

# Notes for Logic (COMP0009)

Raphael Li

Sep 2025

---

## Contents

<b>1</b>	<b>Revision: The syntax and semantics of propositional and first-order logic</b>	<b>2</b>
1.1	Propositional logic . . . . .	2
1.1.1	Syntax . . . . .	2
1.1.2	Semantics . . . . .	2
1.1.3	Truth tables . . . . .	3
1.1.4	Parse trees . . . . .	3
1.1.5	Disjunctive normal form (DNF) . . . . .	4
1.1.6	Conjunctive normal form (CNF) . . . . .	4
1.2	First-order logic . . . . .	5
1.2.1	Syntax . . . . .	5
1.2.2	Semantics . . . . .	5
1.2.3	Example: Arithmetic in the set of natural numbers . . . . .	7
1.2.4	First-order structures and directed graphs . . . . .	8
<b>2</b>	<b>Axiomatic Proofs for Propositional Logic</b>	<b>9</b>
2.1	Hilbert-style proof system . . . . .	9
2.2	Proofs with assumptions and the principle of explosion . . . . .	10
2.3	Soundness, completeness and termination . . . . .	11
<b>3</b>	<b>Propositional tableau</b>	<b>12</b>
3.1	Constructing a tableau . . . . .	12
3.2	Example of constructing a tableau and converting to DNF . . . . .	13

# 1 Revision: The syntax and semantics of propositional and first-order logic

Formally, a *logic* consists of three components:

Component	Describes...
Syntax	The language and grammar for writing formulas
Semantics	How formulas are interpreted
Inference system (or proof system)	A syntactic device for proving true statements

Table 1: The three key components of a logic.

This module concerns algorithms that automatically parse and determine the validity of a formula.

## 1.1 Propositional logic

### 1.1.1 Syntax

Formulas are constructed by applying negation, conjunction and disjunction to propositions.

$$\begin{aligned} \text{proposition} &:= p \mid q \mid r \mid \dots \\ \text{formula} &:= \text{proposition} \mid \neg \text{formula} \mid (\text{formula} \circ \text{formula}) \end{aligned} \quad (\text{where } \circ \text{ is } \wedge, \vee \text{ or } \rightarrow)$$

A proposition or its negation is called a *literal*<sup>1</sup>.

For any formula that isn't a proposition, the *main connective* is the one with the largest scope. In other words, it is not in the scope of any other connective.

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

This is the connective with which evaluation begins. This is especially important when building parsers for algorithmically evaluating formulas.

Note that parsers working according to the above definition will recognise  $(p \wedge q)$ , but not  $p \wedge q$ , as a formula. Regardless, throughout this document we will use a looser definition where brackets may be omitted in unambiguous cases.

### 1.1.2 Semantics

A valuation is a function  $v$  that maps each proposition to a truth value in  $\{\top, \perp\}$ .

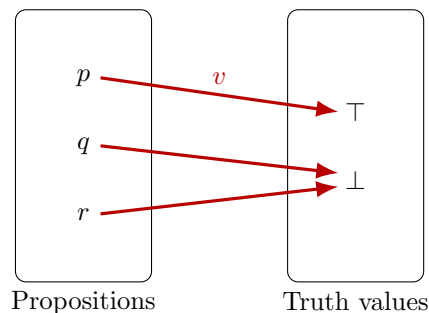


Figure 1: A valuation maps propositions to truth values.

<sup>1</sup>For example,  $p$  and  $\neg p$  are both literals, but  $\neg\neg q$  is not.

A valuation  $v$  can be extended to a unique *truth function* defined on all possible formulas. A truth function  $v'$  must satisfy

$$\begin{aligned} v'(\neg\phi) = \top &\iff v'(\phi) = \perp \\ v'(\phi \vee \psi) = \top &\iff v'(\phi) = \top \text{ or } v'(\psi) = \top \\ v'(\phi \wedge \psi) = \top &\iff v'(\phi) = \top \text{ and } v'(\psi) = \top \\ v'(\phi \rightarrow \psi) = \top &\iff v'(\phi) = \perp \text{ or } v'(\psi) = \top \\ v'(\phi \leftrightarrow \psi) = \top &\iff v'(\phi) = v'(\psi) \end{aligned}$$

for all formulas  $\phi$  and  $\psi$ . From now on we use  $v$  to denote the more general truth function.

The result of applying a valuation  $v$  to a formula  $\phi$  depends only on the propositional letters that occur in  $\phi$ .

A formula  $\phi$  is *valid* if  $v(\phi) = \top$  for all valuations  $v$ , which we denote as  $\models \phi$ . A formula  $\phi$  is *satisfiable* if  $v(\phi) = \top$  for at least one valuation  $v$ . All valid formulas are satisfiable, but *not* vice versa.

Two formulas  $\phi$  and  $\psi$  are *logically equivalent*, written as  $\phi \equiv \psi$ , if and only if for every valuation  $v$  we have  $v(\phi) = v(\psi)$ .

