Programmation en Turbo-Pascal – Rappels

Description d'un ordinateur

Un ordinateur est une boîte capable d'effectuer mécaniquement des instructions données par un utilisateur. En aucun cas il n'est capable de réflexion. Il se contente de savoir effectuer quelques opérations simples, facilement codables par des circuits électroniques (somme, produit, division...), et certaines opérations un peu plus complexes, obtenues en combinant ces opérations élémentaires.

Un ordinateur est constitué de :

- un processeur, qui effectue les opérations ;
- une RAM (mémoire vive), dont l'ordinateur se sert comme une feuille de brouillon ;
- des interfaces d'entrée, permettant à l'utilisateur de donner ses instructions : clavier, souris, lecteur (disquette, CD, ZIP, clé USB...), tout appareil annexe (appareil photo...), scanner etc.
- des interfaces de sortie : écran, lecteur-enregistreur, imprimante, robot...

Le processeur travaille en binaire : toutes les instructions doivent lui être données sous forme d'une succession de 0 et de 1 (c'est plus facile à coder sous forme d'impulsions électriques). Il donne en sortie également un résultat en binaire.

Vocabulaire :

- 1 bit : un chiffre dans l'écriture binaire ;
- 1 octet : 8 bits
- 1 kO (kilo-octet) : 1024 octets ($1024 = 2^{10}$; ainsi, 1 kO comporte un nombre de bit égal à une puissance de 2)

L'utilisation du langage binaire explique l'importance des puissances de 2 en informatique. Par exemple, le code ASCII code 256 = 2^8 caractères en binaire. Chaque caractère est donc codé sur 8 bits, donc 1 octet. Chaque caractère qu'on tape prend donc en mémoire 1 octet.

Comme nous, pauvres humains, trouvons assez abstrait et peu utilisable un langage entièrement binaire, il est bien entendu que nous ne donnons pas les instructions directement en binaire : nous utilisons pour ce faire un langage de programmation qui va se charger de faire la transcription pour nous.

Le langage de programmation

Un langage de programmation est la donnée de :

- une syntaxe très précise pour toutes les opérations qu'on peut envisager de faire faire à l'ordinateur ;
- une interface permettant à l'utilisateur de rentrer un texte suivant cette syntaxe ;
- un programme assez complexe, capable d'effectuer la conversion du texte en des instructions binaires compréhensibles par l'ordinateur : il s'agit du compilateur.

La compilation reste une action complètement mécanique, sans aucune interprétation de la part de l'ordinateur ; afin que le compilateur s'en sorte, il est donc indispensable que le texte entré par l'utilisateur ne comporte aucune ambiguïté ; il est par conséquent impératif de bien suivre la syntaxe, souvent très rigide, des langages de programmation ; cette syntaxe a justement été définie de manière à éviter toute possibilité d'ambiguïté.

Il existe de nombreux langages de programmation : Turbo Pascal, C, C++, Java, Perl, Caml, etc. Turbo Pascal n'est pas le plus efficace (et de fait n'est en pratique jamais utilisé par des professionnels), mais est le plus limpide et le plus structuré des langages de programmation, et est de ce fait idéal pour l'apprentissage de l'informatique.

La structure d'un programme en Pascal

Un programme en Pascal comprend :

1. une entête (facultative, mais conseillée) permettant de donner un nom au programme ;
2. une partie déclarative, dans laquelle on définit tous les objets dont on va se servir, par exemple les variables. Il ne s'agit pas d'allouer des valeurs aux objets, mais une certaine place disponible en mémoire ; il faut donc savoir de quel type sont les objets (réels, entiers...) : de leur type va dépendre la place mémoire qui leur sera réservée. Cette place mémoire leur est réservée une fois pour toute, au début du programme, d'où la nécessité de déclarer tous les objets à ce moment.

On peut déclarer les différentes variables ou constantes dont on se servira, mais également des fonctions ou procédures, qui sont des raccourcis pour certaines expressions ou opérations dont on se sert plusieurs fois.

3. un corps de programme, commençant par `begin`, et s'achevant par `end`. Le corps de programme est constitué d'une succession d'instructions. Les instructions sont séparées les unes des autres par un point-virgule, à ne pas oublier (son importance est capitale : c'est grâce à ce point-virgule que le compilateur sait où commence et où finit une instruction).

