Chambre des Employés Privés

Méthodologie de la programmation

Nino Silverio

Contenu de la formation

- les langages de programmation
  - le langage Leria
- le développement d'un programme
  - la séquence d'instructions, la variable, le type d'une variable, l'affectation
- la structure alternative
  - l'expression conditionnelle, le test d'un programme

Le cours débute par un aperçu général de la programmation d'un ordinateur (langages d'assemblage/langages évolués impératifs). L'accent est mis sur le langage structuré LERIA développé par l'auteur.

Les différentes étapes de la réalisation d'un programme sont examinées en détail et illustrées à travers un exemple concret. Ceci est l'occasion d'étudier les instructions de base et la mise en séquence de ces instructions pour créer le programme désiré.

La structure alternative (simple et complète) permet d'introduire le concept de l'exécution conditionnelle d'une instruction. Ce chapitre permet aussi d'introduire les notions de condition booléenne, l'évaluation d'une condition, les connecteurs logiques ET, OU et NON.

Contenu de la formation

- la structure répétitive
  - la condition d'arrêt, l'invariant d'une boucle
- le concept de fonction
  - les variables locales, la portée d'un identificateur
  - la transmission des arguments
- les tableaux à une dimension

L'étude de la structure répétitive permet d'utiliser toute la puissance d'un ordinateur en lui faisant répéter inlassablement une séquence d'instructions. Dans ce chapitre, nous abordons évidemment le concept de terminaison (arrêt) d'un programme ainsi que la méthodologie de l'écriture de boucles (approche intuitive de l'invariant de boucle).

Après avoir étudié les structures de contrôle fondamentales permettant d'écrire n'importe quel programme, notre attention se porte sur des concepts permettant de réaliser un programme de manière plus efficace et plus élégante.

C'est ainsi que nous commençons par l'étude du concept de fonction (outil de structuration du code) et les notions liées à son utilisation (variables globales/locales, portée d'un identificateur, modes de transmission des arguments...).

Un autre concept important est le tableau à une dimension (vecteur) que l'on retrouve dans la plupart des langages de programmation actuels.

Contenu de la formation

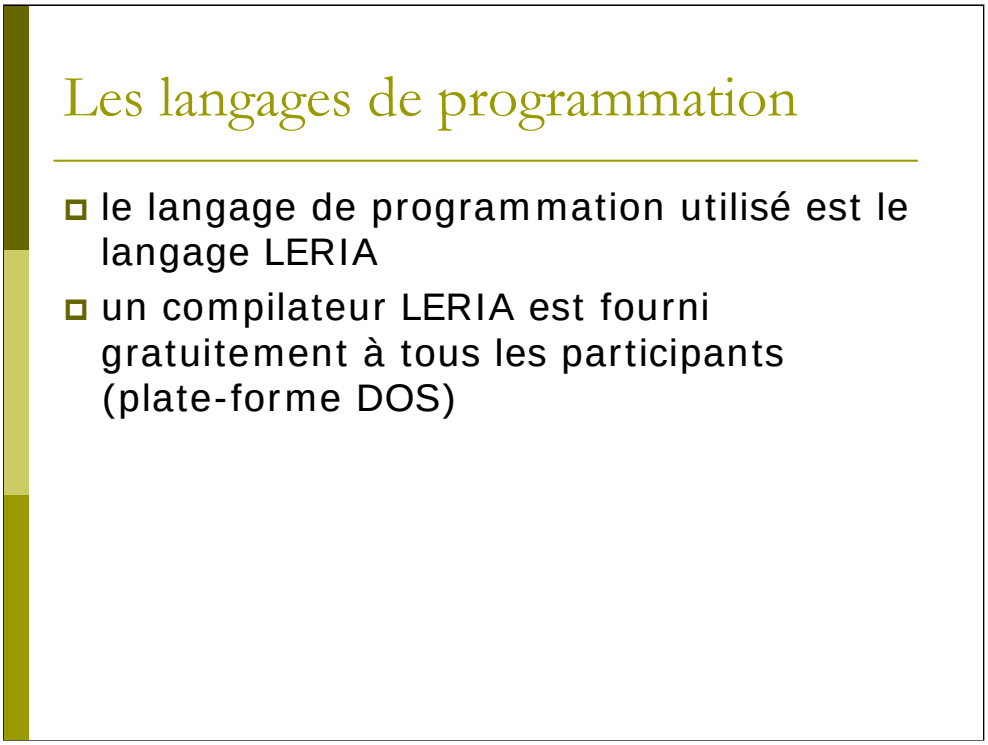
- la notion d'enregistrement
- les tablaux multidimensionnels
- la récursion
- quelques exercices de synthèse

D'autres outils fort pratiques sont les tableaux multidimensionnels ainsi que l'enregistrement (*record*).

Le dernier sujet théorique abordé est la récursion, moyen très puissant pour résoudre certains problèmes de manière très simple et élégante.

Le cours se termine par des exemples de synthèse classiques mettant en œuvre tous les concepts étudiés.

Les langages de programmation



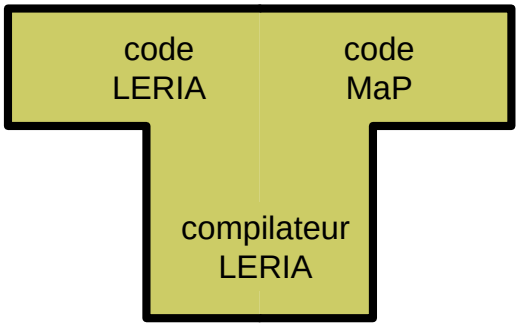
Le but de cette formation est d'apprendre aux participants les concepts de base d'une programmation saine afin de pouvoir poursuivre par la suite des cours de programmation avancés (C, Visual Basic, Cobol, C++, etc.).

Le langage de programmation utilisé a peu d'importance, pourvu qu'il intègre les concepts nécessaires à la transmission de ce savoir. Les langages simples usuels (Pascal, Modula...) employés traditionnellement à cette fin ont le désavantage d'être des langages complets avec beaucoup d'options et de gadgets compliquant inutilement un cours de méthodologie de la programmation.

C'est pourquoi l'auteur a développé un langage de programmation très simple intégrant uniquement les concepts de base nécessaires au cours : le langage LERIA.

Un compilateur LERIA est fourni gratuitement à tout participant au cours lui permettant ainsi de mettre en oeuvre les exemples et exercices étudiés.

Les langages de programmation



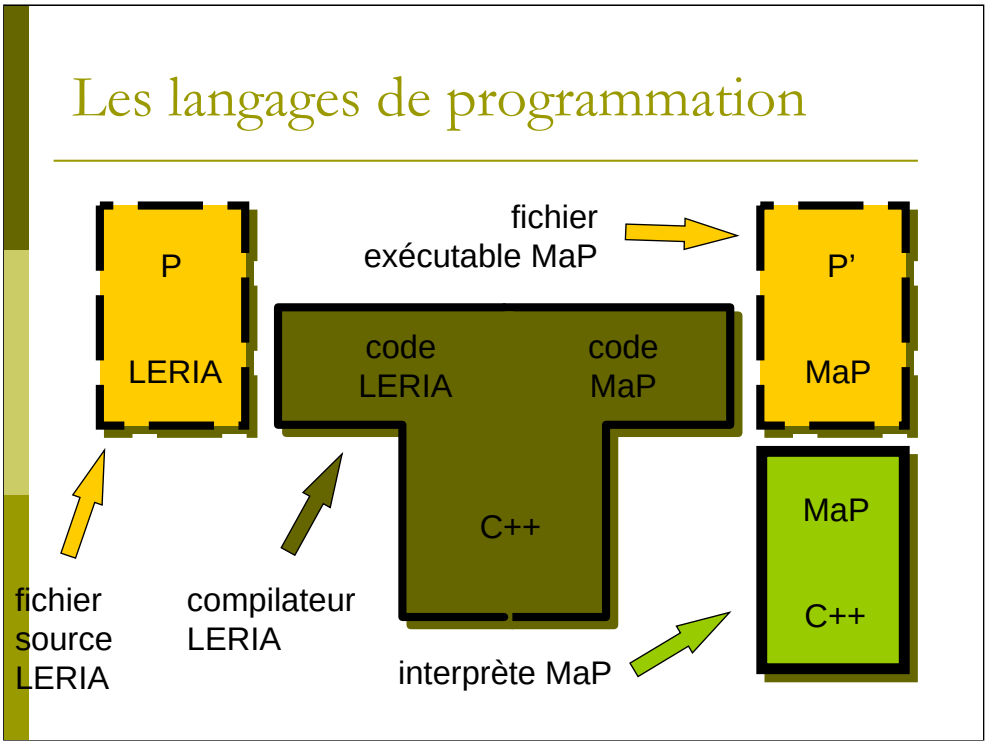
Un T-diagramme

Grâce au compilateur LERIA fourni, nous sommes à même de traduire les programmes écrits dans le langage LERIA en un code exécutable, le code pour le processeur MaP.

La question qui surgit tout de suite est : qu'est-ce que le processeur MaP ? Moi j'ai un ordinateur avec un processeur 80486, Pentium, PowerPC, Sparc... Comment est-ce que ce code va pouvoir s'exécuter sur mon ordinateur ?

La solution consiste à interpréter le code pour le processeur MaP. Cet interprète est aussi fourni dans le cadre de ce cours.

Le compilateur LERIA ne génère donc pas du code pour un processeur réel disponible sur le marché, mais pour une machine virtuelle, le processeur MaP, créé de toutes pièces par l'auteur.



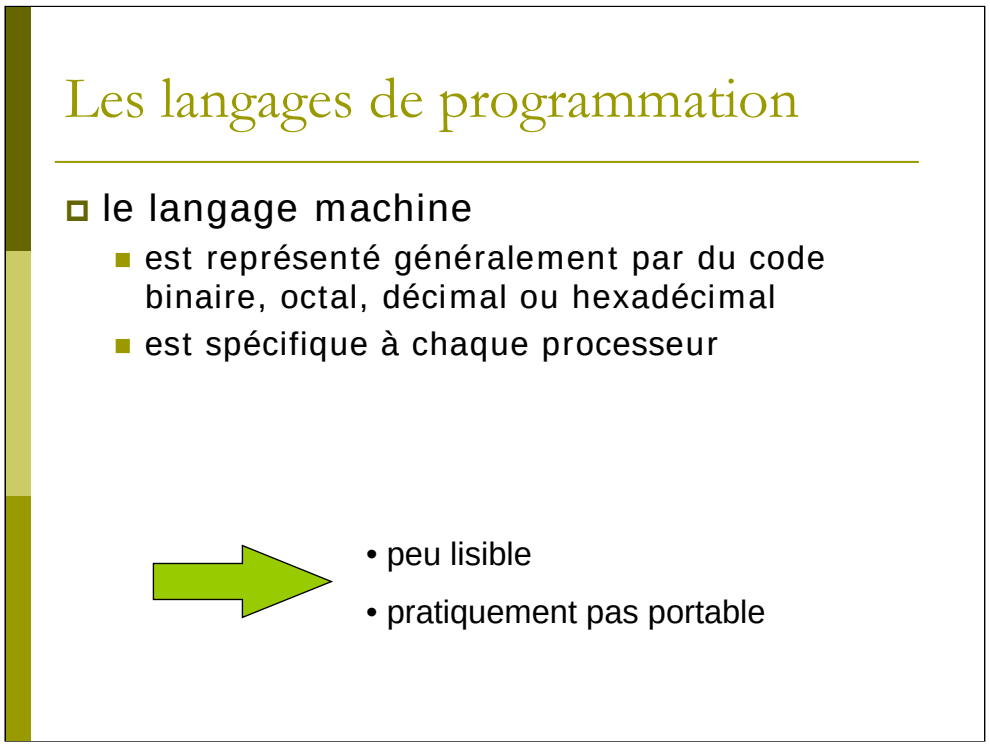
Lors de la traduction d'un programme P écrit en LERIA, le compilateur LERIA, lui-même écrit en C++ standard, produit un programme équivalent P' en code MaP.

Pour que ce programme puisse être exécuté, il faut utiliser un interprète capable d'exécuter des instructions MaP. Cet interprète est lui aussi écrit en C++ standard.

L'avantage principale de l'approche de générer du code pour une machine virtuelle est la grande portabilité du compilateur et la simplicité relative du code du compilateur. En effet, réaliser un compilateur capable de générer du code pour différents processeurs réels est un travail lourd et fastidieux, rendu compliqué du fait des idiosyncrasies de chaque processeur.

Ainsi, par exemple, le portage du compilateur LERIA, développé avec le Borland C++ (sur un Toshiba doté d'un Pentium), sur le Cray du Centre Universitaire (processeur Risc SPARC) a pu se faire en moins d'une heure !

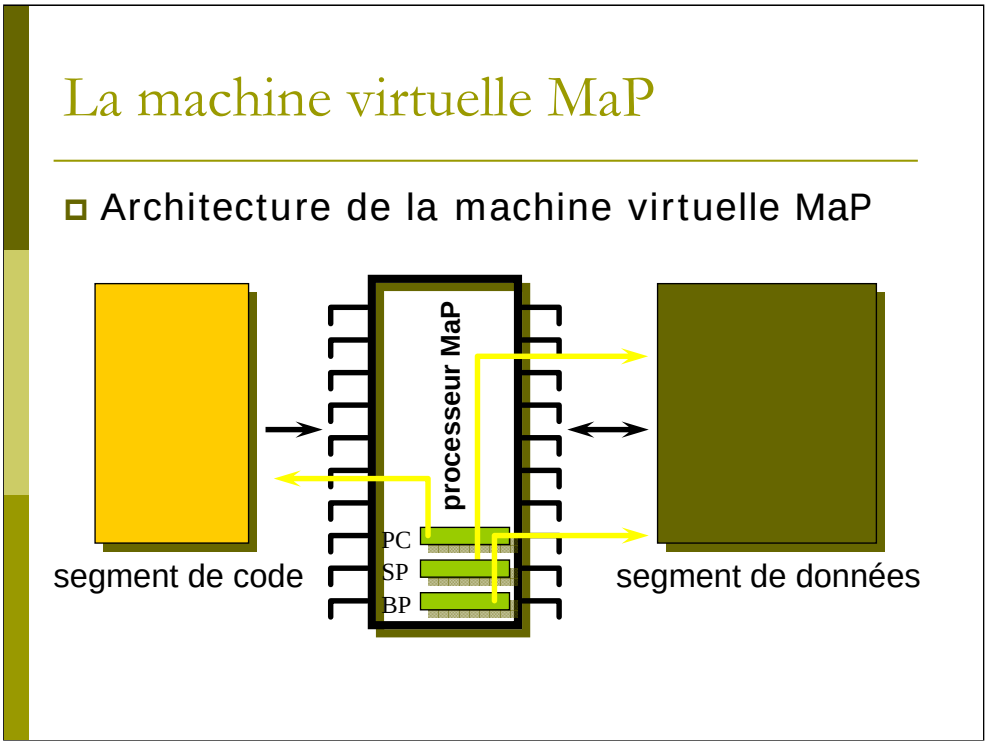
Le principal désavantage de cette technique est la relative lenteur du code lors de l'exécution, puisque ce dernier sera interprété. Cette technique a notamment été utilisée pour réaliser le compilateur portable Pascal-P qui produisait du code objet (P-code) pour une machine abstraite.



Le langage de base de chaque ordinateur est le langage machine qu'il «comprend».

Le programmeur peut programmer son ordinateur directement en langage machine en lui fournissant les instructions à exécuter sous forme de codes numériques en binaire, octal, décimal ou hexadécimal...

Cette façon de faire est évidemment très peu pratique et surtout extrêmement fastidieuse. Nous allons nous en rendre compte tout de suite en examinant de plus près la machine MaP.



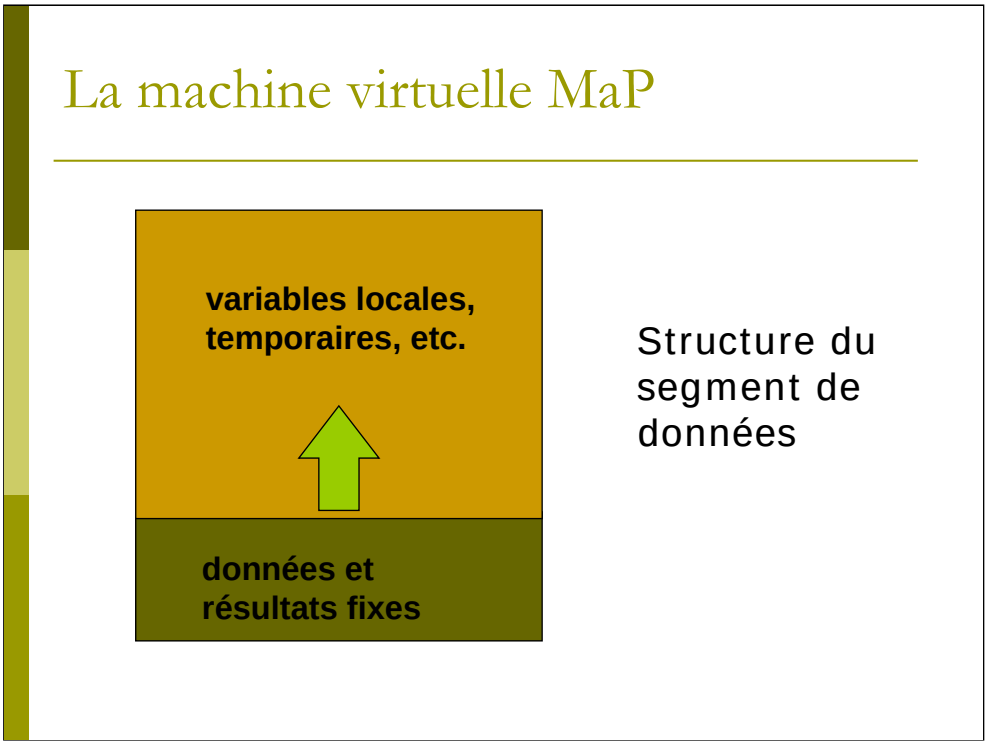
Le cœur de notre machine virtuelle MaP est un processeur orienté pile (stack machine) doté d'un compteur ordinal PC (Program Counter) pointant sur l'instruction à exécuter. Le processeur dispose encore de deux autres registres spéciaux. Le premier est le registre pointeur de pile. Il s'appelle SP (Stack Pointer) et pointe sur le sommet de la pile d'exécution. L'autre est le registre de base BP (Base Pointer) qui est un registre spécialisé permettant d'accéder à des mots mémoire du segment de données via un déplacement relatif au contenu de ce registre.

La mémoire centrale se compose de deux segments :

- le segment de code ;
- et le segment de données.

Ces deux segments sont séparés et ont chacun comme adresse de base 0. La fonction du segment de code est, comme son nom l'indique, de contenir le code objet MaP du programme à exécuter.

Le segment de données sert à contenir les variables globales, locales et temporaires du programme dont le code se trouve dans le segment de code.



Le programmeur MaP peut programmer la machine comme bon lui semble. C'est ainsi qu'il peut organiser le segment de données de beaucoup de manières différentes. En fait, dans les programmes que nous écrirons, le segment de données sera organisé de la manière suivante :

les variables dont la taille nécessaire est connue à la fin de la compilation ou de l'assemblage (variables globales), occupent l'espace mémoire du segment de code à partir de l'adresse 0.

Immédiatement après le dernier mot mémoire occupé par les variables globales débute la *pile d'exécution* qui peut s'étendre jusqu'à la fin du segment de données.

La pile d'exécution contiendra toutes les variables temporaires et l'espace mémoire utilisé croîtra et décroîtra au fur et à mesure de l'exécution du programme.

La taille d'un mot mémoire du segment de données est telle qu'il peut contenir une donnée de type caractère ou entier. Deux mots mémoire sont nécessaires à la représentation d'une valeur de type *numeric*, l'unique type du langage LERIA comme nous le verrons plus loin.

La taille d'un mot mémoire du segment de code est la moitié de la taille d'un mot mémoire du segment de données.

La machine virtuelle MaP

- Le jeu d'instructions de la machine virtuelle comprend 58 instructions que l'on peut regrouper en :
  - instructions arithmétiques et logiques ;
  - instructions de manipulation de la pile ;
  - instructions de rupture de séquence ;
  - instructions d'entrée-sortie ;
  - instructions spéciales.

Le processeur reconnaît 31 instructions de base parmi 59 instructions possibles car la plupart des instructions existent en deux ou trois versions suivant le type des données manipulées (*int*, *numeric*, *char*).

En effet, le processeur MaP est capable de traiter trois types différents de données :

les nombres entiers (*int*) ;

les nombres décimaux (*numeric*) ;

les caractères (*char*).

La machine virtuelle MaP

- les instructions arithmétiques et logiques sont :
  - ADD, SUB, MULT, DIV, NEG
  - AND, OR, NOT, EQ, GT, LS

Les instructions arithmétiques et logiques effectuent des opérations de calcul sur les mots mémoire situés au sommet de la pile d'exécution. Elles ne nécessitent pas d'opérande car ceux-ci se trouvent déjà sur la pile.

Ces instructions existent pour le type entier ainsi que pour le type numérique. Ainsi, par exemple, une opération d'addition de deux entiers se fait à l'aide de l'instruction ADDI, alors qu'une instruction d'addition de deux numériques se fait avec l'instruction ADDN. L'ensemble des instructions arithmétiques s'écrit donc : ADDI, SUBI, MULTI, DIVI, NEGI, ADDN, SUBN, MULTN, DIVN, NEGN.

Les instructions AND, OR et NOT ne sont pas typées. AND et OR fonctionnent sur les deux mots mémoire du sommet de la pile, NOT ne prend en compte que le mot mémoire du sommet de la pile.

La machine virtuelle MaP

- les instructions de manipulation de la pile sont :
  - DECR, INCR, DUPL, ASSGN, LOAD, PAIBP, PUSH, STORE

Ces instructions peuvent être classées en deux catérgories :

• les instructions de manipulation du pointeur de pile ;

• les instructions modifiant le contenu de la pile et par extension le contenu du segment de données.

Les instructions DECR et INCR permettent au programmeur de changer la valeur du registre SP. DUPL, CONT, LOAD, PAIBP, PUSH sont des instructions modifiant le sommet de la pile tandis que ASSGN et STORE permettent de modifier n'importe quel mot mémoire du segment de données.

La machine virtuelle MaP

- les instructions de rupture de séquence sont :
  - JP, JF, CALL, RET

Les instructions de rupture de séquence permettent d'effectuer des branchements vers d'autres instructions du segment de code en changeant la valeur du compteur ordinal.

Les instructions de la machine MaP sont exécutées par défaut en séquence, dans leur ordre d'apparition. A l'aide des instructions CALL, RET, JF et JP, le programmeur peut changer cet ordre afin de refléter la logique de traitement. Les instructions CALL et RET permettent de mettre en œuvre le concept de sous-programme. JP provoque un branchement inconidtionnel alors que JF effectue un branchement en fonction de la valeur présente sur le sommet de la pile.

La machine virtuelle MaP

- les instructions d'entrée-sortie sont :
  - IN, OUT, OUTSTR

Les instructions d'entrée-sortie permettent à la machine virtuelle de communiquer avec le monde extérieur.
Trois instructions de base, déclinées selon le type des données (IN, OUT) sont prévues à cet effet.

La machine virtuelle MaP

- les cinq instructions spéciales de MaP sont :
  - CALLS, HALT, NEWBP, RSTRBP

CALLS est une instruction permettant d'effectuer un « appel système ». L'instruction HALT arrête la machine MaP et rend le contrôle au système d'exploitation hôte.

NEWBP, SAVEBP et RSTRBP sont des instructions spéciales permettant de manipuler le registre BP.

La machine virtuelle MaP

- codes opératoires MaP (nombres décimaux)

addi 0	addn 1	and 2	assgnc 3
assgni 4	assgnn 5	call 6	calls 7
contc 8	conti 9	contn 10	decr 11
divi 12	divn 13	duplc 14	dupli 15
dupln 16	eqc 17	eqi 18	eqn 19
gtc 20	gti 21	gtn 22	halt 23
Inc 24	ini 25	inn 26	incr 27
jf 28	jp 29	loadc 30	loadi 31

Chaque instruction est représentée dans le segment de code à travers un nombre. Dans notre cas, ces codes ont été définis arbitrairement dans l'ordre croissant des mnémoniques des instructions machine.

Le lecteur se rend tout de suite compte que la programmation à l'aide de ces codes opératoires, quelque soit la base utilisée, n'est pas une chose aisée. Voici un exemple d'un programme exécutable LERIA effectuant la somme de deux nombres décimaux entrés au clavier et affichant cette somme.

33 46 0 42 42 42 32 69 114 114 101 117 114 32 100 39 101 120 -126
99 117 116 105 111 110 32 42 42 42 32 58 32 105 110 100 105 99
101 32 104 111 114 115 32 108 105 109 105 116 101 115 32 -123 32
108 97 32 108 105 103 110 101 32 0 0 51 0 27 0 0 0 4 0 46 0 69 110
116 114 101 122 32 100 101 117 120 32 110 111 109 98 114 101 115
32 58 32 0 0 49 0 0 0 0 0 26 0 49 0 0 0 2 0 26 0 46 0 76 101 117 114
32 115 111 109 109 101 32 118 97 117 116 32 58 32 0 0 49 0 0 0 0 0
10 0 49 0 0 0 2 0 10 0 1 0 45 0 23 0

Le premier nombre de ce code représente l'adresse de la première instruction à exécuter, ici 33. L'instruction réelle de départ est par conséquent 27, figurant l'instruction INCR, son argument étant 4.

La machine virtuelle MaP

- codes opératoires MaP (nombres décimaux)

loadn 32	lsc 33	lsi 34	lsn 35
multi 36	multn 37	negi 38	negn 39
newbp 40	not 41	or 42	outc 43
outi 44	outn 45	outstr 46	paibp 47
pushc 48	pushi 49	pushn 50	ret 51
rstrbp 52	storec 53	storei 54	storen 55
subi 56	subn 57		

Un entier est codé sur deux mots mémoire du segment de code et un numérique sur quatre. Comme le processeur MaP a reconnu l'insruction INCR, il sait qu'elle admet un argument entier et par conséquent va lire les quatre octets suivants 0 0 4 0. Les deux premiers octets sont le mot mémoire de poids fort (valeur 65536) et les deux suivants le mot mémoire de poids faible. Chaque mot mémoire est à son tour une paire (octet poids faible, octet poids fort).

Le lecteur se rend tout decompte de la complexité de réaliser des programmes en code décimal (ou binaire, etc.)

C'est pour cette raison, entre autres, que l'on a développé des programmes spéciaux appelés *assembleurs*.

Les langages de programmation

- l'assembleur
  - utilisation de mnémoniques
  - constantes symboliques
  - adresses symboliques
  - commentaires
  - …

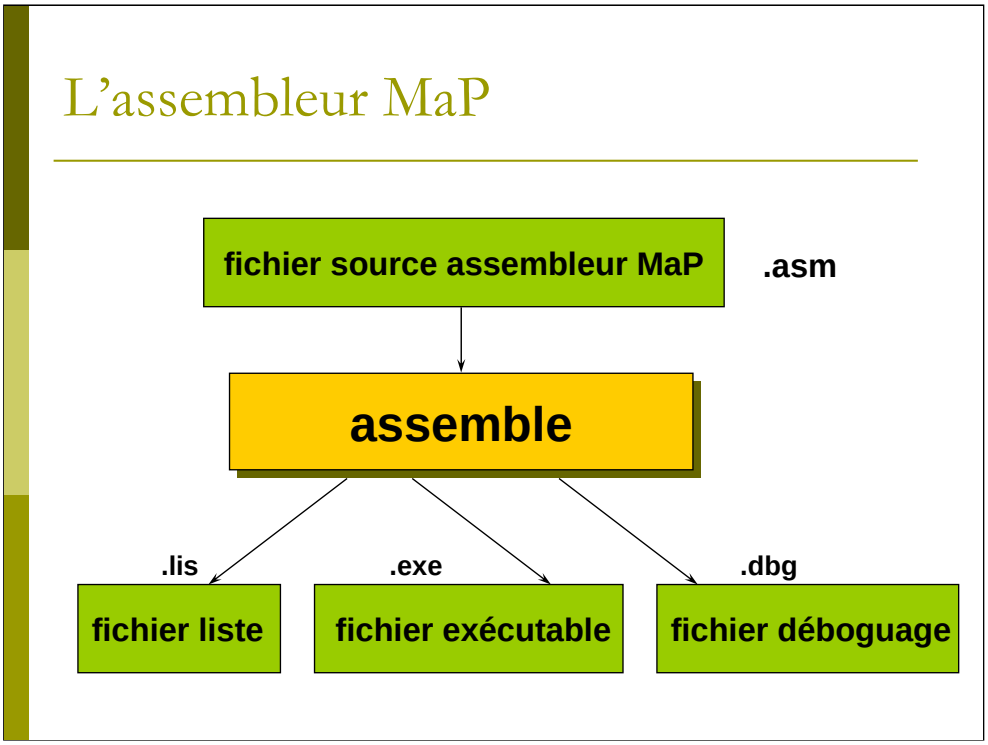
Un assembleur est un programme à même de traduire des mnémoniques et des adresses symboliques en binaire (ou autre).

Le grand avantage des assembleurs est qu'ils permettent d'exprimer de manière textuelle les instructions machine utilisées pour coder un programme. Ils représentent donc un progrès immense par rapport à des codes numériques.

Néanmoins, ils ont le grand désavantage de ne généralement pas être portables d'un type d'ordinateur à un autre puisqu'ils ne permettent que de programmer l'ordinateur sur la base d'un langage qui lui est propre !

Les assembleurs peuvent être plus ou moins évolués, les plus puissants étant les macro-assembleurs. Ceux-ci permettent au programmeur de définir des macros, c'est-à-dire des instructions nouvelles éventuellement paramétrisées, qui sont textuellement remplacées par les instructions définies par le programmeur.

Dans le cadre de ce cours, un assembleur pour le langage MaP est fourni, permettant de développer des programmes en langage machine.



L'assembleur MaP fourni à tous les participants (plate-forme DOS) examine un fichier source contenant du code assembleur MaP et créé avec un éditeur de textes. Le programme source porte par défaut l'extension « .asm ».

Cet assembleur s'appelle "assemble" et produit deux fichiers :

• un fichier liste d'extension « .lis » contenant le programme source original, numéroté avec les messages d'erreur éventuels. Il porte le même nom que le fichier « .asm » d'origine.

• un fichier exécutable d'extension « .exe » et contenant le code machine. Pour des raisons pédagogiques, ce n'est pas un fichier binaire, mais un fichier texte où les codes sont représentés par des nombres décimaux.

Le code machine d'un programme assembleur correct (fichier « .exe ») peut être exécuté avec la commande RUN. Cette commande est aussi fournie à tous les participants. Elle est en fait l'interprète pour le code de la machine virtuelle MaP.

Les langages de programmation

- exemple d'un programme assembleur Map :
  - calcul, puis affichage de la somme de deux nombres entrés au clavier de l'ordinateur.

```
{$d Ce programme calcule et affiche la somme de deux nombres entiers entrés au clavier }

n1        equ   0          ; adresse de n1
n2        equ   1          ; adresse de n2
somme     equ   2          ; adresse de somme

debut_    incr  3          ; réserver 3 mots mémoire pour n1, n2 et somme

          outstr "Entrez le 1er nombre "
          pushi  n1         ; placer l'adresse de n1 sur le sommet de la pile
          ini               ; la valeur lue au clavier est placée dans n1

          outstr "Entrez le 2eme nombre "
          pushi  n2         ; placer l'adresse de n2 sur le sommet de la pile
          ini               ; la valeur lue au clavier est plac,e dans n2

          pushi somme       ; placer l'adresse de la variable somme sur la pile
          pushi n1
          conti             ; la valeur de la variable n1 se trouve sur la pile
          pushi n2
          conti             ; la valeur de la variable n2 se trouve sur la pile
          addi              ; ces deux valeurs sont additionn,es
          storei 1          ; et sont placées dans la variable somme

          outstr "La somme vaut : "
          pushi somme
          conti
          outi              ; le contenu de la variable somme est affiché
          outstr  "\n"      ; passons … la ligne
          halt              ; facultatif

          end  debut_
```

Les langages de programmation

- une étiquette se termine obligatoirement par un « _ »
- en plus des instructions machine, il y a les pseudo-instructions
  - EQU et END
- un commentaire fin de ligne débute par un « ; », un commentaire multi-lignes est placé entre « { } »

Dans l'assembleur MaP fourni, les étiquettes sont des identificateurs (chaînes de caractères de taille maximale égale à 50 débutant par une lettre, suivie de caractères alphanumériques) et se terminant obligatoirement par un caractère « _ ». Chaque étiquette doit être unique et toute étiquette référencée doit être définie.

En plus des mnémoniques définies pour chacune des instructions MaP, il y a deux pseudo-instructions :

EQU permet des définir une constante symbolique ;

END indique l'adresse de la première instruction lors de l'exécution du programme. Cette pseudo-instruction provoque l'insertion de l'instruction HALT dans le programme exécutable.

Un programme MaP devrait être étoffé de commentaires. Les commentaires « fin de ligne » sont précédés du symbole « ; » alors que les commentaires multi-lignes sont placés entre « {} ».

La directive d'assemblage « {$D} » provoque l'inclusion du code généré désassemblé dans le fichier liste.

