«A Paris, ils se contentèrent de me dévisager lorsque je leur parlais en français; je ne suis jamais vraiment parvenu à faire comprendre leur propre langue à ces idiots.»

Mark Twain, « Les innocents à l'étranger », 1869.

1.) Les concepts de base.

1.1.) Algorithmes.

Un ordinateur est une machine construite pour exécuter de manière séquentielle des fonctions primitives telles qu'additionner, soustraire, multiplier, diviser, ou comparer des nombres. On obtient des traitements plus complexes (comme l'extraction d'une racine carrée par exemple) en combinant ces opérations entre elles dans un ordre convenable. Bien entendu, comme la vitesse de l'ordinateur est très grande, il n'est pas possible de lui donner des ordres à mesure de l'avancement du travail, comme on le fait avec une calculette : il faut tout d'abord dresser la liste des instructions auxquelles la machine devra obéir, dans l'ordre de leur exécution. C'est ce qu'on appelle un **programme**. On charge ensuite ce programme dans la mémoire de la machine, où elle puisera les instructions à mesure de leur exécution, à sa propre vitesse.
Donc si l'on veut faire exécuter une tâche à la machine, il faut donc rédiger le programme correspondant, c'est-à-dire avoir trouvé quelle suite d'opérations élémentaires assure l'exécution de la tâche. En un mot, il faut trouver une méthode d'exécution de la tâche ou de résolution du problème, aussi appelée algorithme. On peut donc définir un algorithme de la manière suivante :

un algorithme, c'est une méthode de résolution d'un problème suivant un enchaînement déterminé de règles opératoires.

En fait, le concept d'algorithme existe depuis très longtemps. Ce concept est indissociable du désir humain de transmettre des moyens efficaces de résolution de problèmes, en procédant étape par étape. Il pouvait s'agir de procédures juridiques ou mathématiques comme chez les Babyloniens, de procédés mnémotechniques comme chez les Grecs, de règles linguistiques comme chez les grammairien romains et, dans toutes les civilisations, de recettes divinatoires, médicales, culinaires... De nos jours encore, tout le monde utilise des algorithmes sans le savoir, ne serait-ce quand suivant une recette de cuisine, une fiche de tricot ou le mode d'emploi d'un appareil électroménager.

L'encyclopédie d'Alembert décrit ainsi le mot algorithme : « *Terme Arabe, employé par quelques auteurs, et singulièrement par les Espagnols, pour signifier la pratique de l'algèbre. Le même mot se prend, en général, pour désigner la méthode et la notation de toute espèce de calcul. En ce sens, on dit l'algorithme du calcul intégral, l'algorithme du calcul exponentiel, l'algorithme des sinus etc* .»

Aujourd'hui, principalement sous l'influence de l'informatique, la finitude devient une notion essentielle contenue dans le terme algorithme, le distinguant de procédés plus vagues. On peut ainsi lire, dans l'*Encyclopédia Universalis* : « *Un algorithme est une suite finie de règles à appliquer dans un ordre déterminé à un nombre fini de données pour arriver, en un nombre fini d'étapes, à un certain résultat, et cela indépendamment des données* ».

Un algorithme possède les caractéristiques suivantes :

1. L'algorithme doit pouvoir être écrit dans un certain langage (un langage est un ensemble de mots écrits à partir d'un alphabet fini.)
2. La question posée est déterminée par une donnée, appelée « entrée » pour laquelle l'algorithme sera exécuté.

3. L'algorithme est un processus qui s'exécute étape par étape.
4. L'action à chaque étape est strictement déterminée par l'algorithme, l'entrée, et les résultats obtenus dans les étapes précédentes.
5. la réponse, appelée « sortie », est clairement spécifiée.

1.2.) Programmes.

Un programme est la traduction d'un algorithme dans un langage de programmation.

La première difficulté sera d'adapter notre raisonnement humain à la logique de fonctionnement de l'ordinateur. Il faudra être suffisamment précis pour que la machine puisse exécuter la tâche. Par exemple, si on veut échanger le contenu de deux cellules mémoires qu'on appellera A et B.

A	B
5	6

La première idée qui consiste à dire "je mets le contenu de A dans B et celui de B dans A" aboutirai à

A	B
5	5

Le problème est donc un peu plus complexe puisqu'il faut tout d'abord sauvegarder le contenu de B, recopier le contenu de A dans celui de B puis restaurer le contenu sauvegarder dans la cellule A.

De plus, il faudra adapter notre algorithme aux moyens dont dispose la machine. Par exemple, si on veut trier la liste suivante, il faudra trouver une méthode qui soit à la fois précise et adaptée au fonctionnement de l'ordinateur.

5	3	6	8	6	1	9	7	2	4

La méthode la plus simple consiste à rechercher le plus petit élément de la liste, de le placer en premier, puis de rechercher le plus petit élément restant, de le placer en second etc... Mais d'autres méthodes sont possibles : le tri bulle qui consiste à comparer les éléments deux à deux ou le tri rapide qui consiste à prendre un pivot (par exemple 4) et de diviser la liste en deux sous-listes à trier :

Il ne suffit donc pas d'être capable d'exécuter une tâche pour pouvoir décrire la méthode par laquelle on l'exécute, car nos actions sont rarement méthodiques Il y a une grande distance entre savoir faire et pouvoir faire faire par une machine. Il faut bien définir quelles sont les données du travail, quelles circonstances variées peuvent se produire, quelles actions sont à prendre dans chaque cas pour parvenir au résultat recherché.

1.3.) Les langages de bas niveau.

On classe souvent les langages de programmation en "générations". La première génération regroupe les langages machine, la seconde les langages d'assemblages et la troisième les langages évolués.

1.3.1.) Le langage machine.

Les instructions emmagasinées dans la mémoire de l'ordinateur et exécutées par l'unité de traitement sont représentées sous la forme de chaînes de chiffres binaires. On dit qu'elles sont exprimées en langage machine. Le langage machine constitue le seul langage réellement «compris» par l'ordinateur, tous les autres langages correspondant à des formes de structuration du langage humain. Ainsi sur IBM 370 l'instruction :

0	1	0	1	1	0	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	1

90 (0x5A)	3 (0x3)	517 (0x205)
Code opératoire de l'addition à un registre	N° du registre	Adresse de l'opérande.

signifie "additionner au contenu du registre 3 le contenu du mot d'adresse 517".

