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)

Índice general

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 conjuncion 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 (nanb 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 :=
  | O : 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
    | O    => O
    | 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 O)))).
   ------------------------------------------------------------------ *)

Check (S (S (S (S O)))).
(* ===> 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
    | O        => O
    | S O      => O
    | 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
  | O        => true
  | S O      => 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
    | O    => 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
    | O    => O
    | S n' => suma m (producto n' m)
  end.

Example producto1: (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
  | (O   , _)    => O
  | (S _ , O)    => 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
    | O   => S O
    | S m => mult x (potencia x m)
  end.

Compute (potencia 2 3).
(* ===> 8 : nat *)

(* ---------------------------------------------------------------------
   Ejercicio 1.6.1. Definir la función
      factorial : nat -> nat1
   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
  | O    => 1
  | S n' =>  S n' * factorial n'
  end.
```

```
Example prop_factorial1: factorial 3 = 6.
Proof. simpl. reflexivity.  Qed.

Example prop_factorial2: factorial 5 = mult 10 12.
Proof. simpl. reflexivity.  Qed.

(* ---------------------------------------------------------------------
   Ejemplo 1.6.18. Definir los operadores +, - y * como abreviaturas de
   las funciones plus, rminus y mult.
   ------------------------------------------------------------------ *)

Notation "x + y" := (plus x y)
                       (at level 50, left associativity)
                       : nat_scope.
Notation "x - y" := (minus x y)
                       (at level 50, left associativity)
                       : nat_scope.
Notation "x * y" := (mult x y)
                       (at level 40, left associativity)
                       : nat_scope.

(* ---------------------------------------------------------------------
   Ejemplo 1.6.19. Definir la función
      iguales_nat : nat -> nat -> bool
   tal que (iguales_nat n m) se verifica si n y me son iguales.
   ------------------------------------------------------------------ *)

Fixpoint iguales_nat (n m : nat) : bool :=
  match n with
  | O => match m with
         | O    => true
         | S m' => false
         end
  | S n' => match m with
            | O    => false
            | S m' => iguales_nat n' m'
            end
  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
  | O    => true
  | S n' => match m with
            | O    => 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 ->
  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 -> m = o -> 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 -> 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    => O
  | 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.                   (* n' * 0 = 0 *)
    rewrite IHn'.            (* 0 = 0 *)
    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 *)
    simpl.                   (* S (S (n' + m)) = S (n' + S m) *)
    rewrite IHn'.            (* S (n' + S m) = S (n' + S m) *)
    reflexivity.
Qed.

(* ---------------------------------------------------------------------
```

```
   Ejercicio 1.3. Demostrar que
      forall n m : 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.                   (* m = m + 0 *)
    rewrite <- suma_n_0.     (* m = m *)
    reflexivity.
  +                          (* n', m : nat
                                IHn' : n' + m = m + n'
                                ============================
                                S n' + m = m + S n' *)
    simpl.                   (* S (n' + m) = m + S n' *)
    rewrite IHn'.            (* S (m + n') = m + S n' *)
    rewrite <- suma_n_Sm.    (* S (m + n') = S (m + n') *)
    reflexivity.
Qed.

(* ----------------------------------------------------------------------
   Ejercicio 1.4. Demostrar que
      forall n m p : 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
                                ============================
```

```
                                 0 + (m + p) = (0 + m) + p *)
    reflexivity.
  +                           (* n', m, p : nat
                                 IHn' : n' + (m + p) = n' + m + p
                                 ============================
                                 S n' + (m + p) = (S n' + m) + p *)
    simpl.                    (* S (n' + (m + p)) = S ((n' + m) + p) *)
    rewrite IHn'.             (* S ((n' + m) + p) = S ((n' + m) + p) *)
    reflexivity.
Qed.

(* ---------------------------------------------------------------------
   Ejercicio 1.5. Se considera la siguiente función que dobla su argumento.
      Fixpoint doble (n:nat) :=
        match n with
        | 0    => 0
        | S n' => S (S (doble n'))
        end.

   Demostrar que
      forall n, doble n = n + n.
   ------------------------------------------------------------------ *)

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

Lemma doble_suma : forall n, doble n = n + n .
Proof.
  intros n.                   (* n : nat
                                 ============================
                                 doble n = n + n *)
  induction n as [|n' IHn'].
  +                           (*
                                 ============================
                                 doble 0 = 0 + 0 *)
    reflexivity.
  +                           (* n' : 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'].
  +                             (*
                                   ============================
                                   esPar 1 = negacion (esPar 0) *)
    simpl.                      (*
                                   ============================
                                   false = false *)
    reflexivity.
  +                             (* 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) *)
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.               (* m + n + (p + q) = m + n + (p + q) *)
    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 indución: 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 indución: 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: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 indución: 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.                   (* S (n' * 1) = S n' *)
    rewrite IHn'.            (* S n' = S n' *)
    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.                   (* S (n' + 1) = S (S n') *)
    rewrite HIn'.            (* S (S n') = S (S n') *)
    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'].
  -                            (* m : nat
                                  ============================
                                  0 * (m + 1) = 0 * m + 0 *)
    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.                       (* m + n' * m = m * S n' *)
    rewrite HIn'.                (* m + m * n' = m * S n' *)
    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.                       (* true = menor_o_igual n' n' *)
    rewrite HIn'.                (* 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.                    (* menor_o_igual n m = true *)
    rewrite H.                (* true = true *)
    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.                    (* menor_o_igual (p' + n) (p' + m) = true *)
    rewrite HIp'.             (* true = true *)
    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.          (* 0 = 0 *)
    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 subojetivo.

   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 abreviaura 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
   (cons x 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
   sistintas represerntaciones.
   ------------------------------------------------------------------ *)

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 quivalente a la predefinida repeat.
   ------------------------------------------------------------------ *)

