Par :: Yves Bailly :: http://www.kafka-fr.net ::

Structures de contrôle en Ada

L'introduction de cet article pourrait reprendre le début de celle de l'article du mois dernier : à moins d'être trivial, un programme n'est jamais parfaitement linéaire. Nous allons découvrir aujourd'hui comment contrôler le flux d'instructions, comment choisir entre différentes possibilités ou répéter une portion de code.

Petite remarque avant d'aborder le sujet du jour. Vous trouverez, dispersées dans le texte, des références comme [AARM 5.8-4.b] ou [AARM J.5].

Ce sont des références au Manuel de Référence Ada Annoté ou *Annotated Ada Reference Manual*, pour vous permettre de retrouver la section du standard Ada (ISO/IEC 8652:1995-E) en relation avec le sujet traité.

Le premier nombre est le numéro de chapitre (ou la lettre d'une annexe, dans le deuxième exemple), le deuxième le numéro de section, éventuellement suivi par le numéro de clause (paragraphe). Utilisez ces informations si vous voulez plus de détails, mais cela n'a rien d'obligatoire !

Ce qui figure dans ces articles est normalement suffisant pour nos besoins. Vous trouverez un exemplaire du AARM sur le CD accompagnant le magazine ou par le lien donné en [1].

Le bloc en Ada

Grossièrement, on peut dire qu'un bloc en Ada est une suite d'instructions entre un `begin` et un `end` correspondant, introduite par certains mots-clefs – par exemple, `procedure` ou `function` que nous avons vus le mois dernier.

Si des variables locales doivent être utilisées dans les instructions du bloc, elles doivent être déclarées entre le mot-clef introductif et le `begin` qui suit : il n'est pas possible de déclarer une variable entre deux instructions comme en C++.

Mais il est parfois souhaitable de ne déclarer une variable qu'à un certain moment, dans une suite d'instructions et non dès le début d'un sous-programme. Il existe pour cela le mot-clef `declare` [AARM 5.6], par exemple :

```
 1 procedure Ma_Proc is
 2    v1: Integer ;
 3    procedure Autre_Proc is
 4       v2: Integer ;
 5    begin
 6       -- blablabla...
 7       null ;
 8    end ;
 9    v3: Integer ;
10 begin
11    -- du code...
12    null ;
13    declare
14       v4: Float ;
15       procedure Sous_Proc is
16          v5: Float ;
17       begin
18          -- des instructions...
19          null ;
20       end ;
21    begin
22       -- encore du code...
23       null ;
24    end ;
25    -- et pour finir...
26    null ;
27 end ;
```

Ceci est du code Ada parfaitement valide, il peut être compilé. Quelques remarques :

- La procédure `Autre_Proc`, déclarée ligne 3, est encadrée par la déclaration des variables `v1` et `v3`. Tout naturellement, `Autre_Proc` peut utiliser `v1`, mais pas `v3`.
- Tout aussi naturellement, `v2` n'existe qu'à l'intérieur de `Autre_Proc`, y faire référence dans le code de `Ma_Proc` (de la ligne 10 à la ligne 27) serait une erreur.
- Ces lignes, justement, contiennent deux blocs imbriqués : le premier commence ligne 13 et s'étend jusque ligne 24, le second va de la ligne 15 à ligne 19 (il s'agit en fait d'une procédure). Le premier apparaît effectivement au milieu d'autres instructions (lignes 12 et 26). Là encore, les règles de visibilité usuelles s'appliquent : `v4` n'existe qu'entre les lignes 14 et 24, `v5` n'existe qu'entre les lignes 16 et 20. `v1`, `Autre_Proc` et `v3` sont par contre utilisables.

Remarquez qu'il est parfaitement possible de déclarer un sous-programme dans un autre, voire dans un bloc au sein d'un sous-programme qui se trouve dans un sous-programme contenu dans un bloc...

