Sous-programmes
Corrigé

Objectifs

— Sous-programmes : savoir les spécifier, les implanter, etc.

Solution : Dans les solutions qui suivent, je n'ai pas repris la convention de préfixer les paramètres formels par `F_`.

Exercice 1 : Spécifier des sous-programmes

Pour chacun des énoncés suivants, donner la spécification du sous-programme correspondant.

1. Calculer la puissance entière d'un réel

   Solution :

   (a) **Objectif :** La puissance entière d'un réel

   (b) **Exemples :**
   - nominal puissance positive : $2^4 = 16$
   - nominal puissance nulle $5^0 = 1$
   - nominal puissance négative $2^{-3} = 0.125$
   - nominal puissance négative, nombre négatif $(-2)^{-3} = -0.125$
   - nominal avec un vrai nombre réel $(-1.2)^2 = 1.44$
   - nominal nombre négatif, puissance positive $(-3)^3 = -27$

   Remarque : ici je n'ai pas mis systématiquement des réels car en math en ne le fait pas. En Ada, il faudra le faire.

   Ceci permet de bien identifier les informations fournies au sous-programme : le nombre et l'exposant et les informations attendues la puissance.

   (c) **Paramètres :** (on identifie d'abord le rôle, le mode, le type puis le nom)
   - nombre : in Réel – – nombre réel
   - exposant : in Entier – – l'exposant
   - puissance : out Réel – – la puissance (nombre ** exposant)

   (d) **Type de sous-programme :** Fonction car un seul paramètre en sortie et les autres en entrée.

   Remarque : le nom du paramètre en sortie est un bon candidat pour le nom de la fonction !

(e) **Préconditions :** Non (x = 0 Et n <= 0) <=> x /= 0 Ou n > 0

(f) **Postcondition :** puissance == nombre ** exposant = Produit(nombre) n fois si n > 0...

```
-- La puissance entière d'un nombre réel
-- Paramètres
--       - nombre   : in  Réel   -- le nombre réel
--       - exposant : in  Entier -- l'exposant
-- Retourne
--       - puissance : out Réel  -- nombre à la puissance exposant
--
-- Nécessite  : x /= 0 Ou n > 0
-- Assure : pas si simple à exprimer !
-- Exemples :
--       On peut reprendre quelques uns de ceux qui sont donnés avant
Fonction Puissance (Nombre : in Réel; exposant : in Entier) retourne Réel
```

2. Saisir un entier au clavier.

Solution :

(a) **Objectif :** L'énoncé ci-dessus.

(b) **Exemples :**

— l'utilisateur tape sur les touches 1 2 5 3, l'entier est 1253.

— ...

—

(c) **Paramètres :**

— nombre : out Entier – l'entier saisi par l'utilisateur

(d) **Type de sous-programme :** on choisit de faire une procédure car appelé avec les mêmes entrées on pourra avoir un résultat différent.

Le fait que l'on fasse des E/S est aussi une indication d'une procédure.

(e) **Précondition :** —

(f) **Postcondition :** —

```
-- Saisir une entier.
--
-- Paramètre :
--       nombre : out Entier     -- l'entier saisi par l'utilisateur
--
-- Nécessite : ---
-- Assure :  ----
-- Exemples : 1 2 5 3 -> Nombre = 1253
Procédure Saisir(Nombre : out Entier);
```

Remarque : On aurait pu imaginer un paramètre supplémentaire (en in) correspondant à la consigne affichée à l'utilisateur quand il est sollicité.

3. Savoir si une année est bissextile

Solution :

Remarque : Il faudrait suivre la même démarche que dans les deux premières questions. Ici je ne donne que le résultat.

```
-- Est-ce qu'une année est bissextile ?
--
-- Paramètres :
--      - Année : in Entier
-- Retourne Booléen
--
-- Nécessite : Année > 0
-- Assure : (Année Mod 4 = 0) Et ((Année Mod 100 /= 0) Ou (Année Mod 400 = 0))
-- Exemples :
--      2017 --> FAUX -- n'est pas bissextile
--      2016 --> VRAI -- est bissextile
--      2020 --> VRAI -- est bissextile
--      1900 --> FAUX -- n'est pas bissextile
--      2000 --> VRAI -- est bissextile
Fonction Est_Bissextile (Année : in Entier) retourne Booléen
```

