Version 2008

Notions d'algorithmique et de programmation avec LAZARUS/FreePascal

II^{e} B / I^{re} B

par :
MM. Hubert GLESENER, Jean-Claude HEMMER,
David MANCINI, Alexandre WAGNER

version Lazarus modifiée par M. David Mancini

MCours.com

Table des matières

1 Introduction 7
1.1 Généralités 7
1.2 Cycle de développement 7
1.3 Types d'applications 7
1.4 Exemple d'un cycle de développement 8
2 Affectation 11
2.1 Entrées-sorties 11
2.2 Les opérations arithmétiques. 12
2.3 Variables 13
2.4 Type d'une variable 14
2.5 Déclaration d'une variable 14
2.6 L'affectation 15
2.7 Initialisation d'une variable 16
2.8 L'instruction readln 17
2.9 Exercices 18
2.10 Différents types d'erreurs 19
3 Structure alternative 20
3.1 Problème introductif 20
3.2 Syntaxe 22
3.3 Les conditions (expressions logiques) 23
3.4 Exercices 24
4 Structure répétitive 26
4.1 Introduction 26
4.2 Exemple 26
4.3 Terminaison d'une boucle 27
4.4 Boucles for 27
4.5 Exercices 28
4.6 Un algorithme efficace 29
4.7 Exercices 31
5 Fonctions et procédures 33
5.1 Les fonctions 33
5.2 Les procédures 35
5.3 Portée 37
5.4 Passage des variables dans une fonction ou une procédure 42
6 Les structures de données composées 46

6.1 Les tableaux 46
6.2 Les enregistrements 49
7 Lazarus 55
7.1 Introduction 55
7.2 Les fichiers utilisés en Lazarus 55
7.3 L'approche Orientée-Objet 56
7.4 Passage FreePascal – Lazarus – un premier exemple 57
7.5 Calcul de la factorielle 59
7.6 Equation du second degré 62
7.7 Vérification du numéro de matricule 65
7.8 Une petite machine à calculer 69
7.9 Calcul matriciel - utilisation du composant StringGrid 73
8 La récursivité 78
8.1 Exemple 78
8.2 Définition : « fonction ou procédure récursive » 78
8.3 Etude détaillée d'un exemple 79
8.4 Fonctionnement interne 80
8.5 Exactitude d'un algorithme récursif 80
8.6 Comparaison : fonction récursive – fonction itérative 81
8.7 Récursif ou itératif ? 82
8.8 Exercices 83
9 Comptage et fréquence 84
9.1 Algorithme de comptage 84
9.2 Fréquence d'une lettre dans une liste 84
10 Recherche et tri 85
10.1 Introduction 85
10.2 Sous-programmes utiles 85
10.3 Les algorithmes de tri 86
10.4 Les algorithmes de recherche 91

Première partie :

Cours de II^{e}

Applications en console

1 Introduction

1.1 Généralités

L'objectif de ce cours est d'apprendre l'art de programmer. Au départ un énoncé sera analysé. Il sera ensuite étudié et finalement transformé en un programme bien structuré.

La solution informatique du problème posé est appelée *algorithme*. Celui-ci est indépendant du langage de programmation choisi. Cependant nous devons opter pour un langage de programmation dans lequel l'algorithme sera traduit. Dans ce cours nous allons utiliser le langage *Pascal*.

Un langage de programmation est un ensemble d'instructions qui permettent de décrire à l'ordinateur une solution possible du problème posé. Il existe une multitude de langages de programmation différents, chacun avec ses avantages et ses désavantages. Exemples : Fortran, Pascal, Ada, C, C++, C#, Java, …

Le langage *Pascal* est un langage de programmation évolué et polyvalent, dérivé de l'*Algol-60*. Il a été développé par Niklaus Wirth (*1934 -) au début des années 1970 à l'École Polytechnique Fédérale de Zurich. Au cours des années le langage *Pascal* a fortement évolué et la plus importante modification est sans doute l'incorporation de la notion de programmation orientée objet.

Lazarus pour sa part est la combination du langage *Free Pascal,* un dérivé de *Pascal,* auquel on a incorporé des notions graphiques.

1.2 Cycle de développement

Pour développer efficacement un programme nous suivons les étapes suivantes :

- Analyse du problème :
  - Quelles sont les données d'entrée ?
  - Quelles sont les données à la sortie que le programme doit produire ?
  - Comment devons-nous traiter les données d'entrée pour arriver aux données à la sortie ?
- Établissement d'un algorithme en utilisant un langage de programmation compris par le système.
- Traduction du programme *Pascal* en langage machine à l'aide d'un compilateur.
- Exécution et vérification du programme.

1.3 Types d'applications

Il existe deux types d'applications différents : les applications *en console* et les applications à *base graphique*. Les applications en console se limitent à des entrées et des sorties purement textuelles. Elles n'utilisent pas de graphismes. Ce type d'applications était usuel avant l'arrivée de *Windows*.

Les applications à base graphique nécessitent un environnement graphique comme par exemple *Windows* de Microsoft. Ce type d'applications est caractérisé par l'utilisation de fenêtres pour l'entrée et la sortie d'informations.

En II^{e} nous allons nous limiter aux applications en console. Ils se développent plus facilement que les applications à base graphique qui nécessitent des techniques de programmation plus évoluées.

1.4 Exemple d'un cycle de développement

Pour rédiger, compiler et déboguer[1] un programme *Pascal*, nous avons besoin d'un milieu de développement. Dans le cadre du cours de cette année, nous pourrions utiliser tout compilateur répondant aux spécifications du langage *Pascal*.

Nous allons utiliser le milieu de développement *Lazarus/FreePascal* qui permet de développer des applications du type console aussi bien que des applications à base graphique.

1.4.1 Énoncé du premier programme

Pour commencer nous voulons établir un programme qui affiche un simple message sur l'écran, à savoir : « Bonjour tout le monde ! ».

1.4.2 Analyse du problème

Il n'y a pas de données à l'entrée. Il n'y a pas de traitement de données. À la sortie, il y a un message à afficher : « Bonjour tout le monde ! ».

1.4.3 Programme Pascal

```
program Bonjour;
begin
  writeln('Bonjour tout le monde !')
end.
```

1.4.4 Milieu de développement Lazarus/FreePascal

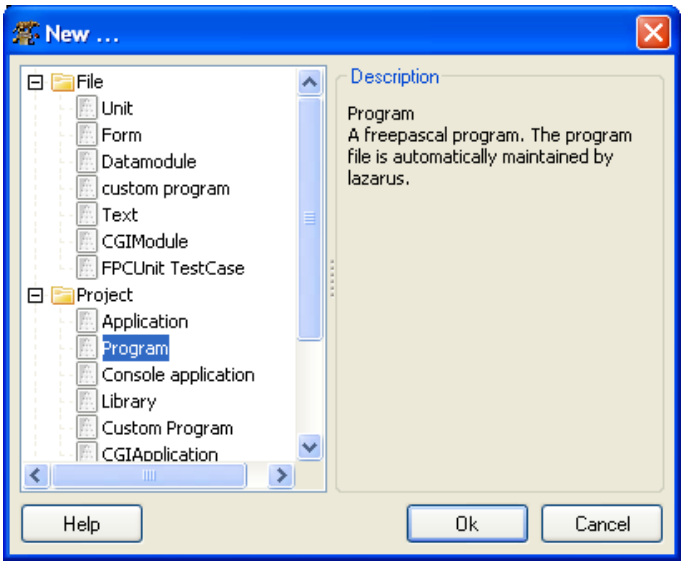
Avant de démarrer *Lazarus*, il est impératif de créer pour chaque programme un sous-répertoire individuel. Tous les fichiers relatifs au programme sont alors sauvegardés dans ce sous-répertoire.

Créons donc un tel sous-répertoire que nous appellerons *Bonjour*.

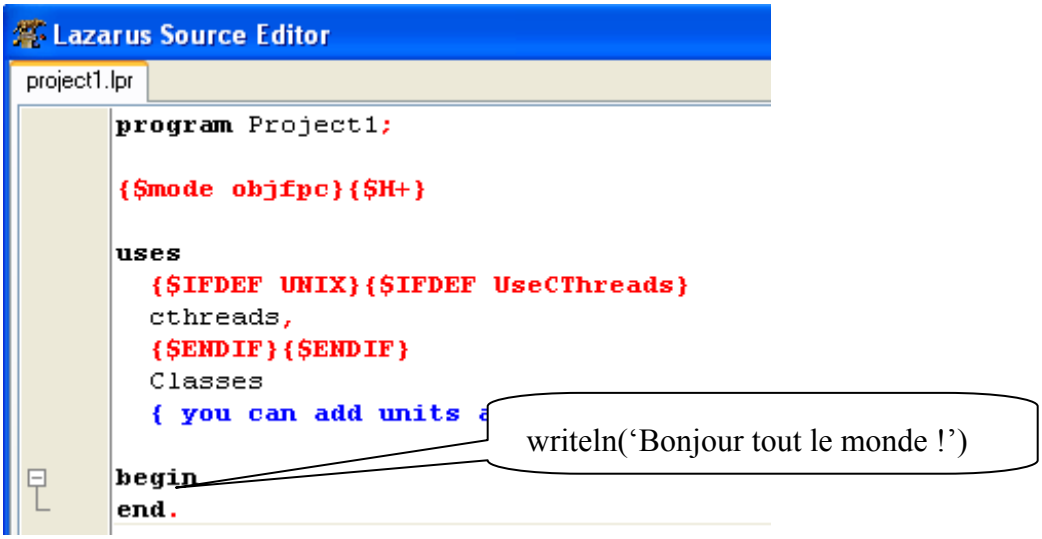
Nous lançons ensuite le programme *Lazarus*. Il nous propose immédiatement les structures pour développer une application graphique. Comme nous n'utilisons pas ce type d'applications pour l'instant, nous fermons ce projet à l'aide du menu : **File → Close**.

Ensuite nous créons une nouvelle application de type console à l'aide du menu : **File → New**.

[1] Chercher à éliminer les erreurs.



Un double clic sur l'option « Program » ouvre une nouvelle fenêtre qui reçoit le code de notre programme :



Recopions ensuite le programme proposé plus haut et sauvegardons-le à l'aide du menu **File → Save**.

L'extension pour les programmes *Lazarus* est « lpi », qui est une abréviation de « Lazarus Project Information file ». Nous constatons alors que dans le texte «**`program`** Project1;» le mot «Project1» est remplacé par le nouveau nom du fichier mais sans l'extension.

1.4.5 Exécution et test du programme

Après avoir copié le programme, nous pouvons essayer de le compiler et de l'exécuter une première fois à l'aide du menu **Run → Run** (raccourci clavier : F9) ou en appuyant sur le bouton (semblabe au bouton « play » d'un magnétoscope) avec le petit triangle vert.

Si nous exécutons le programme, nous constatons qu'une fenêtre apparaît brièvement mais nous ne voyons pas son contenu. Cette fenêtre est supposée contenir les entrées et les sorties de notre programme. Cependant dans notre cas, nous n'apercevons pas la sortie, car la fenêtre disparaît immédiatement. Cette disparition provient du fait que si un programme atteint sa fin, sa fenêtre est immédiatement fermée.

C'est pourquoi **nous devons rajouter à la fin de chaque programme l'instruction *readln***, qui attend que l'utilisateur appuie sur une touche du clavier et permet ainsi de « geler » la fenêtre.

1.4.6 Version définitive du programme

```
program Bonjour;
{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

begin
  writeln('Bonjour tout le monde !');   {affichage}
  readln;                               {attente}
end.
```

En général, les parties entre accolades sont des commentaires que le programmeur peut ajouter pour expliquer le fonctionnement du programme.

Ici, les parties marquées à gauche d'une bordure double ne font pas partie du programme proprement dit. Lazarus/FreePascal a ajouté des instructions au compilateur qui indiquent entre autres au système qu'il s'agit d'une application en mode console et non pas d'une application Windows.[2]

[2] Pour transférer un ancien programme écrit en Pascal (Turbo Pascal, Delphi ou similaire) vers Lazarus/FreePascal, il suffit d'ouvrir une nouvelle application en mode console, de copier la partie située entre **begin** et **end** et de sauvegarder le nouveau programme. Le programme va alors s'exécuter comme attendu.

2 Affectation

2.1 Entrées-sorties

Dans la suite nous allons transformer notre premier programme pour qu'il produise un autre effet-net. Les instructions que nous allons y ajouter vont toutes se placer entre `begin` et `end`.

Les instructions `write` et `writeln` (identique à `write` mais suivi d'un saut à la ligne) permettent d'afficher des données à l'écran. Ces instructions sont suivies d'un point-virgule comme presque toute instruction *Lazarus/FreePascal*.

Exemples :

```
writeln(4785); {affiche le nombre 4785}
writeln(47.85); {affiche le nombre décimal 47,85 (point décimal !!!)}
writeln('Lazarus'); {affiche la chaîne de caractères « Lazarus »}
```

Retenons ici la règle de syntaxe importante :

Chaque instruction se termine par un point-virgule « ; ».

Devant ***end*** *on peut laisser de côté le point-virgule.*

Rappelons qu'il faut terminer par l'instruction `readln` qui fait attendre le programme jusqu'à ce que l'utilisateur appuie sur la touche « enter ». Cela est nécessaire pour laisser à l'utilisateur le temps de lire l'affichage avant que le programme ne se termine et ne ferme la fenêtre.

Après avoir ajouté ces instructions au programme, la partie centrale de celui-ci se présente sous la forme suivante :

```
begin
  writeln(4785);
  writeln(47.85);
  writeln('lazarus');
  readln;
end.
```

Si nous exécutons le programme maintenant, l'affichage est le suivant :

```
4785
4.78500000000000E+0001
lazarus
```

L'affichage de 4.78500000000000E+0001 peut surprendre. Lazarus/FreePascal utilise d'office une écriture lourde mais précise du nombre 47.85. Il le transforme en notation scientifique. Il faut donc lire : $4.785 \cdot 10^{1}$.

L'instruction `writeln` permet de formater l'affichage de la façon suivante :

```
writeln(47.85:10:8); {10 chiffres au total, dont 8 derrière la virgule)
writeln(4785:6); {affichage à 6 places, 2 blancs suivis des 4 chiffres 4785 }
```

Exercice :

Apportez ces modifications dans le programme et relancez-le pour vous en convaincre.

Essayez d'autres paramètres d'affichage pour en constater l'effet !

Ces formatages sont nécessaires pour garantir une bonne présentation à l'écran, surtout si on prévoit d'afficher beaucoup d'informations, éventuellement en colonnes ou en tableaux.

2.2 Les opérations arithmétiques.

Lazarus permet d'effecteur non seulement les opérations arithmétiques mais toutes les opérations mathématiques usuelles.

- **+** est le signe de l'addition,
- **-** celui de la soustraction,
- ***** celui de la multiplication et
- **/** représente une division.

Les parenthèses peuvent être utilisées comme en mathématiques et elles peuvent être imbriquées à plusieurs niveaux. Les crochets et accolades qui apparaissent dans des expressions mathématiques doivent être remplacés par des parenthèses. Lazarus/FreePascal effectue les expressions arithmétiques en appliquant correctement les règles de priorité.

Les opérations **div** et **mod** sont seulement définies pour des nombres entiers :

- **div** donne le quotient (entier) et
- **mod** le reste de la division euclidienne.

`Exemples : 15 div 7 = 2 et 15 mod 7 = 1 car 15 = 7 * 2 + 1`

Les fonctions

- **sqrt** (racine carrée, « square root ») et
- **sqr** (carré, « square »)

sont aussi définies et utilisées comme en mathématiques. Remarquez que les parenthèses sont obligatoires.

Exemples	**Affichage**
`writeln(5+3*7)`	`26`
`writeln((2-7)*5+3/(2+4))`	`-2.45000000000000E+0001`
`writeln(57 div 15)`	`3`
`writeln(57 mod 15)`	`12`
`writeln(sqrt(9))`	`3.00000000000000E+0000`
`writeln(sqr(5))`	`25`
`writeln(sqrt(sqr(3)+sqr(4)))`	`5.00000000000000E+0000`

Exercice 2-1

Écrivez une instruction qui calcule et affiche le résultat de chacune des expressions suivantes :

a) $15+7\cdot 4+2:3$

b) $17-\left[14-3\cdot\left(7-2\cdot 8\right)\right]\cdot 2$

c) $\dfrac{12-5\cdot 7}{13+3\cdot 4}$

d) $\dfrac{(12-5)\cdot 7}{(13+3)\cdot 4}$

e) $\left(\dfrac{2-7}{5}\right)^2-\dfrac{(3-5)^2}{7-5}$

f) $\sqrt{48^2+20^2}$

Exercice 2-2

Effectuez (sans l'aide de l'ordinateur) les expressions suivantes :

a) `86 div 15`
b) `86 mod 15`
c) `(5-(3-7*2)*2)*2`
d) `sqrt(sqr(9)+19)`
e) `145-(145 div 13)*13`
f) `288 div 18`
g) `288/18`

2.3 Variables

Un concept fondamental de l'informatique est celui de la variable. Comme en mathématiques une variable représente une certaine valeur. Cette valeur peut être un nombre, une chaîne de caractères, un tableau, …

Le nom d'une variable n'est pas limité à une seule lettre.

Un nom de variable admissible (accepté par le système) commence par une lettre et peut être suivi d'une suite de symboles qui peuvent être des lettres, des chiffres ou le blanc souligné « _ » (« underscore »).

Un nom valable sera appelé *identificateur* dans la suite. Vu que le nombre de variables d'un programme peut être très grand, **il est utile de choisir des identificateurs qui donnent une indication sur l'usage envisagé de la variable**.

Par exemple : Dans un programme calculant une puissance, les noms de variables `base` et `exposant` seraient appropriés. Il est important de noter que ce choix est seulement fait pour simplifier la lecture et la compréhension du programme et que la signification éventuelle du nom échappe complètement à Lazarus/FreePascal.

Lazarus/FreePascal ne distingue pas entre majuscules et minuscules. Les identificateurs `a` et `A` ne peuvent donc être discernés. Dans la suite nous n'utiliserons que des lettres minuscules.

2.4 Type d'une variable

Chaque variable possède un type précis.

Dans la suite nous utiliserons les types **`integer, real, boolean, char, string`** qui sont utilisés pour représenter :

Type	Valeur	spécifications
`integer`	nombre entier relatif	entre -2^{63} et $2^{63}-1$
`real`	nombre décimal (en notation scientifique)	voir alinéa suivant
`boolean`	**true** ou **false**	indique si une condition est vraie ou fausse
`char`	un caractère	lettre, chiffre ou autre caractère
`string`	chaîne de caractères	p.ex. 'abfzr3_45*' ou '1235'

Le type **`real`** demande plus d'attention. Vu que l'écriture décimale d'un nombre réel (mathématique) peut nécessiter une infinité de chiffres décimaux, une représentation fidèle ne peut pas être garantie dans l'ordinateur et un nombre du type **`real`** représente un nombre appelé décimal en mathématiques.

Chaque langage de programmation impose des contraintes aux nombres.

En Lazarus/FreePascal un nombre de type **`real`** est représenté par un *nombre en virgule flottante* (« floating point »). C'est une notation scientifique où la mantisse peut avoir jusqu'à 15 chiffres décimaux significatifs et où l'exposant est compris entre –324 et +308. Ces bornes sont assez grandes pour garantir le bon fonctionnement de presque tout programme raisonnable.

Le type **`boolean`** qui admet seulement les deux valeurs **true** et **false** peut surprendre. Ce type est utilisé par Lazarus/FreePascal pour représenter l'état de vérité (par exemple d'une condition). Une expression comme *a<5* aura comme valeur **true** si *a* est effectivement inférieur à 5 et **false** dans le cas contraire.

Dans un programme, un caractère ou une chaîne de caractères sont entourés du symbole « ' » (apostrophe). Si la chaîne contient elle-même une apostrophe, alors il faut en mettre deux de suite, sinon Lazarus/FreePascal pense que cette apostrophe termine la chaîne. Ainsi l'expression « l'ordinateur » s'écrit `'l''ordinateur'` en Lazarus/FreePascal. Il ne faut surtout pas mettre de guillemets anglais « " » comme on a l'habitude de les mettre dans un texte.

