Logica per l’informatica

Matteo Lombardi

Indice

Contenuti

▼ **1.0 - Introduzione a Linux**

Cos’è una shell?

Una shell è un programma che permette di interagire con il sistema operativo.

Una shell non fa parte del kernel del sistema operativo, dunque possono esserci più shell che operano sullo stesso sistema operativo.

Le shell si dividono in due categorie:

- Shell testuali

  Vengono eseguite su terminali e permettono di effettuare operazioni e ricavare informazioni dal sistema operativo.

- Shell grafiche

  Permettono di interagire con il sistema operativo tramite un interfaccia di tipo grafica, nella quale è possibile eseguire programmi visualizzabili all’interno di finestre.

Le shell testuali, nonostante siano più antiche, permettono ancora oggi operazioni più avanzate rispetto a quelle effettuabili dalle shell grafiche:

- Eseguire un programma passando parametri non presenti in una shell grafica.
- Lavorare molto più velocemente rispetto alle shell grafiche.
- Sono sempre disponibili, anche quando le shell grafiche non lo sono.
- Forniscono un proprio linguaggio di programmazione, il quale permette di automatizzare e velocizzare task ripetitive.

Identificazione utente

Ogni utente, una volta effettuata l’autenticazione all’interno di un sistema operativo Linux, viene identificazione attraverso un **USERID**, il quale è unico ed è stato creato alla registrazione dell’utente.

A tale USERID verranno associati i file creati e i programmi eseguiti dall’utente.

Tramite il comando **id** dalla shell è possibile visualizzare il proprio id e gli id dei gruppi ai quali l’utente appartiene. Questi gruppi definiscono i file che un utente può leggere/scrivere/eseguire.

L’accesso al file system è regolamentato tramite i **permessi** che un determinato utente ha su un certo file, i quali possono essere modificati solo dal proprietario del file.

Un utente particolare è l’amministratore di sistema, identificato tramite l’id root, il quale ha sempre tutti i permessi per ogni tipo di file.

Il file system

Il **file system** è un sistema di organizzazione di grandi quantità di file, al fine renderne più semplice l’accesso.

All’interno di Linux esistono 4 tipi di file, file normale, directory, link simbolico e link fisico. I file sono organizzati in ordine **gerarchico**, dunque una directory può contenere al suo interno dei file.

Un **link simbolico** è semplicemente un puntatore che punta ad un altro file, mentre tramite un **link fisico** lo stesso file viene memorizzato in più locazioni differenti, dunque per cancellare definitivamente il file occorre eliminarlo da tutte le directory in cui si trova.

Se un pat inizia con la **~** seguita da un username, allora si intende come prima directory del path la cartella **HOME** dell'utente.

Comandi

- **echo $SHELL**

  Ritorna il nome della shell attualmente in utilizzo.

- **echo $HOME**

  Ritona il percorso della cartella HOME dell'utente.

- **bash**

  Cambiare shell ed utilizzare la bash.

- **id**

  Ritorna l'USERID e gli id dei gruppi ai quali l'utente appartiene.

- **ls -al [dir/file]**

  Ritorna una lista dei file presenti nella cartella insieme ai loro permessi, proprietario, dimensione, data i ultima modifica e nome.

  I permessi si leggono in questo modo: la prima lettera specifica se il file è una directory (d) o un file speciale (altre lettere), mentre se vi è un - si tratta di un file normale. Seguono 3 gruppi di 3 lettere (r: read; w: write: x: execute), le quali indicano i permessi collegati a quel file. Il primo gruppo rappresenta i diritti concessi al proprietario del file. Il secondo gruppo rappresenta i permessi concessi agli utenti che appartengono al gruppo del file. Il terzo gruppo rappresenta i permessi concessi a tutti gli altri utenti.

- **touch [file]**

  Creare un file.

- **chgrp [groupid] [file]**

  Cambiare il gruppo di un file.

- **chmod [g/o]+[r/w/x] foo.txt**

  Aggiungere (+) / Togliere (-) un permesso al gruppo (g) o a tutti gli altri utenti (o).

- **ps auxxx**

  Lista dei processi in esecuzione.

  La prima colonna indica l'USERID dell'utente che ha avviato il processo. La seconda colonna indica il numero univoco che identifica il processo.

- **kill -9 PID**

  Determinare la chiusure forzata di un file, inviandogli il segnale 9 (chiusura forza).

- **man [command]**

  Ritorna il manuale di un certo comando.

- **cp [filepath] [destinationpath]**

  Copiare un file in un'altra directory.

- **find [path] -name [filepath]**

  Ritorna una lista dei file file con il path [filepath] all'interno della cartella [path].

- **ln -s [filepath] [linkpath]**

  Creare un link simbolico del file [filepath] dentro [linkpath].

- **ln [filepath] [linkpath]**

  Creare un link fisico del file [filepath] dentro [linkpath].

- **ssh -X [unibousername]@[machinename].cs.unbo.it**

  Connettersi ad una macchina virtuale unibo.

- **w**

  Lanciato da una macchina remota, restituisce una lista di tutti gli utenti connessi ad essa.

▼ **2.0 - Introduzione alla logica**

Introduzione

Nel corso di logica per l'informatica verranno studiate le **dimostrazioni**, ovvero sequenze di frasi che convincono il lettore che un ragionamento è valido. Individueremo linguaggi artificiali per scrivere i passi di una dimostrazione e un computer potrà dire se sono corretti.

La parola **valere** ha diversi significati, in base alla logica a cui si fa riferimento, ad esempio può indicare verità, programmabilità, conoscenza, possesso ecc.

La logica ha molti elementi in comune con la matematica e l'informatica, ma anche altrettanti elementi per cui differisce:

Matematica	Informatica	Logica
Risolve problemi tramite **dimostrazioni**.	Risolve problemi tramite **codice**.	Garantisce la **correttezza** di un ragionamento.
Si interessa all'**esistenza** delle soluzioni di un problema.	Si interessa al modo di **calcolare** le soluzioni di un problema.	

Un **codice è corretto** solo se c'è una dimostrazione che dice che quello che fa è quello che deve fare. I bug logici di un programma sono causati da errori logici che sarebbero evitati esplicitandone la dimostrazione. Per scrivere codice il più delle volte corretto occorre imparare a ragionare logicamente formulando nella propria mente le prove della correttezza.

La dimostrazione di codice viene però utilizzata solo per il **codice critico**, ovvero un codice i quali errori possono causare problemi importanti (es. pilota automatico di un aereo), in quanto è molto dispendiosa sia in termini economici che di tempistiche.

Paradossi

Differenza tra paradossi e antinomie

Un **paradosso** consiste in una conclusione contraria all'intuizione che deriva da premesse accettabili per mezzo di un ragionamento accettato.

Un **antinomia** consiste in una conclusione inaccettabile che deriva da premesse accettabili per mezzo di un ragionamento accettato.

Nel corso si parlerà di paradossi intendendo antinomie al fine di semplificare.

Linguaggio naturale

Il **linguaggio naturale** è il linguaggio alla base delle comunicazioni tra gli esseri umani. Esso viene spesso utilizzato per descrivere procedure di calcolo e ragionamenti logici, ma ciò non è corretto in quanto esso contiene molti paradossi, è ambiguo e dipende fortemente dal contesto.

Paradossi in matematica

Nel linguaggio matematico è semplice introdurre dei nuovi paradossi. Vediamo un esempio di paradosso matematico nel **paradosso di Russell**.

Prendiamo l'insieme di tutti gli insiemi che non contengono sè stessi $X = \{Y | Y \not\in Y\}$ e chiediamoci se X contiene sè stesso:

- Se sì, X contiene un insieme che contiene sè stesso, dunque la premessa viene violata.
- Se no, X non contiene un insieme che non contiene sè stesso, dunque non contiene nessuno dei suoi sottoinsiemi, i quali non contengono sè stessi.

L'insieme X formato da tutti gli insiemi che non contengono sè stessi è dunque un **paradosso**.

Per evitarlo occorre stabilire di non poter formare un insieme partendo da una qualsiasi proprietà, ma è possibile solo selezionare elementi a partire da un insieme già esistente. Ad esempio non è possibile formare l'insieme $\{Y | Y$ è uno studente di informatica$\}$, ma è possibile formare l'insieme $\{Y$ appartiene all'insieme degli studenti $| Y$ studia informatica$\}$.

Inoltre occorre stabilire che la collezione di tutti gli insiemi non è un insieme ma una **classe propria**, dunque non è possibile l'uso metalinguistico della nozione di insieme, ovvero un insieme che parla di sè stesso.

Paradossi in informatica

Molti dei paradossi in informatica sono dati dall'**uso metalinguistico delle funzioni**. Nei linguaggi higher order infatti una funzione può prendere in input/dare in output altre funzioni, mentre nei linguaggi imperativi o a oggetti una funzione può prendere in input/dare in output delle reference ad altre funzioni.

Esempio:

- Siano f e g due funzioni, e sia $f(g) = not(g(g))$, allora $f(f) = not(f(f))$, il che è assurdo.

  Dunque ci sono due possibilità; o f non è scrivibile nel linguaggio di programmazione, il quale risulterebbe molto inespressivo, oppure f non è totale, in quanto $f(f)$ diverge, ovvero non fornisce output in tempo finito.

Chiediamoci inoltre se esista un programma che sia in grado di stabilire se un altro diverga o meno ($\downarrow$: converge, $\uparrow$: diverge):

$f(g, x) =$ true iff $g(x) \downarrow$ è il programma che stabilisce se un altro diverga o meno (g è il programma e x è il suo parametro).

$h(g) =$ if $f(g, g)$ then $\uparrow$ else $\downarrow$ è il programma che abbiamo creato per dimostrare che non per tutti i programmi è possibile stabilire se divergano o meno.

$\implies h(h) \uparrow$ iff $f(h, h) =$ true iff $h(h) \downarrow$, il che è assurdo.

Chiediamoci infine se ogni linguaggio di programmazione può esprimere tutte le funzioni matematiche. Consideriamo un linguaggio di programmazione non tipato e T come l'insieme di tutti i valori possibili nel linguaggio (bool, funzioni ecc.).

$T = \{0, 1\} \cup T^T$ (T^T = tutte le funzioni matematiche che vanno dal dominio T all'immagine T)

T dunque contiene almeno i valori booleani 0 e 1 e tutte le funzioni matematiche che vanno da T a T. Per il teorema della diagonalizzazione di Cantor $|T| < 2 + |T^T|$ ($|| :$ cardinalità), il che è assurdo, dunque non tutti i linguaggi di programmazione possono esprimere tutte le funzioni matematiche.

▼ 3.0 - Teoria assiomatica degli insiemi

▼ 3.1 - Teoria degli insiemi naive

Nella **teoria degli insiemi naive** tutto viene trattato come un insieme, dunque numeri, figure geometriche, funzioni rappresentano insiemi particolari. Questo permette di avere principalmente un vantaggio e uno svantaggio:

- Permette di introdurre e comprendere concetti come i limiti.
- Causa la perdita di alcuni aspetti computazionali (Es. le funzioni come insiemi non si calcolano).

In questa teoria gli insiemi non devono per forza essere omogenei (possono possedere un mix di qualunque cosa, in quanto tutto è considerato come un insieme) e le ripetizioni e l'ordine non contano (Es. {1, 2, 3} = {1, 3, 3, 2, 1}).

I diagrammi di Venn non riescono a rappresentare molto bene la teoria degli insiemi e vanno utilizzati con cautela.

Appartenenza $\in$

Il primo insieme deve essere esattamente contenuto nel secondo, non basta che i suoi insiemi siano contenuti.

- {1, 2} $\notin$ {1, 2, 3}
- 1 $\in$ {1, 2, 3}
- 1 $\notin$ {{1, 2}, 3}
- {1, 2} $\in$ {{1, 2}, 3}
- $\varnothing \notin$ {1, 2, 3}
- $\varnothing \in$ {$\varnothing$, 1}

Inclusione $\subseteq$

Il primo insieme deve essere allo stesso livello del secondo insieme ma può avere meno insiemi al suo interno.

- {1, 3} $\not\subseteq$ {{1, 3}, 2}
- {1, 3} $\subseteq$ {1, 2, 3}
- 1 $\not\subseteq$ {1, 2, 3}
- {1} $\subseteq$ {1, 2, 3}
- {1} $\not\subseteq$ 1
- 1 $\subseteq$ 1

▼ 3.2 - Teoria assiomatica degli insiemi

La teoria assiomatica degli insiemi nasce dal fatto che la teoria degli insiemi naive porta a dei paradossi, occorre:

- Eliminare l'assioma di comprensione (per qualunque proprietà P è possibile formare un insieme $\{x|P(x)\}$) il quale porta al paradosso di Russell.
- Controllare l'uso meta-linguistico (es. nel paradosso di Russell l'insieme X non può essere un insieme ma una classe in quanto altrimenti parlerebbe di sè stesso).

La teoria assiomatica degli insiemi si basa dunque su:

- La non definizione del concetto di insieme, considerato quindi come un elemento primitivo.

- L'utilizzo di assiomi per permettere di creare insiemi di partenza dai quali gli altri insiemi possono essere creati.

Esistono numerose teorie insiemistiche che differiscono per quanto riguarda i loro assiomi, noi utilizzeremo la **teoria di Zermelo-Fraenkel (ZF)**, la quale è la meno controversa ed è sufficiente per sviluppare gran parte della matematica. Nonostante ciò questa teoria non è mai stata dimostrata essere consistente.
Questa teoria si basa sui seguenti assiomi:

Assioma di estensionalità

Due insiemi sono uguali se e solo se hanno gli stessi elementi.

$\forall X, \forall Y, (X = Y \iff \forall Z, (Z \in X \Leftrightarrow Z \in Y))$

Definizione di essere sottoinsieme

X è sottoinsieme di Y se ogni suo elemento appartiene anche a Y.

$X \subseteq Y = \forall Z, (Z \in X \Rightarrow Z \in Y)$

Assioma di separazione

Dato un insieme, è possibile formare un suo sottoinsieme che soddisfi una certa proprietà.

$\forall X, \exists Y, \forall Z, (Z \in Y \iff Z \in X \land P(Z))$

Da questo assioma è poi possibile indicare Y in questo modo:
$Y = \{Z \in X | P(Z)\}$

Assioma dell'insieme vuoto

$\exists X, \forall Z, Z \notin X$

X viene indicato così: $\emptyset$

Definizione di intersezione binaria

$X \in A \cap B \iff X \in A \land X \in B$

Intersezione n-aria

$A_1 \cap ... \cap A_n = ((A_1 \cap A_2) \cap ... \cap A_n)$

Tuttavia l'assioma di intersezione binaria non consente di intersecare un numero infinito di insiemi.

Definizione di intersezione

Dato un insieme di insiemi, esiste un insieme che ne è l'intersezione, il quale viene definito in questo modo:

$\bigcap F = \{X \in A | \forall Y, (Y \in F \Rightarrow X \in Y)\}$ dove $A \in F$

> Preso un qualunque insieme appartenente ad $\bigcap F$, ogni suo elemento appartiene ad ogni altro insieme presente all'interno dell'insieme F.

F non è l'insieme intersezione, ma l'insieme di tutti gli insiemi da intersecare, mentre l'insieme intersezione viene indicato come $\bigcap_{Y \in F} Y$ per esempio con $\bigcap_{Y \in \{A,B,C\}} Y = A \cap B \cap C$.

Assioma dell'unione

$\forall F, \exists X, \forall Z, (Z \in X \iff \exists Y, (Y \in F \land Z \in Y))$

L'insieme X viene indicato con $\bigcup F$ o $\bigcup_{Y \in F} Y$.

> Preso un qualunque insieme F di insiemi da unire, esiste un insieme unione X tale per cui per ogni elemento qualsiasi Z, Z appartiene all'insieme unione X se e solo se

> esiste almeno un insieme Y appartenente a F nel quale Z è contenuto

Teorema dell’unione binaria

$\forall A, \forall B, \exists X, \forall Z, (Z \in X \iff Z \in A \vee Z \in B)$

> Per ogni insieme A e B casuali, esiste un insieme X unione dei due insiemi tale per cui per ogni Z, Z appartiene ad A o Z appartiene a B.

Assioma del singoletto

Un **singoletto** è un insieme contenente un unico elemento.

$\forall X, \exists Y, \forall Z, (Z \in Y \iff Z = X)$

> Per ogni insieme X, esiste un insieme singoletto Y che lo contiene tale per cui, per ogni suo elemento Z, Z = X (esiste un solo elemento Z = X).

Abuso di notazione per indicare un insieme

Per indicare un insieme solitamente viene utilizzata la seguente forma:
$\{A_1, ..., A_n\}$

per indicare:

$\{A_1\} \cup ... \cup \{A_n\}$, il quale esiste per i teoremi del singoletti e dell’unione.

Costruzione dei numeri naturali

Per costruire i numeri naturali ci serviamo degli insiemi, definendo l’insieme vuoto come 0 e poi creando i numeri successivi come l’unione tra il numero precedente e l’insieme contenente il numero precedente:

$N + 1 = N \cup \{N\}$

Esempio:

$0 = \emptyset$

$1 = \{\emptyset\}$

$2 = \{\emptyset, \{\emptyset\}\}$

$3 = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}$

...

Assioma dell’infinito

$\exists Y, (\emptyset \in Y \wedge \forall N, (N \in Y \wedge N \cup \{N\} \in Y))$

> Esiste un insieme infinito Y il quale contiene l’insieme vuoto e, preso un qualsiasi insieme N, N appartiene a Y e N + 1 appartiene a Y

Assioma dell’insieme potenza

Assioma che asserisce l’esistenza dell’insieme dei sottoinsiemi di un insieme dato.

$\forall X, \exists Y, \forall Z, (Z \in Y \iff Z \subseteq X)$

> Per ogni insieme X esiste un insieme Y dei sottoinsiemi di X tale che, per ogni elemento casuale Z, Z appartiene a Y se e solo se Z è contenuto in X.

Y viene indicato come 2^X oppure $P(X)$ (**insieme potenza** o **insieme delle parti**).

Esempio:

$2^{\{1,2\}} = \{\emptyset, \{1\}, \{2\}, \{1,2\}\}$

Assioma di regolarità

Ogni insieme non vuoto possiede almeno un elemento dal quale è disgiunto (due insiemi disgiunti sono due insiemi che non hanno nessun elemento in comune).

Una delle conseguenze di questo assioma è che nessun insieme contiene sè stesso.

Assioma di rimpiazzamento

L'immagine di un insieme rispetto ad una formula che descrive una funzione è sempre un insieme.

In parole povere: tutti gli elementi immagine ricavati tramite una formula che descrive una funzione a partire da un insieme (chiamato in questo caso dominio in quanto parliamo di funzioni) formano sempre un insieme.

▼ **3.3 - Dimostrazioni in teoria assiomatica degli insiemi**

Per ogni ($\forall$)

$\forall$ **introduzione**

Scopo: per dimostrare $\forall x.P(x)$.

Linguaggio naturale: sia x un insieme fissato; [dimostrazione di P(x)]

Matita: **assume x: set. [dimostrazione di $P(x)$]**.

$\forall$ **eliminazione**

Scopo: da un'ipotesi o un risultato intermedio $\forall x.P(x)$ possiamo concludere che P valga per una x a nostra scelta.

Matita: **by [NOME_IPOTESI] we proved [CONCLUSIONE] ([NOME_RISULTATO_OTTENUTO])**.

Esiste ($\exists$)

$\exists$ **introduzione**

Scopo: per dimostrare $\exists x.P(x)$.

Linguaggio naturale: scelgo x e dimostro $P(x)$: [dimostrazione di $P(x)$]

Matita: **by ex_intro, [NOME_IPOTESI_$P(e)$] we proved** $\exists x.P(x)$ ($P(e)$ è una sottoprova che dimostra $P(x)$ scegliendo come x un valore e).

$\exists$ **eliminazione**

Scopo: da un'ipotesi o un risultato intermedio $\exists x.P(x)$ è possibile procedere nella prova utilizzando la x che soddisfa $P(x)$.

Linguaggio naturale: sia x tale che $P(x)$.

Matita: **by [NOME_IPOTESI_$\exists x.P(x)$] let x: set such that $P(x)$ ([NOME_IPOTESI])**.

Implicazione ($\implies$)

$\implies$ **introduzione**

Scopo: dimostrare $P \implies Q$

Linguaggio naturale: assumo P ([NOME_IPOTESI]). [dimostrazione di Q]

Matita: **suppose P ([NOME_IPOTESI]). [dimostrazione di Q]**.

$\implies$ **eliminazione**

Scopo: da un ipotesi o un risultato intermedio $P \implies Q$ e da un ipotesi o un risultato intermedio P è possibile concludere che Q vale.

Matita: **by [NOME_IPOTESI_PQ], [NOME_IPOTESI_P] we proved [Q] ([NOME_RISULTATO_OTTENUTO])**.

$\implies$ **eliminazione (variante)**

Scopo: da un ipotesi o un risultato intermedio $P \implies Q$, per dimostrare Q mi posso ridurre a dimostrare P.

Linguaggio naturale: per $P \implies Q$, per dimostrare Q mi posso ridurre a dimostrare P.

Matita: **we need to prove P ([NOME_IPOTESI_P]). [dimostrazione di P]. by [NOME_IPOTESI_PQ], [NOME_IPOTESI_P] done**.

Se e solo se ($\iff$)

$\iff$ **introduzione**

Scopo: per dimostrare $P \iff Q$ si dimostra sia $P \implies Q$ che $Q \implies P$.

$\iff$ **eliminazione**

Scopo: l'ipotesi $P \iff Q$ può essere utilizzata sia come ipotesi $P \implies Q$ che come ipotesi $Q \implies P$.

And ($\wedge$)

$\wedge$ **introduzione**

Scopo: per dimostrare $P \wedge Q$ si dimostrano sia P che Q.

Matita: **by conj, P, Q we proved $P \wedge Q$ ([NOME_RISULTATO_OTTENUTO])**.

$\wedge$ **eliminazione**

Scopo: un'ipotesi o un risultato intermedio $P \wedge Q$ può essere usato sia come P che come Q.

Matita: **by [NOME_IPOTESI_$P \wedge Q$] we proved P ([NOME_IPOTESI_P]) and Q ([NOME_IPOTESI_Q])**.

Or ($\vee$)

$\vee$ **introduzione**

Scopo: per dimostrare $P \vee Q$ basta dimostrare P o Q.

Matita: **by or_introl, [NOME_IPOTESI_P] we proved $P \vee Q$. by or_intror, [NOME_IPOTESI_Q] we proved $P \vee Q$**.

$\vee$ **eliminazione**

Scopo: per utilizzare un'ipotesi o un risultato intermedio $P \vee Q$, occorre proseguire per casi nella dimostrazione, una volta assumendo che P valga e una volta che Q valga.

Linguaggio naturale: procedo per casi:

- caso in cui valga P (H): [dimostrazione]
- caso in cui valga Q (H): [dimostrazione]

Matita: we proceed by cases on [NOME_IPOTESI_$P \vee Q$] to prove [CONCLUSIONE]:

case or_introl

suppose P (H)

[dimostrazione di [CONCLUSIONE] sotto l'ipotesi H:P]

case or_intror

suppose Q (H)

[dimostrazione di [CONCLUSIONE] sotto l'ipotesi H:Q]

Not ($\neg$)

$\neg$ introduzione

Scopo: per dimostrare $\neg P$ occorre dimostrare $P \implies$ assurdo.

Inoltre, data un ipotesi $\neg P$ e un'altra ipotesi o un risultato intermedio P si conclude l'assurdo.

Assurdo eliminazione

Scopo: se ho dimostrato l'assurdo posso concludere qualsiasi cosa.

Matita: **using (ABSURDUM [NOME_ASSURDO]) done**

Tips

- Tutte le variabili di un enunciato che non sono introdotte da un per ogni o da un esiste si considerano introdotte da dei per ogni (es. l'enunciato n + m = m + n abbrevia $\forall$n, m. n + m = m + n).
- Tutti gli assiomi sono sempre utilizzabili come prove in qualunque momento.
- A volte occorre esplicitare la conclusione corrente (cosa resta da dimostrare) attraverso "**the thesis becomes P**".
- Per indicare che il lettore è in grado di ricostruire la prova per conto suo combinando le ipotesi si utilizza "**done**".
- $\forall x \in A, P(x)$ viene utilizzato per indicare $\forall x, (x \in A \implies P(x))$.
- $\exists x \in A, P(x)$ viene utilizzato per indicare $\exists x, (x \in A \land P(x))$.

