▼ **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:

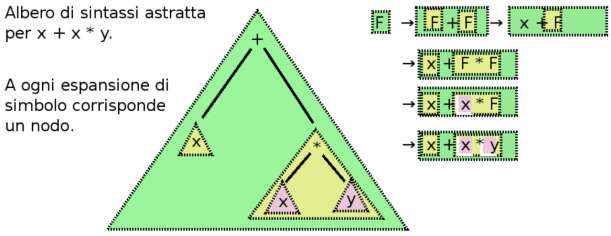
Nel linguaggio naturale la stringa “la vecchia porta la sbarra” può essere costruita in due modi:

- 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:

- ∀ 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
*)
```