### 1.1.3 Truth tables

Consider the propositional formula  $((p \vee \neg q) \wedge \neg(q \wedge r))$ . We can check its validity and satisfiability by constructing its truth table.

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

Table 2: The truth table for the formula  $((p \vee \neg q) \wedge \neg(q \wedge r))$ .

In this case, the formula is satisfiable but not valid.

### 1.1.4 Parse trees

A parser interprets the semantics of a formula by breaking down its symbols into a *parse tree*, which shows the syntactic relation between symbols. For example, the formula  $((p \vee \neg q) \wedge \neg(q \wedge r))$  can be broken down into the following parse tree.

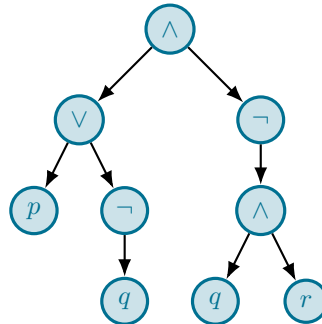


Figure 2: The parse tree for the formula  $((p \vee \neg q) \wedge \neg(q \wedge r))$ .

### 1.1.5 Disjunctive normal form (DNF)

A formula is said to be in *disjunctive normal form* (DNF) if it is a disjunction of one or more conjunctions of one or more literals.

$$\begin{aligned}\text{proposition} &:= p \mid q \mid r \mid \dots \\ \text{literal} &:= \text{proposition} \mid \neg \text{proposition} \\ \text{conjunctiveClause} &:= \text{literal} \mid \text{literal} \wedge \text{conjunctiveClause} \\ \text{DNF} &:= \text{conjunctiveClause} \mid \text{conjunctiveClause} \vee \text{DNF}\end{aligned}$$

Below is an example of a formula in DNF.

$$\underbrace{(p \wedge \neg q \wedge \neg r)}_{\text{conjunctive clause}} \vee \underbrace{(\neg p \wedge \neg q \wedge r)}_{\text{conjunctive clause}} \vee \underbrace{(q \wedge \neg r)}_{\text{conjunctive clause}}$$

Any propositional formula has a DNF equivalent. For instance, the formula  $(p \vee \neg q) \wedge \neg(q \wedge r)$  can be rewritten as follows.

$$\begin{aligned}& (p \vee \neg q) \wedge \neg(q \wedge r) \\ \iff & (p \vee \neg q) \wedge (\neg q \vee \neg r) && \text{(De Morgan's law, to remove outer negation)} \\ \iff & ((p \vee \neg q) \wedge \neg q) \vee ((p \vee \neg q) \wedge \neg r) && \text{(distributing conjunctions over disjunctions)} \\ \iff & (p \wedge \neg q) \vee (\neg q \wedge \neg q) \vee (p \wedge \neg r) \vee (\neg q \wedge \neg r) && \text{(distributing conjunctions over disjunctions)} \\ \iff & (p \wedge \neg q) \vee \neg q \vee (p \wedge \neg r) \vee (\neg q \wedge \neg r)\end{aligned}$$

Alternatively, this can also be achieved by referring to the truth table. From Table 2, we see that the formula can be written in DNF as

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

### 1.1.6 Conjunctive normal form (CNF)

A formula is said to be *conjunctive normal form* (CNF) if it is a conjunction of one or more disjunctions of one or more literals.

$$\begin{aligned}\text{disjunctiveClause} &:= \text{literal} \mid \text{literal} \vee \text{disjunctiveClause} \\ \text{CNF} &:= \text{disjunctiveClause} \mid \text{disjunctiveClause} \wedge \text{CNF}\end{aligned}$$

Below is a formula in CNF.

$$\underbrace{(p \vee \neg q \vee \neg r)}_{\text{conjunctive clause}} \wedge \underbrace{(\neg p \vee q \vee r)}_{\text{conjunctive clause}}$$

To find the CNF equivalent of a formula  $\phi$ , we first express its negation  $\neg\phi$  in DNF. Then, we negate it again to get  $\neg\neg\phi$ . Using De Morgan's law, the resultant formula will be in CNF.

For example, let  $\phi$  be the formula  $(p \vee \neg q) \wedge \neg(q \wedge r)$ . To rewrite it in CNF, we start by constructing the truth table of its negation  $\neg\phi$ . This allows us to express  $\neg\phi$  in DNF.

$p$	$q$	$r$	$((p \vee \neg q) \wedge \neg(q \wedge r))$	Negation of $((p \vee \neg q) \wedge \neg(q \wedge r))$
0	0	0	1	0
0	0	1	1	0
0	1	0	0	1
0	1	1	0	1
1	0	0	1	0
1	0	1	1	0
1	1	0	1	0
1	1	1	0	1

Table 3: The truth table for the negation of  $((p \vee \neg q) \wedge \neg(q \wedge r))$ . This is obtained by flipping the results of Table 2.