Lorsque Lazarus affiche un caractère ou une chaîne de caractères par une instruction `write` ou `writeln` les apostrophes au début et à la fin sont omises. De même, une double apostrophe à l'intérieur d'une chaîne est affichée comme simple apostrophe.

Il est aussi déconseillé d'utiliser des lettres accentuées, car leur affichage en console n'est pas correct.

2.5 Déclaration d'une variable

Avant la première utilisation dans un programme, une variable doit impérativement être déclarée. Toutes les déclarations de variables suivent directement le mot réservé **`var`** et se présentent sous la forme :

```
var   <ident_a1>,<ident_a2>,...,<ident_an>: <type_a>;
      <ident_b1>,<ident_b2>,...,<ident_bm>: <type_b>;
      ...;
```

La syntaxe est à respecter scrupuleusement :

- Différents identificateurs de variables de même type sont séparés par une virgule (« , ») ;
- l'identificateur et le type sont séparés par un deux-points (« : ») ;
- le type est suivi d'un point-virgule (« ; »).

Ces déclarations se trouvent avant le bloc **begin ... end**.

Exemple :

```
program Project1;
{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

var  {déclarations de variables}
  a,exposant:integer;
  base,puissance:real;
  mot:string;
begin
  {instructions du programme}
end.
```

Cette déclaration permet au compilateur de réserver efficacement pour chaque variable l'espace mémoire nécessaire pour stocker la valeur ultérieure.

2.6 L'affectation

Jusqu'à présent les variables ont été déclarées mais elles ne représentent pas encore de valeur déterminée. L'opération qui consiste à attribuer une valeur à une variable est appelée affectation et elle respecte la syntaxe suivante :

```
<nom_de_variable> := <expression>;
```

Par exemple :

```
a:=45;
base:=12.34;
puissance:=3*3*3*3;
mot:='test';
```

Lazarus/FreePascal **effectue d'abord l'expression à droite de « := » et affecte (attribue) le résultat obtenu à la variable dont le nom se trouve à gauche de « := ».**

Pour que l'affectation soit possible, **il faut que le résultat de l'expression de droite existe et soit du même type que la variable !**

Ensuite le nom de la variable peut apparaître dans une expression.

Par exemple :

`nombre:=6;`	la valeur 6 est affectée à la variable `nombre`
`writeln(nombre+2);`	afficher `nombre+2`, c'est-à-dire afficher 8.

L'expression de droite peut aussi contenir des variables, y compris celle qui se trouve à gauche de l'affectation !

`nombre:=6;`	la valeur 6 est affectée à la variable nombre
`nombre:=nombre+2;`	effectuer le membre de droite : `nombre+2=8` et affecter le résultat 8 comme nouvelle valeur à la variable nombre. L'effet de l'instruction est donc `nombre:=8;`
`writeln(nombre);`	afficher la valeur actuelle de `nombre`, donc 8.

2.7 Initialisation d'une variable

Il est important de constater que la valeur d'une variable peut varier (d'où le nom) au cours de l'exécution du programme. La première affectation d'une variable est appelée initialisation de la variable. Avant l'initialisation, la variable a une valeur non définie ! L'initialisation d'une variable doit précéder l'apparition de la variable dans une expression.

En particulier une affectation comme

```
terme:=terme+1;
```

n'est utile que si la variable `terme` a été initialisée auparavant. Sinon le membre de droite, contenant terme n'est pas défini.

Il est important de remarquer que le fait de ne pas initialiser la variable ne va pas empêcher le programme de démarrer, mais le résultat sera vraisemblablement faux!

Exercice 2-3 (résolu)

Les variables a, b et c sont de type **`integer`**.

Faites un tableau reprenant la valeur de chaque variable **après** chaque ligne.

Un « ? » indiquera que la valeur de la variable n'est pas connue à cet instant !

Instruction	**a**	**b**	**c**
`a:=7`	7	?	?
`b:=4`	7	4	?
`a:=a+b`	11	4	?
`c:=2*a-3*b`	11	4	10
`b:=(a+c) div 4`	11	5	10
`c:=c-a`	11	5	-1

Exercice 2-4

Même question que pour l'exercice précédent.

Instruction	a	b	c
`a:=37 mod 7;`			
`b:=a*3;`			
`c:=a-2*b;`			
`a:=(a*3) div 5 + a;`			
`b:=(b-2)*(b-1);`			
`b:=b*b;`			

Exercice 2-5

Même question que pour l'exercice précédent, mais les variables sont maintenant de type **`real`** !

Instruction	a	b
`a:=9;`		
`b:=sqrt(a);`		
`b:=2*b/5+a;`		
`a:=sqr(a);`		
`b:=b/2+a/2;`		
`b:=(b+a)/2;`		

2.8 L'instruction readln

Il existe une autre méthode pour attribuer une valeur à une variable : la lecture de la valeur du clavier. L'instruction

```
readln(nombre);
```

attend que l'utilisateur introduise au clavier une valeur, suivie de « enter ». Le programme lit ensuite cette valeur et l'affecte à la variable nombre. Si la valeur n'est pas compatible avec le type de la variable, alors le système arrête le programme et affiche un message d'erreur. Pour éviter ce type de problème, il est utile d'informer l'utilisateur sur ce qu'il est censé introduire :

```
write('Veuillez introduire un nombre: ');
readln(nombre);
```

Il est important de remarquer qu'**il faut introduire une valeur** et non pas une expression à évaluer.

Si le programme attend un nombre et que l'utilisateur introduit par exemple 2+5, alors le système va interrompre l'exécution avec un message d'erreur !

Si le programme attend une chaîne de caractères, alors ce type d'erreur ne peut pas arriver car '2+5' est une chaîne valable. Il n'est évidemment pas garanti que le programme va en faire un usage « intelligent ». Il ne va en aucun cas évaluer l'expression !

2.9 Exercices

Exercice 2-6 : « Swap »

a) Écrivez un programme qui lit 2 nombres entiers du clavier et les affecte respectivement aux variables `nombre1` et `nombre2`.

b) Écrivez ensuite des instructions qui permettent d'échanger (« swap ») les valeurs des variables `nombre1` et `nombre2`.

Exercice 2-7 : Moyenne arithmétique

Écrivez un programme qui lit 2 nombres réels du clavier, calcule leur moyenne (arithmétique) et l'affiche.

Exercice 2-8 : Moyenne harmonique

Écrivez un programme qui lit 2 nombres réels strictement positifs du clavier, calcule leur moyenne harmonique et l'affiche.

Rappelons que la moyenne harmonique de 2 nombres strictement positifs est donnée par la formule : $\frac{1}{m_h} = \frac{\frac{1}{a} + \frac{1}{b}}{2}$. (Il est recommandé de transformer la formule avant de l'utiliser dans le programme !)

Exercice 2-9 : TVA

Écrivez un programme qui lit un nombre réel strictement positif qui représente le prix d'un article quelconque (TVA de 15% comprise). Le programme calculera le prix hors TVA de l'article et l'affichera.

Exercice 2-10 : Loi des résistances

Écrivez un programme qui lit 2 nombres réels, strictement positifs, représentant 2 résistances, du clavier. Le programme calculera ensuite la résistance résultante par la loi d'Ohm, au cas où on met ces 2 résistances en parallèle et l'affichera.

Exercice 2-11 : Aire d'un triangle par la formule d'Héron

Écrivez un programme qui lit 3 nombres réels, représentant les mesures des 3 côtés d'un triangle. (Ces nombres doivent donc vérifier l'inégalité triangulaire).

Le programme devra ensuite calculer, à l'aide de la formule d'Héron, l'aire du triangle correspondant à ces mesures et l'afficher.

Exercice 2-12 : Division euclidienne

Écrivez un programme qui lit le dividende et le diviseur non nul d'une division et qui affiche la division euclidienne résultante.

Exemple : Si le dividende est 34 et le diviseur 8, le programme devra afficher 34=8*4+2.

Exercice 2-13 : Suite arithmétique

Écrivez un programme qui lit le premier terme et la raison d'une suite arithmétique. Le programme lira ensuite un nombre naturel `n`, calculera le n^e terme ainsi que la somme des `n` premiers termes de la suite et les affichera.

2.10 Différents types d'erreurs

Si un programme ne produit pas le résultat attendu ou pas de résultat du tout, cela peut être dû à des erreurs de nature très variée. Selon l'instant où le problème se manifeste, on distingue les erreurs et les alertes décrites ci-dessous.

2.10.1 Erreur de compilation (« compilation error »)

On parle d'erreur de compilation, si Lazarus/FreePascal n'est pas capable de traduire correctement le programme.

L'erreur la plus typique de ce genre est l'erreur de syntaxe (« syntax error »), qui indique que l'écriture n'a pas été respectée : mot clé mal écrit, « ; » manquant, instruction ou expression mal formées... .

D'autres erreurs de compilation apparaissent, si le programme utilise auparavant une variable non déclarée ou si dans une affectation le type de la valeur et de la variable sont incompatibles.

2.10.2 Erreur d'exécution (« runtime error »)

Pour ce type d'erreur, l'exécution du programme commence correctement, mais elle est interrompue par le système, suite à un événement non prévu.

Des exemples typiques de cette sorte d'erreurs sont la division par zéro ou la lecture d'une valeur ne correspondant pas au type de la variable.

2.10.3 Erreur de programmation

Ce type d'erreur ne provoque aucune réaction du système, mais il se manifeste par un résultat incorrect. Dans ce cas il faut revoir l'algorithme. Le programme est mal conçu !

2.10.4 Alertes (« warnings »)

Une alerte est un message que Lazarus/FreePascal affiche lors de la compilation pour indiquer non pas une erreur proprement dite, mais une instruction suspecte, de mauvais style.

Par exemple une alerte est affichée si une variable non initialisée est utilisée ou si une variable est bien déclarée mais jamais utilisée ou encore si sa valeur n'est jamais utilisée.

Une alerte sert souvent d'indication pour trouver une erreur de programmation.

3 Structure alternative

3.1 Problème introductif

3.1.1 Énoncé

Les compteurs du gaz affichent 5 chiffres et permettent d'indiquer des consommations allant de 00000 à 99999 m^3. Lorsque la consommation est plus élevée que cette valeur, le compteur recommence à 00000.

Pour calculer la consommation, on retranche le dernier affichage connu de l'affichage actuel. La consommation ainsi obtenue est multipliée par le prix du gaz au m^3 afin d'obtenir le prix à payer.

De nos jours, le prix à payer est calculé par ordinateur. Concevez un programme permettant de réaliser ce calcul.

3.1.2 Première solution

```
program gaz1;
{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

var A1,A2,CONSO:integer;
  PM3,PRIX:real;
begin
  write('Entrez le dernier affichage connu et l''affichage actuel : ');
  readln(A1,A2);
  write('Entrez le prix au m3 : ');
  readln(PM3);
  CONSO:=A2-A1;
  PRIX:=CONSO*PM3;
  writeln('Le prix a payer est de : ',PRIX);
  readln;
end.
```

3.1.3 Exemples d'exécution

A1=78000

A2=98000

PM3=0.23

Instruction	A1	A2	PM3	CONSO	PRIX
`readln A1,A2`	78000	98000	?	?	?
`readln PM3`	78000	98000	0.23	?	?
`CONSO:=A2-A1`	78000	98000	0.23	20000	?
`PRIX:=CONSO*PM3`	78000	98000	0.23	20000	4600

A1=98000

A2=18000

PM3=0.23

Instruction	A1	A2	PM3	CONSO	PRIX
`Readln A1,A2`	98000	18000	?	?	?
`readln PM3`	98000	18000	0.23	?	?
`CONSO:=A2-A1`	98000	18000	0.23	-80000	?
`PRIX:=CONSO*PM3`	98000	18000	0.23	-80000	-18400

On constate que dans le premier exemple, le résultat est correct. Par contre, le résultat du deuxième exemple n'a pas de sens. Il ne peut y avoir ni de consommation ni de prix négatif.

La raison pour laquelle on obtient ces résultats est que le programme ne tient pas compte du fait que le compteur peut dépasser 100000 et que dans ce cas, il recommence à 0. Dans le deuxième cas l'état du compteur n'est pas 18000 mais vaut 18000+100000=118000.

En conséquence, il faudrait ajouter 100000 à l'affichage actuel du compteur au cas où cette valeur est inférieure au dernier affichage connu.

La structure alternative permet de réaliser ce genre d'opérations conditionnelles.

3.1.4 Deuxième solution

```
program gaz2;
{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

var A1,A2,CONSO:integer;
    PM3,PRIX:real;
```

```
begin
  write('Entrez le dernier affichage connu et l''affichage actuel : ');
  readln(A1,A2);
  write('Entrez le prix au m3 :');
  readln(PM3);
  if A2>=A1 then
    CONSO:=A2-A1
  else
    CONSO:=A2-A1+100000;
  PRIX:=CONSO*PM3;
  writeln('Le prix a payer est de : ',PRIX);
  readln;
end.
```

On vérifie que le programme ci-dessus fournit les résultats voulus dans les cas de figure précédents. Par contre, le cas où le compteur fait plusieurs fois le tour pour arriver au-delà de 200000 n'est pas traité.

3.2 Syntaxe

3.2.1 Syntaxe simplifiée

```
if <condition> then
  <instruction>;
```

3.2.2 Syntaxe complète

```
if <condition> then
  <instruction>
else
  <instruction>;
```

Remarque importante :

- L'instruction **`if ... then ... else`** est à considérer comme **une seule instruction**. Elle se termine donc par un seul point-virgule « ; » à la fin.
- Il n'y a pas de « ; » après **`then`** ni avant **`else`**
- S'il y a plusieurs instructions, il faut les mettre dans un bloc commençant par **`begin`** et se terminant par **`end.`**

3.3 Les conditions (expressions logiques)

3.3.1 Les opérateurs relationnels

Opération	Opérateur en Lazarus
égal à	=
différent de	<>
strictement supérieur à	>
strictement inférieur à	<
supérieur ou égal à	>=
inférieur ou égal à	<=

Deux variables de type **`string`** (chaîne de caractères) peuvent être comparées à l'aide des opérateurs relationnels.

'ALBERTA' > 'ALBERT' et 'BERNARD' > 'ALBERTA'

L'ordre de classement des chaînes est l'ordre lexicographique (celui appliqué dans les lexiques). Les majuscules précèdent les minuscules.

3.3.2 Les opérateurs logiques AND, OR, NOT

Les opérateurs logiques (NOT, AND, OR respectivement en français non, et, ou) permettent de combiner plusieurs conditions :

p.ex. (A>3) et (B<2) n'est vrai que si les deux conditions sont vérifiées à la fois ;

et

(A>3) ou (B<0) est vrai si au moins une des conditions est vraie.

Valeur de X	Valeur de Y	X AND Y	X OR Y
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
FALSE	TRUE	FALSE	TRUE
FALSE	FALSE	FALSE	FALSE

Valeur de X	NOT X
TRUE	FALSE
FALSE	TRUE

3.3.3 Règles de priorité des opérateurs

1. **NOT**, **-,** + (signe)
2. *, /, **DIV**, **MOD**, **AND**
3. +, -, **OR**
4. = , <= , => , < , > , <>

3.4 Exercices

Exercice 3-1

Quel est le résultat des opérations suivantes ?

- `6 <= 10`
- `3 = 3`
- **`NOT`** `false`
- `(3 > 1)` **`AND`** `(2 > 0)`
- `(3 > 1) OR (2 > 0)`
- `5 > 4`
- `3 <> 3`
- **`NOT`** `true`
- **`NOT`** `(3 < 2)`
- `(3 <= 1)` **`AND`** `(2 > 0)`
- `(3 <= 1)` **`OR`** `(2 > 0)`

Exercice 3-2

Ecrivez un programme qui affiche le quotient de deux nombres lus au clavier. Respectez le cas où le diviseur est égal à 0.

Exercice 3-3

Ecrivez un programme qui calcule la valeur absolue d'un nombre (positif ou négatif) sans utiliser la fonction prédéfinie `abs`.

Exercice 3-4

Ecrivez un programme qui affiche le maximum de 2 nombres.

Exercice 3-5

Ecrivez un programme qui affiche le maximum de 3 nombres.

Exercice 3-6

Écrivez un programme qui vérifie si un nombre lu au clavier est divisible par 2.

Exercice 3-7

Modifiez les valeurs de trois variables A, B, C entrées au clavier de manière à avoir $A \leq B \leq C$.

Exercice 3-8

Afficher la moyenne de 3 nombres lus au clavier. La moyenne est arrondie vers le haut. Le programme affiche un message « moyenne insuffisante », si elle est inférieure à 30, sinon il affiche « moyenne suffisante ».

Exercice 3-9

Imprimez le signe du produit de deux nombres sans faire de multiplication.

Exercice 3-10

Imprimez le signe de la somme de deux nombres sans faire d'addition.

Exercice 3-11

Vérifiez la validité d'un numéro de compte chèque postal.

Le numéro de contrôle est le reste de la division euclidienne du numéro principal par 97. Si la division est sans reste alors le numéro de contrôle est 97.

Exercice 3-12

Soit une équation de la forme $ax^2+bx+c=0$.

a) Résolvez cette équation en supposant que : $a \neq 0$.

b) Écrivez un nouveau programme qui tient compte du fait que les paramètres a, b, c peuvent être nuls.

4 Structure répétitive

4.1 Introduction

Les programmes rencontrés jusqu'à présent sont d'une utilité très limitée. En effet le gain de temps réalisé à l'aide de l'ordinateur est quasi nul (par rapport à une calculatrice par exemple). L'unique avantage provient du fait qu'un même programme peut être réutilisé pour différentes valeurs des données. Cette situation va changer radicalement dans ce chapitre. Le mécanisme de structure répétitive permet d'exécuter plusieurs fois une certaine partie d'un programme !

4.2 Exemple

Soit la partie suivante d'un programme :

```
var i,puiss:integer;
...
puiss:=1;
i:=1;
while i<=15 do
  begin
    puiss:=puiss*2;
    i:=i+1;
  end;
```

Quel est l'effet-net de ce programme ?

D'abord les variables `puiss` et `i` sont initialisées à 1. Ensuite on arrive à la nouvelle instruction

```
while <condition> do <instruction>;
```

qui comme son nom l'indique exécute instruction aussi longtemps que condition est vérifiée. L'instruction suivant le **`do`** est appelée « corps de boucle » ; elle peut être constituée d'une instruction simple ou d'un bloc **`begin ... end`**.

Dans l'exemple :

1. La condition est d'abord testée. Elle est vérifiée, car `i=1` et ainsi `i<=15`.
2. Ensuite `puiss` est multipliée par 2 et i augmentée de 1.
3. L'exécution reprend au point 1 (avec les nouvelles valeurs des variables).
4. La condition est à nouveau testée. Elle est vérifiée, car `i=2` et ainsi `i<=15`.
5. Ensuite `puiss` est à nouveau multipliée par 2 et i augmentée de 1.
6. L'exécution ne quitte la boucle que si la condition n'est plus vérifiée ! Cela arrive lorsque i atteint la valeur 16 (i est entier et augmentée de 1 à chaque passage dans la boucle).

   La boucle est donc exécutée 15 fois, car si on part de `i:=1`, il faut ajouter 15 fois le nombre 1 à i pour arriver à 16.

   En même temps `puiss` aura été multipliée 15 fois de suite par 2.

   Vu que `puiss` avait la valeur 1 avant le premier passage de boucle `puiss` aura la valeur $1*2*2*2*2*2\ldots = 2^{15} = 32768$.

Exercice 4-1

On donne la partie de programme (toutes les variables sont de type **integer**) :

```
resultat:=1;
facteur:=5;
while facteur > 1 do
  begin
    resultat:=resultat*facteur;
    facteur:=facteur-1;
  end;
writeln(resultat);
```

Simulez l'exécution de ce programme en représentant dans un tableau les valeurs successives des variables. Que va afficher le programme ?

