

# Lean Quick Reference

Jeremy Avigad, Leonardo de Moura, Soonho Kong

Version 0a2d14e, updated at 2016-06-07 10:06:27 +0100

## 1 Quick Reference

### 1.1 Displaying Information

---

check <expr>	: check the type of an expression
eval <expr>	: evaluate expression
print <id>	: print information about <id>
print notation	: display all notation
print notation <tokens>	: display notation using any of the tokens
print axioms	: display assumed axioms
print options	: display options set by user or emacs mode
print prefix <namespace>	: display all declarations in the namespace
print coercions	: display all coercions
print coercions <source>	: display only the coercions from <source>
print classes	: display all classes
print instances <class name>	: display all instances of the given class
print fields <structure>	: display all "fields" of a structure
print metaclasses	: show kinds of metadata stored in a namespace
help	: display all available commands
help options	: display all available options

---

### 1.2 Common Options

You can change an option by typing `set_option <option> <value>`. The `<option>` field supports TAB-completion. You can see an explanation of all options using `help options`.

---

pp.implicit	: display implicit arguments
pp.universes	: display universe variables
pp.coercions	: show coercions
pp.notation	: display output using defined notations
pp.abbreviations	: display output using defined abbreviations
pp.full_names	: use full names for identifiers
pp.all	: disable notations, implicit arguments, full names, universe parameters and coercions
pp.beta	: beta reduce terms before displaying them
pp.max_depth	: maximum expression depth
pp.max_steps	: maximum steps for printing expression
pp.private_names	: show internal name assigned to private definitions and theorems

---

```
pp.metavar_args    : show arguments to metavariables
pp.numerals        : print output as numerals
```

---

### 1.3 Attributes

These can generally be declared with a `definition` or `theorem`, or using the `attribute` or `local attribute` commands.

Example: `local attribute nat.add nat.mul [reducible]`.

---

```
reducible          : unfold at any time during elaboration if necessary
quasireducible     : unfold during higher order unification,
                    : but not during type class resolution
semireducible      : unfold when performance is not critical
irreducible        : avoid unfolding during elaboration
coercion           : use as a coercion between types
class              : type class declaration
instance           : type class instance
priority <num>     : add a priority to an instance or notation
parsing-only       : use notation only for input
unfold <num>       : if the argument at position <num> is marked with [constructor]
                    : unfold this and that argument (for iota reduction)
constructor        : see unfold <num>
unfold-full        : unfold definition when fully applied
recursor           : user-defined recursor/eliminator, used for the induction tactic
recursor <num>     : user-defined non-dependent recursor/eliminator
                    : where <num> is the position of the major premise
refl               : reflexivity lemma, used for calc-expressions, tactics and simplifier
symm               : symmetry lemma, used for calc-expressions, tactics and simplifier
trans              : transitivity lemma, used for calc-expressions, tactics and simplifier
subst              : substitution lemma, used for calc-expressions and simplifier
```

---

### 1.4 Proof Elements

#### 1.4.1 Term Mode

---

```
take, assume       : syntactic sugar for lambda
let                : introduce local definitions
have               : introduce auxiliary fact (opaque, in the body)
assert             : like "have", but visible to tactics
show               : make result type explicit
suffices           : show that the goal follows from this fact
obtain ..., from   : destruct structures such as exists, sigma, ...
match ... with     : introduce proof or definition by cases
proof ... qed      : introduce a proof or definition block, elaborated separately
```

---

The keywords `have` and `assert` can be anonymous, which is to say, they can be used without giving a label to the hypothesis. The corresponding element of the context can then be referred to using the keyword `this` until another anonymous element is introduced, or by enclosing the assertion in backticks. To avoid a syntactic ambiguity, the keyword `suppose` is used instead of `assume` to introduce an anonymous assumption.

One can also use anonymous binders (like `lambda`, `take`, `obtain`, etc.) by enclosing the type in backticks, as in `λ `nat`, `nat` + 1`. This introduces a variable of the given type in the context with a hidden name.

