Lean

PENG

November 4, 2026

Contents

1 Introduction **1**

2 Dependent Type Theory **1**

3 Function Abstraction **2**

4 Variable and Sections **3**

5 Theorem **4**

6 Propositional Logic **4**

A Tools **6**

1 Introduction

Definition 1.1 (lean)**.** *Lean is an expression-oriented functional language based on dependent type theory.*

2 Dependent Type Theory

Definition 2.1 (definition)**.** *Declare a new named constant object.*

```
def c : type := expr
-- auxiliary commands begin with # symbol
#check c   -- check type
#eval expr -- evaluate expression
#print c   -- view definition

-- a name defined using def is semireducible
def T := type
def c : T := ... -- error

-- local definition
-- strong means of abbreviation
#check let c := ...; expr -- replace c in expr
```

Definition 2.2 (type)**.** *Every expression has an associated type.*

```
Nat
#check Nat.succ
#check Nat.add
Int
Float
String
Bool -- true or false
```

```
List -- List type
#check List.nil
#check []
#check [1, 2, 3]

#check List.cons
#check List.cons 1 []
#check 1 :: 2 :: 3 :: []

#check List.map
#check List.map func [1, 2, 3]
#check [1, 2, 3].map func

Type -- abbreviation for Type 0
-- infinite hierarchy of types
-- larger universe of types contains lower universe of types as elements
-- Type : Type 1
-- Type n : Type n + 1
universe u -- polymorphic constant
Type u

(expr : type) -- specify the type of an expression
```

Definition 2.3 (simple type theory)**.** *Build new types out of others.*

```
type → type        -- denotes the type of function
type → type → type -- equivalent to type → (type → type)
-- arrows associate to the right in type expressions

type × type    -- denotes the type of Cartesian product
Prod type type -- alternative notation
(m, n)         -- denotes the ordered pair of m and n which is of type type × type
(m, n).1       -- first component
(m, n).2       -- second component
```

Definition 2.4 (dependent type theory)**.** *Dependent type theory extends simple type theory that types themselves are objects which also have a type.*

Example 2.1 (polymorphic)**.** *A function is polymorphic that takes the type to be the (first) argument.*

```
def cons (α : Type) (a : α) (as : List α) : List α :=
  List.cons a as

#check cons Nat
#check cons

-- dependent function type or dependent arrow type
-- the type of the value returned by function depends on its input
-- dependent Cartesian product
def f (α : Type u) (β : α → Type v) (a : α) (b : β a) : (a : α) × β a := ⟨a, b⟩
def g (α : Type u) (β : α → Type v) (a : α) (b : β a) : Σ a : α, β a := Sigma.mk a b
```

3 Function Abstraction

Definition 3.1 (lambda function)**.** *Creating a function from another expression is a process called lambda abstraction.*

```
#check fun (param_1 : type) ... => expr
-- lambda function is an object of type type → ...
```

```
-- λ and fun are the same

#eval (λ (param_1 : type) ... ) arg_1

-- declare a new named object
def f : type → ... := λ (param_1 : type) ... => expr
-- move lambda-abstracted variables to the left of the colon
-- with input parameters and returning type
def f (param_1 : type) ... : type := expr

#eval f arg_1 ... arg_n -- call function
```

Example 3.1. `def compose (α β γ : Type) (g : β → γ) (f : α → β) (x : α) : γ := g (f x)`

Definition 3.2 (implicit argument)**.** *When a function takes an argument that can generally be inferred from context, Lean allows you to specify that argument by default be left implicit.*

```
universe u
def ident {α : Type u} (x : α) := x

#check (ident)
#check @ident -- denotes the same function with all the arguments made explicit
```

4 Variable and Sections

Definition 4.1 (variable)**.** *Instruct Lean to insert the declared variables as bounded variables in definitions that refer to them by name.*

```
variable (T : type) ...

variable (α β γ : Type)
variable (g : β → γ) (f : α → β)
variable (x : α)
def compose : γ := g (f x)
```

Definition 4.2 (section)**.** *Limit the scope of a variable.*

```
section MySec
    ...
end MySec

universe u
section
  variable {α : Type u}
  variable (x : α)
  def ident := x
end
```

Definition 4.3 (namespace)**.** *Group definitions.*

```
namespace MyNamespace
    def c : type := ...
end MyNamespace

#check MyNamespace.c
open MyNamespace -- use shorter names
#check c
```

5 Theorem

Definition 5.1 (proposition-as-types paradigm)**.** *Implication between proposition p and q corresponds to having a function that takes an element of p to an element of q. A proof of $p \to q$ uses "the fact that p is true" to obtain "the fact that q is true."*

$$p \to q \tag{5.1}$$

To formally express a mathematical assertion in the language of dependent type theory, we need to exhibit a term `p : Prop`. To prove that assertion, we need to exhibit a term `t : p`.

