd number of 1s.

Therefore the minimal DFA has exactly two states.

Proof via Infinite Distinguishability

To prove that a language is *not* regular using Myhill-Nerode, it is enough to exhibit infinitely many pairwise distinguishable strings.

Definition 28: A family of strings $x_0, x_1, x_2, \ldots$ is *pairwise L-distinguishable* if for every $i \neq j$ there exists a suffix z such that exactly one of $x_i z$ and $x_j z$ belongs to L.

Theorem 13: If L contains an infinite pairwise L-distinguishable family, then $\equiv_L$ has infinite index. Consequently, L is not regular.

Proof: If the strings are pairwise distinguishable, then no two of them can belong to the same equivalence class of $\equiv_L$. Hence infinitely many different strings yield infinitely many different equivalence classes. By the Myhill-Nerode theorem, a regular language can have only finitely many such classes. Therefore L is not regular. □

Proof via Infinite Distinguishability [2]

Example ($\{0^n 1^n \mid n \geq 0\}$ is not regular): Consider the infinite family $x_i = 0^i$ for $i \in \mathbb{N}$.

Take two distinct indices $i < j$. Choose the suffix $z = 1^i$. Then

- $x_i z = 0^i 1^i \in L$,
- $x_j z = 0^j 1^i \notin L$.

Thus x_i and x_j are distinguishable. Since this works for every pair $i \neq j$, the family is infinite and pairwise distinguishable. Therefore L is not regular.

Example ($\{ww \mid w \in \{0,1\}^*\}$ is not regular): Consider the family $x_i = 0^i 1$ for $i \in \mathbb{N}$.

Take two indices $i < j$ and choose the suffix $z = 0^i 1$. Then

- $x_i z = 0^i 1 0^i 1 = (0^i 1)(0^i 1) \in L$,
- $x_j z = 0^j 1 0^i 1 \notin L$.

Hence the strings x_i are pairwise distinguishable, so the language is not regular.

Comparison: Pumping Lemma vs Myhill-Nerode

Method	Strength	Application
Pumping Lemma	Necessary condition only	Can disprove regularity, cannot prove it
Myhill-Nerode	Necessary and sufficient	Can both prove and disprove regularity
Pumping Lemma	Constructive counterexample	Adversary splits, you choose pump value
Myhill-Nerode	Structural characterization	Exhibit infinite distinguishable set
Pumping Lemma	Based on pigeonhole principle	Focuses on long strings in the language
Myhill-Nerode	Based on equivalence relations	Focuses on distinguishing extensions

Why this matters: While the pumping lemma is taught first due to its simpler structure, Myhill-Nerode provides deeper insight into *why* some languages aren't regular and connects directly to the minimal automaton structure.

Visualizing Equivalence Classes

For a regular language, the equivalence classes correspond to states in the minimal DFA:

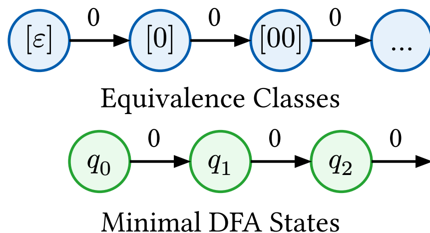


Diagram Part	Meaning
Top row: [ε], [0], [00], ...	Myhill-Nerode equivalence classes of prefixes
Bottom row: q_0, q_1, q_2	States of the minimal DFA
Arrows labeled 0	Appending symbol 0 and moving to the class of the extended prefix

Note: The same principle works for every symbol in the alphabet; only symbol 0 is drawn to keep the picture readable.

Closure and Decision Properties

Closure of Regular Languages

Theorem 14: The class REG is closed under all the following operations:

1. The *union* of two regular languages is regular.
2. The *intersection* of two regular languages is regular.
3. The *complement* of a regular language is regular.
4. The *difference* of two regular languages is regular.
5. The *reversal* of a regular language is regular.
6. The *Kleene star* of a regular language is regular.
7. The *concatenation* of regular languages is regular.
8. A *homomorphism* (substitution of strings for symbols) of a regular language is regular.
9. The *inverse homomorphism* of a regular language is regular.

Closure properties are a *powerful tool* for proving languages are (or are not) regular. Instead of building automata from scratch, we can *compose* known regular languages using these operations.

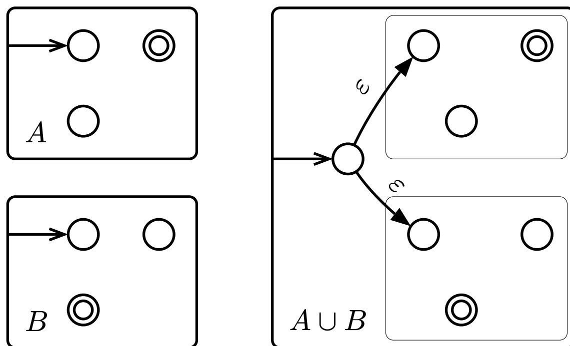
Closure under Union

Theorem 15: If L and M are regular languages, then so is their union $L \cup M$.

Proof: Since L and M are regular, they have regular expressions, *i.e.* $L = \mathcal{L}(R)$ and $M = \mathcal{L}(S)$.

Then $L \cup M = \mathcal{L}(R + S)$ by the definition of the union $(+)$ operator for regular expressions. □

Note: This proof is "algebraic" (via regular expressions). The picture below gives the automata-construction intuition.

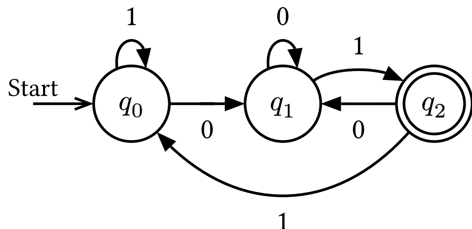


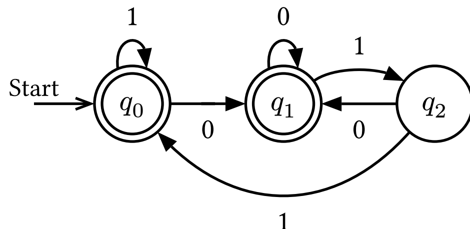
Closure under Complement

Theorem 16: If L is a regular language over the alphabet Σ, then its complement $\overline{L} = \Sigma^* - L$ is also a regular language.

Proof: Let $L = \mathcal{L}(A)$ for some DFA $A = (Q, \Sigma, \delta, q_0, F)$. Then $\overline{L} = \mathcal{L}(B)$, where B is the DFA $(Q, \Sigma, \delta, q_0, Q - F)$. That is, B is exactly like A, but with the accepting states *flipped*. Then w is in $\overline{L}$ if and only if $\delta(q_0, w)$ is in $Q - F$, which occurs if and only if w is not in $\mathcal{L}(A)$. $\square$

Example: The DFA A below on the left accepts only the strings of $0'$s and $1'$s that end in `01`, $\mathcal{L}(A) =$ `(0+1)*01`. The complement of $\mathcal{L}(A)$ is all strings of $0'$s and $1'$s that *do not* end in `01`. Below on the right is the automaton for $\{0, 1\}^* - \mathcal{L}(A)$.





Closure under Intersection

Theorem 17: If L and M are regular languages, then so is their intersection $L \cap M$.

Proof *(simple)*: $L \cap M = \overline{\overline{L} \cup \overline{M}}$. □

Proof: We can directly construct a "product" DFA for the intersection of two regular languages.

Let L and M be the languages of automata $A_L = (Q_L, \Sigma, \delta_L, q_L, F_L)$ and $A_M = (Q_M, \Sigma, \delta_M, q_M, F_M)$.

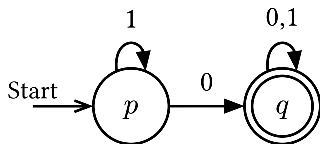
For $L \cap M$, we construct the automaton A that simulates both A_L and A_M. The states of A are the product of the states of A_L and A_M. The initial state is (q_L, q_M), and the accepting states are $F_L \times F_M$. The transitions: $\delta(\langle p, q \rangle, c) = \langle \delta_L(p, c), \delta_M(q, c) \rangle$.

