Programmation en Pascal – Notes de cours
Licence de linguistique – Université Paris 7

Yan Jurski et Ines Klimann

5 novembre 2004

Ces notes représentent un premier jet. N'hésitez pas à nous faire part d'éventuelles erreurs pour que nous les corrigions ou de doutes qu'elles pourraient susciter, pour que nous les rendions plus précises, en cours ou par courrier électronique à `{jurski,klimann}@liafa.jussieu.fr`.

Table des matières

1 Utilité d'un programme **3**

2 Mes premiers pas avec `Pascal` **4**
- 2.1 Comment utiliser `Pascal` . . . 4
- 2.2 Mon premier programme en `Pascal` . . . 4
- 2.3 Les commentaires . . . 4

3 Constantes et variables, ou comment ranger les valeurs qui apparaissent dans un programme **5**
- 3.1 Les constantes . . . 5
- 3.2 Les variables . . . 7
- 3.3 Les types . . . 8

4 Procédures et fonctions **8**
- 4.1 Les procédures . . . 8
- 4.2 Les fonctions . . . 10
- 4.3 Les passages par référence (ou par variable) . . . 11

5 Les structures de contrôle **12**
- 5.1 Tests . . . 12
  - 5.1.1 `if ... then ... [else ...]` . . . 12
  - 5.1.2 `case ... of ... else ...` . . . 14
- 5.2 Boucles . . . 15
  - 5.2.1 `for ... to/downto... do...` . . . 15
  - 5.2.2 `repeat ... until ...` . . . 17
  - 5.2.3 `while ... do ...` . . . 17
  - 5.2.4 Les sorties et sauts de boucle . . . 17

6 Un peu de récursivité **19**

7 Les types structurés **20**
7.1 Déclaration d'un nouveau type 20
7.2 Tableaux 21
7.3 Enregistrements (`record`) 23

1 Utilité d'un programme

Imaginons qu'on ait un fichier informatique contenant *La disparition* de G. Perec et qu'on veuille vérifier qu'effectivement il ne contient pas la lettre `e`. Une première solution consiste à le lire d'un bout à l'autre pour faire cette vérification. Outre que cette méthode prend du temps, elle est assez peu fiable : emporté par l'anecdote, on peut très bien laisser passer un `e` sans s'en apercevoir. Une autre méthode consiste à profiter du fait qu'on a le texte sous forme informatique et à faire cette vérification grâce à l'outil informatique, en appliquant une méthode systématique qui nous donne la réponse à la question "*La disparition* contient-elle un `e` ?". Par exemple :

Exemple 1

"*La disparition* contient-elle un `e` ?" – version 1

Donnée: un fichier informatique contenant le texte de *La disparition*

```
début
    contient_un_e := non;
    pour le caractère courant du texte faire
        si ce caractère est un e alors
            contient_un_e := oui
        finsi
        (avancer le curseur de lecture d'un caractère)
    finpour
    retourner contient_un_e;
fin
```

On remarque que cette méthode n'est pas très efficace si G. Perec a par mégarde laissé échappé un `e`. En effet, dès qu'on voit ce `e`, on peut alors arrêter l'exploration du texte et conclure. On peut donc imaginer une deuxième méthode (que nous appellerons un deuxième *algorithme*), donnée dans l'exemple 2.

Exemple 2

"*La disparition* contient-elle un `e` ?" – version 2

Donnée: un fichier informatique contenant le texte de *La disparition*

```
début
    tant que le caractère courant n'est pas un e faire
        (avancer le curseur de lecture d'un caractère)
    fintq
    si on est à la fin du texte alors
        retourner non
    sinon
        retourner oui
    finsi
fin
```

Le but de la première partie de ce cours est de pouvoir écrire des programmes en langage `Pascal` qui permettent de faire ce genre de vérification (et d'autres). Pour des raisons ayant trait à votre formation, le cours sera centré sur le traitement de fichiers textes, mais il est bien évident que l'on peut programmer bien d'autres choses que du traitement de données texte.

2 Mes premiers pas avec Pascal

2.1 Comment utiliser Pascal

Pascal est un langage qu'il faut compiler : vous écriver un fichier texte lisible par un être humain et un programme appelé *compilateur* transforme ce fichier en fichier binaire, illisible pour un être humain, mais tout à fait compréhensible pour l'ordinateur sur lequel on vient d'exécuter la compilation.

Plusieurs compilateurs sont disponibles, notament en fonction du système d'exploitation que vous utilisez (DOS, Windows, Linux, etc.). Nous utiliserons un compilateur `freepascal`, disponible gratuitement sur internet à l'adresse

`http://www.freepascal.org/`

et dont il existe une version pour la plupart des systèmes que vous pouvez être amenés à utiliser.

Sur un ordinateur sous Linux, une fois installé, le compilateur peut être invoqué grâce à la commande `fpc` suivie du nom du *code source* (le fichier texte dans lequel vous avez écrit le programme). Une fois la commande lancée, le compilateur donne des informations concernant la compilation sur la sortie standard. **Il est important que vous lisiez les informations fournies par le compilateur**, en effet, c'est grâce à celles-ci que vous saurez si la compilation s'est déroulée correctement ou que vous aurez des informations concernant les éventuelles erreurs dans votre programme.

