```
-------------------------------------------------------------------------------
-- NOM DU CSU (principal)           : Prog_Avan2.adb
-- AUTEUR DU CSU                    : Pascal Pignard
-- VERSION DU CSU                   : 1.17a
-- DATE DE LA DERNIERE MISE A JOUR  : 15 août 2010
-- ROLE DU CSU                      : bibliothèque de fonctions mathématiques
--
-- FONCTIONS EXPORTEES DU CSU       :
--
--
-- FONCTIONS LOCALES DU CSU         :
--
--
-- NOTES                            :
--  Basé sur les exemples et exercices du livre "Programmation Avancée"
--  "Algorithmique et structures de données"
--  J.C. Boussard, Robert Mahl
--  Eyrolles 1984
--
-- COPYRIGHT                        : (c) Pascal Pignard 2009
-- LICENCE                          : CeCILL V2 (http://www.cecill.info)
-- CONTACT                          : http://blady.pagesperso-orange.fr
-------------------------------------------------------------------------------

pragma Assertion_Policy (Check);

with Ada.Text_IO;                       use Ada.Text_IO;
with Ada.Float_Text_IO;                 use Ada.Float_Text_IO;
with Ada.Calendar;
with Ada.Numerics.Elementary_Functions;
with Ada.Characters.Handling;
with Ada.Unchecked_Deallocation;

procedure prog_avan2 is

   -- Déclarations des utilitaires :

   -- Tableau d'entiers relatifs
   type TP is array (Positive range <>) of Integer;

   -- Affiche les éléments du tableau T
   procedure Imprime (T : TP);

   -- Renvoie le compteur horaire interne en millisecondes.
   function HorlogeMS return Natural;

   -- Pile simple
   generic
      type Element is private;
      N : Positive;
      with function Image (E : Element) return String;
   package Piles is
      type Pile is tagged private;
      Plein, Vide : exception;
      procedure Empiler (P : in out Pile; Valeur : Element);
      procedure Depiler (P : in out Pile);
      procedure Depiler (P : in out Pile; Valeur : out Element);
      function Sommet (P : Pile) return Element;
      -- nommée Sommet dans le texte original provoquant une confusion avec le composant
      procedure Vider (P : in out Pile);
      procedure Ecrire (P : Pile);
      -- les deux paramètres ont des types objets
   private
      type Tableau_Element is array (1 .. N) of Element;
      type Pile is tagged record
         Pile   : Tableau_Element;
         Sommet : Natural := 0;
      end record;
   end Piles;

   -- Les exemples et exercices du livre :

   -- Suites contiguës bouclée (anneau)
```

```
-- Exemple 5.4 et exercice 5.1
generic
   type TElement is private;
   N : Positive;
   with function Image (E : TElement) return String;
package Suite is
   type TInstance is tagged private;
   -- Ajoute un élément
   procedure Adjonction (O : in out TInstance; Element : TElement);
   -- Supprime et retourne le premier élément (FIFO)
   procedure Suppression (O : in out TInstance; Element : out TElement);
   -- Affiche les éléments
   procedure Imprime (O : TInstance);
   Vide, Plein : exception;
private
   subtype TIndice is Positive range 1 .. N;
   type TTab is array (TIndice) of TElement;
   type TInstance is tagged record
      Tab   : TTab;
      Tete  : TIndice              := 1;
      Queue : Natural range 0 .. N := 0;
      Plein : Boolean              := False;
   end record;
end Suite;

-- Tri d'une suite par interclassement
-- §5.2.5A et exercice 5.2
procedure TriInterClassement (S : in out TP);

-- Tri d'une suite par segmentation
-- §5.2.5B et exemple 5.7
procedure TriSegmentation (A : in out TP);

-- Tri d'une suite selon une base "radix sort"
-- §5.3.4, exemple 5.10 et exercice 5.9
procedure TriBase10 (X : in out TP);

-- Simulation d'un magasin à 2 serveurs
-- Exemple 5.12
procedure Simulation_T;

-- Adressage dispersé contigu linéaire à clé d'entrée entière
-- Exemples 6.9, 6.10 et exercices 6.1, 6.2
type TDispersion is (Linéaire, Optimum, Quadratique);
generic
   -- Nombre d'éléments maximum à stocker
   N : Positive;
   -- Type des éléments à stocker
   type TElement is private;
   -- type de fonction de dispersion
   Dispersion : TDispersion;
package HashCoding is
   Plein : exception;
   procedure Stocke (Clé : Positive; Element : TElement);
   procedure Accede (Clé : Positive; Element : out TElement; Trouvé : out Boolean);
   procedure Supprime (Clé : Positive; Trouvé : out Boolean);
   procedure Imprime;
end HashCoding;

-- Intersection de deux ensembles par adressage dispersé
-- Exemple 6.11
procedure Intersection;

-- Union de deux ensembles par adressage dispersé
-- Exercice 6.3
procedure Union;

-- Dérivation formelle d'une expression
-- Exemple 7.3
function Dérivation (Entrée : String) return String;

-- Transforme une expression arithmétique vers une expression postfixée
-- Exemple 7.4 et exercice 7.10
```

```
function VersExpPostFixée (E : String) return String;

-- Évalue une expression postfixée
-- Exemple 7.5
function ÉvalueExpPostFixée (E : String) return String;

-- Tri arborescent
-- Exercice 7.12
procedure TriArborescent (V : in out TP);

-- Représentation linéaire fonctionnelle d'un graphe
-- Exemple 8.6 1)
generic
   -- Définition des noeuds (énuméré)
   type Noeud is (<>);
   -- Définition des Arcs (qqcn)
   type Arcs is private;
   -- Définition d'un arc fictif signifiant que l'arc correspondant n'existe pas
   Null_Arc : Arcs;
   -- Pour afficher les noeuds
   with function Image (E : Noeud) return String;
   -- Pour afficher les arcs
   with function Image (E : Arcs) return String;
package Graphe_LF is
   NonTrouvé : exception;
   procedure AjouteArc (Origine, Extrémité : Noeud; Arc : Arcs);
   procedure SupprimeArc (Origine, Extrémité : Noeud);
   procedure AfficheListeNoeudsExtrémité (Origine : Noeud);
   procedure AfficheListeNoeudsOrigine (Extrémité : Noeud);
end Graphe_LF;

-- Représentation linéaire associative contigüe d'un graphe
-- Exemple 8.6 2)
generic
   -- Nombre maximal de noeuds
   MaxNoeuds : Positive;
   -- Définition des noeuds (énuméré)
   type Noeud is (<>);
   -- Définition des Arcs (qqcn)
   type Arcs is private;
   -- Pour afficher les noeuds
   with function Image (E : Noeud) return String;
   -- Pour afficher les arcs
   with function Image (E : Arcs) return String;
package Graphe_LAC is
   Plein, NonTrouvé : exception;
   procedure AjouteArc (Origine, Extrémité : Noeud; Arc : Arcs);
   procedure SupprimeArc (Origine, Extrémité : Noeud);
   procedure AfficheListeNoeudsExtrémité (Origine : Noeud);
   procedure AfficheListeNoeudsOrigine (Extrémité : Noeud);
end Graphe_LAC;

-- Représentation linéaire associative non contigüe d'un graphe
-- Exemple 8.6 2) modifié
generic
   -- Nombre maximal de noeuds
   MaxNoeuds : Positive;
   -- Définition des noeuds (énuméré)
   type Noeud is (<>);
   -- Définition des Arcs (qqcn)
   type Arcs is private;
   -- Définition d'un arc fictif signifiant que l'arc correspondant n'existe pas
   Null_Arc : Arcs;
   -- Pour afficher les noeuds
   with function Image (E : Noeud) return String;
   -- Pour afficher les arcs
   with function Image (E : Arcs) return String;
package Graphe_LANC is
   Plein, NonTrouvé : exception;
   procedure AjouteArc (Origine, Extrémité : Noeud; Arc : Arcs);
   procedure SupprimeArc (Origine, Extrémité : Noeud);
   procedure AfficheListeNoeudsExtrémité (Origine : Noeud);
   procedure AfficheListeNoeudsOrigine (Extrémité : Noeud);
```

```
end Graphe_LANC;

-- Représentation matricielle d'un graphe
-- §8.2.2 et exemple 8.9
generic
   -- Définition des noeuds (énuméré)
   type Noeud is (<>);
   -- Définition des Arcs (qqcn)
   type Arcs is private;
   -- Définition d'un arc fictif signifiant que l'arc correspondant n'existe pas
   Null_Arc : Arcs;
   -- Pour afficher les noeuds
   with function Image (E : Noeud) return String;
   -- Pour afficher les arcs
   with function Image (E : Arcs) return String;
package Graphe_Matricielle is
   type Matrice is array (Noeud, Noeud) of Boolean;
   procedure AjouteArc (Origine, Extrémité : Noeud; Arc : Arcs);
   procedure SupprimeArc (Origine, Extrémité : Noeud);
   procedure AfficheListeNoeudsExtrémité (Origine : Noeud);
   procedure AfficheListeNoeudsOrigine (Extrémité : Noeud);
   function Retourne_Matrice_Incidence return Matrice;
   function Retourne_Chemin (Longueur : Positive) return Matrice;
   procedure Affiche_Matrice (M : Matrice);
end Graphe_Matricielle;

-- Représentation par Plexes d'un graphe
-- Exercice 8.1
generic
   type Noeud is (<>);
   type Arcs is private;
   with function Image (E : Noeud) return String;
   -- Pour afficher les arcs
   with function Image (E : Arcs) return String;
package Graphe_Plexe is
   NonTrouvé : exception;
   procedure AjouteArc (Origine, Extrémité : Noeud; Arc : Arcs);
