Contents

Chapitre 1 Algorithme et LDA

I. Notions d'algorithmes

L'informatique a pour objet le *traitement automatique de l'information* considérée comme support des *connaissance*s du monde réel. Le traitement dont il est question ici se fait par un ordinateur[1] ou calculateur. Ce dernier ne peut effectuer des traitements sur des données que parce qu'il reçoit de la part du (des) programmeur (s) des *instructions* (liste des actions à exécuter) lui indiquant comment effectuer ces traitements. Ces instructions écrient en langage machine (Pascal, langage C, etc.) sont d'abord décrites de façon claire (sans ambiguïtés) à l'aide d'*algorithmes*. L'algorithme sert donc au (x) programmeur (s) à décomposer un problème en sous-problèmes (problèmes plus simples) afin d'en faciliter la résolution. On pourrait le définir comme l'énoncé d'une suite d'opérations permettant de donner la réponse à un problème. Voici une autre définition est la suivante : « un algorithme est une procédure de calcul bien définie, qui prend en entrée une valeur, ou un ensemble de valeurs, et qui produit en sortie, une valeur ou un ensemble de valeurs ». Un algorithme est donc une séquence d'étapes de calcul permettant de passer de la valeur d'entrée à la valeur de sortie.

Plus formellement, un algorithme A est une multi-application de l'ensemble des données en entrées vers l'ensemble des données résultats tel que pour tout E données d'entrée et R résultat d'un problème, $A(E) = R$

La programmation consiste à se donner un E et un R et à fournir A (éventuellement exploitable sur une machine donnée).

L'algorithme A est spécifié sur la base d'une décomposition en un nombre fini d'opérations $O_k, k = 1, \dots, n$ avec $A = O_n \circ O_{n-1} \circ \dots \circ O_1$

$$E_1 = E$$
$$O_{k-1}(E_{k-1}) = E_k \, , k = 2, \dots, n-1$$
$$O_n(E_n) = R$$

L'état de sortie (ou prédicat de sortie) d'une opération est l'état d'entrée (prédicat d'entrée) de l'opération suivante.

Une fois un algorithme d'un problème trouvé, il faut le transcrire dans un *langage de programmation* (plus simple que le français dans sa syntaxe). L'algorithme transcrit en langage de programmation est appelé *code*. Celui-ci est compiler (pour le rendre compréhensible par la

[1] Machine automatique qui permet d'effectuer dans le cadre de la programmation des ensembles d'opérations arithmétiques et logiques à des fins scientifiques, administratives ou comptable

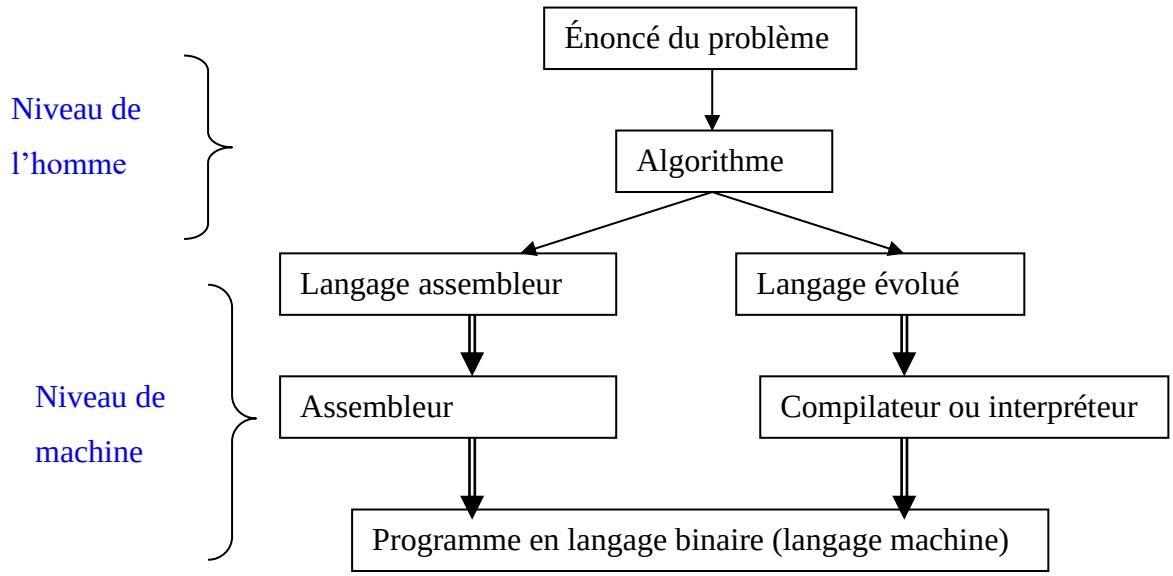
machine) et exécuté par un *compilateur* (celui du langage : compilateur Pascal, compilateur C, etc.) pour donner la réponse au problème.

Apprendre l'algorithmique, c'est apprendre à manier la structure logique des programmes.

II. Résolution de problème et structure d'un algorithme

1. Différentes étapes de déduction d'un programme à partir d'un problème

Un algorithme peut être vu comme un outil de résolution d'un problème calculatoire bien défini. L'*énoncé du problème* spécifie en termes généraux la relation désirée entre l'*entrée* et la *sortie*. L'algorithme décrit une procédure de calcul spécifique permettant d'établir cette relation.
La résolution automatique d'un problème passe par trois des étapes décrites par l'organigramme suivant.



Dans ce chapitre nous avons fait le tour de toutes les étapes importantes à la résolution d'un problème avec un ordinateur. La figure suivante résume ces différentes étapes.

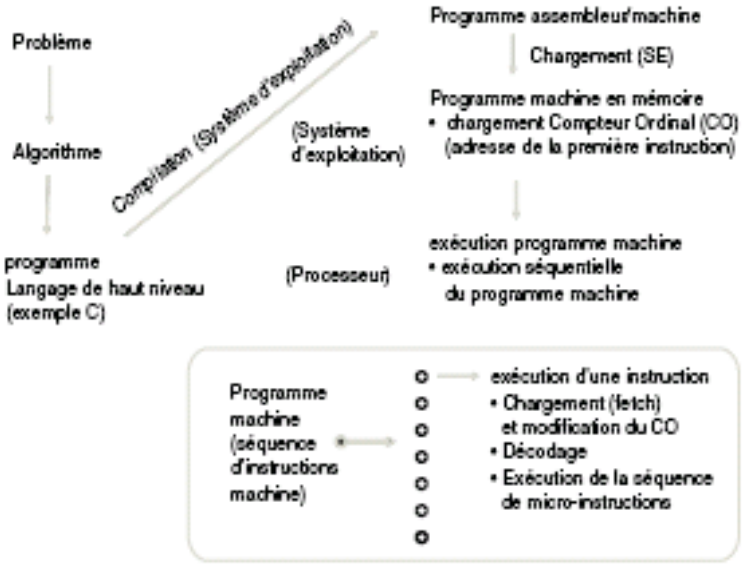


Figure 5.1 *Principales étapes de la résolution d'un problème avec l'ordinateur*

L'Enoncé du problème ou *cahier des charges* doit préciser les données en entrées au problème, les besoins en traitement et les résultats souhaités. Ainsi, l'étape Enoncé-programme va se décomposer en deux :