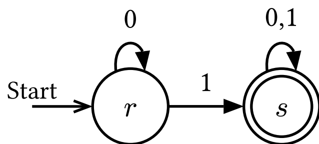
A accepts w iff both A_L and A_M accept w. □

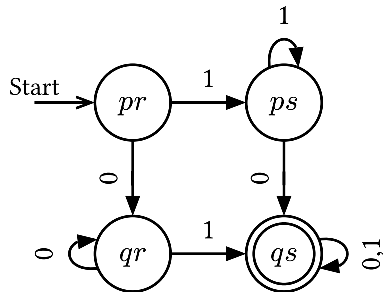
Note: De Morgan's identity gives a short proof; the product construction gives an explicit algorithm.

Intersection Closure: Product Construction

Example: The first automaton accepts all strings that *have a 0*. The second accepts all strings that *have a 1*. The product automaton accepts the *intersection*: all strings that *have both a 0 and a 1*.







Closure under Difference and Reversal

Theorem 18: If L and M are regular languages, then so is their difference $L - M$.

Proof: $L - M = L \cap \overline{M}$. By previous theorems, $\overline{M}$ is regular, and $L \cap \overline{M}$ is also regular. □

Definition 29: The *reversal* of a string $w = a_1 a_2 \ldots a_n$ is the string $w^R = a_n a_{n-1} \ldots a_1$.

Definition 30: The *reversal* of a language L is the language $L^R = \{w^R \mid w \in L\}$.

Theorem 19: If L is a regular language, then so is its reversal L^R.

Proof: Structural induction on the regular expression E defining L:

- *Basis:* If E is ε, $\emptyset$, or a, then $E^R = E$.
- *Induction:* $E = E_1 + E_2 \Rightarrow E^R = E_1^R + E_2^R$; $E = E_1 E_2 \Rightarrow E^R = E_2^R E_1^R$; $E = E_1^* \Rightarrow E^R = \left(E_1^R\right)^*$. □

Decision Properties

Converting among representations

- ε-closure: $\mathcal{O}(n^3)$
- ε-NFA to DFA: $n^3 2^n$
- DFA to ε-NFA: $\mathcal{O}(n)$
- ε-NFA to RegEx: $\mathcal{O}(n^3 4^n)$
- RegEx to ε-NFA: $\mathcal{O}(n)$

Representation changes are computable but can be exponentially expensive in the worst case.

Decidable Questions

Decidable questions about REG

1. Is the language *empty*? $\mathcal{O}(n^2)$
2. Is the language *finite*? $\mathcal{O}(n^2)$
3. Is w *in* the language? $\mathcal{O}(|w|)$ for DFAs
4. Is $L \subseteq M$? Decidable
5. Is $L = M$? Decidable

Threshold Theorems

Theorem 20: The language L accepted by a finite automaton with n states is *non-empty* iff the finite automaton accepts a word of length less than n.

Theorem 21: The language L accepted by a finite automaton M with n states is *infinite* iff the automaton accepts some word of length l, where $n \leq l < 2n$.

Summary: Regular Languages

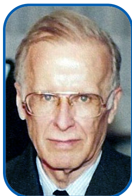
1. **Equivalence** (Kleene's Theorem): DFA $=$ NFA $=$ ε-NFA $=$ Regular Expression. Different representations, same class.
2. **Pumping Lemma:** if L is regular with DFA of n states, every $w \in L$ with $|w| \geq n$ can be pumped: $w = xyz$, $|y| \geq 1$, $|xy| \leq n$, $xy^i z \in L$ for all $i \geq 0$. A *necessary* condition only.
3. **Myhill–Nerode Theorem:** L is regular iff the equivalence $x \equiv_L y$ ("x and y lead to the same future") has *finitely many classes*. This is both necessary and sufficient, and characterizes the minimal DFA.
4. **Closure:** regular languages are closed under union, intersection, complement, concatenation, Kleene star, reversal, homomorphism, and inverse homomorphism.
5. **All key questions are decidable:** emptiness, finiteness, membership, subset, equality.

Hard ceiling: a DFA has only *constant memory* (finitely many states). It cannot count, match, or track unbounded structure. Any language requiring counting or matching at arbitrary depth is *not* regular. Canonical witnesses: $\{a^n b^n\}$, $\{ww\}$, balanced parentheses.

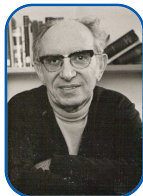
Beyond Regular Languages

"A grammar is a device that specifies the infinite set of well-formed sentences of a language and gives a structural description of each."

— Noam Chomsky

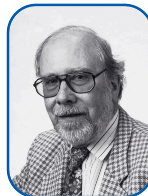

John Backus

Peter Naur


Yehoshua Bar-Hillel

Aad van Wijngaarden

Andrey Terekhov


Niklaus Wirth

John McCarthy

What Regular Languages Cannot Do

We have seen several non-regular languages:

- $L = \{0^n 1^n \mid n \in \mathbb{N}\}$ – requires *counting* to match 0s and 1s.
- $\text{EQUAL} = \{w\#w \mid w \in \{0,1\}^*\}$ – requires *remembering* an entire string.
- $L = \{w \in \{0,1\}^* \mid w \text{ has equal number of 0s and 1s}\}$ – requires a *counter*.

The common pattern: regular languages cannot *count* beyond a bounded amount. A DFA has only *finitely many states*, so it cannot track *unbounded* quantities.

Fundamental limitation: A finite automaton is a machine with *constant memory*. It can remember only a *bounded* amount of information about the input it has read.

Upgrade: give the machine a *stack* (last-in first-out memory). This gives the next level of computational power in the Chomsky hierarchy: *context-free languages*.

Context-Free Languages

Regular languages cannot describe $\{a^n b^n \mid n \geq 0\}$: any DFA only has finite memory.

The key idea is to allow *recursive* rules — a grammar that can expand itself to arbitrary depth.

Definition 31: A *context-free grammar* (CFG) is a 4-tuple $G = (V, \Sigma, R, S)$ where:

- V is a finite set of *variables* (non-terminals),
- Σ is a finite set of *terminals* (disjoint from V),
- $R \subseteq V \times (V \cup \Sigma)^*$ is a finite set of *production rules* $A \to \alpha$,
- $S \in V$ is the *start symbol*.

The language of G is the set of all terminal strings derivable from S:

$$\mathcal{L}(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$$

Example: Grammar G: $S \to 0S1 \mid \varepsilon$ generates $L = \{0^n 1^n \mid n \geq 0\}$.

$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 000S111 \Rightarrow 000\varepsilon111 = 000111$

Grammar Notation: BNF and EBNF

Backus–Naur Form (BNF) is the standard notation for writing CFGs in language specifications — it is exactly the same formalism as CFGs, just a different syntax.

Definition 32: A *BNF rule* has the form:

$$\langle \text{symbol} \rangle \quad ::= \quad \alpha_1 \quad | \quad \alpha_2 \quad | \quad \dots \quad | \quad \alpha_k$$

- $\langle \text{symbol} \rangle$ is a *non-terminal* (written in angle brackets); corresponds to a CFG variable.
- Each α_i is a sequence of *terminals* (quoted strings) and non-terminals.
- "::=" means "is defined as".
- "|" separates alternatives.

Example: BNF grammar for arithmetic expressions (from the ALGOL 60 report, Backus and Naur, 1960):

```
<expr>   ::= <expr> "+" <term>  |  <term>
<term>   ::= <term> "*" <factor>  |  <factor>
<factor> ::= "(" <expr> ")"  |  <id>
```

Grammar Notation: BNF and EBNF [2]

The stratified structure encodes *precedence*: `*` binds tighter than `+`. This is exactly the unambiguous grammar $E \to E + T \mid T, T \to T \times F \mid F, F \to (E) \mid \texttt{id}$.