Cette programmation, lente et fastidieuse, était source de nombreuses erreurs. Elle n'est utilisée aujourd'hui que dans quelques cas isolés :

- Les hackers, pour enlever une protection d'un programme.
- Les virus pour infecter un programme exécutable.
- Les patchs pour corriger un programme exécutable.

Cependant, dans le cadre de l'écriture d'un programme important et complexe, elle est quasi inutilisable et surtout inutile. On développa donc des langages de programmation plus commodes à employer, les langages d'assemblage et les langages évolués.

1.3.2.) Les langage d'assemblage.

Les langages d'assemblage ont permis l'écriture des instructions du langage machine sous forme symbolique ; la traduction en binaire est assurée par un programme, fourni par le constructeur, appelé assembleur.

Les langages d'assemblage sont des langages de programmation les plus proche du langage machine, où chaque instruction correspond à une instruction machine unique. Bien qu'ils introduisent certains allégements (codes opératoires mnémoniques, adresses en base 10 ou 16, utilisation d'étiquettes pour les ruptures de séquence...) ils ne solutionnent pas les autres inconvénients du langage machine (extrême pauvreté des instructions, liaison étroite en le langage et le type d'ordinateur). En effet, le jeu d'instructions d'un tel langage est associé à un certain type de processeur. Ainsi, les programmes écrits en langage assembleur pour un processeur particulier doivent être réécrits pour tourner sur un ordinateur équipé d'un processeur différent. Après écriture d'un programme en langage d'assemblage, le programmeur fait alors appel à l'assembleur spécifique du processeur, qui traduit ce programme en instructions machine.

Cependant, les langages d'assemblage sont encore utilisés aujourd'hui dans quelques cas :

- Quand la vitesse d'exécution est primordiale (par exemple, les jeux vidéo ou l'informatique en temps réel).
- Pour accéder directement à certains périphériques (programmation d'une carte vidéo, par exemple).
- Quand la taille du programme est vitale (par exemple, l'informatique embarquée ou les virus).

Pour pallier aux inconvénients des langages d'assemblage, on a imaginé de créer des langages dits évolués, écrits avec l'alphabet usuel, comportant des sigles ayant une valeur mnémotechnique pour l'utilisateur. Pour chaque langage et chaque type de machine, on a rédigé un programme (baptisé compilateur) qui assure la traduction de tout texte écrit avec le langage évolué en un texte en langage de la machine, en garantissant la fidélité de la traduction : les actions exécutées par la machine sont effectivement celles qui sont décrites par le langage évolué.

Un langage de programmation n'est pas n'importe quelle variante d'un langage ordinaire. Il ne peut souffrir l'ambiguïté ; il ne doit offrir aucune latitude d'interprétation. Le premier langage évolué fut Fortran, inventé par John Backhus en 1955. On a du mal à se représenter aujourd'hui la complexité de cette invention : il fallut reconnaître le champ sémantique que devait couvrir le langage, inventer une grammaire qui élimine toute ambiguïté, toute polysémie. Il fallut trouver des algorithmes de traduction qui garantissent la conservation des opérations désignées dans le langage évolué. Il fallut, à l'époque, que le premier compilateur soit performant, pour ne pas freiner l'usage de Fortran.

1.4.) Les différentes catégories de langages évolués.

1.4.1.) LES LANGAGES PROCEDURAUX.

Les langages impératifs ou procéduraux ne s'éloignent des langages machines que par leur richesse d'expression ; ils en gardent la structure de calcul.

1.4.1.1.) Les variables.

On part du principe qu'un ordinateur est construit autour d'une mémoire divisée en cases dans lesquelles on peut ranger des informations. On peut consulter le contenu d'une case ou le modifier. Le concept de case se retrouve dans celui de variable d'un langage de programmation impératif, désignée par un nom, X par exemple, auquel on peut affecter une valeur :

```
X <-3;
```

L'instruction d'affectation permet de donner une valeur à une variable, mais aussi de la changer. L'instruction

```
X <- X + 1;
```

consulte la valeur de X (ici 3), lui ajoute 1 et donne le résultat comme nouvelle valeur à X. X prend ainsi la valeur 4.

Une variable est donc désignée par un nom. Dans la grande majorité des langages de programmation, il est nécessaire de déclarer les variables avant leur utilisation en leur donnant un nom et un type :

```
VAR X : ENTIER;
```

L'instruction précédente réservera un emplacement mémoire pour une variable nommée X et qui contiendra un entier (donc 32 bits ou 4 octets dans la plupart des systèmes actuels). Il faut donc voir le concept de variable est donc une abstraction de celui d'emplacement mémoire. Les langages de programmation procéduraux proposent en général les types suivants :

Octet (ou byte)	Un octet.
Booléen (ou boolean)	Un octet.
Entier (ou int)	2 ou 4 octets
Flottant (ou real)	
Caractère (ou char)	un octet (parfois 2 pour les caractères orientaux)
Chaîne (ou string)	en fonction de l'implémentation.

1.4.1.2.) Le contrôle de l'enchaînement.

L'instruction

```
SI X > 0 AFFICHER X SINON AFFICHER "X NUL";
```

est dite conditionnelle et fait afficher X s'il est plus grand que 0 ; sinon, "X Nul". Une extension de la conditionnelle est l'alternative à n branches :

```
SELON X
  1 : AFFICHER "UN";
  2 : AFFICHER "DEUX";
  3 : AFFICHER "TROIS";
  SINON AFFICHER "AUTRE VALEUR";
FIN SELON;
```

Les instructions itératives permettent de répéter un processus.

Par exemple, pour calculer la somme des nombres de 1 à 10 on aurait :

```
X <- 1;
I <- 1;
1:
X <- X + I;
I <- I + 1;
SI I <= 10 THEN GOTO 1;
AFFICHER X;
```

L'instruction GOTO est la traduction littérale d'une instruction qui existe dans tous les microprocesseurs : l'instruction de saut (conditionnel ou non). L'utilisation exagérée de cette instruction rend les programmes difficilement lisibles. On lui préferera donc

```
X <- 1;
I <- 1;
TANT QUE I <= 10 FAIRE
  X <- X + I;
  I <- I + 1;
FIN TANT QUE;
AFFICHER X;
```

Ou bien

```
X<-0;
I <- 1;
REPETER
  X<-X+I;
  I <- I + 1;
JUSQU'A I>10;
AFFICHER X;
```

Ou encore

```
X<-0;
POUR I VARIANT DE 1 A 10 FAIRE
  X<-X+I;
FIN POUR;
AFFICHER X;
```

Ou (plus rarement) :

```
X<-0;
I<-0;
BOUCLE
  I<-I+1;
  X<-X+I;
  SORTIR SI I=10;
FIN BOUCLE;
AFFICHER X;
```

1.4.1.3.) Les sous-programmes.

