Introduction à la programmation procédurale en Ada

Ph. PREUX[1]
Université du Littoral Côte d'Opale
Calais

Novembre 1999

[1]philippe.preux@univ-lille3.fr

Résumé

Ce polycopié a pour objectif d'être un support du cours d'initiation à la programmation procédurale en Ada, basé sur le cours de DEUG MIAS 2, à Calais, au 1ersemestre 1999/2000.

En aucun cas, ce polycopié ne se substitue au cours magistral et la présence est absolument indispensable.

Le polycopié a pour unique objet d'aider à la prise de notes et permettre aux étudiants de se concentrer davantage sur ce que dit l'enseignant.

Le cours magistral est complété par des travaux dirigés et des travaux pratiques, eux-mêmes indispensables et complémentaires au cours, et non pas simplement redondant.

La maîtrise de la programmation est un apprentissage de longue haleine qui nécessite beaucoup de rigueur et beaucoup de temps passé devant un papier avec un crayon à la main et ensuite, encore plus de temps passé devant l'ordinateur.

La maîtrise des techniques de programmation des ordinateurs est un outil extrêmement important aujourd'hui pour tout étudiant en sciences. Qu'il soit mathématicien, physicien, chimiste ou biologiste, l'étudiant en science doit impérativement savoir programmer. Que l'on soit amené à programmer en Fortran, Pascal, C ou quelqu'autre langage de programmation par la suite, les principes présentés dans ce cours, dans le cadre du lanagage Ada, demeurent utiles et les règles de programmation à respecter.

Notation utilisée

Dans ce document, nous utilisons des caractères en fonte `courrier` pour indiquer ce qui est frappé (les programmes en général) ou ce qui apparaît à l'écran de l'ordinateur.

Chapitre 1

Introduction

Étant donné un problème à résoudre, son analyse consiste à le décomposer en sous-problèmes plus simples jusqu'à atteindre des problèmes élémentaires que l'ordinateur est capable de résoudre (voir figure 1.1).

Le langage de programmation permet de transcrire le résultat de l'analyse dans un formalisme compréhensible par l'ordinateur.

Il existe différentes classes de langages de programmation. Citons :
- langages procéduraux : Ada, Fortran, Pascal, Cobol, C, ...
- langages fonctionnels : Scheme, Lisp, ...
- langages objets : Java, C++, ...

à chaque classe de langages est associée une méthodologie de conception particulière.

Dans les langages procéduraux, on distingue toujours deux éléments fondamentaux :
- les données qui correspondent en quelques sortes aux variables en mathématiques. Il est important de mentionner immédiatement que les données ne se restreignent pas à des variables numériques ;
- les traitements qui s'appliquent à des données et correspondent en quelques sortes aux fonctions mathématiques. à nouveau, il faut noter que les traitements ne sont pas forcément de nature numérique, bien au contraire.

Il est capital qu'un programme soit auto-documenté : un programme dont on ne comprend pas le fonctionnement en le lisant est inutile. En effet, un programme a une vie : il est écrit par une personne, ou une équipe de personnes plus généralement ; ensuite, il peut être modifié par d'autres personnes, parfois sans aucun lien avec celles qui l'ont écrit. Si cette équipe ne comprend pas ce qui a été écrit, le programme devient inutile. Il faut savoir que la conception et la réalisation d'un logiciel sont des opérations très coûteuses.

Aussi, il est indispensable de prendre de bonnes habitudes de rédaction de programmes dès le début. Il faudra donc toujours présenter et documenter correctement les programmes que nous

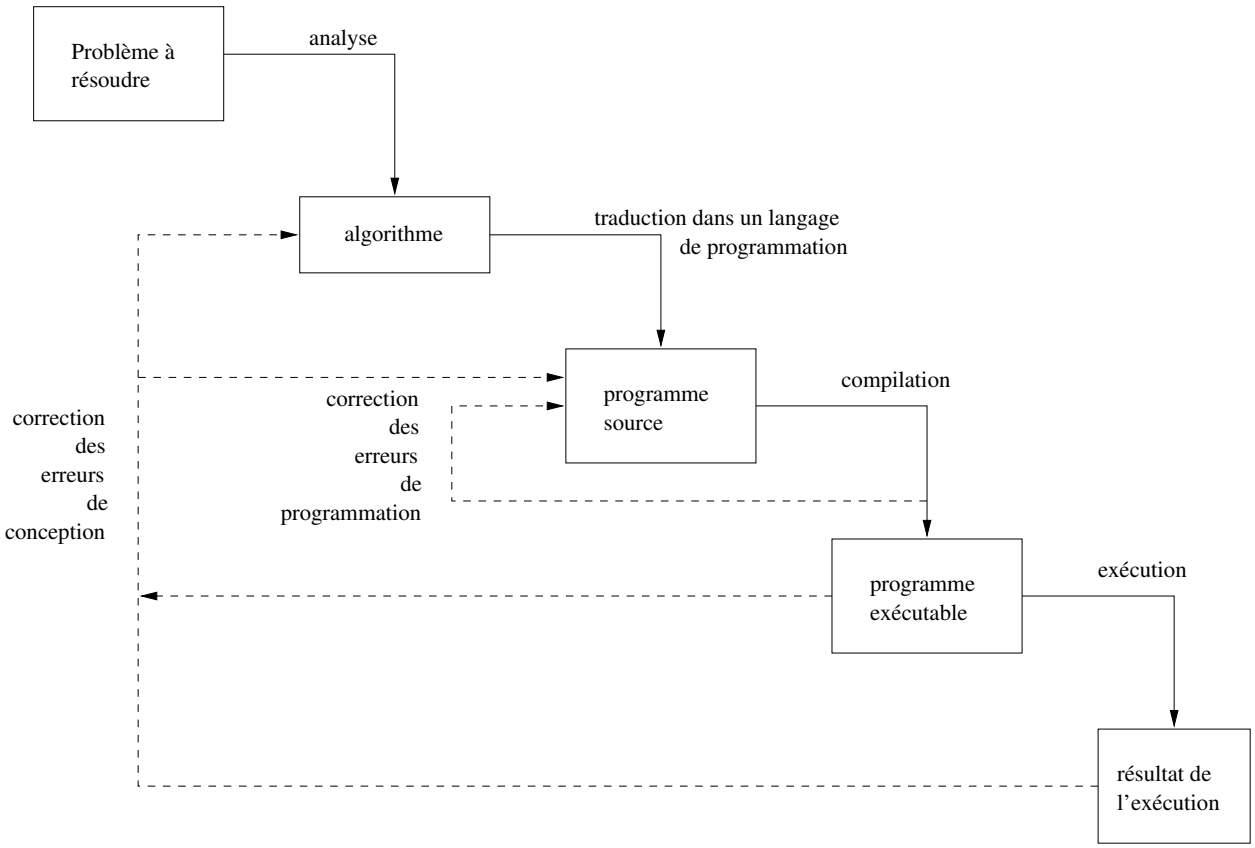

FIG. 1.1 – Résolution informatique d'un problème

écrirons.

On utilise les arbres programmatiques pour exprimer les algorithmes. Les arbres programmatiques sont une représentation graphique indépendante du langage de programmation utilisé. Ils permettent de se concentrer sur l'algorithme et laisser de côté les problèmes de syntaxe particuliers à chaque langage.

On utilise le langage de programmation Ada dans le cadre du cours, des TD et des TP. C'est un langage qui a été conçu pour être très « propre » et très riche dans ses possibilités. Seules quelques-unes de celles-ci seront présentées dans ce cours ; d'autres seront présentées dans le cours de structures de données au deuxième semestre (les accès, les paquetages) ; on le retrouvera en deuxième cycle où d'autres aspects seront étudiés (exceptions, parallélisme, généricité). Du fait de ses qualités, Ada est notamment utilisé pour programmer des applications sensibles, au CNES, à la Nasa et au déartement de la défense américaine, notamment pour le contrôle de mmissiles et de fusées ; bien entendu, Ada est également utilisé pour des applications beaucoup plus modestes. En tout état de cause, c'est un excellent langage d'initiation à la programmation procédurale.

Ada porte le prénom de celle qui fait figure de la première programmeuse, Ada Lovelace, qui a travaillé durant la seconde moitité du XIX^{e}siècle avec Charles Babbage, à la mise au point de calculateurs mécaniques, ancêtres de nos ordinateurs électroniques.

Chapitre 2

Les données, les types, les entiers

Parmi les données, on distingue immédiatement :
- les constantes qui possèdent une valeur qui ne change jamais ; par exemple, π ;
- les variables dont la valeur peut être modifiée.

Une donnée porte :
- un nom, ou identificateur, constitué de lettres, de chiffres et du caractère `_` en respectant la règle qu'un identificateur débute forcément par une lettre. Ada ne fait pas de distinction entre majuscule et minuscule. Par exemple, `abc`, `x3j26` et `a_d` sont des identificateurs valides en Ada ; `abc`, `Abc`, `aBC` sont trois manières d'écrire le même identificateur. Pour simplifier la lecture des programmes, nous prendrons la règle de toujours utiliser des lettres minuscules pour écrire les mots-clés du langage ; les identificateurs seront écrits soit uniquement en majuscules, soit avec des majuscules pour les rendre plus lisibles ; exemples : `V`, `ExempleDeProgrammeAda` ;
- un type qui spécifie l'ensemble des valeurs que peut prendre la donnée et les opérations qui lui sont applicables.

Intuitivement, une donnée est une boîte dans laquelle on peut mettre une valeur.

Dans un programme, toute donnée doit être déclarée avant d'être utilisée ; son nom et son type ne peuvent pas être modifiés par la suite dans le programme.

Quelques types souvent rencontrés sont : entier, réel, caractère, chaîne de caractères.

2.1 Le type `INTEGER`

Le type `INTEGER` représente les valeurs entières, positives et négatives.

2.1.1 Introduction

La déclaration et l'utilisation la plus simple d'une variable de type `INTEGER` s'exprime de la manière suivante en Ada :

```
I : INTEGER;

I := 10;
```

La première ligne `I : INTEGER;` déclare une variable dont le nom est `I` et dont le type est `INTEGER`. Le `;` à la fin de la ligne indique la fin de la déclaration. Déclarée de type `INTEGER`, la variable `I` pourra contenir une valeur entière, positive ou négative.

La ligne suivante `I := 10;` affecte la valeur `10` à la variable `I`. Le `;` indique la fin de l'instruction.

Remarque d'ordre général : toute déclaration et toute instruction se termine par un `;`.

Si l'on omet de déclarer une variable, le compilateur indique une erreur dans le programme. De même, si on essaie d'affecter une valeur qui ne correspond pas au type de la variable, une erreur est indiquée.

On peut également affecter le résultat d'un calcul :

```
I : INTEGER;

I := 10 + 3 * 2;
```

Dans ce cas, `I` reçoit la valeur `16`. On peut également faire des calculs sur des variables :

```
I, J : INTEGER;

I := 5;
J := 3;
I := I * J;
J := I * (J + 5) / 2;
```

Dans cet exemple, on note que l'on peut déclarer plusieurs variables de même type en une seule fois (ligne 1) ; après avoir affecté `5` à la variable `I` et `3` à la variable `J` (lignes 2 et 3), l'instruction de la ligne 4 affecte le résultat du produit de la valeur de `I` par la valeur de `J` ; la valeur de la variable `I` est donc modifiée de `5` à `15`. L'instruction de la ligne 5 modifie la valeur de `J` qui devient `15 * (3 + 5) / 2` soit `60`.

2.1.2 Opérations sur les valeurs de type `INTEGER`

Les opérateurs qui peuvent être utilisés sur des valeurs entières sont :
– `+` pour l'addition ;

- pour la soustraction ;
- `*` pour le produit ;
- `/` pour la division entière : par exemple, `5 / 2` donne `2` ;
- `**` pour l'exponentiation : `a ** b` a pour valeur a^b ;
- `rem` pour le reste de la division entière : par exemple, `5 rem 2` donne `1`, le reste de la division entière de `5` par `2`. Le résultat de `a rem b` prend le signe de `a` ;
- `mod` pour le modulo : indiquons son fonctionnement sur des exemples :

  ```
  11 mod 5     1
  11 mod -5    -4
  -11 mod -5   -1
  -11 mod 5    4
  ```

  que l'on peut résumer par deux règles : si `a` et `b` sont de même signe, la valeur absolue de `a mod b` vaut `a rem b` et son signe est celui de `a` ; si `a` et `b` sont de signe distinct, la valeur absolue de `a mod b` vaut | `b` | $-$`a rem b` et prend le signe de `b` ;
- `abs` pour la valeur absolue : `abs(I)` fournit la valeur absolue de `I`.

Lorsque plusieurs opérateurs sont utilisés dans une expression (par exemple, `2+3*12-4`), l'ordre dans lequel les opérateurs sont évalués doit être fixé ($(2+3) \times (12-4)$, ou $((2+3)*12)-4$, ou $2+3*(12-4)$, ...). Pour cela, une *priorité* est associée à chaque opérateur. On a les règles suivantes :

- `abs` est le plus prioritaire
- `**` ensuite
- `*`, `/`, `rem` et `mod` ensuite
- `+` et `-` enfin.

et les expressions sont évaluées de la gauche vers la droite ; ainsi, `a / b + c` a pour valeur $\frac{a}{b}+c$, alors que `a / (b + c)` vaut $\frac{a}{b+c}$. L'exemple pus haut (`2+3*12-4` a donc pour valeur $2+(3\times12)-4 = 34$).

2.1.3 Attributs du type `INTEGER`

Contrairement aux entiers en mathématiques, le type `INTEGER` ne comprend pas des valeurs arbitrairement grandes ou petites. Une variable de type `INTEGER` prend sa valeur dans un intervalle précis dont on peut connaître les bornes à l'aide des instructions suivantes :

- `INTEGER'FIRST` donne la plus petite valeur d'entier correspondant au type `INTEGER` ;
- `INTEGER'LAST` donne la plus grande valeur d'entier correspondant au type `INTEGER`.

`FIRST` et `LAST` sont qualifiés d'attributs du type `INTEGER`. Les valeurs de `INTEGER'FIRST` et `INTEGER'LAST` sont dépendantes de la machine sur laquelle elles sont exécutées.

2.2 Autres types d'entiers

Ada est un langage dit « fortement typé » qui incite à typer le plus précisément possible toutes les variables, c'est-à-dire, préciser au mieux l'ensemble des valeurs que peut prendre une variable. Aussi, en marge du type `INTEGER`, on a :

- le type `POSITIVE` qui est une restriction du type `INTEGER` aux seuls `INTEGER` strictement positifs (0 est exclu). Les attributs `FIRST` et `LAST` existent également pour ce type. On a :
  - `POSITIVE'FIRST` vaut 1 ;
  - `POSITIVE'LAST` vaut `INTEGER'LAST`.
- le type `NATURAL` qui est une restriction du type `INTEGER` aux seuls `INTEGER` positifs (0 est inclu). Les attributs `FIRST` et `LAST` existent également pour ce type. On a :
  - `NATURAL'FIRST` vaut 0 ;
  - `NATURAL'LAST` vaut `INTEGER'LAST`.

