Algorithmique et programmation : les bases (Algo)
Corrigé

Résumé

Ce document décrit les éléments de base de notre langage algorithmique : la structure d'un algorithmique, les variables, les types, les constantes, les expressions et les instructions.

Table des matières

Liste des exercices

1 Pourquoi définir notre langage algorithmique ?

Plusieurs raisons justifient l'utilisation d'un langage algorithmique spécifique :
— bien insister sur la différence entre programmation et construction d'une solution algorithmique.
— être indépendant d'un langage de programmation particulier : même si ce langage est proche de Pascal dans sa structure, il n'est pas identique, et peut être très facilement traduit dans d'autres langages de programmation tels que C, Ada, Fortran, Modula-2, etc.
— utiliser des concepts intéressants issus de plusieurs langages comme par exemple :
  — les structures de contrôle de Pascal avec les mots-clés de fin de structure de Modula-2 ;
  — les modes de passages des paramètres aux sous-programmes d'Ada ;
  — la variable prédéfinie **`Résultat`** d'Eiffel pour les fonctions, etc.
— favoriser la créativité en ayant un langage souple (permettant par exemple des instructions abstraites) mais suffisamment rigoureux pour que tout le monde puisse comprendre un algorithme écrit dans ce langage.

2 Structure d'un algorithme

Un algorithme (comme un programme) est composé de trois parties principales :

1. La partie **définitions** 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. La partie **déclarations** permet de déclarer les données qui sont utilisées par le programme. Les données sont représentées par des variables.
3. La partie **instructions** constitue le **programme principal**. Les instructions sont exécutées par l'ordinateur pour transformer les données.

2.1 Exemple d'algorithme : calculer le périmètre d'un cercle

Un exemple d'algorithme/programme est donné ci-dessous. Il décrit comment obtenir le périmètre d'un cercle à partir de son diamètre. Cet exemple est volontairement très simple.

Listing 1 – Algorithme pour calculer le périmètre d'un cercle

```
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
```

2.2 Structure de l'algorithme

L'algorithme est composé d'un nom. Ensuite, un commentaire général explique l'objectif de l'algorithme.

Dans le partie « définition », nous avons défini la constante PI.

Dans la partie « déclaration », deux variables sont déclarées pour représenter respectivement le rayon du cercle et son périmètre.

Dans la partie « instruction », les instructions permettent d'afficher (**Écrire**) des informations à l'utilisateur du programme (« Rayon = ») et de lui demander de saisir la valeur du rayon (**Lire**) et d'initialiser la variable périmètre. L'algorithme se termine avec l'affichage du périmètre pour que l'utilisateur puisse le voir sur l'écran.

Exercice 1 : Lien entre raffinage et algorithme

Donner le raffinage qui a conduit à l'algorithme décrit dans le listing 1.

Solution :

```
1   R0 : Déterminer le périmètre d'un cercle à partir de son rayon
2
3   R1 : Raffinage De « Déterminer le périmètre d'un cercle à partir de son rayon »
4       | Saisir le rayon                   rayon: out Réel
5       | Calculer le périmètre             rayon: in ; périmètre: out Réel
6       | Afficher le périmètre             périmètre: in
7
8   R2 : Raffinage De « Saisir le rayon »
9       | Écrire("Donner le rayon du cercle : ")
10      | Lire(rayon)
11
12  R2 : Raffinage De « Calculer le périmètre »
13      | périmètre <- 2 * PI * rayon
14
15  R2 : Raffinage De « Afficher le périmètre »
16      | ÉcrireLn("Le périmètre est : ", périmètre)
17
18  Le dictionnaire des données est :
```

```
19      rayon: Réel          -- le rayon du cercle saisi au clavier
20      périmètre: Réel;     -- le périmètre du cercle
```

2.3 Identificateurs

Les entités qui apparaissent (le programme, les variables, les constantes, les types, les sous-programmes, etc.) doivent avoir un nom. Ce nom permet à l'ordinateur de les distinguer et aux hommes de les comprendre et de les désigner. Les noms ainsi donnés sont appelés **identificateurs**. Un identificateur commence par une lettre ou un souligné (_) et se continue par un nombre quelconque de lettres, chiffres ou soulignés.

2.4 Commentaires

Notre langage algorithmique propose deux types de commentaires, l'un introduit par deux tirets et qui se termine à la fin de la ligne et l'autre délimité par des accolades. Nous donnerons une signification particulière à ces deux types de commentaires :

— Les commentaires introduits par deux tirets seront utilisés pour faire apparaître les raffinages, donc la structure de l'algorithme, dans l'algorithme lui-même. Les commentaires précèdent et sont alignés avec les instructions qu'ils décrivent.
Ils peuvent être également utilisés pour expliquer ce qui a été fait. Dans ce cas, ils sont placés à la fin de la ligne et ont un rôle de justification. Ils sont également utilisés pour expliquer le rôle d'une variable ou d'une constante, etc.

— Les commentaires entre accolades seront utilisés pour mettre en évidence une propriété sur l'état du programme. Ces commentaires contiennent en général une expression booléenne qui doit être vraie à cet endroit du programme.

Dans l'exemple du calcul du périmètre d'un cercle (listing 1), nous trouvons ces différents types de commentaires :

— le premier commentaire explique l'objectif du programme (R0) ;
— les commentaires « saisir le rayon », « calculer le périmètre » et « afficher le périmètre » correspondent aux étapes identifiées lors du raffinage ;
— les commentaires « le rayon du cercle lu au clavier » et « le périmètre du cercle » expliquent le rôle de la variable ;
— le commentaire `{ périmètre = 2 * PI * rayon }` indique qu'à ce moment du programme, la valeur de la variable périmètre est la même que celle de l'expression `2 * PI * rayon`.

3 Variables

3.1 Qu'est ce qu'une variable ?

Un algorithme est fait pour résoudre un ensemble de problèmes semblables (cf définition du terme « algorithmique » dans le petit Robert). Par exemple, un logiciel qui gère la facturation d'une entreprise doit connaître les noms et adresses des clients, les produits commandés, etc. Ces

informations ne peuvent pas être devinées par le programme et doivent donc être saisies par les utilisateurs. Ces informations sont appelées *données en entrée*. Elles proviennent de l'extérieur du programme et sont utilisées dans le programme.

Inversement, le programme effectue des opérations, des calculs dont les résultats devront être transmis aux utilisateurs. Par exemple, le total d'une facture est calculée comme la somme de tous les produits commandés en multipliant le nombre d'articles par leur prix unitaire ; le total des ventes pour une période donnée est obtenu en faisant la somme des montants des factures, etc. Ces informations seront affichées à l'écran, imprimées, stockées dans des bases de données, etc. Ce sont des informations qui sortent du programme. On les appelle *données en sortie*.

Le programmeur, pour décrire son algorithme utilise des variables pour représenter les données manipulées par un programme. Ce peut être les données en entrée, les données en sortie mais également les données résultant de calculs intermédiaires. Ainsi, pour calculer le montant d'une ligne d'une facture, le programmeur expliquera que le montant est obtenu en multipliant le prix unitaire par la quantité d'articles commandés. Il utilisera donc trois variables :

— `prix_unitaire`, le prix unitaire de l'article ;
— `quantité`, la quantité d'articles commandé ;
— et `montant`, le montant de la ligne du bon de commande.

