

# Demostración asistida por ordenador con Coq

José A. Alonso Jiménez

---

Grupo de Lógica Computacional  
Dpto. de Ciencias de la Computación e Inteligencia Artificial  
Universidad de Sevilla  
Sevilla, 31 de julio de 2018 (versión del 3 de agosto de 2018)

Esta obra está bajo una licencia Reconocimiento-NoComercial-CompartirIgual 2.5 Spain de Creative Commons.

**Se permite:**

- copiar, distribuir y comunicar públicamente la obra
- hacer obras derivadas

**Bajo las condiciones siguientes:**

**Reconocimiento.** Debe reconocer los créditos de la obra de la manera especificada por el autor.



**No comercial.** No puede utilizar esta obra para fines comerciales.



**Compartir bajo la misma licencia.** Si altera o transforma esta obra, o genera una obra derivada, sólo puede distribuir la obra generada bajo una licencia idéntica a ésta.

- Al reutilizar o distribuir la obra, tiene que dejar bien claro los términos de la licencia de esta obra.
- alguna de estas condiciones puede no aplicarse si se obtiene el permiso del titular de los derechos de autor.

Esto es un resumen del texto legal (la licencia completa). Para ver una copia de esta licencia, visite <http://creativecommons.org/licenses/by-nc-sa/2.5/es/> o envíe una carta a Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

# Índice general

1 Programación funcional y métodos elementales de demostración en Coq	7
2 Demostraciones por inducción sobre los números naturales en Coq	35
3 Datos estructurados en Coq	57



# Introducción

En este libro se incluye unos apuntes de demostración asistida por ordenador con [Coq](#) para los cursos de

- [Razonamiento automático](#) del [Máster Universitario en Lógica, computación e inteligencia artificial](#) de la [Universidad de Sevilla](#).
- [Lógica matemática y fundamentos](#) del [Grado en Matemáticas](#) de la [Universidad de Sevilla](#).

Esencialmente los apuntes son una adaptación del libro [Software foundations \(Vol. 1: Logical foundations\)](#) de Benjamin Peirce y otros.

Una primera versión de estos apuntes se han usado este año en el [Seminario de Lógica Computacional](#).



# Tema 1

## Programación funcional y métodos elementales de demostración en Coq

*(\* T1: Programación funcional y métodos elementales de demostración en Coq \*)*

*(\* El contenido de la teoría es*

*1. Datos y funciones*

*1. Tipos enumerados*

*2. Booleanos*

*3. Tipos de las funciones*

*4. Tipos compuestos*

*5. Módulos*

*6. Números naturales*

*2. Métodos elementales de demostración*

*1. Demostraciones por simplificación*

*2. Demostraciones por reescritura*

*3. Demostraciones por análisis de casos \*)*

*(\* =====  
§ 1. Datos y funciones  
===== \*)*

*(\* =====  
§§ 1.1. Tipos enumerados  
===== \*)*

*(\* -----*

*Ejemplo 1.1.1. Definir el tipo dia cuyos constructores sean los días de la semana.*

----- \*)

**Inductive** dia: **Type** :=

```
| lunes      : dia
| martes     : dia
| miercoles  : dia
| jueves     : dia
| viernes    : dia
| sabado     : dia
| domingo    : dia.
```

(\* -----  
*Ejemplo 1.1.2. Definir la función  
 siguiente\_laborable : dia -> dia  
 tal que (siguiente\_laborable d) es el día laboral siguiente a d.*  
 ----- \*)

**Definition** siguiente\_laborable (d:dia) : dia:=

**match** d **with**

```
| lunes      => martes
| martes     => miercoles
| miercoles  => jueves
| jueves     => viernes
| viernes    => lunes
| sabado     => lunes
| domingo    => lunes
```

**end.**

(\* -----  
*Ejemplo 1.1.3. Calcular el valor de las siguientes expresiones  
 + siguiente\_laborable jueves  
 + siguiente\_laborable viernes  
 + siguiente\_laborable (siguiente\_laborable sabado)*  
 ----- \*)

Compute (siguiente\_laborable jueves).

(\* ==> viernes : dia \*)



```

Compute (siguiente_laborable viernes).
(* ==> lunes : dia *)

```

```

Compute (siguiente_laborable (siguiente_laborable sabado)).
(* ==> martes : dia *)

```

```

(* -----
   Ejemplo 1.1.4. Demostrar que
       siguiente_laborable (siguiente_laborable sabado) = martes
   ----- *)

```

```

Example siguiente_laborable1:
  siguiente_laborable (siguiente_laborable sabado) = martes.

```

```

Proof.
  simpl.      (* ⊢ martes = martes *)
  reflexivity. (* ⊢ *)

```

```

Qed.

```

```

(* =====
   §§ 1.2. Booleanos
   ===== *)

```

```

(* -----
   Ejemplo 1.2.1. Definir el tipo bool (□) cuyos constructores son true
   y false.
   ----- *)

```

```

Inductive bool : Type :=
| true  : bool
| false : bool.

```

```

(* -----
   Ejemplo 1.2.2. Definir la función
       negacion : bool -> bool
   tal que (negacion b) es la negacion de b.
   ----- *)

```

```

Definition negacion (b:bool) : bool :=
  match b with
  | true  => false

```

```
| false => true
end.
```

```
(* -----
   Ejemplo 1.2.3. Definir la función
       conjuncion : bool -> bool -> bool
   tal que (conjuncion b1 b2) es la conjunción de b1 y b2.
   ----- *)
```

```
Definition conjuncion (b1:bool) (b2:bool) : bool :=
  match b1 with
  | true  => b2
  | false => false
  end.
```

```
(* -----
   Ejemplo 1.2.4. Definir la función
       disyuncion : bool -> bool -> bool
   tal que (disyuncion b1 b2) es la disyunción de b1 y b2.
   ----- *)
```

```
Definition disyuncion (b1:bool) (b2:bool) : bool :=
  match b1 with
  | true  => true
  | false => b2
  end.
```

```
(* -----
   Ejemplo 1.2.5. Demostrar las siguientes propiedades
       disyuncion true  false = true.
       disyuncion false false = false.
       disyuncion false true  = true.
       disyuncion true  true  = true.
   ----- *)
```

```
Example disyuncion1: disyuncion true false = true.
Proof. simpl. reflexivity. Qed.
```

```
Example disyuncion2: disyuncion false false = false.
Proof. simpl. reflexivity. Qed.
```

**Example** disyuncion3: disyuncion false true = true.

**Proof.** simpl. reflexivity. Qed.

**Example** disyuncion4: disyuncion true true = true.

**Proof.** simpl. reflexivity. Qed.

```
(* -----
  Ejemplo 1.2.6. Definir los operadores (&&) y (||) como abreviaturas
  de las funciones conjuncion y disyuncion.
  ----- *)
```

**Notation** "x && y" := (conjuncion x y).

**Notation** "x || y" := (disyuncion x y).

```
(* -----
  Ejemplo 1.2.7. Demostrar que
    false || false || true = true.
  ----- *)
```

**Example** disyuncion5: false || false || true = true.

**Proof.** simpl. reflexivity. Qed.

```
(* -----
  Ejercicio 1.2.1. Definir la función
    nand : bool -> bool -> bool
  tal que (nand x y) se verifica si x e y no son verdaderos.

  Demostrar las siguientes propiedades de nand
    nand true false = true.
    nand false false = true.
    nand false true = true.
    nand true true = false.
  ----- *)
```

**Definition** nand (b1:bool) (b2:bool) : bool :=  
negacion (b1 && b2).

**Example** nand1: nand true false = true.

**Proof.** simpl. reflexivity. Qed.

**Example** nand2: nand false false = true.

**Proof.** simpl. reflexivity. Qed.

**Example** nand3: nand false true = true.

**Proof.** simpl. reflexivity. Qed.

**Example** nand4: nand true true = false.

**Proof.** simpl. reflexivity. Qed.

```
(* -----
  Ejercicio 1.2.2. Definir la función
    conjuncion3 : bool -> bool -> bool -> bool
  tal que (conjuncion3 x y z) se verifica si x, y y z son verdaderos.

  Demostrar las siguientes propiedades de conjuncion3
    conjuncion3 true  true  true  = true.
    conjuncion3 false true  true  = false.
    conjuncion3 true  false true  = false.
    conjuncion3 true  true  false = false.
  ----- *)
```

**Definition** conjuncion3 (b1:bool) (b2:bool) (b3:bool) : bool :=  
b1 && b2 && b3.

**Example** conjuncion3a: conjuncion3 true true true = true.

**Proof.** simpl. reflexivity. Qed.

**Example** conjuncion3b: conjuncion3 false true true = false.

**Proof.** simpl. reflexivity. Qed.

**Example** conjuncion3c: conjuncion3 true false true = false.

**Proof.** simpl. reflexivity. Qed.

**Example** conjuncion3d: conjuncion3 true true false = false.

**Proof.** simpl. reflexivity. Qed.

```
(* =====
  §§ 1.3. Tipos de las funciones
  ===== *)
```

```
(* -----
  Ejemplo 1.3.1. Calcular el tipo de las siguientes expresiones
    + true
    + (negacion true)
    + negacion
  ----- *)
```

**Check true.**

```
(* ==> true : bool *)
```

**Check (negacion true).**

```
(* ==> negacion true : bool *)
```

**Check negacion.**

```
(* ==> negacion : bool -> bool *)
```

```
(* =====
  §§ 1.4. Tipos compuestos
  ===== *)
```

```
(* -----
  Ejemplo 1.4.1. Definir el tipo rva cuyos constructores son rojo, verde
  y azul.
  ----- *)
```

**Inductive** rva : **Type** :=

```
| rojo   : rva
| verde  : rva
| azul   : rva.
```

```
(* -----
  Ejemplo 1.4.2. Definir el tipo color cuyos constructores son negro,
  blanco y primario, donde primario es una función de rva en color.
  ----- *)
```

**Inductive** color : **Type** :=

```
| negro   : color
| blanco  : color
| primario : rva -> color.
```

```
(* -----
Ejemplo 1.4.3. Definir la función
  monocromático : color -> bool
tal que (monocromático c) se verifica si c es monocromático.
----- *)
```

```
Definition monocromático (c : color) : bool :=
match c with
| negro      => true
| blanco     => true
| primario p => false
end.
```

```
(* -----
Ejemplo 1.4.4. Definir la función
  esRojo : color -> bool
tal que (esRojo c) se verifica si c es rojo.
----- *)
```

```
Definition esRojo (c : color) : bool :=
match c with
| negro      => false
| blanco     => false
| primario rojo => true
| primario _   => false
end.
```

```
(* =====
§§ 1.5. Módulos
===== *)
```

```
(* -----
Ejemplo 1.5.1. Iniciar el módulo Naturales.
----- *)
```

```
Module Naturales.
```

```
(* =====
§§ 1.6. Números naturales
===== *)
```

```

===== *)

(* -----
   Ejemplo 1.6.1. Definir el tipo nat de los números naturales con los
   constructores 0 (para el 0) y S (para el siguiente).
   ----- *)

Inductive nat : Type :=
| 0 : nat
| S : nat -> nat.

(* -----
   Ejemplo 1.6.2. Definir la función
   pred : nat -> nat
   tal que (pred n) es el predecesor de n.
   ----- *)

Definition pred (n : nat) : nat :=
  match n with
  | 0    => 0
  | S n' => n'
  end.

(* -----
   Ejemplo 1.6.3. Finalizar el módulo Naturales.
   ----- *)

End Naturales.

(* -----
   Ejemplo 1.6.4. Calcular el tipo y valor de la expresión
   (S (S (S (S 0)))).
   ----- *)

Check (S (S (S (S 0)))).
(* ==> 4 : nat *)

(* -----
   Ejemplo 1.6.5. Definir la función
   menosDos : nat -> nat

```

*tal que (menosDos n) es n-2.*

----- \*)

**Definition** menosDos (n : nat) : nat :=

**match** n **with**

| 0 => 0

| S 0 => 0

| S (S n') => n'

**end.**

(\* -----  
*Ejemplo 1.6.6. Evaluar la expresión (menosDos 4).*  
 ----- \*)

Compute (menosDos 4).

(\* ==> 2 : nat \*)

(\* -----  
*Ejemplo 1.6.7. Calcular et tipo de las funcionse S, pred y menosDos.*  
 ----- \*)

**Check** S.

(\* ==> S : nat -> nat \*)

**Check** pred.

(\* ==> pred : nat -> nat \*)

**Check** menosDos.

(\* ==> menosDos : nat -> nat \*)

(\* -----  
*Ejemplo 1.6.8. Definir la función*  
*esPar : nat -> bool*  
*tal que (esPar n) se verifica si n es par.*  
 ----- \*)

**Fixpoint** esPar (n:nat) : bool :=

**match** n **with**

| 0 => true

| S 0 => false



```
| S (S n') => esPar n'
end.
```

```
(* -----
  Ejemplo 1.6.9. Definir la función
    esImpar : nat -> bool
  tal que (esImpar n) se verifica si n es impar.
  ----- *)
```

**Definition** esImpar (n:nat) : bool :=  
negacion (esPar n).

```
(* -----
  Ejemplo 1.6.10. Demostrar que
    + esImpar 1 = true.
    + esImpar 4 = false.
  ----- *)
```

**Example** esImpar1: esImpar 1 = true.

**Proof.** simpl. reflexivity. Qed.

**Example** esImpar2: esImpar 4 = false.

**Proof.** simpl. reflexivity. Qed.

```
(* -----
  Ejemplo 1.6.12. Iniciar el módulo Naturales2.
  ----- *)
```

**Module** Naturales2.

```
(* -----
  Ejemplo 1.6.13. Definir la función
    suma : nat -> nat -> nat
  tal que (suma n m) es la suma de n y m. Por ejemplo,
    suma 3 2 = 5

  Nota: Es equivalente a la predefinida plus
  ----- *)
```

**Fixpoint** suma (n : nat) (m : nat) : nat :=

```

match n with
  | 0    => m
  | S n' => S (suma n' m)
end.

```

Compute (suma 3 2).

