Lycée Saint-Louis

Classes préparatoires
1ère année

Introduction au

Turbo Pascal

Jean-Philippe Poli
jean-philippe.poli@laposte.net

2008 – 2009

Table des matières

Chapitre 1

Introduction

Dans ce chapitre, nous allons introduire différentes notions d'informatique qui permettront de comprendre ce que nous allons faire tout au long de l'année.

1 Algorithmes et programmes

Les notions d'algorithme et de programme sont des notions liées mais qu'il ne faut pas confondre.

On appelle algorithme une suite d'opérations élémentaires permettant de résoudre un problème ou d'effectuer un calcul et qui peut être écrite dans un langage proche de notre langue naturelle, c'est-à-dire le français.

> **Exemple** ▷ *L'algorithme suivant est une méthode qui permet d'afficher successivement les nombres de 1 à 5.*
>
> ```
> Pour i allant de 1 à 5 faire
>     écrire i
> ```

Un programme est une suite d'instructions écrites dans un langage appelé « langage machine », constitué uniquement de 0 et de 1, et qui est le seul langage compris par un ordinateur. Par exemple, les traitements de texte et les messageries instantanées sont des programmes.

Le rôle d'un programmeur est donc d'écrire une suite d'instructions que l'ordinateur devra exécuter lorsqu'un utilisateur démarre un programme. Il est très difficile, pour des raisons évidentes, d'écrire un programme en langage machine. Le programmeur utilise donc un langage appelé « langage de programmation » qui est un intermédiaire entre la langue naturelle et le langage machine.

2 Compilation

Ainsi, le programmeur écrit dans un fichier (parfois appelé « code source ») l'ensemble des instructions en utilisant le langage de programmation. Puisque l'ordinateur ne comprend pas ces instructions, il faut les traduire en langage machine ; cette phase de traduction est appelée « compilation » (voir figure 1) et est effectuée par un logiciel. Après la compilation, on obtient un programme.

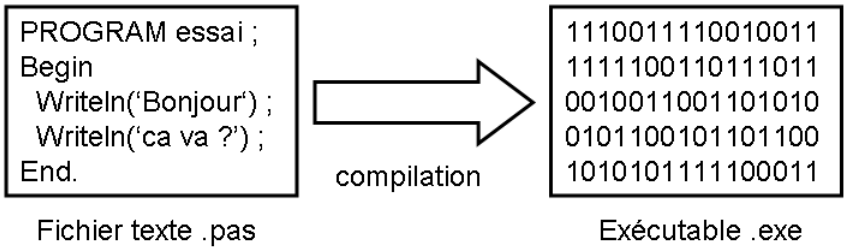


FIG. 1 – La compilation : transformation du code source en un programme

Pour écrire un programme dans un langage donné, nous utilisons un environnement de programmation. Il s'agit d'un ensemble de logiciels destinés au programmeur et comprenant : un éditeur de texte, un compilateur, un système d'aide, un système de débogage ...

3 Syntaxe et sémantique

Tout langage, que ce soit une langue naturelle comme le français ou un langage de programmation comme `Pascal`, possède deux aspects complémentaires que nous allons tenter de décrire brièvement : la syntaxe et la sémantique.

La syntaxe d'un langage est définie par un ensemble de règles qui régissent la construction des phrases et de leurs constituants. Pour une langue naturelle, on parle plutôt de grammaire et d'orthographe. Pour un langage de programmation, il s'agit des mots que l'on peut utiliser et comment ils peuvent être agencés entre eux. La syntaxe d'un langage ne demande aucune compréhension mais un apprentissage par cœur ; nous verrons dans le chapitre 3 l'ensemble des règles de syntaxe du langage `Pascal`.

Lorsque une phrase ou un programme est syntaxiquement correct, on peut alors parler de sémantique. La sémantique d'une langue naturelle correspond au sens que l'on donne aux mots et aux phrases tandis que la sémantique d'un programme définit à quel moment l'exécution de celui-ci est correcte (c'est-à-dire que le programme fait bien ce que l'on voulait).

> **Exemple** ▷ *Afin de bien comprendre la différence entre la syntaxe et la sémantique, prenons l'exemple de la langue française.*
> *La phrase* `souris chat manger` *est syntaxiquement incorrecte puisqu'elle ne respecte pas les règles de grammaires.*
> *A présent, si l'on considère* `la souris mange le chat`*, la phrase est syntaxiquement correcte, puisque les règles de grammaires sont respectées. En revanche, elle n'est pas sémantiquement correcte car elle n'a pas le sens qu'on voulait lui donner (à savoir que le chat mange la souris). Enfin, la phrase* `le chat mange la souris` *est syntaxiquement et sémantiquement correcte.*

En informatique, un programme qui n'est pas syntaxiquement juste ne va pas être compilé par le compilateur ; ce dernier va avertir le programmeur qu'il y a une erreur. En revanche le compilateur ne peut pas avertir le programmeur d'une erreur de sémantique : c'est à l'exécution que l'utilisateur s'en apercevra (c'est ce que l'on appelle un bug).

4 Le langage *Pascal*

Le langage `Pascal` a initialement été créé par Nicklaus Wirth, un professeur suisse, à la fin des années soixante-dix.

Lorsque `Pascal` a été conçu, il existait plusieurs langages de programmation, mais quelques-uns seulement étaient d'un usage répandu, tels FORTRAN, C, Assembleur, COBOL. L'idée maîtresse du nouveau langage était la structuration, organisée à l'aide d'un concept solide de type de données et exigeant des déclarations et des contrôles de programmes structurés. Ce langage devait par ailleurs être un outil d'enseignement pour les cours de programmation.

Turbo `Pascal`, le compilateur de Borland mondialement connu, a été introduit en 1985. Il a été l'un des compilateurs les plus vendus de tous les temps et a rendu le langage particulièrement populaire grâce à son équilibre entre simplicité et puissance.

Turbo `Pascal` a introduit un Environnement de Développement Intégré (EDI) dans lequel on pouvait éditer du code, faire tourner le compilateur, visualiser les erreurs et revenir aux lignes contenant ces erreurs. Considérations triviales à l'heure actuelle, mais auparavant il fallait quitter l'éditeur, revenir au DOS, actionner la ligne de commande du compilateur, noter les lignes erronées, ouvrir l'éditeur et sauter jusque là.

En 1995, après neuf versions de compilateurs Turbo et Borland `Pascal`, qui ont progressivement étendu le langage, Borland a lancé Delphi, faisant de `Pascal` un langage de programmation visuel.

Chapitre 2

Utiliser l'environnement de programmation

Dans ce chapitre, nous allons étudier l'environnement de développement, c'est-à-dire le logiciel qui nous permet d'écrire des programmes en `Pascal`.

1 Description des menus

L'environnement de programmation que nous utilisons apparaît dans une fenêtre en haut de laquelle se trouve la barre des menus (voir figure 2) qui permet d'accéder aux fonctionnalités du logiciel.

FIG. 2 – La barre des menus

Nous allons à présent passer en revue les fonctionnalités dont vous aurez besoin durant les travaux pratiques.

1.1 Menu *File*

Le menu *File* permet d'accéder aux fonctionnalités suivantes :
- **New** : ouvre une nouvelle fenêtre d'édition (dans la quelle le code source doit être écrit).
- **Open** : ouvre un fichier existant.
- **Save** : enregistre le contenu de la fenêtre dans un fichier. Lors de la première sauvegarde, un nom de fichier est demandé. Les sauvegardes suivantes s'effectuent dans le même fichier.
- **Save as** : enregistre le contenu de la fenêtre dans un nouveau fichier.
- **Exit** : quitte le logiciel.

1.2 Menu *Edit*

Le menu *Edit* permet d'accéder aux fonctionnalités concernant l'édition du texte :
- **Undo** : annule la dernière action.
- **Cut** : supprime le texte sélectionné et le copie en mémoire.
- **Copy** : copie le texte sélectionné en mémoire sans le supprimer.
- **Paste** : colle le texte mémorisé à l'emplacement actuel du curseur.

1.3 Menu *Run*

Dans ce menu, nous n'utiliserons que l'option **Run** qui permet d'exécuter le programme correspondant à votre code source. Si votre code source n'est pas compilé, la commande **Run** exécutera préalablement une compilation. Il est toutefois préférable de lancer la compilation manuellement.

1.4 Menu *Compile*

Dans ce menu, l'option **Compile** compile le code source actuel. La compilation crée alors un programme exécutable à partir du menu **Run**.

1.5 Menu *Debug*

Dans ce menu, l'option **User screen** affiche l'écran de l'exécution du programme.