L'utilisation d'un assembleur est un progrès certain par rapport au codage en nombres décimaux ou binaires. Le gain de lisibilité obtenu est évident.

Les plus grands reproches que l'on peut faire à l'encontre de la programmation en assembleur est, d'une part, le fait que le programme réalisé ne peut généralement pas être porté sur un autre type d'ordinateur et de l'autre, que les concepts manipulés sont généralement très éloignés des problèmes pratiques que le programmeur doit résoudre.

Un seul argument qui parle vraiment en faveur de la programmation en assembleur est qu'il offre le moyen de tirer le maximum de la machine, du point de vue compacité et vitesse du code. Mais les situations dans lesquelles cet argument est important, sont très rares en réalité.

La conclusion est qu'il faut, au possible, développer ses programmes à l'aide d'un langage de programmation évolué et éviter le langage machine. Les compilateurs disponibles à l'heure actuelle sur le marché sont, pour la plupart, très performants et génèrent un code machine très efficace !

Les langages de programmation

- problème : quel langage choisir ?
  - FORTRAN
  - ALGOL
  - BASIC, COBOL
  - PASCAL, MODULA-2, C, C++
  - …

LERIA

Pour pallier aux désavantages des assembleurs, on a créé des langages de programmation. Ceux-ci ont l'avantage de permettre au programmeur d'exprimer son programme en un langage indépendant de l'ordinateur sur lequel le programme est exécuté. De plus, ils permettent de manipuler des concepts proches des problèmes à résoudre. Depuis la fin des années 1950, une multitude de langages a été créée :

• le langage FORTRAN (FORmula TRANslation) utilisé surtout pour les calculs scientifiques intensifs ;

• ALGOL (ALGOrithmic Language), premier langage à structure de blocs, ancêtre de la plupart des langages modernes ;

• le langage de programmation COBOL, (COmmon Business Oriented Language), c'est-à-dire «langage commun orienté gestion», a été finalisé en avril 1960. Il incorpore de manière standard toutes les formes de traitement de fichiers ;

• BASIC (Beginners All purpose Symbolic Instruction Code), langage simpliste, non structuré, dans sa version standard, peu portable ;

• PASCAL et son successeur MODULA-2 sont surtout utilisés dans le monde de l'enseignement ;

• C et C++ sont utilisés par les professionnels, malgré leurs défauts.

Dans ce cours d'introduction, nous utiliserons le langage LERIA (inspiré librement des langages PASCAL et C).

Exercices

- Tous les programmes doivent être réalisés avec l'assembleur MaP.
- N'oublier pas de bien les commenter. Cette opération se fait tout au long du développement d'un programme et non à la fin de ce processus !

1) Réaliser un programme qui effectue une transformation de miles en km. L'utilisateur entre le nombre de miles et le programme affiche l'équivalent en km (1 mile = 1609 m).

2) Réaliser un programme calculant la surface d'un disque sur base du rayon r entré par l'utilisateur (surface = πr^2).

3) Réaliser un programme calculant la somme S des n premiers entiers en utilisant la formule : $S = n(n+1)/2$

Exemple :

la somme des 5 premiers entiers vaut 1+2+3+4+5 = 5*6/2 = 15

4) Réaliser un programme calculant la somme des n premiers entiers sans utiliser la formule précédente.

5) Réaliser un programme calculant la factorielle d'un nombre entier positif ou nul. Exemple : calculons la factorielle de 4 que l'on note 4!

4! = 1 * 2 * 3 * 4 = 24

0! = 1 par définition

Le développement d'un programme

Le développement d'un programme

- Ecrire un programme qui effectue une transformation de miles en km. L'utilisateur entre le nombre de miles et le programme affiche l'équivalent en kilomètres (1 mile = 1609 m).

Pour résoudre un problème par des moyens informatiques, il n'y a pas de magie: il faut réaliser un programme qui traite les données qu'on lui fournit pour produire les résultats voulus.

Ceci signifie en clair que généralement nous savons quelles sont les données et quels sont les résultats à produire.

Ainsi pour résoudre un problème, il faut faire l'inventaire des données et des résultats.

Dans le cas de notre premier exemple, nous avons :

donnée : le nombre de miles

résultat : le nombre équivalent en kilomètres

Ces deux valeurs se trouveront quelque part en mémoire centrale de l'ordinateur. Dans un langage de haut niveau, l'emplacement exact et la représentation interne des valeurs ne nous préoccupent généralement pas. Pour manipuler ces valeurs, nous les désignons par des noms symboliques.

Le développement d'un programme

- donnée(s) : identificateur
- le nombre de miles miles

- résultat(s) : identificateur
- l'équivalent en km km

A l'intérieur d'un programme, toute valeur est donc généralement représentée par un identificateur. Le nom de l'identificateur est laissé au choix du programmeur.

Mais celui-ci a naturellement intérêt à choisir un nom proche de la désignation de la valeur que l'identificateur est censé représenter.

Les règles de construction d'un identificateur diffèrent d'un langage de programmation à un autre.

En LERIA, un identificateur est une chaîne de caractères parmi l'alphabet suivant : les 52 lettres (majuscules et minuscules) de A à Z, les 10 chiffres 0 à 9 et le caractère "blanc souligné" qui est "_". Le premier caractère de l'identificateur est obligatoirement une lettre et la taille d'un identificateur est limitée à 50 caractères. Les lettres majuscules ou minuscules peuvent être utilisées indifféremment ce qui n'est pas le cas du langage C par exemple.

Ainsi, en LERIA :

miLEs miles MILES

ne désignent qu'un seul identificateur, alors qu'en C, ces trois noms sont trois identificateurs différents.

Le développement d'un programme

- Les mots réservés :
- AND ARRAY BEGIN BY CONST
- DO ELSE END FOR
- FORWARD FUNCTION HALT
- IF NOT OF OR
- PROGRAM READ RECORD REPEAT
- RETURN SKIP THEN TO
- TYPE UNTIL VAR WHILE WRITE

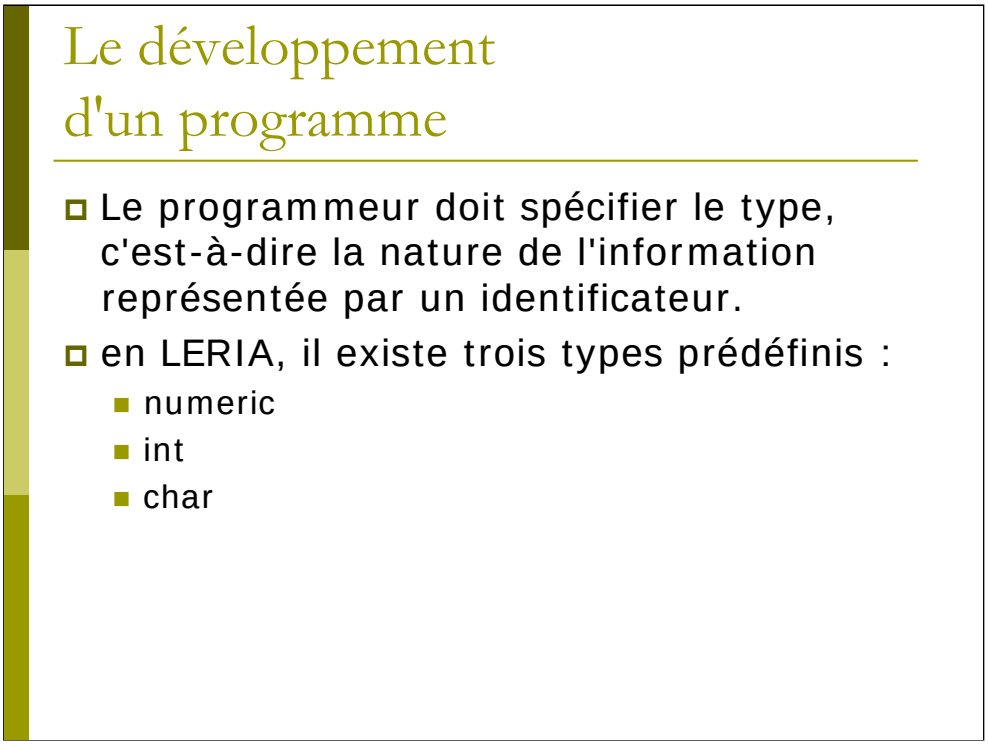
Cette liste représente les mots réservés du langage LERIA. Ils ont un sens prédéfini immuable et ne peuvent jamais être choisis par le programmeur comme identificateurs de données.

Le développement d'un programme

- Les identificateurs prédéfinis :

ABS	ACOS	ASIN	ATAN
CHR	COS	EXP	IMPAIR
INTNUM	LN	NUMINT	ORD
POWER	RAND	RANDOMIZE	
ROUND	SIN	SQR	SQRT
TAN	TIME	TRUNC	

Ces noms représentent des fonctions et types prédéfinis dans LERIA et il n'est pas permis en général de les choisir comme identificateurs de données. Néanmoins, comme nous le verrons plus tard, on pourra le faire dans le cas des champs d'un enregistrement. Ceci est quand même une pratique déconseillée.



Une fois que nous avons fixé un identificateur pour une valeur dans la mémoire de l'ordinateur, il nous reste à indiquer quel est le genre, le type de la valeur.

N'oublions pas, en effet, que la mémoire centrale d'un ordinateur se compose de mots mémoire identiques, quelque soit la nature de l'information que l'on y stocke. Donc à la seule vue du mot mémoire, l'ordinateur n'est pas à même de décider à quel type d'information il est confronté.

C'est au programmeur que revient la tâche d'indiquer la nature ou le type d'une valeur. Ainsi, pour tout identificateur qu'il définit, le programmeur doit indiquer au compilateur le type de la valeur représentée par cet identificateur.

Cela permet au compilateur de réserver assez d'espace mémoire pour la représentation de la valeur dans la mémoire. De plus, vu que le compilateur connaît alors le type de l'identificateur, il peut effectuer des contrôles stricts par la suite et détecter des anomalies où le programmeur tente par exemple de "comparer des pommes et des poires" !

Le développement d'un programme

- Une variable :
- un identificateur
- un type
- une valeur

En programmation impérative (C, Cobol...), les programmes arrivent à produire les résultats désirés en manipulant des variables.

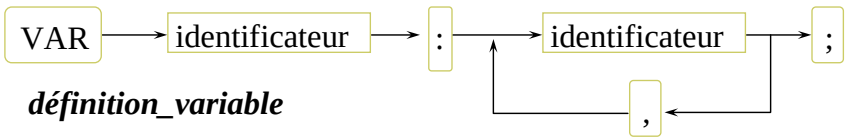
Une variable est un triplet (identificateur, type, valeur). En clair, une variable est désignée par un nom ou identificateur. D'après ce qui précède, un type est associé à l'identificateur en question. Comme cet identificateur désigne des mots dans la mémoire centrale de l'ordinateur, une variable a toujours une valeur.

La valeur est obtenue en interprétant les mots mémoire désignés par l'identificateur de la manière indiqué par le type de la variable.

Cette valeur peut être indéfinie si le programmeur utilise une variable sans que celle-ci n'ait été initialisée.

Le développement d'un programme

- une définition de variable en LERIA a la structure suivante :
- définition_variable ::= "VAR" identificateur ":" identificateur { "," identificateur } ";"



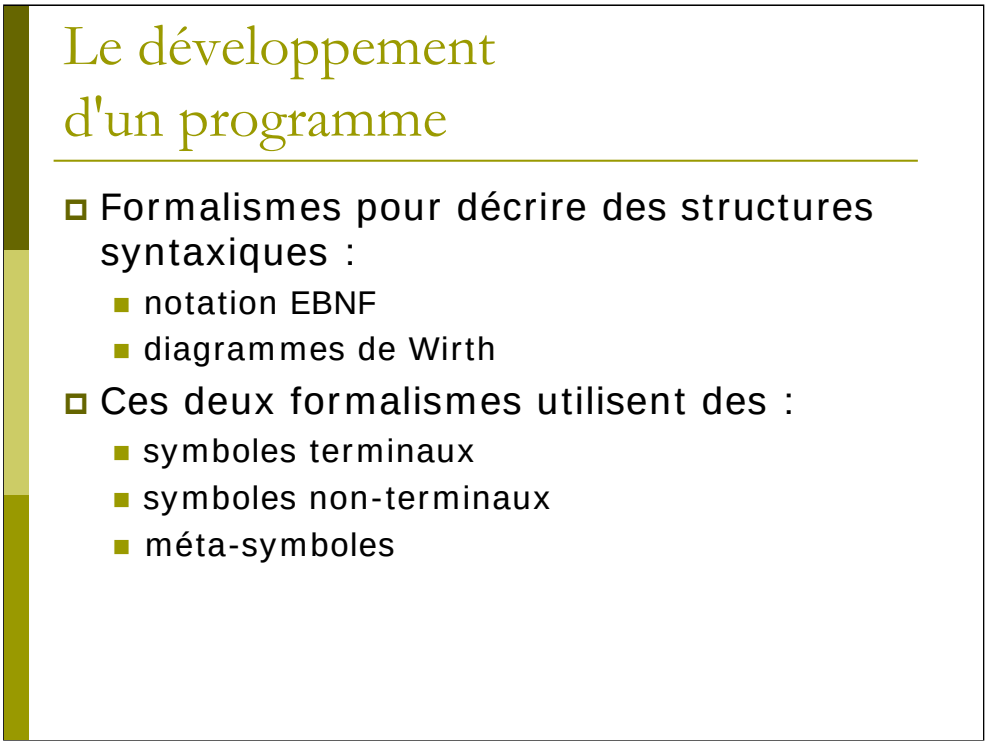
définition_variable

Le type prédéfini *numeric* permet au programmeur de définir des variables numériques.

Une déclaration de variable a la structure syntaxique indiquée ci-dessus et débute avec le mot réservé VAR.

Un mot réservé est un identificateur prédéfini dans le langage et le programmeur ne peut utiliser aucun mot réservé pour désigner un de ses propres identificateurs.

Une déclaration de variable(s) débute donc obligatoirement par le mot réservé VAR précédant un identificateur de type. Viennent ensuite le symbole ":" et tous les identificateurs de variable auxquels le programmeur veut associer le type précité. La déclaration se termine nécessairement par le symbole ";". Plusieurs identificateurs de variable sont séparés par des symboles ",".



La structure grammaticale d'une phrase d'un langage est usuellement représentée, soit sous forme EBNF(Extended Backus-Naur Form), soit à l'aide de diagrammes de Wirth.

Dans une règle grammaticale, on distingue entre les symboles terminaux (qui figurent effectivement dans des phrases concrètes) du langage, les symboles non-terminaux (pour lesquels il existe une autre règle grammaticale) et les méta-symboles (symboles nécessaires à la représentation d'une règle).

Dans la notation EBNF, les symboles terminaux sont placés entre " " alors que dans les diagrammes de Wirth, ils se trouvent dans des rectangles aux coins arrondis. En EBNF, les autres identificateurs désignent des symboles non-terminaux. Dans les diagrammes de Wirth, ceux-ci se retrouvent dans des rectangles normaux.

Le méta-symbole « ::= » représente le fait que le symbole non-terminal à sa gauche peut être réécrit par l'expression à sa droite. Une expression placée entre « { } » signifie que l'expression en question peut être répétée de 0 à autant de fois qu'on le désire. Une expression entre « [ ] » est facultative, c'est-à-dire qu'elle est présente une fois ou pas du tout.

Le méta-symbole « | » indique qu'on a le choix entre l'expression à sa gauche et à sa droite. Ces méta-symboles sont traduits graphiquement à l'aide de flèches dans les diagrammes de Wirth.

Le développement d'un programme

- dans notre exemple, les deux variables utilisées se définiront ainsi :
- VAR numeric : miles, km;
- nous aurions aussi pu écrire :
- VAR numeric : miles;
- VAR numeric : km;

Par convention, nous écrirons tous les mots réservés en majuscules à l'exception des types et fonctions prédéfinis, en l'occurrence ici *numeric*.

Le développement d'un programme

- l'affectation est l'instruction fondamentale dans les langages impératifs
- l'affectation permet de changer la valeur d'une variable
- format simplifié :
- affectation ::= identificateur ":=" expression

affectation

Dans les langages de programmation impératifs, l'instruction d'affectation est l'instruction fondamentale permettant au programmeur de donner aux variables les valeurs attendues.

L'affectation permet de donner explicitement une valeur à une variable. Lors de l'exécution de cette instruction, l'expression placée à droite du symbole ":=" est évaluée. Lorsque cette expression est évaluée, la valeur résultante est transférée dans les cellules mémoire désignées par l'identificateur de variable placé à gauche du symbole d'affectation ":=".

Remarque très importante : la valeur qu'avait la variable à gauche du symbole ":=" juste avant l'exécution de l'affectation est irrémédiablement perdue !

De ce fait, l'ordre dans lequel les instructions sont placées par le programmeur, est primordial.

Le développement d'un programme

- dans notre exemple :
- km := 1.609 * miles
- la variable miles, désignant le nombre de miles à convertir en km est multipliée par 1.609. Cette valeur est affectée ensuite à la variable km.

Le programmeur peut définir des expressions très complexes, comme nous le verrons en détail plus loin. En attendant, nous nous limitons à des expressions simples. Les opérateurs de base sont :

+	addition	
-	soustraction	
*	multiplication	
/	division	3/2 = 1.5
%	division entière	3%2=1
#	modulo	7#4 =3
^	puissance	2^3=8

Ces opérateurs binaires peuvent être combinés à volonté. Le programmeur doit néanmoins savoir que dans LERIA, une expression est évaluée de la droite vers la gauche et que seul un placement de parenthèses peut modifier l'ordre d'évaluation à l'intérieur d'une expression. Exemples :

2–3*5+1-5*7 est évalué comme 2–(3*(5+(1–(5*7)))) ce qui donne 89

2–(3*5)+1 est évalué comme 2–((3*5)+1) = –14

–2–3 est évalué comme –(2–3) = 1

Le développement d'un programme

- il existe deux instructions d'entrée/sortie dans LERIA de format :

- read_instruction ::= "READ" "(" variable { "," variable } ")"
- WRITE_instruction ::= "WRITE" "(" w_expr { "," w_expr } ")"
- w_expr ::= expression [":" expression [":" expression]]
  | "\n"
  | constante_alphanumérique

Pour faire communiquer son programme avec le monde extérieur, le programmeur dispose de deux instructions en LERIA :

• l'instruction READ permet de changer la valeur d'une variable en lui affectant une valeur entrée par l'utilisateur au clavier de l'ordinateur.

Si plusieurs variables figurent dans la liste de l'instruction READ, le programme demandera à l'utilisateur d'entrer successivement des valeurs au clavier, chaque valeur devant être terminée par un « Enter » ou « RETURN ».

• l'instruction WRITE provoque l'affichage à l'écran des valeurs des expressions figurant dans la liste entre parenthèses.

L'affichage du symbole spécial « \n » provoque le déplacement du curseur au début de la ligne suivante à l'écran.

Par défaut, l'affichage de valeurs numériques se fait sur 8 positions dont deux après le point décimal. Le programmeur peut influencer la présentation des résultats numériques en utilisant les expressions de formatage. La première expression de formatage représente le nombre minimal de caractères que le nombre doit occuper, la deuxième expression de formatage représente le nombre de chiffres après le point décimal. Ces expressions sont tronquées et prises en valeur absolue de signe pour obtenir des valeurs entières positives ou nulles.

Le développement d'un programme

- à l'aide de l'instruction READ, l'utilisateur peut entrer des valeurs ayant un type prédéfini qui seront affectées à des variables du programme
- l'emploi de constantes alphanumériques dans l'instruction WRITE permet au programmeur de faire afficher du texte à l'écran de l'ordinateur

Une constante alphanumérique est une chaîne de caractères composée de 0 à 255 caractères du jeu des caractères imprimables (définis par le réalisateur du compilateur), placée entre apostrophes.

Dans notre cas, le jeu de caractères imprimables est un sous-ensemble du jeu de caractères ASCII étendu (tous les caractères sauf ceux ayant un code compris entre 0 et 31 compris, de même que le caractère de code 127).

Si la chaîne de caractères doit contenir une apostrophe, alors celle-ci doit être doublée, c'est-à-dire suivie immédiatement par une autre apostrophe afin qu'il n'y ait pas de confusion avec l'apostrophe de fin de chaîne.

Voici trois exemples de chaînes de caractères permises :

"Coucou" "L'exemple avec blancs permis !" """"

Le dernier exemple (quatre apostrophes) représente une chaîne représentant un seul caractère, le caractère apostrophe.

Le développement d'un programme

- structure simplifiée d'un programme LERIA:
- programme_leria ::= "PROGRAM" identificateur ";" bloc_principal "."
- bloc_principal ::= { définition_variable } instruction_composée
- instruction_composée ::=
- "BEGIN" instruction ";" { instruction ";" } "END"
- instruction ::= affectation |
- read_instruction | write_instruction |
- instruction_composée |
- skip_instruction | halt_instruction

Maintenant que nous avons vu tous les ingrédients de base pour réaliser notre premier exemple, il ne nous reste qu'à mettre tous les éléments en forme dans un programme LERIA complet.

Tout programme LERIA débute par le mot réservé PROGRAM, suivi du nom du programme et d'un point-virgule. Cet en-tête est suivi par le bloc principal et le point terminal.

Les définitions de variables doivent précéder l'instruction composée principale. Une instruction composée contient au moins une instruction. terminée par le symbole "point-virgule".

Une instruction peut être une instruction composée. Ceci trouvera son utilité plus tard lorsque nous étudierons des instructions plus complexes.

L'instruction SKIP est l'instruction vide. Elle n'a pas d'effet !

L'instruction HALT arrête l'exécution du programme et redonne le contrôle au système d'exploitation hôte.

Le développement d'un programme

```
PROGRAM miles_vers_km;
VAR numeric : miles, km;
BEGIN
  { saisie du nombre de miles }
  WRITE("Entrez le nombre de miles : ");
  READ(miles);
  { calcul du nombre équivalent en kilomètres }
  km := 1.609 * miles;
  { affichage du résultat }
  WRITE(miles:5:1, " miles = ", km:6:1, " km", \n);
END.
```

Le programme porte le nom *miles_vers_km*. Ce nom est interne au programme et n'a pas de signification à l'extérieur de celui-ci. Il utilise deux variables numériques, à savoir *miles* et *km*. L'instruction composée (BEGIN...END) du programme se compose de quatre instructions.

La première instruction est WRITE qui affiche les arguments qu'on lui indique à l'écran ; en l'occurrence ici la chaîne de caractères "Entrez le nombre de miles : ". Cette instruction doit être exécutée avant l'instruction suivante READ qui attend que l'utilisateur entre une valeur numérique sur le clavier de l'ordinateur.

Cette valeur que l'utilisateur aura entrée sera affectée à la variable indiquée dans l'instruction READ (ici *miles*).

Vient ensuite l'instruction de calcul déterminant l'équivalent kilomètres *km* en multipliant le nombre de miles par 1.609. Notons que comme dans les systèmes anglo-saxons, la virgule est représentée par le point décimal.

La dernière instruction provoque l'affichage de la donnée est du résultat sur l'écran de l'ordinateur. A la fin, le curseur passe à la ligne suivante (effet du \n).

Le développement d'un programme

- à l'exécution, le programme précédent donne par exemple :

```
Exécution en cours...
Entrez le nombre de miles : 10
 10.0 miles =   16.1 km
Exécution terminée
```

Le nombre de miles est affiché sur 5 positions, dont une position derrière la virgule (point décimal).

Le nombre équivalent en kilomètres est affiché sur 6 positions dont une derrière le point décimal.

Le programme précédent contenait des commentaires.

Un commentaire débute par le symbole accolade ouverte « { » et se termine par le symbole accolade fermée « } ». Tous les caractères se trouvant entre ces deux symboles sont ignorés par le compilateur et servent uniquement au programmeur à documenter son programme. Les commentaires peuvent être imbriqués. Dans ce cas, il faut veiller à placer un nombre correct d'accolades ouvertes et fermées. Un commentaire peut s'étendre sur plusieurs lignes. Exemples :

{ Ceci est un commentaire }

{ Ceci est { un commentaire imbriqué } }

L'utilisation, même très intensive, de commentaires n'a aucune influence sur la vitesse d'exécution d'un programme. Par contre, un programme bien commenté est plus facile à comprendre et à maintenir par la suite.

Le développement d'un programme

- Un deuxième exemple :
  - j'ai dans ma poche un certain nombre de pièces de 1€ et de 2€ que j'entre au clavier de l'ordinateur. Celui-ci m'affiche alors le nombre de pièces que j'ai en tout et la valeur qu'ils représentent.

Procédons comme pour l'exemple précédant en énumérant les données dont nous disposons ainsi que les résultats que le programme doit produire.

valeur	*variable*
nombre de pièces de 1€	*nbre_1*
nombre de pièces de 2€	*nbre_2*
nombre total de pièces	*nbre_pieces*
valeur du total des pièces	*valeur_nbre_pieces*

Ceci nous fait donc *a priori* quatre variables. Quelles sont les relations liant les données aux résultats ?

• *nbre_pieces = nbre_1 + nbre_2*

• *valeur_nbre_pieces = 1 * nbre_1F + 2 * nbre_2*

Cette dernière relation peut être simplifée :

• *valeur_nbre_pieces = nbre_1 + 2 * nbre_2*

Le développement d'un programme

```
PROGRAM pieces;
VAR int : nbre_1,              { nombre de pièces de 1€ }
          nbre_2,              { nombre de pièces de 2€ }
          nbre_pieces,         { nombre de pièces total }
          valeur_nbre_pieces;  { valeur du nombre de pièces total }
BEGIN
  { saisie des données }
  WRITE("Entrez le nombre de pièces de 1€ : "); READ(nbre_1);
  WRITE("Entrez le nombre de pièces de 2€ : "); READ(nbre_2);
  { calculs }
  nbre_pieces := nbre_1 + nbre_2;
  valeur_nbre_pieces := nbre_1 + 2*nbre_2;
  { affichage du résultat }
  WRITE("Vous avez ", nbre_pieces:3:0, " pièces de monnaie en tout",\n);
  WRITE("Elles ont une valeur de ",valeur_nbre_pieces:5:0, "€",\n);
END.
```

Ce programme LERIA implante la méthode de réalisation vue précédemment.

Il est clair que ce programme, comme le précédent, se basent sur l'hypothèse d'école que les données entrées par l'utilisateur sont correctes. Si tel n'est pas le cas, le résultat n'est pas défini !

Exemple correct :

```
Exécution en cours...
Entrez le nombre de pièces de 1€ : 7
Entrez le nombre de pièces de 2€ : 4
Vous avez  11 pièces de monnaie en tout
Elles ont une valeur de    15€
Exécution terminée
```

Le développement d'un programme

- Ecrire un programme calculant la mensualité que l'on doit payer si on emprunte une somme d'argent s à un taux d'intérêt fixe i donné et et qu'on rembourse sur une période de n années.
- Le programme devra afficher en plus le montant total des intérêts que l'emprunteur aura payé après les n années.

Dans cet exercice de synthèse, nous utiliserons tous les concepts vus jusqu'à présent.

Cet exercice sera aussi l'occasion d'introduire le concept de variable d'aide.

Le développement d'un programme

- la mensualité se calcule en utilisant cette formule des mathématiques financières :

$$\textbf{mensualité} = \frac{[\, s \cdot (i / 100) \cdot (1 + i / 100)^n \,]}{[12 \cdot ((1 + i / 100)^n - 1)]}$$

- dans cette formule :

  i est le taux d'intérêt (p. ex. i = 8%)
  n est le nombre d'années
  s est le montant emprunté

Voici les données et les résultats que nous pouvons dégager de la lecture de l'énoncé du problème :

valeur	*variable*	*type*
taux d'intérêt	*i*	*numeric*
durée du prêt en années	*n*	*int*
somme empruntée	*s*	*int*
montant de la mensualité	*mensualite*	*int*
montant du total des intérêts	*total_interet*	*int*

Le programme a une structure classique :

- saisie des données
- calculs des résultats
- affichage des résultats

En ce qui concerne le calcul des résultats, *mensualite* doit être déterminé avant *total_interet*.

Le développement d'un programme

- dans la formule calculant la mensualité, nous remarquons que
  - l'expression (i / 100) intervient 3 fois
  - l'expression (1+(i / 100))n intervient 2 fois

Ne calculer chacune des ces expressions qu'une seule fois !

Il arrive souvent dans des calculs un peu plus compliqués que des sous-expressions communes apparaissent à différents endroits.

Dans ces cas, plutôt que de les calculer à chaque fois, il est plus judicieux de ne les calculer qu'une seule fois, et puis de réutiliser cette valeur aux autres endroits.

Si on adopte cette stratégie, il est évidemment nécessaire de créer des variables supplémentaires servant à mémoriser les valeurs des sous-expressions.

Ceci est une des raisons pour lesquelles le programmeur est amené à créer des variables, autres que celles pour contenir les données et les résultats.

Dans notre exemple de synthèse, nous utiliserons deux variables pour contenir des valeurs intermédiaires :

le taux *i* divisé par 100	*taux_i*
la valeur $(1+(i / 100))^n$	*temp*

Le développement d'un programme

```
PROGRAM emprunt;
VAR numeric : i,              { le taux d'intérêt }
              taux_i,         { taux d'intérêt divisé par 100 }
              mensualite,     {montant de la mensualité }
              total_interet,  { total des intérêts payés }
              temp;           { variable temporaire }
VAR int : s,                  { la somme empruntée }
          n;                  { la durée du prêt, en années}
BEGIN
  { saisie des données }
  WRITE("Entrez le montant de la somme à emprunter : ");
  READ(s);
  WRITE("Entrez la durée du prêt, en années : "); READ(n);
  WRITE("Entrez le taux d'intérêt (p. ex. 7 = 7%) : ");
  READ(i);
```

Le développement d'un programme

```
{ calculs }
   { calcul de la mensualité }
  taux_i := i / 100.;
  temp := (1. + taux_i) ^ intnum(n);
  mensualite := (intnum(s) * taux_i * temp) / (12. * (temp - 1.));

   { calcul du total des intérêts }
  total_interet := (mensualite * 12. * intnum(n)) - intnum(s);

  { affichage des résultats }
  WRITE("Mensualité : ",mensualite:6:2,\n);
  WRITE("Total des intérêts à payer : ",total_interet:6:2,\n);
END.
```

Afin de pouvoir effectuer des opérations combinant des int et des numeric, le programmeur dispose de deux fonctions prédéfinies :

numint(arg1) renvoie l'argument de type numeric transformé en int

intnum(arg1) renvoie l'argument de type int transformé en numeric

Le développement d'un programme

- A l'exécution, ce programme donne par exemple :

- Exécution en cours...
- Entrez le montant de la somme à emprunter : 4000000
- Entrez la durée du prêt, en années : 20
- Entrez le taux d'intérêt (p. ex. 7 = 7%) : 6
- Mensualité : 29061.52
- Total des intérêts à payer : 2974764.56

- Exécution terminée

Exercices

- Tous les programmes doivent être réalisés en langage LERIA.
- N'oublier pas de bien les commenter. Cette opération se fait tout au long du développement d'un programme et non à la fin de ce processus !

1) Réaliser un programme qui permute (échange) le contenu de deux variables. De combien de variables a-t-on besoin pour réaliser cette opération ?

2) Ecrire un programme qui convertit une température exprimée en degrés Celsius en degrés Fahrenheit.

La relation qui lie les deux s'écrit : °F = 1.8 · °C + 32

Exemple : 15°C équivalent à 1.8 ·15 + 32 = 59°F

3) Ecrire un programme qui, lorsqu'on entre un montant en francs, affiche le nombre de billets ou/et de pièces qu'il faut pour réaliser ce montant tout en maximisant les grosses coupures. Par exemple, avec des unités de 1€, 2€, 5€, 10€, 20€, 50€, 100€, 200€ et 500€, pour un montant de 2346€, la réponse doit être 4 fois 500, 1 fois 200, 1 fois 100, 2 fois 20, 1fois 5, 1 fois 1€.

4) Refaire l'exercice de synthèse sans utiliser les variables d'aide. Comparer les sources assembleur des deux versions. Qu'est-ce que vous en dites ?

5) Faire les diagrammes de Wirth pour toutes les règles grammaticales du langage vues jusqu'à présent.

La structure alternative

La structure alternative

- Ecrire un programme qui affiche le salaire imposable d'un salarié. Le salaire brut, ainsi que le statut du salarié sont à entrer au clavier de l'ordinateur.

	assurance pension	*assurance maladie*
ouvrier	8%	4,15%
employé	8%	2,4%

Les exemples réalisés jusqu'à présent avaient la caractéristique commune que *toutes* les instructions de chacun des programmes étaient exécutées *séquentiellement* dans leur ordre d'apparition.

Dans la vie réelle, il existe évidemment une infinité d'exemples où le traitement à effectuer est variable, selon la nature des données.

Les langages de programmation impératifs donnent au programmeur, à travers la structure de contrôle alternative, la possibilité de procéder à des traitements spécifiques.

Tel est évidemment le cas du langage LERIA.

Dans l'exemple simple que nous nous proposons d'étudier en détail, il s'agit de calculer le salaire imposable d'un salarié en se basant sur son salaire brut et son statut. Pour simplifier, nous n'envisageons que deux statuts : ouvrier et employé.

Ce qui est intéressant dans cet exemple, c'est bien sûr que le salaire imposable est différent selon le statut du salarié.