(\* ==> 5: nat \*)

```

(* -----
   Ejemplo 1.6.14. Definir la función
       producto : nat -> nat -> nat
   tal que (producto n m) es el producto de n y m. Por ejemplo,
       producto 3 2 = 6

   Nota: Es equivalente a la predefinida mult.
   ----- *)

```

```

Fixpoint producto (n m : nat) : nat :=
  match n with
  | 0    => 0
  | S n' => suma m (producto n' m)
end.

```

**Example** productol: (producto 2 3) = 6.

**Proof.** **simpl.** **reflexivity.** **Qed.**

```

(* -----
   Ejemplo 1.6.15. Definir la función
       resta : nat -> nat -> nat
   tal que (resta n m) es la diferencia de n y m. Por ejemplo,
       resta 3 2 = 1

   Nota: Es equivalente a la predefinida minus.
   ----- *)

```

```

Fixpoint resta (n m: nat) : nat :=
  match (n, m) with
  | (0 , _)    => 0
  | (S _ , 0)   => n
  | (S n' , S m') => resta n' m'

```

**end.**

```
(* -----
   Ejemplo 1.6.16. Cerrar el módulo Naturales2.
   ----- *)
```

**End** Naturales2.

```
(* -----
   Ejemplo 1.6.17. Definir la función
   potencia : nat -> nat -> nat
   tal que (potencia x n) es la potencia n-ésima de x. Por ejemplo,
   potencia 2 3 = 8

   Nota: En lugar de producto, usar la predefinida mult.
   ----- *)
```

```
Fixpoint potencia (x n : nat) : nat :=
  match n with
  | 0    => S 0
  | S m => mult x (potencia x m)
end.
```

Compute (potencia 2 3).

```
(* ==> 8 : nat *)
```

```
(* -----
   Ejercicio 1.6.1. Definir la función
   factorial : nat -> nat
   tal que (factorial n) es el factorial de n.
   factorial 3 = 6.
   factorial 5 = mult 10 12
   ----- *)
```

```
Fixpoint factorial (n:nat) : nat :=
  match n with
  | 0    => 1
  | S n' => S n' * factorial n'
end.
```



*Ejemplo 1.6.20. Definir la función*

*menor\_o\_igual : nat -> nat -> bool*  
*tal que (menor\_o\_igual n m) se verifica si n es menor o igual que m.*  
 ----- \*)

```
Fixpoint menor_o_igual (n m : nat) : bool :=
match n with
| 0    => true
| S n' => match m with
      | 0    => false
      | S m' => menor_o_igual n' m'
    end
end.
```

(\* -----  
*Ejemplo 1.6.21. Demostrar las siguientes propiedades*  
 + menor\_o\_igual 2 2 = true.  
 + menor\_o\_igual 2 4 = true.  
 + menor\_o\_igual 4 2 = false.  
 ----- \*)

**Example** menor\_o\_igual1: menor\_o\_igual 2 2 = true.

**Proof.** simpl. reflexivity. Qed.

**Example** menor\_o\_igual2: menor\_o\_igual 2 4 = true.

**Proof.** simpl. reflexivity. Qed.

**Example** menor\_o\_igual3: menor\_o\_igual 4 2 = false.

**Proof.** simpl. reflexivity. Qed.

(\* -----  
*Ejercicio 1.6.2. Definir la función*  
*menor\_nat : nat -> nat -> bool*  
*tal que (menor\_nat n m) se verifica si n es menor que m.*  
  
*Demostrar las siguientes propiedades*  
 menor\_nat 2 2 = false.  
 menor\_nat 2 4 = true.  
 menor\_nat 4 2 = false.  
 ----- \*)

**Definition** menor\_nat (n m : nat) : bool :=  
negacion (iguales\_nat (m-n) 0).

**Example** menor\_nat1: (menor\_nat 2 2) = false.  
**Proof.** simpl. reflexivity. Qed.

**Example** menor\_nat2: (menor\_nat 2 4) = true.  
**Proof.** simpl. reflexivity. Qed.

**Example** menor\_nat3: (menor\_nat 4 2) = false.  
**Proof.** simpl. reflexivity. Qed.

```
(* =====
  § 2. Métodos elementales de demostración
  ===== *)
```

```
(* =====
  § 2.1. Demostraciones por simplificación
  ===== *)
```

```
(* -----
  Ejemplo 2.1.1. Demostrar que el 0 es el elemento neutro por la
  izquierda de la suma de los números naturales.
  ----- *)
```

(\* 1ª demostración \*)

**Theorem** suma\_0\_n : forall n : nat, 0 + n = n.

**Proof.**

```
  intros n.      (* n : nat
                  =====
                  0 + n = n *)
  simpl.         (* n = n *)
  reflexivity.
```

**Qed.**

(\* 2ª demostración \*)

**Theorem** suma\_0\_n' : forall n : nat, 0 + n = n.

**Proof.**

```
  intros n.      (* n : nat
```

```

=====
0 + n = n *)
reflexivity.
Qed.

(* -----
   Ejemplo 2.1.2. Demostrar que la suma de 1 y n es el siguiente de n.
   ----- *)

Theorem suma_1_l : forall n:nat, 1 + n = S n.
Proof.
  intros n.      (* n : nat
                  =====
                  1 + n = S n *)
  simpl.         (* S n = S n *)
  reflexivity.
Qed.

Theorem suma_1_l' : forall n:nat, 1 + n = S n.
Proof.
  intros n.
  reflexivity.
Qed.

(* -----
   Ejemplo 2.1.3. Demostrar que el producto de 0 por n es 0.
   ----- *)

Theorem producto_0_l : forall n:nat, 0 * n = 0.
Proof.
  intros n.      (* n : nat
                  =====
                  0 * n = 0 *)
  simpl.         (* 0 = 0 *)
  reflexivity.
Qed.

(* =====
   § 2.2. Demostraciones por reescritura
   ===== *)

```

```
(* -----
Ejemplo 2.2.1. Demostrar que si  $n = m$ , entonces  $n + n = m + m$ .
----- *)
```

**Theorem** suma\_iguales : forall n m:nat,  
 $n = m \rightarrow$   
 $n + n = m + m$ .

**Proof.**

```
intros n m. (* n : nat
              m : nat
              =====
              n = m -> n + n = m + m *)

intros H. (* n : nat
            m : nat
            H : n = m
            =====
            n + n = m + m *)

rewrite H. (* m + m = m + m *)
reflexivity.
```

**Qed.**

```
(* -----
Ejercicio 2.2.1. Demostrar que si  $n = m$  y  $m = o$ , entonces
 $n + m = m + o$ .
----- *)
```

**Theorem** suma\_iguales\_ejercicio : forall n m o : nat,  
 $n = m \rightarrow m = o \rightarrow n + m = m + o$ .

**Proof.**

```
intros n m o H1 H2. (* n : nat
                       m : nat
                       o : nat
                       H1 : n = m
                       H2 : m = o
                       =====
                       n + m = m + o *)

rewrite H1. (* m + m = m + o *)
rewrite H2. (* o + o = o + o *)
```



```

reflexivity.
Qed.

```

```

(* -----
   Ejemplo 2.2.2. Demostrar que  $(0 + n) * m = n * m$ .
   ----- *)

```

```

Theorem producto_0_mas : forall n m : nat,
   $(0 + n) * m = n * m$ .

```

```

Proof.

```

```

  intros n m.          (* n : nat
                        m : nat
                        =====
                         $(0 + n) * m = n * m$  *)
  rewrite suma_0_n.    (*  $n * m = n * m$  *)
  reflexivity.

```

```

Qed.

```

```

(* -----
   Ejercicio 2.2.2. Demostrar que si  $m = S n$ , entonces  $m * (1 + n) = m * m$ .
   ----- *)

```

```

Theorem producto_S_1 : forall n m : nat,
   $m = S n \rightarrow m * (1 + n) = m * m$ .

```

```

Proof.

```

```

  intros n m H. (* n : nat
                  m : nat
                  H :  $m = S n$ 
                  =====
                   $m * (1 + n) = m * m$  *)
  simpl.      (*  $m * S n = m * m$  *)
  rewrite H.   (*  $S n * S n = S n * S n$  *)
  reflexivity.

```

```

Qed.

```

```

(* =====
   § 2.3. Demostraciones por análisis de casos
   ===== *)

```

```

(* -----

```

*Ejemplo 2.3.1. Demostrar que  $n + 1$  es distinto de  $0$ .*

```

----- *)

(* 1º intento *)
Theorem siguiente_distinto_cero_primer_intento : forall n : nat,
  iguales_nat (n + 1) 0 = false.
Proof.
  intros n. (* n : nat
    =====
    iguales_nat (n + 1) 0 = false *)
  simpl. (* n : nat
    =====
    iguales_nat (n + 1) 0 = false *)

Abort.

(* 2º intento *)
Theorem siguiente_distinto_cero : forall n : nat,
  iguales_nat (n + 1) 0 = false.
Proof.
  intros n. (* n : nat
    =====
    iguales_nat (n + 1) 0 = false *)
  destruct n as [| n'].
  - (*
    =====
    iguales_nat (0 + 1) 0 = false *)
    reflexivity.
  - (* n' : nat
    =====
    iguales_nat (S n' + 1) 0 = false *)
    reflexivity.
Qed.

(* -----
  Ejemplo 2.3.2. Demostrar que la negacion es involutiva; es decir, la
  negacion de la negacion de b es b.
  ----- *)

```

**Theorem** negacion\_involutiva : **forall** b : **bool**,  
 negacion (negacion b) = b.

**Proof.**

```

intros b.      (*
                    =====
                    negacion (negacion b) = b *)

destruct b.
-              (*
                    =====
                    negacion (negacion true) = true *)

  reflexivity.
-              (*
                    =====
                    negacion (negacion false) = false *)

  reflexivity.

```

**Qed.**

```

(* -----
   Ejemplo 2.3.3. Demostrar que la conjuncion es conmutativa.
   ----- *)

```

(\* 1ª demostración \*)

**Theorem** conjuncion\_commutativa : **forall** b c,  
 conjuncion b c = conjuncion c b.

**Proof.**

```

intros b c.    (* b : bool
                  c : bool
                  =====
                  b && c = c && b *)

destruct b.
-              (* c : bool
                  =====
                  true && c = c && true *)

  destruct c.
+              (* =====
                  true && true = true && true *)

    reflexivity.
+              (*
                  =====
                  true && false = false && true *)

    reflexivity.
-              (* c : bool

```

```

=====
      false && c = c && false *)
destruct c.
+      (*
=====
      false && true = true && false *)
  reflexivity.
+      (*
=====
      false && false = false && false *)
  reflexivity.
Qed.

(* 2ª demostración *)
Theorem conjuncion_commutativa2 : forall b c,
  conjuncion b c = conjuncion c b.
Proof.
  intros b c.
  destruct b.
  { destruct c.
    { reflexivity. }
    { reflexivity. } }
  { destruct c.
    { reflexivity. }
    { reflexivity. } }
Qed.

(* -----
Ejemplo 2.3.4. Demostrar que
  conjuncion (conjuncion b c) d = conjuncion (conjuncion b d) c.
----- *)

Theorem conjuncion_intercambio : forall b c d,
  conjuncion (conjuncion b c) d = conjuncion (conjuncion b d) c.
Proof.
  intros b c d.
  destruct b.
  - destruct c.
    { destruct d.
      - reflexivity. (* (true && true) && true = (true && true) && true *)

```

```

- reflexivity. } (* (true && true) && false = (true && false) && true *)
{ destruct d.
- reflexivity. (* (true && false) && true = (true && true) && false *)
- reflexivity. } (* (true && false) && false = (true && false) && false *)
- destruct c.
{ destruct d.
- reflexivity. (* (false && true) && true = (false && true) && true *)
- reflexivity. } (* (false && true) && false = (false && false) && true *)
{ destruct d.
- reflexivity. (* (false && false) && true = (false && true) && false *)
- reflexivity. } (* (false && false) && false = (false && false) && false *)
Qed.

```

```

(* -----
   Ejemplo 2.3.5. Demostrar que  $n + 1$  es distinto de 0.
   ----- *)

```

**Theorem** siguiente\_distinto\_cero' : forall n : nat,  
iguales\_nat (n + 1) 0 = false.

**Proof.**

```

intros [|n].
- reflexivity. (* iguales_nat (0 + 1) 0 = false *)
- reflexivity. (* iguales_nat (S n + 1) 0 = false *)
Qed.

```

```

(* -----
   Ejemplo 2.3.6. Demostrar que la conjuncion es conmutativa.
   ----- *)

```

**Theorem** conjuncion\_commutativa'' : forall b c,  
conjuncion b c = conjuncion c b.

**Proof.**

```

intros [] [].
- reflexivity. (* true && true = true && true *)
- reflexivity. (* true && false = false && true *)
- reflexivity. (* false && true = true && false *)
- reflexivity. (* false && false = false && false *)
Qed.