2 Editeur de texte

La figure 3 représente une fenêtre de l'éditeur de texte. Plusieurs fenêtres peuvent être ouvertes simultanément dans l'environnement de développement.

L'éditeur de texte connaît le langage `Pascal` et changera la couleur des mots en fonction de leur rôle dans le code : les mots clés, les commentaires, les chaînes de caractères se distingueront par leur couleur.

3 Utilisation de l'environnement de programmation

Une fois l'environnement de programmation démarré, il est d'usage de créer un nouveau fichier qui contiendra votre code source. Il ne peut y avoir qu'un seul programme par fichier. Sauvegardez le plus rapidement possible le nouveau fichier afin de lui donner un nom. Ensuite, il est recommandé de sauvegarder assez souvent de façon à conserver les modifications que vous apportez à votre programme.

Lorsque vous avez terminé votre programme, vous pouvez en demander la compilation ; la compilation provoque obligatoirement une sauvegarde au préalable. Vous pouvez ensuite exécuter le pro-

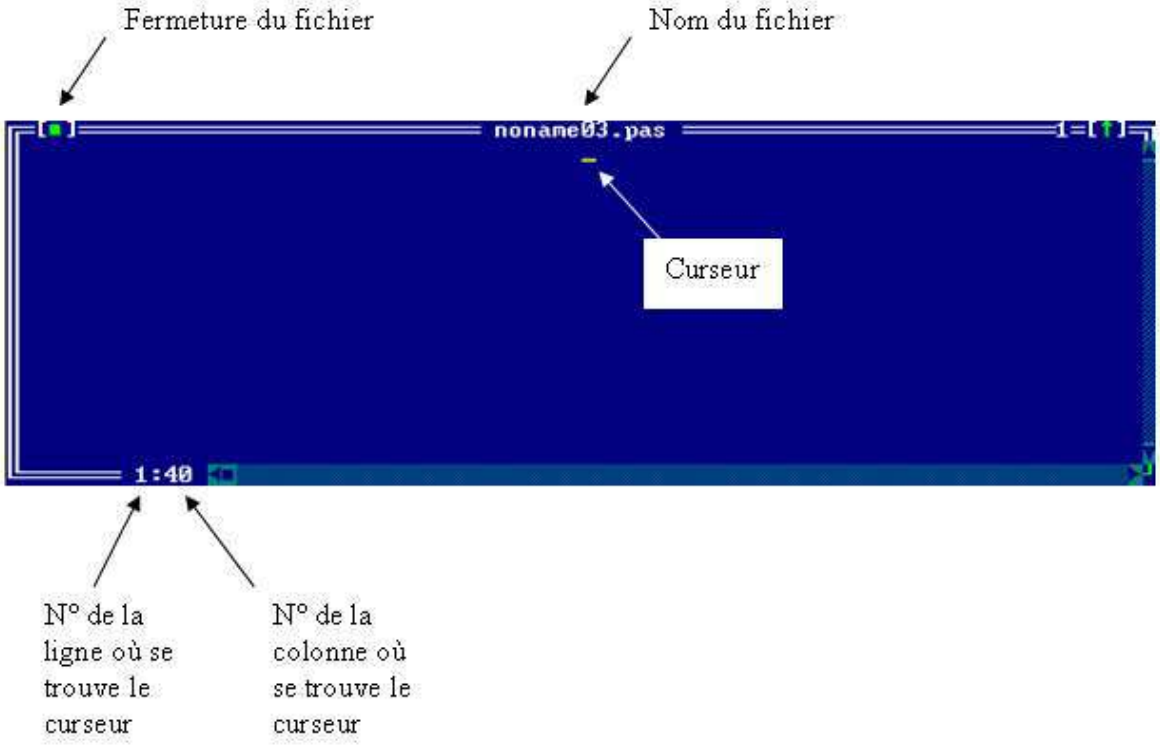


FIG. 3 – L'éditeur de texte de l'environnement de programmation

gramme. Celui-ci s'exécute dans une fenêtre indépendante de l'environnement de programmation. Il ne faut jamais la fermer : si elle ne se ferme pas seule, c'est que votre programme présente un bug.

► Pour gérer vos fichiers et vos programmes, il vous est recommandé d'observer les conseils suivants :

- évitez les accents et les caractères spéciaux dans les noms de fichiers : l'environnement gère mal ces caractères ;
- donnez des noms évocateurs à vos fichiers pour les trouver facilement, et de préférence un nom de la forme `TP1exo5` qui facilite une recherche ultérieure ;
- tout nom de fichier contenant des instructions en `Pascal` doit se terminer par l'extension « .pas » ce qui permet à l'environnement de changer la couleur des mots en fonction de leur rôle ;
- sauvegardez vos fichiers dans votre répertoire personnel (créé en début d'année) ;
- n'ouvrez jamais plusieurs fois le même fichier dans l'environnement ou plusieurs environnements : cela conduit très souvent à la perte des modifications apportées à vos fichiers.

Chapitre 3

Programmer en *Pascal*

A présent, nous allons décrire l'ensemble des règles qui permettent d'écrire un programme en langage `Pascal`. Les mots-clés du langage `Pascal` sont inspirés de mots anglais.

1 Structure générale d'un programme

```
program identificateur;

DECLARATIONS

begin
  instruction_1;
  instruction_2;
  ⋮
  instruction_n;
end.
```

Le code ci-dessus montre la structure générale d'un programme écrit en `Pascal`. Tout programme doit commencer par le mot-clé **program** suivi du nom que vous voulez donner à celui-ci : il est impératif que ce soit la première ligne de votre fichier. Ce nom peut être différent de celui du fichier qui contient votre code et doit être écrit en respectant les règles des identificateurs (voir page 12). Enfin, cette première ligne est terminée par un point-virgule.

Exemple ▷ ***program*** *essai;*

Dans la partie appelée « déclarations », tous les objets (variables, constantes, fonctions, procédures, etc ...) qui seront utilisés dans le programme doivent être décrits.

Enfin, tout programme doit contenir un **begin** et un **end**. entre lesquels seront placés les instructions. Ce bloc est appelé « programme principal » et le **end**. marque la fin de celui-ci (avec le point final).

Après la compilation, l'exécution du programme consistera à exécuter de façon séquentielle chacune des instructions du programme principal jusqu'à atteindre le **end**. final.

- Il ne peut y avoir qu'un seul programme par fichier, donc une seule fois le mot-clé **program**
- Rien ne doit être écrit au-dessous de **end**.

Exemple ▷ *voici un petit programme qui demande à son utilisateur de rentrer un nombre qu'il affiche par la suite.*

```
program facile;

var i : integer ;

begin
  writeln('Entrez un chiffre');
  readln(i) ;
  writeln('Vous avez entré :',i);
end.
```

2 Symboles et séparateurs

Un programme `Pascal` est constitué de symboles et de séparateurs. Les séparateurs sont les blancs (un ou plusieurs espaces, tabulation), les sauts de ligne, et les commentaires.

Les commentaires sont des textes compréhensibles par l'homme et qui ne seront pas lus par la machine. Pour indiquer qu'un texte est en commentaire, il faut le placer entre des accolades { }, ou alors entre (* *).

Exemple ▷ *le code suivant montre les différentes façons d'écrire un commentaire en* `Pascal` *:*

```
{ceci est un commentaire}

(*ça aussi*)

{ceci est un commentaire
sur plusieurs lignes}

(*celui-ci
aussi*)
```

Il existe différents types de symboles : les symboles spéciaux, les mots-clés, les identificateurs, les nombres, les chaînes de caractères ...

Voici l'ensemble des symboles spéciaux du langage `Pascal` que nous allons utiliser (nous verrons leurs sens plus tard) :

```
+   -   *   /   ,   :   ;   =   <>   <=   >=   :=   ..   ( )   [ ]
```

De même, les mots-clés du langage `Pascal` (au programme) sont :

and	**array**	**begin**	**const**	**div**
do	**downto**	**else**	**end**	**for**
function	**if**	**mod**	**not**	**of**
or	**procedure**	**program**	**repeat**	**then**
to	**type**	**until**	**var**	**while**

- Au moins un séparateur doit figurer entre deux identificateurs, mots-clés, ou nombre
- Aucun séparateur ne doit figurer dans un symbole Pascal

Exemple ▷ *Voici deux exemples pour illustrer les règles ci-dessus :*

- *beginX constitue un seul symbole qui est un identificateur, tandis que* ***begin*** *X constitue deux symboles à savoir le mot-clé* ***begin*** *et l'identificateur X*
- *<= n'a pas le même sens que < = qui est syntaxiquement faux en* `Pascal`

Les règles assez souples concernant le nombre de séparateurs entre deux mots-clés ou deux identificateurs permettent de présenter correctement un programme. N'hésitez pas à aérer vos instructions et à les indenter (i.e. les décaler vers la droite) pour que votre code source soit clair.

