EPSIC

ALGORITHMES

1.1. Notion d'algorithme.

a-*Définition*.

Certains voient, à tort, dans l'ordinateur une machine pensante et intelligente, capable de résoudre bien des problèmes. En fait, celui-ci ne serait capable de rien si quelqu'un (le programmeur en l'occurrence) ne lui avait fourni la liste des actions à exécuter. Cette description doit être faite de manière non ambiguë car il ne faut pas s'attendre à la moindre interprétation des ordres fournis. Ils seront exécutés de manière purement mécanique.

De plus, les opérations élémentaires que peut exécuter un ordinateur sont en nombre restreint et doivent être communiquées de façon précise dans un langage qu'il comprendra. Le problème principal de l'utilisateur est donc de lui décrire la suite des actions élémentaires permettant d'obtenir, à partir des données fournies, les résultats escomptés. Cette description doit être précise, envisager le moindre détail et prévoir les diverses possibilités de données.

Cette marche à suivre porte le nom d'algorithme dont l'*Encyclopaedia Universalis* donne la définition suivante:

" 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. "

Le mot algorithme provient du nom d'un célèbre mathématicien arabe de la première moitié du IXe siècle: Muhammad ibn Musa al Khwarizmi.

Le rôle de l'algorithme est fondamental. En effet, sans algorithme, il n'y aurait pas de programme (qui n'est jamais que sa traduction dans un langage compréhensible par l'ordinateur). De plus, les algorithmes sont fondamentaux en un autre sens: ils sont indépendants à la fois de l'ordinateur qui les exécute, des langages dans lequel ils sont énoncés et traduits.

b-*Exemple*.

Calcul de l'intérêt et de la valeur acquise par une somme placée pendant un an à intérêt simple.

L'énoncé du problème indique

les données fournies: deux nombres représentant les valeurs de la somme placée et du taux d'intérêt

les résultats désirés: deux nombres représentant l'intérêt fourni par la somme placée ainsi que la valeur obtenue après placement d'un an.

Il nous faut maintenant décrire les différentes étapes permettant de passer des donnés aux résultats. Nos connaissances générales nous permettent d'exprimer cette règle:

"Pour obtenir l'intérêt fourni par la somme, il suffit de multiplier la somme par le taux d'intérêt divisé par cent; la valeur acquise s'obtient en additionnant ce dernier montant et la somme initiale."

c-*Méthodologie*.

Dans cet exemple simple apparaissent les trois étapes qui caractérisent la résolution d'un problème sur ordinateur:

comprendre la nature du problème posé et préciser les **données** fournies ("entrées" ou "**input**" en anglais)

préciser les résultats que l'on désire obtenir ("sorties" ou "**output**" en anglais)

déterminer le processus de transformation des données en résultats.

Ces trois étapes ne sont pas indépendantes et leur ordre peut être modifié.

Si les résultats fournis par l'ordinateur ne sont pas corrects, c'est qu'une erreur s'est glissée soit dans l'analyse du problème, soit dans la mise au point de l'algorithme, soit dans sa traduction en langage de programmation car l'ordinateur ne fait qu'exécuter scrupuleusement les opérations demandées.

1.2. Formalisation de l'algorithme.

a- *En français .*

L'exemple décrit ci-dessus deviendrait:

(1) prendre connaissance de la somme initiale et du taux d'intérêt

(2) multiplier la somme par le taux; diviser ce produit par 100; le quotient obtenu est l'intérêt de la somme

(3) additionner ce montant et la somme initiale; cette somme est la valeur acquise

(4) afficher les valeurs de l'intérêt et de la valeur acquise.

Il est évident, même sur cet exemple simple, qu'une telle formalisation risque de produire un texte long, difficile à comprendre et ne mettant pas clairement en évidence les différentes étapes du traitement.

b-*Langage de description.*

Dans un langage de description, les actions sont généralement décrites par un symbole ou un verbe à l'infinitif choisi pour éviter les confusions. Ce langage est appelé soit pseudo code soit langage de description d'algorithme (LDA).

Notre exemple devient:

écrire " Introduisez la somme initiale (en francs): "

lire somme_initiale

écrire " Introduisez le taux d'intérêt (ex: 3 pour 3%): "

lire taux

intérêt <-- somme_initiale * taux / 100

valeur_acquise <-- somme_initiale + intérêt

écrire " L'intérêt fourni est de " , intérêt , "francs "

écrire " La somme après un an sera de " , valeur_acquise , "francs "

Nous pouvons remarquer deux verbes particuliers:

lire qui correspond à la saisie, à l'introduction des données;

écrire qui exécute l'affichage à l'écran ou l'impression des résultats.

Ces verbes sont soulignés pour indiquer qu'ils ont un sens particulier, qu'il est interdit de les utiliser dans un autre sens et qu'il seront traduits pour être rendus compréhensibles par la machine.

Les valeurs manipulées dans cet algorithme sont des **constantes** (100) et des **variables** (somme_initiale, taux, intérêt, valeur_acquise). Il est pratique de choisir le nom des variables de manière à rappeler la signification de la valeur qu'elles représentent. Ce nom est souvent appelé identificateur de la variable. Les variables jouent le rôle de "**tiroirs**" dans lesquels on place une valeur durant l'exécution de l'algorithme. Ainsi,

lire somme_initiale

signifie que l'on introduit dans le tiroir baptisé *somme_initiale* la valeur numérique entrée au clavier lors de l'exécution du programme.

Le contenu d'un de ces tiroirs peut être modifié en y plaçant le résultat d'un calcul. Cette instruction porte le nom d'**assignation** ou **affectation** et se représente par une flèche (<--).

Ainsi,

intérêt <-- somme_initiale * taux /100

signifie que l'on place dans le tiroir *intérêt* le résultat de l'opération figurant à droite de la flèche. Cette instruction se lit: assigner à la variable *intérêt* la valeur de l'expression de droite.

Les expressions symbolisant les calculs à effectuer sont représentées par des formules algébriques faisant intervenir les noms des variables, des symboles mathématiques ("+" pour l'addition, "-" pour la soustraction, "*" pour la multiplication, "/" pour la division, ...) et des constantes numériques.

La description d'une action et des objets qui y participent porte le nom d'**instruction**. L'ordre dans lequel les différentes opérations seront écrites indique l'ordre dans lequel elles seront exécutées: de haut en bas et de droite à gauche. Il s'agit d'une **exécution séquentielle**.

1.3. Qualités d'un algorithme, d'un programme.

Tout programme fourni à l'ordinateur n'est que la traduction dans un langage de programmation d'un algorithme mis au point pour résoudre un problème donné. Pour obtenir un bon programme, il faut partir d'un bon algorithme. Il doit, entre autres, posséder les **qualités** suivantes:

être **clair**, facile à comprendre par tous ceux qui le lisent (**structure** et **documentation**)

présenter la plus **grande généralité** possible pour répondre au plus grand nombre de cas possibles

être d'une **utilisation aisée** même par ceux qui ne l'ont pas écrit et ce grâce aux messages apparaissant à l'écran qui indiqueront quelles sont les données à fournir et sous quelle forme elles doivent être introduites ainsi que les différentes actions attendues de la part de l'utilisateur

être conçu de manière à limiter le nombre d'opérations à effectuer et la place occupée en mémoire.

Une des meilleures façons de rendre un algorithme clair et compréhensible est d'utiliser une programmation structurée n'utilisant qu'un petit nombre de structures indépendantes du langage de programmation utilisé. Une technique d'élaboration d'un bon algorithme est appelée **méthode descendante** (**top down**). Elle consiste à considérer un problème dans son ensemble, à préciser les données fournies et les résultats à obtenir puis à **décomposer** le problème en plusieurs sous problèmes plus simples qui seront traités séparément et éventuellement décomposés eux-mêmes de manière plus fine.

Exemple: imaginons un robot domestique à qui nous devons fournir un algorithme lui permettant de préparer une tasse de café soluble. Une première version de l'algorithme pourrait être:

(1) faire bouillir de l'eau

(2) mettre le café dans la tasse

(3) ajouter l'eau dans la tasse

Les étapes de cet algorithme ne sont probablement pas assez détaillées pour que le robot puisse les interpréter. Chaque étape doit donc être affinée en une suite d'étapes plus élémentaires, chacune étant spécifiée d'une manière plus détaillée que dans la première version. Ainsi l'étape

(1) faire bouillir l'eau peut être affinée en

(1.1) remplir la bouilloire d'eau

(1.2) brancher la bouilloire sur le secteur

(1.3) attendre l'ébullition

(1.4) débrancher la bouilloire

De même,

(2) mettre le café dans la tasse pourrait être affiné en

(2.1) ouvrir le pot à café

(2.2) prendre une cuiller à café

(2.3) plonger la cuiller dans le pot

(2.4) verser le contenu de la cuiller dans la tasse

(2.5) fermer le pot à café

et (3) ajouter de l'eau dans la tasse pourrait être affinée en

(3.1) verser de l'eau dans la tasse jusqu'à ce que celle-ci soit pleine

Certaines étapes étant encore trop complexes et sans doute incompréhensibles pour notre robot, il faut les affiner davantage. Ainsi l'étape

(1.1) remplir la bouilloire d'eau

peut nécessiter les affinements suivants:

(1.1.1) mettre la bouilloire sous le robinet

(1.1.2) ouvrir le robinet

(1.1.3) attendre que la bouilloire soit pleine

(1.1.4) fermer le robinet

Quand il procède à des affinements des différentes étapes, le concepteur d'un algorithme doit naturellement savoir où s'arrêter. Autrement dit, il doit savoir quand une étape constitue une primitive adéquate au point de ne pas avoir besoin d'affinement supplémentaire. Cela signifie évidemment qu'il doit connaître quelle sorte d'étape le processeur peut interpréter. Par exemple, le concepteur de l'algorithme précédent doit savoir que le robot peut interpréter "brancher la bouilloire" ce qui de ce fait n'exige pas d'affinement, mais qu'en revanche, il ne peut pas interpréter "remplir la bouilloire" et que dès lors un affinement devient nécessaire.

