

Introduction to Theoretical Computer Science

Zdeněk Sawa

Department of Computer Science, FEI,
Technical University of Ostrava
17. listopadu 15, Ostrava-Poruba 708 33
Czech republic

May 19, 2017

Name: doc. Ing. Zdeněk Sawa, Ph.D.

E-mail: `zdenek.sawa@vsb.cz`

Room: EA413

Web: <http://www.cs.vsb.cz/sawa/uti/index-en.html>

- Information about the course
- Study texts
- Slides from lectures
- Exercises for tutorials
- Recent news for the course
- A link to a page with animations

- **Credit** (22 points):
 - Written test (16 points) — it will be written on a tutorial
 - The minimal requirement for obtaining the credit is 9 points.
 - A correcting test for 14 points.
 - Activity on tutorials (6 points)
 - The minimal requirement for obtaining the credit is 3 points.
- **Exam** (78 points)
 - A written exam consisting of three parts (26 points for each part); it is necessary to obtain at least 10 points for each part.

Theoretical computer science — a scientific field on the border between computer science and mathematics

- investigation of general questions concerning algorithms and computations
- study of different kinds of formalisms for description of algorithms
- study of different approaches for description of syntax and semantics of formal languages (mainly programming languages)
- a mathematical approach to analysis and solution of problems (proofs of general mathematical propositions concerning algorithms)

Examples of some typical questions studied in theoretical computer science:

- Is it possible to solve the given problem using some algorithm?
- If the given problem can be solved by an algorithm, what is the computational complexity of this algorithm?
- Is there an efficient algorithm solving the given problem?
- How to check that a given algorithm is really a correct solution of the given problem?
- What kinds instructions are sufficient for a given machine to perform a given algorithm?

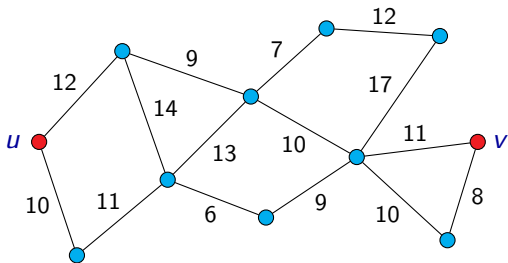
An example of an algorithmic problem

Problem “Finding the shortest path in an (undirected) graph’

Input: An undirected graph $G = (V, E)$ with edges labelled with numbers, and a pair of nodes $u, v \in V$.

Output: The shortest path from node u to node v .

Example:



Theoretical computer science overlaps with many other areas of mathematics and computer science:

- graph theory
- number theory
- computational geometry
- searching in text
- game theory
- ...

Logic — study of reasoning and argumentation

- it studies when a conclusion follows from given assumptions
- it studies questions concerning proofs and provability
- it provides a basic language for mathematics and all sciences based on mathematics
- it is connected with study of foundations of mathematics
- it is used in computer science on many different levels

Propositional Logic

- *If the train arrives late and there are no taxis at the station, then John is late for his meeting.*
 - *John is not late for his meeting.*
 - *The train did arrive late.*
-
- *There were taxis at the station.*

- *If it is raining and Jane does not have her umbrella with her, then she will get wet.*
 - *Jane is not wet.*
 - *It is raining.*
-
- *Jane has her umbrella with her.*

p	The train is late.	It is raining.
q	There are taxis at the station.	Jane has her umbrella with her.
r	John is late for his meeting.	Jane gets wet.

If p and not q , then r .

Not r .

p .

q .

Examples of propositions:

- *"Jane gets wet."*
- *"If it is raining and Jane does not have her umbrella with her, then she will get wet."*
- *"Paris is the capital of Japan."*
- *"There are infinitely many primes."*
- *" $1 + 1 = 3$ "*
- *"Number $\sqrt{2}$ is irrational."*

Atomic proposition — it cannot be decomposed into simpler propositions

“It is raining.”

Compound proposition — it is composed from some simpler propositions

“If it is raining and Jane does not have her umbrella with her, then she will get wet.”

It consists of propositions:

- *“It is raining.”*
- *“Jane has her umbrella with her.”*
- *“Jane gets wet.”*

Logical Connectives

More complicated propositions can be constructed from simpler propositions using **logical connectives**:

Symbol	Log. connective	Example of use	Informal meaning
$\neg$	negation	$\neg p$	"not p "
$\wedge$	conjunction	$p \wedge q$	" p and q "
$\vee$	disjunction	$p \vee q$	" p or q "
$\rightarrow$	implication	$p \rightarrow q$	"if p then q "
$\leftrightarrow$	equivalence	$p \leftrightarrow q$	" p if and only if q "

Atomic propositions — $p, q, r, \dots$
(possibly with indexes — $p_0, p_1, p_2, \dots$)

Logical Connectives

Propositions are represented using **formulas** — formulas have a precisely defined syntax and semantics.

“If it is raining and Jane does not have her umbrella with her, then she will get wet.”

The proposition written as a formula:

$$(p \wedge \neg q) \rightarrow r$$

Atomic propositions:

- p — *“It is raining.”*
- q — *“Jane has her umbrella with her.”*
- r — *“Jane gets wet.”*

Truth Values

1	0
true	false
yes	no

We will use 0 and 1 to denote the truth values.

The truth values are also called **boolean** values.

Negation

The **negation** of a proposition φ is the proposition “not φ ”, or the proposition “*it is not the case that φ* ”. For example, the negation of

“the number 5 is a prime”

is a proposition

“it is not true that the number 5 is a prime”

or

“the number 5 is not a prime”.

In formulas, negation is denoted by symbol “ $\neg$ ”.

Formally: $\neg\varphi$

φ	$\neg\varphi$
0	1
1	0

Conjunction

The **conjunction** of propositions φ and ψ is proposition “ φ and ψ ”.

Example: The conjunction of propositions
“*Copenhagen is the capital of Denmark*” and “ $2 + 2 = 4$ ” is proposition
“*Copenhagen is the capital of Denmark and $2 + 2 = 4$.*”

In formulas, conjunction is denoted by symbol “ $\wedge$ ”.

$$p \wedge q$$

- p — “*Copenhagen is the capital of Denmark*”
- q — “ $2 + 2 = 4$ ”

Conjunction

φ	ψ	$\varphi \wedge \psi$
0	0	0
0	1	0
1	0	0
1	1	1

Examples of false propositions:

- *"Helsinki are the capital of Italy and Charles University was founded in 1348."*
- *"Asia is the largest continent and $3 + 5 = 14$."*
- *"There are only finitely many primes and Pilsen is the capital of USA."*

Disjunction

The **disjunction** of propositions φ and ψ is proposition “ φ or ψ ”.

Example: The disjunction of propositions “*whales are mammals*” and “*Czech Republic is in Europe*” is proposition

“*whales are mammals or Czech Republic is in Europe*”.

In formulas, disjunction is denoted by symbol “ $\vee$ ”.

$$p \vee q$$

- p — “*whales are mammals*”
- q — “*Czech Republic is in Europe*”

Disjunction

Disjunction is “or” in **non-exclusive** sense.

φ	ψ	$\varphi \vee \psi$
0	0	0
0	1	1
1	0	1
1	1	1

Implication

Implication — “if φ then ψ ”

- φ — assumption (hypothesis)
- ψ — conclusion

Example:

“If Peter was well-prepared for the exam, he obtained a good grade.”

Implication is denoted by symbol “ $\rightarrow$ ”.

$$p \rightarrow q$$

- p — “Peter was well-prepared for the exam”
- q — “Peter obtained a good grade”

Implication

φ	ψ	$\varphi \rightarrow \psi$
0	0	1
0	1	1
1	0	0
1	1	1

Remark: Formula $p \rightarrow q$ is true exactly in those cases where the following formula is true:

$$\neg p \vee q$$

Implication does **not** express causal dependence.

Example:

- “If Washington is the capital of USA, then $1 + 1 = 2$.”
- “If Washington is the capital of USA, then $1 + 1 = 3$.”
- “If Tokyo is the capital of USA, then $1 + 1 = 2$.”
- “If Tokyo is the capital of USA, then $1 + 1 = 3$.”

Implication $p \rightarrow q$ can be expressed in a natural language in many different ways:

- “ q if p ”
- “ p only if q ”
- “ p implies q ”
- “ q provided that p ”
- “from p follows q ”
- “ p is a sufficient condition for q ”
- “ q is a necessary condition for p ”

If $\varphi \rightarrow \psi$ is true and also φ is true, we can infer from this that also ψ is true.

Example: When it holds that

- *“if today is Tuesday, then tomorrow is Wednesday”*
- *“today is Tuesday”*

we can infer from this that

- *“tomorrow is Wednesday”*

Equivalence — “ ϕ if and only if ψ ”

Example:

“Triangle ABC has all three sides of the same length if and only if it has all three angles of the same size.”

The logical connective equivalence is denoted by symbol “ $\leftrightarrow$ ”

$$p \leftrightarrow q$$

- p — *“triangle ABC has all three sides of the same length”*
- q — *“triangle ABC has all three angles of the same size”*

Equivalence

φ	ψ	$\varphi \leftrightarrow \psi$
0	0	1
0	1	0
1	0	0
1	1	1

Remark: Formula $p \leftrightarrow q$ says basically the same thing as

$$(p \rightarrow q) \wedge (q \rightarrow p)$$

Alternatives for expressing equivalence $p \leftrightarrow q$:

- “ p is a necessary and sufficient condition for q ”
- “ p iff q ”

Equivalence is often used in **definitions** of new notions:

Example:

- “A triangle is isoscele if and only if at least two of its sides are equal in length.”
- “A triangle is isoscele if at least two of its sides are equal in length.”

Formulas of Propositional Logic

- **Syntax** — what are well-formed formulas of propositional logic
- **Semantics** — assigns meaning to formulas and to individual symbols occurring in these formulas

Syntax of Formulas of Propositional Logic

Formulas — sequences of symbols from a given **alphabet**:

- **atomic propositions** — for example symbols “ p ”, “ q ”, “ r ”, etc.
- **logical connectives** — symbols “ $\neg$ ”, “ $\wedge$ ”, “ $\vee$ ”, “ $\rightarrow$ ”, “ $\leftrightarrow$ ”
- **parentheses** — symbols “ $($ ” and “ $)$ ”

Not every sequence of these symbols is a formula.

For example, this is not a formula:

$\wedge \vee) p \neg ((\neg$

Syntax of Formulas of Propositional Logic

Definition

Well-formed **formulas of propositional logic** are sequences of symbols constructed according to the following rules:

- 1 If p is an atomic proposition, then p is a well-formed formula.
- 2 If φ and ψ are well-formed formulas, then also $(\neg\varphi)$, $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\varphi \rightarrow \psi)$ and $(\varphi \leftrightarrow \psi)$ are well-formed formulas.
- 3 There are no other well-formed formulas than those constructed according to two previous rules.

Syntax of Formulas of Propositional Logic

Examples of well-formed formulas:

- q
- $(\neg q)$
- r
- $((\neg q) \rightarrow r)$
- p
- $(p \leftrightarrow r)$
- $(\neg(p \leftrightarrow r))$
- $((\neg q) \rightarrow r) \wedge (\neg(p \leftrightarrow r))$

An example of a sequence of symbols, which is not a well-formed formula:

- $(p \wedge \vee q)$

Syntax of Formulas of Propositional Logic

Formula ψ is a **subformula** of formula φ if at least one of the following possibilities holds:

- Formula ψ is the same formula as formula φ .
- If formula φ is of the form $(\neg\chi)$, then ψ is a subformula of formula χ .
- If formula φ is of the form $(\chi_1 \wedge \chi_2)$, $(\chi_1 \vee \chi_2)$, $(\chi_1 \rightarrow \chi_2)$, or $(\chi_1 \leftrightarrow \chi_2)$, then ψ is a subformula of formula χ_1 or a subformula of formula χ_2 .

Example: Subformulas of formula $((\neg(p \wedge q)) \leftrightarrow r)$:

p q r $(p \wedge q)$ $(\neg(p \wedge q))$ $((\neg(p \wedge q)) \leftrightarrow r)$

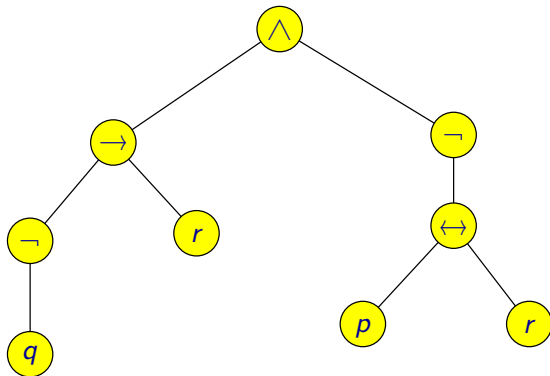
Syntax of Formulas of Propositional Logic

Alternative symbols for logical connectives:

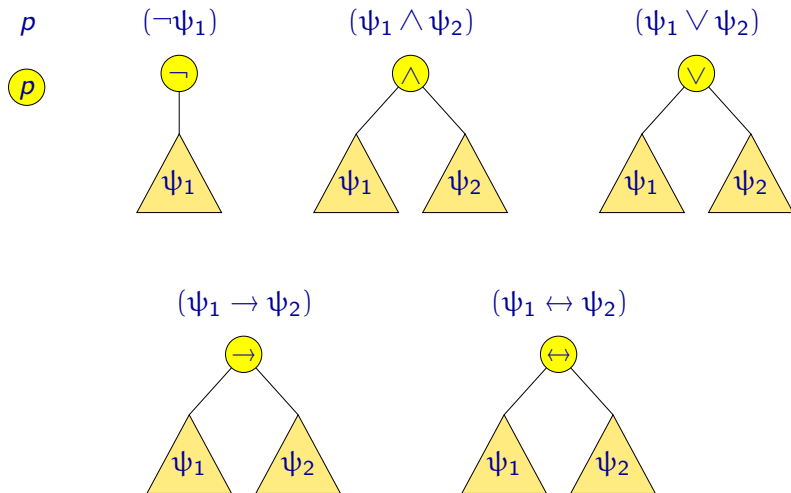
Connective	Symbol	Alternative symbols
negation	$\neg$	$\sim$
conjunction	$\wedge$	$\&$
implication	$\rightarrow$	$\Rightarrow, \supset$
equivalence	$\leftrightarrow$	$\Leftrightarrow, \equiv$

Syntax of Formulas of Propositional Logic

An abstract syntax tree of formula $((\neg q \rightarrow r) \wedge (\neg(p \leftrightarrow r)))$:



Syntax of Formulas of Propositional Logic



Arity of logical connectives:

- **unary connective** (arity 1): $\neg$
- **binary connectives** (arity 2): $\wedge, \vee, \rightarrow, \leftrightarrow$

Syntax of Formulas of Propositional Logic

Conventions for omitting parentheses:

- Outermost pair of parentheses can be omitted.
- Priority of logical connectives (from the highest to the lowest):

$$\neg \quad \wedge \quad \vee \quad \rightarrow \quad \leftrightarrow$$

- Instead of $\neg(\neg\varphi)$, it is possible to write $\neg\neg\varphi$.

Example: Instead of $((\neg p) \wedge (r \rightarrow (q \vee s)))$, it is possible to write

$$\neg p \wedge (r \rightarrow q \vee s)$$

Remark: Other conventions will be described later.

Semantics of Propositional Logic

At — a set of atomic propositions

For example

- $At = \{p, q, r\}$, or
- $At = \{p_0, p_1, p_2, \dots\}$

Definition

A **truth valuation** is an assignment of truth values (i.e., values from the set $\{0, 1\}$) to all atomic propositions from the set At .

(Formally, a truth valuation can be defined as a function $v : At \rightarrow \{0, 1\}$.)

Example: A truth valuation v for $At = \{p, q, r\}$, where

$$v(p) = 1 \quad v(q) = 0 \quad v(r) = 1$$

Semantics of Propositional Logic

If set At is finite and contains n atomic propositions, then there are 2^n truth valuations.

Example: $At = \{p, q, r\}$

v_0 :	$v_0(p) = 0,$	$v_0(q) = 0,$	$v_0(r) = 0$
v_1 :	$v_1(p) = 0,$	$v_1(q) = 0,$	$v_1(r) = 1$
v_2 :	$v_2(p) = 0,$	$v_2(q) = 1,$	$v_2(r) = 0$
v_3 :	$v_3(p) = 0,$	$v_3(q) = 1,$	$v_3(r) = 1$
v_4 :	$v_4(p) = 1,$	$v_4(q) = 0,$	$v_4(r) = 0$
v_5 :	$v_5(p) = 1,$	$v_5(q) = 0,$	$v_5(r) = 1$
v_6 :	$v_6(p) = 1,$	$v_6(q) = 1,$	$v_6(r) = 0$
v_7 :	$v_7(p) = 1,$	$v_7(q) = 1,$	$v_7(r) = 1$

Semantics of Propositional Logic

At truth valuation v , formula φ has truth value 1:

$$v \models \varphi$$

At truth valuation v , formula φ has truth value 0:

$$v \not\models \varphi$$

Definition

Truth values of formulas of propositional logic in a given truth valuation v are defined as follows:

- For atomic proposition p , $v \models p$ iff $v(p) = 1$.
(So, if $v(p) = 0$, then $v \not\models p$.)
- $v \models \neg\varphi$ iff $v \not\models \varphi$.
- $v \models \varphi \wedge \psi$ iff $v \models \varphi$ and $v \models \psi$.
- $v \models \varphi \vee \psi$ iff $v \models \varphi$ or $v \models \psi$.
- $v \models \varphi \rightarrow \psi$ iff $v \not\models \varphi$ or $v \models \psi$.
- $v \models \varphi \leftrightarrow \psi$ iff $v \models \varphi$ and $v \models \psi$, or $v \not\models \varphi$ and $v \not\models \psi$.

Semantics of Propositional Logic

φ	$\neg\varphi$
0	1
1	0

φ	ψ	$\varphi \wedge \psi$	$\varphi \vee \psi$	$\varphi \rightarrow \psi$	$\varphi \leftrightarrow \psi$
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	0	0
1	1	1	1	1	1

Semantics of Propositional Logic

Example: $At = \{p, q, r\}$

valuation v , where $v(p) = 1$, $v(q) = 0$, and $v(r) = 1$

p	q	r	$\neg q$	$\neg q \rightarrow r$	$p \leftrightarrow r$	$\neg(p \leftrightarrow r)$	φ

$$\varphi := (\neg q \rightarrow r) \wedge \neg(p \leftrightarrow r)$$

Semantics of Propositional Logic

Example: $At = \{p, q, r\}$

valuation v , where $v(p) = 1$, $v(q) = 0$, and $v(r) = 1$

- $v \models p$
- $v \not\models q$
- $v \models r$

p	q	r	$\neg q$	$\neg q \rightarrow r$	$p \leftrightarrow r$	$\neg(p \leftrightarrow r)$	φ
1	0	1					

$$\varphi := (\neg q \rightarrow r) \wedge \neg(p \leftrightarrow r)$$

Semantics of Propositional Logic

Example: $At = \{p, q, r\}$

valuation v , where $v(p) = 1$, $v(q) = 0$, and $v(r) = 1$

- $v \models p$
- $v \not\models q$
- $v \models r$
- $v \models \neg q$

p	q	r	$\neg q$	$\neg q \rightarrow r$	$p \leftrightarrow r$	$\neg(p \leftrightarrow r)$	φ
1	0	1	1				

$$\varphi := (\neg q \rightarrow r) \wedge \neg(p \leftrightarrow r)$$

Semantics of Propositional Logic

Example: $At = \{p, q, r\}$

valuation v , where $v(p) = 1$, $v(q) = 0$, and $v(r) = 1$

- $v \models p$
- $v \not\models q$
- $v \models r$
- $v \models \neg q$
- $v \models \neg q \rightarrow r$

p	q	r	$\neg q$	$\neg q \rightarrow r$	$p \leftrightarrow r$	$\neg(p \leftrightarrow r)$	φ
1	0	1	1	1			

$$\varphi := (\neg q \rightarrow r) \wedge \neg(p \leftrightarrow r)$$

Semantics of Propositional Logic

Example: $At = \{p, q, r\}$

valuation v , where $v(p) = 1$, $v(q) = 0$, and $v(r) = 1$

- $v \models p$
- $v \not\models q$
- $v \models r$
- $v \models \neg q$
- $v \models \neg q \rightarrow r$
- $v \models p \leftrightarrow r$

p	q	r	$\neg q$	$\neg q \rightarrow r$	$p \leftrightarrow r$	$\neg(p \leftrightarrow r)$	φ
1	0	1	1	1	1		

$$\varphi := (\neg q \rightarrow r) \wedge \neg(p \leftrightarrow r)$$

Semantics of Propositional Logic

Example: $At = \{p, q, r\}$

valuation v , where $v(p) = 1$, $v(q) = 0$, and $v(r) = 1$

- $v \models p$
- $v \not\models q$
- $v \models r$
- $v \models \neg q$
- $v \models \neg q \rightarrow r$
- $v \models p \leftrightarrow r$
- $v \not\models \neg(p \leftrightarrow r)$

p	q	r	$\neg q$	$\neg q \rightarrow r$	$p \leftrightarrow r$	$\neg(p \leftrightarrow r)$	φ
1	0	1	1	1	1	0	

$$\varphi := (\neg q \rightarrow r) \wedge \neg(p \leftrightarrow r)$$

Semantics of Propositional Logic

Example: $At = \{p, q, r\}$

valuation v , where $v(p) = 1$, $v(q) = 0$, and $v(r) = 1$

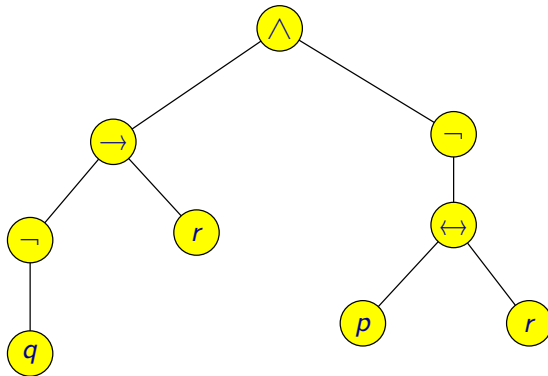
- $v \models p$
- $v \not\models q$
- $v \models r$
- $v \models \neg q$
- $v \models \neg q \rightarrow r$
- $v \models p \leftrightarrow r$
- $v \not\models \neg(p \leftrightarrow r)$
- $v \not\models (\neg q \rightarrow r) \wedge \neg(p \leftrightarrow r)$

p	q	r	$\neg q$	$\neg q \rightarrow r$	$p \leftrightarrow r$	$\neg(p \leftrightarrow r)$	φ
1	0	1	1	1	1	0	0

$$\varphi := (\neg q \rightarrow r) \wedge \neg(p \leftrightarrow r)$$

Semantics of Propositional Logic

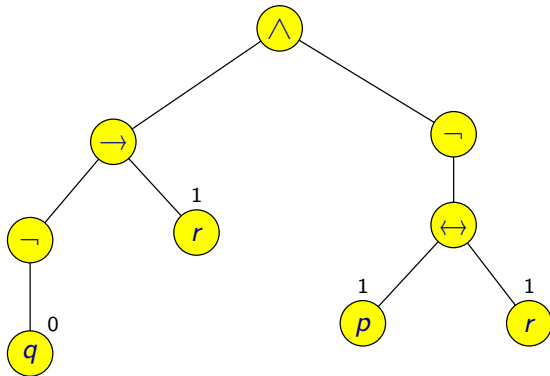
$$\varphi := (\neg q \rightarrow r) \wedge \neg(p \leftrightarrow r)$$



p	q	r	$\neg q$	$\neg q \rightarrow r$	$p \leftrightarrow r$	$\neg(p \leftrightarrow r)$	φ

Semantics of Propositional Logic

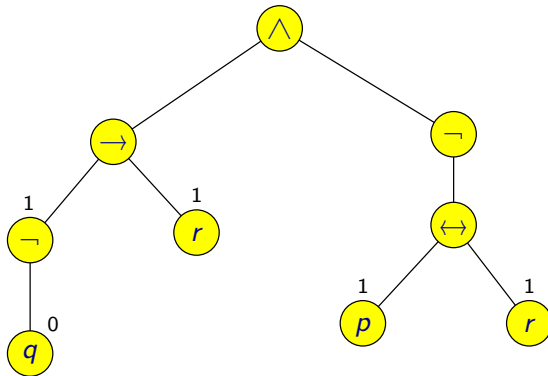
$$\varphi := (\neg q \rightarrow r) \wedge \neg(p \leftrightarrow r)$$



p	q	r	$\neg q$	$\neg q \rightarrow r$	$p \leftrightarrow r$	$\neg(p \leftrightarrow r)$	φ
1	0	1					

Semantics of Propositional Logic

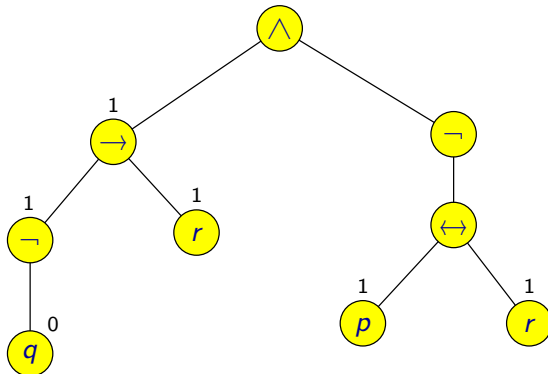
$$\varphi := (\neg q \rightarrow r) \wedge \neg(p \leftrightarrow r)$$



p	q	r	$\neg q$	$\neg q \rightarrow r$	$p \leftrightarrow r$	$\neg(p \leftrightarrow r)$	φ
1	0	1	1				

Semantics of Propositional Logic

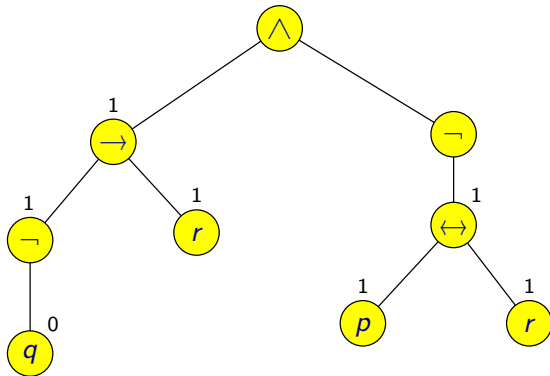
$$\varphi := (\neg q \rightarrow r) \wedge \neg(p \leftrightarrow r)$$



p	q	r	$\neg q$	$\neg q \rightarrow r$	$p \leftrightarrow r$	$\neg(p \leftrightarrow r)$	φ
1	0	1	1	1			

Semantics of Propositional Logic

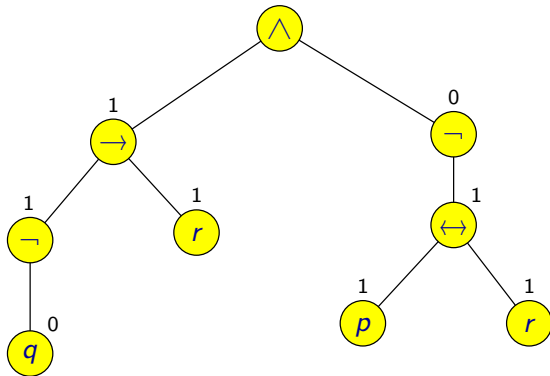
$$\varphi := (\neg q \rightarrow r) \wedge \neg(p \leftrightarrow r)$$



p	q	r	$\neg q$	$\neg q \rightarrow r$	$p \leftrightarrow r$	$\neg(p \leftrightarrow r)$	φ
1	0	1	1	1	1		

Semantics of Propositional Logic

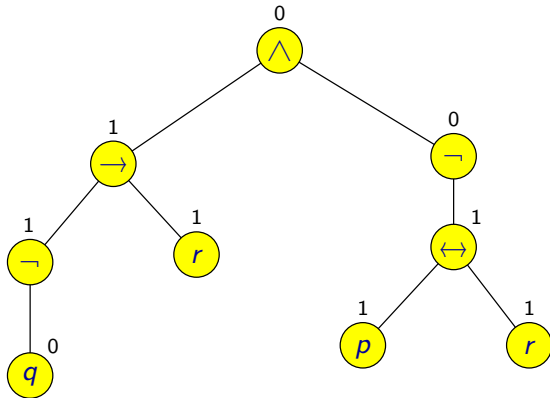
$$\varphi := (\neg q \rightarrow r) \wedge \neg(p \leftrightarrow r)$$



p	q	r	$\neg q$	$\neg q \rightarrow r$	$p \leftrightarrow r$	$\neg(p \leftrightarrow r)$	φ
1	0	1	1	1	1	0	

Semantics of Propositional Logic

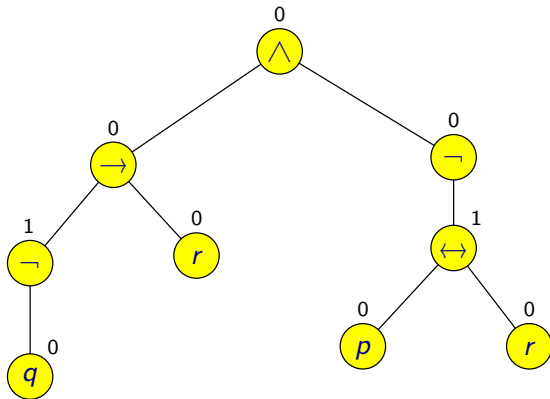
$$\varphi := (\neg q \rightarrow r) \wedge \neg(p \leftrightarrow r)$$



p	q	r	$\neg q$	$\neg q \rightarrow r$	$p \leftrightarrow r$	$\neg(p \leftrightarrow r)$	φ
1	0	1	1	1	1	0	0

Semantics of Propositional Logic

$$\varphi := (\neg q \rightarrow r) \wedge \neg(p \leftrightarrow r)$$



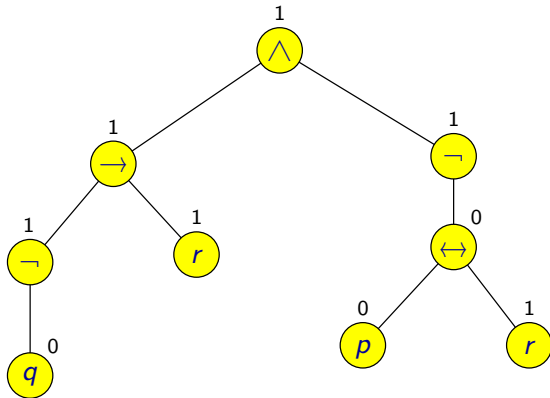
$$v_0(p) = 0$$

$$v_0(q) = 0$$

$$v_0(r) = 0$$

Semantics of Propositional Logic

$$\varphi := (\neg q \rightarrow r) \wedge \neg(p \leftrightarrow r)$$



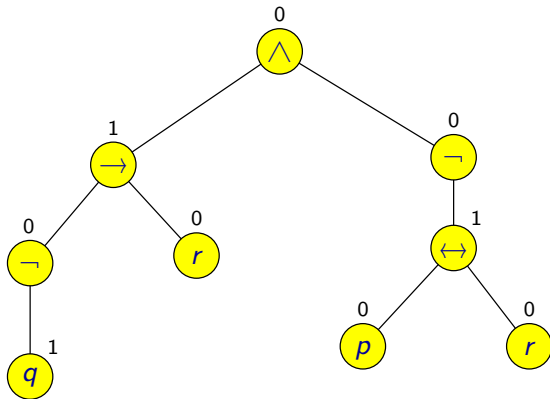
$$v_1(p) = 0$$

$$v_1(q) = 0$$

$$v_1(r) = 1$$

Semantics of Propositional Logic

$$\varphi := (\neg q \rightarrow r) \wedge \neg(p \leftrightarrow r)$$



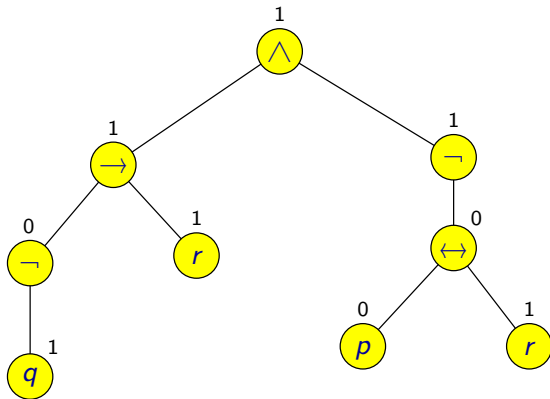
$$v_2(p) = 0$$

$$v_2(q) = 1$$

$$v_2(r) = 0$$

Semantics of Propositional Logic

$$\varphi := (\neg q \rightarrow r) \wedge \neg(p \leftrightarrow r)$$



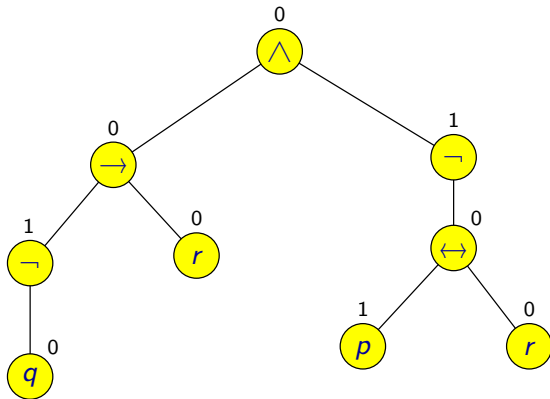
$$v_3(p) = 0$$

$$v_3(q) = 1$$

$$v_3(r) = 1$$

Semantics of Propositional Logic

$$\varphi := (\neg q \rightarrow r) \wedge \neg(p \leftrightarrow r)$$



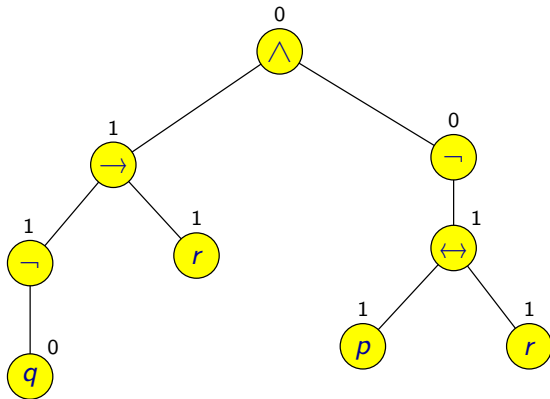
$$v_4(p) = 1$$

$$v_4(q) = 0$$

$$v_4(r) = 0$$

Semantics of Propositional Logic

$$\varphi := (\neg q \rightarrow r) \wedge \neg(p \leftrightarrow r)$$



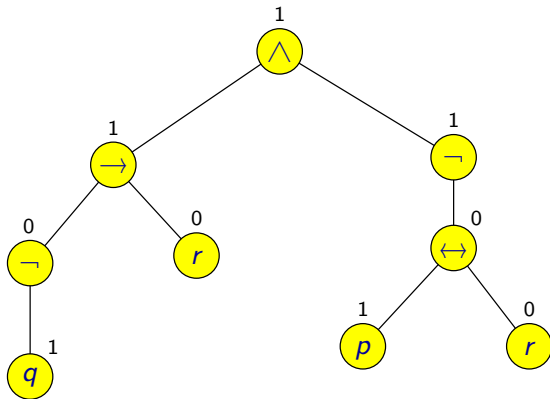
$$v_5(p) = 1$$

$$v_5(q) = 0$$

$$v_5(r) = 1$$

Semantics of Propositional Logic

$$\varphi := (\neg q \rightarrow r) \wedge \neg(p \leftrightarrow r)$$



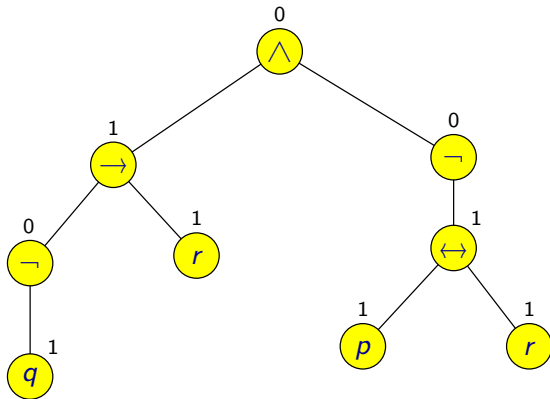
$$v_6(p) = 1$$

$$v_6(q) = 1$$

$$v_6(r) = 0$$

Semantics of Propositional Logic

$$\varphi := (\neg q \rightarrow r) \wedge \neg(p \leftrightarrow r)$$



$$v_7(p) = 1$$

$$v_7(q) = 1$$

$$v_7(r) = 1$$

Semantics of Propositional Logic

$$\varphi := (\neg q \rightarrow r) \wedge \neg(p \leftrightarrow r)$$

p	q	r	$\neg q$	$\neg q \rightarrow r$	$p \leftrightarrow r$	$\neg(p \leftrightarrow r)$	φ
0	0	0	1	0	1	0	0
0	0	1	1	1	0	1	1
0	1	0	0	1	1	0	0
0	1	1	0	1	0	1	1
1	0	0	1	0	0	1	0
1	0	1	1	1	1	0	0
1	1	0	0	1	0	1	1
1	1	1	0	1	1	0	0

Those valuations, where the given formula is true, are called its **models**:

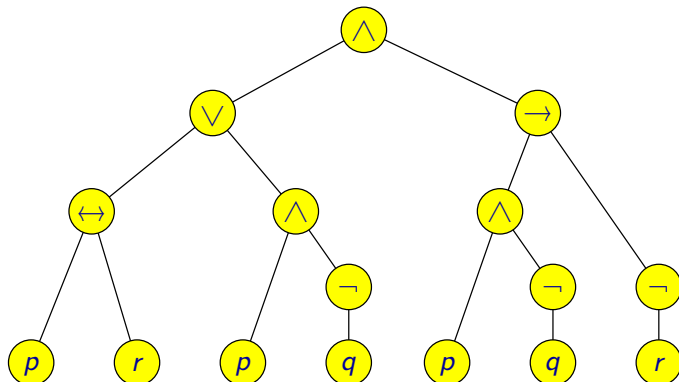
$$v_1: \quad v_1(p) = 0, \quad v_1(q) = 0, \quad v_1(r) = 1,$$

$$v_3: \quad v_3(p) = 0, \quad v_3(q) = 1, \quad v_3(r) = 1,$$

$$v_6: \quad v_6(p) = 1, \quad v_6(q) = 1, \quad v_6(r) = 0,$$

Formulas Represented as Directed Acyclic Graphs

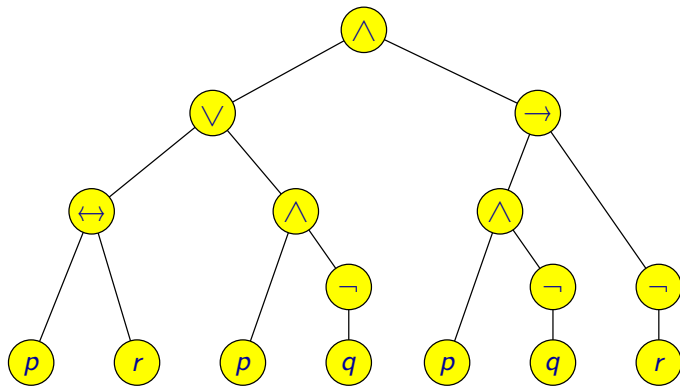
$$\varphi := ((p \leftrightarrow r) \vee (p \wedge \neg q)) \wedge ((p \wedge \neg q) \rightarrow \neg r)$$



In an abstract syntax tree, nodes correspond to **occurrences** of subformulas.

Formulas Represented as Directed Acyclic Graphs

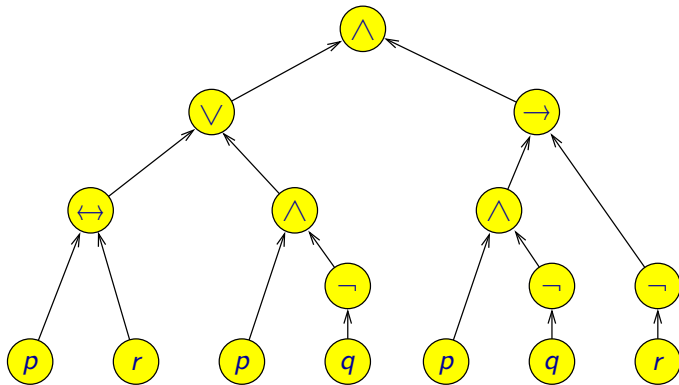
$$\varphi := ((p \leftrightarrow r) \vee (p \wedge \neg q)) \wedge ((p \wedge \neg q) \rightarrow \neg r)$$



Alternatively, a formula can be represented as a **directed acyclic graph**.

Formulas Represented as Directed Acyclic Graphs

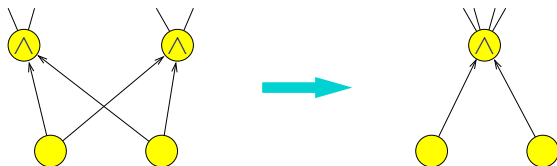
$$\varphi := ((p \leftrightarrow r) \vee (p \wedge \neg q)) \wedge ((p \wedge \neg q) \rightarrow \neg r)$$



Edges are oriented from children to their parents.

Formulas Represented as Directed Acyclic Graphs

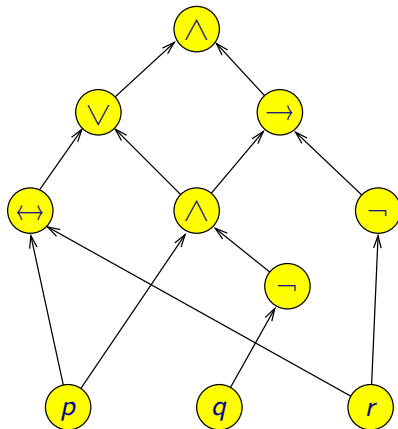
- Leafs labelled with the same atomic proposition are merged together.
- Those nodes, which are labelled with the same symbol and which have the same predecessors, can be merged.



- Such merging of nodes can be done repeatedly — when some nodes are merged, merging of some other nodes may become possible.

Formulas Represented as Directed Acyclic Graphs

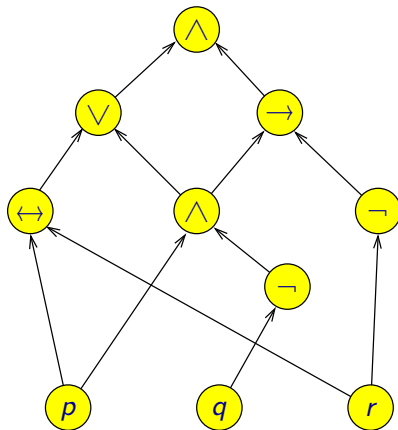
$$\varphi := ((p \leftrightarrow r) \vee (p \wedge \neg q)) \wedge ((p \wedge \neg q) \rightarrow \neg r)$$



- When we merge all nodes that can be possibly merged this way, we obtain a graph where individual nodes correspond to different subformulas of a given formula.
- A directed acyclic graph representing a given formula can be viewed as a **logic circuit**:
 - **Inputs** — nodes labelled with atomic propositions
 - **Output** — the node corresponding to the whole formula

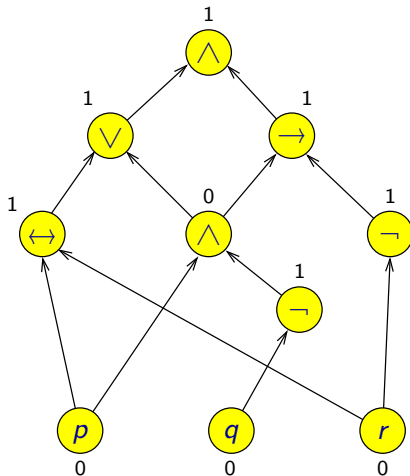
Formulas Represented as Directed Acyclic Graphs

$$\varphi := ((p \leftrightarrow r) \vee (p \wedge \neg q)) \wedge ((p \wedge \neg q) \rightarrow \neg r)$$



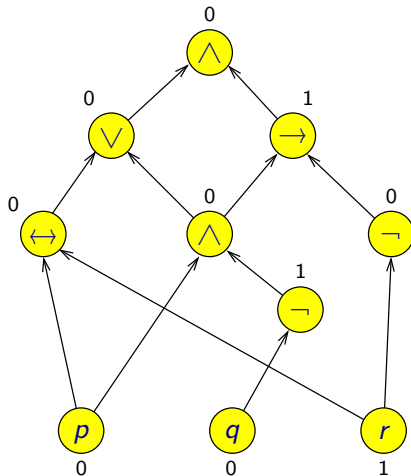
Formulas Represented as Directed Acyclic Graphs

$$\varphi := ((p \leftrightarrow r) \vee (p \wedge \neg q)) \wedge ((p \wedge \neg q) \rightarrow \neg r)$$



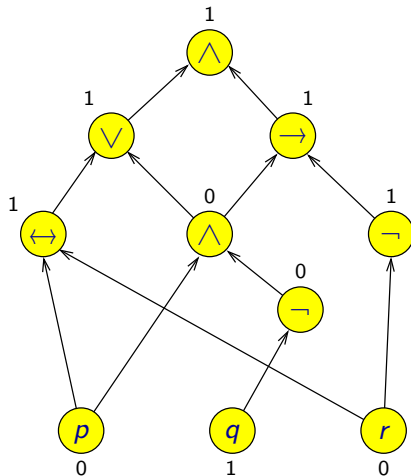
Formulas Represented as Directed Acyclic Graphs

$$\varphi := ((p \leftrightarrow r) \vee (p \wedge \neg q)) \wedge ((p \wedge \neg q) \rightarrow \neg r)$$



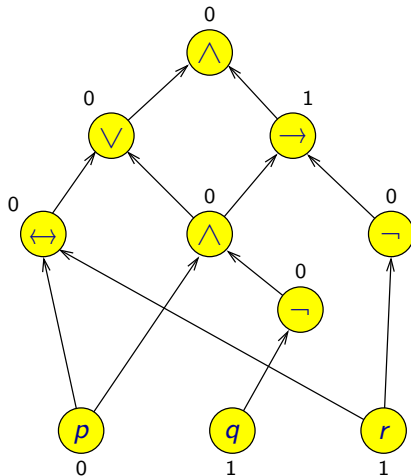
Formulas Represented as Directed Acyclic Graphs

$$\varphi := ((p \leftrightarrow r) \vee (p \wedge \neg q)) \wedge ((p \wedge \neg q) \rightarrow \neg r)$$



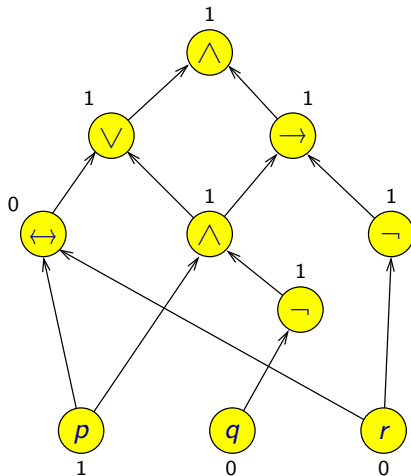
Formulas Represented as Directed Acyclic Graphs

$$\varphi := ((p \leftrightarrow r) \vee (p \wedge \neg q)) \wedge ((p \wedge \neg q) \rightarrow \neg r)$$



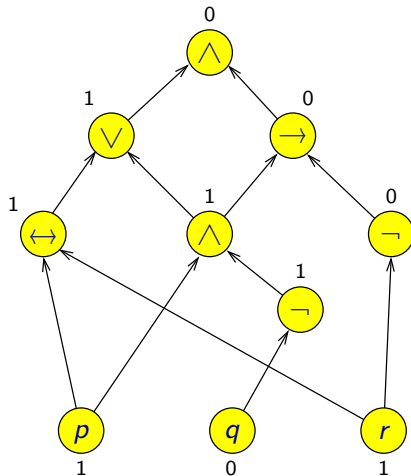
Formulas Represented as Directed Acyclic Graphs

$$\varphi := ((p \leftrightarrow r) \vee (p \wedge \neg q)) \wedge ((p \wedge \neg q) \rightarrow \neg r)$$



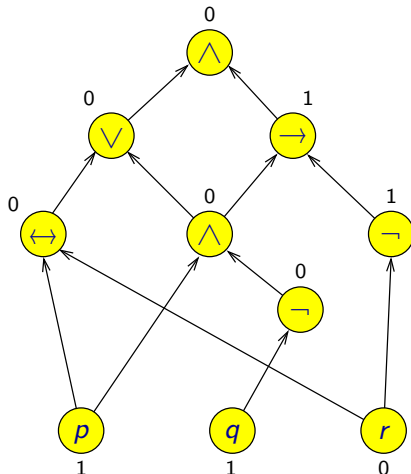
Formulas Represented as Directed Acyclic Graphs

$$\varphi := ((p \leftrightarrow r) \vee (p \wedge \neg q)) \wedge ((p \wedge \neg q) \rightarrow \neg r)$$



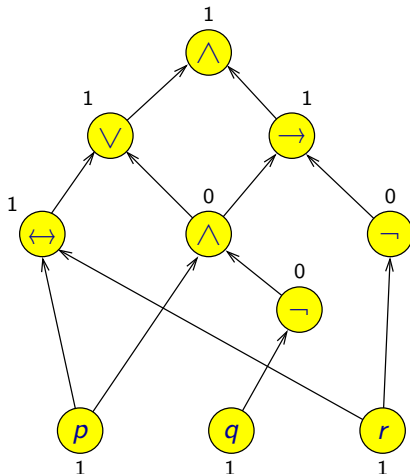
Formulas Represented as Directed Acyclic Graphs

$$\varphi := ((p \leftrightarrow r) \vee (p \wedge \neg q)) \wedge ((p \wedge \neg q) \rightarrow \neg r)$$



Formulas Represented as Directed Acyclic Graphs

$$\varphi := ((p \leftrightarrow r) \vee (p \wedge \neg q)) \wedge ((p \wedge \neg q) \rightarrow \neg r)$$



Definition

Formula φ is a **tautology** if $v \models \varphi$ holds for every truth valuation v (i.e., if φ is true in every valuation).

Example: *"If it is raining, then it is raining."*

$$p \rightarrow p$$

Example: *"It is Friday today, or it is not Friday today."*

$$q \vee \neg q$$

An example of a more complicated tautology:

$$(p \rightarrow q) \rightarrow ((p \rightarrow \neg q) \rightarrow \neg p)$$

p	q	$p \rightarrow q$	$\neg q$	$p \rightarrow \neg q$	$\neg p$	$(p \rightarrow \neg q) \rightarrow \neg p$	φ
0	0	1	1	1	1	1	1
0	1	1	0	1	1	1	1
1	0	0	1	1	0	0	1
1	1	1	0	0	0	1	1

Tautologies

Quite important are tautologies of the form $\varphi \rightarrow \psi$ or $\varphi \leftrightarrow \psi$ — they can be used for logical inference:

- If $\varphi \rightarrow \psi$ holds and φ holds, then also ψ must hold.

In particular, if $\varphi \rightarrow \psi$ is a tautology and φ holds, we can deduce that also ψ holds.

Example: $(p \wedge q) \rightarrow p$ is a tautology.

If $p \wedge q$ holds, then also p holds.

Example: $(p \rightarrow q) \rightarrow (\neg q \rightarrow \neg p)$ is a tautology.

If $p \rightarrow q$ holds and $\neg q$ holds, then $\neg p$ holds.

- If $\varphi \leftrightarrow \psi$ holds and φ holds, then ψ must hold.
Similarly, if $\varphi \leftrightarrow \psi$ holds and ψ holds, then φ must hold.

Example: $(\neg p \rightarrow q) \leftrightarrow (q \vee p)$ is a tautology.

- If $\neg p \rightarrow q$ holds, then also $q \vee p$ must hold.
- If $q \vee p$ holds, then also $\neg p \rightarrow q$ must hold.

Tautologies

When we take a tautology φ and replace all atomic propositions by arbitrary formulas, we obtain a tautology by this replacement.

Example: Formula $p \rightarrow (p \vee q)$ is a tautology.

This means that

$$\psi \rightarrow (\psi \vee \chi)$$

is a tautology for arbitrary formulas ψ and χ .

Replacement of atomic propositions:

- p is replaced with $q \vee \neg(r \rightarrow \neg s)$
- q is replaced with $\neg\neg(q \leftrightarrow p)$

We obtain tautology

$$(q \vee \neg(r \rightarrow \neg s)) \rightarrow ((q \vee \neg(r \rightarrow \neg s)) \vee \neg\neg(q \leftrightarrow p))$$

Contradictions

Definition

A formula φ is a **contradiction** if $v \not\models \varphi$ holds for every truth valuation v (i.e., when φ is false in every valuation).

Example: *"It is Wednesday today, and it is not Wednesday today."*

$$p \wedge \neg p$$

- φ is a tautology iff $\neg\varphi$ is a contradiction
- φ is a contradiction iff $\neg\varphi$ is a tautology

Definition

A formula φ is **satisfiable** if there is at least one truth valuation v such that $v \models \varphi$.

- A formula is satisfiable iff it is not a contradiction.
- Every tautology is satisfiable but not every satisfiable formula is a tautology.

Example: A formula, which is satisfiable but not a tautology:

$$(p \vee q) \rightarrow p$$

- For example in valuation v_1 , where $v_1(p) = 1$ and $v_1(q) = 0$, the formula is true.
- In valuation v_2 , where $v_2(p) = 0$ and $v_2(q) = 1$, it is false.

- φ is a tautology iff $\neg\varphi$ is not satisfiable
 - φ is satisfiable iff $\neg\varphi$ is not a tautology
-
- **Satisfiable formulas:**
 - To show that a formula **is** satisfiable, it is sufficient to find a valuation, in which the formula is true.
 - To show that a formula **is not** satisfiable, it necessary to show that there is no valuation, in which the formula is true.

- **Tautologies:**

- To show that a formula **is not** a tautology, it is sufficient to find a valuation, in which the formula is false.
- To show that a formula **is** a tautology, it necessary to show that there is no valuation, in which the formula is false.

- **Contradictions:**

- To show that formula **is not** a contradiction, it is sufficient to find a valuation, in which the formula is true.
- To show that a formula **is** a contradiction, it necessary to show that there is no valuation, in which the formula is true.

Truth Valuations

For deciding whether a formula φ is or is not a tautology (resp. a contradiction, satisfiable), the **table method** can be used:

- To go through all possible truth valuations systematically.

It is usually not necessary to construct the whole table. It is sufficient to concentrate on “interesting” cases.

- We can draw a graph representing the given formula and try to assign values 0 and 1 to its nodes in such a way that either we find an example of a truth valuation we are looking for (e.g., some valuation where the formula is false), or we find out that such valuation does not exist.

For example, for deciding if a formula is a tautology:

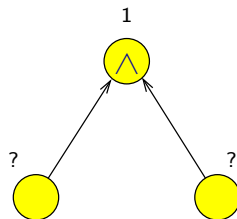
- We need to find out whether there exists some valuation where the formula is false.
- In this valuation, the node corresponding to the whole formula should have value 0.
- So we try to assign value 0 to this node.
- Then we try to assign values to other nodes in such a way that the assigned values are consistent with values assigned previously.
- If we succeed in labelling whole graph consistently, we have a valuation, in which the formula is false.

In this case, it is clear that the formula is not a tautology.

Truth Valuations

- If some values were already assigned to some nodes, this assignment can impose some constraints on values that can be assigned to other nodes.
- Examples where some previously assigned values enforce some particular value at some other node (resp. nodes):

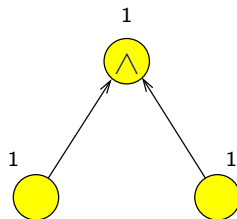
φ	ψ	$\varphi \wedge \psi$
0	0	0
0	1	0
1	0	0
1	1	1



Truth Valuations

- If some values were already assigned to some nodes, this assignment can impose some constraints on values that can be assigned to other nodes.
- Examples where some previously assigned values enforce some particular value at some other node (resp. nodes):

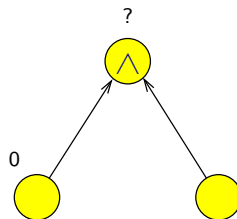
φ	ψ	$\varphi \wedge \psi$
0	0	0
0	1	0
1	0	0
1	1	1



Truth Valuations

- If some values were already assigned to some nodes, this assignment can impose some constraints on values that can be assigned to other nodes.
- Examples where some previously assigned values enforce some particular value at some other node (resp. nodes):

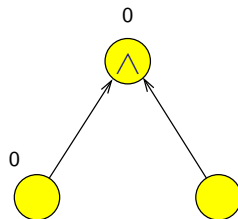
φ	ψ	$\varphi \wedge \psi$
0	0	0
0	1	0
1	0	0
1	1	1



Truth Valuations

- If some values were already assigned to some nodes, this assignment can impose some constraints on values that can be assigned to other nodes.
- Examples where some previously assigned values enforce some particular value at some other node (resp. nodes):

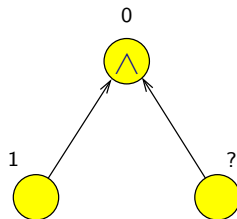
φ	ψ	$\varphi \wedge \psi$
0	0	0
0	1	0
1	0	0
1	1	1



Truth Valuations

- If some values were already assigned to some nodes, this assignment can impose some constraints on values that can be assigned to other nodes.
- Examples where some previously assigned values enforce some particular value at some other node (resp. nodes):

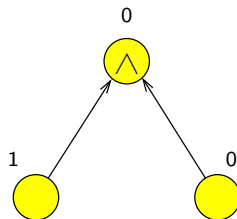
φ	ψ	$\varphi \wedge \psi$
0	0	0
0	1	0
1	0	0
1	1	1



Truth Valuations

- If some values were already assigned to some nodes, this assignment can impose some constraints on values that can be assigned to other nodes.
- Examples where some previously assigned values enforce some particular value at some other node (resp. nodes):

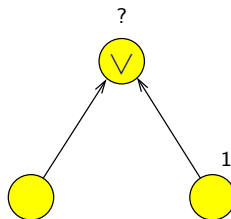
φ	ψ	$\varphi \wedge \psi$
0	0	0
0	1	0
1	0	0
1	1	1



Truth Valuations

- If some values were already assigned to some nodes, this assignment can impose some constraints on values that can be assigned to other nodes.
- Examples where some previously assigned values enforce some particular value at some other node (resp. nodes):

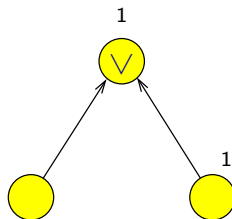
φ	ψ	$\varphi \vee \psi$
0	0	0
0	1	1
1	0	1
1	1	1



Truth Valuations

- If some values were already assigned to some nodes, this assignment can impose some constraints on values that can be assigned to other nodes.
- Examples where some previously assigned values enforce some particular value at some other node (resp. nodes):

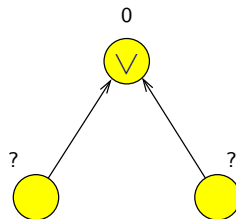
φ	ψ	$\varphi \vee \psi$
0	0	0
0	1	1
1	0	1
1	1	1



Truth Valuations

- If some values were already assigned to some nodes, this assignment can impose some constraints on values that can be assigned to other nodes.
- Examples where some previously assigned values enforce some particular value at some other node (resp. nodes):

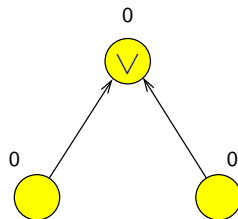
φ	ψ	$\varphi \vee \psi$
0	0	0
0	1	1
1	0	1
1	1	1



Truth Valuations

- If some values were already assigned to some nodes, this assignment can impose some constraints on values that can be assigned to other nodes.
- Examples where some previously assigned values enforce some particular value at some other node (resp. nodes):

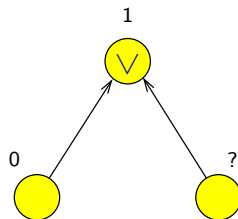
φ	ψ	$\varphi \vee \psi$
0	0	0
0	1	1
1	0	1
1	1	1



Truth Valuations

- If some values were already assigned to some nodes, this assignment can impose some constraints on values that can be assigned to other nodes.
- Examples where some previously assigned values enforce some particular value at some other node (resp. nodes):

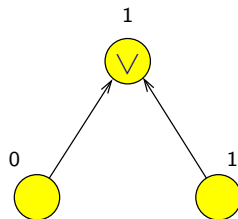
φ	ψ	$\varphi \vee \psi$
0	0	0
0	1	1
1	0	1
1	1	1



Truth Valuations

- If some values were already assigned to some nodes, this assignment can impose some constraints on values that can be assigned to other nodes.
- Examples where some previously assigned values enforce some particular value at some other node (resp. nodes):

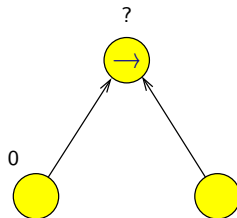
φ	ψ	$\varphi \vee \psi$
0	0	0
0	1	1
1	0	1
1	1	1



Truth Valuations

- If some values were already assigned to some nodes, this assignment can impose some constraints on values that can be assigned to other nodes.
- Examples where some previously assigned values enforce some particular value at some other node (resp. nodes):

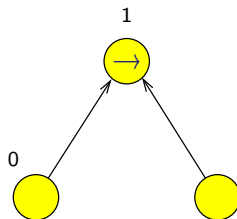
φ	ψ	$\varphi \rightarrow \psi$
0	0	1
0	1	1
1	0	0
1	1	1



Truth Valuations

- If some values were already assigned to some nodes, this assignment can impose some constraints on values that can be assigned to other nodes.
- Examples where some previously assigned values enforce some particular value at some other node (resp. nodes):

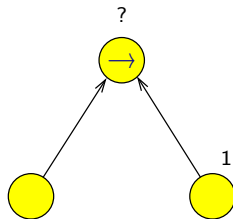
φ	ψ	$\varphi \rightarrow \psi$
0	0	1
0	1	1
1	0	0
1	1	1



Truth Valuations

- If some values were already assigned to some nodes, this assignment can impose some constraints on values that can be assigned to other nodes.
- Examples where some previously assigned values enforce some particular value at some other node (resp. nodes):

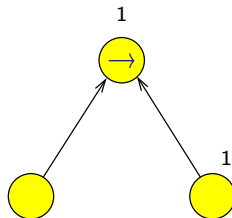
φ	ψ	$\varphi \rightarrow \psi$
0	0	1
0	1	1
1	0	0
1	1	1



Truth Valuations

- If some values were already assigned to some nodes, this assignment can impose some constraints on values that can be assigned to other nodes.
- Examples where some previously assigned values enforce some particular value at some other node (resp. nodes):

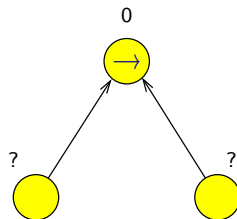
φ	ψ	$\varphi \rightarrow \psi$
0	0	1
0	1	1
1	0	0
1	1	1



Truth Valuations

- If some values were already assigned to some nodes, this assignment can impose some constraints on values that can be assigned to other nodes.
- Examples where some previously assigned values enforce some particular value at some other node (resp. nodes):

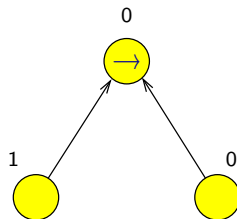
φ	ψ	$\varphi \rightarrow \psi$
0	0	1
0	1	1
1	0	0
1	1	1



Truth Valuations

- If some values were already assigned to some nodes, this assignment can impose some constraints on values that can be assigned to other nodes.
- Examples where some previously assigned values enforce some particular value at some other node (resp. nodes):

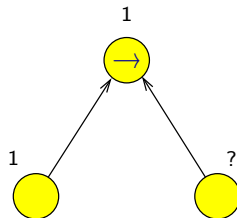
φ	ψ	$\varphi \rightarrow \psi$
0	0	1
0	1	1
1	0	0
1	1	1



Truth Valuations

- If some values were already assigned to some nodes, this assignment can impose some constraints on values that can be assigned to other nodes.
- Examples where some previously assigned values enforce some particular value at some other node (resp. nodes):

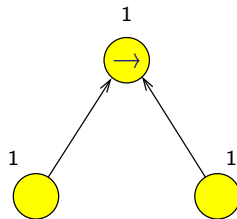
φ	ψ	$\varphi \rightarrow \psi$
0	0	1
0	1	1
1	0	0
1	1	1



Truth Valuations

- If some values were already assigned to some nodes, this assignment can impose some constraints on values that can be assigned to other nodes.
- Examples where some previously assigned values enforce some particular value at some other node (resp. nodes):

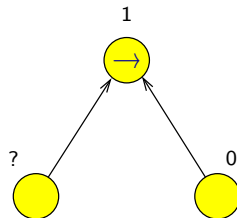
φ	ψ	$\varphi \rightarrow \psi$
0	0	1
0	1	1
1	0	0
1	1	1



Truth Valuations

- If some values were already assigned to some nodes, this assignment can impose some constraints on values that can be assigned to other nodes.
- Examples where some previously assigned values enforce some particular value at some other node (resp. nodes):

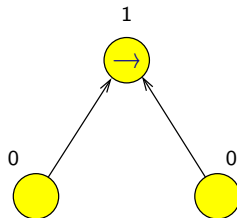
φ	ψ	$\varphi \rightarrow \psi$
0	0	1
0	1	1
1	0	0
1	1	1



Truth Valuations

- If some values were already assigned to some nodes, this assignment can impose some constraints on values that can be assigned to other nodes.
- Examples where some previously assigned values enforce some particular value at some other node (resp. nodes):

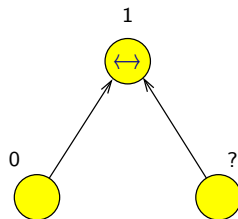
φ	ψ	$\varphi \rightarrow \psi$
0	0	1
0	1	1
1	0	0
1	1	1



Truth Valuations

- If some values were already assigned to some nodes, this assignment can impose some constraints on values that can be assigned to other nodes.
- Examples where some previously assigned values enforce some particular value at some other node (resp. nodes):

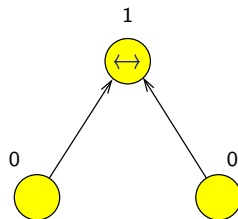
φ	ψ	$\varphi \leftrightarrow \psi$
0	0	1
0	1	0
1	0	0
1	1	1



Truth Valuations

- If some values were already assigned to some nodes, this assignment can impose some constraints on values that can be assigned to other nodes.
- Examples where some previously assigned values enforce some particular value at some other node (resp. nodes):

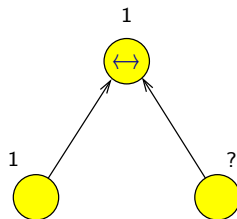
φ	ψ	$\varphi \leftrightarrow \psi$
0	0	1
0	1	0
1	0	0
1	1	1



Truth Valuations

- If some values were already assigned to some nodes, this assignment can impose some constraints on values that can be assigned to other nodes.
- Examples where some previously assigned values enforce some particular value at some other node (resp. nodes):

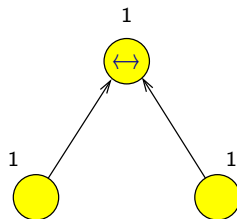
φ	ψ	$\varphi \leftrightarrow \psi$
0	0	1
0	1	0
1	0	0
1	1	1



Truth Valuations

- If some values were already assigned to some nodes, this assignment can impose some constraints on values that can be assigned to other nodes.
- Examples where some previously assigned values enforce some particular value at some other node (resp. nodes):

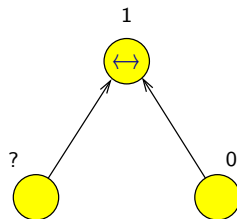
φ	ψ	$\varphi \leftrightarrow \psi$
0	0	1
0	1	0
1	0	0
1	1	1



Truth Valuations

- If some values were already assigned to some nodes, this assignment can impose some constraints on values that can be assigned to other nodes.
- Examples where some previously assigned values enforce some particular value at some other node (resp. nodes):

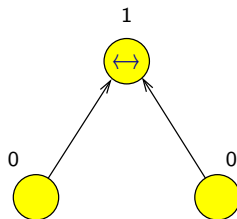
φ	ψ	$\varphi \leftrightarrow \psi$
0	0	1
0	1	0
1	0	0
1	1	1



Truth Valuations

- If some values were already assigned to some nodes, this assignment can impose some constraints on values that can be assigned to other nodes.
- Examples where some previously assigned values enforce some particular value at some other node (resp. nodes):

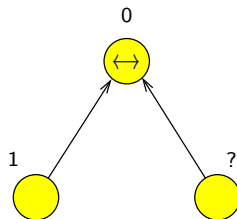
φ	ψ	$\varphi \leftrightarrow \psi$
0	0	1
0	1	0
1	0	0
1	1	1



Truth Valuations

- If some values were already assigned to some nodes, this assignment can impose some constraints on values that can be assigned to other nodes.
- Examples where some previously assigned values enforce some particular value at some other node (resp. nodes):

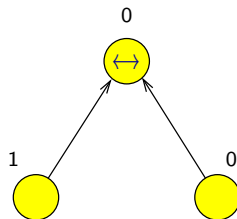
φ	ψ	$\varphi \leftrightarrow \psi$
0	0	1
0	1	0
1	0	0
1	1	1



Truth Valuations

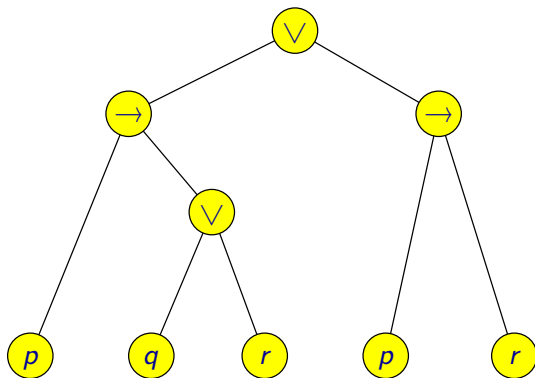
- If some values were already assigned to some nodes, this assignment can impose some constraints on values that can be assigned to other nodes.
- Examples where some previously assigned values enforce some particular value at some other node (resp. nodes):

φ	ψ	$\varphi \leftrightarrow \psi$
0	0	1
0	1	0
1	0	0
1	1	1



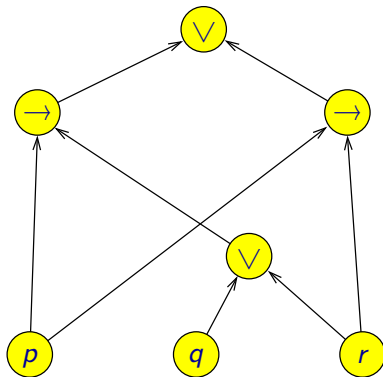
Is a given formula a tautology?