4. Calculer le pgcd de deux entiers strictement positifs

   Solution :

```
-- Le pgcd de deux entiers strictement positifs.
--
-- Paramètres :
--      a, b: in Entier
-- Retourne : Entier
--
-- Nécessite : a > 0 Et b > 0
-- Assure :
--      Résultat >= 1
--      a Mod Résultat = 0
--      b Mod Résultat = 0
--      « C'est le plus grand ! »
-- Exemples :
--  A < B : A = 10, B = 16 --> PGCD = 2
--  A = B : A = 10, B = 10 --> PGCD = 10
--  A > B : A = 21, B = 10 --> PGCD = 1
Fonction Pgcd (A, B : in Entier) retourne Entier
```

5. Obtenir le quotient et le reste d'une division entière

   Solution :

```
-- Le quotient et le reste d'une division entière
--
-- Paramètres :
--      Dividende: in Entier
--      Diviseur : in Entier
--      Quotient : out Entier
--      Reste : out Réel
--
-- Nécessite :
--      Diviseur >= 0
--      Diviseur > 0
--
-- Assure :
--      0 <= Reste
--      Reste < Diviseur
--      Dividende = Quotient * Diviseur + Reste
--
-- Exemples :
--      Nominal (reste non nul) : Dividende = 11, Diviseur = 4 -> Quotient = 2 et Reste = 3
```

```
--      Nominal (reste nul) : Dividende = 12, Diviseur = 4 -> Quotient = 3 et Reste = 0
Procédure Div_Mod (Dividende, Diviseur : in Entier ;
                   Quotient, Reste : out Entier)
```

Remarque : On pourrait aussi en faire une fonction si on pouvait renvoyer les deux valeurs (voir Enregistrement).

Remarque : On pourrait faire deux fonctions l'une appelée Quotient et l'autre Reste (mais on perd en efficacité car on fait deux fois les mêmes calculs).

6. Saisir un entier compris entre une borne inférieure et une borne supérieure. Avant chaque demande à l'utilisateur, une consigne lui est affichée pour lui expliquer ce qui est attendu.

   Solution :

   (a) **Objectif :** L'énoncé ci-dessus.

   (b) **Exemples :**

   — nominal un seul essai inf = 10, sup = 15, utilisateur : 10. Nombre = 10.

   — nominal trop petit : inf = 10, sup = 15, utilisateur 9, 16, 13. Nombre = 13

   — limite : borne inférieure choisie : inf = 10, sup = 15, utilisateur 10. Nombre = 10

   — limite : borne supérieure choisie : inf = 10, sup = 15, utilisateur 15. Nombre = 15

   —

   On pourrait ajouter la consigne et montrer précisément ce qui doit se passer. Ici l'idée est plutôt de spécifier l'interface du programme avec son utilisateur, en particulier en précisant les messages.

   (c) **Paramètres :**

   — inf : in Entier – borne inférieure

   — sup : in Entier – borne supérieure

   — consigne : in Chaine – le message à afficher à l'utilisateur

   — nombre : out Entier – l'entier saisi par l'utilisateur

   (d) **Type de sous-programme :** on choisit de faire une procédure car appelé avec les mêmes entrées on pourra avoir un résultat différent.

   Le fait que l'on fasse des E/S est aussi une indication d'une procédure.

   (e) **Précondition :** inf < sup

   (f) **Question :** faut-il prévoir une contrainte sur le message ?

   (g) **Postcondition :** inf <= nombre Et nombre <= sup

```
-- Saisir une entier compris entre une borne inférieure et une borne
-- supérieure.  Un message d'afficher la consigne à l'utilisateur.
--
-- Paramètre :
--      Inf, Sup: in Entier  -- borne dans lesquelles doit être l'entier saisi
--      Consigne: in Chaine  -- le message à afficher à l'utilisateur
--
```

```
8  -- Nécessite : Inf <= Sup
9  -- Assure :  Inf <= Nombre Et Nombre <= Sup
10 -- Exemple : Saisir
11 Procédure Saisir(Nombre : out Entier ; Inf, Sup: in Entier ; Consigne: in Chaine);
```