▼ 3.4 - Esercizi di dimostrazioni in teoria assiomatica degli insiemi

```
(* Exercise 1 *)
theorem reflexivity_inclusion: ∀A. A ⊆ A.
 assume A:set
 we need to prove (∀Z. Z ∈ A → Z ∈ A) (ZAtoZA)
  assume Z:set
  suppose (Z ∈ A) (ZA)
  by ZA (* Quale ipotesi serve? Osservate cosa bisogna dimostrare *)
 done
 by ax_inclusion2, ZAtoZA (* Quale ipotesi devo combinare con l'assioma? *)
done
qed.

(* Exercise 2 *)
theorem transitivity_inclusion: ∀A,B,C. A ⊆ B → B ⊆ C → A ⊆ C.
 assume A:set
 assume B:set
 assume C:set
 suppose (A ⊆ B) (AB)
 suppose (B ⊆ C) (BC)
 we need to prove (∀Z. Z ∈ A → Z ∈ C) (ZAtoZC)
  assume Z:set
  suppose (Z ∈ A) (ZA)
  by AB, ax_inclusion1, ZA we proved (Z ∈ B) (ZB)
  by BC, ax_inclusion1, ZB we proved (Z ∈ C) (ZC)(* Osservate bene cosa deve essere dimostrato *)
 done
 by ax_inclusion2, ZAtoZC
done
qed.

(* Exercise 3 *)
theorem antisymmetry_inclusion: ∀A,B. A ⊆ B → B ⊆ A → A = B.
 assume A:set
 assume B:set
 suppose (A ⊆ B) (AB)
 suppose (B ⊆ A) (BA)
 we need to prove (∀Z. Z ∈ A ⇔ Z ∈ B) (P)
  assume Z:set
  by AB, ax_inclusion1 we proved (Z ∈ A → Z ∈ B) (AB')
  by BA, ax_inclusion1 we proved (Z ∈ B → Z ∈ A) (BA')
  by conj, AB', BA'
 done
 by ax_extensionality, P  (* Quale assioma devo utilizzare? *)
```

```
done
qed.

(* Exercise 4 *)
theorem intersection_idempotent_1: ∀A. A ⊆ A ∩ A.
 assume A:set
 we need to prove (∀Z. Z ∈ A → Z ∈ (A ∩ A)) (AinAA)
  assume Z:set
  suppose (Z ∈ A) (ZA)
  we need to prove (Z ∈ A ∧ Z ∈ A) (ZAZA)
   by conj, ZA (* Il teorema conj serve per dimostrare una congiunzione (un "and" ∧) *)
  done
  by ax_intersect2, ZAZA (* Cosa stiamo dimostrando? Che assioma serve? *)
 done
 by ax_inclusion2, AinAA
done
qed.

(* Exercise 5*)
theorem intersect_monotone_l: ∀A,A',B. A ⊆ A' → A ∩ B ⊆ A' ∩ B.
 assume A:set
 assume A':set
 assume B:set
 suppose (A ⊆ A') (AA')
 we need to prove (∀Z. Z ∈ A ∩ B → Z ∈ A' ∩ B) (P)
  assume Z:set
  suppose (Z ∈ A ∩ B) (F)
  by ax_intersect1, F we proved (Z ∈ A ∧ Z ∈ B) (ZAZB)
  by ZAZB we have (Z ∈ A) (ZA) and (Z ∈ B) (ZB) (* Da un'ipotesi congiunzione si ricavano due ipotesi distinte *)
  by ax_inclusion1, AA' we proved (Z ∈ A') (ZA')
  by conj  we proved (Z ∈ A' ∧ Z ∈ B) (ZAZB')
  by ax_intersect2 (* Cosa stiamo dimostrando? Che assioma serve? *)
 done
by ax_inclusion2, P
done
qed.

(* Exercise 6*)
theorem intersect_monotone_r: ∀A,B,B'. B ⊆ B' → A ∩ B ⊆ A ∩ B'.
 assume A:set
 assume B:set
 assume B':set
 suppose (B ⊆ B') (BB')
 we need to prove (∀Z. Z ∈ A ∩ B → Z ∈ A ∩ B') (P)
  assume Z:set
  suppose (Z ∈ A ∩ B) (F)
  by ax_intersect1, F we proved (Z ∈ A ∧ Z ∈ B) (ZAZB)
  by ZAZB we have (Z ∈ A) (ZA) and (Z ∈ B) (ZB)
  by ax_inclusion1, BB' we proved (Z ∈ B') (ZB')
  by conj we proved (Z ∈ A ∧ Z ∈ B') (ZAZB')
  by ax_intersect2
 done
by ax_inclusion2, P
done
qed.
```

```
(* Exercise 7 *)
theorem union_inclusion: ∀A, B. A ⊆ A ∪ B.
  assume A: set
  assume B: set
  we need to prove (∀Z. Z ∈ A → Z ∈ A ∪ B) (I)
    assume Z: set
    suppose (Z∈A) (ZA)
    we need to prove (Z ∈ A ∨ Z ∈ B) (I1)
      by ZA,or_introl
    done
    by ax_union2, I1
  done
  by ax_inclusion2, I done
qed.

(* Exercise 8 *)
theorem union_idempotent: ∀A. A ∪ A = A.
 assume A: set
 we need to prove (∀Z. Z ∈ A ∪ A ⇔ Z ∈ A) (II)
 assume Z: set
  we need to prove (Z ∈ A ∪ A → Z ∈ A) (I1)
```

```
   suppose (Z ∈ A ∪ A)(Zu)
   by ax_union1, Zu we proved (Z ∈ A ∨ Z ∈ A) (Zor)
   we proceed by cases on Zor to prove (Z ∈ A)
    case or_introl
       suppose (Z∈A) (H)
       by H
    done
    case or_intror
       suppose (Z∈A) (H)
       by H
    done
  we need to prove (Z ∈ A → Z ∈ A ∪ A) (I2)
  suppose (Z∈A) (ZA)
   by ZA, or_introl we proved (Z ∈ A ∨ Z ∈ A) (Zor)
   by ax_union2, Zor
  done
  by conj, I1, I2
 done
 by II,ax_extensionality
done
qed.

(* Exercize 9 *)
theorem empty_absurd: ∀X, A. X ∈ ∅ → X ∈ A.
 assume X: set
 assume A: set
 suppose (X ∈ ∅) (XE)
 by ax_empty we proved False (bottom)
 using (ABSURDUM bottom)
done
qed.

(* Exercise 10 *)
theorem intersect_empty: ∀A. A ∩ ∅ = ∅.
 assume A: set
 we need to prove (∀Z. Z ∈ A ∩ ∅ ⇔ Z ∈ ∅) (II)
   assume Z: set
   we need to prove (Z ∈ A ∩ ∅ → Z ∈ ∅) (I1)
     suppose (Z ∈ A ∩ ∅) (Ze)
     we need to prove (Z ∈ ∅)
     by Ze, ax_intersect1 we have (Z ∈ A) (ZA) and (Z∈∅) (ZE)
     by ZE
   done
   we need to prove (Z ∈ ∅ → Z ∈ A ∩ ∅) (I2)
     suppose (Z ∈ ∅) (ZE)
     by ZE, ax_empty we proved False (bottom)
     using (ABSURDUM bottom)
   done
   by I1, I2, conj
 done
 by II, ax_extensionality
done
qed.

(* Exercise 11 *)
theorem union_empty: ∀A. A ∪ ∅ = A.
 assume A: set
 we need to prove (∀Z. Z ∈ A ∪ ∅ ⇔ Z ∈ A) (II)
   assume Z:set
   we need to prove (Z ∈ A → Z ∈ A ∪ ∅) (I1)
     suppose (Z ∈ A) (ZA)
     by or_introl, ZA we proved (Z ∈ A ∨ Z ∈ ∅) (Zor)
     by ax_union2,Zor
   done
   we need to prove (Z ∈ A ∪ ∅ → Z ∈ A) (I2)
     suppose (Z ∈ A ∪ ∅) (Zu)
     by ax_union1, Zu we proved (Z ∈ A ∨ Z ∈ ∅) (Zor)
     we proceed by cases on Zor to prove (Z ∈ A)
      case or_introl
        suppose (Z ∈ A) (H)
        by H done
      case or_intror
        suppose (Z ∈ ∅) (H)
         by ax_empty, H we proved False (bottom)
         using (ABSURDUM bottom)
      done
    by conj, I1, I2
  done
  by ax_extensionality, II
```

```
 done
qed.

(* Exercise 12 *)
theorem union_commutative: ∀A,B. A ∪ B = B ∪ A.
 assume A: set
 assume B: set
 we need to prove (∀Z. Z ∈ A ∪ B ⇔ Z ∈ B ∪ A) (II)
   assume Z: set
   we need to prove (Z ∈ A ∪ B → Z ∈ B ∪ A) (I1)
     suppose (Z ∈ A ∪ B) (ZAB)
     we need to prove (Z ∈ B ∪ A)
     we need to prove (Z ∈ B ∨ Z ∈ A) (I)
       by ZAB, ax_union1 we proved (Z∈A∨Z∈B) (Zor)
       we proceed by cases on Zor to prove (Z ∈ B ∨ Z ∈ A)
       case or_intror
         suppose (Z∈B) (H)
         by H,or_introl
       done
       case or_introl
         suppose (Z∈A) (H)
         by H, or_intror
       done
   by I,ax_union2 done
   we need to prove (Z ∈ B ∪ A → Z ∈ A ∪ B) (I2)
     suppose (Z ∈ B ∪ A) (ZBA)
     we need to prove (Z ∈ A ∪ B)
     we need to prove (Z ∈ A ∨ Z ∈ B) (I)
       by ZBA, ax_union1 we proved (Z∈B∨Z∈A) (Zor)
       we proceed by cases on Zor to prove (Z ∈ A ∨ Z ∈ B)
       case or_intror
         suppose (Z∈A) (H)
         by H,or_introl
       done
       case or_introl
         suppose (Z∈B) (H)
         by H, or_intror
       done
   by I,ax_union2 done
  by I1,I2,conj
 done
 by ax_extensionality,II
done
qed.
```

▼ **4.0 - Relazioni, funzioni e insiemi**

▼ **4.1 - Relazioni**

Coppie ordinate

Coppie ordinate vs insiemi

- In un **insieme** l'ordine non conta e nemmeno la numerosità degli elementi: $\{1,2\} = \{2,1\}$ e $\{1,1\} = \{1\}$.
- Una **coppia ordinata** è formata da due elementi di cui uno è identificato come primo e l'altro come secondo. Due coppie sono uguali se e solo se lo solo rispettivamente il primo e il secondo elemento. Una coppia non è l'insieme dei suoi elementi e non deve essere pensata come contenente (nel senso di $\in$) i suoi elementi.

  $\langle 1,2\rangle \neq \langle 2,1\rangle, \langle 1,2\rangle \neq \{1,2\}, 1 \notin \langle 1,2\rangle$.

Codifica di una coppia ordinata

Dati X, Y chiamiamo coppia ordinata di prima componente X e di seconda componente Y, e la indichiamo con $\langle X,Y\rangle$, l'insieme $\{X, \{X,Y\}\}$.

Prodotto cartesiano

> $$\forall A, \forall B, \exists C, \forall Z, (Z \in C \iff \exists a, \exists b, (a \in A \land b \in B \land Z = \langle a,b\rangle))$$

L'insieme C viene chiamato **prodotto cartesiano** di A e B e indicato come $A \times B$.

Esempio: $\{a,b\} \times \{1,2\} = \{\langle a,1\rangle, \langle a,2\rangle, \langle b,1\rangle, \langle b,2\rangle\}$

Relazioni

> Una **relazione** fra A e B è un qualunque sottoinsieme di $A \times B$.

Sia $\mathcal{R}$ una relazione, scriviamo $a\mathcal{R}b$ se $\langle a,b\rangle \in \mathcal{R}$.

Se $\mathcal{R} \subseteq A \times \emptyset$ oppure $\mathcal{R} \subseteq \emptyset \times A$ allora $\mathcal{R} = \emptyset$.

Esempio:

- La relazione $\leq$ sull'insieme $\{0,1,2\}$ è definita come segue: $\leq =$ $\{\langle 0,0\rangle, \langle 0,1\rangle, \langle 0,2\rangle, \langle 1,1\rangle, \langle 1,2\rangle, \langle 2,2\rangle\}$ e $0 \leq 2$ è solo una notazione per $\langle 0,2\rangle \in \leq$.

Proprietà delle relazioni

Sia $\mathcal{R} \subseteq A \times A$, essa gode delle proprietà:

- **Riflessiva** se $\forall X \in A, X\mathcal{R}X$.
- **Simmetrica** se $\forall X, Y \in A, (X\mathcal{R}Y \implies Y\mathcal{R}X)$.
- **Transitiva** se $\forall X, Y, Z \in A, (X\mathcal{R}Y \land Y\mathcal{R}Z \implies X\mathcal{R}Z)$

Tipi di relazioni

> Una relazione è di **ordinamento stretto** se è transitiva e non riflessiva (es. $<$).

> Una relazione è di **ordinamento (lasco)** se è riflessiva, transitiva e antisimmetrica ($\forall X, Y \in A, \left(X \mathcal{R} Y \land Y \mathcal{R} X \implies X = Y\right)$) (es. $=$, $\leq$).

> Una relazione è di **equivalenza** se è riflessiva, simmetrica e transitiva (es. $=$, “essere dello stesso modello”).

Due elementi sono uguali se le proprietà che vengono soddisfatte da uno sono soddisfatte anche dall’altro, mentre due elementi sono equivalenti se entrambi soddisfano delle proprietà che ci interessano per quello che si deve fare.

Esempio: se viene considerato l’insieme delle automobili, due fiat punto sono equivalenti se viene considerata come proprietà solo il modello dell’auto, ma non sono uguali in quanto ad esempio hanno numero di targa differente.

Classi di equivalenza

> Sia $\equiv \subseteq A \times A$ una relazione di equivalenza su A. La sua **classe di equivalenza** di $x \in A$ rispetto a $\equiv$ è definita come segue: $[x]_{\equiv} = \{y \in A | y \equiv x\}$ (tutti gli elementi di A che sono equivalenti a x).

La classe di equivalenza di $x \in A$ è dunque formata da tutti gli elementi di A che sono equivalenti a x.

Esempio: la classe di equivalenza di una fiat punto nell’insieme delle automobili con la relazione di equivalenza che tiene conto del modello è costituita dall’insieme di tutte le fiat punto.

Teorema

> Sia $\equiv \subseteq A \times A$ una relazione di equivalenza su A. Prese due classi di equivalenza $[x]_{\equiv}, [y]_{\equiv}$ date da due elementi $x, y \in A$, abbiamo che $[x]_{\equiv} = [y]_{\equiv}$ (quando $x \equiv y$) oppure $[x]_{\equiv}$ e $[y]_{\equiv}$ sono insiemi disgiunti, ovvero non hanno nessun elemento in comune (quando $x \not\equiv y$).

Insieme quoziente

> Sia $\equiv \subseteq A \times A$ una relazione di equivalenza su A. L’**insieme quoziente** di A rispetto a $\equiv$ è definito come segue: $A_{/\equiv} = \{[x]_{\equiv} | x \in A\}$.

L’insieme quoziente è dunque formato da tutte le classi di equivalenza in A, quindi è un insieme formato da insiemi di A.

Gli insiemi quoziente sono strumenti molto potenti per creare **nuovi concetti** a partire da concetti pre-esistenti.

Esempio:

- Creazione dell’insieme dei numeri interi $\mathbb{Z}$ a partire dai naturali $\mathbb{N}$.

  Intuizione: $Z = \mathbb{Z} \times \mathbb{Z}$, $\langle 2, 4 \rangle \in \mathbb{Z} = 2 - 4$

  $\equiv \subseteq Z \times Z : \langle u_1, l_1 \rangle \equiv \langle u_2, l_2 \rangle \iff u_1 + l_2 = u_2 + l_1$ (es. $\langle 2, 4 \rangle \equiv \langle 3, 5 \rangle \iff 2 + 5 = 4 + 3$)

$\mathbb{Z} = Z_{/\equiv} \to Z = \{\dots, [\langle 0,2\rangle]_\equiv, [\langle 0,1\rangle]_\equiv, [\langle 0,0\rangle]_\equiv, [\langle 1,0\rangle]_\equiv, [\langle 2,0\rangle]_\equiv, \dots\}$

Sintassi: $[\langle 0,n\rangle]_\equiv = -n$, $[\langle n,0\rangle]_\equiv = +n$

$\mathbb{Z} = \{\dots, -2, -1, 0, +1, +2, \dots\}$

▼ **4.2 - Funzioni**

> Una **funzione** di dominio A e di codominio B è una qualunque relazione $f \subseteq A \times B$ tale che: $\forall X, (X \in A \implies \exists! Y \in B, X f Y)$ (per ogni elemento del dominio c'è un unico elemento del codominio in relazione con esso).

Notazione: sia f una funzione, scriviamo $y = f(x)$ per dire xfy, ovvero $\langle x, y\rangle \in f$.

Una funzione non stabilisce dunque il modo in cui vengono calcolate le sue coppie ordinate, ma, visto che è una relazione vera e propria, stabilisce solo un insieme di coppie ordinate facenti parte di essa.

Spazio di funzioni

> $\forall A, \forall B, \exists C, \forall f, (f \in C \iff f$ è una funzione di dominio A e codominio B).

Indichiamo C come B^A, ovvero **spazio di funzioni** da A a B.

Funzioni con insiemi vuoti

- $B^\emptyset = \{\emptyset\}$ (è una funzione)
- $\emptyset^A = \emptyset \iff A \neq \emptyset$ (non è una funzione)

▼ **5.0 - Sintassi in pseudo-linguaggio e ricorsione strutturale**

▼ **5.1 - Nozioni base di sintassi e BNF**

Parole chiave di sintassi

- Un **alfabeto** è un qualunque insieme non vuoto di simboli.

  Es. {a, b, c, …}, {0, 1}

- Una **stringa** è una sequenza finita di simboli dell'alfabeto.

  Es. sull'alfabeto {0, 1}: ϵ (stringa vuota), 0, 1, 011010

- Un **linguaggio** è un insieme finito o infinito di stringhe dello stesso alfabeto.

  Es. su {0, 1} : {ϵ, 0, 01, 10, 11, …} {0, 1}

- Una **grammatica** è un qualsiasi formalismo, ovvero una descrizione che definisce in modo rigoroso un insieme, che descrive un linguaggio.

Backus-Naur Form (BNF)

La **BNF** è un linguaggio che permette di creare grammatiche. In questo senso la BNF è un meta-linguaggio, in quanto è un linguaggio che consente di descrivere altri linguaggi.

La BNF però non riesce a descrivere tutti i linguaggi, in quanto esistono sia linguaggi semplici che complessi, e la BNF riesce a descrivere solo i linguaggi più semplici, come i linguaggi di programmazione.

La BNF è una quadrupla **(T, NT, X, P)** nella quale:

- **T**

  È un alfabeto di simboli detti **terminali** i quali verranno poi utilizzati per creare le stringhe del linguaggio.

- **NT**

  È un alfabeto di simboli detti **non terminali** distinti da T in quanto non verranno poi utilizzati per creare le stringhe del linguaggio, ma solo per definirlo.

- **X**

  È il **simbolo iniziale** dell'insieme NT.

- P

  È un insieme di coppie chiamate produzioni che hanno come primo valore un non terminale, e come secondo valore un insieme di stringhe create dall'alfabeto T e NT.

  La produzione (X, $\{\omega_1, \omega_2, ..., \omega_n\}$) si rappresenta come $X ::= \omega_1 | \omega_2 | ... | \omega_n$ (si legge X è ω_1 oppure ω_2 oppure … oppure ω_n)

Esempio: ({0, 1}, {X, Y}, X, {X ::= 0 | 0Y, Y ::= 1X})

Di solito si utilizza solo la P per descrivere una BNF poichè:

- I simboli a sinistra delle produzioni sono i non terminali.
- Il primo simbolo a sinistra di tutte le produzioni è il simbolo X iniziale dell'insieme NT.

  In realtà P è un insieme dunque l'ordine non conta, ma se P viene utilizzata per descrivere una BNF allora come prima produzione viene inserita quella con il primo valore corrispondente al primo non terminale, mentre per le altre produzioni l'ordine non conta.

- Il simboli a destra esclusi i simboli non terminali e le stringhe vuote (ϵ) sono i simboli terminali.

Come verificare se una stringa appartiene ad un linguaggio

Una stringa appartiene ad un linguaggio se e solo se la riesco ad ottenere con una sequenza di simboli che parte da X e aumenta sostituendo al non terminale una delle stringhe della produzione relative a quel terminale.

Esempio:

Definito un linguaggio con la produzione: {X ::= 0 | 0Y, Y ::= 1X}

01010 appartiene al linguaggio perchè può essere costruito in questo modo: X → 0Y → 01X → 010Y → 0101X → **01010**.

Mentre 000 non appartiene al linguaggio.

Grammatiche BNF ambigue

Una grammatica BNF è ambigua se permette di costruire la stessa stringa seguendo due combinazioni di terminali diverse. Non ci interessano però solo grammatiche non ambigue.

Esempio:

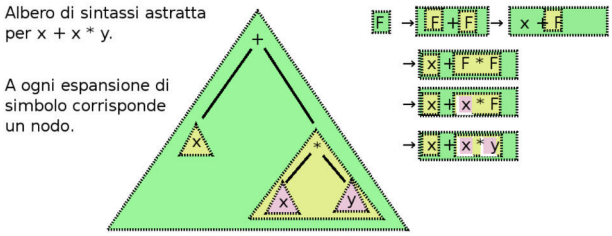
Nel linguaggio naturale la stringa “la vecchia porta la sbarra” può essere costruita in due modi:

- La vecchia porta la sbarra
- La vecchia porta la sbarra

Albero di sintassi astratta

La struttura ricorsiva che permette di ricavare stringhe a partire dall’insieme di produzioni è rappresentabile tramite un albero capovolto chiamato **albero di sintassi**, le quali foglie sono simboli terminali.

Nell’albero di sintassi astratta ritroviamo strutture che si ripetono, ovvero sottoalberi che hanno la stessa struttura degli alberi che li contengono. Questa **ripetizione di pattern** è che ciò che costituisce le fondamenta dell’informatica, la quale permette di eseguire lo stesso codice su input di dimensione diversa. Per esempio è possibile utilizzare la stessa sequenza di codice sia per effettuare operazioni su un singolo dato in input che su molteplici dati, con la sola condizione che il dato sia dello stesso tipo (pensa agli array).



▼ 5.2 - Sintassi in pseudo-linguaggio di programmazione

Lo pseudo-linguaggio di programmazione che utilizzeremo presenta le seguenti proprietà:

- **Funzionale**: le funzioni sono dati come gli altri, è possibile prenderle in input, darle in output a funzioni, memorizzarle in variabili ecc.
- **Puro**: non presenta mutazione, ovvero la possibilità di modificare dei valori, ad esempio come avviene per le variabili.
- **Non tipato**: non esistono i tipi (interi, stringhe ecc.).
- **Assenza di cicli**, replicabili con funzioni ricorsive.

Sintassi della definizione di funzioni unarie

$f(w) = corpo$

- **f**: nome della funzione.

- **w**: stringa sull'alfabeto del linguaggio formato da simboli terminali (costanti, costruttori) e da parametri formali (variabili).
- **corpo**: espressione in pseudo-codice che può utilizzare:
  - i parametri formali e tutte le costanti.
  - espressioni if-then-else.
  - chiamate di funzione della forma g(w).

Pattern matching

Sia p una stringa di terminali e w una stringa di terminali e non terminali, p **fa match** con q se sostituisco ad ogni non terminale di w una sottostringa di p.

- Esempio
  - abba fa match con aXa se sostituisco bb a X
  - 3 + 4 * 2 fa match con X +Y se sostituisco 2 a X e 4 * 2 a Y
  - abba non fa match con bXb

Una **chiamata di funzione** lavora tramite pattern matching.

- Esempio

  Consideriamo una funzione ricorsiva definita in questo modo:

  $f([]) = 0$
  $f(X :: Y) = 1 + f(Y)$

  ▼ Esempio di esecuzione della funzione

  f(2 :: 3 :: [])

  → 1+ f(3 :: [])

  → 1 + (1 + f([]))

  → 1 + (1 + 0)

  → 2

  Pattern matching per la funzione data:

  $\begin{cases} 2 :: 3 :: [] \neq [] \\ 2 :: 3 :: [] = X :: Y \end{cases}$ (proprietà associativa a destra: 2::3::[] = X::Y)

  $\begin{cases} [] = [] \\ [] \neq X :: Y \end{cases}$

▼ **5.3 - Ricorsione strutturale**

Funzioni con ricorsione strutturale

Limiteremo il linguaggio alla r**icorsione strutturale**, la quale si compone di **casi atomici**, nei quali il problema è semplice e si risolve direttamente, e di **casi composti**, nei quali si risolve prima il problema sulle componenti fino ad arrivare in modo ricorsivo ai casi atomici e, una volta ottenute le soluzioni le si utilizzano nel caso composto.

Questo limite che ci imponiamo alla ricorsione strutturale non ci permette di poter esprimere in modo completo qualsiasi altro linguaggio di programmazione, cosa possibile con l'utilizzo completo delle sole funzioni ricorsive.

- Esempio di funzione con ricorsione strutturale

> Funzione che calcola il numero di operandi in una espressione formata solo dagli operatori $+$ e x.

Descrizione dei dati: $F ::= x|F + F|F * F$

Struttura della funzione:

- $size(x) = x$
- $size(F_1 + F_2) = size(F_1) + size(F_2)$
- $size(F_1 * F_2) = size(F_1) + size(F_2)$

Gli **errori** che possono verificarsi nell'utilizzo di funzioni con ricorsione strutturale presentano le seguenti cause:

- Struttura errata della funzione.
- La funzione non considera tutte le possibili produzioni del dato in input.
- Le chiamate ricorsive non avvengono sulle sottoformule del dato in input.

Funzioni n-arie (funzioni con più variabili)

Le **funzioni n-arie** sono funzioni che prendono in input più di un argomento.

Una funzione n-aria viene scritta per ricorsione strutturale su uno dei suoi argomenti, il quale dunque deve soddisfarre i vincoli della ricorsione strutturale su di esso.

- Esempio

> Funzione che calcola il numero degli elementi presenti in una lista.

