Premiers programmes Ada
Corrigé

Notions acquises à l'issue du TP :

- Savoir compiler et exécuter un programme ADA.
- Comprendre l'intérêt d'un compilateur
- Comprendre les entrée-sorties en Ada
- Savoir choisir la « bonne » structure de contrôle dans un algorithme.

Exercice 1 : Compiler et exécuter un programme Ada.

Un programme Ada est écrit dans un fichier dont l'extension est `.adb`, comme par exemple `premier_programme.adb` (listing 1). Ce programme est dans le dossier `tp1`.

Nous utiliserons GNAT (GNU Ada), le compilateur Ada du projet GNU, pour traduire un programme Ada en un exécutable. Il suffit de taper dans un terminal (il faut bien sûr être dans le dossier qui contient le fichier à compiler) :

```
> gnatmake -gnatwa premier_programme.adb
```

```ada
with Text_IO;
use Text_IO;

-- Programme minimal qui affiche juste un message.
procedure Premier_Programme is
begin
    Put_Line ("Bravo ! Vous avez réussi à exécuter le programme.");
end Premier_Programme;
```

Listing 1 – Le fichier Ada `premier_programme.adb`

Cette commande produit un fichier exécutable nommé `premier_programme` [1] (le nom du fichier contenant le source Ada sans l'extension `.adb`). Pour l'exécuter, on fait :

```
> ./premier_programme
```

La commande `gnatclean` efface les fichiers engendrés.

```
> gnatclean premier_programme
```

Attention : L'option `-gnatwa` demande au compilateur de signaler davantage de messages d'avertissement. Même s'ils n'empêchent pas le compilateur d'engendrer un exécutable, les avertissements correspondent généralement à des erreurs ou des maladresses qu'il faut corriger.

Exercice 2 : Intérêt d'un compilateur et du typage statique

Les fichiers `min_max_serie.py` (listing 2) et `min_max_serie.adb` (listing 3) contiennent respectivement un programme Python et Ada affichant le plus petit et le plus grand élément d'une série d'entiers naturels. L'utilisateur indique la fin de la série avec 0 qui ne fait pas partie de la série.

1. Exécuter le programme Python [2] en tapant les entiers 2 9 3 6 3 et 0. Quelles erreurs sont signalées ? Les corriger et recommencer.

Solution : Python refuse d'exécuter ce programme car il contient une erreur de syntaxe (`SyntaxError`) : il manque un « : » pour le `if`.

Une fois cette correction faite, on peut exécuter le programme et, sur les données du sujet, il affiche 2 pour le min et 9 pour le max qui sont bien les résultats attendus.

2. Compiler le programme Ada. Quelles erreurs sont signalées ? Les corriger et recompiler.
Ces erreurs étaient aussi présentes dans le programme Python. Avaient-elles été détectées ? Dans la négative, comment les mettre en évidence ?

Solution : Le compilateur Ada commence pas signaler le `then` manquant (l'équivalent du « : » manquant en Python). Une fois cette erreur corrigée, la compilation signale un nouveau message d'erreur : la variable `Enlier` n'est pas connue. Le compilateur suggère qu'il pourrait s'agir de `Entier` ce qui est effectivement le cas. Après cette deuxième correction le programme compile sans erreur.

L'erreur sur `enlier` au lieu `entier` n'a pas été détectée avec l'exécution faite en Python car la série saisie n'a pas permis d'exécuter cette instruction. Il faudrait avoir une série qui fait changer le min, par exemple : 2, 1, 0.

3. Indiquer l'intérêt du typage statique et du compilateur.

Solution : Le programmeur a donné des indications sur ses choix de conception que le compilateur (ou autre outil d'analyse statique) peut utiliser pour détecter des erreurs. Toute erreur signalée par le compilateur est un problème dans le programme qu'il suffit de corriger. Pas besoin de trouver un exemple d'exécution qui le mettra en évidence.

Bien sûr, il faut faire des tests pour détecter les erreurs de logique...

