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

Le langage Ada : sous-programmes

À moins d'être trivial ou codé vite-fait mal-fait, un programme n'est jamais une suite linéaire d'instructions : certaines sont regroupées pour faciliter leur réutilisation. En C/C++, Python et de nombreux autres langages, on parle simplement de « fonctions ».

Le terme sous-programme, ancien en programmation, a une signification plus générale. Il a été repris par le langage Ada, qui divise les sous-programmes en deux catégories : d'une part les fonctions, d'autres part les procédures. Une distinction qui paraît bien étrange aujourd'hui...

Le terme de procédure lui-même ne date pas d'hier : Algol60, langage créé à la fin des années cinquante et duquel sont issus entre autres le C et le Pascal au début des années soixante-dix, l'utilisait déjà pour désigner un sous-programme. Mais que signifie la distinction entre fonctions et procédures ?

N'oublions pas que l'informatique en général, et la programmation en particulier, tirent leurs origines dans les mathématiques. La différence entre les deux concepts peut être résumée ainsi :

- Une procédure **fait** quelque chose ;
- Une fonction **vaut** quelque chose.

Pour faire le parallèle avec le langage C, une procédure (Ada) est une fonction (C) qui ne retourne rien (ou `void`), tandis qu'une fonction (Ada) est une fonction (C) qui retourne une valeur. Cette subtilité peut paraître de pure forme, la distinction purement sémantique. Mais c'est là justement son importance : plus que beaucoup d'autres, le langage Ada s'attache à la sémantique, au sens de ce que le programmeur veut exprimer. D'une certaine manière, on pourrait dire que Ada est un langage expressif, où des facilités sont données au programmeur pour traduire ce qu'il veut faire, par opposition à la plupart des autres langages que l'on pourrait qualifier d'effectifs, où le programmeur dit comment faire ce qu'il veut. C'est probablement ce qui est le plus déroutant quand on est expérimenté dans un ou plusieurs langages « classiques » et que l'on découvre Ada. Nous sommes habitués à exprimer le comment, alors que Ada nous incite à plutôt exprimer le quoi. Et il est souvent plus difficile (du moins en programmation) d'exprimer ce que l'on a en tête, que d'expliquer comment l'obtenir. Nous allons voir que cette subtile distinction entre fonctions et procédures n'est pas sans conséquences sur l'écriture du code en Ada.

Les procédures

Commençons par une procédure simple, ne prenant pas de paramètre. Il pourrait suffire de reprendre l'exemple « Bonjour, Monde ! » du mois dernier : il n'est composé que d'une unique procédure, laquelle contient le code exécuté au lancement du programme (l'équivalent de la fonction `main()` du C). Reprenons ce programme, en créant une autre procédure invoquée par cette procédure principale :

```
1 with Text_IO ;
2 use Text_IO ;
3 procedure bonjour is
4   procedure affiche is
5   begin
6     put_line("Bonjour") ;
7   end ;
8 begin
9   affiche ;
10 end bonjour ;
```

La nouvelle procédure est définie lignes 4 à 7, à l'intérieur de la procédure principale `bonjour`. Cela peut surprendre, mais en fait c'est parfaitement normal : la procédure principale est le programme (elle constitue ce que l'on appelle une unité de compilation), et dans notre cas il n'y a pas d'autre endroit pour définir variables et sous-programmes (mais rassurez-vous, nous verrons bientôt comment organiser les choses en unités séparées). En fait, en Ada, tout sous-programme peut contenir une ou plusieurs définitions d'autres « sous-sous-programmes ».

La définition de la procédure `affiche` ressemble à s'y méprendre à celle de `bonjour`. Remarquez toutefois que le nom n'est pas répété ligne 7, à la fin de la définition. Cette répétition n'est pas obligatoire, simplement vivement recommandée. Remarquez également l'absence de parenthèses, aussi bien dans la définition (ligne 4) que dans l'invocation (ligne 9). Comme il n'y a pas de paramètre, elles sont inutiles – et même interdites : écrire `affiche();` ligne 9, comme on le ferait en C, est une erreur de syntaxe.

