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

Ada : présentation d'un langage pas comme les autres

Il est des langages de programmation qui semblent surgir d'un passé lointain, marqués d'une aura étrange, parfois un peu inquiétants. Le langage de programmation Ada fait partie de ceux-ci. Méconnu, parfois redouté, souvent dénigré, je vous propose de découvrir un langage puissant et moderne utilisé dans les situations critiques où une défaillance du logiciel est inacceptable.

Lady Augusta Ada Byron, comtesse de Lovelace [1], naquit à Londres le 10 décembre 1815. Effrayée à l'idée qu'elle puisse suivre l'exemple de son père, le grand poète Lord Byron, la mère d'Augusta Ada décida de lui enseigner les sciences en général et les mathématiques en particulier.

En 1933, Lady Ada rencontre Charles Babbage, concepteur de la machine analytique universelle, un assemblage extraordinaire de rouages et engrenages généralement considéré comme le premier ordinateur, même si la machine ne fut jamais achevée. Ada travailla avec l'inventeur à la mécanisation et au perfectionnement des métiers à tisser.

Durant les années 1840, elle proposa une méthode pour calculer les nombres de Bernoulli avec la machine de Babbage : historiquement, il s'agit là du premier programme informatique, même s'il ne portait pas encore ce nom (certains historiens estiment par ailleurs que le programme fut en réalité écrit par Babbage, et « seulement » corrigé par Ada). À ce titre, Lady Augusta Ada Lovelace Byron est considérée comme le premier programmeur de l'Humanité. Elle s'éteignit le 27 novembre 1852 d'un cancer, laissant derrière elle trois enfants, et fut inhumée aux cotés de son poète de père qu'elle ne connut jamais.

Ada 83

Durant les années 1970, le Département de la Défense des États-Unis d'Amérique s'inquiéta de la prolifération des langages de programmation utilisés pour ses besoins : plusieurs centaines furent recensés, certains complètement propriétaires et fermés, d'autres tout simplement dépassés. Un groupe fut chargé d'établir une liste de spécifications qu'un langage de programmation devrait satisfaire pour être utilisé dans le cadre de l'armée des États-Unis : aucun langage existant alors ne remplissait toutes les contraintes. Aussi un concours fut-il organisé en 1977 pour la création d'un nouveau langage.

Plusieurs propositions furent soumises, désignées par des couleurs. En 1979, la « proposition Verte », présentée par Jean Ichbiah (un français) de Honeywell Bull (société française), est retenue par le département de la défense. Le langage est nommé Ada en hommage à Lady Ada Lovelace Byron et son manuel de référence est approuvé le 10 décembre 1980, anniversaire de la naissance de Lady Ada. Le langage devint un standard ANSI en 1983, référencé ANSI/MIL-STD 1815 (MIL pour *military*, et 1815 pour l'année de naissance de Lady Ada) [2], puis un standard ISO en 1987, référencé ISO-8652:1987. Cette version du langage est communément désignée par « Ada83 ». Les caractéristiques de cette version sont, brièvement :

- Un typage fort, et même très strict : il est par exemple impossible d'affecter un entier à une variable réelle sans passer par une conversion explicite ;
- Contrôle à l'exécution : de nombreux tests sont effectués à l'exécution, pour détecter par exemple des dépassements de capacité pour les types entiers ou des conversions de types invalides ;
- Support des exceptions ;
- Support de la généricité, pour utiliser des types inconnus : il s'agit de l'équivalent des *templates* du langage C++ ;
- Support du parallélisme, pour l'exécution simultanée de plusieurs tâches d'un même programme ; on parle plutôt aujourd'hui de *multi-threading*, mais tandis que la plupart des langages nécessitent l'utilisation de bibliothèques externes pour le réaliser, Ada possède dans sa définition même les structures et mots-clefs nécessaires ;
- Facilités pour la création de bibliothèques d'outils (paquetages), afin d'atteindre une bonne réutilisation du code ;
- Une syntaxe claire, inspirée du langage Pascal.

Ces caractéristiques font de Ada un langage particulièrement robuste, utilisé dans des domaines critiques comme l'aérospatial ou l'armement de pointe. Dans la pratique, on peut dire qu'un programme en Ada est généralement plus difficile à écrire que son équivalent dans un autre langage plus répandu, par contre le résultat est presque toujours considérablement plus fiable. Autre aspect intéressant, ne peut se dénommer « compilateur Ada » qu'un compilateur respectant précisément la norme définie. On évite ainsi de nombreux maux de têtes engendrés par des compilateurs plus ou moins compatibles, qui respectent plus

ou moins les standards, comme c'est trop souvent le cas pour des langages comme C/C++ ou même Java.

