From Finite Automata to Turing Machines

Samos Summer School in Combinatorics and Computer Science

24–28 August 2026

Four simple-looking questions

Does a binary word end in 01?

Example: 1101

Are the parentheses balanced?

Example: `(()())`

Does a word contain an even number of 1s?

Example: 1011

Can we determine whether a computation will eventually finish?

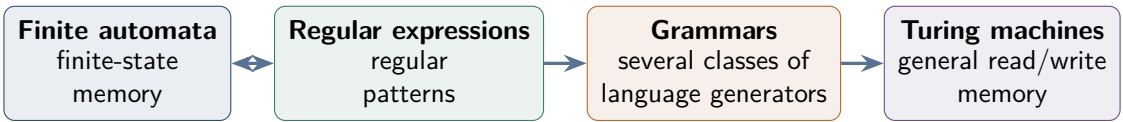
For every possible program?

The surprising part

All four ask a yes/no question about an input, but answering them may require very different kinds of memory and computation.

The journey: from patterns to computation

We will connect **machines**, **language descriptions**, and the memory they require.



The question connecting the course

How does the kind of information a model can retain determine which languages or problems it can handle?

"Grammar" is not one single level of power: regular, context-free, context-sensitive, and unrestricted grammars impose different restrictions on rules.

Why these ideas matter

These models sit underneath many areas of mathematics and computer science:

- text search and pattern matching;
- compilers and programming-language syntax;
- digital circuits and communication protocols;
- verification and formal reasoning;
- the mathematical limits of algorithms.

A recurring question

What changes when a machine is allowed to remember more?

Our destination

From recognising simple patterns to understanding the limits of computation.

Four-hour structure

Chapter	Topic
1	Languages, words, and recognition problems
2	Deterministic and nondeterministic finite automata
3	Regular expressions and Kleene's theorem
–	Break
4	Grammars, context-free languages, and pushdown automata
5	Turing machines
6	Synthesis and exercises

After the theory session

A separate two-hour practical session will use Python and `automata-lib` to construct and simulate these machines.

Learning outcomes

By the end of the theory session, you should be able to:

1. Define alphabet, word, language, automaton, grammar, and Turing machine.
2. Construct and trace simple DFAs and NFAs.
3. Explain the relationship between regular expressions and finite automata.
4. Distinguish regular and context-free grammars and the languages they generate.
5. Construct and trace simple pushdown-automaton computations.
6. Explain why stacks recognise some nonregular languages and where they still fail.
7. Describe how a Turing machine reads, writes, moves, and halts.

Roadmap

1. Languages and recognition problems
2. Finite automata
3. Regular expressions
4. Grammars and pushdown automata
5. Turing machines
6. Synthesis

Alphabet, word, language

Concept	Meaning	Example
Alphabet	finite set of symbols	$\Sigma = \{0, 1\}$
Word	finite sequence over Σ	$w = 010011$
Language	set of words	$L \subseteq \Sigma^*$

Example language:

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

The recognition question is whether a given word belongs to the language.

Recognition problems: a yes/no question

A **recognition problem** asks whether an input belongs to a particular language.

Given a word w, is $w \in L$?

Recognition problems: a yes/no question

A **recognition problem** asks whether an input belongs to a particular language.

Given a word w, is $w \in L$?

For example, let

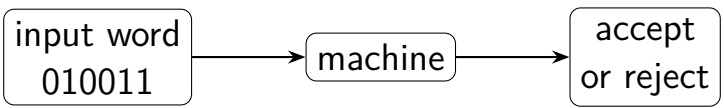
$$L = \{w \in \{0,1\}^* : w \text{ ends in } 01\}.$$

Input	Answer	Why?
1101	accept	ends in 01
1110	reject	does not end in 01

The machine's job is to produce the correct answer for *every* possible input.

A useful mental model

A machine reads a word from left to right and keeps some information about what it has seen.




Different models differ mainly in how much memory they have and how they use it.

Classification warm-up

For each language, guess the kind of memory needed.

Language	What must be remembered?
Strings ending in 01	last few symbols
Strings with an even number of 1s	parity
Balanced parentheses	nesting depth
Strings 0^n1^n	number of 0s
Programs that halt	behavior of an arbitrary program


The course progressively turns this intuition into formal models.

Roadmap

1. Languages and recognition problems
2. Finite automata
3. Regular expressions
4. Grammars and pushdown automata
5. Turing machines
6. Synthesis

A tiny machine that reads one symbol at a time

Imagine a very simple machine.

It reads a word **from left to right**, one symbol at a time:

1 0 1 1

After each symbol, the machine can change its current **state**.

What is a state?

A state is the machine's current situation: the small amount of information it remembers about what it has seen so far.

For our example, the machine only needs two possible situations:

even number of 1s so far odd number of 1s so far.

These become the two states of our first **finite automaton**.

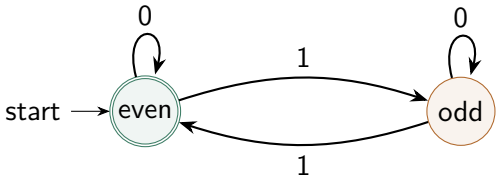
Our first finite automaton

The circles are the two **states**:

even and odd.

The arrows tell us what happens when the next symbol is read.

- Reading 1 switches state.
- Reading 0 keeps the same state.
- The incoming arrow marks the initial state.
- The double circle means “accept”.



For 1011:

$$\text{even} \xrightarrow{1} \text{odd} \xrightarrow{0} \text{odd} \xrightarrow{1} \text{even} \xrightarrow{1} \text{odd}$$

We finish in the non-accepting state, so 1011 is rejected.

The idea to remember

The machine remembers only what still matters about the past, not the whole word.

What does a state represent?

A state is not just a point in a diagram.

A **state** represents the information about the input read so far that is still relevant for processing the rest of the word.

States as compressed memory

The automaton does not remember the whole past; it remembers only what can affect its future behaviour.

What does a state represent?

A state is not just a point in a diagram.

A **state** represents the information about the input read so far that is still relevant for processing the rest of the word.

States as compressed memory

The automaton does not remember the whole past; it remembers only what can affect its future behaviour.

Different input histories may lead to the same state if the machine no longer needs to distinguish them.

Deterministic finite automaton: formal definition

A **deterministic finite automaton** (DFA) is a tuple

$$A = (Q, \Sigma, \delta, q_0, F),$$

where:

- Q is a finite set of states;
- Σ is a finite input alphabet;
- $\delta : Q \times \Sigma \to Q$ is the transition function;
- $q_0 \in Q$ is the initial state;
- $F \subseteq Q$ is the set of accepting states.

Why "deterministic"?

For every current state and input symbol, there is exactly one next state.

Our example

For the even-number-of-1s automaton:

$$Q = \{q_e, q_o\}, \qquad \Sigma = \{0, 1\}, \qquad q_0 = q_e, \qquad F = \{q_e\}.$$

The transition function δ can be written as a table:

State	input 0	input 1
q_e	q_e	q_o
q_o	q_o	q_e

Read one entry

$\delta(q_e, 1) = q_o$ means: if the number of 1s seen so far is even and we read another 1, it becomes odd.

