Support de cours :

INFORMATIQUE

Dr. Abdelouahab Belazoui

Maître de Conférences B en Informatique
Département de Pharmacie
Faculté de Médecine – Université de Batna 2
a.belazoui@univ-batna2.dz

Année universitaire : **2019 - 2020**

Table des matières

CHAPITRE 1 – INITIATION A L'INFORMATIQUE 5

1.1 INTRODUCTION 5

1.2 DESCRIPTION D'UN ORDINATEUR 5

1.2.1 De quoi est composée une machine ? 6

1.2.2 Brancher son ordinateur 7

1.2.3 Principe de fonctionnement d'un ordinateur 8

1.2.3.1 Les programmes 8

1.2.3.2 La mémoire principale 8

1.2.3.3 Le processeur central 9

1.3 INSTRUCTIONS DE BASE D'UN ORDINATEUR 9

1.4 DIFFERENTES PHASES DE RESOLUTION D'UN PROBLEME PAR ORDINATEUR 10

1.4.1 Phase d'étude 11

1.4.2 Phase de réalisation du modèle 11

1.4.3 Phase de spécification 11

1.4.4 Phase de traduction 12

1.5 CONCLUSION 12

CHAPITRE 2 – NOTIONS D'ALGORITHME 13

2.1 DEFINITION 13

2.2 CARACTERISTIQUES D'UN ALGORITHME 14

2.3 DEFINITION D'UNE VARIABLE ET DE SES CARACTERISTIQUES 15

2.3.1 Opérateur, opérande et expression 15

2.3.2 Opérateurs numériques 15

2.3.3 Opérateurs booléens 16

2.4 ACTIONS DE BASE 16
2.4.1 Action de Lecture 17
2.4.2 Action d'écriture (affichage) 17
2.4.3 Action d'affectation 18
2.4.4 Exercices 19
2.5 TESTS 21
2.5.1 Conditions 21
2.5.2 Actions conditionnelles 22
2.5.3 Actions alternatives 23
2.5.4 Tests imbriqués 24
2.5.5 Conditions composées 27
2.6 BOUCLES 28
2.6.1 Boucle Tantque 28
2.6.2 Boucle Pour 30
2.6.3 Boucle Repeter 31

CHAPITRE 3 – FONCTIONS & PROCEDURES 34
3.1 DEFINITIONS 34
3.2 TYPES DE SOUS-ALGORITHME 35
3.2.1 Procédure 35
3.2.2 Fonction 36
3.3 MODE DE PASSAGES DE PARAMETRES 38
3.3.1 Passage paramètres par valeur 38
3.3.2 Passage paramètres par variable 39
3.4 EXEMPLES 40

CHAPITRE 4 – STRUCTURES DE DONNEES STATIQUES 42
4.1 LE TYPE ARRAY 42
4.1.1 Principe 42
4.1.2 Contrôle des bornes 43
4.1.3 Recopie 44
4.2 SUPER TABLEAUX 45
4.2.1 Tableaux à plusieurs dimensions 45
4.2.2 Tableaux de record 46
4.3 LE TYPE STRING 48
4.3.1 Principe 48

4.3.2 Opérateurs sur les strings 48

CHAPITRE 5 – LANGAGES & PROGRAMMATION 50

5.1 QU'EST CE QU'UN LANGAGE DE PROGRAMMATION ? 50

5.2 BREF HISTORIQUE 50

5.3 CLASSIFICATION 51

5.3.1 Langage interprété 51

5.3.2 Langage compilé 51

5.3.3 Langages intermédiaires 52

5.4 ÉTUDE DE CAS : LANGAGE PASCAL 52

5.4.1 Structure générale des programmes Pascal 52

5.4.2 Types, variables, opérateurs 53

5.4.3 L'instruction d'affectation 55

5.4.4 Les instructions d'Entrées-Sorties 55

5.5 EXERCICES CORRIGES 57

5.5.1 Exercices 57

5.5.2 Solutions 58

BIBLIOGRAPHIE 62

Chapitre 1 – Initiation à l'informatique

1.1 Introduction

L'informatique, contraction d'information et automatique, est la science du traitement de l'information. Apparue au milieu du 20ème siècle, elle a connu une évolution extrêmement rapide. A sa motivation initiale, qui était de faciliter et d'accélérer le calcul, se sont ajoutées de nombreuses fonctionnalités, comme l'automatisation, le contrôle et la commande de processus, la communication ou le partage de l'information. L'ordinateur est, pour l'instant, la principale icône de l'informatique. Depuis l'arrivée de l'ordinateur, on a essayé, sans cesse, d'exploiter au maximum cette machine et dans tous les domaines : industrie, gestion, calculs scientifiques et techniques, enseignement, etc.

1.2 Description d'un ordinateur

Un ordinateur est une **machine automatique** commandée par des **programmes** enregistrés dans sa mémoire. Il est capable d'effectuer des opérations variées, sur les données proposées, à une grande vitesse, sans risque d'erreur (à condition que les programmes soient corrects). L'utilisateur fournit des données, l'ordinateur effectue sur ces données les traitements pour rendre des résultats.

La partie matérielle (communément nommée Hardware) est composée de pièces internes (carte mère, processeur, carte graphique, etc…..) et d'autres externes (dits périphériques) alors que le software est l'ensemble des programmes qui restent immatériels même s'ils sont stockés physiquement sur des supports mémoires.

1.2.1 De quoi est composée une machine ?

Les éléments de base composant un ordinateur sont :

1. Un écran pour que votre ordinateur vous parle;
2. Un clavier, pour que vous puissiez écrire à votre ordinateur;
3. Une souris, pour déplacer le curseur à l'écran pour parler à votre ordinateur un langage de signe;
4. Des enceintes pour le son (ce n'est pas obligatoire mais tout de même mieux) pour que vous entendiez ce que vous dit votre ordinateur;
5. Et surtout : une unité centrale[1] qui est le cœur et le cerveau de l'ordinateur.

Figure 1. Un ordinateur et ses différents périphériques

A cette composition de base, il est possible d'y ajouter divers appareils électroniques qui assurent diverses fonctionnalités (Voir figure 1). C'est ce que l'on appelle les périphériques ; tels que :

1. Imprimante pour que votre ordinateur vous écrive des choses;
2. Scanner pour que vous puissiez envoyer des photos à votre ordinateur;
3. Webcam pour que votre ordinateur puisse vous voir;
4. Disque dur externe, clé USB, carte mémoire... pour que votre ordinateur puisse garder en mémoire beaucoup de choses;
5. Manette de jeu pour que vous puissiez jouer avec votre ordinateur… etc.

[1] L'unité centrale est le boitier contenant tout le matériel électronique permettant à l'ordinateur de fonctionner. Les périphériques y sont reliés (C'est dans l'unité centrale que l'on insère un disque par exemple.)

1.2.2 Brancher son ordinateur

Chaque branchement a une forme et une couleur bien définie, ce qui fait qu'il est presque impossible de se tromper. De plus à l'heure actuelle, la plupart des périphériques d'un ordinateur se branchent tous via un branchement universel : le port USB[2].

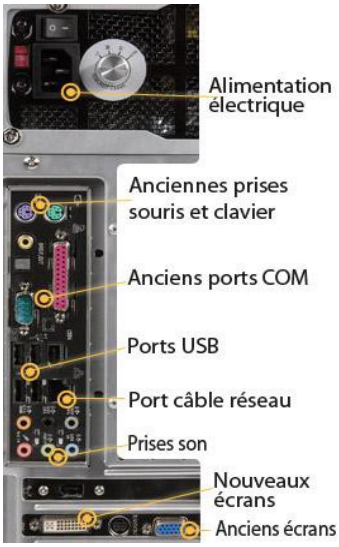


Figure 2. L'arrière d'une unité centrale

Dans l'unité centrale, chaque branchement est indiqué par une couleur (voir figure 2) :

- L'alimentation électrique, qui est reliée directement à une prise secteur. Un bouton 0 - 1 permet de couper l'arriver du courant;
- Les anciennes prises pour clavier et souris, rondes et vertes ou violettes;
- Les anciens ports COM et parallèles, qui ne sont plus utilisés de nos jours;
- Les ports USB, au nombre de 4 sur la photo, permettant de brancher divers périphériques. Ce sont actuellement les ports les plus utilisés !
- Le port pour brancher l'ordinateur à Internet ou sur un réseau;
- Les prises son : pour brancher enceintes, caisson de basses, micro;
- Un branchement DVI (blanc rectangulaire) pour brancher les nouveaux écrans et un VGA (bleu rectangulaire) pour les anciens écrans.

[2] USB est l'acronyme de « Universal Serial Bus » en anglais. C'est un branchement rectangulaire qui se veut universel : presque tout le matériel actuel se branche via USB à votre ordinateur. Les ordinateurs possèdent maintenant des ports USB à l'arrière comme à l'avant de l'unité centrale, mais aussi parfois sur votre écran.

Ces branchements peuvent varier d'un ordinateur à l'autre selon son ancienneté. On retrouvera presque toujours par contre les ports USB.

1.2.3 Principe de fonctionnement d'un ordinateur

Les deux principaux constituants d'un ordinateur sont la mémoire principale et le processeur. Alors que la mémoire principale permet de stocker des informations (programmes et données), le processeur exécute pas à pas les instructions qui composent les programmes.

1.2.3.1 Les programmes

Un programme est une suite d'instructions élémentaires qui vont être exécutée dans l'ordre. Ces instructions correspondent à des actions très simples, comme additionner deux nombres,

Un ordinateur a obligatoirement un programme (appelé système d'exploitation) qui permet de gérer son fonctionnement depuis sa mise sous tension (alimentation en électricité) jusqu'à son extinction. D'autres programmes (dits aussi logiciels) répondent aux différents besoins des utilisateurs que ce soient des besoins de calculs (calculer la paie des employés), d'impression (saisir, mettre en page et imprimer un livre), de divertissement (lecteur vidéo, jeux), de navigation, etc.

1.2.3.2 La mémoire principale

Elle contient les instructions du ou des programmes en cours d'exécution et les données associées à ce(s) programme(s). Physiquement, elle se décompose souvent en :

- une mémoire morte (ROM = Read Only Memory) chargée de stocker le programme. C'est une mémoire à lecture seule.
- une mémoire vive (RAM = Random Access Memory) chargée de stocker les données intermédiaires ou les résultats de calculs. On peut lire ou écrire des données dedans, ces données sont perdues à la mise hors tension.

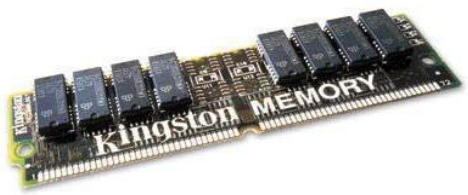


Figure 3. Un aperçu d'une RAM

Note : Les disques durs, clés USB, CDROM, etc. Sont des périphériques de stockage et sont considérés comme des mémoires secondaires.

Une mémoire peut être représentée comme une armoire de rangement constituée de différents tiroirs. Chaque tiroir représente alors une case mémoire qui peut contenir un seul élément : des données. Le nombre de cases mémoires pouvant être très élevé, il est alors nécessaire de pouvoir les identifier par un numéro. Ce numéro est appelé adresse. Chaque donnée devient alors accessible grâce à son adresse.