On peut également indiquer qu'une variable prend sa valeur dans un intervalle d'un ensemble de valeurs comme suit :

```
MOIS : INTEGER range 1..12;
```

`MOIS` est une variable qui prend une valeur entière comprise entre 1 et 12. Si on essaie de lui affecter une valeur qui ne fait pas partie de cet intervalle, une erreur est déclenchée, à la compilation ou à l'exécution, selon le moment où elle est détectée.

On pourra également définir un intervalle dans `NATURAL` ou `POSITIVE` par exemple :

```
Intervalle : NATURAL range 0..5;
AutreIntervalle : POSITIVE range 10..18;
```

2.3 Initialisation de variables lors de leur déclaration

On peut initialiser la valeur d'une variable dès sa déclaration, plutôt que d'avoir une instruction qui le fait explicitement. Pour cela, on utilise la notation suivante :

```
I : INTEGER := 10;
```

qui déclare la variable `I` de type `INTEGER` et l'initialise avec la valeur `10`.

2.4 Les constantes

Une constante est une donnée dont la valeur ne peut pas être modifiée. On la déclare de la manière suivante :

```
C_INT : constant INTEGER := -39;
C_NAT : constant NATURAL := 26;
```

La première ligne déclare une constante dénommée `C_INT` de type `INTEGER` et dont la valeur est `-39`. La deuxième ligne déclare une constante dénommée `C_NAT` de type `NATURAL` et dont la valeur est `26`.

Chapitre 3

Les arbres programmatiques ; les tests et les boucles

Les traitements concernent la réalisation des opérations ; les traitements agissent sur des données : les traitements prennent des données en entrée et fournissent des résultats.

On peut voir l'analogue avec une machine outil qui prend de la matière première en entrée et fournit des pièces usinées en sortie, ou le pétrin du boulanger qui reçoit en entrée farine, levure, sel et eau, et produit en sortie de la pâte à pains.

Dans le chapitre précédent, nous avons rencontré les traitements les plus simples :

- les opérations sur les nombres entiers ;
- l'affectation d'une valeur à une variable.

Nous décrivons maintenant les instructions avec lesquels TOUS[1] les programmes sont finalement réalisés. Nous les décrivons sous la forme d'arbres programmatiques. Outre les opérations de base et l'affectation vues au chapitre précédent, ces instructions se résument en trois structures :

- la séquence ;
- le test ;
- la boucle.

Ensuite, nous indiquerons la traduction des arbres programmatiques en Ada.

[1] véritablement TOUS : les programmes que nous écrirons en TP, les tableurs et autres traitements de textes vendus dans le commerce, les jeux vidéos, les navigateurs Internet, les compilateurs, ...

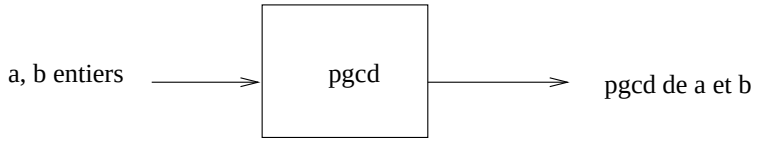

FIG. 3.1 – Un traitement consomme des données et fournit un résultat : ici, deux entiers (les données) sont transformées en un entier, résultat du traitement « pgcd ».

3.1 Les arbres programmatiques, la séquence d'instructions

Pour exprimer le résultat de l'analyse d'un problème, nous utiliserons la notion d'arbre programmatique. Un arbre programmatique permet de se focaliser sur l'algorithme en laissant de côté les particularités de tel ou tel langage. Ainsi, on utilisera les arbres programmatiques en cours d'Ada puis, au deuxième semestre, en cours de C : pour un problème donné, l'arbre programmatique sera traduit de deux manières bien différentes, selon le langage. Dans la suite du cours, nous aborderons en premier lieu les notions de traitement à l'aide d'arbres programmatiques ; de même, en TD et en TP, nous nous attacherons à obtenir l'arbre programmatique du problème posé. Ensuite seulement, nous le traduirons en Ada.

Nous allons immédiatement donner un exemple : donner l'arbre programmatique de l'action suivante :

1. saisie au clavier de la valeur de deux entiers ;
2. calcul de la moyenne de ces deux entiers ;
3. affichage de cette moyenne.

L'arbre est indiqué à la figure 3.2. Généralement, en informatique, un arbre se lit de haut en bas : sa racine est en haut, d'où partent des branches jusqu'à atteindre ses feuilles en bas. C'est simplement une question d'habitude.

On note les éléments suivants :

- un rectangle indiquant le nom de l'action : MOYENNE ;
- en dessous, un symbole $\boxed{\rightarrow}$ qui signifie « séquence d'instructions ». Les instructions en question sont situées en dessous de ce symbole. Les rectangles rattachés sous ce symbole sont exécutées dans l'ordre, de gauche à droite ;
- en dessous, à gauche, un rectangle indiquant l'action réalisée ici : « saisie de deux nombres au clavier » et en dessous, un rectangle indiquant les actions à réaliser : « saisir A et B » ;
- une fois cette action réalisée (les variables A et B contiennent donc les valeurs dont il faut calculer la moyenne), l'exécution se poursuit, en séquence, avec l'action qui se trouve à sa droite, soit « calcul de la moyenne ». à nouveau, l'action est commentée dans un rectangle rattaché à l'action effective de calcul de la moyenne et de son affectation à la variable MOY. Le symbole $\leftarrow$ est celui de l'affectation d'une valeur à une variable. Il signifie ici : affecter à la variable MOY la valeur de $\frac{A+B}{2}$;
- une fois cette action réalisée (les variables A et B contiennent donc toujours les valeurs dont il faut calculer la moyenne et MOY contient cette moyenne), l'exécution se poursuit, en séquence, avec l'action qui se trouve à droite, soit « affichage du résultat ». Encore une fois, l'action est commentée puis elle est décrite dans le rectangle en dessous ;
- puisqu'il n'y a plus d'action à la droite de celle-ci, l'exécution de l'action MOYENNE est terminée.

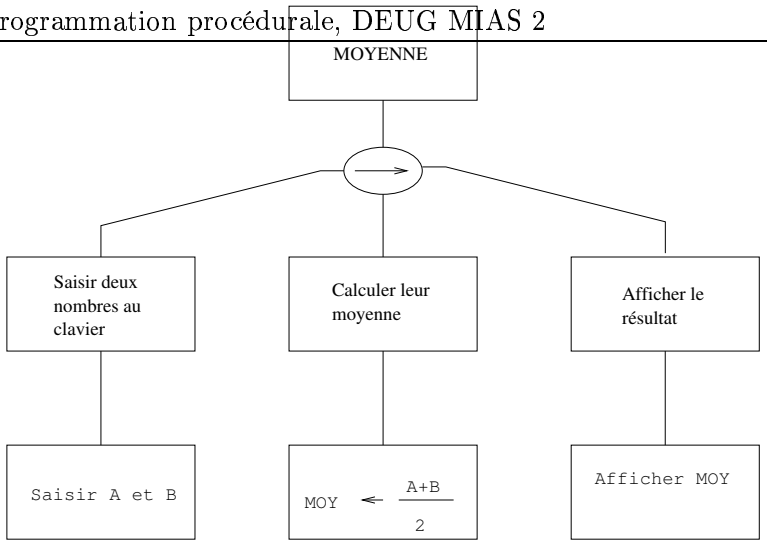

FIG. 3.2 – Arbre programmatique de l'action calcul de moyenne.

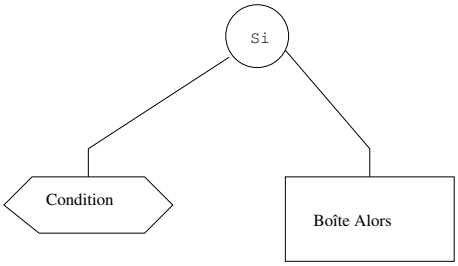

FIG. 3.3 – Arbre programmatique d'un test Si/Alors

3.2 Les tests

Les instructions de test permettent l'exécution d'instructions sous condition. Ces tests peuvent prendre deux formes, les tests Si/Alors et les tests Si/Alors/Sinon.

3.2.1 Test Si/Alors

Sous forme d'arbre programmatique, le test Si/Alors est représenté à la figure 3.3. On trouve une condition est une boîte d'instruction que nous nommerons « boîte-alors ». Remarquons que la condition, ou prédicat, est placée dans une boîte hexagonale, alors que les actions sont placées dans des rectangles : dans tout arbre programmatique, on distingue ainsi toujours condition et action à la forme de la boîte dans laquelle elle se trouve.

Le principe d'exécution est le suivant :

1. la `condition` est évaluée ; sa valeur est soit VRAI soit FAUX ;
2. si elle est vraie, les instructions de la boîte-alors sont exécutées ;

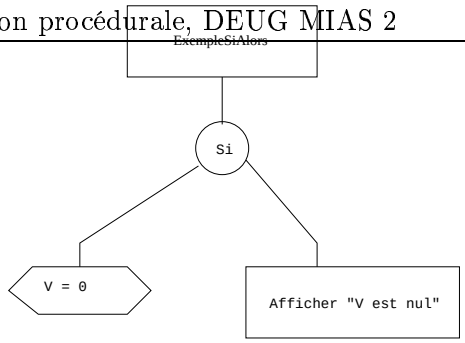

FIG. 3.4 – Exemple d'un test Si/Alors : la condition teste si la valeur de la variable est nulle. Dans ce cas, la boîte-alors est exécutée et affiche un message.

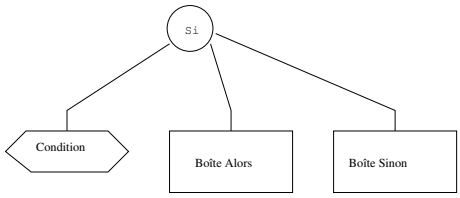

FIG. 3.5 – Arbre programmatique d'un test Si/Alors/Sinon

3. si elle est fausse, aucune instruction n'est exécutée.

La figure 3.4 propose un exemple : la valeur d'une variable est testée et un message est affiché si le nombre est nul.

3.2.2 Test Si/Alors/Sinon

Sous forme d'arbre programmatique, le test Si/Alors/Sinon est représenté à la figure 3.5. On retrouve comme précédemment une condition, mais nous avons cette fois-ci deux boîtes d'instructions, la boîte-alors située à droite de la condition, et la « boîte-sinon » située à sa droite.

Le principe d'exécution est le suivant :

1. la `condition` est évaluée ; sa valeur est soit VRAI soit FAUX ;
2. si elle est vraie, les instructions de la boîte-alors sont exécutées ;
3. si elle est fausse, les instructions de la boîte-sinon sont exécutées.

Dans un test Si/Alors/Sinon, soit la boîte-alors soit la boîte-sinon est exécutée ; à l'exécution du test, l'une des deux est forcément exécutée mais jamais les deux.

La figure 3.6 propose un exemple : la valeur d'une variable est testée et un message est affiché en fonction de la parité du nombre.

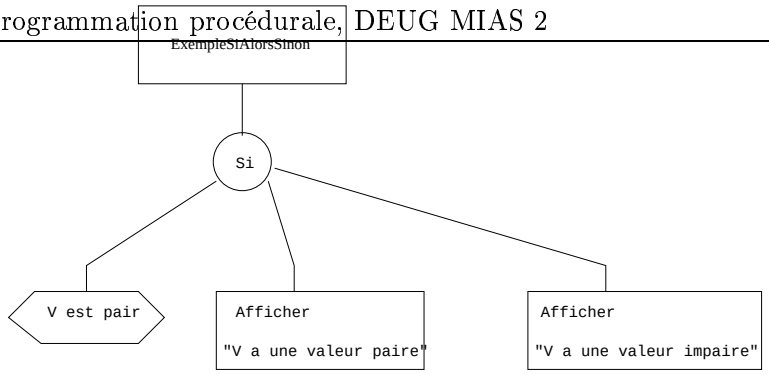

FIG. 3.6 – Exemple d'un test Si/Alors/Sinon : la condition teste si la valeur de la variable est paire. Selon le cas, la boîte-alors ou la boîte-sinon est exécutée et affiche un message.

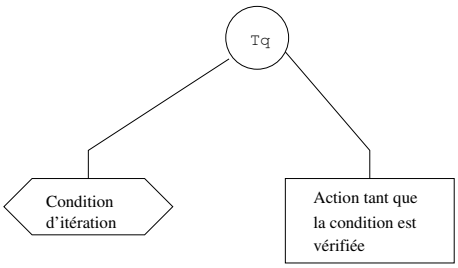

FIG. 3.7 – Arbre programmatique d'une boucle Tant-que

3.3 Les boucles

Les boucles permettent de réaliser plusieurs fois de suite la même séquence d'instructions. Il en existe plusieurs formes et nous traiterons des boucles Tant-que et des boucles Pour.

3.3.1 La boucle Tant-que

Dans une boucle Tant-que, la séquence d'instructions est répétée tant qu'une certaine condition est vérifiée. L'arbre programmatique correspondant est indiqué à la figure 3.7.

Le principe d'exécution d'une boucle Tant-que est le suivant :

1. la condition est calculée. Comme dans un test, sa valeur est VRAI ou FAUX ;
2. si la condition est vérifiée, l'action indiquée à sa droite est exécutée. à l'issue de l'exécution de cette action, on revient à l'étape 1 ;
3. si la condition n'est pas vérifiée, l'exécution du Tant-que est terminée.

On note que si la condition n'est pas vérifiée lors de sa première évaluation, l'action indiquée n'est jamais exécutée.

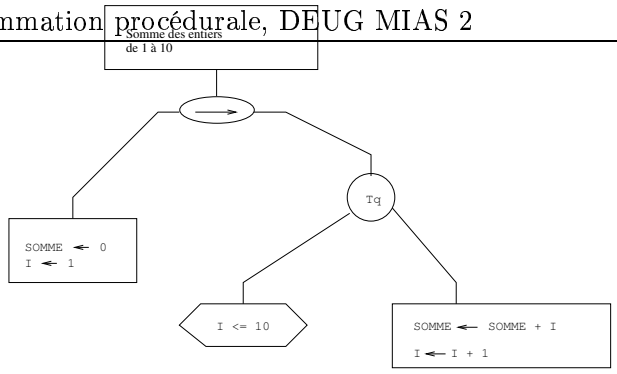

FIG. 3.8 – Un arbre programmatique pour une action consistant à faire la somme des entiers de 1 à 10 et utilisant une boucle Tant-que.

Règle à respecter : il faut impérativement s'assurer que la condition d'itération devient fausse à un moment pour en sortir.

Considérons un exemple : écrire une action qui calcule la somme des 10 premiers entiers naturels (les entiers de 1 à 10). L'arbre programmatique en est indiqué à la figure 3.8. Pour cela, on considére une variable I qui égrène les valeurs de 1 à 10 et, à chaque itération, on ajoutera sa valeur à une autre variable, SOMME initialisée au préalable à 0. Par ailleurs, à chaque itération, la variable I est incrémentée. Lorsque I atteint la valeur 11, on a terminé, ce qui explique la condition d'itération du Tant-que.

On peut composer des actions plus complexes en utilisant plusieurs traitements dans une même action. Ainsi, on peut définir une action qui, pour les entiers de 5 à 1, affiche s'il est pair ou impair (voir figure 3.9).

3.3.2 La boucle Pour

Dans une boucle Pour, on connaît le nombre d'itérations à réaliser. L'arbre programmatique correspondant est indiqué à la figure 3.10.