### 1.4.2 Tactic Mode

At any point in a proof or definition you can switch to tactic mode and apply tactics to finish that part of the proof or definition.

---

<code>begin ... end</code>	: enter tactic mode, and blocking mechanism within tactic mode
<code>{ ... }</code>	: blocking mechanism within tactic mode
<code>by ...</code>	: enter tactic mode, can only execute a single tactic
<code>begin+; by+</code>	: same as <code>=begin=</code> and <code>=by=</code> , but make local results available
<code>have</code>	: as in term mode (enters term mode), but visible to tactics
<code>show</code>	: as in term mode (enters term mode)
<code>match ... with</code>	: as in term mode (enters term mode)
<code>let</code>	: introduce abbreviation (not visible in the context)
<code>note</code>	: introduce local fact (opaque, in the body)

---

Normally, entering tactic mode will make declarations in the local context given by “have”-expressions unavailable. The annotations `begin+` and `by+` make all these declarations available.

## 1.5 Sectioning Mechanisms

---

<code>namespace &lt;id&gt; ... end &lt;id&gt;</code>	: begin / end namespace
<code>section ... end</code>	: begin / end section
<code>section &lt;id&gt; .... end &lt;id&gt;</code>	: begin / end section
<code>variable (var : type)</code>	: introduce variable where needed
<code>variable {var : type}</code>	: introduce implicit variable where needed
<code>variable {{var : type}}</code>	: introduce implicit variable where needed, which is not maximally inserted
<code>variable [var : type]</code>	: introduce class inference variable where needed
<code>variable {var} (var) [var]</code>	: change the bracket type of an existing variable
<code>parameter</code>	: introduce variable, fixed within the section
<code>include</code>	: include variable in subsequent definitions
<code>omit</code>	: undo “include”

---

## 1.6 Tactics

We say a tactic is more “aggressive” when it uses a more expensive (and complete) unification algorithm, and/or unfolds more aggressively definitions.

### 1.6.1 General tactics

---

<code>apply &lt;expr&gt;</code>	: apply a theorem to the goal, create subgoals for non-dependent premises
<code>fapply &lt;expr&gt;</code>	: like <code>apply</code> , but create subgoals also for dependent premises that were

---

	not assigned by unification procedure
eapply <expr>	: like apply, but used for applying recursor-like definitions
exact <expr>	: apply and close goal, or fail
rexact <expr>	: relaxed (and more expensive) version of exact (this will fully elaborate <expr> before trying to match it to the goal)
refine <expr>	: like exact, but creates subgoals for unresolved subgoals
intro <ids>	: introduce multiple variables or hypotheses
intros <ids>	: same as intro <ids>
intro	: let Lean choose a name
intros	: introduce variables as long as the goal reduces to a function type and let Lean choose the names
rename <id> <id>	: rename a variable or hypothesis
generalize <expr>	: generalize an expression
clear <ids>	: remove variables or hypotheses
revert <ids>	: move variables or hypotheses into the goal
assumption	: try to close a goal with something in the context
eassumption	: a more aggressive ("expensive") form of assumption

---

## 1.6.2 Equational reasoning

---

esimp	: simplify expressions (by evaluation/normalization) in goal
esimp at <id>	: simplify hypothesis in context
esimp at *	: simplify everything
esimp [<ids>]	: unfold definitions and simplify expressions in goal
esimp [<ids>] at <id>	: unfold definitions and simplify hypothesis in context
esimp [<ids>] at *	: unfold definitions and simplify everything
unfold <id>	: similar to (esimp <id>)
fold <expr>	: unfolds <expr>, search for convertible term in the goal, and replace it with <expr>
beta	: beta reduce goal
whnf	: put goal in weak head normal form
change <expr>	: change the goal to <expr> if it is convertible to <expr>
rewrite <rule>	: apply a rewrite rule (see below)
rewrite [<rules>]	: apply a sequence of rewrite rules (see below)
krewrite	: using keyed rewriting, matches any subterm with the same head as the rewrite rule
xrewrite	: a more aggressive form of rewrite
subst <id>	: substitute a variable defined in the context, and clear hypothesis and variable
substvars	: substitute all variables in the context

