

Cours d'informatique de licence
première année
premier semestre
Bases de la programmation impérative

Thierry Lecroq
Université de Rouen
Faculté des Sciences et des Techniques
Département d'Informatique de Rouen
ABISS
mél : Thierry.Lecroq@univ-rouen.fr
adresse universelle : <http://www-igm.univ-mlv.fr/~lecroq>

Table des matières

1	Généralités sur le traitement de l'information	5
1.1	La notion de programme	5
1.2	Organisation matérielle	6
1.3	L'unité centrale	6
1.3.1	L'unité arithmétique et logique	6
1.3.2	L'unité de commande	8
1.3.3	La mémoire centrale	8
1.3.4	Le bus	9
1.3.5	L'horloge	9
1.4	Les organes d'entrées/sorties	10
1.5	Petit historique	10
1.5.1	Les premières machines à calculer	10
1.5.2	Les ordinateurs programmables	11
1.5.3	Les ordinateurs programmables	11
1.5.4	Les progrès marquants	11
1.5.5	Les premiers microprocesseurs	11
1.5.6	Les premiers micro-ordinateurs	12
1.6	Les systèmes d'exploitation des ordinateurs	12
1.6.1	Les différents types de systèmes d'exploitation	12
1.7	Multimédia et hypertexte	13
1.8	Réseaux, Internet et WWW	13
1.9	Le langage HTML	13
1.10	La notion d'algorithme	15
1.11	La programmation	15
1.12	Les différentes couches	16
2	Le langage Pascal	17
2.1	La structure d'un programme Pascal	17
2.2	La notion de variable	18
2.2.1	Règles de formation des identificateurs	19
2.2.2	Les types de base et les opérations	19
2.2.3	Les constantes	22
2.2.4	La déclaration des variables	22

2.2.5	La manipulation de variables	23
2.3	L'exécution d'un programme	27
2.3.1	Un exemple de programme Pascal	27
2.4	Les commentaires	29
2.5	Les fonctions	29
2.5.1	Déclaration d'une fonction	30
2.5.2	Liste de paramètres formels	30
2.5.3	La partie déclarations	31
2.5.4	Le résultat	31
2.5.5	Un exemple de fonction	31
2.5.6	Appel d'une fonction	32
2.5.7	Fonctions prédéfinies	32
2.6	L'instruction alternative	32
2.6.1	Les comparaisons	33
2.6.2	Évaluation des expressions booléennes	35
2.6.3	Tests imbriqués	35
2.7	L'instruction itérative	36
2.7.1	Accumulateur	38
2.8	Les tableaux	38
2.8.1	Le type intervalle	39
2.8.2	Déclaration des tableaux	39
2.8.3	Manipulation des tableaux	40
2.8.4	Le type chaînes de caractères	42
2.9	Organisation de la mémoire	44
2.10	Un exemple (presque) complet	45
3	L'organisation des données	48
4	Représentation des données	51
4.1	Les conversions	52
4.1.1	La conversion décimal en binaire	52
4.1.2	La conversion binaire en décimal	53
4.2	La représentation des entiers	54
4.2.1	Arithmétique élémentaire en binaire	54
4.2.2	Complément à 2	55
4.3	La représentation des réels	57
4.3.1	La norme IEEE sur 32 bits	58
4.4	La représentation des caractères	58
5	Les instructions	59
5.1	L'assembleur	59
5.2	Les instructions	59
5.2.1	Les instructions de transfert	60
5.2.2	Les instructions de calcul arithmétique	60

5.2.3 Les instructions de branchement 60

Objectifs

Ce cours s'adresse à des étudiants de premier semestre de première année de licence d'informatique. Il ne nécessite aucun pré-requis en informatique. Il aborde les notions de base d'architecture des ordinateurs, de l'organisation des données, de la représentation des données et des instructions.

Il a aussi pour but de présenter une introduction à la programmation impérative en Pascal. Les notions abordées sont les suivantes :

- types de base ;
- séquentialité ;
- affectation ;
- fonctions ;
- instruction alternative ;
- instruction itérative ;
- tableaux.

Ce cours est accessible sur la Toile aux formats *PostScript* et *Portable Document Format* aux adresses suivantes :

- <http://www-igm.univ-mlv.fr/~lecroq/info1/cours.ps> ;
- <http://www-igm.univ-mlv.fr/~lecroq/info1/cours.pdf>.

Les transparents sont également disponibles :

- <http://www-igm.univ-mlv.fr/~lecroq/info1/transp.ps> ;
- <http://www-igm.univ-mlv.fr/~lecroq/info1/transp.pdf>.

Il en est de même pour les exercices de TD :

- <http://www-igm.univ-mlv.fr/~lecroq/info1/td.ps> ;
- <http://www-igm.univ-mlv.fr/~lecroq/info1/td.pdf>.

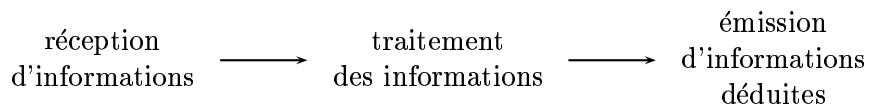
Chapitre 1

Généralités sur le traitement de l'information

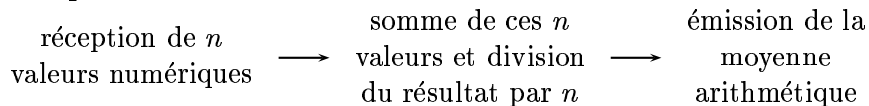
Les ordinateurs sont utilisés pour

- le traitement d'informations;
- le stockage d'informations.

Le schéma global d'une application informatique est toujours le même :



Exemple :



Tout traitement demandé à la machine, par l'utilisateur, se traduit par l'exécution séquencée d'opérations (appelées **instructions**).

1.1 La notion de programme

L'objet et l'intérêt de la programmation est de permettre de spécifier à une machine un travail à effectuer de façon automatique. Pour cela il faut en général fournir à la machine les valeurs des paramètres que l'on appelle les **données**. Ensuite la machine effectue des opérations sur ces données en suivant un schéma de fonctionnement qui lui aura été précisé, c'est ce que l'on appelle un **programme**. Enfin tout ceci n'a d'intérêt que dans la mesure où la machine fournit des **résultats**.



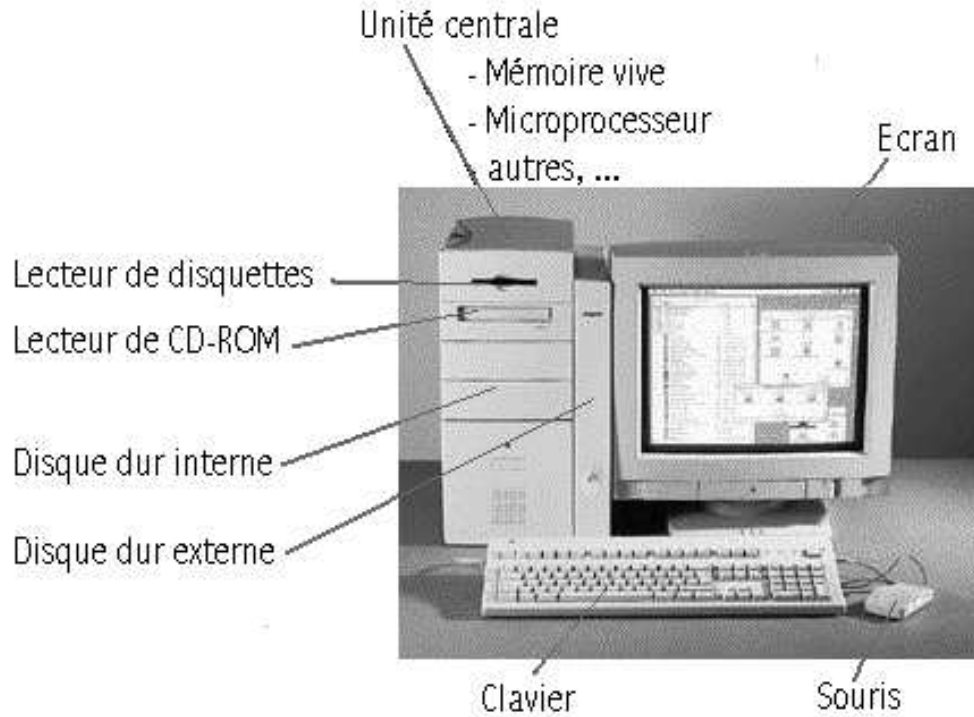


FIG. 1.1 – Vue externe d'un micro-ordinateur.

1.2 Organisation matérielle

Physiquement, un ordinateur s'organise autour d'une unité centrale qui contient une unité arithmétique et logique, une unité de commande, de la mémoire centrale (ou principale ou encore interne) (voir figures 1.1 et 1.2). Tous ces composants sont reliés par un bus et sont cadencés par une horloge.

1.3 L'unité centrale

C'est l'unité centrale qui contient le ou les microprocesseurs, véritables cerveaux de la machine. C'est elle qui effectue les opérations demandées à la machine. Ces opérations sont exprimées grâce à un jeu d'instructions. Le nombre d'instructions d'un microprocesseur est limité. Ces instructions sont soit des instructions élémentaires ou logiques, soit des instructions caractérisant le mouvement des données entre la mémoire et les organes d'entrées/sorties.

1.3.1 L'unité arithmétique et logique

L'unité arithmétique et logique traite les données pour fournir les résultats. Elle est constituée d'une unité de traitement arithmétique (addition,

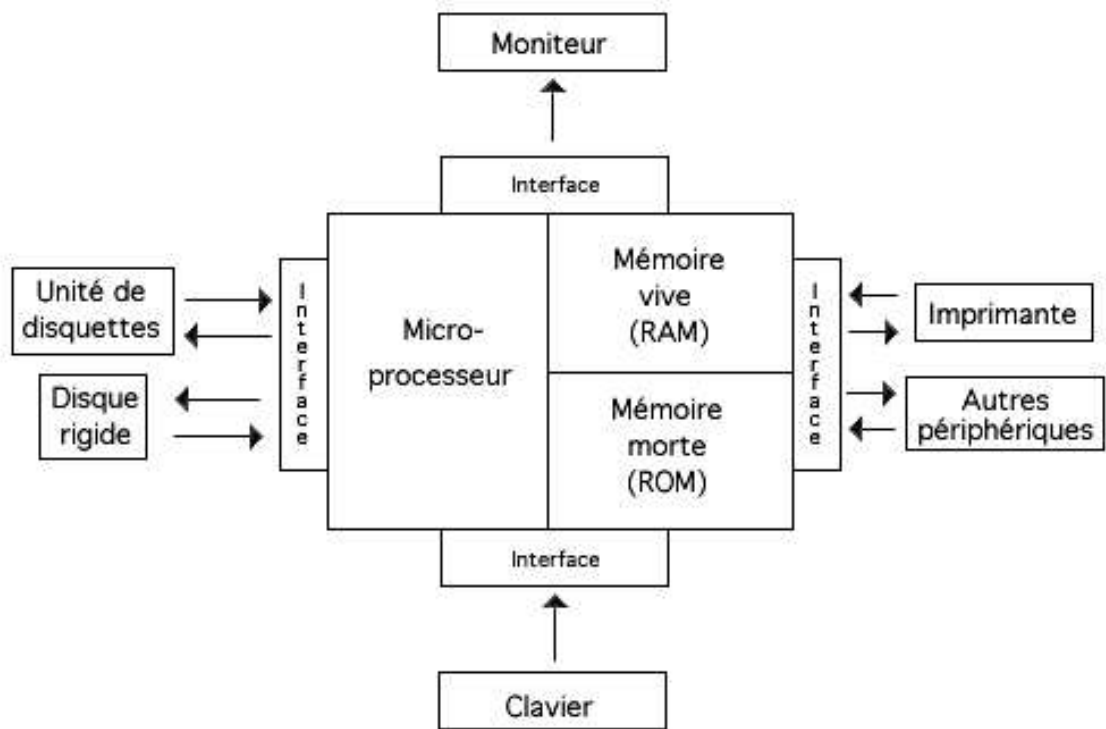


FIG. 1.2 – Architecture conceptuelle d'un ordinateur.

soustraction, multiplication, division, ...) et d'une unité de traitement logique (comparaisons et opérations logiques). Elle est associée à des mémoires spécialisées dans lesquelles sont effectués les traitements et que l'on appelle les **registres**.