   procedure SupprimeArc (Origine, Extrémité : Noeud);
   procedure AfficheListeNoeudsExtrémité (Origine : Noeud);
   procedure AfficheListeNoeudsOrigine (Extrémité : Noeud);
end Graphe_Plexe;

-- Représentation par adressage dispersé d'un graphe
-- Exemple 8.8
generic
   type Noeud is (<>);
   type Arcs is private;
   with function Image (E : Noeud) return String;
   -- Pour afficher les arcs
   with function Image (E : Arcs) return String;
package Graphe_Dispersé is
   NonTrouvé : exception;
   procedure AjouteArc (Origine, Extrémité : Noeud; Arc : Arcs);
   procedure SupprimeArc (Origine, Extrémité : Noeud);
   procedure AfficheListeNoeudsExtrémité (Origine : Noeud);
   procedure AfficheListeNoeudsOrigine (Extrémité : Noeud);
end Graphe_Dispersé;

-- Le code des procédures et fonctions :

procedure Imprime (T : TP) is
begin
   for I in T'Range loop
      Put (T (I)'Img);
   end loop;
   New_Line;
end Imprime;

function HorlogeMS return Natural is
begin
   return Natural (Ada.Calendar.Seconds (Ada.Calendar.Clock) * 1000.0);
end HorlogeMS;
```

```
package body Piles is
   procedure Empiler (P : in out Pile; Valeur : Element) is
   begin
      if P.Sommet >= N then
         raise Plein;
      end if;
      P.Sommet          := P.Sommet + 1;
      P.Pile (P.Sommet) := Valeur;
   end Empiler;
   procedure Depiler (P : in out Pile) is
   begin
      if P.Sommet <= 0 then
         raise Vide;
      end if;
      P.Sommet := P.Sommet - 1;
   end Depiler;
   procedure Depiler (P : in out Pile; Valeur : out Element) is
   begin
      if P.Sommet <= 0 then
         raise Vide;
      end if;
      Valeur   := P.Pile (P.Sommet);
      P.Sommet := P.Sommet - 1;
   end Depiler;
   function Sommet (P : Pile) return Element is
   begin
      if P.Sommet <= 0 then
         raise Vide;
      end if;
      return P.Pile (P.Sommet);
   end Sommet;
   procedure Vider (P : in out Pile) is
   begin
      while P.Sommet > 0 loop
         P.Depiler;
      end loop;
   end Vider;
   procedure Ecrire (P : Pile) is
   begin
      for I in 1 .. P.Sommet loop
         Put (Image (P.Pile (I)));
      end loop;
      New_Line;
   end Ecrire;
end Piles;

package body Suite is
   procedure Adjonction (O : in out TInstance; Element : TElement) is
   begin
      if O.Queue = 0 then
         O.Queue          := O.Tete;
         O.Tab (O.Queue) := Element;
      else
         O.Queue := (O.Queue mod N) + 1;
         if O.Queue /= O.Tete then
            O.Tab (O.Queue) := Element;
         else
            raise Plein;
         end if;
      end if;
   end Adjonction;
   procedure Suppression (O : in out TInstance; Element : out TElement) is
   begin
      if O.Queue = 0 then
         raise Vide;
      end if;
      Element := O.Tab (O.Tete);
      O.Tete  := (O.Tete mod N) + 1;
      if (O.Queue mod N) + 1 = O.Tete then
         O.Queue := 0;
      end if;
   end Suppression;
   procedure Imprime (O : TInstance) is
```

```
         I : TIndice := O.Tete;
      begin
         if O.Queue /= 0 then
            while I /= O.Queue loop
               Put (Image (O.Tab (I)));
               I := (I mod N) + 1;
            end loop;
            Put (Image (O.Tab (I)));
         end if;
         New_Line;
      end Imprime;
   end Suite;

   procedure TriInterClassement (S : in out TP) is
      procedure InterClass
        (A    : TP;
         I, J : Positive;
         B    : TP;
         K, L : Positive;
         C    : in out TP;
         M    : Positive)
      is
         IP : Positive := I;
         KP : Positive := K;
         MP : Positive := M;
      begin
         while IP <= J or else KP <= L loop
            if IP > J then
               C (MP) := B (KP);
               KP     := KP + 1;
            elsif KP > L then
               C (MP) := A (IP);
               IP     := IP + 1;
            elsif IP <= J and then KP <= L then
               if A (IP) < B (KP) then
                  C (MP) := A (IP);
                  IP     := IP + 1;
               else
                  C (MP) := B (KP);
                  KP     := KP + 1;
               end if;
            end if;
            MP := MP + 1;
         end loop;
      end InterClass;

      N : constant Positive := S'Length;
      -- tableau double de travail de taille double
      A : array (1 .. 2) of TP (1 .. N * 2);
      -- K indice des 2 tableaux de travail en alternance
      K        : Positive := 1;
      I, J, M  : Positive;
      Longueur : Natural;
   begin
      -- construction de N sous-listes de 1 élément
      for T in 1 .. N loop
         A (1) (2 * T - 1)  := 1;
         A (1) (2 * T)      := S (T);
      end loop;
      loop
         I        := 1;
         Longueur := 0;
         M        := 1;
         loop
            Longueur := Longueur + A (K) (I);
            if Longueur < N then
               J := I + 1 + A (K) (I);
               InterClass
                 (A (K),
                  I + 1,
                  I + A (K) (I),
                  A (K),
                  J + 1,
```

```
                J + A (K) (J),
                A (3 - K),
                M + 1);
             A (3 - K) (M)  := A (K) (I) + A (K) (J);
             M              := M + 1 + A (K) (I) + A (K) (J);
             I              := J + 1 + A (K) (J);
             Longueur       := Longueur + A (K) (J);
          else
             InterClass (A (K), I + 1, I + A (K) (I), A (K), J + 1, J, A (3 - K), M + 1);
             A (3 - K) (M)  := A (K) (I);
          end if;
          exit when Longueur = N;
       end loop;
       K := 3 - K;
       exit when A (K) (1) = N;
    end loop;
    S := A (K) (2 .. N + 1);
end TriInterClassement;

procedure TriSegmentation (A : in out TP) is
    N  : constant Positive := A'Length;
    N1 : constant Positive := N + 1;
    subtype Indice is Positive range 1 .. N;
    subtype IndInf is Natural range 0 .. N;
    subtype IndSup is Positive range 1 .. N1;
    B    : TP (Indice);
    Part : Integer;
    Inf  : IndInf;
    Sup  : IndSup;
    procedure TriSeg (D, F : Indice) is
    begin
       Part := A (D);
       Inf  := D - 1;
       Sup  := F + 1;
       for I in D + 1 .. F loop
          if A (I) <= Part then
             Inf     := Inf + 1;
             B (Inf) := A (I);
          else
             Sup     := Sup - 1;
             B (Sup) := A (I);
          end if;
       end loop;
       B (Inf + 1) := Part;
       for I in D .. F loop
          A (I) := B (I);
       end loop;
       if Inf > D then
          TriSeg (D, Inf);
       end if;
       if Sup < F then
          TriSeg (Sup, F);
       end if;
    end TriSeg;
begin
    TriSeg (A'First, A'Last);
end TriSegmentation;

procedure TriBase10 (X : in out TP) is
    function NbMaxChiffres (X : TP) return Natural is
       L, N : Natural := 0;
    begin
       for I in X'Range loop
          if X (I) = 0 then
             L := 1;
          else
             L :=
               Natural (Float'Ceiling
                           (Ada.Numerics.Elementary_Functions.Log
                               (Float (abs (X (I))),
                                10.0)));
          end if;
          if L > N then
```

```
            N := L;
         end if;
      end loop;
      return N;
   end NbMaxChiffres;
   function Chiffre (X : Integer; N : Positive) return Natural is
   begin
      return (abs (X) / 10 ** (N - 1)) mod 10;
   end Chiffre;
   N              : constant Natural := NbMaxChiffres (X);
   A              : array (X'Range, 1 .. N) of Natural; -- X nombres de N chiffres
   B              : TP (X'Range); -- pointeur suivant
   D              : TP (1 .. 10); -- pointeur début liste triée
   E              : TP (1 .. 10); -- pointeur fin liste triée
   Y              : TP (X'Range); -- tableau trié
   Tete, II, K, M : Natural;
begin
   for I in X'Range loop
      for J in 1 .. N loop
         A (I, J) := Chiffre (X (I), J) + 1;
      end loop;
      B (I) := I + 1;
   end loop;
   B (X'Last) := 0;
   Tete       := 1;
   -- tri sur le chiffre en position J
   for J in 1 .. N loop
      for P in 1 .. 10 loop
         D (P) := 0;
         E (P) := 0;
      end loop;
      while Tete /= 0 loop
         II   := Tete;
         Tete := B (Tete); -- suivant
         K    := A (II, J); -- chiffre J
         if D (K) = 0 then
            D (K) := II; -- élément II placé en début
         else
            B (E (K))  := II; -- élément II chaîné en fin
         end if;
         B (II) := 0;  -- élément II déchaîné
         E (K)  := II; -- élément II placé en fin
      end loop;
      -- B est reconstitué à partir des sous-chaînes D
      M := 0;
      for P in 1 .. 10 loop
         if D (P) /= 0 then
            if M = 0 then
               Tete := D (P);