Syntaxe :

```
-----  
program nomprog;      (entête)  
-----  
var .....;  
procédure .....;    (partie déclarative)  
fonction .....;  
-----  
begin  
  instruction_1;  
  instruction_2;      (corps du programme)  
  .....  
  instruction_n;  
end.  
-----
```

Un mot de présentation

- Laissez des lignes blanches pour séparer les différents blocs du programme (entre chaque procédure et fonction par exemple). Cela permet de partitionner un peu le programme.
- Vous avez la possibilité de mettre des **commentaires** dans vos programmes : tout texte mis entre accolades {de cette manière} n'est pas pris en considération par le compilateur. Utilisez ces commentaires pour rendre votre programme plus compréhensibles, en indiquant de temps à autres en commentaire ce que fait le programme. Une explication en français est souvent éclairante pour comprendre un programme, la lecture brute d'un programme en langage Pascal et non commenté pouvant parfois s'avérer assez fastidieuse et délicate.
- **INDENTEZ** afin de dégager les structures imbriquées (structures conditionnelles, boucles imbriquées...) À chaque nouvelle structure, indentez de quelques caractères supplémentaires ; à chaque structure composée (cf plus loin), indentez d'avantage ce qui se trouve entre `begin` et `end` ; cela permet de bien mettre le `begin` en face du `end`, et de mieux repérer le début et la fin de chaque structure.

Déclaration des variables

Il faut voir une variable comme une place en mémoire. Suivant le type de donnée qu'on veut stocker dans une variable, la taille de cette place-mémoire n'est pas la même. Il faut donc préciser à l'ordinateur quel est le type de la variable au moment où on déclare celle-ci, afin qu'il lui réserve la bonne place. Il est ensuite impératif de ne pas utiliser cette variable avec un autre type de donnée (le compilateur vous renverra une erreur).

Syntaxe de déclaration d'une variable :

```
var nom : type ;
```

On déclare ainsi une variable dont le nom est `nom`, et du type `type`. Pour déclarer plusieurs variables de même type :

```
var nom1,nom2,nom3 : type ;
```

Pour déclarer plusieurs variables de plusieurs types :

```
var nom1,nom2:type1;  
    nom3,nom4,nom5:type2;  
    nom6:type3;
```

Les alignements ne sont pas nécessaires, mais conseillés. N'oubliez pas les virgules, entre variables de même type, et les points-virgules entre les différents types. On ne donne pas de valeur au moment de la déclaration.

Les types les plus courants sont :

- `shortint` : entiers compris entre -128 et 127 (1 octet) ;
- `integer` : entiers compris entre -32768 et 32767 (2 octets) ;
- `longint` : entiers compris entre $-2^{32} = -2.147.483.648$ et $2^{32} - 1 = 2.147.483.647$ (4 octets)
- `real` : nombres réels, de $2.9 \cdot 10^{-39}$ à $1.7 \cdot 10^{38} +$ un signe ; 11 à 12 chiffres significatifs (6 octets).
- `string` : chaîne de caractère, limitée à 255 caractères (256 octets)

Une chaîne de n caractères est stockée sur $n + 1$ octets : un octet pour le longueur de la chaîne, et un octet pour chaque caractère, dans le code ASCII.

- `boolean` : une variable de ce type peut prendre deux valeurs uniquement : `true` ou `false`. Ce type de variable est très utile lors de tests.
- tableaux (matrices) : on en reparlera.

Par exemple,

```
var x,y:real;
    n:integer;
```

déclare trois variables x et y , de type réel, et n de type entier. Cette déclaration ne donne pas de valeur à x , y et n .

Mise en garde : Turbo Pascal ne fait pas la distinction entre majuscules et minuscules. Vous ne pouvez donc pas définir deux variables différentes n et N .

Les constantes

Dans certaines situations, il peut être utile de déclarer des constantes, c'est-à-dire de donner un nom à une valeur qui *a priori* ne sera pas modifiée à l'intérieur du programme, par exemple la taille d'un tableau, la valeur d'un paramètre fixe etc. L'intérêt de définir une constante plutôt que d'utiliser directement la valeur dans le programme réside dans le fait qu'il est plus aisé de modifier cette valeur par la suite, par exemple si on se rend compte que la taille choisie pour le tableau est insuffisante pour l'usage qu'on veut en faire : il suffit de changer la valeur qu'on a donné à la constante en début de programme ; on n'est pas obligé de changer la valeur de toutes les occurrences au sein du programme

On déclare et définit la valeur d'une constante dans la partie déclarative du programme :

`const Nmax=100` ; définit par exemple une constante appelée `Nmax` dont la valeur est 100.

