# CNAM ALGO-A0 1994-95

## La table des matières correspondant aux transparents

# Algorithmique

# et

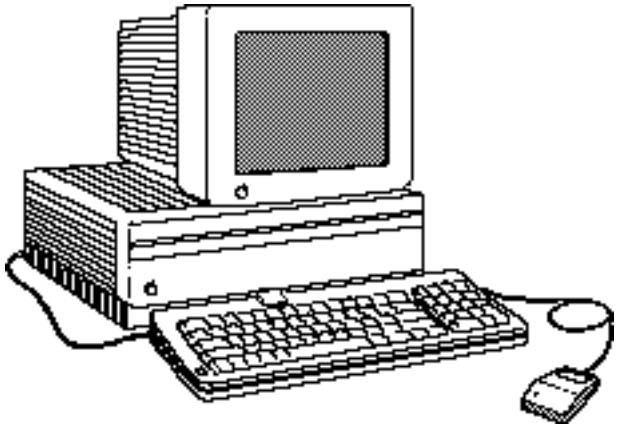
# Programmation

## C . N . A . M .

1994- 95

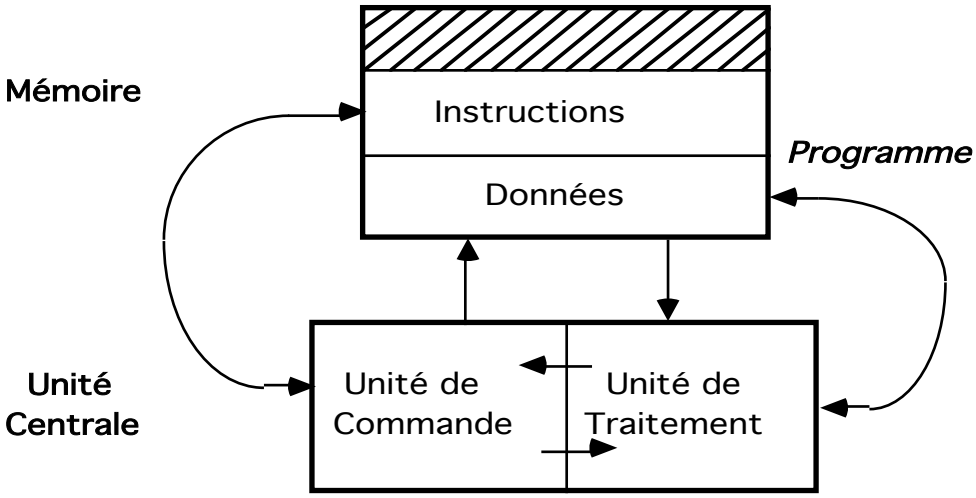
A.S. SAIDI
Ecole Centrale de Lyon
Dépt. M.I.S.

# 1- Introduction : quelques notions



Un ordinateur :

- Un objet de décoration ?
- Un moyen de calcul ?
  - **Mémoire** et l'**Unité Centrale**




***Ordinateur (modèle von Neumann)***

**Comment ça fonctionne ?**

- Matériel / Logiciel
- Est-ce intelligent ?
- Que peut il faire ? (vitesse, répétition, volume, ...)

## Nous et les ordinateurs

- Un outil de travail : faire tourner des applications
  programmer un calculateur
- Un moyen de **traitement d'information**.

**Information** : critère de choix permettant de restreindre la taille de l'ensemble résultat.

**Traitement de l'information** : réduction de la quantité d'information.


Les *Données* permettent de réduire, à l'aide du *programme* P, l'espace des *résultats* à celui qui nous intéresse.

## Programmer un ordinateur

Communiquer avec dans un **langage** qu'il comprend

## Etude des langages de programmation

- Traitement de langues
  - Naturelle (syntaxe, sémantique, déduction)
  - Artificielle

**Problème à résoudre**
Spécification du problème
**Algorithme de la solution**
**Programmation**
Traduction en binaire
(Compilation/ Interprétation)
Exécution
**Réponse au Problème?**

- Algorithmes
- Concepts et paradigmes de programmation
- Langages : ADA, CAML (PASCAL, C)

# 2- L'algorithmie

## 2.1- Notions informelles

- Un algorithme fournit la description appropriée d'une méthode de résolution d'un problème.

- Un algorithme décrit une séquence de phrases portant sur des objets de calcul (valeurs).

Les phrases sont choisies dans un répertoire fini et bien défini d'actions élémentaires identifiées que l'on suppose réalisables a priori.

Ces phrases sont habituellement indépendantes des langages de programmation.

- Un algorithme est une suite générale et totalement définie d'opérations et de leurs enchaînements ayant pour but le traitement (la transformation) d'informations :

- Un algorithme est donc une *abstraction* d'un schéma de calcul.

## 2.2- Exemple

Problème: *madame dupont épluche des pommes de terre*
Algorithme :

-chercher le panier à la cave
-sortir la poêle du placard
-éplucher les pommes de terre
-ranger le panier

D'autres schémas :

Patrons pour le tricot,
Recettes de cuisine,
Indication du chemin (1er à gauche, 2nd à droite, tourner..)

## 2.3- Identification d'un algorithme

- par son nom
- par les données qu'il manipule (paramètres)

Pour les problèmes non triviaux, les algorithmes sont décomposés en sous-algorithmes (abstraction) :

Un sous-algorithme décrit la solution d'une partie du problème.

## 2.4- Structures de données

L'association d'un ou plusieurs noms et d'un **ensemble d'informations** auxquelles ce ou ces noms permettent d'accéder.

| *Algorithme + structure de données = programme* |
|---|

## 2.5- Décomposition des structures de données

• Décomposition fonctionnelle (vue et définition externe)

**Retrait : Compte x entier --> Compte**

• Décomposition logiques (plus de détails)

- Description de le donnée

***Compte***

**I- titulaire : (nom x prénom x adr)**
**I- valeur : entier**
**I- date création : (jour x mois x an)**

- Description de l'action (algorithme)

***Retrait***

• Décomposition physiques
- Décision d'implantation **physique** en terme de tableaux, listes...

## 2.6- Programmes

Pour être exécutés sur un ordinateurs, les algorithmes sont traduits (transformés) en programmes.

Un programme est généralement la description d'un algorithme dans un langage accepté par une machine sur laquelle il doit être exécuté.

Ces programmes sont des schémas de comportement exprimés à l'aide d'un répertoire fini et bien compris d'actions élémentaires.

Ces programmes seront traduits à leur tour en actions plus simples et déterminées exécutables par une machine.

On s'intéresse aux processus séquentiels  bloc de commandes

**Les langages de programmation**

Plusieurs classes (impératifs, fonctionnel, relationnel, ....objet)

*Nous étudions les langages impératif (ADA) et fonctionnel (CAML)*

**Les programmes**

f

- Un programme réalise une fonction **f** : Entrées ====> Sorties ;

- Le but principal d'un programme est de modifier l'état de la mémoire (par l'affectation dans les langages impératifs : **X := X+1**)

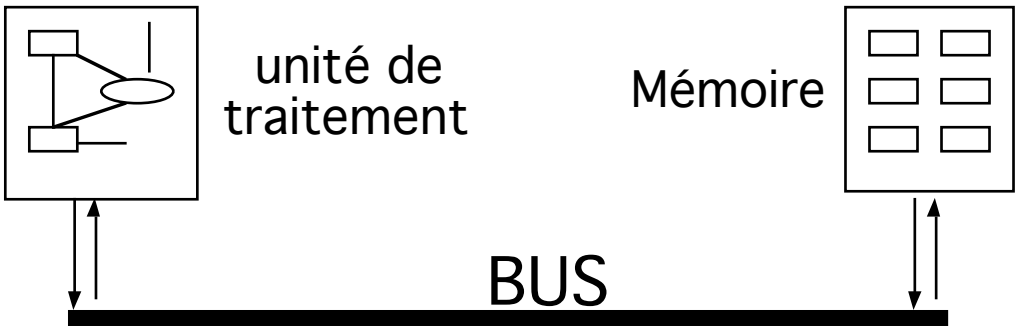
- L'affectation modifie la valeur d'une (ou une suite de) case mémoire par une autre valeur simple (ou composée).

- Le résultat d'un programme est l'état (modifié) de la mémoire.

# 3- Etude d'un langage de bas niveau

Comment les programmes sont exécutés sur une machine.

**La machine** (très simple) :




**Notions :**

- Instruction : un ordre exécutable par l'unité de traitement.
- Les cases mémoires sont numérotées (de 0 .... )
- Elles sont accessibles par leur rang : leur **adresse**
- Elles peuvent être accédées par un nom symbolique tel que X, Z1: Ce nom symbolique est l'**identificateur de la variable**
- La valeur d'une variable est le contenu de la case mémoire associée

**La boucle exécutée par l'unité de traitement :**

Répéter à l'infini
Extraire l'instruction à l'adresse indiquée par le compteur du programme de la zone d'instructions (en mémoire)
Décoder cette instruction
Exécuter cette instruction
Si cette instruction n'était pas un branchement
Alors incrémenter le compteur de programme Fin si
Fin Répéter

**Dans la machine considérée :**

- Le format général d'une instruction :
  **[étiquette] Opération opérande**
- Adresse, étiquette et variable intermédiaire ($t_i$)
- Accumulateur :
  Une case particulière impliquée dans toutes les opérations

Opérandes :

| | |
|---|---|
| Un nom | x, a, b, $t_3$ |
| Une valeur immédiate | #4, #-2 |
| Une adresse | @x, 1432 |
| Une indirection | (x) |

Opérations (les instructions de base) :

| | | |
|---|---|---|
| **load** | **x** | *(x) -> acc* |
| **store** | **x** | *(acc) -> x* |
| **add** | **x** | *(x)+(acc) -> acc* |
| **mult** | **x** | *(x)*(acc) -> acc* |
| **comp** | **x** | *(acc) -(x) -> acc* |

*(acc) > (x) => (acc) > 0*
*(acc) = (x) => (acc) = 0*
*(acc) < (x) => (acc) < 0*

| | |
|---|---|
| **bra etiq** | *continuer l'exécution à l'adresse etiq* |
| **brp etiq** | *si (acc) > 0 alors bra etiq* |
| **brz etiq** | *si (acc) = 0 alors bra etiq* |
| **brn etiq** | *si (acc) < 0 alors bra etiq* |

## 3.1- Exemples

**(I) A := B + 5;**

| | | | |
|---|---|---|---|
| load | B | | (acc) = la valeur de B |
| add | | #5 | (acc) = (acc) + 5 |
| store | | A | (A) = (acc) |

## 3.2- Réalisation d'opérations (structures) de niveau plus élevé

Expressions Arithmétiques
Conditionnel

## 3.3- Exercices : traduire les commandes suivantes

**(II)** **A := A-B**

**(III)** **A := (A+1) * (B+2)**

**(IV)** **Si A > 5 Alors A := A-1 Fin si**

**(V)** **Si A > B alors A := A+B sinon A :=A-B Fin si**

**(VI)** Déduire le format général d'un schéma

***Si Condition alors I1 sinon I2 Fin si***

③ Traduction de proche en proche des commandes et des expressions de haut niveau en un langage plus simple.

③ Ces traductions suivent des schémas prédéfinis dans les compilateurs.

# => Ce que fait un compilateur <=

## 3.4- Solutions

**(II)** **A := A-B;**

| | | | Autre solution | |
|---|---|---|---|---|
| load | B | *(acc) = la valeur de B* | Load | A |
| mult | #-1 | *(acc) = -B* | comp | B |
| add | A | *(acc) = A-B* | store | A |
| store | A | *(A) = A-B* | | |

**(III)** **A := (A+1) * (B+2) ;**

| | | |
|---|---|---|
| load | A | *(acc) = la valeur de A* |
| add | #1 | *(acc) = (acc) +1* |
| store | $t_1$ | *($t_1$) = A+1* |
| load | B | *(acc) = la valeur de B* |
| add | #2 | *(acc) = (acc) +2* |
| mult | $t_1$ | *(acc) = ...* |
| store | A | *(A) = (acc)* |

**(IV)** **Si A > 5 Alors**
**A := A-1;**
**Fin si;**

| | | | |
|---|---|---|---|
| | load | A | *(acc) = la valeur de A* |
| | comp | #5 | *(acc) = (acc) - 5* |
| | brp | suite | *aller à "suite" si (acc) > 0* |
| | bra | fin | *aller à "fin"* |
| suite | load | A | *(acc) = la valeur de A* |
| | add | #-1 | *(acc) = (acc) - 1* |
| | store | A | *(A) = (acc)* |
| fin | ... | | |

**(V)** **Si A > B alors**
**A := A+B**
**sinon**
**A :=A-B;**
**Fin si;**

```
        load A
        comp    B
        brp     alors       si (acc) > 0 alors aller à "alors"
sinon   load B              (acc) = la valeur de B
        mult    #-1         (acc) = -B
        add     A           (acc) = A-B
        store   A           (A) = A-B
        bra     fin         continuer à l'adresse "fin"
alors   load A
        add     B
        store   A
suite   ......
```

Lors de la compilation, ces traductions suivent des schémas prédéfinis de traduction.

## 3.5- Schéma général conditionnel

**si Condition alors Action1**
**sinon Action2**
**fin si**

```
        évaluer Condition
        brn  sinon
        Action1
        bra  fin si
sinon : Action2
Fin si: .....
```

```
        évaluer Condition
        brp  alors
        Action2
        bra  fin si
alors : Action1
Fin si: .....
```

## 3.6- Schémas plus complexes

• Un langage d'aussi bas niveau est difficile à manier.

• Remonter le niveau expressif à l'aide des définitions :
- Le schéma conditionnel avec les comparateurs (=, ≤,..)
- les opérateurs arithmétiques (+, -, ...)
- L'affectation
- Skip
- Goto <étiquette>

Toute commande peut être étiquetée par un label (étiquette).

## 3.7- Formalismes algorithmiques

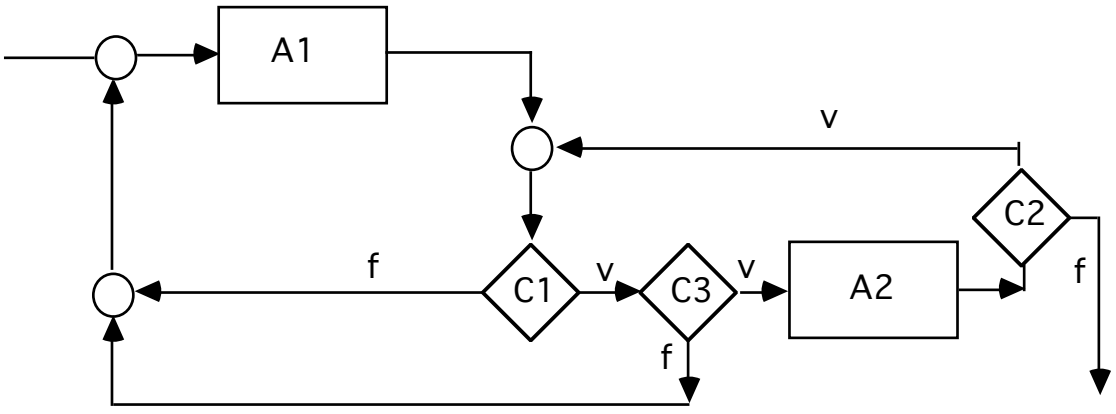
Langage de description d'algorithmes : graphique / textuel

### *3.7.1- Les organigrammes*

• Formalisme algorithmique primitive permettant la description des enchaînements

• Expression d'algorithme à l'aide de symboles graphiques

### *3.7.2- Exemple*



## *3.7.3- Formalisme textuel*

```
étiquette1:     A1
étiquette2 :    si C1 alors
                    si C3 alors
                        A2;
                        si C2 alors goto étiquette2
                        sinon goto fin
                        fin si
                    sinon goto étiquette1
                    fin si;
                sinon goto étiquette1
                fin si
fin :       .....
```

- Comparaison des formalismes
- Utilisation dans visual-XX (générateurs de codes), automates

## *3.7.4- Exemples*

• Le langage Fortran IV ne dispose pas de schéma **si-alors-sinon**.

**Si a=5 Alors a:=a+1 Sinon Si a<5 alors I2 Fin si; Fin si; I3;**

```
début :   si a>5 alors goto I3
          si a<5 goto I2
          a:=a+1
          goto I3
I2 :      .....
I3 :  .....
```

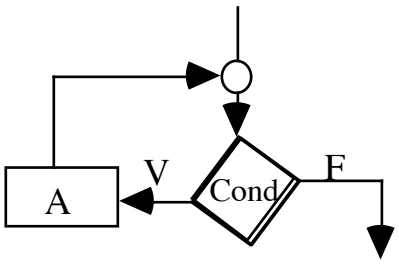
- Traduire **z=x * y** en terme d'addition (x et y positifs)

```
début :   z=0
mult :    si y ≤ 0 alors goto fin
          z := z + x
          y := y - 1
          goto mult
fin :     ...
```

## 3.8- Schémas itératifs

# (I)

Tant que Cond faire
  A
Fin Tant que

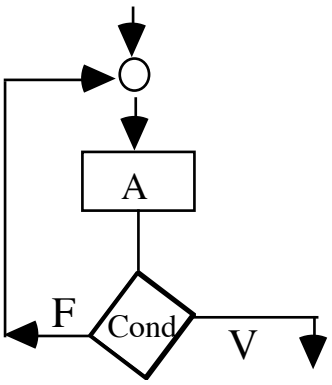



**Exemple :**

Tant qu'il y a quelque chose sur la bande faire
    établir la fiche d'impôts
    passer au contribuable suivant
Fin Tant que

# (II)

Répeter
  A
Jusqu'à Cond

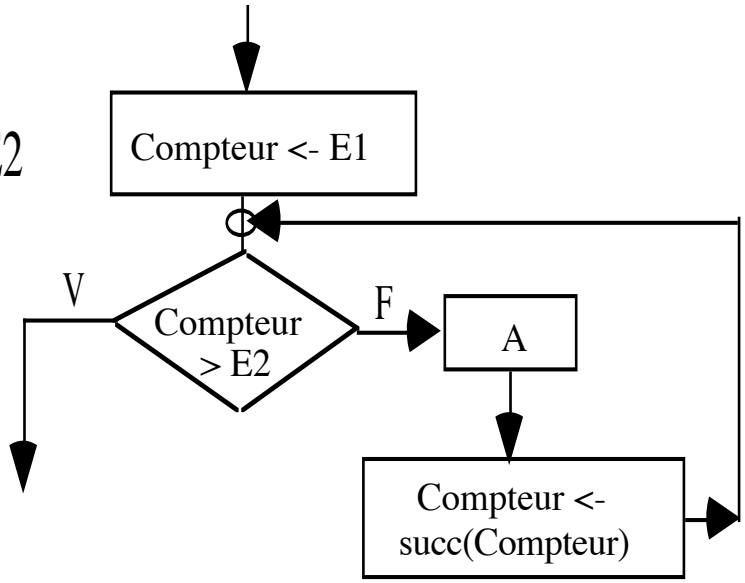



**Exemple :**

Répéter
    Prendre température
jusqu'à température > 50
Déclencher alarme

# (III)

Pour Compteur dans E1 .. E2
faire
A
Fin pour




**Exemple :**

```
Pour i dans 1 .. 10 faire
      écrire t(i)
fin Pour
```

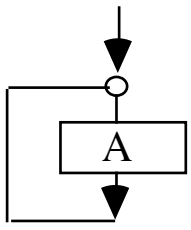
# (IV)

```
Pour Compteur bas E1 .. E2 faire
            C
fin Pour
```

Dans ces deux schémas de POUR:
- [E1..E2] est supposé être un intervalle ordonné;
- Les fonctions *successeur* et *prédécesseur* sont définies sur E1..E2.

# (V)

Répeter à l'infini
A
Fin Répéter



**Exemple :**

Répéter à l'infini
    vérifier la température
Fin répéter

### *3.8.1- Formes itératifs combinées*

(VI)

Pour Compteur dans (bas) intervalle (liste, ensemble)
        Jusqu'à Condition faire
            A
Fin pour

**Comportement du schéma:**

Visiter tous les éléments de l'intervalle (liste, ensemble)
Si l'un d'eux rend la Condition = vraie alors arrêter la boucle
Si tous les éléments sont visités alors arrêter la boucle

**Exemple : quête de 100 francs (non équitable !)**

```
S <-  0;
Pour X dans la liste-employés  Jusqu'à S >= 100 faire
            S <-  S + donne(X)
Fin pour
-- Ici, on a 100 francs (ou on ne les a pas !)
```

## 3.9- Exercice

En ne vous servant des commandes précédentes (§3.7), traduire les schémas TANT QUE, REPETER et POUR.

## 3.10- Solutions

(I) Tant que E faire
C
Fin TANT QUE

```
Tant que :   si non E alors goto Fin fin si
             C
             goto Tant que
Fin :        .....
```

(II) Répéter
C
jusqu'à E

```
Répéter :    C
             si E alors goto Fin fin si
                  goto Répéter
Fin :        .....
```

**Variante :**

```
Répéter :    C
             si non E alors  goto Répéter fin si
Suite :      ......
```

(III) Pour Compteur dans E1 .. E2 faire
C
fin Pour

```
             Compteur := E1
Pour :       si Compteur > E2 goto Fin finsi
             C
             Compteur := successeur(Compteur)
             goto Pour
Fin:      .....
```

(IV) Idem (III)

(V) Répéter à l'infini
C
fin Répéter

```
Répéter :   C
            goto Répéter
Suite:      .....
```

(VI) Pour Compteur dans (bas) E1..E2
Jusqu'à Condition faire
C
Fin pour

```
            Compteur := E1
Pour :      si Compteur > E2 goto Fin  fin si
            C
            Si Condition = vraie alors goto Fin fin si
            Compteur := successeur(Compteur)
            goto Pour
Fin:        .....
```

(VI') Pour X dans liste (ensemble) Jusqu'à Condition faire
C
Fin pour

```
Pour :      Si liste (ensemble) = vide alors goto Fin finsi
            X := premier (un) élément de liste (ensemble)
            liste (ensemble) := liste (ensemble) \ X
            C
            Si Condition = vraie goto Fin fin si
            goto Pour
Fin:        .....
```

**D'autres commandes utiles** (à effet de bord) : Lecture/Ecriture

```
Lire(x);        -- lecture de l'entier x
x := x+1 ;
Ecrire("la valeur de x incrémentée est = ", x);
```

## 3.11- Exemples d'utilisation

• Traduire z = x * y (x et y positifs ou nuls) :

On a : $x * y = 0$ si $y = 0$
$x * y = x + (y-1) * x$ si $y > 0$

Avec Tant que :

```
z := 0;
Tant que y > 0 faire
      z := z+x;
      y := y - 1;
Fin Tant que;
```

Avec Répéter :

```
z := 0;
si y > 0 alors
      répéter
            z := z+x;
            y := y-1;
      jusqu'à y ≤0
Fin si;
```

Avec Pour:

```
z := 0;
pour i dans 1..y faire
      z := z+x;
Fin pour;
```

```
z := 0;
pour i bas 1..y faire
      z := z+x;
Fin pour;
```

# 4- Place et rôle de l'algorithmie

- Méthode de développement des applications (logiciels)
- Maîtrise de la complexité du développement d'une application

**Algorithme dans le cycle de vie**

Problème à résoudre (cahier des charges)
⇓
spécification (formalisation du problème)
⇓
Conception (élaboration d'une solution)
⇓
Conception de l'architecture
⇓
Conception détaillée
⇓
*Algorithmes*
⇓
Réalisation (à l'aide d'un langage de programmation)
⇓
.....

## 4.1- Définition d'algorithme

selon l'Encyclopedia Universalis :

> *Spécification d'un schéma de calcul, sous forme d'une suite finie d'opérations élémentaires obéissant à un enchaînement déterminé.*

**Exemples de schémas de calculs:**

- Algorithme d'Euclide pour calculer le p.g.c.d. de deux nombres entiers;
- Algorithmes de tri pour ranger une suite de noms ;
- Algorithmes de recherche d'une chaîne de caractères dans un texte ;
- Algorithmes d'ordonnancement permettant de décrire la coordination entre différentes tâches, nécessaire pour mener à bien un projet;
- Algorithmes de calcul de toutes les conséquences de tous les coups possibles dans un jeu d'échec; .....

## 4.2- Propriétés des algorithmes

• Un algorithme décrit un traitement sur un certain nombre, fini, de *données* (éventuellement aucune).

• Un algorithme est la composition d'un ensemble fini d'*étapes*, chaque étape étant formée d'un nombre fini d'opérations dont chacune est :

- *définie* de façon rigoureuse et non ambiguë;

- *effective* c'est à dire, pouvant être effectivement réalisée par une machine (c'est le cas si l'on peut faire de même avec un papier et un crayon en un temps fini).

• Quelle que soit la donnée sur laquelle il travaille, un algorithme doit toujours *se terminer* après un nombre fini d'opérations, et fournir un *résultat*.

• On considère les algorithmes *déterministes* :
toute exécution d'un tel algorithme sur les mêmes données donne lieu à la même suite d'opérations.

## 4.3- Des Algorithmes aux programmes

| *Programme = Algorithmes + structures de données* |
|---|

Données : valeurs , types
Algorithmes : séquences de commandes (déjà vues)

## 4.4- Exemple

Evaluation du polynôme

$P_n = a_n X^n + ..... + a_2 X^2 + a_1 X + a_0$

(La racine X et les $a_i$ sont connus)

## 4.5- Décompositions fonctionnelle/ logique/ physique

Une solution : réécrire le polynôme

$P_n = (...((a_n*X+a_{n-1})*X)*X...+(a_i)*X)...*X)+a_0$

Par exemple, pour n=4

$P_4 = ((((a_4)*X+a_3)*X+a_2)*X+a_1)*X+a_0$

Exemple : Que représente, en base 10, le nombre 5732 en base 8

③ On a n=3, X=8

**5 = 5 en base 10**
**(5 * 8)+7 = 47 en base 10**
**(47 * 8) + 3 = 379 en base 10**
**(379 * 8) + 2 = 3034 en base 10**

**Description fonctionnelle :**

$P_0 = a_0$
$P_n = P_{n-1} * X + a_n$

Correspondant à la description (n connu d'avance):

La strcuture "a" contenant le polynôme

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| **5** | **7** | **3** | **2** |

Ou à la description symétrique (inverser le tableau)

$P_n =$ $R_0 = R_1 * X + a_0$
$R_1 = R_2 * X + a_1$
$R_2 = R_3 * X + a_2$
.....
$R_{n-1} = R_n * X + a_{n-1}$
$R_n = a_n$

## Description logique : l'algorithme de l'exemple

X connu, $a_i$ donnés, calculer $P_n$

```
P := an;
i := n-1;
Tant que i ≥ 0 faire
      P := P * X + ai;
      i := i - 1;
Fin Tant que;
```

Avec le schéma Pour :

```
P := an;
Pour i := 0 bas n-1 faire
      P := P * X + ai;
Fin Pour;
```

Avec le schéma Répéter :

```
P := an;
si n > 0 alors
      i := n-1;
      répéter
            P = P * X + ai;
            i:=i-1;
      jusqu'à i<0;
Fin si;
```

## Description physique : le programme

Représentation de l'écriture abstraite $a_i$ (tableau, liste, ...)

## Traduction de l'algorithme :

- ADA

```
coefficient : ARRAY(0..n) OF INTEGER; -- un tableau d'entiers
X,P : FLOAT;                  -- X est la base, P la valeur finale
      ......                  -- remplir le tableau coefficient
P := coefficient(n);
FOR i IN REVERSE 0..n-1
LOOP
      P := P * X + coefficient(i);
END LOOP;
```

- Pascal

```
coefficient : ARRAY [0..n] OF INTEGER;
X,P : REAL;
......
P := coefficient [n];
FOR i:= n-1 DOWNTO 0 DO
      P := P * X + coefficient [i];
```

- CAML :

```
let x=10 and n=5;;              (* base=10, n=5 *)
let coef=make_vect n 1;;    (* un tableau de 5 un (par ex.) *)
let p=ref coef.(n-1);;          (* p initialisé à coef(n-1) *)
for i=n-2 downto 0 do
      p:=!p*x+coef.(i)
done;;
```

# 5- Introduction aux concepts de base

## 5.1- Valeur et type

Entité pouvant être évaluée ou stockée; faire partie d'une structure de données, être passée comme paramètres à une procédure ou fonction, retournée comme le résultat d'une fonction ....

## 5.2- Valeurs primitives

valeurs de vérités (true, false), caractères ('a', 'd'..), énumérées, entiers (1, 5, 12001), réels (2.57),.....

## 5.2'- Valeurs composées

- enregistrements (une personne)
- tableaux (plusieurs entiers ordonnés)
- ensembles (ensemble d'élèves), listes, tuples ,....
- fichiers (fichier d'élèves)

**Opérations sur les valeurs**

## 5.3- Type

Caractérisé par un ensemble de valeurs et un ensemble d'opérations sur ces valeurs : **Valeur (simple, composée) <==> Type(simple, composé)**

### *5.3.1- Type simple*

Les valeurs d'un type simple ne peuvent pas être décomposées en valeurs plus simples.

### *5.3.2- Exemples de types simples*

***booléen*** : {false, true};
***caractères*** : { ...., 'a', 'b',....,'z', ....}
***entiers et réels*** : {... -2, -1, 0, 1, 2,....};{... , -1.0, ..., 0.0, .... , 1.0, ...}
***énuméré*** :

```
type jours  is (lundi, mardi, ....., dimanche);        -- ADA
type jours = lundi | mardi | .....| dimanche;;         --CAML
```

## 5.4- Expression simples

Une expression est une phrase qui sera évaluée pour émettre (produire) une valeur.

Expressions simples :

- Littéraux
- Références aux constantes et aux variables
- Expressions conditionnelles
- Appels de fonctions

### *5.4.1- Littéral*

Dénote une valeur d'un type donnée.

**Exemple: 37, 3.14, "cnam" , 'x'**

### *5.4.2- Références aux constantes et aux variables*

```
pi : constant := 3.14; -- référence à une constante
x : integer;           --  référence à une variable
personne.nom := ..   -- référence à une variable composée
```

### *5.4.3- Expression conditionnelle*

```
if (a>0) then faire_ci else faire_ca;;          -- CAML
if (a>0) then faire_ci else faire_ca; end if;   -- ADA
```

### *5.4.4- Appels de fonctions*

```
sin(45)
factorielle(3)
```

## 5.5- Commandes

**Commande** : une phrase qui est exécutée pour mettre à jour des variables

Les commandes simples :

- skip
- affectation
- commandes séquentielles
- commande de rupture de séquence (Goto)
- commandes conditionnelles
- commandes itératives (déjà vues)

### *5.5.1- Skip*

- La plus simple forme des commandes
- Sans aucun effet !!
- Utilisée dans les conditionnelles

```
if a>0 then a:=a+1 else null; end if;          -- ADA
if (a>0) then a:=a+1 else begin (*rien*) end;  -- PASCAL
```

### *5.5.2- Affectation*

```
n:=0;                          -- ADA
n := n+1;
let n = 5;;                    --CAML
let n = n+1;;
```

### *5.5.3- Commandes séquentielles*

Les commandes mettent à jour des variables
③ il faudra établir un ordre dans ces mises à jours.

*C1; C2* indique que les commandes C1 et C2 doivent s'exécuter l'une après l'autre et dans cet ordre.

## 5.5.4- Commande de rupture de séquence

Goto : branchement à un endroit du programme
Exit : sortie du bloc courant

## 5.5.5- Commande conditionnelle

Schéma général: *Si E Alors Commande1 Sinon Commande2 Fin si;*

*E* peut être de la forme :
*non E1, E1 et E2, E1 ou E2*

```
if a=5 then b:=6 else b:=7 end if;          -- ADA
let x= (if a > 5 then b else c);;           (* CAML *)
(if a > 5 then b else c) = 7;;              (* CAML *)
```

## 5.5.6- D'autres commandes

Une procédure = une commande complexe
Appel de procédure

- Lecture get(X)
- Ecriture put("blabla")
  put(X)

## 5.5.7- Regroupement des commandes

bloc :

```
begin
    commandes
end;
```

# 6- Manipulation de variables : Stockage

- Un stockage est une collection de cellules (cases ou contenants)
- Chaque cellule a un statut : *alloué* ou *libre*
- Chaque cellule allouée possède une valeur courante (contenu)

## 6.1- Variables

- Une variable est un objet contenant une valeur.
- Une variable est désignée (référencée) par un nom.
- Ce nom est l'*identificateur* de la variable

Exemples de (noms de) variables

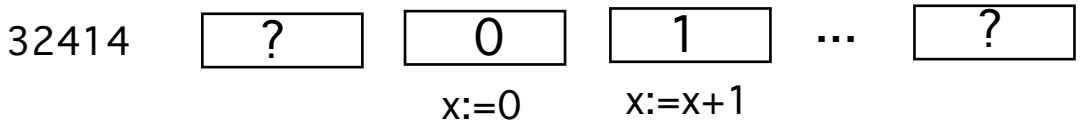
**population_du_monde**
**date**
**température**
**x, y, a**

p Dans les langages que nous considérons, toute variable est typée.

```
x : integer;          --précise que x est un entier
```

## 6.2- Mise à jour des variables

```
.....
x : integer;          -> allocation de la cellule
x := 0;               -> on y met la valeur 0
x := x+1;             -> on modifie la valeur
                      la cellule nommée par x contient 1
.....
```



## 6.3- Environnement

• Toute expression ou commande d'un programme est interprétée dans un environnement (contexte) particulier.

• L'environnement intervient dans l'interprétation
d'expressions telles que $n + 1$ ou $sin(x)$
ou de commandes telle que $x := 0$ ou $x := x+1$

③ Une expression peut avoir différentes interprétations dans différentes parties d'un programme.

• Un *environnement* contient un ensemble de couples
**[identificateur |→ valeur].**

Chaque identificateur est *lié* à une valeur ou à un type.
En général, dans un environnement donné, il existe une seule association pour un identificateur.

• Environnement et la notion de bloc (les limites de l'environnement) :

```
begin
    a:=b;
    c:=factorielle(b);
end;
```

### *6.3.1- Exemple ADA*

```
procedure P is
    z : constant := 0;
    c : character;
begin
♣.....;
end P;
```

L'environnement au point ♣ est
{ c |→ une variable caractère
z |→ l'entier 0
p |→ la procédure }

### *6.3.2- Exemple CAML*

```
let y = 0;;
let f x = □ if x>y then 1 else 0;;
f : int -> int = <fun>
```

L'environnement au point □ est
{ x |→ une variable locale
y |→ l'entier 0 }

## 6.4- Définitions et déclarations

• L'élaboration des environnements se fait par des définitions et des déclarations.

• Les déclarations produisent des liens (créent des couples) :
- Les déclarations de types créent de nouveaux types;
- Les déclarations de variables créent de nouvelles variables.

• Une définition est une déclaration simple qui établit une association.

## 6.5- Définitions

• Elles concernent les constantes, les procédures et les fonctions.
• Elles ont pour rôle d'établir un lien [identificateur |→ ...].
• En général, le programmeur utilise ces définitions pour déclarer une fois pour toute un lien qu'il utilisera dans le reste du programme.

ADA :

```
pi : constant := 3.14;                    -- pi représente 3.14
alpha : constant character := 'a';
```

CAML :

```
let  x = 1 ;;
let a = ref 12;;
```

## 6.6- Définition et déclaration de type

Permet de lier un identificateur à un type existant ou d'en définir un nouveau.

**ADA :**

```
type car is character;
type ligne is new integer range 1..66;
type nom_de_mois is (jan, fev,...., dec);
```

**CAML :**

```
type livre = {titre : string; auteur : string};
```

## 6.7- Définition et Déclaration de variables

• Une définition de variable lie un identificateur à une variable existante.

```
pop : Integer renames population(region);      -- ADA
```

Lie l'identificateur *pop* à une variable entière qui est un composant du tableau *population*.
La valeur de *pop* dépend de la valeur actuelle de *region*.

• Une déclaration de variable permet de créer une nouvelle variable distincte :

```
age : integer;
```

Crée une nouvelle variable de type entier à laqeulle est associé l'identificateur *age*.

```
s : sexe := feminin ;
```

Crée une nouvelle variable de type sexe (masculin, feminin) et lui associe l'identificateur *s*.

## 6.8- Déclaration séquentielle

C'est une série de déclarations traitées les unes après les autres et dans cet ordre.

• En ADA, les déclarations sont séparées par ';'

```
Declare
        compte : integer;
        r : float;
begin
        compte := compte +1;
end ;
```

• En CAML, les déclarations sont séparées par ';;'

```
let x="coucou";;
let y = x ^ "c est moi";;
```

Aussi, par *and* dans les déclarations collatérales :

```
let a=2 and b=3;;
```

## 6.9- Exemple

Les environnements associés aux points ♣ à ⑦ dans le programme suivant :

```
        Declare
                x : constant := 999;
♣               type Nat is 0..x;
□               m,n : Nat;
        begin
⑦               ....
        end ;
```

**Solution (sans les valeurs associées)**

1) {x}
2) {x, Nat}
3) {x, Nat, m , n}

# 7- Exemple de programme ADA

```
-- calcul de la valeur d'un entier presente sous forme d'un nombre romain

WITH text_io; use text_io;
PROCEDURE romain IS
      TYPE chiffre_romain IS ('I','V','X','L','C','D','M');
      TYPE nombre_romain IS ARRAY(Positive range <>) OF chiffre_romain;

      mille_neuf_cent_quatre_vingt_quatre :
                              CONSTANT nombre_romain :="MCMLXXXIV";

      S : chiffre_romain := 'D';   -- non utilisé, pour montrer un exemple
      romain_to_entier : CONSTANT ARRAY(chiffre_romain)  OF
                                    INTEGER := (1,5,10,50,100,500,1000);
      v : INTEGER := 0;
      R : nombre_romain(1..9) := mille_neuf_cent_quatre_vingt_quatre;

BEGIN
      FOR i IN R'range LOOP
            IF i /= R'last AND THEN   -- pour I+1
                        romain_to_entier(R(i)) < romain_to_entier(R(i+1))
            THEN        v := v - romain_to_entier(R(I));
            ELSE        v:= v+ romain_to_entier(R(I));
            END IF;
      END LOOP;
      put_line(integer'image(v));         -- donne 1984 pour "MCMLXXXIV"
END romain;
```

# Notion de programme et d'instruction

# 8- Exemple de programme CAML

```
(* La fonction Fibonacci  *)
let rec fib n =
if n < 2 then 1 else fib(n-1) + fib(n-2);;
fib : int -> int = <fun>
```

**Pourquoi ADA, CAML**

**- Ce cours ne forme pas de programmeur**

**- Pour illustrer les concepts fondamentaux et modernes, il faut un langage qui possède ces concepts.**

**- Les langages classiques tels que Basic, Fortran, C, Pascal n'offrent pas ces concepts**

**- ADA et CAML véhiculent ces concepts.**

**- Les autres candidats possibles :**

**- Impératif : C++ mais :**

**. Problème de standard**

**. Certains concepts ne sont introduits que récement. Le langage n'est pas encore stable de ce point de vue.**

**. C++ véhicule le paradigme objet (dès le début) dont on ne veut pas parler ici (car ca doit s'enseigner sérieusement et non pas croire que le programmeur C d'hier glisse vers C++ sans problème. Il peut glisser masi fera du C en C++.**

**. Le paradigme objets a des problèmes théoriques.**

**. Le paradigme n'a pas tenu ses promesses de réutilisation.**

**. ADA est stable (depuis 83)**

**. La décision de langage s'est pris il y a 4 ans. La décision est universitaire (et donc loin du monde industriel ?)**

**. Il fallait former les élèves de CNAM à un langage en enseignant les concepts importants, il y avait aucun candidat autre qu'ADA**

**. C++ est plus utilisé car C est plus utilisé et on fait croire que l'on passe de l'un à l'autres sans pb.**

**. ADA est moins utilisé car la formation à ADA est plus difficile, on trouve moins de programmeur ADA, les compilos sont plus chers...**

**. Le choix de CAML est fait pour le paradigme fonctionnel (et mathématique).**

**. Des projets sérieux (des entreprises sérieuses) se font en ADA**

**. La partie TR d'ADA est tout à fait industriel**

**. ADA évolue et ADA 9X apporte des concepts nouveaux et POO.**

# 9. Conventions d'écritures et formes algorithmiques

 Intérêt des conventions algorithmiques
 Pourquoi nous ne prenons pas une convention

③ La convention choisie sera un sous-ensemble d'ADA (+CAML)
 Pourquoi ADA & CAML

# 10. Notions de base (suite)

# 11. Type

Ensemble de valeurs et d'opérations définies sur ces valeurs.

**Valeur (simple, composée) <==> Type(simple, composé)**

## 11.1. Type simple

Les valeurs d'un type simple ne peuvent être décomposées en valeurs plus simples.

Exemples de types simples :

*booléen* : valeurs de vérités {false, true}
*entiers* : {... -2, -1, 0, 1, 2,....}
*réels* : {... , -1.0, ..., 0.0, .... , 1.0, ...}
*caractères* : { ...., 'a', 'b',....,'z', ....}
*énuméré* :

```
type couleur  is (rouge, blanc, vert);          -- ADA
type couleur  = rougel blancl vert;;            --CAML
```

## 11.2. Types prédéfinis (ADA)

**integer, float, boolean, character,**
**string,**
**array (...) of, Record...**

## 11.3. Valeurs des types

- character : ‘a’, ’z’ ‘A’, ‘B’, ‘1’, ’9’, ‘&’ , ‘{‘ ...
- integer : -324, 790
- float : 1.25, -32.87
- boolean : true, false
- string : “école centrale”

## 11.4. Déclaration et définition de types

On précise le mot **Type (subtype)**

```
Type Tab is array (1..n) of float;
Type Mot1 is NEW string(1..26);
SubType Mot2 is string(1..26);
Type jours is (lundi, mardi, ..., dimanche);
```

## 11.5. Hiérarchie de types en ADA

### *11.5.1- Type dérivé*

- **Le concept**
- **Exemple**

```
TYPE poids is NEW integer range 1..200;  --TYPE poids is range 1..200;
TYPE mesure is NEW integer;
```

### *11.5.2- Le sous-type*

- **Le concept**
- **Exemple**

```
SUBTYPE mes_couleurs is couleur RANGE vert..noir;
SUBTYPE majuscule is character RANGE ‘A’..’Z’;
SUBTYPE petit is integer RANGE 1..10;
```

*La définition d’un sous type ne crée pas de nouveau type.*

## 11.6. Equivalence de types en ADA

En ADA, l'équivalence de type se fait par nom.

```
type couleur is (rouge, jaune, gris, vert, blanc);
type color is (rouge, jaune, gris, vert, blanc);
```

Introduisent deux types distincts, bien que textuellement identiques.

## 11.7. Déclarations de variables

```
i : integer;
t : tab;
```

- Déclaration avec initialisation

```
i : integer := 12;
t : array (1..6) of integer := (1,2,4,6,8,12);
```

## 11.8. Définitions de constantes

Comme pour les variables + le mot *Constant*

```
pi :    Constant := 3.14;
```

- Pour les types composés, l'initiation se fait par *agrégat*

```
t : Constant array (1..6) of integer := (1,2,4,6,8,12);
tt :    Constant array (1..10) of character :=
          (1..3 => 'a', 7|8 => 'b', 9 => 'c', others => 'z');
ecole :  Constant string (1..8)  := "centrale";
la_petite : Constant personne := (marie, dupont, ...);
```

## 11.9. Les types en CAML

- Types de base : **int, float, char, string, bool, unit**
- Types composés (vecteur, liste, enregistrement, tuple, ...)
- Typage par construction :
  - Descripteur de type $\alpha$ -> $\beta$
  - Typage dynamique/statique (donné par l'utilisateur) de CAML
  - Typage selon la valeur

## 11.10. Exercices

Montrer l'ensemble des valeurs des types suivants:

```
type    serie is (pique, trefle, coeure, carreau);
type    rang is integer range 2..14;
type    voyelle is ('a','e','i','o','u');
subtype  majuscules is character range 'A'..'Z';
```

**Solution**

serie = {pique, trefle, coeure, carreau}
rang = {2,...,14}
voyelle = {'a','e','i','o','u'};
majuscules = {'A' .. 'Z'};

# 12. Commandes

**Rappels**

• Une commande est une phrase qui est exécutée pour mettre à jour des variables

• Les commandes simples :
- skip
- affectation
- commandes séquentielles
- commande de rupture de séquence (Goto)
- commandes conditionnelles
- commandes itératives
- commande d'appel de procédure

❒ Skip, Affectation, séquence et bloc

```
Begin
   a:=b;                    ;                        En CAML
   null;                 begin .... end ;;     En CAML
   c:=;
end;
```

❒ Appel de procédures
- **Lecture** : ADA **get(note)**
  : CAML **read_int()/ read_line() / read_float()**
- **Ecriture** : ADA **put("la note est"); put(note)**
  : CAML **print_string("..."); print_int note;;**

## 12.1. Exercices

**1**- Lecture et écriture d'un nombre, d'un caractère.

| ADA | CAML |
|---|---|
| X : integer ; | |
| C : character ; | -- les caractères en CAML : `a` `z` |
| Begin | |
| Get(X); put(X); | #let x=read_int(); print_int x;; |
| Get(C); put(C); | – La manipulation directe des caractères est |
| End; | – difficile en CAML. Utiliser les conversions. |

**2**- Calcul de la moyenne M de deux entiers X et Y.

| | |
|---|---|
| X,Y,M : integer; | |
| Begin | -- Min/Maj différencié en CAML |
| M := (X + Y) / 2; | # let X= ... and Y=... ;; |
| End; | # let M=(X+Y)/2 |

**3** - Permutation de deux nombres X et Y.

- Solution classique utilisant une variable intermédiaire

```
X,Y,Z : Integer;
Begin
   Z := X; X := Y; Y := Z;
End;
```

- Autre solution (sans utiliser une variable intermédiaire)

```
X,Y : Integer;
Begin
   X := X + Y; Y := X - Y; X := X - Y;
End;
```

# 13. Expressions

## 13.1. Expressions arithmétiques

### *13.1.1- Opérations arithmétiques*

Opérateurs classiques : **+, - , * , / , **, mod, abs, ...**

```
a:= (b+c-d /2) mod 4;
```

## 13.2. Expressions conditionnelles

### *13.2.1- Comparaisons*

>, < , <= , >= , /= , =

Entre les nombres, chaînes, caractères, éléments d'un type énuméré ...

```
1 < 2
1 /= 3
1.5 > r+21.8
"abc" < "bad"
'a' < 'b'
```

## 13.3. Utilisation des expressions conditionnelles

- *If Expr_cond*
  *Then C1*
  *End if;*

- *If Expr_cond*
  *Then C1*
  *Else C2*
  *End if;*

*==> Elsif*

*Expr_cond* est de la forme (*and* prioritaire sur *or*):

| | | |
|---|---|---|
| not Expr1 | not | En CAML |
| Expr1 and Expr2 | & | == |
| Expr1 or Expr2 | or | == |

p Ordre d'évaluation indéfinie

Expr1 and Then Expr2,
Expr1 or Else Expr2

## Exemples :

- ```
  If    a=5
  Then  b:=6;
  Else  b:=7;
  End if;
  ```

- ```
  If    (a /= 5) AND (a<= 10)
  Then  b:=a;
  Else  c:=a;
  End if;
  ```

- ```
  If    (a > 5) OR ((a <= 10) AND NOT b)
  Then  c:=a;
  Else  a:=c;
  End if;
  ```

- ```
  If    (a < b) AND THEN (b < c)
  Then  pluspetit := a; plusgrand := c;
  Elsif (a>b) OR ELSE (b>c)
  Then  pluspetit := c; plusgrand := a;
  Else ...
  End if;
  ```

## 13.4. Case

### Exemples :

```
Case x is
    when   1        =>   a := b;
    when   2..5     =>   b := c;
    when   6|7|9      => f := ...
    when   Others  =>   c :=...
End Case;
```

### Reconnaissance des caractères

```
c : character;
Begin
   Get(c);
   case c IS
        When '0'..'9'     => Put("chiffre");
        When '-'|'_'      => Put("tiret");
        When 'a'..'z' | 'A'..'Z' => Put("lettre");
        When '$'          => Put("dollar");
        When others       => Put("caractère spécial?");
   End case;
End;
```

## 13.5. Exercice

Ecrire un programme qui lit au clavier
- un des opérateurs '+', '-', '/', '*'
- deux opérandes entiers;

effectue l'opération et affiche le résultat.

Ecrire un programme qui lit au clavier
- un des opérateurs '+', '-', '/', '*'
- deux opérandes entiers;

effectue l'opération et affiche le résultat.