1.3.2 L'unité de commande

L'unité de commande coordonne l'ensemble des tâches que doit effectuer l'ordinateur. Elle est composée d'organes qui décodent les instructions à effectuer et les transmettent à l'unité arithmétique et logique. Elle est en relation avec la mémoire principale pour y lire ou y écrire des informations. L'unité de commande est également associée à un registre appelé **registre à instruction** qui contient la prochaine instruction à exécuter.

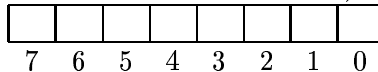
1.3.3 La mémoire centrale

La mémoire centrale est directement reliée à l'unité centrale et contient le ou les programmes à exécuter.

L'organisation de la mémoire

La plus petite information que l'on puisse mémoriser ou manipuler est appelée **bit** (mot-valise contraction de *binary digit*) et représente donc un chiffre binaire : 0 ou 1 (le courant ne passe pas ou le courant passe). Une mémoire peut donc être considérée comme un assemblage de bits contigus. On a l'habitude de regrouper les bits :

- tout d'abord par groupe de 8 bits consécutifs ce qui forme un **octet** (**byte** en anglais, à ne pas confondre avec bit !);



Le bit en position 0 est le bit de poids faible et le bit en position 7 est le bit de poids fort.

- ensuite pour faciliter le traitement et l'adressage des données que contient la mémoire on regroupe les bits par ce que l'on appelle des **mots mémoires**. La taille d'un mot mémoire est généralement une puissance de 2, elle correspond à la taille du **bus**. Cette taille des mots mémoires varie suivant les machines et les constructeurs.

La taille des mémoires (et aussi des informations qu'elles contiennent) est exprimée en un certain nombre d'octets :

- kilooctet : 1 ko = 1 024 ($2^{10} \approx 10^3$) octets ;
- mégaoctet : 1 Mo = 1 048 576 ($2^{20} \approx 10^6$) octets ;
- gigaoctet : 1 Go = 1 073 742 824 ($2^{30} \approx 10^9$) octets ;
- teraoctet : 1 To = 1 099 512 651 776 ($2^{40} \approx 10^{12}$) octets ;

Les différents types de mémoire

Il existe deux types fondamentaux de mémoires :

- les mémoires **RAM** (Random Access Memory) : mémoires volatiles ¹ (« vives ») où sont stockés les programmes et les données des utilisateurs. L'accès à ce type de mémoire est très rapide.
- les mémoires **ROM** (Read Only Memory) : mémoires persistantes (non volatiles ou « mortes ») qui contiennent des données et des programmes figés tels que des codes de caractères ou des programmes d'exploitation de la machine.

1.3.4 Le bus

Le **bus** est un câble électrique dont le rôle est de transporter les données d'un organe vers un ou plusieurs autres. Ce câble est composé de plusieurs fils, chaque fil pouvant être mis au potentiel nul ou au potentiel positif. Chaque fil transporte donc une information qui peut prendre deux valeurs : 0 ou 1.

1.3.5 L'horloge

L'horloge cadence la communication à l'intérieur de l'ordinateur. Chaque organe est relié à l'horloge et fonctionne au rythme qu'elle dicte. Le processeur effectue une instruction à chaque impulsion. Une donnée placée sur le bus n'est disponible que pendant un **cycle** d'horloge (un cycle est la durée qui sépare deux impulsions). Toute opération est décomposable en une suite d'instructions atomiques, chacune d'elles prenant exactement un cycle d'horloge.

Considérons, par exemple, l'opération qui consiste à amener une donnée de la mémoire jusqu'au processeur (accès à la valeur d'une variable). En simplifiant un peu, cette opération se décompose comme suit :

1. le processeur place, sur le bus, le numéro (adresse) de la case dans laquelle se trouve la donnée ;
2. le coprocesseur gestionnaire de la RAM prend ce numéro sur le bus ;
3. le coprocesseur gestionnaire de la RAM lit le contenu de la case dont il vient de recevoir l'adresse, et le place sur le bus ;
4. le processeur récupère la donnée sur le bus.

En général, la cadence de l'horloge est donnée en GHz (gigahertz). On parle, par exemple, d'ordinateurs 1,5 MHz, ce qui signifie que la durée d'un cycle est de $\frac{1}{1\,500\,000}$ seconde soit 0,0006 μ -seconde.

¹Une mémoire est dite volatile lorsque son contenu s'efface lorsque la machine est mise hors tension, elle est dite non volatile sinon.

1.4 Les organes d'entrées/sorties

Les organes d'entrées/sorties servent à faire communiquer la machine avec le monde extérieur et notamment à fournir à la machine les programmes et les données dont l'exécution rendra possible en retour l'obtention de résultats. Ils permettent à l'homme de commander et d'utiliser la machine.

Ces organes sont également appelés organes **périphériques**. Parmi les différents périphériques existants, on peut citer :

- les unités de visualisation (visuel, visu, moniteur) ;
- le clavier (*keyboard*), la souris (*mouse*), le crayon optique ;
- les imprimantes (*printers*), les traceurs de courbes ;
- les modems ;
- un robot, l'alarme de sa maison, un feu tricolore, ...

Il faut aussi ranger dans cette catégorie les unités de mémoires secondaires (ou externes) :

- disques durs ;
- lecteurs de disquettes ;
- lecteurs/graveurs de CD-ROM/DVD-ROM ;
- lecteurs de bandes magnétiques.

Des processeurs spécialisés (appelés coprocesseurs) sont associés à chaque périphérique.

1.5 Petit historique

1.5.1 Les premières machines à calculer

1623	William Schikard	première machine à calculer
1642	Blaise Pascal	Pascaline addition et soustraction
1673	Gottfried Wilhem von Leibniz	multiplication et division
1834	Charles Babbage	deux machines : machine différentielle machine analytique
1890	Herman Hollerith	cartes perforées pour analyser les résultats du recensement aux États Unis

1.5.2 Les ordinateurs programmables

1937	Howard Aiken	Mark I d'IBM 17 m de long, 2,5 m haut 3300 engrenages, 1400 commutateurs 800 km de fils
1938	Konrad Suze	Z3 relais électroniques premier à utiliser le binaire
1947	Mark II	composants électroniques

1.5.3 Les ordinateurs à lampes

1942	John Vincent Atanasoff Clifford Berry	ABC (Atanasoff Berry Computer)
1946	John Mauchly John Eckert	ENIAC (Electronic Numerical Integrator And Computer) programmable manuellement avec des commutateurs ou des câbles à enficher
1946	John Mauchly John Eckert John Von Neuman	EDVAC (Electronic Discrete Variable Computer) 1024 mots en mémoire centrale 20 000 mots sur un support magnétique
1948	John Mauchly John Eckert	UNIVAC (UNIversal Automatic Computer) 1000 mots de 12 bits 8333 additions ou 555 multiplications / s 25 m ² superficie au sol

1.5.4 Les progrès marquants

1948	transistor	Bell Labs
1958	circuit intégré	Texas Instruments
1960	premier ordinateur à base de transistors	IBM 7000

1.5.5 Les premiers microprocesseurs

1971	Intel 4004	4 bits
1972	Intel 8008	8 bits

1.5.6 Les premiers micro-ordinateurs

1971	Kenback 1	256 octets
1973	Micral-N	
1975	Altair 8800	
1976	Apple I	1 MHz
	Steve Wozniak	4ko RAM
	Steve Jobs	1ko mémoire vidéo
1977	Apple II	
1981	IBM PC	Intel 8080 5MHz

1.6 Les systèmes d'exploitation des ordinateurs

Au départ la machine ne comporte que le matériel : unité centrale et périphériques permettant la liaison avec l'utilisateur. À la mise en route du système un petit programme est lancé qui gère l'ensemble des périphériques et la séquence des opérations à effectuer. Ce petit programme s'appelle le **moniteur** et il est stocké en ROM. Il sert également à lancer le **système d'exploitation** de la machine qui, lui, doit être stocké en mémoire secondaire. Un système d'exploitation est un programme qui sert d'interface (intermédiaire) entre un ordinateur et les utilisateurs dans le but de rendre l'utilisation de l'ordinateur facile et efficace. Les principales fonctions d'un système d'exploitation sont :

- la gestion et la conservation des informations par l'intermédiaire d'un système de **fichiers** ;
- la gestion de l'ensemble des ressources (processeurs, mémoires, registres, imprimantes, ...) permettant l'exécution d'un programme ;
- fournir à l'utilisateur un langage de commande facile et efficace.

Exemples : Unix, Windows, Linux, MacOS, BeOS, ...

1.6.1 Les différents types de systèmes d'exploitation

Un système d'exploitation est **mono-tâche** s'il ne peut fournir qu'une seule tâche à la fois au microprocesseur, il est **multi-tâche** s'il peut gérer le suivi simultané de plusieurs tâches par le processeur. Dans ce dernier cas, le système doit faire progresser toutes les tâches simultanément. Il traite un morceau de la première, puis de la seconde, etc. On parle alors d'**ordonnement** des tâches.

Un système d'exploitation est **mono-utilisateur** s'il ne peut gérer les demandes que d'un seul utilisateur à la fois, il est **multi-utilisateur** s'il peut gérer simultanément les demandes de plusieurs utilisateurs.

Remarque : un système multi-utilisateur est toujours multi-tâche.

1.7 Multimédia et hypertexte

Un ordinateur est **multimédia** s'il peut stocker et traiter des textes, des sons, des images fixes et des images vidéos.

Un document **hypertexte** est constitué par un ensemble de pages (fichiers) reliées par des liens (renvois) placés dans le texte.

1.8 Réseaux, Internet et WWW

Un **réseau** permet de connecter plusieurs ordinateurs entre eux. On peut distinguer au moins deux types de réseaux :

- les réseaux locaux permettent de relier des ordinateurs dans un même lieu à l'aide de câbles ou liaisons optiques ;
- les réseaux distants permettent de relier des ordinateurs distants à l'aide de liaison téléphoniques, satellites, ...

L'intérêt des réseaux est entre autres le partage des ressources et des logiciels, la communication et le transfert d'informations.

Internet est défini comme le réseau des réseaux. Les ordinateurs du monde entier sont connectés entre eux à l'aide de câbles, de lignes téléphoniques et de satellites.

Des logiciels spécifiques permettent d'accéder aux principaux services :

- l'accès distant (**telnet**, **ssh**) ;
- la messagerie électronique (**mail**) ;
- le transfert d'informations (**ftp**), **sftp** ;
- la consultation de forums (**news**) ;
- la consultation de la Toile (*Web* ou *WWW*)

La Toile (ou *World Wide Web* ou *Web*) est constituée par un ensemble d'informations multimédia contenant du texte, des images, des vidéos, des sons, ...

Les logiciels permettant de *surfer* sur la Toile sont appelés des navigateurs (*browsers*). Les principaux sont *Netscape* et *Internet Explorer*.

Le langage principal d'écriture de pages *Web* est HTML (*Hyper Text Mark-up Language*).

Il existe des moteurs de recherche (*AltaVista*, *Excite*, *Google*, *HotBot*, *NorthernLight*, *Yahoo*, ...) pour rapidement localiser des informations à partir de mots clés.

1.9 Le langage HTML

Le langage de base de description des sites sur la Toile est un langage à balise. La ligne suivante :

```
<X>y</X>
```

signifie que la commande X sera appliquée au texte y.

Les balises <HTML>, <HEAD>, <TITLE> et <BODY> sont indispensables. Il existe de nombreuses balise pour mettre en forme le texte et y insérer des images, du sons, des liens sur d'autres sites.

