```
-------------------------------------------------------------------------------
-- NOM DU CSU (pricipal)            : Prog_Avan.adb
-- AUTEUR DU CSU                    : Pascal Pignard
-- VERSION DU CSU                   : 1.7a
-- DATE DE LA DERNIERE MISE A JOUR  : 15 avril 2011
-- 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 2008-2011
-- 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.Calendar;

procedure prog_avan is

   -- Déclarations utilitaires :

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

   -- Affiche les éléments du tableau T
   procedure Imprime (T : TP) is
   begin
      for I in T'Range loop
         Put (T (I)'Img);
      end loop;
      New_Line;
   end Imprime;

   -- Renvoie le compteur horaire interne en milisecondes.
   function HorlogeMS return Natural is
   begin
      return Natural (Ada.Calendar.Seconds (Ada.Calendar.Clock) * 1000.0);
   end HorlogeMS;

   -- Retourne les nombres premiers inférieurs à N
   function NombresPremiers (N : Positive) return TP;

   -- Échange de 2 valeurs d'une nature quelconque (non limitée)
   generic
      type Item is private;
   procedure Swap (A, B : in out Item);

   -- Les exemples et exercices du livre :
   -- (le signe ∨ signifie "ou", ∧ signifie "et", ¬ signifie "non")
```

```
-- Retourne le Plus Grand Commun Diviseur
-- Exemple 1.1
function PGCD (X, Y : Positive) return Positive;

-- Tri à bulle d'un tableau d'une taille et d'index quelconques
-- remplie de valeurs de nature quelconque (non limitée)
-- ayant une opérartion d'ordre quelconque
-- Exercice 1.4
generic
   type Item is private;
   type Index is (<>);
   type Row is array (Index range <>) of Item;
   with function "<" (X, Y : Item) return Boolean is <>;
procedure BubbleSort (A : in out Row);

-- Retourne de grandes valeurs de factorielle (jusqu'à 2000!)
-- Exemple 2.6
function BigFact (N : Natural) return TP;

-- Puissance n-ième d'un nombre réel (itératif et récurssif)
-- Exercice 2.9
function PowerI (X : Float; N : Natural) return Float;
function PowerR (X : Float; N : Natural) return Float;

-- Parcours un arbre définit par les éléments fils et frère
-- Exemple 3.12
procedure ParcoursArbre
  (N           : Positive;
   DefiniFils  : not null access function (I : Natural; P : TP) return Positive;
   DefiniFrere : not null access function (I : Natural; P : TP) return Positive;
   ExisteFils  : not null access function (I : Natural; P : TP) return Boolean;
   ExisteFrere : not null access function (I : Natural; P : TP) return Boolean);

-- Affiche les n-uplets de valeur 1 à k
-- Exemple 3.13
procedure N_Uplets (N, K : Positive);

-- Parcours partiellement un arbre définit par
-- les éléments accepte, fils et frère
-- § 3.3.2
procedure ParcoursArbrePartiel
  (N           : Positive;
   Accepte     : not null access function (I : Natural; P : TP) return Boolean;
   DefiniFils  : not null access function (I : Natural; P : TP) return Natural;
   DefiniFrere : not null access function (I : Natural; P : TP) return Natural;
   ExisteFils  : not null access function (I : Natural; P : TP) return Boolean;
   ExisteFrere : not null access function (I : Natural; P : TP) return Boolean);

-- Affiche les permutations de n objets
-- Exemple 3.15 et exercice 3.10
procedure Permutations (N : Positive);

-- Coloriage planaire avec quatre couleurs
-- Exercice 3.11
procedure ColoriagePlanaire;

-- Parcours partiellement avec la méthode du MinMax un arbre définit par
-- les éléments accepte, évalutatin MinMax, fils et frère
-- Exemple 3.16
procedure ParcoursArbreMinMax
```

```
   (N           : Positive;
    Accepte     : not null access function (I : Natural; P : TP) return Boolean;
    EvalMinMax  : not null access function (I : Natural; P : TP) return Integer;
    DefiniFils  : not null access function (I : Natural; P : TP) return Positive;
    DefiniFrere : not null access function (I : Natural; P : TP) return Positive;