7. Ordonner dans l'ordre croissant les valeurs de trois caractères.
   Solution :
   (a) **Objectif :** Voir énoncé ci-avant.
   (b) **Exemples :**
   — 'X' 'C' 'T' -> 'C' 'T' 'X'
   (c) **Paramètres :** (on identifie d'abord le rôle, le mode, le type puis le nom)
   — C1, C2, C3 : in out Caractère – – trois caractères
   (d) **Type de sous-programme :** Procédure car des paramètres en in out (on change la valeur des paramètres).
   (e) **Préconditions :** —
   (f) **Postcondition :** – C1 <= C2 Et C2 <= C3 – variables ordonnées – les valeurs après constitue une permutation de valeurs avant

```
1  -- Ordonner dans l'ordre croissant les valeurs de trois caractères.
2  --
3  -- Paramètre :
4  --      C1, C2, C3: in out Caractère  -- 3 caractères à ordonner
5  --
6  -- Nécessite : ---
7  -- Assure :
8  --     C1 <= C2 Et C2 <= C3     -- ordonnées
9  --     -- les valeurs de C1, C2 et C3 sont une permutation de valeurs
10 --     -- précédentes de C1, C2 et C3.
11 Procédure Ordodnner(C1, C2, C3 : in out Caractère);
```

Exercice 2 : Combinaisons

Le nombre de combinaisons de p parmi m est donné par :

$$C_m^p = \frac{m!}{p!(m-p)!}$$

1. Spécifier un sous-programme calculant le nombre de combinaisons de p parmi m.

Solution :

```
-- Objectif : Obtenir le nombre de combinaisons de p parmi m, p et m entiers naturels
-- Paramètres :
--      - p, m : in Entier
-- Retourne : nombre de combinaisons
-- Nécessite :
--      p >= 0
--      m >= p
-- Assure : Résultat = factorielle(m) Div factorielle(p) Div factorielle(m - p)
Fonction Combinaisons(m, p : in Entier) Retourne Entier
```

2. En utilisant la méthode des raffinages, écrire le corps de ce sous-programme.

Solution :

Pour écrire le corps d'un sous-programme, on utilise la méthode des raffinage avec comme R0 la spécification du sous-programme.

```
R1 : Comment « Fonction Combinaisons... »
    Résultat <- factorielle de m Div factorielle de p Div factorielle de m - p
```

On constate qu'il faut calculer la factorielle de trois nombres : m, p et m - p.

Nous avons donc tout intérêt à définir un sous-programme qui calcule la factorielle d'un nombre, et donc à commencer par définir sa sémantique.

```
-- Objectif : Obtenir la factorielle de n entier naturel.
-- Paramètres :
--      - n : in Entier
-- Résultat : factorielle : out Entier
-- Précondtion : n >= 0
-- Poscondition : On pourrait la donner (voir code en Ada).
fonction factorielle (n : in Entier) Retourne Entier
```

Il nous faut aussi donner le corps de factorielle. On peut le faire en récursif ou en itératif.

Version itérative :

```
fonction factorielle (n : in Entier) Retourne Entier Est
    Variable
        Résultat: Entier
    Début
        Résultat <- 1
        Pour i De 2 À n Faire
            Résultat <- Résultat * i
        FinPour
        Retourne Résultat
    Fin
```

Version récursive :

```
fonction factorielle (n : in Entier) Retourne Entier Est
    Début
        Si n <= 1 Alors
```

```
4               Retourne 1
5           Sinon
6               Retourne n * factorielle(n - 1)
7           FinSi
8       Fin
```

On peut alors définir la fonction factorielle à l'intérieur de la fonction combinaison. Ceci est cohérent avec les méthodes des raffinages. Cependant, plusieurs arguments ici incitent ici à la définir au même niveau :

1. factorielle est une fonction utile, il serait donc dommage de la cacher dans Combinaisons.
2. La postcondition de Combinaisons fait référence à factorielle. Il faut donc que factorielle soit connue de l'appelant.
3. Si un sous-programme est défini à l'intérieur d'un autre, les paramètres du sous-programme englobant sont potentiellement des variables globales.
4. Il est difficile de tester un sous-programme interne puisqu'on y a pas accès.

Remarque : Pour que les sous-programmes soient réellement réutilisables, il faudrait qu'ils soient définis dans des modules (des paquetages en Ada).

Avec les modules (paquetages en Ada), définir un sous-programme à l'intérieur d'un autre perd de son intérêt !

```
1  Fonction Combinaisons(m, p : in Entier) Retourne Entier
2      Début
3          Retourne factorielle(m) Div factorielle (p) Div factorielle (m - p)
4      Fin
```

3. Concevoir un programme principal qui affiche le nombre de combinaisons de deux entiers saisis au clavier.

Solution : On retrouve ici les raffinages classiques. L'étape 2 du R1 correspond au résultat des questions précédentes. Quand on appelle Combinaisons, il faut s'assurer que ses préconditions sont satisfaites.

```
1  R0 : Afficher le nombre de combinaisons de deux entiers lus au clavier.
2
3  Exemples : déjà donnés...
4
5
6  R1 : Comment « Afficher ... »
7      Demander deux entiers m et p          m, p : out Entier
8      { 0 <= p Et p <= m }
9      Calculer le nombre de combinaisons de p dans m     m, p: in ; c_m_p: out
10 Entier
11     Afficher le nombre de combinaisons    c_m_p: in
```

Le reste est immédiat et donné dans le programme ci-après.