Le principe d'exécution d'une boucle Pour est le suivant :

1. une variable (appelée « indice de boucle ») est utilisée ; elle va successivement prendre les valeurs indiquées dans l'intervalle. Elle est tout d'abord initialisée avec la borne inférieure de l'intervalle ;
2. l'action associée est exécutée ; la valeur de l'indice peut être utilisée ;
3. l'indice de boucle est incrémenté ;
4. si l'indice de boucle est toujours inférieur ou égal à la borne supérieure de l'intervalle, on retourne à l'étape 2 ; sinon, l'exécution de la boucle est terminée.

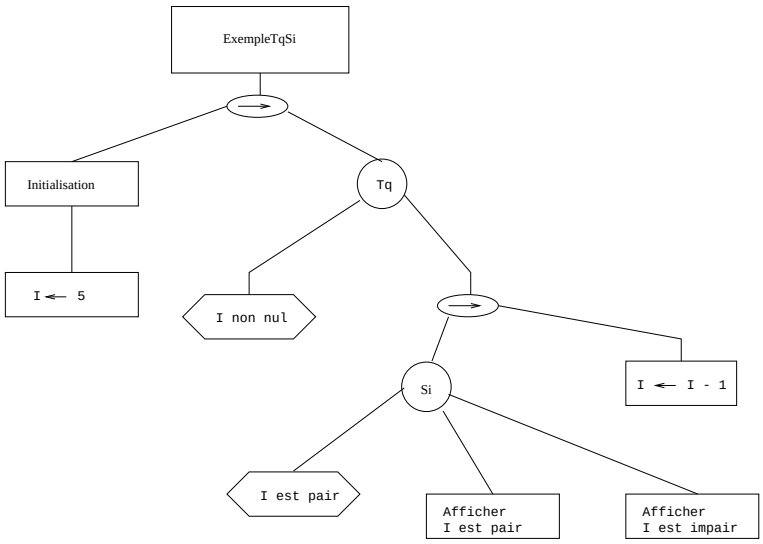

FIG. 3.9 – Un arbre programmatique pour une action consistant, pour les entiers de 5 à 1, à afficher s'il est pair ou impair. On voit que l'on peut composer des actions complexes en combinant les traitements de base : séquence, test, boucle.

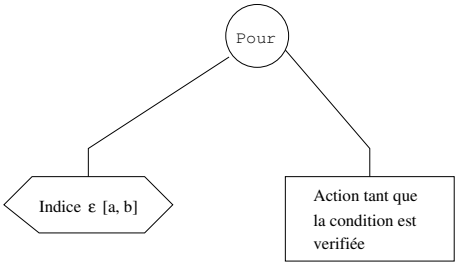

FIG. 3.10 – Arbre programmatique d'une boucle Pour

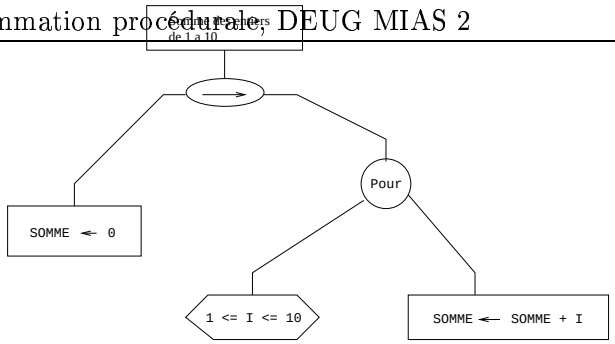

FIG. 3.11 – Un arbre programmatique pour une action consistant à faire la somme des entiers de 1 à 10 et utilisant une boucle Pour.

Considérons à nouveau l'exemple traité avec la boucle Tant-que : écrire une action qui calcule la somme des 10 premiers entiers naturels (les entiers de 1 à 10). On utilise cette fois-ci une boucle Pour. L'arbre programmatique est indiqué à la figure 3.11. Cette fois-ci, la variable I qui égrène les valeurs de 1 à 10 est l'indice de boucle. L'intervalle dans lequel sa valeur varie est spécifié. Son incrémentation est cette fois automatique. L'action action consiste donc simplement à ajouter le contenu courant de l'indice de boucle à la variable SOMME.

La traduction de ce problème à l'aide d'une boucle Pour est beaucoup plus naturelle qu'avec une boucle Tant-que ; en effet, on connaît à l'avance le nombre d'itérations à réaliser. Aussi, la bonne solution à ce problème consiste à utiliser une boucle Pour et non une boucle Tant-que.

Lorsque l'on aborde la programmation structurée, on hésite souvent dans le choix de la boucle à utiliser, entre une boucle Tant-Que et une boucle Pour. En fait, il n'y a aucune hésitation à avoir et il faut suivre le principe suivant : une boucle Pour s'emploie lorsque l'on connaît par avance le nombre d'itérations qui seront effectuées ; une boucle Tant-Que s'emploie lorsque ce nombre d'itérations est inconnu.

Notons que l'indice d'une boucle Pour peut varier de manière croissante ou décroissante. Selon le cas, l'une ou l'autre de ces deux écritures est plus pratique.

3.4 Exercices

Donner un arbre programmatique pour les problèmes suivants :

1. échanger la valeur de deux variables : avant l'action, la variable A contient une valeur x, la variable B contient une valeur y ; à la suite de l'action, A contient la valeur y et B contient la valeur x ;
2. avant l'action, la variable A contient une valeur x, la variable B contient une valeur y ; à la suite de l'action, A contient la valeur la plus petite parmi x et y et B contient la valeur la plus grande parmi x et y ;

3. avant l'action, trois variables A, B et C qui sont les coefficients d'une équation du second degré (A x^2+ B x+ C = 0). Afficher les solutions de l'équation en considérant que certaines variables peuvent avoir une valeur nulle ;

4. avant l'action, la variable A contient une année (un entier). L'action détermine si c'est une année bissextile et affiche un message correspondant ;

5. avant l'action, trois variables A, B et C qui contiennent respectivement les valeurs x, y et z. à l'issue de l'action, la valeur de A est inférieure ou égale à celle de B, elle-même inférieure ou égale à celle de C ;

6. avant l'action, deux horaires, l'un de départ, l'autre d'arrivée sous la forme heure, minute et secondes (HD, MD, SD, HA, MA, SA) dans la même journée ; à l'issue de l'action, la durée du trajet est affichée. D'une part, il est interdit de convertir les horaires en secondes ; d'autre part, il faut définir les champs contenant des secondes comme `range 0..59`, de même pour les minutes ;

7. avant l'action, on a une variable N de type entier. à l'issue de l'action, X contient sa racine carrée par excès, c'est-à-dire le plus petit entier tel que $X^2 \geq N$;

8. avant l'action, deux entiers A et B. L'action affiche la valeur de A élevée à la puissance de la valeur de B. Bien entendu, il est interdit d'utiliser l'opérateur `**` ;

9. avant l'action, la variable N contient un entier. L'action calcule et affiche sa factorielle (attention aux cas particuliers) ;

10. avant l'action, la variable N contient un entier. L'action saisit N nombres entiers au clavier et affiche leur somme ;

11. saisir une suite d'entiers au clavier terminée par la valeur -1 et affiche leur somme ;

12. multiplication égyptienne : c'est une manière de calculer un produit en ne faisant que des additions, des multiplications par 2 et des divisions entières par 2. Considérons un exemple : quel est le produit de 36 par 43 ? On effectue la division entière 43 par 2 (on aurait pu prendre 36 à la place bien entendu) et on recommence sur le résultat jusqu'à atteindre 1 ; en vis-à-vis de chaque résultat, on écrit le produit de 36 par 2, puis encore par 2, ... On obtient :

    $$\begin{array}{ccccccccccc} 43 & \xrightarrow{/2} & 21 & \xrightarrow{/2} & 10 & \xrightarrow{/2} & 5 & \xrightarrow{/2} & 2 & \xrightarrow{/2} & 1 \\ 36 & \xrightarrow{\times 2} & 72 & \xrightarrow{\times 2} & 144 & \xrightarrow{\times 2} & 288 & \xrightarrow{\times 2} & 576 & \xrightarrow{\times 2} & 1152 \end{array}$$

    On fait ensuite la somme des multiples de 36 apparaissant dans la deuxième ligne qui correspondent aux valeurs impaires de la première ligne, soit 36, 72, 288 et 1152 ; c'est le résultat du produit de 36 par 43.

    Donner l'arbre programmatique de la multiplication égyptienne en considérant que les valeurs à multiplier sont dans deux variables A et B avant l'action, et que le produit est affiché à l'écran à l'issue de l'action.

Un arbre programmatique indique une action. En Ada, une action débute par le mot-clé `begin` et se termine par le mot-clé `end` et `;`. En outre, il faut déclarer les variables utilisées dans l'action ; ces déclarations sont réalisées avant le mot-clé `begin`.

3.5.1 Séquence

Les instructions composant une séquence doivent être traduites l'une après l'autre en les considérant de gauche à droite.

On reprend l'action MOYENNE vue à la page 15. On obtient le programme Ada suivant :

```
--
-- Programme moyenne.adb
--
-- 10 septembre 1999.
--
-- Ph. Preux, ULCO
--
-- Ce programme saisit deux nombres entiers au clavier, calcule leur
-- moyenne et l'affiche à l'écran.
--

with Text_IO; -- pour autoriser les entrées/sorties de caractères
use Text_IO;
with Ada.Integer_Text_IO; -- pour autoriser les entrées/sorties d'INTEGER
use Ada.Integer_Text_IO;

procedure MOYENNE is

I, J, MOY : INTEGER;

begin
  GET (I);
  GET (J);
  MOY := (I + J) / 2;
  PUT ("La moyenne de");
  PUT (I);
  PUT (" et");
```

```
    PUT (J);
    PUT (" est");
    PUT_LINE (MOY);
  end MOYENNE;
```

Nous décrivons ce programme qui peut être saisi tel quel et compilé pour être exécuté.

Les 10 premières lignes, celles précédées de `--`, sont des commentaires. Ce ne sont pas des instructions ; elles sont là pour documenter le programme et expliquer son fonctionnement ; leur présence est indispensable : **tout programme non commenté est inutile**. Ce commentaire indique le nom du programme qui est aussi le nom du fichier qui le contient. Il indique la date du jour où il a été écrit ainsi que son auteur. Ensuite, il contient quelques lignes de commentaires expliquant son objet.

On trouve ensuite deux instructions (`with TEXT_IO;` et `use TEXT_IO;`) qui doivent figurer en début de tout programme Ada qui effectue des entrées/sorties de caractères. De même, les deux lignes suivantes doivent figurer en début de tout programme Ada qui effectue des entrées/sorties de variables de type `INTEGER`. On remarquera l'utilisation de commentaires.

à la suite, on trouve la ligne `procedure MOYENNE is` qui indique que l'on va définir une *procédure*, c'est-à-dire un ensemble de déclarations et instructions qui réalisent une action. Cette procédure porte un nom, ici `MOYENNE`, qui est suivi du mot-clé `is` pour indiquer que sa définition suit.

On trouve alors la déclaration de trois variables de type `INTEGER` et de nom `I`, `J` et `MOY`.

à la suite de la déclaration des variables utilisées dans la procédure, les instructions la composant suivent après le mot-clé `begin`. Toutes les instructions situées entre ce `begin` et le `end MOYENNE;` (mot-clé `end` suivi du nom de la procédure en cours de définition) constituent le corps de la procédure. Ces instructions sont donc ici au nombre de 9.

Les deux premières instructions `GET(.)` effectuent la saisie au clavier d'une valeur et la stockent dans la variable indiquée, la variable `I` pour la première, la variable `J` pour la seconde.

La ligne suivante `MOY := (I + J) / 2;` effectue le calcul de la moyenne des valeurs contenues dans les variables `I` et `J` et l'affecte à la variable `MOY`.

Les 6 lignes suivantes affichent le résultat à l'écran. `PUT` affiche la valeur entre `(.)` à l'écran à la queue leu leu. `PUT_LINE` affiche également la valeur entre `(.)` à l'écran puis passe à la ligne suivante.

On note que, conformément à ce qui avait été dit auparavant, les identificateurs sont en majuscules et les mots-clés en minuscule.

Si on tape et compile le programme, on peut ensuite l'exécuter. Si on entre alors les nombres `13` et `17`, on aura l'affichage :

```
La moyenne de 13 et 17 est 15.
```

Ordre d'exécution des instructions dans une séquence : les instructions s'exécutent l'une après l'autre, de « haut en bas ». L'exécution d'une instruction ne débute que lorsque l'exécution de l'instruction précédente est terminée.

3.5.2 Les tests

Traduction des tests Si/Alors

La traduction d'un test si/alors (arbre programmatique de la figure 3.3) est la suivante :

```
if condition then
  instructions de la boîte-alors
end if;
```

`if`, `then` et `end if` sont des mots-clés à reproduire tels quels dans le programme Ada.

Le principe d'exécution de cette instruction est le suivant :

1. la `condition` est calculée ; sa valeur est soit VRAI soit FAUX ;
2. si elle est vraie, les instructions situées entre `then` et `end if` sont exécutées dans l'ordre habituel ;
3. si elle est fausse, les instructions situées entre `then` et `end if` **ne sont pas** exécutées ;
4. l'exécution des instructions se poursuit avec l'instruction qui suit `end if`.

Par exemple, l'arbre programmatique de l'exemple donné à la figure 3.4 se traduit de la manière suivante :

```
  I : INTEGER;

begin
  if I = 0 then
    PUT_LINE ("I est nul");
  end if;
end ExempleSiAlors;
```

La condition permet de tester l'égalité entre deux termes (opérateur `=`), l'inégalité entre deux termes (opérateur `/=`), l'ordre entre deux termes (opérateurs `<`, `<=`, `>`, `>=`).

Traduction des tests Si/Alors/Sinon

La traduction d'un test si/alors/sinon (arbre programmatique de la figure 3.5) est la suivante :

```
  if condition then
    instructions de la boîte-alors
  else
    instructions de la boîte-sinon
  end if;
```

`if`, `then`, `else` et `end if` sont des mots-clés à reproduire tels quels dans le programme Ada.

Dans un test `if`/`then`/`else`, soit les instructions situées entre `then` et `else` sont exécutées, soit les instructions situées entre `else` et `end if` sont exécutées, mais pas les deux.

L'arbre programmatique donné en exemple à la figure 3.6 se traduit de la manière suivante :

```
  V : INTEGER;

begin
  if V rem 2 = 0 then
    PUT_LINE ("V est pair");
  else
    PUT_LINE ("V est impair");
  end if;
end ExempleSiAlorsSinon;
```

3.5.3 Les boucles

Traduction des boucles Tant-que

La traduction en Ada d'une boucle Tant-que est la suivante :

```
  while condition loop
    instructions composant la boucle
  end loop;
```

`while`, `loop` et `end loop` sont des mots-clés à reproduire tels quels dans le programme Ada.

Principe d'exécution d'une boucle Tant-que :

1. la condition est testée ;
2. si elle est vérifiée, les instructions situées entre `loop` et `end loop` sont exécutées en séquence, dans l'ordre habituel ;
3. une fois la dernière instruction exécutée (celle qui précède `end loop`), le processus est itéré, c'est-à-dire que l'on revient à l'étape 1 : la condition est donc à nouveau évaluée et si elle est de nouveau vérifiée, les instructions situées entre `loop` et `end loop` sont ré-exécutées ; ceci continue tant que la condition est vérifiée ;

- si la condition n'est pas, ou plus, vérifiée, il y a sortie de la boucle : l'exécution des instructions se poursuit avec l'instruction qui suit `end loop` et se poursuit ensuite dans l'ordre habituel.

