Algorithmes et programmation en Pascal

Faculté des Sciences de Luminy

Edouard Thiel

Cours

Deug 1 Mass MA

Module de 75 heures

1997 à 2004

Table des matières

I. Les variables en Pascal

1 Premiers programmes

1.1 Le programme bonjour

Un programme est une suite d'instructions, certaines étant des mots clés.

Ce programme affiche la chaîne de caractères « Bonjour » à l'écran :

```
PROGRAM bonjour;
BEGIN
    writeln ('Bonjour');
END.
```

Le compilateur est un logiciel qui lit (analyse) un programme et le traduit en code machine, directement exécutable par le processeur de l'ordinateur.

1.2 Commentaires dans un programme

On place un `{commentaire}` dans un programme au-dessus ou à coté d'une instruction.

Le commentaire n'est pas pris en compte à la compilation. Il sert à rendre le programme plus clair à la lecture, à noter des remarques, etc :

```
{ Edouard Thiel - 21/01/2003 }
PROGRAM bonjour;
BEGIN
    { Affiche Bonjour à l'écran }
    writeln ('Bonjour');
END.
```

1.3 Utilisation d'une variable entière

Une variable est une zone dans la mémoire vive de l'ordinateur, dotée d'un nom et d'un type. Le nom de la variable permet d'accéder au contenu de la zone mémoire ; le type spécifie la nature de ce qui peut être stocké dans la zone mémoire (entier, réel, caractère, etc).

On a coutume de représenter une variable par une boîte ; dessous on met le nom, au dessus le type, et dans la boîte le contenu.

Exemple avec une variable de nom `a` et de type entier :

```
PROGRAM var_entiere;
VAR
    a : integer;                      { Déclaration }
BEGIN
    a := 5;                           { Affectation }
    writeln ('valeur de a = ', a);    { Affichage : a = 5  }
END.
```

La structure de ce programme est en 3 parties : le nom du programme, la partie déclarations, et le corps du programme, qui est une suite d'instructions.

La partie déclaration crée les variables (les boîtes) ; leur contenu est indéterminé (on met un '?' dans chaque boîte). La taille de la zone mémoire de chaque variable est adaptée au type (par exemple 1 octet pour un caractère, 4 octets pour un entier, etc).

1.4 Trace et tableau de sortie

La *trace d'un programme* est obtenue en plaçant des `writeln` pour que le programme affiche les valeurs des variables à l'exécution. Cela sert pour mettre au point un programme en TP.

Le *tableau de sortie* d'un programme est un tableau avec une colonne par variable, où l'on écrit l'évolution des variables pendant le déroulement du programme. Demandé en TD et examen.

1.5 Lecture au clavier d'une valeur

```
PROGRAM lit_ecrit;
VAR
    a : integer;
BEGIN
    write ('Entrez un entier : ');   { pas de retour chariot }
    readln (a);                      { Lecture }
    writeln ('valeur de a = ', a);
END.
```

2 Identificateur

Sert à donner un *nom* à un objet.

Syntaxe

On appelle *lettre* un caractère de `'a'..'z'` ou `'A'..'Z'` ou `'_'`.

On appelle *digit* un caractère de `'0'..'9'`.

Un identificateur Pascal est une suite de lettres ou de digit accolés, commençant par une lettre.

Exemples

`x`, `y1`, `jour`, `mois`, `annee`, `NbCouleurs`, `longueur_ligne`.

Remarques

- ▷ Il n'y a pas de différence entre minuscules et majuscules.
- ▷ On n'a pas le droit de mettre d'accents, ni de caractères de ponctuation.
- ▷ Un identificateur doit être différent des *mots clés* (`begin`, `write`, `real`, …)

On se sert des identificateurs pour : le nom du programme, les noms de variables, les noms de constantes, les noms de types.

3 Types prédéfinis

Un *type* décrit un ensemble de *valeurs* et un ensemble d'*opérateurs* sur ces valeurs.

3.1 Type entier : `integer`

Entier signé en complément à deux sur 16 ou 32 bits, selon machine et compilateur : 16 pour Turbo Pascal, 32 pour Delphi.

Sur 16 bits, à valeur dans $-32\,768 \ldots +32\,767$ $(-2^{15} \ldots +2^{15}-1)$.

Sur 32 bits, à valeur dans $-2\,147\,483\,648 \ldots +2\,147\,483\,647$ $(-2^{31} \ldots +2^{31}-1)$.

- Opérateurs sur les entiers :

`abs(x)`	valeur absolue de $\vert x\vert$.
`pred(x)`	$x-1$.
`succ(x)`	$x+1$.
`odd(x)`	`true` si x est impair, `false` sinon.
`sqr(x)`	le carré de x.
`+ x`	identité.
`- x`	signe opposé.
`x + y`	addition.
`x - y`	soustraction.
`x * y`	multiplication.
`x / y`	division, fournissant un résultat de type réel.
`x div y`	dividende de la division entière de x par y.
`x mod y`	reste de la division entière, avec y non nul.

Remarques

- ▷ Attention, les opérateurs `/`, `div` et `mod`, produisent une erreur à l'exécution si y est nul.
- ▷ Lorsqu'une valeur (ou un résultat intermédiaire) dépasse les bornes au cours de l'exécution, on a une erreur appelée *débordement arithmétique*.

3.2 Type réel : `real`

Leur domaine de définition dépend de la machine et du compilateur utilisés.

On code un réel avec une certaine précision, et les opérations fournissent une valeur *approchée* du résultat dit « juste ». Il faut donc se méfier :

Sous Delphi, `writeln(0.3);` affiche `0.2999999`... Ce n'est pas un bug ; simplement, `0.3` n'est pas représentable en base 2. En effet, en base 2 il s'écrit $0,0\overline{1001}$:

base 10	0,3	0,6	1,2	0,4	0,8	1,6	⋯
base 2	0,	0	1	0	0	1	⋯

Exemples de `real`
`0.0` ; `-21.4E3` ($= -21,4 \times 10^3 = -21400$) ; `1.234E-2` ($= 1,234 \times 10^{-2}$)

- Opérateurs sur un argument x réel : `abs(x)`, `sqr(x)`, `+x`, `-x`.
- Si l'un au moins des 2 arguments est réel, le résultat est réel pour : `x - y`, `x + y`, `x * y`.
- Résultat réel que l'argument soit entier ou réel : `x / y` (y doit être non nul) ; fonctions `sin(x)`, `cos(x)`, `exp(x)`, `ln(x)`, `sqrt(x)` (*square root*, racine carrée).
- Fonctions prenant un argument réel et fournissant un résultat entier : `trunc(x)` (partie entière), `round(x)` (entier le plus proche). Si le résultat n'est pas représentable sur un `integer`, il y a débordement.

3.3 Type caractère : `char`

Le jeux des caractères comportant les lettres, les digits, l'espace, les ponctuations, etc, est codé sur un octet non signé.

Le choix et l'ordre des 256 caractères possible dépend de la machine et de la langue. Sur PC, on utilise le code ASCII, où `'A'` est codé par 65, `'B'` par 66, `'a'` par 97, `' '` par 32, `'{'` par 123, etc.

Le code ascii est organisé comme suit : de **0 à 31**, sont codés les caractères de contrôle (7 pour le signal sonore, 13 pour le saut de ligne, etc). De **32 à 127**, sont codés les caractères et ponctuations standards et internationaux. Enfin de **128 à 255**, sont codés les caractères accentués propres à la langue, et des caractères semi-graphiques.

- Les opérateurs sur les chars sont :

`ord(c)`	numéro d'ordre dans le codage ; ici « code ascii ».
`chr(a)`	le résultat est le caractère dont le code ascii est a.
`succ(c)`	caractère suivant c dans l'ordre ascii $\Leftrightarrow$ `chr(ord(c)+1)`
`prec(c)`	caractère précédent c dans l'ordre ascii.

Remarque Il y a erreur à l'exécution si le caractère n'existe pas.

Exemple

```
PROGRAM caracteres;
VAR
    c, d : char;
    a    : integer;
BEGIN
    c := 'F';
    a := ord(c);    { 70 }
    writeln ('Le code ascii de ', c, ' est ', a);
    a := 122;
    c := chr(a);    { 'z' }
    writeln ('Le caractere de code ascii ', a, ' est ', c);
    c := 'j';
    d := succ(c);   { 'k' }
    writeln ('Le caractere suivant ', c, ' est ', d);
END.
```

Exercice Afficher les caractères de code ascii de 32 a 255 (⟶ sur écran et sur imprimante, les résultats sont parfois différents).

Divers

– On peut remplacer `chr(32)` par `#32`, mais pas `chr(i)` par `#i`.
– Le caractère apostrophe se note `''''`.

• Une suite de caractères telle que `'Il y a'` est une *chaîne de caractères*; il s'agit d'un objet de type `string`, que l'on verra plus loin.

3.4 Type booléen : `boolean`

Utilisé pour les expressions logiques.
Deux valeurs : `false` (faux) et `true` (vrai).

• Opérateurs booléens : `not` (négation), `and` (et), `or` (ou).

Exemple

```
{ Declaration }
    petit, moyen, grand : boolean;
{ Instructions }
    petit := false;
    moyen := true;
    grand := not (petit or moyen);
```

Table de vérité 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

• Opérateurs de comparaison (entre 2 entiers, 2 réels, 1 entier et 1 réel, 2 chars, 2 booléens) :
`<`, `>`, `<=`, `>=`, `=` (égalité, à ne pas confondre avec l'attribution `:=`), `<>` (différent).

Le resultat d'une comparaison est un booléen.
On peut comparer 2 booléens entre eux, avec la relation d'ordre `false < true`.

• En mémoire, les booléens sont codés sur 1 bit, avec 0 pour `false` et 1 pour `true`. De là les relations d'ordre. Les opérateurs booléens `not`, `and`, `or` s'apparentent approximativement à $(1-x)$, $\times$, $+$.

4 Déclarations

4.1 Constantes

Une constante est désignée par un identificateur et une valeur, qui sont fixés en début de programme, entre les mots clés `CONST` et `VAR`.

La valeur **ne peut pas** être modifiée, et ne **peut pas** être une expression.

Syntaxe

`identificateur = valeur_constante;`

ou

`identificateur : type = valeur_constante;`

Dans la première forme, le type est sous-entendu (si il y a un point, c'est un réel, sinon un entier ; si il y a des quotes, c'est un caractère (un seul) ou une chaîne de caractères (plusieurs).

Exemple

```
PROGRAM constantes;
CONST
    faux = false;
    entier = 14;              { constantes NOMMEES }
    reel = 0.0;
    carac = 'z';
    chaine = 'hop';
    pourcent : real = 33.3;   { seconde forme avec type }
VAR
    { variables }
BEGIN
    { instructions }
END.
```

4.2 Variables et affectation

Une variable représente un objet d'un certain type ; cet objet est désigné par un identificateur. Toutes les variables doivent être *déclarées* après le `VAR`.

Syntaxe

`identificateur` : type ;

On peut déclarer plusieurs variables de même type en même temps, en les séparant par des virgules (voir exemple ci-dessous).

À la déclaration, les variables ont une valeur **indéterminée**. On initialise les variables juste après le `BEGIN` (on ne peut pas le faire dans la déclaration).

Utiliser la valeur d'une variable non initialisée est une erreur grave !

Exemple

```
VAR
    a, b, c : integer;
BEGIN
    { Partie initialisation }
```

```
    b := 5;
    { Partie principale }
    a := b + c;  { ERREUR, c n'est pas affecte' }
END.
```

L'opération `identificateur := expression;` est une *affectation*. On n'a pas le droit d'écrire `id1 := id2 := expr` , ni `expr := id` ni `expr1 := expr2` .

5 Expressions

Une *expression* désigne une *valeur*, exprimée par composition d'opérateurs appliqués à des *opérandes*, qui sont : des valeurs, des constantes, des variables, des appels à fonction ou des sous-expressions.

