Formal Methods in Software Engineering

Re-introduction to Logic, Spring 2026

Konstantin Chukharev

Why Formal Methods?

Motivation

Before we define verification formally, here is *why* it matters.

Software bugs have caused deaths, $billion losses, and mission failures — even when tests passed.

Ariane 5 (1996) — Overflow in 64⟶16-bit conversion destroyed a $370M rocket 37 s after launch. Reused code from Ariane 4 *without re-verification*.

Therac-25 (1985–87) — Race condition in radiation machine caused massive overdoses. At least 3 deaths.

Intel FDIV (1994) — Pentium division error in rare cases. $475M recall; discovered by a mathematician, not by Intel's tests.

Knight Capital (2012) — Faulty trading software deployment. Lost $440M in 45 minutes. Bankrupt within days.

In each case, the system's behavior was never *proven* to match its specification.

Formal methods turn correctness into a *mathematical question* that machines can help answer. Instead of checking *some* executions, we reason about *all* of them.

Formal Reasoning: A Taste

Here is the *kind* of reasoning formal methods make precise and machine-checkable.

Scenario: A server has three properties documented in its specification:

1. If authentication succeeds, a session is created.
2. If a session is created, the user can access the resource.
3. Authentication succeeded.

In logical notation, with propositions A = "auth succeeds", S = "session created", R = "resource accessible":

$$\underbrace{A \to S}_{\text{premise 1}} \quad \underbrace{S \to R}_{\text{premise 2}} \quad \underbrace{A}_{\text{premise 3}}$$

Derivation: From A and $A \to S$ we get S (modus ponens). From S and $S \to R$ we get R (MP again).
Conclusion: the user can access the resource.

> This is a *proof* — a finite chain of justified steps from premises to a conclusion. Formal methods scale this idea to entire programs: premises are code + pre-conditions, and we prove that post-conditions follow.

What Is Verification?

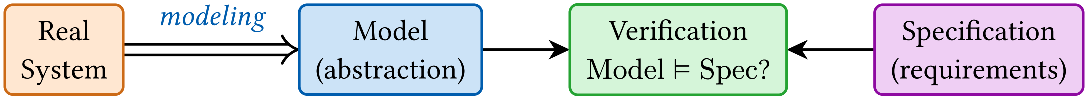
Software verification is the process of establishing that a system *meets its specification*. Three ingredients are needed:

Definition 1: A *model* is a mathematical representation of a system — a finite-state machine, a logical formula, a program abstraction. It captures *what the system does* (or can do), abstracting away irrelevant details.

Definition 2: A *specification* is a precise, formal statement of *desired behavior*: a pre-condition/post-condition pair, an invariant, a temporal property. It captures *what the system should do*.

Definition 3: *Verification* is checking whether a model satisfies a specification: $\text{Model} \vDash \text{Spec}$.

What Is Verification? [2]



Example: Consider a function `abs(x)` that should return $|x|$.
Model: the program code (or its logical encoding).
Specification: result ≥ 0 and (result $= x \lor$ result $= -x$).
Verification: prove that for *all* inputs x, the model satisfies the specification.

Formal methods = build a model + write a specification + prove that Model $\vDash$ Spec.

The three ingredients are inseparable: a model without a spec is meaningless, a spec without verification is wishful thinking.

The Verification Spectrum

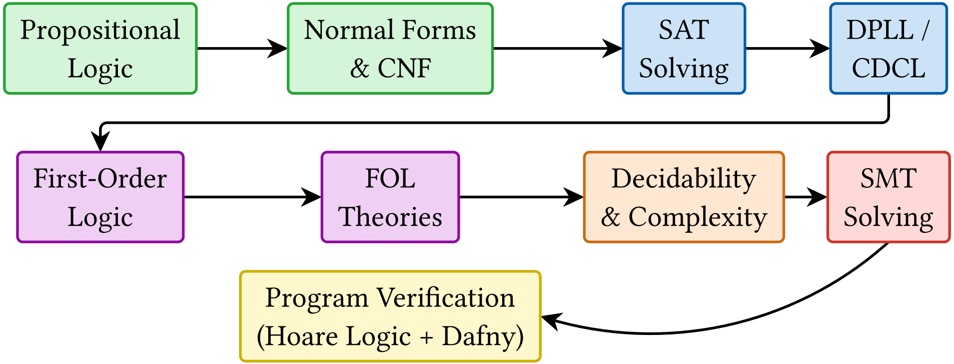
Method	**Rigor**	**Coverage**	**Cost**	**Proves $M \models S$?**
Testing	Low	Partial	Low	No
Static analysis	Medium	Heuristic	Low	Partially
Model checking	High	Exhaustive (bounded)	Medium	Yes (bounded)
Deductive verification	Highest	Complete	High	Yes

This course moves from left to right, ending with deductive verification in Dafny.

> *“Testing shows the presence, not the absence of bugs.”* — Edsger W. Dijkstra (1969)

Course Roadmap

"How do we make machines reason about correctness?"



Propositional Logic Refresher

Syntax vs Semantics: Why Two Perspectives?

Logic has two faces: *syntax* (the strings we write and transform) and *semantics* (what they *mean*).

Syntactic World ($\vdash$)

- Formulas, rewriting rules
- Proof systems, derivations
- *"I can derive α from Γ"*
- Symbol: $\Gamma \vdash \alpha$

Semantic World ($\vDash$)

- Interpretations, truth values
- Truth tables, models
- *"α is true whenever Γ is"*
- Symbol: $\Gamma \vDash \alpha$

For propositional logic, these two worlds are *perfectly aligned* — soundness + completeness gives us $\vdash \Longleftrightarrow \vDash$. So why bother distinguishing them?

Reason 1 — Different algorithms: Semantics gives *truth tables* (2^n rows — brute force). Syntax gives *proof search* (sometimes exponentially shorter). For 300 variables, a truth table has 2^{300} rows (more than atoms in the universe), but a proof might take 50 lines. Same question, vastly different computational cost.

Reason 2 — The gap appears later: For first-order logic over arithmetic, Gödel's Incompleteness Theorem shows there are true statements that *no* proof system can derive: $\vdash \subsetneq \vDash$. When this gap opens, confusing the two perspectives leads to fundamental errors.

Syntax vs Semantics: Why Two Perspectives? [2]

Bottom line:

- Semantics asks *"Is it true?"* (check all interpretations).
- Syntax asks *"Can I derive it?"* (apply inference rules mechanically).

For PL, both always give the same answer.

However, we *train the distinction now* so it is natural when it matters.