Exemple ▷ *Les deux programmes ci-dessous sont identiques pour l'ordinateur, mais vous conviendrez que le premier est beaucoup plus lisible que le second pour un humain :*

```
program essai;
var age:integer;

begin
  writeln('votre age ?');
  readln(age);
  writeln('vous avez ',age,' ans.');
end.
```

```
program essai;var age:integer;
begin writeln('votre age ?');readln(age);
writeln('vous avez ',age,' ans.');end.
```

3 Identificateurs

Les identificateurs servent à donner des noms à des objets de façon à les distinguer. Les règles à observer pour les identificateurs sont les suivantes :

- un identificateur est composé de lettres (non accentuées), de chiffres et du caractère _ (appelé underscore), mais doit obligatoirement commencer par une lettre ;
- la longueur minimale d'un identificateur est 1 caractère, la longueur maximale est 255. Cela permet de donner à des variables ou à un programme un nom qui a du sens, mais attention car manipuler des identificateurs trop longs peut être pénible.

- ▶ En `Pascal`, il n'y a pas de différence entre les majuscules et les minuscules, c'est-à-dire que A et a désignent le même objet
- ▶ Aucun identificateur ne doit avoir la même orthographe qu'un mot-clé ou un identificateur pré-déclaré
- ▶ Aucun identificateur ne peut être le même que celui du programme

Exemple ▷ *Exo_TP1_1, TP1_1, serie_d_euler sont des identificateurs corrects, alors que Série d'euler, suite_réelle, 3a ne le sont pas.*

La liste des identificateurs pré-déclarés (c'est-à-dire qui ont déjà un sens en `Pascal`) que nous allons utiliser, et dont nous verrons le sens après, est la suivante :

abs	**arctan**	**boolean**	**cos**	**exp**
false	**integer**	**ln**	**read**	**readln**
real	**round**	**sin**	**sqr**	**sqrt**
true	**trunc**	**write**	**writeln**	

4 Types et constantes de base

Un type décrit un ensemble de valeurs pouvant être manipulées par un ensemble d'opérateurs. Nous allons étudier quelques types simples, mais nous verrons plus tard des types dits structurés (voir page 28).

4.1 Nombres entiers

Les valeurs de type **integer** sont un sous-ensemble des entiers relatifs, c'est-à-dire les entiers de l'intervalle [-32768 ; 32767]. Si on dépasse les valeurs de cet intervalle, il y aura une erreur à l'exécution (parfois même à la compilation).

Exemple ▷ *Voici des exemples corrects d'entiers : 0 1 32000 -56.*

Soit x et y deux **integer**, les opérateurs sur les entiers sont :

+x	Opérateur unaire d'identité
−x	Opérateur unaire pour le signe opposé
x+y	Addition entière
x−y	Soustraction entière
x∗y	Multiplication entière
x/y	Division **réelle**.
	Si y est nul, il y aura une erreur à l'exécution.
	Le résultat de x/y est le quotient **réel** de la division de x par y
x **div** y	Division entière.
	Si y est nul, il y aura une erreur à l'exécution.
	Le résultat de x **div** y est la partie entière du quotient de la division de x par y
x **mod** y	Modulo.
	Si y est négatif ou nul, il y a une erreur à l'exécution.
	Le résultat est le reste de la division entière de x par y

et les fonctions prédéfinies sur les entiers sont :

abs(x)	Calcule la valeur absolue $\|x\|$ de x
sqr(x)	Calcule x^2

On remarque donc que le résultat d'une opération sur deux **integer** est toujours un **integer** sauf dans le cas de la division réelle.

4.2 Nombres réels

Il s'agit du type **real** couvrant les valeurs de l'intervalle [2.9E-39 ; 1.7E38] aussi bien dans les réels négatifs que positifs et dont la précision peut atteindre 11 à 12 chiffres significatifs. Il faut remarquer qu'il n'y a pas d'espace entres les chiffres, et surtout que le séparateur décimal, qui en français est la virgule, est ici le point.

Exemple ▷ *Voici des exemples corrects de réels : 0.0 1E0 -1.5E4 .*

Soit x et y deux réels, voici les opérateurs du type **real** :

+x	Identité
−y	Opposé
x+y	Addition réelle
x−y	Soustraction réelle
x∗y	Multiplication réelle
x/y	Division réelle. Si y est nul, il y a une erreur à l'exécution

Le résultat de ces opérations est toujours un **real**.

Il existe aussi des fonctions prédéfinies prenant en paramètre un **real** :

abs(x)	Calcule la valeur absolue $\|x\|$ de x
sqr(x)	Calcule x^2
sin(x)	Sinus de x, avec x en radians
cos(x)	Cosinus de x, avec x en radians
exp(x)	e^x
ln(x)	Logarithme népérien de x. Si x n'est pas strictement positif, il y aura une erreur à l'exécution
sqrt(x)	Racine carrée de x. Si x est strictement négatif, il y aura une erreur à l'exécution
arctan(x)	Valeur principale en radians de l'arc tangente de x exprimé en radians

Le résultat de ces fonctions est toujours un **real**.

4.3 Booléens

Il s'agit du type **boolean** qui a deux valeurs possibles notées **true** et **false** et qui est utilisé pour représenter les valeurs logiques « vrai » et « faux ».

Trois opérateurs permettent de manipuler des booléens : la négation (**not**), la conjonction (**and**) et la disjonction (**or**). A titre de rappel, voici les tables de vérités de ces opérateurs :

X	Y	not X	X and Y	X or Y
true	true	false	true	true
true	false	false	false	true
false	true	true	false	true
false	false	true	false	false

Des opérateurs appelés « opérateurs de comparaison » permettent de comparer deux valeurs entières ou réelles tout en renvoyant un résultat booléen :

=	est égal à
<	est inférieur à
>	est supérieur à
<=	est inférieur ou égal à
>=	est supérieur ou égal à
<>	est différent de

Exemple ▷ *La valeur de 1<=2 est* ***true*** *et la valeur de 10<>5+5 est* ***false****.*

► Il ne faut surtout pas confondre l'opérateur de comparaison = et l'opérateur d'affectation := (voir page 20) !

4.4 Chaînes de caractères

Nous utiliserons les chaînes de caractères pour afficher un résultat ; nous n'irons donc pas plus loin dans la définition de ce type. Une chaîne de caractères représente un mot ou une phrase qui n'est pas interprétée par la machine. Une chaîne de caractères est délimitée par les cotes, c'est-à-dire le caractère apostrophe.

Exemple ▷ *Voici quelques chaînes de caractères :*

'bonjour' 'il fait beau' 'un été ensoleillé' .

- ▶ Puisque l'ordinateur reproduit les chaînes de caractères sans les interpréter, il est possible d'écrire en français et d'utiliser les accents ; cependant, les accents s'affichent généralement mal
- ▶ Si dans une chaîne de caractères on veut utiliser l'apostrophe, il faut le doubler : 'l''oiseau'

4.5 Passage d'un type à l'autre

Il est impossible de passer du type **boolean** à un autre type. En revanche il est possible de passer du type **integer** à **real** et vice-versa sous certaines conditions.

Avant d'aller plus loin il est important de comprendre qu'à cause de la façon de coder les entiers et les réels dans la mémoire d'un ordinateur, les entiers sont catégoriquement différents des réels. Ainsi, même si en mathématiques on écrit indifféremment 6 et 6.0, en informatique ce ne sont pas du tout les mêmes nombres : en informatique, les **integer** ne sont pas inclus dans les **real** alors que les entiers sont inclus dans les réels.

Pour remédier à cela, on peut être amené à vouloir transformer un entier en réel ou l'inverse. La conversion des **integer** en **real** est automatique.

- ▶ Cela va avoir une implication assez importante. Nous avons vu précédemment qu'un opérateur comme le + nécessitait soit deux entiers, soit deux réels. Grâce à la capacité de conversion automatique d'un entier en réel, il sera possible d'utiliser tous les opérateurs et toutes les fonctions des **real** avec des **integer** et le résultat sera toujours un **real**.

  Exemple ▷ *On peut considérer les opérations suivantes :*
  - *1+3.0 a pour résultat 4.0 de type* ***real*** *;*
  - ***sqrt****(2) a pour résultat 2.0 de type* ***real****.*

En revanche, la conversion des **real** en **integer** ne peut pas se faire seule, comme en mathématiques d'ailleurs. En effet convertir 6.6 en entier nous laisse le choix entre deux valeurs : 6 ou 7. Il faut donc utiliser l'une des deux fonctions suivantes, où x est un **real** et le résultat est un **integer** :

trunc(x)	Partie entière de x
round(x)	Entier le plus proche de x.