Il n'y a pas de limite théorique à la « profondeur d'imbrication » des blocs. Nous reviendrons plus tard sur les blocs, mais passons maintenant aux outils qui permettent d'agir sur le flux d'instructions du programme.

Goto

Le presque inévitable `goto`, certains disent « l'infâme `goto` », est disponible en Ada. Inutile de s'appesantir sur son utilité, s'il est bien ou mal de s'en servir.

`goto` existe au moins formellement depuis les origines de l'algorithmique avec le mathématicien perse Abu Abdullah Muhammad bin Musa al-Khwarizmi, au

IXème siècle [2], ce qui ne nous rajeunit pas. Alors voici un exemple d'utilisation, avec cinq erreurs, pour montrer qu'on ne peut quand même pas faire n'importe quoi :

```
 1 procedure P is
 2    procedure PP is
 3    begin
 4       <<LABEL_1>>
 5       goto LABEL_1 ;
 6 --      goto LABEL_2 ;  -- erreur !
 7 --      goto LABEL_3 ;  -- erreur !
 8 --      goto LABEL_4 ;  -- erreur !
 9    end ;
10 begin
11    <<LABEL_2>>
12    declare
13       v: Integer ;
14    begin
15       <<LABEL_3>>
16 --      goto LABEL_1 ;  -- erreur !
17       goto LABEL_2 ;
18       goto LABEL_3 ;
19       goto LABEL_4 ;
20    end ;
21 --   goto LABEL_1 ;     -- erreur !
22    goto LABEL_2 ;
23 --   goto LABEL_3 ;     -- erreur !
24    goto LABEL_4 ;
25    <<LABEL_4>>
26    null ;
27 --   goto LABEL_3 ;     -- erreur !
28 end ;
```

Les noms des labels, destination des branchements par `goto`, sont donnés entre double chevrons (ligne 4, par exemple). Ici les noms sont en majuscules, mais c'est par pure convention : vous êtes libres de les nommer comme bon vous semble.

Les erreurs illustrent la règle générale qui gouverne les sauts par `goto` : il est interdit de sortir d'un bloc pour entrer dans un autre qui ne contient pas le bloc de départ [AARM 5.8-6]. En dehors de cette règle, toutes les horreurs sont permises.

L'instruction `null` en ligne 26 est imposée par la syntaxe : un label doit forcément être suivi par une instruction.

Branchement conditionnel

Voyons maintenant quelque chose de plus familier, le branchement conditionnel [AARM 5.3] :

```
 1 procedure P is
 2    i: Integer ;
 3 begin
 4    i := 8 ; -- ou autre chose...
 5    if i = 1
 6    then
 7       -- si i vaut 1, alors ...
 8       null ;
 9    elsif i > 2
10    then
11       -- si i est supérieur à  2, alors...
12       null ;
13    else
14       -- dans tous les autres cas
15       null ;
16    end if ;
17 end ;
```

Les parenthèses autour de la condition qui suit le `if` ne sont pas nécessaires. Par contre, le mot-clef `then` (alors) est indispensable.

Remarquez qu'il est possible d'effectuer plusieurs tests imbriqués, enchaînés en fait, au moyen de `elsif` (Python utilise `elif` pour le même but). Les instructions suivants le `else` final ne sont exécutées que si tous les tests précédents ont échoués. Enfin, l'ensemble de la structure se termine par un `end if`.

Ici aussi, les `nulls` sont imposés par la syntaxe. Par ailleurs, le type du résultat de l'expression qui figure dans les lignes `if` ou `elsif` doit être d'un type booléen, c'est-à-dire d'un type issu de Boolean (ou de Boolean lui-même). À noter qu'en Ada, une affectation avec `:=` n'est pas une expression : c'est une instruction et elle n'a pas de valeur. Donc, si vous écrivez par maladresse quelque chose comme `if i := 1`, vous serez gratifié d'une rouspétance du compilateur. La célèbre confusion entre `=` et `==` qui survient si souvent en C/C++ est ici tout simplement impossible. Voici l'occasion de dresser une liste des opérateurs booléens et de comparaison en Ada, par ordre de précédence croissante (certains des opérateurs suivants sont également utilisés dans d'autres contextes, que nous verrons par la suite) :