Ada 95

Le langage connut une évolution majeure en 1995, avec l'acceptation en février 1995 du standard ISO-8652:1995 [3]. En plus d'un dépoussiérage, cette nouvelle version introduit dans le langage la programmation orientée objet, avec héritage simple et polymorphisme, ainsi que quelques annexes décrivant les possibilités offertes pour les systèmes temps réel, les architectures distribuées ou la programmation système. C'est sur cette version que nous allons nous appuyer dans ce qui suit. Incidemment, Ada95 est le premier langage de programmation orienté objet faisant l'objet d'une normalisation. La norme C++, par exemple, ne sera publiée que trois ans plus tard.

GNAT et Ada 2005

Pour faciliter l'évolution du standard et le développement du langage Ada, en 1992, l'armée de l'air des États-Unis (US Air Force) finança le développement d'un compilateur Ada en source ouvert (*open source*) : le compilateur GNAT (pour GNU Ada Translator), basé sur GCC, et dont le développement est assuré par la société Ada Core Technologie [4] [5] [10]. Depuis 2001, le compilateur GNAT est intégré à GCC.

Précisons qu'il existe actuellement trois versions de GNAT :

- GNAT Pro, compilateur commercial activement maintenu ;
- La version publique de GNAT, non maintenue mais de qualité industrielle ;
- La version de la Free Software Foundation, intégrée à GCC et suivant l'évolution de ce dernier.

Je m'appuierai sur la version de la Free Software Foundation, plus précisément celle intégrée à GCC 3.4 : les versions antérieures de GCC ne sont en effet pas considérées comme finalisées, du moins du point de vue de Ada. Pour l'essentiel, vous pouvez également utiliser la version publique, numérotée 3.15p, mais sachez que cette version n'est plus maintenue. Si vous utilisez une distribution Debian, la version publique est contenue dans le package `gnat`, la version FSF dans le package `gnat-3.3` (pour GCC 3.3) ou `gnat-3.4` (pour GCC 3.4).

Enfin, l'année 2005 devrait voir l'apparition d'une révision du langage Ada.

« Personne n'utilise Ada ! »

Voilà un commentaire que l'on entend régulièrement. S'il est vrai que son usage n'est pas très répandu chez les éditeurs de logiciels « classiques », rappelons que sa création a été l'initiative de l'armée des États-Unis d'Amérique : il est donc très utilisé par cette institution. Le langage Ada est utilisé par nombre d'entreprises et d'institutions, dans des situations où la fiabilité est la principale contrainte. Les avions Airbus A340 ou Boeing 777 utilisent largement Ada. Ainsi que le TGV français ou la ligne 14 du métro parisien (ligne automatique, sans conducteur). On le trouve également dans certaines institutions bancaires. La sonde Huygens, qui vient tout juste de renvoyer d'extraordinaires informations du sol de Titan, lune de Jupiter, embarque du code Ada. Le compilateur GNAT lui-même est écrit en Ada ! Pour plus de détails, consultez la page référencée en [8]. Moralité, « personne » représente quand même pas mal de monde...

Premier programme

Sacrifions à la tradition. Voici le « Bonjour, Monde ! » en Ada.

```
1 with Text_IO ;
2 use Text_IO ;
3 procedure bonjour is
4 begin
5   put_line("Bonjour, Monde !") ;
6 end bonjour ;
```

Stockez ce programme dans un fichier nommé `bonjour.adb`. Attention, le nom du fichier a son importance ! Pour le compiler, utiliser l'outil `gnatmake` fourni avec Gnat, puis exécutez-le :

```
$ gnatmake bonjour.adb
gnatgcc -c bonjour.adb
gnatbind -x bonjour.ali
gnatlink bonjour.ali
$ ./bonjour
Bonjour, Monde !
$
```

Examinons dans le détail ce modèle de trivialité. La première ligne signale que l'on va utiliser le contenu du paquetage `Text_IO`. Dans le verbiage Ada, un paquetage est ce que l'on appelle une bibliothèque en C/C++ ou un module en Python. `Text_IO` contient tout un ensemble de fonctions liées aux entrées-sorties de textes : la première ligne serait donc `#include <stdio.h>` en C ou `#include <iostream>` en C++.