1.2.3.3 Le processeur central

Le processeur est parfois appelé CPU (de l'anglais Central Processing Unit) ou encore MPU (Micro Processing Unit) pour les microprocesseurs. Un microprocesseur n'est rien d'autre qu'un processeur dont tous les constituants sont réunis sur la même puce électronique (pastille de silicium), afin de réduire les coûts de fabrication et d'augmenter la vitesse de traitement. Les microordinateurs sont tous équipés de microprocesseurs. L'architecture de base des processeurs équipant les gros ordinateurs est la même que celle des microprocesseurs.

1.3 Instructions de base d'un ordinateur

Un ordinateur contient un circuit, le processeur, qui permet d'effectuer de petits traitements de base qu'on appelle instructions et qui sont la base de tout ce qu'on trouve sur un ordinateur.

En effet, contrairement à une calculatrice, dont le rôle se limite à réaliser des opérations de calcul (le plus souvent arithmétiques), un ordinateur assure des opérations de traitement de l'information, c'est-à- dire qu'il exécute successivement des opérations en suivant les directives d'un algorithme. Ce traitement est mené à l'aide d'instructions plus ou moins sophistiquées, et plus ou moins proches du microprocesseur.

Une instruction informatique est une commande unique, représentée par un symbole (numérique ou alphanumérique), et dont la finalité est prédéfinie: de la plus simple (déplacer l'index du microprocesseur dans la mémoire, additionner deux nombres) à la plus sophistiquée et abstraite (par exemple les instructions de gestion de classes du langage Java).

Une instruction est l'opération élémentaire que le processeur peut accomplir. Les instructions sont stockées dans la mémoire principale, en vue d'être traitée par le processeur. Une instruction est composée de deux champs :

- Le code opération, représentant l'action que le processeur doit accomplir;
- Le code opérande, définissant les paramètres de l'action. Le code opérande dépend de l'opération. Il peut s'agir d'une donnée ou bien d'une adresse mémoire.

Les instructions peuvent être classées en catégories dont les principales sont :

- Accès à la mémoire : des accès à la mémoire ou transferts de données entre registres.
- Opérations arithmétiques : opérations telles que les additions, soustractions, divisions ou multiplication.
- Opérations logiques : opérations ET, OU, NON, NON exclusif, etc.
- Contrôle : contrôles de séquence, branchements conditionnels, etc.

On appelle jeu d'instructions l'ensemble des opérations élémentaires qu'un processeur peut accomplir. Le jeu d'instruction d'un processeur détermine ainsi son architecture, sachant qu'une même architecture peut aboutir à des implémentations différentes selon les constructeurs. Le processeur travaille effectivement grâce à un nombre limité de fonctions, directement câblées sur les circuits électroniques. La plupart des opérations peuvent être réalisées à l'aide de fonctions basiques. Certaines architectures incluent néanmoins des fonctions évoluées courantes dans le processeur.

Le module architecture des ordinateurs vous donnera plus d'information sur les instructions et leur exécution.

1.4 Différentes phases de résolution d'un problème par ordinateur

Un programme informatique n'est pas un élément de résolution en lui-même. Ce n'est qu'un automate matérialisant un schéma de résolution pour produire les grandeurs cherchées quand on lui injecte les grandeurs connues. Le travail de résolution du problème est donc totalement à la charge du programmeur qui doit transformer une méthode en un composant opérationnel qui est le **programme**.

Parlons plus clairement ! La résolution informatique d'un problème comporte plusieurs phases préliminaires à l'exécution du programme, lesquelles sont présentées ci-dessous.

1.4.1 Phase d'étude

Cette phase sert à inventorier ce qui est connu ou observable et ce qui est à connaître. On identifie ensuite les relations entre les grandeurs connues et les grandeurs à connaître, ce qui détermine le **modèle**.

1.4.2 Phase de réalisation du modèle

Le fait d'avoir un modèle n'implique pas forcément qu'on dispose d'une méthode pour le réaliser. Le travail de réalisation du modèle consiste à déterminer un enchaînement d'opérations produisant les grandeurs cherchées à partir des grandeurs connues, en respectant le modèle. Cet enchaînement d'actions constitue en fait le **schéma de résolution**.

En l'absence du schéma de résolution pour un modèle donné, on essaie de simplifier ce dernier en le remplaçant par un modèle approché pour lequel on dispose d'une méthode de résolution.

Note : les deux phases précédentes constituent la **phase d'Analyse**.

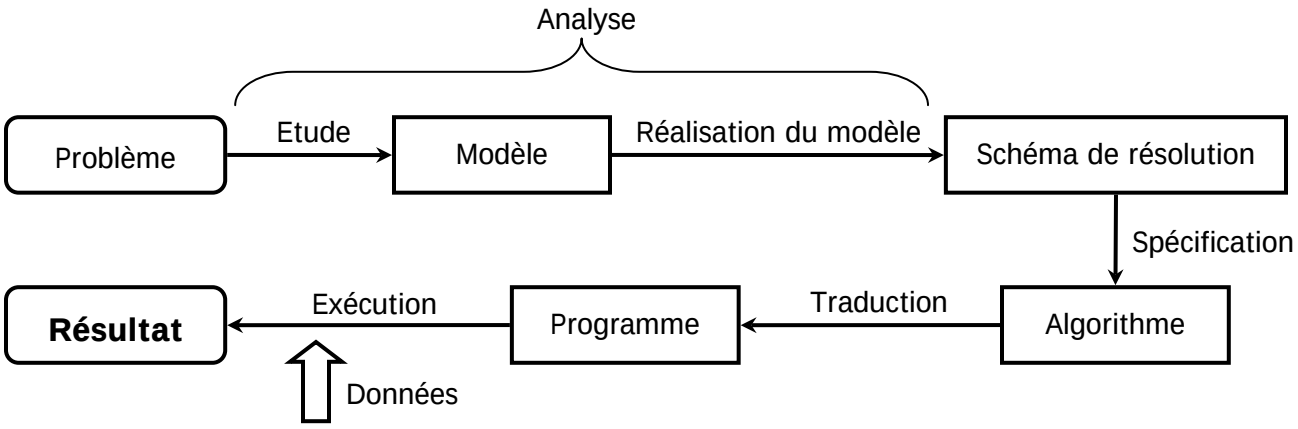


Figure 4. Les phases de résolution d'un problème

Le schéma de résolution est souvent exprimé d'une manière pas assez claire ou il est d'un niveau sémantique trop élevé par rapport aux opérations informatiques. Il convient alors de l'exprimer d'une manière précise et complète en termes informatiques. Cette phase de spécification produit les **algorithmes** (descriptions des traitements) et les **descriptions de données**. A ce niveau, on dispose d'un processus informatique de résolution sans lien avec une machine particulière ; ce processus n'est donc pas opérationnel.

1.4.4 Phase de traduction

Pour la mise en œuvre effective du processus informatique de résolution sur une machine particulière, on doit traduire les algorithmes et les descriptions de données dans un langage de programmation disponible sur cette machine. On obtient alors un programme directement interprétable ou compilable, puis exécutable sur cette machine.

1.5 Conclusion

Ce chapitre bien que comportant beaucoup de détail est nécessaire pour la bonne compréhension de l'algorithmique et de ses bases qui seront détaillés dans les chapitres suivants.

Chapitre 2 – Notions d'algorithme

2.1 Définition

Le mot **Algorithme** est un dérivé du nom de l'illustre savant musulman Muhammad Ibn Musa Al Khawarizmi qui vécut au 9ème siècle (2ème Hidjri), sous le règne du calife abbasside Al- Mamun.

Al Khawarizmi a exposé les méthodes de base pour l'addition, la multiplication, la division, l'extraction de racines carrées ainsi que le calcul des décimales de π. Ces méthodes sont précises, sans ambiguïté, mécaniques, efficaces, correctes. **Ces méthodes sont des algorithmes!**

Rappelons que les étapes de résolution d'un problème quelconque sont :

1. Comprendre l'énoncé du problème
2. Décomposer le problème en sous-problèmes plus simples à résoudre
3. Associer à chaque sous problème, une spécification :
   - Les données nécessaires ;
   - Les données résultantes ;
   - La démarche à suivre pour arriver au résultat en partant d'un ensemble de données.
4. Élaboration d'un plan d'actions (algorithme)

Informatiser une application, facturation de la consommation d'eau par exemple, c'est faire réaliser par un ordinateur, une tâche qui était réalisée par l'Homme. Pour cela il faut tout d'abord, détailler suffisamment les étapes de résolution du problème, pour qu'elles soient exécutables par l'homme.

Ensuite, transférer la résolution en une suite d'étapes élémentaires et simples à exécuter. Toute suite d'étapes, si élémentaires et simples à exécuter, s'appelle un ALGORITHME.

2.2 Caractéristiques d'un algorithme

L'algorithme est un moyen pour le programmeur de présenter son approche du problème à d'autres personnes. En effet, un algorithme est l'énoncé dans un langage bien défini d'une suite d'opérations permettant de répondre au problème. Un algorithme doit donc être :

- **Lisible :** l'algorithme doit être compréhensible même par un non-informaticien
- **De haut niveau :** l'algorithme doit pouvoir être traduit en n'importe quel langage de programmation, il ne doit donc pas faire appel à des notions techniques relatives à un programme particulier ou bien à un système d'exploitation donné
- **Précis :** chaque élément de l'algorithme ne doit pas porter à confusion, il est donc important de lever toute ambiguïté
- **Concis :** un algorithme ne doit pas dépasser une page. Si c'est le cas, il faut décomposer le problème en plusieurs sous-problèmes
- **Structuré :** un algorithme doit être composé de différentes parties facilement identifiables à savoir : une entête où apparaissent le nom de l'algorithme ainsi que les déclarations des objets manipulés puis le corps de l'algorithme qui commence par "Début" et se termine par "Fin. Entre ces deux mots clés se trouvent les actions ordonnées à exécuter par la machine.

```
Algorithme Nom_de_l'algorithme;        ┐
Variables                              ├ Entête
  Liste des variables;                 ┘
Début                                  ┐
  Action 1;                            │
  Action 2;                            │
  Action 3;                            ├ Corps
  ….                                   │
  Action n;                            │
Fin.                                   ┘
```

Les données dont on dispose au début, les résultats que l'on doit obtenir à la fin ainsi que les objets internes utilisés pour arriver à ces résultats sont ceux déclarés dans l'entête alors que les actions du corps lorsqu'elles sont appliquées dans l'ordre permettent d'accéder au résultat attendu.

Note :

- Chaque mot clé est souligné.
- Une marque de terminaison (;) est utilisée après chaque action suivie d'une autre action.

2.3 Définition d'une variable et de ses caractéristiques

Une variable est une entité qui contient une information. Elle possède :

- Un nom, on parle d'identifiant;
- Un type qui caractérise l'ensemble des valeurs que peut prendre la variable :
  - Nombre Entier
  - Nombre flottant (Réel)
  - Booléen (avec pou seules valeurs Vrai et Faux)
  - Caractère (alphabétique, numérique)
  - Chaine de caractères (mot ou phrase)

A un type donné, correspond un ensemble d'opérations définies pour ce type.