L'arbre programmatique de la figure 3.8 se traduit de la manière suivante :

```
  SOMME : NATURAL := 0;
  I : POSITIVE range 1..10 := 1;

begin
  while I <= 10 loop
    SOMME := SOMME + I;
    I := I + 1;
  end loop;
end ExempleTq;
```

On peut traduire des actions qui combinent plusieurs traitements. Par exemple, l'arbre programmatique de la figure 3.9 se traduira comme suit :

```
  with text_io;
  use text_io;
  with Ada.Integer_Text_Io; -- pour autoriser les entrées/sorties d'INTEGER
  use Ada.Integer_Text_Io;

  procedure ExempleTqSi is

    I : INTEGER ;

  begin
    I := 5;
    while I /= 0 loop
      if I rem 2 = 0 then
        PUT (I);
        PUT_LINE (" est pair");
      else
        PUT (I);
        PUT_LINE (" est impair");
      end if;
      I := I - 1;
    end loop;
  end ExempleTqSi;
```

L'exécution de ce programme fournit l'affichage suivant :

```
5 est impair
4 est pair
3 est impair
2 est pair
1 est impair
```

Remarquons que le nombre est précédé d'un grand nombre d'espaces disgracieux : ceci est le comportement de la procédure `put` qui prévoit de l'espace pour écrire un grand entier. Pour faire plus joli, on peut demander à ce que le nombre ne soit écrit par exemple que sur 2 caractères en remplaçant l'appel `put (I)` par `put (I,2)` qui donne un résultat beaucoup plus joli :

```
5 est impair
4 est pair
3 est impair
2 est pair
1 est impair
```

Traduction des boucles Pour

La traduction en Ada d'une boucle Pour a la forme suivante :

```
for indice in a..b loop
  instructions du corps de la boucle
end loop;
```

`for`, `loop` et `end loop` sont des mots-clés à reproduire tels quels dans le programme Ada.
Le fonctionnement en est le suivant :

1. l'`indice` est une variable entière déclarée implicitement (il n'est pas nécessaire de la déclarer auparavant). Cette variable (appelée « indice de boucle ») est initialisée avec la valeur `a` (un entier) ;
2. les instructions situées entre `loop` et `end loop` sont ensuite exécutées dans l'ordre habituel ;
3. une fois exécutée la dernière instruction située avant `end loop`, l'indice de boucle est incrémenté automatiquement ;
4. si l'indice de boucle est inférieur ou égal à `b`, les itérations se poursuivent, c'est-à-dire que l'on revient au point 2 ;

5. si l'indice de boucle est strictement supérieur à `b`, les itérations s'arrêtent. L'exécution du programme se poursuit avec l'instruction qui suit `end loop`.

Ainsi, l'arbre programmatique de la figure 3.11 se traduit de la manière suivante :

```
  SOMME : NATURAL := 0;

begin
  for I in 1..10 loop
    SOMME := SOMME + I;
  end loop;
end ExemplePour;
```

La traduction d'une boucle Pour décroissante est réalisée en ajoutant le mot-clé `reverse` après `in`. Ainsi, on peut écrire la boucle précédente sous la forme d'une boucle décroissante comme suit :

```
  SOMME : NATURAL := 0;

begin
  for I in reverse 1..10 loop
    SOMME := SOMME + I;
  end loop;
end ExemplePour;
```

La valeur de la variable SOMME sera la même dans les deux cas puisque l'addiction est commutative. Par contre, d'une manière générale, une boucle Pour parcourue de manière croissante ne donnera pas le même résultat que la boucle Pour parcourue de manière décroissante.

3.5.4 Exercices

Reprendre tous les arbres programmatiques correspondant aux exercices de la section 3.4 et les traduire en Ada.

3.6 Le type `FLOAT`

Afin de pouvoir réaliser des opérations sur des nombres « réels », on décrit maintenant le type `FLOAT`.

On définit une variable pouvant contenir une valeur réelle de la manière suivante :

```
  X : FLOAT;
```

Pour effectuer des entrées/sorties de valeurs de ce type, il faut ajouter les deux lignes suivantes en début de progrmamme :

```
with Ada.Float_Text_IO;
use Ada.Float_Text_IO;
```

Les constantes ont la forme suivante : `3.15` pour 3,15, `4.23e-3` pour 0.00423, où `e-3` signifie 10^{-3}.

En Ada, il est interdit d'affecter une valeur réelle à une variable de type entier, et vice-versa :

```
procedure AffectationErronnee is

  I : INTEGER;
  X : FLOAT;

begin
  I := X;
  X := I;
end AffectationErronnee;
```

Les deux affectations `I := X` et `X := I` déclenchent une erreur de compilation. Pour pouvoir réaliser ce genre d'opérations, il faut effectuer une conversion comme suit :

```
procedure AffectationCorrecte is

  I : INTEGER;
  X : FLOAT;

begin
  I := INTEGER (X);
  X := FLOAT (I);
end AffectationCorrecte;
```

La conversion d'un nombre `FLOAT` en `INTEGER` par l'instruction `INTEGER(X)` donne la valeur arrondie de `X` ; si la partie décimale de `X` est inférieure à 0.5, la conversion produit la partie entière de `X` ; si la partie décimale de `X` est supérieure ou égale à 0.5, la conversion produit la partie entière de `X+1`.

```
procedure ExempleDeConversion is

   I : INTEGER;
   X : FLOAT;

begin
   X := 10.9;
   I := INTEGER (X);
   -- maintenant, I vaut 11

   X := 10.49;
   I := INTEGER (X);
   -- maintenant, I vaut 10

   X := 10.5;
   I := INTEGER (X);
   -- maintenant, I vaut 11
end ExempleDeConversion;
```

Plus généralement, dans une expression arithmétique, on ne peut pas mélanger des entiers avec des réels. Il est obligatoire de convertir les nombres pour additionner soit des entiers avec des entiers, soit des réels avec des réels. Ainsi, en reprenant les déclarations de l'exemple précédent, on ne peut pas écrire l'instruction `I := I + F` : il faut forcément écrire `I := I + INTEGER (X)`.

3.6.1 Fonctions standards sur les `FLOAT`

Les fonctions d'entrées/sorties et les fonctions transcendentales habituelles sur les nombres réelles sont disponibles à condition d'ajouter les deux lignes :

```
-- pour les fonctions transcendantales
with Ada.Numerics.Elementary_Functions;
use Ada.Numerics.Elementary_Functions;

-- pour les entrées/sorties de FLOAT
with Ada.Float_Text_IO
use Ada.Float_Text_IO;
```

Dès lors, on peut extraire des racines carrées, calculer des fonctions trigonométriques, ... Supposons que `X` et `Y` sont déclarées de type `FLOAT`, on peut utiliser les fonctions suivantes :

- `Sqrt (X)` pour calculer $\sqrt{x}$;
- `Sin (X)` pour calculer $\sin x$;
- `Cos (X)` pour calculer $\cos x$;
- `Tan (X)` pour calculer $\tan x$;
- `Log (X)` pour calculer $\log x$, le logarithme naturel ;
- `X ** Y` pour calculer x^y.

Chapitre 4

Les procédures paramétrées, les fonctions

Comme on l'a vu dans l'introduction, l'objet de l'analyse consiste à décomposer un problème en un ensemble de sous-problèmes, jusqu'à atteindre un stade où ces sous-problèmes sont élémentaires pour l'ordinateur. Il est particulièrement important que cette décomposition apparaisse dans le programme. On introduit donc les notions de procédures et de fonctions qui effectuent la résolution d'un sous-problème. Si un même sous-problème doit être résolu plusieurs fois pour la résolution d'un problème, alors la procédure en question sera exécutée plusieurs fois. Les données de la procédure, ou de la fonction, sont ses paramètres d'entrée ; leurs résultats sont les paramètres en sortie pour les procédures, la valeur renvoyée dans le cas d'une fonction.

Une procédure, ou une fonction, a un en-tête qui a la forme indiquée à la figure 4.1.

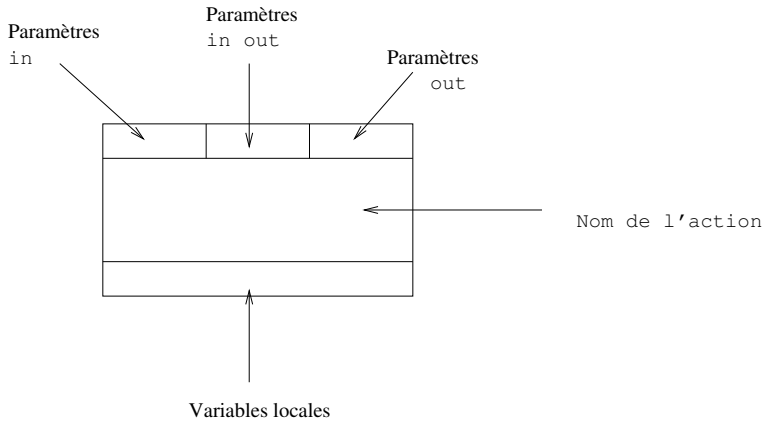

FIG. 4.1 – Arbre programmatique de l'en-tête d'une procédure.

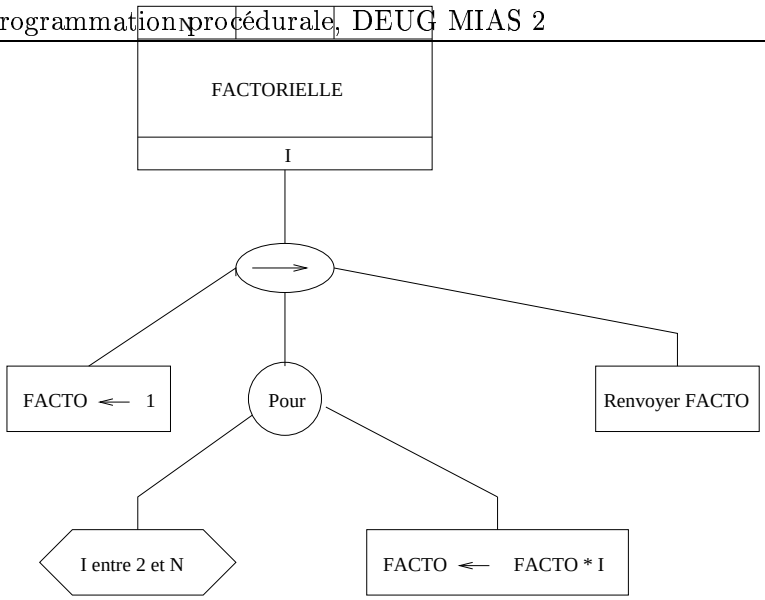

FIG. 4.2 – Arbre programmatique d'une action qui calcule la factorielle d'un nombre entier naturel.

4.1 Exemple de fonction

4.1.1 Définition d'une fonction qui calcule la factorielle d'un naturel

Voyons immédiatement un exemple, une fonction qui calcule la factorielle d'un nombre entier. Nous avons vu au chapitre 3 que la factorielle d'un nombre entier se calcule à l'aide d'une boucle Pour. On obtient donc un arbre tel celui de la figure 4.2.

Cet arbre se traduit comme suit :

```
function FACTORIELLE (N : NATURAL) return POSITIVE is

  FACTO : POSITIVE := 1;

begin
  for I in 2..N loop
    FACTO := FACTO * I;
  end loop;
  return FACTO;
end FACTORIELLE;
```

On note :

– une ligne `function FACTORIELLE (N : NATURAL) return POSITIVE is` qui indique :

- que l'on débute la définition d'une fonction ;
  - que cette fonction se nomme `FACTORIELLE` ;
  - qu'elle prend un paramètre en entrée dont la valeur se trouve dans la variable `N` de type `NATURAL` ; c'est donc un entier positif, celui pour lequel on veut calculer la factorielle ;
  - que la fonction retourne un résultat de type `POSITIVE` ; c'est la valeur de la factorielle du paramètre d'entrée (la factorielle d'un nombre positif est un nombre strictement positif, d'où les types du paramètre et de la fonction) ;
  - le mot-clé `is` indique que le corps de la fonction suit ;
- la définition d'une variable locale à la fonction `FACTORIELLE` qui se nomme `FACTO`, de type `POSITIVE`, initialisée à la valeur 1. Une variable locale à une fonction, ou une procédure, n'existe que dans la fonction, ou la procédure, dans laquelle elle est définie. La notion de variable locale est très importante et nous y reviendrons ;
- ensuite, on trouve, entre `begin` et `end FACTORIELLE`, les instructions qui composent le corps de la fonction. On note l'instruction `return FACTO` qui stoppe l'exécution de la fonction (les instructions qui suivraient ne seraient pas exécutées) et indique la valeur associée à la fonction, ici, celle de la variable `FACTO`.

Le calcul de la factorielle de 0 est un cas particulier. Telle qu'elle a été écrite, la fonction `FACTORIELLE` fournit le bon résultat. En effet, le résultat, la variable `FACTO`, est initialisé avec la valeur 1 ; si `N` vaut 0, la boucle devient :

```
for I in 2..0 loop
  FACTO := FACTO * I;
end loop;
```

L'indice de boucle est initialisé avec la valeur 2 qui est supérieure à la borne supérieure de l'intervalle (0). Aussi, le corps de la boucle n'est pas exécuté. L'instruction `return FACTO` renvoie donc la valeur initiale de `FACTO`, soit 1, qui est bien la valeur de la factorielle de 0.

4.2 Utilisation de la fonction factorielle

4.2.1 Premier exemple

La compilation de la définition précédente de la fonction factorielle ne produit aucun résultat en tant que tel. Par contre, la fonction peut être utilisée pour calculer la factorielle d'un nombre à chaque fois qu'on le souhaite.

Ainsi, on peut maintenant écrire un programme qui saisit un nombre positif au clavier et affiche sa factorielle.

```
with Ada.Integer_Text_Io;
use Ada.Integer_Text_Io;

procedure CalculFactorielle is

  NOMBRE : NATURAL;
  F : POSITIVE;

  function FACTORIELLE (n : NATURAL) return POSITIVE is

    FACTO : POSITIVE := 1;

  begin
    for I in 2..n loop
      FACTO := FACTO * I;
    end loop;
    return FACTO;
  end FACTORIELLE;

begin
  GET (NOMBRE);
  F := FACTORIELLE (NOMBRE);
  PUT (F);
end CalculFactorielle;
```

On a placé la fonction `FACTORIELLE` à l'intérieur de la procédure `CalculFactorielle`. Les variables `NOMBRE` et `F` sont locales à la procédure `CalculFactorielle` ; aussi, elles ne sont accessibles qu'au sein de la procédure `CalculFactorielle` ET des procédures et fonctions internes à la procédure `CalculFactorielle`.

La procédure `CalculFactorielle` effectue les actions suivantes :

1. saisie au clavier de la valeur du nombre dont on veut obtenir la factorielle *via* l'instruction `GET (NOMBRE)` ;
2. calcul de la factorielle de ce nombre et affectation de celle-ci à la variable `F` ;
3. affichage de la valeur de la variable `F`, c'est-à-dire de la factorielle du nombre saisi auparavant.

