Chapitre 8

Fonctions

8.1 Introduction

Nous avons vu au précédent chapitre qu'un segment de code pouvait acquérir de l'autonomie, porter un nom, posséder ses propres variables et recevoir d'un autre segment des valeurs passées en paramètres. Nous appelions ce segment *procédure*. Les procédures ne sont pas les seuls exemples de *sous-routines*. Il existe également des *fonctions*.

Au sens mathématique, une fonction est une relation qui fait correspondre à un élément d'un ensemble appelé domaine un et un seul élément d'un autre ensemble. Les fonctions font l'objet de nombreuses études en mathématiques (on les représente généralement sous la forme de courbes). On peut aussi naïvement présenter une fonction comme un mécanisme qui permet de triturer des données et d'en produire une nouvelle qui est prévisible et obéit à une loi [1].

Des exemples de fonctions courantes sont évidemment toutes les fonctions arithmétiques, trigonométriques, statistiques ou financières. On trouve également des exemples dans la vie de tous les jours : le prix d'une course en taxi résulte d'un calcul fonctionnel (une prise en charge fixe plus un nombre de centimes au kilomètre), de même que le change des monnaies (fonction de la parité monétaire et d'une commission correspondant à un pourcentage).

En informatique, une fonction est une sous-routine qui renvoie une valeur, le plus souvent en tenant compte des paramètres qui lui sont fournis.

8.2 Valeur de retour d'une fonction

La fonction se distingue donc de la procédure par le fait qu'elle renvoie une valeur. Cette remarque entraîne deux conséquences :

– la fonction doit avoir un type (du moins dans les langages qui exigent un type également pour les variables). En général, la déclaration de la fonction utilise un moyen similaire aux déclarations de variables. En théorie, tous les types que nous connaissons peuvent caractériser la valeur renvoyée par une fonction. Nous aurons donc des fonctions entières, flottantes, caractère ou logique. En pratique, le langage C a du mal à renvoyer une chaîne de caractères [2]. Nous nous abstiendrons donc de créer de telles fonctions.

1. L'exemple d'une fonction aléatoire comme `HasardMax()` n'est qu'un leurre. En réalité, la valeur « aléatoire » serait parfaitement prévisible pour peu qu'on connaisse la semence employée. Au départ d'une même semence, une fonction aléatoire produit toujours la même valeur. C'est d'ailleurs pour cela qu'on les appelle des fonctions pseudo-aléatoires.

2. Si la chaîne est gérée comme une variable locale, sa valeur peut difficilement être renvoyée au programme appelant puisqu'en C une chaîne est identifiée par une adresse. Cette adresse, forcément comprise dans l'espace d'adressage local, deviendra illégale sitôt l'exécution de la fonction terminée. Il est possible pourtant de créer en

– la fonction doit disposer d'un moyen syntaxique quelconque pour marquer le renvoi de cette valeur. Les langages se partagent deux méthodes : l'une consiste à utiliser un mot-clé particulier pour terminer la fonction et renvoyer la valeur (il s'agit en général de `return` ou `renvoyer`) ; l'autre méthode utilise le nom de la fonction à gauche d'un opérateur d'affectation (cas du Pascal et de Visual BASIC).

8.3 Définition d'une fonction

Par convention, nous donnerons à nos fonctions un nom commençant par `fn` et par `fl` pour les fonctions renvoyant une valeur logique. On peut aussi employer des préfixes adaptés à nos types de variables (`fi`, `ff`, `fc`, `fs` et `fu`), mais la version 4.0.x du compilateur ne vérifie pas la concordance du préfixe avec le type renvoyé [3]. Nous ferons évidemment précéder ce nom par le type de la valeur renvoyée par la fonction. Les paramètres éventuels viendront après, exactement comme dans la déclaration d'une procédure.