Listes des opérateurs Booléens

OPÉRATEUR	OPÉRATION	TYPE DES OPÉRANDES	TYPE DU RÉSULTAT
`and`, `or`, `xor`	ET, OU, OU exclusif logiques	Boolean tableau de Boolean	Boolean tableau de Boolean
`and then`, `or else`	voir plus bas !	Boolean	Boolean
`= /=`	égalité, différence	Quelconque	Boolean
`<`, `<=`, `>`, `>=`	inférieur, inférieur ou égal, supérieur, supérieur ou égal	Scalaire (numérique) tableau unidimensionnel	Boolean
`in`, `not in`	appartenance à un intervalle ou a un type	scalaire – intervalle quelconque - nom d'un type	Boolean
`not`	Négation	Boolean tableau de Boolean	Boolean tableau de Boolean

Sont donnés ici les opérateurs qui produisent un résultat sous forme de tableau, mais comme nous ne verrons les tableaux que le mois prochain, n'en tenez pas plus compte (c'est juste par souci de complétude). La deuxième ligne appelle probablement quelques commentaires. Considérons deux expressions, X et Y, à valeur booléenne. Ce peuvent être de simples variables ou bien quelque chose de beaucoup plus compliqué contenant des appels de fonctions – n'importe quoi qui vaut une valeur booléenne. En C/C++, l'équivalent de X and Y est X && Y et dans ce cas le terme Y n'est évalué que si le terme X est vrai. Pas en Ada : dans l'expression X and Y, les deux opérandes sont toujours évalués dans un ordre arbitraire. En fait, ceci est vrai pour tous les opérateurs... sauf les deux de la deuxième ligne.

`X and then Y` donne le même résultat que `X and Y`, à ceci prêt que X est toujours évalué avant Y, ce dernier n'étant évalué que si X est vrai – on retrouve là un comportement équivalent à l'opérateur `&&` du C/C++. De la même manière, `X or else Y` donne le même résultat que `X or Y`, sauf que X est évalué en premier et que Y n'est évalué que si X est faux – comme le `||` du C/C++. [AARM 4.5.1-7]

Si le mode de fonctionnement des opérateurs `and`, `or` et consorts vous étonne, sachez que ces opérateurs sont en réalité de simples raccourcis pour des fonctions de la forme suivante :

```
function "and"(Left, Right : Boolean) return Boolean ;
function "or" (Left, Right : Boolean) return Boolean ;
function "xor"(Left, Right : Boolean) return Boolean ;
```

Et ainsi de suite. Or, dans un appel de fonction, les expressions pour chacun des paramètres sont évaluées dans un ordre

quelconque – la plus à droite peut-être avant la plus à gauche. Il est donc tout à fait logique que les opérateurs correspondants suivent la même « incertitude ».

Incidemment, puisque nous avons là des fonctions (bien qu'avec un nom un peu particulier, une chaîne de caractères), cela signifie que nous pouvons les surdéfinir pour nos propres types et nos propres besoins. Il n'est par contre pas permis de définir de nouveaux opérateurs, donc une déclaration comme :

```
function "machin" (a, b: Un_Type) return Un_Autre_Type ;
```

est interdite.

De nombreux autres langages de programmation font souvent l'économie du mot `then` et parfois même de la terminaison par `end if`.

Ils sont obligatoires en Ada. Si cela vous paraît une lourdeur inutile, rappelons que Ada a été conçu pour la réalisation de programmes complexes, avec une très forte exigence de fiabilité et susceptibles de maintenance et modifications sur plusieurs années par de nombreuses personnes différentes.

En sacrifiant la légèreté à la clarté, les concepteurs de Ada ont estimé que ces objectifs seraient plus facilement satisfaits.