Plutôt qu'un traitement monolithique du programme, on a imaginé de diviser un programme en un ensemble de sous programmes. C'est ce qu'on retrouve avec l'instruction suivante :

```
X<-0;
I<-0;
FAIRE DE 1 A 5 JUSQU'A I=10;
AFFICHER X;
FIN DE PROGRAMME
1:
  I<-I+1;
  X<-X+I;
5:
```

La notion de fonction est une extension de cette instruction :

```
FONCTION incremente (x : ENTIER): ENTIER;
DEBUT
  x<-x+1;
  RETOURNER x;
FIN
FONCTION additionne (a,b : ENTIER):ENTIER;
DEBUT
  RETOURNER a+b;
FIN;
DEBUT DE PROGRAMME
X<-0;
I<-0;
TANT QUE I<=10 FAIRE
  I<-incremente(I);
  X<-additionne(X,I);
FIN TANT QUE
FIN DE PROGRAMME
```

Tout cela paraît très simple. Malheureusement, ces écritures décrivent un processus évolutif par lequel il devient difficile de suivre l'effet d'une suite d'instructions : il y a la valeur de X avant l'exécution d'une instruction, et sa valeur après, qui peut ne pas être la même. Une simple lecture ne permet pas de dire ce que fait le morceau de programme que voici :

```
X <- X + Y;
Y <- X - Y;
X = X-Y;
```

Un programme décrit donc une suite d'actions, qui fait passer de la situation initiale (celle des données) à la situation finale (celle des résultats). Pour en connaître l'effet, il faut détailler la suite de situations qui est engendrée par la suite d'actions. Supposons qu'au début X ait la valeur a et Y la valeur b :

INSTRUCTION	VALEUR DE X	VALEUR DE Y
	a	b
`X <- X+Y`	a+b	b
`Y <- X-Y`	a+b	a
`X <- X-Y`	b	a

Nous voyons que les valeurs de X et Y ont été échangées. La programmation impérative repose sur le couple situation-action. Le programme décrit la suite d'actions qui fait passer de la situation initiale à la situation finale. Pour établir la validité d'un programme, on détermine (comme nous venons de le faire dans un exemple simple) la suite de situations qu'il engendre. Si, pour toutes les données possibles, la situation finale est celle que l'on attend, le programme est juste.

L'inconvénient principal de la programmation impérative à dû à l'apparition d'effets de bord. Considérons le programme suivant :

```
VAR x : ENTIER;

FONCTION bizarre : ENTIER;
DEBUT
  RETOURNER x + 1;
FIN
```

Dans ce programme, l'instruction

```
bizarre = bizarre
```

renverrait la valeur FAUX car elle serait évaluée ainsi

```
(x +1)+1 = x+1
```

1.4.1.4.) Exercices.

Ecrire un programme qui ajoute une journée à une date donnée. On dispose de la fonction `mod` qui renvoie le reste de la division entière.

```
PROGRAMME lendemain;

VAR j,m,a : ENTIER;
    max   : ENTIER;

DEBUT
     LIRE (j,m,a);
     max <- 30;
     SI m=1 OU m=3 OU m=5 OU m=7 OU m=8 OU m=10
     OU m=12 ALORS max<-31;
     SI m=2 ALORS
     DEBUT
          max<-28;
          SI (a mod 4 = 0) ET PAS (a mod 100) ALORS max<-29;
          SI (a mod 400=0) ALORS max<-29;
     FIN
     j <-j+1;
     SI j>max ALORS
     DEBUT
          j<-1;
          m <- m+1;
          SI m>12 ALORS
          DEBUT
               m<-1;
               a<-a+1;
          FIN
     FIN
     AFFICHER(j,m,a);
FIN.
```

1.4.2.) LES LANGAGES "OBJETS".

La programmation par objets est une extension de la programmation impérative. L'idée de base est d'associer le code et les données dans une entité appelée "objet".

J'ai dis que le concept de variable était une abstraction de la notion de "case mémoire". Aux zones mémoire, on avait associé un nom et un type de données. Avec la programmation objet, on avance encore dans l'abstraction. Aux cases mémoires on associe en plus du code exécutable. Il faut voir un objet - au sens informatique - comme un ensemble de variables (les propriétés) et des instructions qui lui sont associées.

Pour reprendre l'exemple du calcul de la somme des nombres de 1 à 10, on pourrait définir une classe d'objet "Tsomme" :

```
TYPE Tsomme = CLASSE(Tobject)
         PRIVE
             valeur : reel;
             borne_inf : entier;
             borne_sup : entier;
             PROCEDURE calcule;
         PUBLIC
             CONSTRUCTEUR cree(inf,sup : entier);
             FONCTION donne_valeur : reel;
         FIN DEFINITION;
```

Cet objet possède des propriétés (valeur, borne_inf, borne_sup) et des méthodes (cree, calcule, donne_valeur). Certaines propriétés et méthodes sont privées (c'est à dire qu'elles ne peuvent être accédées qu'à l'intérieur de l'objet). C'est ce qu'on appelle l'encapsulation. D'autres sont publiques (elles peuvent être appelées de l'extérieur de l'objet).

Ainsi, à l'intérieur de la classe Tsomme, la procédure calcule pourrait être appelée à chaque fois qu'un utilisateur appelle la fonction donne_valeur.

```
FONCTION Tsomme.donne_valeur : reel;

DEBUT
  calcule;
  retourne valeur;
FIN;
```

Quand à la fonction calcule, elle pourrait être définie ainsi :

```
PROCEDURE Tsomme.calcule;

DEBUT
  valeur:=0;
  POUR i VARIANT DE borne_inf A borne_sup FAIRE
  valeur:=valeur+i;
FIN;
```

La classe Tsomme pourrait être utilisée au travers d'un objet somme (une instance de la classe) :

```
...
VARIABLE somme : Tsomme;
DEBUT
  somme.Cree(1,10);
  afficher(somme.donne_valeur)
FIN;
...
```

On autre propriété fondamentale de la programmation par objets s'appelle l'héritage. On pourrait ainsi définir une classe d'objet Tsomme_carre, qui hérite de Tsomme :

```
TYPE Tsomme_carre = CLASS(Tsomme)
         PRIVE
            PROCEDURE calcule;
         PUBLIC
         FIN DEFINITION;
```

Cette classe ne fait que redéfinir la procédure calcule pour en faire quelque chose comme :

```
PROCEDURE Tsomme_carre.calcule;

DEBUT
  valeur:=0;
  POUR i VARIENT DE borne_inf A borne_sup FAIRE
  valeur:=valeur+i*i;
FIN;
```

La classe Tsomme_carre hérite des autres propriétés et méthodes de Tsomme.