## Solution :

```
operateur : character;
x,y,r : Integer;
op_valide : Boolean := True;
Begin
    Put("donner l'opérateur"); Get(operateur);
    Put(" donner deux entiers"); Get(x);Get(y);
    Case operateur IS
      When '+' => r:= x + y;
      When '-' => r:= x - y;
      When '*' => r:= x * y;
      When '/' => r:= x / y;          -- faisable si y ≠ 0
      When Others =>
        Put("opérateur inconnu");
        op_valide := False;
    End case;
    IF op_valide THEN
      Put("le résultat est = ");Put( r);
    End IF;
End;
```

## 13.6. L'opérateur IN

On peut utiliser l'opérateur *IN* sous ses différentes formes:

*X IN intervalle*
*X not IN intervalle*
not *X IN intervalle*

Exemples (X de type character) :

- IF X IN 'a'..'z' THEN
  Put("minuscule");..

- IF not(X IN 'a'..'z' or X IN 'A'..'Z') THEN
  Put("ce n'est pas une lettre");..

- IF X not IN '0'..'9' THEN
  Put("ce n'est pas un chiffre");..

**Exemple :**

```
subtype chiffre is  character range '0'..'9';
subtype lettre is  character range 'A'..'Z';

c : character;
begin
   get(c);
      if c in chiffre then
         put_line("chiffre");
      elsif c in lettre then
         put_line("lettre");
      else put_line("je ne sais pas");
      end if;
end;
```

## 13.7. Commandes (suite)

## 13.8. Itération

I) Le schéma général de l'itération :

*Répéter à l'infini*
*C;*
*Fin Répéter;*

*Loop*
*C;*
*End Loop;*

II) *Tant que E Faire*
*C;*
*Fin tQ;*

*While E loop*
*C;*
*End loop;*

**Exemple :**

```
While i /= j
loop
   i := i-j;
   q := q+1;
End loop;
```

III)

*Répéter*
*C;*
*Jusqu'à E;*

*Loop*
*C;*
*Exit when E;*
*End loop;*

- La commande *Exit when E;* est équivalent à *If E then Exit; End If;*

- La commande de rupture de séquence *Exit* (échappement) dans les boucles provoque la sortie de la boucle la plus interne.

**Exemple :**

```
Loop
   i:=i+1;
   ....
   Exit when (i > j);
End loop;
```

## IV) *Pour Compteur Dans E1..E2 Faire*

*C;*

*Fin Pour;*

Ce schéma est traduit par :

*For Compteur IN E1..E2 loop*

*C;*

*End loop;*

Remarques : - *Compteur* n'est pas à déclarer
- *Compteur* ne doit pas être modifié
- E1 .. E2 ne doit pas être modifié

**Exemple :**

```
s := 0;
For i in 1..n loop
   s := s+i;
End loop;
```

## V) *Pour Compt Bas E1..E2 Faire*

*C;*

*Fin Pour;*

traduit par :

*For Compt IN <u>Reverse</u> E1..E2 loop*

*C;*

*End loop;*

**Exemple :**

```
i := 0;
Loop
   While true loop
      i := i + 1; ....          -- modification de "assez"
      If assez Then EXIT;       -- exit when assez
      End if;
   End loop;
   Exit when (I > 10);
End loop;
```

**Exercice :** écrire un programme qui affiche :

*A(jout, C(onsultation , S(uppression, F(in*

*Choix? : _*

et récupère une réponse valide (rejette les réponses non valides par un message d'erreur et redemande un choix).

**Exercice :** écrire un programme qui affiche le menu :

*A(jout, C(onsultation , S(uppression, F(in*

*Choix? : _*

et lit une réponse valide et effectue (symboliquement) l'opération associée. On rejette les réponses non valides par un message d'erreur.

Après un ajout, une consultation ou une suppression, on peut envisager de réafficher le menu et réitérer.

**Une solution :**

On pilote le lancement des procédures adéquates à partir de ce menu. On termine lorsque le choix = Fin.

```
C : character;
   Begin
        Loop
          Put("A(jout, C(onsultation , S(uppression, F(in");
          Put("choix?: ");
          Get(C);
          CASE C is
             When 'A'     => ajouter;
             When 'C'     => consulter;
             When 'S'     => supprimer;
             When 'F'     => null ;       -- rien
             When others => Put("choix non valide");
          End case
          Exit When C='F';
        End Loop;
   End;
```

## 13.9. Exercices

1- Calcul de $2^i$, le ième terme de la suite $U_n$, $n \geq 0$ :

$U_0 = 1$ -- $2^0$

$U_n = U_{n-1} * 2$ -- $2^n$, $n > 0$

2- Calculer le $k^{ième}$ terme de la suite

(a est une constante disponible que l'on peut lire d'abord)

$U_0 = a$

$U_{n+1} = 2^n * U_n / n+1$ <==> $U_n = 2^{n-1} * U_{n-1} / n$

3- Calculer le terme $U_n$ de la suite de Syracuse définie par ($a > 0$):

$U_0 = a$

$U_{n+1} = 3 * U_n + 1$ si $U_n$ est impaire;

$U_{n+1} = U_n/2$ si $U_n$ est paire.

Ecrire un algorithme démontrant (par construction) la conjecture suivante : la suite de Syracuse tend vers 1 (le terme final est 1).

4- Calcul du pgcd, le plus grand diviseur commun de deux nombres M et N (algorithme d'Euclide)

- Exemples

  Pour N= 3 , M=2 , pgcd(N,M)=1

  Pour N=12, MI=8 , pgcd(N,M)=4

- Exposer les méthodes de calcul
  - par '-'
  - amélioration : par reste de la division

## Solutions :

1- Calcul de $2^i$ :

$U_0 = 1$ -- $2^0$

$U_n = U_{n-1} * 2$ -- $2^n$, n >0

```
-- la valeur de n est connue ici
X : integer := 1;
Begin
   For i in 1..n Loop
      X := X * 2;
   End Loop;
End ;
```

2 - Calculer le $k^{ième}$ terme de la suite

$U_0 = a$

$U_{n+1} = 2^n * U_n / n+1$ <==> $U_n = 2^{n-1} * U_{n-1} / n$

```
-- la valeur de n est connue ici
a, s : integer;
Begin
   S := a;
   For I in 1..n loop     -- pas besoin de déclarer I
      S := 2 ** (I-1) * S / I;
   end loop;
End;
```

3- Calculer le terme $U_i$ de la suite de Syracuse définie par :

$U_0 = a$

$U_{n+1} = 3 * U_n + 1$ si $U_n$ est impaire;

$U_{n+1} = U_n/2$ si $U_n$ est paire.

```
-- les valeurs de n et de a sont connues ici
terme : integer;
Begin
   terme := a;
   For I in 1.. n Loop
      If (terme  mod 2) = 0    then    terme := terme / 2;
      Else                      terme := 3 * terme +1;
      End if;
   End loop;
End ;
```

**Conjecture** : la suite de Syracus tend vers 1.

```
Begin
   terme := a;
   While terme > 1 Loop            -- au lieu de "/="
      If (terme  mod 2) = 0 then
               terme := terme / 2;
      Else
               terme := 3 * terme +1;
      End if;
   End loop;
End;
```

L'algorithme se termine et la valeur finale de terme = 1.

4- Calcul du pgcd, le plus grand diviseur commun de deux nombres M et N (algorithme d'Euclide)

- Exemples
  Pour N= 3 , M=2 , pgcd(N,M)=1
  Pour N=12, MI=8 , pgcd(N,M)=4

• Solution sous forme d'un programme

```
x,y,r : Integer;
Begin
   put("donner x et y /= 0 :");          .......
   get(x); get(y);                       ........
   r := 1;                                    .......
-- Avec TANT QUE                    -- Avec REPETER
   while r /= 0 Loop                      Loop
      r := x mod y;                                r := x mod y;
      x := y;                                      x := y;
      y := r;                                      y := r;
                                                   Exit when r=0;
   End loop;                              End loop;
   put("le résultat est ", x);                        .........
End;
```

## 13.9. Quelques structures de contrôle en CAML

```
if cond then exp1 else exp2
```

**Remarque :**

```
if cond then exp  équivalent à
if cond then exp else ()
```

Exemple : # if true then 1;;
Erreur car 1 de type int ne peut pas être utilisé avec le type unit.

### *13.9.1- Case en CAML*

```
match expr with
   exp1   -> exp11
|  exp2   -> exp21
|  expn   -> expn1
```

**Exemple** (lien 0 -> faux; autre = vrai) **:**

```
#match x with
      0  -> false              (* texte du commentaire *)
   |  _  -> true;;          (* '_' veut dire autre *)
- : bool = true                  (* étant donné x=2 *)
```

## 13.10. Notion d'effet (de bord) en CAML

- Les entrées sorties (read_int(), print_int(),...)
- Les données mutables (modifiables)
  - les références et leur affectation (:=)
  - les tableaux (structures mutables) modifié avec '<-'
  - les enregistrements avec un champ mutable (modifié avec '<-').

- La programmation par effet s'appelle la programmation **impérative**. Dans ce paradigme, une suite d'ordre (commandes) sont données à la machine, y compris les déclaration des cases mémoire, etc....
- Par opposition, en programmation **fonctionnelle**, on laisse la machine calculer un résultat à partir d'une formule (fonction) sans trop préciser l'ordre dans lequel elle doit opérer.

**Exemple :**

en CAML: x * x décrite une multiplication (2 * 2 donne 4)
en ADA : il faut déclarer une case, l'initialiser, ....

## 13.11. Les effets de bord par les données mutables

Les références avec **:=**

### *13.11.1- Exemple*

```
#let x1=1;;
x1 : int = 1
#let x1 = x1+1;;
x1 : int = 2
#x1;;
- : int = 2
```

```
        #let x2 = ref 1;;
        x2 : int ref = ref 1
#x2 := !x2 + 1;;
        - : unit = ()
        #x2;;
        - : int ref = ref 2
```

## 13.12. Les boucles en CAML

Les boucles permettent la manipulation des références et des tableaux pour répéter une suite d'effets (de bord).

### *13.12.1- "While" et "for" en CAML*

- La valeur rendue est unit () c'est à dire "rien".

```
while expression_booléenne do
commandes de mise à jour des mutables
done;;
```

```
for compt = expr to [downto] expr do
   commandes de mise à jour des mutables
done;;
```

- L'essentiel de CAML est la fonction !
- Attention aux majuscules/minuscules
- Les commentaires entre (* .... *)

## *13.12.2- Exemples*

```
#for i=0 to 9 do
      print_int i          (* opération à effet de bord *)
   done;;
0123456789- : unit = ()
```

```
#let s = ref 0;;           (* opération à effet de bord *)
   while !s <= 9 do
      print_int !s;        (* opération à effet de bord *)
      s:= !s +1            (* opération à effet de bord *)
   done;;
s : int ref = ref 0
0123456789- : unit = ()
```

Boucle avec effet de bord

```
#let a=ref 1;;
a : int ref = ref 1

#for i=0 to 9 do
a := i
done;;
- : unit = ()
```

Boucle sans effet de bord ==> erreur

```
#for i=0 to 9 do
let e=1    (* sans effet *)
done;;
> Toplevel input:
>done
>^^^^
> Syntax error.
```

# 13.13. Données et Types (suite) : Tableaux

## *Concept d'application (association)*

Tableaux
Fonctions

Notation : $\tau_1$ |–> $\tau_2$

- Un tableau est un objet composé d'éléments qui sont tous du même type.
- Chaque élément est désigné par un indice qui correspond à son rang dans le tableau.
- Un tableau est également considéré comme un produit cartésien (voir plus loin)
- L'indice doit être de type discret (énuméré ou entier).

## *Exemple:*

| *1* | *2* | *6* | *24* | *120* | *720* | ... | ... |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | | 10 |

```
type Tfact is array (1..10) of integer;

-- remplir une variable (fact) du type Tfact avec les
-- valeurs de factorielles de 1 à 10
```

*fact* définit l'application {1..10} |–> integer

{1 |-> 1, 2 |-> 2, 3 |-> 6, ...}  fact(3) donne 3! = 6

## *Remarque sur les notations fonctionnelles*

**|->**
**->**

## *13.13.1- Les tableaux en ADA*

## *Exemples de définition/déclaration :*

```
TYPE couleur is (blanc, noir, rouge, vert, bleu...);
TYPE tableau_couleur IS ARRAY (couleur) OF integer;

TYPE tableau_entier IS ARRAY (1..5) OF integer;

TYPE tableau_caractere is ARRAY ('A'..'Z') OF integer;

TYPE jour is (lundi,....., dimanche);
TYPE semaine is ARRAY(jour) OF boolean;

TYPE ligne is ARRAY(1..max) OF character;
TYPE mot is ARRAY(1..10) OF character;
TYPE phrase is ARRAY(1..5) OF mot;
TYPE texte is ARRAY(1..20) OF phrase ;
```

Pour les tableaux à N dimensions, l'ensemble des indices est formé par le produit cartésien de N types (ou sous-types) discrets:

```
TYPE matrice_2 is ARRAY(1..20, 1..30) OF integer;
TYPE matrice_3 is ARRAY(1..20, 1..30, -4..3) OF integer;
```

Le type des indices doit être discret. Le type des éléments est quelconque y compris un tableau.

```
TYPE matrice_2_2 is ARRAY(1..20,1..30) OF matrice_2 ;
```

Ce qui est différent de :

```
TYPE matrice_4 is ARRAY(1..20,1..30,1..20,1..30) OF integer;
```

- Habituellement, les bornes d'un tableau doivent être connues lors de l'élaboration du type (exécution). Une fois évalué, ces bornes restent fixes.

On peut cependant avoir des tableaux "dynamiques" comme dans :

```
Procedure tab_dyn (N : integer) is
      TYPE tabvarie IS ARRAY (1..N) OF integer;
.....
```

## *13.13.2- Déclaration de variables tableaux*

```
grille : ARRAY(1..80, 5..100) OF boolean;
melange : ARRAY (couleur ) OF integer;
page : ARRAY (1..50) OF ligne;
```

## *13.13.3- Accès aux éléments des tableaux*

- Pour les tableaux à une dimension, on précise l'indice.
- Pour un tableau de tableau T, on précise T(i)(j); T(i) étant considéré comme un tableau pouvant recevoir un autre tableau.
- Pour un tableau à plusieurs dimensions T, on précise T(i,j,...) pour accéder à un élément simple.

## *13.13.4- Opérations sur les tableaux*

Les opérations d'affectation, de test d'égalité et d'inégalité sont autorisées sur les tableaux de même type et de même taille.

## *13.13.5- Opérations d'affectation et d'égalité*

```
mat1,mat2 : matrice;
sem1,sem2 : semaine
ents1,ents2 : tableau_entier,

mat1 := mat2;
sem1 := sem2;
if mat1 = mat2 THEN...
ok := (ents1 = ents2);
```

Dans ces deux derniers cas, les comparaisons sont faites élément par élément. Les deux objets doivent avoir le même nombre d'éléments.

## *13.13.6- Tableaux à une dimension (vecteurs)*

Hormis les opérations définies sur les tableaux en général, les opérations **< , <= , > , >= , &** sont définies sur tableaux à une dimension (appelés vecteurs).

p Les opérations < , <= , > , >= ne sont définies que sur les vecteurs dont les éléments sont d'un type discret (comparables) :

```
IF ents1 <= ents2 THEN.....     -- tableaux de type discret
```

- Illustration de "<=" sur deux vecteurs ents1 et ents2 par différentes boucles (for, while, repeat); introduction aux *tables de vérités*.

- L'opérateur de concaténation & est défini entre deux tableaux de même type et entre un tableau et un élément de tableau :

```
ents1 & ents2
```

représente un tableau de 10 entiers.

- Pour un tableau de booléens, on a les opérateurs logiques :

```
sem1 := sem1 AND sem2;      -- tableaux de booléens
```

## 13.14. Exemple

1- Lecture, remplissage et écriture d'un tableau d'entiers T(1..N) sachant que ces opérations doivent être effectuées élément par élément.

```
          N : Constant Integer :=....;
Type      Tab is Array (1..N) of integer;
          T : Tab;
Begin
      -- Lecture et remplissage
      For Indice In 1..N Loop
         Put("donner l'élément d'indice "); Put(Indice);
         Put( " du tableau ? : ");
         Get(T(Indice));
      End loop;

      -- Ecriture
      For Indice In 1..N loop
         put("l'élément d'indice "); put(Indice);
         put(" du tableau est : ");
         Put(T(Indice)); put_line;
      End loop;
End;
```

## 13.15. Attributs des tableaux en ADA

- Ces attributs ne sont applicables qu'à un type tableau contraint ou qu'à un objet de type tableau et non pas à un type non contraint.

  Soit **objet_type** un type tableau contraint ou un objet de type tableau:

| | |
|---|---|
| **objet_type'first** | borne inférieure du premier indice |
| **objet_type'last** | borne supérieure du premier indice |
| **objet_type'range** | intervalle défini par objet_type'first .. objet_type'last |
| **objet_type'length** | nombre de valeurs du premier indice |

- Pour un tableau à plusieurs dimensions :

| | |
|---|---|
| **objet_type'first(N)** | borne inférieure du N$^{ième}$ indice |
| **objet_type'last (N)** | borne supérieure du Nième indice |
| **objet_type'range(N)** | intervalle défini par objet_type'first(N) ... objet_type'last(N) |
| **objet_type'length(N)** | nombre de valeurs du Nième indice |

## 13.16. Tableaux en CAML

- Un tableau CAML est une donnée mutable (modifiable).
- Les éléments d'un tableau CAML commencent à l'indice 0.

*Notation :*

| [\| e1 ; ... ; en \|] |
|---|

```
#[|1 ; 2 ; 3 |];;
- : int vect = [|1; 2; 3|]
```

*Création* **:**

| make_vect taille valeur-initiale |
|---|

```
#let coef=make_vect 5 1;;      (* un tableau de 5 un *)
- : int vect = [|1; 1; 1; 1; 1|]
```

*Accès :*

| nom_tableau.(indice) |
|---|

```
coef.(3);;
- : int = 1
```

*Modification d'un élément d'un tableau :*

- La modification se fait à l'aide de l'opérateur "<-"

| nom_tableau.(indice) <- expr |
|---|

*Exemple :*

```
#let coef=make_vect 5 1;;
coef : int vect = [|1; 1; 1; 1; 1|]
#coef.(1) <- 12;;
- : unit = ()
#coef;;
- : int vect = [|1; 12; 1; 1; 1|]
#coef.(4) <- coef.(0) + coef.(3);;
- : unit = ()
#coef;;
- : int vect = [|1; 12; 1; 1; 2|]
```

Somme des éléments d'un tableau :

```
#let somme = ref 0;;
#for i=5 downto 0 do
   somme:=!somme +coef.(i)
done;;
```

**somme : int ref = ref 0**
**#- : unit = ()**

**#somme;;**
**- : int ref = ref 6**

## *Quelques opérateurs sur les vecteurs :*

**vect_length** : ‘a vect -> int
**vect_item** : ‘a vect -> int -> ‘a
**concat_vect** : ‘a vect -> ‘a vect -> ‘a vect
**list_of_vect** : ‘a vect -> ‘a list
**vect_of_list** : ‘a list -> ‘a vect
....

## *Lecture du contenu d’un tableau*

```
#for i=0 to vect_length coef -1 do      (* ATTENTION à n-1 *)
   coef.(i) <- read_int()
done;;
9               (* 5 entiers entrées au clavier *)
7
13
45
90
- : unit = ()
#coef;;
- : int vect = [|9; 7; 13; 45; 90|]
```

## *Remplissage et affichage des élément d'un tableau :*

```
#let r=make_vect 3 "bonjour";;
r : string vect = [|"bonjour"; "bonjour"; "bonjour" |]

#r.(1) <- " tout";;
-: unit = ()

#r.(2) <- "le monde! ";;
-: unit = ()
#r;;
- : string vect = [|"bonjour"; " tout"; "le monde! " |]


#for i=0 to vect_length(r) -1 do
       print_string r.(i);
       print_string " - ";
   done;;
bonjour - tout le - monde! - : unit = ()
```

## 13.17. Chaînes de caractères

**Concept**

Un tableau de caractères manipulable
- caractère par caractère
- par sous-chaîne (tranche)
- comme un tout (entièrement)

### *13.17.1- Les chaînes de caractères en ADA*

• ADA possède le type prédéfini STRING. C'est un tableau non-contraint à une dimension dont les éléments sont des caractères :

• La déclaration du type STRING est (déjà) faite par :

```
SUBTYPE POSITIVE is integer RANGE 1..integer'last;
TYPE STRING is ARRAY(positive rang <>) OF character;
```

③ l'indice de début d'un objet de type string est donc forcément ≥1.

### *Exemple :*

```
question : CONSTANT STRING := "combien de caractères?";
```

③ la contrainte d'indice est apportée par la valeur initiale.

On a *question'first = 1*, *question'last = 22* = nombre de caractères.

### *Concaténation de chaînes par &:*

• L'opérateur "&" est défini entre deux vecteurs (tableau à une dimension) de même type et entre un vecteur et un élément de vecteur.
③Il s'applique donc aux chaînes de caractères

```
demander_2_fois : CONSTANT STRING := question & question;
nom, mot : STRING(1..10);              -- contrainte explicite d'indice
prenom : STRING(1..15);              -- même type que nom
prenom := "jean claude      ";       -- 15 caractères
mot := nom;
```

Exemples de concaténation (par &) :

- entre deux chaînes : **nom(1..3) & prenom;**
- entre une chaîne et un caractère : **nom(1..3) & 'A'**
- entre deux caractères pour obtenir une chaîne : **'A' & 'B'.**

## *Exemples :*

1- Définir si un mot est un Palindrome (radar, tot, serres). On peut utiliser la fonction *Longueur(Ch)* qui donne la longueur de la chaîne *Ch*.

```
                Taille_mot : Constant Integer := ... ;
SubType  Mot IS string(1..Taille_mot);
M : Mot;
est_palindrome:  boolean:=true;    -- la variable
i : Integer := 1;
L : Integer;
Begin
      Put("donner un mot : "); Get(M);
      L := Longueur(M);
      While (i <= L / 2) and est_palindrome Loop
         est_palindrome:= (M(i)=M(L-i+1));
         i := i + 1;
      End loop;
      -- (i > L/2 & est_palindrome) OU (i <= L/2 & not  est_palindrome )

      Put ("lo mot" & M ) ;
      If est_palindrome then
         Put_line ("est un palindrome) ;
      Else
         Put_line ("n'est pas un palindrome) ;
      End if;
End ;
```

==> Modifier l'algorithme pour traiter des cas tels que :

ESOPE RESTE ET SE REPOSE

ELU PAR CETTE CRAPULE

## *13.17.2- Les chaînes de caractères en CAML*

- Il existe le type string en CAML (sans borne).
  Exemple : “ecole”

- L’opérateur ‘^’ effectue la concaténation de chaînes en CAML
- Les éléments d'une chaîne CAML commencent à l'indice 0.

## *Exemple :*

```
#let c1 = "école " and c2=" centrale";;
c1 : string = "école "
c2 : string = " centrale"

#c1 ^c2 ;;
- : string = "école  centrale"
```

- L’égalité (‘=‘) et la différence (‘<>‘ ou ‘!=‘) des chaînes.

- Certaines fonctions sont définies sur les chaînes telles que :

  **string_length : string -> int**
  **nth_char : string -> int -> char**
  **set_nth_char : string -> int -> char -> char**
  **sub_string : string -> int -> int -> string**
  ....

- Pour comparer (>, <, ...) les chaînes CAML, on peut utiliser des conversions de caractères en entier (int_of_char : char -> int).

## **Exemple** : Palindrome en CAML

```
#let r = ref 0 and i = ref 0 and l=ref 0 and est_pal = ref true;;

#let s = (print_string "donner une chaine : "; read_line());;
#l := string_length s;;
#while (!i <= !l / 2) & !est_pal do
      if nth_char s !i = nth_char s (!l - !i-1)
      then i:= !i +1
      else  est_pal:=false
done;;
```

(*---------- essai 1----------*)

```
donner une chaine : abc
s : string = "abc"
#r : int ref = ref 0
i : int ref = ref 0
l : int ref = ref 0
est_pal : bool ref = ref true
#- : unit = ()
#- : unit = ()

#est_pal;;          (* le résultat *)
- : bool ref = ref false
```

(*---------- essai 2----------*)

```
donner une chaine : serres
s : string = "serres"
#r : int ref = ref 0
i : int ref = ref 0
l : int ref = ref 0
est_pal : bool ref = ref true
#- : unit = ()
#- : unit = ()
#est_pal;;
- : bool ref = ref true
```

## 13.18. Enregistrements

- *Concept de produit cartésien*

Enregistrement et tuples (tableaux)

Notation : $\tau_1 \times \tau_2$

- Record en ADA

```
type complexe is record
   reel : float;
   imaginaire : float;
end record ;
```

*complexe* est de type "réel × réel"

- Enregistrement en CAML

```
type livre = {titre : string; auteur : string};
```

*livre* est de type "chaîne × chaîne"

**Exemple :**

```
SubType Ch15 IS string(1..15);
Type mois IS (jan, fev,..., dec);
Type Situation IS (marie, célibataire, divorce);
Type Sexe IS (masculin, féminin, neutre);

Type Date IS record
    jour : integer;
    moi : mois;
    annee : integer;
  End record;

Type Personne IS record
    nom : Ch15;
    prenom : Ch15;
    naissance : Date;       -- enregistrement DATE
    Sex : Sexe;
    Age : integer;
    Sf : Situation ;
  End record;
```

## *13.18.1- Manipulation des enregistrements*

## *Mise à jour des variables composées*

```
pierre : personne;
pierre.nom := "pierre          ";          -- 15 caractères !!
pierre.age := 15;
.....
paul : personne;
paul := ("dupont          ","paul          ", (12,fev,95),
                                    masculin,35,marie);
paul.naissance.mois := jan;
...
```

## 13.19. Enregistrements en CAML

- Comme pour les vecteurs, on peut modifier la valeur d'un champ d'enregistrement par "<-" si le champ concerné doit être déclaré *mutable*.

- Le format général de modification : **enreg.champ <- expr**
  ③ La valeur rendue est **unit** () c'est à dire "rien"

```
#type compte = {num : int; mutable solde : int};;
Type compte defined.

#let moncompte = {num=1234; solde = 2000};;
moncompte : compte = {num=1234; solde=2000}

#moncompte.solde <- moncompte.solde - 800;;
- : unit = ()
```

La modification n'est possible que par la présence du mot "mutable".

```
#moncompte;;
- : compte = {num=1234; solde=1200}
```

## 13.20. Enregistrements à champs variants : union

- *Concept d'union*

Choix d'une valeur parmi plusieurs types différents.

Notation : $\tau_1 + \tau_2$

**Exemples :**

1-
```
type appareil is (imprimante, disque);
type perif(unite : appareil) is record
   etat : character;
   case unite is
      when imprimante      =>   ligne : integer;
      when others       =>  cylindre : integer;
                                piste : integer;
   end case;
end record;
```

*perif* est de type "appareil x character × (integer + (integer × integer))"

③ le discriminant fait partie de l'enregistrement (ADA).

2-
```
type couleur  = coeur | carreau | pique | trèfle;;   --CAML
type carte = joker
                | As of couleur
                | ..... ;;
```

*carte* est de type "joker + As x (coeur + pique + .....)"

3-
```
Type individu (Sex : Sexe := neutre) IS RECORD
   nom: Ch15;
   naissance : date;    -- date est un type enregistrement
   case Sex Is
      When masculin   => barbu : Boolean;
      When feminin    => nom_jeune_fille : Ch15;
      When neutre     => null;                -- rien
   End case;
   ...
End RECORD;
```

*Manipulation :*

```
marie : individu(feminin);
mutant : individu;   -- sera de sexe neutre

la_petite : individu(feminin) :=
                (feminin, "carole      ", (13,mai,43),"helene       ");
le_barbu : individu :=(masculin, "robert       ", (1,juin,56), true);
```

**Remarques** : L'affectation directe du discriminant est interdite.
Voir l'exemple Min_Max plus loin

## 13.21. Agrégat (valeur construite par un produit cartésien )

- *Concept :*

  Expression qui construit une valeur composite (un produit cartésien) à partir de ses composants :

  ***(1, true)*** tuple CAML de type "int × bool"

```
(a*2.0, b/5)                          -- tuple CAML

t : table(pieds => 5, couleur => rouge)   -- enregistrement ADA
v : vecteur (1,5,2,4)                 -- Tableau ADA
```

**Nous les avons utilisés dans les tableaux, tuples, enregistrements, ...**

## 13.22. Exercices

1- Montrer l'ensemble des valeurs des types suivants:

```
type    sens is (haut, bas, gauche, droite);
type    rang is integer range 2..14;

type    carte is record
           s: sens;
           r : rang;
        end record;
type    plan is array(rang) of carte;
```

```
type    tour is record
           case passe : boolean is
              when false => jeu : carte;
              when true  => null;
           end case;
        end record;
```

## *Solution*

sens = {haut, bas, gauche, droite}
rang = {2,...,14}
carte = serie x rang
plan = {2..14} |-> carte
tour = carte + null;

2- En vous servant de la déclaration du type *Personne*, écrire la procédure *Recherche(Nom,Tab,N,Ind,trouve)* où *Tab* est un tableau(1..N) de *Personne*.

*Ind* sera l'indice de l'élément du tableau dont le champ nom est égal à *Nom*.

*Trouvé* indique si la personne de ce nom existe dans le tableau.

Afficher les coordonnées de la personne si trouvé est égale à vraie.

D'autres pistes d'exploitation (pour plus tard):

- Trouver les coordonnées des personnes d'age <= 70 ;
- Trouver les coordonnées des personnes de sexe féminin;
- Trouver les coordonnées des personnes divorcées;
- Trouver les coordonnées des personnes nées au printemps;
- .....
- Toute autre combinaison

## Une solution :

```
            Taille : constant Integer := ...;
   Type    Table_Personne  IS array (1..Taille) of Personne;
   Tab : Table_Personne;
   Nb_P, Ind : Integer;
   Nom : Ch15;
   Trouve : Boolean;
   i : Integer := 1;
   Rang : integer;

Begin
      -- Remplissage du tableau Tab avec Nb_P (≤ Taille) Personnes
   Put("quel est le nom de la personne à rechercher ? :");
   Get(Nom);

   -- Recherche
   While (i <= Taille)    AND Tab(i).nom /= nom Loop
         i := i + 1;
   End Loop;
   Rang := i;
   Trouve := (i <= N);
   IF Trouve THEN
            Put(Tab(Ind).nom);
            Put(Tab(Ind).prenom) ;
            ecrire_date(Tab(Ind).date); -- écriture de la date
            .........                   -- écriture des autres champs
   ELSE
      Put( Nom); Put( " : personne non répertoriée");
   End IF;
End;
```

# 14. Ensembles

- *Concept d'ensemble*

  Pour un ensemble S: $\wp(S) = \{s \mid s \subseteq S\}$

**Exemple (Pascal):**

```
type couleur = (bleu, blanc, rouge, vert);
     c : set of couleur;
```

$\wp(c)=\{\},\{bleu\},\{bleu,blanc\}, ...,\{bleu, blanc, rouge,vert\}$

## 14.1. Ensembles en ADA et CAML

• Utiliser les structures existantes pour *simuler* un ensemble.
• En ADA et CAML, une implantation par les tableaux / listes est possible.
• Il faut une définition claire des opérateurs, la partie implantation sera cachée des utilisateurs (TDA)

• Exemple ADA:

```
Type tableau_de_booleen is array(natural range <>) of boolean;

Type ensemble_d_entiers(taille_max : natural :=0) is record
     nb_elements : natural :=taille_max ;
     contenu : tableau_de_booleen(1..taille_max) := (others=>false);
End Record;
```

*nb_element* représente le nombre d'éléments actuel
*contenu(i) = true* veut dir que l'entier i est dans l'ensmble.

**Nous verrons d'autres moyens de définition d'ensembles quelconques**

# 15- Abstractions

## Le concept

• Une abstraction est une entité qui enferme un calcul.

• Regroupement logique (et justifié) d'une séquence d'expressions ou une séquence de commandes.

• Création de command (expression) complexe à partir d'éléments plus simples.

• L'utilisateur n'a pas à savoir comment le calcul est fait dans une abstraction. Il sait simplement ce qui est calculé et comment évoquer cette abstraction.

• Dans

```
put(X);
A := sin(45);
etablir_fiche_salaire(dupont);
X := Y + Z;
```

On utilise des abstractions

# 16- Abstractions en ADA

• Fonction et procédure

• Une fonction est une abstraction qui enferme une expression à évaluer. Cette expression est évaluée lorsque l'abstraction est évoquée (appelée).

• Une procédure est une abstraction qui enferme une commande à exécuter. Cette commande est exécutée lorsque l'abstraction est appelée.

## 16.1- Fonctions

La fonction *mult(x,y)* peut être implantée de différentes manières :

| | |
|---|---|
| x+x*(y-1) | (pour y > 0) |
| y+y*(x-1) | (pour x > 0) |
| x+x+.....+x | (y fois) |
| y+y+....+y | (x fois) |

L'utilisateur sait qu'il faut appeler la fonction *mult* avec deux paramètres et récupérer une valeur.

**La fonction définit une expression, son appel est une expression.**

**Exemples :**

```
Function signe(x :integer) return integer is
Begin
   if  x > 0     then     return +1;
   elsif x < 0 then     return -1;
   else              return 0;
   end if;
End signe;
```

Appel : y := signe(12);

```
Function abs(x : integer) return positive is
Begin
   return signe(x) * x;
End abs;
```

Appel : y := abs(-5);

- **Paramètres formels et effectifs**

## 16.2- Procédures

- Une procédure ne retourne pas de résultat.
- L'appel d'une procédure se fait par la commande

*nom(pe1,...,pen);*

Les $pe_i$ sont les paramètres **effectifs**.

**Exemple :**

```
type vecteur is array (1..10) of integer;
notes : vecteur;

Procedure trier (taille : integer; v : in out vecteur) is
Begin
   -- on trie le vecteur   v(1..taille)
End trier;
```

**Appel :**

```
-- remplissage initial du tableau notes
trier(6, notes);
-- retour de la procédure
```

*6* et *notes* sont les paramètres **effectifs**
Le paramètre *taille* récupère la valeur 6
Le tableau *notes* est copié dans la variable *v*
La procédure *trier* trie ce tableau
Le tableau trié *v* est recopié dans *notes*

# 17- Définitions de procédures et de fonctions

## 17.1- Fonctions

```
Function nom (pf1;...; pfn) return Type_de_la_fonction is
-- déclarations locales
Begin
   ....return ....
End nom;
```

**Appel :**

```
variable := nom(pe1,..., pen) ;
```

**Exemple :**

```
Function signe(x : integer) return integer is
Begin
   If  x > 0 Then    return (+1);
   Else If x < 0 Then     return (-1);
         Else            return (0);
         End if;
   End if;
End signe;
```

Appel :

```
y := signe(12);
```

## 17.2- Procédures

```
Procedure nom (pf1;...; pfn) is
-- déclarations locales
Begin
....
End nom;
```

L'appel d'une procédure se fait par la commande

```
nom(pe1,...,pen);
```

## 17.3- Paramètres

A chaque paramètre formel est associé un type et un mode:

- Le type est au sens habituel;
- Le mode est :
  - IN (C'est le mode par défaut)
  - OUT
  - IN OUT

**Remarques :**

- Tous les paramètres des fonctions sont en mode IN
- Ne pas modifier les paramètres IN
- Ne faire que des affectation dans les paramètres OUT
- On peut tout faire avec les paramètres IN OUT
- ADA permet de définir des valeurs par défaut des paramètres
- ADA permet de redéfinir des opérateurs (par une fonction)

## 17.4- Echappement dans les abstractions

- Dans le cas d'une Procédure : *Return*
  ➥ abandon de la procédure et retour à l'appelant;

- Dans le cas d'une Fonction : *Return (Expr)*
  ➥ abandon de la fonction, retour à l'appelant avec la transmission de la valeur *Expr* .

**Exemples** :

```
procedure p(m,n : IN integer) is
Begin
   if m > n then return; end if;
   put(m); put(n);
End p;
```

```
function f(m,n : IN integer) is
    Begin
      if m > n then return true;
      else return false;
      end if;
    End f;
```

On peut également écrire :

```
function f(m,n : IN integer) is
Begin
   return (m > n);
End f;
```

## 17.5- Mécanismes d'appel et de retour de sous-programmes

**Mise en correspondance des paramètres effectifs et formels.**

**A l'appel :**

- Mise en correspondance des paramètres IN ou IN OUT.
- Les paramètres en mode OUT ne reçoivent aucune valeur.

**Pendant l'exécution :**

- Les paramètres formels en mode IN sont des constantes, ils ne peuvent être accédés qu'en lecture.
- Les paramètres en mode OUT sont des variables; leur valeur n'est pas accessible, on peut seulement leur affecter une valeur.
- Les paramètres formels en mode IN OUT sont des variables à part entière, ils sont accessibles en lecture et en écriture.

**A la fin de l'exécution :**

- Passage de la valeur des paramètres formels dont le mode est OUT ou IN OUT dans les paramètres effectifs correspondant.
- Les paramères en mode IN n'auront pas été modifiés.

## La fin de l'exécution :

La fin de l'exécution d'un sous-programme intervient

• Pour une **procédure** :
- lorsque la dernière instruction du corps a été exécutéé (END)
- lorsque l'instruction RETURN est exécutée
- lorsqu'une exception a été déclenchée

• Pour une **fonction** :
- lorsque l'instruction RETURN est exécutée. Le corps de la fonction doit comporter au moins une instruction RETURN suivie d'une expression donnant un résultat dont le type est celui spécifié dans l'entête de la fonction. Cette expression est la **valeur** de la fonction.
- lorsqu'une exception a été déclenchée

## *Exemple*

```
procedure mult(m,n : IN OUT integer) is
Begin
   m := m * n;
   put(m); put(n);
End mult;
```

Soient i=2 et j=3.

Quelles sont les valeurs affichées dans :

a- mult(i, j);
b- mult(i, i);
c- put(i); put(j); à la suite de (b)

**Réponses** : 6, 3
4, 2
4, 2

## 17.6- Surcharge dans les fonctions

• Additionner deux entiers ou deux réels par le même opérateur “+” est possible grâce à la surcharge de cet opérateur.

• ADA permet de réutiliser plusieurs fois un même nom d’abstraction.

• Pour lever l’ambiguïté sur la surcharge, le compilateur utilise :
- le nombre de paramètres effectifs,
- le type et l’ordre des paramètres effectifs,
- le nom des paramètres formels dans le cas d’une notation nominale,
- le type du résultat dans le cas d’une fonction

### *Exemple :*

Définir une fonction maximum avec des réels, des entiers, ...
L’appel à la “bonne” version est fait selon les types des paramètres :

```
Function maximum (op1,op2 : integer) return integer;
Function maximum (op1,op2 : float) return float;
Function maximum (op1,op2 : matrice) return matrice;
```

• En ADA, on peut également redéfinir des opérateurs pour les types utilisateurs.

### *Exemple :*

```
Function “+”(x,y : matrice) return matrice is      -- “+” surchargé
Begin
   -- l'addition des matrices x et y
end "+";
```

### *Utilisation*

```
m := m1 + m2;          -- utilisation sous forme infixée
m := “+”(m1,m2);       -- une autre manière : notation préfixée
```

**Remarques :**

- On peut surcharger les opérateurs prédéfinis d'ADA :

  **AND, OR, XOR, NOT, <, <=, > , >=**
  **+ , - , * , / , ** , MOD , REM**

- *La surcharge de l'affectation ":=" n'est pas possible.*
- *L'opérateur "/=" est toujours déduit de l'opérateur "=".*
- *La surcharge de l'opérateur "=" n'est admise que sur des opérandes de type limité privé pour lesquels cet opérateur n'est pas défini (voir plus loin).*

## 17.7- Valeurs par défaut des paramètres

Il est possible d'associer aux paramètres formels en mode IN des valeurs par défaut. L'évaluation de l'expression par défaut se fera à chaque appel.

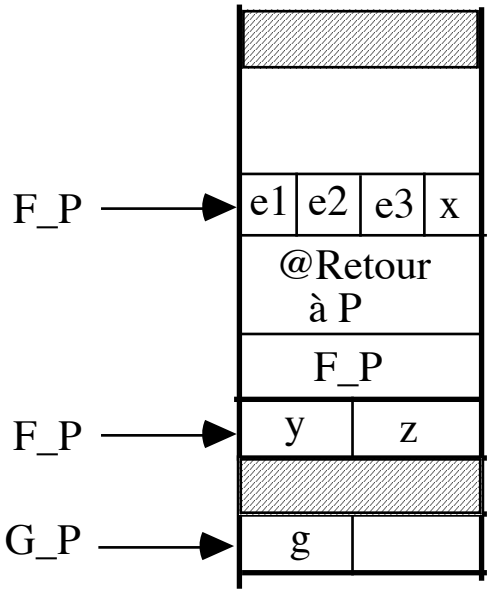
*Exemple :*

```
procedure servez_cafe( marque : string;
                       force_cafe : force := costaud;
                       temp_cafe : temperature := chaud;
                       option_cafe : option := sucre;
                       nombre_cc : quantite := 10);

servez_cafe("hot coffee");                -- prise des options par défaut
servez_cafe("jacques...", nombre_cc => 20);    -- café double
servez_cafe("black", force_cafe => leger, option_cafe => lait);
```

Les paramètres formels qui n'ont pas de valeur par défaut doivent avoir un paramètre effectif correspondant à l'appel.

# 17.8- Evolution de la pile lors d'appel de sous-programmes

```
Procedure principale is
g : integer := 5;        -- variable globale
   procdeure r(e1 : IN integer; e2 : OUT integer; e3 : IN OUT integer) is
   x : integer;
   Begin
      x := e1;
      e2 := e3 + x;
      e3 := g + 5;
   End r;

   procedure p is
      y,z : integer;
   Begin
      y := 3;  z := 6;
      r(9, y, z); ....
   End p;

Begin
   ...p; ...
End principale;
```




**Notions :**

- **Pile**
- **Empiler**
- **Dépiler**
- **pointeur global**
- **Pointeur local**
- **Saut à une adresse**
- **Adresse de retour**
- **Compteur de programme**

- ZZZ : Anticipation sur environnement & abstraction

# 18- Exercices

1- Calcul de la moyenne M de deux entiers X et Y.

- Sous forme de Procédure
- Sous forme de Fonction

2- Recherche d'un élément X dans un tableau de caractères T. Après cette recherche, on doit préciser si l'élément a été trouvé ou non.

3- Soit une matrice (éventuellement carrée) $M_{l,c}$ d'entiers.

A) Trouver les coordonnées (ligne et colonne) du premier élément de M tel que cet élément soit à la fois la maximum de sa ligne et le minimum de sa colonne.

B) Trouver les coordonnées (ligne et colonne) de tous les éléments de M tel que chacun de ces éléments soit à la fois la maximum de sa ligne et le minimum de sa colonne.

4- Calcul de la moyenne et la variance d'un série d'entiers par les méthodes tabulée et calculée.

5- Elimination des mots doubles consécutifs dans une phrase terminée par un '.'

# 18'- Solutions

1- Calcul de la moyenne M de deux entiers X et Y sous forme de Procédure et sous forme de Fonction :

```
Procedure moyenne (X,Y : integer; M : out integer) is
begin
   M := (X + Y) /2 ;
end moyenne;

Function moyenne (X,Y : integer) return integer is
begin
   return (X + Y) /2 ;
end moyenne;
```

2- Recherche d'un élément X dans un tableau de caractères T. Après cette recherche, on doit préciser si l'élément a été trouvé ou non.

```
          N : Constant := ...;
Type   Bornes is Range 1..N;
Type   TabCar is Array (Bornes) of character;

Function Recherche(X: character; M: Bornes; T: TabCar)
                    return boolean is
i : Integer := 1;
Begin
      While (i ≤ M) and then (T(i) /= X) loop
         i:=i + 1;
      End loop;
      -- (i>M & X ∉  T) ou (i≤M & T(i) = X)        Notion d'assertion

      Return (i <= M);
End Recherche;
```

Exemple d'utilisation :

```
 Existe : boolean;
-- On suppose que le tableau TAB est déjà rempli
Existe := Recherche('g', 10, TAB);
```

3- Soit une matrice (éventuellement carrée) $M_{l,c}$ d'entiers.

A) Trouver les coordonnées (ligne et colonne) du premier élément de M tel que cet élément soit à la fois la maximum de sa ligne et le minimum de sa colonne.

B) Trouver les coordonnées (ligne et colonne) de tous les éléments de M tel que chacun de ces éléments soit à la fois la maximum de sa ligne et le minimum de sa colonne.

```
with text_io;
procedure min_max is
package iio is new text_io.integer_io(integer);
type ligne is array(1..5) of integer;
type mat is array(1..5) of ligne;
m : mat := (  (1,3,2,115,4),
              (6,9,4,130,8),
              (3,11,5,230,7),
              (0,7,6,56,9),
              (43,4,7,120,11));
l,c : integer;
trouve : boolean := false;

procedure affiche is
begin
  for i in 1..5 loop
    for j in 1..5 loop iio.put(m(i)(j)); text_io.put(" "); end loop;
    text_io.new_line;
  end loop;
end;

procedure max_ligne(l : integer;c : out integer) is
i : integer := 2;
c1 : integer := 1;
begin
  while i <= 5 loop
    if m(l)(i) > m(l)(c1) then c1 := i; end if;
    i := i + 1;
  end loop;
```

```
  c := c1;
end;
function est_min_col(x,c : integer) return boolean  is
i : integer := 1;
begin
  while i <= 5 loop
    if m(i)(c) < x then return(false); end if;
    i := i + 1;
  end loop;
  return(true);
end;
begin
  affiche; l:=1;
  while l <= 5 loop
     max_ligne(l,c);
      if est_min_col(m(l)(c),c) then trouve := true; exit; end if;
     l := l+1;
  end loop;
  if trouve then
    iio.put(l); text_io.put(" x "); iio.put(c);
    text_io.new_line;
  end if;
end min_max;
```

4- Calcul de la moyenne et la variance d'un série d'entiers par les méthodes tabulée et calculée.