---

**Rewrite rules** You can combine rewrite rules from different groups in the following order, starting with the innermost:

---

e	: match left-hand-side of equation e to a goal subterm, then replace every occurrence with right-hand-side
{p}e	: apply e only where pattern p (which may contain placeholders) matches

---

<code>n t</code>	: apply <code>t</code> exactly <code>n</code> times
<code>n&gt;t</code>	: apply <code>t</code> at most <code>n</code> times
<code>*t</code>	: apply <code>t</code> zero or more times (up to <code>rewriter.max_iter</code> )
<code>+t</code>	: apply <code>t</code> one or more times
<code>-t</code>	: apply <code>t</code> in reverse direction
<code>↑id</code>	: unfold <code>id</code>
<code>↑[ids]</code>	: unfold <code>ids</code>
<code>↓id</code>	: fold <code>id</code>
<code>►expr</code>	: reduce goal to expression <code>expr</code>
<code>►*</code>	: equivalent to <code>esimp</code>
<code>t at {i, ...}</code>	: apply <code>t</code> only at numbered occurrences
<code>t at -{i, ...}</code>	: apply <code>t</code> only at all but the numbered occurrences
<code>t at H</code>	: apply <code>t</code> at hypothesis <code>H</code>
<code>t at H {i, ...}</code>	: apply <code>t</code> only at numbered occurrences in <code>H</code>
<code>t at H -{i, ...}</code>	: apply <code>t</code> only at all but the numbered occurrences in <code>H</code>
<code>t at * ⊢</code>	: apply <code>t</code> at all hypotheses
<code>t at *</code>	: apply <code>t</code> at the goal and all hypotheses

---

### 1.6.3 Induction and cases

---

<code>cases &lt;expr&gt;</code>	: decompose an element of an inductive type
<code>cases &lt;expr&gt; with &lt;ids&gt;</code>	: name newly introduced variables as specified by <code>&lt;ids&gt;</code>
<code>induction &lt;expr&gt; (with &lt;ids&gt;)</code>	: use induction
<code>induction &lt;expr&gt; using &lt;def&gt;</code>	: use the definition <code>&lt;def&gt;</code> to apply induction
<code>constructor</code>	: construct an element of an inductive type by applying the first constructor that succeeds
<code>constructor &lt;i&gt;</code>	: construct an element of an inductive type by applying the <code>i</code> th-constructor
<code>fconstructor</code>	: construct an element of an inductive type by (fapplying) the first constructor that succeeds
<code>fconstructor &lt;i&gt;</code>	: construct an element of an inductive type by (fapplying) the <code>i</code> th-constructor
<code>injection &lt;id&gt; (with &lt;ids&gt;)</code>	: use injectivity of constructors at specified hypothesis
<code>split</code>	: equivalent to (constructor 1), only applicable to inductive datatypes with a single constructor (e.g. and introduction)
<code>left</code>	: equivalent to (constructor 1), only applicable to inductive datatypes with two constructors (e.g. left or introduction)
<code>right</code>	: equivalent to (constructor 2), only applicable to inductive datatypes with two constructors (e.g. right or introduction)
<code>existsi &lt;expr&gt;</code>	: similar to (constructor 1) but we can provide an argument, useful for performing exists/sigma introduction

---

### 1.6.4 Special-purpose tactics

---

<code>contradiction</code>	: close contradictory goal
<code>exfalso</code>	: implements the "ex falso quodlibet" logical principle
<code>congruence</code>	: solve goals of the form $(f\ a_1 \dots a_n = f'\ b_1 \dots b_n)$ by congruence
<code>reflexivity</code>	: reflexivity of equality (or any relation marked with attribute <code>refl</code> )
<code>symmetry</code>	: symmetry of equality (or any relation marked with attribute <code>symm</code> )
<code>transitivity &lt;expr&gt;</code>	: transitivity of equality (or any relation marked with attribute <code>trans</code> )
<code>trivial</code>	: apply true introduction