2.3.1 Opérateur, opérande et expression

- Un opérateur est un symbole d'opération qui permet d'agir sur des variables ou de faire des "calculs"
- Un opérande est une entité (variable, constante ou expression) utilisée par un opérateur
- Une expression est une combinaison d'opérateur(s) et d'opérande(s), elle est évaluée durant l'exécution de l'algorithme, et possède une valeur (son interprétation) et un type.

2.3.2 Opérateurs numériques

- On retrouve tout naturellement les opérations usuelles : +, -, *, /

- Avec en plus pour les entiers div et mod, qui permettent respectivement de calculer une division entière et le reste de cette division.
- L'opérateur d'égalité, que l'on retrouve chez tous les types simples, permet de savoir si les deux opérandes sont égaux. Il est représenté par le caractère = alors que l'opérateur d'inégalité est représenté par ≠. Et pour les types avec des valeurs ordonnées, il existe des opérateurs de comparaison <, ≤, >, ≥

2.3.3 Opérateurs booléens

- Les opérateurs sont essentiellement : Non, Et, Ou et Ou-Exclusif
- Associativité, Commutativité et Distributivité sont les propriétés des opérateurs "Et" et "Ou"

Note :

- Tout comme en arithmétique les opérateurs ont des priorités. Par exemple
  - * et / sont prioritaires sur + et -
  - Pour les booléens, la priorité des opérateurs est : Non, Et, ou-Exclusif et ou
- Pour clarifier les choses (ou pour supprimer toute ambiguïté) on peut utiliser des parenthèses.

2.4 Actions de base

Afin de résoudre un problème quelconque il est impératif de (1) préciser les données ; (2) réaliser le traitement ; (3) Afficher les résultats. En effet, pour réaliser une opération d'addition « A+ B » il est nécessaire de connaitre, au départ, les valeurs des données A et B ; puis réaliser le calcul et à la fin donner le résultat de la somme.

Trois actions sont donc essentielles afin d'écrire des algorithmes de traitement :

- Deux actions qui permettent d'établir la communication entre l'être humain et la machine : Lire / Ecrire
- Une action qui permet à la machine de réaliser des manipulations et des calculs : Affectation.

2.4.1 Action de Lecture

Pourquoi ?

Pour transmettre **les valeurs des données** de chez l'utilisateur (l'être humain) vers la machine.

Lecture : Utilisateur →Machine

Syntaxe :

- Lecture d'une seule variable : **Lire (Nom-variable)** ;
- Lecture de plusieurs (n) variables : **Lire (Nom-variable-1, Nom-variable-2, …, Nom-variable-n)** ;

Exemple :

Afin de résoudre une équation du second degré de la forme a X2 + b X + C = 0 les valeurs des coefficients a, b et c connues par l'utilisateur doivent être transmises à l'ordinateur avant que la machine ne commence le calcul ; à savoir le calcul du discriminant delta.

Le seul moyen d'établir cette communication est le groupe d'instructions :

```
Lire (a);
Lire (b);
Lire (c);
```

Qui peut être regroupé dans une seule instruction :

```
Lire (a, b, c);
```

2.4.2 Action d'écriture (affichage)

Pourquoi ?

Pour transmettre **les valeurs des résultats** calculés par la machine vers l'utilisateur (l'être- humain) ; ou encore transmettre un message textuel.

Écriture : Machine →Utilisateur

Syntaxe :

- Affichage d'une seule variable : **Ecrire (Nom-variable)**;
- Affichage de plusieurs (n) variables : **Ecrire (Nom-variable-1, …, Nom-variable-n)**;
- Affichage d'un message : **Ecrire ('Texte')**;

Exemple :

La résolution d'une équation de second degré dans les réels donne lieu soit à des valeurs de X1 et X2 ou à un texte dans le cas où aucune valeur ne peut être calculée :

- Dans le cas ou delta est supérieur ou égal à zéro : **Ecrire (X1, X2)**;
- Dans le cas ou delta est inférieur à zéro et donc pas de solution dans R : **Ecrire ('pas de solution')**.

2.4.3 Action d'affectation

Pourquoi ?

Pour donner une nouvelle valeur (contenu) à une variable ; cette valeur peut être un résultat de calcul.

Affectation : Machine ← Machine

Syntaxe :

- Affectation d'une valeur à une variable : **Nom-variable ← valeur**;
- Affectation d'une variable à une variable : **Nom-variable1 ← Nom-variable2**;
- Affectation du résultat de calcul à une variable :

  Nom-variable ← nom-variable1 opérateur nom-variable2;
- Ou encore l'affectation du résultat de plusieurs calcul à une variable :

  Nom-variable ← nom-variable1 opérateur1 nom-variable2 … opérateur-n nom-variable n;

Exemple :

La résolution d'une équation de second degré se fait par calcul du discriminant delta :

```
delta ← b*b – 4*a*c;
```

Dans le cas de delta > 0 la machine réalise le calcul des deux solutions X1 et X2 en exécutant les instructions :

```
X1 ← -b + √delta/ 2*a;
X1 ← -b - √delta/ 2*a;
```

2.4.4 Exercices

Exercice 1 : écrire un algorithme qui permet de multiplier un nombre par 2, lui ajouter 7, soustrait le nombre de départ puis afficher le résultat.

```
  Algorithme Exo1;
  Déclaration
     x, y : entier;
  Début
  /* il faut bien connaître la valeur de départ sinon comment faire les
calculs demandés !*/
     Lire (x);
     y ← x*2;
     y ← y + 7;
     y ← y – x;
     Écrire (y);
  /* si on n'affiche pas le résultat sur écran on ne le connaîtra jamais*/
  Fin.
```

Exécution de cet algorithme pour la valeur de x = 2 :

x	y	Écran
2		
	4	
	11	
	9	
		9

Exécution de cet algorithme pour la valeur de x = 10 :

x	y	Écran
10		
	20	
	27	
	17	
		17

Exercice 2 : soit l'algorithme suivant :

```
Algorithme Exo2;
Déclaration
   x, y, z : réel;
Début
   Lire (x, y);
   z ← x;
   x ← y;
   y ← z;
   Écrire (x, y);
Fin.
```

Quel est le rôle de cet algorithme ?

Pour répondre à cette question il est nécessaire d'exécuter l'algorithme.

Exécution de cet algorithme pour les valeurs **x = 2.5** et **y = 6.0**

x	y	z	Écran
2.5	6.0		
		2.5	
6.0			
	2.5		
			6.0 - 2.5

Le résultat est **x = 6.0** et **y = 2.5**

Il parait que cet algorithme permet d'échanger les valeurs de deux réels en utilisant une variable intermédiaire z. Pour vérifier ce rôle il est impératif de faire au moins une autre exécution avec d'autres valeurs. Prenons **x = 101.67** et **y = -18.3**

x	y	z	Écran
101.67	-18.3		
		101.67	
-18.3			
	101.67		
			-18.3 – 101.67

Le résultat est **x = -18.3** et **y = 101.67**

Exercice 3 : écrire un algorithme qui calcul le prix à payer pour une marchandise à partir d'un prix initial et d'une remise sur ce prix. La formule à appliquer est PaP = PI – PI*R/100

Exemple : Prix Initial (PI) = 100 da, Remise (R) = 25 → le Prix à payer (PaP) = 75 da

```
Algorithme Exo3;
Déclaration
   PI, R, PaP : réel;
Début
   Écrire ('Donnez SVP le prix initial et la remise');
   Lire (PI, R);
   PaP ← PI – PI* R / 100;
   Écrire ('Le prix à payer =', PaP);
Fin.
```

2.5 Tests

Toutes les actions ne sont pas réalisables à chaque fois. En effet, s'il pleut l'action prendre le parapluie a un sens sinon elle est annulée ou remplacée par une autre comme prendre les lunettes de soleil.

Les actions de l'algorithmique peuvent, elles aussi, dépendre de certaines conditions et il existe plusieurs formes de tests en algorithmique.

2.5.1 Conditions

Une condition est une comparaison sous la forme

Valeur <opérateur-comparaison> Valeur

Les opérateurs de comparaison sont :

- Egal (=)
- Différent (≠)
- Supérieur (>)
- Inférieur (<)
- Supérieur ou égal (≥)
- Inférieur ou égal (≤)

Note :

- La valeur peut être contenue dans une variable déclarée ou le résultat d'un calcul.
- Attention aux raccourcis du langage naturel qui peuvent regrouper dans une même phrase deux ou plusieurs opérateurs ce qui est valide en mathématique mais non en algorithmique.

Exemple âge compris entre 15 et 18 en mathématique il est correct d'écrire 15 ≤ âge ≤ 18 mais comme il y a deux (2) opérateurs (deux ≤) cela est interdit dans une même condition algorithmique.

2.5.2 Actions conditionnelles

Quand ?

Il s'agit dans ce cas de conditionner un groupe d'actions (qui peut contenir de une à plusieurs instructions) selon un certain test. Si la condition est vérifiée alors le groupe d'action est déclenché puis les actions suivantes sont exécutées. Dans le cas où la condition est fausse (non vérifiée) un saut est effectué vers les actions qui suivent le groupe conditionné.

Syntaxe

```
Si Condition Alors
   Instruction 1;
   Instruction 2;
   …
   Instruction n;
FinSi;
```

Exemple : écrire un algorithme qui calcul et affiche la valeur absolue d'une valeur entière M.

```
Algorithme Condition;
Déclaration
   M : entier;
Début
   Lire (M);
   Si M < 0 Alors
     M ← - M;
   FinSi;
   Écrire (M);
Fin.
```

- L'exécution de cet algorithme pour la valeur **x = 4** fait que les instructions **Lire (x)** et **Écrire (x)** sont les seules à être exécutées car l'évaluation de la condition x < 0 rend **faux**.
- L'exécution de cet algorithme pour la valeur **x = -4** fait que toutes les instructions de l'algorithme sont exécutées car la condition x < 0 est évaluée à **vrai**.

2.5.3 Actions alternatives

Quand ?

Il s'agit dans ce cas de conditionner deux groupes d'actions (qui peuvent contenir chacun de une à plusieurs instructions) selon un certain test. Si la condition est vérifiée alors le groupe d'action 1 est déclenché puis les actions qui suivent le FinSi sont exécutées. Dans le cas où la condition est fausse (non vérifiée) un saut est effectué vers le groupe actions 2 qui sera exécuté puis suivront les instructions qui suivent le FinSi.

Il y a donc obligatoirement une exécution de l'un ou l'autre des groupes d'action mais que si l'un est exécuté l'autre ne l'est pas.