## 5- Elimination des mots doubles consécutifs dans une phrase terminée par un '.'

```
-- exemple de suppression des mots doubles dans une phrase
-- les mots separes par +rs blancs, un point a la fin de la phrase

with text_io; use text_io;
PROCEDURE doublons is
ph,ph1:string(1..300) := (others => ' ');

l,ind_ph1 : integer:=1;

Procedure ajoute(ph:in out string; ind : in out integer;
       mot : string; taille : integer) is
begin
    put_line("%"&ph(1..ind) & "%");
    put_line("%"& mot(1..taille)&"%");
    ph(ind..ind+taille) := mot(1..taille) & ' ';
    ind := ind+taille+1;
end ajoute;

FUNCTION egal(m1,m2 : string) return boolean is
begin
    if m1'last /= m2'last then return false;
    else return m1(1..m1'last)=m2(1..m2'last);
    end if;
end egal;

PROCEDURE prochain_mot(ph:string; i1: integer;
                mot : out string; taille : out integer; o : out integer) is
-- extraire le prochain mot en commencant a i et en s'arretant a o
m : string(1..26):=(others => ' ');
j : integer := 1;
i:integer := i1;
begin
  while ph(i) /= '.' and then ph(i)=' ' loop i:=i+1; end loop;
  while ph(i) /= '.' and then ph(i) /= ' ' loop
       m(j) := ph(i); j:=j+1; i:=i+1;
  end loop;
```

```
  mot(1..j) := m(1..j);
  o:=i;  -- indice de fin
  taille := j-1;    -- taille de mot
end prochain_mot;

PROCEDURE elimine(ph : string; ph1 : in out string;ind_ph1 : in out integer) is
mot_prec,mot_svt : string(1..26):=(others => ' ');
ind , ind1: integer := 1;
taille:integer;
begin
   prochain_mot(ph,1, mot_prec, taille, ind);
   put_line('<'&mot_prec(1..taille)&'>');
   ajoute(ph1, ind_ph1,mot_prec,taille);
   put_line("la phrase actuelle = " & ph1(1..ind_ph1));
   while ph(ind) /= '.' loop
      prochain_mot(ph,ind, mot_svt, taille,ind1);
      put_line('<'&mot_svt(1..taille)&'>');
      if not egal(mot_prec,mot_svt) then
         ajoute(ph1, ind_ph1,mot_svt,taille);
      end if;
   put_line("la phrase actuelle = " & ph1(1..ind_ph1));
   ind := ind1;
   end loop;
   ph1(ind_ph1-1) := '.';  -- dernier ' ' remplace
end elimine;

begin
   put("donner la phrase : ");
   get_line(ph,l);
   elimine(ph(1..l), ph1,ind_ph1);
   put_line(" le resultat est " & ph1(1..ind_ph1));
end doublons;
```

## Une autre solution utilisant les chaînes (tableaux) dynamiques

```
with text_io; use text_io;
PROCEDURE dd is
type chaine (t:natural:=0) is record
  ch : string(1..t) := (others=>' ');
end record;

ph:string(1..300) := (others => ' ');
ph1,ph2 : chaine;

l : integer:=1;

Procedure ajoute(ph:in out chaine; mot : chaine) is
begin
     ph:=(ph.t+mot.t+1, ph.ch(1..ph.t)&mot.ch(1..mot.t)&' ');
end ajoute;

FUNCTION egal(m1,m2 : chaine) return boolean is
begin
     if m1.t /= m2.t then return false;
     else return m1.ch(1..m1.t) = m2.ch(1..m2.t);
     end if;
end egal;

PROCEDURE prochain_mot(ph:chaine; i1:integer;
            mot : out chaine; o : out integer) is

-- extraire le prochain mot en commencant a i et en s'arretant a o
m : string(1..26):=(others => ' ');
j : integer := 1;
i:integer := i1;
begin
 while ph.ch(i) /= '.' and then ph.ch(i)=' ' loop i:=i+1; end loop;
 while ph.ch(i) /= '.' and then ph.ch(i) /= ' ' loop
        m(j) := ph.ch(i);
        j:=j+1; i:=i+1;
 end loop;
```

```
  mot := (j-1,m(1..j-1));
  o:=i;  -- indice de fin
end prochain_mot;

PROCEDURE elimine(ph : chaine; ph1 : in out chaine ) is
mot_prec,mot_svt : chaine;
ind , ind1: integer := 1;
begin
   prochain_mot(ph,1, mot_prec, ind);
   ajoute(ph1, mot_prec);
   while ph.ch(ind) /= '.' loop
      prochain_mot(ph,ind, mot_svt,ind1);
      if not egal(mot_prec,mot_svt) then
         ajoute(ph1, mot_svt);
      end if;
   ind := ind1;mot_prec :=(mot_svt.t,mot_svt.ch);
   end loop;
   ph1:=(ph1.t,ph1.ch(1..ph1.t-1)& '.');  -- dernier ' ' remplace
end elimine;

begin
   put("donner la phrase (terminee par un point) : ");
   get_line(ph,l);
   if ph(l) /='.' then
      put_line("manque le point, bye"); return;
   end if;
   ph1:=(l,ph(1..l));
   elimine(ph1, ph2);
   put_line(" le resultat est : " & ph2.ch(1..ph2.t));
end dd;
```

# 19- Définition de tableaux dynamiques en ADA

## 19.1- A l'aide d'une procédure

**Rappel** :

- Habituellement, les bornes d'un tableau ADA doivent être connues lors de l'élaboration du type (exécution).
- Une fois évalué, ces bornes restent fixes.

- On peut avoir des tableaux "dynamiques" comme dans :

```
Procedure tab_dyn (N : integer) is
        TYPE tabvarie IS ARRAY (1..N) OF integer;
                .....
```

- Le type *tabvarie* est défini dans l'environnement associé à *tab_dyn*.

## 19.2- A l'aide d'un enregistrement à champ variable

```
Type chaine_dyn(l : integer) is record
            ch : string(1..l);
      end record;

c : chaine_dyn(5);
d : chaine_dyn(10) := (10,"azertyuiop");
e : chaine_dyn(6) := (3,"abc");
```

*Mais dans*

```
f : chaine_dyn:= (3,"abc");          --  non car la taille est absente
```

# 20- Abstractions en CAML

• La seule abstraction en CAML est la fonction (pas de procédure).

• Les fonctions CAML possèdent toutes les propriétés des fonctions ADA. En outre, elles sont plus puissantes et plus expressives.

• Une fonction CAML est une abstraction qui enferme une expression à évaluer. Cette expression est évaluée :
- partiellement à la définition (le typage)
- totalement lorsque l'abstraction est appelée.

• CAML calcule le type (le plus général) des fonctions

• Tous les paramètres d'une fonction CAML sont en entrées
➥ Les paramètres d'une fonction CAML sont non modifiables.

• Une fonction CAML implante la définition mathématique de la fonction :
*fonction f : param1 x ... x paramn--> un seul résultat*

## 20.1- Définition d'une fonction CAML

Deux manières :

let **f** x y z= CORPS ... *ou*

let **f** = function x -> function y -> function z -> CORPS

x, y et z sont les paramètres formels.

• On peut mettre les paramètres entre parenthèses.
• S'il y a un seul paramètre pour f, *f x* est équivalent à *f(x)*.
• Si f possède plus d'un paramètre, il n'y a plus d'équivalence :

**let f(x,y) =** est différent de
**let f x y =** équivalente à **let f = fun x -> fun y -> corps_de_f**

## 20.2- Appel d'une fonction en CAML

Une fonction dont la définition est de la forme

*let nom* $pf_1 \ldots pf_n$ = .....

est utilisée sous la forme d'une expression : *(nom* $pe_1 \ldots pe_n$*) ;*

Les *pei* sont les paramètres effectifs sont tous en entrée et non modifiables.

## 20.3- Exemples de fonctions en CAML

### 1- Cas simples

```
#let carre(x) = x*x;;                                    (* int -> int *)
#let somme_des_carres x y = carre(x) + carre(y);;        (* int -> int *)
#let val_abs = function x -> if x>0 then x else -x;;     (* int -> int *)
```

Appels :

```
#carre 3;;                              (* ou carre(3) *)
#somme_des_carres 3 4;;        (* mais pas somme_des_carres(3,4) *)
#(val_abs 3 = val_abs (-3));;
```

### 2- l'égalité générique (polymorphe)

```
#let egal = function x y -> if x=y then true else false;
```

### 3- gestion des comptes

```
#type compte = {num : int; mutable solde : int};;
Type compte defined.

#mon_compte : compte = {num=1234; solde = 2000};;
#let moncompte = {num=1234; solde = 2000};;

# let depot compte montant =
      compte.solde <- compte.solde + montant;;
depot : compte -> int -> unit = <fun>

#depot moncompte 3000;
- : unit = ()
# moncompte.solde;;
-: int = 5000
```

## 20.4- Traitement des fonctions CAML

- Les expressions figurant dans une fonction sont évaluées selon l'environnement.

```
#let x=1;;
x : int = 1
#let f= if x=1 then true else false;;
f : bool = true
#f;;
- : bool = true
```

p f n'est pas une fonction.

```
#let g() = if x=1 then true else false;;
g : unit -> bool = <fun>
```

p g est pas une fonction.

```
#g();;
- : bool = true
```

## *Exemple : Impression des chiffres*

```
(* une expression *)
#let imprime  =
   for i=0 to 9 do
         print_int i
   done;
   print_newline;;

0123456789
imprime : unit -> unit = <fun>

#imprime;;
- : unit -> unit = <fun>
```

```
(* une fonction sans paramètre *)
   #let imprime () =
   for i=0 to 9 do
      print_int i
      done;
print_new_line;;

   imprime : unit -> unit = <fun>


   # imprime;;
0123456789
   -: unit ()
```

## *20.4.1-Evaluation des fonctions contenant des références*

```
#let x=1;;
x : int = 1
#let f y = x+y;;
f : int -> int = <fun>
#let x= 2;;
x : int = 2
#f(0);;
- : int = 1
```

```
#let x=ref 1;;
x: int ref = ref 1
#let f y = !x+y;;
f : int -> int = <fun>
#x := 2;;
- : unit = ()
   #f(0);;
- : int = 2
```

A gauche, la modification de **x** ne modifie rien dans le corps de la fonction **f** qui a été évalué avec **x=1**.

A droite, **x** est lié à une référence. La valeur de **!x** change dans le corps de **f** après sa modification (cependant, **x** est toujours lié à la même valeur : la référence de x)

➥ les fonctions utilisant des références non locales peuvent changer dynamiquement de comportement.

## *20.4.2-Définition locale : le mot clef in*

```
#let  doubler_chaine s =
   let l = string_length s in
   begin   print_string ("la longeur de la chaine " ^s ^s^" est : ");
           print_int (2*l);      -- paranthésage obligatoire
              print_newline()
   end;;
f : string -> unit = <fun>

#f "bla ";;
la longeur de la chaine bla bla  est : 8
- : unit = ()
```

# *Exemple : Fonction factorielle en CAML*

```
#let fact n =
   if n=0 then 1 else
      begin
         let accu = ref 1 in
            for i=1 to n do accu := i * !accu done;
            !accu
         end;;
fact : int -> int = <fun>
#fact 10;;
-: int = 3628800
```

**Simplification** : le test n=0 est inutile

```
#let fact n =
         let accu = ref 1 in
            for i=1 to n do accu := i * !accu done;
            !accu;;
```

## *Remarques :*

```
(* -------- factorielle "normale en ADA" *)
#let rec factrec n =  if n <= 1 then 1  else n * factrec (n-1);;
factrec : int -> int  = <fun>

(*  Aure solution plus efficace : m représente (n-1) ! *)
#let rec fact1 n m =     if n <= 1 then m else fact1 (n-1) (n*m);;
fact1 : int -> int -> int = <fun>
#fact1 3 1;;              (* 1 est X!, X <= 1 *)
- : int = 6

(* donc on peut écrire *)
#let fact2 n = fact1 n 1;;
fact2 : int -> int = <fun>
#fact2 4;;
- : int = 24
```

(*---- ou encre ---- *)

```
#let fact3 n =
    let rec fac x y = if x <= 1 then y else fac (x-1) (x*y) in
    fac n 1;;
fact3 : int -> int = <fun>
#fact3 5;;
- : int = 120
```

## *20.4.3- Données mutables en paramètre de fonctions CAML*

- On peut modifier les données mutables en paramètres

```
#let f x = x := !x + 1;;
f : int ref -> unit = <fun>

#let s = ref 2;;
s : int ref = ref 2

#f s;;
- : unit = ()

#s;;
- : int ref = ref 3

#let g (v : int vect) = v.(0) <- 1;;
g : int vect -> unit = <fun>

#let d = make_vect 5 3;;
d : int vect = [|3; 3; 3; 3; 3|]

#g d;;
- : unit = ()

#d;;
- : int vect = [|1; 3; 3; 3; 3|]
```

## 20.5- Filtrage en CAML

### Le concept

- Ressemble à un case
- Permet une sélection par filtrage des paramètres
- Impose la seconde forme des fonction :

```
let f = function
      val1_de_param      -> expr
    | val2_de_param      -> expr
    |        _           -> ....
```

### Exemples :

```
#let bool_a_ent = fun
   true -> 1
   | false -> 0;;

bool_a_ent : bool -> int = <fun>

#bool_a_ent  true;;
- : int = 1

#let valeur = fun
1 -> "vrai"
| _ -> "faux";;
valeur  : int -> string = <fun>

#valeur 2;;
- : string = "faux"
```

# 21- Valeur rendue par une fonction

- Une fonction est une expression plus ou moins complexe
- L'appel d'une fonction représente une fonction

➥ Que peuvent rendre les fonctions dans les langages de programmation ?

## 21.1- Principe de complétude de type

Aucune opération (y compris les fonctions) ne doit être arbitrairement restreinte à certains types de valeurs impliquées.

➥ Régularité des langages

## 21.2- Complétude de type en ADA et CAML

### *ADA :*

- Une fonction ADA ne peut pas rendre une fonction.

### *CAML :*

- Le résultat d'une fonction CAML peut être n'importe quel objet, y compris une autre fonction :

```
#let f= function x ->
      if (x>1)
      then (function d -> d+1)
      else (function d-> d-1);;
f : int -> int -> int = <fun>

#f 1 ;;              (* appliquer f à 1 ==> la fonction "d -> d -1" *)
- : int -> int = <fun>

#(f 1) 2;;       (* appliquer "d -> d-1" à 2 ==> la valeur 1 *)
- : int = 1
```

## 21.3- Agrégat au retour de fonction (CAML et ADA)

- **Rappel : utilisation dans les initialisations**
- **Cas de retour d'une fonction**

Un exemple de fonction qui met à zéro tous les éléments d'un tableau :

```
Function raz return tab is
begin
   return (OTHERS => 0);
end raz;
```

Utilisation : tab1 := raz;

*Même principe en CAML pour renvoyer des types composés.*

```
#let creer_compte = fun n -> fun s ->
   {num = n; solde = s};;
creer_compte : int -> int -> compte = <fun>

#let ton_compte = creer_compte 2345 23000;;
ton_compte : compte = {num=2345; solde=23000}
```

# 22- Equivalence fonction-tableau

- *Rappel du concept d'association (mapping)*
  Tableaux & fonctions

  Notation : $\tau_1$ |–> $\tau_2$

**Tableaux**

```
type Tab_mult is array (1..10, 1..10) of integer;
-- remplir une variable (tab) du type Tab_mult avec les
-- valeurs de la table de multiplication 1..10
```

*tab* définit l'application {1..10}x{1..10} |–> integer

{(1,1) |-> 1, (1,2) |-> 2, (1,3)|-> 3, ..., (9x9) |-> 81, ...}

tab(4,5) donne 4 x 5 = 20 : **valeur stockée**

## *Fonctions*

```
Function tab (m,n:integer) return integer is
Begin
   return m*n;
End fact;
```

L'application *tab* définie est "integer x integer |–> integer" :

{(1,1) |-> 1, (1,2) |-> 2, (1,3)|-> 3, ..., (9x9) |-> 81, ...}

tab(4,5) donne 4 x 5 = 20 : **valeur calculée**

# 23- Environnement

**Rappels :**

- Un environnement est un ensemble d'associations de couples [identificateur |→ valeur].
- L'environnement intervient dans l'interprétation d'expressions.
- Chaque expression ou commande d'un programme est interprétée dans un environnement particulier.
- Une expression peut avoir différentes interprétations dans différentes parties d'un programme.
- En général, dans un environnement donné, il existe une seule association pour chaque identificateur.

## *Exemple ADA*

```
procedure P is
   z : constant := 0;
   c : character;
begin
①.....;
end P;
```

L'environnement au point ① est

{ c |→ une variable caractère
  z |→ l'entier 0
  p |→ la procédure }

## *Exemple CAML*

```
let y = 0;;
let f x = ②  if x>y then 1 else 0;;
f : int -> int = <fun>
```

L'environnement au point ② est

{ x |→ une variable locale
  y |→ l'entier 0 }

# 23.1- Elaboration d'environnements

# 23.2- Définitions et déclarations

L'élaboration des environnements se fait par des définitions et des déclarations.

## Exemples de définitions

```
pi : constant := 3.14;                    -- pi représente 3.14
alpha : constant character := 'a';
type car is character;
function signe(s : integer) return boolean is ...
procedure p (a : tab) is ....

let  x = 1 ;;                -- CAML
let a = ref 12;;          -- CAML
```

## Exemples de déclarations

```
type octet is array (1..8) of boolean;
type vecteur is array (1..5) of integer;
type ligne is new integer range 1..66;
type nom_de_mois is (jan, fev,...., dec);
type date is record
              jour : integer range 1..31;
              mois : nom_de_mois;
              annee : integer;
          end record;

d : date := (3,jan,1994);
x : integer;
memoire : array (1..64000) of octet;

type livre = {titre : string; auteur : string};     -- CAML
```

Il existe également des définitions / déclarations récursives

Une déclaration récursive utilise le lien qu'elle-même crée.

ADA et CAML permettent cette sorte de définitions / déclarations

# 24- Environnement et abstraction

**Exemple :**

```
procedure P is
   z : constant := 0;
   c : character;
   procedure Q is
      c : constant := 3.15;
      b : boolean;
   begin
   ② .....
   end Q;
begin
①.....;
end P;
```

L'environnement au point ① est

**{ c l→ une variable caractère**
**Q l→ une procédure, z l→ l'entier 0, P l→ une procédure }**

L'environnement au point ② est

**{ b l→ une variable booléenne**
**c l→ le nombre réel 3.15, Q l→ une procédure, z l→ l'entier 0 }**

En CAML :

```
let succ x = x+1 ;;
let incr= function x -> x+1;;
```

# 25- Visibilité

C'est la portée d'une déclaration.
C'est l'accessibilité de la valeur d'un identificateur.

On dit qu'un identificateur est "visible" dans tel ou tel *bloc*

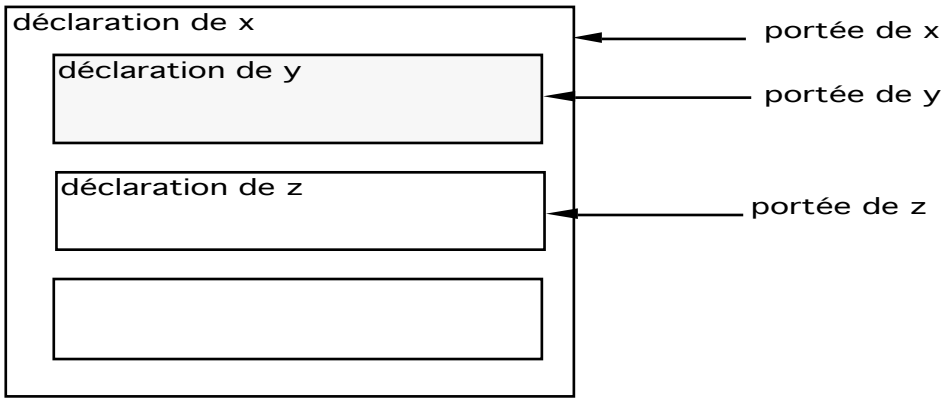
## 25.1- Notion de bloc

Un bloc est une séquence de phrases du programme.

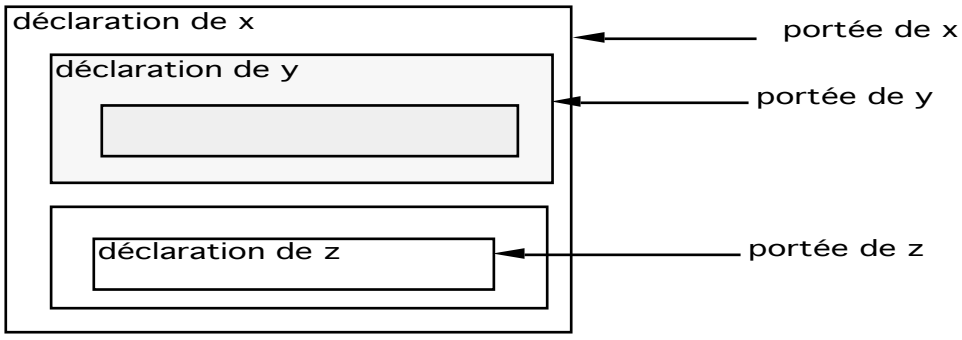
Un bloc délimite la portée des identificateurs qu'il définit.

- ➥ langages à structure de bloc
- ➥ Environnement associé à un bloc
- ➥ imbrication de blocs


*monolithique (Cobol)*

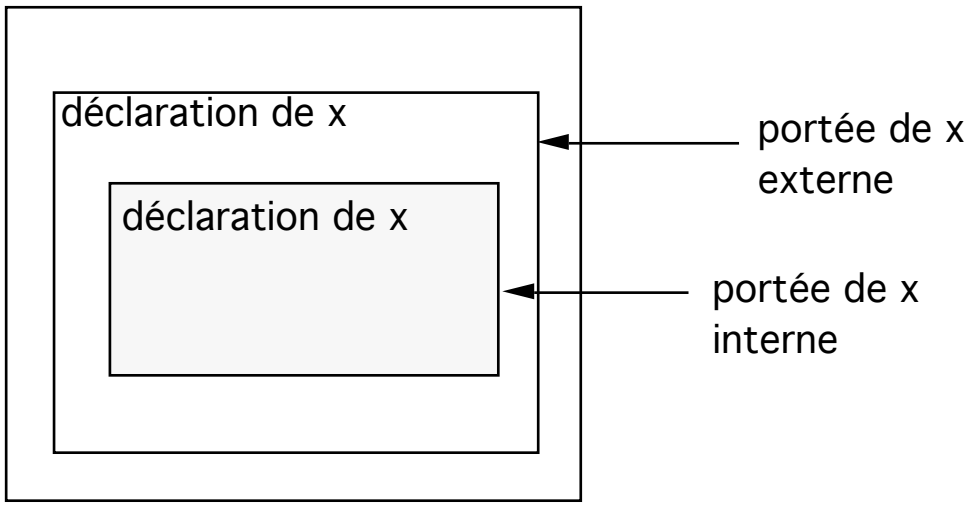



*plat (C, Fortran)*



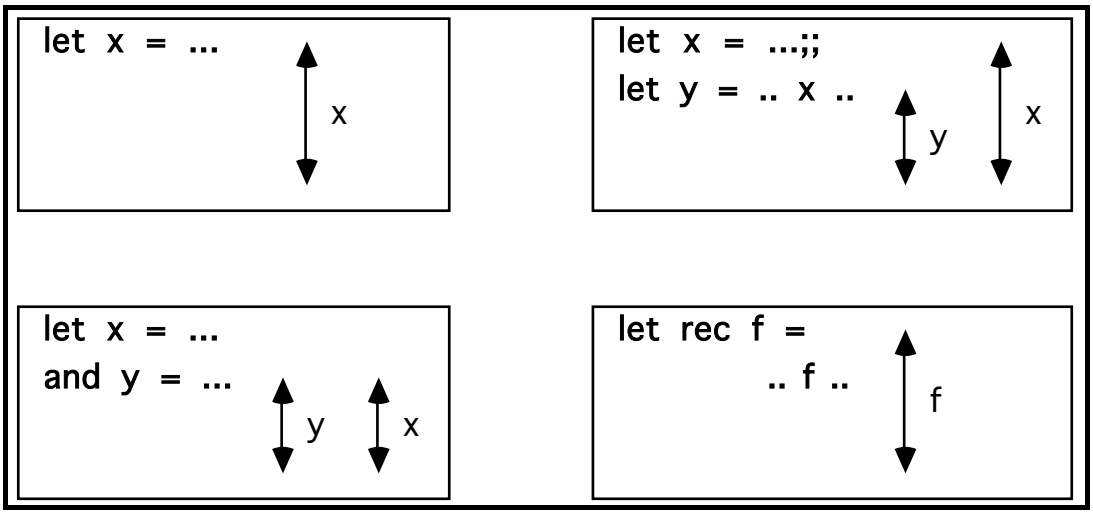

*imbriqué (Ada, Pascal, CAML)*

## 25.2- Règle de visibilité




```
procedure P is
   x : character;

   procedure Q is
      x : integer;
      b : boolean;

      procedure R is
         b : boolean;
         begin
         ③ x:=1; b:=true;
         end R;

      begin
      ② x:=3; b:=false;
      end Q;

   begin
   ① x:='a';
   end P;
```

## 25.3- Règles de visibilité CAML




## 25.4- Notion de durée de vie

intervalle entre la création et la destruction d'une variable

### *25.4.1- Variable locale et globale*

**ADA :**

```
procedure P is
   i : integer :=0;
   procedure Q is
      k: integer :=i;
      i : integer;
   begin
   ......
   end Q;
begin
.... Q ;...
end P;
```

**CAML :**

```
let a=5;;
let x=a+b  in  z = x*2;;
```

Les variables locales sont créés afin d'être utilisées localement;
Les variables globales sont créées pour être accessibles
dans tout le programme.

## 25.5- Exercice

Précisez les environnements à chacun des points ① à ⑧ dans le programme suivant :

```
   Procedure p is
      x : constant := 999;
①     type Nat is 0..x;
②     m,n : Nat;
③     function f(n:Nat) return Nat is
      begin
④       ....
      end f;
⑤     Procedure w(i:Nat) is
⑥       n : constant := 6;
      begin
⑦       ....
      end w;
   begin
⑧  ....
   end p;
```

## *Solution (sans les valeurs associées)*

1) {x}
2) {x, Nat}
3) {x, Nat, m , n}
4) {x, Nat, m , f, n}
5) {x, Nat, m , n, f}
6) {x, Nat, m , n, f, w , i}
7) {x, Nat, m , n, f, w , i}
8) {x, Nat, m , n, f, w}

# 26. Traitement des erreurs, ruptures

- Un programme ne se déroule pas toujours comme prévu,
- Plusieurs cas d'erreurs perturbateurs :
  Division par zéro,
  Dépassement des bornes d'un tableau,
  Débordement de la pile,
  Fichier inexistant,
  Erreur de type repérée à l'exécution...

- Traitement classique :
  Appel d'une procédure d'erreur
  Les difficultés pour rester dans le bloc
  Hiérarchisation
  Qui traite l'erreur : l'appelant ou l'appelé

- En cas d'erreur, on ne peut pas toujours provoquer un arrêt du programme :
  Dans un système embarqué, l'arrêt serait grave :
  fusée, avion, chaîne de production

  ==> Il faut donc être capable de tolérer les pannes.

## 26.1. Exception en ADA

- Un outil pour résoudre ce genre de problèmes.
- Déclenchement d'une exception par :
  le matériel (division par zéro);
  l'exécutif ADA ( erreur de fichier)
  le programme lui même

Dans le dernier cas, deux possibilités existent :
- les exceptions prédéfinies qui se déclenchent automatiquement
- les exception définies dans le programme déclenchées explicitement (par l'instruction RAISE).

Ce déclenchement peut avoir lieu pour signaler un cas non habituel (exceptionnel mais pas erroné).
- Notion de *traiteur* (récupérateur) d'exceptions
- Propagation hiérarchique

• Repropagation vers l'appelant (traitement + Raise)

```
PROCEDURE remplir(in out tab : tableau) IS
ch : string(1..15);
BEGIN
   LOOP
      ch := (others => ' ');
      put("donner une chaine (< 16 caracteres) ");
      get_line(ch,l);
      insere(ch, tab);
   END LOOP;
   EXCEPTION
      when END_ERROR => affiche(tab);
      when OTHERS => put_line("probleme de lecture ");
END remplir;
```

## Différents genres d'exceptions

- prédéfinie
- fournie par un paquetage prédéfini
- déclarée par l'utilisateur

## Les exceptions prédéfinies

- constraint_error
- data_error
- numeric_error
- program_error
- storage_error
- tasking_error

CONSTRAINT_ERROR :
- sortie d'un intervalle, d'une contrainte d'index, d'une contrainte de discriminant
- accès à un champ d'article inexistant (record non contraint)
- accès par un pointeur NULL

DATA_ERROR :
- erreur de type de donnée lue, ...

NUMERIC_ERROR :
- résultat d'une opération prédéfinie hors de l'intervalle numérique de la machine (en particulier division par zéro)

PROGRAM_ERROR :
- en particulier lors qu'un appel à un sous-programme survient alors que le corps de celui-ci n'a pas été élaboré

- STORAGE_EROR :
manque de place lors de l'élaboration des déclarations;
d'une demande d'espace mémoire par un allocateur

- TASKING_ERROR : erreur de tâches.

## Exception fournies par un paquetage

C'est par exemple le cas de l'exception TIME_ERROR du paquetage CALENDAR, ou les exceptions liées aux entrées/sorties sur les fichiers dont les déclarations sont faites dans le paquetage IO_EXCEPTIONS.

## Les exceptions déclarées par l'utilisateur

Forme générale :

***identificateurs : EXCEPTION;***

```
interblocage : EXCEPTION;
pile_vide, pile_pleine : EXCEPTION;
```

## Traitement des exceptions

- Section EXCEPTION
- Le traitement des exception est séquentiel.
- Une exception NE peut être traitée plusieurs fois.
- La clause OTHERS ne peut apparaître que dans le dernier récupérateur. • Elle permet de récupérer n'importe quelle exception.

Exemple :

```
BEGIN
.....
```

```
EXCEPTION
   WHEN pile_pleine | storage_error =>
      put_line("la pile est pleine");
   WHEN pile_vide =>
      put_line("la pile est vide");

   WHEN constraint_error | data_error =>
      put_line("problème de contrainte");
      RAISE constraint_error;
      -- ici, on redéclenche pour l'unité appelant
   WHEN OTHERS =>
      put_line("une exception non prévue");
      un_param_de_sortie := une_valeur_spéciale;
      -- ici par exemple, on positionne une variable
End unité;
```

Certaine de ces exceptions peuvent avoir été déclenchées ou propagées à partir des sous-programmes manipulés.

## Déclenchement et Propagation d'une exception

❒ **explicite** : effectué à l'aide de l'instruction RAISE

Exemple :

```
IF index_pile = 0 THEN RAISE pile_vide;
Elsif index_pile = max_pile THEN RAISE pile_pleine;
End;
```

Il est possible de déclencher explicitement n'importe quelle exception, prédéfinie ou non.

❒ **implicite** (cas d'exception prédéfinie)

1- Si un traiteur est prévu, on l'exécute puis on quitte le bloc (unité). On peut signaler l'anomalie à l'appelant par un paramètre.

2- Si pas de traiteur, le bloc courant le bloc (unité) est terminé et l'exception est propagée à l'unité appelante, etc...
S'il n'y a aucun récupérateur, la propagation arrive au programme principal et l'exécution du programme s'arrête avec un message.
Si l'une des unités appelant est une tâche, il n'y a pas de propagation au-dehors de la tâche. Celle-ci s'arrête et le reste du programme continue de s'exécuter.

Remarques :

• Après une exception DATA_ERROR pendant la lecture d'une valeur, la valeur lue reste disponible et peut être lue sous un autre type à la prochaine lecture. L'instruction SKIP_LINE permet d'ignorer cette valeur.

• Il est possible dans un récupérateur d'exception de modifier les paramètres formels de mode OUT et IN OUT d'un sous-programme pour signaler les modifications éventuelles des données. On peut également exécuter l'instruction RETURN.

```
Procedure P(...;  bien_passe : out boolean) is
Begin
   ....
   bien_passe := true;        -- la procédure s'est bien exécutée
Exception
   When OTHERS =>
      bien_passe := false;         -- y a eu un problème
End P;
```

## 26.2. Exceptions en CAML

- Même principe qu'en ADA
- Il exist des exceptions prédéfinies en CAML :

  **Failure, Division_by_zero, End_of_file, Exit,....**

- La déclaration d'une exception en CAM :

| exception <nom> [of <type>] |
|---|

```
# exception table_vide ;;
exception table_vide defined.

# exception trouve of int;;       (* lors de la levée, un entier est transmis*)
exception trouve defined.

# trouve 5;;
-: exn
-= trouve 5
```

- Levée une exception en CAML par le mot clé

| **raise** |
|---|

**1- Exemple** : vérifier qu'une chaîne commence par un caractère particulier :

```
#exception chaine_vide;;
Exception chaine_vide defined.

#let commence_par chaine car =
   if string_length chaine = 0
   then (raise chaine_vide; false)       (* false étant donné le type *)
   else if nth_char chaine 0 = car   (* de la fonction commence_par *)
            then true
            else false;;

commence_par : string -> char -> bool = <fun>

#commence_par "abc" `a`;;
```

```
- : bool = true
#commence_par "abc" `k`;;
- : bool = false
#commence_par "" `a`;;
Uncaught exception: chaine_vide
```

L'exception n'est pas récupérée.

• La récupération :

| try <expr> with <traitement> |
|---|

```
#try
   commence_par "" `z`
with chaine_vide ->
   (print_string "la chaine est vide"; false);;

la chaine est vide- : bool = false
```

## 2- Une autre méthode :

Déclarer l'exception avec un type :

```
#exception chaine_vide of bool;;
Exception chaine_vide defined.

#let commence_par chaine car =
   if string_length chaine = 0 then raise (chaine_vide false)
   else     if nth_char chaine 0 = car then true
               else false;;
commence_par : string -> char -> bool = <fun>

#try
   commence_par "" `e`
 with chaine_vide b ->
       (print_string "absent" ; b);;
absent - : bool = false
```

• L'utilisation de l'exception prédéfinie *Failure* est simplifiée par :

```
#exception Failure of string;;
#let failwith s = raise (Failure s);;       -- "failwith" est prédéfinie
```

## 3- Exemple d'utilisation de Failure:

```
#let commence_par s c =
   if string_length s = 0 then failwith "la chaine vide"
   else if nth_char s 0 = c then true
   else false;;
commence_par : string -> char -> bool = <fun>

#try commence_par "" `z`
   with Failure p -> (print_string p; false);;
la chaine vide- : bool = false
```

Utilisation de l'exception Exit dans les boucles (une exception peut être utilisée après sa déclaration. Exit n'a pas un effet particulier sur les boucles, c'est une exception comme une autres que l'on aurait pu utiliser dans une boucle pour sortir.

```
try
   while true do
      print_int x;
      raise Exit
   done
with Exit -> print_string "rrr"
;;
2rrr- : unit = ()
```

# 27. Quelques règles méthodologiques *(d'expression des algorithmes)*

✍ Un algorithme doit être présenté de manière *lisible* pour être compréhensible.

• Exemple :Version difficile à lire :

```
i:=0; S:=0; While (i > n) loop
i:=i+1; If T(i) > 0 Then
S:=S+T(i); Else S:=S-T(i); End if; End Loop;
```

• Version plus lisible ? :

```
i:=0; S:=0;
While (i > n)  Loop
   i:=i+1;
   If      T(i) > 0
   Then    S:=S+T(i);
   Else S:=S-T(i);    -- T(i) est négatif, -T(i) positif
   End if;
End loop;
```

③ utiliser des *commentaires* pour rendre un algorithme plus lisible et pour préciser ce qu'il calcule :

*-- la somme des valeurs absolues des éléments...*
*(* somme des $|T_i|$ *)*
*(* $\Sigma_{i:0..n}|T_i|$ *)*

✍ Si l'algorithme décrit est trop long, il faut le décomposer en sous-algorithmes (module).
Dans ce cas, l'interface de chaque sous-algorithme doit décrire précisément :

- le rôle du module : sa *spécification* ;
- les variables d'entrée-sorties;
- les variables globales utilisées et/ou modifiées;
- le cas échéant, les liaisons entre les modules.

✍ Eviter des formes telles que les *branchement* (goto) ;

Il est toujours possible de programmer sans branchement en utilisant uniquement les schémas de boucle (Tant que, Pour), les conditionnelles (Si ... Alors ... Sinon ...) et les abstractions (procédures et fonctions).

Cependant, dans certains cas et pour clarifier les algorithmes, on peut utiliser des mécanismes d'échappement :

- EXIT
- RETURN
- Raise (EXCEPTION)

✍ Bien spécifier le problème résolu par l'algorithme.
(on a trop souvent tendance à vouloir qu'un algorithme contienne sa propre spécification !!)

**Exemple-1 :**

- Une spécification <u>informelle</u>

***-- ce programme calcule la somme S des valeurs absolues***
***-- des éléments du vecteur T. Le vecteur T peut être vide***
***-- auquel cas S sera égale à zéro.***

- Une spécification <u>formelle</u>

$$\text{-- } S = \sum_{i:0..n} |T_i|$$

D'autres formes de spécification formelles existent.
(spécification par le calcul des prédicats...)

## Exemple-2 (spécification informelle):

```
-- Ce programme recherche la place d'un élément X dans une liste L.
-- Si X n'est pas dans la liste le résultat est zéro. La liste est
-- représentée par un tableau. Elle comporte n éléments }
    ....
    j := 1;
    While (j <= n) AND (L(j) /= X)
    Loop
       j:=j+1;
    End loop;
    If j > n Then j:=0; End if ;
```

p La spécification donnée ci-dessus ne précise pas tout :

- Est-ce que la liste L peut être vide?
- Que fait-on s'il y a plusieurs occurrences de X dans la liste L?
- Quelles sont les données et les résultats?

③ Une meilleure spécification (plus précise) pour l'exemple précédent :

```
-- Ce programme recherche la place d'un élément X dans une liste L.
-- La liste est représentée par un tableau. Elle comporte n éléments.
-- Cette liste peut être vide (n=0).
-- On recherche l'indice (j) de la première occurrence de X dans L.
-- Si X n'est pas dans la liste, j sera égal à zéro. }
```

• Une spécification pour le même exemple:

$Card(L) = n,\ n \geq 0$

$X \notin L \Rightarrow j = 0$

$X \in L \Rightarrow (X = L_k)\ \&\ (\forall i \in [1,k-1]\ X \neq L_i) \Rightarrow j = k$

✍ D'autres règles et remarques:

- Une fonction doit calculer une seule valeur. Ce qui correspond davantage à sa définition mathématique.
- Veiller à ce que les paramètres d'une fonction soient tous en mode *entrée*.
- S'il vous faut calculer plusieurs valeurs par une abstraction, écrivez des procédures.
- Ne pas oublier la commande RETURN dans les fonctions.
- Il n'y a pas de distinction entre les minuscules / majuscules.
- Utiliser les Majuscules pour mettre les identificateurs importants en avant.

## 27.1. Règles de décomposition en sous-algorithmes

- Décomposition en Procédures et Fonction (abstraction)
- En procédures (paquets de commandes) chacune réalisant une commande complexe
- En fonction pour évaluer une expression complexe
- Réutilisation de ces abstractions
- Regroupement d'un ensemble homogène d'actions à isoler

p Chaque composant doit avoir une interface bien définie; il sera testé séparément.

# 28. Exercices

**Pour ces exercices, respecter les points suivants :**

- **Analyser chaque exercice, spécifier votre méthode**
- **Etudier la complexité de la solution retenue**
- **Utiliser des exceptions**
- **Pour certains exercices (au choix) proposer une solution en CAML.**
- **Respecter les règles méthodologiques**

**On reprendra quelques un de ces exercice après le chapitre sur la récursivité.**

1- Min Max d'une matrice (solution donnée plus haut).

2- Mots doubles consécutifs à éliminer dans une phrase terminée par un point. (solution donnée plus haut).

3- Exemple du p.g.c.d de deux nombres.

4- Ecrire la fonction factorielle (itérative).

5- Lecture, remplissage et écriture d'un tableau d'entiers T(1..N) sachant que ces opérations doivent être effectuées élément par élément.

==> Faire deux Procédures.

6- Recherche du plus grand élément d'un tableau d'entiers T. Un indice doit préciser le rang de cet élément. Le tableau T est supposé non vide.

7- La même chose qu'en 4 mais pour un tableau de mots.
Les chaînes de caractères sont comparables entre elles.

8 - recherche du rang du plus petit élément d'un tableau d'entiers. Le tableau T est supposé non vide.

9- Comptage du nombre d'occurrences d'un caractère donné dans une phrase. La phrase est une chaîne de caractères.

10- définir si un mot est un Palindrome (radar, tôt, sus). On suppose donnée une fonction Longueur(Ch) qui donne la longueur de la chaîne Ch.

11- Compter le nombre de mots dans une phrase (le blanc est le seul séparateur).

12- Recherche du mot le plus long dans une phrase.

13- calcul des occurrences de tous les caractères de l'alphabet dans une phrase. Le résultat sera rangé dans un tableau d'entiers.

14- Ecrire la procédure de multiplication de deux matrices M1 et M2 de dimension (N x N, N<5) d'entiers.

15- Recherche dans un tableau non ordonné : on recherche l'élément X dans un tableau T. Fournir une procédure qui produit un booléen (trouvé). Si trouvé=vrai, donner le rang de la 1ère occurrence de X.

16 - Recherche dans un tableau ordonné (croissant). On recherche l'élément X dans un tableau ordonné T(1..N). Fournir une procédure qui produit un booléen (trouvé). Si trouvé=vrai, Rang est égal à l'indice de la première occurrence de X. Si trouvé = faux, la valeur de *Rang* est sans importance.

# 29. Solutions

3- Calcul du pgcd, le plus grand diviseur commun de deux nombres M et N (algorithme d'Euclide) sous forme de fonction.

```
Procedure exemple IS
u,v : Integer;
   Function pgcd(x,y : Integer) return Integer IS
   a,b : Integer;
   Begin
      If (x=0) or (y=0) Then
         put("non défini"); Return (0);
      End If;
      If (x=y) Then Return(x); End If;
      a:=x; b:=y;
      While a /= b Loop
         If a>b Then a:=a-b;
         else b:=b-a;
         End If;
      End Loop;
      Return(a);
   End pgcd;

Begin
   put("les valeurs de u et de v?");
   get(u); get(v);
   put("résultat est "); put(pgcd(u,v));
End exemple;
```

4- Ecrire la fonction factorielle (On suppose N > 0)

4-a avec une boucle ascendante :

```
Function Factorielle(N : Integer) return Integer IS
fact : Integer := 1;
Begin
   For i IN 2..N Loop
      fact := fact * i;
   End loop;
   Return(fact);
End Factorielle;
```

4-b avec une boucle descendante :

```
Function Factorielle(N : Integer) return Integer IS
fact : Integer := N;
Begin
   For i in reverse 2..N-1 Loop
      fact := fact * i;
   End loop;
   Return(fact);
End Factorielle;
```

5- Lecture, remplissage et écriture d'un tableau d'entiers T(1..N) avec des procédures.

```
        N : Constant Integer :=....;
Type    Tab is Array (1..N) of integer;
        T : Tab;

Procedure lecture is   -- utilisation des variables globales
Begin -- Lecture et remplissage
   For Indice In 1..N Loop
      Put("donner l''élément d''indice "); Put(Indice);
      Put( " du tableau ? : "); Get(T(Indice));
   End loop;
End Lecture;

Procedure Ecriture is
Begin  -- Ecriture
   For Indice In 1..N loop
      put("l''élément d''indice "); put(Indice);
      put(" du tableau est : "); Put(T(Indice));
   End loop;
End;
End Ecriture ;
Begin
   -- utilisation
End ;
```

6- Recherche de l'indice du plus grand élément d'un tableau d'entiers T. Un indice doit préciser le rang de cet élément. Le tableau T est supposé non vide.

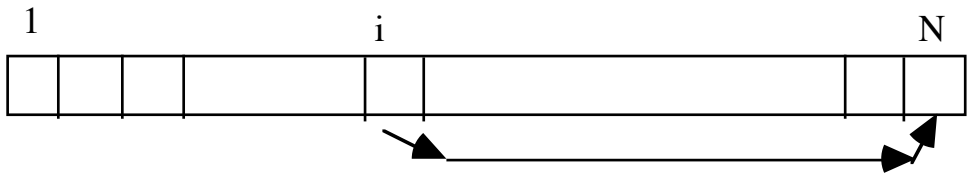
```
        N : Const := ...;
SubType  Bornes is integer range 1..N;
Type     TabEnt is array (Bornes) of Integer;

Function Maximum(M: Bornes; T: TabEnt ) return integer is
Ind_Max : integer;
Begin
   Ind_Max := M;        -- On pourrait prendre le premier
   For i In reverse 1..M-1 loop
      If T(i) > T(Ind_Max) Then Ind_Max := i; End if;
   End Loop;
   Return (Ind_Max );
End Maximum;
```

**==> Préciser l'assertion à la sortie de la boucle Pour.**

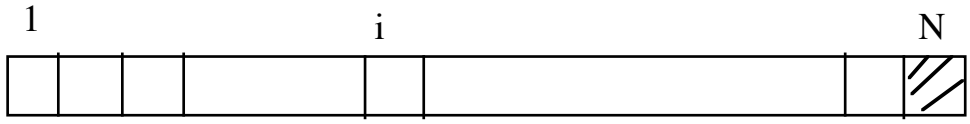
**Exemple d'appel :**

```
        tab : TabEnt;
        -- remplissage de tab(1..16)
        M := Maximum(16, tab);
```

6' - Application : tri d'un tableau d'entiers :
En utilisant la fonction précédente, trier un tableau d'entiers.
Démarche :




Trouver T(i) le max de T(1) .. T(N),
Permuter T(N) et T(i)
Recommencer avec T(1) .. T(N-1)
Jusqu'à T(1) .. T(1)




```
Procedure Trier(M: Bornes; T: IN OUT TabEnt ) is
Ind_Max : integer;
Begin
   For i in reverse 2..M loop
      Ind_Max := Maximum(i, Indice);
      permuter(T(i), T(Ind_Max ));
   End loop;
End Trier;
```

7- La même chose qu'en 4 mais pour un tableau de mots.
Les chaînes de caractères sont comparables entre elles.

8 - recherche du rang du plus petit élément d'un tableau d'entiers. Le tableau T est supposé non vide.

```
                N : Const := ...;
SubType  Bornes is integer range 1..N;
Type          TabEnt is array (Bornes) of Integer;

Function Rang(M: Bornes; T: TabEnt ) return Integer IS
Rang_du_min : Integer;
Begin
   Rang_du_min := 1;            -- On peut prendre M
   For i in 2..M Loop
      If     T(i) < T(Rang_du_min)
      Then   Rang_du_min := i;
      End If;
   End Loop;
   Return (Rang_du_min);
End Rang;
```

9- Comptage du nombre d'occurrences d'un caractère donné dans une phrase. La phrase est une chaîne de caractères.

```
            N : Const := ...;
Type        Bornes is range 1..N;      -- autre forme de
                                              -- déclaration d'entiers
SubType  TPhrase IS string (Bornes);

Fonction Occurrence(X: character; M: Bornes; Ph: TPhrase) return
                                 Integer IS

occ_de_X : Integer := 0;
Begin
   For ind IN  1..M Loop
      If Ph(ind)=X Then
         occ_de_X := occ_de_X +1 ;
      End If;
   End Loop;
   Return (occ_de_X );
End Recherche;
```

*Exemple d'appel :*

```
repetition := Occurrence('x', 53, maphrase);
```

10- Définir si un mot est un Palindrome (radar, tôt, sus). On suppose donnée une fonction *Longueur(Ch)* qui donne la longueur de la chaîne Ch.

10-a Fonction itérative Palindrome:

```
                Taille_mot : Const Integer := ... ;
SubType  Mot IS string(1..Taille_mot);

Function Palindrome(M : Mot) return boolean IS
i : Integer := 1;
L : Integer:= Longueur(M);
continue : boolean :=true;
Begin
   While (i <= L / 2) and continue Loop
      continue := (M(i)=M(L-i+1));
      i := i + 1;
   End loop;
   Return(continue);
End Palindrome;
```

==> Modifier l'algorithme pour traiter des cas tels que :

ELU PAR CETTE CRAPULE

10-b : Fonction itérative Palindrome modifiée pour traiter les cas tels que :

ELU PAR CETTE CRAPULE

```
            Taille_mot : Const Integer := ... ;
SubType  Mot IS string(1..Taille_mot);

Function Palindrome(M : Mot) return  boolean IS
gauche : Integer := 1;
droite : Integer:= Longueur(M);
continue : boolean :=true;
Begin
   While (gauche <= droite) and continue Loop
      If M(gauche)=' ' Then gauche := gauche +1;
      else
         If M(droite)=' ' Then droite := droite -1;
         else  continue := (M(gauche)=M(droite ));
               gauche := gauche +1;
               droite := droite -1;
         End If;
      End If;
   End loop;
   Return(continue);
End Palindrome;
```

11- Compter le nombre de mots dans une phrase (le blanc est le seul séparateur). La phrase se termine par un point et contient au moins un mot. Deux mots peuvent être séparés par plus d'un blanc (utiliser une procédure).

```
      Taille_phrase: Const Integer := ..;
SubType Phrase IS string(1..Taille_phrase);

Procedure compte_mots(T: phrase; Bsup : Integer;
                              NB : out Integer) IS
I : Integer := 1;
Dans_mot : boolean := false;
NBM : Integer := 0;

Begin
   While I <= Bsup   And then T(I) /= '.' Loop
      If T(I)=' ' Then
         If Dans_mot Then
            NBM := NBM +1;
            Dans_mot := false;
         End If;
      else Dans_mot := true;
      End If;
      I := I+1;
   End loop;

   {I>1 & I>Bsup & T(I)='.' & (Dans_mot= Faux I Vrai) }

   If Dans_mot Then  NBM := NBM +1;
   End If;
   NB := NBM;
End compte_mots;
```

12- Recherche du mot le plus grand dans une phrase. Mêmes hypothèses que l'exemple précédent. On demande à extraire le mot le plus long de la phrase.

```
Procedure compte_mots(T: phrase; Bsup : Integer;
                      Mot_long : out Tmot) IS
I : Integer := 1;
ind : Integer := 1;
Mot_courant : Tmot := "";
   -- Mot_courant contient le mot en cours dans  la phrase.
   -- Mot_courant est égal à la chaîne vide si l'on commence ou          -- si le
caractère précédant un blanc est aussi un blanc

Begin
   Mot_long := "";       -- la chaîne vide
   While I <= Bsup and T(I) /= '.' Loop
      If T(I)=' ' Then
         If Mot_courant /= "" Then    -- premier blanc séparateur
            If Mot_courant > Mot_long Then
               Mot_long :=Mot_courant ;
            End If;
            Mot_courant := "";
         End If;
      else
         Mot_courant := Mot_courant & T(I);       -- ajout d'un car
      End If;
      I := I+1;
   End Tq;
```

*{I>1 & I>Bsup & T(I)='.' & (Mot_courant="" | suite de caractères) }*

```
   If Mot_courant /= "" AND Then
      If Mot_courant > MOT Then Mot_long :=Mot_courant ;
   End If;
End compte_mots;
```

13- calcul des occurrences de tous les caractères de l'alphabet figurant dans une phrase. Le résultat sera rangé dans un tableau d'entiers. On suppose que toutes les lettres sont en majuscule.

```
        Taille_phrase: Const Integer := ..;
   SubType  Bornes IS 1..Taille_phrase;
   SubType  Phrase IS string(Bornes);
   SubType  Majuscule IS 'A'..'Z';
   Type       Tab_Occ IS array (Majuscule ) of Integer;

   Procedure nombre_occ(Ph : phrase; Bsup : Bornes ;
                        Tab : out Tab_Occ ) IS
   ind : Integer;
   C : character;
   Begin
      -- initialisation de Tab : Tab('A'..'Z'=> 0)
      For C IN  Majuscule Loop
         Tab(C) := 0;
      End For;

      For ind IN  1..Bsup Loop
         C := Ph(ind);
         Tab(C) := Tab(C) +1;
         -- on peut remplacer ces deux affectations par
         -- Tab(Ph(ind)) := Tab(Ph(ind)) +1;
      End loop;
   End nombre_occ;
```

# 30. Modèles environnement et mémoire

Concept et intérêt :

- caractérise l'état du calcul
- couples représentant les valeurs des identificateurs
- permet de mieux comprendre les règles d'évaluation des expressions, l'exécution des commandes, la visibilité et la portée, ....