Extended BNF (EBNF) adds shorthand notation — no new expressive power, but much more readable:

EBNF syntax	Meaning	BNF equivalent
$[\alpha]$	Optional: zero or one α	$S \to \alpha \mid \varepsilon$
$\{\alpha\}$ or α^*	Repetition: zero or more α	$S \to \alpha S \mid \varepsilon$
α^+	One or more α	$S \to \alpha\alpha^*$
$(\alpha_1 \mid \alpha_2)$	Grouping with alternatives	Inline alternative

Where you'll encounter BNF/EBNF: language reference manuals (Python, Rust, C), RFC protocol specifications, ISO SQL standard, JSON schema. Every modern compiler processes a grammar defined in BNF or EBNF.

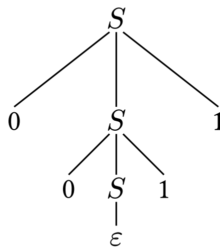
CFG: Derivations and Parse Trees

Definition 33: A *derivation step* $\alpha A \beta \Rightarrow \alpha\gamma\beta$ applies production $A \to \gamma \in R$, replacing one variable A. *Derives* $\Rightarrow^*$ is the reflexive-transitive closure: zero or more steps. A *leftmost derivation* always replaces the leftmost variable first.

The same derivation information can be rendered as a *parse tree*:

Derivation of 0011 from $S \to 0S1 \mid \varepsilon$:

$$\begin{aligned} S &\Rightarrow 0S1 \\ &\Rightarrow 00S11 \\ &\Rightarrow 00\varepsilon 11 = 0011 \end{aligned}$$



Each step expands one node. The *leaves* (left to right) spell out the derived word.

Note: Compilers use parse trees as the *abstract syntax tree* (AST) of source code. Different derivation *orders* (leftmost, rightmost) produce identical parse trees in unambiguous grammars.

CFG: More Examples

Productions	Language	Application
$S \to 0S1 \mid \varepsilon$	$\{0^n1^n \mid n \geq 0\}$	Classic matched pairs
$S \to aSa \mid bSb \mid a \mid b \mid \varepsilon$	Palindromes over $\{a, b\}^*$	Hashing, bioinformatics
$S \to SS \mid (\ S \ ) \mid \varepsilon$	Balanced parentheses $\{(), (()), ()(), ...\}$	Lisp, expression parsers
$E \to E + T \mid T$ $T \to T \times F \mid F$ $F \to (E) \mid \texttt{id}$	Arithmetic expressions with correct precedence	Every compiler (real PLs)
$S \to aSb \mid bSa \mid SS \mid \varepsilon$	$\{w \in \{a, b\}^* \mid \#_a(w) = \#_b(w)\}$	Balance counting

Pattern: A grammar variable acts as a "placeholder" carrying context through recursive structure — exactly what a *stack* does at runtime. This deep connection is Theorem: CFG = PDA.

Ambiguous Grammars

Definition 34: A grammar G is *ambiguous* if some word $w \in \mathcal{L}(G)$ has two distinct *leftmost derivations* (equivalently, two distinct parse trees).

Example: Grammar $E \to E + E \mid E \times E \mid (E) \mid \texttt{id}$ is *ambiguous*. The word $\texttt{id} + \texttt{id} \times \texttt{id}$ has two parse trees with **different** root operators:

- **Tree A** ($\times$ at root): interprets as $(\texttt{id} + \texttt{id}) \times \texttt{id}$ — wrong precedence!
- **Tree B** ($+$ at root): interprets as $\texttt{id} + (\texttt{id} \times \texttt{id})$ — correct!

Both are valid derivations in $E \to E + E \mid E \times E \mid \ldots$, so the grammar cannot determine which.

The fix: *stratify* the grammar so that higher-precedence operators are at deeper nesting levels.

$$E \to E + T \mid T, \quad T \to T \times F \mid F, \quad F \to (E) \mid \texttt{id}$$

Here T (term) binds tighter than E (expression): $\times$ before $+$. The stratified grammar has exactly *one* parse tree per expression, reflecting the intended meaning.

Consequences of Ambiguous Grammars

Inherently ambiguous languages: A *context-free language* L is *inherently ambiguous* if **every** CFG for L is ambiguous. This means no clever reformulation of the grammar can remove the ambiguity.

Classic example: $L = \{a^i b^j c^k \mid i = j\} \cup \{a^i b^j c^k \mid j = k\}$. Any CFG for this union must handle both constraints simultaneously, leading to unavoidable ambiguity on words like $a^n b^n c^n$ (which are in the language for *both* reasons).

Compiler engineering: Parser generators (yacc, Bison, ANTLR) *reject* or *warn* on ambiguous grammars. Ambiguity manifests as shift/reduce or reduce/reduce *conflicts* in LR parsing tables. Resolving them requires either grammar rewriting or explicit precedence/associativity declarations.

Decidability: "Is grammar G ambiguous?" is *undecidable* — there is no algorithm that can check all grammars. Modern tools use heuristics and partial analyses.

Chomsky Normal Form

Definition 35: A CFG G is in *Chomsky Normal Form* (CNF) if every production rule has one of two forms:

- $A \to BC$ where $B, C \in V$ (two variables), or
- $A \to a$ where $a \in \Sigma$ (a single terminal).

The only ε-production allowed is $S \to \varepsilon$ (only if $\varepsilon \in \mathcal{L}(G)$), and S must not appear on any rule's right-hand side.

Theorem 22: Every context-free language has a CNF grammar.

CNF is a canonical form: it looks restrictive (no rule has more than two symbols on the right), but any CFL can be expressed in it without changing the language.

Chomsky Normal Form [2]

Why CNF exists: Many algorithms and proofs about CFLs become significantly simpler when the grammar has a fixed, uniform shape.

Two key applications:

1. **CYK membership algorithm:** $\mathcal{O}(n^3)$ dynamic programming, only works on CNF grammars.
2. **Pumping Lemma proof:** The repeated variable on a long parse tree path is found via binary tree height bounds.

CNF Properties

In a CNF parse tree, every internal node is either a *binary branching* ($A \to BC$) or a *terminal* ($A \to a$). This gives a clean structural guarantee:

Theorem 23: In a CNF parse tree for a word of length n, there are *exactly* $2n - 1$ internal nodes.

This means: the tree height is at most $2n - 1$, and a tree with height $\geq |V| + 1$ must contain a repeated variable on some root-to-leaf path (by Pigeonhole) — the key to the Pumping Lemma proof.

CYK Algorithm

CYK algorithm (Cocke–Younger–Kasami): given CNF grammar G and string $w = a_1 \dots a_n$, fill a triangular table $T[i][j]$ = set of variables that derive $a_i a_{i+1} \dots a_j$.

- Base: $T[i][i] = \{A \mid A \to a_i \in R\}$
- Step: $T[i][j] = \{A \mid A \to BC \in R, B \in T[i][k], C \in T[k+1][j] \text{ for some } k\}$
- Accept iff $S \in T[1][n]$.

Running time: $\mathcal{O}(n^3 \cdot |G|)$.

Historical note: CYK was the first polynomial-time CFL membership algorithm, developed independently by Cocke (1960), Younger (1967), and Kasami (1966). It is the CFL analogue of Floyd–Warshall for shortest paths.

CNF Conversion Algorithm

Any CFG can be mechanically transformed into CNF in five steps (applied in order):

1. **New start symbol.** Add $S_0 \to S$ and make S_0 the start. Ensures S never appears on any RHS, satisfying the CNF constraint.
2. **Eliminate ε-productions.** Find all *nullable* variables: those that can derive ε in one or more steps. For every rule $A \to \ldots B \ldots$ where B is nullable, add a copy of the rule with B omitted. Then remove all $A \to \varepsilon$ rules (except $S_0 \to \varepsilon$ if $\varepsilon \in \mathcal{L}(G)$).
3. **Eliminate unit productions.** A *unit production* is $A \to B$ where B is a single variable. For each such rule, add $A \to \alpha$ for every $B \to \alpha$ (non-unit rule). Remove all $A \to B$ rules. Repeat until none remain.
4. **Binarize long rules.** A rule with $k \geq 3$ symbols on the RHS violates CNF. Replace $A \to X_1 X_2 \ldots X_k$ with a chain: $A \to X_1 A_1, \quad A_1 \to X_2 A_2, \quad \ldots, \quad A_{k-2} \to X_{k-1} X_k$ Each A_i is a fresh variable introduced for this rule.
5. **Isolate terminals in mixed rules.** For each terminal a appearing in a rule body of length ≥ 2, introduce a fresh variable U_a with production $U_a \to a$, and replace every occurrence of a in rule bodies by U_a.