Un paquetage ainsi référencé définit une sorte d'espace de nommage qui lui est propre. Python utilise une technique similaire pour ses modules et on peut rapprocher cela des *namespaces* du C++. La clause `use` utilisée en deuxième ligne permet de rendre le contenu du paquetage directement visible, sans qu'il soit besoin d'utiliser un préfixe. Réécrite en Python, cette ligne deviendrait `from Text_IO import *` ou encore `using namespace Text_IO` en C++. Arrêtons-nous un instant, pour signaler un aspect très important de la syntaxe de Ada. Contrairement à la plupart des langages répandus aujourd'hui, Ada n'est pas sensible à la casse des caractères – aucune différence n'est faite entre majuscules et minuscules. Ainsi, la première ligne aurait tout aussi bien pu être écrite `WitH teXT_iO`. Cela n'aurait fait aucune différence. Au début, c'est assez déroutant, surtout si on ne pratique pas couramment le Pascal ou le Basic. Par ailleurs, l'indentation est libre en Ada. Notre petit programme aurait pu être écrit sur une seule ligne. Il y a toutefois une limite, la longueur de la ligne : le standard Ada précise qu'un compilateur doit accepter des lignes et des identificateurs (noms de variables ou fonctions) jusqu'à au minimum 200 caractères (section 2.2, paragraphe 14), mais n'impose pas de maximum. Par conséquent, pour une portabilité maximale, ne dépassez pas 200 caractères par ligne. Continuons. La troisième ligne identifie ce que nous appellerons notre procédure principale. En C et C++, il s'agit d'une fonction toujours nommée *main* d'un type bien défini. En Ada, il s'agit d'une procédure d'un nom quelconque, ne prenant aucun argument. Petite contrainte, liée au compilateur Gnat : le fichier qui contient la procédure principale doit être nommé du nom de cette procédure, avec l'extension adb. Ici, notre procédure s'appelle bonjour, donc le fichier contenant notre programme doit être nommé `bonjour.adb`.

Pour délimiter les blocs, Ada n'utilise ni les accolades du C/C++/Perl, ni l'indentation de Python. On retrouve le vieux couple `begin/end`, qui existe également en Pascal. Notez que le nom de la procédure est rappelé après le `end` final, ligne 6 : ce n'est pas absolument nécessaire, mais facilite

la relecture. Par contre, si vous changez le nom de la procédure en ligne 3, changez-le également en ligne 6 : si les deux noms ne correspondent pas, vous aurez une erreur de compilation. Finalement, la seule instruction qui fait réellement quelque chose est en ligne 5. `put_line()` est une procédure du paquetage `Text_IO`, qui a pour seule fonction d'afficher la chaîne de caractères donnée en paramètre suivie d'un saut de ligne. Dans d'autres langages, cela pourrait être :

- En C : `printf("Bonjour, Monde !\n")` ;
- En C++ : `std::cout << "Bonjour, Monde !" << std::endl` ;
- En Python : `print 'Bonjour, Monde !'`

Si nous n'avions pas utilisé la clause `use` en ligne 2, l'instruction deviendrait :

```
5   Text_IO.put_line("Bonjour, Monde !") ;
```

Une écriture familière pour ceux qui pratiquent Python. Voilà pour la tradition, passons à autre chose.

Types fondamentaux et variables

Le standard impose l'existence des types suivants :

- `boolean`, type booléen prenant les deux valeurs `true` et `false` ;
- `character`, pour les caractères standards ISO 8859-1 – en gros, les caractères ASCII plus quelques caractères accentués, correspondant aux caractères latin-1 (caractères codés sur 8 bits) ;
- `wide_character`, pour les caractères du code ISO 10646, aussi appelé UCS, correspondant à l'Unicode (caractères codés sur 16 bits) ;
- `integer`, type entier signé défini au minimum dans l'intervalle de $-2^{15}+1$ à $2^{15}-1$, mais qui peut être plus large ;
- `float`, type réel en virgule flottante supportant au minimum six chiffres significatifs (mais peut-être plus) ;
- `string`, pour les chaînes de caractères « normales », en fait un tableau de `characters` ;
- `wide_string`, pour les chaînes de caractères UCS, en fait un tableau de `wide_characters`.