- $size(X :: L, A) = size(L, A + 1)$
- $size([], A) = A$

Per ogni elemento della lista data in input, sulla quale la funzione utilizza la ricorsione strutturale, viene aggiunto 1 all'**accumulatore** A e data in input la lista senza l'elemento già contato, fino a che la lista non diventa vuota: [].

Utilizzo della ricorsione struttuale

Parte del lavoro svolto per creare una funzione per ricorsione struttuale è **meccanico**, ovvero deve essere svolto per forza di cose e dunque ripetuto in ogni funzione di questo tipo. L'altra parte del lavoro è invece **non meccanico**, e prevede un ragionamento in grado di fare scelte giuste al fine di far funzionare in modo corretto la funzione ricorsiva.

Inoltre, per problemi più complessi, occorre individuare dei sottoproblemi e creare delle sottofunzioni che li risolvino al fine di poterli richiamare dalla funzione principale e risolvere correttamente il problema.

▼ 5.4 - Induzione strutturale, dimostrazioni su un codice ricorsivo

Solitamente non viene dimostrata direttamente la correttezza di tutto il codice in quanto è un procedimento molto complesso, ma alcune proprietà più semplici del codice stesso al fine di fornirci una garanzia migliore che il codice è corretto rispetto all'effettuare dei semplici test su di esso.

Teorema

> Tutte le funzioni definite per ricorsione strutturale convergono su ogni input.

Dimostrazione: ad ogni chiamata ricorsiva, siccome viene fatta espandendo un non terminale della stringa in input, il numero di espansioni di non terminali necessarie per arrivare alla stringa in input cala di 1 per, prima o poi, arrivare a 0.

Induzione strutturale

L'induzione struttuale consiste nella tecnica utilizzata per dimostrare proprietà di funzioni che utilizzano la ricorsione strutturale.

La dimostrazione che un certo codice gode di una proprietà su tutte le possibili stringhe date in input generabili da un linguaggio generato con BNF può essere data seguendo le seguenti indicazioni:

1. Fare una sotto-dimostrazione per ogni produzione che genera la stringhe date in input.

   Nota bene: occorre andare per induzione sul parametro nel quale la funzione va per ricorsione, in modo da rendere possibile poi la fase di semplificazione.

2. In ogni sotto-dimostrazione possiamo assumere che ciò che vogliamo dimostrare valga già per tutte le sotto-formule della stringa data in input (come accade per le chiamate ricorsive nella ricorsione struttuale). Queste vengono chiamate ipotesi induttive. (Matita: suppose … (II)).

   È possibile fare ciò poichè l'induzione strutturale ci garantisce che anche sulle sottoparti dell'input verrà richiamata la funzione sulle sottoparti del sottoinput, fino ad arrivare ad un caso base, dove abbiamo dimostrato che la proprietà è vera.

   Le ipotesi induttive serviranno solo nel caso in cui si sta dimostrando una proprietà per una funzione ricorsiva, in quanto le dimostrazioni seguono la struttura delle funzioni, dunque sono ricorsive se le funzioni sono ricorsive.

3. È possibile semplificare ciò che si deve provare in ogni sotto-dimostrazione sostituendo quello che fa la funzione nel sottocaso richiamato. (Matita: that is equivalent to)

Esempio:

- Funzione:

  $X ::= \epsilon | aXa | bxb$

  $length(\epsilon) = 0$
  $length(aXa) = 2 + length(X)$
  $length(bXb) = 2 + length(X)$

- Dimostrazione che per ogni X la $length(X)$ non è dispari:
  - caso ϵ:

    Dobbiamo dimostrare che $length(\epsilon)$ non è dispari. Sappiamo che $length(\epsilon) = 0$, dunque, visto che 0 non è dispari, $length(\epsilon)$ non è dispari.

  - caso aXa:

    Dobbiamo dimostrare che $length(aXa)$ non è dispari. Sappiamo che $length(aXa) = 2 + length(X)$. Sappiamo inoltre che, per ipotesi induttiva, $length(X)$ non è dispari, dunque, siccome 2 + (numero non dispari) non è dispari, allora abbiamo dimostrato che $length(aXa)$ non è dispari.

  - caso bXb: analogo.

In questo caso abbiamo dato per scontato che 2 + (numero non dispari) non è dispari, ma nel caso in cui per dimostrare la tesi finale occorre svolgere dimostrazioni intermedie che si chiamano **lemmi**.

Solitamente occorre creare dei lemmi quando la dimostrazione del teorema richiede l'operazione di ricorsione su due variabili differenti (nel lemma si va per ricorsione sull'altra variabile rispetto a quella scelta inizialmente nel teorema). Bisogna fare attenzione a creare sottoproblemi più semplici di quello di partenza, ovvero dei sottoproblemi risolvibili effettuando la ricorsione su meno variabili del problema di partenza.

Esempio:

- $\forall$ m, n = plus m n = plus n m.

- Occorre andare per ricorsione sia su m che su n, in quanto il primo plus va in ricorsione su m o n, in base a come viene definita la funzione plus, mentre il restante plus va in ricorsione sull'altra variabile.

Dimostrare un if, then, else

Se durante la dimostrazione si arriva ad un livello di semplificazione in cui bisogna dimostrare un **if, then, else** occorre prima di tutto creare un lemma che affermi che un booleano è uguale a true o uguale a false, dopodichè nella dimostrazione si ottiene un or, dimostrabile andando per casi.

▼ 5.5 - Esercizi di ricorsione strutturale

▼ 4.3.1 - Liste

```
-- -- list A ::= [] | A : list A

-- -- Problema 5: dato un elemento e una lista di liste, restituire una lista di tutte le liste della lista
-- -- Funzione: insert_head_in_list_list h ll
-- -- Esempio: insert_head_in_list_list 1 ((2 : 3 : []) : (4 : 2 : []) : []) =
-- -- ((1 : 2 : 3 : []) : (1 : 4 : 2 : []) : [])

insert_head_in_list_list h [] = []
insert_head_in_list_list h (l : ll) = (h : l) : (insert_head_in_list_list h ll)

-- Problema 4: date due liste, concatenarle
-- Funzione: concatenate l1 l2
-- Esempio: concatenate (1 : 3 : 2 : []) (4 : 2 : 5 : []) =
-- 1 : 3 : 2 : 4 : 2 : 5 : []

concatenate [] l2 = l2
concatenate (h : t) l2 = h : (concatenate t l2)

-- Problema 3: dati un elemento e una lista, restituire una lista di liste nelle quali l'elemento è stato in
-- Funzione: insert_everywhere_in_list x l
-- Esempio: insert_everywhere_in_list 1 (2 : 3 : []) =
-- (1 : 2 : 3 : []) : (2 : 1 : 3 : []) : (2 : 3 : 1 : []) : []

insert_everywhere_in_list x [] = (x : []) : []
insert_everywhere_in_list x (h : t) =
  (x : h : t) : (insert_head_in_list_list h (insert_everywhere_in_list x t))

-- Problema 2: dato un elemento e una lista di liste, restituire una lista di liste ottenuta da quella in in
-- Funzione: insert_everywhere_in_list_list x ll
-- Esempio: insert_everywhere_in_list_list 1 ((2 : 3 : []) : (3 : 2 : []) : []) =
-- (1 : 2 : 3 : []) : (2 : 1 : 3 : []) : (2 : 3 : 1 : []) :
-- (1 : 3 : 2 : []) : (3 : 1 : 2 : []) : (3 : 2 : 1 : []) : []

insert_everywhere_in_list_list x [] = []
insert_everywhere_in_list_list x (l : ll) =
  concatenate
    (insert_everywhere_in_list x l)
    (insert_everywhere_in_list_list x ll)

-- Problema 1: data una lista, calcolare l'insieme (lista di liste) di tutte le permutazioni della lista in
-- Funzione: permut l
-- Esempio: permut (1 : 2 : 3 : []) =
-- (1 : 2 : 3 : []) : (2 : 1 : 3 : []) : (2 : 3 : 1 : []) :
-- (1 : 3 : 2 : []) : (3 : 1 : 2 : []) : (3 : 2 : 1 : []) : []

permut [] = [] : []
permut (h : t) = insert_everywhere_in_list_list h (permut t)

-----------------------------------------------------

-- Problema 3: data una sequenza di numeri, restituirne la prima sottosequenza di numeri crescenti
-- Funzione: first_growing x l
-- Esempio: first_growing 2 (3 : 4 : 1 : 5 : 7 : 8 : 2 : []) = 2 : 3 : 4 : []

first_growing x [] = x : []
first_growing x (h : t) =
  if (x < h)
    then (x : (first_growing h t))
    else x : []

-- Problema 2: data una lista, restituirne la lunghezza
-- Funzione: len l
```

```
-- Esempio: len (3 : 2 : 7 : 1 : []) = 4

len [] = 0
len (h : t) = 1 + len t

-- Problema 1: data una sequenza di numeri, trovare la sottosequenza di numeri crescenti di lunghezza massima
-- Funzione: growing l
-- Esempio: growing (3 : 4 : 1 : 5 : 7 : 2 : 2 : []) = 1 : 5 : 7 : []

growing [] = []
growing (h : t) =
  if len (first_growing h t) > len (growing t)
    then (first_growing h t)
    else growing t

-----------------------------------------------------

-- Problema 3: data una coppia (x, y), restituirne il secondo elemento
-- Funzione: second (x, y)
-- Esempio: second (2, 3) = 2

second (x, y) = y

-- Problema 3: data una coppia (x, y), restituirne il primo elemento
-- Funzione: first (x, y)
-- Esempio: first (2, 3) = 2

first (x, y) = x

-- Problema 2: data una lista di numeri, restituire una coppia (len, n) dove len è la lunghezza delle sottoli
-- Funzione: infoincr l
-- Esempio: infoincr (1 : 2 : 3 : 2 : 4 : 0 : 1 : 2 : 0 : []) = (3, 2)

infoincr [] = (0, 1)
infoincr (h : t) =
  if (len (first_growing h t) > first(infoincr t)) then (len (first_growing h t), 1)
  else if (len (first_growing h t) == first(infoincr t)) then (first(infoincr t), second(infoincr t) + 1)
  else infoincr t

-- Problema 1: data una lista di numeri, restituire il numero di sottoliste crescenti di lunghezza massimale
-- Soluzione: numincr l
-- Esempio: numincr (1 : 2 : 3 : 2 : 4 : 0 : 1 : 2 : 0 : []) = 2

numincr l = second (infoincr l)
```

▼ 4.3.2 - Numeri

```
-- nat ::= O | S nat
-- es. 3 = S (S (S O))

data Nat = O | S Nat deriving (Show)

-- Problema: dati due naturali, sommarli
-- Soluzione: plus m n
-- Esempio: plus (S (S (S O))) (S (S O)) = S (S (S (S (S O))))

plus m O = m
plus m (S n) = S (plus m n)

-- Problema: dati due naturali, moltiplicarli
-- Soluzione: mult m n
-- Esempio: mult (S (S O)) (S (S (S O))) = S (S (S (S (S (S O)))))

mult m O = O
mult m (S n) = plus m (mult m n)
```

▼ 4.3.3 - Alberi

```
-- TreeL ::= Node TreeL TreeL ! Leaf Int

data TreeL = Node TreeL TreeL | Leaf Int deriving Show

-- Problema 3: dato un TreeL, restituire il valore della foglia più a sx
```

```
-- Funzione: sxLeaf t
-- Esempio: sxLeaf (Node (Node (Leaf 1) (Leaf 3)) (Node (Node (Leaf 5) (Leaf 7)) = 7

sxLeaf (Leaf n) = n
sxLeaf (Node t1 t2) = sxLeaf t2

-- Problema 2: dato un TreeL, restituire il valore della foglia più a dx
-- Funzione: dxLeaf t
-- Esempio: dxLeaf (Node (Node (Leaf 1) (Leaf 3)) (Node (Node (Leaf 5) (Leaf 7)) = 7

dxLeaf (Leaf n) = n
dxLeaf (Node t1 t2) = dxLeaf t2

-- Problema 1: dato un TreeL, restituire True sse la sequenza di foglie da sx a dx è crescente in senso stre
-- Funzione: incr t
-- Esempio: incr (Node (Node (Leaf 1) (Leaf 3)) (Node (Node (Leaf 5) (Leaf 7)) (Node (Leaf 8) (Leaf 9)))) =

incr (Leaf n) = True
incr (Node t1 t2) = incr t1 && incr t2 && (dxLeaf t1) < (sxLeaf t2)
```

▼ 4.3.4 - Laboratorio

```
(* Esercizio 1
   ===========

   Definire il seguente linguaggio (o tipo) di espressioni riempiendo gli spazi.

   Expr :: "Zero" | "One" | "Minus" Expr | "Plus" Expr Expr | "Mult" Expr Expr
*)
inductive Expr : Type[0] ≝
| Zero: Expr
| One: Expr
| Minus: Expr → Expr
| Plus: Expr → Expr → Expr
| Mult: Expr → Expr → Expr
.

(* La prossima linea è un test per verificare se la definizione data sia
   probabilmente corretta. Essa definisce `test_Expr` come un'abbreviazione
   dell'espressione `Mult Zero (Plus (Minus One) Zero)`, verificando inoltre
   che l'espressione soddisfi i vincoli di tipo dichiarati sopra. Eseguitela. *)

definition test_Expr : Expr ≝ Mult Zero (Plus (Minus One) Zero).

(* Come esercizio, provate a definire espressioni che siano scorrette rispetto
   alla grammatica/sistema di tipi. Per esempio, scommentate la seguenti
   righe e osservate i messaggi di errore:

definition bad_Expr1 : Expr ≝ Mult Zero.
definition bad_Expr2 : Expr ≝ Mult Zero Zero Zero.
definition bad_Expr3 : Expr ≝ Mult Zero Plus.
*)

(* Esercizio 2
   ===========

   Definire il linguaggio (o tipo) dei numeri naturali.

   nat ::= "O" | "S" nat
*)

inductive nat : Type[0] ≝
 O : nat
 | S : nat → nat
.

definition one : nat ≝ S O.
definition two : nat ≝ S (S O).
definition three : nat ≝ S (S (S O)).

(* Esercizio 3
   ===========

   Definire il linguaggio (o tipo) delle liste di numeri naturali.

   list_nat ::= "Nil" | "Cons" nat list_nat
```

```
   dove Nil sta per lista vuota e Cons aggiunge in testa un numero naturale a
   una lista di numeri naturali.

   Per esempio, `Cons O (Cons (S O) (Cons (S (S O)) Nil))` rappresenta la lista
   `[1,2,3]`.
*)

inductive list_nat : Type[0] ≝
 Nil : list_nat
 | Cons : nat → list_nat → list_nat
.

(* La seguente lista contiene 1,2,3 *)
definition one_two_three : list_nat ≝ Cons one (Cons two (Cons three Nil)).

(* Completate la seguente definizione di una lista contenente due uni. *)

definition one_one : list_nat ≝ Cons one (Cons one Nil).


(* Osservazione
   ===========

   Osservare come viene definita la somma di due numeri in Matita per
   ricorsione strutturale sul primo.

   plus O m = m
   plus (S x) m = S (plus x m) *)

let rec plus n m on n ≝
 match n with
 [ O ⇒ m
 | S x ⇒ S (plus x m) ].

(* Provate a introdurre degli errori nella ricorsione strutturale. Per esempio,
   omettete uno dei casi o fate chiamate ricorsive non strutturali e osservate
   i messaggi di errore di Matita. *)

(* Per testare la definizione, possiamo dimostrare alcuni semplici teoremi la
   cui prova consiste semplicemente nell'effettuare i calcoli. *)
theorem test_plus: plus one two = three.
done. qed.

(* Esercizio 4
   ===========

   Completare la seguente definizione, per ricorsione strutturale, della
   funzione `size_E` che calcola la dimensione di un'espressione in numero
   di simboli.

   size_E Zero = 1
   size_E One = 1
   size_E (Minus E) = 1 + size_E E
   ...
*)
let rec size_E E on E ≝
 match E with
  [ Zero ⇒ one
  | One ⇒ one
  | Minus E ⇒ plus one (size_E E)
  | Plus E1 E2 ⇒ plus one (plus (size_E E1) (size_E E2))
  | Mult E1 E2 ⇒ plus one (plus (size_E E1) (size_E E2))
  ]
.

theorem test_size_E : size_E test_Expr = plus three three.
done. qed.

(* Esercizio 5
   ===========

   Definire in Matita la funzione `sum` che, data una `list_nat`, calcoli la
   somma di tutti i numeri contenuti nella lista. Per esempio,
   `sum one_two_three` deve calcolare sei.
*)

definition zero : nat ≝ O.
```

```
let rec sum L on L ≝
 match L with
 [ Nil ⇒ zero
 | Cons N TL ⇒ plus N (sum TL)]
.

theorem test_sum : sum one_two_three = plus three three.
done. qed.



(* Esercizio 6
   ===========

   Definire la funzione binaria `append` che, date due `list_nat` restituisca la
   `list_nat` ottenuta scrivendo in ordine prima i numeri della prima lista in
   input e poi quelli della seconda.

   Per esempio, `append (Cons one (Cons two Nil)) (Cons 0 Nil)` deve restituire
   `Cons one (Cons two (Cons 0 nil))`. *)
let rec append lista1 lista2 on lista1 ≝
 match lista1 with
 [ Nil ⇒ lista2
 | Cons N TL ⇒ append TL (Cons N lista2)]
.

theorem test_append : append (Cons one Nil) (Cons two (Cons three Nil)) = one_two_three.
done. qed.
```

▼ 5.6 - Esercizi di induzione strutturale

▼ Lunghezza di una concatenazione, tutti gli elementi di due liste nella concatenazione

```
include "arithmetics/nat.ma".

(*
list ::= E | C n list
*)

inductive list : Type[0] ≝
    E : list
  | C : nat → list → list.

(*
concat E l2 = l2
concat (C hd tl) l2 = C hd (concat tl l2)
*)

let rec concat l1 l2 on l1 ≝
  match l1 with
  [ E ⇒ l2
  | C hd tl ⇒ C hd (concat tl l2)
  ].

(*
len E = 0
len (C hd tl) = 1 + len tl
*)

let rec len l on l ≝
  match l with
  [ E ⇒ 0
  | C hd tl ⇒ 1 + len tl
  ].

theorem len_concat:
  ∀ l1, l2. len (concat l1 l2) = len l1 + len l2.
assume l1 : list
we proceed by induction on l1 to prove
  (∀ l2. len (concat l1 l2) = len l1 + len l2).
case E
  we need to prove
    (∀ l2. len (concat E l2) = len E + len l2)
  that is equivalent to
    (∀ l2. len l2 = 0 + len l2)
  that is equivalent to
```

```
    (∀ l2. len l2 = len l2)
  done

case C (h : nat) (t : list)
  suppose
    (∀ l2. len (concat t l2) = len t + len l2) (II)
  we need to prove
    (∀ l2. len (concat (C h t) l2) = len (C h t) + len l2)
  that is equivalent to
    (∀ l2. 1 + len (concat t l2) = 1 + len t + len l2)
  assume l2 : list
  by II
done
qed.

(*
In n E ⇔ False
In n (C hd tl) ⇔ n = hd ∨ In n tl
*)

let rec In n l on l ≝
  match l with
  [ E ⇒ False
  | C hd tl ⇒ n = hd ∨ In n tl
  ].

theorem In1:
  ∀ l1, l2, n. In n l1 → In n (concat l1 l2).
assume l1: list
we proceed by induction on l1 to prove
  (∀ l2, n. In n l1 → In n (concat l1 l2))
case E
  we need to prove
    (∀ l2, n. In n E → In n (concat E l2))
  that is equivalent to
    (∀ l2, n. False → In n l2)
  assume l2: list
  assume n: nat
  suppose False (Abs)
  cases Abs (* questa riga dall'assurdo conclude qualsiasi cosa *)

case C (hd : nat) (tl : list)
  suppose
    (∀ l2, n. In n tl → In n (concat tl l2)) (II)
  we need to prove
    (∀ l2, n. In n (C hd tl) → In n (concat (C hd tl) l2))
  that is equivalent to
    (∀ l2, n. n = hd ∨ In n tl → In n (C hd (concat tl l2)))
  that is equivalent to
    (∀ l2, n. n = hd ∨ In n tl → n = hd ∨ In n (concat tl l2))
  assume l2: list
  assume n: nat
  suppose (n = hd ∨ In n tl) (H)
  we proceed by cases on H to prove
    (n = hd ∨ In n (concat tl l2))
  case or_introl
    suppose (n = hd) (EQ)
    by or_introl, EQ
  done
  case or_intror
    suppose (In n tl) (K)
    by or_intror, II, K
  done
qed.

theorem In2:
  ∀ l1, l2, n. In n l2 → In n (concat l1 l2).
assume l1: list
we proceed by induction on l1 to prove
  (∀ l2, n. In n l2 → In n (concat l1 l2))
case E
  we need to prove
    (∀ l2, n. In n l2 → In n (concat E l2))
  that is equivalent to
    (∀ l2, n. In n l2 → In n l2)
  done

case C (hd : nat) (tl : list)
  suppose
```

```
    (∀ l2, n. In n l2 → In n (concat tl l2)) (II)
  we need to prove
    (∀ l2, n. In n l2 → In n (concat (C hd tl) l2))
  that is equivalent to
    (∀ l2, n. In n l2 → In n (C hd (concat tl l2)))
  that is equivalent to
    (∀ l2, n. In n l2 → n = hd ∨ In n (concat tl l2))
  assume l2: list
  assume n: nat
  suppose (In n l2) (H)
  by or_intror, H, II
  done
qed.

theorem In12:
  ∀ l1, l2, n. In n (concat l1 l2) → In n l1 ∨ In n l2.
assume l1: list
we proceed by induction on l1 to prove
  (∀ l2, n. In n (concat l1 l2) → In n l1 ∨ In n l2)
case E
  we need to prove
    ((∀ l2, n. In n (concat E l2) → In n E ∨ In n l2))
  that is equivalent to
    ((∀ l2, n. In n l2 → False ∨ In n l2))
  by or_intror
  done
case C (hd: nat) (tl: list)
  suppose
    (∀ l2, n. In n (concat tl l2) → In n tl ∨ In n l2) (II)
  we need to prove
    (∀ l2, n. In n (concat (C hd tl) l2) → In n (C hd tl) ∨ In n l2)
  that is equivalent to
    (∀ l2, n. In n (C hd (concat tl l2)) →
      (n = hd ∨ In n tl) ∨ In n l2)
  that is equivalent to
    (∀ l2, n. n = hd ∨ In n (concat tl l2) →
      (n = hd ∨ In n tl) ∨ In n l2)
  assume l2: list
  assume n: nat
  suppose (n = hd ∨ In n (concat tl l2)) (OR)
  we proceed by cases on OR to prove
    ((n = hd ∨ In n tl) ∨ In n l2)
  case or_introl
    suppose (n = hd) (EQ)
    by or_introl, EQ
  done

  case or_intror
    suppose (In n (concat tl l2)) (K)
    by II, K we proved (In n tl ∨ In n l2) (OR2)
    we proceed by cases on OR2 to prove
      ((n = hd ∨ In n tl) ∨ In n l2)
    case or_introl
      suppose (In n tl) (InT)
      by or_introl, or_intror, InT
    done
    case or_intror
      suppose (In n l2) (Inl2)
      by or_intror, Inl2
    done
qed.
```

▼ Proprietà commutativa dell’addizione (con lemmi)

```
include "arithmetics/nat.ma".

(* N ::= O | S N *)

inductive N : Type[0] ≝
    O: N
  | S: N → N.

(*
plus E m = m
plus (S n) m = S (plus n m)
*)
```

```
let rec plus n m on n ≝
  match n with
  [ O ⇒ m
  | S x ⇒ S (plus x m)
  ].

lemma plus_O:
  ∀ m. m = plus m O.
assume m: N
we proceed by induction on m to prove
  (m = plus m O)
case O
  we need to prove
    (O = plus O O)
  that is equivalent to
    (O = O)
  done
case S (x : N)
  suppose (x = plus x O) (II)
  we need to prove
    (S x = plus (S x) O)
  that is equivalent to
    (S x = S (plus x O))
  by II
done
qed.

lemma plus_s:
  ∀ m, x. S (plus m x) = plus m (S x).
assume m: N
we proceed by induction on m to prove
  (∀ x. S (plus m x) = plus m (S x))
case O
  we need to prove
    (∀ x. S (plus O x) = plus O (S x))
  that is equivalent to
    (∀ x. S x = S x)
done

case S (y : N)
  suppose (∀ x. S (plus y x) = plus y (S x)) (II)
  we need to prove
    (∀ x. S (plus (S y) x) = plus (S y) (S x))
  that is equivalent to
    (∀ x. S (S (plus y x)) = S (plus y (S x)))
  by II
done
qed.

theorem comm_plus:
  ∀ n, m. plus n m = plus m n.
assume n: N
we proceed by induction on n to prove
  (∀ m. plus n m = plus m n)
case O
  we need to prove
    (∀ m. plus O m = plus m O)
  that is equivalent to
    (∀ m. m = plus m O)
  by plus_O
done
case S (x : N)
  suppose (∀ m. plus x m = plus m x) (II)
  we need to prove (∀ m. plus (S x) m = plus m (S x))
  that is equivalent to
    (∀ m. S (plus x m) = plus m (S x))
  by plus_s, II
  done
qed.
```