Hence we have

$$\begin{aligned}
\neg\phi &= (\neg p \wedge q) \vee (p \wedge q \wedge r) && \text{(DNF of } \neg\phi) \\
\neg\neg\phi &= \neg((\neg p \wedge q) \vee (p \wedge q \wedge r)) && \text{(negating both sides)} \\
\phi &= (p \vee \neg q) \wedge (\neg p \vee \neg q \vee \neg r) && \text{(double negation; De Morgan's laws)}
\end{aligned}$$

which gives us  $\phi$  in CNF.

## 1.2 First-order logic

### 1.2.1 Syntax

A first-order language  $L(C, F, P)$  is determined by a set  $C$  of constant symbols, a set  $F$  of function symbols and a non-empty set  $P$  of predicate symbols. Each function symbol and predicate symbol has an associated *arity*  $n \in \mathbb{N}$ . We write  $f^n$  and  $p^n$  to represent an  $n$ -ary function symbol and an  $n$ -ary predicate symbol respectively. Moreover, let  $V$  be a countably infinite set of variable symbols.

$$\begin{aligned}
\text{term} &:= c \mid v \mid f^n(\text{term}_0, \text{term}_1, \dots, \text{term}_{n-1}) && \text{(where } c \in C, v \in V \text{ and } f^n \in F) \\
\text{atom} &:= p^n(\text{term}_0, \text{term}_1, \dots, \text{term}_{n-1}) && \text{(where } p^n \in P) \\
\text{formula} &:= \text{atom} \mid \neg\text{formula} \mid (\text{formula}_0 \vee \text{formula}_1) \mid \exists v \text{ formula} && \text{(where } v \in V)
\end{aligned}$$

This definition is functionally complete. Formulas involving universal quantifiers, implications and equivalence symbols can always be rewritten using only symbols defined above.

A *closed term* is a term with no variable symbols. A *sentence* is a formula with no free variables.

### 1.2.2 Semantics

For a first-order language  $L(C, F, P)$ , we may construct a corresponding first-order structure<sup>2</sup>  $S = (D, I)$  where  $I = (I_c, I_f, I_p)$ .

$$S = ( \underbrace{D}_{\text{non-empty domain}}, \overbrace{(I_c, I_f, I_p)}^{\text{interpretation } I} )$$

Here,

- $I_c$  maps each constant symbol in  $C$  to an element of  $D$ .
- $I_f$  maps each  $n$ -ary function symbol in  $F$  to an  $n$ -ary function over  $D$ .
- $I_p$  maps each  $n$ -ary predicate symbol  $p \in P$  to an  $n$ -ary relation over  $D$  (i.e. a subset of  $D^n$ ).

---

<sup>2</sup>Also known as an  $L$ -structure.

- We may occasionally use  $I$  to denote a general interpretation function where

$$\begin{aligned} I(c) &= I_c(c) && \text{(for all } c \in C) \\ I(f) &= I_f(f) && \text{(for all } f \in F) \\ I(p) &= I_p(p) && \text{(for all } p \in P) \end{aligned}$$

If  $P$  includes the equality symbol  $=$ , then it is always interpreted as the binary relation of true equality.

$$I_p(=) = \{(d, d) : d \in D\}$$

Given a structure  $S = (D, I)$ , a variable assignment  $A$  is a map from  $V$  to  $D$ . For any variable  $v \in V$ , two variable assignments  $A$  and  $A^*$  are said to be  $v$ -equivalent if  $A(x) = A^*(x)$  for all  $x \in V \setminus \{v\}$ . In other words, two variable assignments are said to be  $v$ -equivalent if they are completely identical except possibly for the element in  $D$  assigned to  $v$ . This is written as  $A \equiv_v A^*$ .

Given a structure  $S$  and a variable assignment  $A$ , we may interpret any term as follows.

$$\begin{aligned} c^{S,A} &= I_c(c) \\ v^{S,A} &= A(v) \\ f^n(t_0, t_1, \dots, t_{n-1})^{S,A} &= \underbrace{(I_f(f^n))}_{\text{interpreted function}}(t_0^{S,A}, t_1^{S,A}, \dots, t_{n-1}^{S,A}) \end{aligned}$$

Formulas are evaluated as follows.

$$\begin{aligned} S \models_A p^n(t_0, t_1, \dots, t_{n-1}) &\iff (t_0^{S,A}, t_1^{S,A}, \dots, t_{n-1}^{S,A}) \in I_p(p^n) \\ S \models_A \neg \text{formula} &\iff S \not\models_A \text{formula} \\ S \models_A (\text{formula}_0 \vee \text{formula}_1) &\iff S \models_A \text{formula}_0 \text{ or } S \models_A \text{formula}_1 \\ S \models_A \exists v \text{ formula} &\iff S \models_{A[x \mapsto d]} \text{formula for some } d \in D \end{aligned}$$

Given a structure  $S$  and a formula  $\phi$ , we say that

- $\phi$  is “valid in  $S$ ” if  $S \models_A \phi$  for every variable assignment  $A$ . This is written as  $S \models \phi$ .
- $\phi$  is “satisfiable in  $S$ ” if  $S \models_A \phi$  for some variable assignment  $A$ .
- $\phi$  is “valid” if  $\phi$  is valid in all possible structures. This is written as  $\models \phi$ .
- $\phi$  is “satisfiable” if there exists some structure in which  $\phi$  is satisfiable.