Exemple . Étant donné une variable `x`, une constante `max` et une fonction `cos()`, chaque ligne contient une expression :

```
5
x + 3.14
2 * cos(x)
(x < max) or (cos(x-1) > 2 * (x+1))
```

5.1 Syntaxe

Certains opérateurs agissent sur 2 opérandes :

```
operande1 operateur_binaire operande2
```

et d'autres agissent sur 1 opérande :

```
operateur_unaire operande
```

- Les opérateurs binaires sont :
  - opérateurs de relation `= <> <= < > >=`
  - opérateurs additifs `+ - or`
  - opérateurs multiplicatifs `* / div mod and`
- Les opérateurs unaires sont :
  - opérateurs de signe `+ -`
  - opérateur de négation `not`
- Les parenthèses sont un opérateur primaire, elles peuvent encadrer tout opérande.
- Une fonction est aussi un opérateur primaire, elle agit sur l'opérande placé entre parenthèses à sa droite. Certaines fonctions ont plusieurs paramètres, séparés par des virgules.

5.2 Type des expressions bien formées

Une expression doit être « bien formée » pour que l'on puisse trouver sa valeur. Par exemple, `3 * 'a' - true` n'est pas bien formée, et la compilation Pascal échouera.

Dans la partie 3, *Types prédéfinis*, on a déjà dit quels opérateurs sont applicables sur quels types. Mais il y a encore d'autres règles, dont le simple bon-sens !

L'expression bien formée a un type, qui dépend des règles d'évaluation de l'expression.

<u>Exemple</u>
Soit `r` un réel, `i` un entier, `e` une constante entière, `c` un caractère. L'expression

```
(round(r+1) > (i/e)) or (c < 'a')
```

est bien formée, et sont type est booléen comme on le montre ici :

$$\overbrace{\texttt{(}\overbrace{\overbrace{\texttt{round(}\overbrace{\overbrace{\texttt{r}}^{\text{réel}}\texttt{+}\overbrace{\texttt{1}}^{\text{entier}}}^{\text{réel}}\texttt{)}}^{\text{entier}}\ \texttt{>}\ \texttt{(}\overbrace{\overbrace{\texttt{i}}^{\text{entier}}\ \texttt{/}\ \overbrace{\texttt{e}}^{\text{entier}}}^{\text{réel}}\texttt{))}}^{\text{booléen}}\ \texttt{or}\ \texttt{(}\overbrace{\overbrace{\texttt{c}}^{\text{car.}}\ \texttt{<}\ \overbrace{\texttt{'a'}}^{\text{car.}}}^{\text{booléen}}\texttt{)}}^{\text{booléen}}$$

<u>Remarque</u>
Le fait qu'une expression est bien formée n'implique pas que son évaluation est sans erreur, ce qui peut être le cas ici si `e` est nul.

5.3 Règles d'évaluation

L'expression `a + b * c` est évaluée `a + (b * c)` et non pas `(a + b) * c` : ceci parce que le `*` est *prioritaire* par rapport à `+`.

On classe les différents opérateurs par *ordre de priorité*, les opérateurs de plus forte priorité étant réalisés *avant* ceux de plus faible priorité.
Lorsque deux opérateurs sont de priorité égale, on évalue de gauche à droite. Par exemple `a + b - c` est évalué `(a + b) - c`, et non pas `a + (b - c)`.

Voici la table des priorités classées par ordre décroissant, les opérateurs sur une même ligne ayant une priorité égale.

`() fonction()`	primaire
`+ - not`	unaire
`* / div mod and`	multiplicatif
`+ - or`	additif
`= <> < <= >= >`	relation

Remarque

Est-ce que l'expression `a < b or c <= d` est bien formée ? Quel est son type ?

Réponse : non ! Ecrire une telle expression booléenne sans parenthèses est une erreur classique.

En effet dans la table de priorités, l'opérateur `or` a une priorité plus élevée que les opérateurs `<` et `<=`, et donc l'expression sera évaluée `a < (b or c) <= d` , ce qui est faux.

L'expression bien formée est ici `(a < b) or (c <= d)` .

6 Nouveaux types

On a vu les types pré-déclarés `boolean`, `integer`, `real` et `char`.

Nous verrons par la suite comment créer de nouveau types. Nous commençons par les plus simples, le type intervalle et le type énuméré.

6.1 Type intervalle

C'est un sous-ensemble de valeurs *consécutives* d'un type hôte.

Syntaxe `N..M`

où `N` et `M` sont des constantes du même type, et sont les bornes inférieures et supérieures de l'intervalle, `N` et `M` inclus.

Exemple

```
VAR
    pourcentage : 0 .. 100;        { le type hote est integer }
    digit       : '0' .. '9';      { le type hote est char    }
    reponse     : false .. true;   { le type hote est boolean }
```

Remarques

- ▷ Il faut impérativement que le type hôte soit codé sur *un entier* (signé ou non, sur un nombre de bits quelconque). On dit alors que ce type hôte est un *type ordinal.*
- ▷ Ainsi les types `integer`, `char` et `boolean` sont des types ordinaux.
- ▷ Seul un type ordinal admet les opérateurs `pred`, `succ` et `ord` (le précédent, le successeur et le numéro d'ordre dans le codage).
- ▷ Par contre le type `real` n'est pas ordinal, et donc on ne pas créer un type intervalle avec des réels, il n'y a pas de notion de « réels consécutifs ».
- ▷ Un autre cas de type non ordinal est le type `string` pour les chaînes de caractères, qui n'est pas codé sur *un* mais sur *plusieurs* entiers. On ne peut donc pas déclarer `'aaa'..'zzz'`.

Bonne habitude Utiliser des constantes nommées pour borner les intervalles : de la sorte on pourra consulter ces valeurs pendant le programme, et ces bornes ne seront écrites qu'une seule fois.

Exemple

```
CONST
    PMin = 0;
    PMax = 100;
VAR
    pourcentage : PMin .. PMax;
BEGIN
    writeln ('L''intervalle est ', PMin, ' .. ', PMax);
END.
```

6.2 Type énuméré

Il est fréquent en programmation que l'on aie à distinguer plusieurs cas, et que l'on cherche à coder le cas à l'aide d'une variable.

Exemple

```
VAR
    feux : 0..3;   { rouge, orange, vert, clignotant }
BEGIN
    { ... }
    if feux = 0
    then Arreter
    else if feux = 1
    then Ralentir
    else if feux = 2
    { ... }
END.
```

Ceci est très pratique mais dans un programme un peu long cela devient rapidement difficile à comprendre, car il faut se souvenir de la signification du code.

D'où l'intérêt d'utiliser un type *énuméré*, qui permet de donner un nom aux valeurs de code :

```
VAR
    feux : (Rouge, Orange, Vert, Clignotant);
BEGIN
    { ... }
    if feux = Rouge
    then Arreter
    else if feux = Orange
    then Ralentir
    else if feux = Vert
    { ... }
END.
```

- En écrivant cette ligne, on déclare en même temps :
  – la variable `feux`, de type énuméré (toujours codée sur un entier),
  – et les constantes nommées `Rouge`, `Orange`, `Vert` et `Clignotant`.

• À ces constantes sont attribuées les valeurs 0, 1, 2, 3 (la première constante prend toujours la valeur 0).

– On ne peut pas choisir ces valeurs soi-même, et ces identificateurs ne doivent pas déjà exister.

– L'intérêt n'est pas de connaître ces valeurs, mais d'avoir des noms explicites.

• Le type énuméré étant codé sur un entier, il s'agit d'un type ordinal et on peut :

– utiliser les opérateurs `pred`, `succ` et `ord` (exemple : `pred(Orange)` est `Rouge`, `succ(Orange)` est `Vert`, `ord(Orange)` est 1).

– déclarer un type intervalle à partir d'un type énuméré (exemple : `Rouge..Vert`).

6.3 Déclarer un type

Créer un type, c'est bien, mais le nommer, c'est mieux. On déclare les noms de types entre les mots clés `TYPE` et `VAR`.

Syntaxe

```
nom_du_type = type;
```

Exemple

```
TYPE
    couleurs_feux_t = (Rouge, Orange, Vert, Clignotant);
VAR
    feux : couleurs_feux_t;
```

De la sorte `couleurs_feux_t` est un nom de type au même titre que `integer` ou `char`.

Exemple complet

```
PROGRAM portrait;
CONST
    TailleMin = 50;   { en cm }
    TailleMax = 250;
TYPE
    taille_t   = TailleMin .. TailleMax;
    couleurs_t = (Blond, Brun, Roux, Bleu, Marron, Noir, Vert);
    cheveux_t  = Blond .. Roux;
    yeux_t     = Bleu .. Vert;
VAR
    taille_bob, taille_luc   : taille_t;
    cheveux_bob, cheveux_luc : cheveux_t;
    yeux_bob, yeux_luc       : yeux_t;
BEGIN
    taille_bob  := 180;
    cheveux_bob := Brun;
    yeux_bob    := Noir;
END.
```

Remarque Observez bien les conventions d'écriture différentes que j'ai employées pour distinguer les constantes des types et des variables ; cela aussi aide à la lecture.

6.4 Type enregistrement

Il s'agit simplement de regrouper des variables `V1`, `V2`, ... de différents types `T1`, `T2`, ... dans une variable « à tiroirs ».

<u>Syntaxe</u>

```
Record
    V1 : T1;
    V2 : T2;
    { ... }
End;
```

Soit `r` une variable de ce type ; on accède aux différents *champs* de `r` par `r.V1`, `r.V2`, ...

Reprenons l'exemple du programme `portrait`.

```
{ ... }
TYPE
    { ... }
    personne_t = Record
                   taille  : taille_t;
                   cheveux : cheveux_t;
                   yeux    : yeux_t;
                 End;
VAR
    bob, luc : personne_t;
BEGIN
    bob.taille  := 180;
    bob.cheveux := Brun;
    bob.yeux    := Noir;
    luc := bob;
END.
```

<u>Remarque</u> La seule opération globale sur un enregistrement est : recopier le contenu de `r2` dans `r1` en écrivant : `r2 := r1;`

Ceci est équivalent (et plus efficace) que de copier champ à champ ; en plus on ne risque pas d'oublier un champ.

Il y a une condition : les 2 variables doivent être exactement du même type.

<u>Intérêt de ce type</u>

Il permet de structurer très proprement des informations qui vont ensemble, de les recopier facilement et de les passer en paramètres à des procédures (on y reviendra).

<u>Remarque générale</u>

Lorsqu'on crée un type `T2` à partir d'un type `T1`, ce type `T1` doit *déjà exister* ; donc `T1` doit être déclaré *avant* `T2`.

II. Procédures

Une procédure est un sous-programme. Écrire des procédures permet de découper un programme en plusieurs morceaux.

Chaque procédure définit une *nouvelle instruction*, que l'on peut appeler en tout endroit du programme. On peut ainsi réutiliser le code d'un sous-programme.

Lorsqu'on découpe un problème en terme de procédures, puis qu'on implémente ces procédures, on fait ce qu'on appelle une *analyse descendante* : on va du plus général au détail.

1 Procédure sans paramètre

1.1 Principe

Il s'agit simplement de donner un nom à un groupe d'instructions.
Ensuite, l'appel de ce nom à divers endroits du programme provoque à chaque fois l'exécution de ce groupe d'instructions.

<u>Exemple</u>

```
PROGRAM exemple1;
VAR x, y, t : integer;

{ Declaration de la procedure Echange_xy }
PROCEDURE Echange_xy;
BEGIN
    { Corps de la procedure }
    t := x; x := y; y := t;
END;

BEGIN
    { Programme principal }
    x := 3; y := 4;
    writeln (x, ' ', y);
    Echange_xy;   { 1er appel de la procedure }
    writeln (x, ' ', y);
    Echange_xy;   { 2eme appel de la procedure }
    writeln (x, ' ', y);
END.
```

Ce programme affiche

```
3 4
4 3
3 4
```

<u>Remarques</u>

▷ Le nom de la procédure est un indentificateur.

▷ On déclare toute procédure *avant* le `BEGIN` du programme principal.

