Les fonctions et les procédures

Motivation

- Dans certains algorithmes, on peut trouver les mêmes calculs ou les mêmes traitements plusieurs fois dans des endroits différents. *A chaque fois on est alors obligé de répéter la séquence d'actions qui représente ces calculs ou ces traitements.*
- Un algorithme qui correspond à un ***problème complexe*** serait très long et difficile. Pour rendre cet algorithme plus simple et plus compréhensible, *on découpe le problème complexe en* ***sous-problèmes*** *(tâches) faciles à résoudre*. Ainsi, l'algorithme sera *découpé à son tour en* ***sous-algorithmes*** *(modules)*. Chaque sous-algorithme est la solution d'un sous-problème.

Exemple:

Considérons le problème des délibérations. Il est difficile de le résoudre par un seul algorithme. Il est plus approprié de le décomposer en tâches suivantes:

- ✓ Saisie des notes par matière;
- ✓ Calcul de la moyenne générale;
- ✓ Affichage de la liste des étudiants admis;
- ✓ Affichage de la liste des rattrapages par module;
- ✓ … etc.

A chaque tâche est associé un sous-algorithme (module).

❖ Ils existent deux types de modules, qui sont *les fonctions* et *les procédures*.

Structure d'un algorithme contenant des modules

Entête de l'algorithme principal

```
algorithme nom_algorithme
```

Partie déclaration de l'algorithme principal

```
Déclaration des constantes et des variables globales
Définition des sous-algorithmes (modules)
```

Corps de l'algorithme principal

```
début    {Algorithme principal}
  Action 1
  Action 2
  …
  Action n
fin
```

Les fonctions

Une fonction est un module (groupe d'instructions) désigné par un nom, qui, à partir des données produit (rend, retourne) une valeur unique (un seul résultat) à travers son nom (c.à.d. le résultat sera stocké dans le nom de la fonction).

Déclaration (définition) d'une fonction

Une fonction est déclarée (définie) dans la partie déclaration de l'algorithme principal en utilisant le mot-clé *fonction* selon la syntaxe suivante:

```
fonction <nom_fonction> (paramètres-formels):<type_fonction>
<partie déclaration de la fonction>
début
| <actions>
| <nom_fonction> ← résultat
finFonction
```

Avec:

- ✓ <*nom_fonction*>: représente le nom de la fonction;
- ✓ *paramètres-formels*: représentent les données de la fonction. Ils doivent être séparés par des virgules;
- ✓ <*type_fonction*>: représente le type de la fonction (c.à.d. le type du résultat retourné par la fonction). Il doit être un type simple (*entier*, *réel*, *car*, *chaine* ou *booléen*).
- ✓ *résulat*: représente de résultat retourné par la fonction.

Remarques:

1. La déclaration des paramètres de la fonction est similaire à la déclaration des variables, mais sans utiliser le mot-clé **var**.

2. La partie déclaration de la fonction peut être vide.

Exemple: déclaration (définition) d'une fonction qui calcule le produit de deux nombres réels.

```
fonction produit (x,y:réel):réel
var z:réel
début
| z ← x*y
| produit ← z
finFonction
```

Ou bien ⟺

```
fonction produit (x,y:réel):réel
début
|  Produit ← x*y
finFonction
```

Appel d'une fonction

Une fonction ne sera exécutée que si elle est *appelée*. Elle peut être appelée (exécutée) dans l'algorithme principal, dans un autre module ou bien dans elle-même, selon la syntaxe suivante:

```
<nom_fonction> (paramètres-effectifs)
```

Exemple: appel de la fonction `produit`.

- ✓ `produit (a,b)`

Paramètres formels et paramètres effectifs

- Les paramètres écrits lors de la définition d'une fonction sont dits des *paramètres formels* ;
- Les paramètres écrits lors de l'appel d'une fonction sont dits des *paramètres effectifs*.

Remarques:

1. *Le nombre, le type et l'ordre des paramètres effectifs doivent être identiques aux nombre, type et ordre des paramètres formels.*

2. *Les paramètres effectifs doivent être séparés par des virgules.*

3. *Un paramètre effectif peut être une variable, une constante, une expression, ou bien l'appel d'une autre fonction.*

Exemples:

- ✓ `produit (a,2.5)`
- ✓ `produit (c,produit(4.0,3.0))`
- ✓ `produit (c,(a+b)/2)`

4. *L'appel d'une fonction ne doit pas, tout seul, constituer une instruction (il doit être une partie d'une instruction).*

Exemples:

- ✓ `Ecrire (produit (a,b))`
- ✓ `d ← produit (a,b)`
- ✓ **`Si`** `(produit(a,b) >0)` **`alors`** …
- ✓ … etc.