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

```
  | O        => 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    => O
  | 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 abreviaura 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
      nOcurrencias : nat -> multiconjunto -> nat
   tal que (nOcurrencias x ys) es el número de veces que aparece el
   elemento x en el multiconjunto ys. Por ejemplo,
      nOcurrencias 1 [1;2;3;1;4;1] = 3.
      nOcurrencias 6 [1;2;3;1;4;1] = 0.
   ------------------------------------------------------------------ *)

Fixpoint nOcurrencias (x:nat) (ys:multiconjunto) : nat :=
  match ys with
  | nil    => 0
  | y::ys' => if iguales_nat y x
              then 1 + nOcurrencias x ys'
              else nOcurrencias x ys'
  end.

Example prop_nOcurrencias1: nOcurrencias 1 [1;2;3;1;4;1] = 3.
Proof. reflexivity. Qed.

Example prop_nOcurrencias2: nOcurrencias 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]
      nOcurrencias 1 (suma [1;2;3] [1;4;1]) = 3.
   ------------------------------------------------------------------ *)

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

Example prop_sum: nOcurrencias 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,
      nOcurrencias 1 (agrega 1 [1;4;1]) = 3.
      nOcurrencias 5 (agrega 1 [1;4;1]) = 0.
   ------------------------------------------------------------------ *)

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

Example prop_agrega1: nOcurrencias 1 (agrega 1 [1;4;1]) = 3.
Proof. reflexivity. Qed.

Example prop_agrega2: nOcurrencias 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 verfica 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 (nOcurrencias 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,
      nOcurrencias 5 (eliminaUna 5 [2;1;5;4;1])     = 0.
      nOcurrencias 4 (eliminaUna 5 [2;1;4;5;1;4])   = 2.
      nOcurrencias 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: nOcurrencias 5 (eliminaUna 5 [2;1;5;4;1]) = 0.
Proof. reflexivity. Qed.

Example prop_eliminaUna2: nOcurrencias 5 (eliminaUna 5 [2;1;4;1]) = 0.
Proof. reflexivity. Qed.

Example prop_eliminaUna3: nOcurrencias 4 (eliminaUna 5 [2;1;4;5;1;4]) = 2.
Proof. reflexivity. Qed.

Example prop_eliminaUna4: nOcurrencias 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,
      nOcurrencias 5 (eliminaTodas 5 [2;1;5;4;1])           = 0.
      nOcurrencias 5 (eliminaTodas 5 [2;1;4;1])             = 0.
      nOcurrencias 4 (eliminaTodas 5 [2;1;4;5;1;4])         = 2.
      nOcurrencias 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: nOcurrencias 5 (eliminaTodas 5 [2;1;5;4;1]) = 0.
Proof. reflexivity. Qed.

Example prop_eliminaTodas2: nOcurrencias 5 (eliminaTodas 5 [2;1;4;1]) = 0.
Proof. reflexivity. Qed.

Example prop_eliminaTodas3: nOcurrencias 4 (eliminaTodas 5 [2;1;4;5;1;4]) = 2.
Proof. reflexivity. Qed.

Example prop_eliminaTodas4: nOcurrencias 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 nOcurrencias y agrega y demostrarla.
   ------------------------------------------------------------------- *)

Theorem nOcurrencias_conc: forall xs ys : multiconjunto, forall n:nat,
  nOcurrencias n (conc xs ys) = nOcurrencias n xs + nOcurrencias n ys.
Proof.
  intros xs ys n.               (* xs, ys : multiconjunto
                                   n : nat
                                   ============================
                                   nOcurrencias n (xs ++ ys) =
                                    nOcurrencias n xs + nOcurrencias n ys *)
  induction xs as [|x xs' HI].
  -                             (* ys : multiconjunto
                                   n : nat
                                   ============================
                                   nOcurrencias n ([ ] ++ ys) =
                                    nOcurrencias n [ ] + nOcurrencias n ys *)
    simpl.                      (* nOcurrencias n ys = nOcurrencias n ys *)
    reflexivity.
  -                             (* x : nat
                                   xs' : ListaNat
                                   ys : multiconjunto
                                   n : nat
                                   HI : nOcurrencias n (xs' ++ ys) =
                                        nOcurrencias n xs' + nOcurrencias n ys
                                   ============================
                                   nOcurrencias n ((x :: xs') ++ ys) =
                                    nOcurrencias n (x :: xs') +
```

```
                                        nOcurrencias n ys *)
    simpl.                         (* (if iguales_nat x n
                                        then S (nOcurrencias n (xs' ++ ys))
                                        else nOcurrencias n (xs' ++ ys)) =
                                       (if iguales_nat x n
                                        then S (nOcurrencias n xs')
                                        else nOcurrencias n xs') +
                                       nOcurrencias n ys  *)
    destruct (iguales_nat x n).
    +                              (* S (nOcurrencias n (xs' ++ ys)) =
                                      S (nOcurrencias n xs') +
                                      nOcurrencias n ys *)
      simpl.                       (* S (nOcurrencias n (xs' ++ ys)) =
                                      S (nOcurrencias n xs' +
                                         nOcurrencias n ys) *)
      rewrite HI.                  (* S (nOcurrencias n xs' + nOcurrencias n ys) =
                                      S (nOcurrencias n xs' + nOcurrencias n ys) *)
      reflexivity.
    +                              (* nOcurrencias n (xs' ++ ys) =
                                      nOcurrencias n xs' + nOcurrencias n ys *)
      rewrite HI.                  (* nOcurrencias n xs' + nOcurrencias n ys =
                                      nOcurrencias n xs' + nOcurrencias 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:ListaNat,
  [] ++ 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.                  (* iguales_nat x x && true = true *)
    rewrite iguales_nat_refl.    (* true && true = true *)
    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 nOcurrencias_agrega: forall (x:nat) (xs:multiconjunto),
  menor_o_igual 1 (nOcurrencias x (agrega x xs)) = true.
Proof.
  intros x xs.                (* x : nat
                                 xs : multiconjunto
                                 ============================
                                 menor_o_igual 1 (nOcurrencias x (agrega x xs)) =
                                 true *)
  simpl.                      (* match
                                  (if iguales_nat x x then S (nOcurrencias x xs)
                                                      else nOcurrencias 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_nOcurrencias: forall (xs : multiconjunto),
  menor_o_igual (nOcurrencias 0 (eliminaUna 0 xs)) (nOcurrencias 0 xs) = true.
Proof.
  induction xs as [|x xs' HI].
  -                               (*
```

```
                                 ============================
                                 menor_o_igual (nOcurrencias 0 (eliminaUna 0
                                               (nOcurrencias 0 [])
                                 = true *)
    reflexivity.
  -                           (* x : nat
                                 xs' : ListaNat
                                 HI: menor_o_igual (nOcurrencias 0 (eliminaUn
                                                   (nOcurrencias 0 xs')
                                     = true
                                 ============================
                                 menor_o_igual (nOcurrencias 0 (eliminaUna 0
                                                (nOcurrencias 0 (x :: xs'))
                                 = true *)
    destruct x.
    +                         (* menor_o_igual (nOcurrencias 0 (eliminaUna 0
                                               (nOcurrencias 0 (0 :: xs'))
                                 = true *)
      simpl.                  (* menor_o_igual (nOcurrencias 0 xs')
                                               (S (nOcurrencias 0 xs'))
                                 = true *)
      rewrite menor_o_igual_n_Sn. (* true = true *)
      reflexivity.
    +                         (* menor_o_igual (nOcurrencias 0
                                                 (eliminaUna 0 (S x :: xs')))
                                               (nOcurrencias 0 (S x :: xs'))
                                 = true *)
      simpl.                  (* menor_o_igual (nOcurrencias 0 (eliminaUna 0
                                               (nOcurrencias 0 xs')
                                 = true *)
      rewrite HI.             (* true = true *)
      reflexivity.
Qed.

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

Theorem nOcurrencias_suma:
```

```
  forall x : nat, forall xs ys : multiconjunto,
   nOcurrencias x (suma xs ys) = nOcurrencias x xs + nOcurrencias x ys.
Proof.
  intros x xs ys.               (* x : nat
                                   xs, ys : multiconjunto
                                   ============================
                                   nOcurrencias x (suma xs ys) =
                                   nOcurrencias x xs + nOcurrencias x ys *)
  induction xs as [|x' xs' HI].
  -                             (* x : nat
                                   ys : multiconjunto
                                   ============================
                                   nOcurrencias x (suma [ ] ys) =
                                   nOcurrencias x [ ] + nOcurrencias x ys *)
    reflexivity.
  -                             (* x, x' : nat
                                   xs' : ListaNat
                                   ys : multiconjunto
                                   HI : nOcurrencias x (suma xs' ys) =
                                        nOcurrencias x xs' + nOcurrencias x ys
                                   ============================
                                   nOcurrencias x (suma (x' :: xs') ys) =
                                   nOcurrencias x (x' :: xs') + nOcurrencias x y
    simpl.                      (* (if iguales_nat x' x
                                       then S (nOcurrencias x (suma xs' ys))
                                       else nOcurrencias x (suma xs' ys))
                                   =
                                   (if iguales_nat x' x
                                       then S (nOcurrencias x xs')
                                       else nOcurrencias x xs') + nOcurrencias x
    destruct (iguales_nat x' x).
    +                           (* S (nOcurrencias x (suma xs' ys)) =
                                   S (nOcurrencias x xs') + nOcurrencias x ys *)
      rewrite HI.               (* S (nOcurrencias x xs' + nOcurrencias x ys) =
                                   S (nOcurrencias x xs') + nOcurrencias x ys *)
      reflexivity.
    +                           (* nOcurrencias x (suma xs' ys) =
                                   nOcurrencias x xs' + nOcurrencias x ys *)
      rewrite HI.               (* nOcurrencias x xs' + nOcurrencias x ys =
                                   nOcurrencias x xs' + nOcurrencias 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 contructores
      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,
      nthOpcional [4;5;6;7] 0 = Some 4.
      nthOpcional [4;5;6;7] 3 = Some 7.
      nthOpcional [4;5;6;7] 9 = 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 (primeroOpciona
  destruct xs as [|x xs'].
  -                         (* d : nat
                               ============================
                               primero d [] = eliminaOpcionalNat d (primeroOpciona
    reflexivity.
  -                         (* x : nat
                               xs' : ListaNat
                               d : nat
                               ============================
                               primero d (x :: xs') =
                               eliminaOpcionalNat d (primeroOpcional (x :: xs')) *
    simpl.                  (* x = x *)
    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 contructores
      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.              (* valor (Id n) (actualiza d (Id n) v) = Some v *)
  simpl.                   (* (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
 *)
```

Tema 4

Polimorfismo y funciones de orden superior en Coq

```
(* T4: Polimorfismo y funciones de orden superior en Coq *)

Require Export T3_Listas.

(* El contenido de la teoría es
   1. Polimorfismo
      1. Listas polimórficas
         1. Inferencia de tipos
         2. Síntesis de los tipos de los argumentos
         3. Argumentos implícitos
         4. Explicitación de argumentos
         5. Ejercicios
      2. Polimorfismo de pares
      3. Resultados opcionales polimórficos
   2. Funciones como datos
      1. Funciones de orden superior
      2. Filtrado
      3. Funciones anónimas
      4. Aplicación a todos los elementos (map)
      5. Plegados (fold)
      6. Funciones que construyen funciones
   3. Ejercicios
   4. Bibliografía
*)

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

```
   § 1. Polimorfismo
   ================================================================== *)

(* ====================================================================
   §§ 1.1. Listas polimórficas
   ================================================================== *)

(* --------------------------------------------------------------------
   Nota. Se suprime algunos avisos.
   ------------------------------------------------------------------ *)

Set Warnings "-notation-overridden,-parsing".