    ExisteFils  : not null access function (I : Natural; P : TP) return Boolean;
    ExisteFrere : not null access function (I : Natural; P : TP) return Boolean);

-- Affiche le meilleur coup et le gain pour le
-- jeu du plus grand diviseur premier basé sur la construction
-- de nombres de n chiffres de 1 à k
-- Exemple 3.17 et exercice 3.14
procedure PlusGrandDiviseurPremier (N, K : Positive);

-- Parcours partiellement avec la méthode du MinMax avec coupure AlphaBeta
-- un arbre définit par les éléments accepte, évalutatin MinMax, fils et frère
-- Exemple 3.18
procedure ParcoursArbreAlphaBeta
  (N           : Positive;
   Accepte     : not null access function (I : Natural; P : TP) return Boolean;
   EvalMinMax  : not null access function (I : Natural; P : TP) return Integer;
   DefiniFils  : not null access function (I : Natural; P : TP) return Positive;
   DefiniFrere : not null access function (I : Natural; P : TP) return Positive;
   ExisteFils  : not null access function (I : Natural; P : TP) return Boolean;
   ExisteFrere : not null access function (I : Natural; P : TP) return Boolean);

-- Affiche la solution d'une équation crypto-arithmétique
-- Exercice 3.16
procedure Crypto;

-- Optimisation du remplisage d'un sac avec divers objets
-- de tailles, de poids et d'importances différents
-- Exercice 3.17
procedure Knapsack;

-- Calcul de factorielle n et du nombre de combinaison de n objets pris p à p
-- Nombre de parties de cardinal p d'un ensemble cardinal n
-- (itératif, récurssif et à l'aide de factorielleÒ)
-- Exemple 4.2 et exercice 4.1
function FactorielleI (N : Natural) return Positive;
function FactorielleR (N : Natural) return Positive;
function CombinaisonI (N : Positive; P : Natural) return Positive;
function CombinaisonR (N : Positive; P : Natural) return Positive;
function CombinaisonF (N : Positive; P : Natural) return Positive;

-- Palindrome (symétérique par rapport à son milieux)
-- et Palindrome par morceaux (conceténation de plusieurs palindromes)
-- Exercice 4.5
function Palindrome (Ch : String) return Boolean;
function PalindromeParMorceau (Ch : String) return Boolean;

-- Le code des procédures et fonctions :

function PGCD (X, Y : Positive) return Positive is
   A : Positive := Positive'Max (X, Y);
   B : Positive := Positive'Min (X, Y);
   C : Natural;
begin
   loop
      C := A mod B;
      exit when C = 0;
```

```
      A := B;
      B := C;
   end loop;
   return B;
end PGCD;

procedure Swap (A, B : in out Item) is
   C : constant Item := A;
begin
   A := B;
   B := C;
end Swap;

procedure BubbleSort (A : in out Row) is
   procedure SwapItem is new Swap (Item);
   J : Index;
begin
   for I in A'Range loop
      J := I;
      while (J /= A'First) and then A (J) < A (Index'Pred (J)) loop
         SwapItem (A (J), A (Index'Pred (J)));
         J := Index'Pred (J);
      end loop;
   end loop;
end BubbleSort;

function BigFact (N : Natural) return TP is
   B    : constant := 1000000; -- base de calcul
   B1   : constant := 999999;
   MaxN : constant := 2000; -- MaxN * B < Integer'Last
   MaxV : constant := 950;  -- B**V > MaxN! (10**(6*V) > 10**5700)

   subtype TV is Positive range 1 .. MaxV;
   subtype TB1 is Natural range 0 .. B1;

   K     : TV;
   P     : TB1;
   V     : array (TV) of Natural := (others => 0);
   Inter : Natural;

   pragma Assert (N <= MaxN, "Number too big!");

begin
   V (MaxV) := 1;
   K        := 1;
   for M in 2 .. N loop
      P := 0;
      for R in 1 .. K loop
         Inter            := P + M * V (MaxV + 1 - R);
         V (MaxV + 1 - R) := Inter mod B;
         P                := Inter / B;
      end loop;
      if P /= 0 then
         V (MaxV - K) := P;
         K            := K + 1;
      end if;
   end loop;
   return TP (V (MaxV + 1 - K .. MaxV));
end BigFact;

function PowerI (X : Float; N : Natural) return Float is
```

```
      Y  : Float   := 1.0;
      Z  : Float   := X;
      NN : Natural := N;
   begin
      while NN /= 0 loop