A formula  $\phi$  is valid if and only if  $\neg\phi$  is not satisfiable.

**Proof.** Let  $\neg\phi$  be a formula that is not satisfiable. Hence we have

$$\begin{aligned} \neg \exists S \exists A \quad S \models_A \neg\phi &\iff \neg \exists S \exists A \quad S \not\models_A \phi \\ &\iff \forall S \neg \exists A \quad S \not\models_A \phi \\ &\iff \forall S \forall A \quad \neg S \not\models_A \phi \\ &\iff \forall S \forall A \quad S \models_A \phi \end{aligned}$$

which means  $S$  is valid.

If  $\phi$  is a sentence, then  $\phi$  is valid in  $S$  if and only if it is also satisfiable in  $S$ .

### 1.2.3 Example: Arithmetic in the set of natural numbers

Consider the first-order language  $L(C, F, P)$  defined as follows. Also assume a countably infinite set  $V$  of variable symbols.

$$\begin{array}{ll} C = 1, 2, 3, \dots & \text{(constant symbols)} \\ F = \{+, \times\} & \text{(function symbols, both binary)} \\ P = \{=, <\} & \text{(predicate symbols, both binary)} \\ V = \{x, y, z, \dots\} & \text{(variable symbols)} \end{array}$$

A term is a string of symbols that represents a “thing” or an “object” — this can be a constant, a variable, or a function output.

- $x$
- $1 + 3$
- $2 \times x + 1$

Of the terms shown above, only the second one is a closed terms because it has no variable symbols.

An atom is a string of symbols that represents the output of a predicate, which is a truth value.

- $1 = 2$
- $y < 3$
- $x + 1 < 2 \times z + 3$

Finally, a formula is constructed by applying negations, disjunctions, and existential quantifiers to atoms.

- $1 = 2 \wedge y < 3$
- $\neg \exists z \ x + 1 < 2 \times z + 3$

The latter example is a sentence because all of its variable symbols are bounded.

For this particular first-order language, we may use the structure of ordinary arithmetic<sup>3</sup>, defined as  $N = \{\mathbb{N}, \{I_c, I_f, I_p\}\}$  where

- $I_c$  is a function that maps numerical symbols to the corresponding natural number.

$$\begin{aligned} I_c(1) &= 1 \\ I_c(2) &= 2 \\ I_c(3) &= 3 \\ &\vdots \end{aligned}$$

- $I_f$  maps  $+$  and  $\times$  to the addition and multiplication operations in arithmetic respectively.
- $I_p$  maps  $=$  and  $<$  to the following relations.

$$\begin{aligned} I_p(=) &= \{(n, n) : n \in \mathbb{N}\} \\ I_p(<) &= \{(m, n) \in \mathbb{N}^2 : m < n\} \end{aligned}$$

---

<sup>3</sup>There is also a similar structure  $R = (\mathbb{R}, I)$  where the domain is the set of real numbers.

### 1.2.4 First-order structures and directed graphs

Consider a first-order language with only one binary predicate symbol  $p$ .

$$L(C, F, \{p\})$$

Any first-order structure  $S = \{D, \{I_c, I_f, I_p\}\}$  for this language can be represented as a directed graph, where each vertex is an element of  $D$  and each directed edge represents an element of the relation  $I_p(p)$ .

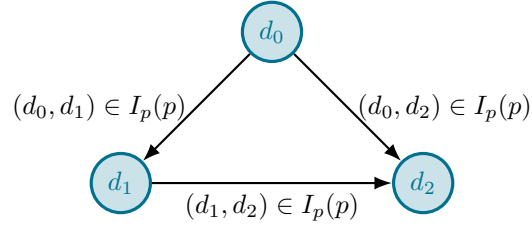


Figure 3: The first-order structure  $S$  can be visualised as a directed graph.



## 2 Axiomatic Proofs for Propositional Logic

A *proof system* is a system for determining the validity of formulas.

An obvious system would be to construct a truth table and check that all rows give a true result. However, this naive approach has an exponential time complexity<sup>4</sup>, meaning that it will become increasingly impractical as more and more propositions are introduced.

To alleviate this issue, we shall introduce a different approach called a *Hilbert-style proof system*. This is an *axiomatic proof system* in which theorems are generated using axioms and inference rules.

### 2.1 Hilbert-style proof system

Firstly, we limit our propositional language to only use the connectives  $\neg$  and  $\rightarrow$ . Double negations are prohibited.