Example: $\varphi_1 := (p \rightarrow (q \vee r)) \vee (p \rightarrow r)$



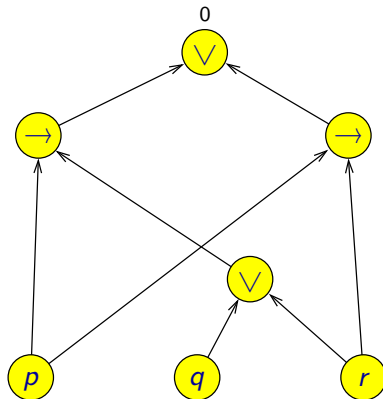
Is a given formula a tautology?

Example: $\varphi_1 := (p \rightarrow (q \vee r)) \vee (p \rightarrow r)$



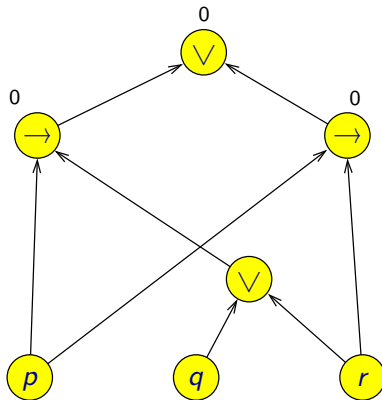
Is a given formula a tautology?

Example: $\varphi_1 := (p \rightarrow (q \vee r)) \vee (p \rightarrow r)$



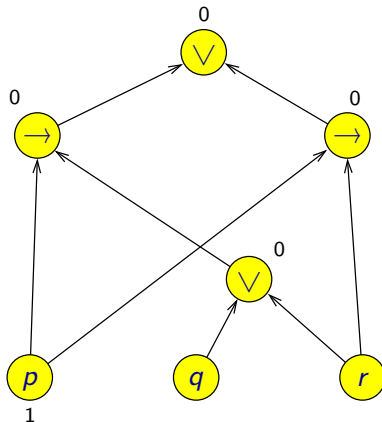
Is a given formula a tautology?

Example: $\varphi_1 := (p \rightarrow (q \vee r)) \vee (p \rightarrow r)$



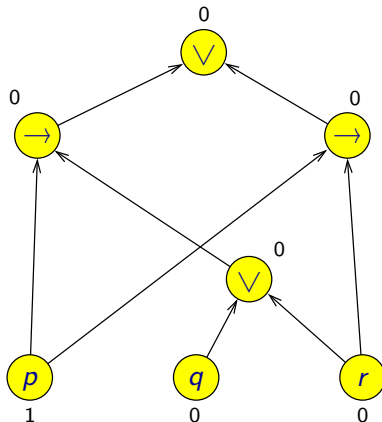
Is a given formula a tautology?

Example: $\varphi_1 := (p \rightarrow (q \vee r)) \vee (p \rightarrow r)$



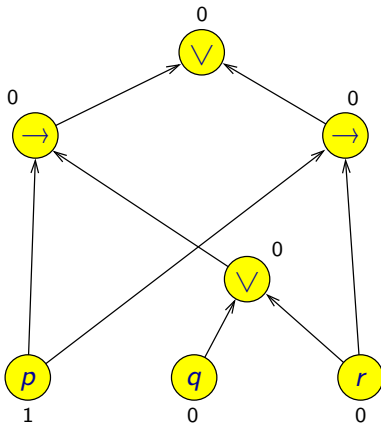
Is a given formula a tautology?

Example: $\varphi_1 := (p \rightarrow (q \vee r)) \vee (p \rightarrow r)$



Is a given formula a tautology?

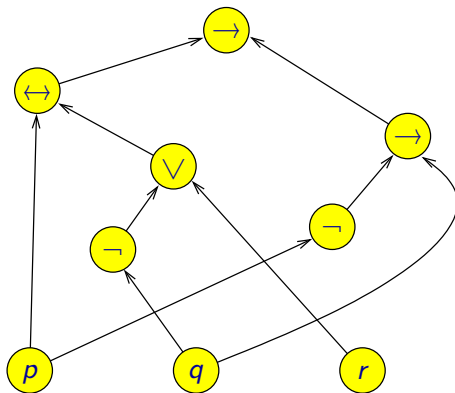
Example: $\varphi_1 := (p \rightarrow (q \vee r)) \vee (p \rightarrow r)$



Formula φ_1 is not a tautology — it is false in valuation v where $v(p) = 1$, $v(q) = 0$, $v(r) = 0$.

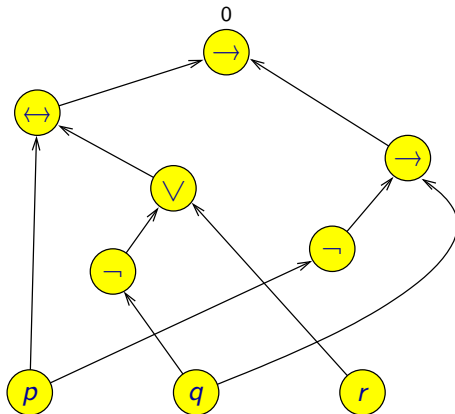
Is a given formula a tautology?

Example: $\varphi_2 := (p \leftrightarrow (\neg q \vee r)) \rightarrow (\neg p \rightarrow q)$



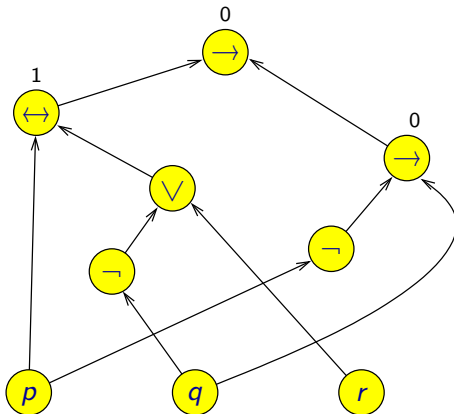
Is a given formula a tautology?

Example: $\varphi_2 := (p \leftrightarrow (\neg q \vee r)) \rightarrow (\neg p \rightarrow q)$



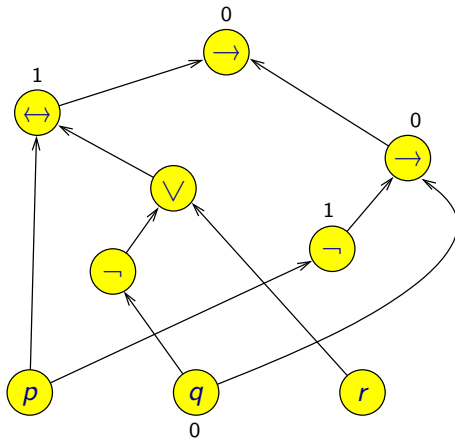
Is a given formula a tautology?

Example: $\varphi_2 := (p \leftrightarrow (\neg q \vee r)) \rightarrow (\neg p \rightarrow q)$



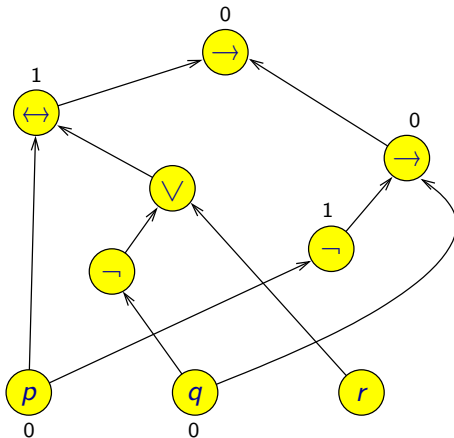
Is a given formula a tautology?

Example: $\varphi_2 := (p \leftrightarrow (\neg q \vee r)) \rightarrow (\neg p \rightarrow q)$



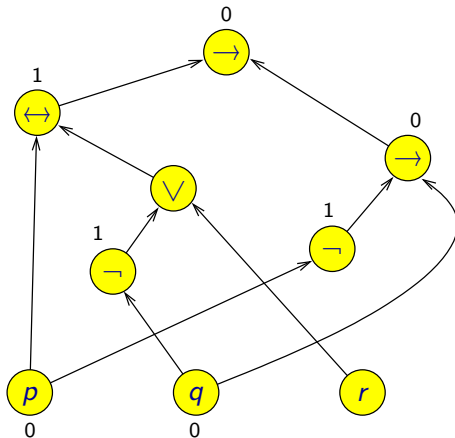
Is a given formula a tautology?

Example: $\varphi_2 := (p \leftrightarrow (\neg q \vee r)) \rightarrow (\neg p \rightarrow q)$



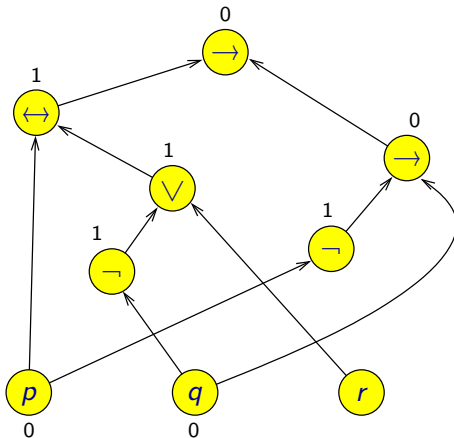
Is a given formula a tautology?

Example: $\varphi_2 := (p \leftrightarrow (\neg q \vee r)) \rightarrow (\neg p \rightarrow q)$



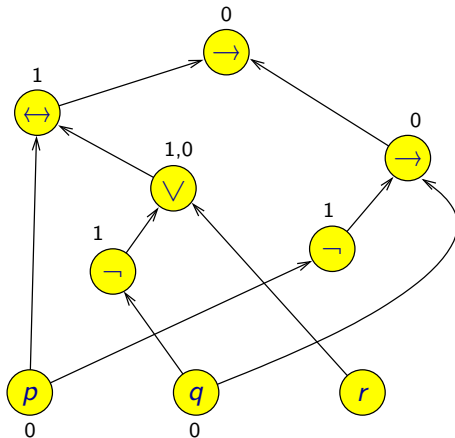
Is a given formula a tautology?

Example: $\varphi_2 := (p \leftrightarrow (\neg q \vee r)) \rightarrow (\neg p \rightarrow q)$



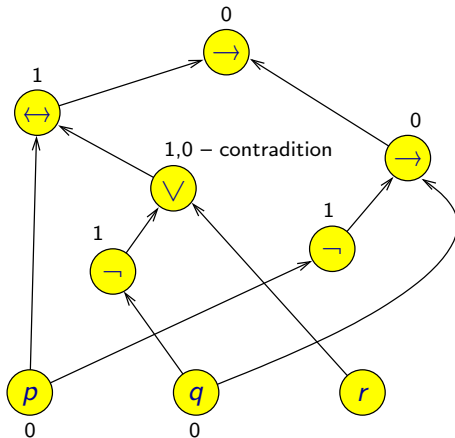
Is a given formula a tautology?

Example: $\varphi_2 := (p \leftrightarrow (\neg q \vee r)) \rightarrow (\neg p \rightarrow q)$



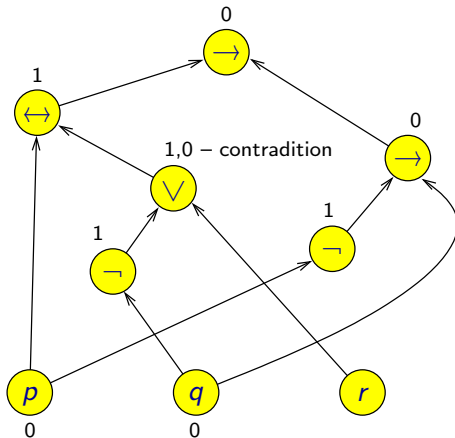
Is a given formula a tautology?

Example: $\varphi_2 := (p \leftrightarrow (\neg q \vee r)) \rightarrow (\neg p \rightarrow q)$



Is a given formula a tautology?

Example: $\varphi_2 := (p \leftrightarrow (\neg q \vee r)) \rightarrow (\neg p \rightarrow q)$



Formula φ_2 is a tautology.

Semantic contradiction — the case when we find out that in a valuation with the given property we are looking for (e.g., a valuation where a given formula is false), some formula should be true and false at the same time.

- There could not exist a valuation where some formula would be true and false at the same time.
- This way we can justify for example that a given formula is a tautology (and so always true), because by finding a semantic contradiction we show there can not exist a valuation where this formula is false.

Is a given formula a tautology?

The approach from the previous example can be described by the following sequence of arguments:

1. Let us assume that $(p \leftrightarrow (\neg q \vee r)) \rightarrow (\neg p \rightarrow q)$ is false. Then:
2. $p \leftrightarrow (\neg q \vee r)$ is true - it follows from 1.
3. $\neg p \rightarrow q$ is false - it follows from 1.
4. $\neg p$ is true - it follows from 3.
5. q is false - it follows from 3.
6. p is false - it follows from 4.
7. $\neg q$ is true - it follows from 5.
8. $\neg q \vee r$ is true - it follows from 7.
9. $\neg q \vee r$ is false - it follows from 2. and 6.
10. It is not possible that $(p \leftrightarrow (\neg q \vee r)) \rightarrow (\neg p \rightarrow q)$ is false because if it would be so, $\neg q \vee r$ would have to be true and false at the same time in this case (see 8. and 9.).

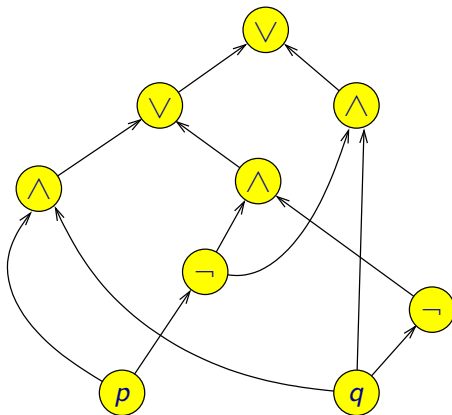
Remark: Note that in this justification the graph representing the given formula is not mentioned at all. We talk there only about truth and falsity of subformulas of this formula.

Is a given formula a tautology?

- It is not always the case that values that could be assigned to some nodes are uniquely determined by values previously assigned to some other nodes.
- When we are in a situation where it is not possible to assign a unique value to a node, it is necessary to try several possibilities.
- We choose some node and a value assigned to it. Then possibly some values that must be assigned to some other nodes are determined.
- If we do not succeed in finding a valuation we are looking for, we must backtrack, assign a different value to the given node, and try this new possibility.

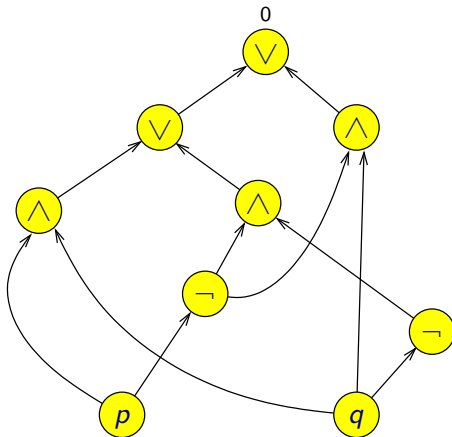
Is a given formula a tautology?

Example: $\varphi_3 := ((p \wedge q) \vee (\neg p \wedge \neg q)) \vee (\neg p \wedge q)$



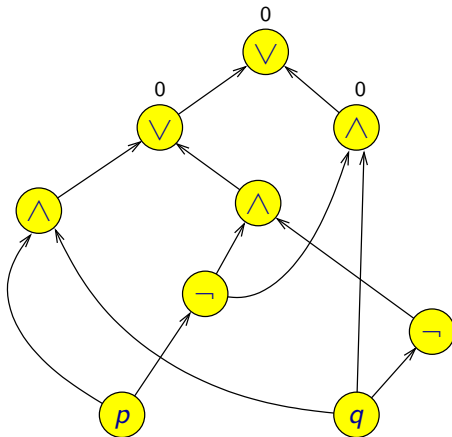
Is a given formula a tautology?

Example: $\varphi_3 := ((p \wedge q) \vee (\neg p \wedge \neg q)) \vee (\neg p \wedge q)$



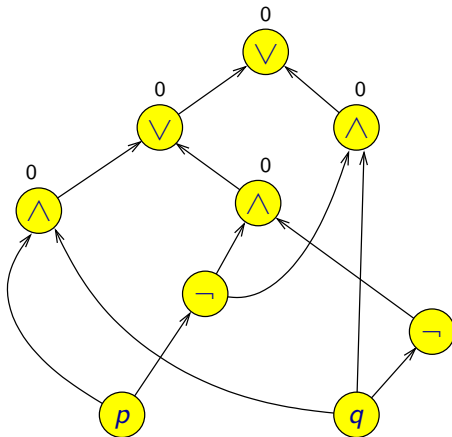
Is a given formula a tautology?

Example: $\varphi_3 := ((p \wedge q) \vee (\neg p \wedge \neg q)) \vee (\neg p \wedge q)$



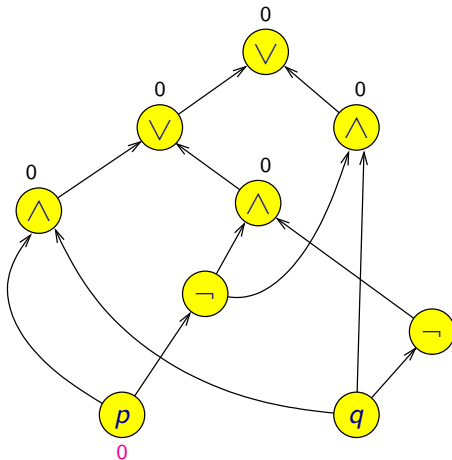
Is a given formula a tautology?

Example: $\varphi_3 := ((p \wedge q) \vee (\neg p \wedge \neg q)) \vee (\neg p \wedge q)$



Is a given formula a tautology?

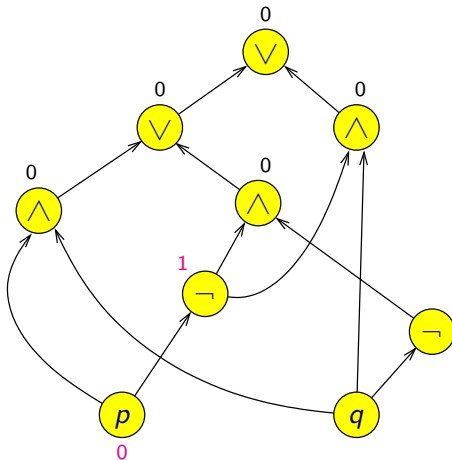
Example: $\varphi_3 := ((p \wedge q) \vee (\neg p \wedge \neg q)) \vee (\neg p \wedge q)$



We try to assign value zero to node p .

Is a given formula a tautology?

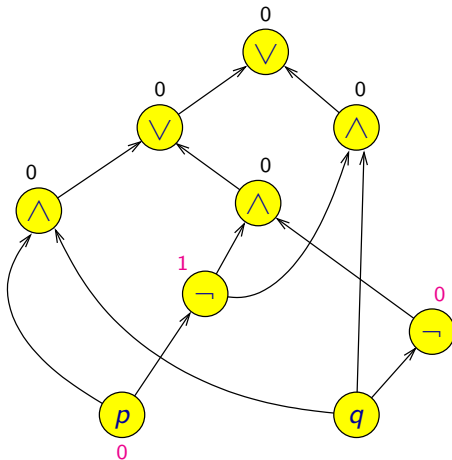
Example: $\varphi_3 := ((p \wedge q) \vee (\neg p \wedge \neg q)) \vee (\neg p \wedge q)$



We try to assign value zero to node p .

Is a given formula a tautology?

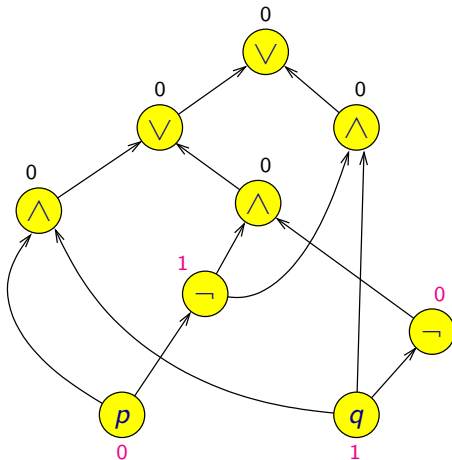
Example: $\varphi_3 := ((p \wedge q) \vee (\neg p \wedge \neg q)) \vee (\neg p \wedge q)$



We try to assign value zero to node p .

Is a given formula a tautology?

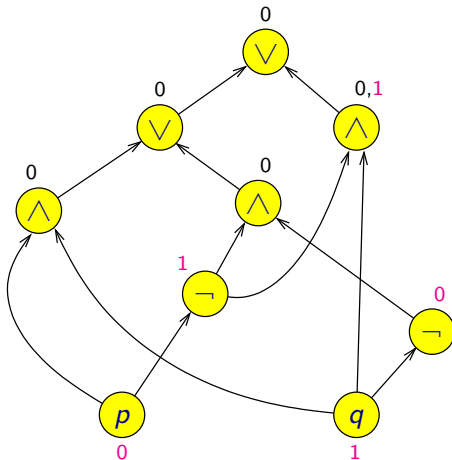
Example: $\varphi_3 := ((p \wedge q) \vee (\neg p \wedge \neg q)) \vee (\neg p \wedge q)$



We try to assign value zero to node p .

Is a given formula a tautology?

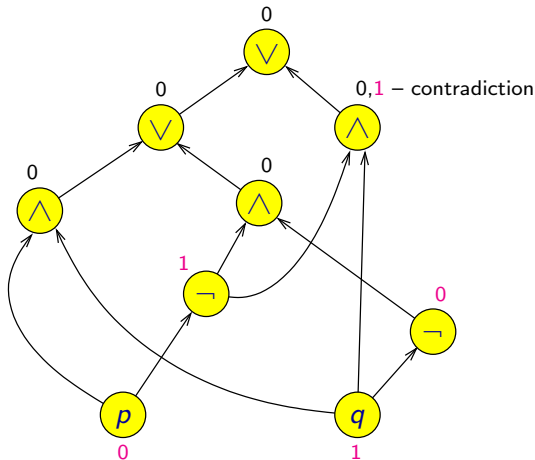
Example: $\varphi_3 := ((p \wedge q) \vee (\neg p \wedge \neg q)) \vee (\neg p \wedge q)$



We try to assign value zero to node p .

Is a given formula a tautology?

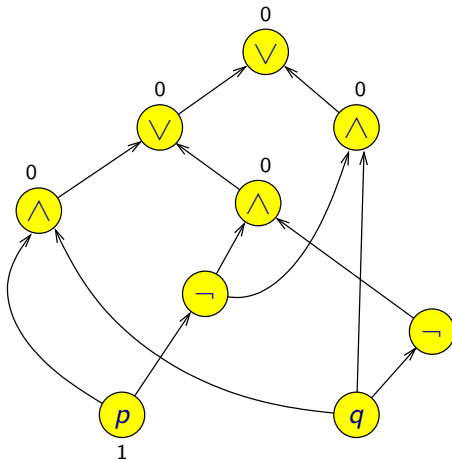
Example: $\varphi_3 := ((p \wedge q) \vee (\neg p \wedge \neg q)) \vee (\neg p \wedge q)$



We try to assign value zero to node p .

Is a given formula a tautology?

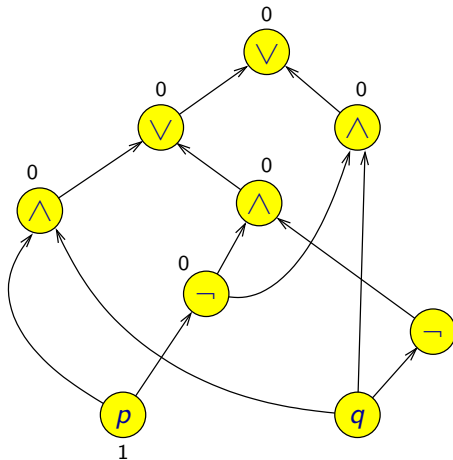
Example: $\varphi_3 := ((p \wedge q) \vee (\neg p \wedge \neg q)) \vee (\neg p \wedge q)$



So node p must have value 1.

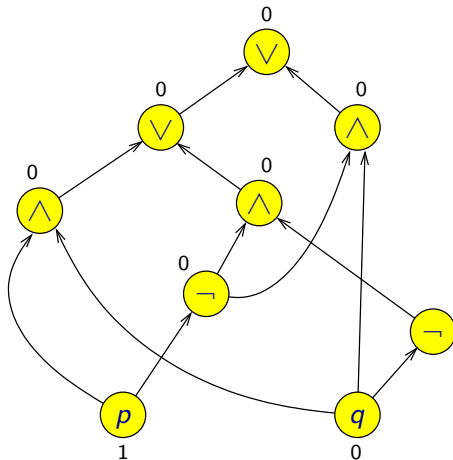
Is a given formula a tautology?

Example: $\varphi_3 := ((p \wedge q) \vee (\neg p \wedge \neg q)) \vee (\neg p \wedge q)$



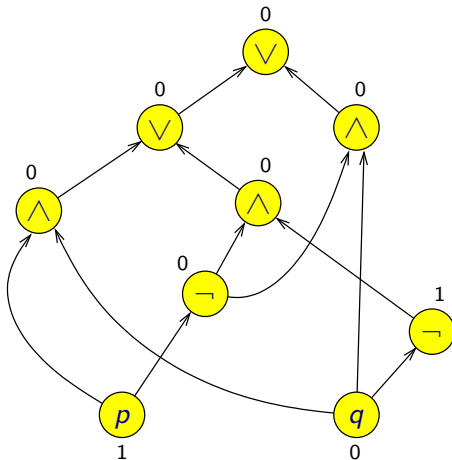
Is a given formula a tautology?

Example: $\varphi_3 := ((p \wedge q) \vee (\neg p \wedge \neg q)) \vee (\neg p \wedge q)$



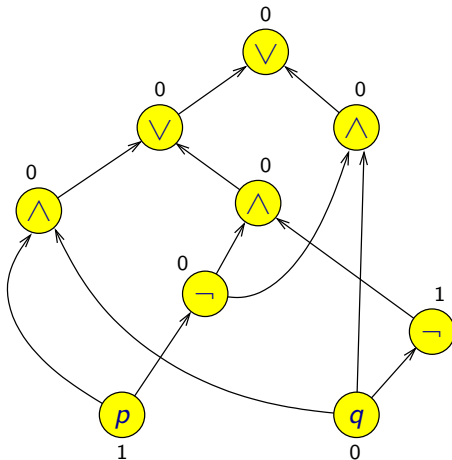
Is a given formula a tautology?

Example: $\varphi_3 := ((p \wedge q) \vee (\neg p \wedge \neg q)) \vee (\neg p \wedge q)$



Is a given formula a tautology?

Example: $\varphi_3 := ((p \wedge q) \vee (\neg p \wedge \neg q)) \vee (\neg p \wedge q)$



Formula φ_3 is not a tautology — it is false in valuation v where $v(p) = 1$, $v(q) = 0$.

Equivalence of Formulas

Definition

Formulas φ and ψ are **logically equivalent** if for each truth valuation v it holds that φ and ψ have the same truth value in valuation v , i.e.,

$$v \models \varphi \quad \text{iff} \quad v \models \psi.$$

The fact that formulas φ and ψ are logically equivalent is denoted

$$\varphi \Leftrightarrow \psi.$$

Formulas φ and ψ are logically equivalent iff $\varphi \leftrightarrow \psi$ is a tautology.

Equivalence of Formulas

Example: $\neg(p \rightarrow q) \Leftrightarrow p \wedge \neg q$

p	q	$p \rightarrow q$	$\neg(p \rightarrow q)$	$\neg q$	$p \wedge \neg q$
0	0	1	0	1	0
0	1	1	0	0	0
1	0	0	1	1	1
1	1	1	0	0	0

Equivalence of Formulas

To show that formulas φ and ψ are **not** equivalent, it is sufficient to find a valuation v such that:

- $v \models \varphi$ and $v \not\models \psi$, or
- $v \not\models \varphi$ and $v \models \psi$.

Example: $p \vee (q \wedge r)$ is not equivalent to $(p \vee q) \wedge r$

Valuation v , where:

- $v(p) = 1$
- $v(q) = 1$
- $v(r) = 0$

In this valuation, $p \vee (q \wedge r)$ holds but $(p \vee q) \wedge r$ does not hold.

Some Important Equivalences

- Equivalences for negation:

$$\neg\neg p \Leftrightarrow p$$

double negation

- Equivalences for conjunction:

$$(p \wedge q) \wedge r \Leftrightarrow p \wedge (q \wedge r)$$

associativity

$$p \wedge q \Leftrightarrow q \wedge p$$

commutativity

$$p \wedge p \Leftrightarrow p$$

idempotence

- Equivalences for disjunction:

$$(p \vee q) \vee r \Leftrightarrow p \vee (q \vee r)$$

associativity

$$p \vee q \Leftrightarrow q \vee p$$

commutativity

$$p \vee p \Leftrightarrow p$$

idempotence

Some Important Equivalences

- Distributivity of $\wedge$ and $\vee$:

$$p \wedge (q \vee r) \Leftrightarrow (p \wedge q) \vee (p \wedge r)$$

$$p \vee (q \wedge r) \Leftrightarrow (p \vee q) \wedge (p \vee r)$$

- De Morgan's laws:

$$\neg(p \wedge q) \Leftrightarrow \neg p \vee \neg q$$

$$\neg(p \vee q) \Leftrightarrow \neg p \wedge \neg q$$

- Equivalences for implication:

$$p \rightarrow q \Leftrightarrow \neg p \vee q$$

$$\neg(p \rightarrow q) \Leftrightarrow p \wedge \neg q$$

Some Important Equivalences

– Equivalences for $\leftrightarrow$:

$$(p \leftrightarrow q) \leftrightarrow r \Leftrightarrow p \leftrightarrow (q \leftrightarrow r)$$

associativity

$$p \leftrightarrow q \Leftrightarrow q \leftrightarrow p$$

commutativity

$$p \leftrightarrow q \Leftrightarrow (p \rightarrow q) \wedge (q \rightarrow p)$$

$$p \leftrightarrow q \Leftrightarrow (p \vee \neg q) \wedge (\neg p \vee q)$$

$$p \leftrightarrow q \Leftrightarrow (p \wedge q) \vee (\neg p \wedge \neg q)$$

Equivalence of Formulas

Let us assume that formulas φ and ψ are logically equivalent, i.e.,

$$\varphi \Leftrightarrow \psi.$$

If we replace atomic propositions in φ and ψ by arbitrary formulas, we obtain again a pair of equivalent formulas.

Example: $\neg(p \vee q) \Leftrightarrow \neg p \wedge \neg q$

Therefore, for arbitrary formulas χ_1 and χ_2 is

$$\neg(\chi_1 \vee \chi_2) \Leftrightarrow \neg\chi_1 \wedge \neg\chi_2$$

Equivalence of Formulas

$$\neg(p \vee q) \Leftrightarrow \neg p \wedge \neg q$$

Replacement of atomic propositions:

- p replaced by $q \vee \neg(r \rightarrow \neg s)$
- q replaced by $\neg(q \leftrightarrow p)$

We obtain

$$\neg((q \vee \neg(r \rightarrow \neg s)) \vee \neg(q \leftrightarrow p)) \Leftrightarrow \neg(q \vee \neg(r \rightarrow \neg s)) \wedge \neg\neg(q \leftrightarrow p)$$

Equivalence of Formulas

Let us assume that φ is a formula and ψ its subformula.

If we replace some occurrence of subformula ψ in formula φ with a formula ψ' such that $\psi \Leftrightarrow \psi'$, we obtain a formula φ' such that

$$\varphi \Leftrightarrow \varphi'.$$

Example: In formula

$$\neg((p \rightarrow q) \vee (\neg(p \rightarrow q) \rightarrow r))$$

we replace the second occurrence of subformula $p \rightarrow q$ with an equivalent formula $\neg p \vee q$.

We obtain

$$\neg((p \rightarrow q) \vee (\neg(\neg p \vee q) \rightarrow r))$$

Equivalent Transformations

For arbitrary formulas φ , ψ , and χ , it holds:

- $\varphi \Leftrightarrow \varphi$.
- If $\varphi \Leftrightarrow \psi$, then $\psi \Leftrightarrow \varphi$.
- If $\varphi \Leftrightarrow \psi$ and $\psi \Leftrightarrow \chi$, then $\varphi \Leftrightarrow \chi$.

When we try to prove equivalence of formulas, we can proceed by smaller steps:

For example, if it holds that $\varphi_1 \Leftrightarrow \varphi_2$, $\varphi_2 \Leftrightarrow \varphi_3$, $\varphi_3 \Leftrightarrow \varphi_4$, and $\varphi_4 \Leftrightarrow \varphi_5$, we can conclude that

$$\varphi_1 \Leftrightarrow \varphi_5.$$

This can be written as

$$\varphi_1 \Leftrightarrow \varphi_2 \Leftrightarrow \varphi_3 \Leftrightarrow \varphi_4 \Leftrightarrow \varphi_5$$

Example: The proof that

$$(p \wedge q) \rightarrow r \Leftrightarrow p \rightarrow (q \rightarrow r)$$

$$\begin{aligned}(p \wedge q) \rightarrow r &\Leftrightarrow \neg(p \wedge q) \vee r \\&\Leftrightarrow (\neg p \vee \neg q) \vee r \\&\Leftrightarrow \neg p \vee (\neg q \vee r) \\&\Leftrightarrow \neg p \vee (q \rightarrow r) \\&\Leftrightarrow p \rightarrow (q \rightarrow r)\end{aligned}$$

Equivalent Transformations

Every formula can be transformed to an equivalent formula that uses only “ $\neg$ ”, “ $\wedge$ ”, and “ $\vee$ ” as logical connectives.

- The connective “ $\leftrightarrow$ ” can be replaced by other connectives using the following equivalences:
 - $p \leftrightarrow q \Leftrightarrow (p \rightarrow q) \wedge (q \rightarrow p)$
 - $p \leftrightarrow q \Leftrightarrow (p \vee \neg q) \wedge (\neg p \vee q)$
 - $p \leftrightarrow q \Leftrightarrow (p \wedge q) \vee (\neg p \wedge \neg q)$
- The connective “ $\rightarrow$ ” can be replaced by “ $\neg$ ” and “ $\vee$ ” using the following equivalence:
 - $p \rightarrow q \Leftrightarrow \neg p \vee q$

Example:

$$\begin{aligned}(\neg q \rightarrow r) \wedge \neg(p \leftrightarrow r) &\Leftrightarrow (\neg \neg q \vee r) \wedge \neg(p \leftrightarrow r) \\ &\Leftrightarrow (\neg \neg q \vee r) \wedge \neg((p \wedge r) \vee (\neg p \wedge \neg r))\end{aligned}$$

Equivalent Transformations

Every formula can be transformed to an equivalent formula, which contains only logical connectives “ $\neg$ ”, “ $\wedge$ ” and “ $\vee$ ”, and where negations are applied only to atomic propositions.

- We can assume that formula contains only “ $\neg$ ”, “ $\wedge$ ” and “ $\vee$ ”.
- Negations can be “pushed” to atomic propositions using the following equivalences:
 - $\neg\neg p \Leftrightarrow p$
 - $\neg(p \wedge q) \Leftrightarrow \neg p \vee \neg q$
 - $\neg(p \vee q) \Leftrightarrow \neg p \wedge \neg q$

Example:

$$\begin{aligned} & (\neg\neg q \vee r) \wedge \neg((p \wedge r) \vee (\neg p \wedge \neg r)) \\ \Leftrightarrow & (q \vee r) \wedge \neg((p \wedge r) \vee (\neg p \wedge \neg r)) \\ \Leftrightarrow & (q \vee r) \wedge (\neg(p \wedge r) \wedge \neg(\neg p \wedge \neg r)) \\ \Leftrightarrow & (q \vee r) \wedge ((\neg p \vee \neg r) \wedge \neg(\neg p \wedge \neg r)) \\ \Leftrightarrow & (q \vee r) \wedge ((\neg p \vee \neg r) \wedge (\neg\neg p \vee \neg\neg r)) \\ \Leftrightarrow & (q \vee r) \wedge ((\neg p \vee \neg r) \wedge (p \vee \neg\neg r)) \\ \Leftrightarrow & (q \vee r) \wedge ((\neg p \vee \neg r) \wedge (p \vee r)) \end{aligned}$$

Logical Constants

For some purposes it can be useful to introduce the following special formulas:

- $\top$ — a formula, which is always true
- $\perp$ — a formula, which is always false

For every truth valuation v it holds:

- $v \models \top$ ($\top$ has always truth value 1)
- $v \not\models \perp$ ($\perp$ has always truth value 0)

Symbols $\top$ and $\perp$ can be viewed as abbreviations:

- $\top$ stands for an arbitrary tautology (e.g., $p \rightarrow p$)
- $\perp$ stands for an arbitrary contradiction (e.g., $p \wedge \neg p$)

Alternatively, we could extend the definition of syntax and semantics of propositional logic.

Symbols $\top$ and $\perp$ can be viewed as logical connectives with arity 0.

Examples of equivalences that hold for $\top$ and $\perp$ (and for arbitrary p):

$$\top \Leftrightarrow p \vee \neg p$$

$$\neg \top \Leftrightarrow \perp$$

$$p \wedge \top \Leftrightarrow p$$

$$p \vee \top \Leftrightarrow \top$$

$$\perp \Leftrightarrow p \wedge \neg p$$

$$\neg \perp \Leftrightarrow \top$$

$$p \vee \perp \Leftrightarrow p$$

$$p \wedge \perp \Leftrightarrow \perp$$

Equivalence of Formulas

It is **not** necessary for equivalent formulas to contain the same atomic propositions.

Example: $(q \rightarrow \neg\neg q) \wedge \neg p \Leftrightarrow p \rightarrow (r \wedge \neg r)$

$$\begin{aligned}(q \rightarrow \neg\neg q) \wedge \neg p &\Leftrightarrow (q \rightarrow q) \wedge \neg p \\ &\Leftrightarrow \top \wedge \neg p \\ &\Leftrightarrow \neg p \\ &\Leftrightarrow \neg p \vee \perp \\ &\Leftrightarrow p \rightarrow \perp \\ &\Leftrightarrow p \rightarrow (r \wedge \neg r)\end{aligned}$$

For example, also all tautologies are logically equivalent.

Conjunctions and Disjunctions of Several Formulas

Due to associativity of conjunction, it holds for example:

$$p \wedge ((q \wedge r) \wedge (s \wedge t)) \Leftrightarrow (p \wedge q) \wedge ((r \wedge s) \wedge t)$$

Both these formulas are also equivalent to formulas

- $p \wedge (q \wedge (r \wedge (s \wedge t)))$
- $((p \wedge q) \wedge r) \wedge s \wedge t$

All these formulas are true iff all propositions p , q , r , s , and t are true.

Convention: Due to associativity of conjunction, the parentheses can be omitted and we can write

$$p \wedge q \wedge r \wedge s \wedge t$$

Conjunctions and Disjunctions of Several Formulas

Because conjunction is not only associative but also commutative, the order of members of such more complicated conjunction is not important. For example:

$$r \wedge t \wedge q \wedge s \wedge p \Leftrightarrow p \wedge q \wedge r \wedge s \wedge t$$

Due to idempotence, also the number of occurrences of each member is not important.

For example:

$$p \wedge q \wedge p \Leftrightarrow q \wedge p \wedge q \wedge q$$

Conjunctions and Disjunctions of Several Formulas

The same holds also for disjunction, e.g.:

$$(p \vee q) \vee (r \vee q) \Leftrightarrow q \vee (p \vee (r \vee r))$$

Convention: Instead of $(p \vee q) \vee (r \vee (s \vee t))$ we can write

$$p \vee q \vee r \vee s \vee t$$

All this holds not only for atomic propositions but also for arbitrary formulas, e.g.:

- Instead of $(\varphi_1 \wedge \varphi_2) \wedge (\varphi_3 \wedge (\varphi_4 \wedge \varphi_5))$ we can write

$$\varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_4 \wedge \varphi_5$$

Conjunctions and Disjunctions of Several Formulas

Conjunction of n formulas $\varphi_1, \varphi_2, \dots, \varphi_n$, where $n \geq 0$, is the formula

$$\varphi_1 \wedge \varphi_2 \wedge \dots \wedge \varphi_n$$

In particular:

- For $n = 0$, the conjunction is the formula $\top$.
- For $n = 1$, the conjunction is the formula φ_1 .

Disjunction of n formulas $\varphi_1, \varphi_2, \dots, \varphi_n$, where $n \geq 0$, is the formula

$$\varphi_1 \vee \varphi_2 \vee \dots \vee \varphi_n$$

In particular:

- For $n = 0$, the disjunction is the formula $\perp$.
- For $n = 1$, the disjunction is the formula φ_1 .

Conjunctions and Disjunctions of Several Formulas

Conjunction $\varphi_1 \wedge \varphi_2 \wedge \dots \wedge \varphi_n$:

- The whole formula is true iff all formulas $\varphi_1, \varphi_2, \dots, \varphi_n$ are true.
- If some formula φ_i is equivalent to $\perp$, then the whole formula is equivalent to $\perp$.
- If some formula φ_i is equivalent to a negation of some formula φ_j (i.e., $\varphi_i \Leftrightarrow \neg\varphi_j$), then the whole formula is equivalent to $\perp$.
- If some formula φ_i is equivalent to $\top$, then it is possible to omit the formula φ_i from the whole formula.

Conjunctions and Disjunctions of Several Formulas

Disjunction $\varphi_1 \vee \varphi_2 \vee \dots \vee \varphi_n$:

- The whole formula is true iff at least one of formulas $\varphi_1, \varphi_2, \dots, \varphi_n$ is true.
- If some formula φ_i is equivalent to $\top$, then the whole formula is equivalent to $\top$.
- If some formula φ_i is equivalent to a negation of some formula φ_j (i.e., $\varphi_i \Leftrightarrow \neg\varphi_j$), then the whole formula is equivalent to $\top$.
- If some formula φ_i is equivalent to $\perp$, then it is possible to omit the formula φ_i from the whole formula.

Normal Forms of Formulas

- **Literal** — an atomic proposition or its negation, e.g.,

$$p \qquad \neg q \qquad \neg r$$

- An **elementary conjunction** — a conjunction of one or more literals, e.g.,

$$(p \wedge \neg q) \qquad (r) \qquad (q \wedge \neg r \wedge p)$$

- An **elementary disjunction (clause)** — a disjunction of one or more literals, e.g.,

$$(p \vee \neg q) \qquad (r) \qquad (q \vee \neg r \vee p)$$

Example:

- Elementary conjunction

$$(p \wedge \neg q \wedge r \wedge \neg s \wedge \neg t)$$

is **true** in exactly those truth valuations v where

$$v(p) = 1 \quad v(q) = 0 \quad v(r) = 1 \quad v(s) = 0 \quad v(t) = 0$$

- Elementary disjunction

$$(p \vee \neg q \vee r \vee \neg s \vee \neg t)$$

is **false** in exactly those truth valuations v where

$$v(p) = 0 \quad v(q) = 1 \quad v(r) = 0 \quad v(s) = 1 \quad v(t) = 1$$

- **Disjunctive normal form (DNF)** — a disjunction of zero or more elementary conjunctions, e.g.,

$$(p \wedge \neg q) \vee (\neg r) \vee (\neg r \wedge \neg p \wedge \neg q)$$

- **Conjunctive normal form (CNF)** — a conjunction of zero or more elementary disjunctions (clauses), e.g.,

$$(p \vee \neg q) \wedge (\neg r) \wedge (\neg r \vee \neg p \vee \neg q)$$

Remark: So formula $\perp$ is a special case of a formula in DNF, and formula $\top$ is a special case of a formula in CNF.

Normal Forms of Formulas

A formula in CNF is a **tautology** iff for each elementary disjunction there is some atomic proposition p such that literals p and $\neg p$ occur in the elementary disjunction.

Example: $(p \vee q \vee \neg r \vee \neg q) \wedge (\neg p \vee \neg s \vee s) \wedge (t \vee \neg r \vee s \vee \neg t \vee q)$

A formula in DNF is a **contradiction** iff for each elementary conjunction there is some atomic proposition p such that literals p and $\neg p$ occur in the elementary conjunction.

Example: $(p \wedge q \wedge \neg r \wedge \neg q) \vee (\neg p \wedge \neg s \wedge s) \vee (t \wedge \neg r \wedge s \wedge \neg t \vee q)$

Transformation of a formula to DNF and CNF:

- We can assume that the formula contains only atomic propositions, connectives “ $\neg$ ” applied to atomic propositions, and connectives “ $\wedge$ ” and “ $\vee$ ”.
- The required form of the formula can be obtained by use of the following equivalences:
 - $p \wedge (q \vee r) \Leftrightarrow (p \wedge q) \vee (p \wedge r)$ — for transformation to DNF
 - $p \vee (q \wedge r) \Leftrightarrow (p \vee q) \wedge (p \vee r)$ — for transformation to CNF

Normal Forms of Formulas

Example: Transformation of formula $q \wedge ((\neg p \vee \neg r) \wedge (p \vee r))$ to DNF:

$$\begin{aligned} & q \wedge ((\neg p \vee \neg r) \wedge (p \vee r)) \\ \Leftrightarrow & (q \wedge (\neg p \vee \neg r)) \wedge (p \vee r) \\ \Leftrightarrow & ((q \wedge \neg p) \vee (q \wedge \neg r)) \wedge (p \vee r) \\ \Leftrightarrow & (((q \wedge \neg p) \vee (q \wedge \neg r)) \wedge p) \vee (((q \wedge \neg p) \vee (q \wedge \neg r)) \wedge r) \\ \Leftrightarrow & (((q \wedge \neg p) \wedge p) \vee ((q \wedge \neg r) \wedge p)) \vee (((q \wedge \neg p) \vee (q \wedge \neg r)) \wedge r) \\ \Leftrightarrow & (q \wedge \neg p \wedge p) \vee (q \wedge \neg r \wedge p) \vee (((q \wedge \neg p) \vee (q \wedge \neg r)) \wedge r) \\ \Leftrightarrow & (q \wedge \perp) \vee (q \wedge \neg r \wedge p) \vee (((q \wedge \neg p) \vee (q \wedge \neg r)) \wedge r) \\ \Leftrightarrow & \perp \vee (q \wedge \neg r \wedge p) \vee (((q \wedge \neg p) \vee (q \wedge \neg r)) \wedge r) \\ \Leftrightarrow & (q \wedge \neg r \wedge p) \vee (((q \wedge \neg p) \vee (q \wedge \neg r)) \wedge r) \\ \Leftrightarrow & (p \wedge q \wedge \neg r) \vee (((q \wedge \neg p) \vee (q \wedge \neg r)) \wedge r) \\ \Leftrightarrow & \dots \end{aligned}$$

...

$$\Leftrightarrow (p \wedge q \wedge \neg r) \vee (((q \wedge \neg p) \vee (q \wedge \neg r)) \wedge r)$$

$$\Leftrightarrow (p \wedge q \wedge \neg r) \vee (((q \wedge \neg p) \wedge r) \vee ((q \wedge \neg r) \wedge r))$$

$$\Leftrightarrow (p \wedge q \wedge \neg r) \vee (q \wedge \neg p \wedge r) \vee (q \wedge \neg r \wedge r)$$

$$\Leftrightarrow (p \wedge q \wedge \neg r) \vee (\neg p \wedge q \wedge r) \vee (q \wedge \neg r \wedge r)$$

$$\Leftrightarrow (p \wedge q \wedge \neg r) \vee (\neg p \wedge q \wedge r) \vee (q \wedge \perp)$$

$$\Leftrightarrow (p \wedge q \wedge \neg r) \vee (\neg p \wedge q \wedge r) \vee \perp$$

$$\Leftrightarrow (p \wedge q \wedge \neg r) \vee (\neg p \wedge q \wedge r)$$

Normal Forms of Formulas

It is easy to construct a formula in CNF or DNF for a given truth table:

p	q	r	φ
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

DNF:

$$(\neg p \wedge \neg q \wedge r) \vee (\neg p \wedge q \wedge \neg r) \vee (p \wedge \neg q \wedge r)$$

CNF:

$$(p \vee q \vee r) \wedge (p \vee \neg q \vee \neg r) \wedge (\neg p \vee q \vee r) \wedge (\neg p \vee \neg q \vee r) \wedge (\neg p \vee \neg q \vee \neg r)$$

Normal Forms of Formulas

When we consider a fixed **finite** set of atomic propositions At :

- **Complete disjunctive normal form (CDNF)** — a formula in DNF, where every elementary conjunction contains every atomic proposition from At exactly once.

Example: $(p \wedge \neg q \wedge \neg r) \vee (p \wedge q \wedge \neg r) \vee (\neg p \wedge q \wedge \neg r)$

- **Complete conjunctive normal form (CCNF)** — a formula in CNF, where every clause contains every atomic proposition from At exactly once.

Example: $(p \vee \neg q \vee \neg r) \wedge (p \vee q \vee \neg r) \wedge (\neg p \vee q \vee \neg r)$

Remark: In the examples is $At = \{p, q, r\}$.

Minimal Sets of Logical Connectives

We can see from the previous discussion that connectives “ $\neg$ ”, “ $\wedge$ ”, and “ $\vee$ ” suffice for constructing a formula for every truth table.

In fact, some smaller sets of logical connectives are sufficient for this purpose:

- “ $\neg$ ”, “ $\wedge$ ”:

$\varphi \vee \psi$ can be expressed as $\neg(\neg\varphi \wedge \neg\psi)$

- “ $\neg$ ”, “ $\vee$ ”:

$\varphi \wedge \psi$ can be expressed as $\neg(\neg\varphi \vee \neg\psi)$

- “ $\neg$ ”, “ $\rightarrow$ ”:

$\varphi \vee \psi$ can be expressed as $\neg\varphi \rightarrow \psi$

$\varphi \wedge \psi$ can be expressed as $\neg(\varphi \rightarrow \neg\psi)$

Minimal Sets of Logical Connectives

- “ $\rightarrow$ ”, “ $\perp$ ”:

$\neg\varphi$ can be expressed as $\varphi \rightarrow \perp$

$\varphi \vee \psi$ can be expressed as $(\varphi \rightarrow \perp) \rightarrow \psi$

$\varphi \wedge \psi$ can be expressed as $(\varphi \rightarrow (\psi \rightarrow \perp)) \rightarrow \perp$

- “ $|$ ” — NAND — Sheffer stroke (also denoted by “ $\uparrow$ ”):

φ	ψ	$\varphi \psi$
0	0	1
0	1	1
1	0	1
1	1	0

$\neg\varphi$ can be expressed as $\varphi | \varphi$

$\varphi \vee \psi$ can be expressed as $(\varphi | \varphi) | (\psi | \psi)$

$\varphi \wedge \psi$ can be expressed as $(\varphi | \psi) | (\varphi | \psi)$

Minimal Sets of Logical Connectives

- “ $\downarrow$ ” — NOR — Peirce's arrow:

φ	ψ	$\varphi \downarrow \psi$
0	0	1
0	1	0
1	0	0
1	1	0

$\neg\varphi$ can be expressed as $\varphi \downarrow \varphi$

$\varphi \vee \psi$ can be expressed as $(\varphi \downarrow \psi) \downarrow (\varphi \downarrow \psi)$

$\varphi \wedge \psi$ can be expressed as $(\varphi \downarrow \varphi) \downarrow (\psi \downarrow \psi)$

Definition

Formula ψ **logically follows** from assumptions $\varphi_1, \varphi_2, \dots, \varphi_n$ if formula ψ is true in every truth valuation v where all these assumptions are true.

The fact the ψ logically follows from $\varphi_1, \varphi_2, \dots, \varphi_n$ is denoted

$$\varphi_1, \varphi_2, \dots, \varphi_n \models \psi.$$

- $\varphi_1, \varphi_2, \dots, \varphi_n$ — assumptions
- ψ — conclusion

Logical Entailment

Example: Conclusion $r \rightarrow p$ logically follows from assumption $p \vee (q \wedge \neg r)$, i.e.,

$$p \vee (q \wedge \neg r) \models r \rightarrow p$$

	p	q	r	$p \vee (q \wedge \neg r)$	$r \rightarrow p$
	0	0	0	0	1
	0	0	1	0	0
*	0	1	0	1	1
	0	1	1	0	0
*	1	0	0	1	1
*	1	0	1	1	1
*	1	1	0	1	1
*	1	1	1	1	1

Example:

- *If the train arrives late and there are no taxis at the station, then John is late for his meeting.*
 - *John is not late for his meeting.*
 - *The train did arrive late.*
-
- *There were taxis at the station.*

$$(p \wedge \neg q) \rightarrow r, \neg r, p \models q$$

$$(p \wedge \neg q) \rightarrow r, \neg r, p \models q$$

p	q	r	$(p \wedge \neg q) \rightarrow r$	$\neg r$	p	q
0	0	0	1	1	0	0
0	0	1	1	0	0	0
0	1	0	1	1	0	1
0	1	1	1	0	0	1
1	0	0	0	1	1	0
1	0	1	1	0	1	0
1	1	0	1	1	1	1
1	1	1	1	0	1	1

*

To find out whether ψ logically follows from assumptions $\varphi_1, \varphi_2, \dots, \varphi_n$, the **table method** can be used:

- If, in all lines corresponding to valuations where all assumptions $\varphi_1, \varphi_2, \dots, \varphi_n$ have value 1, ψ also have value 1 then the conclusion ψ **logically follows** from assumptions $\varphi_1, \varphi_2, \dots, \varphi_n$.
- If there exists at least one valuation in the table where $\varphi_1, \varphi_2, \dots, \varphi_n$ have value 1 and conclusion ψ has value 0 then this conclusion **does not logically follow** from the assumptions.

Remark: Those valuations where at least one of assumptions $\varphi_1, \varphi_2, \dots, \varphi_n$ has value 0 are not important with respect to logical entailment — the conclusion ψ can be true or false in these valuations.

Logical Entailment

To find out whether ψ logically follows from assumptions $\varphi_1, \varphi_2, \dots, \varphi_n$, also **semantic contradiction** can be used:

- A directed acyclic graph common for all formulas $\varphi_1, \varphi_2, \dots, \varphi_n$ and ψ is created.
- Values 1 are assigned to nodes corresponding to assumptions $\varphi_1, \varphi_2, \dots, \varphi_n$ and value 0 to the node corresponding to conclusion ψ .
- If it is possible to assign values to all remaining nodes of the graph, we have an example of a valuation where the assumptions are true but the conclusion is false — i.e., the conclusion **does not logically follow** from the assumptions.
- If we show that there is no such valuation (because every attempt to assign remaining values to nodes leads to a contradiction), the conclusion **logically follows** from the assumptions.

If a formula ψ is a **tautology** then it logically follows from every set of assumptions, i.e., for every set of assumptions $\varphi_1, \varphi_2, \dots, \varphi_n$ it holds that

$$\varphi_1, \varphi_2, \dots, \varphi_n \models \psi$$

In particular, if ψ is a tautology then it follows from the empty set of assumptions:

$$\models \psi$$

Tautologies are the only formulas that logically follow from the empty set of assumptions.

Logical Entailment

The question whether a given conclusion logically follows from given assumptions can be reformulated as a question whether a certain formula is a tautology:

$$\varphi_1, \varphi_2, \varphi_3, \dots, \varphi_n \models \psi$$

iff

$\varphi_1 \rightarrow (\varphi_2 \rightarrow (\varphi_3 \rightarrow (\dots \rightarrow (\varphi_n \rightarrow \psi) \dots)))$ is a tautology

Example:

$$\varphi_1, \varphi_2, \varphi_3, \varphi_4 \models \psi$$

iff

$\varphi_1 \rightarrow (\varphi_2 \rightarrow (\varphi_3 \rightarrow (\varphi_4 \rightarrow \psi)))$ is a tautology

Logical Entailment

The following two formulas are logically equivalent:

- $\varphi_1 \rightarrow (\varphi_2 \rightarrow (\varphi_3 \rightarrow (\dots \rightarrow (\varphi_n \rightarrow \psi) \dots)))$
- $(\varphi_1 \wedge \varphi_2 \wedge \dots \wedge \varphi_n) \rightarrow \psi$

For example

$$\varphi_1 \rightarrow (\varphi_2 \rightarrow (\varphi_3 \rightarrow (\varphi_4 \rightarrow \psi))) \Leftrightarrow (\varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_4) \rightarrow \psi$$

(This can be easily checked by equivalent transformations using the following equivalence: $p \rightarrow (q \rightarrow r) \Leftrightarrow (p \wedge q) \rightarrow r$)

So it also holds that

$$\varphi_1, \varphi_2, \dots, \varphi_n \models \psi$$

iff

$$(\varphi_1 \wedge \varphi_2 \wedge \dots \wedge \varphi_n) \rightarrow \psi \text{ is a tautology}$$

One more possibility how to characterize when a conclusion follows from given assumptions, is provided by the following equivalence:

$$\varphi_1, \varphi_2, \dots, \varphi_n \models \psi$$

iff

$$\varphi_1 \wedge \varphi_2 \wedge \dots \wedge \varphi_n \Leftrightarrow \varphi_1 \wedge \varphi_2 \wedge \dots \wedge \varphi_n \wedge \psi$$

When some conclusion ψ , which logically follows from given assumptions $\varphi_1, \varphi_2, \dots, \varphi_n$, to these assumptions as new additional assumption, it does not change the set of truth valuations where the assumptions are true.

(The sets of assumptions $\varphi_1, \varphi_2, \dots, \varphi_n$ and $\varphi_1, \varphi_2, \dots, \varphi_n, \psi$ are true at the same truth valuations.)

So when an assumption ψ that logically follows from given assumption is added to these assumptions, the set of all conclusions that follow the assumptions is not affected:

$$\varphi_1, \varphi_2, \dots, \varphi_n \models \psi$$

iff

it holds for each formula χ that:

$$\varphi_1, \varphi_2, \dots, \varphi_n \models \chi \quad \text{iff} \quad \varphi_1, \varphi_2, \dots, \varphi_n, \psi \models \chi$$

Logical Entailment

If some conclusions logically follow from given assumptions and some other conclusion follows from these conclusions, then this conclusion logically follows also from the original assumptions.

Let us assume that

$$\varphi_1, \varphi_2, \dots, \varphi_n \models \chi_1$$

$$\varphi_1, \varphi_2, \dots, \varphi_n \models \chi_2$$

and also $\chi_1, \chi_2 \models \psi$.

Then $\varphi_1, \varphi_2, \dots, \varphi_n \models \psi$.

Example:

- If $\varphi_1, \varphi_2, \varphi_3 \models (q \vee \neg p)$ and $\varphi_1, \varphi_2, \varphi_3 \models \neg s$ then

$$\varphi_1, \varphi_2, \varphi_3 \models (q \vee \neg p) \wedge \neg s$$

because $q \vee \neg p, \neg s \models (q \vee \neg p) \wedge \neg s$.

Example:

- If $\varphi_1, \varphi_2, \varphi_3 \models p \rightarrow q$ and $\varphi_1, \varphi_2, \varphi_3 \models p$ then

$$\varphi_1, \varphi_2, \varphi_3 \models q$$

because $p \rightarrow q, p \models q$.

Example:

- If $\varphi_1, \varphi_2, \varphi_3, \varphi_4 \models \neg p \rightarrow \neg q$ then

$$\varphi_1, \varphi_2, \varphi_3, \varphi_4 \models q \rightarrow p$$

because $\neg p \rightarrow \neg q \models q \rightarrow p$.

When we try to prove that ψ logically follows from assumptions $\varphi_1, \varphi_2, \dots, \varphi_n$, we can proceed via smaller steps.

We start with the assumptions, for example:

$\varphi_1, \varphi_2, \varphi_3$

Then we gradually add other formulas in such a way that every newly added formula logically follows from the previous formulas. For example:

$\varphi_1, \varphi_2, \varphi_3, \chi_1, \chi_2, \chi_3, \chi_4, \chi_5, \chi_6, \chi_7, \chi_8, \psi$

Assumptions $\varphi_1, \varphi_2, \dots, \varphi_n$ are **inconsistent** (**contradictory**) if there is no truth valuation v , in which all these assumptions would be true.

Assumptions $\varphi_1, \varphi_2, \dots, \varphi_n$ are inconsistent iff

$$\varphi_1 \wedge \varphi_2 \wedge \dots \wedge \varphi_n$$

is a contradiction.

Example: Assumptions $p \rightarrow q, r \rightarrow p, r, \neg q$ are inconsistent.

From inconsistent assumptions, any conclusion logically follows.

If assumptions $\varphi_1, \varphi_2, \dots, \varphi_n$ are inconsistent then it holds for every formula ψ that

$$\varphi_1, \varphi_2, \dots, \varphi_n \models \psi.$$

So for example also the following conclusions follow from inconsistent assumptions:

- $\perp$
- formulas χ and $\neg\chi$, where χ is an arbitrary formula

Logical Entailment

Formula $\perp$ cannot be true and it is also not possible that formulas χ and $\neg\chi$ are both true at some truth valuation.

So when we find out that

- $\perp$ or,
- χ and also $\neg\chi$,

logically follow from assumptions $\varphi_1, \varphi_2, \dots, \varphi_n$, this means that the assumptions $\varphi_1, \varphi_2, \dots, \varphi_n$ are inconsistent and anything follows from them.

Remark: Note that the following formulas are tautologies:

- $\perp \rightarrow \psi$
- $\chi \rightarrow (\neg\chi \rightarrow \psi)$

So $\perp \models \psi$ and $\chi, \neg\chi \models \psi$.

The principle of proof by contradiction

$$\varphi_1, \varphi_2, \dots, \varphi_n \models \psi$$

iff

assumptions $\varphi_1, \varphi_2, \dots, \varphi_n, \neg\psi$ are inconsistent

So in a proof by contradiction, justification that a given conclusion follows from given assumptions is transformed to justification that it is not possible that the assumptions and the negation of the conclusion would be true at the same time.

The question whether $\varphi_1, \varphi_2, \dots, \varphi_n \models \psi$ can be transformed to the question whether

$$\varphi_1 \wedge \varphi_2 \wedge \dots \wedge \varphi_n \wedge \neg\psi$$

is a **contradiction**.

Resolution Method

The **resolution method** is one of algorithms for finding out whether a given conclusion follows from given assumptions.

It solves the following problem:

Input: Formulas $\varphi_1, \varphi_2, \dots, \varphi_n, \psi$.

Question: Is it true that $\varphi_1, \varphi_2, \dots, \varphi_n \models \psi$?

Remark: The method can be used for finding out whether a given formula is a tautology, a contradiction, or satisfiable.

Different variants of the resolution method are used for example in some systems for automatic theorem proving and also in implementations of logic programming languages such as Prolog.

- It works with formulas in CNF.
- It constructs a proof that the given conclusion follows from the assumptions.
- It is a proof by contradiction — the algorithm generates successively formulas following from the assumptions

$$\varphi_1, \varphi_2, \dots, \varphi_n, \neg\psi$$

- A computation can finish in two different ways:
 - A contradiction is found, i.e., formula $\perp$ is derived — then the conclusion ψ logically follows from assumptions $\varphi_1, \varphi_2, \dots, \varphi_n$.
 - The algorithm does not succeed in deriving formula $\perp$ and no other new formulas can be added — then the conclusion ψ does not follow from the assumptions.