2.2 Mon premier programme en Pascal

A l'aide d'un éditeur de texte, par exemple `emacs`, créez un fichier que vous nommerez `premier.p` et qui contiendra le texte suivant :

programme 1 : `premier.p`

```
program premier;

begin
   write('bonjour');
end.
```

indexmotsclefsbeginindexmotsclefswrite

Dans un terminal, compilez ce programme en utilisant la commande

`fpc premier.p`

Si vous regardez le contenu de votre répertoire (commande `ls`), vous vous apercevrez que plusieurs nouveaux fichiers ont été créés. Celui qui nous intéresse se nomme `premier`, du nom de votre fichier de départ sans l'extension. Exécutez votre programme en lançant la commande

`./premier`

et regardez ce qui se passe à l'écran.

2.3 Les commentaires

Quand on écrit un programme, il ne faut pas oublier que celui-ci sera sûrement utilisé par d'autres personnes et en tout cas par soi-même; soit pour ce qu'il fait directement, soit pour le réadapter à un problème proche. C'est pourquoi il est très important de *commenter* ses programmes. Un minimum étant une description globale de ce que fait ce programme. On verra qu'un programme devient rapidement long et qu'alors il faut entrer plus dans les détails. On peut ainsi réécrire le programme 1 en :

programme 2 : `premier-commente.p`

```
{
programme premier-commente.p
ecrit par I. Klimann et Y. Jurski
exemple de programme commente qui affiche bonjour a l'ecran
}

program premier-commente;

begin
   write('bonjour');
end.
```

Il faut absolument prendre l'habitude de commenté son code *au fur et à mesure* de la programmation, car les commentaires peuvent aider à structurer un programme. Il est par exemple tout a fait possible d'écrire quasi entièrement un commentaire avant de s'attaquer au code lui-même; en revanche il faut éviter d'écrire tout le code en se disant qu'on commentera après.

3 Constantes et variables, ou comment ranger les valeurs qui apparaissent dans un programme

3.1 Les constantes

Que se passe-t-il si on veut calculer le périmètre d'un cercle de rayon 4.5 ou estimer le nombre de mots d'un texte de 237 pages, sachant qu'on a en moyenne 13 mots par ligne et 37 lignes par pages? On peut bien entendu écrire les programmes suivants :

programme 3 : `perimetre_1.p`

```
{
calcul du perimetre d'un cercle de rayon 4.5
- version 1
}

program perimetre_1;

begin
   write('le perimetre d''un cercle de rayon 4.5 est : ');
   writeln(2*3.1415*4.5);
end.
```

programme 4 : `nb_de_mots_1.p`

```
{
estimation du nombre de mots d'un texte de 237 pages,
avec en moyenne 13 mots par ligne et 37 lignes par page
- version 1
}
```

```
program nb_de_mots_1;

begin
   writeln('On estime le nombre de mots a ', 237*13*37, '.');
end.
```

Dans ces deux programmes, ce qui peut être amené à changer, c'est respectivement le rayon du cercle et le nombre de pages du texte. On peut donc considérer les autres données (la valeur de π dans le premier cas, le nombre de mots par ligne et de lignes par page dans le second) comme des *constantes*. On leur attribuera alors un nom, qui servira de la même façon que les valeurs numériques, mais aura plusieurs avantages :

- on n'a pas besoin de retenir les valeurs,
- si on veut modifier ces valeurs (par exemple ajouter des décimales à *pi* ou changer de police de caractères, ce qui modifie a priori la taille des caractères et donc le nombre de mots par ligne et éventuellement le nombre de lignes par page), il suffit de changer ces valeurs une unique fois, au moment où l'on fait l'association entre le nom de la constante et sa valeur; cela évite d'oublier des changements ou de faire des changements qui n'ont pas lieu d'être (on peut très bien imaginer que dans un programme on utilise le nombre de 13 pour autre chose et alors il ne faudra pas modifier cette occurence-là).

Les programmes précédents deviennent alors :

programme 5 : `perimetre_2.p`

```
{
calcul du perimetre d'un cercle de rayon 4.5
- version 2
}

program perimetre_2;

const
   pi = 3.1415;

begin
   write('le perimetre d''un cercle de rayon 4.5 est : ');
   writeln(2*pi*4.5);
end.
```

programme 6 : `nb_de_mots_2.p`

```
{
estimation du nombre de mots d'un texte de 237 pages,
avec en moyenne nb_mots_par_ligne mots par ligne et
nb_lignes_par_page lignes par page
- version 2
}

program nb_de_mots_2;
```

```
const
   nb_mots_par_ligne  = 13;
   nb_lignes_par_page = 37;

begin
   writeln('On estime le nombre de mots a ',
           237*nb_mots_par_ligne*nb_lignes_par_page, '.');
end.
```

Attention : `Pascal` ne distingue pas les majuscules des minuscules.

3.2 Les variables

Dans les programmes précédents, on reste insatisfait par le fait que les quantités 4.5 (rayon du cercle) et 237 (nombre de pages du livre) apparaissent "en dur". On pourrait, comme pour les autres quantités, les faire apparaître comme des constantes, mais ce ne sont pas vraiment des constantes : si on veut calculer le périmètres de tous les cercles dont le rayon est compris entre 4 et 5, espacé de 0.1, ou si, une fois la police fixée, on veut estimer le nombre de mots de plusieurs ouvrages, on n'a pas forcément envie de créer une constante par rayon de cercle ou par ouvrage. En fait, dans ces cas-là, on a plutôt besoin de "boîtes" dans lesquelles mettre des valeurs qui seront amenées à être modifiées : ce sont les *variables*. Le nom d'une variable est composé d'une suite de caractères comprenant les lettres, les chiffres et le caractère *underscore* (_). En aucun cas on ne peut mettre un espace dans le nom d'une variable (ni dans un quelconque identificateur), ni un signe *moins* (`-`).