PL Syntax

Propositional logic studies *Boolean combinations* of atomic statements.
Its syntax defines which strings are "legal" formulas:

Definition 4 (Well-Formed Formula (WFF)): Given propositional variables $P, Q, R, \ldots$ and constants $\top, \bot$, the set of *well-formed formulas* is defined inductively:

1. Every propositional variable and constant is a WFF.
2. If α and β are WFFs, then $\neg\alpha$, $(\alpha \land \beta)$, $(\alpha \lor \beta)$, $(\alpha \to \beta)$, $(\alpha \Longleftrightarrow \beta)$ are WFFs.
3. Nothing else is a WFF.

Conventions:
Operator precedence: $\neg > \land > \lor > \to > \Longleftrightarrow$.
Outer parentheses omitted. Associativity: $\land$, $\lor$ left-to-right; $\to$ right-to-left.

Example: A Boolean guard `if (x > 0 && !done)` in a program corresponds to the propositional formula $P \land \neg Q$, where P stands for `x > 0` and Q for `done`. This is a WFF by rule 2.

PL Semantics

Definition 5: An *interpretation* (valuation) $\nu : V \to \{0, 1\}$ assigns a truth value to each propositional variable.

The *evaluation* $[\![\alpha]\!]_\nu$ of a formula α under ν is defined recursively:

$$
\begin{aligned}
[\![\top]\!]_\nu &= 1, \quad [\![\bot]\!]_\nu = 0, \quad [\![P]\!]_\nu = \nu(P) \\
[\![\neg\alpha]\!]_\nu &= 1 - [\![\alpha]\!]_\nu \\
[\![\alpha \land \beta]\!]_\nu &= \min([\![\alpha]\!]_\nu, [\![\beta]\!]_\nu) \\
[\![\alpha \lor \beta]\!]_\nu &= \max([\![\alpha]\!]_\nu, [\![\beta]\!]_\nu) \\
[\![\alpha \to \beta]\!]_\nu &= \max(1 - [\![\alpha]\!]_\nu, [\![\beta]\!]_\nu)
\end{aligned}
$$

Definition 6: An interpretation ν *satisfies* a formula α, written $\nu \vDash \alpha$, if $[\![\alpha]\!]_\nu = 1$.

A *model* of α is any interpretation that satisfies it.

PL Semantics [2]

Terminology note: The word "model" appears in many distinct contexts:

- **PL model** = an interpretation (truth assignment) satisfying a formula.
- **FOL model** = a structure (domain + interpretation of symbols) satisfying sentences.
- **Model checking** = algorithmic verification technique (checking if a system model satisfies a temporal property).

In this course, context determines which meaning applies.
For now, "model" = satisfying interpretation.

Example: Let $\nu(P) = 1, \nu(Q) = 0$.

- Then $[\![P \to Q]\!]_\nu = \max(1 - 1, 0) = 0$ and $[\![\neg P \lor Q]\!]_\nu = \max(0, 0) = 0$.
- Both agree, as expected from the equivalence $(P \to Q) \equiv (\neg P \lor Q)$.

Since $[\![P \to Q]\!]_\nu = 0$, we say $\nu \nvDash (P \to Q)$ – this interpretation is *not* a model of $P \to Q$.

PL Semantics [3]

Example: Let $\nu(A) = 1, \nu(B) = 0, \nu(C) = 1$. Evaluate $A \wedge (B \vee C)$:

$$[\![B \vee C]\!]_\nu = \max(0, 1) = 1$$
$$[\![A \wedge (B \vee C)]\!]_\nu = \min(1, 1) = 1$$

The formula is *satisfied* by this interpretation: $\nu \vDash A \wedge (B \vee C)$. So ν is a *model* of $A \wedge (B \vee C)$.

Semantic Classification

Formulas are classified by their truth behavior across *all* interpretations:

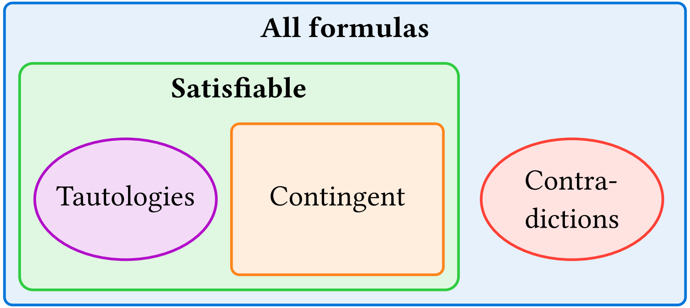
Definition 7 (Semantic Classification): Let α be a WFF.

- α is **valid** (*tautology*), written $\vDash \alpha$, if *every* interpretation is a model: $\nu \vDash \alpha$ for all ν.
- α is **satisfiable** (*consistent*) if it has *at least one* model: $\nu \vDash \alpha$ for some ν.
- α is **unsatisfiable** (*contradiction*) if it has *no* models: $\nu \nvDash \alpha$ for all ν.
- α is **falsifiable** if some interpretation is *not* a model: $\nu \nvDash \alpha$ for some ν.

Example:

- $P \vee \neg P$ – valid (tautology). *Every* interpretation is a model.
- $P \wedge Q$ – satisfiable (has model $\nu(P) = \nu(Q) = 1$) and falsifiable (non-model $\nu(P) = 1, \nu(Q) = 0$). This is *contingent*.
- $P \wedge \neg P$ – unsatisfiable. *No* model exists.

Semantic Classification [2]



Entailment vs Implication

Definition 8 (Semantic Entailment): A set of formulas Γ *semantically entails* α, written $\Gamma \vDash \alpha$, if every model of Γ is also a model of α.

Equivalently: every interpretation satisfying all formulas in Γ also satisfies α.

These two notions — *implication* and *entailment* — are distinct but deeply related:

Implication ($\rightarrow$) is a *connective* — an operator *inside* the language of propositional logic.

- $P \rightarrow Q$ is a well-formed formula with a truth value under each interpretation.
- It can appear in compound formulas: $(P \rightarrow Q) \wedge R$, $\neg(P \rightarrow Q)$, *etc.*
- Defined by a truth table: $[\![P \rightarrow Q]\!]_\nu = \max(1 - [\![P]\!]_\nu, [\![Q]\!]_\nu)$.

Entailment ($\vDash$) is a *metalogical* relation — a claim *about* formulas from outside the logic.

- $P \vDash Q$ is *not* a formula; it is a mathematical statement about all interpretations.
- It cannot be negated or combined using logical connectives.
- Defined by quantifying over models: *"every model of P is also a model of Q."*

Entailment vs Implication [2]

Common mistake: writing $P \vDash Q$ when you mean $P \to Q$, or vice versa.