Complétez (entête, déclarations, etc.) le programme pour qu'il soit accepté par Lazarus/FreePascal et vérifiez votre résultat !

4.3 Terminaison d'une boucle

Lors de la conception d'une boucle, il est essentiel de veiller à ce que la boucle se termine, c'est-à-dire qu'après un nombre fini de passages la condition de boucle va finir par ne plus être vérifiée.

```
i:=1;
while i<=5 do
  begin
    puiss:=puiss*2;
  end;
```

Cette boucle est mal conçue, car la seule variable i de la condition n'est pas modifiée dans le corps de boucle. À chaque passage l'évaluation de la condition donne le même résultat et le programme ne va plus sortir de la boucle.

4.4 Boucles for

4.4.1 Boucles for … to

Dans beaucoup de cas on peut simplifier l'écriture du programme en utilisant une boucle **for**.

Exemple d'une boucle for :

```
puiss:=1;
for i:=1 to 5 do puiss:=puiss*2;
```

Dans la boucle **for**, moins flexible que **while**, la condition est remplacée par un compteur (dans l'exemple précédent le compteur est la variable `i`). Ce compteur est initialisé à une certaine valeur (ici 1, par `i:=1`) et augmenté de 1 à chaque passage de boucle. La boucle va être parcourue jusqu'à ce qu'une valeur finale (ici 5) soit atteinte.

Les valeurs initiale et finale sont calculées une fois pour toute avant l'entrée dans la boucle. Elles ne sont plus réévaluées plus tard.

Dès l'entrée dans la boucle le nombre de passages est donc connu.

Le compteur et les valeurs initiale et finale seront tous de type **integer**.

Si la valeur initiale est supérieure à la valeur finale, alors le corps de boucle n'est pas exécuté du tout.

Remarque :

Une boucle **for** de la forme

```
for i:=a to b do <instruction>;
```

peut toujours être réécrite en boucle **while** de la manière suivante sans que l'effet-net du programme ne change :[3]

```
i:=a;
while i<=b do begin <instruction>; i:=i+1 end;
```

Le passage d'une boucle **while** à une boucle **for** n'est pas toujours possible sans changer la nature de l'algorithme. Cela est clair vu qu'une boucle **for** se termine toujours alors que ce n'est pas le cas de toute boucle **while**.

4.4.2 Boucles `for … downto`

Il existe une variante de la boucle **for** où le compteur est **diminué** de 1 à chaque passage :

```
for i:=a downto b do <instruction>;
```

qui a le même effet-net que

```
i:=a;
while i>=b do begin <instruction>; i:=i-1 end;
```

4.5 Exercices

Exercice 4-2

On donne la partie de programme (toutes les variables sont de type **integer**) :

```
resultat:=0;
terme:=7;
for i :=0 to 5 do
  resultat:=resultat+terme;
writeln(resultat);
```

Simulez l'exécution de ce programme en représentant dans un tableau les valeurs successives des variables. Que va afficher le programme ?

Complétez (entête, déclarations, etc...) le programme pour qu'il soit acceptable par Lazarus/ FreePascal et vérifiez votre résultat !

Exercice 4-3

Réécrivez le programme précédent à l'aide d'une boucle **while**.

Complétez ensuite le programme pour qu'il soit accepté par Lazarus/FreePascal et vérifiez votre résultat !

[3] À condition que les valeurs des variables i et b ne soient pas modifiées à l'intérieur du bloc <instruction>.

Exercice 4-4 Somme de nombres lus du clavier

Ecrivez un programme qui calcule et affiche la somme de nombres réels lus successivement au clavier.

Le programme continue jusqu'à ce que l'utilisateur entre le nombre 0. S'il entre 0 comme premier nombre, le programme affichera 0.

Allez-vous utiliser une boucle **`for`** ou une boucle **`while`** ? Expliquez votre choix !

Exercice 4-5 Somme de n nombres lus du clavier

Ecrivez un programme qui lit d'abord le nombre naturel n, représentant le nombre de termes à ajouter. Ensuite le programme va lire n nombres réels du clavier, calculer leur somme et l'afficher.

Allez-vous utiliser une boucle **`for`** ou une boucle **`while`** ? Expliquez votre réponse et comparez avec l'exercice précédent !

Exercice 4-6 (*) Table de multiplication

Ecrivez un programme qui calcule et affiche une table de multiplication de 0x0 jusqu'à 10x10. L'affichage doit se faire sous forme de tableau « bien formaté ».

4.6 Un algorithme efficace

Le premier programme calcule une puissance (de base réelle non nulle et d'exposant naturel) par un algorithme simple basé sur la définition par multiplications successives : $\begin{cases} a^0 = 1 \\ a^{n+1} = a^n \cdot a \end{cases}$

```
program puisssance;
{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

var
  expo:integer;
  base,puiss:real;
begin
  writeln('Introduisez la base (reelle non nulle): ')
  readln(base);
  writeln('Introduisez l''exposant (naturel): ')
  readln(expo);
  puiss:=1;
  while expo<>0 do
    begin
      puiss:=puiss*base;
      expo:=expo-1 ;
    end;
  writeln('la puissance vaut',puiss)
end.
```

Ce premier programme effectue toujours n multiplications pour une puissance d'exposant n.

Dans le deuxième programme, le nombre de multiplications va se réduire de façon spectaculaire. Il suffit d'ajouter une ligne supplémentaire à la définition pour traiter les exposants pairs.

$$\begin{cases} a^0 = 1 \\ a^{2n} = \left(a^2\right)^n \\ a^{2n+1} = a^{2n} \cdot a \end{cases}$$

Le programme n'est pas très différent de la puissance *simple*.

À l'intérieur de la boucle, l'exécution diffère selon la valeur de la variable `expo`.

Si `expo` est pair, on peut appliquer la formule $a^{2n} = \left(a^2\right)^n$. Cela consiste à remplacer la base par son carré et à réduire l'exposant de moitié.

Si `expo` n'est pas pair, le programme applique le même algorithme que la puissance par produits successifs.

Remarquons que dans ce cas, l'exposant est diminué de 1 et qu'il va donc devenir pair pour le prochain passage dans la boucle ! Donc l'exposant est réduit de moitié après au plus deux passages dans la boucle. Ainsi pour n'importe quel exposant inférieur à 1024 le nombre de produits à calculer ne va pas dépasser 20 ($2^{10} = 1024$), alors que 2^{1000} nécessite exactement 1000 produits avec l'algorithme simple !

Examinons le fonctionnement du programme à l'aide d'un exemple d'exécution. Pour calculer $2^{13} = 8192$, les variables `base`, `expo` et `puiss` vont prendre successivement les valeurs reprises dans le tableau suivant.

On remarquera que l'expression $I = base^{expo} \cdot puiss$ garde la même valeur après chaque passage de la boucle. Cette expression est un **invariant**. La notion d'invariant est utile pour montrer l'exactitude d'un programme.

Base	Expo	puiss	$I = base^{expo} \cdot puiss$
2	13	1	$I = 2^{13} \cdot 1 = 8192$
2	$12 = 13 - 1$	$2 = 1 \cdot 2$	$I = 2^{12} \cdot 2 = 8192$
$4 = 2 \cdot 2$	$6 = 12 : 2$	2	$I = 4^6 \cdot 2 = 8192$
$16 = 4 \cdot 4$	$3 = 6 : 2$	2	$I = 16^3 \cdot 2 = 8192$
16	$2 = 3 - 1$	$32 = 2 \cdot 16$	$I = 16^2 \cdot 32 = 8192$
256	1	32	$I = 256^1 \cdot 32 = 8192$
256	0	8192	$I = 256^0 \cdot 8192 = 8192$

Et voici le programme :

```
program puisssance_rapide;
{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };
```

```
var
  expo:integer;
  base,puiss:real;
begin
  writeln('Introduisez la base (reelle non nulle): ')
  readln(base);
  writeln('Introduisez l''exposant (naturel): ')
  readln(expo);
  puiss:=1;
  while expo<>0 do
    begin
      if (expo mod 2)=0 then
        begin
          base:=base*base;
          expo:=expo div 2
        end
      else
        begin
          puiss:=puiss*base;
          expo:=expo-1 ;
       end;
    end;
  writeln('la puissance vaut',puiss)
end.
```

4.7 Exercices

Exercice 4-7 Suite arithmétique

Écrivez un programme qui lit le premier terme et la raison d'une suite arithmétique. Ensuite le programme lira un nombre naturel **n** et il calculera par sommation et affichera le n^e terme et la somme des n premiers termes de la suite.

Dans cet exercice, contrairement à l'exercice de même nom du chapitre 2, on n'utilisera pas les formules des suites arithmétiques mais seulement la définition par récurrence.

Exercice 4-8 Suite géométrique

Même question pour une suite géométrique.

Exercice 4-9 Factorielle

La factorielle d'un nombre naturel n, notée n! est définie par :

$$\begin{cases} 0! = 1 \\ (n+1)! = n! \cdot (n+1) \end{cases}$$

Ainsi si **n** n'est pas nul on a:

n!=n·(n-1)·(n-2)·...·3·2·1

et en particulier 1!=1, 2!=2·1=2, 3!=3·2·1=6, 4!=4·3·2·1=24 etc.

a) Écrivez un programme qui lit un nombre naturel **n** et qui calculera et affichera ensuite **n!** par une boucle **for...to**.

b) Même question mais avec une boucle **for...downto**.

c) Même question mais avec une boucle **while**.

Exercice 4-10 (*) PGCD par l'algorithme d'Euclide (par soustraction)

Les formules d'Euclide pour trouver le pgcd de 2 nombres naturels non nuls sont :

$$\begin{cases} \text{pgcd}(a,a) = a \\ \text{pgcd}(a,b) = \text{pgcd}(b,a) \\ \text{pgcd}(a,b) = \text{pgcd}(a-b,b) \ si\ a > b \end{cases}$$

Écrivez un programme qui lit les 2 nombres naturels **a** et **b** et qui calcule et affiche leur pgcd en utilisant ces formules.

Exercice 4-11 (*) PGCD par l'algorithme d'Euclide (par division)

Les formules suivantes, aussi attribuées à Euclide, mènent plus vite au résultat :

$$\begin{cases} \text{pgcd}(a,0) = a \\ \text{pgcd}(a,b) = \text{pgcd}(b,a) \\ \text{pgcd}(a,b) = \text{pgcd}(a \bmod b,b) \ si\ a > b > 0 \end{cases}$$

Écrivez un programme qui lit les 2 nombres naturels **a** et **b** et qui calcule et affiche leur pgcd en utilisant ces formules.

Exercice 4-12 (*) Test de primalité

Écrivez un programme qui lit un nombre naturel non nul **n** et qui vérifie ensuite si **n** est premier. Vous pourrez améliorer le programme en n'essayant que les diviseurs potentiels inférieurs ou égaux à $\sqrt{n}$.

Exercice 4-13 (**) Factorisation première

En vous basant sur l'exercice précédent, écrivez un programme qui lit un nombre naturel non nul **n** et qui calcule et affiche ensuite la décomposition en facteurs premiers de ce nombre :

Exemple : Pour 360, le programme affichera : 360=2·2·2·3·3·5

Exercice 4-14 (***) Factorisation première (produit de puissances)

Même question qu'à l'exercice précédent, mais on demande d'afficher le résultat comme produit de puissances.

Exemple : Pour 360, le programme affichera : 360=2^3·3^2·5^1

5 Fonctions et procédures

5.1 Les fonctions

5.1.1 Introduction

Exemple

Écrivons un programme qui calcule le nombre de combinaisons de n objets p à p.

Rappelons que ce nombre se détermine par la formule :

$$\binom{n}{p} = \frac{n!}{p!(n-p)!}$$

En regardant cette formule nous constatons que pour chaque calcul d'un nombre de combinaisons nous devons effectuer trois fois le calcul d'une factorielle et ainsi répéter trois fois le même code. Les fonctions ont été développées pour éviter ces redondances dans les programmes.

Grâce à elles il suffit de noter une seule fois le code de calcul de la factorielle et de l'utiliser à chaque besoin.

Établissons d'abord le calcul de la factorielle sous forme de fonction.

```
function fact(a:integer):integer;
var i, res : integer;
begin
  i:=2;
  res:=1;
  while i<=a do
    begin
      res:=res*i;
      i:=i+1;
    end;
  result:= res;
end;
```

L'identificateur **a** est un paramètre dont la valeur va être fixée à chaque nouvel appel de la fonction.

Il est important de remarquer que la fonction **fact** retourne une valeur qui sera transmise au programme appelant.

Une fonction, une fois définie comme telle, peut être utilisée sous son nom dans le programme principal, qui va donc avoir la forme:

```
program factorielle;
{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };
```

```
var n, p, res_c   : integer;
{************************************}
function fact(a:integer) :integer;
var i, res : integer;

begin
  i:=2;
  res:=1;
  while i<=a do
    begin
      res:=res*i;
 i:=i+1;
    end;
result:= res;
end;
{************************************}
begin
  write('n = ');
  readln(n);
  write('p= ');
  readln(p);
  res_c:= fact(n) div (fact(p)*fact(n-p));
  write('Nombre de combinaisons de ',n,' objets ',p,' a ',p,' =
',res_c);
  readln;
end.
```

5.1.2 Définition

Une fonction est une partie de code d'un programme que nous pouvons utiliser plusieurs fois. Chaque fois que nous utilisons la fonction nous pouvons lui transmettre d'autres valeurs qu'elle utilisera pendant son exécution. Le code de la fonction est placé directement après les déclarations de variables (du programme principal).

Syntaxe de la définition d'une fonction

```
function nom(paramètre1:type1 ; paramètre2:type2 ; … ) : type;
var  …
begin
    <instructions>
    result := <valeur à transmettre>
end;
```

Lorsqu'on utilise plusieurs paramètres de même type alors on peut utiliser la syntaxe simplifiée :

```
function nom(paramètre1A , paramètre1B: type1 ; … ) : type;
```

Avant le mot clef **end** doit toujours figurer l'affectation qui indique le résultat à faire passer au programme appelant. Dans cette affectation, on utilise comme variable à gauche l'identificateur `result` ou le nom de la fonction.

Syntaxe de l'appel d'une fonction dans le programme appelant

```
nom(valeur1,valeur2, … ) ;
```

Les valeurs `valeur1, valeur2, ...` sont affectées aux paramètres formels `paramètre1, paramètre2,...` dans le même ordre qu'ils interviennent dans la partie déclarative de la fonction. Dans le corps de la fonction les paramètres formels se comportent comme des variables et ils peuvent être utilisés comme telles.

On remarquera que dans la définition, des paramètres de types différents sont séparés par un « ; », alors que dans l'appel tous les paramètres sont suivis d'un « , ».

Une fonction peut contenir elle-même une ou plusieurs fonctions dont elle a besoin.

Exercice 5-1

Écrivez une fonction qui retourne le maximum de 3 valeurs.

Exercice 5-2

Écrivez une fonction qui détecte si une chaîne de caractères est vide.

Exercice 5-3

Écrivez une fonction qui compte le nombre de chiffres contenus dans une chaîne composée de chiffres et de lettres.

Exercice 5-4

Écrivez un programme qui lit le numérateur et le dénominateur d'une fraction et fournit la fraction simplifiée.

5.2 Les procédures

Exemple

Écrivez un programme **moy** qui lit successivement trois valeurs positives **a**, **b** et **c** et détermine la moyenne arithmétique. La valeur de **a** doit être inférieure à 10, celle de **b** inférieure à 20 et celle de **c** inférieure à 30.

Si les valeurs ne vérifient pas ces conditions, le programme émet un message d'erreur, aussitôt la valeur saisie. Si les valeurs sont correctement saisies, le programme affiche la moyenne arithmétique.

Ici aussi nous voyons que le message d'erreur peut être plusieurs fois le même. Il n'est donc pas nécessaire de mettre plusieurs fois le code servant à l'afficher. De plus si nous voulons modifier le texte des messages, il suffit de le changer une fois.

Pour ce faire nous utilisons les procédures.

La différence entre une procédure et une fonction se trouve dans le fait qu'une fonction renvoie une valeur au programme appelant tandis qu'une procédure effectue un traitement (affiche un message à l'écran) mais ne renvoie pas de valeur.

```
program moy;
{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };
```

```
var
  a, b, c : integer;
  moy     : real;
(*************************************)
procedure erreur();
 begin
   writeln('Valeur incorrecte. Recommencez.');
 end;
(*************************************)
begin
  a:= 10;
  b:= 20;
  c:= 30;
  while a>=10 do
    begin
     write('Valeur 1 (<10): ');
     readln(a);
     if a>=10 then
      erreur();
    end;
  while b>=20 do
    begin
     write('Valeur 2 (<20): ');
     readln(b);
     if b>=20 then
      erreur();
    end;
  while c>=30 do
    begin
    write('Valeur 3 (<30): ');
    readln(c);
    if c>=30 then
      erreur();
    end;
  moy:= (a+b+c)/3;
  writeln('La moyenne de ',a,',',b,' et ',c,' vaut ',moy:5:2);
  readln;
end.
```

5.2.1 Définition

Une procédure est aussi une partie de code que nous pouvons utiliser plusieurs fois.

La différence entre une fonction et une procédure réside dans le fait qu'une procédure ne transmet pas de valeur au programme appelant. Comme effet-net une procédure peut afficher un message à l'écran ou modifier des variables du programme.

Syntaxe de la définition d'une procédure

```
procedure nom (paramètre1 : type1 ; paramètre2 : type2 ; …);
var  …
begin
     <instructions>
end;
```

Syntaxe de l'appel d'une procédure dans le programme appelant

```
nom(valeur1, valeur2 …) ;
```

Ici aussi, les valeurs *valeur1*, *valeur2*, ... sont affectées aux paramètres formels *var1*, *var2*, ... dans le même ordre qu'ils interviennent dans la partie déclarative de la procédure. Dans le corps de la procédure les paramètres formels se comportent comme des variables et peuvent être utilisés comme telles.

5.2.2 Avantages des fonctions et procédures

- **Nous évitons les redondances :** Avec ces notions nous ne répétons pas inutilement du code. Ceci rend les programmes plus petits.
- **Nous retrouvons facilement les fonctions et/ou procédures :** Si nous voulons modifier un programme nous n'avons pas besoin de passer en revue toutes les pages du code.
- **La lisibilité est améliorée :** Il est plus facile de relire du code, surtout si nous ne l'avons pas rédigé nous-mêmes.
- **Les fonctions et procédures sont portables :** Une fonction et/ou une procédure une fois écrite pour un certain programme peut être réutilisée pour un autre programme sans pour autant la réécrire. Nous pouvons ainsi construire des bibliothèques de telles fonctions ou procédures.

5.2.3 Exercices

Adaptez les programmes des chapitres précédents en les écrivant à l'aide de fonctions et de procédures chaque fois que cela est utile.

5.3 Portée

On appelle *portée d'une fonction, d'une procédure ou d'une variable* la partie du code dans laquelle elles conservent la signification définie par la déclaration, c'est-à-dire dans laquelle elles peuvent être utilisées comme elles ont été définies.

5.3.1 Portée d'une fonction/procédure

Règle 1 :

Une fonction ou une procédure n'est connue que si sa définition précède l'endroit d'où elle est appelée.

Exemple :

```
program por11;
{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };
{*****************************************}
```

```
function f: integer;
var f1 : integer;
begin
  f1 := 23;
  result := f1;
end;
procedure p;
begin
  writeln(f);
end;
{*****************************************}
begin
  p;
  readln;
end.
```

Ce programme donnera comme résultat 23.

Si nous définissons la procédure **p** avant la fonction **f**, le programme marquera comme erreur qu'il ne trouve pas la fonction **f**.

Règle 2 :

Une fonction ou procédure définie à l'intérieur d'une autre fonction ou procédure n'est connue que dans celle-ci.