Le `Pascal` est un langage *typé* : quand on utilise une variable, il faut préciser quel est son type (par exemple si c'est un entier, un réel, un caractère, une chaîne de caractères, etc.). Cette donnée permet au langage d'avoir des traitements spécifiques suivant le type des données, par exemple :

- l'affichage d'un entier se fait sans chiffre après la virgule, alors que l'affichage d'un réel se fait avec un certain nombre de chiffres après la virgule, par défaut,
- le symbole `+` entre deux entiers effectue l'opération somme, le même symbole entre deux caractères effectue l'opération concaténation.

On pourra par exemple regarder ce que donne le programme 7.

programme 7 : `typage.p`

```
{
affichage et "somme" de divers types de variables
}

program typage;

var
   i,j : integer;
   r,s : real;
   c,d : char;

begin
   i := 3; j := 5;
   r := 3; s := 5;
   c := '3'; d := '5';
   writeln('i=',i,'     i+j=',i+j);
```

```
   writeln('r=',r,'     r+s=',r+s);
   writeln('c=',c,'     c+d=',c+d);
end.
```

3.3 Les types

Il existe plusieurs types en `Pascal`, on a déjà vu les types entier (`integer`), réel (`real`) et caractère (`char`). Voici un complément sur ces types et sur d'autres types disponibles en `Pascal`.

- `INTEGER` les entiers
  opérateurs disponibles : + - * div mod
- `REAL` les réels
  opérateurs disponibles : + - * /
- `BOOLEAN` les booléens (deux valeurs : `TRUE` et `FALSE`)
  opérateurs disponibles : AND OR NOT XOR
- `CHAR` les caractères (suivant l'ordre ascii)
- `STRING` les chaînes de caractères

Il ne faut pas confondre un type mathématique avec sa représentation en `Pascal`. Par exemple, d'un point de vue mathématique, un réel peut avoir autant de chiffres après la virgule qu'on veut, ce n'est pas le cas du type `REAL` en `Pascal`, puisque la place mémoire correspondant à une variable de ce type est limitée. De même, une variable de type `INTEGER` ne peut prendre n'importe quelle valeur car elle est codée sur 16 bits : quand on a atteint la plus grande valeur de ce type ($2^{15} - 1 = 32767$), on retombe sur la plus petite ($-2^{15} = -32768$) en ajoutant 1.

Il existe bien entendu d'autres types en `Pascal` et les types déjà cités ont d'autres caractéristiques, nous les verrons au fur et à mesure des besoins.

4 Procédures et fonctions

Supposons qu'on veuille évaluer le nombre de mots de plusieurs ouvrages et/ou afficher ces évaluations. Une première solution est de faire du copié-collé du bout de programme qu'on a écrit. Cette solution est à proscrire : si on veut modifier un détail dans le programme, on serait amené à le modifier à plusieurs endroits, avec tous les risques d'erreur que cela comporte; de plus, si on ne connaît pas à l'avance le nombre d'ouvrages, cette méthode est tout bonnement impossible à mettre en œuvre. Le mieux est de pouvoir enfermer dans une "boîte" un morceau de code et d'appeler cette boîte à chaque utilisation.

Ce type de boîte s'appelle une *fonction* ou une *procédure*, suivant qu'elle renvoie ou non une valeur.

4.1 Les procédures

On reprend l'exemple du paragraphe précédent. On considère le programme 8.

programme 8 : `premiere_procedure.p`

```
{
programme premiere_procedure.p
premier exemple d'utilisation de procedure
}

program premiere_procedure;
```

```
procedure affiche_nb_de_mots(nb_pages : integer);
const
   nb_mots_par_ligne  = 13;
   nb_lignes_par_page = 37;
begin
   writeln('nombre de mots estime : ',
           nb_mots_par_ligne * nb_lignes_par_page * nb_pages, '.');
end; { affiche_nb_de_mots }

begin
   {premier ouvrage : titre : 'Ouvrage 1', nombre de pages : 132}
   write('Ouvrage 1, ');
   affiche_nb_de_mots(132);
   {deusieme ouvrage : titre : 'Ouvrage 2', nombre de pages : 175}
   write('Ouvrage 2, ');
   affiche_nb_de_mots(175);
end.
```

Il serait plus satisfaisant de pouvoir fournir à la procédure `affiche_nb_de_mots` à la fois le nombre de pages de l'ouvrage et son titre, ce qui simplifierait d'autant plus l'écriture de la partie principale du programme. Pour celà, on utilise le type chaîne de caractères, `string` en `Pascal`.

