# Lezione 4: Logica classica e quantificatori

Stefano M. Nicoletti

## 1 Lezione 4: Logica classica e quantificatori

### 1.1 Registrazione della lezione

Registrazione video della Lezione 4

### 1.2 Scopo della lezione

Nella Lezione 3 abbiamo collegato le tattiche di Lean alle regole di introduzione e di eliminazione della deduzione naturale. In questa lezione estendiamo quel quadro in due direzioni. Prima introduciamo la riduzione all'assurdo come regola classica. Poi passiamo dalla logica proposizionale alla logica dei predicati, imparando a rappresentare oggetti, proprietà, funzioni, relazioni e quantificatori.

Il percorso della lezione è il seguente:

- distinguere le regole costruttive dai principi classici e applicare la riduzione all'assurdo;
- costruire un semplice linguaggio del primo ordine in Lean;
- formalizzare enunciati contenenti ∀ ed ∃.

### 1.3 Dalle regole costruttive alla logica classica

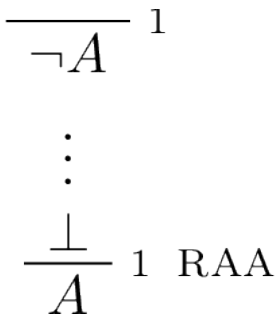
Le regole viste finora hanno una lettura costruttiva diretta: ogni dimostrazione specifica come costruire il termine richiesto. Per dimostrare una congiunzione costruiamo entrambe le componenti; per dimostrare un'implicazione costruiamo una funzione.

Nella logica costruttiva di base di Lean, il terzo escluso `A ∨ ¬A` e l'eliminazione della doppia negazione `¬¬A → A` non sono disponibili automaticamente. Lean permette però di usare principi classici in modo locale ed esplicito (Fonte: Theorem Proving in Lean 4, Classical Logic).

Questa distinzione riguarda gli strumenti ammessi nella dimostrazione. Non significa che Lean vieti la logica classica ma il ricorso a un principio classico deve rimanere esplicitamente visibile nel termine costruito.

### 1.4 Riduzione all'assurdo

La riduzione all'assurdo classica permette di dimostrare `A` mostrando che l'assunzione `¬A` conduce a `False`. In Lean la applichiamo con `Classical.byContradiction`.



*Riduzione all'assurdo, RAA*

Lo schema operativo è il seguente:

1. applichiamo la riduzione all'assurdo; 2. introduciamo l'assunzione `¬A`; 3. usiamo le assunzioni disponibili per ottenere `False`; 4. usiamo esattamente questa contraddizione.

Vogliamo dimostrare che piove, date due assunzioni: se non piove, non prendo l'ombrello; prendo l'ombrello. Dopo aver applicato la riduzione all'assurdo, il nuovo compito è mostrare che supporre che non piova produce una contraddizione.

```
theorem lecture04_reductio_ad_absurdum
    (Piove PrendoOmbrello : Prop)
    (hNonPioveNonOmbrello : ¬Piove → ¬PrendoOmbrello)
    (hPrendoOmbrello : PrendoOmbrello) :
    Piove := by
  apply Classical.byContradiction
  intro hNonPiove
  have hNonPrendoOmbrello :=
    hNonPioveNonOmbrello hNonPiove
  have hContraddizione :=
    hNonPrendoOmbrello hPrendoOmbrello
  exact hContraddizione
```

La prima assunzione trasforma `¬Piove` in `¬PrendoOmbrello`. Quest'ultimo fatto è una funzione da `PrendoOmbrello` a `False`; applicandola all'assunzione `hPrendoOmbrello`, otteniamo la contraddizione richiesta.

## 1.5 Un limite della logica proposizionale

Nella logica proposizionale trattiamo ogni enunciato come una proposizione completa. Potremmo usare:

- `P` per «Ipazia è una filosofa»;
- `Q` per «Hannah Arendt è una filosofa»;
- `R` per «Ipazia è curiosa».

Questa rappresentazione nasconde però la struttura interna degli enunciati. Non mostra che `P` e `Q` attribuiscono la stessa proprietà a due oggetti diversi. Non permette nemmeno di esprimere direttamente «ogni filosofa è curiosa» oppure «esiste una filosofa curiosa».

La logica dei predicati rende visibili gli oggetti di cui parliamo e le proprietà o relazioni che attribuiamo loro.

