Constructeurs de types
Corrigé

Objectifs

- Comprendre et savoir définir les types utilisateurs
- Manipuler les tableaux
- Écrire une méthode de tri simple... mais pas très efficace.
- Raffiner... Toujours !

Exercice 1 : Modéliser un robot de type 1

L'objectif de cet exercice est de modéliser un robot de type 1. Un tel robot se déplace dans un environnement qui peut être modélisé par un quadrillage dont chaque case correspond à une position possible du robot.

Un robot est donc caractérisé par sa position (son abscisse, x, et son ordonnée, y) et sa direction (nord, sud, est ou ouest). La figure 1 décrit un robot à la position $x = 4$ et $y = 2$, sa direction est « ouest ».

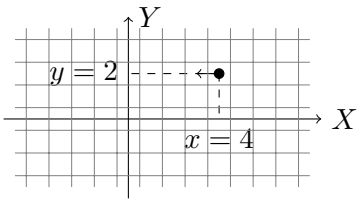

FIGURE 1 – Le robot dans un environnement illimité

Les robots de type 1 ont des possibilités réduites qui se limitent à pivoter de 90^o vers la droite et à avancer d'une case suivant sa direction courante.

Pour simplifier, on considère que le robot évolue dans un environnement infini sans obstacle.

1. Définir les types nécessaires pour modéliser un robot de type 1.

Solution : Un robot de type 1 est caractérisé par :

- une position, c'est-à-dire une abscisse (x) et une ordonnée (y). On peut donc définir un type Position comme un enregistrement.

— une direction. La direction peut prendre 4 valeurs que l'on peut par exemple appelées Nord, Sud, Est et Ouest. On aurait également pu les appeler Gauche, Droite, Haut et Bas. En conséquence, il s'agit d'un type énuméré.

Puisque les valeurs d'un type énuméré sont ordonnées, on peut se demander si un ordre est plus judicieux qu'un autre. Ici on sait que le robot tourne à droite de 90°. Il serait donc naturel de lister les valeurs pour que la suivant corresponde à 90° à droite, par exemple (Nord, Est, Sud, Ouest). On pourra en tirer partie quand on voudra écrire le sous-programme « avancer » du robot.

Le type Robot1 est donc un enregistrement qui regroupe ces deux informations.

```
Type
    T_Position =
        Enregistrement
            x: Entier   -- abscisse
            y: Entier   -- ordonnée
        FinEnregistrement

    T_Direction = (NORD, EST, SUD, OUEST)

    T_Robot1 =
        Enregistrement
            position: T_Position
            direction: T_Direction
        FinEnregistrement
```

Remarques :

1. Nous avons préfixé les noms des types par « `T_` » pour éviter les conflits entre le nom du type et le nom d'une variable ou d'un paramètre. Même si la position (avant ou après un « : ») permettrait de savoir s'il s'agit d'une variable ou d'un type, de nombreux langages interdisent donc d'avoir un type et une variable (ou un paramètre) qui portent le même nom.
2. Les noms « x » et « y » sont trop courts mais ne sont des conventions communément admises. C'est pour cette raison que nous les utilisons.

   Il est préférable d'écrire « x » et « y » plutôt que « abs » et « ord ». Les abréviations sont à proscrire sauf si communément admises. Que signifie « abs » ? Abscisse ? Valeur absolue ?
3. On aurait pu mettre directement les trois champs x, y et direction dans Robot1.

2. L'environnement dans lequel évolue le robot est en fait fini et comporte des obstacles. On suppose qu'un obstacle occupe une case du quadrillage. Il s'agit donc de savoir si la case est libre ou contient un obstacle. Définir un type pour modéliser l'environnement.

Solution : L'environnement dans lequel évolue le robot est un quadrillage, donc il peut être modélisé par un tableau à deux dimensions. Nous utilisons deux constantes pour en préciser les dimensions (`MIN_X..MAX_X` et `MIN_Y..MAX_Y`).

Chaque case du tableau est soit une position qui peut être occupée par le robot, soit un mur, donc deux valeurs possibles. On peut donc représenter cette information par un type booléen (il faut choisir ce que signifie **`VRAI`** et **`FAUX`**) ou définir un type énuméré.