• Lien avec les blocs :

- environnement local à un bloc (crée au début du bloc, il disparait à la fin du bloc),
- environnement global (modifié mais persistant)

## 30.1. Modèle CAML

CAML dispose d'un environnement composé de couples (appelés des **liaisons**) de la forme <identificateur, valeur>.

### *30.1.1- Exemple*

• *ENV* composé de couples *(idf, val)* dénote l'environnement courant.
• *[(idf, val) ΔENV]* dénote le nouvel environnement E, extension de l'environnement ENV par la liaison *(idf,val)*.
• Tout couple est ajouté en tête.
• La recherche de la valeur d'un *idf* commence par la tête de l'environnement courant.
• Par exemple, si E = [(x,1) (x,2) ΔENV], la valeur de x est 1.
• Exemple CAML montrant l'évolution de l'environnement.
Soit ENV l'environnement courant :

| **Expression** | **nouvel environnement** |
|---|---|
| **#let x=10;;** | E0 = [(x,10) ΔENV] |
| **# x;;** donne ==> 10 | E0 inchangé |
| **# x + 2;;** donne ==> 12 | E0 inchangé |
| **#let succx = x +1;;** | E1 = [(succx ,11) (x,10) ΔENV] |

*succx est évalué une fois pour toute et vaut 11*

| **Expression** | **nouvel environnement** |
|---|---|
| **# succx;;** donne ==> 11 | [(succx ,11) (x,10) ΔENV] |
| **#let x=1;** | [(x,1) (succx ,11) (x,10) ΔENV] |
| **# succx;;** donne ==> 11 | |

*On ne recalcule pas succx avec la nouvelle valeur de x, succx a conservé sa valeur (11) déduit de l'environnement présent lors de son évaluation.*

**#x;;** donne ==> 1 [(x,1) (succx ,11) (x,10) ΔENV]
*le premier x rencontré dans l'environnement*

**#x+y;;** donne ==> *erreur,*
***y** n'est pas dans l'environnement*

**#let x=x+1;;** [(x,2) (x,1) (succx ,11) (x,10) ΔENV]

*Pour évaluer x+1, on trouve (x,1) dans l'environnement et un nouveau couple (x,2) est ajouté à l'environnement. Ceci est une différence majeure par rapport à l'affectation dans ADA.*

## *30.1.2- Cas de déclaration collatérale en CAML*

*Dans une déclaration collatérale (avec <u>and</u>), chaque expression est évaluée en présence de l'environnement courant (mais n'y est pas insérée). L'ajoute de l'ensemble des déclarations à l'environnement se fait "d'un seul coup" à la fin de la déclaration collatérale.*

Exemples :

• Soit l'environnement E0 à ce stade des calculs

E0 = [(x,2) (x,1) (succx ,11) (x,10) ΔENV].

**#let z=1 <u>and</u> u=2;;** *ajout de (z,1) (u,2) à l'environnement; z est inconnue quand u est évaluée.*

L'environnement à ce point devient :

E = [(z,1) (u,2) (x,2) (x,1) (succx ,11) (x,10) ΔENV]

**#let s=1 and d=s+1;;** *==> erreur, s inconnue (s n'est pas dans E). L'évaluation échoue et (s,1) <u>n'est pas</u> ajouté à l'environnement.*

## *30.1.3- Cas d'environnement local en CAML*

- Crée au début du bloc, l'environnement disparait à la fin du bloc.
- En CAML, un environnement local est crée à l'aide du mot clef **in**.
- Pour évaluer une expression contenant **in** : *let id=exp1 in exp2*
  - on évalue exp1 donnant la valeur v
  - on ajoute la liaison (id, v) à l'environnement
  - on évalue exp2 dans cet environnement
  - on supprime la liaison (id, v) de l'environnement

Exemple (avec environnement = E ci-dessus) :

**# let prod = 5*5 in prod + prod * prod;;** E'=[(prod, 25) ΔE]
**-: int = 650** *suppression de (prod, 25).* E est rétabli.
**#prod;;** *==> erreur, variable inconnue dans E*

## *30.1.4- Cas d'évaluation d'une fonction en CAML*

- D'une manière générale, une fonction est définie par $x \rightarrow exp\_x$. C'est à dire, l'expression *exp_x* s'applique au paramètre formel *x*.

**Exemple** (avec environnement = E ci-dessus) :

**# let f A = if A > x then true else false;;**
**f : int --> Bool = <fun>**

Cette fonction utilise la variable x. Cette variable doit être définie dans l'environnement courant (puisqu'elle ne figure pas en paramètre formel de f). Dans l'environnement E, x vaut 1.

Toute fonction est évaluée dans l'environnement courant. La valeur de la fonction **f** (appelé le **code de la fonction** f) est donc :

**A → if A > 1 then true else false;;**

- Une *valeur fonctionnelle* est un couple appelé **fermeture**. La premier composante de ce couple est le *code de la fonction* et son seconde composante est l'environnement présent au moment de l'évaluation de la définition de la fonction (nommé Env_def).

• Un fermeture sera notée << X → *exp_X, Env_def*>>.
Pour la fonction **f** ci-dessus et l'environnement de définition

E = [(z,1) (u,2) (x,2) (x,1) (succx ,11) (x,10) ΔENV]

la fermeture de f sera:

**<< A → if A > 1 then true else false**, *E*>>

Cette fermeture est ajoutée à l'environnement E. De ce fait E devient :

E1 = [<< **A → if A > 1 then true else false**, *E*>> ΔE]

Il est important de noter que la fermeture contienne l'environnement présent au moment de l'évaluation de la définition de la fonction car cet environnement évolue et ne doit pas avoir d'effet a posteriori sur la fonction. Ainsi, si nous ajoutons :

**#let A=45 and x=3;;**

Les couples (A,45) (x,3) seront ajouté à E1 pour donner E2:

E2 = [(A,45) (x,3) ΔE1]

E2 = [(A,45) (x,3) << **A → if A > 1 then true else false**, *E*>> ΔE]

**#x;;** ==> 3

*On constate que E2 contient (x,3) mais f a déjà été évaluée dans E avec (x,1).*

## Appel de la fonction f :

**# f z+u*x;;**

Lorsque la fonction f est appelée, on cherche dans l'environnement courant (E2) et obtient la fermeture de f :

**<< A → if A > 1 then true else false**, *E*>>.

L'expression *z+u*x* (paramètre effectif) est évaluée dans l'environnement courant **E2** et donne 5. On reprend alors l'environnement de définition de f (E) et construit l'environnement provisoire Env_prov = [(A,5) Δ*E*].

L'expression **"if A > 1 then true else false"** est évaluée alors dans Env_prov et donne true.
Cette valeur représente le résultat de l'appel de f.
L'environnement Env_prov est alors détruit et l'environnement actuel redevient :

E2= [(A,45) << **A → if A > 1 then true else false**, *E*>> ΔE].

On constate que les couples (A,45) (x,3) n'ont eu aucun rôle dans l'évaluation de f.

## *30.1.5- Cas d'importation de module*

- Les déclaration des modules importés s'insère entre les liaisons déjà importées et les déclarations globales.
- Elles ne masquent donc pas les déclaration faites par l'utilisateur mais peuvent masquer celles d'un autre module importé.

**Exemple :**

- le module mod1 déclare (z,1) , (y,2)
- le module mod2 déclare (x,3), (z,4)
- l'environnement initial est Env_init

**#let x=7 and y=3;;**

③ L'environnement courant devient *E = [(x,7) (y,3) ΔEnv_init]*

**##open "mod1";;**

③ L'environnement devient *E1 = [(x,7) (y,3) (z,1) (y,2) ΔEnv_init]*.
Les déclarations de mod1 s'insèrent après celles de l'utilisateur.

**##open "mod2";;**

③ *E2 = [(x,7) (y,3) (x,3) (z,4) (z,1) (y,2) ΔEnv_init]*
Les déclarations de mod2 s'insèrent après celles de l'utilisateur mais avant celles de mod1.

- La valeur de x=7, y=3, z=4.

**# let z=5;;**

③ *E3 = [(z, 5) (x,7) (y,3) (x,3) (z,4) (z,1) (y,2) ΔEnv_init]*
La valeur de z sera 5.

• On peut cependant accéder aux valeurs masquées en faisant précéder le nom de l'identificateur par le nom du module suivi de "__" :

**mod1__x = 3, mod2__y=2.**

• Les liaisons importées peuvent être supprimées par :
**#close "nom_de_module";;**

*Résumé*

**Le modèle d'environnement de CAML permet d'expliquer le processus d'évaluation (et de passage de paramètres) en CAML.**

## 30.2. Modèle ADA

• Dans un langage sans données mutables, l'évaluation d'une expression dépend de l'environnement courant. Mais dès que des valeurs mutables apparaissent (l'affectation), l'évaluation dépend également des valeurs référencées par les adresses présentes dans l'environnement.

• ADA manipule des données mutables. Le modèle d'environnement de CAML doit donc être étendu pour expliquer l'évaluation.

• ADA permet l'affectation : modification des cases mémoire.
Dans *x := x+1;* le contenu de la case nommée *x* est incrémenté.

• En ADA, une déclaration telle que **x:integer** a deux interprétations:
- x désigne une valeur de type "référence à un entier"
- il désigne une valeur de type entier et dénote la valeur qui est référencée par cette l'adresse.

**En CAML**

```
#let x= ref 1;;
x : int ref = ref 1
#x := !x +1;;
- : unit = ()                    (* effet de bord *)
#x;;
- : int ref = ref 2
#let x=!x+1;;          (* p à gauche, ça n'est plus le même x*)
x : int = 3            (* le nouveau x *)
#x;;
- : int = 3
#let y=!x+1;;   ==>  Erreur, !x est illégal (sur le dernier x).
```

**En ADA**

```
x : integer := 0;
x := x+1;
```

Le sens de l'affectation *x:=x+1* en ADA :
- trouver l'adresse dénotée par *x* et rechercher la valeur associée à cette adresse, c'est à dire 0. Ici, x est interprété comme une adresse.
- calculer x+1. Ici, x est interprété comme une valeur (0).

• L'exemple précédent fait apparaître trois notions :
  - identificateur
  - l'adresse désignée par cet identificateur
  - la valeur référencée

• Pour expliquer cette évaluation, on introduit la notion de **mémoire** : une liste de couples <adresse, valeur_référencée_par adresse>.

• On a également la notion d'**environnement** : une liste de couples <*idf,val*>. Ici, *idf* est l'identificateur et *val* peut être :

- une adresse : idf est alors une variable;
- une valeur appartenant à un type : idf est alors une constante;
- une exception;
- une valeur fonctionnelle (fermeture) : idf désigne une abstraction;
- un environnement : idf désigne un module
- un type

## *30.2.1- Notion d'état*

• Le couple mémoire-environnement (<Mem, Env>) est appelé un **état**. Pour toute variable *idf* dans *Env*, l'adresse liée à *idf* dans *Env* figure dans *Mem*. On notera l'adresse d'*idf* par **@idf**.

**Exemple1** :

L'évaluation dans l'état <[E][M]> de la déclaration *x : integer := 0* crée l'état <[(x, @x) Δ E] [(@x, 0)]>.

## *30.2.2- Traitement des modules*

**Exemple2** :

```
with text_io;
procedure p is ...
      ...
begin
      text_io.put("hello"); text_io.new_line;
end p;
```

• L'état initial <[E0][M0]> (le module *standard* est par défaut présent).
• Le module *text_io* définit des procédures et fonctions (abstractions).
L'environnement de *text_io* est de la forme :

*[....(put, val_put) (new_line, val_new_line) ......]*

où *val_put* désigne la valeur de la commande *put*...

• La première ligne importe *text_io*. L'environnement devient :
*[(text_io, val_text_io)Δ E0]*.
Lors de l'exécution de *put("hello")*; on va chercher dans *val_text_io* la valeur *val_put* associée à la commande *put*.

• La présence de la clause *use* en ADA permet d'ometre les préfixes.
Par exemple avec :

```
with text_io; use text_io;
procedure p is ...
```

L'environnement à ce stade sera (◊ est l'opérateur de combinaison (ajout) d'environnements.

*[(text_io, val_text_io)◊(put, val_put) (new_line, val_new_line)...]*

Ce qui a pour effet de rendre *val_put* et *val_new_line* directement présentes dans l'environnement du programme p.

## *30.2.3- Recherche dans l'environnement*

• Comme en CAML, la recherche de la valeur d'un identificateur s'effectue en parcourant d'abord l'environnement de déclaration puis l'environnement d'importation. Donc, les objets déclarés dans l'unité courante ou dans les unités englobantes ont "priorité" sur les objets d'importation.

## *30.2.4- surcharge et ambiguïté*

• Dans une importation telle que
**with text_io; use text_io;**
**with integer_io; use integer_io;**

L'environnement contiendra des liaisons identiques telles que :
*... (put, val_put)... (put, val_put)...*
où le premier couple se rapporte à **integer_io** et le seconde à **text_io.**
Pourtant, il n'y aura pas d'ambiguïté dans les commandes *put(un_entier)* et *put(un_caractère)*. Ceci est dû à la possibilité de la *surcharge* d'ADA. Le paragraphe suivant montre un autre cas d'ambiguïté.

• Soit le module M définissant *[(a,val_a1)(b,val_b)]* et le module N définissant *[(a,val_a2)(c,val_c)]*.

Si un programme ADA importe M et N par :
**with N; use N;**
**with N; use N;**

L'environnement :
*[(M,val_M) (M,val_M) ◊(a,val_a1)(b,val_b) (a,val_a2)(c,val_c)]*.

L'expression *(b+c)* calculera *(val_b + val_c)* mais l'expression *(a+b)* est ambiguë car l'identificateur *a* est lié à deux valeurs. Pour éviter cette ambiguïté, on écrira par exemple *(M.a+b)*.

p **Cette règle est différente de celle de CAML qui choisit val_a1.**

## *30.2.5- Exemple*

```
with text_io;
procedure p is
        c : constante character := 'a';
        i : integer := 1;
        function f(...) return ... is ....
begin
        text_io.put("hello"); text_io.new_line;
        i := i+1;
        ... := f(....);
end p;
```

L'état eu "begin" de p :

*[(f , val_f) (i, @x) (c, 'a') Δ E_init ◊(text_io, val_text_io] [(@x, 1)].*

On remarque que la fermeture *(f , val_f)* est entièrement différente de celle de CAML. Elle ne dépend pas d'un environnement de définition et sera évalué dans l'environnement de son appel.

• D'une manière générale, les règles dévaluation d'expressions vue en CAML s'appliquent en ADA. Cependant, il faudra envisager des adaptation au modèle d'état défini pour ADA et tenir compte des différences que nous avons signalées ci-dessus.

# 31- Spécification, analyse et programmes

**But** : Décomposer un problème en sous problèmes puis
Traduire les sous problèmes en programme, procédure et fonction

## Trois approches :

• Décomposition (approche **descendante**) d'un problème :

➥ Partir du problème initial et décomposer. La décomposition s'arrêter au niveau des abstractions/opérations qui :
- préexistent (dans une bibliothèque, paquetage, ...)
- sont très simples et de bas niveau

• Décomposition (approche **ascendante**) d'un problème :

➥ Considérer les préexistants (bibliothèques, paquetages, ...) et les opérations de bas niveau pour considérer une décomposition au problème.

• Décomposition (approche **mixte**) d'un problème :

➥ Basée sur l'approche descendante, cette approche considère en parallèle la décomposition descendante en se laissant diriger par les préexistants (ascendante).

Cette approche est la plus réaliste et fréquemment utilisée.

## Intérêts :

- Réfléchir avant d'écrire, faire ressortir les points communs
- évaluer la complexité des solutions
- Réutiliser les préexistants
- Produire des modules réutilisable pour enrichir la bibliothèques.

**Dans la suite, nous considérons en particulier l'approche mixte des problèmes.**

## 31.1- Spécification

Une spécification (d'un problème = son énoncé) est la description dans un langage aussi formel que possible du problème à résoudre.

En d'autre termes, une spécification est une description, dans un langage proche des mathématiques, d'une part d'une correspondance entre des données et des résultats, d'autre part des propriétés de cette correspondance.

**Exemple :** Soit à calculer la somme de deux entiers naturels.

On peut spécifier ce problème de différentes manières. Par exemple :
- calculer S la somme de N et M ($N,M \geq 0$)
- calculer S =N + M ($N,M \geq 0$)

Cette spécification établit une correspondance entre le couple (N,M) d'une part et l'entité S d'autre part.

Les propriétés d'une telle correspondance seront par exemple:

0 + M => M
succ(N') + M => succ(N'+M)

## 31.2- Algorithme

Un algorithme est défini comme la donnée d'une suite d'opérations à effectuer, pour un exécutant donné, pour réaliser effectivement la tâche décrite par une spécification.

## 31.3- Programme

Un programme est le codage de l'algorithme dans un langage de programmation.
Les programmes sont des abstractions nommées (fonctions et procédures) et le passage de la spécification au programme final se fait de façon progressive par décomposition de la spécification en utilisant la *décomposition* des abstractions, la *récursion* et sa variante, l'*itération*.

## 31.4- Spécification : décomposition séquentielle

## 31.5- Exemple-1 : calcul de prix

Soit la spécification donnée par le texte suivant.

> Calculer le prix toute taxes comprises (TTC), arrondi au franc le plus voisin, connaissant le prix hors taxes (HT) et sachant que le taux de TVA est de 18,6%.

Le travail à réaliser semble particulièrement simple. Cependant, d'un lecteur à l'autre, le texte peut avoir différentes interprétations. En fait, cette spécification est trop informelle et il nous faut la préciser. Il faudra ajouter ou détailler des éléments pour construire une spécification formelle qui soit adéquate au problème.

Par exemple, il peut être irritant de devoir entrer plusieurs fois le même taux de TVA. Il peut être tout aussi irritant de ne pas pouvoir changer ce taux. Ces décisions libres n'ont lieu d'être que par ce que la spécification est ambiguë sur certains points.

### *31.5.1- La décomposition*

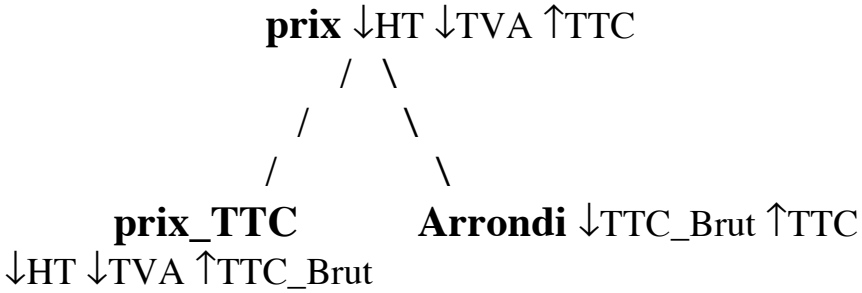
Le problème se décompose en deux étapes: le calcul du prix TTC "brut" puis le calcul de l'arrondi de ce prix TTC "brut".

La spécification est ainsi décomposée en la succession de deux étapes.

Supposons avoir défini une fonction notée *prix_TTC* qui calcule le prix TTC "brut" et une fonction *arrondi* qui fournit l'arrondi. Le résultat recherché est la valeur de la fonction :

**arrondi(prix_TTC(HT, TVA))**

On peut représenter cette décomposition par le schéma suivant qui décrit le flot de données :




*Si une spécification a été décomposée en la succession de plusieurs étapes, elle est obtenue par la composition des fonctions spécifiant chacune de ces étapes.*

Dans le cas de cet exemple

**TTC := arrondi(prix_TTC(HT, TVA))**

Ayant apporté cette première décomposition qui dénote la séquence des opérations à effecteur pour aboutir au résultat recherché, On s'intéresse plus en détail à certains aspects plus concrets concernant les décompositions identifiées.

## *31.5.2-D'autres détailles à préciser*

- Quelles sont les valeurs possibles pour le prix HT?

Nous pouvons décider qu'un prix négatif est une erreur. Il faudra traiter ce cas :
- L'arithmétique choisie doit être celle des nombres réels et entiers.
- De même, On décide que la TVA sera donnée sous forme d'un réel (par exemple 18.6).
- ...

## *31.5.3-Spécification des fonctions*

Une première définition de la fonction prix_TTC :

Prix_TTC(HT,TVA) =
- si HT < 0 => erreur
- si HT ≥ 0 => HT * (1.0 + TVA / 100.0)

Spécification de la fonction arrondi

Arrondi(X) =

- si $X \geq 0$ =>

  Tronc(X) + 1 si X-Tronc(x) $\geq 0.5$

  Tronc(X) sinon

- si $X < 0$ =>

  Tronc(X) si X-Tronc(X) $> -0.5$

  Tronc(X) - 1 sinon

Exemples (résultats attendus):

arrondi(3.85) = 4
arrondi(3.45) = 3
arrondi(-3.85) = -4
arrondi(-3.45) = -3

## *31.5.4- Codage en algorithmes*

Le codage des algorithmes prend en compte la spécification à un niveau plus détaillé. Ce codage manipule les données à un niveau plus concret et règle la plupart des détailes.

```
Function prix_TTC(HT : float; TVA : float) return float is
Begin
   If HT < 0 Then erreur;
   Else return HT * (1.0 + TVA / 100.0);
   End if;
End prix_TTC;
```

En s'appuyant sur la spécification de la fonction arrondi, on peut proposer un algorithme plus efficace.

```
Function arrondi(X:float) :Return  Integer is
signe : Integer := 1;
y : Float := X;
Begin          -- Tronc(x) sera traduit en Integer(x) en ADA
   If y <0 Then signe := -1 End If;
   y := y * signe;
   If y - Tronc(y) ≥ 0.5 Then y := Tronc(y)+1;
   Else y := Tronc(y);
   End If;
   Return y * signe;
End arrondi;
```

Pourquoi a-t-on défini la fonction arrondi d'une manière totale en traitant les nombres négatifs sachant que la fonction prix_TTC rejette ce cas de figure par un message d'erreur?

La réponse est que cette fonction peut devenir un composant logiciel réutilisable. On préférera toujours une définition la plus totale possible de ce type de fonction afin d'éviter de les réécrire lors d'une autre utilisation dans un autre contexte.

On aurait pu écrire une fonction *arrondi_positif* traitant seulement les nombres positifs; mais une telle définition restreint son utilisation au cas particulier de cet exemple.

Il en est de même pour la TVA.

L'exemple précédent a été traité par une *décomposition séquentielle*.

## 31.6- Généralisation : spécification itérative et récursive

Comment procéder si :

- Il y a un nombre connu N de prix à traiter;
- Il y a un nombre inconnu de prix à traiter...

=> deux classes de décomposition :
- décomposition **récursive**
- décomposition **itérative** avec les schémas :
  - faire n fois
  - Tant que ... faire ...
  - Répéter ... jusqu'à ...

## 31.7- Exemple-2 : le problème des télégrammes

### La pécification textuelle (énoncé) du problème :

On veut traiter une série de télégrammes qui arrivent sur un flot quelconque de données (fichier, clavier, ligne téléphonique, au guichet des PTT...).
Chaque télégramme est une suite de mots terminée par le mot 'zzzz'.

Chaque télégramme lu est imprimé ainsi que les informations suivantes :
- le nombre total des mots;
- le nombre de mots longs, mots dont la longueur est > 20 caractères;
- le coût de l'envoi sachant qu'un mot 'ordinaire' coûte n francs et un mot long coûte le double.

Dans un premier temps, on s'intéresse au traitement d'un seul télégramme:

•Définir la séquence d'actions à entreprendre pour réaliser cette tâche;
•Relever les ambiguïtés de la spécification et prendre une décisions dans chaque cas;
•Proposer différentes méthodes de spécifications (récursive, itérative);
Etc...

### *31.7.1-Spécification séquentielle sommaire des actions*

On identifie le rôle principal de chaque télégramme et par là le traitement qui le concerne. Le traitement concernant un télégramme doit être répété jusqu'à ce qu'il n'y en ait plus.

## *31.7.2-Spécification du schéma général*

➠ Tant qu'il y a des télégrammes
Lire et traiter un télégramme

➠ Lire et traiter un télégramme
- lire le télégramme mot par mot et
imprimer chaque mot
jusqu'au mot 'zzzz'
- imprimer le nombre de mots
- imprimer le nombre de mots longs
- calculer le coût
- imprimer le prix

p **Problème** : Qu'est-ce qu'un mot ?
L'énoncé ne précise rien sur la définition d'un mot.

<u>On décide :</u>

- les ponctuations ne sont pas des mots.
- deux mots sont séparés par un ou plusieurs séparateurs.
- tout caractère différent d'une lettre ou d'un chiffre est considéré comme un séparateur.
- toute séquence de lettres et/ou de chiffre est un mot.

➠ Lire-mot :
- sauter les séparateurs
- accumuler les lettres et les chiffres
jusqu'au premier séparateur

## *31.7.3-Traitement concernant un télégramme*

D'après la spécification, un télégramme est une suite non-vide de mots terminée par 'zzzz'.
La spécification ne précise rien sur une éventuelle valeur calculée par le traitement d'un télégramme qui serait utilisée ailleurs.

## *31.7.4-Décomposition itérative*

### *31.7.4.1- Utilisation du schéma itératif Tant que*

L'exemple précédent peut également être traité par le schéma itératif Tant que de la manière suivante :

Télégramme(n) =
- nb_mots :=0
- nb_mots_longs := 0
- soit M un mot lu
- Tant que M <> 'zzzz'
  - imprimer M
  - faire +1 sur nb_mots
  - si longueur(M) > 20 =>
    faire +1 sur nb_mots_longs
  - Lire un nouveau mot dans M

- imprimer nb_mots
- imprimer nb_mots_longs
- prix := (nb_mots+nb_mots_lons) * n
- imprimer prix

### *31.7.4.2- Utilisation du schéma itératif répéter*

De même, on peut appliquer le schéma itératif répéter à l'exemple précédent :

Télégramme(n) =
- nb_mots :=0
- nb_mots_longs := 0
- soit M un mot lu
- si M <> 'zzzz' =>
  Répéter
  - imprimer M
  - faire +1 sur nb_mots
  - si longueur(M) > 20 =>

faire +1 sur nb_mots_longs
- Lire un nouveau mot dans M
•jusqu'à M = 'zzzz'

•imprimer nb_mots
•imprimer nb_mots_longs
•prix = (nb_mots+nb_mots_lons) * n
•imprimer prix

### *31.7.4.3- Schéma global traitant une série de télégrammes*

Le seul paramètre nécessaire est n : le prix de chaque mot

```
Série =
  Répéter
       télégramme(n);
  Jusqu'à fin_télégrammes;

Série =
  Tant non fin_télégrammes faire
       télégramme(n);
  Fin Tq;

Série = .... d'autres solutions existent
```

### *31.7.4.4- Décomposition par le schéma faire n fois*

Ce schéma s'applique lorsqu'un traitement doit être appliqué un nombre N fini de fois. Par exemple, si l'on doit appliquer N fois une fonction F réalisant le traitement A à une valeur donnée X; autrement dit, évaluer $F^N(X)$.

**Exemples :**

• $F^N(X)$ : appliquer 3 fois la fonction *pred* (prédécesseur) à la valeur 7 => pred(pred(pred(7))) = $pred^3(7) = 4$.

• $F(X_{i=1..N})$ : calculer la somme des entier 1..N; i.e. appliquer la fonction somme aux entiers 1..N;

• $F(X_{i=inf..sup})$ : faire un traitement A au éléments d'un tableau T(inf..sup)

• etc...

Comme on peut le constater, un tel schéma peut s'appliquer lorsque la taille des informations à traiter est connue d'avance. Par contre, Il ne pourra pas être utilisé dès que l'on estime que cette taille peut être inconnue. Ce qui est par exemple le cas de l'exemple précédent.

## 31.8- Exemple3 : gestion des notes des élèves

Spécifier puis écrire l'ensemble des algorithmes réalisant la gestion des notes des élèves d'une école.

• Les informations relatives à un élève sont :
- nom, prénom
- sexe (masculin, féminin)
- nom_jeune_fille (seulement si le sexe est féminin)
- Date_de_naissance (après l'année 1900)
- note (entre 0..20)

• le nombre d'élèves est supposé borné (N)

• L'application doit afficher un menu et proposer les actions suivantes :
  - saisie et ajout des coordonnées d'un élève ;
  - suppression d'un élève (par le nom) ;
  - listing de tous les élèves ;
  - consultation des coordonnées d'un élève (par le nom);
  - modification des coordonnées d'un élève.

Décomposer le problème en sous-problèmes. Identifier les procédures et les fonctions nécessaires; préciser leur entête et leur rôle précis.
Identifier les abstractions réutilisables. Ensuite, écrire les corps des abstractions.

## 31.9- Solution

La spécification semble claire. On décide des choix suivants :

• Naturellement, on identifie l'entité ELEVE et les informations concernant un élève seront représentées par un enregistrement.

• Dans les opérations de recherche, la clé d'accès à un elève sera son nom. On pourra cependant faire de la recherche selon un critère particulier, par exemple sur les notes.

• Pour simplifier, on suppose qu'il n'y aura pas deux noms identiques.

• Le nombre d'élèves étant borné, on représentera l'ensemble des élèves par un tableau TAB de N éléments (N enregistrements).

• En fonction des opérations que l'application doit proposer, on peut :
- conserver TAB sous une forme triée (selon les noms). La recherche d'un nom sera ainsi accélérée (O(LogN) par dichotomie).
- ne pas trier TAB. Dans ce cas, toute recherche sera exhaustive (O(N/2)).

p On prend l'option TAB triée
- Décider si toute insertion (ou suppression) conduit à déplacer éventuellement un certain nombre d'éléments. On peut envisager d'introduire la notion de suppression logique : une marque particulière précisera que tel enregistrement est supprimé. Dans ce cas, les insertions devront se faire dans ces emplacements libres.

p On déplacera les éléments lors d'une insertion ou d'une suppression.

p Un entier TAILLE précisera le nombre actuel d'élèves dans la table (TAILLE ≤ N)

p Pour des raisons d’efficacité, TAILLE et TAB seront des données globales.

• Concernant un élève, on précise l'enregistrement (à champ variant sur le sexe) :
- Nom, Prénom : chaîne de caractère de taille ≤ 20
- Sexe : masculin / féminin (énuméré)
- Nom_jeune_fille : chaîne de caractères de taille ≤ 20
  (seulement si le sexe est féminin)
- Date_naissance : Date; -- un enregistrement
- Note (un nombre réel 0..20)
  Pourra être étendue à un tableau de notes + un champ Moyenne

## *31.9.1-Spécifications*

On proposera des abstractions les plus générales. Par exemple, la lecture du Nom, du Prénom et du Nom_jeune_fille seront faites par une même procédure. La recherche doit être la plus générale pour être utilisée par toutes les autres procédures, etc...

- Schéma général du menu :
  -- après initialisation des données
  Répéter
  - afficher le menu
  - saisir un choix
  - si le choix est valide alors activer la procédure correspondant
  Jusqu'à choix=fin

### *31.9.1.1- Enumération des abstractions identifiées*

➠ Ajout d'un élève :
- ♦ Rôle : ....
- ♦ Opérations remarquables:
  - • *Lire_chaîne* pour lire nom, prénom, nom_jeune_fille
  - • *Recherche* recherche d'un élève par son nom
  - • *Lire_sexe* Lecture du sexe
  - • *Lire_date* Lecture d'un enregistrement Date
  - • *lire_réel* Lecture d'une note
  - • *insere_élève* Insertion effective dans TAB
- ♦ Met à jour TAB et TAILLE
- ♦ TAB reste triée

➠ Suppression d'un élève :

♦ Rôle : ....
♦ Opérations remarquables:

• *Lire_chaîne* pour lire nom, prénom, nom_jeune_fille
• *Recherche* recherche d'un élève par son nom
• *supprime_élève* Suppression effective de la TAB

♦ Met à jour TAB et TAILLE
♦ TAB reste triée

➠ Modification d'un élève :
♦ Rôle : ....
♦ Opérations remarquables:

• *Lire_chaîne* pour lire nom, prénom, nom_jeune_fille
• *Recherche* recherche d'un élève par son nom
• *Affiche_élève* Affichage des coordonnées
• *Lire_sexe* Lecture du sexe
• *Lire_date* Lecture d'un enregistrement Date
• *lire_réel* Lecture d'une note
• *Modif_élève* Modification effective dans TAB

♦ Met à jour TAB
♦ TAILLE inchangée
♦ TAB reste triée

➠ Consultation d'un élève :

♦ Rôle : ....
♦ Opérations remarquables:
• *Lire_chaîne* pour lire nom, prénom, nom_jeune_fille
• *Recherche* recherche d'un élève par son nom
• *Affiche_élève* Affichage des coordonnées

♦ Aucune modification de TAB et de TAILLE

➠ Listing :

♦ Rôle : ....
♦ Opérations remarquables:

Pour tout élève
• *Affiche_élève* Affichage des coordonnées

♦ Aucune modification de TAB et de TAILLE

➠ Affichage :

♦ Rôle : ....
♦ Opérations remarquables: ....

♦ Aucune modification de TAB et de TAILLE

➠ Recherche :

♦ Rôle : ....
♦ Opérations remarquables: ....

♦ Aucune modification de TAB et de TAILLE

### *31.9.1.2- Les Algorithmes*

• Algorithme de la saisie des coordonnées d'un élève et son ajout:

```
Procedure Saisie_et_Insertion Is
E : Eleve; Trouve : ... ; Rang : ....
Begin
   lire_chaîne(E.Nom);       -- une chaîne de caractères
   rechercher(E.Nom,Trouve,Rang);
   If Trouve Then afficher_message("existe déjà");
   Else     lire_chaîne(E.Prénom);
         lire_sexe(E.sexe);
         If E.sexe = feminin Then
            lire_chaîne(E. Nom_jeune_fille);
```

```
            End If;
            lire_date(E.Date_naissance);
            lire_reel(E.Note);
            Inserer_eleve(E);
      End If;
End Saisie_et_Insertion;
```

• Algorithme de la recherche d'un élève :

```
Procedure Recherche(Nom:..,Trouve:...,Rang:...) is
      -- recherche DICHOTOMIQUE des coordonnées d'un élève
      -- selon Nom dans la table triée TAB.

milieu : Integer;
inf : Integer := 1;
sup : Integer := TAILLE;
Begin
      trouve := False;
      If sup ≠ 0 Then
            Loop
                  milieu := (inf + sup) / 2;
                  If  Nom < TAB(milieu).Nom  Then sup := milieu  -1
                  Else     inf := milieu  + 1 ;
                  End If;
                  Exit when  (Nom = TAB(milieu).Nom) OR (inf > sup);
            End loop;

            If TAB(milieu).Nom = Nom Then
                  Rang := milieu;
                  Trouve := True;
            End If;
      End If;
End Recherche;
```

• Algorithme de la recherche d'un élève :

Méthode dichotomique avec calcul de la place d'insertion (si non trouvé) :

```
Procedure Recherche(Nom:..,Trouve:...,Rang:...) IS
   -- recherche DICHOTOMIQUE des coordonnées d'un élève
   -- selon Nom dans la table triée TAB.

milieu : Integer;
inf : Integer := 1;
sup : Integer := TAILLE;
Begin
   trouve := False;
   If TAILLE = 0 Then Rang := 1;
   Else
      Loop
         milieu := (inf + sup) / 2;
         If  Nom < TAB(milieu).Nom  Then sup := milieu  -1
         Else     inf := milieu  + 1 ;
         End If;
      Exit when  (Nom = TAB(milieu).Nom) OR (inf > sup);
      End Loop;
      If TAB(milieu).Nom > Nom Then
         Rang := milieu +1 ;
      Else  Rang := milieu;
      End If;
      Trouve := (TAB(milieu).Nom = Nom);
   End If;
End Recherche;
```

• Autre solution (non dichotomique) de la recherche :

```
Procedure Recherche(Nom:.. ;Trouve:.. ;Rang:.. ) IS
I : Integer ;
Begin
   I := 1; Trouve := False;
   While I ≤ Taille and not Trouve Loop
      If Tab(I).Nom = Nom Then      Trouve:=True;
                                        Rang := I;
```

```
      Else I := I+1;
      End If;
   End Loop;
End Recherche;
```

- Algorithme de l'insertion d'un élève :

```
Procedure Inserer_eleve(E:Eleve) IS
Stop : boolean; i : Integer;
Begin
   I := 1; Stop := False;
   While I ≤ Taille and not Stop Loop
      If TAB(I).Nom > E.Nom Then   Stop:=True;
      Else I := I+1;
      End If;
   End Loop;
   TAB(I+1..Taille+1) := TAB(I..Taille);
   TAB(I) := E;
   Taille := Taille + 1 ;
End Inserer_eleve;
```

- Algorithme de la suppression d'un élève :

```
Procedure Suppression IS
Nom : .... ; Trouve : ... ; Rang : ....
Begin
   lire_chaîne(Nom);
   rechercher(Nom,Trouve,Rang);
   If not Trouve Then afficher_message("n'existe pas")
   Else    TAB(Rang..Taille-1) := TAB(Rang+1..Taille);
              Taille := Taille - 1 ;
   End If;
End Suppression;
```

• Algorithme du listing des élèves :

```
Procedure Listing is
Begin      -- un schéma pour convient
   For I in 1..Taille Loop
      afficher(TAB(I));
   End Loop;
End Listing;
```

• Algorithme de la consultation d'un élève :

```
Procedure Consultation IS
Nom : .... ; Trouve : ... ; Rang : ....
Begin
   lire_chaîne(Nom);
   rechercher(Nom,Trouve,Rang);
   If not Trouve Then afficher_message("n'existe pas");
   Else    afficher(TAB(Rang));
   End If;
End Consultation;
```

• Algorithme de la modification d'un élève :

```
Procedure Modification IS
Nom : .... ; Trouve : ... ; Rang : ....;
E : Eleve;
Begin
   lire_chaîne(Nom);
   rechercher(Nom,Trouve,Rang);
   If not Trouve Then afficher_message("n'existe pas")
   Else    afficher(TAB(Rang));
         -- Copier TAB(Rang) dans E
         -- Champ par champ, demander confirmation de l'info
         -- précédente ou accepter la modification de celle-ci
         -- mettre à jour TAB(Rang) par le nouvel E
   End If;
End Modification;
```

## 31.10- Exemple4 : calcul des nombres premiers 1..N

On veut générer les nombres premiers dans l'intervalle 1..N (crible d'Eratosthene).

Rappel : un nombre k est premier s'il n'est divisible que par 1 et par lui même.

## 31.11- Solution

Analyse -1 (génération):

- 1,2 et 3 sont premiers
- si i est premier (N>i>1) alors les multiples de i dans l'intervalle i..N ne sont pas premiers.

==> on peut donc générer les nombres premiers 1..N en évitant ceux qui ne le sont pas.

==> on peut également considérer chaque nombre de l'intervalle 1..N et *tester* s'il est premier :

Analyse -2 (test):

Pour savoir si un nombre k est premier :

- si k=1,2 ou 3 alors k est premier
- si k est pair alors k n'est pas premier
- diviser k par les nombres impairs 3..racine(k). Si k n'est pas multiple d'un de ces nombres alors k est premier.

### *31.11.1- Spécification*

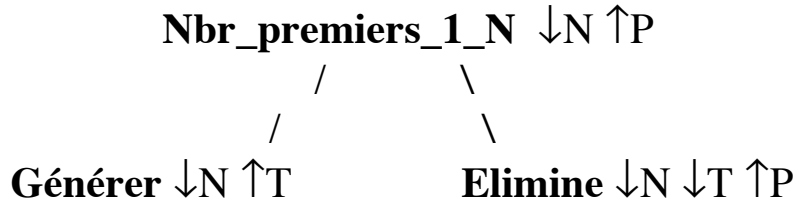
on distingue les étapes :

- Génération des nombres 1..N
- Elimination des nombres non-premiers
- Affichage du résultat

Le résultat final peut donc être représenté par :

**Nbr_premiers_1_N := Elimine(générer(N));**

Ou encore :




## *31.11.2- Les décompositions*

- Elimine :
  - 1,2 sont premiers
  - I := 2
  - Répéter
    - éliminer les multiples de I dans T(I+1..N)
    - I := le prochain nombre non-encore-éliminé
      (ce nombre est premier et impair)
  - Jusqu'à I > N

p Représentation : un tableau de booléens d'où :

- Générer : T(1..N) := vrai

- Solution à la génération des nombres premiers:
  - générer les nombres 1..N
  - pour I dans 2..N tel que I est premier
  - pour J dans I..N/I
    I*J n'est pas premier

## *31.11.3- Algorithmes*

```
Procedure premier IS
   N : constant Integer := ...
   T : Array(1..N) of boolean;
   i : Integer;
Begin
   For j IN 1..N Loop    -- génération des nombres 1..N
      T(j) := True;
   End Loop;
   i := 2;                -- on commence à 2 qui est premier
```

```
   Loop
      For j in i..N/i Loop    -- les multiples de i ne sont
         T(i*j) := False;        -- pas premiers
      End Loop;
      i := i+1;               -- recherche du prochain
      Loop                    -- nombre premier
         If not T(i) Then i:=i+1; End If;
         Exit when  (i>N) or T(i)=True;
      End loop;
      Exit when  (i>N);
   End Loop
End premier;
```

## *31.11.4- Solution avec test d'un nombre premier:*

```
Procedure premier is
   N : constant Integer := ...
   T : Array(1..N) of boolean;
   i : Integer;

Function est_premier(k:Integer) return boolean IS
Begin
   If (k dans 1..3) Then Return True;
   Elsif (k mod 2 = 0) Then Return False;
   Else
      For i IN 3..tronc(racine(k)) Loop
         If (k mod i = 0) Then Return False; End If;
      End Loop;
      Return True;
   End If;
End est_premier;
Begin
   For j IN 1..N Loop    -- génération des nombres 1..N
      T(j) := est_premier(j);
   End Loop;
End premier;
```

## 31.12- Exercice1 : Min max d'une matrice

- Reprendre l'exemple Min-max de matrice (déjà traité)
  - Analyse
  - Etude de la complexité de chaque solution
  - Algorithmes

## 31.13- Exercice2 : Elimination de mots doubles

- Reprendre l'exemple d'élimination des mots doubles (déjà traité)
  - Analyse
  - Etude de la complexité de chaque solution
  - Algorithmes

## 31.14- Exercice3 : Gestion bibliographique

Spécifier puis écrire l'ensemble des algorithmes réalisant la gestion des livres dans une librairie.

• Les informations relatives à un livre sont :

- Titre
- Auteur (nom, prénom)
- Année de bublication
- Maison d'édition
- Résumé
- Mots clés
- Prix
- Quantité en stock

• le nombre de livres est supposé borné (N)

• L'application doit afficher un menu et proposer les actions suivantes :

- saisie et ajout des coordonnées d'un livre ;
- suppression d'un livre (par le titre) ;
- listing de tous les livres ;
- consultation des coordonnées d'un livre (par le titre);
- recherche par mot clé/par auteur
- modification .....

Décomposer le problème en sous-problèmes.
Identifier les procédures et les fonctions nécessaires; préciser leur entête et leur rôle précis.
Identifier les abstractions réutilisables. Ensuite, écrire les corps des abstractions.

## 31.15- Exercice4 : Analyse lexicale

On dispose d'une chaîne de caractères contenant une instruction d'affectation de la forme :

Identificateur = Expression.

L'expression contient une (ou plusieurs) opération d'addition, de multiplication, de division ou de soustraction sur des identificateurs et des nombres. Elle peut contenir des parenthèses. Par exemple

toto = i + 12 - (ind / 15 * k) + x1

On suppose que la chaîne contient une instruction d'affectation correcte et sans erreur. Les opérateurs, les identificateurs et les nombres sont séparés par 0 à n blancs.

Analyser, spécifier puis écrire les procédures et les fonctions nécessaires pour identifier et extraire tous les éléments de cette chaîne et les afficher.

Pour l'exemple ci-dessus, on obtiendrait :

toto = i + 12 - ( ind ..... x1

## *31.15.1--Une solution*

Soit une procédure sauter_blancs qui "consomme" les espaces et s'arrête sur le premier caractère non blanc.

On suppose que *Expression* est une variable globale.

```
Loop
   sauter_blancs(ind);
   IF ind <= Taille_phrase THEN
         Case Expression(ind) Is
            When '+' | '-' | '*' | '/' | '=' | '(' | ')'
                     =>   Put(Expression(ind));
                          ind := ind + 1;

            When 'a' .. 'z' | 'A' .. 'Z'
                     =>   lire_identificateur(ind, Mot);
                          Put(Mot);

            When '0' .. '9'
                     =>   lire_nombre(ind, Mot);
                          Put(Mot);

            When others       =>   Put("symbole inconnu : ") ;
                                   Put(Expression(ind));
                                   ind := ind + 1;
                                  -- on continue la lecture

         End case;
   End IF;
   Exit When (ind > Taille_phrase);
End Loop
```

```
Procedure Sauter_blancs(ind : IN OUT Integer) IS
Begin
   While (ind <= Taille_phrase) ET (Expression(ind) = ' ') Loop
      ind := ind +1 ;
   End LOOP;
End Sauter_blans;

Procedure Lire_identificateur(ind : IN OUT Integer; Mot : Out Tmot) IS
Begin
   Mot := ""      -- la chaîne vide;
   While (ind <= Taille_phrase) ET (Expression(ind)  IN  'a'..'z'
            Or Expression(ind)  IN 'A'..'Z'
                  Or  Expression(ind)  IN '0'..'9')
   Loop
      Mot := Mot & Expression(ind);
      ind := ind +1 ;
   End LOOP;
End Lire_identificateur;

Procedure Lire_Nombre(ind : IN OUT Integer; Mot : Out Tmot) IS
Begin
   Mot := ""      -- la chaîne vide;
   While (ind<=Taille_phrase) AND (Expression(ind)  IN  '0'..'9')
   Loop
      Mot := Mot & Expression(ind); ind := ind +1 ;
   End LOOP;
End Lire_Nombre;
```

# 32- La récursivité

## 32.1- Types récursifs

Un type récursif est défini en faisant référence à lui même.

Exemples :

- Définition du type LISTE

Une **LISTE** est
soit VIDE
soit un Elément suivi d'une **LISTE**.

Cette définition peut être notée par (le formalisme BNF) :

*<Liste> ::= vide | <Elément > <Liste>.*
*<Elément > ::= -- un élément de type quelconque*

- Définition du type CHAINE de caractères

*<Chaîne> ::= vide | caractère <Chaîne>.*

Pour les listes, la définition se fait récursivement.
Pour les chaînes, c'est une question d'interprétation; la définition originelle étant donnée par *string* (tableau de caractères).

- Définition d'arbre binaire :

*<Arbre_bin> ::=vide | <Feuille> |*
*<Feuille> <Arbre_bin> <Arbre_bin>*

*<Feuille> ::= -- une donnée simple*

Les définitions récursives de données constituent un élément important dans l'analyse et l'écriture d'algorithmes manipulant les données récursives. Ces définitions permettent l'emploi des méthodes et des techniques de programmation dirigée par les données (Data-Driven).