CNF Conversion Algorithm [2]

Steps 1–5 are applied in order; each preserves the language. The result is a CNF grammar for the same CFL. The grammar may be larger, but structurally uniform.

CNF Conversion Walkthrough

Example *(Convert G: $S \to aSb \mid ab$ to CNF.)*:

Step 1 (new start): no S on any RHS already, so skip.

Step 2 (ε): neither S rule derives ε directly. Since S is not nullable, skip.

Step 3 (unit productions): no unit rules. Skip.

Step 4 (binarize): $S \to aSb$ has length 3. Introduce A_1:

$$S \to aA_1, \quad A_1 \to Sb$$

Rule $S \to ab$ has length 2 (fine as-is, for now).

Step 5 (isolate terminals): in $S \to aA_1$, a is a terminal in a two-symbol body. Introduce $U_a \to a$ and $U_b \to b$:

$$S \to U_a A_1 \mid U_a U_b, \quad A_1 \to SU_b, \quad U_a \to a, \quad U_b \to b$$

This is the final CNF grammar. Check: every production is $A \to BC$ or $A \to a$. ✓

Pushdown Automata

Definition 36: A *Pushdown Automaton* (PDA) is a 7-tuple $\mathcal{P} = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ where:

- Q is a finite set of states,
- Σ is the *input alphabet*,
- Γ is the *stack alphabet*,
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma^*)$ is the *transition function*,
- $q_0 \in Q$ is the *start state*,
- $Z_0 \in \Gamma$ is the *initial stack symbol* (bottom-of-stack marker),
- $F \subseteq Q$ is the set of *accepting states*.

A PDA is an NFA augmented with an unbounded *stack* that it can push to and pop from.

Pushdown Automata [2]

Note: Each transition reads:

- the current *state*,
- the current *input symbol* (or ε — a "free move"),
- the *top of the stack*,

and produces: a new *state* and a string to *replace* the top stack symbol with. A "pop" is modeled by replacing the top symbol with ε (nothing).

Example: A PDA for $L = \{0^n 1^n \mid n \geq 0\}$:

- On reading 0: push a marker \$ onto the stack.
- On reading the first 1: switch to "popping" mode.
- On reading 1 in popping mode: pop a \$.
- Accept when the input is exhausted and the stack contains only Z_0.

PDA Computation Model

A *configuration* (q, w, γ) captures the complete machine state: $q \in Q$ is the current state, $w \in \Sigma^*$ is remaining input, $\gamma \in \Gamma^*$ is the stack (top on left).

Definition 37: The *step relation* of a PDA:

$$(q, aw, Z\gamma) \vdash (p, w, \alpha\gamma) \quad \text{iff} \quad (p, \alpha) \in \delta(q, a, Z)$$

Epsilon moves: $(q, w, Z\gamma) \vdash (p, w, \alpha\gamma)$ iff $(p, \alpha) \in \delta(q, \varepsilon, Z)$.

$\vdash^*$ is the reflexive-transitive closure (zero or more steps).

Definition 38: Two equivalent *acceptance modes* (both define the same class — CFL):

- **By final state:** $\mathcal{L}(\mathcal{P}) = \{w \mid (q_0, w, Z_0) \vdash^* (f, \varepsilon, \gamma), f \in F\}$
- **By empty stack:** $\mathcal{L}(\mathcal{P}) = \{w \mid (q_0, w, Z_0) \vdash^* (q, \varepsilon, \varepsilon)\}$

Note: PDAs are *non-deterministic* by definition. *Deterministic PDAs* (DPDAs) recognize a strict subclass of CFLs. Example: palindromes require non-determinism — a DPDA cannot recognize them.

PDA Computation Model [2]

LL and LR parsing is a *deterministic* simulation of the PDA for a specific grammar. Parser generators (yacc, ANTLR, Bison) automate exactly this construction.

PDA Trace for $0^n 1^n$

Let's trace the PDA for $L = \{0^n 1^n \mid n \geq 0\}$ on input `0011`:

Step	State	Input remaining	Stack (top ←)	Action
0	q_0	`0011`	Z_0	Initial configuration
1	q_0	`011`	$\$Z_0$	Read `0`, push \$
2	q_0	`11`	$\$\$Z_0$	Read `0`, push \$
3	q_1	`1`	$\$Z_0$	Read `1`, pop \$ (switch to pop mode)
4	q_1		Z_0	Read `1`, pop \$
5	q_{acc}		Z_0	ε-move, **ACCEPT**

Why this works: each `0` pushes a marker; each `1` pops one. If the counts match, the stack returns to exactly Z_0 when input is consumed. If there are more `0`s than `1`s, excess \$ remain. If more `1`s, the stack underflows (no \$ to pop $\Rightarrow$ reject).

PDA Trace for $0^n 1^n$ [2]

Note: The PDA non-deterministically guesses **when** to switch from pushing to popping. For $0^n 1^n$, a deterministic PDA also works: switch when the first `1` appears.

The Context-Free Hierarchy

Theorem 24: A language is context-free iff it is recognized by some pushdown automaton.

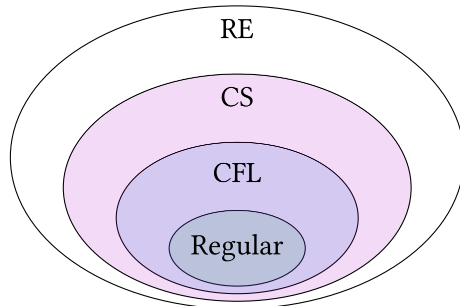
Context-free languages are *strictly more powerful* than regular languages:

- Every regular language is context-free: a DFA is a PDA that ignores its stack entirely.
- $\{a^n b^n \mid n \geq 0\}$ is context-free (by PDA above) but *not* regular (by pumping lemma).

But CFLs still have limits:

- $\{a^n b^n c^n \mid n \geq 0\}$ is **not** context-free.
- Context-sensitive languages (linear-bounded automata) lie above CFLs.

The Chomsky hierarchy captures this nested structure precisely.



Big picture: Regular ⊂ Context-Free ⊂ Context-Sensitive ⊂ Recursive (Decidable) ⊂ RE.

Each level strictly contains the one below, with concrete witness languages separating them.

CFG ⇔ PDA Equivalence

Theorem 25: $\mathcal{L}$ is context-free iff $\mathcal{L} = \mathcal{L}(\mathcal{P})$ for some PDA $\mathcal{P}$.

Proof *(CFG ⇒ PDA)*: Given CFG G, construct a PDA simulating *leftmost derivation*:

1. *Push* the start symbol S onto the stack.
2. While the stack is non-empty:
   - If top is a *variable* A: non-deterministically choose a production $A \to \alpha$ and replace the top with α (ε-move).
   - If top is a *terminal* a: read a from input and pop the top.
3. Accept when the stack and input are both empty.

This PDA non-deterministically simulates all leftmost derivations of G. □

The forward direction (CFG ⇒ PDA) is used by LL/LR parsing: the parser *simulates* the leftmost derivation non-deterministically (with look-ahead to resolve choices). GLL and GLR parsers handle the non-determinism explicitly.

CFG $\Leftrightarrow$ PDA Equivalence [2]

Proof *(PDA $\Rightarrow$ CFG)*: Given PDA $\mathcal{P}$, construct a CFG where each variable $[pXq]$ represents: "$\mathcal{P}$ can go from state p with X on top to state q, emptying X". The construction is mechanical but tedious; the key insight is that each PDA transition translates into one or two grammar productions. □

Pumping Lemma for Context-Free Languages

The CFL counterpart of the Pumping Lemma for regular languages, proved using CNF trees.