Passage de paramètres

Voyons cet exemple contenant trois procédures (en plus de la procédure principale) :

```
1 with Text_IO ;
2 use Text_IO ;
3 procedure Double is
4   procedure Double(i: in Integer) is
5   begin
6     Put(Integer'Image(2*i)) ;
7   end ;
8   procedure Double2(ii: in out Integer) is
9   begin
10    ii := 2*ii ;
11    Put(Integer'Image(ii)) ;
12  end ;
13  procedure Double(i: Integer ; r: out Integer) is
14  begin
15    r := 2*i ;
16    Put(Integer'Image(r)) ;
17  end ;
```

```
18 entier: Integer := 1 ;
19 resultat: Integer := 0 ;
20 begin
21   Double(entier) ;
22   Double2(entier) ;
23   Double(entier, resultat) ;
24   Put_Line(Integer'Image(resultat)) ;
25 end Double ;
```

Ce programme contient trois procédures nommées `Double` : ceci pour montrer que Ada permet la surcharge des noms de sous-programmes. Rappelons, s'il est besoin, qu'il n'est pas fait de différence entre majuscules et minuscules, sauf bien sûr pour le contenu des chaînes de caractères (dont nous reparlerons plus tard en détail). Les langages comme Pascal ou C/C++ distinguent différentes façons de passer les paramètres, soit par valeur, soit par adresse ou par référence. Cela détermine si le sous-programme reçoit une copie de la zone mémoire associée à une variable (passage par valeur) ou s'il reçoit un moyen d'accéder à cette zone mémoire (passage par adresse ou par référence). Cette distinction sur la manière d'obtenir une variable (ou une valeur) transférée à un sous-programme existe en Ada, mais avec une approche différente. On distingue plutôt trois modes de passage :

- Le mode `in`, le mode par défaut, où le paramètre peut être lu et utilisé dans le sous-programme, mais pas modifié ;
- Le mode `in out`, où le paramètre peut être lu et modifié ; cela correspond à un passage par référence ;
- Le mode `out`, plus inhabituel, où le paramètre peut être lu et modifié, mais sa valeur est indéterminée à l'entrée dans la procédure.

Certains commencent peut-être à ouvrir de grands yeux : le mode `in` ressemble à un passage par valeur, n'est-ce pas ? Mais alors, que se passe-t-il si on souhaite donner au sous-programme, en mode « lecture seule », par exemple un gros tableau contenant des millions d'éléments ? En Pascal ou C, passer un tel objet par valeur impliquerait sa duplication sur la pile, ce qui peut être très long, voire provoquer un débordement de pile et un plantage du programme. Le code machine généré pour les paramètres en mode `in` dépend en fait du type du paramètre. Pour les types simples, les types fondamentaux, il s'agit effectivement d'un passage par valeur. Pour les types structurés, comme les tableaux, cela sera (presque) toujours un passage par référence. Le choix du mode de passage interne est dévolu au compilateur. Le programmeur n'a alors pas à se demander de quelle manière passer son paramètre, mais les limites d'accès qui lui sont imposées. Prenons la première procédure, lignes 4 à 7. La déclaration des paramètres ressemble à une déclaration de variable (dont deux exemples sont donnés lignes 18 et 19), en intercalant le mode de passage. Ici le paramètre est passé en mode `in` : sa valeur est donc utilisable par la procédure, mais en aucun cas modifiable. Essayez d'insérer une ligne comme `i:=2;` dans cette procédure : vous obtiendrez une erreur de compilation. Pour mémoire, l'écriture `Integer'Image()`, que nous avons rencontrée le mois dernier, permet de convertir un entier (de type `Integer`) en sa représentation sous forme de chaîne de caractères, au moyen de l'attribut `Image` appliqué au type `Integer`.