Langages "objets ": Smalltalk, C++, DELPHI, Java.

1.4.2.1.) Exercices.

Définition d'une classe Tdate.

```
TYPE Tdate =    CLASSE(OBJET)

                PRIVE
                j,m,a : ENTIER
                PUBLIC
                CONSTRUCTOR INIT(jj,mm,aa : ENTIER);
                FONCTION DONNE_JOUR : ENTIER;
                FONCTION DONNE_MOIS : ENTIER;
                FONCTION DONNE_AN : ENTIER;
                FONCTION DONNE_JOUR_SEMAINE : ENTIER;
                FONCTION LENDEMAIN;
                FONCTION DATE_DIFF( d : TDATE) : ENTIER;
                FIN DEFINITION
```

1.4.3.) LES LANGAGES FONCTIONNELS

La programmation impérative se situe dans la perspective constructiviste des mathématiques, qui définit une fonction par la suite des opérations à effectuer pour la calculer. Elle s'opposait à la perspective intuitionniste, pour laquelle une définition implicite suffit à prouver l'existence d'une fonction. C'est la voie qu'inaugura John Mac Carthy, vers 1960, en créant le langage Lisp (List Processing Language). Elle se fonde, elle aussi, sur la récurrence, mais en l'exploitant d'une autre façon.
Imaginons qu'on veuille calculer la somme des entiers de 0 à 10. On peut dire que pour obtenir cette somme, il suffit d'ajouter 10 à la somme des entiers de 0 à 9. Or, pour calculer la somme des entiers de 0 à 9, il suffit d'ajouter 9 à la somme des entiers de 0 à 8... On continue à raisonner par récurrence jusqu'à ce qu'on trouve une condition d'arrêt. Ici ce sera "la somme des entiers de 0 à 0 est égale à 0". On peut formaliser ce raisonnement par :

$$somme(n) = \begin{cases} 0 \quad si \quad n = 0 \\ \\ n + somme(n-1) \quad si \quad n > 0 \end{cases}$$

Ce qui conduirait donc à écrire :

```
Somme (x) = si x=0 alors renvoyer 0 sinon renvoyer
x + somme(x-1).
```

On aurait donc :

Somme(10)	=	10 + somme(10 - 1) = 10 + somme(9)
	=	10+9+somme(9-1) = 10+9+somme(8)
	=	10+9+8+somme(8-1)=10+9+8+somme(7)
	=	10+9+8+7+somme(7-1) = 10+9+8+7+somme(6)
	=	10+9+8+7+6+somme(6-1) = 10+9+8+7+6+somme(5)
	=	10+9+8+7+6+5+somme(5-1) = 10+9+8+7+6+5+somme(4)
	=	10+9+8+7+6+5+4+somme(4-1) = 10+9+8+7+6+5+4+somme(3)
	=	10+9+8+7+6+5+4+3+somme(3-1)= 10+9+8+7+6+5+4+3+somme(2)
	=	10+9+8+7+6+5+4+3+2+somme(2-1)= 10+9+8+7+6+5+4+3+2+somme(1)
	=	10+9+8+7+6+5+4+3+2+1+somme(0) = 10+9+7+6+5+4+3+2+1+somme(0)
	=	10+9+8+7+6+5+4+3+2+1+0 = 55

Dans les langages fonctionnels, la fonction est l'objet de base. On peut, par exemple, passer une fonction en paramètre ou une fonction peut renvoyer une autre fonction. Par exemple, si on voulait calculer la somme des carrés des nombres entre un et dix on écrirait :

```
carre (x) = x * x.
```

Puis

```
Somme ( f , x) = si x=0 alors renvoyer f (0) sinon
renvoyer f (x) + somme( f , (x-1)).
```

Ce qui conduirait à appliquer f à tous les nombres entre 0 et 10 et à additionner les résultats.

Puis

```
Somme (carre 10).
```

Langages "fonctionnels" : Caml, Lisp, Logo.

1.4.3.1.) Exercices.

En s'inspirant de la définition de la somme, écrire une fonction factorielle(n) qui calcule la factorielle de n.

```
Factorielle(n) = si n=1 alors 1 sinon n*factorielle(n-1);
```

1.4.4.) LES LANGAGES "LOGIQUES".

Pour faire résoudre un problème par un ordinateur, il faut en fournir une méthode explicite de calcul si on utilise la programmation impérative, une méthode implicite par la programmation récursive. Dans les deux cas, il faut trouver une relation de récurrence exploitable. La programmation logique, inventée par le Français Alain Colmerauer, fait sortir de cet univers. Elle vise à donner à l'ordinateur non un algorithme de résolution, mais des informations sur les données et les relations qui les lient aux résultats. Un exemple donnera une idée de ce mode de programmation. Nous prendrons des libertés avec la syntaxe d'un quelconque langage logique, comme nous l'avons déjà fait pour les autres modes.

Je me propose de rechercher un parti possible pour Sophie, sachant que l'homme idéal pour Sophie doit remplir les conditions suivantes :

- Etre un homme, non fumeur et végétarien.
- Ou bien être un homme bien payé.