▼ Laboratorio

```
(* Esercizio 1
   ===========
```

```
   Dimostrare l'associatività della somma per induzione strutturale su x.
*)
theorem plus_assoc: ∀x,y,z. plus x (plus y z) = plus (plus x y) z.
 (* Possiamo iniziare fissando una volta per tutte le variabili x,y,z
    A lezione vedremo il perchè. *)
 assume x : nat
 assume y : nat
 assume z : nat
 we proceed by induction on x to prove (plus x (plus y z) = plus (plus x y) z)
 case O
  (* Scriviamo cosa deve essere dimostrato e a cosa si riduce eseguendo le
     definizioni. *)
  we need to prove (plus O (plus y z) = plus (plus O y) z)
  that is equivalent to (plus y z = plus y z)
  (* done significa ovvio *)
  done
 case S (w: nat)
  (* Chiamiamo l'ipotesi induttiva IH e scriviamo cosa afferma
     Ricordate: altro non è che la chiamata ricorsiva su w. *)
  by induction hypothesis we know (plus w (plus y z) = plus (plus w y) z) (IH)
  we need to prove (plus (S w) (plus y z) = plus (plus (S w) y) z)
  that is equivalent to (S (plus w (plus y z)) = plus (S (plus w y)) z)
  (* by IH done significa ovvio considerando l'ipotesi IH *)
  by IH done
qed.

(* Esercizio 2
   ===========

   Definire il linguaggio degli alberi binari (= dove ogni nodo che non è una
   foglia ha esattamente due figli) le cui foglie siano numeri naturali.

   tree_nat ::= "Leaf" nat | "Node" nat nat
*)

inductive tree_nat : Type[0] ≝
   Leaf : nat → tree_nat
 | Node : tree_nat → tree_nat → tree_nat.

(* Il seguente albero binario ha due foglie, entrambe contenenti uni. *)
definition one_one_tree : tree_nat ≝ Node (Leaf one) (Leaf one).

(* Definite l'albero       /\
                          0 /\
                           1  2  *)
definition zero_one_two_tree : tree_nat ≝
 Node (Leaf O) (Node (Leaf one) (Leaf two)).


(* Esercizio 3
   ===========

   Definire la funzione `rightmost` che, dato un `tree_nat`, restituisca il
   naturale contenuto nella foglia più a destra nell'albero. *)

let rec rightmost tree on tree ≝
  match tree with
  [ Leaf n ⇒ n
  | Node tree1 tree2 ⇒ rightmost tree2
  ].

theorem test_rightmost : rightmost zero_one_two_tree = two.
done. qed.


(* Esercizio 4
   ============

   Definire la funzione `visit` che, dato un `tree_nat`, calcoli la `list_nat`
   che contiene tutti i numeri presenti nelle foglie dell'albero in input,
   nell'ordine in cui compaiono nell'albero da sinistra a destra.

   Suggerimento: per definire tree_nat usare la funzione `append` già definita
   in precedenza.

   Esempio: `visit zero_one_two_tree = Cons O (Cons one (Cons two Nil))`.
*)

let rec visit T on T ≝
```

```
 match T with
 [ Leaf n ⇒ Cons n Nil
 | Node T1 T2 ⇒ append (visit T1) (visit T2)
 ].

theorem test_visit : visit zero_one_two_tree = Cons O (Cons one (Cons two Nil)).
done. qed.


(* Esercizio 5
   ===========

   La somma di tutti i numeri nella concatenazione di due liste è uguale
   alla somma delle somme di tutti i numeri nelle due liste. *)

theorem sum_append: ∀L1,L2. sum (append L1 L2) = plus (sum L1) (sum L2).
 assume L1 : list_nat
 assume L2 : list_nat
 we proceed by induction on L1 to prove (sum (append L1 L2) = plus (sum L1) (sum L2))
 case Nil
  we need to prove (sum (append Nil L2) = plus (sum Nil) (sum L2))
  that is equivalent to (sum L2 = plus O (sum L2))
  done
  case Cons (N: nat) (L: list_nat)
  by induction hypothesis we know (sum (append L L2) = plus (sum L) (sum L2)) (IH)
  we need to prove (sum (append (Cons N L) L2) = plus (sum (Cons N L)) (sum L2))
  that is equivalent to (sum (Cons N (append L L2)) = plus (plus N (sum L)) (sum L2))
  that is equivalent to (plus N (sum(append L L2)) = plus (plus N (sum L)) (sum L2))
  (* Per concludere servono sia l'ipotesi induttiva IH che il teorema plus_assoc
     dimostrato prima. Convincetevene

     Nota: se omettete IH, plus_assoc o entrambi Matita ci riesce lo stesso
     Rendere stupido un sistema intelligente è complicato... Tuttavia non
     abusatene: quando scrivete done cercate di avere chiaro perchè il teorema
     è ovvio e se non vi è chiaro, chiedete. *)
  by IH, plus_assoc done
qed.

(* La funzione `plusT` che, dato un `tree_nat`, ne restituisce la
   somma di tutte le foglie. *)
let rec plusT T on T ≝
 match T with
 [ Leaf n ⇒ n
 | Node t1 t2 ⇒ plus (plusT t1) (plusT t2)
 ].


(* Esercizio 6
   ===========

   Iniziare a fare l'esercizio 7, commentando quel poco che c'è dell'esercizio 6
   Nel caso base vi ritroverete, dopo la semplificazione, a dover dimostrare un
   lemma non ovvio. Tornate quindi all'esercizio 3 che consiste nell'enunciare e
   dimostrare il lemma. *)

lemma plus_O: ∀N. N = plus N O.
  assume N : nat
  we proceed by induction on N to prove (N = plus N O)
  case O
    we need to prove (O = plus O O)
    that is equivalent to (O=O)
    done
  case S (x : nat)
    by induction hypothesis we know (x = plus x O) (II)
    we need to prove (S x = plus (S x) O)
    that is equivalent to (S x = S (plus x O))
    by II
    done
qed.
(* Esercizio 7
   ===========

   Dimostriamo che la `plusT` è equivalente a calcolare la `sum` sul risultato
   di una `visit`. *)


theorem plusT_sum_visit: ∀T. plusT T = sum (visit T).
 assume T : tree_nat
 we proceed by induction on T to prove (plusT T = sum (visit T))
```

```
 case Leaf (N : nat)
  we need to prove (plusT (Leaf N) = sum (visit (Leaf N)))
  that is equivalent to (N = sum (Cons N Nil))
  that is equivalent to (N = plus N (sum Nil))
  that is equivalent to (N = plus N O)
  (* Ciò che dobbiamo dimostrare non è ovvio (perchè?). Per proseguire,
     completate l'esercizio 6 enunciando e dimostrando il lemma che vi serve
     Una volta risolto l'esercizio 6, questo ramo diventa ovvio usando il lemma.*)
  by plus_O done
 case Node (T1:tree_nat) (T2:tree_nat)
  by induction hypothesis we know (plusT T1 = sum (visit T1)) (IH1)
  by induction hypothesis we know (plusT T2 = sum (visit T2)) (IH2)
  we need to prove (plusT (Node T1 T2)=sum (visit (Node T1 T2)))
  that is equivalent to (plus (plusT T1) (plusT T2) = sum (append (visit T1) (visit T2)))
  (* Oltre alla due ipotesi induttive, di quale altro lemma dimostrato in
     precedenza abbiamo bisogno per concludere la prova?*)
  by IH1,IH2,sum_append done
qed.


(* Un altro modo di calcolare la somma di due numeri: per ricorsione strutturale
   sul secondo argomento.

   plus' m O = m
   plus' m (S x) = S (plus' m x)
*)
let rec plus' m n on n ≝
 match n with
 [ O ⇒ m
 | S x ⇒ S (plus' m x) ].


(* Esercizio 8
   ============

   Dimostriamo l'equivalenza dei due metodi di calcolo
   Vi servirà un lemma: capite quale e dimostratelo
*)

lemma plus_O': ∀y. y = plus' O y.
  assume y : nat
  we proceed by induction on y to prove (y = plus' O y)
  case O
    we need to prove (O = plus' O O)
    that is equivalent to (O = O)
    done
  case S (n : nat)
    by induction hypothesis we know (n = plus' O n) (II)
    we need to prove (S n = plus' O (S n))
    that is equivalent to (S n = S (plus' O n))
    by II
    done
qed.

lemma plus_S': ∀y,z. S (plus' z y) = plus' (S z) y.
  assume y : nat
  we proceed by induction on y to prove (∀z. S (plus' z y) = plus' (S z) y)
  case O
    we need to prove (∀z. S(plus' z O) = plus' (S z) O)
    that is equivalent to (∀z. S z = S z)
    done
  case S (g : nat)
    by induction hypothesis we know (∀z. S(plus' z g) = plus' (S z) g) (II)
    we need to prove (∀z. S (plus' z (S g)) = plus' (S z) (S g))
    that is equivalent to (∀z. S (S (plus' z g)) = S (plus' (S z) g))
    by II
    done
qed.


theorem plus_plus': ∀x,y. plus x y = plus' x y.
 (* Nota: la dimostrazione è più facile se andate per induzione su y perchè
    potrete riciclare un lemma già dimostrato.
    Se andate per induzione su x vi verrà lo stesso, ma in tal caso avrete
    bisogno di due lemmi, ognuno dei quali non ancora dimostrati. *)
 assume x: nat
 we proceed by induction on x to prove (∀y. plus x y = plus' x y)
 case O
  we need to prove (∀y. plus O y = plus' O y)
```

```
  that is equivalent to (∀y. y = plus' O y)
  by plus_O'
  done
 case S (z:nat)
  by induction hypothesis we know (∀y. plus z y = plus' z y) (II)
  we need to prove (∀y. plus (S z) y = plus' (S z) y)
  that is equivalent to (∀y. S (plus z y) = plus' (S z) y)
  by II, plus_S'
  done
qed.


(* Esercizio 9: se finite prima o volete esercitarvi a casa
   ========================================================

   Dimostriamo l'equivalenza dei due metodi di calcolo plus e plus',
   questa volta per induzione sul primo argomento x. Avrete bisogno di uno o
   più lemmi, da scoprire. Ovviamente, NON è consentito usare quanto dimostrato
   all'esercizio precedente

lemma …
qed.

theorem plus_plus_new: ∀x,y. plus x y = plus' x y.
…
(* Es*)ercizio 10,11,…
   =================

   Volete esercitarvi a casa su altre dimostrazioni facili come queste?
   Ecco due buoni spunti:

   1) definite la funzione che inserisce un numero in
      coda a una lista e usatela per definire la funzione rev che restituisce
      la lista ottenuta leggendo la lista in input dalla fine all'inizio
      Esempio:
       rev (Cons 1 (Cons 2 (Cons 3 Nil))) = (Cons 3 (Cons 2 (Cons 1 Nil)))
      Poi dimostrate che ∀L. sum (rev L) = sum L
      Per riuscirci vi serviranno una cascata di lemmi intermedi da enunciare
      e dimostrare
   2) definite una funzione leq_nat che dati due numeri naturali ritorni true
      sse il primo è minore o uguale al secondo; usatela per scrivere una funzione
      che aggiunga un elemento in una lista ordinata di numeri;
      poi usatela quest'ultima per definire una funzione "sort" che ordina una lista
      di numeri. Dimostrate che l'algoritmo è corretto procedendo
      come segue:
      a) definite, per ricorsione strutturale, il predicato ``X appartiene
         alla lista L''
      b) dimostrate che X appartiene all'inserimento di Y nella lista ordinata
         L sse X è uguale a Y oppure appartiene a L
      c) dimostrate che se X appartiene alla lista L allora appartiene alla
         lista sort L
      d) dimostrate anche il viceversa
      e) definite, per ricorsione strutturale, il predicato ``X è ordinata''
      f) dimostrate che se L è ordinata lo è anche la lista ottenuta inserendo
         X in L
      g) dimostrate che per ogni L, sort L è ordinata

      Nota: a)-e) sono esercizi semplici. Anche g) è semplice se asserite f)
      come assioma. La dimostrazione di f) invece è più difficile e
      potrebbe richiede altri lemmi ausiliari quali la transitività del
      predicato leq_nat
*)
```

▼ **6.0 - Verità, conseguenza logica e sintassi della logica**

▼ **6.1 - Verità, conseguenza ed equivalenza logica**

Verità

Esistono verità associate al mondo sensibile, come quelle fisiche, chimiche ecc., ed esse sono definite tramite esperimenti ripetibili (es. se lascio un oggetto in aria questo cade). Queste verità sono però parametriche rispetto al mondo sensibile, ovvero cambiano in base alle proprietà del mondo in cui si trovano.

La logica però non si occupa di verità parametriche, ma di quelle assolute, ovvero associate ad aspetti immodificabili.

Esistono inoltre scienze “molli”, che ci parlano del mondo sensibile, come la chimica o la fisica, e scienze “dure”, che non ci parlano del mondo sensibile, come la matematica e l’informatica.

Verità in matematica

In una **teoria matematica**:

- Gli **assiomi** sono come le leggi fisiche, ci dicono tutto quello che supponiamo valere nei mondi che prendiamo in considerazione.
- Un **mondo** è una descrizione completa di un concetto di verità (tutto è definito). Gli assiomi tuttavia individuano una molteplicità di mondi, in quanto non permettono di definire tutto il mondo e molte cose possono essere sia vere che false, individuando in questo modo più mondi differenti tra loro.
- Un **modello matematico** della teoria è un qualunque mondo in cui tutti gli assiomi della teoria valgono.
- Una **proposizione** potrebbe essere sicuramente vera in tutti i modelli della teoria, sicuramente falsa oppure non definita, dunque o vera o falsa.

Esempio:

- Teoria:
  - Enti primitivi:
    - 0
    - $\leq$
  - Assiomi:
    - $\leq$ ha la proprietà simmetrica, transitiva e antiriflessiva.
    - $\forall n, 0 \leq n$.
- Proposizioni:
  - 1: $\forall x . x \leq 0 \Rightarrow x = 0$
  - 2: $\forall x, y . x \leq y \vee y \leq x$
- Modelli:
  - Modello 1:
    - Interpreto gli oggetti come numeri naturali.

        - 0 come numero 0.
        - $\leq$ come $\leq$ sui naturali.

        In questo modello possiamo osservare che tutti gli assiomi sono soddisfatti ed entrambe le proposizioni sono vere, dunque è un modello accettabile.

    - Modello 2:
        - Interpreto gli oggetti come numeri naturali.
        - 0 come numero 1.
        - $\leq$ come “divide”.

        Anche in questo modello possiamo osservare che tutti gli assiomi sono soddisfatti ma in questo caso solo la prima proposizione è vera. Nonostante ciò rimane un modello accettabile.

    In particolare osserviamo che in tutti i modelli derivanti da questa teoria matematica la prima proposizione è vera, mentre la seconda può essere vera o falsa. Questo per via del fatto che gli assiomi della teoria permettono di ricavare la prima proposizione (utilizzando la proprietà antiriflessiva del $\leq$ e il secondo assioma), mentre ciò non avviene per la seconda proposizione.

Conseguenza logica degli assiomi

Dal momento in cui una proposizione può essere indefinita non ha senso chiedersi se sia vera o falsa, in quanto non sappiamo in quale mondo valutarla. Ha senso invece chiedersi se sia vera in tutti i modelli della teoria, e in tal caso diciamo che la proposizione è **conseguenza logica** degli assiomi.

La logica non studia dunque la verità, ma la conseguenza logica.

> Sia Γ un insieme di sentenze (o proposizioni) e F una proposizione, F è **conseguenza logica** di Γ (o $\Gamma \Vdash F$) se F è vera in tutti i mondi in cui tutte le proposizioni $G \in \Gamma$ sono vere.

Intuitivamente si comprende dunque che le sentenze di Γ sono dei vincoli per l’esistenza dei mondi, in quanto più sentenze ci sono meno saranno i mondi in cui la teoria è applicabile. Occorre arrivare al punto in cui gli assiomi descritti sono sufficienti a dimostrare ciò che interessa, lasciando comunque la possibilità di applicare la teoria a diversi contesti o mondi.

Se $\Gamma \Vdash F$ allora la verità di F, ovvero l’insieme dei mondi in cui F è vera, è un **sovrainsieme** della verità di Γ, ovvero l’insieme dei mondi in cui le proposizioni di Γ sono vere.

Equivalenza logica

> Siano F e G due sentenze, F è **logicamente equivalente** a G ($F \equiv G$) se $F \Vdash G$ e $G \Vdash F$, ovvero se F e G sono vere negli stessi mondi.

È possibile inoltre dimostrare che l’equivalenza logica è una relazione di equivalenza, ovvero è riflessiva ($F \equiv F$), simmetrica (se $F \equiv G$ allora $G \equiv F$) e transitiva (se $F \equiv G$ e $G \equiv H$, allora $F \equiv H$).

Quando una teoria è interessante?

Una teoria è **inconsistente** quando non ammette mondi, ovvero in nessun mondo tutti gli assiomi della teoria sono contemporaneamente veri.

Esempio: la teoria con i seguenti assiomi $0 = 1$, $0 \neq 1$, $\forall x.x = x$ è una teoria inconsistente, poichè non esiste un mondo in cui tutti e 3 valgono.

Inoltre, se Γ è una teoria inconsistente, allora per ogni proposizione F si ha che F è una conseguenza logica di Γ. Da questo fatto si ha che anche una proposizione falsa, l'assurdo, è conseguenza logica di Γ ($\Gamma \Vdash \bot$). È vero anche il contrario, ovvero che se l'assurdo è conseguenza logica di Γ ($\Gamma \Vdash \bot$), allora la teoria Γ è inconsistente, dunque:

$\Gamma \Vdash \bot \Leftrightarrow$ la teoria è inconsistente.

Tutte le teorie **consistenti** sono interessanti, in quanto hanno almeno un modello/mondo in cui gli assiomi valgono. In generale più una teoria ha applicazioni in modelli diversi, più essa è interessante.

▼ **6.2 - Connotazione, denotazione, invarianza per sostituzione**

Denotazione e connotazione

La **denotazione** rappresenta ciò che vuole essere identificato, mentre la **connotazione** indica la sintassi utilizzata per farlo. Notiamo dunque che la denotazione e le connotazioni stanno in rapporto uno a tanti, ovvero ci sono tanti modi per descrivere qualcosa.

Esempio:

- Denotazione: *computer*
- Connotazioni: elaboratore, pc, computer, dispositivo elettronico ecc.

Uso meta-linguistico e invarianza per sostituzione

L'uso **meta-linguistico** di un linguaggio avviene quando si utilizzano connotazioni per riferirsi ad altre connotazioni.

Esempio:

- "Io mento" è una connotazione utilizzata per descrivere una connotazione.

Siano x e y due connotazioni per descrivere la stessa denotazione, il principio di **invarianza per sostituzione** vale se per ogni contesto P[ ], dove per contesto si intende una connotazione avente un buco in cui inserire un'altra connotazione (es. "cane __ gatto", al posto di __ può essere inserita un'altra connotazione come mangia, abbaia ecc.), le due connotazioni P[x] e P[y] denotano la stessa cosa.

Il principio di invarianza per sostituzione ci permette di stabilire se un linguaggio è meta-linguistico o meno.

Esempio:

- "cat" e "gatto" denotano la stessa cosa, ma "cat è monosillabico" denota il vero, mentre "gatto è monosillabico denota il falso, dunque "essere monosillabico" fa un uso metalinguistico.
- "2" e "3-1" denotano la stessa cosa, e sia "2 è pari" che "3-1 è pari" denotano il vero, dunque "essere pari" non fa un uso meta-linguistico.

Definizioni

- **Sentenza**: connotazione che denote un valore di verità (vero o falso).

- **Connettivo**: connotazioni che combinano sentenze per ottenere una nuova sentenza. Esistono anche connettivi binari, che si applicano ad una sola sentenza, e quelli 0-ari, ovvero le due costati vero e falso.

  Per fare logica fisseremo la loro interpretazione ad essere la stessa in ogni mondo.

▼ **7.0 - Logica proposizionale classica**

▼ **7.1 - Sintassi della logica proposizionale**

Sintassi

Sintassi: descrizione dell'insieme di tutte le connotazioni alle quali associamo una denotazione.

Formule

$F ::= \bot|\top|A|B|...|\neg F|F \wedge F|F \vee F|F \Rightarrow F$

Semantica intuitiva:

- $\bot$: denota la falsità.
- $\top$: denota la verità.
- $A, B, ...$: denotano un valore di verità sconosciuto/non determinato (dipende dal mondo).
- $\neg F$: negazione di F.
- $F_1 \wedge F_2$: congiunzione di due formule.
- $F_1 \vee F_2$: disgiunzione inclusiva di due formule.
- $F_1 \Rightarrow F_2$: implicazione materiale di due formule.

Precedenza e associatività

- **Precedenze**: $\neg > \wedge > \vee > \Rightarrow$

  Esempio: $\neg A \wedge B \vee \top \Rightarrow C$ si legge $(((\neg A) \wedge B) \vee \top) \Rightarrow C$
- **Associatività**: a destra per tutti gli operatori

  Esempio: $A \Rightarrow B \Rightarrow C$ si legge $A \Rightarrow (B \Rightarrow C)$

Formalizzazione

Con **formalizzazione** di una frase in linguaggio naturale si intende trovare una formula logica che meglio approssima la frase.

Esempio:

- "Oggi piove ma non ho preso l'ombrello".

  Formalizzazione in logica proposizionale: $A \wedge B$, dove A sta per "oggi piove" e B per "non ho preso l'ombrello". È una buona rappresentazione della frase ma approssimativa, in quanto "ma" ed "e" non hanno lo stesso identico significato nel linguaggio naturale.

Difficoltà nella formalizzazione

- Possono esistere diverse connotazioni per gli stessi connettivi.

  Esempio:

  - "Se A allora B", "A implica B", "A è condizione sufficiente per B" ecc. vengono tutte formalizzate in $A \Rightarrow B$.

    In questo caso tutte le connotazioni hanno lo stesso significato.
  - "A e B", "A ma B", "A nonostante B" vengono formalizzate in $A \wedge B$.

    In questo caso però queste connotazioni non hanno lo stesso identico significato nel linguaggio naturale.

- Nel linguaggio naturale esistono sinonimi e contrari che devono essere identificati al fine di formalizzare in maniera corretta.

  Esempio:

  - “Se Mario è **acculturato** allora oggi c'è **bel tempo**”, “oggi **splende il sole** e Mario è **ignorante**” devono essere formalizzate in questo modo: $M \Rightarrow B$, $B \wedge \neg M$.

▼ **7.2 - Semantica classica della logica proposizionale**

Semantica: descrive ciò che viene associato alle connotazioni descritte dalla sintassi. Quello che viene associato sono le denotazioni, le quali vengono prese all’interno di un **dominio di interpretazione** scelto (insiemi, numeri, figure geometriche ecc.).

La semantica dunque descrive l’insieme di tutti i significati che diamo alle connotazioni descritte tramite la sintassi. Queste ovviamente variano a seconda del dominio di interpretazione che prendiamo in considerazione (1 > 0 è una proposizione che possiamo valutare prendendo in considerazione il dominio dei numeri, valutando 1 e 0 come numeri, o anche delle figure geometriche, valutando 1 e 0 come due figure geometriche).

Funzione semantica: funzione che associa ad una connotazione una denotazione derivante dal dominio di interpretazione fissato.

É possibile dare semantiche totalmente diverse allo stesso linguaggio. In genere esiste una **semantica intesa**, la quale corrisponde a quella semantica che viene attribuita ad un linguaggio nel caso in cui la semantica non viene definita in modo rigoroso.

Logica classica

La **logica classica** prevede una visione platonica, la quale vede il mondo come qualcosa di immutabile e non creato dall’uomo, ma preesistente prima dell’esistenza dell’uomo, il quale può solo osservare le sue leggi senza poterle modificare.

Secondo logica classica, in ogni mondo:

- Ogni enunciato è vero o falso, e non può essere vero o falso allo stesso tempo.
- **Staticità**: il valore di verità non muta (se qualcosa è vero/falso lo sarà per sempre).
- **Determinatezza**: il valore di verità di un enunciato è sempre determinato.

Utilizzeremo i naturali **0** e **1** per indicare le denotazioni di verità e falsità.

Funzione di interpretazione o **mondo**: funzione matematica che associa alle variabili proposizionali i valori di verità. {A, B, …} → {0, 1} (Dominio → Immagine).

Da qui in avanti indicheremo le funzioni di interpretazione con V, V', V_1, V_2 ecc.

Semantica classica

Nella **semantica classica** la semantica di $\bot$, $\top$ e dei connettivi è già fissata, mentre varia in base al mondo quella delle variabili proposizionali A, B, ecc.

Data una funzione di interpretazione, definiamo la semantica classica $[\![.]\!]^V : F \to \{0, 1\}$ tramite induzione strutturale nel seguente modo:

- $[\![\bot]\!]^V = 0$
- $[\![\top]\!]^V = 1$
- $[\![A]\!]^V = V(A)$

- $[\![\neg F]\!]^V = 1 - [\![F]\!]^V$
- $[\![F_1 \wedge F_2]\!]^V = min\{[\![F_1]\!]^V . [\![F_2]\!]^V\}$
- $[\![F_1 \vee F_2]\!]^V = max\{[\![F_1]\!]^V . [\![F_2]\!]^V\}$
- $[\![F_1 \Rightarrow F_2]\!]^V = max\{1 - [\![F_1]\!]^V . [\![F_2]\!]^V\}$

Tabelle di verità

$[\![\bot]\!]^V$
0

$[\![\top]\!]^V$
1

Top & Bottom.

$[\![F]\!]^V$	$[\![\neg F]\!]^V$
0	1
1	0

Not.

$[\![F_1]\!]^V$	$[\![F_2]\!]^V$	$[\![F_1 \wedge F_2]\!]^V$
0	0	0
0	1	0
1	0	0
1	1	1

And.

