Paradigmes de programmation — Impératif

Xavier Crégut
<Prénom.Nom@enseeiht.fr>

ENSEEIHT
Sciences du Numérique

Objectifs

- Comprendre les principes de la programmation impérative
- Algorithmique : types primitifs, instructions, structures de contrôle
- Méthode des raffinages.
- Structuration des données : tableaux, enregistrements, types énumérés
- Structuration des instructions : sous-programmes et modules

Sommaire

Premier exemple

```
1   Algorithme périmètre_cercle
2
3       -- Déterminer le périmètre d'un cercle à partir de son rayon
4       -- Attention : aucun contrôle sur la saisie du rayon ==> non robuste !
5
6   Constante
7       PI = 3.1415
8
9   Variable
10      rayon: Réel          -- le rayon du cercle lu au clavier
11      périmètre: Réel      -- le perimètre du cercle
12
13  Début
14      -- Saisir le rayon
15      Écrire("Rayon = ")
16      Lire(rayon)
17
18      -- Calculer le périmètre
19      périmètre <- 2 * PI * rayon          -- par définition
20      { périmètre = 2 * PI * rayon }
21
22      -- Afficher le périmètre
23      Écrire("Le périmètre est : ", périmètre)
24  Fin
```

Exécution de ce programme

Exercice 1 Exécuter, sur papier, le programme précédent.

Modèle de calcul

Modèle de calcul :

- C'est ce qui donne le sens du programme (explique son comportement)

L'état du programme

- La valeur des variables déclarées dans le programme (une zone de la mémoire)
- Le compteur ordinal : le numéro de la prochaine instruction à exécuter

Lancement du programme :

- Au début de l'exécution, les variables ont une valeur indéterminée.
- Le compteur ordinal indique la première instruction du programme principal

Exécution :

- Exécution de l'instruction référencée par le compteur ordinal et
- Mise à jour du compteur ordinal (en général l'instruction suivante : séquence).

Le paradigme « impératif »

Principe

L'exécution d'un programme est une séquence d'affectations qui modifie son état.

Défit

Définir des outils de structuration des programmes pour permettre la conception, l'écriture et la maintenance (évolutive et corrective) de programme de grande taille :

- Structures de contrôle (exemple : bannir les branchements inconditionnels (goto))
- Types de données : organiser les données.
- Sous-programmes : procédures et fonctions (organiser les instructions)
- Modules et bibliothèques (organisation les sous-programmes et les types)
- Mais aussi : raisonner sur le programme (typage, flot de contrôle, assertions, preuve...), tests (fonctionnels, structurels, de performance), documentation, etc.

Sommaire

- Comprendre le problème posé
- Identifier une solution informelle
- Raffiner
- Produire l'algorithme (et le programme)
- Tester le programme

Méthode des raffinages

Principe général

Décomposer le problème posé en sous-problèmes qui seront à leur tour décomposés jusqu'à obtenir des problèmes élémentaires (i.e. compréhensible du processeur).

Ici un problèmes est élémentaire s'il correspond à une instruction ou une expression du langage algorithme.

Exemple fil rouge

Exercice 2 : Pgcd
Écrire un programme qui affiche le pgcd de deux entiers *strictement positifs* dont la valeur a été saisie au clavier.

Comprendre le problème posé

Il est essentiel de bien comprendre le problème posé pour avoir des chances d'écrire le **bon** programme !

Moyens :

- Reformuler le problème en rédigeant R0.
- Lister des « exemples d'utilisation » du programme. Il s'agit de préciser :
  - les données en entrées ;
  - **et** les résultats attendus.

Remarque : Les exemples d'utilisation donneront les jeux de test fonctionnels qui permettront de tester le programme.

Comprendre le problème posé : exemple sur le pgcd

```
1  R0 : Afficher le pgcd de deux entiers strictement positifs
```

Remarque : La reformulation doit être relativement concise et précise, au risque d'être un peu incomplète. Il s'agit de synthétiser le problème posé.
Les **exemples d'utilisation** (ou jeux de tests) pourraient être les suivants :

```
1      a    b       pgcd
2   ------------------------------------------------------------
3     2     4  ==>  2                    -- cas nominal
4    20    15  ==>  5                    -- cas nominal
5    20    20  ==>  20                   -- cas limite
6    20     1  ==>  1                    -- cas limite
7     1     1  ==>  1                    -- cas limite
8     0     4  ==>  Erreur : a <= 0      -- cas d'erreur (robustesse)
9     4    -4  ==>  Erreur : b <= 0      -- cas d'erreur (robustesse)
10  ------------------------------------------------------------
```

Identifier une solution informelle

- Identifier une manière de résoudre le problème.
- Il s'agit d'avoir l'idée, l'intuition de comment traiter le problème.
- Comment trouver l'idée ? **C'est le point difficile !**
- *Exemple :* Pour calculer le pgcd d'un nombre, on peut appliquer l'algorithme d'Euclide :

  Il s'agit de soustraire au plus grand nombre le plus petit. Quand les deux nombres sont égaux, ils correspondent au pgcd des deux nombres initiaux.

- **Remarque :** On peut vérifier son idée sur les exemples d'utilisation identifiés.

Raffinage

Idée : Décomposer un problème « compliqué » en sous-problèmes plus simples que l'on sait résoudre (que l'on espère savoir résoudre !).
Remarque : Il s'agit de formaliser la solution informelle.
Notation : On note R1 la décomposition/le **raffinage** de R0.
Définition : Un raffinage est la décomposition d'une étape en sous-étapes qui, réalisées dans l'ordre indiqué, réalisent exactement le même objectif.
Exemple :

```
1  R1 : Raffinage De « Afficher le pgcd de deux entiers positifs »
2     | Saisir deux entiers
3     | { les deux entiers sont strictements positifs }
4     | Déterminer le pgcd des deux entiers
5     | Afficher le pgcd
```

Important : Entre accolades est exprimée une propriété du programme, vraie à cet endroit.
Remarque : On peut utiliser des structures de contrôle dans un raffinage.

Raffinages et flot de données

Un raffinage décrit un enchaînement d'étapes qui échangent (généralement) des données (des informations).
Il est alors important de caractériser ce qu'une étape fait d'une donnée :

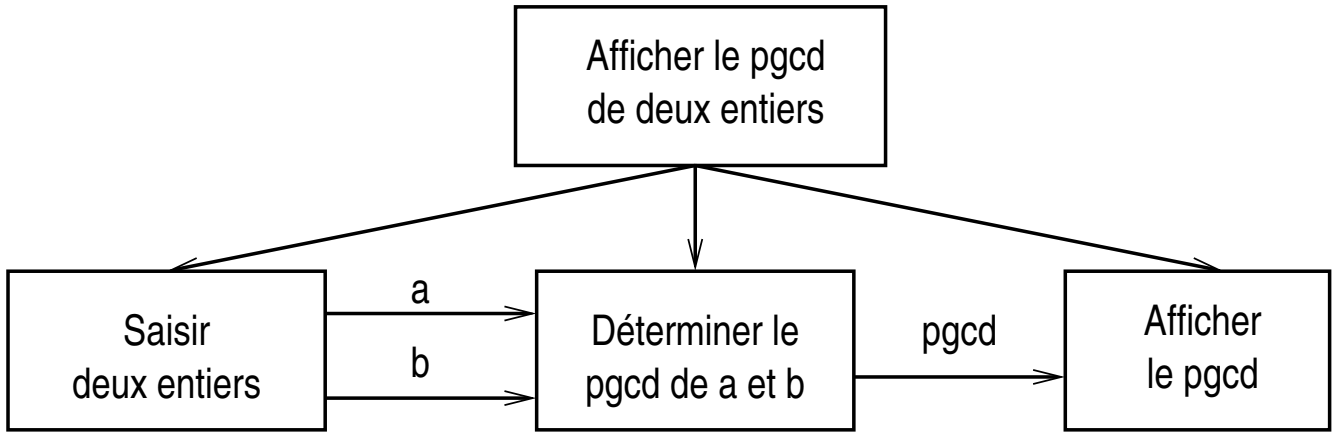
- **in** : elle l'utilise sans la modifier ;
- **out** : elle la produit (modifie) cette donnée sans accéder à sa valeur ;
- **in out** : elle l'utilise, puis la modifie.

```
1  R1 : Raffinage De « Afficher le pgcd de deux entiers positifs »
2     | Saisir deux entiers                a, b: out
3     | { (a > 0) Et (b > 0) }
4     | Déterminer le pgcd de a et b       a, b: in; pgcd: out
5     | Afficher le pgcd                    pgcd: in
```

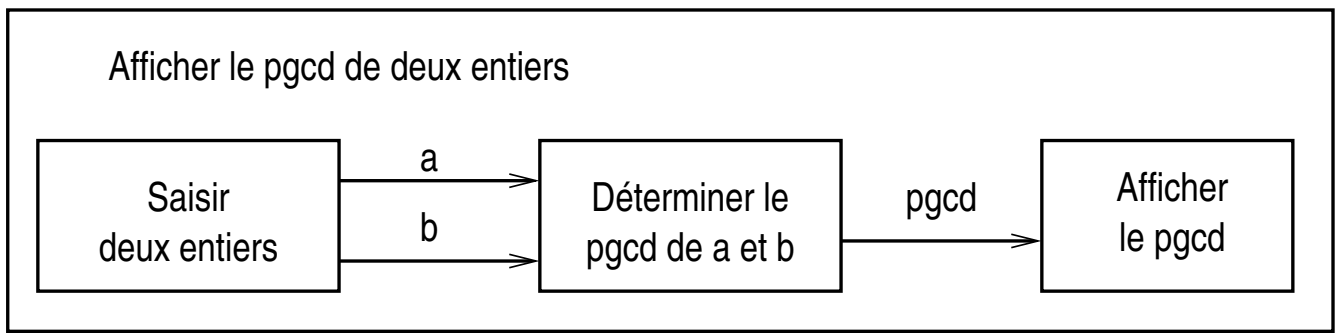
Avantage 1 : Expliciter les données permet de les référencer dans les autres étapes et d'écrire plus formellement les propriétés du programme.
Avantage 2 : Contrôler la chronologie d'une étape : une variable ne peut être en **in** que si elle apparaît avant en **out**.

Représentation graphique

Représentation sous forme d'arbre



Représentation sous forme de boîtes imbriquées



Remarque : Ces représentations permettent de « visualiser » un raffinage mais ne seront pas utilisées car elles sont peu adaptées si les flots de données sont complexes ou les sous-étapes non séquentielles.

Continuer à raffiner

- Si un raffinage introduit des étapes non élémentaires, elles doivent être à leur tour raffinées.
- **Exemple :** Les 3 sous-étapes introduites dans la décomposition de R0 sont non élémentaires et doivent donc être raffinées.
- **Notation :** On note R2 le raffinage des étapes introduites dans R1.
  Attention, il faut préciser l'étape qui correspond au raffinage.
- **Généralisation :** R_i est le raffinage d'une des étapes introduites dans un raffinage de niveau R_{i-1}.
- **Raffinages :** L'ensemble des raffinages constitue un arbre dont la racine est R0 et les feuilles sont des étapes élémentaires.
  R_i indique un raffinage de profondeur i dans l'arbre des raffinages.
- **Remarque :** On peut arrêter de raffiner si une étape est déjà connue ou si sa réalisation ne pose pas de problème (subjectif, dépend de la cible).

Continuer le raffinage du pgcd

```
1  R2 : Raffinage De « Déterminer le pgcd a et b »
2    | na <- a      -- variables auxiliaires car a et b sont en in
3    | nb <- b      -- et ne peuvent donc pas être modifiées.
4    | TantQue na et nb différents Faire                    na, nb: in
5    |   | Soustraire au plus grand le plus petit          na, nb: in out
6    | FinTQ
7    | pgcd <- na  -- pgcd était en out, il doit être initialisé.
8
9
10 R2 : Raffinage De « Afficher le pgcd »
11   | Écrire("pgcd = ")
12   | Écrire(pgcd)
13
14
15 R2 : Raffinage De « Saisir deux entiers »
16   |    ...
```

NB : Un raffinage peut utiliser des structures de contrôle (**Si**, **TantQue**...).

Continuer le raffinage du pgcd

```
1  R3 : Raffinage De « na et nb différents »
2    | Résultat <- na <> nb
3
4  R3 : Raffinage De « Soustraire au plus grand le plus petit »
5    | Si na > nb Alors
6    |   | na <- na - nb
7    | Sinon
8    |   | nb <- nb - na
9    | FinSi
```

Remarque : Quand on raffine une expression (premier R3), on indique la *valeur* de cette expression en initialisant **Résultat**.

Qualités d'un raffinage

- Un raffinage doit être bien présenté (indentation).
- Le vocabulaire utilisé doit être précis et clair.
- Chaque niveau de raffinage doit apporter suffisamment d'information (mais pas trop). Il faut trouver le bon équilibre !
- Le raffinage d'une étape (l'ensemble des sous-étapes) doit décrire complètement cette étape.
- Le raffinage d'une étape ne doit décrire que cette étape.
- Les étapes introduites doivent avoir un niveau d'abstraction homogène.
- La séquence des étapes doit pouvoir s'exécuter logiquement.
- Ne pas employer de structures de contrôle déguisées (si, jusqu'à) **mais** on peut (doit) utiliser des structures de contrôle (**`Si`**, **`TantQue`**, **`Répéter`**...) !
- N'utiliser qu'une seule structure de contrôle par raffinage.

Remarque : Certaines de ces règles sont subjectives !
L'important est que vous puissiez expliquer et justifier les choix faits.

Vérification d'un raffinage

- A-t-on utilisé des verbes à l'infinitif pour les étapes ?
- Pour être correct, un raffinage doit répondre à la question COMMENT !
- Pour vérifier l'appartenance d'une étape *e* au raffinage d'une étape *s*, se demander POURQUOI on fait cette étape *e*.
  $\Longrightarrow$ Ceci permet d'identifier :
  - soit une étape intermédiaire entre *e* et *s* ;
  - soit que *e* n'est pas une sous-étape de *s* (donc pas à sa place).
- Utiliser les flots de données :
  - pour vérifier les communications entre niveaux :
    - les sous-étapes doivent produire les résultats de l'étape ;
    - les sous-étapes peuvent (doivent) utiliser les entrées de l'étape.
  - au sein d'un niveau : enchaînement des **`in`**, **`out`** et **`in out`**. On ne peut utiliser une donnée (**`in`**) que si elle a été produite (**`out`**) avant.

Dictionnaire des données

La liste des données (variables) utilisées avec leur signification.

Algorithme

Un *algorithme* est la mise à plat du raffinage :

- R0 devient le commentaire général de l'algorithme.
- En faisant un parcours en profondeur d'abord, de la gauche vers la droite,
  - les étapes élémentaires sont les instructions ;
  - les étapes intermédiaires deviennent des commentaires.

Remarque : L'obtention de l'algorithme est directe si les derniers niveaux du raffinage ne contiennent que des étapes élémentaires.
Sinon, il faut expliciter ces étapes non raffinées !

Le programme

C'est la traduction de l'algorithme dans un langage de programmation.

