Chapitre G II
LES FONCTIONS
et
LES PROCEDURES

I. GENERALITES SUR LES FONCTIONS.

A. Notion de fonction.

En informatique, une fonction est un module (séquence d'instructions) dont la notion est semblable à celle des mathématiques.

Définition :

> Une fonction informatique est une fonction d'un ensemble E dans un ensemble F. **Les éléments de F doivent être toujours de type scalaire** avec une exception, le **type chaîne de caractères**. L**es variables mathématiques d'une fonction sont dits les paramètres d'entrée de la fonction informatique.**

La définition d'une fonction est donnée par l'algorithme permettant le calcul de son résultat.

B. L'intérêt des fonctions.

Quelques fonctions interviennent systématiquement et plusieurs fois dans le calcul de la solution de nombreux problèmes. La programmation de leur algorithme une fois et une seule présente donc un intérêt certain. Ceci justifie d'ailleurs la présence des fonctions telles que log(x), cos(x), sin(x), abs(x), exp(x) ... tant sur les machines à calculer que dans les langages de programmation. De telles fonctions sont dites prédéfinies et ne nécessitent aucune déclaration.

Dans des domaines particuliers, les statistiques, la finance, l'économie par exemple, l'utilisateur peut ne pas trouver, dans le langage, les fonctions prédéfinies qu'il désire utiliser. Les langages permettent alors la création par l'utilisateur de ses propres fonctions qu'il pourra ensuite utiliser à sa guise.

Enfin, pour un programme particulier, un ensemble d'instructions du programme peut avoir comme seul but le calcul d'une valeur, ce qui justifie la construction d'une fonction. L'écriture de ces instructions sous forme de fonction offre plusieurs avantages. D'une part, cela permet une meilleure lisibilité du programme; d'autre part, la fonction pouvant être écrite indépendamment du programme, sa mise au point et sa vérification sont alors plus faciles (premier commandement : diviser pour régner).

Il en résulte que l'élaboration d'un algorithme comporte une phase essentielle : la décomposition de l'ensemble des instructions en modules, pouvant eux-mêmes être décomposés en sous-modules.

C. Définition et appel d'une fonction

Comme cela a déjà été expliqué dans les paragraphes précédents, l'intérêt de l'écriture d'un tel algorithme est de pouvoir l'utiliser (en termes informatiques : ***l'appeler***) dans des environnements différents : le résultat est toujours calculé par la même séquence d'instructions mais dépend des valeurs initiales spécifiées. Ces valeurs initiales sont des informations qui entrent dans la fonction sous la forme de ***paramètres*.**

Une fonction apparaît alors comme un algorithme tel que ceux déjà étudiés, à la différence près qu'il est ***paramétré*** : une liste de paramètres permet de préciser les informations nécessaires au calcul du résultat.

Les principes de construction d'une fonction sont donc identiques à ceux d'un programme à l'en-tête près. Une déclaration de variables permettant de définir les variables de calcul, dites variables ***locales*** à la fonction, précède la séquence d'instructions à exécuter à chaque ***appel*** à cette fonction.

Le modèle algorithmique d'une fonction est donc le suivant :

fonction <nom de la fonction>(**liste des paramètres**)
Déclaration des paramètres
    <Liste des paramètres et leur type>
Déclaration des variables locales
    <Liste des variables locales à la fonction et leur type>
Début
<Instruction 1>
<Instruction 2>
<Instruction 3>
    .
<Instruction n>
Retourner <le résultat>

Les paramètres, permettant de construire le modèle de la fonction, sont appelés ***paramètres formels***.

Lors de l'exécution de ce sous programme (*appel de la fonction*), le nom de la fonction devra être suivi de la liste des ***paramètres effectifs*** ; chaque paramètre effectif étant associé à (remplaçant) un paramètre formel, cette correspondance étant définie par la position du paramètre dans la liste.

Les paramètres formels, comme les paramètres effectifs d'une fonction sont dits paramètres ***données*** (ou encore passés par valeur) car **la fonction ne peut pas modifier leurs valeurs** : un paramètre donnée retrouve toujours après l'appel la valeur qu'il possédait avant celui-ci. Ce point sera détaillé dans le paragraphe suivant.

Attention : une règle logique impose que l'on ne doit, au sein d'une fonction, ne jamais exécuter d'instruction de lecture et d'écriture : une information ne

doit jamais être lue au clavier, mais importée sous la forme d'un paramètre donnée; un résultat ne doit jamais être affiché à l'écran mais renvoyé comme résultat par la fonction.

II. PRINCIPE DE FONCTIONNEMENT DES FONCTIONS.

Le principe exposé ici est un compromis entre les principes théoriques de l'algorithmique et l'implantation des fonctions dans les langages.

A. Exemple introductif.

Prenons l'exemple d'une fonction qui calcule la plus grande valeur de deux nombres réels. Cet algorithme bien connu devient, sous la forme d'une fonction :

fonction maxi(x,y)

Déclaration des paramètres x: entrée de type réel y: entrée de type réel
Déclaration des variables locales t type réel
si x>y alors t:=x sinon t:=y finsi
Retourner la valeur de t réel

Exemple 1 : Le cadre de fonctionnement le plus simple possible.
Considérons maintenant le programme rudimentaire suivant, permettant d'afficher la plus grande des deux valeurs numériques 6 et 8:

```
Programme essai1
Début
 écrire(maxi(6,8))
fin
```

Description de l'exécution de ce programme :
Lors de l'appel à la fonction, on peut considérer que les variables x, y sont créées et reçoivent respectivement les valeurs 6 et 8. La variable t de calcul est créée à son tour. La séquence d'instructions de la fonction est alors exécutée dans cet environnement : x étant inférieur à y, le contenu de y, soit 8, est affecté à t. La valeur de t est alors retournée comme résultat.