1. D'autres fichiers sont engendrés. La commande `ls` permet de les lister.
2. Pour exécuter un programme Python depuis le terminal : `python3 min_max_serie.py`

```python
# Afficher le plus petit et le plus grand élément d'une série d'entiers
# naturels lus au clavier.  La saisie de la série se termine par 0
# (qui n'appartient pas à la série).
# Exemple : 2, 9, 3, 6, 3, 0 -> min = 2 et max = 9

# afficher la consigne
print('Saisir les valeurs de la série (0 pour terminer) :')

# saisir un premier entier
entier = int(input())

if entier == 0:  # entier n'est pas une valeur de la série
    print('Pas de valeurs dans la série')
else:   # entier est le premier élément de la série
    # initialiser min et max avec le premier entier
    min = entier
    max = entier

    # traiter les autres éléments de la série
    entier = int(input())   # saisir un nouvel entier
    while entier != 0:      # entier est une valeur de la série
        # mettre à jour le min et le max
        if entier > max    # nouveau max
            # mettre à jour le max avec entier
            max = entier
        elif entier < min:  # nouveau min
            # mettre à jour le min avec entier
            min = enlier
        else:
            pass

        # saisir un nouvel entier
        entier = int(input())

    # afficher le min et le max
    print('min =', min)
    print('max =', max)
```

Listing 2 – Le fichier Python `min_max_serie.py`

```
1  with Text_IO;
2  use Text_IO;
3  with Ada.Integer_Text_IO;
4  use Ada.Integer_Text_IO;
5
6  -- Afficher le plus petit et le plus grand élément d'une série d'entiers
7  -- naturels lus au clavier.  La saisie de la série se termine par 0
8  -- (qui n'appartient pas à la série).
9  -- Exemple : 2, 9, 3, 6, 3, 0 -> min = 2 et max = 9
10 procedure Min_Max_Serie is
11     Entier: Integer;    -- un entier lu au clavier
12     Min, Max: Integer;    -- le plus petit et le plus grand élément de la série
13
14 begin
15     -- Afficher la consigne
16     Put("Saisir les valeurs de la série (0 pour terminer) : ");
17
18     -- Saisir un premier entier
19     Get(Entier);
20
21     if Entier = 0 then --{ entier n'est pas une valeur de la série }
22         Put_Line("Pas de valeurs dans la série");
23     else   --{ Entier est le premier élément de la série }
24         -- initialiser Min et Max avec le premier entier
25         Min := Entier;
26         Max := Entier;
27
28         -- traiter les autres éléments de la série
29         Get(Entier);   -- saisir un nouvel entier
30         while Entier /= 0 loop      -- Entier est une valeur de la série
31             -- mettre à jour le Min et le Max
32             if Entier > Max    -- nouveau max
33                 -- mettre à jour le max avec Entier
34                 Max := Entier;
35             elsif Entier < Min then  -- nouveau Min
36                 -- mettre à jour le min avec Entier
37                 Min := Enlier;
38             else
39                 null;
40             end if;
41
42             -- saisir un nouvel entier
43             Get(Entier);
44         end loop;
45
46         -- afficher le min et le max de la série
47         Put("Min = ");
48         Put(Min, 1);
49         New_Line;
50
51         Put("Max = ");
52         Put(Max, 1);
53         New_Line;
54     end if;
55 end Min_Max_Serie;
```

Listing 3 – Le fichier Ada `min_max_serie.adb`

Exercice 3 : Comprendre les saisies d'entiers et de caractères

Le programme du listing 4 permet de saisir une longueur sous la forme d'une valeur (entier) et d'une unité (caractère). Par exemple `52c` correspond à la valeur 52 et l'unité c (pour centimètre).

1. Compiler ce programme. L'exécuter avec les entrées suivantes : `52cm`, `␣␣␣123mn` puis `1..n`.

Solution : Les résultats sont :

1. `52cm` : `52` pour Valeur et `c` pour Unité. Les caractères non utilisés sont : `m`.
2. `␣␣␣123mn` : `123` pour Valeur et `m` pour Unité. Les caractères non utilisés sont : `n`.
   Les caractères blancs sont ignorés pour la lecture d'un entier ou d'un réel.
3. `1..n` : `1` pour Valeur et `.` pour Unité. Les caractères non utilisés sont : `.n`.

Nous avons fait deux appels à `Get`, le premier sur `Valeur` de type entier et le second sur `Unite` de type caractère. Derrière ce même nom `Get`, il y a deux procédures différentes : c'est la *surcharge*. L'une est définie dans le paquetage `Ada.Integer_Text_Io`, l'autre dans `Ada.Text_Io`.