Je vais considérer des faits comme "Pierre est un homme" ou"Edmond est un fumeur" ou "Thomas est végétarien".

```
monsieur(pierre).
monsieur(hugo).
monsieur(jean).
monsieur(thomas).
fumeur(hugo).
mange_de_la_viande(jean).
mange_de_la_viande(hugo).
salaire(pierre,6000).
salaire(hugo,3000).
salaire(jean,12000).
salaire(thomas,5000).
```

Je vais ensuite entrer des relations telles que :

```
vegetarien(X) if not( mange_de_la_viande(X)).
bien_paye(X) if salaire(X,S) and S>10000.

bon_parti(X) if monsieur(X) and not(fumeur(X)) and
vegetarien(X).
bon_parti(X) if monsieur(X) and bien_paye(X).
```

Pour trouver quelle personne serait un bon parti pour Sophie, il faudrait ensuite poser la question :

```
Goal : bon_parti(X)
X=pierre
X=jean
2 solutions.
```

Pour reprendre les exemples précédents, le calcul de la somme des dix premiers entiers s'écrirait ;

```
somme(0,0).
somme(N,Resultat) if
  N >= 0 and
  N1 = N-1 and
  somme(N1,S1) and
  Resultat = N+S1.
```

et se lancerait ainsi :

```
goal : somme(10,X)
X=55
1 solution trouvée.
```

Nous sommes sortis du domaine où programmer veut dire trouver une méthode de résolution. Il n'y a plus création d'algorithme. On fixe les relations entre des éléments. On donne certains éléments, le système trouve tous ceux qui leur sont liés par ces relations. Pour y parvenir, l'ordinateur doit être muni d'un programme complexe qui cherche quelles règles appliquer et, au besoin, quelles variables supplémentaires mettre en jeu. Ce programme, appelé moteur d'inférence , relève des théories de la démonstration automatique.

Langages "logiques" : Prolog.

1.4.5.) RELATION ENTRE CES MODES DE PROGRAMMATION.

Nous avons présenté les divers modes de programmation. En théorie, ils ont tous la même puissance d'expression. Mais ils ne relèvent pas de la même forme de pensée. La programmation impérative demande que l'on explicite la suite d'actions que devra faire l'ordinateur ; c'est une œuvre de stratégie. Elle se fonde sur le couple situation-action, et sur la récurrence pour la création de boucles. La programmation récursive se fonde sur le concept de fonction et sur la récurrence pour la définition implicite de fonction. Elle est peut-être plus proche de la pensée mathématique. La programmation par objets met l'accent sur l'objet du traitement, les actions qu'il peut subir étant décrites impérativement ou récursivement. C'est la perspective qui est modifiée, pas la façon d'agir ou de définir. La programmation logique demande seulement une description formelle des relations entre données et résultats, mais ne peut éviter un recours aux définitions implicites.
Chaque mode a ses langages propres dont les plus connus sont Fortran, Cobol, Basic, C, Pascal, Ada pour la programmation impérative ; Lisp, Logo pour la programmation récursive ; Smalltalk, C++ pour la programmation objet ; Prolog pour la programmation logique. Dans chaque famille, les langages sont construits sur les mêmes concepts. Ils diffèrent par la clarté de leurs structures, la richesse des constructions qu'ils permettent, la variété des objets qu'ils traitent. Mais ce sont des différences de surface, de sorte que, quand on connaît et maîtrise bien l'un d'eux, on passe assez facilement aux autres.

1.5.) Langages interprétés / compilés.

1.5.1.) LANGAGES INTERPRETES.

Les langages interprétés sont des langages décodés et exécutés instruction par instruction à l'aide d'un programme appelé interpréteur. L'interpréteur travaille simultanément sur le programme et les données. Il ne se trouve absolument pas préparé à toute construction du programme, même en cas d'exécution répétée. Il n'utilise pas les renseignements qu'il serait possible de collecter grâce à une inspection du texte du programme et qui sont indépendantes des données d'entrée, telles que le nombre de variables déclarées dans un bloc, une fonction ou une clause. Ces renseignements

pourraient être utilisés pour une allocation mémoire permettant un accès efficace aux valeurs des variables.

1.5.2.) LANGAGES COMPILES.

Les langages compilés sont des langages où toutes les instructions sont traduites en code objet avant d'être exécutées. Cette conversion s'effectue au moyen d'un compilateur. Au sens le plus large, un compilateur traduit un ensemble de symboles en un autre ensemble selon différentes règles logiques, si bien que le langage d'arrivée peut être aussi un autre langage de haut niveau, même si le but de l'opération reste néanmoins d'aller vers un langage plus simple.

Certaines sources d'inefficacité sont liées à l'usage d'un interpréteur, qu'il est souhaitable d'éviter. On utilise dans ce but un principe fréquemment utilisé en informatique désigné par l'expression de **pré-traitement** ou **évaluation partielle** ou encore **calculs mixtes**.

Tandis que l'interpréteur reçoit et traite simultanément ses deux arguments, à savoir le programme P et les données E, on sépare maintenant en deux temps le traitement du programme et celui des données. Tout d'abord le programme P est prétraité, c'est à dire analysé indépendamment de toute donnée d'entrée et traduit dans une autre représentation, qui permet une exécution efficace avec n'importe quel jeu de données. On admet ce faisant que le coût supplémentaire lié au pré-traitement du programme sera amorti lors de l'exécution pour un ou plusieurs jeu de données.
Le pré-traitement consiste à traduire le programme P écrit dans un langage L dans le langage machine M d'un calculateur abstrait ou concret. Cette phase de compilation aboutie à un programme cible PM. Si le programme cible PM est un programme du langage machine d'un vrai calculateur, il n'est plus besoin de décoder les codes d'instruction pour les exécuter.

La figure suivante montre une structure conceptuelle de compilateur où la compilation est décomposée en une séquence de sous-processus. Chaque sous-processus reçoit une représentation du programme et produit une nouvelle représentation qui peut être d'un nouveau type ou bien du même type (où seul le contenu a été modifié).

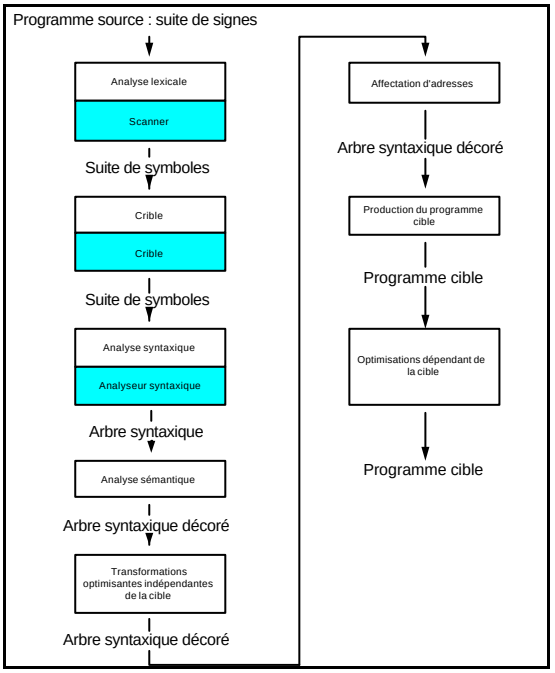


Figure 1 : Les phases de la compilation.