Remarque : On a pris les mêmes noms pour les variables du programme principal et des paramètres du sous-programme. Ceci ne pose pas de problème car ils apparaissent dans des contextes différentes.

4. Est-ce que l'on aurait pu concevoir ce programme principal si les sous-programmes associés n'étaient pas encore conçus ? La réponse doit être justifiée.

Solution : Oui car la seule connaissance de la spécification d'un SP est suffisante pour pouvoir l'utiliser.

```ada
-- Auteur : Xavier Crégut <nom@n7.fr>

with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
with Ada.Exceptions;

procedure Combinaisons is

    function Factorielle (N: in Integer) return Integer with
        Pre  => N >= 0,
        Post => (N <= 1 and then Factorielle'Result = 1)
                or else (N > 1 and then  Factorielle'Result = N * Factorielle (N - 1))
    is
        begin
            if N <= 1 then
                return 1;
            else
                return N * Factorielle (N - 1);
            end if;
        end;

    function Combinaisons (M, P: in Integer) return Integer with
        Pre => P >= 0 and then M >= P
    is
        begin
            return Factorielle (M) / Factorielle (P) / Factorielle (M - P);
        end;

    procedure Tester_Factorielle is
    begin
        pragma Assert (1 = Factorielle (0));
        pragma Assert (1 = Factorielle (1));
        pragma Assert (6 = Factorielle (3));
        pragma Assert (24 = factorielle (4));
        pragma Assert (120 = Factorielle (5));
        pragma Assert (120 = Factorielle (5));
        pragma Assert (479001600 = Factorielle (12));
        -- pragma Assert (6227020800 = Factorielle (13));
            -- Ce dernier test est en commentaire car la valeur attendue
            -- dépasse la capacité des entiers d'Ada.
    end Tester_Factorielle;

    procedure Tester_Combinaisons is
    begin
        pragma Assert (1 = Combinaisons (2, 2));
        pragma Assert (2 = Combinaisons (2, 1));
        pragma Assert (4 = Combinaisons (4, 3));
        pragma Assert (6 = Combinaisons (4, 2));
        pragma Assert (495 = Combinaisons (12, 4));
    end Tester_Combinaisons;
```

```
51
52      M, P  : Integer;
53      C_M_P : Integer;    -- combinaisons de P parmi M.
54  begin
55      Tester_Factorielle;
56      Tester_Combinaisons;
57
58      -- Demander deux entiers m et p (pas de contrôle ici !)
59      Put ("M = ");
60      Get (M);
61      Skip_Line;
62      Put ("P = ");
63      Get (P);
64      Skip_Line;
65
66      pragma Assert (P >= 0);
67      pragma Assert (M >= P);
68
69      -- Calculer le nombre de combinaisons de P parmi M
70      C_M_P := Combinaisons (M, P);
71
72      -- Afficher le nombre de combinaisons
73      Put ("Nombre de combinaisons : ");
74      Put (C_M_P, 1);
75      New_Line;
76
77  end Combinaisons;
```

Si le langage considéré utilise des entiers bornés, l'implantation de la fonction `combinaisons` ci-dessus ne sera pas très utile car elle conduira très vite à un dépassement de la capacité des entiers à cause de l'utilisation de la fonction factorielle.

On peut alors s'appuyer sur une autre définition (ci-après) qui nous permettra de traiter des valeurs de m et p plus grande.

$$\begin{cases} C_0^0 = 1 \\ C_m^0 = 1 \\ C_m^m = 1 \\ C_{m+1}^{p+1} = C_m^{p+1} + C_m^p \end{cases}$$

En s'appuyant sur cette nouvelle définition, on peut écrire une version récursive mais qui sera très peu efficace (complexité exponentielle) ou une version itérative qui consiste à utiliser un tableau pour conserver les combinaisons précédentes. On construit ainsi à chaque itération une nouvelle ligne du triangle de Pascal.

Une version efficace pour calculer C_m^p est de s'appuyer sur la relation de récurrence suivante : $C_m^p = \frac{m}{p} C_{m-1}^{p-1}$ avec $C_m^0 = 1$. Le calcul se fait alors $O(p)$.