Why only two states are enough

Consider the different prefixes

$$0011, \qquad 1100, \qquad 1111.$$

They all contain an **even** number of 1s, so after reading any of them the automaton can be in the same state:

$$0011,\ 1100,\ 1111 \quad \longrightarrow \quad q_e.$$

Likewise,

$$1,\ 001,\ 111 \quad \longrightarrow \quad q_o.$$

Interpretation

A state groups together input histories that require the same future behaviour.

Try it yourself

Run the automaton on

$$101101.$$

Start in q_e, read one symbol at a time, and follow the corresponding transition.

Questions

1. Which state do we end in?
2. Is the word accepted?
3. How many 1s does the word contain?

DFA computation as an extended transition

The transition function reads one symbol.

$$\delta(q, a) = q'$$

To describe reading a whole word, we define an extended transition function:

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

$$\delta^*(q, xa) = \delta(\delta^*(q, x), a)$$

A word w is accepted if:

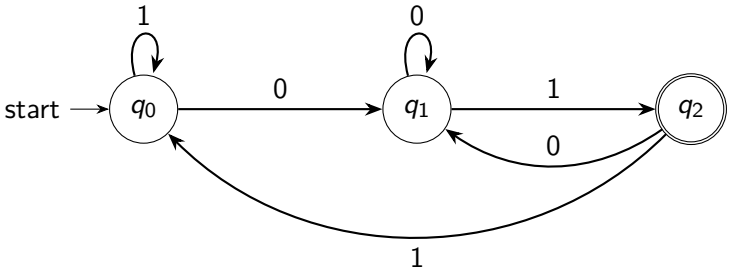
$$\delta^*(q_0, w) \in F.$$

Example: strings ending in 01

We need to remember enough suffix information.

Example: strings ending in 01

We need to remember enough suffix information.



State meanings:

- q_0: no useful suffix yet, or last symbol is 1 not preceded by a useful final 0;
- q_1: the last symbol is 0;
- q_2: the word currently ends in 01.

Exercise: build a DFA

Construct a DFA over $\{0, 1\}$ for each one of the following languages.

1. Strings containing the substring 11.
2. Strings whose length is divisible by 3.
3. Strings that do not contain 00.
4. Strings where the third symbol from the end is 1.

A language recognised by a DFA

For a DFA A, define:

$$L(A) = \{w \in \Sigma^* : A \text{ accepts } w\}.$$

Regular language

A language is regular if it is recognised by some finite automaton.

Finite automata are good for fixed-pattern recognition with finite memory.

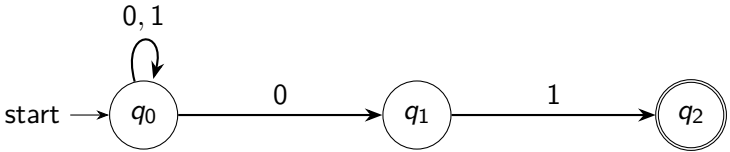
Nondeterministic finite automata

An NFA is like a DFA, except that from a state and input symbol there may be:

- zero possible next states;
- one possible next state;
- several possible next states;
- sometimes ε-transitions that consume no input.

A word is accepted if at least one possible path ends in an accepting state.

Example NFA: strings ending in 01



The NFA may guess that a particular 0 is the beginning of the final suffix 01.

Determinism vs nondeterminism

Feature	DFA	NFA
Next state	exactly one	zero, one, or many
Execution	one path	many possible paths
Acceptance	the path accepts	at least one path accepts
Expressive power	regular languages	regular languages
Implementation	direct	often converted to DFA

Important theorem:

DFA and NFA recognise exactly the same class of languages.

If NFAs are equally powerful, why do we need DFAs?

NFAs and DFAs recognise the same languages, but they are useful for **different reasons**.

NFAs: convenient to design

An NFA can express several possible choices at once.

- often smaller;
- easy to construct;
- natural from a regular expression.

DFAs: convenient to execute

Every state and input symbol has exactly one next state.

- one computation path;
- no alternatives to track;
- straightforward execution.

If NFAs are equally powerful, why do we need DFAs?

NFAs and DFAs recognise the same languages, but they are useful for **different reasons**.

NFAs: convenient to design

An NFA can express several possible choices at once.

- often smaller;
- easy to construct;
- natural from a regular expression.

DFAs: convenient to execute

Every state and input symbol has exactly one next state.

- one computation path;
- no alternatives to track;
- straightforward execution.

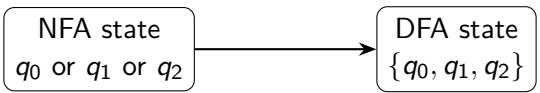
The practical idea

Use an NFA when it is convenient to **describe** the language, then convert it to a DFA when it is convenient to **run** the machine.

NFA: easy to construct $\longrightarrow$ DFA: easy to execute

Subset construction intuition

To simulate an NFA deterministically, track the set of all states where the NFA could currently be.



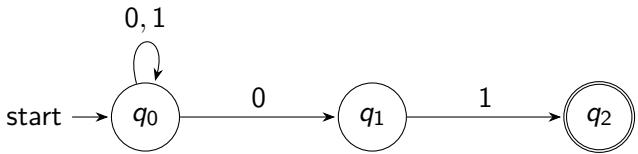
A DFA state can encode a set of possible NFA states.

Example: converting an NFA into a DFA

Consider the language

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

The following NFA recognises L:



- q_0 reads any prefix of the word.
- On a 0, the NFA may guess that the final 01 has started.
- It accepts if the next symbol is 1 and the input then ends.

Step 1: the initial DFA state

A state of the DFA represents a **set of possible NFA states**.

The NFA starts only in q_0, so the initial DFA state is

$$A = \{q_0\}.$$

Now calculate its transitions.

Step 1: the initial DFA state

A state of the DFA represents a **set of possible NFA states**.

The NFA starts only in q_0, so the initial DFA state is

$$A = \{q_0\}.$$

Now calculate its transitions.

$$\delta_{\mathrm{DFA}}(A, 0) = \delta_{\mathrm{NFA}}(\{q_0\}, 0) = \{q_0, q_1\}.$$

We therefore create a new DFA state:

$$B = \{q_0, q_1\}.$$

Step 1: the initial DFA state

A state of the DFA represents a **set of possible NFA states**.

The NFA starts only in q_0, so the initial DFA state is

$$A = \{q_0\}.$$

Now calculate its transitions.

$$\delta_{\mathrm{DFA}}(A, 0) = \delta_{\mathrm{NFA}}(\{q_0\}, 0) = \{q_0, q_1\}.$$

We therefore create a new DFA state:

$$B = \{q_0, q_1\}.$$

For input 1,

$$\delta_{\mathrm{DFA}}(A, 1) = \delta_{\mathrm{NFA}}(\{q_0\}, 1) = \{q_0\} = A.$$

Step 2: process the new subset

We must now calculate the transitions from

$$B = \{q_0, q_1\}.$$

For each symbol, take the union of all possible NFA transitions.