- The resolution method works with formulas that have the form of elementary disjunctions, e.g.,

$$(\neg p \vee q \vee \neg s \vee \neg t)$$

These formulas are called **clauses**.

- A special case of clause is the **empty clause** $\perp$ that represents a found contradiction.
- The algorithm starts the computation by transforming formulas

$$\varphi_1, \varphi_2, \dots, \varphi_n, \neg\psi$$

to CNF. Then it takes all clauses from the transformed formulas as the initial set of assumptions

$$\chi_1, \chi_2, \dots, \chi_m.$$

Resolution Method

For generating other clauses, which are added to already constructed clauses, the algorithm uses so called **resolution rule** (or **resolution principle**):

For each formulas φ , ψ and χ it holds that

$$\varphi \vee \psi, \neg\varphi \vee \chi \models \psi \vee \chi$$

In the resolution method, this principle is used only for clauses. In the resolution method, $\varphi \vee \psi$, $\neg\varphi \vee \chi$ and $\psi \vee \chi$ are always clauses, and φ is always an atomic proposition.

Example: From clauses

$$p \vee \neg q \vee r \vee s \quad \text{a} \quad \neg r \vee t \vee \neg u$$

we can derive the following clause by the resolution rule:

$$p \vee \neg q \vee s \vee t \vee \neg u.$$

Remarks:

- An order of literals in a clause is not important.
- Multiple occurrences of the same literals in one clause can be eliminated.
- If a currently generated clause is the same as some previously generated clause (and differs only in the order of literals), it makes no sense to add it.
- Clauses containing both literals p and $\neg p$ are equivalent to $\top$ and can be eliminated.
- Clauses can be used for the application of the resolution rule repeatedly (with other clauses).

Some special cases of the use of the resolution rule:

- One of clauses contains just one literal and the other more than one literal:

From clauses

$$\neg q \qquad p \vee q \vee \neg t$$

we can derive clause $p \vee \neg t$.

- Both clauses contain just one literal:

From clauses

$$p \qquad \neg p$$

we can derive the empty clause $\perp$, i.e., the contradiction.

We want to check validity of the following deduction:

- *It is not true that Jane is at school and Peter is not at home.*
 - *Jane is not at school or it's a working day or it's raining.*
 - *If it's a working day then Peter is not at home.*
-
- *If Jane is at school then it's raining.*

At first, we formalize the individual propositions by formulae of propositional logic:

$$\neg(j \wedge \neg p)$$

$$\neg j \vee d \vee r$$

$$d \rightarrow \neg p$$

$$j \rightarrow r$$

j – Jane is at school

p – Peter is at home

d – it's a working day

r – it's raining

$$\frac{\begin{array}{l} \neg(j \wedge \neg p) \\ \neg j \vee d \vee r \\ d \rightarrow \neg p \end{array}}{j \rightarrow r}$$

We transform the individual assumptions into CNF:

- $\neg(j \wedge \neg p) \Leftrightarrow \neg j \vee p$
- $\neg j \vee d \vee r$
- $d \rightarrow \neg p \Leftrightarrow \neg d \vee \neg p$

We negate the conclusion and transform it into CNF:

- $\neg(j \rightarrow r) \Leftrightarrow j \wedge \neg r$

Let us write down the individual clauses:

1. $\neg j \vee p$ – assumption 1
 2. $\neg j \vee d \vee r$ – assumption 2
 3. $\neg d \vee \neg p$ – assumption 3
 4. j – clause 1 of the negated conclusion
 5. $\neg r$ – clause 2 of the negated conclusion
-

Resolution Method

Let us write down the individual clauses:

1. $\neg j \vee p$ – assumption 1
2. $\neg j \vee d \vee r$ – assumption 2
3. $\neg d \vee \neg p$ – assumption 3
4. j – clause 1 of the negated conclusion
5. $\neg r$ – clause 2 of the negated conclusion

6. p – resolution: 1,4

Resolution Method

Let us write down the individual clauses:

1. $\neg j \vee p$ – assumption 1
 2. $\neg j \vee d \vee r$ – assumption 2
 3. $\neg d \vee \neg p$ – assumption 3
 4. j – clause 1 of the negated conclusion
 5. $\neg r$ – clause 2 of the negated conclusion
-
6. p – resolution: 1,4
 7. $d \vee r$ – resolution: 2,4

Resolution Method

Let us write down the individual clauses:

- | | | |
|-------|------------------------|--------------------------------------|
| 1. | $\neg j \vee p$ | – assumption 1 |
| 2. | $\neg j \vee d \vee r$ | – assumption 2 |
| 3. | $\neg d \vee \neg p$ | – assumption 3 |
| 4. | j | – clause 1 of the negated conclusion |
| 5. | $\neg r$ | – clause 2 of the negated conclusion |
| <hr/> | | |
| 6. | p | – resolution: 1,4 |
| 7. | $d \vee r$ | – resolution: 2,4 |
| 8. | $\neg d$ | – resolution: 3,6 |

Resolution Method

Let us write down the individual clauses:

- | | | |
|-------|------------------------|--------------------------------------|
| 1. | $\neg j \vee p$ | – assumption 1 |
| 2. | $\neg j \vee d \vee r$ | – assumption 2 |
| 3. | $\neg d \vee \neg p$ | – assumption 3 |
| 4. | j | – clause 1 of the negated conclusion |
| 5. | $\neg r$ | – clause 2 of the negated conclusion |
| <hr/> | | |
| 6. | p | – resolution: 1,4 |
| 7. | $d \vee r$ | – resolution: 2,4 |
| 8. | $\neg d$ | – resolution: 3,6 |
| 9. | r | – resolution: 7,8 |

Resolution Method

Let us write down the individual clauses:

- | | | |
|-------|------------------------|--------------------------------------|
| 1. | $\neg j \vee p$ | – assumption 1 |
| 2. | $\neg j \vee d \vee r$ | – assumption 2 |
| 3. | $\neg d \vee \neg p$ | – assumption 3 |
| 4. | j | – clause 1 of the negated conclusion |
| 5. | $\neg r$ | – clause 2 of the negated conclusion |
| <hr/> | | |
| 6. | p | – resolution: 1,4 |
| 7. | $d \vee r$ | – resolution: 2,4 |
| 8. | $\neg d$ | – resolution: 3,6 |
| 9. | r | – resolution: 7,8 |
| 10. | $\perp$ | – resolution: 5,9 |

A contradiction was derived, so the conclusion really follows from the given assumptions.

Remarks:

- The resolution method can be viewed as a construction of one “big” formula in CNF, which is equivalent to

$$\varphi_1 \wedge \varphi_2 \wedge \dots \wedge \varphi_n \wedge \neg\psi,$$

and which is constructed by a successive addition of clauses.

- If a contradiction can not be generated, then the derived clauses can be used for finding a truth valuation ν where the assumptions $\varphi_1, \varphi_2, \dots, \varphi_n$ are true and the conclusion ψ is false.

- It is also possible to proceed by a direct method, where the algorithm starts only with assumptions

$$\varphi_1, \varphi_2, \dots, \varphi_n$$

and tries to generate all clauses of the conclusion ψ .

In this approach, it is not guaranteed that the algorithm succeeds in all cases when a conclusion ψ logically follows from assumptions

$$\varphi_1, \varphi_2, \dots, \varphi_n.$$

Example: Clause $p \vee q$ cannot be generated this way from the assumption p , although it holds that

$$p \models p \vee q.$$

Predicate Logic

- *Fish are vertebrates living in water.*
 - *Carps are fish.*
 - *There exists at least one carp.*
-
- *There exists at least one vertebrate living in water.*

- *Triangles are convex polygons.*
 - *Equilateral triangles are triangles.*
 - *There exists at least one equilateral triangle.*
-
- *There exists at least one convex polygon.*

- *Fish are vertebrates living in water.*
 - *Carps are fish.*
 - *There exists at least one carp.*
-
- *There exists at least one vertebrate living in water.*

The use of **variables**:

- *For each x it holds that if x is a fish then x is a vertebrate and x lives in water.*
 - *For each x it holds that if x is a carp then x is a fish.*
 - *There exists at least one x such that x is a carp.*
-
- *There exists at least one x such that x is a vertebrate and x lives in water.*

- *Triangles are convex polygons.*
 - *Equilateral triangles are triangles.*
 - *There exists at least one equilateral triangle.*
-
- *There exists at least one convex polygon.*

The use of **variables**:

- *For each x it holds that if x is a triangle then x is a polygon and x is convex.*
 - *For each x it holds that if x is an equilateral triangle then x is a triangle.*
 - *There exists at least one x such that x is an equilateral triangle.*
-
- *There exists at least one x such that x is a polygon and x is convex.*

Predicate Logic

- For each x it holds that if x has property P then x has property Q and x has property R .
 - For each x it holds that if x has property S then x has property P .
 - There exists at least one x such that x has property S .
-
- There exists at least one x such that x has property Q and x has property R .

P	is a fish	is a triangle
Q	is a vertebrate	is a polygon
R	lives in water	is convex
S	is a carp	is an equilateral triangle

Predicate Logic

- For each x it holds that if $P(x)$ then $Q(x)$ and $R(x)$.
 - For each x it holds that if $S(x)$ then $P(x)$.
 - There exists x such that $S(x)$.
-
- There exists x such that $Q(x)$ and $R(x)$.

$P(x)$	x is a fish	x is a triangle
$Q(x)$	x is a vertebrate	x is a polygon
$R(x)$	x lives in water	x is convex
$S(x)$	x is a carp	x is an equilateral triangle

Predicate Logic

- For each x , $(P(x) \rightarrow (Q(x) \wedge R(x)))$.
 - For each x , $(S(x) \rightarrow P(x))$.
 - There exists x such that $S(x)$.
-
- There exists x such that $(Q(x) \wedge R(x))$.

$P(x)$	x is a fish	x is a triangle
$Q(x)$	x is a vertebrate	x is a polygon
$R(x)$	x lives in water	x is convex
$S(x)$	x is a carp	x is an equilateral triangle

Predicate Logic

- $\forall x(P(x) \rightarrow (Q(x) \wedge R(x)))$
 - $\forall x(S(x) \rightarrow P(x))$
 - $\exists x S(x)$
-
- $\exists x(Q(x) \wedge R(x))$

$P(x)$	x is a fish	x is a triangle
$Q(x)$	x is a vertebrate	x is a polygon
$R(x)$	x lives in water	x is convex
$S(x)$	x is a carp	x is an equilateral triangle

- $\forall$ — universal quantifier (“*for all*”)
- $\exists$ — existential quantifier (“*there exists*”)

Formulas of propositional logic express propositions about objects with some properties and which can be in some relationships.

Interpretation or **interpretation structure** — a particular set of these objects, their properties and relationships.

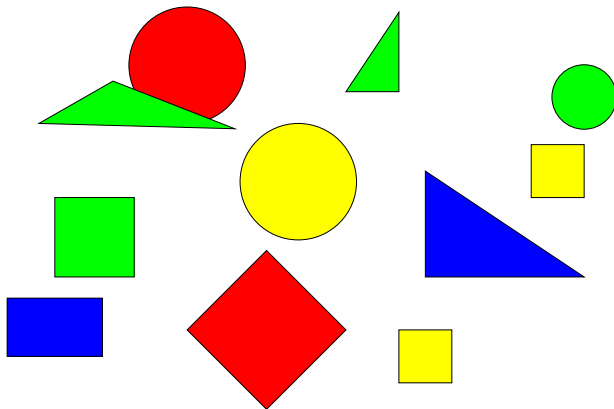
Universe — the set of all objects in a given interpretation

- An arbitrary **non-empty** set can be the universe.
- Objects in a given universe are called the **elements** of the universe.

Valuation — an assignment of elements of the universe to variables

The truth values of formulas depend on a given interpretation and valuation.

An example of a universe:



Other examples of universes:

- Some precisely specified set of people, for example, the set of people that live in some specified house (*“John Smith”, “John Doe”, ...*)
- The set of all books in a given library.
- The set of natural numbers $\mathbb{N} = \{0, 1, 2, 3, \dots\}$.
- The set of all points in a plane.
- The set $\{a, b, c, d, e\}$.
- The set $\{a\}$.

Variables

Variables — $x, y, z, \dots$, possibly with indexes — $x_0, x_1, x_2, \dots$

It is assumed that there are infinitely many variables.

Valuation — an assignment of elements of the universe to the variables

Example:

- Universe — a set of people; valuation v , where:

$$v(x) = \text{"John Doe"}$$

$$v(y) = \text{"Mary Smith"}$$

...

- Universe — the set of natural numbers $\mathbb{N} = \{0, 1, 2, \dots\}$; valuation v , where

$$v(x) = 57$$

$$v(y) = 3$$

$$v(z) = 57$$

...

Predicates — $P, Q, R, \dots$

- **Unary predicates** — they represent **properties** of elements of the universe

Example: Predicate P representing the property “to be blue”:

$$P(x) \quad \text{—} \quad “x \text{ is blue}”$$

A unary predicate assigns truth values to the elements of the universe.

E.g., the value of $P(x)$ can be:

- **1** — the element assigned to variable x has property P (i.e., it is blue)
- **0** — the element assigned to variable x does not have this property P (i.e., it is not blue)

- **Binary predicates** — they represent **relationships** between pairs of elements of the universe

Example: Predicate R representing the relationship “to be a parent of”:

$$R(x, y) \quad \text{—} \quad \text{“}x \text{ is a parent of } y\text{”}$$

A binary predicate assigns truth values to pair of elements of the universe.

E.g., the value of $R(x, y)$ can be:

- **1** — when x and y are in the given relationship (i.e., when x is a parent of y)
- **0** — when x and y are not in the given relationship (i.e., when x is not a parent of y)

We can consider predicates of arbitrary arities.

For example:

- **Ternary** predicate T (i.e., predicate of arity 3) representing the relationship between parents and their child:

$$T(x, y, z)$$

— x and y are parents of child z , and x is his/her mother and y is his/her father

- **Nulary** predicates (i.e., predicates of arity 0) can be viewed as atomic propositions, not related to the elements of the universe.

Formulas of Predicate Logic

Atomic formula — a predicate applied on some variables

Example:

- P — a unary predicate representing property “to be blue”
- Q — a unary predicate representing property “to be a square”
- R — a binary predicate representing relationships “overlaps”

$P(x)$ — “ x is blue”

$P(y)$ — “ y is blue”

$Q(y)$ — “ y is a square”

$R(z, x)$ — “ z overlaps x ”

$R(y, y)$ — “ y overlaps itself”

Remark: Later, we will extend the notion of an atomic formula a little bit.

Formulas of Predicate Logic

Using **logical connectives** ($\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$), more complicated formulas can be created from simpler formulas, similarly as in propositional logic.

Example:

- P — unary predicate representing property “is blue”
- Q — unary predicate representing property “is a square”
- R — binary predicate representing relationship “overlaps”

“If x is a blue square or y does not overlap x , then z is not a square.”

$$((P(x) \wedge Q(x)) \vee \neg R(y, x)) \rightarrow \neg Q(z)$$

Formulas of Predicate Logic

Using **logical connectives** ($\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$), more complicated formulas can be created from simpler formulas, similarly as in propositional logic.

Example:

- P — unary predicate representing property “is a woman”
- Q — unary predicate representing property “has dark hair”
- R — binary predicate representing relationship “is a parent of”

“If x is a woman with dark hair or y is not a parent of x , then z does not have dark hair.”

$$((P(x) \wedge Q(x)) \vee \neg R(y, x)) \rightarrow \neg Q(z)$$

Formulas of Predicate Logic

Using **logical connectives** ($\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$), more complicated formulas can be created from simpler formulas, similarly as in propositional logic.

Example:

- P — unary predicate representing property “is even”
- Q — unary predicate representing property “is a prime”
- R — binary predicate representing relationship “is greater than”

“If x is an even prime or y is not greater than x , then z is not a prime.”

$$((P(x) \wedge Q(x)) \vee \neg R(y, x)) \rightarrow \neg Q(z)$$

Universal quantifier — symbol “ $\forall$ ”

If φ is a formula representing some proposition then

$$\forall x \varphi$$

is a formula representing proposition

“for every x φ holds”.

Example: P — *“to be a square”*

$$\forall x P(x)$$

- *“For every x it holds that x is a square.”*
- *“Every x is a square.”*
- *“All elements are squares.”*

Example:

- “For every x it holds that if x is a square then x is green.”
- “For each x it holds that if x is a square then x is green.”
- “For all x it holds that if x is a square then x is green.”
- “All squares are green.”

$$\forall x(P(x) \rightarrow Q(x))$$

- P — “to be a square” (arity 1)
- Q — “to be green” (arity 1)

Example:

- “If it holds for all x that x is a square or x is green then it holds for all y that y is a triangle.”
- “If every object is a square or is green then all objects are triangles.”

$$\forall x(P(x) \vee Q(x)) \rightarrow \forall yT(y)$$

- P — “to be a square” (arity 1)
- Q — “to be green” (arity 1)
- T — “to be a triangle” (arity 1)

There is a big difference between the following formulas:

- $P(x)$ — “ x is a square”

It claims something about **one** particular element assigned to variable x .

The truth value of this claim depends on the particular element assigned to variable x , i.e., on the particular valuation.

- $\forall x P(x)$ — “every x is a square” (i.e., “all elements are squares”)

It claims something about **all** elements of the universe.

The truth value of this claim does not depend on a valuation.

Example:

- “If x is a prime then x is odd.”

$$P(x) \rightarrow L(x)$$

- “For every x it holds that if x is a prime then it is odd”. (i.e., “all primes are odd”.)

$$\forall x(P(x) \rightarrow L(x))$$

Predicates:

- P — “to be a prime” (arity 1)
- L — “to be odd” (arity 1)

Example:

- *“It holds for every y that if y is green then x overlaps y .”*
- *“Object x overlaps all green objects.”*

$$\forall y (G(y) \rightarrow R(x, y))$$

Predicates:

- R — “overlaps” (arity 2)
- G — “to be green” (arity 1)

Example:

- *"It holds for every x that it holds for every y that if x is a parent of y then x loves y ."*
- *"It holds for each x and y that if x is a parent of y then x loves y ."*
- *"For every pair of elements x and y it holds that if x is a parent of y then x loves y ."*

$$\forall x \forall y (R(x, y) \rightarrow S(x, y))$$

Predicates:

- R — *"is a parent"* (arity 2)
- S — *"loves"* (arity 2)

Existential quantifier — symbol “ $\exists$ ”

If φ is a formula representing some proposition then

$$\exists x \varphi$$

is a formula representing proposition

“there exists x , for which φ holds”.

Example: P — *“to be a square”*

$$\exists x P(x)$$

- *“There exists x , for which it holds that x is a square.”*
- *“There is x such that x is a square.”*
- *“There exists at least one square.”*

Example:

- “There exists x , for which it holds that x is a square and x is green.”
- “There is x such that x is a square and x is green.”
- “For some x it holds that x is a square and x is green.”
- “There exists a green square.”
- “Some squares are green.”
- “At least one x is a green square.”

$$\exists x(P(x) \wedge Q(x))$$

Predicates:

- P — “to be a square” (arity 1)
- Q — “to be green” (arity 1)

Example:

- “There exists x such that for each y it holds that x is greater than y .”

$$\exists x \forall y P(x, y)$$

- “For each y there is x such that x is greater than y .”

$$\forall y \exists x P(x, y)$$

P — “to be greater than” (arity 2)

Alphabet:

- **logical connectives** — “ $\neg$ ”, “ $\wedge$ ”, “ $\vee$ ”, “ $\rightarrow$ ”, “ $\leftrightarrow$ ”
- **quantifiers** — “ $\forall$ ”, “ $\exists$ ”
- **auxiliary symbols** — “(”, “)”, “,”
- **variables** — “ x ”, “ y ”, “ z ”, $\dots$, “ x_0 ”, “ x_1 ”, “ x_2 ”, $\dots$
- **predicate symbols** — for example symbols “ P ”, “ Q ”, “ R ”, etc.
(for each symbol, its arity must be specified)
- $\dots$

Remark: Other types of symbols will be described later.

Syntax of Formulas of Predicate Logic

Definition

Well-formed **atomic formulas** of predicate logic are formulas of the form:

- $P(x_1, x_2, \dots, x_n)$, where P is a predicate symbol of arity n and $x_1, x_2, \dots, x_n$ are (not necessarily different) variables.
- ...

Remark: This is not the whole definition. Later, it will be generalized a little bit and some additional items will be added.

Example:

$P(x, y)$

$R(z, z, z)$

$S(y)$

Definition

Well-formed **formulas of predicate logic** are sequences of symbols constructed according to the following rules:

- 1 Well-formed atomic formulas are well-formed formulas.
- 2 If φ and ψ are well-formed formulas, then also $(\neg\varphi)$, $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\varphi \rightarrow \psi)$ and $(\varphi \leftrightarrow \psi)$ are well-formed formulas.
- 3 If φ is a well-formed formula and x is a variable, then $\forall x\varphi$ and $\exists x\varphi$ are well-formed formulas.
- 4 There are no other well-formed formulas than those constructed according to the previous rules.

Notions like

- subformulas
- an abstract syntax tree

are introduced in a similar way like in propositional logic (they are only extended with the additional constructions not present in propositional logic).

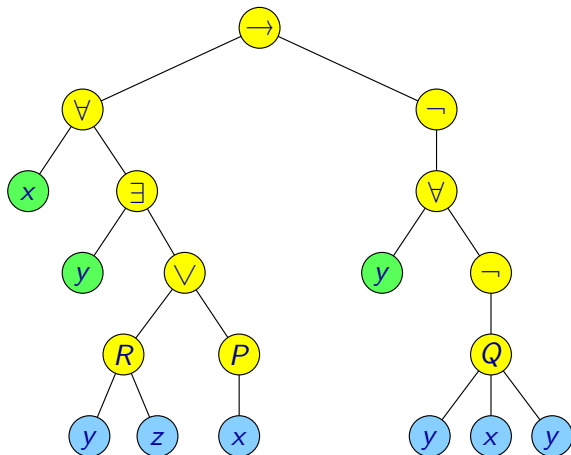
Convention for omitting parentheses:

- The same conventions as in propositional logic.
- Quantifiers ($\forall$ and $\exists$) have the same priority as negation ($\neg$), i.e., the highest priority.

Syntax of Formulas of Predicate Logic

An abstract syntax tree of formula

$$\forall x \exists y (R(y, z) \vee P(x)) \rightarrow \neg \forall y \neg Q(y, x, y)$$



Free and Bound Occurrences of Variables

Every occurrence of variable x in a subformula of the form $\exists x\varphi$ or $\forall x\varphi$ is **bound**.

An occurrence of a variable, which is not bound, is **free**.

Example: Formula

$$\forall x \exists y (R(y, z) \vee P(x)) \rightarrow \neg \forall y \neg Q(y, x, y)$$

- y in subformula $R(y, z)$ — the bound occurrence ($\exists y$)
- z in subformula $R(y, z)$ — the free occurrence
- x in subformula $P(x)$ — the bound occurrence ($\forall x$)
- both occurrences of y in subformula $Q(y, x, y)$ — the bound occurrences ($\forall y$)
- x in subformula $Q(y, x, y)$ — the free occurrence

Free and Bound Occurrences of Variables

The set of those variables, which occur as **free** variables in formula φ , will be denoted $free(\varphi)$.

Example:

- If φ is formula $P(x, y)$, then $free(\varphi) = \{x, y\}$.
- If ψ is formula $\exists x \exists y P(x, y)$, then $free(\psi) = \emptyset$.

- If χ is formula

$$\forall x \exists y (R(y, z) \vee P(x)) \rightarrow \neg \forall y \neg Q(y, x, y)$$

then $free(\chi) = \{x, z\}$.

Free and Bound Occurrences of Variables

The set of free variables $free(\varphi)$ can be described by the following inductive definition:

- $free(P(x_1, x_2, \dots, x_n)) = \{x_1, x_2, \dots, x_n\}$
(where P is a predicate symbol)
- $free(\neg\varphi) = free(\varphi)$
- $free(\varphi \wedge \psi) = free(\varphi) \cup free(\psi)$
(it is similar for formulas of the form $\varphi \vee \psi$, $\varphi \rightarrow \psi$, and $\varphi \leftrightarrow \psi$)
- $free(\forall x\varphi) = free(\varphi) - \{x\}$ (where x is a variable)
- $free(\exists x\varphi) = free(\varphi) - \{x\}$ (where x is a variable)

Free and Bound Occurrences of Variables

A formula φ is **closed** if it contains no free occurrences of variables (i.e., when $\text{free}(\varphi) = \emptyset$).

A formula φ is **open** if it is not closed (i.e., when $\text{free}(\varphi) \neq \emptyset$).

Remark: Closed formulas are sometimes also called **sentences**.

Example:

- Formula $\exists x \exists y P(x, y)$ is closed.
- Formula $\forall x \exists y (R(y, z) \vee P(x)) \rightarrow \neg \forall y \neg Q(y, x, y)$ is open (because it contains free occurrences of variables z and x).

Truth values of closed formulas do not depend on a valuation, only on an interpretation.

Formulas are evaluated in a given **interpretation** (**interpretation structure**) and **valuation**.

The fact that formula φ holds (i.e., it has truth value 1) in interpretation $\mathcal{A}$ and valuation v , is denoted

$$\mathcal{A}, v \models \varphi$$

The fact that formula φ does not hold (i.e., it has truth value 0) in interpretation $\mathcal{A}$ and valuation v , is denoted $\mathcal{A}, v \not\models \varphi$.

Semantics of Predicate Logic

An **interpretation** $\mathcal{A}$ is a structure consisting of the following items:

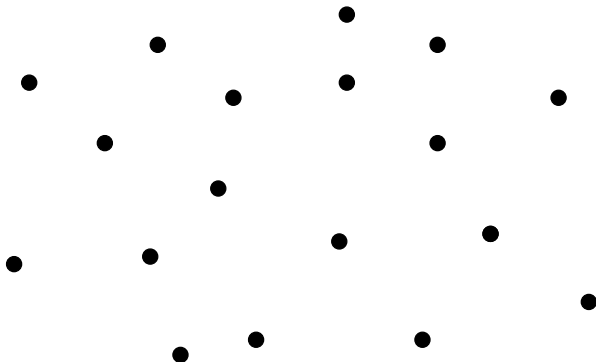
- **Universe** A — an arbitrary non-empty set
- Some subset of the set A is assigned to every unary predicate symbol P — it is denoted $P^{\mathcal{A}}$.
(And so $P^{\mathcal{A}} \subseteq A$.)
- Some binary relation on A is assigned to every binary predicate symbol Q — it is denoted $Q^{\mathcal{A}}$.
(And so $Q^{\mathcal{A}} \subseteq A \times A$.)
- It is similar for predicate symbols with other arities (3, 4, 5, ...).

Remark: This definition is not complete yet, and it will be later extended by other items.

Semantics of Predicate Logic

An example of an interpretation $\mathcal{A}$:

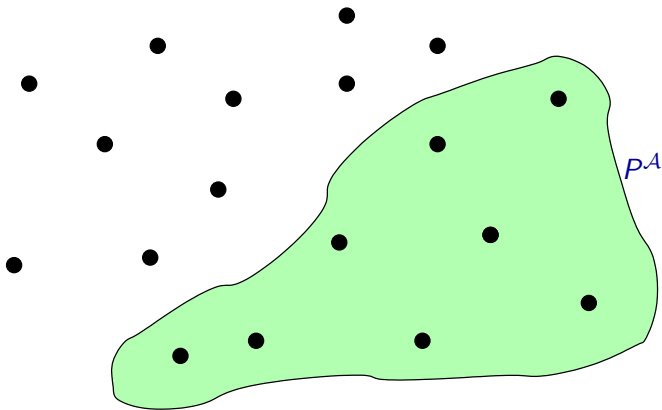
universe A



Semantics of Predicate Logic

An example of an interpretation $\mathcal{A}$:

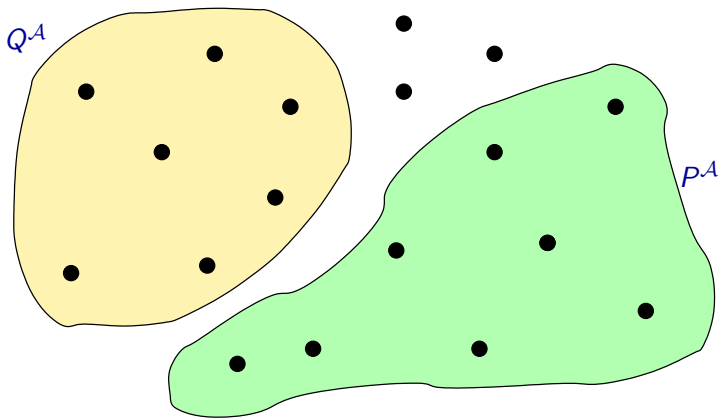
universe A



Semantics of Predicate Logic

An example of an interpretation $\mathcal{A}$:

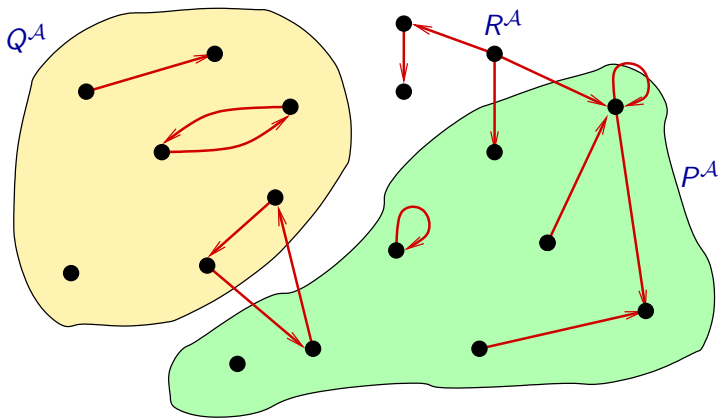
universe A



Semantics of Predicate Logic

An example of an interpretation $\mathcal{A}$:

universe A



Other example of an interpretation $\mathcal{A}$:

- universe $A = \{a, b, c, d, e, f, g\}$
- $P^{\mathcal{A}} = \{b, d, e\}$
- $Q^{\mathcal{A}} = \{a, b, e, g\}$
- $R^{\mathcal{A}} = \{(a, b), (a, e), (a, g), (b, b), (c, e), (f, c), (f, g), (g, a), (g, g)\}$

Semantics of Predicate Logic

Let Var be the set of all variables, i.e.,

$$Var = \{x, y, z, \dots, x_0, x_1, x_2, \dots\}$$

For a given interpretation $\mathcal{A}$ with a universe A , a **valuation** v is an arbitrary function

$$v : Var \rightarrow A$$

that assigns elements of the universe to the variables.

Remark: As we will see, in fact, only values assigned by the valuation v to variables in $free(\varphi)$ are important for determining the truth value of formula φ .

Values assigned by valuation v to the other variables are not important from this point of view.

Semantics of Predicate Logic

Let us consider an interpretation $\mathcal{A}$ with universe A and a valuation v .

Lets assume that (i.e., $x \in Var$) and a is an element of the universe (i.e., $a \in A$).

Notation

$$v[x \mapsto a]$$

denotes the valuation $v' : Var \rightarrow A$, which assigns to every variable the same value as valuation v , except that it assigns value a to variable x .

I.e., for every variable y (where $y \in Var$) is

$$v'(y) = \begin{cases} a & \text{if } y = x \\ v(y) & \text{otherwise} \end{cases}$$

Example:

- universe $A = \{a, b, c, d, e, f, g, \dots\}$

valuation v :

$$v(x_0) = c \quad v(x_1) = e \quad v(x_2) = b \quad v(x_3) = e \quad \dots$$

valuation $v[x_2 \mapsto g]$:

$$v(x_0) = c \quad v(x_1) = e \quad v(x_2) = g \quad v(x_3) = e \quad \dots$$

Definition

Let us assume an interpretation $\mathcal{A}$ with universe A and a valuation v , assigning elements of the universe A to the variables.

The **truth values of formulas of predicate logic** in interpretation $\mathcal{A}$ and valuation v are defined as follows:

- For a predicate P of arity n , $\mathcal{A}, v \models P(x_1, x_2, \dots, x_n)$ iff $(v(x_1), v(x_2), \dots, v(x_n)) \in P^{\mathcal{A}}$.
- $\mathcal{A}, v \models \neg \varphi$ iff $\mathcal{A}, v \not\models \varphi$.
- $\mathcal{A}, v \models \varphi \wedge \psi$ iff $\mathcal{A}, v \models \varphi$ and $\mathcal{A}, v \models \psi$.
- $\mathcal{A}, v \models \varphi \vee \psi$ iff $\mathcal{A}, v \models \varphi$ or $\mathcal{A}, v \models \psi$.
- $\mathcal{A}, v \models \varphi \rightarrow \psi$ iff $\mathcal{A}, v \not\models \varphi$ or $\mathcal{A}, v \models \psi$.
- $\mathcal{A}, v \models \varphi \leftrightarrow \psi$ iff $\mathcal{A}, v \models \varphi$ and $\mathcal{A}, v \models \psi$, or $\mathcal{A}, v \not\models \varphi$ and $\mathcal{A}, v \not\models \psi$.
- ...

Definition (cont.)

- ...
- $\mathcal{A}, v \models \forall x \varphi$ iff for **every** $a \in A$ it holds that $\mathcal{A}, v[x \mapsto a] \models \varphi$.
- $\mathcal{A}, v \models \exists x \varphi$ iff there **exists** some $a \in A$ such that $\mathcal{A}, v[x \mapsto a] \models \varphi$.

A closed formula φ is **true** (i.e., it has truth value 1) in interpretation $\mathcal{A}$ if it holds for each valuation v that $\mathcal{A}, v \models \varphi$.

The fact that formula φ is true in interpretation $\mathcal{A}$ is denoted

$$\mathcal{A} \models \varphi.$$

Remark: A truth value of a closed formula in a given interpretation does not depend on a valuation.

Consider a closed formula φ .

A **model** of the formula φ is an arbitrary interpretation $\mathcal{A}$ such that $\mathcal{A} \models \varphi$.

Evaluation of Truth of Formulas as a Game

Let us consider a formula of the form

$$\exists x_1 \forall x_2 \exists x_3 \forall x_4 \cdots \exists x_{n-1} \forall x_n \varphi,$$

where quantifiers alternate in some arbitrary way, and where φ does not contain quantifiers.

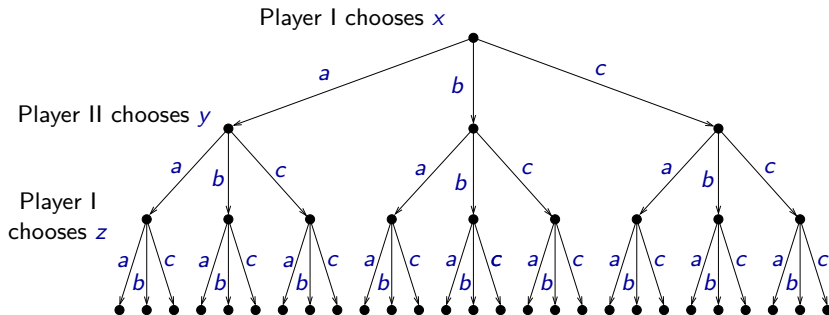
The evaluation of truth values of formulas of this form (in a given interpretation $\mathcal{A}$ and a valuation v) can be viewed as a game:

- It is played by a pair of players — **Player I** and **Player II**.
- Player I wants to show that the formula is true.
- Player II wants to show that the formula is false.
- Player I chooses values of those variables, which are bound by an existential quantifier ($\exists$).
- Player II chooses values of those variables, which are bound by an universal quantifier ($\forall$).

Evaluation of Truth of Formulas as a Game

Example: Formula $\exists x \forall y \exists z (P(x, y) \rightarrow Q(y, z))$

universe $A = \{a, b, c\}$



Evaluation of Truth of Formulas as a Game

- Formula φ is true iff Player I has a **winning strategy** in this game.
- Formula φ is false iff Player II has a winning strategy.

Strategy — determines how a player should play in every situation, i.e., it determines moves of the player for all possible moves of the other player.

Winning strategy — a strategy that guarantees a win of the given player in every play, no matter what the other player does.

Evaluation of Truth of Formulas as a Game

Example: Interpretation where universe is the set of real numbers $\mathbb{R}$ and binary predicate symbol R represents relation “greater or equal” (i.e., $R(x, y)$ iff $x \geq y$).

Formula $\exists x \forall y R(x, y)$ — a winning strategy of Player II:

- Player I chooses number x .
- Player II chooses number $y = x + 1$ — Player II wins since it is obviously not true that $x \geq x + 1$.

Formula $\forall y \exists x R(x, y)$ — a winning strategy of Player I:

- Player II chooses number y .
- Player I chooses number $x = y$ — Player I wins since it is obviously true that $x \geq x$.

Logically Valid Formulas

A formula φ is **logically valid** if it has truth value 1 in every interpretation and valuation, i.e., if for every interpretation $\mathcal{A}$ and valuation v is

$$\mathcal{A}, v \models \varphi.$$

Example:

- $\exists x P(x) \rightarrow \exists y P(y)$
- $\forall x P(x) \wedge \neg \exists y Q(y) \rightarrow \forall z (P(z) \wedge \neg Q(z))$
- $\forall x P(x) \rightarrow \exists x P(x)$

Logically Valid Formulas

If we take an arbitrary tautology of propositional logic and replace in it all atomic propositions with arbitrary formulas of predicate logic, we obtain a logically valid formula.

Example: Tautology $p \rightarrow (q \vee p)$

- p is replaced with $\forall z(P(x, z) \leftrightarrow \neg Q(z, y))$
- q is replaced with $R(x)$

We obtain a logically valid formula

$$\forall z(P(x, z) \leftrightarrow \neg Q(z, y)) \rightarrow (R(x) \vee \forall z(P(x, z) \leftrightarrow \neg Q(z, y)))$$

Logically Equivalent Formulas

Formulas φ and ψ are **logically equivalent** if they have the same truth values in every interpretation and valuation, i.e., if for every interpretation $\mathcal{A}$ and valuation v is

$$\mathcal{A}, v \models \varphi \quad \text{iff} \quad \mathcal{A}, v \models \psi.$$

The fact that φ and ψ are logically equivalent is denoted

$$\varphi \Leftrightarrow \psi.$$

- Similarly as in propositional logic, we can do equivalent transformations in predicate logic.
- All equivalences that hold in propositional logic also hold in predicate logic.

Logically Equivalent Formulas

- There are other equivalences in predicate logic that have no analogy in propositional logic.

Examples of some important equivalences:

$$\neg \forall x \varphi \Leftrightarrow \exists x \neg \varphi$$

$$\neg \exists x \varphi \Leftrightarrow \forall x \neg \varphi$$

$$\forall x \forall y \varphi \Leftrightarrow \forall y \forall x \varphi$$

$$\exists x \exists y \varphi \Leftrightarrow \exists y \exists x \varphi$$

When $x \notin \text{free}(\varphi)$:

$$\forall x \varphi \Leftrightarrow \varphi$$

$$\exists x \varphi \Leftrightarrow \varphi$$

Some other important equivalences:

$$(\forall x\varphi) \wedge (\forall x\psi) \Leftrightarrow \forall x(\varphi \wedge \psi)$$

$$(\exists x\varphi) \vee (\exists x\psi) \Leftrightarrow \exists x(\varphi \vee \psi)$$

When $x \notin \text{free}(\psi)$:

$$(\forall x\varphi) \wedge \psi \Leftrightarrow \forall x(\varphi \wedge \psi)$$

$$(\forall x\varphi) \vee \psi \Leftrightarrow \forall x(\varphi \vee \psi)$$

$$(\exists x\varphi) \wedge \psi \Leftrightarrow \exists x(\varphi \wedge \psi)$$

$$(\exists x\varphi) \vee \psi \Leftrightarrow \exists x(\varphi \vee \psi)$$

Renaming of Bound Variables

If we rename a bound variable in a formula, we obtain an equivalent formula.

Example:

$$\forall xP(x, y) \Leftrightarrow \forall zP(z, y)$$

- If we rename for example x to y in formula $\forall x\varphi$ or $\exists x\varphi$, the variable y **must not** occur in formula φ as a free variable.

$$\exists xP(x, y) \quad \text{is not equivalent to} \quad \exists yP(y, y)$$

- Free occurrences of variables in a subformula **must not** become bound after renaming. E.g.,

$$\exists x\forall yP(x, y) \quad \text{is not equivalent to} \quad \exists y\forall yP(y, y)$$

Substitution

Let us say that we want to replace **free** occurrences of variable x with variable y (i.e., we want to substitute y for x).

This operation on formulas is called **substitution** and the resulting formula is denoted

$$\varphi[y/x].$$

Remark: In general, formulas φ and $\varphi[y/x]$ are **not** equivalent.

Example:

$$P(x, z) \quad \text{is not equivalent to} \quad P(y, z)$$

Renaming of Bound Variables

With the operation of substitution, the renaming of bound variables can be described by the following equivalences.

When $y \notin \text{free}(\forall x \varphi)$:

$$\forall x \varphi \Leftrightarrow \forall y (\varphi[y/x])$$

When $y \notin \text{free}(\exists x \varphi)$:

$$\exists x \varphi \Leftrightarrow \exists y (\varphi[y/x])$$

Example:

$$\exists x \forall y P(x, y) \Leftrightarrow \exists x \forall z P(x, z) \Leftrightarrow \exists y \forall z P(y, z) \Leftrightarrow \exists y \forall x P(y, x)$$

Definition

Conclusion ψ **logically follows** from assumptions $\varphi_1, \varphi_2, \dots, \varphi_n$, which is denoted

$$\varphi_1, \varphi_2, \dots, \varphi_n \models \psi,$$

if in every interpretation $\mathcal{A}$ and valuation v where assumption $\varphi_1, \varphi_2, \dots, \varphi_n$ are true, also the conclusion ψ is true.

- All, what was said about the logical entailment in propositional logic, holds all analogously in predicate logic.

Logical Entailment

If we want to show that a given conclusion ψ **does not** follow from assumptions $\varphi_1, \varphi_2, \dots, \varphi_n$, it is sufficient to find an example of one particular interpretation $\mathcal{A}$ and valuation v , where the assumptions are true and the conclusion ψ is false.

Example:

- *There exists an aquatic animal, which is meat-eating.*
 - *All fish are aquatic animals.*
-
- *There exists a meat-eating fish.*

$$\exists x(P(x) \wedge Q(x))$$

$$\forall x(R(x) \rightarrow P(x))$$

$$\hline \exists x(R(x) \wedge Q(x))$$

$$P(x) \text{ — “}x \text{ is an aquatic animal”}$$

$$Q(x) \text{ — “}x \text{ is meat-eating”}$$

$$R(x) \text{ — “}x \text{ is a fish”}$$

An interpretation $\mathcal{A}$ with universe $A = \{a, b\}$

$$P^{\mathcal{A}} = \{a, b\}$$

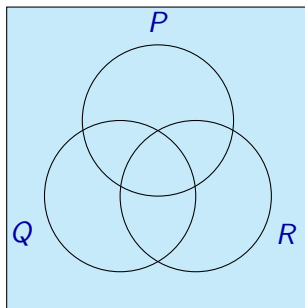
$$Q^{\mathcal{A}} = \{a\}$$

$$R^{\mathcal{A}} = \{b\}$$

Venn Diagrams

In general, it is difficult to find out whether a conclusion does or does not follow from given assumptions.

In cases when we have only unary predicates and there is only a small number of them (e.g., 3), we use so called **Venn diagrams** as an aid for the reasoning.



Venn Diagrams

Example:

- *Fish are vertebrates.*
 - *Fish live in water.*
 - *There exists at least one fish.*
-
- *There exists a vertebrate living in water.*

$$\forall x(P(x) \rightarrow Q(x))$$

$$\forall x(P(x) \rightarrow R(x))$$

$$\exists xP(x)$$

$$\exists x(Q(x) \wedge R(x))$$

$$P(x) \text{ — “}x \text{ is a fish”}$$

$$Q(x) \text{ — “}x \text{ is a vertebrate”}$$

$$R(x) \text{ — “}x \text{ lives in water”}$$

⟨a solution on a whiteboard⟩

An Example of a Proof

$$\frac{\begin{array}{l} \forall x(\neg R(x, x)) \\ \forall x\forall y\forall z(R(x, y) \wedge R(y, z) \rightarrow R(x, z)) \end{array}}{\forall x\forall y(R(x, y) \rightarrow \neg R(y, x))}$$

1. $\forall x(\neg R(x, x))$ - assumption 1
2. $\forall x\forall y\forall z(R(x, y) \wedge R(y, z) \rightarrow R(x, z))$ - assumption 2
3. Lets assume arbitrary elements x and y :
4. Lets assume $R(x, y)$:
5. Lets assume $R(y, x)$:
6. $R(x, y) \wedge R(y, x) \rightarrow R(x, x)$ - from 2.
7. $R(x, x)$ - from 4., 5., 6.
8. $\neg R(x, x)$ - from 1.
9. $\neg R(y, x)$ - contradiction of 7. and 8.,
so 5. does not hold
10. $R(x, y) \rightarrow \neg R(y, x)$ - from 4., 9.
11. $\forall x\forall y(R(x, y) \rightarrow \neg R(y, x))$ - from 3., 10.

Equality

One of the most important relations is **equality** (**identity**).

Elements x and y are equal, written

$$x = y,$$

if they are the same element.

Equality can be expressed as a predicate, e.g., we can choose that $P(x, y)$ represents proposition “ x and y are equal”.

But $P(x, y)$ can be true in some interpretations even when x and y are distinct elements, and in some interpretations, $P(x, x)$ can be false for some element x .

Example: We would like to describe relationship “ x is a sibling of y ” using a binary predicate R , where $R(x, y)$ means “ x is a parent of y ”.

An attempt for a possible solution:

$$\begin{aligned} &\text{“}x \text{ is a sibling of } y\text{”} \\ &\text{iff} \\ &\exists z (R(z, x) \wedge R(z, y)) \end{aligned}$$

Problem: If for a given x there is an element z such that $R(z, x)$, then it is true that

$$\exists z (R(z, x) \wedge R(z, x)),$$

and so it is also true that “ x is a sibling of x ”.

Equality

Alphabet:

- ...
- Symbol for equality: “=”
- ...

Atomic formulas (cont.)

- ...
- If x and y are variables then $x = y$ is a well-formed atomic formula.
- ...

Symbol “=” is interpreted as equality in every interpretation, i.e., in every interpretation $\mathcal{A}$ and valuation v is:

- $\mathcal{A}, v \models x = y$ iff $v(x) = v(y)$.

Example: The relationship “ x is a sibling of y ” can be expressed by formula

$$\neg(x = y) \wedge \exists z(R(z, x) \wedge R(z, y)),$$

where $R(x, y)$ means that “ x is a parent of y ”.

Remark: The notation $x \neq y$ is often used instead of $\neg(x = y)$.

- “There exists **exactly one** x such that $P(x)$ ”:

$$\exists x(P(x) \wedge \forall y(P(y) \rightarrow x = y))$$

- “There exist **at least two** elements x such that $P(x)$ ”:

$$\exists x \exists y(P(x) \wedge P(y) \wedge \neg(x = y))$$

- “There exist **exactly two** elements x such that $P(x)$ ”:

$$\exists x \exists y(P(x) \wedge P(y) \wedge \neg(x = y) \wedge \forall z(P(z) \rightarrow (z = x \vee z = y)))$$

- “There exists **exactly one** x , for which φ holds”:

$$\exists x(\varphi \wedge \forall y(\varphi[y/x] \rightarrow x = y))$$

Sometimes we want to talk about some particular element of the universe.

Example: *“There exists at least one x such that John Smith is a parent of x and x is a woman.”* (i.e., *“John Smith has at least one daughter.”*)

If the value assigned to variable y is “John Smith”:

$$\exists x(R(y, x) \wedge S(x))$$

- $R(x, y)$ — “ x is a parent of y ”
- $S(x)$ — “ x is a woman”

We could introduce an unary predicate N representing property “to be John Smith”:

$$\forall y(N(y) \rightarrow \exists x(R(y, x) \wedge S(x)))$$

If we have some unary predicate N where we are interested only in those interpretations where exists **exactly one** element x , for which $N(x)$ holds, it would be convenient to have some way to name this element and refer to it directly instead of using of the predicate N .

Constant symbols (**constants**) can be used for this purpose.

Alphabet:

- ...
- constant symbols: " a ", " b ", " c ", " d ", ...
- ...

In **atomic formulas**, constants can occur at the same places as variables:

$$P(c, x)$$

$$Q(d)$$

$$R(a, a)$$

$$x = a$$

- Constants **must not** be used in quantifiers — e.g., $\exists c P(x, c)$ **is not** a well-formed formula.

Values assigned to constant symbols are determined by a given **interpretation**:

- A given interpretation $\mathcal{A}$ (with universe A) assigns to every constant symbol c some element of the universe A .
This element is denoted $c^{\mathcal{A}}$. So $c^{\mathcal{A}} \in A$.

Example: *“There exists at least one x such that John Smith is a parent of x and x is a woman.”*

$$\exists x(R(a, x) \wedge S(x))$$

- $R(x, y)$ — “ x is a parent of y ”
- $S(x)$ — “ x is a woman”
- a — constant symbol representing “John Smith”

Example: *“Every prime is greater than one.”*

$$\forall x(P(x) \rightarrow R(x, e))$$

- $P(x)$ — “ x is a prime”
- $R(x, y)$ — “ x is greater than y ”
- e — constant symbol representing value 1

Functions

A binary relation R is a (unary) **function** if for each x there exists at most one y such that

$$(x, y) \in R.$$

This function is **total** if for each x there exists exactly one such y .

Example: Binary relation R on the set of natural numbers $\mathbb{N}$ where

$$(x, y) \in R \quad \text{iff} \quad y = x + 1$$

We have

$$R = \{(0, 1), (1, 2), (2, 3), (3, 4), (4, 5), \dots\}$$

Similarly, a ternary relation T is a (binary) function if for every pair of elements x_1 and x_2 there exists at most one (resp., exactly one for total function) y such that

$$(x_1, x_2, y) \in T.$$

Example: Addition on the set of real numbers $\mathbb{R}$ can be viewed as a ternary relation S (i.e., as a set of triples of real numbers) where

$$(x_1, x_2, y) \in S \quad \text{iff} \quad x_1 + x_2 = y$$

In predicate logic, functions can be expressed using predicates representing the corresponding relations — this is not very straightforward nor convenient.

Example: *“For each x and y it holds that $x + y \geq y + x$.”*

$$\forall x \forall y \exists z \exists w (S(x, y, z) \wedge S(y, x, w) \wedge P(z, w))$$

- $S(x, y, z)$ — *“ z is the sum of values x and y ”*
- $P(x, y)$ — *“ x greater than or equal to y ”*

Remark: Moreover, we must assume that for every pair of elements x and y there exists exactly one element z such that $S(x, y, z)$.

In predicate logic, functions can be represented by **function symbols**.

Alphabet:

- ...
- function symbols: " f ", " g ", " h ", ...
- ...

Every function symbol must have a specified **arity** corresponding to the arity of a function represented by this symbol (i.e., the number of arguments of this function).

Terms — expressions, consisting of variables, constant symbols, and function symbols; values of terms are elements of the universe

Example:

- Let us say that we have a predicate F where we assume that for every x there exists exactly one y such that

$$F(x, y).$$

Instead of binary predicate F , we can use unary function symbol f .

Term

$$f(x)$$

represents this one particular element y , for which $F(x, y)$ holds.

Instead of $\exists y(F(x, y) \wedge P(y))$, we can write $P(f(x))$.

Example:

- Let us say that we have a ternary predicate G where we assume that for every pair of elements x_1 and x_2 there exists exactly one y such that

$$G(x_1, x_2, y).$$

Instead of ternary predicate G , we can use binary function symbol g .

Term

$$g(x_1, x_2)$$

represents this one particular element y , for which $G(x_1, x_2, y)$ holds.

Example: *“For each x and y it holds that $x + y \geq y + x$.”*

$$\forall x \forall y P(f(x, y), f(y, x))$$

- f — binary function symbol where $f(x, y)$ represents the sum of values x and y
- P — binary predicate symbol where $P(x, y)$ represents relation “ x is greater than or equal to y ”

Terms

Variables, constant symbols and function symbols can be composed in terms in arbitrary way — it is only necessary to comply with the arity of the symbols (to apply each function symbol to a correct number of arguments).

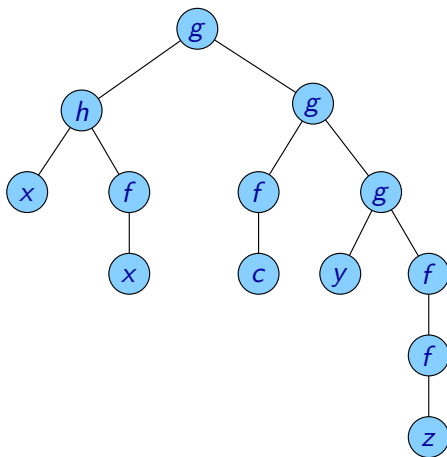
Example:

- c — constant symbol
- f — unary function symbol
- g — binary function symbol
- h — binary function symbol

Examples of terms:

 x $f(y)$ $g(c, x)$ $g(h(x, x), f(c))$ $g(h(x, f(x)), g(f(c), g(y, f(f(z)))))$

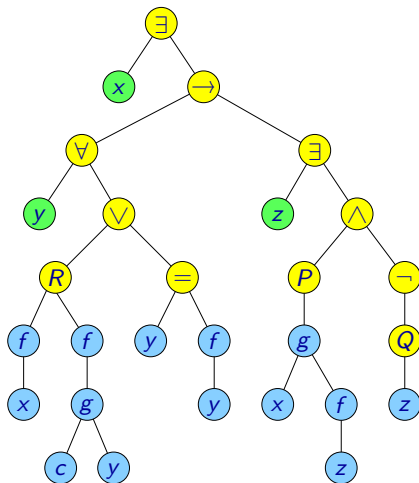
The syntactic tree of term $g(h(x, f(x)), g(f(c), g(y, f(f(z))))))$



Terms in Formulas

The syntactic tree of formula

$$\exists x(\forall y(R(f(x), f(g(c, y))) \vee y = f(y)) \rightarrow \exists z(P(g(x, f(z))) \wedge \neg Q(z)))$$



Example:

- For each x , y , and z it holds that $(x + y) + z = x + (y + z)$:

$$\forall x \forall y \forall z (f(f(x, y), z) = f(x, f(y, z)))$$

- For each x it holds that $x + 0 = x$ and $0 + x = x$:

$$\forall x (f(x, e) = x \wedge f(e, x) = x)$$

- For each x there exists y such that $x + y = 0$:

$$\forall x \exists y (f(x, y) = e)$$

Constant and function symbols:

- f — binary function symbol representing “*addition*” (operation “ $+$ ”)
- e — constant symbol representing element “ 0 ”

Example:

- For each x , y , and z it holds that $x \cdot (y + z) = x \cdot y + x \cdot z$:

$$\forall x \forall y \forall z (g(x, f(y, z)) = f(g(x, y), g(x, z)))$$

- For each x and y such that $x \leq y$ it holds that $x + z \leq y + z$:

$$\forall x \forall y (R(x, y) \rightarrow \forall z R(f(x, y), f(y, z)))$$

Constant and function symbols:

- f — binary function symbol representing “*addition*” (operation “ $+$ ”)
- g — binary function symbol representing “*multiplication*” (operation “ $\cdot$ ”)
- R — binary predicate symbol representing relation “*less than or equal to*” (relation “ $\leq$ ”)

Syntax of Formulas of Predicate Logic

Alphabet:

- **logical connectives** — “ $\neg$ ”, “ $\wedge$ ”, “ $\vee$ ”, “ $\rightarrow$ ”, “ $\leftrightarrow$ ”
- **quantifiers** — “ $\forall$ ” and “ $\exists$ ”
- **equality** — “ $=$ ”
- **auxiliary symbols** — “(”, “)”, and “,”
- **variables** — “ x ”, “ y ”, “ z ”, $\dots$, “ x_0 ”, “ x_1 ”, “ x_2 ”, $\dots$
- **predicate symbols** — for example symbols “ P ”, “ Q ”, “ R ”, etc. (for each symbols, there must be specified its arity)
- **function symbols** — for example symbols “ f ”, “ g ”, “ h ”, etc. (for each symbol, there must be specified its arity)
- **constant symbols** — for example symbols “ a ”, “ b ”, “ c ”, etc.

Remark: Constant symbols can be viewed as function symbols of arity 0.

Definition

Well-formed **terms** are defined as follows:

- 1 If x is a variable then x is a well-formed term.
- 2 If c is a constant symbol then c is a well-formed term.
- 3 If f is a function symbol of arity n and $t_1, t_2, \dots, t_n$ are well-formed terms then

$$f(t_1, t_2, \dots, t_n)$$

is a well-formed term.

- 4 There are no other well-formed terms than those constructed according to the previous rules.

Definition

Well-formed **atomic formulas** are defined as follows:

- 1 If P is a predicate symbol of arity n and $t_1, t_2, \dots, t_n$ are well-formed terms then

$$P(t_1, t_2, \dots, t_n)$$

is a well-formed atomic formula.

- 2 If t_1 and t_2 are well-formed terms then

$$t_1 = t_2$$

is a well-formed atomic formula.

- 3 There are no other well-formed atomic formulas than those constructed according to the previous rules.

Syntax of Formulas of Predicate Logic

Definition (a previously stated definition repeated)

Well-formed **formulas of predicate logic** are sequences of symbols constructed according to the following rules:

- 1 Well-formed atomic formulas are well-formed formulas.
- 2 If φ and ψ are well-formed formulas, then also $(\neg\varphi)$, $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\varphi \rightarrow \psi)$ and $(\varphi \leftrightarrow \psi)$ are well-formed formulas.
- 3 If φ is a well-formed formula and x is a variable, then $\forall x\varphi$ and $\exists x\varphi$ are well-formed formulas.
- 4 There are no other well-formed formulas than those constructed according to the previous rules.

Interpretation $\mathcal{A}$:

- universe A
- to every predicate symbol P of arity n , an n -ary relation $P^{\mathcal{A}}$ is assigned, where $P^{\mathcal{A}} \subseteq A \times A \times \dots \times A$
- to every function symbol f of arity n , an n -ary function $f^{\mathcal{A}}$ is assigned, where $f^{\mathcal{A}} : A \times A \times \dots \times A \rightarrow A$
- to every constant symbol c , an element of the universe $c^{\mathcal{A}}$ is assigned, i.e., $c^{\mathcal{A}} \in A$

Remark: In interpretations, only **total** functions, i.e., functions whose values are defined for all possible values of arguments, are assigned to function symbols.

The **value of a term** in interpretation $\mathcal{A}$ and valuation v :

- Term x , where x is a variable — the value of this term is an element $a \in A$ such that $v(x) = a$.
- Term c , where c is a constant symbol — the value of this term is the element $c^{\mathcal{A}} \in A$.
- Term $f(t_1, t_2, \dots, t_n)$, where f is a function symbol with arity n and $t_1, t_2, \dots, t_n$ are terms — the value of this term is the element $b \in A$ such that

$$b = f^{\mathcal{A}}(a_1, a_2, \dots, a_n),$$

where $a_1, a_2, \dots, a_n$ are values of terms $t_1, t_2, \dots, t_n$ in interpretation $\mathcal{A}$ and valuation v .

Semantics of Predicate Logic

Example: Interpretation $\mathcal{A}$ where the universe is the set of natural numbers $\mathbb{N} = \{0, 1, 2, \dots\}$.

- $a^{\mathcal{A}} = 0$
- $f^{\mathcal{A}}$ is the function “successor”, i.e., $f^{\mathcal{A}}(x) = x + 1$
- $g^{\mathcal{A}}$ is the function “sum”, i.e., $g^{\mathcal{A}}(x, y) = x + y$

Valuation v where $v(x) = 5$, $v(y) = 13$, $v(z) = 2$, ...

Values of terms in interpretation $\mathcal{A}$ and valuation v :

- Term x — value 5
- Term a — value 0
- Term $f(a)$ — value 1 ($0 + 1 = 1$)
- Term $f(f(a))$ — value 2 ($1 + 1 = 2$)
- Term $g(x, f(f(a)))$ — value 7 ($5 + 2 = 7$)
- Term $g(z, y)$ — value 15 ($2 + 13 = 15$)
- Term $f(g(z, y))$ — value 16 ($15 + 1 = 16$)

Truth values of atomic formulas in interpretation $\mathcal{A}$ and valuation v :

- $\mathcal{A}, v \models P(t_1, t_2, \dots, t_n)$, where P is a predicate symbol of arity n and where $t_1, t_2, \dots, t_n$ are terms, holds iff

$$(a_1, a_2, \dots, a_n) \in P^{\mathcal{A}},$$

where $a_1, a_2, \dots, a_n$ are values of terms $t_1, t_2, \dots, t_n$ in interpretation $\mathcal{A}$ and valuation v .

- $\mathcal{A}, v \models t_1 = t_2$, where t_1 and t_2 are terms, holds iff

$$a_1 = a_2,$$

where a_1 and a_2 are values of terms t_1 and t_2 in interpretation $\mathcal{A}$ and valuation v .

Example: Interpretation $\mathcal{A}$ where the universe is the set of natural numbers $\mathbb{N} = \{0, 1, 2, \dots\}$.

- $f^{\mathcal{A}}$ is the function “successor”, i.e., $f^{\mathcal{A}}(x) = x + 1$
- $g^{\mathcal{A}}$ is the function “sum”, i.e., $g^{\mathcal{A}}(x, y) = x + y$
- $P^{\mathcal{A}}$ is the set of all primes
- $Q^{\mathcal{A}}$ is the binary relation “ $<$ ” i.e., $(x, y) \in Q^{\mathcal{A}}$ iff $x < y$

Valuation v where $v(x) = 5$, $v(y) = 13$, $v(z) = 2$, $\dots$

- $\mathcal{A}, v \models P(x)$ (5 is a prime)
- $\mathcal{A}, v \not\models Q(y, z)$ (it is not the case that $13 < 2$)
- $\mathcal{A}, v \models Q(f(f(z)), g(x, y))$ ($((2 + 1) + 1 < 5 + 13)$)
- $\mathcal{A}, v \not\models P(f(g(z, x)))$ ($((2 + 5) + 1 = 8$ and 8 is not a prime)

- The logic described here is **the first order** predicate logic — it is possible to quantify only over the elements of the universe (in the second order predicate logic, it is possible to quantify over relations).

Predicate Logic — Additional Comments

As commonly used in mathematics, it is often the case that formulas are not written according to the precise syntax of predicate logic but many kinds of conventions and abbreviations are used.

- For **binary** function and predicate symbols, it is common to use **infix** notation:

For example, $f(x, y)$ and $R(x, y)$ can be written as

$$x \ f \ y$$

$$x \ R \ y$$

- To denote predicate, function and constant symbols, many different kinds of symbols are used:

For example, $R(f(x, y), g(z))$ can be written as

$$x + \delta \leq |\varepsilon|$$

or for example as

$$x \circ y \sqsupset G(z)$$

Predicate Logic — Additional Comments

Examples of formulas representing propositions from set theory:

- x is an elements of set A :

$$x \in A$$

“ $\in$ ” — binary predicate symbol representing the relation *“to be an element of”*

“ x ”, “ A ” — variables

If we would do the following changes:

- instead of symbol “ $\in$ ”, we would use binary predicate symbol E ,
- instead of variable A we would use variable y ,

then the formula would look as follows:

$$E(x, y)$$

- Two sets are equal iff they contain the same elements:

$$A = B \leftrightarrow \forall x(x \in A \leftrightarrow x \in B)$$

If we use predicate E instead of “ $\in$ ”, and y and z instead of A and B , the formula would look as follows:

$$y = z \leftrightarrow \forall x(E(x, y) \leftrightarrow E(x, z))$$

- The definition of relation “*to be a subset*” (denoted by symbol “ $\subseteq$ ”):

$$A \subseteq B \leftrightarrow \forall x(x \in A \rightarrow x \in B)$$

If we use binary predicate symbol S instead of “ $\subseteq$ ”:

$$S(y, z) \leftrightarrow \forall x(E(x, y) \rightarrow E(x, z))$$

- The definition of operation “*union*” (denoted by symbol “ $\cup$ ”):

$$\forall x (x \in A \cup B \leftrightarrow (x \in A \vee x \in B))$$

If we use binary function symbol f instead of “ $\cup$ ”:

$$\forall x (E(x, f(y, z)) \leftrightarrow (E(x, y) \vee E(x, z)))$$

- Sometimes

$$\exists x(x \in A \wedge \dots)$$

is written in an abbreviated way as

$$(\exists x \in A)(\dots)$$

I.e., instead of

“there exists x such that $x \in A$ and $\dots$ ”

we can say

“there exists $x \in A$ such that $\dots$ ”

- Similarly, $\exists x(x \geq 1 \wedge \dots)$ can be written in an abbreviated way as

$$(\exists x \geq 1)(\dots)$$

- Sometimes

$$\forall x(x \in A \rightarrow \dots)$$

is written in an abbreviated way as

$$(\forall x \in A)(\dots)$$

I.e., instead of

“for each x , for which $x \in A$ holds, we have $\dots$ ”

we can say

“for each $x \in A$ we have $\dots$ ”

- Similarly, $\forall x(x \geq 1 \rightarrow \dots)$ can be written in an abbreviated way as

$$(\forall x \geq 1)(\dots)$$

Formal Languages

Alphabet and Word

Definition

Alphabet is a nonempty finite set of **symbols**.

Remark: An alphabet is often denoted by the symbol Σ (upper case sigma) of the Greek alphabet.

Definition

A **word** over a given alphabet is a finite sequence of symbols from this alphabet.

Example 1:

$\Sigma = \{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z\}$

Words over alphabet Σ : HELLO ABRACADABRA ERROR

Alphabet and Word

Example 2:

$\Sigma_2 = \{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, _ \}$

A word over alphabet Σ_2 : HELLO_WORLD

Example 3:

$\Sigma_3 = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Words over alphabet Σ_3 : 0, 31415926536, 65536

Example 4:

Words over alphabet $\Sigma_4 = \{0, 1\}$: 011010001, 111, 1010101010101010

Example 5:

Words over alphabet $\Sigma_5 = \{a, b\}$: *aababb*, *abbabbba*, *aaab*

Alphabet and Word

Example 6:

Alphabet Σ_6 is the set of all ASCII characters.

Example of a word:

```
class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello, world!");  
    }  
}
```

```
class HelloWorld {  $\hookleftarrow$  public static void main(Str...
```

Language — a set of (some) words of symbols from a given alphabet

Examples of problem types, where theory of formal languages is useful:

- Construction of compilers:
 - Lexical analysis
 - Syntactic analysis
- Searching in text:
 - Searching for a given text pattern
 - Searching for a part of text specified by a regular expression

To describe a language, there are several possibilities:

- We can enumerate all words of the language (however, this is possible only for small finite languages).

Example: $L = \{aab, babba, aaaaaa\}$

- We can specify a property of the words of the language:

Example: The language over alphabet $\{0, 1\}$ containing all words with even number of occurrences of symbol 1.

In particular, the following two approaches are used in the theory of formal languages:

- To describe an (idealized) machine, device, algorithm, that recognizes words of the given language – approaches based on **automata**.
- To describe some mechanism that allows to generate all words of the given language – approaches based on **grammars** or **regular expressions**.