Theorem 26 (Pumping Lemma for CFLs (Bar-Hillel, Perles, Shamir, 1961)): Let L be a context-free language. Then there exists $n \geq 1$ such that for every $w \in L$ with $|w| \geq n$, there exist strings u, v, x, y, z with $w = uvxyz$ satisfying:

1. $|vy| \geq 1$ (the "pump" is non-empty),
2. $|vxy| \leq n$ (the pump is not too long),
3. $uv^ixy^iz \in L$ for all $i \geq 0$ (pumping keeps the string in L).

Compared to the regular pumping lemma ($w = xyz$, pump y), the CFL version pumps *two* substrings v and y *simultaneously*. This reflects the stack: each time the outer variable expands, it contributes one segment v on the left and one segment y on the right.

Example: Grammar $S \to 0S1 \mid \varepsilon$ generates $\{0^n1^n\}$. Taking $w = 0^n1^n$: a valid split is $u = \varepsilon$, $v = 0^k$, $x = \varepsilon$, $y = 1^k$, $z = 0^{n-k}1^{n-k}$. Then $uv^ixy^iz = 0^{n-k+ik}1^{n-k+ik} \in L$ for all $i \geq 0$. ✓

CFL Pumping Lemma: Proof Idea

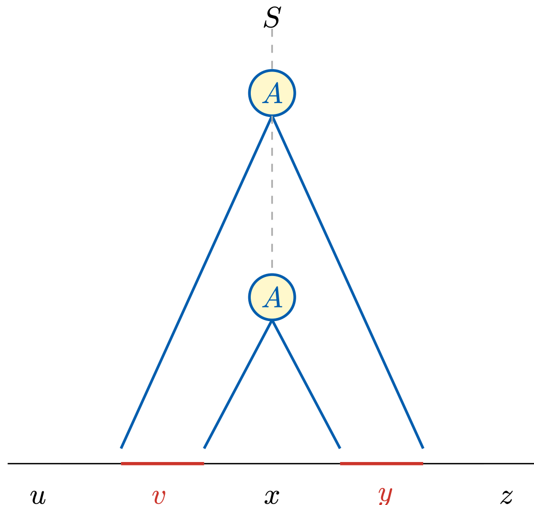
Use CNF to make every parse tree binary.
The pumping length is $n = 2^{|V|+1}$, where $|V|$ is the number of variables.

If $|w| \geq 2^{|V|+1}$, the parse tree has height $\geq |V| + 1$.
By the *Pigeonhole Principle*, some root-to-leaf path visits the same variable A *twice*.

1. Let A_{outer} be the higher occurrence, A_{inner} the lower.
2. The subtree rooted at A_{outer} derives vxy;
   the smaller subtree at A_{inner} derives x.
3. v and y are the “side contributions” between the two As.

Pumping:

- $i = 0$: replace A_{outer}'s subtree with A_{inner}'s subtree.
  Result: $uxz \in L$.
- $i \geq 2$: nest another copy of A_{outer}'s subtree inside itself.
  Result: $uv^ixy^iz \in L$.



$a^n b^n c^n$ is Not Context-Free

Theorem 27: $L = \{a^n b^n c^n \mid n \geq 0\}$ is not context-free.

Proof: Assume L is CFL and let n be the pumping length. Choose $w = a^n b^n c^n \in L$ with $|w| = 3n \geq n$.

By the CFL pumping lemma, $w = uvxyz$ with $|vy| \geq 1$ and $|vxy| \leq n$.

Since $|vxy| \leq n < 3n$, the segment vxy can span *at most two* of the three symbol blocks (as, bs, cs).

- **Case 1:** v and y lie entirely within two adjacent blocks (say as and bs).
  Pumping with $i = 2$ increases the count of as and/or bs but *not* cs.
  The result uv^2xy^2z has unequal symbol counts, so it $\notin L$.
- **Case 2:** v or y spans a block boundary, so pumped strings mix symbols out of order.
  For example, if $v = a^j b^k$, then $v^2 = a^j b^k a^j b^k$, which is not of the form $a^* b^* c^*$. Hence $uv^2xy^2z \notin L$.

In both cases we reach a contradiction. Thus L is not context-free. □

Intuition: A stack can maintain *one* counter (*e.g.*, match as with bs). Matching as, bs, and cs *simultaneously* requires *two* independent counters — beyond what any stack can do.

ww is Not Context-Free

Theorem 28: $L = \{ww \mid w \in \{0,1\}^*\}$ is not context-free.

Proof: Assume L is CFL with pumping length n. Choose $w_0 = 0^n 1^n 0^n 1^n \in L$ (set $w = 0^n 1^n$, so $ww = w_0$, and $|w_0| = 4n$).

By the CFL pumping lemma, $w_0 = uvxyz$ with $|vxy| \leq n$. Since $|vxy| \leq n$ and $|w_0| = 4n$, the segment vxy covers at most n consecutive characters of w_0. It cannot straddle both the first half $0^n 1^n$ and the second half $0^n 1^n$ simultaneously.

Hence v and y together modify the symbol counts in *at most one half* of w_0.

Pumping with $i = 2$ increases character counts in one half but not the other. The resulting uv^2xy^2z cannot equal $w'w'$ for any w' (the two halves become unequal). Thus $uv^2xy^2z \notin L$, contradicting the lemma. □

Context: The language $\{ww \mid w \in \{0,1\}^*\}$ is decidable by Turing machine (it just has two heads, or uses the marking trick from the TM trace slides). This shows Decidable $\neg \subseteq$ CFL.

Closure Properties of CFLs

Operation	CFLs closed?	Proof idea
Union $L_1 \cup L_2$	✓	New start $S \to S_1 \mid S_2$
Concatenation $L_1 \cdot L_2$	✓	New start $S \to S_1 S_2$
Kleene star L^*	✓	New start $S \to SS' \mid \varepsilon$
Reversal L^R	✓	Reverse all production RHSs
Homomorphism $h(L)$	✓	Apply h to each terminal in grammar
$\cap$ with regular, $L \cap R$	✓	Product of PDA stack $\times$ DFA state
Intersection $L_1 \cap L_2$	✗	$\{a^n b^n\} \cap \{b^n c^n\} = \{a^n b^n c^n\} \notin$ CFL
Complement $\overline{L}$	✗	From non-closure under $\cap$ via De Morgan
Difference $L_1 \setminus L_2$	✗	$L \setminus R \in$ CFL for regular R; but $L_1 \setminus L_2$ may not be

Closure Properties of CFLs [2]

Key contrast with regular languages: CFLs are NOT closed under intersection or complement. This means there is no direct "product PDA" construction analogous to the DFA product.

Exception: *Deterministic* CFLs (DCFL) **are** closed under complement. DCFL is the language class of most real programming languages (LR(1) grammars).

Decision Properties of CFLs

Decidable **questions about CFLs:**

- **Empty?** Is $\mathcal{L}(G) = \emptyset$?
  Check whether S can derive any terminal string.
  $\mathcal{O}(|G|)$ time. ✓
- **Membership:** Is $w \in \mathcal{L}(G)$?
  Use CYK algorithm (CNF required):
  $\mathcal{O}(|w|^3 \cdot |G|)$ time. ✓
- **Finite?** Is $\mathcal{L}(G)$ finite?
  Decidable (check for useless variables).
  $\mathcal{O}(|G|)$ time. ✓

Undecidable **questions about CFLs:**

- **Universal?** Is $\mathcal{L}(G) = \Sigma^*$?
  Undecidable. ✗
- **Intersection empty?** Is $\mathcal{L}(G_1) \cap \mathcal{L}(G_2) = \emptyset$?
  Undecidable. ✗
- **Equivalence:** Is $\mathcal{L}(G_1) = \mathcal{L}(G_2)$?
  Undecidable. ✗
- **Ambiguous?** Is G ambiguous?
  Undecidable. ✗

The jump from decidable to undecidable happens exactly at intersection and universality — the same operations that break closure. Decidability and closure are deeply linked.