Algorithme du pgcd

```
1  Algorithme pgcd_euclide
2      -- Afficher le pgcd de deux entiers strictement positifs
3  Variables
4      a, b: Entier          -- deux entiers saisis au clavier
5      pgcd: Entier          -- le pgcd de a et b
6      na, nb: Entier        -- nouveau a et nouveau b (pour calculer le pgcd)
7  Début
8      -- Saisir deux entiers
9          ...
10     { (a > 0) Et (b > 0) }
11
12     -- Déterminer le pgcd de a et b
13     na <- a      -- variables auxiliaires car a et b sont en in
14     nb <- b      -- et ne peuvent donc pas être modifiées.
15     TantQue na <> nb Faire       -- na et nb différents
16         -- Soustraire au plus grand le plus petit
17         Si na > nb Alors
18             na <- na - nb
19         Sinon
20             nb <- nb - na
21         FinSi
22     FinTQ
23     pgcd <- na  -- pgcd était en out, il doit être initialisé.
24
25     -- Afficher le pgcd
26     Écrire("pgcd = ", pgcd)
27 Fin
```

Programme du pgcd en Ada

```
1  with Ada.Text_IO;           use Ada.Text_IO;
2  with Ada.Integer_Text_IO;   use Ada.Integer_Text_IO;
3
4  -- Afficher le pgcd de deux entiers strictement positifs
5  procedure Pgcd_Euclide is
6      a, b    : Integer;  -- deux entiers saisis au clavier
7      pgcd    : Integer;  -- le pgcd de a et b
8      na, nb  : Integer;  -- nouveau a et nouveau b (pour calculer le pgcd)
9  begin
10     -- Saisir deux entiers
11     ...
12     --{ (a > 0) Et (b > 0) }
13
14     -- Déterminer le pgcd de a et b
15     na := a;    -- variables auxiliaires car a et b sont en in
16     nb := b;    -- et ne peuvent donc pas être modifiées.
17     while na /= nb loop -- na et nb différents
18         -- Soustraire au plus grand le plus petit
19         if na > nb then
20             na := na - nb;
21         else
22             nb := nb - na;
23         end if;
24     end loop;
25     pgcd := na; -- pgcd était en out, il doit être initialisé.
26
27     -- Afficher le pgcd
28     Put ("pgcd = ");
29     Put (pgcd, 1);
30     New_Line;
31 end Pgcd_Euclide;
```

Programme du pgcd en Pascal

```
1   program pgcd_euclide;
2       (* Afficher le pgcd de deux entiers strictement positifs *)
3   var
4       a, b: Integer;  (* deux entiers saisis au clavier *)
5       pgcd: Integer;  (* le pgcd de a et b *)
6       na, nb: Integer;    (* utilisées pour le calcul du pgcd *)
7   begin
8       (* Saisir deux entiers *)
9       ...
10      { (a > 0) Et (b > 0) }
11
12      (* Déterminer le pgcd de a et b *)
13      na := a;    (* variables auxiliaires : ceci permet *)
14      nb := b;    (* de conserver les valeurs saisies *)
15      while na <> nb do       (* na et nb différents *)
16          (* Soustraire au plus grand le plus petit *)
17          if na > nb then
18              na := na - nb
19          else
20              nb := nb - na;
21      pgcd := na;
22
23      (* Afficher le pgcd *)
24      write('pgcd = ');
25      writeln(pgcd);
26  end.
```

Programme du pgcd en C

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

/* Afficher le pgcd de deux entiers strictement positifs */
int main() {
    /* Saisir deux entiers */
    int a, b;   /* deux entiers saisis au clavier */
    int pgcd;   /* le pgcd de a et b */
    // ...
    assert(a > 0);
    assert(b > 0);

    /* Déterminer le pgcd de a et b */
    int na = a; /* nouveau a pour ne pas modifier a */
    int nb = b; /* idem avec b */
    while (na != nb) {      /* na et nb différents */
        /* Soustraire au plus grand le plus petit */
        if (na > nb) {
            na = na - nb;
        }
        else {
            nb = nb - na;
        }
    }
    pgcd = na;

    /* Afficher le pgcd */
    printf("pgcd = %d\n", pgcd);

    return EXIT_SUCCESS;
}
```

Programme du pgcd en Python

```
1  # Afficher le pgcd de deux entiers strictement positifs
2
3  # Saisir deux entiers
4  ...
5
6  assert a > 0
7  assert b > 0
8
9  # Déterminer le pgcd de a et b
10 na = a  # nouveau a pour ne pas modifier a
11 nb = b  # idem pour b
12 while na != nb: # na et nb différents
13     # Soustraire au plus grand le plus petit
14     if na > nb:
15         na = na - nb
16     else:
17         nb = nb - na
18 pgcd = na   # le pgcd de a et b
19
20 # Afficher le pgcd
21 print('pgcd ='', pgcd)
```

Programme du pgcd en Java

```
1   /** Afficher le pgcd de deux entiers strictement positifs */
2   class Pgcd {
3       public static void main(String[] args) {
4           // Saisir deux entiers
5           int a, b;   // deux entiers saisis au clavier
6           ...
7           assert a > 0;
8           assert b > 0;
9
10          // Déterminer le pgcd de a et b
11          int na = a; // nouveau a pour ne pas modifier a
12          int nb = b; // idem pour b
13          while (na != nb) {      // na et nb différents
14              // Soustraire au plus grand le plus petit
15              if (na > nb) {
16                  na = na - nb;
17              } else {
18                  nb = nb - na;
19              }
20          }
21          int pgcd = na;      // le pgcd de a et b
22
23          // Afficher le pgcd
24          System.out.println("pgcd␣=␣" + pgcd);
25      }
26  }
```

Attention : Ce n'est pas un programme Java représentatif !

Tester le programme

Tester : processus d'exécution d'un programme avec l'intention de découvrir des erreurs !

Deux grands types de tests :

- *tests fonctionnels :* le programme fait-il ce qu'on veut qu'il fasse ? (programme vu comme une « boîte noire »)
- *tests structurels :* toutes les parties du code ont-elles été exercées ? (programme vu comme une « boîte blanche »)

Principe : Échantillonnage qui repose sur l'hypothèse implicite que si le programme testé fournit des résultats corrects pour l'échantillon choisi, il fournira aussi des résultats corrects pour toutes les autres données.

Problème : Comment choisir l'échantillon ?

- Dans le cas de tests fonctionnels, on peut établir une matrice pour vérifier que chaque exigence a été testée.
- Dans le cas de tests structurels, on peut s'appuyer sur la notion de taux de couverture. Par exemple, a-t-on exécuté au moins une fois toutes les instructions ? Est-on passé au moins une fois par tous les enchaînements.

Pb de l'oracle : Comment savoir que le résultat du programme est correct ?
Attention : On peut commencer à tester dès l'écriture d'un raffinage.

Comment construire un algorithme ?

Exercice :
Écrire un algorithme qui structure les étapes nécessaires pour construire un algorithme. On fera apparaître les flots de données au moins pour les premiers niveaux de raffinage.

Sommaire

- Éléments d'un algorithme/programme
- Constantes
- Expressions
- Instruction d'entrée/sortie
- Affectation
- Structures de contrôle
- Choisir la bonne structure de contrôle

Les commentaires

Deux types de commentaires avec une signification propre :

1. Les commentaires introduits par `--` qui se terminent avec la ligne :
   - pour rappeler les étapes non élémentaires identifiées lors du raffinage ;
   - pour expliciter le rôle d'une variable : après la déclaration ;
   - pour expliquer une instruction : mis à la fin de la ligne
2. Les commentaires entre accolades :
   - pour exprimer une propriété qui doit être vérifiée à l'exécution du programme.
   - Voir aussi structures de contrôle.

Exercice 3 : Lien entre raffinage et algorithme
Donner le raffinage qui a conduit à l'algorithme décrit transparent 4.

Éléments d'un algorithme

Un algorithme est composé de trois catégories principales d'éléments :

1. une **définition** permet de définir les « entités » qui pourront être manipulées dans l'algorithme. En particulier, on définit des *constantes*, des *types* et des *sous-programmes*.
2. une **déclaration** permet de déclarer les données qui sont utilisées par le programme. Les données sont stockées par des variables.
3. Les **instructions** constituent le **programme principal**. Elles sont exécutées par l'ordinateur pour transformer les données.

Les identificateurs

- **Identificateur** : nom donné à une entité d'un programme :
  - le programme,
  - les variables,
  - les constantes,
  - les types,
  - ...
- **Objectifs :** Les identificateurs permettent :
  - à l'ordinateur de distinguer les entités et
  - aux hommes de comprendre leur rôle et intérêt
- **Règle :** Un identificateur commence par une lettre suivie de chiffres, lettres et soulignés.
- **Notation :** `rayon`, `prix_ttc`, `prix_ht`, `n1`, `n2`, etc.

Les variables

- **variable** : une zone dans la mémoire (vive) de l'ordinateur où est conservée une donnée manipulée par le programme.
- Une variable est caractérisée par :
  - **rôle** : ce à quoi va servir la variable ;
  - **nom** : l'identificateur de la variable (significatif !) ;
    $\Longrightarrow$ Compromis entre significatif et longueur
    (exemple : `rayon` plutôt que `le_rayon_du_cercle`)
  - **type** : caractériser l'ensemble des valeurs que peut prendre la variable
    Exemple : Le rayon prend des valeurs réelles (**`Réel`**)
  - **valeur** : l'information stockée en mémoire.
    - Elle peut changer au cours de l'exécution du programme.
    - À la déclaration de la variable, la valeur est indéterminée (notée « ? »).
- Le rôle, le nom et le type sont des informations **statiques** (connues à l'écriture du programme, avant l'exécution).

```
nom: Type        -- rôle
```

- La valeur est une information **dynamique** (connue à l'exécution du programme).

Les types

- Un type :
  - **caractérise les valeurs** que peut prendre une variable ;
  - **définit les opérations** (ou opérateurs) qui pourront être appliquées sur les données de ce type.
- **types fondamentaux** : types prédéfinis dans notre langage algorithme.
  Synonymes : types de base, types primitifs
- Les types ont deux **intérêts principaux** :
  - détecter des incohérences sur les opérations (fait par le compilateur) :
    - soustraction entre un entier et une chaîne de caractères
  - Connaître la place nécessaire en mémoire pour stocker la valeur de la variable (transparent pour le programmeur).
- **Opérateurs de comparaison :** Tous les types fondamentaux sont munis des opérateurs de comparaison : <, >, <=, >=, = et <> (à valeur booléenne).

Les entiers

Le type **Entier** caractérise les entiers relatifs.
Exemples de constantes littérales entières :

```
1  10
2  0
3  -421
```

Les opérations sont :

```
1  +        -- addition                    3 + 4 = 7
2  -        -- soustraction                3 - 4 = -1
3  *        -- multiplication              3 * 4 = 12
4  Div      -- division entière !          3 Div 4 = 0  et  10 Div 3 = 3
5  Mod      -- reste division              3 Mod 4 = 3  et  10 Mod 3 = 1
6  -        -- opposé (unaire)             -3
7  +        -- plus unaire                 +5
```

Les réels

Le type **Réel** caractérise les réels.
Exemples de constantes littérales réelles :

```
10.0
0.0
-10e-4
```

Les opérations sont :

```
+       -- addition                    3 + 4 = 7
-       -- soustraction                3 - 4 = -1
*       -- multiplication               3 * 4 = 12
/       -- division                     3 / 4 = 0.75  et  10 / 3 = 3.333...
-       -- opposé (unaire)              -3
+       -- plus unaire                  +5
```

Surcharge : Le même nom (ici le même symbole) peut être utilisé pour nommer des opérations différentes :

- + unaire et + binaire
- + sur les entier et + sur les réels

Remarque : il aurait été possible d'utiliser / sur les entiers (cf C, Java)

Les booléens

- Le type **Booléen** caractérise les valeurs booléennes, soit **VRAI** soit **FAUX**.

Les opérateurs logiques **Et**, **Ou**, **Non** sont définis par leur table de vérité

A	B	A **Et** B	A **Ou** B	**Non** A
VRAI	**VRAI**	**VRAI**	**VRAI**	**FAUX**
VRAI	**FAUX**	**FAUX**	**VRAI**	**FAUX**
FAUX	**VRAI**	**FAUX**	**VRAI**	**VRAI**
FAUX	**FAUX**	**FAUX**	**FAUX**	**VRAI**

- **Lois de De Morgan :** Les lois de De Morgan sont particulièrement importantes à connaître (cf structures de contrôle, transparent 51).

```
1  Non (A Et B) = (Non A) Ou (Non B)
2  Non (A Ou B) = (Non A) Et (Non B)
3  Non (Non A)  = A
```

- **Exercice 4** Vérifier les formules de De Morgan (avec des tables).

- **Exercice 5** Un éco-prêt à taux zéro permet de financer des travaux d'économie d'énergie. Le bénéficiaire doit être propriétaire du bâtiment. Le bâtiment doit être construit avant le 1er janvier 1990. À quelle condition un éco-prêt peut être obtenu ? Refusé ?

Les booléens (2)

- **Évaluation partielle :** L'évaluation d'une expression booléenne s'arrête dès le résultat connu

```
1  FAUX Et expr    -- toujours FAUX sans avoir à évaluer expr
2  VRAI Ou expr    -- toujours VRAI sans avoir à évaluer expr
```

- **Synonyme :** évaluation en *court-circuit*
- **Intérêt :** `(n <> 0) Et ((s Div n) >= 10)`
  Que se passe-t-il si n vaut 0 ?
- **Attention :** Tous les langages n'ont pas l'évaluation en court-circuit.
- Si A est une expression booléenne, on a :

```
1  A         est équivalent à   A = VRAI
2  Non A     est équivalent à   A = FAUX
```

  Il est préférable d'utiliser A et **Non** A. Les deux autres formulations, si elles ne sont pas fausses, sont des pléonasmes !

Les caractères

- Le type **Caractère** caractérise les caractères.
- Exemples de constantes littérales de type caractère (langage C/Java) :

```
1  'a'     -- le caractère  a
2  '\n'    -- Fin de ligne
3  '\t'    -- Caractère tabulation
4  '\''    -- le caractère  '
5  '\\'    -- le caractère  \
```

- **Remarque :** Les lettres majuscules, les lettres minuscules et les chiffres se suivent. Ainsi, un caractère c est une majuscule ssi `(c >= 'A') Et (c <= 'Z')`.
- **Opérations :**
  - `Ord` : Permet de convertir un caractère en entier.
  - `Chr` : Permet de convertir un entier en caractère.

  Pour tout `c: Caractère` on a : `Chr(Ord(c)) = c`.
- **Question :** Que vaut `ord('0')` ?

Les chaînes de caractères

Le type **Chaîne** caractérise les chaînes de caractères.