Ainsi le fichier suivant :

```
<HTML>
<HEAD>
  <TITLE>Exemple de licence 1</TITLE>
</HEAD>
<BODY>
<A HREF="http://www.univ-rouen.fr">
<IMG SRC="logo-ur.jpg" alt="Universit&eacute; de Rouen">
</A>
<H1>
  Cours d'informatique de licence
</H1>
<H2>
  premier semestre
</H2>
<H3>
  premi&egrave;re ann&eacute;e
</H3>
Ce cours est accessible en ligne aux adresses suivantes
<UL>
  <LI>
    <I>PostScript</I> :
    <A HREF="http://www-igm.univ-mlv.fr/~lecroq/info1.ps">
      http://www-igm.univ-mlv.fr/~lecroq/info1.ps</A>
  </LI>
  <LI>
    <B>Portable Document Format</B> :
    <A HREF="http://www-igm.univ-mlv.fr/~lecroq/info1.pdf">
      http://www-igm.univ-mlv.fr/~lecroq/info1.pdf</A>
  </LI>
</UL>
</BODY>
</HTML>
```

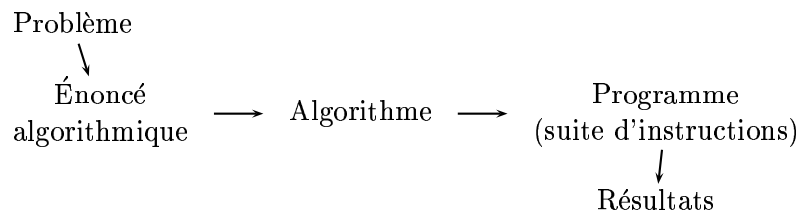
sera visualisé comme dans la figure 1.3.



FIG. 1.3 – Visualisation d'un fichier HTML.

1.10 La notion d'algorithme

Pour résoudre un problème de manière informatique, il y a un cheminement méthodique à respecter :



Un **algorithme** est la description formelle d'un procédé de traitement qui permet à partir d'un ensemble d'informations initiales d'obtenir des informations déduites. C'est une succession finie et non ambiguë d'opérations clairement posée. Un algorithme se termine donc toujours.

Un **programme** est une suite d'instructions définies dans un langage donné pouvant décrire un algorithme.

1.11 La programmation

Le seul langage connu de la machine étant le langage binaire, il est indispensable de pouvoir spécifier les séquences d'instructions et les données sous une forme plus facilement manipulable par l'utilisateur.

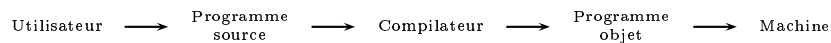
On a ainsi recours à des notations symboliques qui permettent de décrire les actions à effectuer dans des langages symboliques : ce sont les langages de programmation.

On peut succinctement décrire les différents types de langage comme suit :

- langage machine : directement compréhensible par la machine;
- langage d’assemblage (ou **assembleur**) : très facilement traduisible pour être compris par la machine;
- langage de programmation : doit être compilé ou interprété pour être compris par la machine.

Il existe deux types de langages de programmation : les langages compilés et les langages interprétés.

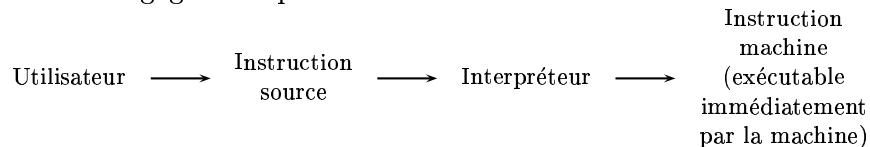
Pour les langages compilés on a le schéma suivant :



Un compilateur est un programme dont les données (fournies par l'utilisateur) sont constituées par un programme source et les résultats (fournis à l'utilisateur) par un programme objet.

Exemples : Pascal, C, ADA, FORTRAN, ...

Pour les langages interprétés la situation est la suivante :



Chaque instruction est traduite en langage machine au fur et à mesure de l'exécution du programme.

Exemples : Basic, LISP, ...

1.12 Les différentes couches

On vient donc de voir qu'il existe différentes couches d'organisation de l'outil informatique :

- une couche matérielle;
- une couche système d'exploitation;
- une couche logiciel d'applications (traitements de textes, tableurs, ...);
- une couche programme utilisateurs.

La première couche correspond au matériel (*hardware*), les autres correspondent au logiciel (*software*).

Chapitre 2

Le langage Pascal

Le langage Pascal a été créé à Zurich par une équipe universitaire dirigée par Nicklaus Wirth. La première version est apparue en 1968. Une codification (ensemble de règles) est apparue en 1973 avec le langage Pascal Standard qui sert de référence pour tous les compilateurs Pascal, ce qui n'exclut pas des extensions. Les caractéristiques du langage Pascal sont :

- langage d'enseignement ;
- langage structuré ;
- implantation aisée sur une grande gamme de matériels ;
- vocation universelle (scientifique, gestion, ...);
- description rigoureuse du langage.

2.1 La structure d'un programme Pascal

Un programme Pascal est une suite d'ordres donnés à la machine. Ces ordres sont donnés en utilisant une syntaxe bien précise et en utilisant des mots clés.

Un programme Pascal est composé de deux parties :

- une partie déclarations ;
- une partie instructions.

La partie déclarations permet de déclarer toutes les entités utilisées dans la partie instructions : constantes, types, variables, procédures et fonctions (dans cet ordre). Elle commence par la déclaration du nom du programme.

La partie instructions est constituée par une suite d'instructions séparées par des points-virgules (;). Elle commence par le mot clé **begin** et se termine par le mot clé **end** suivi d'un point (.). Cette partie est aussi appelée le corps du programme.

Voici la structure générale d'un programme Pascal :

```
program nomDuProgramme(input, output);  
partie déclarations  
begin  
suite d'instructions séparées par des points-virgules  
end.
```

2.2 La notion de variable

Il faut être capable de stocker des informations utiles en mémoire centrale durant l'exécution d'un programme. Pour éviter d'avoir à manipuler directement les adresses réelles des cases mémoires où sont stockées ces informations, les langages de programmation offrent la possibilité d'utiliser des variables.

Dans « Le Trésor, dictionnaire des sciences » [6], on peut trouver la définition suivante pour le mot variable :

« Pour l'informaticien, une variable est une chaîne de caractères, c'est-à-dire un mot qui désigne une case de la mémoire. Grâce à ce mot, il est loisible d'accéder autant de fois qu'on le souhaite à cette case et d'en modifier le contenu chaque fois que l'envie nous en prend, sans jamais avoir à se soucier de la localisation exacte de cette case dans la mémoire, autrement dit de son adresse. En somme, les variables sont des poignées amovibles et provisoires facilitant la manipulation des paniers que sont les cases de la mémoire. »

Ce qui revient à dire que le programmeur donne aux variables des noms de son choix. Ces variables désignent une ou plusieurs cases mémoires afin d'y stocker des valeurs particulières sans se préoccuper de l'adresse physique où seront stockées ces valeurs dans la mémoire.

Une variable possède quatre propriétés :

- un nom ;
- une adresse ;
- un type ;
- une valeur.

On peut penser aux variables comme à des boîtes spécifiques dans la mémoire contenant une suite de 0 et de 1. Le nom de la variable désigne la boîte, l'adresse désigne l'emplacement de la boîte dans la mémoire, le type permet de déterminer la taille de la boîte et le moyen de décoder l'information qu'elle contient, enfin la valeur est le déchiffrement des 0 et des 1 contenus dans la boîte en accord avec le type (voir section 4 sur la représentation des données).

2.2.1 Règles de formation des identificateurs

Les noms des variables (ainsi que les noms des programmes, constantes, types, procédures et fonctions) sont appelés des **identificateurs**. Les identificateurs répondent à des règles de formation bien définies :

- un identificateur est une suite de lettres (minuscules 'a'..'z' ou majuscules 'A'..'Z'), de chiffres ('0'..'9') et de caractères de soulignement ('_');
- le premier caractère d'un identificateur doit être une lettre;
- la longueur d'un identificateur est bornée. Cette borne varie en fonction du compilateur utilisé.

Exemples :

- `a0_C20` est un identificateur;
- `0a_C20` n'est pas un identificateur;
- `a*b` n'est pas un identificateur.

Remarques : il est bon de prendre l'habitude de donner des noms significatifs aux variables. De même, évitez le caractère de soulignement qui se confond très facilement avec le signe moins ('-'). Pour former des identificateurs à l'aide de plusieurs mots, écrivez le premier mot en minuscule, puis pour les mots suivants les initiales en majuscule et les autres lettres en minuscule. (Même si le Pascal ne fait pas la distinction entre les minuscules et les majuscules, ce qui n'est pas le cas de tous les langages de programmation). Donc évitez `S` et `P`, utilisez plutôt `sommeDesNotes` et `produitDesEntiers`.

2.2.2 Les types de base et les opérations

Les différents types de base en Pascal sont :

- Les entiers (mot clé `integer`);
- les réels (mot clé `real`);
- les caractères (mot clé `char`);
- les booléens (mot clé `boolean`).

Les entiers, les caractères et les booléens forment les types ordinaux.

Les entiers

Dans beaucoup de langages de programmation, et en particulier en Pascal, les entiers sont représentés sur un mot machine. Sur une machine à n bits on peut représenter 2^n entiers, soit les entiers compris entre -2^{n-1} et $+2^{n-1} - 1$. Donc sur une machine à 16 bits on peut représenter 2^{16} entiers, soit les entiers compris entre -32768 (-2^{15}) et $+32767$ ($+2^{15} - 1$). Les opérations possibles sur les entiers sont :

- l'opposé (opération unaire, notée `-`);
- l'addition (opération binaire, notée `+`);
- la soustraction (opération binaire, notée `-`);
- la multiplication (opération binaire, notée `*`);

- la division entière (opération binaire, notée `div`);
- le reste de la division entière (opération binaire, notée `mod`);

Attention la multiplication n'est pas implicite, le symbole `*` doit toujours être indiqué explicitement entre les deux opérandes.

Exemples :

opération	résultat
5 + 2	7
5 - 2	3
5 * 2	10
5 div 2	2
5 mod 2	1

Les bornes sont données par `-MAXINT-1` et `MAXINT`.

Les réels

La représentation des réels varie suivant les langages de programmation, les machines et les normes utilisés.

Les opérations possibles sur les réels sont :

- l'opposé (opération unaire, notée `-`);
- l'addition (opération binaire, notée `+`);
- la soustraction (opération binaire, notée `-`);
- la multiplication (opération binaire, notée `*`);
- la division (opération binaire, notée `/`);

Exemples :

opération	résultat
5.6 + 2.9	8.5
5.6 - 2.9	2.7
5.6 * 2.9	16.24
5.6 / 2.9	1.9310345

Les caractères

Les variables de type `char` permettent de stocker un seul caractère. Les valeurs de type caractère sont notées entre apostrophe (exemple : `'A'`). Les caractères utilisables sont ceux de la table ASCII (American Standard Code for Information Interchange) étendue. Cette table associe un numéro unique entre 0 et 255 à chaque caractère, ce qui permet d'introduire un ordre sur les caractères. Pour une variable caractère `c` donnée, on peut obtenir le numéro du caractère stocké dans `c` à l'aide de la fonction `ord(c)`. Réciproquement

pour un nombre donné, on peut obtenir le caractère correspondant à l'aide de la fonction `chr(x)`.

Exemples :

opération	résultat
<code>ord('A')</code>	65
<code>chr(65)</code>	'A'

Les apostrophes autour de 'A' servent à distinguer le caractère 'A' de la variable A.

Les booléens

Les variables de type booléen ne peuvent prendre que deux valeurs :

- vrai (mot clé `true`);
- faux (mot clé `false`).

Les opérations possibles sur les booléens sont :

- la négation (opération unaire, mot clé `not`);
- la conjonction (opération binaire, mot clé `and`);
- la disjonction (opération binaire, mot clé `or`).

Pour calculer le résultat de ces opérations on utilise des tables dites tables de vérité :

	<code>not</code>
<code>false</code>	<code>true</code>
<code>true</code>	<code>false</code>

<code>and</code>	<code>false</code>	<code>true</code>
<code>false</code>	<code>false</code>	<code>false</code>
<code>true</code>	<code>false</code>	<code>true</code>

<code>or</code>	<code>false</code>	<code>true</code>
<code>false</code>	<code>false</code>	<code>true</code>
<code>true</code>	<code>true</code>	<code>true</code>

De plus le type booléen est ordonné : `false < true` .

Pour toute variable de type ordinal, il est possible de récupérer son rang (fonction `ord`), d'accéder à son successeur (fonction `succ`) et d'accéder à son prédécesseur (fonction `pred`).