3.2 Définition d'une variable

Une variable peut être vue comme une zone dans la mémoire (vive) de l'ordinateur qui est utilisée pour conserver les données qui seront manipulées par l'ordinateur.

Une variable est caractérisée par quatre informations :

— son **rôle** : il indique à quoi va servir la variable dans le programme. Par exemple, on peut utiliser une variable pour conserver le prix unitaire d'un article, prix exprimé en euros. Cette information doit apparaître dans l'algorithme sous la forme d'un commentaire informel (texte libre) associé à la variable ;
— son **nom** : composé uniquement de lettres minuscules, majuscules, de chiffres et du caractère souligné, il permet d'identifier la variable. On parle d'« identificateur ». Ce nom doit être significatif (refléter le rôle de la variable). Un compromis doit bien sûr être trouvé entre expressivité et longueur. On peut par exemple appeler `prix_unitaire` une variable qui représente le prix unitaire d'un article. Le nom `prix_unitaire_d_un_article` est un nom beaucoup trop long ;
— son **type** : une variable est utilisée pour représenter des données qui sont manipulées par le programme. Un type est utilisé pour caractériser l'ensemble des valeurs qu'une variable peut prendre. Par exemple le prix unitaire représente le prix d'un article exprimé en euros. On peut considèrer qu'il est représenté par un réel. Il ne peut alors prendre que ce type de valeurs. De la même manière la quantité, ne peut prendre que des valeurs entières et le nom est une chaîne de caractères. On dira respectivement que le type de `prix_unitaire` est **`Réel`**, le type de `quantité` est **`Entier`** et le type de `nom` est **`Chaîne`**.

```
1 prix_unitaire: Réel      -- prix unitaire d'un article (en euros)
2 quantité: Entier         -- quantité d'articles commandés
3 nom: Chaîne              -- nom de l'article
```

— sa **valeur** : La variable contient une information qui peut varier au cours de l'exécution d'un programme. C'est cette information que l'on appelle valeur de la variable. La valeur d'une variable doit correspondre au type de la variable. Ainsi, une variable `quantité` de type entier pourra prendre successivement les valeurs de 10, 25 et 3.

La valeur d'une variable n'existe que lorsque le programme est exécuté. Les autres informations (nom, rôle et type) sont définies lors de la conception du programme, pendant la construction de l'algorithme. Le rôle, le nom et le type sont des informations statiques qui doivent être précisées lors de la déclaration de la variable. En revanche, la valeur est une information dynamique qui changera au cours de l'exécution du programme.

Règle : Une variable doit toujours être initialisée avant d'être utilisée.

Définition : Une variable est un nom désignant symboliquement un emplacement mémoire typé auquel est associé une valeur courante. Cette valeur peut être accédée [1] (expressions, section 6) ou modifiée (affectation, section 8).

Attention : Le nom d'une variable doit être significatif : il doit suggérer, si possible sans ambiguïté, la donnée représentée par cette variable.

Règle : En général, dès qu'on identifie une variable, il faut mettre un commentaire qui explique ce qu'elle représente et le rôle qu'elle joue dans l'algorithme.

L'ensemble de variables et leur description constitue le **dictionnaire des données**.

4 Types fondamentaux

Définition : Un type caractérise les valeurs que peut prendre une variable. Il définit également les opérations, généralement appelées opérateurs, qui pourront être appliquées sur les données de ce type.

On appelle types fondamentaux les types qui sont définis dans notre langage algorithme par opposition aux types structurés (tableaux, enregistrements et types énumérés) qui doivent être définis par le programmeur.

Intérêts : Les types ont deux intérêts principaux :

— Permettre de vérifier automatiquement (par le compilateur) la cohérence de certaines opérations. Par exemple, une valeur définie comme entière ne pourra par recevoir une valeur chaîne de caractères.

— Connaître la place nécessaire pour stocker la valeur de la variable. Ceci est géré par le compilateur du langage de programmation considéré et est en général transparent pour le programmeur.

Opérateur : À chaque type est associé un ensemble d'opérations (ou opérateurs).

Opérateurs de comparaison : Tous les types présentés dans cette partie sont munis des opérations de comparaison suivantes : `<`, `>`, `<=`, `>=`, `=` et `<>`.

1. Le verbe « lire » est parfois utilisé en informatique pour indiquer que l'on accède à la valeur de la variable. Il ne faut pas confondre ce « lire » avec l'instruction d'entrée/sortie de même nom et qui permet d'initialiser une variable à partir d'une valeur lue sur périphérique d'entrée tel que le clavier. Il ne faut donc pas confondre les deux sens de « lire » : « accèder à la valeur d'une variable » et « saisir la valeur d'une variable ».

Les opérateurs de comparaison sont des opérateurs à valeur booléenne (cf type booléen, section 4.3).
Attention : Tous les types manipulés par une machine sont discrets et bornés. Ces limites dépendent du langage de programmation et/ou de la machine cible considérés. Aussi, nous n'en tiendrons pas compte ici. Il faut cependant garder à l'esprit que toute entité manipulée est nécessairement bornée.

4.1 Les entiers

Le type **Entier** caractérise les entiers relatifs.

```
1      -- Exemple de représentants des entiers
2      10
3      0
4      -10
```

Les opérations sont : +, -, ∗, **Div** (division entière), **Mod** (reste de la division entière) et **Abs** (valeur absolue)
- et + peuvent être unaires ou binaires.

```
1  10 Mod 3  -- 1 (le reste de la division entière de 10 par 3)
2  10 Div 3  -- 3 (le quotient de la division entière de 10 par 3)
3  1 Div 2   -- 0 (le quotient de la division entière de 1 par 2)
4  Abs(-5)   -- 5 (l'entier est mis entre parenthèses (cf sous-programmes))
```

4.2 Les réels

Le type **Réel** caractérise les réels.

```
1      -- Exemple de représentants des réels
2      10.0
3      0.0
4      -10e-4
```

Les opérations sont : +, -, ∗, /, **Abs** (valeur absolue), Trunc (partie entière).
- et + peuvent être unaires ou binaires.

4.3 Les booléens

Le type **Booléen** caractérise les valeurs booléennes qui ne correspondent qu'à deux valeurs **VRAI** ou **FAUX**.

Les opérations sont **Et**, **Ou** et **Non** qui sont définies par la table de vérité suivante :

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, section 9).

```
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 2 : Vérifier les formules de De Morgan
En utilisant des tables de vérité, vérifier que les formules de De Morgan sont correctes.

Remarque :

```
1          A: Booléen      -- A est une variable de type Booléen
2
3          A     est équivalent à A = VRAI
4          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 !

4.4 Les caractères

Le type **`Caractère`** caractérise les caractères.

```
1      -- Exemple de représentants des caractères
2      'a'         -- le caractère  a
3      '\''        -- le caractère  '
4      '\\'        -- le caractère  \
5      '\n'        -- retour à la ligne
6      '\t'        -- Caractère tabulation
```

Remarque : On considère que les lettres majuscules, les lettres minuscules et les chiffres se suivent. Ainsi, pour savoir si un variable c de type caractère correspond à un chiffre, il suffit d'évaluer l'expression `(c >= '0') Et (c <= '9')`.
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`.