Les instructions d'entrée et de sortie

Ces instructions sont l'interface avec l'utilisateur : il faut pouvoir afficher quelque chose à l'écran, et il faut pouvoir lire des informations entrées à l'écran par l'utilisateur. Ces instructions, comme toutes instructions, doivent être placées dans le corps du programme.

- `write(A,B,C)` ; : écrit à la suite les un des autres les expressions A , B et C (on peut bien sûr afficher autant de choses qu'on veut ; `write(A)` n'affiche que A).

Les expressions qu'on peut afficher peuvent être des nombres (réels ou entiers), entrés directement, des chaînes de caractère, entrées entre apostrophes, ou la valeur d'une variable, dont le nom est entré sans apostrophe. Exemple : si la variable n vaut 3, `write(n, ' et ', 2)` ; affiche : 3 et 2. Remarquez que j'ai rajouté les espaces dans la chaîne de caractère. Sans cela, tout est collé. Ainsi, pour afficher deux nombres n et m séparés par un espace, je peux faire : `write(n, ' ', m)` ;.

Vous remarquerez que si je veux afficher une apostrophe, je crée une ambiguïté, car ce symbole est utilisé dans la syntaxe ; afin de remédier à ce problème, on double l'apostrophe. Par exemple, pour afficher « c'est », on va écrire : `write('c'est')` ;.

Remarque : souvent, vous vous perdez dans les apostrophes. Gardez en tête que les apostrophes sont là uniquement en tant que délimiteurs de chaînes de caractères.

- `writeln(A,B,C)` ; : pareil mais va à la ligne après avoir écrit A , B et C .
Peut être utile pour sauter une ligne à l'affichage : `writeln` ; n'affiche rien et va à la ligne.
- `read(n,m,p)` ; : le programme s'arrête et attend que l'utilisateur rentre une valeur pour n , validée par la touche entrée, puis une valeur pour m , puis une valeur pour p . Ici, n , m et p sont des variables qui doivent avoir été déclarées ; elles reçoivent les valeurs entrées par l'utilisateur.
- `readln(n,m,p)` ; : idem, mais suivi d'un retour à la ligne.

- `clrscr` : efface l'écran. Pour pouvoir utiliser cette instruction qui n'est pas définie dans le langage Pascal lui-même, il faut introduire une librairie spéciale appelée `crt`. Pour cela, il faut ajouter, entre l'entête et la partie déclarative, la ligne suivante :
`uses crt ;`

Affectation d'une valeur à une variable

Syntaxe : `nomvariable :=valeur ;`

Par exemple, affecter la valeur 1 à `x` : `x :=1 ;`

Doit être utilisé dans le corps du programme uniquement.

Remarques :

- Il faut avoir au préalable déclaré la variable `x`, dans la partie déclarative du programme.
- Il faut que la valeur soit en concordance avec le type de `x`. Par exemple, si `x` est de type entier, `x :=1.1 ;` renvoie une erreur. De même, `x :=a/b ;` renvoie une erreur, car le résultat d'une division est de type réel, même si `a` et `b` sont entiers et que le quotient est un entier.
- La valeur précédente de `x` est écrasée définitivement
- On peut définir une nouvelle valeur de `x` à partir de l'ancienne. Par exemple : `x :=x+1 ;` rajoute 1 à la variable `x`. L'ancienne valeur de `x` n'est écrasée qu'à la fin du calcul, ainsi, tous les calculs sont effectués à l'aide de l'ancienne valeur.
- Au moment de la déclaration, l'ordinateur a donné une place mémoire (= des bits) à une variable `x`. Cette place mémoire n'est généralement pas vierge. Autrement dit, elle contient une succession de 0 et de 1 qui codent une certaine valeur de `x`. Ainsi, initialement, la variable `x` possède une certaine valeur, qui n'a pas beaucoup de sens, et qui dépend essentiellement de l'usage précédent qui a été fait de la place mémoire utilisée pour `x`. Ainsi, n'utilisez pas une variable `x` dont vous n'auriez pas défini vous-même la valeur (initialisation), vous risqueriez de trouver des résultats bizarres.