Les compilateurs procèdent généralement en deux phases. Durant la première, il s'agit de comprendre le code source, c'est-à-dire d'en créer une représentation cohérente en mémoire. Pour cela, on effectue successivement :

- **Une analyse lexicale** (ou "scanner") qui lit un programme source sous la forme d'une suite de caractères et le décompose en une succession d'unités lexicales du langage appelées symboles. L'analyseur lexical fait la différence entre les suites de caractères qui n'ont qu'un rôle de séparateur et qui seront ignorés des suites significatives de ces mêmes caractères.
- **Un crible** qui reconnaît, dans la suite de symboles produite par l'analyseur lexical :
  - Ceux qui ont rôle particulier dans le langage (les mots-clés).
  - Ceux qui sont sans importance et peuvent être éliminés.
  - Ceux qui ne font pas à proprement parler partie du programme mais sont plutôt des directives au compilateur.
- **Une analyse syntaxique** qui est chargée de découvrir la structure du programme à partir des éléments lexicaux. Elle sait comment sont construites les expressions, les instructions, les déclarations et tente donc de reconnaître dans la suite de symboles fournis par l'analyse lexicale la structure du programme. Elle indique les éventuelles erreurs de syntaxe commises par le programmeur.
- **Une analyse sémantique** qui vérifie les autres règles du langage et détecte les éventuelles erreurs. Elle détermine les propriétés du programme indépendantes du contexte que l'on peut calculer à la compilation en se déterminant uniquement sur le texte du programme (par exemple la vérification du typage).

Lors de la seconde phase, la structure du programme évolue selon les exigences du langage d'arrivée, pour produire finalement un code objet qui soit performant.

1.5.3.) LA COMPILATION "A LA VOLEE".

Pour essayer de bénéficier à la fois des avantages des langages interprétés (portabilité, faible taille du code) et des langages compilés (vitesse d'exécution), on a imaginé de compiler les instructions d'un programme « à la volée ». La première fois qu'une fonction, une procédure ou un sous-programme est exécuté, il passe d'abord par une phase de compilation. Quand l'interpréteur rencontrera une seconde fois cette fonction, il ne cherchera plus à traduire le code mais exécutera directement le code déjà compilé.

1.6.) Machines concrètes / abstraites.

1.7.) La notation BNF (Backhus-Naur Form).

Backhus : Un des principaux concepteurs de Fortran.
Naur : Un des concepteurs d'Algol.

1.7.1.) LA SYNTAXE ET LA SEMANTIQUE.

La **syntaxe** décrit les formes correctes que peuvent prendre les éléments d'un programme.
La **sémantique** correspond à la signification des formes syntaxiques (c'est à dire aux constructions) d'un langage.

En simplifiant, on peut dire que, pour une langue déterminée, un dictionnaire permet de connaître la sémantique de la langue, alors que la grammaire décrit la syntaxe (règles de conjugaison, accord des participes, règles du pluriel des mots, etc.). De façon plus générale, étant donné un ensemble d'éléments, l'informatique ne s'intéresse qu'aux règles permettant de combiner ces éléments entre eux, c'est-à-dire aux structures syntaxiques définies sur cet ensemble et aux règles opératoires permettant de passer d'une structure à une autre, en faisant systématiquement abstraction de toute sémantique.

La syntaxe est habituellement exprimée de façon formelle par la notation BNF (Backhus-Naur Form) encore appelée forme normale de Backhus.

BNF est donc un métalangage, c'est à dire un langage utilisé pour décrire d'autres langages.

Une définition BNF est composée de :

- Un ensemble de **symboles terminaux** (qui sont les éléments du langage décrit)
- Un ensemble de **symboles non-terminaux** qui sont définis par des productions qui comprennent une ou plusieurs règles de réécriture.
- Une **série de productions**. Une production comprend une partie gauche où figure le non-terminal défini et une partie droite qui est composée d'une suite d'un ou plusieurs terminaux ou non-terminaux.

Exemples :

```
<chiffre> ::= 0|1|2|3|4|5|6|7|8|9
<entier> ::=    <chiffre>
           |<entier><chiffre>
<lettre> ::= a|b|c|d|...|y|z
<souligne> ::= _
<identificateur> ::= <lettre>
                |<identificateur><chiffre>
                |<identificateur><lettre>
                |<idenficateur><souligne>
```

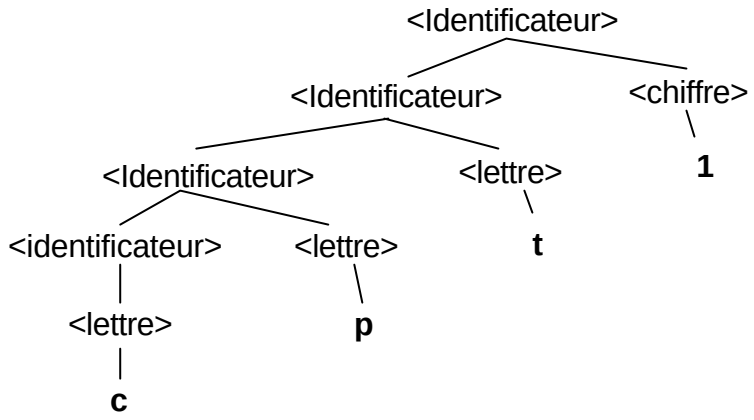
En BNF, un non-terminal s'écrit entre chevrons ('<' et '>').
Le signe "`::=`"signifie "est défini par".
Le signe "`|` " signifie "ou" et permet de regrouper les différentes règles d'une production.

D'après les règles ci-dessus, *cpt1* est un identificateur car il peut être dérivé de *<identificateur>* de la manière suivante :

```
<identificateur>
<identificateur><chiffre>
<identificateur><lettre><chiffre>
<identificateur><lettre><lettre><chiffre>
<lettre><lettre><lettre><chiffre>
c<lettre><lettre><chiffre>
cp<lettre><chiffre>
cpt<chiffre>
cpt1
```

A chaque étape, on remplace le non-terminal le plus à gauche par la partie droite d'une des règles qui le définissent. La suite des terminaux et non-terminaux en cours de dérivation sont des **protophrases** ou **formes sententielles**. La dernière **protophrase**, qui ne contient aucun non-terminal est une **phrase**.