One ($\to$) is a formula you can evaluate; the other ($\vDash$) is a claim you prove.

Why distinguish them? In propositional logic, they coincide (via the Deduction Theorem). But in first-order logic and beyond, the distinction becomes crucial: some truths are not provable, and the syntactic world ($\vdash$) diverges from the semantic world ($\vDash$).

For PL, these two worlds coincide perfectly: " $\vdash$ " $\iff$ " $\vDash$ ". For first-order logic, Gödel showed that in specific theories (like Peano Arithmetic), *true* sentences can be *unprovable* — the syntactic and semantic worlds diverge. We will see this precisely in the Metatheorems section.

Entailment vs Implication [3]

Theorem 1 (Deduction Theorem (Semantic)): For any formulas α, β:

$$\alpha \vDash \beta \quad \Longleftrightarrow \quad \vDash \alpha \to \beta$$

Proof *(⇒)*: Assume $\alpha \vDash \beta$. We must show $\vDash \alpha \to \beta$.

Let ν be any interpretation. We show $\nu \vDash \alpha \to \beta$.

- If $\nu \nvDash \alpha$, then $[\![\alpha \to \beta]\!]_\nu = \max(0, [\![\beta]\!]_\nu) = 1$, so $\nu \vDash \alpha \to \beta$.
- If $\nu \vDash \alpha$, then since $\alpha \vDash \beta$, we have $\nu \vDash \beta$, so $[\![\alpha \to \beta]\!]_\nu = \max(0, 1) = 1$.

In both cases, $\nu \vDash \alpha \to \beta$. Since ν was arbitrary, $\alpha \to \beta$ is valid. □

Proof *(⇐)*: Assume $\vDash \alpha \to \beta$. We must show $\alpha \vDash \beta$.

Let ν be a model of α, *i.e.*, $\nu \vDash \alpha$. Since $\alpha \to \beta$ is valid, $\nu \vDash \alpha \to \beta$. By the definition of $\to$: $\max(1 - [\![\alpha]\!]_\nu, [\![\beta]\!]_\nu) = 1$. Since $[\![\alpha]\!]_\nu = 1$, we have $\max(0, [\![\beta]\!]_\nu) = 1$, so $[\![\beta]\!]_\nu = 1$, *i.e.*, $\nu \vDash \beta$.

Thus every model of α is a model of β. □

Entailment vs Implication [4]

The Deduction Theorem connects semantic entailment to formula validity — it lets us reduce the question *"does α entail β?"* to *"is $\alpha \to \beta$ valid?"* This reduction is what makes automated validity checking possible: entailment becomes a satisfiability check.

Example: $\{P, P \to Q\} \vDash Q$ (modus ponens as entailment)

Via Deduction Theorem: This is equivalent to $P \land (P \to Q) \to Q$ being valid.

Check: any interpretation either falsifies $P \land (P \to Q)$ (making the implication vacuously true), or satisfies both P and $P \to Q$, in which case it must satisfy Q (by the truth table for $\to$).

Example: $\neg(P \land Q) \vDash \neg P \lor \neg Q$ (De Morgan, semantic form)

Via Deduction Theorem: Equivalent to $\vDash \neg(P \land Q) \to (\neg P \lor \neg Q)$, which is a tautology.

Entailment vs Implication [5]

Theorem 2 (Generalized Deduction Theorem): For any set of formulas Γ and formulas α, β:

$$\Gamma \cup \{\alpha\} \vDash \beta \quad \Longleftrightarrow \quad \Gamma \vDash \alpha \to \beta$$

Proof *($\Rightarrow$)*: Assume $\Gamma \cup \{\alpha\} \vDash \beta$. Let ν be a model of Γ. We show $\nu \vDash \alpha \to \beta$.

- If $\nu \nvDash \alpha$, then $\nu \vDash \alpha \to \beta$ (vacuously).
- If $\nu \vDash \alpha$, then ν models all formulas in $\Gamma \cup \{\alpha\}$, so by assumption $\nu \vDash \beta$, hence $\nu \vDash \alpha \to \beta$. $\square$

Proof *($\Leftarrow$)*: Assume $\Gamma \vDash \alpha \to \beta$. Let ν be a model of $\Gamma \cup \{\alpha\}$. Then $\nu \vDash \Gamma$, so $\nu \vDash \alpha \to \beta$ by assumption. Since $\nu \vDash \alpha$, we have $\nu \vDash \beta$ by modus ponens. $\square$

Note: This theorem justifies the *hypothetical reasoning* pattern: to show $\Gamma \vDash \alpha \to \beta$, it suffices to show $\Gamma \cup \{\alpha\} \vDash \beta$ — *i.e.*, *"assume α as an additional hypothesis and derive β."*

SAT vs VALID Duality

Satisfiability and validity are *dual* decision problems:

$$\begin{aligned} \text{SAT:} \quad & \exists \nu.\, \nu \models \alpha \quad \text{(find a model)} \\ \text{VALID:} \quad & \forall \nu.\, \nu \models \alpha \quad \text{(every interpretation is a model)} \end{aligned}$$

α is valid $\iff$ $\neg\alpha$ is unsatisfiable.

Example: $P \vee \neg P$ is valid $\iff \neg(P \vee \neg P) \equiv P \wedge \neg P$ is unsatisfiable.

Checking SAT by truth tables takes $\mathcal{O}(2^n)$ time.
Is there a polynomial algorithm? *This is the P vs NP problem* — a Millennium Prize question.

Fundamental Equivalence Laws

$\alpha \equiv \beta$ iff $\alpha \iff \beta$ is a tautology.

These equivalences form the *toolkit* for normal form transformations. Every conversion (NNF, CNF, DNF) is a sequence of applications of these rewriting rules:

Double Negation:
- $\neg\neg A \equiv A$

De Morgan's Laws:
- $\neg(A \wedge B) \equiv \neg A \vee \neg B$
- $\neg(A \vee B) \equiv \neg A \wedge \neg B$

Implication:
- $(A \to B) \equiv (\neg A \vee B)$

Distributivity:
- $A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$
- $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$

Contraposition:
- $(A \to B) \equiv (\neg B \to \neg A)$

Identity:
- $A \wedge \top \equiv A$
- $A \vee \bot \equiv A$

Complement:
- $A \wedge \neg A \equiv \bot$
- $A \vee \neg A \equiv \top$

Exportation:
- $(A \wedge B) \to C \equiv A \to (B \to C)$

Completeness of Connective Sets

For n Boolean variables, there are 2^{2^n} possible Boolean functions.