## 1.6 Il dominio del discorso

Ogni formalizzazione parte da un *dominio*: la collezione degli oggetti che stiamo considerando. In questa lezione usiamo un solo tipo generale:

```
variable (Cosa : Type)
```

`Cosa` rappresenta il dominio del discorso. Un termine `x : Cosa` è un elemento del dominio. La scelta di un unico dominio corrisponde alla presentazione a un solo dominio tipica della logica del primo ordine (Fonte: Logic and Proof, First Order Logic in Lean).

Possiamo poi introdurre termini che interpretiamo come oggetti nominati:

```
variable (Cosa : Type)
variable (ipazia hannah roma atene : Cosa)
```

Queste dichiarazioni non affermano ancora alcuna proposizione. Stabiliscono soltanto che `ipazia`, `hannah`, `roma` e `atene` sono termini del dominio.

## 1.7 Predicati: proprietà degli oggetti

Un predicato con un argomento prende un oggetto del dominio e produce una proposizione. Per esempio:

```
variable (Cosa : Type)
variable (Persona Citta Filosofa Curiosa : Cosa → Prop)
```

`Filosofa : Cosa → Prop` è una proprietà che possiamo applicare a oggetti diversi:

```
variable (Cosa : Type)
variable (ipazia hannah : Cosa)
variable (Filosofa : Cosa → Prop)

#check Filosofa ipazia  -- Filosofa ipazia : Prop
#check Filosofa hannah  -- Filosofa hannah : Prop
```

Le espressioni `Filosofa ipazia` e `Filosofa hannah` sono proposizioni distinte, ma condividono lo stesso predicato. La struttura comune non viene più nascosta dietro lettere proposizionali indipendenti.

| Linguaggio naturale | Lean |
|---|---|
| Ipazia è curiosa | `Curiosa ipazia` |
| Hannah Arendt è curiosa | `Curiosa hannah` |
| Io è una luna di Giove | `LunaDiGiove io` |
| Roma è piovosa | `Piove roma` |

## 1.8 Funzioni: costruire nuovi termini

Una funzione prende uno o più oggetti del dominio e restituisce un oggetto del dominio. Per esempio:

```
variable (Cosa : Type)
variable (madreDi : Cosa → Cosa)
```

`madreDi ipazia : Cosa` è un nuovo termine, che interpretiamo come «la madre di Ipazia». Poiché il risultato ha ancora tipo `Cosa`, possiamo usarlo come argomento di un predicato:

```
variable (Cosa : Type)
variable (ipazia : Cosa)
variable (madreDi : Cosa → Cosa)
variable (Curiosa : Cosa → Prop)

#check madreDi ipazia
#check Curiosa (madreDi ipazia)
```

Le funzioni permettono di costruire termini progressivamente più complessi:

| Termine | Lettura |
|---|---|
| `ipazia` | Ipazia |
| `madreDi ipazia` | la madre di Ipazia |
| `madreDi (madreDi ipazia)` | la madre della madre di Ipazia |

Nella logica del primo ordine una funzione è totale e restituisce un solo risultato per ogni argomento. Una funzione restituisce un oggetto; un predicato restituisce una proposizione.

## 1.9 Relazioni tra oggetti

Un predicato con due o più argomenti esprime una relazione. In Lean possiamo dichiarare:

```
variable (Cosa : Type)
variable (Conosce Visita : Cosa → Cosa → Prop)
```

Otteniamo così proposizioni come:

```
variable (Cosa : Type)
variable (ipazia hannah atene : Cosa)
variable (Conosce Visita : Cosa → Cosa → Prop)

#check Conosce ipazia hannah
#check Visita ipazia atene
```

L'ordine degli argomenti è significativo. `Conosce ipazia hannah` rappresenta «Ipazia conosce Hannah Arendt»; `Conosce hannah ipazia` rappresenta invece «Hannah Arendt conosce Ipazia».

## 1.10 Parametri locali e assunzioni

La parola chiave `variable` introduce parametri locali. Dentro una `section` possiamo assumere temporaneamente un dominio e un vocabolario:

```
section LinguaggioDeiPredicati

variable (Cosa : Type)
variable (ipazia hannah : Cosa)
variable (Filosofa Curiosa : Cosa → Prop)
variable (madreDi : Cosa → Cosa)
variable (Conosce : Cosa → Cosa → Prop)

end LinguaggioDeiPredicati
```

La sezione delimita la portata di questi parametri. Inoltre Lean aggiunge a un teorema soltanto i parametri che compaiono effettivamente nel suo enunciato.