La structure alternative

- Structure de la solution :
  - saisie salaire_brut et statut
  - calculer salaire_imposable
  - afficher salaire_imposable
- Pour raffiner la deuxième étape, il faut faire une distinction de cas selon le statut du salarié !

Ce programme utilise trois variables dont les identificateurs spécifient clairement les valeurs qu'elles sont censées représenter : *salaire_brut*, *statut* et *salaire_imposable*.

La saisie et l'affichage ne présentent aucune difficulté. Le seul point plus délicat est le calcul du salaire imposable. Il est clair que ce calcul dépend de la valeur du statut du salarié. Il faut donc procéder à une distinction de cas.

La structure alternative

- En LERIA, le format de l'instruction alternative s'écrit :

- if_instruction ::= "IF" expression
  "THEN" instruction
  ["ELSE" instruction]

if_instruction ::= "IF" expression "THEN" instruction ["ELSE" instruction]

Dans l'instruction conditionnelle, si l'évaluation de l'expression donne un résultat différent de 0, alors c'est l'instruction suivant le mot réservé THEN qui est exécutée. La partie ELSE, même si elle est présente, ne sera pas exécutée.

Si l'évaluation de l'expression donne la valeur 0, alors c'est l'instruction qui suit le mot réservé ELSE qui est exécutée, mais uniquement dans le cas où la partie ELSE existe. Dans le cas où il n'y a pas de partie ELSE, l'exécution continue en séquence avec l'instruction qui suit logiquement l'instruction IF.

Lorsque des instructions IF sont imbriquées, la partie ELSE se rapporte toujours au IF THEN immédiatement précédent.

Remarquons que seule *une* instruction peut suivre le THEN ou le ELSE. Ceci explique la nécessité d'utiliser une instruction composée dans ces cas de figure.

La structure alternative

```
PROGRAM calcul_salaire_imposable;
CONST  taux_pension_ouvrier = 0.08;
CONST  taux_pension_employe = 0.08;
CONST  taux_maladie_ouvrier = 0.0415;
CONST  taux_maladie_employe = 0.024;

VAR char : statut;
VAR numeric : salaire_brut, salaire_imposable;

BEGIN
  { saisie des données }
  write("Entrez le salaire brut : "); read(salaire_brut);
  write("Entrez le statut (employé, = E, ouvrier = O) : ");
  read(statut);
```

Le programme présenté ci-dessus utilise l'instruction conditionnelle d'après la syntaxe vue précédemment.

Mais cet exemple illustre un autre concept très important, la *constante*.

Le programmeur peut définir des constantes dans la partie de définitions de constantes. Cette section de définitions doit être la première du bloc avant les variables. Sa structure est la suivante :

partie_définition_constantes ::= { définition_constante }

définition_constante ::= "CONST" identificateur "=" "[" – "]" const_numérique | const_caractère ";"

const_numérique ::= littéral_numérique | identificateur

const_caractère ::= littéral_caractère | identificateur

Une constante définit un nom symbolique qui sert à identifier une valeur numérique éventuellement signée. Une définition de constante ne peut être récursive. Le nom d'une constante peut être utilisé par la suite, dans le domaine de définition de l'identificateur de constante, en lieu et place de la valeur numérique. La valeur numérique attachée au nom symbolique ne peut plus être modifiée dans le programme. Exemples :

```
CONST nbre_colonnes = 80;
CONST nbre_lignes = 25;
CONST bidon = nbre_colonnes;
nbre_lignes := 24;  { erreur à la compilation }
```

La structure alternative

```
{ calcul du salaire imposable }

IF (statut = 'E') OR (statut = 'e')
   THEN salaire_imposable :=
            salaire_brut * ((1. - taux_pension_employe)
                                  - taux_maladie_employe)
   ELSE salaire_imposable :=
            salaire_brut * ((1. - taux_pension_ouvrier)
                                  - taux_maladie_ouvrier);
  { affichage du résultat }
  write("Salaire imposable : ",salaire_imposable:8:0,\n);
END.
```

La structure alternative

- Avantages du concept de constante :
  - lisibilité accrue des sources
  - maintenabilité des programmes plus simple
  - sûreté d'emploi

Les avantages du concept de constante sont multiples :

• le programme est plus lisible. Au lieu de contenir des valeurs obscures, il contient un texte en clair décrivant au mieux la valeur ;

• le programme est plus facilement maintenable car il suffit en général de modifier la valeur de la constante en un endroit et puis de recompiler le programme. Dans le cas contraire, le programmeur doit vérifier ligne par ligne son code source et corriger les valeurs manuellement, ce qui n'est pas sans risque d'erreur ;

• le fait qu'une valeur soit définie comme une constante interdit au programmeur de changer accidentellement cette valeur. En effet, bien qu'une constante ait un identificateur tout comme une variable, le compilateur ne permet pas de l'utiliser comme une variable et affiche un message d'erreur le cas échéant.

La structure alternative

- Pour exprimer des conditions logiques complexes, le programmeur dispose des connecteurs logiques AND, OR et NOT.
- Il dispose aussi des opérateurs relationnels
  <, <=, >, >=, = et <>.
- Dans tous le cas en LERIA, une expression est évaluée dans son entièreté.

LERIA définit les opérateurs logiques AND, OR et NOT et les opérateurs relationnels <, <=, >, >=, <>, =. L'évaluation d'un opérateur relationnel donne comme résultat l'entier 1 (vrai) si la relation est vérifiée et l'entier 0 dans le cas contraire. L'opérateur AND est évalué à 1 (vrai) si ses deux opérandes de type entier ont comme valeur 1 (vrai), et à 0 (faux) dans les cas contraires. L'opérateur OR est évalué à 1 (vrai) si un des ses deux opérandes de type entier au moins a comme valeur 1 (vrai), et à 0 dans le cas contraire.

Exemples :

5 < 3 est évalué à 0 (faux)

5 > 3 est évalué à 1 (vrai)

(5 > 3) AND (4 > 3) est évalué à 1 (vrai)

(3 > 2) OR (1 > 2) est évalué à 1 (vrai)

(3 > 2) AND (1 > 2) est évalué à 0 (faux)

L'opérateur NOT appliqué à une expression entière donne 1 si la valeur de l'expression vaut 0 et 1 dans tous les autres cas. Exemples :

NOT 3 est évalué à 0 (faux)

NOT 0 est évalué à 1 (vrai)

Ainsi, dans l'évaluation d'une expression, toute valeur différente de 0 est considérée comme *vrai* logiquement, la valeur 0 est considérée comme *faux*.

La structure alternative

Exemple d'instructions IF imbriquées :

```
IF (statut = 'E') OR (statut = 'e')
   THEN salaire_imposable := salaire_brut * ((1. - taux_pension_employe)
                                              - taux_maladie_employe)
   ELSE
     IF (statut = 'O') OR (statut = 'o')
       THEN salaire_imposable := salaire_brut * ((1. - taux_pension_ouvrier)
                                                  - taux_maladie_ouvrier)
       ELSE BEGIN
         write("Statut erroné !",\n);
         halt;
       END;
```

Le programmeur débutant n'aime généralement pas utiliser des instructions IF imbriquées, mais préfère les placer les unes à la suite des autres en pensant que cela aura le même effet.

Cela est faux en général comme le montre l'exemple suivant.

La structure alternative

- Version 1 :
- IF x < 10
- THEN WRITE("haha",\n)
- ELSE IF x < 20 THEN WRITE("hihi",\n);

- Version 2 :
- IF x < 10 THEN WRITE("haha",\n);
- IF x < 20 THEN WRITE("hihi",\n);

A l'exécution on obtient pour :

	version 1	version 2
x = 30	*rien*	*rien*
x = 15	hihi	hihi
x = 5	haha	haha
		hihi

Nous remarquons que, pour une valeur inférieure à 10, le comportement des deux extraits de programme n'est pas le même. Cela est dû au fait que, dans la version 1, la partie ELSE n'est exécutée que si la première condition est évaluée à faux, c'est-à-dire si $x \geq 10$.

La condition d'entrée dans la deuxième instruction IF est donc implicitement $(x \geq 10)$ AND $(x < 20)$. Pour que les deux extraits de programme soient équivalents, il faut donc modifier en conséquence la condition du deuxième IF.

La structure alternative

- Version 1 :
- IF x < 10
- THEN WRITE("haha",\n)
- ELSE IF x < 20 THEN WRITE("hihi",\n);

- Version 2 modifiée :
- IF x < 10 THEN WRITE("haha",\n);
- IF (x >= 10) AND (x < 20)
  THEN WRITE("hihi",\n);

Mais, même sous cette forme, la version 2 modifiée n'est pas encore équivalente la version 1 car *x* est testé deux fois sur la valeur de 10 alors que dans le programme utilisant des IF imbriqués, *x* n'est testé qu'une seule fois sur la valeur 10 !

Dans les deux exercices qui suivent, nous allons nous familiariser davantage avec la structure alternative et montrer le genre de problèmes que l'on arrive à résoudre à l'aide de cette instruction.

La structure alternative

- Attention :
- BEGIN
- READ(x);
- IF x > 3
- THEN
- IF x > 5 THEN WRITE("Ok",\n)
- ELSE WRITE("Ko",\n);
- END.

A l'exécution :

x	4	7	2
écran	Ko	Ok	

ERREUR ?

Lors de la programmation d'instructions IF imbriquées, il est important de bien placer la clause ELSE par rapport au THEN correspondant sous peine d'induire le lecteur en erreur comme dans l'exemple ci-dessus.

Dans cet exemple, la clause ELSE est placée sous le premier THEN faisant croire que si *x* est inférieur ou égal à 3, le message *Ko* est affiché. En fait, il n'en est rien ! En effet, un ELSE se rapporte toujours au THEN immédiatement précédent, ce qui fait que le message *Ko* n'est affiché que dans le cas où *x* est supérieur à 3 et inférieur ou égal à 5.

L'extrait de programme devrait donc s'écrire de manière non ambigue comme ci-dessous :

```
BEGIN
  READ(x);
  IF x > 3
     THEN
        IF x > 5 THEN WRITE("Ok",\n)
                 ELSE WRITE("Ko",\n);
END.
```

La structure alternative

- Exercice: calcul de la date du lendemain
  - Il s'agit d'écrire un programme qui calcule la date du lendemain à partir d'une date correcte entrée au clavier.
  - Ce problème d'apparence simple n'est pas si anodin que cela, car il montre une des difficultés de la programmation : la précision des concepts manipulés par le programme.

Description des données et résultats :

année entrée	*a*
mois entré	*m*
jour entré	*j*
année du lendemain	*al*
mois du lendemain	*ml*
jour du lendemain	*jl*

Méthode de résolution :

1. saisie de la date entrée par l'utilisateur
2. calcul de la date du lendemain
3. affichage de la date du lendemain

La structure alternative

```
si pas fin de mois
    alors { passer simplement au jour suivant }
        jl ← j + 1      ml ← m      al ← a
    sinon { c'est la fin d'un mois. Est-ce la fin d'une année ? }
        si pas fin d'année
            alors { c'est une fin de mois normale }
                jl ← 1  ml ← m + 1   al ← a
            sinon { c'est bien la St. Sylvestre }
                jl ← 1  ml ← 1       al ← a + 1
        finsi
finsi
```

Il ne reste qu'à formaliser les différentes conditions. La condition fin de mois peut être écrite ainsi :

$j = 31$

ou $j = 30$ et $m = 4, 6, 9$ ou 11

ou $j = 29$ et $m = 2$

ou $j = 28$ et $m = 2$ et ce n'est pas une année bissextile

Remarquons encore une fois que nous avons supposé que la date entrée est correcte.

La condition année bissextile peut, quant à elle, être formalisée de la façon suivante :

((a divisible par 4) et (a pas divisible par 100)) ou (a divisible par 400)

La condition fin d'année s'écrit : fin de mois et $m = 12$

```
PROGRAM date_du_lendemain; {$d}

VAR
  int : a,    { année de la date actuelle }
        m,    { mois de la date actuelle }
        j,    { jour de la date actuelle }
        al,   { année de la date du lendemain }
        ml,   { mois de la date du lendemain }
        jl;   { jour de la date du lendemain }

VAR
  int : quotient,                    { variable temporaire }
        annee_divisible_par_4,       { variables booléennes }
        annee_divisible_par_100,
        annee_divisible_par_400;
BEGIN
  { saisie de la date actuelle }
  write("Entrez la date sous la forme JJ MM AAAA s,par,s par des
<Return>",\n);
  read(j); read(m); read(a);

  { année bissextile ? }
  quotient := a % 4;
  annee_divisible_par_4 := NOT(a - (quotient * 4));     { si 0 alors VRAI }
  quotient := a % 100;
  annee_divisible_par_100 := NOT(a - (quotient * 100));
  quotient := a % 400;
  annee_divisible_par_400 := NOT(a - (quotient * 400));
```

```
{ calcul de la date du lendemain }
  IF NOT ((j = 31) OR
          ((j = 30) AND ((m = 4) OR (m = 6) OR (m = 9) OR (m = 11))) OR
          ((j = 29) AND (m = 2)) OR
          ((j = 28) AND (m = 2) AND
                    NOT ((annee_divisible_par_4 AND
                          NOT annee_divisible_par_100) OR
                          annee_divisible_par_400)
          )
         )
    THEN BEGIN { on n'est pas à la fin d'un mois }
      jl := j + 1;
      ml := m;
      al := a;
    END
    ELSE IF m <> 12 { fin de mois mais pas fin d'année }
    THEN BEGIN
      al := a;
      ml := m + 1;
      jl := 1;
    END
    ELSE BEGIN { c'est la St. Sylvestre }
      al := a + 1 ;
      ml := 1;
      jl := 1;
    END;
  { affichage du résultat }
  write("La date du lendemain est : ",jl:2:0, " ",ml:2:0," ",al:4:0,\n);
END.
```

La structure alternative

- Exercice de synthèse : tri de trois nombres
  - On entre trois nombres au clavier. Il faut écrire un programme qui affiche ces trois nombres en ordre ascendant.
- Description des données :
  - les trois nombres entrés n1, n2, n3
- Description des résultats :
  - les mêmes trois nombres, mais triés en ordre ascendant avec n1 <= n2 <= n3

Méthode de résolution :

1. saisir les trois nombres *n1*, *n2*, *n3*
2. trier les trois nombres *n1*, *n2*, *n3*
3. afficher les trois nombres *n1*, *n2*, *n3* dans l'ordre trié

Raffiner 2 :

Pour trier trois variables, on peut utiliser plusieurs méthodes. Une méthode consiste à :

2.1. trier les deux premières variables pour obtenir $n1 \leq n2$

2.2. trier la troisième variable par rapport aux deux premières déjà triées.

Raffiner 2.1 :

si $n1 > n2$ alors permuter le contenu de *n1* et *n2*

Cette opération de permutation a déjà été vue (exercice 1 de la leçon précédente). Elle s'obtient à l'aide de trois opérations d'affectation et d'une variable temporaire appelée ici *tmp*

($tmp \leftarrow n1$, $n1 \leftarrow n2$, $n2 \leftarrow tmp$).

Raffiner 2.2 :

Il faut envisager trois cas, car comme les deux premières variables sont triées, la troisième ne peut être que plus grande que les deux premières, comprise entre les deux premières ou être plus petite que les deux premières. On obtient le tableau suivant :

condition	action	exemple		
		n1	*n2*	*n3*
n3 ≥ *n2*	ne rien faire	2	4	6
n3 < *n2*		2	4	3
et	permuter *n3* et *n2*	2	3	4
n3 ≥ *n1*				
		2	4	1
n3 < *n1*	permuter *n3* et *n1*	1	4	2
	permuter *n3* et *n2*	1	2	4

Le programme LERIA qui en découle est immédiat :

```
PROGRAM tri_trois_nombres;
VAR
  int : n1, n2, n3,  { les trois nombres à trier }
        tmp;         { variable temporaire nécessaire pour l'échange }
BEGIN
  { saisie des trois nombres à trier }
  WRITE("Entrez trois entiers : ",\n);
  READ(n1, n2, n3);

  { tri }
  IF n1 > n2
     THEN BEGIN
       tmp := n1; n1 := n2; n2 := tmp;
     END;
  IF n3 < n1
     THEN BEGIN
       tmp := n3; n3 := n1; n1 := tmp;
       tmp := n3; n3 := n2; n2 := tmp;
     END ELSE
       IF (n3 < n2) AND (n3 >= n1)
          THEN BEGIN
            tmp := n3; n3 := n2; n2 := tmp;
          END;
  { affichage des trois nombres triés }
  WRITE("Voici les trois nombres triés : ",n1," ",n2," ",n3,\n);
END.
```

Exercices

- Tous les programmes doivent être réalisés en langage LERIA.
- N'oublier pas de bien les commenter. Cette opération se fait tout au long du développement d'un programme et non à la fin de ce processus !

1) On entre trois nombres entiers différents au clavier, l'ordre est quelconque. Afficher à l'écran le nombre compris entre les deux autres. Il ne faut pas trier les nombres pour arriver au résultat.

2) L'exercice 3 du chapitre précédent doit être modifié de façon que, dans l'affichage, les lignes dans lesquelles le nombre de coupures est zéro soient supprimées. Les mots « billet » et « pièce » doivent être affichés correctement au pluriel ou au singulier.

3) On entre les cœfficients *a*, *b* et *c* de l'équation $ax^2 + bx + c = 0$.

Ecrire un programme qui résout cette équation. Pour calculer la racine carrée d'un nombre, il suffit de savoir que $\sqrt{x} = x^{0.5}$ ou utiliser la fonction prédéfinie *sqrt*().

La structure répétitive

La structure répétitive

- Pour introduire cette structure de contrôle, nous réalisons un programme qui calcule la somme des *n* premiers nombres entiers positifs, où *n* entier positif ou nul est entré au clavier de l'ordinateur.
- En clair, il s'agit de calculer la somme : 1+ 2 + 3 + ... + *n*, mais en supposant que nous n'ayons pas la formule pour la calculer directement.

Description des données et des résultats :

entier positif ou nul entré au clavier	*n*
somme des entiers	*somme*

Méthode de résolution :

1. entrer *n*
2. calculer *somme*
3. afficher *n* et *somme*

Raffiner 2 :

Pour obtenir la somme, il suffit d'ajouter successivement le nombre suivant *i* à la somme déjà calculée. Initialement *somme* vaut 0 et *i* vaut 1. Le nombre entier suivant se trouve dans la variable *i*. Cette répétition s'arrête lorsque *i* > *n*.

somme ← 0
i ← 1
] initialisations

tant que *i* ≤ *n* faire
    somme ← *somme* + *i*
    i ← *i* + 1
] corps de la boucle
fintantque

La structure répétitive

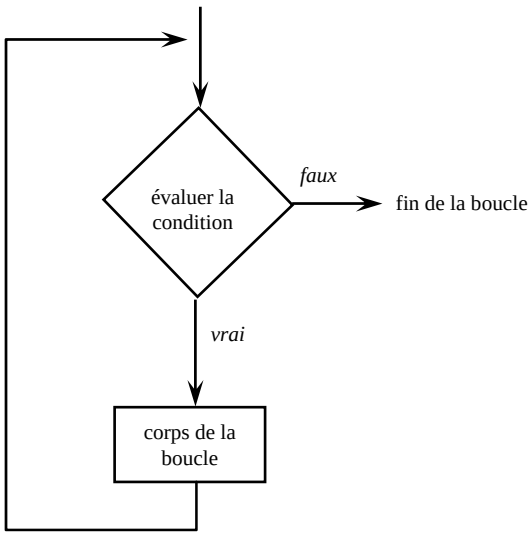
- La structure de contrôle de répétition s'écrit:

```
tant que condition de maintien faire
   corps de la boucle
fintantque
```

La sémantique de la structure de contrôle « tant que … fintantque » est celle-ci :

la condition de maintien dans la boucle est d'abord évaluée. Si elle est vérifiée, c'est-à-dire si sa valeur est *vrai*, le corps du « tant que … fintantque » est exécuté. Après exécution de la dernière instruction de ce corps, l'exécution continue avec l'évaluation de la condition de maintien, d'où l'effet de boucle ou répétition.

La boucle ou répétitive s'arrête lorsque la condition d'arrêt de la boucle est vérifiée. La condition qui apparaît dans le « tant que … fintantque » est une condition de maintien dans la boucle. La condition d'arrêt de la boucle s'obtient donc en niant la condition de maintien ou, inversement, la condition de maintien s'obtient en niant la condition d'arrêt.



La structure répétitive

- Attention :
  - à partir de maintenant, il est possible d'écrire des programmes pour lesquels l'exécution ne s'arrête jamais (sauf intervention externe) !

Comme le corps d'une boucle est exécuté jusqu'à la réalisation de la condition d'arrêt, il est primordial, lorsque l'on écrit une boucle, de s'assurer que le corps de la boucle contienne des instructions qui garantissent effectivement la possibilité de la réalisation de la condition d'arrêt. Si tel n'était pas le cas, alors le programme contenant la boucle ne s'arrêterait jamais de lui-même vu que la condition d'arrêt ne serait jamais réalisée. Lorsque l'on est dans un cas pareil, on dit que le programme « boucle » ou qu'il est dans une boucle infinie. Voici un exemple simple de boucle infinie :

$i \leftarrow 1$

tant que $i < 10$ faire

    afficher('Coucou')

fintantque

Vu que *i* vaut 1 initialement, le corps de la boucle est exécuté. Ensuite la condition est réévaluée ; comme *i* n'a pas changé, la condition donne de nouveau *vrai* et ainsi de suite. Le corps de cette boucle ne contient pas d'instruction pouvant modifier un des opérandes de la condition d'arrêt. Si on programme cet algorithme et qu'on l'exécute sur machine, il faut généralement une intervention externe de l'utilisateur pour l'arrêter, au pis en redémarrant la machine (sur un système simple).

La structure répétitive

- Format LERIA :

while_instruction ::= "WHILE" expression "DO" instruction

Tant que l'expression est évaluée à une valeur différente de 0 (*vrai*), l'instruction qui suit le DO est exécutée (cette instruction peut évidemment être une instruction composée).

Si l'expression est évaluée à 0 (*faux*), la *while_instruction* est terminée et l'exécution continue dans le programme avec l'instruction suivant logiquement l'instruction *while*. Ainsi, si à la première évaluation de l'expression, celle-ci a comme valeur 0, l'instruction suivant le DO n'est jamais exécutée.

La structure répétitive

```
PROGRAM somme_repetitive;
VAR int : i,n,somme;
BEGIN
  { saisie de n }
  WRITE("Entrez la valeur de N : "); READ(n);
  { calcul de la somme }
  i := 1; somme := 0;
  WHILE i <= n DO BEGIN
    somme := somme + i;
    i := i + 1;
  END;
  { affichage du résultat }
  WRITE("La somme des ",n, " premiers entiers vaut : ", somme,\n);
END.
```

Remarquons que nous écrirons toujours le corps d'une boucle légèrement décalé vers la droite afin de bien montrer la portée de la structure de contrôle.

Il reste une remarque à faire sur l'obtention de la condition d'arrêt à partir de la condition de maintien ou vice versa. Comme nous l'avons vu plus haut, une condition s'obtient en niant l'autre. Pour ce faire, on peut bien entendu utiliser l'opérateur logique NOT.

Mais on peut aussi utiliser les lois de « de Morgan » qui s'écrivent :

non(A ou B) = non(A) et non(B)

non(A et B) = non(A) ou non(B)

Supposons, par exemple, que la condition d'arrêt d'une boucle s'écrive :

$(x \leq 3)$ ou $(x \geq 5)$

La condition de maintien dans la boucle est donc :

non$[(x \leq 3)$ ou $(x \geq 5)]$ = non$(x \leq 3)$ et non$(x \geq 5)$ = $(x > 3)$ et $(x < 5)$

La structure répétitive

- Calculer la somme d'une suite de nombres positifs ou nuls entrés au clavier de l'ordinateur.
- La fin de la suite est marquée par un nombre négatif.

Le problème précédent était un peu spécifique en ce sens que la condition d'arrêt de la boucle ne dépendait que d'un paramètre fixé une fois pour toutes avant l'exécution de la boucle, lors de la saisie du nombre entier.

On peut aussi être confronté à la situation où le facteur déterminant la condition d'arrêt de la boucle est lui-même déterminé dans le corps de la boucle.

Nous allons étudier ce cas sur un petit programme qui calcule la somme d'une suite de nombres positifs ou nuls entrés au clavier de l'ordinateur. La fin de la suite est marquée par un nombre négatif.

Pour résoudre ce problème, il faut additionner la dernière valeur lue à la somme des valeurs déjà traitées. L'historique de ces valeurs ne nous intéresse pas et il suffit d'une variable contenant la dernière valeur lue et non encore traitée.

Description des données et des résultats :

dernière valeur lue et non encore traitée	*nbre*
somme des valeurs traitées	*somme*

Méthode de résolution :

Il est clair qu'il faut utiliser une boucle pour résoudre ce problème. Le corps de boucle s'écrit :

somme ← *somme* + *nbre*
lire(*nbre*)

La condition d'arrêt de la boucle est : *nbre* < 0, c'est-à-dire que l'on arrête si l'utilisateur entre un nombre négatif.

Pour que la boucle fonctionne correctement, il faut initialiser la somme à zéro et lire le premier nombre. Nous obtenons ainsi :

```
somme ← 0
lire(nbre)          { saisie d'avance : valeur lue et non encore traitée }
tant que nbre ≥ 0 faire
    somme ← somme  + nbre
    lire(nbre)    { saisie courante : valeur lue et non encore traitée }
fintantque
afficher(somme)
```

Cet algorithme révèle donc qu'il y a deux opérations de saisie, la saisie d'avance et la saisie courante. Cela peut paraître surprenant à première vue, mais en fait c'est normal puisque c'est la valeur lue qui détermine l'arrêt de la boucle.

On aurait pu être tenté de changer l'ordre des instructions du corps de la boucle.

lire(*nbre*)
somme ← *somme* + *nbre*

La condition d'arrêt de la boucle ne change pas, c'est toujours *nbre* < 0. Alors que faire si *nbre* est négatif ? On doit empêcher l'addition de ce *nbre* à *somme*, ce qui peut se faire par un test. On obtient :

lire(*nbre*)
si *nbre* ≥ 0 alors
    somme ← *somme* + *nbre*
finsi

L'initialisation de *somme* ne change pas, par contre pour *nbre*, il y a un problème. Il ne reste qu'à utiliser un « truc » pour que ça marche quand même : on initialise *nbre* à zéro ce qui permet d'entrer dans le corps de la boucle :

Nous obtiendrions l'algorithme suivant :

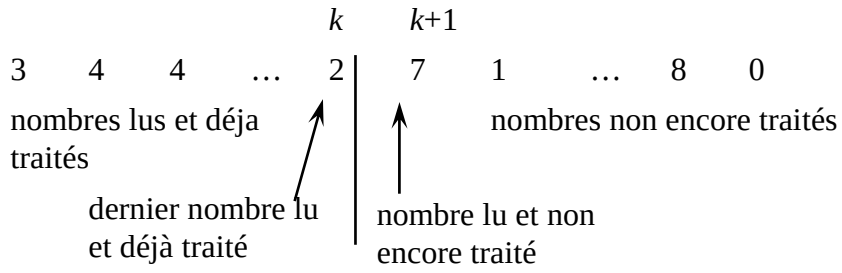
somme ← 0
nbre ← 0
tant que *nbre* ≥ 0 faire
    lire(*nbre*)
    si *nbre* ≥ 0 alors
        somme ← *somme* + *nbre*
    finsi
fintantque
afficher(*somme*)

Cet algorithme est mauvais car il nous force à utiliser des trucs pour empêcher l'arrêt prématuré de la boucle ; de plus, le corps de la boucle est inutilement compliqué à cause de la structure alternative qui n'est pas économique du tout car le test si *nbre* ≥ 0 se fait deux fois pour chaque passage dans la boucle ! Une solution pareille doit donc être rejetée.

La structure répétitive

- On entre au clavier une suite de nombres entiers positifs. Ecrire un programme qui détermine le nombre de fois que deux valeurs identiques apparaissent de façon consécutive dans la suite de nombres entrés.
- Exemple : 4 1 34 34 5 1 1 1 6 6 8 8.
- Cette suite contient 5 doublons.

Pour résoudre ce problème, il faut examiner successivement tous les nombres de la suite. Il faut donc utiliser une boucle. Faisons un schéma où nous supposons que le problème posé soit déjà en partie résolu et essayons de cerner les variables impliquées à partir de cet état, en plus des variables découlant de la description des données et des résultats :



Nous supposons que la fin de la suite est indiquée par 0. Pour passer de l'état *k* à l'état *k*+1, il faut éventuellement ajuster le nombre de doublons. Cet ajustement dépend évidemment de la valeur lue et non encore traitée, mais aussi de la dernière valeur lue et déjà traitée. Voici la description des variables impliquées :

nombre de la suite lu et non encore traité	*nbre_lu*
nombre de doublons dans la suite traitée	*nbre_doublons*
dernier nombre lu et déjà traité	*dernier_traite*

Comme, de plus, la valeur entrée détermine la fin de la boucle, il faut procéder à la lecture d'un nombre dans le corps de la boucle. La structure générale de l'algorithme est la suivante :

1. saisie d'avance de *nbre_lu*
2. tant que pas(*nbre_lu* = 0) faire

   2.1 exploiter *nbre_lu*
   2.2 saisie courante de *nbre_lu*

   fintantque
3. afficher *nbre_doublons*

Les descriptions informelles des trois variables forment ce que l'on appelle en informatique *l'invariant de la boucle*. L'invariant décrit la sémantique de la boucle. C'est une propriété de la boucle vérifiée en trois endroits bien précis :

1. saisie d'avance de *nbre_lu*

   l'invariant est vérifié juste avant la boucle

2. tant que pas(*nbre_lu* = 0) faire

   l'invariant est vérifié juste avant le corps de la boucle

2.1 exploiter *nbre_lu*

2.2 saisie courante de *nbre_lu*

   l'invariant est vérifié juste après le corps de la boucle

   fintantque

3. afficher *nbre_doublons*

Cette propriété ne change donc pas quel que soit le nombre de fois que la boucle est exécutée, d'où son nom d'invariant.

Juste après la boucle, nous pouvons affirmer que l'invariant est vérifié ainsi que la condition d'arrêt de la boucle. Dans notre cas, cela signifie que *nbre_doublons* représente bien le nombre de doublons de toute la suite.

Une fois que l'invariant est déterminé, celui-ci nous guide pour déterminer les instructions à placer dans le corps de la boucle ainsi que les initialisations à effectuer.

Raffiner 2.1 :

Afin que l'invariant soit toujours vrai à la fin du corps de la boucle, il faut éventuellement ajuster *nbre_doublons* en fonction de *nbre_lu* et de *dernier_traite.*

si *nbre_lu = dernier_traite*
alors ajouter 1 à *nbre_doublons*
finsi
dernier_traite ← *nbre_lu*

dernier_traite doit être changé pour être conforme avec sa signification dans l'invariant.

Ainsi, le corps de la boucle que nous venons d'écrire laisse inchangé l'invariant tel que nous l'avions défini.

Pour que l'algorithme fonctionne correctement, il faut aussi que l'invariant soit vérifié juste avant la boucle.

nbre_doublons est initialisé à 0 et *dernier_traite* avec une valeur qu'aucun nombre de la suite ne peut avoir, dans ce cas, par exemple, –1.

Le programme LERIA s'écrit finalement :

```
PROGRAM nombre_doublons;
CONST fin_suite = 0;
VAR int : nbre_lu,dernier_traite, nbre_doublons;
BEGIN
  nbre_doublons := 0;
  dernier_traite := -1;
  WRITE("Entrez un entier > 0, 0 pour arrêter : "); { saisie d"avance }
  READ(nbre_lu);

  WHILE nbre_lu <> fin_suite DO BEGIN
    IF nbre_lu = dernier_traite THEN nbre_doublons := nbre_doublons + 1;
    dernier_traite := nbre_lu;

    WRITE("Entrez un entier > 0, 0 pour arrêter : "); { saisie courante }
    READ(nbre_lu);
  END;

  WRITE("La suite comporte ",nbre_doublons, " doublons",\n);
END.
```

La structure répétitive

- Ecrire un programme qui, pour un entier *max* entré au clavier, affiche un carré de la forme suivante :

```
Entrez la taille du carré : 4
     1  2  3  4
     2  3  4  5
     3  4  5  6
     4  5  6  7
```

Cet exercice nous permet d'utiliser deux boucles dont une est imbriquée dans l'autre.

La structure de la solution est simple :

1. Saisir la taille du carré
2. Afficher le carré

La taille du carré est mémorisée dans la variable *max*.

Raffiner 2.

L'affichage à produire consiste en *max* lignes. Ceci nous amène à la boucle suivante :

nbre_ligne ← 1

Tant que *nbre_ligne* ≤ *max* faire

    afficher une ligne

    nbre_ligne ← *nbre_ligne* + 1 { assurer la terminaison de la boucle }

FinTantque

La variable *nbre_ligne* désigne le numéro de la ligne en cours de traitement.

Il ne nous reste qu'à raffiner le sous-problème : afficher une ligne.

Pour afficher une ligne, il faut afficher un nombre fixe d'entiers. Ce nombre nous est donné par la variable *max*.

Tout comme dans la boucle précédente, nous aurons une nouvelle variable, à savoir *nbre_colonne*. Elle désigne le numéro de la colonne en cours de traitement. Pour chaque colonne de la ligne, une valeur doit être affichée. Cette valeur sera représentée par la variable *valeur_colonne*.