2. LA SEQUENCE.

2.1. Langage de description.

a- *Instructions*

Dans les algorithmes décrivant des calculs sur les quantités numériques, seront utilisées essentiellement les instructions que nous avons déjà étudiées.

1° les instructions de lecture (d'entrée) notées:

lire *variables*

indiquant la saisie des données

exemples:

lire somme_initiale
lire taux

2° les instructions d'écriture (de sortie) de la forme:

exemple:

écrire *expression*

indiquant l'affichage d'un message et/ou du contenu d'une variable (ou du résultat d'un calcul)

exemples:

écrire " Introduisez la somme initiale (en francs): "

écrire " L'intérêt fourni est de " , intérêt

écrire intérêt

écrire a, b, (a+b)/2

3° les instructions d'assignation (d'affectation) représentées par

variable <-- *expression*

exemple:

intérêt <-- somme-initiale * taux / 100
a <-- 0
i <-- i + 1

Les expressions sont des formules mathématiques symbolisant des opérations sur des variables et/ou des constantes numériques.

Les **variables** y sont représentées par un **identificateur** (un nom) comme en algèbre et les constantes sont des nombres écrits en chiffres. Nous utiliserons la convention anglo-saxonne utilisée par la plupart des ordinateurs (et des calculatrices) et qui consiste à employer le **point** ("**.**") pour séparer la partie entière de la partie décimale d'un nombre.

Les opérations sur des nombres sont représentées par +, -, * (pour ne pas confondre le symbole de multiplication avec la lettre "x" ou avec le point décimal), /.

D'autres fonctions mathématiques usuelles sont couramment utilisées: ln x, sin x, arctg x, [x] (signifie prendre la partie entière de x), a **mod** b (fournit le reste de la division de a par b), x^y, $\log_a x$, ...

Mais l'ordinateur peut également manipuler des variables contenant des **chaînes de caractères alphanumériques** (alphabétiques et/ou numériques et/ou spéciaux) pour les modifier, en extraire des sous-chaînes, ... Ces chaînes de caractères sont placées entre guillemets pour les distinguer des noms de variables. L'opération de **concaténation** (juxtaposition de 2 chaînes pour en former une nouvelle) est symbolisée par **//** séparant les 2 chaînes originelles. La fonction qui permet d'**extraire** une sous-chaîne est représentée par le nom de la variable avec en indice les positions des lettres à extraire. Ainsi la sous-chaîne formée des caractères occupant les positions 2, 3, 4 dans la variable *prénom* sera symbolisée par: **$prénom_{2<--4}$**

Enfin, la fonction qui fournit la **longueur** (le nombre de caractères) de la chaîne contenue dans la variable *prénom* est symbolisée par **|prénom|**

b- *Exemples*

1- Exprimer un nombre de secondes sous forme d'heures, minutes, secondes. La seule donnée est le nombre total de secondes que nous appellerons nsec; les résultats consistent en 3 nombres h, m, s.

```
écrire " Introduisez le nombre de secondes"
lire nsec
s <-- nsec mod 60
m <-- (nsec \ 60) mod 60
h <-- nsec \ 3600
écrire nsec, "valent: ", h, "heure(s) ", m, "minute(s) et", s, "seconde(s)"
```

2- Transformer un prénom et un nom en une chaîne contenant l'initiale du prénom séparée du nom par un point

```
écrire "Quel est votre prénom?"
lire prénom
écrire "et votre nom?"
lire nom
pr <-- prénom1
lpr <-- |prénom|
ident <-- pr // "." // nom
écrire "Votre prénom de", lpr, "lettres a été abrégé
et votre identification est : ", ident
```

c-*Déclaratives*

Il est aussi nécessaire de préciser ce que les variables utilisées contiendront comme type de données. Il peut s'agir de nombres entiers, de nombres réels, de chaînes de caractères, ... Il faut faire précéder la description de l'algorithme par une partie dite **déclarative** où l'on regroupe les caractéristiques des variables manipulées.

La partie déclarative est placée en tête de l'algorithme et regroupe une ou plusieurs indications de la forme:

entier *variables*
ou
réel *variables*

L'algorithme complété de l'exemple 1 devient:

```
entier nsec, h, m, s
écrire "Introduisez le nombre de secondes:"
lire nsec
s <-- nsec mod 60
m <-- (nsec \ 60) mod 60
h <-- nsec \ 3600
écrire nsec, "valent: ", h, "heure(s)", m, "minute(s) et", s, "seconde(s)"
```

Et l'algorithme de l'exemple 2:

```
entier lpr
chaîne prénom, nom, ident
écrire "Quel est votre prénom?"
lire prénom
écrire "et votre nom?"
lire nom
pr <-- prénom1
lpr <-- |prénom|
ident <-- pr // "." // nom
écrire "Votre prénom de",lpr,"lettres a été abrégé
et votre identification est : ",ident
```

2.2. Exercices

Résumé:
Pour l'échange de données entre le programme et l'utilisateur (ou le disque du PC), 2 mots sont utilisés: (1) **lire** pour recevoir de l'info du monde extérieur: lire N où N est le nom de la variable qui va recevoir l'information fournie par l'utilisateur lire N sur *Fichier* où N est le nom de la variable qui va recevoir l'information récupérée dans le fichier *Fichier* (2) **écrire** pour fournir de l'info au monde extérieur: écrire "Bonjour tout le monde." où la partie entre guillemet est le message à afficher à l'écran écrire N où N est le nom de la variable qui contient l'information à écrire

écrire N sur *Fichier* où N est le nom de la variable qui contient l'information à écrire sur le fichier *Fichier*

Lorsque le programme *travaille dans sa tête*, on utilise l'assignation <-- pour symboliser la mémorisation dans une variable.
N <-- N+2
x <-- 2*3+5/2
St <-- "Hello"

2.3. Traduction en Pascal.

Généralités

Examinons les programmes complets correspondant aux algorithmes décrits en LDA.

```
entier nsec, h, m, s
écrire "Introduisez le nombre de secondes:"
lire nsec
s <-- nsec mod 60
m <-- (nsec \ 60) mod 60
h <-- nsec \ 3600
écrire nsec, "valent: ", h, "heures",m, "minutes et", s, "secondes"
```

deviendra en Pascal:

```
PROGRAM secondes;
VAR nsec, h, m, s : INTEGER;
BEGIN
WRITE('Introduisez le nombre de secondes: ');
READLN (nsec) ;
s:=nsec MOD 60 ;
m:=nsec DIV 60 MOD 60 ;
h:=nsec DIV 3600 ;
WRITELN (nsec, 'valent: ', h, 'heures', m, 'minutes et', s, 'secondes')
END.
```

Les règles de base.

Dans ces exemples, nous retrouvons déjà dix règles de base:

1° Un programme Pascal se compose de trois parties:

un en-tête, caractérisé par le mot **PROGRAM**

une section déclarative introduite ici par le mot **VAR**

une section instruction ou corps du programme, délimitée par les mots **BEGIN** et **END.**

Attention: Le programme se termine par un point.

2° L'en tête (facultative) sert à donner un nom au programme selon la forme:

PROGRAM identificateur;

3° Un identificateur en Pascal doit débuter par une lettre suivie par un nombre quelconque de **lettres**, **chiffres** ou de "_" (caractère souligné). Les identificateurs ne peuvent contenir d'espacement (caractère "blanc") ou de caractères tels que %, ?, *, ., - ,... mais peuvent être aussi longs que l'on veut.

4° Les variables **doivent** faire l'objet d'une déclaration de **type** de la forme:

VAR *liste des variables* : type;

5° Des points-virgules sont obligatoires pour séparer les trois parties et pour séparer les instructions

6° Les instructions de lecture et d'écriture se traduisent respectivement par **READLN** et **WRITE** (ou **WRITELN**) suivis d'une liste de variables ou d'expressions placées entre parenthèses et séparées par des virgules.

L'ajout de **LN** après **WRITE** (**WRITELN**) force le passage à la ligne lors de l'affichage suivant à l'écran.

7° L'assignation se représente par "**:=**"

8° Les opérateurs arithmétiques sont identiques à ceux du langage de description d'algorithme (LDA). Toutefois, nsec\3600 est traduit par nsec **DIV** 3600. En effet, outre les quatres opérations + - * / , Pascal utilise deux opérateurs supplémentaires:

DIV fournissant la partie entière du quotient de deux nombres entiers

MOD fournissant le reste de la division de deux nombres entiers

Ainsi, **13 / 5** fournit la valeur 2.6

13 **DIV** 5 fournit 2

et 13 **MOD** 5 fournit 3.

9° Les mots **PROGRAM**, **VAR**, **BEGIN**, **END**, **DIV**, **MOD**, ... ont un sens précis dans le langage: ce sont des **mots réservés** qui ne peuvent être choisis comme identificateurs par le programmeur.

Dans un programme en LDA, les mots réservés sont soulignés.

Un certain nombre de mots tels que **INTEGER**, **READLN**, **WRITE**, ... ont une signification prédéfinie. Pour éviter toute erreur, on s'abstiendra de les choisir comme identificateur.

10° Les mots du langage et les identificateurs doivent être séparés les uns des autres par un ou plusieurs blancs.

Lecture.

Les lectures sont symbolisées par le mot **READLN**. C'est la procédure **READLN** qui transfère les nombres ou chaînes de caractères du clavier vers la mémoire centrale. Ceux-ci doivent respecter la forme des constantes de Pascal et doivent être séparés par un blanc au moins.