Exemple ▷ *Ces deux fonctions peuvent donner les résultats suivants :*
- *avec* ***trunc**(6.6) on obtiendra 6 (de type* ***integer**)*
- *avec* ***round**(6.6) on obtiendra 7 (de type* ***integer**)*
- *avec* ***round**(6.3) on obtiendra 6 (de type* ***integer**)*

5 Variables

Le mot « variable » n'a pas le même sens en informatique et en mathématiques. En informatique, une variable est un objet qui permet de contenir une valeur : c'est un emplacement dans la mémoire de l'ordinateur accessible par un nom (son identificateur, voir page 12) et dont la taille dépend du type (voir page 12) de la valeur qu'il contient.

La différence majeure entre le sens informatique et le sens mathématique d'une variable réside surtout dans la manière d'affecter une valeur à une variable. En mathématiques, une variable peut :
- représenter n'importe quelle valeur d'un ensemble, lorsqu'on écrit par exemple « $\forall x \in \mathbb{R}$ » ;
- avoir une valeur définie explicitement, lorsqu'on écrit « pour $x = 2$ » ;
- avoir une valeur définie implicitement, c'est-à-dire que la valeur de la variable n'est pas calculée directement mais tous les éléments nécessaires à son calcul sont fournis, comme dans « $2x = 6$ ».

En informatique, une variable ne peut être définie que de façon explicite par une instruction que l'on appelle « affectation » (voir page 20). On ne pourra jamais demander à l'ordinateur de résoudre l'équation $2x = 6$ à notre place : il faudra lui spécifier que x vaut $\frac{6}{2}$.

Toute variable utilisée dans un programme doit faire l'objet d'une déclaration, c'est-à-dire que l'on va nommer toutes les variables dont on va avoir besoin et leur donner un type à l'aide du mot-clé **var** :

```
var liste_de_variables1 : type1 ;
var liste_de_variables2 : type2 ;
    ⋮
var liste_de_variablesn : typen ;
```

Une liste de variables consiste en fait à une liste d'identificateurs séparés par des virgules. Les différents $type_i$ sont les types simples **real**, **integer**, **boolean** (voir page 12) et les types structurés ou définis que nous verrons plus tard (voir page 28). Il peut y avoir autant de listes de variables à déclarer que nécessaire mais en principe il n'y en a qu'une par type.

▶ Le mot clé **var** n'a pas besoin d'être répété au début de chaque ligne de déclaration. On peut très bien écrire :

```
var liste_de_variables_1 : type_1 ;
    liste_de_variables_2 : type_2 ;
    ⋮
    liste_de_variables_n : type_n ;
```

▶ Il est conseillé de donner des noms évocateurs à vos variables : si une variable doit représenter le poids d'un colis par exemple, il est toujours préférable de l'appeler p plutôt que x, et d'autant plus préférable de l'appeler directement *poids*.

Exemple ▷ *Nous allons déclarer trois variables a, b et c de type* ***integer****, une variable taille de type* ***real*** *et deux conditions, condition*1 *et condition*2*, de type* ***boolean*** *:*

```
var a,b,c:integer;
    taille:real;
    condition1,condition2:boolean;
```

▶ La déclaration d'une variable réserve un emplacement en mémoire mais ne lui affecte pas de valeur par défaut. Par exemple, dans l'exemple précédent, a, b et c ont été déclarées comme des valeurs entières mais leur valeur à ce moment est indéterminée (elles ne valent pas 0 !). Dans de très nombreux cas, il faudra donc initialiser les variables au début du programme principal, c'est-à-dire leur donner une valeur initiale.

Exercice 1▶ *Déclarez les variables nécessaires pour représenter : le nombre de litres de lait, le nombre de cuillères à café de sucre en poudre, la quantité de beurre en grammes et la quantité de farine en kilogrammes qu'il est nécessaire de mélanger pour une recette de cuisine.*

Exercice 2▶ *Trouvez les 2 fautes de déclaration qui se sont glissées dans l'extrait de programme suivant :*

```
program calcul;
var abs,a,b,c:integer;
    calcul,B:real;

begin
...
end.
```

6 Expressions

Nous avons l'habitude, en mathématiques, de manipuler des expressions. Une expression est une formule composée de constantes, de variables, d'appels de fonctions et d'opérateurs. Une expression

a donc une valeur et un type. On dit qu'une expression est bien formée lorsque l'on peut calculer sa valeur et déterminer son type. Une expression mal formée ne permet pas au programme d'être compilé.

Exemple ▷ *Le tableau ci-dessous présente des exemples d'expressions bien et mal formées.*

Expressions bien formées	*Expressions mal formées*
1+10.0	*1+**true***
true and false	*1 / **false***
*1 **mod** 2*	*1 **div sqrt**(4)*

*S'il est nécessaire de donner une explication pour les expressions mal formées, on peut dire que la première tente d'additionner un entier et un booléen, la seconde tente d'effectuer une division par un booléen et la dernière tente d'utiliser l'opérateur entier **div** avec un réel puisque **sqrt**(4) est bien un réel.*

Pour déterminer si une expression est bien formée ou pas, il suffit d'appliquer les règles suivantes :

1. Une expression formée d'une seule constante est bien formée. Le type de l'expression est alors celui de la constante.

   Exemple ▷ *10.0 est une expression bien formée de type **real***

2. Une expression formée d'une seule variable est bien formée. Le type de l'expression est celui de la variable.

   Exemple ▷ *si on a **var** Z : **INTEGER**;, alors Z est une expression bien formée de type **integer***

3. Une expression fonctionnelle est bien formée si la fonction désignée est définie, si les sous-expressions arguments sont bien formées, et si le nombre et le type des sous-expressions arguments sont corrects. L'expression est du type de la fonction.

   Exemple ▷ ***sin**(3.1415) est une expression bien formée de type **real***

4. Une expression formée d'une sous-expression entre parenthèses ( ) est bien formée si la sous-expression l'est et son type est celui de la sous-expression.

   Exemple ▷ *si C est une expression bien formée de type **boolean**, alors (C) l'est aussi*

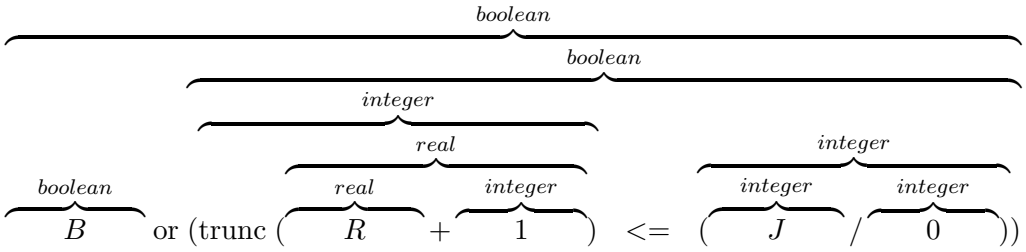
5. Une expression formée d'un opérateur unaire appliqué à une sous-expression est bien formée si la sous-expression l'est et si le type de la sous-expression est celui de l'opérande indiqué ci-après. Le type de l'expression est alors le type du résultat correspondant ci-dessous :

Opérateur	Type de l'opérande	Type du résultat
not	**boolean**	**boolean**
+ −	**integer**	**integer**
+ −	**real**	**real**

Exemple ▷ *-12.0 est une expression bien formée de type* ***real***

6. Une expression formée d'un opérateur binaire appliqué à deux sous-expressions est bien formée si les deux sous-expressions le sont et si le type des sous-expressions respecte les règles ci-après. Le type de l'expression est alors le type du résultat comme indiqué ci-dessous :
   - les opérandes de <, <=, =, >=, > sont de même type simple. Le résultat est de type **boolean** ;
   - les opérandes et le résultat de **or** et **and** sont de type **boolean** ;
   - si les opérandes de +, -, * sont de type **integer** alors le résultat est aussi **integer**. Si l'un des opérandes est de type **real** alors le résultat est **real** ;
   - l'opérateur / a des opérandes de type **integer** ou **real** et le résultat est de type **real** ;
   - les opérateurs **div** et **mod** ont des opérandes et un résultat de type **integer**.