- Enoncé- algorithme : a ce niveau la question de l'analyse et de la conception d'une solution informatique du problème spécifié par son énoncé est posée. Ce travail devra aboutir à une description algorithmique de ce traitement que l'on appelle algorithme.
- Algorithme-programme : la question de l'obtention d'un programme (avec toutes les préoccupations techniques qui sont légitimes) est ramenée à une phase de production par traduction en un langage cible choisi

  2. Moyen à mettre en œuvre pour passer d'une étape à l'autre

- Passage de l'algorithme au programme : C'est la transition qui doit être la plus facile dans la mesure où l'algorithme est un automate décrivant rigoureusement la suite finie des opérations permettant de faire passer les données d'un état d'entré à un état de résultat : solution du problème demandé. Il s'agira essentiellement d'une réécriture de l'algorithme dans un langage cible de programmation sur une machine cible. C'est une phase qui doit être indépendante de toute l'activité d'analyse et de conception.
- Passage de l'énoncé à l'énoncé explicite : l'énoncé fournit une propriété générale du résultat et ne permet pas toujours d'obtenir la solution algorithmique. Par contre l'énoncé explicite doit spécifier le procédée de calcul permettant d'obtenir l'algorithme. Il s'agit essentiellement d'une analyse de l'énoncé de manière à :
  - Identifier les informations fondamentales manipulées (reconnaissance)
  - Classer ces informations en entrée (hypothèse du problème), sortir (résultat ou conclusion du problème) et de définir leur organisation (compréhension)
  - Proposer une solution explicite des procédés de calcul et de mise en œuvre qui permettent d'obtenir ces résultats (modélisation)

  Les outils utilisés dans cette description dépendent évidemment de la nature du problème. Ils peuvent être graphique, de type texte, algébrique, etc.
- Passage de l'énoncé explicite à l'algorithme : l'algorithme sera obtenu par réécriture de l'énoncé explicite en incluant un ordre dans le procédé de traitement. Cela nécessite un langage adapté à la reformulation que l'on appelle Langage de Description (ou notation) Algorithmique ou LDA. Comme tout langage, ce langage se définit au mois par deux ensembles : Un ensemble formé du vocabulaire général du lange et un autre décrivant sa grammaire.

Nous allons développer ce dernier point.

III. Le LDA (langage de description algorithmique)

a. Définition

- Un texte du LDA est la concaténation de mots de V_t séparé par un ou plusieurs espaces et respectant rigoureusement les règles grammaticales correspondantes et édictées dans G.
- Pour des raisons de lisibilité et de clarté de l'écriture les verbes du langage seront soulignés pour les mettre en évidence par rapport aux objets du programmeur.
- Un texte du LDA représente un algorithme et a impérativement la structure générale suivante :

ALGORITHME identificateur ;
Description des Ressources de l'algorithme
{
Description de la séquence des opérations à effectuer.
}

b. Syntaxe

- Ce texte algorithmique commence nécessairement par le mot clef ALGORITHME et se termine par un point (fin de l'algorithme).
- Le corps du texte est formé de deux parties : une première qui sert à décrire toutes les ressources (données ou actions) utilisées dans le procédé opératoire de l'algorithme qui sera décrit dans la deuxième partie. D'où découle la conséquence syntaxique suivante : il ne peut y avoir de ressources utilisées dans le traitement qui n'aient été décrites
- Règle syntaxique de construction d'un identificateur : un identificateur est le nom donné par le programmeur à tout objet manipulé dans l'algorithme. La règle de construction est : un identificateur est une suite de caractère commençant par un caractère alphabétique. Exemple : somme, impôt, salaire_de_base. Tous les caractères ont le même niveau d'écriture : pas d'indice ni d'exposant. Il n'y'a pas de limitation du nombre de caractères à utiliser, cependant le bon sens nous amène à profiter de cette souplesse de manière à créer les identificateurs s'approchant le plus possible de la signification de l'objet qu'ils désignent sans peut-être exagérer par sa longueur.

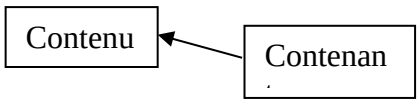
Pour construire des identificateurs pertinents, on pourra se référer aux indications suivantes :

- Choisir des identificateurs facilement prononçables
- Eviter les abréviations non courantes
- Choisir des constructions de préférences courtes et dans le cas des compositions de mots utiliser un caractère de concaténation (comme le sous tiret _) pour marquer la suite des sens véhiculés par l'identificateur. Exemple : bourse_etudiant_inphb
- Identifier distinctement les informations, chaque fois que cela est possible. L'identificateur bourse_etudiant_inphb est plus évocateur que bourse.

Par ailleurs, le texte algorithmique est constitué de phrases logiques séparées par le point-virgule. Par conséquent, il n'y'a pas forcément une phrase par ligne (de notre feuille), seules des considérations de *lisibilité* et de *clarté* peuvent nous amener à ne pas effectivement trop la surcharger. Comme exemple, la première phrase de notre LDA présenté ci-dessus sous sa forme générale est **ALGORITHME** identificateur ;

c. Description des Ressources Données

Toute données, qu'elle soit d'entrée ou de sortie ou intermédiaire de calcul, manipulée dans la partie description des ressources d'un algorithme est caractérisée par la double indication de



son *contenant*, représenté par le nom de la donnée ou identificateur et son *contenu*, valeur de la constante prise par la donnée.

Remarque

- Le contenu d'une donnée peut varier d'un instant à l'autre mais seul le dernier est évidemment conservé et le contenant lui à priori ne change jamais.
- Une *donnée* est souvent appelée *variable*, ce qu'il faut accepter avec la description ci-dessus (seuls les contenus peuvent varier …)

d. Quelques types

Concernant les types, certains sont bien sûr prédéfinis, ce sont ceux qui seront considérés comme de *base* et permettront par construction d'obtenir des types plus sophistiqués.
Dans la classification simplifiée des types on distingue les *types simples* dont les constantes atomiques (constituées d'un seul tenant ou indécomposable) et les *types organisés* dont les constantes sont composées d'autres constantes.

- Quelques types simples : les entiers relatifs, les nombres réels, les constantes logiques et l'ensemble de tous les caractères du LDA sont désignés respectivement par les mots clés **ENTIER, REEL, LOGIQUE et CARACTERE**.

Type	Description	Exemple
ENTIER	est décrit en décimal signée, le signe + pouvant être omis s'il est positif.	On a les exemples suivants. 1969, -56.
REEL	est aussi écrit en décimale signée avec le point (ou la virgule) pour séparer la partie entière de la partie réelle	651.48, -5,07.
LOGIQUE	est formé des deux constantes VRAI et FAUX.	
CARACTERES	Représente l'ensemble de tous les caractères (en d'autres termes son alphabet). Les constants caractères sont mis entre quotte.	On a les exemples suivants. 'a', 'A', '1'

- Le type organisé : On dit qu'une information est organisée ou structurée si elle est une collection finie et dénombrable d'autres informations. Elle est alors dite *composite* et chacune des informations qui la composent (ou composantes) peut être aussi structurée de la même manière. Il apparait une ''récursivité'' au niveau de la définition de l'objet structuré du fait qu'il se définit par lui-même, la condition d'arrêt de reproduction de la définition est qu'au dernier niveau les objets soient de types simples : entier, réel, logique, etc.).