Au début de la ligne, la valeur de cette variable est initialisée avec le numéro de la ligne et chaque fois qu'une colonne a été traitée, cette variable est incrémentée.

L'affichage d'une ligne s'écrit par conséquent :

```
nbre_colonne ← 1
valeur_colonne ← nbre_ligne
Tant que nbre_ colonne ≤ max faire
    afficher valeur_colonne
    valeur_colonne ← valeur_ colonne + 1
    nbre_ colonne ← nbre_ colonne + 1  { assurer la terminaison de la boucle }
FinTantque
```

Voici l'algorithme obtenu en fusionnant cette dernière partie dans la première :

```
nbre_ligne ← 1
Tant que nbre_ligne ≤ max faire
    nbre_colonne ← 1
    valeur_colonne ← nbre_ligne
    Tant que nbre_ colonne ≤ max faire
        afficher valeur_colonne
        valeur_colonne ← valeur_ colonne + 1
        nbre_ colonne ← nbre_ colonne + 1  { terminaison de la boucle }
    FinTantque
    nbre_ligne ← nbre_ligne + 1  { assurer la terminaison de la boucle }
FinTantque
```

La traduction en LERIA de l'algorithme précédent est immédiate :

```
PROGRAM carre;
VAR int : max, nbre_ligne, nbre_colonne, valeur_colonne;
BEGIN
  { saisie de la taille du carré }
  WRITE("Entrez la taille du carré : "); READ(max);

  { affichage}
  nbre_ligne := 1;
  WHILE nbre_ligne <= max DO BEGIN
    { afficher une ligne }
    valeur_colonne := nbre_ligne;
    nbre_colonne := 1;
    WHILE nbre_colonne <= max DO BEGIN
      WRITE(valeur_colonne:3);
      nbre_colonne := nbre_colonne + 1;
      valeur_colonne := valeur_colonne + 1;
    END;
    WRITE(\n);                        { passer à la ligne }
    nbre_ligne := nbre_ligne + 1;
  END;
END.
```

Il faut que le lecteur se rende absolument compte que le développement d'un programme n'est pas un processus linéaire.

Un programme ne s'écrit pas d'une traite, de la première ligne à la dernière, mais se développe d'une manière systématique, en partant du plus général et en raffinant progressivement chacune des étapes. Cette méthode de développement de programmes préconisée par Niklaus Wirth (inventeur du Pascal, Modula, Oberon...) s'appelle encore la méthode de développement par raffinements successifs. C'est la méthode que nous utiliserons pour développer tous nos programmes.

Dans le chapitre suivant, nous verrons que les langages de programmation usuels donnent au programmeur un outil pour écrire les programmes dans cet esprit : c'est le concept de fonction.

Exercices

- Tous les programmes doivent être réalisés en langage LERIA.
- N'oublier pas de bien les commenter. Cette opération se fait tout au long du développement d'un programme et non à la fin de ce processus !

1) Ecrire un programme qui calcule le produit des nombres entrés au clavier. Trouver une façon pour l'utilisateur d'indiquer la fin de la suite des nombres.

2) Ecrire un programme qui reconstitue le nombre formé par la suite de chiffres entrés au clavier. Exemple : 2 3 5 1 donne le nombre 2351.

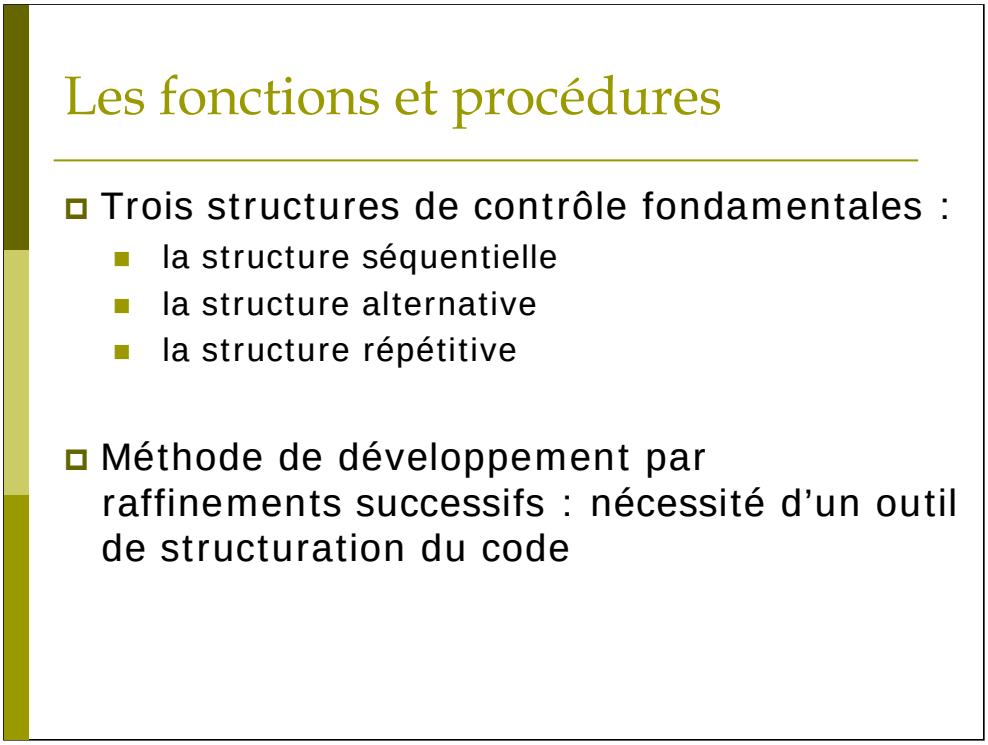
3) Une bactérie se multiplie par deux tous les jours. A la surface d'un morceau de viande on a mesuré *x* bactéries par cm^2. Sachant qu'une viande saine ne peut contenir plus de 2000 bactéries par cm^2, écrire un programme qui détermine si pour une valeur *x* entrée au clavier, la viande peut être considérée comme saine, et dans ce cas pour combien de temps, ou sinon depuis combien de temps la viande doit-elle être considérée comme avariée.

4) Ecrire un programme qui vérifie si un nombre entier positif est un nombre premier. Un nombre est premier s'il n'est divisible que par lui-même et par 1. Le premier nombre premier est 2.

5) Ecrire un programme qui affiche tous les nombres premiers compris entre 2 et un nombre entier positif entré au clavier.

6) Ecrire un progamme qui affiche une table de multiplication des nombres entiers de 1 à 10.

Les fonctions et procédures



Jusqu"à présent, nous avons étudié les trois structures de contrôle fondamentales permettant d"écrire tout programme calculable effectivement. Rappelons encore une fois ces trois structures :

• la structure séquentielle: les instructions sont exécutées en séquence, l"une après l"autre, dans leur ordre d"apparition ;

• la structure alternative: les instructions sont exécutées en fonction de la réalisation d'une condition;

• la structure répétitive: les instructions sont exécutées un nombre variable de fois dépendant de la réalisation de la condition d"arrêt.

Mais l"écriture de programmes *corrects* n"est pas une chose aussi facile que cela paraît à première vue. Il est intéressant de disposer d"une méthode de développement permettant de construire des programmes corrects. C"est ce que nous avons fait dans les chapitres précédents en utilisant la méthode de développement de programmes par raffinements successifs tout en s"aidant d"une méthode de construction de boucles basée sur le concept d"invariant.

Dans la méthode de développement par raffinements successifs, le but est de décomposer un problème en sous-problèmes, chacun de ces sous-problèmes étant plus facile à résoudre que le problème initial. C"est ce que nous avons fait dans les programmes réalisés, mais l'inconvénient est que le programme ne garde plus aucune trace de cet effort de structuration et de décomposition, à part les commentaires.

Les fonctions et procédures

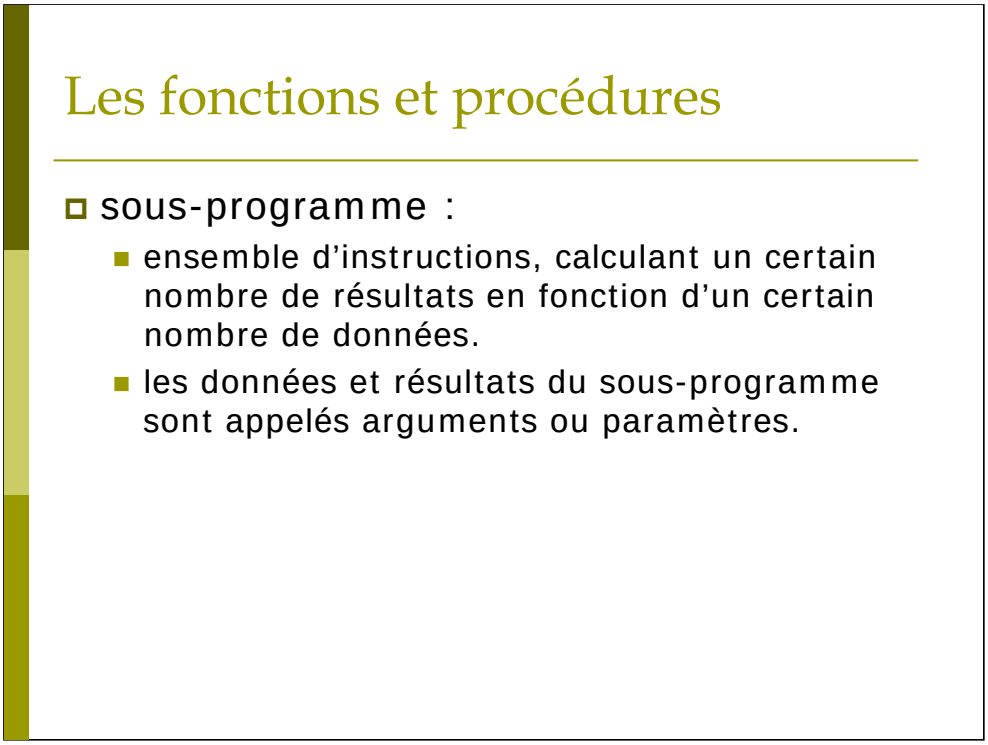
- Outil de structuration du code :
  - le sous-programme
    - fonction
    - procédure
- Ce sont des concepts que l'on retrouve dans tous les langages de programmation impératifs.

Cette lacune est comblée par le concept de sous-programme (*procedure*, *subroutine*). Nous verrons qu'il existe deux types de sous-programmes : les fonctions et les procédures.

Ces notions favorisent, supportent ce style de développement en ce sens qu"elles permettent au programmeur de traduire en LERIA, telles quelles, les différentes phases de raffinement.

Le concept de fonction ou procédure a encore d"autres avantages. Il permet notamment de pouvoir réutiliser du code existant et, de ce fait, augmente la productivité. De plus, il permet d"introduire un niveau d"abstraction en permettant de construire et de manipuler des structures qui n"existent pas d"office dans le langage.

Le concept de fonction et/ou de procédure est présent dans tous les langages de programmation impératifs usuels.



Un sous-programme est un moyen dont dispose le programmeur pour nommer une action complexe.

Cette action est décrite au moyen d'un ensemble d'instructions du langage. Le programmeur a le moyen de paramétriser cette action à travers des arguments qu'il fournit au sous-programme.

Pour réaliser l'action décrite par un sous-programme, un programme ou un sous-programme peut *appeler* ou *invoquer* le sous-programme en question.

La différence essentielle entre un programme et un sous-programme est donc qu'un sous-programme peut agir pour le compte d'un programme ou d'un sous-programme.

Lorsqu'il n'y a pas de confusion possible, nous utiliserons indifféremment les termes programme et sous-programme.

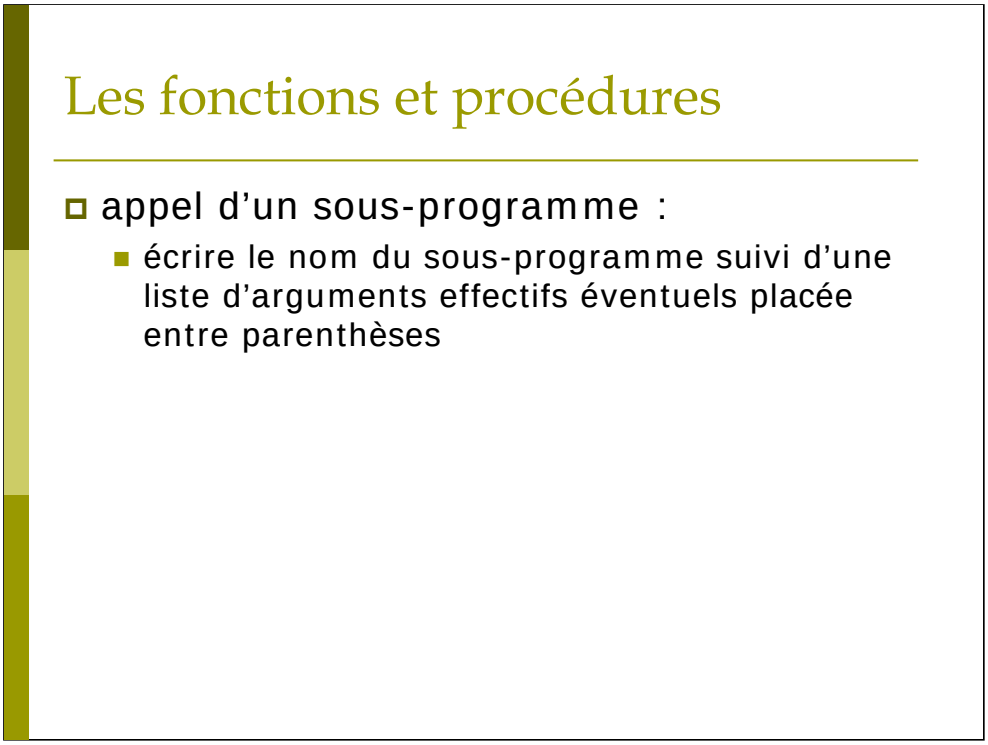
Lorsqu'un programme appelle un sous-programme, nous distinguerons entre le *programme appelé* et le *programme appelant*. A la fin de l'exécution du programme appelé se produit un retour automatique vers le programme appelant à l'instruction suivant logiquement l'instruction d'appel du sous-programme.

Les fonctions et procédures

- la définition d'un sous-programme comporte :
  - le nom du sous-programme
  - le nombre et le type de ses arguments
  - une description de l'action

Lorsque le programmeur définit un sous-programme, il doit l'identifier en lui donnant un nom et en précisant le nombre et le type de ses arguments. Ensuite, il fournit la suite d'instructions produisant l'action voulue au moment de l'exécution du sous-programme.

Ces instructions manipulent généralement les arguments éventuels du sous-programme. Il est donc nécessaire de donner un nom à chacun des arguments du sous-programme. On les appelle *arguments formels* ou *paramètres formels*.



Pour appeler un sous-programme, il suffit, dans le programme appelant, d'écrire le nom du sous-programme suivi d'une liste d'arguments effectifs (réels) placée entre parenthèses.

Dans LERIA, les parenthèses sont obligatoires même si le sous-programme invoqué n'admet pas d'arguments.

Chacun de ces arguments effectifs est censé remplacer l'argument formel correspondant tel qu'il est défini dans la définition du sous-programme. Nous verrons plus loin comment se fait très précisément ce remplacement. Dans la plupart des langages de programmation impératifs, la correspondance entre les arguments effectifs et formels doit être totale, tant du point de vue nombre que de leurs types respectifs. Il en sera ainsi pour le langage LERIA.

Au moment de l'exécution, dans le programme appelant, à l'endroit de l'appel du sous-programme, les instructions de ce dernier seront exécutées avec les arguments effectifs fournis dans l'appel.

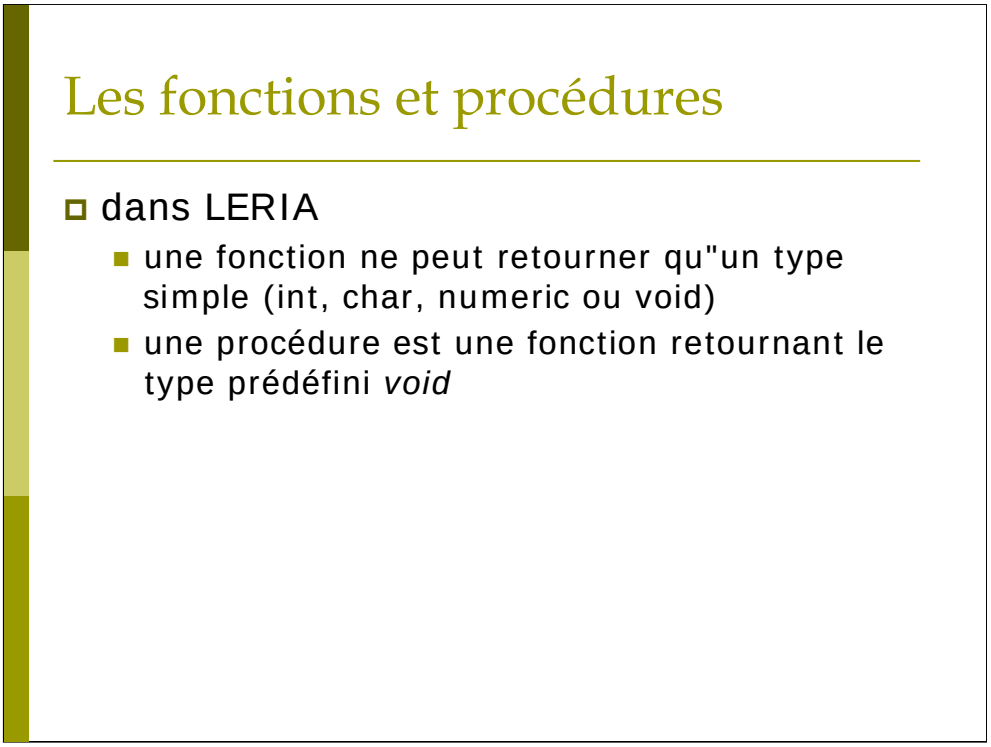
Une fonction qui s'appelle elle-même est appelée une *fonction récursive*. LERIA supporte la récursion et nous étudierons ce concept en détail dans un chapitre ultérieur.

Les fonctions et procédures

- un sous-programme ne représentant qu'une action est appelé une procédure
- un sous-programme dénotant une valeur est une fonction

A travers l''utilisation de procédures, le programmeur a la possibilité de définir de nouvelles primitives (instructions). A la différence des instructions prédéfinies dans le langage, ces nouvelles instructions ne sont comprises que par le programme dans lequel ces procédures sont définies.

Une fonction ne peut être appelée que dans une expression. Il faut évidemment que la fonction en question soit définie dans le programme appelant !



LERIA supporte les fonctions qui retournent une valeur de type *int*, *char* ou *numeric* et les procédures. Ces dernières ne retournent pas de valeur et on l'indique par le type prédéfini *void*. Dans un programme, les fonctions éventuelles doivent être définies soit au niveau du bloc principal dans la partie définition de fonctions située après la section des variables, soit dans le bloc d'une fonction. Dans ce dernier cas, nous parlons alors de fonctions imbriquées.

partie_définition_de_fonction ::= { définition_de_fonction }

définition_fonction ::= signature bloc ";"
signature ::= identificateur_type "FUNCTION" identificateur "("
liste_paramètres_formels] ")" ";"

Le type de retour de la fonction précède le mot réservé FUNCTION. Voici un exemple d'un programme définissant une procédure n'admettant pas de paramètres :

```
PROGRAM test_procedure;
  void FUNCTION message_coucou();
  BEGIN
    WRITE("coucou");
  END;
BEGIN
  message_coucou();  { appel de la procédure }
END.
```

Les fonctions et procédures

- l'instruction RETURN, qui ne peut être utilisée que dans une fonction, provoque le retour inconditionnel dans le programme appelant
- format :

  return_instruction ::= "RETURN" [expression]

- elle est superflue à la fin d'une procédure

Cette instruction ne peut être utilisée que dans un bloc fonction et provoque l''arrêt de l''exécution de la fonction et le retour au programme appelant.

Si la fonction n''est pas du type *void*, l''instruction RETURN doit être suivie d''une expression dont l''évaluation fournira une valeur qui sera la valeur retournée par la fonction. Cette expression doit être du type *char, int* ou *numeric*.

Une fonction de type procédure, c'est-à-dire dont le type de retour est *void*, ne doit pas nécessairement contenir une instruction RETURN. En effet, la fin de l''instruction composée du bloc fonction provoque un retour automatique au programme appelant. Une fonction peut contenir plusieurs instructions RETURN, mais du point de vue méthodologique, cette pratique est à déconseiller.

Le principe sain qui prévaut est : une fonction a un point d'entrée et un point de sortie !

Les fonctions et procédures

```
PROGRAM exemple_fonction;
VAR numeric : a,b;
  numeric FUNCTION min(numeric : i; numeric :j);
  BEGIN
    IF i < j
       THEN RETURN i
        ELSE RETURN j;
  END;
BEGIN
  a := 4.; b := 3.;
  WRITE("Le minimum est : ",min(a,b),\n);
  WRITE(min(5.*a,7.*b):5:0,\n);
END.
```

arguments formels

arguments effectifs

appels de fonction

Une fonction (procédure) peut admettre des arguments. Dans ce cas, la définition de la fonction (procédure) doit contenir la liste des arguments avec leurs types. Dans cette définition, chaque argument a un identificateur unique pour la fonction en question.

Le compilateur vérifie que le type de chaque argument effectif est égal à celui de l'argument formel correspondant. Il doit y avoir exactement autant d'arguments effectifs que d'arguments formels.

Les fonctions et procédures

- Dans LERIA il y a deux modes de passage des arguments :
  - le passage par valeur
  - le passage par variable
- Format :

liste_paramètres_formels ::=
    définition_paramètre { ";" définition_paramètre }

définition_paramètre ::=
    identificateur ":" ["VAR"] identificateur

Il nous reste à préciser comment s'effectue la transmission des paramètres entre le programme appelant et appelé.

Le mode par défaut de passage d'un argument en LERIA est le *passage par valeur*. Dans ce cas, l'argument effectif est une expression qui est évaluée au moment de l'exécution. Cette valeur est placée en mémoire centrale à un endroit appelé la pile d'exécution. Le sous-programme a accès a cette valeur et peut en faire ce qu'il veut. A la fin de l'exécution du sous-programme, cette valeur est détruite automatiquement (enlevée de la pile d'exécution) et n'existe plus.

Dans le deuxième mode de passage, le *passage par variable* ou *par adresse (par référence)*, l'argument effectif doit être un accès à une variable. Maintenant, lorsque la fonction est invoquée, l'argument formel est un alias pour l'argument effectif correspondant. Ceci signifie en clair, que partout dans le corps de la fonction où le l'argument formel est manipulé, en réalité, c'est l'argument effectif qui l'est. Un argument effectif du programme appelant passé par variable peut donc être modifié par le sous-programme appelée, ce qui n'est jamais le cas lors d'un passage par valeur.

En LERIA, pour signifier le passage par variable, l'argument formel doit être précédé du mot réservé VAR.

Les fonctions et procédures

```
PROGRAM test_valeur_variable;
VAR int : i, i_carre;
  void FUNCTION carre_valeur(int : i; int : res);
  BEGIN
    res := i * i;
  END;
  void FUNCTION carre_variable(int : i; int : VAR res);
  BEGIN
    res := i * i;
  END;
BEGIN
  i := 5; i_carre := -1;
  carre_valeur(i,i_carre);
  WRITE(i_carre,\n);   <-   affiche -1
  carre_variable(i,i_carre);
  WRITE(i_carre,\n);   <-   affiche 25
END.
```

par valeur

par variable

Cet exemple illustre la différence de comportement entre deux procédures dont un argument est passé tantôt par valeur, tantôt par variable.

Lors du passage par variable dans la procédure *carre_variable*, l'argument formel *res* est lié à l'argument effectif *i_carre*. Cette liaison reste active tout le temps que la procédure *carre_variable* existe. A chaque fois que l'on modifie alors *res*, en fait on modifie *i_carre* ! A la fin de l'exécution de la procédure *carre_variable*, cette liaison est détruite.

Dans le cas du passage par valeur, au moment de l'appel, l'argument formel est initialisé avec la valeur résultant de l'expression *i_carre*. A la fin de l'exécution de la fonction, les arguments formels *i* et *res* sont détruits et *i_carre* n'aura pas changé.

Notons que lors d'un appel par variable, l'argument effectif doit être un accès à une variable, alors que lors d'un passage par valeur, l'argument effectif peut être une expression !

Les fonctions et procédures

- une fonction peut accéder aux identificateurs définis au niveau principal
- LERIA permet au programmeur de définir des constantes et des variables locales à une fonction
- portée d'un identificateur ?

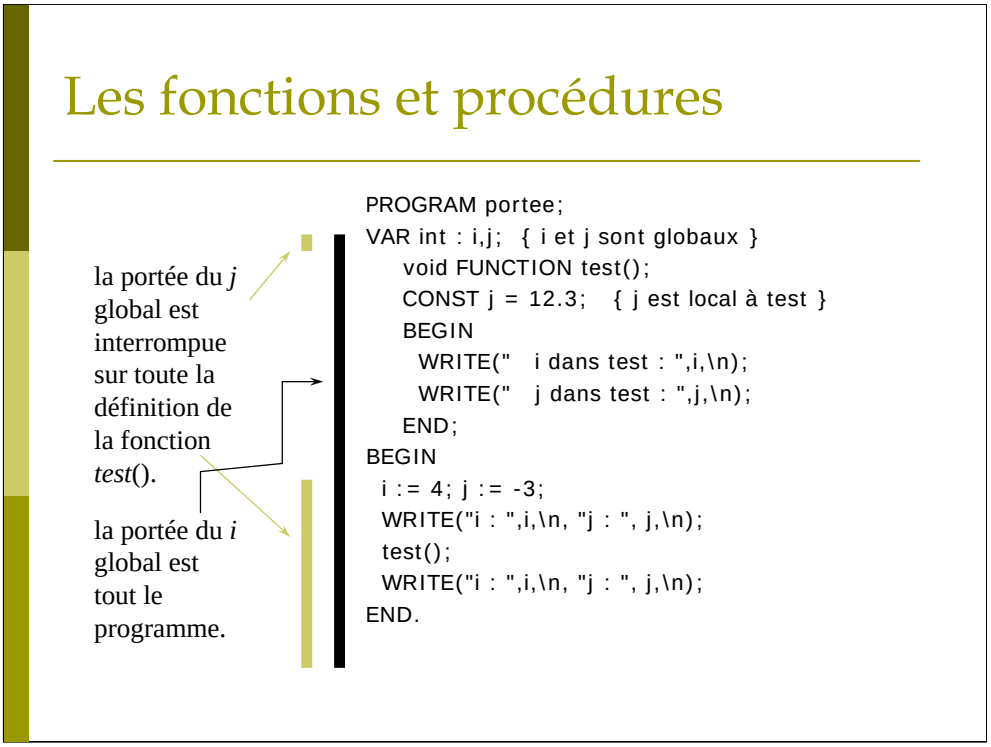
En LERIA, une fonction a accès à tous les identificateurs définis au niveau du programme principal. On dit que ce sont des *identificateurs globaux*.

Les paramètres formels ainsi que les identificateurs définis dans une fonction sont des *identificateurs locaux* à la fonction. Ils ne peuvent être référencés que dans le corps de cette fonction.

La *portée d'un identificateur* est la partie du texte du programme où un identificateur garde la signification qui lui a été donnée lors de sa définition.

Ainsi la portée d'un identificateur global est généralement tout le texte du programme depuis sa définition jusqu'à la fin du texte à l'exception des fonctions définissant localement le même identificateur.

La portée d'une variable est un concept lié à la seule vue statique du texte du programme. Cette notion ne doit pas être confondue avec la notion d'existence d'une variable.



Dans cet exemple, on définit deux symboles globaux : *i* et *j*. *i* garde sa signification depuis son endroit de définition jusqu'à la fin du programme principal : sa portée est par conséquent tout le texte du programme, y compris dans le corps de la fonction *test*(). Ceci est d'ailleurs illustré par l'affichage du contenu de la variable *i* dans le corps de la fonction *test*().

La portée de l'identificateur global *j* par contre est interrompue sur le corps de la fonction *test*(). Ceci est dû au fait que cette dernière définit le symbole *j* localement comme une constante. A l'intérieur de la fonction *test*(), la signification globale de *j* n'est plus accessible !

Du point de vue méthodologique, il n'est pas recommandé dans une fonction de travailler avec des variables globales définies à l'extérieur de la fonction. Une fonction doit en effet être vue comme une "boîte noire" réalisant un travail déterminé et doit être insensible à des "perturbations" externes. Ceci n'est plus garanti si dans la fonction on travaille avec une variable globale, car si on change le nom de la variable au niveau global, la fonction ne peut plus fonctionner correctement, il faut aussi modifier la fonction, ce qui n'est pas souhaitable. On dit encore qu'une fonction ou procédure modifiant une variable d'un programme appelant autres que celles passées par variable a un *effet de bord*. Les effets de bord sont à éviter !

Les fonctions et procédures

- existence d'une variable
  - notion liée à l'exécution du programme
  - une variable globale existe depuis le chargement du programme en mémoire jusqu'à la fin de l'exécution du programme
  - une variable locale est créée au moment de l'activation de la fonction la définissant et existe jusqu'à la fin de l'exécution de la fonction en question.

Le concept de portée d'une variable ne doit pas être confondu avec celui d'existence d'une variable. Une variable ne peut exister que si le programme est exécuté. Dans ce cas, il faut encore envisager deux cas, celui d'une variable globale et celui d'une variable locale.

Une variable globale existe depuis que le programme la définissant est chargé en mémoire centrale. Lorsque l'exécution du programme est terminée, le programme et les variables globales qu'il définit sont enlevés de la mémoire centrale de l'ordinateur. A ce moment-là, la variable globale a cessé d'exister.

Le problème est tout autre pour une variable locale. Celle-ci n'existe pas lorsque le programme est chargé en mémoire. Elle est créée au moment où la fonction la définissant est activée par un programme appelant. Cette variable locale existe aussi longtemps que la fonction la définissant. Si cette dernière meurt (exécution de la fonction terminée), la variable locale est détruite.

Les variables locales sont donc créées dynamiquement au moment de l'exécution du programme selon la séquence d'activation des fonctions.

Les fonctions et procédures

Exemple d'une procédure :

```
PROGRAM echange;
VAR int : x,y;
  void FUNCTION swap(int : VAR a; int : VAR b);
  VAR int : tmp;
  BEGIN
    tmp := a; a := b; b := tmp;
  END;
BEGIN
  x := 6; y := 12;
  WRITE("x=",x," y=",y,\n);  { affiche 6 et 12  }
  swap(x,y);                 { échanger x et y }
  WRITE("x=",x," y=",y,\n);  { affiche 12 et 6  }
END.
```

si on n'utilise pas le *passage par variable*, la procédure ne fonctionne pas correctement.

Cet exemple illustre la définition et l'utilisation d'une procédure admettant deux arguments passés par variable.

La procédure *swap*() a pour but de permuter le contenu des deux variables passées comme arguments.

Dans une section précédente, nous avons déjà vu que pour effectuer ce travail, il faut utiliser une troisième variable temporaire. C'est ce qui est fait ici à travers la variable *tmp* locale à la procédure *swap*().

Pour que la procédure permute les valeurs des arguments effectifs, il est essentiel que les arguments soient passés par variable. De cette façon, la procédure travaille sur les variables réelles utilisées par le programme appelant.

Le lecteur n'a qu'à vérifier le résultat du programme présenté ci-dessus en omettant de déclarer les arguments formels de la procédure *swap*() comme étant *par variable*.

Les fonctions et procédures

- Pour terminer ce chapitre traitant du concept de sous-programme, nous allons refaire l'exercice du tri de trois nombres en utilisant cette fois-ci des sous-programmes avec des paramètres.

Le corps principal du programme consiste à :

• saisir les trois nombres. Dans ce cas, le programme appelant veut qu"au retour le sous-programme ait placé les valeurs dans les variables respectives. Il faut donc obligatoirement utiliser le passage par variable.

• trier les trois valeurs en ordonnant d"abord les deux premières. Ensuite, la première ayant la plus petite valeur des deux est ordonnée avec la troisième (dès lors, la première variable contient la plus petite valeur des trois), et finalement la deuxième est ordonnée avec la troisième.

• afficher les trois nombres triés.

```
PROGRAM tri_trois_nombres;
VAR
  numeric : n1, n2, n3;  { les trois nombres à trier }

  void FUNCTION affichage_3_nombres(numeric : x; numeric : y; numeric : z);
  BEGIN
    WRITE(x:5:2," ",y:5:2," ",z:5:2);
  END;

  void FUNCTION saisie_3_nombres(numeric : VAR x; numeric : VAR y; numeric : VAR z);
  BEGIN
    WRITE("Entrez trois nombres séparés par des RETURN : ");
    READ(x,y,z);
  END;

  void FUNCTION permute(numeric : VAR a; numeric : VAR b);
  VAR numeric: tmp;         { variable temporaire nécessaire pour l"échange }
  BEGIN
    tmp := a; a := b; b := tmp;
  END;

  void FUNCTION ordonne(numeric : VAR i; numeric : VAR j);
  BEGIN
    IF i > j THEN permute(i,j);
  END;

BEGIN
  saisie_3_nombres(n1, n2, n3);

  { tri }
  ordonne(n1,n2);
  ordonne(n1,n3);
  ordonne(n2,n3);

  affichage_3_nombres(n1,n2,n3);
END.
```

Puisque le contenu des variables peut être changé, de nouveau, nous utilisons le passage par variable. Remarquons que le sous-programme *ordonne* appelle le sous-programme *permute* et que les paramètres formels de *ordonne* jouent ainsi le rôle de paramètres effectifs.