programme 9 : `plusieurs_ouvrages.p`

```
{
programme plusieurs_ouvrages.p
introduction des chaines de caracteres
}

program plusieurs_ouvrages;

procedure affiche_nb_de_mots(titre : string; nb_pages : integer);
const
   nb_mots_par_ligne  = 13;
   nb_lignes_par_page = 37;
begin
   writeln(titre, ', nombre de mots estime : ',
           nb_mots_par_ligne * nb_lignes_par_page * nb_pages, '.');
end; { affiche_nb_de_mots }

begin
   {premier ouvrage : titre : 'Ouvrage 1', nombre de pages : 132}
   affiche_nb_de_mots('Ouvrage 1', 132);
   {deusieme ouvrage : titre : 'Ouvrage 2', nombre de pages : 175}
   affiche_nb_de_mots('Ouvrage 2', 175);
end.
```

Exercice 4.1 *Ecrire une procédure qui prend en argument une chaîne de caractères correspondant au nom d'un auteur, son année de naissance, son année de décès et le nombre de livres écrits par*

cet auteur, et affiche une phrase du type "Tartampion est mort à tel âge, après avoir écrit tant d'ouvrages.".

On remarque sur le programme de l'exercice 4.1 que si plusieurs arguments ont le même type, on peut se contenter de mettre à la suite les noms des arguments séparés par des virgules et ne préciser le type qu'une unique fois, comme on le faisait en déclarant des variables de même type.

4.2 Les fonctions

Si maintenant on ne veut plus afficher un texte à l'écran, mais récupérer le résultat d'un calcul, on utilise une *fonction*.

Le programme 10 montre comment on peut se servir de fonctions.

---

programme 10 : `polices.p`

```
{
programme polices.p
compare le nombre de mots autorises suivant la taille de la police
}

program polices;

function nb_de_mots_10_points(nb_pages : integer) : integer;
const
   nb_mots_par_ligne  = 18;
   nb_lignes_par_page = 38;
begin
   nb_de_mots_10_points := nb_mots_par_ligne * nb_lignes_par_page * nb_pages;
end; { nb_de_mots_10_points }

function nb_de_mots_12_points(nb_pages : integer) : integer;
const
   nb_mots_par_ligne  = 13;
   nb_lignes_par_page = 37;
begin
   nb_de_mots_12_points := nb_mots_par_ligne * nb_lignes_par_page * nb_pages;
end; { nb_de_mots_12_points }


procedure comparaison(nb_de_pages : integer);
var
   m10, m12 : integer;
begin
   m10 := nb_de_mots_10_points(nb_de_pages);
   m12 := nb_de_mots_12_points(nb_de_pages);
   writeln('Si vous avez ', nb_pages, ' pages, vous avez droit a :');
   writeln('* ', m10, ' mots en 10 points,');
   writeln('* ', m12, ' mots en 12 points.');
   writeln('Vous gagnez donc ', m10-m12, ' mots en 10 points.');
end;
```

```
begin
   comparaison(21);
end.
```

Exercice 4.2 *Ecrire une fonction* `longevite` *qui prend en argument les années de naissance et de décès de quelqu'un et renvoie l'âge de la personne au moment de sa mort.*

4.3 Les passages par référence (ou par variable)

Jusqu'à présent, tous les arguments des procédures et des fonctions que nous avons vus étaient passée *par valeur*: c'est en fait une copie de la valeur qui était passée, le code de la procédure ou de la fonction appelée ne pouvant altérer la valeur de la variable transmise.

Testez le programme 11 pour voir ce qui se passe.

programme 11 : `valeur.p`

```
{
 exemple de procedure inutile
}

program valeur;

procedure inutile(i : integer);
begin
   i := 5 ;
end; { inutile }


var
   v : integer;

begin
   v := 3;
   writeln('au debut : v=', v);
   inutile(v);
   writeln('apres la procedure inutile : v=', v);
end.
```

En `Pascal`, on a la possibilité de faire un passage de paramètre par *référence* (ou par *variable*): les opérations effectuées sur le paramètre dans une procédure ou une fonction altèrent effectivement celui-ci. Pour cela, dans l'en-tête de la fonction, avant le nom de l'argument, on rajoute le mot-clef du langage `var`. Comparer le programme 11 au programme 12.

programme 12 : `reference.p`

```
{
 exemple de passage par reference
}

program reference;
```

```
procedure passage_par_reference(var i : integer);
begin
   i := 5 ;
end; { passage_par_reference }


var
   v : integer;

begin
   v := 3;
   writeln('au debut : v=', v);
   passage_par_reference(v);
   writeln('apres la procedure passage_par_reference : v=', v);
end.
```

Exercice 4.3 *Ecrire une procédure* `echange` *qui prend en argument deux paramètres entiers et échange leurs valeurs.*

5 Les structures de contrôle

Jusqu'à présent, tous les programmes que nous avons vu avaient un déroulement linéaire. Le but des structures de contrôle est d'introduire un peu de variété. Dans la section 5.1 dédiées aux tests, nous verrons comment introduire des bifurcations dans ce déroulement. Puis dans la section 5.2 dédiées aux boucles, nous verrons comment introduire des retours en arrière dans ce déroulement.