(* --------------------------------------------------------------------
   Ejemplo 1.1.1. Definir el tipo (list X) para representar las listas
   de elementos de tipo X con los constructores nil y cons tales que
   + nil es la lista vacía y
   + (cons x ys) es la lista obtenida añadiendo el elemento x a la
     lista ys.
   ------------------------------------------------------------------ *)

Inductive list (X:Type) : Type :=
  | nil  : list X
  | cons : X -> list X -> list X.

(* --------------------------------------------------------------------
   Ejemplo 1.1.2. Calcular el tipo de list.
   ------------------------------------------------------------------ *)

Check list.
(* ===> list : Type -> Type *)

(* --------------------------------------------------------------------
   Ejemplo 1.1.3. Calcular el tipo de (nil nat).
   ------------------------------------------------------------------ *)

Check (nil nat).
(* ===> nil nat : list nat *)

(* --------------------------------------------------------------------
```

```
   Ejemplo 1.1.4. Calcular el tipo de (cons nat 3 (nil nat)).
   ----------------------------------------------------------------- *)

Check (cons nat 3 (nil nat)).
(* ===> cons nat 3 (nil nat) : list nat *)

(* ---------------------------------------------------------------------
   Ejemplo 1.1.5. Calcular el tipo de nil.
   ----------------------------------------------------------------- *)

Check nil.
(* ===> nil : forall X : Type, list X *)

(* ---------------------------------------------------------------------
   Ejemplo 1.1.6. Calcular el tipo de cons.
   ----------------------------------------------------------------- *)

Check cons.
(* ===> cons : forall X : Type, X -> list X -> list X *)

(* ---------------------------------------------------------------------
   Ejemplo 1.1.7. Calcular el tipo de
      (cons nat 2 (cons nat 1 (nil nat))).
   ----------------------------------------------------------------- *)

Check (cons nat 2 (cons nat 1 (nil nat))).
(* ==> cons nat 2 (cons nat 1 (nil nat)) : list nat *)

(* ---------------------------------------------------------------------
   Ejemplo 1.1.8. Definir la función
      repite (X : Type) (x : X) (n : nat) : list X
   tal que (repite X x n) es la lista, de elementos de tipo X, obtenida
   repitiendo n veces el elemento x. Por ejemplo,
      repite nat 4 2 = cons nat 4 (cons nat 4 (nil nat)).
      repite bool false 1 = cons bool false (nil bool).
   ----------------------------------------------------------------- *)

Fixpoint repite (X : Type) (x : X) (n : nat) : list X :=
  match n with
  | 0    => nil X
```

```
  | S n' => cons X x (repite X x n')
  end.

Example prop_repite1 :
  repite nat 4 2 = cons nat 4 (cons nat 4 (nil nat)).
Proof. reflexivity.  Qed.

Example prop_repite2 :
  repite bool false 1 = cons bool false (nil bool).
Proof. reflexivity.  Qed.

(* =====================================================================
   §§§ 1.1.1. Inferencia de tipos
   ================================================================== *)

(* ---------------------------------------------------------------------
   Ejemplo 1.1.9. Definir la función
      repite' X x n : list X
   tal que (repite' X x n) es la lista obtenida repitiendo n veces el
   elemento x. Por ejemplo,
      repite' nat 4 2 = cons nat 4 (cons nat 4 (nil nat)).
      repite' bool false 1 = cons bool false (nil bool).
   ------------------------------------------------------------------ *)

Fixpoint repite' X x n : list X :=
  match n with
  | 0    => nil X
  | S n' => cons X x (repite' X x n')
  end.

(* ---------------------------------------------------------------------
   Ejemplo 1.1.10. Calcular los tipos de repite' y repite.
   ------------------------------------------------------------------ *)

Check repite'.
(* ===> forall X : Type, X -> nat -> list X *)
Check repite.
(* ===> forall X : Type, X -> nat -> list X *)