Moreover, we will note some *axioms* that are known to be valid, and then try to derive other valid formulas from the axioms. Below we list three examples of *schemas*, from which axioms may be obtained by substituting any formulas in place of  $p$ ,  $q$  and  $r$ .

I.  $p \rightarrow (q \rightarrow p)$  (implication is true if consequent is true)

II.  $(p \rightarrow (q \rightarrow r)) \rightarrow ((p \rightarrow q) \rightarrow (p \rightarrow r))$  (implication chain as hypothetical syllogism)

III.  $(\neg p \rightarrow \neg q) \rightarrow (q \rightarrow p)$  (contrapositive)

Axioms on their own are insufficient in establishing a proof system. We also need *inference rules*, which stipulate how conclusions can be derived from premises. One of the main inference rules is *modus ponens*, which states that if you have proved both the formula  $\phi$  and the implication  $(\phi \rightarrow \psi)$ , then you may deduce the conclusion  $\psi$ .

$$\frac{\phi \quad (\phi \rightarrow \psi)}{\psi} \quad (\text{modus ponens})$$

In this system, a *proof* is a sequence of formulas

$$\phi_0, \phi_1, \phi_2, \dots, \phi_n$$

such that for each  $i \leq n$ , the formula  $\phi_i$  is either

- an axiom; or
- obtained from two previous formulas  $\phi_j$  and  $\phi_k$  in the sequence via modus ponens (for some  $j, k < i$ ).

If such a proof exists, then the final formula  $\phi_n$  is called a *theorem* and we may write  $\vdash \phi_n$ .

---

<sup>4</sup>Using this system, checking the validity of a formula with  $n$  proposition symbols requires  $2^n$  computations.

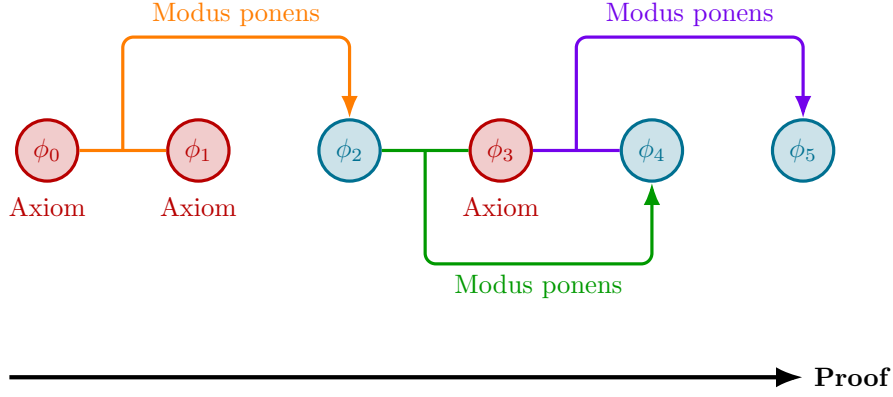


Figure 4: In a proof, every formula must be either an axiom, or derived from previous formulas via modus ponens.

For example, the theorem

$$\vdash (p \rightarrow p)$$

may be proved using the above proof system as follows.

1.  $(p \rightarrow ((p \rightarrow p) \rightarrow p)) \rightarrow ((p \rightarrow (p \rightarrow p)) \rightarrow (p \rightarrow p))$  (Axiom I, replacing  $p, q, r$  by  $p, (p \rightarrow p), p$ )
2.  $p \rightarrow ((p \rightarrow p) \rightarrow p)$  (Axiom II, replacing  $p, q$  by  $p, (p \rightarrow p)$ )
3.  $(p \rightarrow (p \rightarrow p)) \rightarrow (p \rightarrow p)$  (modus ponens, via 1 and 2)
4.  $p \rightarrow (p \rightarrow p)$  (Axiom I, replacing  $p, q$  by  $p, p$ )
5.  $p \rightarrow p$  (modus ponens, via 3 and 4)

To include double negations and other connectives like  $\wedge$  and  $\vee$ , we may add more axioms to our proof system.

- IV.  $p \rightarrow \neg\neg p$  and  $\neg\neg p \rightarrow p$  (double negation)
- V.  $(p \vee q) \rightarrow (\neg p \rightarrow q)$  and  $(\neg p \rightarrow q) \rightarrow (p \vee q)$  (implication as disjunction)
- VI.  $(p \wedge q) \rightarrow \neg(p \rightarrow \neg q)$  and  $\neg(p \rightarrow \neg q) \rightarrow (p \wedge q)$  (implication as conjunction)

## 2.2 Proofs with assumptions and the principle of explosion

Let  $\Gamma$  be a set of *assumptions*, i.e. formulas that are assumed to be true. Under these assumptions, a proof is defined as a sequence of formulas

$$\phi_0, \phi_1, \phi_2, \dots, \phi_n$$

such that for each  $i \leq n$ , the formula  $\phi_i$  is either

- an axiom;
- an assumption  $\phi_i \in \Gamma$ ; or
- obtained from two previous formulas  $\phi_j$  and  $\phi_k$  in the sequence via modus ponens (for some  $j, k < i$ ).