Toutefois, selon la plate-forme sur laquelle opère le compilateur et notamment le processeur sur lequel est censé tourner le programme, d'autres types peuvent être fournis. Les deux donnés ici sont très répandus :

- `long_integer`, entiers longs signés, au minimum dans l'intervalle de $-2^{31}+1$ à $2^{31}-1$;
- `long_float`, type réel en virgule flottante supportant au minimum 11 chiffres significatifs.

Il est intéressant et important de noter que le standard ne définit véritablement que des contraintes minimales. Par exemple, pour une plate-forme donnée, le standard n'interdit pas le type `integer` de couvrir l'intervalle autorisé par une représentation sur 128 bits. Par ailleurs, ne sont donnés ici que les types les plus fondamentaux et élémentaires (à l'exception de `string` et `wide_string`, qui sont des tableaux) : d'autres types existent, comme les nombres décimaux à virgule fixe, que nous rencontrerons par la suite. Enfin, curieusement ces types ne sont pas des mots-clefs du langage : ils sont définis dans un paquetage nommé `Standard`. Inutile de requérir l'utilisation de `Standard` avec la clause `with`, comme nous l'avons fait pour le paquetage `Text_IO` : il est toujours disponible et son contenu directement accessible.

La déclaration d'une variable, car en Ada les variables doivent être déclarées avant d'être utilisées, se fait ainsi :

```
mon_entier: integer ;
```

On est donc à l'inverse du C/C++ : ici, on donne d'abord le nom de la variable, puis son type. Pour initialiser la variable avec une valeur :

```
mon_entier: integer := 1 ;
```

Et enfin, si cette variable doit être une constante :

```
mon_entier: constant integer := 1 ;
```

Vous l'aurez compris, en Ada l'opérateur d'affectation est repris du Pascal : il s'agit du symbole composé `:=`.

Ada propose diverses possibilités pour l'écriture des entiers :

```
e1: integer := 1_000_000 ;
e2: integer := 16#12AB# ;
e3: integer := 7#456# ;
```

La première écriture montre l'utilisation du caractère de soulignement pour séparer les groupes de chiffres dans un nombre : cela permet de faciliter la lecture, mais cela n'a rien d'obligatoire. La deuxième montre comment donner un nombre en base 16 (hexadécimale) : il suffit d'encadrer le nombre par des dièses `#`, et de faire précéder le tout de la base voulue - 16 pour l'hexadécimal. Voyez la troisième ligne, où un nombre est donné en base 7. Naturellement, si vous donnez un nombre incohérent avec la base demandée (par exemple, un chiffre plus grand que 6 en base 7), vous serez gratifié d'une erreur de compilation. Pour ce qui est des nombres à virgule flottante, Ada utilise la notation usuelle. Les caractères sont quant à eux donnés comme en C, entre apostrophes. Je n'en dirai pas plus pour l'instant, les chaînes de caractères méritant un traitement spécial.

Les attributs

Chaque type en Ada possède un certain nombre de caractéristiques ou fonctions intrinsèques, appelées attributs. Ce ne sont pas des fonctions en tant que telles. On ne peut pas vraiment les rapprocher de données ou méthodes membres d'une classe. Un attribut est désigné en faisant précéder son nom d'une apostrophe, elle-même précédée d'un nom de type (ou d'un objet, selon les cas). Avant de trop m'embrouiller dans une définition difficile à exprimer, voyons quelques exemples.

- `integer'first` va donner un entier égal à la plus petite valeur que peut prendre le type `integer` ;
- `float'last` va donner un nombre réel égal à la plus grande valeur que peut prendre le type `float` ;
- `integer'size` donne le nombre de bits (et non d'octets) occupés par une variable de type `integer`.