2.2.3 Les constantes

Pascal permet de manipuler des **constantes** de chaque type de base. La valeur d'une constante est donnée au début d'un programme et ne peut être changée par la suite. Le mot clé **const** permet d'introduire la partie de déclaration de constantes (cette partie est optionnelle). Les constantes peuvent ensuite être déclarées de la manière suivante :

```
nomDeConstante = valeur ;
```

L'utilisation de constantes permet entre autres une meilleure lisibilité et une mise à jour plus aisée des programmes. Par exemple, si on doit écrire un programme de calcul d'intérêt qui manipule un nombre fixe d'annuités sur lequel sont calculés ces intérêts et si ce nombre n'est susceptible d'être modifié qu'une fois par décennie, alors le déclarer en constante permet de localiser très rapidement le seul et unique endroit dans le programme où doit être effectué le changement de valeur. De plus la valeur d'une constante peut être exprimée en fonction d'autres constantes déclarées au préalable.

Exemples :

```
const
  nombreDAnnuites = 3;
  nombreDeMois = 12 * nombreDAnnuites;
```

2.2.4 La déclaration des variables

Toutes les variables utilisées dans la partie instructions doivent avoir été déclarées au préalable dans la partie déclarations.

La partie de déclarations de variables est introduite par le mot clé **var**. Cette partie se place après la partie de déclarations des constantes. Elle est également optionnelle, si le programme ne manipule pas de variables (ce qui est très rare).

La déclaration de variables prend ensuite la forme suivante :

```
liste de noms de variables (séparées par des virgules) : type des variables ;
```

où les noms de variables sont des identificateurs correctement formés.

Exemples :

```
var
  x : integer;
  a, b : char;
```

2.2.5 La manipulation de variables

Après sa définition (déclaration + allocation d'une case mémoire automatiquement effectuée lors de l'exécution), une variable possède une valeur qui correspond à l'état de la case mémoire au moment de sa définition.

L'affectation

L'opération d'affectation permet de donner une nouvelle valeur à une variable. La syntaxe de cette opération est la suivante :

nomDeVariable := expression

La sémantique en est : calcul (ou évaluation) de la valeur de l'expression et rangement de cette valeur dans la case mémoire associée à cette variable.

Exemples :

opération	instruction	valeur de la variable
affecter la valeur 1 à la variable x	x := 1	x : 1
affecter la valeur 3 à la variable y	y := 3	y : 3

Ces opérations sont des opérations d'**affectation**. Le symbole d'affectation est :=. Ce qui figure à gauche de ce symbole est obligatoirement un identificateur de variable. La partie droite est une **expression**. La variable et l'expression doivent être de même type (à moins que l'expression ne soit du type **integer** et la variable du type **real** auquel cas la valeur entière est convertie implicitement).

Une expression peut être :

- une valeur constante (exemples : 2, 56.7, 'u' ou true) ;
- une variable ;
- toutes combinaisons d'opérations valides mettant en œuvre des constantes et/ou des variables.

Les expressions sont évaluées de gauche à droite suivant l'ordre de priorité décroissant suivant :

Unaire	Binaire
not	* / div mod and
+ -	+ - or
	< <= = <> >= >

On peut utiliser des parenthèses pour modifier cet ordre de priorité.

Exemples :

opération	valeur
$5 + 4 * 2$	13
$(5 + 4) * 2$	18

On peut modifier la valeur des variables tout au long du programme.

Exemples :

opération	instruction	valeur de la variable
affecter la valeur $x + 1$ à la variable x	$x := x + 1$	x : 2
affecter la valeur $y + x$ à la variable y	$y := y + x$	y : 5

Exemples d'affectations

Une fois définies les variables possèdent une valeur même si elle n'ont fait l'objet d'aucune instruction d'affectation.

```
program Affectations;
var
  a, b, c, d : integer;
begin
  writeln('a = ', a, 'b = ', b, 'c = ', c, 'd = ', d);
  a := 1;
  writeln('a = ', a, 'b = ', b, 'c = ', c, 'd = ', d);
  b := 3;
  writeln('a = ', a, 'b = ', b, 'c = ', c, 'd = ', d);
  c := a + b;
  writeln('a = ', a, 'b = ', b, 'c = ', c, 'd = ', d);
  d := a - b;
  writeln('a = ', a, 'b = ', b, 'c = ', c, 'd = ', d);
  a := a + 2 * b;
  writeln('a = ', a, 'b = ', b, 'c = ', c, 'd = ', d);
  b := c + b;
  writeln('a = ', a, 'b = ', b, 'c = ', c, 'd = ', d);
  c := a * b;
  writeln('a = ', a, 'b = ', b, 'c = ', c, 'd = ', d)
end.
```

La saisie (ou lecture)

La saisie permet d'affecter à une variable une valeur tapée sur le clavier ou de manière plus générale de toute donnée provenant de l'entrée standard.

Syntaxe : `readln(nomDeVariable)`

Les caractères tapés sur le clavier sont codés (ASCII) puis placés dans la mémoire tampon (**buffer**) du clavier (zone mémoire située sur la carte d'extension chargée de la communication avec le clavier). Cette mémoire tampon fonctionne comme une file d'attente (premier arrivé, premier servi). Deux données successives sont séparées par un retour chariot (touche «*Entrée*»), un espace ou une tabulation (touche «*Tab*»). Lorsque la mémoire tampon est vide, l'instruction `readln` attend (c'est-à-dire que le processeur est bloqué jusqu'à la fin de l'instruction) qu'une donnée arrive dans cette mémoire tampon. Lorsque la mémoire tampon contient une (ou plusieurs) données, l'instruction `readln` prend la première, la supprime du tampon, la traduit éventuellement au bon format et place la valeur ainsi obtenue dans la variable. La saisie est donc une affectation, la valeur stockée n'étant pas le résultat de l'évaluation d'une expression mais une donnée demandée à l'utilisateur.

On peut également mettre entre les parenthèses, plusieurs noms de variables, séparés par des virgules.

Syntaxe : `readln(liste de noms de variables séparés par des virgules)`

La valeur de chaque variable sera alors saisie au clavier.

L'exécution de l'instruction `readln` consiste à déplacer le curseur de lecture immédiatement après le prochain symbole de fin de ligne (symbole renvoyé par «*Entrée*» lorsque le flux d'entrée est le clavier).

Il existe également une variante de l'instruction `readln` désignée par le mot clé `read` (voir TP N°2).

L'exécution de l'instruction `read(x)` consiste à lire, à partir de la position courante du curseur de lecture, une suite de caractères ayant la syntaxe des constantes du type de la variable x et à déplacer le curseur au fur et à mesure de la lecture.

Dans le cas de la lecture d'une variable de type `integer` ou `real`, le symbole de fin de ligne et le caractère de tabulation sont assimilés au (caractère) blanc. Les blancs placés en tête sont ignorés. C'est une erreur à l'exécution si, après avoir sauté les blancs de tête, on trouve un caractère ne correspondant pas à la syntaxe des constantes de type `integer` ou `real`. Sinon, la lecture cesse dès que le curseur de lecture précède un blanc ou un caractère qui, ajouté aux caractères déjà lus pour former la constante, ne correspond pas à la syntaxe des constantes de type `integer` ou `real`. C'est une erreur à l'exécution si la valeur de la constante qui a été lue n'est pas compatible pour l'affectation avec le type de la variable.

L'affichage (ou écriture)

L'affichage permet d'écrire une valeur sur l'écran ou plus généralement sur la sortie standard. Cette valeur peut être le contenu d'une variable comme le résultat du calcul d'une expression.

Syntaxe : `writeln(expression)`

La sémantique en est la suivante, l'expression est évaluée et la valeur de son résultat est affiché à l'écran.

Remarque : le résultat affiché à l'écran est perdu. S'il devait être réutilisé par la suite il serait préférable de le stocker dans une variable et ensuite d'afficher cette variable :

```
x := expression ;  
writeln(x) ;
```

Il est possible d'écrire plusieurs expressions avec une seule instruction `writeln` en mettant entre parenthèse la liste des expressions séparées par des virgules.

Syntaxe : `writeln(liste d'expressions séparées par des virgules)`

L'exécution de l'instruction `writeln` consiste à placer le symbole de fin de ligne sur le flux de sortie (ce qui provoque un passage à la ligne lorsque le flux de sortie est l'écran).

Il existe également une variante de l'instruction `writeln` qui est désignée par le mot clé `write` qui fonctionne comme l'instruction `writeln` sans faire passer le curseur à la ligne après son exécution.

L'exécution de l'instruction `write(e)` consiste à évaluer l'expression e , puis à écrire sur le flux de sortie une constante qui dénote sa valeur.

Par défaut, une constante de type `real` est affichée sous la forme

$$s.c.c\cdots cE s.c.c\cdots c$$

où chaque s signifie un blanc, $+$ ou $-$, et un c un chiffre.

Il est possible de spécifier le format d'affichage en donnant une instruction de la forme

$$\text{writeln}(e : m)$$

pour afficher la valeur de e dans une plage de m caractères en complétant à gauche par des blancs si nécessaire. Si e est de type `real`, on peut écrire du

$$\text{writeln}(e : m : n)$$

pour forcer l'affichage de n chiffres après la virgule.

2.3 L'exécution d'un programme

Après avoir tapé un programme Pascal (à l'aide d'un éditeur de texte, intégré ou non à un système de développement) et sauvegardé le texte de ce programme dans un fichier, il doit alors être compilé. Le compilateur procède à deux phases d'analyse (lexicale et syntaxique) avant de produire un fichier exécutable. La phase d'analyse lexicale (de lexique) vérifie que chaque mot apparaissant dans le programme est correct. La phase d'analyse syntaxique (de syntaxe) vérifie que l'organisation de ces mots entre eux est correcte.

Un fois cette étape de compilation franchie, une étape d'édition de liens permet de produire un fichier exécutable. Il est ensuite possible d'exécuter ce programme (soit en cliquant sur l'icône correspondant, soit en tapant son nom à l'invite du système d'exploitation).

Remarque : les compilateurs vérifient la syntaxe des programmes, il n'en vérifient pas la sémantique.

2.3.1 Un exemple de programme Pascal

Nous allons écrire un programme Pascal très simple qui saisit la valeur d'un entier, en calcule son carré et son cube et affiche la valeur de ce dernier.

```

program Cube(input, output);
var
    unEntier, sonCarre, sonCube : integer;
begin
    write('Entrez un entier : ');
    readln(unEntier);
    sonCarre := unEntier * unEntier;
    sonCube := sonCarre * unEntier;
    writeln('Le cube de ', unEntier, ' est ', sonCube)
end.

```

Au moment où l'utilisateur spécifie au système d'exploitation d'exécuter un programme, le système attribue une partie (ou **segment**) de la mémoire centrale au **processus** qui va exécuter le programme. Ce segment est découpé en deux zones : une zone code qui contient les instructions du programme et une zone mémoire qui contient les valeurs des variables que le programme manipule. Ensuite le programme exécute une à une les instructions de la zone code.

Le programme **Cube** réserve en mémoire trois cases pour stocker des variables entières nommées **unEntier**, **sonCarre** et **sonCube**. Ensuite il affiche le message 'Entrez un entier : ' à l'écran en laissant le curseur à la fin de cette ligne. Puis il prend une valeur dans le tampon du clavier (cette valeur devra être tapée par l'utilisateur du programme). Ensuite il calcule le produit **unEntier * unEntier** et stocke la valeur du résultat dans la variable **sonCarre**, puis il calcule le produit **sonCarre * unEntier** et stocke la valeur du résultat dans la variable **sonCube**. Enfin il affiche le message 'Le cube de ', suivi de la valeur de la variable **unEntier**, suivi du message ' est ', suivi de la valeur de la variable **sonCube**.

Si l'utilisateur rentre la valeur 5 pour **unEntier**, la variable **sonCarre** prendra alors la valeur 25 après l'exécution de l'instruction **sonCarre := unEntier * unEntier** et la variable **sonCube** prendra la valeur 125 après l'exécution de l'instruction **sonCube := sonCarre * unEntier**.

On vient de voir un principe fondamental de l'exécution des programmes qui est le principe de **séquentialité**, c'est-à-dire que toutes les instructions du programme sont exécutées dans l'ordre où elles apparaissent et que l'exécution d'une instruction ne commence que lorsque l'instruction précédente est complètement terminée.

Remarque : il n'y a *a priori* aucun rapport entre le nom d'une variable et l'utilisation qui peut en être faite. Par exemple le programme précédent fonctionnerait exactement de la même manière si on remplace toutes les occurrences de **sonCarre** par **teeShirt**.

Il est à noter que le programme **CubeBis** suivant a exactement le même comportement pour l'utilisateur que le programme **Cube**.

```

program CubeBis(input, output);
var
  unEntier : integer;
begin
  write('Entrez un entier : ');
  readln(unEntier);
  writeln('Le cube de ', unEntier, ' est ',
          unEntier * unEntier * unEntier)
end.