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

```
   §§§ 1.1.2. Síntesis de los tipos de los argumentos
   ================================================================== *)

(* --------------------------------------------------------------------
   Ejemplo 1.1.11. Definir la función
      repite'' X x n : list X
   tal que (repite'' X x n) es la lista obtenida repitiendo n veces el
   elemento x, usando argumentos implícitos. Por ejemplo,
      repite'' nat 4 2 = cons nat 4 (cons nat 4 (nil nat)).
      repite'' bool false 1 = cons bool false (nil bool).
   ------------------------------------------------------------------ *)

Fixpoint repite'' X x n : list X :=
  match n with
  | 0    => nil _
  | S n' => cons _ x (repite'' _ x n')
  end.

(* --------------------------------------------------------------------
   Ejemplo 1.1.12. Definir la lista formada por los números naturales 1,
   2 y 3.
   ------------------------------------------------------------------ *)

Definition list123 :=
  cons nat 1 (cons nat 2 (cons nat 3 (nil nat))).

Definition list123' :=
  cons _ 1 (cons _ 2 (cons _ 3 (nil _))).

(* ====================================================================
   §§§ 1.1.3. Argumentos implícitos
   ================================================================== *)

(* --------------------------------------------------------------------
   Ejemplo 1.1.13. Especificar las siguientes funciones y sus argumentos
   explícitos e implícitos:
   + nil
   + constructor
   + repite
   ------------------------------------------------------------------ *)
```

```
Arguments nil {X}.
Arguments cons {X} _ _.
Arguments repite {X} x n.

(* ---------------------------------------------------------------------
   Ejemplo 1.1.14. Definir la lista formada por los números naturales 1,
   2 y 3.
   ------------------------------------------------------------------ *)

Definition list123'' := cons 1 (cons 2 (cons 3 nil)).

(* ---------------------------------------------------------------------
   Ejemplo 1.1.15. Definir la función
      repite''' {X : Type} (x : X) (n : nat) : list X
   tal que (repite'' X x n) es la lista obtenida repitiendo n veces el
   elemento x, usando argumentos implícitos. Por ejemplo,
      repite'' nat 4 2 = cons nat 4 (cons nat 4 (nil nat)).
      repite'' bool false 1 = cons bool false (nil bool).
   ------------------------------------------------------------------ *)

Fixpoint repite''' {X : Type} (x : X) (n : nat) : list X :=
  match n with
  | 0    => nil
  | S n' => cons x (repite''' x n')
  end.

Example prop_repite'''1 :
  repite''' 4 2 = cons 4 (cons 4 nil).
Proof. reflexivity.  Qed.

Example prop_repite'''2 :
  repite false 1 = cons false nil.
Proof. reflexivity.  Qed.

(* ---------------------------------------------------------------------
   Ejemplo 1.1.16. Definir el tipo (list' {X}) para representar las
   listas de elementos de tipo X con los constructores nil' y cons'
   tales que
   + nil' es la lista vacía y
```

```
   + (cons' x ys) es la lista obtenida añadiendo el elemento x a la
     lista ys.
   ------------------------------------------------------------------ *)

Inductive list' {X:Type} : Type :=
  | nil'  : list'
  | cons' : X -> list' -> list'.

(* ---------------------------------------------------------------------
   Ejemplo 1.1.17. Definir la función
      conc {X : Type} (xs ys : list X) : (list X)
   tal que (conc xs ys) es la concatenación de xs e ys.
   ------------------------------------------------------------------ *)

Fixpoint conc {X : Type} (xs ys : list X) : (list X) :=
  match xs with
  | nil        => ys
  | cons x xs' => cons x (conc xs' ys)
  end.

(* ---------------------------------------------------------------------
   Ejemplo 1.1.18. Definir la función
      inversa {X:Type} (l:list X) : list X
   tal que (inversa xs) es la inversa de xs. Por ejemplo,
      inversa (cons 1 (cons 2 nil)) = (cons 2 (cons 1 nil)).
      inversa (cons true nil)       = cons true nil.
   ------------------------------------------------------------------ *)

Fixpoint inversa {X:Type} (xs:list X) : list X :=
  match xs with
  | nil        => nil
  | cons x xs' => conc (inversa xs') (cons x nil)
  end.

Example prop_inversa1 :
  inversa (cons 1 (cons 2 nil)) = (cons 2 (cons 1 nil)).
Proof. reflexivity.  Qed.

Example prop_inversa2:
  inversa (cons true nil) = cons true nil.
```

```
Proof. reflexivity.  Qed.

(* ---------------------------------------------------------------------
   Ejemplo 1.1.19. Definir la función
      longitud {X : Type} (xs : list X) : nat
   tal que (longitud xs) es el número de elementos de xs. Por ejemplo,
      longitud (cons 1 (cons 2 (cons 3 nil))) = 3.
   ------------------------------------------------------------------ *)

Fixpoint longitud {X : Type} (xs : list X) : nat :=
  match xs with
  | nil        => 0
  | cons _ xs' => S (longitud xs')
  end.

Example prop_longitud1:
  longitud (cons 1 (cons 2 (cons 3 nil))) = 3.
Proof. reflexivity.  Qed.

(* =====================================================================
   §§§ 1.1.4. Explicitación de argumentos
   ================================================================== *)

(* ---------------------------------------------------------------------
   Ejemplo 1.1.20. Evaluar la siguiente expresión
      Fail Definition n_nil := nil.
   ------------------------------------------------------------------ *)

Fail Definition n_nil := nil.
(* ==> Error: Cannot infer the implicit parameter X of nil. *)

(* ---------------------------------------------------------------------
   Ejemplo 1.1.21. Completar la definición anterior para obtener la
   lista vacía de números naturales.
   ------------------------------------------------------------------ *)

(* 1ª solución *)
Definition n_nil : list nat := nil.

(* 2ª solución *)
```

```
Definition n_nil' := @nil nat.

(* ---------------------------------------------------------------------
   Ejemplo 1.1.22. Definir las siguientes abreviaturas
   + "x :: y"          para (cons x y)
   + "[ ]"             para nil
   + "[ x ; .. ; y ]" para (cons x .. (cons y []) ..).
   + "x ++ y"          para (conc x y)
   ------------------------------------------------------------------ *)

Notation "x :: y"         := (cons x y)
                             (at level 60, right associativity).
Notation "[ ]"            := nil.
Notation "[ x ; .. ; y ]" := (cons x .. (cons y []) ..).
Notation "x ++ y"         := (conc x y)
                              (at level 60, right associativity).

(* ---------------------------------------------------------------------
   Ejemplo 1.1.23. Definir la lista cuyos elementos son 1, 2 y 3.
   ------------------------------------------------------------------ *)

Definition list123''' := [1; 2; 3].

(* =====================================================================
   §§§ 1.1.5. Ejercicios
   ================================================================== *)

(* ---------------------------------------------------------------------
   Ejercicio 1.1.1. Demostrar que la lista vacía es el elemento neutro
   por la derecha de la concatenación.
   ------------------------------------------------------------------ *)

Theorem conc_nil: forall (X:Type), forall xs:list X,
  xs ++ [] = xs.
Proof.
  induction xs as [|x xs' HI].
  -                            (* X : Type
                                  ============================
                                  [ ] ++ [ ] = [ ] *)
    reflexivity.
```

```
  -                             (* X : Type
                                   x : X
                                   xs' : list X
                                   HI : xs' ++ [ ] = xs'
                                   ============================
                                   (x :: xs') ++ [ ] = x :: xs' *)
    simpl.                      (* x :: (xs' ++ [ ]) = x :: xs' *)
    rewrite HI.                 (* x :: xs' = x :: xs' *)
    reflexivity.
Qed.

(* ---------------------------------------------------------------------
   Ejercicio 1.1.2. Demostrar que la concatenación es asociativa.
   ------------------------------------------------------------------ *)

Theorem conc_asociativa : forall A (xs ys zs:list A),
  xs ++ (ys ++ zs) = (xs ++ ys) ++ zs.
Proof.
  intros A xs ys zs.            (* A : Type
                                   xs, ys, zs : list A
                                   ============================
                                   xs ++ (ys ++ zs) = (xs ++ ys) ++ zs *)
  induction xs as [|x xs' HI].
  +                             (* A : Type
                                   ys, zs : list A
                                   ============================
                                   [ ] ++ (ys ++ zs) = ([ ] ++ ys) ++ zs *)
    reflexivity.
  +                             (* A : Type
                                   x : A
                                   xs', ys, zs : list A
                                   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.

(* ---------------------------------------------------------------------
   Ejercicio 1.1.3. Demostrar que la longitud de una concatenación es la