Les attributs `first`, `last` et `size` sont applicables à tous les types élémentaires que nous avons rencontrés. Si ceux-ci ressemblent à de simples variables, il en est d'autres qui prennent la forme d'appels de fonctions :

- `integer'min(a, b)` retourne le plus petit des deux entiers a et b ;
- `float'max(x, y)` retourne le plus grand des deux réels x et y ;
- `integer'value("123")` retourne un entier résultant de la conversion de la chaîne de caractères ; cela fonctionne également avec la notation spéciale pour choisir une base, par exemple

`integer'value("16#FF#")` retourne l'entier 255 ;

`float'image(3.14)` est la réciproque du précédent et retourne une chaîne de caractères représentant la valeur donnée en paramètre. Nous verrons plus tard les possibilités pour afficher autre chose que des chaînes de caractères. Mais pour l'heure, l'attribut précédent nous permet d'afficher simplement des valeurs numériques :

```
 1 with Text_IO ;
 2 use Text_IO ;
 3 procedure attributs is
 4   i_min: integer := integer'first ;
 5   i_max: integer := integer'last ;
 6   f_min: float := float'first ;
 7   f_max: float := float'last ;
 8 begin
 9   put_line("i_min = " & integer'image(i_min)) ;
10   put_line("i_max = " & integer'image(i_max)) ;
11   put_line("f_min = " & float'image(f_min)) ;
12   put_line("f_max = " & float'image(f_max)) ;
13 end attributs ;
```

Le résultat de ce programme, qui peut varier selon votre architecture, est le suivant :

```
$ gnatmake attributs.adb
gnatgcc -c attributs.adb
gnatbind -x attributs.ali
gnatlink attributs.ali
$ ./attributs
i_min = -2147483648
i_max =  2147483647
f_min = -3.40282E+38
f_max =  3.40282E+38
```

L'attribut `image` ne permet pas de mise en forme précise, comme le nombre de chiffres significatifs. Mais encore une fois, nous verrons cela plus tard.

Sous-types et types dérivés

Il est assez courant de définir des types personnalisés à partir des types de base, même pour les types élémentaires. En C/C++, cela se fait usuellement à l'aide de l'instruction `typedef`, qui ne fait guère plus que définir un synonyme (dans ce cas très limité). En Ada, par contre, de tels types personnalisés sont à la base de la robustesse des programmes : ils font bien plus que simplement définir un nouveau nom. Supposons que dans un programme, nous ayons à manipuler des dimensions, certaines exprimées en millimètre, d'autres en pouce (unité de mesure obsolète fort appréciée de nos amis anglo-saxons).

Il semble évident qu'additionner sans précaution des valeurs en millimètre avec des valeurs en pouce conduirait à des résultats aberrants. Voyons trois programmes minimalistes réalisant cette opération, le premier en C++, les deux suivants en Ada (*voir tableau Programmes*).

Le programme C++ compile sans aucun problème et affichera la valeur 3, dont on peut bien se demander quelle est son unité. Le premier programme Ada, qui utilise les sous-types, compilera également sans rien de particulier et affichera également la valeur 3. Par contre, le troisième programme ne compilera tout simplement pas :

```
$ gnatmake ada_types_derives.adb
gnatgcc -c ada_types_derives.adb
ada_types_derives.adb:9:25: invalid operand types for operator "+"
ada_types_derives.adb:9:25: left operand has type "d_mm" defined at line 4
ada_types_derives.adb:9:25: right operand has type "d_in" defined at line 5
gnatmake: "ada_types_derives.adb" compilation error
```

Programmes

Programme 1

```
#include <iostream>
typedef float d_mm ;
typedef float d_in ;
int main (int argc, char* argv[])
{ d_mm a = 1.0 ;
  d_in b = 2.0 ;
  std::cout << a+b << "\n" ;
  return 0 ;
}
```

Programme 2

```
with Text_IO ;
use Text_IO ;
procedure ada_sous_types is
  subtype d_mm is float ;
  subtype d_in is float ;
a: d_mm := 1.0 ;
b: d_in := 2.0 ;
begin
  put_line(float'image(a+b)) ;
end ada_sous_types ;
```

Programme 3

```
with Text_IO ;
use Text_IO ;
procedure ada_types_derives is
  type d_mm is new float ;
  type d_in is new float ;
a: d_mm := 1.0 ;
b: d_in := 2.0 ;
begin
  put_line(float'image(a+b)) ;
end ada_types_derives ;
```

Si dans le premier programme Ada, `d_mm` et `d_in` sont des sous-types (*subtype*) du type `float`, dans le deuxième, ce sont bel et bien deux types différents, construits à partir d'un même type ! Dans le premier cas, ils restent des `float` et peuvent donc être combinés. Dans le deuxième, ils ont chacun leur identité propre. Une telle « dureté » de typage est inconnue du C++ et, en fait, de la plupart des langages de programmation. Cela peut paraître une contrainte supplémentaire et désagréable, mais la technique des *types* dérivés permet d'éviter de nombreuses erreurs de logique dans les calculs (celles-ci sont détectées par le compilateur) et confère donc une grande robustesse aux programmes Ada qui en font usage.