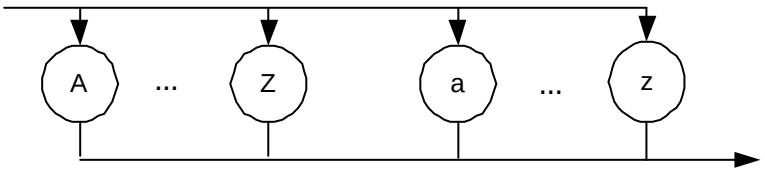
On peut mieux voir la structure d'une dérivation en utilisant un **arbre de dérivation**.



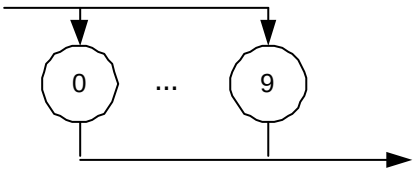
1.7.2.) LES DIAGRAMMES SYNTAXIQUES.

Les diagramme syntaxiques (dits diagrammes de Conway) permettent d'illustrer de manière graphique la syntaxe d'un langage.

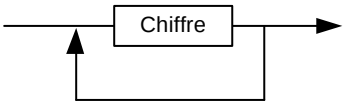
Lettre



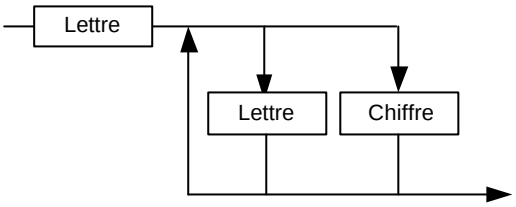
Chiffre



Entier



Identificateur

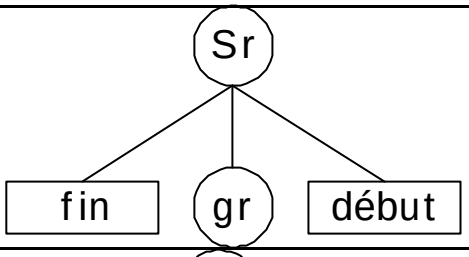
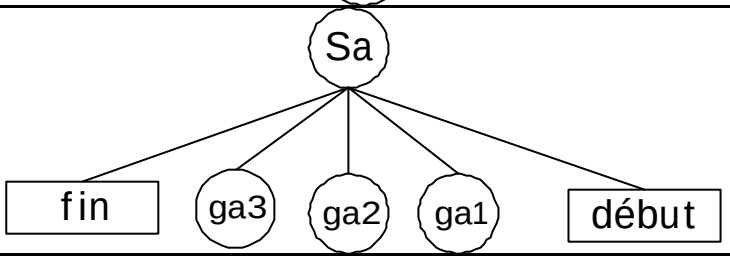


2.) Notations.

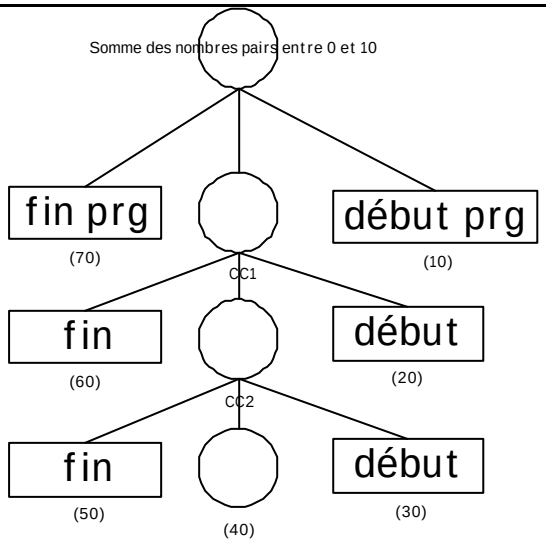
2.1.) Les ordinogrammes.

2.2.) Les arbres programmatiques.

Les arbres programmatiques ont été surtout utilisés pour enseigner la programmation structurée. Avec cette modélisation, tout programme informatique se résume à l'assemblage de deux structures de base : la répétitive et l'alternative.

Structure répétitive :	
Structure alternative :	

En programmation structurée, l'arbre programmatique se construit du haut vers le bas et de droite à gauche.



2.3.) La notation en pseudo-langage.

3.) Annexe.

LANGAGE	ORIGINE DU NOM	ANNÉE	APPLICATION/REMARQUES
FORTRAN	FORmula TRANslation	1954	Destiné tout d'abord aux scientifiques et aux ingénieurs, ce langage compilé de haut niveau est maintenant universel. Il est à l'origine de plusieurs concepts tels que : variables, instructions conditionnelles et compilation séparée de sous-programmes.
COBOL	COmmon Business-Oriented Language	1959	Langage de programmation proche de l'anglais, particulièrement attaché à la structure des données. Très employé, surtout en gestion.
ALGOL	ALGOrithmic Language	1960	Premier langage de programmation structuré et procédural. Il a introduit des concepts dont se sont inspirés de nombreux langages (Pascal, C, ou ADA, qu'on appelle parfois langages de type ALGOL).
LISP	LISt Processing	1960	Langage de programmation orienté-liste et interpété, employé principalement dans la manipulation de listes de données. Souvent employé dans la recherche, il est généralement considéré comme le langage "standard" en intelligence artificielle.
APL	A Programming Language	1961	Langage interprété possédant un grand nombre de symboles spécifiques et une syntaxe concise. Employé surtout par les mathématiciens.
PL/1	Programming Language One	1964	Langage de programmation complexe, conçu pour regrouper les principaux éléments du FORTRAN, du COBOL et de l'ALGOL. Ce langage compilé et structuré est capable de traiter les erreurs et de travailler en mode multitâche. Il est utilisé dans certaines universités et certains centres de recherche.
BASIC	Beginners All-Purpose Symbolic Instruction Code	1965	Langage de programmation de haut niveau, très populaire. Souvent utilisé par les programmeurs débutants.
LOGO	Dérivé du grec *logos*, qui veut dire parole, discours	1968	Langage de programmation destiné principalement aux enfants, à usage éducatif. Il comporte un environnement simple de dessins et un ensemble d'instructions dérivées du LISP.
PILOT	Programmed Inquiry, Language Or Teaching	1969	Langage de programmation principalement utilisé pour créer des applications d'enseignement assisté par ordinateur. Il comporte une syntaxe très réduite.
FORTH	FOuRTH generation language	1970	Langage structuré et interprété, que les programmeurs peuvent facilement étendre. Il offre un grand nombre de fonctionnalités dans un espace réduit d'instructions.
Pascal	Blaise PASCAL, mathématicien et inventeur de la première machine à calculer	1971	Langage compilé et structuré, dérivé d'ALGOL. Il ajoute des types de données et des structures tout en simplifiant la syntaxe. Comme le langage C, c'est un langage de développement standard pour les micro-ordinateurs.
C	C'est une version améliorée du langage de programmation B du Bell Laboratory, créé en 1972	1972	Langage de programmation structuré et compilé, très largement employé.
PROLOG	PROgramming in LOGic	1972	Langage de programmation utilisé en intelligence artificielle. Il manipule des relations logiques plutôt que mathématiques.
ADA	Augusta ADA Byron (Lady Lovelace)	1979	Dérivé du Pascal, développé par le Département de la Défense américain.
Modula-2	Langage MODULAire, conçu comme deuxième forme du Pascal (conçus tous deux par Niklaus Wirth)	1980	Langage de haut niveau utilisant la programmation modulaire. Dérivé du Pascal, il se caractérise par ses carences en fonctions et procédures standard.
Camel	Categorical Abstract Machine Language	1984	Langage fonctionnel développé et distribué par l'INRIA depuis 1984

