```
-------------------------------------------------------------------------------
-- NOM DU CSU (principal)           : calculdematrices.adb
-- AUTEUR DU CSU                    : Pascal Pignard
-- VERSION DU CSU                   : 1.1c
-- DATE DE LA DERNIERE MISE A JOUR  : 25 avril 2013
-- ROLE DU CSU                      : Opérations sur les matrices.
--
--
-- FONCTIONS EXPORTEES DU CSU       :
--
-- FONCTIONS LOCALES DU CSU         :
--
--
-- NOTES                            : Ada 2005, UTF8
--
-- COPYRIGHT                        : (c) Pascal Pignard 1989-2013
-- LICENCE                          : CeCILL V2 (http://www.cecill.info)
-- CONTACT                          : http://blady.pagesperso-orange.fr
-------------------------------------------------------------------------------

with Ada.Text_IO;               use Ada.Text_IO;
with Ada.Numerics.Float_Random;
with Ada.Containers.Vectors;

procedure CalculDeMatrices is

   generic
      type TElement is digits <>;
   package Matrices is
      type Matrice (<>) is tagged private;
      type Tableau is array (Positive range <>, Positive range <>) of TElement;
      function Créé_Matrice
        (Lignes, Colonnes : Positive;
         Valeur           : TElement := 0.0)
         return             Matrice;
      function Créé_Matrice (T : Tableau) return Matrice;
      function Nb_Lignes (M : Matrice) return Natural;
      function Nb_Colonnes (M : Matrice) return Natural;
      function Element (M : Matrice; Ligne, Colonne : Positive) return TElement;
      procedure Positionne (M : in out Matrice; Ligne, Colonne : Positive; Element : TElement
      function Trace (M : Matrice) return TElement;
      function Transpose (M : Matrice) return Matrice;
      function Identité (Taille : Positive) return Matrice;
      function Aléatoire (Lignes, Colonnes : Positive) return Matrice;
      function "+" (Gauche, Droite : Matrice) return Matrice;
      function "-" (M : Matrice) return Matrice;
      function "-" (Gauche, Droite : Matrice) return Matrice;
      function "*" (Gauche, Droite : Matrice) return Matrice;
      function "**" (M : Matrice; Exposant : Integer) return Matrice;
      function Déterminant (M : Matrice) return TElement;
      function Inverse (M : Matrice) return Matrice;
      function "*" (M : Matrice; Lambda : TElement) return Matrice;
      function "*" (Lambda : TElement; M : Matrice) return Matrice;
      procedure Affiche (M : Matrice);
   private
      package IntMatrices is new Ada.Containers.Vectors (Positive, TElement);
      type Matrice is new IntMatrices.Vector with record
         Lignes, Colonnes : Natural;
      end record;
      function To_Vector (Length : Ada.Containers.Count_Type) return Matrice;
      function To_Vector
```

```
      (New_Item : TElement;
       Length   : Ada.Containers.Count_Type)
       return     Matrice;
   function "&" (Left, Right : Matrice) return Matrice;
   function "&" (Left : Matrice; Right : TElement) return Matrice;
   function "&" (Left : TElement; Right : Matrice) return Matrice;
   function "&" (Left, Right : TElement) return Matrice;
end Matrices;

package body Matrices is

   function Créé_Matrice
     (Lignes, Colonnes : Positive;
      Valeur           : TElement := 0.0)
      return             Matrice
   is
   begin
      return (IntMatrices.To_Vector (Valeur, Ada.Containers.Count_Type (Lignes * Colonnes)
      with Lignes, Colonnes);
   end Créé_Matrice;

   function Créé_Matrice (T : Tableau) return Matrice is
   begin
      return M : Matrice :=
           (IntMatrices.To_Vector
               (Ada.Containers.Count_Type ((T'Last (1) + 1 - T'First (1)) *
                                           (T'Last (2) + 1 - T'First (2)))) with