5.1 Tests

5.1.1 `if ... then ... [else ...]`

Maintenant que l'on sait faire des calculs, on aimerait bien pouvoir faire des tests, par exemple : si le lecteur a une bonne vue, on affiche en 10 points, sinon on affiche en 12 points.

programme 13 : `test1.p`

```
{
programme test1.p
premiere utilisation du test if
}

program test1;

function taille_police(vue : integer) : integer;
begin
   if vue>=8
   then
      taille_police := 10  { pas de point-virgule avant le else !!! }
   else
      taille_police := 12;
end; { taille_police }
```

```
var
   acuite_visuelle : integer;
begin
   writeln('Quelle est votre acuite visuelle ? ');
   read(acuite_visuelle);
   writeln('Pour vous, les textes seront affiches avec une taille de ',
           taille_police(acuite_visuelle), ' points.');
end.
```

La syntaxe d'un test `if` est la suivante :

`if` *condition* `then` *action1*[;] [`else` *action2*;]

Si la *condition* est vérifiée, alors l'*action1* est exécutée, sinon c'est l'*action2* qui est exécutée, puis le programme passe à la suite. La partie `else` est facultative; si elle est omise et que la *condition* n'est pas vérifiée, alors le programme passe à la suite directement.

Attention : s'il y a un `else`, alors la dernière instruction avant ce `else` ne se termine pas par un point-virgule.

Une *action* est une composée soit d'une unique instruction, soit d'une suite d'instructions encadrée par les mots-clefs `begin` et `end`.

Quelle est la nature de *condition*? En fait *condition* est ce qu'on appelle un *expression booléenne*, c'est-à-dire une expression qui s'évalue soit en vrai (`true`), soit en faux (`false`). On peut en voir quelques exemples ci-dessous (attention, ce sont des morceaux de programme) :

```
if true then write('a'); {action toujours effectuee}
...
{n est un entier auquel on a affecte une valeur}
if n>=10 then writeln('au moins 10') else writeln('moins de 10');
...
{est_adherent est une fonction qui prend en argument une chaine de
caracteres et renvoie un booleen,
numero_adherent est une fonction qui prend en argument une chaine de
caracteres et renvoie un entier}
if est_adherent('Alain')
then writeln('numero d'adherent : ', numero_adherent('Alain'))
else writeln('non adherent');
```

Les opérateurs de comparaisons qui peuvent apparaîtrent sont les suivants : `=` `<>` (différent de) `<` `<=` `>` `>=`. Il s'utilisent aussi bien avec des types numériques qu'avec des caractères ou des chaînes de caractères (dans ces 2 derniers cas, on considère l'ordre `ascii`).

On peut également mettre comme condition une expression booléenne plus compliquée, en effectuant une opération entre des expressions simples. Les opérations sont celles vues à la section 3.3 pour les variables booléennes : `and or not xor`.

```
if (existe_ouvrage('Cours de Pascal')) and not (disponible('Cours de Pascal'))
then write('Voulez-vous commander ce livre ?');
```

Exercice 5.1 *Ecrire une fonction* `est_pair` *qui prend en argument un entier et renvoit* `true` *si cet entier est pair,* `false` *sinon.*

Exercice 5.2 *Ecrire une fonction* `est_voyelle` *qui prend en argument un caractère (qu'on suppose être une lettre de l'alphabet, sans faire de vérification à ce sujet) et renvoie* `true` *si cette lettre est une voyelle,* `false` *si c'est une consonne.*

Remarque : on peut tester si un caractère représente une lettre minuscule en vérifiant qu'il appartient à l'intervalle délimité par `'a'` et `'z'` de la façon suivante :

```
c : char;
if c in ['a'..'z'] then ...
```

Exercice 5.3 *On suppose que dans un magasin de photocopies, trois tarifs sont affichés : 0,20 € / photocopie jusqu'à 10 photocopies, 0,15 € / photocopie de 11 à 100 photocopies et 0,10 € / photocopie à partir de 100 photocopies.*

Ecrire une fonction qui prend en argument le nombre de photocopies effectuées et renvoie le prix à payer.

5.1.2 `case ... of ... else ...`

Le test `if` est bien adapté quand un ensemble de valeurs pour une expression donnée provoque un certain comportement du programme et que l'ensemble complémentaire provoque un autre comportement. Si on veut spécifier plusieurs comportement en fonction d'une partition des valeurs possibles en plus de deux sous-ensembles, il faut imbriquer les tests `if`, ce qui est tout à fait possible, mais fastidieux. Une méthode plus simple est mise à notre disposition : le test `case`.

La syntaxe de `case` est la suivante :

```
case expression of
value1.1, value1.2, ..., value1.n1 : action1;
value2.1, value2.2, ..., value2.n2 : action2;
...
valuep.1, valuep.2, ..., valuep.np : actionp;
else
action_par_défaut;
end;
```

Remarque : le compilateur `fpc` n'admet pas que des cas se chevauchent (ce qui est autorisé par le Turbo Pascal).

Tout comme pour le test `if`, une *action* est composée d'une instruction unique ou bien d'une suite d'instructions entourée des mots-clefs `begin` et `end`.

Vous avez un exemple d'utilisation de `case` dans le programme 14.

---

programme 14 : `premier_case.p`

```
{
ma premiere utilisation de case
}

program premier_case;

procedure action();
begin
   writeln('action effectuee');
```

```
end;


var
   rep : char;
begin
   write('Voulez-vous effectuez une action (o/n) ? ');
   read(rep);
   case rep of
     'o', 'O' : action();
     'n', 'N' : writeln('je ne fais rien');
   else
      writeln('mauvaise reponse...');
   end; { case }
end.
```

Exercice 5.4 *Réécrire une version plus courte de la fonction* `est_voyelle` *de l'exercice 5.2.*

5.2 Boucles

Une des caractéristiques de la programmation est de pouvoir exécuter des tâches de façon répétitive. On a pour cela deux familles de méthodes : la récursion qui fait l'objet de la section 6 et l'itération qui fait l'objet de la présente section.