  Le type TABLEAU est un cas particulier ou l'objet est une collection d'objets de même type et en nombre fixe. Chaque élément du tableau est appelé *composante* et le domaine de valeurs est le produit cartésien des domaines de valeurs de chaque composante. La syntaxe de désignation du type est : TABLEAU [indice de début .. indice de fin] DE type d'une composante. Indice de début et indice de fin sont des constantes d'un type simple quelconque mais ordonné totalement. Ils permettront de fixer le nombre de composantes et de pouvoir identifier chaque composante par un indice dont le contenu est compris dans l'intervalle [indice de début ; indice de fin].

La syntaxe d'une description de ressources données est la suivante :

Liste d'identificateurs : type

Liste d'identificateurs est soit un identificateur soient plusieurs identificateurs (ayant le même type) séparés par une virgule. Par exemple, on a :

```
sal1, sal2          : REEL ;
Etat                : LOGIQUE ;
i, j                : ENTIER ;
catégorie           : CARACTERE ;
```

Dans le cadre de notre LDA, nous avons la possibilité d'identifier des types. Un intérêt possible est le cas où une description de types est utilisée à plusieurs endroits de l'algorithme (nous y reviendrons) mais aussi pour apporter de la clarté dans les descriptions de type organisés :

```
Identificateur d'un type=type
```

Identificateur de type est le nom donné au type et type est la description du type. Comme exemple, on a :

```
occupe=LOGIQUE ;
siege=occupe ;
compartiment=TABLEAU[1..6] DE siège ;
wagon=TABLEAU[1 .. 12] DE compartiment ;
train=TABLEAU[1 .. 10] DE wagon ;
untrain : train
```

Occupe, siège, compartiment, wagon et train sont des noms de type. Ils doivent servir de descripteurs de données. Dans notre cas seul la donnée untrain est créée (instanciation) et décrite comme de type train. L'accès à une composante se fait par l'intermédiaire d'une variable de type simple correspondant au type de l'intervalle.
Ainsi si i, j, k : ENTIER ; est une déclaration alors untrain[i,j,k] est un logique indiquant l'état d'occupation du siège k du compartiment J du wagon i du train.

e. Quelques primitives

Ce sont des facilités d'écriture qui permettent une action spécifique comme l'affectation, les expressions arithmétiques et logiques et les entrées-sortie.

- *L'affectation* : le symbole utilisé est la flèche renversée ⟵. On a donc la syntaxe suivante :

  ```
  identificateur⟵ expression du même type.
  ```

  La constante provenant de l'évaluation de l'expression est transférée comme contenu de l'identificateur, ce qui impose qu'elle soit du même type que celui de

l'identificateur. Une expression garde le sens habituel d'une formule comportant des opérateurs et des opérandes. Elle est évaluée de la gauche vers la droite en respectant les ordres de priorité prédéfinis.

- *Les expressions arithmétiques* : les opérandes sont des données de type ENTIER ou REEL, et les opérateurs concernés sont :

Opérateurs	Noms
+	Addition
-	Soustraction
*	Multiplication
/	Division réelle
DIV	Division entière
MOD	Reste de la division entière

L'ordre de priorité est le même qu'habituellement, c'est-à-dire dans décroissant :

	Règle de priorité
()	Le parenthésage pour forcer la priorité
* / mod div	Même priorité
+ -	Même priorité

- *Les expressions logiques* : celles prédéfinies sont : la négation (NON), la conjonction ET et la disjonction OU.

  Les opérateurs relationnels sont applicables aux données de types simples ordonnées.

  Ce sont :

Opérateurs relationnel	Noms
<	Strictement inférieur
<=	Inférieur ou égal
>	Strictement supérieur
>=	Supérieur ou égal
=	Egalité
<>	Différent

Comme exemple on a :

```
x1, x2 :   ENTIER
L          : LOGIQUE
L'action suivante est correcte L← x1=x2
```

- Les ENTREE/SORTIE : les primitives d'entrée/sortie vont permettre d'assurer le transfert de constantes entre l'algorithme et une unité périphérique quelconque. Pour l'instant nous nous limiterons au cas où l'unité périphérique est l'écran-clavier.
  En entrée le verbe utilisé est LIRE. Sa syntaxe est

```
LIRE(liste d'expression de type simple)
```

f. Les structures de contrôles fondamentales

On distingue trois structures de contrôle fondamentale : la séquence, l'alternative et l'itérative.

- *La séquence* : elle a pour effet de séquencer dans le temps deux actions. Syntaxiquement on utilise le point-virgule (;) comme séparateur d'actions (action1 ; action2, exprime qu'action 1 s'exécute avant action 2). En guise d'illustration considérons l'algorithme suivant.

```
ALGORITHME identificateur ;
A, B, C, D                    : ENTIER;
{A←1 ; B← 3 ;
 C←A+B ; D←A-B ;
 A←C+2*B ; B←C+B ;
 C←A*B ; D← B+D; A← D*C
}.
```

Dans l'algorithme précédent la dernière action n'est pas suivie de point-virgule puisqu'aucune action ne vient en séquence. Par ailleurs on a pris la précaution de n'utiliser aucune donnée dans une expression qui n'ait un contenu au préalable (soit par initialisation soit par calcul). Pour voir la séquence des opérations on construit le tableau de simulation qui permet de suivre le déroulement successif des états des données de l'algorithme (le dernier état sera état des résultats).

A	B	C	D
1	3		
		4	
			-2
10	7		
		70	
			5
350			

On voit bien que l'algorithme calcule effectivement les contenus de A, B, C, D à partir de A et B, mais ne donne aucun moyen de s'en apercevoir. Il est judicieux, de

terminer le traitement par une instruction ECRIRE. De même les calculs étant toujours réalisés pour les mêmes contenus, il peut être intéressant de rendre l'algorithme plus général en remplaçant les instructions d'initialisations par une instruction LIRE. Soit :

```
ALGORITHME identificateur ;
A, B, C, D                    : ENTIER;
{
  LIRE(A, B) ;
 C←A+B ; D←A-B ;
 A←C+2*B ; B←C+B ;
 C←A*B ; D← B+D;
A← D*C
ECRIRE(A,B,C,D)
}.
```

Cette organisation des traitements est une bonne structure de calcul : en effet chaque fois qu'il est possible de rassembler toutes les initialisations au début, de terminer par toutes les opérations d'affichage et entre les deux la partie calcul vous créez une disposition qui peut apporter de la clarté et de la lisibilité à votre algorithme.