Exemple :

```
program por12;
{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

var v : integer;
{*********************************************}
procedure p1(a:integer);
  {***********************************}
  function carre(p: integer): integer;
  var res: integer;
  begin
    res:=p*p;
    result:=res;
  end;
  {***********************************}

begin
  writeln('a*a = ',carre(a));
end;

{*********************************************}
```

```
begin
  v:=4;
  p1(v);
  readln;
end.
```

Ce programme donnera comme résultat 16.

Si nous remplaçons l'appel **p1(v)** par **writeln('v*v = ',carre(v));** le compilateur nous donne l'erreur *Undeclared identifier: 'carre'*, puisque la fonction **carre** est déclarée dans la procédure **p1** et ainsi elle n'est pas connue à l'extérieur de cette procédure**.**

5.3.2 Portée des variables

Nous distinguons entre deux types de variables, les *variables locales* à une fonction/procédure et *les variables globales*.

Une variable est *locale* à une fonction/procédure, si elle est déclarée dans la celle-ci.

Exemple :

```
program por21;
var a: integer;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

{p1**********************}
procedure p1;
var a_1: integer;
  {p11***************}
  procedure p11;
  var a_11 : integer;
  begin
    …
  end;
  {p11**************}
begin
  …
end;
{p1**********************}

begin
  …
end.
```

Dans cet exemple la variable **a** est globale, tandis que les variables **a_1** et **a_11** sont des variables internes respectivement à **p1** et à **p11.**

Règle 3 :

La portée d'une variable locale est strictement limitée à la fonction ou à la procédure où elle est déclarée.

La portée d'une variable est interrompue chaque fois que dans la lecture du code nous rencontrons le code d'une fonction ou d'une procédure interne dans lequel le même identificateur est utilisé.

Exemple :

```
program por22;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

{*************************************}
procedure p1;
var a : integer;
  procedure p11;
  var a : integer;
  begin
    a := 12;
    write('Valeur de a dans  p11 :');
    writeln(a);
  end;

begin
  a:= 5;
  write('Valeur de a avant p11 :');
  writeln(a);
  p11;
  write('Valeur de a apres p11 :');
  writeln(a);
end;
{*************************************}

begin
  p1;
  readln;
end.
```

Voici un tableau qui montre le déroulement du programme.

Programme principal	procedure p1	procedure p11	a
program Project1; {$APPTYPE CONSOLE}			
begin			
p1;	procedure p1;		
	var a : integer;		
	a:= 5;		5
	write('Valeur de a avant p11 :');		5
	writeln(a);		5
	p11;	procedure p11;	5
		var a : integer;	
		begin	
		a := 12;	12
		write('Valeur de a dans p11 :');	12
		writeln(a);	12
		end;	5
	write('Valeur de a apres p11 :');		5
	writeln(a);		5
	end;		
readln;			
end.			

5.3.3 Exercices

Exercice 5-5

Soit le code suivant :

```
program expor;
{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };
```

```
var a, b, c : integer;
procedure p1();
var a, b:integer;
begin
a:= 2;
b:= 4;
writeln('a= ',a,',  b= ',b,',  c= ',c);
end;
{************************************}
begin
a:= 10;
b:= 15;
c:= 30;
writeln('a= ',a,',  b= ',b,',  c= ',c);
p1();
writeln('a= ',a,',  b= ',b,',  c= ',c);
readln;
end.
```

Déterminez les valeurs qui seront sorties par le programme.

5.4 Passage des variables dans une fonction ou une procédure

Exercice : Écrivez une procédure qui échange deux variables.

Écrivons notre programme de la façon suivante:

```
program pass1;
{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

var a,b : integer;
procedure ech(v1, v2: integer);
var temp: integer;
begin
  temp:= v1;
  v1:= v2;
  v2:= temp
end;

begin
  write('a= '); readln(a);
  write('b= '); readln(b);
  ech(a,b);
  write('a= ',a,'   b= ',b)
  readln;
end.
```

Ce programme ne réalisera pas l'effet-net escompté, mais laissera inchangées les valeurs des variables.

Ceci est dû au fait que le passage des paramètres se fait, sauf indication spéciale, *par valeurs.*

Passons en revue les valeurs des différentes variables lors du déroulement du programme:

Supposons qu'on donne à ***a*** la valeur 5 et à ***b*** la valeur 8.

`program pass1;`					
`var a,b : integer;`	création de a	création de b			
`begin`					
`write('a= '); readln(a);`	5				
`write('b= '); readln(b);`		8			
`procedure ech( v1 , v2: integer);`			création de v1 avec la valeur 5	création de v2 avec la valeur 8	
`var temp: integer`					création de temp
`temp:= v1;`					5
`v1:= v2;`			8		
`v2:=temp`				5	
`end; {ech)`			destruction de v1	destruction de v2	destruction de temp
`write('a= ',a,' b= ',b)`	sortie de la valeur de a: 5	sortie de la valeur de b: 8			
`end.`	destruction de a	destruction de b			

Au moment du passage des valeurs une copie est faite, sur laquelle agit la procédure **ech**. En sortant de la procédure **ech** toute référence à cette copie est détruite et nous nous retrouvons avec les anciennes valeurs de **a** et de **b**.

Une autre méthode de passage des valeurs est celle qu'on appelle normalement par *référence* ou par *variable*.

Avec cette méthode de passage des valeurs une référence est passée au programme appelé qui permet d'accéder directement à la variable du programme appelant.

Règle :

Pour arriver en Lazarus/FreePascal à un passage des valeurs par *variable* nous devons faire précéder les différents paramètres formels par le préfixe VAR.

Dans notre exemple la ligne de définition de la procédure **ech** aura donc la forme :

```
procedure ech(var V1, V2: integer);
```

Si nous voulons donc qu'une variable définie dans le programme appelant change de valeur par une action dans le programme appelé, nous devons passer la valeur par référence.

5.4.1 Exercices

Exercice 5-6

Utilisez la procédure `ech` pour classer 4 valeurs lues au clavier lors du déroulement du programme.

Exercice 5-7

Soit le code suivant :

```
program pass;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

var a,b : integer;

procedure p1(a,b :integer);
begin
  a:=a+2;
  b:=b*b;
  writeln('Dans la procedure p1');
  writeln('a=',a,' b=',b);
end;

procedure p2(a: integer; var b :integer);
begin
  a:=a+2;
  b:=b*b;
  writeln('Dans la procedure p2');
  writeln('a=',a,' b=',b);
end;
```

```
begin
  a:=5;
  b:=7;
  writeln('Avant la procedure p1');
  writeln('a=',a,' b=',b);
  p1(a,b);
  writeln('Apres la procedure p1');
  writeln('a=',a,' b=',b);
  p2(a,b);
  writeln('Apres la procedure p2');
  writeln('a=',a,' b=',b);
  readln ;
end.
```

Quel est l'effet-net de ce programme ?

6 Les structures de données composées

Jusqu'à présent nous n'avons utilisé que des données de type simple : *entier*, *réel*, *booléen* et *chaînes de caractères*. Une variable de type simple ne peut représenter qu'une seule donnée.

Dans ce chapitre nous allons découvrir les données de type structuré. Ces données ont la propriété que chaque identificateur individuel peut représenter des données multiples. On peut traiter l'ensemble de ces données multiples aussi bien que les données individuelles. Les données de type structuré se manipulent donc collectivement ou individuellement.

6.1 Les tableaux

6.1.1 Généralités

Un tableau est une structure de données qui regroupe des données de type identique. On peut définir des tableaux d'entiers, de réels, de booléens, de chaînes de caractères ou bien de tout autre type identique. L'accès à chaque élément d'un tableau se fait moyennant un indice.

6.1.2 Déclaration

Comme pour les variables de type simple, les tableaux doivent être déclarés. Cette déclaration doit comporter un nom et le type de données contenues dans le tableau. En plus on doit indiquer le nombre d'éléments. On obtient la syntaxe suivante :

```
nom: array [A..B] of type
```

avec :

`nom` :	nom du tableau
`A` :	début de l'indice
`B` :	fin de l'indice
`type` :	type de donnée des éléments du tableau

Exemples :

- Tableau de 100 entiers :

```
Entiers : array [1..100] of integer;
```

- Tableau de 50 noms :

```
Noms : array [51..100] of string;
```

- Tableau de 30 réels :

```
Reels: array [30..59] of real;
```

- Tableau de 15 booléens :

```
Booleens: array [31..45] of boolean;
```

6.1.3 Accès aux éléments

Après avoir déclaré un tableau, nous pouvons utiliser les différents éléments du tableau comme des variables : Nous pouvons faire des accès en écriture (affectation) ou en lecture.

L'accès aux différents éléments du tableau est effectué moyennant un indice. Cet indice est noté entre crochets après le nom du tableau.

Exemples :

Les exemples suivants utilisent les déclarations ci-dessus.

- Écriture de la valeur 5 dans l'élément 3 du tableau Entiers :

```
Entiers[3]:=5;
```

- Affichage de la valeur de l'élément 3 du tableau Entiers :

```
writeln(Entiers[3]);
```

- Affichage de toutes les valeurs du tableau Noms :

```
for I:=51 to 100 do
  writeln(Noms[I]);
```

6.1.4 Remarques

Comme pour les variables simples, les éléments d'un tableau doivent être initialisés avant tout accès en lecture.

On doit respecter les limites de l'indice : Tout accès à un élément portant un indice non défini lors de la déclaration est interdit.

6.1.5 Exemple

Écrivez un programme qui demande à l'utilisateur 10 entiers, les stocke dans un tableau et les affiche ensuite.

```
program Exemple;
{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

var
  I:integer;
  Tableau:array[1..10] of integer;
begin
  for I:=1 to 10 do
      readln(Tableau[I]);
  for I:=1 to 10 do
    writeln('La ',I,'ième valeur vaut : ',Tableau[I]);
end.
```

6.1.6 Exercices

Exercice 6-1

Écrivez un programme qui demande à l'utilisateur 5 réels, les stocke dans un tableau et affiche à la fin le contenu du tableau sur l'écran.

Exercice 6-2

Remplissez un tableau avec 20 entiers aléatoires[4]. Affichez-le puis recherchez le minimum et le maximum du tableau et affichez-les.

Exercice 6-3

Remplissez un tableau avec 10 entiers aléatoires. Calculez la somme, la moyenne et l'écart-type des valeurs du tableau. Affichez le contenu du tableau ainsi que la somme, la moyenne et l'écart-type.

Exercice 6-4

Remplissez un tableau avec 100 entiers aléatoires compris entre 10 et 25. Comptez le nombre d'occurrences (fréquence) d'un entier entré par l'utilisateur.

Exercice 6-5

Rédigez un programme qui remplit un tableau avec 10 entiers aléatoires.

Affichez-le.

Inversez ensuite l'ordre des nombres dans le tableau et affichez ensuite le tableau réarrangé.

Exercice 6-6

On vous donne le numéro d'un jour d'une année. En supposant qu'il ne s'agit pas d'une année bissextile, déterminez le jour et le mois en question.

Exercice 6-7

Demandez à l'utilisateur les coefficients d'un polynôme de degré 5.

Calculez les coefficients de la dérivée de ce polynôme.

Demandez à l'utilisateur une valeur réelle pour évaluer le polynôme ainsi que sa dérivée en utilisant le schéma de Horner.

Exercice 6-8

Demandez à l'utilisateur les coefficients de deux polynômes de degré 4.

Calculez les coefficients de la somme, de la différence et du produit de ces deux polynômes et affichez-les.

Exercice 6-9 (**) Conversion décimale → binaire

Écrivez un programme qui lit un nombre naturel (en notation décimale) non nul *n.* Le programme transformera et affichera ensuite ce nombre en notation binaire.

Exemple : Pour 43, on affichera : 101011 car $43 = 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$

Exercice 6-10 (**) Conversion binaire → décimale

Écrivez un programme qui lit un nombre naturel (en notation binaire) non nul *n.* Le programme transformera et affichera ensuite ce nombre en notation décimale.

---

[4] Pour obtenir un nombre aléatoire, il faut utiliser la fonction **`random`** (consulter l'aide en ligne (F1) pour plus de détails).

Exemple : Pour 101011, on affichera : 43.

Le programme n'a pas besoin de vérifier si le nombre donné est effectivement en écriture binaire (seulement chiffres 0 et 1).

Exercice 6-11 (***) Simulation d'un tirage Lotto

Un joueur peut choisir 6 nombres entiers entre 1 et 49. Faites la saisie des nombres proposés par le joueur et mémorisez-les à l'aide d'un tableau de booléens. Veillez à ce les 6 nombres soient distincts deux à deux.

Effectuez ensuite le tirage. Utilisez à nouveau un tableau de booléens pour mémoriser le tirage. Veillez à ce qu'il y ait exactement 6 nombres différents.

Déterminez ensuite combien de nombres le joueur a deviné correctement et affichez le tirage ainsi que les nombres correctement devinés.

6.2 Les enregistrements

6.2.1 Généralités

Un enregistrement est une structure de données qui permet de regrouper des données n'ayant pas besoin d'être de même type. Les données qui composent un enregistrement sont appelés : *champs*. Un champ peut avoir n'importe quel type : integer, real, boolean, string, array ou même un autre enregistrement.

6.2.2 Déclaration

Comme pour les variables de type simple, les enregistrements doivent être déclarés. Cette déclaration doit comporter un nom et doit énumérer les différents champs contenus dans l'enregistrement. Pour chaque champ il faut spécifier son nom et son type. On obtient la syntaxe suivante :

```
nom:record
      champ1:type1;
      champ2:type2;
      champ3:type3;
      …
      champN:typeN;
    end;
```

avec :

`nom :`	nom de l'enregistrement,
`champ1 … champN :`	noms des différents champs de l'enregistrement,
`type1 … typeN :`	type de donnée du champ concerné.

Exemples :

- Enregistrement décrivant une date :

```
date:record
       jour:integer;
       mois:string;
       annee:integer;
     end;
```

- Enregistrement décrivant un vecteur du plan (ici 2 vecteurs sont déclarés) :

```
vecteur1,vecteur2:record
                  x:real;
                  y:real;
                  end;
```

- Enregistrement décrivant un élève :

```
eleve:record
        nom:string;
        prenom:string;
        classe:string;
        naissance:record
                    annee:integer;
                    mois:string;
                    jour:integer;
                   end;
      end;
```

- Enregistrement décrivant une voiture :

```
voiture:record
          marque:string;
          modele:string;
          cylindree:integer;
          places:integer;
          essence:boolean;
          options:string;
        end;
```

6.2.3 Accès aux éléments

Après avoir déclaré un enregistrement, nous pouvons utiliser les différents champs de l'enregistrement comme des variables, nous pouvons faire des accès en écriture (affectation) ou en lecture.

L'accès aux différents champs de l'enregistrement est effectué moyennant la notation suivante : **`enregistrement.champ`**

Exemples :

Les exemples suivants utilisent les déclarations ci-dessus.

- On inscrit la date du 3 juin 2003 dans l'enregistrement `date` :

```
date.jour:=3;
date.mois:='juin';
date.annee:=2003;
```

- Le vecteur2 est égal à 3 fois le vecteur1 :

```
vecteur2.x:=3*vecteur1.x;
vecteur2.y:=3*vecteur1.y;
```

- Affichage de la norme du vecteur2 :

```
writeln('Norme : ',sqrt(sqr(vecteur2.x)+sqr(vecteur2.y));
```

- L'élève Toto Tartempion de la classe 2B1 est né le 3 mai 1985.

```
eleve.nom:='Tartempion';
eleve.prenom:='Toto';
eleve.classe:='2B1';
eleve.naissance.jour:=3;
eleve.naissance.mois:='mai';
eleve.naissance.annee:=1985;
```

Remarque

Comme pour les variables simples, les champs d'un enregistrement doivent être initialisés avant tout accès en lecture.

6.2.4 Exemple

Écrivez un programme qui demande à l'utilisateur d'entrer 2 vecteurs du plan et qui calcule et affiche la somme et le produit scalaire de ces deux vecteurs.

```
program Exemple;
var
  scalaire:integer;
  vecteur1,vecteur2,vecteur3:record
                               x:integer;
                               y:integer;
                             end;
begin
  writeln('Entrez les composants du premier vecteur :')
  readln(vecteur1.x,vecteur1.y);
  writeln('Entrez les composants du deuxième vecteur :')
  readln(vecteur2.x,vecteur2.y);
  {Calcul de la somme}
  vecteur3.x:=vecteur1.x+vecteur2.x;
  vecteur3.y:=vecteur1.y+vecteur2.y;
{ Calcul du produit scalaire }
  scalaire:=vecteur1.x*vecteur2.x+vecteur1.y*vecteur2.y;
  writeln('La somme vaut :(',vecteur3.x,';',vecteur3.y,')';
  writeln('Le produit scalaire vaut :',scalaire);
end.
```

6.2.5 Exercices

Exercice 6-12

Donnez la déclaration d'un enregistrement qui permet de mémoriser les données relatives à un article d'un supermarché.

Exercice 6-13

Donnez la déclaration d'un enregistrement qui permet de mémoriser les données relatives à un livre d'une bibliothèque.

Exercice 6-14 (**) Tableau des valeurs

Écrivez un programme qui :

- demande à l'utilisateur les coefficients d'un polynôme de degré 4 et les mémorise dans un tableau ;

- remplit ensuite un tableau de valeurs, représenté par un tableau dont les éléments sont des enregistrements, de première composante X et de deuxième composante Y, X variant de -5 à 5 ;
- affiche le tableau des valeurs.

Deuxième partie :

Cours de I^{re}

Applications Lazarus

7 Lazarus

7.1 Introduction

Après son lancement, Lazarus se présente sous la forme de 4 fenêtres.

La première fenêtre occupe la partie supérieure de l'écran. Elle correspond à l'environnement de programmation proprement dit.

Cette fenêtre contient :

- la barre de titre ;
- la barre de menu de Lazarus ;
- une zone « barre d'outils » (sur la gauche) ;
- une zone contenant les divers composants regroupés par familles.

La seconde fenêtre se trouve par défaut à gauche de l'écran : c'est *l'inspecteur d'objets*. Il permet de visualiser, pour chaque objet ou composant, les propriétés et les événements auxquels l'objet peut répondre.

La troisième fenêtre constitue la fiche principale de la future application Lazarus. Il s'agit, au départ, d'une fenêtre vide dans laquelle on placera les divers objets.

La dernière fenêtre, cachée sous la précédente constitue l'éditeur proprement dit, contenant le code source de l'application.

Pour démarrer une nouvelle application, il faut choisir l'option **New Application** du menu **File**.

Pour sauvegarder une application, il faut choisir l'option **Save All** du menu **File**. Une règle à suivre absolument est de créer un répertoire par application. Comme Lazarus crée plusieurs fichiers pour une application donnée, il est plus facile de les retrouver s'ils ne sont pas enregistrés avec d'autres fichiers de noms pratiquement identiques.

Lors du premier « tout enregistrement » de l'application, une fenêtre permet de choisir l'emplacement de sauvegarde et même de le créer.

Pour exécuter une application, il faut choisir l'option **Run** du menu **Run**. Si les options d'auto-enregistrement ont été sélectionnées et que l'application n'a encore jamais été sauvegardée, la fenêtre d'enregistrement s'affiche. L'application est ensuite compilée puis exécutée, si elle ne contient pas d'erreur.

7.2 Les fichiers utilisés en Lazarus

Les fichiers d'un projet :

.LPR	**fichier source projet**	***Lazarus Program File***
.LFM	**fichier fiche**	***Lazarus Form File***
.PAS	**fichier unité - code source**	
.LPI	fichier information projet	*Lazarus Project Information File*
.EXE	fichier exécutable (le programme développé)	
.PPU, .O, .A	fichier unité - code compilé	*Compiled Unit*
.LRS	fichier ressource (icônes, bitmaps, curseurs, . . .)	
.LPK	fichier paquet	*Lazarus Package Information File*

Les fichiers .LPR, .LFM et .PAS sont les fichiers nécessaires à la programmation et doivent être copiés pour continuer le développement sur une autre machine.