4.2.2 Deuxième exemple

Une fois une fonction ou une procédure définie, on peut l'utiliser plusieurs fois si cela est nécessaire. Par exemple, on peut utiliser la fonction `FACTORIELLE` dans une fonction `PERMUTATION` qui calcule le nombre de permutations possibles pour choisir p objets parmi n. Mathématiquement, on sait que la formule est :

$$P_p^n = \frac{n!}{(n-p)!}$$

On voit qu'il faut calculer 2 factorielles pour cela. On peut écrire la fonction `PERMUTATION` comme suit :

```
function PERMUTATION (M, P : NATURAL) return POSITIVE is

  function FACTORIELLE (N : NATURAL) return POSITIVE is

    FACTO : POSITIVE := 1;

  begin
    for I in 2..n loop
      FACTO := FACTO * I;
    end loop;
    return FACTO;
  end FACTORIELLE;

begin
  return FACTORIELLE (M) / FACTORIELLE (M-P);
end PERMUTATION;
```

On retrouve la fonction `FACTORIELLE` dans la fonction `PERMUTATION` qui lui fait appel deux fois.

Au niveau vocabulaire, les paramètres qui sont indiqués dans la définition de la fonction (`M` et `P` pour la fonction `PERMUTATION` et `N` pour la fonction `FACTORIELLE` dans l'exemple) sont qualifiés de « formels ». Ceux qui apparaissent dans l'appel de la fonction (`M` et `M-P` dans l'exemple) sont qualifiés d'«effectifs ».

4.3 Les procédures

En marge des fonctions, il existe également les procédures qui doivent être mises en œuvre dans les cas suivants :

- pour réaliser un traitement (procédure) qui ne produit pas de résultat ;
- pour réaliser un traitement (procédure) un sous-problème) qui ne produit pas de résultat ;
- pour réaliser un traitement qui produit plusieurs résultats : en effet, une fonction ne peut produire qu'un seul résultat.

Dans une procédure, il faut indiquer pour chaque paramètre si :

- c'est un paramètre d'entrée dont la valeur est lue dans la procédure mais jamais modifiée, c'est-à-dire, que la valeur de ce paramètre est une constante durant l'exécution de la procédure, constante dont la valeur est celle du paramètre effectif. Il est alors interdit d'écrire une instruction qui modifierait sa valeur (message d'erreur à la compilation). Dans ce cas, le mot-clé `in` précède le type du paramètre ;
- c'est un paramètre de sortie dont la valeur est affectée dans la procédure, mais jamais lue. Il est alors interdit d'écrire une instruction qui lirait sa valeur (message d'erreur à la compilation). Dans ce cas, le mot-clé `out` précède le type du paramètre ;
- c'est un paramètre d'entrée/sortie dont la valeur est à la fois lue et affectée dans la procédure. Dans ce cas, les mots-clés `in out` précèdent le type du paramètre.

Considérons un exemple fictif :

```
procedure ExempleParametres (N : in NATURAL ;
  X : in FLOAT ; XX : in out FLOAT ; L : out POSITIVE) is
...
```

dans lequel ... indique les instructions de la procédure.

La procédure `ExempleParametres` déclare 4 paramètres :

- `N` est un paramètre en entrée de type `NATURAL` ;
- `X` est un paramètre en entrée de type `FLOAT` ;
- `XX` est un paramètre en entrée/sortie de type `FLOAT` ;
- `L` est un paramètre en sortie de type `POSITIVE`.

Ainsi, il est interdit d'écrire des instructions telles `N := 78` et `get (N)` dans cette procédure, puisque `N` est un paramètre de mode `in`. De même, il est interdit d'écrire des instructions telles `... := L + 3` et `put (L)` puisque `L` est de mode |out|.

Notons que dans une fonction, tous les paramètres sont forcément en entrée. Aussi, le mot-clé `in` n'est pas mentionné.

On peut ajouter les points suivant :

- à l'intérieur d'une procédure ou d'une fonction, un paramètre `in` ne peut être rencontré qu'à droite d'un `:=`, dans un prédicat, comme paramètre d'une fonctione, ou encore comme paramètre `in` d'une procédure appelée (par exemple, la procédure `put`) ;
- à l'intérieur d'une procédure, un paramètre `in` ne peut être rencontré qu'à gauche d'un `:=` ou comme paramètre `out` d'une procédure appelée (par exemple, la procédure `get`) ;
- à l'intérieur d'une procédure, un paramètre `in out` peut être rencontré n'importe où.

Bien entendu, il faut toujours spécifier le plus précisément possible le caractère `in`, `out` ou `in out` des paramètres : il ne faut pas déclarer `in out` un paramètre qui peut n'être déclaré que `in` ou `out`.

4.4 Visibilité des variables locales

Règle : une variable locale est accessible (ou visible) dans la procédure ou la fonction où elle est définie, ainsi que dans les procédures et fonctions qui sont elles-mêmes locales.

Une variable locale peut porter le nom d'une variable déjà définie par ailleurs. Dans ce cas, la nouvelle définition masque (on dit aussi « surchage ») la précédente dans la procédure, ou la fonction, où elle est définie, ainsi que dans les procédures et fonctions elles-mêmes locales.

Considérons l'exemple suivant :

```
procedure P1 is
  A, B : INTEGER;

  procedure P2 is
    A, C : POSITIVE;

  begin
    -- ici, les variables visibles sont :
    --    A est une variable de type POSITIVE déclarée dans P2
    --    B est une variable de type INTEGER déclarée dans P1
    --    C est une variable de type POSITIVE déclarée dans P2
    ...
  end P2;

  procedure P3 is
    C, D : FLOAT;

  begin
    -- ici, les variables visibles sont :
    --    A et B sont des variables de type INTEGER déclarée dans P1
    --    C et D sont des variables de type FLOAT déclarée dans P3
    ...
  end P3;

begin
  -- ici, les variables visibles sont :
```

```
    -- A et B sont des variables de type INTEGER
    ...
  end P1;
```

De même, du point de vue de sa visibilité, un paramètre peut être considéré comme une variable locale :

```
  procedure P1 (X, Y : in FLOAT) is
    A, B : INTEGER;

    procedure P2 (A, Y : in NATURAL) is
      C : POSITIVE;

    begin
      -- ici, les variables visibles sont :
      --    A est une variable de type NATURAL : paramètre de P2
      --       cette définition surchage la précédente, dans P1
      --    B est une variable de type INTEGER : variable définie dans P1
      --    C est une variable de type POSITIVE : variable définie dans P2
      --    X est une variable de type FLOAT : paramètre de P1
      --    Y est une variable de type NATURAL : paramètre de P2
      --       cette définition surchage la précédente, dans P1
      ...
    end P2;

    procedure P3 (B, C : in FLOAT) is
      D : FLOAT;

    begin
      -- ici, les variables visibles sont :
      --    A est une variable de type INTEGER : définie dans P1
      --    B et C sont des variables de type FLOAT : paramètres de P3
      --    D est une variable de type FLOAT : variable locale de P3
      --    X est une variable de type FLOAT : paramètre de P1
      --    Y est une variable de type FLOAT : paramètre de P1
      ...
    end P3;
```

```
begin
   -- ici, les variables visibles sont :
   --    A et B sont des variables de type INTEGER
   --    X est une variable de type FLOAT (paramètre)
   --    Y est une variable de type FLOAT (paramètre)
   ...
end P1;
```

Comme pour les variables, les procédures et les fonctions peuvent être définies localement. Ainsi, dans ce dernier exemple, les procédures `P2` et `P3` sont locales à la procédure `P1` ; aussi, seule la procédure `P1` peut les utiliser. La procédure `P3` peut appeler `P2` car `P3` est définie avant `P2` ; par contre, `P2` ne peut pas appeler `P3`.

4.5 Exercices

1. pour la procédure suivante, indiquer pour chacun des paramètres (`a`, `b`, `c`, `xd`) s'il est `in`, `out` ou `in out`.

```
procedure test (a, b, c, xd)
begin
  a := b + 3 * c - xd;
  if a = 7 then
    c := b * xd - 21;
  end if;
  put (xd - a * b);
end test;
```

2. qu'imprime le programme suivant :

```
with text_io;
use text_io;
with Ada.Integer_text_IO;
use Ada.Integer_text_IO;

  procedure p1 is
    a, b : INTEGER;
    x : INTEGER;
    z : INTEGER := 7;

    procedure p2 (a : in INTEGER; b : in out INTEGER ; c, d : in out INTEGER) is
```

```
    x : NATURAL := 10;
    y : INTEGER := -25;
    begin
      put (a); put (b); put (c); put (d); put (x); put (y); put (z); new_line;
      b := a;
      d := x;
      z := a;
      c := z * y;
      put (a); put (b); put (c); put (d); put (x); put (y); put (z); new_line;
    end p2;

  function f (g : INTEGER) return INTEGER is
    z : INTEGER := 1;
  begin
    put (a); put (b); put (g); put (x); put (z); new_line;
    return g * z / a;
  end f;

begin
  a := 3;
  b := 78;
  x := a * b;
  put (a); put (b) ; put (x); put (z); new_line;
  p2 (x, b, a, x);
  put (a); put (b) ; put (x); put (z); new_line;
  p2 (f (x), b, a, x);
  put (a); put (b) ; put (x); put (z); new_line;
end p1;
```

3. écrire l'arbre programmatique de l'action qui affiche les `N` premiers termes de la suite de Fibonacci. Le traduire en une procédure Ada. Rappelons que la suite de Fibonacci se définit par :

$$\begin{aligned} u_0 &= u_1 = 1 \\ u_n &= u_{n-1} + u_{n-2} \end{aligned}$$

4. écrire l'arbre programmatique, puis le traduire en Ada sous forme d'une fonction, de l'action qui calcule le nombre de combinaisons de p objets parmi n ;

5. écrire l'arbre programmatique, puis le traduire en Ada sous forme d'une fonction, de l'action qui calcule le PGCD de deux nombres entiers naturels et le renvoie. Cette fonction sera par exemple appelée par : `put (PGCD (65, 27))` qui affichera à l'écran le pgcd de 65 et 27 ;

6. écrire l'arbre programmatique, puis le traduire en Ada sous forme d'une fonction, de l'action qui calcule le PPCM de deux nombres entiers naturels et le renvoie. Cette fonction sera par exemple appelée par : `put (PPCM (12, 20))` qui affichera le ppcm de 12 et 20 ;

7. écrire l'arbre programmatique, puis le traduire en Ada sous forme d'une fonction, de l'action qui convertit un entier d'une base dans une autre. Cette fonction prendra trois paramètres `NATURAL` : le nombre à convertir, la base dans laquelle il est exprimé, la base dans laquelle il faut l'exprimer. Par exemple, `put (conversion (27, 10, 2))` affichera la valeur 11011 (c'est-à-dire, la représentation en base 2 de 27 exprimé en base 10), tandis que `put (conversion (2012, 3, 5))` affichera la valeur 214 (c'est-à-dire, la représentation en base 5 de la valeur 2012 exprimé en base 3).

   Idée : la base 10 constituant la base pivot de transformation d'une base dans une autre, il peut être judicieux d'écrire deux fonctions locales à `conversion`, l'une qui transforme un nombre de la base 10 en une autre base, l'autre qui transforme un nombre exprimé dans une base quelconque en base 10.

Chapitre 5

Les tableaux

5.1 Introduction

Les variables que l'on a rencontrées jusqu'à maintenant possèdent une valeur ; elles forment l'analogue aux valeurs scalaires en mathématiques. Il est possible de définir et utiliser des variables analogues aux vecteurs et matrices : ce sont les tableaux.

Intuitivement, un tableau est un ensemble de cases numérotées, chaque case contenant une valeur d'un type bien précis ; toutes les cases du tableau contiennent une valeur de même type. Ainsi,

```
T : array (1..10) of INTEGER;
```

définit `T` comme étant un tableau de 10 éléments de type `INTEGER`. Les cases sont numérotées de `1` à `10`. Pour accéder à la valeur d'un élément de tableau, on utilise la notation `T(3)` pour accéder au 3^{e}élément de `T`.

5.2 Les tableaux mono-dimensionnels

5.2.1 Déclaration

La déclaration d'un tableau en Ada s'effectue de la manière suivante :

```
nom_du_tableau : array (a..b) of type_des_elements_du_tableau;
```

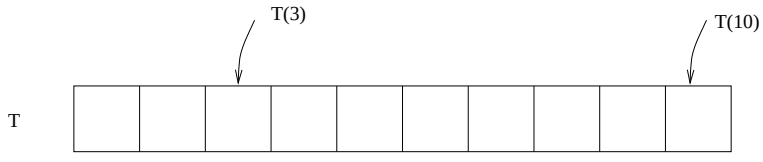

FIG. 5.1 – Représentation graphique d'un tableau : un ensemble ordonné de boîtes.

5.2.2 Égalité de deux tableaux

On peut tester l'égalité de deux tableaux avec l'opérateur `=` et leur différence avec l'opérateur `/=`. Deux tableaux sont égaux si tous leurs éléments sont deux à deux égaux. Il faut pour cela qu'ils soient de même type.

5.2.3 Les attributs des tableaux

Considérons la définition de tableau suivante :

```
T : array (5..20) of INTEGER;
```

- `T'FIRST` est la borne inférieure des indices du tableau `T`, soit `5` dans l'exemple ;
- `T'LAST` est la borne supérieure des indices du tableau `T`, soit `20` dans l'exemple ;
- `T'RANGE` est l'intervalle des valeurs prises par l'indice du tableau `T`, soit `5..20` dans l'exemple ;

- `T'LENGTH` est le nombre de valeurs différentes prises par l'indice du tableau `T`, soit `16` dans l'exemple.

L'attribut `RANGE` est très pratique pour écrire une boucle qui parcourt tous les éléments du tableau. En effet, on peut écrire :

```
procedure initTableau is

T : array (5..20) of INTEGER;

begin
  for I in T'RANGE loop
    T (I) := 0;
  end loop;
end initTableau;
```

Cette procédure initialise tous les éléments du tableau `T` à la valeur 0. L'intérêt de l'utilisation de l'attribut est que l'on peut modifier la définition de `T` sans avoir à modifier la boucle.

5.2.4 Les agrégats

Un agrégat est un tableau constant. Considérons la déclaration d'un tableau de 5 entiers :

```
T : array (1..5) of INTEGER;
```

On peut lui affecter une valeur comme suit :

```
-- exemple 1
t := (45, -84, 3, 1, 9);

-- exemple 2
t := (2 | 3 => 0, others => 1);

-- exemple 3
t := (10, 20, others => 1);

-- exemple 4
t := (2..4 => 10, others => 1);
```

Ce qui s'explique comme suit :

1. affecte `45` à `t(1)`, `-84` à `t(2)`, `3` à `t(3)`, `1` à `t(4)`, `9` à `t(5)` ;
2. affecte `0` à `t(2)` et `t(3)`, 1 aux autres éléments du tableau ;
3. affecte `10` à `t(1)`, `20` à `t(2)` et `1` aux autres éléments ;
4. affecte `10` aux éléments `t(2)`, `t(3)`, `t(4)`, 1 aux autres éléments.