Note : l'alternative n'admet que deux chemins (vrai ou faux)

Syntaxe

```
Si Condition Alors
   Instruction 1;          ┐
   Instruction 2;          │
   …                       ├  Groupe action 1
   Instruction n;          │
Sinon                      ┘
   Instruction 1;          ┐
   Instruction 2;          │
   …                       ├  Groupe action 2
   Instruction m;          │
FinSi;                     ┘
```

Exemple : écrire un algorithme qui calcul et affiche la nature (positive ou négative) d'un nombre entier. Le zéro est considéré comme valeur positive.

```
Algorithme Alternative;
Declaration
   X : entier ;
Debut
   Lire (x);
```

```
    Si x < 0 Alors
       Ecrire ('Le nombre est négatif')
    Sinon
       Ecrire ('Le nombre est positif')
    FinSi;
Fin.
```

L'exécution de cet algorithme pour la valeur x = 6 nous conduit à suivre le chemin avec les instructions Lire (x) et Écrire ('Le nombre est positif') de même que dans le cas de x = 0 ; alors que l'exécution avec x = -4 le chemin comprend les instructions Lire (x) et Écrire ('Le nombre est négatif')

2.5.4 Tests imbriqués

Quand ?

Dans certains cas il existe plus d'une alternative à une condition. Il s'avère alors que le test sur une valeur peut avoir plus de deux chemins possibles et des tests sont inclus dans d'autres. Cette forme est celle des tests imbriqués.

Syntaxe :

```
Si Condition 1 Alors
   Instruction 1;          ┐
   Instruction 2;          │
   …                       ├─ Groupe action 1
   Instruction n;          │
Sinon                      ┘
   Si Condition 2 Alors    ┐
      Instruction 1;       │
      …                    ├─ Groupe action 2
      Instruction m;       │
   Sinon                   ┘
      …                    ┐
      Instruction p;       ├─ Groupe action k
   FinSi;                  ┘
FinSi;
```

Prenons le cas de l'algorithme de la nature d'un nombre entier (section précédente) et que l'on considère que le zéro est une valeur qui n'est ni positive ni négative mais qu'il est déclaré nul. Ce changement implique trois alternatives et non pas deux ce qui oblige l'informaticien à la solution avec tests imbriqués comme suit :

```
Algorithme Imbriqués;
Declaration
   X : entier;
Debut
   Lire (x);
   Si x < 0 Alors
      Ecrire ('Le nombre est négatif')
   Sinon
      Si x > 0 Alors
        Ecrire ('Le nombre est positif')
      Sinon
        Ecrire ('Le nombre est une valeur nulle')
      FinSi;
   FinSi
Fin.
```

A l'exécution de cet algorithme un et un seul des trois chemins (groupe d'actions) est suivi selon que la première condition est vraie :

- Lire (x)
- Écrire ('Le nombre est négatif')

Ou si la première condition est fausse et que la deuxième est vraie :

- Lire (x)
- Écrire ('Le nombre est positif')

Ou encore si les deux premières conditions sont fausses :

- Lire (x)
- Écrire ('Le nombre est une valeur nulle')

Les tests imbriqués remplacent un ensemble d'actions conditionnelles particulièrement lorsque les conditions portent sur la (ou les) même(s) variable(s).

Note : Dans le cas où les conditions des tests imbriqués portent sur des égalités par rapport à des valeurs précises ou des intervalles, ces tests peuvent être modélisés avec le constructeur Selon qui modélise le choix multiple en suivant la syntaxe suivante :

```
Selon variable
   Valeur1 : groupe instructions 1;
   Valeur2 : groupe instructions 2;
   …
   Valeurn : groupe instructions n;
FinSelon;
```

Dans le cas ou la variable est égale à la valeur1 alors le groupe d'instruction 1 est le seul qui est exécuté ; si elle est égale à la valeur2 c'est le groupe 2 et ainsi de suite.

La syntaxe peut aussi suivre la forme suivante s'il s'agit de tester l'appartenance de la variable à des intervalles :

```
Selon variable
   Intervalle1 : groupe instructions 1;
   Intervalle2 : groupe instructions 2;
   …
   Intervalle_n : groupe instructions n;
FinSelon;
```

Note : Dans le cas ou il existe un groupe d'instructions à exécuter en dernier recours (la variable n'est égale a aucun(e) intervalle (ou valeur) cité(e)) une ligne est rajoutée en fin pour le cas autre selon la syntaxe :

```
Selon variable
   Valeur1 : groupe instructions 1;
   Valeur2 : groupe instructions 2;
   …
   Valeur_n : groupe instructions n;
   Autre : groupe instructions n+1;
FinSelon;
```

Prenons l'exemple d'un algorithme qui affiche l'action équivalente à la lumière active dans des feux de circulations.

```
Algorithme choix;
Declaration
   lumiere : chaine de caractère;
Debut
   Lire (lumiere);
   Selon lumiere
      'orange' : Ecrire ('Attention !')
      'vert' : Ecrire ('voiture départ')
      'rouge' : Ecrire ('Voiture arrêt')
```

```
    Finselon;
Fin.
```

2.5.5 Conditions composées

Quand ?

Comme déclaré précédemment (section 1) certains problèmes exigent parfois de formuler des conditions qui ne peuvent pas être exprimées sous la forme simple désignée ci-dessus. Les opérateurs Logiques (essentiellement **et**, **ou**, **non**) servent alors à composer les conditions. L'exemple du test 15 ≤ âge ≤ 18 est alors formulé par la condition :

Si (Age ≥ 15) et (Age ≤ 18) Alors

Syntaxe

```
Si (Cond. 1) op. logique 1 (Cond. 2) op. logique 2 … (Cond. n) Alors
```

- Pour l'opérateur « **et** » il est nécessaire que toutes les conditions soient vraies pour que le test soit vrai. Dans le test ci-dessous il est nécessaire que l'Age appartient à [15, 18] pour que le test soit vrai.
- Pour l'opérateur « **ou** » il suffit que l'une des conditions soit vraie pour que le test soit vrai
- Dans le test **Si (Age ≥ 15) ou (Age ≤ 18) Alors** il suffit que l'Age soit supérieur ou égal à 15 ou qu'il soit inférieur ou égal à 18 pour que le test soit vrai.
- Pour l'opérateur « **non** » si la condition est vraie le test devient faux et si elle est fausse alors le test sera évalué à vrai.
- Dans le test **Si Non (Age ≥ 15) Alors**. Ce test est vrai si l'Age est inférieur à 15 et faux dans le cas contraire.

Exemple : écrire un algorithme qui affiche si le produit de deux nombres est positif ou non sans calculer le produit.

```
Algorithme Composition1;
Declaration
   a, b : entier;
Debut
   Lire (a, b);
   Si ((a≥0) et (b≥0)) ou ((a<0) et (b<0)) Alors
      Ecrire ('Le produit est positif')
   Sinon
      Ecrire ('Le produit est négatif')
   FinSi;
Fin.
```

Une autre solution peut être la suivante :

```
Algorithme Composition2;
Declaration
   a, b : entier;
Debut
   Lire (a, b);
   Si ((a<0) et (b>0)) ou ((a>0) et (b<0)) Alors
      Ecrire ('Le produit est négatif')
   Sinon
      Ecrire ('Le produit est positif')
   FinSi;
Fin.
```

2.6 Boucles

Dans un algorithme, certaines instructions peuvent être répétées et il ne serait pas très intelligeant de les réécrire. Les boucles sont alors apparues afin de modéliser la répétition. Trois formes se présentent aux informaticiens, l'utilisation de l'une ou l'autre est en rapport avec le test de répétition bien que la plus utilisée soit celle du Tantque.

2.6.1 Boucle Tantque

Une boucle Tantque permet de répéter plusieurs fois le même bloc d'instructions tant qu'une certaine condition reste vraie. La syntaxe est la suivante :

```
Tantque Condition Faire
   Instruction 1;
   Instruction 2;
   …
   Instruction n;
FinTantque;
```

Si la condition est vraie le groupe d'instructions est exécuté puis le FinTantque remet l'exécution à la condition et ainsi de suite jusqu'à ce que la condition soit évaluée à faux.

Note :

- Le groupe d'instruction peut ne jamais être exécuté si dès le départ la condition de la boucle n'est pas satisfaite. Attention donc aux boucles qui ne sont jamais exécutées.
- A l'intérieur de la boucle il est impératif de faire des changements de manière à ce que le test soit erroné à un moment ou l'autre sinon si la condition reste toujours vraie on se retrouve dans une situation de boucle infinie qui est formellement interdite en algorithmique.

Exemple 1 : écrire un algorithme qui calcul le nombre de valeurs entières saisies pour aboutir à une somme ne dépassant pas 500.

```
Algorithme Somme;
Declaration
   x, som, co : entier;
Debut
   sim ← 0;  co ← 0;
   Tant que  som < 500 Faire
     Lire (x);
     som ← som + x;
     co ← co + 1;
   FinTantque;
   Si som > 500 Alors
     co ← co - 1;
   FinSi;
   Ecrire (co)
Fin.
```

Exemple 2 : écrire un algorithme qui affiche la division de 15 sur x avec obligation de ne jamais diviser sur le zéro.

```
Algorithme boucle1;
Declaration
   x : entier;
```

```
   y : reel;
Debut
   Lire (x);
   Tantque (x<0) Faire
      Ecrire ('Donnez un nombre positif');
      Lire (x);
   Fintantque;
   y ← 15/x;
   Ecrire (y);
Fin.
```

2.6.2 Boucle Pour

Lorsque le nombre de répétitions est connu, la boucle **Pour** utilise un compteur pour calculer le nombre de fois qu'est répété le bloc d'instructions de la boucle. La syntaxe est la suivante :

```
Pour compteur ← valeur-initiale à valeur-finale Pas = nombre Faire
   Instruction 1;
   Instruction 2;
   …
   Instruction n;
FinPour;
```

Le compteur est initialisé à la valeur-initiale puis le groupe d'instruction est exécuté. Une fois arrivée à la FinPour, celle-ci remet l'exécution à la ligne Pour en rajoutant, **automatiquement**, la valeur du pas au compteur puis sa valeur est comparée à la valeur-finale. Dans le cas où il n'a pas atteint cette valeur la boucle est encore une fois exécutée mais si l valeur est atteinte la boucle Pour signale que c'est la dernière fois qu'elle peut être exécutée et donc arrivé à FinPour le retour n'est plus permis et l'exécution continue avec les instructions qui suivent la boucle.

Note :

- Le pas, dont la valeur est entière, peut être positif (et le compteur est incrémenté par cette valeur à chaque itération) ou négatif et c'est une décrémentation.
- Si le pas = 1 il est généralement omis.
- La valeur du compteur étant automatiquement modifiée par la boucle Pour, il est strictement interdit d'écrire à l'intérieur de la boucle des instructions changeant la valeur de ce compteur.

Exemple : écrire un algorithme qui calcul la factorielle d'un nombre entier positif selon la formule :

Fact (x) = 1*2*3*….*x

```
Algorithme Factorielle;
Declaration
   x, fact, co: entier ;
Debut
   Lire (x);
   fact ← 1;
   Pour co ← 2 à x Faire
      fact ← fact * co;
   FinPour;
   Ecrire (fact);
Fin.
```

2.6.3 Boucle Repeter

La répétition en utilisant Repeter répond aux mêmes conditions que celles du Tantque sauf que la condition se trouve en fin de la boucle et non pas au début.