Exemple 2 : Où l'on montre l'indépendance entre les noms des paramètres formels et des identificateurs des autres variables d'un programme.

Considérons le programme essai2 suivant:

Programme essai2

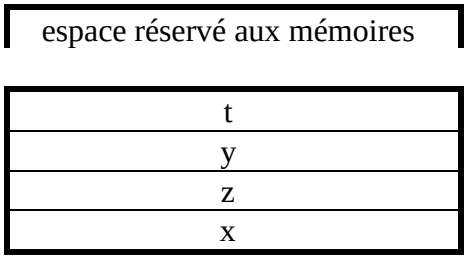
Déclarations des variables x,y,z,t de type réel y n'est pas utilisé
Lire(y)
Lire (x)
Lire (z)
t:=maxi(x,z)
Ecrire(t)
Fin

Si, lors de l'exécution, sont saisies au clavier les valeurs 3,6et 8, on peut prévoir que la valeur 8 sera affichée à l'écran. Mais ce qui nous intéresse ici est le principe de fonctionnement et non le résultat. En particulier, comme c'est le cas dans cet exemple, lorsque les mêmes identificateurs (x,y,t ici) existent dans la fonction et le programme. Comment sont-ils gérés? A-t-on ou non des conflits entre ces variables.

Pour répondre à ces questions, nous allons ici étudier le principe de fonctionnement de l'unité centrale qui sera simplifié au maximum (la réalité étant plus complexe mais de même nature). Soit donc la fonction maxi et le programme où ont été numérotées certaines instructions pour faciliter les explications.

fonction maxi(x,y) Déclaration des paramètres x :entrée réel y :entrée réel Déclaration des variables t :réel Début si x>=y alors t:=x sinon t:=y **(5)** finsi Retourner t réel	programme essai2 Déclaration des variables x,y,z,t :réel Début Lire(y) lire(x) (1) lire(z) (2) t:=maxi(x,z) **(3)** écrire (t) **(4)** fin

Après la compilation ou l'interprétation de ce programme en langage machine, l'unité centrale (UC) peut exécuter ces instructions après avoir créé en mémoire centrale les variables globales x,z,y,t. La mémoire a alors la forme suivante :



Simulons l'exécution du programme par l'ordinateur en supposant qu'ont été saisies au clavier les valeurs 3,6 et 8, qui sont alors stockées dans les mémoires y,x et z ci-dessus.

Appel de la fonction.

L'instruction (3) du programme, exécute la fonction maxi. L'UC (simplifiée) effectue alors les opérations suivantes dites d'appel de fonction :

1- Elle crée une mémoire de nom maxi de type réel.

2- Elle crée une mémoire adresse retour où elle stocke le numéro (3) de l'instruction du programme à exécutée, pour reprendre son exécution après le calcul de la fonction.

3- Elle crée les mémoires des paramètres de la fonction soit x,y distinctes de variables déjà existantes.

4- Elle initialise les mémoires des paramètres avec les valeurs des paramètres effectifs en respectant l'ordre dans lequel ils apparaissent dans la liste (le premier paramètre reçoit la valeur de la première variable et ainsi de suite le dernier paramètre reçoit la valeur de la dernière variable).

5- Elle se débranche totalement du programme et exécute la fonction maxi avec les variables nouvellement créées.

6- Elle crée les variables de la fonction dites variables locales (ici t variable évidemment différente de la variable globale du même nom).

La mémoire a alors la forme suivante :

espace réservé aux mémoires	Commentaire
t	variable locale de la fonction
y 8	paramètres d'entrée
x 6	de la fonction
adresse de retour (3)	adresse de retour dans le programme
maxi	mémoire pour récupérer le résultat de la fonction
t	variables dites globales
y……..3	du programme principal déjà
z 8	créées (ou créées lors de la
x 6	compilation)

On constate qu'il y a deux mémoires x, deux mémoires y, et deux mémoires t, mais les dernières mémoires, lors de l'exécution de la fonction ne sont plus accessibles par l'UC : **les variables x,y,t de la fonction cachent les variables globales x,y,t.** Elles seront de nouveau accessibles lorsque la fonction aura été exécutée (lors du retour au programme appelant).

L'UC exécute alors l'algorithme de la fonction. Juste après l'instruction (5) la mémoire t contient 8.

Retour au programme appelant.

Il ne reste plus qu'à effectuer le retour au programme principal ou appelant; L'UC pratique ainsi:

1- Elle affecte à la mémoire maxi la valeur de t, ici 8 :

espace réservé aux mémoires	commentaire
t 8	variable locale de la fonction
y 8	paramètres d'entrée
x 6	de la fonction
Adresse de retour (3)	adresse de retour dans le programme
maxi 8	mémoire pour récupérer le résultat de la fonction
t	variables dites globales
y 3	du programme principal déjà
z 8	créées (ou créées lors de la
x 6	compilation)

2- Elle détruit (libère) les mémoires correspondant aux variables de la fonction:

espace réservé aux mémoires	commentaire
Adresse de retour (3)	adresse de retour dans le programme
maxi 8	mémoire pour récupérer la valeur de la fonction
t	variables dites globales
y 3	du programme principal déjà
z 8	créées (ou créées lors de la
x 6	compilation pour les langages compilés)

3- Elle trouve dans adresse de retour le numéro de l'instruction du programme appelant , ici (3). Elle détruit (libère) cette mémoire et se branche à l'instruction (3) (t:=maxi(x,z);) On a la configuration suivante:

espace réservé aux mémoires	commentaire
Maxi 8	mémoire pour récupérer le résultat de la fonction
t	variables
y 3	du programme principal déjà
z 8	créées (ou créées lors de la
x 6	compilation)