La deuxième procédure, lignes 8 à 12, montre un passage en mode `in out`. Cette fois le paramètre peut être modifié, ce qui est fait ligne 10. Dans ce cas, comme lors d'un passage par référence en C++, la procédure ne peut être appelée qu'en lui donnant une variable, alors que la première procédure pouvait être invoquée en donnant une valeur littérale. À l'issu de l'exécution de `Double2()`, la valeur de la variable qui lui aura été donnée sera modifiée. Enfin, la dernière procédure, lignes 13 à 17, montre l'utilisation du mode `out`... et l'absence de mode pour le premier paramètre. En l'absence de mode, c'est le mode `in` qui est appliqué, donc le paramètre `i` est passé en mode `in`. Concernant le paramètre `r`, la grande différence avec le mode `in out` est que la valeur de `r` est indéterminée quand on rentre dans la procédure, même si une valeur a été affectée préalablement à la variable associée. C'est ce que précise le standard du langage. Pour un compilateur donné, le paramètre peut être la variable associée passée par référence, auquel cas la valeur du paramètre sera celle de la variable au moment de l'appel. Un autre compilateur est libre d'adopter une autre stratégie, comme un passage par valeur, suivi d'une affectation à la terminaison de la procédure. Faire l'hypothèse du mode effectif de passage et en tenir compte conduit à un programme généralement non portable sur un autre compilateur. La signification du mode `out` en Ada95 est un peu différente de celle en Ada83 : dans ce dernier, il était explicitement interdit de lire la valeur du paramètre au sein du code de la procédure. La ligne 16 est alors une erreur de syntaxe, car pour afficher la valeur de `r`, il faut lire la valeur de `r`... Dans la pratique, il est préférable de traiter de cette manière les paramètres en mode `out` : ne pas tenter de lire leur valeur, tant qu'ils n'ont pas subi une affectation. Essayez de dupliquer la ligne 16 juste avant la ligne 15 : le compilateur devrait vous gratifier d'un avertissement, car vous tentez d'utiliser la valeur de `r` avant d'avoir affecté une valeur à `r` et ce malgré l'initialisation faite ligne 19.

Saurez-vous trouver l'affichage résultat de l'exécution du programme précédent ? Le voici :

```
$ ./double
 2 2 4 4
```

La procédure `Put()`, fournie par `Text_IO`, affiche une chaîne de caractères sans ajouter de saut à la ligne, contrairement à `Put_Line()`.

Les fonctions

La déclaration d'une fonction est un peu différente de celle d'une procédure. Voyons sur cet exemple, stocké dans `double2.adb` :

```
 1 with Text_IO ;
 2 use Text_IO ;
 3 procedure Double2 is
 4   function Double(i: in Integer) return Integer is
 5   begin
 6     return 2*i ;
 7   end ;
 8   function Double(i: in Integer) return String is
 9   begin
10     return Integer'Image(2*i) ;
11   end ;
12 begin
13   Put_Line(Integer'Image(Double(3))) ;
14   Put_Line(Double(4)) ;
15 end Double2 ;
```

Le mot clef `function` remplace le mot clef `procedure` et cette fois on indique le type retourné par la fonction au moyen du mot clef `return`. Ce même mot clef doit d'ailleurs être présent (une ou plusieurs fois) dans le corps de la fonction et faire partie du flux d'instructions : si le compilateur détecte qu'il est possible d'arriver à la fin des instructions de la fonction sans avoir rencontré `return`, alors une erreur de

compilation est affichée – tandis que ce n'est qu'un avertissement en C++.

Les programmeurs C++ seront peut-être surpris de l'exemple donné : nous avons en effet deux fonctions, de même nom, et prenant les mêmes paramètres... Elles ne se distinguent que par le type de retour. En effet, en Ada le type de retour fait partie de la signature de la fonction, ce qui n'est pas le cas en C++.

Il existe une autre particularité propre aux fonctions. Les paramètres ne peuvent pas être en mode `out` ou `in out`. Ceci provient des inspirations mathématiques du langage : une fonction au sens mathématique du terme ne peut que retourner un résultat, en aucun cas modifier les valeurs qui lui sont éventuellement données à traiter. Aussi les paramètres d'une fonction Ada ne peuvent qu'être en mode `in`, le mode par défaut – la présence du `in` dans l'exemple plus haut est en fait superflue. Toutefois, il est parfois bien pratique et même nécessaire de pouvoir malgré tout modifier un paramètre passé à une fonction. Nous verrons plus tard un quatrième mode de passage, le mode `access`, qui permet cela.