Some Basic Concepts

The **length of a word** is the number of symbols of the word.

For example, the length of word *abaab* is 5.

The length of a word w is denoted $|w|$.

For example, if $w = \textit{abaab}$ then $|w| = 5$.

We denote the number of occurrences of a symbol a in a word w by $|w|_a$.

For word $w = \textit{ababb}$ we have $|w|_a = 2$ and $|w|_b = 3$.

An **empty word** is a word of length 0, i.e., the word containing no symbols.

The empty word is denoted by the letter ε (epsilon) of the Greek alphabet. (Remark: In literature, sometimes λ (lambda) is used to denote the empty word instead of ε .)

$$|\varepsilon| = 0$$

Concatenation of Words

One of operations we can do on words is the operation of **concatenation**: For example, the concatenation of words **OST** and **RAVA** is the word **OSTRAVA**.

The operation of concatenation is denoted by symbol $\cdot$ (similarly to multiplication). It is possible to omit this symbol.

$$\text{OST} \cdot \text{RAVA} = \text{OSTRAVA}$$

Concatenation is **associative**, i.e., for every three words u , v , and w we have

$$(u \cdot v) \cdot w = u \cdot (v \cdot w)$$

which means that we can omit parenthesis when we write multiple concatenations. For example, we can write $w_1 \cdot w_2 \cdot w_3 \cdot w_4 \cdot w_5$ instead of $(w_1 \cdot (w_2 \cdot w_3)) \cdot (w_4 \cdot w_5)$.

Concatenation of Words

Concatenation is not **commutative**, i.e., the following equality does not hold in general

$$u \cdot v = v \cdot u$$

Example:

$$\text{OST} \cdot \text{RAVA} \neq \text{RAVA} \cdot \text{OST}$$

It is obvious that the following holds for any words v and w :

$$|v \cdot w| = |v| + |w|$$

For every word w we also have:

$$\varepsilon \cdot w = w \cdot \varepsilon = w$$

Definition

A word x is a **prefix** of a word y , if there exists a word v such that $y = xv$.

A word x is a **suffix** of a word y , if there exists a word u such that $y = ux$.

A word x is a **subword** of a word y , if there exist words u and v such that $y = uxv$.

Example:

- Prefixes of the word **abaab** are ε , **a**, **ab**, **aba**, **abaa**, **abaab**.
- Suffixes of the word **abaab** are ε , **b**, **ab**, **aab**, **baab**, **abaab**.
- Subwords of the word **abaab** are ε , **a**, **b**, **ab**, **ba**, **aa**, **aba**, **baa**, **aab**, **abaa**, **baab**, **abaab**.

The set of all words over alphabet Σ is denoted Σ^* .

Definition

A **(formal) language** L over an alphabet Σ is a subset of Σ^* , i.e., $L \subseteq \Sigma^*$.

Example 1: The set $\{00, 01001, 1101\}$ is a language over alphabet $\{0, 1\}$.

Example 2: The set of all syntactically correct programs in the C programming language is a language over the alphabet consisting of all ASCII characters.

Example 3: The set of all texts containing the sequence `hello` is a language over alphabet consisting of all ASCII characters.

Set Operations on Languages

Since languages are sets, we can apply any set operations to them:

Union – $L_1 \cup L_2$ is the language consisting of the words belonging to language L_1 or to language L_2 (or to both of them).

Intersection – $L_1 \cap L_2$ is the language consisting of the words belonging to language L_1 and also to language L_2 .

Complement – $\overline{L_1}$ is the language containing those words from Σ^* that do not belong to L_1 .

Difference – $L_1 - L_2$ is the language containing those words of L_1 that do not belong to L_2 .

Remark: It is assumed the languages involved in these operations use the same alphabet Σ .

Set Operations on Languages

Formally:

Union: $L_1 \cup L_2 = \{w \in \Sigma^* \mid w \in L_1 \vee w \in L_2\}$

Intersection: $L_1 \cap L_2 = \{w \in \Sigma^* \mid w \in L_1 \wedge w \in L_2\}$

Complement: $\overline{L_1} = \{w \in \Sigma^* \mid w \notin L_1\}$

Difference: $L_1 - L_2 = \{w \in \Sigma^* \mid w \in L_1 \wedge w \notin L_2\}$

Remark: We assume that $L_1, L_2 \subseteq \Sigma^*$ for some given alphabet Σ .

Set Operations on Languages

Example:

Consider languages over alphabet $\{a, b\}$.

- L_1 — the set of all words containing subword baa
- L_2 — the set of all words with an even number of occurrences of symbol b

Then

- $L_1 \cup L_2$ — the set of all words containing subword baa or an even number of occurrences of b
- $L_1 \cap L_2$ — the set of all words containing subword baa and an even number of occurrences of b
- $\overline{L_1}$ — the set of all words that do not contain subword baa
- $L_1 - L_2$ — the set of all words that contain subword baa but do not contain an even number of occurrences of b

Concatenation of Languages

Definition

Concatenation of languages L_1 and L_2 , where $L_1, L_2 \subseteq \Sigma^*$, is the language $L \subseteq \Sigma^*$ such that for each $w \in \Sigma^*$ it holds that

$$w \in L \leftrightarrow (\exists u \in L_1)(\exists v \in L_2)(w = u \cdot v)$$

The concatenation of languages L_1 and L_2 is denoted $L_1 \cdot L_2$.

Example:

$$\begin{aligned} L_1 &= \{abb, ba\} \\ L_2 &= \{a, ab, bbb\} \end{aligned}$$

The language $L_1 \cdot L_2$ contains the following words:

abba abbab abbbbb baa baab babbb

Iteration of a Language

Definition

The **iteration (Kleene star) of language** L , denoted L^* , is the language consisting of words created by concatenation of some arbitrary number of words from language L .

I.e. $w \in L^*$ iff

$$\exists n \in \mathbb{N} : \exists w_1, w_2, \dots, w_n \in L : w = w_1 w_2 \cdots w_n$$

Example: $L = \{aa, b\}$

$$L^* = \{\varepsilon, aa, b, aaaa, aab, baa, bb, aaaaaa, aaaab, aabaa, aabb, \dots\}$$

Remark: The number of concatenated words can be 0, which means that $\varepsilon \in L^*$ always holds (it does not matter if $\varepsilon \in L$ or not).

Iteration of a Language – Alternative Definition

At first, for a language L and a number $k \in \mathbb{N}$ we define the language L^k :

$$L^0 = \{\varepsilon\}, \quad L^k = L^{k-1} \cdot L \quad \text{for } k \geq 1$$

This means

$$\begin{aligned} L^0 &= \{\varepsilon\} \\ L^1 &= L \\ L^2 &= L \cdot L \\ L^3 &= L \cdot L \cdot L \\ L^4 &= L \cdot L \cdot L \cdot L \\ L^5 &= L \cdot L \cdot L \cdot L \cdot L \\ &\dots \end{aligned}$$

Example: For $L = \{aa, b\}$, the language L^3 contains the following words:

aaaaaa aaaab aabaa aabb baaaa baab bbaa bbb

Alternative definition

The **iteration (Kleene star) of language** L is the language

$$L^* = \bigcup_{k \geq 0} L^k$$

Remark:

$$\bigcup_{k \geq 0} L^k = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \dots$$

Remark: Sometimes, notation L^+ is used as an abbreviation for $L \cdot L^*$, i.e.,

$$L^+ = \bigcup_{k \geq 1} L^k$$

The **reverse** of a word w is the word w written from backwards (in the opposite order).

The reverse of a word w is denoted w^R .

Example: $w = \text{HELLO}$ $w^R = \text{OLLEH}$

Formally, for $w = a_1 a_2 \cdots a_n$ (where $a_i \in \Sigma$) is $w^R = a_n a_{n-1} \cdots a_1$.

The **reverse** of a language L is the language consisting of reverses of all words of L .

Reverse of a language L is denoted L^R .

$$L^R = \{w^R \mid w \in L\}$$

Example: $L = \{ab, baaba, aaab\}$
 $L^R = \{ba, abaab, baaa\}$

Order on Words

Let us assume some (linear) order $<$ on the symbols of alphabet Σ , i.e., if $\Sigma = \{a_1, a_2, \dots, a_n\}$ then

$$a_1 < a_2 < \dots < a_n.$$

Example: $\Sigma = \{a, b, c\}$ with $a < b < c$.

The following (linear) order $<_L$ can be defined on Σ^* :

$x <_L y$ iff:

- $|x| < |y|$, or
- $|x| = |y|$ there exist words $u, v, w \in \Sigma^*$ and symbols $a, b \in \Sigma$ such that

$$x = uav \quad y = ubw \quad a < b$$

Informally, we can say that in order $<_L$ we order words according to their length, and in case of the same length we order them lexicographically.

Order on Words

All words over alphabet Σ can be ordered by $<_L$ into a sequence

$$w_0, w_1, w_2, \dots$$

where every word $w \in \Sigma^*$ occurs exactly once, and where for each $i, j \in \mathbb{N}$ it holds that $w_i <_L w_j$ iff $i < j$.

Example: For alphabet $\Sigma = \{a, b, c\}$ (where $a < b < c$) , the initial part of the sequence looks as follows:

$$\varepsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, aab, aac, aba, abb, abc, \dots$$

For example, when we talk about the first ten words of a language $L \subseteq \Sigma^*$, we mean ten words that belong to language L and that are smallest of all words of L according to order $<_L$.

Regular Expressions

Regular Expressions

Regular expressions describing languages over an alphabet Σ :

- $\emptyset$, ε , a (where $a \in \Sigma$) are regular expressions:

$\emptyset$... denotes the empty language

ε ... denotes the language $\{\varepsilon\}$

a ... denotes the language $\{a\}$

- If α , β are regular expressions then also $(\alpha + \beta)$, $(\alpha \cdot \beta)$, (α^*) are regular expressions:

$(\alpha + \beta)$... denotes the union of languages denoted α and β

$(\alpha \cdot \beta)$... denotes the concatenation of languages denoted α
and β

(α^*) ... denotes the iteration of a language denoted α

- There are no other regular expressions except those defined in the two points mentioned above.

Regular Expressions

Example: alphabet $\Sigma = \{0, 1\}$

- According to the definition, 0 and 1 are regular expressions.

Regular Expressions

Example: alphabet $\Sigma = \{0, 1\}$

- According to the definition, 0 and 1 are regular expressions.
- Since 0 and 1 are regular expression, $(0 + 1)$ is also a regular expression.

Regular Expressions

Example: alphabet $\Sigma = \{0, 1\}$

- According to the definition, 0 and 1 are regular expressions.
- Since 0 and 1 are regular expression, $(0 + 1)$ is also a regular expression.
- Since 0 is a regular expression, (0^*) is also a regular expression.

Regular Expressions

Example: alphabet $\Sigma = \{0, 1\}$

- According to the definition, 0 and 1 are regular expressions.
- Since 0 and 1 are regular expression, $(0 + 1)$ is also a regular expression.
- Since 0 is a regular expression, (0^*) is also a regular expression.
- Since $(0 + 1)$ and (0^*) are regular expressions, $((0 + 1) \cdot (0^*))$ is also a regular expression.

Regular Expressions

Example: alphabet $\Sigma = \{0, 1\}$

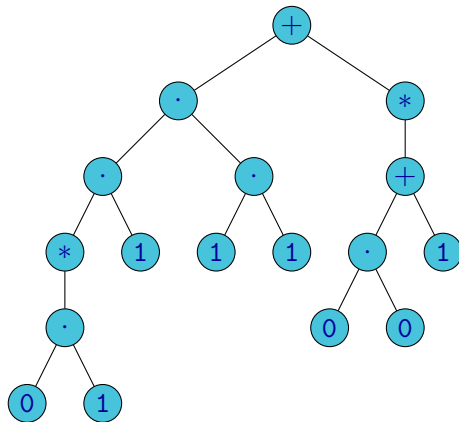
- According to the definition, 0 and 1 are regular expressions.
- Since 0 and 1 are regular expression, $(0 + 1)$ is also a regular expression.
- Since 0 is a regular expression, (0^*) is also a regular expression.
- Since $(0 + 1)$ and (0^*) are regular expressions, $((0 + 1) \cdot (0^*))$ is also a regular expression.

Remark: If α is a regular expression, by $[\alpha]$ we denote the language defined by the regular expression α .

$$[((0 + 1) \cdot (0^*))] = \{0, 1, 00, 10, 000, 100, 0000, 1000, 00000, \dots\}$$

Regular Expressions

The structure of a regular expression can be represented by an abstract syntax tree:



$(((((0 \cdot 1)^*) \cdot 1) \cdot (1 \cdot 1)) + (((0 \cdot 0) + 1)^*))$

The formal definition of semantics of regular expressions:

- $[\emptyset] = \emptyset$
- $[\varepsilon] = \{\varepsilon\}$
- $[a] = \{a\}$
- $[\alpha^*] = [\alpha]^*$
- $[\alpha \cdot \beta] = [\alpha] \cdot [\beta]$
- $[\alpha + \beta] = [\alpha] \cup [\beta]$

Regular Expressions

To make regular expressions more lucid and succinct, we use the following conventions:

- The outward pair of parentheses can be omitted.
- We can omit parentheses that are superfluous due to associativity of operations of union ($+$) and concatenation ($\cdot$).
- We can omit parentheses that are superfluous due to the defined priority of operators (iteration ($*$) has the highest priority, concatenation ($\cdot$) has lower priority, and union ($+$) has the lowest priority).
- A dot denoting concatenation can be omitted.

Example: Instead of

$$((((((0 \cdot 1)^*) \cdot 1) \cdot (1 \cdot 1)) + (((0 \cdot 0) + 1)^*))$$

we usually write

$$(01)^*111 + (00 + 1)^*$$

Regular Expressions

Examples: In all examples $\Sigma = \{0, 1\}$.

0 ... the language containing the only word 0

Regular Expressions

Examples: In all examples $\Sigma = \{0, 1\}$.

0 ... the language containing the only word 0

01 ... the language containing the only word 01

Regular Expressions

Examples: In all examples $\Sigma = \{0, 1\}$.

0 ... the language containing the only word 0

01 ... the language containing the only word 01

0 + 1 ... the language containing two words 0 and 1

Regular Expressions

Examples: In all examples $\Sigma = \{0, 1\}$.

0 ... the language containing the only word 0

01 ... the language containing the only word 01

$0 + 1$... the language containing two words 0 and 1

0^* ... the language containing words $\varepsilon, 0, 00, 000, \dots$

Regular Expressions

Examples: In all examples $\Sigma = \{0, 1\}$.

0 ... the language containing the only word 0

01 ... the language containing the only word 01

$0 + 1$... the language containing two words 0 and 1

0^* ... the language containing words $\varepsilon, 0, 00, 000, \dots$

$(01)^*$... the language containing words $\varepsilon, 01, 0101, 010101, \dots$

Regular Expressions

Examples: In all examples $\Sigma = \{0, 1\}$.

0 ... the language containing the only word 0

01 ... the language containing the only word 01

$0 + 1$... the language containing two words 0 and 1

0^* ... the language containing words $\varepsilon, 0, 00, 000, \dots$

$(01)^*$... the language containing words $\varepsilon, 01, 0101, 010101, \dots$

$(0 + 1)^*$... the language containing all words over the alphabet $\{0, 1\}$

Regular Expressions

Examples: In all examples $\Sigma = \{0, 1\}$.

0 ... the language containing the only word 0

01 ... the language containing the only word 01

$0 + 1$... the language containing two words 0 and 1

0^* ... the language containing words $\varepsilon, 0, 00, 000, \dots$

$(01)^*$... the language containing words $\varepsilon, 01, 0101, 010101, \dots$

$(0 + 1)^*$... the language containing all words over the alphabet $\{0, 1\}$

$(0 + 1)^*00$... the language containing all words ending with 00

Regular Expressions

Examples: In all examples $\Sigma = \{0, 1\}$.

0 ... the language containing the only word 0

01 ... the language containing the only word 01

$0 + 1$... the language containing two words 0 and 1

0^* ... the language containing words $\varepsilon, 0, 00, 000, \dots$

$(01)^*$... the language containing words $\varepsilon, 01, 0101, 010101, \dots$

$(0 + 1)^*$... the language containing all words over the alphabet $\{0, 1\}$

$(0 + 1)^*00$... the language containing all words ending with 00

$(01)^*111(01)^*$... the language containing all words that contain a subword 111 preceded and followed by an arbitrary number of copies of the word 01

$(0 + 1)^*00 + (01)^*111(01)^*$... the language containing all words that either end with 00 or contain a subwords 111 preceded and followed with some arbitrary number of words 01

$(0 + 1)^*00 + (01)^*111(01)^*$... the language containing all words that either end with 00 or contain a subwords 111 preceded and followed with some arbitrary number of words 01

$(0 + 1)^*1(0 + 1)^*$... the language of all words that contain at least one occurrence of symbol 1

Regular Expressions

$(0 + 1)^*00 + (01)^*111(01)^*$... the language containing all words that either end with 00 or contain a subwords 111 preceded and followed with some arbitrary number of words 01

$(0 + 1)^*1(0 + 1)^*$... the language of all words that contain at least one occurrence of symbol 1

$0^*(10^*10^*)^*$... the language containing all words with an even number of occurrences of symbol 1

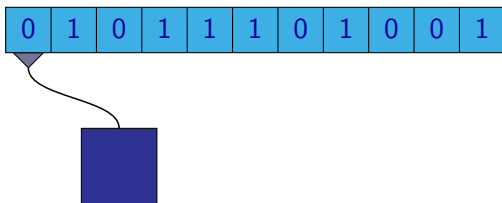
Finite Automata

Recognition of a Language

Example: Consider words over alphabet $\{0, 1\}$.

We would like to recognize a language L consisting of words with even number of symbols 1 .

We want to design a device that reads a word and then tells us if the word belongs to the language L or not.

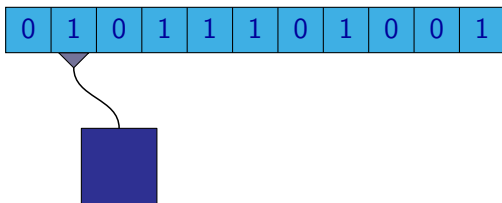


Recognition of a Language

Example: Consider words over alphabet $\{0, 1\}$.

We would like to recognize a language L consisting of words with even number of symbols 1 .

We want to design a device that reads a word and then tells us if the word belongs to the language L or not.

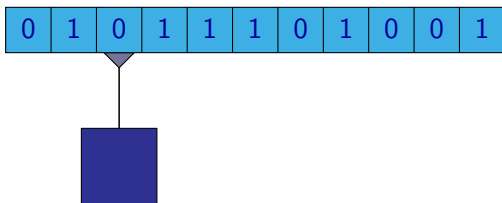


Recognition of a Language

Example: Consider words over alphabet $\{0, 1\}$.

We would like to recognize a language L consisting of words with even number of symbols 1 .

We want to design a device that reads a word and then tells us if the word belongs to the language L or not.

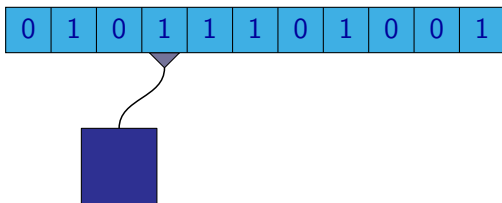


Recognition of a Language

Example: Consider words over alphabet $\{0, 1\}$.

We would like to recognize a language L consisting of words with even number of symbols 1 .

We want to design a device that reads a word and then tells us if the word belongs to the language L or not.

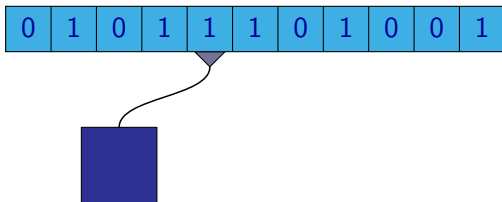


Recognition of a Language

Example: Consider words over alphabet $\{0, 1\}$.

We would like to recognize a language L consisting of words with even number of symbols 1 .

We want to design a device that reads a word and then tells us if the word belongs to the language L or not.

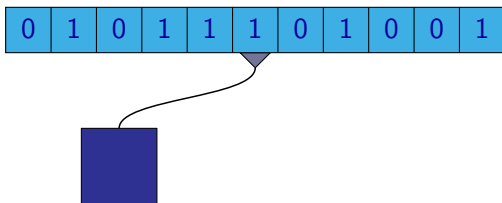


Recognition of a Language

Example: Consider words over alphabet $\{0, 1\}$.

We would like to recognize a language L consisting of words with even number of symbols 1 .

We want to design a device that reads a word and then tells us if the word belongs to the language L or not.

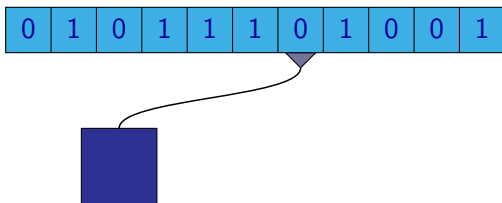


Recognition of a Language

Example: Consider words over alphabet $\{0, 1\}$.

We would like to recognize a language L consisting of words with even number of symbols 1 .

We want to design a device that reads a word and then tells us if the word belongs to the language L or not.

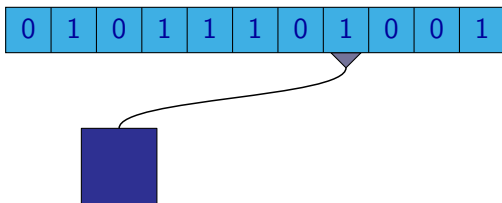


Recognition of a Language

Example: Consider words over alphabet $\{0, 1\}$.

We would like to recognize a language L consisting of words with even number of symbols 1 .

We want to design a device that reads a word and then tells us if the word belongs to the language L or not.

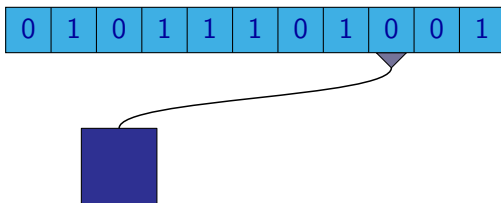


Recognition of a Language

Example: Consider words over alphabet $\{0, 1\}$.

We would like to recognize a language L consisting of words with even number of symbols 1 .

We want to design a device that reads a word and then tells us if the word belongs to the language L or not.

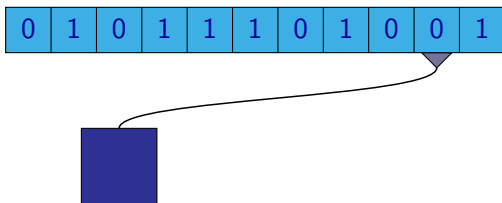


Recognition of a Language

Example: Consider words over alphabet $\{0, 1\}$.

We would like to recognize a language L consisting of words with even number of symbols 1 .

We want to design a device that reads a word and then tells us if the word belongs to the language L or not.

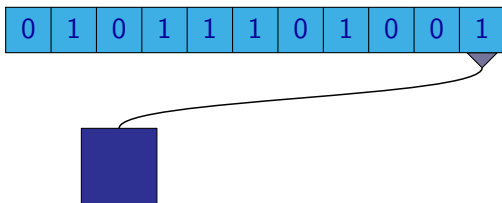


Recognition of a Language

Example: Consider words over alphabet $\{0, 1\}$.

We would like to recognize a language L consisting of words with even number of symbols 1 .

We want to design a device that reads a word and then tells us if the word belongs to the language L or not.

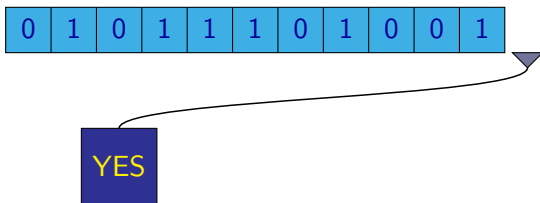


Recognition of a Language

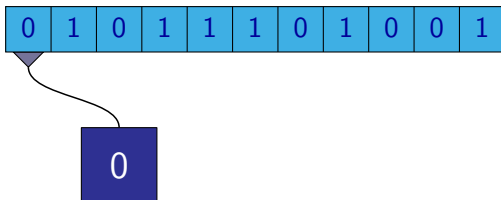
Example: Consider words over alphabet $\{0, 1\}$.

We would like to recognize a language L consisting of words with even number of symbols 1 .

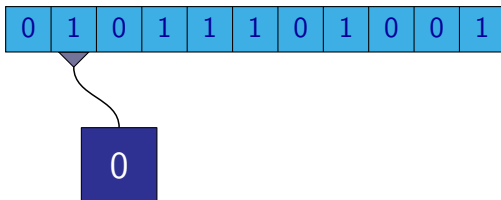
We want to design a device that reads a word and then tells us if the word belongs to the language L or not.



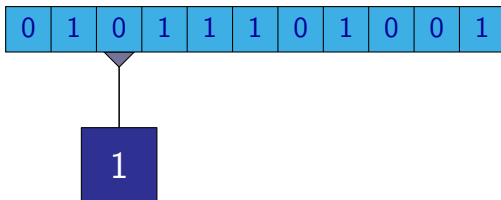
The first idea: To count the number of occurrences of symbol **1**.



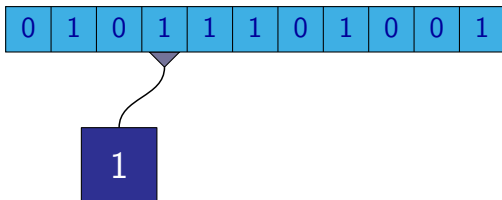
The first idea: To count the number of occurrences of symbol 1.



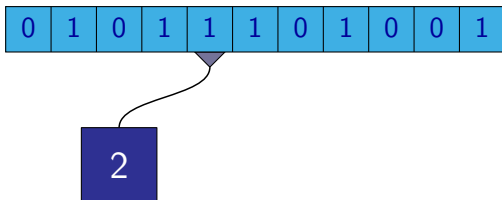
The first idea: To count the number of occurrences of symbol 1.



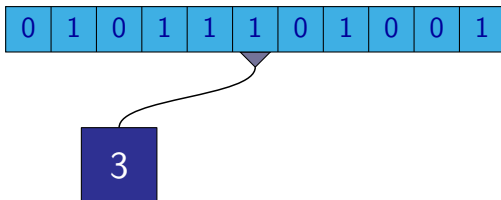
The first idea: To count the number of occurrences of symbol 1.



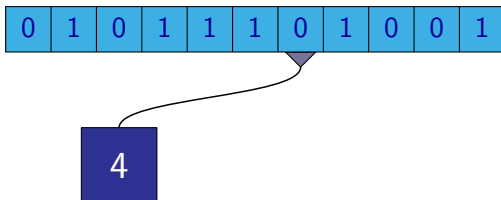
The first idea: To count the number of occurrences of symbol 1.



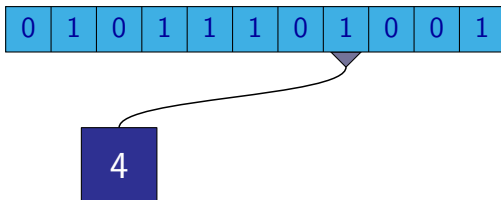
The first idea: To count the number of occurrences of symbol 1.



The first idea: To count the number of occurrences of symbol 1.

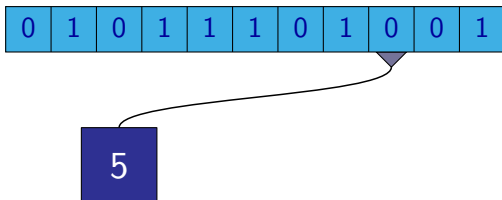


The first idea: To count the number of occurrences of symbol 1.

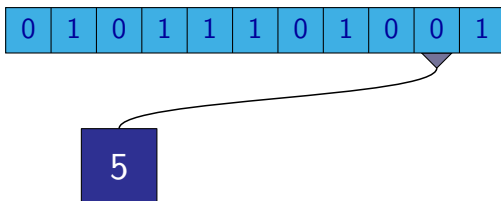


Recognition of a Language

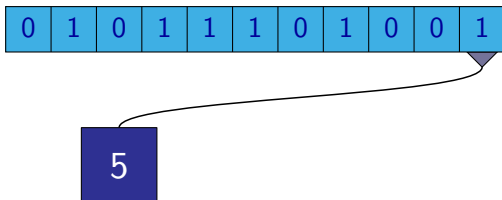
The first idea: To count the number of occurrences of symbol 1.



The first idea: To count the number of occurrences of symbol 1.

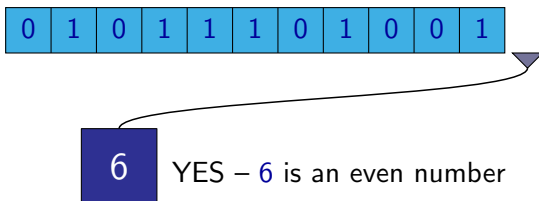


The first idea: To count the number of occurrences of symbol 1.



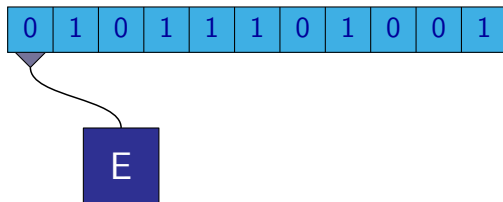
Recognition of a Language

The first idea: To count the number of occurrences of symbol 1.



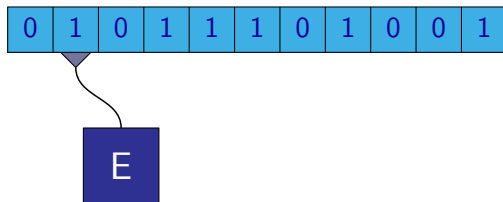
Recognition of a Language

The second idea: In fact, we just need to remember if the number of symbols **1** read so far is even or odd (i.e., it is sufficient to remember only the last bit of the number).



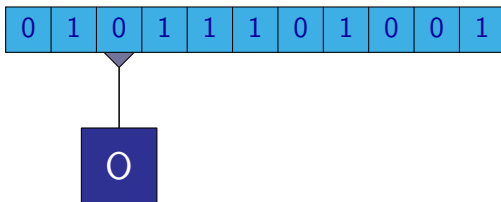
Recognition of a Language

The second idea: In fact, we just need to remember if the number of symbols **1** read so far is even or odd (i.e., it is sufficient to remember only the last bit of the number).



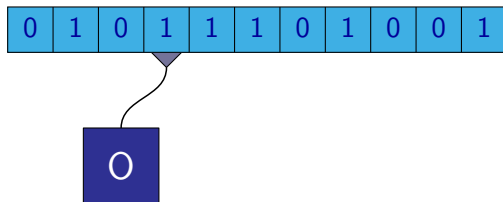
Recognition of a Language

The second idea: In fact, we just need to remember if the number of symbols **1** read so far is even or odd (i.e., it is sufficient to remember only the last bit of the number).



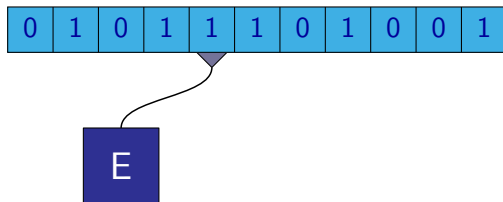
Recognition of a Language

The second idea: In fact, we just need to remember if the number of symbols **1** read so far is even or odd (i.e., it is sufficient to remember only the last bit of the number).



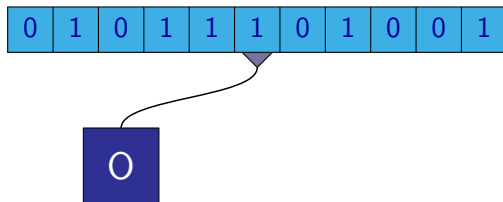
Recognition of a Language

The second idea: In fact, we just need to remember if the number of symbols **1** read so far is even or odd (i.e., it is sufficient to remember only the last bit of the number).



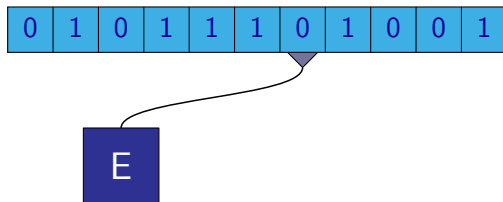
Recognition of a Language

The second idea: In fact, we just need to remember if the number of symbols **1** read so far is even or odd (i.e., it is sufficient to remember only the last bit of the number).



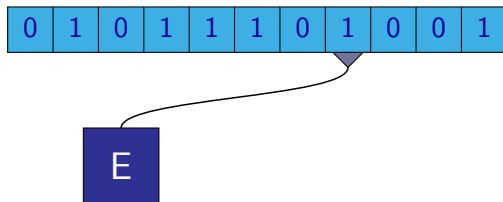
Recognition of a Language

The second idea: In fact, we just need to remember if the number of symbols **1** read so far is even or odd (i.e., it is sufficient to remember only the last bit of the number).



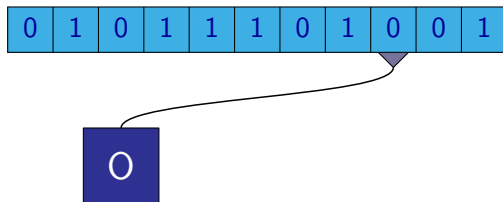
Recognition of a Language

The second idea: In fact, we just need to remember if the number of symbols **1** read so far is even or odd (i.e., it is sufficient to remember only the last bit of the number).



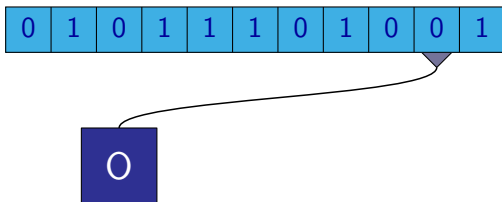
Recognition of a Language

The second idea: In fact, we just need to remember if the number of symbols **1** read so far is even or odd (i.e., it is sufficient to remember only the last bit of the number).



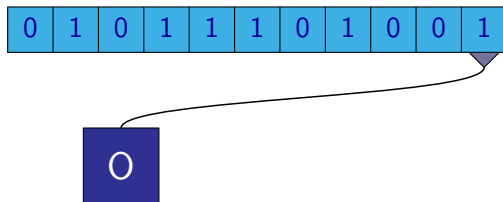
Recognition of a Language

The second idea: In fact, we just need to remember if the number of symbols **1** read so far is even or odd (i.e., it is sufficient to remember only the last bit of the number).



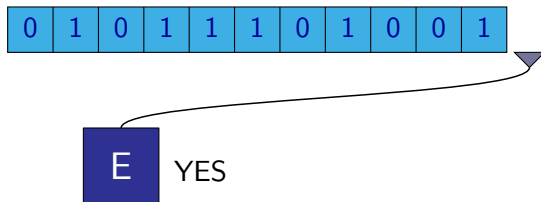
Recognition of a Language

The second idea: In fact, we just need to remember if the number of symbols **1** read so far is even or odd (i.e., it is sufficient to remember only the last bit of the number).



Recognition of a Language

The second idea: In fact, we just need to remember if the number of symbols **1** read so far is even or odd (i.e., it is sufficient to remember only the last bit of the number).



Recognition of a Language

The behaviour of the device can be described by the following graph:



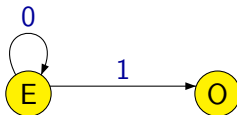
Recognition of a Language

The behaviour of the device can be described by the following graph:



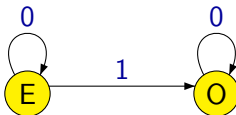
Recognition of a Language

The behaviour of the device can be described by the following graph:



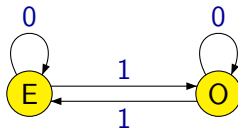
Recognition of a Language

The behaviour of the device can be described by the following graph:



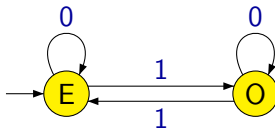
Recognition of a Language

The behaviour of the device can be described by the following graph:



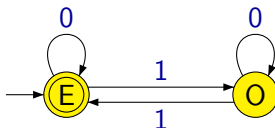
Recognition of a Language

The behaviour of the device can be described by the following graph:



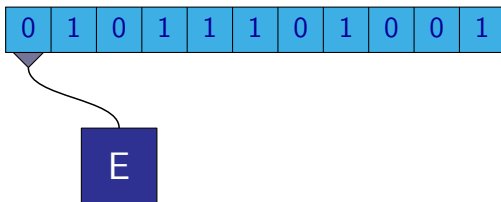
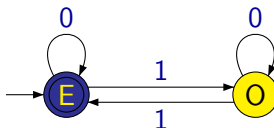
Recognition of a Language

The behaviour of the device can be described by the following graph:



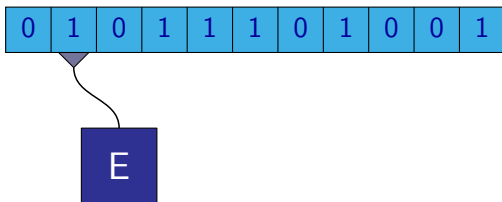
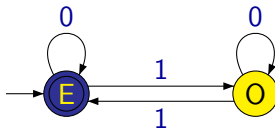
Recognition of a Language

The behaviour of the device can be described by the following graph:



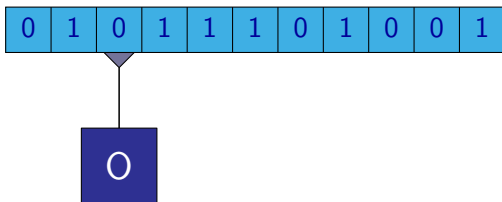
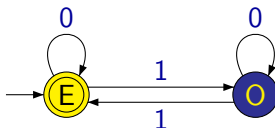
Recognition of a Language

The behaviour of the device can be described by the following graph:



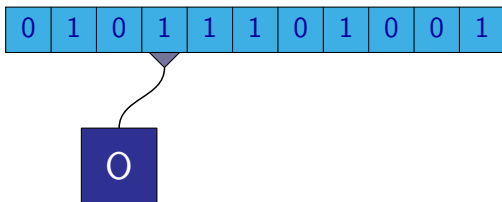
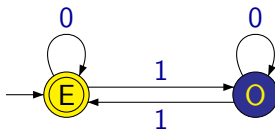
Recognition of a Language

The behaviour of the device can be described by the following graph:



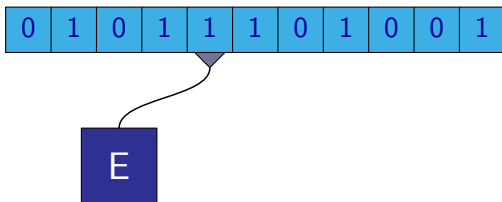
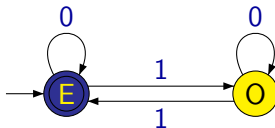
Recognition of a Language

The behaviour of the device can be described by the following graph:



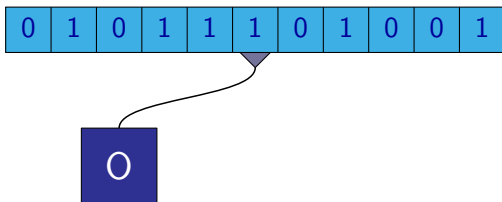
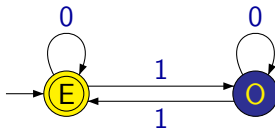
Recognition of a Language

The behaviour of the device can be described by the following graph:



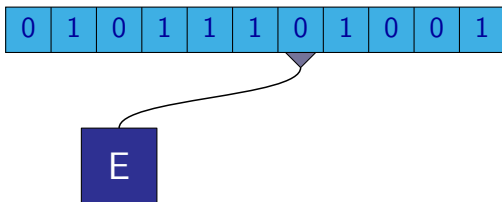
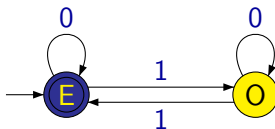
Recognition of a Language

The behaviour of the device can be described by the following graph:



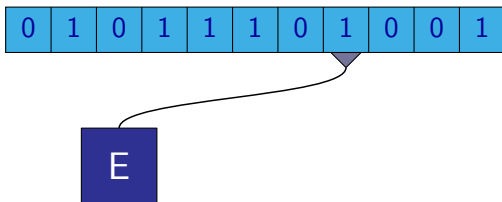
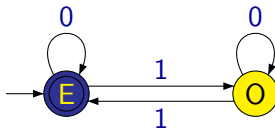
Recognition of a Language

The behaviour of the device can be described by the following graph:



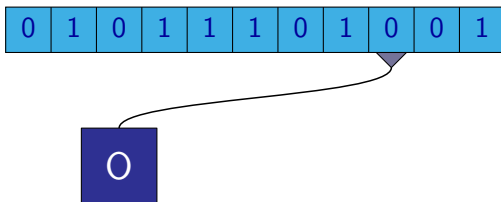
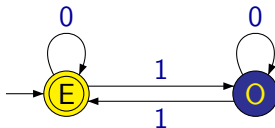
Recognition of a Language

The behaviour of the device can be described by the following graph:



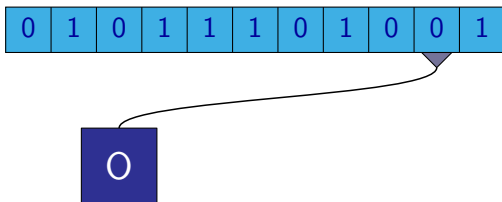
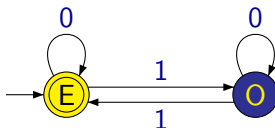
Recognition of a Language

The behaviour of the device can be described by the following graph:



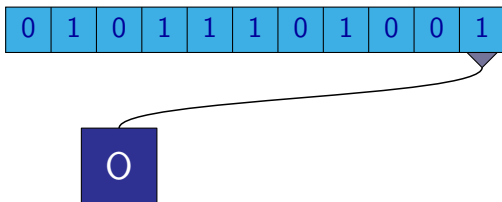
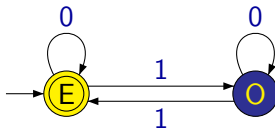
Recognition of a Language

The behaviour of the device can be described by the following graph:



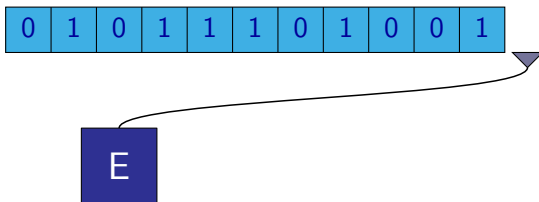
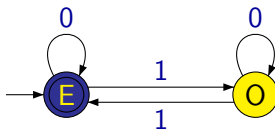
Recognition of a Language

The behaviour of the device can be described by the following graph:

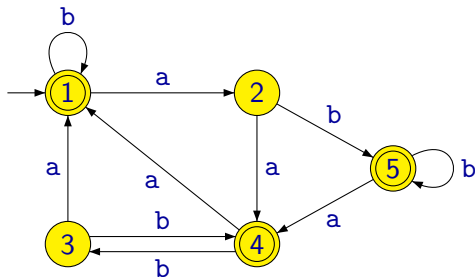


Recognition of a Language

The behaviour of the device can be described by the following graph:



Deterministic Finite Automaton



A **deterministic finite automaton** consists of **states** and **transitions**. One of the states is denoted as an **initial state** and some of states are denoted as **accepting**.

Deterministic Finite Automaton

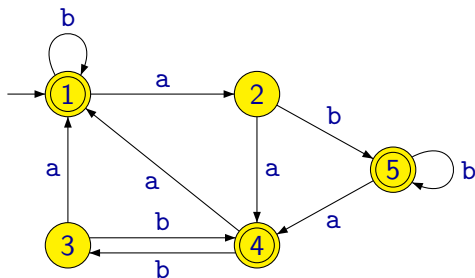
Formally, a **deterministic finite automaton (DFA)** is defined as a tuple

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

where:

- Q is a nonempty finite set of **states**
- Σ is an **alphabet** (a nonempty finite set of symbols)
- $\delta : Q \times \Sigma \rightarrow Q$ is a **transition function**
- $q_0 \in Q$ is an **initial state**
- $F \subseteq Q$ is a set of **accepting states**

Deterministic Finite Automaton



- $Q = \{1, 2, 3, 4, 5\}$

- $\Sigma = \{a, b\}$

- $q_0 = 1$

- $F = \{1, 4, 5\}$

$$\delta(1, a) = 2$$

$$\delta(1, b) = 1$$

$$\delta(2, a) = 4$$

$$\delta(2, b) = 5$$

$$\delta(3, a) = 1$$

$$\delta(3, b) = 4$$

$$\delta(4, a) = 1$$

$$\delta(4, b) = 3$$

$$\delta(5, a) = 4$$

$$\delta(5, b) = 5$$

Deterministic Finite Automaton

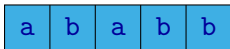
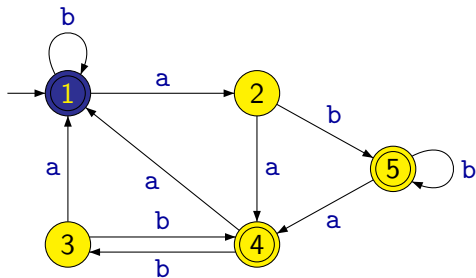
Instead of

$$\begin{array}{ll} \delta(1, a) = 2 & \delta(1, b) = 1 \\ \delta(2, a) = 4 & \delta(2, b) = 5 \\ \delta(3, a) = 1 & \delta(3, b) = 4 \\ \delta(4, a) = 1 & \delta(4, b) = 3 \\ \delta(5, a) = 4 & \delta(5, b) = 5 \end{array}$$

we rather use a more succinct representation as a table or a depicted graph:

δ	a	b
$\leftrightarrow 1$	2	1
2	4	5
3	1	4
$\leftarrow 4$	1	3
$\leftarrow 5$	4	5

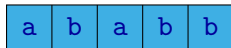
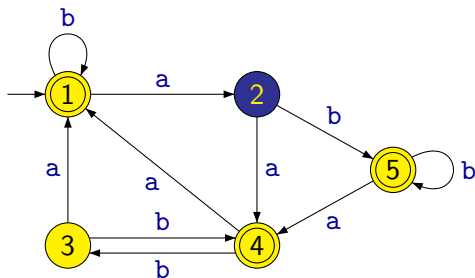
Deterministic Finite Automaton



1

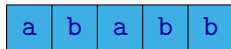
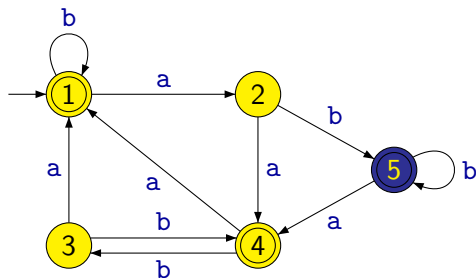


Deterministic Finite Automaton



$1 \xrightarrow{a} 2$

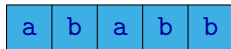
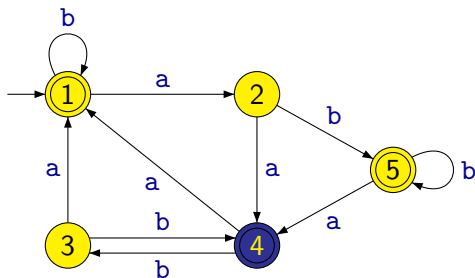
Deterministic Finite Automaton



$1 \xrightarrow{a} 2 \xrightarrow{b} 5$

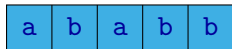
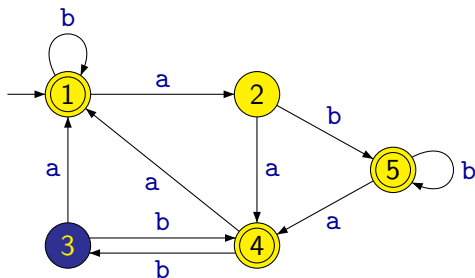


Deterministic Finite Automaton



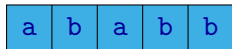
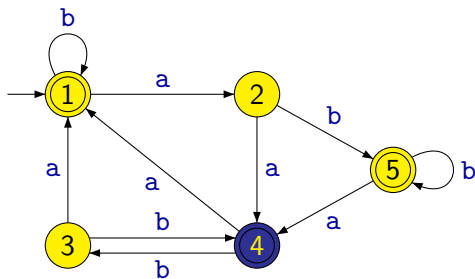
$1 \xrightarrow{a} 2 \xrightarrow{b} 5 \xrightarrow{a} 4$

Deterministic Finite Automaton



$1 \xrightarrow{a} 2 \xrightarrow{b} 5 \xrightarrow{a} 4 \xrightarrow{b} 3$

Deterministic Finite Automaton



$1 \xrightarrow{a} 2 \xrightarrow{b} 5 \xrightarrow{a} 4 \xrightarrow{b} 3 \xrightarrow{b} 4$

Definition

Let us have a DFA $A = (Q, \Sigma, \delta, q_0, F)$.

By $q \xrightarrow{w} q'$, where $q, q' \in Q$ and $w \in \Sigma^*$, we denote the fact that the automaton, starting in state q goes to state q' by reading word w .

Remark: $\longrightarrow \subseteq Q \times \Sigma^* \times Q$ is a ternary relation.

Instead of $(q, w, q') \in \longrightarrow$ we write $q \xrightarrow{w} q'$.

It holds for a DFA that for each state q and each word w there is exactly one state q' such that $q \xrightarrow{w} q'$.

Relation $\longrightarrow$ can be formally defined by the following inductive definition:

- $q \xrightarrow{\varepsilon} q$ for each $q \in Q$
- For $a \in \Sigma$ and $w \in \Sigma^*$:
 $q \xrightarrow{aw} q'$ iff there is $q'' \in Q$ such that $\delta(q, a) = q''$ and $q'' \xrightarrow{w} q'$.

Deterministic Finite Automaton

A word $w \in \Sigma^*$ is **accepted** by a deterministic finite automaton $A = (Q, \Sigma, \delta, q_0, F)$ iff there exists a state $q \in F$ such that $q_0 \xrightarrow{w} q$.

Definition

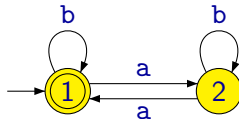
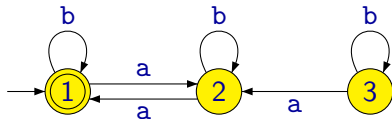
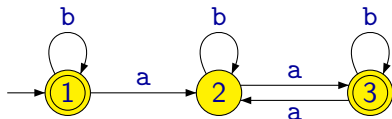
A **language** accepted by a given deterministic finite automaton $A = (Q, \Sigma, \delta, q_0, F)$, denoted $L(A)$, is the set of all words accepted by the automaton, i.e.,

$$L(A) = \{w \in \Sigma^* \mid \exists q \in F : q_0 \xrightarrow{w} q\}$$

Definition

A language L is **regular** iff there exists some deterministic finite automaton accepting L , i.e., DFA A such that $L(A) = L$.

Equivalence of Automata



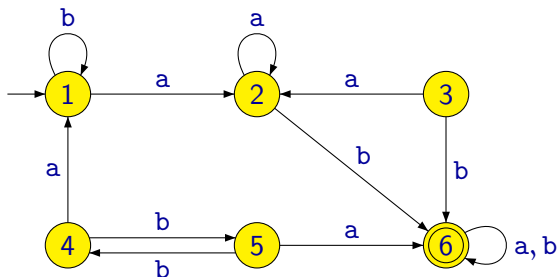
All 3 automata accept the language of all words with an even number of a 's.

Equivalence of Automata

Definition

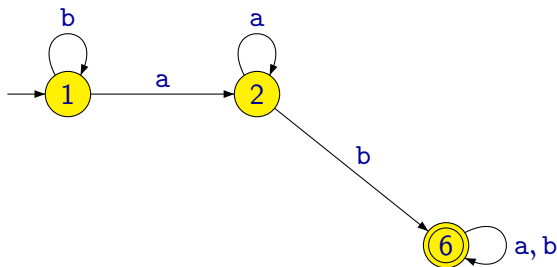
We say automata A_1, A_2 are **equivalent** if $L(A_1) = L(A_2)$.

Unreachable States of an Automaton



- The automaton accepts the language
 $L = \{w \in \{a, b\}^* \mid w \text{ contains subword } ab\}$
- There is no input sequence such that after reading it, the automaton gets to states 3, 4, or 5.

Unreachable States of an Automaton



- The automaton accepts the language $L = \{w \in \{a, b\}^* \mid w \text{ contains subword } ab\}$
- There is no input sequence such that after reading it, the automaton gets to states 3, 4, or 5.
- If we remove these states, the automaton still accepts the same language L .

Unreachable States of an Automaton

Definition

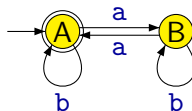
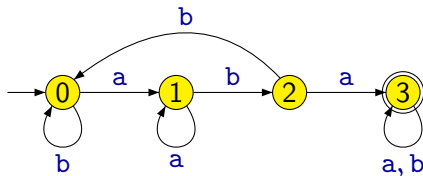
A state q of a finite automaton $A = (Q, \Sigma, \delta, q_0, F)$ is **reachable** if there exists a word w such that $q_0 \xrightarrow{w} q$.

Otherwise the state is **unreachable**.

- There is no path in a graph of an automaton going from the initial state to some unreachable state.
- Unreachable states can be removed from an automaton (together with all transitions going to them and from them). The language accepted by the automaton is not affected.

An Automaton for Intersection of Languages

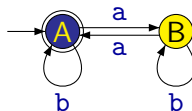
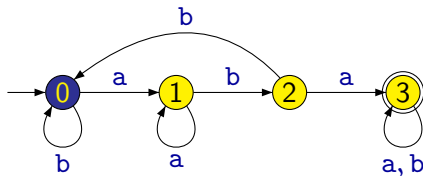
Let us have the following two automata:



Do both of them accept the word **ababb**?

An Automaton for Intersection of Languages

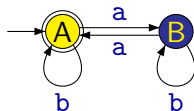
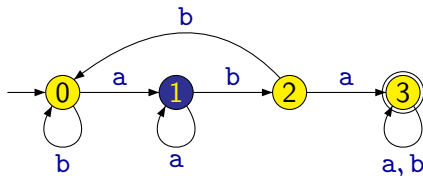
Let us have the following two automata:



Do both of them accept the word **a**babbb?

An Automaton for Intersection of Languages

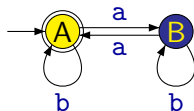
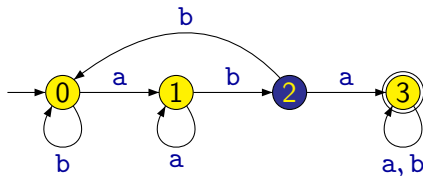
Let us have the following two automata:



Do both of them accept the word **a****b**abb?

An Automaton for Intersection of Languages

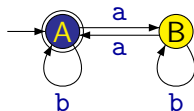
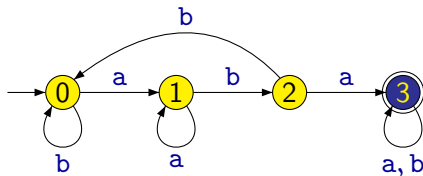
Let us have the following two automata:



Do both of them accept the word **ababb**?

An Automaton for Intersection of Languages

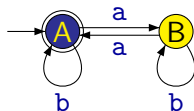
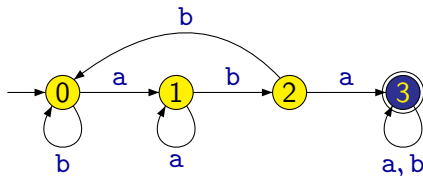
Let us have the following two automata:



Do both of them accept the word **ababbb**?

An Automaton for Intersection of Languages

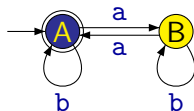
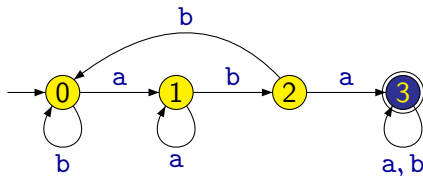
Let us have the following two automata:



Do both of them accept the word **abab****b**?

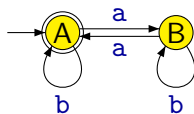
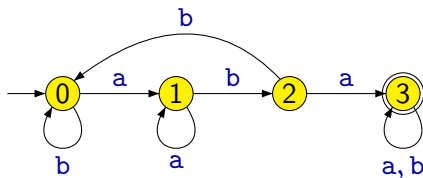
An Automaton for Intersection of Languages

Let us have the following two automata:

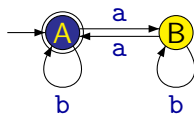
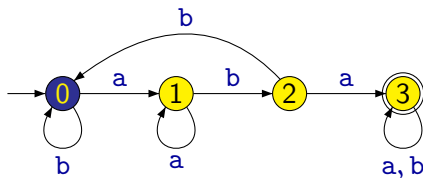


Do both of them accept the word **ababb**?