            else
               B (E (M))  := D (P);
            end if;
            M := P;
         end if;
      end loop;
   end loop;
   -- tableau trié reconstitué dans Y
   II := Tete;
   K  := 1;
   while II /= 0 loop
      Y (K) := X (II);
      II    := B (II);
      K     := K + 1;
   end loop;
   -- sortie dans X
   X := Y;
end TriBase10;

procedure Simulation_T is
   -- Nombre maximum d'évènements à un instant donné
   NMaxEv            : constant := 100;
   TempsService      : constant := 70.0; -- secondes
   IntervalleArrivee : constant := 30.0; -- secondes
```

```
      TempsFin          : constant := 1000.0; -- secondes
      type Evenement is (FinServ1, FinServ2, NouveauClient, FinSimul);
      IdentEv : array (1 .. NMaxEv) of Evenement;
      TempsEv : array (1 .. NMaxEv) of Float;
      -- Nombre d'évènements futurs, indice du prochain évènement
      NEv : Natural range 0 .. NMaxEv := 0;
      -- Temps simulé en secondes
      Temps : Float := 0.0;
      -- Longeurs des files d'attente
      L1, L2 : Natural range 0 .. 5 := 0;
      -- Identification des évènements successifs
      Ident : Evenement := NouveauClient;
      -- Indicateur de fin de simulation
      Fin : Boolean := False;
      type Bit32 is mod 65536;
      Memoire : Bit32 := 99;

      function Aleat return Float is
      begin
         Memoire := Memoire * 25173 + 13849;
         return Float (Memoire) / 65536.0 + 0.00001;
      end Aleat;
      function Poisson (Interval : Float) return Float is
      begin
         return -Interval * Ada.Numerics.Elementary_Functions.Log (Aleat);
      end Poisson;
      procedure Arrive (Temps : in out Float; Ident : in out Evenement) is
      begin
         if NEv /= 0 then
            Temps := TempsEv (NEv);
            Ident := IdentEv (NEv);
            NEv   := NEv - 1;
         else
            Put_Line ("Erreur : aucun évènement en attente.");
         end if;
      end Arrive;
      procedure Futur (Temps : Float; Ident : Evenement) is
         -- Insère un évènement à sa place
         I, J : Natural range 0 .. NMaxEv;
      begin
         if NEv < NMaxEv then
            -- Il y a encore au moins une place pour insérer
            -- au bon endroit recherché par dichotomie
            NEv := NEv + 1;
            J   := NEv;
            I   := 0;
            while J - I > 1 loop
               if Temps < TempsEv ((I + J) / 2) then
                  I := (I + J) / 2;
               else
                  J := (I + J) / 2;
               end if;
            end loop;
            if J < NEv then
               for K in reverse J + 1 .. NEv loop
                  TempsEv (K) := TempsEv (K - 1);
                  IdentEv (K) := IdentEv (K - 1);
               end loop;
            end if;
            TempsEv (J) := Temps;
            IdentEv (J) := Ident;
         else
            Put_Line ("Erreur : La suite des évènements est pleine.");
         end if;
      end Futur;
   begin
      -- Initialisation de la simulation
      Futur (TempsFin, FinSimul);
      Futur (Poisson (IntervalleArrivee), NouveauClient);
      -- Début de la simulation
      loop
         Arrive (Temps, Ident);
         case Ident is
```

```
            when FinServ1 =>
               L1 := L1 - 1;
               Put ("Fin de service chez le premier serveur à ");
               Put (Temps, 5, 2, 0);
               New_Line;
               Put_Line ("Longueur des queues 1 et 2 : " & L1'Img & L2'Img);
               if L1 > 0 then
                  Futur (Temps + Poisson (TempsService), FinServ1);
               end if;
            when FinServ2 =>
               L2 := L2 - 1;
               Put ("Fin de service chez le deuxième serveur à ");
               Put (Temps, 5, 2, 0);
               New_Line;
               Put_Line ("Longueur des queues 1 et 2 : " & L1'Img & L2'Img);
               if L2 > 0 then
                  Futur (Temps + Poisson (TempsService), FinServ2);
               end if;
            when NouveauClient =>
               Put ("Arrivée nouveau client à ");
               Put (Temps, 5, 2, 0);
               New_Line;
               Put_Line ("Longueur des queues 1 et 2 : " & L1'Img & L2'Img);
               if L1 <= 4 or else L2 <= 4 then
                  if L1 <= L2 then
                     Put_Line ("Le client choisi le premier serveur.");
                     if L1 = 0 then
                        Futur (Temps + Poisson (TempsService), FinServ1);
                     end if;
                     L1 := L1 + 1;
                  else
                     Put_Line ("Le client choisi le deuxième serveur.");
                     if L2 = 0 then
                        Futur (Temps + Poisson (TempsService), FinServ2);
                     end if;
                     L2 := L2 + 1;
                  end if;
               else
                  Put_Line ("Le client abandonne...");
               end if;
               Futur (Temps + Poisson (IntervalleArrivee), NouveauClient);
            when FinSimul =>
               Fin := True;
         end case;
         exit when Fin;
      end loop;
   end Simulation_T;

   package body HashCoding is
      -- Nombre premier immédiatement supérieur à N
      function Premier (N : Positive) return Positive is
         -- La conjecture de Legendre affirme qu'il existe un nombre premier entre n^2 et (n+1)^2
         -- Nous cherchons alors un nombre premier entre N+1 et (sqrt(N)+1)^2
         Max    : constant Positive           :=
            (Positive (Ada.Numerics.Elementary_Functions.Sqrt (Float (N))) + 1) ** 2;
         Crible : array (2 .. Max) of Boolean := (others => True);
         I      : Positive range 2 .. Max     := 2;
      begin
         while I * I <= Max loop
            if Crible (I) then
               for J in I + 1 .. Max loop
                  if J mod I = 0 then
                     Crible (J) := False;
                  end if;
               end loop;
            end if;
            I := I + 1;
         end loop;
         I := N + 1;
         while not Crible (I) loop
            I := I + 1;
         end loop;
         return I;
```

```
      end Premier;

      -- Taille de l'emplacement de stockage légèrement supérieur au nombre de valeurs
      C : constant Positive := Premier (4 * N / 3);
      -- Emplacements de stockage (0 valeur spéciale indiquant l'emplacement disponible)
      Z : TP (1 .. C) := (others => 0);
      -- Stockage des éléments
      S : array (1 .. C) of TElement;
      -- Fonction H de dispersion généralisée (hash coding)
      function H (E, P : Positive) return Positive is
         function H1 (E : Positive) return Positive is
         begin
            return E mod C + 1;
         end H1;
         function H2 (E : Positive) return Positive is
         begin
            return E mod (C - 1) + 1;
         end H2;
      begin
         case Dispersion is
            when Linéaire =>
               return (H1 (E) + P - 2) mod C + 1;
            when Optimum =>
               return (H1 (E) + P * H2 (E) - 2) mod C + 1;
            when Quadratique =>
               return (H1 (E) + P * P - 2) mod C + 1;
         end case;
      end H;
      procedure Stocke (Clé : Positive; Element : TElement) is
         P : Positive := 1;
      begin
         while Z (H (Clé, P)) /= 0 and then P < C loop
            P := P + 1;
            Put_Line ("Collision : " & P'Img & H (Clé, P)'Img);
         end loop;
         if Z (H (Clé, P)) = 0 then
            Z (H (Clé, P))  := Clé;
            S (H (Clé, P))  := Element;
         else
            raise Plein;
         end if;
      end Stocke;
      procedure Accede (Clé : Positive; Element : out TElement; Trouvé : out Boolean) is
         P : Positive := 1;
      begin
         while Z (H (Clé, P)) /= Clé and then P < C loop
            P := P + 1;
         end loop;
         if Z (H (Clé, P)) = Clé then
            Element := S (H (Clé, P));
            Trouvé  := True;
         else
            Trouvé := False;
         end if;
      end Accede;
      procedure Supprime (Clé : Positive; Trouvé : out Boolean) is
         P : Positive := 1;
      begin
         while Z (H (Clé, P)) /= Clé and then P < C loop
            P := P + 1;
         end loop;
         if Z (H (Clé, P)) = Clé then
            Z (H (Clé, P))  := 0;
            Trouvé          := True;
         else
            Trouvé := False;
         end if;
      end Supprime;
      procedure Imprime is
      begin
         Imprime (Z);
      end Imprime;
   end HashCoding;
```

```
procedure Intersection is
   -- M : constant := 500; -- nombre maximum d'éléments dans E et F
   -- C sensiblement supérieur à M, autour de 4 * M / 3 !