```

```

(* -----

```

*Ejercicio 2.2.3. Demostrar que si  
conjuncion b c = true, entonces c = true.*

----- \*)

**Theorem** conjuncion\_true\_elim : **forall** b c : **bool**,  
conjuncion b c = **true** -> c = **true**.

**Proof.**

```

intros b c.      (* b : bool
                  c : bool
                  =====
                  b && c = true -> c = true *)

destruct c.
-
  (* b : bool
  =====
  b && true = true -> true = true *)

  reflexivity.
-
  (* b : bool
  =====
  b && false = true -> false = true *)

  destruct b.
+
  (*
  =====
  true && false = true -> false = true *)

  simpl.
  (*
  =====
  false = true -> false = true *)

  intros H.
  (* H : false = true
  =====
  false = true *)

  rewrite H.
  (* H : false = true
  =====
  true = true *)

  reflexivity.
+
  (*
  =====
  false && false = true -> false = true *)

  simpl.
  (*
  =====
  false = true -> false = true *)

  intros H.
  (* H : false = true

```

```

=====
      false = true *)
rewrite H.   (* H : false = true
=====
      true = true *)

reflexivity.

Qed.

(* -----
   Ejercicio 2.2.4. Demostrar que 0 es distinto de n + 1.
   ----- *)

Theorem cero_distinto_mas_uno: forall n : nat,
  iguales_nat 0 (n + 1) = false.
Proof.
  intros [| n'].
  - reflexivity. (* iguales_nat 0 (0 + 1) = false *)
  - reflexivity. (* iguales_nat 0 (S n' + 1) = false *)
Qed.

(* =====
   § 3. Ejercicios complementarios
   ===== *)

(* -----
   Ejercicio 3.1. Demostrar que
   forall (f : bool -> bool),
     (forall (x : bool), f x = x) -> forall (b : bool), f (f b) = b.
   ----- *)

Theorem aplica_dos_veces_la_identidad : forall (f : bool -> bool),
  (forall (x : bool), f x = x) -> forall (b : bool), f (f b) = b.
Proof.
  intros f H b. (* f : bool -> bool
                  H : forall x : bool, f x = x
                  b : bool
                  =====
                  f (f b) = b *)
  rewrite H.   (* f b = b *)
  rewrite H.   (* b = b *)

```

**reflexivity.**  
**Qed.**

```
(* -----
Ejercicio 3.2. Demostrar que
  forall (b c : bool),
    (conjuncion b c = disyuncion b c) -> b = c.
----- *)
```

**Theorem** conjuncion\_igual\_disyuncion: **forall** (b c : **bool**),  
 (conjuncion b c = disyuncion b c) -> b = c.

**Proof.**

```
intros [] c.
-
  (* c : bool
  =====
    true && c = true || c -> true = c *)
simpl.
  (* c : bool
  =====
    c = true -> true = c *)
intros H.
  (* c : bool
    H : c = true
    =====
    true = c *)
rewrite H.
  (* c : bool
    H : c = true
    =====
    true = true *)
reflexivity.
-
  (* c : bool
  =====
    false && c = false || c -> false = c *)
simpl.
  (* c : bool
  =====
    false = c -> false = c *)
intros H.
  (* c : bool
    H : false = c
    =====
    false = c *)
rewrite H.
  (* c : bool
    H : false = c
```



```

=====
c = c *)

reflexivity.
Qed.

(* -----
Ejercicio 3.3. En este ejercicio se considera la siguiente
representación de los números naturales
  Inductive nat2 : Type :=
    | C : nat2
    | D : nat2 -> nat2
    | SD : nat2 -> nat2.
donde C representa el cero, D el doble y SD el siguiente del doble.

  Definir la función
    nat2Anat : nat2 -> nat
  tal que (nat2Anat x) es el número natural representado por x.

  Demostrar que
    nat2Anat (SD (SD C))      = 3
    nat2Anat (D (SD (SD C))) = 6.
----- *)

Inductive nat2 : Type :=
| C : nat2
| D : nat2 -> nat2
| SD : nat2 -> nat2.

Fixpoint nat2Anat (x:nat2) : nat :=
  match x with
  | C    => 0
  | D n  => 2 * nat2Anat n
  | SD n => (2 * nat2Anat n) + 1
  end.

Example prop_nat2Anat1: (nat2Anat (SD (SD C))) = 3.
Proof. reflexivity. Qed.

Example prop_nat2Anat2: (nat2Anat (D (SD (SD C)))) = 6.
Proof. reflexivity. Qed.

```

```
(* =====  
  § Bibliografía  
  ===== *)  
  
(*  
  + "Functional programming in Coq" de Peirce et als. http://bit.ly/2zRCL6t  
  *)
```

## Tema 2

# Demostraciones por inducción sobre los números naturales en Coq

```
(* T2: Demostraciones por inducción sobre los números naturales en Coq *)
```

```
Require Export T1_PF_en_Coq.
```

```
(* El contenido de la teoría es  
  1. Demostraciones por inducción.  
  2. Demostraciones anidadas.  
  3. Demostraciones formales vs demostraciones informales.  
  4. Ejercicios complementarios *)
```

```
(* =====  
  § 1. Demostraciones por inducción  
  ===== *)
```

```
(* -----  
  Ejemplo 1.1. Demostrar que  
    forall n:nat, n = n + 0.  
  ----- *)
```

```
(* 1º intento: con métodos elementales *)
```

```
Theorem suma_n_0_a: forall n:nat, n = n + 0.
```

```
Proof.
```

```
  intros n. (* n : nat
```

```
            =====
```

```

      n = n + 0 *)
simpl.    (* n : nat
            =====
            n = n + 0 *)

```

Abort.

(\* 2º intento: con casos \*)

**Theorem** suma\_n\_0\_b : **forall** n:**nat**,  
 n = n + 0.

**Proof.**

```

intros n.          (* n : nat
                    =====
                    n = n + 0 *)

destruct n as [| n'].
-
  (*
  =====
  0 = 0 + 0 *)

  reflexivity.
-
  (* n' : nat
  =====
  S n' = S n' + 0 *)

  simpl.          (* n' : nat
                    =====
                    S n' = S (n' + 0) *)

```

Abort.

(\* 3ª intento: con inducción \*)

**Theorem** suma\_n\_0 : **forall** n:**nat**,  
 n = n + 0.

**Proof.**

```

intros n.          (* n : nat
                    =====
                    n = n + 0 *)

induction n as [| n' IHn'].
+
  (*
  =====
  0 = 0 + 0 *)

  reflexivity.
+
  (* n' : nat
     IHn' : n' = n' + 0

```

```

                                =====
                                S n' = S n' + 0 *)
simpl.                        (* S n' = S (n' + 0) *)
rewrite <- IHn'.              (* S n' = S n' *)
reflexivity.

Qed.

(* -----
   Ejemplo 1.2. Demostrar que
   forall n, n - n = 0.
   ----- *)

Theorem resta_n_n: forall n, n - n = 0.
Proof.
  intros n.                    (* n : nat
                                =====
                                n - n = 0 *)

  induction n as [| n' IHn'].
  +
    (*
      =====
      0 - 0 = 0 *)

    reflexivity.
  +
    (* n' : nat
       IHn' : n' - n' = 0
       =====
       S n' - S n' = 0 *)
    simpl.                    (* n' - n' = 0 *)
    rewrite -> IHn'.           (* 0 = 0 *)
    reflexivity.

Qed.

(* -----
   Ejercicio 1.1. Demostrar que
   forall n:nat, n * 0 = 0.
   ----- *)

```

**Theorem** multiplica\_n\_0: **forall** n:nat, n \* 0 = 0.

**Proof.**

```

intros n.                    (* n : nat
                                =====

```

```

                                n * 0 = 0 *)
induction n as [| n' IHn'].
+
                                (*
                                =====
                                0 * 0 = 0 *)

                                reflexivity.

+
                                (* n' : nat
                                IHn' : n' * 0 = 0
                                =====
                                S n' * 0 = 0 *)

                                simpl.
                                rewrite IHn'.
                                reflexivity.
Qed.

(* -----
   Ejercicio 1.2. Demostrar que,
   forall n m : nat, S (n + m) = n + (S m).
   ----- *)

Theorem suma_n_Sm: forall n m : nat, S (n + m) = n + (S m).
Proof.
  intros n m.
                                (* n, m : nat
                                =====
                                S (n + m) = n + S m *)

  induction n as [| n' IHn'].
+
                                (* m : nat
                                =====
                                S (0 + m) = 0 + S m *)

                                simpl.
                                (* m : nat
                                =====
                                S m = S m *)

                                reflexivity.

+
                                (* S (S n' + m) = S n' + S m *)
                                (* S (S (n' + m)) = S (n' + S m) *)
                                simpl.
                                rewrite IHn'.
                                (* S (n' + S m) = S (n' + S m) *)
                                reflexivity.
Qed.

(* -----
```

*Ejercicio 1.3. Demostrar que*  
 $\text{forall } n \ m : \text{nat}, n + m = m + n.$

----- \*)

**Theorem** suma\_conmutativa: **forall** n m : **nat**,  
 n + m = m + n.

**Proof.**

```

intros n m.                                (* n, m : nat
                                             =====
                                             n + m = m + n *)

induction n as [|n' IHn'].
+
  (* m : nat
   =====
   0 + m = m + 0 *)

  simpl.
  rewrite <- suma_n_0.
  reflexivity.
+
  (* n', m : nat
   IHn' : n' + m = m + n'
   =====
   S n' + m = m + S n' *)

  simpl.
  rewrite IHn'.
  rewrite <- suma_n_Sm.
  reflexivity.

```

**Qed.**

(\* -----  
*Ejercicio 1.4. Demostrar que*  
 $\text{forall } n \ m \ p : \text{nat}, n + (m + p) = (n + m) + p.$   
 ----- \*)

**Theorem** suma\_asociativa: **forall** n m p : **nat**, n + (m + p) = (n + m) + p.

**Proof.**

```

intros n m p.                                (* n, m, p : nat
                                             =====
                                             n + (m + p) = (n + m) + p *)

induction n as [|n' IHn'].
+
  (* m, p : nat
   =====

```





```

                                IHn' : doble n' = n' + n'
                                =====
                                doble (S n') = S n' + S n' *)
simpl.                        (* S (S (doble n')) = S (n' + S n') *)
rewrite IHn'.                 (* S (S (n' + n')) = S (n' + S n') *)
rewrite suma_n_Sm.           (* S (n' + S n') = S (n' + S n') *)
reflexivity.
Qed.

(* -----
   Ejercicio 1.6. Demostrar que
   forall n : nat, esPar (S n) = negacion (esPar n).
   ----- *)

Theorem esPar_S : forall n : nat,
  esPar (S n) = negacion (esPar n).
Proof.
  intros n.                    (* n : nat
                                =====
                                esPar (S n) = negacion (esPar n) *)

  induction n as [|n' IHn'].
  +
    simpl.                    (*
                                =====
                                esPar 1 = negacion (esPar 0) *)

    reflexivity.             (*
                                =====
                                false = false *)

  +
    (* n' : nat
       IHn' : esPar (S n') = negacion (esPar n')
       =====
       esPar (S (S n')) =
         negacion (esPar (S n')) *)
    rewrite IHn'.              (* esPar (S (S n')) =
                                negacion (negacion (esPar n')) *)
    rewrite negacion_involutiva. (* esPar (S (S n')) = esPar n' *)
    simpl.                    (* esPar n' = esPar n' *)
    reflexivity.
Qed.

```

```
(* =====
§ 2. Demostraciones anidadas
===== *)
```

```
(* -----
Ejemplo 2.1. Demostrar que
  forall n m : nat, (0 + n) * m = n * m.
----- *)
```

**Theorem** producto\_0\_suma': forall n m : nat, (0 + n) \* m = n \* m.

**Proof.**

```
  intros n m.          (* n, m : nat
                        =====
                        (0 + n) * m = n * m *)

  assert (H: 0 + n = n).
-
    (* n, m : nat
    =====
    0 + n = n *)

    reflexivity.

-
    (* n, m : nat
    H : 0 + n = n
    =====
    (0 + n) * m = n * m *)

    rewrite -> H.      (* n * m = n * m *)
    reflexivity.
```

**Qed.**

```
(* -----
Ejemplo 2.2. Demostrar que
  forall n m p q : nat, (n + m) + (p + q) = (m + n) + (p + q)
----- *)
```

(\* 1º intento sin assert\*)

**Theorem** suma\_reordenada\_1: forall n m p q : nat,  
(n + m) + (p + q) = (m + n) + (p + q).

**Proof.**

```
  intros n m p q.      (* n, m, p, q : nat
                        =====
                        (n + m) + (p + q) = (m + n) + (p + q) *)

  rewrite -> suma_conmutativa. (* n, m, p, q : nat
```

$$p + q + (n + m) = m + n + (p + q) \quad *)$$

Abort.

(\* 2º intento con assert \*)

**Theorem** suma\_reordenada: **forall** n m p q : **nat**,  
 $(n + m) + (p + q) = (m + n) + (p + q)$ .

**Proof.**

```

intros n m p q.
    (* n, m, p, q : nat
       =====
       (n + m) + (p + q) = (m + n) + (p + q) *)

assert (H: n + m = m + n).
-
    (* n, m, p, q : nat
       =====
       n + m = m + n *)
    rewrite -> suma_conmutativa. (* m + n = m + n *)
    reflexivity.
-
    (* n, m, p, q : nat
       H : n + m = m + n
       =====
       (n + m) + (p + q) = (m + n) + (p + q) *)
    rewrite -> H.
    reflexivity.

```

**Qed.**

```

(* =====
   § 3. Demostraciones formales vs demostraciones informales
   ===== *)

(* -----
   Ejercicio 3.1. Escribir la demostración informal (en lenguaje natural)
   correspondiente a la demostración formal de la asociatividad de la
   suma del ejercicio 1.4.
   ----- *)

```

(\* Demostración por inducción en n.

- Caso base: Se supone que n es 0 y hay que demostrar que  
 $0 + (m + p) = (0 + m) + p$ .  
 Esto es consecuencia inmediata de la definición de suma.

- Paso de inducción: Suponemos la hipótesis de inducción

$$n' + (m + p) = (n' + m) + p.$$

Hay que demostrar que

$$(S\ n') + (m + p) = ((S\ n') + m) + p.$$

que, por la definición de suma, se reduce a

$$S\ (n' + (m + p)) = S\ ((n' + m) + p)$$

que por la hipótesis de inducción se reduce a

$$S\ ((n' + m) + p) = S\ ((n' + m) + p)$$

que es una identidad. \*)

(\* -----  
Ejercicio 3.2. Escribir la demostración informal (en lenguaje natural)  
correspondiente a la demostración formal de la asociatividad de la  
suma del ejercicio 1.3.  
----- \*)

(\* Demostración por inducción en  $n$ .

- Caso base: Se supone que  $n$  es  $0$  y hay que demostrar que

$$0 + m = m + 0$$

que, por la definición de la suma, se reduce a

$$m = m + 0$$

que se verifica por el lema `suma_n_0`.