Step 2: process the new subset

We must now calculate the transitions from

$$B = \{q_0, q_1\}.$$

For each symbol, take the union of all possible NFA transitions.
On input 0:

$$\begin{aligned}\delta_{\mathrm{DFA}}(B, 0) &= \delta(q_0, 0) \cup \delta(q_1, 0) \\ &= \{q_0, q_1\} \cup \varnothing \\ &= \{q_0, q_1\} = B.\end{aligned}$$

Step 2: process the new subset

We must now calculate the transitions from

$$B = \{q_0, q_1\}.$$

For each symbol, take the union of all possible NFA transitions.
On input 0:

$$\begin{aligned}\delta_{\mathrm{DFA}}(B, 0) &= \delta(q_0, 0) \cup \delta(q_1, 0) \\ &= \{q_0, q_1\} \cup \varnothing \\ &= \{q_0, q_1\} = B.\end{aligned}$$

On input 1:

$$\begin{aligned}\delta_{\mathrm{DFA}}(B, 1) &= \delta(q_0, 1) \cup \delta(q_1, 1) \\ &= \{q_0\} \cup \{q_2\} \\ &= \{q_0, q_2\}.\end{aligned}$$

This creates another DFA state: $C = \{q_0, q_2\}$.

Step 3: complete the transition table

Finally, calculate the transitions from $C = \{q_0, q_2\}$.
Since q_2 has no outgoing transitions,

$$\delta(C, 0) = \{q_0, q_1\} = B, \qquad \delta(C, 1) = \{q_0\} = A.$$

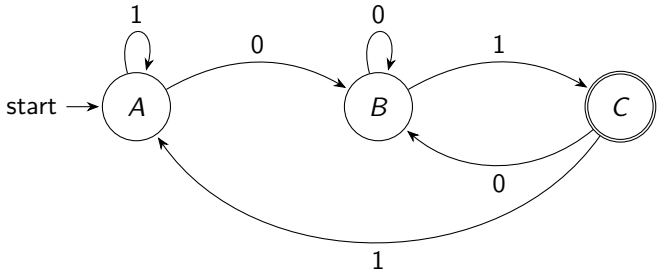
The complete DFA transition table is therefore:

DFA state	**Input** 0	**Input** 1
$A = \{q_0\}$	B	A
$B = \{q_0, q_1\}$	B	C
$C = \{q_0, q_2\}$	B	A

Accepting subsets

A DFA state is accepting whenever it contains an accepting NFA state.
Since q_2 is accepting, $C = \{q_0, q_2\}$ is accepting.

The resulting DFA



where

$$A = \{q_0\}, \qquad B = \{q_0, q_1\}, \qquad C = \{q_0, q_2\}.$$

What changed?

The NFA could be in several states simultaneously.
The DFA stores that entire set of possibilities as one state.

Checking the construction

Consider the input: 1101.
The DFA follows the path: $A \xrightarrow{1} A \xrightarrow{1} A \xrightarrow{0} B \xrightarrow{1} C$.
Since C is accepting, the word is accepted.

This agrees with the language definition: 1101 ends in 01.

Checking the construction

Consider the input: 1101.
The DFA follows the path: $A \xrightarrow{1} A \xrightarrow{1} A \xrightarrow{0} B \xrightarrow{1} C$.
Since C is accepting, the word is accepted.

This agrees with the language definition: 1101 ends in 01.

Subset construction

Repeat the following process until no new subsets appear:

1. Start with the set of initial NFA states.
2. For each input symbol, compute all reachable NFA states.
3. Treat each resulting subset as one DFA state.
4. Mark every subset containing an NFA final state as accepting.

The pigeonhole principle

Pigeonhole principle

If more objects are placed into fewer containers, then at least one container must contain more than one object.

For example, if 4 pigeons enter 3 pigeonholes, then at least one pigeonhole must contain at least 2 pigeons.

$$\underbrace{4}_{\text{objects}} > \underbrace{3}_{\text{containers}} \implies \text{some container receives at least two objects.}$$

The pigeonhole principle

Pigeonhole principle

If more objects are placed into fewer containers, then at least one container must contain more than one object.

For example, if 4 pigeons enter 3 pigeonholes, then at least one pigeonhole must contain at least 2 pigeons.

$$\underbrace{4}_{\text{objects}} > \underbrace{3}_{\text{containers}} \implies \text{some container receives at least two objects.}$$

If $n+1$ objects are placed into n containers, then at least two objects must be placed in the same container.

The pigeonhole principle

Pigeonhole principle

If more objects are placed into fewer containers, then at least one container must contain more than one object.

For example, if 4 pigeons enter 3 pigeonholes, then at least one pigeonhole must contain at least 2 pigeons.

$$\underbrace{4}_{\text{objects}} > \underbrace{3}_{\text{containers}} \implies \text{some container receives at least two objects.}$$

If $n + 1$ objects are placed into n containers, then at least two objects must be placed in the same container.

Why is this useful in automata theory?

A finite automaton has only finitely many states.
If it reads more input prefixes than it has states, then at least two different prefixes must lead to the same state.

What finite automata cannot do

Consider:

$$L = \{0^n 1^n : n \geq 0\}.$$

Examples in L:

$$\varepsilon,\ 01,\ 0011,\ 000111.$$

Examples not in L:

$$001,\ 011,\ 0101,\ 00011.$$

A finite automaton would need to remember how many 0s were seen before the 1s start.

Finite memory limitation

Suppose a DFA has k states.
After reading:

$$\varepsilon, 0, 00, 000, \ldots, 0^k$$

there are $k + 1$ prefixes but only k states.

By the **pigeonhole principle**, two different prefixes must lead to the same state.

Finite memory limitation

Suppose a DFA has k states.
After reading:

$$\varepsilon, 0, 00, 000, \ldots, 0^k$$

there are $k + 1$ prefixes but only k states.

By the **pigeonhole principle**, two different prefixes must lead to the same state.


The automaton has forgotten an important difference.

Two prefixes collide?

Because the DFA has only k states, there must exist two integers

$$0 \leq i < j \leq k$$

such that the prefixes 0^i and 0^j lead to the same state.

$$\delta^*(q_0, 0^i) = \delta^*(q_0, 0^j)$$

Although the two prefixes contain different numbers of 0s, the automaton has exactly the same internal state after reading them.

What has the automaton forgotten?

It can no longer distinguish between having read i zeros and having read j zeros.

Why does this forgotten difference matter?

