Normal Form

Theory, Examples, and Proofs

Sara Sorahi

Introduction to the Theory of Computation

Normal Forms: Basic Idea

A **normal form** is a restricted, standardized way of writing mathematical objects (formulas, grammars, rules) *without changing their meaning*.

Key principle:

Expressive Power is Preserved

Only the *syntax* is restricted — not what can be expressed.

Why Do We Use Normal Forms?

Normal forms are useful because they:

- simplify proofs
- enable algorithms
- make comparison easier
- provide a common standard

Many fundamental results in computation assume inputs are in a normal form.

Conjunctive Normal Form (CNF)

A Boolean formula is in **CNF** if:

- it is a conjunction ($\wedge$) of clauses
- each clause is a disjunction ($\vee$) of literals
- a literal is a variable or its negation

Example:

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

Disjunctive Normal Form (DNF)

A Boolean formula is in **DNF** if:

- it is a disjunction ($\vee$) of terms
- each term is a conjunction ($\wedge$) of literals

Example:

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

Normal Forms for Grammars

In formal language theory, grammars are often converted into restricted formats called **grammar normal forms**.

These transformations:

- do not change the generated language
- simplify proofs and parsing algorithms

Chomsky Normal Form (CNF)

A context-free grammar is in **Chomsky Normal Form** if all rules are of the form:

$$A \rightarrow BC \quad \text{or} \quad A \rightarrow a$$

(with a limited exception for ϵ).

The restricted rule shapes enable efficient parsing algorithms (e.g. CYK).

Normal Forms: A Unifying Tool

Across logic and language theory, normal forms share:

- equivalence to unrestricted representations
- restricted syntax
- algorithmic usefulness

Normal forms balance **expressiveness** and **computability**.

Example: Converting to CNF (Step 1)

Convert the formula to CNF:

$$\varphi = (p \to q) \land (r \to (p \lor s)).$$

Step 1: Eliminate implications using

$$(a \to b) \equiv (\neg a \lor b).$$

So,

$$\varphi \equiv (\neg p \lor q) \land (\neg r \lor (p \lor s)).$$

Example: Converting to CNF (Already Done!)

We obtained:

$$(\neg p \lor q) \land (\neg r \lor (p \lor s)).$$

This is already in CNF:

- a conjunction ($\land$) of clauses
- each clause is a disjunction ($\lor$) of literals

Example: Distribution to Reach CNF

Convert to CNF:

$$\psi = (p \vee (q \wedge r)).$$

Use distribution:

$$x \vee (y \wedge z) \equiv (x \vee y) \wedge (x \vee z).$$

So,

$$\psi \equiv (p \vee q) \wedge (p \vee r).$$

Example: A Slightly Longer CNF Conversion

Convert to CNF:

$$\theta = (p \rightarrow (q \wedge r)).$$

Step 1: remove implication

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

Step 2: distribute

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

Proof Idea: Every Formula Has an Equivalent CNF

Claim. Every Boolean formula is equivalent to a CNF formula.

Proof strategy (constructive):

1. Eliminate $\to$ and $\leftrightarrow$ using equivalences.
2. Push negations inward using De Morgan + double negation:

$$\neg(a \wedge b) \equiv (\neg a \vee \neg b), \quad \neg(a \vee b) \equiv (\neg a \wedge \neg b).$$

3. Distribute $\vee$ over $\wedge$ until you obtain an AND of ORs.

Each step preserves equivalence, so the final CNF is equivalent to the original.

Note: Size Blow-up

CNF conversion by distribution can cause exponential growth.

Example pattern:

$$(x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee \cdots \vee (x_n \wedge y_n)$$

Distributing fully produces many clauses (can be exponential in n).

Example CFG (Not in Chomsky Normal Form)

Consider the grammar G generating $L = \{a^n b^n : n \geq 0\}$:

$$S \to aSb \mid \epsilon.$$

This is context-free, but not in CNF because:

- $S \to aSb$ has 3 symbols on the right
- $S \to \epsilon$ is an ϵ-rule

Step: Replace Terminals in Long Rules

Goal: make rules look like $A \to BC$ or $A \to a$.

Introduce new variables for terminals:

$$A \to a, \quad B \to b.$$

Rewrite:

$$S \to ASB \mid \epsilon.$$

Now terminals are isolated, but the right-hand side is still too long.