Exercices

- Tous les programmes doivent être réalisés en langage LERIA.
- N'oublier pas de bien les commenter. Cette opération se fait tout au long du développement d'un programme et non à la fin de ce processus !

1) Que calcule le programme suivant ?

```
PROGRAM passage_parametres;
VAR int : x,y;
  void FUNCTION p(int : n; int : VAR m);
  BEGIN
    n := n - 1; m := m - 1;
  END;
BEGIN
  x := 10; y := 10;
  p(x,y);
  WRITE("X= ",x:4:0," Y = ",y:4:0,\n);
END.
```

2) Quel est le résultat de ce programme ?

```
PROGRAM passage_parametres;
VAR int : i;
  void FUNCTION p(int : VAR k; int : VAR l);
  BEGIN
    k := k + i; l := l + i;
  END;
BEGIN
  i := 1;
  p(i,i);
  WRITE("I= ",i:4:0,\n);
END.
```

Exercices

- Tous les programmes doivent être réalisés en langage LERIA.
- N'oublier pas de bien les commenter. Cette opération se fait tout au long du développement d'un programme et non à la fin de ce processus !

3) Que fait le programme suivant ?

```
PROGRAM bizarrerie;
VAR int : i ;

  void FUNCTION bizarre(int : VAR k; int : VAR m);
  BEGIN
    k := k * m;
    m := m + i - k;
  END;

BEGIN
  WRITE("Entrez un nombre : "); READ(i);
  bizarre(i,i);
  WRITE("i = ",i);
END.
```

4) Refaire les exercices 4 et 5 du chapitre précédent (nombres premiers) en utilisant cette fois-ci des procédures et/ou fonctions.


Les tableaux

Les tableaux

- un tableau permet de définir un ensemble fini d'éléments de même type
- un tableau est conservé dans la mémoire centrale de l'ordinateur
- tous les éléments d'un tableau portent le même nom, mais chacun porte un numéro (*indice*) différent

Tout programme peut être écrit avec les trois structures de contrôle de base. Néanmoins, nous venons de voir que les concepts de fonction et de procédure peuvent faciliter la mise au point de programmes.

Jusqu'à présent, nous avons manipulé des variables simples de type *numeric*. Or dans certains cas, cette approche se révèle assez fastidieuse. C'est pour cela que le concept de tableau est présent dans la plupart des langages de programmation.

Un tableau est une structure de données conservée dans la mémoire centrale de l"ordinateur. L"utilisation d"un tableau permet de définir un ensemble fini d"éléments de même type.

Chaque élément d"un tableau est identifié par le même identificateur (le nom du tableau) et par un numéro unique appelé indice de l"élément du tableau. Grâce à ces deux données, on peut manipuler chacun des éléments d"un tableau.

Le temps d"accès à un élément est indépendant de la valeur de l"indice, et les éléments peuvent être accédés dans n"importe quel ordre. On dit que c"est une structure de données à accès aléatoire.

Les tableaux

- en LERIA, le programmeur doit préalablement définir un type tableau
- ensuite, le programmeur peut définir des variables de ce type tableau
- dans cette première partie, nous nous limitons aux seuls tableaux de nombres (vecteurs)

A côté des types prédéfinis *numeric* et *void*, le programmeur peut définir de nouveaux types tableaux dans la partie définition de types (après les constantes et avant les variables) :

partie_définition_types ::= { définition_type }
définition_type ::= "TYPE" identificateur "=" définition_tableau ";"
définition_tableau ::= "ARRAY" "[" const_positive "]" "OF" identificateur

L"identificateur intervenant dans *définition_tableau* doit être un identificateur de type préalablement défini, à l'exception du type *void*. Une définition de type ne peut être récursive. Exemples :

```
CONST max = 20;
TYPE tableau1 = ARRAY[200] OF numeric;
TYPE tableau2 = ARRAY[max] OF numeric;
TYPE tab1 = ARRAY[4] OF tab1;  { erreur !  définition
récursive}
```

Les tableaux

- exemple :

```
TYPE t_tableau = ARRAY[5] OF numeric;
VAR t_tableau : t;
```

Tous les éléments d'un même tableau sont du même type !

La taille des tableaux doit être fixée à la compilation. L'indice du premier élément d"un tableau est toujours un. Dans l'exemple ci-dessus, un tableau de type *t_tableau* se compose de 5 éléments numérotés de 1 à 5.

Avant d'être initialisés par le programmeur, les éléments d'un tableau ont des valeurs indéfinies.

Un tableau peut être utilisé comme argument d'un fonction, mais une fonction ne peut pas retourner un tableau autrement que comme un de ses arguments passés par variable.

Il est permis d'affecter une variable de type tableau à une autre, pourvu que les deux variables soient de même type, c"est-à-dire s"ils ont été déclarés à l"aide du même identificateur de type de même portée.

Les tableaux

- accès à un élément d'un tableau

  Le nom du tableau doit être suivi par une expression placée entre crochets. L'expression est évaluée à l'exécution et représente l'indice.

- exemples :

```
t[3] := 12.5;
i := 5;
WRITE(t[i]);
```

L"accès à une variable de type tableau, c'est-à-dire à un élément d"un tableau, est régi par la règle syntaxique suivante :

accès_variable ::= identificateur { "[" expression "]" }

Si l'identificateur est suivi de crochets, l'identificateur en question doit évidemment être du type tableau.

Exemples : pour accéder à l'élément numéroté 2 du tableau *t* défini ci-dessus, on écrit *t*[2].

Il est interdit qu'à l'exécution, l'indice ait une valeur plus petite que 1 ou plus grande que l'indice maximum indiqué dans la définition du tableau. Si tel est quand même le cas, l'exécution s'arrête avec un message d'erreur. Le système indique en plus le numéro de la ligne du programme en cause avec la valeur de l'indice.

```
PROGRAM test_tableau1;
CONST
  max = 10;
TYPE
  t_tableau = ARRAY[max] OF numeric;
VAR
  t_tableau : tab;
VAR
  numeric : i;
BEGIN
  i := 1;
  WHILE i <= max DO BEGIN
    tab[i] := i*i;
    i := i+1;
  END;
  i := 1;
  WHILE i <= max DO BEGIN
    WRITE(tab[i]:4:0);
    i := i+1;
  END;
  WRITE(\n);
  WRITE(tab[-1]);      { erreur }
  WRITE(tab[max+1]);   { erreur }
END.
```

Dans ce programme, le tableau *tab* se composant de 10 éléments est rempli avec les valeurs 1, 4, 9, 16, 25, 36, 49, 64, 81, 100.

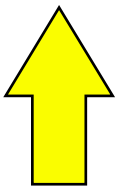
Ensuite, tous les éléments du tableau sont affichés.

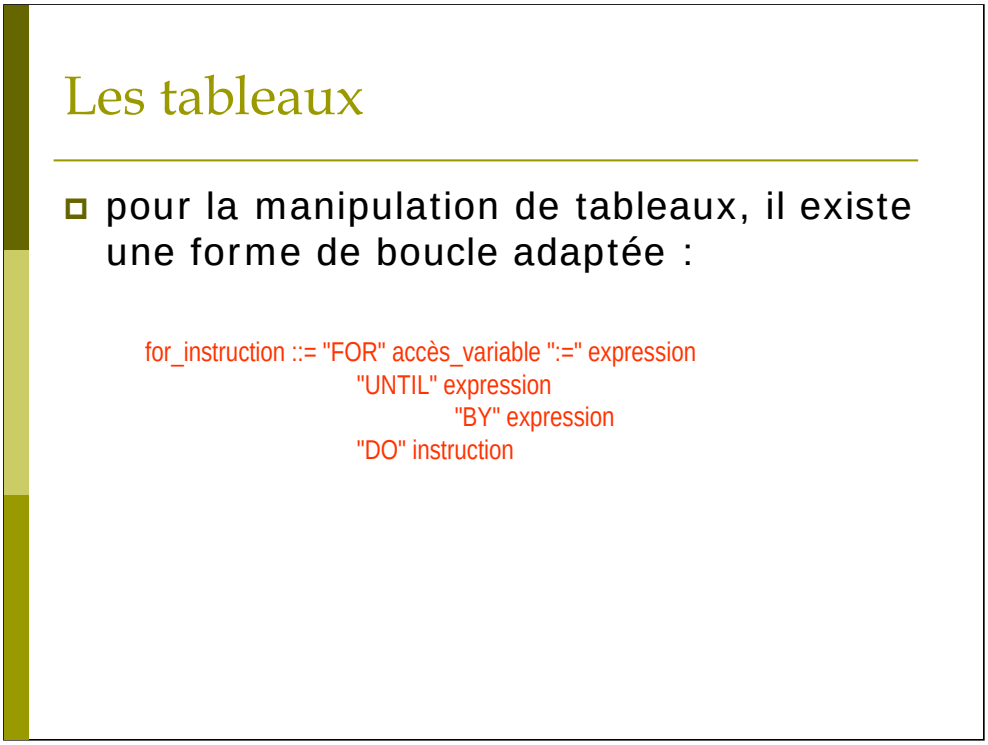
Cet exemple illustre aussi de manière triviale deux causes d'erreurs possibles à l'exécution (indice hors des limites).

Voici la trace d'exécution du programme :

```
Exécution en cours...
   1   4   9  16  25  36  49  64  81 100
*** Erreur d"exécution *** : indice hors limites à la ligne 22(-1)
Exécution terminée
```

l'indice en cause vaut -1





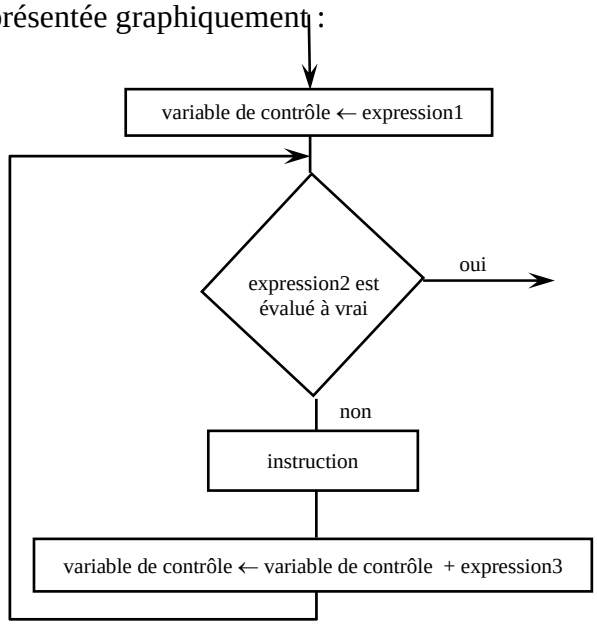
La variable accédée tout de suite après le mot réservé FOR est appelée *variable de contrôle*. L'instruction FOR est une instruction WHILE adaptée en ce sens que l'incrémentation de la variable de contrôle s'effectue automatiquement à la fin de l'exécution du corps de la boucle désignée par instruction. Les trois expressions sont évaluées à chaque tour de boucle.

L'instruction FOR a la sémantique suivante : la variable de contrôle est initialisée avec la valeur résultant de l'évaluation de l'expression suivant le symbole « := ». Si l'expression suivant le UNTIL n'est pas vérifiée, alors instruction est exécuté. Dans le cas contraire, l'exécution continue avec l'instruction suivant logiquement l'instruction FOR. À la fin de l'exécution de instruction, la variable de contrôle est incrémentée de la valeur fournie par l'évaluation de l'expression suivant BY. Le fait de modifier la valeur de la variable de contrôle dans instruction a un effet indéfini

La sémantique de l"instruction

for_instruction ::= "FOR" variable de contrôle ":=" expression1
"UNTIL" expression2 "BY" expression3 "DO"
instruction

peut être représentée graphiquement :



La sémantique de l"instruction

for_instruction ::= "FOR" accès_variable ":=" expression
"DOWN" expression "BY" expression "DO" instruction
peut être représentée graphiquement :

Les tableaux

Exemple :

```
PROGRAM test_tableau1;
CONST max = 10;
TYPE  t_tableau = ARRAY[max] OF int;
VAR   t_tableau : tab;
VAR   int : i;
BEGIN
  { initialisation }
  FOR i := 1 UNTIL i > max BY 1 DO tab[i] := i*i;
  { affichage à l'endroit }
  FOR i := 1 UNTIL i > max BY 1 DO WRITE(tab[i]:4:0);
  WRITE(\n);
  { affichage à l'envers }
  FOR i := max UNTIL i < 1 BY -1 DO WRITE(tab[i]:4:0);
  WRITE(\n);
END.
```

A l'exécution, ce programme affiche :

```
Exécution en cours...
   1   4   9  16  25  36  49  64  81 100
 100  81  64  49  36  25  16   9   4   1

Exécution terminée
```

Les tableaux

- Retournement d'un tableau :
  - Soit un tableau d'entiers *t*. On demande d'écrire un programme qui retourne le tableau en mémoire. Exemple :

Avant le retournement :

t	1	6	21	3	7
	1	2	3	4	5

Après le retournement :

t	7	3	21	6	1
	1	2	3	4	5

Afin de nous exercer à la manipulation du concept de tableau, nous nous proposons de réaliser quelques exemples en détail.

Pour résoudre le problème proposé, il suffit de permuter T[1] avec T[5], T[2] avec T[4].

Généralisons cette réflexion pour obtenir: permuter T[*i*] avec T[5–*i*+1].

Si on veut retourner un tableau contenant *nbre_elem* éléments (ici 5), on écrit : permuter T[*i*] avec T[*nbre_elem* - *i* + 1].

Les limites de variation des indices s"établissent ainsi :

• si *nbre_elem* est impair, il suffit de laisser l"élément central en place. Il ne bouge pas. Dans ce cas, *i* doit varier de 1 à *nbre_elem* % 2 (division entière !).

• si *nbre_elem* est pair, l"indice *i* doit varier de 1 à *nbre_elem* % 2 (division entière !).

Donc quelque soit la valeur de *nbre_elem*, l'indice doit varier de 1 à *nbre_elem* % 2.

```
PROGRAM test_tableau1;
CONST
  maximum = 100;
TYPE
  t_tableau = ARRAY[maximum] OF int;
VAR
  t_tableau : t;
VAR
  int : i,tmp,nbre_elem;
BEGIN
  { saisie du tableau }
  WRITE("Nombre d'éléments du tableau (<=", maximum,") : ");
  READ(nbre_elem);
  FOR i := 1 UNTIL i >  nbre_elem BY 1 DO BEGIN
    WRITE("t[",i:2:0,"]="); READ(t[i]);
  END;

  { retournement }
  FOR i := 1 UNTIL i > (nbre_elem % 2) BY 1 DO BEGIN
    tmp := t[i]; t[i] := t[(nbre_elem-i)+1]; t[(nbre_elem-i)+1] := tmp;
  END;

  { affichage du tableau }
  FOR i := 1 UNTIL i > nbre_elem BY 1 DO WRITE(t[i]:4:0);
  WRITE(\n);
END.
```

Comme la taille d'un tableau est statique, le programmeur doit déclarer une taille maximale pour son tableau (ici *maximum*). La taille effective doit bien-sûr être inférieure à la taille réelle. Elle est donnée ici par *nbre_elem*, le nombre d'éléments du tableau fourni par l'utilisateur. Voici une trace d'exécution :

```
Nombre d'éléments du tableau (<=100) : 5
t[ 1]=1
t[ 2]=2
t[ 3]=3
t[ 4]=4
t[ 5]=5

   5   4   3   2   1
```

Les tableaux

- Comparaison de deux tableaux :
  - Il faut écrire un sous-programme LERIA qui détermine si deux tableaux passés comme paramètres sont identiques. Le sous-programme doit retourner le résultat au programme appelant à travers une variable booléenne (ne prenant que les valeurs logiques 0 *faux* ou 1 *vrai*).

Deux tableaux *A* et *B* contenant chacun *max* éléments sont égaux si :

(*A*[1] = *B*[1]) et (*A*[2] = *B*[2]) et … et (*A*[*max*] = *B*[*max*])

A la première différence, on peut conclure que les deux tableaux ne sont pas égaux. Les éléments du tableau sont comparés un à un en commençant par le premier. L"indice doit donc varier de 1 à *max* au plus, en cas d"égalité des deux tableaux.

Pour résoudre le problème, il faut donc utiliser une boucle. Supposons que le corps de la boucle ait déjà été exécuté un certain nombre de fois.

Les éléments d"indice 1 à *i*-1 des tableaux *A* et *B* sont égaux. La variable booléenne *egalite* représente cet état d"égalité des sous-tableaux allant de 1 à *i*-1. i représente l"indice du premier élément non encore traité de *A* et de *B*. Pour que cette propriété de la variable *egalite* continue à être vérifiée, il suffit de tester l"égalité entre *A*[*i*] et *B*[*i*]. En cas d"égalité, *i* est incrémenté de 1, autrement *egalite* devient faux.

La boucle s"arrête s"il y a inégalité ou si tous les éléments des deux tableaux ont été testés, c’est-à-dire si *i* dépasse la valeur maximale permise.

Les initialisations consistent à mettre i à 1, indice du premier des éléments non encore traités et *egalite* à vrai puisque deux tableaux vides sont égaux.

```
PROGRAM egalite;
CONST maximum = 99;
TYPE  t_tableau = ARRAY[maximum] OF int;
VAR   int : max;
VAR   t_tableau : a,b;

  void FUNCTION saisie_tableau(t_tableau : VAR t; int : nbre_elem);
  VAR int : i;
  BEGIN
    FOR i := 1 UNTIL i > nbre_elem BY 1 DO BEGIN
      WRITE("Elément ",i:1:0, " : ");
      READ(t[i]);
    END;
  END;

  void FUNCTION affiche_tableau(t_tableau : t; int : nbre_elem);
  VAR int : i;
  BEGIN
    FOR i := 1 UNTIL i > nbre_elem BY 1 DO
WRITE("t[",i:1:0,"]=",t[i]," ");
    WRITE(\n);
  END;

  int FUNCTION egalite_tableau(t_tableau : a; t_tableau : b;
                                     int : nbre_elem);
  CONST vrai = 1;
  CONST faux = 0;
  VAR int : i,egalite;
  BEGIN
    i := 1;
    egalite := vrai;
    WHILE egalite AND (i <= nbre_elem) DO
      IF a[i] = b[i]
         THEN i := i+1
         ELSE egalite := faux;
    RETURN egalite;
  END;

BEGIN
  WRITE("Nombre d'éléments d""un tableau <=",maximum," : ");
  READ(max);
  saisie_tableau(a,max);
  affiche_tableau(a,max);
  saisie_tableau(b,max);
  affiche_tableau(b,max);
  IF egalite_tableau(a,b,max)
     THEN WRITE("Les deux tableaux sont égaux",\n)
     ELSE WRITE("Les deux tableaux sont différents",\n);
END.
```

Les tableaux

- Bien que, du point de vue méthodologique, il soit absolument correct de passer un tableau par valeur, on ne le fait jamais en pratique ; on utilise le passage par variable

Cela s"explique par des raisons de performance. En effet, comme nous l"avons vu précédemment, lors du passage par valeur, une copie du paramètre effectif est réalisée dans une zone mémoire accessible au programme appelé.

Dans le cas d"un tableau, cette copie peut nécessiter un temps non négligeable. C"est pour cette raison, et uniquement pour cela, que l"on passe généralement un tableau par variable, même si du point de vue méthodologique il faudrait le passer par valeur.

Les tableaux

- Voici une variante pour la fonction *egalite* :

```
i := 1;
WHILE (i <= nbre_elem) AND (a[i] = b[i])
       DO i := i+1;
IF a[i] <> b[i]
   THEN RETURN faux
   ELSE RETURN vrai;
```

- Cette variante est fausse ! Pourquoi ?

Le problème survient lorsque tout le tableau est effectivement occupé et que les tableaux sont égaux. Dans ce cas, la variable *i* finit par avoir la valeur *nbre_elem+1* et logiquement la boucle devrait s'arrêter car la sous-condition (*i* < =*nbre_elem*) n'est plus vérifiée. Ceci est vrai, mais il ne faut pas oublier que dans LERIA, une condition est toujours évaluée dans sa totalité. En clair, la sous-condition (*a*[*i*] = *b*[*i*]) est aussi évaluée une dernière fois provoquant cette fois-ci une erreur à l'exécution.

L'erreur est provoquée par le fait que nous tentons d'accéder à un élément ne faisant plus partie physiquement du tableau. Si par contre, nous travaillons sur des tableaux de taille effective inférieure à la taille physique, ce code ne provoque pas d'erreur visible. L'erreur persiste néanmoins car nous essayons d'accéder un élément ne faisant pas partie logiquement du tableau effectif en cours de traitement.

De toute façon, la condition du IF mettra tout le monde d'accord. Si les tableaux sont remplis jusqu'au dernier élément physique et qu'ils sont égaux, le test tente de porter sur des éléments inexistants. Ceci va provoquer une erreur à l'exécution. Plus grave encore : sur des tableaux égaux de taille effective inférieure à la taille physique, la fonction renvoie une valeur basée sur une valeur ne faisant pas partie logiquement des tableaux !

Les tableaux

- Tous les programmes doivent être réalisés en langage LERIA.
- N'oublier pas de bien les commenter. Cette opération se fait tout au long du développement d'un programme et non à la fin de ce processus !

1) Ecrire une fonction retournant le nombre d'occurrences d'une valeur donnée dans un tableau. La valeur, le tableau et la taille effective du tableau sont des arguments de la fonction.

2) Générer les *n* premiers nombres premiers en utilisant la méthode du crible d'Eratosthène. Tous les nombres de 2 à *n* sont marqués dans un tableau comme étant premiers. Ensuite, on examine successivement tous les nombres, depuis 2 à *n*. Pour chaque nombre *x*, on marque tous les nombres multiples de *x* comme étant non premiers. A la fin de cet examen de tous les nombres, il suffit de parcourir le tableau et d'afficher les nombres encore marqués comme premiers.

3) Développer une procédure *permut_centre*() qui réalise la permutation des éléments d'un tableau de nombres par rapport au "centre" du tableau. Exemples :

avant	23	12	3	-23	38
après	-23	38	3	23	12

avant	24	16	12	78
après	12	78	24	16

Le concept d'enregistrement

Le concept d'enregistrement

- un type enregistrement (record) se compose d'un nombre fixe d'éléments, de types éventuellement différents.
- ces différents composants et leurs types sont définis par la liste des champs de l'enregistrement.
- l'ordre des champs n'a pas d'importance.

Jusqu'à présent nous avions la possibilité de définir des variables simples de type *numeric* ou alors des variables de type tableau, où chacun des éléments du tableau était du même type.

Avec le concept d'enregistrement, nous avons la possibilité de définir des variables de type complexe par composition (agglomération) de types existants.

Chaque élément faisant partie de l'enregistrement est appelé un *champ*. Le type du champ est quelconque, pourvu qu'il soit défini (excepté le type *void*). L'ordre dans lequel les champs sont déclarés dans la définition de l'enregistrement n'a pas d'importance.

Le concept d'enregistrement

- exemple : les nombres rationnels sont des nombres de la forme a/b où a et b sont deux entiers avec $b \neq 0$.
- comment représenter élégamment ce concept dans LERIA ?
- solution : l'enregistrement

Le concept d'enregistrement

- Dans LERIA, la définition d'enregistrements se fait dans la section des types :

définition_type ::= "TYPE" identificateur "="
définition_record ";"
définition_record ::= "RECORD" définition_champs
{ définition_champs } "END"
définition_champs ::= identificateur ":" identificateur
{ "," identificateur } ";"

Tout comme dans le cas des tableaux, une définition d'enregistrement ne peut être récursive. Ceci signifie que le type d'enregistrement en cours de définition ne peut être utilisé comme type d'un champ de l'enregistrement.

Le concept d'enregistrement

Pour notre exemple, nous écrirons par exemple :

```
TYPE t_rat = RECORD
       numeric : numerateur;
       numeric : denominateur;
     END;
```

Nous aurions tout aussi bien pu écrire :

```
TYPE t_rat = RECORD
       numeric : numerateur, denominateur;
     END;
```

comme le suggèrent les règles syntaxiques.

Le concept d'enregistrement

- l'accès à un champ d'un enregistrement s'effectue par le symbole "."
- un objet de type enregistrement peut être utilisé comme argument formel/effectif
- la valeur d'une fonction ne peut être un enregistrement
- il est permis de faire des affectations entre deux objets du même type enregistrement

Pour accéder à un champ d'un enregistrement, il suffit d'indiquer le nom de l'enregistrement, suivi du symbole "." et du nom du champ en question.

A l'intérieur d'un enregistrement, tous les identificateurs de champs doivent être uniques et ne doivent pas être des mots réservés.

Le symbole "." peut évidemment être utilisé en cascade si un champ d'un enregistrement est lui-même un enregistrement.

Tout comme pour les tableaux, il est permis d'utiliser comme argument formel/effectif un objet de type *record*, mais une fonction ne peut retourner comme valeur un *record*.

Il est de même permis d'effectuer des affectations entre des objets de même type *record*.

```
PROGRAM rationnel;
TYPE t_rat = RECORD
       int : numerateur;
       int : denominateur;
     END;
VAR t_rat : r1, r2, r3;

  void FUNCTION init_rat(int : num; int : denom; t_rat : VAR r);
  BEGIN
    r.numerateur := num;
    r.denominateur := denom;
  END;

  void FUNCTION affiche_rat(t_rat : r);
  BEGIN
    WRITE(r.numerateur:1:0,'/',r.denominateur:1:0);
  END;

  void FUNCTION add_rat(t_rat :r1; t_rat : r2; t_rat : VAR r3);
  BEGIN
    r3.numerateur := (r1.numerateur * r2.denominateur) +
                     (r2.numerateur * r1.denominateur);
    r3.denominateur := r1.denominateur * r2.denominateur;
  END;

BEGIN
  init_rat(1,3,r1); init_rat(2,5,r2);
  affiche_rat(r1); WRITE(\n); affiche_rat(r2); WRITE(\n);
  add_rat(r1,r2,r3);
  affiche_rat(r3); WRITE(\n);
END.
```

Ce programme illustre la manipulation simple d'enregistrements. A l'exécution, nous obtenons :

```
Exécution en cours...
1/3
2/5
11/15

Exécution terminée
```

Le concept d'enregistrement permet d'*encapsuler* des données ayant un lien logique entre elles dans une unité. Une conséquence intéressante est que l'on diminue de ce fait le nombre d'arguments lors d'un appel de fonction et une multiplicité d'identificateurs. Voici une illustration de cette idée pour le programme testant l'égalité de deux tableaux :

```
PROGRAM egalite;
CONST maximum = 99;
TYPE  t_tableau = ARRAY[maximum] OF numeric;
TYPE  t_tab = RECORD
        t_tableau : tab;
        int : nbre_elem;
      END;
VAR   int : max;
VAR   t_tab : a,b;

  void FUNCTION saisie_tableau(t_tab : VAR t; int : nbre_elem);
  VAR int : i;
  BEGIN
    FOR i := 1 UNTIL i > nbre_elem BY 1 DO BEGIN
      WRITE("Elément ",i:1:0, " : ");
      READ(t.tab[i]);
    END;
    t.nbre_elem := nbre_elem;
  END;

  void FUNCTION affiche_tableau(t_tab : t);
  VAR int : i;
  BEGIN
    FOR i := 1 UNTIL i > t.nbre_elem BY 1 DO WRITE("t[",i:1:0,"]=",t.tab[i]," ");
    WRITE(\n);
  END;

  int FUNCTION egalite_tableau(t_tab : a; t_tab : b);
  CONST vrai = 1;
  CONST faux = 0;
  VAR int : i,egalite;
  BEGIN
    i := 1;
    egalite := vrai;
    WHILE egalite AND (i <= a.nbre_elem) DO
      IF a.tab[i] = b.tab[i]
         THEN i := i+1
         ELSE egalite := faux;
    RETURN egalite;
  END;

BEGIN
  WRITE("Nombre d'éléments d'un tableau <=",maximum," : ");READ(max);
  saisie_tableau(a,max);
  affiche_tableau(a);
  saisie_tableau(b,max);
  affiche_tableau(b);
  IF egalite_tableau(a,b)
     THEN WRITE("Les deux tableaux sont égaux",\n)
     ELSE WRITE("Les deux tableaux sont différents",\n);
END.
```

Exercices

- Tous les programmes doivent être réalisés en langage LERIA.
- N'oublier pas de bien les commenter. Cette opération se fait tout au long du développement d'un programme et non à la fin de ce processus !

1) Compléter le programme suivant de manière à ce qu'il produise les résultats indiqués à droite (les champs x, y, t de r1 puis x, y et t de r2 sont affichés en séquence).

```
PROGRAM test_record;
TYPE tab1 = ARRAY[3] OF numeric;
TYPE rec = RECORD
       numeric : x,y;
       tab1 : t;
     END;
TYPE couple = RECORD
       rec : r1,r2;
     END;
TYPE tab2 = ARRAY[4] OF couple;

VAR tab2 : t;

BEGIN
  { initialisation de t }
  { affichage de t }
END.
```

```
Exécution en cours...
t[1]=
0 1 2 3 4
5 6 7 8 9
t[2]=
10 11 12 13 14
15 16 17 18 19
t[3]=
20 21 22 23 24
25 26 27 28 29
t[4]=
30 31 32 33 34
35 36 37 38 39

Exécution terminée
```

Les tableaux multidimensionnels

Les tableaux multidimensionnels

- Dans beaucoup de problèmes il est intéressant de travailler avec des tableaux multidimensionnels, c'est-à-dire des tableaux de tableaux.
- Le plus simple des tableaux multidimensionnels est le tableau à deux dimensions.

Dans un tableau à deux dimensions, chaque élément du tableau est en fait un tableau à une dimension ! Evidemment, chaque élément de ce tableau peut être accédé comme nous l'avons vu précédemment.

Pour définir un tableau multidimensionnel, il suffit de définir un tableau dont les éléments sont des tableaux.

Les tableaux multidimensionnels

- considérons l'exemple suivant :

```
TYPE tab1 = ARRAY[3] OF numeric;
TYPE tab2 = ARRAY[2] OF tab1;
VAR tab2 : t;
```

- *t* est un tableau à deux dimensions composé de 2 lignes et 3 colonnes
- *t*[1] est la première ligne de *t*
- *t*[1][3] est le dernier élément de la première ligne de *t*

Graphiquement, le tableau *t* peut être représenté ainsi :

t	*1*	*2*	*3*
t[1]	*t*[1][1]	*t*[1][2]	*t*[1][3]
t[2]	*t*[2][1]	*t*[2][2]	*t*[2][3]

L'accès aux éléments d'un tableau multidimensionnel se fait en indiquant les indices dans l'ordre qui va du tableau le plus "externe" vers le plus "interne".

Les tableaux multidimensionnels

- Pour illustrer ce nouveau concept, nous écrivons un programme qui initialise, puis affiche un tableau à deux dimensions. Ses valeurs sont égales à la somme des indices (ligne, colonne) des éléments.
- Exemple (2 lignes et 3 colonnes) :

	1	*2*	*3*
1	2	3	4
2	3	4	5

Dans l'exemple présenté, le nombre de lignes et de colonnes est fixé arbitrairement à 3, respectivement 4.

Nous parcourons le tableau, ligne par ligne (boucle externe sur *i*). Pour chaque ligne, nous initialisons chaque élément de la ligne (boucle interne sur *j*).

L'affichage du tableau a la même structure algorithmique.

```
{ initialisation }
FOR i := 1 UNTIL i > 2 BY 1 DO
  FOR j := 1 UNTIL i > 3 BY 1 DO t[i][j] := i+j;

{ affichage }
FOR i := 1 UNTIL i > 2 BY 1 DO BEGIN
  FOR j := 1 UNTIL i > 3 BY 1 DO WRITE(t[i][j]:1:0,' ');
  WRITE(\n);
END;
```

Exercices

- Tous les programmes doivent être réalisés en langage LERIA.
- N'oublier pas de bien les commenter. Cette opération se fait tout au long du développement d'un programme et non à la fin de ce processus !

1) Modifier l'exemple du cours de manière à ce que l'utilisateur puisse entrer au moment de l'exécution le nombre de lignes et de colonnes.

2) On donne un tableau à deux dimensions *t* de *n* lignes et de *n* colonnes. Ecrire un sous-programme qui transpose le tableau passé comme paramètre, c'est-à-dire dans lequel les colonnes deviennent les lignes, et cela sans utiliser un tableau d'aide. Exemple :

	1	2	3			1	4	7
avant *t* =	4	5	6		après *t* =	2	5	8
	7	8	9			3	6	9

3) On donne un tableau carré *t* de *n* lignes et *n* colonnes. Ecrire une fonction booléenne vérifiant que le tableau est symétrique, c'est-à-dire que pour tout (*i*, *j*) dans le domaine du tableau, on a $t[i][j] = t[j][i]$.

Méthodologie de la programmation

La récursion

Un programme est dit récursif s'il contient un sous-programme qui s'appelle lui-même, directement ou indirectement. Par la suite, nous ne considérons que le cas d'appels récursifs directs.

