# Fonctions

## 1 Les fonctions

Une *fonction* permet d'*abstraire* et de *nommer* un calcul ou une expression. Une fois définie, une fonction peut être utilisée comme une nouvelle *expression* du langage.

Contrairement aux instructions, un appel à une fonction ne doit pas modifier l'état courant de l'environnement.

### 1.1 Spécification d'une fonction

Spécifier une fonction, c'est
- choisir un identificateur pour la nommer,
- préciser le nombre et le type[1] de ses paramètres, et les nommer,
- indiquer les conditions d'utilisation (CU) que doivent vérifier les paramètres lors d'un appel à la fonction,
- indiquer le type de la valeur qu'elle retourne,
- et indiquer quelle est la relation entre la valeur retournée et les paramètres qui lui sont passés.

### 1.2 Déclaration d'une fonction en Pascal

```
function <Nom_Fonction>(<liste_parametres>) : <type_resultat>;
  (* déclarations *)
begin
  (* séquence d'instructions *)
end {function};
```

où
- <Nom_Fonction> est l'identificateur qui nomme la fonction,
- <liste_parametres> est la liste des *paramètres formels*,
- et <type_resultat> est le type de la valeur qu'elle retourne.

L'*entête* de la fonction est la première ligne de sa déclaration.

La dernière instruction d'une fonction est toujours une *affectation* de la forme :

```
<Nom_Fonction> := <expression>
```

où <expression> est une expression dont la valeur est du type retournée par la fonction.

#### Exemple

```
// prédicat qui retourne le booléen true lorsque la carte
// au sommet du tas Tas est un coeur ou un carreau
// CU : le tas Tas ne doit pas être vide
function rouge(Tas : TASPOSSIBLES) : BOOLEAN;
begin
  rouge := (SommetCoeur(Tas) or SommetCarreau(Tas));
end {rouge};
```

Les quatre premières lignes établissent la spécification de la fonction
- les trois premières (les commentaires) précisent le lien entre les paramètres et la valeur retournée, et indiquent une condition d'utilisation (CU) (ces précisions sont destinées au programmeur et non au compilateur qui ignore les commentaires),
- la quatrième (l'entête) indique le nom de la fonction, le nombre et le type des paramètres, ainsi que le type de la valeur retournée (informations utiles à la fois pour le programmeur et pour le compilateur)

[1] pour l'instant on peut se contenter des types `Boolean`, `Couleurs` et `TasPossibles`

**Où ?** La déclaration d'une fonction se fait dans la partie déclaration d'un programme.

```
1  // auteur : EW
2  // date   : nov 2007
3  // objet  : vider les cartes rouges dans le tas 2
4  //        et les noires dans le tas 3
5  program separerRougesEtNoires;
6  uses
7     cartes;
8
9  // prédicat qui retourne le booléen true lorsque la carte
10 // au sommet du tas Tas est un coeur ou un carreau
11 // CU : le tas Tas ne doit pas être vide
12 function rouge(Tas : TASPOSSIBLES) : BOOLEAN;
13 begin
14    rouge := (SommetCoeur(Tas) or SommetCarreau(Tas));
15 end {rouge};
16
17 begin
18    // initialisation des tas
19    initTas(1,'[T+K+C+P]');
20    initTas(2,'');
21    initTas(3,'');
22    initTas(4,'');
23
24    // separation des noires et des rouges
25    while tasNonVide(1) do begin
26       if rouge(1) then begin
27          deplacerSommet(1,2);
28       end else begin
29          deplacerSommet(1,3);
30       end {if};
31    end {while};
32 end.
```

Dans cette exemple, on trouve la déclaration de la fonction `rouge` (lignes 9-15) après la clause **uses** `cartes` (lignes 6-7) et avant le corps du programme principal (lignes 17-32).

## 1.3 Appel à une fonction

Pour faire appel à une fonction, il suffit d'écrire son nom accompagné des *paramètres effectifs*.