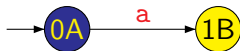
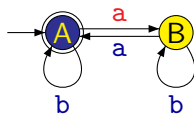
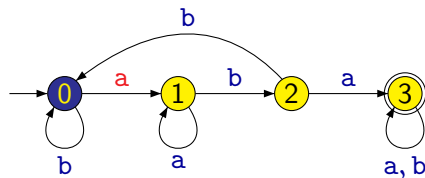
An Automaton for Intersection of Languages



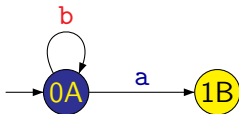
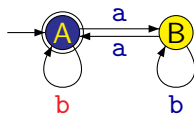
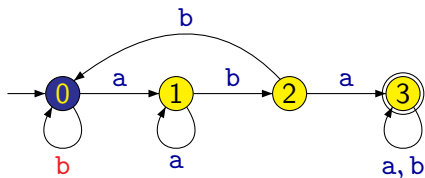
An Automaton for Intersection of Languages



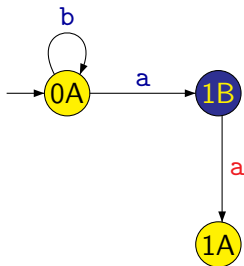
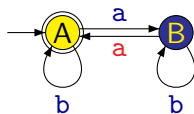
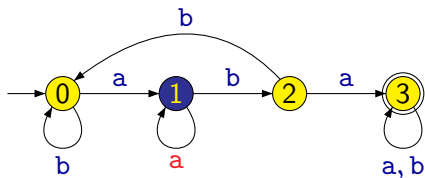
An Automaton for Intersection of Languages



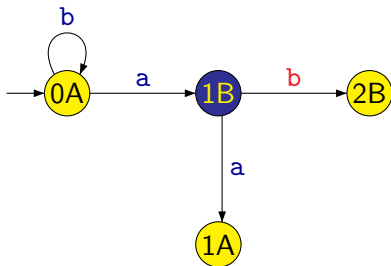
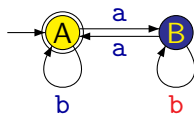
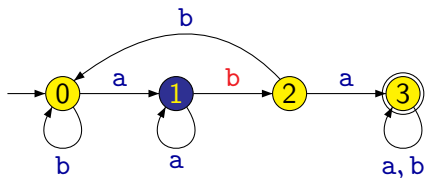
An Automaton for Intersection of Languages



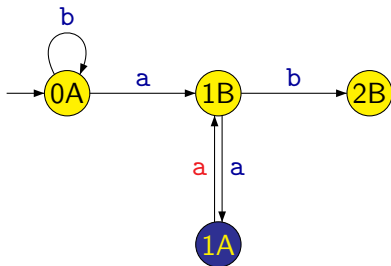
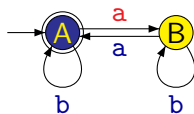
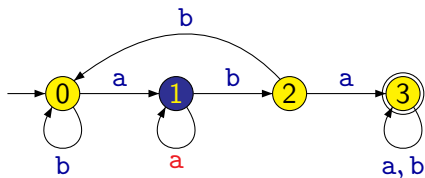
An Automaton for Intersection of Languages



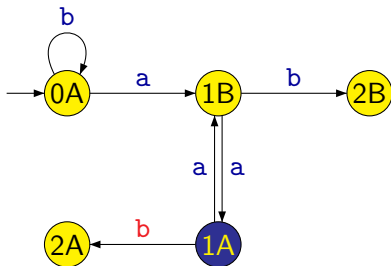
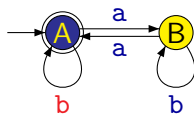
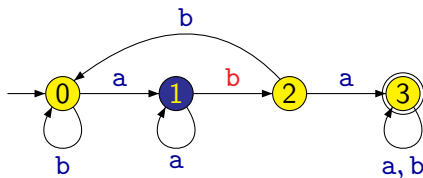
An Automaton for Intersection of Languages



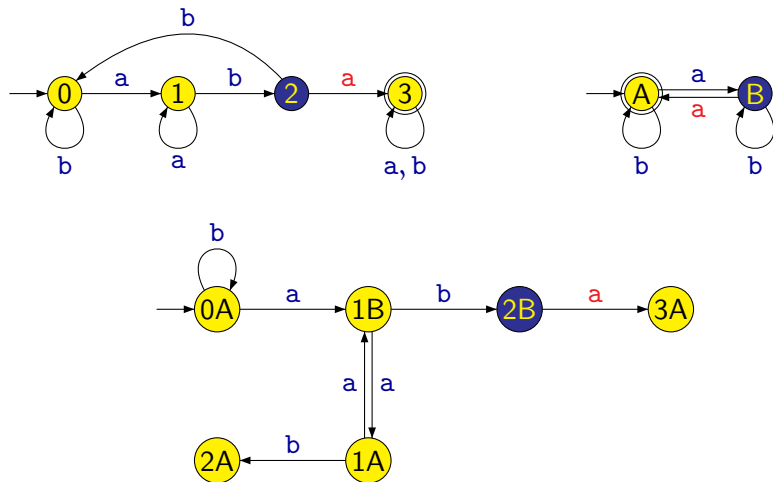
An Automaton for Intersection of Languages



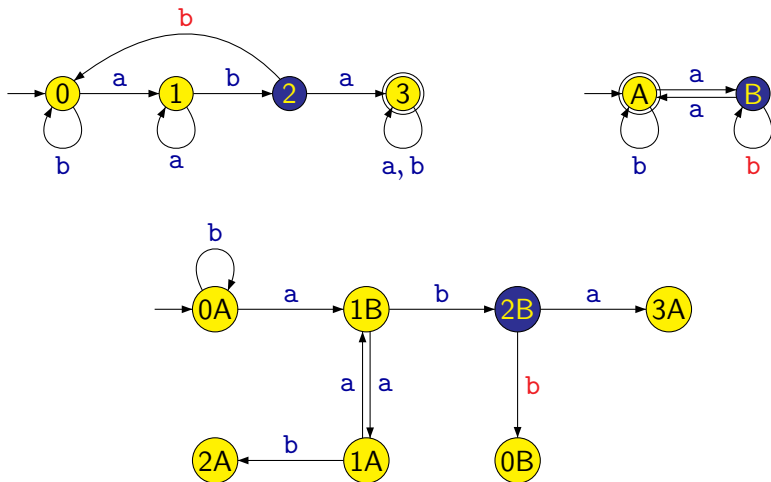
An Automaton for Intersection of Languages



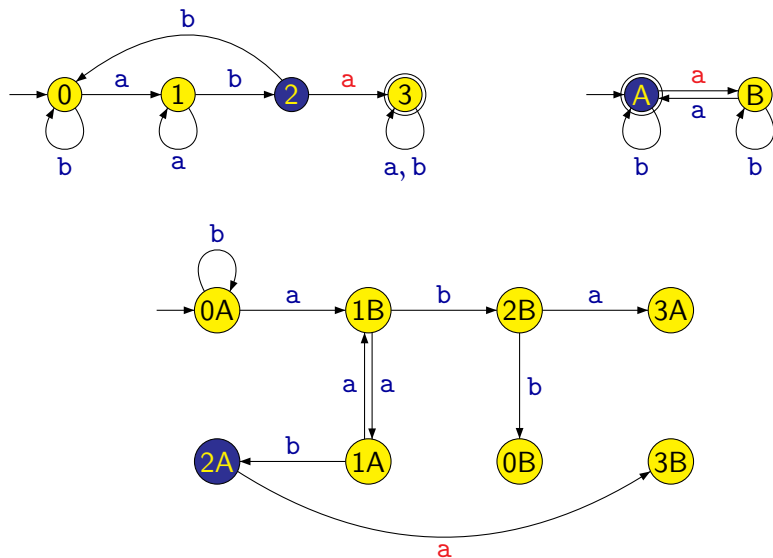
An Automaton for Intersection of Languages



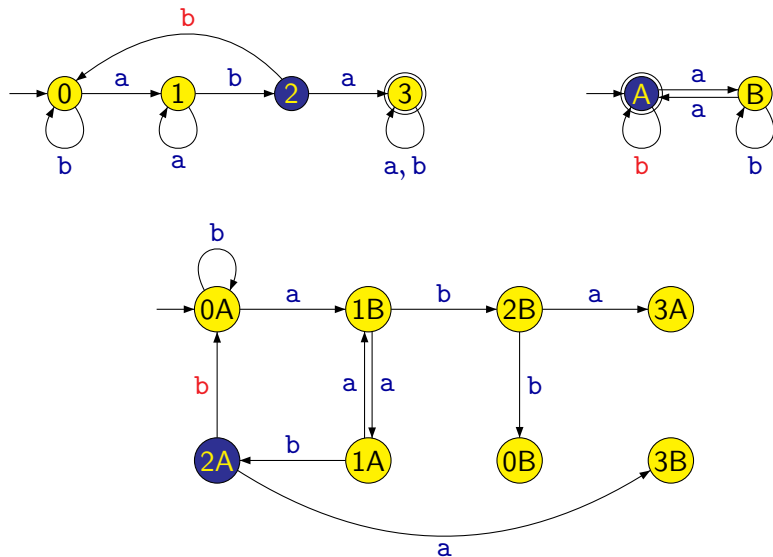
An Automaton for Intersection of Languages



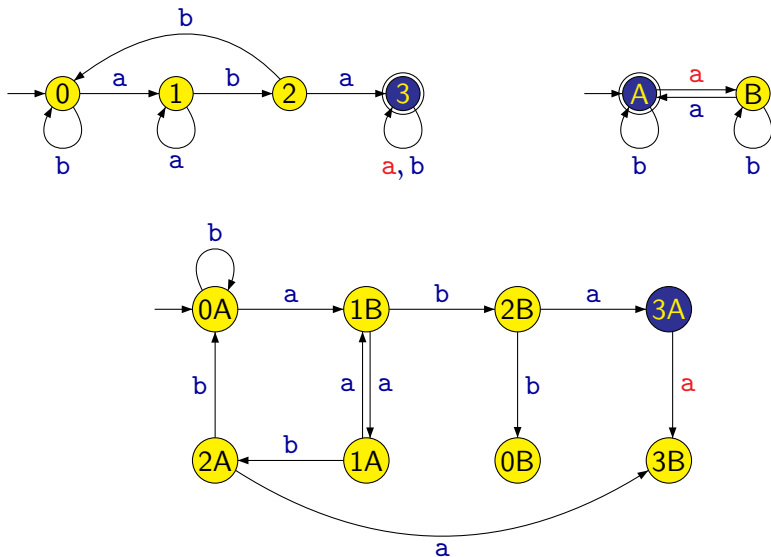
An Automaton for Intersection of Languages



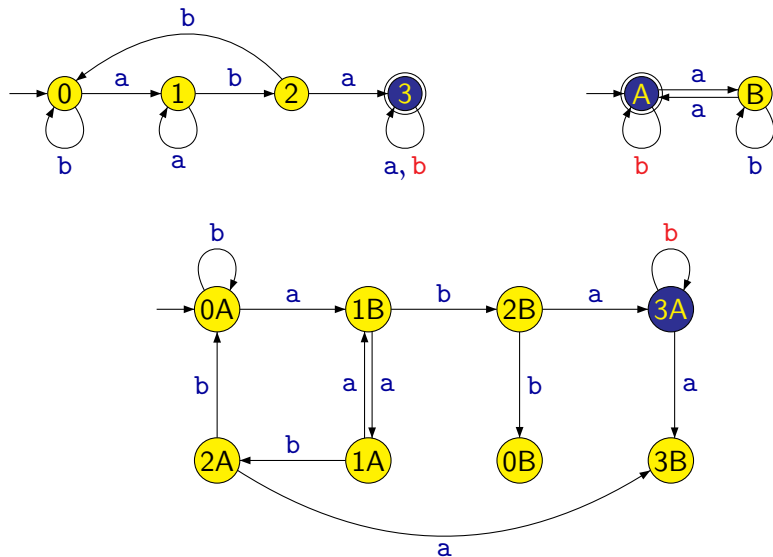
An Automaton for Intersection of Languages



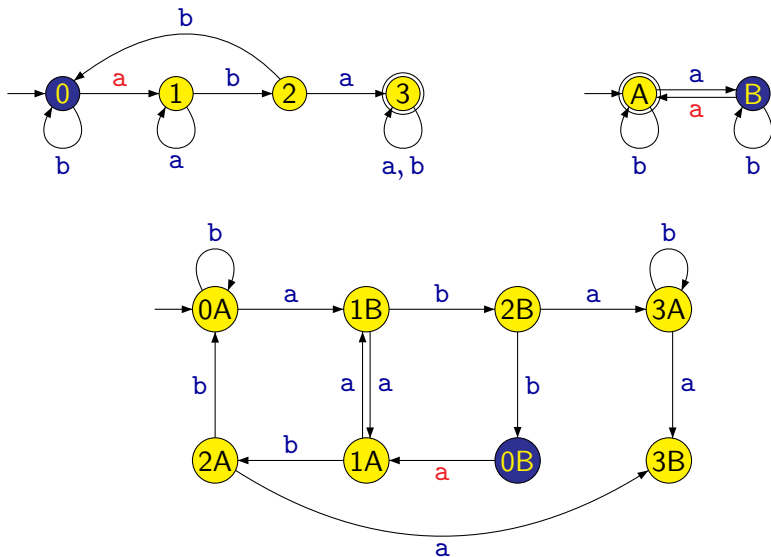
An Automaton for Intersection of Languages



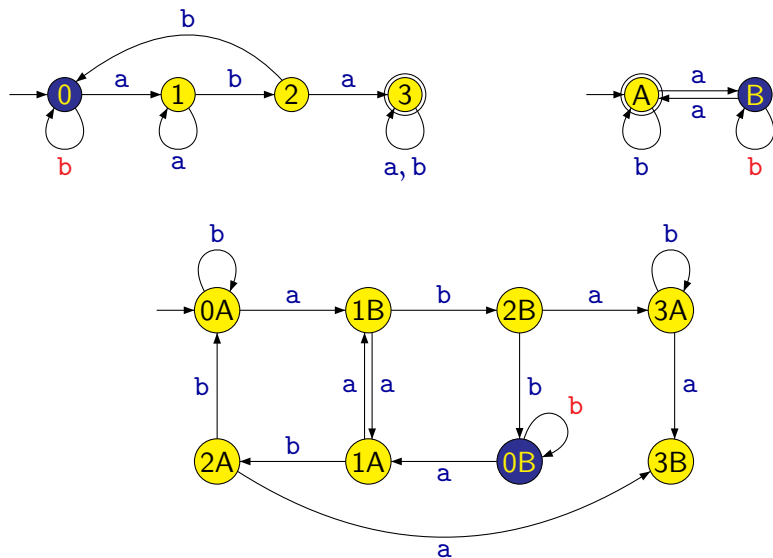
An Automaton for Intersection of Languages



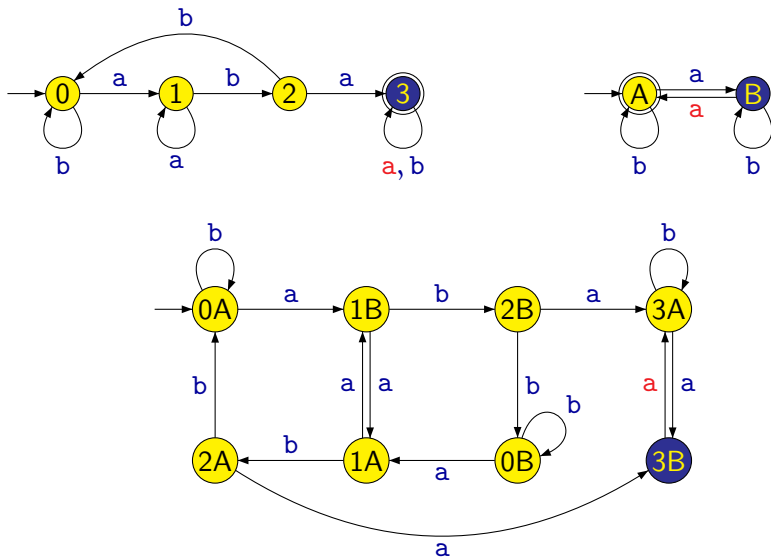
An Automaton for Intersection of Languages



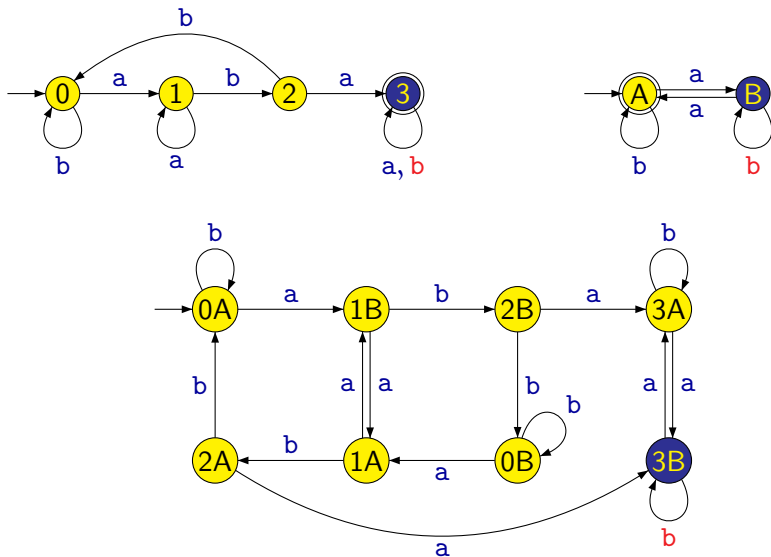
An Automaton for Intersection of Languages



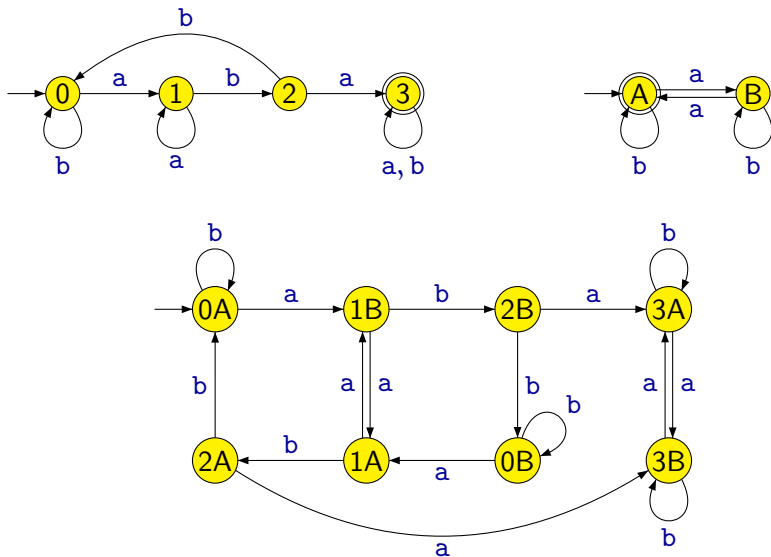
An Automaton for Intersection of Languages



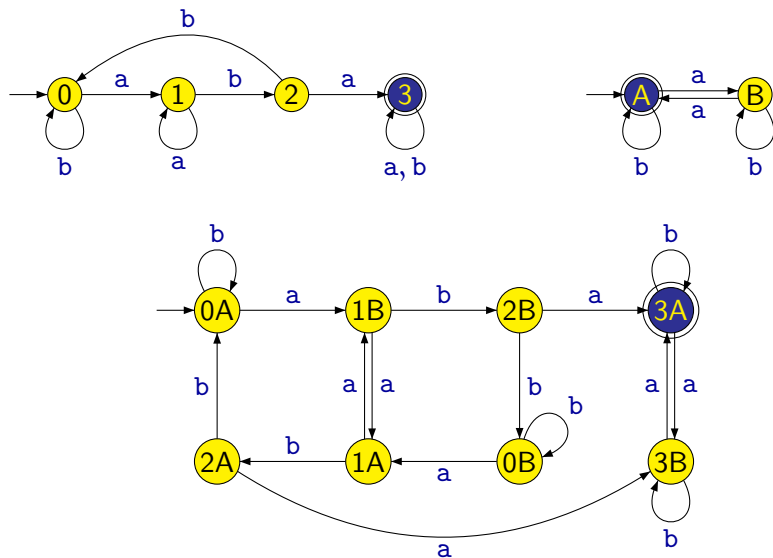
An Automaton for Intersection of Languages



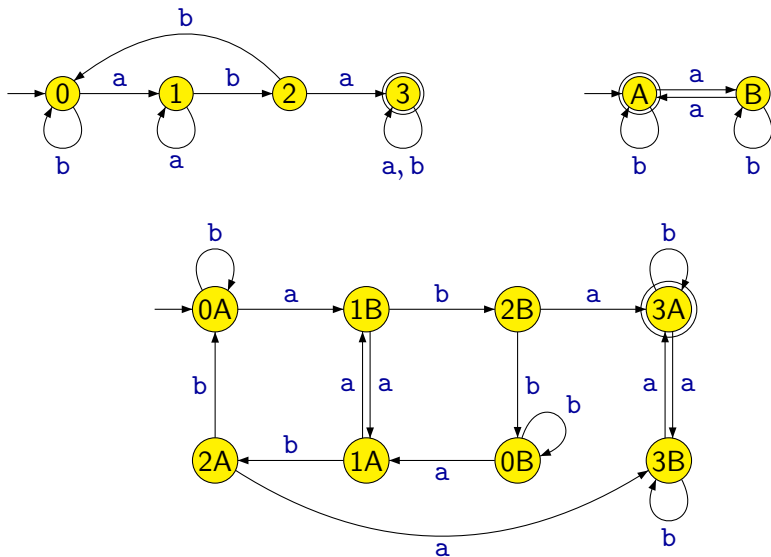
An Automaton for Intersection of Languages



An Automaton for Intersection of Languages



An Automaton for Intersection of Languages



An Automaton for Intersection of Languages

Formally, the construction can be described as follows:

We assume we have two deterministic finite automata $A_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$.

We construct DFA $A = (Q, \Sigma, \delta, q_0, F)$ where:

- $Q = Q_1 \times Q_2$
- $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$ for each $q_1 \in Q_1, q_2 \in Q_2, a \in \Sigma$
- $q_0 = (q_{01}, q_{02})$
- $F = F_1 \times F_2$

It is not difficult to check that for each word $w \in \Sigma^*$ we have $w \in L(A)$ iff $w \in L(A_1)$ and $w \in L(A_2)$, i.e.,

$$L(A) = L(A_1) \cap L(A_2)$$

Intersection of Regular Languages

Theorem

If languages $L_1, L_2 \subseteq \Sigma^*$ are regular then also the language $L_1 \cap L_2$ is regular.

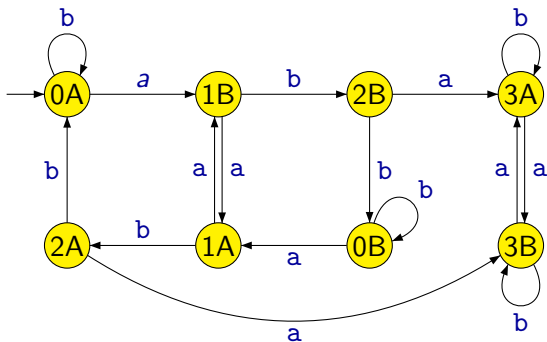
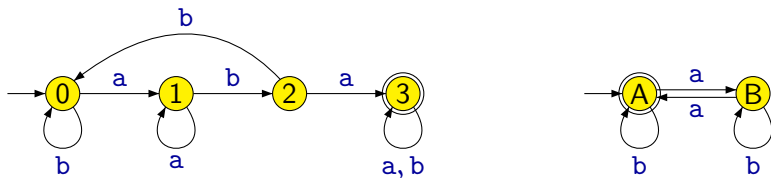
Proof: Let us assume that A_1 and A_2 are deterministic finite automata such that

$$L_1 = L(A_1) \qquad L_2 = L(A_2)$$

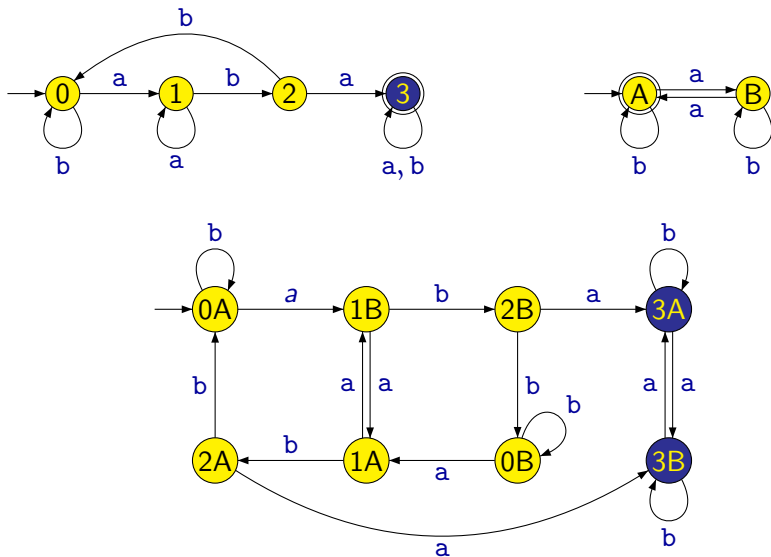
Using the described construction, we can construct a deterministic finite automaton A such that

$$L(A) = L(A_1) \cap L(A_2) = L_1 \cap L_2$$

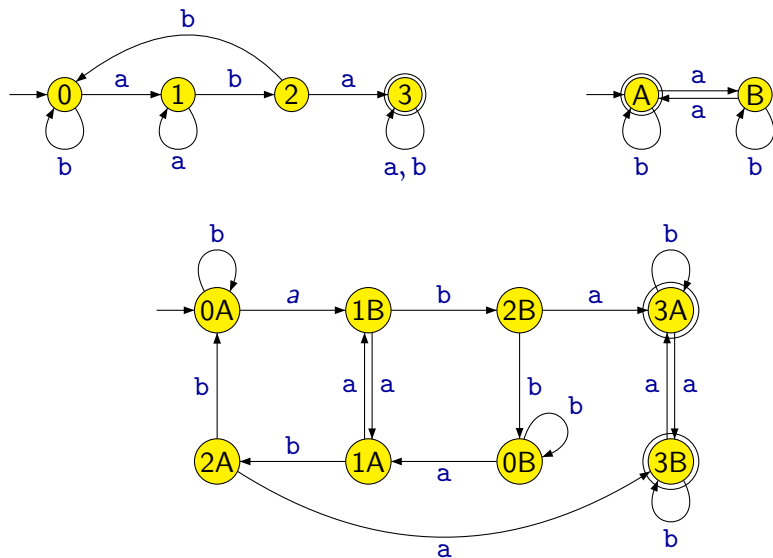
An Automaton for the Union of Languages



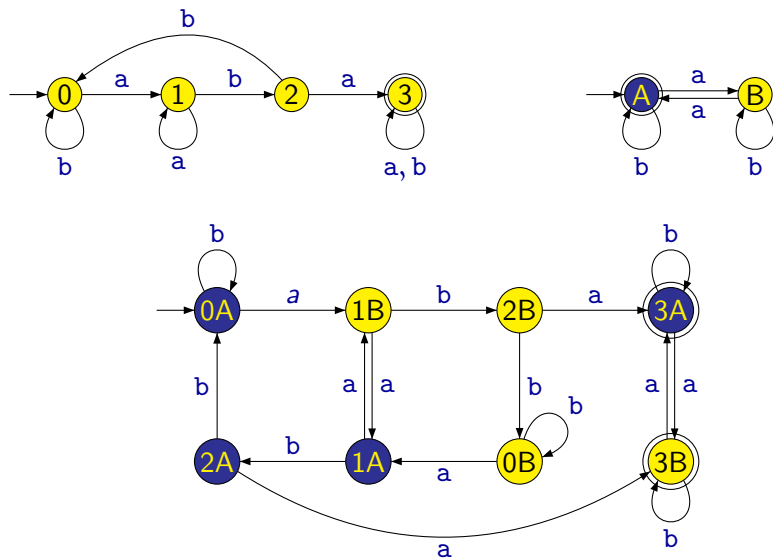
An Automaton for the Union of Languages



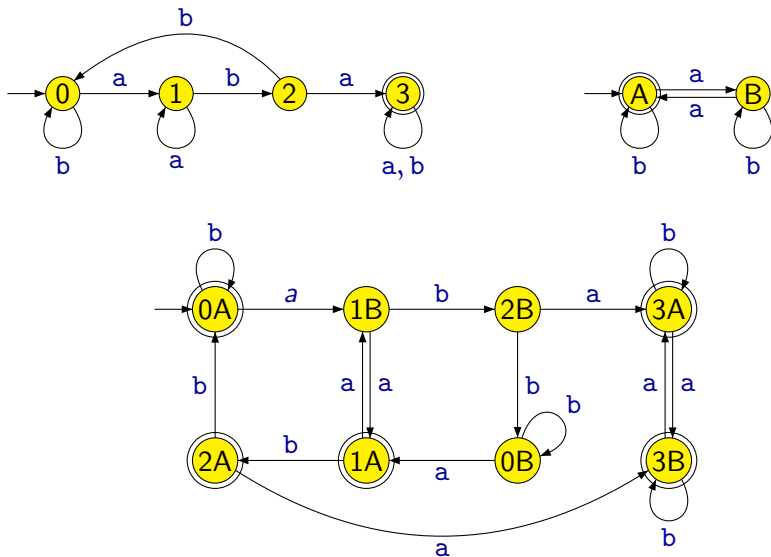
An Automaton for the Union of Languages



An Automaton for the Union of Languages



An Automaton for the Union of Languages



Union of Regular Languages

The construction of an automaton A that accepts the **union** of languages accepted by automata A_1 and A_2 , i.e., the language

$$L(A_1) \cup L(A_2)$$

is almost identical as in the case of the automaton accepting $L(A_1) \cap L(A_2)$.

The only difference is the set of accepting states:

- $F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$

Union of Regular Languages

The construction of an automaton A that accepts the **union** of languages accepted by automata A_1 and A_2 , i.e., the language

$$L(A_1) \cup L(A_2)$$

is almost identical as in the case of the automaton accepting $L(A_1) \cap L(A_2)$.

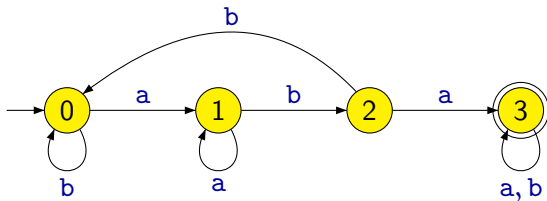
The only difference is the set of accepting states:

- $F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$

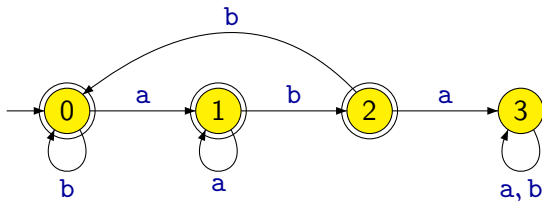
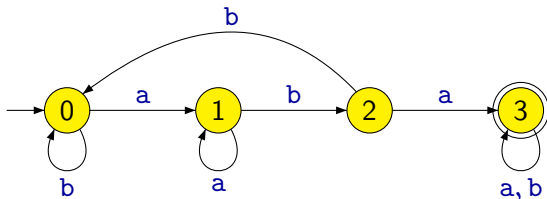
Theorem

If languages $L_1, L_2 \subseteq \Sigma^*$ are regular then also the language $L_1 \cup L_2$ is regular.

An Automaton for the Complement of a Language



An Automaton for the Complement of a Language



Complement of a Regular Language

Given a DFA $A = (Q, \Sigma, \delta, q_0, F)$ we construct DFA $A' = (Q, \Sigma, \delta, q_0, Q - F)$.

It is obvious that for each word $w \in \Sigma^*$ we have $w \in L(A')$ iff $w \notin L(A)$, i.e.,

$$L(A') = \overline{L(A)}$$

Complement of a Regular Language

Given a DFA $A = (Q, \Sigma, \delta, q_0, F)$ we construct DFA $A' = (Q, \Sigma, \delta, q_0, Q - F)$.

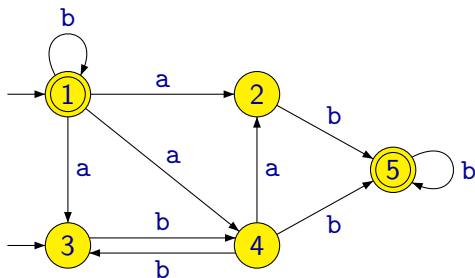
It is obvious that for each word $w \in \Sigma^*$ we have $w \in L(A')$ iff $w \notin L(A)$, i.e.,

$$L(A') = \overline{L(A)}$$

Theorem

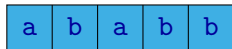
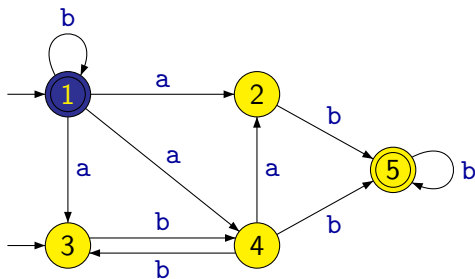
If a language L is regular then also its complement $\overline{L}$ is regular.

Nondeterministic Finite Automaton



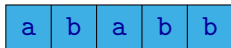
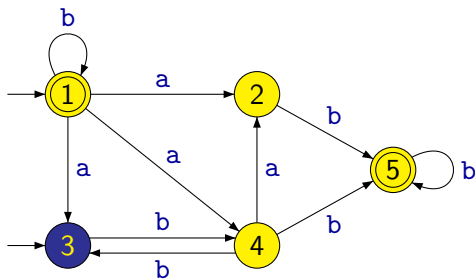
- The number of transitions going from one state and labelled with the same symbol can be arbitrary (including zero).
- There can be more than one initial state in the automaton.

Nondeterministic Finite Automaton



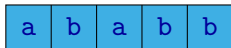
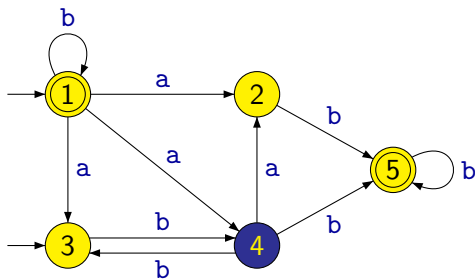
1

Nondeterministic Finite Automaton



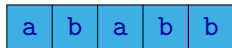
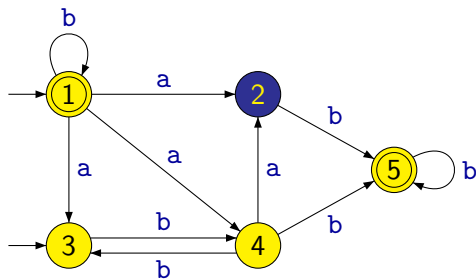
$$1 \xrightarrow{a} 3$$

Nondeterministic Finite Automaton



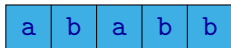
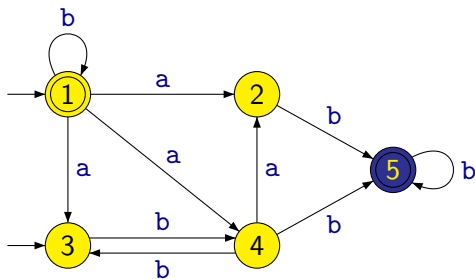
$$1 \xrightarrow{a} 3 \xrightarrow{b} 4$$

Nondeterministic Finite Automaton



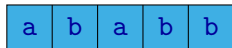
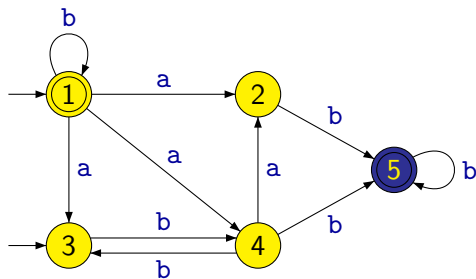
$$1 \xrightarrow{a} 3 \xrightarrow{b} 4 \xrightarrow{a} 2$$

Nondeterministic Finite Automaton



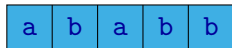
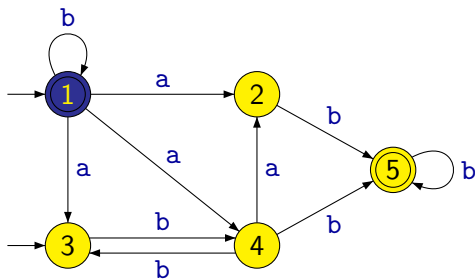
$$1 \xrightarrow{a} 3 \xrightarrow{b} 4 \xrightarrow{a} 2 \xrightarrow{b} 5$$

Nondeterministic Finite Automaton



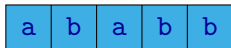
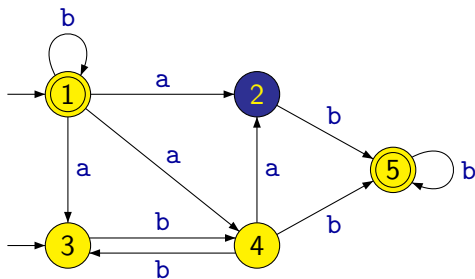
$1 \xrightarrow{a} 3 \xrightarrow{b} 4 \xrightarrow{a} 2 \xrightarrow{b} 5 \xrightarrow{b} 5$

Nondeterministic Finite Automaton



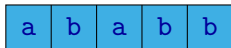
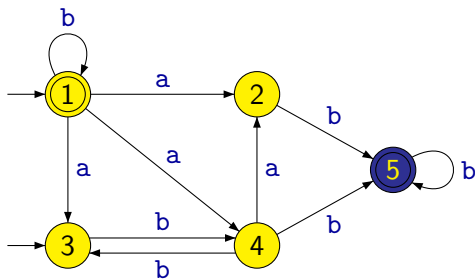
1

Nondeterministic Finite Automaton



$$1 \xrightarrow{a} 2$$

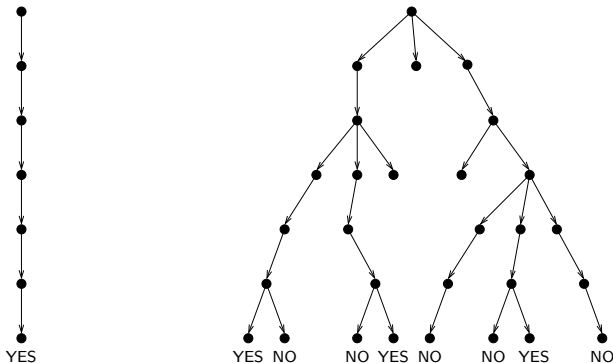
Nondeterministic Finite Automaton



$$1 \xrightarrow{a} 2 \xrightarrow{b} 5$$

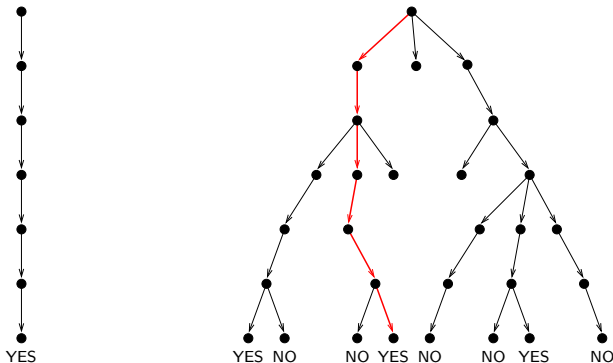
Nondeterministic Finite Automaton

A nondeterministic finite automaton accepts a given word if there **exists** at least one computation of the automaton that accepts the word.



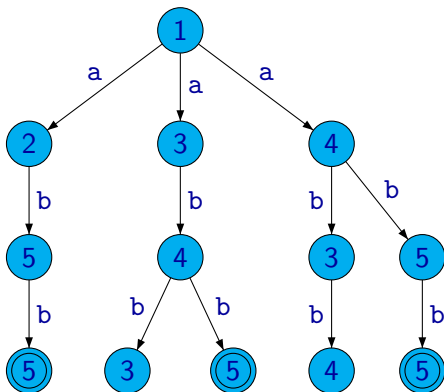
Nondeterministic Finite Automaton

A nondeterministic finite automaton accepts a given word if there **exists** at least one computation of the automaton that accepts the word.



Nondeterministic Finite Automaton

	a	b
$\leftrightarrow 1$	2, 3, 4	1
2	—	5
$\rightarrow 3$	—	4
4	2	3, 5
$\leftarrow 5$	—	5



Example: A forest representing all possible computations over the word **abb**.

Nondeterministic Finite Automaton

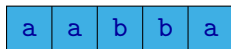
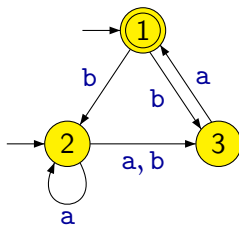
Formally, a **nondeterministic finite automaton** (**NFA**) is defined as a tuple

$$(Q, \Sigma, \delta, I, F)$$

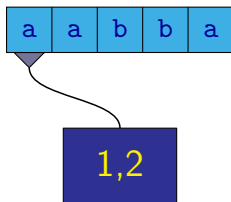
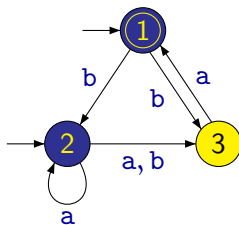
where:

- Q is a finite set of **states**
- Σ is a finite **alphabet**
- $\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$ is a **transition function**
- $I \subseteq Q$ is a set of **initial states**
- $F \subseteq Q$ is a set of **accepting states**

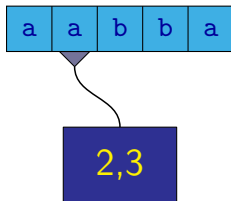
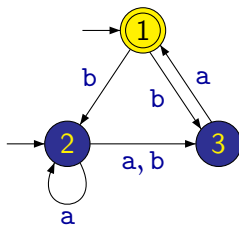
Transformation of NFA to DFA



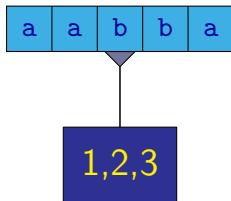
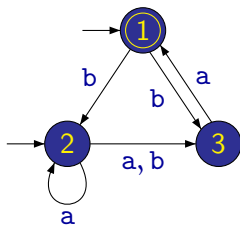
Transformation of NFA to DFA



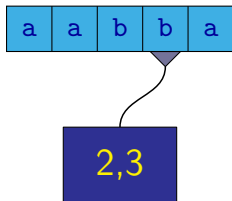
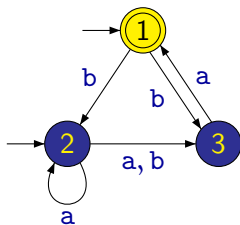
Transformation of NFA to DFA



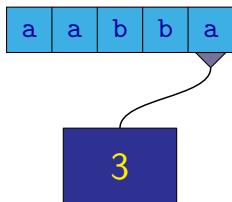
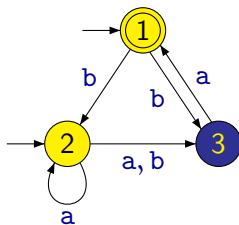
Transformation of NFA to DFA



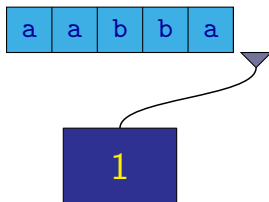
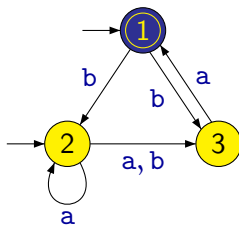
Transformation of NFA to DFA



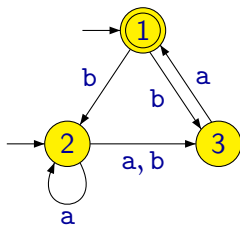
Transformation of NFA to DFA



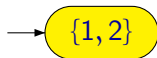
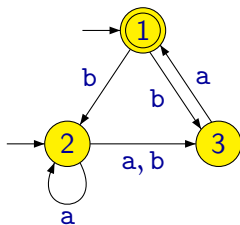
Transformation of NFA to DFA



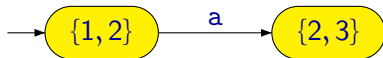
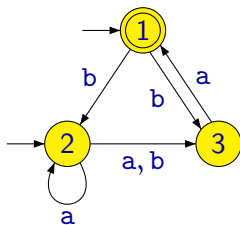
Transformation of NFA to DFA



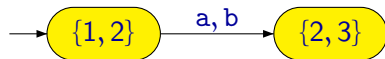
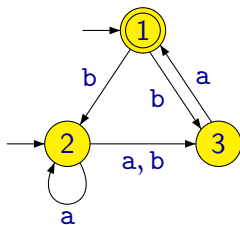
Transformation of NFA to DFA



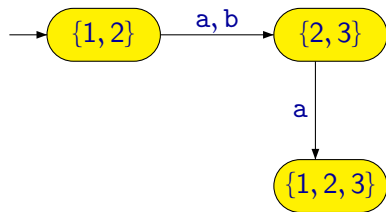
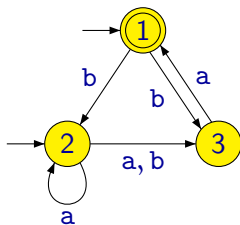
Transformation of NFA to DFA



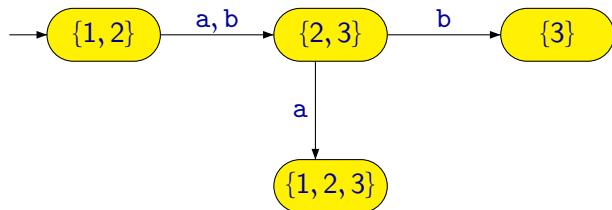
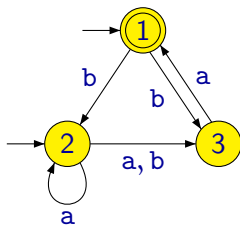
Transformation of NFA to DFA



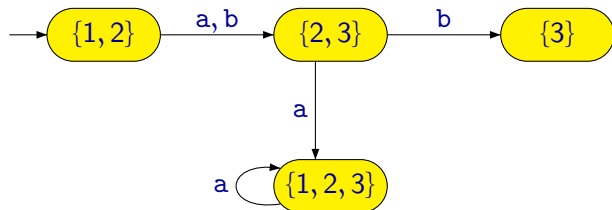
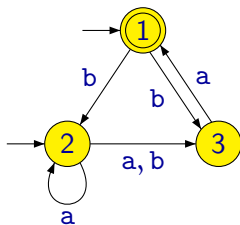
Transformation of NFA to DFA



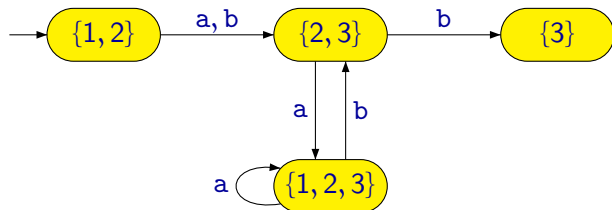
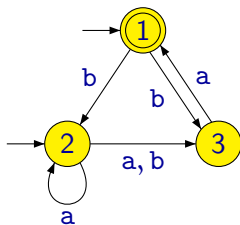
Transformation of NFA to DFA



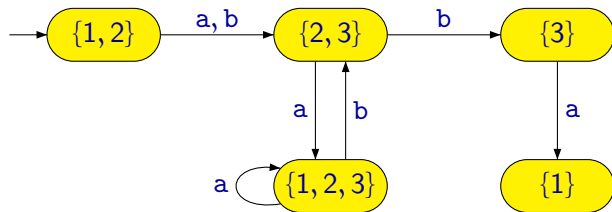
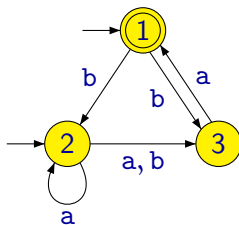
Transformation of NFA to DFA



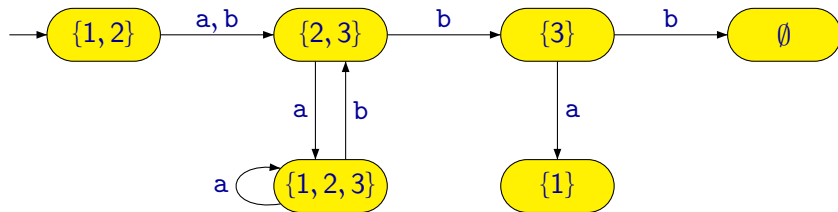
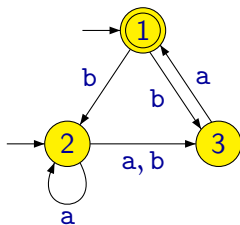
Transformation of NFA to DFA



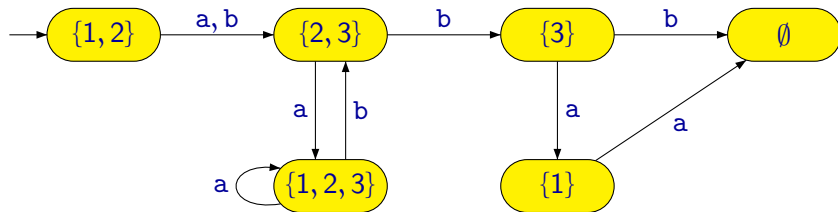
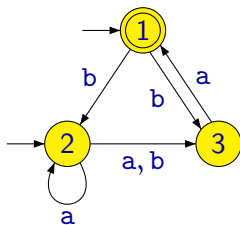
Transformation of NFA to DFA



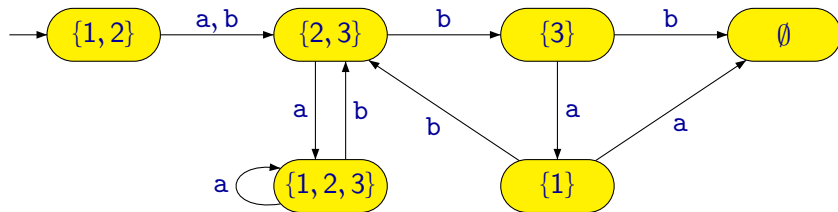
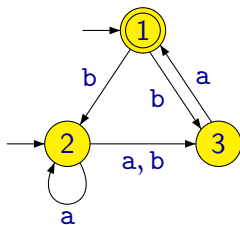
Transformation of NFA to DFA



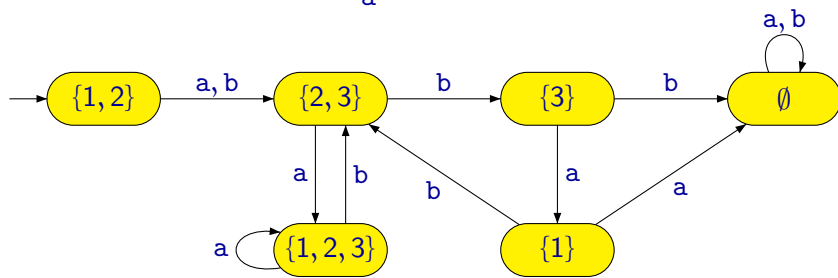
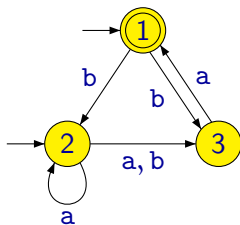
Transformation of NFA to DFA



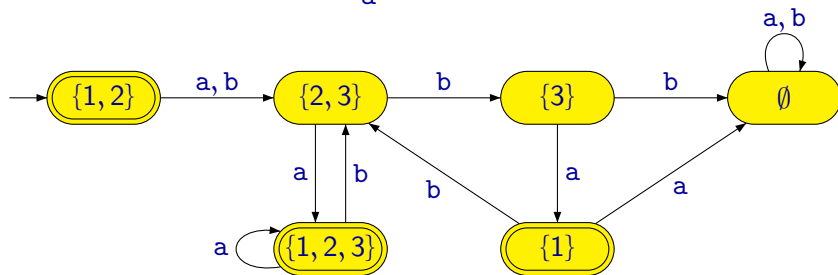
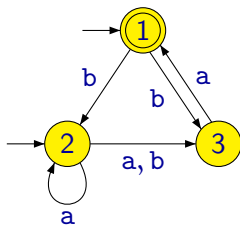
Transformation of NFA to DFA



Transformation of NFA to DFA



Transformation of NFA to DFA



Transformation of NFA to DFA

	a	b
$\leftrightarrow 1$	—	2, 3
$\rightarrow 2$	2, 3	3
3	1	—

Transformation of NFA to DFA

	a	b
$\leftrightarrow 1$	—	2,3
$\rightarrow 2$	2,3	3
3	1	—

	a	b

Transformation of NFA to DFA

	a	b
$\leftrightarrow 1$	—	2, 3
$\rightarrow 2$	2, 3	3
3	1	—

	a	b
$\leftrightarrow \{1, 2\}$		

Transformation of NFA to DFA

	a	b
$\leftrightarrow 1$	—	2, 3
$\rightarrow 2$	2, 3	3
3	1	—

	a	b
$\leftrightarrow \{1, 2\}$	$\{2, 3\}$	

Transformation of NFA to DFA

	a	b
$\leftrightarrow 1$	—	2,3
$\rightarrow 2$	2,3	3
3	1	—

	a	b
$\leftrightarrow \{1, 2\}$	$\{2, 3\}$	
$\{2, 3\}$		

Transformation of NFA to DFA

	a	b
$\leftrightarrow 1$	—	2, 3
$\rightarrow 2$	2, 3	3
3	1	—

	a	b
$\leftrightarrow \{1, 2\}$	$\{2, 3\}$	$\{2, 3\}$
$\{2, 3\}$		

Transformation of NFA to DFA

	a	b
$\leftrightarrow 1$	—	2, 3
$\rightarrow 2$	2, 3	3
3	1	—

	a	b
$\leftrightarrow \{1, 2\}$	$\{2, 3\}$	$\{2, 3\}$
$\{2, 3\}$	$\{1, 2, 3\}$	

Transformation of NFA to DFA

	a	b
$\leftrightarrow 1$	—	2, 3
$\rightarrow 2$	2, 3	3
3	1	—

	a	b
$\leftrightarrow \{1, 2\}$	$\{2, 3\}$	$\{2, 3\}$
$\{2, 3\}$	$\{1, 2, 3\}$	
$\leftarrow \{1, 2, 3\}$		

Transformation of NFA to DFA

	a	b
$\leftrightarrow 1$	—	2, 3
$\rightarrow 2$	2, 3	3
3	1	—

	a	b
$\leftrightarrow \{1, 2\}$	$\{2, 3\}$	$\{2, 3\}$
$\{2, 3\}$	$\{1, 2, 3\}$	$\{3\}$
$\leftarrow \{1, 2, 3\}$		

Transformation of NFA to DFA

	a	b
$\leftrightarrow 1$	—	2, 3
$\rightarrow 2$	2, 3	3
3	1	—

	a	b
$\leftrightarrow \{1, 2\}$	$\{2, 3\}$	$\{2, 3\}$
$\{2, 3\}$	$\{1, 2, 3\}$	$\{3\}$
$\leftarrow \{1, 2, 3\}$		
$\{3\}$		

Transformation of NFA to DFA

	a	b
$\leftrightarrow 1$	—	2, 3
$\rightarrow 2$	2, 3	3
3	1	—

	a	b
$\leftrightarrow \{1, 2\}$	$\{2, 3\}$	$\{2, 3\}$
$\{2, 3\}$	$\{1, 2, 3\}$	$\{3\}$
$\leftarrow \{1, 2, 3\}$	$\{1, 2, 3\}$	
$\{3\}$		

Transformation of NFA to DFA

	a	b
$\leftrightarrow 1$	—	2, 3
$\rightarrow 2$	2, 3	3
3	1	—

	a	b
$\leftrightarrow \{1, 2\}$	$\{2, 3\}$	$\{2, 3\}$
$\{2, 3\}$	$\{1, 2, 3\}$	$\{3\}$
$\leftarrow \{1, 2, 3\}$	$\{1, 2, 3\}$	$\{2, 3\}$
$\{3\}$		

Transformation of NFA to DFA

	a	b
$\leftrightarrow 1$	—	2, 3
$\rightarrow 2$	2, 3	3
3	1	—

	a	b
$\leftrightarrow \{1, 2\}$	$\{2, 3\}$	$\{2, 3\}$
$\{2, 3\}$	$\{1, 2, 3\}$	$\{3\}$
$\leftarrow \{1, 2, 3\}$	$\{1, 2, 3\}$	$\{2, 3\}$
$\{3\}$	$\{1\}$	

Transformation of NFA to DFA

	a	b
$\leftrightarrow 1$	—	2, 3
$\rightarrow 2$	2, 3	3
3	1	—

	a	b
$\leftrightarrow \{1, 2\}$	$\{2, 3\}$	$\{2, 3\}$
$\{2, 3\}$	$\{1, 2, 3\}$	$\{3\}$
$\leftarrow \{1, 2, 3\}$	$\{1, 2, 3\}$	$\{2, 3\}$
$\{3\}$	$\{1\}$	
$\leftarrow \{1\}$		

Transformation of NFA to DFA

	a	b
$\leftrightarrow 1$	—	2, 3
$\rightarrow 2$	2, 3	3
3	1	—

	a	b
$\leftrightarrow \{1, 2\}$	$\{2, 3\}$	$\{2, 3\}$
$\{2, 3\}$	$\{1, 2, 3\}$	$\{3\}$
$\leftarrow \{1, 2, 3\}$	$\{1, 2, 3\}$	$\{2, 3\}$
$\{3\}$	$\{1\}$	$\emptyset$
$\leftarrow \{1\}$		

Transformation of NFA to DFA

	a	b
$\leftrightarrow 1$	—	2, 3
$\rightarrow 2$	2, 3	3
3	1	—

	a	b
$\leftrightarrow \{1, 2\}$	$\{2, 3\}$	$\{2, 3\}$
$\{2, 3\}$	$\{1, 2, 3\}$	$\{3\}$
$\leftarrow \{1, 2, 3\}$	$\{1, 2, 3\}$	$\{2, 3\}$
$\{3\}$	$\{1\}$	$\emptyset$
$\leftarrow \{1\}$		
$\emptyset$		

Transformation of NFA to DFA

	a	b
$\leftrightarrow 1$	—	2, 3
$\rightarrow 2$	2, 3	3
3	1	—

	a	b
$\leftrightarrow \{1, 2\}$	$\{2, 3\}$	$\{2, 3\}$
$\{2, 3\}$	$\{1, 2, 3\}$	$\{3\}$
$\leftarrow \{1, 2, 3\}$	$\{1, 2, 3\}$	$\{2, 3\}$
$\{3\}$	$\{1\}$	$\emptyset$
$\leftarrow \{1\}$	$\emptyset$	
$\emptyset$		

Transformation of NFA to DFA

	a	b
$\leftrightarrow 1$	—	2, 3
$\rightarrow 2$	2, 3	3
3	1	—

	a	b
$\leftrightarrow \{1, 2\}$	$\{2, 3\}$	$\{2, 3\}$
$\{2, 3\}$	$\{1, 2, 3\}$	$\{3\}$
$\leftarrow \{1, 2, 3\}$	$\{1, 2, 3\}$	$\{2, 3\}$
$\{3\}$	$\{1\}$	$\emptyset$
$\leftarrow \{1\}$	$\emptyset$	$\{2, 3\}$
$\emptyset$		

Transformation of NFA to DFA

	a	b
$\leftrightarrow 1$	—	2, 3
$\rightarrow 2$	2, 3	3
3	1	—

	a	b
$\leftrightarrow \{1, 2\}$	$\{2, 3\}$	$\{2, 3\}$
$\{2, 3\}$	$\{1, 2, 3\}$	$\{3\}$
$\leftarrow \{1, 2, 3\}$	$\{1, 2, 3\}$	$\{2, 3\}$
$\{3\}$	$\{1\}$	$\emptyset$
$\leftarrow \{1\}$	$\emptyset$	$\{2, 3\}$
$\emptyset$	$\emptyset$	$\emptyset$

Transformation of NFA to DFA

	a	b
$\leftrightarrow 1$	—	2, 3
$\rightarrow 2$	2, 3	3
3	1	—

	a	b
$\leftrightarrow \{1, 2\}$	$\{2, 3\}$	$\{2, 3\}$
$\{2, 3\}$	$\{1, 2, 3\}$	$\{3\}$
$\leftarrow \{1, 2, 3\}$	$\{1, 2, 3\}$	$\{2, 3\}$
$\{3\}$	$\{1\}$	$\emptyset$
$\leftarrow \{1\}$	$\emptyset$	$\{2, 3\}$
$\emptyset$	$\emptyset$	$\emptyset$

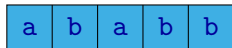
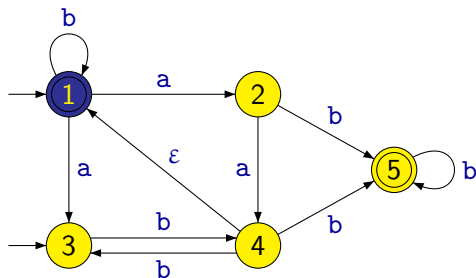
	a	b
$\leftrightarrow 1$	2	2
2	3	4
$\leftarrow 3$	3	2
4	5	6
$\leftarrow 5$	6	2
6	6	6

Remark: When a nondeterministic automaton with n states is transformed into a deterministic one, the resulting automaton can have 2^n states.

For example when we transform an automaton with 20 states, the resulting automaton can have $2^{20} = 1048576$ states.

It is often the case that the resulting automaton has far less than 2^n states. However, the worst cases are possible.

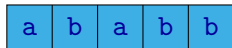
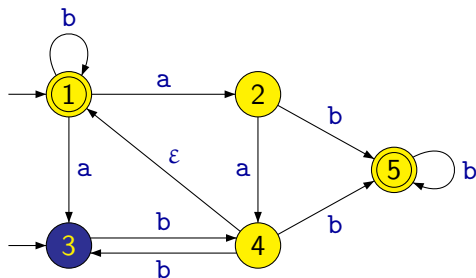
Generalized Nondeterministic Finite Automaton



1

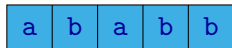
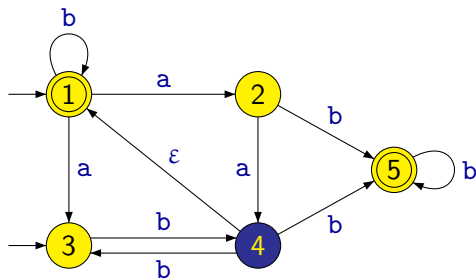


Generalized Nondeterministic Finite Automaton



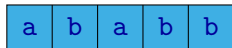
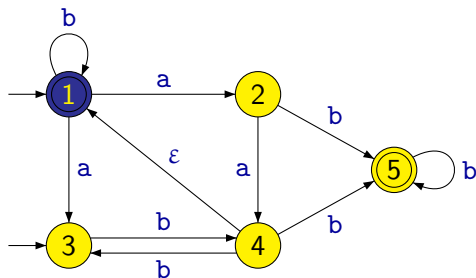
$1 \xrightarrow{a} 3$

Generalized Nondeterministic Finite Automaton



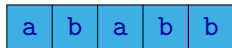
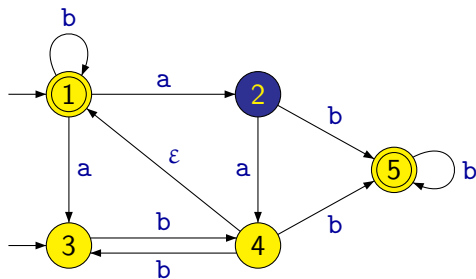
$$1 \xrightarrow{a} 3 \xrightarrow{b} 4$$

Generalized Nondeterministic Finite Automaton



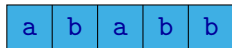
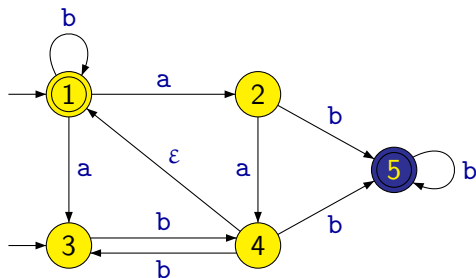
$$1 \xrightarrow{a} 3 \xrightarrow{b} 4 \xrightarrow{\epsilon} 1$$

Generalized Nondeterministic Finite Automaton



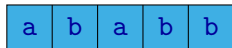
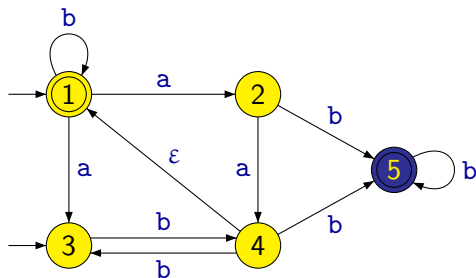
$$1 \xrightarrow{a} 3 \xrightarrow{b} 4 \xrightarrow{\epsilon} 1 \xrightarrow{a} 2$$

Generalized Nondeterministic Finite Automaton



$1 \xrightarrow{a} 3 \xrightarrow{b} 4 \xrightarrow{\epsilon} 1 \xrightarrow{a} 2 \xrightarrow{b} 5$

Generalized Nondeterministic Finite Automaton



$1 \xrightarrow{a} 3 \xrightarrow{b} 4 \xrightarrow{\epsilon} 1 \xrightarrow{a} 2 \xrightarrow{b} 5 \xrightarrow{b} 5$

Generalized Nondeterministic Finite Automaton

Compared to a nondeterministic finite automaton, a **generalized nondeterministic finite automaton** has the so called **ϵ -transitions**, i.e., transitions labelled with symbol ϵ .

When ϵ -transition is performed, only the state of the control unit is changed but the head on the tape is not moved.

Remark: The computations of a generalized nondeterministic automaton can be of an arbitrary length, even infinite (if the graph of the automaton contains a cycle consisting only of ϵ -transitions) regardless of the length of the word on the tape.

Generalized Nondeterministic Finite Automaton

Formally, a **generalized nondeterministic finite automaton** (**GNFA**) is defined as a tuple

$$(Q, \Sigma, \delta, I, F)$$

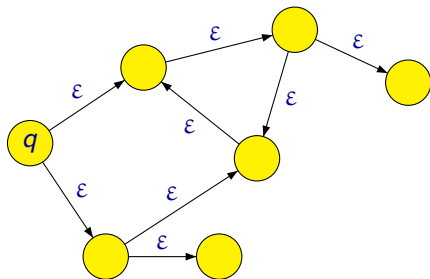
where:

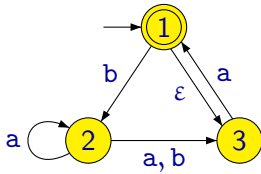
- Q is a finite set of **states**
- Σ is a finite **alphabet**
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(Q)$ is a **transition function**
- $I \subseteq Q$ is a set of **initial states**
- $F \subseteq Q$ is a set of **accepting states**

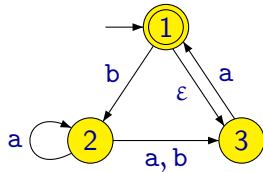
Remark: NFA can be viewed as a special case of GNFA, where $\delta(q, \varepsilon) = \emptyset$ for all $q \in Q$.

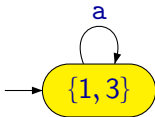
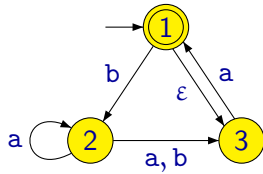
Transformation to a Deterministic Finite Automaton

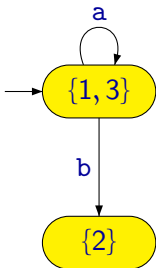
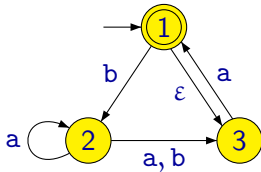
A generalized nondeterministic finite automaton can be transformed into a deterministic one using a similar construction as a nondeterministic finite automaton with the difference that we add to sets of states also all states that are reachable from already added states by some sequence of ϵ -transitions.

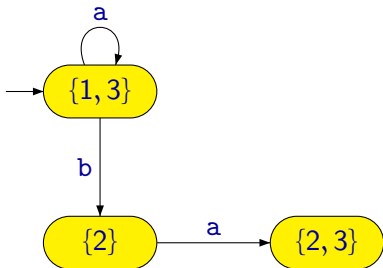
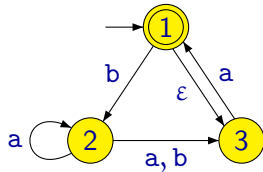


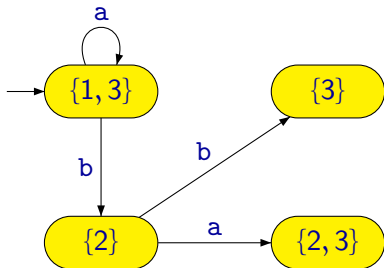
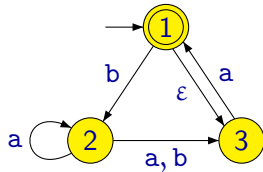


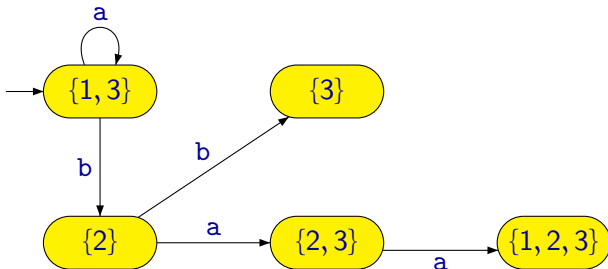
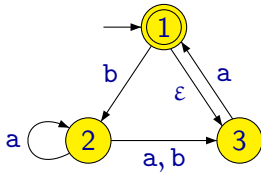


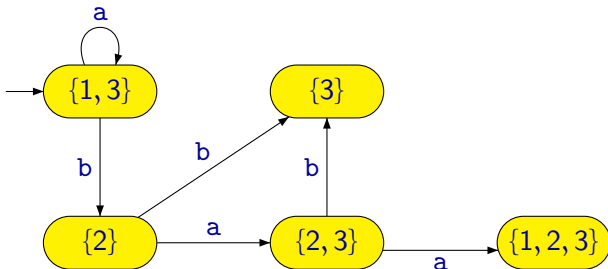
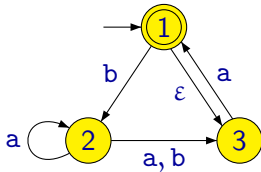


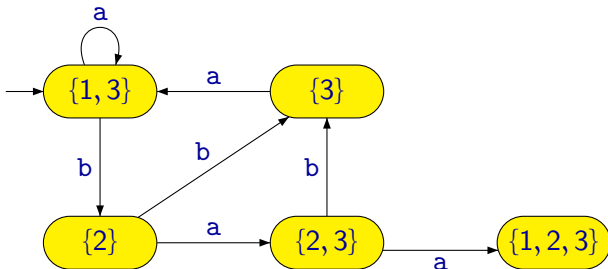
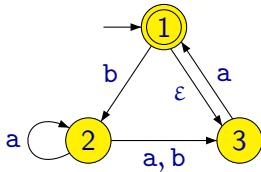


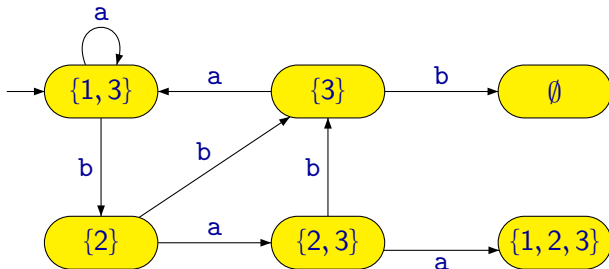
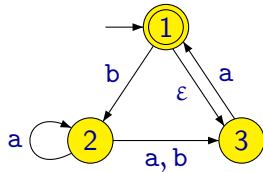


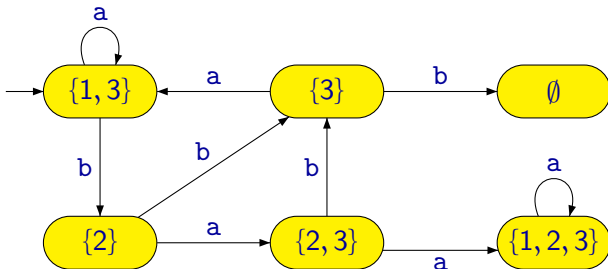
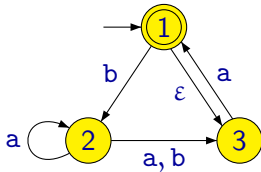


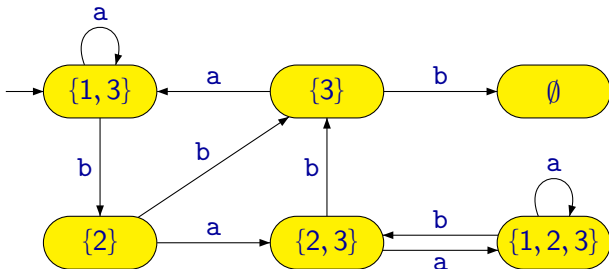
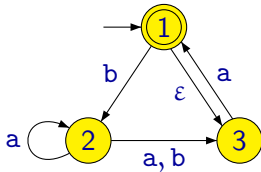


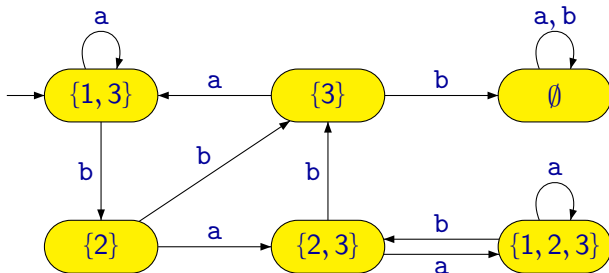
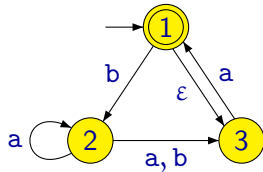


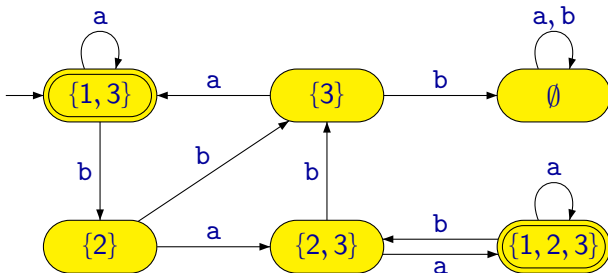
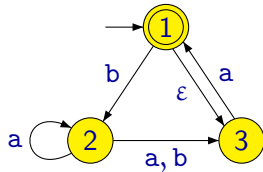












Transformation of GNFA to DFA

Before formally describing the transition of GNFA to DFA, let us introduce some auxiliary definitions.

Let us assume some given GNFA $A = (Q, \Sigma, \delta, I, F)$.

Let us define the function $\hat{\delta} : \mathcal{P}(Q) \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(Q)$ so that for $K \subseteq Q$ and $a \in \Sigma \cup \{\varepsilon\}$ there is

$$\hat{\delta}(K, a) = \bigcup_{q \in K} \delta(q, a)$$

Transformation of GNFA to DFA

For $K \subseteq Q$, let $Cl_\varepsilon(K)$ be the all states reachable from the states from the set K by some arbitrary sequence of ε -transitions.

This means that the function $Cl_\varepsilon : \mathcal{P}(Q) \rightarrow \mathcal{P}(Q)$ is defined so that for $K \subseteq Q$ is $Cl_\varepsilon(K)$ the smallest (with respect to inclusion) set satisfying the following two conditions:

- $K \subseteq Cl_\varepsilon(K)$
- For each $q \in Cl_\varepsilon(K)$ it holds that $\delta(q, \varepsilon) \subseteq Cl_\varepsilon(K)$.

Remark: Let us note that $Cl_\varepsilon(Cl_\varepsilon(K)) = Cl_\varepsilon(K)$ for arbitrary K .

Let us also note that in the case of NFA (where $\delta(q, \varepsilon) = \emptyset$ for each $q \in Q$) is $Cl_\varepsilon(K) = K$.

Transformation of GNFA to DFA

For a given GNFA $A = (Q, \Sigma, \delta, I, F)$ we can now construct DFA $A' = (Q', \Sigma, \delta', q'_0, F')$, where:

- $Q' = \mathcal{P}(Q)$ (so $K \in Q'$ means that $K \subseteq Q$)
- $\delta' : Q' \times \Sigma \rightarrow Q'$ is defined so that for $K \in Q'$ and $a \in \Sigma$:

$$\delta'(K, a) = Cl_\varepsilon(\hat{\delta}(Cl_\varepsilon(K), a))$$

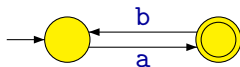
- $q'_0 = Cl_\varepsilon(I)$
- $F' = \{K \in Q' \mid Cl_\varepsilon(K) \cap F \neq \emptyset\}$

It is not difficult to verify that $L(A) = L(A')$.