4- Elle exécute cette instruction t:=maxi(x,z); en prenant la valeur de la mémoire maxi qu'elle met dans t et détruit la mémoire maxi. La configuration est :

espace réservé aux mémoires		commentaire
t	8	variables dites globales
y	3	du programme principal déjà
z	8	créées (ou créées lors de la
x	6	compilation)

B. Principes de base.

1. L'indépendance.

Dans l'exemple précédent, on constate que les mémoires intervenant dans la fonction sont parfaitement distinctes de celles du programme. En particulier, des instructions d'affectation sur les paramètres de la fonction ou sur les variables locales de cette fonction sont sans effet sur les variables globales. Les identificateurs des paramètres et des variables locales de la fonction peuvent être choisis librement. La fonction maxi ci-dessus pourra d'ailleurs être utilisée, c'est-à-dire appelée, dans tout futur programme, exactement comme une fonction prédéfinie. La parfaite compatibilité d'une fonction avec tout futur programme est une propriété de cette fonction et son résultat ne dépend que de l'environnement dans lequel elle est appelée.

2. Passage des paramètres par valeur.

L'appel d'une fonction dans un programme conduit simplement à initialiser les valeurs de chaque paramètre de la fonction et ne peut pas modifier la valeur des paramètres effectifs, d'où l'expression ***passage par valeur*****;** Toute instruction d'initialisation (par lecture ou affectation) d'un ou plusieurs paramètres apparaît alors comme **une erreur logique.**

3. Appel d'une fonction.

Un programme ne peut utiliser une fonction que par l'intermédiaire d'une instruction qui exploitera la valeur fournie par la fonction. La syntaxe de l'appel impose de donner la liste des paramètres ***effectifs*** permettant d'initialiser les paramètres ***formels*** (de construction) de la fonction.

4. Utilisation d'une variable globale dans une fonction.

Il est possible d'utiliser une variable, du programme appelant, dans une fonction. Il faut évidemment que l'identificateur de cette variable soit déclaré dans le programme et soit différent de tous les identificateurs des paramètres et des variables locales de la

fonction. Dans un tel cas, l'utilisation de cette fonction dans un programme futur implique des contraintes. La conception du programme doit tenir compte de l'utilisation, par la fonction, de cette variable et doit reprendre le même identificateur. A priori, cette possibilité peut apparaître comme la négation du concept de fonction et doit être évitée. Dans certains cas elle peut, pourtant, présenter une certaine pertinence.

Exemple d'utilisation pertinente:
On envisage l'écriture d'un programme utilisant à mainte reprises la fonction maxi et on souhaite obtenir l'affichage à l'écran du nombre d'appels de cette fonction à chaque exécution d'un programme.

```
            fonction maxbis(x,y)
Déclaration des paramètres
      x,y:réel

Déclarations des variables locales
      t:réel
début
si x>=y alors t:=x sinon t:=y
finsi
nbr_max:=nbr_max+1
retourner t en réel
```

```
            programme exemple1
Déclaration des variables du
programme
      nbr_max:entier
      autres variables....

début
  nbr_max:=0
  les autres instructions
   utilisant maxbis....
écrire nb_max
fin
```

Lorsque la fonction maxbis est appelée par le programme, son exécution s'effectue. Elle cherche la variable nbr_max pour l'incrémenter de 1, et ainsi calcule le nombre d'appels. Cette variable n'apparaissant dans l'environnement de la fonction construit au moment de l'appel, l'UC va la chercher dans l'environnement suivant, qui est celui du programme principal.

C. Généralisation : appel d'une fonction dans un sous programme.

Nous avons vu le mécanisme permettant l'appel d'une fonction par un programme principal. Ce mécanisme se généralise au cas où le programme appelle une fonction f1, qui appelle, elle même une fonction f2. Les paramètres effectifs de l'appel de f2 doivent alors être choisis parmi les paramètres formels de f1 ou les variables locales de f1.

D. Fonction définie dans une fonction.

Une fonction peut être définie à l'intérieur d'une autre fonction selon le schéma suivant:

bloc 1

Titre fonction fonc(x,y)
Déclaration des paramètres de fonc
x,y:réel
Déclaration des variables locales de fonc
u,t:réel
Déclaration des sous-fonctions

sous bloc 1
Titre fonc1(u,y)
Déclaration des paramètres de fonc1
u,y:entier
Déclaration des variables locales de fonc1a
t: entier;
z:réel;
début des instructions de fonc1
...
...
Retourner z pour la fonction fonc1

début des instructions de fonc
...
utilisation de fonc1 dans une instruction
...
Retourner (u) pour la fonction fonc

La fonction Fonc1 est dite emboîtée dans la fonction fonc. Les langages admettent généralement cette organisation. Selon les règles algorithmiques classiques, le sous bloc 1 correspondant à la fonction fonc1 ne peut être appelé que par les instructions du bloc 1. Aucun programme ne peut utiliser directement la sous-fonction fonc1.

Le Bloc 1 utilise les paramètres x, y et les variables u et t. Lors de l'appel de la sous-fonction fonc1, les variables u,t,y seront cachées par les paramètres u et y et la variable local t de fonc1. On admet que le paramètre x du bloc 1 reste accessible dans le sous bloc1 comme une variable globale.

Attention

Ces règles algorithmiques peuvent ne pas s'appliquer selon les logiciels programmables, notamment en MAPLE. Dans ce logiciel les variables locales du

bloc1 ne sont pas connues dans le sous bloc. On peut contourner cette implantation soit en la passant comme paramètre de la sous fonction, soit en la déclarant comme globale, elle restera alors dans l'environnement de la procédure.