5.2.5 Les tranches de tableau

On peut accéder à une tranche de tableau, c'est-à-dire, un ensemble d'éléments consécutifs d'un tableau. Ainsi, ayant la déclaration :

```
t : array (1..20) of INTEGER;
```

on peut accéder aux éléments d'indice 5 à 10 comme s'ils constituaient un tableau en écrivant : `t(5..10)`. Par exemple, on pourra avoir l'instruction : `t(5..10) := (1, 2, 3, 4, 5, 6)` qui affecte la valeur `1` à `t(5)`, `2` à `t(6)`, ...

5.3 Passage de tableau en paramètre

Deux possiblités s'offrent à nous pour passer des tableaux en paramètre :

- soit nous souhaitons définir une procédure qui travaille sur des tableaux d'un type bien précis (un nombre d'éléments et un typage des éléments du tableau précis). Dans ce cas, on doit définir un type tableau ;
- soit nous voulons définir une procédure qui travaille sur des tableaux dont les éléments sont d'un certain type, mais leur nombre variable. Dans ce cas, on doit utiliser des tableaux dits « non contraints ».

Nous décrivons ces deux possibilités.

5.3.1 Déclaration de type tableau

La déclaration d'un type tableau se fait de la manière suivante :

```
type Type_Tableau is array (NATURAL range 1..10) of INTEGER;
```

qui indique :
- `Type_Tableau` le nom du type qui est défini dans la ligne ;
- `NATURAL range 1..10` spécifie les indices des éléments du tableau qui sont de type `NATURAL` et portent les indices de `1` à `10` ; on retrouve ici la syntaxe déclarant un intervalle de valeurs (voir la section 2.2) ;
- `INTEGER` spécifie le type des éléments du tableau.

En résumé, on a déclaré un tableau de 10 éléments de type `INTEGER` portant les indices `1` à `10`.

On peut ensuite déclarer des variables de type `Type_Tableau` :

```
T1, T2 : Type_Tableau;
```

On peut également déclarer des paramètres formels de ce type puis les utiliser dans le corps de la procédure qui peut être appelée librement avec des paramètres effectifs de type `Type_Tableau` :

```
procedure ExempleTableau is

  type Type_Tableau is array (NATURAL range 1..10) of INTEGER;

  T1, T2 : Type_Tableau;

  procedure ExempleParametreTableau (T : in out Type_Tableau) is

  begin
    for I in T'range loop
      T (I) := 0;
    end loop;
  end ExempleParametreTableau;

begin
  ExempleParametreTableau (T1);
  ExempleParametreTableau (T2);
end ExempleTableau;
```

Ainsi, cet exemple montre une procédure, `ExempleParametreTableau` qui reçoit un paramètre de type `Type_Tableau` et affecte tous les éléments du tableau à une valeur nulle. Ainsi, après l'instruction `ExempleParametreTableau (T1);`, tous les éléments du tableau `T1` vallent `0`.

Par contre, l'exemple suivant produit une erreur à la compilation :

```
procedure ExempleTableauErronne is

  type Type_Tableau is array (NATURAL range 1..10) of INTEGER;

  T1 : Type_Tableau;
  T2 : array (NATURAL range 1..10) of INTEGER;

  procedure ExempleParametreTableau (T : in out Type_Tableau) is

  begin
    for I in T'range loop
      T (I) := 0;
    end loop;
  end ExempleParametreTableau;

begin
  ExempleParametreTableau (T1);
  ExempleParametreTableau (T2);
end ExempleTableauErronne;
```

Notons bien que le type spécifié dans la déclaration du tableau `T2` est une copie rigoureuse de la spécification du type `Type_Tableau`.

En fait, en Ada, si l'on veut indiquer que deux variables sont de mêmes types, on doit déclarer un type et spécifier ce type pour les deux variables. Si l'on ne déclare pas les deux variables avec le même nom de type, cela signifie, en Ada, que, même si les deux types ont la même définition, ils ne représentent pas la même chose.

Ainsi dans l'exemple précédent, les tableaux `T1` et `T2` ont des types spécifiés de la même manière, mais le fait de ne pas utiliser un unique nom de type dans leur déclaration entraîne le compilateur à considérer que la ressemblance entre les deux déclarations n'est qu'apparente et que dans le fond, les deux tableaux correspondent à deux choses bien différentes.

5.3.2 Tableaux non contraints

Si l'on souhaite maintenant écrire une procédure qui met à `0` tous les éléments d'un tableau comportant un nombre quelconque d'entiers, la méthode précédente n'offre pas de solutions. Il faut pour cela déclarer un type de tableau dit « non contraint », c'est-à-dire un tableau dont on spécifie le type des éléments, mais pas leur nombre. Pour cela, on écrit :

```
procedure ExempleTableauNonContraint is

  type Type_Tableau is array (NATURAL range <>) of INTEGER;

  T1 : Type_Tableau (1..5);
  T2 : Type_Tableau (23..54);

  procedure ExempleParametreTableau (T : in out Type_Tableau) is

  begin
    for I in T'range loop
      T (I) := 0;
    end loop;
  end ExempleParametreTableau;

begin
  ExempleParametreTableau (T1);
  ExempleParametreTableau (T2);
end ExempleTableauNonContraint;
```

On notera la déclaration du type `Type_TableauNonContraint` dans laquelle aucun intervalle d'indices n'est spécifié ; au contraire on note le symbole `<>`.

La déclaration des deux variables tableaux `T1` et `T2` demande alors de spécifier les bornes de ces tableaux, de `1` à `5` pour `T1`, de `23` à `54` pour `T2`. Donc, `T1` et `T2` sont de même type, mais ne contiennent pas le même nombre d'éléments.

Le paramètre formel de la procédure `ExempleParametreTableau` est déclaré du même type `Type_Tableau`.

Dans le corps de la procédure `ExempleTableauNonContraint`, on peut alors appeler la procédure `ExempleParametreTableau` avec les deux tableaux `T1` et `T2`.

On comprend bien maintenant tout l'intérêt des attributs de tableaux en regardant la boucle Pour dans la procédure `ExempleParametreTableau` : le paramètre étant un tableau non contraint,

les bornes des indices de ces tableaux ne peuvent pas être connues avant l'exécution de la procédure, lorsque la valeur du paramètre effectif est connue, donc son nombre d'éléments.

5.4 Exercice

1. On se donne le type suivant :

   `type Vecteur is array (NATURAL range <>) of FLOAT`. Écrire l'arbre programmatique puis la fonction qui calcule la somme de deux vecteurs ;

2. avec la même définition de `Vecteur`, écrire l'arbre programmatique puis la fonction qui calcule le produit d'un vecteur par un scalaire ;

3. avec la même définition de `Vecteur`, écrire l'arbre programmatique puis la fonction qui calcule le produit scalaire de deux vecteurs ;

4. avec la même définition de `Vecteur`, écrire l'arbre programmatique puis la fonction qui recherche une valeur dans un tableau et renvoie son indice, -1 si la valeur n'est pas trouvée dans le tableau, le plus petit indice s'il y en a plusieurs occurences ;

5. avec la même définition de `Vecteur`, écrire l'arbre programmatique puis la fonction qui calcule le produit vectoriel de deux vecteurs ;

6. avec la même définition de `Vecteur`, écrire l'arbre programmatique puis la fonction qui recherche le début de la plus longue monotonie croissante dans le vecteur et renvoie l'indice de son 1erélément. (Une monotonie croissante est une sous-suite dont les éléments ont des valeurs croissantes, non strictement.)

5.5 Les matrices et tableaux multi-dimensionnels

5.5.1 Déclaration de tableaux multi-dimensionnels

On peut déclarer des tableaux multi-dimensionnels pour représenter, par exemple, des matrices. On a une déclaration du genre :

```
matrice : array (NATURAL range 1..3, NATURAL range 2..6) of FLOAT;
```

Cette ligne déclare un tableau dénommé `matrice` ayant 3 lignes de 5 éléments de type `FLOAT`.

On accède alors à un élément en spécifiant ses indices dans les différentes dimensions, en respectant l'ordre de la déclaration ; par exemple, on aura `matrice(1,5)` ou `matrice(3,2)` ; les lignes et colonnes de `matrice` sont numérotées comme suit :

	2	3	4	5	6
1					
2					
3					

On peut déclarer des tableaux ayant autant de dimensions que l'on veut.

5.5.2 Les tranches et agrégats multi-dimensionnels

Comme pour les tableaux mono-dimensionnels, on peut spécifier des agrégats pour les tableaux multi-dimensionnels. Par exemple, pour la variable `matrice` déclarée plus haut, on pourra écrire :

```
matrice := (1 => (1.0, 2.0, others => 0.0),
            2 => (3.0, 4.0, others => 0.0),
            3 => (5.0, 6.0, others => 0.0));
```

ce qui signifie que les éléments de `matrice` seront affectés comme suit :

1.0	2.0	0.0	0.0	0.0
3.0	4.0	0.0	0.0	0.0
5.0	6.0	0.0	0.0	0.0

Pour initialiser tous les éléments d'une matrice à une valeur, on pourra également écrire :

```
matrice := (others => (others => 0.0));
```

5.5.3 Attributs

Les attributs existent pour les tableaux multi-dimensionnels. Il faut alors spécifier la dimension en paramètre de l'attribut. Pour la variable `matrice` déclarée plus haut, on aura :

- `T'FIRST(1)` vaut `1`, `T'FIRST(2)` vaut `2` ;
- `T'LAST(1)` vaut `3`, `T'LAST(2)` vaut `6` ;
- `T'RANGE(1)` vaut `1..3`, `T'RANGE(2)` vaut `2..6`;
- `T'LENGTH(1)` vaut `2`, `T'LENGTH(2)` vaut `5.` ;

5.5.4 Les tableaux multi-dimensionnels non contraints

Comme pour les tableaux mono-dimensionnels, on peut déclarer des tableaux multi-dimensionnels non contraints. Ainsi, on peut écrire :

```
procedure ExempleMatriceNonContrainte is
```

```
  type Type_Matrice is array (NATURAL range <>, NATURAL range <>) of INTEGER;

  M1 : Type_Matrice (1..5, 10..20);
  M2 : Type_Matrice (23..54, 123..129);

  procedure ExempleParametreMatrice (M : in out Type_Matrice) is

  begin
    for I in M'range(1) loop
      for J in M'range(2) loop
        M (I, J) := 0;
      end loop;
    end loop;
  end ExempleParametreMatrice;

begin
  ExempleParametreMatrice (M1);
  ExempleParametreMatrice (M2);
end ExempleMatriceNonContrainte;
```

5.6 Exercices

1. On se donne le type suivant :

   `type Matrice is array (NATURAL range <>, NATURAL range <>) of FLOAT.`

   Écrire l'arbre programmatique puis la fonction qui calcule la somme de deux matrices ;
2. avec la même définition de `Matrice`, écrire l'arbre programmatique puis la fonction qui calcule la transposée d'une matrice ;
3. avec la même définition de `Matrice`, écrire l'arbre programmatique puis la fonction qui calcule le produit de deux matrices ;
4. avec la même définition de `Matrice`, écrire l'arbre programmatique puis la fonction qui calcule le déterminant d'une matrice ;
5. écrire l'arbre programmatique puis la procédure qui calcule un carré magique d'ordre impair ; cet ordre est passé en paramètre à la procédure. Les carrés magiques seront construits sur le modèle suivant :

4	9	2
3	5	7
8	1	6

11	24	7	20	3
4	12	25	8	16
17	5	13	21	9
10	18	1	14	22
23	6	19	2	15

22	47	16	41	10	35	4
5	23	48	17	42	11	29
30	6	24	49	18	36	12
13	31	7	25	43	19	37
38	14	32	1	26	44	20
21	39	8	33	2	27	45
46	15	40	9	34	3	28

6. écrire l'arbre programmatique puis la procédure qui calcule le triangle de Pascal jusqu'à une certaine ligne passée en paramètre. On résoudra cet exercice de deux manières :
   (a) en utilisant un tableau bi-dimensionnel, chaque ligne correspondant à une itération. Pour 5, on obtient la table suivante :

1	1				
1	2	1			
1	3	3	1		
1	4	6	4	1	
1	5	10	10	5	1

   (b) en utilisant un tableau mono-dimensionnel dans lequel on ne stocke qu'une seule ligne, la ligne en cours de calcul.

5.7 Problème

(Ce problème est extrait du DS de novembre 2000.)

On souhaite écrire un programme qui construit un histogramme à partir de valeurs numériques saisies au clavier. Rappelons d'abord ce qu'est un histogramme.

Supposons que les données soient les suivantes : 1, 8, 6, 9, 2, 2, 1, 2, 8, 5 (comprises entre 1 et 10). Supposons que l'on souhaite réaliser un histogramme pour chaque valeur entière : il y a 2 fois la donnée 1, 3 fois la donnée 2, 1 fois la donnée 5, 1 fois 6, 1 fois 8, 1 fois 9, 0 fois les autres entiers compris entre 1 et 10. Cela nous donne donc l'histogramme : 2, 3, 0, 0, 1, 1, 0, 1, 1, 0.

On aurait tout aussi bien pu faire un histogramme avec un pas de 2 unités : on aurait alors :
- 5 données dans $\{1, 2\}$,
- 0 données dans $\{3, 4\}$,
- 2 données dans $\{5, 6\}$,

– 1 données dans $\{7, 8\}$,

– 1 données dans $\{9, 10\}$,

donc l'histogramme 5, 0, 2, 1, 1. Chaque valeur de l'histogramme correspond à une classe ou un ensemble de valeurs de données.

Bien entendu, on peut prendre un pas quelconque du moment qu'il est strictement positif. S'il est supérieur au nombre de données, toutes les données seront regroupées dans une seule classe.

Question 1

Donner l'arbre programmatique d'une action `Construction Histogramme` qui construit l'histogramme d'un ensemble de données.

Ces données sont des nombres entiers strictement positifs. Elles sont saisies au clavier et sont traitées à la volée. Les données sont terminées par une valeur nulle ou négative (qui ne doit pas être prise en compte). L'histogramme sera stocké dans un tableau d'éléments indicés de 1 à N+1. Le pas sera stocké dans une variable. L'élément N+1 de ce tableau histogramme contiendra le nombre de valeurs trop grandes pour être stockées dans les N premiers éléments de l'histogramme. Ainsi, si l'on reprend l'exemple précédent avec le jeu de données suivant : 1, 8, 6, 18, 9, 2, 25, 2, 1, 38, 2, 8, 5, un histogramme de pas 1 pour N=10 sera : 2, 3, 0, 0, 1, 1, 0, 1, 1, 0, 3. Le dernier élément du tableau (3) indique qu'il y a 3 valeurs (18, 25, 38) qui sont trop grandes pour être mises dans l'histogramme.

Outre la construction de l'histogramme, l'action comptera le nombre de valeurs lues et mémorisera la valeur minimale et la valeur maximale des valeurs comptabilisées dans le N+1^{e} élément de l'histogramme (dans l'exmple, ce serait 18 et 38).

Question 2

Donner la traduction de cette action sous la forme d'une procédure Ada. Le tableau histogramme ne sera pas passé en paramètre mais sera une variable globale du programme. Le pas sera passé en paramètre. La procédure renverra le nombre de valeurs lues et les valeurs minimales et maximales.

On prendra soin à la déclaration des paramètres de la procédure.

On inclura cette procédure dans un programme Ada complet qui déclare le tableau histogramme et dont le programme principal appelle la procédure précédente correctement.

