

Lean Quick Reference

Jeremy Avigad, Leonardo de Moura, Soonho Kong

Version 46dcada, updated at 2015-08-02 00:34:43 -0400

1 Quick Reference

1.1 Displaying Information

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

1.2 Setting Options

You can also see a list of all options by typing `set_option` with a space at the end and pressing TAB.

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

```
pp.numerals      : print output as numerals
pp.abbreviations : show abbreviations
```

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]`.

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

1.4 Proof Elements

1.4.1 Term Mode

<code>take, assume</code>	: syntactic sugar for <code>lambda</code>
<code>let</code>	: introduce local definitions
<code>have</code>	: introduce auxiliary fact (opaque, in the body)
<code>assert</code>	: like "have", but visible to tactics
<code>show</code>	: make result type explicit
<code>obtain ..., from</code>	: destruct structures such as <code>exists</code> , <code>sigma</code> , ...
<code>match ... with</code>	: introduce proof or definition by cases
<code>proof ... qed</code>	: 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

<code>begin ... end</code>	: enter tactic mode, and blocking mechanism within tactic mode
<code>{ ... }</code>	: blocking mechanism within tactic mode
<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 local fact (opaque, in the body)

1.5 Sectioning Mechanisms

<code>namespace <id> ... end <id></code>	: begin / end namespace
<code>section ... end</code>	: begin / end section
<code>section <id> end <id></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 class inference variable where needed
<code>variable {var} (var) [var]</code>	: change the bracket of an existing variable
<code>parameter</code>	: introduce variable, fixed within the section
<code>include</code>	: include variable in subsequence definitions
<code>omit</code>	: undoes "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 <expr></code>	: apply a theorem to the goal, create subgoals for non-dependent premises
<code>fapply <expr></code>	: like apply, but create subgoals also for dependent premises that were not assigned by unification procedure
<code>eapply <expr></code>	: like apply, but used for applying recursor-like definitions
<code>exact <expr></code>	: apply and close goal, or fail
<code>rexact <expr></code>	: relaxed (and more expensive) version of exact
<code>refine <expr></code>	: like exact, but creates subgoals for unresolved subgoals
<code>intro <ids></code>	: introduce multiple variables or hypotheses
<code>intros <ids></code>	: same as intro <ids>
<code>intro</code>	: let Lean choose a name
<code>intros</code>	: introduce variables as long as the goal reduces to a function type and let Lean choose the names
<code>rename <id> <id></code>	: rename a variable or hypothesis
<code>generalize <expr></code>	: generalize an expression

clear <ids>	: clear 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 <expr>	: apply a rewrite rule
rewrite <expr-list>	: apply a sequence of rewrites
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

1.6.3 Induction and cases

cases <expr>	: decompose an element of an inductive type
cases <expr> with <ids>	: names newly introduces variables as specified by <ids>
induction <expr> (with <ids>)	: use induction
induction <expr> using <def>	: uses the definition <def> to apply induction
constructor	: construct an element of an inductive type by applying the first constructor that succeeds
constructor <i>	: construct an element of an inductive type by applying the ith-constructor
fconstructor	: construct an element of an inductive type by (fapply)ing the first constructor that succeeds
fconstructor <i>	: construct an element of an inductive type by (fapply)ing the ith-constructor
injectivity	: use injectivity of constructors
split	: equivalent to (constructor 1), only applicable to inductive datatypes with a single constructor (e.g. and introduction)
left	: equivalent to (constructor 1), only applicable to inductive datatypes with two constructors (e.g. left or introduction)
right	: equivalent to (constructor 2), only applicable to inductive datatypes with two constructors (e.g. right or introduction)
existsi <expr>	: similar to (constructor 1) but we can provide an argument, useful for performing exists/sigma introduction

1.6.4 Special-purpose tactics

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

1.6.5 Combinators

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

1.6.6 Goal management

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

1.6.7 Information and debugging

state	: display the current proof state
check_expr <expr>	: display the type of the given expression in the current goal
trace <string>	: display the current string

1.7 Emacs Lean-mode commands

1.7.1 Flycheck commands

C-c ! n	: next error
C-c ! p	: previous error
C-c ! l	: list errors
C-c C-x	: 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	: experimental feature to 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	<code>\not, \neg</code>
\wedge	<code>/\</code>	<code>\and</code>
\vee	<code>\ </code>	<code>\or</code>
\rightarrow	<code>-></code>	<code>\to, \r, \implies</code>
\leftrightarrow	<code><-></code>	<code>\iff, \lr</code>
\forall	forall	<code>\all</code>
\exists	exists	<code>\ex</code>
λ	fun	<code>\lam, \fun</code>
\neq	<code>~=</code>	<code>\ne</code>

1.8.2 Types

Π	Pi	<code>\Pi</code>
\rightarrow	<code>-></code>	<code>\to, \r, \implies</code>
Σ	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
α	<code>\alpha</code>
β	<code>\beta</code>
γ	<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><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>\< t \></code>
t		<code>\<< t \>></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>