## 32.2- Introduction aux abstractions récursives

Dans la définition d'une abstraction, le nom de l'abstraction est visible à l'intérieur de l'abstraction même et peut donc être utilisé. Lorsque l'abstraction "s'appelle" elle-même, on dira que l'abstraction est récursive.
Une abstraction récursive est de la forme :

*Function **F**(....) return .... is*
*Begin*
*.... **F**(....);*
*.....*
*End F;*

*Procedure **P**(....) is*
*Begin*
*.... **P**(....);*
*.....*
*End P;*

Dans le cas ci-dessus, la récursivité est dite *directe*. Une récursivité *indirecte* est de la forme (F --> P --> F) :

*Function **F**(....) return .... is*
*Begin*
*.... **P**(....);*
*.....*
*End F;*

*Procedure **P**(....) is*
*Begin*
*.... **F**(....);*
*.....*
*End P;*

**Exemple** d'abstraction récursive: définition de $x^y$ , $Y \geq 0$

$$x^y = 1 \quad \text{si } y=0$$
$$= x * x^{y-1} \quad \text{si } y > 0$$

```
function power (x ,y : integer) return integer is
begin
   if y = 0 then return 1;
   else return x * power(x,y-1);
   end if;
end power;
```

## 32.3- Quelques exemples d'abstractions récursives

### 1- Fonction factorielle :

```
function FACTORIELLE (N:positive) return positive is
begin
   if N=1 then return 1;
   else return N * FACTORIELLE (N-1);
   end if;
end FACTORIELLE;
```

**Appel :** y := FACTORIELLE(4);

### 2- trouver le maximum d'un tableau T(1..N) d'entiers

```
type Tableau is array (1..M) of integer;
A : Tableau;
procedure max(T : Tableau; N : integer; Max : out integer) is
begin
   if N > 0 then
      if T(N) > Max then   Max :=T(N);
      end if;
      max(T, N-1, Max);
   end if;
end;
```

**Appel** : max(A, 10, Maximum);

## 32.4- Fonctions récursives CAML

**Exemple1** : définition de $x^y$

$$x^y = 1 \quad \text{si } y=0$$
$$= x * x^{y-1} \quad \text{si } y > 0$$

```
let rec power n m = if m=0  then 1 else n* power n (m-1);;
```

**Exemple2** : définition de factorielle : x!

```
let rec fact n = if n=0 then 1 else n*fact(n-1);;
```

p Lors de la définition d'une abstraction récursive CAML, le mot clef **rec** rend le nom de l'abstraction visible à l'intérieur d'elle même.

**Exemple3** : La fonction Fibonacci

```
let rec fib n =
  if n < 2 then 1 else fib(n-1) + fib(n-2);;
fib : int -> int = <fun>
```

**Exemple4** : palindrome en CAML

```
let rec pal s =
       let l = (string_length s)
       in
          if l <=1 then true
          else if nth_char s 0 = nth_char s (l-1)
                then pal (sub_string s 1 (l-2))
              else false;;

pal : string -> bool = <fun>

#pal "serres";;
- : bool = true
```

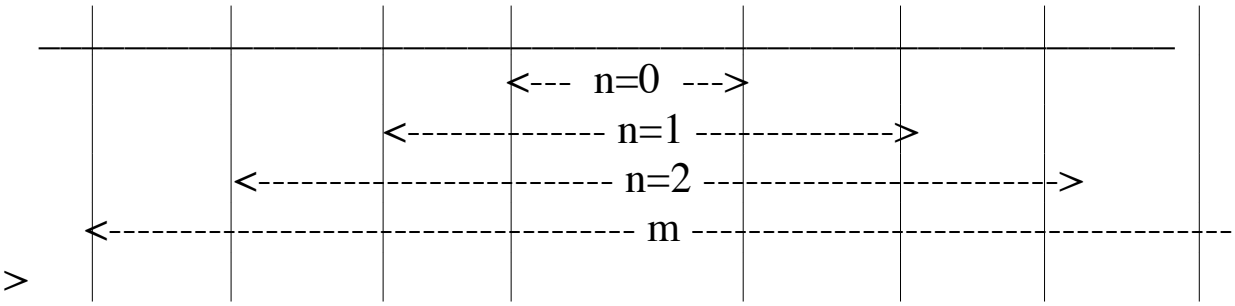
**Exemple5-** PGCD en CAML

```
let rec pgcd x y =
       if x=0 then 0
       else if y=0 then 0
          else if (x > y) then pgcd (x-y) y
              else if x=y then x
                  else pgcd x (y-x);;
```

## 32.5- Récursivité et durée de vie de variables

```
Procedure P is
   m : integer
   procedure R(n : integer) is
         begin
               if n > 0 then R(n-1);
               end if;
         end R;
   begin
         R(2);
   end P;
```

début entrée R(2) entrée R(1) entrée R(0) sortie R(0) sortie R(1) sortie R(2) fin




## 32.6- Analyse récursive

L'analyse récursive est un type important d'analyse qui conduit à un sous-problème similaire au problème initial.

Par exemple, le calcul de la dérivée d'une fonction somme de deux fonctions f et g se décompose en :

- calcul de la dérivée de f
- calcul de la dérivée de g
- calcul de la somme des deux fonctions ainsi obtenues :
  d(f+g) = d(f) + d(g).

On se ramène donc par *récurrence* au même problème portant sur des données plus simples.

La notion d'abstraction (algorithme sous forme d'une fonction ou d'une procédure) fournit un bon outil pour résoudre de tels sous problèmes si l'on admet que le texte d'une abstraction contienne un ou plusieurs appels à elle-même.

**Exemple:**
La définition récursive suivante de factorielle n (n!) :

0! = 1;
n! = (n-1)! * n pour n>= 1

Ce qui donne :

```
function fact(n:integer) return integer IS
Begin
   -- n >= 0, fact(n) = n!
   IF n=0 then return 1 ;
   Else return fact(n-1) * n;
   End if;
End fact;
```

La fonction fact est utilisée dans sa propre définition.
Le calcul de la fonction *fact(n)* nécessite n multiplications : la longueur des calculs n'est donc pas bornée a priori puisqu'elle dépend de n.

## *32.6.1-Décomposition récurrente*

Une décomposition récursive est fondée sur la démarche suivante :

*Résoudre le problème dans le cas général en se ramenant aux cas particuliers où la solution est naturellement simple.*

Plus précisément, pour résoudre un problème, nous suivons le cheminement suivant :

*Trouver au moins un cas où la solution est déjà connue puis tenter de ramener le problème posé pour une valeur quelconque au même problème mais posé sur une valeur plus simple, c'est à dire plus "proche" du cas connu.*

# Exemple :

Spécification textuelle (énoncé) :

Calculer la somme des entiers naturels pairs, compris entre 0 et n inclus.

Solution:

soit la fonction *som_1_n* répondant à la spécification.

Comment la construire?

③On connaît sa valeur pour certaines valeurs de n :

som_1_n(0) = som_1_n(1) = 0

som_1_n(n) n'est pas définie si n n'est pas un entier naturel.

③ Hypothèse à faire pour avancer :

"supposons savoir calculer la valeur de som_1_n(p-1)".

On en déduit la valeur de *som_1_n(p)* et la définition de som_1_n :

| | | | |
|---|---|---|---|
| som_1_n(p) | = | 0 | si p = 0 |
| | = | p + som_1_n(p-1) | si p est pair |
| | = | som_1_n(p-1) | si p est impair |

Ce qui se traduit par :

```
Function som_1_n(p : Integer) Return Integer Is
Begin
   If p<0 Then Stop;       -- arrêt total des calculs
   elsif p=0 Then returne 0;
   Elsif (p mod 2 = 0) Then returne p + som_1_n(p-1);
   Else return som_1_n(p-1);
   Fin If;
End som_1_n;
```

Dans le cas ci-dessus :

- les valeurs concernées sont des entiers.
- la décomposition récursive de la spécification a été obtenue par un raisonnement par récurrence.

## *32.6.2- Principe de récurrence*

*Soit P une propriété (prédicat) sur N*

*1- si P(0) est vérifiée;*

*2- si, de l'hypothèse "p(n-1) est vérifiée", on peut déduire la conclusion "P(n) est vérifiée" alors quel que soit l'entier n, p(n) est vérifiée.*

Ce principe peut être étendu :

*1- si $\exists n_0 \in N$ tel que $P(n_0)$ est vérifiée*

*2- si, de l'hypothèse "p(n-1) est vérifiée et (n-1)>= $n_0$", on peut déduire la conclusion "P(n) est vérifiée" alors quel que soit l'entier n>= $n_0$, p(n) est vérifiée.*

## *32.6.3- Principe de récurrence complète*

*Soit P une propriété (prédicat) sur N*

*1- si P(0) est vérifiée*

*2- si, de l'hypothèse "pour tout k < n, p(k) est vérifiée", on peut déduire la conclusion "P(n) est vérifiée" alors quel que soit l'entier n, p(n) est vérifiée.*

**Un autre Exemple :** calcul du reste de la division de deux entiers naturels a et b sans utiliser la division.

**Spécification :**

- pour b <= a , le reste de la division de a par b est le même que celui de la division de a-b par b, i.e.

  b <= a ⇒ reste(a,b) = reste(a-b, b);

- pour a < b, le reste de la division est égal à a , i.e.

  a < b ⇒reste(a,b) = a

```
function reste(a,b : Integer) Return Integer Is
Begin  -- b > 0
   If b <= a Then Return reste(a-b , b);
   Else Return a;
   End If;
End reste;
```

## *32.6.4-Exemples et traces*

1- Ecrire une procédure récursive qui affiche les valeurs N..1 d'entier : (simulation du schéma “For i In reverse 1..N Loop”).

```
procedure descendant(N : Integer) Is
Begin
   If N > 0 Then
      put(N);
      descendant(N-1);
   End If ;
End descendant;
```

*Trace d'exécution de descendant(4) :*

descendant(4) :
écrire(4) ;
descendant(3) 
descendant(3) :
écrire(3) ;
descendant(2) 
descendant(2) :
écrire(2) ;
descendant(1) 
descendant(1) :
écrire(1) ;
descendant(0) 
descendant(0) :
**sortie**
sortie (plus de commande après l'appel récursif)
sortie
sortie
sortie

On obtient donc dans l’ordre les valeurs 4, 3, 2 et 1.

2- qu'obtient-on de la fonction suivante :

```
procedure descendant(N : Integer) Is
Begin
   If N > 0 Then
      put(N);
      descendant(N-1);
      put("retour au contexte:");  put(N);
   End If ;
End descendant;
```

On obtient :

4 3 2 1 retour au contexte:1 retour au contexte:2...4

3- qu'obtient-on de la fonction suivante :

```
procedure descendant(N : Integer) Is
Begin
   If N > 0 Then
         Put(N);
         descendant(N-1);
         Put("retour au contexte:"); Put(N);
   Else Put("Zéro");
   End If ;
End descendant;
```

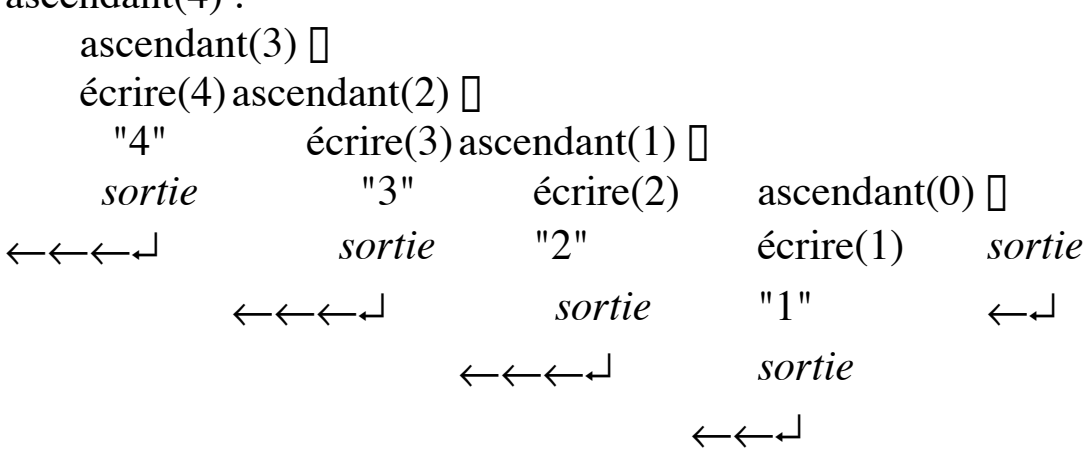
On obtient : 4 3 2 1 Zéro retour au contexte:1 retour au contexte:2...4

4- Redéfinir la procédure pour afficher les valeurs de 1 à N (simulation du schéma "For i in 1..N").

La définition suit la même démarche.

```
procedure ascendant(N : Integer) Is
Begin
   If N > 0 Then
      ascendant(N-1);
      Put(N);
   End If ;
End descendant;
```

Trace d'exécution de ascendant(4) :




5- Ecrire la fonction récursive somme(N) qui calcule la somme des éléments 1 à N (N>=0).

Spécification :

somme(n) = non-définie n < 0
somme(0) = 0
somme(n) = n + somme(n-1) n > 0

```
Function somme(N : Integer) Return Integer Is
Begin  -- N >= 0
   If N = 0 Then Return(0);
   Else Return N + somme(N - 1);
   End If ;
End somme;
```

Une trace :

somme(3) -> 3 + somme(2)
-> 3 + 2 + somme(1)
-> 3 + 2 + 1
-> 3 + 3
-> 6

6- On se donne deux fonctions :

- Tete(chaîne) qui à toute chaîne non-vide, associe son premier caractère;
- Suite(chaîne) qui à toute chaîne non-vide, associe la chaîne privée de son premier caractère;
- Vide(chaîne) qui à toute chaîne vide, associe vrai.

6-1- L'opérateur & permet d'ajouter un caractère en tête d'une chaîne pour produire une nouvelle chaîne. Définir la fonction de concaténation de deux chaînes en vous servant de l'opérateur &.

Spécification :

conc(s1,s2) = s2 Si vide(s1)

conc(s1,s2) = tete(s1) & conc(suite(s1), s2) Si s1 ≠ ""

Remarque : lorsque s1n'est pas vide, on peut réécrie s1 en
tete(s1) & suite(s1)

```
Function conc(s1, s2 : chaine) Return chaine Is
Begin
   If vide(s1) Then      Return s2;
   Else Return tete(s1) & conc(suite(s1)), s2);
   End If ;
End conc;
```

==> Trace de conc("école"," centrale")

6-2- Ecrire la fonction transformant toute chaîne en la chaîne réfléchie (e.g. "leon" en "noel"). Dans un premier temps, on se sert de l'opérateur de concaténation & (avec son profile complet).

Spécification :

reflechie(chaine_vide)=chaine_vide
reflechie(un_carac) = un_carac
reflechie(un_carac & le_reste) =
reflechie(le_reste) & un_carac.

Le premier cas trivial peut couvrir le deuxième.

```
Function reflechie(s : chaine) Return chaine Is
Begin
   If vide(s) Then Return(s);
   Else Return reflechie(suite(s)) & tete(s);
   End If ;
End reflechie;
```

## 32.7- Exercices (énoncés)

0- Que peut-on dire de la fonction suivante :

```
Function vis(x:Integer) Return Integer Is
Begin
   If vis(x) = 1 Then Return (1);
   Else Return(0);
   End If ;
End vis;
```

1- calculer le $k^{ième}$ terme de la suite suivante. La constante *a* est lue au clavier. On dénote $2^n$ par 2 ** n.

$U_0 = a$

$U_{n+1} = 2^n * U_n / n+1$ Si $n \geq 1$

2- Calcul du pgcd, le plus grand diviseur commun de deux nombres M et N.

3- Ecrire une procédure récursive qui dessine un segment reliant deux points de coordonnées (x1,y1) et (x2,y2). On dispose d'une procédure utilitaire *dessine_point(x,y)* qui dessine un point (x,y).
On trace le segment dans le plan XY avec $x_i$ et $y_i \geq 0$.

4- Schéma de Horner :
Calcul de la valeur d'un polynôme de degré N.

Soit $P_n$ un un polynôme de degré N noté par :

$$P_n = a_0 + a_1X^1 + a_2X^2 + .....+ a_nX^n$$

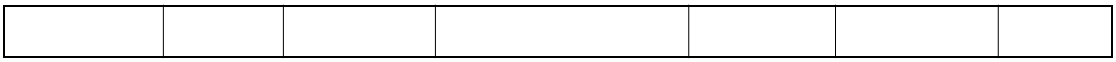
Remarque :
On peut décrire ce polynôme par le schéma général :

$$P_n = X(....(X(Xa_n + a_{n-1})+a_{n-2})....+a1) + a_0$$

5- Le triangle de Pascal : Déterminer de combien de manières distinctes on peut constituer une équipe de P personnes prises dans un ensemble de N joueurs.

Exemple : Soit {A,B,C,D} l'ensemble des joueurs (N=4).
On veut constituer des équipes de deux joueurs (P=2).
Les équipes seront : {A,B} , {A,C} , {A,D}, {B,C}, {B,D} , {C,D}

6 - La marelle de Fibonacci :
Des enfants jouent sur une marelle composée d'une suite de n cases numérotées de n à 1. *n* représente le départ et 1 l'arrivée.




Pour progresser du départ à l'arrivée, ils peuvent sauter à cloche-pied:
- soit d'une case à la suivante (petit saut)
- soit d'une case à la post-suivante (grand saut).

*Combien y a-t-il de parcours distincts entre le départ et l'arrivée sur cette marelle?*

- Pour n=1, on considère qu'il y a un seul parcours à faire (c'est de rester sur la case 1)
- Pour n=2, le seul parcours correspond à faire un petit saut; ce qui nous place dans le cas précédent où le seul parcours est de rester sur place (i.e. pour n=2, il y a un parcours possible);

6- On veut distribuer une prime de N francs entre M employés d'une entreprise. En supposant $e_1 ... e_m$ la série ordonnée qui représente le personnel dans l'ordre de leur ancienneté, la distribution doit tenir compte de cet ordre et le montant affecté à $e_i$ doit être >= à $e_{i+1}$. Certaines primes peuvent être nulles.

En d'autres termes, on veut décomposer un Integer N en M sommants $e_1 ... e_m$ tels que $e_i >= e_{i+1}$. Certains $e_i$ peuvent être nuls.

• Ecrire une fonction pour calculer le nombre de manières de répartir N en M sommants.

• Déterminer le montant affecté à chaque personne.

## 32.8- Solutions

Que peut-on dire de la fonction suivante :

```
Function vis(x:Integer) Return Integer Is
Begin
   If vis(x) = 1 Then Return (1);
   Else Return(0);
   End If ;
End vis;
```

1- calculer le $k^{ième}$ terme de la suite suivante. La constante *a* est lue au clavier. On dénote $2^n$ par 2 ** n.

$U_0 = a$

$U_{n+1} = 2^n * U_n / n+1$ Si $n \geq 1$

```
Function Suite(a,n: Integer) Return Integer Is
Begin
   If n = 0 Then Return(a);
   Else    Return  2 ** n-1 / n * Suite(a, n-1) ;
   End If ;
Fin Suite;
```

Appel : lire(a); lire(k); x := Suite(a,k);

1'- On peut définir $2^n$ par la suite suivante :

$U_0 = 1$ -- $2^0$

$U_n = U_{n-1} * 2$ -- $2^n$ If $n \geq 1$

```
Function puiss2(n : Integer) Return Integer Is
Begin   -- n>=0
   If n=0 Then Return(1);
   Else Return n * puiss2(n-1 );
   End If ;
End puiss2;
```

2- Calcul du pgcd, le plus grand diviseur commun de deux nombres X et Y.

Une spécification du pgcd(x,y) avec x, y ≠ 0 :

pgcd(x,y) = x Si y est multiple de x
pgcd(x,y) = y Si x est multiple de y
pgcd(x,y) = pgcd(y, x mod y) Si x>y
pgcd(x,y) = pgcd(x, y mod x) Si y>x

```
Function pgcd (N,M : Integer) Return Integer Is
R : Integer;
Begin
   R:= N mod M;            -- R reçoit le reste de (N ÷ M)
   If R=0 Then  Return(M);
   Else       Return( pgcd(M,R) );
   End If ;
End pgcd;
```

2'- Autre solution :

pgcd(x,y) = x si y = x
pgcd(x,y) = pgcd(x-y, y) si x>y
pgcd(x,y) = pgcd(x, y-x) si y>x

```
Procedure exemple Is
u,v : Integer;
   Function pgcd(x,y : Integer) Return Integer Is
   Begin
      If (x=0) or (y=0) Then
         Put("pgcd pas défini"); Return (0);
      Elsif x=y Then Return (x);
      Elsif x>y Then Return (pgcd(x-y,y));
      Else Return (pgcd(x,y-x));
      End If ;
   End pgcd;
Begin
   Put("les valeurs de u et de v?");
   get(u); get(v);
   Put("résultat est ");
   Put(pgcd(u,v));
End exemple;
```

3- Ecrire une procédure récursive qui dessine un segment reliant deux points de coordonnées (x1,y1) et (x2,y2) en utilisant la procédure *dessine_point(x,y)* qui dessine un point (x,y). On trace le segment dans le plan XY avec $x_i$ et $y_i \geq 0$

**Spécification :** On dessine le point du milieu, puis pour chaque demi segment, on applique la même procédure.

dessine(x1,y1,x2,y2) =

- Si <x1,y1> *est_voisin* <x2,y2> -- $|x1-x2| \leq 1$ et $|y1-y2| \leq 1$
  dessine_point(x1,y1)
  dessine_point(x2,y2)
- Sinon
  xm = (x1+x2)/2
  ym = (y1+y2)/2
  dessine_point(xm,ym) -- est inutile
  dessine(x1, y1, xm, ym)
  dessine(xm, ym, x2, y2)

Remarque : Il est plus naturel de représenter chaque point par une structure comportant les deux coordonnées. On obtient :

```
Type point IS Record
        x , y : Integer;
    End Record;

Procedure dessine(p1, p2 : point) IS
p : point;
Begin
   If abs(p1.x - p2.x) <= 1 and  abs(p1.y - p2.y) <= 1
   Then
      dessine_point(p1); dessine_point(p2);
   Else
      p.x := (p1.x+p2.x)/2 ;   p.y := (p1.y+p2.y)/2 ;
      dessine(p1, p); dessine(p2, p);
   End If ;
End dessine;
```

## 4- Schéma de Horner :

$P_n = a_0 + a_1X^1 + a_2X^2 + .....+ a_nX^n$

$P_n = X(....(X(Xa_n + a_{n-1})+a_{n-2})....+a_1) + a_0$

$S_n = a'_n + a'_{n-1}X^1 + a'_{n-2}X^2 + .....+ a'_0X^n$

$S_n = X(....(X((Xa'_0 + a'_1)+a'_2)....).....+a'_{n-1}) + a'_n$

|--| $S_0 = a'_0$

|----------| $S_1 = XS_0 + a'_1$

|--------------------| $S_2 = XS_1 + a'_2$

$S_0 = a'_0$

$S_n = X\,S_{n-1} + a'_n$

**Exemple** : 7534 est représenté par : 10*(10*(10*7+5)+3)+4

Où, le nombre étant représenté par un tableau d'entiers A[0..n], X=10, N=3, $a'_0$=7, $a'_1$=5, $a'_2$=3 et $a'_3$=4.

A : tableau[0..3] d'entier;

| 7 | 5 | 3 | 4 |
|---|---|---|---|
| 0 | 1 | 2 | 3 |

k=0 => $S_0$ = $a'_0$ = A[0] = 7

k=1 => $S_1$ = $a'_0X + a'_1$ = 70 + A[1] = 75

k=2 => $S_2$ = $S_1X$ + A[2] = 753

k=3 => $S_n$ = $S_2X$ + A[3] = 7534

Calculer la valeur d'un polynôme de degré N. Les coefficients sont donnés dans un tableau de taille A[0..N]; X est fourni.

```
Type Coef Is Array (0..n) Of Integer;
SubType Base is Integer range ...;
Fonction Horner(A: coef; k : Integer; X : base ) Return  Integer Is
Begin
   If k = 0 Then Return A(0);
   Else Return X*Horner(A, k-1 , X) + A(K);
   End If;
```

**End Horner;**

5- Le triangle de Pascal : Déterminer de combien de manières distinctes on peut constituer une équipe de P personnes prises dans un ensemble de N joueurs.

Exemple : {A,B,C,D}, N=4, constituer des équipes de deux joueurs (P=2).
③ {A,B} , {A,C} , {A,D}, {B,C}, {B,D} , {C,D}

♦ Cas général :
❿ On retire une personne (par exemple A). On calcule le nombre d'équipes de P-1 personnes que l'on peut constituer avec les N-1 joueurs restant. On combinera ces équipes avec A. Ce sont les équipes dont A fait partie.
❡ On retire une personne (par exemple A). On calcule le nombre d'équipes de P personnes que l'on peut constituer dans lesquelles A ne figure pas.
Le nombre total d'équipes est ❿ + ❡.

♦ Cas triviaux : N<P => 0équipe
P=0 ou N=P => 1 équipe

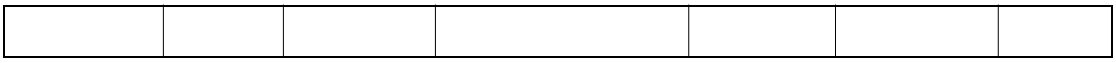
Le modèle mathématique = la formule de triangle de Pascal $C_n^p$ (N>=0, P>=0)
- n<p => $C_n^p = 0$
- p = 0 ou n=p => $C_n^p = 1$
- 0<p<n => $C_n^p = C^p_{n-1} + C_{n-1}^{p-1}$

```
Function triangle (n,p : Integer ) Return Integer Is
Begin  -- n>=0, p>= 0
   Si      n<p Then Return 0;
   Elsif   p=0 ou n=p Then Return 1;
   Else    Return triangle(n-1, p) + triangle(n-1, p-1);
   End If ;
End triangle;
```

Remarque : $C_n^p$ par la fonction factorielle :

$$C_n^p = n!/(p! * (n-p)!)$$

6 - La marelle de Fibonacci : Des enfants jouent sur une marelle composée d'une suite de n cases numérotées de n à 1. *n* représente le départ et 1 l'arrivée.




Pour progresser du départ à l'arrivée, ils peuvent sauter à cloche-pied:

- soit d'une case à la suivante (petit saut)
- soit d'une case à la post-suivante (grand saut).

*Combien y a-t-il de parcours distincts entre le départ et l'arrivée sur cette marelle?*

- Pour n=1, il y a un seul parcours à faire (c'est de rester sur la case 1)
- Pour n=2, le seul parcours correspond à faire un petit saut; puis on reste sur place;
- Pour n>2,Tout parcours sur cette marelle est formé :
  - soit d'un petit saut suivi d'un parcours sur une marelle à n-1 cases ;
  - soit d'un grand saut suivi d'un parcours sur une marelle à n-2 cases ;

Si p(n) représente le nombre de parcours possibles, on a

**n = 1 => p(n) = 1**
**n = 2 => p(n) = 1**
**n > 2 => p(n) = p(n-1) + p(n-2)**

```
Function p(n:Integer) Return  Integer Is
Begin      -- n > 0
   If n <= 2 Then Return 1;
   Else Return  p(n-1) + p(n-2);
   End If ;
End P;
```

Exemple : tracer p(4) -> 3 parcours

6- On veut distribuer une prime de N francs entre M employés d'une entreprise. En supposant $e_1 ... e_m$ la série ordonnée qui représente le personnel dans l'ordre de leur ancienneté, la distribution doit tenir compte de cet ordre et le montant affecté à $e_i$ doit être >= à $e_{i+1}$. Certaines primes peuvent être nulles.

En d'autres termes, on veut décomposer un Integer N en M sommants $e_1 ... e_m$ tels que $e_i >= e_{i+1}$. Certains $e_i$ peuvent être nuls.

• Ecrire une fonction pour calculer le nombre de manières de répartir N en M sommants.

• Déterminer le montant affecté à chaque personne.

**Solution :** soit N le montant restant, M le nombre de personnes encore sans prime et P le montant affecté à la dernière personne traitée.

Analyse et spécification :

prime(P,N,M) =

0 Si (M<1) ou (N < 0)

0 Si (M=1) et (N > P)

car on ne peut pas donner >P au dernier

1 Si (M=1) et (N ≤ P)

on peut donner N au $M^{ième}$

1 Si (M>1) et (N=0)

on peut donner 0 aux $M^{ième} ... 1^{ère}$

X Si (M>1) et (N>0) où

$$X := \sum_{I \in 1..\min(P,N)} \text{prime}(I, N-I, M-1)$$

à chaque pas, on peut donner I au $M^{ième}$

```
Function prime(P,N,M : Integer) Return Integer Is

Begin
   If    (M < 1) or (N < 0)
   Then   Return 0;
   Elsif    M=1
   Then
      If  N > P
      Then  Return 0;
      Else Return 1;
      End If ;
   Elsif  N = 0
   Then Return 1;
   Else
      X := 0;
      Min := minimum(P,N)
      For I In reverse 1..Min Loop
         X := X+prime(I, N-I, M-1);
      End Loop;
   End If ;
End prime;
```

**Appel initial** : prime(N,N,M)

Pour N=10, M=3, on obtient 14 réponses.
Pour N=10, M=5, on obtient 30 réponses.

## 32.9- Procédures récursives

Mêmes principes que les fonctions.

### *32.9.1- Exemple des tours de Hanoï*

Trois pieux *a,b et c*; de *n* disques de diamètres tous différents placés les uns sur les autres par ordre croissant de taille, le plus grand en bas.
Les disques sont initialement placés sur le pieu a. On veut déplacer ces *n* disques sur le pieu *c* en se servant du pieu intermédiaire *b*.

Les règle de déplacement à respecter sont :
- ne déplacer qu'un disque à la fois
- ne poser un disque que sur un disque plus grand.

Ecrire la procédure récursive HANOI(n, dep, arr, inter) qui affiche les configurations successives du jeu.

**Spécification :**

HANOI(N, dep, arr, inter) =
- Si N=0 ==> "rien"
- SI N > 0

❿ Déplacer les N-1 disques qui sont sur le pieu *dep* en utilisant *inter* comme l'arrivé et le pieu *arr* comme pieu intermédiaire;

¶ Déplacer effectivement le N$^{ième}$ disque du pieu *dep* vers *arr;*

① Déplacer les N-1 disques qui sont sur le pieu *inter* (issu de ❿) pour les mettre sur le pieu *arr* en se servant du pieu *dep*

```
Procedure HANOI(n, dep, arr, inter : Integer) Is
Begin
   If n > 0 Then
      HANOI(n-1 , dep, inter, arr);
      put( dep, "-->"); put(arr);
      HANOI(n-1 , inter, arr, dep);
   End If ;
End HANOI;
```

pIl y a $2^n$-1 mouvements corrects (sans se tromper). Cet exemple montre l'intérêt de la récursivité (une solution itérative est difficile à trouver).

## 32.10- Récurrence structurelle

La récurrence structurelle représente la traduction informatique de l'induction structurelle. Elle s'appuie sur la définition (syntaxique) récursive des données.

Nous étudions quelques structures de données récursives. Cette étude sera reprise lors du traitement des listes.

### *32.10.1- Cas des chaînes*

Définition récursive d'une chaîne de caractères.

*<chaîne> :: Chaine_vide | car <chaîne>.*

CH une chaîne non vide = C & CH1 où C=tete(CH) et CH1 = Reste(CH).

Fonction récursive *length* définie sur une chaîne :

• La valeur de *length(chaine_vide)* ne peut être que 0.

Supposons savoir calculer la longueur L d'un chaîne ch, alors la longueur de la chaîne (c & ch) est L+ l.

*le principe de récurrence structurelle* énoncé sur les chaînes:

*Soit P une propriété définie sur les chaînes :*
*1- If P("") est vérifiée*
*2- si, de l'hypothèse "p est vérifiée par* ch*", on peut déduire la conclusion "quelle que soit* c*, P est vérifiée par* (c⊕ch)*" Then quelle que soit la chaîne* ch*, P(* ch*) est vérifiée.*

```
Fonction length(ch : chaîne) Return Integer Is
Begin
   If vide(ch ) Then Return 0;
   Else
      -- ch = c & ch1, ch1 obtenu par une fonction Reste(ch)
      Return 1 + length(Reste(ch));
   End If ;
End length;
```

**Exercice :** A l'aide de :
- Tete(chaîne) qui à toute chaîne non-vide, associe son premier caractère;
- Reste(chaîne) qui à toute chaîne non-vide, associe la chaîne privée de son premier caractère.
- Vide(chaîne) renvoie vrai si *chaine* est vide.

⓾ Ecrire la fonction *membre* qui vérifie si un caractère C figure dans la chaîne CH.

Spécification :

membre(C,CH) =
- Si vide(CH) => faux
- Si non vide(CH) =>
  - Si C = tete(CH) => vrai
  - Si C ≠ tete(CH) => membre(C,Reste(CH))

```
Function membre(C : character; CH : Chaine) Return  boolean Is
Begin
   If CH="" Then Return false;
   Else Return (C=tete(CH) OR membre(C,Reste(CH));
   End If ;
End membre;
```

¶ Ecrire la fonction *occurence* qui compte le nombre d'occurence d'un caractère C dans une chaîne CH.

Spécification :

occurence(C,CH) =
- Si vide(CH) => 0
- Si non vide(CH) =>
  - Si C = tete(CH) => 1 + occurence(C,Reste(CH))
  - Si C ≠ tete(CH) => occurence(C,Reste(CH))

```
Function occurence(C : character; CH : Chaine) Return Integer Is
Begin
   If vide(CH) Then Return 0;
   Elsif C=tete(CH)  Then Return 1+occurence(C,Reste(CH));
   Else Return occurence(C,Reste(CH));
   End If ;
End membre;
```

## *32.10.2- Un exemple ADA de la récursivité sur les chaînes*

Remplacer une sous-chaîne par une autre sous-chaîne dans une chaîne :

```
Function remplace(dans, quoi, par : string) return string is
Begin
   IF dans'length < quoi'length THEN return dans;
   ELSIF
      dans(dans'first .. dans'first+quoi'length-1)=quoi THEN
      return  par &
            remplace(
               dans(dans'first+quoi'length..dans'last),
               quoi, par);
   ELSE
      return  dans(dans'first) &
            remplace(
               dans(dans'first+1..dans'last),
               quoi, par);
   End if;
End remplace;
```

## *Utilisation :*

```
   reponse  :=
      remplace("ada est grand, mais ada n'est  pas simple !!",
            "ada", "ADA") ;
```

ce qui donne :

```
"ADA est grand, mais ADA n'est  pas simple !!"
```

## *32.10.3- Cas des tableaux*

Comme pour les chaînes, on peut considérer un tableau par la définition suivante :

Un tableau d'éléments E est
- soit vide
- soit un élément de type E suivi d'un tableau de E.

Ce qui peut être exprimé par la notation :

*<tableau> :: vide | un_élément <tableau>.*

De ce fait, nous considérons un tableau dont l'identificateur est T par T(inf..sup) où (inf..sup) est un intervalle discret.

Dans ce cas, on a :

Un tableau de E noté T(inf..sup) est
- soit vide (quand inf > sup)
- soit un élément de type E repéré par T(inf) suivi d'un tableau de E noté T(successeur(inf) .. sup). (quand inf ≤ sup)

### *32.10.3.1- Un exemple*

Trouver le minimum du tableau T[1..N] d'entiers.

Spécification

Supposons que la fonction *minimum(X,Y)* renvoie le plus petit entre X et Y.

min(vide) = indéfini
min(T(I..I)) = T(I)
min(T(I..J), J > I = minimum(T(I), min(T(I+1..J)))

```
type tab is array(1..N) of integer;
indefini : exception;

function min(T : tab) return integer is
begin
   if T'first > T'last then raise indefini;    -- tableau vide
   elsif T'first = T'last then return T(T'first);
   esle return minimum(T(I), min(T(I+1..J)));
   end if;
end min;
```

## Appel :

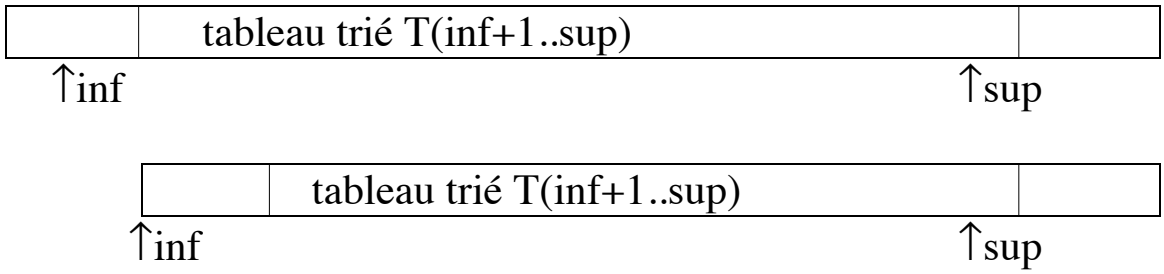
```
   T : tab;
   Begin
   -- remplir T
   put("le minimum est : ");
   put(min(T)); new_line;
   Exception
      when indefini => put_line(" tableau vide ");
      when others => put_line(" problèmes ");
   End appelant;
```

### *32.10.3.2- Exemple de tri par la méthode d'insertion*

Le principe :

pour trier un tableau T(inf..sup), on prélève un élément du tableau, on trie le reste du tableau puis on insère l'élément prélevé à sa place dans le tableau résultant.

Habituellement, l'élément prélevé est le premier (ou le dernier) élément du tableau.




Rappelons qu'un tableau vide (inf > sup) est trié ainsi qu'un tableau à un seul élément (inf = sup).

La procédure principale sera alors :

```
Procedure Tri_ins(T: in out tableau) IS
Begin
   if T'First >= T'Last Then return;
   End if;

DECLARE
   Debut : NATURAL := T'First;
   Fin : NATURAL := T'Last;

Begin  -- de la zone déclare
   Tri_ins(T(Debut+1 .. Fin));        -- trier le reste
   Insere(T(Debut .. Fin));           -- inserer T(Debut) à sa place
                                      -- dans T(Debut .. Fin)
End;                                  -- de la zone déclare
End Tri_ins;
```

La procédure **insere** chargée d'insérer le premier élément du tableau T(inf..sup) à sa place.

Indications :

Lors que le tableau T(inf..sup) est vide ou lorsqu'il contient un seul élément, il n'y a rien à faire. De plus, If l'élément T(inf) est lui même plus petit que T(inf+1) Then on peut déduire que T(inf..sup) est déjà trié en l'état. Dans les autres cas, on permute T(inf) et T(inf+1) et l'on réitère sur T(inf+1..sup).

Ce qui donne l'algorithme suivant :

```
Procedure insere(T: in out tableau) IS
Debut : NATURAL := T'First;
Fin : NATURAL := T'Last;
Begin
   If Debut < Fin  Then
      If T(Debut) > T(Debut+1) Then
         echanger(T, Debut, Debut+1);
         insere(T, Debut+1, Fin);
      End If ;
   End If ;
End insere;
```

## 32.11- Exercices (énoncés)

1- Ecrire une procédure récursive qui affiche les éléments d'un tableau de caractères T(1..N) du premier au dernier. Réécrire l'algorithme pour afficher les caractères de ce tableau en commençant par le dernier.

3- Ecrire un algorithme récursif qui détermine le rang K de la première occurence d'un entier X dans un tableau T(inf..sup) d'Entiers.

4- Ecrire une fonction récursive qui vérifie qu'un tableau T(inf..sup) de réels est ordonné (ordre croissant).

Indications (cas triviaux) :
un tableau vide est ordonné
un tableau à un élément est ordonné

5- Recherche du plus grand élément d'un tableau d'Integers T(inf..sup) par un algorithme récursif.

6- Ecrire une fonction récursive qui, par la méthode dichotomique vérifie If un élément X figure dans un tableau ordonné T(inf..sup).

7- Ecrire la procédure récursive minmax qui calcule à la fois le minimume et le maximume d'un tableau T(inf..sup) d'entiers par la méthode dichotomique.

8- Il peut exister plusieurs définitions pour une structure de données. Par exemple, un tableau peut être défini par :

< tableau> :: vide | élément <tableau>.
ou par < tableau> :: vide | <tableau> élément .
ou par < tableau> :: vide | élément | élément <tableau> .
ou < tableau> :: vide | élément |
élément <tableau> élément .

*Exercice* : en considérant la dernière définition, écrire une fonction pour déterminer si un mot représenté par un tableau de caractères est un Palindrome (radar, elle, tôt, ADA, laval, sus).

*Remarque* : En ADA, les information concernant la borne inférieure, la borne supérieure et la longueur sont disponibles par les attributs *first* , *last* et *length*.

## 32.12- Solutions

1- Ecrire une procédure récursive qui affiche les éléments d'un tableau de caractères T(1..N) du premier au dernier. Réécrire l'algorithme pour afficher les caractères de ce tableau en commençant par le dernier.

```
Procedure Affiche(T:Tableau) Is
Begin   -- du premier au dernier
   If T'first ≤ T'last   Then   Put(T(T'first));
                                Affiche(T(T'first+1 .. T'last));
   End If ;
End Affiche;
```

Autre solution :

```
Procedure Affiche(T:Tableau) Is
Begin      -- du premier au dernier
   If T'first≤ T'last Then   affiche(T(T'first .. T'last-1));
                             put(T(T'last));
   End If ;
End Affiche;
```

Affichage du dernier élément au premier :

```
Procedure Affiche(T:Tableau) Is
Begin   -- du dernier au premier
   If T'first≤ T'last Then   Put(T(T'last));
                             Affiche(T(T'first .. T'last-1));
   End If ;
End Affiche;
```

Autre solution :

```
Procedure Affiche(T:Tableau) Is
Begin      -- du dernier au premier
   If T'first≤ T'last Then   affiche(T(T'first+1 .. T'last));
                             Put(T(T'first));
   End If ;
End Affiche;
```

3- Ecrire un algorithme récursif qui détermine le rang K de la première occurence d'un entier X dans un tableau T(T'first..T'last) d'Entiers.

```
Procedure cherche_rang(T: Tableau; X:Integer;
                Trouve: out booleen; Rang: out Integer) Is
Begin
   If T'first ≤ T'last Then
      If T(T'first) = X  Then    Trouve := true;
                                 Rang := T'first;
      Else cherche_rang(T(T'first+1 .. T'last),X,Trouve,Rang);
      End If ;
   Else Trouve := false;
   End If ;
End cherche_rang;
```

4- Ecrire une fonction récursive qui vérifie qu'un tableau T(T'first..T'last) de réels est ordonné (ordre croissant).

Indications (cas triviaux) :

un tableau vide est ordonné

un tableau à un élément est ordonné

```
Function est_ordonne(T:Tableau;) Return boolean Is
Begin
   If T'first ≥ T'last Then Return true;
   Else -- T'first < T'last    => T'first+1 ≤ T'last
      If T(T'first) ≤ T(T'first+1) Then
            Return est_ordonne(T(T'first+1 .. T'last));
      Else Return false;
      End If ;
   End If ;
End est_ordonné;
```

5- Recherche du plus grand élément d'un tableau d'entiers T(T'first..T'last) par un algorithme récursif.

Spécification :

Maximum(T,Inf,Sup) = indéfini Si Inf > Sup
Maximum(T,Inf,Sup) = T(Inf) Si Inf = Sup
Maximum(T,Inf,Sup) = Sup(T(Inf), Maximum(T,Inf+1,Sup)) Si Inf < Sup

Avec : Sup(x,y) = x Si x> y
Sup(x,y) = y Sinon

```
   N : Constant := ...;
Type Bornes Is range 1..N; -- ... new integer range ...
Type TabEnt IS array (Bornes) of Integer;
Fonction Maximum(T: TabEnt) Return  Integer Is
Begin
   If T'firt > T'last Then Raise Erreur;
   Elsif T'firt = T'last Then Return T(T'first);
   Else M :=  Maximum(T(T'first+1 .. T'last));
        If T(T'first) > M Then Return(T(T'first));
        Else Return M ;
        End If ;
   End If ;
End Maximum;
```

6- Ecrire une fonction récursive qui, par la méthode dichotomique vérifie si un élément X figure dans un tableau ordonné T(T'first..T'last).

Spécification :

Recherche(T,T'first,T'last,X)=
- Si T'first > T'last => faux
- Si T'first = T'last => (X = T(T'first))
- Si T'first < T'last =>
  milieu = (T'first+T'last) /2
  T(milieu) = X => vrai
  T(milieu) < X => Recherche(T, milieu+1, T'last, X)
  T(milieu) > X => Recherche(T, T'first, milieu-1, X)

```
Function Recherche(T: Tableau; X :...) Return  booleen Is
milieu : Integer;
Begin
   If T'first > T'last Then Return  false;
   Else
      milieu := (T'first+T'last) /2
      If T(milieu) = X Then Return true;
      Elsif T(milieu) < X Then
         Return Recherche(T(milieu+1.. T'last), X);
      Else Return Recherche(T(T'first .. milieu-1), X);
      End If ;
   End If ;
End Recherche;
```

8- Il peut exister plusieurs définitions pour une structure de données. Par exemple, un tableau peut être défini par :

< tableau> :: vide | élément <tableau>.
ou par < tableau> :: vide | <tableau> élément .
ou par < tableau> :: vide | élément | élément <tableau> .
ou < tableau> :: vide | élément |
élément <tableau> élément .

**Exercice** : en considérant la dernière définition, écrire une fonction pour déterminer si un mot représenté par un tableau de caractères est un Palindrome

```
Taille_mot : Constant Integer := ... ;
SubType  Mot Is string(1..Taille_mot);
Function Palindrome(M : Mot) Return booléen Is
      --on ne traite pas le cas d'espace dans M
Begin
   If M'first < M'last Then
      If M(M'first) = M(M'last) Then
         Return(Palindrome(M(M'first+1 ..  M'last-1));
      Else Return(False);
      End If ;
   Else Return(True);
   End If ;
End Palindrome;
```

## 32.13- Exemple des télégrammes

Une déscription récursive du problème des télégrammes
*Voir l'énoncé dans le chapitre spécification*

```
Télégramme(nb_mots, nb_mots_longs,n) =
  • lire un mot dans M
  •si M <> 'zzzz'    =>
        - imprimer M
        - faire +1 sur nb_mots
        - si longueur(M) > 20 =>
                faire +1 sur nb_mots_longs
        - Télégramme(nb_mots, nb_mots_longs,n)

  •si M = 'zzzz'      =>
        imprimer nb_mots
        imprimer  nb_mots_longs
        prix = (nb_mots+nb_mots_longs) * n
        imprimer prix
```

### *32.13.1- Schéma global traitant une série de télégrammes*

 On peut envisager l'un des schémas suivants pour la version récursive du traitement d'un télégramme.
 Pour la version itérative, le seul paramètre nécessaire est n : le prix de chaque mot

```
Série =
  Répéter
       télégramme(0,0,n);
  Jusqu'à fin_télégrammes;

Série =
  Tant non fin_télégrammes faire
       télégramme(0,0,n);
  Fin Tq;
```

Série =
  Si non fin_télégrammes alors
    télégramme(0,0,n);
    Série;
  Fin si;

## Algorithme de traitement d'un télégramme :

```
Procedure Télégramme(nb_mots, nb_mots_longs, n : integer) IS
m_long : integer:= nb_mots_longs;
Begin
   lire_mot(M);        -- lecture d'un mot
   If M /= 'zzzz' Then
      put(M);
      If longueur(M) > 20 Then
         m_longs := m_longs + 1;
      End If;
      Télégramme(nb_mots+1, m_longs,n);
   Else
      put("nombre de mots = " & integer'image(nb_mots));
      put("dont " &  integer'image(m_longs) & "mots longs");
      put("prix = " & integer'image (nb_mots+m_longs)*n));
   End If;
End Télégramme;
```

**Appel** : télégramme(0,0,n);

# 33-Récursion terminale et itération

Un schéma récursif terminal est de la forme générale

```
f_rec(X) =
      si p(X) alors a(X)
      sinon
            b(X)
            f_rec(nouveau(X))
      Finsi;
```

p(X) est une propriété à vérifier sur X. Lorsque cette propriété est vérifiée alors un traitement terminal a(X) a lieu (dans lequel X intervient éventuellement) sinon; après un traitement éventuel sur X par b(X), une nouvelle valeur (plus simple rapprochant X à la vérification de la propriété p) est produite avec laquelle la fonction récursive f_rec est rappelée.

Exemple : afficher les caractères d'une chaîne Ch :

```
afficher(Ch) =
      •si Ch="" alors "rien"
      •sinon
            écrire(tête(Ch))
            afficher(Reste(Ch))
```

Il existe un lien entre les schémas de spécification itératifs que l'on a étudié et les fonctions récursives terminales.