If such a proof exists, then we may write  $\Gamma \vdash \phi_n$ .

For example, given the set of assumptions  $\Gamma = \{p\}$ , we may prove that  $q \rightarrow p$  using the Hilbert-style proof system, as demonstrated below.

1.  $p \rightarrow (q \rightarrow p)$  (Axiom I)
2.  $p$  (Assumption)
3.  $q \rightarrow p$  (modus ponens, via 1 and 2)

Proving with assumptions can be quite tricky due to the *principle of explosion*<sup>5</sup>, which states that any statement can be proven from a contradiction. In other words, it is possible to prove any given statement, true or false, using a proof system as long as at least one of the assumptions in  $\Gamma$  is false.

We shall illustrate this principle as follows. Let  $\Gamma$  be the set containing the invalid assumption  $\neg(q \rightarrow q)$ . We will use the Hilbert-style proof system to prove an arbitrary formula  $p$  under this assumption.

5.  $q \rightarrow q$  (proven previously)
6.  $(q \rightarrow q) \rightarrow \neg\neg(q \rightarrow q)$  (Axiom IV, replacing  $p$  by  $q$ )
7.  $\neg\neg(q \rightarrow q)$  (modus ponens, via 5 and 6)
8.  $\neg\neg(q \rightarrow q) \rightarrow (\neg p \rightarrow \neg\neg(q \rightarrow q))$  (Axiom I, replacing  $p, q$  by  $\neg\neg(q \rightarrow q), \neg p$ )
9.  $\neg p \rightarrow \neg\neg(q \rightarrow q)$  (modus ponens, via 7 and 8)
10.  $(\neg p \rightarrow \neg\neg(q \rightarrow q)) \rightarrow (\neg(q \rightarrow q) \rightarrow p)$  (Axiom III, replacing  $p, q$  by  $p, \neg\neg(q \rightarrow q)$ )
11.  $\neg(q \rightarrow q) \rightarrow p$  (modus ponens, via 9 and 10)
12.  $\neg(q \rightarrow q)$  (assumption)
13.  $p$  (modus ponens, via 11 and 12)

## 2.3 Soundness, completeness and termination

A proof system is said to be *sound* if it can only prove valid theorems. In other words, anything proven using a sound system must be valid.

$$\underbrace{\vdash \phi}_{\text{proven}} \implies \underbrace{\models \phi}_{\text{valid}} \quad (\text{soundness})$$

Conversely, a proof system is said to be *complete* if it can prove any given valid theorem. In other words, if a formula is valid, it must be possible to prove it under a complete system.

$$\underbrace{\models \phi}_{\text{valid}} \implies \underbrace{\vdash \phi}_{\text{proven}} \quad (\text{completeness})$$

The main problem with the Hilbert-style proof system is that although it is relatively easy to check that a proof of a formula is correct, there is no systematic way for efficiently constructing proofs.

Moreover, even if a system is sound and complete, we don't know how long the proof for a given formula might be. Since it is impossible for us to check all the possibilities to see if a proof exists, testing the validity of a formula remains undecidable — there is no effective method for determining validity that terminates in finite time.

---

<sup>5</sup>This principle is sometimes referred to in Latin as *ex falso quodlibet*, which literally translates to “from falsehood, anything [follows]”.

### 3 Propositional tableau

In view of the impracticality of Hilbert-style proof systems, we introduce below an easier and more implementable method for determining a formula's validity — *tableaux*.

Here is a brief overview of how a tableau works. Suppose we want to check the satisfiability of a formula  $\phi$ . This formula will be placed at the root of a binary tree, called a tableau. We use a variety of expansion rules to grow the tree until it is complete. An *open* tableau indicates that  $\phi$  is satisfiable, while a *closed* tableau indicates that  $\phi$  is unsatisfiable.

To determine the validity of a formula, simply construct a tableau for  $\neg\phi$ . If the resultant tableau is open, then  $\neg\phi$  is satisfiable, so  $\phi$  is invalid. On the contrary, if the resultant tableau is closed, then  $\neg\phi$  must be unsatisfiable, so  $\phi$  is valid.

#### 3.1 Constructing a tableau

In a tableau, every node is marked with a formula. To build a tableau for a formula  $\phi$ , begin by placing  $\phi$  at the root of a binary tree. Then, we repeat the following process:

1. Select a formula in the tree that has not been selected before. The formula must not be a literal.
2. Choose the expansion rule (see below) that applies to the selected formula.
3. For each leaf node, add new children nodes in accordance to the chosen expansion rule.
4. Place a tick beside the selected formula to make sure we don't expand it again.

There are two types of expansion rules:

- $\alpha$ -rules, which create one new child per leaf node; and
- $\beta$ -rules, which create two new children per leaf node.