         if NN mod 2 = 1 then
            Y := Y * Z;
         end if;
         NN := NN / 2;
         Z  := Z * Z;
      end loop;
      return Y;
   end PowerI;

   function PowerR (X : Float; N : Natural) return Float is
      Y : Float;
   begin
      if N = 0 then
         return 1.0;
      end if;
      Y := PowerR (X, N / 2);
      if N mod 2 = 0 then
         return Y * Y;
      else
         return Y * Y * X;
      end if;
   end PowerR;

   procedure ParcoursArbre
     (N           : Positive;
      DefiniFils  : not null access function (I : Natural; P : TP) return Positive;
      DefiniFrere : not null access function (I : Natural; P : TP) return Positive;
      ExisteFils  : not null access function (I : Natural; P : TP) return Boolean;
      ExisteFrere : not null access function (I : Natural; P : TP) return Boolean)
   is
      P : TP (1 .. N);
      I : Natural range 0 .. N := 0;

   begin
      loop
         if ExisteFils (I, P) then
            I     := I + 1;
            P (I) := DefiniFils (I, P);
         else
            Put ('(');
            for J in P'Range loop
               Put (P (J)'Img);
            end loop;
            Put (')');
            while I /= 0 and then not ExisteFrere (I, P) loop
               I := I - 1; --  remonte au père
            end loop;
            if I /= 0 then
               P (I) := DefiniFrere (I, P); --  créé frère
            end if;
         end if;
         exit when I = 0;
      end loop;
      New_Line;
   end ParcoursArbre;
```

```
procedure ParcoursArbrePartiel
  (N           : Positive;
   Accepte     : not null access function (I : Natural; P : TP) return Boolean;
   DefiniFils  : not null access function (I : Natural; P : TP) return Natural;
   DefiniFrere : not null access function (I : Natural; P : TP) return Natural;
   ExisteFils  : not null access function (I : Natural; P : TP) return Boolean;
   ExisteFrere : not null access function (I : Natural; P : TP) return Boolean)
is
   P         : TP (1 .. N);
   I         : Natural range 0 .. N := 0;
   Term, Acc : Boolean;

begin
   loop
      Term := not ExisteFils (I, P);
      Acc  := Accepte (I, P);
      if Acc then
         if Term then
            Put ('(');
            for J in P'Range loop
               Put (P (J)'Img);
            end loop;
            Put (')');
         else
            I     := I + 1;
            P (I) := DefiniFils (I, P);
         end if;
      end if;
      if Term or else not Acc then
         while I /= 0 and then not ExisteFrere (I, P) loop
            I := I - 1; --  remonte au père
         end loop;
         if I /= 0 then
            P (I) := DefiniFrere (I, P); --  créé frère
         end if;
      end if;
      exit when I = 0;
   end loop;
   New_Line;
end ParcoursArbrePartiel;

procedure ParcoursArbreMinMax
  (N           : Positive;
   Accepte     : not null access function (I : Natural; P : TP) return Boolean;
   EvalMinMax  : not null access function (I : Natural; P : TP) return Integer;
   DefiniFils  : not null access function (I : Natural; P : TP) return Positive;
   DefiniFrere : not null access function (I : Natural; P : TP) return Positive;
   ExisteFils  : not null access function (I : Natural; P : TP) return Boolean;
   ExisteFrere : not null access function (I : Natural; P : TP) return Boolean)
is
   P         : TP (1 .. N);
   V         : array (Natural range 0 .. N) of Integer;
   I         : Natural range 0 .. N := 0;
   C         : Natural              := 0;
   Term, Acc : Boolean;

begin
   loop
      Term := not ExisteFils (I, P);
      Acc  := Accepte (I, P);
      if Acc then
```

```
         if Term then
            -- feuille
            V (I)     := EvalMinMax (I, P);
            V (I - 1) := Positive'Max (V (I), V (I - 1));
         else
            -- non terminal en préordre
            V (I) := Positive'Last * (-1) ** I;
            I     := I + 1;
            P (I) := DefiniFils (I, P);
