Lean

PENG

Created: 20260910, Version: 20260910

Contents

1 Introduction **1**
1.1 Overview . . . 1

2 Basics **2**
2.1 Types . . . 2
2.2 Functions and Definitions . . . 2
2.2.1 Defining Functions . . . 2
2.2.2 Defining Types . . . 3
2.3 Structures . . . 3
2.4 Datatypes and Patterns . . . 3
2.4.1 Pattern Matching . . . 4
2.4.2 Recursive Functions . . . 4
2.5 Polymorphism . . . 5
2.5.1 Linked List Datatype . . . 5
2.5.2 Implicit Arguments . . . 5
2.5.3 More Built-In Structures and Inductive Datatypes . . . 6

3 Interlude: Propositions, Proofs, and Indexing **7**
3.1 Propositions and Proofs . . . 7
3.2 Tactics . . . 7

1 Introduction

References:

- `https://lean-lang.org/doc/reference/latest/`
- `https://leanprover-community.github.io/mathematics_in_lean/`
- `https://lean-lang.org/functional_programming_in_lean/`

Lean is an expression tactic functional language based on **dependent type theory**.

1.1 Overview

Every expression has a type (printed by `#check`).

Definition 1.1 (mathematical object)**.**

```
#check 2 + 2 -- Nat

def f (x : Nat) :=
  x+3

#check f -- Nat -> Nat
```

Definition 1.2 (mathematical statement: Prop)**.**

```
#check 2 + 2 = 4 -- Prop

def FermatLastTheorem :=
  \all x y z n : Nat, n > 2 \and x * y *z \neq 0 \to x^n + y^n \neq z^n

#check FermatLastTheorem -- Prop
```

Definition 1.3 (proof of the proposition)**.**

```
theorem easy : 2 + 2 = 4 :=
  rfl

#check easy -- 2 + 2 = 4

theorem hard : FermatLastTheorem :=
  sorry

#check hard -- FermatLastTheorem
```

Construct an expression of type `FermatLastTheorem` and make Lean accepts it as a term of that type.

2 Basics

2.1 Types

Definition 2.1 (type)**.**

- `Nat`*: default type for non-negative integer literals*
- `Int`
- `Float`
- `String`
- `Bool`
- `List`*:* `List type`
- `Type`*: describes other types, so* `Nat`*,* `String`*,* `List`*, etc. all have type* `Type`

Example 2.2.

```
#eval (1 - 2 : Nat) -- 0
#check (1 - 2 : Int) -- check the type of an expression
```

2.2 Functions and Definitions

Definition 2.3 (definition)**.**

```
def var : type := val
```

Example 2.4.

```
def lean : String := "Lean"
#eval String.append "hello " lean
```

2.2.1 Defining Functions

Definition 2.5 (function)**.**

```
def func (arg1 : type) ... (argn : type) : type := body
```

Example 2.6.

```
def maximum (n : Nat) (m : Nat) : Nat := if n < m then m else n
#eval maximum 1 5
```

Definition 2.7 (currying)**.** `Nat -> (Nat -> Nat)`

2.2.2 Defining Types

Definition 2.8 (Type)**.**

```
def Str : Type := String -- overloaded
abbrev N : Type := Nat -- unfolded
```

Definition 2.9 (overloaded)**.**

Definition 2.10 (reducible)**.** *Definitions that are to be unfolded are called reducible.*

2.3 Structures

Definition 2.11 (structure)**.**

```
structure Point where
x : Float
y : Float
def origin : Point := {x:=0.0,y:=0.0}
#eval origin.x
#check {x:=0.0,y:=0.0:Point}
```

Definition 2.12 (updating structures)**.** `def zeroX (p : Point) := {p with x:=0}`

Definition 2.13 (constructor)**.**

```
structure Point where
point::
x : Float
y : Float
#check Point.point 1.0 2.0
```

Definition 2.14 (accessor)**.**

```
#eval Point.x {x:=1.0,y:=2.0}
#check {x:=1.0,y:=2.0 : Point}.x
-- TARGET.func ARG1 ... ARGn
#eval "hello".append " Lean"
def Point.modifyBoth (f : Float -> Float) (p : Point) : Point := {x:=f p.x,y:=f p.y}
#eval {x:=3,y:=4 : Point}.modifyBoth Float.floor
```

2.4 Datatypes and Patterns

Definition 2.15 (datatype)**.**

- `sum`
- `recursive`
- `inductive` *: recursive sum type*

Example 2.16 (inductive)**.**

```
inductive Nat where
| zero : Nat
| succ (n : Nat) : Nat
#eval Nat.succ (Nat.succ Nat.zero)
```

2.4.1 Pattern Matching

Definition 2.17 (pattern matching)**.**

```
def isZero (n:Nat) : Bool :=
match n with
| Nat.zero => true
| Nat.succ k => false
#eval isZero Nat.zero

def pred (n : Nat) : Nat :=
match n with
| Nat.zero => Nat.zero
| Nat.succ k => k
#eval pred 5

def depth (p:Point3D) :Float :=
match p with
| {x:=h,y:=w,z:=d} => d
#eval depth {x:=1,y:=2,z:=3}
```

2.4.2 Recursive Functions

Definition 2.18 (recursive definition)**.** *Definitions that refer to the name being defined are called recursive definition.*