```
"Une chaîne de caractères"                    -- un exemple de Chaîne
```

Remarque : La manière de traiter les chaînes de caractères peuvent être très différentes suivant les langages.

Attention : Il ne faut pas confondre le nom d'une variable et une constante chaîne de caractères !
Exemple :

```
Écrire("prix")
Écrire(prix)
```

Constantes

- **Définition :** Une constante est une information dont la valeur ne change pas au cours de l'exécution d'un programme.
- **Exemple :**

```
1  PI = 3.1415                -- Valeur de PI
2  MAJORITÉ = 18              -- Âge correspondant à la majorité
3  TVA = 19.6                 -- Taux de TVA en vigueur au 17/09/2012 (en %)
4  CAPACITÉ = 120             -- Nombre maximum d'étudiants dans une promotion
5  INTITULÉ = "Algorithmique␣et␣programmation"     -- par exemple
```

- **Règle :** Ne jamais utiliser une constante littérale (sauf -1, 0 et 1) mais une constante symbolique !
- **Intérêt :** Deux avantages principaux :
  - rendre plus lisible le programme en ayant des noms clairs ;
  - paramètrer le programme et en faciliter les évolutions.

Expressions

- **Définition :** Une expression est une phrase syntaxique qui a une valeur.
- Exemples :
  - une constante littérale ou symbolique ;
  - une variable ;
  - un opérateur appliqué sur des opérandes (donc des expressions).
- **Exemple :** Quelques exemples d'expressions à valeur réelle :

```
1  10.0             -- constante littérale
2  PI               -- constante symbolique
3  rayon            -- variable de type réel
4  2 * rayon        -- opérateur * appliqué sur 2 et rayon
5  2 * PI * rayon   -- expression mêlant opérateurs, constantes, variables
```

- **Attention :** Pour qu'une expression soit acceptable, il est nécessaire que les types des opérandes d'un opérateur soient compatibles.

Typages

Typage statique : Les types sont connus et vérifiés avant l'exécution du programme.

- **explicite :** les types sont donnés par le programmeur (Ada, C, Java...)
- **implicite :** les types sont inférés (Caml, Scala, etc.)
- **Typage fort :** pas de compatibilité : les types doivent être identiques (Ada, Caml....).
- **Conversions implicites :** Certains langages offrent une compatibilité entre type via des conversions implicites (coercitions) :
  si un type A est compatible avec un type B, on peut mettre une expression de type A partout où une expression de type B est attendue.

  Règles de compatibilité :

  - Un type est compatible avec lui-même !
  - un type A est compatible avec un type B si et seulement si le passage d'un type A à un type B se fait sans perte d'information.
    - En d'autres termes, tout élément du type A a un correspondant dans le type B.
    - Par exemple, **`Entier`** est compatible avec **`Réel`** mais l'inverse est faux.
    - Il y a **coercition** : transformation des données.
  - Un sous-type est compatible avec ses super-types (voir sous-typage).

Typage dynamique : les types sont considérés à l'exécution (Python, etc.)

Évaluation d'une expression

- Une expression est évaluée :
  - en commençant par les opérateurs de priorité la plus élevée,
  - à priorité égale, les opérateurs les plus à gauche (associativité à gauche)

```
a + b * c + d ≡ ((a + (b * c)) + d)
```

- Table de priorité des opérateurs

1	*+, -,* ***Non*** *(unaires)*	*priorité la plus forte*
2	**, /,* ***Div***, ***Mod***, ***Et***	
3	*+, -,* ***Ou***	
4	*<, >, <=, >=, = et <>*	*priorité la plus faible*

- Les parenthèses permettent de modifier les priorités.

```
prix_ht * (1 + tva)          -- prix_ttc
```

- Comment est évaluée : `x > 0 Et y <= 100` ?

Exemple : les expressions booléennes

Une expression booléenne peut être constituée par :

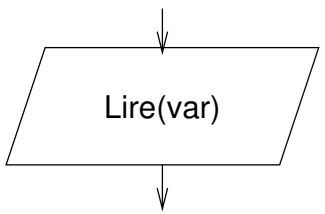
- une constante booléenne (**VRAI** ou **FAUX**) ;
- une variable booléenne ;
- une comparaison entre deux expressions de même type ($=, \neq, <, >, \leq, \geq$) ;
- la négation d'une expression booléenne (**Non** `expression`) ;
- la composition d'un opérateur logique binaire (**Et** ou **Ou**) sur deux expressions booléennes.

Instructions d'entrée/sorties

- Elles permettent de faire le **lien entre le programme et l'extérieur** :
  - les périphériques de sortie (généralement l'écran) ;
  - les périphériques d'entrée (généralement le clavier).
- Le **point de vue** est celui du programme :
  - l'information vient du clavier pour aller vers le programme (entrée)
  - l'information sort du programme pour aller sur l'écran (sortie)
- L'écran et le clavier sont des périphériques qui **ne traitent que des caractères** (les touches du clavier !).
  - les opérations de sortie (**`Écrire`**) transforment les données du programme en une suite du caractères ;
  - les opérations d'entrée (**`Lire`**) transforment une suite de caractères en données pour le programme.
  - Ces transformations dépendent du type des données du programme.

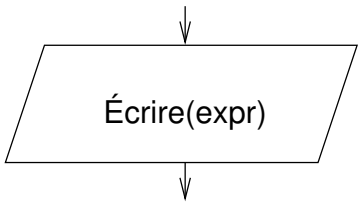
Opération d'entrée : Lire

- **`Lire(var)`** : lire sur le périphérique d'entrée une valeur et la ranger dans la variable `var`.
- Autre formulation : affecter la variable `var` avec une valeur lue au clavier.
- **Règle :** Le paramètre « var » est nécessairement une variable. En effet, la valeur lue sur le périphérique d'entrée doit être rangée dans « var ».
- **Évaluation :** L'évaluation se fait en deux temps :
  - récupérer l'information sur le périphérique d'entrée (elle est convertie dans le type de la variable var) ;
  - ranger cette information dans la variable var.
- **Organigramme :**



Opération de sortie : Écrire

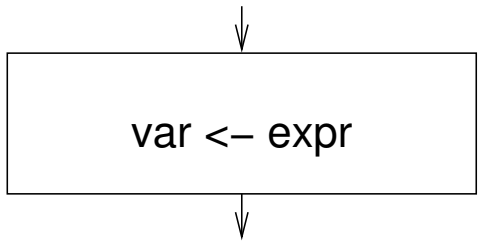
- **`Écrire`**`(expr)` : transférer (afficher, imprimer...) la valeur de l'expression `expr` vers le périphérique de sortie.
- **Règle :** « expr » est une expression quelconque.
- **Évaluation :** L'évaluation se fait en deux temps :
  - évaluer l'expression (c'est-à-dire calculer sa valeur) ;
  - afficher (ajouter) sur le périphérique de sortie la valeur de l'expression.
- **Organigramme :**



- **Variante :** `EcrireLn(expr)` ajoute un saut de ligne sur le périphérique de sortie après avoir affiché la valeur de l'expression.

Affectation

```
var <- expr
```



- **Définition :** L'affectation permet de modifier la valeur associée à une variable.
- **Règle :** Le type de `expr` doit être compatible avec le type de `var`.
- **Évaluation :** L'évaluation de l'affectation se fait en deux temps :
  1. calculer la valeur de l'expression `expr` ;
  2. ranger cette valeur dans la variable `var`.

Exemples :

```
1  rayon <- 10
2  diamètre <- 2 * rayon
3  périmètre <- PI * diamètre
```

Exercice 6 Quelles valeurs prend n lors de l'exécution des instructions suivantes ?

```
1  n <- 10
2  n <- n + 1
```

Structures de contrôle

- L'état à l'exécution d'un programme (langage impératif) est défini par :
  - l'ensemble des valeurs des variables du programme ;
  - l'instruction qui doit être exécutée.
- L'exécution d'un programme est alors une séquence d'affectations qui font passer d'un état initial (valeurs indéterminées) à un état final (résultat).
- Les **structures de contrôle** décrivent l'enchaînement des affectations (le transfert du contrôle après la fin de l'exécution d'une instruction) :
  1. enchaînement séquentiel (une instruction puis la suivante) ;
  2. traitements conditionnels ;
  3. traitements répétitifs (itératifs) ;
  4. appel d'un sous-programme (voir sous-programmes).
- **Attention :** Le `goto` est banni... depuis 1970 ! ! !

Enchaînement séquentiel

- Les instructions sont exécutées dans l'ordre où elles apparaissent.

```
1  instruction_1
2  ...
3  instruction_n
```

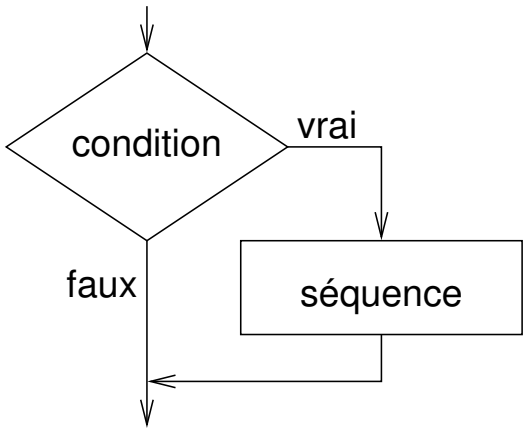
- **Exemple :**

```
1  mémoire <- a              -- première instruction
2  a <- b                    -- deuxième instruction
3  b <- mémoire              -- troisième instruction
```

- **Question :** Quand utilisera-t-on cette séquence de trois instructions ? Que permet-elle de faire ?
- **Remarque :** On utilise parfois le terme d'*instruction composée*.

Conditionnelle Si ... Alors ... FinSi

```
1  Si condition Alors
2      séquence -- d'instructions
3  FinSi
```

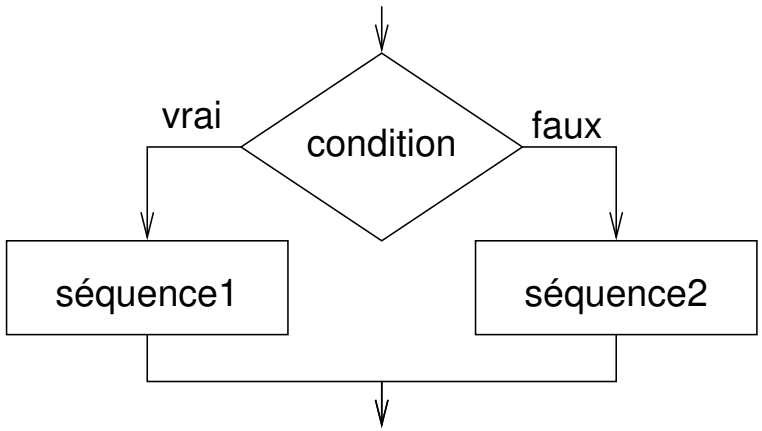


- **Règle :** La condition est nécessairement une expression booléenne.
- **Évaluation :**
  - la condition est évaluée ;
  - si la condition est vraie, la séquence est exécutée puis le contrôle passe à l'instruction qui suit le **FinSi** ;
  - si la condition est fausse, le contrôle passe à l'instuction qui suit le **FinSi**.
- La séquence est exécutée si et seulement si la condition vaut **VRAI**.

Exercice 7 Afficher "paire" si une valeur entière est paire.

Conditionnelle Si ... Alors ... Sinon ... Finsi

```
1  Si condition Alors
2      séquence1
3  Sinon   { Non condition }
4      séquence2
5  FinSi
```



- **Évaluation :** Si la condition est vraie, c'est séquence_1 qui est exécutée, sinon c'est séquence_2. Dans les deux cas, après l'exécution de la séquence, l'instruction suivante à exécuter est celle qui suit le **FinSi**.
- Possible de faire apparaître dans un commentaire la condition associée à la partie **Sinon**

Exercice 8 Déterminer la plus grande de deux valeurs réelles.

Exercice 9 Réécrire un **Si ... Sinon** sans utiliser le **Sinon**. Équivalent ? Commentaires ?

Solution de l'exercice 8

```
1  Comment « Déterminer le plus grand des deux entiers » ?
2              x1, x2: in Integer  -- les deux entiers
3              max: out Integer    -- le plus grand des deux
4      Si x1 > x2 Alors
5          max <- x1
6      Sinon
7          max <- x2
8      FinSi
```

La clause SinonSi

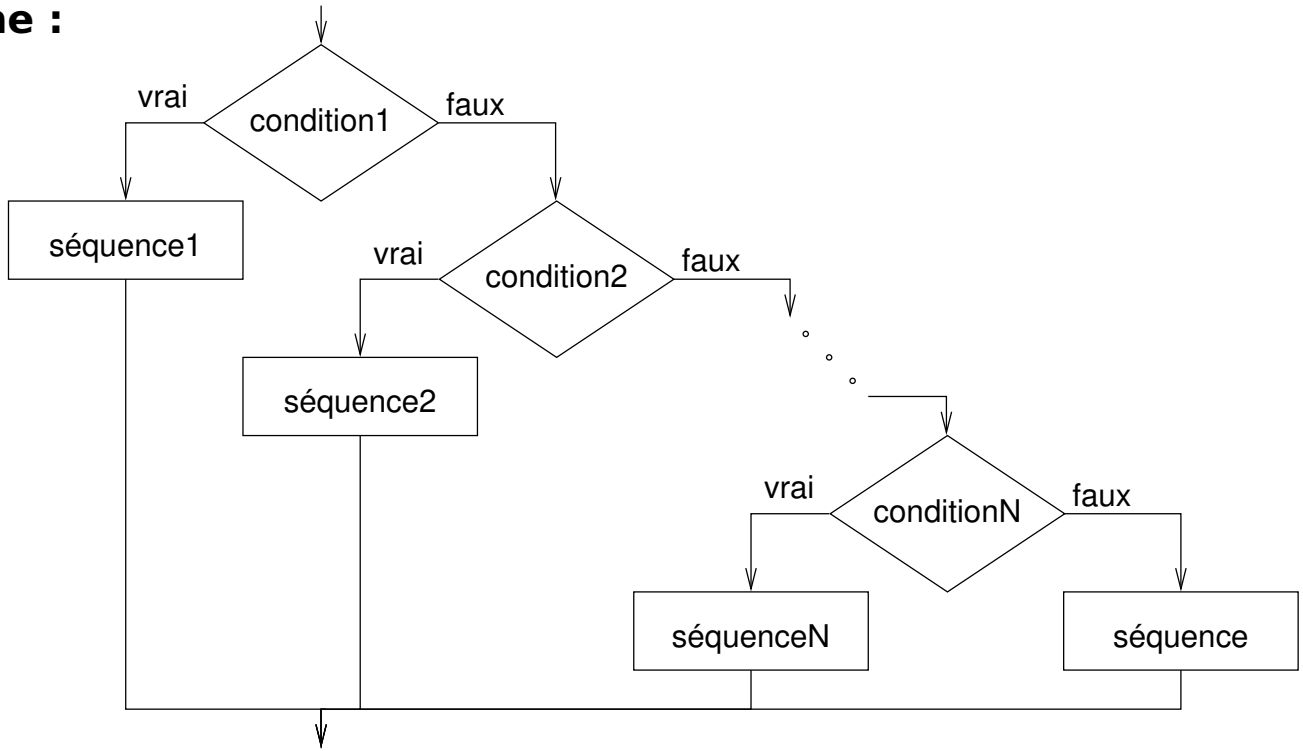
La conditionnelle **Si ... Alors ... Sinon ... FinSi** peut être complétée avec des clauses **SinonSi** suivant le schéma suivant :

```
1  Si condition_1 Alors
2      séquence_1
3  SinonSi condition_2 Alors
4      séquence_2
5  ...
6  SinonSi condition_N Alors
7      séquence_N
8  Sinon   { Expliciter la condition ! }
9      séquence
10 FinSi
```

Évaluation : Les conditions sont évaluées dans l'ordre d'apparition. Dès qu'une condition est vraie, la séquence associée est exécutée. L'instruction suivante à exécuter sera alors celle qui suit le **FinSi**. Si aucune condition n'est vérifiée, la séquence associée au **Sinon**, si elle existe, est exécutée.

La clause SinonSi (2)

Organigramme :



Exercice 10 Afficher si un entier est positif, négatif ou nul.

Solution de l'exercice 10