- *L'alternative* : Supposons qu'on ait deux actions : action1 et action2. On voudrait à partir de l'évaluation d'une expression logique condition exécuter exclusivement l'une ou l'autre. La syntaxe est :

```
SI (condition) ALORS action1 SINON action2
```

Une illustration est donnée par l'exemple qui suit.

```
SI (x <= y) ALORS
                x ← x + y
            SINON
                x ← x − y
```

Une variante alternative au cas où action2 est vide, en d'autres termes, on voudrait juste exécuter une action si une condition était réalisée. La syntaxe suivante est admise et s'appelle la conditionnelle :

```
SI (condition) ALORS action1
```

Remarquons que, dans le cas où action1 ou action 2 est composé de plusieurs actions, un problème de syntaxe peut se poser. Cependant, si nous avons besoin de faire considérer syntaxiquement un groupe d'action comme une seule, cela peut être réalisée

par la primitive dite de blocage d'actions (on met toutes les actions concernées entre accolade) :

```
SI (condition) ALORS
                { action11 ;
                  action12 ;
                  action13
                  }
                 SINON
                        { action21 ;
                          action22
                         }
```

- *L'itérative* : comme son nom l'indique, on dispose d'une action (élémentaire ou bloquée) : action et d'une expression logique : condition. La phrase suivante :

```
TANT QUE (condition) FAIRE action
```

A pour effet d'exécuter action autant de fois que nécessaire et de ne s'arrêter que si condition bascule à FAUX. Action peut n'être exécuté aucune fois (si dès le départ condition est FAUX) ou plusieurs fois, ce qui implique que dans action des opérations sont prévues pour faire basculer à un moment donné condition à FAUX sinon on aurait un traitement infini ce qui n'est pas algorithmique. Une variante de l'itérative : LA REPETITIVE : supposons qu'on veuille qu'action soit faite au moins une fois avant de rentrer dans une itérative, situation qui est courante. On peut écrire :

```
action ;
TANT QUE (condition) FAIRE action
```

Une syntaxe est prévue pour ce cas

```
REPETER
 action
JUSQUA (condition)
```

Action est exécutée au moins une fois ensuite condition est évaluée. Si condition est VRAI le processus est arrêté sinon l'itération est reprise. On remarquera que le sens du test d'arrêt est contraire à celui de l'itérative et par construction action est faite au moins une fois. Si action est composée de plusieurs il n'est pas utile de le bloquer puisque REPETER JUSQUA joue déjà ce rôle : formellement il n'y'a pas

d'ambiguïté. Une caractéristique de ces deux ordres d'itération est qu'on ne connait pas à priori le nombre de fois qu'i faut exécuter action. Lorsque ce nombre est connu, on utilise la variante : BOUCLE :

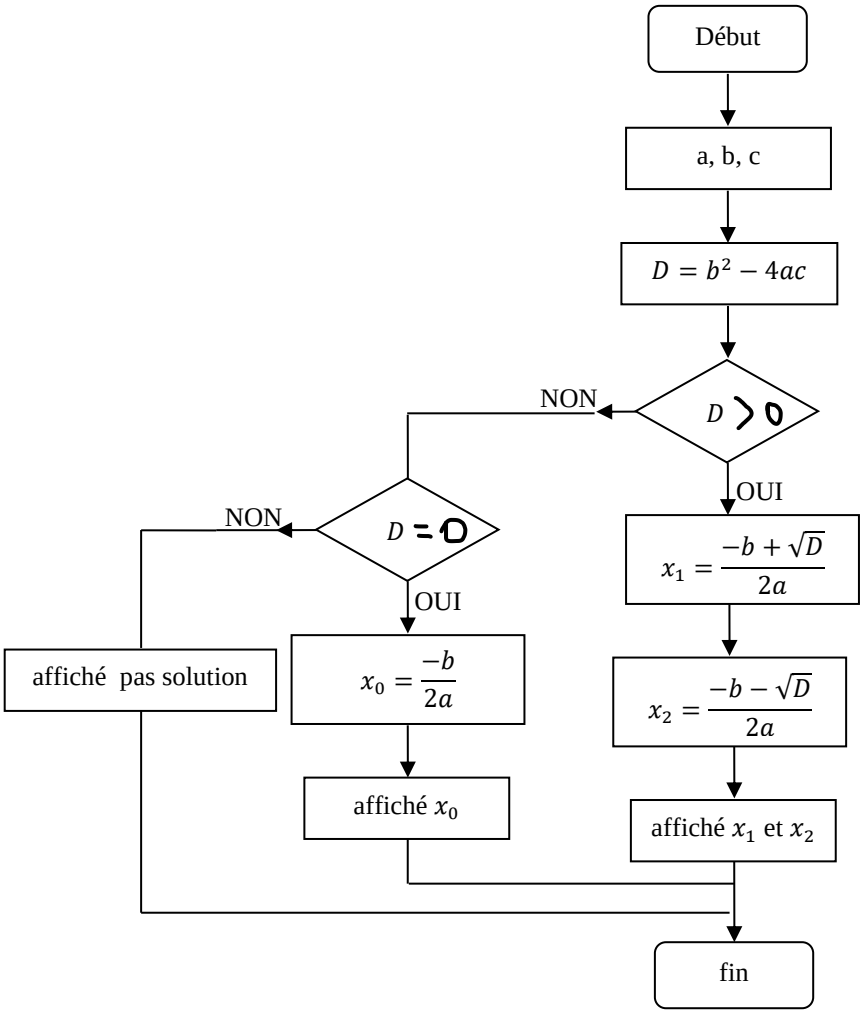
POUR identificateur←expression initiale JUSQUA expression finale FAIRE action

Identificateur va jouer le rôle de compteur, il doit donc être d'un type simple énumérable et ordonné, d'où pour le moment ENTIER, CARACTERES, et LOGIQUE ; traduisons cette variante en utilisant TANT QUE avec la convention suivante succ(identificateur) dépend du type d'identificateur est la constante successeur dans l'ordre correspondant au type de l'identificateur.

```
identificateur←expression initiale ;
TANTQUE (identificateur→expression finale) FAIRE
                { action ;
                   identificateur←succ(identificateur)
                  }
```

Exemple d'écriture d'algorithme sous forme d'organigramme

Organigramme



Application

Ecrire l'algorithme (en utilisant un organigramme) de la résolution d'un système linéaire de deux équations à deux inconnues par la méthode de Cramer (ou des déterminant)

Pseudo-code

Notre pseudo-code pour la résolution de l'équation du second degré se présente sous la forme d'une procédure appelé POLY2-SOLVE. Elle prend comme paramètre un tableau …qui contient les coefficients d'une équation du second degré à résoudre

- Analyse des données en entrées et sortie, méthode de résolution, écriture de l'algorithme
- Traduction dans *pseudo-code* ou *langage de description des algorithmique* (LDA)

IV. Les langages de programmation

1. Le langage machine

Un ordinateur ne ''comprend'' qu'un seul langage dit langage machine (de première génération) que son unité de contrôle est capable d'analyser et d'exécuter. C'est le langage qui se situe au niveau le plus bas et ses caractéristiques sont celles vue précédemment.

2. Le langage d'assemblage

Vue la complexité des langages machines, les constructeurs ont développé un autre type de langage dit langage d'assemblage (deuxième génération) moins ésotérique et plus facile à utiliser. Il est constitué de la manière suivante :

- Toute instruction machine est représentée par une et une seule instruction en langage d'assemblage.
- Les codes opérations qui étaient binaires dans le langage machine deviennent mnémonique, exemple : ADD au lieu du code binaire de l'addition
- Les adresse des opérandes ne sont plus gérer par leur valeur numérique mais par des symboles qui les représentent qu'on appelle identificateurs.