Suppose the language is

$$L = \{0^n 1^n : n \geq 0\}.$$

After reading 0^i, the automaton must **accept** the continuation 1^i:

$$0^i 1^i \in L.$$

But after reading 0^j, where $j > i$, the same continuation should be **rejected**:

$$0^j 1^i \notin L.$$

The contradiction

We know that $\delta^*(q_0, 0^i) = \delta^*(q_0, 0^j)$.
A deterministic automaton that is in the same state and reads the same remaining input must behave in the same way.
Therefore, after appending 1^i, it must either accept both words or reject both words:

$$0^i 1^i \qquad \text{and} \qquad 0^j 1^i.$$

The contradiction

We know that $\delta^*(q_0, 0^i) = \delta^*(q_0, 0^j)$.
A deterministic automaton that is in the same state and reads the same remaining input must behave in the same way.
Therefore, after appending 1^i, it must either accept both words or reject both words:

$$0^i 1^i \qquad \text{and} \qquad 0^j 1^i.$$

But the correct answers are different:

$$0^i 1^i \in L, \qquad 0^j 1^i \notin L.$$

By contradiction

No DFA can recognise the language

$$L = \{0^n 1^n : n \geq 0\}.$$

Why finite memory is insufficient

To recognise 0^n1^n, a machine must remember exactly how many 0s appeared before the first 1.

The required number may be arbitrarily large:

$$1,\ 2,\ 3,\ \ldots$$

A DFA has only **finitely many states**, so it can distinguish only **finitely many memory situations**.

Why finite memory is insufficient

To recognise 0^n1^n, a machine must remember exactly how many 0s appeared before the first 1.

The required number may be arbitrarily large:

$$1,\ 2,\ 3,\ \ldots$$

A DFA has only **finitely many states**, so it can distinguish only **finitely many memory situations**.

General idea about DFA

Finite automata can remember properties with finitely many possibilities, such as:

- even or odd;
- the last symbol read;
- a count modulo k.

They cannot store an arbitrary, unbounded count.

Roadmap

1. Languages and recognition problems
2. Finite automata
3. Regular expressions
4. Grammars and pushdown automata
5. Turing machines
6. Synthesis

From machines to descriptions

So far, we have described languages by building machines:

$$\text{input word} \longrightarrow \text{finite automaton} \longrightarrow \text{accept or reject.}$$

For example, we can construct a DFA that recognises all binary words ending in 01.

From machines to descriptions

So far, we have described languages by building machines:

$$\text{input word} \longrightarrow \text{finite automaton} \longrightarrow \text{accept or reject.}$$

For example, we can construct a DFA that recognises all binary words ending in 01.

BUT drawing a complete automaton is not always the most convenient way to describe a language.

From machines to descriptions

A different question

Can we describe the same set of accepted words using a compact symbolic expression?

From machines to descriptions

A different question

Can we describe the same set of accepted words using a compact symbolic expression?

For words ending in 01, we want to express: Any sequence of 0s and 1s, followed by 01.
This can be written as: $(0 \mid 1)^*01$.

Such symbolic descriptions are called **regular expressions**.

Two views of the same language

Finite automata and regular expressions describe languages in two different ways.

Finite automaton

An **operational** description.

It explains how a machine reads a word step by step and decides whether to accept it.

Regular expression

A **declarative** description.

It describes directly the structure that accepted words must have.

$$\underbrace{\text{DFA or NFA}}_{\text{recognises the language}} \iff \underbrace{\text{regular expression}}_{\text{describes the language}}$$

Fundamental result

Finite automata and regular expressions have exactly the same expressive power: they define precisely the regular languages.

Regular expressions

Regular expressions describe regular languages using algebraic operations.

Basic constructions:

Expression	Meaning	Example
a	symbol	0
RS	concatenation	01
$R \mid S$	union / choice	$0 \mid 1$
R^*	zero or more repetitions	0^*
(R)	grouping	$(01)^*$

Example regular expressions

Language	Regular expression
Binary strings ending in 01	$(0 \mid 1)^*01$
Binary strings with exactly one 1	0^*10^*

Reading the expressions

- $(0 \mid 1)^*01$: any binary prefix, followed by 01.
- 0^*10^*: any number of 0s, exactly one 1, then any number of 0s.

Exercise: write the regular expression

Find a regular expression for each language.

1. Binary strings containing **at least one** 1.
2. Any number of repetitions of 01.
3. Strings over $\{a, b\}$ containing the substring ab.

Hint

Think in terms of the three basic operations:

choice | concatenation | repetition

using

$|$, RS, R^*.

Exercise: solutions

1. Binary strings containing at least one 1:

$$(0 \mid 1)^* 1 (0 \mid 1)^*$$

2. Any number of repetitions of 01:

$$(01)^*$$

3. Strings over $\{a, b\}$ containing the substring ab:

$$(a \mid b)^* ab (a \mid b)^*$$

From regex to automaton

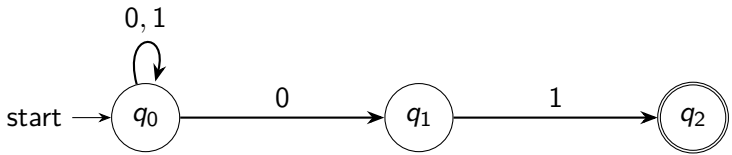
The expression

$$(0 \mid 1)^*01$$

means:

1. read any binary prefix;
2. then read a 0;
3. then read a final 1.

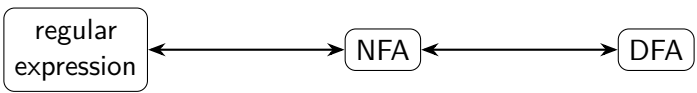
This naturally creates an NFA:



Kleene's theorem

Kleene's theorem

A language is recognised by a finite automaton if and only if it can be described by a regular expression.



Different descriptions, same class of languages.

Exercise: match language and expression

Match each language with a regular expression.

1. Strings ending in 1.
2. Strings containing exactly one 1.
3. Strings with any number of 0s followed by any number of 1s.
4. Strings containing the substring 101.

A. 0^*10^*
B. $(0 \mid 1)^*101(0 \mid 1)^*$
C. $(0 \mid 1)^*1$
D. 0^*1^*

Regular expressions are not magic

Regular expressions do not give more power than finite automata.

They can describe:

- fixed suffixes;
- finite alternatives;
- repeated local patterns;
- parity or modular conditions.

They cannot describe all forms of unbounded dependency, such as:

$$\{0^n 1^n : n \geq 0\}.$$

Roadmap

1. Languages and recognition problems
2. Finite automata
3. Regular expressions
4. Grammars and pushdown automata
5. Turing machines
6. Synthesis

Beyond regular patterns

Regular expressions can describe many useful patterns:

$$(0 \mid 1)^*01, \qquad 0^*1^*, \qquad (01)^*.$$

They are well suited to fixed prefixes or suffixes, repeated local patterns, finitely many alternatives, and counting modulo a fixed number.

Beyond regular patterns

Regular expressions can describe many useful patterns:

$$(0 \mid 1)^*01, \qquad 0^*1^*, \qquad (01)^*.$$

They are well suited to fixed prefixes or suffixes, repeated local patterns, finitely many alternatives, and counting modulo a fixed number.