Dans le programme donné en exemple ci-dessus, on trouve un appel à la fonction `rouge` (ligne 26) avec l'entier 1 comme paramètre effectif.

Il faut bien entendu que les paramètres effectifs lors de l'appel vérifient les conditions d'utilisation. Ainsi, l'appel à la fonction `rouge` dans l'instruction conditionnelle de l'exemple ci-dessus n'est pas valide si le tas 1 est vide.

## 1.4 Variables locales

Certaines fonctions nécessitent des variables qui leur sont propres pour écrire l'algorithme qui les réalise. Ces variables sont appelées des *variables locales*. Ces variables sont déclarées dans la partie déclaration de la fonction, entre son entête et le bloc **begin end** qui la définit.

Par exemple, la fonction `factorielle` qui suit uilise deux variables locales `f` et `i`.

```
// factorielle(n) =n!
function factorielle(n : CARDINAL) : CARDINAL;
var
   f, i : CARDINAL;
begin
   f := 1;
   for i := 1 to n do begin
```

```
      f := f*i;
   end {for};
   factorielle := f;
end {factorielle};
```

Les variables locales à une fonction ne sont définies que pour cette fonction et ne peuvent pas être utilisées dans d'autres contextes, en particulier dans le programme principal.

Par exemple, si le programme principal utilise une variable `f`, celle-ci doit être déclarée comme une *variable globale* et n'a rien à voir avec la variable locale du même nom déclarée dans `factorielle`. Le programme principal qui suit

```
var
  f : CARDINAL;
begin
  f := 4;
  writeln(f,'! = ',factorielle(f));
  writeln('f = ',f);
end.
```

produira l'affichage

```
4! = 24
f = 4
```

ce qui montre que la variable globale `f`, qui vaut 4 après le calcule de `factorielle(f)`, n'est pas la même que la variable locale `f` qui vaut 24 après ce calcul.

Une fonction peut utiliser toute variable globale ou constante, à consition qu'elle soit déclarée avant la focntion.

Par exemple

```
const
   UN = 1;

// plusUn(n) = n + 1
function plusUn(n : CARDINAL) : CARDINAL;
begin
   plusUn := n + UN;
end {plusUn};
```

## 1.5 Fonctions à plusieurs paramètres

Une fonction peut avoir plus d'un paramètre. Ce paramètres peuvent être de types identiques ou non.

```
// puissance1(a,n) = a^n
function puissance1(a,n : CARDINAL) : CARDINAL;
var
  p, i : CARDINAL;
begin
  p := 1;
  for i := 1 to n do begin
    p := p*a;
  end  {for};
  puissance1 := p;
end {puissance1};
```

```
// puissance2(a,n) = a^n
function puissance2(a : REAL; n : CARDINAL) : REAL;
var
```

```
  p : REAL;
  i : CARDINAL;
begin
  p := 1;
  for i := 1 to n do begin
    p := p*a;
  end  {for};
  puissance2 := p;
end {puissance2};
```

Un appel à une fonction à plusieurs paramètres doit se faire avec le bon nombre et le bon type des paramètres. Ainsi, dans l'exemple

```
1  var
2    n,m : CARDINAL;
3    x : REAL;
4  begin
5    ...
6    writeln(puissance1(n,m));
7    writeln(puissance2(x,m));
8    writeln(puissance1(m));
9    writeln(puissance1(x,m));
10 end.
```

les instructions des lignes 6 et 7 sont correctes car les appels de fonction se font avec le bon nombre et le bon type des paramètres. En revanche les instructions des lignes 8 et 9 ne le sont pas, la première car il manque un paramètre à l'appel de `puissance1`, et la seconde car le premier paramètre est de mauvais type.

## 1.6 Fonctions sans paramètre

Il est possible de définir des fonctions sans paramètre.

```
// un() = 1
function un() : CARDINAL;
begin
   un := 1;
end {un};
```

Cette fonction est constante : l'instruction **`writeln(un())`** affichera toujours 1.