Les sous-programmes de lecture consomment les caractères que l'utilisateur saisit au clavier (sur l'entrée standard du programme) et les utilisent pour produire la valeur attendue. Pour la lecture d'un entier, les blancs sont ignorés et les chiffres sont assemblés pour produire l'entier. Par exemple, si l'utilisateur a saisi `␣52cm`, l'espace initial est ignoré, les caractères '5' et '2' sont reconnus comme faisant partie d'un entier, le caractère suivant 'c' ne fait pas partie de l'entier. La procédure `Get(Integer)` consomme donc l'espace, '5' et '2' pour reconnaître l'entier 52. Les caractères suivants restent sur l'entrée standard : 'c', 'm' et « retour à la ligne ». `Get(Character)` récupère le prochain caractère sur l'entrée standard qui n'est pas un retour à la ligne et le fournit à l'appelant. Le caractère 'c' sera utilisé les deux autres restant sur l'entrée standard.

Ce comportement permet à l'utilisateur de saisir plusieurs informations d'un coup comme ici une longueur (entier) suivie d'une unité (caractère) que l'utilisateur peut saisir comme `52c`.

Souvent, ce n'est pas le comportement souhaité : on préfère repartir avec une nouvelle saisie à chaque nouvelle sollicitation de l'utilisateur. Ceci évite que les caractères non consommés par la saisie précédente soient utilisés pour la saisie suivante. Pour ce faire, la procédure `Skip_Line` consomme les caractères de l'entrée standard jusqu'au prochain retour à la ligne inclus.

Remarque : la fonction `Get_Line`, utilisée dans le dernier `Put_Line`, retourne la chaîne constituée des caractères de l'entrée standard jusqu'au prochain retour à la ligne exclu mais consommé.

2. Décommenter les `Skip_Line` de `comprendre_get_skip_line.adb` (listing 4) et mettre en commentaire le dernier `Put_Line`. Compiler et exécuter les exemples précédents. Exécuter en saisissant `52ccc`, deux lignes vides et `mno`.

Solution : On devrait `52` pour valeur et `m` pour l'unité.

Attention : S'il n'y a pas de caractère en attente, et donc pas de retour à la ligne, `Skip_Line` attend une nouvelle saisie de l'utilisateur. Ceci ne se produira jamais si on le fait après un Get d'un entier ou d'un caractère.

Exercice 4 : Structures de contrôle

L'objectif de cet exercice est de manipuler les structures de contrôle de notre pseudo-langage algorithmique et leur pendant en Ada. Les programmes listés ci-après sont à compléter dans

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

-- Objectif : Comprendre le comportement de Get et Skip_Line.

-- Saisir une longueur (valeur et unité) et l'afficher.
procedure Comprendre_Get_Skip_Line is
    Valeur: Integer;    -- la valeur du longeur
    Unite: Character;    -- l'unité de la longueur : (c)entimètre, (m)ètre...
begin
    -- saisir la longueur
    Put("Longueur = ");
    Get(Valeur);
    -- Skip_Line;
    Get(Unite);
    -- Skip_Line;

    -- afficher la longueur
    Put("Valeur = ");
    Put(Valeur, 1);
    New_Line;
    Put_Line("Unité = >" & Unite & "<");

    -- afficher les caractères en attente sur l'entrée standard (fin de ligne)
    Put_Line("Reste de la dernière ligne saisie : " & '"' & Get_Line & '"');
end Comprendre_Get_Skip_Line;
```

Listing 4 – Le fichier Ada `comprendre_get_skip_line.adb`

l'ordre. Chaque programme commence par une description de son objectif et des exemples d'utilisation qui vous permettront de tester vos programmes. Il faudra faire attention à bien utiliser la bonne structure de contrôle.

On ne traitera pas la robustesse de ces programmes : on considère que l'utilisateur saisira toujours une donnée valide.

1. `permuter_caracteres.adb`
2. `tarif_place.adb`
3. `classer_caractere.adb`
4. `compte_jules_objectif.adb`
5. `chiffre_significatif.adb`
6. `table_7.adb`
7. `table_pythagore.adb`
8. `score_21.adb`
9. `somme_serie_double.adb`

Pour aller plus loin, pour les plus rapides ou ceux qui ont envie de s'entrainer voici quelques exercices supplémentaires et optionnels.

Exercice 5 : Racine carrée d'un nombre (Méthode de Newton)

La k^{ieme} approximation de la racine carrée de x est donnée par $a_{k+1} = (a_k + x/a_k)/2$ et $a_0 = 1$.

On arrête le calcul quand la distance entre a_{k+1} et a_k est inférieure à une précision donnée.

1. Écrire un programme (fichier `newton.adb`) qui affiche une valeur approchée de la racine carrée d'un nombre en utilisant la méthode précédente. Nombre et précision seront lus au clavier.

2. On peut aussi arrêter le calcul des a_k quand a_k^2 est proche de x à la précision près. Ajouter cette nouvelle approche dans le programme précédent.

Exercice 6 : Puissance

Afficher la puissance entière d'un réel en utilisant somme et multiplication (`puissance.adb`). On traiter d'abord le cas où l'exposant est positif avant de généraliser aux entiers relatifs.

Solution :

Dans la solution de cet exercice, nous utilisons la méthode des raffinages qui sera l'objet du prochain cours. Ceci permet de donner un exemple de son application. Pour la réponse à cet exercice, c'est le programme qui en découle qui est attendu.

On peut, dans un premier temps, se demander si la puissance entière d'un réel a toujours un sens. En fait, ceci n'a pas de sens si le nombre est nul et si l'exposant est strictement négatif.