Opérations arithmétiques et fonctions usuelles

Un principe : **quand vous faites une opération, il faut toujours dire quelle en est la destination** (ce que l'ordinateur doit faire du résultat) : l'affecter à une variable, l'afficher... Une instruction qui se limiterait à : `x+y` ; n'est donc pas recevable. on peut en revanche faire : `z :=x+y ;`, ou `write(x+y) ;`. Dans ce qui suit, je me contente de décrire les opérations, sachant qu'il faut ensuite faire quelque chose du résultat.

- l'addition : `x+y` ;
- la soustraction : `x-y` ;
- le produit : `x*y` ;
- la division de réels ou entiers : `x/y` ; donne un résultat de type `real`.
- la division entière (quotient) de deux entiers : `x div y` ; résultat de type entier.
- le reste de la division de deux entiers `x mod y` ;
- logarithme népérien : `ln(x)` ;
- exponentielle : `exp(x)` ;
- sinus, cosinus : `sin(x)` ; `cos(x)` ;
- racine carrée : `sqrt(x)` ;
- la valeur absolue : `abs(x)` ;
- la troncature d'un réel : `trunc(x)` ; résultat de type entier.
Attention, `trunc(-2.3)=-2` ; il ne s'agit pas de la valeur absolue sur les valeurs négatives.
- la troncature d'un réel : `int(x)` ; résultat de type réel
(la seule différence avec `trunc` est le type de la sortie)
- arrondi : `round(x)` résultat de type entier, l'entier le plus proche.
- Pour un entier `n`, `random(n)` : valeur aléatoire comprise entre 0 et `n - 1`. Le résultat est de type entier.
Pour utiliser la fonction `random` de manière efficace il faut rajouter l'instruction `randomize` ; en début de corps de programme, afin d'initialiser le choix aléatoire.
Attention, `random(n)` donne une valeur entière comprise dans un intervalle d'entiers débutant à 0. Si l'on désire générer des valeurs aléatoires sur un autre intervalle d'entiers, il faut faire une translation. Par exemple, un nombre aléatoire entre 1 et 50 est obtenu par `random(50)+1`.
- Si on ne précise pas d'argument, `random` renvoie un réel aléatoire entre 0 et 1, de façon uniforme. Pour cet usage également, il faut initialiser à l'aide de l'instruction `randomize`.

Comparaisons et booléens

On peut comparer l'égalité de deux objets, qu'ils soient numériques ou non (on peut comparer des booléens, des chaînes de caractère etc.) L'instruction pour comparer deux objets est `A=B`.

Le résultat d'un tel test est un booléen (`true` si l'égalité est vraie, `false` sinon). Comme pour tout résultat d'opération, il faut dire à l'ordinateur quoi en faire : l'affecter à une variable de type booléen, l'afficher, ou l'utiliser directement dans une structure conditionnelle (voir plus loin)

Dans le cas de valeurs numériques, d'autres tests :

`a=b` ; `a<b` ; `a>b` ; `a<=b` ; `a>=b` ; `a<>b` ; (le dernier teste si `a` est différent de `b`).

À chaque fois, le résultat est un booléen.

On peut effectuer les opérations suivantes sur les booléens : `and` ; `or` ; `not` ; `xor` ; (ou exclusif)

Les opérations booléennes sont prioritaires sur les tests. Ne pas oublier les parenthésages !

Instruction composée

Il est parfois nécessaire de faire en sorte qu'une succession d'instructions soit considérée par le compilateur comme une seule et même instruction. Pour cela, on crée une structure composée en entourant la succession d'instructions d'un `begin` et d'un `end` ; :

```
begin
  instruction_1;
  ...
  instruction_n;
end
```

Tout l'ensemble de ces n instructions est considérée syntaxiquement comme une seule instruction.

Structures conditionnelles

Cette structure permet de n'effectuer une instruction que si certaines conditions sont remplies (si... alors... sinon...)

Syntaxe : deux structures suivant qu'il y a une alternative ou non :

```
if boolean then instr1;
if boolean then instr1 else instr2;
```

Dans les deux structures, on n'effectue l'instruction `instr1` que si la valeur de `boolean` est `true` ; de plus, si ce n'est pas le cas (et seulement dans ce cas), on effectue l'instruction 2 dans la deuxième structure.