But some languages have a more complex dependency. For example,

$$L = \{0^n1^n : n \geq 0\}.$$

A word belongs to this language only when the number of 1s matches the number of preceding 0s.

Limitation

No finite automaton, and therefore no regular expression, can describe this language.

Languages with recursive structure

Consider balanced parentheses:

$$\varepsilon, \qquad (), \qquad (()), \qquad ()(), \qquad (()()).$$

Whether a closing parenthesis is valid may depend on an opening parenthesis arbitrarily far away.

The structures can also be nested to arbitrary depth:

$$(), \qquad (()), \qquad ((())), \qquad (((()))), \qquad \ldots$$

What is missing?

We need a way to describe structures that can contain smaller instances of themselves.

This property is called **recursion**.

From recognition to generation

So far, we have described languages using machines and patterns:

$$\underbrace{\text{finite automata}}_{\text{recognise words}} \quad \Longleftrightarrow \quad \underbrace{\text{regular expressions}}_{\text{describe patterns}}.$$

We now introduce another point of view:

How can we generate all valid words of a language?

Instead of reading a finished word, we begin with a symbol and repeatedly apply construction rules. For example,

$$S \to 0S1 \qquad \text{or} \qquad S \to \varepsilon.$$

These rules generate

$$\varepsilon, \quad 01, \quad 0011, \quad 000111, \quad \ldots$$

Terminal and nonterminal symbols

We will use two kinds of symbols :

Terminal symbols

These are the symbols that may appear in the final words of the language. They are not replaced during the derivation.

For example, if $\Sigma = \{0, 1\}$, then 0 and 1 are terminals.

Terminal and nonterminal symbols

We will use two kinds of symbols :

Terminal symbols

These are the symbols that may appear in the final words of the language. They are not replaced during the derivation.

For example, if $\Sigma = \{0, 1\}$, then 0 and 1 are terminals.

Nonterminal symbols

These are variables used while constructing a word. They can be replaced according to production rules.

The symbol S is often a nonterminal and the **start symbol**.

Terminal and nonterminal symbols

We will use two kinds of symbols :

Terminal symbols

These are the symbols that may appear in the final words of the language. They are not replaced during the derivation.

For example, if $\Sigma = \{0, 1\}$, then 0 and 1 are terminals.

Nonterminal symbols

These are variables used while constructing a word. They can be replaced according to production rules.

The symbol S is often a nonterminal and the **start symbol**.

A useful distinction

Terminals belong to the final word; nonterminals belong to the construction process.

What is a grammar?

A **grammar** is a finite collection of symbols and production rules used to generate words of a language.

Formally, we write

$$G = (V, \Sigma, P, S),$$

where:

- V is a finite set of nonterminal symbols;
- Σ is the terminal alphabet;
- P is a finite set of production rules;
- $S \in V$ is the start symbol.

The language generated by a grammar

$L(G)$ is the set of all terminal words that can be derived from S.

Production rules and derivations

Consider the production rules

$$S \to 0S1, \qquad S \to \varepsilon.$$

A production rule tells us which substitutions are allowed. The symbol $\Rightarrow$ means that one such rule has been applied.

For example,

$$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 0011.$$

The last step applies $S \to \varepsilon$:

$$00S11 \Rightarrow 00\varepsilon 11 = 0011.$$

Notation

$\to$ defines a production rule; $\Rightarrow$ denotes a derivation step.

Not all grammars are the same

The word **grammar** describes a general framework. Different restrictions on the production rules define different grammar classes.

Grammar class	Restriction, informally	Language class
Regular	very restricted linear rules	regular
Context-free	one nonterminal on the left	context-free
Context-sensitive	rules may depend on context	context-sensitive
Unrestricted	essentially general rewriting	Turing-recognisable

Main idea

No: not every grammar is context-free. "Context-free" names one important class of grammars.

Regular grammars

In this course we use **right-linear grammars**, with rules of the form

$$A \to aB, \qquad A \to a, \qquad A \to \varepsilon,$$

where A, B are nonterminals and a is a terminal.
It is called right-linear because, if a nonterminal appears on the right-hand side, it appears at the far right:

$$\underbrace{a}_{\text{terminal}} \underbrace{B}_{\text{nonterminal}} .$$

Key result

Right-linear grammars generate exactly the regular languages. They have the same expressive power as DFAs, NFAs, and regular expressions.

Example: a regular grammar

Consider binary strings ending in 01.
A right-linear grammar for this language is

$$S \to 0S \mid 1S \mid 0A, \qquad A \to 1.$$

For example,

$$S \Rightarrow 1S \Rightarrow 11S \Rightarrow 110A \Rightarrow 1101.$$

Connection to what we already know

This is the same regular language that we previously described with a DFA, an NFA, and the regular expression $(0 \mid 1)^*01$.

What is a context-free grammar?

A **context-free grammar** (CFG) is a grammar in which every production rule has the form

$$A \to \alpha,$$

where:

- A is a **single nonterminal**;
- α is any finite sequence of terminals and nonterminals, possibly ε.

Examples include

$$S \to 0S1, \qquad S \to SS, \qquad S \to (S), \qquad S \to \varepsilon.$$

Why "context-free"?

The rule for A can be applied whenever A appears; its replacement does not depend on the symbols surrounding it.

What is a context-free language?

A grammar is a **description**; a language is a **set of words**.

Context-free language

A language L is called **context-free** if there exists a context-free grammar G such that

$$L = L(G).$$

What is a context-free language?

A grammar is a **description**; a language is a **set of words**.

Context-free language

A language L is called **context-free** if there exists a context-free grammar G such that

$$L = L(G).$$

For example,

$$G: \qquad S \to 0S1 \mid \varepsilon$$

is a context-free grammar, and

$$L(G) = \{0^n 1^n : n \geq 0\}.$$

Therefore,

$\{0^n 1^n : n \geq 0\}$ is a context-free language.

Regular grammars are a special case

A right-linear grammar permits only restricted rules such as

$$A \to aB, \qquad A \to a, \qquad A \to \varepsilon.$$

A context-free grammar permits the more general form

$$A \to \alpha.$$

Therefore every right-linear grammar is context-free, but not every context-free grammar is right-linear.

Regular grammars are a special case

A right-linear grammar permits only restricted rules such as

$$A \to aB, \qquad A \to a, \qquad A \to \varepsilon.$$

A context-free grammar permits the more general form

$$A \to \alpha.$$

Therefore every right-linear grammar is context-free, but not every context-free grammar is right-linear.

For example,

$$S \to 0S1$$

is context-free, but it is **not** right-linear because the nonterminal is not at the far right.

Consequence

Every regular language is context-free, but some context-free languages are not regular.