Il s'agit ici de faire des boucles dans lesquelles on exécute une suite d'instructions. Il faut donc savoir quand on arrête de boucler.

5.2.1 `for ... to/downto... do...`

On utilise la boucle `for` quand on connaît le nombre d'itérations à effectuer. On introduit alors un compteur qui nous permet de connaître le numéro de l'itération en cours.

La syntaxe d'une boucle `for` est la suivante :

`for` *compteur* `:=` *debut* `to / downto` *fin* `do` *action*

On utilise `to` quand l'indice va en augmentant et `downto` quand il descend.

programme 15 : `boucle_for.p`

```
{
indices ascendant et descendant dans une boucle for
}

program boucle_for;

procedure monter(n : integer);
var
   i : integer;
begin
   for i:=1 to n do
      write(i, ' ');
```

```
   writeln();
end; { monter }

procedure descendre(n : integer);
var
   i : integer;
begin
   for i:=n downto 1 do
   begin
      write('indice     ', i);
      writeln(' dans la boucle descendante');
   end;
end;


begin
   writeln('essai de for ascendant :');
   monter(5);
   writeln();
   writeln('essai de for descendant :');
   descendre(4);
end.
```

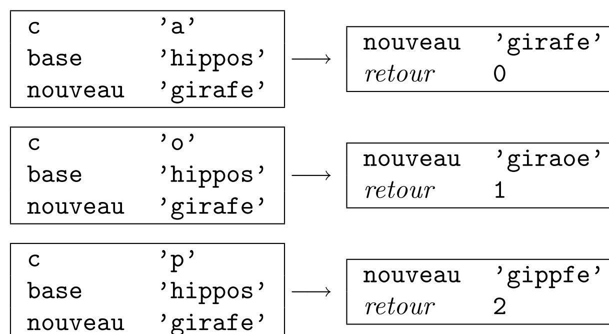
Exercice 5.5 *La* `i`*-ème lettre d'une chaîne de caractères est obtenue en ajoutant* `[i]` *après le nom d'une variable contenant cette chaîne, la première lettre ayant un indice* `1` *en* `Pascal`*.*

On dispose d'une fonction `length` *qui prend en argument une chaîne de caractères et qui renvoie un entier contenant sa longueur (ie son nombre de lettres).*

Ecrire une fonction `nombre_de_voyelles` *qui prend en argument une chaîne de caractères et renvoie le nombre de voyelles contenues dans cette chaîne. Pour cela, vous pouvez récupérer la fonction* `est_voyelle` *de l'exercice 5.4.*

Exercice 5.6 *Ecrire une fonction* `remplacer` *qui prend en argument un paramètre* `c` *de type* `char` *et deux paramètres* `base` *et* `nouveau` *de type* `string` supposés de même longueur *(aucune vérification à ce sujet ne doit être faite) et recopie les occurences de* `c` *qui apparaissent dans* `base` *dans la chaîne* `nouveau`*. Cette fonction doit renvoyer le nombre de caractères ainsi mis à jour.*

exemples :



5.2.2 `repeat ... until ...`

On utilise la boucle `repeat` quand on connaît une condition d'arrêt.
La syntaxe d'une boucle `repeat` est la suivante :

`repeat` *action* `until` *condition*

On remarque donc que l'*action* qui apparaît dans une boucle `repeat` est toujours exécutée au moins une fois.

Dans une boucle, on peut avoir besoin d'un compteur. Dans une boucle `for`, il apparaît naturellement, mais ce n'est pas le cas dans une boucle `repeat` ou dans une boucle `while` (section 5.2.3). Dans ces derniers cas, on ne doit pas oublier de mettre à jour le paramètre. Cela peut être fait grâce aux fonctions standards `inc` (qui *incrémente* le paramètre, c'est-à-dire augmente sa valeur de 1) et `dec` (qui *décrémente* le paramètre, c'est-à-dire diminue sa valeur de 1).

Exercice 5.7 *Ecrire une procédure qui demande à l'utilisateur un entier entre 0 et 20 : si l'utilisateur répond correctement, le mot* `merci` *s'affiche, sinon la procédure continue à demander un tel entier tant que la réponse donnée n'est pas correcte.*

5.2.3 `while ... do ...`

On utilise la boucle `while` quand on connaît une condition de non arrêt.
La syntaxe d'une boucle `while` est la suivante :

`while` *condition* `do` *action*

Si la *condition* d'une boucle `while` est fausse dès le départ, l'*action* de la boucle n'est jamais exécutée.

Exercice 5.8 *Ecrire une fonction* `existe` *qui prend en argument un paramètre de type* `char` *et un paramètre de type* `string` *et renvoie* `true` *si le caractère apparaît dans la chaîne de caractères et* `false` *sinon.*

Exercice 5.9 *Ecrire un programme qui permet de jouer au pendu. On supposera que le mot à trouver est rentré en dur dans le programme. Vous pouvez vous servir des fonctions écrites aux exercices 5.6 et 5.8.*

5.2.4 Les sorties et sauts de boucle

Pour sortir d'une boucle *avant* sa fin "normale", on utilise `break` : le reste de la boucle n'est pas exécuté, le test de fin de boucle pas évalué et on passe directement à la suite.