Dichiarare `Filosofa : Cosa → Prop` non significa assumere che qualcuno sia una filosofa. Per avere una dimostrazione che Ipazia è una filosofa serve una vera assunzione:

```
variable (Cosa : Type)
variable (ipazia : Cosa)
variable (Filosofa : Cosa → Prop)
variable (hIpaziaFilosofa : Filosofa ipazia)

#check hIpaziaFilosofa
```

Il vocabolario determina quali proposizioni possiamo formulare; le assunzioni forniscono dimostrazioni di alcune di quelle proposizioni.

## 1.11 Variabili e quantificatori

Una variabile come `x : Cosa` rappresenta un oggetto non ancora specificato del dominio. Possiamo costruire espressioni aperte come:

```
variable (Cosa : Type)
variable (x : Cosa)
variable (Persona Filosofa Curiosa : Cosa → Prop)

#check Filosofa x
#check Persona x ∧ Filosofa x
#check Persona x → Filosofa x → Curiosa x
```

Per ottenere enunciati che riguardano tutti gli oggetti oppure almeno un oggetto usiamo i quantificatori:

- ∀ si legge «per ogni»;
- ∃ si legge «esiste almeno un».

| Linguaggio naturale | Lean |
|---|---|
| Ogni persona è curiosa | `∀ x : Cosa, Persona x → Curiosa x` |
| Qualche persona è curiosa | `∃ x : Cosa, Persona x ∧ Curiosa x` |
| Ogni filosofa è curiosa | `∀ x : Cosa, Filosofa x → Curiosa x` |
| Esiste una filosofa curiosa | `∃ x : Cosa, Filosofa x ∧ Curiosa x` |

All'inizio scriviamo esplicitamente `x : Cosa`: il quantificatore introduce la variabile `x` e ne dichiara il dominio. Lean può talvolta ricavare il tipo dai predicati, ma la forma esplicita rende più chiara la struttura logica.

## 1.12 Implicazione e congiunzione nei quantificatori

Due schemi compaiono frequentemente nelle formalizzazioni:

```
variable (Cosa : Type)
variable (Filosofa Curiosa : Cosa → Prop)

#check ∀ x : Cosa, Filosofa x → Curiosa x
#check ∃ x : Cosa, Filosofa x ∧ Curiosa x
```

Nel primo caso prendiamo una cosa qualsiasi e diciamo: *se* è una filosofa, allora è curiosa. Il quantificatore universale copre l'intero dominio; l'implicazione restringe l'affermazione agli oggetti che soddisfano `Filosofa`.

Nel secondo caso cerchiamo un singolo testimone che sia *sia* una filosofa *sia* curiosa. Per questo le due proprietà sono collegate da una congiunzione.

Consideriamo altri esempi:

| Enunciato | Lean |
|---|---|
| Ogni studentessa legge | `∀ x : Cosa, Studentessa x → Legge x` |
| Qualche studentessa legge | `∃ x : Cosa, Studentessa x ∧ Legge x` |
| Ogni studentessa che legge comprende | `∀ x : Cosa, Studentessa x → Legge x → Comprende x` |
| Esiste una studentessa che legge e comprende | `∃ x : Cosa, Studentessa x ∧ Legge x ∧ Comprende x` |

Possiamo verificare le stesse formalizzazioni in Lean:

```
variable (Cosa : Type)
variable (Studentessa Legge Comprende : Cosa → Prop)

#check ∀ x : Cosa, Studentessa x → Legge x
#check ∃ x : Cosa, Studentessa x ∧ Legge x
#check ∀ x : Cosa, Studentessa x → Legge x → Comprende x
#check ∃ x : Cosa, Studentessa x ∧ Legge x ∧ Comprende x
```

`#check` verifica che ogni espressione sia ben formata e ne mostra il tipo. Non dimostra che l'enunciato sia vero.

## 1.13 Fonti e approfondimenti

- Jeremy Avigad, Robert Y. Lewis e Floris van Doorn, *Logic and Proof*, «Natural Deduction Rules»: https://leanprover.github.io/logic_and_proof_lean3/nd_quickref.html.
- Lean developers, *The Lean Language Reference*, «Quantifiers»: https://lean-lang.org/doc/reference/latest/Basic-Propositions/Quantifiers/.
- Jeremy Avigad, Robert Y. Lewis e Floris van Doorn, *Logic and Proof*, «First Order Logic in Lean»: https://leanprover.github.io/logic_and_proof_lean3/first_order_logic_in_lean.html.
- Jeremy Avigad, Leonardo de Moura, Soonho Kong e Sebastian Ullrich, *Theorem Proving in Lean 4*, «Classical Logic»: https://lean-lang.org/theorem_proving_in_lean4/Propositions-and-Proofs/.