```
1  Comment « Afficher le signe d'un entier » ?
2              n: in Entier          -- l'entier
3      Si n > 0 Alors
4          Écrire("positif");
5      SinonSi  n < 0 Alors
6          Écrire("négatif");
7      Sinon  { Non (n > 0) Et Non (n < 0)  donc  n = 0 }
8          Écrire("nul");
9      FinSi
```

Conditionnelle Selon

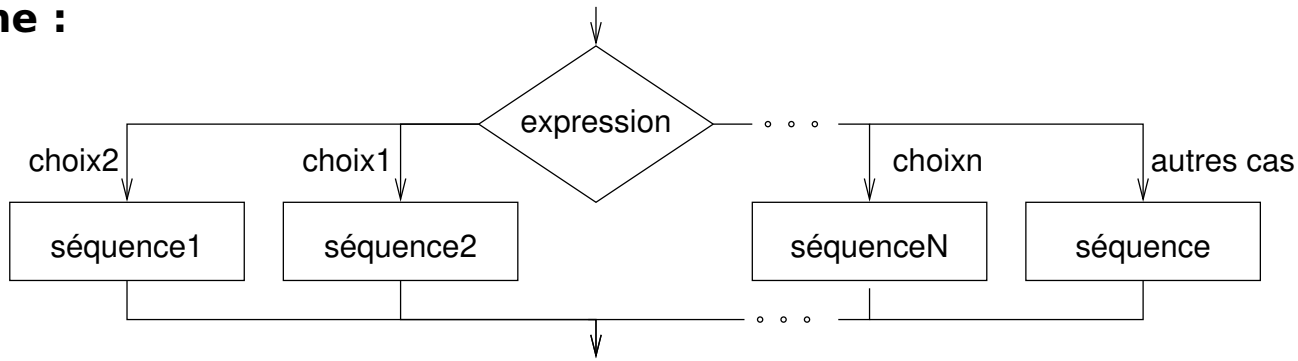
```
1  Selon expression Dans
2      choix_1 :
3          séquence_1
4      ...
5      choix_N :
6          séquence_N
7  Sinon
8      séquence
9  FinSelon
```

Règles :

- expression est nécessairement une expression de type scalaire.
- choix_i est une liste de choix séparés par des virgules. Chaque choix est soit une constante, soit un intervalle (`10..20`, par exemple).
  Selon est donc un cas particulier de **Si ... SinonSi ... FinSi**.

Conditionnelle Selon (2)

Évaluation : L'`expression` est évaluée, puis sa valeur est successivement comparée à chacun des ensembles choix$_i$. Dès qu'il y a correspondance, la séquence associée est exécutée et l'exécution continue après le **`FinSelon`**. Si aucun choix$_i$ ne correspond, la séquence du **`Sinon`** est exécutée.
Conséquence : Les différents choix sont donc *exclusifs*.
Remarque : La clause **`Sinon`** est optionnelle... mais il faut être sûr de traiter tous les cas (toutes les valeurs possibles de l'expression).
Organigramme :



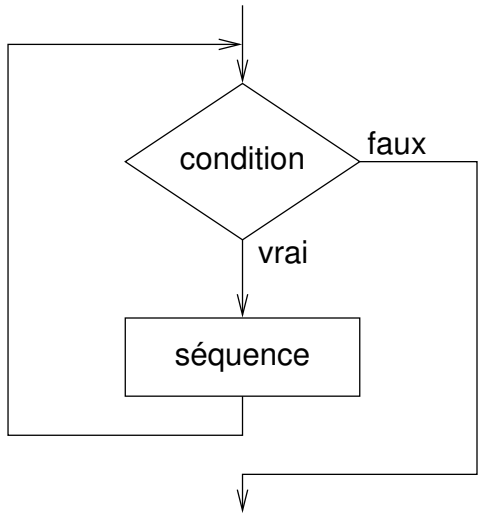
Exercice 11 Indiquer si un caractère est une lettre minuscule, une lettre majuscule, un chiffre ou un caractère de ponctuation.

Solution de l'exercice 11

```
1  Comment « Afficher la nature d'un caractère » ?
2              c: in Caractère
3      Selon c Dans
4      '0'..'9':
5          Écrire("chiffre");
6      'a'..'z':
7          Écrire("lettre minuscule");
8      'A'..'Z':
9          Écrire("lettre majuscule");
10     '.', ',', ';', '!', '?', ':':
11         Écrire("ponctuation");
12     Sinon
13         Écrire("autre")
14     FinSelon
```

Répétition **TantQue**

```
1  TantQue condition Faire
2      séquence
3  FinTQ
```



Règles :

- la condition doit être une expression booléenne ;
- la séquence *doit* modifier la condition (terminaison).

Évaluation : La condition est évaluée. Si elle vaut **FAUX** alors la boucle se termine. Si elle vaut **VRAI** alors la séquence d'instructions est exécutée et la condition est de nouveau évaluée.

Répétition **TantQue** (2)

Exercice 12 : Condition après un `FinTQ`

Que sait-on sur l'état du programme lorsque l'instruction suivante à exécuter est celle qui suit le **`FinTQ`** ?

Remarque : Comme le test de la condition est fait en premier, la séquence peut ne pas être exécutée. Il suffit que la condition soit fausse dès le début.

Exercice 13 : Somme des premiers entiers (TantQue)

Calculer la somme des n premiers entiers.

Exercice 14 : Saisie contrôlée d'un numéro de mois

On souhaite réaliser la saisie du numéro d'un mois (compris entre 1 et 12) avec vérification. Le principe est que si la saisie est incorrecte, le programme affiche un message expliquant l'erreur de saisie et demande à l'utilisateur de resaisir la donnée.

14.1 Utiliser un **`TantQue`** pour réaliser la saisie contrôlée.

14.2 Généraliser l'algorithme au cas d'une saisie quelconque.

Solution de l'exercice 13

```
1  Comment « somme des n premiers entiers » ?
2              n: in Entier -- positif
3              somme: out Entier
4
5      Principe : ajouter successivement chacun des entiers de 1 à n.
6
7  Variable
8      i: Entier              -- parcourir les entiers de 1 à n
9  Début
10     somme <- 0
11     i <- 1
12     TantQue i <= n Faire
13         somme <- somme + i
14         i <- i + 1
15     FinTQ
16 Fin.
```

Solution de l'exercice 14

```
1   R0 : Saisir avec contrôle un numéro de mois compris entre 1 et 12
2
3       Exemples :
4               1  -> 1
5               5  -> 5
6               15 ( -> trop grand) 0 (-> trop petit) 12 -> 12
7
8   R1 : Comment « Saisir avec contrôle un numéro de mois »
9       Saisir numéro de mois                 numéro : out Entier
10      TantQue numéro invalide Faire         numéro : in
11          Expliquer l erreur                numéro : in
12          Saisir un nouveau numéro          numéro : out
13      FinTQ
14
15  R2: Comment « Saisir un numéro de mois »
16      Écrire("Numéro de mois (1-12) : ")
17      Lire(numero)
18
19  R2: Comment « numéro invalide »
20      Résultat <- (numéro < 1) Ou (numéro > 12)
21
22  R2: Comment « Expliquer l erreur »
23      Si numéro < 1 Alors
24          Écrire("Le numéro doit être > 0")
25      SinonSi numéro > 12 Alors
```

Solution de l'exercice 14 (2)

```
26          Écrire("Le numéro doit être < 12")
27      FinSi
28
29  R2: Comment << Saisir un nouveau numéro »
30      Écrire("Recommencez !")
31      Écrire("Numéro de mois (1-12) : ")
32      Lire(numero)
```

Version générale :

```
1       Saisir les données
2       TantQue saisie incorrecte Faire
3           Signaler l'erreur de saisie
4           Saisir les nouvelles données
5       FinTQ
```

Cohérence d'une répétition : variant et invariant

Variant : Quantité positive qui décroit strictement à chaque itération.
Intérêt : Garantir (prouver) la terminaison de la boucle.
Invariant : Expression booléenne vraie avant et après chaque itération.
Intérêt : Permet de prouver que la répétition est juste.
Attention : Il faut prouver le variant et l'invariant !

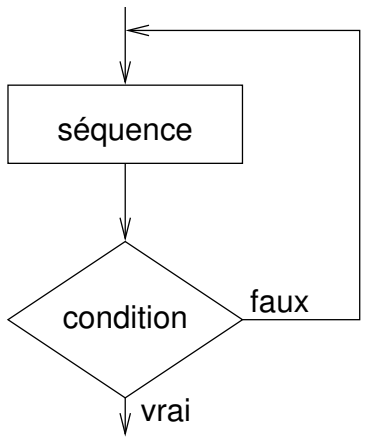
```
1  TantQue condition Faire
2      { Variant : V }
3      { Invariant : I }
4      instructions
5  FinTQ
6  { Non condition Et (V >= 0) Et I }
```

Attention : Si le variant est généralement simple, l'invariant est souvent dur à formaliser. Aussi, on se contentera d'un commentaire informel qui explique l'invariant de la boucle (l'objet d'une itération).

Exercice 15 Donner le variant et l'invariant des exercices 13 et 14.

Répétition Répéter ... JusquÀ

```
1  Répéter
2      séquence
3  JusquÀ condition
```



Évaluation : La séquence est exécutée, puis la condition est évaluer. Si la condition est fausse, on recommence : exécution de la séquence et évaluation de la condition.

Remarques :

- la condition n'est évaluée qu'après l'exécution de la séquence ;
- la séquence est exécutée au moins une fois ;
- la séquence *doit* modifier la condition (terminaison).

Répétition Répéter ... JusquÀ (2)

Exercice 16 : Plusieurs sommes des n premiers entiers

Écrire un programme qui affiche la somme des n premiers entiers naturels, n étant un entier saisi au clavier. Le programme devra proposer la possibilité à l'utilisateur de recommencer le calcul pour un autre entier.

Exercice 17 : Saisie contrôlée d'un numéro de mois

On souhaite réaliser la saisie du numéro d'un mois (compris entre 1 et 12) avec vérification. Le principe est que si la saisie est incorrecte, le programme affiche un message expliquant l'erreur de saisie et demande à l'utilisateur de resaisir la donnée.
On utilisera un **`Répéter`** pour réaliser la saisie contrôlée.
Généraliser l'algorithme au cas d'une saisie quelconque.

Exercice 18 : TantQue et Répéter

Écrire la répétition **`Répéter`** à partir du **`TantQue`** et réciproquement.

Solutions

Exercice 16

```
Comment « Afficher la somme des n premiers entiers avec possibilité de recommencer » ?
    Répéter
       Afficher la somme des n premiers entiers
       Demander si l'utilisateur veut recommencer        reponse: out
    JusquÀ réponse est non
```

Exercice 17

```
1  Comment « Saisir un numéro de mois avec contrôle » ?
2      Répéter
3          Saisir les données
4          Si saisie incorrecte Alors
5              Signaler l'erreur de saisie
6              Indiquer qu'une nouvelle saisie doit être réalisée
7          FinSi
8      JusquÀ saisie correcte
```

Répétition Pour

```
1  Pour var <- val_min [ Décrémenter ] JusquÀ var = val_max Faire
2      séquence
3  FinPour
4
5  -- Une variante
6  Pour Chaque var Dans val_min..val_max [ Renversé ] Faire
7      séquence
8  FinPour
```

Règle :

- `var` : variable d'un type scalaire, dite *variable de contrôle*
- expressions `val_min` et `val_max` compatibles avec le type de `var`
- La séquence d'instructions ne doit pas modifier la valeur de `var`

Évaluation : Les expressions `val_min` et `val_max` sont évaluées. `var` prend alors successivement chacune des valeurs de l'intervalle `[val_min..val_max]` dans l'ordre indiqué et pour chaque valeur, la séquence est exécutée.

Répétition Pour (2)

Remarques :

- Cette structure est utilisée lorsqu'on connaît à l'avance le nombre d'itérations à faire.
- Les expressions `val_min` et `val_max` ne sont évaluées qu'une seule fois.
- La séquence peut ne pas être exécutée (`val_min > val_max`).

Attention : Il est interdit de modifier la valeur de la variable de contrôle `var` dans la boucle.

Exercice 19 : Somme des premiers entiers

Calculer la somme des n premiers entiers.

Exercice 20 : Alphabet

Écrire les lettres de l'alphabet.

Exercice 21 : Pour et TantQue

Réécrire la boucle **`Pour`** en utilisant seulement le **`TantQue`**.

Solutions

Exercice 19

```
1  Comment « somme des n premiers entiers » ?
2              n: in Entier -- positif
3              somme: out Entier
4      somme <- 0
5      Pour i <- 1 JusquÀ i = n Faire
6          { Variant : n - i + 1 }
7          { Invariant : Résultat = ∑_{j=1}^{i} j }
8          somme <- somme + i
9      FinTQ
```

Exercice 20

```
1  Comment « Afficher les lettres de l'alphabet » ?
2      Pour lettre <- 'a' JusquÀ lettre = 'z' Faire
3          Écrire(lettre)
4      FinPour
```

Quelle structure de contrôle choisir ?

Théorème

Tout algorithme (calculable) peut être exprimé à l'aide de l'affectation et des trois structures **`Si Alors FinSi`**, **`TantQue`** et l'enchaînement séquentiel.
... Mais ce n'est pas forcément pratique, aussi des structures supplémentaires ont été introduites. Il s'agit donc d'utiliser la bonne !

Quelle conditionnelle choisir ?

- Préférer le **`Selon`** s'il y a au moins 2 cas car il est plus lisible et plus facile à étendre.
- Toujours mettre des **`SinonSi`** et des **`Sinon`** (s'ils sont possibles).

Quelle répétition choisir ?

Si on connaît a priori le nombre d'itérations, on utilise un **`Pour`**.
Dans les autres cas, on a le choix entre **`TantQue`** et **`Répéter`**.

- En général, utiliser un **`Répéter`** implique de dupliquer une condition
  et utiliser un **`TantQue`** implique de dupliquer une instruction.
- Le **`Répéter`** est « dangereux » car la séquence est exécutée au moins une fois. Attention au cas limite !
  $\Longrightarrow$ Si on répète au moins une fois alors on peut prendre un **`Répéter`**.
- Dans le doute, préférer un **`TantQue`**
  et utiliser l'exercice 18 pour vérifier si c'est élégant

Remarque : Pour aider au choix, il est judicieux de se demander :

- Qu'est ce qui est répété (quelle est la séquence) ?
- Quand est-ce qu'on arrête (ou continue) ?

Sommaire

- Motivation
- Types énumérés
- Types tableaux
- Types enregistrement

Motivation

- **Constat :** Les types élémentaires ne suffisent pas pour représenter toutes les données :
  - Comment représenter les couleurs d'un feu tricolore de circulation ?
  - Comment conserver les valeurs mesurées par un capteur ?
  - Comment représenter un nombre complexe, une date, un rendez-vous, etc.
- **Moyen :** Permettre au programmeur de définir ses propres types :
  1. **type énuméré**
  2. **type tableau**
  3. **type enregistrement** (classe dans un langage objet)

Motivation

Exercice 22 : Définir un type Mois

Comment définir un type mois pour représenter les mois de l'année ?

Il existe une convention largement utilisée qui consiste à prendre 1 pour janvier, 2 pour février, ..., et 12 pour décembre.

On peut donc définir ainsi le type Mois :

```
1  Type
2      T_Mois = 1..12      -- 1 pour Janvier, ..., 12 pour décembre.
```

Remarque : Il est important de préciser la convention même si elle est classique !

1..12 est un type intervalle qui correspond aux entiers de 1 à 12.

Motivation (2)

Exercice 23 : Définir un type Jour

Comment définir un type Jour pour représenter les sept jours de la semaine ?

Comme pour l'exercice précédent, on peut utiliser un sous-type intervalle des entiers.

Cependant, il n'y a pas consensus :

- Faut-il prendre l'intervalle `1..7` ou `0..6` ?
- Que signifie la première valeur ? Lundi ? Dimanche ? Un autre jour ?

Dans ce cas, il est préférable de nommer chaque valeur possible du type pour en préciser la signification.

C'est ce que permet le type énuméré :

```
1  Type
2    Jour = ( LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI,
3           SAMEDI, DIMANCHE )
```

Définition

- **Définition :** Un *type énuméré* consiste à définir un type en énumérant les valeurs possibles.
  Chaque valeur est désignée par un nom symbolique (identificateur).
- **Exemple :**

```
1  Type
2    Mois = ( JANVIER, FÉVRIER, MARS, AVRIL, MAI, JUIN, JUILLET,
3               AOÛT, SEPTEMBRE, OCTOBRE, NOVEMBRE, DÉCEMBRE )
4    Jour = ( LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI,
5               SAMEDI, DIMANCHE )
```

- Un type énuméré devrait définir au moins deux valeurs !
- Les valeurs d'un type énuméré sont équivalentes à des constantes dont la valeur réelle est attribuée par le compilateur.
- **Notation :** Le nom des valeurs d'un type énuméré s'écrivent en majuscules.

Intérêt

- **Intérêt :** Avoir des programmes plus lisibles et évolutifs car les constantes littérales sont remplacées par des constantes symboliques.
- **Exemple :** Que signifie la valeur 5 dans un programme ?
  - le mois de mai ?
  - le vendredi ? ou le jeudi ?
  - la couleur rouge ?
  - une note (sur 20) ?

  Que faire si vous devez changer la note éliminatoire 5 par 3 ? Il faut faire attention à ne pas changer le mois, ou le jour de la semaine ou la couleur...
- **Attention :** Les types énumérés sont une convention définie par le programmeur qui n'a rien à voir avec la communication avec l'utilisateur du programme (interface de saisie et interface d'affichage).
  $\Longrightarrow$ Ne pas confondre représentation et présentation !
- **Idéalement,** des types énumérés différents devraient définir des types non compatibles pour éviter les mélanges (ORANGE, KAKI : fruit ? couleur ?).

Opérations

- **affectation** : <-

```
1  Variable
2      jour: Jour
3  Début
4      jour <- MERCREDI
```

- **opérateurs de comparaison** usuels : = et <> mais aussi <, <=, > et >=.
  On a : `e1 < e2` ssi e1 appraraît avant e2 dans la définition du type.