Le type de la constante donnée et celui de la variable d'accueil doivent correspondre selon la règle des assignations.

Pascal admet aussi la procédure **READ** (qui a le même effet que **READLN** sans passage à la ligne pour le prochain affichage) mais il s'est révélé à l'usage, que celle-ci était parfois source de problème et il est préférable de l'éviter.

Ecriture.

Les écritures se font de façon semblable aux lectures, à l'aide de la procédure **WRITE**. Les valeurs à afficher apparaîtront sur l'écran à la queue-leu-leu sur une seule ligne, parfois sans espacement entre elles.

Pour améliorer la lisibilité, on peut:

utiliser la procédure **WRITELN** qui force le passage à la ligne suivante pour le prochain affichage

faire usage des **formats d'édition** qui précisent le nombre de caractères à utiliser pour afficher chacun des résultats:

WRITE(*valeur_entière* : n) affiche la valeur entière sur n positions (insertion d'espacement à gauche du nombre si il y a trop peu de chiffres et ajustement automatique, si n est insuffisant)

WRITE(*valeur_réelle*) affiche le nombre en notation scientifique (x.xxxxxE+x précédé d'un espacement)

WRITE(*valeur_réelle* : n) affiche le nombre en notation scientifique sur n positions

WRITE(*valeur_réelle* : n1 : n2) affiche le nombre sur n1 positions avec n2 décimales (avec ajustement).

WRITE(*chaîne* : n) affiche la chaîne de caractère sur n positions (insertion d'espacement à gauche de la chaîne si il y a trop peu de caractères et ajustement automatique, si n est insuffisant)

Exemples:

Si la variable entière x contient 12345, (**^ symbolise l'espacement**)

WRITE(x) affiche 12345

WRITE(x:8) affiche ^^^12345

WRITE(x:2) affiche 12345

Si la variable réelle x contient 123.4567, (**^ symbolise l'espacement**)

WRITE(x) affiche ^1.23456E+2

WRITE(x:7) affiche ^1.2E+2

WRITE(x:8:2) affiche ^^123.46

WRITE(x:2) affiche 1.2E+2

Si la variable du type chaîne x contient 'AZERTY', (**^ symbolise l'espacement**)

WRITE(x) affiche AZERTY

WRITE(x:8) affiche ^^AZERTY

WRITE (x:3) affiche AZERTY

Manipulation de nombres.

Si la mathématique distingue plusieurs types de nombres directement manipulables par les langages informatiques, Pascal n'en reconnaît que deux: les types **entier** et **réel**.

Le type entier.

En LDA, nous placerons dans la déclaration des variables une ligne telle que:

entier age, note_de_français

Mais Pascal a subdivisé les entiers en 5 types pour mieux adapter le type aux valeurs que peuvent prendre une variable, et ce pour optimiser l'occupation de la mémoire.

Type	Valeurs autorisées	Occupation en mémoire
SHORTINT	de **-128** à **+127**	**1 octet**
BYTE	de **0** à **255**	**1 octet**
INTEGER	de **-32768** à **+32767**	**2 octets**
WORD	de **0** à **65535**	**2 octets**
LONGINT	de **-2147483648** à **2147483647**	**4 octets**

Le type réel.

En LDA, nous placerons dans la déclaration des variables une ligne telle que:

réel taux_de_TVA, note_moyenne

Mais Pascal a subdivisé les réels en 4 types pour mieux adapter le type aux valeurs que peuvent prendre une variable, et ce pour optimiser l'occupation de la mémoire.

Type	Valeurs autorisées	Nombre de chiffres significatifs	Occupation en mémoire
SINGLE	de -10^{38} à $9{,}8\ 10^{38}$	**7 chiffres**	**4 octets**
REAL	de -10^{38} à $9{,}8\ 10^{38}$	**11 chiffres**	**6 octets**
DOUBLE	de -10^{308} à 10^{308}	**15 chiffres**	**8 octets**
EXTENDED	de -10^{4932} à $9{,}8\ 10^{4932}$	**19 chiffres**	**10 octets**

Il existe aussi le type **COMP** stocké sur 8 octets et offrant une gamme de nombres allant de $-9{,}8\ 10^{18}$ à $9{,}8\ 10^{18}$ mais ne conservant que la partie entière du nombre.

Assignation.

Dans une assignation, le type de l'expression doit correspondre au type de variable de destination. Cette règle admet une seule exception: une variable réelle peut recevoir une valeur entière.

Evaluation des expressions arithmétiques.

Pascal respecte la même convention de priorité que l'arithmétique: les multiplications et les divisions (opérateurs *** / DIV** et **MOD**) sont effectuées en premier lieu, puis les additions et les soustractions (opérateurs **+** et **-**); lorsqu'une expression contient plusieurs opérateurs de même priorité, les opérations sont effectuées de gauche à droite. Pour modifier cet ordre, il suffit d'introduire des parenthèses.

Exemple: **WRITELN**(1/2*3) n'affichera pas la valeur de 1/6 mais bien de 3/2 car la division se fera avant la multiplication.

Type des opérandes et du résultat.

Les opérateurs +, - et * peuvent agir sur des opérandes réels ou entiers et le résultat est réel sauf si les deux opérandes sont entiers. L'opérateur / peut agir sur des entiers et des réels mais le résultat est toujours réel. Les opérateurs **DIV** et **MOD** ne peuvent être utilisés qu'avec des opérandes entiers et fournissent un résultat entier.

Constantes numériques.

Le type d'une variable est défini dans la partie déclarative du programme. C'est la forme d'écriture qui détermine le type d'une constante. Ainsi 50 est une constante entière, tandis que 3.1416 et 50.0 sont des constantes réelles car elles contiennent une partie fractionnaire.

Les constantes entières ne peuvent contenir que des chiffres décimaux (0 à 9) précédés éventuellement d'un signe + ou -.

Les constantes réelles doivent contenir en plus:

soit par une partie fractionnaire d'au moins un chiffre, séparée de la partie entière par un point:

+1.2 -56 0.01 0.0

soit une partie exposant sous forme d'une constante entière précédée par un **E** indiquant la puissance de 10 par laquelle il faut multiplier la valeur qui précède la lettre **E** :

1E4 vaut 1 * 10^4 = 10000.0 6E-2 vaut 6 * 10^{-2} = 0.06

soit les deux:

3.14E+4 vaut 3.14 * 10^4 = 31400.0

Dans ce cas et si il n'y a qu'un seul chiffre non nul dans la partie entière, on parle de **notation scientifique**.

Fonctions mathématiques.

Notation mathématique	Fonction Pascal	Type de x	Type du résultat	Signification
|x|	ABS(x)	Entier ou réel	Type de x	Valeur absolue de x
x^2	SQR(x)	Entier ou réel	Type de x	Carré de x
$x^{1/2}$	SQRT(x)	Entier ou réel	Réel	Racine carré de x
sin x	SIN(x)	Entier ou réel	Réel	sin de x (x en radians)
cos x	COS(x)	Entier ou réel	Réel	cos de x (x en radians)
arctg x	ARCTAN(x)	Entier ou réel	Réel	Angle (en radians) dont la tangente vaut x
e^x	EXP(x)	Réel	Réel	Exponentielle de x
ln x	LN(x)	Réel	Réel	Logarithme népérien de x
[x]	TRUNC(x)	Réel	Entier	Partie entière de x
[x]	INT(x)	Réel	Réel	Partie entière de x
arrondi de x	ROUND(x)	Réel	Entier	Entier le plus proche de x
décimal de x	FRAC(x)	Réel	Réel	Partie décimale de x

On notera l'absence des fonctions tg x et x^y qui se traduiront, en employant les formules de mathématique adéquates, respectivement par SIN(x)/COS(x) et EXP(y*LN(x)).

Manipulation de chaînes de caractères.

Nous avons déjà vu précédemment un exemple de ce type de problème et il est évidemment nécessaire dans la vie professionnelle de manipuler des données autres que numériques (noms de clients, d'articles vendus, ...). Celles-ci sont alors dites du type **alphabétique**.

Lorsqu'elles contiennent en plus des chiffres, des symboles tels que (+, =, §, &, ...), nous parlerons de données **alphanumériques**.

Si la manière de coder en binaire des nombres entiers ou réels se fait d'une manière assez naturelle par un changement de base de numération, par contre la manière de stocker des caractères en binaire est totalement arbitraire. Ainsi, l'**ANSI** (American National Standard Institute) a défini un codage des caractères sur deux octets (ou bytes). Ce code porte le nom de code ASCII (American Standard Code for Information Interchange) et respecte, entre autres, l'ordre alphabétique. Comme deux octets permettent de stocker 256 valeurs différentes, nous disposerons de 256 caractères différents.

Il existe deux types de variables alphanumériques: les **caractère** et **chaîne**.

Le type caractère.

Le type caractère est réservé aux variables contenant **un** seul caractère (lettre, symbole, ponctuation, ...) et il est possible d'en déterminer les successeur/prédécesseur/position dans la liste des codes ASCII. Ainsi le successeur de "B" est "C", son prédécesseur "A" et son code ASCII 66.

En LDA, nous placerons dans la déclaration des variables une ligne telle que:

caractère lettre, initiale

qui se traduira en Pascal par:

VAR lettre, initiale: **CHAR;**

Le type chaîne.

Les variables du type chaîne peuvent contenir

soit une suite de caractères (un mot, une phrase, ...),

soit un caractère (mais dont, par exemple, il est impossible de déterminer le suivant),

soit aucun caractère (on parle alors de chaîne vide).

En LDA, nous placerons dans la déclaration des variables une ligne telle que:

chaîne nom, adresse

qui se traduira en Pascal par:

VAR nom, adresse: **STRING;**

Cependant, Pascal permet aussi de préciser la taille maximale (25 dans l'exemple ci-dessous) que pourra avoir la chaîne qui sera affectée à la variable:

VAR nom, adresse: **STRING[25]**;

En l'absence de précision de longueur, Pascal réserve automatiquement la taille maximale, à savoir 255 caractères.

