Lean cheat sheet

1. Constructs

Symbol	Notation	Type name	Destruction*	Construction**	Computational equivalent
$\wedge$	\and	And	`obtain ⟨x, y⟩`	`constructor`	Product type
$\vee$	\or	Or	`obtain x|y`	`left`, `right`	Sum type
$\to$	\->	–[1]	`apply`	`intro`	Function type
$\leftrightarrow$	\iff	Iff	`obtain ⟨x, y⟩`	`constructor`	Bijection
False	False	False	`destruct`	-[2]	Empty type
True	True	True	`destruct`	`constructor`	Unit type
$\forall$	\forall	–[1]	`apply`	`intro`	Dependent function type
$\exists$	\exists	Exists	`choose` `obtain ⟨a, ha⟩`	`exists`	Dependent pair (type x proof)
$=$	=	Eq	`rw [x]`	`rfl`	Identity type
$\neq$	\neq	–[3]	`apply`	`intro`	- (nothing interesting)

*Using `match` is an option for all non-fundamental types
**Direct use of constructors is an option for all non-fundamental types

[1]Fundamental type
[2]False does not have constructors
[3]$x \neq y$ is short for `(x=y)` $\to$ False

2. More tactics (also see the tactic reference and the tactic language chapters in the Lean Language Reference)

Name	Effect	Example/comment
`apply`	applies a lemma or a theorem	`apply h`
`rw`	replaces (sub)terms with something equivalent	`rw [h1, ← h2]`
`simp`	automatically applies `rw` steps	`simp [def1, def2]`; `@[simp]` tag
`unfold`	replaces a term with its definition	`unfold def`
`have`	introduces a subproof	`have h : 0 = 0 := by rfl`
`let`	introduces a hypothesis	`let e := 8 + c`
`constructor`	applies the first compatible constructor	may pick the wrong constructor!
`injection`	uses injectivity of inductive constructors	`a::b = c::d` iff `a = c` and `b = d`
`exfalso`	replaces the goal with `False`	useful for applying inequalities
`cases`/`match`	proof by cases	same syntax as `match`
`induction`	proof by induction (stronger)	`induction h with`

`apply` vs. `rw`: `apply` replaces a hypothesis/conclusion, `rw` swaps one of it subterms for something equivalent.

`apply`: if the conclusion is `C`, applying a hypothesis of the form `A → B → C` (can be read as "give me a proof of `A` and a proof of `B` and I'll give you a proof of `C`") leaves you with two branches: one for proving `A`, one for proving `B`.

3. Commands

`#print x`: see the definition of `x`
`#check x`: see the type of `x`
`#eval x`: compute `x`

`rw`: if you know that `a = b`, then you can turn the goal/hypothesis `a + c = a - b` into `b + c = b - b`.

4. Forward vs. backward reasoning

Forward reasoning: from the hypotheses to the conclusion
Backward reasoning: from the conclusion to the hypotheses (default in Lean)
`at` keyword for forward reasoning; e.g., `apply h at h2` or `rw [h] at h2`.

5. Syntax reference

```
def name
  (arg1 arg2: type1) (arg3: type2)
: type :=
  match arg2 with
  | case1 c_arg1 c_arg2 => c_arg1 + c_arg2
  | case2 => other_function arg3 arg1
```

Note that `cases` and `induction` uses the same syntax as `match`.

```
lemma modus_ponens
  {A B: Prop} (H: A → B) : A → B
:= by
  intro a; apply H; apply a
```

```
structure Coords where
  x: ℕ
  y: ℕ
def p := {x := 2, y := 8}

inductive MyNat where
| zero
| succ (pred: MyNat)
open MyNat
def one := succ zero
```

`open` brings the constructors of the type in the global namespace.

`(H: X)` vs. `{H: X}`: the first needs to be passed explicitly, the second can be deduced by Lean (or passed explictly through the `@` syntax, e.g., `@modus_ponens (even a) (even (S (S a))) even_plus_two_even`).