Le corps de la fonction possède une structure similaire à celle du corps d'une procédure, notamment par la présence de déclarations de variables locales. Remarquons cependant que la dernière instruction est généralement `renvoyer`.

```
Code Fonction entière fnMaFonction(Valeur entière iPara)
Définitions locales
     Variables entières iNbre2, iNbre2
Fin des locales
  instruction1
  instruction2
  renvoyer iNbre1
Fin de fonction
```

Remarque : l'instruction `renvoyer` a pour effet d'interrompre le déroulement de la fonction. Toute instruction placée après elle sera tout simplement ignorée. Notons qu'il en va autrement si l'instruction est placée dans une structure alternative. À ce moment, l'interruption de la fonction ne se produira qu'au cas où la condition serait vraie :

```
Si condition alors
   renvoyer 1
Fin de si
iVar3 <- 7 * iMachin
Renvoyer iVar3
```

Il faut donc distinguer `renvoyer` qui marque **la fin de l'exécution** de la fonction de `fin de fonction` qui désigne la **fin du code** de la fonction.

C des fonctions qui renvoient une valeur pointeur qui a été allouée dans un autre espace (le tas). On fait appel pour cela à la fonction `malloc`. Cela suppose des mécanismes que notre pseudo-code n'est pas en mesure de gérer.

3. L'implémentation d'un tel contrôle ne me paraît pas d'une grande urgence. Il nécessiterait la réécriture d'une bonne série de règles internes. La reconnaissance des expressions logiques joue par contre un rôle capital dans la conception de la version courante de laf2c. Il est donc obligatoire de faire la distinction entre le préfixe `fl` et tous les autres.

Remarque importante :

Aucun compilateur n'interdit de placer une instruction `renvoyer` à l'intérieur d'une boucle. Il s'agit pourtant d'une grave erreur de programmation. En procédant de la sorte, on place en fait deux points de sortie dans la boucle, ce qui est explicitement interdit par les règles de l'algorithmique [4].

Exemple

Voici un exemple de fonction calculant la factorielle d'un nombre donné en paramètre. Rappelons que la factorielle d'un nombre entier est le produit de ce nombre par tous les nombres entiers qui lui sont inférieurs.

```
Code Fonction entière fnFactorielle(Valeur entière iNombre)
   Définitions locales
      Variables entières iCompteur, iFac
   Fin des locales
   Init
      iFac <- 1
   Compter avec iCompteur de 1 à iNombre
      iFac <- iFac * iCompteur
   Fin de compter
   Renvoyer iFac
fin de fonction
```

8.4 Utilisation d'une fonction

Comme une procédure, une fonction doit être connue du compilateur au moment où il rencontre son appel. Nous reprendrons la technique du langage C qui consiste à déclarer la fonction à l'aide d'un prototype

```
Prototype Fonction entière fnMafonction
   (Valeur entière iPara)
```

L'appel de la fonction consiste simplement à utiliser le nom de la fonction suivi de ses éventuels arguments dans une expression quelconque. Voici quelques exemples :

```
Début du traitement
   iVar1 <- fnMafonction(5)
   Afficher fnMafonction(iVar1) + 7
   iVar2 <- racine(fnMafonction(iVar3)) + 7
Fin du traitement
```

Une fonction peut figurer en n'importe quel endroit où pourrait figurer une variable, une expression ou une constante du même type :

- dans une affectation

```
iVar <- fnValeurabsolue(iBenefice)
```

- dans une expression

4. S'il faut insister plus encore, on peut dire qu'une instruction `Renvoyer` dans une boucle est un moyen radical pour s'offrir une seconde session (ou une troisième).

```
iVar <- fnValeurabsolue(iX)*1.25
```

– dans une instruction d'affichage

```
Afficher fnValeurabsolue(iVar)
```

– dans un appel de sous-routine

```
iX <- fnFactorielle(fnValeurAbsolue(iVar))
```

– dans un test (dans le cas d'une fonction logique).

```
Tant que
   flTravailNonTermine()
Répéter
...
```