7.3 L'approche Orientée-Objet

Dans la programmation en Lazarus, nous allons manipuler des objets. Ces objets sont définis par leurs propriétés, leurs méthodes et leurs événements.

Dans la vie courante, un objet peut être toute chose vivante ou non (par exemple : une voiture, une montre, …). En informatique, un objet est souvent un bouton, une fenêtre, un menu, …

7.3.1 Les propriétés

Cependant, chaque personne « voit » l'objet différemment. Par exemple chacun aura une perception différente de l'objet voiture, selon l'importance qu'il attribue aux caractéristiques de l'objet.

Une propriété est une information décrivant une caractéristique de l'objet.

Ainsi, il est facile d'énumérer quelques propriétés pour l'objet voiture : vitesse maximale, cylindrée, marque, modèle, couleur, …

Nous pouvons consulter les propriétés et également les modifier.

Par exemple, nous pouvons définir les propriétés d'une voiture dans un jeu de course, tel que la couleur. Ceci se fait de la manière suivante :

Voiture1.Couleur := Rouge

Bien entendu, il faut que la constante « Rouge » soit définie.

Les objets (dans Lazarus ces objets sont appelés *composants*) que nous allons utiliser sont prédéfinis (boutons, fenêtres, menus, …). Pour afficher les propriétés d'un objet, il suffit de cliquer dessus. Les propriétés s'affichent alors dans l'inspecteur d'objet.

Il existe des composants en lecture seule.

7.3.2 Les méthodes

Pour simplifier, on peut se représenter une méthode comme un ordre du style « fais ceci ». Cet ordre provoque l'exécution d'une certaine action par l'objet.

Par exemple, pour l'objet voiture, on peut énumérer les méthodes suivantes : accélérer, freiner, changer de vitesse, …

Donc, l'instruction

Voiture1.Accélérer(10)

indique à la voiture qu'elle doit accélérer d'un facteur 10.

Les propriétés ne font que changer une caractéristique d'un objet alors que les méthodes effectuent une action. On n'utilise pas de signe d'affectation lorsqu'on exécute une méthode.

7.3.3 Les événements

Pour chaque objet, il peut survenir certains événements, qui déclenchent des réactions.

Dans l'exemple de la voiture, lorsqu'on tourne la clé dans le contact ou lorsqu'on appuie sur l'accélérateur, la voiture respectivement démarre ou accélère.

Pour les objets informatiques il leur arrive des événements auxquels ils peuvent réagir.

Par exemple, un bouton peut avoir les événements *OnMouse*… (événements liés à la souris), *OnKey*… (événements liés au clavier), *OnEnter* (réception du focus), *On Exit* (perte du focus), …

Les événements existants pour un objet sont visibles dans l'inspecteur d'objet.

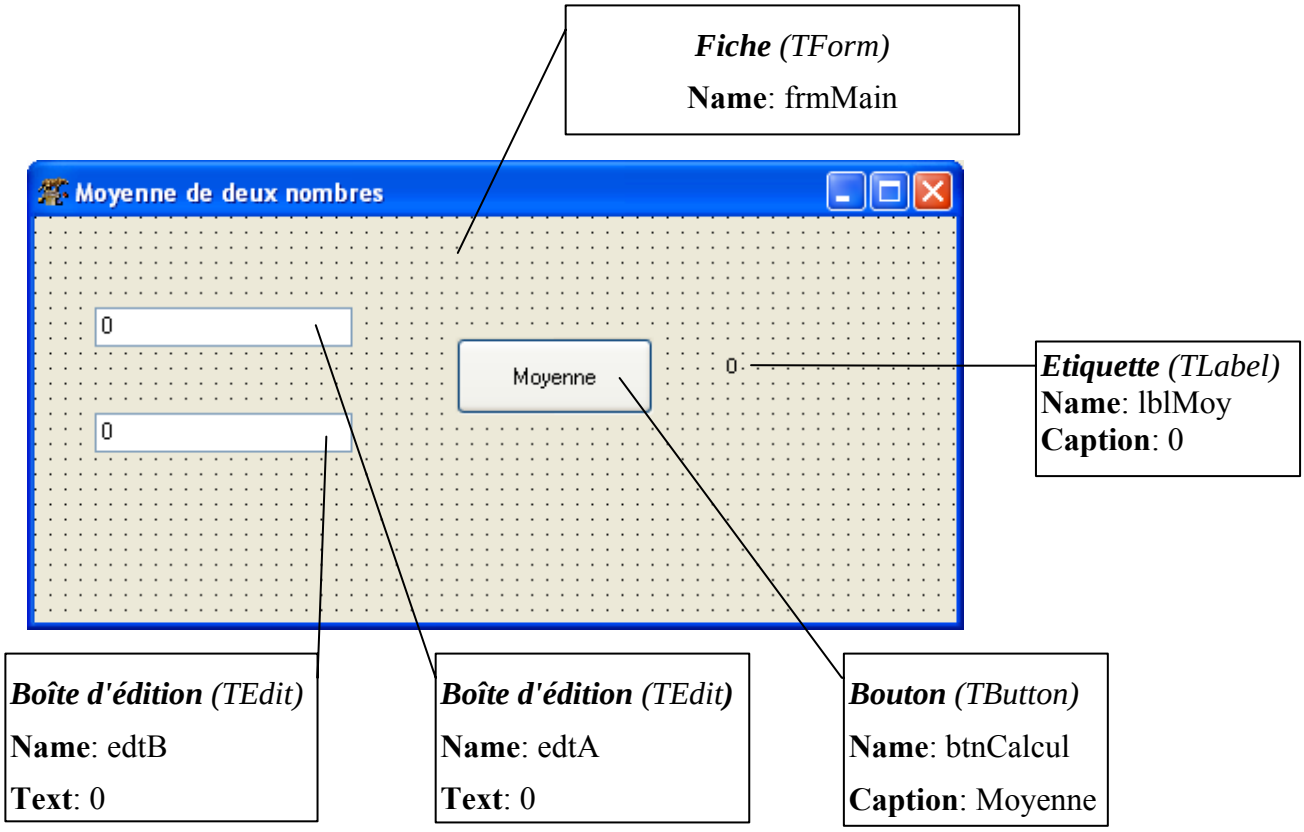
7.4 Passage FreePascal – Lazarus – un premier exemple

7.4.1 L'interface

En Lazarus, nous utiliserons les composants de l'interface graphique (les fenêtres de Windows) pour entrer les données et afficher les résultats. Les algorithmes PASCAL que nous avons utilisés jusqu'à maintenant pour obtenir les résultats pourront rester inchangés.

D'abord, nous allons créer l'interface du programme. Nous allons adapter les noms internes (propriété **Name**) de chaque composant que nous utilisons.

En plus, nous allons modifier les inscriptions sur les différents composants (propriétés **Caption** ou **Text**).



7.4.2 Les conversions de types

Les données inscrites dans les boîtes d'édition sont de type texte (string). Nous devons donc les transformer afin de pouvoir effectuer des calculs.

Voici quelques fonctions permettant d'effectuer certaines conversions :

`StrToInt(string)` : convertit une chaîne de caractères en un nombre entier (type integer)

`StrToFloat(string)` : convertit une chaîne de caractères en un nombre réel (type real).

De même, pour pouvoir afficher le résultat, nous devons le transformer en texte. Ceci peut se faire grâce aux fonctions `FloatToStr` et `IntToStr`.

7.4.3 Le traitement

Après la saisie des données dans les boîtes d'édition, l'utilisateur va cliquer sur le bouton `btnCalcul`. À cet instant l'événement `OnClick` du bouton est généré et la méthode `btnCalculClick` est lancée. Nous allons donc entrer les instructions à effectuer dans la méthode `btnCalculClick` :

```
procedure TfrmMain.btnCalculClick(Sender: TObject);
var A,B,MOY : real;
begin
  A := StrToFloat(edtA.Text);
  B := StrToFloat(edtB.Text);
  MOY := (A+B)/2;
  lblMoy.Caption := FloaToStr(MOY);
end;
```

7.4.4 Exercices

Exercice 7-1

Ecrivez un programme qui affiche le plus grand de trois nombres réels A, B, C.

Exercice 7-2

Ecrivez un programme qui calcule la somme d'une série de nombres entrés au clavier, en utilisant deux boîtes d'édition et un bouton pour la remise à zéro de la somme.

Exercice 7-3

Réalisez le programme PUISSANCE qui calcule et affiche la puissance $\mathbf{x^n}$ (puissance x exposant n pour un réel x et un entier n positif, négatif ou zéro).

Pour les cas où $\mathbf{x^n}$ ne se laisse pas calculer, affichez un message d'erreur !

Exercice 7-4

a) Réalisez un programme qui permet de simplifier une fraction.

b) Utilisez une partie du programme réalisé sous a) pour faire un programme qui additionne deux fractions.

7.5 Calcul de la factorielle

Comme premier programme essayons d'implémenter en Lazarus le calcul de la factorielle.

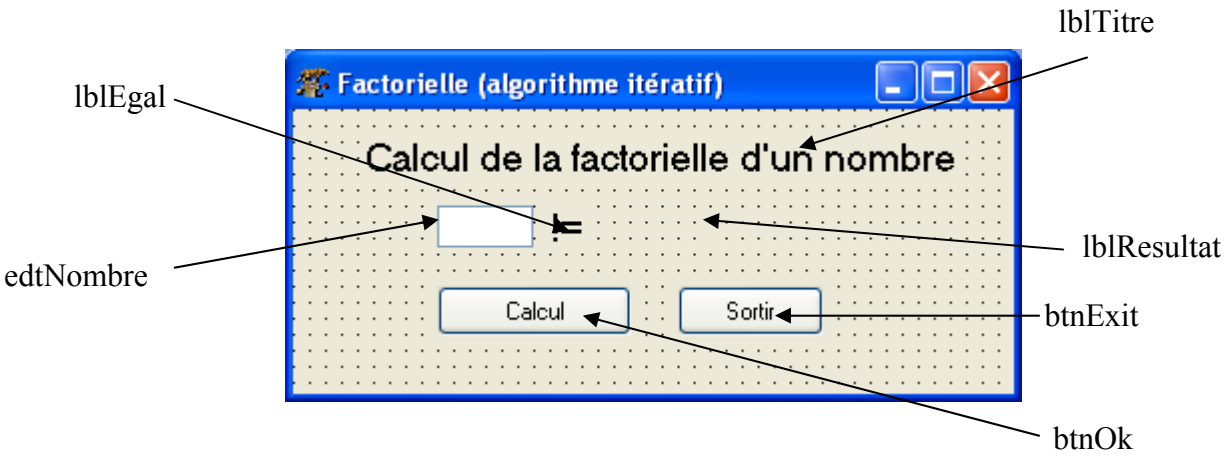
Rappelons que pour tout entier naturel x, $x! = \begin{cases} 1 \cdot \ldots \cdot x & \text{si } x \geq 1 \\ 1 & \text{si } x = 0 \end{cases}$

Pour élaborer ce calcul nous pouvons utiliser le programme développé dans le cours de 2e et l'incorporer dans celui en Lazarus.

7.5.1 Présentation visuelle

Commençons par établir un formulaire dans lequel nous notons les valeurs et éditons les résultats.

Voici un exemple d'un tel formulaire.



Ce formulaire est un nouvel objet que nous appellerons *Tformulaire*, de capture (Caption) : **Factorielle (algorithme itératif)**. Il est composé des propriétés suivantes :

Name	**type**	**text**	**Caption**
lblTitre	TLabel		Calcul de la factorielle d'un nombre
lblEgal	TLabel		! =
edtNombre	TEdit	*valeur de la factorielle à calculer*	
lblResultat	TLabel		
btnOk	TButton		Calcul
btnExit	TButton		Sortir

Il est évident que tous ces champs possèdent encore davantage de propriétés. Nous n'avons énuméré ici que les plus importantes.

7.5.2 Code

Une fois ce formulaire établi, nous pouvons écrire le code nécessaire pour calculer la factorielle. Rappelons que nous y utiliserons le code *Pascal* établi en 2e (voir également les « Algorithmes obligatoires »).

Lazarus s'occupera de la déclaration du formulaire et de ses propriétés.

La seule partie du code que nous devons écrire est celle de la procédure `btnOkClick` qui va être exécutée, comme son nom le dit, après que l'utilisateur ait poussé sur le bouton `Ok`. Nous dirons que la procédure s'exécute après l'événement `onClick` appliqué à la propriété `btnOk`.

Le tout se trouvera dans l'unité `Unit1`.

Voici une possibilité de code pour la procédure en question.

```
procedure Tformulaire.btnOkClick(Sender: TObject);
var n,fact:integer;
begin
  n:=StrToInt(edtNombre.Text);
  fact:=factorielle(n);
  lblResultat.Caption:=IntToStr(fact)
end;
```

Bien entendu, cette procédure suppose que la fonction `factorielle(n:integer):integer` est définie.

7.5.3 Explication du programme.

En regardant de près ce code quelques remarques s'imposent :

- Comme la procédure s'emploie dans le formulaire `Tformulaire`, elle s'appellera sous son nom complet : `Tformulaire.btnOkClick`.
- La valeur saisie du nombre est la valeur de la propriété `Text` du champ `edtNombre`. Nous notons donc cette valeur par `edtNombre.Text`. De plus, comme il s'agit d'une chaîne de caractères, nous devons encore transformer cette chaîne en une valeur numérique par la fonction `StrToInt`, fonction prédéfinie dans Lazarus.
- La valeur de la factorielle calculée sera affectée à la propriété *Caption* du champ `lblResultat` que nous noterons `lblResultat.Caption`. Comme de plus cette valeur doit être du type chaîne de caractères, nous devons transformer `fact` par la fonction `IntToStr`, autre fonction prédéfinie dans Lazarus.

L'unité `Unit1` se présentera finalement ainsi :

```
unit Unit1;
interface
uses
Classes, SysUtils, LResources, Forms, Controls, Graphics, Dialogs,
StdCtrls;

type
  { TFormulaire }
```

```
  TFormulaire = class(TForm)
    edtNombre: TEdit;
    btnOk: TButton;
    lblTitre: TLabel;
    lblEgal: TLabel;
    lblResultat: TLabel;
    btnExit: TButton;
    procedure btnOkClick(Sender: TObject);
    procedure btnExitClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Formulaire: TFormulaire;
implementation

//fonction permettant de calculer une factorielle
function factorielle(n:integer):integer;
var fact,i:integer;
begin
  fact:=1;
  for i:=2 to n do
    fact:=fact*i;
  factorielle:=fact
end;

procedure TFormulaire.btnOkClick(Sender: TObject);
var n,fact:integer;
begin
  n:=StrToInt(edtNombre.Text);
  fact:=factorielle(n);
  lblResultat.Caption:=IntToStr(fact)
end;

procedure TFormulaire.btnExitClick(Sender: TObject);
begin
  Application.Terminate;
end;

initialization
  {$I unit1.lrs}
end.
```

7.5.4 Exécution du programme

Une fois ce code saisi, nous sauvegardons le tout dans un répertoire réservé à cette application. Nous cliquons ensuite sur le bouton qui représente un petit triangle vert et le programme s'exécutera.

7.5.5 Remarque

- La méthode `Tformulaire.btnExitClick` aura comme seule commande `Application.Terminate`. De cette manière l'événement `Click` lié au bouton `btnExit` aura comme effet net d'arrêter l'application.
- La saisie fautive respectivement d'un nombre négatif ou décimal ne conduira pas à un message d'erreur de la part du programme mais nous affichera un résultat erroné. Nous laissons au lecteur le soin de corriger le programme pour l'améliorer de ce point de vue.

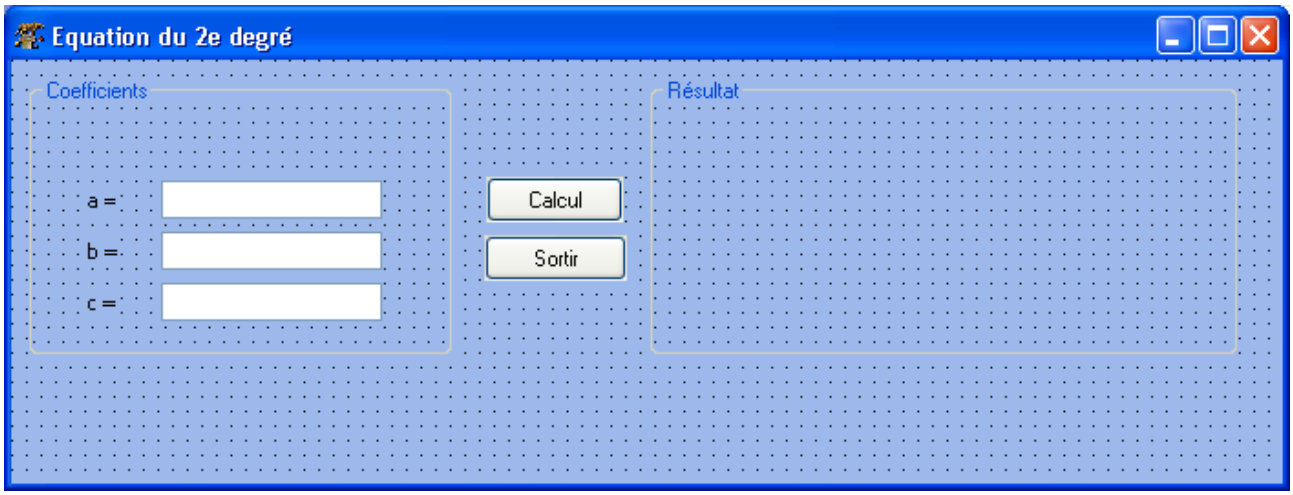
7.6 Equation du second degré

Écrivons maintenant un programme qui demande à la saisie les trois coefficients *a*, *b* et *c* d'une équation du second degré et qui calcule, si elles existent, les racines de l'équation $ax^2 + bx + c = 0$.

Ce même programme a été demandé comme exercice dans le cours de 2e. Nous en faisons ici un programme Lazarus.

7.6.1 Présentation visuelle

Comme dans l'exemple précédent nous commençons par dessiner un formulaire que nous appellerons `Tformulaire`, dont l'instance `formulaire` nous permet de saisir les coefficients et de lire le(s) résultat(s). La propriété `Caption` de l'objet `Tformulaire` aura comme valeur : **équation du 2e degré**.



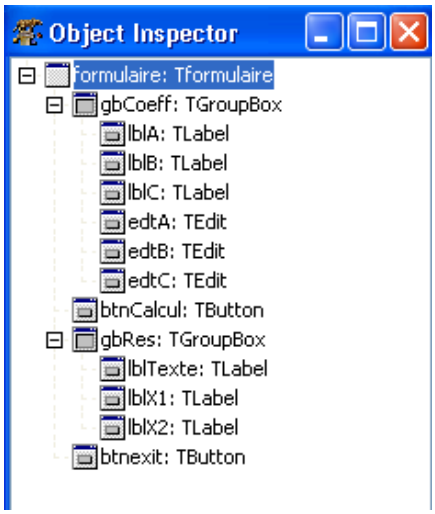
Ce formulaire est composé des champs suivants :

name	type	text	caption
lblA	TLabel		a =
lblB	TLabel		b =
lblC	TLabel		c =
lblTexte	TLabel		*texte sur le résultat*
lblX1	TLabel		*valeur de la racine*
lblX2	TLabel		*valeur de la racine*

edtA	TEdit	*valeur de a*	
edtB	TEdit	*valeur de b*	
edtC	TEdit	*valeur de c*	
btnCalcul	TButton		Calcul
gbCoeff	TGroupBox		Coefficients
gbRes	TGroupBox		Résultat
btnExit	TButton		Sortir

Nous remarquons tout-de-suite une nouvelle notion :

– les champs du type `TGroupBox` : Ils servent à regrouper différents champs dans un même groupe qu'ils affichent avec un cadre et un nom donné sous *Caption*. Dans notre exemple la TGroupBox `gbCoeff` regroupe les champs `lblA`, `lblB` et `lblC`, tandis la TGroupBox `gbRes` affichera les résultats et contient ainsi les champs `lblTexte`, `lblX1` et `lblX2`. Nous définissons ces `TGroupBox` comme suit :



– Nous venons déjà de remarquer que le résultat sera affiché dans la TGroupBox `gbRes`. Mais les étiquettes (labels) `lblTexte`, `lblX1` et `lblX2` sont invisibles pour l'instant. Si nous avons un résultat à afficher, `lblTexte` contiendra une des phrases suivantes :
Il n'y a pas de solution réelle ! ,

Il existe une solution réelle ! ou

Il y a deux solutions réelles différentes ! (ceci en fonction du résultat du calcul), tandis que `lblX1` et `lblX2` contiendront les valeurs des racines éventuelles. Comme actuellement ces étiquettes ne contiennent pas de texte (*caption vide*), elles n'apparaîtront pas à l'écran.