1.2 Appels

On peut très bien appeler une procédure `P1` depuis une procédure `P2`, mais il faut que la procédure `P1` aie été déclarée *avant* la procédure `P2`.
Exemple donnant le même résultat.

```
PROGRAM exemple2;
VAR x, y, t : integer;

PROCEDURE Affiche_xy;
BEGIN
    writeln (x, ' ', y);
END;

PROCEDURE Echange_xy;
BEGIN
    t := x; x := y; y := t;
    Affiche_xy;
END;

BEGIN
    x := 3; y := 4;
    Affiche_xy;
    Echange_xy;
    Echange_xy;
END.
```

Remarque
On peut aussi appeler une procédure depuis elle-même : c'est la *récursivité*, que l'on n'étudiera pas dans ce module.

1.3 Variables locales

Les objets du programme qui ne sont utiles que dans la procédure peuvent être définis dans les *déclarations locales* de la procédure.
Exemple Reprenons `exemple1` et changeons `t` :

```
PROGRAM exemple3;
VAR x, y : integer;

PROCEDURE Echange_xy;
VAR t : integer;   { Declaration locale }
BEGIN
    t := x; x := y; y := t;
END;

BEGIN
    { ... }
END.
```

– Une variable déclarée localement n'existe que pendant l'exécution de la procédure, et ne sert que à cette procédure.
– Le programme principal n'a jamais accès à une variable locale de procédure.
– Une procédure n'a jamais accès à une variable locale d'une autre procédure.
⟶ Améliore la lisibilité du programme.

1.4 Portée des variables

Les variables déclarées dans le `VAR` du programme principal sont appelées *variables globales*. Elles existent pendant toute la durée du programme et sont accessible de partout.

Une variable locale à une procédure `P`, portant le même nom `x` qu'une variable globale, *masque* la variable globale pendant l'exécution de `P`.

Exemple

```
PROGRAM exemple4;
VAR x : integer;

PROCEDURE Toto;
VAR x : integer;
BEGIN
    x := 4;
    writeln ('toto x = ', x);
END;

BEGIN
    x := 2;
    writeln ('glob x = ', x);
    Toto;
    writeln ('glob x = ', x);
END.
```

Ce programme affiche

glob $x = 2$
toto $x = 4$
glob $x = 2$

1.5 Effet de bord

Voici le scénario catastrophe :

- ▷ On est dans une procédure `P` et on veut modifier une variable `x` locale à `P`.
- ▷ Il existe déjà une variable globale ayant le même nom `x`.
- ▷ On oublie de déclarer la variable locale `x` au niveau de `P`.
- ▷ À la compilation tout va bien !
- ▷ À l'exécution, `P` modifie le `x` global alors que le programmeur ne l'avait pas voulu.
- ▷ Conséquence : le programme ne fait pas ce qu'on voulait, le `x` global a l'air de changer de valeur tout seul !

⟶ Erreur très difficile à détecter ; être très rigoureux et prudent !

2 Procédure paramétrée

2.1 Pseudo-passage de paramètres

Ecrivons une procédure `Produit` qui calcule $z = xy$.

```
PROGRAM exemple5;
VAR x, y, z, a, b, c, d : real;

PROCEDURE Produit;
BEGIN
    z := x * y;
END;
```

On veut se servir de `Produit` pour calculer $c = ab$ et $d = (a-1)(b+1)$.

```
BEGIN
    write ('a b ? '); readln (a, b);

    x := a; y := b;         { donnees }
    Produit;
    c := z;                 { resultat }

    x := a-1; y := b+1;     { donnees }
    Produit;
    d := z;                 { resultat }

    writeln ('c = ', c, ' d = ', d);
END.
```

Remarques

- ▷ L'écriture est un peu lourde.
- ▷ Il faut savoir que la procédure « communique » avec les variables `x`, `y`, `z`.
- ▷ Cela interdit de se servir de `x`, `y`, `z` pour autre chose que de communiquer avec la procédure ; sinon gare aux effets de bord !
- ▷ Deux sortes de paramètres : données et résultats.

2.2 Paramétrage

La solution élégante consiste à déclarer des *paramètres* à la procédure :

[Dire que c'est équiv à 2.1 ; mettre les progs côte à côte]

```
PROGRAM exemple5bis;
VAR a, b, c, d : real;

PROCEDURE Produit (x, y : real; var z : real);  { parametres }
BEGIN
    z := x * y;
END;

BEGIN
    write ('a b ? '); readln (a, b);

    Produit (a, b, c);                          { passage de }
    Produit (a-1, b+1, d);                      { parametres }

    writeln ('c = ', c, ' d = ', d);
END.
```

2.3 Comment ça marche

• À l'appel, on donne des paramètres dans les parenthèses, séparés par des virgules, et dans un certain *ordre* (ici `a` puis `b` puis `c`).

L'exécution de la procédure commence ; la procédure reçoit les paramètres et identifie chaque paramètre à une variable dans le même ordre (ici `x` puis `y` puis `z`).

[Dessiner des flêches `a` $\longrightarrow$ `x` , `b` $\longrightarrow$ `y` , `c` $\longrightarrow$ `z`]

• Les types doivent *correspondre* ; ceci est vérifié à la compilation.

• Il y a deux sorte de passage de paramètres : le passage *par valeur* et le passage *par référence.*

▷ Passage par valeur : à l'appel, le paramètre est une variable ou une expression. C'est la valeur qui est transmise, elle sert à initialiser la variable correspondante dans la procédure (ici `x` est initialisé à la valeur de `a` et `y` à la valeur de `b`).

▷ Passage par référence : à l'appel, le paramètre est une variable uniquement (jamais une expression). C'est l'adresse mémoire (la référence) de la variable qui est transmise, non sa valeur. La variable utilisée dans la procédure est en fait la variable de l'appel, mais sous un autre nom (ici `z` désigne la même variable (zone mémoire) que `a`).

C'est le mot-clé `var` qui dit si le passage se fait par valeur (pas de `var`) ou par référence (présence du var).

Pas de `var` = donnée ; présence du `var` = donnée/résultat. [dessiner une double flêche `c` $\longleftrightarrow$ `z`]

Erreurs classiques

▷ Mettre un `var` quand il n'en faut pas : on ne pourra pas passer une expression en paramètre.

▷ Oublier le `var` quand il en faut un : la valeur calculée ne pourra pas « sortir » de la procédure.

Exemples d'erreurs à l'appel de `Produit (a-1, b+1, d);`

```
PROCEDURE Produit (var x : real; y : real; var z : real);
```
ne compile pas à cause du paramètre 1, où une variable est attendue et c'est une expression qui est passée.

```
PROCEDURE Produit (x, y, z : real);
```
produit une erreur à l'exécution : `d` ne reçoit jamais le résultat `z` car il s'agit de 2 variables distinctes.

• Portée des variables :
dans `Exemple5bis`, les paramètres `x`, `y`, `z` de la procédure `Produit` sont des variables *locales* à `Produit`.

Leur *nom* n'est donc pas visible de l'extérieur de la procédure.
Attention : redéclarer un paramètre comme variable locale $\longrightarrow$ erreur à la compilation. Exemple :

```
PROCEDURE Produit (x, y : real; var z : real);
VAR
    t : real;    { déclaration d'une var locale : permis }
    x : real;    { redéclaration d'un paramètre : interdit }
BEGIN
    z := x * y;
END;
```

2.4 Bons réflexes

Le seul moyen pour une procédure de *communiquer* avec l'extérieur, c'est à dire avec le reste du programme, ce sont les variables globales et les paramètres.

Il faut toujours éviter soigneusement les effets de bords. Le meilleur moyen est de *paramétrer complètement* les procédures, et d'éviter la communication par variables globales.

Les variables de travail tels que compteur, somme partielle, etc doivent être locales à la procédure, surtout pas globale.

Prendre l'habitude de prendre des *noms de variables différents* entre le programme principal et les procédures : on détecte plus facilement à la compilation les effets de bords.

Chaque fois que l'on appelle une procédure, on *vérifie* particulièrement le bon ordre des paramètres et la correspondance des types. La compilation est très pointilleuse sur les types, mais par contre elle ne détecte pas les inversions de paramètres de même type.

III. Les instructions en Pascal

1 Instruction composée

Une instruction spécifie une opération ou un enchaînement d'opérations à exécuter sur des objets.

Les instructions sont séparées par des `;` et sont exécutées séquentiellement, c'est-à-dire l'une après l'autre, depuis le `BEGIN` jusqu'au `END.` final.

<u>Instruction déjà vues</u>

- `a := 5` *affectation*
- `writeln ('Bonjour')` *affichage*
- `readln (x)` *lecture*
- `ma_procedure (parametres)` *appel procédure*

<u>Plus généralement</u> Soient `I1`, `I2`, etc, des instructions.

- On fabrique dans la suite de nouvelles instructions :
  - `if expr then I1`
  - `while test do I1`
  - etc

- Il est possible de regrouper l'enchaînement `I1; I2; I3;` en une instruction *unique* en l'encadrant entre un `begin` et un `end`

```
begin I1; I2; I3; end
```

<u>Intérêt</u> On veut faire `I1` puis `I2` dans un `if`.

$$\overbrace{\texttt{if B then }\overbrace{\texttt{I1}}}\texttt{;}\overbrace{\texttt{I2}}\texttt{;} \qquad \overbrace{\texttt{if B then }\overbrace{\texttt{begin }\overbrace{\texttt{I1}}\texttt{;}\overbrace{\texttt{I2}}\texttt{;end}}}\texttt{;}$$

(on met des accolades sur les instructions).
⟶ Dans la $1^{ère}$ forme, `I2` ne fait pas partie du `if`, dans la 2^{nde} oui.

2 Les branchements

Étant donné une expression et plusieurs instructions, la valeur de l'expression va déterminer laquelle de ces instructions exécuter.

En Pascal il y a 2 types de branchements, le `if` et le `case`.

2.1 Le test booléen `if`

L'instruction ci-dessous prend 2 formes, elle signifie *si … alors … sinon*.

Syntaxe
```
if B then I1;
if B then I1 else I2;
```
`B` est une expression booléenne, `I1` et `I2` sont des instructions.

L'expression `B` est évaluée ; si elle est vraie, alors `I1` est exécutée, sinon `I2` est exécutée.

Remarque On peut se passer de `else` en n'employant que des `if then`, mais c'est moins efficace, et on peut facilement se tromper : l'exemple suivant ne donne pas les mêmes résultats !

```
                          a := 1;
{ sans else }                       { avec else }
if a = 1 then a := 2;               if a = 1 then a := 2
if a <> 1 then a := 3;                       else a := 3;
```

On peut imbriquer des `if then else` de différentes manières :

```
{ forme 1 }                          { forme 2 }
if B1                                if B1
then I1                              then if B2
else if B2                                then Ia
     then I2                              else Ib
     else if B3                      else if B3
          then I3                         then Ic
          else Iautre;                    else Id;
```

Règles

- ▷ Il n'y a jamais de `;` avant le `else` .
- ▷ Le `else` se rapporte toujours au dernier `then` rencontré.

Problème Dans la deuxieme forme, comment supprimer l'instruction `Ib` ?

On ne peut pas simplement supprimer la ligne `else Ib`, car alors le `else if B3` se rapporterait à `then Ia`.

On ne peut pas non plus rajouter un `;` car il y a un `else` après.

La solution consiste à « protéger » `if B2 then Ia;` dans un `begin end` :

```
if B1
then begin
        if B2
        then Ia;
     end
else if B3
     then Ic
     else Id;
```

Remarque Il faut faire très attention aux tests multiples, imbriqués ou non, et être très rigoureux dans l'écriture. La règle est d'indiquer entre `{}` le cas précis dans lequel on se trouve.

```
{ forme 1 }
if B1
then { B1 }
     I1
else if B2
then { !B1 et B2 }
     I2
else if B3
then { !B1 et !B2 et B3 }
     I3
else { !B1 et !B2 et !B3 }
     Iautre;
```

```
{ forme 2 }
if B1
then if B2
     then { B1 et B2 }
          Ia
     else { B1 et !B2 }
          Ib
else if B3
     then { !B1 et B3 }
          Ic
     else { !B1 et !B3 }
          Id;
```

2.2 Sélection de cas avec `case`

Syntaxe

```
case E of
     C1 : Ia;
     C2 : Ib;
     C3, C4 : Ic;   { liste }
     C5..C6 : Id;   { intervalle }
     { ... }
     else Iautre;   { en option }
end;
```

Cette instruction signifiant *choix selon* permet d'exécuter l'une des instructions `Ix` selon le cas `E`.