```
2 puissance -3    ->  0.125 -- cas nominal (puissance négative)
```

On choisit de contrôler la saisie de manière à garantir que nous ne sommes pas dans l'un de ces cas. Nous souhaitons donner à l'utilisateur un message le plus clair possible lorsque la saisie est invalide.

Nous envisageons les jeux de tests suivants :

```
nombre exposant  ->    x^n

   2        3    ->      8 -- cas nominal
   3        2    ->      9 -- cas nominal
   1        1    ->      1 -- cas nominal
   2       -3    ->  0.125 -- cas nominal (exposant négative)
   0        2    ->      0 -- cas nominal (x est nul)
   0        0    ->      1 -- par convention
 1.5        2    ->   2.25 -- x peut être réel
   0       -2    -> ERREUR -- division par zéro
```

Voyons maintenant le principe de la solution. Par exemple, pour calculer 2^3, on peut faire $2 * 2 * 2$. Plus généralement, on multiplie n fois x par lui-même (donc une boucle **`Pour`**).

Si on essaie ce principe, on constate qu'il ne fonctionne pas dans le cas d'une puissance négative. Prenons le cas de 2^{-3}. On peut le réécrire $(1/2)^3$. On est donc ramené au cas précédent. Le facteur multiplicatif est dans ce cas $1/x$ et le nombre de fois est $-n$.

Nous introduisons donc deux variables intermédiaires qui sont :

```
facteur: Réel    -- facteur multiplicatif pour obtenir les puissances
                 -- successives de x
puissance: Réel -- abs(n).  On a : facteur^puissance = x^n
```

On peut maintenant formaliser notre raisonnement sous la forme du raffinage suivant.

```
R0 : Afficher la puissance entière d'un réel

R1 : Raffinage De « R0 »
      | Saisir avec contrôle les valeurs de x et n      x: out Réel; n: out Entier
      | { (x <> 0) Ou (n > 0) }
      | Calculer x à la puisssance n                    x, n: in ; xn: out Réel
      | Afficher le résultat                            xn: in Réel

R2 : Raffinage De « Saisir avec contrôle les valeurs de x et n »
      | Répéter
      |   | Saisir la valeur de x et n          x: out Réel; n: out Entier
      |   | Contrôler x et n                    x, n: in ; valide: out Booléen
      | JusquÀ valide                           valide: in

R1 : Raffinage De « Calculer x à la puisssance n »
      | Si x = 0 Alors
      |   | xn := 0
      | Sinon
      |   | Déterminer le facteur multiplicatif et la puissance
      |   |                     x, n: in ;
      |   |                     facteur out Réel ;
      |   |                     puissance: out Entier
      |   | Calculer xn par itération (accumulation)
      |   |                     n, facteur, puissance: in ; xn : out
      | FinSi

R3 : Raffinage De « Calculer xn par itération (accumulation) »
      | xn <- 1;
      | Pour i <- 1 JusquÀ i = puissance Faire
      |   | { Invariant : xn = facteur^i }
      |   | xn <- xn * facteur
      | FinPour
```

On peut alors en déduire l'algorithme suivant :

```
Algorithme puissance

    -- Afficher la puissance entière d'un réel

Variables
    x: Réel             -- valeur réelle lue au clavier
    n: Entier           -- valeur entière lue au clavier
    valide: Booléen     -- x^n peut-elle être calculée
    xn: Réel            -- x à la puissance n (en fait x^(i-1))
    facteur: Réel       -- facteur multiplicatif pour obtenir
                        -- les puissances successives
    exposant: Entier    -- abs(n).  On a facteur^exposant = x^n
    i: Entier           -- variable de boucle