Nous choisissons ici cette deuxième solution qui est plus explicite :

```
Constante
    Min_X = -10
    MAX_X = 20
    MIN_Y = -20
    MAX_Y = 30

Type
    T_Case = (LIBRE, MUR)
        -- indique si la case est LIBRE et peut donc être occupé par le robot
        -- ou si elle contient un mur

    T_Environnement = Tableau [MIN_X..MAX_X, MIN_Y..MAX_Y] De T_Case
```

Notons que dans ce cas, on utilise toutes les cases du tableau. Il n'y a donc pas de notion de taille effective.

Une autre solution serait de conserver les coordonnées des cases qui contiennent un obstacle. Dans ce cas, nous aurions besoin de définir une capacité (on pourrait la définir à partir de nos constantes) et une taille effective car toutes les cases ne seront pas nécessairement utilisées. Le type des éléments du tableau serait une position.

```
Constante
    MAX_OBSTACLES = 20    -- le nombre maximal d'obstacles dans l'environnement

Type
    T_Obstables = Tableau [1..MAX_OBSTACLES] De T_Position

    T_Environnement =
        Enregistrement
            Nombre: Entier         -- le nombre d'obstacles dans l'environnement
            Obstacles: T_Obstacles -- la position des obstacles
        FinEnregistrement
```

Le premier type sera plus efficace si on veut savoir si une case est libre connaissant ses coordonnées car on a un accès direct à la case. Le deuxième serait plus efficace en mémoire si le nombre d'obstacles n'est pas trop grand.

3. Écrire un sous-programme qui, étant donnés un robot et son environnement, fait avancer ce robot toujours tout droit jusqu'à ce qu'il rencontre un obstacle ou qu'il arrive à la limite de son environnement. On supposera le robot initialement sur une position valide en direction de l'est.

Solution : On considère un robot particulier dans un environnement donné. Dans les programmes de test, il s'agira donc d'initialiser ces deux informations de manière à ce que l'on puisse tester notre algorithme dans des cas significatifs. Notons que la direction est fixé à l'EST.

Les cas en prendre en compte sont :

— pas de mur : la limite est atteinte

— un mur : s'arrête devant le mur

— le robot est juste à l'ouest d'un mur : le robot ne bouge pas

La spécification du sous-programme est la suivante.

```
-- Faire avancer un robot au maximum. Le robot s'arrête quand il atteint les
-- limites de l'environnement ou qu'il est bloqué par un obstacle.
--
-- Paramètres
--     robot : le robot à faire avancer
```

```
--     environnement : l'environnement dans lequel le robot évolue
--
-- Nécessaire
--     -- robot dans l'environnement
--     MIN_X <= robot.position.x Et robot.position.x <=  MAX_X
--     MIN_Y <= robot.position.y Et robot.position.y <=  MAX_Y
--
--     -- le robot est sur une case libre d l'environnement
--     environnement[robot.position.x, robot.position.y] = LIBRE
--
--     -- le robot est en direction Est
--     robot.direction = EST
--
-- Assure
--     -- le robot est toujours dans l'environnement
--     MIN_X <= robot.position.x Et robot.position.x <=  MAX_X
--     MIN_Y <= robot.position.y Et robot.position.y <=  MAX_Y
--
--     -- il a avancé dans la direction (Est)
--     robot.direction = \old(robot.direction)
--     robot.position.y = \old(robot.position.y)
--     robot.position.y >= \old(robot.position.x)
--
--     -- il est à la limite ou devant un obstacle
--     robot.position.x = MAX_X Ou Alors
--             environnement[robot.position.x + 1, robot.position.y] = MUR
--
--     -- les cases entre la position initiale et la position finale sont libres
--     PourChaque x Dans \old(robot.position.x)..robot.position.x :
--          environnement[x, robot.position.y] = LIBRE
Procédure avancer_max (robot : in out T_Robot1 ; environnement : in T_Environnement)
```

```
R1 : Raffinage De  « Faire avancer le robot au maximum »
    TantQue (limite environnement non atteinte) Et Alors (Pas de mur) Faire
        Faire avancer le robot d'une case (vers l'est)
    FinTQ

R2 : Raffinage De « limite environnement non atteinte »
    Résultat <- robot.position.x < MAX_X

R2 : Raffinage De « Pas de mur »
    Résultat <- environnement[robot.position.x + 1, robot.position.y] <> MUR

R2 : Raffinage De « Faire avancer le robot d'une case »
    robot.position.x <- robot.position.x + 1
```