   C : constant := 683;
   subtype IndZ is Positive range 1 .. C;
   -- E et F ensembles dont on va faire l'intersection
   E : constant TP := (2, 30, 10, 23, 4, 33, 5);
   F : constant TP := (4, 32, 18, 30, 22, 45, 5, 33);
   -- Z ensemble résultat de l'intersection
   -- (0 valeur spéciale indiquant l'emplacement disponible)
   Z             : TP (IndZ) := (others => 0);
   P             : Natural range 0 .. C;
   J             : IndZ;
   Rangé, Trouvé : Boolean;
   function H (X : Positive; P : IndZ) return IndZ is
   begin
      return (X mod C + P * (X mod (C - 1) + 1) - 1) mod C + 1;
   end H;
begin
   -- les éléments de E sont rangés dans Z suivant la fonction de dispertion
   for I in E'Range loop
      -- P numéro d'essai effectué
      P     := 0;
      Rangé := False;
      loop
         P := P + 1;
         J := H (E (I), P);
         if Z (J) = 0 then
            Z (J) := E (I);
            Rangé := True;
         end if;
         exit when Rangé;
      end loop;
   end loop;
   -- chaque élément de F est recherché dans Z
   Put ("Intersection : ");
   for I in F'Range loop
      -- P numéro d'essai effectué
      P      := 0;
      Trouvé := False;
      loop
         P := P + 1;
         J := H (F (I), P);
         if Z (J) = F (I) then
            Put (F (I)'Img);
            Trouvé := True;
         end if;
         exit when Trouvé or Z (J) = 0;
      end loop;
   end loop;
   New_Line;
end Intersection;

procedure Union is
   -- E et F ensembles dont on va faire l'union
   E      : constant TP := (2, 30, 10, 23, 4, 33, 5);
   F      : constant TP := (4, 32, 18, 30, 22, 45, 5, 33);
   J      : Positive;
   Trouvé : Boolean;
   -- Recherche de l'existence par adressage dispersé avec clé d'entrée donnée par E
   package HashE is new HashCoding (E'Length, Positive, Linéaire);

begin
   -- les éléments de E sont rangés suivant la fonction de dispersion
   Put ("Union : ");
   for I in E'Range loop
      HashE.Stocke (E (I), E (I));
      Put (E (I)'Img);
   end loop;
   -- chaque élément de F est recherché
   for I in F'Range loop
      HashE.Accede (F (I), J, Trouvé);
```

```
         if not Trouvé then
            Put (F (I)'Img);
         end if;
      end loop;
      New_Line;
   end Union;

   function Dérivation (Entrée : String) return String is
      -- Dérivation formelle d'une expression parenthésée contenant :
      -- x ( ) 0..9 + - * / sans espace
      type Noeud;
      type PNoeud is access Noeud;
      type Noeud is record
         Val    : Character;
         C1, C2 : PNoeud;
      end record;
      subtype Ligne is Positive range 1 .. 80;
      Chaîne : String renames Entrée;
      K      : Ligne := 1;
      function Nouveau (NVal : Character; NC1, NC2 : PNoeud) return PNoeud is
      -- Créé un nouveau noeud dans l'arbre
      begin
         return new Noeud'(NVal, NC1, NC2);
      end Nouveau;
      function Edite (I : PNoeud) return String is
      -- Transforme en chaîne de caractères l'expression arborescente de racinne I
      begin
         if I.C1 /= null then
            return '(' & Edite (I.C1) & I.Val & Edite (I.C2) & ')';
         else
            return (1 => I.Val);
         end if;
      end Edite;
      function Construit return PNoeud is
         -- Construit l'arbre qui correspond à la sous-expression commençant
         -- au caractère K
         -- Fixe l'index K au caractère qui suit immédiatement cette sous-expression
         P1, P2, Res : PNoeud;
         Opérateur   : Character;
      begin
         if Chaîne (K) /= '(' then
            Res := Nouveau (Chaîne (K), null, null);
            K   := K + 1;
         else
            K         := K + 1;
            P1        := Construit;
            Opérateur := Chaîne (K);
            K         := K + 1;
            P2        := Construit;
            Res       := Nouveau (Opérateur, P1, P2);
            if Chaîne (K) /= ')' then
               Put_Line ("Erreur de ) !");
            end if;
            K := K + 1;
         end if;
         return Res;
      end Construit;
      function Copie (I : PNoeud) return PNoeud is
      -- Créé un arbre identique à celui de la racine I
      begin
         if I = null then
            return null;
         else
            return Nouveau (I.Val, Copie (I.C1), Copie (I.C2));
         end if;
      end Copie;
      function Dérive (I : PNoeud) return PNoeud is
      begin
         if I.Val = 'x' then
            return Nouveau ('1', null, null);
         elsif I.Val in '0' .. '9' then
            return Nouveau ('0', null, null);
         elsif I.Val = '+' or I.Val = '-' then
```

```
            return Nouveau (I.Val, Dérive (I.C1), Dérive (I.C2));
         elsif I.Val = '*' then
            return Nouveau
                     ('+',
                      Nouveau ('*', I.C1, Dérive (I.C2)),
                      Nouveau ('*', Dérive (I.C1), I.C2));
         elsif I.Val = '/' then
            return Nouveau
                     ('/',
                      Nouveau
                         ('-',
                          Nouveau ('*', Dérive (I.C1), I.C2),
                          Nouveau ('*', Dérive (I.C2), I.C1)),
                      Nouveau ('*', Copie (I.C2), Copie (I.C2)));
         else
            Put_Line ("Erreur '" & I.Val & "' symbole inconnu !");
            return null;
         end if;
      end Dérive;
   begin
      return Edite (Dérive (Construit));
   end Dérivation;

   function VersExpPostFixée (E : String) return String is
      -- Traduit une expression de nombres entiers en notation postfixée
      -- Opérations valides +, -, *, /, ^, (, )
      -- L'opérateur d'empilement d'une valeur est une opération ou ','
      use Ada.Characters.Handling;
      subtype TPriorité is Natural range 0 .. 3;
      P, F : String (1 .. E'Length * 2);
      -- P : pile intermédiaire
      -- F : résultat
      Car              : Character;
      IndE, IndF, IndP : Natural := 0;
      Opérande         : Boolean := False;
      function Priorité (Opérateur : Character) return TPriorité is
      begin
         case Opérateur is
            when '+' | '-' =>
               return 1;
            when '*' | '/' =>
               return 2;
            when '^' =>
               return 3;
            when others =>
               return 0;
         end case;
      end Priorité;
   begin
      IndE := E'First;
      while IndE <= E'Last or else IndP > 0 loop
         if IndE <= E'Last then
            Car := E (IndE);
         else
            -- caractère vide
            Car := '#';
         end if;
         if Is_Decimal_Digit (Car) then
            -- Car est opérande
            Opérande := True;
            IndF     := IndF + 1;
            F (IndF) := Car;
            IndE     := IndE + 1;
         elsif Car = '(' then
            -- empiler parenthèse
            IndP     := IndP + 1;
            P (IndP) := Car;
            IndE     := IndE + 1;
         elsif Priorité (Car) /= 0 and then (IndP = 0 or else P (IndP) = '(') then
            -- ajout opérateur d'empilement
            if Opérande then
               IndF     := IndF + 1;
               F (IndF) := ',';
```

```
            Opérande := False;
         end if;
         -- empiler opérateur
         IndP     := IndP + 1;
         P (IndP) := Car;
         IndE     := IndE + 1;
      elsif Priorité (Car) /= 0 and then Priorité (P (IndP)) /= 0 then
         if Priorité (Car) > Priorité (P (IndP)) then
            -- empiler opérateur
            IndP     := IndP + 1;
            P (IndP) := Car;
            IndE     := IndE + 1;
         else
            -- dépiler
            IndF     := IndF + 1;
            F (IndF) := P (IndP);
            IndP     := IndP - 1;
            -- on n'avance pas dans E
         end if;
      elsif (IndE > E'Last or else Car = ')') and then Priorité (P (IndP)) /= 0 then
         -- dépiler
         IndF     := IndF + 1;
         F (IndF) := P (IndP);
         IndP     := IndP - 1;
      -- on n'avance pas dans E
      elsif Car = ')' and then P (IndP) = '(' then
         -- dépiler parenthèse
         IndP := IndP - 1;
         IndE := IndE + 1;
      else
         Put_Line ("Erreur dans l'expression : " & Car'Img);
      end if;
   end loop;
   --Put_Line (E & ", " & P (1 .. IndP) & ", " & F (1 .. IndF));
   return F (1 .. IndF);
end VersExpPostFixée;

function ÉvalueExpPostFixée (E : String) return String is
   -- prendre pile.ads et adb !!!!
