▼ 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
*)
```