            T'Last (1) + 1 - T'First (1),
            T'Last (2) + 1 - T'First (2)) do
         for I in T'Range (1) loop
            for J in T'Range (2) loop
               Positionne (M, I + 1 - T'First (1), J + 1 - T'First (2), T (I, J));
            end loop;
         end loop;
      end return;
   end Créé_Matrice;

   function Nb_Lignes (M : Matrice) return Natural is
   begin
      return M.Lignes;
   end Nb_Lignes;

   function Nb_Colonnes (M : Matrice) return Natural is
   begin
      return M.Colonnes;
   end Nb_Colonnes;

   function Element (M : Matrice; Ligne, Colonne : Positive) return TElement is
   begin
      if Ligne > M.Lignes or Colonne > M.Colonnes then
         raise Constraint_Error;
      end if;
      return Element (M, (Ligne - 1) * M.Colonnes + Colonne);
   end Element;

   procedure Positionne (M : in out Matrice; Ligne, Colonne : Positive; Element : TElement
   begin
      if Ligne > M.Lignes or Colonne > M.Colonnes then
         raise Constraint_Error;
      end if;
      Replace_Element (M, (Ligne - 1) * M.Colonnes + Colonne, Element);
```

```
   end Positionne;

   function Trace (M : Matrice) return TElement is
   begin
      if M.Lignes /= M.Colonnes then
         raise Constraint_Error;
      end if;
      return R : TElement := 0.0 do
         for I in 1 .. M.Lignes loop
            R := R + Element (M, I, I);
         end loop;
      end return;
   end Trace;

   function Transpose (M : Matrice) return Matrice is
   begin
      return R : Matrice := Créé_Matrice (M.Lignes, M.Colonnes) do
         for I in 1 .. R.Lignes loop
            for J in 1 .. R.Colonnes loop
               Positionne (R, I, J, Element (M, J, I));
            end loop;
         end loop;
      end return;
   end Transpose;

   function Identité (Taille : Positive) return Matrice is
   begin
      return R : Matrice := Créé_Matrice (Taille, Taille) do
         for I in 1 .. R.Lignes loop
            Positionne (R, I, I, 1.0);
         end loop;
      end return;
   end Identité;

   function Aléatoire (Lignes, Colonnes : Positive) return Matrice is
      G : Ada.Numerics.Float_Random.Generator;
   begin
      return R : Matrice := Créé_Matrice (Lignes, Colonnes) do
         for I in 1 .. R.Lignes loop
            for J in 1 .. R.Colonnes loop
               Positionne (R, I, J, TElement (Ada.Numerics.Float_Random.Random (G)));
            end loop;
         end loop;
      end return;
   end Aléatoire;

   function "+" (Gauche, Droite : Matrice) return Matrice is
   begin
      if Gauche.Lignes /= Droite.Lignes or Gauche.Colonnes /= Droite.Colonnes then
         raise Constraint_Error;
      end if;
      return R : Matrice := Créé_Matrice (Gauche.Lignes, Gauche.Colonnes) do
         for I in 1 .. R.Lignes loop
            for J in 1 .. R.Colonnes loop
               Positionne (R, I, J, Element (Gauche, I, J) + Element (Droite, I, J));
            end loop;
         end loop;
      end return;
   end "+";

   function "-" (M : Matrice) return Matrice is
```

```
   begin
      return (-1.0) * M;
   end "-";

   function "-" (Gauche, Droite : Matrice) return Matrice is
   begin
      return -Droite + Gauche;
   end "-";

   function "*" (Gauche, Droite : Matrice) return Matrice is
   begin
      if Gauche.Colonnes /= Droite.Lignes then
         raise Constraint_Error;
      end if;
      return R : Matrice := Créé_Matrice (Gauche.Lignes, Droite.Colonnes) do
         for I in 1 .. R.Lignes loop
            for J in 1 .. R.Colonnes loop
               declare
                  Terme : TElement := 0.0;
               begin
                  for K in 1 .. Gauche.Colonnes loop