On peut réaliser des fonctions sana paramètre non constantes.

```
// valeurdex() = valeur de la variable globale x
function valeurdex() : CARDINAL;
begin
   un := x;
end {valeurdex};
```

La valeur retournée par cette fonction dépend de la valeur de la variable globale `x` (qui doit être déclarée dans le programme avant la fonction, et qui doit être de type `CARDINAL`).

On peut réaliser des fonctions sans paramètres un peu plus intéressante. En voici un exemple.

```
// pileOuFace() = résultat aléatoire valant 0 ou 1
function pileOuFace() : CARDINAL;
begin
   pileOuFace := random(2);
end {pileOuFace};
```

La fonction prédéfinie **`random(`**$n$**`)`** renvoie un entier au hasard compris entre 0 et $n-1$ inclus.

L'instruction

```
if pileOuFace()=1 then begin
   writeln('gagné !');
end else begin
   writeln('perdu !');
end {if};
```

### 1.7 Intérêt des fonctions

- reflet de l'analyse du problème
- modularité
- lisibilité des programmes
- factorisation du code

## 2 Erreurs à ne pas commettre

### 2.1 Appel de fonction ≠ instruction

Attention, un appel à une fonction n'est pas une instruction.

Dans le programme ci-dessous, il n'y a pas de sens à considérer que `f(3)` (ligne 9) est une instrution : `f(3)` est une expression qui a une valeur (ici 6), mais n'agit en aucune façon sur son environnement (pas d'affichage produit, pas de modification de la valeur d'une variable).

```
1  program appelIncorrect;
2
3  function f (n : CARDINAL) : CARDINAL;
4  begin
5     f := 2*n;
6  end ;
7
8  begin
9     f(3);
10 end.
```

Avec Free Pascal, si on ajoute la directive de compilation `{$X-}` en tête du programme, avant la ligne **program**, le compilateur refusera de considérer tout appel à une fonction comme une instruction. Par exemple, pour le programme précédent, le compilateur signalera le message d'erreur `Error : Illegal expression`.

Un appel de fonction est une expression, et ne peut donc être effectué que dans un contexte où une expression est valide, comme par exemple la partie droite d'une affectation, ou bien la condition d'une instruction conditionnelle ou répétitive Tant Que, ...

### 2.2 Fonction ≠ variable locale

On pourrait être tenté d'utiliser le nom de la fonction comme une variable locale (après tout la dernière instruction est une affectation dont la partie gauch est le nom de la fonction).

Par exemple, on pourrait être tenté de réécrire la fonction `factorielle` de la façon suivante

```
function factorielle(n : CARDINAL) : CARDINAL;
var
   i : CARDINAL;
begin
   factorielle := 1;
   for i := 1 to n do begin
      factorielle := factorielle*i;
```

```
    end {for};
end {factorielle};
```

Si cela est autorisé avec les options par défaut du compilateur FREE PASCAL, cela n'est pas possible avec l'option `-Mdelphi` utilisée dans la commande `fpcompile`, qui provoque un message d'erreur de types incompatibles qui est, pour cet exemple,

`Error : Incompatible types : got "factorielle(LongWord) :DWord" expected "LongInt"`.

# 3 Exercices

**Exercice 1.** Écrire les entêtes des fonctions de l'unité cartes. (Le type qui définit les couleurs se nomme `Couleurs`, et celui qui définit les tas se nomme `TasPossibles`)

**Exercice 2.** Réaliser et tester une fonction `TasTousVides` qui retourne un booléen indiquant si tous les tas sont vides.

**Exercice 3.**

**Question 1.** Réaliser une fonction `TasMax` qui donne le numéro du tas dont le sommet a la carte de valeur la plus élevée, parmi les deux tas passés en paramètres. Ne pas oublier les CU.
**Question 2.** Réaliser une fonction `TasMax3` analogue à la précédente pour trois tas passés en paramètres.
**Question 3.** Même chose pour quatre tas ! (paramètres ou non ?)