How many connectives do we *really* need?

Definition 9 (Functional Completeness): A set S of connectives is *functionally complete* if every Boolean function can be expressed using only connectives from S.

Example:

- $\{\neg, \land, \lor\}$ — the standard Boolean basis.
- $\{\neg, \land\}$ — And-Inverter Graphs (AIGs), used in hardware verification.
- $\{\neg, \lor\}$
- $\{\overline{\land}\}$ — NAND alone suffices.
- $\{\overline{\lor}\}$ — NOR alone suffices.

Example ($\{\land, \to\}$ *is* not *complete.*): Let α be any WFF using only $\land$ and $\to$, and let ν assign 1 to every variable. By structural induction, $[\![\alpha]\!]_\nu = 1$ for all such α.
But $\neg P$ evaluates to 0 under this ν — so $\neg P$ is not expressible.

Completeness of Connective Sets [2]

Why FM cares: In hardware verification, circuits are built from NAND/NOR gates — completeness guarantees these gates can implement *any* Boolean function. In SAT solving, CNF uses only $\{\neg, \wedge, \vee\}$ — so no expressiveness is lost.

Normal Forms

Why Normal Forms?

We know PL formulas can express any Boolean function. But SAT solvers don't accept *arbitrary* formulas — they need a standardized input format. Normal forms provide exactly this: every formula is rewritten into a restricted shape that algorithms can uniformly process.

Three normal forms, each with different trade-offs:

- **Negation Normal Form (NNF)** — negations pushed to atoms; cheap to compute, preserves structure
- **Conjunctive Normal Form (CNF)** — conjunction of clauses; *the language of SAT solvers*
- **Disjunctive Normal Form (DNF)** — disjunction of cubes; dual of CNF

Every propositional formula can be converted to an *equivalent* formula in any of these forms. The key question is: *at what cost?*

Literals and Their Complements

Definition 10 (Literal): A *literal* is a propositional variable (p — *positive*) or its negation ($\neg p$ — *negative*).

Definition 11 (Complement): The *complement* of a literal ℓ is denoted $\overline{\ell}$:

$$\overline{\ell} = \begin{cases} \neg p \text{ if } \ell \equiv p & \text{(positive)} \\ p \quad \text{if } \ell \equiv \neg p & \text{(negative)} \end{cases}$$

Complementary literals ℓ and $\overline{\ell}$ always satisfy $\ell \wedge \overline{\ell} \equiv \bot$ and $\ell \vee \overline{\ell} \equiv \top$.

Clauses and Cubes

Definition 12 (Clause): A *clause* is a disjunction of literals: $\ell_1 \lor \ell_2 \lor \ldots \lor \ell_k$.

- An *empty clause* $\square$ contains no literals and is *unsatisfiable* (false in every interpretation).
- A *unit clause* contains exactly one literal.
- A *Horn clause* contains at most one positive literal.

Definition 13 (Cube): A *cube* is a conjunction of literals: $\ell_1 \land \ell_2 \land \ldots \land \ell_k$.

Clauses and cubes are *dual* notions:

- A clause is *falsified* only if *every* literal in it is false.
- A cube is *satisfied* only if *every* literal in it is true.

Horn clauses are computationally special: SAT restricted to Horn clauses is solvable in *linear time* via unit propagation. Prolog's inference engine works exclusively with Horn clauses.

Negation Normal Form

Definition 14 (Negation Normal Form (NNF)): A formula is in NNF if:

1. Negation ($\neg$) is applied only to *atoms* (propositional variables).
2. The only connectives are $\land$, $\lor$, and $\neg$ (applied to atoms).

Grammar:

$$
\begin{aligned}
\langle \text{Atom} \rangle &::= \top \mid \bot \mid \langle \text{Variable} \rangle \\
\langle \text{Literal} \rangle &::= \langle \text{Atom} \rangle \mid \neg \langle \text{Atom} \rangle \\
\langle \text{Formula} \rangle &::= \langle \text{Literal} \rangle \mid \langle \text{Formula} \rangle \land \langle \text{Formula} \rangle \mid \langle \text{Formula} \rangle \lor \langle \text{Formula} \rangle
\end{aligned}
$$

In words: only $\land$, $\lor$, and negation applied directly to variables — no $\rightarrow$, no $\Longleftrightarrow$, no nested $\neg$.

Example:
- $(p \land q) \lor (\neg p \land \neg q)$ — in NNF.
- $\neg(p \land q)$ — *not* in NNF (negation applied to a compound formula).

NNF Transformation

Rewriting rules (apply until no rule matches):

Description	Rewrite rule
Eliminate implications	$(A \to B) \implies (\neg A \lor B)$
Eliminate biconditionals	$(A \iff B) \implies (\neg A \lor B) \land (A \lor \neg B)$
De Morgan (conjunction)	$\neg(A \land B) \implies (\neg A \lor \neg B)$
De Morgan (disjunction)	$\neg(A \lor B) \implies (\neg A \land \neg B)$
Double negation	$\neg\neg A \implies A$

NNF Transformation [2]

Theorem 3: Every formula *not containing* $\iff$ can be converted to an equivalent NNF with a *linear increase* in size.

Formulas *containing* $\iff$ may suffer *exponential blowup* when converted to NNF.

Why the blowup? The biconditional $A \iff B$ expands to $(\neg A \lor B) \land (A \lor \neg B)$ — producing *two copies* of A and B. A chain of n biconditionals $p_1 \iff p_2 \iff \ldots \iff p_n$ doubles the formula at each level, producing 2^n copies. The Tseitin transformation (coming soon) avoids this.

NNF Transformation: Worked Example

Convert $(P \to Q) \to R$ to NNF step by step:

$$\begin{aligned}
& (P \to Q) \to R \\
\implies\ & \neg(P \to Q) \lor R && \text{(eliminate outer} \to \text{)} \\
\implies\ & \neg(\neg P \lor Q) \lor R && \text{(eliminate inner} \to \text{)} \\
\implies\ & (\neg\neg P \land \neg Q) \lor R && \text{(De Morgan)} \\
\implies\ & (P \land \neg Q) \lor R && \text{(Double negation)}
\end{aligned}$$

Result: $(P \land \neg Q) \lor R$ – negations only on atoms.

Conjunctive Normal Form

Definition 15 (Conjunctive Normal Form (CNF)): A formula is in CNF if it is a conjunction of clauses:

$$\alpha = \bigwedge_i \bigvee_j \ell_{ij}$$

Example: $\alpha = (\neg p \vee q) \wedge (\neg p \vee q \vee r) \wedge (\neg q)$ — CNF with 3 clauses.