         end if;
      end if;
      if Term or else not Acc then
         -- postordre sur les feuilles
         while I /= 0 and then not ExisteFrere (I, P) loop
            I := I - 1; --  remonte au père
            -- non terminal en postordre
            if I /= 0 then
               if I mod 2 = 0 then
                  -- non terminal i pair
                  V (I - 1) := Positive'Max (V (I), V (I - 1));
               else
                  -- non terminal i impair
                  V (I - 1) := Positive'Min (V (I), V (I - 1));
                  if I = 1 and then V (1) = V (0) then
                     C := P (1);
                  end if;
               end if;
            end if;
         end loop;
         if I /= 0 then
            P (I) := DefiniFrere (I, P); --  créé frère
         end if;
      end if;
      exit when I = 0;
   end loop;
   Put_Line (C'Img & V (0)'Img);
end ParcoursArbreMinMax;

procedure ParcoursArbreAlphaBeta
  (N           : Positive;
   Accepte     : not null access function (I : Natural; P : TP) return Boolean;
   EvalMinMax  : not null access function (I : Natural; P : TP) return Integer;
   DefiniFils  : not null access function (I : Natural; P : TP) return Positive;
   DefiniFrere : not null access function (I : Natural; P : TP) return Positive;
   ExisteFils  : not null access function (I : Natural; P : TP) return Boolean;
   ExisteFrere : not null access function (I : Natural; P : TP) return Boolean)
is
   P         : TP (1 .. N);
   V         : array (Natural range 0 .. N) of Integer;
   I         : Natural range 0 .. N := 0;
   C         : Natural              := 0;
   Term, Acc : Boolean;

begin
   loop
      Term := not ExisteFils (I, P);
      Acc  := Accepte (I, P);
      if Acc then
         if Term then
            -- feuille
            V (I) := EvalMinMax (I, P);
```

```
            if V (I) < V (I - 2) then
               V (I - 1) := Positive'Max (V (I), V (I - 1));
            else
               -- coupure Beta
               I   := I - 1;
               Acc := False;
            end if;
         else
            -- non terminal en préordre
            V (I) := Positive'Last * (-1) ** I;
            I     := I + 1;
            P (I) := DefiniFils (I, P);
         end if;
      end if;
      if Term or else not Acc then
         -- postordre sur les feuilles
         while I /= 0 and then not ExisteFrere (I, P) loop
            I := I - 1; --  remonte au père
            -- non terminal en postordre
            if I /= 0 then
               if I mod 2 = 0 then
                  -- non terminal i pair
                  if V (I) < V (I - 2) then
                     V (I - 1) := Positive'Max (V (I), V (I - 1));
                  else
                     -- coupure Beta
                     I := I - 1;
                  end if;
               else
                  -- non terminal i impair
                  if I >= 3 and then V (I) <= V (I - 2) then
                     -- coupure Alpha
                     I := I - 1;
                  end if;
                  if I = 1 or else (I >= 3 and then V (I) > V (I - 2)) then
                     V (I - 1) := Positive'Min (V (I), V (I - 1));
                     if I = 1 and then V (1) = V (0) then
                        C := P (1);
                     end if;
                  end if;
               end if;
            end if;
         end loop;
         if I /= 0 then
            P (I) := DefiniFrere (I, P); --  créé frère
         end if;
      end if;
      exit when I = 0;
   end loop;
   Put_Line (C'Img & V (0)'Img);
end ParcoursArbreAlphaBeta;

procedure N_Uplets (N, K : Positive) is
   Max : constant := 10;
   pragma Assert (N <= Max, "Number N too big!");
   pragma Assert (K <= Max, "Number K too big!");

   function DefiniFils (I : Natural; P : TP) return Positive is
   begin
      return 1;
   end DefiniFils;
```

```
      function DefiniFrere (I : Natural; P : TP) return Positive is