Assignation.

Dans une assignation, le type de l'expression doit correspondre au type de variable de destination. C'est ainsi que les assignations suivantes sont illégales:

initiale <-- "Einstein"

initiale <-- ""

avec la variable initiale déclarée du type **caractère**.

Par contre, si cette variable a été déclarée du type **chaîne**, celles-ci sont tout à fait légales.

Les fonctions alphanumériques.

Notation en LDA	Fonction Pascal	Type du résultat	Signification
|Ch|	LENGTH(Ch)	Entier	Nombre de caractères dans Ch
Ch1 // Ch2	CONCAT(Ch1,Ch2) Ch1 + Ch2	Chaîne	Concaténation (juxtaposition) de Ch1 et Ch2
$Ch_{i<--j}$	COPY(Ch, i, j-i+1)	Chaîne	Extraction, dans Ch, des caractères de la postion i à la position j
Ch_i	COPY(Ch, i, 1) Ch[i]	Chaîne Caractère	Extraction, dans Ch, du caractère à la postion i

L'ALTERNATIVE.

3. STRUCTURES DE CHOIX.

3.1. Introduction

Les algorithmes décrits dans le chapitre précédent étaient très simples et une calculatrice aurait suffi pour leur exécution. En effet, pour l'instant, nous sommes seulement en mesure de décrire une suite d'opérations, chacune devant être exécutée une et une seule fois. Nous ne pouvons par exemple utiliser notre algorithme de calcul d'intérêt avec différents taux d'intérêt. Pour faire cela, il nous faut introduire de nouvelles structures.

Il a été démontré que **pour représenter n'importe quel algorithme**, il faut disposer des trois possibilités suivantes:

- la structure de séquence qui indique que les opérations doivent être exécutées les unes après les autres
- la structure de choix qui indique quel ensemble d'instructions doit être exécuté suivant les circonstances
- la structure de répétition qui indique qu'un ensemble d'instructions doit être exécuté plusieurs fois.

Jusqu'à présent, nous avons décrit la structure de séquence. Nous décrirons dans ce chapitre les structures de choix: l'alternative et le choix multiple.

3.2. La structure alternative.

Voici les règles d'un jeu très simple: deux joueurs A et B se cachent la main droite derrière le dos. Chacun choisit de tendre un certain nombre de doigts (de 0 à 5), toujours derrière le dos. Les deux joueurs se montrent la main droite en même temps. Si la somme des nombres de doigts montrés est paire, le premier joueur a gagné, sinon c'est le second. Le problème est de faire prendre la décision par l'ordinateur.

Exprimé en français, l'algorithme se présente comme suit:
- prendre connaissance du nombre de doigts de A
- prendre connaissance du nombre de doigts de B
- calculer la somme de ces deux nombres
- si la somme est paire, A est le gagnant
- si la somme est impaire, B est le gagnant.

Pour déterminer si un nombre est pair ou impair, il suffit de calculer le reste de la division par 2 (.. modulo 2): il vaut 0 dans le premier cas et 1 dans le second.

En langage de description d'algorithme, l'algorithme s'écrira:

```
entier na,nb,reste
lire na,nb
reste <-- (na + nb) mod 2
si reste = 0 alors écrire "Le joueur A a gagné."
sinon écrire "Le joueur B a gagné."
fsi
écrire "Bravo pour le gagnant!"
```

REMARQUES

(1) La structure alternative se présente en général sous la forme

```
si expression alors première séquence d'instructions
sinon deuxième séquence d'instructions
fsi
```

où *expression* conditionne le choix d'un des deux ensembles d'instructions. Cette *expression* peut être soit vraie soit fausse. Si l'expression est vraie, la première séquence d'instruction sera exécutée et la seconde sera ignorée; si l'expression est fausse, seule la seconde séquence d'instructions sera effectuée.

Le mot **sinon** indique où se termine la première séquence d'instructions et où commence la seconde. Le mot **fsi** (abrégé de "fin de si") indique où se termine la seconde séquence d'instructions.

(2) Dans certains cas, lorsque l'expression est fausse, aucune instruction ne doit être exécutée. La condition s'exprime alors plus simplement sous la forme:

si *expression* **alors** *séquence d'instructions*
fsi

(3) Quelle que soit la séquence choisie et exécutée, les instructions qui suivent **fsi** seront exécutées.

(4) Chacune des séquences d'instructions d'un **si** ... **fsi** peut contenir des **si**...**fsi**. On dit alors que les structures sont imbriquées.

(5) Pour faire apparaître plus clairement la structure, on écrit les séquences d'instructions légèrement en retrait des mots-clefs (**si**, **alors**, **sinon**, **fsi**). On dit qu'on **indente** le texte de l'algorithme.

Considérons l'exemple suivant écrit sans indentation et où les fins de si (**fsi**) ne sont pas indiquées:

```
si a > 0 alors
si b > 0 alors
c <-- a+b
sinon
c <-- a-b
```

Cet algorithme peut aussi bien être une mauvaise écriture de:

(2.1)

```
si a > 0 alors  si b > 0 alors c <-- a+b
                sinon c <-- a-b
                fsi
fsi
```

que de:

(2.2)

```
si a > 0 alors  si b > 0 alors c <-- a+b
                fsi
sinon c <-- a-b
fsi
```

Dans (2.1), si a est négatif, aucun traitement n'est effectué et dans (2.2) si a est négatif, c vaut a-b.

3.3.Expressions logiques et variables booléennes.

Expressions logiques.

Les expresssions logiques se construisent à partir d'affirmations qui sont soit vraies soit fausses. Une condition telle que "reste=0" n'impose pas à la variable *reste* d'être nulle. Il ne s'agit en effet pas d'une assignation mais de l'expression d'une condition qui ne sera réalisée (et n'aura donc la valeur "**vrai**") que si la variable *reste* a été assignée à 0. Dans les autres cas, cette condition prendra la valeur "**faux**".

On peut combiner des affirmations à l'aide d'**opérateurs logiques** ,à savoir: **ou**, **et** et **non**, les deux premiers portent sur deux opérandes et le dernier sur un seul. Il est évident que ces opérandes ne peuvent prendre que deux valeurs: **vrai** ou **faux**.

Par définition:

*op1***ou** *op2* n'a la valeur **faux** que si les deux opérandes ont la valeur **faux**, sinon l'expression a la valeur **vrai**.

*op1***et** *op2* n'a la valeur **vrai** que si les deux opérandes ont la valeur **vrai**, sinon l'expression a la valeur **faux**.

non *opérande* a la valeur **vrai** si l'opérande a la valeur **faux** et inversement.

Supposons par exemple qu'on exécute les assignations suivantes:

```
a <-- 1
a <-- 1
b <-- 2
c <-- 3
```

alors

(b > 8) **ou** (c < 1) a la valeur faux
(b > 0) **ou** (c > 1) a la valeur vrai
(b > 9) **ou** (c > 1) a la valeur vrai
(b > a) **et** (c > b) a la valeur vrai
(b > a) **et** (c < 0) a la valeur faux
non (c < a) a la valeur vrai
non ((b > a) **et** (c > b)) a la valeur faux
((b > a) **et** (c > b)) **ou** (a < 0) a la valeur vrai

Les relations d'égalité ou d'inégalité ci-dessus font comparer des nombres ou des variables à valeur numérique.

On peut également comparer des lettres: l'affirmation "b" **précède** "x" **dans l'ordre alphabétique** a pour valeur **vrai**. De même, si la variable *car* contient le caractère "v", l'affirmation *car* **suit** "w" **dans l'ordre alphabétique** a pour valeur **faux**.

Comme nous l'avons déjà vu, pour ramener des nombres à la seule représentation que connaissent les ordinateurs (le binaire), il suffit de faire un changement de base et passer de la base 10 à la base 2. Par contre, il n'existe pas de manière naturelle pour représenter un caractère sous forme binaire. On associe donc arbitrairement un code à chaque caractère connu par l'ordinateur, à savoir les signes du clavier (lettres, chiffres décimaux, signes de

ponctuation et caractères spéciaux) et les caractères de contrôle. Le code importe peu mais l'ordre de classement (**collating sequence**) qu'il définit est par contre très important. Le code **ASCII**, le plus fréquemment utilisé sur les micro-ordinateurs, respecte l'ordre croissant des caractères "0", "1", ..., "9" et l'ordre alphabétique des 26 lettres de l'alphabet, et ceci sans mêler aucun autre caractère dans ces séquences. Afin de simplifier les notations, on convient d'utiliser les symboles "**<**" pour signifier "**précède dans la collating sequence**", "" pour "**suit dans la collating sequence ou est égal**",...

Variables booléennes.

Il est possible de stocker la valeur d'une expression logique dans une variable (comme le résultat d'une opération arithmétique est stocké dans une variable numérique). Cette variable ne peut prendre que les valeurs **vrai** et **faux**. Ces variables sont appelées variables logiques ou **booléennes**.

En LDA, elles se déclarent comme ceci:

booléen: OK, pair

et en Pascal par:

```
VAR BOOLEAN: OK, pair;
```

El l'assignation d'une variable booléenne se fait de la manière suivante:

```
OK <-- Vrai
```

3.4. Exercices.

Pour voir les exercices et les solutions, cliquez ici.