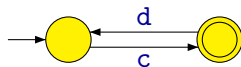
Concatenation of Languages

$$\Sigma = \{a, b, c, d\}$$

A_1 :



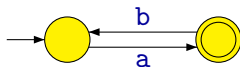
A_2 :



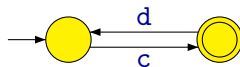
Concatenation of Languages

$$\Sigma = \{a, b, c, d\}$$

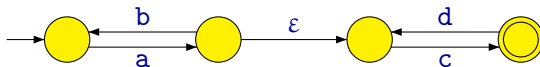
A_1 :



A_2 :



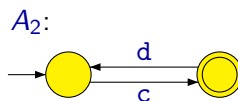
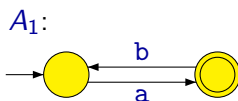
A :



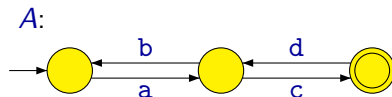
$$L(A) = L(A_1) \cdot L(A_2)$$

Concatenation of Languages

$$\Sigma = \{a, b, c, d\}$$

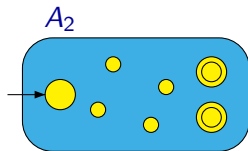
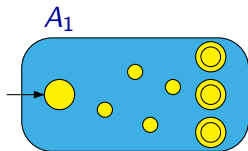


An incorrect construction:

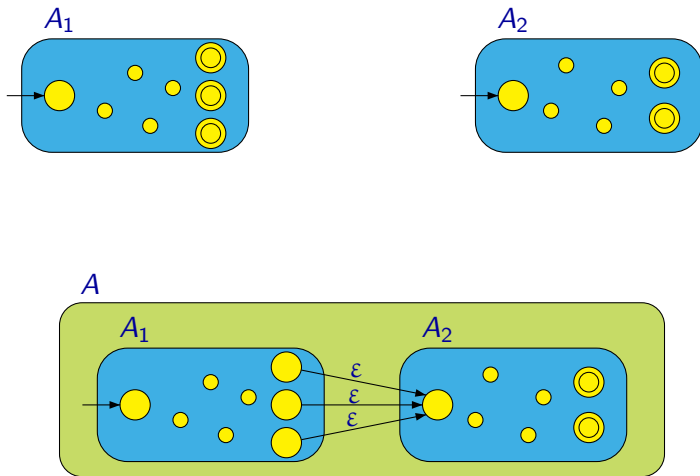


$acdbac \in L(A)$ but $acdbac \notin L(A_1) \cdot L(A_2)$

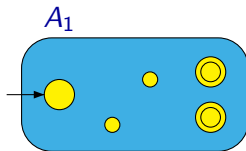
Concatenation of Languages



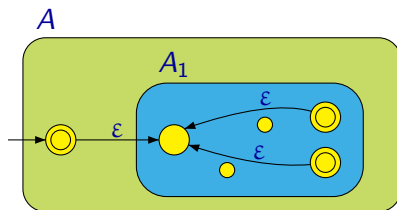
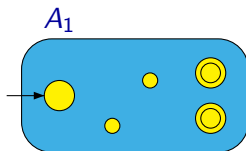
Concatenation of Languages



Iteration of a Language

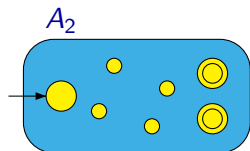
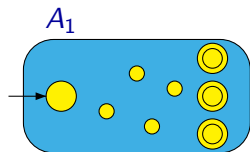


Iteration of a Language



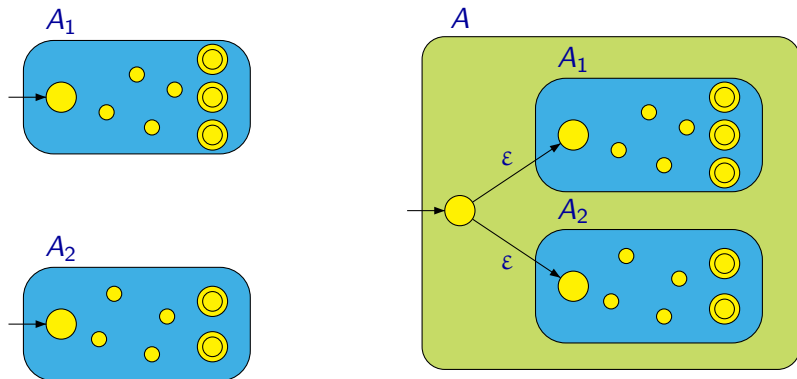
Union of Languages

An alternative construction for the union of languages:



Union of Languages

An alternative construction for the union of languages:



Closure Properties of the Class of Regular Languages

The set of (all) regular languages is closed with respect to:

- union
- intersection
- complement
- concatenation
- iteration
- . . .

Transformation of a Regular Expression to a Finite Automaton

Proposition

Every language that can be represented by a regular expression is regular (i.e., it is accepted by some finite automaton).

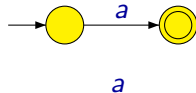
Proof: It is sufficient to show how to construct for a given regular expression α a finite automaton accepting the language $[\alpha]$.

The construction is recursive and proceeds by the structure of the expression α :

- If α is a elementary expression (i.e., $\emptyset$, ε or a):
 - We construct the corresponding automaton directly.
- If α is of the form $(\beta + \gamma)$, $(\beta \cdot \gamma)$ or (β^*) :
 - We construct automata accepting languages $[\beta]$ and $[\gamma]$ recursively.
 - Using these two automata, we construct the automaton accepting the language $[\alpha]$.

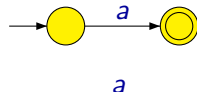
Transformation of a Regular Expression to a Finite Automaton

The automata for the elementary expressions:

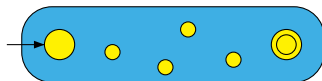
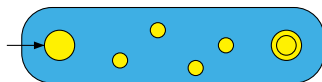


Transformation of a Regular Expression to a Finite Automaton

The automata for the elementary expressions:

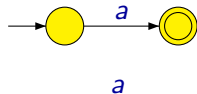


The construction for the union:

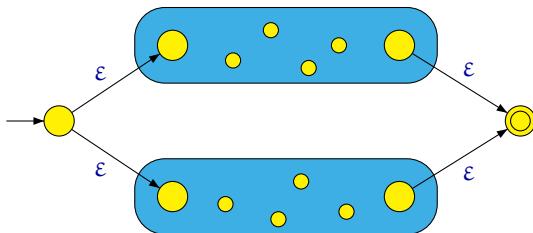


Transformation of a Regular Expression to a Finite Automaton

The automata for the elementary expressions:

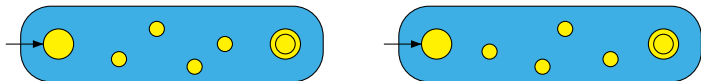


The construction for the union:



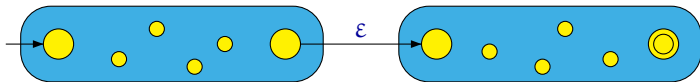
Transformation of a Regular Expression to a Finite Automaton

The construction for the concatenation:



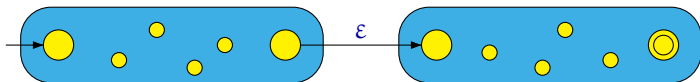
Transformation of a Regular Expression to a Finite Automaton

The construction for the concatenation:

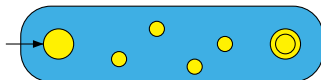


Transformation of a Regular Expression to a Finite Automaton

The construction for the concatenation:

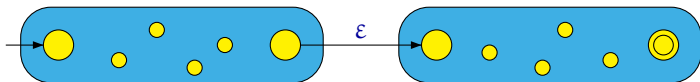


The construction for the iteration:

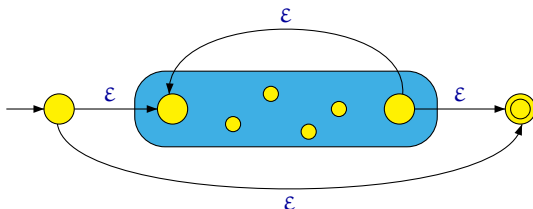


Transformation of a Regular Expression to a Finite Automaton

The construction for the concatenation:



The construction for the iteration:



Transformation of a Regular Expression to a Finite Automaton

Example: The construction of an automaton for expression $((0 + 1) \cdot 1)^*$:

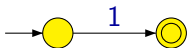
Transformation of a Regular Expression to a Finite Automaton

Example: The construction of an automaton for expression $((0 + 1) \cdot 1)^*$:



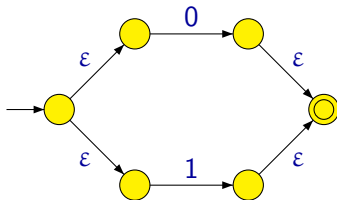
Transformation of a Regular Expression to a Finite Automaton

Example: The construction of an automaton for expression $((0 + 1) \cdot 1)^*$:



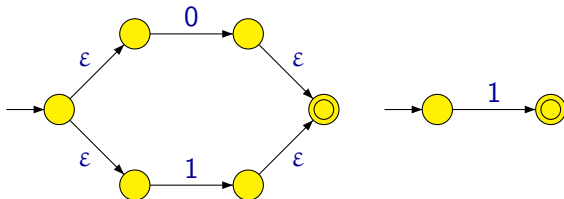
Transformation of a Regular Expression to a Finite Automaton

Example: The construction of an automaton for expression $((0 + 1) \cdot 1)^*$:



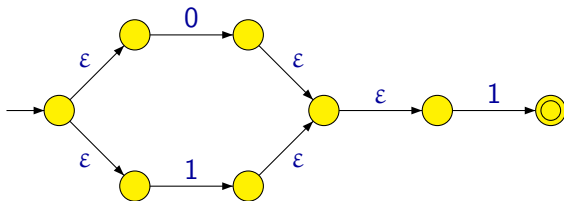
Transformation of a Regular Expression to a Finite Automaton

Example: The construction of an automaton for expression $((0 + 1) \cdot 1)^*$:



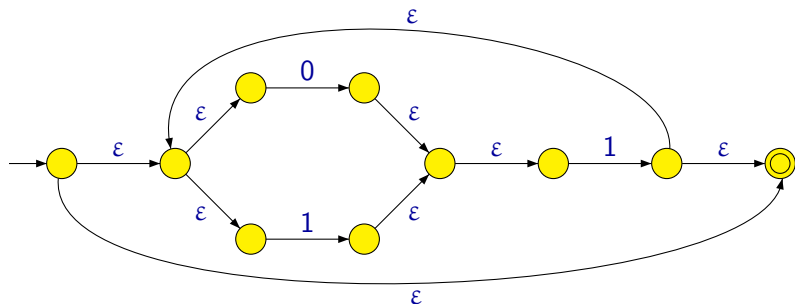
Transformation of a Regular Expression to a Finite Automaton

Example: The construction of an automaton for expression $((0 + 1) \cdot 1)^*$:



Transformation of a Regular Expression to a Finite Automaton

Example: The construction of an automaton for expression $((0 + 1) \cdot 1)^*$:



Transformation of a Regular Expression to a Finite Automaton

If an expression α consists of n symbols (not counting parenthesis) then the resulting automaton has:

- at most $2n$ states,
- at most $4n$ transitions.

Remark: By transforming the generalized nondeterministic automaton into a deterministic one, the number of states can grow exponentially, i.e., the resulting automaton can have up to $2^{2n} = 4^n$ states.

Transformation of an Automaton to a Regular Expression

Proposition

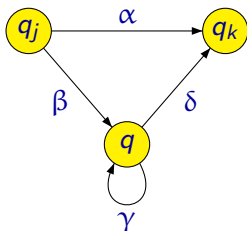
Every regular language can be represented by some regular expression.

Proof: It is sufficient to show how to construct for a given finite automaton A a regular expression α such that $[\alpha] = L(A)$.

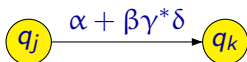
- We modify A in such a way that ensures it has exactly one initial and exactly one accepting state.
- Its states will be removed one by one.
- Its transitions will be labelled with regular expressions.
- The resulting automaton will have only two states – the initial and the accepting, and only one transition labelled with the resulting regular expression.

Transformation of an Automaton to a Regular Expression

The main idea: If a state q is removed, for every pair of remaining states q_j , q_k we extend the label on a transition from q_j to q_k by a regular expression representing paths from q_j to q_k going through q .

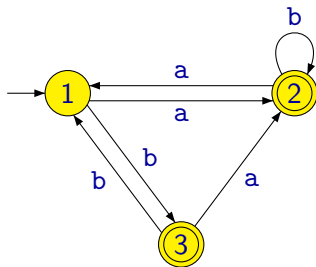


After removing of the state q :



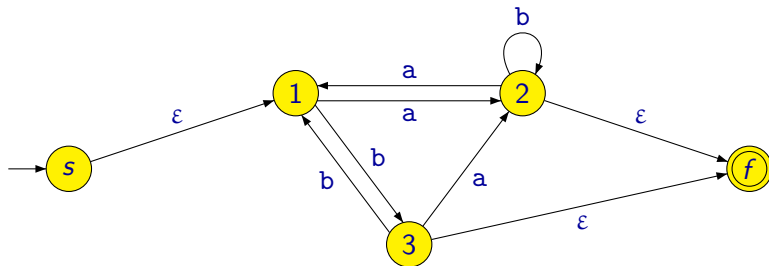
Transformation of an Automaton to a Regular Expression

Example:



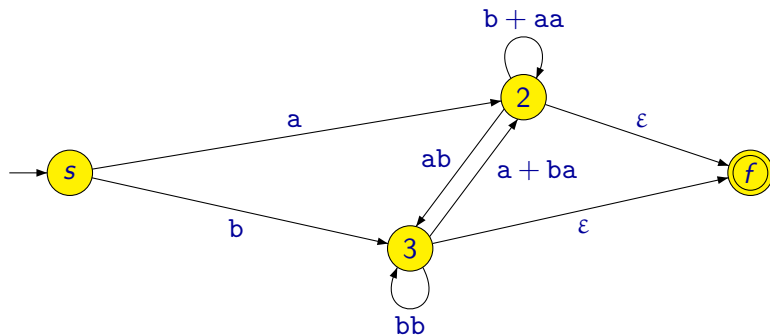
Transformation of an Automaton to a Regular Expression

Example:



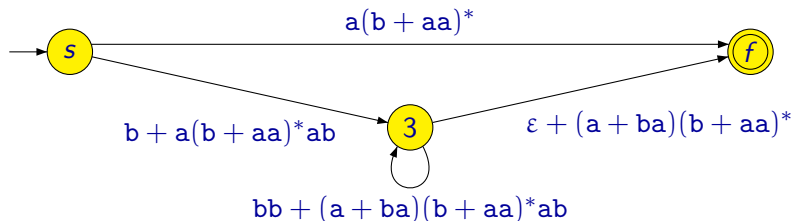
Transformation of an Automaton to a Regular Expression

Example:



Transformation of an Automaton to a Regular Expression

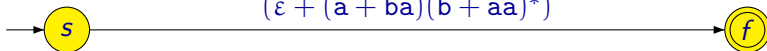
Example:



Transformation of an Automaton to a Regular Expression

Example:

$$\begin{aligned} & a(b + aa)^* + \\ & (b + a(b + aa)^* ab) \\ & (bb + (a + ba)(b + aa)^* ab)^* \\ & (\varepsilon + (a + ba)(b + aa)^*) \end{aligned}$$



Theorem

A language is regular iff it can be represented by a regular expression.

Context-Free Grammars

Example: We would like to describe a language of arithmetic expressions, containing expressions such as:

175 (9+15) (((10-4)*((1+34)+2))/(3+(-37)))

For simplicity we assume that:

- Expressions are fully parenthesized.
- The only arithmetic operations are “+”, “-”, “*”, “/” and unary “-”.
- Values of operands are natural numbers written in decimal — a number is represented as a non-empty sequence of digits.

Alphabet: $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, *, /, (,)\}$

Example (cont.): A description by an inductive definition:

- **Digit** is any of characters 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
- **Number** is a non-empty sequence of digits, i.e.:
 - If α is a digit then α is a number.
 - If α is a digit and β is a number then also $\alpha\beta$ is a number.
- **Expression** is a sequence of symbols constructed according to the following rules:
 - If α is a number then α is an expression.
 - If α is an expression then also $(-\alpha)$ is an expression.
 - If α and β are expressions then also $(\alpha+\beta)$ is an expression.
 - If α and β are expressions then also $(\alpha-\beta)$ is an expression.
 - If α and β are expressions then also $(\alpha*\beta)$ is an expression.
 - If α and β are expressions then also (α/β) is an expression.

Context-Free Grammars

Example (cont.): The same information that was described by the previous inductive definition can be represented by a **context-free grammar**:

New auxiliary symbols, called **nonterminals**, are introduced:

- D — stands for an arbitrary digit
- C — stands for an arbitrary number
- E — stands for an arbitrary expression

$$D \rightarrow 0$$

$$D \rightarrow 1$$

$$D \rightarrow 2$$

$$D \rightarrow 3$$

$$D \rightarrow 4$$

$$D \rightarrow 5$$

$$D \rightarrow 6$$

$$D \rightarrow 7$$

$$D \rightarrow 8$$

$$D \rightarrow 9$$

$$C \rightarrow D$$

$$C \rightarrow DC$$

$$E \rightarrow C$$

$$E \rightarrow (-E)$$

$$E \rightarrow (E+E)$$

$$E \rightarrow (E-E)$$

$$E \rightarrow (E * E)$$

$$E \rightarrow (E / E)$$

Example (cont.): Written in a more succinct way:

$$D \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

$$C \rightarrow D \mid DC$$

$$E \rightarrow C \mid (-E) \mid (E+E) \mid (E-E) \mid (E * E) \mid (E/E)$$

Example: A language where words are (possibly empty) sequences of expressions described in the previous example, where individual expressions are separated by commas (the alphabet must be extended with symbol “,”):

$$S \rightarrow T \mid \varepsilon$$

$$T \rightarrow E \mid E, T$$

$$D \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

$$C \rightarrow D \mid DC$$

$$E \rightarrow C \mid (-E) \mid (E+E) \mid (E-E) \mid (E * E) \mid (E/E)$$

Context-Free Grammars

Example: Statements of some programming language (a fragment of a grammar):

$$\begin{aligned} S &\rightarrow E; \mid T \mid \text{if } (E) \ S \mid \text{if } (E) \ S \text{ else } S \\ &\quad \mid \text{while } (E) \ S \mid \text{do } S \text{ while } (E); \mid \text{for } (F; F; F) \ S \\ &\quad \mid \text{return } F; \\ T &\rightarrow \{ \ U \} \\ U &\rightarrow \varepsilon \mid SU \\ F &\rightarrow \varepsilon \mid E \\ E &\rightarrow \dots \end{aligned}$$

Remark:

- S — statement
- T — block of statements
- U — sequence of statements
- E — expression
- F — optional expression that can be omitted

Context-Free Grammars

Formally, a **context-free grammar** is a tuple

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

where:

- Π is a finite set of **nonterminal symbols** (**nonterminals**)
- Σ is a finite set of **terminal symbols** (**terminals**),
where $\Pi \cap \Sigma = \emptyset$
- $S \in \Pi$ is an **initial nonterminal**
- $P \subseteq \Pi \times (\Pi \cup \Sigma)^*$ is a finite set of **rewrite rules**

Remarks:

- We will use uppercase letters $A, B, C, \dots$ to denote nonterminal symbols.
- We will use lowercase letters $a, b, c, \dots$ or digits $0, 1, 2, \dots$ to denote terminal symbols.
- We will use lowercase Greek letters $\alpha, \beta, \gamma, \dots$ to denote strings from $(\Pi \cup \Sigma)^*$.
- We will use the following notation for rules instead of (A, α)

$$A \rightarrow \alpha$$

A – left-hand side of the rule

α – right-hand side of the rule

Example: Grammar $G = (\Pi, \Sigma, S, P)$ where

- $\Pi = \{A, B, C\}$
- $\Sigma = \{a, b\}$
- $S = A$
- P contains rules

$$A \rightarrow aBBb$$

$$A \rightarrow AaA$$

$$B \rightarrow \varepsilon$$

$$B \rightarrow bCA$$

$$C \rightarrow AB$$

$$C \rightarrow a$$

$$C \rightarrow b$$

Remark: If we have more rules with the same left-hand side, as for example

$$A \rightarrow \alpha_1$$

$$A \rightarrow \alpha_2$$

$$A \rightarrow \alpha_3$$

we can write them in a more succinct way as

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \alpha_3$$

For example, the rules of the grammar from the previous slide can be written as

$$A \rightarrow aBBb \mid AaA$$

$$B \rightarrow \varepsilon \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

Context-Free Grammars

Grammars are used for generating words.

Example: $G = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$A \rightarrow aBBb \mid AaA$$

$$B \rightarrow \varepsilon \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

For example, the word *abbabb* can be in grammar G generated as follows:

Context-Free Grammars

Grammars are used for generating words.

Example: $G = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$A \rightarrow aBBb \mid AaA$$

$$B \rightarrow \varepsilon \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

For example, the word *abbabb* can be in grammar G generated as follows:

A

Context-Free Grammars

Grammars are used for generating words.

Example: $G = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$\underline{A} \rightarrow aBBb \mid AaA$$

$$B \rightarrow \varepsilon \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

For example, the word *abbabb* can be in grammar G generated as follows:

A

Context-Free Grammars

Grammars are used for generating words.

Example: $G = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$\underline{A} \rightarrow \underline{aBBb} \mid AaA$$

$$B \rightarrow \varepsilon \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

For example, the word $\underline{abbabb}$ can be in grammar G generated as follows:

$$\underline{A} \Rightarrow \underline{aBBb}$$

Context-Free Grammars

Grammars are used for generating words.

Example: $G = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$A \rightarrow aBBb \mid AaA$$

$$B \rightarrow \varepsilon \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

For example, the word *abbabb* can be in grammar G generated as follows:

$$A \Rightarrow aBBb$$

Context-Free Grammars

Grammars are used for generating words.

Example: $G = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$A \rightarrow aBBb \mid AaA$$

$$\underline{B} \rightarrow \varepsilon \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

For example, the word $abbabb$ can be in grammar G generated as follows:

$$A \Rightarrow a\underline{B}Bb$$

Context-Free Grammars

Grammars are used for generating words.

Example: $G = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$A \rightarrow aBBb \mid AaA$$

$$\underline{B} \rightarrow \varepsilon \mid \underline{bCA}$$

$$C \rightarrow AB \mid a \mid b$$

For example, the word $abbabb$ can be in grammar G generated as follows:

$$A \Rightarrow a\underline{B}Bb \Rightarrow a\underline{bCA}Bb$$

Context-Free Grammars

Grammars are used for generating words.

Example: $G = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$A \rightarrow aBBb \mid AaA$$

$$B \rightarrow \varepsilon \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

For example, the word *abbabb* can be in grammar G generated as follows:

$$A \Rightarrow aBBb \Rightarrow abCABb$$

Context-Free Grammars

Grammars are used for generating words.

Example: $G = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$\underline{A} \rightarrow aBBb \mid AaA$$

$$B \rightarrow \varepsilon \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

For example, the word *abbabb* can be in grammar G generated as follows:

$$A \Rightarrow aBBb \Rightarrow abC\underline{A}Bb$$

Context-Free Grammars

Grammars are used for generating words.

Example: $G = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$\underline{A} \rightarrow \underline{aBBb} \mid AaA$$

$$B \rightarrow \varepsilon \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

For example, the word *abbabb* can be in grammar G generated as follows:

$$A \Rightarrow aBBb \Rightarrow abC\underline{A}Bb \Rightarrow abC\underline{aBBb}Bb$$

Context-Free Grammars

Grammars are used for generating words.

Example: $G = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$A \rightarrow aBBb \mid AaA$$

$$B \rightarrow \varepsilon \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

For example, the word *abbabb* can be in grammar G generated as follows:

$$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb$$

Context-Free Grammars

Grammars are used for generating words.

Example: $G = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$A \rightarrow aBBb \mid AaA$$

$$\underline{B} \rightarrow \varepsilon \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

For example, the word *abbabb* can be in grammar G generated as follows:

$$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaB\underline{B}bBb$$

Context-Free Grammars

Grammars are used for generating words.

Example: $G = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$A \rightarrow aBBb \mid AaA$$

$$\underline{B} \rightarrow \underline{\varepsilon} \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

For example, the word *abbabb* can be in grammar G generated as follows:

$$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaB\underline{B}bBb \Rightarrow abCaBbBb$$

Context-Free Grammars

Grammars are used for generating words.

Example: $G = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$A \rightarrow aBBb \mid AaA$$

$$B \rightarrow \varepsilon \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

For example, the word *abbabb* can be in grammar G generated as follows:

$$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb$$

Context-Free Grammars

Grammars are used for generating words.

Example: $G = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$A \rightarrow aBBb \mid AaA$$

$$B \rightarrow \varepsilon \mid bCA$$

$$\underline{C} \rightarrow AB \mid a \mid b$$

For example, the word *abbabb* can be in grammar G generated as follows:

$$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow ab\underline{C}aBbBb$$

Context-Free Grammars

Grammars are used for generating words.

Example: $G = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$A \rightarrow aBBb \mid AaA$$

$$B \rightarrow \varepsilon \mid bCA$$

$$\underline{C} \rightarrow AB \mid a \mid \underline{b}$$

For example, the word *abbabb* can be in grammar G generated as follows:

$$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow ab\underline{C}aBbBb \Rightarrow ab\underline{b}aBbBb$$

Context-Free Grammars

Grammars are used for generating words.

Example: $G = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$A \rightarrow aBBb \mid AaA$$

$$B \rightarrow \varepsilon \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

For example, the word *abbabb* can be in grammar G generated as follows:

$$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb$$

Context-Free Grammars

Grammars are used for generating words.

Example: $G = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$A \rightarrow aBBb \mid AaA$$

$$\underline{B} \rightarrow \varepsilon \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

For example, the word *abbabb* can be in grammar G generated as follows:

$$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBb\underline{B}b$$

Context-Free Grammars

Grammars are used for generating words.

Example: $G = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$A \rightarrow aBBb \mid AaA$$

$$\underline{B} \rightarrow \underline{\varepsilon} \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

For example, the word *abbabb* can be in grammar G generated as follows:

$$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBb\underline{B}b \Rightarrow abbaBbb$$

Context-Free Grammars

Grammars are used for generating words.

Example: $G = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$A \rightarrow aBBb \mid AaA$$

$$B \rightarrow \varepsilon \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

For example, the word *abbabb* can be in grammar G generated as follows:

$$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb \Rightarrow abbaBbb$$

Context-Free Grammars

Grammars are used for generating words.

Example: $G = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$A \rightarrow aBBb \mid AaA$$

$$\underline{B} \rightarrow \varepsilon \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

For example, the word *abbabb* can be in grammar G generated as follows:

$$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb \Rightarrow abba\underline{B}bb$$

Context-Free Grammars

Grammars are used for generating words.

Example: $G = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$A \rightarrow aBBb \mid AaA$$

$$\underline{B} \rightarrow \underline{\varepsilon} \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

For example, the word *abbabb* can be in grammar G generated as follows:

$$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb \Rightarrow abba\underline{B}bb \Rightarrow abbabb$$

Context-Free Grammars

Grammars are used for generating words.

Example: $G = (\Pi, \Sigma, A, P)$ where $\Pi = \{A, B, C\}$, $\Sigma = \{a, b\}$, and P contains rules

$$A \rightarrow aBBb \mid AaA$$

$$B \rightarrow \varepsilon \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

For example, the word *abbabb* can be in grammar G generated as follows:

$$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb \Rightarrow abbaBbb \Rightarrow abbabb$$

Context-Free Grammars

On strings from $(\Pi \cup \Sigma)^*$ we define relation $\Rightarrow \subseteq (\Pi \cup \Sigma)^* \times (\Pi \cup \Sigma)^*$ such that

$$\alpha \Rightarrow \alpha'$$

iff $\alpha = \beta_1 A \beta_2$ and $\alpha' = \beta_1 \gamma \beta_2$ for some $\beta_1, \beta_2, \gamma \in (\Pi \cup \Sigma)^*$ and $A \in \Pi$ where $(A \rightarrow \gamma) \in P$.

Example: If $(B \rightarrow bCA) \in P$ then

$$aCBbA \Rightarrow aCbCAbA$$

Remark: Informally, $\alpha \Rightarrow \alpha'$ means that it is possible to derive α' from α by one step where an occurrence of some nonterminal A in α is replaced with the right-hand side of some rule $A \rightarrow \gamma$ with A on the left-hand side.

Context-Free Grammars

On strings from $(\Pi \cup \Sigma)^*$ we define relation $\Rightarrow \subseteq (\Pi \cup \Sigma)^* \times (\Pi \cup \Sigma)^*$ such that

$$\alpha \Rightarrow \alpha'$$

iff $\alpha = \beta_1 A \beta_2$ and $\alpha' = \beta_1 \gamma \beta_2$ for some $\beta_1, \beta_2, \gamma \in (\Pi \cup \Sigma)^*$ and $A \in \Pi$ where $(A \rightarrow \gamma) \in P$.

Example: If $(B \rightarrow bCA) \in P$ then

$$aC\underline{B}bA \Rightarrow aC\underline{bCA}bA$$

Remark: Informally, $\alpha \Rightarrow \alpha'$ means that it is possible to derive α' from α by one step where an occurrence of some nonterminal A in α is replaced with the right-hand side of some rule $A \rightarrow \gamma$ with A on the left-hand side.

Context-Free Grammars

A **derivation** of length n is a sequence $\beta_0, \beta_1, \beta_2, \dots, \beta_n$, where $\beta_i \in (\Pi \cup \Sigma)^*$, and where $\beta_{i-1} \Rightarrow \beta_i$ for all $1 \leq i \leq n$, which can be written more succinctly as

$$\beta_0 \Rightarrow \beta_1 \Rightarrow \beta_2 \Rightarrow \dots \Rightarrow \beta_{n-1} \Rightarrow \beta_n$$

The fact that for given $\alpha, \alpha' \in (\Pi \cup \Sigma)^*$ and $n \in \mathbb{N}$ there exists some derivation $\beta_0 \Rightarrow \beta_1 \Rightarrow \beta_2 \Rightarrow \dots \Rightarrow \beta_{n-1} \Rightarrow \beta_n$, where $\alpha = \beta_0$ and $\alpha' = \beta_n$, is denoted

$$\alpha \Rightarrow^n \alpha'$$

The fact that $\alpha \Rightarrow^n \alpha'$ for some $n \geq 0$, is denoted

$$\alpha \Rightarrow^* \alpha'$$

Remark: Relation $\Rightarrow^*$ is the reflexive and transitive closure of relation $\Rightarrow$ (i.e., the smallest reflexive and transitive relation containing relation $\Rightarrow$).

Sentential forms are those $\alpha \in (\Pi \cup \Sigma)^*$, for which

$$S \Rightarrow^* \alpha$$

where S is the initial nonterminal.

A **language** $L(G)$ generated by a grammar $G = (\Pi, \Sigma, S, P)$ is the set of all words over alphabet Σ that can be derived by some derivation from the initial nonterminal S using rules from P , i.e.,

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

Example: We want to construct a grammar generating the language

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

Example: We want to construct a grammar generating the language

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

Grammar $G = (\Pi, \Sigma, S, P)$ where $\Pi = \{S\}$, $\Sigma = \{a, b\}$, and P contains

$$S \rightarrow aSb \mid \varepsilon$$

Example: We want to construct a grammar generating the language

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

Grammar $G = (\Pi, \Sigma, S, P)$ where $\Pi = \{S\}$, $\Sigma = \{a, b\}$, and P contains

$$S \rightarrow aSb \mid \varepsilon$$

$$S \Rightarrow \varepsilon$$

$$S \Rightarrow aSb \Rightarrow ab$$

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$$

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

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaaaSbbbb \Rightarrow aaaabbbb$$

...

Example: We want to construct a grammar generating the language consisting of all palindroms over the alphabet $\{a, b\}$, i.e.,

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

Remark: w^R denotes the **reverse** of a word w , i.e., the word w written backwards.

Example: We want to construct a grammar generating the language consisting of all palindroms over the alphabet $\{a, b\}$, i.e.,

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

Remark: w^R denotes the **reverse** of a word w , i.e., the word w written backwards.

Solution:

$$S \rightarrow aSa \mid bSb \mid a \mid b \mid \varepsilon$$

Example: We want to construct a grammar generating the language consisting of all palindroms over the alphabet $\{a, b\}$, i.e.,

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

Remark: w^R denotes the **reverse** of a word w , i.e., the word w written backwards.

Solution:

$$S \rightarrow aSa \mid bSb \mid a \mid b \mid \varepsilon$$

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abaSaba \Rightarrow abaaaba$$

Example: We want to construct a grammar generating the language L consisting of all correctly parenthesised sequences of symbols '(' and ')'.
For example $((()())((() \in L$ but $)() \notin L$.

Example: We want to construct a grammar generating the language L consisting of all correctly parenthesised sequences of symbols '(' and ')'. For example $((())()) \in L$ but $)() \notin L$.

Solution:

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

Example: We want to construct a grammar generating the language L consisting of all correctly parenthesised sequences of symbols '(' and ')'. For example $((())()) \in L$ but $)() \notin L$.

Solution:

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

$$\begin{aligned} S &\Rightarrow SS \Rightarrow (S)S \Rightarrow (S)(S) \Rightarrow (SS)(S) \Rightarrow ((S)S)(S) \Rightarrow \\ &((()S)(S) \Rightarrow (() (S))(S) \Rightarrow (() () (S) \Rightarrow (() () ((S)) \Rightarrow \\ &(() () ()) \end{aligned}$$

Example: We want to construct a grammar generating the language L consisting of all correctly constructed arithmetic expressions where operands are always of the form ' a ' and where symbols $+$ and $*$ can be used as operators.

For example $(a + a) * a + (a * a) \in L$.

Example: We want to construct a grammar generating the language L consisting of all correctly constructed arithmetic expressions where operands are always of the form ' a ' and where symbols $+$ and $*$ can be used as operators.

For example $(a + a) * a + (a * a) \in L$.

Solution:

$$E \rightarrow a \mid E + E \mid E * E \mid (E)$$

Example: We want to construct a grammar generating the language L consisting of all correctly constructed arithmetic expressions where operands are always of the form ' a ' and where symbols $+$ and $*$ can be used as operators.

For example $(a + a) * a + (a * a) \in L$.

Solution:

$$E \rightarrow a \mid E + E \mid E * E \mid (E)$$

$$\begin{aligned} E &\Rightarrow E + E \Rightarrow E * E + E \Rightarrow (E) * E + E \Rightarrow (E + E) * E + E \Rightarrow \\ &(a + E) * E + E \Rightarrow (a + a) * E + E \Rightarrow (a + a) * a + E \Rightarrow (a + a) * a + (E) \Rightarrow \\ &(a + a) * a + (E * E) \Rightarrow (a + a) * a + (a * E) \Rightarrow (a + a) * a + (a * a) \end{aligned}$$

Derivation Tree

$A \rightarrow aBBb \mid AaA$

$B \rightarrow \varepsilon \mid bCA$

$C \rightarrow AB \mid a \mid b$

A

$$A \rightarrow aBBb \mid AaA$$

$$B \rightarrow \varepsilon \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

A

A

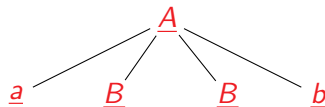
A $\rightarrow aBBb \mid AaA$

$B \rightarrow \varepsilon \mid bCA$

$C \rightarrow AB \mid a \mid b$

A

Derivation Tree



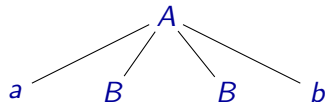
$\underline{A} \rightarrow \underline{aBBb} \mid AaA$

$B \rightarrow \varepsilon \mid bCA$

$C \rightarrow AB \mid a \mid b$

$\underline{A} \Rightarrow \underline{aBBb}$

Derivation Tree



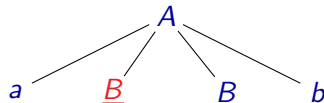
$$A \rightarrow aBBb \mid AaA$$

$$B \rightarrow \varepsilon \mid bCA$$

$$C \rightarrow AB \mid a \mid b$$

$$A \Rightarrow aBBb$$

Derivation Tree



$A \rightarrow aBBb \mid AaA$

$\underline{B} \rightarrow \varepsilon \mid bCA$

$C \rightarrow AB \mid a \mid b$

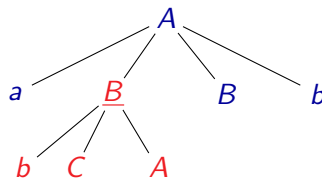
$A \Rightarrow a\underline{B}Bb$

Derivation Tree

$$A \rightarrow aBBb \mid AaA$$

$$\underline{B} \rightarrow \varepsilon \mid \underline{bCA}$$

$$C \rightarrow AB \mid a \mid b$$



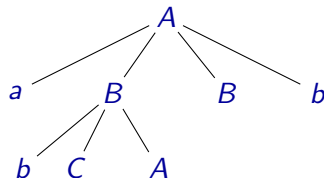
$$A \Rightarrow a\underline{B}Bb \Rightarrow a\underline{bCA}Bb$$

Derivation Tree

$A \rightarrow aBBb \mid AaA$

$B \rightarrow \varepsilon \mid bCA$

$C \rightarrow AB \mid a \mid b$



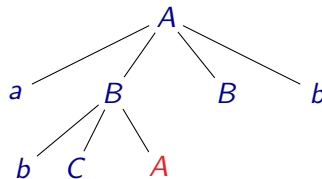
$A \Rightarrow aBBb \Rightarrow abCABb$

Derivation Tree

A $\rightarrow aBBb \mid AaA$

$B \rightarrow \varepsilon \mid bCA$

$C \rightarrow AB \mid a \mid b$



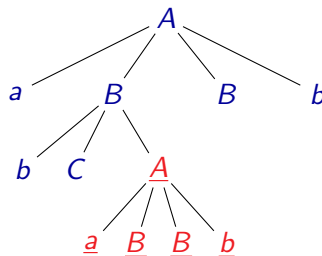
$A \Rightarrow aBBb \Rightarrow abC\underline{A}Bb$

Derivation Tree

$\underline{A} \rightarrow \underline{aBBb} \mid AaA$

$B \rightarrow \varepsilon \mid bCA$

$C \rightarrow AB \mid a \mid b$



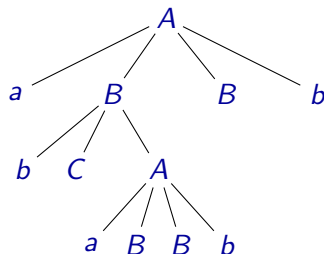
$A \Rightarrow aBBb \Rightarrow abC\underline{A}Bb \Rightarrow abC\underline{aBBb}Bb$

Derivation Tree

$A \rightarrow aBBb \mid AaA$

$B \rightarrow \varepsilon \mid bCA$

$C \rightarrow AB \mid a \mid b$



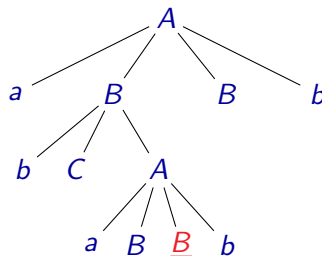
$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb$

Derivation Tree

$A \rightarrow aBBb \mid AaA$

$\underline{B} \rightarrow \varepsilon \mid bCA$

$C \rightarrow AB \mid a \mid b$



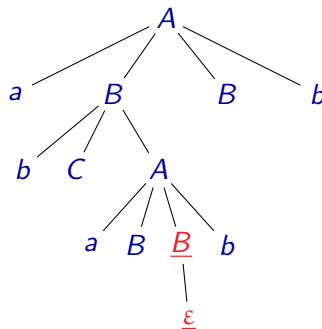
$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaB\underline{B}bBb$

Derivation Tree

$A \rightarrow aBBb \mid AaA$

$\underline{B} \rightarrow \underline{\epsilon} \mid bCA$

$C \rightarrow AB \mid a \mid b$



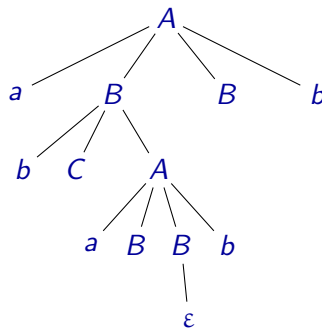
$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaB\underline{B}bBb \Rightarrow abCaBbBb$

Derivation Tree

$A \rightarrow aBBb \mid AaA$

$B \rightarrow \varepsilon \mid bCA$

$C \rightarrow AB \mid a \mid b$



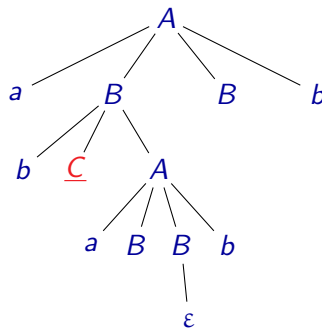
$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb$

Derivation Tree

$A \rightarrow aBBb \mid AaA$

$B \rightarrow \varepsilon \mid bCA$

$\underline{C} \rightarrow AB \mid a \mid b$



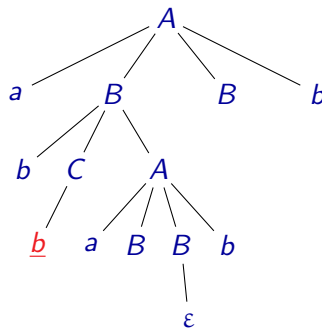
$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow ab\underline{C}aBbBb$

Derivation Tree

$A \rightarrow aBBb \mid AaA$

$B \rightarrow \varepsilon \mid bCA$

$\underline{C} \rightarrow AB \mid a \mid \underline{b}$



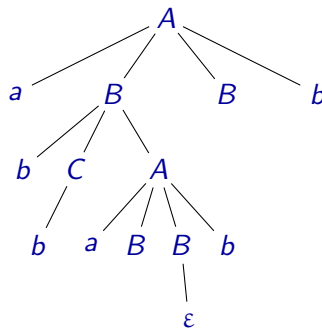
$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow ab\underline{C}aBbBb \Rightarrow ab\underline{b}aBbBb$

Derivation Tree

$A \rightarrow aBBb \mid AaA$

$B \rightarrow \varepsilon \mid bCA$

$C \rightarrow AB \mid a \mid b$



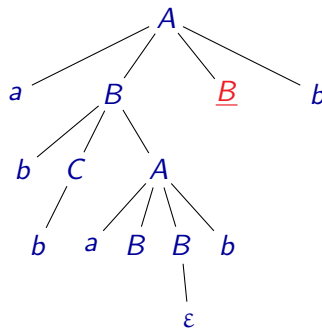
$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb$

Derivation Tree

$A \rightarrow aBBb \mid AaA$

$\underline{B} \rightarrow \varepsilon \mid bCA$

$C \rightarrow AB \mid a \mid b$



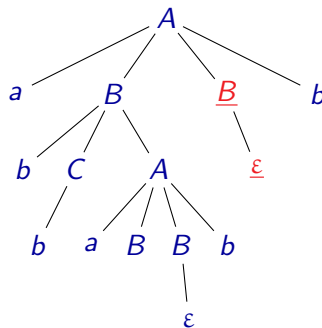
$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBb\underline{B}b$

Derivation Tree

$A \rightarrow aBBb \mid AaA$

$\underline{B} \rightarrow \underline{\epsilon} \mid bCA$

$C \rightarrow AB \mid a \mid b$



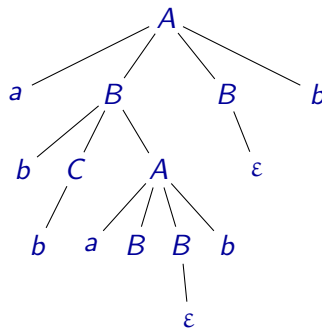
$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBb\underline{B}b \Rightarrow abbaBbb$

Derivation Tree

$A \rightarrow aBBb \mid AaA$

$B \rightarrow \varepsilon \mid bCA$

$C \rightarrow AB \mid a \mid b$



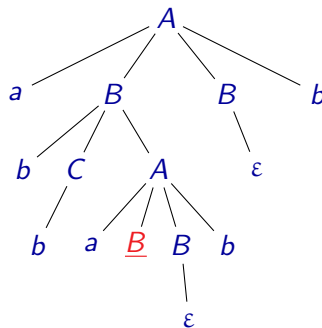
$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb \Rightarrow abbaBbb$

Derivation Tree

$A \rightarrow aBBb \mid AaA$

$\underline{B} \rightarrow \varepsilon \mid bCA$

$C \rightarrow AB \mid a \mid b$



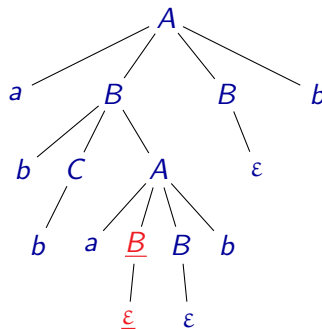
$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb \Rightarrow abba\underline{B}bb$

Derivation Tree

$A \rightarrow aBBb \mid AaA$

$\underline{B} \rightarrow \underline{\epsilon} \mid bCA$

$C \rightarrow AB \mid a \mid b$



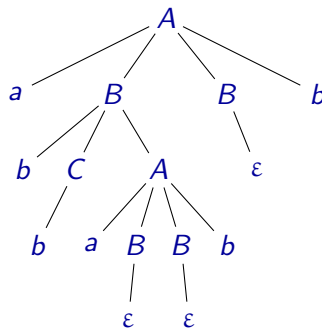
$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb \Rightarrow$
 $abba\underline{B}bb \Rightarrow abbabb$

Derivation Tree

$A \rightarrow aBBb \mid AaA$

$B \rightarrow \varepsilon \mid bCA$

$C \rightarrow AB \mid a \mid b$



$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb \Rightarrow$
 $abbaBbb \Rightarrow abbabb$

For each derivation there is some **derivation tree**:

- Nodes of the tree are labelled with terminals and nonterminals.
- The root of the tree is labelled with the initial nonterminal.
- The leafs of the tree are labelled with terminals or with symbols ϵ .
- The remaining nodes of the tree are labelled with nonterminals.
- If a node is labelled with some nonterminal A then its children are labelled with the symbols from the right-hand side of some rewriting rule $A \rightarrow \alpha$.

Left and Right Derivation

$$E \rightarrow E + E \mid E * E \mid (E) \mid a$$

A **left derivation** is a derivation where in every step we always replace the leftmost nonterminal.

$$\underline{E} \Rightarrow \underline{E} + E \Rightarrow \underline{E} * E + E \Rightarrow a * \underline{E} + E \Rightarrow a * a + \underline{E} \Rightarrow a * a + a$$

A **right derivation** is a derivation where in every step we always replace the rightmost nonterminal.

$$\underline{E} \Rightarrow E + \underline{E} \Rightarrow \underline{E} + a \Rightarrow E * \underline{E} + a \Rightarrow \underline{E} * a + a \Rightarrow a * a + a$$

A derivation need not be left or right:

$$\underline{E} \Rightarrow \underline{E} + E \Rightarrow E * \underline{E} + E \Rightarrow E * a + \underline{E} \Rightarrow \underline{E} * a + a \Rightarrow a * a + a$$

- There can be several different derivations corresponding to one derivation tree.
- For every derivation tree, there is exactly one left and exactly one right derivation corresponding to the tree.

Equivalence of Grammars

Grammars G_1 and G_2 are **equivalent** if they generate the same language, i.e., if $L(G_1) = L(G_2)$.

Remark: The problem of equivalence of context-free grammars is algorithmically undecidable. It can be shown that it is not possible to construct an algorithm that would decide for any pair of context-free grammars if they are equivalent or not.

Even the problem to decide if a grammar generates the language Σ^* is algorithmically undecidable.

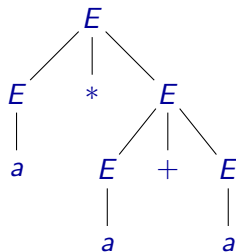
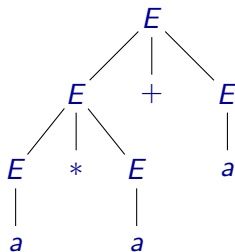
Ambiguous Grammars

A grammar G is **ambiguous** if there is a word $w \in L(G)$ that has two different derivation trees, resp. two different left or two different right derivations.

Example:

$E \Rightarrow E + E \Rightarrow E * E + E \Rightarrow a * E + E \Rightarrow a * a + E \Rightarrow a * a + a$

$E \Rightarrow E * E \Rightarrow E * E + E \Rightarrow a * E + E \Rightarrow a * a + E \Rightarrow a * a + a$



Ambiguous Grammars

Sometimes it is possible to replace an ambiguous grammar with a grammar generating the same language but which is not ambiguous.

Example: A grammar

$$E \rightarrow E + E \mid E * E \mid (E) \mid a$$

can be replaced with the equivalent grammar

$$E \rightarrow T \mid T + E$$

$$T \rightarrow F \mid F * T$$

$$F \rightarrow a \mid (E)$$

Remark: If there is no unambiguous grammar equivalent to a given ambiguous grammar, we say it is **inherently ambiguous**.

Definition

A language L is **context-free** if there exists some context-free grammar G such that $L = L(G)$.

The class of context-free languages is closed with respect to:

- concatenation
- union
- iteration

The class of context-free languages is not closed with respect to:

- complement
- intersection

Context-Free Languages

We have two grammars $G_1 = (\Pi_1, \Sigma, S_1, P_1)$ and $G_2 = (\Pi_2, \Sigma, S_2, P_2)$, and can assume that $\Pi_1 \cap \Pi_2 = \emptyset$ and $S \notin \Pi_1 \cup \Pi_2$.

- Grammar G such that $L(G) = L(G_1)L(G_2)$:

$$G = (\Pi_1 \cup \Pi_2 \cup \{S\}, \Sigma, S, P_1 \cup P_2 \cup \{S \rightarrow S_1 S_2\})$$

- Grammar G such that $L(G) = L(G_1) \cup L(G_2)$:

$$G = (\Pi_1 \cup \Pi_2 \cup \{S\}, \Sigma, S, P_1 \cup P_2 \cup \{S \rightarrow S_1, S \rightarrow S_2\})$$

- Grammar G such that $L(G) = L(G_1)^*$:

$$G = (\Pi_1 \cup \{S\}, \Sigma, S, P_1 \cup \{S \rightarrow \varepsilon, S \rightarrow S_1 S\})$$

Lexical and Syntactic Analysis — an example

Example: We would like to recognize a language of arithmetic expressions containing expressions such as:

34 $x+1$ $-x * 2 + 128 * (y - z / 3)$

- The expressions can contain number constants — sequences of digits 0, 1, ..., 9.
- The expressions can contain names of variables — sequences consisting of letters, digits, and symbol “_”, which do not start with a digit.
- The expressions can contain basic arithmetic operations — “+”, “-”, “*”, “/”, and unary “-”.
- It is possible to use parentheses — “(” and “)”, and to use a standard priority of arithmetic operations.

Lexical and Syntactic Analysis — an example

- **Input:** a sequence of characters (e.g., a string, a text file, etc.)
- **Output:** an abstract syntax tree representing the structure of a given expression, or an information about a syntax error in the expression

Lexical and Syntactic Analysis — an example

Construction of an **abstract syntax tree**:

- An enumerated type representing binary arithmetic operations:

enum *Bin_op* { **Add**, **Sub**, **Mul**, **Div** }

- An enumerated type representing unary arithmetic operations:

enum *Un_op* { **Un_minus** }

- Functions for creation of different kinds of nodes of an abstract syntax tree:

- **MK-VAR**(*ident*) — creates a leaf representing a variable
- **MK-NUM**(*num*) — creates a leaf representing a number constant
- **MK-UNARY**(*op*, *e*) — creates a node with one child *e*, on which a unary operation *op* (of type *Un_op*) is applied
- **MK-BINARY**(*op*, *e1*, *e2*) — creates a node with two children *e1* and *e2*, on which a binary operation *op* (of type *Bin_op*) is applied

Enumerated type *Token_kind* representing different kinds of **tokens**:

T_EOF	— the end of input
T_Ident	— identifier
T_Number	— number constant
T_LParen	— “(”
T_RParen	— “)”
T_Plus	— “+”
T_Minus	— “-”
T_Star	— “*”
T_Slash	— “/”

Variable `c` : a currently processed character (resp. a special value `<eof>` representing the end of input):

- at the beginning, the first character in the input is read to variable `c`
- function `NEXT-CHAR()` returns a next character from the input

Some helper functions:

- `ERROR()` — outputs an information about a syntax error and aborts the processing of the expression
- `is-ident-start-char(c)` — tests whether `c` is a character that can occur at the beginning of an identifier
- `is-ident-normal-char(c)` — tests whether `c` is a character that can occur in an identifier (on other positions except beginning)
- `is-digit(c)` — tests whether `c` is a digit

Some other helper functions:

- `CREATE-IDENT(s)` — creates an identifier from a given string *s*
- `CREATE-NUMBER(s)` — creates a number from a given string *s*

Auxiliary variables:

- *last-ident* — the last processed identifier
- *last-num* — the last processed number constant

Function `NEXT-TOKEN()` — the main part of the lexical analyser, it returns the following token from the input

```

1 NEXT-TOKEN():
2 begin
3   while  $c \in \{ " ", "\t" \}$  do
4      $c := \text{NEXT-CHAR}()$ ;
5   end
6   if  $c == \langle \text{eof} \rangle$  then
7     return T_EOF
8   else
9     switch  $c$  do
10      case "(": do  $c := \text{NEXT-CHAR}()$ ; return T_LParen
11      case ")": do  $c := \text{NEXT-CHAR}()$ ; return T_RParen
12      case "+": do  $c := \text{NEXT-CHAR}()$ ; return T_Plus
13      case "-": do  $c := \text{NEXT-CHAR}()$ ; return T_Minus
14      case "*": do  $c := \text{NEXT-CHAR}()$ ; return T_Star
15      case "/": do  $c := \text{NEXT-CHAR}()$ ; return T_Slash
16      otherwise do
17        if is-ident-start-char( $c$ ) then
18          return SCAN-IDENT()
19        else if is-digit( $c$ ) then
20          return SCAN-NUMBER()
21        else ERROR()
22      end
23    end
24  end
25 end
```

Lexical Analysis

```
1 SCAN-IDENT():  
2 begin  
3    $s := c$   
4    $c := \text{NEXT-CHAR}()$   
5   while is-ident-normal-char( $c$ ) do  
6      $s := s \cdot c$   
7      $c := \text{NEXT-CHAR}()$   
8   end  
9    $\text{last-ident} := \text{CREATE-IDENT}(s)$   
10  return T_Ident  
11 end
```

Lexical Analysis

```
1 SCAN-NUMBER():  
2 begin  
3    $s := c$   
4    $c := \text{NEXT-CHAR}()$   
5   while is-digit( $c$ ) do  
6      $s := s \cdot c$   
7      $c := \text{NEXT-CHAR}()$   
8   end  
9    $\text{last-num} := \text{CREATE-NUMBER}(s)$   
10  return T_Number  
11 end
```

Variable t :

- the last processed token

A helper function:

- `INIT-SCANNER()`:
 - initializes the lexical analyser
 - reads the first character from the input into variable c

The context-free grammar for the given language:

$$S \rightarrow E \langle eof \rangle$$

$$E \rightarrow T G$$

$$G \rightarrow \varepsilon \mid A T G$$

$$A \rightarrow + \mid -$$

$$T \rightarrow F U$$

$$U \rightarrow \varepsilon \mid M F U$$

$$M \rightarrow * \mid /$$

$$F \rightarrow - F \mid (E) \mid \langle ident \rangle \mid \langle num \rangle$$

One of the often used methods of syntactic analysis is **recursive descent**:

- For each nonterminal there is a corresponding function — the function corresponding to nonterminal A implements all rules with nonterminal A on the left-hand side.
- In a given function, the next token is used to select between corresponding rules.
- Instructions in the body of a function correspond to processing of right-hand sides of the rules:
 - an occurrence of nonterminal B — the function corresponding to nonterminal B is called
 - an occurrence of terminal a — it is checked that the following token corresponds to terminal a , when it does, the next token is read, otherwise an error is reported

$$S \rightarrow E \langle \text{eof} \rangle$$

```
1 PARSE():  
2 begin  
3   INIT-SCANNER()  
4    $t := \text{NEXT-TOKEN}()$   
5    $e := \text{PARSE-E}()$   
6   if  $t == \text{T\_EOF}$  then return  $e$   
7   else ERROR()  
8 end
```

$$E \rightarrow T G$$

```
1 PARSE-E ():  
2 begin  
3   e1 := PARSE-T()  
4   return PARSE-G(e1)  
5 end
```

$$G \rightarrow \varepsilon \mid A T G$$

```
1 PARSE-G (e1):  
2 begin  
3   if t ∈ {T_Plus, T_Minus} then  
4     op := PARSE-A()  
5     e2 := PARSE-T()  
6     return PARSE-G(MK-BINARY(op, e1, e2))  
7   else return e1  
8 end
```

$$T \rightarrow F U$$

```
1 PARSE-T ():  
2 begin  
3    $e1 := \text{PARSE-F}()$   
4   return PARSE-U( $e1$ )  
5 end
```

$$U \rightarrow \varepsilon \mid M F U$$

```
1 PARSE-U ( $e1$ ):  
2 begin  
3   if  $t \in \{\mathbf{T\_Star}, \mathbf{T\_Slash}\}$  then  
4      $op := \text{PARSE-M}()$   
5      $e2 := \text{PARSE-F}()$   
6     return PARSE-U(MK-BINARY( $op, e1, e2$ ))  
7   else return  $e1$   
8 end
```

$$A \rightarrow + \mid -$$

```
1  PARSE-A ():  
2  begin  
3      if  $t == \mathbf{T\_Plus}$  then  
4           $t := \text{NEXT-TOKEN}()$   
5          return Add  
6      else if  $t == \mathbf{T\_Minus}$  then  
7           $t := \text{NEXT-TOKEN}()$   
8          return Sub  
9      else ERROR()  
10 end
```

$$M \rightarrow * \mid /$$

```
1 PARSE-M():  
2 begin  
3   if  $t == \mathbf{T\_Star}$  then  
4      $t := \text{NEXT-TOKEN}()$   
5     return Mul  
6   else if  $t == \mathbf{T\_Slash}$  then  
7      $t := \text{NEXT-TOKEN}()$   
8     return Div  
9   else ERROR()  
10 end
```

$$\begin{array}{l}
 F \rightarrow - F \\
 \quad | (E) \\
 \quad | \langle ident \rangle \\
 \quad | \langle num \rangle
 \end{array}$$

```

1  PARSE-F ():
2  begin
3      switch t do
4          case T_Minus: do
5              t := NEXT-TOKEN()
6              e := PARSE-F()
7              return MK-UNARY(Un_minus, e)
8          case T_LParen: do
9              t := NEXT-TOKEN()
10             e := PARSE-E()
11             if t == T_RParen then
12                 t := NEXT-TOKEN()
13                 return e
14             else ERROR()
15         case T_Ident: do
16             e := MK-VAR(last-ident)
17             t := NEXT-TOKEN()
18             return e
19         case T_Number: do
20             e := MK-NUM(last-num)
21             t := NEXT-TOKEN()
22             return e
23         otherwise do ERROR()
24     end
25 end

```

- If a function ends with a recursive call of itself, as for example function `PARSE-G()`, it is possible to replace this recursion with an iteration.
- Functions `PARSE-E()` and `PARSE-G()` can be merged into one function.
- Similarly, it is possible to replace a recursion with an iteration in function `PARSE-U()`, and functions `PARSE-T()` and `PARSE-U()` can be merged into one function.

```
1  PARSE-E ():
2  begin
3      e1 := PARSE-T()
4      while t ∈ {T_Plus, T_Minus} do
5          op := PARSE-A()
6          e2 := PARSE-T()
7          e1 := MK-BINARY(op, e1, e2)
8      end
9      return e1
10 end
```

```
1  PARSE-T ():
2  begin
3      e1 := PARSE-F()
4      while t ∈ {T_Star, T_Slash} do
5          op := PARSE-M()
6          e2 := PARSE-F()
7          e1 := MK-BINARY(op, e1, e2)
8      end
9      return e1
10 end
```

Algorithms

Example: An algorithm described by **pseudocode**:

Algorithm 1: An algorithm for finding the maximal element in an array

```
1 FIND-MAX ( $A, n$ ):  
2 begin  
3    $k := 0$   
4   for  $i := 1$  to  $n - 1$  do  
5     if  $A[i] > A[k]$  then  
6        $k := i$   
7     end  
8   end  
9   return  $A[k]$   
10 end
```

Algorithm

- processes an **input**
- generates an **output**

From the point of view of an analysis how a given algorithm works, it usually makes only a little difference if the algorithm:

- reads input data from some input device (e.g., from a file, from a keyboard, etc.)
- writes data to some output device (e.g., to a file, on a screen, etc.)

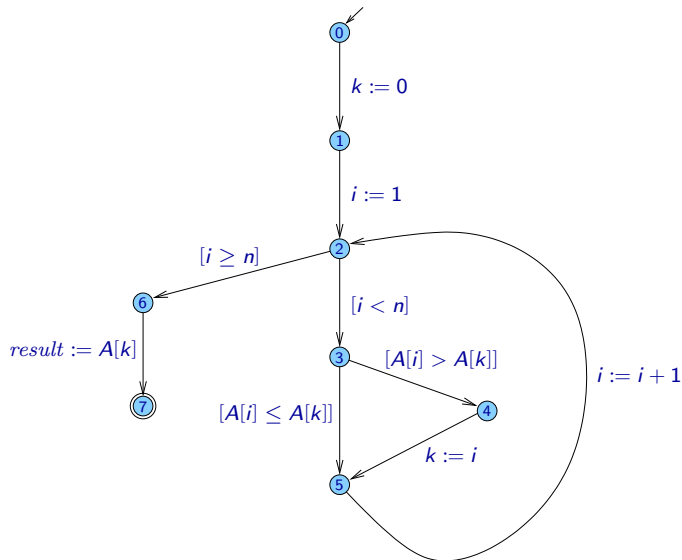
or

- reads input data from a memory (e.g., they are given to it as parameters)
- writes data somewhere to memory (e.g., it returns them as a return value)

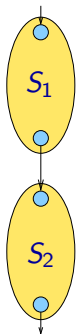
Instructions can be roughly divided into two groups:

- instructions working directly with data:
 - assignment
 - evaluation of values of expressions in conditions
 - reading input, writing output
 - ...
- instruction affecting the **control flow** — they determine, which instructions will be executed, in what order, etc.:
 - branching (if, switch, ...)
 - cycles (while, do .. while, for, ...)
 - organisation of instructions into blocks
 - returns from subprograms (return, ...)
 - ...

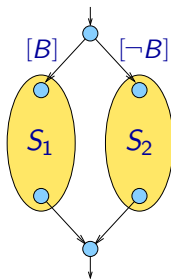
Control Flow Graph



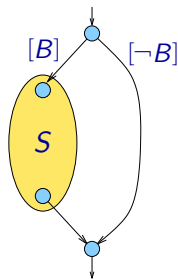
Some Basic Constructions of Structured Programming



$S_1; S_2$

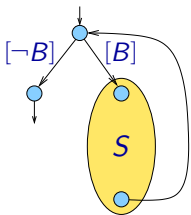


if B then S_1 else S_2

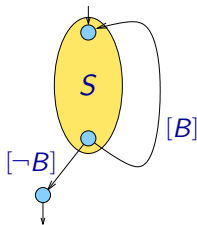


if B then S

Some Basic Constructions of Structured Programming

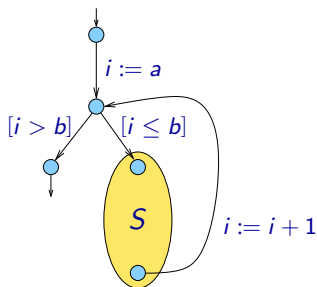


while B **do** S



do S **while** B

Some Basic Constructions of Structured Programming



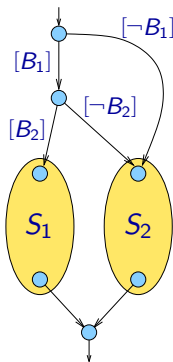
```
 $i := a$   
while  $i \leq b$  do  
     $S$   
     $i := i + 1$   
end
```

```
for  $i := a$  to  $b$  do  $S$ 
```

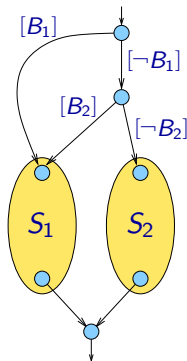
Some Basic Constructions of Structured Programming

Short-circuit evaluation of compound conditions, e.g.:

while $i < n$ **and** $A[i] > x$ **do** ...



if B_1 **and** B_2 **then** S_1 **else** S_2



if B_1 **or** B_2 **then** S_1 **else** S_2

Control-flow Realized by GOTO

- **goto** ℓ — **unconditional jump**
- **if** B **then goto** ℓ — **conditional jump**

Example:

```
0:  $k := 0$   
1:  $i := 1$   
2: goto 6  
3: if  $A[i] \leq A[k]$  then goto 5  
4:  $k := i$   
5:  $i := i + 1$   
6: if  $i < n$  then goto 3  
7: return  $A[k]$ 
```

Control-flow Realized by GOTO

- **goto** ℓ — **unconditional jump**
- **if** B **then goto** ℓ — **conditional jump**

Example:

```
start:  $k := 0$   
       $i := 1$   
      goto  $L3$   
 $L1$ : if  $A[i] \leq A[k]$  then goto  $L2$   
       $k := i$   
 $L2$ :  $i := i + 1$   
 $L3$ : if  $i < n$  then goto  $L1$   
      return  $A[k]$ 
```

Evaluation of Complicated Expressions

Evaluation of a complicated expression such as

$$A[i + s] := (B[3 * j + 1] + x) * y + 8$$

can be replaced by a sequence of simpler instructions on the lower level, such as

$$\begin{aligned}t_1 &:= i + s \\t_2 &:= 3 * j \\t_2 &:= t_2 + 1 \\t_3 &:= B[t_2] \\t_3 &:= t_3 + x \\t_3 &:= t_3 * y \\t_3 &:= t_3 + 8 \\A[t_1] &:= t_3\end{aligned}$$

Computation of an Algorithm

An algorithm is executed by a machine — it can be for example:

- real computer — executes instructions of a machine code
- virtual machine — executes instructions of a bytecode
- some idealized mathematical model of a computer
- ...

The machine can be:

- specialized — executes only one algorithm
- universal — can execute arbitrary algorithm, given in a form of **program**

The machine performs **steps**.

The algorithm processes a particular **input** during its computation.