8.5 Pièges fréquents avec les fonctions

Il arrive fréquemment que des étudiants ne résistent pas à la tentation de faire parler les fonctions. Voici quelques conseils à méditer :

– une fonction qui calcule une valeur sur base d'autres valeurs doit obtenir ces valeurs par le passage des paramètres, non en les demandant à l'utilisateur. Il faut penser que la fonction risque d'être employée dans un programme qui tourne sans intervention humaine. S'il apparaît que les valeurs doivent être lues au clavier, c'est le programme appelant qui s'en chargera. Voici deux versions d'une fonction qui calcule la valeur absolue d'un nombre entier. La première est pratiquement inutilisable.

```
/* Version très maladroite */
Code Fonction entière fnAbs1()
   Définitions locales
      Variable entière iNombre
   Fin des locales
   Afficher "Entrez le nombre : "
   Lire iNombre
   Si iNombre < 0 alors
      Renvoyer -iNombre
   sinon
      Renvoyer iNombre
   fin de si
Fin de fonction

/* Version conseillée */
Code Fonction entière fnAbs2(Valeur entière iNombre)
   si iNombre < 0 alors
      Renvoyer -iNombre
   sinon
      Renvoyer iNombre
   Fin de si
Fin de fonction
```

En pratique, on ne trouvera pas d'instructions `lire` ou `afficher` dans une sous-routine, sauf si l'utilité de cette routine est de réaliser un affichage ou une saisie.

– de même, si une fonction demande des paramètres précis (par exemple, positifs dans le cas d'une racine ou non nul dans le cas du calcul de l'inverse d'un nombre), il n'appartient pas à la fonction de réagir à des valeurs non prévues. La première version va afficher un message inattendu si l'argument vaut 0. La seconde provoquera un plantage, dont seul le programmeur du programme appelant sera responsable.

```
Code Fonction flottante fnInverse1
  (Valeur flottante fNombre)
   Si fNombre == 0 alors
      Afficher "0 n'a pas d'inverse"
   sinon
      Renvoyer 1/fNombre
   fin de si
Fin de fonction

Code Fonction flottante fnInverse2
  (Valeur flottante fNombre)
   Renvoyer 1/fNombre
Fin de fonction
```

Une troisième méthode permet de régler le problème dans notre pseudo code. L'instruction `Vérifier <condition> Message <texte>` constitue un moyen de s'assurer que le « contrat » est bien respecté.

```
Code Fonction flottante fnInverse2
   (Valeur flottante fNombre)
   Vérifier fNombre != 0 Message "0 n'a pas d'inverse"
   Renvoyer 1/fNombre
Fin de fonction
```

Quand la condition n'est pas respectée, le programme s'arrête et affiche le message prévu. Exemple d'exécution :

```
Assertion non respectée: 0 n'a pas d'inverse
Erreur d'exécution dans fnInverse2.
```

8.6 Sous-routines récursives

L'une des utilisations les plus étonnantes des fonctions se trouve dans l'emploi de fonctions récursives. Une fonction récursive est une fonction qui s'appelle elle-même. L'esprit trouve choquant, à première vue, qu'on puisse écrire une portion de code qui s'utilise lui-même pour mener sa tâche à bien. Si on réfléchit bien, les mathématiques nous ont habitué à ce type de démarche avec les *démonstrations par récurrence*. Ce qui serait choquant, ce serait une fonction qui s'appelle elle-même par erreur [5]. Une récursivité bien contrôlée permet de réaliser des programmes extrêmement concis.

Le contrôle de la récursivité consiste à trouver un cas de base dans lequel la récursivité ne s'applique pas. Si chaque appel récursif nous rapproche de ce cas de base, nous sommes assurés que la série infinie des appels récursifs sera brisée.

5. Il existe une forme plus insidieuse de récursivité nommé *récursivité indirecte*. Elle survient quand une sous-routine A appelle une sous-routine B qui à son tour fait appel à A. Il peut éventuellement y avoir des intermédiaires. On peut facilement créer un programme comprenant une récursivité indirecte sans s'en apercevoir.