Ce type de schéma récursif trouve une traduction simple dans un des schémas itératifs précédents. Cette traduction sera de la forme :

```
f_iter(X) =
  Tant que p(X) = faux faire
      b(X);
      X := nouveau(X);
  Fin Tq;
  a(X);
```

Etant donné l'équivalence entre les schémas Tant que et répéter, on déduit aisément un schéma répéter équivalent :

```
f_iter(X) =
  si p(X) = faux alors
  répéter
        b(X);
        X := nouveau(X);
  jusqu'à p(X)=vrai;
  a(X);
```

On en déduit :
*L'évaluation d'une fonction récursive terminale est une itération.*

**Remarque** :
Nous avons constaté la puissance de l'analyse récursive. On apporte plus facilement la preuve de la justesse d'un algorithme récursif comparé au même objectif dans les algorithmes itératifs. De plus, dans le cas de certaine classe de problèmes (parcours d'arbres, de graphes, ...), l'analyse récursive s'impose naturellement car les solutions itératives sont très difficilement trouvées.
Cependant, l'exécution d'un programme récursif nécessite des ressources plus importantes que celle des programmes itératifs équivalents.
Il faut alors trouver un compromis entre la simplicité de l'analyse et l'efficacité de l'exécution.
Pour ce faire, on utilise les techniques de transformation de la récursivité vers l'itération dont la justesse et la fiabilité a été prouvé.
Etant donné la simplicité de l'analyse récursive dans certains problèmes, une méthode de travail courante pour résoudre ces problèmes (complexes) est de trouver un algorithme récursif à partir d'une analyse récursive puis, par les techniques de transformation, produire une solution itérative plus efficace mais dont la justesse est vérifiée par la justesse du schéma récursif; la transformation étant vérifiée juste.

# 34. Structures récursives

## 34.1. Rappels et compléments sur la récursivité (induction)

*Avantages*

- Analyse naturelle et intuitive de certains problèmes
- Apport de la preuve des algorithmes facilité
- Quasi obligation d'utilisation dans certains problèmes (traitant des structures comme les arbres, les graphes...)

*Inconvénients*

- coûteux en espace mémoire et en temps de calcul.

*Solution aux inconvénients:*

Il existe des techniques fiables de transformation des schémas récursifs en schémas itératifs. Ces techniques sont prouvées justes et complètes. La démarche à suivre est alors :

- d'écrire les algorithmes récursifs
- les tester et valider
- leur appliquer les techniques de transformation pour obtenir une version itérative plus efficace mais dont la validité a été vérifié par l'étape précédente.

### *34.1.1-Exemples simples de transformation*

• La récursivité terminale se transforme simplement en itération.

**Exemple 1** : récursivité terminale

```
procedure p1_rec (i : in integer) is
begin
   if i > 0 then
      action1;      -- ne contenant pas d'appel à p1_rec
      p(i-1);
   else
      action2;      -- ne contenant pas d'appel à p1_rec
   end if;
end p1_rec ;
```

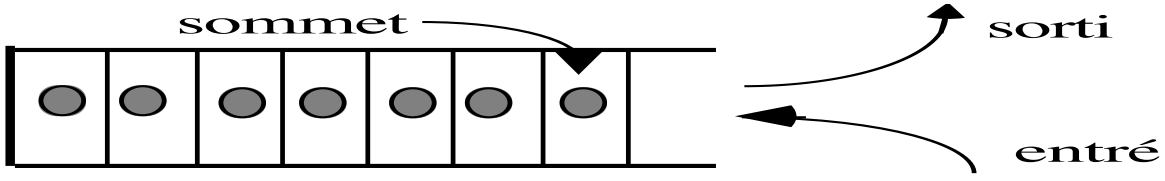
La solution itérative supprime la récursivité terminale.

```
procedure p1_iter (i : in integer) is
j : integer :=i;
begin
   while i > 0 loop
      action1;
      i:=i-1;
   end loop;
   action2;
end p1_iter;
```

**Exemple 2** : récursivité non terminale

```
procedure p2_rec (i : in integer) is
begin
   if i > 0 then
      p(i-1);
      action1;      -- ne contenant pas d'appel à p2_rec
   else
      action2    -- ne contenant pas d'appel à p2_rec
   end if;
end p2_rec ;
```

La version itérative utilise une **pile** (comme une pile d'assiettes) dont le mode de gestion est : *le dernier arrivé est le premier servi*




Les opérateurs habituellement définis sur les piles sont :
- *vide* : la constante pile vide
- *empiler(élément, pile)* : procédure d'ajout d'un élément au sommet ,
- *dépiler(pile, élément)* : procédure d'extraction du sommet,
- *sommet(pile)* : fonction d'interrogation de l'élément du sommet, ...

```
procedure p2_iter (i : in integer) is
j : integer :=i;
pile : pile_entier := vide;
begin
   while i > 0 loop
      empiler(j, pile);
      j:=j-1;
   end loop;
   while pile /= vide loop
      depiler(pile, j);
      action1;
   end loop;
   action2
end p2_iter ;
```

Un autre exemple de transformation est donné dans le chapitre traitant des listes.

### *34.1.2-Exercice*

Trouver une version itérative pour la procédure récursive simulant la boucle "for i in reverse 1..N loop ..." vue précédemment.

## 34.2. La récursivité structurelle

• Il existe deux sortes de récursivité :
- dans le traitement (Ex : la fonction factorielle)
- dans la structure (Ex: les structures tableaux vues sous un angle récursif, les chaînes). Les algorithmes exploitent cette récursivité (Ex. les algorithmes *palindrome*, *recherche d'un car dans une chaîne*, ...)

• En toute rigueur, on peut ramener le récursivité de traitement à celle de la structure. Dans ce cas, les données manipulés doivent être formellement décrites d'une manière récursive.
On peut par exemple définir les entiers par les axiomes de Peano à (l'aide de la constante *zéro* et le constructeur *succ*). Dans ce cas, l'entier 3 est représenté par *succ(succ(succ(0)))*.

• La **récursivité** (induction) **structurelle** est une généralisation de la récursivité (induction). Elle permet la définition et la manipulation des structures de données plus complexes (arbres, graphes).

• Elle facilite l'écriture des algorithmes récursifs sur les structures de données récursives :

Hormis les caractéristiques de la récursivité cités plus haut, on peut décrire la structures de données récursives et **calquer les algorithmes sur cette structures de données**. Il suffit pour cela de préciser la valeur de la fonction (ou d'un paramètre de la procédure) en cours de définition pour chaque cas dans la définition récursive de la structures de données.

**Exemple** : la fonction palindrome récursive

En considérant le tableau de caractères défini comme :

**tableau** est dans l'un des trois cas suivants :

| | |
|---|---|
| - vide | le tableau est vide |
| - caractère | il contient un seul caractère |
| - caractère, **tableau**, caractère | ... |

La fonction palindrome définit une valeur pour chaque cas :

| **Le tableau S** | **La valeur de la fonction palindrome(S)** |
|---|---|
| 1- vide | vrai |
| 2- caractère | vrai |
| 3- $car_1$ *tableau* $car_n$ | ($car_1 = car_n$) et (palindrome(*tableau*)=vrai) |

Ce qui donne la fonction palindrome dans sa version récursive en ADA:

```
Taille_mot : Constant Integer := ... ;
SubType  Mot Is string(1..Taille_mot);
Function Palindrome(M : Mot) Return boolean Is
Begin
   If M'first < M'last Then    -- 3ème cas ci dessus
      If M(M'first) = M(M'last) Then
         Return(Palindrome(M(M'first+1 ..  M'last-1));
      Else Return(False);
      End If ;
   Else Return(True);              -- 1er et 2ème cas ci dessus
   End If ;
End Palindrome;
```

### *34.2.1-Exercice*

Suivre la même démarche pour définir la fonction *max* trouvant le maximum d'un tableau(1..N) d'entiers.

## 34.3. Résumé de la récursivité

On peut appeler une procédure (ou fonction) dans le texte de sa propre définition. Dans ce cas, on dit que la fonction est récursive. Il y a lieu de vérifier qu'une fonction récursive se termine et qu'elle est correcte.

D'une manière générale, la récursivité peut intervenir dans
- la définition de structure (chaîne, liste, arbre....)
- la définition de traitement (algorithme)

La récursivité s'identifie :
lorsque dans la définition d'un objet, un composant fait intervenir l'objet lui même. Cela s'applique à la fois aux données et aux traitements.

La récursivité est par fois le moyen le plus naturel de spécification (de données ou de traitement).

**Exemple** : définition de lien de parenté (sans alliance):
X et Y sont parents
- soit Si Y est père, mère, fils ou fille... de X
- soit s'il existe un individu Z tel que X est parent de Z et Z et parent de Y.

Exprimer la même définition de façon non récursive obligerait à recourir à la notion de "chaîne de parenté" et à introduire dans la définition la longueur d'une telle chaîne.

Soit le type de données Personne.

```
Function parent(x,y : Personne) return boolean is
Begin
   p := false;
   for_all z ∈ famille(x)        -- extension de la notation ADA
      while not p loop
         p := (z=y) or parent(z,y);
      End loop;
   Return p;
End parent;
```

Un cas particulier de définition récursive est la notion mathématique de définition par récurrence (e.g. le triangle de Pascal).

Les algorithmes récursifs partagent un certain nombre de caractéristiques avec "les structures de données récursives" ainsi qu'avec les "définitions récursives";

notamment :

les objets engendrés par une définition récursive (structures de données ou de calculs) doivent être finis pour permettre la terminaison.

En d'autres termes, il faut que des cas triviaux existent.

Un problème se prête particulièrement bien à l'analyse récursive lorsqu'il peut être décomposé en plusieurs sous-problèmes de même type mais de taille plus petite.

Dans l'écriture des algorithmes récursifs, on considère les deux problèmes suivants :

**Terminaison :**

Soit x un paramètre effectif de la fonction f. l'exécution de f(x) provoque un ou plusieurs appels f(y) ou, pour certaines valeurs de x, ne provoque pas d'appel récursif de f. Soit ω l'ensemble de valeurs du paramètre x pour lesquelles l'appel f(x) ne provoque pas d'appel récursif. Pour que f termine, il faut et il suffit que pour tout paramètre effectif x possible, l'on débouche finalement sur l'ensemble ω après un nombre fini d'appels récursifs.

**Correction :**

On recherche le résultat r (dépendant du paramètre effectif x). Pour vérifier que la procédure f calcule r, on montre, par récurrence sur x ou sur le nombre d'appels, que les conditions finales sont vérifiées par x et r.

## 34.4. Les listes et la récursivité structurelle

• Nous avons étudié peu de structures de données récursive à travers les tableaux et les chaînes. On étudie ci-dessous une des plus importante structures récursives : LISTE.

• Il s'agit de représenter une séquence linéaire d'informations S=<e1,e2,....,en>. Cette suite est ordonnée sur les positions (places) des éléments et l'ordre entre les informations mêmes n'intervient pas dans la représentation de S.
La question de représentation de S peut être abordée de deux manières :
1 - utilisation du type TABLE (si la taille n est connue) ou du type LISTE (si n est inconnue). Ceci concerne le niveau abstrait.
2 - utilisation des Tableaux ou des listes chaînées pour la représentation physique.

• Quelque soit le choix au niveau de (1), on peut envisager une implantation par le choix fait en (2).

### Choix du type selon les caractéristiques de S :

• Dans une représentation contiguë (par tableaux), on remarque les points suivants :
- La taille n est connue (au moins bornée) d'avance
- Si $e_i$ est la ième information et $e_{i+1}$ est son successeur, on a :

$$succ^n(e_i) = e_{succ^n(i)}.$$

- L'accès aux informations est directe et simple et l'information $e_{i+1}$ est accessible indépendamment de l'information $e_i$.

• Par contre et dans une représentation chaînée on remarque :
- La connaissance a priori de la taille n n'est pas importante (mémoire supposée illimitée).
- L'information $e_{i+1}$ est accessible via $e_i$. De ce fait, la première place est d'une grande importance et l'accès à l'information $e_n$ se fait par $succ^n$(tête(S)) où succ est une fonction définie sur les positions 1..n-1.

• Des exemples de structures chaînées sont :
- l'accès à l'étage i d'un immeuble passe par les étages précédents,
- dans un jeux (ou un tournoi), les étapes se succèdent en commençant par la première
- dans un cursus d'étude, l'accès à un niveau j nécessite le passage par les niveaux antérieurs, à commencer par le premier....

Dans ces exemples, on suppose la taille n inconnue ou variable.

### *34.4.1- Les listes*

• Une liste est une structure chaînée. Sa définition est donnée par :
Une liste est
(1) soit vide
(2) soit un élément suivi d'une liste.

Les algorithmes s'écrivent facilement en calquant les traitements sur la description récursive des listes. Le cas (1) est la cas trivial (condition d'arrêt) et le cas (2) est le cas général (condition de récurrence).

• Caractéristiques et avantages des listes:
- permettent la manipulation des données de tailles illimitées
- idéal quand la taille d'informations à traiter est inconnue d'avance
- facilitent la définition et la manipulation des structures de données plus complexes (arbres, graphes)

## 34.5. Listes en CAML

• Une liste est une donnée composite qui dénote la séquence <a1,..., an>
• Définition des listes :
liste ::
soit vide
soit un élément puis une liste

• On peut avoir une liste de n'importe quel type (homogène)

• Un minimum d'opérateurs prédéfinis permettent leur manipulation (présentation informelle) :

**[]** = la liste vide
**x :: xs** = cons (x, xs) séparation de la tête et de la queue
**[a; b; ... ; z]** = a:: b :: ... :: z :: nil

• Pour décrire la séquence <1,2,3>, il suffit d'écrire *[1;2;3]* pour que la liste existe. Il n'y a rien d'autre à faire (voir le cas d'ADA plus loin).

## *34.5.1-Exemples sur les listes CAML*

1- fonction affiche d'une liste CAML

```
#let rec affiche = fun
      []        ->     new_line()
   | (t :: q) ->     ecrire t ; affiche q;;

#affiche [1;2;3;4];;
```

==> trace de déroulement

2- fonction somme d'une liste CAML avec trace.

```
#let rec sum = fun
      [] -> 0
   |  (n :: ns) -> n + sum ns;;

sum : int list -> int = <fun>
#sum [1;2;3];;
- : int = 6
```

3- fonction prod d'une liste CAML

```
#let rec prod = fun
      [] -> 1
   |  (n :: ns) -> n * prod ns;;

prod : int list -> int = <fun>
#prod [1;2;3];;
- : int = 6
#prod (1::2::4::[]);;
- : int = 8
```

4- fonction tête d'une liste CAML, la fonction cons (insere_tete), inser_fin
5- fonction queue d'une liste CAML
6- fonction longueur d'une liste CAML

Les exercices suivants montre la nécessité d'avoir **hd** et **tl** (pour éviter la notation par -> ... -> ) :

7- fonction concat d'une liste CAML
8- fonction dernier d'une liste CAML

## *34.5.2-Prédéfinies CAML sur les listes*

• Une collection de fonctions prédéfinies rendent l'utilisation des listes plus simple.

• La notation *fonc : X -> Y* veut dire : la fonction *fonc* prend en entrée un objet de type *X* et renvoie un objet de type *Y*. Les fonctions CAML bien définies renvoient ce genre de descripteur de type. Par exemple, nous avons vu ci-dessus la définition de la fonction sum renvoyant
sum : int list -> int = <fun>
c'est à dire, *sum* prend une liste d'entier en entrée et produit un entier en sortie. De plus *sum* est une fonction (<fun>).

Nous étudierons cette notation dans les TDAs.

Dans les notations ci-dessous, *'a list* veut dire : une liste d'objets de type *a*. Le type *a* est quelconque et pourra devenir par exemple *int*, dans ce cas, *'a list* devient *int liste* (comme dans le cas de la fonction *sum* ci-dessus).

| | | | | |
|---|---|---|---|---|
| **hd** | : | 'a list | -> 'a | (la tête) |
| **tl** | : | 'a liste | -> 'a liste | (la queue) |
| **length** | : | 'a list | -> integer | ( la longueur) |
| **@** | : | 'a list × 'a liste -> 'a list | | (concaténation) |

- En CAML, l'itération est souvent exprimée par l'application de fonctions aux listes. Cette notation abstraite peut être traduite en CAML en utilisant les fonctions d'ordre supérieur.

**Exemple** : Produire une liste d'entiers *de x jusqu'à y* :

```
##infix "jusqua";;          (* directive précédée de # *)

#let rec prefix jusqua = fun x y -> if x > y then [] else x :: ((x+1) jusqua y);;
jusqua : int -> int -> int list = <fun>

#4 jusqua 10;;
- : int list = [4; 5; 6; 7; 8; 9; 10]
```

La notation infixée autorisée grâce à la déclaration *#infix "jusqua"* ci-dessus.

- Définition de la fonction factorielle à l'aide de "jusqu'à" et "prod" :

```
#let factorielle n = produit (1 jusqu'à n);;
factorielle : int -> int = <fun>
#factorielle 5;;
- : int = 120
```

## 34.6. Les listes en ADA

- Avant de voir l'implantation des listes en ADA, on se donne les moyens de manipuler les listes en ADA en étendant la syntaxe d'ADA. On étudiera ensuite la vraie syntaxe de manipulation des listes en ADA.

- Comme dans le cas de CAML, on se donne les opérateurs :
  *vide* : la constante liste vide
  *cons* : ajout d'un élément en tête
  *tête* : interrogation de l'élément de tête
  *queue* : la liste dépourvue de son élément de tête
  *est_vide* : la liste est vide ?.
  *inser_fin* : insertion à la fin de la liste

• Pour déclarer une liste, on écrira par exemple

```
L : liste(entier);          -- L est une liste d'entiers

type élève is record
   ....
end élève;
type liste_élève is liste(élève);
```

• Pour l'affectation, on écrira par exemple :

```
L := <1,2,3>            -- affectation de la listes <1,2,3> dans L
....
```

Ces écriture trouveront leur traduction quand on aura étudié les paquetages et la généricité (comme avec integer_io);

- Les détails d'implantation des listes plus loin (§36).

## *34.6.1-Exercices*

Ecrire les mêmes fonctions vue en CAML en ADA.
**Remarque** : Pour chaque fonction, les versions **Itérative et récursive** sont demandées.

## *34.6.2-Exercice : occurrences avec les listes*

On dispose d'un texte contenant des mots. Les mots peuvent être séparés par un ou plusieurs espaces, un point, une virgule ou le retour chariot....
On veut obtenir le nombre d'occurrences de chaque mot de ce texte.
Pour cela, on utilisera une liste de boites. Chaque boite contient :
- un mot
- le nombre de fois où celui-ci apparaît dans le texte
- les numéros de lignes où celui-ci apparaît dans le texte.

**Discuter** des avantages et des inconvénients de maintenir les listes triées.

# 35. TDA

• Rappel et introduction aux types de données abstraits (TDA).
• Le but : moyen formel de spécification et de définition de structures de données.
• La définition d'un TDA comporte différentes informations :

**sorte :** les type que l'on veut définir (par exemple T)

**utilise :** les type utilisés pour définir T

**opérations :**
le profil des opérateurs définis sur T. Ce sont des fonctions totales et partielles. Une fonctions est notée par :

*nom : type_de_par1 x .... x type_de_par n --> type_du_résultat*

Par exemple, l'opérateur and sur les booléens d'ADA:

*and : booléen x booléen --> booléen*
*not : booléen --> booléen*

La définition du ième élément d'un tableau quelconque d'ADA :

*accès : tableau x indice -/-> élément*

-/-> indique que la fonction est *partielle* car indice doit être une valeur entre les bornes inférieure (inf) et supérieure (sup) du tableau.

Les constantes sont considérés comme des fonction sans argument. Par exemple, les constante booléennes *vrai* et *faux* sont définies par :

*vrai : --> booléen*
*faux : --> booléen*

Le type de la valeur renvoyée par un opérateur peut être le type en cours de définition ou un type mentionné dans la section **utilise**. Dans le premier cas, l'opérateur est une loi de composition interne (appelé opérateur interne), dans le seconde, il s'agit d'une loi de composition externe (appelé opérateur observateur)

**préconditions :** conditions à respecter sur les fonctions partielles.
Par exemple, pour la fonction accès, on écrit
*accès(tableau,indice) est définie si indice ∈ {inf..sup}*

**axiomes :** leur but est de définir les propriétés des opérateurs. Ce sont des "vérités" sur les opérateurs; des équations. On dira par exemple :
*and(vrai, X) = X* *X:booléen*
*and(faux, X) = faux*
not(not(X)) = X

Le symbole "=" ici est utilisé dans le sens équationnel. On peut également se servir des axiomes comme des règles de réécriture. Par exemple, on se sert des axiomes pour démontrer que *not(X) or and(vrai,X)=vrai* quelque soit X : booléen.
Pour décrire les axiomes, on doit systématiquement combiner tous les opérateurs et conserver les combinaisons significatives. Cette combinaison (par fois très longue) est appelée la complétude. Dans la pratique, on combine uniquement les observateurs avec les internes (appelé complétude suffisante). Cependant, certains cas importants de combinaison des opérateurs internes peuvent apporter un éclairage aux définitions et lever certaines ambiguïtés.

## 35.1. Exemple : le TDA entier

**sorte** entier
**utilise :** booléen
**opérateurs :**

```
zéro : --> entier                          --  0
succ : entier --> entier                   --  incrément
pred : entier --> entier                   --  décrément
add : entier x entier --> entier
soust : entier x entier --> entier
mult : entier x entier --> entier
div : entier x entier -/-> entier          -- fonction partielle
égal : entier x entier --> booléen
....
```

**préconditions :** pour les fonctions partielles,
div(x,y) est défini si y≠zéro. x,y : entier
ou encore div(x,y) est défini si not égal(y,zéro).

**axiomes :** pour X,Y : entier
succ(pred(X)) = X
pred(succ(X)) = X
add(0,X) = X
add(succ(X),Y) = succ(add(X,Y))
égal(0,0) = vrai
égal(succ(X),succ(Y)) = égal(X,Y)
...

NB : pour la complétude suffisante, il suffit seulement combiner *égal* avec les autres.

**Remarques sue la gestion des préconditions :**

pour les fonctions partielles, qui teste les préconditions : l'appelant de la fonction ou la fonction elle-même ? Les deux approches ont leurs avantages et inconvénients. Il est souvent difficile pour la fonction elle-même de juger de l'action à entreprendre lorsqu'une précondition n'est pas respectée. De plus, en cas d'erreur, comment doit-elle régler le problème de l'envoi d'un objet du type calculé par la fonction ? D'autre part, l'utilisateur préfère souvent ne pas avoir à tester des valeurs avant d'appeler une fonction.
Au niveau de l'implantation, ce sont les moyens du langage hôte qui permettent de décider. Les langages ayant le traitement des exceptions sont des bons candidats pour la seconde approche.
Les cas d'erreurs sont ainsi gérées du simple affichage d'un message suivi d'un arrêt du programme jusqu'un traitement poussé par les exceptions.

On étudiera plus loin le traitement des préconditions en CAML et en ADA à l'aide des exceptions.

## 35.2. Le TDA Liste simple

**Sort** Liste, Place, Iter_Data
**Utilise** Element, Bool, Entier , *Flot*
**Operations** :

```
Vide :          --> Liste
est_vide :      Liste --> Bool
tete :          Liste -/-> Element
reste :         Liste -/-> Liste
cons :          Element x Liste --> Liste
premier :       Liste -/-> Place
dernier :       Liste -/-> Place
est_dernier :   Liste x Place --> Bool
inser_fin : Element x Liste --> Liste
existe:         Element x Liste --> Bool
recherche       Element x liste --> Place
lg :            Liste --> Entier
nieme       Entier x Liste -/-> Place
nb_occ          Element x Liste --> Entier
concat          Liste x Liste --> Liste
copy            Liste -->  Liste
affiche :       Liste --> Flot
```

**Pour place:**

```
contenu :   Liste x Place --> Element              -- contenu d'une place
modifier :  Liste x Place x Element --> Place   -- Liste modifiée
next :      Liste x Place -/-> Place
before ? :      Liste x Place -/-> Place
valide :        Liste x Place --> Bool      -- Place valide
(éventuellement)
```

**L'itérateur :** (permet les expressions "pour tout X dans L faire Action")

```
init_iterateur : Liste --> iter_Data
itérateur :  iter_Data --> Place
```

L'opérateur "itérateur" modifie son paramètre. On doit donc écrire une procédure à deux paramètres ou bien transformer "itérateur" en :

```
itérateur :  iter_Data --> iter_Data                -- avancer Place_crt
Place_crt : iter_Data --> Place                 -- extraire place_crt
```

**Les preconditions**

tete(l) : non est_vide(l)
reste(l): non est_vide(l)
premier(l): non vide (l)
dernier(l): non est_vide(l)
next(l,p): non est_dernier(l,p)
nieme(i,l) : (i > 0) & (lg(l) >= i)
nieme_place(i,l) : (i > 0) & (lg(l) >= i)
.....

**Les axiomes**

croiser les observateurs et les internes. Conserver les combinaisons significatives.

**Remarques :**

Les cinq premiers opérateurs sont considérés comme le minimum permettant de représenter n'importe quel autre.
La définition de l'itérateur simplifie grandement les écritures. De même, étant donné les définitions concernant Place, il vaut mieux préférer un résultat de type Place à celui de type Element. Ces deux remarques rendent les traductions et l'implantation plus efficaces et plus concises.
Par exemple, étant donné l'itérateur, l'opérateur "existe(L,E)" s'écrit :

*pour tout X dans L*
*si égal(X,E) alors renvoie- vrai fin si*

Ou encore :

*pour tout X dans L* ***jusqu'à*** *trouve*
*si égal(X,E) alors trouve <- vrai fin si*

## *35.2.1- Les variantes des Listes abstraites*

bornée, circulaire, ....

# 35.3. Gestion des préconditions sur les TDA

Le problème de traitement des erreurs

## *35.3.1-Exemple d'exception CAML avec récursivité sur listes*

```
# exception trouve of int;;
exception trouve defined.
# trouve 5;;
-: exn
-= trouve 5
```

- Lever une exception en CAML par le mot clé
  **raise**

```
(* recherche d'une valeur "n" dans une liste : si "n" n'est pas trouvé, on lève
   l'exception*)

#exception pas_dedans;;
#let rec trouve n = fun
   []        ->    raise pas_dedans
   l (x :: l)    -> if x=n then true
                    else trouve n l;;

trouve : 'a -> 'a list -> bool = <fun>
#trouve 1 [12; 3];;
Uncaught exception: pas_dedans
```

- L'exception n'est pas récupérée.
- La récupération :
  **try <expr> with <traitement>**

```
#try trouve 1 [12; 3]
with pas_dedans-> (print_string "n'est pas dans la liste"; true);;
n'est pas dans la liste- : bool = true
```

### *35.3.2-Exemple d'exception ADA avec récursivité sur listes*

Le même principe qu'en CAML. Ayant étudié les exceptions en cours, prévoir dans les exercices sur les listes le traitement des exceptions pour les préconditions.

## 35.4. Exercice : les TDA Tables, Dlistes, File, pile, ....

Proposer un TDA pour :

- les piles (voir "transformations récursivité-itération" mais définir des fonctions)
- les files (comme devant un guichet)
- les tables (comme la table de <élève x note_info>)
- les listes doublement chaînées (Dlistes) : comme les listes mais chaque élément a un successeur et un prédécesseur. le prédécesseur du premier et le successeur du dernier ne sont pas définis.

# 36. Implantation des listes en ADA

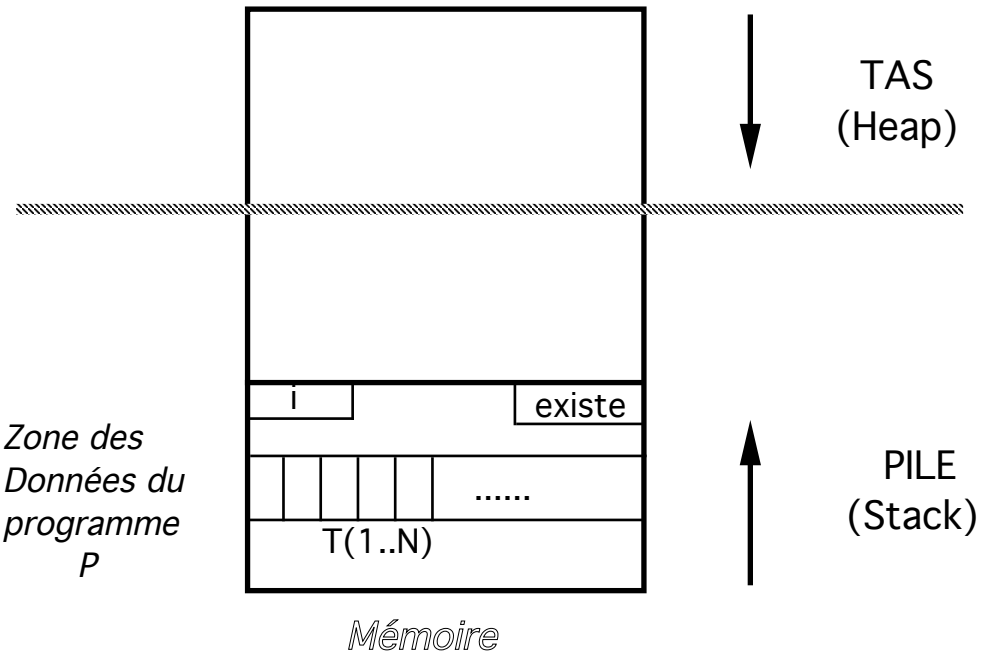
Avant d'étudier cette implantation, le cadre général des structures de données dynamiques est exposé.

## 36.1. Structures de Données dynamiques

### Introduction :

Les objets vus jusqu'à présent sont connus par leur nom (identificateur). Ces objets sont créés lors de l'élaboration de leur déclaration.
Il disparaissent à la fin de l'exécution de l'unité de programme qui les déclare. Leur nom est le seul moyen d'accéder à leur valeur.

```
Procedure P;
   i : integer;
   existe : boolean;
   T : array (1..N) of ...
```




*Mémoire*

Les objets statiques sont stockés dans une zone de mémoire appelée la *pile*. La taille limitée de la pile dans certains systèmes ainsi qu'une limite à la taille des structures de données complexes (tableaux ou enregistrements) sont des contraintes de la création des objets statiques dans un programme.

Ce type d'objet statique ne permet pas de résoudre certains problèmes :

• le traitement d'information dont on ne connaît pas le nombre
• la construction de structure de données plus complexes (listes, arbres,...)

Il arrive par fois que l'on alloue un espace assez grand (un tableau) dont seuls certains éléments sont réellement utiles à un traitement particulier. De plus, l'espace nécessaire aux objets statiques reste occupé pendant la durée d'exécution du programme.

Il existe une autre façon de créer un objet en se servant d'un *allocateur*.

La plupart des langages de programmation offrent, pour cela, un type *accès*. Les objets d'un type accès ont une valeur (une valeur d'accès) qui est un moyen d'accéder à un autre objet d'un type que l'on appelle le type *accédé*.

## Les caractéristiques d'un type accès :

❒ les objets du type accédé peuvent être créés en cours d'exécution sur demande en utilisant l'allocateur *new*. L'exécution de *new* crée une valeur de type accès, c'est à dire, un moyen d'accéder à l'objet créé.

***new integer;*** crée un objet de type entier et retourne une valeur d'un type accès. Cette valeur correspond à l'adresse de l'objet entier créé.

❒ l'objet ainsi créé n'a pas de nom. Un accès (son adresse) à cet objet est donné à sa création. Cette adresse doit être conservée sous peine d'impossibilité d'accéder à l'objet.

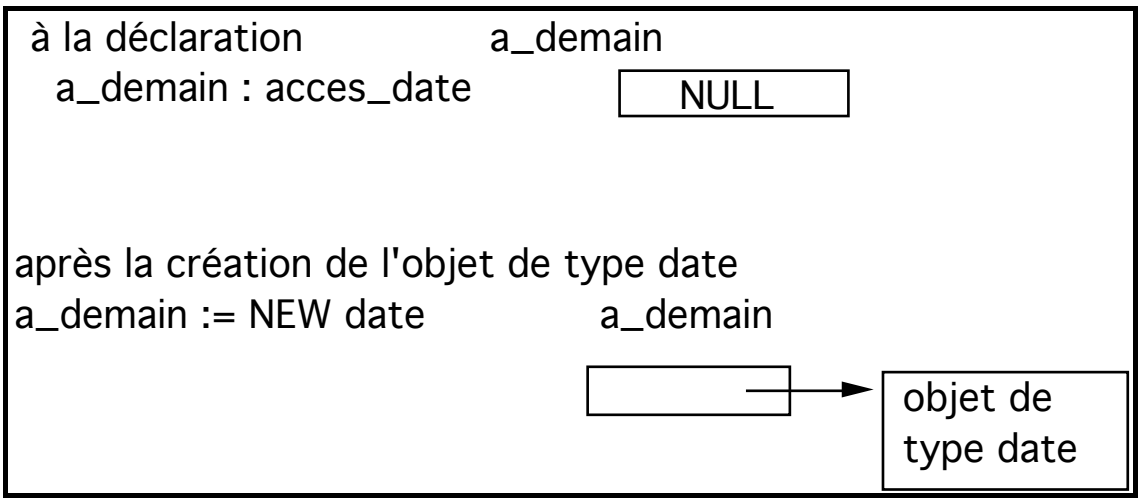
❒ les objets qui conservent ces accès sont typés. Il ne permettent d'accéder qu'à des objets du type prévu à la déclaration du type accès.

❒ les objets créés par accès peuvent être de n'importe quel type.

❒ Habituellement, la destruction d'un objet créé par accès doit être demandée explicitement. Cependant, certains objets dynamiques sont détritus implicitement.

• Toute déclaration d'une variable de type accès initialise cet objet (le pointeur) à la constante NULL. Cette valeur indique que la variable ne contient pas d'accès sur un objet.

```
a_entier1 := new integer;                    -- a_entier1 contient NULL
a_tableau1 := new tab;
a_demain := new date;
a_entier2 := new integer'(45);               -- initialisation à 45
```



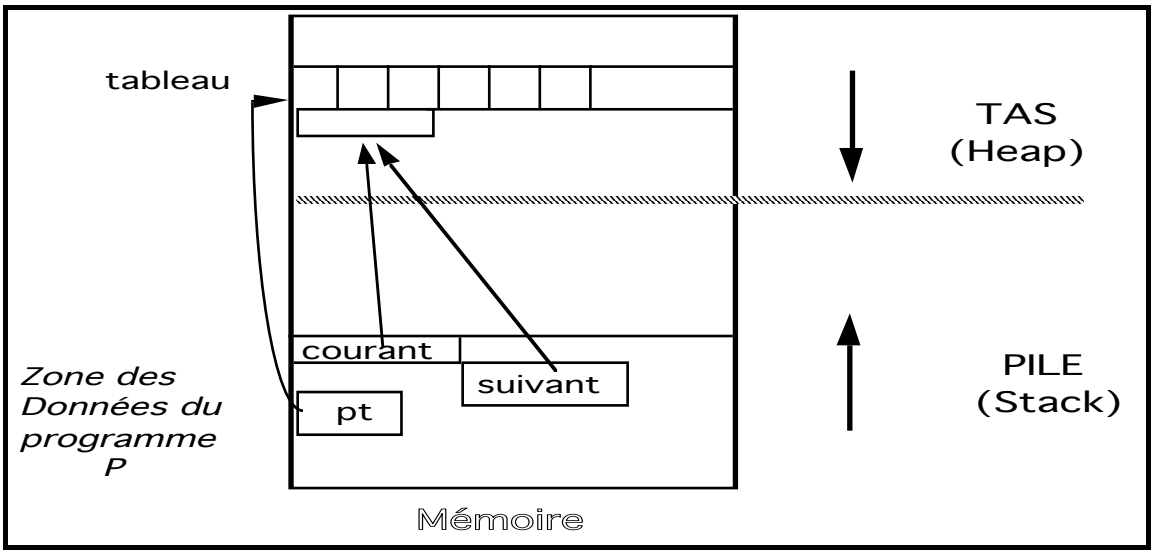

## Accès aux objets créés

• Comme en ADA, un premier niveau de déréférenciation est effectué automatiquement. Le symbole de déréférenciation est représenté par le mot clé *all* pour accéder à l'objet pointé dans sa totalité. Hors mis ce cas, les autres accès n'ont pas besoin de déréférencer l'accès.

L'accès se fait par la notation pointée. L'utilisation du mot clé **all** permet d'accéder globalement à l'objet.

```
a_entier1.all := 20;          -- l'objet accédé par a_entier1 vaut 20
a_hier.all := demain;         -- affectation
put(a_entier1.all);           -- écrit 20
put(a_entier2.all);           -- écrit 45
```

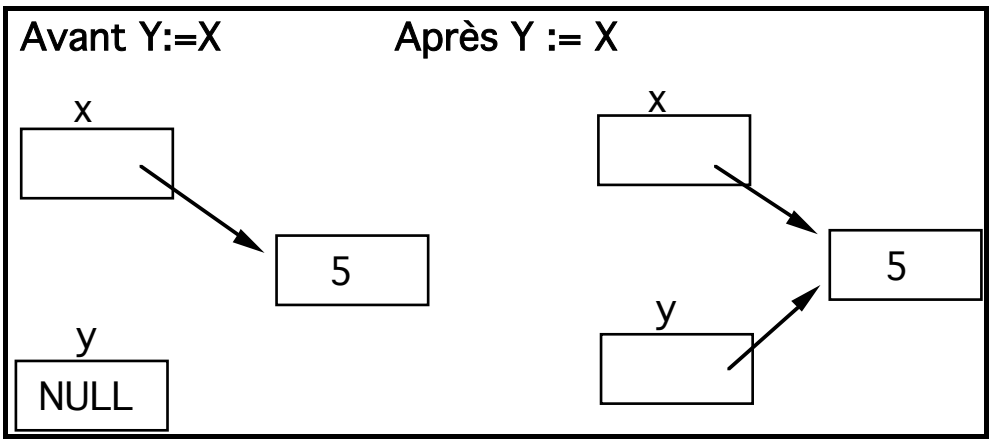
```
Procedure p is
   Type acces_entier is access integer;
   courant, suivant : acces_entier;
         -- ces deux objets permettant d'accéder à des entiers

   Type tab is array (1..N) of integer;
   Type acces_tab is access tab;       -- type pointeur sur tab
   pt : acces_tab;                     -- pt  permet d'accéder à
Begin                                        --  un tableau d'entiers
   courant := new integer;          -- un entier est créé via l'objet courant.
   suivant := courant ;         -- l'entier est aussi accessible via suivant
```




```
   suivant.all := 1;                 -- l'entier accédé reçoit 1
   suivant.all:= suivant.all+1;      -- l'entier accédé contient 2
   pt := new tab ;                   -- pt accède à un tableau d'entiers
   pt(5) := 76;                      -- le 5ème élément du tableau reçoit 76
   pt(4..9) := pt(5..10);           -- décalage d'une tranche
   suivant := NULL;                  -- constante prédéfinie.
End p;                               -- suivant ne pointe plus rien.
```

## *36.1.1-Exemple*

```
Procedure affectation_acces IS
      type acces_entier is ACCESS integer;
   x,y : acces_entier;
Begin
   x := NEW integer;
   x.ALL := 5;
   y := x;
   put(y.ALL);      -- écrit 5
   y.all := 7;
   put(x.ALL);      -- écrit 7
End affectation_acces;
```




## *36.1.2-Avantages et inconvénients des pointeurs:*

• Représentation d'une collection d'informations dont on ne connaît pas la taille;
• Construction de structures de données plus complexes (listes, arbres..);
• Gestion personnalisée plus lourde :
- Création par *new*.
- Libération nécessaire dans la plupart des langages de programmation (certaines implantations d'ADA disposent d'un ramasse-miettes);
• Souplesse dans les ajouts, suppressions... des informations.

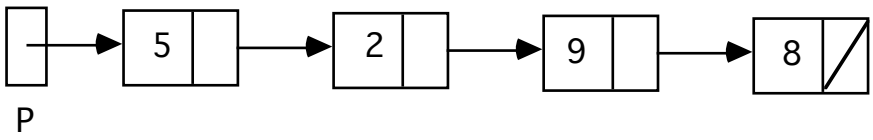
## 36.2. Les listes chaînées

• Une des utilisations importantes des objets par accès est la définition de structures de données récursives. La plus simple de ces structures est appelée **liste chaînée**.

• Une liste est
1- soit vide i.e. NULL
2- soit un élément suivi d'une liste.

On se servira de la définition récursive des listes. Les algorithmes s'écrivent facilement en calquant les traitements sur la description récursive des listes. Le cas (1) sera la condition d'arrêt, le cas (2) la condition de récurrence.

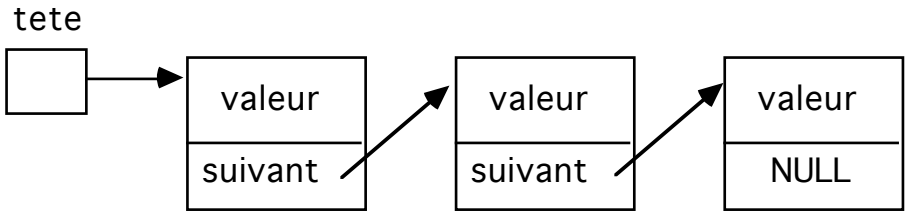
• Exemple d'une liste de 4 entiers dont le premier est accédé par P.




### *36.2.1-Déclaration et utilisation en ADA*

```
Type Boite;      -- déclaration incomplète; ne peut être utilisée
                    -- que dans une déclaration de type accès
Type liste IS ACCESS Boite;

Type Boite IS RECORD       -- déclaration complète
   info : integer;
   svt : liste;          -- récursivité dans la déclaration
End record;

tete : liste;
```

Une liste simplement chaînée sera une suite de cellules. Une liste comporte toujours un accès sur la première cellule (tête).




Déclaration et utilisation de variables :

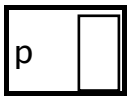
```
l1,l2 : liste;                  -- déclaration des variables l1 et l2
l1:= new liste;                 -- allocation
l1 := null;
l1.info := 546;
l1.svt := null;

l2 := new liste'(125,null); -- allocation et initialisation
```

## *36.2.2-Exemples de listes chaînées*

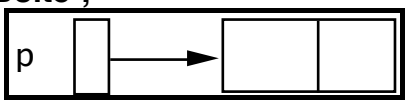
1- Représentation textuelle et construction d'une liste simple (par exemple <5,3,8,2,...>) boîte par boîte.

Première solution simple :

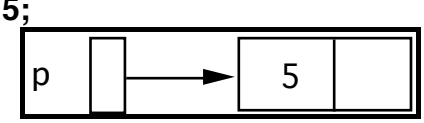
```
Procedure simple(p : out liste) is
Begin
```



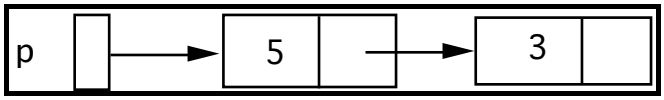

```
p := new Boite ;
```



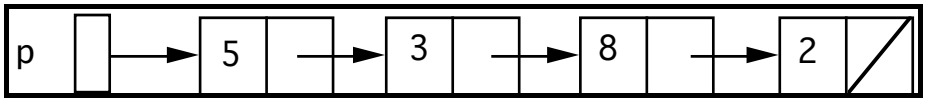

```
p.info := 5;
```




```
p.svt := new Boite'(3, Null);       -- création et affectation
```



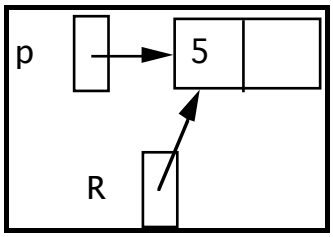
```
p.svt.svt := new Boite; p.svt.svt.info := 8;
p.svt.svt.svt := new Boite; p.svt.svt.svt.info := 2;
p.svt.svt.svt.svt := Null;
```



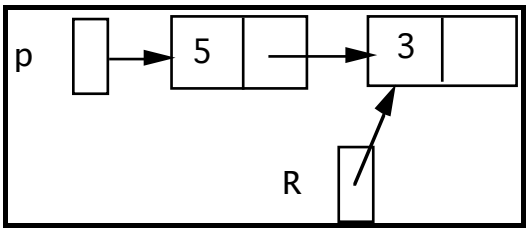

```
End simple;
```

**Amélioration** : utilisation d'un pointeur intermédiaire qui référence la dernière boîte créée:

```
Procedure simple(p : in out liste) IS
R : liste;
begin
   p := new Boite ;
   p.info := 5;
```




```
   R := p;

   ① R.svt := new Boite; R :=R.svt; R.info := 3;
```




```
   ② R.svt := new Boite; R :=R.svt; R.info :=  8;
   ③ R.svt := new Boite; R :=R.svt; R.info :=  2;
   R.svt := NULL;
End Simple;
```

Remarquons que 1, 2 et 3 sont identiques modulo l'information. En outre, on note le traitement réservé à la tête ainsi que l'affectation finale de NULL.

## 2- Application :

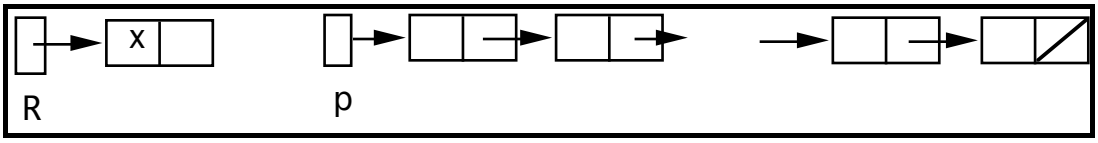
Pour ajouter, supprimer, trier et effectuer d'autres opérations, on veut transférer les informations présentes dans un tableau T(Inf..Sup) à une liste P. Ecrire l'algorithme.

```
Procedure constr(T: tableau...; Inf,Sup : indice ; P : in out liste) is
R : liste;
Begin
   P := NULL;
   If sup >= inf then
      P:= new Boite; p.info := T(inf); R := P;
      i := succ(inf);
      While (i <= Sup) loop
         R.svt := new boite; R := R.svt;
         R.info := T(i); i := succ(i);
      End Loop;
      R.svt := Null;
   End If;
End constr;
```

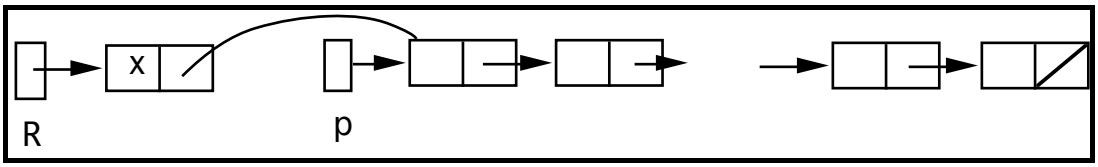
Remarque : On peut parcourir le tableau T de Sup vers Inf. Ce qui nécessite une succession de chaînage (insertion) en tête.

3- Ecrire la procédure *chaîne_tête* qui insère une information X en tête d'une liste P.

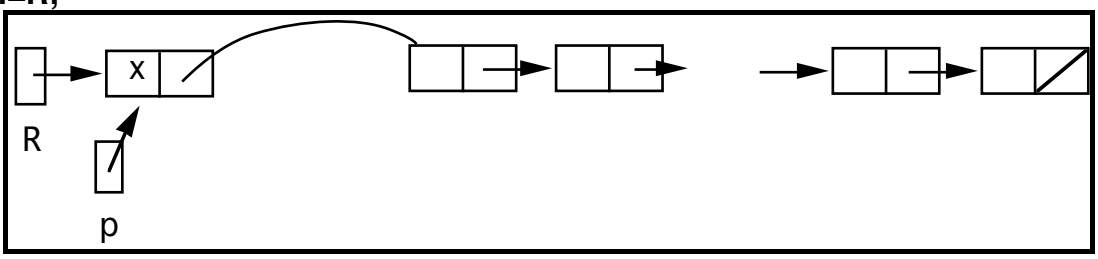
```
Procedure chaîne_tête(X: type_info; P : IN OUT liste) IS
R : liste;
Begin
   R := new Boite;
   R.info := X;
```




```
   R.svt :=P;
```




**P :=R;**




**End chaîne_tête;**

p L'ordre des opérations est importante.

p Plus simple : P := new Boite'(X,P);

p Ambiguïté dans l'accès dans le cas de pointeur sur tableau et pointeur sur record.