`E` est une expression ordinale (dont le type est un entier, un caractère, un booléen, ou un énuméré, mais pas un réel ni une chaîne de caractères). Les `Cx` sont des **constantes** ordinales du même type que `E`.

Comment ça marche `E` est évalué. Ensuite, est recherchée parmi les valeurs possibles `Cx`, laquelle est égale à `E`. L'instruction correspondante `Ix` est alors exécutée. Sinon, l'instruction après le `else` (s'il y en a un) est exécutée.

- On peut donner une liste de constantes, ou des intervalles de constantes.

  Attention, chaque valeur possible ne doit être représentée qu'une fois au plus (sinon il y a erreur à la compilation). Par exemple, on ne peut pas faire des intervalles se chevauchant, comme `3..6` et `5..10`, les cas `5` et `6` étant représentés 2 fois.

- L'exemple donné ci-dessus est équivalent à une forme en `if then else` imbriqués.

```
V := E; { evalue' une seule fois au debut }
if V = C1 then Ia
else if V = C2 then Ib
else if (V = C3) or (V = C4) then Ic
else if (V >= C5) and (V <= C6) then Id
else Iautre;
```

⟶ On préfère la forme avec le `case`, qui est plus lisible et plus efficace.

Exercice

Réécrire l'exemple sur les feux du §I.6.2, (*Type énuméré*) avec un `case`.

Exemple complet

Écrire un programme qui lit un caractère, puis classe ce caractère comme espace, lettre, digit ou autre.

```
PROGRAM caractere;
TYPE
    nat_t = (Espace, Lettre, Digit, Autre);
VAR
    nat : nat_t;    { nature }
    c   : char;
BEGIN
    write ('Rentrez un caractere :');
    readln(c);

    { analyse de c }
    case c of
      'a'..'z', 'A'..'Z', '_' :  nat := Lettre;
      '0'..'9' :                 nat := Digit;
      ' ' :                      nat := Espace;
      else                       nat := Autre;
    end; { case c }

    { affichage de nat }
    case nat of
      Espace : writeln ('Espace');
      Lettre : writeln ('Lettre');
      Digit  : writeln ('Digit');
      Autre  : writeln ('Autre');
      else { case nat }
               writeln ('Erreur case nat : ', ord(nat), ' non prevu');
    end; { case nat }
END.
```

Bonnes habitudes

- ▷ Après le `else` et le `end`, marquer en commentaire qu'ils se rapportent au `case`.
- ▷ Faire afficher un message d'erreur après le `else` : aide à la mise au point du programme.

3 Les boucles

3.1 La boucle `while`

Cette instruction signifie *tant que*. Elle permet de répéter l'exécution d'une instruction de boucle `I` :

Syntaxe

```
while B do I;
```

B est une expression booléenne.

(∗) B est évaluée. Si B est vraie, alors I est exécutée, et on recommence depuis (∗).

Remarques

- ▷ Les variables de l'expression B doivent être initialisées *avant* le `while`, pour que au premier passage B puisse être évalué.
- ▷ Le `while` continue de boucler tant que B n'est pas faux. Pour éviter une boucle infinie, qui « plante » le programme, il faut **obligatoirement** que dans I il y aie une sous-instruction rendant B faux à un moment donné.

Exemple Programme calculant la somme des nombres de 1 à 100.

```
PROGRAM Somme;
VAR
    s, k : integer;
BEGIN
    s := 0; k := 1;
    while k <= 100 do
    begin
        s := s + k;
        k := k + 1;
    end;
    writeln (s);
END.
```

- On se sert souvent d'un booléen dans une boucle `while` :

```
continuer := true;
while (k <= 100) and continuer do
begin
  { ... }
  if ( ... ) then continuer := false;
end;
```

3.2 La boucle repeat

Cette instruction signifie *répéter . . . jusqu'à* . Elle permet comme le `while` de répéter l'exécution d'une instruction de boucle I :

Syntaxe

```
repeat I; until B;
```

B est une expression booléenne.

(∗) I est exécutée, puis B est évaluée. Si B est vraie, alors on s'arrête, sinon on recommence depuis (∗).

Différences avec `while`

▷ L'instruction `I` est exécutée au moins une fois.

▷ Le test `B` étant évalué *après* `I`, `B` peut être affecté *dans* `I`. Pour le `while` il faut avoir initialisé `B` *avant*.

▷ Pas besoin d'encadrer un groupe d'instructions par un `begin end`, le `repeat until` joue déjà ce rôle.

Exemple Le `while` de `Somme` s'écrit avec un `repeat` :

```
s := 0; k := 1;
repeat s := s + k; k := k + 1; until k > 100;
```

- Traduction d'une boucle `while B do I;` avec un `repeat` :

```
if B then
  repeat
    I;
  until not B;
```

- On se sert souvent d'un booléen dans une boucle `repeat` :

```
repeat
  { ... }
  arreter := ... ;
until (k > 100) or arreter;
```

3.3 La boucle `for`

Cette instruction signifie *pour*. Elle permet de répéter l'exécution d'une *instruction de boucle* `I` :

Syntaxe

```
for k := E1 to E2 do I;
```

`k` est le compteur de boucle, `E1` et `E2` sont les bornes inférieures et supérieures.
`E1` et `E2` sont des expressions ordinales, du même type que la variable `k`.

`E1` et `E2` sont d'abord évaluées, puis `k` prend la valeur `E1`. (∗) Si `k <= E2`, alors `I` est exécutée, puis `k` est incrémenté de 1, et on recommence depuis (∗).

- Pour avoir une boucle décroissante, on écrit

```
for k := E2 downto E1 do I;
```

- On peut écrire une boucle `for k := E1 to E2 do I;` avec un `while` :

```
k := E1;   { init de k }
m := E2;   { on evalue E2 une fois pour toutes }
while k <= m do
begin
  I;
  k := k+1;
end;
```

• On en déduit l'écriture d'une boucle `for k := E1 to E2 do I;` avec un `repeat` :

```
k := E1;     { init de k }
m := E2;     { on evalue E2 une fois pour toutes }
if k <= m then
  repeat
    I;
    k := k+1;
  until k > m;
```

Remarques

▷ L'instruction de boucle `I` n'est pas exécutée du tout si `E1 > E2`.

▷ Modifier pendant la boucle la valeur de `E1` ou `E2` n'a pas d'effet.

▷ Il est totalement interdit de modifier la valeur du compteur `k` dans le corps de la boucle.

▷ L'incrément de 1 n'est pas modifiable (contrairement au Basic avec `step`).

▷ À la fin de l'exécution de la boucle, la variable `k` redevient **indéterminée** : elle a une valeur qui dépend du compilateur. Par exemple sous Delphi, elle vaut `E2+1`, et sous Turbo Pascal 7.0, elle vaut `E2`.

Exemple d'application des règles : dire la valeur affichée [c'est 10240]

```
a := 5;
for i := a to a+10 do a := a*2;
writeln(a);
```

Exemple Le `while` de `Somme` s'écrit avec un `for` :

```
s := 0;
for k := 1 to 100 do s := s + k;
```

• On peut bien entendu imbriquer des boucles.

```
PROGRAM table_multiplication;
VAR
    i, j : integer;
BEGIN
    for i := 1 to 10 do
    begin
      for j := 1 to 10 do write (i*j : 3);
      writeln;
    end;
END.
```

Variante

```
for i := 1 to 10 do
  for j := 1 to 10 do
  begin
    write (i*j : 3);
    if j = 10 then writeln;
  end;
```

3.4 Choix de la boucle

La règle est simple (l'apprendre par cœur) :

Si le nombre d'itérations est connu *a priori*, alors on utilise un `for`.
Sinon : on utilise le `repeat` (quand il y a toujours au moins une itération), ou le `while` (quand le nombre d'itérations peut être nul).

IV. Fonctions

Une fonction est une procédure qui renvoie un résultat, de manière à ce qu'on puisse l'appeler dans une expression.

Exemples `y := cos(x) + 1; c := chr(x + ord('0'));`

1 Fonction sans paramètre

1.1 Principe

Syntaxe

```
FUNCTION nom_fonction : type_resultat;
BEGIN
    { ... corps de la fonction ... }

    { Résultat de la fonction, du type type_resultat }
    nom_fonction := expression;
END;
```

La nouveauté par rapport à une procédure est que l'on « sort » le résultat de la fonction `nom_fonction` en écrivant une affectation sur son nom.

Attention

• `nom_fonction` n'est pas une variable, et à l'intérieur de la fonction il ne faut surtout pas l'utiliser dans une expression, car cela provoquerait un appel récursif.

• Une fonction doit toujours avoir un résultat (i.e on ne peut pas le laisser indéterminé).

1.2 Appel

```
PROGRAM ex1;
VAR x : type_resultat;
{ ici déclaration de la fonction }
BEGIN
    { appel fonction et stockage du résultat dans x }
    x := nom_fonction;
END.
```

1.3 Variables locales

```
FUNCTION nom_fonction : type_resultat;
VAR locales : types_locales;
BEGIN
    { ... }
    nom_fonction := expression;   { du type type_resultat }
END;
```

Bonne habitude

Passer par une variable locale `res` : on fait ce qu'on veut de `res` dans la fonction, et à la fin de la fonction on écrit `nom_fonction := res;`

```
FUNCTION nom_fonction : type_resultat;
VAR res : type_resultat;
BEGIN
    { ... dans le corps, on fait ce qu'on veut de res ...}

    { on dit que le résultat est res }
    nom_fonction := res;
END;
```

2 Fonction avec paramètres

Syntaxe

```
FUNCTION nom_fonction ( parametres : types_params ) : type_resultat;
VAR locales : types_locales;
    res : type_resultat;
BEGIN
    { ... }
    nom_fonction := res;
END;
```

Tout ce que l'on a dit sur le paramétrage des procédures reste valable pour les fonctions.

2.1 Procédure vs fonction

Exemple du produit.

```
                        PROGRAM exemple5ter;
                        VAR a, b, c, d : real;

PROCEDURE Produit (x, y : real;  |  FUNCTION Produit (x, y : real) : real;
                  var z : real); |  VAR res : real;
BEGIN                            |  BEGIN
    z := x * y;                  |      res := x * y;
END;                             |      Produit := res;
                                 |  END;

                        BEGIN
                            write ('a b ? '); readln (a, b);

    Produit (a, b, c);           |      c := Produit (a, b);
    Produit (a-1, b+1, d);       |      d := Produit (a-1, b+1);

                            writeln ('c = ', c, ' d = ', d);
                        END.
```

2.2 Passage de types enregistrement

Exemple On veut savoir si un couple d'amis est assorti. On fixe les règles suivantes : le couple est assorti si ils ont moins de 10 ans d'écart, ou si le mari est agé et riche.

```
PROGRAM assorti;
TYPE
    humain_t = Record
                 age, taille : integer;
                 riche       : boolean;
               End;
    couple_t = Record
                 homme, femme : humain_t;
                 nb_enfant    : integer;
               End;

FUNCTION difference_age (h, f : humain_t) : integer;
VAR res : integer;
BEGIN
    res := abs (h.age - f.age);
    difference_age := res;
END;

FUNCTION couple_assorti (c : couple_t) : boolean;
VAR res : boolean;
BEGIN
    res := false;
    if difference_age (c.homme, c.femme) < 10 then res := true;
    if (c.homme.age > 75) and c.homme.riche then res := true;
    couple_assorti := res;
END;

VAR amis : couple_t;
BEGIN
    { ... }
    write ('Ce couple avec ', amis.nb_enfant, ' enfant(s) est ');
    if couple_assorti (amis) then writeln ('assorti.')
                             else writeln ('non assorti.');
END.
```

3 Fonction avec plusieurs résultats

Il est fréquent que l'on écrive une fonction qui renvoie un booléen qui dit si tout s'est bien passé, tandis que les vrais résultats sont passés dans les paramètres.

