II Sous-programmes

Fonctions et Procédures

II.1 Notion de sous-programme et paramètres

Définition : *Sous-programme*

Un sous-programme est donc un *élément de programme* possédant un *nom* qui peut être *appelé* à partir d'un programme (ou d'un autre sous-programme) afin de lui faire exécuter la tâche prévue ou lui faire calculer certaines valeurs à partir de données (*arguments*) qui lui sont fournies.

En fait, un sous-programme représente une *entité logique homogène*. C'est un moyen d'abstraction et d'aide à la conception ➜ il permet de nommer une action complexe.

Un sous-programme calcule un certain nombre de résultats en fonction d'un certain nombre d'arguments. On appellera *paramètres* l'ensemble des arguments du sous-programme.

- *donnée* : un tel paramètre correspond à une donnée de l'algorithme. Il ne peut donc figurer qu'en partie droite d'une instruction ;
- *donnée modifiée* : ce sont des paramètres qui correspondent à des données de l'algorithme, mais que l'on pourra aussi utiliser en partie gauche d'une instruction et donc leur affecter une nouvelle valeur faisant office de résultat ;
- *résultat* : ce type de paramètres est utilisé par certains langages qui n'autorisent normalement qu'un seul résultat pour un sous-programme ; il permet alors de renvoyer plusieurs résultats.

II.2 Fonctions

Définition : *Fonctions*

Une *fonction* est un sous-programme considéré comme une expression dont l'évaluation correspond au résultat de l'exécution du sous-programme.

La fin du calcul du résultat et donc la fin de l'exécution de la fonction seront marqués par l'utilisation de l'opération *Return*. Dans ce cas, le type du résultat définit le type de l'expression (*type de la fonction*).

Soit la fonction mathématique :

$$f : int \times int \rightarrow int$$

$$(x, y) \mapsto f(x, y) = x^9 + 8x^3u7 + xy^3$$

On dit que `x` et `y` sont des variables muettes.

Le sous-programme correspondant à la fonction `f` peut être utilisée en lui fournissant des valeurs réelles qui seront affectées aux variables muettes

Définition : *Paramètres formels et réels*

Les *paramètres formels* correspondent aux variables muettes.

Les *paramètres réels* sont les paramètres formels affectés de valeurs effectives.

Définition : *Déclaration* de fonction

La *déclaration* d'une *fonction* comprend son nom, la liste des *paramètres formels* et le *type* de chacun d'eux, et le type de la fonction.

Exemple de syntaxe pour une déclaration de fonction :

```
func f(x: Int, y: Int) -> Int
```

ou plus simplement :

```
func f(x, y: Int) -> Int
```

Définition : *Définition de fonction*

La *définition* d'une *fonction* comprend sa *déclaration* et la *séquence d'instructions* décrivant l'action à entreprendre par la fonction ; dans la séquence d'instructions doit *obligatoirement* figurer une instruction `Return` qui spécifie quel est le résultat de la fonction.

Exemple de syntaxe pour une déclaration de fonction :

```
func min(x: Int, y: Int) -> Int
  var r : Int
  r = (x + y)/2
  if r == x then
    return r+1
  else
    return r
  endif
endfunc
```

Définition : *Définition de fonction*

La *définition* d'une *fonction* comprend sa *déclaration* et la *séquence d'instructions* décrivant l'action à entreprendre par la fonction ; dans la séquence d'instructions doit *obligatoirement* figurer une instruction `Return` qui spécifie quel est le résultat de la fonction.

Demander l'exécution d'une fonction (d'un sous-programme particulier) se dit réaliser l'*appel de fonction*

Une fonction étant une expression, il suffit d'écrire son nom en partie droite d'une instruction.

Elle est évaluée et son évaluation demande l'affectation des paramètres réels aux paramètres formels ainsi que l'exécution des lignes d'instructions de la fonction se terminant par l'exécution de l'instruction `Return`.

II.3 Procédures

Les fonctions sont des sous-programmes particuliers qui sont des expressions (donc évaluables et ayant un type). Lorsqu'on a besoin d'un sous-programme pour factoriser une séquence d'instructions sans nécessairement être une expression évaluable, ou nécessitant des paramètres *données modifiées*, on utilisera la notion de *Procédure*.

II.3 Procédures

Les fonctions sont des sous-programmes particuliers qui sont des expressions (donc évaluables et ayant un type). Lorsqu'on a besoin d'un sous-programme pour factoriser une séquence d'instructions sans nécessairement être une expression évaluable, ou nécessitant des paramètres *données modifiées*, on utilisera la notion de *Procédure*.

Le mot clef `Procedure` sert à spécifier un sous-programme comme une procédure. On utilisera les mots clef `Donnée`, `Donnée modifiée` ou `Résultat` pour spécifier les paramètres.