Summary: Context-Free Languages

1. **Equivalence** (CFL analogue of Kleene): CFG $=$ PDA. Grammars and stack automata define the same class.
2. **CNF + CYK:** every CFL has a grammar in Chomsky Normal Form. CYK decides membership in $\mathcal{O}(n^3 \cdot |G|)$ time via dynamic programming on the parse tree structure.
3. **Pumping Lemma:** if L is context-free with pumping length n, every long word $w \in L$ has a split $w = uvxyz$ with $|vy| \geq 1$, $|vxy| \leq n$, and $uv^ixy^iz \in L$ for all $i \geq 0$. The pair (v, y) comes from a repeated variable in the CNF parse tree.
4. **Closure:** union, concatenation, Kleene star, reversal, homomorphism, and intersection with regular languages. **Not** closed under intersection or complement.
5. **Decidable:** emptiness, membership (CYK), finiteness. **Undecidable:** universality ($\mathcal{L}(G) = \Sigma^*$?), equivalence, intersection emptiness, ambiguity.

Summary: Context-Free Languages [2]

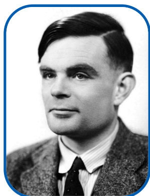
Hard ceiling: a PDA has one *stack* (LIFO memory). It can match one pair of counts, but not two independent ones simultaneously. Canonical witnesses above CFLs: $\{a^n b^n c^n\}$, $\{ww\}$, $\{w\#w\}$.

Where CFLs appear in practice: the syntax of virtually every programming language is a CFL (or close to one: DCFL). Parser generators process CFG specifications; parse trees are the AST. The non-closure under intersection is why type-checking and semantic analysis go *beyond* parsing.

Turing Machines

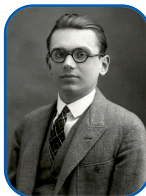
"We may compare a man in the process of computing ...to a machine."

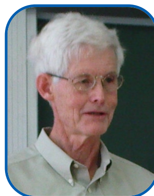
— Alan Turing, 1936

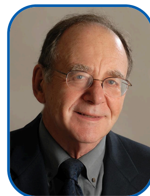

Alan Turing

Alonzo Church

Emil Post


Kurt Gödel


Stephen Cook


Richard Karp

Motivation

Finite automata have *constant memory* (finitely many states).
Pushdown automata have a *stack* (LIFO memory).
What if we give the machine *unrestricted read/write access* to a tape?

This leads us to the *Turing machine* — the most general model of computation we will study.

The fundamental question: What problems can be solved *algorithmically*? Turing machines give us a precise mathematical framework to answer this.

Turing Machine: Definition

Definition 39: A *Turing Machine* (TM) is a 7-tuple $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ where:

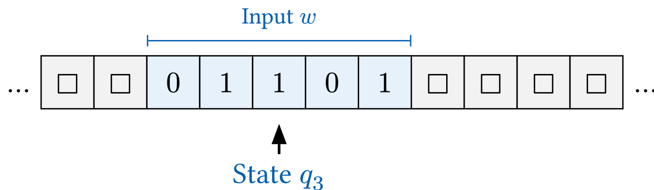
- Q is a finite set of states,
- Σ is the *input alphabet* (not containing the blank symbol $\square$),
- Γ is the *tape alphabet*, where $\square \in \Gamma$ and $\Sigma \subset \Gamma$,
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the *transition function*,
- $q_0 \in Q$ is the *start state*,
- $q_{\text{accept}} \in Q$ is the *accept state*,
- $q_{\text{reject}} \in Q$ is the *reject state* ($q_{\text{reject}} \neq q_{\text{accept}}$).

Note: A Turing machine differs from finite automata in several crucial ways:

- The tape is *infinite* (unbounded in both directions).
- The head can move *both left and right*.
- The machine can *write* to the tape.
- The machine *halts* when it enters q_{accept} or q_{reject}.

Turing Machine: Visualization

A Turing machine operates on an infinite tape divided into cells, each containing a symbol from Γ.



TM: One Step Rule

The machine starts with the input w written on the tape, the head on the leftmost symbol, in state q_0. At each step:

1. Read the symbol under the head.
2. Based on the current state and symbol, the transition function determines:
   - The *new state* to enter.
   - The *symbol to write* in the current cell.
   - The *direction to move* the head (L or R).

TM: Computation

Definition 40: A *configuration* of a TM is a triple: the current state, the tape contents, and the head position. We write configurations as uqv, where q is the current state, u is the tape content to the left of the head, and v is the tape content starting at the head position.

Definition 41: A TM $\mathcal{M}$ on input w:

- *Accepts* w if the computation reaches q_{accept}.
- *Rejects* w if the computation reaches q_{reject}.
- *Loops* if it never halts (neither accepts nor rejects).

Definition 42: The language *recognized* by $\mathcal{M}$ is:

$$\mathcal{L}(\mathcal{M}) = \{w \in \Sigma^* \mid \mathcal{M} \text{ accepts } w\}$$

TM: Computation [2]

Warning: Unlike DFAs, a Turing machine may *never halt* on some inputs. “Not accepted” and “rejected” are *different* — the machine might just run forever.

TM: Example

Example: A TM that recognizes $L = \{0^n 1^n \mid n \geq 0\}$:

Idea: Repeatedly scan the tape, crossing off one 0 and one 1 in each pass.

1. Scan right from the leftmost uncrossed 0. If no 0 found, check that no 1 remains — if so, *accept*.
2. Cross off this 0 (replace with $\times$).
3. Continue scanning right to find the leftmost uncrossed 1. If no 1 found, *reject*.
4. Cross off this 1 (replace with $\times$).
5. Move the head back to the left end of the tape.
6. Repeat from step 1.

Checkpoint: each full pass removes one matching pair $(0, 1)$. If counts are equal and ordered as $0^n 1^n$, the machine eventually reaches all $\times$ and accepts.

TM Trace on 0011, Part 1A

Let's trace configurations on tape (omitting trailing $\square$ cells for readability).

Step 1: Initial				
q_0				
0	0	1	1	$\square$
$\uparrow$				

Step 2				
	q_1			
$\times$	0	1	1	$\square$
	$\uparrow$			

Step 3				
		q_1		
$\times$	0	1	1	$\square$
		$\uparrow$		

Note: After Step 3, the first uncrossed 1 is located.

TM Trace on 0011, Part 1B

Step 4				
			q_2	
$\times$	0	$\times$	1	$\square$
			$\uparrow$	

Step 5				
	q_3			
$\times$	0	$\times$	1	$\square$
	$\uparrow$			

Step 6				
q_3				
$\times$	0	$\times$	1	$\square$
$\uparrow$				

Note: After Step 6, one pair has been removed and the head is back on the left, ready for the second pass.

TM Trace on 0011, Part 2A

Step 7				
	q_0			
$\times$	0	$\times$	1	$\square$
	$\uparrow$			

Step 8				
		q_1		
$\times$	$\times$	$\times$	1	$\square$
		$\uparrow$		

Step 9				
			q_1	
$\times$	$\times$	$\times$	1	$\square$
			$\uparrow$	

Note: The second pass mirrors the first one: find the remaining 0, then the remaining 1.

TM Trace on 0011, Part 2B

Step 10				
				q_2
$\times$	$\times$	$\times$	$\times$	$\square$
				$\uparrow$

Step 11: Accept				
			q_3	
$\times$	$\times$	$\times$	$\times$	$\square$
			$\uparrow$	

Formal trace (abbreviated):

$$q_0\,0011 \Rightarrow \times\, q_1\,011 \Rightarrow \cdots \Rightarrow \times\, q_0\,0 \times 1 \Rightarrow \cdots \Rightarrow \text{accept}$$

Visual insight: The tape acts as writable memory. Crossing symbols out lets the machine remember progress without storing large counters in finite control.