Figures 5 and 6 depict the  $\alpha$ - and  $\beta$  rules respectively. Nodes that are newly created by each rule are highlighted in blue.

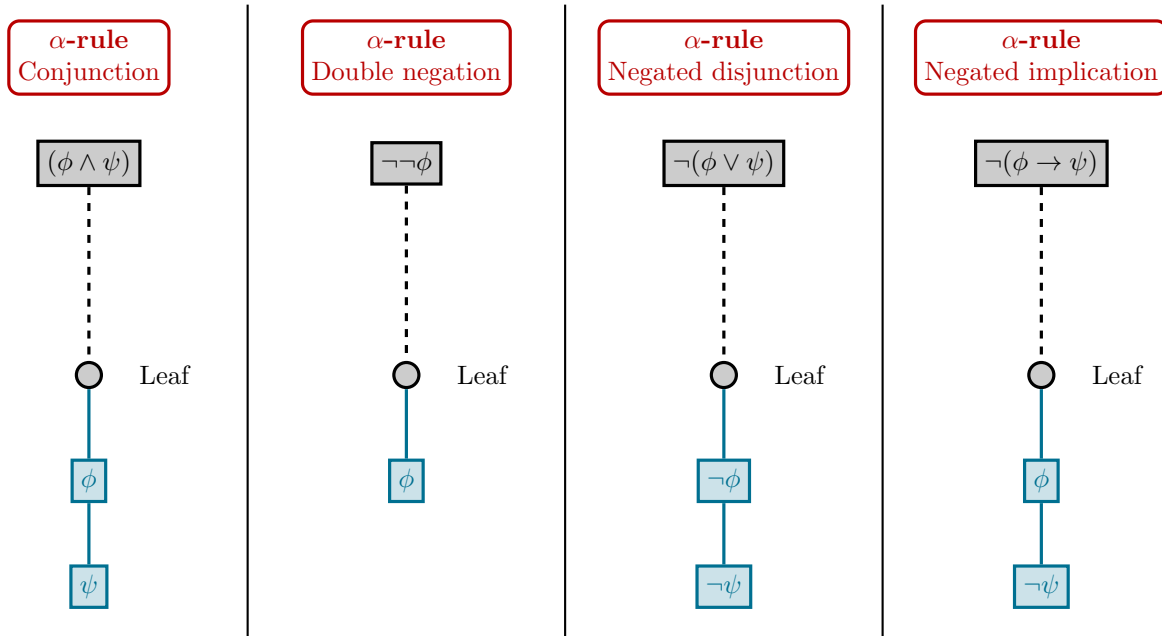


Figure 5: The four  $\alpha$ -rules for constructing propositional tableaux.

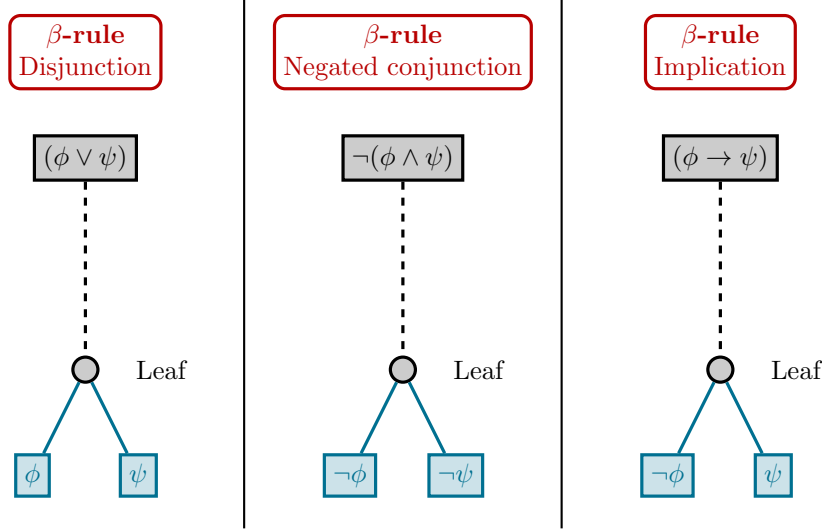


Figure 6: The three  $\beta$ -rules for constructing propositional tableaux.

In general, nodes located in the same branch<sup>6</sup> are considered in conjunction while the different branches are considered to be disjunctive. As a result, a tableau is a tree-like representation of a formula that is a disjunction of conjunctions, à la disjunctive normal form (DNF).

A tableau is considered *complete* if every node is either ticked (already expanded) or a literal. When a tableau is complete, we can determine the original formula's satisfiability as follows.

- A branch containing both a propositional letter and its negation ( $p$  and  $\neg p$ ) is said to be *closed*, which we denote as  $\oplus$ . Otherwise, it is *open*.
- A tableau where all branches are closed is said to be *closed*, meaning that the formula at its root is unsatisfiable. Contrarily, a tableau with at least one open branch is said to be *open*, indicating that the formula is satisfiable.