Exemple: écriture et appel de la fonction *produit* dans un algorithme.

Ecrire un algorithme qui lit deux nombres réels a et b, ensuite il affiche leur produit en utilisant une fonction.

```
algorithme prod
var a,b:réel
fonction produit (x,y:réel):réel
début
  produit ← x*y
finFonction
début      {algorithme principal}
écrire("donner la valeur de a:")
lire (a)
écrire("donner la valeur de b:")
lire (b)
écrire("a*b= ", produit(a,b))
fin
```

- Paramètres formels: x,y
- Paramètres effectifs: a,b

Exercices

1. Dans un algorithme**,**

- Ecrire une fonction (*fact*) qui calcule le factoriel d'un nombre entier positif.
- Lire deux valeurs Q et P, ensuite calculer et afficher C = (Q!(Q-P)!)/P!

2. Ecrire un algorithme qui permet de:

- Remplir un tableau T de 100 éléments avec des nombres entiers.
- En utilisant une fonction (*existe*), vérifier si un élément *e* existe dans le tableau T ou non.
  - Si l'élément existe dans le tableau, l'algorithme doit afficher le message "*l'élément e existe dans T*".
  - Dans le cas contraire, il doit afficher le message "*l'élément e n'existe pas dans T*".

3. Ecrire une fonction qui calcule le PGCD de deux nombres entiers naturels.

- Dans l'algorithme principal, calculer et afficher le PPMC de deux nombres entiers naturels A et B, tel que: PPMC (A,B) = A*B/PGCD (A,B).

Solutions

```
Algorithme exo1
Var Q,P,C:entier
Fonction fact(n:entier):entier
var i,f:entier
Début
    f ← 1
    Pour i=2 à n faire
        f ← f*i
    FinPour
    fact ← f
FinFonction
Début          {Algorithme principal}
    Ecrire ("Entrer la valeur de Q:")
    Lire(Q)
    Ecrire ("Entrer la valeur de P:")
    Lire(P)
    C ← fact(Q)*fact(Q-P)/fact(P)
    Ecrire ("C = ",C)
Fin
```

```
Algorithme exo3
var A,B:entier
Fonction pgcd(x, y:entier):entier;
Début
    TantQue (x*y ≠ 0) faire
        Si x>y alors
            x ← x-y
        Sinon
            y ← y-x;
        FinSi
    FinTQ
    Si x=0 alors
        pgcd ← y
    Sinon
        pgcd ← x;
    FinSi
FinFonction
Début      {Algorithme principal}
    Ecrire("A = ")
    Lire(A)
    Ecrire("B = ")
    Lire(B)
    Ecrire("Le PPMC de A et B = ", A*B/pgcd(A,B))
Fin
```

```
Algorithme exo2
Const N = 100
Type Tab = tableau [1..N]de entier
var T:Tab
    e,i:entier
Fonction existe(A:Tab,x:entier):booléen
Var j:entier
    b:booléen
Début
    j ← 1
    b ← faux
    TantQue (j≤N ET b=faux)faire
        Si (A[i]=x) alors
            b ← vrai
        FinSi
        j ← j+1
    FinTQ
    existe ← b
FinFonction
Début          {Algorithme principal}
    Ecrire ("Entrer les éléments de T:")
    Pour i=1 à N faire
        Lire(T[i])
    FinPour
    Ecrire ("Entrer l'élément e")
    Lire(e)
    Si (existe(T,e) = vrai)alors
        Ecrire("L'élément e existe dans T")
    Sinon
        Ecrire("L'élément e n'existe pas dans T")
    Finsi
Fin
```

Les procédures

Une procédure est une suite d'instructions, désignée par un *nom*, qui permet d'effectuer des actions par un simple appel dans un algorithme ou dans un autre sous-algorithme. Au contraire à une fonction, une procédure ne retourne pas de valeur. Par contre, elle peut générer un résultat composé ou ne générer aucun résultat. ***Les résultats d'une procédure seront stockés dans les paramètres*.**

Déclaration (définition) d'une procédure

D'une façon similaire à une fonction, une procédure est déclarée (définie) dans la partie déclaration de l'algorithme principal. On utilise plutôt le mot-clé *Procédure* selon la syntaxe suivante:

```
Procédure <nom_procédure> (paramètres-formels)
<partie déclaration de la procédure>
Début
    <actions>
FinProc
```

Avec:

- ✓ <*nom_procédure*>: représente le nom de la procédure;
- ✓ *paramètres-formels*: représentent les DONNEES et les RESULTATS de la procédure. Ils doivent être séparés par des virgules;

Exemple: on suppose qu'on a un type *vecteur* définit comme suit:

```
Type vecteur = tableau[1..100] de réel
```

- La procédure qui affiche à l'écran les éléments d'un tableau de type *vecteur* est:

```
Procédure afficher_tableau (T:vecteur)
var i:entier
Début
    Pour i=1 à 100 faire
        Ecrire(T[i])
    FinPour
FinProc
```

Appel d'une procédure

Comme dans le cas d'une fonction, une procédure ne sera exécutée que si elle est *appelée*. Elle peut être appelée (exécutée) dans l'algorithme principal, dans un autre module ou dans elle même, selon la syntaxe suivante:

```
<nom_procédure> (paramètres-effectifs)
```

Exemple: appel de la procédure `afficher_tableau` dans un algorithme:

```
Algorithme exemple_appel
Type vecteur = tableau[1..100] de réel
var A,B:vecteur
-----------------------------------------------
 Procédure afficher_tableau (T:vecteur)
 var i:entier
 Début
 |  Pour i=1 à 100 faire
 |  |  Ecrire(T[i])
 |  FinPour
 FinProc
-----------------------------------------------
Début   {Algorithme principal}
|  Afficher_tableau(A)
|  Afficher_tableau(B)
Fin
```

Appels de la procédure *afficher_tableau* (affichage des tableaux A et B)

Remarques:

- L'appel d'une procédure doit constituer, tout seul, une instruction.
- La notion de paramètre formel et paramètre effectif avec une procédure reste inchangée par rapport à une fonction. (T est un paramètre formel et A et B sont des paramètres effectifs de la procédure *afficher_tableau*).
- Le nombre, le type et l'ordre des paramètres effectifs doivent être identiques aux nombre, type et ordre des paramètres formels d'une procédure.

Procédure simple et procédure avec arguments (paramètres)

- La procédure *afficher_tableau* de l'exemple précédent a un paramètre (T). Ce type de procédure est dit procédure avec arguments.
- Quand une procédure ne possède pas de paramètres, elle est dite procédure simple.

Exemple: une procédure simple qui affiche les jours de la semaine:

```
Procédure jours_semaine ()
Début
|  Ecrire("Jour 1: samedi")
|  Ecrire("Jour 2: dimanche")
|  Ecrire("Jour 3: lundi")
|  Ecrire("Jour 4: mardi")
|  Ecrire("Jour 5: mercredi")
|  Ecrire("Jour 6: jeudi")
|  Ecrire("Jour 7: vendredi")
FinProc
```

Pour appeler cette procédure, on écrit: `jours_semaine ()`

Variables locales et variables globales

- ✓ Lorsqu'une variable est utilisée seulement dans un module, elle sera déclarée dans la partie déclaration de ce module. Dans ce cas, on dit que cette variable est une **variable locale**. Par conséquent, elle *n'est pas visible* à l'extérieur de ce module.
- ✓ Une variable déclarée dans la partie déclaration de l'algorithme principal est dite **variable globale**. Une variable globale est accessible (utilisable) dans l'algorithme principal, ainsi que dans tous les modules qui appartiennent à cet algorithme.

Exemple. Soit l'algorithme suivant:

```
Algorithme locales_globales
var a,b,c,d:réel

Procédure proc1 (x:réel)
Var n:réel
Début
  a ← a*x
  n ← c*2
FinProc

Procédure proc2 (y:réel)
Var m:réel
Début
  b ← b*y
  m ← d*2
FinProc

Début  {Algorithme principal}
  Lire (a,b,c,d)
  proc1(c)
  proc2(d)
Fin
```

- ✓ **a**, **b**, **c** et **d** : *variables globales* (accessibles dans l'algorithme principal, ainsi que dans les procédures `proc1` et `proc2`).
- ✓ **n**: *variable locale* à la procédure `proc1` (accessible seulement dans la procédure `proc1`).
- ✓ **m**: *variable locale* à la procédure `proc2` (accessible seulement dans la procédure `proc2`)

Remarques:

- ❖ On peut avoir une variable locale dans un module, qui a le même nom qu'une variable globale. Dans ce cas, le module utilise sa variable locale et non pas la variable globale.
- ❖ Deux variables locales de deux modules différents peuvent avoir le même nom.

Passage des paramètres

- ✓ Un paramètre est dit *en entrée* s'il fait partie des données de la procédure.
- ✓ Un paramètre est dit *en sortie* s'il fait partie des résultats de la procédure.
- ✓ Un paramètre est dit *en entrée/sortie* s'il fait partie, à la fois, des données et des résultats de la procédure.