La différence entre les deux formes est qu'une boucle avec Tantque peut ne jamais être exécutée car la condition peut être fausse au départ alors que dans la boucle Repeter le groupe d'instructions est exécuté au moins une fois pour arriver à la condition.

La condition du Repeter est donc une condition de fin (la répétition est terminée quand la condition est vraie) alors que la condition du Tantque est une condition de répétition (la répétition est terminée quand la condition est évaluée à faux).

```
Repeter
   Instruction 1;
   Instruction 2;
   …
   Instruction n
Jusqu'à Condition;
```

Note : La boucle Repeter est particulièrement recommandée pour les saisies avec vérification.

Exemple : écrire un algorithme qui calcul la division d'un nombre réel par un entier selon la formule :

(z = x/y).

Solution avec Repeter :

```
Algorithme Division ;
Declaration
   x, z : reel;
```

```
   y : entier;
Debut
   Lire (x);
   Repeter
      Lire (y)
   Jusqu'à y ≠ 0;
   z ← x / y;
   Ecrire (z);
Fin.
```

Lors de l'exécution de cet algorithme le passage à l'instruction de calcul de z ne se fait que lorsque la saisie de la valeur de y est différente du zéro.

Une autre solution en utilisant la boucle Tantque est la suivante :

```
Algorithme Division2;
Declaration
   x, z : reel;
   y : entier;
Debut
   Lire (x, y);
   Tantque y = 0 Faire
      Lire (y)
   FinTantque;
   z ← x / y;
   Ecrire (z);
Fin.
```

L'instruction de lecture est alors écrite deux fois : la première pour tester la condition et la deuxième pour répéter la lecture jusqu'à saisie d'une valeur non nulle. Résumons maintenant avec un exemple pour lequel on donne 3 solutions pour les 3 types de boucles. Cet algorithme calcul et affiche 100 fois la somme de deux nombres entiers lus à chaque itération.

Version Tantque :

```
Algorithme boucle2;
Declaration
   x, y, co, som : entier;
Debut
   co ← 0;
   Tantque (co ≠ 100) Faire
      Lire (x, y);
      co ← co + 1;
      Som ← x + y;
      Ecrire ('la somme numéro', co, ' = ', som);
   Fintantque;
Fin.
```

Version Répéter :

```
Algorithme boucle3;
Declaration
   x, y, co, som : entier;
Debut
   co ← 0;
   Répeter
      Lire (x, y);
      co ← co + 1;
      Som ← x + y;
      Ecrire ('la somme numéro', co, ' = ', som);
   Jusqu'à co = 100;
Fin.
```

Version Pour :

```
Algorithme boucle3;
Declaration
   x, y, co, som : entier;
Debut
   Pour co ← 1 a 100 Faire
      Lire (x, y);
      Som ← x + y;
      Ecrire ('la somme numéro', co, ' = ', som);
   FinPour;
Fin.
```

Chapitre 3 – Fonctions & procédures

Dès qu'on commence à écrire des programmes importants, il devient difficile d'avoir une vision globale sur son fonctionnement et de traquer les erreurs. Par exemple, Il nous est demandé d'écrire un algorithme qui calcul la partie entière de deux nombres réels puis d'afficher la plus grande valeur des deux avant de calculer la somme et la moyenne des deux nombres entiers obtenus.

> **Que faire ?** Décomposer le problème en sous problèmes et trouver une solution à chacun puis regrouper le tout dans un seul algorithme

En Algorithmique, chaque solution partielle donne lieu à un sous-algorithme qui fera partie d'un algorithme complet pour pouvoir être exécuté.

3.1 Définitions

Un sous-algorithme est un bloc faisant partie d'un algorithme. Il est déclaré dans la partie entête (avant le début de l'algorithme) puis appelé dans le corps de l'algorithme.

Étant donné qu'il s'agit d'un bloc à part entière, il possède éventuellement un en-tête, une série de traitements, et une gestion des résultats tout comme l'algorithme qui le contient.

Note :

A Un sous-algorithme utilise les variables déclarées dans l'algorithme (appelées variables globales). Il peut aussi avoir ses propres variables (dites locales) déclarées dans l'espace qui lui est réservé; mais qui ne peuvent être utilisées que dans ce sous-algorithme et nulle part ailleurs car sa portée

(visibilité) est limitée au bloc qui la contient. L'espace de ces variables locales n'est réservé que lorsque le sous-algorithme est appelé et est libéré dès la fin de l'exécution.

B Un sous-algorithme est déclaré de manière générale c.-à-d qu'il peut être appelé plusieurs fois avec différentes valeurs grâce à des arguments. Ces derniers, bien qu'ils soient facultatifs, sont dits paramètres et sont clairement déclarés, au besoin, dans l'entête du sous-algorithme.

Un paramètre est une valeur du bloc principal dont le sous-algorithme a besoin pour exécuter avec des données réelles l'enchaînement d'actions qu'il est chargé d'effectuer. On distingue deux types de paramètres :

- Les paramètres formels sont la définition du nombre et du type de valeurs que devra recevoir le sous-algorithme pour se mettre en route avec succès. On déclare les paramètres formels pendant la déclaration du sous-algorithme.
- Les paramètres effectifs sont des valeurs réelles (constantes ou variables) reçues par le sous-algorithme au cours de l'exécution du bloc principal. On les définit indépendamment à chaque appel du sous-algorithme dans l'algorithme principal.

C L'exécution d'un sous-algorithme (procédure ou fonction) se fait par une instruction d'appel (voir sections suivantes). L'application de cette instruction génère un saut vers le sous-algorithme appelé. La terminaison de ce sous-algorithme redémarre la suite d'instruction interrompue par l'appel.

3.2 Types de sous-algorithme

Un sous-algorithme peut se présenter sous forme de fonction ou de procédure. Une fonction est un sous-algorithme qui, à partir de donnée(s), calcul et rend à l'algorithme Un et Un seul résultat alors qu'en général, une procédure affiche le(s) résultat(s) demandé(s).

3.2.1 Procédure

Une procédure est un bloc d'instructions nommé et déclaré dans l'entête de l'algorithme et appelé dans son corps à chaque fois que le programmeur en a besoin.

Déclaration d'une procédure :

```
Procédure Nom_Procédure (Nom_Paramètre : Type_praramètre;……) ;
Déclaration
   Nom_variable : Type_variable;     } Variables locales
   …
Début                                }
   …                                 }
   Instructions;                     } Corps de la procédure
   …                                 }
Fin;                                 }
```

L'appel d'une procédure peut être effectué en spécifiant, au moment souhaité, son nom et éventuellement ses paramètres ; cela déclenche l'exécution des instructions de la procédure.

Exemple : Voici un algorithme utilisant une procédure qui calcule une somme de 100 nombres.

```
Algorithme essai;
Variable
   I, S : entier;
Procedure Somme ;
Debut        /*Début de la Procédure*/
   S ← 0;
   Pour I ← 1 a 100 Faire
     S ← S + i
   FinPour;
   Ecrire ('La somme des 100 premiers nombres est', S);
Fin          /*Fin de la Procédure*/
Debut        /*Début de l'algorithme*/
   Somme
Fin.         /*Fin de l'algorithme*/
```

3.2.2 Fonction

Une fonction est un bloc d'instructions qui retourne obligatoirement une et une seule valeur résultat à l'algorithme appelant. Une fonction n'affiche jamais la réponse à l'écran car elle la renvoie simplement à l'algorithme appelant.

Déclaration d'une fonction :

```
Fonction Nom_Fonction (Nom_Paramètre: Type_praramètre;……): type_Fonction;
Déclaration
   Nom_variable: Type_variable;        } Variables locales
   …
Début
   …
   Instructions;
   …                                   } Corps de la procédure
   Nom_Fonction ← Résultat
Fin;
```

Étant donné qu'une fonction a pour but principal de renvoyer une valeur, il est donc nécessaire de préciser le type de la fonction qui est en réalité le type de cette valeur.

Un appel de fonction est une expression d'affectation de manière à ce que le résultat soit récupéré dans une variable globale : `Nom_variable-globale ← Nom_Fonction (paramètres);`

Exemple : L'algorithme précédent, qui calcule une somme de N nombres, peut utiliser une fonction au lieu d'une procédure.

```
Algorithme essai;
Variable
   I, Som : entier;
Fonction Somme: entier;
Variable
   S : entier;
Debut          /*Début de la fonction*/
   S ← 0;
   Pour I ← 1 a 100 Faire
      S ← S + I
   FinPour;
   Somme ← S
Fin            /*Fin de la Fonction */
Debut          /*Début de l'algorithme*/
   Som ← Somme;
   Ecrire ('La somme des ', N, 'premiers nombres est', Som);
Fin.           /*Fin de l'algorithme*/
```

Note : De même qu'une procédure, une fonction peut appeler d'autres sous-algorithmes à condition qu'ils soient définis avant elle ou qu'ils soient déclarés dans son entête.

3.3 Mode de passages de paramètres

Un sous-algorithme avec paramètres est très utile parce qu'il permet de répéter une série d'opérations complexes pour des valeurs qu'on ne connaît pas à l'avance. Il existe deux types de passage de paramètres : par valeur et par variable (dite aussi par référence ou encore par adresse).

3.3.1 Passage paramètres par valeur

C'est le mode de transmission par défaut, il y a **copie** de la valeur, des paramètres effectifs dans les variables locales issues des paramètres formels de la procédure ou de la fonction appelée.

Dans ce mode, le contenu des paramètres effectifs ne peut pas être modifié par les instructions de la fonction ou de la procédure ; car nous ne travaillons pas directement avec la variable, mais sur une copie. À la fin de l'exécution du sous-algorithme la variable conservera sa valeur initiale. Les paramètres dans ce cas sont utilisés comme données.

Syntaxe :

```
Procedure nom_procédure (param1 :type1 ; param2, param3 :type2);
Fonction <nom_fonction> (param1 :type1 ; param2 :type2) : Type_fonction;
```

Exemple : soit l'algorithme suivant.

```
Algorithme pas-val;
Déclaration
   M : entier;
Procedure P1 (nombre : entier);
Debut
   Si nombre < 0 Alors
      nombre ← - nombre
   FinSi;
   Ecrire (nombre)
Fin;
Debut
   Lire (M);
   P1 (M);
   Ecrire (M)
Fin.
```

Exécutons cet algorithme pour la valeur (-6)

Avant l'appel de procédure : la seule variable déclarée est la variable globale (M)

M	Écran
-6	

Après l'appel de procédure : la variable-paramètre "nombre" est déclarée et reçoit en copie la valeur de M.

M	Nombre	Écran
-6	-6	
-6	6	
-6	6	6

Au retour à l'algorithme (au niveau de l'appel) il ne reste que la variable globale avec sa valeur initiale

M	Écran
-6	
-6	-6

3.3.2 Passage paramètres par variable

Ici, il s'agit non plus d'utiliser simplement la valeur de la variable, mais également son emplacement dans la mémoire (d'où l'expression « par adresse »). En fait, le paramètre formel se substitue au paramètre effectif durant le temps d'exécution du sous-programme et à la sortie il lui transmet sa nouvelle valeur. Un tel passage de paramètre se fait par l'utilisation du mot-clé **Var**.