Voici un sous-programme *prog* contenant un seul appel récursif direct :

```
sous-programme prog
        …
        prog
        …
fin sous-programme prog
```

Le problème qui vient tout de suite à l'esprit est celui de la terminaison de ce processus. Nous sommes en présence d'un sous-programme qui, avant la fin de son exécution, s'appelle de nouveau provoquant, par conséquent, une nouvelle exécution du même sous-programme, etc. En réfléchissant un peu, il devient clair que le sous-programme récursif doit avoir une structure telle que, à un certain moment au bout du *n*ième appel, il n'y ait plus de nouvel appel récursif.

Le cas pour lequel la solution du problème est immédiate et ne nécessite plus d'appel récursif est appelé le *cas de base*.

Dans l'autre cas de figure, il doit être possible d'exprimer le cas général en fonction d'un cas plus simple se rapprochant du cas de base. C'est ce que l'on appelle le *cas inductif*.

```
sous-programme prog
        si cas-de-base alors … { plus d'appel récursif }
        sinon { cas inductif }
                …
                prog
                …
        finsi
fin sous-programme prog
```

Considérons l'exemple archiclassique de la factorielle d'un entier positif :

$$factorielle(n) = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n\,.$$

Elle peut être définie ainsi :

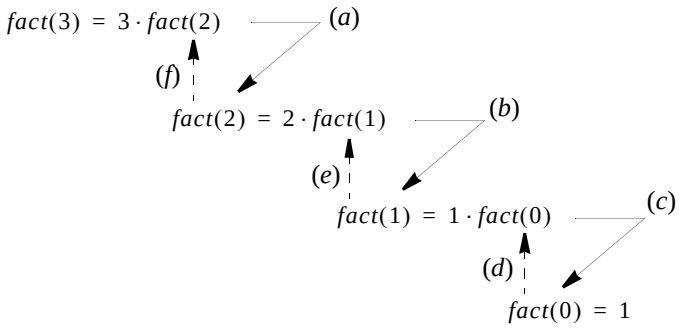
$fact(n) = n \cdot fact(n-1)$ avec n entier > 0
$fact(0) = 1$

Cette définition du concept de factorielle est récursive car *fact* est défini en fonction de *fact*. Néanmoins le cas inductif assure qu'à un certain moment le cas de base $n = 0$ sera atteint du fait que *fact* (n) est défini à l'aide d'un cas « plus simple » $fact(n-1)$ se rapprochant du cas de base, si n est un entier > 0.

C'est la réalisation du cas de base qui assure généralement la terminaison de l'algorithme récursif.

Afin de mieux comprendre le fonctionnement d'un algorithme récursif, simulons l'exécution du calcul de la factorielle de 3.

Pour calculer *fact*(3), il faut multiplier 3 avec *fact*(2). Comme la valeur de *fact*(2) n'est pas encore connue, on suspend la multiplication pour effectuer *fact*(2). C'est l'étape (*a*) dans le schéma ci-dessous.



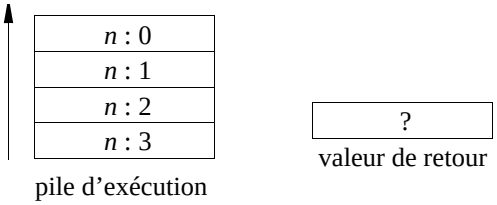
Ce processus doit être répété (*b*) et (*c*) jusqu'à la réalisation du cas de base. La récursion s'arrête avec *fact*(0) valant 1. Comme il n'y a plus d'appel récursif, l'exécution continue juste après l'instruction ayant provoqué le dernier appel récursif. C'est l'étape (*d*), et maintenant *fact*(1) peut être calculé pour donner comme valeur 1. L'exécution continue avec l'étape (*e*) qui permet d'évaluer *fact*(2) donnant 2, provoquant (*f*) et finalement l'évaluation de *fact*(3), c'est-à-dire $3 \cdot 2 = 6$.

Ce qu'il est important de remarquer dans cet exemple, c'est que des « calculs » sont laissés en suspens au moment de l'exécution de l'appel récursif et que, lorsque l'exécution finit par continuer à cet endroit, les valeurs laissées en suspens sont retrouvées intactes.

La structure de données utilisée de manière interne par le système est ce que l'on appelle la pile d'exécution. Avant d'effectuer l'appel récursif, les variables locales au sous-programme sont sauvées automatiquement sur la pile et, lorsque l'appel récursif est terminé, ces valeurs sont restaurées automatiquement.

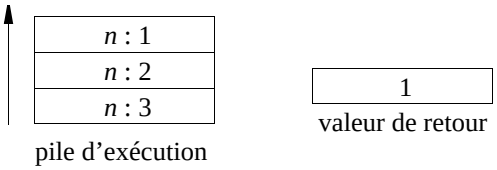
```
PROGRAM fonction_factorielle;
VAR int : f;
  int FUNCTION fact(int : n);
  BEGIN
    IF n = 0 THEN RETURN 1
             ELSE RETURN n*fact(n-1);  { appel récursif }
  END;
BEGIN
  WRITE("Entrez un entier : "); READ(f);
  WRITE(f,"! = ",fact(f):1:0);
END.
```

Lors de chaque appel de *fact*(), une nouvelle instance de *n* est créée. A la fin de l'exécution de la fonction, ces variables sont automatiquement enlevées du sommet de la pile. Après quatre appels successifs de *fact*(), la situation est la suivante :

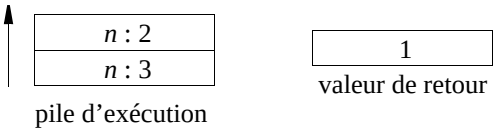


Sur cette figure, la pile croît du bas vers le haut. Les valeurs sont uniquement empilées sur le sommet de la pile, et le dernier élément empilé sera le premier à pouvoir être enlevé.

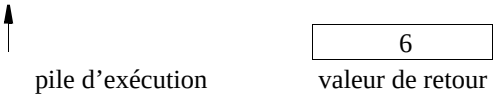
Il y a bien quatre valeurs de *n* qui sont conservées sur la pile, avec la valeur 0 au sommet de la pile. Puisque *n* vaut 0, la valeur 1 est retournée, et l'exécution de cette instance de *fact*() se termine. De ce fait, toutes les variables locales à l'exécution de cette instance de *fact*() sont détruites (ici, il n'y a que l'argument formel *n*)) !



A ce moment, la variable *n* manipulée par le programme est bien celle de son instance qui vaut 1 puisque le retour dans l'instance appelante a provoqué le dépilement de l'élément au sommet de la pile. La situation de la pile d'exécution à ce moment est illustrée sur la figure ci-dessus. Après l'instruction de calcul, l'exécution de cette instance de *fact*() se termine, provoquant le dépilement du sommet de la pile et le retour dans le programme appelant à l'instruction suivant l'instruction d'appel, donc à l'instruction de calcul.



La pile d'exécution et la valeur de retour évoluent de la façon suivante :



Lorsque l'exécution de la dernière instance de *fact*() se termine et que l'on retourne dans le programme appelant, c'est-à-dire *fonction_factorielle*, l'expression *fact(f)* a bien la valeur voulue, et la pile d'exécution est vide.

L'exemple du programme calculant la factorielle d'un nombre n'est évidemment pas un exemple démontrant l'utilité de la récursion. Néanmoins, cet exemple très simple permet de comprendre la mise en œuvre en LERIA et les mécanismes impliqués dans le bon déroulement de la récursion.

L'exemple suivant est un exemple de programme qu'il est plus difficile de réaliser de façon non récursive mais dont la solution récursive est immédiate.

1 Le problème des tours de Hanoi

Ce problème est le suivant : dans un pays lointain il y a un monastère dans lequel 64 anneaux (disques) sont empilés par ordre de taille décroissante sur un pieu que nous appellerons le pieu « A ». Le travail des moines du monastère consiste à déplacer les 64 anneaux de ce pieu vers un pieu « B » en utilisant un pieu intermédiaire, le pieu « C ». Les mouvements autorisés sont le déplacement d'un anneau du sommet de la pile sur un anneau de taille supérieure d'un autre pieu ou sur un pieu vide. La légende veut que la fin du monde coïncide avec la fin de ce travail.

Sur la figure 1, le lecteur trouvera une illustration du travail à faire en ne considérant que trois anneaux.

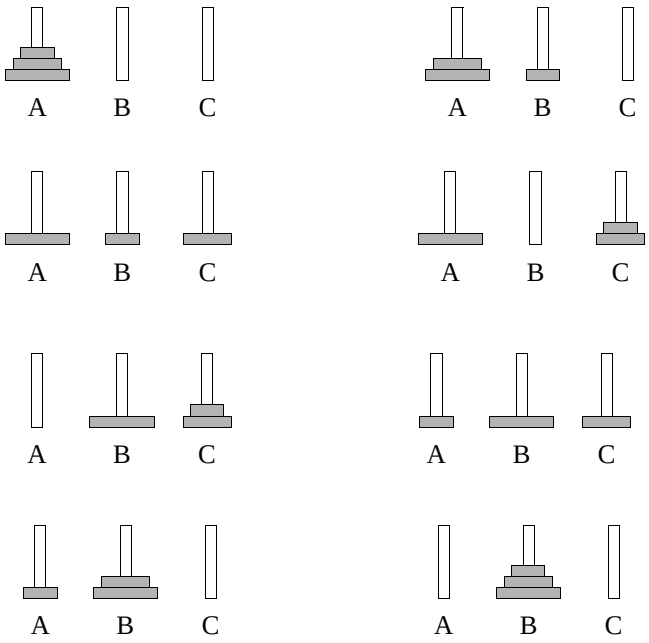


Figure 1. - Les déplacements à effectuer pour résoudre le problème avec trois disques.

La solution récursive s'énonce assez simplement : pour déplacer *n* disques du pieu *A* vers le pieu *B*, il suffit de déplacer *n*–1 disques du pieu *A* vers le pieu *C*, puis de déplacer le disque restant du pieu *A* vers le pieu *B* et enfin déplacer les *n*–1 disques du pieu *C* vers le pieu *B*.

Ainsi, pour résoudre un problème de taille *n*, nous nous ramenons à un problème de taille *n*–1. Ce processus est fini car il arrivera forcément le moment où *n* vaudra 1. A ce moment la solution est immédiate : il suffit de déplacer ce dernier anneau vers sa destination. Nous avons donc bien dégagé le cas inductif et le cas de base. Le sous-programme récursif *hanoi* s'écrit :

sous-programme *deplace*(*n*, *a*, *b*, *c*)
  si *n* = 1
    alors afficher "déplacer le disque de", *a*, "vers", *b*
    sinon
      deplace(*n*–1, *a*, *c*, *b*)
      afficher "déplacer le disque de", *a*, "vers", *b*
      deplace(*n*–1, *c*, *b*, *a*)
  finsi
fin sous-programme *deplace*

Les paramètres de ce sous–programme sont passés par valeur. Si on est en présence du cas de base, il suffit de déplacer l'unique anneau de *A* vers le pieu *B*. Dans le cas contraire, *n*–1 anneaux sont déplacés récursivement du pieu *A* vers le pieu *C* en s'aidant du pieu *B*. Lorsque ce travail est réalisé, l'anneau restant est déplacé du pieu A vers le pieu *B* et ensuite les *n*–1 anneaux sont redéplacés récursivement du pieu *C* vers le pieu *B* en s'aidant du pieu *A*.

```
PROGRAM hanoi;
CONST ndisksmax = 8;
VAR int : ndisks,nombre_depl;
{$sd}
void FUNCTION move(int : n; char : a;char : b; char : c;
                   int : VAR nbre);
BEGIN
  IF n > 0 THEN BEGIN
     move(n-1,a,c,b,nbre);
     write(\n,"Déplacer un plateau de ",a:2:0," vers ",b:2:0);
     nbre := nbre+1;
     move(n-1,c,b,a,nbre);
  END;
END;

BEGIN
  write(\n,"TOURS DE HANOI !!");
  REPEAT
    write(\n,"Combien de disques ? < 9 ");
    read(ndisks);
  UNTIL ndisks < 9;
  nombre_depl := 0;
  move(ndisks,'X','Y','Z',nombre_depl);
  write(\n,"Il a y eu",nombre_depl:4:0," déplacements de disques!");
END.
```

Il n'est pas évident de suivre mentalement l'exécution d'un tel programme. Une exécution symbolique donne :

deplace(3, a, b, c) = *deplace*(2, a, c, b); a → b; *deplace*(2, c, b, a)

deplace(2, a, c, b) = *deplace*(1, a, b, c); a → c; *deplace*(1, b, c, a)

deplace(1, a, b, c) = a → b

deplace(1, b, c, a) = b → c

deplace(2, c, b, a) = *deplace*(1, c, a, b); c → b; *deplace*(1, a, b, c)

deplace(1, c, a, b) = c → a

deplace(1, a, b, c) = a → b

où a → b signifie que le disque du sommet doit être déplacé du pieu *a* vers le pieu *b*. En remplaçant les différents déplacements dans l'expression initiale *deplace(3, a, b, c)* , on obtient comme suite de sept déplacements :

a → b; a → c; b→ c; a → b; c → a; c → b; a → b

Intéressons-nous un moment au nombre de déplacements à effectuer pour estimer la date de la fin du monde. Soit *d*(*n*) le nombre de déplacements à effectuer pour résoudre le problème avec *n* disques.

On a : $\begin{cases} d(1) = 1 \\ d(n) = 2 \cdot d(n-1) + 1 \end{cases}$ donc $d(n) = 2^n - 1$

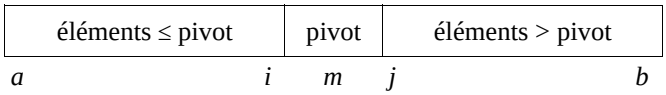
Nous sommes en présence d'un problème où la solution est de complexité exponentielle ! En supposant que les moines travaillent à une vitesse diabolique où le déplacement d'un disque ne nécessite que 1 seconde et qu'ils travaillent nuit et jour, il leur faut $2^{64} - 1$ secondes pour terminer le travail, c'est-à-dire approximativement $2^4 \cdot 2^{60} \approx 2^4 \cdot (10^3)^6 = 2^4 \cdot 10^{18}$ secondes. Comme une année se compose d'un peu plus de $3 \cdot 10^7$ secondes, la durée du travail s'étend sur environ $5 \cdot 10^{11}$ années, soit plus de 500 milliards d'années. L'âge actuel de l'univers est estimé à 15 milliards d'années ! Cet exemple montre qu'il existe des problèmes pour lesquels on a une solution, mais que cette solution n'est pas pratiquable même avec les ordinateurs les plus puissants. Supposons que nous soumettions notre programme des tours de Hanoi à un supercalculateur capable d'effectuer 10^9 (un milliard) de déplacements à la seconde. Comme il y a environ $2^4 \cdot 10^{18}$ déplacements à effectuer, il lui faut quand même encore 16 milliards de secondes, soit plus de 500 années pour résoudre le problème posé avec 64 disques !

2 Un algorithme de tri récursif : tri rapide

L'exemple du programme récursif donnant la solution au problème des tours de Hanoi peut prêter à sourire puisque, en fin de compte, il est inutile. Mais tout le travail que nous avons réalisé sur cet exemple de jeu peut être appliqué sur un programme beaucoup plus intéressant et pratique : un programme de tri. Cette

méthode de tri est d'ailleurs une des méthodes les plus rapides sinon la plus rapide que l'on ait inventée.

Le principe du tri rapide est de partitionner le tableau à trier en trois parties en fonction d'un élément pivot.



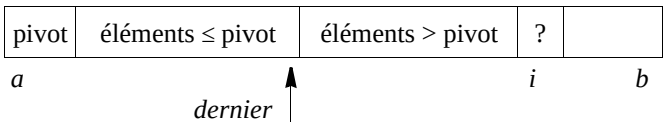
Il suffit de réappliquer ce principe récursivement à chacun des deux sous-tableaux pour obtenir à la fin un tableau trié. La récursion s'arrête lorsque le sous-tableau à partitionner contient un élément.

Cela nous donne le corps de programme suivant :

```
sous-programme trirapide(tableau, a, b)
        si b > a alors
                partition(tableau, a, b, i, j)
                trirapide(tableau, a, i)
                trirapide(tableau, j, b)
        finsi
fin sous-programme trirapide
```

Le plus gros du travail est fait par le sous-programme *partition*() qui doit d'abord choisir l'élément pivot. Ce choix est arbitraire, mais pour que le tri soit effectivement rapide, il faudrait que le pivot subdivise le sous-tableau en deux parties plus ou moins égales[1] et que le choix du pivot soit lui-même rapide. Une façon simple de procéder est de prendre chaque fois le premier ou le dernier élément du sous-tableau, mais si le tableau est déjà « presque » trié, on risque de choisir une mauvaise valeur comme pivot. Pour éviter cela, nous choisissons comme pivot l'élément se trouvant au milieu du sous-tableau à partitionner.

Pour arriver à la partition du sous-tableau, il est clair qu'il faut examiner tous les éléments du sous-tableau. Si, avant de commencer l'examen de chacun des éléments du sous-tableau, on échange l'élément pivot avec le premier élément du tableau, il faudra examiner tous les éléments, du deuxième au dernier. Supposons le problème partiellement résolu. Au moment de l'examen de l'élément *i*, nous avons la situation suivante où *dernier* est l'indice du dernier élément inférieur ou égal au pivot :

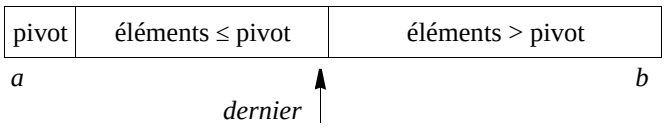


1. En fait, Tri Rapide (*Quicksort*) est un des tris les plus performants avec un nombre de comparaisons de l'ordre de $n \cdot \log_2 n$, n étant le nombre d'éléments à trier. Mais il peut dégénérer si le pivot est chaque fois l'élément le plus grand ou le plus petit du tableau à trier. Le nombre de comparaisons devient alors de l'ordre de n^2 .

Si l'élément d'indice *i* est strictement supérieur au pivot, il suffit de passer à l'élément suivant, c'est-à-dire
$i \leftarrow i + 1$.

Si l'élément d'indice *i* est inférieur ou égal au pivot, il faut échanger l'élément d'indice *dernier*+1 avec l'élément d'indice *i* et incrémenter *dernier.* Ensuite on passe à l'élément suivant, c'est-à-dire $i \leftarrow i + 1$.

En procédant de la sorte, nous sommes sûrs d'avoir, à la fin de l'exécution de la boucle, la situation suivante :



Il suffit dès lors de permuter l'élément d'indice *dernier* avec l'élément d'indice *a*, c'est-à-dire le pivot. Nous avons alors l'effet net désiré.

Voici un programme LERIA complet :

```
{$s} {$r-}
PROGRAM test_tri_rapide;
CONST maximum  = 4000;
TYPE t_vecteur = ARRAY[maximum] OF numeric;
TYPE t_tableau = RECORD
                   t_vecteur : t;
                   int       : max;
                 END;
VAR t_tableau : t;
VAR int : t1,t2;

  void FUNCTION tri_rapide(t_tableau : VAR t; int : a; int : b);
  VAR int : i, j;

    void FUNCTION partition(t_tableau : VAR tab; int : a; int : b;
                            int : VAR i; int : VAR j);
    VAR int : m,dernier;
    VAR numeric : pivot;

      void FUNCTION echanger(numeric : VAR x; numeric : VAR y);
      VAR numeric : tmp;
      BEGIN
        tmp := x; x := y; y := tmp;
      END;

   BEGIN
     m := (a+b) % 2;
     echanger(tab.t[a],tab.t[m]);
     pivot := tab.t[a];
     dernier := a;
     FOR i := a+1 UNTIL i > b BY 1 DO
          IF tab.t[i] > pivot
             THEN SKIP
             ELSE BEGIN
                echanger(tab.t[dernier+1],tab.t[i]);
                dernier := dernier + 1;
             END;
     echanger(tab.t[dernier],tab.t[a]);
```

```
    i := dernier - 1;
    j := dernier + 1;
  END;

  BEGIN
    IF b > a THEN BEGIN
       partition(t,a,b,i,j);
       tri_rapide(t,a,i);
       tri_rapide(t,j,b);
    END;
  END;

  void FUNCTION cree_tableau(t_tableau : VAR t);
  VAR int : k;
  BEGIN
    randomize(time());
    WRITE("Entrez le nombre d""éléments du tableau <= ",maximum:5:0," ");
    READ(t.max);
    FOR k := 1 UNTIL k > t.max BY 1 DO t.t[k] := intnum(rand());
  END;

  void FUNCTION affiche_tableau(t_tableau : VAR t);
  VAR int : k;
  BEGIN
    FOR k := 1 UNTIL k > t.max BY 1 DO BEGIN
      WRITE(\n,"t[",k:2:0,"]=");
      WRITE(t.t[k]:6:1);
    END;
    WRITE(\n);
  END;

  void FUNCTION affiche_temps_mis(int : t1; int : t2);
  VAR int : secondes ;
  BEGIN
    secondes := t2 - t1;
    WRITE("Temps mis : ",secondes % 3600 : 3:0," h ",(secondes#3600)%60,
          " m ",((secondes#3600)#60), " s",\n);
  END;
BEGIN
  cree_tableau(t);
  affiche_tableau(t);
  t1 := time();
  tri_rapide(t,1,t.max);
  t2 := time();
  affiche_tableau(t);
  affiche_temps_mis(t1,t2);
END.
```

Comparons les temps d'exécution de ce programme avec ceux du tri par sélection.

taille du vecteur	tri par sélection	tri rapide
100	0h0m0s	0h0m0s
200	0h0m2s	0h0m1s
400	0h0m7s	0h0m1s
800	0h0m26s	0h0m1s
1600	0h1m45s	0h0m3s

Ces résultats ont évidemment été obtenus sur le même ordinateur dans les mêmes conditions !

Les détails de la génération automatique du tableau et de la mesure du temps mis sont expliqués en détail dans l'exercice de synthèse 1.

3 Exercices à faire

1) Ecrire un programme qui détermine le nombre de fois qu'un nombre entier peut être décomposé en sommes différentes d'entiers. L'ordre n'a pas d'importance. Exemple : 5 a sept décompositions, à savoir 5, 4 + 1, 3 + 2, 3 + 1 + 1, 2 + 2 + 1, 2 + 1 + 1 + 1, 1 + 1 + 1 + 1 + 1.

Idée de solution :

Le problème consiste donc à déterminer le nombre de fois qu'un nombre entier positif m peut être décomposé à l'aide d'entiers ð n. Soit *partition*(m, n) ce nombre qu'il faut calculer. Il peut être caractérisé de la manière suivante :

- *partition*(m, 1) = 1 comme il n'existe qu'une seule façon d'écrire un nombre entier positif à l'aide d'une somme de 1 ;
- *partition*(1, n) = 1 car le nombre 1 n'admet qu'une seule décomposition, à savoir 1 = 1 ;
- *partition*(m, n) = *partition*(m, m) si $n > m$. En effet, la décomposition d'un nombre m ne peut contenir un nombre $n > m$!
- *partition*(m, m) = 1 + *partition*(m, $m - 1$) du fait de la partition $m = m$. Les autres partitions ne font intervenir que des nombres inférieurs ou égaux à $m - 1$;
- *partition*(m, n) = *partition*(m, $n - 1$) + *partition*($m - n$, n) si $n < m$. Dans cette somme, *partition*(m, $n - 1$) représente le nombre de partitions quand n ne figure pas dans la partition. Si, par contre, n figure dans la partition, alors le nombre restant $m - n$ peut être partitionné de *partition*($m - n$, n) manières différentes.

2) Ecrire un programme qui affiche la solution au problème classique des n reines. On a un échiquier n sur n, et le but du jeu est de placer n dames sur les cases de l'échiquier de manière à ce qu'aucune dame ne menace une autre.

Voici une solution possible pour un échiquier de taille 5.:

	1	*2*	*3*	*4*	*5*
1	R				
2				R	
3		R			
4					R
5			R		

3) Que calcule le programme LERIA suivant ?:

```
PROGRAM fonction_91;
VAR int : n;
int FUNCTION f(int : x);
  BEGIN
    IF x > 100
       THEN RETURN x-10
       ELSE RETURN f(f(x+11));
  END;
BEGIN
  write("Entrez un entier : "); read(n);
  write("Fonction(",n:4:0,") = ",f(n));
END.
```

Méthodologie de la programmation

Exercices de synthèse

Le but de ce chapitre est d'illustrer tous les concepts étudiés dans le cadre de ce cours sur des exemples plus complexes.

1 Tri d'un tableau

Ecrire un programme qui trie un tableau (vecteur) de nombres. Le nombre effectif de nombres est fourni par l'utilisateur au moment de l'exécution. Les nombres sont à générer aléatoirement par l'ordinateur. Mesurer le temps d'exécution du programme obtenu (en fonction du nombre d'éléments à trier).

Il existe beaucoup de méthodes pour trier un vecteur, des méthodes très efficaces et des méthodes moins efficaces. Nous présentons ici une solution non optimale, mais qui a l'avantage d'être simple : c'est le tri par sélection.

1.1 Principe et algorithme

Le tri par sélection fonctionne d'après le principe suivant :

- sélectionner l'élément du tableau ayant la plus grande clé ;
- échanger cet élément avec le dernier élément du sous-tableau non encore trié. Un élément vient d'être trié. Appliquer la méthode sur le sous-tableau restant non encore trié.

Voici un exemple où *max*, c'est-à-dire le nombre d'éléments effectif dans le tableau à trier, vaut 5.

	1	2	3	4	5
le tableau *t*[1:5] initial est non trié	44	55	12	42	18
i=4, le sous-tableau *t*[5:5] est trié	44	18	12	42	*55*
i=3, le sous-tableau *t*[4:5] est trié	42	18	12	*44*	*55*
i=2, le sous-tableau *t*[3:5] est trié	12	18	*42*	*44*	*55*
i=1, le tableau *t*[2:5] est trié et, par conséquent, *t*[1:5] est trié	12	*18*	*42*	*44*	*55*

L'algorithme peut s'écrire :

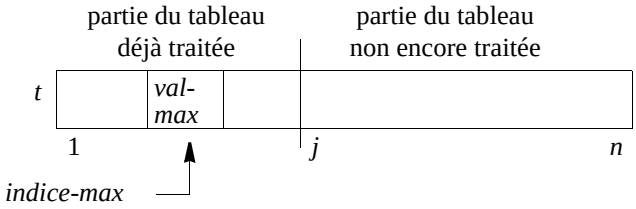
```
i ← max
tant que i > 1 faire
    { chercher l'indice de l'élément ayant la plus grande clé dans le sous-tableau non trié t[1:i]
      et renvoyer cette position dans pos_max}
    sélectionner(i, pos_max)
    échanger(t[i],t[pos_max])
    i ← i – 1
    { le tableau t[i+1:max] est trié }
fintantque
```

Cet algorithme se termine car *i* valant initialement *max* est décrémenté à chaque tour de boucle. A la fin de l'exécution, *i* vaut 1 et, par conséquent, le sous-tableau *t*[2:*max*] est trié. Comme l'élément en position 1 est à ce moment l'élément le plus petit du tableau, tout le tableau est trié.

Le sous-programme *echanger*() ne mérite pas de commentaires supplémentaires car nous avons déjà vu l'opération d'échange du contenu de deux variables à plusieurs reprises.

Il ne reste qu'à raffiner *selectionner*() qui possède deux paramètres formels que nous appelons *n* et *indice-max*. *n* indique l'indice du dernier élément du tableau qu'il faut examiner et, au retour, *indice-max* contiendra l'indice du plus grand élément du tableau *t*[1:*n*].

Tous les éléments de *t*[1:*n*] doivent être examinés : il faut utiliser une boucle. Cette boucle peut être caractérisée par :



j est l'indice du premier élément non encore traité. *indice-max* représente l'indice de l'élément ayant la plus grande valeur dans le sous-tableau déjà traité et *val-max* la valeur de l'élément ayant la plus grande valeur dans le sous-tableau déjà traité.

Pour maintenir cet invariant de boucle, il suffit d'écrire comme corps de la boucle :

```
si t[j ] > val-max alors
    val-max ← t[j ]
    indice-max ← j
finsi
j ← j + 1
```

La boucle s'arrête si *j* > *n*, c'est-à-dire lorsque tous les éléments du tableau ont été examinés. L'initialisation la plus simple consiste à prendre le premier élément du tableau comme élément le plus grand :

```
val-max ← t[1]
indice-max ← 1      { le maximum se trouve initialement en position 1}
j ← 2               { l'examen commence avec l'élément 2}
```

Il nous reste le problème de faire générer par l'ordinateur lui-même les jeux de données nécessaires. Dans le cas de notre programme de tri, il peut être intéressant de mesurer le comportement du programme avec des valeurs toutes identiques. Cela ne pose pas de problèmes particuliers. De même, il pourrait être intéressant de remplir le tableau par une suite croissante ou décroissante de nombres pour étudier des cas spéciaux importants. Il est, par ailleurs, un peu plus difficile de faire générer par l'ordinateur des données « au hasard ». Nous allons nous servir de la

fonction prédéfinie *rand*() qui génère des nombres pseudo-aléatoires compris entre 0 et 1. Ils sont appelés pseudo-aléatoires car ils sont en fait calculés d'une façon tout à fait déterministe, contrairement à ce que le mot aléatoire laisse croire. Bien que ces nombres soient pseudo-aléatoires, ils ont les propriétés statistiques de nombres aléatoires, et c'est cela qui compte.

La méthode le plus couramment utilisée pour générer des nombres pseudo-aléatoires est celle que l'on appelle : la méthode linéaire congruentielle. Dans cette méthode, chaque nombre r_k de la séquence est calculé en se basant sur le nombre prédécesseur r_{k-1} lié à r_k par la relation de récurrence :

$$r_k = (a \cdot r_{k-1} + c) \text{modulo } m$$

Le choix de *a*, *c* et *m* est délicat. Si *m* = 65 536, *a* = 25 173 et *c* = 13 849, alors la séquence de nombres obtenue est une séquence de 65 536 nombres différents, et ensuite la séquence se répète.

Pour que le calcul de ces nombres soit possible, il faut une valeur initiale r_0 appelée graine (seed). La fonction prédéfinie *randomize*() permet justement de faire cela. Si la graine n'est pas modifiée à chaque exécution, les nombres pseudo-aléatoires générés seront toujours les mêmes. Ceci est intéressant lors de la mise au point d'un programme, mais généralement fort gênant autrement. C'est pour cela que nous utiliserons la fonction *time*() pour initialiser le générateur dans la fonction *cree_tableau*().

Voici pour terminer le programme complet *Leria* réalisant le travail demandé :

```
PROGRAM tri_par_selection;
CONST maximum  = 1000;
TYPE t_vecteur = ARRAY[maximum] OF int;
TYPE t_tableau = RECORD
                   t_vecteur : t;
                   int   : max;
                 END;
VAR t_tableau : t;
  void FUNCTION tri_selection(t_tableau : VAR t);
  VAR int : i,pos_max;

    void FUNCTION echanger(int : VAR x; int : VAR y);
    VAR int : tmp;
    BEGIN
      tmp := x; x := y; y := tmp;
    END;

    void FUNCTION selectionne(t_tableau : VAR t; int : n;
                                              int : VAR indice_max);
    VAR int : val_max, i;
    BEGIN
      val_max := t.t[1];
      indice_max := 1;
      FOR i := 2 UNTIL i > n BY 1 DO BEGIN
        IF t.t[i] > val_max THEN BEGIN
           val_max := t.t[i];
           indice_max := i;
        END;
      END;
    END;
```

```
  BEGIN { tri_selection }
    i := t.max;
    WHILE i > 1 DO BEGIN
      selectionne(t,i,pos_max);
      echanger(t.t[i],t.t[pos_max]);
      i := i-1;
    END;
  END;
  void FUNCTION cree_tableau(t_tableau : VAR t);
  VAR int : k;
  BEGIN
    randomize(time());
    WRITE("Entrez le nombre d'éléments du tableau <= ",maximum:5:0," ");
    READ(t.max);
    FOR k := 1 UNTIL k > t.max BY 1 DO t.t[k] := rand();
  END;
  void FUNCTION affiche_tableau(t_tableau : VAR t);
  VAR int : k;
  BEGIN
    FOR k := 1 UNTIL k > t.max BY 1 DO BEGIN
      WRITE(\n,"t[",k,"]=");
      WRITE(t.t[k]);
    END;
  END;
BEGIN
  cree_tableau(t);
  write(\n,"Voici le tableau non trié :");
  affiche_tableau(t);
  tri_selection(t);
  write(\n,"Voici le tableau trié :");
  affiche_tableau(t);
END.
```

Pour mesurer le comportement de l'algorithme de tri, nous modifions le programme ci-dessus de manière à ce qu'il affiche le temps mis pour trier le vecteur de nombres. Nous supposons que nous travaillons dans un environnement mono-tâche, mono-utilisateur : ainsi le temps mis aura été quasi exclusivement été réservé au programme de tri.