Les deux fonctions définies sont invoquées lignes 13 et 14 : c'est le contexte qui permet de déterminer laquelle effectivement appeler. Ligne 13, l'attribut `Image` appliqué au type `Integer` (ce que l'on pourrait aisément assimiler à un appel de fonction) ne peut accepter qu'un paramètre de type `Integer` : c'est donc la fonction `Double()` qui retourne un `Integer` qui doit être invoquée. De la même manière, la procédure `Put_Line()` (issue du paquetage `Text_IO`) n'accepte qu'une chaîne de caractères de type `String` : ligne 14, on n'a donc d'autre choix que d'invoquer la deuxième fonction.

Une telle souplesse n'est permise que grâce à l'aspect strictement typé de Ada : il n'y a en fait pas de conversion implicite entre types, comme cela est si courant en C/C++. Nous reviendrons plus précisément sur les notions de types et sous-types en Ada, mais l'exemple précédent pourrait être encore poussé plus loin en introduisant un type dérivé :

```
1 with Text_IO ;
2 use Text_IO ;
3 procedure Double3 is
4   type Entier is new Integer ;
5   function Double(i: in Integer) return Integer is
6   begin
7     return 2*i ;
8   end ;
9   function Double(i: in Integer) return Entier is
10  begin
11    return 2*Entier(i) ;
12  end ;
13 begin
14   Put_Line(Integer'Image(Double(3))) ;
15   Put_Line(Entier'Image(Double(4))) ;
16 end Double3 ;
```

Le type `Entier` déclaré ligne 4 possède toutes les caractéristiques du type prédéfini `Integer` et très certainement sa représentation interne binaire est exactement identique. Pourtant, il s'agit bien d'un type différent ! Ce programme compile et s'exécute sans problème, moyennant un transtypage explicite ligne 11. En effet, le type de `i` étant `Integer`, le type de l'expression `2*i` est également `Integer`. Or, la fonction doit retourner un `Entier`. Le typage strict de Ada, répétons-le, interdit la conversion implicite d'un `Integer` vers un `Entier`. Donc si nous avions inscrit simplement `return 2*i;` ligne 11, le compilateur signalerait une erreur. Deux solutions s'offrent alors : soit convertir le paramètre `i` en `Entier` avant de faire le calcul, comme ici, soit faire le calcul en `Integer` et convertir le tout en `Entier`, en écrivant `return Entier(2*i);`.

Dernier mot concernant les fonctions, la valeur retournée ne peut pas être ignorée, comme en C, Python et d'autres langages. Une instruction comme :

```
Double(3) ;
```

sera refusée par le compilateur : elle est assimilée à un appel de procédure, or il n'existe pas de procédure nommée `Double()` et prenant un paramètre entier. Incidemment, cela signifie que l'on peut déclarer une fonction et une procédure ayant le même nom et prenant les mêmes paramètres - encore une fois, le contexte permet de les distinguer.

Valeurs par défaut et paramètres nommés

Comme de nombreux langages, Ada permet de donner une valeur par défaut à certains paramètres d'une procédure (ou d'une fonction), toutefois seulement pour les paramètres en mode `in`. Mais en plus, il est possible de nommer explicitement les paramètres au moment de l'invocation, comme cela se fait également en Python.

Voyons cela sur un exemple :

```
1 with Text_IO ;
2 use Text_IO ;
3 procedure default is
4   procedure proc(a: in Integer ;
5                  b: in Integer := 22 ;
6                  c: in Integer ;
7                  d: in Integer := 44) is
8   begin
9     Put_Line("a = " & Integer'Image(a) &
10              ", b = " & Integer'Image(b) &
11              ", c = " & Integer'Image(c) &
12              ", d = " & Integer'Image(d)) ;
13  end ;
14 begin
15  proc(1, 2, 3, 4) ;
16  proc(d=>4, c=>3, b=>2, a=>1) ;
17  proc(1, c=>3) ;
18 end default ;
```

La déclaration de la procédure `proc`, qui s'étale des lignes 4 à 7 pour plus de lisibilité, prend quatre paramètres dont deux ont une valeur par défaut. Elle se contente d'afficher les valeurs reçues – remarquez à cette occasion l'opérateur `&`, qui permet de réaliser la concaténation de chaînes de caractères : l'équivalent du `+` entre chaînes en Python ou du `.` en Perl.