7.6.2 Code

Une fois ce formulaire établi nous écrivons le code suivant :

```
unit Unit1;
interface
uses
Classes, SysUtils, LResources, Forms, Controls, Graphics, Dialogs,
StdCtrls;
type
  { Tformulaire }
  Tformulaire = class(TForm)
    gbCoeff: TGroupBox;
    lblA: TLabel;
    lblB: TLabel;
    lblC: TLabel;
    edtA: TEdit;
    edtB: TEdit;
    edtC: TEdit;
    btnCalcul: TButton;
    gbRes: TGroupBox;
    lblTexte: TLabel;
    lblX1: TLabel;
    lblX2: TLabel;
    btnExit: TButton;
    procedure btnCalculClick(Sender: TObject);
    procedure btnExitClick(Sender: TObject);
  private
    { Private-Deklarations}
  public
    { Public-Deklarations }
  end;
var
  formulaire: Tformulaire;
implementation

procedure Tformulaire.btnCalculClick(Sender: TObject);
var a,b,c,disc : real;
begin
  a:=StrtoFloat(edta.text);
  b:=StrtoFloat(edtb.text);
  c:=StrtoFloat(edtc.text);
  disc:=b*b-4*a*c;
  if disc < 0 then
  begin
    lbltexte.caption:='Il n''y a pas de solution réelle !';
    lblx1.caption:='';
    lblx2.caption:='';
  end;
  if disc = 0 then
  begin
    lbltexte.caption:='Il existe une solution réelle !';
    lblx1.caption:='x = ' + FloattoStr(-b/(2*a));
    lblx2.caption:='';
  end;
  if disc > 0 then
```

```
  begin
    lbltexte.caption:='Il y a deux solutions réelles différentes
!';
    lblx1.caption:='x1 = ' + FloattoStr((-b-sqrt(disc))/(2*a));
    lblx2.caption:='x2 = ' + FloattoStr((-b+sqrt(disc))/(2*a));
  end;
end;

procedure Tformulaire.btnExitClick(Sender: TObject);
begin
  Application.Terminate;
end;

initialization
  {$I unit1.lrs}
end.
```

7.6.3 Explication du code

Nous lisons d'abord les 3 coefficients *a*, *b* et *c* de l'équation. Comme nous les avons définis comme variables réelles, nous devons utiliser la fonction `StrtoFloat` pour faire la transformation entre la chaîne de caractères que représente `edt*.Ttext` et les variables `a`, `b` et `c`.

Nous calculons ensuite le discriminant. En fonction du signe du discriminant nous envisageons les 3 cas :

- `disc<0` : il n'y a pas de résultat réel ;
- `disc=0` : il y a une racine ;
- `disc>0` : il y a deux racines réelles distinctes.

En fonction des différents cas nous calculons les valeurs des racines.

À la fin il nous reste encore à transformer les valeurs réelles, résultats des calculs, en chaînes de caractères pour les affecter à la propriété `Caption` des différentes étiquettes. Nous faisons ceci avec la fonction `FloatToStr`.

La méthode `Tformulaire.btnExitClick` aura comme seule commande `Application.Terminate`. De cette manière l'événement `Click` lié au bouton `btnExit` aura comme effet-net d'arrêter l'application.

7.7 Vérification du numéro de matricule

Développons ici un exercice qui prend en entrée le numéro de matricule d'une personne et qui vérifie que le chiffre de contrôle est correct.

7.7.1 Rappelons la méthode de calcul du numéro de contrôle.

Si nous notons $a_1a_2a_3a_4m_1m_2j_1j_2n_1n_2c$ un numéro de matricule, nous formons le dernier chiffre en parcourant les étapes suivantes :

- Nous formons la somme :
  $sum = 5*a_1 + 4*a_2 + 3*a_3 + 2*a_4 + 7*m_1 + 6*m_2 + 5*j_1 + 4*j_2 + 3*n_1 + 2*n_2$;

- soit *n* le reste de la division de *sum* par 11 ;
- si *n=1* il y a une faute ;
- si *n=0* le chiffre de contrôle reste *0 ;*
- si $n \neq 0$ et $n \neq 1$ alors le chiffre de contrôle vaut $11 - n$.

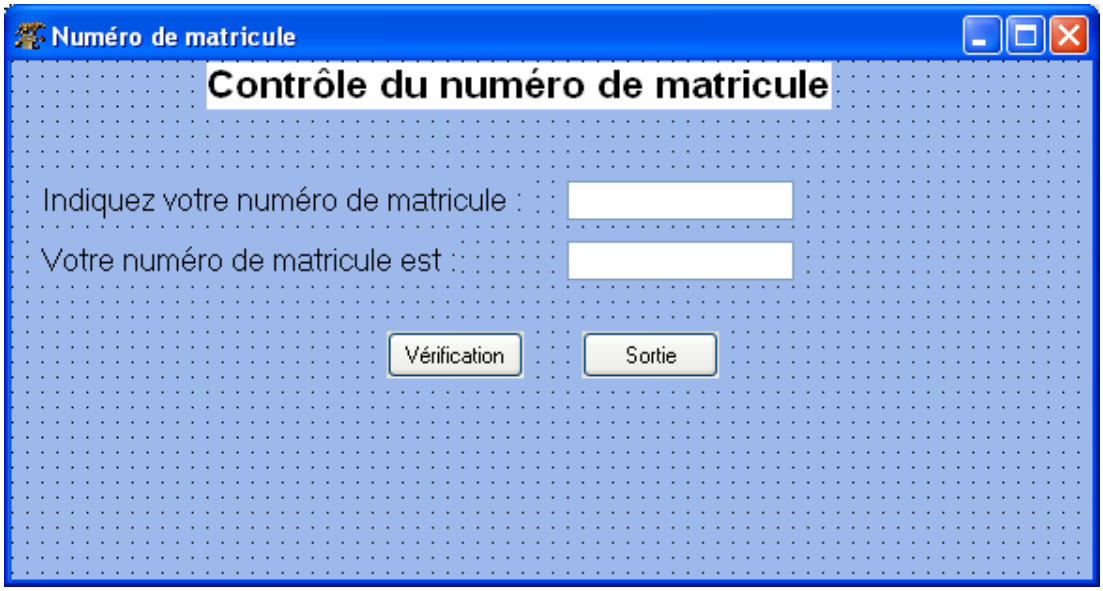
Nous essayons de traduire ceci en Lazarus.

7.7.2 Présentation visuelle

Pour cela il nous faut d'abord un formulaire que nous appellerons, comme toujours, `Tformulaire` dont une instance nous servira à manipuler les entrées et sorties.

La valeur de la propriété `Caption` de l'objet `Tformulaire` sera : Numéro de matricule.

Voici un exemple d'un tel formulaire.



Il contient les éléments suivants :

nom	**type**	**text**	**caption**
lblTitre	TLabel		Contrôle du numéro de matricule
lblSaisie	TLabel		Indiquez votre numéro de matricule :
lblResultat	TLabel		Votre numéro de matricule est :
edtSaisie	TEdit	*numéro de matricule*	

edtResultat	TEdit	*correct/faux*	
btnVerif	TButton		Vérification
btnExit	TButton		Sortie

7.7.3 Code

Une fois ce formulaire établi, nous devons programmer le code nécessaire.

Voici un exemple d'implémentation.

```
unit Unit1;
interface
uses
Classes, SysUtils, LResources, Forms, Controls, Graphics, Dialogs,
StdCtrls;
type
  TfrmMatricule = class(TForm)
    lblTitre: TLabel;
    lblSaisie: TLabel;
    edtSaisie: TEdit;
    edtResultat: TEdit;
    btnVerif: TButton;
    lblResultat: TLabel;
    btnExit: TButton;
    procedure btnVerifClick(Sender: TObject);
    procedure btnExitClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  frmMatricule: TfrmMatricule;
implementation
procedure TfrmMatricule.btnVerifClick(Sender: TObject);
var
  a1,a2,a3,a4,m1,m2,j1,j2,n1,n2,nc,c: integer;
  sum : integer;
  s : string;
begin
 s:=edtSaisie.Text;
 if length(s) <> 11 then
   ShowMessage('Numéro de matricule incomplet')
 else
 begin
   a1:= StrToInt(copy(s,1,1));
   a2:= StrToInt(copy(s,2,1));
   a3:= StrToInt(copy(s,3,1));
```

```
    a4:= StrToInt(copy(s,4,1));
    m1:= StrToInt(copy(s,5,1));
    m2:= StrToInt(copy(s,6,1));
    j1:= StrToInt(copy(s,7,1));
    j2:= StrToInt(copy(s,8,1));
    n1:= StrToInt(copy(s,9,1));
    n2:= StrToInt(copy(s,10,1));
    nc:= StrToInt(copy(s,11,1));
    sum := 5*a1+4*a2+3*a3+2*a4+7*m1+6*m2+5*j1+4*j2+3*n1+2*n2;
    sum := sum mod 11;
    if sum = 1 then s:='faux'
    else
    begin
      if sum = 0 then c:= 0
      else c:= 11-sum;
      if nc=c then s:='correct'
      else s:='faux';
    end;
    edtResultat.Text:=s;
 end;
end;
procedure TfrmMatricule.btnExitClick(Sender: TObject);
begin
  Application.Terminate;
end;
initialization
  {$I unit1.lrs}
end.
```

7.7.4 Explication du code

La variable `s` va contenir le numéro de matricule saisi. C'est la valeur saisie.

Pour éviter qu'un utilisateur ne donne qu'une partie d'un numéro de matricule, nous faisons un test sur la longueur du numéro et nous affichons une erreur si la longueur ne correspond pas.

Le message d'erreur est affiché par la procédure `ShowMessage` dont la syntaxe est la suivante :

```
ShowMessage(msg: string);
```

où

msg chaîne de caractères à afficher.

Ensuite nous extrayons les différentes valeurs du numéro de matricule. Comme toutes les valeurs saisies sont des caractères nous devons les transformer en entiers par la fonction `StrToInt`.

La fonction `copy` sert à extraire des parties de chaînes de caractères. Sa syntaxe est la suivante :

```
Copy(s, index, count: Integer): string;
```

où :

S	chaîne de laquelle est extraite la partie ;
Index	indice de début de la chaîne à extraire (commence par 1) ;
Count	nombre de caractères à extraire.

Les étapes suivantes correspondent à l'algorithme énoncé.

Pour terminer nous éditons le résultat comme texte du champ `edtResultat`.

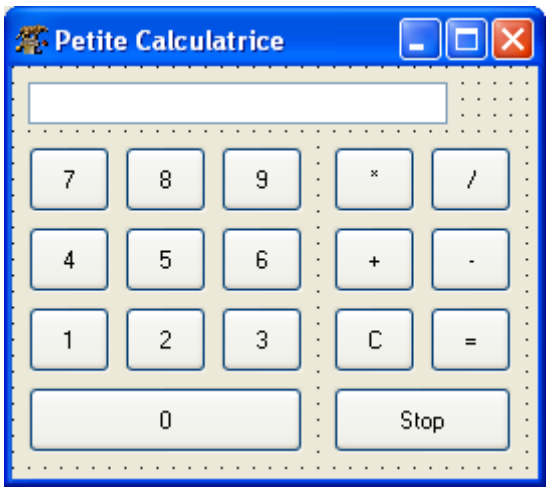
La procédure `Tformulaire.btnExitClick` sert de nouveau à arrêter l'application.

7.8 Une petite machine à calculer

Dans ce prochain exercice nous nous proposons de mettre en œuvre une petite machine à calculer, qui effectuera les 4 opérations élémentaires.

7.8.1 Présentation visuelle

Comme toujours il nous faut définir d'abord un formulaire. Voici une possibilité d'une telle interface entre l'opérateur et la machine.



Nous appellerons, comme toujours Tformulaire l'objet que nous allons définir ci-dessous.

Nom	type	text	caption
edtNum	TEdit	Saisie des nombres et des opérateurs qui interviennent dans le calcul	
btnButton0	TButton		0
btnButton1	TButton		1
btnButton2	TButton		2
btnButton3	TButton		3
btnButton4	TButton		4
btnButton5	TButton		5
btnButton6	TButton		6
btnButton7	TButton		7
btnButton8	TButton		8

btnButton9	TButton		9
btnButtonclear	TButton		C
btnButtondiv	TButton		/
btnButtonequal	TButton		=
btnButtonminus	TButton		-
btnButtonmult	TButton		*
btnButtonplus	TButton		+
btnButtonarret	TButton		Stop

7.8.2 Code

Une possibilité de code pour cette machine à calculer est le suivant :

```
unit Unit1;

interface

uses
Classes, SysUtils, LResources, Forms, Controls, Graphics, Dialogs,
StdCtrls;
type
  Tformulaire = class(TForm)
    edtNum: TEdit;
    btnButton7: TButton;
    btnButton1: TButton;
    btnButton9: TButton;
    btnButton8: TButton;
    btnButton6: TButton;
    btnButton5: TButton;
    btnButton2: TButton;
    btnButton4: TButton;
    btnButton3: TButton;
    btnButton0: TButton;
    btnButtonmult: TButton;
    btnButtondiv: TButton;
    btnButtonclear: TButton;
    bnButtonminus: TButton;
    btnButtonplus: TButton;
    btnButtonequal: TButton;
    btnButtonArret: TButton;
    procedure FormCreate(Sender: TObject);
    procedure btnButton0Click(Sender: TObject);
    procedure btnButtonplusClick(Sender: TObject);
    procedure bnButtonminusClick(Sender: TObject);
    procedure btnButtonmultClick(Sender: TObject);
    procedure btnButtondivClick(Sender: TObject);
    procedure btnButtonclearClick(Sender: TObject);
    procedure btnButtonequalClick(Sender: TObject);
```

```
    procedure btnButtonArretClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
    flag, n1, n2, op : integer;
    n3 : real;
    st : string;
  end;

var
  formulaire: Tformulaire;

implementation


procedure Tformulaire.FormCreate(Sender: TObject);
begin
  flag:=0;
end;

procedure Tformulaire.btnButton0Click(Sender: TObject);
var strNum:string;
begin
  if Sender=btnButton0 then  strNum:='0' else
  if Sender=btnButton1 then  strNum:='1' else
  if Sender=btnButton2 then  strNum:='2' else
  if Sender=btnButton3 then  strNum:='3' else
  if Sender=btnButton4 then  strNum:='4' else
  if Sender=btnButton5 then  strNum:='5' else
  if Sender=btnButton6 then  strNum:='6' else
  if Sender=btnButton7 then  strNum:='7' else
  if Sender=btnButton0 then  strNum:='8' else
  strNum:='9';

  if flag=0 then edtNum.Text:=edtNum.Text + strNum
  else
  begin
    edtNum.Text:=strNum;
    flag:=0
  end;
end;

procedure Tformulaire.btnButtonplusClick(Sender: TObject);
begin
  n1:=strtoint(edtNum.Text);
  flag := 1;
  op := 1;
end;

procedure Tformulaire.bnButtonminusClick(Sender: TObject);
begin
```

```
  n1:=strtoint(edtNum.Text);
  flag := 1;
  op := 2;
end;

procedure Tformulaire.btnButtonmultClick(Sender: TObject);
begin
  n1:=StrToInt(edtNum.Text);
  flag := 1;
  op := 3;
end;

procedure Tformulaire.btnButtondivClick(Sender: TObject);
begin
  n1:=StrToInt(edtNum.Text);
  flag := 1;
  op := 4;
end;

procedure Tformulaire.btnButtonclearClick(Sender: TObject);
begin
  edtNum.Text := '';
  n1 := 0;
end;

procedure Tformulaire.btnButtonequalClick(Sender: TObject);
begin
  n2:=StrToInt(edtNum.Text);
  case op of
    1: n3:=n1+n2;
    2: n3:=n1-n2;
    3: n3:=n1*n2;
    4: n3:=n1/n2;
  end;//case
  edtNum.Text:=FloatToStr(n3);
  flag := 1;
  op := 4;
end;

procedure Tformulaire.btnButtonArretClick(Sender: TObject);
begin
  Application.Terminate;
end;
initialization
  {$I unit1.lrs}
end.
```

7.8.3 Explication du code

La variable `flag` est initialisée à 0 lors du lancement du formulaire (événement `FormCreate` du formulaire). Si tel est le cas, tout nombre saisi dans `edtNum` est concaténé à la chaîne de caractères déjà saisie. Si on clique sur un signe d'opération, le contenu de `edtNum` constituera un des deux

opérants. Dans ce cas, `flag` devient 1, `edtNum.Text` est copié comme valeur dans `n1` ou `n2`, `edtNum.Text` est remis à la chaîne vide, `flag` redevient 0 et `edtNum` est prêt à contenir le prochain opérant.

Comme les boutons 0…9 contiennent chacun un code très similaire, on s'est servi de la variable `Sender`, qui détermine quel est le bouton qui a lancé l'événement `btnButton0Click`. De cette manière, on pourra utiliser le même événement `btnButton0Click` pour chacun des boutons.

Le reste du programme s'explique facilement à la lecture. Il reste à remarquer que la calculatrice ne respecte pas la priorité des opérations.

7.9 Calcul matriciel - utilisation du composant StringGrid

Le prochain programme que nous allons établir est un programme qui manipule les opérations sur les matrices 2x2.

Exercice :

Il est laissé au lecteur la possibilité de changer ce programme pour la manipulation des matrices à 3 dimensions.

7.9.1 Le composant StringGrid

Dans cet exercice nous utilisons le type prédéfini : matrice ou `StringGrid` qui se trouve dans la barre des objets sous *Additional*. Il possède de nouvelles propriétés dont nous énumérons ici les plus importantes :

propriété	**Type**	**explications**
ColCount	Integer	nombre de colonnes
RowCount	Integer	nombre de lignes
FixedCols	Integer	nombre de colonnes d'en-têtes
FixedRows	Integer	nombre de lignes d'entêtes
DefaultColWidth	Integer	largeur des colonnes (pixels)
DefaultRowHeight	Integer	hauteur des colonnes (pixels)
Cells		ensemble de cellules
goEditing (Options)	Boolean	indique si l'utilisateur peut introduire des valeurs dans les cellules

Nous référençons une cellule par `Cells[colonne,ligne]`.

Attention : nous devons faire attention que le comptage des lignes et des colonnes commence, comme si souvent, par 0.