      begin
         return P (I) + 1;
      end DefiniFrere;
      function ExisteFils (I : Natural; P : TP) return Boolean is
      begin
         return I < N;
      end ExisteFils;
      function ExisteFrere (I : Natural; P : TP) return Boolean is
      begin
         return P (I) < K;
      end ExisteFrere;

   begin
      Put_Line
        (Positive'Image (K ** N) &
         " n-uplets de" &
         N'Img &
         " éléments de valeur de 1 à" &
         K'Img &
         " :");
      ParcoursArbre
        (N,
         DefiniFils'Access,
         DefiniFrere'Access,
         ExisteFils'Access,
         ExisteFrere'Access);
   end N_Uplets;

   procedure Permutations (N : Positive) is
      Max : constant := 10;
      pragma Assert (N <= Max, "Number N too big!");

      function Accepte (I : Natural; P : TP) return Boolean is
         Acc : Boolean := True;
      begin
         if I > 1 then
            for J in 1 .. I - 1 loop
               if P (I) = P (J) then
                  Acc := False;
               end if;
            end loop;
         end if;
         return Acc;
      end Accepte;
      function DefiniFils (I : Natural; P : TP) return Natural is
      begin
         return 1;
      end DefiniFils;
      function DefiniFrere (I : Natural; P : TP) return Natural is
      begin
         return P (I) + 1;
      end DefiniFrere;
      function ExisteFils (I : Natural; P : TP) return Boolean is
      begin
         return I < N;
      end ExisteFils;
      function ExisteFrere (I : Natural; P : TP) return Boolean is
      begin
         return P (I) < N;
      end ExisteFrere;
```

```
begin
   Put ("Nombres de permutations de" & N'Img & " éléments :");
   Imprime (BigFact (N));
   ParcoursArbrePartiel
     (N,
      Accepte'Access,
      DefiniFils'Access,
      DefiniFrere'Access,
      ExisteFils'Access,
      ExisteFrere'Access);
end Permutations;

procedure ColoriagePlanaire is
   NombreRégions : constant := 6;
   subtype Index is Positive range 1 .. NombreRégions;
   type Carte is array (Index, Index) of Boolean;
   A : Carte := (others => (others => False));
   procedure EnContact (Origine : Index; Avec : TP) is
   begin
      for I in Avec'Range loop
         A (Origine, Avec (I))  := True;
      end loop;
   end EnContact;
   function Accepte (I : Natural; P : TP) return Boolean is
      Acc : Boolean := True;
   begin
      if I > 1 then
         for J in 1 .. I - 1 loop
            -- merci à Jean-Claude Daudin pour la correction du test
            if A (I, J) and then P (I) = P (J) then
               Acc := False;
            end if;
         end loop;
      end if;
      return Acc;
   end Accepte;
   function DefiniFils (I : Natural; P : TP) return Natural is
   begin
      return 1;
   end DefiniFils;
   function DefiniFrere (I : Natural; P : TP) return Natural is
   begin
      return P (I) + 1;
   end DefiniFrere;
   function ExisteFils (I : Natural; P : TP) return Boolean is
   begin
      return I < NombreRégions;
   end ExisteFils;
   function ExisteFrere (I : Natural; P : TP) return Boolean is
   begin
      return P (I) < 4; -- quatre couleurs
   end ExisteFrere;

begin
   -- Définition des 6 régions à colorier
   EnContact (1, (1 => 2));
   EnContact (2, (1, 3, 4));
   EnContact (3, (2, 4, 5, 6));
   EnContact (4, (2, 3, 6));
   EnContact (5, (3, 6));
```

```
      EnContact (6, (3, 4, 5));
      Put_Line ("Solutions avec quatre couleurs :");
      ParcoursArbrePartiel
        (NombreRégions,
         Accepte'Access,
         DefiniFils'Access,
         DefiniFrere'Access,
         ExisteFils'Access,
         ExisteFrere'Access);
   end ColoriagePlanaire;

   function NombresPremiers (N : Positive) return TP is
      -- Crible d'Ératosthène
      pragma Assert (N /= 1, "1 is not a prime number!");
      type PTP is access TP;
      Crible : array (2 .. N) of Boolean := (others => True);
      Res    : PTP                       := null;