La première invocation, ligne 15, ne présente rien de particulier : la position de chaque paramètre réel détermine à quel paramètre formel il correspondra. La deuxième, par contre, montre comment il est possible de nommer les paramètres : ceux-ci sont donnés en ordre inverse, mais le résultat est exactement identique à la première. Pour nommer un paramètre, il suffit de donner le nom de celui-ci, puis le symbole `=>`, puis la valeur voulue. Lorsque l'on invoque un sous-programme recevant de nombreux paramètres, cette écriture permet d'améliorer considérablement la lisibilité du code.

Rassurez-vous, il n'y a pas de confusion possible entre noms de paramètres et noms de variables : ce qui est à gauche de `=>` est forcément un nom de paramètre, ce qui est à droite forcément une valeur ou une variable. Ainsi rien n'interdit de définir, par exemple, une variable nommée `b` dans le programme précédent et de la passer à la procédure avec une écriture comme `proc(a=>1, b=>b, c=>3, d=>4);`. Peut-être le membre `b=>b` vous paraît-il étrange, mais le compilateur s'y retrouve sans ambiguïté. Par contre, l'utilisation du *nommage* des paramètres impose une certaine précision : si vous donnez un nom qui n'existe pas dans la liste des paramètres du sous-programme, le compilateur renverra une erreur. Enfin, la dernière invocation ligne 17 montre que les paramètres `b` et `d` peuvent être omis

(puisqu'ils ont une valeur par défaut) et qu'il est possible de mélanger paramètres positionnés et paramètres nommés. En fait, du fait de la forme de la déclaration de la procédure, le nommage du paramètre `c` est pratiquement obligatoire si `b` est absent. En effet, une écriture comme `proc(1,2);` serait incorrecte : par le positionnement, la première valeur sera attribuée au paramètre `a`, la deuxième au paramètre `b`... et de fait, le paramètre `c` n'aura pas de valeur, alors qu'il en requiert une.

À noter qu'à partir du point où vous utilisez le nommage de paramètres, les paramètres suivants doivent obligatoirement être nommés, l'attribution des valeurs par simple positionnement n'étant plus possible. Voici ce qu'affiche ce programme à l'exécution :

```
$ ./default
a = 1, b = 2, c = 3, d = 4
a = 1, b = 2, c = 3, d = 4
a = 1, b = 22, c = 3, d = 44
```

Déclaration et récursivité

Comme tout langage de haut niveau, Ada supporte la récursivité : un sous-programme peut s'invoquer lui-même, deux ou plusieurs sous-programmes peuvent s'inter-invoquer. Mais pour qu'un sous-programme puisse en invoquer un autre, encore faut-il que le deuxième ait été déclaré au moment où apparaît le code du premier. La récursivité croisée n'est alors possible que par l'usage de la déclaration anticipée des sous-programmes. Par exemple :

```
 1 procedure recurse is
 2    procedure p1 ;
 3    procedure p2 ;
 4    procedure p1 is
 5    begin
 6       p2 ;
 7    end ;
 8    procedure p2 is
 9    begin
10       p1 ;
11       p2 ;
12    end ;
13 begin
14    p1 ;
15 end ;
```

Deux procédures, `p1` et `p2`, sont déclarées lignes 2 et 3. Au moment où arrive le code de `p1`, celle-ci peut invoquer `p2` (ligne 6), puisqu'elle a déjà été déclarée. D'un autre coté, la déclaration de `p1` n'est pas ici absolument nécessaire pour qu'elle soit invoquée dans `p2` (ligne 10) : la définition de `p1`, lignes 4 à 7, vaut déclaration.