   suma de las longitudes de las listas (es decir, es un homomorfismo).
   ------------------------------------------------------------------- *)

Lemma conc_longitud: forall (X:Type) (xs ys : list X),
  longitud (xs ++ ys) = longitud xs + longitud ys.
Proof.
  intros X xs ys.              (* X : Type
                                  xs, ys : list X
                                  ============================
                                  longitud (xs ++ ys) =
                                  longitud xs + longitud ys *)
  induction xs as [|x xs' HI].
  +                            (* X : Type
                                  ys : list X
                                  ============================
                                  longitud ([ ] ++ ys) =
                                  longitud [ ] + longitud ys *)
    reflexivity.
  +                            (* X : Type
                                  x : X
                                  xs', ys : list X
                                  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.

(* ---------------------------------------------------------------------
   Ejercicio 1.1.4. Demostrar que
      inversa (xs ++ ys) = inversa ys ++ inversa xs.
```

```
   ------------------------------------------------------------------- *)

Theorem inversa_conc: forall X (xs ys : list X),
  inversa (xs ++ ys) = inversa ys ++ inversa xs.
Proof.
  intros X xs ys.              (* X : Type
                                  xs, ys : list X
                                  ============================
                                  inversa (xs ++ ys) =
                                  inversa ys ++ inversa xs *)
  induction xs as [|x xs' HI].
  -                            (* X : Type
                                  ys : list X
                                  ============================
                                  inversa ([ ] ++ ys) =
                                  inversa ys ++ inversa [ ] *)
    simpl.
    rewrite conc_nil.
    reflexivity.
  -                            (* X : Type
                                  x : X
                                  xs', ys : list X
                                  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 1.1.5. Demostrar que la inversa es involutiva; es decir,
      inversa (inversa xs) = xs.
   ------------------------------------------------------------------- *)
```

```
Theorem inversa_involutiva : forall X : Type, forall xs : list X,
  inversa (inversa xs) = xs.
Proof.
  intros X xs.                (* X : Type
                                 xs : list X
                                 ============================
                                 inversa (inversa xs) = xs *)
  induction xs as [|x xs' HI].
  +                           (* X : Type
                                 ============================
                                 inversa (inversa [ ]) = [ ] *)
    reflexivity.
  +                           (* X : Type
                                 x : X
                                 xs' : list X
                                 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' *)
    rewrite HI.               (* inversa [x] ++ xs' = x :: xs' *)
    reflexivity.
Qed.

(* =====================================================================
   §§ 1.2. Polimorfismo de pares
   ===================================================================== *)

(* ---------------------------------------------------------------------
   Ejemplo 1.2.1. Definir el tipo prod (X Y) con el constructor par tal
   que (par x y) es el par cuyas componentes son x e y.
   ------------------------------------------------------------------ *)

Inductive prod (X Y : Type) : Type :=
  | par : X -> Y -> prod X Y.

Arguments par {X} {Y} _ _.
```

```
(* ---------------------------------------------------------------------
   Ejemplo 1.2.2. Definir la abreviaturas
      "( x , y )" para (par x y).
   ------------------------------------------------------------------ *)

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

(* ---------------------------------------------------------------------
   Ejemplo 1.2.3. Definir la abreviatura
      "X * Y" para (prod X Y)
   ------------------------------------------------------------------ *)

Notation "X * Y" := (prod X Y) : type_scope.

(* ---------------------------------------------------------------------
   Ejemplo 1.2.4. Definir la función
      fst {X Y : Type} (p : X * Y) : X
   tal que (fst p) es la primera componente del par p. Por ejemplo,
      fst (par 3 5) = 3
   ------------------------------------------------------------------ *)

Definition fst {X Y : Type} (p : X * Y) : X :=
  match p with
  | (x, y) => x
  end.

Compute (fst (par 3 5)).
(* = 3 : nat*)

Example prop_fst: fst (par 3 5) = 3.
Proof. reflexivity. Qed.

(* ---------------------------------------------------------------------
   Ejemplo 2.2.5. Definir la función
      snd {X Y : Type} (p : X * Y)
   tal que (snd p) es la segunda componente del par p. Por ejemplo,
      snd (par 3 5) = 5
   ------------------------------------------------------------------ *)

Definition snd {X Y : Type} (p : X * Y) : Y :=
```

```
  match p with
  | (x, y) => y
  end.

Compute (snd (par 3 5)).
(* = 5 : nat*)

Example prop_snd: snd (par 3 5) = 5.
Proof. reflexivity. Qed.

(* ---------------------------------------------------------------------
   Ejercicio 2.2.1. Definir la función
      empareja {X Y : Type} (xs : list X) (ys : list Y) : list (X*Y)
   tal que (empareja xs ys) es la lista obtenida emparejando los
   elementos de xs y ys. Por ejemplo,
      empareja [2;6] [3;5;7]     = [(2, 3); (6, 5)].
      empareja [2;6;4;8] [3;5;7] = [(2, 3); (6, 5); (4, 7)].
   ------------------------------------------------------------------ *)

Fixpoint empareja {X Y : Type} (xs : list X) (ys : list Y) : list (X*Y) :=
  match xs, ys with
  | []      , _       => []
  | _       , []      => []
  | x :: tx, y :: ty => (x, y) :: (empareja tx ty)
  end.

Compute (empareja [2;6] [3;5;7]).
(* = [(2, 3); (6, 5)] : list (nat * nat)*)
Compute (empareja [2;6;4;8] [3;5;7]).
(* = [(2, 3); (6, 5); (4, 7)] : list (nat * nat)*)

Example prop_combina1: empareja [2;6] [3;5;7] = [(2, 3); (6, 5)].
Proof. reflexivity. Qed.

Example prop_combina2: empareja [2;6;4;8] [3;5;7] = [(2,3);(6, 5);(4,7)].
Proof. reflexivity. Qed.

(* ---------------------------------------------------------------------
   Ejercicio 2.2.2. Evaluar la expresión
      Check @empareja
```

```
   ---------------------------------------------------------------- *)

Check @empareja.
(* ==> forall X Y : Type, list X -> list Y -> list (X * Y)*)

(* -------------------------------------------------------------------
   Ejercicio 2.2.3. Definir la función
      desempareja {X Y : Type} (ps : list (X*Y)) : (list X) * (list Y)
   tal que (desempareja ps) es el par de lista (xs,ys) cuyo
   emparejamiento es l. Por ejemplo,
      desempareja [(1,false);(2,false)] = ([1;2],[false;false]).
   ---------------------------------------------------------------- *)

Fixpoint desempareja {X Y:Type} (ps:list (X*Y)) : (list X) * (list Y) :=
  match ps with
  | []            => ([], [])
  | (x, y) :: ps' => let ps'' := desempareja ps'
                     in (x :: fst ps'', y :: snd ps'')
end.

Compute (desempareja [(2, 3); (6, 5)]).
(* = ([2; 6], [3; 5]) : list nat * list nat*)

Example prop_desempareja:
  desempareja [(2, 3); (6, 5)] = ([2; 6], [3; 5]).
Proof. reflexivity. Qed.

(* ===================================================================
   §§ 1.3. Resultados opcionales polimórficos
   ================================================================ *)

(* -------------------------------------------------------------------
   Ejemplo 1.3.1. Definir el tipo (Opcional X) con los constructores Some
   y None tales que
   + (Some x) es un valor de tipo X.
   + None es el valor nulo.
   ---------------------------------------------------------------- *)

Inductive Opcional (X:Type) : Type :=
  | Some : X -> Opcional X
```

```
  | None : Opcional X.

Arguments Some {X} _.
Arguments None {X}.

(* ---------------------------------------------------------------------
   Ejercicio 1.3.1. Definir la función
      nthOpcional {X : Type} (xs : list X) (n : nat) : Opcional X :=
   tal que (nthOpcional xs n) es el n-ésimo elemento de xs o None
   si la lista tiene menos de n elementos. Por ejemplo,
      nthOpcional [4;5;6;7] 0 = Some 4.
      nthOpcional [[1];[2]] 1 = Some [2].
      nthOpcional [true] 2    = None.
   ------------------------------------------------------------------ *)

Fixpoint nthOpcional {X : Type} (xs : list X) (n : nat) : Opcional X :=
  match xs with
  | []       => None
  | x :: xs' => if iguales_nat n O
                then Some x
                else nthOpcional xs' (pred n)
  end.

Example prop_nthOpcional1 : nthOpcional [4;5;6;7] 0 = Some 4.
Proof. reflexivity. Qed.

Example prop_nthOpcional2 : nthOpcional [[1];[2]] 1 = Some [2].
Proof. reflexivity. Qed.

Example prop_nthOpcional3 : nthOpcional [true] 2 = None.
Proof. reflexivity. Qed.

(* ---------------------------------------------------------------------
   Ejercicio 1.3.2. Definir la función