Exemple ▷ *Soit B de type* ***boolean****, J de type* ***integer*** *et R de type* ***real****. Alors l'expression B* ***or*** *(****trunc****(R+1) <= (J/0)) est bien formée et de type* ***boolean****, même si elle est sémantiquement fausse à cause de la division par 0 :*



Exercice 3▶ *Déterminez si les expressions suivantes sont bien formées (si c'est le cas, déterminez leur valeur et leur type) :*

1 < 2 < 3	*1 + 2 − **cos**(30)*	*1 / 2 **div** 3*
***ln** (2 **div** 3)*	*(1.0 **div** 2)*	*(1 **mod** 2) / 5*

7 Instructions

Une instruction spécifie une opération ou un enchaînement d'opérations à exécuter sur des objets. Les instructions s'adressent toutes à l'ordinateur.

Chaque instruction doit se terminer par un point-virgule : rien d'autre ne peut séparer deux instructions.

▶ Lorsque plusieurs instructions doivent être exécutées, on a tendance à se dire « il faut exécuter l'instruction$_1$ et l'instruction$_2$ » ce que l'on a envie de traduire par instruction $_1$ **AND** instruction$_2$ ce qui est absolument faux.

Nous allons à présent passer en revue toutes les instructions qui sont au programme.

7.1 Affectation

L'affectation est peut-être l'instruction la plus importante puisqu'elle permet d'attribuer une valeur à une variable. Cette attribution d'une valeur se fait par le remplacement de la valeur précédente par la nouvelle valeur. L'instruction d'affectation se note :=. Si V est une variable d'un type quelconque et *expression* est une expression de même type, alors :

```
V:=expression;
```

Exemple ▷ *Nous allons illustrer ce qu'il se passe lorsqu'une affectation est effectuée. Une fois que la variable X de type* ***integer*** *est déclarée, elle ne contient pas de valeur. Nous procédons à une première affectation en lui attribuant la valeur 10. A présent, la variable contient 10. Puis, nous faisons une affectation un peu plus compliquée : on lui affecte sa valeur courante augmentée de 5. Une fois cette affectation effectuée, X contient 15.*

Avant	Affectation	Après
X ?	X:=10;	X 10
X 10	X:=X+5;	X 15

Exemple ▷ *Nous allons écrire un programme qui va utiliser une variable x de type* ***integer*** *et qui va calculer son carré dans une variable carre de type* ***integer*** *aussi :*

```
program affectation;

var x,carre:integer;

begin
  x:=5;
  carre:=sqr(x);
end.
```

7.2 Entrées-sorties

Un programme doit pouvoir communiquer avec son utilisateur :
- un programme peut informer l'utilisateur d'un résultat ou lui donner des explications en affichant un message ;
- un programme peut demander à l'utilisateur de saisir des valeurs pour pouvoir fonctionner.

Pour afficher un message à l'écran, il faut employer les instructions **write** et **writeln** :

```
write(expression_1,...,expression_n)
writeln(expression_1,...,expression_n)
```

Ces deux instructions affichent la valeur des expressions qui lui sont passées en paramètres. La différence entre **write** et **writeln** est que **writeln** va à la ligne après avoir affiché les expressions. Si une

expression est une chaîne de caractères, la chaîne va être reproduite à l'écran. Sinon, si l'expression est un calcul ou une variable, la valeur est affichée.

L'ordinateur peut aussi lire ce que l'utilisateur a saisi au clavier. Pour cela, il faut lui demander de lire ce qui a été entré et lui donner la variable dans laquelle la valeur va être stockée :

```
readln(variable)
```

En pratique, l'instruction **readln** affiche un curseur qui permet à l'utilisateur de taper du texte (dans notre cas, il ne peut s'agir que d'un nombre réel ou d'un nombre entier). Une fois que l'utilisateur a appuyé sur la touche « entrée », le texte est convertit en **real** ou **integer** puis est stocké dans la variable spécifiée.

Il peut s'avérer pratique de demander à l'utilisateur de rentrer plusieurs valeurs d'un coup. Pour cela, les deux formes suivantes sont équivalentes :

```
readln(variable_1,...,variable_n);
```

```
readln(variable_1);
⋮
readln(variable_n);
```

Nous verrons des exemples d'utilisation de **writeln** et **readln** dans les parties suivantes.

► Nous avons vu que lorsque l'ordinateur atteint le **end**. final du programme, celui-ci s'arrête. Parfois, cela ne laisse pas le temps de lire un résultat qu'on attendait. Il est possible de placer un **readln**; avant le **end**. final pour bloquer le programme jusqu'à ce que la touche « ENTREE » soit pressée.

Exercice 4► *Ecrivez les instructions nécessaires pour demander à l'utilisateur de saisir les quantités de beurre, de lait, de farine et de sucre pour une recette en utilisant les variables déclarées dans l'exercice 1 (page 17) et afficher le message La recette nécessite l'utilisation des quantités suivantes et affichez l'ensemble des ingrédients et leur quantité sur des lignes différentes.*

7.3 Branchement conditionnel

L'instruction de branchement conditionnel permet d'exécuter une instruction en fonction d'une certaine condition. Elle s'écrit :

```
if condition then instruction_1

if condition then instruction_1 else instruction_2
```

Dans l'encadré ci-dessus, *condition* est une expression booléenne, c'est-à-dire dont la valeur peut être soit **true**, soit **false** ; cette expression peut être aussi complexe que nécessaire. De plus, $instruction_1$

et $instruction_2$ représentent n'importe qu'elle instruction, y compris une nouvelle instruction **if then else**. Cette instruction peut se traduire par « si . . .alors . . .sinon . . . ».

La première forme de branchement conditionnel exécute $instruction_1$ si *condition* est vraie et la seconde forme exécute $instruction_1$ si *condition* est vraie et $instruction_2$ si *condition* est fausse. On remarque alors que le **else** est facultatif mais pas le **then**.

► Il n'y a **jamais de point-virgule avant le else**, même s'il est en fait précédé d'une instruction.

Exemple ▷ *Voici un programme qui demande son âge à l'utilisateur et lui dit s'il est majeur ou pas :*

```
program majorite;

var age:integer;

begin
  writeln('Quel age avez-vous?');
  readln(age);
  if age>18 then
    writeln('vous etes majeur(e)')
  else
    if age=18 then writeln('ouf! de justesse!')
        else writeln('vous etes trop jeune');
end.
```

Exercice 5► *Ecrivez la portion de programme qui correspond à l'algorithme suivant (en utilisant les variables déclarées dans l'exercice 1, page 17) :*

si la recette contient plus de 200g de beurre alors afficher "la recette est trop grasse" sinon afficher "la recette est bonne".

7.4 Blocs d'instructions

Nous venons de voir qu'une instruction peut être exécutée en fonction d'une condition. Mais que se passe-t-il si plusieurs instructions doivent être exécutées en fonction de la même condition ?

Dans ce cas, on groupe les instructions dans un bloc qui commence par **begin** et se termine par **end**; :

```
begin
  instruction_1;
  ⋮
  instruction_n;
end;
```

On comprend donc que le programme principal est un bloc d'instruction particulier puisqu'il est le seul à se terminer par un point plutôt que par un point virgule.

- Les blocs d'instructions ne doivent être utilisés qu'après un **then**, un **else** ou un **do**
- Si un bloc d'instruction est placé devant un **else**, il ne faut pas mettre de point-virgule après le **end** du bloc

Exemple ▷ *Ecrivons un programme qui affiche le produit de deux nombres. Si le premier nombre entré est 0, le résultat sera forcément 0. Dans le cas où le premier nombre n'est pas nul, le programme demande alors le deuxième nombre pour afficher le résultat de leur produit :*

```
program produit;
var a,b:integer;

begin
  writeln('Entrez un nombre');
  readln(a);
  if a=0 then
    writeln('le résultat est 0')
  else
    begin
      writeln('Entrez un nombre');
      readln(b);
      writeln('le résultat est ',a*b);
    end;
end.
```

7.5 Structures répétitives

Une instruction d'itération permet de répéter l'exécution d'une ou plusieurs instructions sous certaines conditions. Il existe trois types de boucles.

Boucle *for*

La boucle **for** est la boucle la plus simple. Elle permet de faire varier la valeur d'une variable d'une valeur à une autre en l'augmentant (ou la diminuant) de 1 à chaque étape. Cette variable, appelée « sentinelle », doit être, à notre niveau, de type **integer**. Si *variable* dénote une variable entière et si inf et *sup* sont deux expressions entières, la boucle **for** peut s'écrire :

```
for variable:=inf to sup do instruction
for variable:=sup downto inf do instruction
```

La boucle **for** se traduit par « pour *variable* allant de inf à sup répéter l'instruction ». Cette boucle ne peut donc être utilisée que lorsque l'on connaît a priori le nombre d'itérations (i.e. de passage dans la boucle). Pour qu'elle fonctionne correctement, c'est-à-dire qu'il y ait effectivement une répétition, il faut au moins que la condition $inf \leq sup$ soit vérifiée. Si $sup < inf$, aucune erreur n'est générée mais la boucle ne sera jamais exécutée.