Contrôles à l'exécution

Toutes les erreurs ne peuvent être détectées à la compilation. Mais contrairement à d'autres langages qui laissent passer pas mal de choses, Ada permet de détecter de nombreuses sources de problèmes et de prévoir une réaction adaptée.

Comme exemple, considérez le petit programme suivant :

```
1 procedure test is
2 i: integer ;
3 f: float := 1.0E+35 ;
4 begin
5   i := integer(f) ;
6 end ;
```

Rien de bien terrible : une valeur en virgule flottante est explicitement convertie en un entier, à l'aide d'un *transtypage*, ligne 5. Syntaxiquement, tout est correct... sauf que la valeur de la variable `f` est trop grande pour être stockée dans la variable `i`, qui va alors prendre une valeur probablement incohérente. Mais cela, le compilateur ne peut pas le détecter. Avec un langage comme le C, il n'existe aucun moyen de se prémunir de ce genre d'erreurs. Ada permet de le faire, en activant le contrôle de débordement à la compilation au moyen du paramètre `-gnato` passé à `gnatmake` :

```
$ gnatmake -gnato test.adb
gnatgcc -c -gnato test.adb
gnatbind -x test.ali
gnatlink test.ali
gnatlink: warning: executable name "test" may conflict with
shell command
$ ./test
raised CONSTRAINT_ERROR : test.adb:5 overflow check failed
```

Remarquez l'avertissement qui signale que le nom du programme est le même que celui d'une commande de l'interpréteur de commande. Mais plus intéressante est l'exécution de notre programme. Au moment de l'affectation, le système de contrôle détecte un dépassement de capacité et provoque alors une exception. Ici, aucun traitement n'a été prévu dans cette situation (nous verrons plus tard la gestion des exceptions), aussi le programme se termine-t-il immédiatement. Mais dans une situation réelle, nous pourrions intercepter l'exception et réagir en conséquence. Naturellement, l'activation de ces contrôles à l'exécution génère des exécutables plus volumineux et plus lents. Mais ils sont précieux en phase de test et, dans une situation critique, il n'est pas recommandé de les désactiver.

Ariane 5 Vol 501

Si vous pensez que ce genre de technique est bien lourde, bien inutile et contraignante, permettez-moi d'évoquer le cas du vol 501 d'Ariane 5, le premier vol de test de la fusée Ariane 5, le 4 juin 1996. Ce premier vol, annoncé en fanfare, se conclut après quelques secondes par la destruction de la fusée devenue incontrôlable. Après enquête, il fut découvert que le responsable était une petite portion de code, justement écrite en Ada, reproduite ici (d'après [7]) :

```
1 -- ...
2 declare
3   vertical_veloc_sensor: float;
4   horizontal_veloc_sensor: float;
5   vertical_veloc_bias: integer;
6   horizontal_veloc_bias: integer;
7 --  ...
 8 begin
 9   declare
10     pragma suppress(numeric_error, horizontal_veloc_bias);
11   begin
12     sensor_get(vertical_veloc_sensor);
13     sensor_get(horizontal_veloc_sensor);
14     vertical_veloc_bias := integer(vertical_veloc_sensor);
15     horizontal_veloc_bias := integer(horizontal_veloc_
sensor);
16 --    ...
17   exception
18     when numeric_error => calculate_vertical_veloc();
19     when others => use_irs1();
20   end;
21 end irs2;
```

Pour information, les lignes qui commencent par deux tirets (1, 7 et 16) sont des commentaires : le double tiret est le symbole utilisé par Ada pour commencer un commentaire, qui se poursuit jusqu'à la fin de la ligne.

Sans entrer dans les détails, remarquez que la variable `horizontal_veloc_sensor` est de type réel, tandis que la variable `horizontal_veloc_bias` est de type entier. Ligne 15, la valeur de la première est placée dans la seconde, par l'intermédiaire d'un transtypage. Ce code a été récupéré d'Ariane 4, et certainement insuffisamment testé dans le cadre d'Ariane 5 : la poussée produite par Ariane 5 est très supérieure à celle d'Ariane 4.