Remarques :

- `instr1` et `instr2` ne peuvent être constitués que d'une instruction. Si on a plusieurs instructions à donner conditionnellement, utiliser une instruction composée (donc entourer la succession d'instructions par `begin` et `end`).
- La structure dans son ensemble est considérée comme une seule instruction.
- Par conséquent, surtout pas de point virgule avant la fin de l'instruction (sauf à l'intérieur d'éventuelles instructions composées bien entendu). Notamment, pas de point virgule avant `else` (cela permet d'éviter la confusion entre les deux syntaxes). En particulier, lorsque `instr1` est une instruction composée, ne pas mettre de point-virgule après le `end` terminant cette instruction.
- l'expression `boolean` peut être la valeur d'une variable booléenne, le résultat d'un test (if `x<2` ...) ou le résultat d'une opération booléenne.
- On peut imbriquer des tests les uns dans les autres.

Structures répétitives (boucles)

On est parfois amené à faire exécuter plusieurs fois à l'ordinateur la même instruction (ou succession d'instructions). Le nombre d'itérations peut être déterminé à l'avance, ou dépendre d'un test d'arrêt. L'objet des structures répétitives est d'éviter au programmeur de taper plusieurs fois les instructions en question. Il existe en Turbo Pascal trois types de structures répétitives.

1. La boucle For ... do ...

- Syntaxe (`n1` et `n2` sont deux valeurs de type entier (soit numériques, soit des variables de type entier) :

```
For i := n1 to n2 do
  instruction;
```

- L'ordinateur incrémente la variable *i* *après* avoir effectué l'instruction. Il effectue donc successivement l'instruction donnée pour les valeurs de *i* égales tout d'abord à *n1*, puis à *n1+1*, et ainsi de suite jusqu'à *n2*. Si *n1=n2*, il n'effectue qu'une fois l'instruction. Si *n1>n2*, il n'effectue pas l'instruction.
- Cette instruction peut utiliser la variable *i*. Attention à ne pas modifier la valeur de *i*, sous risque de grosses surprises, et de grosses frayeurs (boucle sans fin)
- La variable *i* doit avoir été déclarée, comme toute variable. Elle doit être d'un des types entiers.
- L'ordinateur ne tient compte que d'une instruction après la commande `do`. Si on veut effectuer plusieurs instructions dans la boucle, il est donc nécessaire d'utiliser une instruction composée, c'est-à-dire d'entourer les instructions de `begin` et de `end` ;
- Pour cette boucle, on connaît à l'avance le nombre d'itération de d'instruction : $n_2 - n_1 + 1$. Elle est donc inadéquate dans le cas où la condition d'arrêt s'exprime par un test, cas dans lequel on ne sait en général pas d'avance le nombre de passages dans la boucle.
- `For i := n1 downto n2 do ...` fait la même chose, mais en décrémentant la variable *i* à chaque passage.

2. La boucle `While ... do ...`

- Syntaxe :


```
While expression_booléenne do
  instruction;
```
- Tant que l'expression booléenne a la valeur `true`, l'ordinateur répète l'instruction.
- L'expression booléenne peut être un test, une variable booléenne, et comporter des opérateurs booléens `not`, `and`, `or` et `xor`.
- Il est du ressort du programmeur de s'assurer que l'expression booléenne prendra forcément la valeur `false` à un moment ou à un autre. Si ce n'est pas le cas, la boucle s'effectue indéfiniment.
- En particulier, on ne connaît pas au départ le nombre exact d'itération : l'arrêt est déterminé par une condition.
- Afin que la syntaxe soit non ambiguë, on ne peut mettre qu'une seule instruction après `do`. Si on veut mettre une succession d'instructions, il va donc falloir créer une instruction composée, c'est-à-dire entourer la succession d'instructions par `begin` et `end`.
- La boucle peut ne pas être effectuée du tout si au premier passage, l'expression booléenne est `false`.