Definition 2.19 (structural recursion)**.**

```
def even (n:Nat) : Bool :=
match n with
| Nat.zero => true
| Nat.succ k => not (even k)
#eval even 2

-- addition = interated Nat.succ
def plus (n:Nat) (k:Nat) : Nat :=
match k with
| Nat.zero => n
| Nat.succ k' => Nat.succ (plus n k')
#eval plus 1 2

-- multiplication = interated addition
def times (n:Nat) (k:Nat) : Nat :=
match k with
| Nat.zero => 0
| Nat.succ k' => plus n (times n k')
#eval times 1 2

-- subtraction = interated predecessor
def minus (n:Nat) (k:Nat) : Nat :=
match k with
| Nat.zero => n
| Nat.succ k' => Nat.pred (minus n k')
#eval minus 2 1
```

Remark 2.20. *Lean ensures by default that every recursive function will eventually reach a base case to avoid accidental infinite loops*

Example 2.21.

```
def div (n:Nat) (k:Nat) : Nat :=
```

```
if n < k then
0
else Nat.succ (div (n-k) k)
```

2.5 Polymorphism

Definition 2.22 (polymorphism)**.** *polymorphism refers to datatypes and definitions that take types as arguments, where* ***types*** *are ordinary* ***expressions*** *in Lean.*

Example 2.23.

```
-- structure take types as arguments
structure PPoint (\alpha : Type) where
x : \alpha
y : \alpha
def natOrigin : PPoint Nat := {x:=Nat.zero,y:=Nat.zero}
#check natOrigin

-- definition take types as arguments
def replaceX (\alpha : Type) (point : PPoint \alpha) (newX : \alpha) : PPoint \alpha :=
{point with x:=newX}
#check replaceX
#eval replaceX Nat natOrigin 2

-- argument is a sign
inductive Sign where
| pos
| neg
def posOrNegThree (s : Sign) :
match s with | Sign.pos => Nat | Sign.neg => Int :=
match s with
| Sign.pos => (3:Nat)
| Sign.neg => (-3:Int)
#check posOrNegThree Sign.pos
```

Remark 2.24. *It is customary to use Greek letters to name type arguments.*

2.5.1 Linked List Datatype

Definition 2.25 (List)**.** *A* `List a` *can be built with either* `nil` *or* `cons`*. A list that contains n entries contains n* `cons` *constructors, the last of which has* `nil` *as its tail.*

```
inductive List (\alpha : Type) where
| nil : List \alpha
| cons : \alpha -> List \alpha -> List \alpha
def primesUnder10 : List Nat := [2,3,5,7]
#eval List.cons 2 (List.cons 3 (List.cons 5 (List.cons 7 List.nil)))

-- functions that consume Lists
def length (\alpha : Type) (xs : List \alpha) : Nat :=
match xs with
| List.nil => Nat.zero -- [] for List.nil
| List.cons y ys => Nat.succ (length \alpha ys) -- y :: ys for List.cons
#eval length String ["hello", "lean"]
```

2.5.2 Implicit Arguments

Definition 2.26 (implicit type)**.**

```
def length {\alpha : Type} (xs : List \alpha) : Nat :=
match xs with
```

```
| [] => Nat.zero
| y :: ys => Nat.succ (length ys)
#eval length ["hello", "lean"]
-- List.length
#eval ["hello", "lean"].length
```

2.5.3 More Built-In Structures and Inductive Datatypes

Definition 2.27 (option)**.** *List is empty or has a first entry.*

```
inductive Option (\alpha : Type) : Type where
| none : Option \alpha
| some (val : \alpha) : Option \alpha

def List.head? {\alpha : Type} (xs : List \alpha) : Option \alpha :=
match xs with
| [] => none
| y :: _ => some y
#eval [1,2,3].head?
#eval ([] : List Int).head?
```

Remark 2.28. *Type was unavailable because the empty list does not have any entries from which the type can be found.*

name convention of suffixes for operations:

- ? for operations that might fail
- ! for crash due to invalid input
- D for default value when the operation would otherwise fail

Definition 2.29 (Prod)**.** *Constructs a pair.*

```
structure Prod (\alpha : Type) (\beta : Type) : Type where
fst : \alpha
snd : \beta
-- def fives : Prod String Int := {fst:="five",snd:=5}
-- def fives : String \times Int := {fst:="five",snd:=5}
def fives : String \times Int := ("five",5)
```

Definition 2.30 (Sum)**.** *Allow a choice between values of two different types.*

```
inductive Sum (\alpha : Type) (\beta : Type) : Type where
| inl : \alpha -> Sum \alpha \beta
| inr : \beta -> Sum \alpha \beta

-- def PetName : Type := Sum String String
def PetName : Type := String \oplus String

def animals : List PetName :=
[Sum.inl "Spot", Sum.inr "Tiger",Sum.inl "Fifi",Sum.inl "Rex",Sum.inr "Floor"]

def howManyDogs (pets : List PetName) : Nat :=
match pets with
| [] => 0
| Sum.inl _ :: morePets => howManyDogs morePets + 1
| Sum.inr _ :: morePets => howManyDogs morePets

#eval howManyDogs animals
```

Definition 2.31 (Unit)**.**

Definition 2.32 (Empty)**.**

3 Interlude: Propositions, Proofs, and Indexing

Definition 3.1 (arrays and lists)**.**

```
def lists : List type := [ele1, ..., elen]
#eval lists[0] -- or lists[k]
```

3.1 Propositions and Proofs

Definition 3.2 (proposition)**.** *A proposition is a statement that can be true or wrong.*

Definition 3.3 (proof)**.** *A proof is a convincing argument that a proposition is true.*

Definition 3.4 (theorem)**.**

3.2 Tactics