4- Ecrire la procédure chaine_queue qui ajoute un élément X à la Fin d'une liste P.

```
Procedure chaine_queue(x:... ; p: IN OUT liste )  IS
R : liste := p;
Begin
   IF p = NULL Then
      p := NEW Boite'(x, Null);
   Else
       While R.svt /= NULL Loop
          R := R.svt;
      End Loop;
      R.svt := NEW Boite'(x, NULL);
   End If;
End chaine_queue;
```

5- Ecrire une procédure qui affiche les information présentes dans une liste dans l'ordre.

```
Procedure Edit(p : liste)  IS
R : liste := P;
Begin
   While R /= NULL Loop
      put(R.info);
      R := R.svt;
   End Loop;
End Edit;
```

## 36.3. Comparaison des structures de données dynamiques-Statiques

| **Dynamique (liste chaînée):** | **Statique (tableau)** |
|---|---|
| - liberté d'allocation, libération | Alloué une fois pour toute |
| - gestion personnelle plus lourd | accès plus rapide |
| - ajout/suppression facile | décalage/tassement |
| - taille illimitée, souple | borné, rigide (fixe) |
| - taille variable, inconnue d'avance | taille bornée |
| - manipulation des structures de | structures de données simples |

données plus complexes (arbres, graphes)

## 36.4. Implantation des listes simples avec des tableaux

proposer une solution pour une implantation avec tableaux/matrices.

# 37. Langages fonctionnels et les pointeurs

- Les langages fonctionnels utilisent beaucoup les pointeurs (comme dans les langages impératifs).

- Une différence majeur est que le programmeur n'a pas accès à ces pointeurs.
  La gestion (comme la création de "noeud(...)" par un allocateur de cellules dans le TAS) est laissée aux techniques avancées de gestion de mémoire.

- Ramasse miettes

- Espace mémoire nécessaire est important

- Le programmeur est libéré de la tâche de gestion de mémoire.
  Il pense seulement à sa méthode et ne s'occupe pas des pointeurs.
  D'autant plus que savoir ou pas si des noeuds de l'arbre sont partagés n'a pas d'effet sur le comportement de son programme fonctionnel.

- Les structures récursive telles que les listes et les arbres peuvent être manipulées avec une efficacité raisonnable dans les langages fonctionnels.

# 38. Exercices sur les listes avec pointeur

**Ecrire les algorithmes RECURSIFS suivants. Etudier leur solutions itératives.**

4- Ecrire une fonction qui calcule la longueur d'une liste.

5- Ecrire une procédure qui affiche les information présentes dans une liste dans l'ordre; puis dans l'ordre inverse.

6- Ecrire une fonction recherche de l'information X dans la liste P.

7- Ecrire la procédure chaine_queue qui ajoute un élément X à la fin d'une liste P.

8- Ecrire la procédure *dechaine_queue* qui supprime le dernier élément d'une liste P.

9- Ecrire La procédure Nth qui Trouve le N$^{ième}$ élément (N ≥1) d'une liste P. Cette procédure renvoie dans R l'adresse de la boîte contenant le N$^{ième}$ élément s'il est présent; NULL autrement.

10- Ecrire la procédure inser_trie qui insère l'élément X à sa place dans une liste triée P.

11- Ecrire la fonction de concaténation de deux listes

12 - Ecrire une procédure permettant de supprimer l'élément X de la liste P.

13 - Ecrire une procédure qui copie les éléments de la liste P dans la liste (nouvelle) R.
On utilise chaine_tete : Elément x Liste --> Liste.

# 39. Solutions

5- Ecrire une procédure qui affiche les information présentes dans une liste dans l'ordre; puis dans l'ordre inverse.

```
Procedure Edit(p : lien) Is
Begin
   If p /= NULL
      put(p.info);
      edit(p.svt);
   End If;
End Edit;
```

Edition inversée des éléments :

```
Procedure Edit_inv(p : lien) IS
Begin
   If p /= NULL Then
         edit_inv(p.svt);
         ecrire(p.info);
      End If;
   End Edit_inv;
```

==> Une solution itérative est plus difficile à obtenir car la récursivité est non-terminale. Une solution récursive s'impose à moins d'inverser la liste d'abord!!.

Une solution itérative à l'aide d'une pile :

```
Procedure Edit_inv_iteratif(R : lien) is
P : lien := R; pile : pile_d_info := vide;
Begin
      Initialiser_pile;
      While p /= NULL loop
            empiler(p.info,pile); p:=p.svt;
      End Loop;
      While  pile /= Null Loop
            depiler(pile,x); ecrire(x);
      End Loop;
End Edit_inv;
```

6- Ecrire une fonction recherche de l'information X dans la liste P.

```
Fonction recherche(x:.. ; p: lien) return boolean IS
Begin
   If p = NULL Then Return False;
   Else Return   (p.info=x) OR recherche(x, p.svt);
   End If;
End recherche;
```

==> possibilité de récupérer l'adresse de la boîte contenant X....

7- Ecrire la procédure chaine_queue qui ajoute un élément X à la fin d'une liste P.

```
procedure chaine_queue(x: ... ; p: in out lien ) =
Begin
   If p = NULL Then
      p := new boite; p.info := x; p.svt := NULL;
   Else chaine_queue(x,p.svt);
   End If;
End chaine_queue;
```

8- Ecrire la procédure *dechaine_queue* qui supprime le dernier élément d'une liste P.

```
Procedure dechaine_queue(p : in out lien) =
Begin
   If p /= NULL Then
      If p.svt = NULL Then p := NULL;
      Else dechaine_queue(p.svt);
      End If;
   End If,
End dechaine_queue;
```

==> possibilité de récupérer l'adresse de la dernière boîte....

9- Ecrire La procédure Nth qui Trouve le N$^{ième}$ élément (N ≥1) d'une liste P. Cette procédure renvoie dans R l'adresse de la boîte contenant le N$^{ième}$ élément s'il est présent; NULL autrement.

```
Procedure Nth(P: lien; N: Integer; R: out lien) IS
Begin
   If (P=NULL) OR (N=1) Then R :=P;
   Else nth(P.svt, N-1, R);
   End If;
End Nth;
```

9'- Ecrire une fonction qui compte le nombre d'occurrences d'un élément X dans une liste P.

```
Function nb_occ(X: ... ; P : lien) return Integer is
Begin
   If P=NULL Then Return 0;
   Elsif p.info=X Then Return 1+nb_occ(X,P.svt);
   Else Return nb_occ(X,P.svt);
   End If;
End nb_occ;
```

10- Ecrire la procédure inser_trie qui insère l'élément X à sa place dans une liste triée P.

```
Procedure inser_trie(X: ... ;P : in out lien) IS
R : lien;
Begin
   If P=NULL Then p := new boite'(X, NULL);
   Elsif P.info ≥ X Then R := new Boite'( X,P) ; P := R;
   Else inser_trie(X,P.svt);
   End If;
End inser_trie;
```

11- Ecrire la fonction de concaténation de deux listes

```
Function concat(l1 : in out : lien;  l2 : lien) return lien IS
Begin
   If l1=NULL Then Return l2;
   Else l1.svt := concat(l1.svt, l2);
         Return l1;
   End If;
End concat;
```

12 - Ecrire une procédure permettant de supprimer l'élément X de la liste P.

```
Procedure supprime(X:...;P: in out lien) IS
Begin
   If P /= NULL Then
      If P.info = X Then
         P := P.svt;
      Else supprime(X, P.svt);
      End If;
   End If;
End supprime;
```

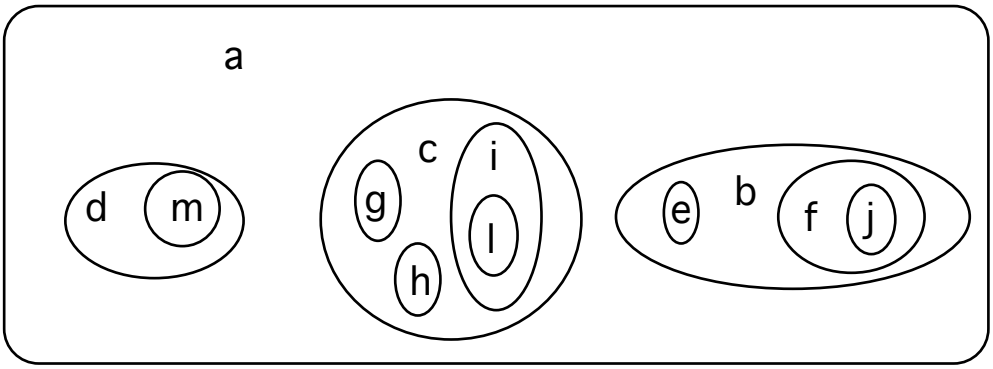
13 - Ecrire une procédure qui copie les éléments de la liste P dans la liste (nouvelle) R. On utilise chaine_tete.

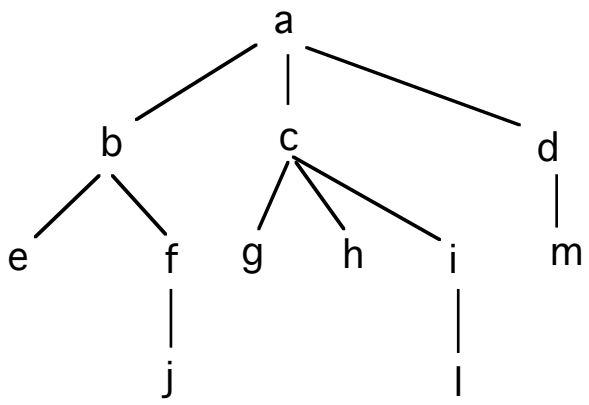
```
Procedure copie(P: lien; in R: out lien) IS
Begin
   If P = NULL Then R := NULL;
   Else
      copie(P.svt, R);
      chaine_tete(P.info, R);
   End If;
End copie;
```

# 40. Les Arbres

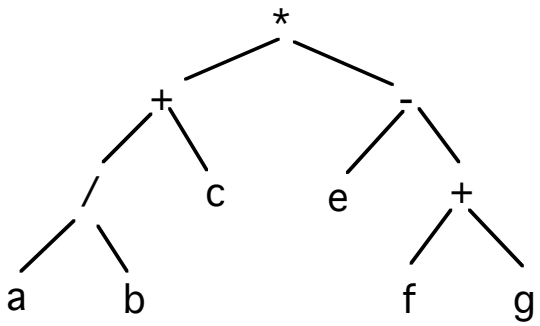
Moyen de structuration et de représentation hiérarchique.

Exemples :




L'expression (a/b+c) * (e-f+g)



## Quelques définitions

- noeud, racine
- descendant (fils, fille), ascendant (père, mère)
- sous-arbre
- feuille
- arbre n-aire, binaire
- hauteur (niveau)
- mot des feuilles
- ....

## Définition et déclaration d'un arbre binaire :

Un arbre est
- soit vide (NULL)
- soit un élément , un arbre (gauche) et un arbre (droit)

```
Type   noeud;
Type   lien IS access noeud;
Type   Noeud IS record
            info : type_d_information;
            gauche,
            droit : lien;
       End record;

Arbre : lien;
```

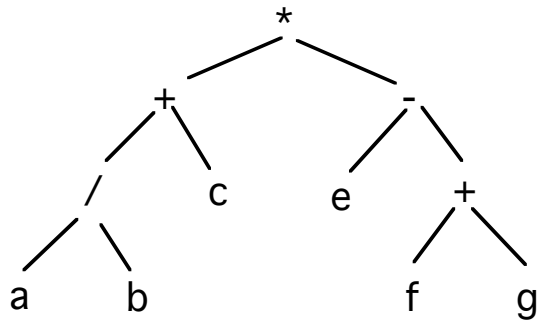
==> La constante NULL joue le même rôle que pour les listes.

**Différents types de parcours d'arbre binaire:**

Soit :
R = Racine
D = sous-arbre Droit
G = sous-arbre Gauche

Il y a trois types (principaux) de parcours :
- **Préfixé** : R G D -- pré-ordre
- **Infixé** : G R D
- **Postfixé** : G D R -- post-ordre

Exemples :




Les noeuds visités selon le mode de parcours :

- **Préfixé** : *+/abc-e+fg
- **Infixé** : a/b+c*e-f+g ==> (a/b+c) * (e-f+g)
- **Postfixé** : ab/c+efg+-*

On remarque que lors d'une évaluation de l'expression, le mode *infixé* nécessite des parenthèses pour lever des ambiguïtés.

Les algorithmes de parcours :

```
Procedure Prefixe ( R : lien) is
Begin
   If R /= NULL Then
p     traiter(R);                    -- R
      Pefixe (R.gauche);                    -- G
      Pefixe (R.droit);                     -- D
   End If;
End Préfixe ;

Procedure Infixe ( R : lien) is
Begin
   If R /= NULL Then
      Infixe (R.gauche);
p     traiter(R);          -- un traitement quelconque
      Infixe (R.droit);
   End If;
End Infixe ;

Procedure Postfixe ( R : lien) is
Début
   If R /= NULL Then
      Postfixe (R.gauche);
      Postfixe (R.droit);
p     traiter(R);
   End If;
End Postfixe ;
```

## 40.1. Exercice : le TDA arbre binaire

proposer un TDA pour les arbres binaires.

## 40.2. Exercices sur les arbres

Quelques algorithmes sur les arbres binaires :

1- Ecrire la fonction *taille* qui calcule le nombre de noeuds d'un arbre binaire.

2- Ecrire la fonction *feuille* qui test si un noeud d'un arbre binaire est une feuille.

3- Ecrire la fonction *nb_feuilles* qui calcule le nombre de feuilles d'un arbre binaire.

4- Ecrire la fonction *hauteur* qui calcule la hauteur d'un arbre binaire.
NB : La hauteur est définie par le maximum de nombre d'arcs allant de la racine jusqu'à toute feuille + 1.

5 - Ecrire la fonction *recherche* qui cherche l'élément X dans un arbre binaire.

6 - Ecrire la fonction *isomorpohe* qui vérifie si deux arbres contiennent les mêmes informations dans le même ordre.

**Quelques algorithmes sur les ABOH:**

7- Ecrire la fonction de recherche d'un élément X dans un ABOH.

8- Ecrire la procédure d'insertion d'un élément X dans un ABOH.

## 40.2'. Solutions

1- Ecrire la fonction *taille* qui calcule le nombre de noeuds d'un arbre binaire.

```
Function taille(R : lien) return Integer IS
Begin
   If R = NULL Then Return 0;
   Else  Return 1+taille(R.gauche)+taille(R.droit);
   End If;
End Taille;
```

==> le type du parcours

2- Ecrire la fonction *feuille* qui test si un noeud d'un arbre binaire est une feuille.

```
Function feuille(R : lien) return boolean IS
Begin
   If R = NULL Then Return False;
   Else  Return (R.gauche=NULL) and (R.droit=NULL);
   End If;
End feuille;
```

==> le type du parcours

3- Ecrire la fonction *nb_feuilles* qui calcule le nombre de feuilles d'un arbre binaire.

```
Function nb_feuille(R : lien) return Integer IS
Begin
   If R = NULL Then Return 0;
   Elsif  feuille(R) Then Return 1;
   Else Return nb_feuille(R.gauche) + nb_feuille(R.droit);
   End If;
End nb_feuille;
```

4- Ecrire la fonction *hauteur* qui calcule la hauteur d'un arbre binaire.

NB : La hauteur est définie par le maximum de nombre d'arcs allant de la racine jusqu'à toute feuille + 1.

```
Function hauteur(R : lien) return Integer IS
Begin
   If R = NULL Then Return 0;
   Elsif  Return Max(hauteur(R.gauche) , hauteur(R.droit)) +1 ;
   End If;
End hauteur;
```

5 - Ecrire la fonction *recherche* qui cherche l'élément X dans un arbre binaire.

```
Function recherche(X: type_info; R : lien) return boolean IS
Begin
   If R=NULL Then Return False;
   Elsif R.info = X Then Return True;
   Else Return
        recherche(X, R.gauche) OR recherche(X, R.droit);
   End If;
End recherche;
```

==> quel type de parcours à choisir

6 - Ecrire la fonction *isomorpohe* qui vérifie si deux arbres contiennent les mêmes informations dans le même ordre.

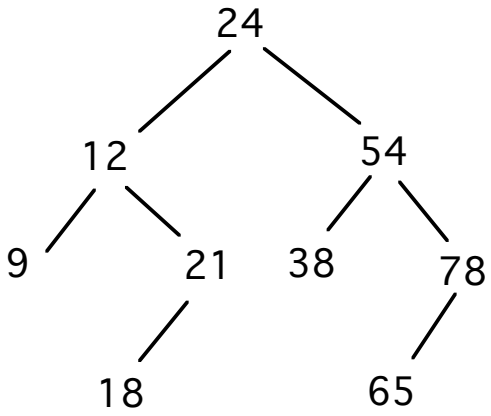
```
Function isomorphe(R1, R2 : lien) return boolean IS
Begin
   If R1 = NULL et R2 = NULL Then Return True;
   Else If R1 /= NULL et R2 /= NULL Then Return
           (R1.info = R2 .info) AND
           isomorphe(R1.gauche, R2.gauche) AND
           isomorphe(R1.droit, R2.droit);
        Else Return False;
        End If;
   End If;
End isomorphe;
```

## 40.3. Variante : Arbre Binaire Ordonné horizontalement (ABOH)

Relation vérifiée sur chaque noeud d'un ABOH:

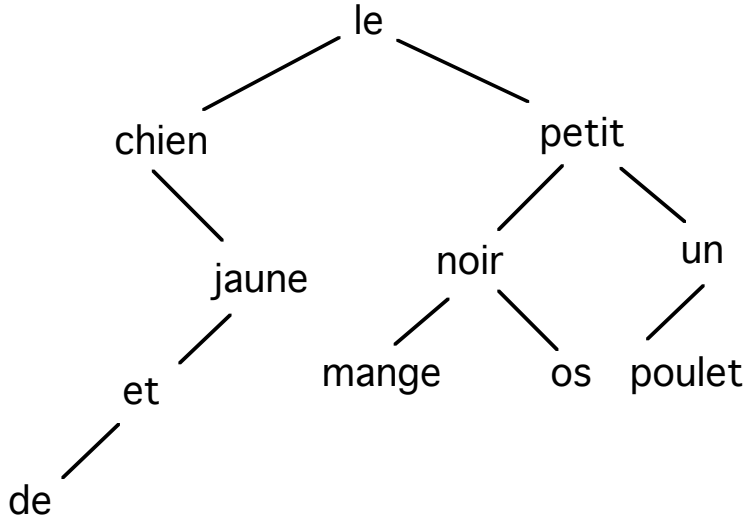
Max(info(ss-arbre-G)) < info(R) < Min(info(ss-arbre-D))

Exemple :




**Intérêts** : Structure ordonnée facilitant les recherches et les insertions.

==> "le petit chien jaune et noir mange un os de poulet"




Question : Quel type de parcours pour reTrouver les informations dans l'ordre ?

## *40.3.1- Quelques algorithmes sur les ABOH*

7- Ecrire la fonction de recherche d'un élément X dans un ABOH :

```
Function recherche (X : type_info; R : lien) return boolean IS
Begin
   If R=NULL Then Return False;
   Elsif R.info = X Then Return True;
   Elsif X < R.info Then Return recherche(X, R.gauche)
   Else Return recherche(X, R.droit);
   End If;
End recherche;
```

Complexité ≈ N (N = la hauteur de l'arbre) ou
logM (M = nombre de noeuds)

Version itérative

```
Function recherche (X : type_info; R : lien) return  boolean IS
Trouve : boolean := False;
Begin
   While R /= NULL AND not Trouve loop
      If R.info = X Then Trouve := True;
      Elsif X < R.info Then R := R.gauche;
      Else R := R.droit;
      End If;
   End loop;
   Return Trouve;
End recherche;
```

8- Ecrire la procédure d'insertion d'un élément X dans un ABOH.

```
Procedure insere (X : type_info; R : in out lien)  IS
Begin
   If R=NULL Then
      R := new Boite; R.info := X;
      R.gauche := NULL; R.droit := NULL;`
   Elsif X < R.info Then insere(X, R.gauche)
   Else insere(X, R.droit);
   End If;
End insere;
```

## 40.3.2-Exemple ABOH en ADA

```
-- ecrire un programme qui lit une série des chaînes au clavier et les insère
-- dans un ABOH puis affiche le contenu de l'arbre dans l'ordre

WITH text_io; USE text_io;
PROCEDURE arbre IS
TYPE noeud;
TYPE lien IS access noeud;
TYPE noeud IS record
    info : string(1..10);
    gauche,droit : lien;
END record;
arbre : lien := NULL;
ch : string(1..10);
l : integer;

PROCEDURE insere(e : string; arbre : in out lien) IS
BEGIN
    IF arbre = NULL THEN
        arbre := new noeud'(e,null,null);
     ELSIF arbre.info > e THEN insere(e,arbre.gauche);
     ELSE insere(e,arbre.droit);
    END IF;
END insere;
```

```
PROCEDURE affiche(arbre : lien) IS
BEGIN
    IF arbre /= NULL THEN  affiche(arbre.gauche);
                           put(arbre.info);
                           affiche(arbre.droit);
  END IF;
END affiche;

BEGIN
    LOOP
        ch := (others => ' ');
        put("donner une chaine (< 10 caracteres) ");
        get_line(ch,l);
        insere(ch,arbre);
    END LOOP;

    EXCEPTION
        when END_ERROR => affiche(arbre);
        when OTHERS =>     put_line("probleme de lecture ");
                           affiche(arbre);
END arbre;
```

## 40.3.3-Exemple ABOH en CAML

Exemple : insertion dans un ABOH d'entiers :

```
type arbre = nul | noeud of (arbre * int * arbre);;

let rec insere = function
      (new , nul) -> noeud (nul, new, nul)
   |  (new , noeud (gauche , racine , droit)) ->
         if new <= racine then
         noeud(insere (new, gauche), racine, droit)
      else noeud (gauche, racine, insere(new, droit));;
```

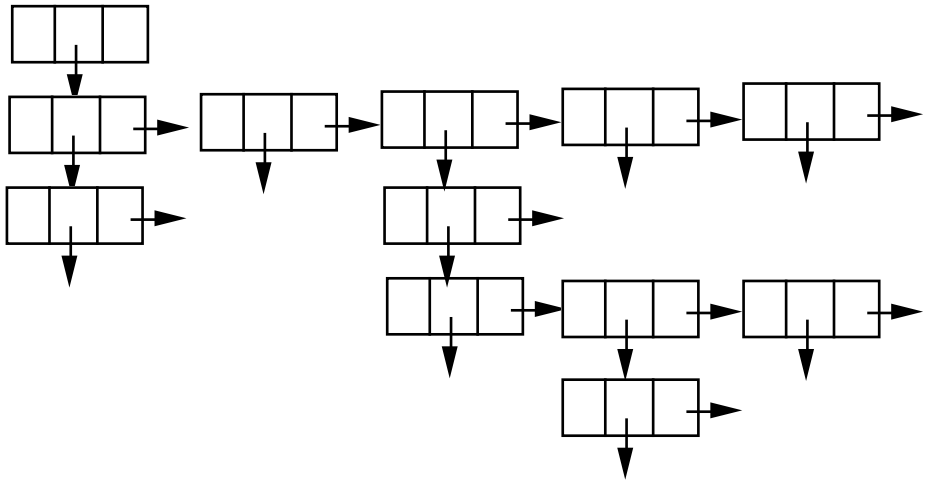
Exemples d'utilisation :

```
#insere (1, nul);;
- : arbre = noeud (nul, 1, nul)
#insere(1, noeud(nul, 2, nul));;
- : arbre = noeud (noeud (nul, 1, nul), 2, nul)
```

- On copie seulement les noeuds sur le chemin racine -> noeud inséré.

## 40.3.4- Implantation des arbres N-aires

- Si N est connu d'avance
- Si N n'est pas connu d'avance ==> forêt



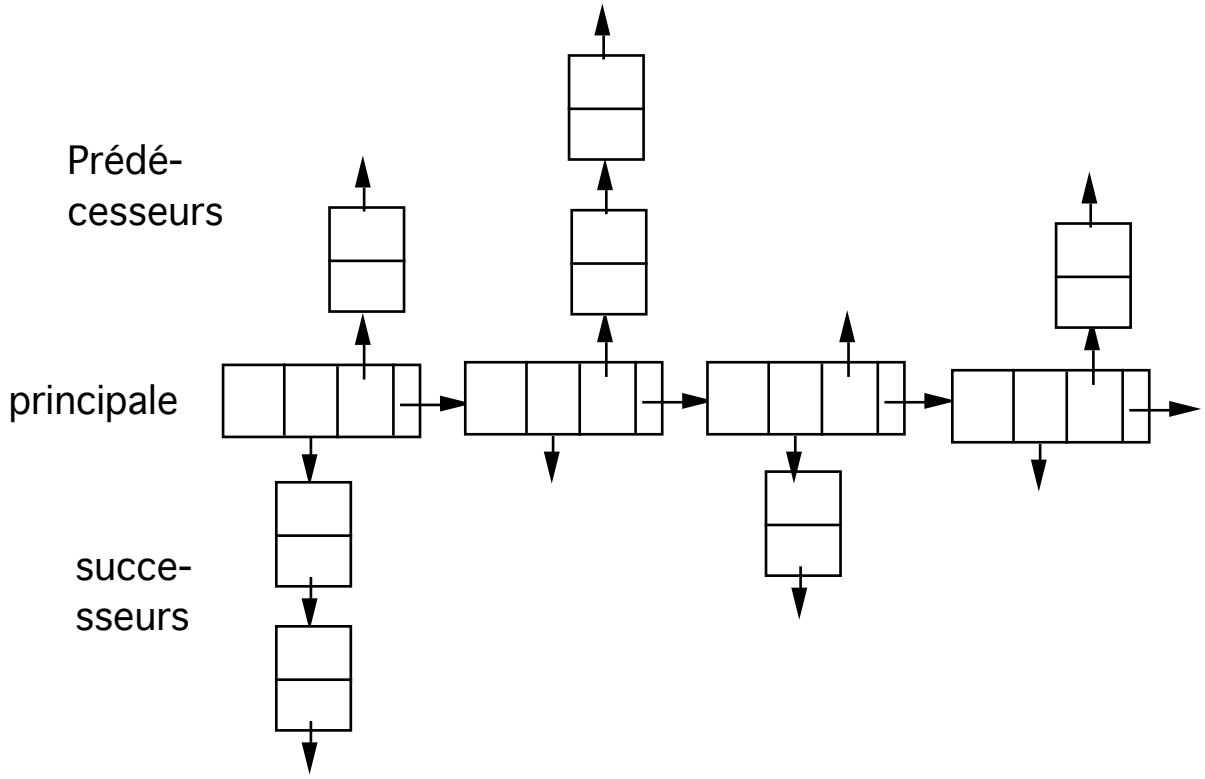
Chaque Noeud contient :

- Une information
- Un pointeur vers le premier fils
- Un pointeur vers la liste de ses frères

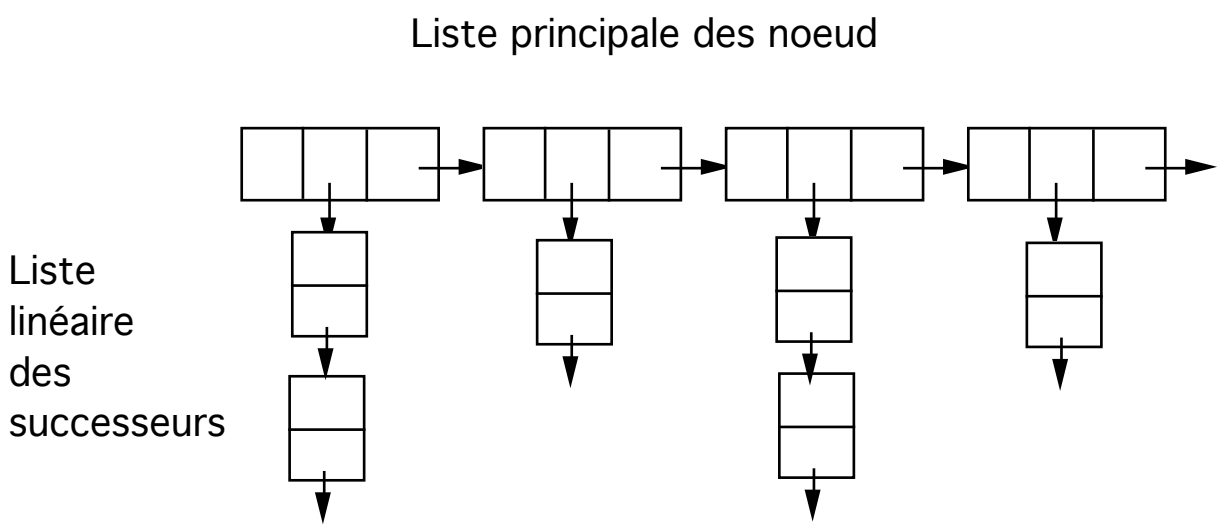
p Le père ne conserve qu'un seul accès sur son premier fils

**Implantation des graphes**

Forme généralisé :

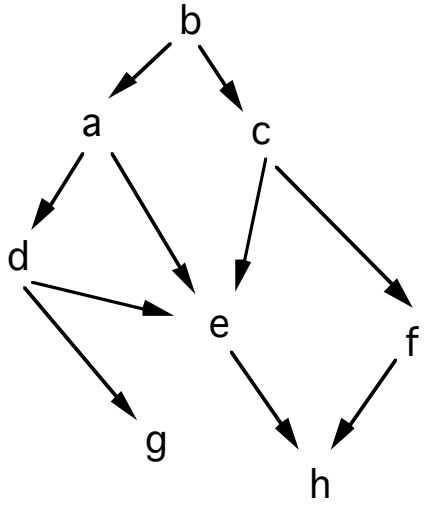



Forme réduite :

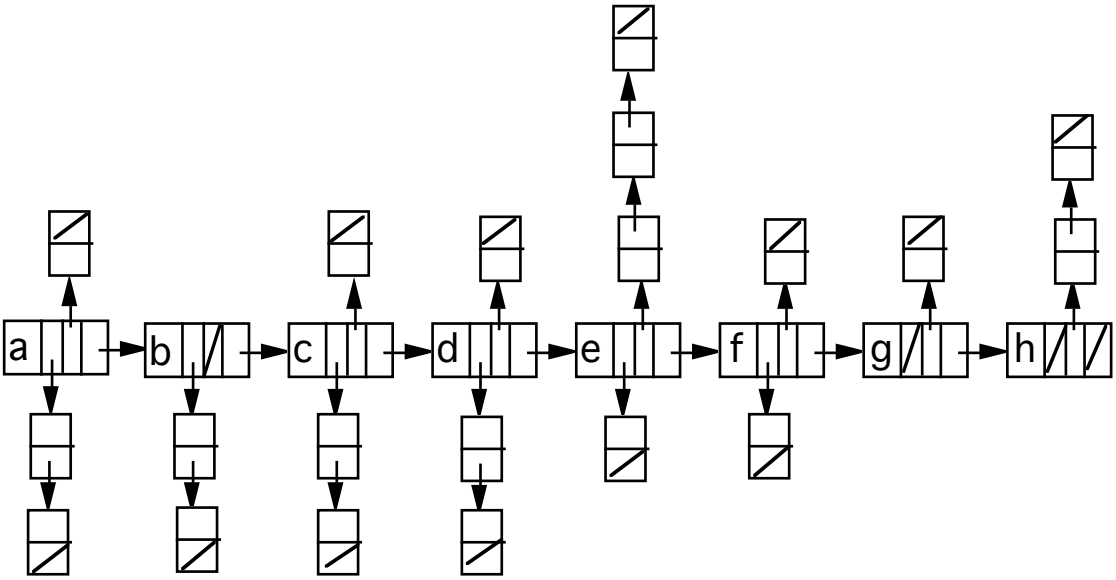


**Exemple :**

Soit le graphe :



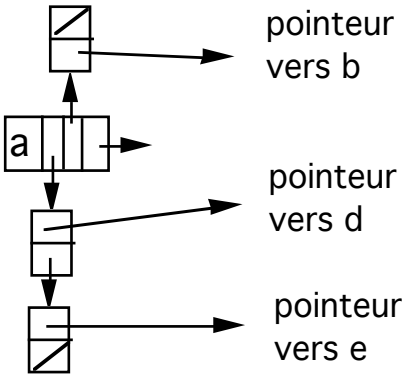

La représentation interne de ce graphe est donnée par




où une liste linéaire principale contient les informations sur les noeuds du graphe et à chacune des boîtes de cette liste est relié une liste secondaire des successeurs et une des prédécesseurs de ce noeud.

En général, les listes secondaires ne dupliquent pas les informations mais contiennent des pointeurs vers les boîtes de la liste principale.

Par exemple, la boîte associée au noeud "a" du graphe sera :




- Choix d'implantation des structures abstraites de données Ensemble, Pile, File, Tableau, Arbre, Graphe ....

## *40.3.5-Exercices*

1- Proposer une structure de données concrète pour représenter un ensemble de chaîne de caractères. En fonction de ce choix, définir les opérateurs classiques sur les ensembles.

2- Donner une représentation d'un arbre binaire par tableaux.

3- Définir les TDA graphe.
4 - Proposer une implantation de graphe (d'entiers). Considérer des implantations par listes et par matrices. Utiliser cette implantation pour réaliser les programmes suivants :

A- Recherche de chemin dans un graphe (Attention aux boucles).
B- Recherche de chemin dans un graphe et extraction des trajets.
C - Recherche du chemin le plus court dans un graphe (+ trajet).
D - Coloration d'un graphe

# 41. Paquetages en ADA

Les modules sont les briques de base dans la construction d'un programme. La notion de module en ADA est supportée par le paquetage. Les paquetages sont compilés séparément et permettent de se créer un environnement riche en les accumulant dans des bibliothèques.

Notions liées au paquetage :
• Encapsulation : Un paquetage est une collection d'entités et de ressources logicielles "encapsulées". Il permet de regrouper dans une même unité de programme des déclarations de constantes, de types, d'objets et de sous-programmes.

• Abstraction : Un paquetage permet de séparer la spécification d'un type (ensemble de valeurs et d'opérations sur ces valeurs) de sa mise en oeuvre (notion de type abstrait).

Comme dans les sous-programmes, un paquetage comprend deux parties :
- une spécification
- un corps

La spécification constitue l'interface entre unité et environnement. Elle comporte une partie visible contenant les déclarations des objets exportés et une partie cachée (privée) qui sert essentiellement dans le cas des types abstraits (voir plus loin).
Le corps réalise la mise en oeuvre des ressources définies dans la spécification. Elle peut comporter une partie déclaration et éventuellement une séquence d'initialisation du paquetage. Le corps doit obligatoirement contenir les corps des unités de programmes définies dans la spécification.

## 41.1. La spécification de paquetage

Exemple :

```
Package les_complexes IS
Type complexe is record
   re , im : float;
END record;
Function "+"(x,y : complexe) return complexe;
   -- entêtes d'autres fonctions telles que "*", "/"..
END les_complexes;
```

## 41.2. Utilisation des paquetages

Visibilité à l'extérieur et à l'intérieur du paquetage :
Les entités déclarées dans la partie visible (spécification) d'un paquetage ont pour portée celle du paquetage. Elles ne sont directement visibles qu'à l'intérieur du paquetage.
A l'extérieur du paquetage, il faut utiliser la notation pointée :

```
WITH les_complexes;
.........
c1,c2 : les_complexes.complexe;
c1 := les_complexes."+"(c1,c2);
```

Pour éviter la notation pointée, on peut soit utiliser le mécanisme de renommage (voir plus loin) soit utiliser la clause USE. Cette clause permet normalement de rendre directement visible les entités exportées.

```
WITH les_complexes; USE les_complexes;
.........
c1,c2 : complexe;
c1 := c1 + c2;
```

## 41.3. Remarque sur la clause USE

*Il est impossible avec la clause USE de masquer/surcharger une déclaration directement visible à l'endroit d'apparition de la clause USE; les entités exportées n'ont pas la priorité.*

```
Package probleme IS
   subtype chaine is STRING(1..20);
   Function "&"(x,y : string) return chaine;
END probleme ;
```

```
WITH probleme; USE probleme;
procedure plante is
   CH,b : chaine;
begin
   CH := "Ada " & "est grand";
end plante;
```

L'exécution de ce programme provoque une exception CONSTRAINT_ERROR car l'opérateur & est celui qui est prédéfini sur les STRINGs. Il n'y a pas surcharge car les *chaine* est sous-type de *string*. L'affectation de la chaîne "Ada est grand" à CH provoque l'exception car cette chaîne ne fait pas 20 caractères exactement.

Dans l'exemple ci-dessus, on utilise le paquetage les_complexes qui est compilé séparément. Ce type d'usage correspond à un usage ascendante où l'on utilise des briques de base.
Une autre manière est d'appliquer une méthode descendante en définissant le paquetage à l'intérieur de l'unité qui l'utilise. On peut en suite compiler le corps séparément :

```
Procedure utilisateur is
   Package les_complexes IS
      Type complexe is....
      -- entêtes des fonctions du paquetage
   END les_complexes;

   USE les_complexes;
   c1,c2 : complexes;

   PACKAGE BODY les_complexes IS SEPARATE;
   -- le corps sera compilé séparément
Begin
   c1 := c1 + c2;
END utilisateur;
```

Remarquons que sans la clause USE, la fonction "+" n'est pas directement visible et il faut utiliser la notation pontée.

## 41.4. Visibilité et héritage

Seuls les entités exportées par le paquetage, c'est à dire définies dans sa spécification sont accessibles de l'extérieur.
Il n'y a donc pas d'héritage en ADA : les entités que le paquetage utilise (par une clause WITH) ne sont évidemment pas accessibles à l'extérieur. La seule solution est de reprendre le même contexte par les mêmes clauses WITH. La clause WITH ne s'applique que sur l'unité où elle est présente.

Exemple :

```
WITH les_complexes; use les_complexes;
Package imaginaire IS
.......
End imaginaire;
```

Les paquetages qui utilisent imaginaire n'ont pas accès au paquetage les_complexes. Ils doivent donc importer les_complexes s'ils veulent l'utiliser.

## 41.5. Le renommage

Le renommage est utilisé pour simplifier les références aux objets lors que ceux-ci ne sont pas directement visibles et alléger les écritures préfixées, ces références Une déclaration de renommage permet de désigner une entité existante sous un nouveau nom sans que la liaison entre un ancien nom et l'entité soit détruite.
Différents usages sont possibles :

• Utiliser un nom plus simple :

```
x : integer renames gestion.enseignant.compte;
```

Ainsi, x désigne l'entité compte du paquetage enseignant interne au paquetage gestion.

• Rendre plus rapide l'accès à un objet :

```
TYPE tab is ARRAY(etudiant) of note;
Procedure calcul(a: tab; i:etudiant) is
   major_promo : note renames a(i);
```

Ici, major_promo dénote l'élément i du tableau a.

• Renommage de sous-programmes, paquetages ... (voir plus loin)

## 41.6. Le corps du paquetage

Forme générale :

```
Package BODY nom IS
-- déclarations éventuelles

-- eventuellemnet        Begin
                                -- suite d'instructions
                                -- Exception        éventuelles
                                        -- traitement des exceptions
END nom; -- END obligatoire
```

Le corps du paquetage peut être compilé séparément.

Les entités déclarées dans le corps ont leur portée réduite à ce corps et ne sont pas visibles à l'extérieur du paquetage.

L'élaboration d'un corps de paquetage comprend :

- l'élaboration de la partie déclarative. Les objets locaux du corps sont installés. Cette partie comprend les descriptions des sous-programmes définis dans la partie spécification;
- l'exécution des instructions du corps (entre le Begin éventuel et le End du paquetage). Les objets locaux sont initialisés. Ces initialisations survivent (même après la fin de l'exécution du corps). Ils restent accessibles, en particulier via les sous-programmes exportés par le paquetage. (un appel à un de ces sous-programmes de l'extérieur manipulera les données dans leur forme initialisée).

Exemple :

```
Package Body les_complexes IS
   Function "+"(x,y : complexe) return complexe is
   Begin
      return(x.re+y.re, x.im+y.im);
   END "+";
```

```
   Function "*"(x,y : complexe) return complexe is
   Begin
      return( re => x.re*y.re - x.im*y.im ,
            im => x.re*y.im + x.im*y.re);
   END "*";

   --  d'autres fonctions sur les complexes
-- pas de Begin du corps donc pas d'initialisation
END les_complexes;
```

> Le corps du paquetage doit obligatoirement contenir les corps de toutes les unités de programmes définies dans la spécification.

## 41.7. Paquetage et types privés

Ce mécanisme permet d'implanter les types abstraits, c'est à dire une définition de type et des opérations qui portent sur les valeurs du domaine défini par le type sans toute fois laisser voir les détails d'implantation du type et des opérations.
La spécification comporte une partie visible et une partie privée. Dans la partie visible, on trouve les déclarations d'entités exportées :

• une ou plusieurs déclarations de types privés;
• les déclarations des opérations définies sur les objets des types privés (sous forme de spécification d'entête de sous-programme);
• des déclarations d'exceptions correspondant à des cas d'erreurs de manipulation de ces types;
• déclarations de constantes...

Il y a deux sortes de types privés :
- types privés simples :

```
type germe is PRIVATE;
```

- types privés limités

```
type cle is LIMITED PRIVATE;
```

La structure interne d'un type privé est invisible à l'extérieur du paquetage. L'accès à ces objets n'est alors possible qu'à travers les sous-programmes définis dans la partie spécification du paquetage; Ce qui permet une meilleure protection des objets.
Les opérations d'affectation et les tests d'égalité et d'inégalité restent possibles pour les objets d'un type privé (mais pas les autres tests).

> La limitation des opérations sur les types privés ne s'applique qu'à l'extérieur du paquetage qui le déclare. Dans le corps même du paquetage, ce type conserve toutes ses opérations.

Exemple :

```
Package les_complexes IS
   Type complexe is PRIVATE;
   Function "+"(x,y : complexe) return complexe;
      -- entêtes d'autres fonctions telles que "*", "/"..

   PRIVATE
      Type complexe is record
         re , im : float;
      END record;
   END les_complexes;
```

Ainsi, les utilisateurs du paquetage les_complexes n'ont plus accès à la structure d'un objet de type complexe. Ils ne peuvent donc pas faire référence aux champs *re* et *im* de ce type.
Si l'on doit donner la possibilité d'accès à ces champs, on déclare dans la partie spécification du paquetage les sous-programmes permettant cet accès.

Par exemple, on déclare :

```
Function reelle(nombre:complexe) return float;
   -- rend la valeur de la partie réelle d'un nombre complexe
Function imaginaire(nombre:complexe) return float;
   -- rend la valeur de la partie réelle d'un nombre complexe
Procedure initialiser(re,im : float; nombre : out complexe);
   -- initialise un nombre complexe
```

Les détails d'implantation d'un nombre complexe sont cachés. On peut décider de les implanter autrement; par exemple, par un tableau à deux éléments :

```
Private
   type indice is (re,im);
   type complexe is ARRAY(indice) of float;
```

Remarque : la déclaration de type privé peut comporter une liste de contraintes de discriminants. Le type caché est alors un type article non contraint :

```
Type symbole(size : positive) is Private;
```

Il est possible de déclarer des constantes d'un type privé dans la partie visible (constante différées). La valeur de la constante est inconnue à l'extérieur du paquetage. La déclaration ne comporte alors que le nom et le type de la constante. La partie privée doit déclarer la valeur de la constante :

```
Package protection IS
   Type mot_de_passe is private;
   code : CONSTANT mot_de_passe;
   Function codage(nom : string) return mot_de_passe;
   Function "<"(op1,op2 : mot_de_passe) return mot_de_passe;

PRIVATE
   Type mot_de_passe is range 0..9999;
   code : CONSTANT mot_de_passe := 0;
End protection;
```

Exemple d'utilisation :

```
Use protection;
mon_code : mot_de_passe := code;
```

Il est possible de masquer un opérateur prédéfini en le redéfinissant dans la partie visible. Dans l'exemple précédent, l'opérateur "<" masque celui déclaré sur les entiers.
La surcharge/masquage de l'opérateur "=" n'est possible que pour les types limités privés sur lesquels cette opération est interdite.

## 41.8. Les paquetages prédéfinis

L'annexe C du manuel de référence fournit la spécification du paquetage STANDARD. Ce paquetage constitue l'environnement de tout programme ADA. Il contient les déclarations des types, constantes, exceptions et opérateurs prédéfinis du langage. ceux-ci sont visibles directement. On les utilisera avec la notation pointée dans le cas de masquage.

## 41.9. Méthodologie

Sous-programmes et paquetages jouent un rôle fondamental dans la programmation ADA et en particulier la structuration de gros logiciels.
Les paquetages permettent de :
- regrouper une collection de déclarations;
- former des bibliothèques d'unité de programmation;
- définir des types abstraits;

Collection de déclarations :

Afin de faciliter les modifications et améliorer la lisibilité des programmes, on regroupe logiquement les entités telles que :
- les constantes d'un même système d'unité;
- l'ensemble des types et des objets globaux d'un programme et leur initialisations.

Le paquetage peut être utilisé pour regrouper des objets communs à plusieurs modules. Cette pratique a cependant le défaut d'encourager le programmeur à utiliser des variables globales.

Regroupement d'unités de programmation
Cet usage correspond à la définition de bibliothèques de sous-programmes. La spécification du paquetage contient alors des déclarations de type, de constantes, de sous-programmes, voir d'exceptions. Le corps du paquetage ne contient alors que les corps des sous-programmes définis dans la partie spécification.

Définition d'objets abstraits :
Une bonne pratique de programmation consiste à n'exporter qu'un type privé par paquetage avec les opérations applicables sur ce type.
On étudie ci-dessous un exemple de paquetage d'un type abstrait ensemble de caractère. Lors de l'étude de la généricité, nous verrons comment définir un ensemble de n'importe quel type.
La définition d'un ensemble est celle des mathématiques. Les valeurs possible d'un ensemble de caractères sont données par tous les caractères possibles.

## 41.10. La Compilation Séparée

Constat : Les environnements de programmation sont de plus en plus riches. La construction d'un système informatique ressemble de plus en plus à un assemblage de pièces d'un jeu de "légo"; c'est à dire, par un assemblage d'unités.

Dans une telle démarche, le concept de "module" est central. Ces modules qui peuvent être "génériques" (modules paramétrés) sont produits par la compilation "séparée" de programmes réalisant des tâches bien définies avec une interface propre et claire.

En ADA, le concept paquetage, bien supérieure au sous-programme, est le moyen de construction de briques de base de grosses applications. Un paquetage permet au programmeur non seulement de regrouper des déclarations de sous-programmes, mais également d'assembler des

déclarations de types et d'objets utiles à d'autres parties de son application.

On peut par exemple acheter des paquetages (sous forme source ou binaire), en produire par les membres de l'équipe chargée du développement et les assembler à l'aide de la compilation séparée.

La compilation séparée en ADA supporte deux méthodes de construction de programmes :

•La méthode **descendante** (sous-problèmes auxquels on fait correspondre des sous-unités)
• La méthode **ascendante** qui consiste à mettre en bibliothèque des unités de programme dont on a un usage important et varié.

La généricité permet d'élargir le domaine d'utilisation d'une unité. Une unité générique (paramétrée) définit une famille d'unités ne différant que par un certains nombre de caractéristiques (différentes valeurs des paramètres). La définition d'une unité générique procède par généralisation puis, par la création d'un exemplaire spécialisé.

## La compilation séparée

Fait partie intégrante d'ADA. Par sa définition, le langage ADA met en place les mêmes contrôles pour un programme écrit en un seul morceau que pour un programme formé de plusieurs unités compilées séparément.

L'utilité de cette compilation est évidente (et nécessaire) dans les grosses applications sur lesquelles travaillent plusieurs personnes. Une fois définies et compilées les interfaces; le travail peut être réparti.

La compilation séparée permet également de se constituer progressivement un environnement de travail de plus en plus riche lorsque l'on travaille dans un environnement personnalisé.

On peut compiler séparément le corps d'une unité et ainsi d'utiliser dans la construction d'un programme une unité non encore réalisée. Ce qui permet de retarder des choix de mise en oeuvre.

Le découpage en sous-unités permet d'éviter des recompilations inutiles (nécessaires pour une application écrite en un seul morceau).

Le langage ADA distingue deux sortes d'unités de compilation :

❒ Les unités **primaires** :
Ce sont :

- spécification de sous-programme
- spécification de sous-programme générique
- spécification de paquetage
- spécification de paquetage générique
- corps de sous-programme

Ces unités sont les briques de bases dans l'approche ascendante. Il s'agit d'une unité de compilation "importée" (par la clause WITH) réutilisable dans divers contextes.

Exemples :
les unités de la bibliothèque d'entrées-sorties,
les unités de l'allocation / libération, de conversion,
les paquetages personnels...