      primeroOpcional {X : Type} (xs : list X) : Opcional X
   tal que (primeroOpcional xs) es el primer elemento de xs, si xs es no
   vacía; o es None, en caso contrario. Por ejemplo,
      primeroOpcional [1;2]     = Some 1.
      primeroOpcional [[1];[2]] = Some [1].
   ------------------------------------------------------------------ *)
```

```
Definition primeroOpcional {X : Type} (xs : list X) : Opcional X :=
 match xs with
    | []     => None
    | x :: _ => Some x
 end.

Check @primeroOpcional.

Example prop_primeroOpcional1 : primeroOpcional [1;2] = Some 1.
Proof. reflexivity. Qed.

Example prop_primeroOpcional2 : primeroOpcional  [[1];[2]]  = Some [1].
 Proof. reflexivity. Qed.

(* ===================================================================
   § 2. Funciones como datos
   ================================================================ *)

(* ===================================================================
   §§ 2.1. Funciones de orden superior
   ================================================================ *)

(* -------------------------------------------------------------------
   Ejemplo 2.1.1. Definir la función
      aplica3veces {X : Type} (f : X -> X) (n : X) : X
   tal que (aplica3veces f) aplica 3 veces la función f. Por ejemplo,
      aplica3veces menosDos 9     = 3.
      aplica3veces negacion true  = false.
   ---------------------------------------------------------------- *)

Definition aplica3veces {X : Type} (f : X -> X) (n : X) : X :=
  f (f (f n)).

Check @aplica3veces.
(* ===> aplica3veces : forall X : Type, (X -> X) -> X -> X *)

Example prop_aplica3veces: aplica3veces menosDos 9 = 3.
Proof. reflexivity.  Qed.
```

```
Example prop_aplica3veces': aplica3veces negacion true = false.
Proof. reflexivity.  Qed.

(* =====================================================================
   §§ 2.2. Filtrado
   ================================================================== *)

(* ---------------------------------------------------------------------
   Ejemplo 2.2.1. Definir la función
      filtra {X : Type} (p : X -> bool) (xs : list X) : (list X)
   tal que (filtra p xs) es la lista de los elementos de xs que
   verifican p. Por ejemplo,
      filtra esPar [1;2;3;4] = [2;4].
   ------------------------------------------------------------------ *)

Fixpoint filtra {X : Type} (p : X -> bool) (xs : list X) : (list X) :=
  match xs with
  | []       => []
  | x :: xs' => if p x
                then x :: (filtra p xs')
                else filtra p xs'
  end.

Example prop_filtra1: filtra esPar [1;2;3;4] = [2;4].
Proof. reflexivity.  Qed.

(* ---------------------------------------------------------------------
   Ejemplo 2.2.2. Definir la función
      unitarias {X : Type} (xss : list (list X)) : list (list X) :=
   tal que (unitarias xss) es la lista de listas unitarias de xss. Por
   ejemplo,
      unitarias [[1;2];[3];[4];[5;6;7];[];[8]] = [[3];[4];[8]]
   ------------------------------------------------------------------ *)

Definition esUnitaria {X : Type} (xs : list X) : bool :=
  iguales_nat (longitud xs) 1.

Definition unitarias {X : Type} (xss : list (list X)) : list (list X) :=
  filtra esUnitaria xss.
```

```
Compute (unitarias [[1; 2]; [3]; [4]; [5;6;7]; []; [8]]).
(* = [[3]; [4]; [8]] : list (list nat)*)

Example prop_unitarias:
  unitarias [[1; 2]; [3]; [4]; [5;6;7]; []; [8]]
  = [[3]; [4]; [8]].
Proof. reflexivity.  Qed.

(* ---------------------------------------------------------------------
   Ejercicio 2.2.3. Definir la función
      nImpares (xs : list nat) : 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 : list nat) : nat :=
  longitud (filtra esImpar 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.

(* =====================================================================
   §§ 2.3. Funciones anónimas
   ================================================================== *)

(* ---------------------------------------------------------------------
   Ejemplo 2.3.1. Demostrar que
      aplica3veces (fun n => n * n) 2 = 256.
   ------------------------------------------------------------------ *)

Example prop_anon_fun':
```

```
  aplica3veces (fun n => n * n) 2 = 256.
Proof. reflexivity.  Qed.

(* ---------------------------------------------------------------------
   Ejemplo 2.3.2. Calcular
      filtra (fun xs => iguales_nat (longitud xs) 1)
             [ [1; 2]; [3]; [4]; [5;6;7]; []; [8] ]
   ------------------------------------------------------------------ *)

Compute (filtra (fun xs => iguales_nat (longitud xs) 1)
                [ [1; 2]; [3]; [4]; [5;6;7]; []; [8] ]).
(* = [[3]; [4]; [8]] : list (list nat)*)

(* ---------------------------------------------------------------------
   Ejercicio 2.3.3. Definir la función
      filtra_pares_menores7 (xs : list nat) : list nat
   tal que (filtra_pares_mayores7 xs) es la lista de los elementos de xs
   que son pares y mayores que 7. Por ejemplo,
      filtra_pares_mayores7 [1;2;6;9;10;3;12;8] = [10;12;8].
      filtra_pares_mayores7 [5;2;6;19;129]      = [].
   ------------------------------------------------------------------ *)

Definition filtra_pares_mayores7 (xs : list nat) : list nat :=
  filtra (fun x => esPar x && menor_o_igual 7 x) xs.

Example prop_filtra_pares_mayores7_1 :
  filtra_pares_mayores7 [1;2;6;9;10;3;12;8] = [10;12;8].
Proof. reflexivity. Qed.

Example prop_filtra_pares_mayores7_2 :
  filtra_pares_mayores7 [5;2;6;19;129] = [].
Proof. reflexivity. Qed.

(* ---------------------------------------------------------------------
   Ejercicio 2.3.4. Definir la función
      partition {X : Type} (p : X -> bool) (xs : list X) : list X * list X
   tal que (patition p xs) es el par de listas (ys,zs) donde xs es la
   lista de los elementos de xs que cumplen p y zs la de las que no lo
   cumplen. Por ejemplo,
      partition esImpar [1;2;3;4;5]         = ([1;3;5], [2;4]).
```

```
     partition (fun x => false) [5;9;0] = ([], [5;9;0]).
   ------------------------------------------------------------------ *)

Definition partition {X : Type}
                     (p : X -> bool)
                     (xs : list X)
                   : list X * list X :=
  (filtra p xs, filtra (fun x => negacion (p x)) xs).

Example prop_partition1: partition esImpar [1;2;3;4;5] = ([1;3;5], [2;4]).
Proof. reflexivity. Qed.

Example prop_partition2: partition (fun x => false) [5;9;0] = ([], [5;9;0]).
Proof. reflexivity. Qed.

(* =====================================================================
   §§ 2.4. Aplicación a todos los elementos (map)
   ================================================================== *)

(* ---------------------------------------------------------------------
   Ejercicio 2.4.1. Definir la función
      map {X Y:Type} (f : X -> Y) (xs:list X) : list Y
   tal que (map f xs) es la lista obtenida aplicando f a todos los
   elementos de xs. Por ejemplo,
      map (fun x => plus 3 x) [2;0;2] = [5;3;5].
      map esImpar [2;1;2;5] = [false;true;false;true].
      map (fun n => [evenb n;esImpar n]) [2;1;2;5]
        = [[true;false];[false;true];[true;false];[false;true]].
   ------------------------------------------------------------------ *)

Fixpoint map {X Y:Type} (f : X -> Y) (xs : list X) : list Y :=
  match xs with
  | []       => []
  | x :: xs' => f x :: map f xs'
  end.

Example prop_map1:
  map (fun x => plus 3 x) [2;0;2] = [5;3;5].
Proof. reflexivity.  Qed.
```

```
Example prop_map2:
  map esImpar [2;1;2;5] = [false;true;false;true].
Proof. reflexivity.  Qed.

Example prop_map3:
    map (fun n => [esPar n ; esImpar n]) [2;1;2;5]
  = [[true;false];[false;true];[true;false];[false;true]].
Proof. reflexivity.  Qed.

(* ---------------------------------------------------------------------
   Ejercicio 2.4.2. Demostrar que
      map f (inversa l) = inversa (map f l).
   ------------------------------------------------------------------ *)

Lemma map_conc: forall (X Y : Type) (f : X -> Y) (xs ys : list X),
    map f (xs ++ ys) = map f xs ++ map f ys.
Proof.
  intros X Y f xs ys.          (* X : Type
                                  Y : Type
                                  f : X -> Y
                                  xs, ys : list X
                                  ============================
                                  map f (xs ++ ys) = map f xs ++ map f ys *)
  induction xs as [|x xs' HI].
  +                            (* X : Type
                                  Y : Type
                                  f : X -> Y
                                  ys : list X
                                  ============================
                                  map f ([ ] ++ ys) = map f [ ] ++ map f ys *)
    reflexivity.
  +                            (* X : Type
                                  Y : Type
                                  f : X -> Y
                                  x : X
                                  xs', ys : list X
                                  HI : map f (xs' ++ ys) =
                                       map f xs' ++ map f ys
                                  ============================
                                  map f ((x :: xs') ++ ys) =
```

```