Exercice 3 : Saisir un entier

Pour saisir un entier naturel au clavier, l'utilisateur entre plusieurs chiffres sous forme de caractères qui sont convertis en un entier. Si l'utilisateur entre un caractère autre qu'un chiffre, alors la saisie de caractères s'arrête, et l'entier courant est produit. Ainsi, « 12X » donnera l'entier 12.

1. Définir la spécification du sous-programme associé au problème ci-dessus

Solution :

```
-- Objectif : Saisir un entier naturel au clavier
-- Paramètres :
--      Nombre: out Entier
-- Nécessite : Vrai
-- Assure : -- L'entier est affiché sur la sortie standard (le terminal).
-- Exemples :
--      421 -> 421
--      3x -> 3
--      -5 -> 0
Procédure Saisir(Nombre : out Entier)
```

Remarque : On retrouve la signature du Get d'un entier en Ada. Ceci n'est pas surprenant puisqu'on cherche à réaliser le même objectif.

2. En utilisant la méthodes des raffinages, concevoir le corps de ce sous-programme.

Solution : Le principe est d'utiliser le schéma de Horner. Si l'utilisateur saisit 231, il a en fait tapé les caractères `'2'`, `'3'`, `'1'` et retour à la ligne. Le principe du schéma de Horner est de multiplié par 10 le nombre déjà reconnu et de lui ajouter le chiffre qui correspond au nouveau caractère.

On part de 0 (pas encore de caractère lu).

Le premier caractère est `'2'`, c'est bien un chiffre. On fait donc 0 * 10 + 2 (2 est le chiffre qui correspond au caractère `'2'`) = 2.

Le caractère suivant est `'3'`. On fait donc 2 * 10 + 3 = 23.

Le caractère suivant est `'1'`. On fait donc 23 * 10 + 1 = 231.

Le caractère suivant est le retour à la ligne. Ce n'est pas un chiffre. On s'arrête et on a donc obtenu le chiffre 231.

```
R1 : Comment « Saisir (Nombre : out Entier) »
    Nombre_Courant <- 0
    Lire(Caractère)
    Tantque '0' <= Caractère Et Caractère <= '9' Faire -- Caractère est un chiffre
        chiffre <- Ord(Caractère) - Ord('0')
        Nombre_Courant <- Nombre_Courant * 10 + chiffre
        Lire(Caractère)
    FinTQ
    -- Attention ! On a lu un caractère en avant (donc en trop) !
    Nombre <- Nombre_Courant
```

Nous utilisons une variable locale `Nombre_Courant` pour deux raisons. La première est que `Nombre` est en **`out`** et ne peut théoriquement être qu'affecté. On ne peut donc théoriquement pas écrire `Nombre <- Nombre * 10 + Chiffre`. La deuxième raison est que l'on ne devrait modifier un paramètre en sortie que pour lui affecter la bonne valeur. En effet, si lors du traitement une erreur se produisait, il ne faudrait pas avoir modifié `Nombre` qui se trouverait dans un état indéterminé. Ceci serait particulièrement vrai pour un paramètre en **`in out`**.

`C_Lu` est une variable locale qui est nécessaire pour pouvoir lire un nouveau caractère.

`Chiffre` n'est pas obligatoire mais permet d'expliquer le code en donnant un nombre à l'expression qui permet d'obtenir le chiffre correspondant à un caractère.

Notons que dans cet algorithme, nous avons lu un caractère en avant (celui qui permet de décider que la lecture du nombre est terminée) et nous n'en avons rien fait. Il est donc perdu pour le reste du programme.

```
1  -- Auteur : Xavier Crégut <nom@n7.fr>
2
3  with Ada.Text_Io;    use Ada.Text_Io;
4
5  procedure Saisir_Entier is
6
7      -- Équivalent de Get autorisant la lecture d'un retour à la ligne.
8      -- Le Get par défaut ignore les retours à la ligne et positionne
9      -- End_Of_Line si le caractère suivant est un retour à la ligne.
10     -- Attention : le caractère est immédiatement disponible sans attendre le
11     -- retour à la ligne (pas de bufferisation des entrées).
12     procedure My_Get (C: out Character) is
13         C_Lu: Character;
14     begin
15         Get_Immediate(C_Lu);    -- le caractère lu n'est pas affiché
16         Put(C_Lu);              -- on l'affiche donc explicitement
17         C := C_Lu;
18     end My_Get;
19
20
21     -- Saisir un entier dans une base donnée.
22     -- Paramètres
23     --     Nombre : l'entier saisi
24     --     Base   : la base à utiliser
25     --  Attention : le débordement des entiers n'est pas traité
26     procedure Saisir (Nombre : out Integer) is
27
28         C_Lu: Character;
29         Chiffre: Integer;    -- le chiffre correspondant à C_Lu
30         Nombre_Courant: Integer; -- le nombre en cours de lecture
31     begin
32         Nombre_Courant := 0;
33         My_Get(C_Lu);
34         -- Attention : le retour à la ligne a un traitement spécifique
35         -- et n'est pas lu.
36         while '0' <= C_Lu and C_Lu <= '9' loop
37             Chiffre := Character'Pos(C_Lu) - Character'Pos('0');
38             Nombre_Courant := Nombre_Courant * 10 + Chiffre;
39             My_Get(C_Lu);
40         end loop;
41         -- Attention : un caractère lu en avant !
42         nombre := Nombre_Courant;
43     end;
44
```

```
45      N: Integer;
46  begin
47      -- Saisir un entier
48      Put ("Entier ? ");
49      Saisir(N);
50
51      -- Afficher l'entier saisi
52      Put("Entier saisi :" & Integer'Image(N));
53  end Saisir_Entier;
```