   package Valeurs is new Piles (Integer, E'Length, Integer'Image);
   Valeur  : Valeurs.Pile;
   N, U, V : Integer := 0;
   Nombre  : Boolean := False;
begin
   for I in E'Range loop
      case E (I) is
         when '0' .. '9' =>
            N      := N * 10 + Character'Pos (E (I)) - Character'Pos ('0');
            Nombre := True;
         when ',' =>
            Valeur.Empiler (N);
            N := 0;
         when '+' =>
            if Nombre then
               Valeur.Empiler (N);
               N      := 0;
               Nombre := False;
            end if;
            Valeur.Depiler (U);
            Valeur.Depiler (V);
            Valeur.Empiler (V + U);
         when '-' =>
            if Nombre then
               Valeur.Empiler (N);
               N      := 0;
               Nombre := False;
            end if;
            Valeur.Depiler (U);
            Valeur.Depiler (V);
            Valeur.Empiler (V - U);
         when '*' =>
            if Nombre then
               Valeur.Empiler (N);
```

```
                N      := 0;
                Nombre := False;
             end if;
             Valeur.Depiler (U);
             Valeur.Depiler (V);
             Valeur.Empiler (V * U);
          when '/' =>
             if Nombre then
                Valeur.Empiler (N);
                N      := 0;
                Nombre := False;
             end if;
             Valeur.Depiler (U);
             Valeur.Depiler (V);
             Valeur.Empiler (V / U);
          when '^' =>
             if Nombre then
                Valeur.Empiler (N);
                N      := 0;
                Nombre := False;
             end if;
             Valeur.Depiler (U);
             Valeur.Depiler (V);
             Valeur.Empiler (V ** U);
          when others =>
             Put_Line ("Erreur dans l'expression : " & E (I)'Img);
       end case;
    end loop;
    Valeur.Depiler (N);
    --Put_Line (E & " = " & N'Img);
    return Integer'Image (N);
 end ÉvalueExpPostFixée;

 procedure TriArborescent (V : in out TP) is
    R        : TP (V'Range);
    Vide     : constant := Integer'First;
    Sup      : constant := Integer'Last;
    J, J2, K : Natural;
    X, VV    : Integer;
    Y_A_Fils : Boolean;
 begin
    -- Ordonnancement de l'arbre verticalement
    -- présente toujours le plus petit élément au sommet
    for I in V'Range loop
       J := I;
       while J > 1 loop
          J2 := J / 2;
          if V (J) < V (J2) then
             VV     := V (J);
             V (J)  := V (J2);
             V (J2) := VV;
             J      := J2;
          else
             exit;
          end if;
          -- À ce stade la branche V(1) - V(I) est en ordre croissant
       end loop;
    end loop;
    for I in V'Range loop
       -- Sortie de la racine et remontée dans l'arbre
       R (I) := V (1);
       J     := 1;
       loop
       -- Chercher le plus petit fils s'il y en a
          Y_A_Fils := False;
          X        := Sup;
          for M in 2 * J .. 2 * J + 1 loop
             if M <= V'Last then
                if (V (M) < X) and then (V (M) /= Vide) then
                   K        := M;
                   X        := V (M);
                   Y_A_Fils := True;
                end if;
```

```
            end if;
         end loop;
         if Y_A_Fils then
            -- Remonter le plus petit fils et s'y positionner
            V (J) := X;
            J     := K;
         end if;
         exit when not Y_A_Fils;
      end loop;
      V (J) := Vide;
   end loop;
   V := R;
end TriArborescent;

package body Graphe_LF is
   -- Tableau de taille fixe augmentant avec le carré du nombre de noeuds
   -- Exige la définition d'un valeur d'arc représentant l'absence d'arc
   -- Les actions ajoute et supprime sont rapides
   subtype IndexNoeud is Positive range 1 .. Noeud'Pos (Noeud'Last) ** 2;
   TableauArcs : array (IndexNoeud) of Arcs := (others => Null_Arc);
   function Index (Origine, Extrémité : Noeud) return IndexNoeud is
   begin
      return Noeud'Pos (Noeud'Last) * (Noeud'Pos (Origine) - 1) + Noeud'Pos (Extrémité);
   end Index;
   procedure AjouteArc (Origine, Extrémité : Noeud; Arc : Arcs) is
   begin
      TableauArcs (Index (Origine, Extrémité))  := Arc;
   end AjouteArc;
   procedure SupprimeArc (Origine, Extrémité : Noeud) is
   begin
      TableauArcs (Index (Origine, Extrémité))  := Null_Arc;
   end SupprimeArc;
   procedure AfficheListeNoeudsExtrémité (Origine : Noeud) is
   begin
      for Extrémité in Noeud'Range loop
         if TableauArcs (Index (Origine, Extrémité)) /= Null_Arc then
            Put_Line
              ("Origine : " &
               Image (Origine) &
               ", Extrémité : " &
               Image (Extrémité) &
               ", arc : " &
               Image (TableauArcs (Index (Origine, Extrémité))));
         end if;
      end loop;
   end AfficheListeNoeudsExtrémité;
   procedure AfficheListeNoeudsOrigine (Extrémité : Noeud) is
   begin
      for Origine in Noeud'Range loop
         if TableauArcs (Index (Origine, Extrémité)) /= Null_Arc then
            Put_Line
              ("Origine : " &
               Image (Origine) &
               ", Extrémité : " &
               Image (Extrémité) &
               ", arc : " &
               Image (TableauArcs (Index (Origine, Extrémité))));
         end if;
      end loop;
   end AfficheListeNoeudsOrigine;
end Graphe_LF;

package body Graphe_LAC is
   -- Tableau de taille fixe déterminé à l'instanciation
   -- Les actions ajoute et supprime sont dépendantes d'une nombre d'arcs
   type Element is record
      Origine, Extrémité : Noeud;
      Arc                : Arcs;
   end record;
   subtype IndexElement is Natural range 0 .. MaxNoeuds;
   NombreElements  : IndexElement := 0;
   TableauElements : array (IndexElement range 1 .. MaxNoeuds) of Element;
   function RechecherArc (Origine, Extrémité : Noeud) return IndexElement is
```

```
      ElementCourant : IndexElement := NombreElements;
   begin
      while ElementCourant > 0
        and then (TableauElements (ElementCourant).Origine /= Origine
                  or else TableauElements (ElementCourant).Extrémité /= Extrémité)
      loop
         ElementCourant := ElementCourant - 1;
      end loop;
      return ElementCourant;
   end RechecherArc;
   procedure AjouteArc (Origine, Extrémité : Noeud; Arc : Arcs) is
      ElementTrouvé : constant IndexElement := RechecherArc (Origine, Extrémité);
   begin
      if ElementTrouvé = 0 then
         if NombreElements < MaxNoeuds then
            NombreElements                  := NombreElements + 1;
            TableauElements (NombreElements) := (Origine, Extrémité, Arc);
         else
            raise Plein;
         end if;
      else
         TableauElements (ElementTrouvé) := (Origine, Extrémité, Arc);
      end if;
   end AjouteArc;
   procedure SupprimeArc (Origine, Extrémité : Noeud) is
      ElementTrouvé : constant IndexElement := RechecherArc (Origine, Extrémité);
   begin
      if ElementTrouvé = 0 then
         raise NonTrouvé;
      end if;
      TableauElements (ElementTrouvé .. NombreElements - 1) :=
        TableauElements (ElementTrouvé + 1 .. NombreElements);
      NombreElements                                        := NombreElements - 1;
   end SupprimeArc;
   procedure AfficheListeNoeudsExtrémité (Origine : Noeud) is
   begin
      for ElementCourant in 1 .. NombreElements loop
         if TableauElements (ElementCourant).Origine = Origine then
            Put_Line
              ("Origine : " &
               Image (TableauElements (ElementCourant).Origine) &
               ", Extrémité : " &