A CFG for 0^n1^n

Consider the context-free grammar

$$S \to 0S1 \mid \varepsilon.$$

Each application of $S \to 0S1$ creates one 0 and one matching 1:

$$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 000S111 \Rightarrow 000111.$$

Hence

$$L(G) = \{0^n1^n : n \geq 0\}.$$

The key mechanism

The recursive nonterminal carries a dependency across an arbitrarily long word.

Balanced parentheses: another context-free language

A CFG for balanced parentheses is

$$S \to SS \mid (S) \mid \varepsilon.$$

The rules have intuitive meanings:

- $S \to SS$: concatenate two balanced expressions;
- $S \to (S)$: put a balanced expression inside parentheses;
- $S \to \varepsilon$: stop with the empty word.

For example,

$$S \Rightarrow (S) \Rightarrow ((S)) \Rightarrow (()).$$

What makes this possible?

The rule $S \to (S)$ can be applied recursively, creating arbitrary nesting.

Why does nesting require memory?

Consider the word `((()))`.
While reading from left to right, we must remember how many opening parentheses have not yet been closed.

symbol read	(	(	(	)	)	)
nesting depth	1	2	3	2	1	0

For a word to be balanced, the depth must never become negative and must be 0 at the end.

The problem for finite automata

The nesting depth can be arbitrarily large. A finite automaton has only finitely many states, so it cannot remember arbitrary nesting depth.

From finite automata to pushdown automata

A finite automaton remembers only its current state.

$$\boxed{\text{finite state}} \quad \Longrightarrow \quad \boxed{\text{finite memory}}$$

This is enough for regular languages, but not for

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

From finite automata to pushdown automata

A finite automaton remembers only its current state.

$$\boxed{\text{finite state}} \quad \Longrightarrow \quad \boxed{\text{finite memory}}$$

This is enough for regular languages, but not for

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

Add an external memory

We extend a finite-state machine with a memory structure that can grow with the input:

$$\boxed{\text{finite-state control}} \quad + \quad \boxed{\text{stack}} \quad = \quad \boxed{\text{pushdown automaton}}.$$

Why a stack?

There are many possible ways to organise memory. Two familiar examples are:

Queue: first in, first out (FIFO)

The item that was stored **first** is removed first.

$$\underbrace{A}_{\text{oldest}}\ B\ \underbrace{C}_{\text{newest}} \longrightarrow B\ C$$

Here A is at the **front** of the queue.

Useful when the order of arrival matters.

Stack: last in, first out (LIFO)

The item that was stored **last** is removed first.

$$\underbrace{A}_{\text{oldest}}\ B\ \underbrace{C}_{\text{top}} \longrightarrow A\ B$$

Here C is the **top** of the stack.

Only the top item can be removed directly.

Why stacks fit nesting

The most recently opened structure is the first one that must be closed. This is exactly a **last-in, first-out** (LIFO) approach.

Stack memory: push, pop, and top

We write a stack as a word over a stack alphabet Γ.


In this course, the **top of the stack is written on the left**.

Suppose the current stack is

$$YZZY.$$

Then:

Operation	Result	Meaning
push$(X, YZZY)$	$XYZZY$	put X on top
pop$(YZZY)$	ZZY	remove the top symbol
top$(YZZY)$	Y	inspect the top symbol

Important restriction

The stack can grow without a fixed bound, but the machine can directly inspect or remove only the **top** symbol.

A PDA is an NFA with a stack

A pushdown automaton behaves like a nondeterministic finite automaton, but every transition may also inspect and modify the stack.

At each step, the machine may use:

1. the current state;
2. the next input symbol — or ε, meaning "read nothing";
3. the top stack symbol — or ε, meaning "do not inspect/pop".

A move may then change state, consume an input symbol, and update the stack by popping and/or pushing symbols.

The key change

For a DFA/NFA, the control state is the memory. For a PDA, the **control state + stack** together determine what the machine remembers.

Pushdown automaton: formal definition

A pushdown automaton can be written as

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

where:

- Q is a finite set of states;
- Σ is the input alphabet;
- Γ is the stack alphabet;
- $q_0 \in Q$ is the initial state;
- $Z_0 \in \Gamma$ is the initial stack symbol;
- $F \subseteq Q$ is the set of accepting states;
- δ specifies the possible moves.

For a nondeterministic PDA, a standard form is

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma^*).$$

A transition answers:

(1.) what may I read? (2.) what is on top? (3.) where may I go? (4.) what replaces the top?

How do we read a PDA arrow?

We label a transition by

$$\boxed{a,\ A/B}$$

and read it from left to right:

For example,

$$a,\ A/B$$

can be taken when the next input symbol is a and the top of the stack is A. The machine consumes a, removes A, and puts B on top.

Three pieces of information

INPUT, **POP**, **PUSH**.

Common PDA transition types

The symbol ε means "nothing" in the corresponding position.

Label	Action	Interpretation
$a, A/B$	read a, pop A, push B	normal stack update
$a, A/\varepsilon$	read a, pop A	pop only
$a, \varepsilon/B$	read a, push B	push only
$a, \varepsilon/\varepsilon$	read a	ignore stack
$\varepsilon, A/B$	pop A, push B	change stack, no input

Two easy mistakes to avoid

- $a, \varepsilon/B$ does **not** require the stack to be empty; it simply pops nothing.
- $a, A/\varepsilon$ removes only the top A; it does **not** empty the whole stack.

A PDA configuration: state + unread input + stack

For a DFA, the current control state is enough to describe the machine. For a PDA, it is not. A complete **configuration** is written as

$$\langle q,\ w,\ \alpha \rangle,$$

where:

- q is the current state;
- w is the input still unread;
- α is the current stack content.

For example,

$$\langle q_0,\ abb,\ AA \rangle$$

means:

state q_0 | input abb remains | stack AA.

Why this matters

Two computations in the same control state can behave differently if their stacks are different.

Example 1: recognise $a^n b^n$

Consider

$$L = \{a^n b^n : n \geq 0\} = \{\varepsilon, ab, aabb, aaabbb, \ldots\}.$$

Idea: use the stack as an unbounded counter

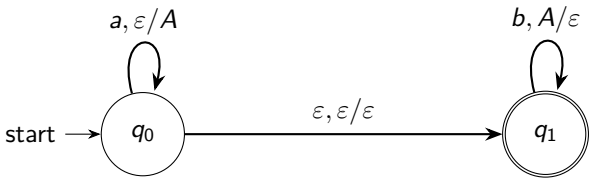
1. While reading a's, push one marker A for each a.
2. At some point, switch to the b-phase.
3. For each b, pop one A.
4. Accept only when the input and stack are both finished.

$$\underbrace{aaa}_{\text{push } A,A,A} \qquad \underbrace{bbb}_{\text{pop } A,A,A}$$

What is stored?

The stack contains exactly the number of a's that still need to be matched.

A PDA for $a^n b^n$



State q_0

Read a's and push one A for each one.

State q_1

Read b's and pop one A for each one.

The ε-transition

The machine changes from the a-phase to the b-phase without consuming an input symbol and without touching the stack.