Exemple ▷ *Voici un programme qui affiche la table de multiplication de 6 à l'envers :*

```
program multiplication;

var i:integer;

begin
  for i:=10 downto 1 do
    writeln('6 x ',i,' = ',6*i);
end.
```

▶ Il n'est pas obligatoire d'impliquer la sentinelle dans un calcul : elle peut ne servir que de compteur d'itérations sans intervenir dans un calcul.

Exemple ▷ *Pour illustrer cette astuce, nous allons faire un programme qui affiche 5 fois le mot « coucou » :*

```
program coucou5;

var i:integer;

begin
  for i:=1 to 5 do
    writeln('coucou');
end.
```

▶ Il est peu recommandé de modifier la valeur de la sentinelle à l'intérieur d'une boucle **for** car cela peut rendre le nombre d'itérations imprévisible. Il vaut mieux laisser la boucle **for** faire varier toute seule la sentinelle.

Exemple ▷ *Voici un programme dans lequel la boucle ne se termine jamais : chaque fois que la boucle augmente la sentinelle de 1, l'instruction à répéter la diminue de 1.*

```
program infini;

var i:integer;

begin
  for i:=1 to 5 do
    i:=i-1;
end.
```

Exercice 6▶ *Ecrire un programme qui demande à l'utilisateur de rentrer un entier $N > 0$, puis calculer et afficher la somme des N premiers nombres.*

Boucle while

La boucle **while** permet de répéter une instruction sans connaître à l'avance le nombre de répétitions. Si *condition* dénote une expression booléenne, la boucle **while** est de la forme :

```
while condition do instruction
```

Cette boucle peut se traduire par « tant que la condition est vraie répéter l'instruction ». Lorsque l'ordinateur arrive sur une telle boucle, il va en effet évaluer la valeur de la condition : si celle-ci vaut **false**, la boucle est sautée et l'ordinateur passe à l'instruction suivante. En revanche, si la condition vaut **true**, l'instruction de la boucle est exécutée puis l'ordinateur réévalue la condition et décide de poursuivre ou non la boucle en fonction de sa valeur. On dit alors que la condition est une condition d'entrée dans la boucle ce qui fait qu'une boucle **while** peut ne jamais exécuter l'instruction à répéter.

Exemple ▷ *Nous allons écrire un programme qui demande à l'utilisateur de rentrer un nombre que l'on va diviser par 2 tant que le résultat est supérieur à 1.*

```
program division;

var valeur:real;

begin
  writeln('Entrez un nombre svp? ');
  readln(valeur);

  while valeur>1 do
    valeur:=valeur/2;

  writeln(valeur);
end.
```

▶ L'instruction de la boucle **while** doit forcément changer la valeur de la condition ; si ce n'est pas le cas, cela peut conduire à une boucle qui s'exécute indéfiniement (« boucle infinie »).

Exemple ▷ *Voici à présent un exemple de boucle* **while** *infinie. Nous allons reprendre le programme précédent et ajouter maladroitement une seconde variable. La première contient la valeur entrée par l'utilisateur et la seconde le résultat des divisions par 2. Dans ce cas, si la condition de la boucle* **while** *est vraie une fois, alors elle le sera toujours et nous obtiendrons une boucle infinie. En effet, la condition fait intervenir la variable depart alors qu'elle n'est jamais modifiée.*

```
program division;

var depart,valeur:real;

begin
  writeln('Entrez un nombre svp? ');
  readln(depart);
  valeur:=depart;

  while depart>1 do
    valeur:=valeur/2;

  writeln(valeur);
end.
```

Exercice 7▶ *Ecrire un programme dans lequel vous déclarez une variable* r *de type* **real**. *Affectez-lui la valeur 32000. Puis, tant que* r *est supérieure strictement à 1, divisez* r *par 2 et affichez sa nouvelle valeur.*

Boucle repeat

Comme la boucle **while**, la boucle **repeat** permet de répéter une instruction sans connaître à l'avance le nombre de répétitions. Si *condition* dénote une expression booléenne, la boucle **repeat** est de la forme :

```
repeat
  ⋮
  instructions
  ⋮
until condition;
```

La boucle **repeat** peut se traduire par « répéter les instructions jusqu'à ce que la condition soit vraie ». Lorsque l'ordinateur arrive sur une telle boucle, il va commencer par exécuter les instructions qui se trouvent entre le mot-clé **repeat** et le mot-clé **until**. Une fois ces instructions exécutées, la condition est évaluée ; si elle vaut **false** l'ordinateur passe à l'instruction qui suit la boucle, sinon l'ordinateur réexécute la boucle. Cette fois-ci, la condition est une condition de sortie de la boucle, ce qui fait que les instructions sont répétées au moins une fois.

Comme pour la boucle **while**, il est important que les instructions dans la boucle modifient la condition de sortie afin d'éviter une boucle infinie.

Exemple ▷ *Le programme suivant demande à l'utilisateur de rentrer un entier strictement positif. Tant que l'utilisateur se trompe et rentre un nombre négatif ou nul, l'ordinateur lui redemande un entier.*

```
program question;

var entier:integer;

begin
  repeat
    writeln('Entrer un nombre >0: ');
    readln(entier);
  until entier>0;
end.
```

Exercice 8▶ *Ecrire un programme qui implémente l'algorithme suivant :*

répéter
demander à l'utilisateur d'entrer un nombre n
jusqu'à ce que $n = 10$.

Choix d'une boucle

Dans la plupart des cas, une boucle peut en remplacer une autre, même si la boucle **for** est la moins souple de toutes. Cependant, selon le problème, les boucles ne sont pas équivalentes au niveau de la simplicité de mise en œuvre. Nous allons considérer un programme qui affiche les nombres de 1 à 10 à l'écran et nous allons proposer une version pour chaque boucle :

```
program compteur;
var i:integer;

begin
  for i:=1 to 10 do
    writeln(i);
end.
```

```
program compteur;
var i:integer;

begin
  i:=1;
  while i<=10 do
    begin
      writeln(i);
      i:=i+1;
    end;
end.
```

```
program compteur;
var i:integer;

begin
  i:=1;
  repeat
        writeln(i);
        i:=i+1;
  until i>10;
end.
```

Lorsque l'on compare les trois programmes, il semble évident que la boucle **for** est la plus appropriée lorsque l'on connaît le nombre d'itérations. Par exemple, la boucle **for** augmente toute seule la variable

i, alors que dans les autres boucles, il est nécessaire de le faire soi-même. On remarque aussi que la condition d'entrée du **while** est exactement la négation de la condition de sortie du **repeat**.

Il est donc important de bien choisir la boucle appropriée en fonction du programme : même si le résultat sera le même au final, l'écriture du programme peut être plus facile et plus rapide.

7.6 Nombres aléatoires

Un ordinateur ne peut pas réellement fournir des nombres qu'il « choisit » au hasard. En informatique, on a recours à des générateurs de nombres pseudo-aléatoires.

Dès qu'un programme nécessite des nombres aléatoires, il faut utiliser une seule fois l'instruction randomize pour initialiser le générateur, généralement en début de programme principal.

Pour obtenir un nombre aléatoire (de façon uniforme) dans un intervalle, il faut utiliser la fonction random :
- random(n) où n est un **integer** renvoie un **integer** compris dans $[0; n[$;
- random, sans paramètre, renvoie un **real** compris entre $[0; 1[$.

> **Exercice 9▶** *Reprenez l'exercice 8 (page 27) afin d'en faire un jeu de devinette. Prenez un nombre aléatoire alea entre 1 et 10 et demandez à l'utilisateur de le deviner jusqu'à ce qu'il le trouve.*

8 Tableaux

Les tableaux représentent un moyen facile de ranger en mémoire et d'accéder à des données du même type. Nous nous limiterons aux tableaux à une ou deux dimensions.

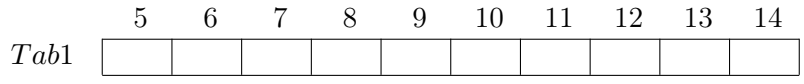
Le type « tableau à une dimension » s'écrit :

```
array [a .. b] of type
```

où a est l'indice de la première case du tableau et b celui de la dernière, et où *type* est le type de chacune des valeurs qui seront contenues dans les cases du tableau. a et b sont forcément de type énuméré, donc pour nous **boolean** ou **integer**.