Voici le programme modifié :

```
PROGRAM tri_par_selection; {$r-}
CONST maximum  = 8000;
TYPE t_vecteur = ARRAY[maximum] OF int;
TYPE t_tableau = RECORD
                   t_vecteur : t;
                   int  : max;
                 END;
VAR t_tableau : t;
VAR int: t1,t2;

  void FUNCTION tri_selection(t_tableau : VAR t);
  VAR int: i,pos_max;

    void FUNCTION echanger(int: VAR x; int: VAR y);
    VAR int: tmp;
    BEGIN
      tmp := x; x := y; y := tmp;
    END;
    void FUNCTION selectionne(t_tableau : VAR t; int: n;
                                                 int: VAR indice_max);
```

```
    VAR int: val_max, i;
    BEGIN
      val_max := t.t[1];
      indice_max := 1;
      FOR i := 2 UNTIL i > n BY 1 DO BEGIN
        IF t.t[i] > val_max THEN BEGIN
           val_max := t.t[i];
           indice_max := i;
        END;
      END;
    END;

  BEGIN { tri_selection }
    i := t.max;
    WHILE i > 1 DO BEGIN
      selectionne(t,i,pos_max);
      echanger(t.t[i],t.t[pos_max]);
      i := i-1;
    END;
  END;

  void FUNCTION cree_tableau(t_tableau : VAR t);
  VAR int: k;
  BEGIN
    randomize(time());
    WRITE("Entrez le nombre d'éléments du tableau <= ",maximum:5:0," ");
    READ(t.max);
    FOR k := 1 UNTIL k > t.max BY 1 DO t.t[k] := rand();
  END;

  void FUNCTION affiche_tableau(t_tableau : VAR t);
  VAR int: k;
  BEGIN
    FOR k := 1 UNTIL k > t.max BY 1 DO BEGIN
      WRITE(\n,"t[",k,"]=");
      WRITE(t.t[k]);
    END;
  END;

  void FUNCTION affiche_temps_mis(int: t1; int: t2);
  VAR int: secondes ;
  BEGIN
    secondes := t2 - t1;
    WRITE("Temps mis : ",secondes % 3600 : 3:0," h ",
          (secondes#3600)%60, " m ",
          ((secondes#3600)#60), " s",\n);
  END;

BEGIN
  cree_tableau(t);
  t1 := time();
  tri_selection(t);
  t2 := time();
  affiche_temps_mis(t1,t2);
END.
```

En exécutant ce programme pour différentes tailles de vecteur, il affiche les temps d'exécution suivants :

100	0h0m0s
200	0h0m1s

400	0h0m2s
800	0h0m4s
1600	0h0m12s
3200	0h0m46s
6400	0h3m3s

On constate donc empiriquement (on peut le démontrer formellement) que le comportement de cet algorithme de tri est tel que, si le nombre de données *double*, le temps d'exécution *quadruple* ! La complexité en temps d'exécution du tri par insertion est bien de l'ordre du carré du nombre de données à trier.

2 Calcul de l'impôt

Au grand-duché de Luxembourg, le calcul des impôts se fait sur la base de l'article 118 de la loi concernant le budget des recettes et des dépenses de l'Etat, dont voici un extrait pour l'exercice 1994 :

Art. 118. – L'impôt sur le revenu est déterminé, conformément aux dispositions des articles 119 à 122 et 124, sur la base du tarif suivant :

0% pour la tranche de revenu inférieure à 237 600 francs,

10% pour la tranche de revenu comprise entre 237 600 et 346 200 francs,

20% pour la tranche de revenu comprise entre 346 200 et 414 000 francs,

22% pour la tranche de revenu comprise entre 414 000 et 481 200 francs,

24% pour la tranche de revenu comprise entre 481 200 et 548 400 francs,

26% pour la tranche de revenu comprise entre 548 400 et 616 200 francs,

28% pour la tranche de revenu comprise entre 616 200 et 683 400 francs,

30% pour la tranche de revenu comprise entre 683 400 et 750 000 francs,

32% pour la tranche de revenu comprise entre 750 000 et 818 400 francs,

34% pour la tranche de revenu comprise entre 818 400 et 885 600 francs,

36% pour la tranche de revenu comprise entre 885 600 et 952 800 francs,

38% pour la tranche de revenu comprise entre 952 800 et 1 020 000 francs,

40% pour la tranche de revenu comprise entre 1 020 000 et 1 087 200 francs,

42% pour la tranche de revenu comprise entre 1 087 200 et 1 155 000 francs,

44% pour la tranche de revenu comprise entre 1 155 000 et 1 221 600 francs,

46% pour la tranche de revenu comprise entre 1 221 600 et 1 288 800 francs,

48% pour la tranche de revenu comprise entre 1 288 800 et 1 357 200 francs,

50% pour la tranche de revenu dépassant 1 357 200 francs.

Art. 120. – L'impôt à charge des contribuables de la classe 1 est déterminé par application du tarif de l'article 118 au revenu imposable.

Ecrire un programme qui affiche l'impôt dû pour un contribuable de la classe 1 en fonction du revenu imposable entré au clavier.

Exemple : pour un revenu imposable de 417 000 francs, il faut payer un impôt de 0 + 10 860 + 13 560 + 660 = 25 080 francs.

10 860 provient des 10% de la tranche : 108 600 = 346 200 – 237 600

13 560 provient des 20% de la tranche : 67 800 = 414 000 – 346 200

660 provient des 22% de la tranche : 3000 = 417 000 – 414 000

La réalisation du programme en *Leria* nous donne l'occasion de traiter un tableau d'enregistrements. Le tableau que nous créerons sera un reflet fidèle du texte de loi. Chaque entrée dans la table comporte le pourcentage *taux* à appliquer au montant du revenu imposable compris entre les valeurs *min* et *max*. L'impôt dû pour chaque tranche est accumulé dans la variable *impot*. Il faut veiller à ne pas oublier d'ajouter à ce total le montant dû pour la dernière tranche.

```
PROGRAM impot_classe_1;
TYPE t_taux_imposition = RECORD
       NUMERIC : taux, min, max;
     END;

TYPE t_table = ARRAY [18] OF t_taux_imposition;

VAR numeric : impot,revenu_imposable;
VAR int : i;
VAR t_table : t;

  VOID FUNCTION init_table_imposition(t_table : VAR t);
  BEGIN
    t[1].taux  :=  0.;  t[1].min  :=       0.; t[1].max  :=  237600.;
    t[2].taux  := 10.;  t[2].min  :=  237600.; t[2].max  :=  346200.;
    t[3].taux  := 20.;  t[3].min  :=  346200.; t[3].max  :=  414000.;
    t[4].taux  := 22.;  t[4].min  :=  414000.; t[4].max  :=  481200.;
    t[5].taux  := 24.;  t[5].min  :=  481200.; t[5].max  :=  548400.;
    t[6].taux  := 26.;  t[6].min  :=  548400.; t[6].max  :=  616200.;
    t[7].taux  := 28.;  t[7].min  :=  616200.; t[7].max  :=  683400.;
    t[8].taux  := 30.;  t[8].min  :=  683400.; t[8].max  :=  750000.;
    t[9].taux  := 32.;  t[9].min  :=  750000.; t[9].max  :=  818400.;
    t[10].taux := 34.;  t[10].min :=  818400.; t[10].max :=  885600.;
    t[11].taux := 36.;  t[11].min :=  885600.; t[11].max :=  952800.;
    t[12].taux := 38.;  t[12].min :=  952800.; t[12].max := 1020000.;
    t[13].taux := 40.;  t[13].min := 1020000.; t[13].max := 1087200.;
    t[14].taux := 42.;  t[14].min := 1087200.; t[14].max := 1155000.;
    t[15].taux := 44.;  t[15].min := 1155000.; t[15].max := 1221600.;
    t[16].taux := 46.;  t[16].min := 1221600.; t[16].max := 1288800.;
    t[17].taux := 48.;  t[17].min := 1288800.; t[17].max := 1357200.;
    t[18].taux := 50.;  t[18].min := 1357200.; t[18].max := 999999999.;
  END;
BEGIN
  init_table_imposition(t);
```

```
  { saisie du revenu imposable }
  WRITE("Entrez le revenu imposable : "); READ(revenu_imposable);

  { calcul de l'impôt }
  impot := 0.;
  i := 1;
  WHILE revenu_imposable > t[i].max DO BEGIN
    impot := impot + ((t[i].taux / 100.) * (t[i].max - t[i].min));
    i := i+1;
  END;
  impot := impot + ((t[i].taux / 100.) * (revenu_imposable - t[i].min));
  { affichage du résultat }
  WRITE("Impôt dû : ",impot:4:0,\n);
END.
```

Voici deux traces d'exécution du programme précédent :

```
Entrez le revenu imposable : 1500000
Impôt dû :    426024

Entrez le revenu imposable : 3000000
Impôt dû :   1176024
```

3 *Le jeu MasterMind*

Il s'agit d'écrire un programme de jeu dans lequel l'utilisateur joue contre l'ordinateur. Lorsque le jeu démarre, le programme détermine une combinaison de nombres (couleurs) au hasard. Un nombre n'apparaît qu'une seule fois dans la combinaison, et l'utilisateur doit deviner la combinaison générée dans l'ordre exact. Lorsque l'utilisateur entre une combinaison, l'ordinateur détermine combien de nombres sont corrects et se trouvent à la bonne position. C'est le nombre de positions noires. Le nombre de couleurs correctes mais utilisées à une place incorrecte dans la combinaison détermine le nombre de positions blanches. Après chaque tentative de l'utilisateur, l'ordinateur le renseigne sur le nombre de positions blanches et noires obtenues. Le jeu s'arrête lorsque le nombre de positions noires correspond à la taille de la combinaison à trouver.

Pour illustrer ce jeu, voici une partie jouée contre l'ordinateur en utilisant le programme *Leria* ci-dessous.

```
Vous jouez contre moi. Je génère une combinaison au hasard...
Chaque couleur est unique !
Vous avez le choix entre les couleurs de 1 à 6
Entrez 4 nombres compris entre 1 et 6
1
2
3
4
Vous avez entré la combinaison 1  2  3  4
Vous avez 0 noir(s) et 3 blanc(s)
Entrez 4 nombres compris entre 1 et 6
2
3
4
5
Vous avez entré la combinaison 2  3  4  5
Vous avez 1 noir(s) et 1 blanc(s)
Entrez 4 nombres compris entre 1 et 6
```

```
1
3
4
6
Vous avez entré la combinaison 1  3  4  6
Vous avez 2 noir(s) et 2 blanc(s)
Entrez 4 nombres compris entre 1 et 6
3
1
4
6
Vous avez entré la combinaison 3  1  4  6
Vous avez 4 noir(s) et 0 blanc(s)
Bravo, félicitations, vous avez gagné en 4 tentative(s)
```

La structure du programme principal est assez simple :

générer une combinaison au hasard de nombres uniques. La taille de la combinaison est fixée par la constante *max* et le nombre de nombres (couleurs) possibles l'est par la constante *nbre_couleurs*.

nbre_essais ← 0
répéter
    saisie d'une combinaison de l'utilisateur et affichage de cette combinaison entrée
    nbre_essais ← *nbre_essais* + 1
    vérification de la combinaison entrée
    affichage du résultat de la vérification
jusqu'à ce que la combinaison soit découverte
afficher un message de félicitations ainsi que *nbre_essais*

Chaque couleur est représentée par un nombre entier positif unique. Le nombre maximal de couleurs autorisées est donné par la constante *nbre_couleurs*. La combinaison générée aléatoirement par l'ordinateur est conservée dans le tableau *combinaison_a_trouver.*

Occupons-nous d'abord de savoir comment l'ordinateur génère la combinaison de couleurs au hasard.

Pour générer des nombres aléatoires, nous allons nous servir des fonctions prédéfinies *rand*() et *randomize*(). Rappelons le fonctionnement de ces fonctions : l'argument spécifié est utilisé pour démarrer la séquence de nombres pseudo-aléatoires. Cela permet d'empêcher le programme d'utiliser, à chaque nouvelle exécution du programme, la même séquence de nombres aléatoires et donc d'avoir véritablement une nouvelle combinaison à chaque nouveau jeu. C'est ainsi que nous démarrons le générateur de nombres pseudo-aléatoires avec une « graine » différente à chaque exécution, à savoir l'heure actuelle. Cet argument ne doit être spécifié qu'une seule fois par jeu. Par la suite, la fonction *rand*() sera appelée sans argument, ce qui fait que nous obtenons à chaque appel un nombre compris entre 0 et 1. Donc, pour obtenir une couleur au hasard comprise entre 1 et *nbre_couleurs*, le nombre aléatoire compris entre 0 et 1 obtenu est multiplié par *nbre_couleurs*-1,

puis arrondi. Enfin, cette valeur est augmentée de 1. Cela nous donne un nombre entier compris entre 1 et *nbre_couleurs*.

Il subsiste une dernière difficulté : comment procéder pour que chaque couleur ne soit présente qu'une seule fois dans une combinaison ? Ce problème se résout facilement si l'on s'aide d'un second tableau disposant d'autant d'éléments que de couleurs. Au départ, tous ses éléments auront une valeur indiquant qu'aucune des couleurs possibles n'est encore utilisée dans la combinaison. Ensuite, chaque fois qu'une couleur aléatoire est générée, on vérifie si cette couleur a déjà été sélectionnée. Si oui, alors la couleur est écartée, et une nouvelle couleur aléatoire est générée. Si non, alors la couleur est retenue dans la combinaison et, dans ce tableau d'aide, nous mémorisons le fait que cette couleur vient d'être sélectionnée.

Attaquons-nous à présent au sous-problème suivant : la saisie d'une combinaison entrée par le joueur. Le programmeur doit veiller à trois points :

- la combinaison proposée par le joueur doit comporter *max* couleurs ;
- la valeur de chaque couleur (nombre) entrée doit être comprise entre 1 et *nbre_couleurs* ;
- toute couleur proposée doit être unique dans la combinaison entrée.

La première contrainte sera mise en œuvre à l'aide d'une boucle parcourue dans sa totalité, exactement *max* fois : à chaque tour de boucle accompli complètement, une couleur supplémentaire est ajoutée à la combinaison proposée. Pour qu'une couleur soit acceptée, il faut donc qu'elle soit dans le domaine des couleurs permises et qu'elle n'ait pas encore été choisie préalablement dans la même combinaison. Comme pour la génération d'une combinaison au hasard, nous nous aidons d'un tableau supplémentaire mémorisant les couleurs sélectionnées lors de la saisie de la combinaison en cours.

Ne subsiste que le problème de la vérification de la combinaison par le joueur. Le programme doit fournir au joueur le nombre de positions « noires » et « blanches ». C'est ainsi que le sous-programme *verification*() détermine d'abord les positions blanches en comparant systématiquement tous les nombres de la combinaison proposée aux couleurs de la solution. Le *i*ème nombre de la combinaison *essai*(*i*) à devoir être vérifié est comparé à tous les *comb*(*j*) de la solution, pour *j* allant de 0 à *max*-1. Comme tous les *nbre*(*i*) doivent être vérifiés, cette boucle s'applique à tous les *i* compris entre 1 et *max*-1. A chaque égalité, le nombre de positions blanches est augmenté de 1.

Le nombre de positions noires est déterminé ensuite. Ici, la vérification est plus simple du fait qu'il suffit de comparer chaque *essai*(*i*) avec son *comb*(*i*) correspondant. Une boucle simple est suffisante.

De la façon dont les positions blanches ont été obtenues précédemment, elles contiennent évidemment les positions noires. Celles-ci doivent, par conséquent, être décomptées du total des positions blanches calculées. Lorsque le nombre de positions noires est égal à la taille de la combinaison, le joueur a deviné la solution correcte à trouver, et ce fait est signalé au programme appelant.

Le lecteur trouvera ci-dessous une implantation possible en *Leria*.

```
{$d}
PROGRAM mastermind;

CONST              max = 4;
CONST    nbre_couleurs = 6;
CONST             vrai = 1;
CONST             faux = 0;

TYPE t_combinaison = ARRAY[max] OF int;
TYPE t_resultats = RECORD
       int : noirs;    { nombre de noirs }
       int : blancs;   { nombre de blancs }
     END;

VAR t_combinaison : combinaison_a_trouver,essai;
VAR       int : trouve_combinaison, nbre_essais;
VAR   t_resultats : resultats;

void FUNCTION cree_combinaison_a_trouver(t_combinaison : VAR c);
 TYPE t_selectionne = ARRAY[nbre_couleurs] OF int;
 VAR  t_selectionne : selectionne;
 VAR  int : i,couleur_hasard;
BEGIN
  WRITE("Vous jouez contre moi. Je génère une combinaison au hasard...",\n);
  WRITE("Chaque couleur est unique !",\n);
  WRITE("Vous avez le choix entre les couleurs de 1 à ",nbre_couleurs:1:0,\n);
  FOR i := 1 UNTIL i > nbre_couleurs BY 1 DO selectionne[i] := faux;
  randomize(time());
  i := 1;
  WHILE i <= max DO BEGIN
    couleur_hasard := 1+rand()#nbre_couleurs;
    IF NOT selectionne[couleur_hasard] THEN BEGIN
       selectionne[couleur_hasard] := vrai;
       c[i] := couleur_hasard;
       i := i+1;
    END;
  END;
END;

void FUNCTION affiche(t_combinaison : c);
VAR int : i;
BEGIN
  FOR i := 1 UNTIL i > max BY 1 DO WRITE(c[i]:3:0);
  WRITE(\n);
END;

void FUNCTION saisie_combinaison(t_combinaison : VAR c);
 TYPE t_selectionne = ARRAY[nbre_couleurs] OF int;
 VAR  t_selectionne : selectionne;
 VAR        int : i, nombre_entre_ok,nbre_entre;
BEGIN
  FOR i := 1 UNTIL i > nbre_couleurs BY 1 DO selectionne[i] := faux;
  WRITE("Entrez ",max, " nombres compris entre 1 et ",nbre_couleurs,\n);
  FOR i := 1 UNTIL i > max BY 1 DO BEGIN
    nombre_entre_ok := faux;
    WHILE NOT nombre_entre_ok DO BEGIN
      READ(nbre_entre);
      IF (nbre_entre >= 1) AND (nbre_entre <= nbre_couleurs)
         THEN IF NOT selectionne[nbre_entre]
                 THEN BEGIN
                   selectionne[nbre_entre] := vrai;
                   nombre_entre_ok := vrai;
```

```
                  c[i] := nbre_entre;
                END
                ELSE WRITE("Erreur : nombre déjà entré",\n)
         ELSE WRITE("Erreur : nombre doit être entre 1 et ",nbre_couleurs,\n);
    END;
  END;
END;

int FUNCTION verifier(t_combinaison : comb; t_combinaison : essai;
                          t_resultats : VAR r);
VAR int : i,j;
BEGIN
  { déterminer le nombre de positions blanches }
  r.blancs := 0;
  FOR i := 1 UNTIL i > max BY 1 DO
    FOR j := 1 UNTIL j > max BY 1 DO
      IF essai[i] = comb[j] THEN r.blancs := r.blancs + 1;
  { déterminer le nombre de positions noires }
  r.noirs := 0;
  FOR i := 1 UNTIL i > max BY 1 DO
    IF essai[i] = comb[i] THEN r.noirs := r.noirs + 1;
  { ajuster le nombre de blancs }
  r.blancs := r.blancs - r.noirs;
  IF r.noirs = max
     THEN RETURN vrai
     ELSE RETURN faux;
END;
BEGIN { programme principal }
  cree_combinaison_a_trouver(combinaison_a_trouver);
  affiche(combinaison_a_trouver);
  nbre_essais := 0;
  REPEAT
    saisie_combinaison(essai);
    nbre_essais := nbre_essais + 1;
    WRITE("Vous avez entrŽ la combinaison ");
    affiche(essai);
    trouve_combinaison := verifier(combinaison_a_trouver,essai, resultats);
    WRITE("Vous avez ",resultats.noirs:1:0, " noir(s) et ",
          resultats.blancs, " blanc(s)",\n);
  UNTIL trouve_combinaison;
  WRITE("Bravo, félicitations, vous avez gagné en ",nbre_essais,
        " tentative(s)",\n);
END.
```

Définition et utilisation du langage Leria

Le langage de programmation *Leria* est un langage de programmation structuré, destiné à être utilisé à des fins pédagogiques. Il est librement inspiré des langages de programmation modernes tels que Pascal, C… Il n'a pas la prétention d'offrir des concepts nouveaux, ni d'améliorer les concepts connus. Il peut surprendre par certains choix et par conséquent être la source de discussions et d'« améliorations » intéressantes. Le langage permet entre autres :

- la définition et l'utilisation de constantes ;
- la définition et l'utilisation de vecteurs et tableaux multidimensionnels ;
- la définition et l'utilisation d'enregistrements (structures) ;
- l'utilisation de fonctions et procédures éventuellement récursives avec ou sans paramètres ;
- les modes de passage par valeur et par variable (référence) ;
- le langage est impératif à structures de blocs, admettant notamment les types prédéfinis entier, caractère et numérique (réel).

1 Les symboles du langage Leria

Un programme *Leria* se compose d'une séquence de symboles terminaux respectant la syntaxe du langage *Leria*.

Un symbole terminal est l'unité de texte la plus petite ayant un sens pour *Leria*. On distingue notamment les terminaux suivants: identificateurs, symboles spéciaux, littéraux numériques, caractère et alphanumériques. Deux terminaux (identificateurs ou littéraux) sont séparés par un séparateur qui est, soit un blanc, soit un commentaire, soit un retour à la ligne.

Pour créer des identificateurs, *Leria* permet l'utilisation des lettres de A à Z (il n'y a pas de distinction entre majuscules et minuscules), les chiffres de 0 à 9 ainsi que le caractère « blanc souligné » (« _ »). Les identificateurs sont classés en trois catégories : les mots réservés, les identificateurs prédéfinis et les identificateurs créés par le programmeur.

1.1 Les mots réservés et les symboles spéciaux

Voici la liste complète des 29 mots réservés :

```
AND  ARRAY  BEGIN  BY  CONST  DO  ELSE  END  FOR  FORWARD
FUNCTION  HALT  IF  NOT  OF  OR  PROGRAM  READ  RECORD  REPEAT
RETURN  SKIP THEN TO  TYPE  UNTIL  VAR  WHILE  WRITE
```

Ces mots-clés ne peuvent jamais être utilisés par le programmeur comme identificateurs.

Les symboles spéciaux utilisés dans *Leria* sont les suivants :

```
[  ]  {  }  (  )  <  >  <=  >=  <>  =  :=  :  ,  .  ;  '  \n  +  –  /  *  %^  _  #
```

Ils ne peuvent être employés que dans des contextes précis où ils ont une signification bien définie.

1.2 Les identificateurs

La structure d'un identificateur peut être décrite par des règles formelles :

```
identificateur ::= lettre { caractère }
caractère ::= lettre | chiffre | blanc_souligné
```

Ce sont deux règles de production écrites dans un formalisme appelé EBNF (Extended Backus Naur Form). La première règle se lit ainsi : un identificateur est une lettre suivie de 0 à *n* caractères. Dans cette notation, le fait qu'un symbole ou une suite de symboles puisse être répété 0 à *n* fois est indiqué par le placement du symbole ou de la suite de symboles en question entre accolades « { } ». Les accolades sont des méta-symboles au même titre que le symbole « ::= ». Un méta-symbole est un symbole utilisé pour décrire un langage, mais ne faisant pas partie du langage.

La deuxième règle signifie qu'un caractère est soit une lettre, soit un chiffre, soit un blanc souligné. Le choix est indiqué par le méta-symbole « | ». Il existe une autre

paire de méta-symboles, les crochets « [] ». Placer des symboles entre crochets signifie que ces symboles sont optionnels, c'est-à-dire qu'ils peuvent être présents une seule fois ou alors être carrément absents.

Les mots lettre, chiffre et blanc_souligné restent encore à être définis. On dit que ce sont des symboles non-terminaux, à opposer aux symboles terminaux. Le mot qui figure à gauche du méta-symbole « ::= » est toujours un non-terminal. Afin de distinguer ces deux types de symboles dans ce formalisme, nous placerons les symboles terminaux entre guillemets « "" ». Pour tout symbole non-terminal, il doit y avoir une règle qui nous permet d'aboutir en fin de compte à des symboles terminaux. Dans notre cas, leurs définitions sont les suivantes :

lettre ::= "a" | "b" | "c" |...| "y" | "z" | "A" | "B" | "C" |...| "Y" | "Z"
chiffre ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
blanc_souligné ::= "_"

Dans *Leria*, la longueur maximale d'un identificateur est fixée à cent caractères. Le programmeur peut néanmoins utiliser des identificateurs plus longs, mais uniquement les cent premiers caractères seront pris en compte par *Leria*. Voici quatre exemples d'identificateurs corrects :

a_faire a123 a____1 abcdefghijabcdefghij

Les trois identificateurs suivants sont incorrects, car ils ne respectent pas la structure syntaxique définie pour un identificateur :

_Nom 2ased 2345

Leria comprend toute une liste d'identificateurs prédéfinis, c'est-à-dire d'indentificateurs qui ont un sens prédéfini, mais que le programmeur peut modifier :

char int numeric void

abs acos asin atan chr cos exp impair intnum ln numint ord

power randomize rand round sin sqr sqrt tan time trunc

Voici un petit exemple illustrant comment le programmeur peut redéfinir des identificateurs prédéfinis :

```
PROGRAM test;
TYPE cos = ARRAY[5] OF int;
TYPE numeric = RECORD
       numeric : x;
       int : y;
     END;
VAR cos : t;
VAR numeric : r,erreur;
BEGIN
  t[1] := 5;
  write(t[1],\n);
{  write(cos(3.1415),\n); { erreur }}
  r.x := 2.345;
  r.y := 5;
  write(r.x,' ',r.y,\n);
{  erreur := 2.718;  { erreur }}
END.
```

1.3 Les littéraux numériques et entiers

Leria reconnaît les littéraux numériques et entiers qui respectent les formats généraux suivant :

littéral_numérique ::= chiffre { chiffre } ["." { chiffre }]

littéral_entier ::= chiffre { chiffre }

La valeur minimale et maximale d'un nombre est dépendante de la machine sur laquelle *Leria* est implanté[1]. 0.123, 123.4 et 001.00 sont des littéraux numériques corrects, alors que .234 est un exemple d'un littéral numérique incorrect. De même, 987 et 345764 sont des littéraux entiers corrects.

1.4 Les littéraux alphanumériques

Une chaîne de caractères est une séquence de 0 à 255 caractères du jeu des caractères imprimables, définis par le réalisateur du compilateur, placée entre guillemets. Dans notre cas, le jeu de caractères imprimables est un sous-ensemble du jeu de caractères ASCII étendu (tous les caractères sauf ceux ayant un code compris entre 0 et 31 compris, de même que le caractère de code 127). Si la chaîne de caractères doit contenir un guillemet, alors celui-ci doit être doublé, c'est-à-dire suivi immédiatement par un autre guillemet afin qu'il n'y ait pas de confusion avec le guillemet de fin de chaîne. Voici trois exemples de chaînes de caractères correctes :

"Coucou" "L'exemple avec blancs permis !" """"

Le dernier exemple (quatre guillemets) représente une chaîne représentant un seul caractère, le caractère guillemet.

1.5 Le littéral caractère

Un littéral caractère est un caractère du jeu de caractères de la machine, placé entre « '' ». Pour indiquer un caractère non représentable directement, il suffit d'indiquer en décimal son code ASCII précédé par le symbole « \ ». Exemples :

'A' '\65' '\\' '\7'

Les deux premiers exemples représentent le même littéral, à savoir le caractère « A », puisque 65 est son code ASCII. Le troisième exemple représente le caractère « \ », tandis que le dernier représente le caractère dont le code ASCII vaut 7 (un signal sonore), caractère pour lequel il n'y a pas de touche sur le clavier.

1. Dans notre implantation, la précision d'un littéral numérique coïncide avec celles du type *double* en 64 bits du langage C++ (c'est-à-dire une valeur comprise entre $2{,}2 \cdot 10^{-308}$ et $1.8 \cdot 10^{308}$). Un entier peut prendre une valeur allant de –2 147 483 648 à +2 147 483 647).

1.6 Les commentaires

Un programme écrit en *Leria*, comme d'ailleurs dans n'importe quel langage de programmation, devrait être commenté. Un commentaire débute par le symbole accolade ouverte « { » et se termine par le symbole accolade fermée « } ». Tous les caractères se trouvant entre ces deux symboles sont ignorés par le compilateur et servent uniquement au programmeur à documenter son programme. Les commentaires peuvent être imbriqués. Dans ce cas, il faut veiller à placer un nombre correct d'accolades ouvertes et fermées. Un commentaire peut s'étendre sur plusieurs lignes. Exemples :

```
{ Ceci est un commentaire }

{ Ceci est { un commentaire imbriqué } }
```

Un commentaire qui contient immédiatement après le caractère « { » un caractère « $ » est une *directive de compilation*.

1.7 Les directives de compilation

Une directive de compilation est un commentaire à part qui provoque certaines actions spéciales du compilateur. Ces actions spéciales peuvent être les suivantes :

- activer ou désactiver le contrôle de validité des indices d'un tableau au moment de l'exécution. La syntaxe est {$R+} ou {$R–}. Par défaut, le compilateur effectue la génération de code avec le contrôle de validité des indices actifs {$R+} pour tout le programme. Lorsque la mise au point du programme est terminée, le programmeur peut désactiver le contrôle de validité {$R–} pour accélérer l'exécution et diminuer la taille du programme. Cette directive peut aussi être utilisée localement ;
- provoquer l'affichage du code généré, désassemblé, après la compilation du programme. Cette directive {$D} peut se trouver n'importe où dans la source du programme.

2 *Structure d'un programme et notion de bloc*

Tout programme *Leria* a la structure suivante :

```
programme_leria ::= "PROGRAM" identificateur ";" bloc "."
```

où identificateur doit respecter les règles vues plus haut et désigne le nom interne du programme.

Leria supporte la notion de bloc. Un bloc se compose de définitions suivies d'instructions. Un bloc fait toujours partie, soit d'une fonction, soit du programme principal. Il a la structure suivante :

```
bloc ::=    partie_définition_constantes
            partie_définition_types
            partie_définition_variables
            partie_définition_de_fonctions
            instruction_composée
```

Cette règle se lit ainsi : le bloc comporte une partie de définitions de constantes suivie d'une partie de définitions de types suivie d'une partie de définitions de variables suivie d'une partie de définitions de fonctions et se termine obligatoirement par une instruction composée. En fait, comme nous le verrons plus loin, les quatre premières parties peuvent être vides.

Un programme *Leria* contient toujours au moins le bloc du programme principal. Un identificateur défini dans un bloc est dit local à ce bloc. Tous les identificateurs définis dans un même bloc doivent être uniques et complètement définis avant leur utilisation. L'exception à cette règle est le nom d'une fonction qui est connu depuis le début de sa définition jusqu'à la fin du bloc qui la contient, permettant ainsi des définitions récursives. De même, une fonction peut être utilisée avant d'avoir été définie si elle a fait l'objet d'une déclaration en avant (cf. 6).

Les considérations précédentes impliquent que deux objets peuvent porter le même nom à condition qu'ils soient définis dans deux blocs différents. Si dans le bloc principal on définit un objet d'identificateur *A* et que dans un bloc fonction on définit un objet avec le même identificateur *A*, alors à l'intérieur du bloc fonction, l'objet d'identificateur *A* du bloc principal n'est plus accessible. Au niveau du bloc principal, aucun objet défini dans un bloc fonction n'est accessible.

3 Les constantes, types et variables

Le programmeur peut définir des constantes dans la partie de définitions de constantes. Cette section de définitions doit être la première du bloc. Sa structure est la suivante :

```
partie_définition_constantes ::= { définition_constante }
définition_constante ::= "CONST" identificateur "="     "[" – "]" const_numérique |
                                                        "[" – "]" const_entier | const_caractère ";"
const_numérique ::= littéral_numérique | identificateur
const_entier ::= littéral_entier | identificateur
const_caractère ::= littéral_caractère | identificateur
```

Une constante définit un nom symbolique qui sert à identifier une valeur numérique ou une valeur de type caractère. Une définition de constante ne peut être récursive. Le nom d'une constante peut être utilisé par la suite, dans le domaine de définition de l'identificateur de constante, en lieu et place de la valeur qui ne peut plus être modifiée par la suite dans le programme. Exemples :

```
CONST nbre_colonnes = 80;
CONST nbre_lignes = 25;
CONST dollar = '$';
CONST pi = 3.1415;
CONST bidon = -nbre_colonnes;
```

Dans un programme *Leria*, tout objet est typé. Le type d'un objet définit les valeurs possibles que peut prendre cet objet ainsi que les opérations autorisées sur cet objet. Il existe quatre types prédéfinis, à savoir le type numeric (numérique), le type int (entier) et le type char (caractère) ainsi que le type void (néant) qui ne peut être utilisé que pour indiquer qu'une fonction ne retourne pas de valeur du tout. À côté de ces types prédéfinis, le programmeur peut définir de nouveaux types "tableau" et "enregistrement" dans la partie de définition des types :

partie_définition_types ::= { définition_type }
définition_type ::= "TYPE" identificateur "=" définition_tableau ";" |
"TYPE" identificateur "=" définition_record ";"
définition_tableau ::= "ARRAY" "[" const_entière "]" "OF" identificateur
définition_record ::= "RECORD" définition_champs ";" { définition_champs ";" } "END"
définition_champs ::= identificateur ":" identificateur { "," identificateur }

L'identificateur intervenant dans définition_tableau doit être un identificateur de type préalablement défini à l'exception du type void. La taille d'un tableau est une valeur entière positive. Une définition de type ne peut être récursive. Exemples :

```
CONST max = 20;
TYPE tableau1 = ARRAY[200] OF numeric;
TYPE tableau2 = ARRAY[max] OF tableau1;
TYPE enreg =   RECORD
                  numeric : x,y;
                  tableau1 : z;
               END;
```

La taille des tableaux doit être fixée à la compilation. L'indice du premier élément d'un tableau est toujours un. Dans l'exemple précédent, un tableau de type **`tableau1`** se compose de 200 éléments numérotés de 1 à 200.