8.6.1 Retour sur la factorielle

Reprenons le cas de la factorielle. À la définition donnée plus haut, nous pouvons opposer une autre définition, équivalente :

$0! = 1$

$n! = n.(n-1)!$

Cette définition peut facilement être traduite en termes fonctionnels :

```
Code Fonction entière fnFactorielle(Valeur entière iNombre)
   Si iNombre < 2 alors
      Renvoyer 1
   sinon
      Renvoyer iNombre*fnFactorielle(iNombre-1)
   fin de si
Fin de fonction
```

De manière évidente, chaque niveau d'appel de la fonction diminue de 1 la valeur du paramètre qu'il a reçu avant de faire l'appel récursif. Quelle que soit la valeur de `iNombre` transmise au premier appel de la fonction, on est certain qu'un appel récursif aura finalement 1 comme argument. 0 et les nombres négatifs renverront 1, bien que la factorielle ne soit pas définie pour les nombres négatifs.

Le mécanisme de la récursivité se base sur l'existence d'un contexte propre à chaque activation de la fonction. Lorqu'on exécute `fnFactorielle` avec l'argument 5, la fonction va s'activer cinq fois, avec cinq contextes comprenant chaque fois l'adresse de retour et la valeur locale du paramètre `iNombre`. La conservation des différents contextes se fait généralement sur la pile du processeur. La profondeur possible des appels n'est évidemment pas infinie. Dans une programmation responsable, il faudra donc évaluer le risque d'un plantage consécutif à une saturation de la pile. Dans le cas de la factorielle, nous aurons des problèmes de dépassement dans les registres entiers bien avant la saturation de la pile.

Si le mécanisme de la récursivité nous dispense parfois d'une programmation pénible, il n'est pas très performant. En effet chaque appel récursif nécessite un empilement du contexte et son dépilement au retour. Il sera toujours possible de traduire un algorithme récursif à l'aide d'une boucle. Dans le cas de la factorielle, la valeur renvoyée à chaque niveau est mémorisée dans un accumulateur.

8.6.2 Affichage en binaire

Autre exemple de récursivité : afficher un nombre entier dans sa forme binaire. Pour afficher un nombre entier, il suffit de procéder à une série de divisions par 2 et d'afficher le reste à chaque fois.

```
13:2=6 -> reste 1
6:2=3 -> reste 0
3:2=1 -> reste 1
1:2=0 -> reste 1
```

13 s'écrit bien 1101 en binaire. Ce qui pose problème, c'est que si on affiche les nombres au fur et à mesure, on lira 1011, parce que l'ordinateur n'est pas capable d'écrire de droite à gauche. On peut mémoriser les différents nombres dans un tableau, mais la solution récursive est beaucoup plus simple à mettre en œuvre. Nous allons déterminer un cas trivial : l'affichage d'un nombre ne comportant qu'un seul bit (0 ou 1). Pour les autres cas, nous traitons d'abord récursivement le nombre divisé par 2 avant d'afficher le reste à la droite des chiffres produits

par ce traitement. Nous obtenons une procédure composée d'une seule instruction alternative et ne comportant pour toute variable que le nombre à afficher. Difficile de faire plus court.

```
Code Procédure prBinaire(Valeur entière iNombre)
   Si iNombre < 2 alors
      Afficher iNombre
   sinon
     /* traitement des chiffres les plus à gauche */
      prBinaire(iNombre/2)
     /* traitement du dernier chiffre de droite */
      Afficher iNombre % 2
   fin de si
Fin de procédure
```

8.6.3 Les tours de Hanoï

Une vieille légende prétend que des moines vietnamiens sont occupés depuis des siècles à déplacer des disques empilés en tours. Initialement, 64 disques de tailles décroissantes étaient empilés. Les moines se relaient pour déplacer les disques un à un, de manière à reconstituer une tour à côté de la première. Lorsqu'ils auront terminé, le temps de la fin du monde sera venu.