```
1  LUNDI < MARDI < ... < DIMANCHE
```

- `succ(expr)` (et `succ(expr)`) : **valeur suivante (précédente)** de expr

```
1  succ(LUNDI) = MARDI
2  pred(MARDI) = LUNDI
```

- `ord(expr)` : le **numéro d'ordre** (Entier) de expr dans la définition du type

```
1  numéro <- ord(LUNDI)     -- numéro vaut 0
2  ord(MARDI) - ord(LUNDI) vaut 1
```

- **Type**`(expr)` : la valeur du type énuméré de numéro d'ordre expr (Entier)

```
1  jour <- Jour(0)          -- jour est donc LUNDI
```

Un type énuméré est un type scalaire

- **Remarque :** Un type énuméré est un type scalaire (comme les entiers, les caractères, les booléens).
- **Conséquence :** un type énuméré peut être utilisé partout où type scalaire peut l'être :
  - L'expression d'un **Selon** :

```
1  Selon jour Dans
2      LUNDI:
3          Ecrire("Lundi")
4      ...
5  FinSelon
```

  - La variable de contrôle d'une boucle **Pour** peut être de type énuméré.
    Exemple :

```
1  Pour Chaque jour Dans Jour Faire
2      ...
3  FinPour
```

Les tableaux comme une facilité d'écriture

Exercice 24 : Occurrences des chiffres d'un entier
Écrire un programme qui compte le nombre d'occurrences des 10 chiffres dans un entier naturel donné.
Exemple : l'entier 4214 a une occurrence du chiffre 1, une de 2 et deux de 4.

Que penser du programme suivant ?

D'autres solutions ?

```
1   Algorithme compter_occurrences
2       -- Compter le nombre d'occurrences des chiffres d'un nombre
3   Variable
4       nombre: Entier      -- entier saisi par l'utilisateur
5       nb0: Entier         -- nb de  0 dans nombre
6       nb1: Entier         -- '' de  1   ''    ''
7       nb2: Entier         -- '' de  2   ''    ''
8       nb3: Entier         -- '' de  3   ''    ''
9       nb4: Entier         -- '' de  4   ''    ''
10      nb5: Entier         -- '' de  5   ''    ''
11      nb6: Entier         -- '' de  6   ''    ''
12      nb7: Entier         -- '' de  7   ''    ''
13      nb8: Entier         -- '' de  8   ''    ''
14      nb9: Entier         -- '' de  9   ''    ''
15      unité: Entier       -- unité extraite de nombre
16  Début
17      -- saisir le nombre
18
19      -- initialiser les compteurs
20      nb0 <- 0
21      nb1 <- 0
22      nb2 <- 0
23      nb3 <- 0
24      nb4 <- 0
25      nb5 <- 0
26      nb6 <- 0
27      nb7 <- 0
28      nb8 <- 0
29      nb9 <- 0
30
31      -- comptabiliser chaque chiffre de nombre
32      Répéter
33          -- comptabiliser l'unité de nombre
34          unité <- nombre Mod 10
35          Selon unité Dans
```

```
36                0: nb0 <- nb0 + 1
37                1: nb1 <- nb1 + 1
38                2: nb2 <- nb2 + 1
39                3: nb3 <- nb3 + 1
40                4: nb4 <- nb4 + 1
41                5: nb5 <- nb5 + 1
42                6: nb6 <- nb6 + 1
43                7: nb7 <- nb7 + 1
44                8: nb8 <- nb8 + 1
45                9: nb9 <- nb9 + 1
46            FinSelon
47
48            -- supprimer l'unité de nombre
49            nombre <- nombre Div 10
50        JusquÀ nombre = 0
51
52        -- afficher les occurrences
53        ÉcrireLn("nb de 0 ", nb0)
54        ÉcrireLn("nb de 1 ", nb1)
55        ÉcrireLn("nb de 2 ", nb2)
56        ÉcrireLn("nb de 3 ", nb3)
57        ÉcrireLn("nb de 4 ", nb4)
58        ÉcrireLn("nb de 5 ", nb5)
59        ÉcrireLn("nb de 6 ", nb6)
60        ÉcrireLn("nb de 7 ", nb7)
61        ÉcrireLn("nb de 8 ", nb8)
62        ÉcrireLn("nb de 9 ", nb9)
63    Fin
```

Les tableaux comme une nécessité

Exercice 25 : Trier une série de valeurs
Afficher dans l'ordre croissant une série de valeurs réelles saisies au clavier. La série se termine par la valeur zéro qui ne fait pas partie de la série.
Voici quelques exemples :

```
1 2 3 0          --> 1 2 3
3 2 1 0          --> 1 2 3
5 2 3 8 -1 0     --> -1 2 3 5 8
```

Solution : Les premières étapes du développement donnent :

```
R0 : Afficher une série dans l'ordre croissant

R1 : Raffinage De « Afficher une série dans l'ordre croissant »
   | Saisir la série                    valeurs: out
   | Trier les valeurs de la série     valeurs: in out
   | Afficher les valeurs               valeurs: in
```

Quel type prendre pour valeurs ? Comment représenter plusieurs réels ?
Remarque : Ces trois étapes constituent des problèmes récurrents.

Définitions

Définition : Un tableau est un *type de données* qui permet de regrouper un *nombre fini* d'éléments ayant *tous le même type*.
Conséquence : Un tableau est donc caractérisé par :

- le type des éléments qu'il peut contenir ;
- le nombre d'éléments qu'il peut contenir. On parle de *capacité*.
- le moyen d'accéder à un élément. Ce sont les *indices* ou *index*.

Exemple : Déclaration possible pour les `valeurs` de l'exercice précédent :

```
1  Variable
2      nb: Tableau [10] De Entier           -- nb[i] = nombre d'occurrences de
3      valeurs: Tableau [CAPACITÉ] De Réel  -- les valeurs de la série
```

Tableaux à une dimension

Définition : Un tableau à une dimension est un tableau pour lequel on accède aux éléments par un unique indice.
Notation : La déclaration d'un type tableau est la suivante

```
1    Tableau [capacité] De TypeÉlément
```

- `TypeÉlément` : type des éléments du tableau (tout type, tableau compris).
- `capacité` : nombre maximal d'éléments du tableau

Exemples :

```
1  Tableau [40] De Entier
2  Tableau [7] De Tableau [24] De Réel
```

Propriétés :

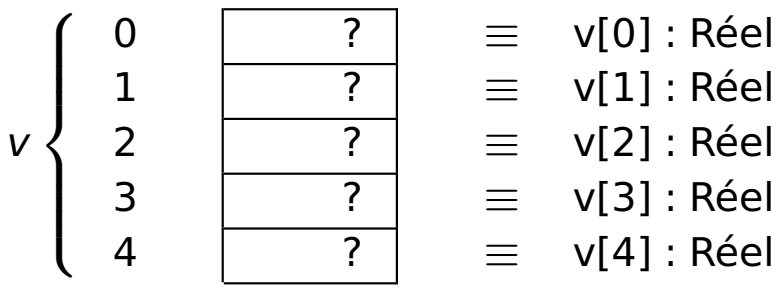
- Certains langages imposent que la capacité soit une constante du programme
- La capacité d'un tableau ne peut pas être changée durant sa durée de vie

Déclaration d'une variable

Une variable tableau se déclare comme toute variable :

```
1  Variable
2      v: Vecteur
```

Représentation en mémoire Déclarer une variable de type Tableau revient à réserver en mémoire de quoi stocker capacité fois une variable du type des éléments du tableau.



Remarque : La zone mémoire allouée est contiguë.
Remarque : La zone mémoire allouée n'est pas initialisée, les valeurs sont donc indéterminées.

Accès à un élément

- **Principe :** Manipuler individuellement un élément du tableau.

```
1       tab[expression]     -- l'élément à l'indice expression de tab
```

```
1       v[1]          -- élément du tableau v à l'indice 1
```

- `tab[expr]` est strictement équivalent à une variable de type `TypeÉlément`.
- **Contraintes :**
  1. `tab` doit être une variable de type tableau (compilateur) ;
  2. `expr` doit être un entier compris dans [0, CAPACITÉ[
     sinon c'est une erreur de programmation mais :
     - elle ne sera pas toujours détectée par le compilateur :
       exemple : -1 est un entier !
     - elle devrait provoquer une erreur à l'exécution (dépend du langage !)

Affectation

Principe : Initialiser un tableau à partir d'un autre.

```
1  Variable
2      v1, v2: Vecteur  -- Remarque : fonctionne pour tout type tableau !
3  Début
4      ...
5      v1 <- v2    -- Copier des éléments de v2 dans v1
6  Fin
```

Remarque : Ceci est équivalent à copier tous les éléments du tableau

```
1  Pour i <- 1 JusquÀ i = DIM Faire
2      v1[i] <- v2[i]
3  FinPour
```

Attention : On parle de **sémantique de la valeur** : le tableau désigne l'ensemble des cases (des valeurs).
Ceci n'est pas vrai dans tous les langages.

Utilisation d'un tableau

- Il faut pouvoir dimensionner le tableau : définir sa capacité.
- **Attention :** La capacité ne peut pas être modifiée pendant la durée de vie du tableau.
- **Deux utilisations possibles du tableau :**
  1. Toutes les cases du tableau sont utilisées.
     Exemple : compter le nombre d'occurrences des chiffres d'un entier.
  2. Le tableau a été surdimensionné et seule une partie est utilisée. Il faut alors définir et gérer la **taille effective** du tableau, nombre d'éléments effectivement stockés dans le tableau (variable supplémentaire).
     Exemple : stocker une série d'entiers naturels saisie au clavier.

```
1  tab: Tableau [CAPACITÉ] De Type   -- le tableau
2  taille_effective: Entier          -- nombre de cases utilisées
```

Invariant : `0 <= taille_effective <= CAPACITÉ`.
Contraintes :

- En accès : `tab[indice]` est valide ssi `0 <= indice < taille_effective`.
- En modification : tab[indice] est généralement valide si `0 <= indice <= taille_effective`
  (on utilise toujours le début du tableau, « sans trou »)

Exemple de tableau sans taille effective

```
1  Algorithme compter_occurrences
2      -- Compter le nombre d'occurrences des chiffres d'un nombre
3  Variable
4      nombre: Entier                -- entier saisi par l'utilisateur
5      nb: Tableau [10] De Entier    -- nb[i] = nb de i dans nombre
6      unité: Entier                 -- unité extraite de nombre
7      i: Entier                     -- variable de boucle
8  Début
9      -- Saisir le nombre
10     ...
11
12     -- Initialiser les compteurs à 0
13     Pour i <- 0 JusquÀ i = 9 Faire
14         nb[i] <- 0
15     FinPour
16
17     -- Comptabiliser les occurrences
18     Répéter
19         unité <- nombre Mod 10          -- Déterminer l'unité
20         nb[unité] <- nb[unité] + 1      -- Comptabiliser l'unité de nombre
21         nombre <- nombre Div 10         -- Supprimer l'unité de nombre
22     JusquÀ nombre = 0
23
24     -- Afficher les compteurs
25     Pour i <- 0 JusquÀ i = 9 Faire
26         ÉcrireLn("Nombre␣de␣", i, "␣:␣", nb[i])
27     FinPour
28 Fin
```

Exemple de tableau avec taille effective

```
1   Algorithme aficher_serie
2       -- saisir une série, puis l'afficher
3   Constante
4       MAX = 10    -- nb max de valeurs dans une série
5   Variable
6       valeurs: Tableau [MAX] De Réel      -- de la série
7       nb: Entier          -- nombre de valeurs dans la série
8       x: Réel             -- entier lu au clavier
9       i: Entier           -- variable de boucle
10  Début
11      -- Saisir la série
12      nb <- 0             -- pas encore de valeurs lues
13      Lire(x)
14      TantQue (x <> 0)            -- valeur de la série
15              Et (nb < MAX)       -- encore de la place dans le tableau
16      Faire
17          valeurs[nb] <- x        -- stocker la valeur
18          nb <- nb + 1            -- une valeur de plus
19          Lire(x)                 -- saisir un nouvel entier
20      FinPour
21      { (x = 0) Ou (nb >= MAX) }  -- SIGNIFICATION ?
22
23      -- Afficher les valeurs
24      Pour i <- 0 JusquÀ i = nb - 1 Faire
25         Écrire(valeurs[i])
26      FinPour
27  Fin
```

Tableaux à plusieurs dimensions

Définition : Un tableau est dit à plusieurs dimensions s'il est nécessaire de préciser plusieurs indices pour accéder à un élément.

Principe : Pour accéder à un élément, il faut alors donner une valeur pour chacun des indices.

Exemple :

```
1  Constante
2      NB_LIGNES = 5
3      NB_COLONNES = 10
4  Type
5      Matrice = Tableau [NB_LIGNES, NB_COLONNES] De Réel
6  Variable
7      m1: Matrice
8  Début
9      m1[0, 0] <- 1
10     m1[4, 9] <- 50
11 Fin
```

Remarque : Généralement, les éléments sont contigus en mémoire (comme un tableau à une dimension) et l'accès à l'élément [i, j] se fait avec :

```
1          i * NB_LIGNES + j
```

tableau à plusieurs dimensions et tableaux de tableaux

Remarque : Les tableaux à plusieurs dimensions sont isomorphes à des tableaux de tableaux.
Exemple :

```
1  Type
2      Vecteur = Tableau [NB_COLONNES] De Réel
3      MatriceBis = Tableau [NB_LIGNES] De Vecteur
4  Variable
5      m2: MatriceBis
6  Début
7      m2[1]       -- c'est un vecteur
8      m2[1][2]    -- deuxième élément du premier vecteur (m1[1, 2])
9  Fin
```

Motivation

Exercice 26 : Définir des dates

Comment représenter une date ? Et une deuxième date ?

Solution : On peut représenter les dates sous la forme de trois entiers représentant le jour, le mois et l'année.