Le choix multiple

Il s'agit ici de déterminer les instructions à exécuter selon la valeur d'une expression parmi une série de valeurs possibles.

On peut réaliser cela en empilant des `if..then..elsif..else..end if`, mais la structure `case` [AARM 5.4] est précisément dédiée à une telle situation :

```
 1 with Text_IO ; use Text_IO ;
 2 procedure P is
 3    i: Integer := 7 ;
 4 begin
 5    case i is
 6       when 1 =>
 7          Put_Line("C'est 1 !") ;
 8       when 2..5 =>
 9          Put_Line("Entre 2 et 5 !") ;
10       when 6 | 8 | 10 =>
11          Put_Line("6, 8, ou 10 !") ;
12       when 9 | 11..15 | 7 =>
13          Put_Line("Heu, compliqué...") ;
14       when others =>
15          Put_Line("Cas général.") ;
16    end case ;
17 end
```

L'exemple précédent, disons-le immédiatement, ne montre pas toute la richesse possible de cette structure – mais il donne une idée que j'espère assez alléchante. Il n'aura échappé à personne qu'il s'agit là de l'équivalent de la structure `switch...case` du C/C++ ou de Java (entre autres), avec tout de même quelques nuances de taille. Pour commencer, la structure s'écrit plutôt `case..when`.

Ensuite, remarquez l'absence d'équivalent au `break` du C : contrairement à ce langage (ou à C++ ou à Java...), chacun des cas du choix multiple s'arrête là où commence le suivant.

Par ailleurs, si les valeurs déterminantes du choix doivent être statiques (c'est-à-dire connues à la compilation), il est possible de regrouper plusieurs valeurs en une seule expression : la ligne 8 définit un intervalle de valeurs, la ligne 10 une liste, la ligne 12 combine les deux procédés. Ainsi :

- La ligne 9 ne sera exécutée que si i est compris entre 2 et 5 inclus ;
- La ligne 11 ne sera exécutée que si i vaut 6, 8 ou 10 ;
- La ligne 12 ne sera exécutée que si i est compris entre 11 et 15 inclus ou bien vaut 7 ou bien vaut 9.

La dernière clause ligne 14, `when others`, n'est exécutée que si i ne rentre dans aucun des cas précédents. Cette clause est optionnelle, mais si elle est présente elle doit forcément être la dernière. Enfin, la structure se termine par un `end case`.

Ici une seule instruction est donnée pour chaque cas. Rien ne vous empêche d'en donner plusieurs. Si vous avez besoin de déclarer une variable, il faudra avoir recours au `declare..begin..end` que nous avons vu au début de cet article.

Pour information, un `goto` ne permet pas de « sauter » d'un choix à un autre.

Les boucles générales

Comme dans la plupart des langages, Ada propose différentes manières d'écrire les boucles. Voici la même boucle sous trois formes différentes :

```
1 procedure P is
2    i: Integer := 0 ;
3 begin
4    i := 0 ;
5    loop
6       -- des instructions...
7       i := i + 1 ;
8       if i > 5
9       then
10         exit ;
11      end if ;
12   end loop ;
13   --
14   i := 0 ;
15   loop
16      -- instructions...
17      i := i + 1 ;
18      exit when i > 5 ;
19   end loop ;
20   --
21   i := 0 ;
22   while not (i > 5)
23   loop
24      -- du code...
25      i := i + 1 ;
26   end loop ;
27 end ;
```

Une boucle est essentiellement définie par des instructions comprises entre les mots `loop` et `end loop` [AARM 5.5]. La différence réside en fait dans la façon de noter la condition de sortie de la boucle.

Dans le premier exemple, lignes 5 à 12, on fait simplement appel au mot-clef `exit` [AARM 5.7] au sein d'une condition : celui-ci permet de sortir de la boucle la plus proche qui le contient.