Step: Binarize Long Right-Hand Sides

We want to eliminate length-3 productions like:

$$S \to ASB.$$

Introduce a new variable C:

$$C \to SB.$$

Then replace the old rule by:

$$S \to AC \mid \epsilon, \qquad C \to SB.$$

Now the non-ϵ rules are of the form $A \to BC$.

Handling the ϵ-Rule Carefully

Chomsky Normal Form allows ϵ only via:

$$S \to \epsilon$$

and only if $\epsilon \in L(G)$.

In our language $a^n b^n$, ϵ is generated (when $n = 0$), so keeping $S \to \epsilon$ is acceptable under the standard CNF convention.

Theorem: Every CFG Has an Equivalent CNF Grammar

Theorem. For every context-free grammar G, there exists a grammar G' in Chomsky Normal Form such that:

$$L(G') = L(G)$$

(or $L(G') = L(G) \setminus \{\epsilon\}$, with a standard fix if needed).

Meaning: CNF is a restriction on rule *shape*, not on expressive power.

Proof Sketch (1): Cleaning Steps

Start from any CFG G.

Apply standard equivalence-preserving transformations:

1. Remove useless symbols (non-generating / unreachable variables).
2. Eliminate ϵ-productions (except possibly $S \to \epsilon$).
3. Eliminate unit productions ($A \to B$).

Each step produces a new grammar generating the same language (with the usual ϵ caveat).

Proof Sketch (2): Put Rules Into CNF Shape

After cleaning, remaining rules have the form:

$$A \to X_1 X_2 \cdots X_k \quad (k \geq 1),$$

where each X_i is a terminal or variable.

Two final transformations:

1. **Isolate terminals:** if a terminal appears in a long RHS, replace it by a new variable (e.g., $T_a \to a$).
2. **Binarize:** replace $A \to X_1 X_2 \cdots X_k$ $(k \geq 3)$ by introducing new variables so that all rules become binary.

Result: only $A \to BC$ and $A \to a$ remain (plus optional $S \to \epsilon$).

Why the Transformations Preserve the Language

Key invariant: Every new variable introduced is a “name” for a substring pattern.

Examples:

- Terminal isolation: $T_a \to a$ ensures T_a derives exactly a.
- Binarization: if $C \to X_2 \cdots X_k$, then $A \to X_1 C$ derives exactly what $A \to X_1 \cdots X_k$ derived before.

Thus, derivations in G correspond to derivations in G' and vice versa.

Proof Pattern: Equivalence via Mutual Simulation

To prove $L(G) = L(G')$, use two directions:

$$L(G) \subseteq L(G') \quad \text{and} \quad L(G') \subseteq L(G).$$

Method:

- Show how to convert any derivation in G into a derivation in G'.
- Show how to convert any derivation in G' into a derivation in G.

This is the standard "simulation" proof style for normal-form theorems.

Target Language: Simple Arithmetic Expressions

We start with a standard expression grammar (with precedence):

$$
\begin{aligned}
E &\to E + T \mid T \\
T &\to T * F \mid F \\
F &\to (E) \mid id
\end{aligned}
$$

Goal: Convert to **Chomsky Normal Form** (CNF):

$$A \to BC \quad \text{or} \quad A \to a$$

(and optionally $S \to \epsilon$ if needed).

Step 0: Add a New Start Symbol

Introduce a fresh start symbol S:

$$S \to E$$

This helps when removing ϵ-rules or unit productions later, and guarantees the start symbol never appears on a RHS in the final grammar.

Step 1: Remove Unit Productions

Unit productions are rules like $A \to B$. Here we have:

$$S \to E, \quad E \to T, \quad T \to F$$

Eliminate them by "copying" non-unit rules upward:

- Since $E \Rightarrow T$, add $E \to T * F \mid (E) \mid id$
- Since $T \Rightarrow F$, add $T \to (E) \mid id$
- Since $S \Rightarrow E$, let S inherit all E rules

Then delete unit productions.

Grammar After Unit-Removal (Readable Form)

After removing unit rules, we can use:

$$
\begin{aligned}
S &\rightarrow E + T \mid T * F \mid (E) \mid id \mid T \mid F \\
E &\rightarrow E + T \mid T * F \mid (E) \mid id \mid F \\
T &\rightarrow T * F \mid (E) \mid id \\
F &\rightarrow (E) \mid id
\end{aligned}
$$

We can simplify: many rules are redundant, but redundancy is OK. Next we enforce CNF *shape*.