3.5. L'alternative en Pascal.

En Pascal, l'exemple du jeu décrit en 2 se traduit par:

```
PROGRAM Jeu;
VAR NA, NB, Reste:INTEGER;
BEGIN
WRITE('Introduisez le nombre de doigts montrés par le joueur A');
READLN(NA);
WRITE('Introduisez le nombre de doigts montrés par le joueur B);
READLN(NB);
RESTE := (NA + NB) MOD 2;
IF RESTE=0 THEN WRITELN('Le joueur A a gagné.')
ELSE WRITELN('Le joueur B a gagné.');
WRITELN('Bravo pour le gagnant!')
END.
```

La structure alternative se traduit presque mot à mot de LDA en Pascal: **si** se traduit par **IF**, **alors** par **THEN** et **sinon** par **ELSE**. Les séquences d'instructions à exécuter dans le cas où la condition est vraie (cette séquence suit le mot **alors**) et dans le cas où la condition est

fausse (cette séquence suit le mot **sinon**) doivent être encadrées par le mots **BEGIN** et **END** qui en indiquent le début et la fin. Dans le cas où ces séquences se réduisent à une seule instruction, les mots **BEGIN** et **END** peuvent être omis (comme dans l'exemple ci-dessus).

Ainsi, en ajoutant le message **WRITELN**('Bravo pour le gagnant!') juste après avoir indiqué le nom du vainqueur, nous serions obligés d'utiliser les **BEGIN** et **END**:

```
PROGRAM Jeu;
VAR NA, NB, Reste:INTEGER;
BEGIN
WRITE('Introduisez le nombre de doigts montrés par le joueur A');
READLN(NA);
WRITE('Introduisez le nombre de doigts montrés par le joueur B);
READLN(NB);
RESTE := (NA + NB) MOD 2;
IF RESTE=0 THEN BEGIN
            WRITELN('Le joueur A a gagné.');
            WRITELN('Bravo pour le gagnant!')
            END
ELSE BEGIN
    WRITELN('Le joueur B a gagné.');
    WRITELN('Bravo pour le gagnant!')
    END
END.
```

La forme:

si *expression* **alors** *séquence d'instructions*
sinon *séquence d'instructions*
fsi

se traduit en Pascal par:

```
IF expression THEN BEGIN
            séquence d'instructions
            END
ELSE BEGIN
    séquence d'instructions
    END;
```

Il est interdit de mettre un **;** après le **END** indiquant la fin de la séquence d'instructions qui suit le **THEN**.

Et la forme:

si *expression* **alors** *séquence d'instructions*
fsi

se traduit en Pascal par:

```
IF expression THEN BEGIN
```

séquence d'instructions
END;

Remarques:

(1) La traduction en Pascal de (2.1) est:

```
IF A>0 THEN IF B>0 THEN C:=A+B
          ELSE C:=A-B;
```

En effet, en Pascal, le **ELSE** se rapporte toujours au **IF ... THEN** le plus proche.

Pour traduire (2.2), il faut soit modifier les alternatives pour que le **sinon** se rapporte au **alors** le plus proche:

```
si NOT (a=0) alors c <-- a - b
sinon si b > 0 alors  c <-- a + b
     fsi
fsi
```

ou faire apparaître que le **sinon** de la première condition est absent:

```
IF A>0 THEN IF B>0 THEN C:=A+B
          ELSE
ELSE C:=A-B;
```

(2) Le programme de jeu listé ci-dessus a un gros défaut: lorsque le premier joueur tape au clavier son nombre stocké dans NA, celui-ci reste affiché et le deuxième joueur peut choisir NB de façon à être certain de gagner.

Il existe deux remèdes à ce problème. La première consiste à effacer l'écran après que A ait tapé son nombre. Ceci se fait en Pascal par l'instruction **CLRSCR** qui nécessite le chargement d'une librairie. Celui-ci est activé par la ligne: **USES Crt;** écrite juste en-dessous de **PROGRAM** Jeu;

La deuxième solution est de saisir le caractère (le chiffre dans ce cas) tapé au clavier sans l'afficher. Cela se fait grâce à la fonction **READKEY** qui scrute en permanence le clavier et qui capte la touche qui vient d'être enfoncée. Cependant cette fonction ne peut être employée que pour l'affectation à une variable de type caractère.

Dans ce cas notre programme deviendrait:

```
PROGRAM Jeu;
USES Crt;
VAR NA, NB, Reste, Err : INTEGER;
   A, B : CHAR;
BEGIN
CLRSCR;
WRITE('Introduisez le nombre de doigts montrés par le joueur A');
A:=READKEY;
VAL(A, NA, Err);
IF Err<>0 THEN WRITELN('Erreur de format numérique!')
ELSE BEGIN
    WRITE('Introduisez le nombre de doigts montrés par le joueur B);
    B:=READKEY;
```

```
    VAL(B, NB, Err);
    IF Err<>0 THEN WRITELN('Erreur de format numérique!')
    ELSE BEGIN
        RESTE := (NA+NB) MOD 2;
        IF RESTE=0 THEN WRITELN('Le joueur A a gagné.')
        ELSE WRITELN('Le joueur B a gagné.');
        WRITELN('Bravo pour le gagnant!')
        END
END.
```

La procédure **VAL**(*Chaine*, *Nombre*, *Erreur*) permet de convertir *Chaine* en sa valeur numérique.qui sera stockée dans *Nombre*. La variable *Erreur* contiendra la valeur 0 si la conversion s'est faite sans problème ou la position du premier caractère inadéquat. *Erreur* doit être du type **INTEGER**, *Chaine* du type **STRING** ou **CHAR** et *Nombre* d'un type numérique (entier ou réel).

3.6. Le choix multiple.

Supposons que l'on veuille demander à l'utilisateur de choisir dans un menu une des 3 possibilités offertes. Le choix présenté ne se limite pas à une alternative (soit - soit).

Nous nous trouvons en présence d'un choix multiple qui s'écrit en LDA:

```
entier i
lire i
selon que
 i=1  faire bloc1
 oq i=2  faire bloc2
 oq i=3  faire bloc3
autrement écrire "Mauvais choix"
fselon
```

Le mot **oq** est l'abréviation de "**ou que**".

La structure de choix multiple peut, comme l'alternative, prendre deux formes:

selon que
première expression logique **faire** *première séquence d'instructions*
oq *deuxième expression logique* **faire** *deuxième séquence d'instructions*
...
oq *n ème expression logique* **faire** *n ème séquence d'instructions*
autrement (*n+1) ème séquence d'instructions*
fselon

et

selon que
première expression logique **faire** *première séquence d'instructions*
oq ...
oq *n ème expression logique* **faire** *n ème séquence d'instructions*
fselon

La première forme généralise le **si** ... **alors** ... **sinon** ... **fsi**, la seconde le **si** ... **alors** ... **fsi**.

Si la i ème expression logique a la valeur **vrai**, la i ème séquence d'instructions est exécutée puis il y a passage aux instructions qui suivent le mot **fselon**. Si toutes les expressions

logiques ont la valeur **faux**, on exécute dans le premier cas la séquence d'instructions qui suit le mot **autrement** , dans le deuxième cas on passe directement à ce qui suit **fselon**.

Afin d'éviter toute ambiguïté, on exige que les différentes expressions logiques soient mutuellement exclusives.

3.7. Le choix multiple en Pascal.

La traduction de **selon que** ...**fselon** en Pascal n'est pas très commode parce que la structure du choix multiple y est très restrictive: Il est seulement possible de traduire un **selon que** ... **fselon** pour lequel l'expression prend les valeurs prises par une variable de type énuméré (dont les valeurs appartiennent à un ensemble; par exemple entier, caractère).

La forme générale du choix multiple est:

```
CASE <expression> OF
  valeur1: première séquence d'instructions;
  valeur2: deuxième séquence d'instructions;
   ...
  valeurN: Nème séquence d'instructions;
ELSE (N+1) ème séquence d'instructions;
END;
```

Il est à noter que les différentes séquences d'instructions sont délimitées par **BEGIN** et **END** qui peuvent seulement être omis si une séquence se résume à une seule instruction.

Pour traduire en Pascal un **selon que** plus général, il faut utiliser des **IF** imbriqués. Par exemple,

```
 ...
lire x
selon que
x<0 faire écrire x,"est négatif"
 og x=0 faire écrire x,"est nul"
autrement écrire x,"est positif"
fselon
 ...
```

se traduit par:

```
...
READLN(x);
IF x<0 THEN WRITELN(x, ' est négatif.')
ELSE IF x=0 THEN WRITELN(x, ' est nul.')
     ELSE WRITELN(x, ' est positif.');
```

3.8. Exercices.

1) Lire 3 nombres a, b et c. Déterminer si l'équation ax+by+c=0 représente l'équation d'une droite parallèle à l'un des axes (et si oui, lequel) ou une droite oblique par rapport aux axes. Tenir compte du fait qu'on pourrait avoir a=b=0.

2) Lire 3 nombres a, b et c où a est différent de 0. Déterminer si la parabole d'équation $y=ax^2+bx+c$ coupe l'axe des x en 0, 1 ou 2 points.

3) Demander et lire les valeurs de R en Ohm, I en Ampère et t en seconde. Déterminer un algorithme qui proposerait de calculer la différence de potentiel, la puissance et l'énergie.

4. STRUCTURES REPETITIVES.

4.1. *Introduction.*

L'intérêt d'utiliser un ordinateur n'apparaît clairement que lors de la manipulation de données nombreuses ou traitées de manière répétitive.

Exemple 1: Chercher dans une liste de noms et d'adresses, l'adresse d'une personne à partir de son nom. Le nombre de fois qu'il faudra comparer le nom donné aux noms de la liste est dans ce cas inconnu.

Exemple 2 : Calculer la N ème puissance entière d'un nombre x par multiplications successives du nombre par lui-même. Ici, le nombre de répétition (N) de l'instruction de multiplication est connu.

S'il est théoriquement possible de se contenter d'une seule structure de répétition, bien choisie, pour exprimer tous les algorithmes, l'expérience a montré l'utilité d'en définir plusieurs, chacune bien adaptée à des circonstances particulières. Ce sont les boucles **tant que**, **répéter** ... **jusqu'à** et **pour**.