4.5 Les chaînes de caractères

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

```
1      "Une␣chaîne␣de␣caractères"          -- un exemple de Chaîne
2      "Une␣chaîne␣avec␣guillement␣(\")"   -- un exemple de Chaîne
```

Remarque : Nous verrons plus en détail les chaînes de caractères lorsque nous traiterons de la structure de données « tableau ».
Attention : Il ne faut pas confondre le nom d'une variable et une constante chaîne de caractères !

5 Constantes

Certaines informations manipulées par un programme ne changent jamais. C'est par exemple la cas de la valeur de π, du nombre maximum d'étudiants dans une promotion, etc.

Ces données ne sont donc pas variables mais constantes. Plutôt que de mettre explicitement leur valeur dans le texte du programme (constantes littérales), il est préférable de leur donner un nom symbolique (et significatif). On parle alors de constantes (symboliques).

```
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 15/09/2000 (en %)
4 CAPACITÉ = 120   -- Nombre maximum d'étudiants dans une promotion
5 INTITULÉ = "Algorithmique␣et␣programmation"     -- par exemple
```

π peut être considérée comme une constante absolue. Son utilisation permet essentiellement de gagner en clarté et lisibilité. Les constantes MAJORITÉ, TVA, CAPACITÉ, etc. sont utilisées pour paramètrer le programme. Ces quantités ne peuvent pas changer au cours de l'exécution d'un programme. Toutefois, si un changement doit être réalisé (par exemple, la précision de PI utilisée n'est pas suffisante), il suffit de changer la valeur de la constante symbolique dans sa définition (et de recompiler le programme) pour que la modification soit prise en compte dans le reste de l'algorithme. [2]

Question : Quel est l'intérêt d'une constante symbolique ?

Solution : Au moins les deux explications ci-dessus.

6 Expressions

Définition : Une expression est « quelque chose » qui a une valeur. Ainsi, en fonction de ce qu'on a vu jusqu'à présent, une expression est une constante, une variable ou toute combinaison d'expressions en utilisant les opérateurs arithmétiques, les opérateurs de comparaison ou les opérateurs logiques.

Exemple : Voici quelques exemples de valeurs réelles :

```
1 10.0            -- une constante litérale
2 PI              -- une constante symbolique
3 rayon           -- une variable de type réel
4 2*rayon         -- l'opérateur * appliqué sur 2 et rayon
5 2*PI*rayon      -- une expression mêlant opérateurs, constantes et variables
6 rayon >= 0      -- expression avec un opérateur de comparaison
```

Attention : Pour qu'une expression soit acceptable, il est nécessaire que les types des opérandes d'un opérateur soient compatibles. Par exemple, faire l'addition d'un entier et d'un booléen n'a pas de sens. De même, on ne peut appliquer les opérateurs logiques que sur des expressions booléennes.

Quelques règles sur la compatibilité :

— Deux types égaux sont compatibles.

2. Ceci suppose d'arrêter le programme, de le recompiler et de l'exécuter à nouveau.

— On peut ensuite prendre comme règle : le type A est compatible avec le 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.
Si l'on écrit `n + x` avec `n` entier et `x` réel, alors il y a « coertion » : l'entier `n` est converti en un réel puis l'addition est réalisée sur les réels, le résultat étant bien sûr réel.

Règle d'évaluation d'une expression : Une expression est évaluée en fonction de la priorité des opérateurs qui la composent, en commençant par ceux de priorité plus forte. À priorité égale, les opérateurs sont évalués de gauche à droite.

Voici les opérateurs rangés par priorité décroissante [3] :

`.` (accès à un champ)	la priorité la plus forte
`+`, `-`, **`Non`** (unaires)	
`*`, `/`, **`Div`**, **`Mod`**, **`Et`**	
`+`, `-`, **`Ou`**	
`<`, `>`, `<=`, `>=`, `=` et `<>`	priorité la plus faible

Il est toujours possible de mettre des parenthèses pour modifier la priorité.

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

En cas d'ambiguïté (ou de doute), il est préférable de mettre des parenthèses.

Exercice 3 : Parenthèser

Parenthéser complètement les expressions suivantes. Peuvent-elles être considérées comme correctes et si oui, à quelles conditions, si non pourquoi ?

```
1  2 + x * 3
2  - x + 3 * y <= 10 + 3
3  x = 0 Ou y = 0
```

Solution :

```
1  (2 + (x * 3))                          -- ok si x Entier (ou Réel)
2  (((- x) + (3 * y)) <= (10 + 3))        -- ok si x et y Entier (ou Réel)
3  ((x = (0 Ou y)) = 0)                   -- incorrecte car 0 non booléen
```

Exercice 4 : Validité d'instructions

À quelles conditions les instructions suivantes sont-elles cohérentes ?

```
1      (a Mod b)
2      (rep = 'o') Ou (rep = 'n')
3      (R <> 'O') Et (R <> 'o')
4      (C >= 'A') Et (C <= 'Z') Ou (C >= 'a') Et (C <= 'z')
5      termine Ou (nb > 10)
6      Non trouve Ou (x = y)
```

Solution :

3. Attention, nous donnons ici les priorités du pseudo-langage. Si la priorité des opérateurs arithmétiques est la même quelque soit le langage, il n'en va de même pour les opérateurs de comparaison.

```
1       a, b, nb : Entier
2       rep, R, C : Caractère
3       termine, trouve : Booléen
```

Remarque : Les fonctions que nous verrons par la suite sont également des expressions. Elles sont assimilables à des opérateurs définis par le programmeur.
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 composition de une ou deux expressions booléennes reliées par un opérateur booléen (**Et**, **Ou**, **Non**).

7 Instructions d'entrée/sorties

Le point de référence est le programme. Ainsi les informations d'entrée sont les informations produites à l'extérieur du programme et qui rentrent dans le programme (saisie clavier par exemple), les informations de sortie sont élaborées par le programme et transmises à l'extérieur (l'écran par exemple).

7.1 Opération d'entrée

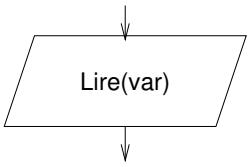
Lire(var) : lire sur le périphérique d'entrée une valeur et la ranger dans la variable var.

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 :

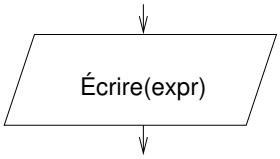


7.2 Opération de sortie

Écrire(expr) : transférer (afficher, imprimer...) la valeur de l'expression expr vers le périphérique de sortie.
Règle : « expr » est une expression quelconque d'un type fondamental.

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 :



Variantes : Nous utiliserons une variante `EcrireLn(expr)` qui ajoute un saut de ligne sur le périphérique de sortie après avoir affiché la valeur de l'expression.
Remarque : **`Lire`** et **`Écrire`** sont deux « instructions » qui peuvent prendre en paramètre n'importe quel type fondamental. Si la variable est entière, **`Lire`** lit un entier, si c'est une chaîne de caractères, alors c'est une chaîne de caractères qui est saisie. On dit que **`Lire`** et **`Écrire`** sont polymorphes.