TM vs DFA vs PDA: Comparison

Feature	DFA	PDA	TM
Memory	Finite (states only)	Stack (LIFO)	Infinite tape
Head movement	Left to right only	Left to right only	Both directions
Write?	No	Push/pop stack	Yes (arbitrary)
Halting	Always halts	Always halts	May loop forever
Languages	Regular	Context-free	Recursively enumerable

Historical context: Alan Turing introduced his machine model in 1936, before electronic computers existed. He was trying to formalize the notion of "mechanical procedure" to resolve Hilbert's *Entscheidungsproblem* (decision problem). The Turing machine remains the gold standard model of computation.

Variants of Turing Machines

Several variants of TMs exist, all *equivalent in power*:

- *Multi-tape TMs:* multiple tapes with independent heads.
- *Non-deterministic TMs:* δ returns a *set* of possible transitions.
- *Two-way infinite tape:* tape extends infinitely in both directions.
- *Multi-track tape:* each cell holds a tuple of symbols.

Theorem 29: Every multi-tape TM can be simulated by a single-tape TM.

Theorem 30: Every non-deterministic TM can be simulated by a deterministic TM.

Note: These simulations may involve a *polynomial* (multi-tape) or *exponential* (non-deterministic) slowdown, but they always terminate.

The equivalence of deterministic and non-deterministic TMs is *not* about efficiency.

Whether they can solve the same problems *efficiently* is the famous **P vs NP** problem — one of the most important open questions in mathematics and computer science.

Church–Turing Thesis

Thesis 31: Every "effectively computable" function is computable by a Turing machine.

This is a *thesis*, not a theorem — it cannot be proved mathematically because "effectively computable" is an informal notion. However, every formal model of computation ever proposed has turned out to be *equivalent* to Turing machines:

- Lambda calculus (Church, 1936)
- Recursive functions (Gödel, Kleene)
- Post systems (Post, 1936)
- Register machines
- Modern programming languages (C, Python, Haskell, ...)

The Church–Turing thesis says that Turing machines capture *everything* that can be computed.

If a problem cannot be solved by a TM, it cannot be solved by *any* mechanical procedure — regardless of the programming language or hardware.

Decidability and Recognizability

Definition 43: A language L is *Turing-recognizable* (or *recursively enumerable*, **RE**) if some TM *recognizes* L. That is, the TM accepts every $w \in L$, and either rejects or loops on $w \notin L$.

Definition 44: A language L is *decidable* (or *recursive*, **R**) if some TM *decides* L. That is, the TM halts on *every* input: it accepts $w \in L$ and rejects $w \notin L$.

Definition 45: A language L is *co-recognizable* (**co-RE**) if its complement $\overline{L}$ is Turing-recognizable.

Theorem 32: A language L is decidable iff $L \in \mathrm{RE}$ and $L \in \text{co-RE}$, *i.e.*, $\mathrm{R} = \mathrm{RE} \cap \text{co-RE}$.

Recognize = "*accept yes*-instances, maybe loop on no". **Decide** = "*always halt* with yes/no".

Programs as Data: Encodings

To ask “does M halt on w?” we need to *feed M itself as an input* to another TM. That requires encoding machines as strings.

Definition 46: The *encoding* $\langle M \rangle \in \{0,1\}^*$ of a TM M is a canonical binary string representation of its description: the state list, alphabet, and transition table, encoded in some fixed format.

We write $\langle M, w \rangle$ for the encoding of the pair (M, w).

Key insight: TMs can read their *own descriptions* as input. This is *programs as data* — and it is why the Halting Problem is well-formed. The question “does M halt on $\langle M \rangle$?” is perfectly meaningful.

Cardinality argument (why undecidable languages exist):

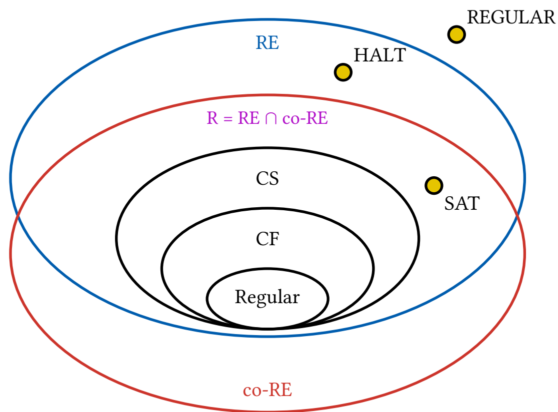
- There are *countably many* TMs (each $\langle M \rangle$ is a finite string).
- There are *uncountably many* languages ($\mathcal{P}(\Sigma^*)$ is uncountable by Cantor’s theorem).
- Therefore *most languages cannot be recognized by any TM* — they are not even RE.

Programs as Data: Encodings [2]

This idea underlies all of modern computing:

- *Compilers:* programs that take source code (a string) as input.
- *Interpreters:* programs that simulate another program on data.
- *Formal verification:* tools that analyse program behaviour — and therefore hit exactly the limits exposed by Rice's Theorem.

Map of Decidability and Recognizability



Classes overview:

$$\text{Regular} \subset \text{CF} \subset \text{CS} \subset \text{R} \subset \text{RE}$$

- **R** = **RE** ∩ **co-RE** — decidable languages
- **HALT** ∈ **RE** \ **R** — recognizable but *not* decidable; TM accepts but may loop
- **SAT** ∈ **R** — decidable (exhaustive search); NP-complete
- **REGULAR** ∈ {"neither RE nor co-RE"} — no TM can even confirm or deny it

Warning: Decidable $\subsetneq$ Recognizable.

- There exist languages that are recognizable but *not* decidable (*e.g.*, HALT).
- There also exist languages that are *not even recognizable* (*e.g.*, $\overline{\text{HALT}}$, $\text{REGULAR}_{\text{TM}}$).

The Halting Problem

Definition 47 (Halting Problem): Given a TM $\mathcal{M}$ and an input w, does $\mathcal{M}$ halt on w?

$$\text{HALT} = \{\langle \mathcal{M}, w \rangle \mid \mathcal{M} \text{ is a TM that halts on input } w\}$$

Theorem 33: HALT is *undecidable*.

Proof: By *diagonalization*. Assume for contradiction that some TM H decides HALT.

Construct a TM D that, on input $\langle \mathcal{M} \rangle$:

1. Run H on $\langle \mathcal{M}, \langle \mathcal{M} \rangle \rangle$.
2. If H accepts (*i.e.*, $\mathcal{M}$ halts on $\langle \mathcal{M} \rangle$), then *loop forever*.
3. If H rejects, then *accept*.

Now consider D on input $\langle D \rangle$:

- If D halts on $\langle D \rangle$, then H accepts, so D loops. Contradiction.
- If D does not halt on $\langle D \rangle$, then H rejects, so D accepts (halts). Contradiction.

The Halting Problem [2]

In either case we get a contradiction, so H cannot exist. □

Key insight: There exist well-defined mathematical problems that *no algorithm can solve*.

This is not a limitation of current technology — it is a *fundamental* impossibility.

The proof uses the same diagonal argument as Cantor's proof that $\mathbb{R}$ is uncountable.

Rice's Theorem

The Halting Problem is just one undecidable problem. Rice's theorem shows that *any* non-trivial property of what a TM computes is undecidable.

Definition 48 (Semantic property): A property P of Turing machines is *semantic* if it depends only on the *language recognized* by the TM, not on its internal structure. Formally: if $\mathcal{L}(M_1) = \mathcal{L}(M_2)$, then $P(M_1) \iff P(M_2)$.

A semantic property is *non-trivial* if some TMs satisfy it and some do not.

Example:

- "*$\mathcal{L}(M)$ is finite*" — semantic (about the language), non-trivial.
- "*$\mathcal{L}(M)$ is regular*" — semantic, non-trivial ($\mathrm{REGULAR}_{\mathrm{TM}}$).
- "$\mathcal{L}(M) = \varnothing$" — semantic, non-trivial ($\mathrm{EMPTY}_{\mathrm{TM}}$).
- "*M has at most 5 states*" — **not** semantic (depends on machine, not its language).