                                  map f (x :: xs') ++ map f ys *)
    simpl.                     (* f x :: map f (xs' ++ ys) =
                                  f x :: (map f xs' ++ map f ys) *)
    rewrite HI.                (* f x :: (map f xs' ++ map f ys) =
                                  f x :: (map f xs' ++ map f ys) *)
    reflexivity.
Qed.

Theorem map_inversa : forall (X Y : Type) (f : X -> Y) (xs : list X),
  map f (inversa xs) = inversa (map f xs).
Proof.
  intros X Y f xs.             (* X : Type
                                  Y : Type
                                  f : X -> Y
                                  xs : list X
                                  ==============================
                                  map f (inversa xs) = inversa (map f xs) *)
  induction xs as [|x xs' HI].
  +                            (* X : Type
                                  Y : Type
                                  f : X -> Y
                                  ==============================
                                  map f (inversa [ ]) = inversa (map f [ ]) *)
    reflexivity.
  +                            (* X : Type
                                  Y : Type
                                  f : X -> Y
                                  x : X
                                  xs' : list X
                                  HI : map f (inversa xs') = inversa (map f xs')
                                  ==============================
                                  map f (inversa (x :: xs')) =
                                  inversa (map f (x :: xs')) *)
    simpl.                     (* map f (inversa xs' ++ [x]) =
                                  inversa (map f xs') ++ [f x] *)
    rewrite map_conc.          (* map f (inversa xs') ++ map f [x] =
                                  inversa (map f xs') ++ [f x] *)
    rewrite HI.                (* inversa (map f xs') ++ map f [x] =
                                  inversa (map f xs') ++ [f x] *)
    reflexivity.
```

```
Qed.

(* ---------------------------------------------------------------------
   Ejercicio 2.4.3. Definir la función
      conc_map {X Y : Type} (f : X -> list Y) (xs :list X) : (list Y)
   tal que (conc_map f xs) es la concatenación de las listas obtenidas
   aplicando f a l. Por ejemplo,
      conc_map (fun n => [n;n;n]) [1;5;4] = [1; 1; 1; 5; 5; 5; 4; 4; 4].
   ------------------------------------------------------------------ *)

Fixpoint conc_map {X Y : Type} (f : X -> list Y) (xs : list X) : (list Y) :=
   match xs with
  | []        => []
  | x :: xs' => f x ++ conc_map f xs'
  end.

Example prop_conc_map:
  conc_map (fun n => [n;n;n]) [1;5;4]
  = [1; 1; 1; 5; 5; 5; 4; 4; 4].
Proof. reflexivity. Qed.

(* ---------------------------------------------------------------------
   Ejercicio 2.4.4. Definir la función
      map_opcional {X Y : Type} (f : X -> Y) (o : Opcional X) : Opcional Y
   tal que (map_opcional f o) es la aplicación de f a o. Por ejemplo,
      map_opcional S (Some 3) = Some 4
      map_opcional S None     = None
   ------------------------------------------------------------------ *)

Definition map_opcional {X Y : Type} (f : X -> Y) (o : Opcional X)
                        : Opcional Y :=
  match o with
    | None    => None
    | Some x => Some (f x)
  end.

Compute (map_opcional S (Some 3)).
(* = Some 4 : Opcional nat*)
Compute (map_opcional S None).
(* = None : Opcional nat*)
```

```
Example prop_map_opcional1: map_opcional S (Some 3) = Some 4.
Proof. reflexivity. Qed.

Example prop_map_opcional2: map_opcional S None = None.
Proof. reflexivity. Qed.

(* ======================================================================
   §§ 2.5. Plegados (fold)
   ================================================================== *)

(* ----------------------------------------------------------------------
   Ejemplo 2.5.1. Definir la función
      fold {X Y:Type} (f: X -> Y- > Y) (xs : list X) (b : Y) : Y
   tal que (fold f xs b) es el plegado de xs con la operación f a partir
   del elemento b. Por ejemplo,
      fold mult [1;2;3;4] 1                       = 24.
      fold conjuncion [true;true;false;true] true = false.
      fold conc  [[1];[];[2;3];[4]] []            = [1;2;3;4].
   ------------------------------------------------------------------ *)

Fixpoint fold {X Y:Type} (f: X -> Y -> Y) (xs : list X) (b : Y) : Y :=
  match xs with
  | nil    => b
  | x :: xs' => f x (fold f xs' b)
  end.

Check (fold conjuncion).
(* ===> fold conjuncion : list bool -> bool -> bool *)

Example fold_example1:
  fold mult [1;2;3;4] 1 = 24.
Proof. reflexivity. Qed.

Example fold_example2 :
  fold conjuncion [true;true;false;true] true = false.
Proof. reflexivity. Qed.

Example fold_example3 :
  fold conc  [[1];[];[2;3];[4]] [] = [1;2;3;4].
```

```
Proof. reflexivity. Qed.

(* =====================================================================
   §§ 2.6. Funciones que construyen funciones
   ================================================================== *)

(* ---------------------------------------------------------------------
   Ejemplo 2.6.1. Definir la función
      constante {X : Type} (x : X) : nat -> X
   tal que (constante x) es la función que a todos los naturales le
   asigna el x. Por ejemplo,
      (constante 5) 99 = 5.
   ------------------------------------------------------------------ *)

Definition constante {X : Type} (x : X) : nat -> X :=
  fun (k : nat) => x.

Example prop_constante: (constante 5) 99 = 5.
Proof. reflexivity. Qed.

(* ---------------------------------------------------------------------
   Ejemplo 2.6.2. Calcular el tipo de plus.
   ------------------------------------------------------------------ *)

Check plus.
(* ==> nat -> nat -> nat *)

(* ---------------------------------------------------------------------
   Ejemplo 2.6.3. Definir la función
      plus3 : nat -> nat
   tal que (plus3 x) es tres más x. Por ejemplo,
      plus3 4                = 7.
      aplica3veces plus3 0    = 9.
      aplica3veces (plus 3) 0 = 9.
   ------------------------------------------------------------------ *)

Definition plus3 := plus 3.

Example prop_plus3a: plus3 4 = 7.
Proof. reflexivity.  Qed.
```

```
Example prop_plus3b: aplica3veces plus3 0 = 9.
Proof. reflexivity.  Qed.

Example prop_plus3c: aplica3veces (plus 3) 0 = 9.
Proof. reflexivity.  Qed.

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

Module Exercises.

(* --------------------------------------------------------------------
   Ejercicio 3.1. Definir, usando fold, la función
      longitudF {X : Type} (xs : list X) : nat
   tal que (longitudF xs) es la longitud de xs. Por ejemplo,
      longitudF [4;7;0] = 3.
   ----------------------------------------------------------------- *)

Definition longitudF {X : Type} (xs : list X) : nat :=
  fold (fun _ n => S n) xs 0.

Example prop_longitudF1: longitudF [4;7;0] = 3.
Proof. reflexivity. Qed.

(* --------------------------------------------------------------------
   Ejemplo 3.2. Demostrar que
      longitudF l = longitud l.
   ----------------------------------------------------------------- *)

Theorem longitudF_longitud: forall X (xs : list X),
  longitudF xs = longitud xs.
Proof.
  intros X xs.                (* X : Type
                                 xs : list X
                                 ============================
                                 longitudF xs = longitud xs *)
  unfold longitudF.           (* fold (fun (_ : X) (n : nat) => S n) xs 0 =
                                 longitud xs *)
```

```
  induction xs as [|x xs' HI].
  -                           (* X : Type
                                 ============================
                                 fold (fun (_ : X) (n : nat) => S n) [ ] 0 =
                                 longitud [ ] *)
    reflexivity.
  -                           (* X : Type
                                 x : X
                                 xs' : list X
                                 HI : fold (fun (_:X) (n:nat) => S n) xs' 0 =
                                      longitud xs'
                                 ============================
                                 fold (fun (_:X) (n:nat) => S n) (x::xs') 0 =
                                 longitud (x :: xs') *)
    simpl.                    (* S (fold (fun (_:X) (n:nat) => S n) xs' 0) =
                                 S (longitud xs') *)
    rewrite HI.               (* S (longitud xs') = S (longitud xs') *)
    reflexivity.
Qed.

(* ---------------------------------------------------------------------
   Ejercicio 3.2. Definir, usando fold, la función
      mapF {X Y : Type} (f : X -> Y) (xs : list X) : list Y
   tal que (mapF f xs) es la lista obtenida aplicando f a los
   elementos de l.
   ------------------------------------------------------------------ *)

Definition mapF {X Y : Type} (f : X -> Y) (xs : list X) : list Y :=
   fold (fun x t => (f x) :: t)  xs [].

(* ---------------------------------------------------------------------
   Ejercicio 3.4. Demostrar que mapF es equivalente a map.
   ------------------------------------------------------------------ *)

Theorem mapF_correct : forall (X Y : Type) (f : X -> Y) (xs : list X),