```
1  Variable
2      j1, m1, a1: Entier  -- la première date
3      j2, m2, a2: Entier  -- la deuxième date
```

Remarque : On peut prendre un type plus précis pour le jour et le mois !

Quels sont les défauts de la solution proposée ?

Solution : La date est représentée par trois variables indépendantes. Il faut donc faire attention à ne pas les mélanger, etc.

Définition

Définition : Un enregistrement est le produit cartésien de plusieurs types T_1, ..., T_n.
À chaque composante de type T_i est associé un identificateur choisi par le programmeur. Il permet de sélectionner la composante.
Le couple (identificateur, Type) est appelé **champ** ou **attribut** de l'enregistrement.
Notation :

```
1  Types
2      T = -- Le nom du type (qui doit être significatif !)
3          Enregistrement  -- sa définition
4              nom_champ_1 : T_1      -- un champ, son nom,  son type et son rôle
5              ...
6              nom_champ_n : T_n      -- le dernier champ, son nom,  son type, son rôle
7          FinEnregistrement
```

Exemple :

```
1  Date =
2      Enregistrement
3          jour: 1..31              -- Le numéro du jour
4          mois: 1..12              -- le numéro du mois : 1 janvier...
5          année: Entier            -- année > 0
6      FinEnregistrement
```

Intérêt

- Regrouper les différentes variables nécessaires à la définition d'une donnée (le jour, le mois et l'année de la date).
- Pouvoir définir des types structurés (ou complexes) qui ne peuvent pas être représentés par les types élémentaires.

Voici quelques exemples d'enregistrements :

- Un **complexe** est défini par une partie réelle et une partie imaginaire.
- Une **date** est composée d'un numéro de jour (1..31), d'un numéro de mois (1..12) et d'une année (strictement positive).
- Un **stage** peut être défini comme un intitulé (une chaîne de caractères), une date de début et une date de fin (deux dates), un nombre de places (entier).
- Une **fiche bibliographique** est définie par le titre du livre (Chaîne), les auteurs (noms et prénoms), la date de parution, l'éditeur, le numéro ISBN (Chaîne), etc.

Déclaration d'une variable

```
1  Variable
2      d1: Date     -- d1 est une variable de type Date.
3                   -- Elle occupe 3 entiers en mémoire.
```

Une valeur du type enregistrement T est un n-uplet $(v_1, v_2, ..., v_n)$ où chaque valeur v_i est du type T_i.

Représentation en mémoire La place occupée en mémoire est la somme des places nécessaires pour représenter chacun des champs de types T_i.

```
1      Place(T) = Place(T1) + ... + Place(Tn)
```

d1. { jour | ? | ≡ d1.jour : 1..31
mois | ? | ≡ d1.mois : 1..12
année | ? | ≡ d1.année : Entier

Attention : Les T_i sont n'importe quel type (y compris enregistrement).
La seule contrainte c'est qu'un type enregistrement T ne peut pas avoir un champ (direct ou indirect) de type T.

Opérations

Principe : Manipuler individuellement un élément de l'enregistrement.

```
1       var.id_champ
```

Exemple :

```
1       d1.jour
```

`var.id_champ` est strictement équivalent à une variable de type `TypeÉlément`.

Contraintes : Deux contraintes à respecter (vérifiées par le compilateur).

1. `var` doit être une variable de type enregistrement ;
2. `id_champ` doit être champ du type de `var`.

Affectation

Principe : Initialiser un enregistrement à partir d'un autre.

```
1  Variable
2      v1, v2: TypeEnregistrement
3  Début
4      ...
5      v1 <- v2    -- Copier tous les champs de v2 dans v1
6  Fin
```

Remarque : Pour une date, `d1 <- d2` est équivalent à :

```
1  d1.jour <- d2.jour
2  d1.mois <- d2.mois
3  d1.année <- d2.année
```

Attention : Il s'agit d'une **sémantique de la valeur**. Ce n'est pas le cas de tous les langages.

Exemple

```
1  Variables
2      d1, d2: Date          -- deux dates
3      année: Entier;
4  Début
5      -- initialiser la date d1
6      d1.jour <- 21
7      d1.mois <- 10
8      d1.année <- 2003
9
10     -- initialiser la date d2 à partir de d1
11     d2 <- d1
12
13     -- afficher la date d2
14     Écrire(d2.jour)
15     Écrire('/', d2.mois)
16     Écrire('/', d2.année)
17
18     -- conserver l'année de d2
19     année <- d2.année
20
21     -- saisir l'année de d2
22     Lire(d2.année)
23 Fin
```

Introduction

Motivation : Un sous-programme permet au programmeur d'étendre le langage de programmation en définissant :

- ses propres instructions
- et ses propres opérateurs

qu'il pourra ensuite (ré)utiliser. Il s'agit de regrouper plusieurs instructions sous un nom commun. Elles seront exécutées de manière atomique.

Définition : Un sous-programme est un regroupement d'instructions constituant un « programme » autonome (indépendant de son contexte), réutilisable dans plusieurs contextes.

Le terme « sous-programme » est un terme générique qui englobe à la fois :

- les **procédures** : instructions définies par le programmeur ;
- et les **fonctions** : opérateurs définis par le programmeur.

Intérêt des sous-programmes

- **Structuration de l'algorithme :** programme découpé en « morceaux ».
  - Un SP doit être court (ne pas dépasser une page).
  - Souvent quelques lignes.
- **Compréhensibilité de l'algorithme :**
  - chaque sous-programme est individuellement plus facile à comprendre
  - pour comprendre l'ensemble, la spécification des SP suffit
  - les SP sont la trace dans le code des raffinages
- **Factorisation :** un appel d'un SP exécute son code (pas de copier/coller)
- **Mise au point plus facile :** le programme est testé SP par SP
  $\Longrightarrow$ les erreurs sont détectées plus tôt, plus faciles à localiser, à corriger
- **Amélioration de la maintenance**
  - car le programme est plus facile à comprendre ;
  - car un changement peut rester localisé dans quelques SP.
- **Réutilisation** dans le programme (plusieurs appels au SP) et dans d'autres (SP autonomes, voir modules)

Communication entre programme et sous-programmes

En fait, il s'agit de la communication entre :

- le *programme appelant* (programme ou sous-programme) et
- le *programme appelé* (forcément un sous-programme).

Deux techniques sont possibles :

1. l'utilisation de *variables globales* : c'est la **mauvaise** !
2. l'utilisation de *paramètres formels* : c'est la **bonne** !

Communication par variables globales

```
1   Variable        -- variables pour communication avec calculer_max
2       a, b: Entier
3       max: Entier         -- le plus grand de a et b
4
5   Procédure calculer_max Est
6       Début
7           Si a > b Alors
8               max <- a
9           Sinon
10              max <- b
11          FinSi
12      Fin
13
14  Variable        -- variables du programme principal
15      i, j: Entier
16  Début
17      Lire(i)
18      a <- 2*i            -- initialiser les variables globales
19      b <- 20
20      calculer_max        -- appeler le sous-programme
21      Écrire(max)         -- exploiter le résultat
22  Fin
```

C'est la **mauvaise façon** car elle rend difficile :

- la compréhension de la communication (plusieurs lignes) ;
- la réutilisation du sous-programme (non autonome) ;
- la compréhension du programme : il faut lire tout le code pour savoir quels sous-programmes utilisent quelles variables ;
- la maintenance : comment évaluer l'impact d'une modification ?

Conclusion : Ne jamais utiliser les variables globales sauf si vous avez de très bonnes raisons !

Principe : Un sous-programme, pour être réutilisable, doit être considéré comme une boîte noire : appelé avec les mêmes paramètres effectifs, il doit **toujours** donner les mêmes résultats.
$\Longrightarrow$ La communication doit se faire exclusivement au travers de ses paramètres.
Les variables globales ne permettent pas de respecter ce principe. Ce sont des « paramètres » non explicites, difficiles à contrôler.

Communication par paramètres formels

```
1   Procédure calculer_max(a, b: in Entier; max: out Entier) Est
2           -- Calculer max le plus grand des deux entiers a et b
3           --
4           -- Assure
5           --      (max >= a) Et (max >= b)        -- plus grand que a et b
6           --      (max = a) Ou (max = b)          -- l'un des deux
7       Début
8           Si a > b Alors
9               max <- a
10          Sinon
11              max <- b
12          FinSi
13      Fin
14
15  Procédure application Est
16      Variable
17          i: Entier
18          r: Entier
19      Début
20          Lire(i)
21          calculer_max(2*i, 20, r)
22          Ecrire(r)
23      Fin
```

Paramètres formels et effectifs

Définitions : Un paramètre est le moyen de communication entre programme appelé et programme appelant.
Un paramètre formel apparaît dans la signature du sous-programme.

```
1  Procédure calculer_max(a, b: in Entier; max: out Entier) Est
```

a, b et max sont les paramètres, la matière première du sous-programme :

« Étant donné a et b deux entiers, calculer max, le plus grand des deux. »

Un paramètre effectif, fourni par le programme appelant lors de l'appel d'un sous-programme, sert à initialiser le paramètre formel correspondant.

```
1  calculer_max(2*i, 20, r)
```

`2*i`, `20` et `r` sont les paramètres effectifs correspondant à `a`, `b` et `max`.

Remarque : Un paramètre formel est équivalent à une variable locale initialisée automatiquement à partir du paramètre effectif correspondant.

Image : Un SP doit être une boîte noire : on doit pouvoir l'utiliser sans avoir à regarder son implantation !

Définition d'un paramètre formel

Un paramètre formel est caractérisé par :

- un nom (qui doit être significatif par rapport au rôle du paramètre) ;
- un type ;
- un mode de passage (**in**, **out** ou **in out**).

Syntaxe : La syntaxe est la suivante

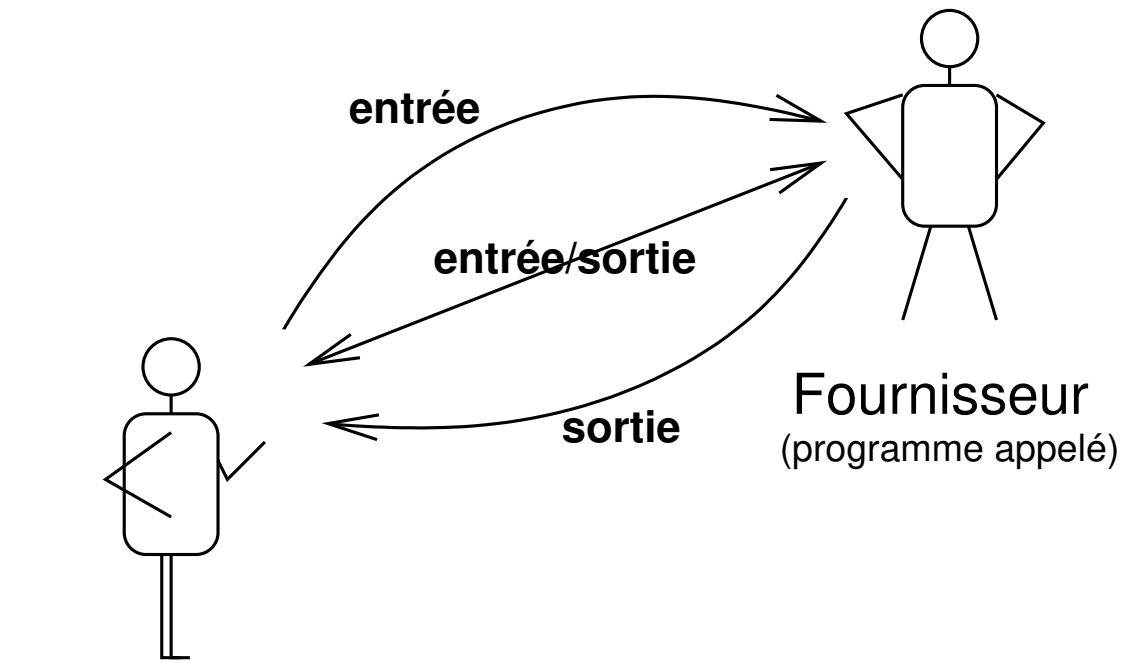
```
1  nom: mode Type
2  nom1, nom2: mode Type -- plusieurs paramètres de même type et même mode
```

Le mode de passage spécifie la direction des informations échangées entre programme appelant et programme appelé.
Le mode de passage impose des contraintes sur :

- l'utilisation des paramètres dans le sous-programme appelé ;
- les paramètres effectifs donnés par le sous-programme appelant.

Mode de passage des paramètres

Le mode de passage de paramètre définit dans quelles directions les données sont échangées entre le programme appelant et le programme appelé.
Le point de vue pris est celui du programme appelé.



Pour caractériser une donnée,
on se place du point de vue du fournisseur

Paramètre en **in**

Motivation : La donnée est seulement utilisée par le sous-programme.

Notation : On utilise le mot-clé **in**.

```
1       nom: in Type
```

Contrainte sur le paramètre effectif : Le paramètre effectif est toute *expression* dont le type est compatible avec celui du paramètre (Type).

Initialisation : La valeur du paramètre formel est la valeur du paramètre effectif.

Contraintes pour le SP : Le SP ne peut qu'accéder à la valeur du paramètre effectif et ne peut en aucun cas la modifier.

Intérêt : L'appelant est sûr que la donnée transmise ne sera pas modifiée par le sous-programme.

Tolérance : Le SP peut modifier la valeur du paramètre formel, en fait la *copie* du paramètre effectif (sans modification des données du SP appelant).

Exemple : Afficher la couleur d'un feu tricolore

```
1   Type
2       CouleurFeu = (ROUGE, VERT, ORANGE)
3
4   Procédure afficher_feu(couleur: in CouleurFeu) Est
5           -- afficher la couleur du feu
6       Début
7           Selon couleur Faire
8               ROUGE:      Écrire("rouge")
9               VERT:       Écrire("vert")
10              ORANGE:     Écrire("orange")
11          FinSelon
12      Fin
13
14  Procédure tester_afficher_feu Est
15          -- tester l'affichage du feu
16      Variable
17          c: CouleurFeu
18      Début
19          afficher_feu(ROUGE)
20          afficher_feu(succ(ROUGE))
21          Pour Chaque c Dans CouleurFeu Faire
22              afficher_feu(c)
23          FinPour
24      Fin
```

Paramètre en **out**

Motivation : La donnée est élaborée par le sous-programme.

Notation : On utilise le mot-clé **out**.

```
1       nom: out Type
```

Contrainte sur le paramètre effectif : Le paramètre effectif est toute *variable* dont le type est compatible avec celui du paramètre (Type).

Initialisation : Le paramètre formel donne accès à la zone mémoire occupée par le paramètre effectif. Il « contient » donc son adresse.

Contraintes pour le SP : Le SP ne peut que changer la valeur du paramètre et ne peut en aucun cas y accéder.

Intérêt : L'appelant verra les modifications faites aux variables transmises.

Tolérance : Le SP peut accéder à la valeur du paramètre formel (seulement après une première affectation !).

Exemple : saisir la couleur d'un feu

```
1   Procédure saisir_feu(couleur: out CouleurFeu) Est
2           -- saisir au clavier la couleur du feu.
3           -- L'initiale de la couleur est demandée à l'utilisateur.
4       Variable
5           c: Caractère    -- caractère saisi par l'utilisateur
6           erreur: Booléen -- y a-t-il une erreur dans la saisie ?
7       Début
8           Répéter
9               -- saisir la couleur sous forme d'une lettre
10              Lire(c)
11
12              -- décoder la couleur
13              erreur <- FAUX
14              Selon c Dans
15                  'r', 'R':  couleur <- ROUGE
16                  'v', 'V':  couleur <- VERT
17                  'o', 'O':  couleur <- ORANGE
18              Sinon
19                  Ecrire("Erreur de saisie.  Donnez r, v ou o.")
20                  erreur <- VRAI
21              FinSelon
22          JusquÀ Non erreur
23      Fin
24
25  Procédure tester_saisir_feu Est
26          -- tester la procédure de saisie de la couleur du feu
27      Variable
28          c: CouleurFeu
29      Début
30          saisir_feu(c)
31          afficher_feu(c)
32          saisir_feu(ROUGE)   -- ERREUR : ROUGE n'est pas variable
33          saisir_feu(succ(ROUGE)) -- ERREUR : expr et non variable
34      Fin
```

Paramètre en **`in out`**

Motivation : La donnée est modifiée (utilisée puis changée) par le sous-programme.

Notation : On utilise le mot-clé **`in out`**.