      I      : Positive                  := 2;
   begin
      while I * I <= N loop
         if Crible (I) then
            if Res = null then
               Res := new TP'(Positive'First => I);
            else
               Res := new TP'(Res (Res'Range) & I);
            end if;
            for J in I + 1 .. N loop
               if J mod I = 0 then
                  Crible (J) := False;
               end if;
            end loop;
         end if;
         I := I + 1;
      end loop;
      for K in I .. N loop
         if Crible (K) then
            if Res = null then
               Res := new TP'(Positive'First => K);
            else
               Res := new TP'(Res (Res'Range) & K);
            end if;
         end if;
      end loop;
      return Res.all;
   end NombresPremiers;

   procedure PlusGrandDiviseurPremier (N, K : Positive) is
      Max : constant := 10;
      pragma Assert (N <= Max, "Number N too big!");
      pragma Assert (K <= Max, "Number K too big!");
      Q : constant TP := NombresPremiers (10 ** N);
      S : array (Natural range 0 .. N) of Natural;

      function Accepte (I : Natural; P : TP) return Boolean is
      begin
         if I = 0 then
            S (0) := 0;
         else
            S (I) := 10 * S (I - 1) + P (I);
         end if;
         return True;
```

```
      end Accepte;
      function EvalMinMax (I : Natural; P : TP) return Integer is
         J   : Positive := 1;
         Res : Positive;
      begin
         loop
            if S (I) mod Q (J) = 0 then
               Res := Q (J);
            end if;
            J := J + 1;
            exit when Q (J) > S (I);
         end loop;
         return Res;
      end EvalMinMax;
      function DefiniFils (I : Natural; P : TP) return Positive is
      begin
         return 1;
      end DefiniFils;
      function DefiniFrere (I : Natural; P : TP) return Positive is
      begin
         return P (I) + 1;
      end DefiniFrere;
      function ExisteFils (I : Natural; P : TP) return Boolean is
      begin
         return I < N;
      end ExisteFils;
      function ExisteFrere (I : Natural; P : TP) return Boolean is
      begin
         return P (I) < K;
      end ExisteFrere;

   begin
      Put ("Jeu du Plus Grand Diviseur Premier MinMax (1er coup gagnant, gain) :");
      ParcoursArbreMinMax
        (N,
         Accepte'Access,
         EvalMinMax'Access,
         DefiniFils'Access,
         DefiniFrere'Access,
         ExisteFils'Access,
         ExisteFrere'Access);
      Put ("Jeu du Plus Grand Diviseur Premier AlphaBeta (1er coup gagnant, gain) :");
      ParcoursArbreAlphaBeta
        (N,
         Accepte'Access,
         EvalMinMax'Access,
         DefiniFils'Access,
         DefiniFrere'Access,
         ExisteFils'Access,
         ExisteFrere'Access);
   end PlusGrandDiviseurPremier;

   procedure Crypto is
      -- Equation : AB + BA = CAC
      -- avec P(1)=A, P(2)=B, ...
      -- 3 chiffres de 0 à 9
      N : constant := 3;
      K : constant := 9;
      function Accepte (I : Natural; P : TP) return Boolean is
         Acc : Boolean := True;
      begin
```

```
      for J in 1 .. I - 1 loop
         if P (I) = P (J) then
            Acc := False;
         end if;
      end loop;
      if Acc and then I = N then
         -- l'équation :
         if P (1) * 10 + P (2) + P (2) * 10 + P (1) /=
            P (3) * 100 + P (1) * 10 + P (3)
         then
            Acc := False;
         end if;
      end if;
      return Acc;
   end Accepte;
   function DefiniFils (I : Natural; P : TP) return Natural is
   begin
      return 0;
   end DefiniFils;
   function DefiniFrere (I : Natural; P : TP) return Natural is
   begin
      return P (I) + 1;
   end DefiniFrere;
   function ExisteFils (I : Natural; P : TP) return Boolean is
   begin
      return I < N;
   end ExisteFils;
   function ExisteFrere (I : Natural; P : TP) return Boolean is
   begin
      return P (I) < K;
