Cours Structures de Données Avancées

Option G.L.
Semestre 3

Chapitre 1 : Rappels

1. Fonctions et procédures

Utiliser le concept de ”conception modulaire” des algorithmes consiste à construire des algorithmes constitués de plusieurs sous algorithmes. Chaque sous algorithme effectue un travail bien précis (correspondant à la résolution d'un sous problème); il reçoit des données et transmet des résultats, qui sont alors réutilisés par le reste de l'algorithme. Ceci permet l'utilisation répétitive d'un même ensemble d'actions primitives en différents points de l'algorithme. Chaque fois que l'on désire exécuter un sous algorithme, on utilise une action pour l'appeler, qui entraînera son exécution. La dernière action du sous algorithme doit provoquer le retour à l'algorithme appelant (principal).
Selon le nombre de résultats rendus par une telle unité, deux schémas ont été définis. Le schéma fonction qui permet de représenter les actions abstraites qui rendent un unique résultat. Le schéma procédure qui permet de représenter une action abstraite quelconque.

1.1 Fonctions

Une fonction est un ensemble d'actions effectuant un calcul sur des paramètres dont les valeurs lui sont 'passées'. La fonction se sert alors de ces paramètres pour calculer un résultat unique qu'elle renvoie en fin de traitement.
Les fonctions qu'on utilise fréquemment dans l'écriture des algorithmes comme par exemple sin, cos, ...etc. sont considérées connues par le programmeur, nous n'avons pas à les redéfinir; nous les utilisons directement, en les insérant dans une expression, comme par exemple: $A \leftarrow a+2*\sin(8, 5)$.
L'utilisateur peut également définir des fonctions qui seront utilisées de la même façon que les fonctions prédéfinies.

Déclaration de fonction

La déclaration d'une fonction se fait de différentes façons suivant le langage de programmation utilisé. Dans notre langage de description algorithmique, nous adoptons la syntaxe suivante:

```
fonction nom_de_la_fonction (mode_de_passage X1 : Type1 , ..., mode_de_passage Xn : Typen
): Type_de_la_fonction

déclaration
... /*déclaration de variables locales à cette fonction*/
Début
... /* une séquence d'actions constituant le corps de la fonction*/
retourner (expression)
/*C'est la dernière action exécutée dans la fonction pour retourner le résultat à l'algorithme appelant*/
fin
```

X1,X2,....,Xn : sont appelés paramètres formels, ce sont les noms génèriques donnés aux entrées futures de la fonction. Ils servent à désigner, dans le corps de la fonction, les objets mis en oeuvre par ses actions.

Appel d'une fonction

Une fonction s'utilise, quelque soit le langage de programmation, en insérant:

nom_de_la_fonction(V1, ... , Vn)

dans une expression (à droite d'une affectation) ou dans une action d'écriture ou comme paramètre d'une autre fonction ou procédure.

• V1, ... , Vn : sont appelés paramètres effectifs.
• Chaque paramètre effectif Vi doit être de même type que le paramètre formel associé. Ainsi, le nombre de paramètres effectifs doit être le même que le nombre de paramètres formels.

Exemple 1: Ecrire une fonction (déclaration) qui calcule le minimum de deux nombres entiers.

```
fonction minimum(donnée a , b : entier) :entier
déclaration
        m :entier
début
        si a < b alors m ←− a sinon m ←− b finsi
        retourner m
fin
X ←− minimum(3,4)
```

Après exécution de cet appel X = 1+3 = 4
L'appel de la fonction minimum(3,4) se déroule ainsi:
• Substitution de x par 3 et y par 4 dans le corps de minimum,
• Le contrôle du programme passe au début des actions dans le corps de la fonction ainsi changé,
• Une fois la fonction finie, le contrôle revient au point ou la fonction a été appélée.

1.2 Procédures

A la différence des fonctions, les procédures ne calculent pas un résultat; elles permettent seulement de modifier l'état du corps de l'algorithme qui les appelle. En effet, les effets de bord (modification des valeurs en mémoire, entrées-sorties avec les dispositifs externes) sont généralement produites par les procédures. Une programmation correcte ne doit pas permettre des effets de bord dans les fonctions. Par contre, une procédure peut souvent remplacer une fonction dans la réalisation d'un calcul. Si par exemple le calcul doit retourner plus d'un résultat, il est plus naturel d'utiliser une procédure. Pour le calcul d'un résultat, on préfère en principe les fonctions. Néanmoins, il est toujours possible de remplacer une fonction par une procédure équivalente. La procédure reprend tous les paramètres de la fonction (en mode donnée) et en rajoute un pour le résultat (en mode résultat).