III. L'implantation des fonctions dans un langage.

Ces implantations prennent des libertés par rapport aux principes algorithmiques ci-dessus. L'utilisateur doit consulter la documentation de son logiciel qui normalement doit ou devrait signaler les particularités du fonctionnement des fonctions.

Le langage Pascal a été élaboré pour une formation algorithmique des étudiants. Il respecte donc les grands principes algorithmiques exposés ici. Les logiciels professionnels adaptent et parfois contournent pour des raisons de rapidité et d'efficacité ces grands principes. L'étudiant doit ainsi acquérir les bases selon les règles de l'art conforme à tout enseignement, règles qu'un logiciel comme Maple ne respecte pas à la lettre.

Pour le néophyte, signalons ci-dessous les implantations particulières qu'on peut rencontrer dans divers logiciels programmables avec une mention particulière pour Maple.

Extension du type de F

Les fonctions ne peuvent pas retourner une donnée de type structuré à l'exception des chaînes de caractères. De nombreux langages de programmation contournent cette difficulté, notamment Maple, le langage C etc., en donnant à la fonction non la donnée structurée, mais son adresse mémoire qui est de type entier donc scalaire. Lorsqu'une fonction retourne l'adresse on dit qu'elle retourne un pointeur. Ainsi une fonction peut dans ce type de langage retourner tout type d'objet.

Passage par valeur

Un passage par valeur signifie toujours que les paramètres effectifs ne peuvent par être modifiés par l'algorithme de la fonction appelée. Maple respecte cette règle en interdisant toute affectation des paramètres formels de la fonction (sauf pour les array et tables). La traduction en Maple d'un algorithme modifiant un paramètre formel conduit à l'introduction d'une variable locale qu'on initialisera au paramètre formel, et qui pourra être modifiée.

Les paramètres formels de type structuré (les listes) ne sont pratiquement jamais passés par valeur pour économiser les mémoires. Une affectation du paramètre formel peut donc modifier les valeurs des paramètres effectifs de l'appel. En Maple, les tableaux (array) ou table qu'on verra, peuvent être modifiés.

Appel d'une fonction.

Normalement une fonction est appelée dans une instruction (a:=2*f(x) ou écrire(f(t)). Dans certain langage une fonction peut être appelé directement comme une commande (exemple de syntaxe =f(t) ou même directement f(t)) dans ce cas la

valeur retournée par la fonction n’est pas prise en compte. En Maple f(t); affiche la valeur retournée, mais f(t): calcule la valeur qui est non utilisée.

Les variables locales.

Lorsque les variables locales sont non déclarées explicitement, certains logiciels les déclarent implicitement comme des variables locales (Maple), d'autres les considèrent implicitement comme locale, ou globale s'il existe déjà une variable globale ayant le même identificateur.

Les variables globales.

En Maple, il faut déclarer explicitement les variables globales utilisées dans une fonction. En particulier une variable non déclarée globale ne peut pas être utilisée dans une sous-fonction.

IV. Les Fonctions récursives.

A. Définition des fonctions récursives.

Les fonctions récursives reposent sur le même principe que les fonctions mathématiques définies par une formule de récurrence. Par exemple on peut considérer la fonction mathématique définie par:

$$f(n) = \begin{cases} 1 \text{ si } n = 0 \\ n * f(n-1) \text{ si } n > 0 \end{cases}$$

Le lecteur aura reconnu la fonction factorielle.

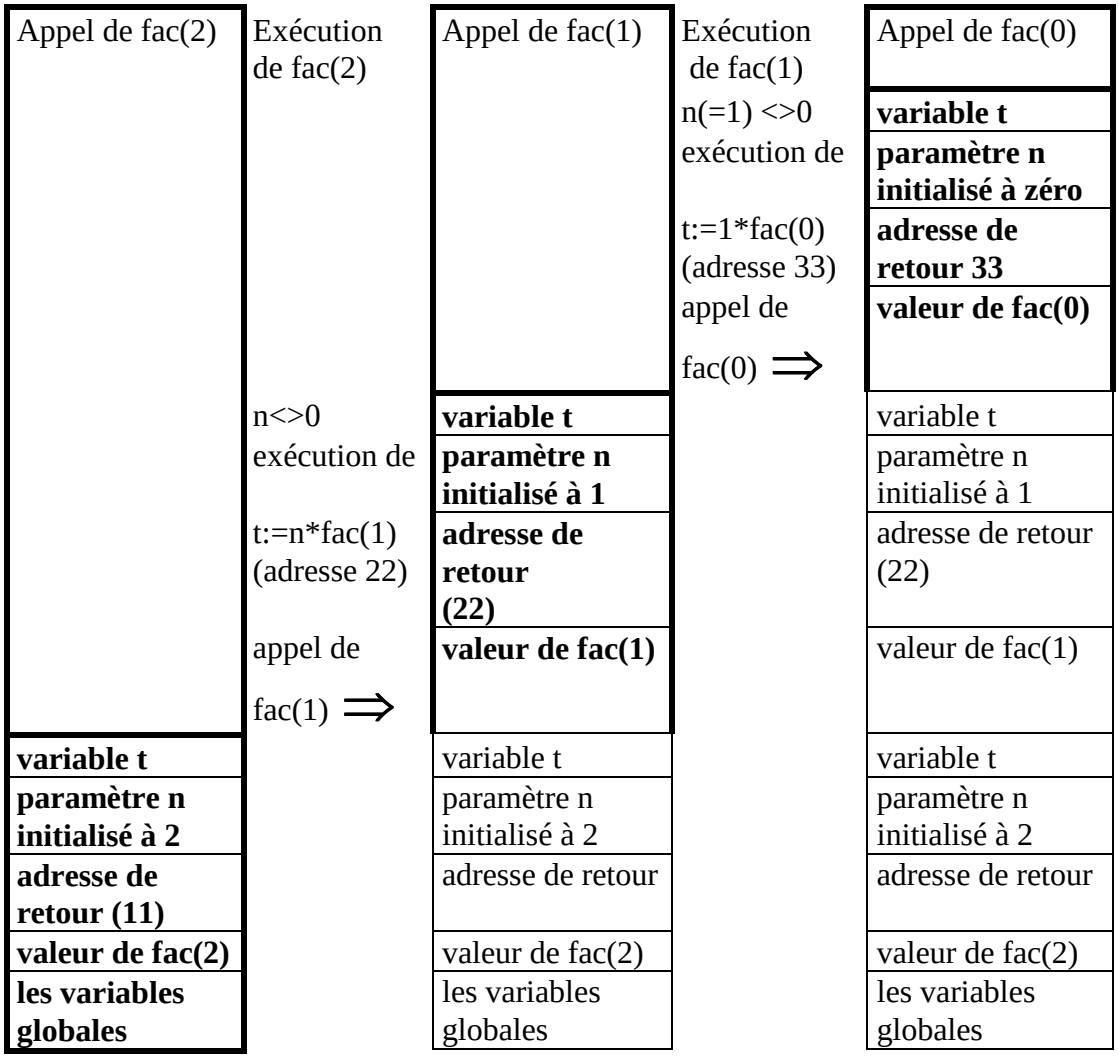
Les informaticiens ont implanté ce type de calcul sous le nom de fonction récursive. L'algorithme est défini comme pour une fonction classique par son algorithme. Soit pour l'exemple donné:

```
Fonction fac(n)
Déclaration des paramètres
        n:entier
début
   si n=0 alors t:=1
          sinon
          t:=n*fac(n-1)
  finsi
Retourner t
```