Why CNF? Every modern SAT solver (MiniSat, CaDiCaL, Kissat) operates on CNF. Satisfaction requires *at least one* literal per clause — this “one per clause” structure is what makes unit propagation and resolution work.

Disjunctive Normal Form

Definition 16 (Disjunctive Normal Form (DNF)): A formula is in DNF if it is a disjunction of cubes:

$$\alpha = \bigvee_i \bigwedge_j \ell_{ij}$$

Example: $\alpha = (p \land q) \lor (\neg p \land q \land r) \lor (\neg q)$ — DNF with 3 cubes.

CNF vs DNF — dual complexities:

Problem	**On CNF**	**On DNF**
SAT check	NP-complete	Polynomial
VALID check	Polynomial	co-NP-complete

- SAT on DNF is polynomial: check if any cube has no complementary literals.
- VALID on CNF is polynomial: check if every clause contains complementary literals.

CNF Transformation

Any formula can be converted to CNF:

1. Apply NNF transformation rules.
2. Distribute $\lor$ over $\land$ (flattening):
   - $A \lor (B \land C) \Longrightarrow (A \lor B) \land (A \lor C)$
   - $(A \land B) \lor C \Longrightarrow (A \lor C) \land (B \lor C)$
3. Normalize: flatten nested $\land$ and $\lor$.

Theorem 4: Every formula can be converted to an *equivalent* CNF, but the size may grow *exponentially*.

The *distributive law* is the culprit:

$$\underbrace{(x_1 \land y_1) \lor (x_2 \land y_2) \lor \dots \lor (x_n \land y_n)}_{n \text{ cubes}} \overset{\text{CNF}}{\Longrightarrow} \underbrace{\dots}_{2^n \text{ clauses}}$$

Question: Can we avoid this blowup? *Yes — by relaxing "equivalence" to "equisatisfiability".*

Equisatisfiability

Definition 17: Two formulas α and β are *equisatisfiable* if α is satisfiable iff β is satisfiable.

Note: Equisatisfiability is *weaker* than logical equivalence. For SAT solving, equisatisfiability suffices — we only care *whether* a satisfying assignment exists. Any model of the equisatisfiable formula can be *restricted* to the original variables.

Example: $P \land Q$ and $(P \land Q) \land (n \iff P)$ are equisatisfiable but *not* equivalent — the second formula has an extra variable n and is defined over a strictly larger language.

Tseitin Transformation

Definition 18: The *Tseitin transformation* converts any formula to CNF in *polynomial time* by introducing *fresh* variables.

For each non-literal subformula A of a formula F:

1. Introduce a fresh propositional variable n_A.
2. Add a *definitional clause* $n_A \iff A$ (asserting equivalence).
3. Replace A with n_A in F.

The resulting formula is *equisatisfiable* with the original:

- Every model of F extends to a model of the Tseitin encoding.
- Every model of the encoding restricted to original variables satisfies F.

Tseitin Transformation [2]

□ Cost

- $\mathcal{O}(n)$ fresh variables and $\mathcal{O}(n)$ clauses, where n is the formula size.
- Each definitional clause $n \iff A$ for a binary connective produces a *constant number* of clauses.

Example: The definition $n \iff (A \land B)$ is equivalent to:

$$(n \to (A \land B)) \land ((A \land B) \to n) \quad \equiv \quad (\neg n \lor A) \land (\neg n \lor B) \land (\neg A \lor \neg B \lor n)$$

That is: 3 clauses.

Tseitin Transformation: Example

Example: $F = p_1 \Longleftrightarrow (p_2 \Longleftrightarrow (p_3 \Longleftrightarrow (p_4 \Longleftrightarrow (p_5 \Longleftrightarrow p_6))))$

Equivalent CNF: $2^5 = 32$ clauses (exponential).

Tseitin transformation: introduce fresh variables n_3, n_4, n_5:

$$\begin{aligned} S = p_1 &\Longleftrightarrow (p_2 \Longleftrightarrow n_3) \wedge \\ n_3 &\Longleftrightarrow (p_3 \Longleftrightarrow n_4) \wedge \\ n_4 &\Longleftrightarrow (p_4 \Longleftrightarrow n_5) \wedge \\ n_5 &\Longleftrightarrow (p_5 \Longleftrightarrow p_6) \end{aligned}$$

Each biconditional definition produces a constant number of clauses (4 clauses for $\Longleftrightarrow$).

Equisatisfiable CNF: 16 clauses, 3 fresh variables. *Linear growth.*

Clausal Form

Definition 19: A *clausal form* of a formula F is a *set* of clauses S_F which is satisfiable iff F is satisfiable. Moreover, F and S_F have the same models when restricted to the language of F.

The main advantage: any formula can be converted to clausal form in *almost linear* time.

Algorithm:

1. If $F = C_1 \land \ldots \land C_n$ where each C_i is already a clause, then $S_F = \{C_1, \ldots, C_n\}$.
2. Otherwise, apply Tseitin transformation: name each non-literal subformula with a fresh variable.

Key insight: The clausal form is the bridge between arbitrary formulas and SAT solvers. It preserves satisfiability while keeping the representation compact.

Proof Systems

Why Proof Systems?

Clausal form gives us the *input format* for SAT solvers. But before we build solvers (next lecture), we need to understand what it means to *prove* things mechanically — that is the purpose of proof systems.

Truth tables require 2^n rows for n variables. For 300 variables (modest by industrial standards), that exceeds the number of atoms in the universe.

Proof systems *derive* validity step by step using inference rules. A clever proof can be *exponentially shorter* than brute-force enumeration.

Definition 20: A **proof system** derives valid formulas (or entailments) by applying *inference rules* to *axioms* and *assumptions*.

Three main traditions:
- **Hilbert-style:** many axiom schemas, one rule (modus ponens). Compact to define, hard to use.
- **Natural deduction** (Gentzen, 1934): no axioms, intro/elim rules per connective. *Our primary tool.*
- **Sequent calculus** (Gentzen, 1934): manipulates structured judgments. Foundation of automated proof search.

Natural Deduction

A proof system with *no axioms* — only inference rules. Each connective has *introduction* rules (**how to build** a compound formula) and *elimination* rules (**how to use** one).

We present proofs in **Fitch notation**: a numbered list of steps. Each step contains a formula and a *justification* (the rule applied + referenced line numbers).