Déclaration d'une procédure
La syntaxe de la déclaration d'une procédure est la suivante:

```
procédure nom_de_la_procédure(mode_de_passage X1 : Type1 , ..., mode_de_passage Xn : Typen
)
déclaration
... /*déclaration de variables locales à cette procédure*/
Début
... /* une séquence d'actions constituant le corps de la procédure*/
fin
```
La structure est similaire à celle des fonctions.

Appel d'une procédure

L'appel d'une procédure se fait, à partir d'un autre algorithme en écrivant le nom de la procédure appelée, suivi d'une liste de paramètres effectifs en même nombre et de même types que les paramètres formels.

Exemple 2: (Appel de procédure)

```
procédure double (données/résultat x : entier)
        début
        x ←− 2*x;
        fin
algorithme XXX
déclarations
        a : entier
début
        ...
        a ←− 4;
        écrire(a);
        double(a);
        écrire(a);
        ...
fin
```
La procédure double a un effet de bord: la modification de la variable a de l'algorithme XXX.

1.3 Modes de passage des arguments

Les paramètres formels et effectifs sont mis en correspondance deux à deux d'après l'ordre qu'ils occupent dans leur liste respective. En général, cette correspondance se fait selon les trois modes de passage suivants:

• **mode donnée:** le paramètre formel en mode donnée est considéré comme une variable locale dans le corps du sous algorithme. A l'appel, il est initialisé par la valeur du paramètre effectif.

Exemple 3 (Mode donnée)

```
procédure affichage_date (donnée x : date)
début
        écrire(x.jour);
        écrire("/");
        écrire(x.mois);
        écrire("/");
        écrire(x.année);
fin /* x n'a pas été modifié */
algorithme principal
déclarations
        d : date
début
```

```
        d ←− (14, 7, 1999);
        affichage_date(d);
fin
```

Lors de l'appel affichage_date(d):
— la valeur de d (14,7,1999) est transmise à x, qui est ainsi initialisé,
— x est utilisé dans la procédure sans être modifié,
— d a la même valeur avant et après l'appel affichage_date(d).
Il est à noter que tous les paramètres des fonctions sont généralement en mode donnée.

• **mode résultat**: un argument en mode résultat est considéré comme une variable non initialisée dans le corps du sous algorithme. A la fin de l'exécution de ce dernier, la valeur calculée pour cet argument est alors affectée au paramétre effectif.

Exemple 4 (Mode résultat)

```
procédure lecture_date (résultat x : date)
début
        écrire(”jour=”);
        lire(x.jour);
        écrire(”mois=”);
        lire(x.mois);
        écrire(”annee=”);
        lire(x.annee);
fin
algorithme principal
déclarations
        d : date; /* non initialisée */
début
        /* jusqu'ici dans d il n'y a rien /
        lecture_date(d);
        /* ici d est initialisée par les valeurs saisies */
fin
```

Lors de l'appel lecture_date(d):
-la valeur de d n'est pas prise en compte, x n'est pas initialisé, x est calculé dans la procédure, la valeur finale de x est transmise à d.

• **mode donnée/résultat:** ce mode est une combinaison des deux modes précédents. A l'appel, la valeur du paramètre effectif sert à initialiser le paramètre formel. A la fin de l'exécution du sous algorithme la valeur calculée pour l'argument formel en mode donnée/résultat est retournée dans le paramètre effectif.

Exemple 5 : (Mode donnée/résultat)

```
procédure incrémenter (donnée/résultat x : entier)
début
        x ←− x + 1
fin
Algorithme principal
Déclarations
        a :entier
Début
        a ←− 3
        ...
        /* jusqu'ici a = 3 */
        incrémenter(a)
        /* maintenant a = 4 */
Fin
```

Après l'appel incrémenter(a):
• La valeur de a est transmise à x, qui est ainsi initialisé
• x est modifié dans la procédure,
• La valeur finale de x est transmise à a.

2. Récursivité

Aborder le problème de la récursivité dans les sous programme permet une écriture élégante et compréhensible des programmes. Dans certains cas, le sous-problème est une illustration du problème initial, mais pour un cas "plus simple" traité. Par conséquent, la solution du problème s'exprime par rapport à elle-même. On appelle ce phénomène récursivité.