Exercice 5 : Cube d'un réel
Écrire un programme qui affiche le cube d'un nombre réel saisi au clavier.
Solution :

```
1  R0 : Afficher le cube d'un nombre réel
2
3  Tests :
4      0 -> 0
5      1 -> 1
6      2 -> 8
7      -2 -> -8
8      1.1 -> 1,331
9
10 R1 : Raffinage De « Afficher le cube d'un nombre réel »
11   | Saisir un nombre réel       x: out Réel
12   | Afficher le cube de x       x: in Réel
13
14 R2 : Raffinage De « Afficher le cube de x »
15   | Écrire(x * x * x)
```

On en déduit alors l'algorithme suivant :

```
1  Algorithme cube
2
3      -- afficher le cube d'un nombre réel
4
5  Variable
6      x: Réel     -- un nombre saisi par l'utilisateur
7
8  Début
9     -- Saisir un nombre réel
10    Écrire("Nombre = ")
11    Lire(x)
12
13    -- Afficher le cube de x
```

```
14      Écrire("Son cube est : ")
15      Écrire(x * x * x)
16  Fin
```

8 Affectation

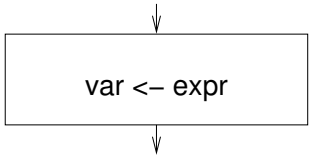
Définition : L'affectation permet de modifier la valeur associée à une variable.

```
1       var <-  expr
```

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`.

Organigramme :



Exemple : Voici quelques exemples d'affectation :

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

Exercice 6 : Comprendre l'affectation
Quelle est la valeur de `n` après l'exécution de chacune des instructions suivantes ?

```
1       n <- 5
2       c <- 'c'
3       Lire (n)
4       n <- 10
5       n <- n + 1
```

Solution :
La variable n prend pour valeurs successives : 5, la valeur saisie par l'utilisateur du programme, 10 et enfin 11.
Après ces affectations les valeurs respectives de `n` et `c` sont `11` et `'c'`.

Exercice 7 : Permuter deux caractères
Écrire un programme qui permute la valeur de deux variables `c1` et `c2` de type caractère.
Solution : Le principe est d'utiliser une variable intermédiaire (tout comme on utilise un récipient intermédiaire si l'on veut échanger le contenu de deux bouteilles). On en déduit donc l'algorithme suivant :

```
Variable
    c1, c2: Caractère   -- les deux caractères à permuter
    tmp: Caractère      -- notre intermédiaire
Début
    -- initialiser c1 et c2
    ...

    -- permuter c1 et c2
    tmp <- c1
    c1 <- c2
    c2 <- tmp

    { les valeurs de c1 et c2 ont été permutées }

    ...
Fin
```

Exercice 8 : Cube d'un réel (avec une variable)
Reprenons l'exercice 5.

8.1 Utiliser une variable intermédiaire pour le résoudre.
Solution : On reprend le même R0 et les mêmes tests. En fait, seule la manière de résoudre le problème change.

```
R1 : Raffinage De « Afficher le cube d'un nombre réel »
  | Saisir un nombre réel       x: out Réel
  | Calculer le cube de x       x: in Réel ; cube: out Réel
  | Afficher le cube

R2 : Raffinage De « Afficher le cube de x »
  | cube <- x * x * x
```

On en déduit alors l'algorithme suivant :

```
Algorithme cube

    -- afficher le cube d'un nombre réel

Variable
    x: Réel     -- un nombre saisi par l'utilisateur
    cube: Réel  -- le cube de x

Début
   -- Saisir un nombre réel
   Écrire("Nombre = ")
   Lire(x)

   -- Calculer le cube de x
   cube <- x * x * x

   -- Afficher le cube
   Écrire("Son cube est : ")
   Écrire(cube)
```

```
20 Fin
```

8.2 Quel est l'intérêt d'utiliser une telle variable ?
Solution : L'intérêt d'utiliser une variable intermédiaire est d'améliorer la lisibilité du programme car elle permet de mettre un nom sur une donnée manipulée. Ici on nomme `cube` la donnée `x * x * x`.

De plus, ceci nous a permis, au niveau du raffinage, de découpler le calcul du cube de son affichage. Il est toujours souhaitable de séparer calcul des opérations d'entrées/sorties car l'interface avec l'utilisateur est la partie d'une application qui a le plus de risque d'évoluer.

8.3 Exécuter à la main l'algorithme ainsi écrit.
Solution : À faire soi-même !

9 Structures de contrôle

Dans un langage impératif, on peut définir l'état d'un programme en cours d'exécution par deux choses :

— 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 (les valeurs des variables sont indéterminées) à un état final considéré comme le résultat.

Les structures de contrôle décrivent comment les affectations s'enchaînent séquentiellement. Elles définissent dont le transfert du contrôle après la fin de l'exécution d'une instruction.
Remarque : Ceci était avant réalisé en utilisant des « goto », instruction qui a été banni en 1970 avec la naissance de la programmation structurée.

En programmation structurée, le transfert de contrôle s'exprime par :

1. enchaînement séquentiel (une instruction puis la suivante) ;
2. traitements conditionnels ;
3. traitements répétitifs (itératifs) ;
4. appel d'un sous-programme (un autre programme qui réalise un traitement particulier).

Pour chacune des structures de contrôle présentées ci-après, sont données la syntaxe de la structure dans notre langage algorithmique, les règles à respecter (en particulier pour le typage), la sémantique (c'est-à-dire la manière dont elle doit être comprise et donc interprétée), des exemples ou exercices à but illustratifs.

9.1 Enchaînement séquentiel

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

```
1     opération_1
2     ...
3     opération_n
```

Exemple :

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

Question : Quand utilisera-t-on cette séquence de trois instructions ? Que permet-elle de faire ?
Solution : Quand on veut permuter les valeurs des deux variables a et b.
Remarque : On utilise parfois le terme d'instruction composée.

9.2 Instructions conditionnelles

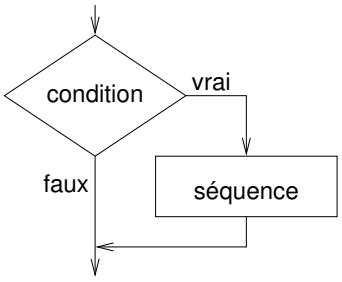
9.2.1 Conditionnelle `Si ... Alors ... FinSi`

```
1 Si condition Alors
2     séquence    -- une 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`**.

En d'autres termes, la séquence est exécutée si et seulement si la condition est **`VRAI`**.
Organigramme :



Exercice 9 : Une valeur entière est-elle paire ?
Écrire un algorithme qui lit une valeur entière au clavier et affiche « paire » si elle est paire.
Solution :

```
1  R0 : Afficher « paire » si une valeur entière saisie au clavier est paire
2
3  tests :
4      2 -> paire
5      5 -> -----
6      0 -> paire
7
8  R1 : Raffinage De « Afficher ... »
9    | Saisir la valeur entière n
10   | Afficher le verdict de parité
```