Rice's Theorem [2]

Theorem 34 (Rice's Theorem): Every non-trivial semantic property of TMs is undecidable. That is, if P is non-trivial and semantic, then $\{\langle M \rangle \mid P(M)\}$ is undecidable.

Proof: Assume WLOG that $P(M_\emptyset) = \text{false}$ (where $\mathcal{L}(M_\emptyset) = \emptyset$). Since P is non-trivial, some M_P satisfies $P(M_P) = \text{true}$.

Reduce HALT to P: given $\langle M, w \rangle$, build M' that on input x:

1. Simulate M on w. If M halts and accepts, simulate M_P on x.

Then M halts on $w \to \mathcal{L}(M') = \mathcal{L}(M_P) \to P(M') = \text{true}$;
and M does not halt on $w \to \mathcal{L}(M') = \emptyset \to P(M') = \text{false}$.

A decider for P would decide HALT — contradiction. □

Consequences of Rice's Theorem

Property of $\mathcal{L}(M)$	**Decidable?**	**Reason**
$\mathcal{L}(M) = \emptyset$ (empty language)	✗	Semantic, non-trivial: Rice
$\mathcal{L}(M)$ is finite	✗	Semantic, non-trivial: Rice
$\mathcal{L}(M)$ is infinite	✗	Semantic, non-trivial: Rice
$\mathcal{L}(M)$ contains string w_0	✗	Semantic, non-trivial: Rice
$\mathcal{L}(M)$ is regular	✗	Semantic, non-trivial: Rice
$\mathcal{L}(M) = \Sigma^*$ (accepts all)	✗	Semantic, non-trivial: Rice
$\mathcal{L}(M_1) = \mathcal{L}(M_2)$ (equivalence)	✗	Semantic, non-trivial: Rice
M has fewer than 100 states	✓	**Syntactic** (structural) — NOT semantic!
M halts on the empty string ε within 100 steps	✓	**Syntactic:** simulate 100 steps directly
Is $w \in \mathcal{L}(M)$? for a *fixed known* machine M	✓	Fixed machine $\Rightarrow$ just simulate it

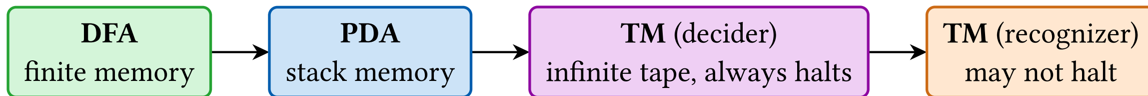
Consequences of Rice's Theorem [2]

The critical distinction: a property is *decidable* only when it depends on the *machine description* (syntactic) rather than on the *language it computes* (semantic).

"How many states does M have?" — we can read off the encoding. Decidable. "Does M accept every input?" — this is about the language. Undecidable.

Practical consequence: No static analysis tool, linter, or type checker can ever be *both complete and sound* for general semantic properties of programs. Every real tool is either *approximate* (may miss bugs) or *conservative* (may report false positives).

The Landscape of Computability



Language Class	Machine	Closure
Regular	DFA/NFA	All Boolean operations
Context-Free	PDA	Union, concat, star; not intersection or complement
Decidable	TM (decider)	All Boolean operations
Recognizable	TM (recognizer)	Union, intersection; not complement

Turing Machine: Exercises

1. **Design a TM** for $L = \{a^n b^n c^n \mid n \geq 0\}$. Describe the states and tape actions in plain English (no need for a full transition table). What does the tape look like mid-computation?
2. **Tracing:** Run the TM for $0^n 1^n$ (from the trace slides) on input `000111`. Write out the full configuration sequence. What happens on input `0011`? On `001`?
3. **Variants:** A two-tape TM can copy the first half of its input to the second tape. Use this to sketch a TM for $L = \{ww \mid w \in \{0,1\}^*\}$. Why is this easy with two tapes but hard with one?
4. **Decidability:** Classify each property as decidable (D), recognizable but not decidable (R), or not even recognizable (N). Justify each answer:
   - $\{\langle M \rangle \mid M \text{ accepts the empty string}\}$
   - $\{\langle M \rangle \mid M \text{ halts on all inputs}\}$
   - $\{\langle M \rangle \mid M \text{ has exactly 7 states}\}$
   - $\{\langle M_1, M_2 \rangle \mid \mathcal{L}(M_1) = \mathcal{L}(M_2)\}$
5. **Rice's Theorem:** For each property below, state whether Rice's Theorem applies. If it does, conclude undecidability. If it does not, say why (and determine decidability separately):

Turing Machine: Exercises [2]

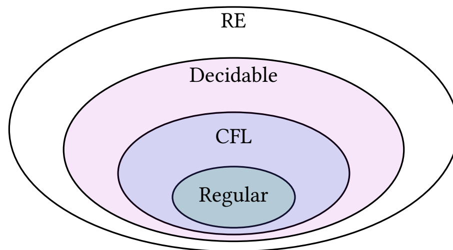
- $\mathcal{L}(M)$ is a regular language.
- M visits state q_3 on input ε.
- M accepts at least one palindrome.
- M has a transition on symbol \$.

Challenge: prove that $\mathrm{HALT} \leq_m \{\langle M \rangle \mid \mathcal{L}(M) \neq \emptyset\}$ by an explicit reduction. Then conclude $\mathrm{EMPTY}_{\mathrm{TM}}$ is undecidable.

Summary: Formal Languages and Automata

1. **Formal languages:** every decision problem is a language $L \subseteq \Sigma^*$. Deciding it = recognizing it.
2. **Regular languages** (DFA $=$ NFA $=$ ε-NFA $=$ RegExp) — Kleene's Theorem.
   - Memory model: finitely many states (constant memory).
   - Pumping Lemma: *necessary* condition for regularity.
   - Myhill-Nerode: *necessary and sufficient* — L regular iff $\equiv_L$ has finite index.
   - Closed under all Boolean operations; all key questions decidable.
3. **Context-free languages** (CFG $=$ PDA):
   - Memory model: a stack (unbounded LIFO memory).
   - CNF and CYK: uniform parse-tree structure, $\mathcal{O}(n^3)$ recognition.
   - CFL Pumping Lemma: two-sided pump v, y from repeated variable in parse tree.
   - Closed under union, concatenation, star, reversal, homomorphism, $\cap$ regular.
   - **Not** closed under intersection or complement.
   - Undecidable: universality, equivalence, intersection emptiness, ambiguity.

Summary: Formal Languages and Automata [2]



Summary: Turing Machines and Limits of Computation

1. **Turing machines** — the universal model of computation (Church–Turing Thesis).
   - Infinite read/write tape; can simulate any other reasonable model.
   - Variants (multi-tape, non-deterministic, two-way) are all equivalent in *power* (not in *speed*).
2. **Decidability landscape:**
   - **R (Decidable):** TM always halts with yes/no. Closed under all Boolean ops.
   - **RE (Recognizable):** TM accepts yes-instances; may loop on no. Closed under $\cup$, $\cap$ only.
   - **co-RE:** complement is RE.
   - $\mathrm{R} = \mathrm{RE} \cap \text{co-RE}$.
   - There exist languages in neither RE nor co-RE (*e.g.*, $\mathrm{REGULAR}_{\mathrm{TM}}$).
3. **The Halting Problem (HALT) is undecidable** — proved by diagonalization. $\mathrm{HALT} \in \mathrm{RE}$ (simulate and accept if halts) but $\mathrm{HALT} \notin \mathrm{R}$.
4. **Rice's Theorem:** every non-trivial *semantic* property of TM behavior is undecidable. Distinguishing semantic (language property) from syntactic (machine structure) is the key.
5. **Counts argument:** countably many TMs, uncountably many languages — most are not even RE.

Summary: Turing Machines and Limits of Computation [2]

The *language hierarchy* (each level strictly contains the one below):

$$\text{Regular} \subsetneq \text{CFL} \subsetneq \text{Decidable} \subsetneq \text{RE} \subsetneq \text{All Languages}$$

Each step up requires a strictly more powerful machine model: DFA $\to$ PDA $\to$ TM-decider $\to$ TM-recognizer.