Mais comme cette forme est très commune, elle peut être abrégée comme montré dans le deuxième exemple, ligne 18 : l'effet est rigoureusement le même, mais avec moins de caractères à taper au clavier.

À noter que dans ces deux exemples, si le flux de code ne tombe jamais sur une instruction `exit`, la boucle tournera jusqu'à la fin des temps.

Enfin, le dernier exemple (lignes 22 à 26) fait intervenir le mot-clef `while`.

Cette boucle est identique aux précédentes, à ceci près que la condition est évaluée avant l'exécution des instructions dans le corps de la boucle.

La boucle for

Les boucles précédentes pouvaient être qualifiées de non bornées : elles ne s'arrêtent que lorsqu'une certaine condition est remplie, c'est-à-dire potentiellement jamais. Ada propose une dernière forme de boucle, faisant intervenir un compteur [AARM 5.5-9] :

```
1 procedure P is
2    type Entier is new Integer ;
3 begin
4    for i in 1..5
5    loop
6       -- du code...
7       null ;
8    end loop ;
9    for i in Entier range 1..5
```

```
10    loop
11       -- des instructions...
12       null ;
13    end loop ;
14 end ;
```

Contrairement à l'instruction du même nom en C/C++, l'instruction `for` n'est pas un simple synonyme pour une boucle fondée sur un `while`.

D'abord, le compteur (i dans les deux exemples) n'est pas n'importe quelle variable : remarquez qu'il n'est pas déclaré avant la première boucle.

En fait, l'en-tête introduit par `for` vaut implicitement déclaration de la variable qui suit. Cette variable n'existe que pour la durée de la boucle : c'est pourquoi nous pouvons réutiliser le nom i ligne 9, il s'agit d'une toute autre variable.

Ensuite, ce compteur doit obligatoirement être d'un type discret, c'est-à-dire en gros équivalent à un entier. Le type du compteur est défini par le type des bornes de l'intervalle qu'il devra parcourir, lequel intervalle est défini après le mot-clef `in` : ainsi, la ligne 4 définit un compteur nommé i, implicitement de type `Integer`, qui va parcourir successivement les valeurs de 1 à 5 incluses.

La ligne 9 définit un compteur nommé i de type Entier (revoyez éventuellement le premier article de cette série sur la définition d'un type dérivé), qui va parcourir la même plage de valeurs.

Les bornes de l'intervalle ne sont pas nécessairement des valeurs littérales : 1 et 5 pourraient parfaitement être remplacées par des variables, pourvu qu'elles soient de même type. La plage de valeurs ainsi définie peut être nulle, par exemple 2..0 : dans ce cas, les instructions dans la boucle ne sont pas exécutées.

Par ailleurs, les bornes de l'intervalle ne sont évaluées qu'une seule fois, au moment où le `for` est rencontré : cela signifie que si l'une des deux bornes (ou les deux) est une variable, modifier la valeur de cette variable à l'intérieur de la boucle n'aura aucune influence sur le déroulement de celle-ci (contrairement à ce qu'il est possible de faire en C/C++, par exemple). Enfin, si le `in` est suivi du mot-clef `reverse`, les valeurs de l'intervalle sont parcourues en ordre inverse (de la fin vers le début).

Pour finir, au sein de la boucle le compteur est considéré comme une constante : il est interdit de modifier sa valeur par une affectation.

Toute tentative dans ce but sera rejetée par le compilateur. Il est par contre toujours possible d'utiliser l'une des formes du mot-clef `exit` si on souhaite sortir prématurément de la boucle.

On pourra regretter une petite limitation, à savoir que les valeurs de l'intervalle sont forcément parcourues par incréments de 1 : la boucle `for` ne permet pas, par exemple, de n'envisager qu'une valeur sur deux. Pour cela, il faut se rabattre sur la forme générale basée sur `while`.