### 3.2 Example of constructing a tableau and converting to DNF

To check if the formula

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

is satisfiable, we construct its tableau, as shown in figure 7.

Since only one of the four branches is closed, this formula is satisfiable. In fact, the literals in each open branch give a possible valuation that satisfies the given formula. For instance, the second branch from the left contains the literals  $p$  and  $\neg q$ . This indicates that the formula is true when  $p$  is true and  $q$  is false.

<sup>6</sup>A *branch* is defined as a path from the root of the tableau to one of its leaves.

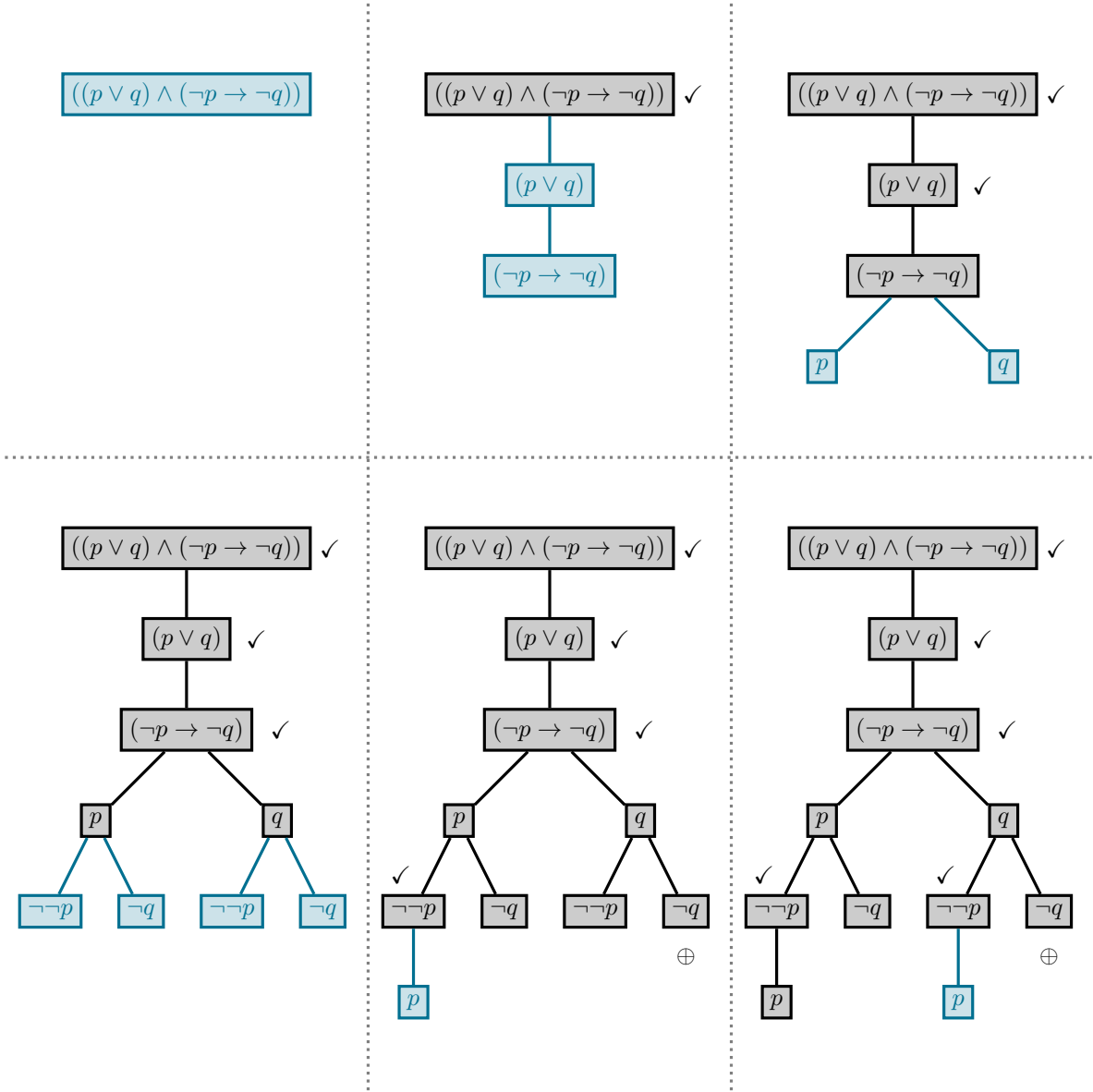


Figure 7: Constructing the tableau of  $((p \vee q) \wedge (\neg p \rightarrow \neg q))$ . Read from left to right and from top to bottom.

It follows that given the tableau of a formula, its DNF equivalent can be expressed as

$$\bigvee_{\text{open branch } \Theta} \left( \bigwedge \{ \text{literals in } \Theta \} \right).$$

As always, the CNF of a formula can be obtained by negating the DNF form of its negation.