3. La boucle `Repeat ... until ...`

- Syntaxe :


```
Repeat
  instruction_1;
  instruction_2;
  ...
  instruction_n
until
  expression_booléenne;
```
- L'ordinateur répète la succession des instructions 1 à *n*, jusqu'à ce que l'expression booléenne prenne la valeur `true`.
- L'expression booléenne peut être un test, une variable booléenne, et comporter des opérateurs booléens `not`, `and`, `or` et `xor`.
- Il est du ressort du programmeur de s'assurer que l'expression booléenne prendra forcément la valeur `true` à un moment ou à un autre. Si ce n'est pas le cas, la boucle s'effectue indéfiniment.
- En particulier, on ne connaît pas au départ le nombre exact d'itération : l'arrêt est déterminé par une condition.
- Contrairement aux deux boucles précédentes, il n'y a pas besoin de mettre de `begin` et `end` autour des instructions s'il y en a plusieurs : l'ordinateur prend en considération toutes les instructions situées entre la commande `repeat` et la commande `until`.
- Contrairement à la boucle `while`, la succession d'instructions est effectuée au moins une fois, le premier test se faisant à la fin de la première boucle.

Fonctions

Il est parfois utile de définir une fonction ou une procédure en début de programme, c'est-à-dire une succession d'instructions à laquelle on donne un nom. Les avantages :

1. On écrit une seule fois la succession d'instructions ;
2. La fonction (ou procédure) est plus facilement accessible (en début de programme). Plus facile de la modifier ! (pas besoin d'aller chercher dans le corps du programme)
3. Cela dégage le corps du programme de toute technicité : il ne comprend plus que la structure du programme, et est donc plus facilement compréhensible. Toute la partie technique est rejetée dans la définition des fonctions et procédures.

Une fonction est un bloc d'instructions qui prend en entrée une variable X d'un certain type, et calcule une valeur (c'est-à-dire un objet d'un type donné) de sortie $f(X)$. On déclare une fonction dans la partie déclarative d'un programme, généralement après avoir déclaré les variables. La déclaration consiste en la description de la fonction.

Syntaxe (à mettre entièrement dans la partie déclarative) :

```
function NomFonction(X1:type1; ... ;Xn:typen):typef; {typef:type du résultat}
                                {X1,...,Xf: variables d'entrée (paramètres)}
var Y1:typey etc.;              {variables locales}
begin
  instructions;
  NomFonction:=valeur;
end;
```

- La valeur de la fonction doit être affectée à `NomFonction`. La valeur prise par la fonction est la valeur de la dernière affectation. Ainsi, la dernière ligne de la fonction consiste toujours en l'affectation d'une valeur à `NomFonction`
- `NomFonction` ne se comporte pas comme une variable quelconque : on ne peut que lui affecter une valeur, et pas réutiliser cette valeur au sein de la fonction. Il n'y a donc aucune raison de lui affecter plusieurs fois des valeurs. Ainsi, cette instruction d'affectation n'apparaît en général (et dans l'idéal) qu'une fois dans la fonction, tout à la fin.
- Dans le corps du programme, on fait appel à une fonction en écrivant son nom suivi d'une spécification des paramètres : `Y :=NomFonction(a,6,b)` ; affectera à la variable `Y` la valeur prise par la fonction pour `X1=a`, `X2=6` et `X3=b`.
- On peut définir des variables locales (`Y1,...`). Ces variables n'ont un sens qu'au sein de la fonction (variables muettes). En particulier, elles sont indépendantes des autres variables portant éventuellement le même nom déclarées dans d'autres fonctions et procédures. Si une fonction fait appel à une autre fonction, déclarer localement des variables permet donc d'éviter les interférences entre variables auxquelles on a données le même nom (par exemple `i` pour indexer une boucle `for`.) On a donc tout intérêt à déclarer localement toute variable qui peut l'être.
- Les variables `X1,...,Xn` n'ont pas besoin d'avoir été déclarée auparavant : l'entête de la fonction fait office de déclaration locale, valable au sein de la fonction.
- En principe, une fonction ne fait que calculer une valeur ; en particulier, elle ne modifie pas la valeur des variables globales. C'est pour cela qu'on utilise des variables locales.
- Même si c'est possible de le faire, évitez dans une fonction toute instruction qui ne soit pas du ressort du calcul. En particulier, **pas de lecture, pas d'affichage du résultat etc.**

Procédures

Une procédure est un bloc d'instructions qui prend en entrée un certain nombre de variables en entrée, et effectue les instructions avec les valeurs données des variables d'entrée. On déclare une procédure dans la partie déclarative du programme. La déclaration consiste en la définition de la procédure.