4.2. *La boucle " tant que "*.

Résolvons l'exemple 1:

```
...
lire nom_donné
lire nom1
si nom1 = nom_donné alors écrire adresse1
sinon lire nom2
    si nom2 = nom_donné alors écrire adresse2
    sinon lire nom3
        si nom3 = nom_donné alors ...
```

L'inconvénient de cet algorithme (en dehors de l'empiètement inévitable sur la marge de droite) tient au fait que l'auteur ne sait pas quand il doit s'arrêter d'écrire. Plus précisément, il lui est impossible de savoir combien de fois il doit écrire l'instruction de comparaison au nom_donné.

Un problème identique surgit dans l'écriture d'un algorithme qui décrit comment calculer le premier nombre premier qui soit plus grand qu'un nombre entier positif donné N.

```
lire N
i <-- N+1
si i est premier alors écrire i
sinon i <-- i+1
    si i est premier alors écrire i
```

```
    sinon i <-- i+1
        si i est premier alors écrire i
        sinon i <-- i+1
            si ...
```

Le test "si i est premier alors" ne sera exécuté qu'une fois si N=4, il le sera quatre fois si N=13, mais combien de fois faudra-t-il réécrire le test si N=7394485?

Ces exemples montrent que la séquence et l'alternative ne sont pas en elles-mêmes suffisantes pour exprimer des algorithmes dont la longueur peut varier selon les circonstances. Il est donc nécessaire d'introduire le moyen de répéter certaines instructions d'un algorithme un nombre quelconque de fois. La structure qui permet cela est appelée structure **répétitive**.

En utilisant la boucle "**tant que**", l'exemple 1 (avec la liste) s'écrit:

```
lire nom_donné
i <-- 1
lire nom_i
tant que NOT ((nom_i = nom_donné) ou (fin de liste)) faire
  i <-- i+1
  lire nom_i
ftant
si nom_i = nom_donné alors écrire adresse_i
sinon écrire "Le nom demandé ne se trouve pas dans la liste."
fsi
```

L'exemple avec le nombre premier s'écrira:

```
lire N
i <-- N+1
tant que NOT (i est premier) faire
  i <-- i+1
ftant
écrire i
```

Notons qu'il faudra exprimer autrement les conditions " (fin de liste) " et " (i est premier) ", ces formes n'étant pas compréhensibles par les compilateurs/interpréteurs.

Considérons aussi l'exemple suivant: étant donnés deux nombres entiers m et n positifs ou nuls, on demande d'en calculer le PGCD. L'algorithme d'Euclide permet de résoudre ce problème en prenant d'abord le reste de la division de m par n, puis le reste de la division de n par ce premier reste, etc jusqu'à ce qu'on trouve un reste nul. Le dernier diviseur utilisé est le PGCD de m et n. Pour m=1386 et n=140, on a successivement:

1386 = 140 * 9 + 126 140 = 126 * 1 + 14 126 = 14 * 9 + 0

et le PGCD de 1386 et 126 est bien 14.

Remarquons que par définition, si un des nombres est nul, l'autre nombre est le PGCD .

Si nous avions pris m=140 et n=1386, nous aurions obtenu la suite de calculs suivants:

140 = 1386 * 0 + 140 1386 = 140 * 9 + 126

140 = 126 * 1 + 14 126 = 14 * 9 + 0

et le PGCD est le même. L'ordre de m et n n'a donc pas d'importance.

Dans cet exemple, nous devons répéter le calcul du reste de la division d'un nombre par un autre. Pour fixer les idées, appelons a le dividende, b le diviseur et r le reste. Le calcul du reste de la division de a par b se fait simplement au moyen de l'instruction :

```
r <--  a mod b
```

L'algorithme s'écrit donc:

```
entier m,n,a,b,r,PGCD
lire m, n
a <-- m
b <-- n
tant que NOT(b = 0) faire
  r <-- a mod b
  a <-- b
  b <-- r
ftant
PGCD <-- a
écrire "Le PGCD de",m,"et",n,"est",PGCD
```

Commentaires

1) Les mots **faire** et **ftant** (abréviation de "fin de tant que") encadrent les instructions qui doivent être exécutées plusieurs fois. On indique entre **tant que** et **faire** les conditions dans lesquelles on doit exécuter le corps de la boucle.

2) Une boucle "**tant que**" se présente donc comme suit:

```
tant que expression logique faire
   séquence d'instructions
ftant
```

En premier lieu, l'*expression logique* est évaluée (il faut donc veiller à sa valeur lors de l'entrée dans la boucle): si sa valeur est **vrai**, le corps de la boucle est exécuté puis l'*expression logique* est réévaluée (il faut donc qu'elle puisse changer de valeur pour sortir de la boucle) et si elle a la valeur **faux**, on exécute l'instruction qui suit **ftant**.

3) Il est à noter qu'il est préférable d'exprimer l'*expression logique* sous la forme **NOT** (*condition(s) d'arrêt*). Il est en effet plus simple de déterminer les raisons d'arrêter le processus répétitif que celles de continuer.
La forme de ce type de boucle devient donc:

```
tant que NOT condition(s) d'arrêt faire
   séquence d'instructions
ftant
```

4.3. ***La boucle " pour"***.

Reprenons l'exemple 2 de l'introduction. Une boucle "tant que" permet de le résoudre:

```
entier N, i
réel x, puiss
lire N, x
```

```
puiss <-- 1
i <-- 1
tant que NOT(i > N) faire
  puiss <-- puiss * x
  i <-- i + 1
ftant
écrire "La puissance",N,"ème de",x,"est",puiss
```

Cependant, le nombre d' exécutions du corps de la boucle étant connu à l'avance, une boucle "pour" est d'un emploi plus simple:

```
entier N, i
réel x, puiss
lire N, x
puiss <-- 1
pour i de 1 à N faire
   puiss <-- puiss * x
fpour
écrire "La puissance",N,"ème de",x,"est",puiss
```

Commentaires

1° Les mots **faire** et **fpour** (abréviation de "fin du pour") encadrent les instructions qui doivent être exécutées plusieurs fois. On précise entre **pour** et **faire** comment seront contrôlées les répétitions. On y définit une variable appelée **variable de contrôle** (i dans l'exemple) et les valeurs que prendra cette variable: une *première valeur* ou *valeur initiale* indiquée après le mot **de**, une *dernière valeur* ou *valeur finale* indiquée après le mot **à**. La variable de contrôle est initialisée à la première valeur. **Avant** chaque exécution du corps de la boucle, la valeur de la variable de contrôle est comparée à la *valeur finale*. Si la variable de contrôle ne dépasse pas cette valeur, on exécute le corps de la boucle, sinon on passe à l'instruction qui suit le mot **fpour**. **Après** chaque exécution du corps de la boucle, la variable de contrôle est augmentée d'une unité.

2° Dans l'exemple ci-dessus, la variable de contrôle sert uniquement de compteur du nombre d'exécutions du corps de la boucle. Si on le souhaite, on peut utiliser cette valeur dans le corps de la boucle, entre autres pour faire un calcul fondé sur cette valeur. Dans tous les cas, il faut éviter de modifier la valeur de cette variable.

3° Il est parfois utile de faire varier la variable de contrôle par valeurs décroissantes, ou par incréments différents de l'unité. Dans ce cas, il faut utiliser la forme la plus générale de la boucle "**pour**":

```
pour v.c. de prem.val. à dern.val. par incr faire
   séquence d'instructions
fpour
```

où

- *v.c.* est le nom de la variable de contrôle
- *prem.val.* est la première valeur (ou valeur initiale)
- *dern.val.* est la dernière valeur (ou valeur finale)
- *incr.* est l'incrément, c'est-à-dire la quantité non nulle ajoutée à la variable de contrôle à la fin de chaque exécution du corps de la boucle.

Il est important de noter ici l'impossibilité de traduire en Pascal les boucles avec des incréments différents de 1 et -1.

Avant chaque exécution du corps de la boucle, la valeur de la variable de contrôle est comparée à la dernière valeur. L'exécution s'arrête

- lorsque la variable de contrôle a une valeur supérieure (strictement) à la dernière valeur si l'incrément est positif
- lorsque la variable de contrôle a une valeur inférieure (strictement) à la dernière valeur si l'incrément est négatif.

Par convention, lorsque l'incrément est égal à 1, il n'est pas indiqué.

4° La première valeur, la dernière valeur et l'incrément peuvent être des expressions numériques. Les instructions du corps de la boucle ne peuvent en aucun cas modifier ces valeurs.

5° En fin de boucle, la valeur de la variable de contrôle n'est pas toujours égale à la valeur finale (à vérifier avec votre compilateur). Il est donc dangereux d'utiliser celle-ci.

6° Il est théoriquement possible de modifier la valeur de la variable de contrôle à l'intérieur de la boucle mais cette technique est à proscrire et ne masque pas un manque de réflexion lors de l'analyse du problème. Cette manière de faire revient en fait à transformer artificiellemnt une boucle *pour* en boucle *tant que*.

4.4. ***La boucle " répéter ... jusqu'à "***.

Comme la boucle "tant que", ce type de répétitive est utilisé lorsque le nombre de fois que la séquence d'instructions à répéter est inconnu au moment où cette séquence est abordée pour la premiè fois mais le corps de la boucle est **toujours exécuté au moins une fois**.