La structure de cet algorithme est typique de la récursivité. Elle comprend un branchement conditionnel permettant de fixer la valeur de la fonction pour la condition initiale ou triviale en fonction du ou des paramètres, et l'implantation de la formule de récurrence sinon. La valeur initiale est en faite une condition d'arrêt, comme nous le verrons dans le paragraphe suivant. Le terme de récursivité provient de l'instruction t:=n*f(n-1). En effet si n est diffèrent de zéro, alors pour calculer la fonction il faut exécuter cette instruction donc appeler de nouveau la même fonction. La fonction s'appelle elle même.

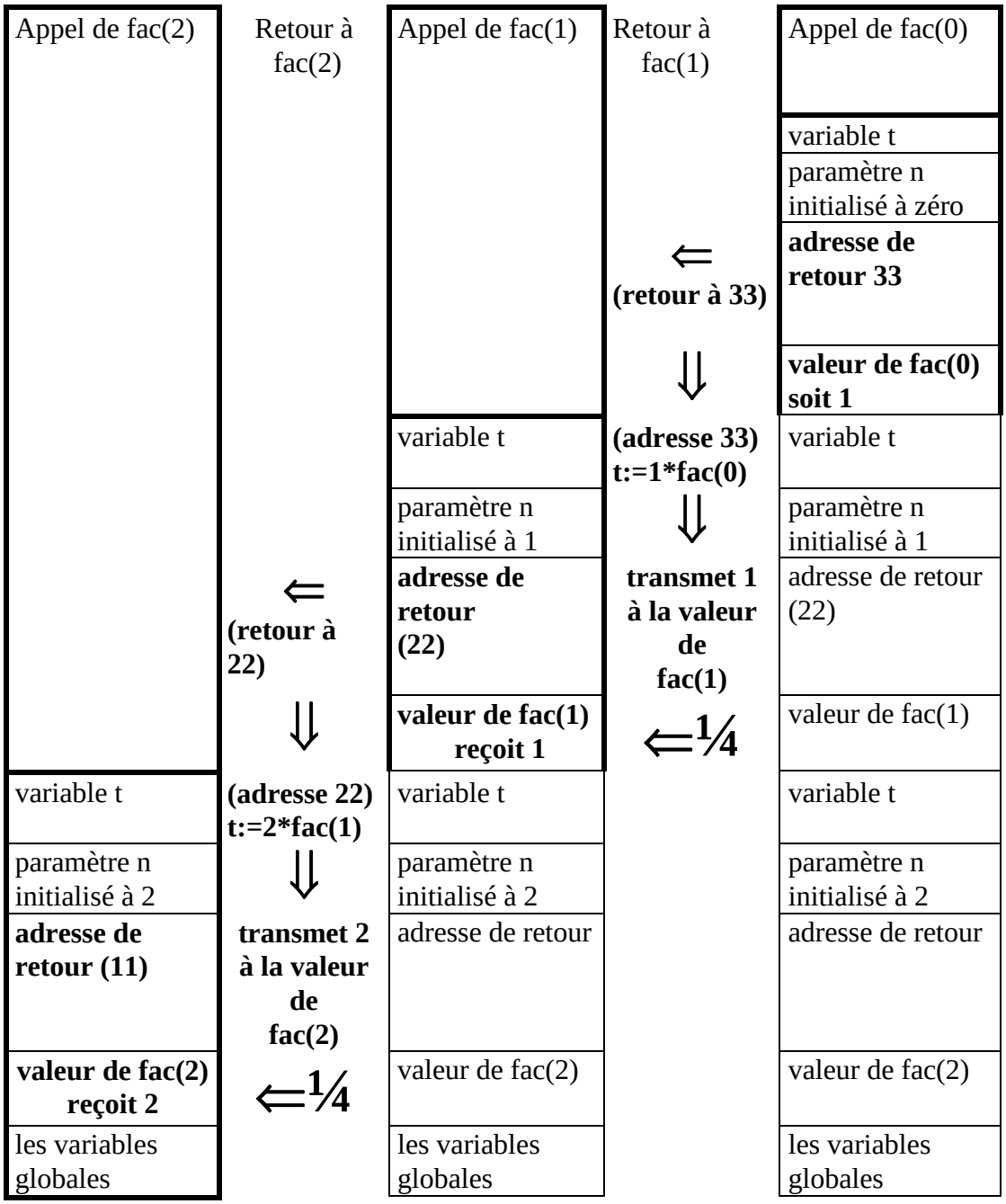
B. Fonctionnement des fonctions récursives.