Ils existent deux types de passage des paramètres: *passage par valeur* et *passage par adresse* (par variable ou par référence). Si un paramètre est précédé par le mot-clé **var**, il est passé par adresse (par variable). S'il n'est pas précédé par mot mot-clé **var**, il est passé par valeur.

- Les paramètres en sortie ou en entrée/sortie doivent être passés *par adresse*.
- Les paramètres en entrée doivent être passés *par valeur*.

a. Passage par valeur

Dans ce type de passage des paramètres, la procédure n'a pas le droit de modifier les valeurs des paramètres effectifs. Elle crée une copie du paramètre effectif et utilise cette copie. Donc, la valeur d'un paramètre effectif ne sera jamais modifiée.

Exemple:

```
Algorithme passage_par_valeur
var N:entier
Procédure proc1 (A:entier)
Début
  A ← A*2
FinProc
Début   {Algorithme principal}
N ← 5
proc1(N)
Ecrire("N = ",N)
Fin
```

✓ Après l'exécution de cet algorithme, il affiche:

```
N = 5
```

✓ La procédure `proc1` n'a pas modifié la valeur du paramètre `N`.

a. Passage par adresse

Dans ce type de passage des paramètres, la procédure peut modifier les valeurs des paramètres effectifs.

Exemple:

```
Algorithme passage_par_adresse
var N:entier
Procédure proc2 (var A:entier)
Début
  A ← A*2
FinProc
Début   {Algorithme principal}
N ← 5
proc2(N)
Ecrire("N = ",N)
Fin
```

✓ Après l'exécution de cet algorithme, il affiche:

```
N = 10
```

✓ La procédure `proc2` a modifié la valeur du paramètre `N`.

Différences entre une fonction et une procédure

Fonction	Procédure
✓ *Entête:* Fonction <*nom*>(paramètres):<*type_résultat*> ✓ *Appel:* variable ← <*nom*>(paramètres_effectis) Ecrire(<*nom*>(paramètres_effectis)) … Etc. ✓ Doit retourner une valeur unique. ✓ Passage des paramètres par valeur.	✓ *Entête:* Procédure <*nom*>(paramètres) ✓ *Appel:* <*nom*>(paramètres_effectis) ✓ Peut ne pas calculer de résultats, ou calculer un résultat composé. ✓ Passage des paramètres par valeur et par adresse.

Exercices:

1. Ecrire une procédure qui permet de calculer la périphérie et la surface d'un rectangle.

2. Ecrire une procédure (*minmax*) qui permet de calculer le minimum et le maximum de deux nombres entiers passés en paramètres.

3. On considère le type *vecteur* qui correspond aux tableaux de 100 éléments de type entier.

- Faire la déclaration du type *vecteur*.
- Ecrire une fonction (*abs*) qui calcule la valeur absolue d'un nombre entier.
- Ecrire procédure (*remplir_tableau*) qui permet de remplir un tableau de type *vecteur*.
- Ecrire une procédure (*remplacer*) qui permet de remplacer les éléments d'un tableau de type *vecteur* par leurs valeurs absolues (utiliser la fonction *abs*).
- **Dans l'algorithme principal**:
  - Remplir deux tableaux A et B.
  - Remplacer les éléments des tableaux A et B en utilisant la procédure *remplacer*.

Solutions

Exercice 1.

```
Procédure rectangle (L,W:réel, var P,S:réel)
Début
    P ← (L+W)*2
    S ← L*W
FinProc
```

Exercice 2.

```
Procédure minmax (a,b:entier, var min,max:entier)
Début
    Si (a ≤ b) alors
        min ← a
        max ← b
    Sinon
        min ← a
        max ← b
    FinSi
FinProc
```

Exercice 3.

```
Algorithme exo3
Type vecteur = tableau [1..100]de entier
var A,B:vecteur

Fonction abs(x:entier):entier
Début
  Si (x < 0) alors
    abs ← -x
  Sinon
    abs ← x
  FinSi
FinFonction

Procédure remplir_tableau(var T:vecteur)
var i:entier
Début
  Pour i=1 à 100 faire
    Lire(T[i])
  FinPour
FinProc

Procédure remplacer(var T:vecteur)
var i:entier
Début
  Pour i=1 à 100 faire
    T[i]← abs(T[i])
  FinPour
FinProc

Début  {Algorithme principal}
  remplir_tableau(A)
  remplir_tableau(B)
  remplacer(A)
  remplacer(B)
Fin
```