- Paso de inducción: Suponemos la hipótesis de inducción

$$n' + m = m + n'$$

Hay que demostrar que

$$S\ n' + m = m + S\ n'$$

que, por la definición de suma, se reduce a

$$S\ (n' + m) = m + S\ n'$$

que, por la hipótesis de inducción, se reduce a

$$S\ (m + n') = m + S\ n'$$

que, por el lema `suma_n_Sm`, se reduce a

$$S\ (m + n') = S\ (m + n')$$

que es una identidad. \*)

(\* -----  
Ejercicio 3.3. Demostrar que  
forall  $n:\text{nat}$ , `iguales_nat n n = true`.

```

----- *)

Theorem iguales_nat_refl: forall n : nat,
  iguales_nat n n = true.
Proof.
  intros n.                                (* n : nat
                                           =====
                                           iguales_nat n n = true *)

  induction n as [|n' IHn'].
  -                                         (*
                                           =====
                                           iguales_nat 0 0 = true *)

    reflexivity.
  -                                         (* n' : nat
                                           IHn' : iguales_nat n' n' = true
                                           =====
                                           iguales_nat (S n') (S n') = true *)

    simpl.                                (* iguales_nat n' n' = true *)
    rewrite <- IHn'.                       (* true = true *)
    reflexivity.

Qed.

(* -----
   Ejercicio 3.4. Escribir la demostración informal (en lenguaje natural)
   correspondiente la demostración del ejercicio anterior.
   ----- *)

(* Demostración por inducción en n.

- Caso base: Se supone que n es 0 y hay que demostrar que
  true = iguales_nat 0 0
  que se verifica por la definición de iguales_nat.

- Paso de inducción: Suponemos la hipótesis de inducción
  true = iguales_nat n' n'
  Hay que demostrar que
  true = iguales_nat (S n') (S n')
  que, por la definición de iguales_nat, se reduce a
  true = iguales_nat n' n
  que, por la hipótesis de inducción, se reduce a

```

```

    true = true
    que es una identidad. *)

(* =====
   § 4. Ejercicios complementarios
   ===== *)

(* -----
   Ejercicio 4.1. Demostrar, usando assert pero no induct,
   forall n m p : nat, n + (m + p) = m + (n + p).
   ----- *)

Theorem suma_permutada: forall n m p : nat,
  n + (m + p) = m + (n + p).
Proof.
  intros n m p. (* n, m, p : nat
                  =====
                  n + (m + p) = m + (n + p) *)
  rewrite suma_asociativa. (* n, m, p : nat
                             =====
                             (n + m) + p = m + (n + p) *)
  rewrite suma_asociativa. (* n, m, p : nat
                             =====
                             n + m + p = m + n + p *)
  assert (H : n + m = m + n).
  - (* n, m, p : nat
      =====
      n + m = m + n *)
    rewrite suma_conmutativa. (* m + n = m + n *)
    reflexivity.
  - (* n, m, p : nat
      H : n + m = m + n
      =====
      (n + m) + p = (m + n) + p *)
    rewrite H. (* (m + n) + p = (m + n) + p *)
    reflexivity.
Qed.

(* -----
   Ejercicio 4.2. Demostrar que la multiplicación es conmutativa.
   ----- *)

```

```

----- *)

Lemma producto_n_1 : forall n: nat,
  n * 1 = n.
Proof.
  intro n.
  (* n : nat
     =====
     n * 1 = n *)

  induction n as [|n' IHn'].
  -
  (*
     =====
     0 * 1 = 0 *)

    reflexivity.
  -
  (* n' : nat
     IHn' : n' * 1 = n'
     =====
     S n' * 1 = S n' *)
    simpl.
    rewrite IHn'.
    reflexivity.
Qed.

Theorem suma_n_1 : forall n : nat,
  n + 1 = S n.
Proof.
  intro n.
  (* n : nat
     =====
     n + 1 = S n *)

  induction n as [|n' HIn'].
  -
  (*
     =====
     0 + 1 = 1 *)

    reflexivity.
  -
  (* n' : nat
     HIn' : n' + 1 = S n'
     =====
     S n' + 1 = S (S n') *)
    simpl.
    rewrite HIn'.
    reflexivity.

```

**Qed.**

**Theorem** producto\_n\_Sm: **forall** n m : **nat**,

$$n * (m + 1) = n * m + n.$$

**Proof.**

```

intros n m.                                (* n, m : nat
                                             =====
                                             n * (m + 1) = n * m + n *)

induction n as [|n' IHn'].
-
  reflexivity.
-
  (* n', m : nat
     IHn' : n' * (m + 1) = n' * m + n'
     =====
     S n' * (m + 1) = S n' * m + S n' *)
  simpl.
  (* (m + 1) + n' * (m + 1) =
     (m + n' * m) + S n' *)
  rewrite IHn'.
  (* (m + 1) + (n' * m + n') =
     (m + n' * m) + S n' *)
  rewrite suma_permutada.
  (* n' * m + ((m + 1) + n') =
     (m + n' * m) + S n' *)
  rewrite <- suma_asociativa.
  (* n' * m + (m + (1 + n')) =
     (m + n' * m) + S n' *)
  rewrite <- suma_n_1.
  (* n' * m + (m + (n' + 1)) =
     (m + n' * m) + S n' *)
  rewrite suma_n_1.
  (* n' * m + (m + S n') = (m + n' * m) + S n' *)
  rewrite suma_permutada.
  (* m + (n' * m + S n') = (m + n' * m) + S n' *)
  rewrite suma_asociativa.
  (* m + (n' * m + S n') = (m + n' * m) + S n' *)
  reflexivity.

```

**Qed.**

**Theorem** producto\_conmutativa: **forall** m n : **nat**,

$$m * n = n * m.$$

**Proof.**

```

intros n m.                                (* n, m : nat
                                             =====
                                             n * m = m * n *)

induction n as [|n' HIn'].

```



```

-
      (* m : nat
      =====
      0 * m = m * 0 *)
rewrite multiplica_n_0. (* 0 * m = 0 *)
reflexivity.
-
      (* n', m : nat
      HIn' : n' * m = m * n'
      =====
      S n' * m = m * S n' *)
simpl.
rewrite HIn'.
rewrite <- suma_n_1. (* m + m * n' = m * (n' + 1) *)
rewrite producto_n_Sm. (* m + m * n' = m * n' + m *)
rewrite suma_conmutativa. (* m * n' + m = m * n' + m *)
reflexivity.
Qed.

(* -----
   Ejercicio 4.3. Demostrar que
   forall n : nat, true = menor_o_igual n n.
   ----- *)

Theorem menor_o_igual_refl: forall n : nat,
  true = menor_o_igual n n.
Proof.
  intro n.
      (* n : nat
      =====
      true = menor_o_igual n n *)
  induction n as [| n' HIn'].
  -
      (*
      =====
      true = menor_o_igual 0 0 *)
      reflexivity.
  -
      (* n' : nat
      HIn' : true = menor_o_igual n' n'
      =====
      true = menor_o_igual (S n') (S n') *)
      simpl.
      rewrite HIn'.
      (* true = menor_o_igual n' n' *)
      (* menor_o_igual n' n' = menor_o_igual n' n' *)

```

```

    reflexivity.
Qed.

```

```

(* -----
   Ejercicio 4.4. Demostrar que
     forall n : nat, iguales_nat 0 (S n) = false.
   ----- *)

```

```

Theorem cero_distinto_S: forall n : nat,
  iguales_nat 0 (S n) = false.

```

```

Proof.

```

```

  intros n.      (* n : nat
                  =====
                  iguales_nat 0 (S n) = false *)
  simpl.         (* false = false *)
  reflexivity.
Qed.

```

```

(* -----
   Ejercicio 4.5. Demostrar que
     forall b : bool, conjuncion b false = false.
   ----- *)

```

```

Theorem conjuncion_false_r : forall b : bool,
  conjuncion b false = false.

```

```

Proof.

```

```

  intros b.      (* b : bool
                  =====
                  b && false = false *)
  destruct b.
  -              (*
                  =====
                  true && false = false *)
    simpl.       (* false = false *)
    reflexivity.
  -              (*
                  =====
                  false && false = false *)
    simpl.       (* false = false *)
    reflexivity.

```

**Qed.**

```
(* -----
  Ejercicio 4.6. Demostrar que
    forall n m p : nat, menor_o_igual n m = true ->
      menor_o_igual (p + n) (p + m) = true.
  ----- *)
```

**Theorem** menor\_o\_igual\_suma: **forall** n m p : **nat**,  
 menor\_o\_igual n m = **true** -> menor\_o\_igual (p + n) (p + m) = **true**.

**Proof.**

```
  intros n m p H. (* n, m, p : nat
                    H : menor_o_igual n m = true
                    =====
                    menor_o_igual (p + n) (p + m) = true *)

  induction p as [|p' HIp'].
  - (* n, m : nat
      H : menor_o_igual n m = true
      =====
      menor_o_igual (0 + n) (0 + m) = true *)
    simpl.
    rewrite H.
    reflexivity.
  - (* n, m, p' : nat
      H : menor_o_igual n m = true
      HIp' : menor_o_igual (p' + n) (p' + m) = true
      =====
      menor_o_igual (S p' + n) (S p' + m) = true *)
    simpl.
    rewrite HIp'.
    reflexivity.
```

**Qed.**

```
(* -----
  Ejercicio 4.7. Demostrar que
    forall n : nat, iguales_nat (S n) 0 = false.
  ----- *)
```

**Theorem** S\_distinto\_0 : **forall** n:**nat**,  
 iguales\_nat (S n) 0 = **false**.

**Proof.**

```

intro n.      (* n : nat
                =====
                iguales_nat (S n) 0 = false *)
simpl.      (* false = false *)
reflexivity.
Qed.

```

```

(* -----
   Ejercicio 4.8. Demostrar que
   forall n:nat, 1 * n = n.
   ----- *)

```

**Theorem** producto\_1\_n: **forall** n:**nat**, 1 \* n = n.

**Proof.**

```

intro n.      (* n : nat
                =====
                1 * n = n *)
simpl.      (* n + 0 = n *)
rewrite suma_n_0. (* n + 0 = n + 0 *)
reflexivity.
Qed.

```

```

(* -----
   Ejercicio 4.9. Demostrar que
   forall b c : bool, disyuncion (conjuncion b c)
                       (disyuncion (negacion b)
                                   (negacion c))
                       = true.
   ----- *)

```

**Theorem** alternativas: **forall** b c : **bool**,

```

disyuncion
  (conjuncion b c)
  (disyuncion (negacion b)
              (negacion c))
= true.

```

**Proof.**

```

intros [] [].
- reflexivity. (* (true && true) || (negacion true || negacion true) = true *)

```

```
- reflexivity. (* (true && false) || (negacion true || negacion false) = true *)
- reflexivity. (* (false && true) || (negacion false || negacion true) = true *)
- reflexivity. (* (false && false) || (negacion false || negacion false)=true *)
```

**Qed.**

```
(* -----
Ejercicio 4.10. Demostrar que
  forall n m p : nat, (n + m) * p = (n * p) + (m * p).
----- *)
```

**Theorem** producto\_suma\_distributiva\_d: **forall** n m p : **nat**,  
 (n + m) \* p = (n \* p) + (m \* p).

**Proof.**

```
intros n m p. (* n, m, p : nat
=====
(n + m) * p = n * p + m * p *)

induction n as [|n' HIn'].
- (* m, p : nat
=====
(0 + m) * p = 0 * p + m * p *)

  reflexivity.
- (* n', m, p : nat
   HIn' : (n' + m) * p = n' * p + m * p
   =====
   (S n' + m) * p = S n' * p + m * p *)

  simpl. (* p + (n' + m) * p = (p + n' * p) + m * p *)
  rewrite HIn'. (* p + (n' * p + m * p) = (p + n') * p + m * p *)
  rewrite suma_asociativa. (* (p + n' * p) + m * p = (p + n' * p) + m * p *)
  reflexivity.
```

**Qed.**

```
(* -----
Ejercicio 4.11. Demostrar que
  forall n m p : nat, n * (m * p) = (n * m) * p.
----- *)
```

**Theorem** producto\_asociativa: **forall** n m p : **nat**,  
 n \* (m \* p) = (n \* m) \* p.

**Proof.**

```
intros n m p. (* n, m, p : nat
```

```

=====
n * (m * p) = (n * m) * p *)
induction n as [|n' HIn'].
-
  (* m, p : nat
  =====
  0 * (m * p) = (0 * m) * p *)
  simpl.
  reflexivity.
-
  (* n', m, p : nat
  HIn' : n' * (m * p) = (n' * m) * p
  =====
  S n' * (m * p) = (S n' * m) * p *)
  simpl.
  (* m * p + n' * (m * p) = (m + n' * m) * p *)
  rewrite HIn'.
  (* m * p + (n' * m) * p = (m + n' * m) * p *)
  rewrite producto_suma_distributiva_d.
  (* m * p + (n' * m) * p = m * p + (n' * m) * p *)
  reflexivity.
Qed.

