

LEAN 4 TACTIC CHEATSHEET

Last updated: October 29, 2024

If a tactic is not recognized, write `import Mathlib.Tactic` at the top of your file.

Logical symbol	Appears in goal	Appears in hypothesis
\forall (for all)	<code>intro x</code>	<code>apply h</code> or <code>specialize h x</code>
\rightarrow (implies)	<code>intro h</code>	<code>apply h</code> or <code>specialize h1 h2</code>
\neg (not)	<code>intro h</code>	<code>apply h</code> or <code>contradiction</code>
\leftrightarrow (if and only if)	<code>constructor</code>	<code>rw [h]</code> or <code>rw [← h]</code> or <code>apply h.1</code> or <code>apply h.2</code>
\wedge (and)	<code>constructor</code>	<code>obtain ⟨h1, h2⟩ := h</code>
\exists (there exists)	<code>use x</code>	<code>obtain ⟨x, hx⟩ := h</code>
\vee (or)	<code>left</code> or <code>right</code>	<code>obtain h1 h2 := h</code>
$a = b$ (equality)	<code>rfl</code> or <code>ext</code>	<code>rw [h]</code> or <code>rw [← h]</code>
True	<code>trivial</code>	—
False	—	<code>contradiction</code>

Tactic	Effect
Applying Lemmas	
<code>exact <i>expr</i></code>	prove the current goal exactly by <i>expr</i> .
<code>apply <i>expr</i></code>	prove the current goal by applying <i>expr</i> to some arguments.
<code>refine <i>expr</i></code>	like <code>exact</code> , but <i>expr</i> can contain <code>?</code> that will be turned into a new goal.
<code>convert <i>expr</i></code>	prove the goal by showing that it is equal to the type of <i>expr</i> .
Context manipulation	
<code>have h : <i>prop</i> := <i>expr</i></code>	add a new hypothesis <i>h</i> of type <i>prop</i> . ▲ Do not use for data!
<code>have h : <i>prop</i> := by <i>tac</i></code>	add hypothesis <i>h</i> after proving it using tactics. ▲ Do not use for data!
<code>set x : <i>type</i> := <i>expr</i></code>	add an abbreviation <i>x</i> with value <i>expr</i> .
<code>clear h</code>	remove hypothesis <i>h</i> from the context.
<code>rename_i x h</code>	rename the last inaccessible names with the given names.
<code>show <i>expr</i></code>	Replaces the goal by <i>expr</i> , if they are equal by definition.
Rewriting and simplifying	
<code>rw [<i>expr</i>]</code>	in the goal, replace (all occurrences of) the left-hand side of <i>expr</i> by its right-hand side. <i>expr</i> must be an equality, iff statement or definition.
<code>rw [←<i>expr</i>]</code>	... rewrites using <i>expr</i> from right-to-left.
<code>rw [<i>expr</i>] at h</code>	... rewrite in hypothesis <i>h</i> .
<code>nth_rw n [<i>expr</i>]</code>	rewrite only the <i>n</i> -th occurrence of the rewrite rule <i>expr</i> .
<code>simp</code>	simplify the goal using all lemmas tagged <code>@[simp]</code> and basic reductions.
<code>simp at h</code>	... simplify in hypothesis <i>h</i> .
<code>simp [* , <i>expr</i>]</code>	... also simplify with all hypotheses and <i>expr</i> .
<code>simp only [<i>expr</i>]</code>	... only simplify with <i>expr</i> and basic reductions (not with simp-lemmas).
<code>simp?</code>	... let Lean speed up <code>simp</code> by specifying which lemmas were used.
<code>simp_rw [<i>expr</i>1 , ...]</code>	like <code>rw</code> , but uses <code>simp only</code> at each step.
<code>simp_all</code>	repeatedly simplify the goal and all hypothesis using all hypotheses.
<code>norm_num</code>	simplify numerical expressions by calculating.
<code>norm_cast</code>	simplify the expression by moving casts (\uparrow) outwards.
<code>push_cast</code>	push casts inwards.
<code>conv => <i>conv-tac</i></code>	apply rewrite rules to only part of the goal. Use <code>congr</code> , <code>skip</code> , <code>ext</code> , <code>lhs</code> , <code>rhs</code> , ... to navigate to the desired subexpression. See TPIL.
<code>change <i>expr</i></code>	Change the current goal to <i>expr</i> , if it is equal by definition to the current goal.