Exemple 6: Le calcul du factoriel d'une valeur entière positive n est donné par:
n! = 1 x 2 x 3.. x (n − 1) x n
Or, (1 x 2 x 3.. x (n − 1) x n = (n − 1)! x n.
Donc, il nous suffit de savoir calculer (n − 1)! pour pouvoir calculer la valeur de n! par multiplication du factoriel de (n-1) avec n. Le sous-problème du calcul de (n − 1)! est le même que le problème initial, mais pour un cas "plus simple", car n − 1 < n.
Le problème a été réduit à lui-même, mais pour un cas plus simple. En programmation, le sous-programme récursif traitant un problème donné, fait un appel à lui-même pour traiter un cas plus simple, ce qui correspond à un appel avec des paramètres différents ("plus simples"): appel récursif. Un appel récursif produit lui-même un autre appel récursif, ..., etc.
Ceci produira une suite infinie d'appels, il faut donc arrêter cette suite au moment où le sous-problème peut être résolu directement: condition d'arrêt.
Dans l'exemple du factoriel, la fonction s'énonce ainsi:

```
fonction factoriel ( donnée n : entier) : entier
déclarations
        sous_résultat, résultat : entier;
début
        si n=1 alors résultat ←− 1 /* condition d'arrêt */
        sinon sous_résultat ←− factorielle (n-1) /* appel récursif */
              résultat ←− sous_résultat * n
        finsi
        retourner résultat;
fin factoriel
```

2.1 Evaluation d'un appel récursif

• Comment est-il possible d'appeler un sous-programme pendant que celui-ci est en train de s'exécuter?
• Comment se fait-il que les données gérées par ces différents appels du même sous-programme ne se "mélangent" pas?
Pour répondre à ces questions, il faut d'abord comprendre le modèle de mémoire utilisé dans l'exécution des sous- programmes.

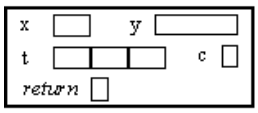
Modèle de mémoire
Chaque sous-programme (le programme principal aussi) utilise une zone de mémoire pour stocker ses arguments et ses variables locales et une place pour l'adresse de retours au programme appelant (le programme principal n'a pas besoin de cette adresse). Dans le cas d'une fonction, elle réserve de plus, une place pour le résultat retourné. L'allocation de cette zone de mémoire se fait au moment de l'appel du sousprogramme, dans une zone de mémoire

spéciale associé au programme, appelée la pile d'exécution. Les zones de mémoire de tous les sous-programmes appelés sont empilées suivant l'ordre de leur appel et dépilées un à un, dès que le sous-programme correspondant se termine. Par conséquent, une zone de mémoire d'un sous-programme n'existe physiquement que pendant l'exécution de celui ci.

Exemple 7 : Soit l'exemple de la fonction suivante:

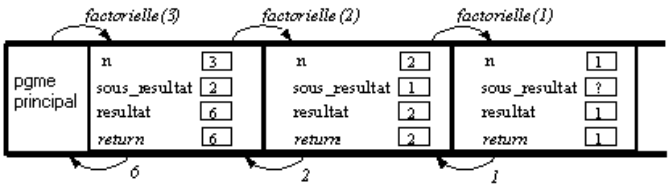
```
fonction exemple (donnée x: entier; donnée y: réel) : booléen
déclarations
        t: tableau[3] d' entier;
        c: caractère;
début
...
fin exemple;
```

On représentera la zône de mémoire occupée par cette fonction de la manière suivante:



Déroulement des appels récursifs

Dans le cas des programmes récursifs, la pile est remplie par des appels du même sousprogramme. L'enchaînement des appels et des calculs dans l'exemple du factoriel est illustré par le schéma ci-dessous :



Chaque instance de factoriel récupère le résultat de l'appel récursif qui suit dans la variable sous-résultat, ce qui lui permet de calculer résultat et de le retourner à son appelant.

2.2 Récursivité et itération

Par l'appel répété d'un même sous-programme, la récursivité permet de réaliser des traitements répétitifs. Or, jusqu'ici, pour réaliser des répétitions nous avons utilisé les boucles (itération). Suivant le type de problème, la solution s'exprime plus naturellement par la récursivité ou par itération. Souvent, il est aussi simple d'exprimer les deux types de solutions.

Exemple 8: Calcul du nombre d'occurrences n d'un caractère c dans une chaîne s

Solution récursive:

• Décomposition récursive:

1. Elément de récursivité : la chaîne s, dont la taille diminue. Un cas plus simple est la chaîne sans son premier caractère (nous la notons s1).

2. Soit n1 le nombre d'occurrences de c dans s1 (la solution du problème plus simple). Soit c1 le premier caractère de la chaîne initiale s. Alors, si c1 = c, alors le résultat est n = n1 + 1, sinon n = n1.