3.1.) Exercices de cours.

EXERCICE 3.1.1..

Notation de nombres négatifs : principe du complément à deux. Pour un nombre codé sur 16 bits, le seizième bit vaut $-1 * 2^{15}$.
Quel est le plus grand nombre positif qu'on puisse écrire ? 32767
Le plus grand nombre négatif ? -32768

Base 10	Base 2
123	0000000001111011
-123	1111111110000101
-1	1111111111111111
0	0000000000000000
1	0000000000000001

EXERCICE 3.1.2.:

Donnez une notation BNF pour :

un nombre entier sans signe.
un nombre réel sans signe qui doit comprendre au moins un chiffre avant et après la virgule.

EXERCICE 3.1.3.:

Donnez la liste de toutes les phrases syntaxiquement correctes composées d'un, deux ou trois caractères et qu'il est possible de dériver avec la grammaire EBNF suivante :

```
sequence->'A' {'B'|'C'} ['D']
```

EXERCICE 3.1.4.:

Ecrire un algorithme fini qui calcule la racine carré d'un nombre N en utilisant la méthode de calcul suivante :

```
1) Remplacer X par 1 ;
2) Diviser N par X ;
3) Additionner X et prendre la moitié ;
4) Remplacer X par le résultat obtenu ;
5) Continuer sous 2)
```

Par exemple, pour n=2 on obtiendra successivement 1, 1.5, 1.146, 1.413 ...

3.2.) Exercices.

EXERCICE 1.1.

Soit le programme suivant :

```
PROGRAM puissance;

VAR X,N : ENTIER;
    a   : ENTIER;
    i   : ENTIER;
    ETIQUETTES 1,2;

DEBUT
     AFFICHER('Ce programme calcule X … la puissance N');
     AFFICHER('Donnez-moi X svp : ');LIRE(X);
     1:
     AFFICHER('Donnez-moi N svp : ');LIRE(N);
     SI N<0 ALORS
     DEBUT
          AFFICHER('N doit etre positif svp.');
          ALLER EN 1;
     FIN SI;
     a<-X;
     i<-0;
     2:
     i<-i+1;
     a<-a*X;
     SI i<(N-1) ALORS ALLER EN 2;
     AFFICHER(X,' à la puissance ',N,' = ',a);
FIN.
```

Montrez que ce programme est faux. Réécrivez ce programme pour qu'il fonctionne correctement.

EXERCICE 1.2.

Ecrivez en pseudo-code une fonction calculant le jour de la semaine correspondant à une date suivant la formule de Zeller :

- On pose m=mois, j=jour.
- Si le mois *m* est supérieur ou égal à 3 on le change en *m*-2 et sinon en *m*+10 ainsi que l'année en *a*-1.
- On pose *na* = numéro de l'année dans le siècle et *s* = numéro du siècle.
- On calcule *f* = *j*+*na*-2**s*+*na* DIV 4+*s* div 4 + (26**m*-2) div 10.
- Le résultat est donné par f mod 7 (0=dimanche, 1=lundi, 2=mardi...).

Vérifiez l'algorithme en le déroulant pour plusieurs exemples.

EXERCICE 1.3.

En supposant que le paramètre *n* du programme ci-dessous est une puissance positive de 2 et n>0 (c'est à dire *n*=2,4,8,16...), donner la formule exprimant la valeur de la variable *compteur* en fonction de celle de *n* après exécution de la procédure.

```
PROGRAM mystere;

VAR x,compteur,n : ENTIER;

DEBUT
 compteur<-1;
 x<-2;
 AFFICHER('Donnez n :');LIRE(n);
 TANT QUE x<n FAIRE
      x<-2*x;
      compteur<-compteur+1;
 FIN TANT QUE
 ECRIRE(compteur);
FIN.
```

EXERCICE 1.4.

Ecrivez l'algorithme qui multiplie deux nombres entiers (N1 et N2) en utilisant la méthode suivante :

On place des 2 nombres sur 2 colonnes puis on double le facteur de gauche tout en divisant par 2 celui de droite (quotient entier) jusqu'à obtenir 1 à droite. La somme des nombres qui sont placés à gauche d'un nombre impair donne alors le résultat.

Exemples :

Multiplication de 21 par 12			**Multiplication de 263 par 42**	
~~21~~	12		~~263~~	42
~~42~~	6		**526**	**21**
84	**3**		~~1052~~	10
168	**1**		**2104**	**5**
252			~~4208~~	2
			8416	**1**
			11046	

On suppose qu'on dispose d'une fonction `DIV` qui effectue la division entière (ex. `5 DIV 2 = 2`) et d'une fonction `PAIR` qui renvoie vrai si son argument est pair (ex. `PAIR(5) = FAUX`).