Pour sortir de l'exécution courante de la boucle et sauter à la suivante, on utilise `continue` : le reste du code correspondant à l'exécution courante n'est pas exécuté, le test de fin de boucle est évalué avant de repasser (éventuellement) dans la boucle.

Des exemples sont fournis par les programmes 16 et 17.

programme 16 : `break.p`

```
{
utilisation de la sortie de boucle break
}

program sortie_break;

function carre(n : integer):boolean;
{teste si un entier est un carre}
var
   i : integer;
begin
   carre:=false;
   i:=1;
   while i*i<= n do begin
      if i*i=n then begin
         carre:=true;
         break;
      end;
      inc(i);
   end;
end;

function exemple(borne_inf, borne_sup : integer):integer;
{renvoie le plus petit carre entre borne_inf et borne_sup }
var
   i : integer;
begin
   exemple:=-1;
   for i:=borne_inf to borne_sup do
      if carre(i) then begin
         exemple:=i;
         break;
      end;
end; { exemple }


var
   borne_inf, borne_sup, res : integer;
begin
   write('borne inf et borne sup : ');
   read(borne_inf);
   read(borne_sup);
   res:=exemple(borne_inf, borne_sup);
   if res=-1 then
      writeln('Pas de carre entre ', borne_inf, ' et ', borne_sup, '.')
   else
      writeln('Le plus petit carre entre ', borne_inf, ' et ', borne_sup, ' est ', res, '.');
end.
```

programme 17 : `continue.p`

```
{
utilisation du saut de boucle continue
}

program saut_continue;

procedure compter(ch : string);
var
   lg, i              : integer;
   nb_mots, nb_lettres : integer;
begin
   nb_mots:=1;
   nb_lettres:=0;
   lg:=length(ch);
   for i:=1 to lg do begin
      if ch[i]=' ' then begin
         inc(nb_mots);
         continue;
      end;
      inc(nb_lettres);
   end;
   writeln(ch, ' : ', nb_lettres, ' lettres, ', nb_mots, ' mots.');
end; { compter }


var
   ch : string;
begin
   write('Ecrivez une phrase : ');
   read(ch);
   compter(ch);
end.
```

6 Un peu de récursivité

On a déjà vu plusieurs exemple de fonction ou procédure qui font appel à une autre fonction ou procédure (par exemple les programmes 10, 16 ou encore le programme de l'exercice 5.5).

Rien n'empêche qu'une fonction ou une procédure s'appelle elle-même, on parle alors de *récursivité* (ou de fonction/procédure *récursive*).

Un exemple est donné par le bout de programme 18; on suppose qu'on dispose d'une fonction `est_voyelle` (voir exercices 5.2 et 5.4).

programme 18 : `rec.p`

```
{
exemple de fonction recursive
ceci est un morceau de programme, on suppose deja acquis :
 - la fonction est_voyelle
 - le programme principal
}

function nb_voyelles(ch : string):integer;
begin
   if length(ch)=0 then
      nb_voyelles:=0
   else begin
      if est_voyelle(ch[1]) then
         nb_voyelles:=1+nb_voyelles(copy(ch, 2, length(ch)-1))
      else
         nb_voyelles:=nb_voyelles(copy(ch, 2, length(ch)-1));
   end;
end; { nb_voyelles }
```

Exercice 6.1 *Ecrire une procédure qui propose un menu avec deux possibilités pour l'utilisateur : "arrêter" ou "continuer", chacune associée à un numéro. Si l'utilisateur demande à arrêter, la procédure rend la main, sinon elle se relance.*

7 Les types structurés

Pour l'instant nous avons vu des types de base en `Pascal` (`boolean`, `char`, `integer` ou `real` essentiellement) et un type qui n'existe pas de base, mais qui nous est fourni (`string`). Le but de cette section est d'apprendre à créer nous-mêmes des nouveaux types.

7.1 Déclaration d'un nouveau type

On peut déclarer un nouveau type de la façon suivante :

type *nouveau_type* = *ancien_type*;

Dans le programme 19, on définit un type `entier` qui n'est autre qu'un synonyme du type `integer` et qu'on utilise par la suite.

programme 19 : `type.p`

```
{
definition d'un nouveau type : le type entier, synonyme du type integer
}

type
   entier = integer;

var
   n : entier;
```

```
begin
   n:=3;
   writeln(n);
end.
```

7.2 Tableaux

Quand on veut avoir un certain nombre de renseignements du même type (par exemple 10 entiers), on utilise des *tableaux*, en précisant les indices des cases extrêmales *et* le type de chaque case.

exemple :

```
{ tableau d'entiers, cases numerotees de 1 a 10 }
T : array [1..10] of integer;
```

Dans le programme 20, on suppose qu'on a un code qui à une lettre de l'alphabet en associe une autre. Dans la fonction `codage`, on prend en argument une chaîne de caractères et on renvoie la chaîne codée (pour simplifier, on suppose qu'on se restreint à des chaînes de caractères constituées uniquement de lettres minuscules, d'espaces et de signes de ponctuation). La fonction `initialisation` est là uniquement pour initialiser le code.