Exercice 2 : Afficher un tableau

Écrire un sous-programme qui affiche à l'écran un tableau d'entiers. Le tableau sera délimité par des crochets et les éléments seront séparés par une virgule. Voici quelques exemples d'affichage :

```
[]
[1, 2, 3]
[4]
[10, 2, 4, 7]
```

Solution : On constate que l'on veut afficher des tableaux avec des nombres d'éléments différents. Nous définissons donc un type Tableau avec la possibilité de stocker un certain nombre d'élément et en précisant sa taille (le nombre d'éléments effectivement enregistrés).

```
Constante
    MAX = 10

Type
    Tableau =
        Enregistrement
            éléments: Tableau [1..MAX] De Entier
            taille: Entier
            -- invariant : 0 <= taille Et taille <= MAX
        FinEnregistrement
```

En Ada, on ne peut mettre qu'un type nommé pour type d'un champ d'un enregistrement. Il faudra donc nommer le type `Tableau [1..MAX] De Entier`.

On constate que l'on ne peut pas afficher directement chaque élément du tableau car la virgule ne doit pas être mise après le dernier élément. le principe est donc d'afficher d'abord le premier élément puis, pour tous les éléments suivants, d'afficher la virgule séparateur et l'élément lui-même. Une alternative serait de particulariser le dernier élément plutôt que le premier. Ceci ne peut bien sûr être fait que si le tableau contient au moins un élément.

On pourrait aussi ajouter une conditionnelle dans la boucle pour afficher ou non la virgule mais ceci consisterait à évaluer la condition pour chaque élément du tableau alors qu'elle ne sera fausse qu'une seule fois, lors de la première itération ou de la dernière (suivant comment les instructions son organisées).

```
-- Afficher un tableau.
--
-- Paramètre
--     tab: le tableau à afficher
Procédure afficher(tab: in Tableau)
Début
    Écrire('[')
    Si tab.taille > 0 Alors
        Écrire(tab.éléments[1])
        Pour I in 2..tab.taille Faire
            Écrire(', ')
            Écrire(tab.éléments[i])
        FinPour
    Sinon
        Rien
    FinSi
    Écrire(']')
Fin
```

Exercice 3 : Tri par insertion séquentielle

L'objectif est de trier le vecteur$A$ de N entiers relatifs quelconques. Le vecteur A est trié si $A[i] \leq A[i+1]$.

Le tri utilisé est le tri par insertion séquentielle. C'est un tri en $(N-1)$ étapes. L'étape i consiste à placer le $(i+1)^{\text{e}}$ élément du vecteur à sa place dans le sous-vecteur $i+1$ premiers éléments sachant que sous-vecteur des i premiers éléments est trié par les étapes précédentes. Dans le cas du tri par insertion séquentielle, la recherche de la position d'insertion, se fait séquentiellement en comparant successivement l'élément à insérer aux i premiers éléments du vecteur.

Exemple : Voici les différentes valeurs du vecteur 8 2 9 5 1 7 après chaque étape (la partie encadrée correspond à la partie du vecteur déjà traitée et donc triée) :

vecteur initial	:	[8] 2 9 5 1 7
après l'étape 1	:	[2 8] 9 5 1 7
après l'étape 2	:	[2 8 9] 5 1 7
après l'étape 3	:	[2 5 8 9] 1 7
après l'étape 4	:	[1 2 5 8 9] 7
après l'étape 5	:	[1 2 5 7 8 9]

1. Écrire un sous-programme pour trier dans l'ordre croissant les éléments d'un vecteur. On utilisera le principe du tri par insertion séquentielle.

Solution :

```
R0 : Trier un tableau A de taille effective N (les indices sont 1..N).
```

Le problème posé consiste à trier une liste. On peut en déduire un sous-programme qui prend en paramètre cette liste et dont la spécification pourrait être la suivante.

```
-- Trier un tableau dans l'ordre croissant de ses éléments.
--
-- Paramètres
--     tableau : le tableau à trier (notation énoncé)
Procédure trier(tableau: in out Tableau)
```

Remarque : On pourrait avoir d'autres paramètres pour préciser la partie du tableau à trier, par exemple l'indice du premier élément à trier et du dernier élément.