$[\![F_1]\!]^V$	$[\![F_2]\!]^V$	$[\![F_1 \vee F_2]\!]^V$
0	0	0
0	1	1
1	0	1
1	1	1

Bottom.

$[\![F_1]\!]^V$	$[\![F_2]\!]^V$	$[\![F_1 \Rightarrow F_2]\!]^V$
0	0	1
0	1	1
1	0	0
1	1	1

Se, allora.

Il **se, allora** ha un significato differente rispetto al pensiero comune in quanto quest'ultimo non comprende il caso 0 0 → 1, il quale sta a significare che il falso implica qualunque cosa (es. $[\![$se 2 + 2 = 5, allora gli asini volano$]\!]^V$ = 1).

▼ **7.3 - Conseguenza ed equivalenza logica in logica proposizionale classica**

Conseguenza logica

> F è **conseguenza logica** di Γ ($\Gamma \Vdash F$) quando per ogni mondo V si ha che, se $[\![G]\!]^V = 1$ per ogni $G \in \Gamma$, allora $[\![F]\!]^V = 1$.

Equivalenza logica

> F è **logicamente equivalente** a G ($F \equiv G$) quando per ogni mondo V, $[\![F]\!]^V = [\![G]\!]^V$.

A livello di tabella di verità, due formule sono logicamente equivalenti se le loro tabelle di verità coincidono.

Tautologie

> F è una **tautologia** quando $\Vdash F$, ovvero F è conseguenza logica dell'insieme vuoto di formule.

Ciò sta a significare che F è una verità assoluta, ovvero per ogni mondo si ha $[\![F]\!]^V = 1$.

Esempio:

- $\top = 1$ è una tautologia, e di conseguenza anche $[\![\top]\!]^V \wedge [\![\top]\!]^V = 1$ è una tautologia, in quanto vera in tutti i mondi.
- $[\![A]\!]^V \Rightarrow [\![A]\!]^V = 1$ è una tautologia, in quanto la sua tabella di verità è la seguente.

$v(A)$	$[\![A \Rightarrow A]\!]^v$
0	1
1	1

Soddisfacibilità e insoddisfacibilità

> F è **soddisfatta** in un mondo V ($V \Vdash F$) se $V(F) = 1$.

> F è **soddisfacibile** se esiste un mondo V in cui $V \Vdash F$.

La tabella di verità corrispondente ha almeno un 1.

> F è **tautologica** quando per ogni mondo V si ha $V \Vdash F$.

La tabella di verità corrispondente ha tutti 1.

> F è **insoddisfacibile** quando non esiste un mondo V per cui $V \Vdash F$

La tabella di verità corrispondente ha tutti 0, infatti insoddisfacibile è il contrario di tautologica.

Una formula F può dunque essere di 3 tipologie:

- **Tautologica** (tutti **1**).
- **Soddisfacibile** ma non tautologica (sia **0** che **1**).
- **Insoddisfacibile** (tutti **0**).

▼ 7.4 - Teorema di invarianza per sostituzione nella logica proposizionale classica

Prima di definire in maniera rigorosa il **teorema di invarianza per sostituzione** nella logica proposizionale occorre definire il concetto di **contesto**, che nella logica proposizionale viene definito come una formula che contiene uno o più **buchi**. A sua volta un buco in logica proposizionale corrisponde a una variabile proposizionale, e riempire il buco corrisponde a rimpiazzare la variabile con una formula.

Esempi di sostituzione di una formula G al posto della variabile A nella formula/contesto F (scritto $F[G/A]$) sono i seguenti (avvengono per ricorsione strutturale su F):

- $\bot[G/A] = \bot$
- $A[G/A] = G$
- $B[G/A] = B$
- $(\neg F)[G/A] = \neg F[G/A]$
- $(F_1 \wedge F_2)[G/A] = F_1[G/A] \wedge F_2[G/A]$ (analogo ai casi $\vee$ e $\Rightarrow$)

Dopo aver definito il concetto di contesto in logica proposizionale è possibile definire il teorema di invarianza per sostituzione:

> Si ha **invarianza sostitutiva** se per tutte le formule F, G_1, G_2 e per A, se $G_1 \equiv G_2$, allora $F[G_1/A] = F[G_2/A]$.

Esempi di dimostrazione dell'induzione strutturale:

- Caso $\bot$: $\bot[G_1/A] = \bot \equiv \bot = \bot[G_2/A]$.
- Caso A: $A[G_1/A] = G_1 \equiv G_2 = A[G_2/A]$.
- Caso B: $B[G_1/A] = B \equiv B = B[G_2/A]$.
- Caso $F_1 \wedge F_2$ (analogo ai casi $\vee$ e $\Rightarrow$):
  - Dobbiamo dimostrare: $(F_1 \wedge F_2)[G_1/A] \equiv (F_1 \wedge F_2)[G_2/A]$
  - Per ipotesi induttiva sappiamo: $[\![F_1[G_1/A]]\!]^V \equiv [\![F_1[G_2/A]]\!]^V$ e $[\![F_1[G_1/A]]\!]^V \equiv [\![F_1[G_2/A]]\!]^V$.
  - Dimostrazione:

    $$
    \begin{aligned}
    &[\![(F_1 \wedge F_2)[G_1/A]]\!]^V \\
    &= [\![F_1[G_1/A] \wedge F_2[G_1/A]]\!]^V \\
    &= min\{[\![F_1[G_1/A]]\!]^V . [\![F_2[G_1/A]]\!]^V\} \\
    &= min\{[\![F_1[G_2/A]]\!]^V . [\![F_2[G_2/A]]\!]^V\} \\
    &= [\![F_1[G_1/A] \wedge F_2[G_1/A]]\!]^V \\
    &= [\![(F_1 \wedge F_2)[G_2/A]]\!]^V
    \end{aligned}
    $$

▼ 7.5 - Connettivi e tabelle di verità

Ogni **connettivo n-ario** è una funzione $f : \{0,1\}^n \to \{0,1\}$ e viene definito da una tabella di verità con 2^n **righe**.

Inoltre n valori in input sono in grado di descrivere 2^{2^n} **connettivi differenti**. Solo per alcuni di questi è stata però data una connotazione.

Connettivi 0-ari

$[\![\bot]\!]^v$	$[\![\top]\!]^v$
0	1

Connettivi 0-ari.

Tutti i $2^{2^0} = 2^1 = 2$ connettivi 0-ari hanno una connotazione ($\bot$ e $\top$).

Connettivi 1-ari

$v(F)$			$[\![\neg F]\!]^v$	
0	0	0	1	1
1	0	1	0	1

Connettivi 1-ari.

Dei $2^{2^1} = 2^2 = 4$ connettivi 1-ari solo il $\neg$ ha una connotazione, in quanto gli altri non sono particolarmente utili.

Connettivi 2-ari

F_1	F_2		$\wedge$					$\oplus$	$\vee$	$\tilde{\vee}$	$\iff$				$\Rightarrow$	$\tilde{\wedge}$	
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Connettivi 2-ari.

Dei $2^{2^2} = 2^4 = 16$ connettivi 2-ari solo alcuni hanno una connotazione.

Riduzione fra connettivi

Spesso è possibile esprimere un connettivo utilizzandone altri e l'equivalenza logica, in questi casi si dice che il connettivo è **riducibile** ad altri.

Esempi:

- $A \Rightarrow B \equiv \neg A \vee B$
- $\neg A \equiv A \Rightarrow \bot$

Ridondante

> Un insieme di connettivi è **ridondante** se contiene almeno un connettivo riducibile ai restanti.

Funzionalmente completo

> Un insieme di connettivi è **funzionalmente completo** se ogni altro connettivo è riducibile a questi.

Notazione: Siano S e T due insiemi di connettivi, si scrive $S \triangleright T$ se ogni connettivo di S è riducibile ai connettivi di T.

Teorema:

> Se S è funzionalmente completo e $S \triangleright T$, allora anche T è funzionalmente completo.

Perchè i connettivi della logica proposizionale classica?

$\{\vee, \wedge, \bot, \top, \neg\}$ è l'insieme dei connettivi scelto dalla logica proposizionale classica. Questo insieme è funzionalmente completo ma anche ridondante, ed è stato scelto per via di un compromesso fra l'esigenza di considerare un insieme **piccolo** e **funzionalmente completo**, e un insieme che permetta di esprimere in modo semplice ragionamenti effettuati tramite il **linguaggio naturale**.

Inoltre la riduzione fra i vari connettivi dipende dalla semantica utilizzata, e l'insieme dei connettivi scelto è funzionalmente completo in quasi **tutte le semantiche** utilizzate.

Infine sono stati scelti questi connettivi per via delle loro rilevanza sia in ambito **matematico** che **informatico**.

Equivalenze logiche notevoli

Nota: le equivalenze in rosso valgono solo in logica proposizionale classica)

Commutatività

$A \vee B \equiv B \vee A$, $A \wedge B \equiv B \wedge A$

Associatività

$A \vee (B \vee C) \equiv (A \vee B) \vee C$, $A \wedge (B \wedge C) \equiv (A \wedge B) \wedge C$

Idempotenza

$A \vee A \equiv A$, $A \wedge A \equiv A$

Distributività

$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$, $A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$

Assorbimento

$A \vee (A \wedge B) \equiv A$, $A \wedge (A \vee B) \equiv A$

Elemento neutro

$A \vee \bot \equiv A$, $A \wedge \top \equiv A$

Annichilamento

$A \vee \top \equiv \top$, $A \wedge \bot \equiv \bot$

Doppia negazione

$\neg\neg A \equiv A$

De morgan

$\neg(A \vee B) \equiv \neg A \wedge \neg B$, $\neg(A \wedge B) \equiv \neg A \vee \neg B$

Implicazione

$A \Rightarrow \equiv \neg A \vee B$

Equivalenze logiche aggiuntive

Modus barbara: $A \Rightarrow B \equiv \neg B \Rightarrow \neg A$

Regola di deduzione:

- $\neg A \vee B, A \Vdash B$
- $\neg A \vee B, A \vee B \Vdash B \vee C$

Teorema di completezza

> Siano P e Q due formule della logica proposizionale, $P \equiv Q$ se e se solo se posso dimostrarlo tramite le equivalenze logiche notevoli appena presentate.

▼ 7.6 - Deduzione sintattica

Vediamo ora una serie di teoremi che permettono di chiarire i rapporti che esistono tra alcuni connettivi ($\Rightarrow$, $\Leftrightarrow$, $\neg$, $\bot$, $\top$) e le nozioni di conseguenza ed equivalenza logica.

Nota: le dimostrazioni dei seguenti teoremi si trovano nelle slide, sono da imparare a memoria.

Teorema di deduzione semantica

Lemma

> Per ogni formula F e G si ha $\Gamma \Vdash F \Rightarrow G$ se e solo se $\Gamma, F \Vdash G$.

Teorema

> Per tutte le formula $F_1, ..., F_n, G$ si ha $F_1, ..., F_n \Vdash G$ se e solo se $\Vdash F_1 \Rightarrow ... \Rightarrow F_n \Rightarrow G$.

Il teorema di deduzione semantica è molto importante in quanto ci dice che il concetto di **conseguenza logica** è riducibile al concetto di **tautologia** e viceversa.

Altri teoremi

Teorema

> $\Vdash F$ se e solo se $F \Rightarrow \top$.

Teorema

> $\Vdash F$ se e solo se $\neg F$ è insoddisfacibile.

Teorema

> $\Gamma \Vdash F$ se e solo se l'insieme $\Gamma, \neg F$ è insoddisfacibile.

Teorema

> Γ è **insoddisfacibile** se e solo se $\Gamma \models \bot$.

Teorema

> Γ è **soddisfacibile** se e solo se $\Gamma \not\models \bot$.

▼ **8.0 - Deduzione naturale per la logica proposizionale**

> $\Gamma \vdash F$ (si legge “dalle ipotesi Γ deduco la conclusione F”).

Il simbolo appena introdotto $\vdash$ è equivalente ad affermare che F è **dimostrabile** partendo dalle ipotesi Γ tramite una prova esplicita.

▼ **8.1 - Albero di deduzione naturale**

Un **albero di deduzione naturale** per $\Gamma \vdash F$ è una struttura dati ad albero che consente di rappresentare una dimostrazione in modo grafico.

Struttura dell’albero

Esso è composto da:

- I **Nodi**, etichettati con delle formule di inferenza.
- Le **foglie**, che possono essere sia **formule scaricate** $[G]$ (ipotesi locali) che **formule non scaricate** G (ipotesi globali).
  - Le formule non scaricate sono etichettate con formule appartenenti a Γ, ovvero le ipotesi globali.
  - Le formule scaricate lo sono solo nella regola di inferenza della radice, mentre nel resto della dimostrazione sono formule locali valide.
- La **radice**, etichettata con la formula da dimostrare.
- I **nodi interni**, oltre alla formula, sono etichettati con delle **regole di inferenza**, le quali stabiliscono anche il modo in cui la dimostrazione va sviluppata.

  Consiglio: nella costruzione di una dimostrazione, scrivere prima la regola di inferenza da utilizzare e poi svilupparla, non viceversa.

Esempio:

- $\dfrac{\dfrac{\dfrac{[A \wedge (B \Rightarrow C)]}{B \Rightarrow C}\wedge e^2 \qquad B}{C}\Rightarrow e}{A \wedge (B \Rightarrow C) \Rightarrow C} \Rightarrow i$

Struttura ricorsiva

Gli alberi di deduzione naturale vengono dunque indicati componendo **ricorsivamente** regole di interferenza. La struttura ricorsiva permette di definire funzioni per ricorsione strutturale su alberi di deduzione e di effettuare prove per induzione strutturale.

Esempio:

- $\dfrac{\dfrac{F_1 \dots F_n}{H_1}(regola{-}1) \dots \dfrac{G_1 \dots G_m}{H_l}(regola{-}2)}{H}(regola - x)$

▼ **8.2 - Regole di inferenza**

Regole di inferenza

Le regole di inferenza presentano la seguente sintassi:
$\dfrac{F_1 ... F_n}{F}$ (nome regola)

- F è la **conclusione** della regola.

- $F_1 \dots F_2$ sono le **premesse** della regola.

Per indicare che è possibile assumere localmente A per concludere F_1, scriviamo la singola premessa F_1 nel seguente modo:

- $\begin{matrix}[A] \\ \vdots \\ F_1\end{matrix}$

Una regola che non presenta premesse si dice **assioma**. I casi base sono dunque rappresentati da foglie o da assiomi.

Passi di inferenza

Per ogni connettivo esistente esistono due tipi di passi di inferenza:

- **Regole di introduzione.**

  Ci dicono tutti i modi in cui è possibile concludere una formula che presenta un determinato connettivo.

- **Regole di eliminazione.**

Ogni passo di inferenza ammette sempre due letture:

- **Bottom-up**: dalle premesse alla conclusione.
- **Top-down**: dalla conclusione alle premesse.

I **matematici**, al fine di trovare una nuova dimostrazione, utilizzano solitamente un procedimento top-down, in quanto la conclusione è unica mentre le ipotesi sono spesso molteplici, dunque lo spazio di ricerca partendo dalle ipotesi è più ampio. Dopo averla trovata però la presentano in maniera prevalentemente bottom-up per aumentarne l'eleganza.

Il metodo top-down per fare dimostrazioni permette dunque di trovare la strada giusta per dimostrare, ma, a differenza del metodo bottom-up, presenta la possibilità di sbagliarsi nel momento in cui si applicano in maniera errata regole non invertibili. A volte è più facile alternare le due strategie applicandole in maniera mista.

Correttezza

> Una regola $\frac{F_1 \dots F_n}{H}$ è **corretta** quando $F_1 \dots F_n \Vdash H$.

Se in una regola la premessa contempla **ipotesi scaricate**, esse vanno integrate nella formula finale per dimostrarne la correttezza in questo modo:

$\begin{matrix}[F] \\ \vdots \\ \frac{E \quad G}{F_3}\end{matrix}$ $(\wedge_e)$ è corretta quando $E, F \Rightarrow G \Vdash H$.

Se un insieme di regole è corretto e questo viene combinato per creare un albero di deduzione, allora tutto l'albero sarà corretto in quanto la conclusione della dimostrazione sarà una conseguenza logica delle premesse.

Invertibilità

> Una regola $\frac{F_1 \dots F_n}{H}$ è **invertibile** quando per ogni i si ha che $H \Vdash F_i$.

Come per la correttezza, se le premesse contemplano ipotesi scaricate, esse vanno incluse con una implicazione (es. {}).

L'invertibilità gioca un ruolo fondamentale nella ricerca di **dimostrazioni** in quanto se una regola è invertibile può essere sempre applicata nella ricerca top-down senza portare a vicoli ciechi.

Inoltre solitamente un buon metodo per dimostrare è quello di applicare in **maniera meccanica** tutte le regole invertibili che si possono applicare, per poi iniziare a ragionare solo nel momento in cui si è costretti ad applicare una regola non invertibile.

Infine, se nel fare una dimostrazione si arriva ad un vicolo cieco, occorre fare **backtracking**, ovvero fare dei passi indietro nella dimostrazione, ma ha senso fare questo procedimento per le regole non invertibili, in quanto la correttezza delle altre nei due versi è garantita.

Regole del connettivo $\wedge$

Regola di introduzione

$$\frac{F_1 \quad F_2}{F_1 \wedge F_2} (\wedge_i)$$

- Lettura **bottom-up**: se F_1 e F_2, allora $F_1 \wedge F_2$.
- Lettura **top-down**: per dimostrare $F_1 \wedge F_2$, devo dimostrare sia F_1 che F_2.

La regola è sempre **invertibile**.

Dimostrazione della **correttezza**: $F_1, F_2 \Vdash F_1 \wedge F_2$ in quanto, per ogni mondo v, se $[\![F_1]\!]^v = [\![F_2]\!]^v = 1$ allora $[\![F_1 \wedge F_2]\!]^v = min\{[\![F_1]\!]^v, [\![F_2]\!]^v\} = 1$.

Dimostrazione dell'**invertibilità**: $F_1 \wedge F_2 \Vdash F_i$ per $i \in \{1, 2\}$ in quanto, per ogni mondo v, se $[\![F_1 \wedge F_2]\!]^v = min([\![F_1]\!]^v, [\![F_2]\!]^v) = 1$ allora $[\![F_i]\!]^v = 1$ per $i \in \{1, 2\}$.

Regola di eliminazione

$$\frac{\begin{array}{c}[F_1][F_2] \\ \vdots \\ F_1 \wedge F_2 \qquad F_3\end{array}}{F_3} \quad (\wedge_e)$$

- Lettura **bottom-up**: se $F_1 \wedge F_2$ e se ipotizzando F_1 e F_2 concludo F_3, allora F_3.
- Lettura **top-down**: per dimostrare F_3 data l'ipotesi $F_1 \wedge F_2$ è sufficiente dimostrare F_3 sotto le ipotesi F_1 e F_2.

La regola è **invertibile** solo quando $F_1 \wedge F_2$ è dimostrabile.

(Dimostrazione della correttezza nella slide 13 del pacco "Deduzione naturale").

Regole alternative di eliminazione

$$\frac{F_1 \wedge F_2}{F_1} (\wedge_{e_1}), \ \frac{F_1 \wedge F_2}{F_2} (\wedge_{e_2})$$

- Lettura **bottom-up**: se $F_1 \wedge F_2$ allora F_1 (o F_2).
- Lettura **top-down**: per dimostrare F_1 (o F_2) basta dimostrare $F_1 \wedge F_2$.

Le due regole sono **invertibili** solo quando $F_1 \wedge F_2$ è dimostrabile.

(Dimostrazione della correttezza nella slide 17 del pacco "Deduzione naturale").

Regole del connettivo $\vee$

Regole di introduzione

$$\frac{F_1}{F_1 \vee F_2}(\vee_{i_1}),\ \frac{F_2}{F_1 \vee F_2}(\vee_{i_2})$$

- Lettura **bottom-up**: se F_1 (o F_2) vale, allora vale anche $F_1 \vee F_2$.
- Lettura **top-down**: per dimostrare $F_1 \vee F_2$ è sufficiente dimostrare F_1 (o F_2).

Le due regole sono **invertibili** solo quando F_1 (o F_2) è dimostrabile.

(Dimostrazione della correttezza nella slide 24 del pacco “Deduzione naturale”).

Regola di eliminazione

$$\frac{F_1 \vee F_2 \qquad \begin{array}{c}[F_1]\\ \vdots \\ F_3\end{array} \qquad \begin{array}{c}[F_2]\\ \vdots \\ F_3\end{array}}{F_3}\ (\vee_e)$$

- Lettura **bottom-up**: se vale $F_1 \vee F_2$ e F_3 vale sia quando vale F_1 che quando vale F_2, allora vale F_3.
- Lettura **top-down**: per dimostrare qualunque cosa sapendo che $F_1 \vee F_2$ vale è sufficiente procedere per casi, dimostrando la stessa cosa assumendo prima che F_1 valga e poi che F_2 valga.

La regola è **invertibile** solo quando $F_1 \wedge F_2$ è dimostrabile.

(Dimostrazione della correttezza nella slide 26 del pacco “Deduzione naturale”).

Regole del connettivo $\bot$

Regole di introduzione

Nessuna, poichè non è possibile dimostrare il $\bot$.

Regole di eliminazione

$$\frac{\bot}{F}(\bot_e)$$

- Lettura **bottom-up**: dal falso segue qualunque cosa.
- Lettura **top-down**: per dimostrare qualunque cosa posso ridurmi a dimostrare l’assurdo.

La regola è **invertibile** solo quando $\bot$ è dimostrabile.

Dimostrazione della **correttezza**: $\bot \Vdash F$.

Regole del connettivo $\top$

Regole di introduzione

$$\frac{}{\top}(\top_i)$$

Questa regola è utile per concludere un ramo della dimostrazione senza generare sottorami.

Regola di eliminazione (inutile)

$$\frac{\top \quad F}{F}(\top_e)$$

- Lettura **bottom-up**: il $\top$ è vero.
- Lettura **top-down**: per dimostrare $\top$ non debbo fare nulla.

La regola è **invertibile**.

Dimostrazione della **correttezza**: $\Vdash \top$.

Dimostrazione dell'**invertibilità**: la regola è invertibile.

Regole del connettivo $\Rightarrow$

Regola di introduzione

$$\frac{\begin{array}{c}[F_1]\\ \vdots \\ F_2\end{array}}{F_1 \Rightarrow F_2}(\Rightarrow_i)$$

- Lettura **bottom-up**: se ipotizzando F_1 dimostro F_2, allora $F_1 \Rightarrow F_2$.
- Lettura **top-down**: per dimostrare $F_1 \Rightarrow F_2$ basta assumere F_1 e dimostrare F_2.

La regola è **invertibile**.

Dimostrazione della **correttezza** e dell'**invertibilità**: $F_1 \Rightarrow F_2 \Vdash F_1 \Rightarrow F_2$.

Regola di eliminazione

$$\frac{F_1 \Rightarrow F_2 \quad F_1}{F_2}(\Rightarrow_e)$$

- Lettura **bottom-up**: se F_1 e $F_1 \Rightarrow F_2$, allora F_2.
- Lettura **top-down**: per dimostrare F_2 devo trovare un F_1 che valga e tale per cui $F_1 \Rightarrow F_2$.

La regola **non è invertibile**.

(Dimostrazione della correttezza nella slide 37 del pacco "Deduzione naturale").

Regole del connettivo $\neg$

Siccome in logica classica $\neg \equiv (F_1 \Rightarrow \bot)$ le regole del connettivo $\neg$ vengono riprese da quelle del connettivo $\Rightarrow$.

Regola di introduzione

$$\frac{\begin{array}{c}[F]\\ \vdots \\ \bot\end{array}}{\neg F}(\neg_i)$$

- Lettura **bottom-up**: se ipotizzando F_1 dimostro l'assurdo, allora $\neg F_1$.
- Lettura **top-down**: per dimostrare $\neg F_1$ basta assumere F_1 e dimostrare l'assurdo.

La regola è **invertibile**.

(Dimostrazioni della correttezza e dell'invertibilità sono le stesse del connettivo $\Rightarrow$).

Regola di eliminazione

$$\frac{\neg F \quad F}{\bot}(\neg_e)$$

- Lettura **bottom-up**: è assurdo avere sia F_1 e $\neg F_1$.
- Lettura **top-down**: per dimostrare l'assurdo basta dimostrare qualcosa e il suo contrario.

La regola **non è invertibile**.

(Dimostrazione della correttezza è la stessa del connettivo $\Rightarrow$).

Completezza e correttezza delle regole introdotte

Dopo aver introdotto le regole di inferenza per la logica proposizionale classica, vogliamo verificare che siano soddisfatti i due teoremi di correttezza e completezza.

Teorema di correttezza

$$\forall \Gamma, F . \Gamma \vdash F \Rightarrow \Gamma \Vdash F$$

Enunciato: se F è dimostrabile da Γ, allora F è conseguenza logica di Γ.

Intuizione: tutte le regole introdotte sono corrette.

Dimostrazione: il teorema di correttezza delle regole visto fino ad ora è facilmente dimostrabile tramite induzione strutturale sull'albero di derivazione $\Gamma \vdash F$ (dimostrazione slide 48 del pacco di slide "Deduzione naturale").

Teorema di completezza

$$\forall \Gamma, F . \Gamma \Vdash F \Rightarrow \Gamma \vdash F$$

Enunciato: se F è conseguenza logica di Γ, allora F è dimostrabile da Γ.