Les plus observateurs auront remarqué que ce programme présente deux problèmes majeurs : d'une part, `p1` et `p2` vont s'invoquer l'une l'autre sans fin, ce qui en soit suffirait à provoquer un débordement de pile et donc un plantage du programme ; de plus, `p2` s'invoque elle-même également sans fin, avec les mêmes conséquences. Le compilateur Gnat a la gentillesse de nous prévenir de la possible récursivité infinie sur `p2` :

```
$ gnatmake recurse.adb
gcc-3.4 -c recurse.adb
recurse.adb:11:07: warning: possible infinite recursion
recurse.adb:11:07: warning: Storage_Error may be raised at
run time
gnatbind -x recurse.ali
gnatlink recurse.ali
```

Remarquez l'avertissement sur la ligne 11. Par contre, la récursivité croisée infinie ne sera malheureusement pas détectée.

Si vous exécutez ainsi ce programme, cela résultera en une erreur de segmentation (`Segmentation fault`), donc un plantage brutal. Nous verrons les exceptions plus tard, mais sachez qu'il est possible de prévenir ce genre de soucis en demandant un contrôle à l'exécution sur l'encombrement de la pile, au moyen du paramètre `-fstack-check` passé à `gnatmake` :

```
$ gnatmake -fstack-check recurse.adb
gcc-3.4 -c -fstack-check recurse.adb
recurse.adb:11:07: warning: possible infinite recursion
recurse.adb:11:07: warning: Storage_Error may be raised at
run time
gnatbind -x recurse.ali
gnatlink recurse.ali
$ ./recurse
raised STORAGE_ERROR : stack overflow detected
```

Le programme ne se termine plus brutalement lors du débordement de pile. Au lieu de cela, une exception est déclenchée (comme annoncé dans l'avertissement, d'ailleurs), ce qui laisse la possibilité de réagir dans le programme.

Comme aucun traitement n'a été prévu pour cela dans notre code, le programme se termine tout de même. Le prix de cette sécurité étant, naturellement, un code un peu plus gros et un peu plus lent. Exercice : essayez d'intercepter ce genre d'erreur en C++.

Renommer un sous-programme

Pour terminer, voyons une possibilité offerte par Ada pour renommer un sous-programme :

```
 1 with Text_IO ;
 2 use Text_IO ;
 3 procedure Renomme is
 4    procedure Une_Procedure_Avec_Un_Nom_Impossible is
 5    begin
 6       Put_Line("Hello !");
 7    end ;
 8    procedure Proc
 9       renames Une_Procedure_Avec_Un_Nom_Impossible ;
10
11    function Une_Fonction_Qui_N_Est_Pas_Mieux (i: in Integer)
12       return Integer is
13    begin
14       Put_Line("i = " & Integer'Image(i)) ;
15       return 0 ;
16    end ;
17    function Fonct(i: in Integer) return Integer
18       renames Une_Fonction_Qui_N_Est_Pas_Mieux ;
19
20 a: Integer ;
21 begin
22    Proc ;
23    a := Fonct(1) ;
24 end Renomme ;
```

Les lignes 4 à 7 définissent une procédure, dont le nom est un peu pénible à taper au clavier. Les lignes 11 à 16 définissent une fonction, dont le nom n'est guère plus sympathique.

Il est possible d'éviter des crampes aux doigts en renommant ces deux sous-programmes. Le renommage revient en fait à déclarer un nouveau sous-programme, en indiquant que le nom est un synonyme d'un autre sous-programme préalablement déclaré.

Voyez les lignes 8-9 d'une part et 17-18 d'autre part : les premières définissent un synonyme pour la procédure, les secondes un synonyme pour la fonction. Ces définitions prennent la forme de déclarations usuelles, avec toutefois le mot clef `renames` suivi du nom du sous-programme ainsi renommé.

Si l'intérêt de la technique ne paraît pas évident sur cet exemple, nous verrons qu'elle peut s'avérer fort appréciable quand nous aborderons les paquetages Ada, c'est-à-dire en gros les bibliothèques.

Conclusion

Voilà pour cette présentation des sous-programmes en Ada. Étonnamment, nous avons vu que le typage strict de Ada, s'il impose certaines contraintes fortes, offre également une certaine souplesse dans la déclaration de sous-programmes et permet aux compilateurs de détecter certains problèmes.

Le mois prochain, nous continuerons la découverte du langage avec les structures de contrôles que sont les boucles et autres alternatives.