```

```

(* -----
Ejercicio 11. La táctica replace permite especificar el subtérmino
que se desea reescribir y su sustituto:
  replace t with u
sustituye todas las copias de la expresión t en el objetivo por la
expresión u y añade la ecuación (t = u) como un nuevo subobjetivo.

El uso de la táctica replace es especialmente útil cuando la táctica
rewrite actúa sobre una parte del objetivo que no es la que se desea.

Demostrar, usando la táctica replace y sin usar
[assert (n + m = m + n)], que
  forall n m p : nat, n + (m + p) = m + (n + p).
----- *)

```

**Theorem** suma\_permutada' : **forall** n m p : **nat**,  
 n + (m + p) = m + (n + p).

**Proof.**

```

intros n m p.
(* n, m, p : nat
=====
n + (m + p) = m + (n + p) *)

```

```

rewrite suma_asociativa.      (* (n + m) + p = m + (n + p) *)
rewrite suma_asociativa.      (* (n + m) + p = (m + n) + p *)
replace (n + m) with (m + n).
-                               (* n, m, p : nat
                               =====
                               (m + n) + p = (m + n) + p *)

  reflexivity.
-                               (* n, m, p : nat
                               =====
                               m + n = n + m *)

  rewrite suma_conmutativa.    (* n + m = n + m *)
  reflexivity.
Qed.

(* =====
   § Bibliografía
   ===== *)

(*
+ "Demostraciones por inducción" de Peirce et als. http://bit.ly/2NRSWTF
*)

```





## Tema 3

# Datos estructurados en Coq

*(\* T3: Datos estructurados en Coq \*)*

**Require Export** T2\_Induccion.

*(\* El contenido de la teoría es*

- 1. Pares de números*
- 2. Listas de números*
  - 1. El tipo de la lista de números.*
  - 2. La función repite (repeat)*
  - 3. La función longitud (length)*
  - 4. La función conc (app)*
  - 5. Las funciones primero (hd) y resto (tl)*
  - 6. Ejercicios sobre listas de números*
  - 7. Multiconjuntos como listas*
- 3. Razonamiento sobre listas*
  - 1. Demostraciones por simplificación*
  - 2. Demostraciones por casos*
  - 3. Demostraciones por inducción*
  - 4. Ejercicios*
- 4. Opcionales*
- 5. Diccionarios (o funciones parciales)*
- 6. Bibliografía*

*\*)*

*(\* =====*  
*§ 1. Pares de números*  
*===== \*)*

```
(* -----
   Nota. Se iniciar el módulo ListaNat.
   ----- *)
```

**Module** ListaNat.

```
(* -----
   Ejemplo 1.1. Definir el tipo ProdNat para los pares de números
   naturales con el constructor
       par : nat -> nat -> ProdNat.
   ----- *)
```

**Inductive** ProdNat : **Type** :=  
 par : **nat** -> **nat** -> ProdNat.

```
(* -----
   Ejemplo 1.2. Calcular el tipo de la expresión (par 3 5)
   ----- *)
```

**Check** (par 3 5).

```
(* ==> par 3 5 : ProdNat *)
```

```
(* -----
   Ejemplo 1.3. Definir la función
       fst : ProdNat -> nat
   tal que (fst p) es la primera componente de p.
   ----- *)
```

**Definition** fst (p : ProdNat) : **nat** :=  
**match** p **with**  
 | par x y => x  
**end**.

```
(* -----
   Ejemplo 1.4. Evaluar la expresión
       fst (par 3 5)
   ----- *)
```

**Compute** (fst (par 3 5)).

```
(* ==> 3 : nat *)
```

```
(* -----
  Ejemplo 1.5. Definir la función
    snd : ProdNat -> nat
  tal que (snd p) es la segunda componente de p.
  ----- *)
```

```
Definition snd (p : ProdNat) : nat :=
  match p with
  | par x y => y
  end.
```

```
(* -----
  Ejemplo 1.6. Definir la notación (x,y) como una abreviatura de
  (par x y).
  ----- *)
```

```
Notation "( x , y )" := (par x y).
```

```
(* -----
  Ejemplo 1.7. Evaluar la expresión
    fst (3,5)
  ----- *)
```

```
Compute (fst (3,5)).
```

```
(* ==> 3 : nat *)
```

```
(* -----
  Ejemplo 1.8. Redefinir la función fst usando la abreviatura de pares.
  ----- *)
```

```
Definition fst' (p : ProdNat) : nat :=
  match p with
  | (x,y) => x
  end.
```

```
(* -----
  Ejemplo 1.9. Redefinir la función snd usando la abreviatura de pares.
  ----- *)
```

**Definition** snd' (p : ProdNat) : nat :=  
 match p with  
 | (x,y) => y  
 end.

(\* -----  
 Ejemplo 1.10. Definir la función  
 intercambia : ProdNat -> ProdNat  
 tal que (intercambia p) es el par obtenido intercambiando las  
 componentes de p.  
 ----- \*)

**Definition** intercambia (p : ProdNat) : ProdNat :=  
 match p with  
 | (x,y) => (y,x)  
 end.

(\* -----  
 Ejemplo 1.11. Demostrar que para todos los naturales  
 (n,m) = (fst (n,m), snd (n,m)).  
 ----- \*)

**Theorem** par\_componentes1 : forall (n m : nat),  
 (n,m) = (fst (n,m), snd (n,m)).

**Proof.**

reflexivity.

**Qed.**

(\* -----  
 Ejemplo 1.12. Demostrar que para todo par de naturales  
 p = (fst p, snd p).  
 ----- \*)

(\* 1º intento \*)

**Theorem** par\_componentes2 : forall (p : ProdNat),  
 p = (fst p, snd p).

**Proof.**

simpl. (\*

=====

forall p : ProdNat, p = (fst p, snd p) \*)

Abort.

*(\* 2º intento \*)*

**Theorem** par\_componentes : **forall** (p : ProdNat),  
p = (fst p, snd p).

**Proof.**

```

intros p.          (* p : ProdNat
                    =====
                    p = (fst p, snd p) *)

destruct p as [n m]. (* n, m : nat
                    =====
                    (n, m) = (fst (n, m), snd (n, m)) *)

simpl.           (* (n, m) = (n, m) *)
reflexivity.

```

**Qed.**

```

(* -----
   Ejercicio 1.1. Demostrar que para todo par de naturales p,
   (snd p, fst p) = intercambia p.
   ----- *)

```

**Theorem** ejercicio\_1\_1: **forall** p : ProdNat,  
(snd p, fst p) = intercambia p.

**Proof.**

```

intro p.          (* p : ProdNat
                    =====
                    (snd p, fst p) = intercambia p *)

destruct p as [n m]. (* n, m : nat
                    =====
                    (snd (n, m), fst (n, m)) = intercambia (n, m) *)

simpl.           (* (m, n) = (m, n) *)
reflexivity.

```

**Qed.**

```

(* -----
   Ejercicio 1.2. Demostrar que para todo par de naturales p,
   fst (intercambia p) = snd p.
   ----- *)

```

**Theorem** ejercicio\_1\_2: **forall** p : ProdNat,

`fst (intercambia p) = snd p.`

**Proof.**

```

intro p.                (* p : ProdNat
                           =====
                           fst (intercambia p) = snd p *)
destruct p as [n m]. (* n, m : nat
                           =====
                           fst (intercambia (n, m)) = snd (n, m) *)
simpl.                 (* m = m *)
reflexivity.

```

**Qed.**

```

(* =====
   § 2. Listas de números
   ===== *)

```

```

(* =====
   §§ 2.1. El tipo de la lista de números.
   ===== *)

```

```

(* -----
   Ejemplo 2.1.1. Definir el tipo ListaNat de la lista de los números
   naturales y cuyo constructores son
   + nil (la lista vacía) y
   + cons (tal que (cons x ys) es la lista obtenida añadiéndole x a ys.
   ----- *)

```

```

Inductive ListaNat : Type :=
| nil   : ListaNat
| cons  : nat -> ListaNat -> ListaNat.

```

```

(* -----
   Ejemplo 2.1.2. Definir la constante
   ejLista : ListaNat
   que es la lista cuyos elementos son 1, 2 y 3.
   ----- *)

```

**Definition** ejLista := cons 1 (cons 2 (cons 3 nil)).

```

(* -----

```

*Ejemplo 2.1.3. Definir la notación  $(x :: ys)$  como una abreviatura de  $(\text{cons } x \text{ } ys)$ .*

----- \*)

**Notation** " $x :: l$ " := (cons x l)  
(at level 60, **right** associativity).

(\* -----  
*Ejemplo 2.1.4. Definir la notación de las listas finitas escribiendo sus elementos entre corchetes y separados por puntos y comas.*  
----- \*)

**Notation** "[ ]" := nil.

**Notation** "[  $x ; .. ; y$  ]" := (cons x .. (cons y nil) ..).

(\* -----  
*Ejemplo 2.1.5. Definir la lista cuyos elementos son 1, 2 y 3 mediante distintas representaciones.*  
----- \*)

**Definition** ejLista1 := 1 :: (2 :: (3 :: nil)).

**Definition** ejLista2 := 1 :: 2 :: 3 :: nil.

**Definition** ejLista3 := [1;2;3].

(\* =====  
*§§ 2.2. La función repite (repeat)*  
===== \*)

(\* -----  
*Ejemplo 2.2.1. Definir la función*  
*repite : nat -> nat -> ListaNat*  
*tal que (repite n k) es la lista formada por k veces el número n. Por*  
*ejemplo,*  
*repite 5 3 = [5; 5; 5]*

*Nota: La función repite es equivalente a la predefinida repeat.*

----- \*)

**Fixpoint** repite (n k : **nat**) : ListaNat :=  
  **match** k **with**

```

| 0      => nil
| S k' => n :: (repite n k')
end.

```

Compute (repite 5 3).

```
(* ==> [5; 5; 5] : ListaNat*)
```

```
(* =====
  §§ 2.3. La función longitud (length)
  ===== *)
```

```
(* -----
  Ejemplo 2.3.1. Definir la función
    longitud : ListaNat -> nat
  tal que (longitud xs) es el número de elementos de xs. Por ejemplo,
    longitud [4;2;6] = 3

  Nota: La función longitud es equivalente a la predefinida length
  ----- *)
```

```

Fixpoint longitud (l:ListaNat) : nat :=
match l with
| nil      => 0
| h :: t => S (longitud t)
end.

```

Compute (longitud [4;2;6]).

```
(* ==> 3 : nat *)
```

```
(* =====
  §§ 2.4. La función conc (app)
  ===== *)
```

```
(* -----
  Ejemplo 2.4.1. Definir la función
    conc : ListaNat -> ListaNat -> ListaNat
  tal que (conc xs ys) es la concatenación de xs e ys. Por ejemplo,
    conc [1;3] [4;2;3;5] = [1; 3; 4; 2; 3; 5]

```

Nota: La función conc es equivalente a la predefinida app.



```

----- *)

Fixpoint conc (xs ys : ListaNat) : ListaNat :=
  match xs with
  | nil    => ys
  | x :: zs => x :: (conc zs ys)
  end.

Compute (conc [1;3] [4;2;3;5]).
(* ==> [1; 3; 4; 2; 3; 5] : ListaNat *)

(* -----
   Ejemplo 2.4.2. Definir la notación (xs ++ ys) como una abreviatura de
   (conc xs ys).
   ----- *)

Notation "x ++ y" := (conc x y)
                  (right associativity, at level 60).

(* -----
   Ejemplo 2.4.3. Demostrar que
   [1;2;3] ++ [4;5] = [1;2;3;4;5].
   nil ++ [4;5] = [4;5].
   [1;2;3] ++ nil = [1;2;3].
   ----- *)

Example test_conc1: [1;2;3] ++ [4;5] = [1;2;3;4;5].
Proof. reflexivity. Qed.

Example test_conc2: nil ++ [4;5] = [4;5].
Proof. reflexivity. Qed.

Example test_conc3: [1;2;3] ++ nil = [1;2;3].
Proof. reflexivity. Qed.

(* =====
   §§ 2.5. Las funciones primero (hd) y resto (tl)
   ===== *)

(* -----
```

*Ejemplo 2.5.1. Definir la función*

```

    primero : nat -> ListaNat -> ListaNat
    tal que (primero d xs) es el primer elemento de xs o d, si xs es la lista
    vacía. Por ejemplo,
        primero 7 [3;2;5] = 3
        primero 7 []      = 7

```

*Nota. La función primero es equivalente a la predefinida hd*

----- \*)

**Definition** primero (d : nat) (xs : ListaNat) : nat :=  
**match** xs **with**  
 | nil => d  
 | y :: ys => y  
**end.**

Compute (primero 7 [3;2;5]).

(\* ==> 3 : nat \*)

Compute (primero 7 []).

(\* ==> 7 : nat \*)

(\* -----  
*Ejemplo 2.5.2. Demostrar que*  
 primero 0 [1;2;3] = 1.  
 resto [1;2;3] = [2;3].  
 ----- \*)

**Example** prop\_primero1: primero 0 [1;2;3] = 1.

**Proof.** reflexivity. **Qed.**

**Example** prop\_primero2: primero 0 [] = 0.

**Proof.** reflexivity. **Qed.**

(\* -----  
*Ejemplo 2.5.3. Definir la función*  
 resto : ListaNat -> ListaNat  
 tal que (resto xs) es el resto de xs. Por ejemplo.  
 resto [3;2;5] = [2; 5]  
 resto [] = [ ]  
 ----- \*)

*Nota. La función resto es equivalente la predefinida tl.*

----- \*)

**Definition** resto (xs:ListaNat) : ListaNat :=  
**match** xs **with**  
 | nil => nil  
 | y :: ys => ys  
**end.**

Compute (resto [3;2;5]).  
 (\* ==> [2; 5] : ListaNat \*)  
 Compute (resto []).  
 (\* ==> [ ] : ListaNat \*)

(\* -----  
*Ejemplo 2.5.4. Demostrar que*  
*resto [1;2;3] = [2;3].*  
 ----- \*)

**Example** prop\_resto: resto [1;2;3] = [2;3].  
**Proof.** reflexivity. **Qed.**

(\* =====  
 §§ 2.6. Ejercicios sobre listas de números  
 ===== \*)

(\* -----  
*Ejercicio 2.6.1. Definir la función*  
*noCeros : ListaNat -> ListaNat*  
*tal que (noCeros xs) es la lista de los elementos de xs distintos de*  
*cero. Por ejemplo,*  
*noCeros [0;1;0;2;3;0;0] = [1;2;3].*  
 ----- \*)

**Fixpoint** noCeros (xs:ListaNat) : ListaNat :=  
**match** xs **with**  
 | nil => nil  
 | a::bs => **match** a **with**  
 | 0 => noCeros bs  
 | \_ => a :: noCeros bs

```

    end
end.

```

```

Compute (noCeros [0;1;0;2;3;0;0]).
(* ==> [1; 2; 3] : ListaNat *)

```

```

(* -----
   Ejercicio 2.6.2. Definir la función
   impares : ListaNat -> ListaNat
   tal que (impares xs) es la lista de los elementos impares de
   xs. Por ejemplo,
   impares [0;1;0;2;3;0;0] = [1;3].
   ----- *)

```

```

Fixpoint impares (xs:ListaNat) : ListaNat :=
  match xs with
  | nil    => nil
  | y::ys => if esImpar y
             then y :: impares ys
             else impares ys
  end.