Exemple Une fonction qui prend une lettre, la met en majuscule ou renvoie une erreur si le caractère n'est pas une lettre.

```
FUNCTION maj_lettre (     lettre : char;
                      var maj    : char  ) : boolean;
VAR res : boolean;
BEGIN
    { init }
    maj := lettre;
    res := true;   { pas d'erreur }
```

```
    case lettre of
      'a' .. 'z' : maj := chr(ord(lettre) - ord('a') + ord('A'));
      'A' .. 'Z', '_' : ; { rien }
      else res := false;
    end; { case lettre }

    maj_lettre := res;
END;
```

L'appel de cette fonction :

```
VAR c, m : char;
BEGIN
    readln (c);
    if maj_lettre (c,m)
    then writeln ('La majuscule de ', c, ' est ', m)
    else writeln ('Le caractère ', c, ' n''est pas une lettre');
END.
```

Autre avantage : on fait tous les affichages et messages d'erreur *en dehors* de la fonction.

<u>Exemple</u> [non vu en cours faute de temps]

On veut calculer $\sum_{i=a}^{i=b} \frac{i+k}{\cos(i-k)}$ or il risque d'y avoir des divisions par 0.

On écrit d'abord une fonction `calc` qui renvoie un booléen qui dit si le calcul de $\frac{x+y}{\cos(x-y)}$ a pu se faire, tandis que le résultat numérique est passé en paramètre `z`.

```
FUNCTION calc (     x : integer;
                    y : real;
                var z : real  ) : boolean;
VAR ok : boolean;
    d  : real;
BEGIN
    ok := true; { init pas d'erreur }

    d := cos (x-y);
    if d = 0.0
    then ok := false     { division par 0 }
    else z := (x+y) / d; { resultat numerique }

    calc := ok;
END;
```

On écrit ensuite une fonction `somme` qui appelle `calc`.

```
FUNCTION somme ( a, b : integer;
                 k    : real    ) : real;
VAR res, f : real;
    i      : integer;
BEGIN
    res := 0.0; { init somme a 0 }
```

```
        for i := a to b do
          if calc (i, k, f)
            then res := res + f;

        somme := res;
    END;
```

L'appel de cette fonction :

```
    VAR ga, gb : integer;
        gk, gs : real;
    BEGIN
        readln (ga, gb, gk);
        gs := somme (ga, gb, gk);
        writeln (gs);
    END.
```

<u>Exercice</u>

– Modifier la fonction `somme` pour que elle renvoie un booléen disant que tous les calculs de la somme ont pu se faire, tandis que le résultat numérique est passé en paramètre.
– Adapter le programme principal appelant `somme`.

4 Gestion d'erreurs

[non vu en cours, tombe un peu à plat ..]

On veut généraliser l'usage de fonctions renvoyant un code d'erreur, et dont les résultats sont passés en paramètres.

– Soient `F1`, `F2`, etc, de telles fonctions renvoyant un booléen.
– Soit `ok` un booléen.
– Soient `I1`, `I2`, etc, des instructions.

Considérons la séquence d'instruction suivante

```
    I1;
    ok := F1 ( ... );
    I2;
    ok := F2 ( ... );
    I3;
    ok := F3 ( ... );
    I4;
    { ... }
```

On veut exécuter ce traitement, mais l'interrompre dés qu'il y a une erreur.

On devrait normalement écrire :

```
I1;
if F1 ( ... )
then begin
       I2;
       if F2 ( ... )
       then begin
              I3;
              if F3 ( ... )
              then begin
                     I4;
                     { ... }
                   end;
            end;
     end;
```

C'est lourd, on se perd rapidement dans tous ces `begin end`.
Il est beaucoup simple d'écrire

```
I1;
ok := F1 ( ... );

if ok then
begin
  I2;
  ok := F2 ( ... );
end;

if ok then
begin
  I3;
  ok := F3 ( ... );
end;

if ok then
begin
  I4;
  { ... }
end;
```

Dés que `ok` est faux, plus aucun bloc suivant n'est exécuté.

V. Tableaux

Les tableaux permettent de manipuler plusieurs informations de même type, de leur mettre un indice : la 1ère info, la 2ème info, ..., la i^{ème} info, ...

Ils sont stockés en mémoire centrale comme les autres variables, contrairement aux fichiers qui sont stockés sur le disque.

Une propriété importante des tableaux est de permettre un accès direct aux données, grâce à l'indice.

On appelle souvent *vecteur* un tableau en une dimension.

1 Le type `array`

1.1 Principe

Syntaxe

```
array [ I ] of T
```

`I` étant un type intervalle, et `T` un type quelconque.

Ce type définit un tableau comportant un certain nombre de cases de type `T`, chaque case est repérée par un indice de type `I`.

Exemple

```
TYPE vec_t = array [1..10] of integer;
VAR  v : vec_t;
```

`v` est un tableau de 10 entiers, indicés de 1 à 10.

indice :	1	2	3	4	5	6	7	8	9	10
case mémoire :										

▷ À la déclaration, le contenu du tableau est indéterminé, comme toute variable.

▷ On accède à la case indice `i` par `v[i]` (et non `v(i)`).

▷ Pour mettre toutes les cases à 0 on fait

```
for i := 1 to 10 do v[i] := 0;
```

Remarque L'intervalle du `array` peut être de tout type intervalle, par exemple `1..10`, `'a'..'z'`, `false..true`, ou encore un intervalle d'énumérés `Lundi..Vendredi`.

On aurait pu déclarer `vecteur` comme ceci (peu d'intérêt) :

```
TYPE interv = 1..10 ; vec_t = array [ interv ] of integer;
```

1.2 Contrôle des bornes

Il est en général conseillé de repérer les bornes de l'intervalle avec des constantes nommées : si on décide de changer une borne, cela est fait à un seul endroit dans le programme.

L'écriture préconisée est donc

```
CONST vec_min = 1; vec_max = 10;
TYPE  vec_t = array [vec_min..vec_max] of integer;
```

Règle 1

Il est totalement interdit d'utiliser un indice en dehors de l'intervalle de déclaration, sinon on a une erreur à l'exécution.

Il faut donc être très rigoureux dans le programme, et ne pas hésiter à tester si un indice `i` est correct *avant* de se servir de `v[i]`.

Exemple Programme demandant à rentrer une valeur dans le vecteur.

```
CONST vec_min = 1; vec_max = 10;
TYPE  vec_t = array [vec_min..vec_max] of integer;
VAR   v : vect_t; i : integer;
BEGIN
    write ('i ? '); readln(i);
    if (i >= vec_min) and (i <= vec_max)
    then begin
           write ('v[', i, '] ? '); readln(v[i]);
         end
    else writeln ('Erreur, i hors intervalle ',
                  vec_min, '..', vec_max);
END.
```

Règle 2 Le test d'un indice `i` et de la valeur en cet indice `v[i]` dans la même expression sont interdits.

Exemple

```
if (i >= vec_min) and (i <= vec_max) and (v[i] <> -1) then ...
                                                       else ...;
```

Une expression est toujours évaluée en intégralité ; donc si `(i <= vec_max)`, le test `(v[i] <> -1)` sera quand même effectué, alors même que l'on sort du vecteur ! Solution : séparer l'expression en 2.

```
if (i >= vec_min) and (i <= vec_max)
then if (v[i] <> -1) then ...
                     else ...
else ... { erreur hors bornes } ;
```

1.3 Recopie

En Pascal, la *seule* opération globale sur un tableau est : recopier le contenu d'un tableau `v1` dans un tableau `v2` en écrivant : `v2 := v1;`

Ceci est équivalent (et plus efficace) que
`for i := vec_min to vec_max do v2[i] := v1[i];`

Il y a une condition : les 2 tableaux doivent être *exactement* de mêmes types, i.e issus de la *même déclaration*.

```
TYPE
    vecA = array [1..10] of char;
    vecB = array [1..10] of char;
VAR
    v1 : vecA; v2 : vecA; v3 : vecB;
BEGIN
    v2 := v1;  { legal car meme type vecA }
    v3 := v1;  { illegal, objets de types <> vecA et vecB }
```

2 Super tableaux

Quelques types un peu plus complexes à base de tableaux, et de combinaisons entre types.

2.1 Tableaux à plusieurs dimensions

Exemple : dimension 1 = vecteur ; dimension 2 = feuille excel ; dimension 3 = classeur excel [faire petit schéma].

On peut créer des tableaux à plusieurs dimensions de plusieurs manières :

Faire des schémas

- `v1 : array [1..10] of array [1..20] of real`
  ⟶ Tableau de 10 éléments, chaque élément étant un tableau de 20 réels.
  On accède à l'élément d'indice `i` dans `1..10` et `j` dans `1..20` par `v1[i][j]`.
- `v2 : array [1..10, 1..20] of real`
  ⟶ Tableau de 10×20 réels.
  On accède à l'élément d'indice `i` dans `1..10` et `j` dans `1..20` par `v2[i,j]`.

Exemple Mise à 0 du tableau `v2`.

```
VAR
    v2 : array [1..10, 1..20] of real;
    i, j : integer;
BEGIN
    for i := 1 to 10 do
      for j := 1 to 20 do
        v2[i,j] := 0.0;
END.
```

2.2 Tableaux de `record`

On peut créer des tableaux d'enregistrements, et des enregistrements qui contiennent des tableaux.

```
PROGRAM Ecole;
CONST
    MaxEleves = 35;
    MaxNotes  = 10;
TYPE
    note_t = array [1..MaxNotes] of real;

    eleve_t = Record
                age, nb_notes : integer;
                notes : note_t;
                moyenne : real;
              End;
    classe_t = array [1..MaxEleves] of eleve_t;
VAR
    c : classe_t;
    nb_eleves, i, j : integer;
BEGIN
    { ... }
    for i := 1 to nb_eleves do
    begin
      writeln ('Eleve n.', i);
      writeln ('  age   : ', c[i].age);
      write   ('  notes :');
      for j := 1 to c[i].nb_notes do write (' ', c[i].notes[j]);
      writeln;
      writeln ('  moy   : ', c[i].moyenne);
    end;
END.
```

- On a comme d'habitude le droit de faire une copie globale entres variables du même type :

```
VAR c1, c2 : classe_t;
    e : eleve_t;  i, j : integer;
BEGIN
    { copie globale de type classe_t }
    c2 := c1;
    { échange global de type eleve_t }
    e := c1[i]; c1[i] := c1[j]; c1[j] := e;
END.
```

- Exemple de passages de paramètres : on écrit une procédure affichant un `eleve_t`.

```
PROCEDURE affi_eleve (e : eleve_t);
VAR j : integer;
BEGIN
    writeln ('  age   : ', e.age);
    write   ('  notes : ');
    for j := 1 to e.nb_notes do write (e.notes[j]);
    writeln;
    writeln ('  moy   : ', e.moyenne);
END;
```

```
BEGIN
    { ... }
    for i := 1 to nb_eleves do
    begin
      writeln ('Eleve n.', i);
      affi_eleve (c[i]);
    end;
END.
```

`affi_eleve(e)` ne connait pas le numéro de l'élève ; l'appelant, lui, connait le numéro, et l'affiche avant l'appel.

On peut encore écrire une procédure `affi_classe` :

```
PROCEDURE affi_classe (c : classe_t ; nb : integer);
VAR i : integer;
BEGIN
    for i := 1 to nb do
    begin
      writeln ('Eleve n.', i);
      affi_eleve (c[i]);
    end;
END;

BEGIN
    { ... }
    affi_classe (c, nb_eleves);
END.
```

3 Le type string

On code une chaîne de caractère telle que `'bonjour'` dans un objet de type `string`.