• Condition d'arrêt :
si s = "" (la chaîne vide) alors n = 0.
Le sous-programme associé (une fonction) est le suivant:

```
fonction nb_occurrences (donnée c: caractère; donnée s: chaine) : entier
déclarations
        sous_résultat : entier;
début
   si s = "" alors retourner 0 sinon sous_résultat ← nb_occurrences(c, reste(s));
        si premier(s) = c alors retourner 1 + sous_résultat
        sinon retourner sous_résultat
        finsi
   finsi
fin
```

Solution itérative:
La fonction donnée utilise une fonction notée taille(s): entier qui retourne le nombre de caractères dans une chaîne s.

```
fonction nb_occurrences (donnée c: caractère; donnée s: chaîne) : entier
déclarations
        acc, i : entiers
début
        accu ←− 0;
        pour i ←− 1 jusqu'à taille(s) faire
        si s(i)=c alors accu ←− accu + 1 finsi;
        finpour
        retourner accu;
fin
```

En principe tout programme récursif peut être écrit à l'aide d'actions itératives (boucles), sans récursivité. Inversement, chaque type d'action itérative (répéter, tantque, pour) peut être simulé par récursivité. Il s'agit en fait de deux manières différentes de programmation. Utiliser l'une ou l'autre est souvent une question de "goût" et de style de programmation. Cependant chacune peut s'avèrer mieux adaptée que l'autre à certaines classes de problèmes.

L'avantage essentiel de l'itération est l'efficacité (exécution rapide) . Un programme qui utilise des actions itératives décrit précisément chaque action à réaliser. Ce style de programmation est appelé impératif ou procédural. Le code compilé d'un tel programme est une image assez fidèle des actions décrites par le programme, traduites en code machine. Les choses sont différentes pour un programme récursif.

L'avantage de la récursivité est qu'elle se situe à un niveau d'abstraction supérieur par rapport à l'itération. Une solution récursive décrit comment aboutir à la solution à partir d'un cas plus simple. Ce style de programmation est appelé déclaratif. Au lieu de préciser chaque action à réaliser, on décrit ce qu'on veut obtenir, c'est ensuite, au système de réaliser les actions nécessaires pour obtenir le résultat demandé. Souvent la solution récursive d'un problème est plus intuitive que l'itérative et le programme à écrire est plus court et plus lisible. Néanmoins, elle correspond à un type de raisonnement particulier. De plus, le style déclaratif associé, produit souvent du code moins efficace (car entre le programme descriptif et le code compilé, il y a un espace d'interprétation rempli par le système, difficile à optimiser).
Dans les langages (comme C) qui offrent à la fois l'itération et la récursivité, nous préférons utiliser la récursivité surtout dans les situations où la solution itérative est difficile à obtenir telles que:
1. Manipulation des structures de données récursives (par exemple: les arbres).

2. Le raisonnement lui même est récursif.

3. Pointeurs

Quand on veut stocker/traiter des données de manière répétitive, on n'a souvent pas le choix que d'utiliser les tableaux. Ils sont très faciles utiliser et permettent d'accéder à tout élément, d'une manière directe (non séquentielle). Mais, le seul vrai problème posé par l'usage d'un tableau est sa taille. En effet, celui-ci doit être déclaré explicitement dans la partie déclarative (avant la compilation) et doit donc avoir une taille fixée à l'avance et celle-ci est souvent trop grande (sinon toujours). Il y a donc un gaspillage de mémoire non négligeable qui dans certains cas pose des problèmes.

Définition *Une variable du type pointeur est une variable appelée à contenir une adresse où est logée la donnée que nous désirons stocker/traiter.*

En effet, les pointeurs sont entre autres utilisés pour contourner le problème décrit ci-dessus à propos des tableaux. La déclaration du type pointeur se fait ainsi:

```
type_point = pointeur de type_element
```

Exemple 9: (Type pointeur)

```
t_e = pointeur d'entier
t_chaîne = pointeur de chaine
t_étudiant = pointeur de type t_personne
p : t_e
q :t_chaîne
tête :t_étudiant
```

La déclaration des variables p, q et tête indiquent au compilateur qu'il doit chercher et stocker dans:

p une adresse où il pourra stocker une donnée de type entier,
q une adresse où il pourra stocker une chaîne,
Tete une adresse où il pourra stocker une entité du type t_étudiant.

Pour accéder à ces parties réservées de la mémoire, nous utilisons par exemple les affectations suivantes:

```
[p] ←− 270
[q] ←− "bonjour"
[Tête].nom ←− "benali"
[Tête].prenom ←− "mohamed"
```