               Image (TableauElements (ElementCourant).Extrémité) &
               ", arc : " &
               Image (TableauElements (ElementCourant).Arc));
         end if;
      end loop;
   end AfficheListeNoeudsExtrémité;
   procedure AfficheListeNoeudsOrigine (Extrémité : Noeud) is
   begin
      for ElementCourant in 1 .. NombreElements loop
         if TableauElements (ElementCourant).Extrémité = Extrémité then
            Put_Line
              ("Origine : " &
               Image (TableauElements (ElementCourant).Origine) &
               ", Extrémité : " &
               Image (TableauElements (ElementCourant).Extrémité) &
               ", arc : " &
               Image (TableauElements (ElementCourant).Arc));
         end if;
      end loop;
   end AfficheListeNoeudsOrigine;
end Graphe_LAC;

package body Graphe_LANC is
   -- Tableau de taille fixe déterminé à l'instanciation
   -- Exige la définition d'un valeur d'arc représentant l'absence d'arc
   -- Les actions ajoute et supprime sont inversement dépendantes d'une nombre d'arcs vides
   type Element is record
      Origine, Extrémité : Noeud;
      Arc                : Arcs := Null_Arc;
   end record;
   subtype IndexElement is Natural range 0 .. MaxNoeuds;
```

```
   TableauElements : array (IndexElement range 1 .. MaxNoeuds) of Element;
   function RechecherArc (Origine, Extrémité : Noeud) return IndexElement is
      ElementTrouvé : IndexElement := 0;
   begin
      for ElementCourant in TableauElements'Range loop
         if TableauElements (ElementCourant).Arc /= Null_Arc
           and then TableauElements (ElementCourant).Origine = Origine
           and then TableauElements (ElementCourant).Extrémité = Extrémité
         then
            ElementTrouvé := ElementCourant;
         end if;
      end loop;
      return ElementTrouvé;
   end RechecherArc;
   function RechecherVide return IndexElement is
      ElementTrouvé : IndexElement := 0;
   begin
      for ElementCourant in TableauElements'Range loop
         if TableauElements (ElementCourant).Arc = Null_Arc then
            ElementTrouvé := ElementCourant;
         end if;
      end loop;
      return ElementTrouvé;
   end RechecherVide;
   procedure AjouteArc (Origine, Extrémité : Noeud; Arc : Arcs) is
      ElementTrouvé : constant IndexElement := RechecherVide;
   begin
      if ElementTrouvé = 0 then
         raise Plein;
      else
         TableauElements (ElementTrouvé) := (Origine, Extrémité, Arc);
      end if;
   end AjouteArc;
   procedure SupprimeArc (Origine, Extrémité : Noeud) is
      ElementTrouvé : constant IndexElement := RechecherArc (Origine, Extrémité);
   begin
      if ElementTrouvé = 0 then
         raise NonTrouvé;
      end if;
      TableauElements (ElementTrouvé).Arc := Null_Arc;
   end SupprimeArc;
   procedure AfficheListeNoeudsExtrémité (Origine : Noeud) is
   begin
      for ElementCourant in TableauElements'Range loop
         if TableauElements (ElementCourant).Arc /= Null_Arc
           and then TableauElements (ElementCourant).Origine = Origine
         then
            Put_Line
              ("Origine : " &
               Image (TableauElements (ElementCourant).Origine) &
               ", Extrémité : " &
               Image (TableauElements (ElementCourant).Extrémité) &
               ", arc : " &
               Image (TableauElements (ElementCourant).Arc));
         end if;
      end loop;
   end AfficheListeNoeudsExtrémité;
   procedure AfficheListeNoeudsOrigine (Extrémité : Noeud) is
   begin
      for ElementCourant in TableauElements'Range loop
         if TableauElements (ElementCourant).Arc /= Null_Arc
           and then TableauElements (ElementCourant).Extrémité = Extrémité
         then
            Put_Line
              ("Origine : " &
               Image (TableauElements (ElementCourant).Origine) &
               ", Extrémité : " &
               Image (TableauElements (ElementCourant).Extrémité) &
               ", arc : " &
               Image (TableauElements (ElementCourant).Arc));
         end if;
      end loop;
   end AfficheListeNoeudsOrigine;
```

```
end Graphe_LANC;

package body Graphe_Matricielle is
   -- Tableau de taille fixe augmentant avec le carré du nombre de noeuds
   -- Exige la définition d'un valeur d'arc représentant l'absence d'arc
   -- Les actions ajoute et supprime sont rapides
   MatriceArcs : array (Noeud, Noeud) of Arcs := (others => (others => Null_Arc));
   MATRICE_ERROR : exception;
   procedure AjouteArc (Origine, Extrémité : Noeud; Arc : Arcs) is
   begin
      MatriceArcs (Origine, Extrémité) := Arc;
   end AjouteArc;
   procedure SupprimeArc (Origine, Extrémité : Noeud) is
   begin
      MatriceArcs (Origine, Extrémité) := Null_Arc;
   end SupprimeArc;
   procedure AfficheListeNoeudsExtrémité (Origine : Noeud) is
   begin
      for Extrémité in Noeud loop
         if MatriceArcs (Origine, Extrémité) /= Null_Arc then
            Put_Line
              ("Origine : " &
               Image (Origine) &
               ", Extrémité : " &
               Image (Extrémité) &
               ", arc : " &
               Image (MatriceArcs (Origine, Extrémité)));
         end if;
      end loop;
   end AfficheListeNoeudsExtrémité;
   procedure AfficheListeNoeudsOrigine (Extrémité : Noeud) is
   begin
      for Origine in Noeud loop
         if MatriceArcs (Origine, Extrémité) /= Null_Arc then
            Put_Line
              ("Origine : " &
               Image (Origine) &
               ", Extrémité : " &
               Image (Extrémité) &
               ", arc : " &
               Image (MatriceArcs (Origine, Extrémité)));
         end if;
      end loop;
   end AfficheListeNoeudsOrigine;
   function Retourne_Matrice_Incidence return Matrice is
   begin
      return M : Matrice do
         for Origine in Noeud loop
            for Extrémité in Noeud loop
               M (Origine, Extrémité) := not (MatriceArcs (Origine, Extrémité) = Null_Arc);
            end loop;
         end loop;
      end return;
   end Retourne_Matrice_Incidence;
   function "*" (Left, Right : Matrice) return Matrice is
   begin
      if Left'First (2) /= Right'First (1) or Left'Last (2) /= Right'Last (1) then
         raise MATRICE_ERROR;
      end if;
      return M : Matrice do
         for I in Left'Range (1) loop
            for J in Right'Range (2) loop
               M (I, J) := False;
               for K in Left'Range (2) loop
                  -- produit booléen des deux matrices
                  M (I, J) := M (I, J) or (Left (I, K) and Right (K, J));
               end loop;
            end loop;
         end loop;
      end return;
   end "*";
   function Retourne_Chemin (Longueur : Positive) return Matrice is
   begin
```

```
         return M : Matrice := Retourne_Matrice_Incidence do
            for I in 1 .. Longueur / 2 loop
               M := M * M;
            end loop;
            if Longueur rem 2 = 1 then
               M := M * Retourne_Matrice_Incidence;
            end if;
         end return;
      end Retourne_Chemin;
      procedure Affiche_Matrice (M : Matrice) is
         type Bin is range 0 .. 1;
         type TB is array (Boolean) of Bin;
         TabBin : constant TB := (0, 1);
      begin
         for Origine in Noeud loop
            for Extrémité in Noeud loop
               Put (TabBin (M (Origine, Extrémité))'Img);
            end loop;
            New_Line;
         end loop;
      end Affiche_Matrice;
   end Graphe_Matricielle;

   package body Graphe_Plexe is
      -- Liste chainée de taille augmentant avec le nombre de noeuds
      -- Les actions ajoute et supprime sont dépendantes d'une nombre d'arcs
      type Plexe;