3. On veut pouvoir saisir l'entier en base 2, 3, 4, etc. Adapter le sous-programme précédent.

Solution : La procédure précédente fonctionne pour une base fixée à 10. Ici on veut pouvoir travailler avec différentes bases. L'idée est donc de faire de la base un paramètre de notre sous-programme. Il ne sera donc plus fixé dans le sous-programme mais choisi par l'appelant du sous-programme.

Notons qu'il ne suffit pas de remplacer 10 par Base. Il faut aussi remplacer `'9'`. Ceci montre qu'il est souvent préférable de s'appuyer sur une constante nommée plutôt que sur des constantes littérales car il n'est pas forcément évident de voir que `'9'` est lié à 10.

Nous nous limitons aux bases de 2 à 10 car au délà il faudrait utiliser d'autres symboles que les chiffres. Par exemple en base 16, on utilise aussi les lettres de `'A'` à `'F'`. Ceci compliquerait donc la condition du TantQue et le passage du caractère au nombre correspond mais n'apporte rien d'un point de vu algorithmique.

Remarque : On peut utiliser les valeurs par défaut des paramètres pour fixer par défaut à 10 la valeur de la base et retrouver la comportement de la procédure définie dans l'exercice précédent.

```
1   -- Auteur : Xavier Crégut <nom@n7.fr>
2
3   with Ada.Text_Io;    use Ada.Text_Io;
4
5   procedure Saisir_Entier_Base is
6
7       -- Équivalent de Get autorisant la lecture d'un retour à la ligne.
8       -- Le Get par défaut ignore les retours à la ligne et positionne
9       -- End_Of_Line si le caractère suivant est un retour à la ligne.
10      -- Attention : le caractère est immédiatement disponible sans attendre le
11      -- retour à la ligne (pas de bufferisation des entrées).
12      procedure My_Get (C: out Character) is
13          C_Lu: Character;
14      begin
15          Get_Immediate(C_Lu);    -- le caractère lu n'est pas affiché
16          Put(C_Lu);              -- on l'affiche donc explicitement
17          C := C_Lu;
18      end My_Get;
19
20
21      -- Saisir un entier dans une base donnée.
22      -- Paramètres
23      --     Nombre : l'entier saisi
```

```
24      --     Base   : la base à utiliser
25      --  Nécessite
26      --     2 <= Base and Base <= 10
27      --  Attention : le débordement des entiers n'est pas traité
28      procedure Saisir (Nombre : out Integer; Base: in Integer := 10)
29          with pre => Base >= 2 and Base <= 10
30      is
31
32
33          C_Lu: Character;
34          Chiffre: Integer;    -- le chiffre correspondant à C_Lu
35          Nombre_Courant: Integer; -- le nombre en cours de lecture
36          Limite: Character;    -- dernier chiffre autorisé
37      begin
38          Limite := Character'Val(Character'Pos('0') + Base - 1);
39          Nombre_Courant := 0;
40          My_Get(C_Lu);
41          -- Attention : le retour à la ligne a un traitement spécifique
42          -- et n'est pas lu.
43          while '0' <= C_Lu and C_Lu <= Limite loop
44              Chiffre := Character'Pos(C_Lu) - Character'Pos('0');
45              Nombre_Courant := Nombre_Courant * Base + Chiffre;
46              My_Get(C_Lu);
47          end loop;
48          -- Attention : un caractère lu en avant !
49          nombre := Nombre_Courant;
50      end;
51
52      N: Integer;
53      La_Base: Integer;
54  begin
55      -- Saisir la base (pas de contrôle)
56      Put ("Base ? ");
57      Saisir (La_Base);
58
59      -- Saisir un entier
60      Put ("Entier ? ");
61      Saisir(N, La_Base);
62
63      -- Afficher l'entier saisi
64      Put("Entier saisi (en base 10) :" & Integer'Image(N));
65  end Saisir_Entier_Base;
```