```
11
12 R2 : Raffinage De « Afficher le verdict de parité »
13   | Si n est paire Alors
14   |   | Écrire("paire")
15   | FinSi
16
17 R3 : Raffinage De « n est paire »
18   | Résultat <- n Mod 2 = 0
```

Dans le raffinage précédent un point est à noter. Il s'agit du raffinage R2 qui décompose « Afficher le verdict de parité ». Nous n'avons pas directement mis la formule « `n Mod 2 = 0` ». L'intérêt est que la formulation « n est paire » est plus facile à comprendre. Avec la formule, il faut d'abord comprendre la formule, puis en déduire sa signification. « n est paire » nous indique ce qui nous intéresse comme information (facile à lire et comprendre) et son raffinage (R3) explique comment on détermine si n est paire. Le lecteur peut alors vérifier la formule en sachant ce qu'elle est sensée représenter.

Raffiner est quelque chose de compliquer car on a souvent tendance à descendre trop vite dans les détails de la solution sans s'arrêter sur les étapes intermédiaires du raffinage alors que ce sont elles qui permettent d'expliquer et de donner du sens à la solution.

Dans cet exercice, vous vous êtes peut-être posé la question : « mais comment sait-on que n est paire ». Si vous avez trouvé la solution vous avez peut-êre donnée directement la formule alors que le point clé est la question. Il faut la conserver dans l'expression de votre algorithme ou programme, donc en faire une étape du raffinage.

Si vous arrivez sur une étape que vous avez du mal à décrire, ce sera toujours une indication d'une étape qui doit apparaître dans le raffinage. Cependant, même pour quelque chose de simple, que vous savez faire directement, il faut être capable de donner les étapes intermédiaires qui conduisent vers et expliquent la solution proposée. Ceci fait partie de l'activité de construction d'un programme ou algorithme.

Remarque : Il est généralement conseillé d'éviter de mélanger traitement et entrées/sorties. C'est pourtant ce qui a été fait ci-dessus. On aurait pu écrire le premier niveau de raffinage différemment en faisant.

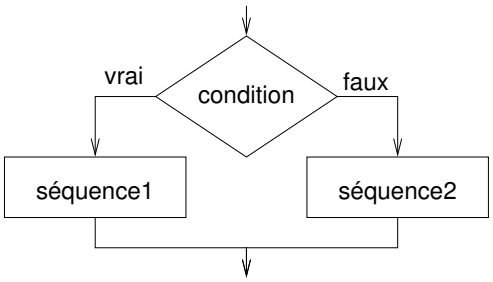
```
1  R1 : Raffinage De « Afficher ... »
2    | Saisir la valeur entière            n: out Entier
3    | Déterminer la parité de n           n: in ; paire: out Booléen
4    | Afficher le verdict de parité       paire: in Booléen
5
6  R2 : Raffinage De « Déterminer la parité de n »
7    | parité <- (n Mod 2) = 0
8
9  R2 : Raffinage De « Afficher le verdict de parité »
10   | Si paire Alors
11   |   | Écrire("paire")
12   | FinSi
```

On constate ici que la variable intermédiaire « paire » permet d'avoir un programme plus lisible car on a donné un nom à la quantité `(n Mod 2) = 0`.

9.2.2 Conditionnelle `Si ... Alors ... Sinon ... FinSi`

```
1     Si condition Alors
2         séquence1        -- séquence exécutée ssi condition est VRAI
3     Sinon        { Non condition }
4         séquence2        -- séquence exécutée ssi condition est FAUX
5     FinSi
```

Évaluation : Si la condition est vraie, c'est $\texttt{séquence}_1$ qui est exécutée, sinon c'est $\texttt{séquence}_2$. Dans les deux cas, après l'exécution de la séquence, l'instruction suivante à exécuter est celle qui suit le **`FinSi`**.
Remarque : Il est recommandé de faire apparaître sous forme de commentaire la condition associée à la partie **`Sinon`**.
Organigramme :



Exercice 10 : Maximum de deux valeurs réelles
Étant données deux valeurs réelles lues au clavier, afficher à l'écran la plus grande des deux.
Solution :

```
1  R0 : Afficher le plus grand de deux réels saisis au clavier
2
3  tests :
4      1 et 2 -> 2
5      2 et 1 -> 1
6      3 et 3 -> 3
7
8  R1 : Raffinage De « Afficher le plus grand de deux réels ... »
9    | Saisir les deux réels       x1, x2 : out Réel
10   | Déterminer le maximum       x1, x2 : in ; max : out Réel
11   | Afficher le maximum
12
13 R2 : Raffinage De « Déterminer le maximum »
14   | Si x1 > x2 Alors
15   |   | max <- x1
16   | Sinon
17   |   | max <- x2
18   | FinSi
```

L'algorihme est alors le suivant :

```
1  Algorithme calculer_max
2
3      -- Afficher le plus grand de deux réels saisis au clavier
```

```
4
5  Variable
6      x1, x2: Réel        -- les deux réels saisis au clavier
7      max: Réel           -- le plus grand de x1 et x2
8
9  Début
10     -- Saisir les deux réels
11     Lire(x1, x2)
12
13     -- Déterminer le maximum
14     Si x1 > x2 Alors
15         max <- x1
16     Sinon
17         max <- x2
18     FinSi
19
20     -- Afficher le maximum
21     Écrire(max)
22 Fin
```

Exercice 11 : Sinon et Si

Réécrire une instruction **Si ... Alors ... Sinon ... FinSi** en utilisant seulement la structure **Si ... Alors ... FinSi** (sans utiliser le **Sinon**).

Solution : Soit la structure

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

On la réécrit sans utiliser le **Sinon** de la manière suivante :

```
1  Si condition Alors
2      séquence1
3  FinSi
4  Si Non condition Alors
5      séquence2
6  FinSi
```

Quel est l'intérêt du **Sinon** ?

Solution :

- Il apparaît clairement que les deux groupes d'instructions (les deux séquences) sont exclusifs. Au delà de la structure syntaxique, ceci est aussi renforcé par la présentation (et l'indentation).
- Accessoirement, le traitement peut être plus efficace puisque la condition ne sera évaluée qu'une seule fois.
- Si on veut faire du test par couverture en prenant comme critère de couverture « tous les chemins » on ne pourra jamais atteindre les 100% ! ! !

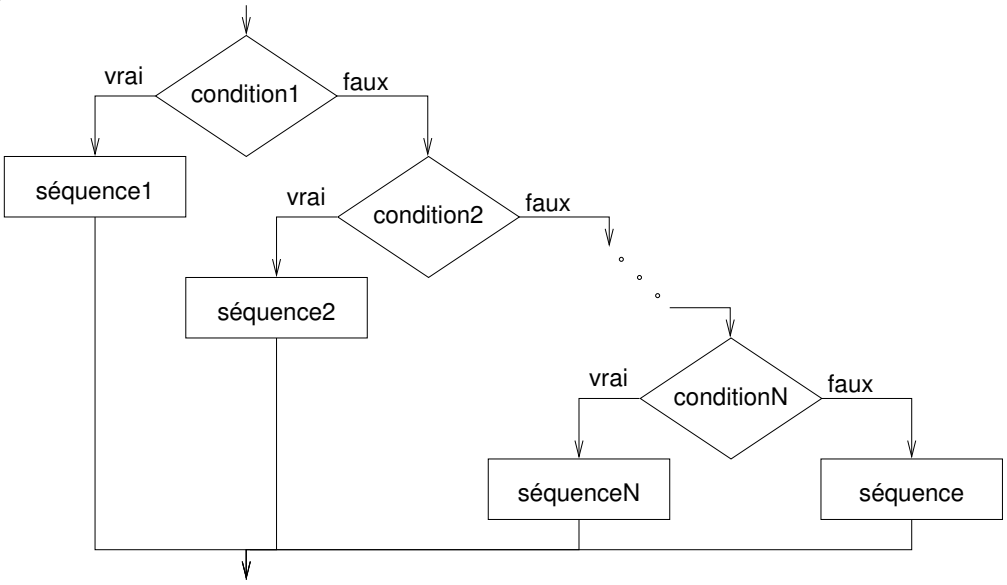
9.2.3 La clause `SinonSi`

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, alors la séquence associée au **`Sinon`**, si elle existe, est exécutée.

Organigramme :



Exercice 12 : Signe d'un entier

Étant donné un entier lu au clavier, indiquer s'il est nul, positif ou négatif.

Solution :