programme 20 : `code.p`

```
{
exemple d'utilisation de tableaux dans des codes
}

program code;


procedure initialisation(var code_engendre : array of char);
var
   n : integer;
   c : char;
begin
   c:='f';
   for n:=1 to 26 do begin
      if c>'z' then c:='a';
      code_engendre[n]:=c;
      inc(c);
   end;
end; { initialisation }



function codage(phrase : string; code : array of char):string;
var
   phrase_codee : string;
   i, len, num  : integer;
begin
   len:=length(phrase);
   for i:=1 to len do begin
```

```
      num := ord(phrase[i]) - ord('a');
      phrase_codee := phrase_codee + code[num];
   end;
   codage:=phrase_codee;
end; { codage }



var
   mon_code  : array [1..26] of char;
   ma_phrase : string;
begin
   initialisation(mon_code);
   write('votre phrase : ');
   read(ma_phrase);
   writeln('phrase codee : ', codage(ma_phrase, mon_code));
end.
```

On remarque deux points particuliers :

- On ne renvoie pas de résultat de type `array ...`. Pour pallier à ce problème, dans `initialisation`, on prend un paramètre transmis *par valeur*.
- Les paramètres de type `array ...` sont fournis aux fonctions et procédures *sans spécification de leur taille*.

Exercice 7.1 *Ecrire une procédure qui prend en argument un paramètre de type tableau d'entiers correspondant à un numéro de téléphone et demande à l'utilisateur de mettre à jour son numéro s'il n'est pas correct.*

Exercice 7.2 *Ecrire la fonction* `decodage` *qui prend en argument un code fourni de la même façon que dans le programme 20 et une chaîne de caractères et renvoie la chaîne de caractères correspondant à l'argument décodé.*

Pour trouver le "décodage" d'un caractère, on pourra procéder de la façon suivante : on parcourt le tableau donnant le code jusqu'à trouver la lettre souhaitée, puis on transforme l'indice du tableau en caractère.

On peut bien entendu définir un nouveau type "tableau de tant de caractères". Dans ce cas, ce nouveau type peut être utilisé comme type de retour d'une fonction. Comparez les programmes 20 et 21.

programme 21 : `code2.p`

```
{
exemple d'utilisation d'un nouveau type "tableau" dans des codes
}

program code2;

type
   t_code = array[1..26] of char;
```

```
function initialisation():t_code;
var
   n             : integer;
   c             : char;
   code_engendre : t_code;
begin
   c:='f';
   for n:=1 to 26 do begin
      if c>'z' then c:='a';
      code_engendre[n]:=c;
      inc(c);
   end;
   initialisation:=code_engendre;
end; { initialisation }



function codage(phrase : string; code : t_code):string;
var
   phrase_codee : string;
   i, len, num  : integer;
begin
   len:=length(phrase);
   for i:=1 to len do begin
      num := ord(phrase[i]) - ord('a') + 1;
      phrase_codee := phrase_codee + code[num];
   end;
   codage:=phrase_codee;
end; { codage }



var
   mon_code  : t_code;
   ma_phrase : string;
begin
   mon_code:=initialisation();
   write('votre phrase : ');
   read(ma_phrase);
   writeln('phrase codee : ', codage(ma_phrase, mon_code));
end.
```

7.3 Enregistrements (record)

Dans la section 7.2, on a utilisé un nouveau type qui permet de stocker plusieurs informations *de même type*. Nous allons maintenant voir comment stocker plusieurs informations de *types différents*.

Imaginons qu'on veuille stocker des renseignements concernant un auteur (nom, prénom, date de naissance, nombre d'œuvres écrites) : on peut commencer par créer un type `date` contenant trois entiers représentant la date, puis un type `auteur` contenant tous les renseignements nécessaires concernant

l'auteur, comme dans le programme 22.

---

programme 22 : `auteur.p`

```
{
exemples d'enregistrements (record)
}

program p_auteur;

type
   date   = record
               mm,jj,aa : integer;
            end;
   auteur = record
               nom, prenom : string;
               naissance   : date;
               oeuvres     : integer;
            end;


function saisie_date():date;
var
   d : date;
begin
   write('jour : ');
   readln(d.jj);
   write('mois : ');
   readln(d.mm);
   write('annee : ');
   readln(d.aa);
   saisie_date:=d;
end; { saisie_date }



procedure affiche_date(d : date);
begin
   write(d.jj, '/', d.mm, '/', d.aa);
end; { affiche_date }




function saisie_auteur():auteur;
var
   aa : auteur;
begin
   writeln('Saisie d''un nouvel auteur :');
   write('son nom : ');
   readln(aa.nom);
```

```
   write('son prenom : ');
   readln(aa.prenom);
   write('sa date de naissance : ');
   aa.naissance:=saisie_date();
   write('nombre d''oeuvres ecrites : ');
   readln(aa.oeuvres);
   saisie_auteur:=aa;
end; { saisie_auteur }



procedure affiche_auteur(a : auteur);
begin
   write(a.prenom, ' ', a.nom, ', ne(e) le ');
   affiche_date(a.naissance);
   writeln(', a ecrit ', a.oeuvres, ' oeuvre(s).');
end; { affiche_auteur }




var
   x : auteur;
begin
   x:=saisie_auteur();
   affiche_auteur(x);
end.
```

Exercice 7.3 *A l'aide d'un tableau, réécrire une procédure* `affiche_date` *qui affiche la date sous un format plus agréable, comme "14 juillet 1789" au lieu de 14/07/1789.*