Solution informelle : On note qu'après chaque étape i du tri, les (i+1) premiers éléments sont à leur place et sont donc triés. Ainsi, au bout de la (N-1)^e étape l'ensemble du tableau est trié.

L'étape i consiste à trier le tableau des (i+1) premiers éléments sachant le sous-tableau des i premiers éléments est déjà trié.

```
R1 : Raffinage De « Trier un tableau A de taille effective N »
  | Pour indice := 2 JusquÀ indice = N Faire
  |    Insérer A[indice] dans A(1..indice) sachant que A(1..indice-1) est trié
  | FinPour

R2 : Raffinage De « Insérer A[indice] dans A(1..indice) sachant que A(1..indice-1) est trié »
  | Déterminer la position théorique de A[indice] dans A(1..indice)
  | Décaler les éléments compris entre la position théorique et indice-1
  | Ranger l'élément A[indice]
```

Si on essaie d'exécuter mentalement ce raffinage [1], on constate que l'on a perdu la valeur A[indice] lorsque l'on veut la ranger. Elle a été écrasée lors du décalage. Il est donc nécessaire de la conserver avant et donc de revoir notre raffinage.

```
R2 : Raffinage De « Insérer A[indice] dans A(1..indice) sachant que A(1..indice-1) est trié »
  | Mémoriser la valeur de A[indice]                    memoire: out Entier
  | Déterminer la position pos de A[indice] dans A(1..indice)
  | Décaler les éléments compris entre pos et indice-1
  | Ranger l'élément mémorisé
```

1. Ce qui doit toujours être fait, puisque ceci fait partie des choses à faire pour vérifier qu'un raffinage est correct.

```
6
7  R3 : Raffinage De « Déterminer la position théorique de A[indice] dans A(1..indice) »
8    | pos <- 1
9    | TantQue (pos < indice ) Et (memoire >= A[pos]) Faire
10   |     pos <- pos + 1
11   | FinTQ
12
13 R3 : Raffinage De « Décaler les éléments compris entre la position théorique et indice-1 »
14   | Pour i <- indice - 1 Décrémenter JusquÀ i = pos Faire
15   |   A[i+1] <- A[i]
16   | FinPour
```

On peut en déduire l'algorithme.

```
1  Constante
2      CAPACITÉ = 100
3  Type
4      Vecteur = Tableau [1..CAPACITÉ] De Entier
5
6  Variable
7      A: Vecteur  -- le tableau à trier
8      N: Entier   -- le nombre d'éléments de A
9      indice: Entier      -- l'indice de l'élément à insérer
10     mémoire: Entier     -- mémoriser A[indice]
11     pos: Entier -- position de l'élément mémoire dans A(1..indice)
12     i: Integer; -- parcourir les éléments du tableau
13
14 Début
15     Pour indice := 2 JusquÀ indice = N Faire
16         -- Insérer A[indice] dans A(1..indice) sachant que A(1..indice-1) est trié
17         --   Conserver la valeur de A[indice]
18         mémoire <- A[indice]
19
20         --   Déterminer la position pos de A[indice] dans A(1..indice)
21         pos <- 1
22         TantQue (pos < indice ) Et (memoire >= A[pos]) Faire
23             pos <- pos + 1
24         FinTQ
25
26         --   Décaler les éléments compris entre pos  et indice-1
27         Pour i <- indice - 1 Décrémenter JusquÀ i = pos Faire
28             A[i+1] <- A[i]
29         FinPour
30
31         --   Ranger l'élément mémorisé
32         A[pos] <- mémoire
33     FinPour
34 Fin
```