7.9.2 Le composant ListBox

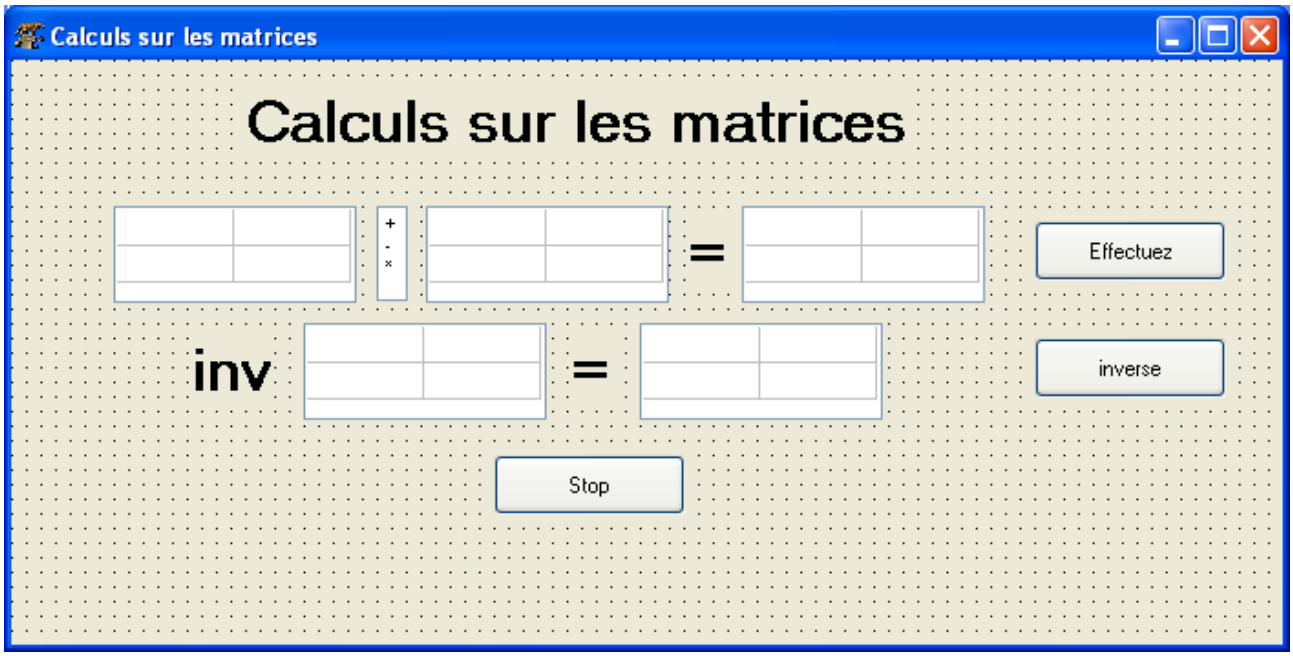
Dans cet exemple nous utilisons aussi un nouveau type, la `ListBox`. Elle sert à afficher plusieurs lignes de caractères et à donner la possibilité à l'utilisateur de choisir une ligne précise.

Nous trouvons la `ListBox` dans la barre des composants standards. Trois propriétés sont importantes à relever :

Propriété	**type**	**explications**
Items	string	tableau des lignes de la liste
ItemIndex	entier	indice de la ligne sélectionnée
Sorted	booléen	lignes triées ou non

7.9.3 Présentation visuelle

Commençons d'abord, comme dans les exercices précédents, par établir un formulaire qui nous sert à saisir les données et à les afficher. Voici un exemple d'un tel formulaire.



Ce formulaire présente les composants suivants :

name	**type**	**text**	**caption**
lblTitre	TLabel		Calculs sur les matrices
lblEg1	TLabel		=
lblEg2	TLabel		=
lblInv	TLabel		inv
btnEff	TButton		effectuez
btnInv	TButton		inverse

btnArret	TButton		Stop
name	**type**	**items**	
lbOp	TListBox	+ - *	
name	**type**	**colcount / rowcount**	**fixedcols / fixedrows**
sgMat1	TStringGrid	2/2	0/0
sgMat2	TStringGrid	2/2	0/0
sgMat3	TStringGrid	2/2	0/0
sgMatInv	TStringGrid	2/2	0/0
sgMatRes	TStringGrid	2/2	0/0

Les composants de type `StringGrid` qui servent à introduire des matrices doivent avoir l'option `goEditing` avec la valeur `True`.

Le champ `lbOp` de type `ListBox` sert à énumérer les différentes opérations et à donner à l'utilisateur la possibilité de choisir.

Après l'élaboration de ce formulaire nous pouvons écrire le code nécessaire. Voici un exemple possible.

```
unit Unit1 ;
interface
uses
Classes, SysUtils, LResources, Forms, Controls, Graphics, Dialogs,
StdCtrls, Grids;
type
  Tformulaire = class(TForm)
    lblTitre : Tlabel ;
    sgMat1 : TstringGrid ;
    sgMat2 : TstringGrid ;
    sgMatRes : TstringGrid ;
    sgMat3 : TstringGrid ;
    sgMatInv : TstringGrid ;
    lblEg1: Tlabel;
    lblEg2: Tlabel;
    lblInv: Tlabel;
    lbOp: TlistBox;
    btnEff: Tbutton;
    btnInv: Tbutton;
    btnArret: Tbutton;
    procedure btnEffClick(Sender: Tobject);
    procedure btnInvClick(Sender: Tobject);
    procedure btnArretClick(Sender: Tobject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;//Tformulaire
```

```
var
  formulaire: Tformulaire;

implementation
procedure Tformulaire.btnEffClick(Sender: TObject);
var a,i,j : integer;
begin
  if lbOp.ItemIndex=0 then
    for i:=0 to 1 do
      for j:=0 to 1 do
      begin
        a:=StrToInt(sgMat1.Cells[i,j])+StrToInt(sgMat2.Cells[i,j]);
        sgMatRes.Cells[i,j]:=IntToStr(a);
      end;
      //j
    //i
  //fi index=0
  if lbOp.ItemIndex=1 then
    for i:=0 to 1 do
      for j:=0 to 1 do
      begin
        a:= StrToInt(sgMat1.Cells[i,j])-StrToInt(sgMat2.Cells[i,j]);
        sgMatRes.Cells[i,j]:=IntToStr(a);
      end;
      //j
    //i
  //fi index=1
  if lbOp.ItemIndex=2 then
    for i:=0 to 1 do
      for j:=0 to 1 do
      begin
        a:=StrToInt(sgMat1.Cells[0,j])*StrToInt(sgMat2.Cells[i,0])
        +StrToInt(sgMat1.Cells[1,j])*StrToInt(sgMat2.Cells[i,1]);
        sgMatRes.Cells[i,j]:=IntToStr(a);
      end;
      //j
    //i
  //fi index=0
end;//btnEffClick

procedure Tformulaire.btnInvClick(Sender: TObject);
var a,det:real;
begin
  det:=StrToInt(sgMat3.Cells[0,0])*StrToInt(sgMat3.Cells[1,1])
     -StrToInt(sgMat3.Cells[1,0])*StrToInt(sgMat3.Cells[0,1]);
  a:= 1/det*StrToFloat(sgMat3.Cells[1,1]);
  sgMatInv.Cells[0,0]:=FloatToStr(a);
  a:= (-1/det)*StrToInt(sgMat3.Cells[1,0]);
  sgMatInv.Cells[1,0]:=FloatToStr(a);
  a:= (-1/det)*StrToInt(sgMat3.Cells[0,1]);
  sgMatInv.Cells[0,1]:=FloatToStr(a);
  a:= 1/det*StrToInt(sgMat3.Cells[0,0]);
```

```
  sgMatInv.Cells[1,1]:=FloatToStr(a);
end;//btnInvClick

procedure Tformulaire.btnArretClick(Sender: TObject);
begin
  Application.Terminate
end;//btnArret
initialization
  {$I unit1.lrs}
end.//Unit1
```

7.9.4 Explication du code

Comme les opérations +, - et * exigent deux opérateurs, mais que l'opération *inverse* n'a besoin que d'un seul, nous avons pu regrouper les trois premiers sous une seule procédure.

La `TListBox lbOp` nous donne les valeurs suivantes :

ItemIndex	opération
0	addition
1	soustraction
2	multiplication

Pour le reste, le contenu des différentes parties des procédures correspond aux règles mathématiques qui définissent les opérations sur les matrices.

8 La récursivité

8.1 Exemple

La fonction suivante calcule une puissance de base réelle non nulle et d'exposant naturel :

```
function puissance(x:real;m:integer):real;
begin
  if m=0 then result:=1
  else result:=x*puissance(x,m-1)
end;
```

Cette fonction présente une grande différence par rapport à toutes les fonctions que nous avons définies précédemment. Dans la définition même on trouve déjà un appel à la fonction puissance. Il s'agit ici d'un mécanisme très puissant, présent dans tous les langages de programmation modernes : la récursivité. Le fonctionnement exact de ce mécanisme ainsi que les conditions d'utilisation seront étudiées en détail dans les paragraphes suivants. Remarquons cependant qu'il existe un lien étroit entre la récursivité en informatique et la récurrence en mathématique. La définition de la fonction puissance présentée ici est une transcription quasi directe des formules $\begin{cases} x^0 = 1 \\ x^m = x \cdot x^{m-1} \end{cases}$, valables pour x non nul[5].

8.2 Définition : « fonction ou procédure récursive »

On dit qu'une fonction ou une procédure est *récursive* (de manière directe) si elle s'appelle elle-même. Une fonction ou une procédure est *récursive de manière indirecte* si elle appelle une autre fonction ou procédure qui rappelle la première de façon directe ou indirecte.

La fonction puissance du paragraphe précédent est bien sûr une fonction récursive **directe**.

Le mécanisme de la récursivité est très puissant, mais il faut une certaine expérience pour pouvoir l'utiliser dans de bonnes conditions. Dans la suite nous allons élucider principalement les aspects suivants :

- Sous quelles conditions et pourquoi une fonction récursive donne-t-elle le résultat attendu ? Comment vérifier qu'une telle fonction est correcte ?
- Comment le système gère-t-il une fonction récursive ? C'est-à-dire comment est-ce que ce mécanisme fonctionne en pratique ?
- Est-ce qu'une fonction récursive est « meilleure » ou « moins bonne » qu'une fonction itérative (normale) ?

Il est clair que ces différents aspects ne sont pas indépendants les uns des autres, mais qu'il faut une vue d'ensemble pour bien les comprendre.

[5] La fonction «`puissance`» donne un résultat incorrect si x et m sont nuls. De plus, il est nécessaire que m soit un entier positif !

8.3 Etude détaillée d'un exemple

Dans ce paragraphe nous revenons à la fonction « puissance » de la page précédente et nous allons commencer par étudier quelques exemples d'exécution.

`puissance(7,0)` : donne bien sûr comme résultat 1 vu que la condition `m=0` est vérifiée.

`puissance(7,1)` : la condition `m=0` n'est pas vérifiée et la fonction calcule donc d'abord «`x*puissance(x,m-1)`» c'est-à-dire «`7*puissance(7,0)`» ce qui donne dans une deuxième étape «`7*1=7`».

`puissance(7,2)` : ici la condition m=0 n'est pas non plus vérifiée et la fonction calcule donc aussi d'abord «`x*puissance(x,m-1)`» c'est-à-dire «`7*puissance(7,1)`» . Suivent ensuite les deux étapes précédentes. A la fin de la troisième étape le résultat obtenu est «`7*puissance(7,1)=7*[7*puissance(7,0)]=7*7*1=49`».

Il est important de remarquer ici que le système refait chaque fois toutes les étapes et « ne se souvient pas » des appels de fonctions précédents. L'exécution de `puissance(7,12)` nécessite 13 passages dans la fonction : d'abord 12 appels récursifs et ensuite un dernier passage où `m=0`.

L'exemple `puissance(7,-2)` est particulièrement intéressant. La condition `m=0` n'est pas vérifiée et la fonction calcule donc «`x*puissance(x,m-1)`» c'est-à-dire «`7*puissance(7,-3)`». Ensuite elle va évaluer «`7*puissance(7,-4)`», «`7*puissance(7,-5)`», «`7*puissance(7,-6)`», etc. Il est clair que cet appel ne va certainement pas donner le résultat 1/49. Mais la situation est plus grave, l'exécution de la fonction ne va pas donner de faux résultat, mais cette exécution ne va pas se terminer[6] vu que la condition `m=0` ne sera plus jamais vérifiée.

Pour qu'une fonction ou une procédure récursive s'arrête, il est nécessaire que le code vérifie les conditions suivantes :

- **Pour une ou plusieurs valeurs des données, appelées « cas de base », la fonction calcule directement (sans appel récursif) le résultat.**
- **Dans le code de la fonction il doit être assuré que chaque suite d'appels récursifs va toujours finir par atteindre un cas de base.**

Il est clair que le fait que la fonction s'arrête, signifie seulement qu'elle va fournir un résultat, mais non pas que ce résultat est correct. L'arrêt de la fonction est une condition préalable !

Dans notre exemple, il est donc nécessaire de préciser que la fonction « puissance » ne convient pas pour les exposants négatifs[7].

Dans une démonstration par récurrence en mathématiques la situation est semblable : Pour montrer qu'une propriété est vraie pour tout entier naturel, on montre d'abord qu'elle est vraie pour le cas de base n=0 et ensuite on montre que si la formule est vraie pour le naturel n, alors elle reste vraie pour n+1. La véracité de la propriété pour n=4 est alors ramenée successivement à n=3, n=2, n=1 jusqu'au cas de base n=0, que l'on sait vérifié.

---

[6] Dans cette situation le système risque d'entrer dans un état indéfini provoqué par un débordement de mémoire. Dans le pire cas il sera nécessaire de redémarrer l'ordinateur (RESET) avec toutes les conséquences que cela peut impliquer. Dans un cas plus favorable, Lazarus détecte le problème et avorte l'exécution de la fonction.

[7] Elle ne convient pas non plus pour les exposants non entiers, mais ceux-ci sont de toute façon exclus vu que la variable exposant m est de type integer.

8.4 Fonctionnement interne

Dans ce paragraphe on va chercher une réponse à la question comment le système gère l'exécution d'une procédure récursive. La bonne compréhension de ce mécanisme est importante pour pouvoir rédiger des programmes efficaces.

Revenons à l'exemple de la fonction `factorielle` qui calcule la factorielle d'un nombre naturel donné. Cet exemple, déjà implémenté de façon itérative au chapitre précédent, se prête particulièrement bien à être programmé de façon récursive vu que la définition mathématique de la fonction se base sur des formules de récurrence.

$$\begin{cases} 0! = 1 \\ n! = n \cdot (n-1)! \; si \; n \in N_0 \end{cases} \text{ ou bien } \begin{cases} 0! = 1 \\ n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1 \; si \; n \in N_0 \end{cases}$$

```
function factorielle(n:integer):integer;
begin
  if n=0 then result:=1
  else result:=n*factorielle(n-1)
end;
```

On peut l'intégrer dans le programme Lazarus du chapitre précédent. Le reste du code reste identique.

Lors de l'exécution de `factorielle(1)` le système exécute la fonction jusqu'à l'appel récursif « `factorielle(n-1)` » A ce stade le système mémorise l'état actuel de toutes les variables locales de la fonction. Lors de l'exécution de « `factorielle(n-1)` » le système recommence à exécuter la fonction factorielle avec le paramètre `n-1=0`. Lors de ce deuxième passage dans la fonction, les variables locales sont réinitialisées, leurs anciennes valeurs ne sont pas accessibles à ce moment mais elles sont sauvegardées pour une réutilisation ultérieure. Maintenant on a donc `n=0` et le système termine ce passage dans la fonction avec `result:=1`. Ensuite le système revient dans le premier passage de la fonction à l'endroit `factorielle(n-1)`. Les variables locales récupèrent leur ancienne valeur et l'expression `n*factorielle(n-1)` est évaluée avec `n=1` et `factorielle(n-1)=1`. La variable `result` prend la valeur 1 et l'exécution quitte définitivement la fonction.

Pour des appels de fonction avec des arguments plus grands, par exemple `factorielle(15)` le système doit donc conserver autant de copies de toutes les variables locales qu'il y a d'appels récursifs avant d'atteindre le cas de base. Cela peut nécessiter une quantité appréciable de mémoire et de temps d'exécution pour la gestion qui en découle.

8.5 Exactitude d'un algorithme récursif

Il est souvent «facile » de montrer qu'une fonction récursive donne le bon résultat. Normalement le raisonnement par récurrence s'impose. Pour la fonction `factorielle`, par exemple :

Base : `n=0`, le résultat est effectivement 1.

Hypothèse de récurrence : supposons que la fonction `factorielle` donne le bon résultat jusqu'au rang `n-1` pour `n>0`, c'est-à-dire que `factorielle(n-1)=(n-1)!`.

Pont : montrons que la fonction donne encore le bon résultat au rang `n`, donc que `factorielle(n)=n!`. En effet, la fonction donne le résultat `n*factorielle(n-1)` qui est

égal à `n*(n-1)!` par l'hypothèse de récurrence et qui est encore égal à `n!` par la définition mathématique de la factorielle.

Cette démonstration ne prend pas en compte des problèmes de dépassement de mémoire si `n` est « trop grand ».

8.6 Comparaison : fonction récursive – fonction itérative[8]

L'exemple suivant est particulièrement impressionnant.

La célèbre suite de Fibonacci définie par $\begin{cases} u_0 = 1 \; et \; u_1 = 1 \\ u_n = u_{n-1} + u_{n-2} \;,\; n \in N - \{0;1\} \end{cases}$, s'implémente directement par la fonction récursive suivante :

```
function fibo(n:integer):longint;
begin
 if n<=1 then result:=1
 else result:=fibo(n-1)+fibo(n-2)
end;
```

Le type du résultat est `longint`, une variante de `integer` qui permet de représenter des nombres entiers plus grands. La condition `n<=1` prend en charge les deux cas de base `n=0` et `n=1`.

La fonction suivante, dont l'algorithme est basé sur une simple boucle n'est pas vraiment difficile à comprendre non plus. Si `n` est différent de 0 et de 1, alors tous les termes de la suite jusqu'au n-ième sont calculés de proche en proche (comme on le ferait si on n'avait pas d'ordinateur à disposition).

```
function fibo_it(n:integer):longint;
var t0,t1:longint;
begin
 if n<=1 then result:=1
 else
  begin
   t0:=1;t1:=1;
   while n>1 do
     begin
        result:=t0+t1;
        t0:=t1;
        t1:=result;
        n:=n-1;
    end;
  end;
end;
```

Pour se faire une idée de la situation, il est conseillé de tester les 2 fonctions. En prenant successivement les arguments 5, 10, 20, 30, 40, 50…, on va finir par constater la même chose indépendamment de l'ordinateur utilisé. Même sur un ordinateur rapide la fonction récursive va finir par devenir terriblement lente alors que la fonction itérative va rester rapide (résultat immédiat) même sur un ordinateur très faible ! Pourquoi ?

[8] Iteratif: algortihme basé sur une boucle.

Que fait la fonction itérative ? Si `n>1`, cette fonction effectue exactement `n-1` additions sur des nombres du type `longint`, et un nombre approximativement proportionnel à `n` d'affectations, de soustractions et de comparaisons. Le temps d'exécution de ces opérations est négligeable par rapport à la croissance fulgurante des résultats : c'est-à-dire la fonction sera limitée par la capacité de représentation du type `longint` avant qu'on remarque un quelconque ralentissement dans l'exécution !

Que fait la fonction récursive ? Etudions les exécutions de la fonction pour les arguments 2 à 5.

Pour `fibo(2)` [résultat : 2], la condition n'est pas vérifiée et la fonction calcule donc l'expression `fibo(1)+fibo(0)`. Les deux appels récursifs donnent directement le résultat 1 et l'exécution se termine donc « assez rapidement » après seulement 3 passages (chaque fois 1 appel avec les arguments 0, 1 et 2) dans la fonction.

Pour `fibo(3)` [résultat : 3], la fonction calcule d'abord `fibo(2)+fibo(1)`. Le premier appel nécessite trois passages (voir alinéa précédent) dans la fonction et le deuxième appel donne directement 1. Cela fait donc un total de 5 appels.

Pour `fibo(4)` [résultat : 5], la fonction calcule `fibo(3)+fibo(2)`. Le nombre d'appels se calcule par 5 (pour `fibo(3)`) + 3 (pour `fibo(2)`) + 1 (pour `fibo(4)`) = 9.

Sans entrer dans tous les détails : Pour `fibo(10)` [résultat : 89], il faut 177 appels, pour `fibo(20)` [résultat : 10946], il faut 21891 appels et pour `fibo(30)` [résultat : 1346269], il faut 2692537 appels.