Syntaxe (à mettre entièrement dans la partie déclarative) :

```
procedure NomProc(X1:type1; X2:type2; var Y1,Y2:type3)
                                {X1,X2,Y1,Y2: variables d'entrée (paramètres)}
var Z1:type4;                    {variables locales}
begin
  instructions;
end;
```

- On fait appel à une procédure en écrivant son nom suivi d'une spécification des paramètres : par exemple, `NomProc(a,6,b,1)` ; effectue la procédure `NomProc` avec les valeurs d'entrée $X1=a$, $X2=6$, $Y1=b$, $Y2=1$.
- Les différents paramètres $X1, X2, Y1, Y2$ n'ont pas besoin d'être déclarés avant.
- Deux sortes de paramètres pour la procédure `NomProc` :
 - * $X1$ et $X2$, appelés *paramètres valeurs*, non précédés du mot `var` : la valeur des variables globales $X1$ et $X2$ n'est pas modifiée par la procédure; elle peut être modifiée au cours de la procédure mais retrouve automatiquement sa valeur initiale à la fin. C'est une manière de « protéger » les variables globales. Ce sont donc des paramètres « non variables », reprenant leur valeur initiale à la fin de l'exécution de programme.
 - * les paramètres $Y1$ et $Y2$, appelés *paramètres adresses*, précédés du mot `var` : les valeurs des variables globales $Y1$ et $Y2$ peuvent être modifiées. Les modifications ne sont pas oubliées à la fin de la procédure. Ce sont donc des paramètres « variables ».
- En particulier, lors de l'utilisation d'une procédure, on peut spécifier comme valeur d'utilisation pour un paramètre valeur, soit des variables, soit des constantes, soit une valeur numérique. En revanche, **la spécification d'un paramètre adresse est obligatoirement une variable**, car la valeur pouvant être modifiée, il faut pouvoir tenir compte de cette modification, ce qui n'est le cas ni pour une constante, ni pour une valeur numérique
- Tout comme pour les fonctions, on peut définir des variables locales.

Tableaux

Une variable de type tableau est un ensemble de données de même type rangées dans un tableau rectangulaire (de dimension $n \geq 1$)

- On déclare un tableau de la manière suivante :
`var tab: array[1..5 , 1..4 , 1..6] of real;`
 Je définis ainsi une variable `tab` comme un tableau de dimension 3, de taille $5 \times 4 \times 6$, dont les entrées sont de type `real`. Il s'agit donc ici d'un pavé dans l'espace.
- On accède aux différentes valeurs du tableau en spécifiant le nom de la variable suivi de la position entre crochets :
`tab[2,3,1] := tab[2,3,1]/3;`
 remplace la valeur de l'entrée en position (2,3,1) par son tiers.
- La taille des tableaux doit être un nombre, ou une constante, mais ne peut pas être une variable. Ainsi, la taille du tableau est déterminée précisément au moment de la déclaration.
- Le cas que vous rencontrerez le plus fréquemment est le cas de tableaux unidimensionnels : `var tab : array[1..5] of real` ; introduit une variable tableau de dimension 1, à 5 cases, les valeurs étant accessibles par la commande `tab[i]`, pour $i \in \llbracket 1, 5 \rrbracket$.

Tableaux en paramètre de fonctions ou procédures

On ne peut spécifier de manière directe un objet de type tableau comme paramètre d'une fonction ou d'une procédure, le problème provenant du fait que c'est un type dépendant d'un certain nombre de paramètres (dimension, taille, type des entrées...).

On peut contourner ce problème en définissant un nouveau type de tableaux, spécifiant la taille du tableau et le type de ses entrées. Ce nouveau type ne dépend plus de paramètres ; on peut alors utiliser des variables de ce type comme paramètre de fonctions ou procédures.

La commande pour définir un nouveau type est `type`. On donne un exemple ci-dessous.

```
type matrice=array[1..10;1..10] of real;

procedure truc(A:matrice; var B:matrice);
begin
  ...
end;
```

Ici, `matrice` désigne un nouveau type (qui s'utilise comme `real` ou `integer`), les objets de ce type étant des tableaux 10×10 de réels. La procédure `truc` prend en paramètre valeur un objet A de type `matrice` et en paramètre adresse un objet B de type `matrice` également.