   end ExisteFrere;

begin
   Put ("Crypto, équation : AB + BA = CAC, solution : ");
   ParcoursArbrePartiel
     (N,
      Accepte'Access,
      DefiniFils'Access,
      DefiniFrere'Access,
      ExisteFils'Access,
      ExisteFrere'Access);
end Crypto;

procedure Knapsack is
   -- 5 objets caractérisés par leur taille, poids et importance
   -- 1 sac caractérisé par sa taille maximale et son poids maximal
   -- Solutions de valeur 0 à 1 suivant que l'objet reste ou est pris dans le sac
   N             : constant := 5;
   K             : constant := 1;
   Taille        : constant TP (1 .. N) := (1, 2, 3, 4, 5);
   Poids         : constant TP (1 .. N) := (2, 4, 6, 8, 10);
   Importance    : constant TP (1 .. N) := (5, 4, 3, 2, 1);
   TailleMax     : constant Positive    := 10;
   PoidsMax      : constant Positive    := 15;
   ImportanceMax : Natural              := 0;
   Solution      : TP (1 .. N);
   function SommePond (X : TP; P : TP) return Natural is
      Somme : Natural := 0;
   begin
      for I in 1 .. N loop
         Somme := Somme + X (I) * P (I);
```

```
         end loop;
         return Somme;
      end SommePond;
      function Accepte (I : Natural; P : TP) return Boolean is
         Acc           : Boolean := True;
         ImportanceSac : Natural;
      begin
         if I = N then
            if SommePond (Taille, P) > TailleMax or else SommePond (Poids, P) > PoidsMax
then
               Acc := False;
            end if;
         end if;
         if I = N and then Acc then
            ImportanceSac := SommePond (Importance, P);
            if ImportanceSac > ImportanceMax then
               ImportanceMax := ImportanceSac;
               Solution      := P;
            end if;
         end if;
         return Acc;
      end Accepte;
      function DefiniFils (I : Natural; P : TP) return Natural is
      begin
         return 0;
      end DefiniFils;
      function DefiniFrere (I : Natural; P : TP) return Natural is
      begin
         return P (I) + 1;
      end DefiniFrere;
      function ExisteFils (I : Natural; P : TP) return Boolean is
      begin
         return I < N;
      end ExisteFils;
      function ExisteFrere (I : Natural; P : TP) return Boolean is
      begin
         return P (I) < K;
      end ExisteFrere;

   begin
      Put ("Knapsack, solutions avec crtières taille et poids max : ");
      ParcoursArbrePartiel
        (N,
         Accepte'Access,
         DefiniFils'Access,
         DefiniFrere'Access,
         ExisteFils'Access,
         ExisteFrere'Access);
      Put ("Knapsack, la solution avec crtière relatif importance la meilleure : ");
      Imprime (Solution);
   end Knapsack;

   function FactorielleI (N : Natural) return Positive is
      F : Positive := 1;
   begin
      for I in 2 .. N loop
         F := F * I;
      end loop;
      return F;
   end FactorielleI;
```

```
function FactorielleR (N : Natural) return Positive is
begin
   if N <= 1 then
      return 1;
   else
      return FactorielleR (N - 1) * N;
   end if;
end FactorielleR;

function CombinaisonI (N : Positive; P : Natural) return Positive is
   C : array (0 .. N, 0 .. N) of Positive;
-- Construction du tableau des C(N,P) de l'ordre de N**2
begin
   for X in 0 .. N loop
      for Y in 0 .. P loop
         if X = Y or else Y = 0 then
            C (X, Y) := 1;
         else
            if X > Y then
               C (X, Y) := C (X - 1, Y) + C (X - 1, Y - 1);
            end if;
         end if;
      end loop;
   end loop;
   return C (N, P);
end CombinaisonI;

function CombinaisonR (N : Positive; P : Natural) return Positive is
-- Ordre Exp(N)
begin
   if P = 0 or else P = N then
      return 1;
   else
      return CombinaisonR (N - 1, P - 1) + CombinaisonR (N - 1, P);
   end if;
end CombinaisonR;

function CombinaisonF (N : Positive; P : Natural) return Positive is
begin
   return FactorielleR (N) / (FactorielleR (P) * FactorielleR (N - P));
end CombinaisonF;