Sa formulation générale est:

répéter
séquence d'instructions
jusqu'à (*expression logique*)

L'*expression logique* est évaluée aprés l'exécution du corps de la boucle: si sa valeur est **faux**, le corps de la boucle est exécuté à nouveau puis l'*expression logique* est réévaluée (il faut donc qu'elle puisse changer de valeur pour sortir de la boucle) et si elle a la valeur **vrai**, on exécute l'instruction qui suit **jusqu'à**. Attention, si ceci correspond tout à fait à l'usage familier que nous faisons de l'expression répéter ... jusqu'à, le test effectué est la négation de celui utilisé dans la boucle "tant que" et ceci peut parfois prêter à confusion.
Il faut en effet remarquer que dans

tant que *expression logique* **faire**
séquence d'instructions
ftant

expression logique exprime les raisons de continuer et dans

répéter
séquence d'instructions
jusqu'à (*expression logique*)

expression logique exprime les raisons d'arrêter.

4.5. ***Exercices.***

1) Lire un nombre entier et déterminer s'il est premier.

2) Lire un nombre entier et déterminer tous ses diviseurs.

3) L'ordinateur "choisit" un nombre entier compris entre 0 et 100. Un joueur essaie de le deviner. Lors de chaque essai, l'ordinateur affiche la "fourchette" dans laquelle se trouve le nombre qu'il a choisi.

4) Lire un entier N positif et non nul. Ecrire un algorithme qui calcule le Nème nombre de la suite de Fibonacci. Ceux-ci se calculent ainsi:

F(0) = 0, F(1) = 1 et F(i) = F(i-1) + F(i-2) pour i > 1.

Résumé:
Le problème principal est de bien choisir le type de boucle à utiliser. Voilà les questions à se poser pour faire le bon choix: Le nombre de répétitions est-il connu *au moment du premier passage dans la boucle*? Oui et bien alors, il ne faut pas hésiter. Il faut utiliser une boucle pour...fpour Non. Alors il faut se demander si on est sûr que le corps de la boucle sera effectué au moins une fois. Oui et bien alors, il semble raisonnable d'utiliser la boucle répéter...jusqu'à. Non. Alors pas d'hésitation, c'est la boucle tant que...ftant qu'il faut utiliser. Dernier conseil: sachant que les conditions d'arrêt d'une boucle sont en général plus simples à trouver que les raisons de continuer, surtout parce qu'elles s'expriment souvent sous forme condition1 ou condition2 ou ..., il est préférable d'utiliser la boucle tant que sous la forme tant que **NOT** (condition1 ou condition2 ou ...) .... ftant.

En pratique:
Imaginons que vous ayiez: écrire "Introduisez un nombre positif non nul" lire N si N>0 alors 'arrêter de demander'(*) écrire "Introduisez un nombre positif non nul" lire N si N>0 alors 'arrêter de demander' écrire "Introduisez un nombre positif non nul"

lire N
si N>0 alors 'arrêter de demander'
....
On remarque qu'entre 2 tests d'arrêt (si N>0 alors 'arrêter de demander') se situe ce qu'on appelle le corps de la boucle
écrire "Introduisez un nombre positif non nul"
lire N

Même s'il semble évident que le test d'arrêt se trouve en (*), il y a deux manières de traduire en boucle:
(1) placer le test d'arrêt après l'exécution du corps de la boucle donne:

```
répéter
  écrire "Introduisez un nombre positif non nul"
  lire N
jusqu'à N>0
```

(2) placer le test d'arrêt avant l'exécution du corps de la boucle donne:

```
écrire "Introduisez un nombre positif non nul"
lire N
tant que NOT (N>0) faire
  écrire "Introduisez un nombre positif non nul"
  lire N
ftant
```

Vous remarquerez que le test d'arrêt est bien toujours en 3ème ligne.

Alors quelle version choisir?

Au nombre de lignes, c'est la première version qui l'emporte mais pourtant beaucoup préfèrent la seconde.

Pourquoi?

Simplement parce qu'elle peut être très facilement modifiée en une version qui prévient l'utilisateur de sa faute:

```
écrire "Introduisez un nombre positif non nul"
lire N
tant que NOT (N>0) faire
  écrire "Le nombre introduit n'est pas positif!!"
  écrire "Introduisez un nombre positif non nul"
  lire N
ftant
```

En effet juste après le test tant que NOT (N>0) faire, nous sommes sûrs de ne pas avoir N>0.
Nous voyons ainsi que bien que le corps de la boucle soit exécuté au moins une fois, nous avons utilisé une boucle tant que et non une boucle répéter.

4.6. ***Tableaux ou variables indicées.***

Reprenons l'exercice 3) et décidons de garder en mémoire les différents essais du joueur. Il est évidemment impossible de stocker chaque essai dans une variable de nom différent car cette technique nous empêche d'employer une boucle (les noms de variables du corps de la

boucle ne peuvent en effet pas changer d'une répétition à l'autre); or ici, l'emploi d'une boucle est obligatoire car on ne connaît pas à l'avance le nombre d'essais qui seront tentés.

Exemple

Nous voudrions stocker les notes de l'examen semestriel pour le calcul de la moyenne de l'examen et pour un futur calcul du total semestriel.

S'il fallait calculer uniquement la moyenne, on pourrait utiliser l'algorithme suivant:

```
entier note,total,max,maxnote,moyenne
total <-- 0
max <-- 0
pour i de 1 à 16 faire
   lire note,maxnote
   si NOT(note < 0) alors  total <-- total + note
                           max <-- max + maxnote
   fsi
fpour
moyenne <-- [total/max * 100 + .5]
écrire "La moyenne est de",moyenne,"%"
```

où pour indiquer une absence ou une dispense pour une certaine branche, nous avons attribué une note négative dont il ne faut évidemment pas tenir compte dans le calcul du total des points obtenus et du total des points attribués.

Mais comme il faut retenir les notes pour un calcul ultérieur, cet algorithme doit être modifié. L'utilisation d'une **variable indicée** (ou t**ableau** ou **table**) nous fournit une solution, $note_i$ **représentant** la note de la ième branche et $maxnote_i$ le maximum attribué dans la branche numéro i.

Dans l'algorithme qui suit, on retient le nom de la branche en utilisant une autre variable indicée liste appelée branche ($branche_i$ **représentant** le nom de la ième branche).

```
tableau note_i, i=1,2,...,10 d'entier
        maxnote_i , i=1,2,...,10 d'entier
        branche_i, i=i,...,10 de chaine
entier max,total, moyenne
total <-- 0
max <-- 0
pour i de 1 à 16 faire
    écrire "Introduisez le nom de la branche ",i
    lire branche_i
    écrire "Introduisez la note de la branche ",i
    lire note_i
    écrire "Introduisez le maximun de la branche ",i
    lire maxnote_i
    si NOT(note_i < 0) alors  total <-- total + note_i
                              max <-- max + maxnote_i
    fsi
fpour
moyenne <-- [total / max * 100 + .5]
écrire "La moyenne est de",moyenne,"%"
```

Définitions et commentaires

Pour définir les variables indicées, il faut introduire un nouveau type de variable dans le langage de description que nous utilisons: le type **tableau**. Chaque variable indicée -chaque tableau- qui sera utilisée doit être décrite dans la partie déclarative de l'algorithme; la description comprend trois parties:

1° La première partie décrit le nom de la variable et le nombre de ses indices: note, maxnote et branche dans notre exemple, chacune avec un indice. Pour indiquer que t, u sont des variables à 2, 3 indices, on aurait écrit:

$t_{i,j}$ ...**de** ...
$u_{i,j,k}$ ...**de** ...
Les symboles utilisés (i, j, k, ...) pour indiquer le nombre d'indices servent uniquement dans la deuxième partie de la description. Si on le désire, on peut utiliser les mêmes symboles dans le reste de l'algorithme, mais on n'y est pas obligé.

2° La deuxième partie décrit pour chaque variable les valeurs que peuvent prendre ces indices: les entiers de 1 à 16 pour chaque variable de notre exemple. Comme on le voit, cet ensemble de valeurs peut être représenté de plusieurs façons différentes.

On appelle $note_1$,...,$note_{16}$ les **composantes** (ou **éléments**) du tableau note, de même pour $maxnote_i$ et $branche_i$. Chaque composante se comporte comme une variable semblable à celles que nous avons manipulées jusqu'à présent: on peut la lire ou l'écrire, utiliser sa valeur dans une expression, lui assigner une valeur ...

3° Comme toute variable, chaque composante d'un tableau doit avoir un type, défini dans la troisième partie de la description du tableau: les composantes du tableau branche sont des chaînes, celles de note et maxnote sont des entiers. Toutes les composantes d'un tableau doivent être du même type: on ne peut pas définir un tableau dont certaines composantes seraient des nombres et d'autres des chaînes de caractères.

Lorsqu'on utilise une composante d'un tableau, les indices peuvent être un nom de variable comme dans l'algorithme, ou un nombre, ou même une expression. Par exemple si i=3 et N=16, $note_i$, $note_3$ et $note_{N-13}$ représentent tous la même composante du tableau note. Par analogie avec les variables indicées utilisées en mathématique, on appelle souvent **vecteur** un tableau à 1 indice et **matrice** un tableau à 2 indices: les déclarations suivantes sont équivalentes:

tableau a_i , i=1,2,...,20 **de réel**
$b_{i,j}$,i=1,2,...,9 , j=1,2,...,5 **de chaîne**

équivaut à:

vecteur a_i ,i=1,2,...,20 **de réel**
matrice $b_{i,j}$, i=1,2,...,9 , j=1,2,...,5 **de chaîne**

4.7. ***Exercices***.

1) Etant donné un vecteur de nombres triés par ordre croissant, chercher si un nombre donné x figure parmi les composantes. Si oui, indiquer la valeur de l'indice correspondant.

2) Bonhomme pendu. L'algorithme lit un mot proposé par un premier joueur. L'algorithme affiche ensuite le mot où toutes les lettres sauf la première et la dernière sont remplacées par un tiret. Un deuxième joueur propose des lettres une à une. Chaque fois que la lettre se trouve dans le mot, l'algorithme remplace les tirets qui remplaçaient cette lettre et réaffiche le mot. Le second joueur a droit à un maximum de 6 essais infructueux (lettre ne se trouvant pas dans le mot).