                     Terme := Terme + Element (Gauche, I, K) * Element (Droite, K, J);
                  end loop;
                  Positionne (R, I, J, Terme);
               end;
            end loop;
         end loop;
      end return;
   end "*";

   function "**" (M : Matrice; Exposant : Integer) return Matrice is
      R : Matrice := Identité (M.Lignes);
      S : Matrice := M;
      N : Natural := abs Exposant;
   begin
      if M.Lignes /= M.Colonnes then
         raise Constraint_Error;
      end if;
      while N /= 0 loop
         if N mod 2 = 1 then
            R := R * S;
         end if;
         N := N / 2;
         S := S * S;
      end loop;
      if Exposant < 0 then
         return Inverse (R);
      end if;
      return R;
   end "**";

   function Déterminant (M : Matrice) return TElement is
      I, N    : Positive;
      LDet, D : TElement;
      LM      : Matrice := M;
   begin
      if M.Lignes /= M.Colonnes then
         raise Constraint_Error;
      end if;
      N    := LM.Lignes;
      LDet := 1.0;
```

```
      for K in 1 .. N - 1 loop
         if Element (LM, K, K) = 0.0 then
            I := K + 1;
            while (Element (LM, I, K) = 0.0) and (I /= N) loop
               I := I + 1;
            end loop;
            if Element (LM, I, K) = 0.0 then
               return 0.0;
            else
               for J in 1 .. N loop
                  Positionne (LM, K, J, Element (LM, K, J) + Element (LM, I, J));
               end loop;
            end if;
         end if;
         for I in K + 1 .. N loop
            D := Element (LM, I, K) / Element (LM, K, K);
            for J in 1 .. N loop
               Positionne (LM, I, J, Element (LM, I, J) - D * Element (LM, K, J));
            end loop;
         end loop;
         LDet := LDet * Element (LM, K, K);
      end loop;
      return LDet * Element (LM, N, N);
   end Déterminant;

   function Inverse (M : Matrice) return Matrice is
      K, N, Im, Jm  : Positive;
      Pivot, Max, A : TElement;
      El, Ec        : array (1 .. M.Lignes) of Positive;
      M2            : Matrice := M;
   begin
      if M.Lignes /= M.Colonnes then
         raise Constraint_Error;
      end if;
      N := M2.Lignes;
      for Lc in 1 .. N loop
         Max := 0.0;
         for I in Lc .. N loop
            for J in Lc .. N loop
               if abs (Element (M2, I, J)) > Max then
                  Im  := I;
                  Jm  := J;
                  Max := abs (Element (M2, I, J));
               end if;
            end loop;
         end loop;
         Pivot   := Element (M2, Im, Jm);
         El (Lc) := Im;
         Ec (Lc) := Jm;
         if Im /= Lc then
            for J in 1 .. N loop
               A := Element (M2, Im, J);
               Positionne (M2, Im, J, Element (M2, Lc, J));
               Positionne (M2, Lc, J, A);
            end loop;
         end if;
         if Jm /= Lc then
            for I in 1 .. N loop
               A := Element (M2, I, Lc);
               Positionne (M2, I, Lc, Element (M2, I, Jm));
               Positionne (M2, I, Jm, A);
```

```
            end loop;
         end if;
         for I in 1 .. N loop
            if I /= Lc then
               for J in 1 .. N loop
                  if J /= Lc then
                     Positionne
                       (M2,
                        I,
                        J,
                        Element (M2, I, J) -
                        Element (M2, I, Lc) * Element (M2, Lc, J) / Pivot);
                  end if;
               end loop;
            end if;
         end loop;
         for K in 1 .. N loop
            if K /= Lc then
               Positionne (M2, K, Lc, Element (M2, K, Lc) / Pivot);