Exercice 4 : Afficher un entier

On s'intéresse à la réalisation d'un sous-programme permettant d'afficher un entier sous forme d'une suite de chiffres représentés par des caractères.

1. Définir la spécification du sous-programme associé au problème ci-dessus

Solution :

```
-- Objectif : Afficher un entier naturel.
-- Contrainte : On n'utilisera que l'écriture d'un caractère.
-- Paramètres :
--      - Nombre : in Entier -- l'entier à afficher
-- Nécessite : Nombre >= 0      -- On ne traitera que ce cas.
-- Assure : ---
-- Exemple
--      421 -> affiche "421"    -- nominal
--        0 -> affiche "0"      -- limite
--       10 -> affiche "10"     -- une puissance de 10
--     Faire aussi un test avec : Integer'Last
Procédure Afficher(Nombre : in Entier)
```

2. En utilisant la méthodes des raffinages, concevoir le corps de ce sous-programme.

Solution : On peut envisager plusieurs solutions :

1. On remarque que pour afficher N >= 10, on peut d'abord afficher N Div 10 puis N Mod 10. On peut alors s'appuyer sur la récursivité.

   La taille du problème est le nombre de chiffre du nombre à afficher.

   Le cas d'arrêt : le nombre de chiffre est égal à 1.

   ```
   R1 : Comment « Afficher(Nombre: in Entier) »
       Si N < 10 Alors      -- 0 <= N < 10
           Afficher le chiffre N
       Sinon
           Afficher (N Div 10)
           Afficher le chiffre N Mod 10
       FinSi
   ```

   On remarque que l'on peut factoriser « Afficher le chiffre en affichant N Mod 10 ». On obtient alors :

   ```
   R1 : Comment « Afficher(Nombre: in Entier) »
       Si N >= 10 Alors
           Afficher (N Div 10)
       Sinon
           Rien
       FinSi
       Afficher le chiffre N Mod 10

   R2 : Comment « Afficher le chiffre C »
       Ecrire( Chr(Ord('0') + C ) )
   ```

2. Si on ne veut pas utiliser la récursivité. On constate que N mod 10 permet de récupérer les unités et N Div 10 est ce qu'il reste à afficher. Sur ce principe, on peut récupérer successivement tous les chiffres. Malheureusement, on les obtient dans l'ordre inverse de l'affichage souhaité. Ainsi, 421 donnera 1, 2 puis 4.

Si on avait les tableaux ou une autre structure de donnée à disposition (une pile serait bien adaptée !), on pourrait stocker les chiffres avant de les afficher (voir `Afficher_Tab`, listing 1).

Si on n'a pas de possibilité de stocker les chiffres, il faut donc commencer par récupérer le chiffre le plus à gauche. L'idée est alors d'utiliser une puissance de 10 et faire des Div et des Mod avec cette puissance. Il faut commencer la trouver la première puissance de 10 à considérer, celle qui est juste inférieure au nombre.

Dans ces calculs, il faut aussi éviter de déborder la capacité des entiers ! En Ada, ceci est signalé par une `CONSTRAINT_ERROR` avec un message *overflow check failed*.

```
R1 : Comment « Afficher (N: in Entier) »
    Trouver la plus grande puissance du 10 inférieure ou égale à N      N: in; puissance: o
    { puissance <= N}
    { puissance > N Div 10 }   -- dans ce sens pour éviter les débordements
    (Extraire et) afficher les chiffres de N    puissance: in (out), N : in

R2 : Comment « Trouver la plus grande puissance de 10 ...
    puissance <- 1
    TantQue puissance <= Nombre / 10 Faire
        puissance <- puissance * 10
    FinTQ

R2 : Comment « (Extraire et) Afficher les chiffres de N
    Répéter
        Chiffre <- (N Div Puissance) Mod 10
        Écrire Chiffre
        Puissance <- Puissance Div 10
    JusquÀ Puissance = 0
```

Remarque dans le code Ada, j'ai ajouté un compteur `Nombre_Chiffres` dans « Trouver la plus grande puissance... » et j'ai fait une boucle Pour pour « (Extraire et) afficher les chiffres ».