Début
    -- Saisir avec contrôle les valeurs de x et n
    Répéter
        -- saisir la valeur de x et n
        Écrire("x = ")
        Lire(x)
        Écrire("n = ")
        Lire(n)
```

```
23
24          -- contrôler x et n
25          valide <- VRAI
26          Si x = 0 Alors
27              Si n = 0 Alors
28                  ÉcrireLn("x et n sont nuls. x^n est indéterminée.")
29                  valide <- FAUX
30              SinonSi n < 0 Alors
31                  ÉcrireLn("x nul et n négatif. x^n n'a pas de sens.")
32                  valide <- FAUX
33              FinSi
34          FinSi
35
36          Si Non valide Alors
37              ÉcrireLn("  Recommencez !")
38          FinSi
39      JusquÀ valide
40      { x <> 0 Ou n >= 0 }
41
42      -- Calculer x à la puisssance n
43      --   Déterminer le facteur multiplicatif et l'exposant
44      Si n >= 0 Alors
45          facteur <- x
46          exposant <- n
47      Sinon
48          facteur <- 1/x
49          exposant <- -n
50      FinSi
51
52      --   Calculer xn par itération (accumulation)
53      xn <- 1;
54      Pour i <- 1 JusquÀ i = exposant Faire
55          { Invariant : xn = facteur^i}
56          xn <- xn * facteur
57      FinPour
58
59      -- Afficher le résultat
60      ÉcrireLn(x, "^", n, " = ", xn)
61  Fin.
```

Exercice 7 : Amélioration du calcul de la puissance entière

Améliorer l'algorithme de calcul de la puissance(`puissance_mieux.adb`) en remarquant que :

$$x^n = \begin{cases} (x^2)^p & \text{si } n = 2p \\ (x^2)^p \times x & \text{si } n = 2p+1 \end{cases}$$

Ainsi, pour calculer 3^5, on peut faire $3 * 9 * 9$ avec bien sûr $9 = 3^2$.

Solution : Nous nous appuyons sur les mêmes variables que pour le calcul « naturel » de la puissance (exercice 6). Nous allons continuer à calculer x^n par accumulation. Nous avons l'invariant suivant :

$$x^n = \text{xn} * \text{facteur}^{\text{puissance}}$$

Lors de l'initialisation, cet invariant est vrai (`xn` vaut 1 et facteur et puissance sont tels qu'ils valent x^n).

D'après la formule donnée dans l'énoncé, à chaque itération de la boucle, deux cas sont à envisager :

— soit `puissance` est paire. On peut alors l'écrire $2 * p$. On a alors :

$$\begin{aligned} x^n &= \text{xn} * \text{facteur}^{\text{puissance}} \\ &= \text{xn} * \text{facteur}^{(2 * \text{p})} \\ &= \text{xn} * (\text{facteur}^2)^{\text{p}} \end{aligned}$$

On peut donc faire :

```
facteur <- facteur * facteur
puissance <- puissance Div 2    -- car p = puissance Div 2
```

On constate que l'invariant est préservé d'après les égalités ci-dessus et la puissance a strictement diminué (car divisée par 2).

— soit `puissance` est impaire. On peut alors l'écrire $2 * p + 1$. On a alors :

$$\begin{aligned} x^n &= \text{xn} * \text{facteur}^{\text{puissance}} \\ &= \text{xn} * \text{facteur}^{(2 * \text{p} + 1)} \\ &= \text{xn} * (\text{facteur} * \text{facteur}^{(2*\text{p})}) \\ &= (\text{xn} * \text{facteur}) * \text{facteur}^{(2*\text{p})} \end{aligned}$$

On peut donc faire :

```
xn <- xn * facteur
puissance <- puissance - 1
```

On constate que l'invariant est préservé d'après les égalités ci-dessus et la puissance a strictement diminué (car diminuée de 1).

On peut alors en déduire le raffinage suivant :

```
1  R3 : Raffinage De « Calculer xn par itération (accumulation) »
2       | xn <- 1;
3       | TantQue puissance > 0 Faire
4       |    | { Variant : puissance }
5       |    | { Invariant : x^n = xn * facteur^puissance }
6       |    | Si puissance Div 2 = 0 Alors          { puissance = 2 * p }
7       |    |  | puissance <- puissance Div 2
8       |    |  | facteur <- facteur * facteur
9       |    | Sinon                                 { puissance = 2 * p + 1 }
10      |    |  | puissance <- puissance - 1
11      |    |  | xn <- xn * facteur
12      |    | FinSi
13      | FinTQ
```