> **Exemple ▷** *Voici une déclaration d'un tableau à une dimension ainsi qu'une représentation graphique de celui-ci :*

```
var Tab1 : array [5 .. 14] of integer;
```



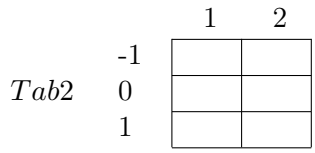
De la même façon, le type « tableau à deux dimensions » s'écrit :

```
array[a .. b, c .. d] of type
```

où [a ;b] représente les valeurs des indices de la première coordonnée, et [c ;d] celles de la seconde. Par convention, la première coordonnée représente les lignes et la seconde les colonnes.

Exemple ▷ *Voici une déclaration d'un tableau à deux dimensions ainsi qu'une représentation graphique de celui-ci :*

```
var Tab2 : array [-1..1, 1..2] of integer;
```



Un tableau n'est pas une expression et n'a donc pas de valeur lorsqu'on le considère dans sa globalité ; en revanche, chacune de ses cases est équivalente à une variable dont le type est spécifié par la déclaration du tableau. Il est possible d'accéder à une case d'un tableau en utilisant la syntaxe suivante :

```
tableau[coordonnées]
```

Exemple ▷ *Si x est une variable de type* ***integer*** *et en reprenant les tableaux déclarés précédemment, on peut écrire :*

- *Tab1[6] :=10 pour affecter à la case d'indice 6 du tableau $Tab1$ la valeur 10 ;*
- *Tab1[0,2] :=10 pour affecter à la case située à la ligne d'indice 0 et la colonne d'indice 2 du tableau $Tab2$ la valeur 10 ;*
- *x:=Tab1[8]+1 pour utiliser la valeur contenue dans la case d'indice 6 du tableau $Tab1$;*
- *x:=Tab1[−1,2]+1 pour utiliser la valeur contenue dans la case située à la ligne d'indice -1 et la colonne d'indice 2 du tableau $Tab2$.*

▶ Il faut faire attention aux coordonnées des cases des tableaux pour éviter de demander à l'ordinateur d'accéder à une case qui n'existe pas : cela provoque un bug et l'arrêt du programme. Dans des cas bien précis mais trop rares, de telles erreurs peuvent être détectées à la compilation.

Puisqu'un tableau est une structure particulière, il est impossible d'afficher un tableau dans sa globalité avec l'instruction **writeln**, de le remplir avec **readln**, ou d'appliquer les opérateurs arithmétiques ou logiques sur le tableau entier. L'ensemble des opérateurs et des instructions que nous avons vu fonctionnent en revanche sur les cases du tableau. La seule opération possible sur un tableau entier est l'affectation ; pour cela, il faut que les tableaux soient de même type. L'affectation d'un tableau à un autre consiste à copier chacune des cases de l'un vers l'autre.

Exemple ▷ *Si $t1$ et $t2$ sont déclarés comme* ***array [1..10] of real****, il est impossible d'écrire* ***writeln****(t1),* ***readln****(t1), t1+t2, etc ...En revanche, on peut écrire t1:=t2 qui copie chacune des valeurs des cases de $t2$ dans les cases correspondantes de $t1$.*

► Pour parcourir un tableau, on utilise souvent la boucle **for** en lui donnant pour valeur de départ l'indice de la première case et pour valeur d'arrivée celui de la dernière case du tableau.

Exemple ▷ *Voici un programme qui remplit aléatoirement un tableau de 10 entiers et l'affiche.*

```
program tableaux;
var tab:array [1..10] of integer;
    i:integer;

begin
  (*initialise les nombres aléatoires*)
  randomize;

  (*parcours des 10 cases
    et affectation d'un nombre aléatoire*)
  for i:=1 to 10 do
    tab[i]:=random(100);

  (*parcours des 10 cases
    et affichage de la case courante*)
  for i:=1 to 10 do
    writeln(tab[i]);
end.
```

Exercice 10► *Ecrivez un programme dans lequel vous déclarez un tableau d'entiers à 100 cases indexées de 0 à 99. Affectez aux cases 0 et 1 le chiffre 1, puis affectez à la case i la somme des cases $i-1$ et $i-2$.*

9 Procédures et fonctions

La décomposition d'un programme en sous-programmes (procédures, fonctions) est indispensable pour mettre en évidence la structure en blocs du programme et le rendre compréhensible par parties. Les procédures permettent d'introduire de nouvelles instructions et les fonctions des nouvelles expressions.

9.1 Procédures

Une déclaration de procédure permet de donner un nom à une suite d'instructions. Une procédure sans paramètre est utilisée pour éviter d'avoir à écrire plusieurs fois une même suite d'instructions figurant plusieurs fois dans un programme, tandis qu'une procédure paramétrée permet d'exécuter des instructions en fonction de valeurs et de variables différentes à chaque appel. L'utilisation des procédures nécessitent de les déclarer et de les définir, si elles ne le sont pas déjà, puis de les appeler lorsqu'elles sont nécessaires.

Déclaration d'une procédure

La déclaration d'une nouvelle procédure intervient dans la partie « déclarations » d'un programme (voir page 9). Pour déclarer une procédure, il faut établir son en-tête (ou prototype) de la façon suivante :

```
procedure nom (liste des paramètres);
```

où *nom* est un identificateur qui permet d'identifier cette procédure. La liste des paramètres est une liste d'identificateurs séparés par une virgule suivis de leur type. S'il existe des paramètres de types différents, il faut alors créer plusieurs listes de paramètres (une au moins pour chaque type) séparées par un point-virgule. Ces paramètres sont appelés « paramètres formels ».

Exemple ▷ *Voici des en-têtes de procédures avec ou sans paramètres :*

```
procedure action1;
procedure action2 (a,b:integer);
procedure action3 (a,b:integer;r:real);
```

Ces identificateurs permettent de donner un nom aux différents paramètres afin de les manipuler dans la procédure. Ces paramètres ainsi déclarés n'existent qu'à l'intérieur de celle-ci.

Définition de la procédure

Une fois que la nouvelle procédure a été déclarée, il faut la définir en indiquant les instructions qui la composent et qui vont manipuler ses paramètres. Pour cela, on va associer un bloc d'instruction au prototype de la procédure en écrivant tout simplement :

```
procedure nom (liste des paramètres);
begin
  instruction_1;
  ⋮
  instruction_n;
end;
```

Ce bloc d'instruction est alors appelé le « corps » de la procédure.

Appel d'une procédure

Une fois qu'une procédure est déclarée et définie, il est possible de l'utiliser : l'ordinateur connaît alors une nouvelle instruction que l'on peut utiliser au même titre que les instructions que nous avons vu précédemment. L'appel d'une procédure s'écrit :

```
nom (liste des paramètres);
```

On remarque que le mot clé **procedure** a disparu. Lors de l'appel, les paramètres qui sont précisés entre les parenthèses sont dits « effectifs ». La liste des paramètres effectifs est une suite d'expressions et de variables séparées par des virgules ; il n'est plus nécessaire de préciser leur type.

La liste des paramètres effectifs doit correspondre à la liste des paramètres formels :
- le nombre de paramètres doit être le même ;
- les différents paramètres doivent être donnés dans le même ordre ;
- et les types des paramètres doivent correspondre.

Exemple ▷ *Si l'on souhaite appeler les procédures définies précédemment, on peut écrire, si x est un* ***integer*** *:*

```
action1;
action2(14+2,5);
action3(x,12,ln(12+5));
```

Lors de l'appel d'une procédure, les paramètres effectifs (ou arguments) sont évalués, c'est-à-dire que si ce sont des expressions, leur valeur est calculée et les instructions qui composent la procédure sont exécutées.

▶ Cela ne vous rappelle rien ? **writeln** et **readln** sont effectivement des procédures prédéfinies.

9.2 Fonctions

Une déclaration de fonction donne un nom à une suite d'instructions dont l'objet est de calculer une valeur, qui dépend ou pas des paramètres formels. Cette définition se rapproche du sens mathématique excepté qu'une fonction en `Pascal` ne peut renvoyer qu'une seule valeur.

Comme précédemment pour les procédures, une fonction doit être déclarée et définie avant de pouvoir être appelée.

Déclaration d'une fonction

La déclaration d'une nouvelle fonction intervient dans la partie « déclarations » d'un programme (voir page 9). Pour déclarer une fonction, il faut établir son en-tête (ou prototype) de la façon suivante :

```
function nom (liste des paramètres):type;
```

où *nom* est identificateur indiquant le nom que l'on souhaite donner à la fonction. Contrairement aux procédures, il faut donner un type à une fonction. Ainsi, *nom* fait référence à une fonction qui associe à ses paramètres une valeur de type *type* qui est uniquement un type simple (c'est-à-dire **boolean**, **integer** et **real**). La liste des paramètres d'une fonction est facultative (mais il est rare de ne pas en avoir) et se construit comme la liste des paramètres formels d'une procédure.

