De Pascal à Ada95

Yann Rouzaud
(remis à jour par Roger Mohr)

Dans la suite ADA *désigne toujours* ADA95
Comme en Pascal, les identificateurs Ada peuvent indifféremment être écrits en minuscules ou en majuscules :`EnSiMaG` ≡ `ensIMAG`.

1 Programme

Il n'y a pas de mot-clé *program* ; il faut déclarer les paquetages utilisés :
- Ada.Text_IO pour les entrées-sorties de caractères
- Ada.Integer_Text_IO pour les entrées-sorties d'entiers
- Ada.Float_Text_IO pour les entrées sorties de réels simple précision
- Ada.Numerics.Elementary_Functions pour les fonctions numeriques simple précision

D'autres paquetages peuvent être utilisés et sont paramétrés par les types utilisés, comme par exemple
`package couleur_IO is new text_IO.enumeration_IO(couleur) ;`

Un premier exemple :

```
program p ;
begin
   writeln('Hello, world') ;
end.
```

```
with Ada.Text_IO; use Text_IO ;
procedure p is
begin
   put("Hello, world"); new_line ;
end p ;
```

2 Déclarations

Il n'est pas nécessaire de les regrouper par nature : constantes, types, variables, etc. En contrepartie, leur nature est rappelée pour chaque déclaration.

```
const max = 10 ;
      pi  = 3.14 ;
      msg = 'hello, world' ;
      ch  = 'A' ;
```

```
max : constant integer  := 10 ;
pi  : constant float    := 3.14 ;
msg : constant string   := "hello, world";
ch  : constant character := 'A' ;
```

3 Constantes primitives

```
true
false
```

```
true
false
```

```
maxint                                  integer'last (2**31 -1)
-maxint                                 integer'first (-2**31)
```

4 Types primitifs

```
integer                                 integer
                                        natural       -- entiers positifs ou nuls
                                        positive      -- entiers positifs
real                                    float         -- simple precision
                                        long_float    -- double precision
boolean                                 boolean
char                                    character     -- ISO 8 bits Latin-1
                                        wide.charater -- ISO 16 bits MPL
```

5 Déclarations de type

```
type                                    type T1 is array(1..10) of integer ;
   T1 = array[1..10] of integer ;       type T2 is array(1..2, 1..5) of float ;
   T2 = array[1..2, 1..5] of real ;     subtype S  is string(1..6) ;
   S  = packed array[1..6] of char ;    subtype indice is integer range 1..20 ;
   indice = 1..20 ;                     type couleur is (rouge, vert, bleu) ;
   couleur = (rouge, vert, bleu) ;      type enreg ;
                                        type liste is access enreg ;
   liste = ^enreg ;                     type enreg is
   enreg =                                 record
      record                               val : integer ;
         val : integer ;                   suiv : liste ;
         suiv : liste ;                end record ;
      end ;
```

6 Déclarations de variables

```
var
   B  : boolean ;                       B  : boolean ;
   V1 : T1 ;                            V1 : T1 ;
   M  : packed array[1..5] of char ;    M  : string(1..5) ;
```

7 Déclarations de procédures et fonctions

```
procedure p(...) ;
   ...
begin
   ...
end ;

function f(...) : ... ;
   ...
begin
   ...
   f := ... ;
end ;

/* reference en avant */
procedure q(...) ; forward ;
procedure r(...) ;
begin
   q(...) ;
end ;
/* on ne repete pas les parametres */
procedure q ; ...
```

```
procedure p(...) is
   ...
begin
   ...
end p ;

function f(...) return ... is
   ...
begin
   ...
   return ... ;
end f ;


procedure q(...) ;
procedure r(...) is
begin
   q(...) ;
end r ;
-- on repete les parametres
procedure q(...) is ...
```

8 Paramètres de procédures et de fonctions

En Pascal comme en Ada, le type d'un paramètre doit être un identificateur. Il n'y a pas d'équivalence directe pour le mode de passage. En particulier, il y a toujours recopie pour les paramètres scalaires.

En Pascal, les données non modifiées sont généralement passées par valeur si elles sont simples, par variable si elles sont structurées (tableaux) ; en Ada, il faut utiliser le mot-clé *in* si on souhaite être dans ce cas.

En Pascal, les paramètres résultats sont passés par variable ; en Ada, utiliser le mot-clé *out* pour les résultats purs, et le mot-clé *in out* pour les données – résultats.

Attention : en Ada, les paramètres *in* ne peuvent pas être modifiés (contrairement aux paramètres valeurs de Pascal) ; la paramètres *out* peuvent en revanche être lus après avoir été initialisés.

```
procedure swap(var x, y : integer) ;
var z : integer ;
begin
   z := x ;
   x := y ;
   y := z ;
end ;
```

```
procedure swap(x, y : in out integer) is
   z : integer ;
begin
   z := x ;
   x := y ;
   y := z ;
end swap ;
```

```
procedure p(var Tab : T1) ;
/* Tab non modifie */ ...

function f(x : integer) : integer ;
is

begin
   if x < 0 then
      x := -x ;

   f := x ;
end ;
```

```
procedure p(Tab : in T1) is
...

function f(x : in integer) return integer

   res : integer := x ;
begin
   if x < 0 then
      res := - res ;
   end if ;
 return res ;
end f ;
```