```
1       nom: in out Type
```

Contrainte sur le paramètre effectif : Le paramètre effectif est toute *variable* dont le type est compatible avec celui du paramètre (Type).

Initialisation : Le paramètre formel donne accès à la zone mémoire occupée par le paramètre effectif. Il « contient » donc son adresse.

Contraintes pour le SP : Le SP peut utiliser librement le paramètre, en général il commence par y accéder puis il le modifie.

Intérêt : L'appelant transmet une donnée (valeur du paramètre effectif) et en verra les modifications.

Exemple : permuter la valeur de deux réels

```
1   Procédure permuter(r1, r2: in out Réel) Est
2           -- permuter la valeur des deux réels r1 et r2
3       Variable
4           tmp: Réel       -- pour stocker la valeur de r1
5       Début
6           tmp <- r1
7           r1 <- r2
8           r2 <- tmp
9       Fin
10
11  Procédure tester_permuter Est
12          -- tester le programme permutation
13      Variable
14          a, b: Réel
15      Début
16          a <- 10                 -- initialiser les variables a et b
17          b <- 20
18          permuter(a, b)          -- permuter les valeurs de a et b
19          Si a <> 20 Alors        -- vérifier a
20              Écrire("Erreur sur la valeur de la variable a");
21          FinSi
22          Si b <> 10 Alors        -- vérifier b
23              Écrire("Erreur sur la valeur de la variable b");
24          FinSi
25      Fin
```

Sous-programme et paramètres formels

Un sous-programme peut avoir un nombre quelconque de paramètres.
S'il n'a aucun paramètre, alors on ne met pas les parenthèses.

Attention : Le nombre de paramètres d'un SP est fixé par sa spécification.

```
1  Fonction max2(a, b: in Entier): Entier Est
2      -- le plus grand des deux entiers a et b...
3  Fonction max3(a, b, c: in Entier): Entier Est
4      -- le plus grand des trois entiers a, b et c...
```

Surcharge : Possibilité d'utiliser le même nom pour plusieurs SP.
Exemple : l'opérateur + est défini sur les entiers et les réels.

```
1  Fonction max(a, b: in Entier): Entier Est
2  Fonction max(a, b, c: in Entier): Entier Est
```

C'est le nombre et le type des paramètres effectifs qui permettent de choisir.
Attention : En l'absence de surcharge, il faut trouver des noms différents !

Appel à un sous-programme

Vérifications réalisées par le compilateur

Lors de l'appel d'un sous-programme, le compilateur vérifie :

- l'**existence du SP** (le nom doit correspondre au nom d'un SP dont la spécification est accessible) ;
- l'**utilisation du SP** (une procédure comme une instruction, une fonction comme une expression) ;
- le **nombre de paramètres effectifs** : il doit y avoir autant de paramètres effectifs que de paramètres réels ;
- le **type des paramètres effectifs** : le type du i^eparamètre effectif doit être compatible avec le type du i^eparamètre formel ;
- la **compatibilité** entre le paramètre effectif et le **mode de passage** de son paramètre formel.

Si toutes ces conditions sont remplies, l'appel est accepté sinon il est refusé !

Appel à un sous-programme

Exécution

Interprétation de l'appel d'un sous-programme :

- évaluer les paramètres effectifs ;
- conserver dans la pile d'exécution le numéro de l'instruction contenant l'appel au SP. Ceci constitue le début du bloc d'activation ;
- réserver de la place pour les paramètres formels et les initialiser :
  - avec une copie de la valeur du paramètre effectif si le mode est **`in`** ;
  - avec l'adresse du paramètre effectif si le mode est **`out`** ou **`in out`** ;
- réserver de la place pour les variables locales (indéterminées) ;
- mettre dans le compteur ordinal la première instruction du corps du SP ;

Interprétation de la fin d'un sous-programme :

- déterminer l'instruction suivante à partir du numéro conservé à l'appel ;
- libérer toute la mémoire occupée par le bloc d'activiation du SP ;

Programmation par contrat

Motivation : La programmation par contrat (Design By Contract, DBC) consiste à définir les responsabilités des sous-programmes au moyen de :

- **préconditions** : conditions sur les paramètres formels en **`in`** et **`in out`**.
  Propriétés qui doivent être vérifiées avant de pouvoir appeler le SP.
  $\Longrightarrow$ C'est au SP appelant de les vérifier.
- **postconditions** : conditions sur les paramètres en **`out`** et **`in out`** (et **`in`**).
  Propriétés qui doivent être vérifiées après l'exécution du SP.
  $\Longrightarrow$ À respecter par le SP appelé.
- **invariants** : propriétés associées à un type
  Propriétés qui doivent toujours être vraies sur les variables du type.
  $\Longrightarrow$ À respecter par tout SP (après initialisation).

Syntaxe d'une assertion (condition) = syntaxe du langage considéré avec :

- `\Old(expr)` dans une postcondition = la valeur de `expr` avant exécution du SP.

Exemple de spécifications

```
1   Procédure permuter(r1, r2: in out Entier) Est
2           -- permuter la valeur des deux entiers r1 et r2
3           --
4           -- Assure
5           --      r1 = \Old(r2)
6           --      r2 = \Old(r1)
7
8   Fonction racine_carrée(x: in Réel): Réel Est
9           -- la racine carrée du réel x
10          --
11          -- Nécessite
12          --      x >= 0
13          -- Assure
14          --      Résultat * Résultat = x
15          --      Résultat >= 0
16  Type
17      Fraction = Enregistrement
18              numerateur: Entier
19              dénominateur: Entier
20          -- Invariant : -- la fraction est normalisée
21          --    dénominateur > 0
22          --    (numerateur = 0) ==> (dénominateur == 1)
23          --    (numerateur <> 0) ==> pgcd(abs(numérateur), dénominateur) = 1
24          FinEnregistrement
```

Attention : Utiliser l'égalité sur des réels est une erreur !

Bénéfices et Difficuléts

Bénéfices :

- *définition des responsabilités :* chacun sait qui fait quoi ;
- *documentation :* les types et SP sont documentés formellement donc sans ambiguités ;
- *aide à la mise au point* (par instrumentation du code) :
  - vérification dynamique des assertions ;
  - détection des erreurs au plus tôt (près de leur origine) ;
- *code final optimisé* (non vérification des assertions).

Difficultés :

- difficile d'exprimer complétement les postconditions ;
- difficile d'exprimer dans le langage cible les quantificateurs existentiel (il existe...) et universel (quelque soit...).

Spécification d'une procédure

Les éléments de la spécification sont :

- la **signature** (ou **prototype**) : seule partie utilisée par le compilateur :
  - le **nom** de la procédure (significatif) ;
    prendre un verbe à l'infinitif.
  - les **paramètres formels** en précisant leur nom, type et mode ;
- la **sémantique** : pour les lecteurs humains (et outils spécifiques)
  - texte informel : objectif du SP et signification des paramètres ;
  - préconditions : conditions sur les paramètres en **`in`** et **`in out`** ;
  - postconditions : conditions sur les paramètres en **`out`** et **`in out`** (et **`in`**).

Ces rubriques sont nécessaires mais il est possible d'en ajouter d'autres :

- complexité de l'algorithme utilisé ;
- principe de l'algorithme utilisé...

Implémentation d'une procédure

La spécification décrit ce que fait la procédure (le QUOI).
L'implantation décrit COMMENT elle le fait.
La spécification décrit donc le problème à résoudre et l'implantation est l'algorithme pour le résoudre. Il est donc composé de :

- variables locales ;
- d'instructions.

Remarque : Pour construire l'implantation, on peut (on doit !) appliquer la méthode des raffinages.

Spécification d'une fonction

Les éléments de la spécification sont :

- la **signature** (ou **prototype**) : seule partie utilisée par le compilateur :
  - le **nom** de la fonction (significatif) ;
    prendre un nom ou adjectif qui caractérise le résultat de la fonction ;
  - les **paramètres formels** en précisant leur nom et leur type ;
    Contrainte : Il y a au moins un paramètre et ils sont tous en `in`.
  - le **type de retour** : type de l'information retournée.
- la **sémantique** : pour les lecteurs humains (et outils évolués)
  - texte informel : résultat de la fonction et signification des paramètres ;
  - préconditions : conditions sur les paramètres en entrée ;
  - postconditions : conditions sur le résultat.

Exception : Dans le cas d'un sous-programme d'initialisation, on fait une procédure (voir T. 132).

Implémentation d'une fonction

L'implantation d'une fonction décrit comment le résultat est obtenu.
Elle contient :

- une variable **prédéfinie `Résultat`** dont :
  - le type est le type de retour de la fonction et ;
  - la valeur à la fin de l'exécution de la fonction est la valeur de retour ;
- des variables locales (autre que **`Résultat`** !) ;
- des instructions.

Contrainte : Les instructions doivent nécessairement donner une valeur à **`Résultat`**.
Remarque : Pour construire l'implantation, on peut (on doit !) appliquer la méthode des raffinages.

Procédure ou fonction

Théorème : Toute fonction peut être écrite sous la forme d'une procédure.

Principe : Transformer le retour de la fonction en paramètre en **out**.

```
1  Fonction max(a, b : in Entier)
2                : Entier Est
3      -- le plus grand de a et b
4      --
5      --
6  Début
7      Si a > b Alors
8          Résultat <- a
9      Sinon
10         Résultat <- b
11     FinSi
12 Fin
```

```
1  Procédure calculer_max(
2          max: out Entier;
3          a, b: in Entier) Est
4      -- Calculer le plus grand
5      -- de a et b
6  Début
7      Si a > b Alors
8          max <- a
9      Sinon
10         max <- b
11     FinSi
12 Fin
```

Procédure ou fonction (2)

Avantage d'une fonction : Son utilisation est plus simple.

Exemple : max de 4 nombres n1, n2, n3 et 15.

```
1  grand <- max(max(n1, n2), max(n3, 15))
```

```
1  -- solution 1
2  calculer_max(grand1, n1, n2)
3  calculer_max(grand2, n3, 15)
4  calculer_max(grand, grand1, grand2)
```

```
1  -- solution 2
2  calculer_max(grand, n1, n2)
3  calculer_max(grand, grand, n3)
4  calculer_max(grand, grand, 15)
```

Règle : Définir une fonction plutôt qu'une procédure quand c'est possible.

Conseil : Éviter d'avoir des expressions avec trop d'appels à des fonctions.

Un SP peut être défini comme fonction si :

- il a un seul paramètre formel en sortie (**out**) ;
- il y a d'autres paramètres formels ;
- tous ses autres paramètres formels sont en entrée (**in**).

Exception : Un SP d'initialisation sera défini comme une procédure (ex : `initialiser_complexe(c: out Complexe: pr, pi: in Réel)`)

Comment concevoir un SP ?

1. **Définir la spécification du SP**
   1. Définir l'*objectif* du SP (équivalent du R0) ;
   2. Identifier les *paramètres formels* : rôle, mode puis nom.
   3. Choisir entre procédure et fonction.
   4. En déduire un *nom significatif* pour le SP.
   5. Identifier les *préconditions* et les *postconditions* sur ses paramètres ;
   6. Rédiger la spécification du SP à partir des informations ci-dessus.
2. **Écrire les programmes de test** (test unitaire)
3. **Définir l'implantation du SP :** Appliquer la méthode des raffinages avec la spécification du SP comme R0.
4. **Tester l'implantation du sous-programme**

Exercice 27 Écrire un SP qui calcule le pgcd de deux entiers.

Raffinages et sous-programmes

Chaque étape non élémentaire d'un raffinage est candidat à être un SP mais :

- ce n'est pas une condition nécessaire (quoique !) ;
- ce n'est pas une condition suffisante.

Critères pour faire un SP d'une étape non élémentaire d'un raffinage :

- le *sous-programme est réutilisable* : est-ce que le traitement est général et pourra être réutilisé dans l'algorithme ou d'autres algorithmes ?
- l'*amélioration de la lisibilité* : le SP représente quelques lignes de code qui, si elle étaient laissées dans le programme appelant, le rendrait difficile à lire (trop long, trop détaillé...) ;
- le *sous-programme correspond à un traitement bien défini*, relativement indépendant (pas trop spécifique) des autres étapes de l'algorithme.
  En particulier, le nombre de paramètres du SP doit être limité.

Remarque : Utiliser des SP c'est avoir un programme proche des raffinages.

Conseils sur la définition de SP

- Éviter de mélanger traitement (calcul) et interactions avec l'utilisateur du programme (affichage ou saisie) : Les traitements sont plus stables que l'IHM.
- Une fonction ne doit faire ni saisie, ni affichage.
- Un SP doit être une boîte noire $\Rightarrow$ ne pas dépendre de variables globales.
- Un SP ne doit pas avoir trop de paramètres
  - soit mauvais découpage,
  - soit regrouper les paramètres avec un type enregistrement.
- Un SP ne doit pas être trop long (sinon le découper en SP).
- Un SP ne doit pas avoir trop de structures de contrôle imbriquées (faire des SP).
- On doit être capable d'exprimer l'objectif du SP (commentaire)...
  sinon c'est qu'il est mal compris !
- Formaliser la spécification des SP avec des pré- et des postconditions.

Définition

Définition : Un sous-programme récusif est un sous-programme dont l'implantation contient un appel à lui-même.
Exemple : Fonction récursive qui calcule la factorielle :

```
1   Fonction factorielle(n: in Entier): Entier Est
2           -- Factorielle de n
3           --
4           -- Nécessite :
5           --      n >= 0
6       Début
7           Si n <= 1 Alors
8               Résultat <- 1
9           Sinon
10              Résultat <- n * factorielle(n-1)
11          FinSi
12      Fin
```

Remarque : `factorielle(n-1)` est l'appel récursif.

Fondement mathématique

Le corps de la fonction `factorielle` précédente correspond à la définition mathématique de la factorielle donnée sous forme de récurrence :

$$n! = \begin{cases} 1 & \text{si } n = 0 \text{ ou } n = 1 \\ n \times (n-1)! & \text{sinon} \end{cases}$$

Remarque : On pourrait mathématiquement utiliser la notation suivante pour définir la factorielle :

$$n! = \prod_{i=1}^{i=n} i = 1 \times 2 \times 3 \ldots \times (n-1) \times n$$

Elle conduirait à un sous-programme non récursif utilisant une boucle **`Pour`**.

Remarque : Les définitions par récurrence (et donc les SP récursifs) sont souvent plus concises et claires que leur équivalent itératif.

Terminaison

Danger : Il faut s'assurer que les appels récursifs s'arrêtent pour garantir la terminaison du programme.

Régle : Un SP récursif doit toujours faire apparaître deux éléments :

- le cas de base où l'on sait écrire le code sans nouvel appel récursif ;
- le cas général dans le quel on fait des appels récursifs.

Terminaison : Les appels récursifs doivent porter sur un problème de taille strictement inférieure.

Attention : Cette condition est suffisante mais pas nécessaire !

Exemple : Dans le cas de la factorielle :

- on définit la taille du problème de `factorielle(n)` comme étant n ;
- le cas terminal correspond à $n \leq 1$ ($n = 0$ ou 1) où la factorielle est 1 ;
- dans le cas général ($n > 1$), on utilise l'appel récursif `factorielle(n-1)` de taille strictement inférieure ($n - 1 < n$).

Récursivité terminale

Définition : Une fonction est *récursive terminale* quand le résultat de l'appel initial est directement celui du dernier appel récursif.
Régle : Aucune opération n'est réalisée sur le retour d'un appel récursif.