Subproofs (indented blocks) introduce a *temporary assumption*. Everything derived inside a subproof depends on that assumption. When the subproof closes, the assumption is *discharged* — you may no longer cite its internal lines, but you can reference the subproof *as a whole*.

Mental model: A subproof says *"if I temporarily assume α, I can derive β."* When it closes, you conclude $\alpha \to \beta$ — without assuming α anymore.

Conjunction and Implication Rules

$\wedge$-introduction ($\wedge$i):
From α and β on separate lines, conclude $\alpha \wedge \beta$.

1	α	*Premise*
2	β	*Premise*
3	$\alpha \wedge \beta$	*$\wedge$i 1, 2*

$\wedge$-elimination ($\wedge$e):
From $\alpha \wedge \beta$, conclude α (or β).

1	$\alpha \wedge \beta$	*Premise*
2	α	*$\wedge$e 1*
3	β	*$\wedge$e 1*

Conjunction and Implication Rules [2]

$\rightarrow$-introduction ($\rightarrow$i):

Open a subproof assuming α, derive β.

Close subproof, conclude $\alpha \rightarrow \beta$.

1	$\quad \alpha$	*assumption*
2	$\quad \vdots$	
3	$\quad \beta$	$\vdots$
4	$\alpha \rightarrow \beta$	*$\rightarrow$i 1–3*

$\rightarrow$-elimination ($\rightarrow$e):

Modus ponens.

From α and $\alpha \rightarrow \beta$, conclude β.

1	α	*Premise*
2	$\alpha \rightarrow \beta$	*Premise*
3	β	*$\rightarrow$e 1, 2*

Disjunction Rules

∨-introduction (∨i):

From α, conclude $\alpha \vee \beta$ (or $\beta \vee \alpha$).

1	α	*Premise*
2	$\alpha \vee \beta$	∨*i 1*

∨-elimination (∨e):

From $\alpha \vee \beta$, with subproofs deriving γ from each disjunct, conclude γ.

1	$\alpha \vee \beta$	*Premise*
2	$\quad\alpha$	*assumption*
3	$\quad\gamma$	$\vdots$
4	$\quad\beta$	*assumption*
5	$\quad\gamma$	$\vdots$
6	γ	∨*e 1, 2–3, 4–5*

Negation and Absurdity Rules

¬-introduction (¬i):

Assume α, derive $\perp$ (contradiction).

Close subproof, conclude $\neg\alpha$.

1	α	*assumption*
2	$\vdots$	
3	$\perp$	$\vdots$
4	$\neg\alpha$	*¬i 1–3*

¬-elimination (¬e):

From α and $\neg\alpha$, derive $\perp$.

1	α	*Premise*
2	$\neg\alpha$	*Premise*
3	$\perp$	*¬e 1, 2*

Negation and Absurdity Rules [2]

$\bot$-elimination (ex falso quodlibet, $\bot$e):
From $\bot$, derive *any* formula.

1	$\bot$	*Premise*
2	α	*$\bot$e 1*

Reductio ad absurdum (RAA):
Assume $\neg\alpha$, derive $\bot$. Conclude α.

1	$\quad\neg\alpha$	*assumption*
2	$\quad\vdots$	
3	$\quad\bot$	$\vdots$
4	α	*RAA 1–3*

Classical vs Intuitionistic: RAA and LEM ($\alpha \vee \neg\alpha$) are *classical* rules. Dropping them gives *intuitionistic* logic, where $P \vee \neg P$ is not provable.

Two key proof patterns that cover the majority of ND proofs:

- To prove $\neg\varphi$: assume φ, derive $\bot$, apply $\neg$i.
- To prove $\alpha \to \beta$: assume α, derive β, apply $\to$i.

Fitch Proofs: Basic Examples

Example: **Conjunction rearrangement:** $p \land q, r \vdash q \land r$

1	$p \land q$	*Premise*
2	r	*Premise*
3	q	$\land$*e 1*
4	$q \land r$	$\land$*i 3, 2*

Each step cites the rule and the line numbers it depends on. Line 3 *eliminates* the conjunction to extract q; line 4 *introduces* a new conjunction.

Fitch Proofs: Basic Examples [2]

Example: **Modus Tollens:** $A \to B, \neg B \vdash \neg A$

1	$A \to B$	*Premise*
2	$\neg B$	*Premise*
3	$\quad A$	*assumption*
4	$\quad B$	$\to$*e 3, 1*
5	$\quad \bot$	$\neg$*e 4, 2*
6	$\neg A$	$\neg$*i 3–5*

Lines 3–5 form a *subproof*: we temporarily assume A, derive $\bot$, then discharge the assumption to conclude $\neg A$ via ¬i. The vertical bar shows the scope of the assumption — lines 4 and 5 are only accessible *within* the subproof.

Fitch Proofs: Implication Chains

Example: **Hypothetical Syllogism:** $A \to B, B \to C \vdash A \to C$

1	$A \to B$	*Premise*
2	$B \to C$	*Premise*
3	$\quad A$	*assumption*
4	$\quad B$	$\to$*e 3, 1*
5	$\quad C$	$\to$*e 4, 2*
6	$A \to C$	$\to$*i 3–5*

To prove an implication $A \to C$, we *assume* A (line 3), derive C (line 5), and close with $\to$i.

Note: More worked Fitch proofs (disjunctive syllogism, De Morgan) appear in the exercises. The key patterns: $\vee$-elimination requires two subproofs (one per disjunct); $\neg$-introduction assumes φ, derives $\bot$, concludes $\neg\varphi$.

Soundness and Completeness

Definition 21 (Soundness): A proof system is *sound* if every provable formula is valid:

$$\text{If } \Gamma \vdash \alpha \text{ then } \Gamma \vDash \alpha. \qquad \text{(Nothing false is provable.)}$$

Definition 22 (Completeness): A proof system is *complete* if every valid formula is provable:

$$\text{If } \Gamma \vDash \alpha \text{ then } \Gamma \vdash \alpha. \qquad \text{(Nothing true is unprovable.)}$$

Theorem 5: Propositional natural deduction is both *sound* and *complete*.

$$\Gamma \vdash \alpha \quad \Longleftrightarrow \quad \Gamma \vDash \alpha$$

Proof: *(Soundness.)* By induction on the derivation. Each inference rule preserves validity: a small truth-table check per rule.

Soundness and Completeness [2]

(Completeness.) Build a derivation by induction on α's structure, case-splitting on which variables Γ forces. *(Kalmár, 1935.)* $\square$

Soundness: verified properties *actually hold*. Completeness: every true property *can* be proven.