```

Les programmes peuvent être écrits sur une seule ligne cependant ce type de présentation mène à des programmes illisibles. On imposera ici qu'il y ait au plus une instruction par ligne. L'**indentation** consiste à espacer les lignes de code par rapport au bord gauche de la fenêtre de saisie du texte. Observez cela dans l'exemple précédent. Cette indentation ne doit pas (bien que cela ne change rien au sens du programme) être menée à la légère. La taille de l'espacement doit être proportionnelle au niveau d'imbrication des instructions du programme. Ainsi, et même sans lire le code, on visualise très rapidement la structure du programme. La plupart des éditeurs de texte offrent des facilités pour réaliser une bonne indentation.

2.4 Les commentaires

L'idéal pour pouvoir comprendre rapidement un programme c'est qu'il soit annoté de commentaires (significatifs c'est encore mieux). En effet quasiment tous les langages de programmation permettent de placer du texte dans un programme sans qu'il agisse sur l'exécution. Ce texte sert seulement à faciliter la lecture du programme. Pour placer un commentaire dans un programme Pascal, il suffit de l'encadrer le texte entre accolades (`{` et `}`) ou entre parenthèses-étoiles (`(* et *)`).

Exemple :

```

v := 4/3*pi*r*r*r; { Calcul du volume d'une sphère
                   de rayon r }
{ un commentaire
  peut s'étendre
  sur plusieurs
  lignes. }

```

2.5 Les fonctions

Durant l'écriture d'un programme, il est fréquent d'utiliser la notion de fonction comme elle est définie en mathématiques : étant données une

ou plusieurs valeurs, appliquer une suite de transformations à ces valeurs jusqu'à obtenir un résultat. Ces transformations se traduisent en programmation par des instructions. Si, plusieurs fois au cours d'un programme nous avons besoin du résultat d'une fonction appliquée à plusieurs valeurs d'entrée différentes, il est fastidieux pour le programmeur de réécrire à chaque fois ces instructions. Il vaut beaucoup mieux les écrire une fois pour toute ailleurs, et y faire appel quand on en a besoin. C'est également plus agréable pour la personne qui lit le programme, car cela introduit une structuration qui permet une compréhension plus simple de l'ensemble (selon le principe que pour comprendre un ensemble complexe, il est plus simple d'en comprendre d'abord séparément les composantes). C'est pourquoi a été introduite, dans tous les langages de programmation, la notion de sous-programme.

Dans la suite nous étudions l'exemple de fonction suivant :

$$f(x, n) = 3x + 4x/n + 5n$$

où f est une fonction réelle à deux variables, x réel et n entier.

Une fonction peut être vue comme un programme autonome, à qui on transmet des valeurs et qui retourne un résultat.

Dans le cadre de ce cours, la philosophie est de déclarer une fonction lorsqu'on peut être amené à répéter un calcul dont les entrées sont clairement identifiées et qui renvoie un seul et unique résultat.

2.5.1 Déclaration d'une fonction

En Pascal, une fonction est un objet qui doit être déclaré au même titre qu'une variable. Cette déclaration, qui se trouve dans la partie déclarative du programme après la déclaration des variables, contient en fait tous les renseignements nécessaires à la connaissance de la fonction. Celle-ci étant un sous-programme, sa syntaxe est proche de celle du programme principal.

```
function nomDeLaFonction(liste de paramètres formels) :
    type du résultat ;
    partie déclarations
begin
    suite d'instructions séparées par des points-virgules
    nomDeLaFonction := expression
end ;
```

2.5.2 Liste de paramètres formels

Les valeurs données en entrée d'une fonction peuvent être de différents types, qu'il est essentiel de spécifier au compilateur. Celui-ci contrôle ainsi les possibles incohérences. De plus, ces valeurs doivent être associées à des

identificateurs pour pouvoir être utilisées dans le corps de la fonction : un tel identificateur est appelé paramètre formel de la fonction, par opposition avec la valeur qui servira effectivement au calcul et qui sera donnée lors de l'appel de la fonction : paramètre réel. Les paramètres formels, munis de leur type, sont séparés par des « ; ».

2.5.3 La partie déclarations

Si le calcul du résultat de la fonction est complexe, il peut être utile de définir des objets locaux comme des variables intermédiaires, des constantes, ou même d'autres fonctions plus simples. La définition de la fonction comprend donc une partie déclarative, exactement comme le programme principal. La déclaration de variables locales peut poser des problèmes. Une telle variable a une existence qui est locale au sous-programme qui l'a déclarée. La valeur qu'elle peut avoir lors d'un appel au sous-programme est perdue lors du retour au programme appelant et ne peut être récupérée lors d'un appel ultérieur. Par contre, une variable déclarée par le programme principal (variable globale) est accessible à l'intérieur de la fonction. Sa valeur peut même y être modifiée, cependant l'utilisation de variable globale dans une fonction est fortement déconseillée. Il est autorisé d'avoir une variable locale du même nom qu'une variable globale, dans ce cas, seule la variable locale est accessible lors de l'exécution de la fonction. La variable locale masque la variable globale. Durant l'exécution de la fonction, la valeur de la variable globale n'est pas altérée. De manière à éviter toute ambiguïté, on interdira cette pratique dans le cadre de ce cours.

2.5.4 Le résultat

La fonction doit nécessairement contenir une instruction de la forme `nomDeLaFonction := expression`. Cette instruction permet de fournir le résultat de la fonction au programme appelant. On imposera ici, que cette instruction constitue la dernière ligne de la fonction.

2.5.5 Un exemple de fonction

```
function f(x : real; n : integer) : real;
var
  xcarre : real;
begin
  xcarre := x * x;
  f := 3 * xcarre + 4*x/n + 5 * n
end;
```

2.5.6 Appel d'une fonction

Pour calculer le résultat d'une fonction pour certaines valeurs, le programme appelant doit appeler la fonction en lui transmettant ces valeurs. La syntaxe est `nomDeLaFonction(liste des paramètres réels)` qui doit apparaître dans une expression. La liste des paramètres réels est constituée d'une liste d'expressions séparées par des virgules. Les paramètres réels doivent correspondre en type et en nombre aux paramètres formels. Il est important de respecter l'ordre des paramètres formels. La valeur de chaque expression est calculée et devient ainsi la valeur du paramètre formel correspondant. Si le type de l'expression ne correspond pas au type du paramètre formel, une erreur est détectée lors de la compilation.

Exemples d'appel :

```
a := f(b, 4);
a := f(14.5, n) + 4 * f(b, m + 12);
writeln('La valeur est : ', f(55.12, 42):0:4);
```

avec `a` et `b` des variables déclarées de type `real`, `m` et `n` des variables déclarées de type `integer`.

2.5.7 Fonctions prédéfinies

Il existe une bibliothèques de fonctions prédéfinies appelables dans n'importe quel programme.

Exemples de fonctions prédéfinies :

```
abs(x : real) : real { retourne la valeur absolue de x }
round(x : real) : real { retourne l'entier le plus proche de
                        x }
sqrt(x : real) : real { retourne la racine carrée de x }
odd(i : integer) : boolean { retourne true si i est impair,
                             false sinon }
```

2.6 L'instruction alternative

Il est souvent utile d'effectuer un choix en fonction du résultat d'un test ou d'une condition. Il est possible dans tous les langages de programmation d'effectuer un groupe d'instructions en fonction du résultat d'un test. L'instruction permettant de réaliser cela, appelée instruction alternative ou instruction conditionnelle, peut généralement prendre deux formes.

Syntaxe :

```

if condition then begin
    groupe d'instructions 1
end

```

et

```

if condition then begin
    groupe d'instructions 1
end
else begin
    groupe d'instructions 2
end

```

La condition est en fait une expression de type booléen. Cette expression est évaluée, si sa valeur est vrai alors le groupe d'instructions 1 est exécuté. Si sa valeur est faux, le groupe d'instructions 2 est exécuté lorsqu'il est présent.

La condition résulte dans la majorité des cas d'une ou plusieurs comparaisons.

2.6.1 Les comparaisons

Les symboles de comparaisons sont les suivants :

symbole Pascal	symbole mathématique
<	<
<=	≤
=	=
<>	≠
>=	≥
>	>

Il est possible de combiner plusieurs comparaisons à l'aide d'opérateurs booléens (sans oublier de parenthéser car les opérateurs booléens sont prioritaires sur les opérateurs de comparaisons).

En particulier les encadrements ne peuvent pas être écrits directement en Pascal, ils doivent être réalisés à l'aide de deux comparaisons connectées par l'opérateur **and**.

Exemple :

L'encadrement mathématique

$$0 \leq x \leq 15$$

se traduit en Pascal par l'expression booléenne

`(0 <= x) and (x <= 15)`

Nous allons voir comment calculer le maximum entre deux variables `x` et `y` d'un même type ordinal. Supposons que `maximum` soit une variable de même type que `x` et `y`, alors l'instruction suivante permet de stocker $\max\{x, y\}$ dans `maximum`.

```
if x > y then begin
    maximum := x
end
else begin
    maximum := y
end
```

Supposons maintenant que l'on veuille calculer à la fois le maximum et le minimum de `x` et `y`. Alors l'instruction suivante réalise cela.

```

if x > y then begin
    maximum := x;
    minimum := y
end
else begin
    maximum := y;
    minimum := x
end
end

```

Remarque : le point-virgule est un séparateur d'instruction. L'instruction `if then else` constitue une seule instruction, donc il ne faut jamais de point virgule avant un `else`.

2.6.2 Évaluation des expressions booléennes

- Il existe deux types d'évaluation des expressions booléennes :
- l'évaluation complète consiste à évaluer tous les opérandes des expressions booléennes ;
 - l'évaluation paresseuse consiste à stopper l'évaluation d'une connexion dès qu'un opérande a la valeur `false` (puisque `false and x` a la valeur `false` quelque soit x) et de stopper l'évaluation d'une disjonction dès qu'un opérande a la valeur `true` (puisque `true or x` a la valeur `true` quelque soit x).

En Pascal standard l'évaluation des expressions booléennes est complète, toutefois la plupart des implantations spécifiques de compilateurs Pascal réalise l'évaluation des expressions booléennes de manière paresseuse.

2.6.3 Tests imbriqués

Il est bien sur possible d'imbruquer des instructions `if then else`. Il faut alors faire bien attention qu'un `else` se rapporte au dernier `if` rencontré.

Exemple :

```

if x >= y then begin
    if x = y then begin
        writeln(x, '=', y)
    end
else begin
    writeln(x, '<', y)
end
end
end

```

Que se passe-t-il lorsque $x = 5$ et $y = 4$?

Remarque : l'indentation ne fait pas la structuration.

2.7 L'instruction itérative

L'instruction itérative permet de répéter un certain nombre de fois l'exécution d'une suite d'instructions sous une certaine condition. De façon imagée, on appelle **boucle** cette méthode permettant de répéter l'exécution d'un groupe d'instructions.

Syntaxe :

```
while condition do begin
  groupe d'instructions
end
```

Tant que le résultat de l'évaluation de la condition est vrai le groupe d'instructions est exécuté. Lorsque la condition devient fausse, on passe alors, en séquence, à l'exécution de l'éventuelle instruction suivante. Il est à noter que la condition peut être fausse dès sa première évaluation auquel cas le groupe d'instructions n'est jamais exécuté. Bien évidemment pour que cela présente un quelconque intérêt ou pour éviter les boucles infinies, le groupe d'instructions doit agir sur, au moins, une des variables apparaissant dans la condition.

Prenons un exemple pour comprendre l'intérêt d'une telle instruction. Essayons d'afficher la valeur des dix premiers entiers strictement positifs. Une première solution est la suivante :

```
program QuiCompte(input, output);
begin
  writeln(1);
  writeln(2);
  writeln(3);
  writeln(4);
  writeln(5);
  writeln(6);
  writeln(7);
  writeln(8);
  writeln(9);
  writeln(10)
end.
```

Cette solution devient vite fastidieuse lorsque l'on désire obtenir plus d'entiers (même si les éditeurs de texte offrent des possibilités intéressantes), mais elle s'avère complètement inadaptée lorsque l'utilisateur du programme doit spécifier le nombre d'entiers souhaité. La solution est alors d'employer l'instruction itérative **while do**.

```

program QuiCompteMieux(input, output);
var
  nombreDEntiers, compteur : integer;
begin
  write('Entrez le nombre d''entiers : ');
  readln(nombreDEntiers);
  compteur := 1;
  while compteur <= nombreDEntiers do begin
    writeln(compteur);
    compteur := compteur + 1
  end
end.