```
1  Fonction factorielle(n: in Entier; p: in Entier): Entier Est
2      Début
3          Si n <= 1 Alors
4              Résultat <- p
5          Sinon
6              Résultat <- factorielle(n-1, n * p)
7          FinSi
8      Fin
```

Le calcul de la factorielle de 4 donne :

```
1  fact(4) =  f(4, 1) -> f(3, 4) -> f(2, 12) -> f(1, 24)
2       <--24--    <--24--    <--24--     <--24--
```

Remarque : On a : fact(4) = f(4, 1) = f(3, 4) = f(2, 12) = f(1, 24) = 24

Intérêt : Inutile de remonter les appels récursifs (gain de temps).
Cette propriété peut être exploitée par certains compilateurs.

Remarque : Toute fonction récursive terminale peut être réécrite simplement en utilisant une itération.

```
1  Fonction factorielle(a: in Entier): Entier Est
2      Variable
3          n: Entier        -- équivalent de a mais modifiable !
4      Début
5          n <- a
6          Résultat <- 1
7          TantQue n > 1 Faire
8              Résultat <- n * Résultat
9              n <- n - 1
10         FinTQ
11     Fin
```

Remarque : On constate que :

- **Résultat** est équivalent à p ;
- initialisée à 1, valeur à fournir lors du premier appel à factorielle ;
- le **TantQue** correspond au cas général.

Récursivité mutuelle

Définition : On dit que deux SP sont mutuellement récursifs si chacun des deux appels l'autre (éventuellement indirectement).
Plus généralement, un ensemble de SP est mutuellement récursif si la relation « f appelle g » admet un cycle : f_1 appelle ...appelle f_n appelle f_1.

Exercice 28 La parité possède les propriétés suivantes :

- un nombre est pair s'il est nul ou si son prédecesseur est impair ;
- un nombre est impair s'il est non nul et si son prédecesseur est pair.

Écrire deux SP qui indiquent si un entier est pair ou impair.

```
1   Fonction pair(n: in Entier): Booléen
2           -- Est-ce que n est pair ?
3           -- Nécessite : n >= 0
4       Début
5           Si n = 0 Alors
6               Résultat <- VRAI
7           Sinon
8               Résultat <- impair(n - 1)
9           FinSi
10      Fin
11
12  Fonction impair(n: in Entier): Booléen
13          -- Est-ce que n est impair ?
14          -- Nécessite : n >= 0
15      Début
16          Si n = 0 Alors
17              Résultat <- FAUX
18          Sinon
19              Résultat <- pair(n - 1)
20          FinSi
21      Fin
```

Exercice 29 : Suite de Fibonacci

Les termes de la suite de Fibonacci sont définis par la relation de récurrence suivante :

$$fib(0) = 0$$
$$fib(1) = 1$$
$$fib(n) = fib(n-1) + fib(n-2) \; si \; n \geq 2$$

29.1 Écrire un sous-programme qui calcule la valeur du n^{e}élément de cette suite en utilisant la récursivité.

29.2 Indiquer tous les appels récursifs pour calculer la valeur du 4^{e}élément ($fib(4)$) de la suite.

29.3 Critiquer le sous-programme de la question 29.1.

Exercice 30 : Les tours de Hanoï

C'est Lucas, mathématicien français, qui sous le pseudonyme de Claus a inventé ce jeu.
Il se présente sous la forme d'un support en bois sur lequel sont plantés trois tiges A, B et C. Sur ces trois tiges peuvent être enfilés des disques de diamètres différents (8 dans la version originale, mais N de manière générale). Dans la configuration initiale (figure 1), les disques sont empilés par ordre de taille décroissante sur la tige A.

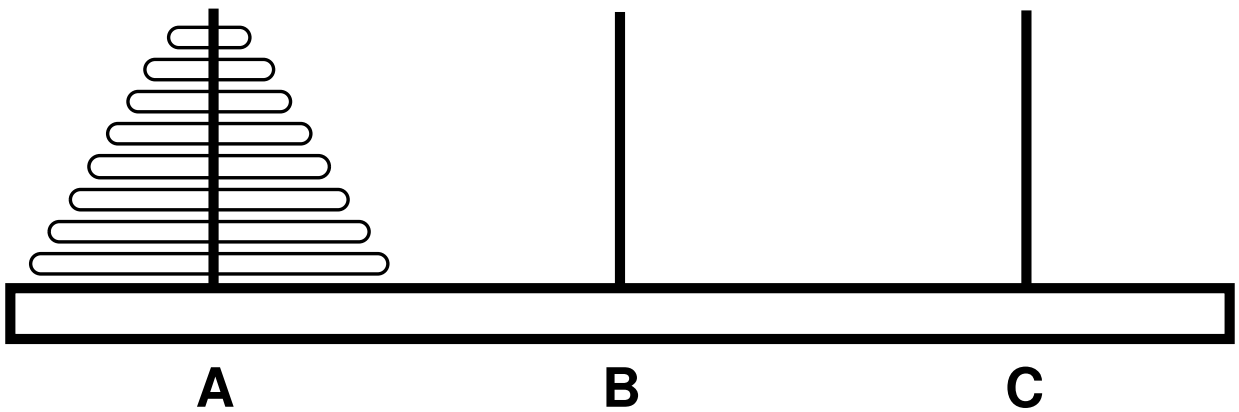


Figure – Configuration initiale du jeu des tours de Hanoï

Écrire un programme qui donne la solution de ce jeu (c'est-à-dire, la liste des coups à jouer).
Pour cela, on remarquera qu'un coup est complètement déterminé par la donnée de la tige de départ et de la tige d'arrivée, car on ne peut déplacer qu'un disque à la fois (celui qui se trouve en haut de la tige).

Motivation

Le but des modules est de pouvoir regrouper au sein d'une même unité syntaxique (le *module*) des informations (constantes, types, SP...) qui pourront être utilisées (et réutilisées) dans plusieurs programmes.
C'est la notion d'**encapsulation**.
Exemples : Voici quelques exemples de modules possibles :

- un module mathématique regroupant les opérations mathématiques telles que la racine carrée, la puissance, les fonctions trigonométriques, etc ;
- un module définissant un type Date et les opérations associées.

Synonymes : Paquetage, unité... classe (une classe est plus qu'un module).

Structure d'un module

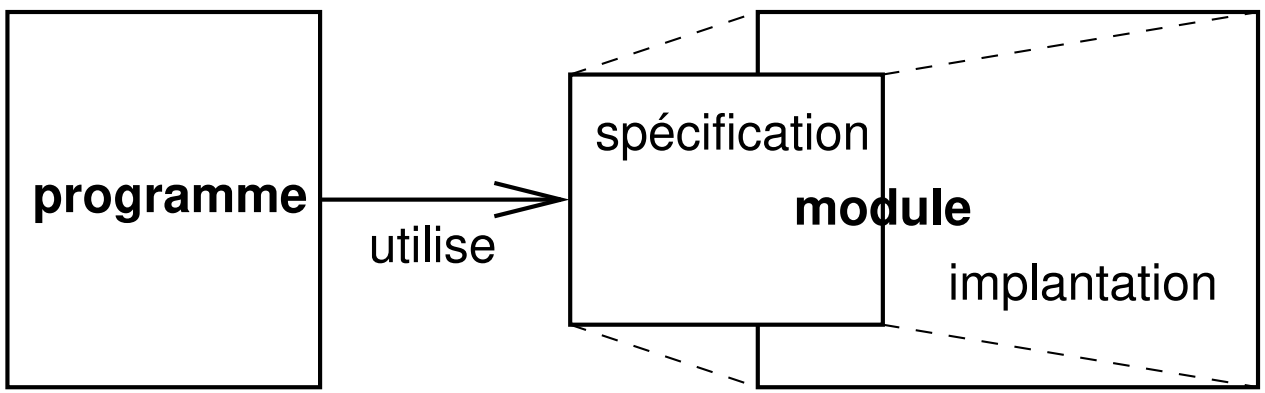
Un module est composé de deux parties :

- la **spécification** qui *déclare* les informations qui sont fournies par le module et donc utilisables par les autres modules et programmes :
  - des constantes ;
  - des types ;
  - des sous-programmes.
- une **implémentation** qui définit l'implémentation des sous-programmes.

Propriété : Seule la spécification est connue de l'utilisateur, l'implémentation lui est cachée (**masquage d'information**).
Intérêt : Critère de « *continuité* ».
Danger : Même si c'est possible, il est interdit de déclarer une variable dans la spécification d'un module car ce serait une variable globale !

Structure d'un module (suite)

Règle : L'implémentation d'un module doit au moins définir le corps des sous-programmes déclarés dans la spécification de ce module.
Remarque : L'implémentation contient généralement plus d'informations que la spécification... mais elles ne sont pas accessibles de l'extérieur.
Justification : Dans l'implémentation du module, pour définir le corps des SP, on peut définir de nouveaux types, constantes et SP mais ils seront locaux au module.



Exemple

Exercice 31 : Définir un module Date
Écrire un module qui définit le type Date et les opérations associées que nous limiterons à :

- initialiser une date à partir d'un jour, d'un mois et d'une année ;
- afficher une date sous la forme jj/mm/aaaa.

Remarque : Ce module est rudimentaire. Il pourrait être complété par d'autres opérations utiles sur les dates (comparaison, nom du jour dans la semaine, vérification de la validité d'une date, incrémentation, décrémentation...)

Intérêt : Si nous avons besoin de manipuler une date, il suffit d'utiliser le module et les opérations qu'il fournit.

Spécification du module Date

```
Module Date Spécification Est

Type
    Date =
        Enregistrement
            jour: 1..31          -- Le numéro du jour
            mois: 1..12          -- le numéro du mois
            année: Entier        -- année > 0
        FinEnregistrement

Procédure initialiser_date(une_date: out Date;
                           jj, mm, aaaa: in Entier)
    -- Initialiser une_date à partir de jj/mm/aaaa.
    --
    -- Nécessite :
    --          est_date_valide(jj, mm, aaaa)

Procédure afficher_date(une_date: in Date)
    -- Afficher une_date au format jj/mm/aaaa.

Fin
```

Remarque : La fonction `est_date_valide` devrait être définie !

Utilisation d'un module dans un programme

Lorsque l'on veut utiliser un module, il faut le dire explicitement (**Utilise**).

```
Algorithme Application Est

Utilise
    Date

Variable
    d1, d2: Date
Début
    initialiser_date(d1, 1, 1, 2000)
    afficher_date(d1)
    initialiser_date(d2, 31, 12, 2000)
    afficher_date(d2)
Fin
```

Remarque : Utilise Date donne accès à toutes les informations déclarées dans la spécification du module Date. Elles peuvent donc être utilisées dans le programme.
Remarque : Connaître l'implémentation du module n'est pas nécessaire !

Implémentation du module Date

```
Module Date Implémentation Est

Procédure initialiser_date(une_date: out Date
                           jj, mm, aaaa: in Entier)
    Début
        une_date.jour <- jj
        une_date.mois <- mm
        une_date.année <- aaaa
    Fin

Procédure afficher_sur_deux_chiffres(un_entier: in Entier)
        -- Afficher un_entier sur au moins deux chiffres en
        -- ajoutant éventuellement un zéro en tête.
        --
        -- Nécessite
        --      (un_entier >= 0) Et (un_entier <= 99)
    Début
        Si un_entier < 10 Alors
            Écrire('0')
        FinSi
        Écrire(un_entier)
    Fin

Procédure afficher_date(une_date: in Date)
    Début
```

Implémentation du module Date (2)

```
        -- Afficher le jour
        afficher_sur_deux_chiffres(une_date.jour)
        Écrire('/')

        -- Afficher le mois
        afficher_sur_deux_chiffres(une_date.mois)
        Écrire('/')

        -- Afficher l'année
        Écrire(une_date.année)
    Fin
Fin
```

- Le SP `afficher_sur_deux_chiffres` est local au module.
- Seule est donnée la sémantique des SP locaux aux modules.
- Dans l'implémentation d'un module, il est possible de mettre un bloc d'instructions pour initialiser le module (si c'est nécessaire).
  Attention, il ne s'agit en aucun cas d'un programme principal !

Exercice 32 Que faudrait-il changer pour contrôler la validité des dates ?

Utilisation d'un module dans un autre module

Si un module veut utiliser une information d'un autre module, il suffit de l'indiquer par un **`Utilise`** qui peut être placé :

- soit dans sa spécification,
- soit dans son implémentation.

Attention : L'information **doit** être dans la spécification de l'autre module !
Remarque : Les modules « utilisés » dans la spécification le sont également pour l'implémentation (inutile de remettre un **`Utilise`**).
Règle : Toujours mettre le **`Utilise`** dans l'implémentation, sauf s'il est nécessaire de le mettre dans la spécification.
Justification : La dépendance est moins forte si **`Utilise`** est mis dans l'implémentation.
Attention : Deux modules A et B ne peuvent pas s'utiliser mutuellement dans la spécification (mais ils le peuvent dans l'implémentation).

Exercices

Exercice 33 : Définition d'un module RendezVous
Écrire un module qui définit un type RendezVous sachant qu'un rendez-vous est caractérisé par une heure de début, une heure de fin et un intitulé décrivant la nature du rendez-vous.
Deux opérations sont définies sur un rendez-vous, l'initialiser et l'afficher.

Exercice 34 : Définition d'un module Point
Écrire un module qui définit un type Point et les opérations associées. Un point est représenté dans un repère cartésien par son abscisse et son ordonnée. Les opérations permettent de l'initaliser, de calculer la distance entre deux points et d'obtenir les coordonnées du point dans un repère polaire (module et argument).
On suppose qu'un module « math » définit les fonctions trigonométriques.

Intérêt des modules

Les modules ont plusieurs intérêts :

- **structuration** de l'application à un niveau supplémentaire par rapport aux sous-programmes (conception globale du modèle en V) ;
  Critères de *décomposabilité*, *composabilité* et *compréhensibilité*
- **factorisation/réutilisation** de code (SP) entre applications ;
- **amélioration de la maintenance** (une évolution dans l'implémentation d'un module n'a pas d'impact sur les autres modules d'une application) ;
  Critère de *continuité* : un petit changement dans la spécification du problème doit se traduire par un changement dans un (ou quelques) module(s).
- **amélioration du temps de compilation** grâce à la compilation séparée de chaque module.

Taxonomie des modules

Un module est un regroupement d'information sans sémantique imposée. Cependant, on peut identifier deux catégories de modules.

- Les **modules utilitaires** qui sont un regroupement de sous-programmes liés à un domaine :
  - module mathématique ;
  - module de gestion de l'affichage...

  Les sous-programmes sont reliés par un thème commun.
- Les **modules issus d'un type abstrait de données**. Ils encapsulent un type et les opérations associées :
  - un module date ;
  - un module fraction...