Syntaxe :

```
Procedure nom_procédure (Var param1: type1, param2, param3: type2);
Fonction <nom_fonction> (Var param1: type1, param2: type2): Type_fonction;
```

Note : Les paramètres passés par valeur et par adresse peuvent cohabiter à l'intérieur d'un même sous-algorithme. Il suffit de partager les deux types de passage par un (;).

Syntaxe :

```
Procedure nom_procédure (Var param1: type1; param2, param3: type2);
```

Dans ce cas **param1** est passé par référence alors que les deux autres ont par valeur

```
Fonction <nom_fonction> (param1: type1; Var param2: type2): Type_fonction;
```

Dans ce cas **param1** est passé par valeur alors que le deuxième est passé par valeur

Exemple : soit l'algorithme précédent modifié dans le type de passage de paramètre

```
Algorithme pas-val;
Déclaration
   M : entier;
Procedure P1 (Var nombre : entier);
Debut
   Si nombre < 0 Alors
      nombre ← - nombre
   FinSi;
   Ecrire (nombre)
Fin;
Debut
   Lire (M);
   P1 (M);
   Ecrire (M)
Fin.
```

Exécutons cet algorithme toujours pour la valeur (-6)

Avant l'appel de procédure : la seule variable déclarée est la variable globale (M)

M	Écran
-6	

Après l'appel de procédure : la variable-paramètre *nombre* se substitue à la variable *M*

(~~M~~) Nombre	Écran
-6	
6	
6	6

Au retour à l'algorithme il ne reste que la variable globale avec sa nouvelle valeur.

M	Écran
6	
6	6

3.4 Exemples

Exemple 1 : un algorithme qui calcule et affiche la valeur absolue d'une valeur en utilisant une fonction

```
Algorithme exemple1;
Declaration
   a, b: Entier;
Fonction abs (unEntier: Entier): Entier;
Declaration
   valeurAbsolue: Entier;
Debut
   si unEntier ≥ 0 alors
      valeurAbsolue ← unEntier
   sinon
      valeurAbsolue ← - unEntier
   finsi;
   abs ← valeurAbsolue
fin;
Debut
   Ecrire ('Entrez un entier : ');
   Lire (a);
   b ← abs (a);
   Ecrire ('la valeur absolue de ', a, ' est ', b)
Fin.
```

Lors de l'exécution de la fonction ***abs***, la variable ***a*** et le paramètre ***unEntier*** sont associés par un passage de paramètre en entrée : La valeur de ***a*** est copiée dans ***unEntier***.

Exemple 2 : Il est demandé d'écrire un algorithme qui demande à l'utilisateur d'entrer une valeur entière positive appelée (Valeur) puis

a. qui indique à l'utilisateur si Valeur est un nombre à 2 chiffres,
b. qui affiche la factorielle de Valeur,
c. et qui saisit des valeurs au nombre de Valeur puis affiche la plus grande valeur saisie.

C'est à vous de jouer sur ce coup !

Chapitre 4 – Structures de données statiques

Les tableaux permettent de manipuler plusieurs informations de même type, de leur mettre un indice : la 1ère info, la 2ème info, . . ., la ième info, . . . Ils sont stockés en mémoire centrale comme les autres variables, contrairement aux fichiers qui sont stockés sur le disque. Une propriété importante des tableaux est de permettre un accès direct aux données, grâce à l'indice. On appelle souvent vecteur un tableau en une dimension.

4.1 Le type array

4.1.1 Principe

⇨ **Syntaxe**

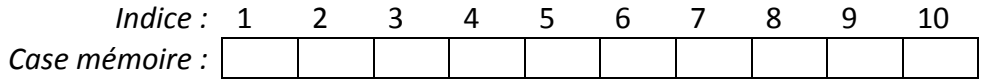
```
array [I] of T
```

I étant un type intervalle, et T un type quelconque. Ce type définit un tableau comportant un certain nombre de cases de type T, chaque case est repérée par un indice de type I.

⇨ **Exemple**

```
TYPE vec_t = array [1..10] of integer;
VAR v : vec_t;
```

v est un tableau de 10 entiers, indicés de 1 à 10.



- À la déclaration, le contenu du tableau est indéterminé, comme toute variable.
- On accède à la case indice `i` par `v[i]` (et non `v(i)`).
- Pour mettre toutes les cases à 0 on fait

```
for i:=1 to 10 do v[i]:=0;
```

⇨ **Remarque :** l'intervalle du `array` peut être de tout type intervalle, par exemple `1..10`, `'a'..'z'`, `false..true`, ou encore un intervalle d'énumérés `Lundi..Vendredi`.

On aurait pu déclarer `vecteur` comme ceci (peu d'intérêt) :

```
TYPE interv = 1..10;
vec_t = array [interv] of integer;
```

4.1.2 Contrôle des bornes

Il est en général conseillé de repérer les bornes de l'intervalle avec des constantes nommées : si on décide de changer une borne, cela est fait à un seul endroit dans le programme. L'écriture préconisée est donc :

```
CONST vec_min = 1; vec_max = 10;
TYPE vec_t = array [vec_min..vec_max] of integer;
```

⇨ **Règle 1**

Il est totalement interdit d'utiliser un indice en dehors de l'intervalle de déclaration, sinon on a une erreur à l'exécution.

Il faut donc être très rigoureux dans le programme, et ne pas hésiter à tester si un indice `i` est correct avant de se servir de `v[i]`.

⇨ **Exemple :** Programme demandant à rentrer une valeur dans le vecteur.

```
CONST vec_min = 1; vec_max = 10;
TYPE vec_t = array [vec_min..vec_max] of integer;
VAR v: vect_t; i: integer;
BEGIN
```

```
    write ('i ? '); readln(i);
    if (i >= vec_min) and (i <= vec_max) then
      begin
         write ('v[', i, '] ? ');
         readln(v[i]);
      end
    else
      writeln ('Erreur, i hors intervalle ', vec_min, '..', vec_max);
END.
```

⇨ **Règle 2 :** Le test d'un indice `i` et de la valeur en cet indice `v[i]` dans la meme expression sont interdits.

⇨ **Exemple :**

```
if (i >= vec_min) and (i <= vec_max) and (v[i] <> -1)    then ...
                                                         else ...;
```

Une expression est toujours évaluée en intégralité ; donc si `(i<=vec_max)`, le test `(v[i]<>-1)` sera quand même effectué, alors même que l'on sort du vecteur !

Solution : séparer l'expression en 2.

```
if (i >= vec_min) and (i <= vec_max) then
   if (v[i] <> -1) then ...
   else ...
else ... { erreur hors bornes };
```

4.1.3 Recopie

En Pascal, la *seule* opération globale sur un tableau est : recopier le contenu d'un tableau `v1` dans un tableau `v2` en écrivant : `v2 := v1;`

Ceci est équivalent (et plus efficace) que

```
for i := vec_min to vec_max do v2[i] := v1[i];
```

Il y a une condition : les 2 tableaux doivent être exactement de mêmes types, i.e. issus de la même déclaration.

```
TYPE
   vecA = array [1..10] of char;
   vecB = array [1..10] of char;
VAR
   v1 : vecA; v2 : vecA; v3 : vecB;
BEGIN
   v2 := v1; {legal car meme type vecA}
   v3 := v1; {illegal, objets de types <> vecA et vecB}
```

4.2 Super tableaux

Quelques types un peu plus complexes à base de tableaux, et de combinaisons entre types.

4.2.1 Tableaux à plusieurs dimensions

Exemple : dimension 1 = vecteur; dimension 2 = feuille Excel; dimension 3 = classeur Excel [faire petit schéma].

On peut créer des tableaux à plusieurs dimensions de plusieurs manières : *Faire des schémas*

```
v1: array [1..10] of array [1..20] of real
```

- Tableau de 10 éléments, chaque élément étant un tableau de 20 réels.
  On accède à l'élément d'indice `i` dans `1..10` et `j` dans `1..20` par `v1[i][j]`.

```
v2: array [1..10, 1..20] of real
```

- Tableau de 10 X 20 réels.
  On accède à l'élément d'indice `i` dans `1..10` et `j` dans `1..20` par `v2[i,j]`.

Exemple : Mise à 0 du tableau `v2`.

```
VAR
   v2: array [1..10, 1..20] of real;
   i, j: integer;
BEGIN
   for i := 1 to 10 do
     for j := 1 to 20 do
```

```
        v2[i,j] := 0.0;
END.
```

4.2.2 Tableaux de record

On peut créer des tableaux d'enregistrements, et des enregistrements qui contiennent des tableaux.

```
PROGRAM Ecole;
CONST
   MaxEleves = 35;
   MaxNotes = 10;
TYPE
   note_t = array [1..MaxNotes] of real;
   eleve_t = Record
     age, nb_notes : integer;
     notes : note_t;
     moyenne : real;
   End;
   classe_t = array [1..MaxEleves] of eleve_t;
VAR
   c : classe_t;
   nb_eleves, i, j : integer;
BEGIN
   {...}
   for i := 1 to nb_eleves do
   begin
     writeln ('Eleve n.', i);
     writeln (' age : ', c[i].age);
     write (' notes :');
     for j := 1 to c[i].nb_notes do write (' ', c[i].notes[j]);
     writeln;
     writeln (' moy : ', c[i].moyenne);
   end;
END.
```

⇨ On a comme d'habitude le droit de faire une copie globale entres variables du même type :

```
VAR c1, c2 : classe_t;
    e : eleve_t; i, j : integer;
BEGIN
   {copie globale de type classe_t}
   c2 := c1;
   {échange global de type eleve_t}
```

```
   e := c1[i]; c1[i] := c1[j]; c1[j] := e;
END.
```

⇨ Exemple de passages de paramètres : on écrit une procédure affichant un `eleve_t`.

```
PROCEDURE affi_eleve (e: eleve_t);
VAR j: integer;
BEGIN
   writeln (' age : ', e.age);
   write (' notes : ');
   for j := 1 to e.nb_notes do write (e.notes[j]);
   writeln;
   writeln (' moy : ', e.moyenne);
END;

BEGIN
   {...}
   for i := 1 to nb_eleves do
   begin
     writeln ('Eleve n.', i);
     affi_eleve (c[i]);
   end;
END.
```

`affi_eleve(e)` ne connait pas le numéro de l'élève ; l'appelant, lui, connait le numéro, et l'affiche avant l'appel. On peut encore écrire une procédure `affi_classe` :

```
PROCEDURE affi_classe (c: classe_t ; nb: integer);
VAR i : integer;
BEGIN
   for i := 1 to nb do
   begin
     writeln ('Eleve n.', i);
     affi_eleve (c[i]);
   end;
END;

BEGIN
   {...}
   affi_classe (c, nb_eleves);
END.
```

4.3 Le type string

On code une chaîne de caractère telle que 'bonjour' dans un objet de type `string`.

4.3.1 Principe

⇨ **Syntaxe :** `string [m]`

Où `m` est une constante entière donnant le nombre maximum de caractères pouvant être mémorisés.