Exemple ▷ *On peut déclarer une fonction addition qui à deux entiers associe leur somme, qui sera elle-même un entier :*

```
function addition (a,b:integer):integer;
```

Définition d'une fonction

Comme pour les procédures, il faut à présent définir le corps de la fonction en associant à son prototype un bloc d'instructions. La particularité cette fois-ci est que la suite d'instructions a pour but de calculer une valeur :

```
function nom (liste des paramètres):type;
begin
  instruction_1;
  ⋮
  instruction_n;
  nom:=expression;
end;
```

Une fois que le calcul est effectué, il faut affecter à la fonction le résultat de ce calcul qui doit être du même type que la fonction. Sans cette affectation, la fonction renverra toujours une valeur indéterminée.

► Dans le corps de la fonction *nom*, il ne faut pas que l'identificateur *nom* apparaîsse dans la partie droite d'une affectation.

Appel d'une fonction

Une fois qu'une fonction est déclarée et définie, il est possible de l'utiliser. L'appel d'une fonction s'écrit :

```
nom (liste des paramètres)
```

On remarque que le mot clé **function** a disparu. La liste des paramètres effectifs obéit aux même règles que les paramètres effectifs d'une procédure. Une fonction peut être appelée à n'importe quel emplacement destiné à une expression.

Exemple ▷ *Si l'on souhaite appeler la fonction définie précédemment, on peut écrire, si x et y sont deux* ***integer*** *:*

```
writeln(addition(1,1));
writeln(addition(1,1+5)*5);
writeln(addition(x,y));
```

Lors de l'appel d'une fonction, les paramètres effectifs (ou arguments) sont évalués, c'est-à-dire que si ce sont des expressions, leur valeur est calculée et l'appel de la fonction est remplacé par la valeur qu'elle a calculé. A ce titre, une fonction ne doit pas utiliser de **readln** et ne doit utiliser **writeln** uniquement pour afficher un message d'erreur.

9.3 Passages de paramètres

Nous venons de voir que dans la déclaration ou l'appel d'une procédure ou d'une fonction, des paramètres pouvaient intervenir. Nous allons voir maintenant comment utiliser ces paramètres. Il faut distinguer deux sortes de paramètres :

- on appellera les paramètres d'entrée les paramètres dont la valeur, connue avant l'appel de la fonction ou de la procédure, ne sera pas modifiée par celle-ci. On dit aussi que ces paramètres sont passés par valeur (car seule la valeur est fournie à la fonction ou à la procédure) ;
- on appellera les paramètres de sortie les paramètres dont la valeur est modifiée par la fonction. On dit que ces paramètres sont passés par variable (car cette fois-ci, la variable entière est passée afin d'être modifiée). Ces paramètres doivent être précédés du mot clé **var**.

Une fonction ne prend pas de paramètre de sortie (elle retourne déjà un résultat). En pratique, les paramètres effectifs correspondants à des paramètres d'entrée peuvent être soit des expressions, soit des valeurs (ce sera alors leur valeur qui sera envoyée à la procédure ou à la fonction). Au contraire, les paramètres effectifs correspondants à des paramètres de sortie sont forcément des variables puisqu'ils vont être modifiés. De plus, il ne faut pas oublier que le nom des paramètres effectifs et des paramètres formels n'a pas d'importance : seul l'ordre dans lequel les paramètres sont fournis et déclarés compte.

Exemple ▷ *Nous allons écrire une procédure qui fournit à la fois le quotient et le reste de la division. Le* ***var*** *s'applique aux paramètres formels q et r de façon à conserver les modifications apportées par la procédure.*

```
program division;

procedure euclide(a,b:integer;var q,r:integer);
begin
  r:=a;
  while r>=b do
    begin
      r:=r-b;
      q:=q+1;
    end;
end;

var x,y,quotient,reste:integer;
begin
  writeln('Entrez x et y:'); readln(x,y);
  euclide(x,y,quotient,reste);
  writeln('quotient = ',quotient,' et reste = ',reste);
end.
```

► Les paramètres d'une fonction ou d'une procédure ne peuvent pas être de type **array** (voir page 37).

9.4 Variables locales et variables globales

Les variables ont une visibilité différente selon l'endroit où elles sont déclarées. Une variable est dite « globale » quand elle est visible par tous les sous-programmes (procédures et fonctions). Au contraire, une variable est dite « locale » quand elle n'est visible que par un des sous-programmes.

En effet, lorsqu'une variable est nécessaire à l'exécution d'une procédure ou d'une fonction mais pas au reste du programme, il faut la déclarer localement à cette procédure ou fonction. Les variables locales sont déclarées à l'aide du mot-clé **var** placé entre l'en-tête de la procédure ou de la fonction et son corps.

Exemple ▷ *Soit un programme qui permet d'afficher la somme des n premiers nombres entiers. Nous allons écrire une fonction qui calcule cette somme pour un n donné :*

```
program SommeNPremiers;

function somme(n:integer):integer;
var s,i:integer;
begin
  (*initialisation de la somme à 0*)
  s:=0;
  (*Calcul de la somme*)
  for i:=1 to n do
    s:=s+i;
  (*Affectation de la valeur à la fonction*)
  somme:=s;
end;

var nb:integer;
begin
  writeln('Entrez n:');
  readln(nb);
  writeln('la somme vaut ',somme(nb));
end.
```

Dans l'exemple ci-dessus, les variables *s* et *i* ne sont pas considérées comme des paramètres (elles n'ont pas de valeurs fixées à l'appel de la fonction) et ne servent qu'à l'intérieur de la fonction : ce sont donc des variables locales à la fonction *somme*. De même, la variable *nb* est locale au programme principal puisqu'elle n'est utilisée que par celui-ci.

Exemple ▷ *Prenons à présent un exemple jouet plus compliqué et plus complet :*

```
program Essai;
var i:integer;

procedure P;
var i,j:integer;
begin
  ⋮
end;

procedure Q;
begin
  ⋮
  writeln(i);
  ⋮
end;

var k,l:integer;
begin
  ⋮
end.
```

La variable i déclarée à la deuxième ligne de ce code est une variable dite globale. Elle est visible dans tous les sous-programmes (fonctions et procédures) et le programme principal. Les variables k et l déclarées juste avant le programme principal sont des variables locales au programme principal. La procédure Q n'a pas de variable locale, mais comme i est déclarée globalement, Q peut y accéder. Enfin, la procédure P déclare une variable locale nommée i et une autre nommée j. La variable j ne peut être utilisée que dans le corps de P. Nous sommes dans le cas où une variable locale porte le nom d'une variable globale. Dans ce cas, `Pascal` considère que le i utilisé dans le corps de P est le i déclaré en local, et non plus celui déclaré en global : ce cas est donc à éviter car il peut prêter à confusion.

10 Déclaration de types

En `Pascal`, il est possible de définir de nouveaux types. Dans notre cas, nous nous limiterons à définir des types basés sur le type **array** :

```
type nouveau_type = array [a..b] of type;
type nouveau_type = array [a..b,c..d] of type;
```

où a,b,c,d sont les bornes des indices du tableau et *type* est un type simple (**boolean**, **integer** et **real**).

L'avantage des types définis est qu'ils peuvent être utilisés comme paramètre d'une fonction ou d'une procédure et qu'ils sont plus courts à écrire qu'une déclaration de type **array**.

En pratique, lorsqu'on aura besoin de faire passer un tableau en paramètre, nous serons obligés de passer par un type défini. La déclaration des nouveaux types se fait dans la partie déclaration, en général avant la déclaration des variables globales, des procédures et des fonctions.

Exemple ▷ *Voici un programme qui permet de manipuler un histogramme des températures en France. L'histogramme est représenté par un type défini. Il est d'abord rempli aléatoirement puis affiché :*

```
program nouveau_type;
type histogramme = array [-4..40] of integer;

(*remplit un histogramme*)
procedure remplir(var h:histogramme);
var i:integer;
begin
  for i:=-4 to 40 do
    h[i]:=random(100);
end;

(*affiche un histogramme*)
procedure affichage(h:histogramme);
var i:integer;
begin
  for i:=-4 to 40 do
    writeln(i,'degrés : ',h[i],' fois');
end;

var histo:histogramme;
begin
  randomize;
  remplir(histo);
  affichage(histo);
end.
```