On peut en écrire une version légèrement optimisée en réalisant les décalages en même temps que l'on cherche la position d'insertion. Mais est-ce vraiment une optimisation intéressante ?

```
1  Variable
2      A: Vecteur  -- le tableau à trier
3      N: Entier   -- le nombre d'éléments de A
4      indice: Entier      -- l'indice de l'élément à insérer
5      mémoire: Entier     -- mémoriser A[indice]
6      pos: Entier -- pour chercher la position d'insertion dans A(1..indice)
7
```

```
Début
    Pour indice := 2 JusquÀ indice = N Faire
        -- Insérer A[indice] dans A(1..indice) sachant que A(1..indice-1) est trié
        --   Conserver la valeur de A[indice]
        mémoire <- A[indice]

        --   Décaler les éléments pour faire de la place pour A[indice] dans A(1..indice)
        pos <- indice
        TantQue (pos > 1 ) Et (memoire < A[pos-1]) Faire
            A[pos] <- A[pos-1]
            pos <- pos - 1
        FinTQ

        --   Ranger l'élément mémorisé
        A[pos] <- mémoire
    FinPour
Fin
```

2. En conservant le principe du tri par insertion, expliquer comment améliorer l'efficacité de cet algorithme.
Solution : Si on respecte le principe du tri par insertion, il faudra toujours réaliser les décalages. On ne peut donc gagner en performance qu'en optimisant la recherche (l'optimisation présentée précédemment n'est pas une optimisation réellement intéressante !). Comme on sait que le vecteur des indice-1 premiers éléments est trié, on peut utiliser une recherche par dichotomie pour rechercher la position théorique de l'élément dans le vecteur.

Exercice 4 : Saisir un tableau

Écrire un programme qui initialise un tableau d'entiers en lisant des valeurs au clavier. On considèrera trois modes de lecture :

1. lecture de la taille effective du tableau, puis des valeurs ;
2. lecture des valeurs les unes après les autres et arrêt sur une valeur particulière (une valeur négative par exemple) ;
3. lecture de couples (indice, valeur) avec arrêt lorsque l'indice donné est -1.

Quels sont les avantages et les inconvénients de ces différentes approches ?
Remarque : on veillera à ce que la capacité du tableau ne soit pas dépassée.

Solution : Dans les trois cas, on considère que le tableau a déjà été déclaré. Sa capacité est donc déjà définie et ne peut pas être modifiée. L'objectif de ces opérations est simplement d'initialiser (de remplir) le tableau.

Premier cas. On demande à l'utilisateur le nombre de données, on sait donc ensuite combien de valeurs doivent être lues. On peut donc utiliser une boucle **Pour**.

La vérification du dépassement se fait soit lors de la lecture du nombre d'éléments (on signale le dépassement et on demande une nouvelle taille) ou on limite à la capacité du tableau.

```
    -- Initialiser un tableau Tab à partir d'éléments lus au clavier.
    -- La nombre d'éléments est d'abord demandée, suivi des éléments.
    -- Les éléments en surnombre par rapport à la capacité du tableau
    -- sont ignorés et le message "Données tronquées" est affiché.
```

```
5       --
6       -- Paramètres
7       --     Tab : le tableau à initialiser
8       procedure Lire (Tab: out T_Tableau) is
9           Taille_Souhaitee: Integer;
10          Nb_Elements: Integer;   -- Nombre d'éléments à lire
11      begin
12          -- Demander la taille
13          Put ("Nombre d'éléments ? ");
14          Get (Taille_Souhaitee);
15
16          -- Déterminer le nombre d'éléments à saisir
17          if Taille_Souhaitee > Capacite then
18              Nb_Elements := Capacite;
19          elsif Taille_Souhaitee < 0 then
20              Nb_Elements := 0;
21          else
22              Nb_Elements := Taille_Souhaitee;
23          end if;
24
25          -- Demander les éléments du tableau
26          for N in 1..Nb_Elements loop
27              Put ("Element ");
28              Put (N, 1);
29              Put (" ? ");
30              Get (Tab.Elements (N));
31          end loop;
32          Tab.Taille := Nb_Elements;
33
34          if Nb_Elements < Taille_Souhaitee then
35              Put_Line ("Données tronquées");
36          else
37              null;
38          end if;
39      end Lire;
```

Deuxième cas. On doit lire les valeurs, et c'est quand on rencontre une valeur négative que l'on sait que la série est terminée et donc la tableau rempli. On peut alors en déduire la taille.