Question 3

Écrire une procédure `AfficheHistogramme` qui affiche à l'écran le contenu de l'histogramme sous la forme suivante :

```
Histogramme de pas 1 :
2, 3, 0, 0, 1, 1, 0, 1, 1, 0, 3
```

si le pas demandé est 1 et les données fournies celles indiquées plus haut.

Question 4

Écrire une procédure `AfficheHistogrammeAvecDesEtoiles` qui affiche le contenu de l'histogramme à l'écran sous la forme suivante (toujours pour l'exemple ci-dessus) :

```
   1 **
   2 ***
   3
   4
   5 *
   6 *
   7
   8 *
   9 *
  10
> 10 ***
```

Si le pas est égal à 2, on aurait :

```
   1 *****
   3
   5 **
   7 *
   9 *
> 10 ***
```

Question 5

Donner l'arbre programmatique d'une fonction `DistributionSymetrique` qui retourne un résultat `BOOLEAN` qui indique si la distribution des valeurs est symétrique. Pour cela, on utilisera l'histogramme construit précédemment qui doit avoir une symétrie d'axe central et avoir 0 valeurs dans la classe N+1. Clairement, la distribution ci-dessus n'est pas symétrique.

Question 6

Traduire cet arbre en une fonction Ada. Comme pour les procédures précédentes, le tableau histogramme ne sera pas passé en paramètre mais sera une variable globale du programme.

Chapitre 6

Types caractère, chaînes de caractères et autres types prédéfinis

6.1 Type CHARACTER

6.1.1 Introduction

Pour l'instant, nous n'avons rencontré que des valeurs numériques. Hors, cela ne représente qu'une petite partie des types existants. Il est très courant d'effectuer des traitements sur des caractères, des mots, des textes, ... Pour cela, nous allons tout d'abord présenter le type `CHARACTER`

Un `CHARACTER` peut être une lettre, majuscule ou minuscule, un chiffre, un signe de ponctuation, un espace blanc, ... La déclaration d'une variable de type `CHARACTER` se fait comme suit :

```
C : CHARACTER;
```

On peut lui affecter ensuite une valeur comme suit :

```
C := '5';
```

Cette instruction affecte le caractère `5` à la variable `C`. **Il ne faut surtout pas confondre le caractère `'5'` avec la valeur numérique `5` : ce sont deux objets très différents.** Les valeurs numériques correspondent aux nombres mathématiques ; les valeurs de type `CHARACTER` n'ont pas d'analogue en mathématiques. Pour les distinguer, on note `'5'` le caractère (entre `'`) et simplement `5` le nombre.

On peut utiliser une variable `CHARACTER` dans n'importe quelle instruction, du moment que cela a un sens. On peut par exemple tester la valeur d'une variable `CHARACTER` :

```
with Text_IO;
use Text_IO;

procedure EssaiCharacter is

  C : CHARACTER;

begin
  GET (C);
  if C = 'o' then
    PUT_LINE ("vous avez tape o");
  elsif C = 'n' then
    PUT_LINE ("vous avez tape n");
  else
    PUT_LINE ("vous n'avez tape ni o, ni n");
  end if;
end EssaiCharacter;
```

Les caractères sont ordonnés selon l'ordre lexicographique :

'0' < '1' < ... < '9' < ... < 'A' < 'B' < 'C' < ... < 'Z' < 'a' < 'b' < 'c' < ... < 'z' <

On peut utiliser les opérateurs de test = et /=, et les opérateurs de comparaison <, <=, > et >= dans des conditions de tests ou de boucles.

6.1.2 Les attributs du type CHARACTER

- CHARACTER'PRED('f') fournit 'e' ;
- CHARACTER'SUCC('p') fournit 'q'.

À chaque caractère est associé un nombre ; on peut convertir un CHARACTER en NATURAL comme suit :

- conversion CHARACTER en NATURAL par : CHARACTER'POS('a') fournit 48 ;
- conversion NATURAL en CHARACTER par : CHARACTER'VAL(48) fournit 'a' ;

6.2 Les chaînes de caractères

6.2.1 Opérations élémentaires

Définition et déclaration

Les tableaux de CHARACTER sont omni-présents en informatique : on les appelle des « chaînes de caractères ». Une chaîne de caractères est donc un tableau comportant un certain nombre d'éléments

de type `CHARACTER`. Par leur importance, un type particulier est associé à ces tableaux, le type `STRING`. Par exemple, la déclaration :

```
CHAINE : STRING (1..10);
```

définit la variable `CHAINE` comme étant une chaîne de 10 caractères. Comme pour les tableaux de nombres, on peut manipuler soit la chaîne dans son ensemble, soit par tranche, soit caractère par caractère.

Affectation

On peut lui affecter une valeur constante :

```
CHAINE := "abcdefghij";
```

Il faut alors impérativement que la constante soit composée du nombre de caractères spécifié dans la déclaration de la chaîne. Ainsi,

```
CHAINE := "ab";
```

entraîne une erreur de compilation car `CHAINE` a été déclarée comme comportant 10 `CHARACTER`. Si l'on veut affecter le caractère `'a'` au premier élément de `CHAINE`, `'b'` au deuxième élément de `CHAINE` sans plus, il faut impérativement indiquer que les huit caractères suivants sont blancs :

```
CHAINE := "ab        ";
```

et l'on peut écrire également :

```
CHAINE := ('a', 'b', others => ' ');
--
-- autre écriture :
-- CHAINE := ("ab", others = ' ');
```

Si l'on veut affecter le caractère `'a'` au premier élément de `CHAINE`, `'b'` au deuxième élément de `CHAINE` sans modifier les autres caractères de la chaîne, on peut écrire :

```
CHAINE(1..2) := "ab";
```

qui spécifie que le premier caractère de `CHAINE` est `'a'`, le deuxième est `'b'`. Il faut bien noter que les autres éléments de la chaîne de caractères ne sont pas modifiés ; en particulier, ils ne sont pas mis à blanc.

Concaténation

On peut encore concaténer[1] des chaînes de caractères avec l'opérateur `&` :

```
CHAINE := "abc" & "d" & efg" & 'h' & "ij";
```

qui permet (comme on le voit dans l'exemple) de mixer des chaînes de caractères et des caractères simples (`'h'`). On peut également concaténer des variables `STRING` comme suit :

```
-- quelques déclarations
CH1 : STRING (1..10);
CH2 : STRING (25..75);
CH3 : STRING (3..6);
CHAINE : STRING (10..30);

-- instructions
CHAINE := CH1 (2..7) & "jkiop" & CH3 (4..6) & CH2 (70..75) & 'p' & CH2 (35..45);
```

Comparaisons

On peut comparer des chaînes de caractères à l'aide des opérateurs habituels `=`, `/=`, `<`, `<=`, `>`, `>=`.

Deux chaînes sont égales si et seulement si elles contiennent le même nombre de caractères et que ceux-ci sont rigoureusement les mêmes, dans le même ordre, dans les deux chaînes.

Pour déterminer si une chaîne est inférieure à une autre, les caractères des deux chaînes sont comparés un par un, à partir du premier. Dès qu'en une position les caractères respectifs des deux chaînes diffèrent, l'ordre des deux chaînes est déterminé en fonction de l'ordre sur ces deux caractères. Ainsi, `"abc" < "adbe"` car pour les deux premiers caractères qui diffèrent, on a `'b' < 'd'`. Si l'une des deux chaînes est plus courte et débutte la seconde, alors la plus courte est la plus petite. Ainsi, `"abcd" > "abc"`.

On peut exprimer les chaînes sous différentes formes dans un test. Par exemple :

- `ch = "oie"`, en supposant que `ch` est une chaîne de caractères ;
- `'S' & "ab" = "Sab"` ;
- `(1 => 'A') = "A"`.

Conversion

On peut convertir un nombre en `STRING` à l'aide de l'attribut `IMAGE` des entiers. Ainsi, on pourra écrire :

[1] mettre bout à bout

```
procedure Essai is
   N : INTEGER := 43;
   CHAINE : STRING (1..10) := "N =       ";

begin
   CHAINE (5..6) := INTEGER'IMAGE (N);
   PUT_LINE (CHAINE);
end Essai;
```

qui affiche `N = 43` à l'écran.

Lecture d'une chaîne de caractères au clavier

Comme pour tout objet d'un type préfini, on peut utiliser la procédure `get` pour lire la valeur d'une chaîne de caractères au clavier. Si le paramètre de `get` est une `STRING` de 10 caractères, il faut impérativement saisir 10 caractères. Souvent, on souhaite lire une chaîne de caractères dont on ne connaît pas *a priori* la longueur. Pour cela, il faut utiliser la procédure `get_line` qui prend deux paramètres :

1. le premier est une variable de type `STRING` ;
2. le deuxième est une variable de type entier.

Lors de l'appel de cette procédure, une chaîne de caractères est saisie et stockée dans le premier paramètre ; cette chaîne est constituée des caractères frappés jusqu'à ce qu'un retour-chariot le soit, ce qui termine la chaîne. À la suite de l'appel de cette procédure, le premier paramètre contien donc la chaîne de caractères saisie et le second contient le nombre de caractères de la chaîne.

Par exemple, si le premier paramètre est une `string` de 10 caractères et l'utilisateur tape `hello` suivi d'un retour-chariot, au retour, les cinq premiers caractères de cette chaîne seront `hello` et le second paramètre prend la valeur 5. Attention, les éléments d'indice 6 à 10 de la chaîne sont inchangés : en particulier, ils ne sont pas mis automatiquement à blanc. Deuxième cas de figure : si l'utilisateur tape plus de 10 caractères avant de taper un retour-chariot, seuls les 10 premiers caractères frappés sont stockés dans la `string`.

6.2.2 Passage de paramètre de type `STRING`

Les `STRING` étant de simples tableaux mono-dimensionnels, on se reportera à la section 5.3 pour tout ce qui concerne le passage de variables de ce type en paramètre.

6.3 Types énumérés

Dans un type énuméré, on donne la liste des valeurs admissibles en extension ; c'est une liste de symboles ; ces symboles sont ordonnés.

6.3.1 Définition et déclaration

On déclare un type énuméré en listant les valeurs que peuvent prendre les données de ce type. Ainsi, le type `Type_Jour` peut être déclaré comme suit :

```
type Type_Jour is (LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE);
```

Dès lors, une variable peut être déclarée de type `Type_Jour` :

```
aujourdHui : Type_Jour;
```

et on peut lui affecter une valeur comme suit :

```
aujourdHui := JEUDI;
```

Dans l'énumération des valeurs possibles pour un type énuméré, une valeur numérique est associée à chacun des symboles. Les valeurs possibles (de `LUNDI` à `DIMANCHE` pour `Type_Jour`) sont numérotées dans l'ordre, à partir de 0 ; dans le cas de `Type_jour` :

- 0 est associé à `LUNDI` ;
- 1 est associé à `MARDI` ;
- 2 est associé à `MERCREDI` ;
- 3 est associé à `JEUDI` ;
- 4 est associé à `VENDREDI` ;
- 5 est associé à `SAMEDI` ;
- 6 est associé à `DIMANCHE` ;

La valeur numérique associée à un symbole peut être obtenue *via* un attribut (voir ci-dessous).

Comme pour tout type, on peut utiliser les opérateurs :

- `:=` pour l'affectation ;
- `=`, `/=` pour tester l'égalité ou l'inégalité ;
- `<`, `<=`, `>` et `>=` pour tester l'ordre de deux valeurs, l'ordre reposant sur la valuer numérique associée par l'attribut `POS` (voir ci-dessous).

6.3.2 Les attributs

- `Type_Jour'FIRST` fournit `LUNDI` ;
- `Type_Jour'LAST` fournit `DIMANCHE` ;
- `Type_Jour'SUCC(MARDI)` fournit `MERCREDI` ;
- `Type_Jour'PRED(JEUDI)` fournit `MERCREDI` ;
- `Type_Jour'POS(VENDREDI)` fournit `4` ;
- `Type_Jour'VAL(6)` fournit `DIMANCHE` ;
- `Type_Jour'VALUE("JEUDI")` fournit `JEUDI` (sans les ") ;
- `Type_Jour'IMAGE(LUNDI)` fournit `"LUNDI"` (la chaîne de caractères).

6.3.3 Type énuméré et indices

On peut utiliser une variable d'un type énuméré comme indice d'une boucle pour. Par exemple, en reprenant le `Type_Jour` vu plus haut, on peut écrire :

```
for jour in MARDI..VENDREDI loop
  ...
end loop;
```

On peut également utiliser un type énuméré pour indicer les éléments d'un tableau. Par exemple, on peut déclarer un tableau :

```
T : array (Type_Jour) of natural;
```

puis l'utiliser dans `T (MARDI)` par exemple.

6.4 Le type `BOOLEAN`

Le type `BOOLEAN` correspond aux objets booléens en mathématiques. Aussi, une valeur `BOOLEAN` peut prendre deux valeurs : `TRUE` et `FALSE`. En fait, le type `BOOLEAN` est un type énuméré définit par : `type BOOLEAN is (FALSE, TRUE);`|. Aussi, le type `BOOLEAN` possède les attributs de tous les types énumérés.

Les conditions que l'on a rencontrées dans les instructions de test et les boucles produisent une valeur de type `BOOLEAN`.

Sur les données de type `BOOLEAN`, on peut utiliser les opérateurs :

- `:=` pour l'effectation ;
- `=`, `/=` pour tester l'égalité et l'inégalité ;
- `<`, `<=`, `>`, `>=` pour tester l'ordre : on a FAUX $<$ VRAI ;

- `not` pour l'inversion ;
- `and` pour le ET-logique ;
- `or` pour le OU-logique
- `xor` pour le OU-EXCLUSIF-logique.

6.4.1 Conditions composées

Jusqu'alors, nous n'avons rencontré que des tests aux conditions simples. Des conditions peuvent être exprimées comme conjonction, disjonction et négation de prédicats à l'aide des opérateurs `and`, `or` et `not`.

Par exemple, on pourra écrire :

```
if (N > 5 and N <= 10) or N = 0 then
  ...
end if;
```

pour tester qu'une variable `N` a une valeur dans l'ensemble d'entiers $]5, 10] \cup \{0\}$.

Des cascades de tests peuvent également être écrites plus simplement. Ainsi, `A`, `B` et `C` étant les coefficients d'un polynome du deuxième degré, on peut écrire :

```
if A=0.0 then
  -- équation linéaire
else
  if B**2-4.0*A*C>=0.0 then
    -- racines réelles
  else
    -- racines complexes
  end if;
end if;
```

On peut écrire cela sous la forme moins lourde :

```
if A=0.0 then
  -- équation linéaire
elsif B**2-4.0*A*C>=0.0 then
    -- racines réelles
else
    -- racines complexes
end if;
```

On peut avoir autant de `elsif` en séquence qu'il est nécessaire.

On rencontre également souvent des constructions ayant la forme suivante :

```
if condition1 then
  if condition2 then
    -- traitement
  end if;
end if;
```

On peut écrire cela sous la forme moins lourde :

```
if condition1 and then condition2 then
  -- traitement
end if;
```

La `condition2` n'est évaluée que si la `condition1` est vérifiée. En effet, si la `condition1` n'est pas vérifiée, même si la `condition2` l'était, le `traitement` ne serait pas exécuté.