❒ Les unités **secondaires**

Ce sont les unités que l'on déclare SEPARATE et qui sont détachées de leur contexte de définition et d'utilisation. Une unité secondaire dépend obligatoirement (directement ou non) d'une unité primaire. Elle ne peut pas être utilisée seule.
Une unité secondaire peut être soit un corps d'unité primaire, soit une sous-unité. Ces unités couvrent les corps des sous-programmes, les corps de paquetages et les corps des tâches. Elles permettent de mettre en ouvre l'approche descendante.

## *41.10.1- Approche descendante*

Dans cette approche, on procède par décomposition de problèmes en sous-problèmes et par raffinement successifs. A un niveau donné de la décomposition, quand on effectue la réalisation d'une unité, on précise seulement la spécification des sous-unités utilisées; la réalisation de ces sous-unités est repoussée à plus tard.

Le corps d'une sous-unité peut donc constituer une unité de compilation (secondaire). La spécification de celle-ci doit apparaître dans l'unité qui l'utilise (unité parente). Le corps est remplacé dans l'unité parente par une **souche** (stub) et le mot clé SEPARATE indique que ce corps est compilé séparément.

Une souche peut remplacer un corps de :

• sous-programme :

```
Procedure placer(tab : in out vecteur; i : integer)
                                IS SEPARATE;
```

• paquetage :

```
Package BODY fifo        IS SEPARATE;
```

• type tâche :

```
TASK BODY semaphore IS SEPARATE;
```

Ainsi, l'unité parente contenant les souches de sous-unités peut être compilée sans que le corps des sous-unités l'aient été.

L'unité parente peut être elle-même sous-unité d'une autre unité. Lors de la définition du corps d'une sous-unité, on indique :

SEPARATE(chemin_d'accès_à_la_sous-unité)

*Les objets visibles, directement ou non, à l'endroit où apparaît la souche de la sous-unité sont aussi visibles dans le corps de la sous-unité. Dans l'exemple suivant, les variables globales sont toutes visibles. La dépendance entre une sous-unité et l'unité parente induit un ordre de compilation et de recompilation (voir plus loin).*

Le chemin d'accès à une sous-unité part d'un corps d'unité primaire et désigne de façon non ambiguë une sous-unité. Il s'agit d'un chemin dans un arbre. Ce chemin ne contient que des noms d'unités compilées séparément car une souche ne peut apparaître que dans la partie déclarative du corps d'une unité de compilation.

## *41.10.2- L'approche Ascendante : La clause WITH*

Cette approche consiste à bâtir un programme à partir d'unités de programme dont la spécification a été préalablement compilée et mise dans une bibliothèque ADA (unités primaires).

Pour utiliser ces unités, on insère dans l'unité utilisatrice, dans la partie appelée *contexte*, des clauses WITH indiquant les unités primaires que l'on veut utiliser. Cette partie contexte précède le texte de l'unité en cours de définition :

```
WITH text_io; Use text_io;
WITH integer_io; Use integer_io;
Package BODY mon_paquetage IS
.....
End mon_paquetage ;
```

Dans cet exemple, on indique que le corps du paquetage *mon_paquetage* utilise l'unité de compilation prédéfinie *text_io*. La clause **Use** permet d'accéder directement, sans passer par la notation pointée, aux objets exportés par *text_io*.

L'effet (la portée) de la partie contexte d'une unité s'étend :
- à l'unité de compilation
- à son corps s'il s'agit d'une spécification
- à ces sous-unités

Placer une clause WITH devant une spécification alors que seul le corps utilise l'unité exportée par la clause WITH peut entraîner des recompilations inutiles. Il est de bonne pratique de ne placer une clause

WITH que là où elle est strictement nécessaire : Un corps ou une sous-unité peut posséder une partie contexte.

## 41.11. Quelques paquetages prédéfinis

- calendar : gestion de temps
- system : caractéristiques dépendant de la machine
- machine_code : programmation langage machine
- unchecked_conversion : conversion de type sans contrôle
- unchecked_deallocation : désallocation mémoire
- sequential_io : entrées/sortie séquentielles
- direct_io : fichiers à accès direct
- text_io : fichiers texte
- io_exceptions : exceptions dues aux entrées/sorties
- low_level_io : entrées/sorties physiques

# 42. La Généricité

• Dans une vue synthétique en phase d'analyse d'un problème, on peut constater qu'un certain (ensemble de) traitement s'applique aux différents types de données.
En se plaçant à un niveau abstrait, on peut avoir besoin de :

- *permuter* deux éléments (de n'importe quel type)
- *trier* des éléments d'une *table* (de n'importe quel type)
- *rechercher* un élément dans une *table* (de n'importe quel type)
- *utiliser* une *pile* (de n'importe quel type) ....

p Les structures et les traitements sont **abstraits**.

Pour l'exemple de permutation, on dira :
"*Pour permuter deux élément X et Y (du même type), on se sert d'une variable intermédiaire Z et ....*"

On traduit cette "spécification" par :

```
X,Y : Type_de_X_Y;
Z :  Type_de_X_Y := X;
Begin
   X := Y; Y := X;
End ;
```

p Ici, le type de X et de Y importent peu.
p Il faudra ensuite "prendre" un exemplaire de cette procédure,
par exemple pour permuter des entiers.

La procédure *echange* ci-dessus est dite *générique* (polymorphe). Elle accepte un couple X et Y de n'importe quel type et permute leur contenus.

La **Généricité** permet de définir des familles paramétrées d'unités de programmes.
Les unités d'une même famille ne diffèrent que par un certain nombre de caractéristiques décrites par les paramètres formels génériques.

## 42.1. La généricité en ADA

```
GENERIC
   type param is private;          -- tout type acceptant  ":="
   procedure echange(X,Y : In Out param);    -- la spécification
procedure echange(X,Y : IN out param) is             -- le corps
   Z : param := X;
   begin
      Y := X; X := Z;
   end echange;
```

p On donne la spécification et le corps. *Param* est générique.

### *42.1.1- Instanciation*

Création d'une unité de programme à partir d'une unité générique. Lors de cette instanciation on associe des paramètres effectifs aux paramètres formels génériques.

```
procedure echange_entier is NEW  echange(integer);
Procédure echange_eleve IS NEW  echange(param =>  eleve);
```

**Remarques :** En ADA, on est amené à utiliser la généricité dès les premiers programmes.

```
GENERIC
   type Num is range <>;
package Integer_io is
   procedure Get(item : out Num; .....);
   procedure Put(item : in Num; ...);
....
Package mes_entiers_io is NEW integer_io(mon_type_entier);
```

**Généricité et TDA** : la généricité permet d'écrire un moule à partir duquel on peut créer des procédures spécifiques à chaque type sans avoir à réécrire ces procédures.

## *42.1.2- Les unités génériques en ADA*

La généricité peut s'appliquer (unité pouvant être générique) :

- aux paquetages
- aux abstractions (procédures et fonctions)

## *42.1.3- Les paramètres génériques*

- des paramètres **valeurs** (Caractéristique de dimensionnement),
- des paramètres **objets** (Liaison d'une unité à un objet global),
- des paramètres **types** (Tris d'entiers - Tris de nombres flottants),
- des paramètres **sous-programmes** (Tri croissant/ décroissants)

Une unité générique comporte comme toute unité de programme une spécification et, éventuellement, un corps qui peuvent être compilés séparément.

## 42.2. La généricité simple : procédures génériques

Cas de l'exemple précédent : avec type en paramètre

```
GENERIC
   type param is private;                          -- un type à affectation
procedure echange(X,Y : IN out param);        -- spécification
 procedure echange(X,Y : IN out param) is     -- corps
 Z : param := X;
 begin
   Y := X; X := Z;
 end echange;
```

p **IS PRIVATE** signifie que le paramètre effectif qui sera associé à **"param"** pourra être de tout type sur lequel l'affectation et la comparaison sont définis (type à affectation)

==> "**type à affectation**" en ADA : tous les types sauf **privé limité** et **tâche**.

### Exemples d'instanciation :

```
Procedure echange_caractere IS NEW
      echange(element => character) ;

Type eleve is .....
Procedure echange_eleve IS NEW echange(eleve) ;
```

## *42.2.1- Cas de plusieurs procédures génériques*

• Dépendantes des types différents (dans le même fichier éventuellement) :

```
GENERIC
   type leger is private;
procedure voler(X : leger );
procedure voler(X : leger ) is
Begin .... -- faire voler un objet léger
End voler;

GENERIC        -- Generic répété
   type lourd is private;
procedure tomber(X : lourd );
procedure tomber(X : lourd ) is
Begin ....-- faire tomber un objet lourd
End tomber;
```

La compilation de cette unité crée deux procédures génériques en bibliothèque.

**Exemples d'instanciation :**

```
procedure voler_leger is new voler(leger => pigeon);
procedure tomber_lourd is new tomber(lourd => elephant);
```

p Mise en correspondance par les notations *positionnelles*, *nominales* et *mixtes* entre les paramètres formels et effectifs.

## *42.2.2-Vers les paquetages génériques*

### *42.2.2.1- Les procédures génériques dépendant du même type*

==> Notion de type (abstrait)
==> On peut faire comme dans le cas précédent mais
==> Il vaut mieux créer un paquetage : **regrouper**

```
GENERIC
   type poids is private;            -- ce type sera précisé au moment
                                     -- d'instanciation
package mouvement is
   procedure avant(X : poids );
   procedure arriere(X : poids ) ;
   .....
end mouvement;

 package BODY mouvement is    -- le corps
   procedure avant(X : poids ) is .....;
   procedure arriere(X : poids ) is ....;
end mouvement;
```

**Exemples d'instanciation :**

```
package va_et_vient is new mouvement(poids => moyen);
```

## 42.3. Un exemple : paquetage de gestion de piles

La notion de pile est indépendante du type des éléments qu'elle contient. Le type des éléments à empiler peut constituer donc le paramètre générique.

```
GENERIC
   Type element is PRIVATE;          -- paramètre générique
PACKAGE pile_generique IS
   Type pile is PRIVATE;             -- le type exporté
   Procedure empiler (la_pile : IN OUT pile ;
                      l_element : element) ;
   Procedure depiler (la_pile : IN OUT pile;
                      l_element : OUT element);
   Function longueur (la_pile : pile) return natural ;
PRIVATE
   - implantation du type pile non spécifiée
End pile_generique ;
```

*Element* est ici un type *paramètre formel* générique du paquetage *pile_generique*. A l'intérieur de la spécification et du corps du paquetage générique, on peut utiliser *element* en tant que type.

### La création d'exemplaires spécialisés :

```
PACKAGE pile_entiers IS NEW  pile_generique (integer) ;
PACKAGE pile_flottants IS NEW pile_generique(element  => float) ;
```

entier et float sont ici les *paramètres effectifs* correspondant à element (l'égalité et l'affectation sont autorisés sur les entiers et les flottants).

Après cette instanciation, on dispose de deux types différents de piles exportés par le paquetage générique : le type pile d'entiers et le type pile de flottants :

```
pile_ent : pile_entiers.pile;      -- déclaration d'une pile d'entiers
pile_reel : pile_flottants.pile;   --          ==          ==          réels
```

### Remarques

• Pour alléger les notations, on utilise la clause **USE**, qui rend l'intérieur des spécifications des paquetages directement visible :

```
PACKAGE pile_entiers IS NEW  pile_generique (integer) ;
USE pile_entiers ;
pile_ent : pile;        -- déclaration d'une pile d'entiers
empiler (pile_ent, 3) ;
```

• **USE** s'applique toujours à une instance (jamais aux génériques).

• Les **RECORD** ADA proposent une autre alternative pour obtenir une structure de données permettant de disposer à la fois d'une pile d'entiers ou de réels.

## 42.4. Expression des contraintes

**Un exemple plus complexe :** Paquetage générique pour trier un tableau d'entiers, de caractères ou de réels, ....

```
GENERIC
   Type table is private ;          -- paramètre générique
Procedure tri_tout (tab_a_trier : in table ; tab_trie : out table) ;
```

Cette procédure décrit le tri d'un tableaux de type table; ce type n'est pas connu pour l'instant.

Exemples d'instanciation :

```
Type table_entier is ARRAY (1 .. 50) OF integer ;
Procedure tri_entier IS NEW tri_tout(table_entier) ;

Type table_caractere is ARRAY (1 .. 100) OF character ;
Procedure tri_caractere IS NEW  tri_tout(table_caractere) ;
```

**Problèmes :**

**P1•** Le paramètre précisé lors de l'instanciation doit ici être un type tableau (car l'algorithme de tri utilise des attributs propres au type tableau).
==> Il faut donc dans la description de **tri_tout** spécifier que seuls les paramètres de type tableau sont acceptés lors de l'instanciation.
==> Un paramètre TYPE générique doit donc, dans certains cas, préciser les **contraintes** que doivent satisfaire les types fournis en paramètres effectifs lors de l'instanciation.

**P2•** Les éléments de ce tableau doivent être comparables. Pour un tableau d'articles, il faut indiquer ce que veut dire comparer deux articles (un article n'est pas un type scalaire, donc pas de relation d'ordre).

**Solutions :**

• Contraindre le paramètre générique à être un tableau.

• Fournir les moyens de comparer ses éléments.
==> On fournit en paramètre générique la fonction de comparaison (voir plus loin).

## 42.5. Les paramètres génériques

L'exemple précédent montre qu'il faut disposer des paramètres types avec contraintes et des paramètres **sous-programmes**. Le langage ADA permet également de passer des paramètres **valeurs** ou **objets**.

### *42.5.1-Paramètres types*

L'indication **IS PRIVATE** impose déjà une contrainte sur le type qui peut être passé en paramètre effectif : il doit être un type à affectation (par exemple, il ne peut pas être une tâche).

Les différentes façons d'imposer une contrainte sont données ci-contre :

1) **TYPE** aucune_contrainte **IS LIMITED PRIVATE** ;
   *-- Tout type autorisé; aucune opération prédéfinie n'est exigée,*
   *-- **pas même** l'affectation ou l'égalité*
2) **TYPE** un_type **IS PRIVATE** ;
   *-- tout type pour lequel l'affectation et l'égalité sont permises*

p On peut avoir ***type** un_type(...) is [limited] private*. Le paramètre formel aura un discriminent avec le même type.

3) **TYPE** un_pointeur **IS ACCESS** un_type_quelconque ;
   *-- N'importe quel type accès permettant*
   *-- de pointer sur des objets de type* **un_type_quelconque**

4) **TYPE** un_discret **IS** (<>) ;
   *-- Un type discret (entier ou énuméré, pour les indices)*
5) **TYPE** un_entier **IS RANGE** <> ; *-- N'importe quel type entier*

6) **TYPE** un_flottant **IS DIGITS** <>;*-- N'importe quel type réel flottant*
7) **TYPE** un_fixe **IS DELTA** <>; *-- N'importe quel type réel fixe*

8) **TYPE** tableau_contraint **IS ARRAY** (un_type_indice) **OF** element ;
   *-- N'importe quel type tableau dont le type indice est*
   *-- **un_type_indice** et le type des éléments **element***

9) **TYPE** tableau_non_contraint **IS ARRAY**
   (un_type_indice **RANGE <>**) OF element ;
   *-- N'importe quel type tableau dont le type indice est un sous-*
   *-- type de* **un_type_indice** *et le type des éléments* **element**

La méthode de programmation consiste donc à passer du particulier au général **en regardant quelles sont les opérations prédéfinies et les attributs utilisés dans l'unité de programme.**

Classification des types selon le critère de généralité :
A chaque classe de types correspond un ensemble d'opérations prédéfinies et d'attributs. L'arbre ci-dessous montre ces classes en utilisant la même numérotation pour les classes que pour les contraintes.

On remarque que certaines classes ne sont pas numérotées. Il n'existe donc pas de règle pour fixer une contrainte sur cette classe. Ainsi, on ne peut pas demander en ADA à ce que les paramètres effectifs soient des scalaires, des réels (fixe et flottant) ou des types composés.
Avec l'indication **limited private**, on indique qu'on n'utilise aucune opération ou attribut prédéfini. Les opération dont on a besoin devront être explicitement précisées; sauf ":=" (voir plus loin).

## *42.5.2-Exemple-1*

Type scalaire circulaire :

```
GENERIC
Type T is (<>); -- discret
Function Next(x:T) return T;
Function Next(x:T) return T is
Begin
   If X = T'Last then return T'First;
   Else return T'Succ(X);
   End If;
End Next;
```

## Utilisation :

```
Type jour is (lundi, mardi, ....., dimanche);
Function lendemain is New Next(jour);
...
premier_jour : jour := lendemain(dimanche);

SubType chiffre is integer range 0..9;
Function suivant is New Next(chiffre);
...
zero : chiffre := suivant (9);
```

## *42.5.3-Exemple-2*

Implantation d'ensemble générique d'un type discret à l'aide d'un tableau de booléens :

```
Generic
   type base is (<>);
Package ensemble_de is
   type ensemble is private;
   type liste is array(positive range <>) of base;
   vide, plein : constant ensemble;
   function creer(X : base) return ensemble;
   function "+"(X,Y : ensemble) return ensemble;     -- l'union
         "*"                                                    -- l'intersection
   ..................
   function card(x:ensemble) return Natural;
private
   type ensemble is array(base) of boolean;
   vide : constant ensemble := (ensemble'range => false);
   plein : constant ensemble := (ensemble'range => true);
end ensemble_de;
```

## Utilisation

```
type primaire is (bleu, blanc, rouge, vert, ...);
package spectre is new ensemble_de(primaire);
```

## *42.5.4-Exemple-3*

Un paquetage générique possédant quatre paramètres *pixel*, *abcisse*, *ordonnee* et *image* :

```
GENERIC
   type pixel is range <>;              -- entier
   type abcisse is range <>;            -- entier
   type ordonnee is range <>;    -- entier
   type image is ARRAY (abcisse, ordonnee) OF pixel ;
PACKAGE logiciel_image is
   -- Partie visible et partie privée du paquetage
End logiciel_image ;
```

Ce paquetage fournit des logiciels de traitement d'images de taille quelconque (le domaine de valeur des pixels non fixé).

## Instanciation

```
type petit IS range 0..255 ;
type long IS RANGE 0..511 ;
type petit_pixel  IS range 0..127 ;
type gros_pixel  IS range 0..255 ;
type image1 IS ARRAY (petit, petit) OF gros_pixel;
type image2 IS ARRAY (long, long) OF  petit_pixel ;

PACKAGE logiciel1 IS NEW logiciel_image
        (petit, petit, gros_pixel, image1) ;

PACKAGE logiciel2 IS NEW logiciel_image
        (long, long, petit_pixel, image2) ;
```

## *42.5.5-Application à l'exemple de tri*

• Déclaration de la procédure générique qui trie des tableaux dont les éléments sont de type discret (on sait ces éléments comparables) :

```
GENERIC
   Type indice is (<>);     -- discret : entier/énuméré (pour l'indice)
   Type element is (<>); -- entier ou énuméré comparables
   Type table is ARRAY (indice) OF element;
Procedure tri_discret (tab_a_trier : in table; tab_trie : Out table) ;
```

### Instanciation :

```
Type l_indice is integer range 1..50;
Type tableau_entier is ARRAY (l_indice) OF integer;
Procedure tri_entier IS NEW
      tri_discret(l_indice, integer, table_entier);

Type tableau_caracteres is ARRAY (l_indice) OF character;
Procedure tri_caracteres IS NEW
      tri_discret(l_indice, character, table_caracteres);
```

### Remarques

L'arbre des type permet de constater une insuffisance : il n'y a pas de paramètre générique **scalaire** regroupant les entiers et les réels.
On ne peut pas indiquer une contrainte numérique et définir par exemple une fonction somme qui calcule la **somme** des éléments d'un tableau dont les éléments sont numériques.

```
GENERIC
   Type un_numerique is NUMERIC;  -- NON, pas en ADA
```

Pour ce faire, on pourra définir *trois* unités génériques, chacune paramétrée par un des types numériques :

```
GENERIC
   Type un_numerique is RANGE <> ;            -- les entiers
GENERIC
   Type un_numerique is DELTA <>;        -- les virgules fixes
GENERIC
   Type un_numerique is DIGITS <>;       -- les virgules flottantes
```

==> Il est donc impossible de généraliser "naturellement" la procédure *tri_tout* pour traiter des tableaux d'entiers ou de réels.

==> Une solution serait d'utiliser *IS PRIVATE*. On pourra ainsi instancier la procédure générique *tri_tout* y compris avec des paramètres éléments de type non scalaire (accès ou composé).
L'inconvénient de cette solution est que les types non scalaires n'étant pas comparables, il sera nécessaire de préciser en paramètre générique *une fonction de comparaison*.

## *42.5.6- Les paramètres sous-programmes*
**(suite exemple de tri)**

Dans la déclaration suivante, on indique que n'importe quel type privé convient (PRIVATE). On dispose de l'opération d'affectation(:=) et de comparaison (=, /=). Par contre, pour faire des comparaisons telles que (<=, >=, ..), on précise une fonction *compare* en paramètre :

```
GENERIC
   Type element is PRIVATE;
   WITH Function compare (op1,op2: element) return boolean;
   Type indice is (<>);     -- discret
   Type tableau is ARRAY (indice) OF element;
Procedure tri_tout (tab : IN OUT tableau);
```

### Utilisation :

```
Type nationalite is (breton, catalan, basque, berrichon);
Type tab1 is ARRAY (nationalite) OF integer;

PROCEDURE tri_croissant IS NEW
      tri_tout(integer, "<=", nationalite, tab1);
PROCEDURE tri_decroissant IS NEW
      tri_tout(integer, ">=", nationalite, tab1);
```

Afin d'éviter de spécifier la fonction de comparaison, ADA permet de se servir des paramètres sous programmes par défaut.

### *42.5.6.1- Paramètres sous-programmes par défaut*

On associe des paramètres par défauts aux sous-programmes. Ces paramètres seront pris si l'on ne précise pas le nom du sous-programme lors de l'instanciation :

Si le paramètre effectif sous programme n'est pas fourni, par défaut

① **IS <>** prendre un sous-programme du même nom que le paramètres formel.

② **IS *nom*** prendre un sous-programme de nom *nom*

**Exemple**

```
GENERIC
   Type element is PRIVATE;
①  WITH Function "<=" (op1,op2: element) return boolean is <>;
      -- si ce paramètre n'est pas précisé, on prend "<="
   Type indice is (<>);    -- type discret
   Type tableau is ARRAY (indice) OF element;
Procedure tri_tout (tab : IN OUT tableau);
```

**Utilisation :**

```
Type nationalite is (breton, catalan, basque, berrichon);
Type tab_entier is ARRAY (nationalite) OF integer;
Type tab_reel is ARRAY (nationalite) OF float;

PROCEDURE tri_entier IS NEW
   tri_tout(integer, nationalite, tab_entier );
   -- Par défaut, "<=" prédéfini sur les entiers est pris

PROCEDURE tri_réel IS NEW tri_tout(float, nationalite, tab_reel );
   -- Par défaut, "<=" prédéfini sur les réels est pris
```

Si l'on veut dans certains cas fixer soi-même l'opération d'égalité, et dans d'autres cas, prendre par défaut l'égalité prédéfinie :

```
GENERIC
   ......
②  WITH Function egal(op1,op2 : element) return boolean IS "=";
      -- par défaut, prendre "=" pour "egal"
```

**Remarque**

Le mécanisme de passage de sous-programme est utilisé pour généraliser l'utilisation des sous-programmes et "les passer en paramètre d'un autre sous-programme" comme dans certains langages.

Dans l'exemple suivant, la fonction *integrale* calcule l'intégrale du fonction *f* entre deux valeurs a et b. la fonction à intégrer constitue un paramètre générique :

```
GENERIC
   WITH FUNCTION f(x : float) return float;
FUNCTION integrale (a,b : float; nb_pas : integer) Return float ;
FUNCTION integrale (a,b : float; nb_pas : integer) Return float is
Begin
   ......
   y := a;
   x := f(y);      -- utilisation de la fonction passée
   .........      --en paramètre
   Return(z);
End integrale ;
```

Une instanciation :

```
   Function cos(d:float) return float;          -- sera définie plus loin
   Function integer_cos IS NEW integral(f=> cos);
```

Une utilisation :

```
   integer_cos(a => 3.2, b=> 4.1, nb_pas => 1000);
```

## *42.5.7- Les paramètres valeurs*

Il est possible d'associer des valeurs par défaut aux paramètres formels. Un paramètre par valeur constitue une constante (mode IN) à l'intérieur de l'unité générique. On peut préciser une valeur par défaut pour un tel paramètre :

```
GENERIC
   ligne : IN integer := 24;              -- 24 par défaut
   colonne : IN integer := 80;
PACKAGE terminal is ...
```

**Exemples d'instanciation :**

```
PACKAGE minitel IS NEW terminal(24,40);
PACKAGE imprimante IS NEW
   terminal(lignes => 66, colonne => 132);
PACKAGE console IS NEW terminal;        -- par défaut 24 et 80
```

Remarquons que l'on peut réaliser le même effet avec les tableaux dynamiques, tableaux non contraints ou articles non contraints.

**Un cas d'utilisation des paramètres valeurs :**

Certaines implantations d'ADA n'autorisent pas la procédure principale à avoir des paramètres. Pour ce faire, il est possible de compiler une instance de procédure générique :

```
GENERIC
   nb_phi : positive := 4;
Procedure table_de_phi ;          -- procédure générique

-- instanciation
Procedure table_de_2_phi is NEW table_de_phi(2);
```

*table_de_2_phi* peut servir de programme principal pour toute implantation.

## *42.5.8-Paramètres Objets*

Un paramètre objet joue le rôle d'une variable globale détachée de son environnement. Celle-ci peut être passée en paramètre à une procédure masi avec un coût éventuellement important. On utilise alors un paramètre générique objet.
Le paramètre effectif associé doit être une variable.

```
GENERIC
   var_globale : IN OUT un_type;
Procedure travail;
```

**Instanciation :**

```
Procedure travail1 is NEW travail(globale1);
```

# 43. Généricité et Récursivité

A l'extérieur de l'unité générique, le nom d'une fonction générique ne peut être utilisée qu'à instancier un exemplaire. Par contre, à l'intérieur de l'unité générique, on peut utiliser ce nom et donc pour la récursivité :

```
GENERIC
   Type type_entier is Range <>;
   Function Factorielle(d : type_entier) return type_entier;
   Function Factorielle(d : type_entier) return type_entier is
   Begin
      -- rappel récursif de la fonction Factorielle
   End Factorielle;
```

## Généricité et compilation séparée :

Contrairement aux autre unités non-génériques, le corps d'une unité générique doit être compilé avant toute instanciation. C'est la spécification plus le corps de l'unité générique que constituent un moule.

Une instanciation d'unité générique peut constituer une unité de compilation. Dans l'exemple ci-dessous, le paquetage *entier_io* est crée par instanciation du paquetage générique *integer_io* interne au paquetage *text_io* :

```
WITH text_io; use text_io;
Package entier_io is NEW integer_io(integer);
   -- création du paquetage entier_io pour faire des entrées sorties d'entiers.
```

# Exemple: package générique implantant un ensemble

```
GENERIC
    TYPE element IS private;
    WITH function egal(e1,e2:element) return boolean IS "=";
    WITH procedure afficher(x:element);
PACKAGE ensemble_generique IS
    TYPE ensemble IS PRIVATE;
    ens_vide : constant ensemble;
    function dans(x:element; e:ensemble) return boolean;
    function "*"(e1,e2: ensemble) return ensemble;
    function "+"(e1,e2: ensemble) return ensemble;
    function "+"(e:ensemble; x:element) return ensemble;
    function "+"(x:element;e:ensemble) return ensemble;
    procedure affiche_ensemble(e:ensemble);

PRIVATE    -- implantation de l'ensemble sous forme de liste
    TYPE ele;
    TYPE ensemble IS access ele;
    TYPE ele IS record
        info : element;
        svt : ensemble;
    END record;
    ens_vide : constant ensemble := NULL;
END ensemble_generique;
```

```
PACKAGE BODY ensemble_generique IS
function dans(x:element; e:ensemble) return boolean IS
BEGIN
    if e=ens_vide then return false;
    else return ((e.info = x) or else dans(x,e.svt));
    END if;
END dans;

function "*"(e1,e2:ensemble) return ensemble IS
temp : ensemble;
BEGIN
     if e1=ens_vide then return ens_vide;
    elsif dans(e1.info,e2) then return(e1.info + (e1.svt * e2));
    else return(e1.svt * e2);
    END if;
END "*";

function "+"(e1,e2:ensemble) return ensemble IS
BEGIN
    if e1=ens_vide then return e2;
    elsif dans(e1.info,e2) then return (e1.svt + e2);
    else return (e1.info + (e1.svt + e2));
    END if;
END "+";
```

```
function "+"(e:ensemble;x:element) return ensemble IS
temp : ensemble;
BEGIN
    temp := NEW ele'(x,e);
    return temp;
END "+";

function "+"(x:element;e:ensemble) return ensemble IS
BEGIN
    return (e+x);
END "+";

procedure affiche_ensemble(e:ensemble) IS
BEGIN
     if e /= ens_vide then  afficher(e.info); affiche_ensemble(e.svt); END if;
    END affiche_ensemble;
END ensemble_generique;
```

## Exemples d'utilisation

```
WITH text_io;
WITH ensemble_generique;
procedure test_ensemble_entier IS
procedure aff(e:integer);
PACKAGE mon_ens IS NEW ensemble_generique(element => integer,afficher => aff);
use mon_ens;
e1,e2,e3,e4 : ensemble:=ens_vide;

procedure aff(e:integer) IS
BEGIN  text_io.put(integer'image(e));
END aff;

BEGIN
    e1 := e1 +1; affiche_ensemble(e1);text_io.NEW_line;
    e2 := 10 +e2; affiche_ensemble(e2);text_io.NEW_line;
    e3 := e1 + e2; affiche_ensemble(e3);text_io.NEW_line;
    e4 := e1 * e2; affiche_ensemble(e4);text_io.NEW_line;
END test_ensemble_entier;
```

```
WITH text_io;
WITH ensemble_generique;
procedure test_ensemble_chaine IS
procedure aff(e:string);
subTYPE chaine IS string(1..6);
PACKAGE mon_ens IS NEW ensemble_generique(element => chaine,afficher => aff);
use mon_ens;
e1,e2,e3,e4 : ensemble:=ens_vide;
procedure aff(e:string) IS
BEGIN
     text_io.put(e(e'first..e'last));
END aff;
BEGIN
    e1 := e1 + "ada   ";
    e1 := e1 + "est   "; affiche_ensemble(e1);text_io.NEW_line;
    e2 := "genial" + e2; affiche_ensemble(e2);text_io.NEW_line;
    e3 := e1 + e2; affiche_ensemble(e3);text_io.NEW_line;
    e4 := e1 * e2; affiche_ensemble(e4);text_io.NEW_line;
    IF dans("ada   ",e1) then text_io.put_line("vrai"); else text_io.put_line("faux");
    END if;
END test_ensemble_chaine;
```

# 44. Application : type et machine abstraits

**Un exemple de paquetage générique : Pile**

**1°**- approch TDA :
La signature de la pile sera :

*Sorte pile;*
*utilise booléen, element*
*init_pile : -> pile*
*pile_vide : pile -> booléen*
*empiler : pile x element -> pile*
*depiler : pile -> pile*
*sommet : pile -> element.*

**2°**- pour aller vers une implantation (par une approch machine abstraite) : modification du TDA puis mise en facteur

*Machine pile;*
*utilise booléen, element*

*init_pile : -> pile*
*pile_vide : pile -> booléen*
*empiler : (pile x element) -> pile*
*depiler : pile -> (pile x element)*
*sommet : pile -> element*

Modification : le paramètre *pile* disparaît des opérations.

L'entité pile devient un descripteur d'état de la machine :

| | |
|---|---|
| *Machine pile;* | sera implanté par une classe, un objet, un paquetage, ... |
| *utilise booléen, element* | |
| *init_pile : ->* | une procédure d'initialisation |
| *pile_vide : -> booléen* | une fonction/variable booléenne |
| *empiler : element ->* | une procédure qui empile un élément |
| *depiler : -> element* | une fonction qui renvoie un élément |
| *sommet : -> element* | une fonction qui renvoie un élément |

Sommet peut éventuellement être une variable hors mis les problèmes des préconditions.

## Implantation en ADA :

Définition de la machine abstraite pile :

Ici, on ne définit pas de type PILE mais une structure abstraite PILE.
Donc, le type Pile n'est pas exporté.

```
generic
   Type elem is private;
PACKAGE pile IS
   Function est_vide return boolean;
   Procedure init;
   Procedure empiler (l_element : elem) ;
   Procedure depiler (l_element : out elem);
   Function sommet  return elem ;
   vide, plein : exception;                    -- spécificité d'ADA
end pile;
PACKAGE body pile IS
   -- on décide d'implanter la pile par un tableau de N éléments
   -- les données ci-dessous sont invisibles a l'extérieur
   N : constant integer := 10;
   s: array(1..N) of elem;
   nb_ele : natural := 0;

Procedure empiler (l_element : elem) is
   begin
      if nb_ele > N then raise plein;
      else nb_ele := nb_ele+1; s(nb_ele) := l_element;
      end if;
   end empiler;
```

```
Procedure depiler (l_element : out elem) is
   begin
      if nb_ele <=0 then raise vide;
         else l_element := s(nb_ele); nb_ele := nb_ele - 1;
      end if;
   end depiler;

Function sommet return elem is
begin
   if nb_ele <=0 then raise vide;
   else return S(nb_ele); end if;
end sommet;

Function est_vide  return boolean is
begin
   return (nb_ele =0);
end est_vide;

Procedure init is
begin   nb_ele := 0;
end init;
end pile;
```

Exemple d'utilisation :

```
with pile; with text_io;
use text_io;
procedure test_pile_machine is
package pile1 is new pile(elem => integer);

subtype ch is string(1..10);
package pile2 is new pile(elem => ch );

x : integer;
c : ch;
```

```
begin
   pile1.init;
   pile1.empiler(1);
   pile1.empiler(3);
   pile1.depiler(x); put(integer'image(x));
   put(natural'image(pile1.sommet));

   pile2.init;
   pile2.empiler("ecole ecl ");
   pile2.empiler("de lyon.fr");
   pile2.depiler(c); put_line(c);
   pile2.depiler(c); put_line(c);
   pile2.depiler(c); put_line(c);
   put(pile2.sommet);        -- jamais exécuté

exception
   when pile2.vide => put_line("pile de chaine est vide");
   when others => put_line ("problemes");
end;
```

# 45. Compléments sur CAML

• Pour toute fonction, on infère un type qui est le plus général possible.

## 45.1. Calcul de type d'une fonction en CAML

• Une constante (0,1,2..., true,false) dont le type est simple (int ou bool);

• Un identificateur (x, y, z, succ, eq, gt, positive...) dont le type est définie dans un environnement *ENV* par un type simple, une variable de type ou par*(α -> β)* où α et β sont des types. L'environnement est structuré par le règle BNF : *ENV ::= void ; ID, TYPE, ENV*.

• Une conditiannelle de la forme *if e1 then e2 else e3* dont le type est *T* si le type de *e1* est *bool* et *T* est le type de *e2* et de *e3*.

• Une lambda abstraction of the form *(function x -> e)* dont le type est *(T1 -> T2)* où *T1* est le type de *x* et *T2* est le type de l'expression *e*;

• Une application de la forme *(e e')* dont le type de la partie function *e* est *(T1 -> T2); T1* est le type associé à *e'* et *T2* est le type de l'application.

On note que l'application partielle est autorisée car le constructeur de type "->" est associatif à droite.

**Exemples:**

-Le type de la lambda abstraction *lambda x . (x > 0)* sera *(int-> bool)* si le type de ">" est (int -> (int -> bool)). Par conséquent, le type de *x* est *int*.

-Le type de l'application *(positive x)* est *bool* où *x* est de type *int* et *positive* a pour type *(int-> bool)*.

-Le type de l'application *(function x -> (eq x x))(succ y)* est *bool* où *y* est de type *int*, *succ* a pour type *(int-> int)* et *eq* a pour type *(α -> (α -> bool))*.

-Le type de l'expression
*(function x -> if (positive x) then (succ y) else (pred y))* est *int* où *y* est de type *int*, *succ* et *pred* ont le type *(int-> int)* et *positive* est de type *(int-> bool)*.

## 45.2. Comment définir le type d'une expression

**Exemple :**

```
#let comp = function f -> function g -> function x -> f(g x);;
comp : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

Soit les types suivants :

f : σ1
g : σ2
x : σ3
(g x) : σ4
f(g x) : σ5

Soit le type de *(g x)* = σ4

g : σ2 = σ3 -> σ4
↓
x

Soit le type de *f(g x)* = σ5

f : σ1 = σ4 -> σ5
↓
(g x)

**Remarque : si g : A -> B et f : B -> C alors f(g) : A -> C**

Donc :

f : σ1 = σ4 -> σ5
g : σ2 = σ3 -> σ4
x : σ3
f(g) : σ3 -> σ5 par la remarque ci-dessus

Posons :

σ3 = ‘c, σ4 = ‘a et σ5 = ‘b

on a :

σ1 = ‘a -> ’b

$\sigma 2$ = ‘c -> ‘a

Donc comp : (‘a -> ‘b) -> (‘c -> ‘a) -> ‘c -> ‘b
↓↓ ↓ ↓
f g x f(g x)

Application :
quel est le type de l’application partielle *(comp succ)* étant donné
succ : int -> int ?

On a vu :
comp : (‘a -> ‘b) -> (‘c -> ‘a) -> ‘c -> ‘b
↓
succ : int -> int car succ est le 1er paramètre ==> ‘a = int, ‘b = int

(comp succ) aura pour type le reste,
i.e. (‘c -> ‘a) -> ‘c -> ‘b
i.e. (‘c -> int) -> ‘c -> int
ou encore d’une manière générale
(comp succ) : (‘a -> int) -> (‘a -> int)

Par exemple, on a
((comp succ) succ) : int -> int

et

```
#((comp succ) succ) 1;;
- : int = 3
```

## 45.3. Récursivité et les fonctions d'ordre supérieur

- L'itération est souvent exprimée par l'application de fonctions aux listes. Cette itération sur les listes étant importante, certains langages fonctionnels tels que Miranda (ou Gofer, mais pas CAML) proposent une notation spéciale, appelée "list comprehension", pour spécifier les listes comme pour les ensembles au sens mathématique.

  Par exemple, si "ints" est une liste d'entiers, on peut en extraire une liste des nombres paires en écrivant :

  **[n | n <- ints ; n mod 2 = 0]**

  qui est à comparer avec la notation mathématique ensembliste appliquée aux listes :

  **{n | n ∈ ints; n mod 2 = 0}**

  L'occurrence de "n" dans "n <- ..." est une assignation dont la portée est le "list comprehension".

- Par exemple, pour construire une liste d'entiers chacun le successeur des éléments de la liste "ints", on écrit :

  [n+1 | n <- ints ] (i.e. {n+1 | n∈ ints})

- Pour écrire une fonction qui calcule la somme des carrés des n premiers entiers (en syntaxe CAML) :

```
#let somme_sq(n) = somme [i * i | i <- 1 jusqua n]
```

- Et pour écrire la fonction quick-sort :

```
#let sort = fun
   [] -> []
|  (n::ns) ->
   sort [i | i <- ns; i < n] @
   [n] @
   sort [i | i <- ns; i >= n]
```

- Cette notation abstraite peut être traduite en CAML en utilisant les fonctions d'ordre supérieur.

## 45.4. Les fonctions d'ordre supérieur CAML

- Les langages fonctionnels traitent les fonctions comme des objets de **première classe** : elles peuvent être passés comme argument, être retournées comme résultat, faire partie d'une valeur composite, etc. Ceci a un impact important sur le style de programmation.

- On appelle *fonction de premier ordre* une fonction dont les paramètres et les résultats sont tous non fonctionnels.
  Une fonction qui a des paramètres le résultat fonctionnels est appelée une *fonction d'ordre supérieur*.

  Les fonctions d'ordre supérieur sont souvent utilisées pour produire du code réutilisable. ==> De cette manière, les fonction sont utilisées pour représenter les données d'une façon nouvelle.

- Permettre à une fonction de retourner une autre fonction comme résultat ouvre des possibilités intéressantes qui sont largement utilisées en programmation fonctionnelle.

- La plus simple manière d'engendrer une nouvelle fonction est de *composer* deux fonctions déjà existantes.
  ==> On se donne l'opérateur binaire "°" (noté *compose* ci-dessous) qui prend en arguments deux fonctions f et g et retourne leur composition : c'est à dire une fonction h telle que h(x) = f(g(x)).

```
#let compose (f : 'b -> 'c) (g : 'a -> 'b) = fun (x : 'a) -> f(g(x));;
compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
#let even = fun x -> x mod 2 =0;;
even : int -> bool = <fun>
#let odd= compose (prefix not) even;;
even : int -> bool = <fun>
#odd 1;;
- : bool = true
#odd 2;;
- : bool = false
```

**Exemple -2 :**

```
#let double (f : 'a -> 'a) = compose f f;;
double : ('a -> 'a) -> 'a -> 'a = <fun>

#let inc x = x+1;;
inc : int -> int = <fun>
```

Exemples d'utilisation :

```
#double inc;;
- : int -> int = <fun>

#(double inc) 1;;
- : int = 3
```

## 45.5. Curryfication

- Soit la fonction suivante qui calcule le $x^n$ ($n \geq 0$, x : réel).

```
#let rec pow (n, x) = if n = 0 then 1.0 else x *. pow((n-1),x);;
pow : int * float -> float = <fun>
```

- L'application de cette fonction à un couple (entier,réel) retourne un réel. Par exemple pow(2,a) retourne $a^2$.

- Considérons maintenant la fonction powC proche de pow :

```
#let rec powC n x = if n = 0 then 1.0 else x *. powC (n-1) x;;
powC : int -> float -> float = <fun>
```

- Le type de powC est *int -> float -> float = int -> (float -> float)*, L'application de powC à un couple entier-réel retourne un réel.
- powC est plus intéressante que *pow* car elle peut être utilisée avec un seul argument :

```
#let sqr = powC 2 and cube = powC 3;;   (* déclaration collatérale *)
sqr : float -> float = <fun>
cube : float -> float = <fun>
```

- On notera que *pow* prend en fait un argument qui est un couple alors que *powC* prend un couple d'argument (deux arguments).

La version powC de pow est appelée une version **curryfiée** (d'après le logicien Haskel Curry). La technique d'appeler une fonction curryfiée avec un nombre d'arguments inférieur au maximum de ses arguments est appelée une *application partielle* de *powC*.

La fonction (prédéfinie en ML) *filtre : ('a -> bool) -> ('b list -> 'b list)* applique une fonction f aux éléments x d'une liste L et retourne une liste d'éléments de L pour lesquels (f x) est vrai.

```
#let rec filtre (f : 'a -> bool) = fun
      []     -> []
   | (x :: xs)  -> if (f x) then (x:: filtre f xs) else filtre f xs;;

filtre : ('a -> bool) -> 'a list -> 'a list = <fun>
```

Etant donné la fonction

```
#odd;;
- : int -> bool = <fun>
```

on peut filtrer les éléments d'une liste d'entiers et ne conserver que ceux qui sont impaires :

```
#filtre odd [1;2;3;4];;
- : int list = [1; 3]
```

- La fonction *filtre* nous permet de traduire le schéma ensembliste : $\{n \mid n \in ints; n \bmod 2 = 0\}$.

- Le langage CAML dispose d'un certains nombre de fonction prédéfinies d'ordre supérieur. Par exemple, les fonctions suivantes pour la manipulation de listes (il en existe un nombre important) :

  *for_all : ('a -> bool ) -> 'a list -> bool* où
  for_all f [a1;...; an] = (f a1) & ... & f(an) & : et logique

  *map ('a -> 'b) -> 'a list -> 'b list* où
  map f [a1;...; an] = [(f a1) ; ... ; f(an)]

*flat_map ('a -> 'b list) -> 'a list -> 'b list* où
flat_map f [l1;...; ln] = (f l1) @ ... @ f(ln) @ : concaténation

## *45.5.1-Exemple : quick-sort générique*

Pour cela, nous allons nous définir une fonction d'ordre supérieur *filtre2 f x l* qui prend en entrée une fonction booléenne binaire f, un paramètre x et une liste l=[a1;...; an] et calcule la liste [(f x a1); ... ; (f x an)].

```
#let rec filtre2 (f : 'a -> 'a -> bool) (x : 'a) = fun
   []     -> []
   | (y :: ys)  -> if (f x y) then (y:: filtre2 f x ys) else (filtre2 f x ys);;

filtre2 : ('a -> 'a -> bool) -> 'a -> 'a list -> 'a list = <fun>
```

Par exemple :

```
#filtre2 (prefix <) 3 [1;2;3];; (* les éléments plus_grand que 3 *)
- : int list = []

#filtre2 (prefix <) 2 [1;2;3];; (* les éléments plus_grand que 2 *)
- : int list = [3]
```

On se donne également la fonction de récupération de l'inverse d'un ordre :

```
#let inverse = fun
   (prefix <) -> (prefix >=)
|  (prefix >) -> (prefix <=)
|  ...        -> .. ;;
```

La fonction générique quick_sort sera :

```
let rec gen_q_sort ordre = fun
   []     -> []
   | (x :: xs) -> gen_q_sort ordre (filtre2 ordre x xs)
            @ [x]
            @ gen_q_sort ordre (filtre2 (inverse ordre) x xs);;
```

**Remarques** : on ne peut pas remplacer inverse par not car par exemple not (x= y) est différent de ((not =) x y).

Exemple d'utilisation :

```
#gen_q_sort (prefix <) [1;4;5;3];;
-: int list =[5;4;3;1]

#gen_q_sort (prefix >) [1;4;5;3];;
-: int list =[1;3;4;5]
```

- On peut également compliquer le paramètre "ordre" passé à "quick-sort" générique. Par exemple, on peut trier une liste de tuples (jour, mois, an) chacun représentant une date en définissant une fonction *plus_petit* sur les dates; puis en triant une liste de dates.

## Autre solution à la fonction q_sort :

```
#let rec gen_q_sort ordre = fun
      []     -> []
      | (x :: xs)   -> let l = (filtre2 ordre x xs) in
                     (gen_q_sort ordre l)
                     @ [x]
                     @ gen_q_sort ordre (subtract xs l);;
gen_q_sort : ('a -> 'a -> bool) -> 'a list -> 'a list = <fun>
```

La fonction *subtract l1 l2 : 'a liste -> 'a list -> 'a list* retourne la liste l1 privée de ses éléments figurant dans la liste l2.

## 45.6. Fonctions d'ordre supérieur : accumulation sur les listes

**Exemple**

Les fonctions *somme* et *produit* que nous avions définies ont une structure similaire. Elles diffèrent seulement en l'opérateur appliqué aux éléments d'une liste non-vide ainsi qu'en l'élément neutre de cet opérateur. On peut donc définir une fonction *list_it* qui généralise ce types de fonctions.

Ayant :

```
#let rec somme_1 = function
   []  -> 0
   | tete::reste -> tete+ (somme_1 reste);;
somme_1 : int list -> int = <fun>

#let rec prod_1 = function
   []  -> 1.0
   | tete::reste -> tete *.  (prod_1 reste);;
prod_1 : float list -> int = <fun>

#let rec concat_1 = function
   []  -> ""
   | tete::reste -> tete ^  (concat_1 reste);;
concat_1 : string list -> int = <fun>
```

- Dans ces différents cas et d'une manière générale, on applique une fonction binaire f à une liste [a1; a2;...; ap]. Le développement de la récursivité fait apparaître la forme
  f a1 ( f a2 ( ..(f a_p neutre) .. )
  où neutre est l'élément neutre de la fonction f : (f x neutre) = x.

On généralise avec :

```
#let rec list_it = function f -> function neutre -> function
   []  -> neutre
   | tete :: reste -> f  tete (list_it f neutre reste);;
list_it : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b = <fun>
```

```
#list_it (prefix ^) "" ["aa"; "bb"];;
- : string = "aabb"
```

- Remarque : également, la function d'ordre supérieur *it_list* réalise la forme f(..(f(f neutre a1) a2) .. ap).

Ici, les opérandes (paramètres) de f sont $f(a_{i+1}...)$ et $a_i$.
La différence réside dans la position du premier paramètre de f :
x1+x2+...+xp =
x1 + (x2+...+xp) = -- list_it
(x1+x2+...)+xp -- it_list

- Ces exemples montrent que très souvent, lorsque l'on fait abstraction sur la fonction, on fait également souvent abstraction sur les types.

  Il y a peu d'intérêt à réutiliser une fonction pour ordonner une liste d'entiers selon différents ordre. En revanche, il serait plus intéressant de pouvoir ordonner des listes de types différents. Ainsi, la plupart des langages fonctionnels possèdent un système **polymorphe** ou bien il sont dynamiquement typés.

# SOLUTIONS

# aux

# exercices