Intuizione: tutto ciò che è conseguenza logica è dimostrabile con le regole introdotte.

Per convincersi che con le regole introdotte non abbiamo raggiunto la completezza basta dare uno sguardo ai seguenti due teoremi, che sono tautologie in logica classica ma che non sono dimostrabili con le regole introdotte finora:

- $\Vdash \neg\neg A \Rightarrow A$ (ragionamento per assurdo - RAA)
- $\Vdash A \vee \neg A$ (excluded middle - EM)

Questi due teoremi sono tautologie in logica classica per il fatto che in quest'ultima logica i valori di verità sono 2 (vero e falso), mentre in altre logiche possono non valere. Visto però che nessuna delle regole introdotte finora si basa sul concetto che i valori di verità sono esclusivamente 2, allora non siamo in grado di dimostrare queste tautologie.

Per questo motivo occorre inserire un'altra regola al fine di raggiungere la completezza in logica classica.

Regola di dimostrazione per assurdo (RAA)

$$\begin{array}{l} \neg[F] \\ \quad \vdots \\ \quad \bot \\ \hline F \end{array}(RAA)$$

- Lettura **bottom-up**: se F_1 e $F_1 \Rightarrow F_2$, allora F_2.
- Lettura **top-down**: per dimostrare F_2 devo trovare un F_1 che valga e tale per cui $F_1 \Rightarrow F_2$.

La regola è **invertibile**.

(Dimostrazione della correttezza e dell'invertibilità nella slide 53 del pacco "Deduzione naturale").

Una dimostrazione per assurdo avviene in maniera differente rispetto a come abbiamo visto finora, in quanto per dimostrare l'assurdo possono essere utilizzate tutte le regole, invertibili e non invertibili, con l'unico scopo di accumulare più ipotesi possibili.

Tutte le dimostrazioni effettuate senza l'utilizzo della regola di dimostrazione per assurdo possono essere tradotte in programmi informatici, mentre ciò non è detto nel caso in cui questa regola venga utilizzata.

Utilizzo frequente della RAA

Un utilizzo frequente della RAA è il seguente:

$$\dfrac{\begin{array}{c}[\neg A] \\ \vdots \\ A \qquad \dfrac{\bot}{}[\neg A](\neg_e)\end{array}}{A}(RAA)$$

Principio del terzo escluso

> $\vdash A \vee \neg A$

Spesso le dimostrazioni effettuate utilizzando la RAA risultano molto laboriose e anti-intuitive, tuttavia però è possibile dimostrare tramite la RAA il principio del terzo escluso, il quale fornisce uno schema di prova molto potente per via del fatto che inserisce all'interno della dimostrazione il concetto che in logica classica esistono solo due valori di verità.

Uno schema molto potente di dimostrazione consiste infatti nell'utilizzo del principio del terzo escluso combinato alla regola di eliminazione dell'or:

$$\dfrac{\begin{array}{ccc} & [A] & [\neg A] \\ \vdots & \vdots & \vdots \\ A \vee \neg A & F & F\end{array}}{F}(\vee_e)$$

▼ **9.0 - Semantica intuizionista**

La **semantica intuizionista** è una teoria logica chiamata in questo modo in quanto si basa sulle idee dell'intuizionismo filosofico. In tale semantica infatti la matematica viene vista come un'invenzione, in quanto nessun suo teorema esiste in natura, ma sono tutti prodotti della mente umana.

Questa semantica viene anche detta semantica dell'evidenza, della conoscenza diretta o della calcolabilità, in quanto sostiene che la verità di un'affermazione matematica può essere stabilita solo attraverso la costruzione di un esempio concreto che la dimostri, e non attraverso la dimostrazione di una contraddizione, come può avvenire nella semantica classica.

So che c'è vs So chi è

- In logica **classica**: $\exists x.P(x)$ significa "so che **c'è** un x tale che P(x)".
- In logica **intuizionista**: $\exists x.P(x)$ significa "so **chi è** quell'x tale che P(x)", dunque dalla prova utilizzata per ottenere un $\exists$ è possibile ricavare un **algoritmo** per capire chi è.

Per questo motivo la logica intuizionista è molto più vicina e utile all'informatica. Le prove in logica intuizionista spesso sono dunque più complicate di quelle in logica classica, ma hanno un valore maggiore in quanto forniscono più informazioni.

Esempio:

- Dimostrare il seguente teorema: per ogni n, n è pari o n non è pari.
  - Dimostrazione classica: ovvio per il teorema del terzo escluso, in quanto un numero può essere solo o pari o dispari.

    Questa dimostrazione non ci fornisce un algoritmo per sapere se un certo numero è pari oppure non lo è, dunque ad esempio non possiamo sapere se il numero 5 è pari, ma sappiamo solo che o è pari o è dispari.
  - Dimostrazione intuizionista: procediamo per induzione strutturale su n.

    Caso 0: 0 = 2 * 0, dunque 0 è pari.

    Caso S m (successivo di m): per ipotesi induttiva m è pari oppure non lo è, dunque procediamo per casi:
    - m è pari: $\exists k.m = 2 \times k$, dunque $Sm = 2 \times k + 1$ non è pari in quanto non divisibile per 2.
    - m non è pari: $\exists k.m = 2 \times k + 1$, dunque $Sm = 2 \times k + 2 = 2 \times (k + 1)$ è pari in quanto divisibile per 2.

    Questa dimostrazione dunque contiene un algoritmo che ci consente di sapere se un certo numero è pari o non lo è, ad esempio sappiamo che il numero 5 è dispari perchè 4 è pari perchè 3 è dispari perchè 2 è pari perchè 1 è dispari perchè 0 è pari.

Un qualunque teorema del tipo $\forall i.\exists o.P(i, o)$, ovvero per ogni input i esiste un output o in relazione P con l'input, se viene dimostrato in logica intuizionistica la sua dimostrazione contiene un algoritmo per calcolare o a partire da i, mentre se viene dimostrato in logica classica la sua dimostrazione diventa molto più breve perchè svolge la metà del lavoro, in quanto ci dice solo se o esiste oppure no. Quando però non esiste alcun algoritmo in grado di calcolare o a partire da i le uniche dimostrazioni possibili sono classiche.

Valori di verità

Nella semantica classica:

- Il valore di verità di ogni enunciato è **sempre determinato**.
- Il valore di verità di ogni enunciato è **immutabile**.

Nella semantica intuizionista:

- Il valore di verità di ogni enunciato è determinato solo nel momento in cui se ne ha una prova/evidenza diretta (un **algoritmo**).
- Il valore di verità **può passare** dall'essere indeterminato all'avere un determinato valore di verità che non cambia più (nel momento in cui si scopre almeno un algoritmo o dimostro che non può esserci alcun algoritmo).

▼ 10.0 - Logica del prim’ordine

La logica proposizionale classica non riesce in modo corretto a tradurre i **quantificatori** del linguaggio naturale, fin’ora abbiamo infatti utilizzato una metalogica per giustificare il per ogni e l’esiste nelle dimostrazioni.

Nella logica del prim’ordine introduciamo quindi 2 identificatori: $\forall$ e $\exists$. In questo modo non siamo ancora completi rispetto al linguaggio naturale, ma riusciamo comunque a tradurre un numero maggiore di frasi.

▼ 10.1 - Sintassi e semantica

Sintassi della logica del prim’ordine

La logica del prim’ordine è composta da:

- Termini

  $$t ::= x|c|f^n(t_1, ..., t_n)$$

  - x: variabili (x, y ecc.).
  - c: costanti (0, 1, π ecc.).
  - $f^n(t_1 + ... + t_n)$: funzioni che prendono in input n termini e restituiscono un nuovo termine (funzione di addizione, sottrazione ecc.).

- Preposizioni

  $$P ::= P^n(t_1, ..., t_n)|\bot|\top|\neg P|P \land P|P \lor P|P \implies P|\forall x.P|\exists x.P$$

  Funzioni che prendono in input dei termini e restituiscono dei valori di verità.

Viene chiamata logica del prim’ordine in quanto i quantificatori $\forall$ e $\exists$ possono essere applicati solo agli **elementi appartenenti ad un insieme**, e non a funzioni o predicati, come avviene nelle logiche del second’ordine, terz’ordine e così via.

Esempi:

- $\forall x.x > 4$ **appartiene** alla logica del prim’ordine.
- $\forall f.f(x) > 4$ **non appartiene** alla logica del prim’ordine.
- $\forall P.P(x)$ **non appartiene** alla logica del prim’ordine.

La logica proposizionale classica è un caso particolare di logica del prim’ordine in cui non vengono utilizzati quantificatori e termini e vengono presi in considerazione solo predicati 0-ari. Ogni variabile proposizionale $A, B, C...$ rappresenta infatti un predicato $P^0, Q^0, R^0...$ che possono assumere i valori 0 e 1.

Semantica classica della logica del prim’ordine

Seguendo la semantica classica della logica del prim’ordine ciascun mondo v fissa un insieme non vuoto A e assegna:

- A ogni simbolo di costante c un elemento appartenente all’insieme A (es. $[\![\pi]\!]^v = 2$).
- A ogni simbolo di funzione n-aria f^n una funzione n-aria appartenente all’insieme A (es. $[\![+]\!]^v = \times$).

- A ogni simbolo di predicato n-ario P^n un predicato n-ario appartenente all'insieme A (es. $[\![>]\!]^v = |$, ovvero "divide").

Quindi, seguendo gli esempi, $[\![(\pi + \pi) > \pi]\!]^v = (2 \times 2) | 2 = 4 | 2$ è falso, poichè 4 non divide 2.

Semantica dei quantificatori:

- $[\![\forall x.P]\!]^v = 1$ se e solo se il valore di P nel mondo v è sempre 1 al variare della semantica di x su tutti gli elementi dell'insieme A.
- $[\![\exists x.P]\!]^v = 1$ se e solo se il valore di P nel mondo v è almeno una volta 1 al variare della semantica di x su tutti gli elementi dell'insieme A.

▼ **10.2 - Equivalenze logiche**

Equivalenze logiche notevoli

Quantificatori dello **stesso tipo commutano**:

- $\forall x.\forall y.P \equiv \forall y.\forall x.P$
- $\exists x.\exists y.P \equiv \exists y.\exists x.P$

Quantificatori di **tipo diverso non commutano**:

- $\exists x.\forall y.P \Vdash \forall y.\exists x.P$, ma $\forall x.\exists y.P \not\Vdash \exists y.\forall x.P$ (provare con $P : x \leq y$)

($\wedge$ e $\vee$) Le seguenti equivalenze vengono utilizzate per spostare i quantificatori in **posizione di testa** nelle formule:

- $(\forall x.P) \wedge (\forall x.Q) \equiv \forall x.(P \wedge Q)$
- $(\exists x.P) \vee (\exists x.Q) \equiv \exists x.(P \vee Q)$

($\neg$) Leggi di **De Morgan**:

- $\neg\forall x.P \equiv \exists x.\neg P$, in logica classica, ma in logica intuizionista vale solo la parte $\exists x.\neg P \Vdash \neg\forall x.P$.
- $\neg\exists x.P \equiv \forall x.\neg P$, sia in logica classica che intuizionista.

Nota: per dimostrare $\neg\forall x.P$ basta dimostrare $\exists x.\neg P$, ovvero è sufficiente un **controesempio**, ma per dimostrare $\neg\exists x.P$ bisogna dimostrare $\forall x.\neg P$, ovvero occorre una vera e proprio **dimostrazione**.

($\implies$) Per portare fuori i quantificatori da una premessa questi devono essere trasformati nel loro duale:

- $(\forall x.P) \implies Q \equiv \exists x.(P \implies Q)$
- $(\exists x.P) \implies Q \equiv \forall x.(P \implies Q)$

($\implies$) Per portare fuori i quantificatori da una conclusione questi non devono essere trasformati:

- $Q \implies (\forall x.P) \equiv \forall x.(Q \implies P)$
- $Q \implies (\exists x.P) \equiv \exists x.(Q \implies P)$

Quantificazioni limitate

Spesso vengono utilizzate delle quantificazioni limitate a un particolare dominio o proprietà:

- $\forall x \in A.P(x)$, la quale corrisponde a $\forall x.(x \in A \implies P(x))$
- $\exists x \in A.P(x)$, la quale corrisponde a $\exists x.(x \in A \wedge P(x))$

Le leggi di De Morgan in questo caso sono:

- $\neg \forall x \in A . P(x) \equiv \exists x \in A . \neg P(x)$, la quale in logica intuizionista vale solo nella parte $\exists x \in A . \neg P(x) \Vdash \neg \forall x \in A . P(x)$
- $\neg \exists x \in A . P(x) \equiv \forall x \in A . \neg P(x)$

▼ **11.0 - Strutture algebriche in informatica**

▼ **11.1 - Astrazione e Generalizzazione**

Astrazione

> L'**astrazione** è il tipico processo del pensiero scientifico che permette di definire enti, concetti o procedure matematiche, estraendo e isolando caratteristiche comuni a più oggetti e trascurandone altre.

Esempio: per astrarre il concetto di sedia si può estrarre la proprietà riguardante lo scopo, in quanto tutte le sedie hanno lo scopo di far sedere una persona, e trascurare il materiale con il quale è stata fatta, poichè non tutte le sedie sono fatte dello stesso materiale.

L'**algebra astratta** introduce le strutture algebriche per astrarre e classificare le operazioni che sono possibili su determinate classi di oggetti.

Generalizzazione

> La **generalizzazione** consiste nel prendere una definizione che si applica a certi casi e definirla in un nuovo modo tale per cui si applichi anche ad altri casi oltre ai precedenti.

Astrazione e generalizzazione in informatica

Un esempio di astrazione in informatica consiste nell'utilizzo di **interfacce** per nascondere come una struttura dati è stata implementata ma accentuarne il suo utilizzo, mostrandone solo i metodi che si possono utilizzare su di essa.

Esempio:

- Implementazione concreta di una lista di interi:

```
struct node {
  int item,
  node* next;
}
```

- Concetto astratto:

```
node* empty();
insert(int n, node* l);
remove(int n, node* l);
```

Esistono molti **benefici** nell'attuare astrazione e generalizzazione in informatica, eccoli elencati:

- **Riuso**: le buone generalizzazioni si applicano moltissime volte.
- **Chiarezza**: le generalizzazioni catturano concetti di alto livello, introducendo nome comprensibili.
- **Decoupling**: tramite astrazione e generalizzazione è possibile fare in modo che la correttezza di una parte non dipenda dall'implementazione di un'altra.

- **Correttezza**: l'astrazione consente di non dover utilizzare la tecnica del cut&paste, la quale può portare facilmente con sè problemi di correttezza ed errori, i quali devono poi essere corretti in tutte le copie.

I **linguaggi di programmazione** Turing completi, ovvero la maggior parte di quelli in circolazione, presentano la possibilità di effettuare le stesse operazioni e risolvere gli stessi problemi. Per questo motivo la scelta del linguaggio di programmazione da usare per un determinato scopo non sta nel fatto che un linguaggio può risolvere un determinato problema e un altro no, ma piuttosto nel livello di astrazione e generalizzazione che questo presenta. Ad esempio un linguaggio come c o c++ è un linguaggio poco astratto e generalizzato, in quanto consente di programmare ad un livello molto basso, mentre altri linguaggi di programmazione come Haskell presentano un livello di astrazione più elevato.

Dalla generalizzazione alle strutture algebriche in matematica

(Dimostrazioni dei teoremi presentati nelle slide del 29/11)

Esempio di generalizzazione

Avendo questi **due teoremi di partenza**:

- **Teorema T1**: $\forall e \in N.((\forall x.x + e = x) \Rightarrow e = 0)$

  Definizione: e è un **elemento neutro a destra** per $\circ$ sse $\forall x.x \circ e = x$.

- **Teorema T2**: $\forall e \in N.((\forall x.e * x = x) \Rightarrow e = 1)$

  Definizione: e è un **elemento neutro a sinistra** per $\circ$ sse $\forall x.e \circ x = x$.

Vista la loro similitudine, è possibile trovare una **generalizzazione** comune:

Teorema G: $\forall A.\forall \circ : A \times A \to A.\forall e_r, e_l \in A.(\forall x.e_l \circ x = x) \wedge (\forall x.x \circ e_r = x) \Rightarrow e_l = e_r$

- **G** è **T2** nel caso particolare: $A = N, \circ = *, e_r = 1$

Definizione: e è un **elemento neutro** per $\circ$ sse è neutro sia a sinistra che a destra.

È una generalizzazione in quanto:

- **G** è **T1** nel caso particolare: $A = N, \circ = +, e_l = 0$

Da G possiamo scoprire **nuovi concetti** e comprendere meglio i precedenti, ad esempio possiamo arrivare ai seguenti teoremi:

- T**eorema H**: se $\circ$ è un'operazione commutativa, allora se ha un elemento neutro a destra o a sinistra allora tale elemento è neutro.
- Teorema: l'elemento neutro, se esiste, è unico.
- Corollario (per H): l'elemento neutro di un'operazione commutativa, se esiste, è unico.

Per questo motivo G è una **generalizzazione informativa**, in quanto utile. Se una teoria è interessante, vi sono molti teoremi dimostrabili a partire da essi.

Definizione di struttura algebrica

> Una **struttura algebrica** è una tupla formata da uno o più **insiemi**, zero o più **elementi**, zero o più **funzioni** su tali insiemi (chiamate operazioni), e zero o più **assiomi** che devono essere soddisfatti.

Esempi:

- Un **left unital magma** è una tripla $(A, \circ, e)$ tale che $\circ : A \times A \to A$ e e è un elemento neutro a sinistra per $\circ$.
- Un **right unital magma** è una tripla $(A, \circ, e)$ tale che $\circ : A \times A \to A$ e e è un elemento neutro a destra per $\circ$.
- Un **unital magma** è una tripla $(A, \circ, e)$ tale che $\circ : A \times A \to A$ e e è sia un left unital magma che un right unital magma.

Morfismo di strutture algebriche

> Un **morfismo** fra due strutture algebriche dello stesso tipo è una funzione che mappa gli elementi della prima in quelli della seconda rispettandone le proprietà.

Esempio:

- Siano $(A, \circ, a)$ e $(B, \circ, b)$ due left unital magma. Un morfismo da $(A, \circ, a)$ a $(B, \circ, b)$ è una funzione $f \in B^A$ tale che:
  - $f(a) = b$
  - $\forall x, y. f(x \circ y) = f(x) \bullet f(y)$

  Esempio:
  - $(N, +, 0)$ e $(B, *, 1)$ sono due left unital magma. La funzione $f(n) = 2^n$ è un morfismo dal primo al secondo in quanto:
    - $2^0 = 1$
    - $\forall x, y. 2^{x+y} = 2^x * 2^y$

> Un **isomorfismo** di due strutture algebriche è un morfismo dalla prima alla seconda che sia una funzione biettiva e la cui inversa sia un morfismo.

Due strutture algebriche sono **isomorfe** se c'è almeno un isomorfismo fra di esse.

Esempio:

- $(R_0^+, +, 0)$ e $(R^+, *, 1)$ sono isomorfe come testimoniato dal morfismo $e^{\cdot}$ e dal suo morfismo inverso $log_{\cdot}$.

Dalla generalizzazione alle strutture algebriche in informatica

(I seguenti esempi verranno implementati in Haskell)

(Dimostrazioni dei teoremi presentati nelle slide del 29/11)

Esempio di generalizzazione

Problema 1: data una lista di interi, restituirne la somma.

```
sum [] = 0
sum (n:l) = n + sum l
```

Notiamo inoltre che il problema soddisfa la seguente proprietà (ci servirà poi per verificare che anche la generalizzazione la soddisferà):

```
sum (l1 ++ l2) = sum l1 + sum l2
```

(++ indica la concatenazione tra due liste)

Problema 2: data una lista di booleani, restituire la loro congiunzione.

```
conj [] = True
conj (b:l) = b && conj l
```

Questi due problemi presentano due soluzione simili ma non uguali.

Possiamo dunque tentare di trovare una generalizzazione utile partendo da sum: data una lista $[x_1, \dots, x_n]$, un valore e e un'operazione op, restituisce $op(x1, op(x2, \dots op(x_n, e) \dots))$.

```
foldr op e [] = e
foldr op e (n:l) = op n (foldr op e l)
```

Ritroviamo dunque i problemi sum e conj come istanze:

```
sum = foldr (+) 0
conj = foldr (&&) True
```

Inoltre è facilmente dimostrabile che anche la generalizzazione soddisfa le stesse proprietà dei problemi iniziali:

```
foldr op e (l1 ++ l2) = op (foldr op e l) (foldr op e l2)
```

Strutture algebriche in programmazione

Foldr dunque lavora su una **struttura algebrica** (A, op, e), dove A è un tipo di dato, op un'operazione binaria su A consigliata essere associativa, e e un elemento consigliato essere l'elemento neutro dell'operazione.

Ogni linguaggio di programmazione ha i suoi meccanismi per dichiarare tali strutture algebriche.

Ad esempio Haskell ha le **type classes**, e presenta anche già in maniera predefinita la struttura che ci serve in questo caso, ovvero il **monoide**: una tripla $(A, \circ, e)$ tale che $\circ$ è associativa e $(A, \circ, e)$ è un unital magma, ovvero e è l'elemento neutro di $\circ$. Possiamo dunque riscrivere la funzione foldr restringendoci alla type class Monoid:

```
foldr [] = e
foldr (n:l) = op n (foldr l)
```

In questo modo Haskell capisce da solo che per questa funzione deve usare il tipo Monoid, in quanto va a guardare tra le type class definite (Monoid è definita di default). A questo punto è possibile richiamare la funzione sulla lista [1, 2, 3] semplicemente scrivendo **foldr [1, 2, 3]** piuttosto che scrivere **foldr (+) 0 [1, 2, 3]**, in quanto comprende autonomamente l'operazione più adatta è il + (in quanto è associativa per i numeri interi) e l'elemento neutro è lo 0, aumentando dunque la leggibilità e la comprensione del codice.

A questo punto, come abbiamo visto per il caso della matematica, una volta che abbiamo introdotto la struttura algebrica (sotto forma di type class) possiamo costruirci una libreria sopra.

Morfismi in programmazione

Come abbiamo visto per la matematica, è possibile applicare i morfismi anche in informatica, avendo così la possibilità di passare da un'istanza di una struttura algebrica che soddisfi una certa proprietà ad un'altra che soddisfi le stessa proprietà.

Esempio:

- Partiamo da una struttura algebrica molto generica che chiamiamo **foldable** e costituita in questo modo (a, b, e, op), dove a e b sono insiemi, $e \in b$ e op è una funzione $(a \times b) \to b$ (da questa struttura algebrica è possibile definire liste, alberi, array ecc.).

  Un morfismo di foldable da (a, b, e, op) a (a', b', e', op') è una coppia di funzioni $f_a : a \to a'$ e $f_b : b \to b'$ tali che:

  - $f_b(e) = e$'
  - $f_b(op(x, l)) = op'(f_a(x), f_b(l))$

  Definiamo le liste con elementi dell'insieme a come un'istanza di foldable $(a, [a], [], :)$, dove $:$ è la concatenazione tra un elemento di a e una lista di a.

  Applicando il morfismo alle liste con le funzioni $a' = a$ e $f_a(x) = x$ otteniamo:

  - $f_b([]) = e'$
  - $f_b(x : l) = op'(x, f_b(l))$

  Notiamo che f_b è esattamente la **foldr**! Mettendo in pratica quanto visto dunque si riesce ad arrivare a scrivere un morfismo foldr super-generico che data una qualunque istanza di foldable ne sintetizza un'altra combinando ogni elemento della prima con certe funzioni passate dall'utente.

▼ **11.2 - Strutture algebriche interessanti**

Monoidi e gruppi

> $(\mathbb{A}, \circ)$, con $\mathbb{A}$ un insieme e $\circ$ un'operazione binaria del tipo $\mathbb{A} \times \mathbb{A} \to \mathbb{A}$, è un **magma**.

> $(\mathbb{A}, \circ)$ è un **semigruppo** se $\circ$ è associativa.

> $(\mathbb{A}, \circ, e)$ è un **monoide** se è un semigruppo e e è l'elemento neutro di $\circ$.

> $(\mathbb{A}, \circ, e, \cdot^{-1})$, con $\cdot^{-1}$ una generica operazione unaria, è un **gruppo** se $(\mathbb{A}, \circ, e)$ è un monoide e $\forall x \in \mathbb{A}.x \circ x^{-1} = e = x^{-1} \circ x$.

Ognuna di queste strutture algebriche viene detta **abeliana** se $\circ$ è commutativa, ovvero $x \circ y = y \circ x$.

Anelli

Gli **anelli** nascono dal bisogno di descrivere insiemi su cui agiscono due operazioni differenti che interagiscono "bene" tra di loro.

> $(\mathbb{A}, +, 0, *)$, dove $+$ e $*$ sono due operazioni binarie, è un **semianello** se $(\mathbb{A}, +, 0)$ è un monoide abeliano, $(\mathbb{A}, *, 1)$ è un monoide e valgono le seguenti proprietà:
>
> - $\forall x . x * 0 = 0 = 0 * x$.
> - $\forall x, y . (x + y) * z = xz + yz$.

> $(\mathbb{A}, +, 0, \cdot^{-1}, *)$ è un **anello** se $(\mathbb{A}, +, 0, *)$ è un semianello e $(\mathbb{A}, +, 0, \cdot^{-1})$ è un gruppo abeliano.