6.4.2 Les tableaux de `BOOLEAN`

Certaines opérations particulières sont possibles sur les tableaux de `BOOLEAN`.

On peut écrire des agrégats de `BOOLEAN` sous la forme suivante :

```
A, B, C : array (1..4) of BOOLEAN;

A := (TRUE, FALSE, FALSE, TRUE);
B := (FALSE, others => TRUE);
```

sur lesquels des opérations booléennes peuvent être effectuées :

```
C := A and B; -- C vaut ensuite (FALSE, FALSE, FALSE, TRUE)
C := A or B;  -- C vaut ensuite (TRUE, TRUE, TRUE, TRUE)
C := A xor B; -- C vaut ensuite (TRUE, TRUE, TRUE, FALSE)
C := not A;   -- C vaut ensuite (FALSE, TRUE, TRUE, FALSE)
```

6.5 Exercices

1. écrire l'arbre programmatique d'une fonction qui renvoie le miroir d'une chaîne de caractères. Par exemple, si on lui fournit la chaîne `abcdef`, la fonction renvoie `fedcba` ;
2. écrire un arbre programmatique d'une fonction qui saisit un caractère frappé au clavier et attend que ce soit `'o'` ou `'n'` dans une boucle Tant-que. La fonction renvoie ensuite le caractère frappé ;
3. écrire un arbre programmatique qui prend en entrée une chaîne de caractères et détermine si c'est un palindrome, c'est-à-dire une chaîne qui peut se lire à l'identique dans les deux sens. Traduire ensuite cet arbre en fonction renvoyant un `BOOLEAN` ;
4. écrire un arbre programmatique de l'action qui recherche une lettre dans une chaîne de caractères et renvoie un `BOOLEAN` si la lettre est présente. Traduire l'arbre en fonction Ada ;
5. écrire un arbre programmatique de l'action qui compte le nombre de voyelles présentes dans une chaîne de caractères et renvoie ce nombre (un `NATURAL`). Traduire l'arbre en fonction Ada ;
6. écrire un arbre programmatique de l'action qui compte le nombre de voyelles différentes qui sont présentes dans une chaîne de caractères et renvoie ce nombre (un `NATURAL`). Traduire l'arbre en fonction Ada ;
7. écrire un arbre programmatique de l'action qui compte le nombre de lettres différentes dans une chaîne de caractères et renvoie ce nombre (un `NATURAL`). Traduire l'arbre en fonction Ada ;
8. écrire un arbre programmatique de l'action qui retire toutes les occurences d'un caractère donné dans une chaîne de caractères. Traduire l'arbre en fonction Ada ;
9. écrire un arbre programmatique de l'action qui teste la présence d'une sous-chaîne dans une chaîne de caractères et renvoie un `BOOLEAN`. Traduire l'arbre en fonction Ada ;
10. en utilisant un tableau de `BOOLEAN`, écrire l'arbre programmatique de la procédure qui réalise le crible d'Ératosthène ;
11. on se donne les types

```
type Tricolore is (bleu, rouge, vert);
type Tabtricolore is array (POSITIVE range 1..N) of Tricolore;
```

Ce tableau est initialement rempli de manière aléatoire. Écrire l'arbre programmatique d'une procédure qui trie le tableau de telle manière que tous les éléments de valeur `bleu` se trouvent aux indices 1 à `m`, les éléments de valeur `rouge` se trouvent aux indices `m+1` à `p`, les éléments de valeur `vert` se trouvent aux indices `p+1` à `N`.

Pour effectuer le tri des couleurs, les seules opérations qui soient autorisées sont : le test de la couleur d'un élément du tableau et l'échange du contenu de deux éléments du tableau.

Chapitre 7

Les enregistrements

7.1 Définition

Un enregistrement est une variable dont la valeur est composée de données de plusieurs types. Ainsi, on peut dire qu'une Personne est caractérisée par un nom qui est de type `STRING`, une année, un mois et un jour de naissance qui sont des entiers positifs appartenant à un certain intervalle de valeurs. Les différents composants d'un enregistrement sont ses « champs ». On peut alors définir un type `PERSONNE` et l'utiliser comme suit :

```
type TYPE_MOIS is (Jan, Fev, Mar, Avr, .... Dec);

type PERSONNE is record
  nom : STRING (1..20);
  anneeNaissance : POSITIVE range 1900..2100;
  moisNaissance : TYPE_MOIS;
  jourNaissance : POSITIVE 1..31;
end record;

TOTO : PERSONNE;
TITI : PERSONNE;

X : constant PERSONNE := (nom => ("XYZ", others => ' '),
                          anneeNaissance => 1900,
                          moisNaissance => Jan,
                          jourNaissance => 1);

begin
```

```
   TOTO. nom := (toto, others => ' ');
   TOTO. anneeNaissance := 1923;
   TOTO. moisNaissance := Jun;
   TOTO. jourNaissance := 17;

   -- agrégats
   TITI := (nom => ("titi", others => ' '),
            anneeNaissance => 1954,
            moisNaissance => Avr,
            jourNaissance => 17);
```

Ces instructions indiquent :

- la déclaration d'un type dénommé `PERSONNE` composé de 4 champs s'appelant `nom`, `anneeNaissance`, `moisNaissance`, `jourNaissance` ;
- la déclaration de deux variables de type `PERSONNE` dénommées `TOTO` et `TITI` ;
- la déclaration d'une constante portant le nom `X` et initialisée avec le nom `XYZ` et née le 1erjanvier 1900 ;
- les 4 instructions qui suivent le mot-clé `begin` affectent les 4 champs de la variable `TOTO`. On note l'utlisation du point . pour accéder à un champ d'un enregistrement ;
- après le commentaire, on indique également comment on spécifie un agrégat d'enregistrement en nommant les champs et en leur associant une valeur. Tous les champs ne sont pas nécessairement spécifiés ; dans ce cas, la valeur des champs non indiqués n'est pas modifiée.

Un synonyme de « enregistrement » est « article » ou « structure ».

Un champ d'enregistrement peut être de n'importe quel type ; en particulier, ce peut être un enregistrement également. Par exemple, on peut définir un type `Type_Date` :

```
type Type_Date is record
  annee : POSITIVE range 1900..2100;
  mois : TYPE_MOIS;
  jour : POSITIVE 1..31;
end record;
```

et l'utiliser ensuite pour définir le type `PERSONNE` comme suit :

```
type PERSONNE is record
  nom : STRING (1..20);
  dateNaissance : Type_Date;
end record;
```

```
TOTO : PERSONNE;
TITI : PERSOONNE;

begin
  TOTO. nom := ("toto", others => ' ');
  TOTO. dateNaissance. annee := 1923;
  TOTO. dateNaissance. mois := Jun;
  TOTO. dateNaissance. jour := 17;

  -- agrégats
  TITI := (nom => ("titi", others => ' '),
           dateNaissance => (annee => 1954,
                             mois => Avr,
                             jour => 17));
```

7.2 Quelques caractéristiques des enregistrements

Les enregistrements peuvent être utilisés comme n'importe quelle donnée. Ainsi, on peut les passer en paramètre et une fonction peut retourner un enregistrement. On peut affecter un enregistrement à une variable du moment que les types sont identiques.

On peut tester l'égalité et l'inégalité avec les opérateurs habituels `=` et `/=`. Deux enregistrements dont égaux s'ils sont de même type et que tous leurs champs sont respectivement égaux.

On peut définir des tableaux dont les éléments sont des enregistrements.

7.3 Exercices

1. définir un type `Type_Horaire` comprenant des champs `heure`, `minute` et `seconde`. Écrire ensuite des fonctions `SommeHoraire` et `DifferenceHoraire` qui font respectivement la somme et la différence de deux horaires passés en paramètre et renvoient le résultat ;
2. définir le type `Complexe` représentant les nombres complexes. Écrire ensuite les fonctions `SommeComplexe`, `DifferenceComplexe`, `ProduitComplexe`, `InverseComplexe` et `DivisionComplexe` ;
3. définir un type `Point` qui représente un point dans un espace bidimensionnel. Définir ensuite un type `Triangle` comme un triplet de points. Écrire les arbres programmatiques et les fonctions correspondant aux actions suivantes :
   (a) `LongueurCotes`, une procédure qui renvoie 3 valeurs, la longueur des 3 côtés du triangle ;

(b) `Angles`, une procédure qui renvoie 3 valeurs, la valeur des 3 angles du triangle. Rappel : si a, b et c sont la longueur des côtés du triangle et A, B et C ses angles (en reprenant la notation classique), on a par exemple la relation : $c^2 = a^2 + b^2 - 2ab\ \cos C$;

(c) `Acutangle` qui renvoie une valeur `BOOLEAN` qui indique si le triangle est acutangle (c'est-à-dire que les 3 angles sont aigus) ;

(d) `Perimetre` qui renvoie le périmètre d'un `Triangle` passé en paramètre ;

(e) `Surface` qui renvoie la surface d'un `Triangle` passé en paramètre. Rappel : la surface d'un triangle est $\mathcal{S} = \sqrt{s(s-a)(s-b)(s-c)}$ où $s = \frac{a+b+c}{2}$;

(f) `Dans` qui indique si un `Point` est localisé à l'intérieur d'un `Triangle`. Pour cela, la méthode est la suivante : soient A, B et C les sommets du triangle et X un point ; $X \in$ triangle si les composantes z des trois produits vectoriels $\vec{AX} \wedge \vec{AC}$, $\vec{BX} \wedge \vec{BC}$ et $\vec{CX} \wedge \vec{CA}$ sont tous trois de même signe ;

(g) `CentreDeGravite` qui renvoie les coordonnées du centre de gravité du `Triangle` passé en paramètre. Rappelons que le centre de gravité dun triangle se trouve à l'intersection de ses médianes et qu'une médiane passe par un sommet du triangle et le milieu du côté opposé.

7.4 La surcharge de procédures et d'opérateurs

Lorsque l'on définit un type enregistrement, il est naturel de devoir ensuite définir des procédures qui effectuent les opérations de base : saisir une valeur au clavier, afficher la valeur d'une structure, ... Pour cela, Ada permet de re-définir des procédures déjà existantes telles `GET` et `PUT`.

Supposons que nous définissions le type `Type_Horaire` comme suit :

```
type Type_Horaire is record
  heure : NATURAL range 0..23;
  minute : NATURAL range 0..59;
  seconde : NATURAL range 0..59;
end record;
```

On peut alors définir la procédure `GET` qui saisit une donnée de type `Type_Horaire` :

```
procedure GET (h : out Type_Horaire) is
begin
  GET (h. heure);
  GET (h. minute);
  GET (h. seconde);
end GET;
```

En Ada, on peut définir plusieurs fois une procédure ou une fonction portant le même nom à partir du moment où les paramètres (leur type ou leur nombre) sont différents, afin que le compilateur puisse déterminer la définition de la procédure qu'il faut utiliser.

Ce point permet d'éviter le problème classique dans les langages procéduraux (comme Fortan, C ou Pascal) d'avoir, pour une même fonctionnalité (par exemple saisir la valeur d'une donnée au clavier) des procédures ayant des noms distincts; ainsi, en Ada, on prend la règle que la fonction `GET` effectue la saisie d'une donnée au clavier, quel que soit le type de la donnée; à la charge au programmeur de redéfinir correctement la fonction pour chaque type de données.

Dans le même esprit, on peut également redéfinir les opérateurs (tels `+`, `-`, `*`, `/`, `=` et `<`) et les utiliser exactement comme s'il s'agissait de nombres.

à la suite de l'exemple précédent, on peut ainsi définir l'opérateur d'addition sur le type `Type_Horaire` :

```
function "+" (h1, h2 : Type_Horaire) return Type_Horaire is
  res : Type_Horaire;
begin
  res. heure := ...;
  res. minute := ...;
  res. seconde := ...;
end "+";
```

On peut ensuite utiliser l'opérateur `+` tout à fait normalement. L'opérande à gauche de l'opérateur correspond au paramètre formel `h1`, l'opérande à droite correspond au paramètre formel `h2`. Si on suppose que `PUT` a été défini correctement pour le type `Type_Horaire`, on peut écrire :

```
procedure EssaiHoraire is
  H1, H2, H3 : Type_Horaire;
begin
  GET (H1);
  GET (H2);
  H3 := H1 + H2;
  PUT (H3);
end EssaiHoraire;
```

La priorité sur les opérateurs est la même que sur les nombres (voir section 2.1.2).

On peut surcharger les opérateurs suivants : `+`, `-`, `*`, `/`, `=`, `<`, `<=`, `>`, `>=`, `mod`, `rem`, `**`, `abs`, `not`, `and`, `or`, `xor`. Notons que `/=` ne peut pas être surchargé : sa définition se déduit de celle de `=`.

Les opérateurs `=`, `<`, `<=`, `>`, `>=` doivent retourner une valeur `BOOLEAN`. On peut ensuite les utiliser dans leur contexte habituel.

7.4.1 Exercices

1. pour le type `Type_Horaire`, définir la procédure `PUT`, les opérateurs `+`, `-`, `=`, `<`, `<=`, `>`, `>=` entre deux données de ce type ;
2. pour le type `Complexe` vu plus haut, définir les opérateurs `+`, `-`, `=`, `<`, `<=`, `>`, `>=`, `**`, `abs` comme donnant le module du complexe qui lui est passé en argument, les procédures `GET` et `PUT`. Utiliser ensuite tout cela pour écrire un programme qui résoud les équations du 2^{e}degré dans l'ensemble des complexes ;
3. définir le type `Fraction` représentant les nombres rationnels sous leur forme a/b. Écrire ensuite les arbres programmatiques et les fonctions pour les opérateurs usuels, les procédures `GET` et `PUT` ;
4. définir le type `GrandEntier` représentant des entiers écrits sur `MAXCHIFFRE` dans un tableau de `MAXCHIFFRE` chiffres. (Un chiffre est de type `type Chiffre is NATURAL range 0..9`.) Par exemple, l'entier 6528 sera représenté par un tableau dont l'élément d'indice 1 vaut 8, celui d'indice 2 vaut 2, celui d'indice 3 vaut 5, celui d'indice 4 vaut 6 et les autres éléments sont nuls. Écrire ensuite les arbres programmatiques et les fonctions pour les opérateurs usuels (arithmétiques et de comparaison), la fonction `GET` et la procédure `PUT`. La valeur des nombres entiers (de type `INTEGER`) ne peut pas dépasser `INTEGER'LAST` ; en définissant ce type et les opérations associées, on pourra traiter des nombres entiers bien plus grands, à l'unité près (si `MAXCHIFFRE` vaut 1000, on pourra traiter des nombres jusque 10^{1000}) ;
5. on souhaite définir le type `Ensemble`. Pour cela, on utilisera un champ de type énéméré qui liste les valeurs que peuvent prendre les éléments de l'ensemble et un champ de type tableau de booléen indiquant, pour chacune des valeurs du type énuméré, si cette valeur est contenue dans l'ensemble. On définira ensuite les opérations usuelles sur les ensemble en surchargeant les opérateurs arithmétiques : `E1 + E2` est l'union des ensembles `E1` et `E2`, `E1 * E2` est l'intersection, `E1 - E2` est l'ensemble constitué des éléments de `E1` duquel on retire ceux appartenant à `E2` ; on définira également les opérateurs de comparaison et les procédures `put` et `get`.

Index

Table des matières