```

```

Compute (impares [0;1;0;2;3;0;0]).
(* ==> [1; 3] : ListaNat *)

```

```

(* -----
   Ejercicio 2.6.3. Definir la función
   nImpares : ListaNat -> nat
   tal que (nImpares xs) es el número de elementos impares de xs. Por
   ejemplo,
   nImpares [1;0;3;1;4;5] = 4.
   nImpares [0;2;4]      = 0.
   nImpares nil          = 0.
   ----- *)

```

```

Definition nImpares (xs:ListaNat) : nat :=
  longitud (impares xs).

```

```

Example prop_nImpares1: nImpares [1;0;3;1;4;5] = 4.

```

```

Proof. reflexivity. Qed.

```

**Example** prop\_nImpares2: nImpares [0;2;4] = 0.

**Proof.** reflexivity. Qed.

**Example** prop\_nImpares3: nImpares nil = 0.

**Proof.** reflexivity. Qed.

```
(* -----
Ejercicio 2.6.4. Definir la función
  intercaladas : ListaNat -> ListaNat -> ListaNat
tal que (intercaladas xs ys) es la lista obtenida intercalando los
elementos de xs e ys. Por ejemplo,
  intercaladas [1;2;3] [4;5;6] = [1;4;2;5;3;6].
  intercaladas [1] [4;5;6]     = [1;4;5;6].
  intercaladas [1;2;3] [4]     = [1;4;2;3].
  intercaladas [] [20;30]      = [20;30].
----- *)
```

**Fixpoint** intercaladas (xs ys : ListaNat) : ListaNat :=

**match** xs **with**

  | nil     => ys

  | x::xs' => **match** ys **with**

    | nil     => xs

    | y::ys' => x::y::intercaladas xs' ys'

**end**

**end.**

**Example** prop\_intercaladas1: intercaladas [1;2;3] [4;5;6] = [1;4;2;5;3;6].

**Proof.** reflexivity. Qed.

**Example** prop\_intercaladas2: intercaladas [1] [4;5;6] = [1;4;5;6].

**Proof.** reflexivity. Qed.

**Example** prop\_intercaladas3: intercaladas [1;2;3] [4] = [1;4;2;3].

**Proof.** reflexivity. Qed.

**Example** prop\_intercaladas4: intercaladas [] [20;30] = [20;30].

**Proof.** reflexivity. Qed.

```
(* =====
```

## §§ 2.7. Multiconjuntos como listas

===== \*)

(\* -----

*Ejemplo 2.7.1. Un multiconjunto es una colección de elementos donde no importa el orden de los elementos, pero sí el número de ocurrencias de cada elemento.*

*Definir el tipo multiconjunto de los multiconjuntos de números naturales.*

----- \*)

**Definition** multiconjunto := ListaNat.

(\* -----

*Ejercicio 2.7.2. Definir la función*

*n0currencias : nat -> multiconjunto -> nat*  
*tal que (n0currencias x ys) es el número de veces que aparece el elemento x en el multiconjunto ys. Por ejemplo,*

*n0currencias 1 [1;2;3;1;4;1] = 3.*

*n0currencias 6 [1;2;3;1;4;1] = 0.*

----- \*)

**Fixpoint** n0currencias (x:nat) (ys:multiconjunto) : nat :=

**match** ys **with**

| nil => 0

| y::ys' => **if** iguales\_nat y x  
           **then** 1 + n0currencias x ys'  
           **else** n0currencias x ys'

**end.**

**Example** prop\_n0currencias1: n0currencias 1 [1;2;3;1;4;1] = 3.

**Proof.** reflexivity. **Qed.**

**Example** prop\_n0currencias2: n0currencias 6 [1;2;3;1;4;1] = 0.

**Proof.** reflexivity. **Qed.**

(\* -----

*Ejercicio 2.7.3. Definir la función*

*suma : multiconjunto -> multiconjunto -> multiconjunto*

tal que (suma xs ys) es la suma de los multiconjuntos xs e ys. Por ejemplo,

```
suma [1;2;3] [1;4;1] = [1; 2; 3; 1; 4; 1]
n0currencias 1 (suma [1;2;3] [1;4;1]) = 3.
```

----- \*)

**Definition** suma : multiconjunto -> multiconjunto -> multiconjunto :=  
conc.

**Example** prop\_sum: n0currencias 1 (suma [1;2;3] [1;4;1]) = 3.

**Proof.** reflexivity. Qed.

```
(* -----
Ejercicio 2.7.4. Definir la función
  agrega : nat -> multiconjunto -> multiconjunto
tal que (agrega x ys) es el multiconjunto obtenido añadiendo el
elemento x al multiconjunto ys. Por ejemplo,
  n0currencias 1 (agrega 1 [1;4;1]) = 3.
  n0currencias 5 (agrega 1 [1;4;1]) = 0.
----- *)
```

**Definition** agrega (x:nat) (ys:multiconjunto) : multiconjunto :=  
x :: ys.

**Example** prop\_agrega1: n0currencias 1 (agrega 1 [1;4;1]) = 3.

**Proof.** reflexivity. Qed.

**Example** prop\_agrega2: n0currencias 5 (agrega 1 [1;4;1]) = 0.

**Proof.** reflexivity. Qed.

```
(* -----
Ejercicio 2.7.5. Definir la función
  pertenece : nat -> multiconjunto -> bool
tal que (pertenece x ys) se verifica si x pertenece al multiconjunto
ys. Por ejemplo,
  pertenece 1 [1;4;1] = true.
  pertenece 2 [1;4;1] = false.
----- *)
```

**Definition** pertenece (x:nat) (ys:multiconjunto) : bool :=

```
negacion (iguales_nat 0 (n0currencias x ys)).
```

**Example** prop\_pertenece1: pertenece 1 [1;4;1] = true.

**Proof.** reflexivity. Qed.

**Example** prop\_pertenece2: pertenece 2 [1;4;1] = false.

**Proof.** reflexivity. Qed.

```
(* -----
Ejercicio 2.7.6. Definir la función
  eliminaUna : nat -> multiconjunto -> multiconjunto
tal que (eliminaUna x ys) es el multiconjunto obtenido eliminando una
ocurrencia de x en el multiconjunto ys. Por ejemplo,
  n0currencias 5 (eliminaUna 5 [2;1;5;4;1]) = 0.
  n0currencias 4 (eliminaUna 5 [2;1;4;5;1;4]) = 2.
  n0currencias 5 (eliminaUna 5 [2;1;5;4;5;1;4]) = 1.
----- *)
```

```
Fixpoint eliminaUna (x:nat) (ys:multiconjunto) : multiconjunto :=
match ys with
| nil      => nil
| y :: ys' => if iguales_nat y x
               then ys'
               else y :: eliminaUna x ys'
end.
```

**Example** prop\_eliminaUna1: n0currencias 5 (eliminaUna 5 [2;1;5;4;1]) = 0.

**Proof.** reflexivity. Qed.

**Example** prop\_eliminaUna2: n0currencias 5 (eliminaUna 5 [2;1;4;1]) = 0.

**Proof.** reflexivity. Qed.

**Example** prop\_eliminaUna3: n0currencias 4 (eliminaUna 5 [2;1;4;5;1;4]) = 2.

**Proof.** reflexivity. Qed.

**Example** prop\_eliminaUna4: n0currencias 5 (eliminaUna 5 [2;1;5;4;5;1;4]) = 1.

**Proof.** reflexivity. Qed.

```
(* -----
Ejercicio 2.7.7. Definir la función
```



```

    eliminaTodas : nat -> multiconjunto -> multiconjunto
    tal que (eliminaTodas x ys) es el multiconjunto obtenido eliminando
    todas las ocurrencias de x en el multiconjunto ys. Por ejemplo,
    n0currencias 5 (eliminaTodas 5 [2;1;5;4;1])      = 0.
    n0currencias 5 (eliminaTodas 5 [2;1;4;1])        = 0.
    n0currencias 4 (eliminaTodas 5 [2;1;4;5;1;4])    = 2.
    n0currencias 5 (eliminaTodas 5 [2;1;5;4;5;1;4;5;1;4]) = 0.
    ----- *)

```

```

Fixpoint eliminaTodas (x:nat) (ys:multiconjunto) : multiconjunto :=
  match ys with
  | nil      => nil
  | y :: ys' => if iguales_nat y x
                then eliminaTodas x ys'
                else y :: eliminaTodas x ys'
  end.

```

**Example** prop\_eliminaTodas1: n0currencias 5 (eliminaTodas 5 [2;1;5;4;1]) = 0.

**Proof.** **reflexivity.** **Qed.**

**Example** prop\_eliminaTodas2: n0currencias 5 (eliminaTodas 5 [2;1;4;1]) = 0.

**Proof.** **reflexivity.** **Qed.**

**Example** prop\_eliminaTodas3: n0currencias 4 (eliminaTodas 5 [2;1;4;5;1;4]) = 2.

**Proof.** **reflexivity.** **Qed.**

**Example** prop\_eliminaTodas4: n0currencias 5 (eliminaTodas 5 [1;5;4;5;4;5;1]) = 0.

**Proof.** **reflexivity.** **Qed.**

```

(* -----
   Ejercicio 2.7.8. Definir la función
   submulticonjunto : multiconjunto -> multiconjunto -> bool
   tal que (submulticonjunto xs ys) se verifica si xs es un
   submulticonjunto de ys. Por ejemplo,
   submulticonjunto [1;2] [2;1;4;1] = true.
   submulticonjunto [1;2;2] [2;1;4;1] = false.
   ----- *)

```

```

Fixpoint submulticonjunto (xs:multiconjunto) (ys:multiconjunto) : bool :=
  match xs with

```

```

| nil      => true
| x::xs'   => pertenece x ys && submulticonjunto xs' (eliminaUna x ys)
end.

```

**Example** prop\_submulticonjunto1: submulticonjunto [1;2] [2;1;4;1] = true.

**Proof.** reflexivity. Qed.

**Example** prop\_submulticonjunto2: submulticonjunto [1;2;2] [2;1;4;1] = false.

**Proof.** reflexivity. Qed.

```

(* -----
   Ejercicio 2.7.9. Escribir una propiedad sobre multiconjuntos con las
   funciones n0currencias y agrega y demostrarla.
   ----- *)

```

**Theorem** n0currencias\_conc: forall xs ys : multiconjunto, forall n:nat,  
n0currencias n (conc xs ys) = n0currencias n xs + n0currencias n ys.

**Proof.**

```

intros xs ys n.
(* xs, ys : multiconjunto
   n : nat
   =====
   n0currencias n (xs ++ ys) =
   n0currencias n xs + n0currencias n ys *)

induction xs as [|x xs' HI].
-
(* ys : multiconjunto
   n : nat
   =====
   n0currencias n ([ ] ++ ys) =
   n0currencias n [ ] + n0currencias n ys *)

simpl.
reflexivity.
-
(* x : nat
   xs' : ListaNat
   ys : multiconjunto
   n : nat
   HI : n0currencias n (xs' ++ ys) =
        n0currencias n xs' + n0currencias n ys
   =====
   n0currencias n ((x :: xs') ++ ys) =
   n0currencias n (x :: xs') +

```

```

                                n0currencias n ys *)
simpl.                        (* (if iguales_nat x n
                                then S (n0currencias n (xs' ++ ys))
                                else n0currencias n (xs' ++ ys)) =
                                (if iguales_nat x n
                                then S (n0currencias n xs')
                                else n0currencias n xs') +
                                n0currencias n ys *)

destruct (iguales_nat x n).
+                               (* S (n0currencias n (xs' ++ ys)) =
                                S (n0currencias n xs') +
                                n0currencias n ys *)

simpl.                        (* S (n0currencias n (xs' ++ ys)) =
                                S (n0currencias n xs' +
                                n0currencias n ys) *)

rewrite HI.                    (* S (n0currencias n xs' + n0currencias n ys) =
                                S (n0currencias n xs' + n0currencias n ys) *)

reflexivity.
+                               (* n0currencias n (xs' ++ ys) =
                                n0currencias n xs' + n0currencias n ys *)

rewrite HI.                    (* n0currencias n xs' + n0currencias n ys =
                                n0currencias n xs' + n0currencias n ys *)

reflexivity.

Qed.

(* =====
  § 3. Razonamiento sobre listas
  ===== *)

(* =====
  §§ 3.1. Demostraciones por simplificación
  ===== *)

(* -----
  Ejemplo 3.1.1. Demostrar que, para toda lista de naturales xs,
    [] ++ xs = xs
  ----- *)

Theorem nil_conc : forall xs:ListNat,
  [] ++ xs = xs.