Par exemple pour réaliser l'instruction $X = A + B$ on pourrait avoir : $ADD\ A, B, X$ ce qui est plus parlant. Ce programme ne sera pas directement exécutable car il nécessite une phase de traduction en langage machine. L'utilitaire chargé de cette traduction est appelé *assembleur*.

Programme en langage d'assemblage	ASSEMBLEUR →	Programme en langage d'assemblage

3. Les langages évolués

- *Les langages de troisième génération* : ces langages se rapprochent des langages naturels mais sont définis par des règles de grammaires précises. Elles ont facilité la conception et la programmation et ont permis un développement fulgurant de la science informatique (en se détachant de la machine par leur universalité) et de la facilité d'utilisation des ordinateurs. L'informatique devient d'un accès beaucoup plus aisé et tous les domaines des sciences humaines sont touchés.

- *Les langages de quatrième génération* : environnement logiciel offrant des outils prêt à l'emploi que l'on peut aisément utiliser pour concevoir une application, un projet, etc. La programmation reste bien sûr algorithmique mais elle s'élève à des niveaux d'abstraction plus proches des méthodes naturelles de résolution qu'un utilisateur adopte face à un problème. Un programme n'est plus uniquement un processus (composition d'instruction) permettant de faire passer les données (manipulées par ces instructions) d'un état d'hypothèse (entrée) à un état de conclusion (résultat) mais plutôt abstrait à être une collection d'objets (entité représentant une catégorie quelconque d'informations caractérisé par son état (ses données) et son comportement (les traitements spécifiques qu'elle peut réaliser) s'envoyant des messages (demande d'informations) en vue d'amener le système à un état désiré.

EXERCICES

1. Calculer le salaire d'un fonctionnaire ou auxiliaire, sachant que s'il est fonctionnaire il cotise 6% de son salaire de base pour sa retraite 2.75 % du salaire de base pour sa protection sociale si son salaire est plus petit que le plafond fixé par l'organisme de gestion de la protection sociale, dans le cas contraire il cotise pour 2.75% du même plafond.

   S'il est auxiliaire il ne cotise pas pour la retraite, par contre il cotise pour la protection sociale au taux de 5% de son salaire de base si ce dernier est inférieur ou égal au plafond sinon il cotise pour 6.75% du plafond augmenté de 1.5% de la différence entre le salaire de base et le plafond.
2. Résoudre dans R une équation du second degré
3. Afficher dans l'ordre trois nombres entiers et distincts.
4. On considère un texte formé de caractères et terminé par un caractère unique : '#', on voudrait connaitre :
   a. Le nombre de 'a'
   b. Le nombre de voyelle
   c. Le nombre de 'le'
   d. Le nombre de mots
5. On considère la suite de FIBONACCI :

$$u_0 = u_1 = 1$$
$$u_n = u_{n-1} + u_{n-2}$$

   Calculer la somme des q premiers termes.

   La structure d'un algorithme comporte 3 étapes : la présentation du traitement, le traitement et l'édition des résultats.
6. Ecrire un algorithme qui détermine tous les couples $(m; n)$ avec $1 \leq m, n \leq 2014$ et vérifiant la relation $(n^2 + mn - m^2)^2 = 1$

Chapitre 2 Procédure et Fonction

Si un problème apparait difficile, il faut éviter la recherche directe de la solution ce qui amènerait à penser à plusieurs choses à la fois et à rentrer trop vite dans le détail des actions. Bien au contraire il s'agit :

1. D'apprécier le problème dans sa totalité ensuite de le morceler (… de le décomposer, de le factoriser ou de le raffiner) en des sous-problèmes maitrisables (représentent une fonction du problème analysé) et indépendants (chacun doit représenter une difficulté dans la résolution du problème de départ) dans leur fonction liée dans l'ordre de leur exécution.
2. De spécifier les sous problèmes identifiés, de décrire leur utilisation dans la solution du problème et de procéder à une validation à cette étape.
3. Reprendre chacun de ces sous problèmes spécifiés par un énoncé en leur appliquant le même principe.

Quoique les sous problèmes soient indépendants ils doivent être liés par un enchaînement particulier : la solution du problème. Par exemple, si le problème posé est la résolution d'un système linéaire par la méthode de Gauss, on peut le factoriser en les sous-problèmes suivants :

- Lecture de la matrice
- Lecture du second membre
- Triangularisation de la matrice
- Résolution d'un système linéaire
- Affichage des solutions

Au concept de sous-programme point de départ d'une étape de factorisation nous allons faire correspondre celui de ressources actions.
Syntaxiquement une ressource action va avoir la même structure qu'un algorithme. Elle contient ses propres ressources (données et actions) et son traitement mais sera activé dans la partie action de la ressource père. Comme toute ressource, les ressources actions sont décrites formellement dans la partie description des ressources et sont utilisées dans la partie description des actions. Elles ont plusieurs formes :

- *Procédure sans paramètre* : la syntaxe d'une procédure (ressource action) est la suivante.

```
PROCEDURE identificateur ;
Liste des identificateurs : type
corps d'un algorithme
```

Ainsi, une ressource action est nommée par un identificateur et la structure d'un algorithme devient :

```
ALGORITHME identificateur ;
Liste des identificateurs : type
PROCEDURE identificateur ;
    Description des ressources propres (données et actions)
    { description du processus opératoire de la ressource action
     } ;
{description du processus opératoire de l'algorithme
}.
```

L'algorithme utilisant les procédures est appelé *algorithme principal*, la partie description des actions est appelée *partie principale*. De la même manière toute procédure pouvant contenir des procédures comme ressources propres a aussi une partie principale. Une partie principale comprend un ou plusieurs appels de procédure qui se font en nommant la procédure concernée. Tout se passe comme si la partie principale de la procédure nommée s'insère à l'endroit de l'appel : on dit qu'elle est activée. Pour faciliter la lecture d'un algorithme, le LDA nous permet de décrire une procédure à l'extérieur de l'algorithme principal mais en indiquant dans les parties description des ressources correspondantes les entêtes de leur description suivi du bloc de description des actions vide : ce que l'on appelle *prototype*. Par exemple on peut avoir :

```
ALGORITHME pp
    PROCEDURE p1 ; {}
    PROCEDURE p2 ; {}
{ partie principale de l'algorithme
}.
```

- *Procédure avec paramètres*: dans la partie description des ressources on utilise la syntaxe

```
PROCEDURE (description des ressources paramètres
formels) ;
    corps d'un algorithme ;
```

Où paramètres formels désigne les données identifiées dans la description des paramètres lors de la description de la procédure. Une description de paramètres formels est une séquence de description d'une liste de paramètres formels d'un même type. Les paramètres réels (ou effectifs) sont les données passées lors de l'activation de la procédure, ils peuvent varier d'un appel à l'autre. Les paramètres réels se substituent aux paramètres formels en respectant l'ordre par lequel ils ont été décrits.