Listing 1 – Plusieurs version de « afficher un entier naturel » en Ada

```
-- Auteur : Xavier Crégut <nom@n7.fr>

with Ada.Text_Io;   use Ada.Text_Io;

procedure Afficher_Entier is

    -- Afficher l'entier Nombre en base 10 sur la sortie standard
    -- Nécessite : Nombre >= 0;
    -- Assure : ---
    procedure Afficher (Nombre : in Integer)
        with Pre => Nombre >= 0
    is
    begin
        -- Principe : afficher N, c'est afficher N / 10 et écrire N Mod 10 (si
        -- N >= 10)

        if Nombre >= 10 then
```

```
18              Afficher (Nombre / 10);
19          else
20              null;
21          end if;
22          Put (Character'Val (Character'Pos ('0') + Nombre Mod 10));
23      end;
24
25
26      procedure Afficher_Iteratif (Nombre : in Integer)
27          with Pre => Nombre >= 0
28      is
29          Puissance: Integer;            -- puissances de 10
30          Nombre_Chiffres:  Integer;     -- Nombre de chiffres de Nombre
31          Chiffre: Integer;             -- un chiffre de Nombre
32      begin
33          -- Principe : Utiliser Div et Mod avec des puissances de 10
34          -- décroissantes pour extraires successivemetn chacun des chiffres.
35
36          -- Trouver la plus grande puissance de 10 strictement ou égale à Nombre
37          Puissance := 1;
38          Nombre_Chiffres := 1;
39          while Puissance <= Nombre / 10 loop
40              Puissance := Puissance * 10;
41              Nombre_Chiffres := Nombre_Chiffres + 1;
42          end loop;
43
44          -- Afficher les chiffres de Nombre
45          for I in 1..Nombre_Chiffres loop
46              Chiffre := Nombre / Puissance Mod 10;
47              Put (Character'Val (Character'Pos ('0') + Chiffre));
48              Puissance := Puissance / 10;
49          end loop;
50      end;
51
52
53      procedure Afficher_Tab (Nombre : in Integer)
54          with Pre => Nombre >= 0
55      is
56          Chiffres: array (1..Integer'WIDTH) of Character;    -- les chiffres de Nombre
57          Nombre_Bis: Integer;         -- Nombre modifié pour en extraire les chiffres
58          Nombre_Chiffres:  Integer;     -- Nombre de chiffres de Nombre
59          Chiffre: Integer;             -- Un chiffre de Nombre
60      begin
61          -- Principe : Faire des Mod 10 et Div 10 pour extraire les chiffres de
62          -- Nombre, les ranger dans un tableau pour les afficher dans l'ordre
63          -- inverse de leur obtention.
64
65          -- Extraire les chiffres de Nombre_Bis
66          Nombre_Bis := Nombre;     -- Nombre est en in
67          Nombre_Chiffres := 0;
```

```
68          loop
69              Chiffre := Nombre_Bis Mod 10;
70              Nombre_Chiffres := Nombre_Chiffres + 1;
71              Chiffres (Nombre_Chiffres) := Character'Val (Character'Pos ('0') + Chiffre);
72              Nombre_Bis := Nombre_Bis / 10;
73              exit when Nombre_Bis = 0;
74          end loop;
75
76          -- Afficher les chiffres
77          for I in reverse 1..Nombre_Chiffres loop
78              Put (Chiffres (I));
79          end loop;
80      end;
81
82
83
84  begin
85      Put ("Version récursive :");
86      New_Line; Afficher (421);
87      New_Line; Afficher (0);
88      New_Line; Afficher (10);
89      New_Line; Afficher (9);
90      New_Line; Afficher (Integer'LAST);
91      New_Line;
92
93      New_Line; Put ("Version itérative sans tableau :");
94      New_Line; Afficher_Iteratif (421);
95      New_Line; Afficher_Iteratif (0);
96      New_Line; Afficher_Iteratif (10);
97      New_Line; Afficher_Iteratif (9);
98      New_Line; Afficher_Iteratif (Integer'LAST);
99      New_Line;
100
101     New_Line; Put ("Version itérative avec tableau :");
102     New_Line; Afficher_Tab (421);
103     New_Line; Afficher_Tab (0);
104     New_Line; Afficher_Tab (10);
105     New_Line; Afficher_Tab (9);
106     New_Line; Afficher_Tab (Integer'LAST);
107     New_Line;
108
109 end Afficher_Entier;
```