3.1 Principe

Syntaxe `string [m]`
où `m` est une constante entière donnant le nombre maximum de caractères pouvant être mémorisés. Exemple :

```
VAR s : string[80];
BEGIN
    s := 'Le ciel est bleu.';
    writeln (s);
END.
```

Codage
Ayant déclaré `s : string[80]`, comment sont codés les caractères ?
En interne, Pascal réserve un `array [0..80] of char`.
Le premier caractère est `s[1]`, le deuxième est `s[2]`, etc.
La longueur courante de la chaîne est codé dans la case 0 (⟶ `ord(s[0])`).

Remarque
– Affecter une chaîne plus longue que l'espace réservé à la déclaration est une erreur.
– Comme la longueur courante est codée sur un `char`, elle est limitée à 255.

3.2 Opérateurs sur les strings

`a := ''`	Chaîne vide (longueur 0).
`a := b`	Recopie de `b` dans `a`.
`a := c + d`	Concaténation en une seule chaîne. `c` et `d` de types `string` ou `char`; le résultat est un `string`.
`length(a)`	Longueur courante de `a`, résultat entier.

```
CONST  Slogan = 'lire la doc';
VAR    s1, s2 : string[100];
       i : integer;
BEGIN
    s1 := 'veuillez ';
    s2 := s1 + Slogan;
    writeln ('s2 = ''', s2, '''');
    writeln ('Longueur courante de s2 : ', length(s2) );
    write ('Indices des ''l'' dans s2 : ');
    for i := 1 to length(s2) do
      if s2[i] = 'l' then write(i, ' ');
    writeln;
END.
```

Comparaison entre 2 `string` : les opérateurs `=`, `<>`, `<`, `>`, `<=`, `>=`, sont utilisables, et le résultat est un booléen.

La comparaison se fait selon l'ordre lexicographique du code ASCII.

Exemple Soit `b` un booléen; `b` est-il vrai ou faux ?

```
b := 'A la vanille' < 'Zut';     { vrai }
b := 'bijou' < 'bidon';          { faux, c'est > car 'j' > 'd' }
b := 'Bonjour' = 'bonjour';      { faux, c'est < car 'B' < 'b' }
b := ' zim boum' > 'attends !';  { faux, c'est < car ' ' < 'a' }
```

Exercice On considère le type `LongString` suivant.

```
CONST longStringMax = 4096;
TYPE  LongString = record
                     c : array [1..LongStringMax] of char;
                     l : interer; { longueur courante }
                   end;
```

Écrire les procédure et fonctions suivantes :

```
FUNCTION longueur (s1 : LongString ) : integer;
FUNCTION est_inferieur (s1, s2 : LongString ) : boolean;
FUNCTION est_egal (s1, s2 : LongString ) : boolean;
PROCEDURE concatene (s1, s2 : LongString ; var s3 : LongString);
```

VI. Fichiers séquentiels

Les entrées/sorties dans un ordinateur sont la communication d'informations entre la mémoire de l'ordinateur et ses périphériques (disques, clavier, écran, imprimante, etc).

Les entrées/sorties se font par le biais de *fichiers séquentiels.*

Un fichier séquentiel est une collection de données de même type (souvent de caractères), dans laquelle les données ne peuvent être lues ou écrites que les unes après les autres, en commençant par le début et sans retour possible en arrière.

Un fichier peut être vide ; il peut avoir une fin ou non ; il peut être ouvert (accessible) ou fermé ; une lecture peut être « en attente ».

1 Le clavier et l'écran

Le clavier et l'écran sont gérés comme des fichiers particuliers : ce sont des fichiers texte, toujours ouverts et sans fin ; ils sont désignés par les variables prédéfinies `input` et `output` (dont on ne se sert quasiment jamais).

1.1 Affichage avec `write`

La procédure `write()` permet d'afficher un ou plusieurs paramètres. `writeln()` fait la même chose puis rajoute un saut de ligne.

⟶ Trois écritures équivalentes :

```
writeln (a, b, c, d);
write (a, b, c, d); writeln;
write(a); write(b); write(c); write(d); writeln;
```

• Le résultat de l'affichage dépend du type du paramètre :

```
VAR e : integer; c : char; b : boolean; r : real; s : string[32];
BEGIN
    e := 12; c := 'A'; b := true; r := 23.0; s := 'toto';
    writeln (e, '|', c, '|', b, '|', r, '|', s);
END.
```

affiche : `12|A|TRUE|2.300000E+01|toto`

Formatter l'impression des variables

1) Soit `v` un entier, un booléen, un caractère ou un string.
`write(v:8)` dit à `write` d'afficher `v` sur au moins 8 caractères.
Si le nombre de caractères (signe éventuel compris) est > 8, `v` est complètement affiché ; si il est < 8, des espaces sont rajoutés à gauche pour compléter.

Ainsi `writeln (e:5, '|', c:3, '|', b:5, '|', s:6);`
affiche : `␣␣␣12|␣␣A|␣TRUE|␣␣toto`

2) Soit `r` un réel.

`write(r:10);` dit à `write` d'afficher `r` en notation scientifique, sur au moins 10 caractères, signes de la mantisse et de l'exposant compris.

Cette fois c'est d'abord le nombre de chiffres après la virgule qui change de 1 à 10, puis au besoin des espaces sont ajoutés à gauche.

De plus le dernier chiffre de la mantisse affichée est arrondi.

```
r := 2 / 3;
writeln (r:8, '|', r:10, '|', r:18 );
```

affiche : `␣6.7E-01|␣6.667E-01|␣␣6.6666666667E-01`

3) Autre formattage de `r` réel.

`write(r:8:4);` dit à `write` d'afficher `r` en notation simple, sur au moins 8 caractères, dont 4 chiffres après la virgule (le dernier étant arrondi).

Ainsi `writeln (r:8:4);`
affiche : `␣␣0.6667`

<u>Bilan</u>

- ▷ Le formattage permet d'aligner des chiffres.
- ▷ Ne pas oublier de mettre des espaces autour des variables pour que le résultat ne soit pas tout aglutiné et illisible.
- ▷ On ne peut afficher que des types simples.

1.2 Lecture avec `read`

La procédure `read()` permet de lire un ou plusieurs paramètres. `readln()` fait la même chose puis fait un `readln;`

⟶ Trois écritures équivalentes :

```
readln (a, b, c, d);
read (a, b, c, d); readln;
read(a); read(b); read(c); read(d); readln;
```

<u>Remarques</u>

- ▷ À l'exécution d'une de ces lignes, on peut rentrer les données en les séparant par des espaces, des tabulations ou des retours chariot ↩ .
- ▷ Il faut que les données lues correspondent au type attendu de chaque variable, sinon il y a une erreur à l'exécution.

• Le comportement de `read()` et `readln;` étant complexe, regardons plus en détail se qui se passe à l'exécution.

▷ L'utilisateur tape une série de caractères, avec de temps à autres des retours chariot $\hookleftarrow$.

▷ Pendant la frappe, les caractères sont stockés dans un *buffer* (une mémoire tampon) ; à chaque $\hookleftarrow$, le contenu du buffer est envoyé au programme Pascal (y compris le $\hookleftarrow$).

▷ De son côté, `read(v);` lit une donnée dans le buffer, ou attend le buffer suivant. Le `read(v);` attend donc quand

- on n'a pas tapé de $\hookleftarrow$,
- ou qu'on a tapé une ligne vide,
- ou que toutes les données dans le buffer ont déjà été lues.

▷ `readln;` attend le prochain $\hookleftarrow$, puis vide le buffer. Attention les données non lues dans le buffer sont alors perdues pour de futurs `read()`.

Exemple

```
VAR a, b, c, d : integer;
BEGIN
    readln (a, b);  { = read(a); read(b); readln; }
    readln (c, d);
    writeln ('Lu : ', a, ' ', b, ' ', c, ' ', d);
END.
```

Essais :	1 $\hookleftarrow$ 2 $\hookleftarrow$ 3 $\hookleftarrow$ 4 $\hookleftarrow$ `Lu : 1 2 3 4`	1 2 $\hookleftarrow$ 3 4 $\hookleftarrow$ `Lu : 1 2 3 4`	1 2 3 $\hookleftarrow$ 4 5 6 $\hookleftarrow$ `Lu : 1 2 4 5` *3 et 6 perdus*

Le même programme avec `read (a, b, c, d); readln;`

produit :	1 $\hookleftarrow$ 2 $\hookleftarrow$ 3 $\hookleftarrow$ 4 $\hookleftarrow$ `Lu : 1 2 3 4` *idem*	1 2 $\hookleftarrow$ 3 4 $\hookleftarrow$ `Lu : 1 2 3 4` *idem*	1 2 3 $\hookleftarrow$ 4 5 6 $\hookleftarrow$ `Lu : 1 2 3 4` *5 et 6 perdus*

Remarque

Le type des objets lus a une grande importance pour `read()`.

Dans notre programme exemple, voici ce qui se passe si on déclare les 4 variables en `char` :

`VAR a, b, c, d : integer;`	`VAR a, b, c, d : char;`
1 2 3 4 $\hookleftarrow$ `Lu : 1 2 3 4`	1 2 3 4 $\hookleftarrow$ `Lu : 1 ␣ 2 ␣`

Remarque

On ne peut faire un `read()` que sur un entier, un réel, un caractère ou une chaîne de caractères ; on ne peut pas faire un `read()` sur un booléen.

<u>Algorithme de lecture</u> d'une suite de caractères tapés au clavier, se terminant par un `'.'`

Option on affiche le code ASCII de chaque caractère.

```
CONST CarFin = '.';
VAR c : char;
BEGIN
    read(c);                    { premier caractère }
    while c <> CarFin do
    begin
      writeln (c, ' ', ord(c)); { option }
      read(c);                  { caractère suivant }
    end;                        {   lu à la fin du while  }
    readln;                     { vide buffer et retour chariot }
END.
```

Exécution :

Salut ↩	t 116	C 67
S 83	13	i 105
a 97		a 97
l 108	10	o 111
u 117	Ciao.bye ↩	

2 Fichiers de disque

Les fichiers de disque permettent de stocker des informations de manière permanente, sur une disquette ou un disque dur.

Ces informations persistent même lorsque l'ordinateur est éteint.

L'inconvénient est que ces données ne sont pas en mémoire vive ; on n'y accède pas *directement*, comme c'est le cas avec un vecteur.

En fait on va *lire* ou *écrire* des données une à une sur le disque, étant donné qu'il s'agit de fichiers séquentiels.

2.1 Notions générales

Sur un disque, un fichier a un *nom*, par exemple

`'a :\mass\tp4.pas'`

On peut coder ce nom dans un string, par exemple `nomf`.

Dans un programme qui doit manipuler ce fichier, il faut une *variable* pour le désigner, par exemple `f`.

<u>Déroulement des opérations</u>

a) Déclarer la variable `f`

`f : text ;` ou `f : file of qqchose ;`

b) Assigner la variable `f` au fichier de nom `nomf`

`assign (f, nomf) ;`

c) Ouvrir le fichier `f` pour pouvoir y lire ou y écrire les données

`reset (f) ;` ou `rewrite (f) ;`

d) Lire ou écrire des données

`read (f, donnee) ;` ou `write (f, donnee) ;`

e) Quand on a fini, on ferme le fichier

`close (f) ;`

<u>Lecture *ou* écriture</u>

On ouvre un fichier soit en lecture, soit en écriture. On ne peut pas faire les deux en même temps.

- ▷ En lecture : on fait `reset(f);` puis des `read(f, ...);`
  Si le fichier n'existe pas, il y a une erreur.
- ▷ En écriture : on fait `rewrite(f);` puis des `write(f, ...);`
  Si le fichier n'existe pas, un `rewrite` le crée. Si il existe déjà, le `rewrite` l'*écrase*, c'est-à-dire que l'ancien contenu est définitivement perdu.

<u>Fin du fichier</u>

En lecture, avant de faire un `read`, il faut tester si il y a encore quelque chose à lire ; on n'a pas le droit de faire un `read` si la fin du fichier est atteinte.

La fonction `eof(f)` retourne `true` si la fin du fichier est atteinte.

<u>Deux familles de fichiers</u>

On distingue les fichiers de texte des fichiers d'éléments.