```

Si la valeur de la condition est fausse dès le départ alors le groupe d'instructions n'est jamais exécuté.

Exemple :

```

program QuiFaitRien(input, output);
var
  compteur : integer;
begin
  compteur := 1;
  while compteur < 0 do begin
    writeln(compteur);
    compteur := compteur + 1
  end
end.

```

Par contre si la valeur de la condition est vraie et que le groupe d'instructions ne permet pas d'altérer cette valeur alors le groupe d'instructions est exécuté à l'infini : on a alors affaire à une boucle infinie.

Exemple :

```

program QuiBoucle(input, output);
var
  compteur : integer;
begin
  compteur := 1;
  while compteur > 0 do begin
    writeln(compteur);
    compteur := compteur + 1
  end
end.

```

Par conséquent, lorsqu'un utilise un environnement de programmation, tel Turbo Pascal, il est impératif d'effectuer une sauvegarde des fichiers sources avant toute exécution. Il est, par ailleurs, fortement conseillé d'effectuer des sauvegardes régulièrement.

2.7.1 Accumulateur

Une technique classique consiste, on vient de le voir, à utiliser un compteur. Une autre technique classique consiste à utiliser un accumulateur. Prenons l'exemple d'une fonction qui calcule la somme des n premiers entiers positifs.

```
function Somme(n : integer) : integer;
var
  compteur, accumulateur : integer;
begin
  compteur := 0;
  accumulateur := 0;
  { accumulateur = somme de 0 a compteur
    et compteur = 0 }
  while compteur < n do begin
    { accumulateur = somme de 0 a compteur
      et compteur < n }
    compteur := compteur + 1;
    { accumulateur = somme de 0 a compteur - 1
      et compteur <= n }
    accumulateur := accumulateur + compteur
    { accumulateur = somme de 0 a compteur
      et compteur <= n }
  end;
  { accumulateur = somme de 0 a compteur
    et compteur = n }
  Somme := accumulateur
end;
```

2.8 Les tableaux

Les tableaux permettent de manipuler un nombre fini de composants de même type. Chaque élément du tableau peut être manipulé comme une variable en étant désigné non pas par un nom particulier mais par sa position dans le tableau. Cette position ou **indice** est de **type** intervalle.

2.8.1 Le type intervalle

Un type peut être défini à partir de n'importe quel type. La déclaration de type est introduite par le mot clé `type` et intervient entre la déclaration des constantes et celles des variables. Les types ainsi définis peuvent servir à déclarer des variables.

La syntaxe de définition d'un type est la suivante :

```
nomDuType = définition du type ;
```

Exemple :

```
type
    nombreEntier = integer;
var
    x : nombreEntier;
```

En particulier, un type intervalle peut être défini à partir de n'importe quel type ordinal, et donc des types `integer` et `char`. Les types intervalles sont très utiles pour écrire des programmes rigoureux puisqu'il y a un contrôle plus étroit sur le domaine de variation des variables.

Syntaxe : `nomDuTypeIntervalle = borneInférieure..borneSupérieure ;`

`borneInférieure` et `borneSupérieure` doivent être des constantes (et non pas des variables) d'un même type ordinal. Une variable du type intervalle peut alors prendre toutes les valeurs entre `borneInférieure` et `borneSupérieure`.

Exemple :

```
type
    jourDeLaSemaine = 1..7;
var
    Jour : jourDeLaSemaine;
```

2.8.2 Déclaration des tableaux

La déclaration d'un tableau s'effectue à l'aide du mot clé `array` :

```
nomDuTableau : array [typeInter1,typeInter2,...,typeIntern] of
type des éléments du tableau ;
```

où les `typeInter i` , pour $1 \leq i \leq n$ sont des types intervalle.

Il y a autant de type intervalle que de dimension au tableau.

Exemples :

```
var
  t1 : array [1..10] of char;
  t2 : array ['A'..'J'] of integer;
  matriceCarre : array [1..10,1..10] of real;
```

t1 est un tableau de dix caractères : t1[1], t1[2], t1[3], t1[4], t1[5], t1[6], t1[7], t1[8], t1[9], t1[10] sont des variables de type char .

t2 est un tableau de dix entiers : t2['A'], t2['B'], t2['C'], t2['D'], t2['E'], t2['F'], t2['G'], t2['H'], t2['I'], t2['J'] sont des variables de type integer .

matriceCarre est un tableau de 100 (10 × 10) réels :

les matriceCarre[i,j] pour $1 \leq i \leq 10$ et $1 \leq j \leq 10$ sont des variables de type real .

Chacun des éléments d'un tableau peut être manipulé comme n'importe quelle variable. En particulier il peut apparaître à gauche d'un symbole d'affectation ou dans une expression.

Exemple :

```
t1[3] := t1[2] + t1[1]
```

Attention : il ne faut pas confondre le type des indices du tableau et le type des éléments du tableau.

À condition qu'ils aient été déclarés de même type, deux tableaux peuvent être affectés globalement.

Exemple :

```
u, v : array [inf..sup] of integer;
```

Alors l'instruction

```
u := v
```

est correcte.

2.8.3 Manipulation des tableaux

On suppose pouvoir disposer des déclarations suivantes :

```
const
  borneInf = ?;
  borneSup = ?;
```

```

type
  intervalle = borneInf..borneSup;
  tableau = array[intervalle] of integer;
var
  t : tableau;

```

On suppose que `borneInf` et `borneSup` sont des constantes d'un type ordinal. On définit une variable `i` de même type que `borneInf` et `borneSup`.

Alors la lecture de tous les éléments du tableau `t` peut s'effectuer de la manière suivante :

```

i := borneInf;
while i <= borneSup do begin
  read(t[i]);
  i := succ(i)
end

```

L'écriture des valeurs de tous les éléments du tableau `t` peut être effectuée comme suit :

```

i := borneInf;
while i <= borneSup do begin
  write(t[i]);
  i := succ(i)
end

```

La recherche de l'indice d'un élément `x` est possible avec le code qui suit :

```

trouve := false;
i := borneInf;
while not trouve and (i <= borneSup) do begin
  if t[i] = x then begin
    trouve := true
  end
  else begin
    i := succ(i)
  end
end
{ trouve and t[i] = x } or (i > borneSup) }
if trouve then begin { t[i] = x }
  writeln(i)
end
else begin { i > borneSup }
  writeln(-1)
end

```

Le programme complet pour la lecture et l'écriture des n^2 éléments d'un tableau d'entiers à deux dimensions suit.

```
program TableauADeuxDimensions (input, output);
const
  TAILLE = 10;
type
  intervalle = 1..TAILLE;
  tableau = array [intervalle, intervalle] of integer;
var
  t : tableau;
  i, j, n : integer;
begin
  read(n);
  i := 1;
  while i <= n do begin
    j := 1;
    while j <= n do begin
      read(t[i, j]);
      write(t[i, j]:3);
      j := j + 1
    end;
    i := i + 1;
    writeln
  end
end.
```

2.8.4 Le type chaînes de caractères

Le type chaînes de caractères, introduit par le mot clé **string** permet de manipuler des mots. Lors de la déclaration, il faut spécifier la longueur maximale du mot.

Syntaxe : `nomDuMot : string[dimension];`

où dimension est un entier compris entre 1 et 255.

On peut concevoir les variables de type **string** comme des tableaux de caractères : il est possible d'accéder à chaque lettre du mot en donnant sa position dans le mot.

Des fonctions spécifiques permettent de manipuler les variables de type **string** telle la fonction `length(s)` qui permet de connaître la longueur de la chaîne de caractères `s`.

Exemple :

```
mot : string[10];

mot[1], mot[2], mot[3], mot[4], mot[5], mot[6], mot[7], mot[8],
mot[9] et mot[10] sont des variables de type char .
Les constantes de type string apparaissent entre apostrophes.
```

Exemples :

```
const
    chaineSimple = 'Bonjour tout le monde !';
    chaineAvecApostrophe = 'L''apostrophe';
```

L'apostrophe étant le délimiteur des chaînes de caractères, pour en inclure une dans une chaîne, il faut la faire précéder d'une autre apostrophe.

Les fonctions prédéfinies

La fonction `length(s)` donne la longueur de la chaîne de caractères s (à ne pas confondre avec sa dimension).

La fonction `concat(s1, s2, s3, ..., sn)` concatène les chaînes de caractères spécifiées s_1, s_2, \dots, s_n en une seule et même chaîne.

Syntaxe :

```
s := 'bonjour';
m := length(s);    { m = 7 }
```

On peut se passer de cette fonction grâce à l'opérateur `+` : $s_1 + s_2 + s_3 + \dots + s_n$.

Syntaxes :

```
s1 := 'hello';
s2 := 'world';
s := concat(s1, ' ', s2); { s = 'hello world' }
s := s1 + ' ' + s2;      { s = 'hello world' }
```

La fonction `copy(s, i, j)` retourne la sous-chaîne de caractère $s[i..i+j-1]$ (les j caractères de s à partir de la position i incluse).

Syntaxe :

```
s := 'informatique';
t := copy(s, 3, 6);    { t = 'format' }
```

La fonction `pos(x, y)` retourne la position de la première occurrence de la chaîne x dans la chaîne y . Si x n'apparaît pas dans y , alors la fonction `pos(x, y)` retourne 0.

Syntaxe :

```
s := 'informatique';  
i := pos('format', s); { i = 3 }  
j := pos('math', s);   { j = 0 }
```

Recherche de mot

Le code suivant permet de rechercher les indices de toutes les occurrences d'un mot x dans un texte y .

```
m := length(x);  
n := length(y);  
j := 1;  
while j <= n - m + 1 do begin  
  i := 1;  
  egalite := true;  
  while (i <= m) and egalite do begin  
    if x[i] = y[i + j - 1] then begin  
      i := i + 1  
    end  
    else begin  
      egalite := false  
    end  
  end;  
  if egalite then begin  
    writeln(j)  
  end;  
  j := j + 1  
end
```

2.9 Organisation de la mémoire

En Turbo Pascal, une fois compilé, un programme gère la mémoire de la façon suivante :

Zone de code	toutes les instructions du programme	64 ko
Segment de données	variables du programme	64 ko
Pile (<i>Stack</i>)	paramètres formels et variables locales des sous-programmes	16 ko
Tas (<i>Heap</i>)	variables dynamiques	640 ko

Si une variable globale est du type suivant :

```
array[1..66000] of char;
```

alors une erreur numéro 49 (*Data Segment too large*) apparaîtra lors de la compilation.

Lors de l'appel d'une fonction les valeurs des paramètres réels sont recopiées dans les paramètres formels. Si une variable locale ou un paramètre formel d'une fonction est du type suivant :

```
array[1..17000] of char ;
```

alors une erreur numéro 202 (*Stack overflow error*) apparaîtra lors de l'exécution du programme.

Il existe des moyens pour modifier les valeurs par défaut des tailles du segment des données, de la pile et du tas.

Tous les compilateurs Pascal gère la mémoire de manière similaire.

2.10 Un exemple (presque) complet

On veut écrire un programme qui permet de gérer un stock de cahiers et un stock de ramettes.