Un nom pour un bloc

Nous en arrivons à une possibilité très intéressante offerte par Ada : celle de nommer un bloc. Voici un premier exemple, où un même nom de variable a bêtement été utilisé dans trois blocs imbriqués :

```
 1 with Text_IO ; use Text_IO ;
 2 procedure P is
 3    i: Integer ;
 4 begin
 5    i := 1 ;
 6    declare
 7       i: Integer ;
 8       procedure PP is
 9          i: Integer := 3 ;
10       begin
11          Put(Integer'Image(i)&" ") ;
12       end ;
13    begin
14       i := 2 ;
15       PP ;
16       Put(Integer'Image(i)&" ") ;
17    end ;
18    Put_Line(Integer'Image(i)) ;
19 end ;
```

Ce n'est pas très malin, mais parfois la vie mouvementée d'un programme compliqué abouti à ce genre de chose. Dans ces situations, chaque nouvelle déclaration de i « masque » la déclaration précédente : le i de la ligne 11 est celui déclaré ligne 9, celui des lignes 14 et 16 est celui déclaré ligne 7, celui de la ligne 18 est déclaré ligne 5.

Il n'y a aucune ambiguïté. Mais comment faire alors, si on veut utiliser dans PP le i déclaré ligne 5 ou celui ligne 7 ? Utiliser dans le bloc entre les lignes 6 et 17 le i déclaré ligne 5 ?

Il est possible de préfixer une variable avec le nom du bloc dans lequel elle est définie, en utilisant la notation pointée. Dans le cas d'une procédure ou d'une fonction, son nom est celui du bloc : par exemple, si on écrivait `P.i` ligne 11, on ferait référence au i déclaré ligne 3. Il est par contre évidemment aberrant d'écrire `PP.i` ligne 14, car la variable i déclarée ligne 9 n'existe pas à cet endroit : ce qui est déclaré dans un bloc est invisible en dehors de celui-ci.

Voyons maintenant le cas du i déclaré ligne 7. Nous ne sommes pas dans une procédure : le bloc en question est anonyme.

Pour lui donner un nom, il suffit de faire précéder le `declare` ligne 6 par un identifiant suivi des deux-points. Voici le même programme, le bloc en question étant nommé :

```
 1 with Text_IO ; use Text_IO ;
 2 procedure P is
 3    i: Integer ;
 4 begin
 5    i := 1 ;
 6    Nom_Du_Bloc:
 7    declare
 8       i: Integer ;
 9       procedure PP is
10          i: Integer := 3 ;
11       begin
12          Put(Integer'Image(Nom_Du_Bloc.i)&" ") ;
13       end ;
14    begin
15       i := 2 ;
16       PP ;
17       Put(Integer'Image(i)&" ") ;
18    end Nom_Du_Bloc ;
19    Put_Line(Integer'Image(i)) ;
20 end ;
```

Remarquez la ligne 6, qui donne un nom au bloc. Dans ce cas, il est obligatoire de répéter ce nom après le `end` qui termine le bloc, comme cela est fait ligne 18 : si vous oubliez cette répétition, le compilateur vous rappellera à l'ordre. Dès lors, on peut faire référence aux variables de ce bloc explicitement, par exemple ligne 12.

Mieux que l'accès aux variables locales, cette manière de nommer les blocs permet de réaliser des branchements très intéressants, qui seraient assez pénibles à reproduire dans de nombreux autres langages.

Ceci grâce à la possibilité de nommer les structures de contrôle, notamment les boucles que nous avons vues précédemment.