function Pal (Ch : String; Deb, Fin : Positive) return Boolean is
   P : Boolean  := True;
   I : Positive := Deb;
   J : Positive := Fin;
begin
   if Deb < Fin then
      while P and then (I < J) loop
         if Ch (I) /= Ch (J) then
            P := False;
         else
            I := I + 1;
            J := J - 1;
         end if;
      end loop;
      return P;
   else
      return False;
   end if;
end Pal;
```

```
   function Palindrome (Ch : String) return Boolean is
   begin
      return Pal (Ch, Ch'First, Ch'Last);
   end Palindrome;

   function PalindromeParMorceau (Ch : String) return Boolean is
      function PalParMorceau (Ch : String; Deb, Fin : Positive) return Boolean is
         P : Boolean  := False;
         I : Positive := Deb + 1;
      begin
         while not P and I <= Fin loop
            if Pal (Ch, Deb, I) and then (PalParMorceau (Ch, I + 1, Fin) or else I = Fin)
then
               P := True;
            else
               I := I + 1;
            end if;
         end loop;
         return P;
      end PalParMorceau;
   begin
      return PalParMorceau (Ch, Ch'First, Ch'Last);
   end PalindromeParMorceau;

   type tabindex is range 1 .. 100;
   type tab is array (tabindex range <>) of Integer;
   X : tab := (25, 30, -4, 0, -10, 31, 4, -2, 26, -11);
   -- Y : tab := (1 => 25);
   procedure TriTab is new BubbleSort (Integer, tabindex, tab);
   T0 : Natural;

begin
   T0 := HorlogeMS;

   Put_Line ("* Exemple 1.1 :");
   Put ("PGCD de 72 et 16 :");
   Put_Line (PGCD (72, 16)'Img);
   -- Put_Line (PGCD (1, 1)'Img);

   Put_Line ("* Exercice 1.4 :");
   Put_Line ("Tableau de nombres entiers triés :");
   TriTab (X);
   for i in X'Range loop
      Put (X (i)'Img);
   end loop;
   New_Line;
   -- TriTab (Y);

   Put_Line ("* Exemple 2.6 :");
   Put ("Factorielle 10 :");
   Imprime (BigFact (10));
   Put ("Factorielle 11 :");
   Imprime (BigFact (11));
   -- Imprime (BigFact (1100000));

   Put_Line ("* Exercice 2.9 :");
   Put ("3 puissance 5 (itératif) :");
   Put_Line (PowerI (3.0, 5)'Img);
   Put ("1,5 puissance 4 (itératif) :");
   Put_Line (PowerI (1.5, 4)'Img);
```

```
   Put ("3 puissance 5 (récursif) :");
   Put_Line (PowerR (3.0, 5)'Img);
   Put ("1,5 puissance 4 (récursif) :");
   Put_Line (PowerR (1.5, 4)'Img);

   Put_Line ("* Exemple 2.13 :");
   N_Uplets (3, 2);
   N_Uplets (2, 3);
   --  N_Uplets(2, 11);
   --  N_Uplets(11, 3);

   Put_Line ("* Exemple 3.15 et exercice 3.10 :");
   Permutations (3);

   Put_Line ("* Exercice 3.11 :");
   Put_Line ("Coloriage planaire :");
   ColoriagePlanaire;

   Put_Line ("* Exemple 3.17 et exercice 3.14 :");
   -- Put_Line ("Nombres premiers < 100 :");
   -- Imprime (NombresPremiers (100));
   -- Put_Line ("Nombres premiers < 1 :");
   -- Imprime (NombresPremiers (1));
   -- Imprime (NombresPremiers (4));
   PlusGrandDiviseurPremier (4, 2);

   Put_Line ("* Exercice 3.16 :");
   Crypto;

   Put_Line ("* Exercice 3.17 :");
   Knapsack;

   Put_Line ("* Exemple 4.2 et exercice 4.1 :");
   Put ("Factorielle(11) (itératif) :");
   Put_Line (FactorielleI (11)'Img);
   -- Put_Line (FactorielleI(0)'Img);
   Put ("Factorielle(11) (récursif) :");
   Put_Line (FactorielleR (11)'Img);
   -- Put_Line (FactorielleR(0)'Img);
   Put ("Combinaison(11, 3) (itératif) :");
   Put_Line (CombinaisonI (11, 3)'Img);
   Put ("Combinaison(11, 3) (récursif) :");
   Put_Line (CombinaisonR (11, 3)'Img);
   Put ("Combinaison(11, 3) (factorielle) :");
   Put_Line (CombinaisonF (11, 3)'Img);

   Put_Line ("* Exercice 4.5 :");
   Put ("ABCBA palindrome ? ");
   Put_Line (Palindrome ("ABCBA")'Img);
   Put ("00100100 palindrome ? ");
   Put_Line (Palindrome ("00100100")'Img);
   Put ("01010010 palindrome par morceaux ? ");
   Put_Line (PalindromeParMorceau ("01010010")'Img);
   Put ("010100101 palindrome par morceaux ? ");
   Put_Line (PalindromeParMorceau ("010100101")'Img);

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