Proof Strategies

Forward reasoning (bottom-up):

- Start from premises Γ
- Apply rules to derive new facts
- Continue until α is derived
- Risk: combinatorial explosion

Refutation (top-down):

- Assume $\neg\alpha$ together with Γ
- Derive a contradiction ($\perp$)
- Conclude that α must hold
- Advantage: *goal-directed* search

Refutation is the basis for *automated* reasoning: searching for a contradiction in $\Gamma \cup \{\neg\alpha\}$ is equivalent to a SAT problem.

Other proof systems: *Semantic tableaux* (truth trees) are another refutation-based method: negate the goal, decompose formulas, and check if every branch closes. Open branches yield counterexamples. We skip the details — resolution (below) is more directly relevant to SAT solving.

Bridge to automated reasoning: Resolution is the foundation of SAT solvers. Unlike natural deduction (designed for humans), resolution has a single rule — perfect for implementation.

Resolution

Resolution reduces propositional proof theory to a single inference rule, but requires *clausal form* (CNF). This makes it the natural foundation for automated theorem proving.

Definition 23 (Resolution Rule): Given two clauses containing complementary literals:

$$C_1 = (\ell_1 \vee \ldots \vee \ell_m \vee p) \quad \text{and} \quad C_2 = (\ell'_1 \vee \ldots \vee \ell'_k \vee \neg p)$$

derive the *resolvent*:

$$C = C_1 \otimes_p C_2 = \ell_1 \vee \ldots \vee \ell_m \vee \ell'_1 \vee \ldots \vee \ell'_k$$

The variable p is called the *pivot*.

Example: $(\neg P \vee Q)$ and $(P \vee R)$ resolve on P to produce $(Q \vee R)$.

Resolution is a *refutation* system: to prove $\Gamma \vDash \alpha$, convert $\Gamma \cup \{\neg\alpha\}$ to CNF and derive the *empty clause* $\square$.

Resolution Refutation: Example

Prove by resolution: $\{P, P \to Q\} \models Q$.

Add $\neg Q$ (negation of goal) and convert to clauses:

$$\begin{aligned} C_1 &= \{P\} && (\text{from } P) \\ C_2 &= \{\neg P, Q\} && (\text{from } P \to Q \equiv \neg P \vee Q) \\ C_3 &= \{\neg Q\} && (\text{negation of goal}) \end{aligned}$$

Derive:

$$\begin{aligned} C_4 &= \{Q\} && \text{resolve } C_1 \text{ and } C_2 \text{ on } P \\ C_5 &= \square && \text{resolve } C_4 \text{ and } C_3 \text{ on } Q \end{aligned}$$

The empty clause $\square$ is derived $\Rightarrow$ the original entailment holds. $\square$

Resolution Refutation: Example [2]

Theorem 6 (Completeness of Resolution): Resolution is *refutation-complete*: a set of clauses S is unsatisfiable if and only if the empty clause $\square$ can be derived from S by resolution.

Why this matters for SAT solving: Every CDCL solver is *implicitly* building a resolution proof. When a solver reports UNSAT, its learned clauses form a resolution refutation. This is how SAT solvers produce *verifiable certificates* of unsatisfiability — a crucial property for trustworthy verification.

Proof Systems: Comparison

System	Human-friendly	Automateable	For SAT	Certify UNSAT
Truth tables	Medium	Trivial	No	No
Natural Deduction	High	Low	No	No
Semantic Tableaux	Medium	Medium	Partial	Yes
Resolution	Low	High	Yes	Yes

Progression from left to right mirrors the course: human methods $\Rightarrow$ machine methods.

SAT solvers are *resolution engines* augmented with heuristics (VSIDS, restarts, phase saving).

First-Order Logic

Why First-Order Logic?

Propositional logic handles Boolean constraints well, and SAT solvers are remarkably efficient.
But *specifications* in formal methods require quantification:

```
def binary_search(arr, key):
    # Precondition: arr is sorted
    # Postcondition: returns index of key, or -1
```

The postcondition says: *"For all inputs $(\mathrm{arr}, \mathrm{key})$ where `arr` is sorted, if the function returns $i \geq 0$, then `arr[i] == key`; if it returns -1, then `key` is not in `arr`."*

In FOL: $\forall\ \mathrm{arr}, \mathrm{key}.\, \mathrm{Sorted}(\mathrm{arr}) \to (\mathrm{result} \geq 0 \to \mathrm{arr}[\mathrm{result}] = \mathrm{key}) \land (\mathrm{result} = -1 \to \forall i.\, \mathrm{arr}[i] \neq \mathrm{key})$

Why PL fails: Specifications quantify over *unbounded* or *infinite* domains:

- "All array indices are in bounds" $\longrightarrow$ quantifies over all integers i in range
- "No null pointer dereferences" $\longrightarrow$ quantifies over all pointers in the heap
- "The loop preserves the invariant" $\longrightarrow$ quantifies over all iterations

PL would need infinitely many propositions. FOL makes this expressible and *checkable*.

Why First-Order Logic? [2]

FOL adds three key ingredients:

- **Variables** ranging over objects in a domain (x, y, z, ...)
- **Quantifiers** ($\forall$, $\exists$) for generalization
- **Functions** and **predicates** giving structure to the domain

Example: "Every prime greater than 2 is odd":

$$\forall p. \left(\mathrm{Prime}(p) \land p > 2\right) \to \mathrm{Odd}(p)$$

One formula replaces infinitely many propositional checks.

FOL at a Glance

A first-order *signature* $\Sigma = \langle \mathcal{F}, \mathcal{R} \rangle$ declares function symbols (including constants) and relation symbols.

- *Terms* are built from variables and functions.
- *Formulas* combine terms via predicates, connectives, and quantifiers $\forall$, $\exists$.

Syntax example (arithmetic):

- Signature: $\Sigma = \langle \{0, S, +, \times\}, \{<, =\} \rangle$
- Formula: $\forall x.\, (x = 0 \vee \exists y.\, S(y) = x)$
- Meaning: “every natural number is either 0 or a successor”

Verification example (arrays):

- Signature: array operations `read`, `write`, `len`
- Formula: $\forall a, i, v.\, 0 \leq i < \mathrm{len}(a) \rightarrow \mathrm{read}(\mathrm{write}(a, i, v), i) = v$
- Meaning: “writing then reading gives back the value”

A *structure* (model) $\mathfrak{A}$ gives meaning to the symbols: a domain A, concrete functions, concrete relations. The *same* formula can be true in one structure and false in another — *validity* means truth in *all* structures.