Successful computation: aabb

Start in q_0 with the whole input unread and an empty stack:

$$\langle q_0, aabb, \varepsilon \rangle.$$

Then one successful computation is

$$\begin{aligned}
\langle q_0, aabb, \varepsilon \rangle &\Rightarrow \langle q_0, abb, A \rangle \\
&\Rightarrow \langle q_0, bb, AA \rangle \\
&\Rightarrow \langle q_1, bb, AA \rangle \\
&\Rightarrow \langle q_1, b, A \rangle \\
&\Rightarrow \langle q_1, \varepsilon, \varepsilon \rangle.
\end{aligned}$$

Success

The input is exhausted, the stack is empty, and the machine is in accepting state q_1. Therefore aabb is accepted.

What does it mean for a PDA to get stuck?

Consider the invalid word

aba.

One possible computation begins

$$\langle q_0, aba, \varepsilon \rangle \Rightarrow \langle q_0, ba, A \rangle \Rightarrow \langle q_1, ba, A \rangle \Rightarrow \langle q_1, a, \varepsilon \rangle.$$

Now the next input symbol is a, but state q_1 has no transition that can read a.

no transition applies $\Longrightarrow$ this computation is stuck.

Nondeterminism does not mean magic

A word is accepted only if **at least one complete computation** succeeds. If every possible computation gets stuck or fails, the word is rejected.

Example 2: a stack can remember order, not only count

Consider

$$L = \{wcw^R : w \in \{a, b\}^*\},$$

where w^R is the reverse of w.
Examples include c, $abcba$, and $bbcbb$. For $abcba$, we have $w = ab$ and $w^R = ba$.

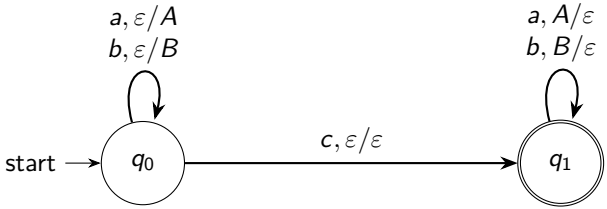
Idea

1. Before the separator c, push the letters of w onto the stack.
2. Read c and switch phase.
3. After c, pop and compare with the remaining input.

Why a stack is perfect here

Because it is last-in, first-out, the symbols of w come back in **reverse order**.

A PDA for wcw^R



Interpret the two phases

- q_0: store the first half;
- reading c: the separator tells us exactly where the middle is;
- q_1: compare the second half with the stored symbols in reverse order.

Tracing abcba

A successful computation is

$$\begin{aligned}\langle q_0, abcba, \varepsilon\rangle &\Rightarrow \langle q_0, bcba, A\rangle \\ &\Rightarrow \langle q_0, cba, BA\rangle \\ &\Rightarrow \langle q_1, ba, BA\rangle \\ &\Rightarrow \langle q_1, a, A\rangle \\ &\Rightarrow \langle q_1, \varepsilon, \varepsilon\rangle.\end{aligned}$$

Read the stack from the left:

$$\underbrace{BA}_{\text{top first}}$$

so the first symbol expected after c is b, then a.

The stack stores a reversed memory of the prefix

We pushed a and then b, so b is on top and is checked first.

Why does the separator c matter?

Compare two languages:

$$L_1 = \{wcw^R : w \in \{a, b\}^*\}, \qquad L_2 = \{ww^R : w \in \{a, b\}^*\}.$$

With a separator

In wcw^R, the symbol c tells the PDA exactly when to stop pushing and start popping.
This language can be handled deterministically.

Without a separator

For ww^R, there is no symbol marking the middle.
A PDA must in effect **guess** when to switch from pushing to popping.

A new role for nondeterminism

Unlike DFAs and NFAs, deterministic and nondeterministic PDAs do not have the same expressive power.

Balanced parentheses revisited

Balanced parentheses use exactly the same stack principle:

$$\texttt{(} \Rightarrow \text{push } X, \qquad \texttt{)} \Rightarrow \text{pop } X.$$

For `(()())`:

Input consumed	Stack
ε	ε
`(`	X
`((`	XX
`(()`	X
`(()(`	XX
`(()()`	X
`(()())`	ε

What does the stack mean for parentheses?

Each X represents

$$\boxed{\text{one opening parenthesis waiting to be closed}}.$$

Therefore:

- reading (creates an obligation: push X;
- reading) satisfies one obligation: pop X;
- if we need to pop but the stack is empty, a closing parenthesis has no match;
- if the input ends with X's remaining, some opening parentheses were never closed.

Same language, two viewpoints

The CFG

$$S \to SS \mid (S) \mid \varepsilon$$

generates balanced parentheses; a PDA with a stack **recognises** them.

When does a PDA accept?

Several equivalent conventions are common for nondeterministic PDAs.

Convention	Successful computation ends with
Final state + empty stack	input empty, stack empty, state in F
Final state only	input empty, state in F
Empty stack only	input empty, stack empty

For our worked examples

We use the first convention because it makes successful traces especially easy to read:

$$\langle q_0, w, \varepsilon \rangle \Rightarrow^* \langle q_f, \varepsilon, \varepsilon \rangle.$$

Context-free grammars and PDAs

We now have two complementary descriptions of the same language class.

Fundamental result

A language is context-free if and only if it is accepted by some nondeterministic pushdown automaton.

Examples:

$\{a^n b^n : n \geq 0\}$, $\{wcw^R : w \in \{a, b\}^*\}$, balanced parentheses.

PDAs cannot simply be determinised

For finite automata we proved

$$\boxed{\text{DFA} \equiv \text{NFA}}.$$

Every NFA can be converted to an equivalent DFA by subset construction.

PDAs cannot simply be determinised

For finite automata we proved

$$\boxed{\text{DFA} \equiv \text{NFA}}.$$

Every NFA can be converted to an equivalent DFA by subset construction.

For pushdown automata:

$$\boxed{\text{DPDA languages}} \subsetneq \boxed{\text{NPDA languages}}.$$

Some context-free languages genuinely need nondeterminism.

Examples

- $\{a^n b^n : n \geq 0\}$: deterministic PDA is enough.
- $\{wcw^R : w \in \{a, b\}^*\}$: deterministic PDA is enough because c marks the middle.
- palindromes $\{ww^R : w \in \{a, b\}^*\}$: an NPDA can guess the midpoint.

But one stack is still a restricted memory

A PDA has unbounded memory, but that memory is organised as one stack:

only the top is directly accessible.

This is enough for one unbounded matching dependency, such as

$$\{a^n b^n : n \geq 0\}.$$

But consider

$$L = \{a^n b^n c^n : n \geq 0\}.$$

Now the machine must enforce

$$\#a = \#b = \#c$$

for arbitrarily large n.

Beyond context-free languages

$\{a^n b^n c^n : n \geq 0\}$ is not context-free, so no one-stack PDA can recognise it.

Next: give the machine more flexible read/write memory.