Computation of an Algorithm

During a computation, the machine must remember:

- the current instruction
- the content of its working memory

It depends on the type of the machine:

- what is the type of data, with which the machine works
- how this data are organized in its memory

Depending on the type of the algorithm and the type of analysis, which we want to do, we can decide if it makes sense to include in memory also the places

- from which the input data are read
- where the output data are written

Computation of an Algorithm

Configuration — the description of the global state of the machine in some particular step during a computation

Example: A configuration of the form

$$(q, mem)$$

where

- q — the current control state
- mem — the current content of memory of the machine — the values assigned currently to variables.

An example of a content of memory mem :

$$\langle A: [3, 8, 1, 3, 6], \quad n: 5, \quad i: 1, \quad k: 0, \quad result: ? \rangle$$

Computation of an Algorithm

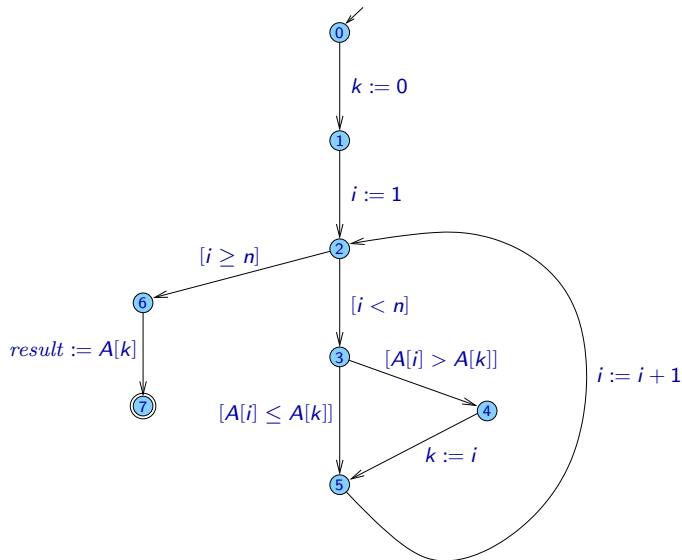
An example of a configuration:

$(2, \langle A: [3, 8, 1, 3, 6], \ n: 5, \ i: 1, \ k: 0, \ result: ? \rangle)$

A **computation** of a machine $\mathcal{M}$ executing an algorithm Alg , where it processes an input w , in a sequence of configurations.

- It starts in an **initial configuration**.
- In every step, it goes from one configuration to another.
- The computation ends in a **final configuration**.

Computation of an Algorithm



Computation of an Algorithm

Example: A computation, where algorithm `FIND-MAX` processes an input where $A = [3, 8, 1, 3, 6]$ and $n = 5$.

α_0 : (0, $\langle A: [3, 8, 1, 3, 6], n: 5, i: ?, k: ?, result: ? \rangle$)

Computation of an Algorithm

Example: A computation, where algorithm `FIND-MAX` processes an input where $A = [3, 8, 1, 3, 6]$ and $n = 5$.

α_0 : (0, $\langle A: [3, 8, 1, 3, 6], n: 5, i: ?, k: ?, result: ? \rangle$)

α_1 : (1, $\langle A: [3, 8, 1, 3, 6], n: 5, i: ?, k: 0, result: ? \rangle$)

Computation of an Algorithm

Example: A computation, where algorithm **FIND-MAX** processes an input where $A = [3, 8, 1, 3, 6]$ and $n = 5$.

α_0 : (0, $\langle A: [3, 8, 1, 3, 6], n: 5, i: ?, k: ?, result: ? \rangle$)

α_1 : (1, $\langle A: [3, 8, 1, 3, 6], n: 5, i: ?, k: 0, result: ? \rangle$)

α_2 : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)

Computation of an Algorithm

Example: A computation, where algorithm **FIND-MAX** processes an input where $A = [3, 8, 1, 3, 6]$ and $n = 5$.

α_0 : (0, $\langle A: [3, 8, 1, 3, 6], n: 5, i: ?, k: ?, result: ? \rangle$)

α_1 : (1, $\langle A: [3, 8, 1, 3, 6], n: 5, i: ?, k: 0, result: ? \rangle$)

α_2 : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)

α_3 : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)

Computation of an Algorithm

Example: A computation, where algorithm **FIND-MAX** processes an input where $A = [3, 8, 1, 3, 6]$ and $n = 5$.

α_0 : (0, $\langle A: [3, 8, 1, 3, 6], n: 5, i: ?, k: ?, result: ? \rangle$)
 α_1 : (1, $\langle A: [3, 8, 1, 3, 6], n: 5, i: ?, k: 0, result: ? \rangle$)
 α_2 : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_3 : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_4 : (4, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)

Computation of an Algorithm

Example: A computation, where algorithm **FIND-MAX** processes an input where $A = [3, 8, 1, 3, 6]$ and $n = 5$.

α_0 : (0, $\langle A: [3, 8, 1, 3, 6], n: 5, i: ?, k: ?, result: ? \rangle$)
 α_1 : (1, $\langle A: [3, 8, 1, 3, 6], n: 5, i: ?, k: 0, result: ? \rangle$)
 α_2 : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_3 : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_4 : (4, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_5 : (5, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 1, result: ? \rangle$)

Computation of an Algorithm

Example: A computation, where algorithm **FIND-MAX** processes an input where $A = [3, 8, 1, 3, 6]$ and $n = 5$.

α_0 : (0, $\langle A: [3, 8, 1, 3, 6], n: 5, i: ?, k: ?, result: ? \rangle$)
 α_1 : (1, $\langle A: [3, 8, 1, 3, 6], n: 5, i: ?, k: 0, result: ? \rangle$)
 α_2 : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_3 : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_4 : (4, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_5 : (5, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 1, result: ? \rangle$)
 α_6 : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 2, k: 1, result: ? \rangle$)

Computation of an Algorithm

Example: A computation, where algorithm **FIND-MAX** processes an input where $A = [3, 8, 1, 3, 6]$ and $n = 5$.

α_0 : (0, $\langle A: [3, 8, 1, 3, 6], n: 5, i: ?, k: ?, result: ? \rangle$)
 α_1 : (1, $\langle A: [3, 8, 1, 3, 6], n: 5, i: ?, k: 0, result: ? \rangle$)
 α_2 : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_3 : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_4 : (4, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_5 : (5, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 1, result: ? \rangle$)
 α_6 : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 2, k: 1, result: ? \rangle$)
 α_7 : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 2, k: 1, result: ? \rangle$)

Computation of an Algorithm

Example: A computation, where algorithm **FIND-MAX** processes an input where $A = [3, 8, 1, 3, 6]$ and $n = 5$.

α_0 : (0, $\langle A: [3, 8, 1, 3, 6], n: 5, i: ?, k: ?, result: ? \rangle$)
 α_1 : (1, $\langle A: [3, 8, 1, 3, 6], n: 5, i: ?, k: 0, result: ? \rangle$)
 α_2 : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_3 : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_4 : (4, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_5 : (5, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 1, result: ? \rangle$)
 α_6 : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 2, k: 1, result: ? \rangle$)
 α_7 : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 2, k: 1, result: ? \rangle$)
 α_8 : (5, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 2, k: 1, result: ? \rangle$)

Computation of an Algorithm

Example: A computation, where algorithm **FIND-MAX** processes an input where $A = [3, 8, 1, 3, 6]$ and $n = 5$.

α_0 : (0, $\langle A: [3, 8, 1, 3, 6], n: 5, i: ?, k: ?, result: ? \rangle$)
 α_1 : (1, $\langle A: [3, 8, 1, 3, 6], n: 5, i: ?, k: 0, result: ? \rangle$)
 α_2 : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_3 : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_4 : (4, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_5 : (5, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 1, result: ? \rangle$)
 α_6 : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 2, k: 1, result: ? \rangle$)
 α_7 : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 2, k: 1, result: ? \rangle$)
 α_8 : (5, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 2, k: 1, result: ? \rangle$)
 α_9 : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 3, k: 1, result: ? \rangle$)

Computation of an Algorithm

Example: A computation, where algorithm **FIND-MAX** processes an input where $A = [3, 8, 1, 3, 6]$ and $n = 5$.

α_0 : (0, $\langle A: [3, 8, 1, 3, 6], n: 5, i: ?, k: ?, result: ? \rangle$)
 α_1 : (1, $\langle A: [3, 8, 1, 3, 6], n: 5, i: ?, k: 0, result: ? \rangle$)
 α_2 : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_3 : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_4 : (4, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_5 : (5, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 1, result: ? \rangle$)
 α_6 : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 2, k: 1, result: ? \rangle$)
 α_7 : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 2, k: 1, result: ? \rangle$)
 α_8 : (5, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 2, k: 1, result: ? \rangle$)
 α_9 : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 3, k: 1, result: ? \rangle$)
 α_{10} : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 3, k: 1, result: ? \rangle$)

Computation of an Algorithm

Example: A computation, where algorithm **FIND-MAX** processes an input where $A = [3, 8, 1, 3, 6]$ and $n = 5$.

α_0 : (0, $\langle A: [3, 8, 1, 3, 6], n: 5, i: ?, k: ?, result: ? \rangle$)
 α_1 : (1, $\langle A: [3, 8, 1, 3, 6], n: 5, i: ?, k: 0, result: ? \rangle$)
 α_2 : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_3 : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_4 : (4, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_5 : (5, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 1, result: ? \rangle$)
 α_6 : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 2, k: 1, result: ? \rangle$)
 α_7 : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 2, k: 1, result: ? \rangle$)
 α_8 : (5, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 2, k: 1, result: ? \rangle$)
 α_9 : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 3, k: 1, result: ? \rangle$)
 α_{10} : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 3, k: 1, result: ? \rangle$)
 α_{11} : (5, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 3, k: 1, result: ? \rangle$)

Computation of an Algorithm

Example: A computation, where algorithm **FIND-MAX** processes an input where $A = [3, 8, 1, 3, 6]$ and $n = 5$.

α_0 : (0, $\langle A: [3, 8, 1, 3, 6], n: 5, i: ?, k: ?, result: ? \rangle$)
 α_1 : (1, $\langle A: [3, 8, 1, 3, 6], n: 5, i: ?, k: 0, result: ? \rangle$)
 α_2 : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_3 : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_4 : (4, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_5 : (5, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 1, result: ? \rangle$)
 α_6 : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 2, k: 1, result: ? \rangle$)
 α_7 : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 2, k: 1, result: ? \rangle$)
 α_8 : (5, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 2, k: 1, result: ? \rangle$)
 α_9 : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 3, k: 1, result: ? \rangle$)
 α_{10} : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 3, k: 1, result: ? \rangle$)
 α_{11} : (5, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 3, k: 1, result: ? \rangle$)
 α_{12} : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 4, k: 1, result: ? \rangle$)

Computation of an Algorithm

Example: A computation, where algorithm **FIND-MAX** processes an input where $A = [3, 8, 1, 3, 6]$ and $n = 5$.

α_0 : (0, $\langle A: [3, 8, 1, 3, 6], n: 5, i: ?, k: ?, result: ? \rangle$)
 α_1 : (1, $\langle A: [3, 8, 1, 3, 6], n: 5, i: ?, k: 0, result: ? \rangle$)
 α_2 : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_3 : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_4 : (4, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_5 : (5, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 1, result: ? \rangle$)
 α_6 : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 2, k: 1, result: ? \rangle$)
 α_7 : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 2, k: 1, result: ? \rangle$)
 α_8 : (5, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 2, k: 1, result: ? \rangle$)
 α_9 : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 3, k: 1, result: ? \rangle$)
 α_{10} : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 3, k: 1, result: ? \rangle$)
 α_{11} : (5, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 3, k: 1, result: ? \rangle$)
 α_{12} : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 4, k: 1, result: ? \rangle$)
 α_{13} : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 4, k: 1, result: ? \rangle$)

Computation of an Algorithm

Example: A computation, where algorithm **FIND-MAX** processes an input where $A = [3, 8, 1, 3, 6]$ and $n = 5$.

α_0 : (0, $\langle A: [3, 8, 1, 3, 6], n: 5, i: ?, k: ?, result: ? \rangle$)
 α_1 : (1, $\langle A: [3, 8, 1, 3, 6], n: 5, i: ?, k: 0, result: ? \rangle$)
 α_2 : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_3 : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_4 : (4, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_5 : (5, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 1, result: ? \rangle$)
 α_6 : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 2, k: 1, result: ? \rangle$)
 α_7 : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 2, k: 1, result: ? \rangle$)
 α_8 : (5, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 2, k: 1, result: ? \rangle$)
 α_9 : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 3, k: 1, result: ? \rangle$)
 α_{10} : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 3, k: 1, result: ? \rangle$)
 α_{11} : (5, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 3, k: 1, result: ? \rangle$)
 α_{12} : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 4, k: 1, result: ? \rangle$)
 α_{13} : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 4, k: 1, result: ? \rangle$)
 α_{14} : (5, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 4, k: 1, result: ? \rangle$)

Computation of an Algorithm

Example: A computation, where algorithm **FIND-MAX** processes an input where $A = [3, 8, 1, 3, 6]$ and $n = 5$.

α_0 : (0, $\langle A: [3, 8, 1, 3, 6], n: 5, i: ?, k: ?, result: ? \rangle$)
 α_1 : (1, $\langle A: [3, 8, 1, 3, 6], n: 5, i: ?, k: 0, result: ? \rangle$)
 α_2 : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_3 : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_4 : (4, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_5 : (5, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 1, result: ? \rangle$)
 α_6 : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 2, k: 1, result: ? \rangle$)
 α_7 : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 2, k: 1, result: ? \rangle$)
 α_8 : (5, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 2, k: 1, result: ? \rangle$)
 α_9 : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 3, k: 1, result: ? \rangle$)
 α_{10} : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 3, k: 1, result: ? \rangle$)
 α_{11} : (5, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 3, k: 1, result: ? \rangle$)
 α_{12} : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 4, k: 1, result: ? \rangle$)
 α_{13} : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 4, k: 1, result: ? \rangle$)
 α_{14} : (5, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 4, k: 1, result: ? \rangle$)
 α_{15} : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 5, k: 1, result: ? \rangle$)

Computation of an Algorithm

Example: A computation, where algorithm **FIND-MAX** processes an input where $A = [3, 8, 1, 3, 6]$ and $n = 5$.

α_0 : (0, $\langle A: [3, 8, 1, 3, 6], n: 5, i: ?, k: ?, result: ? \rangle$)
 α_1 : (1, $\langle A: [3, 8, 1, 3, 6], n: 5, i: ?, k: 0, result: ? \rangle$)
 α_2 : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_3 : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_4 : (4, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_5 : (5, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 1, result: ? \rangle$)
 α_6 : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 2, k: 1, result: ? \rangle$)
 α_7 : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 2, k: 1, result: ? \rangle$)
 α_8 : (5, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 2, k: 1, result: ? \rangle$)
 α_9 : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 3, k: 1, result: ? \rangle$)
 α_{10} : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 3, k: 1, result: ? \rangle$)
 α_{11} : (5, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 3, k: 1, result: ? \rangle$)
 α_{12} : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 4, k: 1, result: ? \rangle$)
 α_{13} : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 4, k: 1, result: ? \rangle$)
 α_{14} : (5, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 4, k: 1, result: ? \rangle$)
 α_{15} : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 5, k: 1, result: ? \rangle$)
 α_{16} : (6, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 5, k: 1, result: ? \rangle$)

Computation of an Algorithm

Example: A computation, where algorithm **FIND-MAX** processes an input where $A = [3, 8, 1, 3, 6]$ and $n = 5$.

α_0 : (0, $\langle A: [3, 8, 1, 3, 6], n: 5, i: ?, k: ?, result: ? \rangle$)
 α_1 : (1, $\langle A: [3, 8, 1, 3, 6], n: 5, i: ?, k: 0, result: ? \rangle$)
 α_2 : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_3 : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_4 : (4, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 0, result: ? \rangle$)
 α_5 : (5, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 1, k: 1, result: ? \rangle$)
 α_6 : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 2, k: 1, result: ? \rangle$)
 α_7 : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 2, k: 1, result: ? \rangle$)
 α_8 : (5, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 2, k: 1, result: ? \rangle$)
 α_9 : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 3, k: 1, result: ? \rangle$)
 α_{10} : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 3, k: 1, result: ? \rangle$)
 α_{11} : (5, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 3, k: 1, result: ? \rangle$)
 α_{12} : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 4, k: 1, result: ? \rangle$)
 α_{13} : (3, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 4, k: 1, result: ? \rangle$)
 α_{14} : (5, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 4, k: 1, result: ? \rangle$)
 α_{15} : (2, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 5, k: 1, result: ? \rangle$)
 α_{16} : (6, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 5, k: 1, result: ? \rangle$)
 α_{17} : (7, $\langle A: [3, 8, 1, 3, 6], n: 5, i: 5, k: 1, result: 8 \rangle$)

Computation of an Algorithm

By executing an instruction I , the machine goes from configuration α to configuration α' :

$$\alpha \xrightarrow{I} \alpha'$$

A computation can be:

- **Finite:**

$$\alpha_0 \xrightarrow{I_0} \alpha_1 \xrightarrow{I_1} \alpha_2 \xrightarrow{I_2} \alpha_3 \xrightarrow{I_3} \alpha_4 \xrightarrow{I_4} \dots \xrightarrow{I_{t-2}} \alpha_{t-1} \xrightarrow{I_{t-1}} \alpha_t$$

where α_t is a final configuration

- **Infinite:**

$$\alpha_0 \xrightarrow{I_0} \alpha_1 \xrightarrow{I_1} \alpha_2 \xrightarrow{I_2} \alpha_3 \xrightarrow{I_3} \alpha_4 \xrightarrow{I_4} \dots$$

A computation can be described in two different ways:

- as a sequence of configurations $\alpha_0, \alpha_1, \alpha_2, \dots$
- as a sequence of executed instructions $l_0, l_1, l_2, \dots$

Algorithms are used for solving **problems**.

- **Problem** — a specification **what** should be computed by an algorithm:
 - Description of inputs
 - Description of outputs
 - How outputs are related to inputs
- **Algorithm** — a particular procedure that describes **how** to compute an output for each possible input

Correctness of Algorithms

Example: The problem of finding a maximal element in an array:

Input: An array A indexed from zero and a number n representing the number of elements in array A . It is assumed that $n \geq 1$.

Output: A value $result$ of a maximal element in the array A , i.e., the value $result$ such that:

- $A[j] \leq result$ for all $j \in \mathbb{N}$, where $0 \leq j < n$, and
- there exists $j \in \mathbb{N}$ such that $0 \leq j < n$ and $A[j] = result$.

An **instance** of a problem — concrete input data, e.g.,

$$A = [3, 8, 1, 3, 6], \quad n = 5.$$

The output for this instance is value 8.

Definition

An algorithm Alg **solves** a given problem P , if for **each** instance w of problem P , the following conditions are satisfied:

- (a) The computation of algorithm Alg on input w halts after finite number of steps.
- (b) Algorithm Alg generates a correct output for input w according to conditions in problem P .

An algorithm that solves problem P is a correct solution of this problem.

Correctness of Algorithms

Algorithm *Alg* is **not** a correct solution of problem *P* if there exists an input *w* such that in the computation on this input, one of the following incorrect behaviours occurs:

- some incorrect illegal operation is performed (an access to an element of an array with index out of bounds, division by zero, ...),
- the generated output does not satisfy the conditions specified in problem *P*,
- the computation never halts.

Testing — running the algorithm with different inputs and checking whether the algorithm behaves correctly on these inputs.

Testing can be used to show the presence of bugs but not to show that algorithm behaves correctly for **all** inputs.

Generally, it is reasonable to divide a proof of correctness of an algorithm into two parts:

- Showing that the algorithm never does anything “wrong” for any input:
 - no illegal operation is performed during a computation
 - if the program halts, the generated output will be “correct”
- Showing that for every input the algorithm halts after a finite number of steps.

Invariant — a condition that must be always satisfied in a given position in a code of the algorithm (i.e., in all possible computations for all allowed inputs) whenever the algorithm goes through this position.

We say that a configuration α is **reachable** if there exists an input w such that α is one of configurations through which the algorithm goes in the computation on input w .

If an algorithm is represented by a control-flow graph, for a given **control state** q (i.e., a node of the graph) we can specify invariants that hold in every reachable configuration with control state q .

Invariants can be written as formulas of predicate logic:

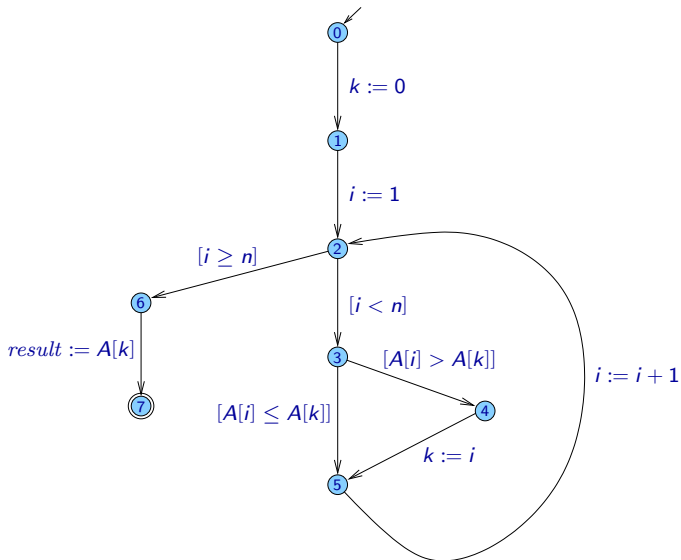
- **free** variables correspond to variables of the program
- a **valuation** is determined by values of program variables in a given configuration

Example: Formula

$$(1 \leq i) \wedge (i \leq n)$$

holds for example in a configuration where variable i has value 5 and variable n has value 14.

Invariants



Examples of invariants:

- an invariant in a control state q is represented by a formula φ_q

Invariants for individual control states (so far only hypotheses):

- $\varphi_0: (n \geq 1)$
- $\varphi_1: (n \geq 1) \wedge (k = 0)$
- $\varphi_2: (n \geq 1) \wedge (1 \leq i \leq n) \wedge (0 \leq k < i)$
- $\varphi_3: (n \geq 1) \wedge (1 \leq i < n) \wedge (0 \leq k < i)$
- $\varphi_4: (n \geq 1) \wedge (1 \leq i < n) \wedge (0 \leq k < i)$
- $\varphi_5: (n \geq 1) \wedge (1 \leq i < n) \wedge (0 \leq k \leq i)$
- $\varphi_6: (n \geq 1) \wedge (i = n) \wedge (0 \leq k < n)$
- $\varphi_7: (n \geq 1) \wedge (i = n) \wedge (0 \leq k < n)$

Examples of invariants:

- an invariant in a control state q is represented by a formula φ_q

Invariants for individual control states (so far only hypotheses):

- $\varphi_0: n \geq 1$
- $\varphi_1: n \geq 1, k = 0$
- $\varphi_2: n \geq 1, 1 \leq i \leq n, 0 \leq k < i$
- $\varphi_3: n \geq 1, 1 \leq i < n, 0 \leq k < i$
- $\varphi_4: n \geq 1, 1 \leq i < n, 0 \leq k < i$
- $\varphi_5: n \geq 1, 1 \leq i < n, 0 \leq k \leq i$
- $\varphi_6: n \geq 1, i = n, 0 \leq k < n$
- $\varphi_7: n \geq 1, i = n, 0 \leq k < n$

Invariants

Checking that the given invariants really hold:

- It is necessary to check for each instruction of the algorithm that under the assumption that a specified invariant holds before an execution of the instruction, the other specified invariant holds after the execution of the instruction.

Let us assume the algorithm is represented as a control-flow graph:

- edges correspond to instructions
- consider an edge from state q to state q' labelled with instruction I
- let us say that (so far non-verified) invariants for states q and q' are expressed by formulas φ and φ'
- for this edge we must check that for every configurations

$\alpha = (q, mem)$ and $\alpha' = (q', mem')$ such that $\alpha \xrightarrow{I} \alpha'$, it holds that if

- φ holds in configuration α ,
then
- φ' holds in configuration α'

Invariants

Checking instructions, which are conditional tests:

- an edge labelled with a conditional test $[B]$

A content of memory is not modified.

It is sufficient to check that the following implication holds

$$(\varphi \wedge B) \rightarrow \varphi'$$

Remark: The given implication must hold for all possible values of variables.

Example: Let us assume that formulas contain only variables n , i , k , and that values of these variables are integers:

$$(\forall n \in \mathbb{Z})(\forall i \in \mathbb{Z})(\forall k \in \mathbb{Z}) (\varphi \wedge B \rightarrow \varphi')$$

Checking those instructions that assign values to variables (they modify a content of memory):

- an edge labelled with assignment $x := E$

φ'' — a formula obtained from formula φ' by renaming of all free occurrences of variable x to x'

It is necessary to check the validity of implication

$$(\varphi \wedge (x' = E)) \rightarrow \varphi''$$

Example: Assignment $k := 3 * k + i + 1$:

$$(\forall n \in \mathbb{Z})(\forall i \in \mathbb{Z})(\forall k \in \mathbb{Z})(\forall k' \in \mathbb{Z}) (\varphi \wedge (k' = 3 * k + i + 1) \rightarrow \varphi'')$$

Finishing the checking that the algorithm for finding maximal element in an array returns a correct result (under assumption that it halts):

- $\psi_0: \varphi_0$
- $\psi_1: \varphi_1 \wedge (\forall j \in \mathbb{N})(0 \leq j < 1 \rightarrow A[j] \leq A[k])$
- $\psi_2: \varphi_2 \wedge (\forall j \in \mathbb{N})(0 \leq j < i \rightarrow A[j] \leq A[k])$
- $\psi_3: \varphi_3 \wedge (\forall j \in \mathbb{N})(0 \leq j < i \rightarrow A[j] \leq A[k])$
- $\psi_4: \varphi_4 \wedge (\forall j \in \mathbb{N})(0 \leq j < i \rightarrow A[j] \leq A[k]) \wedge (A[i] > A[k])$
- $\psi_5: \varphi_5 \wedge (\forall j \in \mathbb{N})(0 \leq j \leq i \rightarrow A[j] \leq A[k])$
- $\psi_6: \varphi_6 \wedge (\forall j \in \mathbb{N})(0 \leq j < n \rightarrow A[j] \leq A[k])$
- $\psi_7: \varphi_7 \wedge (result = A[k]) \wedge (\forall j \in \mathbb{N})(0 \leq j < n \rightarrow A[j] \leq result) \wedge (\exists j \in \mathbb{N})(0 \leq j < n \wedge A[j] = result)$

Usually it is not necessary to specify invariants in all control states but only in some “important” states — in particular, in states where the algorithm enters or leaves loops:

It is necessary to verify:

- That the invariant holds before entering the loop.
- That if the invariant holds before an iteration of the loop then it holds also after the iteration.
- That the invariant holds when the loop is left.

Example: In algorithm `FIND-MAX`, state 2 is such “important” state.

In state 2, the following holds:

- $n \geq 1$
- $1 \leq i \leq n$
- $0 \leq k < i$
- For each j such that $0 \leq j < i$ it holds that $A[j] \leq A[k]$.

Two possibilities how an infinite computation can look:

- some configuration is repeated — then all following configurations are also repeated
- all configurations in a computation are different but a final configuration is never reached

Finiteness of a Computation

One of standard ways of proving that an algorithm halts for every input after a finite number of steps:

- to assign a value from a set W to every (reachable) configuration
- to define an order $\leq$ on set W such that there are no infinite (strictly) decreasing sequences of elements of W
- to show that the values assigned to configuration decrease with every execution of each instruction, i.e., if $\alpha \xrightarrow{I} \alpha'$ then

$$f(\alpha) > f(\alpha')$$

$(f(\alpha), f(\alpha'))$ are values from set W assigned to configurations α and α')

Finiteness of a Computation

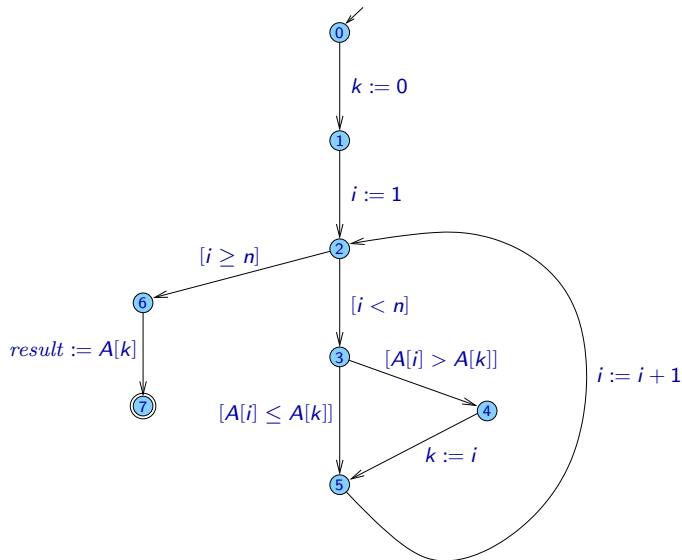
As a set W , we can use for example:

- The set of natural numbers $\mathbb{N} = \{0, 1, 2, 3, \dots\}$ with ordering $\leq$.
- The set of vectors of natural numbers with lexicographic ordering, i.e., the ordering where vector $(a_1, a_2, \dots, a_m)$ is smaller than $(b_1, b_2, \dots, b_n)$, if
 - there exists i such that $1 \leq i \leq m$ and $i \leq n$, where $a_i < b_i$ and for all j such that $1 \leq j < i$ it holds that $a_j = b_j$, or
 - $m < n$ and for all j such that $1 \leq j \leq m$ is $a_j = b_j$.

For example, $(5, 1, 3, 6, 4) < (5, 1, 4, 1)$ and $(4, 1, 1) < (4, 1, 1, 3)$.

Remark: The number of elements in vectors must be bounded by some constant.

Finiteness of a Computation



Example: Vectors assigned to individual configurations:

- State 0: $f(\alpha) = (4)$
- State 1: $f(\alpha) = (3)$
- State 2: $f(\alpha) = (2, n - i, 3)$
- State 3: $f(\alpha) = (2, n - i, 2)$
- State 4: $f(\alpha) = (2, n - i, 1)$
- State 5: $f(\alpha) = (2, n - i, 0)$
- State 6: $f(\alpha) = (1)$
- State 7: $f(\alpha) = (0)$

Computational Complexity of Algorithms

Complexity of an Algorithm

- Computers work fast but not infinitely fast. Execution of each instruction takes some (very short) time.
- The same problem can be solved by several different algorithms. The time of a computation (determined mostly by the number of executed instructions) can be different for different algorithms.
- We can implement the algorithms and then measure the time of their computation. By this we find out how long the computation takes on particular data on which we test the algorithm.
- We would like to have a more precise idea how long the computation takes on all possible input data.

Complexity of an Algorithm

- A running time is affected by many factors, e.g.:
 - the algorithm that is used
 - the amount of input data
 - used hardware (e.g., the frequency at which a CPU is running can be important)
 - the used programming language — its implementation (compiler/interpreter)
 - ...
- If we need to solve problem for “small” input data, the running time is usually negligible.
- With increasing amount of input data (the size of input), the running time can grow, sometimes significantly.

Complexity of an Algorithm

- **Time complexity of an algorithm** — how the running time of the algorithm depends on the amount of input data
- **Space complexity of an algorithm** — how the amount of a memory used during a computation grows with respect to the size of input

Remark: The precise definitions will be given later.

Remark:

- There are also other types of computational complexity, which we will not discuss here (e.g., communication complexity).

Complexity of an Algorithm

To determine the precise running time or the precise amount of used memory just by an analysis of an algorithm can be extremely difficult.

Usually the analysis of complexity of an algorithm involves many simplifications:

- It is usually not analysed how the running time or the amount of used memory depends precisely on particular input data but how they depend on the **size of the input**.
- Functions expressing how the running time or the amount of used memory grows depending on the size of the input are not computed precisely — instead **estimations** of these functions are computed.
- Estimations of these functions are usually expressed using **asymptotic notation** — e.g., it can be said that the running time of MergeSort is $O(n \log n)$, and that the running time of BubbleSort is $O(n^2)$.

Size of the input — a value describing how “big” is an input instance

- In most cases, the size of an input is just one number — it is usually denoted n or N .
- Sometimes it is more appropriate to express the size of an input by pair (sometimes even with three, four, etc.) of parameters — in this case, they are often denoted n and m (or N and M).
- We can choose what should be considered as the size of an input.

Size of Input

Examples, what the size of an input can be:

- An input is a sequence of some values, an array of elements, etc. (e.g., in problems like sorting, searching in an array, finding the maximal element, etc.):

n — the number of elements in this sequence or array

- An input is a string of characters (a word from some alphabet):

n — the number of characters in this string

- An input consists of two strings, e.g., a (long) text that will be searched through, and a (shorter) searched pattern:

n — the number of characters in the text

m — the number of characters in the searched pattern

- An input is a set of strings:

One possibility:

n — the sum of lengths of all strings

Other variant:

n — the sum of lengths of all strings, m — the number of strings

- The input is a graph:

n — the number of nodes, m — the number of edges

- The input is one number (e.g., in the primality testing):

One possibility:

n — the number of bits of the number — e.g., the size of input 962261 is 20

Other variant:

n — the value of the number — the size of input 962261 is 962261

- The input is a sequence of numbers, and the running time is affected by the values of the numbers (e.g., in the problem where the goal is to compute the greatest common divisor of all numbers in a given sequence):
 n — the sum of numbers of bits of all numbers in the given sequence

Running Time

Let us say that we have:

- an algorithm Alg solving a problem P (resp. a particular implementation of algorithm Alg),
- a machine $\mathcal{M}$ executing the algorithm Alg ,
- an input w from the set In , which is a set of all inputs of problem P

An example:

- a particular implementation of Quicksort in C++ solving the problem of sorting,
- a computer with some particular type of processor working on some particular frequency, with some particular amount of memory, operating system, etc.
- input: array $[6, 13, 1, 8, 4, 5, 8]$
(remark: a more realistic example would be an array with one million elements)

Running Time

$t(w)$ — the running time of the algorithm *Alg* on input w on machine $\mathcal{M}$

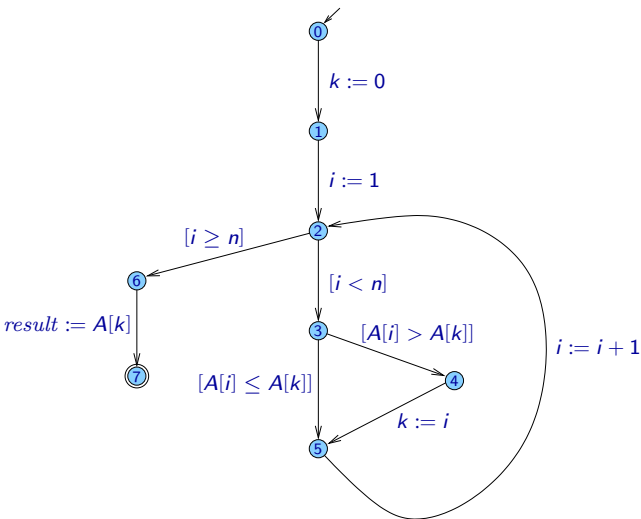
What units should be used for expressing time? (As we will see, this is not important when asymptotic notation is used.)

- **in seconds** — it depends on too many details of implementation, it is difficult to determine it in other way than by measurement (even on the same computer with the same data the running time can fluctuate)
- **the number of steps** — it must be specified what is considered as one step, for example:
 - one statement of a high level programming language
 - one instruction of machine code or bytecode
 - one tick of a processor
 - one operation of some particular type — e.g., a comparison, an arithmetic operation, etc. (while all other operations are ignored)
 - ...

Let us say that an algorithm is represented by a control-flow graph:

- To every instruction (i.e., to every edge) we assign a value specifying how long it takes to perform this instruction once.
- The execution time of different instructions can be different.
- For simplicity we assume that an execution of the same instruction takes always the same time — the value assigned to an instruction is a number from the set $\mathbb{R}^+$ (the set of nonnegative real numbers).

Running Time



Instr.	time
$k := 0$	c_0
$i := 1$	c_1
$[i < n]$	c_2
$[i \geq n]$	c_3
$[A[i] \leq A[k]]$	c_4
$[A[i] > A[k]]$	c_5
$k := i$	c_6
$i := i + 1$	c_7
$result := A[k]$	c_8

Running Time

Example: The execution times of individual instructions could be for example:

Instr.	symbol	time
$k := 0$	c_0	4
$i := 1$	c_1	4
$[i < n]$	c_2	10
$[i \geq n]$	c_3	12
$[A[i] \leq A[k]]$	c_4	14
$[A[i] > A[k]]$	c_5	12
$k := i$	c_6	5
$i := i + 1$	c_7	6
$result := A[k]$	c_8	5

For a particular input w , e.g., for $w = ([3, 8, 4, 5, 2], 5)$, we could simulate the computation and determine the precise running time $t(w)$.

Time Complexity of an Algorithm

Let us say that:

- For a given algorithm Alg and machine $\mathcal{M}$, and for every input w from the set of all inputs In , the running time $t(w)$ is precisely defined.
- To each input w from set In , a number $size(w)$ describing the size of the input w is assigned .

(Formally, it is a function $size : In \rightarrow \mathbb{N}$.)

Definition

The **time complexity of algorithm Alg in the worst case** is the function $T : \mathbb{N} \rightarrow \mathbb{R}^+$ that assigns to each natural number n the maximal running time of the algorithm Alg on an input of size n .

So for each $n \in \mathbb{N}$ we have:

- For each input $w \in In$ such that $size(w) = n$ is $t(w) \leq T(n)$.
- There exists an input $w \in In$ such that $size(w) = n$ and $t(w) = T(n)$.

Time Complexity of an Algorithm

It is obvious from this definition that the time complexity of an algorithm is a function whose precise values depend not only on the given algorithm Alg but also on the following things:

- on a machine $\mathcal{M}$, on which the algorithm Alg runs,
- on the precise definition of the running time $t(w)$ of algorithm Alg on machine $\mathcal{M}$ with input $w \in In$,
- on the precise definition of the size of an input (i.e., on the definition of function $size$).

Time Complexity of an Algorithm

Sometimes, the time complexity in the **average case** is also analyzed:

- Some particular **probabilistic distribution** on the set of inputs must be assumed.
- Instead of the maximal running time on inputs of size n , the expected value of the running times is considered.
- Usually, the analysis of the average case is much more complicated than the analysis of the worst case.
- Often, these two functions are not very different but sometimes the difference is significant.

Remark: It usually makes little sense to analyze the time complexity in the best case.

Time Complexity of an Algorithm

An example of an analysis of the time complexity of algorithm **FIND-MAX** **without** the use of asymptotic notation:

- Such precise analysis is almost never done in practice — it is too tedious and complicated.
- This illustrates what things are ignored in an analysis where asymptotic notation is used and how much the analysis is simplified by this.
- We will compute with constants $c_0, c_1, \dots, c_8$, which specify the execution time of individual instructions — we won't compute with concrete numbers.

Time Complexity of an Algorithm

The inputs are of the form (A, n) , where A is an array and n is the number of elements in this array (where $n \geq 1$).

We take n as the size of input (A, n) .

Consider now some particular input $w = (A, n)$ of size n :

- The running time $t(w)$ on input w can be expressed as

$$t(w) = c_0m_0 + c_1m_1 + \dots + c_8m_8,$$

where $m_0, m_1, \dots, m_8$ are numbers specifying how many times is each instruction performed in the computation on input w .

Time Complexity of an Algorithm

Instr.	time	occurrences	value of m_i
$k := 0$	c_0	m_0	1
$i := 1$	c_1	m_1	1
$[i < n]$	c_2	m_2	$n - 1$
$[i \geq n]$	c_3	m_3	1
$[A[i] \leq A[k]]$	c_4	m_4	$n - 1 - \ell$
$[A[i] > A[k]]$	c_5	m_5	ℓ
$k := i$	c_6	m_6	ℓ
$i := i + 1$	c_7	m_7	$n - 1$
$result := A[k]$	c_8	m_8	1

ℓ — the number of iterations of the cycle where $A[i] > A[k]$
(obviously $0 \leq \ell < n$)

Time Complexity of an Algorithm

By assigning values to

$$t(w) = c_0 m_0 + c_1 m_1 + \dots + c_8 m_8,$$

we obtain

$$t(w) = d_1 + d_2 \cdot (n - 1) + d_3 \cdot (n - 1 - \ell) + d_4 \cdot \ell,$$

where

$$d_1 = c_0 + c_1 + c_3 + c_8$$

$$d_3 = c_4$$

$$d_2 = c_2 + c_7$$

$$d_4 = c_5 + c_6$$

After simplification we have

$$t(w) = (d_2 + d_3) \cdot n + (d_4 - d_3) \cdot \ell + (d_1 - d_2 - d_3)$$

Remark: $t(w)$ is not the time complexity but the running time for a particular input w

Time Complexity of an Algorithm

For example, if the execution times of instructions will be:

Instr.	symb.	time
$k := 0$	c_0	4
$i := 1$	c_1	4
$[i < n]$	c_2	10
$[i \geq n]$	c_3	12
$[A[i] \leq A[k]]$	c_4	14
$[A[i] > A[k]]$	c_5	12
$k := i$	c_6	5
$i := i + 1$	c_7	6
$result := A[k]$	c_8	5

then $d_1 = 25$, $d_2 = 16$, $d_3 = 14$, and $d_4 = 17$.

In this case is $t(w) = 30n + 3\ell - 5$.

For the input $w = ([3, 8, 4, 5, 2], 5)$ is $n = 5$ and $\ell = 1$, therefore $t(w) = 30 \cdot 5 + 3 \cdot 1 - 5 = 148$.

Time Complexity of an Algorithm

It can depend on details of implementation and on the precise values of constants, for which inputs of size n the computation takes the longest time (i.e., which are the worst cases):

The running time of algorithm **FIND-MAX** for an input $w = (A, n)$ of size n :

$$t(w) = (d_2 + d_3) \cdot n + (d_4 - d_3) \cdot \ell + (d_1 - d_2 - d_3)$$

- If $d_3 \geq d_4$ — the worst cases are those where ℓ has the smallest value $\ell = 0$ — for example inputs of the form $[0, 0, \dots, 0]$ or of the form $[n, n-1, n-2, \dots, 2, 1]$
- If $d_3 \leq d_4$ — the worst are those cases where ℓ has the greatest value $\ell = n-1$ — for example inputs of the form $[0, 1, \dots, n-1]$

Time Complexity of an Algorithm

The time complexity $T(n)$ of algorithm **FIND-MAX** in the worst case is given as follows:

- If $d_3 \geq d_4$:

$$T(n) = (d_2 + d_3) \cdot n + (d_1 - d_2 - d_3)$$

- If $d_3 \leq d_4$:

$$\begin{aligned} T(n) &= (d_2 + d_3) \cdot n + (d_4 - d_3) \cdot (n - 1) + (d_1 - d_2 - d_3) \\ &= (d_2 + d_4) \cdot n + (d_1 - d_2 - d_4) \end{aligned}$$

Example: For $d_1 = 25$, $d_2 = 16$, $d_3 = 14$, $d_4 = 17$ is

$$\begin{aligned} T(n) &= (16 + 17) \cdot n + (25 - 16 - 17) \\ &= 33n - 8 \end{aligned}$$

Time Complexity of an Algorithm

In both cases (when $d_3 \geq d_4$ or when $d_3 \leq d_4$), the time complexity of the algorithm **FIND-MAX** is a function

$$T(n) = an + b$$

where a and b are some constants whose precise values depend on the execution time of individual instructions.

Remark: These constants could be expressed as

$$a = d_2 + \max\{d_3, d_4\} \qquad b = d_1 - d_2 - \max\{d_3, d_4\}$$

For example

$$T(n) = 33n - 8$$

Time Complexity of an Algorithm

If it would be sufficient to find out that the time complexity of the algorithm **FIND-MAX** is some function of the form

$$T(n) = an + b,$$

where the precise values of constants a and b would not be important for us, the whole analysis could be considerably simpler.

- In fact, we usually do not want to know precisely how function $T(n)$ look (in general, it can be a very complicated function), and it would be sufficient to know that values of the function $T(n)$ “approximately” correspond to values of a function $S(n) = an + b$, where a and b are some constants.

Time Complexity of an Algorithm

For a given function $T(n)$ expressing the time or space complexity, it is usually sufficient to express it approximately — to have an **estimation** where

- we ignore the less important parts
(e.g., in function $T(n) = 15n^2 + 40n - 5$ we can ignore $40n$ and -5 , and to consider function $T(n) = 15n^2$ instead of the original function),
- we ignore multiplication constants
(e.g., instead of function $T(n) = 15n^2$ we will consider function $T(n) = n^2$)
- we won't ignore constants in exponents — for example there is a big difference between functions $T_1(n) = n^2$ and $T_2(n) = n^3$.
- we will be interested how function $T(n)$ behaves for “big” values of n , we can ignore its behaviour on small values

Growth of Functions

A program works on an input of size n .

Let us assume that for an input of size n , the program performs $T(n)$ operations and that an execution of one operation takes $1\ \mu\text{s}$ (10^{-6} s).

	n							
$T(n)$	20	40	60	80	100	200	500	1000
n	20 μs	40 μs	60 μs	80 μs	0.1 ms	0.2 ms	0.5 ms	1 ms
$n \log n$	86 μs	0.213 ms	0.354 ms	0.506 ms	0.664 ms	1.528 ms	4.48 ms	9.96 ms
n^2	0.4 ms	1.6 ms	3.6 ms	6.4 ms	10 ms	40 ms	0.25 s	1 s
n^3	8 ms	64 ms	0.216 s	0.512 s	1 s	8 s	125 s	16.7 min.
n^4	0.16 s	2.56 s	12.96 s	42 s	100 s	26.6 min.	17.36 hours	11.57 days
2^n	1.05 s	12.75 days	36560 years	$38.3 \cdot 10^9$ years	$40.1 \cdot 10^{15}$ years	$50 \cdot 10^{45}$ years	$10.4 \cdot 10^{136}$ years	–
$n!$	77147 years	$2.59 \cdot 10^{34}$ years	$2.64 \cdot 10^{68}$ years	$2.27 \cdot 10^{105}$ years	$2.96 \cdot 10^{144}$ years	–	–	–

Growth of Functions

Let us consider 3 algorithms with complexities

$T_1(n) = n$, $T_2(n) = n^3$, $T_3(n) = 2^n$. Our computer can do in a reasonable time (the time we are willing to wait) 10^{12} steps.

Complexity	Input size
$T_1(n) = n$	10^{12}
$T_2(n) = n^3$	10^4
$T_3(n) = 2^n$	40

Growth of Functions

Let us consider 3 algorithms with complexities

$T_1(n) = n$, $T_2(n) = n^3$, $T_3(n) = 2^n$. Our computer can do in a reasonable time (the time we are willing to wait) 10^{12} steps.

Complexity	Input size
$T_1(n) = n$	10^{12}
$T_2(n) = n^3$	10^4
$T_3(n) = 2^n$	40

Now we speed up our computer 1000 times, meaning it can do 10^{15} steps.

Complexity	Input size	Growth
$T_1(n) = n$	10^{15}	$1000\times$
$T_2(n) = n^3$	10^5	$10\times$
$T_3(n) = 2^n$	50	+10

Asymptotic Notation

In the following, we will consider functions of the form $f : \mathbb{N} \rightarrow \mathbb{R}$, where:

- The values of $f(n)$ need not to be defined for all values of $n \in \mathbb{N}$ but there must exist some constant n_0 such that the value of $f(n)$ is defined for all $n \in \mathbb{N}$ such that $n \geq n_0$.

Example: Function $f(n) = \log_2(n)$ is not defined for $n = 0$ but it is defined for all $n \geq 1$.

- There must exist a constant n_0 such that for all $n \in \mathbb{N}$, where $n \geq n_0$, is $f(n) \geq 0$.

Example: It holds for function $f(n) = n^2 - 25$ that $f(n) \geq 0$ for all $n \geq 5$.

Asymptotic Notation

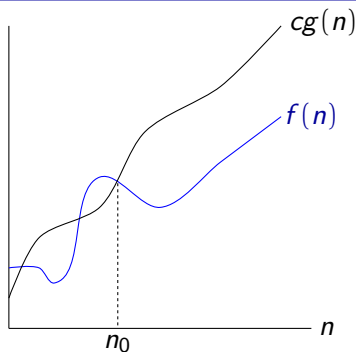
Let us take an arbitrary function $f : \mathbb{N} \rightarrow \mathbb{R}$. Expressions $O(f)$, $\Omega(f)$, and $\Theta(f)$ denote **sets of functions** of the type $\mathbb{N} \rightarrow \mathbb{R}$, where:

- $O(f)$ – the set of all functions that grow at most as fast as f
- $\Omega(f)$ – the set of all functions that grow at least as fast as f
- $\Theta(f)$ – the set of all functions that grow as fast as f

Remark: These are not definitions! The definitions will follow on the next slides.

- O – big “O”
- Ω – uppercase Greek letter “omega”
- Θ – uppercase Greek letter “theta”

Asymptotic Notation – Symbol O



Definition

Let us consider an arbitrary function $g : \mathbb{N} \rightarrow \mathbb{R}$. For a function $f : \mathbb{N} \rightarrow \mathbb{R}$ we have $f \in O(g)$ iff

$$(\exists c > 0)(\exists n_0 \geq 0)(\forall n \geq n_0)(f(n) \leq c g(n)).$$

Remarks:

- c is a positive real number (i.e., $c \in \mathbb{R}$ and $c > 0$)
- n_0 and n are natural numbers (i.e., $n_0 \in \mathbb{N}$ and $n \in \mathbb{N}$)

Asymptotic Notation – Symbol O

Example: Let us consider functions $f(n) = 2n^2 + 3n + 7$ and $g(n) = n^2$.

We want to show that $f \in O(g)$, i.e., $f \in O(n^2)$:

- Approach 1:

Let us take for example $c = 3$.

$$cg(n) = 3n^2 = 2n^2 + \frac{1}{2}n^2 + \frac{1}{2}n^2$$

We need to find some n_0 such that for all $n \geq n_0$ it holds that

$$2n^2 \geq 2n^2 \qquad \frac{1}{2}n^2 \geq 3n \qquad \frac{1}{2}n^2 \geq 7$$

We can easily check that for example $n_0 = 6$ satisfies this.

For each $n \geq 6$ we have $cg(n) \geq f(n)$:

$$cg(n) = 3n^2 = 2n^2 + \frac{1}{2}n^2 + \frac{1}{2}n^2 \geq 2n^2 + 3n + 7 = f(n)$$

Asymptotic Notation – Symbol O

The example where $f(n) = 2n^2 + 3n + 7$ and $g(n) = n^2$:

- Approach 2:

Let us take $c = 12$.

$$cg(n) = 12n^2 = 2n^2 + 3n^2 + 7n^2$$

We need to find some n_0 such that for all $n \geq n_0$ we have

$$2n^2 \geq 2n^2 \qquad 3n^2 \geq 3n \qquad 7n^2 \geq 7$$

These inequalities obviously hold for $n_0 = 1$, and so for each $n \geq 1$ we have $cg(n) \geq f(n)$:

$$cg(n) = 12n^2 = 2n^2 + 3n^2 + 7n^2 \geq 2n^2 + 3n + 7 = f(n)$$

Asymptotic Notation – Symbol O

Proposition

Let us assume that a and b are constants such that $a > 0$ and $b > 0$, and k and ℓ are some arbitrary constants where $k \geq 0$, $\ell \geq 0$ and $k < \ell$.

Let us consider functions

$$f(n) = a \cdot n^k \qquad g(n) = b \cdot n^\ell$$

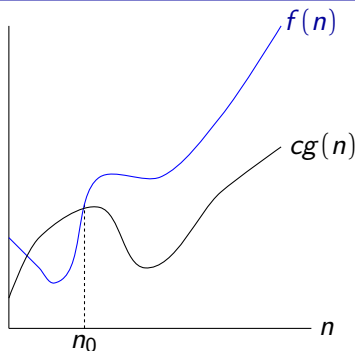
For each such functions f and g it holds that $f \in O(g)$:

Proof: Let us take $c = \frac{a}{b}$.

Because for $n \geq 1$ we obviously have $n^k \leq n^\ell$ (since $k \leq \ell$), for $n \geq 1$ we have

$$c \cdot g(n) = \frac{a}{b} \cdot g(n) = \frac{a}{b} \cdot b \cdot n^\ell = a \cdot n^\ell \geq a \cdot n^k = f(n)$$

Asymptotic Notation – Symbol Ω



Definition

Let us consider an arbitrary function $g : \mathbb{N} \rightarrow \mathbb{R}$. For a function $f : \mathbb{N} \rightarrow \mathbb{R}$ we have $f \in \Omega(g)$ iff

$$(\exists c > 0)(\exists n_0 \geq 0)(\forall n \geq n_0)(c g(n) \leq f(n)).$$

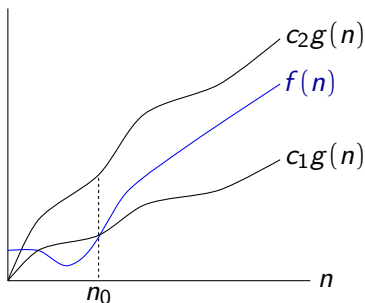
Asymptotic Notation – Symbol Ω

It is not difficult to prove the following proposition:

For arbitrary functions f and g we have:

$$f \in O(g) \quad \text{iff} \quad g \in \Omega(f)$$

Asymptotic Notation – Symbol Θ



Definition

Let us consider an arbitrary function $g : \mathbb{N} \rightarrow \mathbb{R}$. For a function $f : \mathbb{N} \rightarrow \mathbb{R}$ we have $f \in \Theta(g)$ iff

$$(\exists c_1 > 0)(\exists c_2 > 0)(\exists n_0 \geq 0)(\forall n \geq n_0)(c_1 g(n) \leq f(n) \leq c_2 g(n)).$$

Asymptotic Notation – Symbol Θ

For arbitrary functions f and g we have:

$$f \in \Theta(g) \quad \text{iff} \quad f \in O(g) \text{ a } f \in \Omega(g)$$

$$f \in \Theta(g) \quad \text{iff} \quad f \in O(g) \text{ a } g \in O(f)$$

$$f \in \Theta(g) \quad \text{iff} \quad g \in \Theta(f)$$

Asymptotic Notation

For arbitrary functions f , g , and h we have:

- if $f \in O(g)$ and $g \in O(h)$ then $f \in O(h)$
- if $f \in \Omega(g)$ and $g \in \Omega(h)$ then $f \in \Omega(h)$
- if $f \in \Theta(g)$ and $g \in \Theta(h)$ then $f \in \Theta(h)$

Examples:

$$n \in O(n^2)$$

$$1000n \in O(n)$$

$$2^{\log_2 n} \in \Theta(n)$$

$$n^3 \notin O(n^2)$$

$$n^2 \notin O(n)$$

$$n^3 + 2^n \notin O(n^2)$$

$$n^3 \in O(n^4)$$

$$0.00001n^2 - 10^{10}n \in \Theta(10^{10}n^2)$$

$$n^3 - n^2 \log_2^3 n + 1000n - 10^{100} \in \Theta(n^3)$$

$$n^3 + 1000n - 10^{100} \in O(n^3)$$

$$n^3 + n^2 \notin \Theta(n^2)$$

$$n! \notin O(2^n)$$

- There are pairs of functions f and g such that

$$f \notin O(g) \quad \text{and} \quad g \notin O(f),$$

for example

$$f(n) = n \qquad g(n) = n^{1+\sin(n)}.$$

- $O(1)$ is the set of all **bounded** functions, i.e., functions whose function values can be bounded from above by a constant.

Asymptotic Notation

- For any pair of functions f, g we have:
 - $\max(f, g) \in \Theta(f + g)$
 - if $f \in O(g)$ then $f + g \in \Theta(g)$
- For any functions f_1, f_2, g_1, g_2 we have:
 - if $f_1 \in O(f_2)$ and $g_1 \in O(g_2)$ then $f_1 + g_1 \in O(f_2 + g_2)$ and $f_1 \cdot g_1 \in O(f_2 \cdot g_2)$
 - if $f_1 \in \Theta(f_2)$ and $g_1 \in \Theta(g_2)$ then $f_1 + g_1 \in \Theta(f_2 + g_2)$ and $f_1 \cdot g_1 \in \Theta(f_2 \cdot g_2)$

Asymptotic Notation

- A function f is called:
 - logarithmic**, if $f(n) \in \Theta(\log n)$
 - linear**, if $f(n) \in \Theta(n)$
 - quadratic**, if $f(n) \in \Theta(n^2)$
 - cubic**, if $f(n) \in \Theta(n^3)$
 - polynomial**, if $f(n) \in O(n^k)$ for some $k > 0$
 - exponential**, if $f(n) \in O(c^{n^k})$ for some $c > 1$ and $k > 0$
- Exponential functions are often written in the form $2^{O(n^k)}$ when the asymptotic notation is used, since then we do not need to consider different bases.

Asymptotic Notation

As mentioned before, expressions $O(g)$, $\Omega(g)$, and $\Theta(g)$ denote certain sets of functions.

In some texts, these expressions are sometimes used with a slightly different meaning:

- an expression $O(g)$, $\Omega(g)$ or $\Theta(g)$ does not represent the corresponding set of functions but **some** function from this set.

This convention is often used in equations and inequations.

Example: $3n^3 + 5n^2 - 11n + 2 = 3n^3 + O(n^2)$

When using this convention, we can for example write $f = O(g)$ instead of $f \in O(g)$.

Complexity of Algorithms

Let us say we would like to analyze the time complexity $T(n)$ of some algorithm consisting of instructions $l_1, l_2, \dots, l_k$:

- If $m_1, m_2, \dots, m_k$ are the numbers of executions of individual instructions for some input w (i.e., the instruction l_i is performed m_i times for the input w), then the total number of executed instructions for input w is

$$T(n) = c_1 m_1 + c_2 m_2 + \dots + c_k m_k.$$

- Let us consider functions $f_1, f_2, \dots, f_k$, where $f_i : \mathbb{N} \rightarrow \mathbb{R}$, and where $f_i(n)$ is the maximum of numbers of executions of instruction l_i for all inputs of size n .
- Obviously, $T \in \Omega(f_i)$ for any function f_i .
- It is also obvious that $T \in O(f_1 + f_2 + \dots + f_k)$.

- Let us recall that if $f \in O(g)$ then $f + g \in O(g)$.
- If there is a function f_i such that for all f_j , where $j \neq i$, we have $f_j \in O(f_i)$, then

$$T \in O(f_i).$$

- This means that in an analysis of the time complexity $T(n)$, we can restrict our attention to the number of executions of the instruction that is performed most frequently (and which is performed at most $f_i(n)$ times for an input of size n), since we have

$$T \in \Theta(f_i).$$

Complexity of Algorithms

Example: In the analysis of the complexity of the searching of a number in a sequence we obtained

$$f(n) = an + b.$$

If we would not like to do such a detailed analysis, we could deduce that the time complexity of the algorithm is $\Theta(n)$, because:

- The algorithm contains only one cycle, which is performed $(n - 1)$ times for an input of size n , the number of iterations of the cycle is in $\Theta(n)$.
- Several instructions are performed in one iteration of the cycle. The number of these instructions is bounded from both above and below by some constant independent on the size of the input.
- Other instructions are performed at most once, and so they contribute to the total running time by adding a constant.

Complexity of Algorithms

Let us try to analyze the time complexity of the following algorithm:

Algorithm 2: Insertion sort

```
1 INSERTION-SORT ( $A, n$ ):  
2 begin  
3   for  $j := 1$  to  $n - 1$  do  
4      $x := A[j]$   
5      $i := j - 1$   
6     while  $i \geq 0$  and  $A[i] > x$  do  
7        $A[i + 1] := A[i]$   
8        $i := i - 1$   
9     end  
10     $A[i + 1] := x$   
11  end  
12 end
```

I.e., we want to find a function $T(n)$ such that the time complexity of the algorithm INSERTION-SORT in the worst case is in $\Theta(T(n))$.

Complexity of Algorithms

Let us consider inputs of size n :

- The outer cycle **for** is performed at most $n - 1$ times.
- The inner cycle **while** is performed at most $(j - 1)$ times for a given value j .
- There are inputs such that the cycle **while** is performed exactly $(j - 1)$ times for each value j from 2 to n .
- So in the worst case, the cycle **while** is performed exactly m times, where

$$m = 1 + 2 + \dots + (n - 1) = (1 + (n - 1)) \cdot \frac{n-1}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$$

- This means that the total running time of the algorithm **INSERTION-SORT** in the worst case is $\Theta(n^2)$.

In the previous case, we have computed the total number of executions of the cycle **while** accurately.

This is not always possible in general, or it can be quite complicated. It is also not necessary, if we only want an asymptotic estimation.

Complexity of Algorithms

For example, if we were not able to compute the sum of the arithmetic progression, we could proceed as follows:

- The outer cycle **for** is not performed more than n times and the inner cycle **while** is performed at most n times in each iteration of the outer cycle.

So we have $T \in O(n^2)$.

- For some inputs, the cycle **while** is performed at least $\lceil n/2 \rceil$ times in the last $\lfloor n/2 \rfloor$ iterations of the cycle **for**.

So the cycle **while** is performed at least $\lfloor n/2 \rfloor \cdot \lceil n/2 \rceil$ times for some inputs.

$$\lfloor n/2 \rfloor \cdot \lceil n/2 \rceil \geq (n/2 - 1) \cdot (n/2) = \frac{1}{4}n^2 - \frac{1}{2}n$$

This implies $T \in \Omega(n^2)$.

Space Complexity of Algorithms

- So far we have considered only the time necessary for a computation
- Sometimes the size of the memory necessary for the computation is more critical.

The **amount of memory** used by machine $\mathcal{M}$ in a computation on input w can be for example:

- the maximal number of bits necessary for storing all data for each configuration
- the maximal number of memory cells used during the computation

Definition

A **space complexity** of algorithm Alg running on machine $\mathcal{M}$ is the function $S : \mathbb{N} \rightarrow \mathbb{N}$, where $S(n)$ is the maximal amount of memory used by $\mathcal{M}$ for inputs of size n .

Space Complexity of Algorithms

- There can be two algorithms for a particular problem such that one of them has a smaller time complexity and the other a smaller space complexity.
- If the time-complexity of an algorithm is in $O(f(n))$ then also the space complexity is in $O(f(n))$ (note that the number of memory cells used in one instruction is bounded by some constant that does not depend on the size of an input).
- The space complexity can be much smaller than the time complexity — the space complexity of **INSERTION-SORT** is $\Theta(n)$, while its time complexity is $\Theta(n^2)$.

Complexity of Algorithms

Some typical values of the size of an input n , for which an algorithm with the given time complexity usually computes the output on a “common PC” within a fraction of a second or at most in seconds.

(Of course, this depends on particular details. Moreover, it is assumed here that no big constants are hidden in the asymptotic notation)

$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^3)$
1 000 000 – 100 000 000	100 000 – 1 000 000	1000 – 10 000	100 – 1000
	$2^{O(n)}$	$O(n!)$	
	20 – 30	10 – 15	

Complexity of Algorithms

When we use asymptotic estimations of the complexity of algorithms, we should be aware of some issues:

- Asymptotic estimations describe only how the running time grows with the growing size of input instance.
- They do not say anything about exact running time. Some big constants can be hidden in the asymptotic notation.
- An algorithm with better asymptotic complexity than some other algorithm can be in reality faster only for very big inputs.
- We usually analyze the time complexity in the worst case. For some algorithms, the running time in the worst case can be much higher than the running time on “typical” instances.

- This can be illustrated on algorithms for sorting.

Algorithm	Worst-case	Average-case
Bubblesort	$\Theta(n^2)$	$\Theta(n^2)$
Heapsort	$\Theta(n \log n)$	$\Theta(n \log n)$
Quicksort	$\Theta(n^2)$	$\Theta(n \log n)$

- Quicksort has a worse asymptotic complexity in the worst case than Heapsort and the same asymptotic complexity in an average case but it is usually faster in practice.

Complexity of Algorithms

Polynomial — an expression of the form

$$a_k n^k + a_{k-1} n^{k-1} + \dots + a_2 n^2 + a_1 n + a_0$$

where $a_0, a_1, \dots, a_k$ are constants.

Examples of polynomials:

$$4n^3 - 2n^2 + 8n + 13$$

$$2n + 1$$

$$n^{100}$$

Function f is called **polynomial** if it is bounded from above by some polynomial, i.e., if there exists a constant k such that $f \in O(n^k)$.

For example, the functions belonging to the following classes are polynomial:

$$O(n)$$

$$O(n \log n)$$

$$O(n^2)$$

$$O(n^5)$$

$$O(\sqrt{n})$$

$$O(n^{100})$$

Complexity of Algorithms

Function such as 2^n or $n!$ are not polynomial — for arbitrarily big constant k we have

$$2^n \in \Omega(n^k)$$

$$n! \in \Omega(n^k)$$

Polynomial algorithm — an algorithm whose time complexity is polynomial (i.e., bounded from above by some polynomial)

Roughly we can say that:

- polynomial algorithms are efficient algorithms that can be used in practice for inputs of considerable size
- algorithms, which are not polynomial, can be used in practice only for rather small inputs

Complexity of Algorithms

The division of algorithms on polynomial and non-polynomial is very rough — we cannot claim that polynomial algorithms are always efficient and non-polynomial algorithms are not:

- an algorithm with the time complexity $\Theta(n^{100})$ is probably not very useful in practice,
- some algorithms, which are non-polynomial, can still work very efficiently for majority of inputs, and can have a time complexity bigger than polynomial only due to some problematic inputs, on which the computation takes long time.

Remark: Polynomial algorithms where the constant in the exponent is some big number (e.g., algorithms with complexity $\Theta(n^{100})$) almost never occur in practice as solutions of usual algorithmic problems.

Complexity of Algorithms

For most of common algorithmic problems, one of the following three possibilities happens:

- A polynomial algorithm with time complexity $O(n^k)$ is known, where k is some very small number (e.g., 5 or more often 3 or less).
- No polynomial algorithm is known and the best known algorithms have complexities such as $2^{\Theta(n)}$, $\Theta(n!)$, or some even bigger.

In some cases, a proof is known that there does not exist a polynomial algorithm for the given problem (it cannot be constructed).

- No algorithm solving the given problem is known (and it is possibly proved that there does not exist such algorithm)

Complexity of Algorithms

A typical example of polynomial algorithm — matrix multiplication with time complexity $\Theta(n^3)$ and space complexity $\Theta(n^2)$:

Algorithm 3: Matrix multiplication

```
1  MATRIX-MULT ( $A, B, C, n$ ):  
2  begin  
3      for  $i := 1$  to  $n$  do  
4          for  $j := 1$  to  $n$  do  
5               $x := 0$   
6              for  $k := 1$  to  $n$  do  
7                   $x := x + A[i][k] * B[k][j]$   
8              end  
9               $C[i][j] := x$   
10         end  
11     end  
12 end
```

Complexity of Algorithms

- For a rough estimation of complexity, it is often sufficient to count the number of nested loops — this number then gives the degree of the polynomial

Example: Three nested loops in the matrix multiplication — the time complexity of the algorithm is $O(n^3)$.