```

**Proof.**

**reflexivity.**

**Qed.**

```
(* =====
  §§ 3.2. Demostraciones por casos
  ===== *)
```

```
(* -----
  Ejemplo 3.2.1. Demostrar que, para toda lista de naturales xs,
    pred (longitud xs) = longitud (resto xs)
  ----- *)
```

**Theorem** resto\_longitud\_pred : **forall** xs:ListaNat,  
pred (longitud xs) = longitud (resto xs).

**Proof.**

```
intros xs.                (* xs : ListaNat
                           =====
                           Nat.pred (longitud xs) = longitud (resto xs) *)

destruct xs as [|x xs'].
-
  (*
    =====
    Nat.pred (longitud []) = longitud (resto []) *)

  reflexivity.
-
  (* x : nat
     xs' : ListaNat
     =====
     Nat.pred (longitud (x :: xs')) =
       longitud (resto (x :: xs')) *)

  reflexivity.
```

**Qed.**

```
(* =====
  §§ 3.3. Demostraciones por inducción
  ===== *)
```

```
(* -----
  Ejemplo 3.3.1. Demostrar que la concatenación de listas de naturales
  es asociativa; es decir,
    (xs ++ ys) ++ zs = xs ++ (ys ++ zs).
```

```

----- *)

Theorem conc_asociativa: forall xs ys zs : ListaNat,
  (xs ++ ys) ++ zs = xs ++ (ys ++ zs).
Proof.
  intros xs ys zs.
  (* xs, ys, zs : ListaNat
     =====
     (xs ++ ys) ++ zs = xs ++ (ys ++ zs) *)

  induction xs as [|x xs' HI].
  -
    (* ys, zs : ListaNat
       =====
       ([ ] ++ ys) ++ zs = [ ] ++ (ys ++ zs) *)

    reflexivity.
  -
    (* x : nat
       xs', ys, zs : ListaNat
       HI : (xs' ++ ys) ++ zs = xs' ++ (ys ++ zs)
       =====
       ((x :: xs') ++ ys) ++ zs =
        (x :: xs') ++ (ys ++ zs) *)

    simpl.
    (* (x :: (xs' ++ ys)) ++ zs =
       x :: (xs' ++ (ys ++ zs)) *)

    rewrite -> HI.
    (* x :: (xs' ++ (ys ++ zs)) =
       x :: (xs' ++ (ys ++ zs)) *)

    reflexivity.
Qed.

(* -----
   Ejemplo 3.3.2. Definir la función
   inversa : ListaNat -> ListaNat
   tal que (inversa xs) es la inversa de xs. Por ejemplo,
   inversa [1;2;3] = [3;2;1].
   inversa nil = nil.

   Nota. La función inversa es equivalente a la predefinida rev.
   ----- *)

Fixpoint inversa (xs:ListaNat) : ListaNat :=
  match xs with
  | nil => nil
  | x::xs' => inversa xs' ++ [x]

```

**end.**

**Example** prop\_inversa1: inversa [1;2;3] = [3;2;1].

**Proof.** **reflexivity.** **Qed.**

**Example** prop\_inversa2: inversa nil = nil.

**Proof.** **reflexivity.** **Qed.**

```
(* -----
   Ejemplo 3.3.3. Demostrar que
       longitud (inversa xs) = longitud xs
   ----- *)

(* 1º intento *)
Theorem longitud_inversa1: forall xs:ListaNat,
  longitud (inversa xs) = longitud xs.
Proof.
  intros xs.
  induction xs as [|x xs' HI].
  -
    (* =====
       longitud (inversa [ ]) = longitud [ ] *)

    reflexivity.

  -
    (* x : nat
       xs' : ListaNat
       HI : longitud (inversa xs') = longitud xs'
       =====
       longitud (inversa (x :: xs')) =
         longitud (x :: xs') *)

    simpl.
    (* longitud (inversa xs' ++ [x]) =
       S (longitud xs') *)

    rewrite <- HI.
    (* longitud (inversa xs' ++ [x]) =
       S (longitud (inversa xs')) *)

Abort.

(* Nota: Para simplificar la última expresión se necesita el siguiente lema. *)

Lemma longitud_conc : forall xs ys : ListaNat,
  longitud (xs ++ ys) = longitud xs + longitud ys.
Proof.
```

```

intros xs ys.
(* xs, ys : ListaNat
=====
longitud (xs ++ ys) =
longitud xs + longitud ys *)

induction xs as [| x xs' HI].
-
(* ys : ListaNat
=====
longitud ([ ] ++ ys) =
longitud [ ] + longitud ys *)

reflexivity.
-
(* x : nat
xs', ys : ListaNat
HI : longitud (xs' ++ ys) =
longitud xs' + longitud ys
=====
longitud ((x :: xs') ++ ys) =
longitud (x :: xs') + longitud ys *)

simpl.
(* S (longitud (xs' ++ ys)) =
S (longitud xs' + longitud ys) *)

rewrite -> HI.
(* S (longitud xs' + longitud ys) =
S (longitud xs' + longitud ys) *)

reflexivity.
Qed.

(* 2º intento *)
Theorem longitud_inversa : forall xs:ListaNat,
longitud (inversa xs) = longitud xs.
Proof.
intros xs.
(* xs : ListaNat
=====
longitud (inversa xs) = longitud xs *)

induction xs as [| x xs' HI].
-
(*
=====
longitud (inversa [ ]) = longitud [ ] *)

reflexivity.
-
(* x : nat
xs' : ListaNat
HI : longitud (inversa xs') = longitud xs'
=====

```

```

                                longitud (inversa (x :: xs')) =
                                longitud (x :: xs') *)
simpl.                        (* longitud (inversa xs' ++ [x]) =
                                S (longitud xs') *)
rewrite longitud_conc.        (* longitud (inversa xs') + longitud [x] =
                                S (longitud xs') *)
rewrite HI.                  (* longitud xs' + longitud [x] =
                                S (longitud xs') *)
simpl.                      (* longitud xs' + 1 = S (longitud xs') *)
rewrite suma_conmutativa.    (* 1 + longitud xs' = S (longitud xs') *)
reflexivity.
Qed.

(* =====
   §§ 3.4. Ejercicios
   ===== *)

(* -----
   Ejercicio 3.4.1. Demostrar que la lista vacía es el elemento neutro
   por la derecha de la concatenación de listas.
   ----- *)

Theorem conc_nil: forall xs:ListaNat,
  xs ++ [] = xs.
Proof.
  intros xs.                      (* xs : ListaNat
                                   =====
                                   xs ++ [ ] = xs *)

  induction xs as [| x xs' HI].
  -
    (* =====
       [ ] ++ [ ] = [ ] *)

    reflexivity.
  -
    (* x : nat
       xs' : ListaNat
       HI : xs' ++ [ ] = xs'
       =====
       (x :: xs') ++ [ ] = x :: xs' *)

    simpl.                      (* x :: (xs' ++ [ ]) = x :: xs' *)
    rewrite HI.                  (* x :: xs' = x :: xs' *)

```



**reflexivity.**  
**Qed.**

```
(* -----
   Ejercicio 3.4.2. Demostrar que inversa es un endomorfismo en
   (ListaNat,++); es decir,
       inversa (xs ++ ys) = inversa ys ++ inversa xs.
   ----- *)
```

**Theorem** inversa\_conc: **forall** xs ys : ListaNat,  
 inversa (xs ++ ys) = inversa ys ++ inversa xs.

**Proof.**

```
intros xs ys. (* xs, ys : ListaNat
                =====
                inversa (xs ++ ys) =
                inversa ys ++ inversa xs *)

induction xs as [|x xs' HI].
- (* ys : ListaNat
   =====
   inversa ([ ] ++ ys) =
   inversa ys ++ inversa [ ] *)
simpl. (* inversa ys = inversa ys ++ [ ] *)
rewrite conc_nil. (* inversa ys = inversa ys *)
reflexivity.

- (* x : nat
   xs', ys : ListaNat
   HI : inversa (xs' ++ ys) =
       inversa ys ++ inversa xs'
   =====
   inversa ((x :: xs') ++ ys) =
   inversa ys ++ inversa (x :: xs') *)
simpl. (* inversa (xs' ++ ys) ++ [x] =
          inversa ys ++ (inversa xs' ++ [x]) *)
rewrite HI. (* (inversa ys ++ inversa xs') ++ [x] =
              inversa ys ++ (inversa xs' ++ [x]) *)
rewrite conc_asociativa. (* inversa ys ++ (inversa xs' ++ [x]) =
                           inversa ys ++ (inversa xs' ++ [x]) *)
reflexivity.
Qed.
```

```
(* -----
  Ejercicio 3.4.3. Demostrar que inversa es involutiva; es decir,
    inversa (inversa xs) = xs.
  ----- *)
```

**Theorem** inversa\_involutiva: **forall** xs:ListaNat,  
inversa (inversa xs) = xs.

**Proof.**

**induction** xs **as** [|x xs' HI].

```
- (*
    =====
    inversa (inversa [ ]) = [ ] *)

  reflexivity.

- (* x : nat
    xs' : ListaNat
    HI : inversa (inversa xs') = xs'
    =====
    inversa (inversa (x :: xs')) = x :: xs' *)
  simpl. (* inversa (inversa xs' ++ [x]) = x :: xs' *)
  rewrite inversa_conc. (* inversa [x] ++ inversa (inversa xs') =
    x :: xs' *)
  simpl. (* x :: inversa (inversa xs') = x :: xs' *)
  rewrite HI. (* x :: xs' = x :: xs' *)
  reflexivity.
```

**Qed.**

```
(* -----
  Ejercicio 3.4.4. Demostrar que
    xs ++ (ys ++ (zs ++ vs)) = ((xs ++ ys) ++ zs) ++ vs.
  ----- *)
```

**Theorem** conc\_asociativa4 : **forall** xs ys zs vs : ListaNat,  
xs ++ (ys ++ (zs ++ vs)) = ((xs ++ ys) ++ zs) ++ vs.

**Proof.**

```
intros xs ys zs vs. (* xs, ys, zs, vs : ListaNat
    =====
    xs ++ (ys ++ (zs ++ vs)) =
    ((xs ++ ys) ++ zs) ++ vs *)
rewrite conc_asociativa. (* xs ++ (ys ++ (zs ++ vs)) =
    (xs ++ ys) ++ (zs ++ vs) *)
```

```

rewrite conc_asociativa. (* xs ++ (ys ++ (zs ++ vs)) =
                           xs ++ (ys ++ (zs ++ vs)) *)

reflexivity.
Qed.

(* -----
   Ejercicio 3.4.5. Demostrar que al concatenar dos listas no aparecen ni
   desaparecen ceros.
   ----- *)

Lemma noCeros_conc : forall xs ys : ListaNat,
  noCeros (xs ++ ys) = (noCeros xs) ++ (noCeros ys).
Proof.
  intros xs ys. (* xs, ys : ListaNat
                  =====
                  noCeros (xs ++ ys) =
                  noCeros xs ++ noCeros ys *)

  induction xs as [|x xs' HI].
  - (* ys : ListaNat
      =====
      noCeros ([]) ++ ys =
      noCeros [] ++ noCeros ys *)

    reflexivity.
  - (* x : nat
      xs', ys : ListaNat
      HI : noCeros (xs' ++ ys) =
          noCeros xs' ++ noCeros ys
      =====
      noCeros ((x :: xs') ++ ys) =
      noCeros (x :: xs') ++ noCeros ys *)

    destruct x.
    + (* noCeros ((0 :: xs') ++ ys) =
        noCeros (0 :: xs') ++ noCeros ys *)

      simpl.
      (* noCeros (xs' ++ ys) =
         noCeros xs' ++ noCeros ys *)

      rewrite HI.
      (* noCeros xs' ++ noCeros ys =
         noCeros xs' ++ noCeros ys *)

      reflexivity.
    + (* noCeros ((S x :: xs') ++ ys) =
        noCeros (S x :: xs') ++ noCeros ys *)

```

```

simpl.                                (* S x :: noCeros (xs' ++ ys) =
                                         (S x :: noCeros xs') ++ noCeros ys *)
rewrite HI.                            (* S x :: (noCeros xs' ++ noCeros ys) =
                                         (S x :: noCeros xs') ++ noCeros ys *)
reflexivity.
Qed.

(* -----
Ejercicio 3.4.6. Definir la función
    iguales_lista : ListaNat -> ListaNat -> bool
tal que (iguales_lista xs ys) se verifica si las listas xs e ys son
iguales. Por ejemplo,
    iguales_lista nil nil           = true.
    iguales_lista [1;2;3] [1;2;3] = true.
    iguales_lista [1;2;3] [1;2;4] = false.
----- *)

Fixpoint iguales_lista (xs ys : ListaNat) : bool :=
  match xs, ys with
  | nil,    nil      => true
  | x::xs', y::ys' => iguales_nat x y && iguales_lista xs' ys'
  | _, _          => false
end.

Example prop_iguales_lista1: (iguales_lista nil nil = true).
Proof. reflexivity. Qed.

Example prop_iguales_lista2: iguales_lista [1;2;3] [1;2;3] = true.
Proof. reflexivity. Qed.

Example prop_iguales_lista3: iguales_lista [1;2;3] [1;2;4] = false.
Proof. reflexivity. Qed.

(* -----
Ejercicio 3.4.7. Demostrar que la igualdad de listas cumple la
propiedad reflexiva.
----- *)

Theorem iguales_lista_refl : forall xs:ListaNat,
  iguales_lista xs xs = true.

```

**Proof.**

```

induction xs as [|x xs' HI].
-
    (*
      =====
      iguales_lista [ ] [ ] = true *)

    reflexivity.
-
    (* x : nat
       xs' : ListaNat
       HI : iguales_lista xs' xs' = true
       =====
       iguales_lista (x :: xs') (x :: xs') = true *)

    simpl.
    (* iguales_nat x x &&
       iguales_lista xs' xs' = true *)

    rewrite HI.
    rewrite iguales_nat_refl.
    reflexivity.

```

**Qed.**

```

(* -----
   Ejercicio 3.4.8. Demostrar que al incluir un elemento en un
   multiconjunto, ese elemento aparece al menos una vez en el
   resultado.
   ----- *)

```

**Theorem** n0currencias\_agrega: **forall** (x:nat) (xs:multiconjunto),  
 menor\_o\_igual 1 (n0currencias x (agrega x xs)) = **true**.

**Proof.**

```

intros x xs.
    (* x : nat
       xs : multiconjunto
       =====
       menor_o_igual 1 (n0currencias x (agrega x xs)) =
       true *)

    simpl.
    (* match
       (if iguales_nat x x then S (n0currencias x xs)
        else n0currencias x xs)
       with
       | 0 => false
       | S _ => true
       end =
       true *)

```

```

rewrite iguales_nat_refl. (* true = true *)
reflexivity.
Qed.

```

```

(* -----
   Ejercicio 3.4.9. Demostrar que cada número natural es menor o igual
   que su siguiente.
   ----- *)

```

**Theorem** menor\_o\_igual\_n\_Sn: **forall** n:**nat**,  
 menor\_o\_igual n (S n) = **true**.

**Proof.**

```

intros n. (* n : nat
           =====
           menor_o_igual n (S n) = true *)

induction n as [|n' HI].
- (*
   =====
   menor_o_igual 0 1 = true *)

  reflexivity.
- (* n' : nat
   HI : menor_o_igual n' (S n') = true
   =====
   menor_o_igual (S n') (S (S n')) = true *)

  simpl. (* menor_o_igual n' (S n') = true *)
  rewrite HI. (* true = true *)
  reflexivity.

```

**Qed.**

```

(* -----
   Ejercicio 3.4.10. Demostrar que al borrar una ocurrencia de 0 de un
   multiconjunto el número de ocurrencias de 0 en el resultado es menor
   o igual que en el original.
   ----- *)

```

**Theorem** remove\_decreases\_n0currencias: **forall** (xs : multiconjunto),  
 menor\_o\_igual (n0currencias 0 (eliminaUna 0 xs)) (n0currencias 0 xs) = **true**.