---

---

### 1.6.5 Combinators

---

<code>and_then &lt;tac1&gt; &lt;tac2&gt;</code>	(notation: <code>&lt;tac1&gt; ; &lt;tac2&gt;</code> )
	: execute <code>&lt;tac1&gt;</code> and then execute <code>&lt;tac2&gt;</code> , backtracking when needed (aka sequential composition)
<code>or_else &lt;tac1&gt; &lt;tac2&gt;</code>	(notation: <code>(&lt;tac1&gt;   &lt;tac2&gt;)</code> )
	: execute <code>&lt;tac1&gt;</code> if it fails, execute <code>&lt;tac2&gt;</code>
<code>&lt;tac1&gt;: &lt;tac2&gt;</code>	: apply <code>&lt;tac1&gt;</code> and then apply <code>&lt;tac2&gt;</code> to all subgoals generated by <code>&lt;tac1&gt;</code>
<code>par &lt;tac1&gt; &lt;tac2&gt;</code>	: execute <code>&lt;tac1&gt;</code> and <code>&lt;tac2&gt;</code> in parallel
<code>fixpoint (fun t, &lt;tac&gt;)</code>	: fixpoint tactic, <code>&lt;tac&gt;</code> may refer to <code>t</code>
<code>try &lt;tac&gt;</code>	: execute <code>&lt;tac&gt;</code> , if it fails do nothing
<code>repeat &lt;tac&gt;</code>	: repeat <code>&lt;tac&gt;</code> zero or more times (until it fails)
<code>repeat1 &lt;tac&gt;</code>	: like (repeat <code>&lt;tac&gt;</code> ), but fails if <code>&lt;tac&gt;</code> does not succeed at least once
<code>at_most &lt;num&gt; &lt;tac&gt;</code>	: like (repeat <code>&lt;tac&gt;</code> ), but execute <code>&lt;tac&gt;</code> at most <code>&lt;num&gt;</code> times
<code>do &lt;num&gt; &lt;tac&gt;</code>	: execute <code>&lt;tac&gt;</code> exactly <code>&lt;num&gt;</code> times
<code>determ &lt;tac&gt;</code>	: discard all but the first proof state produced by <code>&lt;tac&gt;</code>
<code>discard &lt;tac&gt; &lt;num&gt;</code>	: discard the first <code>&lt;num&gt;</code> proof-states produced by <code>&lt;tac&gt;</code>

---

### 1.6.6 Goal management

---

<code>focus_at &lt;tac&gt; &lt;i&gt;</code>	: execute <code>&lt;tac&gt;</code> to the <code>i</code> th-goal, and fail if it is not solved
<code>focus &lt;tac&gt;</code>	: equivalent to (focus_at <code>&lt;tac&gt;</code> 0)
<code>rotate_left &lt;num&gt;</code>	: rotate goals to the left <code>&lt;num&gt;</code> times
<code>rotate_right &lt;num&gt;</code>	: rotate goals to the right <code>&lt;num&gt;</code> times
<code>rotate &lt;num&gt;</code>	: equivalent to (rotate_left <code>&lt;num&gt;</code> )
<code>all_goals &lt;tac&gt;</code>	: execute <code>&lt;tac&gt;</code> to all goals in the current proof state
<code>fail</code>	: tactic that always fails
<code>id</code>	: tactic that does nothing and always succeeds
<code>now</code>	: fail if there are unsolved goals

---

### 1.6.7 Information and debugging

---

<code>state</code>	: display the current proof state
<code>check_expr &lt;expr&gt;</code>	: display the type of the given expression in the current goal
<code>trace &lt;string&gt;</code>	: display the current string
<code>with_options [&lt;options&gt;] &lt;tac&gt;</code>	: execute a single tactic with different options ( <code>&lt;options&gt;</code> is a comma-separated list)

---

## 1.7 Emacs Lean-mode commands

### 1.7.1 Flycheck commands

---

<code>C-c ! n</code>	: next error
<code>C-c ! p</code>	: previous error
<code>C-c ! l</code>	: list errors
<code>C-c C-x</code>	: execute Lean (in stand-alone mode)