Voyez cet exemple :

```
1 with Text_IO ; use Text_IO ;
2 procedure P is
3 begin
4    put_line("Avant Boucle_1") ;
5    Boucle_1:
```

```
 6   for i in 11..19
 7   loop
 8     put_line("   Avant Boucle_2") ;
 9     Boucle_2:
10     for j in 21..29
11     loop
12       put_line("      Avant Boucle_3") ;
13       Boucle_3:
14       for k in 31..39
15       loop
16         if i > 12
17         then
18           exit Boucle_1 ;
19         end if ;
20         exit Boucle_2 when j > 22 ;
21         exit when k > 32 ;
22         put_line("         "&
23           Integer'Image(i)&" "&
24           Integer'Image(j)&" "&
25           Integer'Image(k)) ;
26       end loop Boucle_3 ;
27       put_line("      Après Boucle_3") ;
28     end loop Boucle_2 ;
29     put_line("   Après Boucle_2") ;
30   end loop Boucle_1 ;
31   put_line("Après Boucle_1") ;
32 end ;
```

Nous avons là trois boucles `for` imbriquées (mais ceci s'applique également aux autres types de boucles), chacune programmée pour faire 9 tours, donc un total théorique de 729 exécutions du code le plus intérieur (lignes 16 à 25).

Chacune des boucles est nommée (lignes 5, 9 et 13), les noms étant répétés au moment de terminer le bloc de la boucle (lignes 26, 28 et 30).

Ici aussi cette répétition est obligatoire, imposée par le langage [AARM 5.5-5].

Ce programme n'effectuera pas 729 tours de boucles : les lignes 16 à 21 provoquent des sorties prématurées de chacune des boucles.

Remarquez comment nous sortons de la boucle la plus extérieure à partir du code exécuté dans la boucle la plus intérieure : l'instruction `exit` ligne 18 est tout simplement suivie du nom du bloc duquel nous voulons sortir.

De même pour la ligne 20, qui utilise la forme abrégée de `exit`.

En l'absence de nom, la ligne 21 provoquera implicitement la sortie de la boucle la plus proche, c'est-à-dire la troisième (dont le nom ne sert en fait à rien).

Saurez-vous deviner l'affichage de ce programme ?

Le voici :

```
Avant Boucle_1
   Avant Boucle_2
      Avant Boucle_3
          11  21  31
          11  21  32
      Après Boucle_3
      Avant Boucle_3
          11  22  31
          11  22  32
      Après Boucle_3
      Avant Boucle_3
   Après Boucle_2
   Avant Boucle_2
      Avant Boucle_3
          12  21  31
          12  21  32
      Après Boucle_3
      Avant Boucle_3
          12  22  31
          12  22  32
      Après Boucle_3
      Avant Boucle_3
   Après Boucle_2
   Avant Boucle_2
      Avant Boucle_3
Après Boucle_1
```

Remarquez les deux dernières lignes qui montrent bien que l'on entre dans la troisième boucle (la plus intérieure), pour sortir directement après la boucle la plus extérieure du fait des lignes 16 à 19.

Des langages conçus bien après Ada, comme Java ou Perl, ont retenu cette technique, sans aller pourtant jusqu'à nommer des blocs de nature similaire à `declare`. On la trouve également sous une forme amoindrie, par exemple dans PHP ou le *shell* Bash, où il est possible de donner un nombre de niveaux d'imbrications à « dépiler » à la suite de l'équivalent de l'instruction `exit`. Les langages les plus utilisés que sont le C et le C++ n'ont par contre rien de comparable. Au-delà des facilités et perspectives qu'ouvre le nommage des blocs, son utilisation permet bien souvent de clarifier un code autrement difficile à suivre et facilite grandement la maintenance : ajouter ou retirer un niveau d'imbrication est moins problématique si chaque niveau est clairement nommé et identifié. En plus d'apporter une grande souplesse et une grande rigueur, nommer les blocs conduit à écrire du code « auto-documenté ».

Conclusion

Voilà pour cette présentation des structures de contrôle fondamentales du langage Ada. Il en existe de plus sophistiquées, si l'on considère la gestion des exceptions et le parallélisme comme des structures de contrôle – nous les explorerons plus tard. Le mois prochain, nous aborderons les structures de données que sont les tableaux et les enregistrements.

Références

[1] *Annotated Ada Reference Manual (AARM)*, http://adaic.org/standards/ada95.html

[2] Les algorithmes, http://en.wikipedia.org/wiki/Algorithm