9 Expressions arithmétiques et logiques

Si un opérateur Ada attend un réel, on ne peut pas utiliser de constante entière. Pour les opérateurs binaires, les opérandes doivent nécessairement être de même type : il n'y a pas de conversion implicite.

```
x + 1
x - 1
x * 3
x / 2.5
x div 2
x mod y



/* conversion implicite */



round(x)
trunc(x)


1)
```

```
x + 1
x - 1
x * 3
x / 2.5
x / 2
x rem y       -- meme signe que x
x mod y       -- meme signe que y

abs(x)        -- valeur absolue
x**3          -- elevation a la puissance

float(1)      -- conversion entier -> reel
float(x)
long_float(x)

integer(x)    -- conversion reel -> entier
integer(x)

-- fonctions numeriques (cf paragraphe

sqrt(2.0)     -- racine carree
log(0.0)      -- log neperien
log(0.0, 10.0)-- log en base x
exp(0.0)      -- e ** x
sin(1.0)      -- sinus en radians
sin(1.0, 360.0)-- sinus en degres
-- idem pour cos, tan, cot
arcsin(0.5)    -- arcsinus en radians
arcsin(0.5, 360.0)-- arcsinus en degres
```

```
                                    -- idem pour arccos, arctan, arccot
                                    sinh(1.0)      -- sinus hyperbolyque
                                    -- idem pour cosh, tanh, coth
                                    arcsinh(1.0)   -- arc sinus hyperbolique
                                    -- idem pour arccosh, arctanh, arccoth
```

```
T[i, j]                             T(i, j)       -- element de tableau
R.val                               R.val         -- champ d'enregistrement
 L^.val                              L.val        -- acces a un enregistrement
L^                                  L.all         -- tout l'enregistrement
nil                                 null          -- pointeur nul
```

```
x = 1                               x = 1
x <> 1                              x /= 1
x < 1                               x < 1
x >= 3                              x >= 3
```

```
not b                               not b
a and b                             a and b           -- parallele
                                    a and then b      -- sequentiel
a or b                              a or b            -- parallele
                                    a or else b       -- sequentiel
```

10 Instructions

Le point-virgule est un séparateur d'instructions en Pascal, un terminateur d'instructions en Ada ; une instruction vide se note *null* en Ada. Les instructions structurées Ada ne nécessitent pas l'usage de *begin ... end*.

```
x := x + 1 ;                        x := x + 1 ;
```

```
if x > 0 then                       if x > 0 then
begin
   y := 0 ;                            y := 0 ;
end ;                               end if ;
```

```
if x > 5 then                       if x > 5 then
   y := 1                              y := 1 ;       -- noter le ";"
else if x > 0 then                  elsif x > 0 then
        y := 2                         y := 2 ;
     else                           else
        y := 3 ;                        y := 3 ;
                                    end if ;
```

```
while x < y do                      while x < y loop
   x := x * 2 ;                        x := x * 2 ;
                                    end loop ;
```

```
/* i doit etre declaree */
for i := 1 to 10 do
   x := x + i ;

/* i n'a pas de valeur ici */

for i := 10 downto 1 do
   x := x + i ;
```

```
-- i ne doit pas etre declaree
for i in 1..10 loop
   x := x + i ;
end loop ;
-- i n'est pas utilisable ici

for i in reverse 1..10 loop
   x := x + i ;
end loop ;
new enreg     -- delivre un pointeur
new enreg'(0, null)
```

```
/* var L : liste ; */
new(L) ;
L^.val := 1 ;
L^.suiv := nil ;
```

```
-- L : liste ;
L := new enreg'(1, null) ;
```

11 Entrées-sorties

```
read(i, j) ;
readln(k) ;
write(i, j) ;
writeln(k) ;
write("bonjour") ;
```

```
get(i) ; get(j) ;
get(k) ; skip_line ;
put(i) ; put(j) ;
put(k) ; new_line ;
put("bonjour") ;
```

12 Un exemple complet : pgcd par soustractions successives

12.1 Pascal

```
program test_pgcd ;

   function pgcd(x, y : integer) : integer ;
   /* precondition : x et y doivent etre strictement positifs */
   /*    resultat  : le pgcd de x et y                        */
   /* c'est au progammeur de garantir cette precondition */
   begin
      while x <> y do
         /* durant l'iteration, x et y gardent le meme pgcd */
         if x > y then
            x := x - y
         else
            y := y - x ;
      pgcd := x
    end ;

    procedure tester ;
       var x, y, z : integer ;
    begin
       write('Entrez x et y : ') ;
       read(x, y) ;
       z := pgcd(x, y) ;
    /* notez qu'aucune verification de la precondition n'a ete faite*/
       writeln('Le pgcd vaut : ', z) ;
    end ;

begin
   /* le programme boucle si une des deux valeurs lues est nulle */
   tester ;
end.
```

12.2 Ada

```
with Ada.Text_IO, Ada.Integer_Text_IO ;
use  Ada.Text_IO, Ada.Integer_Text_IO ;

procedure test_pgcd is

   function pgcd(x, y : in positive) return positive is
      a : positive := x ;
      b : positive := y ;
   begin
      while a /= b loop
          --  pgcd(a, b)  = pgcd(x, y)
         if a > b then
            a := a - b ;
         else
            b := b - a ;
         end if ;
      end loop ;
      return a ;
    end pgcd ;

    procedure tester is
       x, y, z : positive ; -- les valeurs pour le calcul du pgcd
    begin                   -- z est le resultat du calcul
       put("Entrez x et y : ") ;
       get(x) ;
       get(y) ;
       z := pgcd(x, y) ;
       put("Le pgcd vaut : ") ;
       put(z) ;
       new_line ;
    end tester ;

begin
   -- il y a une erreur d'execution si une des deux valeurs lues est negative
   --   ou nulle
   tester ;
end test_pgcd ;
```