⇨ **Exemple :**

```
VAR s: string[80];
BEGIN
   s := 'Le ciel est bleu.';
   writeln (s);
END.
```

⇨ **Codage**

- Ayant déclaré `s: string [80]`, comment sont codés les caractères ?
- En interne, Pascal réserve un `array [0..80] of char`.
- Le premier caractère est `s[1]`, le deuxième est `s[2]`, etc.
- La longueur courante de la chaîne est codé dans la case 0 (=> `ord(s[0])`).

⇨ **Remarque**

- Affecter une chaîne plus longue que l'espace réservé à la déclaration est une erreur.
- Comme la longueur courante est codée sur un `char`, elle est limitée à 255.

4.3.2 Opérateurs sur les strings

`a := ''`	Chaîne vide (longueur 0).
`a := b`	Recopie de b dans a.
`a := c + d`	Concaténation en une seule chaîne. `c` et `d` de types `string` ou `char` ; le résultat est un `string`.
`length(a)`	Longueur courante de `a`, résultat entier.

```
CONST   Slogan = 'lire la doc';
VAR     s1, s2 : string[100];
```

```
        i : integer;
BEGIN
   s1 := 'veuillez ';
   s2 := s1 + Slogan;
   writeln ('s2 = ''', s2, '''');
   writeln ('Longueur courante de s2 : ', length(s2) );
   write ('Indices des ''l'' dans s2 : ');
   for i := 1 to length(s2) do
   if s2[i] = 'l' then write(i, ' ');
   writeln;
END.
```

⇨ **Comparaison entre 2 string :** les opérateurs =, <>, <, >, <=, >=, sont utilisables, et le résultat est un booléen.

La comparaison se fait selon l'ordre lexicographique du code ASCII.

⇨ **Exemple :** Soit b un booléen ; b est-il vrai ou faux ?

```
b := 'A la vanille' < 'Zut';     {vrai}
b := 'bijou' < 'bidon';          {faux, c'est > car 'j' > 'd'}
b := 'Bonjour' = 'bonjour';      {faux, c'est < car 'B' < 'b'}
b := 'zim boum' > 'attends !';   {faux, c'est < car ' ' < 'a'}
```

⇨ **Exercice :** On considère le type LongString suivant.

```
CONST longStringMax = 4096;
TYPE LongString = record
     c: array [1..LongStringMax] of char;
     l: interer; { longueur courante }
   end;
```

Ecrire les procédure et fonctions suivantes :

```
FUNCTION longueur (s1 : LongString ) : integer;
FUNCTION est_inferieur (s1, s2 : LongString ) : boolean;
FUNCTION est_egal (s1, s2 : LongString ) : boolean;
PROCEDURE concatene (s1, s2 : LongString ; var s3 : LongString);
```

Chapitre 5 – Langages & programmation

5.1 Qu'est ce qu'un langage de programmation ?

On appelle langage de programmation un langage destiné à décrire l'ensemble des actions consécutives qu'un ordinateur doit exécuter. Les langages naturels (l'anglais, le français) représentent l'ensemble des façons qu'ont un groupe d'individu de communiquer. Les langages servant aux ordinateurs à communiquer n'ont rien à voir avec des langages de programmation, on parle dans ce cas de protocoles, ce sont deux notions totalement différentes. Un langage de programmation est une façon pratique pour nous (humains) de donner des instructions à un ordinateur.

Il faut connaître que l'ordinateur ne "*comprenant*" que le langage binaire, il lui faut donc un "*interprète*" qui lui traduise en binaire exécutable, les instructions que l'humain lui fournit en langage évolué. Cette traduction est assurée par un programme appelé *compilateur*. Le compilateur du langage L est donc un programme chargé de traduire un programme "*source*" écrit en L par un humain, en un programme "*cible*" écrit en binaire exécutable par l'ordinateur.

5.2 Bref historique

La communication entre l'homme et la machine s'effectue à l'aide de plusieurs moyens physiques externes. Les ordres que l'on donne à l'ordinateur pour agir sont fondés sur la notion d'instruction comme nous l'avons déjà vu. Ces instructions constituent un langage de programmation. Il faut connaître l'histoire des langages et se rendre compte qu'à ce jour, malgré les nouveaux langages du

marché et leur efficacité, ce sont Cobol et Fortran qui sont les plus utilisés dans le monde. L'investissement intellectuel et matériel prédomine sur la nouveauté. Cette remarque est la clef de la compréhension de l'évolution actuelle et future des langages. Les langages ont commencé directement sur des instructions machines écrites en binaire et donc rudimentaires sur le plan sémantique. Les améliorations sur cette catégorie de langages se sont limitées à construire des langages symboliques et des macro-assembleurs. J. Backus d'IBM avec son équipe a mis au point dès 1956-1958 le premier langage évolué de l'histoire, uniquement conçu pour le calcul scientifique (à l'époque l'ordinateur n'était qu'une calculatrice géante). Les années 70 ont vu s'éloigner un rêve d'informaticien : parler et communiquer en langage naturel avec l'ordinateur. Actuellement les langages évolués se diversifient et augmentent en qualité d'abstraction et de convivialité.

5.3 Classification

Les langages de programmation peuvent grossièrement se classer en deux catégories: les langages interprétés et les langages compilés.

5.3.1 Langage interprété

Un langage de programmation est par définition différent du langage machine. Il faut donc le traduire pour le rendre intelligible du point de vue du processeur. Un programme écrit dans un langage interprété a besoin d'un programme auxiliaire (l'interpréteur) pour traduire au fur et à mesure les instructions du programme.

5.3.2 Langage compilé

Un programme écrit dans un langage dit "*compilé*" va être traduit une fois pour toutes par un programme annexe (le compilateur) afin de générer un nouveau fichier qui sera autonome, c'est-à-dire qui n'aura plus besoin d'un programme autre que lui pour s'exécuter (on dit d'ailleurs que ce fichier est exécutable).

Un programme écrit dans un langage compilé a comme avantage de ne plus avoir besoin, une fois compilé, de programme annexe pour s'exécuter. De plus, la traduction étant faite une fois pour toute, il est plus rapide à l'exécution. Toutefois il est moins souple que programme écrit avec un langage interprété car à chaque modification du fichier source (fichier intelligible par l'homme: celui qui va être compilé) il faudra recompiler le programme pour que les modifications prennent effet. D'autre part, un

programme compilé a pour avantage de garantir la sécurité du code source. En effet, un langage interprété, étant directement intelligible (lisible), permet à n'importe qui de connaître les secrets de fabrication d'un programme et donc de copier le code voire de le modifier. Il y a donc risque de non-respect des droits d'auteur.

D'autre part, certaines applications sécurisées nécessitent la confidentialité du code pour éviter le piratage (transaction bancaire, paiement en ligne, communications sécurisées, ...).

5.3.3 Langages intermédiaires

Certains langages appartiennent en quelque sorte aux deux catégories (LISP, Java, Python, ..) car le programme écrit avec ces langages peut dans certaines conditions subir une phase de compilation intermédiaire vers un fichier écrit dans un langage qui n'est pas intelligible (donc différent du fichier source) et non exécutable (nécessité d'un interpréteur).

5.4 Étude de cas : Langage Pascal

5.4.1 Structure générale des programmes Pascal

L'alphabet du Pascal est tout simplement l'ensemble des caractères du clavier. Nous allons voir un certain nombre de mots du vocabulaire, mais il est important de noter, avant de poursuivre, que l'espace et le retour à la ligne sont des séparateurs (ils séparent les mots) et que la syntaxe en autorise autant qu'on le désire. En conséquence, on peut aérer les programmes pour en faciliter la lecture !

D'autre part, tout ce qui est entre accolades est considéré comme du commentaire et est ignoré par le compilateur. Cela va nous servir pour commenter nos programmes et expliquer chaque étape.

Un programme Pascal respecte toujours la structure suivante :

```
Program nomprog;
 {Définition éventuelle des constantes et des types}
 {Début de la déclaration des variables}
 var
  nomvar1, nomvar2, …: type1;
  nomvar3: type2;
  … {Il peut y en avoir autant qu'on veut}
 {Fin de la déclaration des variables}
 {Début du corps du programme}
 Begin
  instruction1;
```

```
 instruction2;
 … {Il peut y en avoir autant qu'on veut}
End.
{Fin du programme}
```

- "Program", "Type", "Var", "Begin" et "End" sont des mots du vocabulaire.
- "nomprog", "nomvar1", "nomvar2" et "nomvar3" sont des identificateurs.
- "type1" et "type2" sont des types parmi ceux autorisés.
- "instruction1" et "instruction2" sont des instructions quelconques.

5.4.2 Types, variables, opérateurs

Un type est un domaine de définition. Il existe notamment les types suivants :

- **Integer :** tous les entiers entre -32768 et +32767 (s'ils sont codés sur 16 bits).
- **Real :** les réels, codés sur 4 octets, ou parfois sur 8.
- **Char :** les caractères du clavier.
- **Boolean :** c'est un ensemble formé des 2 éléments suivants {true; false}
- **String :** ce type non standard (tous les Pascal ne disposent pas de ce type) est utile pour représenter les chaînes de caractères. 'bonjour' est une chaîne de caractères.

Une variable "a" en informatique le même sens qu'en mathématique. Plus précisément, une variable "a" les caractéristiques suivantes :

- Un identificateur (autrement dit un nom). On peut donner le nom que l'on veut ou presque : un identificateur doit commencer par une lettre de l'alphabet, suivie ou non de signes alphanumériques. Par exemples, `x`, `total`, `x1`, `x234tt`, `AaBb` sont des noms de variable possibles, mais `2x`, `gh?st`, `kk**` ne le sont pas.
- Un type parmi ceux cités plus haut
- Une valeur

Remarque : Attention, pour différencier le nom des variables de type caractère des caractères eux-mêmes, on met chaque caractère entre 2 apostrophes. Ex : 'x' est un caractère, x est une variable.

- Une adresse en mémoire (mémoire vive, bien entendu). On ne se servira pas directement des adresses mémoire des variables, mais il est tout de même bon de savoir qu'elles existent.

Il existe de nombreux opérateurs et fonctions pour tous les types. Nous classons ci-dessous les opérateurs par rapport aux domaines de définition des opérandes et du résultat :

```
Integer x Integer --> Integer : +, -, *, div, mod
```

- `x div y` est le résultat entier de la division de x par y. Par exemple `5 div 2 = 2`
- `x mod y` est le résultat du reste de la division entière de x par y. Par exemple, `5 mod 2 = 1`

```
Real x Real --> Real : +, -, *, /
```

(Sachant qu'un integer est aussi un real)

```
Real --> Integer : trunc, round
```[3]

- `trunc (x)` est la valeur entière de x. Exemple, `trunc (4.8) = 4`
- `round (x)` est la valeur entière approchée de x. Exemple, `round (4.8) = 5`

```
Real --> Real : cos, sin, atan, ln, sqrt, sqr, abs
```

- `sqrt (x)` est la racine carrée de x (SQuaRe Transpose en Anglais)
- `sqr (x)` est le carré de x

```
Char --> Char : succ, pred
```

- `succ (x)` est le caractère qui suit x dans la table ASCII qui code les caractères.
- `pred (x)` est le caractère qui précède x.

```
Char --> integer : ord
```

- `ord (x)` donne le rang du caractère x dans la table ASCII.

```
Integer --> Char : chr
```

- `chr (x)` donne le caractère qui est codé par le nombre x.

```
Real x Real --> Boolean : >, <, =, >=, <=, <>
```

[3] Notez la nécessité des parenthèses autour de l'opérande lorsque la fonction est unaire.