      type PPlexe is access Plexe;
      type Plexe is record
         Origine   : Noeud;
         Extrémité : PPlexe := null;
         Arc       : Arcs;
         Frère     : PPlexe := null;
         Suivant   : PPlexe := null;
      end record;
      Premier : PPlexe := null;
      Courant : PPlexe := null;
      procedure Chaine (Noeud : PPlexe) is
      begin
         if Courant = null then
            Premier := Noeud;
            Courant := Noeud;
         else
            Courant.Suivant := Noeud;
            Courant         := Courant.Suivant;
         end if;
      end Chaine;
      function RechercheNoeud (UnNoeud : Noeud) return PPlexe is
         NoeudCourant : PPlexe := Premier;
      begin
         while NoeudCourant /= null and then NoeudCourant.Origine /= UnNoeud loop
            NoeudCourant := NoeudCourant.Suivant;
         end loop;
         return NoeudCourant;
      end RechercheNoeud;
      procedure AjouteArc (Origine, Extrémité : Noeud; Arc : Arcs) is
         NoeudOrigine   : PPlexe := RechercheNoeud (Origine);
         NoeudExtrémité : PPlexe := RechercheNoeud (Extrémité);
      begin
         if NoeudExtrémité = null then
            NoeudExtrémité         := new Plexe;
            NoeudExtrémité.Origine := Extrémité;
            Chaine (NoeudExtrémité);
         end if;
         if NoeudOrigine = null then
            NoeudOrigine := new Plexe'(Origine, NoeudExtrémité, Arc, null, null);
            Chaine (NoeudOrigine);
         else
            while NoeudOrigine.Frère /= null loop
               NoeudOrigine := NoeudOrigine.Frère;
            end loop;
            NoeudOrigine.Frère := new Plexe'(Origine, NoeudExtrémité, Arc, null, null);
            Chaine (NoeudOrigine.Frère);
```

```
      end if;
   end AjouteArc;
   procedure SupprimeArc (Origine, Extrémité : Noeud) is
      NoeudCourant : PPlexe := Premier;
      Trouvé       : PPlexe;
      procedure Libère is new Ada.Unchecked_Deallocation (Plexe, PPlexe);
   begin
      while NoeudCourant /= null loop
         if NoeudCourant.Extrémité /= null
           and then NoeudCourant.Origine = Origine
           and then NoeudCourant.Extrémité.Origine = Extrémité
         then
            NoeudCourant.Extrémité := null;
         end if;
         if NoeudCourant.Frère /= null
           and then NoeudCourant.Origine = Origine
           and then NoeudCourant.Frère.Extrémité.Origine = Extrémité
         then
            NoeudCourant.Frère := null;
         end if;
         if NoeudCourant.Suivant /= null
           and then NoeudCourant.Suivant.Origine = Extrémité
         then
            Trouvé               := NoeudCourant.Suivant;
            NoeudCourant.Suivant := NoeudCourant.Suivant.Suivant;
         end if;
         NoeudCourant := NoeudCourant.Suivant;
      end loop;
      if Trouvé = null then
         raise NonTrouvé;
      else
         Libère (Trouvé);
      end if;
   end SupprimeArc;
   procedure AfficheListeNoeudsExtrémité (Origine : Noeud) is
      NoeudOrigine : PPlexe := RechercheNoeud (Origine);
   begin
      if NoeudOrigine.Extrémité /= null then
         Put_Line
           ("Origine : " &
            Image (NoeudOrigine.Origine) &
            ", Extrémité : " &
            Image (NoeudOrigine.Extrémité.Origine) &
            ", arc : " &
            Image (NoeudOrigine.Arc));
      end if;
      while NoeudOrigine.Frère /= null loop
         Put_Line
           ("Origine : " &
            Image (Origine) &
            ", Extrémité : " &
            Image (NoeudOrigine.Frère.Extrémité.Origine) &
            ", arc : " &
            Image (NoeudOrigine.Frère.Arc));
         NoeudOrigine := NoeudOrigine.Frère;
      end loop;
   end AfficheListeNoeudsExtrémité;
   procedure AfficheListeNoeudsOrigine (Extrémité : Noeud) is
      NoeudCourant : PPlexe := Premier;
   begin
      while NoeudCourant /= null loop
         if NoeudCourant.Extrémité /= null
           and then NoeudCourant.Extrémité.Origine = Extrémité
         then
            Put_Line
              ("Origine : " &
               Image (NoeudCourant.Origine) &
               ", Extrémité : " &
               Image (NoeudCourant.Extrémité.Origine) &
               ", arc : " &
               Image (NoeudCourant.Arc));
         end if;
         NoeudCourant := NoeudCourant.Suivant;
```

```
      end loop;
   end AfficheListeNoeudsOrigine;
end Graphe_Plexe;

package body Graphe_Dispersé is
   type TCouple is record
      Origine, Extrémité : Noeud;
      Arc                : Arcs;
   end record;
   -- Stockage par adressage dispersé avec clé d'entrée donnée par l'origine et l'extrémité
   package Hash is new HashCoding (Noeud'Pos (Noeud'Last) ** 2, TCouple, Optimum);
   procedure AjouteArc (Origine, Extrémité : Noeud; Arc : Arcs) is
   begin
      Hash.Stocke
        (Noeud'Pos (Origine) * Noeud'Pos (Noeud'Last) + Noeud'Pos (Extrémité),
         (Origine, Extrémité, Arc));
   end AjouteArc;
   procedure SupprimeArc (Origine, Extrémité : Noeud) is
      Trouvé : Boolean;
      pragma Unreferenced (Trouvé);
   begin
      Hash.Supprime
        (Noeud'Pos (Origine) * Noeud'Pos (Noeud'Last) + Noeud'Pos (Extrémité),
         Trouvé);
   end SupprimeArc;
   procedure AfficheListeNoeudsExtrémité (Origine : Noeud) is
      Couple : TCouple;
      Trouvé : Boolean;
   begin
      for Extrémité in Noeud loop
         Hash.Accede
           (Noeud'Pos (Origine) * Noeud'Pos (Noeud'Last) + Noeud'Pos (Extrémité),
            Couple,
            Trouvé);
         if Trouvé then
            Put_Line
              ("Origine : " &
               Image (Couple.Origine) &
               ", Extrémité : " &
               Image (Couple.Extrémité) &
               ", arc : " &
               Image (Couple.Arc));
         end if;
      end loop;
   end AfficheListeNoeudsExtrémité;
   procedure AfficheListeNoeudsOrigine (Extrémité : Noeud) is
      Couple : TCouple;
      Trouvé : Boolean;
   begin
      for Origine in Noeud loop
         Hash.Accede
           (Noeud'Pos (Origine) * Noeud'Pos (Noeud'Last) + Noeud'Pos (Extrémité),
            Couple,
            Trouvé);
         if Trouvé then
            Put_Line
              ("Origine : " &
               Image (Couple.Origine) &
               ", Extrémité : " &
               Image (Couple.Extrémité) &
               ", arc : " &
               Image (Couple.Arc));
         end if;
      end loop;
   end AfficheListeNoeudsOrigine;
end Graphe_Dispersé;

package PQ is new Suite (Integer, 10, Integer'Image);
Q : PQ.TInstance;
I : Integer;

X, Y, V : TP := (25, 30, -4, 0, -10, 31, 4, -2, 26, -11, 1);
Z       : TP := (1 => 25);
```

```
   U       : TP := (79, 123, 468, 19, 256, 64, 27, 12, 842, 20, 122, 88, 54, 67, 0, 74);

   T : constant array (1 .. 8, 1 .. 2) of Positive :=
     ((2, 4),
      (2, 7),
      (3, 2),
      (4, 5),
      (6, 7),
      (7, 1),
      (7, 4),
      (7, 5));
   -- Stockage par adressage dispersé avec clé d'entrée donnée par une combinaison du couple T(i)
   package HashL is new HashCoding (T'Length (1), Positive, Linéaire);
   package HashO is new HashCoding (T'Length (1), Positive, Optimum);
   package HashQ is new HashCoding (T'Length (1), Positive, Quadratique);

   subtype Noeud is Character range 'A' .. 'D';
   package GraphLF is new Graphe_LF (Noeud, Integer, 0, Noeud'Image, Integer'Image);
   package GraphLAC is new Graphe_LAC (10, Noeud, Integer, Noeud'Image, Integer'Image);
   package GraphLANC is new Graphe_LANC (10, Noeud, Integer, 0, Noeud'Image, Integer'Image);
   package GraphMat is new Graphe_Matricielle (Noeud, Integer, 0, Noeud'Image, Integer'Image);
   package GraphPlexe is new Graphe_Plexe (Noeud, Integer, Noeud'Image, Integer'Image);