Dans ce cas, il faut faire attention au risque de dépassement de la capacité du tableau. Par exemple, on peut stopper la saisie si le tableau est plein et, bien sûr, avertir l'utilisateur dans ce cas.

```
1       -- Initialiser un tableau Tab à partir d'éléments lus au clavier.
2       -- Les éléments sont saisis les uns à la suite des autres.  La fin
3       -- est indiqué par un élément particulier : FIN.
4       -- Les éléments en surnombre par rapport à la capacité du tableau
5       -- sont ignorés et le Depassement vaut vrai.  Il vaut faux sinon.
6       --
7       -- Paramètres
8       --     Tab : le tableau à initialiser
9       --     Depassement : Vrai si nombre de valeur > capacité du tableau
10      procedure Lire (Tab: out T_Tableau; Depassement : out Boolean) is
11          Entier : Integer;       -- un entier lu au clavier
12          Nb_Elements: Integer;   -- Nombre d'éléments à lire
13          FIN: constant Integer := -1;    -- marqueur de fin de la série
14      begin
```

```
15          -- Salsir un premier entier
16          Get (Entier);
17
18          Tab.Taille := 0;
19          while Entier /= FIN             -- un élément du tableau
20              and Tab.Taille < Capacite   -- y a de la place dans le tableau
21          loop
22              -- stocker l'élément
23              Tab.Taille := Tab.Taille + 1;
24              Tab.Elements (Tab.Taille) := Entier;
25
26              -- Lire un nouvel entier
27              Get (Entier);
28          end loop;
29
30          -- Positionner Depassement
31          Depassement := Entier /= FIN;
32      end Lire;
```

Troisième cas. Dans ce cas, on demande successivmeent des couples (indice, valeur) jusqu'à avoir un indice négatif (ou égale à -1). Pour chaque indice lu, il faut s'assurer qu'il correspond à un indice possible du tableau.

Remarquons tout d'abord qu'il n'est pas nécessaire (en fait il ne faut pas) demander la valeur si l'indice est négatif.

Comment déterminer la taille du tableau ? On peut considérer que c'est le maximum des indices des valeurs données. On pourrait soit considérer que la taille effective est la capacité du tableau ou demander explicitement la taille effective à l'utilisateur. Dans ce dernier cas, on peut alors faire des vérifications entre les indices données et la taille.

Que se passe-t-il pour les valeurs du tableau non initialisées ? On peut considérer qu'elles sont initialisées avec une valeur par défaut.

```
1       -- Initialiser un tableau Tab à partir d'éléments lus au clavier.
2       -- L'utilisateur donne un indice, puis la valeur à cet indice.  Ceci permet
3       -- de modifier une valeur déjà renseignée.  La taille du tableau est le plus
4       -- grand indice dont la valeur est positionnée.  Les valeurs non affectés
5       -- auront la valeur par défaut.
6       --
7       -- Paramètres
8       --     Tab : le tableau à initialiser
9       --     Valeur_Defaut : la valeur à utiliser pour les cases non initialisées
10      procedure Lire (Tab: out T_Tableau; Valeur_Defaut: in Integer := 0) is
11
12          FIN: constant Integer := -1 ; -- valeur de l'indice qui termine la série
13
14          -- Demander un indice (pas de vérification).
15          procedure Demander_Indice (Indice: out Integer) is
16          begin
17              Put ("Indice ( ");
18              Put (FIN, 1);
19              Put (" pour arrêter) ? ");
20              Get (Indice);
21          end Demander_Indice;
22
23          Indice: Integer;    -- indice de la valeur à modifier
```

```
24
25      begin
26          -- Initialiser avec la valeur par défaut
27          Tab.Elements (1..Capacite) := (others => Valeur_Defaut);
28          Tab.Taille := 0;    -- le tableau est vide au départ
29
30          -- Proposer de modifier les valeurs du tableau
31          Demander_Indice (Indice);
32          while Indice /= FIN loop
33              if 1 <= Indice and Indice <= Capacite then
34                  -- Demander la valeur
35                  Put ("Valeur ? ");
36                  Get (Tab.Elements (Indice));
37
38                  -- Adapter la taille
39                  if Indice > Tab.Taille then
40                      Tab.Taille := Indice;
41                  else
42                      null;
43                  end if;
44              else
45                  -- Signaler l'erreur sur l'indice
46                  Put ("Erreur : l'indice doit être entre 1 et ");
47                  Put (Capacite, 1);
48                  Put_Line (".");
49              end if;
50
51              Demander_Indice (Indice);
52          end loop;
53      end Lire;
```