    mapF f xs = map f xs.
Proof.
  intros X Y f xs.            (* X : Type
                                 Y : Type
                                 f : X -> Y
```

```
                                 xs : list X
                                 ============================
                                 mapF f xs = map f xs *)
  unfold mapF.                (* fold (fun (x:X) (t:list Y) => f x::t) xs []
                                 = map f xs *)
  induction xs as [|x xs' HI].
  -                           (* X : Type
                                 Y : Type
                                 f : X -> Y
                                 ============================
                                 fold (fun (x:X) (t:list Y) => f x :: t) [] []
                                 = map f [ ] *)
    reflexivity.
  -                           (* X : Type
                                 Y : Type
                                 f : X -> Y
                                 x : X
                                 xs' : list X
                                 HI : fold (fun (x:X) (t:list Y) => f x :: t)
                                           xs' [ ]
                                      = map f xs'
                                 ============================
                                 fold (fun (x0:X) (t:list Y) => f x0 :: t)
                                      (x :: xs') [ ]
                                 = map f (x :: xs') *)
    simpl.                    (* f x :: fold (fun (x0:X) (t:list Y) =>
                                                f x0 :: t) xs' [ ]
                                 = f x :: map f xs' *)
    rewrite HI.               (* f x :: map f xs' = f x :: map f xs' *)
    reflexivity.
Qed.

(* ---------------------------------------------------------------------
   Ejemplo 3.5. Definir la función
      curry {X Y Z : Type} (f : X * Y -> Z) (x : X) (y : Y) : Z
   tal que (curry f x y) es la versión curryficada de f. Por ejemplo,
      curry fst 3 5 = 3.
   ------------------------------------------------------------------ *)

Definition curry {X Y Z : Type}
```

```
  (f : X * Y -> Z) (x : X) (y : Y) : Z := f (x, y).

Compute (curry fst 3 5).
(* = 3 : nat*)

Example prop_curry: curry fst 3 5 = 3.
Proof. reflexivity. Qed.

(* ---------------------------------------------------------------------
   Ejercicio 3.6. Definir la función
      uncurry {X Y Z : Type} (f : X -> Y -> Z) (p : X * Y) : Z
   tal que (uncurry f p) es la versión incurryficada de f.
   ------------------------------------------------------------------ *)

Definition uncurry {X Y Z : Type}
  (f : X -> Y -> Z) (p : X * Y) : Z := f (fst p) (snd p).

Compute (uncurry mult (2,5)).
(* = 10 : nat*)

Example prop_uncurry: uncurry mult (2,5) = 10.
Proof. reflexivity. Qed.

(* ---------------------------------------------------------------------
   Ejercicio 3.7. Calcular el tipo de las funcciones curry y uncurry.
   ------------------------------------------------------------------ *)

Check @curry.
(* ===> forall X Y Z : Type, (X * Y -> Z) -> X -> Y -> Z *)

Check @uncurry.
(* forall X Y Z : Type, (X -> Y -> Z) -> X * Y -> Z *)

(* ---------------------------------------------------------------------
   Ejercicio 3.8. Demostrar que
      curry (uncurry f) x y = f x y
   ------------------------------------------------------------------ *)

Theorem uncurry_curry : forall (X Y Z : Type)
                               (f : X -> Y -> Z)
```

```
                          x y,
  curry (uncurry f) x y = f x y.
Proof. reflexivity. Qed.

(* ---------------------------------------------------------------------
   Ejercicio 3.9. Demostrar que
      uncurry (curry f) p = f p.
   ------------------------------------------------------------------ *)

Theorem curry_uncurry : forall (X Y Z : Type)
                          (f : (X * Y) -> Z)
                          (p : X * Y),
  uncurry (curry f) p = f p.
Proof.
  intros.      (* X : Type
                  Y : Type
                  Z : Type
                  f : X * Y -> Z
                  p : X * Y
                  ============================
                  uncurry (curry f) p = f p *)
  destruct p.  (* uncurry (curry f) (x, y) = f (x, y) *)
  reflexivity.
Qed.

Module Church.

(* ---------------------------------------------------------------------
   Ejercicio 3.11.1. En los siguientes ejercicios se trabajará con la
   definición de Church de los números naturales: el número natural n es
   la función que toma como argumento una función f y devuelve como
   valor la aplicación de n veces la función f.

   Definir el tipo nat para los números naturales de Church.
   ------------------------------------------------------------------ *)

Definition nat := forall X : Type, (X -> X) -> X -> X.

(* ---------------------------------------------------------------------
   Ejercicio 3.11.2. Definir la función
```

```
      uno : nat
   tal que uno es el número uno de Church.
   ------------------------------------------------------------------- *)

Definition uno : nat :=
  fun (X : Type) (f : X -> X) (x : X) => f x.

(* ---------------------------------------------------------------------
   Ejercicio 3.11.3. Definir la función
      dos : nat
   tal que dos es el número dos de Church.
   ------------------------------------------------------------------- *)

Definition dos : nat :=
  fun (X : Type) (f : X -> X) (x : X) => f (f x).

(* ---------------------------------------------------------------------
   Ejercicio 3.11.4. Definir la función
      cero : nat
   tal que cero es el número cero de Church.
   ------------------------------------------------------------------- *)

Definition cero : nat :=
  fun (X : Type) (f : X -> X) (x : X) => x.

(* ---------------------------------------------------------------------
   Ejercicio 3.11.5. Definir la función
      tres : nat
   tal que tres es el número tres de Church.
   ------------------------------------------------------------------- *)

Definition tres : nat := @aplica3veces.

(* ---------------------------------------------------------------------
   Ejercicio 3.11.5. Definir la función
      suc (n : nat) : nat
   tal que (suc n) es el siguiente del número n de Church. Por ejemplo,
      suc cero = uno.
      suc uno  = dos.
      suc dos  = tres.
```

```
   ------------------------------------------------------------------ *)

Definition suc (n : nat) : nat :=
   fun (X : Type) (f : X -> X) (x : X) => f (n X f x).

Example prop_suc_1: suc cero = uno.
Proof. reflexivity. Qed.

Example prop_suc_2: suc uno = dos.
Proof. reflexivity. Qed.

Example prop_suc_3: suc dos = tres.
Proof. reflexivity. Qed.

(* ---------------------------------------------------------------------
   Ejercicio 3.11.6. Definir la función
      suma (n m : nat) : nat
   tal que (suma n m) es la suma de n y m. Por ejemplo,
      suma cero uno            = uno.
      suma dos tres            = suma tres dos.
      suma (suma dos dos) tres = suma uno (suma tres tres).
   ------------------------------------------------------------------ *)

Definition suma (n m : nat) : nat :=
  fun (X : Type) (f : X -> X) (x : X) => m X f (n X f x).

Example prop_suma_1 : suma cero uno = uno.
Proof. reflexivity. Qed.

Example prop_suma_2 : suma dos tres = suma tres dos.
Proof. reflexivity. Qed.

Example prop_suma_3 :
  suma (suma dos dos) tres = suma uno (suma tres tres).
Proof. reflexivity. Qed.

(* ---------------------------------------------------------------------
   Ejercicio 3.11.7. Definir la función
      producto (n m : nat) : nat
   tal que (producto n m) es el producto de n y m. Por ejemplo,
```

```
      producto uno uno              = uno.
      producto cero (suma tres tres) = cero.
      producto dos tres             = suma tres tres.
   ------------------------------------------------------------------ *)

Definition producto (n m : nat) : nat :=
   fun (X : Type) (f : X -> X) (x : X) => n X (m X f) x.

Example prop_producto_1: producto uno uno = uno.
Proof. reflexivity. Qed.

Example prop_producto_2: producto cero (suma tres tres) = cero.
Proof. reflexivity. Qed.

Example prop_producto_3: producto dos tres = suma tres tres.
Proof. reflexivity. Qed.

(* ---------------------------------------------------------------------
   Ejercicio 3.11.8. Definir la función
      potencia (n m : nat) : nat
   tal que (potencia n m) es la potencia m-ésima de n. Por ejemplo,
      potencia dos dos   = suma dos dos.
      potencia tres dos  = suma (producto dos (producto dos dos)) uno.
      potencia tres cero = uno.
   ------------------------------------------------------------------ *)

Definition potencia (n m : nat) : nat :=
  ( fun (X : Type) (f : X -> X) (x : X) => (m (X -> X) (n X) f) x).

Example prop_potencia_1: potencia dos dos = suma dos dos.
Proof. reflexivity. Qed.

Example prop_potencia_2:
  potencia tres dos = suma (producto dos (producto dos dos)) uno.
Proof. reflexivity. Qed.

Example prop_potencia_3: potencia tres cero = uno.
Proof. reflexivity. Qed.

End Church.
```

```
End Exercises.

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

(*
 + ”Polymorphism and higher-order functions” de Peirce et als.
   http://bit.ly/2Mj5gMf *)
```