```
1  R0 : Afficher le signe d'un entier
2
3  tests :
4       2 -> positif
5       0 -> nul
6      -1 -> négatif
7
8  R1 : Raffinage De « Afficher le signe d'un entier »
```

```
9    | Saisir un entier n          n: out Entier
10   | Afficher le signe de n      n: in
11
12 R2 : Raffinage De « Afficher le signe de n »
13   | Si n > 0 Alors
14   |  | Écrire("positif");
15   | SinonSi  n < 0 Alors
16   |  | Écrire("positif");
17   | Sinon  { Non (n > 0) Et Non (n < 0)  donc  N = 0 }
18   |  | Écrire("nul");
19   | FinSi
```

Le principe est d'utliser un **SinonSi** car les trois cas sont exclusifs.

```
1  Algorithme signe_entier
2
3      -- Afficher le signe d'un entier
4
5  Variable
6      n: Entier   -- entier saisi au clavier
7
8  Début
9      -- Saisir un entier n
10     Lire(n)
11
12     -- Afficher le signe de n
13     Si n > 0 Alors
14         Écrire("positif");
15     SinonSi  n < 0 Alors
16         Écrire("positif");
17     Sinon  { Non (n > 0) Et Non (n < 0)  donc  N = 0 }
18         Écrire("nul");
19     FinSi
20 Fin
```

9.2.4 Conditionnelle `Selon`

```
1      Selon expression Dans
2          choix_1 :
3              séquence_1
4          choix_2 :
5              séquence_2
6          ...
7          choix_N :
8              séquence_N
9      Sinon
10         séquence
11     FinSelon
```

Règles :

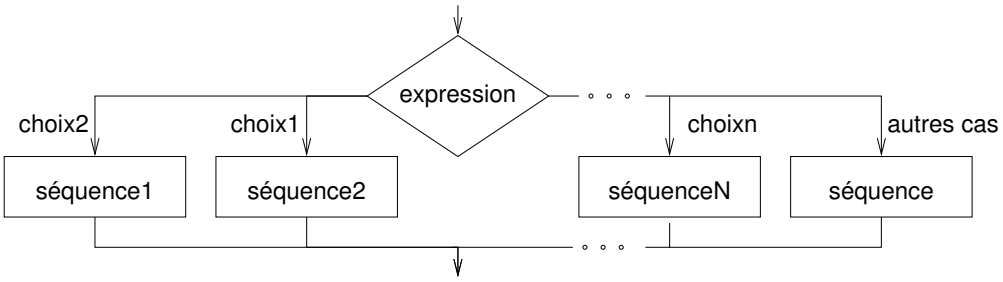
— expression est nécessairement une expression de type scalaire.

— choix$_i$ est une liste de choix séparés par des virgules. Chaque choix est soit une constante, soit un intervalle (`10..20`, par exemple).
L'instruction **`Selon`** peut donc être considérée comme un cas particulier de la conditionnelle
`Si ... SinonSi ... FinSi`.

Évaluation : L'`expression` est évaluée, puis sa valeur est successivement comparée à chacun des ensembles choix$_i$. Dès qu'il y a correspondance, les comparaisons sont arrêtées et la séquence associée est exécutée. Les différents choix sont donc *exclusifs*. Si aucun choix ne correspondant, alors la séquence associée au **`Sinon`**, si elle existe, est exécutée.

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 13 : Réponse

Écrire un programme qui demande à l'utilisateur de saisir un caractère et qui affiche « affirmatif » si le caractère est un « o » (minuscule ou majuscule), « négatif » si c'est un « n » (minuscule ou majuscule) et « ? ! ? ! ? ! ? » dans les autres cas.

Solution :

```
1  Algorithme repondre
2
3      -- Répondre par « affirmatif », « négatif » ou « ?!?!?!? ».
4
5  Variable
6      reponse: Charactere -- caractère lu au clavier
7
8  Début
9      -- saisir le caractère
10     Écrire("Votre␣réponse␣(o/n)␣:␣")
11     Lire(reponse)
12
13     -- afficher la réponse
14     Selon reponse Dans
15         'o', 'O':       { réponse positive }
16             Écrireln("Affirmatif␣!")
17
18         'n', 'N':       { réponse négative }
19             Écrireln("Négatif␣!")
20
21     Sinon
```

```
22          Écrireln("?!?!?!?")
23      FinSelon
24  Fin
```

9.3 Instructions de répétitions

9.3.1 Répétition `TantQue`

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

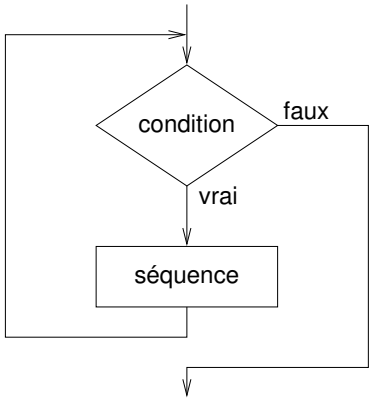
Règles :

— La condition doit être une expression booléenne.

— Pour que la boucle se termine, il est nécessaire que la séquence modifie la condition.

Évaluation : La condition est évaluée. Si elle vaut **`FAUX`** alors la boucle se termine et l'exécution se poursuit avec l'instruction qui suit **`FinTQ`**. Si elle vaut **`VRAI`** alors la séquence d'instructions est exécutée et la condition est de nouveau évaluée.

Organigramme :



Exercice 14 : Condition après un `FinTQ`

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

Solution : Puisqu'on est sortie de la boucle c'est que la condition du **`TantQue`** est fausse. On peut utiliser les commentaires *{}* pour l'exprimer (et les formules de De Morgan pour simplifier l'expression booléenne).