- Les opérateurs de comparaison fournissent un résultat nécessairement vrai ou faux, ce qui correspond au type booléen.

5.4.3 L'instruction d'affectation

L'instruction d'affectation a pour but d'affecter (de donner) une valeur à une variable. Sa syntaxe est la suivante :

```
nomvar := expression mathématique
```

- `nomvar` est le nom d'une variable.
- L'expression mathématique est quelconque mais son évaluation doit donner une valeur du même type que celui de la variable. Il existe des niveaux de priorité pour les opérateurs, le niveau le plus bas étant le plus fort :
  - Niveau 1 : tous les opérateurs unaires
  - Niveau 2 : `*, /, div, mod, and`
  - Niveau 3 : `+, -, or`
  - Niveau 4 : `=, <, >, <=, >=, <>`

Des parenthèses peuvent être utilisées pour modifier les priorités d'évaluation.

5.4.4 Les instructions d'Entrées-Sorties

Pour afficher des caractères à l'écran, il existe deux instructions :

A. Write (liste d'expressions)

Une liste d'expressions est une ou plusieurs expressions mathématiques séparées par des virgules. Chaque expression est évaluée et son résultat est affiché[4].

Exemple :
```
x := 5 ;  {On affecte 5 à la variable x, supposée de type integer}
Write ('Bonjour chers amis…');
Write (10, 3*(x-2), '8');
```

⇨ **Écran :** `Bonjour chers amis… 10 9 8`

[4] Pour afficher un apostrophe, qui sert de signe de fin de chaîne, il faut en écrire 2 à la suite. Ex: 'L''autre'

B. Writeln (liste d'expressions)

Writeln agit exactement comme **Write**, mais ajoute un retour à la ligne après le dernier affichage. Notons que **Writeln** sans paramètre ni parenthèse permet de passer à la ligne suivante.

Exemple :
```
x := 5 ;  {affectation de 5 à la variable x, supposée de type integer}
Writeln ('Bonjour chers amis…');
Writeln (10, 3*(x-2), '8');
```

⇨ **Écran**[5] **:**
```
Bonjour chers amis…
10 9 8
```

Pour que l'utilisateur attribue une valeur à une variable pendant l'exécution du programme, il faut utiliser l'instruction **Readln**. Sa syntaxe est la suivante :

```
readln (nomvar)
```

- nomvar doit être une variable définie dans la déclaration des variables.
- A l'exécution du programme, lorsque l'instruction **Readln** est rencontrée, l'utilisateur doit taper une valeur, puis la touche return (ou entrée). A ce moment là, la valeur tapée est donnée à la variable nomvar et le programme se poursuit.

Nous pouvons maintenant écrire des petits programmes. Exemple :

```
Program moyenne; {Calcul de la moyenne de 2 nombres}
   var x, y: integer;
   moy: real;
   Begin
     writeln ('Entrez la valeur du premier nombre'); readln (x);
     writeln ('Entrez la valeur du deuxième nombre'); readln (y);
     moy:=(x + y)/2;
     writeln ('La moyenne des 2 nombres est : ', moy :4:1);
   End.
```

⇨ **Écran :**
```
Entrez la valeur du premier nombre
5
```

[5] Pour améliorer l'affichage et réduire l'espace entre les nombres, on peut mettre un format. Ainsi, si on veut que l'affichage des entiers ne prennent que 2 caractères, il suffit de rajouter :2 après l'expression. Si on avait eu des réels, le format comprend 2 paramètres :p1 :p2, le premier pour spécifier le nombre total de caractères et le second pour spécifier le nombre de chiffres après la virgule.

```
Entrez la valeur du deuxième nombre
28
La moyenne des 2 nombres est : 16,5
```

5.5 Exercices corrigés

5.5.1 Exercices

Exercice 1. Écrire un programme qui demande à l'utilisateur les coordonnées de 2 points distincts du plan et qui affiche les coordonnées du point milieu.

Exercice 2. Quel est le résultat à l'écran du programme suivant :

```
Program simple;
   Var x,y : integer;
   Begin
      x:=1; y:=2;
      x:=x+1; y:=y+x;
      writeln ('y vaut :', y');
   End;
```

Exercice 3. Écrire un programme qui demande à l'utilisateur une valeur pour U_0, `r` et `n` et qui affiche la énième valeur de la suite arithmétique définie par U_0 et $U_{n+1} = U_n + r$. (On rappelle la propriété : $U_n = U_0 + n.r$)

Exercice 4. Écrire un programme qui demande à l'utilisateur les valeurs de 2 entiers `x, y`, qui permute leur valeur et qui les affiche.

Exercice 5. Application physique : Au temps $t_0=0$, un obus est lancé verticalement en l'air à partir d'une plate-forme située à y_0 mètres du sol avec une vitesse initiale de v_0 mètres par seconde. En prenant la constante de gravitation égale à `9.81`, écrire un programme qui demande à l'utilisateur les valeurs de y_0, v_0, une valeur de temps `t` et qui affiche la vitesse `v` ainsi que la hauteur `y` de l'obus par rapport au sol précisément à l'instant `t`. (On évitera de prendre des valeurs de `t` trop grandes pour éviter de trouver des valeurs de `y` négatives)

Exercice 6. Écrire un programme qui demande à l'utilisateur la valeur d'une durée exprimée en secondes et qui affiche sa correspondance en heures minutes secondes. Ex: 3800 s --> 1 heure 3 minutes 20 secondes.

Exercice 7. Trouvez toutes les erreurs de syntaxe et toutes les erreurs sémantiques dans le programme suivant qui calcule la moyenne exacte de 3 nombres entiers.

```
Programme calcule moyenne;
   Var x1: entier x2: entier
   x3: entier
   moy: entier
   Begin
     writeln (Entrez 3 nombres entiers);
     readln(x1);
     readln(x2);
     readln(x3);
     moy:=(x1+x2+x3/3);
     writeln('La moyenne est ', 'moy');
   End.
```

Exercice 8. Écrire un programme qui demande à l'utilisateur un entier plus petit que 8 et qui affiche le nombre binaire correspondant (utilisation de `div` et `mod`).

Exercice 9. Écrire un programme qui demande à l'utilisateur un nombre entier de 4 chiffres commençant par 1 et composé de 0 et de 1, puis, en supposant ce nombre binaire, qui affiche le nombre en base 10.

5.5.2 Solutions

Exercice 1.

```
Program milieu;
   Var x1, y1, x2, y2, xm, ym: real;
   Begin
     writeln ('Entrez les coordonnées du premier point');
     readln (x1);
     readln (y1);
     writeln ('Entrez les coordonnées du deuxième point');
     readln (x2);
     readln (y2);
     xm := (x1+x2)/2;
     ym := (y1+y2)/2;
     writeln ('Les coordonnées du point milieu sont :', xm :5:2, ym:5:2);
   End.
```

Exercice 2.

⇨ **Écran :** y vaut 4

Notez que même si `y := y+x` est sur la même ligne que `x := x+1, x := x+1` est exécuté avant.

Exercice 3.

```
Program suite;
   Var U0, Un, r : real;
   n: integer;
   Begin
     writeln ('Entrez les valeurs de U0, r et n');
     readln (U0);
     readln (r);
     readln (n);
     Un := U0 + n*r;
     writeln ('La ',n:3,'ième valeur de la suite est :', Un :5:2);
   End.
```

Exercice 4.

```
Program permutation;
   Var x, y, stockage : integer;
   Begin
     writeln ('Entrez la valeur de x et y');
     readln(x);
     readln(y);
     stockage := x;   {On mémorise la valeur de x pour ne pas l'oublier}
     x := y;          {On met la valeur de y dans x}
     y := stockage;   {On met dans y l'ancienne (!!) valeur de x}
     writeln('x vaut maintenant ',x:4,' et y ', y:4);
   End.
```

Exercice 5.

```
Program physique;
   Var v0, y0, t, v, y: real;
   Begin
     writeln ('Entrez les valeurs de v0 et y0');
     readln (v0);
     readln (y0);
     writeln ('Entrez une valeur pour le temps t');
     readln(t);
     v := -9.81*t + v0;
```

```
      y := -0.5*9.81*sqr(t) + v0*t + y0;
      writeln ('Vitesse : ',v:5:2,' Altitude : ',y:5:2);
    End.
```

Exercice 6.

```
  Program temps;                {Une des solutions possibles}
    Var t, h, mn, s: integer;
    Begin
      writeln('Entrez la valeur d''une durée en secondes');
      readln (t);
      h := t div 3600;
      mn := (t mod 3600) div 60;
      s := t mod 60;
      writeln('Cette durée équivaut à ', h:2,' heures,', mn :2, ' minutes
et ',s:2,' secondes.');
    End.
```

Exercice 7.

Voici le programme corrigé. Toutes les erreurs sémantiques sont signalées en commentaires. Notez que le programme n'est pas changé si on insère ou si on supprime des sauts de ligne ou des blancs entre chaque mot (1 minimum).

```
  Program calculemoyenne;       {Erreur de syntaxe: pas de blanc dans
                                 l'identificateur}
    Var x1: integer; x2: integer; x3: integer;
    moy: real;                  {Erreur sémantique: la moyenne est un réel!}
    Begin
      writeln ('Entrez 3 nombres entiers');
      readln(x1);
      readln(x2);
      readln(x3);
      moy:=(x1+x2+x3)/3;  {Erreur sémantique : attention aux parenthèses}
      writeln('La moyenne est ', moy);
                    {Erreur sémantique : affichage de la valeur de moy!}
    End.
```

Exercice 8.

```
  Program decimalversbinaire;
    Var x, a1, a2, a3 : integer;
    Begin
       writeln('Entrez un entier inférieur à 8');
       readln (x);
```

```
        a1 := x div 4;
        a2 := (x mod 4) div 2;
        a3 := x mod 2;
        write ('Le nombre binaire correspondant est: ',a1:1, a2:1, a3:1);
     End.
```

Exercice 9.

```
  Program binaireversdecimal;
     Var x, resultat : integer;
     Begin
       writeln('Entrez un entier de 4 chiffres commençant par 1 et composé
de 0 et de 1');
       readln (x);
       resultat := 8+((x-1000)div100)*4 + ((x mod 100)div 10)*2 + x mod 10;
       writeln('En base 10, ce nombre est : ',resultat:2);
     End.
```

Bibliographie

[1] Thomas H. Cormen (2013), *Algorithmes : Notions de base*.

[2] Edouard Thiel (2004), *Algorithmes et programmation en Pascal : Cours, TD corrigés, TP et Annales corrigées*, Faculté des Sciences de Luminy, Université d'Aix-Marseille.

[3] Jean Marc Salotti (1998), *Cours et exercices corrigés en Pascal*, UFR SM / Université Bordeaux 2.

[4] Philipe Mercier (1987), *Turbo Pascal facile*.