▼ **11.3 - Costruzioni dell'algebra universale**

Sottoinsiemi chiusi e sottostrutture algebriche

Insieme chiuso

> Sia $(\mathbb{A}, \circ)$ un semigruppo e $\mathbb{B} \subseteq \mathbb{A}$. $\mathbb{B}$ si dice **chiuso** rispetto a $\circ$ sse $\forall x, y \in \mathbb{B} . x \circ y \in \mathbb{B}$.

Esempio: l'insieme $\mathbb{P}$ dei numeri pari è chiuso rispetto alla somma, mentre l'insieme $\mathbb{D}$ dei numeri dispari non lo è.

Sottostrutture algebriche

> Data una struttura algebrica che si applica ad un insieme $\mathbb{A}$ e un $\mathbb{B} \subseteq \mathbb{A}$, $\mathbb{B}$ forma una **sottostruttura algebrica** della prima se tutte le operazioni sono chiuse rispetto a $\mathbb{B}$ e tutte le costanti della struttura appartengono a $\mathbb{B}$.

Esempio: considera l'insieme $\mathbb{P}$ dei numeri naturali pari. $\mathbb{P}$ è il sostegno di un sottosemigruppo di $(\mathbb{N}, *)$, di un sottomonoide di $(\mathbb{N}, +, 0)$, di un sottosemianello di $(\mathbb{N}, +, 0, *)$, ma non forma un sottomonoide di $(\mathbb{N}, *, 1)$ perchè $1 \notin \mathbb{P}$.

Intersezione e unione di sottostrutture algebriche

> Data una struttura di cui $\mathbb{B}$ e $\mathbb{C}$ sono sottostrutture, allora anche $\mathbb{B} \cap \mathbb{C}$ lo è.

(Dimostrazione slide 20 del pacco di slide "slides_alg2_Fabio").

> L'unione di sottostrutture non è in generale una sottostruttura.

Prodotto cartesiano di strutture algebriche

> Date due strutture algebriche A e B rispettivamente di sostegno $\mathbb{A}$ e $\mathbb{B}$, il loro **prodotto cartesiano** è una struttura algebrica dello stesso tipo e presenta le seguenti proprietà:
>
> - il **sostegno** è $\mathbb{A} \times \mathbb{B}$.

> - le **costanti** $e_{A\times B}$ richieste dal tipo di struttura sono coppie $< e_A, e_B >$ di costanti corrispondenti nelle due strutture.
> - le **operazioni** $\circ_{A\times B}$ richiesti dal tipo di struttura agiscono applicando l'operazione corrispondente sugli elementi della coppia: $< x_1, y_1 > \circ_{A\times B} < x_2, y_2 >=< x_1 \circ_A x_2, y_1 \circ_B y_2 >$.

Dunque è possibile costruire nuove istanze di sottostrutture algebriche usando intersezioni e prodotti cartesiani.

Sottostrutture tramite i morfismi

Morfismi di strutture algebriche

> Date due strutture algebriche dello stesso tipo A di sostegno $\mathbb{A}$ e B di sostegno $\mathbb{B}$, un morfismo da A a B è una funzione $f : A \to B$ tale che:
>
> - per ogni costante e_A di A, $f(e_A) = e_B$.
> - per ogni operazione op_A di A: $\forall x_1, \ldots, x_n . f(op_A(x_1, \ldots, x_n)) = op_B(f(x_1), \ldots, f(x_n))$.

Immagini di morfismi

> Sia f un morfismo da una struttura algebrica A a una struttura algebrica B. $Imm(f)$ è una sottostruttura di B.

Morfismo come osservazione

Un morfismo $f : \mathbb{A} \to \mathbb{B}$, data una qualche struttura algebrica sui sostegni $\mathbb{A}$ e $\mathbb{B}$, può essere pensato come un modo per **osservare** sugli elementi di $\mathbb{A}$ delle **proprietà** $\mathbb{B}$.

Esempio: dato un insieme X, la funzione $|\cdot| = P(X) \to \mathbb{N}$ (cardinalità) osserva per ogni sottoinsieme di Xquanto sia la sua cardinalità, ovvero il suo numero di elementi.

Supponiamo che tali osservazioni siano le uniche che ci interessano in un determinato momento, dunque vogliamo attuare un'astrazione degli elementi di $\mathbb{A}$ che mantenga solo le loro proprietà osservabili $\mathbb{B}$.

> Data un morfismo $f : \mathbb{A} \to \mathbb{B}$, la **relazione di equivalenza** indotta da f, $\sim_f$, è definita come segue: $x_1 \sim_f x_2$ sse $f(x_1) = f(x_2)$.

Esempio: se $f$calcola l'età di una persona allora $\sim_f$ è la relazione "essere coetanei".

A questo punto possiamo introdurre la definizione di proiezione $[\cdot]$, molto importante per arrivare al teorema fondamentale dei morfismi per strutture algebriche:

> Chiamiamo $[\cdot]$: $\mathbb{A} \to \mathbb{A}_{/\sim_f}$ (**proiezione**) la funzione che mappa ogni $x \in \mathbb{A}$ nella sua classe di equivalenza $\sim_f$.

Esempio: se f calcola l'età di una persona allora $\mathbb{A} \to \mathbb{A}_{/\sim_f}$ mappa ogni persona nella classe di equivalenza dei suoi coetanei. In questo modo si ottiene l'insieme quoziente i cui elementi sono le classi di equivalenza "età".

Teorema fondamentale dei morfismi per strutture algebriche

> Per ogni f morfismo da A a B si ha che:
>
> 1. $\mathbb{A}_{/\sim_f}$ è il sostegno di una struttura algebrica dello **stesso tipo**.
> 2. $\mathbb{A}_{/\sim_f}$ è **isomorfo** a $Imm(f)$.

▼ **11.4 - Teoria dei gruppi**

> Un **gruppo** $(\mathbb{A}, \circ, e, \cdot^{-1})$ è un monoide con un elemento aggiuntivo $\cdot^{-1}$ tale che: $\forall x \in \mathbb{A}. x \circ x^{-1} = e = x^{-1} \circ x$. a^{-1} si chiama **elemento opposto** di a.

Teoremi elementari sui gruppi

- L'elemento opposto di a è **unico**.

  (Dimostrazione slide 8 del pacco di slide n. 3 di algebra)
- In un gruppo, $\forall x, y. (x \circ y)^{-1} = y^{-1} \circ x^{-1}$.

  (Dimostrazione slide 9 del pacco di slide n. 3 di algebra)
- In un gruppo, $\forall x. (x^{-1})^{-1} = x$.

  (Dimostrazione slide 10 del pacco di slide n. 3 di algebra)
- In un gruppo, $x = y \iff x \circ y^{-1} = e \iff x^{-1} \circ y = e$

  (Dimostrazione slide 11 del pacco di slide n. 3 di algebra)

Esempi di gruppi

- Il gruppo degli **interi con l'addizione** è $(\mathbb{Z}, +, 0, \cdot^{-1})$. Scriviamo $\cdot^{-1}$ come $-n$, infatti ad esempio $3 + (-3) = 0 = (-3) + 3$.
- Il gruppo degli **interi modulo 2 con l'addizione** corrisponde al gruppo $(\mathbb{Z}_2, +, 0, \cdot^{-1})$, nel quale $\mathbb{Z}_2 = \{0, 1\}$. Costituisce un gruppo in quanto l'addizione viene definita secondo l'aritmetica modulare, ad esempio $1 + 1 = 0$ e $0 + 1 = 1$, e per quanto riguarda gli elementi opposti abbiamo che $1^{-1} = 1$ e $0^{-1} = 0$, infatti $1 + 1^{-1} = 1 + 1 = 0$ e $0 + 0^{-1} = 0 + 0 = 0$.

  Più in generale, per ogni $n \in \mathbb{N}$, gli interi modulo n formano un gruppo $(\mathbb{Z}_n, +, 0, \cdot^{-1})$.
- Dato un poligono regolare P_n con n lati, le isometrie su di esso formano un gruppo chiamato il gruppo diedrale D_n in cui:
  - L'operazione (composizione) del gruppo è la composizione di isometrie.
  - L'elemento neutro è l'isometria identità.
  - L'inverso di un'isometria è l'isometria inversa.

I gruppi di permutazioni

Una famiglia importante di gruppi è formata dai gruppi di permutazioni.

> Sia $\mathbb{A}$ un insieme. Una **permutazione** di $\mathbb{A}$ è semplicemente una funzione biettiva $\pi : \mathbb{A} \to \mathbb{A}$ (vengono modificate le posizioni degli elementi all'interno dell'insieme).

Dato un insieme $\mathbb{A}$, abbiamo che:

- $Perm(\mathbb{A})$ è l'insieme di tutte le permutazioni dell'insieme $\mathbb{A}$, spesso chiamato anche **gruppo simmetrico** di $\mathbb{A}$.
- Siano π_1, π_2 due permutazioni di $\mathbb{A}$, anche $\pi_1 \circ \pi_2$ lo è.
- La funzione identità $id : \mathbb{A} \to \mathbb{A}$ è una permutazione.
- Sia π una permutazione di $\mathbb{A}$, anche π^{-1}, ovvero la funzione inversa di π, lo è.

> $(Perm(\mathbb{A}), \circ, id, \cdot^{-1})$ è un gruppo, chiamato **gruppo delle permutazioni** di $\mathbb{A}$.

Teorema di Cayley

> Il **teorema di Cayley** afferma che ogni gruppo che ha come sostegno un insieme $\mathbb{A}$ è isomorfo a un sottogruppo del gruppo $Perm(\mathbb{A})$ delle permutazioni di $\mathbb{A}$.

Supponiamo infatti di avere un gruppo che ha come sostegno un insieme $\mathbb{A}$ composto in questo modo $\{a_1, a_2, a_3, \dots, a_n\}$. Ora prendiamo uno alla volta tutti gli elementi dell'insieme ed effettuiamo l'operazione $\circ$ del gruppo per ogni elemento dell'insieme, ottenendo un insieme di permutazioni $\{a \circ a_1, a \circ a_2, a \circ a_3, \dots, a \circ a_n\}$, dove al posto di a vengono inseriti tutti gli elementi dell'insieme $\mathbb{A}$ uno ad uno. Otteniamo una **permutazione** in quanto dobbiamo vedere l'operazione $a \circ a_i$ ad una posizione differente dell'insieme.

Rappresentiamo l'operazione effettuata in questo modo $\pi_a : \{1, 2, 3, ..., n\} \to \{1, 2, 3, ..., n\}$ dove $\pi_a(i)$ è definito come $a_j = a \circ a_i$. È possibile inoltre dimostrare che il morfismo $\pi = a \mapsto \pi_a$ è un **isomorfismo** tra $(\mathbb{A}, \circ, e, \cdot^{-1})$ ed un sottogruppo del gruppo $Perm(\mathbb{A})$ delle permutazioni di $\mathbb{A}$ (dimostrazioni da slide 40 a 42 del pacco di slide della lezione n. 3 di algebra).

L'isomorfismo mappa ad un **sottogruppo** di $Perm(\mathbb{A})$ e non a tutto il gruppo in quanto se l'insieme $\mathbb{A}$ ha n elementi l'isomorfismo creerà n permutazioni diverse, ognuna data moltiplicando ogni elementi dell'insieme $\mathbb{A}$ per uno degli n elementi dell'insieme $\mathbb{A}$. Il gruppo di $Perm(\mathbb{A})$ però è formato invece da $n!$ permutazioni differenti, dunque se $n > 2$ l'isomorfismo mapperà ad un sottogruppo di $Perm(\mathbb{A})$.

Generalizzando, questa costruzione ci dice che qualsiasi gruppo può essere visto come un gruppo di permutazioni, dunque, in un certo senso, anche la teoria dei gruppi non è altro che la teoria delle permutazioni.

Esempio:

- Applichiamo il morfismo π al gruppo $(\mathbb{Z}_2, +, 0, \cdot^{-1})$, in cui $Z_2 = \{0, 1, 2\}$.

  Otteniamo le seguenti permutazioni:

  - $\pi_0 : 0 + 0 = 0, 1 + 0 = 1, 2 + 0 = 2$.
  - $\pi_1 : 0 + 1 = 1, 1 + 1 = 2, 2 + 1 = 0$

- $\pi_2 : 0 + 2 = 2, 1 + 2 = 0, 2 + 2 = 1$

Otteniamo dunque un sottogruppo del gruppo $(Perm(Z_2), \circ, id, \cdot^{-1})$, in quanto ad esempio la permutazione $f : 0 \mapsto 0, 1 \mapsto 2, 2 \mapsto 1$ non l'abbiamo ottenuta, dunque non è nell'immagine dell'isomorfismo.

▼ 12.0 - Esercizi con Matita

▼ Lab 1

```
(* Saltate le righe seguenti dove vengono dati gli assiomi e definite
   le notazioni e cercate Exercise 1. *)

include "basics/logic.ma".
include "basics/core_notation.ma".

notation "hvbox(A break ⇔ B)" left associative with precedence 30 for
@{'iff $A $B}.
interpretation "iff" 'iff A B = (iff A B).

(* set, ∈ *)
axiom set: Type[0].
axiom mem: set → set → Prop.
axiom incl: set → set → Prop.

notation "hvbox(a break ∈ U)" non associative with precedence 50 for
@{'mem $a $U}.
interpretation "mem" 'mem = mem.

notation "hvbox(a break ⊆ U)" non associative with precedence 50 for
@{'incl $a $U}.
interpretation "incl" 'incl = incl.

(* Extensionality *)
axiom ax_extensionality: ∀A, B. (∀Z. Z ∈ A ⇔ Z ∈ B) → A = B.

(* Inclusion *)
axiom ax_inclusion1: ∀A, B. A ⊆ B → (∀Z. Z ∈ A → Z ∈ B).
axiom ax_inclusion2: ∀A, B. (∀Z. Z ∈ A → Z ∈ B) → A ⊆ B.

(* Emptyset  ∅ *)
axiom emptyset: set.

notation "∅" non associative with precedence 90 for @{'emptyset}.
interpretation "emptyset" 'emptyset = emptyset.

axiom ax_empty: ∀X. (X ∈ ∅)→ False.

(* Intersection ∩ *)
axiom intersect: set → set → set.

notation "hvbox(A break ∩ B)" left associative with precedence 80 for
@{'intersect $A $B}.
interpretation "intersect" 'intersect = intersect.

axiom ax_intersect1: ∀A,B. ∀Z. (Z ∈ A ∩ B → Z ∈ A ∧ Z ∈ B).
axiom ax_intersect2: ∀A,B. ∀Z. (Z ∈ A ∧ Z ∈ B → Z ∈ A ∩ B).

(* Union ∪ *)
axiom union: set → set → set.

notation "hvbox(A break ∪ B)" left associative with precedence 70 for
@{'union $A $B}.
interpretation "union" 'union = union.

axiom ax_union1: ∀A,B. ∀Z. (Z ∈ A ∪ B → Z ∈ A ∨ Z ∈ B).
axiom ax_union2: ∀A,B. ∀Z. (Z ∈ A ∨ Z ∈ B → Z ∈ A ∪ B).

notation "'ABSURDUM' A" non associative with precedence 89 for @{'absurdum $A}.
interpretation "ex_false" 'absurdum A = (False_ind ? A).

(* Da qui in avanti riempite i puntini e fate validar quello che scrivete
   a Matita usando le icone con le frecce. *)

(* Exercise 1 *)
theorem reflexivity_inclusion: ∀A. A ⊆ A.
 assume A:set
 we need to prove (∀Z. Z ∈ A → Z ∈ A) (ZAtoZA)
  assume Z:set
  suppose (Z ∈ A) (ZA)
  by ZA (* Quale ipotesi serve? Osservate cosa bisogna dimostrare *)
 done
```

```
 by ax_inclusion2, ZAtoZA (* Quale ipotesi devo combinare con l'assioma? *)
done
qed.

(* Exercise 2 *)
theorem transitivity_inclusion: ∀A,B,C. A ⊆ B → B ⊆ C → A ⊆ C.
 assume A:set
 assume B:set
 assume C:set
 suppose (A ⊆ B) (AB)
 suppose (B ⊆ C) (BC)
 we need to prove (∀Z. Z ∈ A → Z ∈ C) (ZAtoZC)
  assume Z:set
  suppose (Z ∈ A) (ZA)
  by AB, ax_inclusion1, ZA we proved (Z ∈ B) (ZB)
  by BC, ax_inclusion1, ZB we proved (Z ∈ C) (ZC)(* Osservate bene cosa deve essere dimostrato *)
 done
 by ax_inclusion2, ZAtoZC
done
qed.

(* Exercise 3 *)
theorem antisymmetry_inclusion: ∀A,B. A ⊆ B → B ⊆ A → A = B.
 assume A:set
 assume B:set
 suppose (A ⊆ B) (AB)
 suppose (B ⊆ A) (BA)
 we need to prove (∀Z. Z ∈ A ⇔ Z ∈ B) (P)
  assume Z:set
  by AB, ax_inclusion1 we proved (Z ∈ A → Z ∈ B) (AB')
  by BA, ax_inclusion1 we proved (Z ∈ B → Z ∈ A) (BA')
  by conj, AB', BA'
 done
 by ax_extensionality, P  (* Quale assioma devo utilizzare? *)
done
qed.

(* Exercise 4 *)
theorem intersection_idempotent_1: ∀A. A ⊆ A ∩ A.
 assume A:set
 we need to prove (∀Z. Z ∈ A → Z ∈ (A ∩ A)) (AinAA)
  assume Z:set
  suppose (Z ∈ A) (ZA)
  we need to prove (Z ∈ A ∧ Z ∈ A) (ZAZA)
   by conj, ZA (* Il teorema conj serve per dimostrare una congiunzione (un "and" ∧) *)
  done
  by ax_intersect2, ZAZA (* Cosa stiamo dimostrando? Che assioma serve? *)
 done
 by ax_inclusion2, AinAA
done
qed.

(* Exercise 5*)
theorem intersect_monotone_l: ∀A,A',B. A ⊆ A' → A ∩ B ⊆ A' ∩ B.
 assume A:set
 assume A':set
 assume B:set
 suppose (A ⊆ A') (AA')
 we need to prove (∀Z. Z ∈ A ∩ B → Z ∈ A' ∩ B) (P)
  assume Z:set
  suppose (Z ∈ A ∩ B) (F)
  by ax_intersect1, F we proved (Z ∈ A ∧ Z ∈ B) (ZAZB)
  by ZAZB we have (Z ∈ A) (ZA) and (Z ∈ B) (ZB) (* Da un'ipotesi congiunzione si ricavano due ipotesi distinte *)
  by ax_inclusion1, AA' we proved (Z ∈ A') (ZA')
  by conj  we proved (Z ∈ A' ∧ Z ∈ B) (ZAZB')
  by ax_intersect2 (* Cosa stiamo dimostrando? Che assioma serve? *)
 done
by ax_inclusion2, P
done
qed.

(* Exercise 6*)
theorem intersect_monotone_r: ∀A,B,B'. B ⊆ B' → A ∩ B ⊆ A ∩ B'.
 assume A:set
 assume B:set
 assume B':set
 suppose (B ⊆ B') (BB')
 we need to prove (∀Z. Z ∈ A ∩ B → Z ∈ A ∩ B') (P)
  assume Z:set
```

```
  suppose (Z ∈ A ∩ B) (F)
  by ax_intersect1, F we proved (Z ∈ A ∧ Z ∈ B) (ZAZB)
  by ZAZB we have (Z ∈ A) (ZA) and (Z ∈ B) (ZB)
  by ax_inclusion1, BB' we proved (Z ∈ B') (ZB')
  by conj we proved (Z ∈ A ∧ Z ∈ B') (ZAZB')
  by ax_intersect2
 done
by ax_inclusion2, P
done
qed.
```