---

## 1.7.2 Lean-specific commands

---

C-c C-k	: show how to enter unicode symbol
C-c C-o	: set Lean options
C-c C-e	: execute Lean command
C-c C-r	: restart Lean process
C-c C-p	: print the definition of the identifier under the cursor in a new buffer
C-c C-g	: show the current goal at a line of a tactic proof, in a new buffer
C-c C-f	: fill a placeholder by the printed term in the minibuffer. Note: the elaborator might need more information to correctly infer the implicit arguments of this term

---

## 1.8 Unicode Symbols

This section lists some of the Unicode symbols that are used in the Lean library, their ASCII equivalents, and the keystrokes that can be used to enter them in the Emacs Lean mode.

### 1.8.1 Logical symbols

Unicode	Ascii	Emacs
<hr/>		
true		
false		
$\neg$	not	\not, \neg
$\wedge$	/\	\and
$\vee$	\	\or
$\rightarrow$	->	\to, \r, \implies
$\leftrightarrow$	<->	\iff, \lr
$\forall$	forall	\all
$\exists$	exists	\ex
$\lambda$	fun	\l, \fun
$\neq$	~=	\ne

### 1.8.2 Types

$\Pi$	Pi	<code>\Pi</code>
$\rightarrow$	->	<code>\to, \r, \implies</code>
$\Sigma$	Sigma	<code>\S, \Sigma</code>
$\times$	prod	<code>\times</code>
	sum	<code>\union, \u+, \uplus</code>
$\mathbb{N}$	nat	<code>\nat</code>
$\mathbb{Z}$	int	<code>\int</code>
$\mathbb{Q}$	rat	<code>\rat</code>
$\mathbb{R}$	real	<code>\real</code>

When you open the namespaces `prod` and `sum`, you can use `*` and `+` for the types `prod` and `sum` respectively. To avoid overwriting notation, these have to have the same precedence as the arithmetic operations. If you don't need to use notation for the arithmetic operations, you can obtain lower-precedence versions by opening the namespaces `low_precedence_times` and `low_precedence_plus` respectively.

### 1.8.3 Greek letters

Unicode	Emacs
$\alpha$	<code>\alpha</code>
$\beta$	<code>\beta</code>
$\gamma$	<code>\gamma</code>
...	...

### 1.8.4 Equality proofs (open `eq.ops`)

Unicode	Ascii	Emacs
$\overset{-1}{\cdot}$	<code>eq.symm</code>	<code>\sy, \inv, \-1</code>
$\cdot$	<code>eq.trans</code>	<code>\tr</code>
$\blacktriangleright$	<code>eq.subst</code>	<code>\t</code>

### 1.8.5 Symbols for the rewrite tactic

Unicode	Ascii	Emacs
$\uparrow$	<code>^</code>	<code>\u</code>
$\downarrow$	<code>&lt;d</code>	<code>\d</code>



### 1.8.6 Brackets

Unicode	Ascii	Emacs
$\lfloor t \rfloor$	?(t)	<code>\c11 t \clr</code>
$\{ \{ t \} \}$	$\{\{t\}\}$	<code>\{\{ t \}\}</code>
$\langle t \rangle$		<code>\&lt; t \&gt;</code>
$t$		<code>\&lt;&lt; t \&gt;&gt;</code>

### 1.8.7 Set theory

Unicode	Ascii	Emacs
$\in$	mem	<code>\in</code>
$\notin$		<code>\nin</code>
$\cap$	inter	<code>\i</code>
$\cup$	union	<code>\un</code>
$\subseteq$	subseteq	<code>\subeq</code>

### 1.8.8 Binary relations

Unicode	Ascii	Emacs
$\leq$	$\leq$	<code>\le</code>
$\geq$	$\geq$	<code>\ge</code>
$ $	dvd	<code>\ </code>
$\equiv$		<code>\equiv</code>
$\approx$		<code>\eq</code>

### 1.8.9 Binary operations

Unicode	Ascii	Emacs
$\circ$	comp	<code>\comp</code>