Reprenons notre exemple, et imaginons que nous appelions fac(2) dans l'instruction d'adresse 11



Les adresses 11, 22, 33 sont arbitraires et correspondent aux adresses respectives de l'instruction fac(2), de l'instruction t:=2*fac(1), de l'instruction t:=1*fac(0).

On constate que fac(2) appelle la fonction fac(1) qui elle-même appelle fac(0). Le schéma ci-dessus représente ces différents appels. Il ne reste que l'exécution du dernier appel fac(0), qui va permettre de redescendre cette cascade, et fournir la valeur de fac(2). Lors de l'exécution de fac(0), n vaut zéro donc t=1 est la valeur retournée par cette fonction. On peut représenter ainsi le retour de chaque appel.



Il reste à effectuer le retour de fac(2). Le pointeur de commande recherche l'instruction d'adresse 11 et substitue à l'appel de fac(2) la valeur 2 calculée.

C. Puissance et limite des fonctions récursives.

Les fonctions récursives permettent une programmation facile à partir de la formule de récurrence entre F(n) et F(n-1) dite formule de récursivité, cette formule de récursivité est parfois plus facile à établir que les formules de récurrence dans les boucles itératives, mais surtout sa programmation est très courte et très rapide. On préfère souvent une implantation récursive à un algorithme itératif.

La programmation récursive est limitée par la capacité de la mémoire vive, car à chaque appel récursif, il y a une implantation des paramètres et variables locales de la fonction. De plus selon la formule de récursivité, le temps de calcul peut être très long, car certains calculs sont effectués inutilement plusieurs fois. La suite de Fibonnacci illustre parfaitement ce dernier point.

La suite de Fibonnacci est définie mathématiquement ainsi

Un=Un-1+Un-2
avec U0=1 et U1=1

On peut proposer pour calculer la valeur de Un l'algorithme récursif suivant

```
Fonction fibo(n)
Déclaration des paramètres
n: entier
Déclaration des variables
t:entier
début
si n<2 alors t:=1
        sinon t:=fibo(n-1)+fibo(n-2)
finsi
Retourner t
```

Si on veut calculer fibo(5), alors il faudra calculer fibo(4) et fibo(3) correspondant à l'instruction t:=fibo(n-1)+fibo(n-2). Or pour calculer fibo(4) il faudra calculer fibo(3) nécessitant le calcul de fibo(2) et fibo(1), et fibo(2), ces calculs seront refaits pour fibo(3) correspondant à fibo(n-2). On constate donc qu'un algorithme récursif peut recommencer les mêmes calculs.

Aujourd'hui tous les langages en principe acceptent la récursivité. De plus les informaticiens ont prévu des techniques évitant de recalculer plusieurs fois la fonction aux mêmes valeurs. En Maple, on dispose dans les fonctions de l'option remember, cette option crée une table de toutes les valeurs déjà calculées. Lorsqu'on appelle la fonction, on consulte toujours la table pour vérifier que cette valeur n'a pas déjà été calculée. Ainsi la vitesse de calcul est considérablement accélérée.

V. Les procédures.

Rappelons qu'une fonction algorithmique ne peut retourner qu'une valeur (scalaire ou une chaîne) même si certains langages dérogent à cette règle. Pour contourner cette restriction a été introduit le concept de pointeur, que l'on retrouve dans Maple (ou Foxpro) sous une forme simplifiée. Mais il existe un concept plus simple, la procédure, qui évite l'utilisation généralisée, plus complexe, des pointeurs. Ce concept de procédure est souvent incorporé, sous une forme plus ou moins rigoureuse, dans de nombreux logiciels programmables et notamment sous Maple (ou Foxpro). Les principes fondamentaux de cette notion seront d'abord présentés en langage

algorithmique. L’implantation sous Maple sera exposée dans le chapitre M IV (M pour Maple) consacré aux Procédures.

A. GENERALITES SUR LES PROCEDURES.

1. Notion de procédure.

Les notions de paramètres formels (de construction du modèle) et de paramètres effectifs (d’appel) sont les mêmes dans le cadre de l’étude des fonctions et des procédures.

Pour retourner plusieurs valeurs au programme appelant, le principe de construction du sous programme aurait pu être le suivant : à la liste des paramètres d'entrée on adjoint une liste de variables, dites de sortie ou paramètres de sortie, qui, lors du retour, communiquerait les valeurs à des variables du programme appelant, l'équivalent, dans le sens opposé, du passage des paramètres données entre le programme et la fonction.

La solution retenue fut différente. Les paramètres de sortie ou résultats sont rajoutés, et sont utilisés dans l'algorithme de la procédure. Bien qu'ayant des identificateurs différents, ils correspondent aux mémoires des paramètres effectifs du programme appelant. Pour ces paramètres de sorties le programme passe non pas les valeurs de ces mémoires mais les mémoires elles mêmes; c'est-à-dire les variables informatiques ou encore l'adresse de chaque mémoire correspondante d'où les expressions équivalentes de **passage par variable, passage par référence ou passage par adresse**.

2. Intérêt des procédures.

Le lecteur, ayant compris que la procédure permet le calcul de plusieurs résultats, conçoit l'intérêt de ce concept. Comme pour les fonctions, il est possible de définir ses propres procédures, et ainsi de se constituer sa bibliothèque de procédures (et fonctions) dans un domaine particulier.