**Proof.**

```

induction xs as [|x xs' HI].
- (*

```

```

=====
menor_o_igual (n0currencias 0 (eliminaUna 0
                                (n0currencias 0 []))
              = true *)

reflexivity.
-
(* x : nat
   xs' : ListaNat
   HI: menor_o_igual (n0currencias 0 (eliminaUna 0
                                           (n0currencias 0 xs'))
                     = true
   =====
   menor_o_igual (n0currencias 0 (eliminaUna 0
                                           (n0currencias 0 (x :: xs'))
                     = true *)

destruct x.
+
(* menor_o_igual (n0currencias 0 (eliminaUna 0
                                      (n0currencias 0 (0 :: xs'))
                    = true *)

simpl.
(* menor_o_igual (n0currencias 0 xs')
   (S (n0currencias 0 xs'))
   = true *)

rewrite menor_o_igual_n_Sn. (* true = true *)
reflexivity.
+
(* menor_o_igual (n0currencias 0
                  (eliminaUna 0 (S x :: xs')))
   (n0currencias 0 (S x :: xs'))
   = true *)

simpl.
(* menor_o_igual (n0currencias 0 (eliminaUna 0
                                      (n0currencias 0 xs'))
                    = true *)

rewrite HI.
reflexivity.

Qed.

(* -----
   Ejercicio 3.4.11. Escribir un teorema con las funciones n0currencias
   y suma de los multiconjuntos.
   ----- *)

```

**Theorem** n0currencias\_suma:

```

forall x : nat, forall xs ys : multiconjunto,
  n0currencias x (suma xs ys) = n0currencias x xs + n0currencias x ys.
Proof.
intros x xs ys.
induction xs as [|x' xs' HI].
-
  reflexivity.
-
  simpl.
destruct (iguales_nat x' x).
+
  rewrite HI.
  reflexivity.
+
  rewrite HI.

```

```

(* x : nat
   xs, ys : multiconjunto
   =====
   n0currencias x (suma xs ys) =
   n0currencias x xs + n0currencias x ys *)

(* x : nat
   ys : multiconjunto
   =====
   n0currencias x (suma [ ] ys) =
   n0currencias x [ ] + n0currencias x ys *)

(* x, x' : nat
   xs' : ListaNat
   ys : multiconjunto
   HI : n0currencias x (suma xs' ys) =
        n0currencias x xs' + n0currencias x ys
   =====
   n0currencias x (suma (x' :: xs') ys) =
   n0currencias x (x' :: xs') + n0currencias x ys *)

(* (if iguales_nat x' x
    then S (n0currencias x (suma xs' ys))
    else n0currencias x (suma xs' ys))
   =
   (if iguales_nat x' x
    then S (n0currencias x xs')
    else n0currencias x xs') + n0currencias x ys *)

(* S (n0currencias x (suma xs' ys)) =
   S (n0currencias x xs') + n0currencias x ys *)

(* S (n0currencias x xs' + n0currencias x ys) =
   S (n0currencias x xs') + n0currencias x ys *)

(* n0currencias x (suma xs' ys) =
   n0currencias x xs' + n0currencias x ys *)

(* n0currencias x xs' + n0currencias x ys =
   n0currencias x xs' + n0currencias x ys *)

```



**reflexivity.**

**Qed.**

```
(* -----
Ejercicio 3.4.12. Demostrar que la función inversa es inyectiva; es
decir,
  forall (xs ys : ListaNat), inversa xs = inversa ys -> xs = ys.
----- *)
```

**Theorem** inversa\_inyectiva: **forall** (xs ys : ListaNat),  
inversa xs = inversa ys -> xs = ys.

**Proof.**

```
intros xs ys H.                                (* xs, ys : ListaNat
                                                H : inversa xs = inversa ys
                                                =====
                                                xs = ys *)
rewrite <- inversa_involutiva. (* xs = inversa (inversa ys) *)
rewrite <- H.                  (* xs = inversa (inversa xs) *)
rewrite inversa_involutiva.    (* xs = xs *)
reflexivity.
```

**Qed.**

```
(* =====
§ 4. Opcionales
===== *)
```

```
(* -----
Ejemplo 4.1. Definir el tipo OpcionalNat con los constructores
  Some : nat -> OpcionalNat
  None : OpcionalNat.
----- *)
```

**Inductive** OpcionalNat : **Type** :=

```
| Some : nat -> OpcionalNat
| None : OpcionalNat.
```

```
(* -----
Ejemplo 4.2. Definir la función
  nthOpcional : ListaNat -> nat -> OpcionalNat
tal que (nthOpcional xs n) es el n-ésimo elemento de la lista xs o None
```

*si la lista tiene menos de  $n$  elementos. Por ejemplo,*  
 *$\text{nthOpcional } [4;5;6;7] \ 0 = \text{Some } 4.$*   
 *$\text{nthOpcional } [4;5;6;7] \ 3 = \text{Some } 7.$*   
 *$\text{nthOpcional } [4;5;6;7] \ 9 = \text{None}.$*

----- \*)

```
Fixpoint nthOpcional (xs:ListaNat) (n:nat) : OpcionalNat :=
match xs with
| nil      => None
| x :: xs' => match iguales_nat n 0 with
| true  => Some x
| false => nthOpcional xs' (pred n)
end
end.
```

**Example** prop\_nthOpcional1 : nthOpcional [4;5;6;7] 0 = Some 4.

**Proof.** reflexivity. **Qed**.

**Example** prop\_nthOpcional2 : nthOpcional [4;5;6;7] 3 = Some 7.

**Proof.** reflexivity. **Qed**.

**Example** prop\_nthOpcional3 : nthOpcional [4;5;6;7] 9 = None.

**Proof.** reflexivity. **Qed**.

(\* Introduciendo condicionales nos queda: \*)

```
Fixpoint nthOpcional' (xs:ListaNat) (n:nat) : OpcionalNat :=
match xs with
| nil      => None
| x :: xs' => if iguales_nat x 0
|         => then Some x
|         => else nthOpcional' xs' (pred n)
end.
```

(\* -----

*Ejemplo 4.3. Definir la función*

*eliminaOpcionalNat -> OpcionalNat -> nat*

*tal que (option\_elim d o) es el valor de o, si o tiene valor o es d en caso contrario. Por ejemplo,*

*eliminaOpcionalNat 3 (Some 7) = 7*

*eliminaOpcionalNat 3 None = 3*

```

----- *)

Definition eliminaOpcionalNat (d : nat) (o : OpcionalNat) : nat :=
  match o with
  | Some n' => n'
  | None    => d
  end.

Compute (eliminaOpcionalNat 3 (Some 7)).
(* ==> 7 : nat *)
Compute (eliminaOpcionalNat 3 None).
(* ==> 3 : nat *)

(* -----
   Ejercicio 4.1. Definir la función
   primeroOpcional : ListaNat -> OpcionalNat
   tal que (primeroOpcional xs) es el primer elemento de xs, si xs es no
   vacía; o es None, en caso contrario. Por ejemplo,
   primeroOpcional []      = None.
   primeroOpcional [1]     = Some 1.
   primeroOpcional [5;6]   = Some 5.
   ----- *)

Definition primeroOpcional (xs : ListaNat) : OpcionalNat :=
  match xs with
  | nil      => None
  | x::xs'   => Some x
  end.

Example prop_primeroOpcional1 : primeroOpcional [] = None.
Proof. reflexivity. Qed.

Example prop_primeroOpcional2 : primeroOpcional [1] = Some 1.
Proof. reflexivity. Qed.

Example prop_primeroOpcional3 : primeroOpcional [5;6] = Some 5.
Proof. reflexivity. Qed.

(* -----
   Ejercicio 4.2. Demostrar que

```

```

    primero d xs = eliminaOpcionalNat d (primeroOpcional xs).
----- *)

Theorem primero_primeroOpcional: forall (xs:ListaNat) (d:nat),
  primero d xs = eliminaOpcionalNat d (primeroOpcional xs).
Proof.
  intros xs d.          (* xs : ListaNat
                          d : nat
                          =====
                          primero d xs = eliminaOpcionalNat d (primeroOpcional xs) *)

  destruct xs as [|x xs'].
  - (* d : nat
      =====
      primero d [] = eliminaOpcionalNat d (primeroOpcional []) *)

    reflexivity.

  - (* x : nat
      xs' : ListaNat
      d : nat
      =====
      primero d (x :: xs') =
      eliminaOpcionalNat d (primeroOpcional (x :: xs')) *)

    simpl.
    reflexivity.
Qed.

(* -----
   Nota. Finalizar el módulo ListaNat.
   ----- *)

End ListaNat.

(* =====
   § 5. Diccionarios (o funciones parciales)
   ===== *)

(* -----
   Ejemplo 5.1. Definir el tipo id (por identificador) con el
   constructor
       Id : nat -> id.
   ----- *)

```

```
Inductive id : Type :=
| Id : nat -> id.
```

```
(* -----
Ejemplo 5.2. Definir la función
    iguales_id : id -> id -> bool
tal que (iguales_id x1 x2) se verifica si tienen la misma clave. Por
ejemplo,
    iguales_id (Id 3) (Id 3) = true : bool
    iguales_id (Id 3) (Id 4) = false : bool
----- *)
```

```
Definition iguales_id (x1 x2 : id) :=
match x1, x2 with
| Id n1, Id n2 => iguales_nat n1 n2
end.
```

```
Compute (iguales_id (Id 3) (Id 3)).
```

```
(* ==> true : bool *)
```

```
Compute (iguales_id (Id 3) (Id 4)).
```

```
(* ==> false : bool *)
```

```
(* -----
Ejercicio 5.1. Demostrar que iguales_id es reflexiva.
----- *)
```

```
Theorem iguales_id_refl : forall x:id, iguales_id x x = true.
```

```
Proof.
```

```
  intro x.                                (* x : id
                                           =====
                                           iguales_id x x = true *)
  destruct x.                             (* iguales_id (Id n) (Id n) = true *)
  simpl.                                  (* iguales_nat n n = true *)
  rewrite iguales_nat_refl. (* true = true *)
  reflexivity.
```

```
Qed.
```

```
(* -----
Ejemplo 5.3. Iniciar el módulo Diccionario que importa a ListaNat.
```

```

----- *)

Module Diccionario.
Export ListaNat.

(* -----
   Ejemplo 5.4. Definir el tipo diccionario con los constructores
   vacio      : diccionario
   registro : id -> nat -> diccionario -> diccionario.
   ----- *)

Inductive diccionario : Type :=
| vacio      : diccionario
| registro : id -> nat -> diccionario -> diccionario.

(* -----
   Ejemplo 5.5. Definir los diccionarios cuyos elementos son
   + []
   + [(3,6)]
   + [(2,4), (3,6)]
   ----- *)

Definition diccionario1 := vacio.
Definition diccionario2 := registro (Id 3) 6 diccionario1.
Definition diccionario3 := registro (Id 2) 4 diccionario2.

(* -----
   Ejemplo 5.6. Definir la función
   valor : id -> diccionario -> OpcionalNat
   tal que (valor i d) es el valor de la entrada de d con clave i, o
   None si d no tiene ninguna entrada con clave i. Por ejemplo,
   valor (Id 2) diccionario3 = Some 4
   valor (Id 2) diccionario2 = None
   ----- *)

Fixpoint valor (x : id) (d : diccionario) : OpcionalNat :=
match d with
| vacio      => None
| registro y v d' => if iguales_id x y
then Some v

```

```

        else valor x d'
    end.

```

```

Compute (valor (Id 2) diccionario3).
(* = Some 4 : OpcionalNat *)
Compute (valor (Id 2) diccionario2).
(* = None : OpcionalNat*)

```

```

(* -----
   Ejemplo 5.7. Definir la función
       actualiza : diccionario -> id -> nat -> diccionario
   tal que (actualiza d x v) es el diccionario obtenido a partir del d
   + si d tiene un elemento con clave x, le cambia su valor a v
   + en caso contrario, le añade el elemento v con clave x
   ----- *)

```

```

Definition actualiza (d : diccionario)
              (x : id) (v : nat)
              : diccionario :=
    registro x v d.

```

```

(* -----
   Ejercicio 5.2. Demostrar que
       forall (d : diccionario) (x : id) (v: nat),
       valor x (actualiza d x v) = Some v.
   ----- *)

```

```

Theorem valor_actualiza: forall (d : diccionario) (x : id) (v: nat),
    valor x (actualiza d x v) = Some v.

```

**Proof.**

```

    intros d x v.
    (* d : diccionario
       x : id
       v : nat
       =====
       valor x (actualiza d x v) = Some v *)
    destruct x.
    simpl.
    (* valor (Id n) (actualiza d (Id n) v) = Some v *)
    (* (if iguales_nat n n then Some v
        else valor (Id n) d)
        = Some v *)
    rewrite iguales_nat_refl. (* Some v = Some v *)

```

**reflexivity.**

**Qed.**

```
(* -----
Ejercicio 5.3. Demostrar que
  forall (d : diccionario) (x y : id) (o: nat),
    iguales_id x y = false -> valor x (actualiza d y o) = valor x d.
----- *)
```

**Theorem** actualiza\_neq :

```
forall (d : diccionario) (x y : id) (o: nat),
  iguales_id x y = false -> valor x (actualiza d y o) = valor x d.
```

**Proof.**

```
intros d x y o p. (* d : diccionario
                    x, y : id
                    o : nat
                    p : iguales_id x y = false
                    =====
                    valor x (actualiza d y o) = valor x d *)
simpl. (* (if iguales_id x y then Some o
          else valor x d)
        = valor x d *)
rewrite p. (* valor x d = valor x d *)
```

**reflexivity.**

**Qed.**

```
(* -----
Ejemplo 5.8. Finalizar el módulo Diccionario
----- *)
```

**End** Diccionario.

```
(* =====
§ Bibliografía
===== *)
```

```
(*
+ "Working with structured data" de Peirce et als.
  http://bit.ly/2LQABsv
*)
```