```
1  TantQue condition Faire
2      séquence
3  FinTQ
4  { Non condition }
```

Remarque : On fera apparaître explicitement cette condition sous forme de commentaire après 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.

Problème de la terminaison. Nous avons vu qu'une propriété importante d'un algorithme/programme est qu'il doit se terminer. Jusqu'à présent la terminaison était assurée car avec seulement la séquence et les conditionnelles, l'exécution du programme se fait toujours vers l'avant. On doit donc nécessairement atteindre la fin.

Avec les répétitions, on peut revenir en arrière dans le programme. Il est alors important de s'assurer que l'on finira toujours par sortir des répétitions d'un programme. Sinon, le programme risque de s'exécuter indéfiniment. On parle alors de boucle sans fin.

Pour garantir qu'une boucle (une répétition) se termine, on peut mettre en évidence un variant. C'est une expression entière qui doit toujours être positive et décroître strictement à chaque passage dans la boucle. Si on arrive à exhiber ce variant et prouver les deux propriétés, alors on est assuré que la répétition se termine.

Un programme qui se termine n'est pas forcément un programme valide. Pour vérifier la validité d'une boucle, on peut mettre en évidence un invariant. C'est une expression booléenne qui est vraie avant et après chaque passage dans la boucle. Elle aide à prouver la validité d'une boucle et d'un programme. L'invariant est cependant généralement difficile à trouver et la preuve difficile à faire. Cependant, dans les exercices nous essaierons de mettre en évidence des invariants, même s'ils ne sont qu'intuitifs et incomplets.

Exercice 15 : Somme des premiers entiers (TantQue)
Calculer la somme des n premiers entiers.

Solution : Une solution algorithmique sous forme de raffinages peut-être la suivante :

```
1  R0 :  Afficher la somme des n premiers entiers
2
3  R1 : Raffinage De « Afficher la somme des n premiers entiers »
4      Saisir la valeur de n (pas de contrôle)     n: out Entier
5      { n >= 0 }
6      Calculer la somme des n premiers entiers    n: in; somme: out Entier
7      Afficher la somme                           somme: in Entier
8
9  R2 : Raffinage De « Calculer la somme des n premiers entiers »
10     somme <- 0                  somme: out
11     i <- 1                      i: out Entier
12     TantQue i <= n Faire        i, n: in
13         { Variant : n - i + 1 }
14         { Invariant : somme = sum_{j=0}^{i-1} j }
15         somme <- somme + i      somme, i: in; somme: out
16         i <- i + 1              i: in; i: out
17     FinTQ
```

Intéressons nous à la condition après le **TantQue**. On a la propriété suivante :

```
    (i > n)              -- sortie du TantQue : Non (i <= n)
Et (n - i + 1 >= 0)      -- variant >= 0
Et (somme = sum_{j=1}^{i-1} j) -- invariant
```

Les deux premières expressions s'écrivent

```
(i > n ) Et (i <= n+1)
```

On en déduit : `i = n + 1`.

La troisième donne alors :

$$somme = \sum_{j=1}^{n} j$$

C'est bien le résultat demandé !

Bien entendu, il faut aussi prouver que le variant est toujours positif et qu'il décroit strictement (on incrémente i de 1 donc on diminue le variant de 1). Il faut également prouver que l'invariant est toujours vrai.

Commençons par le variant. Montrons par récurrence sur le nombre de passage dans la boucle que la **variant est toujours positif**.

Si le nombre de passage est nul, donc avant le premier passage, on a : $V_0 = n - i1 = n - 1 + 1 = n$. Par hypothèse sur n (saisie contrôlée), on a bien $V_0 \geq 0$

Supposons la propriété vraie pour le passage p. On a donc : $V_p = \geq 0$

Montrons que V_{p+1} est vraie. On notera avec des primes les variables de V_{p+1} au lieu d'utiliser des indices en p.

On a : $V_{p+1} = n' - i' + 1$

Si on parle de V_{p+1} c'est qu'on est passé dans la boucle. Donc la condition du **`TantQue`** est vraie. On a donc $i \leq n$.

Or on a $n' = n$ et $i' = i + 1$ (passage une fois dans la boucle).

Donc $V_{p+1} = n - (i + 1) + 1 = n - i + 1 - 1 = n - i$ Comme $i \leq n$, on a bien $V_{p+1} \geq 0$.

Par récurrence, on a montrer que le variant est toujours positif.

Montrons que le **variant décroit strictement**. On $V_{p+1} = n' - i' + 1 = n - i + 1 - 1 = (n - i + 1) - 1 = V_p - 1$. On a bien $V_{p+1} < V_p$.

On a donc montrer la terminaison de la boucle.

Remarque : Dans la formulation initiale du R1, j'avais oublié la propriété `{ n >= 0 }`. Elle était bien sûr implicite. Essayer de montrer que le variant était toujours positif m'a permis de penser à l'expliciter.

Montrons maintenant que l'**invariant est toujours vrai**. On utilise aussi une récurrence sur le nombre p de passage dans la boucle.

Avant le premier passage, on a $\sum_{j=0}^{i-1} j = \sum_{j=0}^{0} j = 0$ et on a $somme = 0$. Donc I_0 est vrai.

Supposons I_p vrai. On a donc : $somme = \sum_{j=0}^{i-1} j$

Montrons que I_{p+1} est vrai. Si on parle de I_{p+1}, c'est qu'on passe une nouvelle fois dans la boucle. On a donc : $i \geq n$.

Les valeurs de n, i et somme deviennent :

$s' = s + i$

$i' = i + 1$

$n' = n$

$s' = \sum_{j=0}^{i-1} j + i$ par hypothèse de récurrence. $s' = \sum_{j=0}^{i} j$ $s' = \sum_{j=0}^{(i+1)-1} j$ $s' = \sum_{j=0}^{i'-1} j$

Donc on a bien I_{p+1}.

Par récurrence, on montre donc que l'invariant est toujours vrai.

On en déduit alors l'algorithme suivant :

```
Algorithme somme_n

    -- Afficher la somme des n premiers entiers

Variable
    n: Entier   -- valeur lue au clavier
    i: Entier   -- parcourir les entiers de 1 à n
    somme: Entier       -- somme des entiers de 0 à i

Début
    -- Saisir la valeur de n (pas de contrôle)
    Écrire("Nombre d'entiers : ");
    Lire(n)

    -- Calculer la somme des n premiers entiers
    somme <- 0
    i <- 1
    TantQue i <= n Faire
        { Variant : n - i + 1 }
        { Invariant : somme = ∑_{j=0}^{i-1} j }
        somme <- somme + i
        i <- i + 1
    FinTQ

    -- Afficher la somme
    Écrire("La somme est : ")
    Écrire(somme)
Fin.
```

Exercice 16 : 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.

16.1 Utiliser un **TantQue** pour réaliser la saisie contrôlée.

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

Solution : Voici l'algorithme pour le cas d'une saisie quelconque.