Il faut pouvoir ajouter ou retirer des cahiers et des ramettes.

```
program Stock;
var
  quantite, ramettes, cahiers : integer;
  article, encore, operation : char;

function QuestionReponse(question : string;
                          choix1, choix2 : char) : char;
const
  diff = ord('a') - ord('A');
var
  reponse : char;
begin
  write(question, ' (', choix1, ' ou', choix2, ') : ');
  readln(reponse);
  while (reponse <> choix1) and
        (reponse <> chr(ord(choix1) + diff)) and
        (reponse <> choix2) and
        (reponse <> chr(ord(choix2) + diff)) do begin
    write(question, ' (', choix1, ' ou', choix2, ') : ');
    readln(reponse);
  end;
  QuestionReponse := reponse
end;
```

```

function SaisieEntierPositif(question : string) : integer;
var
    reponse : integer;
begin
    write(question, ' : ');
    read(reponse);
    while reponse < 0 do begin
        write(question, ' : ');
        readln(reponse);
    end;
    SaisieEntierPositif := reponse
end;

begin
    { Initialisations }
    ramettes := SaisieEntierPositif('Quantite de ramettes');
    cahiers := SaisieEntierPositif('Quantite de cahiers');
    { Gestion }
    encore := 'o';
    while (encore = 'o') or (encore = 'O') do begin
        { Saisie de l'article }
        article := QuestionReponse('Quel article', 'R', 'C');
        { Saisie de la quantite }
        write('Quelle quantite : ');
        readln(quantite);
        { Saisie de l'operation a effectuer }
        operation := QuestionReponse('Ajout ou retrait',
                                     'A', 'R');
        { Traitement de l'operation }
        if (article = 'R') or (article = 'r') then begin
            { ramettes }
            if (operation = 'A') or (operation = 'a') then begin
                ramettes := ramettes + quantite
            end
            else begin
                if ramettes >= quantite then begin
                    ramettes := ramettes - quantite
                end
                else begin
                    writeln('Retrait de ramettes impossible')
                end
            end
        end
    end
    else begin { cahiers }

```

```

    if (operation = 'A') or (operation = 'a') then begin
        cahiers := cahiers + quantite
    end
    else begin
        if cahiers >= quantite then begin
            cahiers := cahiers - quantite
        end
        else begin
            write('Retrait de cahiers impossible')
        end
    end
end;
{ Encore un article a traiter ? }
encore := QuestionReponse('Encore un article a traiter',
                           'O', 'N');

end;
{ Stocks finaux }
writeln('Il reste', rammettes, ' rammettes et ',
        cahiers, ' cahiers.')
end.

```

Chapitre 3

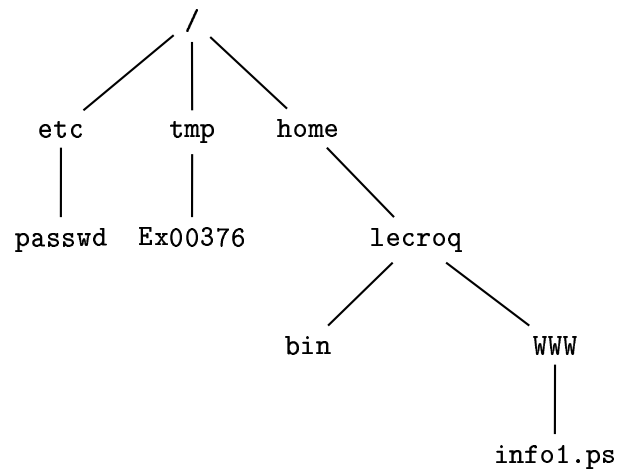
L'organisation des données

On effectue souvent l'analogie entre l'organisation des données en informatique sur un support magnétique et celle dans un bureau sur un support papier :

papier	magnétique
armoire	disque
malette	disquette
tiroir	répertoire
chemise	sous-répertoire
sous-chemise	sous-sous-répertoire
⋮	⋮
fiche	fichier
symbole	bit

Cette organisation est le plus souvent de type arborescent. Il est commode de donner une représentation graphique de l'organisation des données d'un disque où d'une disquette.

Exemple :



Le répertoire racine ou racine (*root*), souvent notée / ou \ en fonction du système d'exploitation, est alors positionnée en haut de l'arbre. Les répertoires et les fichiers directement stockés à la racine sont positionnés au niveau du dessous. Les sous-répertoires et les fichiers contenus dans les répertoires directement contenus dans la racine apparaissent au troisième niveau, et ainsi de suite... Seuls les fichiers contiennent les informations, ils ne peuvent cependant contenir ni répertoire ni fichiers. Ils constituent obligatoirement des nœuds terminaux (ou feuilles) de l'arbre.

Remarque : un fichier est obligatoirement contenu dans un répertoire.

Lorsque l'on manipule des informations sur une machine, on se place à un nœud de l'arborescence. Le nœud où on se trouve ne peut être qu'un répertoire et non pas un fichier. On a alors la notion de **répertoire courant**. On peut alors accéder directement à toutes les informations contenues dans ce répertoire. Pour accéder à des informations contenues dans un autre répertoire ou pour s'y déplacer, on fait référence à cet autre répertoire en utilisant soit un **chemin relatif** soit un **chemin absolu**.

Un chemin relatif décrit tous les répertoires intermédiaires entre le répertoire courant et les informations à atteindre. En règle générale, le point (.) désigne le répertoire courant et les deux points (..) désigne le répertoire parent du répertoire courant.

Un chemin absolu décrit tous les répertoires intermédiaires entre le répertoire racine et les informations à atteindre.

Exemple :

Si on suppose que, dans l'arborescence ci-dessus, le répertoire `lecroq` est le répertoire courant, alors pour faire référence au fichier `info1.ps` on a le choix entre le chemin relatif :

`WWW/info1.ps`

et le chemin absolu :
`/home/lecroq/WWW/info1.ps.`

Pour faire référence au fichier Ex00376 on a le choix entre le chemin relatif :

`../../tmp/Ex00376`
et le chemin absolu
`/tmp/Ex00376.`

En règle générale on choisira l'expression la plus concise pour faire référence à un endroit de l'arborescence.

Chapitre 4

Représentation des données

La représentation en base décimale d'un entier naturel s'exprime comme suit :

$$a_n a_{n-1} \cdots a_1 a_0 = \sum_{i=0}^n a_i \times 10^i$$

avec $0 \leq a_i \leq 9$ pour $0 \leq i \leq n$.

On peut généraliser en exprimant la représentation en base B , pour un entier $B \geq 2$, d'un entier naturel comme suit :

$$\sum_{i=0}^n a_i \times B^i$$

avec $0 \leq a_i \leq B - 1$ pour $0 \leq i \leq n$.

La représentation d'un nombre en base B est unique (si l'on omet les 0 non significatifs à gauche).

En informatique les bases les plus utilisées sont 2, 8 et 16.

En base B on utilise B symboles différents.

base	dénomination	symboles
2	binaire	0,1
8	octale	0,1,2,3,4,5,6,7
10	décimale	0,1,2,3,4,5,6,7,8,9
16	hexadécimale	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

Les conversions d'une base à l'autre sont fréquentes car la représentation externe des données est généralement en octal, décimal ou hexadécimal alors que la représentation interne est en binaire.

Il existe une représentation interne de :

- 2003;
- 20.03;
- '2003';
- 'Deux mille trois'.

Chaque donnée est représentée par une suite de bits.

Les représentations internes sont de longueurs finies et fixées, puisque les éléments internes de la machine ne sont pas extensibles. Généralement les longueurs sont des puissances de deux (8, 16, 32, 64, 128, ...).

Une conséquence immédiate et importante est que tous les nombres ne peuvent être représentés.

Les entiers représentables sont compris dans un intervalle.

Les réels non rationnels ne sont pas représentables. Ils sont approximés par des rationnels. Le terme *réel* utilisé en informatique est en réalité un abus de langage, les nombres ainsi désignés sont des *rationnels*.

La représentation d'un nombre peut être différente de la valeur de ce nombre à cause d'un déficit de précision.

La même suite de bits s'interprète différemment selon qu'il s'agit d'un entier, d'un réel ou d'un caractère.

Les premières puissances de deux sont :

i	2^i	i	2^i
0	1	6	64
1	2	7	128
2	4	8	256
3	8	9	512
4	16	10	1024
5	32	11	2048

4.1 Les conversions

On va décrire des méthodes pour convertir des nombres d'une base à une autre. Pour éviter toute ambiguïté on spécifiera la base dans laquelle sera représenté chaque nombre de la manière suivante : le nombre x en base B sera écrit : $[x]_B$.

Lorsque le nombre de symboles utilisés pour écrire un nombre sera fixé, on utilisera la notation suivante : $[x]_B^n$, qui s'interprêtera comme l'écriture de x en base B sur n symboles.

4.1.1 La conversion décimal en binaire

Pour convertir en binaire un nombre entier naturel exprimé en décimal, on lui applique successivement des divisions par 2 jusqu'à obtenir un quotient égal à 0. La liste inversée des restes obtenus constitue le nombre binaire.

Exemple :

nombre	reste	
275	1	
137	1	
68	0	
34	0	
17	1	
8	0	
4	0	
2	0	
1	1	↑ sens de lecture
0		

Donc $[275]_{10} = [100010011]_2$.

Une autre méthode consiste à décomposer le nombre x en une somme de puissance de deux. Pour cela on détermine d'abord la plus grande puissance de deux inférieure à x , on la mémorise, on la soustrait à x et on réitère le même procédé sur le nombre obtenu.

Exemple :

$$\begin{aligned}275 &= 256 + 19 \\19 &= 16 + 3 \\3 &= 2 + 1 \\275 &= 2^8 + 2^4 + 2^1 + 2^0\end{aligned}$$

4.1.2 La conversion binaire en décimal

Pour convertir en décimal un nombre x exprimé en binaire, on peut additionner les puissances de deux correspondants aux bits positionnés à 1 dans l'écriture binaire de x .

Exemple :

$$[100010011]_2 = 2^8 + 2^4 + 2^1 + 2^0 = 256 + 16 + 2 + 1 = [275]_{10}$$

Un autre méthode, dite de Hörner, et analogue à la lecture des nombres exprimé en décimal, consiste à considérer les bits de l'écriture binaire de x de gauche à droite. On part d'un résultat intermédiaire initialisé à 0. Ensuite pour chaque bit de x , on multiplie le résultat intermédiaire par 2 et on lui ajoute la valeur du bit.

Exemple :

$$x = [100010011]_2$$

$$\begin{array}{rcl}
0 & & \\
2 \times 0 & + & 1 = 1 \\
2 \times 1 & + & 0 = 2 \\
2 \times 2 & + & 0 = 4 \\
2 \times 4 & + & 0 = 8 \\
2 \times 8 & + & 1 = 17 \\
2 \times 17 & + & 0 = 34 \\
2 \times 34 & + & 0 = 68 \\
2 \times 68 & + & 1 = 137 \\
2 \times 137 & + & 1 = 275
\end{array}$$

4.2 La représentation des entiers

On veut représenter un nombre x entier positif ou négatif sur un nombre limité de positions binaires. On note n le nombre de bits utilisés pour représenter les entiers. On prendra $n = 8$ dans les exemples considérés dans la suite.

4.2.1 Arithmétique élémentaire en binaire

Addition

Les règles d'addition en binaire sont extrêmement simples :

$$\begin{array}{r}
0 \\
+ 0 \\
\hline
0
\end{array}
\quad
\begin{array}{r}
0 \\
+ 1 \\
\hline
1
\end{array}
\quad
\begin{array}{r}
1 \\
+ 0 \\
\hline
1
\end{array}
\quad
\begin{array}{r}
1 \\
+ 1 \\
\hline
0
\end{array}
\quad \text{avec une retenue égale à 1}$$

Multiplication et division par 2

En décimal la multiplication d'un entier naturel par 10 consiste à ajouter un zéro à la droite de la représentation décimale de l'entier naturel. La division entière par dix consiste à supprimer le chiffre le plus à droite de la représentation décimale de l'entier naturel (sauf s'il est inférieur à 10 auquel cas le résultat est 0).

Il en est exactement de même en binaire. La multiplication d'un entier naturel par 2 consiste à ajouter un zéro à droite de la représentation binaire de l'entier naturel. Puisqu'en machine, la représentation s'effectue sur un nombre de positions fixe, cela revient à décaler toutes les positions binaires vers la gauche (en perdant le bit le plus à gauche) et à fixer le bit le plus à droite à 0.

Exemple :

$$[14]_{10} \times [10]_{10} = [140]_{10}$$

$$[00001110]_2^8 \times [2]_{10} = [00011100]_2^8 = [28]_{10}$$

La division entière d'un entier naturel par 2 consiste à supprimer le bit le plus à droite de la représentation binaire de l'entier naturel. Puisque en machine, la représentation s'effectue sur un nombre de positions fixe, cela revient à décaler toutes les positions binaires vers la droite (en perdant le bit le plus à droite) et à fixer le bit le plus à gauche à 0.

Exemple :

$$[14]_{10} \div [10]_{10} = [1]_{10}$$

$$[00001110]_2^8 \div [2]_{10} = [00000111]_2^8 = [7]_{10}$$

4.2.2 Complément à 2

Sur n bits on peut représenter 2^n informations. Pour le codage des entiers on veut pouvoir coder des positifs ainsi que des négatifs (sans oublier le zéro). Il faut donc adopter une méthode qui permette de représenter sensiblement le même nombre de positifs que de négatifs. La méthode la plus souvent adoptée est le **complément à 2**.