2.2 Fichiers de texte

Les fichiers de textes sont les fichiers que vous pouvez éditer, comme par exemple vos fichiers pascal.

<u>Déclaration</u>

```
VAR
    f : text;
    c : char;
    s : string[255];
    x : integer;
    r : real;
```

<u>Lecture</u>

`read (f, c);` lit un caractère dans `f`.
`readln (f, s);` lit une ligne complète dans `f` (toujours `readln` sur un string).
`read (f, x);` lit un entier dans `f`. On peut de la même manière lire un réel.

<u>Écriture</u>

`write (f, c)` ; écrit un caractère dans `f`.

`write (f, s);` écrit le string dans `f`. Pour passer à la ligne on fait `writeln(f);`

`write (f, x, ' ');` écrit un entier dans `f`. On peut de la même manière écrire un réel ou un booléen. Il vaut mieux rajouter un espace (ou un retour chariot) après chaque donnée pour que lors d'une relecture ultérieure, les données ne soit pas accolées en un bloc illisible.

`write (f, r:8:2);` écrit un réel formatté dans `f`.

<u>Morale</u>

En lecture comme en écriture, la manipulation des fichiers texte se passe très naturellement, de la même façon que la lecture au clavier ou l'écriture à l'écran.

Tous les algorithmes de lecture vus en TD (Horner, compter les `'LE'`) sont directement applicables sur les fichiers texte.

En fait, le clavier et l'écran sont tout simplement considérés comme des fichiers texte, les fichiers `input` et `output`.

2.3 Fichiers d'élements

Les fichiers d'éléments sont des copies de la mémoire vive, les éléments étant tous du même type.

Le type d'élément peut être un type simple, un enregistrement, un tableau, etc.

<u>Déclaration</u>

```
TYPE element_t = record
                    age    : integer;
                    majeur : boolean;
                  end;
VAR  f : file of element_t;
     e : element_t;
```

<u>Lecture</u>

`read (f, e);` lit un élément dans `f`. On ne fait jamais de `readln`.

<u>Écriture</u>

`write(f, e);` écrit un élément dans `f`. On ne fait jamais de `writeln`.

<u>Schémas types</u>

- Mettre un vecteur `vec` de `nb` éléments dans un fichier.

```
VAR
    vec   : array [1..vmax] of element_t;
    nb, i : integer;
BEGIN
    assign (f, nomf);
    rewrite (f);

    for i := 1 to nb do
      write (f, vec[i]);
```

```
        close (f);
    END;
```

- Opération inverse ; on ne connait pas `nb` au départ.

```
    BEGIN
        assign (f, nomf);
        reset (f);

        nb := 0;
        while not eof(f) and (nb < vmax) do
          begin nb := nb+1; read(f, vec[nb]); end;

        close (f);
    END;
```

2.4 Gestion des erreurs

L'ouverture d'un fichier peut provoquer une erreur, qui plante le programme. Par exemple, si on veut ouvrir un fichier en lecture, ce fichier doit exister. Si on veut créer un fichier, le chemin du fichier doit être valide.

Chaque compilateur fournit sa propre méthode pour éviter un plantage. Sous Delphi et sous Turbo Pascal, on encadre `reset` ou `rewrite` entre 2 options de compilations spéciales :

`{$I-}` désactive temporairement le contrôle des entrées/sorties
`{$I+}` le rétablit.

Juste après on regarde si il y a eu une erreur en testant la variable `IoResult`.

Exemple En écriture

```
    BEGIN
        assign (f, nomf);

        {$I-} rewrite (f); {$I+}
        ok := IoResult = 0;

        if not ok
        then writeln ('Erreur création fichier ', nomf)
        else begin
                ...
                write (f, ...);
                ...
                close (f);
             end;
    END;
```

Exemple En lecture

```
BEGIN
    assign (f, nomf);

    {$I-} reset (f); {$I+}
    ok := IoResult = 0;

    if not ok
    then writeln ('Erreur lecture fichier ', nomf)
    else begin
           ...
           while not eof(f) do
             begin
               read (f, ...);
               ...
             end;
           ...
           close (f);
         end;
END;
```

Remarque

On peut aussi utiliser `IoResult` dans la lecture au clavier, en encadrant `read` entre `{$I-}` et `{$I+}`.

Par exemple lorsqu'on attend un réel et qu'une lettre est tapée, le programme, au lieu de planter, détectera l'erreur et pourra redemander une frappe.

VII. Algorithmes avec des vecteurs

Rappel : on appelle *vecteur* un tableau en une dimension.

1 Recherche séquentielle d'un élément

Prenons par exemple un tableau d'entiers.

On déclare le type `vec_t` suivant :

```
CONST VMax  = 1000;
TYPE  vec_t = array [1..VMax] of integer;
```

Soit `v` un `vec_t` de `vn` éléments ($1 \leq$ `vn` $\leq$ `VMax`).

On veut écrire une fonction booléenne qui dit si un entier `x` se trouve dans le tableau `v`.

1.1 Dans un vecteur non trié

On parcourt le tableau et on s'arrête dès que `x` est trouvé ou que la fin du tableau est atteinte.

Première implémentation

```
FUNCTION cherche1 (v : vec_t; vn, x : integer) : boolean;
VAR i : integer;
BEGIN
    i := 1;
    while (i <= vn) and (v[i] <> x) do
      i := i+1;
    cherche1 := i <= vn;
END;
```

On sort du `while` dans deux situations :
– Si `i > vn`, on a parcouru tout le tableau sans trouver `x`.
– Sinon `i <= vn` et `v[i] = x` : on a trouvé `x`.

On ne peut donc pas écrire comme résultat `cherche1 := v[i] = x` puisque `i` peut sortir du tableau ; c'est pourquoi on écrit `cherche1 := i <= vn`.

Il y a un problème : dans le `while`, l'expression `(i <= vn) and (v[i] <> x)` est toujours complétement évaluée, même si le premier terme est faux.
En effet, le `and` ne signifie pas « et sinon », comme dans d'autres langages.

La conséquence est que si `x` n'est pas dans `v`, à la dernière itération on évaluera `(vn+1 <= vn) and (v[vn+1] <> x)` et **on sortira du vecteur** !!

⟶ Implémentation à éviter absolument.

Deuxième implémentation avec un booléen

On va décomposer le test dans le `while`.

```
FUNCTION cherche2 (v : vec_t; vn, x : integer) : boolean;
VAR i : integer;
    continuer : boolean;
BEGIN
    i := 1; continuer := true;
    while continuer do
      if i > vn
      then continuer := false
      else if v[i] = x
           then continuer := false
           else i := i+1;
    cherche2 := i <= vn;
END;
```

<u>Troisième implémentation</u>

```
FUNCTION cherche3 (v : vec_t; vn, x : integer) : boolean;
VAR i : integer;
BEGIN
    i := 1;
    while (i < vn) and (v[i] <> x) do  { i < n  strict }
      i := i+1;
    cherche3 := v[i] = x;
END;
```

On sort toujours du `while` avec `i <= vn`, donc on ne sort jamais du tableau.

Par contre pour le résultat, il faut changer le test, et on a le droit d'écrire `cherche3 := v[i] = x`.

<u>Le coût</u> Le vecteur `v` étant non trié, il faut

- ▷ `vn` itérations si `x` $\notin$ `v`.
- ▷ `vn/2` itérations en moyenne si `x` $\in$ `v`.

1.2 Dans un vecteur trié

Lorsqu'on parle de vecteur trié, on suppose toujours que les vecteurs sont triés par ordre *croissant* : $\forall i,\ v[i] \leq v[i+1]$.

On parcourt le tableau et on s'arrête dès que :
– `x` est trouvé
– ou la fin du tableau est atteinte
– ou `v[i] > x` : ça veut dire que tous les éléments qui suivent seront plus grands que `x`, inutile de continuer.

On peut adapter facilement la méthode 3 :

```
FUNCTION cherche4 (v : vec_t; vn, x : integer) : boolean;
VAR i : integer;
BEGIN
    i := 1;
```

```
        while (i < vn) and (v[i] < x) do  { v[i] < x  strict }
          i := i+1;
        cherche4 := v[i] = x;
    END;
```

Le coût Le vecteur v étant trié, il faut en moyenne vn/2 itérations, que x appartienne ou non à v.

2 La dichotomie

Prenons l'exemple du correcteur orthographique dans un traitement de texte, utilisé pour corriger une lettre.

Le dictionnaire du correcteur contient tous les mots de la langue, orthographiés de toutes les façons possibles, soit par exemple 1 million d'entrées.

Avec la recherche séquentielle sur un dictionnaire trié, il faudra donc en moyenne 500 000 itérations pour trouver un mot !

Mettons que le texte à corriger fasse 2000 mots. Il faudra donc $2000 \times 500\,000 = 1$ milliard d'itérations pour corriger le texte !

Sachant qu'en plus, la comparaison de deux mots est nettement plus lente que la comparaison de 2 entiers, la correction va durer plusieurs jours !! C'est tout à fait inacceptable.

Pour accélérer les choses, on va s'inspirer du jeu des 1000 francs.

2.1 Le jeu des 1000 francs

Jeu1 L'ordinateur choisit un prix secret entre 1 et 1000 F, et le joueur doit le deviner en un nombre minimum de coups.

```
    PROCEDURE jeu1 (secret : integer);
    VAR n, essai  : integer;
        continuer : boolean;
    BEGIN
        continuer := true; n := 1;
        while continuer do
        begin
          write ('Essai ', n, ' : '); readln (essai);
          if essai < secret then writeln ('+')
          else if essai > secret then writeln ('-')
          else begin
                 writeln ('Gagné en ', n, ' coups');
                 continuer := false;
               end;
          n := n+1;
        end;
    END;
```

Ce qui nous intéresse c'est la stratégie du joueur : admettons que secret $= 326$.

Essai	$\longrightarrow$ Réponse	Intervalle possible
500	$-$	1–500
250	$+$	250–500
375	$-$	250–375
312	$+$	312–375
343	$-$	312–343
328	$-$	312–328
321	$+$	321–328
324	$+$	324–328
326	Gagné	

La solution est trouvée en seulement 9 coups !

C'est le principe de la dichotomie : on a un intervalle de possibilités, et à chaque itération on réduit de moitié la taille de cet intervalle.

De la sorte, le nombre maximum d'itération est $\log_2$`vn`, c'est à dire 10 pour `vn` = 1000, de 20 pour `vn` = 1 million.

Jeu2 La réciproque : l'utilisateur choisit un prix secret entre 1 et 1000 F, et l'ordinateur doit le deviner en un nombre minimum de coups.

Le programme va donc gérer un intervalle de possibilités, c'est-à-dire un début et une fin, proposer le milieu de l'intervalle, puis changer l'intervalle en fonction de la réponse.

La recherche dichotomique consiste à faire la même chose sur les indices, et est donc très performante : sur l'exemple du correcteur orthographique, il faudra 20 itérations pour trouver un mot, donc 40 000 itérations pour corriger tout le texte, ce qui est quasiment instantanné.

2.2 Recherche dichotomique

La dichotomie se fait toujours sur un vecteur trié.

Cela consiste à considérer une certaine plage de recherche `inf..sup` sur le vecteur, plage que l'on réduit d'un facteur 2 à chaque itération.

Au départ, la plage de recherche est tout le vecteur.

À une itération donnée, on a une plage `[inf..sup]`
et son milieu est `m := (inf + sup) div 2`.
On a donc la subdivision : `[inf..m-1]`, `[m]`, `[m+1..sup]`.

- Soit `v[m] = x` et on a fini.
- Soit `v[m] < x`, donc `x` $\notin$ `[inf..m]` et la nouvelle plage sera `[m+1..sup]`.
- Soit `v[m] > x`, donc `x` $\notin$ `[m..sup]` et la nouvelle plage sera `[inf..m-1]`.