```
R0 : Réaliser une saisie avec vérification

R1 : Raffinage De « Réaliser une saisie avec vérification »
     |  Saisir les données
     |  Traiter les erreurs éventuelles

R2 : Raffinage De « Traiter les erreurs éventuelles »
     |  TantQue saisie incorrecte Faire
     |   |  Signaler l'erreur de saisie
     |   |  Saisir les nouvelles données
     |  FinTQ

-- Remarque : je pense que le raffinage R1 n'apporte pas suffisament
```

```
14 -- d'information.  Il serait donc préférable de ne faire qu'un seul
15 -- niveau de raffinage avec les R1 et R2 ci-dessus
```

Théorème : Tout algorithme (calculable) peut être exprimé à l'aide de l'affectation et des trois structures **`Si Alors FinSi`**, **`TantQue`** et enchaînement séquentiel.

... Mais ce n'est pas forcément pratique, aussi des structures supplémentaires ont été introduites.

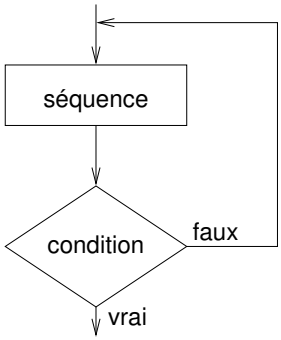
9.3.2 Répétition Répéter ... JusquÀ

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

Remarques :

— la condition n'est évaluée qu'après l'exécution de la séquence ;
— la séquence est exécutée au moins une fois ;
— la condition doit être modifiée par la séquence ;

Organigramme :



Exercice 17 : 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.

Solution : Le raffinage peut être décrit ainsi.

```
1  R0 : Afficher la somme des n premiers entiers avec possibilité de recommencer
2
3  R1 : Raffinage De « R0 »
4    | Répéter
5    |   | Afficher la somme des n premiers entiers
6    |   | Demander si l'utilisateur veut recommencer       reponse: out
7    | JusquÀ réponse est non
```

Le raffinage de « Afficher la somme des n premiers entiers » a déjà été donné dans un exercice précédent. On peut donc directement en déduire l'algorithme.

```
1  Algorithme sommes
2      -- Afficher la somme des n premiers entiers avec possibilité de recommencer
```

```
Variables
    n: Entier   -- un entier saisi au clavier
    s: Entier   -- la somme des n premiers entiers
    i: Entier   -- parcourir les entiers de 1 à n
    réponse: Caractère  -- réponse lue au clavier
Début
    Répéter
        -- Afficher la somme des n premiers entiers

        --   saisir n (il faudrait la contrôler !)
        Écrire("Valeur de n = ")
        Lire(n)

        --   calculer la somme des n premiers entiers
        somme <- 0
        i <- 1
        TantQue i <= n Faire
            { Variant : n - i + 1 }
            { Invariant : somme = \sum_{j=1}^{i-1} j }
            somme <- somme + i
            i <- i + 1
        FinTQ

        --   afficher le résultat
        ÉcrireLn("La somme des entiers est : ", somme);

        -- Demander si l'utilisateur veut recommencer
        Écrire("Encore (o/n) ? ")
        Lire(réponse)

    JusquÀ (réponse = 'n') Ou (réponse = 'N')
Fin.
```

Exercice 18 : 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.

Solution :

```
R1 : Raffinage De « Faire une saisie avec vérification »
     |  Répéter
     |   |  Saisir les données
     |   |  Si saisie incorrecte Alors
     |   |   |  Signaler l'erreur de saisie
     |   |   |  Indiquer qu'une nouvelle saisie doit être réalisée
     |   |  FinSi
     |  JusquÀ saisie correcte
```

Exercice 19 : TantQue et Répéter
Écrire la répétition **Répéter** à partir du **TantQue** et réciproquement.
Solution : Écrire un **Répéter** en utilisant le **TantQue** :

```
Répéter
    séquence
JusquÀ condition
```

devient :

```
séquence
TantQue Non condition Faire
    séquence
FinTQ
```

Inversement,

```
TantQue condition Faire
    séquence
FinTQ
```

devient :

```
Si condition Alors
    Répéter
        séquence
    JusquÀ Non condition
FinSi
```

9.3.3 Répétition Pour

```
    Pour var <- val_min [ Décrémenter ] JusquÀ var = val_max Faire
        sequence
    FinPour

    -- Une variante
    Pour Chaque var Dans val_min..val_max [ Renversé ] Faire
        sequence
    FinPour
```

Règle :
- La variable `var` est une variable d'un type scalaire. Elle est dite *variable de contrôle*.
- Les expressions `val_min` et `val_max` sont d'un type compatible avec celui de `var`.
- La séquence d'instructions ne doit pas modifier la valeur de la variable `var`.

Évaluation : Les expressions `val_min` et `val_max` sont évaluées. La variable `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.
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 (intervalle vide : `val_min > val_max`).

— La séquence termine nécessairement (le variant est `val_max - var + 1`).

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

Remarque : Il n'y a pas d'équivalent du **`Pour`** dans les organigrammes. On peut utiliser la traduction du **`Pour`** sous forme de **`TantQue`** ou de **`Répéter`**.

Exercice 20 : Somme des premiers entiers

Calculer la somme des n premiers entiers.

Solution :

```
1  Algorithme somme_n
2      -- calculer la somme des n premiers entiers
3  Variables
4      n: Entier   -- valeur lue au clavier
5      i: Entier   -- parcourir les entiers de 1 à n
6      somme: Entier       -- somme des entiers de 0 à i
7  Début
8      -- saisir la valeur de n (pas de contrôle)
9      Écrire("Nombre d'entiers : ");
10     Lire(n)
11
12     -- calculer la somme des n premiers entiers
13     somme <- 0
14     Pour i <- 1 JusquÀ i = n Faire
15         { Variant : n - i + 1 }
16         { Invariant : somme = ∑_{j=1}^{i} j }
17         somme <- somme + i
18     FinPour
19
20     -- afficher la somme
21     Écrire("La somme est : ")
22     Écrire(somme)
23 Fin.
```

9.3.4 Quelle répétition choisir ?

La première question à se poser est : « Est-ce que je connais a priori le nombre d'itérations à effectuer ? ». Dans l'affirmative, on choisit la boucle **`Pour`**.

Dans la négative, on peut généralement employer soit un **`TantQue`**, soit un **`Répéter`**. En général, utiliser un **`Répéter`** implique de dupliquer une condition et utiliser un **`TantQue`** implique de dupliquer une instruction.

Dans le cas, où on choisit d'utiliser un **`Répéter`**, il faut faire attention que les instructions qu'il contrôle seront exécutées au moins une fois. S'il existe des cas où la séquence d'instructions ne doit pas être exécutée, alors il faut utiliser un **`TantQue`** (ou protéger le **`Répéter`** par un **`Si`**).

Une heuristique pour choisir entre **`TantQue`** et **`Répéter`** est de se demander combien de fois on fait l'itération. Si c'est au moins une fois alors on peut utiliser un **`Répéter`** sinon on préfèrera un **`TantQue`**.

Remarque : Pour aider au choix, il est parfois judicieux de se poser les questions suivantes :

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

```
1  R1 : Comment { Quelle répétition choisir ? }
2           Si nombre d'itérations connu Alors
3               Résultat <- Pour
4           Sinon
5               Si itération exécutée au moins une fois Alors
6                   Résultat <- Répéter
7               Sinon
8                   Résultat <- TantQue
9               FinSi
10          FinSi
```