Dans définition_champs, l'identificateur précédant le symbole « : » doit être un identificateur de type autre que void. Un identificateur de champ peut être n'importe quel identificateur respectant les règles du langage.

À l'intérieur d'un bloc, les variables doivent être définies dans la section de définitions de variables qui a la structure suivante :

partie_définition_variables ::= { définition_variable }
définition_variable ::= "VAR" identificateur ":" identificateur { "," identificateur } ";"

L'identificateur compris entre le mot réservé VAR et le symbole spécial « : » doit être un identificateur de type défini préalablement à l'exception du type void. La liste des identificateurs comprise entre les symboles spéciaux « : » et « ; » où deux identificateurs consécutifs sont séparés par une virgule, est la liste des variables qui seront toutes du type indiqué par l'identificateur de type. Voici un exemple en reprenant les définitions des types ci-dessus :

```
VAR tableau1 : x,y;          { 2 tableaux à une dimension }
VAR numeric : i;             { une variable numérique simple }
VAR tableau2 : ta,tb,tc,td; { 4 tableaux à deux dimensions }
VAR enreg : r;               { une variable de type enregistrement }
```

L'accès à une variable simple, à une variable de type tableau ou à un enregistrement, c'est-à-dire à un élément d'un tableau, est régi par la règle syntaxique suivante :

accès_variable ::= identificateur { "[" expression "]" | "." identificateur }

Si l'identificateur est suivi de crochets, l'identificateur en question doit évidemment être du type tableau. S'il est suivi par un point, il doit désigner un objet de type record. Exemples : pour accéder à l'élément numéroté 2 du tableau **`x`** défini ci-dessus, on écrit **`x[2]`**. Pour les tableaux à deux dimensions comme le tableau **`ta`**, on peut écrire **`ta[20][120]`** pour accéder à un nombre précis ou **`ta[5]`** pour accéder à un élément de type **`tableau1`**. Dans le cas d'un accès à un élément d'un tableau multidimensionnel, l'ordre d'évaluation des indices n'est pas défini.

Pour accéder aux champs **x** et **y** de l'enregistrement **r**, on écrit : **r.x** et **r.y**. Le champ **z** est accédé de la même façon, sauf qu'il s'agit d'un vecteur, et que par conséquent, ses éléments sont accédés d'après la notation expliquée ci-dessus. Exemple : **r.z[2]**.

4 Les expressions

Dans *Leria*, une expression a la structure suivante :

expression ::=terme | "–" expression | "NOT" expression | expression opérateur_binaire expression
terme ::= accès_variable | "(" expression ")" | appel_fonction | const_numérique | const_caractère
opérateur_binaire ::="+" | "–" | "*" | "/" | "%" | "#" | "^" | ">" | "<" | ">=" | "<=" | "<>" | "=" | "OR" | "AND"

Le type des opérandes dans une expression où intervient un opérateur peut être char, int ou numeric selon la nature de l'opérateur. Les opérandes d'un opérateur sont évalués de la gauche vers la droite. Tous les opérateurs ont la même priorité ! Dans une expression impliquant plusieurs opérateurs, l'évaluation de ceux-ci se fait de la droite vers la gauche à moins que le programmeur ne force la priorité d'évaluation en utilisant des parenthèses. Exemples :

2–3*5+1–5*7 est évalué comme 2–(3*(5+(1–(5*7)))) ce qui donne 89
2–(3*5)+1 est évalué comme 2–((3*5)+1) = –14
–2–3 est évalué comme –(2–3) = 1

L'évaluation d'un opérateur relationnel (<, <=, >, >=, <>, =) donne comme résultat l'entier 1 (vrai) si la relation est vérifiée et l'entier 0 dans le cas contraire. Il n'existe pas de type booléen dans *Leria*. Il est néanmoins possible d'utiliser les opérateurs logiques AND, OR et NOT sur des opérandes de type int. L'opérateur AND est évalué à 1 (vrai) si ses deux opérandes ont comme valeur 1 (vrai), et à 0 (faux) dans les cas contraires. L'opérateur OR est évalué à 1 (vrai) si un des ses deux opérandes au moins a comme valeur 1 (vrai), et à 0 dans le cas contraire. Ces valeurs 0 et 1 sont toujours de type int.

Exemples :

5 < 3 est évalué à 0 (faux)
5 > 3 est évalué à 1 (vrai)
(5 > 3) AND (4 > 3) est évalué à 1 (vrai)
(3 > 2) OR (1 > 2) est évalué à 1 (vrai)
('a' <= 'b') AND ('b' < 'c') est évalué à 1 (vrai)
(3.1 > 2.6) AND (1.4 > 2.8) est évalué à 0 (faux)

L'opérateur NOT appliqué à une expression de type int donne 1 si la valeur de l'expression vaut 0 et 1 dans tous les autres cas. Exemples :

NOT 3 est évalué à 0 (faux)
NOT 0 est évalué à 1 (vrai)

Ainsi, dans l'évaluation d'une expression, toute valeur entière différente de 0 est considérée comme vraie logiquement, la valeur entière 0 est considérée comme faux.

L'opérateur binaire % effectue une division entière contrairement à l'opérateur binaire / qui effectue une division réelle habituelle. Exemples :

3 / 2 est évalué à 1
3 % 2 est évalué à 1
–1 / 2 est évalué à 0
–1 % 2 est évalué à 0

3. / 2. est évalué à 1.5
3. % 2. est évalué à 1
–1. / 2. est évalué à –0.5
–1. % 2. est évalué à 0

Le symbole # permet d'effectuer une opération « modulo », c'est-à-dire d'obtenir le reste d'une division entière. Exemples :

7 # 3 est évalué à 1
5.2 # 3. est évalué à 2.2
–1 # 3 est évalué à –1

L'opérateur ^ permet d'effectuer l'exponentiation avec comme base l'opérande de gauche et comme exposant l'opérande de droite. Exemples :

2 ^ 3 est évalué à 8
2. ^ 0.5 est évalué à 1.414214

Les expressions 0^0 et x^y avec x négatif et y non entier ne sont pas définies.

5 L'instruction composée

Une instruction composée est une séquence d'instructions placée entre deux mots réservés indiquant le début BEGIN et la fin END de l'instruction composée. Voici sa syntaxe :

instruction_composée ::= "BEGIN" instruction ";" { instruction ";" } "END"

Chaque instruction se termine obligatoirement par un point-virgule, et une instruction composée contient au moins une instruction.

Dans *Leria*, une instruction est définie ainsi :

instruction ::= affectation | if_instruction | while_instruction | read_instruction | write_instruction | return_instruction | instruction_vide | halt_instruction | repeat_instruction | appel_fonction | instruction_composée | for_instruction

5.1 L'instruction d'affectation

affectation ::= accès_variable ":=" expression

Lors de l'exécution d'une instruction d'affectation, l'expression à droite du symbole d'affectation « := » est évaluée. Ensuite la valeur résultant de cette évaluation est affectée à la variable identifiée par l'identificateur de variable à gauche du symbole affectation, c'est-à-dire que la valeur qu'avait cette variable de gauche avant l'affectation est perdue et est remplacée par la valeur résultant de l'évaluation de l'expression. Exemples :

```
PROGRAM test;
VAR int i,j;
```

```
BEGIN
  i := 4;                    { avant : i indéfini  après : i = 4 }
  j := i+5;                  { avant : j indéfini  après : j = 9 }
END;
```

La valeur de l'expression doit être du même type que la variable à gauche du symbole d'affectation. Deux opérandes sont de types compatibles ou de même type s'ils ont été déclarés à l'aide du même identificateur de même portée. Exemple :

```
PROGRAM test;
TYPE t1 = ARRAY[2] OF numeric;
TYPE t2 = ARRAY[2] OF numeric;
VAR t1 : i;
VAR t2 : j;
BEGIN
  i := i;       { ok }
  i[2] := j[1]; { ok }
  i := j;       { erreur : types non compatibles }
END.
```

5.2 Les instructions d'entrée/sortie

Il existe deux instructions d'entrée/sortie dans *Leria* :

read_instruction ::= "READ" "(" accès_variable { "," accès_variable } ")"
write_instruction ::= "WRITE" "(" w_expr { "," w_expr } ")"
w_expr ::= expression [":" expression [":" expression]] | "\n" | littéral_alphanumérique

L'instruction READ permet de changer la valeur d'une variable en lui affectant une valeur entrée par l'utilisateur au clavier de l'ordinateur.

Si plusieurs variables figurent dans la liste de l'instruction READ, le programme demandera à l'utilisateur d'entrer des valeurs au clavier. À la fin de la saisie, l'utilisateur doit entrer un « Enter » ou « Return ».

L'instruction WRITE provoque l'affichage à l'écran des valeurs des expressions figurant dans la liste entre parenthèses. L'affichage du symbole spécial « \n » ou « \N » provoque le déplacement du curseur au début de la ligne suivante à l'écran. Exemple :

```
PROGRAM somme;
VAR int : i,j;
BEGIN
  WRITE(\n,"Entrez deux nombres : ");
  READ(i,j);
  WRITE(\n,"La somme de ",i," et ",j, " vaut : ",i+j);
END.
```

Une exécution de ce programme donne par exemple :

```
Entrez deux nombres : 2 3

La somme de 2 et 3 vaut 5
```

Par défaut, l'affichage de valeurs de type numeric se fait sur 8 positions dont deux après le point décimal. Le programmeur peut influencer la présentation des résultats numériques en utilisant les expressions de formatage. La première expression de formatage représente le nombre minimal de caractères que le nombre doit occuper, la deuxième expression de formatage représente le nombre de chiffres

après le point décimal. Ces expressions sont tronquées et changées de signe pour obtenir des valeurs entières positives ou nulles. Exemples :

t	f	WRITE(123.456:t:f)	WRITE(–123.456:t:f)
1	1	123.5	–123.5
5	1	123.5	–123.5
6	1	^123.5	–123.5
7	1	^^123.5	^–123.5
8	5	123.45600	–123.45600

Dans ce tableau, les caractères ^ représentent des espaces.

5.3 L'instruction conditionnelle

Dans l'instruction conditionnelle, si l'évaluation de l'expression donne un résultat différent de 0, alors c'est l'instruction suivant le mot réservé THEN qui est exécutée. La partie ELSE, même si elle est présente, ne sera pas exécutée.

```
if_instruction ::= "IF" expression "THEN" instruction ["ELSE" instruction]
```

Si l'évaluation de l'expression donne la valeur 0, alors c'est l'instruction qui suit le mot réservé ELSE qui est exécutée, mais uniquement dans le cas où la partie ELSE existe. Dans le cas où il n'y a pas de partie ELSE, l'exécution continue en séquence avec l'instruction qui suit logiquement l'instruction IF.

Lorsque des instructions IF sont imbriquées, la partie ELSE se rapporte toujours au IF THEN immédiatement précédent.

5.4 Les instructions répétitives

Leria supporte trois instructions de répétition. La première forme est la plus générale :

```
while_instruction ::= "WHILE" expression "DO" instruction
```

Tant que l'expression est évaluée à une valeur différente de 0, l'instruction qui suit le DO est exécutée (cette instruction peut évidemment être une instruction composée). Si l'expression vaut 0, la while_instruction est terminée. Ainsi, si à la première évaluation de l'expression, celle-ci a comme valeur 0, l'instruction suivant le DO n'est jamais exécutée.

Dans la deuxième forme de répétition, les instructions placées entre REPEAT et UNTIL sont exécutées toujours au moins une fois.

```
repeat_instruction ::= "REPEAT" instruction ";" { instruction ";" } "UNTIL" expression
```

Les instructions comprises entre REPEAT et UNTIL sont exécutées aussi longtemps que l'expression est évaluée à 0. Une fois que l'expression est évaluée à une valeur différente de 0, l'instruction REPEAT UNTIL est terminée.

La dernière forme de répétition s'écrit :

for_instruction ::= "FOR" accès_variable ":=" expression "UNTIL" expression "BY" expression
"DO" instruction

La variable accédée tout de suite après le mot réservé FOR est appelée *variable de contrôle*. L'instruction FOR est une instruction WHILE adaptée en ce sens que l'incrémentation de la variable de contrôle s'effectue automatiquement à la fin de l'exécution du corps de la boucle désignée par instruction. Les trois expressions sont évaluées à chaque tour de boucle.

L'instruction FOR a la sémantique suivante : la variable de contrôle est initialisée avec la valeur résultant de l'évaluation de l'expression suivant le symbole « := ». Si l'expression suivant le UNTIL n'est pas vérifiée, alors instruction est exécuté. Dans le cas contraire, l'exécution continue avec l'instruction suivant logiquement l'instruction FOR. À la fin de l'exécution de instruction, la variable de contrôle est incrémentée de la valeur fournie par l'évaluation de l'expression suivant BY. Le fait de modifier la valeur de la variable de contrôle dans instruction a un effet indéfini. Exemple :

```
PROGRAM test_for;
VAR int : i;
VAR numeric : j;
BEGIN
  FOR i := 1 UNTIL i > 15 BY 2 DO WRITE(i:3:0); WRITE(\n);
  FOR j := 10. UNTIL j < 0. BY -1.1 DO WRITE(j:4:1); WRITE(\n);
END.
```

Ce programme produit le résultat suivant :

```
  1  3  5  7  9 11 13 15
10.0 8.9 7.8 6.7 5.6 4.5 3.4 2.3 1.2 0.1
```

Attention : il n'est pas permis d'utiliser l'instruction RETURN dans le corps d'une instruction FOR !

5.5 L'instruction vide et l'instruction d'arrêt

Dans certains cas, il peut être intéressant de signifier au compilateur que l'on ne veut pas exécuter d'instructions. Plutôt que de laisser cette place vide (comme en Pascal), il faut l'indiquer clairement à travers une instruction :

instruction_vide ::= "SKIP"

Elle ne provoque l'exécution d'aucune instruction machine au moment de l'exécution du programme. Exemple :

```
IF x > 3 THEN { ne rien faire } SKIP ELSE write("ok",\n);
```

L'instruction HALT provoque l'arrêt immédiat de l'exécution du programme et le retour au système d'exploitation hôte :

halt_instruction ::= "HALT"

5.6 L'instruction RETURN

return_instruction ::= "RETURN" [expression]

Cette instruction ne peut être utilisée que dans un bloc fonction et provoque l'arrêt de l'exécution de la fonction et le retour au programme appelant.

Si la fonction n'est pas du type void, l'instruction RETURN doit être suivie d'une expression dont l'évaluation fournira une valeur qui sera la valeur retournée par la fonction. Cette expression doit être du même type que le type de retour indiqué dans la définition/déclaration de la fonction.

Une fonction de type procédure, c'est-à-dire dont le type de retour est void, ne doit pas nécessairement contenir une instruction RETURN. En effet, la fin de l'instruction composée du bloc fonction provoque un retour automatique au programme appelant. Une fonction peut contenir plusieurs instructions RETURN. Le corps d'une instruction FOR ne peut contenir une instruction RETURN. De même, il n'est pas permis d'utiliser cette instruction dans l'instruction composée du bloc principal d'un programme.

5.7 L'appel de fonction

Une fonction dont le type de retour est int, char ou numeric peut être utilisée comme un opérande quelconque dans une expression. Une fonction dont le type de retour est void est appelée une procédure et ne peut être utilisée que comme une instruction.

```
appel_fonction ::= identificateur "(" [liste_paramètres_effectifs] ")"
liste_paramètres_effectifs ::= expression {"," expression}
```

La liste des paramètres effectifs doit correspondre exactement en nombre et en type à la liste des paramètres formels déclarés dans la définition de la fonction.

6 Les fonctions

Il existe deux types de fonctions dans *Leria*, les fonctions qui retournent un des types simples char, int ou numeric et celles qui ne retournent rien. Ces dernières sont alors de type procédure (void). Dans un programme, les fonctions éventuelles doivent être définies soit au niveau du bloc principal dans la partie définition de fonctions, soit dans le bloc d'une fonction. Dans ce dernier cas, nous parlons alors de fonctions imbriquées.

```
partie_définition_fonctions ::= { définition_fonction | declaration_avant }
declaration_avant ::= "FORWARD" signature
définition_fonction ::= signature bloc ";"
signature ::=  identificateur_type "FUNCTION" identificateur "(" [ liste_paramètres_formels ] ")" ";"
liste_paramètres_formels ::= définition_paramètre { ";" définition_paramètre }
définition_paramètre ::= identificateur ":" ["VAR"] identificateur
```

Le type de retour de la fonction précède le mot réservé FUNCTION. Voici un exemple d'une procédure n'admettant pas de paramètres :

```
void FUNCTION message_coucou();
BEGIN
  WRITE("coucou");
END;
```

Il est souvent plus intéressant d'utiliser des fonctions avec paramètres. *Leria* supporte deux modes de passage de paramètres, le passage par valeur (par défaut) et le passage par variable (référence) qui est obtenu en faisant précéder le nom du paramètre par le mot réservé VAR. Le type du paramètre est indiqué au compilateur à l'aide de l'identificateur de type précédant le symbole spécial « : ». Voici une fonction utilisant un paramètre passé par valeur :

```
numeric FUNCTION carre(numeric : x);
BEGIN
  RETURN x*x;
END;
```

Lorsqu'un paramètre est passé par valeur, l'argument effectif est une expression dont la valeur est évaluée au moment de l'activation de la fonction. Cette valeur est placée dans la zone mémoire réservée au paramètre formel correspondant. Dans le cas où le paramètre est passé par variable, l'argument effectif doit être un accès à une variable. Lors de l'activation de la fonction, un lien temporaire est créé entre le nom de l'argument effectif et le paramètre formel. Le paramètre formel est en quelque sorte un alias pour le paramètre effectif avec comme conséquence que la fonction a tous les droits sur le paramètre effectif. À la fin de l'exécution de la fonction, ce lien est détruit.

Comme les expressions peuvent contenir des appels de fonction, les expressions suivantes sont bien formées :

`carre(3.)` est évalué à 9
`carre(carre(3.)-1.)` est évalué à 64
si **`i`** est une variable de type numeric valant 4 alors
`carre(i)` est évalué à 16 et **`i`** est inchangé.

Lors du passage d'un paramètre par valeur, l'argument effectif passé lors de l'appel de la fonction n'est jamais modifié. Ceci n'est pas le cas lorsqu'on utilise le passage par variable (référence). Exemple : pour obtenir le carré d'un nombre, on aurait pu définir une procédure avec deux paramètres, dont le deuxième passé par variable contiendrait le résultat au retour dans le programme appelant.

```
void FUNCTION carre_bis(numeric : x; numeric : VAR carre)
BEGIN
  carre := x*x;
END;
```

Si avant l'appel de cette procédure, **`i`** est une variable de type numeric de valeur 5, après exécution de l'instruction **`carre_bis(i,i)`**, la variable **`i`** est changée et vaut 25 !

Comme le montre l'exemple précédent, ce mode de passage doit être utilisé avec prudence. Il est néanmoins très utile pour passer des variables de type tableau à une fonction. En effet, dans le cas d'un passage par valeur il y a une copie du tableau qui est passé à la fonction et cette opération est très coûteuse en temps.

Leria supporte la récursivité, c'est-à-dire que l'identificateur définissant le nom de la fonction peut être utilisé à l'intérieur de l'instruction composée du bloc fonction. Voici un exemple d'une fonction récursive :

```
PROGRAM factorielle;
VAR int : f;

  int FUNCTION fact(int : n);
  BEGIN
```

```
    IF n < 2 THEN RETURN 1
             ELSE RETURN n*fact(n-1);  { appel récursif }
  END;

BEGIN
  WRITE("Entrez un entier : "); READ(f);
  WRITE(f,"! = ",fact(f));
END.
```

Leria connaît de nombreuses fonctions prédéfinies :

abs(*arg1*)	retourne la valeur absolue de *arg1*
acos(*arg1*)	retourne le cosinus inverse de *arg1*, *arg1* est un réel compris entre -1 et 1
asin(*arg1*)	retourne le sinus inverse de *arg1*, *arg1* est un réel compris entre -1 et 1
atan(*arg1*)	retourne la tangente inverse de *arg1*, *arg1* est un réel
cos(*arg1*)	retourne le cosinus de *arg1*, *arg1* est exprimé en radians
exp(*arg1*)	retourne e^{arg1}
ln(*arg1*)	retourne le logarithme népérien de *arg1*
power(*arg1*, *arg2*)	retourne $arg1^{arg2}$
round(*arg1*)	retourne *arg1* arrondi au numérique entier le plus proche
sin(*arg1*)	retourne le sinus de *arg1*, *arg1* est exprimé en radians
sqr(*arg1*)	retourne le carré de *arg1*
sqrt(*arg1*)	retourne la racine carrée de *arg1*
tan(*arg1*)	retourne la tangente de *arg1*, *arg1* est exprimé en radians
trunc(*arg1*)	retourne *arg1* tronqué de ses chiffres décimaux

Toutes les fonctions précédentes admettent des arguments de type numeric et renvoient une valeur de ce type.

Les fonctions suivantes sont particulières en ce sens qu'elles travaillent avec le type int défini dans le processeur *MaP.*

randomize(*arg1*)	initialise le générateur de nombres aléatoires avec *arg1*qui doit être un int !
rand()	retourne un nombre pseudo-aléatoire de type int compris entre 0 et 32767 inclus (7FFF)

time() retourne le nombre de secondes écoulées depuis une date fixée définie parle réalisateur du compilateur (ici 1.1.1970, 0h 0m 0s GMT). Ce nombre est un int.

impair(*arg1*) retourne 1 (vrai) si *arg1* est impair, 0 (faux) autrement. *arg1* est un nombre de type int.

numint(*arg1*) renvoie l'argument de type numeric transformé en int

intnum(*arg1*) renvoie l'argument de type int transformé en numeric

Les deux fonctions suivantes permettent de travailler avec des caractères :

ord(*arg1*) renvoie un int qui est le code ASCII de l'argument *arg1* qui doit être un char

chr(*arg1*) retourne un char qui est le caractère dont l'argument *arg1* de type int est le code ASCII

Une fonction peut être déclarée sans que l'on doive la définir de suite. Ceci est possible grâce au mot réservé FORWARD qui doit alors précéder la signature de la fonction qui ne doit pas être suivie du bloc de la fonction.

Lors d'une déclaration en avant, le programmeur doit spécifier complètement la fonction en précisant le type de retour, le nom de la fonction ainsi que le type et le mode de transmission de chacun de ses arguments. Les noms de ces arguments ne jouent aucun rôle. Une même fonction peut être déclarée de multiples fois en avant.

Cette facilité permet la création de fonctions s'appelant mutuellement, comme dans l'exemple suivant :

```
PROGRAM mutuellement;

VAR int : i;

FORWARD int FUNCTION fy(int :);

int FUNCTION fx(int : x);
BEGIN
  IF x > 0 THEN RETURN fy(x-1) { appelle fy }
           ELSE RETURN 0;
END;

int FUNCTION fy(int : x);
BEGIN
  IF x > 0 THEN RETURN fx(x-1) { appelle fx }
           ELSE RETURN 1;
END;

int FUNCTION pair(int : x);
BEGIN
  IF x < 0 THEN RETURN fy(-x)
           ELSE RETURN fy(x);
END;

BEGIN
  FOR i := -5 UNTIL i > 5 BY 1 DO
    IF pair(i) THEN WRITE(i, " est pair",\n)
```

```
                ELSE WRITE(i, " est impair",\n);
END.
```

7 La grammaire du langage Leria

programme_leria ::= "PROGRAM" identificateur ";" bloc "."
bloc ::= partie_définition_constantes
partie_définition_types
partie_définition_variables
partie_définition_fonctions
instruction_composée

partie_définition_constantes ::= { définition_constante }
définition_constante ::= "CONST" identificateur "=" ["–"] const_numérique |
"[" – "]" const_entier | const_caractère ";"
const_numérique ::= littéral_numérique | identificateur
const_entier ::= littéral_entier | identificateur
const_caractère ::= littéral_caractère | identificateur

partie_définition_types ::= { définition_type }
définition_type ::= "TYPE" identificateur "=" définition_tableau ";" |
"TYPE" identificateur "=" définition_record ";"
définition_tableau ::= "ARRAY" "[" const_entière "]" "OF" identificateur
définition_record ::= "RECORD" définition_champs ";" { définition_champs ";" } "END"
définition_champs ::= identificateur ":" identificateur { "," identificateur }

partie_définition_variables ::= { définition_variable }
définition_variable ::= "VAR" identificateur ":" identificateur { "," identificateur } ";"

accès_variable ::= identificateur { "[" expression "]" | "." identificateur }

expression ::= terme | "–" expression | "NOT" expression |
expression opérateur_binaire expression
terme ::= accès_variable | "(" expression ")" | appel_fonction | const_numérique | const_caractère
opérateur_binaire ::="+" | "–" | "*" | "/" | "%" | "#" |"^" | ">" | "<" | ">=" | "<=" | "<>" | "=" | "OR" | "AND"

partie_définition_fonctions ::= { définition_fonction | declaration_avant }
declaration_avant ::= "FORWARD" signature
définition_fonction ::= signature bloc ";"
signature ::= identificateur_type "FUNCTION" identificateur "(" liste_paramètres_formels] ")" ";"
liste_paramètres_formels ::= définition_paramètre { ";" définition_paramètre }
définition_paramètre ::= identificateur ":" ["VAR"] identificateur

instruction_composée ::= "BEGIN" instruction ";" { instruction ";" } "END"
instruction ::= affectation | if_instruction | while_instruction | read_instruction |
write_instruction | return_instruction | instruction_vide | halt_instruction |
repeat_instruction | appel_fonction | instruction_composée | for_instruction

affectation ::= accès_variable ":=" expression

read_instruction ::= "READ" "(" accès_variable { "," accès_variable } ")"
write_instruction ::= "WRITE" "(" w_expr { "," w_expr } ")"
w_expr ::= expression [":" expression [":" expression]] | "\n" | littéral_alphanumérique

if_instruction ::= "IF" expression "THEN" instruction ["ELSE" instruction]

while_instruction ::= "WHILE" expression "DO" instruction

```
repeat_instruction ::= "REPEAT" instruction ";" { instruction ";" } "UNTIL" expression

for_instruction ::= "FOR" accès_variable ":=" expression "UNTIL" expression "BY" expression
                        "DO" instruction
instruction_vide ::= "SKIP"

halt_instruction ::= "HALT"

return_instruction ::= "RETURN" [ expression ]

appel_fonction ::= identificateur "(" [liste_paramètres_effectifs] ")"
liste_paramètres_effectifs ::= expression {"," expression}

identificateur ::= lettre { caractère }
caractère ::= lettre | chiffre | blanc_souligné
littéral_entier ::= chiffre { chiffre }
littéral_numérique ::= chiffre { chiffre }   [ "." { chiffre } ]

lettre ::= "a" | "b" | "c" |...| "y" | "z" | "A" | "B" | "C" |...| "Y" | "Z"
chiffre ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
blanc_souligné ::= "_"
```

8 *Exemple d'un programme Leria*

Afin de mieux comprendre les possibilités et limites de *Leria*, voici un programme complet écrit à l'aide du langage de programmation *Leria* illustrant à peu près toutes les possibilités offertes.

Le programme calcule la solution au problème classique du cavalier opiniâtre. Nous avons un échiquier 5 sur 5, et au départ le cavalier est placé sur la case de coordonnées (1, 1). Le but du jeu est de faire passer le cavalier par toutes les cases de l'échiquier en ne visitant chaque case qu'une seule fois et en effectuant le déplacement du cavalier tel qu'il est défini dans le jeu d'échecs[1]. La solution présentée dans le programme ci-dessous est récursive et utilise un tableau à deux dimensions ainsi que deux tableaux à une dimension servant à stocker les coordonnées des différents sauts permis au cavalier.

+2		3		2	
+1	4				1
y			C		
–1	5				8
–2		6		7	
	–2	–1	x	+1	+2

Le cavalier de coordonnées (x, y) peut visiter une des 8 cases indiquées.

1. Dans le jeu d'échecs, le cavalier avance en une fois d'une case horizontale ou verticale puis d'une case en diagonale (mouvement en L).

Partant du point de coordonnées (1, 1), une solution à trouver est la suivante :

	1	**2**	**3**	**4**	**5**
1	1	6	15	10	21
2	14	9	20	5	16
3	19	2	7	22	11
4	8	13	24	17	4
5	25	18	3	12	23

Un ordre de visite possible des 25 cases de l'échiquier par un cavalier partant de la position 1 de coordonnées (1,1).

```
{$r- désactiver le contrôle de validité des indices à l'exécution, cela
     augmente la vitesse d'exécution du programme } {$d}
PROGRAM prog_cavalier;
CONST n = 6;
CONST n_carre = 36;
CONST false = 0;
CONST true = 1;
CONST libre = 0;

TYPE tab_n = ARRAY[n] OF int;
TYPE tab_nn = ARRAY[n] OF tab_n;
TYPE tab_8 = ARRAY[8] OF int;

VAR int : i,j,succes;
VAR tab_8: a,b;
VAR tab_nn : damier;

void FUNCTION afficher_damier(tab_nn : VAR d);
VAR int : i,j;
BEGIN
  i := 1;
  write(\n);
  WHILE i <= n DO BEGIN
    j := 1;
    WHILE j <= n DO BEGIN
      write(d[i][j]:3:0,"  ");
      j := j+1;
    END;
    write(\n);
    i := i+1;
  END;
END;

void FUNCTION initialiser_damier(tab_nn : VAR d);
VAR int : i,j;
BEGIN
  i := 1;
  WHILE i <= n DO BEGIN
    j := 1;
    WHILE j <= n DO BEGIN
      d[i][j] := libre;
      j := j+1;
    END;
    i := i+1;
  END;
END;
```

```
void FUNCTION essai(int : i; int : x; int : y;
                    int : VAR f);
VAR int : k,u,v,termine;

      int FUNCTION case_libre(int : i; int : j);
      BEGIN
        RETURN damier[i][j] = libre;
      END;

      int FUNCTION valide(int : i; int : j);
      BEGIN
        RETURN (i >= 1) AND (i <= n) AND (j >= 1) AND (j <= n);
      END;

BEGIN
  k := 0;
  REPEAT
    k := k +1;
    termine := false;
    u := x+a[k]; v := y+b[k];
    IF valide(u,v)
       THEN IF case_libre(u,v)
              THEN BEGIN
                  damier[u][v] := i;
                    IF i < n_carre THEN BEGIN
                          essai(i+1,u,v,termine);
                          IF NOT termine THEN damier[u][v] := libre;
                    END
                    ELSE termine := true;
       END;
  UNTIL termine OR (k=8);
  f := termine;
END;

BEGIN
  { initialiser les coordonnées des différents sauts du cavalier }
  a[1] := 2;      b[1] := 1;
  a[2] := 1;      b[2] := 2;
  a[3] := -1;     b[3] := 2;
  a[4] := -2;     b[4] := 1;
  a[5] := a[4];   b[5] := -b[4];
  a[6] := a[3];   b[6] := -b[3];
  a[7] := a[2];   b[7] := -b[2];
  a[8] := a[1];   b[8] := -b[1];

  initialiser_damier(damier);
  damier[1][1] := 1;               { fixer la case de départ }
  afficher_damier(damier);
  essai(2,1,1,succes);
  IF succes THEN afficher_damier(damier)
            ELSE write(\n,"Pas de solution");
END.
```

9 *Utilisation du compilateur Leria*

Le compilateur *Leria* s'utilise comme décrit ci-dessous. Pour activer le compilateur, il suffit d'entrer la commande *Leria*. Dans ce cas, le système affiche un message indiquant à l'utilisateur la syntaxe de la commande[1] :

```
Compilateur LERIA v3.0 Copyright N. Silverio CU/CRP-CU (24/10/99)
Format : leria  nom_de_fichier[.ler]
```

Le compilateur produit toujours un fichier contenant les lignes numérotées du programme avec la liste des messages d'erreurs éventuels. Ce fichier porte le même nom que le fichier contenant le programme source, mais il a toujours l'extension *.lst*. Si l'utilisateur spécifie la directive de compilation {$d}, cette liste est augmentée du code machine déassemblé. De plus, il produit le fichier traduit en assembleur *MaP.* Ce fichier porte le même nom que le fichier contenant le programme source, mais il a toujours l'extension *.asm*. Le compilateur produit un troisième fichier, portant toujours le même nom que le fichier source, mais contenant cette fois-ci le code exécutable. Ce fichier porte l'extension *.int*. Finalement, le compilateur crée un dernier fichier avec l'extension *.dbg*.

Un des programmes les plus courts que l'on puisse écrire en *Leria* est le suivant :

```
PROGRAM bonjour;
BEGIN
  WRITE("Bonjour");
END.
```

Le lecteur peut entrer ce petit programme à l'aide de l'éditeur de son choix et le sauver sous un nom quelconque. Appelons ce fichier *test.ler* et soumettons-le au compilateur en tapant la commande :

```
leria test.ler
```

ou

```
leria test
```

Le système traduit le programme source en langage machine *MaP.* Pour exécuter ce programme, l'utilisateur doit entrer la commande RUN :

```
run test
```

Il faut bien remarquer que ceci n'est possible que parce que le programme *test.ler* a été compilé provoquant la création d'un fichier exécutable portant le même nom que le fichier source, mais avec l'extension *.int*.

1. La date indiquée peut être plus récente que dans ce document !