Les moines respectent scrupuleusement deux règles :

- il est permis d'utiliser une troisième tour intermédiaire formée de disques empilés.
- chaque disque doit toujours être plus petit que celui sur lequel il repose, à l'exception du disque de base qui repose directement sur le sol.

La solution de ce problème peut être extrêmement complexe. La première question étant de savoir sur quel endroit du sol il a fallu déposer le plus petit disque lorsque le travail a commencé. À certains moment, il est possible de déposer un disque sur l'une ou l'autre des tours de destination. La moindre erreur peut avoir des conséquences dramatiques. Cependant l'algorithme de résolution récursif est d'une limpidité extraordinaire :

```
Code Procédure prHanoi
   (valeur entière iTourOrigine,
    valeur entière iTourDestination,
    valeur entière iTourIntermediaire,
    valeur entière iHauteur)

   Si iHauteur==1 alors
      prDeplacerDisque(iTourOrigine,iTourDestination)
   Sinon
      prHanoi(iTourOrigine,iTourIntermediaire,
              iTourDestination,iHauteur-1)
      prDeplacerDisque(iTourOrigine,iTourDestination)
      prHanoi(iTourIntermediaire,iTourDestination,
              iTourOrigine,iHauteur-1)
   Fin de si
Fin de procédure
```

Le cas trivial est celui du déplacement d'une tour formée d'un seul disque : il suffit de déplacer ce disque. Pour les tours plus hautes, il faut déplacer la tour formée par tous les anneaux sauf celui du bas vers une tour intermédiaire, déplacer l'anneau du bas vers la destination puis replacer la tour déposée sur l'intermédiaire sur sa destination finale. Le déplacement de la tour

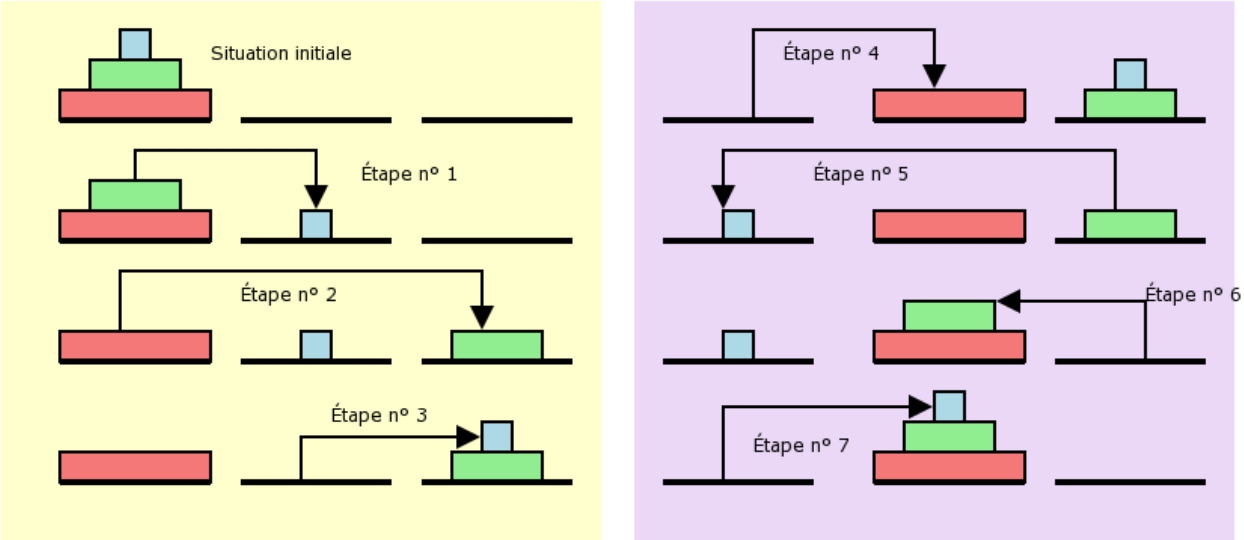
se fait par un appel récursif et comme on ne s'occupe pas du plus grand anneau, la dimension décroît à chaque appel récursif. Elle finira par valoir 1.