La notion de procédure permet aussi (comme pour les fonctions) de structurer un long programme en procédures et fonctions écrites et testées séparément, le programme n'étant plus alors qu'un assemblage.

Enfin, signalons un autre intérêt non négligeable. Comme nous le verrons, les variables nécessaires à l'algorithme de la procédure sont, comme pour les fonctions, créées à l'appel et disparaissent lorsque l’on retourne au programme appelant ; ainsi, l'espace réservé aux mémoires dans la RAM peut être assez petit, alors que, sans fonction ni procédure, toutes les mémoires utilisées seraient des variables globales existant dès la compilation.

3. Définition et appel d'une procédure.

Comme pour les fonctions, l'intérêt de l'écriture d'un tel algorithme est de pouvoir l'utiliser (*l'appeler)* dans des environnements différents: Les résultats sont toujours calculés par la même séquence d'instructions mais dépendent des valeurs initiales spécifiées. Les valeurs initiales sont des informations qui rentrent dans la procédure sous la forme de ***paramètres de données***. Les résultats calculés sortent de la procédure sous la forme de ***paramètres résultats***.

Les principes de construction d'une procédure sont identiques à ceux d'un programme à l'en-tête près. La déclaration des paramètres permet de préciser la nature des paramètres (données, résultats). Une déclaration de variables permettant de définir les variables de calcul, dites variables locales à la procédure, précède la séquence d'instructions à exécuter à chaque appel de cette procédure.

Le modèle algorithmique d'une procédure est donc le suivant:

procédure <nom de la procédure> (liste des paramètres)
déclaration des paramètres:
    <liste des paramètres sorties<
    <liste des paramètres entrées>
déclaration des variables locales
Début
<instruction 1>
<instruction 2>
<instruction 3>

<instruction n>
Fin procédure

Les paramètres permettant de construire le modèle de la procédure sont appelés ***paramètres formels***. Lors de l'exécution de ce programme (*appel de la procédure*), le nom de la procédure devra être suivi de la liste des ***paramètres effectifs***. Chaque paramètre effectif est associé à (remplaçant) un paramètre formel, cette correspondance étant définie par la position du paramètre dans la liste.

Les paramètres effectifs (comme les paramètres formels correspondants) d'une procédure sont dits les ***paramètres données*** (ou encore passée par valeur), si **la procédure ne modifie pas leur valeur**. Les paramètres effectifs (comme les paramètres formels correspondants) sont dits paramètres résultats si **la procédure modifie leur valeur**.

B. PRINCIPE DE FONCTIONNEMENT DES PROCEDURES.

Comme pour les fonctions, les explications seront données à partir d'algorithmes. Là encore, le fonctionnement proposé n'a que de vagues liens avec la réalité, mais offre

un modèle simple de compréhension des procédures, ne conduisant à aucune erreur. De plus, il donne une approche de la gestion de la mémoire vive (RAM).

1. Les paramètres d'entrée et de sortie sont différents.

Considérons la procédure, ainsi formalisée, qui à partir de deux réels x, y, calcule leur somme s et leur produit p.

procédure sompro(s,p,x,y)
déclaration des paramètres s sortie de type réel p sortie de type réel x entrée de type réel y entrée de type réel
s:=x+y
p:=x*y
fin de procédure

programme essai
déclaration des variables principales x,y de type réel
sompro (x,y,5,2)
écrire x
écrire y
fin

Bien sûr, on peut prévoir que l'exécution du programme conduira à l'affichage de 7 et 10. D'après l'étude des fonctions, on peut déduire que, lors de l'appel de la procédure sompro, les mémoires des paramètres formels x, y sont créées et initialisées avec, respectivement, les valeurs 5 et 2, conformément au passage des paramètres par valeur. On remarque aussi que l'appel d'une procédure est différent de celui d'une fonction : une procédure étant en quelque sorte une instruction. Afin de comprendre le lien entre les paramètres formels de sortie (s,p) et les paramètres effectifs des résultats (x,y) du programme, envisageons le programme suivant et la même procédure, où les instructions ont été numérotées pour donner les explications.

procédure sompro(s,p,x,y)	
déclaration des paramètres	
s sortie réel	
p sortie réel	
x entrée réel	
y entrée réel	
début	
s:=x+y	(3)
p:=x*y	(4)
fin procédure	

programme essai2	
déclaration des variables	
x,y de type réel	
début	
sompro(y,x,5,2)	(1)
sompro(x,y,y,x)	(2)
écrire(x)	(5)
écrire(y)	
fin	

Dès la compilation du programme essai2 les variables globales du programme x, y sont créées dans la RAM. Nous admettrons que **chaque mémoire créée possède un numéro dit adresse de la mémoire** et nous supposerons que ces adresses commencent à 1.

espace réservé aux mémoires	adresse des mémoires	
y	n°2	variables globales
x	n° 1	du programme

a) Le premier appel de la procédure.

A l'exécution du programme essai2, la première instruction est l'appel de la procédure, sompro(y,x,5,2). L'UC (simplifiée) réalise les opérations suivantes :

1- Elle crée une mémoire dite adresse de retour qu'elle initialise avec (2), numéro de l'instruction suivante du programme.

2- Elle crée les mémoires des paramètres de sortie, ici s et p. Un mécanisme complexe conduit pratiquement à **rediriger, toute lecture ou écriture dans ces mémoires, vers les paramètres effectifs de l'appel**, ici les mémoires y,x du programme. Ce mécanisme sera symbolisé par les affiches ci-dessous, posées sur chaque mémoire s et p, et utilisant les adresses de la RAM.