Recognising vs generating

Perspective	Question	Example model
Recognition	Is $w \in L$?	automaton
Generation	How can w be produced?	grammar

For context-free languages:

context-free grammar $\iff$ pushdown automaton.

Example:

- a CFG describes how syntactically valid expressions can be formed;
- a parser can recognise whether a given expression follows that syntax.

The Chomsky hierarchy: the big picture

The standard hierarchy classifies grammars by how restricted their production rules are.

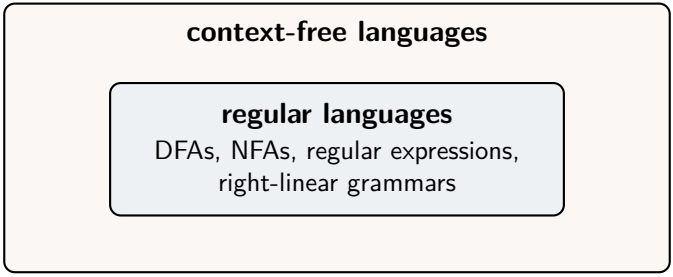
Type	Grammar	Language	Corresponding machine
3	regular	regular	finite automaton
2	context-free	context-free	pushdown automaton
1	context-sensitive	context-sensitive	linear-bounded automaton
0	unrestricted	Turing-recognisable	Turing machine

$$\text{regular} \subsetneq \text{context-free} \subsetneq \text{context-sensitive} \subsetneq \text{Turing-recognisable}.$$

For this course

We study regular and context-free languages in detail, then move to Turing machines. The intermediate context-sensitive level is shown for orientation.

The hierarchy we have used so far



regular languages $\subsetneq$ context-free languages.

Witness

The language $\{0^n 1^n : n \geq 0\}$ is context-free but not regular.

Exercise: derivations

Using

$$S \to SS \mid (S) \mid \varepsilon,$$

give a derivation for

$$()()$$

and for

$$((())).$$

Then answer:

1. Why is `(()` not generated?
2. Which grammar class does this grammar belong to?
3. Is the generated language regular or context-free?

Roadmap

Why Turing machines?

Finite automata have only finite-state memory.
Pushdown automata add an unbounded **stack**, which is enough for many nested and recursive structures.

But general computation needs more flexible memory that can:

- store unbounded intermediate information;
- revisit earlier information;
- modify that information;
- move in both directions through the memory.

Next step

This leads to the Turing machine: finite control plus an unbounded read/write tape.

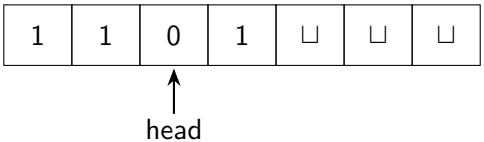
Components of a Turing machine

A deterministic Turing machine has:

- a finite set of states;
- an input alphabet;
- a tape alphabet;
- an unbounded tape;
- a read/write head;
- a transition function;
- accepting and rejecting or halting states.

The machine is finite, but its tape is unbounded.

The tape



At each step, the machine:

1. reads the current symbol;
2. writes a symbol;
3. moves left or right;
4. changes state.

Transition function

A transition has the form:

$$\delta(q, a) = (q', b, D)$$

meaning:

- in state q;
- reading symbol a;
- go to state q';
- write symbol b;
- move in direction $D \in \{L, R\}$.

First example: unary increment

Unary representation:

$$4 = 1111$$

Increment by one:

$$1111\sqcup \quad \longrightarrow \quad 11111\sqcup$$

Algorithm:

1. scan right while reading 1;
2. when the first blank is found, write 1;
3. halt.

Unary increment machine

States:

$$Q = \{q_{scan}, q_{halt}\}$$

Transitions:

$$\delta(q_{scan}, 1) = (q_{scan}, 1, R)$$

$$\delta(q_{scan}, \sqcup) = (q_{halt}, 1, R)$$

This is a machine that computes a function, not only a recogniser.

Unary increment run

Input:

$111\sqcup$

Step	Tape	State/head
0	$\underline{1}11\sqcup$	q_{scan}
1	$1\underline{1}1\sqcup$	q_{scan}
2	$11\underline{1}\sqcup$	q_{scan}
3	$111\underline{\sqcup}$	q_{scan}
4	$1111\underline{\sqcup}$	q_{halt}

Recognising 0^n1^n with a Turing machine

High-level strategy:

1. Find the leftmost unmarked 0 and replace it by X.
2. Move right to find the leftmost unmarked 1 and replace it by Y.
3. Return to the beginning.
4. Repeat.
5. Accept if only marked symbols remain.

Example:

$$000111 \to X00111 \to X00Y11 \to XX0YY1 \to XXXYYY.$$

Why this is impossible for a DFA

The Turing machine uses the tape as working memory.

It can mark symbols, move back, and compare separated parts of the input.

A DFA cannot do this:

- it cannot rewrite the input;
- it cannot move back;
- it has only finitely many states;
- it cannot remember an arbitrary number n.

Turing machines and algorithms

A Turing machine is a mathematical model of algorithmic computation.

It is not meant to be a convenient programming language.

It is useful because it is simple enough to reason about formally, yet powerful enough to capture what we usually mean by computation.

A first undecidable problem

The halting problem asks:

Given a program and an input, will the program eventually halt?

There is no general algorithm that solves this question for every possible program and input.

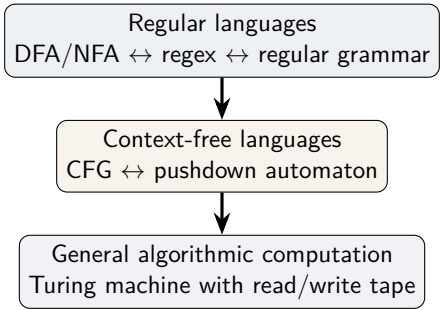
This shows that even Turing machines have limits.

Roadmap

Comparison of the main models

Model	Memory / form	Language or use	Main limitation
DFA/NFA	finite states	regular languages	no arbitrary nesting/counting
Regex	symbolic patterns	regular languages	same power as FA
Regular grammar	linear rules	regular languages	same power as FA
PDA	finite states + stack	context-free languages	stack-only memory
CFG	recursive productions	context-free languages	not general computation
Turing machine	unbounded R/W tape	general algorithms	undecidable problems remain

The main conceptual progression



Memory is the connecting idea

Finite state → stack → general read/write memory.

Exit challenge

For each language/problem, choose a suitable model.

1. Binary strings ending in 01.
2. Strings with balanced parentheses.
3. Strings of the form 0^n1^n.
4. Programs that eventually halt.

Explain not only which model works, but what memory is needed.

Expected answers

Problem	Suitable model	Reason
ends in 01	DFA / regex	finite suffix memory
balanced parentheses	CFG / PDA	nesting memory
0^n1^n	PDA / CFG / TM	stack can match the counts
halting	TM model, no decider	undecidable in general