   package GraphDispersé is new Graphe_Dispersé (Noeud, Integer, Noeud'Image, Integer'Image);

   J  : Positive;
   OK : Boolean;
   T0 : Natural;

begin
   T0 := HorlogeMS;

   Put_Line ("Exemple 5.4 et exercice 5.1 :");
   Put ("Empilement et dépilement :");
   Q.Adjonction (4);
   Q.Suppression (I);
   Put_Line (I'Img);
   Q.Imprime;
   Q.Adjonction (11);
   Q.Adjonction (12);
   Q.Adjonction (13);
   Q.Adjonction (14);
   Q.Adjonction (15);
   Q.Adjonction (16);
   Q.Adjonction (17);
   Q.Adjonction (18);
   Q.Adjonction (19);
   Q.Adjonction (20);
   --Q.Adjonction (21);
   --PQ.Adjonction (O => Q,Element => 21);
   Q.Suppression (I);
   Q.Suppression (I);
   Q.Suppression (I);
   Q.Suppression (I);
   Q.Suppression (I);
   Q.Adjonction (28);
   Q.Adjonction (29);
   Q.Adjonction (30);
   Q.Imprime;

   Put_Line ("§5.2.5A et exercice 5.2 :");
   Put_Line ("Tableaux de nombres entiers triés par interclassement :");
   TriInterClassement (X);
   Imprime (X);
   TriInterClassement (Z);
   Imprime (Z);
   Put_Line ("§5.2.5B et exemple 5.7 :");
   Put_Line ("Tableaux de nombres entiers triés par segmentation :");
   TriSegmentation (Y);
   Imprime (Y);
   TriSegmentation (Z);
   Imprime (Z);
   Put_Line ("§5.3.4, exemple 5.10 et exercice 5.9 :");
   Put_Line ("Tableaux de nombres entiers triés par base 10 :");
```

```
   TriBase10 (U);
   Imprime (U);
   TriBase10 (Z);
   Imprime (Z);
   TriBase10 (V);
   Imprime (V);

   Put_Line ("Exemple 5.12 :");
   Put_Line ("Simulation d'un magasin à deux serveurs pendant 1000 secondes :");
   Simulation_T;

   Put_Line ("Exemple 6.9, 6.10 et exercices 6.1, 6.2 :");
   Put_Line ("Collisions avec rangement linéaire :");
   for k in T'Range (1) loop
      HashL.Stocke (T (k, 1) * T (k, 2) - 1, k);
   end loop;
   Put ("Rangement linéaire :");
   HashL.Imprime;
   Put_Line ("Collisions avec rangement optimum :");
   for k in T'Range (1) loop
      HashO.Stocke (T (k, 1) * T (k, 2) - 1, k);
   end loop;
   Put ("Rangement optimum:");
   HashO.Imprime;
   Put_Line ("Collisions avec rangement quadratique :");
   for k in T'Range (1) loop
      HashQ.Stocke (T (k, 1) * T (k, 2) - 1, k);
   end loop;
   Put ("Rangement quadratique :");
   HashQ.Imprime;
   HashQ.Accede (4 * 5 - 1, J, OK);
   if OK then
      Put_Line (J'Img);
   else
      Put_Line ("Non trouvé !");
   end if;
   HashQ.Supprime (4 * 5 - 1, OK);
   Put_Line (OK'Img);
   HashQ.Accede (4 * 5 - 1, J, OK);
   if OK then
      Put_Line (J'Img);
   else
      Put_Line ("Non trouvé !");
   end if;

   Put_Line ("Exemple 6.11 :");
   Intersection;

   Put_Line ("Exercice 6.3 :");
   Union;

   Put_Line ("Exemple 7.3 :");
   Put_Line ("Expression : (((x*x)+(2*x))+9)");
   Put ("Dérivée : ");
   Put_Line (Dérivation ("(((x*x)+(2*x))+9)"));

   Put_Line ("Exemple 7.4 et exercice 7.10 :");
   Put ("Vers expression post-fixée : 10+(23-309)*3/490 -> ");
   Put_Line (VersExpPostFixée ("10+(23-309)*3/490"));

   Put_Line ("Exemple 7.5 :");
   Put ("90,800-70,14*2+* = ");
   Put_Line (ÉvalueExpPostFixée ("90,800-70,14*2+*"));
   Put ("9,8+7,4*2+* = ");
   Put_Line (ÉvalueExpPostFixée ("9,8+7,4*2+*"));
   Put ("10+(23-309)*3/490 : ");
   Put_Line (ÉvalueExpPostFixée (VersExpPostFixée ("10+(23-309)*3/490")));

   Put_Line ("Exercice 7.12 :");
   Put_Line ("Tableaux de nombres entiers triés par arborescence :");
   TriArborescent (X);
   Imprime (X);
```

```
    Put_Line ("Exemple 8.6 1) :");
    Put_Line ("Graphe à représentation linéaire fonctionnelle :");
    GraphLF.AjouteArc ('A', 'B', 20);
    GraphLF.AjouteArc ('B', 'C', 50);
    GraphLF.AjouteArc ('B', 'D', 10);
    GraphLF.AjouteArc ('B', 'A', 100);
    GraphLF.AjouteArc ('C', 'D', 80);
    GraphLF.AjouteArc ('D', 'C', 30);
    GraphLF.SupprimeArc ('B', 'A');
    Put_Line ("Affiche la liste des noeuds Extrémité de B :");
    GraphLF.AfficheListeNoeudsExtrémité ('B');
    Put_Line ("Affiche la liste des noeuds origine de D :");
    GraphLF.AfficheListeNoeudsOrigine ('D');

    Put_Line ("Exemple 8.6 2) :");
    Put_Line ("Graphe à représentation associative contigüe :");
    GraphLAC.AjouteArc ('A', 'B', 20);
    GraphLAC.AjouteArc ('B', 'C', 50);
    GraphLAC.AjouteArc ('B', 'D', 10);
    GraphLAC.AjouteArc ('B', 'A', 100);
    GraphLAC.AjouteArc ('C', 'D', 80);
    GraphLAC.AjouteArc ('D', 'C', 30);
    GraphLAC.SupprimeArc ('B', 'A');
    Put_Line ("Affiche la liste des noeuds Extrémité de B :");
    GraphLAC.AfficheListeNoeudsExtrémité ('B');
    Put_Line ("Affiche la liste des noeuds origine de D :");
    GraphLAC.AfficheListeNoeudsOrigine ('D');

    Put_Line ("Exemple 8.6 2) modifié :");
    Put_Line ("Graphe à représentation associative non contigüe :");
    GraphLANC.AjouteArc ('A', 'B', 20);
    GraphLANC.AjouteArc ('B', 'C', 50);
    GraphLANC.AjouteArc ('B', 'D', 10);
    GraphLANC.AjouteArc ('B', 'A', 100);
    GraphLANC.AjouteArc ('C', 'D', 80);
    GraphLANC.AjouteArc ('D', 'C', 30);
    GraphLANC.SupprimeArc ('B', 'A');
    Put_Line ("Affiche la liste des noeuds Extrémité de B :");
    GraphLANC.AfficheListeNoeudsExtrémité ('B');
    Put_Line ("Affiche la liste des noeuds origine de D :");
    GraphLANC.AfficheListeNoeudsOrigine ('D');

    Put_Line ("§8.2.2 :");
    Put_Line ("Graphe à représentation matricielle :");
    GraphMat.AjouteArc ('A', 'B', 20);
    GraphMat.AjouteArc ('B', 'C', 50);
    GraphMat.AjouteArc ('B', 'D', 10);
    GraphMat.AjouteArc ('B', 'A', 100);
    GraphMat.AjouteArc ('C', 'D', 80);
    GraphMat.AjouteArc ('D', 'C', 30);
    GraphMat.SupprimeArc ('B', 'A');
    Put_Line ("Affiche la liste des noeuds Extrémité de B :");
    GraphMat.AfficheListeNoeudsExtrémité ('B');
    Put_Line ("Affiche la liste des noeuds origine de D :");
    GraphMat.AfficheListeNoeudsOrigine ('D');

    Put_Line ("Exercice 8.1 :");
    Put_Line ("Graphe à représentation par plexes :");
    GraphPlexe.AjouteArc ('A', 'B', 20);
    GraphPlexe.AjouteArc ('B', 'C', 50);
    GraphPlexe.AjouteArc ('B', 'D', 10);
    GraphPlexe.AjouteArc ('B', 'A', 100);
    GraphPlexe.AjouteArc ('C', 'D', 80);
    GraphPlexe.AjouteArc ('D', 'C', 30);
    GraphPlexe.SupprimeArc ('B', 'A');
    Put_Line ("Affiche la liste des noeuds Extrémité de B :");
    GraphPlexe.AfficheListeNoeudsExtrémité ('B');
    Put_Line ("Affiche la liste des noeuds origine de D :");
    GraphPlexe.AfficheListeNoeudsOrigine ('D');

    Put_Line ("Exemple 8.8 :");
    Put_Line ("Graphe à représentation par adressage dispersé :");
    GraphDispersé.AjouteArc ('A', 'B', 20);
```

```
    GraphDispersé.AjouteArc ('B', 'C', 50);
    GraphDispersé.AjouteArc ('B', 'D', 10);
    GraphDispersé.AjouteArc ('B', 'A', 100);
    GraphDispersé.AjouteArc ('C', 'D', 80);
    GraphDispersé.AjouteArc ('D', 'C', 30);
    GraphDispersé.SupprimeArc ('B', 'A');
    GraphDispersé.AjouteArc ('D', 'A', 60);
    GraphDispersé.AjouteArc ('D', 'B', 70);
    GraphDispersé.AjouteArc ('C', 'A', 40);
    Put_Line ("Affiche la liste des noeuds Extrémité de B :");
    GraphDispersé.AfficheListeNoeudsExtrémité ('B');
    Put_Line ("Affiche la liste des noeuds origine de D :");
    GraphDispersé.AfficheListeNoeudsOrigine ('D');

    Put_Line ("Exemple 8.9 :");
    GraphMat.AjouteArc ('D', 'A', 60);
    GraphMat.AjouteArc ('D', 'B', 70);
    GraphMat.AjouteArc ('C', 'A', 40);
    Put_Line ("Affiche la matrice incidente :");
    GraphMat.Affiche_Matrice (GraphMat.Retourne_Matrice_Incidence);
    Put_Line ("Affiche chemin de longueur 2 :");
    GraphMat.Affiche_Matrice (GraphMat.Retourne_Chemin (Longueur => 2));

    Put_Line ("Temps d'exécution :" & Natural'Image (HorlogeMS - T0) & " millisecondes");
end prog_avan2;
```