En fait, durant le vol d'essai cette portion de code s'est trouvée dans la même situation que mon petit exemple de cinq lignes : la valeur réelle est devenue trop grande pour pouvoir être stockée dans la variable entière. Pourtant, cette portion de code prévoit un traitement des exceptions, lignes 17 à 20, justement destinée à gérer ce type de problème. Seulement, une directive du langage a été utilisée pour supprimer explicitement le contrôle des erreurs numériques pour la variable `horizontal_veloc_bias` : c'est le sens de la ligne 10. On peut s'interroger sur les motivations de l'écriture de cette ligne, probablement des considérations de vitesse d'exécution. Toujours est-il qu'au moment de l'exécution de la ligne 15, tout se passe comme si le programme n'avait pas été compilé avec le paramètre `-gnato` évoqué plus haut : un débordement de capacité survient, mais il est superbement ignoré.

D'après les rapports d'enquête, ce dépassement est survenu 36.7 secondes après le décollage. Un peu plus de deux secondes plus tard, la fusée était détruite. Coût estimé : environ deux milliards de francs de l'époque, soit environ 300

millions d'euros.Ce que l'on peut retenir de cette expérience, entre autres, c'est que bien que le langage offre toutes les facilités pour prévenir et éviter la catastrophe, la désactivation forcée d'une sécurité précise, associée à des tests insuffisants, a suffi à transformer un engin spatial exceptionnel en poussière de métal.

Conclusion

Voilà pour cette rapide présentation du langage Ada. J'espère qu'elle vous aura intéressé et vous donnera l'envie de poursuivre dans l'apprentissage de ce langage. Vous aurez remarqué que cet article est parsemé de comparaisons entre Ada et d'autres langages et peut-être n'aurez-vous pas apprécié certains commentaires.

Alors, abordons brièvement le délicat sujet du choix d'un langage de programmation. Il est probablement impossible de dire qu'un langage est meilleur qu'un autre dans toutes les situations. Chaque langage a été défini dans un contexte particulier, avec une certaine idée préconçue de son utilisation, qui diffère parfois de la réalité d'utilisation du langage dans la pratique. Forces et faiblesses d'un langage donné trouvent leur origine dans les objectifs considérés comme prioritaires de ses concepteurs.

Sans doute le langage Ada est-il plus contraignant qu'un autre. Si vous voulez écrire rapidement un programme prototype d'une idée fulgurante, Python est bien indiqué. Si vous dispose d'un peu plus de temps, et que vous voulez avoir assez vite un programme très rapide à l'exécution, alors C ou C++ sont probablement de bons choix. Si le programme doit être le plus solide possible, tolérant aux erreurs, et impliquer des sommes importantes ou des vies humaines, alors Ada est sûrement la voie à emprunter. Ce ne sont là que quelques aspects qui motivent le choix d'un langage plutôt que d'un autre.

Il en existe de nombreux autres qui peuvent faire pencher la balance. Il me semble que l'important est de connaître plusieurs langages, pour pouvoir effectuer le meilleur choix ou le choix le moins mauvais, quand la question se pose. C'est dans cet esprit que je souhaite vous présenter Ada. Et pour cela, nous réaliserons, au fil du temps, un petit programme. Et pour suivre ce qui va peut-être devenir une autre tradition dans ces pages, nous nous attaquerons une fois de plus aux sempiternelles Tours de Hanoï. Hé si !

Références

- [1] Lady Augusta Ada Lovelace Byron : http://en.wikipedia.org/wiki/Ada_Lovelace
- [2] Ada 83 (ANSI/MIL-STD 1815) : http://archive.adaic.com/standards/83lrm/html/Welcome.html
- [3] Ada 95 (ISO-8652:1995) : http://www.adaic.org/standards/95lrm/html/RM-TTL.html
- [4] Ada Core Technologies : http://www.gnat.com/
- [5] ACT Europe : http://www.act-europe.fr/
- [6] AdaPower : http://www.adapower.com/
- [7] Code Ariane 5, vol 501 : http://www-aix.gsi.de/~giese/swr/ariane5.html
- [8] Who's using Ada : http://www.seas.gwu.edu/~mfeldman/ada-project-summary.html
- [9] AdaWorld : http://adaworld.com/
- [10] Version publique de GNAT et autres outils : http://libre.act-europe.fr/