En complément à 2, le codage d'un entier positif $x \leq 2^{n-1}$ est immédiat. Ensuite pour coder l'opposé de x on réalise le complément à 1 puis on ajoute 1. Le bit de poids fort est un bit de signe : il est à 0 pour les positifs et à 1 pour les négatifs.

On a la propriété suivante $[x]_2^n + [-x]_2^n = 2^n$ lorsque $x \neq 0$.

Exemples :

$$[14]_{10} = [00001110]_2^8$$

$$[-14]_{10} = [11110010]_2^8$$

$[14]_{10}$	0	0	0	0	1	1	1	0
complémentation	1	1	1	1	0	0	0	1
+ 1	1	1	1	1	0	0	1	0

On peut ainsi représenter tous les entiers entre -2^{n-1} et $2^{n-1} - 1$:

Débordements

Le nombre de bits de la représentation d'un entier étant limité, il peut y avoir *débordement de capacité*. Un débordement arithmétique se produit lorsque le nombre de retenues à la position la plus à gauche et sortant de la position la plus à gauche est égal à un.

Exemples :

$$\begin{array}{r} [127]_{10} + [1]_{10} \\ \begin{array}{r} 1\ 1\ 1\ 1\ 1\ 1\ 1 \\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1 \\ +\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1 \\ \hline =\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \end{array} \end{array}$$

$$\begin{array}{r} [-128]_{10} + [-1]_{10} \\ \begin{array}{r} 1 \\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\ +\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1 \\ \hline =\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1 \end{array} \end{array}$$

$$\begin{array}{r} [-126]_{10} + [127]_{10} \\ \begin{array}{r} 1\ 1\ 1\ 1\ 1\ 1\ 1 \\ 1\ 0\ 0\ 0\ 0\ 0\ 1\ 0 \\ +\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1 \\ \hline =\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1 \end{array} \end{array}$$

$$\begin{array}{r} [126]_{10} + [1]_{10} \\ \begin{array}{r} 0\ 1\ 1\ 1\ 1\ 1\ 1\ 0 \\ +\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1 \\ \hline =\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1 \end{array} \end{array}$$

C'est pourquoi le programme `QuiBoucle` présenté en fin de la section 2.7 termine.

4.3 La représentation des réels

Pour représenter les nombres réels en virgule flottante on utilise une partie fractionnaire (ou **mantisse**) et un **exposant**. La norme IEEE-754 pour la représentation des nombres réels en virgule flottante est la plus utilisée. Il y a trois formats : 32, 64 et 80 bits.

4.3.1 La norme IEEE sur 32 bits

Le format sur 32 bits est souvent appelé format simple précision. La mantisse m est représentée sur 23 bits. L'exposant e est biaisé et représenté sur 8 bits. Un bit est consacré au signe s .



Un nombre est représenté par

$$(-1)^s(1.m)2^{e-127}$$

Zéro ne peut pas être représenté de cette manière donc la norme spécifie que la suite de 32 bits à zéro représente le nombre zéro.

Le résultat est normalisé après chaque opération.

4.4 La représentation des caractères

Les caractères sont codés sur un octet. Les valeurs correspondent à la table ASCII étendue (0 à 255). À l'origine la table ASCII a été prévue sur sept bits. Coder les caractères sur un octet permet de coder des caractères nationaux sur les codes de 128 à 255. Les 32 premiers codes sont réservés à des codes systèmes.

Voici les valeurs de 0 à 127 de la table ASCII :

0	NU	16	DL	32		48	0	64	@	80	P	96	'	112	p
1	SH	17	D1	33	!	49	1	65	A	81	Q	97	a	113	q
2	SX	18	D2	34	"	50	2	66	B	82	R	98	b	114	r
3	EX	19	D3	35	#	51	3	67	C	83	S	99	c	115	s
4	ET	20	D4	36	\$	52	4	68	D	84	T	100	d	116	t
5	EQ	21	NK	37	%	53	5	69	E	85	U	101	e	117	u
6	AK	22	SY	38	&	54	6	70	F	86	V	102	f	118	v
7	BL	23	EB	39	'	55	7	71	G	87	W	103	g	119	w
8	BS	24	CN	40	(56	8	72	H	88	X	104	h	120	x
9	HT	25	EM	41)	57	9	73	I	89	Y	105	i	121	y
10	LF	26	SB	42	*	58	:	74	J	90	Z	106	j	122	z
11	VT	27	EC	43	+	59	;	75	K	91	[107	k	123	{
12	FF	28	FS	44	,	60	<	76	L	92	\	108	l	124	
13	CR	29	GS	45	-	61	=	77	M	93]	109	m	125	}
14	SO	30	RS	46	.	62	>	78	N	94	^	110	n	126	~
15	SI	31	US	47	/	63	?	79	O	95	-	111	o	127	DT

Chapitre 5

Les instructions

Les instructions ont également une représentation sous forme de suite de bits. Le nombre d'instructions élémentaires d'un microprocesseur est limité. Les opérations sont effectuées dans des cases mémoires spéciales appelées registres. Une bonne partie du travail effectué par le processeur va donc consister en des transferts entre mémoire principale et registres. Ce modèle porte le nom de **transfert-registres**.

Comme il est très difficile d'étudier le langage machine (suite de 0 et de 1), nous allons étudier ce modèle à l'aide d'un langage d'assemblage, qui reste très proche du modèle tout en permettant une meilleure expressivité.

5.1 L'assembleur

Chaque instruction est associée à un sigle mnémotique.

L'outil qui permet de réaliser la traduction des sigles vers les suites de bits est appelé **assembleur**.

La syntaxe n'a de valeur que pour un processeur spécifique.

Il n'y a pas de langage d'assemblage normalisé.

Une instruction est généralement de la forme :

- type de l'opération ;
- opérandes.

5.2 Les instructions

Nous allons décrire un petit langage d'assemblage, très similaire au langage d'instructions des microprocesseurs. Nous allons manipuler des cases mémoires et des registres et toutes les opérations seront effectuées sur les registres. Les adresses des cases mémoires seront précédées du symbole \$. Notre langage va nous permettre de manipuler deux registres : *R1* et *R2*.

On distingue plusieurs types d'instructions suivant le type d'action qu'elles permettent d'effectuer.

5.2.1 Les instructions de transfert

Les instructions de transfert permettent de dupliquer une valeur de ou vers un registre. Elles peuvent prendre différentes formes.

- LOAD registre valeur : permet de charger la valeur dans le registre ;
- LOAD registre adresse : permet de charger la valeur stockée à l'adresse dans le registre ;
- LOAD registre registre : permet de charger la valeur contenue dans le second registre dans le premier ;
- STORE registre adresse : permet de stocker la valeur contenue dans le registre à l'adresse indiquée.

5.2.2 Les instructions de calcul arithmétique

Les instructions de calcul arithmétique permettent d'effectuer les opérations d'addition et multiplication. Les opérandes sont stockés dans les deux registres et le résultat est stocké dans l'un deux.

- ADD registre registre : additionne le contenu des deux registres et stocke le résultat dans le premier ;
- SUB registre registre : soustrait le contenu du second registre au premier et stocke le résultat dans le premier ;
- MUL registre registre : multiplie le contenu du second registre au premier et stocke le résultat dans le premier.

5.2.3 Les instructions de branchement

En principe les instructions sont exécutées en séquence les unes à la suite des autres dans leur ordre d'apparition. Une instruction de branchement permet d'exécuter une instruction autre que celle qui apparaît à la ligne suivante du programme.

L'adresse de la prochaine instruction à exécuter se trouve généralement dans un registre spécial (compteur ordinal). Un branchement revient donc à une modification du contenu de ce registre.

Il y a deux manières pour changer l'ordre d'exécution du programme : il est possible d'effectuer un saut inconditionnel vers une instruction ou bien le saut peut être conditionné au résultat d'une condition :

- JUMP adresse : saut inconditionnel à l'instruction se trouvant à l'adresse indiquée ;
- JEQU registre adresse : saut conditionnel à l'adresse indiquée ; si le contenu du registre est égal à zéro alors on saute à l'instruction située à l'adresse indiquée sinon on continue en séquence ;
- JGTZ registre adresse : saut conditionnel à l'adresse indiquée ; si le contenu du registre est strictement supérieur à zéro alors on saute à l'instruction située à l'adresse indiquée sinon on continue en séquence.

Nous aurons également à disposition deux instructions pour lire et écrire des valeurs et une instruction spéciale pour terminer l'exécution du programme :

- WRITE registre : écrit la valeur contenue dans le registre à l'écran ;
- READ registre : lit une valeur dans le tampon d'entrée et la stocke dans le registre ;
- STOP : termine l'exécution du programme.

Exemples :

Reprenons tout d'abord le programme qui saisit un entier a , puis l'affiche ainsi que son cube.

	assembleur	pseudo-code	description
\$100	READ $R1$	$R1 \leftarrow a$	$(R1 = a, R2 = ?, \$705 = ?)$
\$101	STORE $R1$ \$705	$\$705 \leftarrow R1$	$(R1 = a, R2 = ?, \$705 = a)$
\$102	MUL $R1$ $R1$	$R1 \leftarrow R1 \times R1$	$(R1 = a^2, R2 = ?, \$705 = a)$
\$103	STORE $R1$ $R2$	$R2 \leftarrow R1$	$(R1 = a^2, R2 = a^2, \$705 = a)$
\$104	LOAD $R1$ \$705	$R1 \leftarrow \$705$	$(R1 = a, R2 = a^2, \$705 = a)$
\$105	MUL $R2$ $R1$	$R2 \leftarrow R2 \times R1$	$(R1 = a, R2 = a^3, \$705 = a)$
\$106	WRITE $R1$		écrire le contenu de $R1 = a$
\$107	WRITE $R2$		écrire le contenu de $R2 = a^3$

Reprenons maintenant le programme qui saisit un entier n puis affiche les n premiers entiers strictement positifs.

\$100	READ $R1$	$R1 \leftarrow n$	$(R1 = n, R2 = ?, \$705 = ?)$
\$101	STORE $R1$ \$705	$\$705 \leftarrow R1$	$(R1 = n, R2 = ?, \$705 = n)$
\$102	LOAD $R2$ 1	$R2 \leftarrow 1$	$(R1 = n, R2 = 1, \$705 = n)$
\$103	SUB $R1$ $R2$	$R1 \leftarrow R1 - R2$	$(R1 = R1 - R2, \$705 = a)$
\$104	JGTZ $R1$ \$200		si $R1 > 0$ alors aller à la ligne 200
\$105	WRITE $R2$		écrire le contenu de $R2$
\$106	STOP		terminer l'exécution du programme
\$200	WRITE $R2$		écrire le contenu de $R2$
\$201	ADD $R2$ 1	$R2 \leftarrow R2 + 1$	
\$202	LOAD $R1$ \$705	$R1 \leftarrow \$705$	$(R1 = n)$
\$203	JUMP \$105		aller à la ligne 105

Comme on peut le voir dans ce deuxième exemple, dès qu'un programme comporte des sauts il devient très difficile d'en simuler l'exécution. En particulier, construire un programme et montrer qu'il est correct (c'est-à-dire qu'il se termine en produisant les résultats escomptés) demande un forma-

lisme adapté. C'est l'algorithmique qui traite de ce sujet.

Remerciements

Ce cours a profité de nombreuses remarques d'enseignants du DIR.

Bibliographie

- [1] Cardon (Alain) et Charras (Christian). – *Introduction à l'algorithmique et à la programmation*. – Ellipses, 1996.
- [2] Charles (Henri-Pierre). – *Initiation à l'informatique : Programmation, Algorithmique, Architectures*. – Eyrolles, 1999.
- [3] Delannoy (Claude). – *Initiation à la programmation*. – Eyrolles, 1997, seconde édition.
- [4] Delannoy (Claude). – *Programmer en Turbo Pascal 7*. – Eyrolles, 1998, seconde édition.
- [5] Goupille (Pierre-Alain). – *Technologie des ordinateurs et des réseaux*. – Dunod, 2004, 7^e édition.
- [6] Serres (Michel) et Farouki (Nayla). – *Le Trésor, dictionnaire des sciences*. – Flammarion, 1997.