▼ Lab 2

```
(* Saltate le righe seguenti dove vengono dati gli assiomi e definite
   le notazioni e cercate Exercise 1. *)

include "basics/logic.ma".
include "basics/core_notation.ma".

notation "hvbox(A break ⇔ B)" left associative with precedence 30 for
@{'iff $A $B}.
interpretation "iff" 'iff A B = (iff A B).

(* set, ∈ *)
axiom set: Type[0].
axiom mem: set → set → Prop.
axiom incl: set → set → Prop.

notation "hvbox(a break ∈ U)" non associative with precedence 50 for
@{'mem $a $U}.
interpretation "mem" 'mem = mem.

notation "hvbox(a break ⊆ U)" non associative with precedence 50 for
@{'incl $a $U}.
interpretation "incl" 'incl = incl.

(* Extensionality *)
axiom ax_extensionality: ∀A, B. (∀Z. Z ∈ A ⇔ Z ∈ B) → A = B.

(* Inclusion *)
axiom ax_inclusion1: ∀A, B. A ⊆ B → (∀Z. Z ∈ A → Z ∈ B).
axiom ax_inclusion2: ∀A, B. (∀Z. Z ∈ A → Z ∈ B) → A ⊆ B.

(* Emptyset  ∅ *)
axiom emptyset: set.

notation "∅" non associative with precedence 90 for @{'emptyset}.
interpretation "emptyset" 'emptyset = emptyset.

axiom ax_empty: ∀X. (X ∈ ∅)→ False.

(* Intersection ∩ *)
axiom intersect: set → set → set.

notation "hvbox(A break ∩ B)" left associative with precedence 80 for
@{'intersect $A $B}.
interpretation "intersect" 'intersect = intersect.

axiom ax_intersect1: ∀A,B. ∀Z. (Z ∈ A ∩ B → Z ∈ A ∧ Z ∈ B).
axiom ax_intersect2: ∀A,B. ∀Z. (Z ∈ A ∧ Z ∈ B → Z ∈ A ∩ B).

(* Union ∪ *)
axiom union: set → set → set.

notation "hvbox(A break ∪ B)" left associative with precedence 70 for
@{'union $A $B}.
interpretation "union" 'union = union.

axiom ax_union1: ∀A,B. ∀Z. (Z ∈ A ∪ B → Z ∈ A ∨ Z ∈ B).
axiom ax_union2: ∀A,B. ∀Z. (Z ∈ A ∨ Z ∈ B → Z ∈ A ∪ B).

notation "'ABSURDUM' A" non associative with precedence 89 for @{'absurdum $A}.
interpretation "ex_false" 'absurdum A = (False_ind ? A).

(* Da qui in avanti riempite i puntini e fate validar quello che scrivete
   a Matita usando le icone con le frecce. *)
```

```
(* Exercise 1 *)
theorem reflexivity_inclusion: ∀A. A ⊆ A.
 assume A:set
 we need to prove (∀Z. Z ∈ A → Z ∈ A) (ZAtoZA)
  assume Z:set
  suppose (Z ∈ A) (ZA)
  by ZA (* Quale ipotesi serve? Osservate cosa bisogna dimostrare *)
 done
 by ax_inclusion2, ZAtoZA (* Quale ipotesi devo combinare con l'assioma? *)
done
qed.

(* Exercise 2 *)
theorem transitivity_inclusion: ∀A,B,C. A ⊆ B → B ⊆ C → A ⊆ C.
 assume A:set
 assume B:set
 assume C:set
 suppose (A ⊆ B) (AB)
 suppose (B ⊆ C) (BC)
 we need to prove (∀Z. Z ∈ A → Z ∈ C) (ZAtoZC)
  assume Z:set
  suppose (Z ∈ A) (ZA)
  by AB, ax_inclusion1, ZA we proved (Z ∈ B) (ZB)
  by BC, ax_inclusion1, ZB (* Osservate bene cosa deve essere dimostrato *)
 done
 by ZAtoZC, ax_inclusion2
done
qed.

(* Exercise 3 *)
theorem antisymmetry_inclusion: ∀A,B. A ⊆ B → B ⊆ A → A = B.
 assume A: set
 assume B: set
 suppose (A ⊆ B) (AB)
 suppose (B ⊆ A) (BA)
 we need to prove (∀Z. Z ∈ A ⇔ Z ∈ B) (P)
  assume Z: set
  by AB, ax_inclusion1 we proved (Z ∈ A → Z ∈ B) (AB')
  by BA, ax_inclusion1  we proved (Z ∈ B → Z ∈ A) (BA')
  by conj, AB', BA'
 done
 by ax_extensionality, P  (* Quale assioma devo utilizzare? *)
done
qed.

(* Exercise 4 *)
theorem intersection_idempotent_1: ∀A. A ⊆ A ∩ A.
 assume A: set
 we need to prove (∀Z. Z ∈ A → Z ∈ A ∩ A) (AinAA)
  assume Z: set
  suppose (Z ∈ A) (ZA)
  we need to prove (Z ∈ A ∧ Z ∈ A) (ZAZA)
   by conj, ZA (* Il teorema conj serve per dimostrare una congiunzione (un "and" ∧) *)
  done
  by ax_intersect2, ZAZA (* Cosa stiamo dimostrando? Che assioma serve? *)
 done
 by ax_inclusion2, AinAA
done
qed.

(* Exercise 5*)
theorem intersect_monotone_l: ∀A,A',B. A ⊆ A' → A ∩ B ⊆ A' ∩ B.
 assume A: set
 assume A': set
 assume B: set
 suppose (A ⊆ A') (H)
 we need to prove (∀Z. Z ∈ A ∩ B → Z ∈ A' ∩ B) (K)
  assume Z: set
  suppose (Z ∈ A ∩ B) (ZAB)
  by ax_intersect1, ZAB we proved (Z ∈ A ∧ Z ∈ B) (ZAZB)
  by ZAZB we have (Z ∈ A) (ZA) and (Z ∈ B) (ZB) (* Da un'ipotesi congiunzione si ricavano due ipotesi distinte *)
  by ax_inclusion1,ZA we proved (Z ∈ A') (ZA')
  by conj,ZA',ZB we proved (Z ∈ A' ∧ Z ∈ B) (ZAZB')
  by ax_intersect2,ZAZB' (* Cosa stiamo dimostrando? Che assioma serve? *)
 done
by ax_inclusion2,K
done
qed.
```

```
(* Exercise 6*)
theorem intersect_monotone_r: ∀A,B,B'. B ⊆ B' → A ∩ B ⊆ A ∩ B'.
 assume A: set
 assume B: set
 assume B': set
 suppose (B ⊆ B') (H)
 we need to prove (∀Z. Z ∈ A ∩ B → Z ∈ A ∩ B') (K)
  assume Z: set
  suppose (Z ∈ A ∩ B) (ZAB)
  by ax_intersect1, ZAB we proved (Z ∈ A ∧ Z ∈ B) (ZAZB)
  by ZAZB we have (Z ∈ A) (ZA) and (Z ∈ B) (ZB)
  by ax_inclusion1,ZB we proved (Z ∈ B') (ZB')
  by conj,ZA,ZB' we proved (Z ∈ A ∧ Z ∈ B') (ZAZB')
  by ax_intersect2,ZAZB'
 done
 by ax_inclusion2,K
done
qed.

(* Nuovi esercizi, secondo laboratorio *)

(* Exercise 7 *)
theorem union_inclusion: ∀A, B. A ⊆ A ∪ B.
  assume A: set
  assume B: set
  we need to prove (∀Z. Z ∈ A → Z ∈ A ∪ B) (I)
    assume Z: set
    suppose (Z∈A) (ZA)
    we need to prove (Z ∈ A ∨ Z ∈ B) (I1)
      by ZA,or_introl
    done
    by ax_union2, I1
  done
  by ax_inclusion2, I done
qed.

(* Exercise 8 *)
theorem union_idempotent: ∀A. A ∪ A = A.
 assume A: set
 we need to prove (∀Z. Z ∈ A ∪ A ⇔ Z ∈ A) (II)
 assume Z: set
  we need to prove (Z ∈ A ∪ A → Z ∈ A) (I1)
   suppose (Z ∈ A ∪ A)(Zu)
   by ax_union1, Zu we proved (Z ∈ A ∨ Z ∈ A) (Zor)
   we proceed by cases on Zor to prove (Z ∈ A)
    case or_introl
       suppose (Z∈A) (H)
       by H
    done
    case or_intror
       suppose (Z∈A) (H)
       by H
    done
  we need to prove (Z ∈ A → Z ∈ A ∪ A) (I2)
  suppose (Z∈A) (ZA)
   by ZA, or_introl we proved (Z ∈ A ∨ Z ∈ A) (Zor)
   by ax_union2, Zor
  done
  by conj, I1, I2
 done
 by II,ax_extensionality
done
qed.

(* Exercize 9 *)
theorem empty_absurd: ∀X, A. X ∈ ∅ → X ∈ A.
 assume X: set
 assume A: set
 suppose (X ∈ ∅) (XE)
 by ax_empty we proved False (bottom)
 using (ABSURDUM bottom)
done
qed.

(* Exercise 10 *)
theorem intersect_empty: ∀A. A ∩ ∅ = ∅.
 assume A: set
 we need to prove (∀Z. Z ∈ A ∩ ∅ ⇔ Z ∈ ∅) (II)
   assume Z: set
```

```
   we need to prove (Z ∈ A ∩ ∅ → Z ∈ ∅) (I1)
     suppose (Z ∈ A ∩ ∅) (Ze)
     we need to prove (Z ∈ ∅)
     by Ze, ax_intersect1 we have (Z ∈ A) (ZA) and (Z∈∅) (ZE)
     by ZE
   done
   we need to prove (Z ∈ ∅ → Z ∈ A ∩ ∅) (I2)
     suppose (Z ∈ ∅) (ZE)
     by ZE, ax_empty we proved False (bottom)
     using (ABSURDUM bottom)
   done
   by I1, I2, conj
 done
 by II, ax_extensionality
done
qed.

(* Exercise 11 *)
theorem union_empty: ∀A. A ∪ ∅ = A.
 assume A: set
 we need to prove (∀Z. Z ∈ A ∪ ∅ ⇔ Z ∈ A) (II)
   assume Z:set
   we need to prove (Z ∈ A → Z ∈ A ∪ ∅) (I1)
     suppose (Z ∈ A) (ZA)
     by or_introl, ZA we proved (Z ∈ A ∨ Z ∈ ∅) (Zor)
     by ax_union2,Zor
   done
   we need to prove (Z ∈ A ∪ ∅ → Z ∈ A) (I2)
     suppose (Z ∈ A ∪ ∅) (Zu)
     by ax_union1, Zu we proved (Z ∈ A ∨ Z ∈ ∅) (Zor)
     we proceed by cases on Zor to prove (Z ∈ A)
      case or_introl
        suppose (Z ∈ A) (H)
        by H done
      case or_intror
        suppose (Z ∈ ∅) (H)
         by ax_empty, H we proved False (bottom)
         using (ABSURDUM bottom)
      done
    by conj, I1, I2
  done
  by ax_extensionality, II
 done
qed.

(* Exercise 12 *)
theorem union_commutative: ∀A,B. A ∪ B = B ∪ A.
 assume A: set
 assume B: set
 we need to prove (∀Z. Z ∈ A ∪ B ⇔ Z ∈ B ∪ A) (II)
   assume Z: set
   we need to prove (Z ∈ A ∪ B → Z ∈ B ∪ A) (I1)
     suppose (Z ∈ A ∪ B) (ZAB)
     we need to prove (Z ∈ B ∪ A)
     we need to prove (Z ∈ B ∨ Z ∈ A) (I)
       by ZAB, ax_union1 we proved (Z∈A∨Z∈B) (Zor)
       we proceed by cases on Zor to prove (Z ∈ B ∨ Z ∈ A)
       case or_intror
         suppose (Z∈B) (H)
         by H,or_introl
       done
       case or_introl
         suppose (Z∈A) (H)
         by H, or_intror
       done
   by I,ax_union2 done
   we need to prove (Z ∈ B ∪ A → Z ∈ A ∪ B) (I2)
     suppose (Z ∈ B ∪ A) (ZBA)
     we need to prove (Z ∈ A ∪ B)
     we need to prove (Z ∈ A ∨ Z ∈ B) (I)
       by ZBA, ax_union1 we proved (Z∈B∨Z∈A) (Zor)
       we proceed by cases on Zor to prove (Z ∈ A ∨ Z ∈ B)
       case or_intror
         suppose (Z∈A) (H)
         by H,or_introl
       done
       case or_introl
         suppose (Z∈B) (H)
         by H, or_intror
```

```
      done
   by I,ax_union2 done
  by I1,I2,conj
 done
 by ax_extensionality,II
done
qed.
```

▼ Lab 3

```
include "basics/logic.ma".


(* Esercizio 1
   ===========

   Definire il seguente linguaggio (o tipo) di espressioni riempiendo gli spazi.

   Expr :: "Zero" | "One" | "Minus" Expr | "Plus" Expr Expr | "Mult" Expr Expr
*)
inductive Expr : Type[0] ≝
| Zero: Expr
| One: Expr
| Minus: Expr → Expr
| Plus: Expr → Expr → Expr
| Mult: Expr → Expr → Expr
.

(* La prossima linea è un test per verificare se la definizione data sia
   probabilmente corretta. Essa definisce `test_Expr` come un'abbreviazione
   dell'espressione `Mult Zero (Plus (Minus One) Zero)`, verificando inoltre
   che l'espressione soddisfi i vincoli di tipo dichiarati sopra. Eseguitela. *)

definition test_Expr : Expr ≝ Mult Zero (Plus (Minus One) Zero).

(* Come esercizio, provate a definire espressioni che siano scorrette rispetto
   alla grammatica/sistema di tipi. Per esempio, scommentate la seguenti
   righe e osservate i messaggi di errore:

definition bad_Expr1 : Expr ≝ Mult Zero.
definition bad_Expr2 : Expr ≝ Mult Zero Zero Zero.
definition bad_Expr3 : Expr ≝ Mult Zero Plus.
*)

(* Esercizio 2
   ===========

   Definire il linguaggio (o tipo) dei numeri naturali.

   nat ::= "O" | "S" nat
*)

inductive nat : Type[0] ≝
 O : nat
 | S : nat → nat
.

definition one : nat ≝ S O.
definition two : nat ≝ S (S O).
definition three : nat ≝ S (S (S O)).

(* Esercizio 3
   ===========

   Definire il linguaggio (o tipo) delle liste di numeri naturali.

   list_nat ::= "Nil" | "Cons" nat list_nat

   dove Nil sta per lista vuota e Cons aggiunge in testa un numero naturale a
   una lista di numeri naturali.

   Per esempio, `Cons O (Cons (S O) (Cons (S (S O)) Nil))` rappresenta la lista
   `[1,2,3]`.
*)

inductive list_nat : Type[0] ≝
 Nil : list_nat
```

```
 | Cons : nat → list_nat → list_nat
.

(* La seguente lista contiene 1,2,3 *)
definition one_two_three : list_nat ≝ Cons one (Cons two (Cons three Nil)).

(* Completate la seguente definizione di una lista contenente due uni. *)

definition one_one : list_nat ≝ Cons one (Cons one Nil).


(* Osservazione
   ===========

   Osservare come viene definita la somma di due numeri in Matita per
   ricorsione strutturale sul primo.

   plus O m = m
   plus (S x) m = S (plus x m) *)

let rec plus n m on n ≝
 match n with
 [ O ⇒ m
 | S x ⇒ S (plus x m) ].

(* Provate a introdurre degli errori nella ricorsione strutturale. Per esempio,
   omettete uno dei casi o fate chiamate ricorsive non strutturali e osservate
   i messaggi di errore di Matita. *)

(* Per testare la definizione, possiamo dimostrare alcuni semplici teoremi la
   cui prova consiste semplicemente nell'effettuare i calcoli. *)
theorem test_plus: plus one two = three.
done. qed.

(* Esercizio 4
   ===========

   Completare la seguente definizione, per ricorsione strutturale, della
   funzione `size_E` che calcola la dimensione di un'espressione in numero
   di simboli.

   size_E Zero = 1
   size_E One = 1
   size_E (Minus E) = 1 + size_E E
   ...
*)
let rec size_E E on E ≝
 match E with
  [    ⇒ one
  | One ⇒ one
  | Minus E ⇒ plus one (size_E E)
  | Plus E1 E2 ⇒ plus one (plus (size_E E1) (size_E E2))
  | Mult E1 E2 ⇒ plus one (plus (size_E E1) (size_E E2))
  ]
.

theorem test_size_E : size_E test_Expr = plus three three.
done. qed.

(* Esercizio 5
   ===========

   Definire in Matita la funzione `sum` che, data una `list_nat`, calcoli la
   somma di tutti i numeri contenuti nella lista. Per esempio,
   `sum one_two_three` deve calcolare sei.
*)

definition zero : nat ≝ O.

let rec sum L on L ≝
 match L with
 [ Nil ⇒ zero
 | Cons N TL ⇒ plus N (sum TL)]
.

theorem test_sum : sum one_two_three = plus three three.
done. qed.
```

```
(* Esercizio 6
   ===========

   Definire la funzione binaria `append` che, date due `list_nat` restituisca la
   `list_nat` ottenuta scrivendo in ordine prima i numeri della prima lista in
   input e poi quelli della seconda.

   Per esempio, `append (Cons one (Cons two Nil)) (Cons 0 Nil)` deve restituire
   `Cons one (Cons two (Cons 0 nil))`. *)
let rec append lista1 lista2 on lista1 ≝
 match lista1 with
 [ Nil ⇒ lista2
 | Cons N TL ⇒ append TL (Cons N lista2)]
.

theorem test_append : append (Cons one Nil) (Cons two (Cons three Nil)) = one_two_three.
done. qed.
```

▼ Lab 4

```
(* ATTENZIONE
   ==========

   Non modificare la seguente riga che carica la definizione di uguaglianza.
*)

include "basics/logic.ma".

(* ATTENZIONE
   ==========

   Quanto segue sono definizioni di tipi di dato/grammatiche e funzioni
   definite per ricorsione strutturale prese dall'esercitazione della volta
   scorsa. Non cambiarle e procedere con i nuovi esercizi di dimostrazione
   che sono intervallati con le definizioni.
*)

(* nat ::= "O" | "S" nat   *)
inductive nat : Type[0] ≝
   O : nat
 | S : nat → nat.

definition one : nat ≝ S O.
definition two : nat ≝ S (S O).
definition three : nat ≝ S (S (S O)).

(* list_nat ::= "Nil" | "Cons" nat list_nat *)
inductive list_nat : Type[0] ≝
   Nil : list_nat
 | Cons : nat → list_nat → list_nat.

(* plus O m = m
   plus (S x) m = S (plus x m)
*)
let rec plus n m on n ≝
 match n with
 [ O ⇒ m
 | S x ⇒ S (plus x m) ].

(* La funzione `sum` che, data una `list_nat`, calcola la
   somma di tutti i numeri contenuti nella lista. *)
let rec sum L on L ≝
 match L with
 [ Nil ⇒ O
 | Cons N TL ⇒ plus N (sum TL)
 ].

(* La funzione binaria `append` che, date due `list_nat` restituisca la
   `list_nat` ottenuta scrivendo in ordine prima i numeri della prima lista in
   input e poi quelli della seconda.
*)
let rec append L1 L2 on L1 ≝
 match L1 with
 [ Nil ⇒ L2
 | Cons HD TL ⇒ Cons HD (append TL L2)
 ].
```

```
(* Esercizio 1
   ===========

   Dimostrare l'associatività della somma per induzione strutturale su x.
*)
theorem plus_assoc: ∀x,y,z. plus x (plus y z) = plus (plus x y) z.
 (* Possiamo iniziare fissando una volta per tutte le variabili x,y,z
    A lezione vedremo il perchè. *)
 assume x : nat
 assume y : nat
 assume z : nat
 we proceed by induction on x to prove (plus x (plus y z) = plus (plus x y) z)
 case O
  (* Scriviamo cosa deve essere dimostrato e a cosa si riduce eseguendo le
     definizioni. *)
  we need to prove (plus O (plus y z) = plus (plus O y) z)
  that is equivalent to (plus y z = plus y z)
  (* done significa ovvio *)
  done
 case S (w: nat)
  (* Chiamiamo l'ipotesi induttiva IH e scriviamo cosa afferma
     Ricordate: altro non è che la chiamata ricorsiva su w. *)
  by induction hypothesis we know (plus w (plus y z) = plus (plus w y) z) (IH)
  we need to prove (plus (S w) (plus y z) = plus (plus (S w) y) z)
  that is equivalent to (S (plus w (plus y z)) = plus (S (plus w y)) z)
  (* by IH done significa ovvio considerando l'ipotesi IH *)
  by IH done
qed.

(* Esercizio 2
   ===========

   Definire il linguaggio degli alberi binari (= dove ogni nodo che non è una
   foglia ha esattamente due figli) le cui foglie siano numeri naturali.

   tree_nat ::= "Leaf" nat | "Node" nat nat
*)

inductive tree_nat : Type[0] ≝
   Leaf : nat → tree_nat
 | Node : tree_nat → tree_nat → tree_nat.

(* Il seguente albero binario ha due foglie, entrambe contenenti uni. *)
definition one_one_tree : tree_nat ≝ Node (Leaf one) (Leaf one).

(* Definite l'albero       /\
                          0 /\
                           1  2  *)
definition zero_one_two_tree : tree_nat ≝
 Node (Leaf O) (Node (Leaf one) (Leaf two)).


(* Esercizio 3
   ===========

   Definire la funzione `rightmost` che, dato un `tree_nat`, restituisca il
   naturale contenuto nella foglia più a destra nell'albero. *)

let rec rightmost tree on tree ≝
  match tree with
  [ Leaf n ⇒ n
  | Node tree1 tree2 ⇒ rightmost tree2
  ].

theorem test_rightmost : rightmost zero_one_two_tree = two.
done. qed.


(* Esercizio 4
   ============

   Definire la funzione `visit` che, dato un `tree_nat`, calcoli la `list_nat`
   che contiene tutti i numeri presenti nelle foglie dell'albero in input,
   nell'ordine in cui compaiono nell'albero da sinistra a destra.

   Suggerimento: per definire tree_nat usare la funzione `append` già definita
   in precedenza.
```

```
   Esempio: `visit zero_one_two_tree = Cons O (Cons one (Cons two Nil))`.
*)

let rec visit T on T ≝
 match T with
 [ Leaf n ⇒ Cons n Nil
 | Node T1 T2 ⇒ append (visit T1) (visit T2)
 ].

theorem test_visit : visit zero_one_two_tree = Cons O (Cons one (Cons two Nil)).
done. qed.


(* Esercizio 5
   ===========

   La somma di tutti i numeri nella concatenazione di due liste è uguale
   alla somma delle somme di tutti i numeri nelle due liste. *)

theorem sum_append: ∀L1,L2. sum (append L1 L2) = plus (sum L1) (sum L2).
 assume L1 : list_nat
 assume L2 : list_nat
 we proceed by induction on L1 to prove (sum (append L1 L2) = plus (sum L1) (sum L2))
 case Nil
  we need to prove (sum (append Nil L2) = plus (sum Nil) (sum L2))
  that is equivalent to (sum L2 = plus O (sum L2))
  done
  case Cons (N: nat) (L: list_nat)
  by induction hypothesis we know (sum (append L L2) = plus (sum L) (sum L2)) (IH)
  we need to prove (sum (append (Cons N L) L2) = plus (sum (Cons N L)) (sum L2))
  that is equivalent to (sum (Cons N (append L L2)) = plus (plus N (sum L)) (sum L2))
  that is equivalent to (plus N (sum(append L L2)) = plus (plus N (sum L)) (sum L2))
  (* Per concludere servono sia l'ipotesi induttiva IH che il teorema plus_assoc
     dimostrato prima. Convincetevene

     Nota: se omettete IH, plus_assoc o entrambi Matita ci riesce lo stesso
     Rendere stupido un sistema intelligente è complicato... Tuttavia non
     abusatene: quando scrivete done cercate di avere chiaro perchè il teorema
     è ovvio e se non vi è chiaro, chiedete. *)
  by IH, plus_assoc done
qed.

(* La funzione `plusT` che, dato un `tree_nat`, ne restituisce la
   somma di tutte le foglie. *)
let rec plusT T on T ≝
 match T with
 [ Leaf n ⇒ n
 | Node t1 t2 ⇒ plus (plusT t1) (plusT t2)
 ].


(* Esercizio 6
   ===========

   Iniziare a fare l'esercizio 7, commentando quel poco che c'è dell'esercizio 6
   Nel caso base vi ritroverete, dopo la semplificazione, a dover dimostrare un
   lemma non ovvio. Tornate quindi all'esercizio 3 che consiste nell'enunciare e
   dimostrare il lemma. *)

lemma plus_O: ∀N. N = plus N O.
  assume N : nat
  we proceed by induction on N to prove (N = plus N O)
  case O
    we need to prove (O = plus O O)
    that is equivalent to (O=O)
    done
  case S (x : nat)
    by induction hypothesis we know (x = plus x O) (II)
    we need to prove (S x = plus (S x) O)
    that is equivalent to (S x = S (plus x O))
    by II
    done
qed.
(* Esercizio 7
   ===========

   Dimostriamo che la `plusT` è equivalente a calcolare la `sum` sul risultato
   di una `visit`. *)
```

```
theorem plusT_sum_visit: ∀T. plusT T = sum (visit T).
 assume T : tree_nat
 we proceed by induction on T to prove (plusT T = sum (visit T))
 case Leaf (N : nat)
  we need to prove (plusT (Leaf N) = sum (visit (Leaf N)))
  that is equivalent to (N = sum (Cons N Nil))
  that is equivalent to (N = plus N (sum Nil))
  that is equivalent to (N = plus N O)
  (* Ciò che dobbiamo dimostrare non è ovvio (perchè?). Per proseguire,
     completate l'esercizio 6 enunciando e dimostrando il lemma che vi serve
     Una volta risolto l'esercizio 6, questo ramo diventa ovvio usando il lemma.*)
  by plus_O done
 case Node (T1:tree_nat) (T2:tree_nat)
  by induction hypothesis we know (plusT T1 = sum (visit T1)) (IH1)
  by induction hypothesis we know (plusT T2 = sum (visit T2)) (IH2)
  we need to prove (plusT (Node T1 T2)=sum (visit (Node T1 T2)))
  that is equivalent to (plus (plusT T1) (plusT T2) = sum (append (visit T1) (visit T2)))
  (* Oltre alla due ipotesi induttive, di quale altro lemma dimostrato in
     precedenza abbiamo bisogno per concludere la prova?*)
  by IH1,IH2,sum_append done
qed.


(* Un altro modo di calcolare la somma di due numeri: per ricorsione strutturale
   sul secondo argomento.

   plus' m O = m
   plus' m (S x) = S (plus' m x)
*)
let rec plus' m n on n ≝
 match n with
 [ O ⇒ m
 | S x ⇒ S (plus' m x) ].


(* Esercizio 8
   ============

   Dimostriamo l'equivalenza dei due metodi di calcolo
   Vi servirà un lemma: capite quale e dimostratelo
*)

lemma plus_O': ∀y. y = plus' O y.
  assume y : nat
  we proceed by induction on y to prove (y = plus' O y)
  case O
    we need to prove (O = plus' O O)
    that is equivalent to (O = O)
    done
  case S (n : nat)
    by induction hypothesis we know (n = plus' O n) (II)
    we need to prove (S n = plus' O (S n))
    that is equivalent to (S n = S (plus' O n))
    by II
    done
qed.

lemma plus_S': ∀y,z. S (plus' z y) = plus' (S z) y.
  assume y : nat
  we proceed by induction on y to prove (∀z. S (plus' z y) = plus' (S z) y)
  case O
    we need to prove (∀z. S(plus' z O) = plus' (S z) O)
    that is equivalent to (∀z. S z = S z)
    done
  case S (g : nat)
    by induction hypothesis we know (∀z. S(plus' z g) = plus' (S z) g) (II)
    we need to prove (∀z. S (plus' z (S g)) = plus' (S z) (S g))
    that is equivalent to (∀z. S (S (plus' z g)) = S (plus' (S z) g))
    by II
    done
qed.


theorem plus_plus': ∀x,y. plus x y = plus' x y.
 (* Nota: la dimostrazione è più facile se andate per induzione su y perchè
    potrete riciclare un lemma già dimostrato.
    Se andate per induzione su x vi verrà lo stesso, ma in tal caso avrete
    bisogno di due lemmi, ognuno dei quali non ancora dimostrati. *)
```

```
 assume x: nat
 we proceed by induction on x to prove (∀y. plus x y = plus' x y)
 case O
  we need to prove (∀y. plus O y = plus' O y)
  that is equivalent to (∀y. y = plus' O y)
  by plus_O'
  done
 case S (z:nat)
  by induction hypothesis we know (∀y. plus z y = plus' z y) (II)
  we need to prove (∀y. plus (S z) y = plus' (S z) y)
  that is equivalent to (∀y. S (plus z y) = plus' (S z) y)
  by II, plus_S'
  done
qed.


(* Esercizio 9: se finite prima o volete esercitarvi a casa
   ========================================================

   Dimostriamo l'equivalenza dei due metodi di calcolo plus e plus',
   questa volta per induzione sul primo argomento x. Avrete bisogno di uno o
   più lemmi, da scoprire. Ovviamente, NON è consentito usare quanto dimostrato
   all'esercizio precedente

lemma …
qed.

theorem plus_plus_new: ∀x,y. plus x y = plus' x y.
…
(* Esercizio 10,11,…
   =================

   Volete esercitarvi a casa su altre dimostrazioni facili come queste?
   Ecco due buoni spunti:

   1) definite la funzione che inserisce un numero in
      coda a una lista e usatela per definire la funzione rev che restituisce
      la lista ottenuta leggendo la lista in input dalla fine all'inizio
      Esempio:
       rev (Cons 1 (Cons 2 (Cons 3 Nil))) = (Cons 3 (Cons 2 (Cons 1 Nil)))
      Poi dimostrate che ∀L. sum (rev L) = sum L
      Per riuscirci vi serviranno una cascata di lemmi intermedi da enunciare
      e dimostrare
   2) definite una funzione leq_nat che dati due numeri naturali ritorni true
      sse il primo è minore o uguale al secondo; usatela per scrivere una funzione
      che aggiunga un elemento in una lista ordinata di numeri;
      poi usatela quest'ultima per definire una funzione "sort" che ordina una lista
      di numeri. Dimostrate che l'algoritmo è corretto procedendo
      come segue:
      a) definite, per ricorsione strutturale, il predicato ``X appartiene
         alla lista L''
      b) dimostrate che X appartiene all'inserimento di Y nella lista ordinata
         L sse X è uguale a Y oppure appartiene a L
      c) dimostrate che se X appartiene alla lista L allora appartiene alla
         lista sort L
      d) dimostrate anche il viceversa
      e) definite, per ricorsione strutturale, il predicato ``X è ordinata''
      f) dimostrate che se L è ordinata lo è anche la lista ottenuta inserendo
         X in L
      g) dimostrate che per ogni L, sort L è ordinata

      Nota: a)-e) sono esercizi semplici. Anche g) è semplice se asserite f)
      come assioma. La dimostrazione di f) invece è più difficile e
      potrebbe richiede altri lemmi ausiliari quali la transitività del
      predicato leq_nat
*)
```