3) Carrés magiques d'ordre impair. Un carré magique d'ordre n est un tableau carré à n lignes et n colonnes, donc de n^2 cases, dans lesquelles on écrit une et une seule fois les nombres entiers de 1 à n^2, de telle sorte que la somme des n nombres de chaque ligne, de chaque colonne et de chaque diagonale soit toujours la même. On dispose pour les carrés magiques d'ordre impair de l'algorithme suivant:
En désignant par (i,j) la case de la ligne i et de la colonne j:
(a) on place 1 en (n , (n+1)/2)
(b) ayant placé le nombre k dans la case (i,j) on place le nombre k+1 dans la case (i+1,j+1), toutefois si cette case est occupée on place le nombre k+1 dans la case (i-1,j).
Lorsque les indices calculés sont supérieurs à n, on prend 1 comme indice. Ces opérations se font pour $1<=k<=n^2-1$.

4) Jeu de Marienbad. On place des allumettes sur 4 rangées: 1 allumette sur la première, 3 sur la seconde, 5 sur la troisième et 7 sur la dernière. Deux joueurs peuvent alternativement ôter, sur une seule rangée à la fois, autant d'allumettes qu'ils le souhaitent mais au moins une. Le joueur qui enlève la dernière allumette a perdu.

5) Lire un texte d'au moins 120 caractères. Compter et afficher le nombre d'occurences (d'apparition) de chacune des lettres de l'alphabet.

6) Nous désignerons par a_1, a_2, ..., a_n les éléments d'un tableau à trier par ordre croissant. On commence par chercher l'indice du plus petit des éléments, soit j cet indice. On permute alors les valeurs de a_1 et a_j .On cherche ensuite l'indice du plus petit des éléments a_2, a_3, ..., a_n et on permute avec a_2, etc.

Résumé:
Seuls les tableaux (et les listes chaînées) permettent de stocker/traiter des données d'une manière répétitive. C'est la seule manière de générer automatiquement des noms de variable.

4.8. ***Les boucles en Pascal***.

Et l'exemple du calcul du PGCD de deux nombres se traduit par:

```
PROGRAM Plus_Grand_Commun_Diviseur;
VAR m,n,a,b,r,PGCD : INTEGER;
BEGIN
WRITELN('Nous allons calculer le PGCD de deux nombres entiers.');
WRITE('Introduisez le premier nombre: '); READLN(m);
```

```
 WRITE('Introduisez le deuxième nombre: '); READLN(n);
 a := m; b := n;
 WHILE NOT(b = 0) DO
   BEGIN
   r := a mod b;
   a := b;
   b := r
   END;
 PGCD := a;
 WRITELN('Le PGCD de ',m,' et ',n,' est ',PGCD)
 END.
```

En Pascal, la boucle **tant que** se traduit par

```
 WHILE expression logique DO
   BEGIN
   séquence d'instructions;
   END;
```

où la *séquence d'instructions* est entourée comme d'habitude par un **BEGIN** et un **END** qui peuvent être omis lorsque la séquence se réduit à une seule instruction.

Le calcul de la puissance se traduit en Pascal:

```
 PROGRAM Puissance;
 VAR N,i : INTEGER;
    x, Puiss: REAL;
 BEGIN
 WRITELN('Calcul de la puissance N ème du réel x par multiplications successives.');
 WRITE('Introduisez le nombre réel x: ');READLN(x);
 WRITE('Introduisez l'exposant de x (entier positif): ');READLN(N);
 Puiss := 1;
 FOR i:=1 TO N DO
    Puiss:=Puiss*x;
 WRITELN('La puissance ',N,' ème de ', x,' est ',Puiss)
 END.
```

La boucle **pour** en Pascal n'exige pas de spécifier explicitement l'incrément si celui-ci est égal à un. Elle se traduit dans ce cas, comme dans notre exemple, par:

```
 FOR v.c.:=prem.val. TO dern.val. DO
   BEGIN
   séquence d'instructions
   END;
```

Le Pascal n'accepte, en dehors de l'incrément égal à 1, que l'incrément -1. Dans ce cas, la traduction est:

```
 FOR v.c.:=prem.val. DOWNTO dern.val.DO
   BEGIN
   séquence d'instructions
   END;
```

où **TO**et **DOWNTO** indiquent que la variable de contrôle v.c. sera incrémentée (augmentée de une unité pour **TO** ou diminuée de une unité pour **DOWNTO**) après chaque exécution du corps de la boucle. Si la valeur de la v.c. dépasse (est strictement plus grande pour **TO** ou strictement plus petite pour **DOWNTO**) la dern. val., le corps de la boucle n'est plus exécuté et le programme exécutera les instructions qui suivent la dernière instruction du corps de la boucle.

Le corps de la boucle **NE** sera **PAS** exécuté si la prem.val. est supérieure à dern.val. pour **TO** ou si la prem.val. est inférieure à dern.val. pour **DOWNTO**.

Dans le cas de boucles **FOR ... DO** imbriquées, les variables de contrôle doivent avoir des noms différents et le la fin de la boucle intérieure doit apparaître avant celle de la boucle extérieure.

La traduction en Pascal de:

répéter
séquence d'instructions
jusqu'à (*expression logique*)
se fait pratiquement mot à mot:
REPEAT
séquence d'instructions
UNTIL (*expression logique*);
Remarque: La *séquence d'instructions* n'est pas entourée par **BEGIN** ... **END**.

4.9. ***Les tableaux en Pascal***.

En Pascal, un tableau est déclaré du type **ARRAY** et le nombre de dimensions (indices) du tableau s'indique entre crochets en citant les valeurs minimales et maximales successives des différents indices, séparées par deux points (**..**). La déclaration du type des éléments contenus dans le tableau se fait en ajoutant **OF INTEGER**, ou **OF REAL**, ...

Exemple:

tableau $a_{i,j,k}$ i=0, ..., 12 , j=1, ..., 5 , k=12, ..., 19 **d'entiers**
b_i i=12, ..., 25 **de chaînes**
$c_{i,j,k,l}$ i=1,...,6 , j=1,...,15 , k=1,...,9 , l="a",...,"z" **de réels**

se traduit en Pascal par:

```
VAR a: ARRAY[0..12, 1..5, 12..29] OF INTEGER;
    b: ARRAY[12..25] OF STRING;
    c: ARRAY[1..6, 1..15, 1..9, 'a'..'z'] OF REAL;
```

Il faut remarquer que l'indice d'un tableau ne doit pas nécessairement être du type entier. Tout type énuméré (dont le suivant est connu) peut être utilisé. L'élément $a_{i,j,k}$ de ce tableau s'écrit **a[i,j,k]**en Pascal. De même, l'élément $a_{5,n-2,[k/2]}$ se traduira par: **a[5,n-2,TRUNC(k/2)]**.

En Pascal, comme toute autre type de variable, un tableau est remis à "zéro" ou à "blanc" mais il est préférable de ne pas préjuger de cette situation qui peut évoluer avec les versions du compilateur.

Il faut remarquer que la valeur maximale permise pour un indice ne peut être lue ou calculée en cours d'exécution d'un programme (**allocation dynamique des tableaux**) et elle ne peut donc varier d'une exécution à l'autre. Il faut par conséquent surestimer le nombre d'éléments nécessaires pour en disposer dans tous les cas d'un nombre suffisant en cours d'exécution du programme.

4.10 ***Exercices***.

EPSIC

1) Calculer la racine carrée de a grâce à la formule récurrente x = 1/2 (x + a/x). Les calculs commenceront avec 1 comme valeur initiale de x et stopperont quand la valeur absolue de la différence entre les deux dernières valeurs calculées sera inférieure à 10^{-6}.

2) Lire deux polynômes et en calculer la somme et le produit.

3) Lire un polynôme et en calculer la valeur pour x = a (a est lu au clavier).

4) Lire un nombre entier, la base dans lequel il est exprimé et celle dans lequel il doit être exprimé. Effectuer la conversion et afficher le résultat.

5) Lire une chaîne de caractères et la mettre en majuscules.

6) Lire une chaîne de caractères et en déterminer le nombre de mots.

7) Lire une chaîne de caractères et en déterminer le nombre de caractères différents.

8) Lire deux nombres entiers N1 et N2. Si un des nombres est nul, l'autre est le PGCD sinon il faut soustraitre le plus petit du plus grand et laisser le plus petit inchangé. Puis, recommencer ainsi avec la nouvelle paire jusqu'à ce que un des deux nombres soit nul. Dans ce cas, l'autre nombre est le PGCD

9) Lire deux nombres entiers N1 et N2. Assigner à N1 la valeur de N2 et à N2 la valeur du reste de la division de N1 par N2. Puis recommencer jusqu'à ce que le reste de la division soit nul. A ce moment, N1 contient le PGCD.

10) Calculer a^b avec a réel et b entier par multiplications successives.

11) Rédigez en LDA un algorithme qui réalise le jeu suivant:
(a) A tour de rôle, l'ordinateur et le joueur choisissent un nombre qui ne peut prendre que 3 valeurs: 0, 1 ou 2.
Note: l'instruction

```
N <-- RANDOM(3)
```

réalise le choix de l'ordinateur.
(b) Si la différence entre les nombres choisis vaut

- 2, le joueur qui a proposé le plus grand nombre gagne un point
- 1, le joueur qui a proposé le plus petit nombre gagne un point
- 0, aucun point n'est marqué.

(c) Le jeu se termine quand un des deux joueurs (l'ordinateur ou le joueur humain) totalise 10 points ou quand l'être humain introduit un nombre négatif qui indique sa volonté d'arrêter de jouer.