- Les fonctions : la fonction est une procédure particulière pour laquelle on privilégie un résultat spécifique à condition qu'il soit unique et de type simple. Ce résultat unique sera constitué par le nom de la fonction au lieu de figurer dans la liste des paramètres formels. La syntaxe d'une fonction est la suivante :

```
FONCTION identificateur : type simple ;
  Corps d'un algorithme
```

```
FONCTION identificateur (déclaration des paramètres formels) : type
simple ;
    Corps d'un algorithme
```

Ainsi, on assimile le résultat de la fonction à son nom. Par conséquent dans la description formelle de la fonction il doit y avoir au moins une action lui donnant le contenu résultat du traitement. L'appel d'une fonction se fait en invoquant directement son nom dans une expression utilisant le résultat de manière correcte et en passant éventuellement tous les paramètres réels. En d'autres termes la fonction rendra une valeur qui peut être utilisée pour initialiser une autre variable ou dans une expression de même type.

EXERCICES

1. Ecrire une fonction qui donne le maximum de trois nombres réels
2. Ecrire une fonction qui prend un tableau de 5 entiers, puis retourne la valeur Vraie ou Faux selon que le tableau est trié par ordre croissant ou non
3. Ecrire une fonction qui calcule le nombre de combinaison
4. Soit f une application de l'ensemble des entiers strictement positifs dans l'ensemble des entiers positifs ou nul, vérifiant les propriétés suivantes :
   a. Pour tout $(m;n)$, $f(m+n)-f(m)-f(n)$ l'une des valeurs 0 ou 1
   b. $f(2)=0\ ;\ f(3)>0\ \ et\ f(999)=3333$

   Déterminer $f(2014)$
5. L'ensemble des entiers naturels strictement positifs est la réunion de deux sous ensemble disjoints
   $$\{f(1),f(2),\cdots,f(n),\cdots\}\ et\ \{g(1),g(2),\cdots,g(n),\cdots\}$$
   Vérifiant les conditions :
   a. Les suites f et g sont croissantes
   b. $g(n)=f(f(n))+1$ pour tout $n\geq 1$

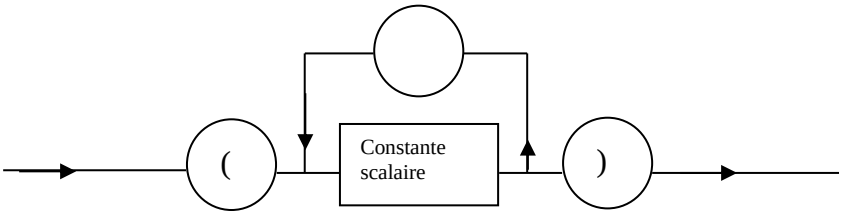
   Déterminer $f(2014)$

4. Ecrire un algorithme qui
   – Lit un entier positif n puis
   – Affiche tous les nombres impairs inférieurs à n
5. Reprendre l'algorithme de tri et le développer cette fois-ci en utilisant des fonctions et des procédures
6. Procédure de saisie du tableau
7. Fonction qui retourne l'indice de la valeur max dans une partie du tableau
8. Procédure qui échange les valeurs de deux cases
9. Procédure qui fait le tri en utilisant la fonction et les 2 procédures ci-dessus

Chapitre 3 Types complémentaires

I. Le type simple construit par énumération

En plus des quatre types simples déjà vu, nous avons la possibilité de construire des types simples propres en énumérant les différentes constantes créées dites constantes scalaires par la syntaxe suivante



Exemple

JOUR={LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE}
SEXE=(FEMININ, MASCULIN)
Si J1 et J2 sont de type JOUR, on peut utiliser toutes les opérations habituelles sur les types simples.

```
J1 :=J2 ;

If (J1<J2) then …… : l'ordre est celui de l'énumération

etc.
```

II. Les types simples construits par intervalle

Dans ce cas l'ensemble des valeurs du type est défini comme partie d'un autre ensemble de type simple et énumérable. Ainsi, l'intervalle est une partie fermée d'un type simple et énumérable de base.

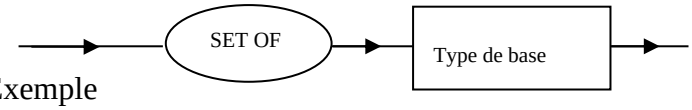
Exemple

```
Type
    INT=-10..15 ;
     JOUROUVRABLE=
LUNDI..VENDREDI ;
 Var
      I : INT ;
      Jour : JOUROUVRABLE
```

I. Le type ensemble

A partir d'un type simple de base on peut définir une structure d'ensemble comme étant un ensemble un ensemble au sens mathématique d'objets de ce type.

Une déclaration (instanciation) d'une variable de type ensemble a donc une structure d'ensemble.



Exemple

```
S : SET OF JOUR
```

Les opérations suivantes sont possibles :

Opération	Syntaxe	Exemple
L'affectation	:=	S :=[lundi, mercredi, jeudi] ; s :=[] (…)
L'intersection	*	[1..56]*[34..76] vaut [34..56]
L'union	+	[11..34]+[1..34] vaut [1 .. 34]
La différence	-	
L'égalité	=	S1=s2
L'inclusion	< <=	S1<s2 s1<=s2
L'appartenance	IN	IF (reponse IN ['o', 'n']) THEN ..

II. Le type string

C'est une variante du type tableau pour nous permettre de travailler sur les chaînes de caractères, c'est donc l'équivalent d'une description en tableau de caractères

Exemple :

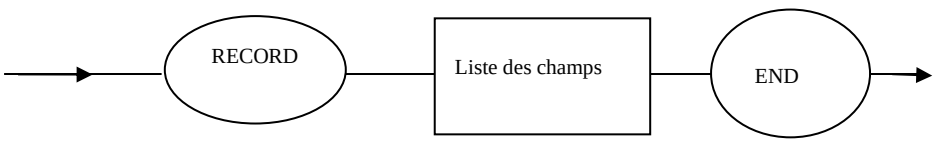
```
A : STRING [30] ;
```

Déclare une chaîne A de 30 caractères commençant à partir de 1.

- A peut être manipulé comme un tableau, mais en outre il peut être traité globalement : toutes les opérations relationnelles deviennent valables. (L'ordre considéré est l'ordre lexicographique induit à partir de l'ordre des codes ASCII des caractères).
- Pour déterminer la longueur de la chaine contenue dans une variable de type string, le principe suivant est adopté : la position 0 de la variable string codé sur un octet contiendra la longueur de la chaîne courante de caractères contenue dans A. La longueur maximale d'une chaîne de caractères est donc de 255.

Plusieurs fonctions et procédure prédéfinies sur les chaînes de caractères existent dans turbo Pascal.