Step 2: Isolate Terminals

CNF does not allow terminals mixed inside long RHSs like:

$$E \to E + T, \quad F \to (E)$$

Introduce variables for terminals:

$$P \to +, \quad M \to *, \quad L \to (, \quad R \to)$$

Now rewrite:

$$E \to EPT, \quad T \to TMF, \quad F \to LER \mid id$$

Step 3: Binarize Long Right-Hand Sides

We must eliminate length ≥ 3 RHSs:

$$E \to EPT, \qquad F \to LER$$

Introduce new variables to make everything binary:

$$X \to PT$$
$$E \to EX$$
$$Y \to ER$$
$$F \to LY$$

Now all nonterminal rules are of the form $A \to BC$.

Final CNF Grammar (One Clean Version)

A CNF grammar generating the same expression language:

$$
\begin{aligned}
S &\to EX \mid TM \mid LY \mid id \\
E &\to EX \mid TM \mid LY \mid id \\
T &\to TM \mid LY \mid id \\
F &\to LY \mid id
\end{aligned}
$$

$$
\begin{aligned}
X &\to PT \\
M &\to MF \\
Y &\to ER
\end{aligned}
$$

$$
P \to +, \quad M_t \to *, \quad L \to (, \quad R \to)
$$

Notes:

- All rules are $A \to BC$ or $A \to a$.

Why This Works (Derivation Correspondence)

Claim: Every derivation step in the original grammar can be simulated.

Examples:

- Original: $E \Rightarrow E + T$ becomes

$$E \Rightarrow EX \Rightarrow EPT \Rightarrow E + T$$

  (since $P \to +$).
- Original: $F \Rightarrow (E)$ becomes

$$F \Rightarrow LY \Rightarrow LER \Rightarrow (E)$$

New variables are just "helpers" that preserve the same strings.

SAT and CNF: The Core Connection

SAT asks: given a Boolean formula φ, is there an assignment that makes φ true?

In practice and in theory, formulas are often converted to **CNF**:

$$\varphi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$$

where each clause C_i is an OR of literals. This is the standard input format for SAT solvers.

3SAT: A Restricted Normal Form Problem

3SAT is SAT where every clause has exactly 3 literals:

$$(x \vee y \vee z) \wedge (\neg x \vee y \vee w) \wedge \cdots$$

3SAT is a **normal-form restriction**:

- fewer allowed shapes (only 3-literal clauses)
- but still captures the full computational difficulty of SAT

Theorem: SAT is in NP

Claim. SAT $\in$ NP.

Proof sketch. A certificate is a truth assignment to the variables. Given an assignment, we can evaluate φ in time polynomial in $|\varphi|$ (by computing each gate/clause once). So SAT has polynomial-time verification.

Theorem: 3SAT is NP-Complete (What It Means)

Theorem. 3SAT is NP-complete.

Two parts:

- (**Membership**) $3SAT \in NP$ (same verification idea).
- (**Hardness**) Every language in NP reduces to 3SAT in polynomial time.

So 3SAT is a canonical "hardest" problem in NP.

Cook–Levin (Big Picture Proof Sketch)

Cook–Levin Theorem: SAT is NP-complete.

Idea: Encode a polynomial-time NTM computation as a Boolean formula.

- Variables represent symbols/states at positions in a computation tableau.
- Clauses enforce:
  - valid start configuration
  - valid transitions (local consistency)
  - an accepting configuration occurs

The formula is satisfiable *iff* the machine accepts the input.

From SAT to 3SAT (Normal-Form Reduction)

Claim. SAT $\leq_p$ 3SAT.

Proof sketch: Convert an arbitrary CNF clause to 3-literal clauses.

- If a clause has 1 or 2 literals, pad by repeating literals:

$$(x) \;\equiv\; (x \vee x \vee x), \qquad (x \vee y) \;\equiv\; (x \vee y \vee y)$$

- If a clause has $k > 3$ literals:

$$(l_1 \vee l_2 \vee \cdots \vee l_k)$$

  introduce new variables $y_1, \ldots, y_{k-3}$ and replace by:

$$(l_1 \vee l_2 \vee y_1) \wedge (\neg y_1 \vee l_3 \vee y_2) \wedge \cdots \wedge (\neg y_{k-3} \vee l_{k-1} \vee l_k)$$

Satisfiability is preserved, and the size grows only linearly.

Takeaway: Normal Forms Power NP-Completeness

Normal forms are not just “pretty formatting”: they enable clean reductions and standard problem statements.

Examples:

- CNF makes SAT solver inputs uniform.
- 3CNF makes hardness proofs modular and reusable.
- Tableau-style encodings rely on strict local constraints (a kind of normal form).