- If it is not the case that all the loops go from 0 to n but the number of iterations of inner loops are different for different iterations of an outer loops, a more precise analysis can be more complicated.
It is often the case, that the sum of some sequence (e.g., the sum of arithmetic or geometric progression) is then computed in the analysis.
The results of such more detailed analysis often does not differ from the results of a rough analysis but in many cases the time complexity resulting from a more detailed analysis can be considerably smaller than the time complexity following from the rough analysis.

Arithmetic progression — a sequence of numbers $a_0, a_1, \dots, a_{n-1}$, where

$$a_i = a_0 + i \cdot d,$$

where d is some constant independent on i .

Remark: So in an arithmetic progression, we have $a_{i+1} = a_i + d$ for each i .

The sum of an arithmetic progression:

$$\sum_{i=0}^{n-1} a_i = a_0 + a_1 + \dots + a_{n-1} = \frac{1}{2}n(a_{n-1} + a_0)$$

Example:

$$1 + 2 + \dots + n = \frac{1}{2}n(n+1) = \frac{1}{2}n^2 + \frac{1}{2}n = \Theta(n^2)$$

For example, for $n = 100$ we have

$$1 + 2 + \dots + 100 = 50 \cdot 101 = 5050.$$

Remark: To see this, we can note that

$$1 + 2 + \dots + 100 = (1 + 100) + (2 + 99) + \dots + (50 + 51),$$

where we compute the sum of 50 pairs of number, where the sum of each pair is 101.

Geometric progression — a sequence of numbers $a_0, a_1, \dots, a_n$, where

$$a_i = a_0 \cdot q^i,$$

where q is some constant independent on i .

Remark: So in a geometric progression we have $a_{i+1} = a_i \cdot q$.

The sum of a geometric progression (where $q \neq 1$):

$$\sum_{i=0}^n a_i = a_0 + a_1 + \dots + a_n = a_0 \frac{q^{n+1} - 1}{q - 1}$$

Example:

$$1 + q + q^2 + \dots + q^n = \frac{q^{n+1} - 1}{q - 1}$$

In particular, for $q = 2$:

$$1 + 2^1 + 2^2 + 2^3 + \dots + 2^n = \frac{2^{n+1} - 1}{2 - 1} = 2 \cdot 2^n - 1 = \Theta(2^n)$$

Complexity of Algorithms

An **exponential** function: a function of the form c^n , where c is a constant — e.g., function 2^n

Logarithm — the inverse function to an exponential function: for a given n ,

$$\log_c n$$

is the value x such that $c^x = n$.

Complexity of Algorithms

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024
11	2048
12	4096
13	8192
14	16384
15	32768
16	65536
17	131072
18	262144
19	524288
20	1048576

n	$\lceil \log_2 n \rceil$
0	—
1	0
2	1
3	2
4	2
5	3
6	3
7	3
8	3
9	4
10	4
11	4
12	4
13	4
14	4
15	4
16	4
17	5
18	5
19	5
20	5

n	$\log_2 n$
1	0
2	1
4	2
8	3
16	4
32	5
64	6
128	7
256	8
512	9
1024	10
2048	11
4096	12
8192	13
16384	14
32768	15
65536	16
131072	17
262144	18
524288	19
1048576	20

Proposition

For any $a, b > 1$ and any $n > 0$ we have

$$\log_a n = \frac{\log_b n}{\log_b a}$$

Proof: From $n = a^{\log_a n}$ it follows that $\log_b n = \log_b(a^{\log_a n})$. Since $\log_b(a^{\log_a n}) = \log_a n \cdot \log_b a$, we obtain $\log_b n = \log_a n \cdot \log_b a$, from which the above mentioned conclusion follows directly. $\square$

Due to this observation, the base of a logarithm is often omitted in the asymptotic notation: for example, instead of $\Theta(n \log_2 n)$ we can write $\Theta(n \log n)$.

Complexity of Algorithms

Examples where exponential functions and logarithms can appear in an analysis of algorithms:

- Some value is repeatedly decreased to one half or is repeatedly doubled.

For example, in the **binary search**, the size of an interval halves in every iteration of the loop.

Let us assume that an array has size n .

What is the minimal size of an array n , for which the algorithm performs at least k iterations?

The answer: 2^k

So we have $k = \log_2(n)$. The time complexity of the algorithm is then $\Theta(\log n)$.

Complexity of Algorithms

- Using n bits we can represent numbers from 0 to $2^n - 1$.
- The minimal numbers of bits, which are sufficient for representing a natural number x in binary is

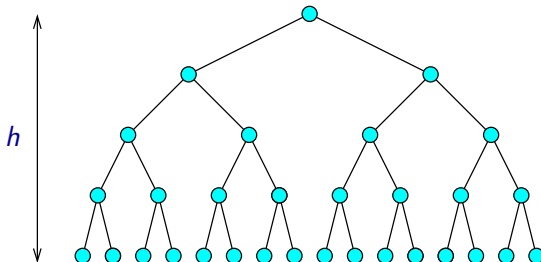
$$\lceil \log_2(x + 1) \rceil.$$

- A perfectly balanced tree of height h has $2^{h+1} - 1$ nodes, and 2^h of these nodes are leaves.
- The height of a perfectly balanced binary tree with n nodes is $\log_2 n$.

An illustrating example: If we would draw a balanced tree with $n = 1\,000\,000$ nodes in such a way that the distance between neighbouring nodes would be 1 cm and the height of each layer of nodes would be also 1 cm, the width of the tree would be 10 km and its height would be approximately 20 cm.

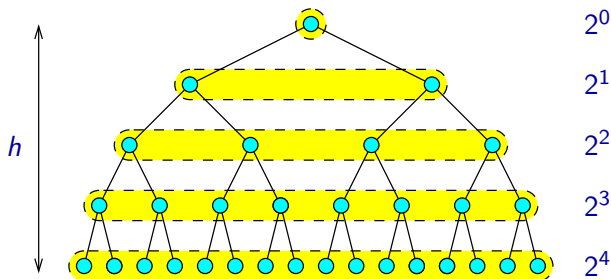
Complexity of Algorithms

A perfectly balanced binary tree of height h :



Complexity of Algorithms

A perfectly balanced binary tree of height h :



Example: Algorithm `MERGE-SORT`.

The main idea of the algorithm: Two sorted sequences can be easily merged into one sorted sequence.

If both sequences have together n elements then this operation can be done in n steps.

34	42	58	61
----	----	----	----

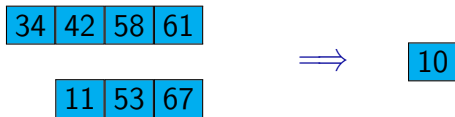


10	11	53	67
----	----	----	----

Example: Algorithm **MERGE-SORT**.

The main idea of the algorithm: Two sorted sequences can be easily merged into one sorted sequence.

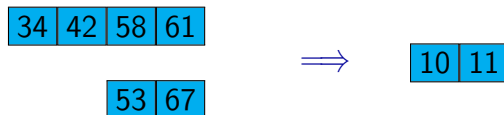
If both sequences have together n elements then this operation can be done in n steps.



Example: Algorithm **MERGE-SORT**.

The main idea of the algorithm: Two sorted sequences can be easily merged into one sorted sequence.

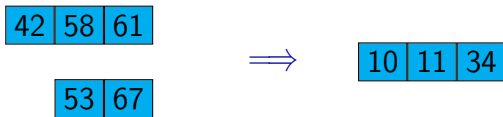
If both sequences have together n elements then this operation can be done in n steps.



Example: Algorithm **MERGE-SORT**.

The main idea of the algorithm: Two sorted sequences can be easily merged into one sorted sequence.

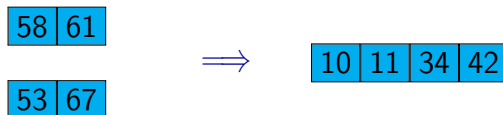
If both sequences have together n elements then this operation can be done in n steps.



Example: Algorithm `MERGE-SORT`.

The main idea of the algorithm: Two sorted sequences can be easily merged into one sorted sequence.

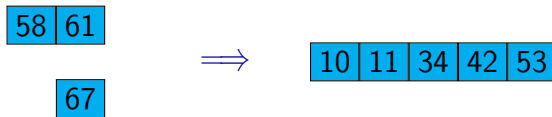
If both sequences have together n elements then this operation can be done in n steps.



Example: Algorithm `MERGE-SORT`.

The main idea of the algorithm: Two sorted sequences can be easily merged into one sorted sequence.

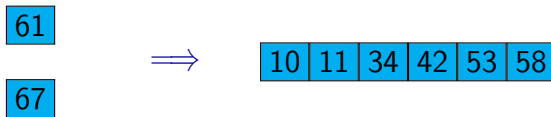
If both sequences have together n elements then this operation can be done in n steps.



Example: Algorithm `MERGE-SORT`.

The main idea of the algorithm: Two sorted sequences can be easily merged into one sorted sequence.

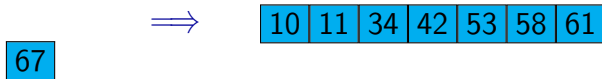
If both sequences have together n elements then this operation can be done in n steps.



Example: Algorithm `MERGE-SORT`.

The main idea of the algorithm: Two sorted sequences can be easily merged into one sorted sequence.

If both sequences have together n elements then this operation can be done in n steps.



Example: Algorithm `MERGE-SORT`.

The main idea of the algorithm: Two sorted sequences can be easily merged into one sorted sequence.

If both sequences have together n elements then this operation can be done in n steps.



Complexity of Algorithms

Algorithm 4: Merge sort

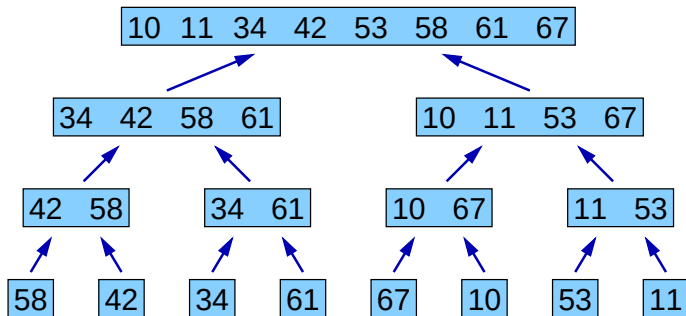
```
1 MERGE-SORT ( $A, p, r$ ):  
2 begin  
3   if  $r - p > 1$  then  
4      $q := \lfloor (p + r) / 2 \rfloor$   
5     MERGE-SORT( $A, p, q$ )  
6     MERGE-SORT( $A, q, r$ )  
7     MERGE( $A, p, q, r$ )  
8   end  
9 end
```

To sort an array A containing elements $A[0], A[1], \dots, A[n-1]$ we call $\text{MERGE-SORT}(A, 0, n)$.

Remark: Procedure $\text{MERGE}(A, p, q, r)$ merges sorted sequences stored in $A[p \dots q-1]$ and $A[q \dots r-1]$ into one sequence stored in $A[p \dots r-1]$.

Complexity of Algorithms

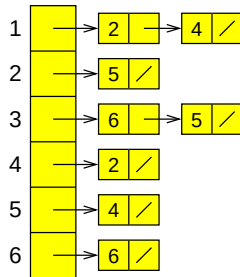
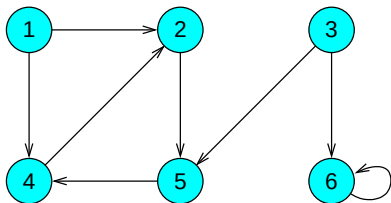
Input: 58, 42, 34, 61, 67, 10, 53, 11



The tree of recursive calls has $\Theta(\log n)$ layers. On each layer, $\Theta(n)$ operations are performed. The time complexity of **MERGE-SORT** is $\Theta(n \log n)$.

Complexity of Algorithms

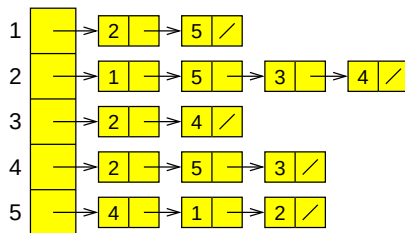
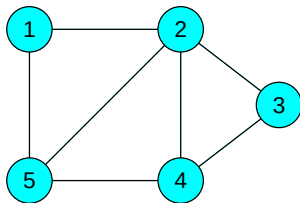
Representations of a graph:



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Complexity of Algorithms

Representations of a graph:



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Finding the shortest path in a graph where edges are not weighted:

- Breadth-first search
- The input is a graph G (with a set of nodes V) and an initial node s .
- The algorithm finds the shortest paths from node s for all nodes.
- For a graph with n nodes and m edges, the running time of the algorithm is $\Theta(n + m)$.

Complexity of Algorithms

Algorithm 5: Breadth-first search

```
1 BFS ( $G, s$ ):
2 begin
3   BFS-INIT( $G, s$ )
4   ENQUEUE( $Q, s$ )
5   while  $Q \neq \emptyset$  do
6      $u :=$  DEQUEUE( $Q$ )
7     for each  $v \in \text{edges}[u]$  do
8       if  $\text{color}[v] = \text{WHITE}$  then
9          $\text{color}[v] := \text{GRAY}$ 
10         $d[v] := d[u] + 1$ 
11         $\text{pred}[v] := u$ 
12        ENQUEUE( $Q, v$ )
13      end
14    end
15     $\text{color}[u] := \text{BLACK}$ 
16  end
17 end
```

Complexity of Algorithms

Algorithm 6: Breadth-first search — initialization

```
1 BFS-INIT ( $G, s$ ):  
2 begin  
3   for each  $u \in V - \{s\}$  do  
4      $color[u] := \text{WHITE}$   
5      $d[u] := \infty$   
6      $pred[u] := \text{NIL}$   
7   end  
8    $color[s] := \text{GRAY}$   
9    $d[s] := 0$   
10   $pred[s] := \text{NIL}$   
11   $Q := \emptyset$   
12 end
```

Problem “Primality”

Input: A natural number x .

Output: YES if x is a prime, NO otherwise.

Remark: A natural number x is a **prime** if it is greater than 1 and is divisible only by numbers 1 and x .

Few of the first primes: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, ...

The problems, where the set of outputs is $\{\text{YES}, \text{NO}\}$ are called **decision problems**.

Decision problems are usually specified in such a way that instead of describing what the output is, a question is formulated.

Example:

Problem “Primality”

Input: A natural number x .

Question: Is x a prime?

Primality Test

A simple algorithm solving the “Primality” problem can work like this:

- Test the trivial cases (e.g., if $x \leq 2$ or if x is even).
- Try to divide x successively by all odd numbers in interval $3, \dots, \lfloor \sqrt{x} \rfloor$.

Let n be the number of bits in the representation of number x , e.g., $n = \lceil \log_2(x + 1) \rceil$.

This value n will be considered as the size of the input.

Note that the number $\lfloor \sqrt{x} \rfloor$ has approximately $n/2$ bits.

There are approximately $2^{(n/2)-1}$ odd numbers in the interval $3, \dots, \lfloor \sqrt{x} \rfloor$, and so the time complexity of this simple algorithm is in $2^{\Theta(n)}$.

Primality Test

This simple algorithm with an exponential running time (resp. also different improved versions of this) are applicable to numbers with thousands of bits in practice.

A primality test of such big numbers plays an important role for example in **cryptography**.

Only since 2003, a polynomial time algorithm is known. The time complexity of the original version of the algorithm was $O(n^{12+\epsilon})$, later it was improved to $(O(n^{7.5}))$. The currently fastest algorithm has time complexity $O(n^6)$.

In practice, **randomized algorithms** are used for primality testing:

- Solovay–Strassen
- Miller–Rabin

(The time complexity of both algorithms is $O(n^3)$.)

Primality Test

A **randomized algorithm**:

- It uses a random-number generator during a computation.
- It can produce different outputs in different runs with the same input.
- The output need not be always correct but the probability of producing an incorrect output is bounded.

For example, both above mentioned randomized algorithms for primality testing behave as follows:

- If x is a prime, the answer **YES** is always returned.
- If x is not a prime, the probability of the answer **No** is at least 50% but there is at most 50% probability that the program returns the incorrect answer **YES**.

Primality Test

The program can be run repeatedly (k times):

- If the program returns at least once the answer **No**, we know (with 100% probability) that x is not a prime.
- If the program always returns **Yes**, the probability that x is not a prime is at most $\frac{1}{2^k}$.

For sufficiently large values of k , the probability of an incorrect answer is negligible.

Remark: For example for $k = 100$, the probability of this error is smaller than the probability that a computer, on which the program is running, will be destroyed by a falling meteorite (assuming that at least once in every 1000 years at least 100 m^2 of Earth surface is destroyed by a meteorite).

At first sight, the following problem looks very similar as the primality test:

Problem “Factorization”

Input: A natural number x , where $x > 1$.

Output: Primes $p_1, p_2, \dots, p_m$ such that $x = p_1 \cdot p_2 \cdot \dots \cdot p_m$.

In fact, this problem is (supposed to be) much harder than primality testing.

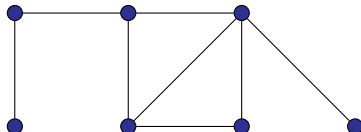
No efficient (polynomial) algorithm is known for this problem (nor a randomized algorithm).

Other Examples of Problems

Independent set (IS) problem

Input: An undirected graph G , a number k .

Question: Is there an independent set of size k in the graph G ?



$k = 4$

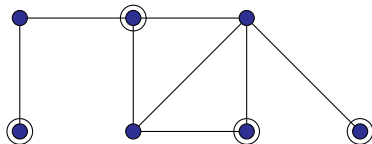
Remark: An **independent set** in a graph is a subset of nodes of the graph such that no pair of nodes from this set is connected by an edge.

Other Examples of Problems

Independent set (IS) problem

Input: An undirected graph G , a number k .

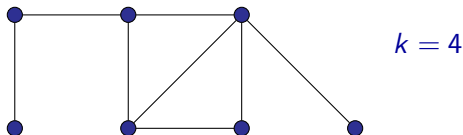
Question: Is there an independent set of size k in the graph G ?

 $k = 4$

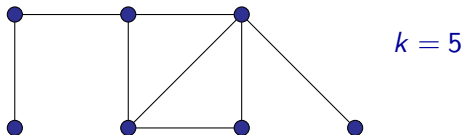
Remark: An **independent set** in a graph is a subset of nodes of the graph such that no pair of nodes from this set is connected by an edge.

Other Examples of Problems

An example of an instance where the answer is **YES**:



An example of an instance where the answer is **No**:



Other Examples of Problems

- A set containing n elements has 2^n subsets.

Consider for example an algorithm solving a given problem by **brute force** where it tests the required property for each subset of a given set.

- It is sufficient to consider only subsets of size k . The total number of such subsets is

$$\binom{n}{k}$$

For some values of k , the total number of these subsets is not much smaller than 2^n :

For example, it is not too difficult to show that

$$\binom{n}{\lfloor n/2 \rfloor} \geq \frac{2^n}{n}.$$

Other Examples of Problems

Let us have an algorithm solving the independent set problem by brute force in such a way that it tests for each subset with k elements of the set of nodes (with n nodes), if it forms an independent set. The time complexity of the algorithm is $2^{\Theta(n)}$.

Undecidable Problems

Algorithmically Solvable Problems

Let us assume we have a problem P .

If there is an algorithm solving the problem P then we say that the problem P is **algorithmically solvable**.

If P is a decision problem and there is an algorithm solving the problem P then we say that the problem P is **decidable (by an algorithm)**.

If we want to show that a problem P is algorithmically solvable, it is sufficient to show some algorithm solving it (and possibly show that the algorithm really solves the problem P).

Algorithmically Unsolvable Problems

A problem that is not algorithmically solvable is **algorithmically unsolvable**.

A decision problem that is not decidable is **undecidable**.

Surprisingly, there are many (exactly defined) problems, for which it was proved that they are not algorithmically solvable.

Halting Problem

Let us consider some general programming language $\mathcal{L}$.

Futhermore, let us assume that programs in language $\mathcal{L}$ run on some idealized machine where a (potentially) unbounded amount of memory is available — i.e., the allocation of memory never fails.

Example: The following problem called the **Halting problem** is undecidable:

Halting problem

Input: A source code of a $\mathcal{L}$ program P , input data x .

Question: Does the computation of P on the input x halt after some finite number of steps?

Halting Problem

Let us assume that there is a program that can decide the Halting problem.

So we could construct a subroutine H , declared as

Bool $H(\text{String code}, \text{String input})$

where $H(P, x)$ returns:

- true if the program P halts on the input x ,
- false if the program P does not halt on the input x .

Remark: Let us say that subroutine $H(P, x)$ returns false if P is not a syntactically correct program.

Halting Problem

Using the subroutine H we can construct a program D that performs the following steps:

- It reads its input into a variable x of type `String`.
- It calls the subroutine $H(x, x)$.
- If subroutine H returns `true`, program D jumps into an infinite loop

loop: goto loop

In case that H returns `false`, program D halts.

What does the program D do if it gets its own code as an input?

Halting Problem

If D gets its own code as an input, it either halts or not.

- If D halts then $H(D, D)$ returns **true** and D jumps into the infinite loop. A contradiction!
- If D does not halt then $H(D, D)$ returns **false** and D halts. A contradiction!

In both case we obtain a contradiction and there is no other possibility. So the assumption that H solves the Halting problem must be wrong.

Reduction between Problems

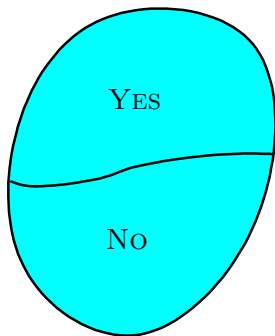
If we have already proved a (decision) problem to be undecidable, we can prove undecidability of other problems by reductions.

Problem P_1 can be **reduced** to problem P_2 if there is an algorithm Alg such that:

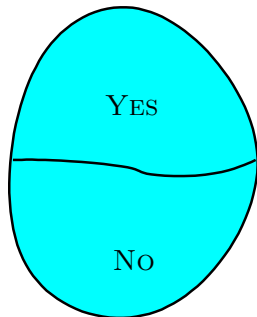
- It can get an arbitrary instance of problem P_1 as an input.
- For an instance of a problem P_1 obtained as an input (let us denote it as w) it produces an instance of a problem P_2 as an output.
- It holds i.e., the answer for the input w of problem P_1 is YES iff the answer for the input $Alg(w)$ of problem P_2 is YES.

Reductions between Problems

Inputs of problem P_1



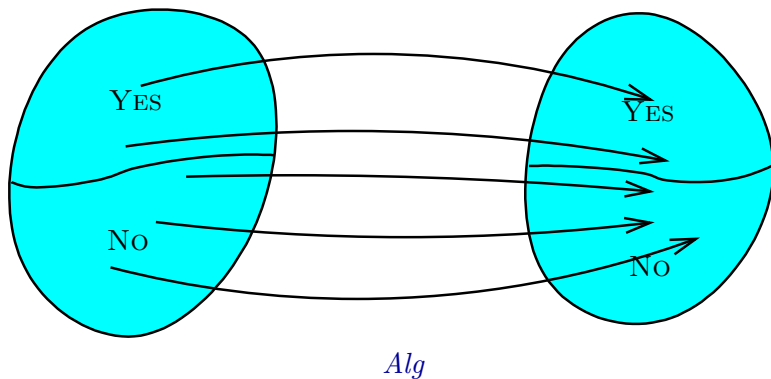
Inputs of problem P_2



Reductions between Problems

Inputs of problem P_1

Inputs of problem P_2



Reductions between Problems

Let us say there is some reduction Alg from problem P_1 to problem P_2 .

If problem P_2 is decidable then problem P_1 is also decidable.

Solution of problem P_1 for an input x :

- Call Alg with x as an input, it returns a value $Alg(x)$.
- Call the algorithm solving problem P_2 with input $Alg(x)$.
- Write the returned value to the output as the result.

It is obvious that if P_1 is undecidable then P_2 cannot be decidable.

Other Undecidable Problems

By reductions from the Halting problem we can show undecidability of many other problems dealing with a behaviour of programs:

- Is for some input the output of a given program **YES**?
- Does a given program halt for an arbitrary input?
- Do two given programs produce the same outputs for the same inputs?
- ...

For the use in proofs and in reductions between problems, it is convenient to have the language $\mathcal{L}$ and the machine running programs in this language as simple as possible:

- the number of kinds of instructions as small as possible
- instructions as primitive as possible
- the datatypes, with which the algorithm works, as simple as possible
- it is irrelevant how difficult is to write programs in the given language (it can be extremely user-unfriendly)

On the other hand, such language (resp. machine) must be general enough so that any program written in an arbitrary programming language can be compiled to it.

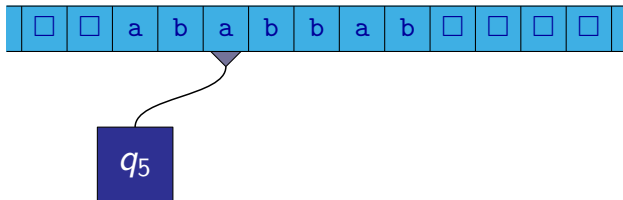
Such languages (resp. machines), which are general enough, so that programs written in any other programming language can be translated to them, are called **Turing complete**.

Examples of such Turing complete **models of computation** (languages or machines) often used in proofs:

- Turing machine (Alan Turing)
- Lambda calculus (Alonzo Church)
- Minsky machine (Marvin Minsky)
- ...

Turing machine:

- Let us extend a deterministic finite automaton in the following way:
 - the reading head can move in both directions
 - it is possible to write symbols on the tape
 - the tape is extended into infinity



Church-Turing thesis

Every algorithm can be implemented as a Turing machine.

It is not a theorem that can be proved in a mathematical sense – it is not formally defined what an algorithm is.

The thesis was formulated in 1930s independently by Alan Turing and Alonzo Church.

Halting Problem

For purposes of proofs, the following version of Halting problem is often used:

Halting problem

Input: A description of a Turing machine M and a word w .

Question: Does the computation of the machine M on the word w halt after some finite number of steps?

Other Undecidable Problems

We have already seen the following example of an undecidable problem:

Problem

Input: Context-free grammars G_1 and G_2 .

Question: Is $L(G_1) = L(G_2)$?

respectively

Problem

Input: A context-free grammar generating a language over an alphabet Σ .

Question: Is $L(G) = \Sigma^*$?

Other Undecidable Problems

An input is a set of types of tiles, such as:

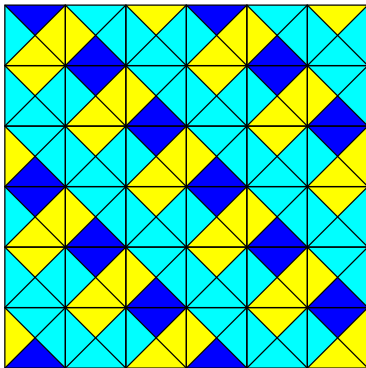


The question is whether it is possible to cover every finite area of an arbitrary size using the given types of tiles in such a way that the colors of neighboring tiles agree.

Remark: We can assume that we have an infinite number of tiles of all types.

The tiles cannot be rotated.

Other Undecidable Problems



Other Undecidable Problems

An input is a set of types of cards, such as:

abb	a	bab	baba	aba
bbab	aa	ab	aa	a

The question is whether it is possible to construct from the given types of cards a non-empty finite sequence such that the concatenations of the words in the upper row and in the lower row are the same. Every type of a card can be used repeatedly.

a	abb	abb	baba	abb	aba
aa	bbab	bbab	aa	bbab	a

In the upper and in the lower row we obtained the word
[aabbabbbabaabbaba](#).

Other Undecidable Problems

Undecidability of several other problems dealing with context-free grammars can be proved by reductions from the previous problem:

Problem

Input: Context-free grammars G_1 and G_2 .

Question: Is $L(G_1) \cap L(G_2) = \emptyset$?

Problem

Input: A context-free grammar G .

Question: Is G ambiguous?

Other Undecidable Problems

Problem

Input: A closed formula of the first order predicate logic where the only predicate symbols are $=$ and $<$, the only function symbols are $+$ and $*$, and the only constant symbols are 0 and 1 .

Question: Is the given formula true in the domain of natural numbers (using the natural interpretation of all function and predicate symbols)?

An example of an input:

$$\forall x \exists y \forall z ((x * y = z) \wedge (y + 1 = x))$$

Remark: There is a close connection with Gödel's incompleteness theorem.

It is interesting that an analogous problem, where real numbers are considered instead of natural numbers, is decidable (but the algorithm for it and the proof of its correctness are quite nontrivial).

Also when we consider natural numbers or integers and the same formulas as in the previous case but with the restriction that it is not allowed to use the multiplication function symbol $*$, the problem is algorithmically decidable.

Other Undecidable Problems

If the function symbol $*$ can be used then even the very restricted case is undecidable:

Hilbert's tenth problem

Input: A polynomial $f(x_1, x_2, \dots, x_n)$ constructed from variables $x_1, x_2, \dots, x_n$ and integer constants.

Question: Are there some natural numbers $x_1, x_2, \dots, x_n$ such that $f(x_1, x_2, \dots, x_n) = 0$?

An example of an input: $5x^2y - 8yz + 3z^2 - 15$

I.e., the question is whether

$$\exists x \exists y \exists z (5 * x * x * y + (-8) * y * z + 3 * z * z + (-15) = 0)$$

holds in the domain of natural numbers.

Other Undecidable Problems

Also the following problem is algorithmically undecidable:

Problem

Input: A closed formula φ of the first-order predicate logic.

Question: Is $\models \varphi$?

Remark: Notation $\models \varphi$ denotes that formula φ is logically valid, i.e., it is true in all interpretations.

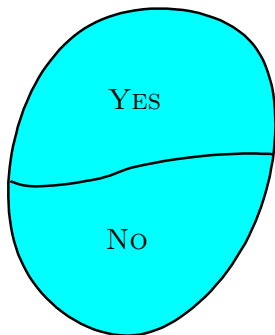
NP-Complete Problems

Polynomial Reductions between Problems

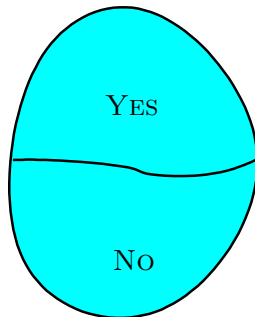
There is a **polynomial reduction** of problem P_1 to problem P_2 if there exists an algorithm Alg with a polynomial time complexity that reduces problem P_1 to problem P_2 .

Polynomial Reductions between Problems

Inputs of problem P_1



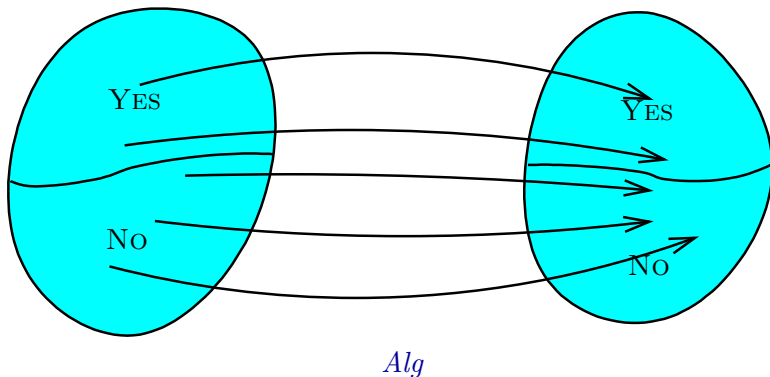
Inputs of problem P_2



Polynomial Reductions between Problems

Inputs of problem P_1

Inputs of problem P_2



Polynomial Reductions between Problems

Let us say that problem A can be reduced in polynomial time to problem B , i.e., there is a (polynomial) algorithm P realizing this reduction.

If problem B is in the class PTIME then problem A is also in the class PTIME .

A solution of problem A for an input x :

- Call P with input x and obtain a returned value $P(x)$.
- Call a polynomial time algorithm solving problem B with the input $P(x)$.

Write the returned value as the answer for A .

That means:

If A is not in PTIME then also B can not be in PTIME .

Polynomial Reductions between Problems

There is a big class of algorithmic problems called **NP-complete** problems such that:

- these problems can be solved by exponential time algorithms
- no polynomial time algorithm is known for any of these problems
- on the other hand, for any of these problems it is not proved that there cannot exist a polynomial time algorithm for the given problem
- every NP-complete problem can be polynomially reduced to any other NP-complete problem

Remark: This is not a definition of NP-complete problems. The precise definition will be described later.

Problem SAT

A typical example of an NP-complete problem is the SAT problem:

SAT (boolean satisfiability problem)

Input: Boolean formula φ .

Question: Is φ satisfiable?

Example:

Formula $\varphi_1 = x_1 \wedge (\neg x_2 \vee x_3)$ is satisfiable:

e.g., for valuation v where $v(x_1) = 1$, $v(x_2) = 0$, $v(x_3) = 1$, the formula φ_1 is true.

Formula $\varphi_2 = (x_1 \wedge \neg x_1) \vee (\neg x_2 \wedge x_3 \wedge x_2)$ is not satisfiable:
it is false for every valuation v .

Problem 3-SAT

3-SAT is a variant of the SAT problem where the possible inputs are restricted to formulas of a certain special form:

3-SAT

Input: Formula φ is a conjunctive normal form where every clause contains exactly 3 literals.

Question: Is φ satisfiable?

Problem 3-SAT

Recalling some notions:

- A **literal** is a formula of the form x or $\neg x$ where x is an atomic proposition.
- A **clause** is a disjunction of literals.

Examples: $x_1 \vee \neg x_2$ $\neg x_5 \vee x_8 \vee \neg x_{15} \vee \neg x_{23}$ x_6

- A formula is in a **conjunctive normal form (CNF)** if it is a conjunction of clauses.

Example: $(x_1 \vee \neg x_2) \wedge (\neg x_5 \vee x_8 \vee \neg x_{15} \vee \neg x_{23}) \wedge x_6$

So in the 3-SAT problem we require that a formula φ is in a CNF and moreover that every clause of φ contains exactly three literals.

Example:

$(x_1 \vee \neg x_2 \vee x_4) \wedge (\neg x_1 \vee x_3 \vee x_3) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4) \wedge (x_2 \vee \neg x_3 \vee x_4)$

Problem 3-SAT

The following formula is satisfiable:

$$(x_1 \vee \neg x_2 \vee x_4) \wedge (\neg x_1 \vee x_3 \vee x_3) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4) \wedge (x_2 \vee \neg x_3 \vee x_4)$$

It is true for example for valuation v where

$$v(x_1) = 0$$

$$v(x_2) = 1$$

$$v(x_3) = 0$$

$$v(x_4) = 1$$

On the other hand, the following formula is not satisfiable:

$$(x_1 \vee x_1 \vee x_1) \wedge (\neg x_1 \vee \neg x_1 \vee \neg x_1)$$

Polynomial Reductions between Problems

As an example, a polynomial time reduction from the 3-SAT problem to the independent set problem (IS) will be described.

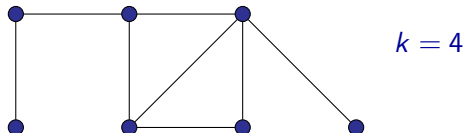
Remark: Both 3-SAT and IS are examples of NP-complete problems.

Independent Set (IS) Problem

Independent set (IS) problem

Input: An undirected graph G , a number k .

Question: Is there an independent set of size k in the graph G ?



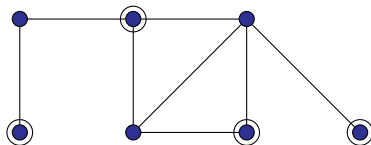
Remark: An **independent set** in a graph is a subset of nodes of the graph such that no pair of nodes from this set is connected by an edge.

Independent Set (IS) Problem

Independent set (IS) problem

Input: An undirected graph G , a number k .

Question: Is there an independent set of size k in the graph G ?

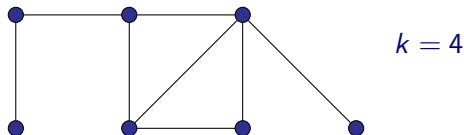


$k = 4$

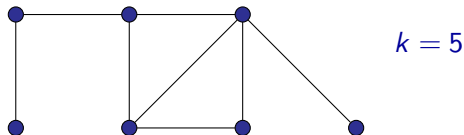
Remark: An **independent set** in a graph is a subset of nodes of the graph such that no pair of nodes from this set is connected by an edge.

Independent Set (IS) Problem

An example of an instance where the answer is **YES**:



An example of an instance where the answer is **No**:



A Reduction from 3-SAT to IS

We describe a (polynomial-time) algorithm with the following properties:

- **Input:** An arbitrary instance of 3-SAT, i.e., a formula φ in a conjunctive normal form where every clause contains exactly three literals.
- **Output:** An instance of IS, i.e., an undirected graph G and a number k .
- Moreover, the following will be ensured for an arbitrary input (i.e., for an arbitrary formula φ in the above mentioned form):

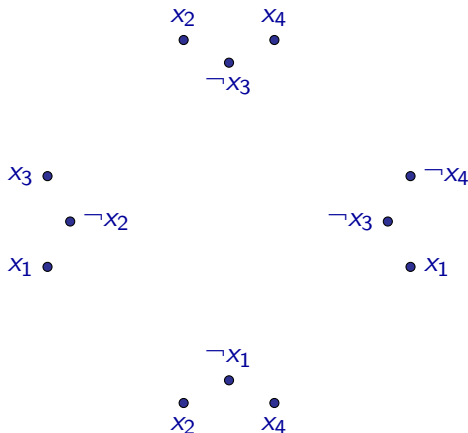
There will be an independent set of size k in graph G iff formula φ will be satisfiable.

A Reduction from 3-SAT to IS

$$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee x_4)$$

A Reduction from 3-SAT to IS

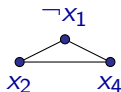
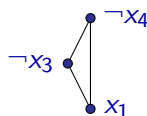
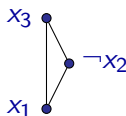
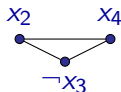
$$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee x_4)$$



For each occurrence of a literal we add a node to the graph.

A Reduction from 3-SAT to IS

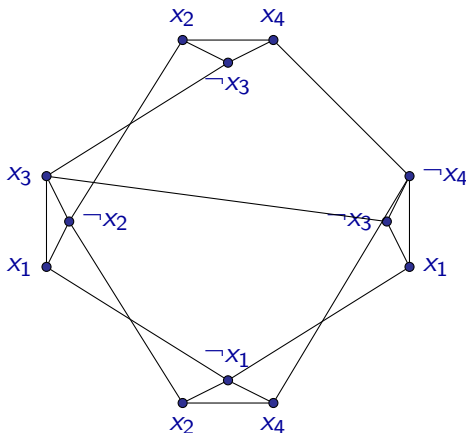
$$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee x_4)$$



We connect with edges the nodes corresponding to occurrences of literals belonging to the same clause.

A Reduction from 3-SAT to IS

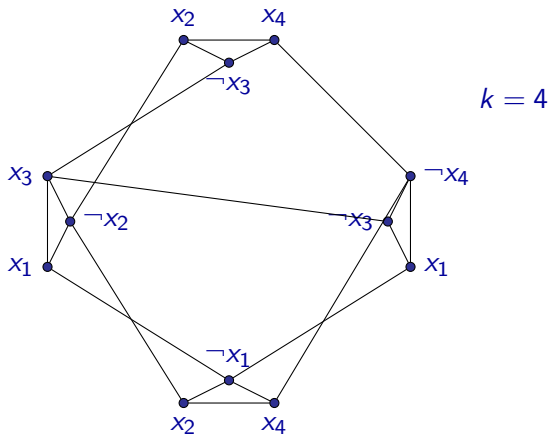
$$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee x_4)$$



For each pair of nodes corresponding to literals x_i and $\neg x_i$ we add an edge between them.

A Reduction from 3-SAT to IS

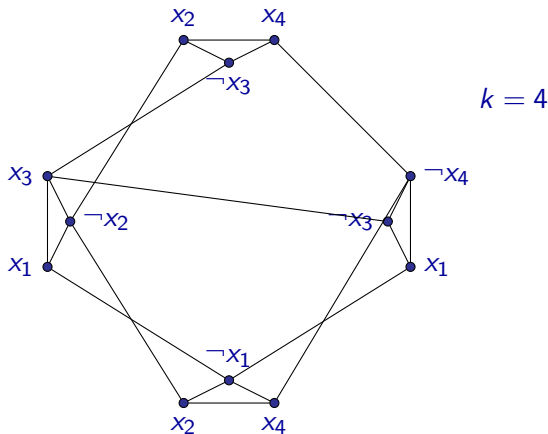
$$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee x_4)$$



We put k to be equal to the number of clauses.

A Reduction from 3-SAT to IS

$$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee x_4)$$



The constructed graph and number k are the output of the algorithm.

A Reduction from 3-SAT to IS

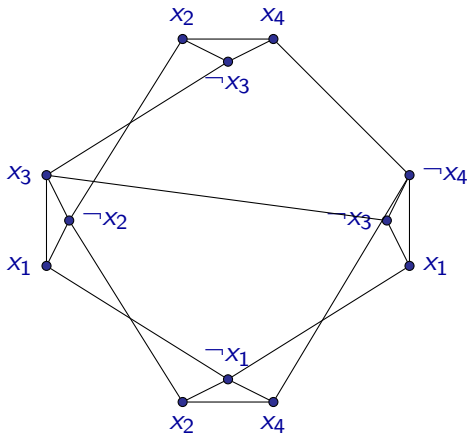
$$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee x_4)$$

$$v(x_1) = 1$$

$$v(x_2) = 1$$

$$v(x_3) = 0$$

$$v(x_4) = 1$$

$$k = 4$$


If the formula φ is satisfiable then there is a valuation v where every clause contains at least one literal with value 1.

A Reduction from 3-SAT to IS

$$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee x_4)$$

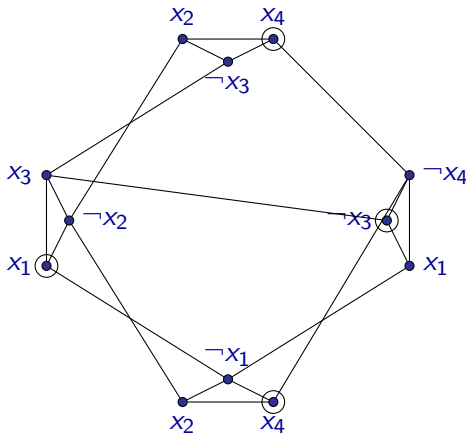
$$v(x_1) = 1$$

$$v(x_2) = 1$$

$$v(x_3) = 0$$

$$v(x_4) = 1$$

$$k = 4$$



We select one literal that has a value **1** in the valuation v , and we put the corresponding node into the independent set.

A Reduction from 3-SAT to IS

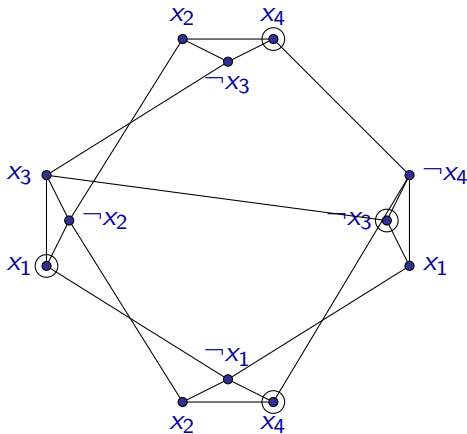
$$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee x_4)$$

$$v(x_1) = 1$$

$$v(x_2) = 1$$

$$v(x_3) = 0$$

$$v(x_4) = 1$$

$$k = 4$$


We can easily verify that the selected nodes form an independent set.

A Reduction from 3-SAT to IS

The selected nodes form an independent set because:

- One node has been selected from each triple of nodes corresponding to one clause.
- Nodes denoted x_i and $\neg x_i$ could not be selected together. (Exactly of them has the value 1 in the given valuation v .)

A Reduction from 3-SAT to IS

On the other hand, if there is an independent set of size k in graph G , then it surely has the following properties:

- At most one node is selected from each triple of nodes corresponding to one clause.

But because there are k clauses and k nodes are selected, exactly one node must be selected from each triple.

- Nodes denoted x_i and $\neg x_i$ cannot be selected together.

We can choose a valuation according to the selected nodes, since it follows from the previous discussion that it must exist.

(Arbitrary values can be assigned to the remaining variables.)

For the given valuation, the formula φ has surely the value 1 , since in each clause there is at least one literal with value 1 .

A Reduction from 3-SAT to IS

It is obvious that the running time of the described algorithm polynomial:
Graph G and number k can be constructed for a formula φ in time $O(n^2)$,
where n is the size of formula φ .

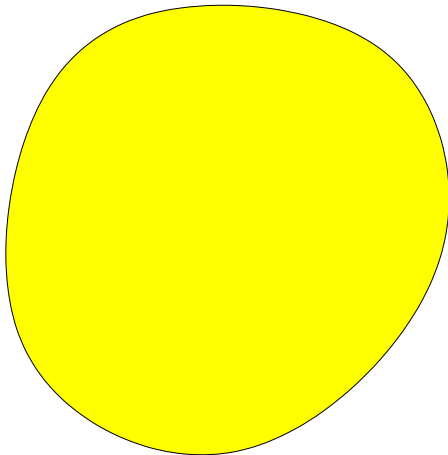
We have also seen that there is an independent set of size k in the
constructed graph G iff the formula φ is satisfiable.

The described algorithm shows that 3-SAT can be reduced in polynomial
time to IS.

- **PTIME** — the class of all algorithmic problems that can solve by a (deterministic) algorithm in polynomial time
- **NPTIME** — the class of algorithmic problems that can be solved by a **nondeterministic** algorithm in polynomial time

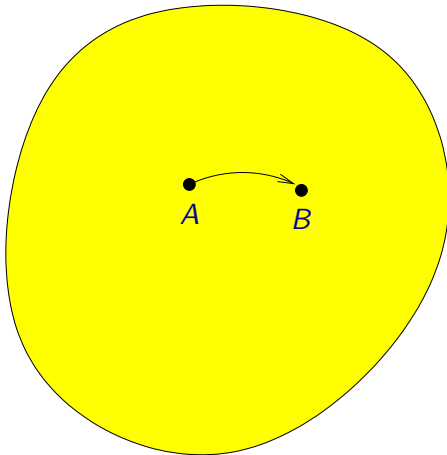
NP-Complete Problems

Let us consider a set of all decision problems.



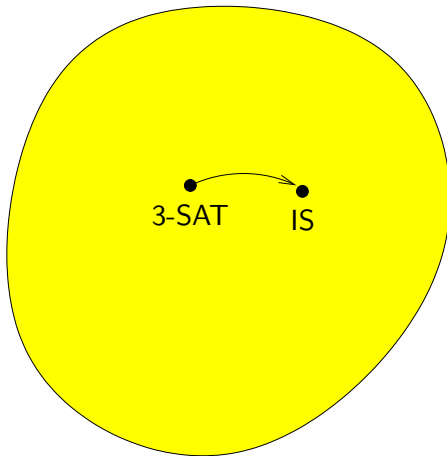
NP-Complete Problems

By an arrow we denote that a problem A can be reduced in polynomial time to a problem B .



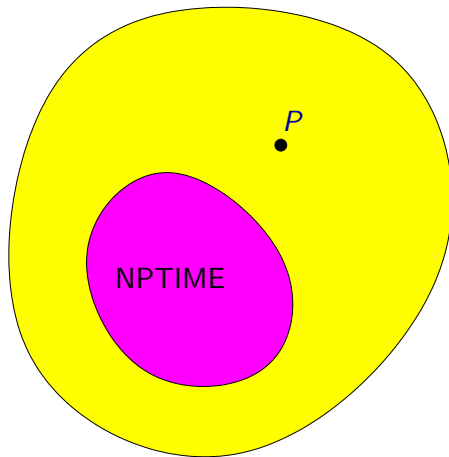
NP-Complete Problems

For example 3-SAT can be reduced in polynomial time to IS.



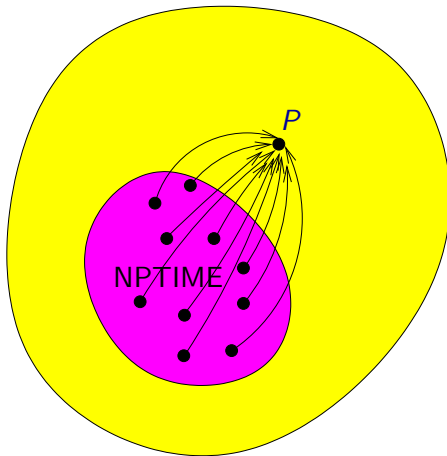
NP-Complete Problems

Let us consider now the class **NPTIME** and a problem P .



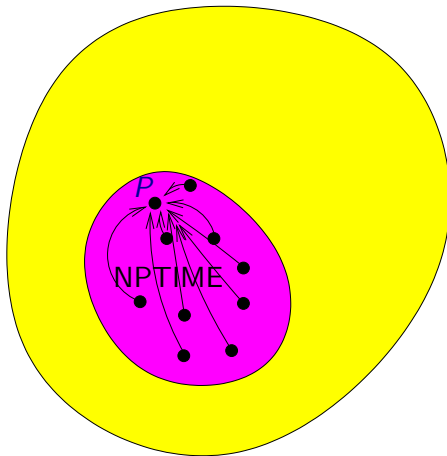
NP-Complete Problems

A problem P is **NP-hard** if every problem from **NPTIME** can be reduced in polynomial time to P .



NP-Complete Problems

A problem P is **NP-complete** if it is NP-hard and it belongs to the class NPTIME.



NP-Complete Problems

If we have found a polynomial time algorithm for some NP-hard problem P , then we would have polynomial time algorithms for all problems P' from NPTIME :

- At first we would apply an algorithm for the reduction from P' to P on an input of a problem P' .
- Then we would use a polynomial algorithm for P on the constructed instance of P and returned its result as the answer for the original instance of P' .

Is such case, $\text{PTIME} = \text{NPTIME}$ would hold, since for every problem from NPTIME there would be a polynomial-time (deterministic) algorithm.

On the other hand, if there is at least one problem from **NPTIME** for which a polynomial-time algorithm does not exist, then it means that for none of **NP**-hard problems there is a polynomial-time algorithm.

It is an open question whether the first or the second possibility holds.

NP-Complete Problems

It is not difficult to see that:

If a problem A can be reduced in a polynomial time to a problem B and problem B can be reduced in a polynomial time to a problem C , then problem A can be reduced in a polynomial time to problem C .

So if we know about some problem P that it is NP-hard and that P can be reduced in a polynomial time to a problem P' , then we know that the problem P' is also NP-hard.

NP-Complete Problems

Theorem

Problem SAT is NP-complete.

It can be shown that SAT can be reduced in a polynomial time to 3-SAT and we have seen that 3-SAT can be reduced in a polynomial time to IS.

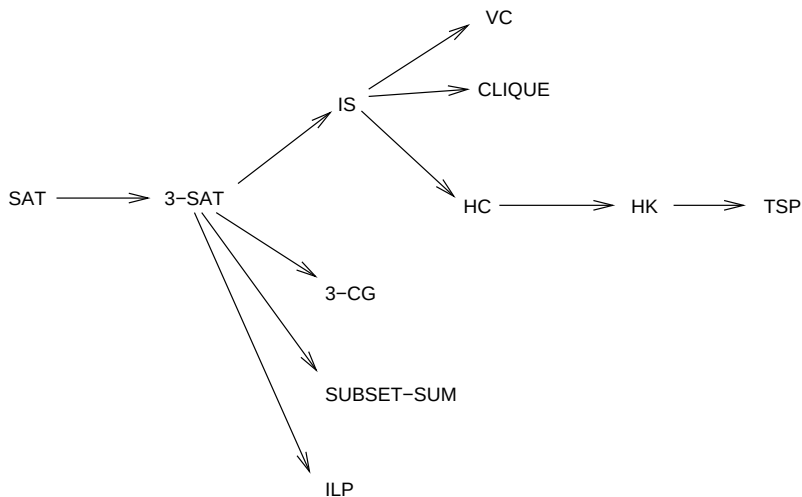
This means that problems 3-SAT and IS are NP-hard.

It is not difficult to show that 3-SAT and IS belong to the class NPTIME.

Problems 3-SAT and IS are NP-complete.

NP-Complete Problems

By a polynomial reductions from problems that are already known to be NP-complete, NP-completeness of many other problems can be shown:



Examples of Some NP-Complete Problems

The following previously mentioned problems are NP-complete:

- SAT (boolean satisfiability problem)
- 3-SAT
- IS — independent set problem

On the following slides, examples of some other NP-complete problems are described:

- CG — graph coloring (remark: it is NP-complete even in the special case where we have 3 colors)
- VC — vertex cover
- CLIQUE — clique problem
- HC — Hamiltonian cycle
- HK — Hamiltonian circuit
- TSP — traveling salesman problem
- SUBSET-SUM
- ILP — integer linear programming

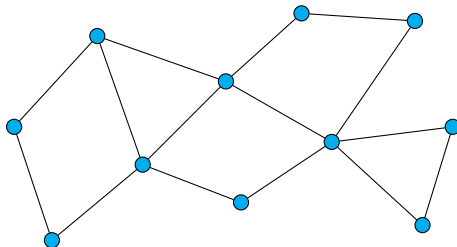
Graph Coloring

Graph coloring

Input: An undirected graph G , a natural number k .

Question: Is it possible to color nodes of the graph G using k colors in such a way that there is no pair of nodes where both nodes are colored with the same color and connected with an edge?

Example: $k = 3$



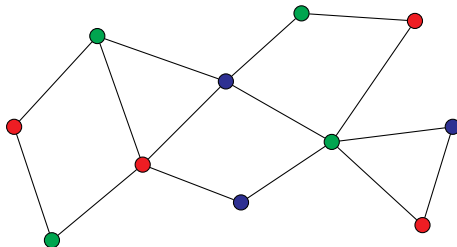
Graph Coloring

Graph coloring

Input: An undirected graph G , a natural number k .

Question: Is it possible to color nodes of the graph G using k colors in such a way that there is no pair of nodes where both nodes are colored with the same color and connected with an edge?

Example: $k = 3$



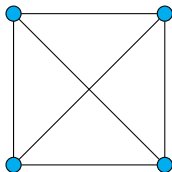
Graph Coloring

Graph coloring

Input: An undirected graph G , a natural number k .

Question: Is it possible to color nodes of the graph G using k colors in such a way that there is no pair of nodes where both nodes are colored with the same color and connected with an edge?

Example: $k = 3$



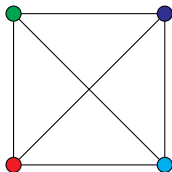
Graph Coloring

Graph coloring

Input: An undirected graph G , a natural number k .

Question: Is it possible to color nodes of the graph G using k colors in such a way that there is no pair of nodes where both nodes are colored with the same color and connected with an edge?

Example: $k = 3$



Answer: No

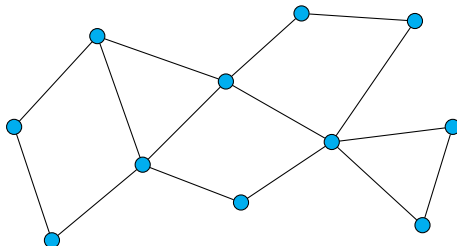
VC – Vertex Cover

VC – vertex cover

Input: An undirected graph G and a natural number k .

Question: Is there some subset of nodes of G of size k such that every edge has at least one of its nodes in this subset?

Example: $k = 6$



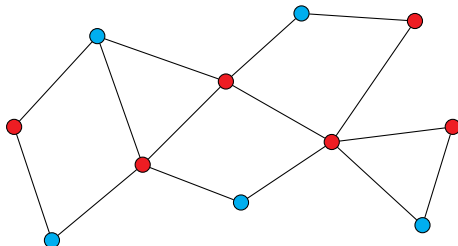
VC – Vertex Cover

VC – vertex cover

Input: An undirected graph G and a natural number k .

Question: Is there some subset of nodes of G of size k such that every edge has at least one of its nodes in this subset?

Example: $k = 6$



Answer: YES

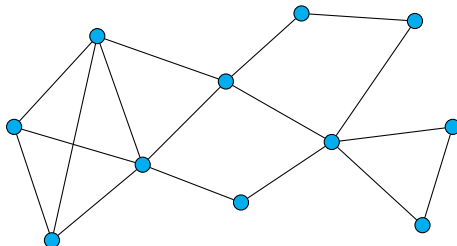
CLIQUE

CLIQUE

Input: An undirected graph G and a natural number k .

Question: Is there some subset of nodes of G of size k such that every two nodes from this subset are connected by an edge?

Example: $k = 4$



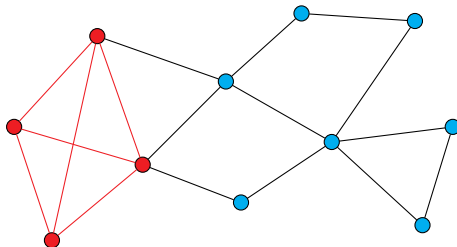
CLIQUE

CLIQUE

Input: An undirected graph G and a natural number k .

Question: Is there some subset of nodes of G of size k such that every two nodes from this subset are connected by an edge?

Example: $k = 4$



Answer: YES

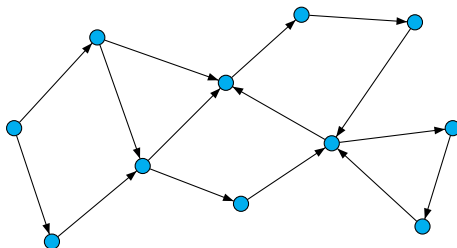
Hamiltonian Cycle

HC – Hamiltonian cycle

Input: A directed graph G .

Question: Is there a Hamiltonian cycle in G (i.e., a directed cycle going through each node exactly once)?

Example:



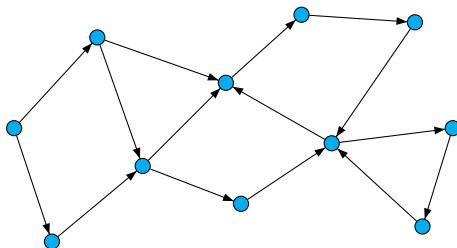
Hamiltonian Cycle

HC – Hamiltonian cycle

Input: A directed graph G .

Question: Is there a Hamiltonian cycle in G (i.e., a directed cycle going through each node exactly once)?

Example:



Answer: No

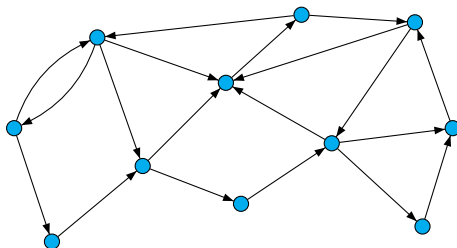
Hamiltonian Cycle

HC – Hamiltonian cycle

Input: A directed graph G .

Question: Is there a Hamiltonian cycle in G (i.e., a directed cycle going through each node exactly once)?

Example:



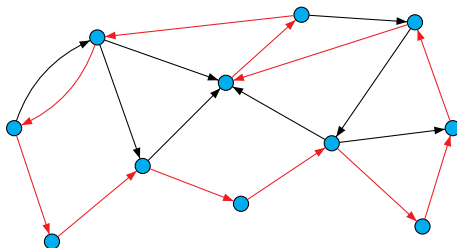
Hamiltonian Cycle

HC – Hamiltonian cycle

Input: A directed graph G .

Question: Is there a Hamiltonian cycle in G (i.e., a directed cycle going through each node exactly once)?

Example:



Answer: YES

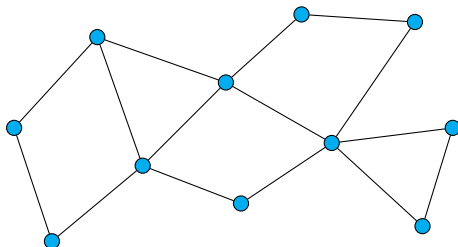
Hamiltonian Circuit

HK – Hamiltonian circuit

Input: An undirected graph G .

Question: Is there a Hamiltonian circuit in G (i.e., an undirected cycle going through each node exactly once)?

Example:



Answer: No

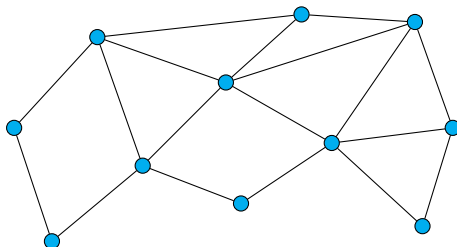
Hamiltonian Circuit

HK – Hamiltonian circuit

Input: An undirected graph G .

Question: Is there a Hamiltonian circuit in G (i.e., an undirected cycle going through each node exactly once)?

Example:



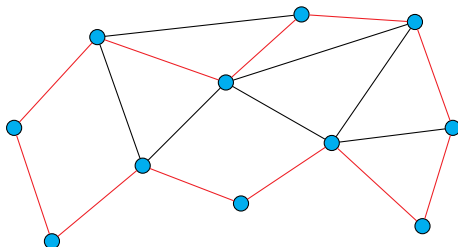
Hamiltonian Circuit

HK – Hamiltonian circuit

Input: An undirected graph G .

Question: Is there a Hamiltonian circuit in G (i.e., an undirected cycle going through each node exactly once)?

Example:



Answer: YES

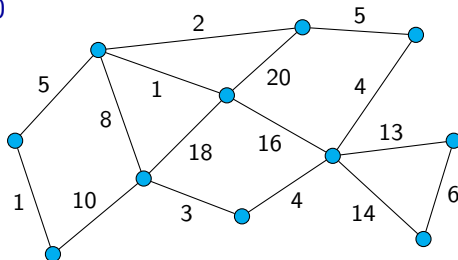
Traveling Salesman Problem

TSP - traveling salesman problem

Input: An undirected graph G with edges labelled with natural numbers and a number k .

Question: Is there a closed tour going through all nodes of the graph G such that the sum of labels of edges on this tour is at most k ?

Example: $k = 70$



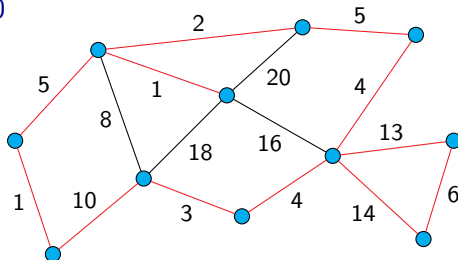
Traveling Salesman Problem

TSP - traveling salesman problem

Input: An undirected graph G with edges labelled with natural numbers and a number k .

Question: Is there a closed tour going through all nodes of the graph G such that the sum of labels of edges on this tour is at most k ?

Example: $k = 70$



Answer: YES, since there is a tour with the sum 69.

SUBSET-SUM

Problem SUBSET-SUM

Input: A sequence $a_1, a_2, \dots, a_n$ of natural numbers and a natural number s .

Question: Is there a set $I \subseteq \{1, 2, \dots, n\}$ such that $\sum_{i \in I} a_i = s$?

In other words, the question is whether it is possible to select a subset with sum s of a given (multi)set of numbers.

Example: For the input consisting of numbers $3, 5, 2, 3, 7$ and number $s = 15$ the answer is **YES**, since $3 + 5 + 7 = 15$.

For the input consisting of numbers $3, 5, 2, 3, 7$ and number $s = 16$ the answer is **No**, since no subset of these numbers has sum 16 .

Remark:

The order of numbers $a_1, a_2, \dots, a_n$ in an input is not important.

Note that this is not exactly the same as if we have formulated the problem so that the input is a set $\{a_1, a_2, \dots, a_n\}$ and a number s — numbers cannot occur multiple times in a set but they can in a sequence.

Problem SUBSET-SUM is a special case of a **knapsack problem**:

Knapsack problem

Input: Sequence of pairs of natural numbers $(a_1, b_1), (a_2, b_2), \dots, (a_n, b_n)$ and two natural numbers s and t .

Question: Is there a set $I \subseteq \{1, 2, \dots, n\}$ such that $\sum_{i \in I} a_i \leq s$ and $\sum_{i \in I} b_i \geq t$?

SUBSET-SUM

Informally, the knapsack problem can be formulated as follows:

We have n objects, where the i -th object weights a_i grams and its price is b_i dollars.

The question is whether there is a subset of these objects with total weight at most s grams (s is the capacity of the knapsack) and with total price at least t dollars.

Remark:

Here we have formulated this problem as a decision problem.

This problem is usually formulated as an optimization problem where the aim is to find such a set $I \subseteq \{1, 2, \dots, n\}$, where the value $\sum_{i \in I} b_i$ is maximal and where the condition $\sum_{i \in I} a_i \leq s$ is satisfied, i.e., where the capacity of the knapsack is not exceeded.

That SUBSET-SUM is a special case of the Knapsack problem can be seen from the following simple construction:

Let us say that $a_1, a_2, \dots, a_n, s_1$ is an instance of SUBSET-SUM.

It is obvious that for the instance of the knapsack problem where we have the sequence $(a_1, a_1), (a_2, a_2), \dots, (a_n, a_n), s = s_1$ and $t = s_1$, the answer is the same as for the original instance of SUBSET-SUM.

SUBSET-SUM

If we want to study the complexity of problems such as SUBSET-SUM or the knapsack problem, we must clarify what we consider as the size of an instance.

Probably the most natural it is to define the size of an instance as the total number of bits needed for its representation.

We must specify how natural numbers in the input are represented – if in binary (resp. in some other numeral system with a base at least 2 (e.g., decimal or hexadecimal) or in unary.

- If we consider the total number of bits when numbers are written in **binary** as the size of an input, no polynomial time algorithm is known for SUBSET-SUM.
- If we consider the total number of bits when numbers are written in **unary** as the size of an input, SUBSET-SUM can be solved by an algorithm whose time complexity is polynomial.

Problem ILP (integer linear programming)

Input: An integer matrix A and an integer vector b .

Question: Is there an integer vector x such that $Ax \leq b$?

An example of an instance of the problem:

$$A = \begin{pmatrix} 3 & -2 & 5 \\ 1 & 0 & 1 \\ 2 & 1 & 0 \end{pmatrix} \quad b = \begin{pmatrix} 8 \\ -3 \\ 5 \end{pmatrix}$$

So the question is if the following system of inequations has some integer solution:

$$\begin{aligned} 3x_1 - 2x_2 + 5x_3 &\leq 8 \\ x_1 + x_3 &\leq -3 \\ 2x_1 + x_2 &\leq 5 \end{aligned}$$

ILP – Integer Linear Programming

One of solutions of the system

$$\begin{aligned}3x_1 - 2x_2 + 5x_3 &\leq 8 \\x_1 + x_3 &\leq -3 \\2x_1 + x_2 &\leq 5\end{aligned}$$

is for example $x_1 = -4$, $x_2 = 1$, $x_3 = 1$, i.e.,

$$x = \begin{pmatrix} -4 \\ 1 \\ 1 \end{pmatrix}$$

because

$$\begin{aligned}3 \cdot (-4) - 2 \cdot 1 + 5 \cdot 1 &= -9 \leq 8 \\-4 + 1 &= -3 \leq -3 \\2 \cdot (-4) + 1 &= -7 \leq 5\end{aligned}$$

So the answer for this instance is **YES**.

Remark: A similar problem where the question for a given system of linear inequations is whether it has a solution in the set of **real** numbers, can be solved in a polynomial time.