- Le type enregistrement : Le type enregistrement correspond à une collection d'objets de types différents qu'on appelle champs



Exemple 1

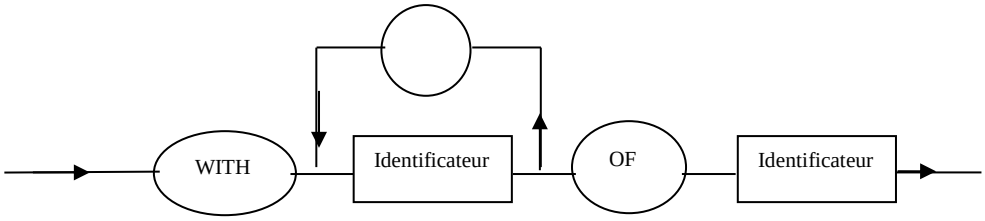
```
Type
Datenaiss=RECORD
                    Jour : 1..31 ;
                    Mois : 1..12 ;
Annee : 1950..200
                  END ;
Employe=RECORD
Nom : STRING[30] ;
sexe: (MASCULIN, FEMININ);
Date : datenais ;
```

Référencer un champ consiste en un parcours de la racine jusqu'au nœud du champ en indiquant les nœuds intermédiaires séparés par des points.

```
d :date ;
```

Les champs jour, mois et année seront référencés par d.jour, d.mois et d.annee.

Remarque : on peut utiliser l'accès avec WITH s'il n'y'a pas d'ambigüité ce qui évite de répéter le nom de l'enregistrement :



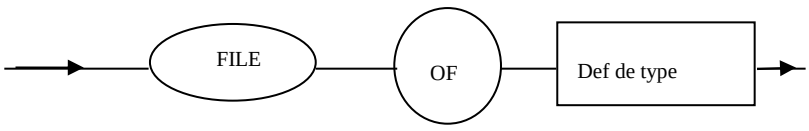
Exemple :

```
WITH d DO
        BEGIN
              Jour :=12 ;
              mois:=1;
              annee:=2O14
         END
```

III. Le type Fichier

Un fichier est un objet de type FILE est définie comme une suite d'objets de même type sauf le type fichier et en nombre indéterminé

Syntaxe



Si les composantes d'une variable de type fichier sont traitées de manière séquentielle on dit que le fichier est à accès séquentiel.

Cependant si nous accédons à n'importe quelle composante par l'intermédiaire de son rang le fichier est à accès direct.

En Turbo Pascal, à la création du fichier un index (file pointer) automatique (transparent au programmeur) prend la valeur zéro (rang de la première composante). A chaque lecture ou écriture dans la variable de type fichier cet index est incrémenté.

Exemple

```
Femploye=FILE OF employé
Fich : femploye ;
```

Opérations fondamentales sur les fichiers

Pour pouvoir travailler avec les fichiers il faut obligatoirement passer par les trois phases suivantes :

- Etablir la liaison entre la variable interne de type fichier et son nom externe tel qu'il est connu par le système d'exploitation.
- Ouvrir ce fichier soit en lecture soit en écriture
- Après les traitements sur ce fichier réaliser une fermeture logique.

Des procédures prédéfinies ont été écrites et permettent de réaliser ces trois objectifs.

Procédures	Syntaxe	
Assignation	ASSIGN (fich, ''c: employe.dat)	fich nom interne c: employe.dat nom externe
Ouverture en lecture	RESET(fich)	Fich : identificateur d'1 variable de type fichier. Le file pointe vaut 0 (il pointe sur la 1ère composante)
Ouverture en écriture	REWRITE(fich)	
Fermeture	CLOSE(fich)	

Opérations de traitements de fichier

Procédure ou fonction	Syntaxe	
Lecture d'une composante	READ (fich, ident)	Ident : type composante de fich
Ecriture d'une composante	WRITE (fich, ident)	Ecrit dans fich à l'emplacement pointeur par le file pointer puis s'incrémente

Accès direct	SEEK (fich, n)	Fait pointer le file pointer au rang n
Taille du fichier	FILISIZE (fich)	
Test de fin de fichier	EOF (fich)	true si fin de fichier pointé false sinon

Chapitre 4 Introduction à la récursivité et à la programmation récursive

I. Notion de récursivité

Bien que tous les chapitres précédents soient axés principalement sur les techniques de programmation (dite itératives), nous avons été confrontés à plusieurs reprises à la récursivité surtout dans les tentatives de décrire formellement la syntaxe d'un langage
Dans le cadre de ce chapitre, charnière entre cette première partie et la deuxième relative aux techniques d'analyse des algorithmes et des structures de données.
Les objectifs sont :

- Illustrer la récursivité en apportant des précisions sur sa définition, sa représentation algorithmique et son mode de fonctionnement
- Donner des indications sur son utilisation dans la conception des algorithmes dans le cadre de ce qu'il est souvent convenu d'appeler la programmation récursive.
- Le chapitre suivant nous donnera l'occasion de porter quelques réflexions sur d'une part les possibilités de passer d'une version récursive à une version itérative (dérécursivation) et d'autre part sur la portée des deux techniques de programmation.

1. Définition générale

Une notion est récursive si elle se contient elle-même en partie ou si elle est partiellement définie à partir d'elle-même. Elle est dite auto-référenciée.
Pour retourner l'écran : ALT+ ↑
Quelques commentaires sur cette définition

1. La récursivité est un concept que l'on rencontre fréquemment :
    a. Dans la vie courante (des histoires à l'intérieur d'histoires, des représentations à l'intérieur de représentation, des objets à l'intérieur d'objets, …)
    b. En mathématiques : les suites récurrentes, méthodes itératives ; etc.
    c. En théorie des langages etc.
2. Ce processus d'auto-référencement doit être fini ! en effet il ne s'agit pas d'une référence à l'exacte elle-même ce qui serait circulaire mais à une copie de la version précédente. La terminaison est assurée par le fait que cette suite de versions doit converger vers un état connu
3. Une fois cet état atteint, il faut retourner à la version précédente afin de la terminer pour reprendre la version encore précédente, la terminer etc.

Ce mécanisme de fonctionnement impose de garder la trace de toute la chaîne des versions et dans l'ordre de leur création puisqu'on doit les reprendre dans le sens contraire (ordre LIFO : last in first out)

Exemple de notions récursives :

1. Calcul de la somme des premiers termes d'une suite

$$S = \sum_{k=0}^{n} u_k$$

Règle i/ $S(0) = u_0$
Règle ii/ $S(n) = u_n + S(n-1)$

2. Factorielle $n!$
    a. Règle i/ $0! = 1$
    b. Règle ii/ $n! = n.(n-1)!$

- PGCD de deux nombre entiers u et v
    a. $pgcd(u, 0) = u$
    b. $pgcd(u, v) = pgcd(u, u\ modulo\ v)$
- Les informations relevées sur chaque individu sont :
    a. Son nom
    b. Sa date de naissance
    c. Et celle de son père (qui est un individu) etc.

L'analyse de ces quelques exemples fait ressortir les remarques suivantes :

1- Pour assurer une terminaison à l'application des règles de définition d'une notion récursive :
    a. Une première condition nécessaire est que la partie faisant appel à la redéfinition récursive soit conditionnelle et doit être fausse au moins une fois. De ce fait, la définition d'une notion récursive doit prévoir une sortie qui intervient quand la condition d'application de la partie récursive devient fausse.