Il est clair que le nombre d'appels de fonctions doit être supérieur ou égal au résultat. En effet, les deux seuls cas de bases donnent le résultat 1 et l'appel récursif consiste à ajouter deux résultats intermédiaires. Le résultat final est donc atteint en ajoutant tant de fois le nombre 1. A côté du nombre impressionnant de calculs que la fonction effectue, il ne faut pas oublier la quantité d'espace mémoire nécessaire pour sauvegarder toutes les valeurs intermédiaires de la variable `n`.

8.7 Récursif ou itératif ?

Contrairement à ce que l'exemple précédent pourrait suggérer, un algorithme récursif n'est pas automatiquement « moins bon » qu'un algorithme itératif. Si on tient compte de l'évaluation interne d'une fonction récursive, on peut parfois trouver une variante efficace de la fonction récursive.
Voici par exemple une formulation récursive « efficace » de la fonction `fibo`. L'astuce consiste à utiliser une fonction auxiliaire récursive avec deux arguments supplémentaires, nécessaires pour passer les deux termes précédents à l'appel suivant. On remarquera que dans le code de cette fonction il n'y a qu'un seul appel récursif, de façon à ce que le nombre d'appels pour calculer le n^e terme de la suite de Fibonacci ne dépasse pas `n`.

```
function fibo(n:integer):longint;
  function fib_aux(n:integer;i,j:longint):longint;
  begin
    if n<=1 then result:=i
    else result:=fib_aux(n-1,i+j,i)
  end;
begin
  result:=fib_aux(n,1,1)
end;
```

Un certain nombre de problèmes admettent une solution récursive « très élégante » (algorithme simple à rédiger et à comprendre, mais pas nécessairement efficace). Nous verrons des exemples de ce genre dans le chapitre sur les recherches et les tris[9].

Pour certains de ces problèmes, une formulation à l'aide de boucles est très compliquée. Mais une telle formulation existe toujours. Dans le pire des cas, il est nécessaire de « simuler » l'algorithme récursif en stockant toutes les valeurs intermédiaires des variables dans un tableau. C'est de cette façon-là que Lazarus lui-même évalue les fonctions récursives ; en effet le langage machine, seul langage compris par le processeur, est un langage relativement faible qui ne connaît pas le mécanisme de la récursivité.

8.8 Exercices

Exercice 8-1

a) Ecrivez un programme qui calcule la fonction d'Ackermann à deux variables naturelles définie par les égalités :

$$\begin{cases} Ack(0,m) = m+1 \\ Ack(k,0) = Ack(k-1,1) \; si \; k \geq 1 \\ Ack(k,m) = Ack(k-1, Ack(k,m-1)) \; si \; k,m \geq 1 \end{cases}$$

b) Essayez cette fonction pour quelques arguments. Par exemple `ack(3,5)=253`, `ack(4,0)=13`, `ack(4,1)=65533`,

Exercice 8-2

Complétez la fonction puissance pour qu'elle calcule aussi correctement des puissances à exposant entier négatif.

Exercice 8-3

Ecrivez une version récursive de la fonction puissance rapide.

Aide : utilisez les formules $\begin{cases} a^0 = 1 \\ a^{2n} = \left(a^2\right)^n \\ a^{2n+1} = a^{2n} \cdot a \end{cases}$

Expliquez pour quelles valeurs de `a` et de `n`, l'exécution va s'arrêter et donner un résultat correct.

Exercice 8-4

Ecrivez une version récursive de la fonction pgcd.

Exercice 8-5

Ecrivez un programme récursif qui transforme un nombre binaire en notation décimale.

Exercice 8-6

Ecrivez un programme récursif qui transforme un nombre décimal en notation binaire.

[9] Voir par exemple « recherche dichotomique » et « quicksort ».

9 Comptage et fréquence

9.1 Algorithme de comptage

On veut réaliser une fonction qui compte le nombre d'occurrences d'une chaîne de caractères dans une liste donnée.

Pour réaliser cet algorithme on parcourt la liste élément après élément et on le compare avec la chaîne trouvée. Si les deux chaînes sont identiques, on augmente un compteur.

```
function compter(liste:TListbox;chaine:string):integer;
var
  i,r:integer;
begin
  r:=0;
  for i:=0 to liste.Items.Count-1 do
    if liste.Items[I] = chaine then r:=r+1;
  result:=r;
end;
```

Exercice 9-1

Réalisez une fonction qui permet de compter le nombre d'occurrences d'une lettre dans une chaîne de caractères donnée.

9.2 Fréquence d'une lettre dans une liste

Soit une liste dont les éléments sont des lettres. On veut savoir combien de fois chaque lettre est représentée dans la liste. Ce type d'algorithme est un élément très important des algorithmes de compression, comme par exemple celui de Huffman.

La solution proposée utilise de manière un peu inhabituelle les tableaux, mais de ce fait-là devient très élégante. On utilise comme indice du tableau des lettres.

Le résultat de l'analyse de la liste se trouve dans une deuxième liste passée comme paramètre et a le format suivant : `lettre : nb. d'occurrences`.

```
procedure frequence(liste:TListbox; var resultat:TListbox);
var
  i:integer;
  c:char;
  tableau:array['A'..'Z'] of integer;
  element:string;
begin
  for c:='A' to 'Z' do tableau[c]:=0;
  for i:=0 to liste.Items.Count-1 do
    begin
      element:=liste.Items[i];
      tableau[element[1]]:=tableau[element[1]]+1;
    end;
  resultat.Items.Clear;
  for c:='A' to 'Z' do
    resultat.Items.Append(c+' : '+inttostr(tableau[c]));
end;
```

Exercice 9-2

Réalisez un sous-programme qui permet de compter le nombre d'occurrences des différentes lettres dans une chaîne de caractères donnée. Le résultat du sous-programme est un tableau.

10 Recherche et tri

10.1 Introduction

Les algorithmes de recherche et de tri ont une très grande importance en informatique. C'est surtout dans le contexte des bases de données que ces algorithmes sont utilisés. Nous allons traiter en premier lieu les algorithmes de tri car pour fonctionner certains algorithmes de recherche présument des données triées.

Bien que l'illustration des différents algorithmes se base sur un petit nombre de données, il ne faut pas oublier que ces algorithmes sont en général appliqués sur un nombre très grand de données (plus que 10000, comme par exemple dans un annuaire téléphonique).

10.2 Sous-programmes utiles

Nous allons définir deux sous-programmes qui vont nous permettre de faciliter la programmation et le test des algorithmes de tri et de recherche.

10.2.1 Echange du contenu de deux variables entières

```
procedure echange (var liste:TListbox; posa,posb:integer);
var
  temp: string;
begin
  temp:=liste.Items[posa];
  liste.Items[posa]:=liste.Items[posb];
  liste.Items[posb]:=temp;
end;
```

Cette procédure échange le contenu de deux éléments d'une liste.

10.2.2 Remplissage d'une liste

```
procedure remplissage(var liste: TListbox; n: integer);
const
  a=ord('A');
var
  i,j:integer;
  s : string ;
begin
  liste.Clear;
  for i:= 1 to n do
   begin
    s:='';
    for j:=0 to random(5)+1 do
      s:=s+chr(random(26)+a);
    liste.Items.Append(s);
   end;
end;
```

Cette procédure remplit une liste avec `n` mots (comportant de 2 à 6 lettres) pris au hasard.

10.3 Les algorithmes de tri

Le but d'un algorithme de tri est d'arranger des données selon un critère imposé. Ce critère représente une relation entre les données comme par exemple le tri d'une liste de mots selon l'ordre lexicographique ou le tri d'une liste de nombre en ordre croissant.

10.3.1 Tri par sélection

10.3.1.1 Idée

On cherche dans la liste le plus petit élément et on l'échange avec le premier élément de la liste. Ensuite on recommence l'algorithme en ne prenant plus en considération les éléments déjà triés et ceci jusqu'à ce qu'on arrive à la fin de la liste.

10.3.1.2 Solution récursive

```
procedure tri_selection_r(var liste:TListbox; debut:integer);
var
  j,min:integer;
begin
  min:=debut;
  for j:=debut+1 to liste.Items.Count-1 do
    if liste.Items[j]<liste.Items[min] then min:=j;
  echange(liste,debut,min);
  if debut < liste.Items.Count-2 then
    tri_selection_r(liste,debut+1);
end;
```

10.3.1.3 Solution itérative

```
procedure tri_selection_i(var liste:TListbox);
var
  i,j,min:integer;
begin
  for i:=0 to liste.Items.Count-2 do
    begin
      min:=i;
      for j:=i+1 to liste.Items.Count-1 do
        if liste.Items[j]<liste.Items[min] then min:=j;
      echange(liste,i,min);
    end;
end;
```

10.3.1.4 Exercice

Exécutez pas à pas les deux algorithmes en utilisant la liste suivante :

E ;X ;E ;M ;P ;L ;E.

10.3.2 Tri par insertion

10.3.2.1 Idée

On considère un élément après l'autre dans la liste et on cherche sa position dans la liste déjà triée. Ensuite on l'insère à la juste position. De ce fait les éléments suivants de la liste doivent être déplacés.

10.3.2.2 Solution récursive

```
procedure tri_insertion_r(var liste: TListbox;gauche,droite: integer);
var j:integer;
    candidat:string;
begin
  if gauche<droite then
   begin
    tri_insertion_r(liste,gauche,droite-1);
    candidat:=liste.Items[droite];
    j:=droite;
    while (j>0) and (liste.Items[j-1] > candidat) do
        begin
          liste.Items[j]:=liste.Items[j-1];
          j:=j-1;
        end;
    liste.Items[j]:=candidat;
   end;
end;
```

10.3.2.3 Solution itérative

```
procedure tri_insertion_i(var liste:TListbox);
var  i,j:integer;
     candidat:string;
begin
  for i:= 1 to liste.Items.Count-1 do
    begin
      candidat:=liste.Items[i];
      j:=i;
      while (j>0) and (liste.Items[j-1] > candidat) do
        begin
          liste.Items[j]:=liste.Items[j-1];
          j:=j-1;
        end;
      liste.Items[j]:=candidat;
    end;
end;
```

10.3.2.4 Exercice

Exécutez pas à pas les deux algorithmes en utilisant la liste suivante :

C;A;R;T;O;O;N.

10.3.3 Tri rapide

10.3.3.1 Introduction

Le tri rapide (*Quicksort*) est l'algorithme de tri le plus utilisé. Il est réputé pour sa vitesse de tri qui pour des listes de grande taille est souvent bien supérieure à celle d'autres algorithmes de tri. Nous allons développer le tri rapide en plusieurs étapes.

10.3.3.2 Idée

L'algorithme de tri va diviser la liste en deux parties. Ces parties qui ne sont pas forcément de taille égale, sont alors triées séparément par le même algorithme. L'important dans cet algorithme est la stratégie comment la liste est divisée en deux sous-listes.

10.3.3.3 Développement d'une solution récursive

L'idée de base nous conduit à un algorithme récursif très simple :

```
procedure tri_rapide_r(var liste:TListbox;g,d:integer);
var i:integer;
begin
  if d>g then
    begin
      i:=division(liste,g,d);
      tri_rapide_r(liste,g,i-1);
      tri_rapide_r(liste,i+1,d);
    end;
end;
```

Les paramètres `g` et `d` limitent la partie de la liste qui doit être triée. Le premier appel de la procédure de tri a comme paramètres l'indice du premier et l'indice du dernier élément de la liste.

10.3.3.4 Division de la liste

Pour que cette procédure récursive fonctionne correctement il faut que la fonction division remplisse trois conditions :

1. L'élément avec l'indice i (indice qui est retourné comme résultat) se trouve à l'endroit définitif.
2. Tous les éléments à gauche de l'élément avec l'indice i sont plus petits ou égaux à celui-ci.
3. Tous les éléments à droite de l'élément avec l'indice i sont plus grands ou égaux à celui-ci.

Ces trois conditions nous amènent à une situation très agréable :

La première nous dit que l'élément avec l'indice i n'a plus besoin d'être déplacé.

La deuxième implique qu'aucun élément de la sous-liste gauche ne sera déplacé au-delà de l'élément avec l'indice i.

La troisième implique qu'aucun élément de la sous-liste droite ne sera déplacé avant l'élément avec l'indice i.

En plus on peut affirmer que les deux sous-listes obtenues de cette manière-ci peuvent être triées par la suite d'une manière indépendante.

La fonction division se présente de la manière suivante :

```
function division(var liste: TListbox;g,d:integer):integer;
var i,j : integer;
    candidat :string;
begin
  candidat:=liste.Items[d];
  j:=d-1; i:=g;
  while i<=j do
   begin
    if liste.Items[i]< candidat then i:=i+1
    else if liste.Items[j]> candidat then j:=j-1
    else begin echange(liste,i,j); i:=i+1 ; j:=j-1; end;
   end;
  echange(liste,i,d);
  result:=i;
end;
```

Pour commencer, on choisit l'élément qui se trouve à la fin de liste comme candidat. On va déterminer la position définitive du candidat dans la liste et s'arranger que tous les éléments de la liste qui se trouvent avant sont plus petits et tous ceux qui se trouvent après sont plus grands que le candidat.

La boucle parcourt la liste en commençant par le début (`if`) jusqu'au premier élément supérieur au candidat, ensuite (`else if`) elle parcourt la liste en commençant par la fin jusqu'au premier élément inférieur au candidat. Ces 2 éléments sont alors (`else`) échangés et on recommence.

A la fin de la boucle le candidat est mis à sa place définitive et l'indice de cette place est rendu comme résultat de la fonction. On vérifie facilement qu'au cas où le candidat se trouvait à sa bonne place (lorsqu'il est le plus grand élément de la liste), alors i=d et le dernier échange n'a aucun effet.

10.3.4 Tri par fusion

Attention : **Cet algorithme ne figure pas au programme.**

10.3.4.1 Idée

Si l'on dispose de deux listes triées, on peut assez facilement les fusionner[10] (mettre ensemble) afin d'obtenir une seule liste triée. Un tri par fusion utilisant cette approche peut être implémenté de façon récursive et de façon itérative. Les deux versions utilisent la même procédure `fusion`.

10.3.4.2 Fusion de deux liste triées

L'algorithme de fusion utilise un tableau auxiliaire de chaînes de caractères, de même taille que la liste à trier. Ce tableau peut être déclaré globalement ; cela évite de redéclarer le tableau à chaque appel de la procédure.

Les deux sous-listes à fusionner se trouvent de façon consécutive dans la liste. Les paramètres g et m sont les indices de début et de fin de la première sous-liste, alors que la seconde sous-liste commence à l'indice m+1 et se termine en d. Les deux sous-listes sont d'abord copiées dans le tableau auxiliaire : la 1re de façon croissante et la 2e de façon décroissante de sorte que les éléments les plus grands se trouvent au milieu. Ensuite la procédure commence à comparer les plus petits

[10] La fusion est en quelque sorte une généralisation de l'insertion où l'on fusionne aussi deux listes triées, dont l'une n'est composée que d'un seul élément.

éléments des deux-sous-listes (le plus à gauche et le plus à droite) et replace le plus petit des deux dans la liste principale et ainsi de suite.

```
procedure fusion(var liste: TListbox; g,m,d: integer);
var i,j,k : integer;
  tt : array[1..1000] of string; {mieux: déclarer tt globalement}
begin
  for i:=g to m do tt[i]:=liste.Items[i];
  for j:=m+1 to d do tt[d+m+1-j]:=liste.Items[j];
  i:=g;   j:=d;
  for k:=g to d do
   if tt[i]<tt[j]
    then begin liste.Items[k]:=tt[i]; i:=i+1 end
    else begin liste.Items[k]:=tt[j]; j:=j-1 end;
end;
```

10.3.4.3 Solution récursive

Il suffit de diviser la liste en deux, de trier séparément les deux parties et de fusionner les deux parties triées.

```
procedure tri_fusion_re(var liste: TListbox;g,d:integer);
var m:integer;
begin
 if g<d then
  begin
    m:=(g+d) div 2;
    tri_fusion_re(liste,g,m);
    tri_fusion_re(liste,m+1,d);
    fusion(liste,g,m,d);
  end;
end;
```

10.3.4.4 Solution itérative

La version itérative commence par considérer des sous-listes de longueur 1 qui sont ensuite fusionnées 2 à 2. Ensuite la liste n'est composée que de sous-listes de longueur 2 déjà triées. Celles-ci sont encore fusionnées 2 à 2 et ainsi de suite. D'étape en étape la longueur des sous-listes triées double (variable `step`). Toute la liste est triée lorsqu'il n'existe plus qu'une seule grande sous-liste.

```
procedure tri_fusion_it(var liste: TListbox);
var i,m,step:integer;
begin
 m:=liste.Items.Count;
 step:=1;
 i:=0;
 while step<m do
 begin
   while (i+2*step-1)<m do
   begin
     fusion(liste,i,i+step-1,i+2*step-1) ;
     i:=i+2*step;
   end;
   if (i+step)<=m then fusion(liste,i,i+step-1,m-1);
   {s'il reste une liste et une partie d'une 2e liste}
```

```
    step:=step*2;
    i:=0 ;
  end;
end;
```

10.4 Les algorithmes de recherche

Les algorithmes de recherche ont pour but de déterminer l'indice d'un élément d'une liste qui répond à un certain critère. Si l'on ne trouve pas l'élément, on retourne par convention –1. L'élément recherché, encore appelé *clé*, peut figurer plusieurs fois dans la liste. On suppose que la liste n'est pas vide.

10.4.1 Recherche séquentielle

10.4.1.1 Idée

Il s'agit de l'algorithme de recherche le plus simple qui puisse exister : on commence à examiner la liste dès le début jusqu'à ce qu'on ait trouvé la clé. L'algorithme donne l'indice de la première occurrence de la clé. La liste ne doit pas être triée.

10.4.1.2 Solution itérative

```
function recherche_seq_i(liste:TListbox;cle:string):integer;
var
   index:integer;
begin
  index:=0;
  while (index <= liste.Items.Count-1) and (liste.Items[index]<>cle) do
    index:=index+1;
  if index <= liste.Items.Count-1 then result:=index
  else result:=-1;
end;
```

10.4.2 Recherche dichotomique

10.4.2.1 Idée

La recherche dichotomique est très rapide et utilise une méthode bien connue : Si par exemple, vous devez chercher une localité dans l'annuaire téléphonique, vous ne commencez pas à la première page pour ensuite consulter une page après l'autre, mais vous ouvrez l'annuaire au milieu pour ensuite regarder si la localité se situe dans la première partie ou dans la deuxième partie de l'annuaire. Ensuite vous recommencez l'opération, c'est-à-dire vous divisez de nouveau la partie contenant la localité recherchée en deux et ainsi de suite jusqu'à ce que vous ayez trouvé la localité. L'algorithme de la recherche dichotomique utilise le même procédé :

On divise la liste en deux parties. On regarde si la clé correspond à l'élément au milieu de la liste. Si c'est le cas on a terminé, sinon on détermine la partie qui contient la clé et on recommence la recherche dans la partie contenant la clé. La liste ne doit pas forcément contenir la clé, mais elle doit être triée.

10.4.2.2 Solution récursive

```
function dicho_r(liste:TListbox;cle:string;g,d:integer):integer;
var
  milieu:integer;
begin
  if g>d then result:=-1
  else
    begin
      milieu:=(g+d) div 2;
      if liste.Items[milieu] = cle then result:=milieu
      else if cle<liste.Items[milieu] then
        result:= dicho_r(liste,cle,g,milieu-1)
      else
        result:= dicho_r(liste,cle,milieu+1,d);
    end;
end;
```

10.4.2.3 Solution itérative

```
function dicho_i(liste:TListbox;cle:string):integer;
var
  milieu,g,d:integer;
begin
  g:=0;
  d:=liste.Items.Count-1;
  while (cle<>liste.Items[milieu]) and (g<=d) do
  begin
    milieu:=(g+d) div 2;
    if cle<liste.Items[milieu] then d:=milieu-1
    else g:=milieu+1;
  end;
  if cle=liste.Items[milieu] then result:=milieu
  else result:=-1;
end;
```

MCours.com