**Exercice 4.** *Ou exclusif* Réalisez une fonction qui retourne le ou-exclusif des deux booléens passés en paramètres.

**Exercice 5.** *Années bissextiles* Une année *bissextile* est une année qui comprend 366 jours au lieu des 365 pour les années ordinaires.

Depuis l'instauration du calendrier grégorien, sont bissextiles, les années :
- divisibles par 4 mais non divisibles par 100
- ou bien divisibles par 400.

Réalisez la fonction spécifiée ci-dessous

```
// estBissextile(annee) = vrai si annee est une année bissextile
//                      = faux sinon
funtion estBissextile(annee : CARDINAL) : BOOLEAN;
```

**Exercice 6.** *Le plus grand*
**Question 1.** Réalisez une fonction qui retourne le plus grand des deux entiers passés en paramètres.
**Question 2.** Réalisez de deux façons une fonction qui retourne le plus grand des trois entiers passés en paramètres

1. en utilisant définie dans la question précédente ;
2. puis sans utiliser cette fonction.

**Exercice 7.** *Coefficients binomiaux*

Étant donnés deux entiers $0 \leq p \leq n$, on définit le coefficient binomial par

$$\left( \begin{array}{c} n \\ p \end{array} \right) = \frac{n!}{(n-p)!p!}.$$

Réalisez de deux façons la fonction qui à deux entiers associe le coefficient binomial $\left( \begin{array}{c} n \\ p \end{array} \right)$

1. une première façon utilisant la focntion `factorielle` ;
2. une seconde façon exploitant la simplification

$$\left( \begin{array}{c} n \\ p \end{array} \right) = \frac{n \times (n-1) \times \ldots \times (n-p+1)}{p \times (p-1) \times \ldots \times 1}.$$

**Exercice 8.** *Fonction polynomiale*

Réalisez une fonction qui permet de calculer les valeurs de la fonction réelle de variable réelle

$$f(x) = 3x^4 - x^2 + 2x + 1.$$

**Exercice 9.** *Plancher et plafond d'un nombre réel*

On appelle *plancher* d'un nombre réel le plus grand nombre entier inférieur ou égal à $x$. On le note $\lfloor x \rfloor$. Ce nombre est aussi appelé partie entière de $x$.

On appelle *plafond* d'un nombre réel le plus petit nombre entier supérieur ou égal à $x$. On le note $\lceil x \rceil$. Ces deux nombres coïncident lorsque $x$ est un nombre entier.

$$\begin{aligned} \lfloor \sqrt{2} \rfloor &= 1 \\ \lceil \sqrt{2} \rceil &= 2 \\ \lfloor -\sqrt{2} \rfloor &= -2 \\ \lceil -\sqrt{2} \rceil &= -1 \end{aligned}$$

**Question 1.** Réalisez une fonction nommée `plancher` qui calcule le plancher du nombre réel $x$ passé en paramètre[2]. Il est conseillé de se concentrer d'abord sur le cas des réels positifs ou nuls, puis de généraliser.

**Question 2.** De même, réalisez une fonction nommée `plafond` pour le plafond dun réel.

**Exercice 10.** *simulation d'un dé* Réalisez une fonction qui simule un dé. Cette fonction renvoie au hasard un entier compris entre 1 et 6.

**Exercice 11.** *Tas ordonné ?* Réaliser et tester un prédicat `tas_1_ordonne` qui retourne vrai si le tas numéro 1 est rangé dans l'ordre croissant. Le tas vide est ordonné, les tas d'une cartes le sont aussi. Un tas est ordonné lorsque toute carte du tas est supérieure au sens large à toutes celles qu'elle surmonte. (comme pour l'exercice précédent plusieurs versions sont possibles).

[2]Cette fonction est prédéfinie dans l'unité `math` et se nomme `floor`.