Definition 5.2 (theorem)**.** *A theorem is a named proof term whose type is a proposition.*

```
-- named theorem
-- proposition is the type of its proof
-- proving the theorem is the same as defining an element of the associated type
theorem t : proposition := ...

theorem t {p q : Prop} : p → q → p :=
    -- proof of p → (q → p) assumes p and q are true to establish that the conclusion p is true
    -- if p, then if r, then p
    fun hp : p =>  -- hp is a proof of proposition p
    fun hq : q =>  -- hq is a proof of proposition q
    show p from hp -- specify the type of final term hp to improve the clarity of a proof

#print t
theorem t : ∀ {p q : Prop}, p → q → p :=
  fun {p q : Prop} (hp : p) (hq : q) => hp

-- move the lambda-abstracted variables to the left of the colon
variable {p q : Prop}
theorem t (hp : p) (hq : q) : p := hp

-- use theorem as function application
axiom hp : p -- postulate the existence of an element of the given type
theorem t2 : q → p := t hp

-- anonymous theorem
example
```

6 Propositional Logic

Definition 6.1 (proposition logical connectives)**.**

```
True
False
¬ -- Not
∧ -- And
∨ -- Or
→ -- introduce
↔ -- Iff
```

Definition 6.2 (conjunction)**.**

```
variable (p q : Prop)

-- And.intro : p ∧ q
example (hp : p) (hq : q) : p ∧ q := And.intro hp hq -- ⟨hp, hq⟩

-- And.left : p
example (h : p ∧ q) : q := And.right h
```

```
-- And.right : q
example (h : p ∧ q) : p := And.left h

-- p ∧ q → q ∧ p
example (h : p ∧ q) : q ∧ p :=
    And.intro (And.right h) (And.left h)

-- anonymous constructor notation
example (h : p ∧ q) : q ∧ p ∧ q :=
  ⟨h.right, h.left, h.right⟩
```

Definition 6.3 (disjunction)**.**

```
variable (p q r : Prop)

-- Or.intro_left : p ∨ q
example (hp : p) : p ∨ q := Or.intro_left q hp

-- Or.intro_right : p ∨ q
example (hq : q) : p ∨ q := Or.intro_right p hq

-- Or.elim -- if p ∨ q, then if p → r, then if q → r, then r

-- p ∨ q → q ∨ p
example (h : p ∨ q) : q ∨ p :=
  Or.elim h
    (fun hp : p =>
    show q ∨ p from Or.intro_right q hp)
    (fun hq : q =>
    show q ∨ p from Or.intro_left p hq)
```

Definition 6.4 (negation)**.**

$$p \to q \Longrightarrow \neg q \to \neg p \tag{6.1}$$

```
variable (p q : Prop)

example (hpq : p → q) (hnq : ¬q) : ¬p :=
    -- ¬p and ¬q are notations for p → False and q → False, respectively
    fun hp : p =>
    Show False from hnq (hpq hp)
```

Definition 6.5 (logical equivalence)**.**

```
variable (p q : Prop)

-- Iff.intro : p ↔ q
-- Iff.mp    : p → q
-- Iff.mpr   : q → p

theorem and_swap : p ∧ q ↔ q ∧ p :=
  Iff.intro
    (fun h : p ∧ q =>
     show q ∧ p from And.intro (And.right h) (And.left h))
    (fun h : q ∧ p =>
     show p ∧ q from And.intro (And.right h) (And.left h))

variable (h : p ∧ q)
example : q ∧ p := Iff.mp (and_swap p q) h -- (and.swap p q).mp : p ∧ q → q ∧ p
```

7 Auxiliary Subgoals

Definition 7.1 (have constructor)**.**

A Tools

Definition A.1 (comment)**.**

```
/- comment blocks can be nested -/
-- line comment
```