Définition : *Déclaration d'une Procédure*

```
Proc nomProc(in x: Int, inout y: Int, out z: Int)
    # explication de la procédure
    # données
    # x représente…
    # y représente…
    # z aura pour résultat…
    var v : Type
    ligne 1
    ligne 2
    …
endProc
```

II.4 Transmission des données entre appelant et appelé

-> *Transmission des données entre appelant et appelé*

La question qui est posée est la suivante :

> *Quel est l'effet, sur un argument réel passé à un sous-programme, d'une modification éventuelle de la valeur de l'argument formel correspondant dans le corps du sous-programme ?*

II.4.1 Pass. de param. : *par valeur*

C'est le type de passage par défaut de nombreux langages.

Il concerne essentiellement les paramètres *donnée*.

Le sous-programme se contente alors de recopier la valeurs des paramètres réels dans une zone mémoire locale au sous-programme et y accédera par l'intermédiaire du nom du paramètre formel qui agira comme une variable locale.

La valeur du paramètre réel ne sera donc pas affectée par tout changement du paramètre puisque l'on agit sur une *copie* de la valeur et non plus sur la valeur initiale.

II.4.2 Pass. de param. : *par recopie*

Ce type de passage est utilisé essentiellement pour les paramètres *résultat*

On utilisera pour le paramètre formel une zone mémoire locale non initialisée dans laquelle on placera la valeur résultat lors de l'affectation de celui-ci. Au moment du retour du sous-programme, cette valeur sera recopiée dans le paramètre réel.

II.4.3 Pass. de param. : *copie/recopie*

Ce type de passage concerne en particulier les paramètres *donnée modifiée*.

Il correspond à l'application du passage par valeur à l'appel du sous-programme et au passage par recopie au moment du retour du sous-programme.

Il cumule donc les propriétés des deux types de passage de paramètres.

II.4.4 Pass. de param. : *par adresse*

Ce type de passage peut être utilisé pour les paramètres *donnée modifiée* ou *résultat*.

Il consiste à permettre au sous-programme de manipuler directement le paramètre réel en indiquant au sous-programme l'adresse du paramètre réel.

Ce type de passage est également utilisé pour les types de données pour lesquels il serait trop coûteux en ressource processeur de copier sa valeur.

II.5 Les fonctions Swift

En swift, une procédure se déclare comme une fonction qui n'a pas de type de retour.

Le schéma de déclaration d'une fonction est le suivant :

```
func nomFonction(paramètres) -> Type{
  instructions
  …
}
```

Les paramètres sont passés par *valeur.* Il ne peuvent donc pas être en partie gauche d'une expression.

Exemple (Fonction de calcul de la suite de Fibonacci).

```
/** Return a list containing the Fibonacci series up to n, i.e une
liste où chaque terme est la somme des deux termes qui le précèdent.
 - parameter n: valeur maximum pour un nombre de la suite de Fibonnaci
 - returns: un tableau contenant *n* nombres de la suite de Fibonnaci
 - Precondition: *n* > 0
 - Postcondition: ∀ *i* ∈ fib(*n*), *i* < *n*
 */
func fib(n: Int) -> [Int]{
    var result = [Int]()
    var a = Int(0)
    var b = Int(1)
    var c : Int
    while a < n {
        result.append(a)    // see below
        c = b
        b = a+b
        a = c // a prend la valeur de b à la dernière
itération
```

Quick Help

Declaration `func fib(n: Int) -> [Int]`

Description Return a list containing the Fibonacci series up to n, i.e une liste où chaque terme est la somme des deux termes qui le précèdent.

Precondition

$n > 0$

Postcondition

$\forall\, i \in \text{fib}(n),\ i < n$

Parameters n
valeur maximum pour un nombre de la suite de Fibonnaci

Returns un tableau contenant la suite des nombres de Fibonnaci inférieurs à *n*

Declared In AlgoExemple.playground

Exemple 3.2 (Valeur par défaut des paramètres).

```
func demander_confirmation(message : String, nbmaxessai :
Int = 4, avertissement : String = "Oui ou Non?!") -> Bool{
   var reponse_faite : Bool    = false
   var reponseok     : Bool    = false
   var reponse       : String? = ""
   var nbessai       : Int     = nbmaxessai
   while !reponse_faite && (nbessai > 0){
      if nbessai > 1{print(message) ; reponse = readLine()
}
      else{          print(avertissement) ; reponse =
readLine() }
      if ["o","oui","ok"].contains(reponse){
         reponseok = true ; reponse_faite = true
      }
      if ["n","non"].contains(reponse) {
         reponseok = false ; reponse_faite = true
```

Il est donc possible de spécifier des valeurs par défaut pour les arguments d'une fonction sous la forme d'une affectation dans la déclaration du paramètre.

L'appel à `demander_confirmation` pourra se faire ainsi :

- en ne donnant que le paramètre obligatoire :
  `demander_confirmation`(message: "Confirmez-vous votre réponse?")
- en donnant un des paramètres optionnels :
  `demander_confirmation`(message: "Confirmez-vous votre réponse?",nbmaxessai: 3)
  `demander_confirmation`(message: "Confirmez-vous votre réponse?",avertissement: "Alors !?")
- en donnant tous les paramètres :
  `demander_confirmation`(message: "Confirmez-vous votre