<code>split_ifs</code>	case split on every occurrence of <code>if h then <i>expr</i> else <i>expr</i></code> in the goal.
<hr/>	
Reasoning with equalities, inequalities, and other relations	
<code>calc a = b := by tac</code>	perform a calculation
<code> _ ≤ c := by tac</code>	💡 after writing “ <code>calc _</code> ” Lean can generate a basic <code>calc</code> -block for you.
<code> _ < d := by tac</code>	💡 after a <code>by</code> shift-click on a subterm in the goal to create a new step.
<code>rfl</code>	prove the current goal by reflexivity.
<code>symm</code>	swap a symmetric relation.
<code>trans <i>expr</i></code>	split a transitive relation into two parts with <code><i>expr</i></code> in the middle.
<code>subst h</code>	if <code>h</code> equates a variable with a value, substitute the value for the variable.
<code>ext</code>	prove an equality in a specified type (e.g. functions).
<code>apply_fun <i>expr</i> at h</code>	apply <code><i>expr</i></code> to both sides of the (in)equality <code>h</code> .
<code>linear_combination</code>	prove an equality by specifying it as a linear combination of hypotheses.
<code>congr</code>	prove an equality using congruence rules.
<code>gcongr</code>	prove an inequality using congruence rules.
<code>positivity</code>	prove goals of the form $0 < x$, $0 \leq x$ and $x \neq 0$.
<code>bound</code>	prove inequalities based on the expression structure.
<code>omega</code>	solve linear arithmetic problems over \mathbb{N} or \mathbb{Z} .
<code>linarith</code>	prove linear (in)equalities from the hypotheses.
<code>nlinarith</code>	stronger variant of <code>linarith</code> that can solve some nonlinear inequalities.
<hr/>	
Reasoning techniques	
<code>exfalso</code>	replace the current goal by <code>False</code> .
<code>by_contra h</code>	proof by contradiction; adds the negation of the goal as hypothesis <code>h</code> .
<code>push_neg</code> or <code>push_neg at h</code>	push negations into quantifiers and connectives in the goal (or in <code>h</code>).
<code>by_cases h : <i>prop</i></code>	case-split on <code><i>prop</i></code> .
<code>induction n with</code> <code> zero => <i>tac</i></code> <code> succ n ih => <i>tac</i></code>	prove a goal by induction on <code>n</code> . 💡 after writing “ <code>induction n</code> ” Lean can generate the cases for you.
<code>choose f h using <i>expr</i></code>	extract a function from a forall-exists statement <code><i>expr</i></code> .
<code>lift n to <i>type</i> using h</code>	lifts a variable to <code><i>type</i></code> (e.g. \mathbb{N}) using side-condition <code>h</code> .
<code>zify / qify / rify</code>	shift an (in)equality to \mathbb{Z} / \mathbb{Q} / \mathbb{R} .
<hr/>	
Searching	
<code>exact?</code>	search for a single lemma that closes the goal using the current hypotheses.
<code>apply?</code>	gives a list of lemmas that can apply to the current goal.
<code>rw?</code>	gives a list of lemmas that can be used to rewrite the current goal.
<code>have? using h1, h2</code>	try to find facts that can be concluded by using both <code>h1</code> and <code>h2</code> .
<code>hint</code>	run a few common tactics on the goal, reporting which one succeeded.
<hr/>	
General automation	
<code>ring / noncomm_ring / module</code> <code>field_simp / abel / group</code>	prove the goal by using the axioms of a commutative ring / ring / module / field / abelian group / group.
<code>aesop</code>	simplify the goal, and use various techniques to prove the goal.
<code>tauto</code>	prove logical tautologies.
<code>decide</code>	run a decision procedure to prove the goal (if it is decidable).
<hr/>	
Operations on goals/tactics	
<code>swap</code>	swap the first two goals.
<code>pick_goal n</code>	move goal <code>n</code> to the front.
<code>all_goals <i>tac</i></code>	run <code><i>tac</i></code> to all goals.
<code>try <i>tac</i></code>	run <code><i>tac</i></code> only if it succeeds.
<code><i>tac1</i>; <i>tac2</i></code>	run <code><i>tac1</i></code> and then <code><i>tac2</i></code> (same as putting them on separate lines).
<code><i>tac1</i> <;> <i>tac2</i></code>	run <code><i>tac1</i></code> and then <code><i>tac2</i></code> on all goals generated by <code><i>tac1</i></code> .
<code>sorry</code>	admit the current goal.

Domain-specific tactics

<code>fin_cases h</code>	split a hypothesis <code>h</code> into finitely many cases.
<code>interval_cases n</code>	if split the goal into cases for each of the possible values for <code>n</code> .
<code>compute_degree</code>	prove (in)equalities about the degree of a polynomial
<code>monicity</code>	prove that a polynomial is monic
<code>fun_prop</code>	prove that a function satisfies a property (continuity, measurability, ...).
<code>measurability</code>	prove that a set or function is measurable.
<code>filter_upwards [h1, h2]</code>	Show that an <code>Eventually</code> goal follows from the given hypotheses.
<code>slice_lhs, slice_rhs</code>	Focus on a part of a composition in a category. See the source code for some other category theory tactics.

Usage note

This is a quick overview of the most common tactics in Lean with only a short description. To learn more about a tactic and to learn its precise syntax or variants, consult its docstring or use `#help tactic tac`. This list is not complete, and various tactics are intentionally left out.

Some useful commands (Some of these also work as tactics)

<code>#loogle query</code>	🌐 use Loogle! to find declarations.
<code>#leansearch "query."</code>	🌐 use LeanSearch to find declarations.
<code>#exit</code>	don't compile code after this command.
<code>#lint</code>	run linters to find common mistakes in the code <i>above</i> this command.
<code>#where</code>	print current opened namespaces, universes, variables and options.
<code>#help tactic tac</code>	find information about <i>tac</i> .
<code>#help category</code>	list all tactics/commands/attributes/options/notations.

Legend

- 💡 describes a code action for this tactic.
- 🌐 requires internet access.