Pour que le programme fonctionne, il faut évidemment donner une représentation des tours, ce que nous n'avons pas fait ici. À titre d'exemple, voici une réécriture d'une version texte de la procédure `prDeplacerDisque()`.

```
Code Procédure prDeplacerDisque
    (valeur entière iTourOrigine,
     valeur entière iTourDestination)

   Long texte
      iDeplacements &
      ". Déplacer le sommet de la tour " &
      iTourOrigine & " vers la tour " &
      iTourDestination & ~|
   fin de texte
   Incrémenter iDeplacements
fin de procédure
```

la variable globale `iDeplacement` est uniquement employée pour numéroter les déplacements. Voici un exemple d'exécution pour des tours d'une hauteur de trois disques :




```
profc@p3:~/personnel/laf/notes > hanoi1
hauteur de la tour 3
1. Déplacer le sommet de la tour 1 vers la tour 2
2. Déplacer le sommet de la tour 1 vers la tour 3
3. Déplacer le sommet de la tour 2 vers la tour 3
4. Déplacer le sommet de la tour 1 vers la tour 2
5. Déplacer le sommet de la tour 3 vers la tour 1
6. Déplacer le sommet de la tour 3 vers la tour 2
7. Déplacer le sommet de la tour 1 vers la tour 2
nombre de déplacements : 7
profc@p3:~/personnel/laf/notes >
```

Vocabulaire

FONCTION, PROCÉDURE, RÉCURSIVITÉ, SOUS-ROUTINE.

Exercices

VIII-1. Écrire une fonction qui calcule le périmètre d'un rectangle dont les dimensions sont passées en paramètres.

VIII-2. Écrire une fonction qui calcule la moyenne des valeurs contenues dans un tableau de 10 entiers.

VIII-3. Écrire une fonction qui calcule l'inverse d'un nombre (5 devient $\frac{1}{5}$).

VIII-4. Écrire une fonction qui calcule la nième puissance entière d'un nombre quelconque.

VIII-5. Écrire une fonction qui calcule le δ d'une équation du second degré, dont on fournit les trois coefficients. Rappelons que $\delta = b^2 - 4ac$.

VIII-6. Écrire une fonction qui renvoie le plus grand des deux nombres passés en arguments. Faire de même avec le plus petit (`fnMax` et `fnMin`).

VIII-7. Écrire une fonction `fnArrondi` qui renvoie l'arrondi commercial d'un nombre non entier passé en argument. Rappelons qu'on arrondit à partir de 0,5. ex. 4.49 à 4 et 4.50 à 5.

VIII-8. Écrire deux fonctions `fnEuro` et `fnFb` qui opèrent la conversion des monnaies ancienne et nouvelle. Rappelons que le coefficient de conversion, pour le franc belge, est 40,3399. La loi prévoit que la conversion en euro propose 2 décimales et que la conversion en FB opère un arrondi commercial.

VIII-9. Écrire une fonction qui demande à l'utilisateur d'entrer un nombre compris entre 1 et 10. En cas d'erreur, la demande sera reformulée jusqu'à ce que l'utilisateur tape un nombre valable. Ce nombre sera renvoyé au programme principal.

VIII-10. Écrire une fonction qui renvoie la position éventuelle dans un tableau du premier nombre supérieur à une valeur donnée. En cas d'échec, elle renverra -1. Le tableau n'est pas trié. Il ne doit pas être modifié.

VIII-11. Écrire une fonction qui calcule le quotient exact et le reste de la division de deux nombres. Pour rappel, une fonction ne peut renvoyer qu'une seule valeur. Il faut donc trouver une astuce pour en renvoyer deux !