               Positionne (M2, Lc, K, -Element (M2, Lc, K) / Pivot);
            end if;
         end loop;
         Positionne (M2, Lc, Lc, 1.0 / Pivot);
      end loop;
      for Lc in 1 .. N loop
         K  := N - Lc + 1;
         Im := El (K);
         if Im /= K then
            for I in 1 .. N loop
               A := Element (M2, I, K);
               Positionne (M2, I, K, Element (M2, I, Im));
               Positionne (M2, I, Im, A);
            end loop;
         end if;
         Jm := Ec (K);
         if Jm /= K then
            for J in 1 .. N loop
               A := Element (M2, K, J);
               Positionne (M2, K, J, Element (M2, Jm, J));
               Positionne (M2, Jm, J, A);
            end loop;
         end if;
      end loop;
      return M2;
   end Inverse;

   function "*" (M : Matrice; Lambda : TElement) return Matrice is
   begin
      return R : Matrice := Créé_Matrice (M.Lignes, M.Colonnes) do
         for I in 1 .. R.Lignes loop
            for J in 1 .. R.Colonnes loop
               Positionne (R, I, J, Lambda * Element (M, I, J));
            end loop;
         end loop;
      end return;
   end "*";

   function "*" (Lambda : TElement; M : Matrice) return Matrice is
   begin
      return M * Lambda;
   end "*";
```

```
      procedure Affiche (M : Matrice) is
      begin
         for I in 1 .. M.Lignes loop
            for J in 1 .. M.Colonnes loop
               Ada.Text_IO.Put (TElement'Image (Element (M, I, J)));
            end loop;
            Ada.Text_IO.New_Line;
         end loop;
         Ada.Text_IO.New_Line;
      end Affiche;

      function To_Vector (Length : Ada.Containers.Count_Type) return Matrice is
      begin
         return (IntMatrices.To_Vector (Length) with 0, 0);
      end To_Vector;

      function To_Vector
        (New_Item : TElement;
         Length   : Ada.Containers.Count_Type)
         return     Matrice
      is
      begin
         return (IntMatrices.To_Vector (New_Item, Length) with 0, 0);
      end To_Vector;

      function "&" (Left, Right : Matrice) return Matrice is
      begin
         return (IntMatrices."&" (IntMatrices.Vector (Left), IntMatrices.Vector (Right)) with
      end "&";

      function "&" (Left : Matrice; Right : TElement) return Matrice is
      begin
         return (IntMatrices."&" (IntMatrices.Vector (Left), Right) with 0, 0);
      end "&";

      function "&" (Left : TElement; Right : Matrice) return Matrice is
      begin
         return (IntMatrices."&" (Left, IntMatrices.Vector (Right)) with 0, 0);
      end "&";

      function "&" (Left, Right : TElement) return Matrice is
      begin
         return (IntMatrices."&" (Left, Right) with 0, 0);
      end "&";

   end Matrices;

   package MatricesRéelles is new Matrices (Float);
   use MatricesRéelles;
   V0 : constant Matrice := Créé_Matrice (4, 1, 1.0 / 4.0);
   M  : Matrice          := Créé_Matrice (((1.0, 2.0), (3.0, 1.0 / 4.0)));
   M2 : Matrice          := Créé_Matrice (4, 4, 2.0);
begin
   Affiche (Identité (3) * 2.5 + Identité (3));
   Affiche (2.0 * Identité (3) + Aléatoire (3, 3));
   Affiche (M);
   Put_Line (Déterminant (M)'Img);
   New_Line;
   Affiche (M * Inverse (M));
   Affiche (M * M ** (-1));
```

```
   Affiche (M * M * M * M * M * M * M);
   Affiche (M ** 7);
   Affiche (Inverse (M * M * M * M * M * M * M));
   Affiche (M ** (-7));
   Put_Line (Trace (M)'Img);
   New_Line;
   Affiche (Transpose (M));
   M :=
     Créé_Matrice (((1 => 1.0), (1 => 2.0), (1 => 3.0))) *
     Créé_Matrice ((1 => (1.0, 2.0, 3.0))) +
     Créé_Matrice (3, 3, 10.0);
   Affiche (M);
   Put_Line (Element (M, 2, 3)'Img);
   New_Line;
   M2 := M2 + Identité (4);
   Affiche (M2 * V0);
end CalculDeMatrices;
```