On implémente l'algorithme avec un booléen `trouve`.

```
FUNCTION cherche5 (v : vec_t; vn, x : integer) : boolean;
VAR inf, sup, m : integer;
    trouve      : boolean;
BEGIN
    trouve := false;
    inf := 1; sup := vn;

    while (inf <= sup) and not trouve do
    begin
      m := (inf + sup) div 2;
      if v[m] = x
      then trouve := true
      else if v[m] < x
           then inf := m+1
           else sup := m-1;
    end;

    cherche5 := trouve;
END;
```

Remarque Le fait de prendre `m-1` ou `m+1` n'est pas une simple optimisation, mais est essentiel pour que l'algorithme se termine.

Le coût Il faut

▷ $\log_2$vn itérations si $\mathtt{x} \notin \mathtt{v}$.

▷ $\log_2$vn itérations au plus si $\mathtt{x} \in \mathtt{v}$.

On peut améliorer l'efficacité dans le cas où $\mathtt{x} \notin \mathtt{v}$:

```
BEGIN
    trouve := false;

    if (v[1] <= x) and (x <= v[vn]) then
    begin
      inf := 1; sup := vn;

      while {...}
    end;

    cherche6 := trouve;
END;
```

3 Tri d'un vecteur

Soit `v[1..vn]` un vecteur non trié. Nous voulons construire un vecteur `w[1..vn]` qui contienne les même éléments que `v`, et qui soit trié.

Il existe de très nombreuses méthodes de tris, qui sont plus ou moins faciles à implémenter, et dont certaines sont nettement plus efficaces que d'autres.

Dans certaines méthodes on peut se passer du second vecteur `w`, en travaillant directement sur `v` où seront permutés des éléments.

3.1 Tri par remplacement

La méthode consiste à sélectionner des minimums successifs dans `v`, et à les ranger au fur et à mesure dans `w`.

Au départ on recherche quel est le max.

À chaque pas :

- ▷ on cherche `min(v)`
- ▷ on le met au bout de `w`
- ▷ on le remplace dans `v` par `max(v)`

Exemple Trier dans l'ordre alphabétique les lettres du mot `'ETABLES'`.
Le max est la lettre `'T'`.

i	v	indice du min dans v	w trié	v après remplacement
1	E T A B L E S	3	A	E T T B L E S
2	E T T B L E S	4	A B	E T T T L E S
3	E T T T L E S	1	A B E	T T T T L E S
4	T T T T L E S	6	A B E E	T T T T L T S
5	T T T T L T S	5	A B E E L	T T T T T T S
6	T T T T T T S	7	A B E E L S	T T T T T T T
7	T T T T T T T	fini	A B E E L S T	

```
FUNCTION maximum (v : vec_t; vn : integer) : char;
VAR i : integer; m : char;
BEGIN
    m := v[1];
    for i := 2 to vn do
      if v[i] > m then m := v[i]:
    maximum := m;
END;

FUNCTION ind_min (v : vec_t; vn : integer) : integer;
VAR i, j : integer;
BEGIN
    j := 1;
    for i := 2 to vn do
      if v[i] < v[j] then j := i:
    ind_min := j;
END;

PROCEDURE tri_remplacement (     v  : vec_t;
                                 vn : integer;
                             var w  : vec_t );
VAR max  : char;
    i, j : integer;
BEGIN
    { recherche du max }
    max := maximum (v,vn);
```

```
        { pas a pas }
        for i := 1 to vn-1 do
        begin
          j := ind_min (v,vn);
          w[i] := v[j];
          v[j] := max;
        end;

        { on met le max dans la derniere case }
        w[vn] := max;
    END;
```

Le coût

Les performances sont faibles : il y a environ $\texttt{vn}^2$ comparaisons, et $2,5\,\texttt{vn}$ affectations en moyenne.

Par exemple si `vn` = 1000, on aura 1 000 000 de comparaisons et 2500 affectations.

3.2 Tri par permutation

Dans la méthode précédente, la place occupée est double ; on peut éviter de créer un nouveau vecteur en travaillant directement sur `v`.

Principe On est à l'étape `i`.

▷ Supposons déjà trié `v[1..i-1]`, et non traité `v[i..vn]`.

▷ On cherche le min dans `v[i..vn]`

▷ On le permute avec `v[i]`.

▷ Maintenant `v[1..i]` est trié.

Exemple Trier les lettres du mot `'ETABLES'`.

i	v trié / non traité	indice du min	lettres à permuter	v après permutation
1	/ETABLES	3	E et A	A/TEBLES
2	A/TEBLES	4	T et B	AB/ETLES
3	AB/ETLES	3	non	ABE/TLES
4	ABE/TLES	6	T et E	ABEE/LTS
5	ABEE/LTS	5	non	ABEEL/TS
6	ABEEL/TS	7	T et S	ABEELS/T
7	ABEELS/T		fini	ABEELST/

Le coût

Les performances sont meilleures que le tri par remplacement : il y a environ $\texttt{vn}^2\,/\,2$ comparaisons, et $\texttt{vn}\,/\,2$ permutations en moyenne.

Par exemple si `vn` = 1000, on aura 500 000 comparaisons et 1500 affectations.

3.3 Tri à bulles

C'est une variante du tri par permutation, un peu moins efficace ; il y a une version optimisée qui est un peu meilleure que le tri par permutation.

Principe On est à l'étape `i`.

- ▷ Supposons déjà trié `v[1..i-1]`, et non traité `v[i..vn]`.
- ▷ On parcourt `v[i..vn]` en descendant et, chaque fois que deux éléments consécutifs ne sont pas dans l'ordre, on les permute.
- ▷ En fin de parcours le min de `v[i..vn]` se retrouve dans `v[i]`.
- ▷ Maintenant `v[1..i]` est trié.

Le nom du « tri à bulles » vient de ce que à chaque étape `i`, les éléments les plus « légers » remontent vers la surface, sont transportés à gauche.

On constate aussi que à chaque étape, l'ordre général est accru.

Tri à bulles optimisé

Si lors d'une étape `i`, aucune permutation n'a lieu, c'est que `[i..vn]` est déjà dans l'ordre, et le tri est fini.
⟶ booléen `apermute` ou encore mieux, indice `dp` de dernière permutation.

3.4 Tri par comptage

Cette méthode consiste à construire un vecteur d'indices `ind`, où l'on calcule la position que devrait avoir chaque élément pour que le vecteur soit trié.

Exemple Résultat sur le tri des lettres du mot `'ETABLES'`.

i	1	2	3	4	5	6	7
v	E	T	A	B	L	E	S
ind	3	7	1	2	5	4	6
w	A	B	E	E	L	S	T

```
PROCEDURE tri_comptage (    v  : vec_t;
                            vn : integer;
                        var w  : vec_t);
VAR i, k : integer;
    ind  : array [1..VMax] of integer;
BEGIN
    { init }
    for i := 1 to vn do ind[i] := 1;

    { construit ind }
    for i := 1 to vn-1 do
      for k := i+1 to vn do
        if v[k] < v[i]
        then ind[i] := ind[i]+1
        else ind[k] := ind[k]+1;
```

```
        { construit w }
        for i := 1 to vn do w[ind[i]] := v[i];
    END;
```

Le coût

La place occupée est importante (3 vecteurs).

Le nombre de comparaisons est constant ($\texttt{vn} \times (\texttt{vn} - 1)/2$). C'est du même ordre que le tri par permutation ou à bulles non optimisé.

Il y a très peu d'affectations (vn) ; cela est très intéressant si les éléments de vn sont « lourds », par exemple des strings ou des records triés sur un champ.

4 Mise à jour d'un vecteur

Soit v[1..vn] un vecteur, avec $1 \leq \texttt{vn} \leq \texttt{VMax}$.

On regarde comment insérer ou supprimer un élément de v, en conservant l'ordre ou non des éléments.

4.1 Insertion dans un vecteur non trié

Pour insérer un élément x en queue de v, il suffit de faire

```
if vn < VMax then
begin vn := vn+1; v[vn] := x; end;
```

Si on veut insérer x à une autre position, c'est que l'on considère que le vecteur est trié avec un certain ordre (croissant, décroissant, d'apparition, etc).

4.2 Insertion dans un vecteur trié

On veut insérer x à la position i dans v, avec $1 \leq \texttt{i} \leq \texttt{vn}$.

On doit décaler v[i..vn] vers la droite :

```
if vn < VMax then
begin
  vn := vn+1;
  for j := vn downto i+1 do   { en descendant }
    v[j] := v[j-1];
  v[i] := x;
end;
```

4.3 Suppression dans un vecteur non trié

On veut supprimer l'élément v[i], avec $1 \leq \texttt{i} \leq \texttt{vn}$.

Si l'ordre des éléments dans v est indifférent, on place l'élément de queue dans le trou.

```
v[i] := v[vn];  { si i = vn, ca ne fait rien }
vn := vn-1;
```

4.4 Suppression dans un vecteur trié

On décale `v[i+1..vn]` vers la gauche :

```
for j := i to vn-1 do   { en montant }
  v[j] := v[j+1];
vn := vn-1;
```

5 Tri par insertion

Voici un algorithme de tri basé sur l'insertion dans un vecteur trié.

Principe On est à l'étape `i`.

- ▷ Supposons déjà trié `v[1..i-1]`, et non traité `v[i..vn]`.
- ▷ On pose `x = v[i]`. La case `i` est disponible.
- ▷ On cherche `k` tel que `v[k-1]` $\leq$ `x` $<$ `v[k]`.
- ▷ On insère `x` à la position `k` dans `v[1..i-1]`, ce qui oblige à décaler d'abord `v[k..i-1]` vers `v[k+1..i]` ; le trou en `i` est bouché.
- ▷ Maintenant `v[1..i]` est trié et `v[i+1..vn]` est non traité.

Exemple Trier les lettres du mot `'ETABLES'`.

Au départ, on considère que `v[1..1]` est trié ; on va donc insérer les éléments suivants, de `2` à `vn`.

i	v trié / non traité	lettre à insérer	v après insertion
2	E/TABLES	T	ET/ABLES
3	ET/ABLES	A	AET/BLES
4	AET/BLES	B	ABET/LES
5	ABET/LES	L	ABELT/ES
6	ABELT/ES	E	ABEELT/S
7	ABEELT/S	S	ABEELST/

Implémentation du tri

```
PROCEDURE tri_insertion ( var v : vec_t; vn : integer);
VAR i : integer;
BEGIN
    for i := 2 to vn do
      insérer_trié (v, i);
END;
```

Implémentation de l'insertion

```
PROCEDURE insérer_trié ( var v : vec_t; i : integer);
VAR j, k  : integer;
    x : type_element;
BEGIN
    { élément à insérer }
    x := v[i];

    { recherche position d'insertion de x }
    k := posit_ins (v, i, x);

    { décalage : en descendant }
    for j := i downto k+1 do v[j] := v[j-1];

    { insertion }
    v[k] := x;
END;
```

Optimisation de la recherche du point d'insertion

La recherche du point d'insertion k peut se faire séquentiellement ; mais on a tout intérêt à employer une recherche dichotomique, bien plus efficace.

```
FUNCTION posit_ins ( var v : vec_t;
                         i : integer;
                         x : type_element) : integer;
VAR inf, sup, m  : integer;
BEGIN
    { le cas des extrémités }
    if x < v[1] then posit_ins := 1
    else if v[i-1] <= x then posit_ins := i
    else begin
           { init dichotomie : les cas 1 et i sont déjà traités }
           inf := 2; sup := i-1;

           { recherche position m tel que v[m-1] <= x < v[m] }
           { : variante de la dichotomie habituelle,         }
           {   sans le booléen trouve                        }
           while inf < sup do
           begin
             m := (inf + sup) div 2;
             if v[m] <= x
             then inf := m+1
             else sup := m;
           end;

           posit_ins := sup;
         end;
END;
```

Le coût

Méthode nettement plus efficace que les autres pour vn grand :

La recherche dichotomique sur v[1..i] est en $\log_2 \mathtt{i}$. L'insertion dans v[1..i] coute en moyenne $\mathtt{i}/2$. Le coût total est donc $\sum_2^{\mathtt{vn}}(\log_2 \mathtt{i} + \mathtt{i}/2)$.

Par exemple avec $\mathtt{vn} = 1000$, le coût est de 10 000, contre 500 000 pour les autres tris.