- PL: 2^n truth assignments (finite) $\Rightarrow$ SAT is decidable (NP-complete)
- FOL: structures can have *infinite* domains $\Rightarrow$ validity is *undecidable* (Church–Turing, 1936)
- **FM response:** Restrict to *decidable fragments* $\Rightarrow$ SMT theories (linear arithmetic, arrays, *etc.*)

FOL: Key Concepts Preview

A variable x in $\forall x.\, \varphi$ is *bound*; a variable not in scope of any quantifier is *free*. A formula with no free variables is a *sentence* (has a definite truth value in each structure).

Concept	Meaning	Example
Free variables	Unquantified, act as parameters	$x + y > 0$ has free x, y
Bound variables	Under $\forall$ or $\exists$ scope	$\forall x.\, x + y > 0$ binds x, y free
Sentence	No free vars, definite truth value	$\forall x. \exists y.\, y > x$ (true in $\mathbb{Z}$)
Structure/Model	Domain + interpretation of symbols	$\mathbb{N}, \mathbb{Z}, \mathbb{R}$ with standard $+, \times, <$
Validity	True in *all* structures	$\forall x.\, x = x$ (valid)
Satisfiability	True in *some* structure	$\exists x.\, x \cdot x = 2$ (sat in $\mathbb{R}$, unsat in $\mathbb{Q}$)

Why FOL Matters: From Theory to Verification

Theoretical significance:

- Gödel completeness (1930): syntactic provability = semantic truth
- Church–Turing (1936): decision problem undecidable
- Compactness: infinite theories have finite character
- Incompleteness (1931): arithmetic has true unprovable statements

Practical impact on FM:

- *Dafny* specifications: `requires`, `ensures`, `invariant` are FOL formulas
- *SMT solvers* (Z3, CVC5): decide satisfiability in restricted FOL theories
- *Separation logic*: FOL + heap reasoning for pointer programs
- *Temporal logic*: FOL + time for reactive systems

Why FOL Matters: From Theory to Verification [2]

Example (Real verification scenario): Dafny method specification:

```
method Find(a: array<int>, key: int) returns (index: int)
  ensures index >= 0 ==> 0 <= index < a.Length && a[index] == key
  ensures index == -1 ==> forall i :: 0 <= i < a.Length ==> a[i] != key
```

The `forall i :: ...` is FOL quantification over integers. Z3 (the SMT solver behind Dafny) checks this by:

1. Translating to many-sorted FOL (separate sorts for `int`, `array<int>`)
2. Applying decision procedures for linear integer arithmetic + array theory
3. Returning SAT (code correct) or UNSAT + counterexample (bug found)

Automated Reasoning

From Logic to SAT Solving

The thread from logic to automated reasoning:

1. **Formulas** express constraints about system behavior and specifications.
2. **Normal forms** (especially CNF) provide the *input format* for SAT solvers.
3. **Equisatisfiability** (Tseitin) ensures compact, polynomial-size encodings.
4. **Resolution** is the *theoretical backbone* of DPLL and CDCL solvers.
5. **FOL and theories** motivate SMT solvers — SAT + specialized theory reasoning.
6. **Soundness and completeness** guarantee correctness of the entire pipeline.

Theorem 7 (Cook–Levin Theorem (1971)): The Boolean satisfiability problem (SAT) is **NP-complete**: every problem in NP can be reduced to SAT in polynomial time.

SAT is the *universal search problem*: if you can verify a solution efficiently, you can encode the search as a SAT instance. Modern CDCL solvers routinely handle formulas with *millions* of variables.

Next: SAT encodings, DPLL, CDCL, then FOL theories and SMT.

Key Takeaways

Propositional Logic:

- Syntax (WFFs) vs Semantics (truth values)
- SAT $\iff$ VALID duality via negation
- Equivalence laws $\Rightarrow$ NNF $\Rightarrow$ CNF
- Tseitin: polynomial equisatisfiable CNF

Proof Systems:

- Natural deduction: intro/elim symmetry
- Fitch notation for human-readable proofs
- Refutation: assume $\neg\alpha$, derive $\perp$
- Resolution: single rule on clausal form
- Cook–Levin: SAT is NP-complete

First-Order Logic (taste):

- Quantifiers ($\forall$, $\exists$) + predicates + functions
- Structures give meaning to symbols
- Same formula: true in one model, false in another
- Full treatment in Weeks 4–5

What's next:

- Week 3: SAT encodings, DPLL, CDCL
- Weeks 4–5: FOL deep dive + metatheorems
- Week 6: SMT = decidable fragments of FOL
- Weeks 9–12: Dafny (verification in practice)

The pipeline we build in this course:
Specification $\Rightarrow$ logical formula $\Rightarrow$ normal form $\Rightarrow$ solver (SAT/SMT) $\Rightarrow$ verdict.

Exercises: Propositional Logic

1. Show that $\{\to, \bot\}$ is a functionally complete set of connectives.
   Hint: Express $\neg p$ and $p \land q$ using only $\to$ and $\bot$.

2. Convert the formula $(P \to Q) \to R$ to:
   - NNF
   - CNF (using the distributive law)
   - Clausal form (using the Tseitin transformation)

3. For a chain of n biconditionals $p_1 \iff p_2 \iff \dots \iff p_{n+1}$:
   - How many clauses does the *equivalent* CNF have?
   - How many clauses does the *Tseitin* encoding produce? Explain the asymptotic difference.

4. Show that the satisfiability problem for DNF formulas is solvable in polynomial time.

5. $\star$ Show that *any* propositional proof system has a *tautology* whose shortest proof is exponential in the formula size (assuming NP $\neq$ co-NP). What does this imply about the possibility of efficient general-purpose provers?

Exercises: Proof Systems

1. Prove the following using natural deduction (Fitch notation):
   - $A \to B, \neg B \vdash \neg A$ *(modus tollens)*
   - $\vdash (A \to B) \lor (B \to A)$
   - $P \to \neg P \vdash \neg P$
   - $\neg(A \land B) \vdash \neg A \lor \neg B$ *(De Morgan, requires classical reasoning)*
2. Construct a semantic tableau to test the validity of: $P \to (Q \to R) \models (P \to Q) \to (P \to R)$
3. Use resolution refutation to show that $\{P \lor Q, \neg P \lor R, \neg Q \lor R\} \models R$.
4. $\star$ Prove that resolution is *not* polynomially bounded: the *pigeonhole principle* PHP_n^{n+1} (in CNF) requires exponentially long resolution proofs. *(State the formulation and explain why this matters for SAT solving.)*