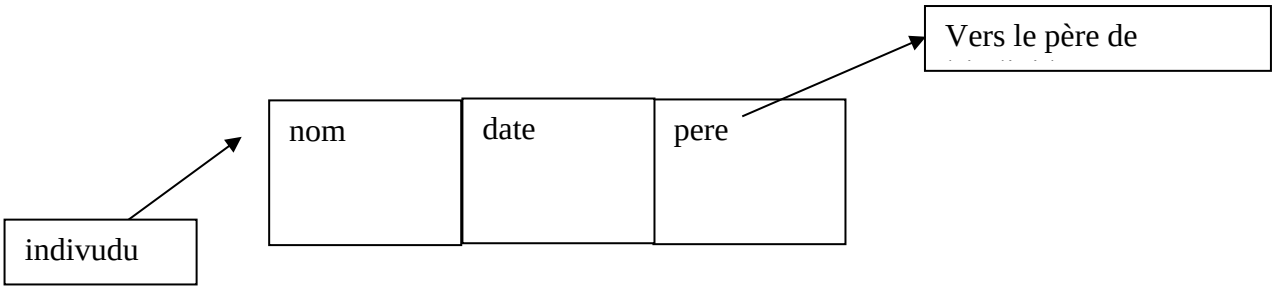
2- La récursivité peut se présenter de deux manières :
    a. Elle est dite directe si la règle de redéfinition récursive utilise directement la notion sinon elle est indirecte, c'est-à-dire qu'elle passe par une ou plusieurs autres règles avant de faire appel à la notion en définition

3- La récursivité s'illustre en algorithmique aussi bien au niveau des actions qu'au niveau des objets. On parlera
    a. De récursivité des actions si l'énoncé du problème fait appel directement ou indirectement à lui-même et
    b. De récursivité des structures de données si un objet est défini comme une collection d'objets dont certains ont la même structure que l'objet en définition

    La récursivité des structures de données est un concept extrêmement puissant pour représenter les modèles fondamentaux d'organisation des données. Elle sera utilisée pratiquement dans toute la deuxième partie de ce cours, néanmoins dans le chapitre 2 nous avons vu un moyen classique de représentation algorithmique en passant par les pointeurs. En effet, si l'objet a un type T on ne peut décrire sa composante par T ce qui provoquerait une définition circulaire. On passe alors par les pointeurs qui font référence aux objets de type T pour définir le type de la composante et l'arrêt de la chaîne des définitions se faisant par une constante spéciale pointeur indiquant le fait de ne pointer sur rien (nil). L'exemple 4 pourrait se décrire par :

```
PPERS=↑ PERSONNE ;
PERSONNE= RECORD
                Nom          : STRING[30]
                Date_naiss : RECORD jour : 1 ..31     MOIS/ 1..12 ;
année : 2000..2013 END ;
                Père         : PPERS
                END
```

Et une instanciation par individu : PPERS ; donnerait :



II. Récursivité des actions

1. Réalisation algorithmique

Pour exprimer un énoncé récursif, il nous faut un moyen de le nommer ce que nous avons par l'appel de procédure ou de fonction. Ainsi, une procédure P (ou fonction F) contient un ou plusieurs appels explicite à elle-même on dit qu'elle est directement récursive autrement si P (ou F) contient un appel à P (ou F) nous dirons que P (ou F) est indirectement récursive ou encore que P (ou F) et Q (ou G) sont des récursivités croisées.

Récursivité directe

```
PROCEDURE P(D : TD) ;

{
   P(B(D)) ;
}
```

```
FONCTION Q( D : TD) : RD ;

{
   F(B(D)) ;
}
```

Récursivité directe

```
PROCEDURE P(D : TD) ;

{
   Q(B(D));
}
```

```
FONCTION Q( D : TD) : RD ;

{
   P(C(D)) ;
}
```

Nous parlerons aussi de procédure (ou fonction) récursive primitive si les paramètres ne s'expriment pas par appel de la procédure (ou de la fonction) autrement elle est dite générale.
Exemple de fonction générale (non primitive) :

```
FONCTION ack (m,n : ENTIER) : ENTIER ;

{ SI  (m=0)   ALORS      ack← 1
              SINON
                    SI (n=0)   ALORS ack←ack(m-1,1)
                     SINON ack← ack (m-1, ack(m,n-1)
}
```

Dans tout ce qui suit, nous parlerons généralement des procédures, l'adaptation aux fonctions sera laissée aux soins du lecteur, sauf dans le cas contraire ou une différenciation s'impose.

2. Visibilité des objets d'une procédure (ou fonction)

   i. Récursivité et mécanisme de fonctionnement

A chaque procédure est associée un ensemble d'objets accessibles qui sont :

- Ses objets locaux
- Ses objets paramètres
- Ses objets plus globaux

L'ensemble des objets propres (locaux et paramètres passés par valeur) a une certaine existence (durée de vie) pendant l'exécution de la procédure.
Nous dirons qu'un appel de la procédure entraine la création d'un environnement formé d'un exemplaire particulier de cet ensemble d'objets qui ne cesse d'exister que quand cet appel se termine (cette existence continuant au-delà pour les objets plus globaux ou les objets paramètres passés par adresse) en d'autres termes, Si une procédure est appelée récursivement, cela signifie qu'un nouvel environnement va naître sous celui du précédent (celui qui l'a appelé). Il va donc exister plusieurs environnements différents pour la même procédure et par conséquent plusieurs incarnations différentes pour un même objet dans chaque environnement sans qu'il n'y ait aucun rapport ni aucune communication entre ces différents environnements et la désignation d'un objet local à une procédure n'est pas ambiguë puisqu'elle se réfère toujours au dernier environnement créé et encore en cours d'existence. Quand un appel récursif se termine, les objets de l'environnement correspondant à l'appel précédent redeviennent accessibles parce qu'ils n'ont pas cessé d'exister entre temps

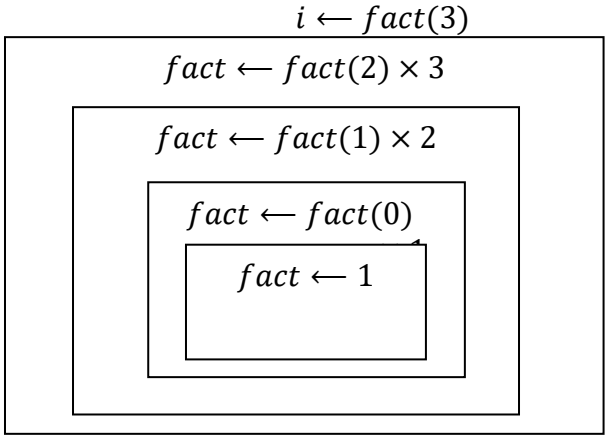
3. Exemple de procédures récursives

   i. Calcul de n !

```
FONCTION fact (n : ENTIER) : ENTIER

{ SI  (n=0)  ALORS  fact⟵1
              SINON   fact⟵ fact(n − 1) × n
}
```

Simulation pour n=3



4. Programmation récursive

Les exemples traités permettent de mettre en évidence les idées fortes dans la conception en programmation récursive.
La première est de transformer (de manière équivalente) l'énoncé du problème en un autre dont les spécifications sont récursives

Exemple 1

On considère deux chaînes de caractères w1 et w2 et on désire savoir si w1 est anagramme de w2, c'est-à-dire que w1 est égale à w2 à une permutation près de caractères

Eléments d'analyse

1- Sur quelles données faut-il porter la récursivité ?
2- Toute définition récursive doit avoir un état connu (point de sortie) qui permet de ne plus appliquer la règle récursive