Compte tenu de leur fonctionnement, les paramètres résultats sont dits passés *par adresse, par référence ou par variable.*

espace réservé aux mémoires	adresse des mémoires	
p pour toute utilisation de p prendre la mémoire ayant pour adresse le n° **1** ===>	n° 5	**paramètres de sortie** de la procédure
s pour toute utilisation de s prendre la mémoire ayant pour adresse le n° **2** ===>	n° 4	**passage par adresse ou variable**
adresse de retour (2)	n°3	
y	n°2	variables globales
x	n° 1	du programme

3 Elle crée les mémoires des paramètres d'entrée, ici x, y qu'elle initialise avec les valeurs des paramètres effectifs de l'appel, ici 5 et 2. Les variables x, y (d'adresse 1 et 2) du programme appelant sont alors **cachées** et non accessibles.

espace réservé aux mémoires	adresse des mémoires	
		variables locales
y 2	n°7	**paramètre d'entrée**
x 5	n°6	**passage par valeur**
p pour toute utilisation de p prendre la mémoire ayant pour adresse le n° **1** ===>	n° 5	**paramètres de sortie** de la procédure
s pour toute utilisation de s prendre la mémoire ayant pour adresse le n° **2** ===>	n° 4	**passage par adresse ou variable**
adresse de retour (2)	n°3	
y	n°2	**variables globales**
x	n° 1	du programme

4- **Elle crée les variables locales** nécessaires à l'algorithme, comme pour les fonctions. Dans notre exemple elles n'existent pas.

5- **Elle se débranche du programme appelant et exécute** l'algorithme de **la procédure** en tenant compte des affiches posées sur les paramètres de sorties.

Les identificateurs x, y correspondent aux paramètres de la procédure. Les variables globales x, y du programme existent mais ne sont plus accessibles. Pour les paramètres de sortie s, p toute affectation sera exécutée respectivement dans les mémoires d'adresse numéro 2 et numéro 1. L' utilisation des numéros des mémoires évite tout conflit entre les identificateurs.

b) L'exécution de la procédure.

L'exécution, par l'UC des instructions s:=x+y, et p:=x*y conduit à l'écriture des valeurs 7 dans s et 10 dans p, c'est-à-dire dans les mémoires numéro 2 et numéro 1. Ces affectations modifient de fait les variables globales x, y du programme qui sont théoriquement, ici, non accessibles, car **cachées** par les paramètres d'entrée x, y de la procédure. A la fin de la procédure les mémoires du programme appelant contiennent les valeurs souhaitées.

espace réservé aux mémoires	adresse des mémoires	
		variables locales
y 2	n°7	**paramètre d'entrée**
x 5	n°5	**passage par valeur**
p pour toute utilisation de p prendre la mémoire ayant pour adresse le n° **1** ===>	n° 5	**paramètres de sortie** de la procédure
s pour toute utilisation de s prendre la mémoire ayant pour adresse le n° **2** ===>	n° 4	**passage par adresse ou variable**
adresse de retour (2)	n°3	
y 7	n°2	**variables globales**
x 10	n° 1	du programme

Il suffit de reprendre l'exécution du programme appelant à l'instruction suivante.

c) Retour au programme appelant.

Le retour au programme essai2 s'effectue ainsi

1- L'UC détruit (libère) **les variables locales de la procédure**, qu'elle a créées à l'appel (ici pas de destruction car la procédure n'a pas de variables locales).
2- L'UC détruit, libère, **les mémoires de tous paramètres** qu'ils soient d'entrée ou de sortie, ici x, y, p et s.
3- L'UC retrouve l'adresse de l'instruction suivante, **dans** la mémoire **adresse de retour, qu'elle détruit** (libère) et **retourne** au programme essai2 en se branchant à l'instruction sompro(x,y,y,x). La RAM est ainsi :

espace réservé aux mémoires	adresse des mémoires	
y 7	n°2	variables globales
x 10	n° 1	du programme

d) Le deuxième appel de la procédure.

L'exécution de l'instruction suivante sompro(x,y,y,x) du programme déclenche le même mécanisme :
L'appel :
- **création de la mémoire adresse de retour** initialisée à (5).
- **créations des paramètres de sortie** s, p redirigés vers les mémoires d'adresse 1,2 pour toutes les utilisations possibles; ainsi s et p sont, de fait, automatiquement initialisés par les valeurs 10 et 7. conformément au schéma de la RAM.
- **créations des mémoires des paramètres d'entrée** x, y initialisées respectivement à 7, 10 conformément à la liste des paramètres effectifs.

espace réservé aux mémoires	adresse des mémoires	
		variables locales
y 10	n°7	**paramètre d'entrée**
x 7	n°6	**passage par valeur**
p pour toute utilisation de p prendre la mémoire ayant pour adresse le n° **2** ===>	n° 5	**paramètres de sortie** de la procédure
s pour toute utilisation de s prendre la mémoire ayant pour adresse le n° **1** ===>	n° 4	**passage par adresse ou variable**
adresse de retour (5)	n°3	
y 7	n°2	**variables globales**
x 10	n° 1	du programme

L'exécution :

- écriture dans s et p de 17 et 70, c'est-à-dire dans les mémoires n°1 et n°2; ainsi la variable globale x contient la valeur 17 et la variable globale y la valeur 70.

Le retour :

- **destruction des paramètres formels** de la procédure.
- **retour à l'instruction (5) du programme** et destruction de la mémoire adresse de retour.

On constate que les identificateurs s, p de la procédure modifient les variables x, y du programme principal bien que les identificateurs x, y n'apparaissent pas dans les instructions exécutées. Un tel phénomène est dit en informatique un **effet de bord.** Dans le deuxième appel sompro(x,y,y,x), l'utilisation des identificateurs x,y comme paramètres d'entrée, puis comme paramètres de sortie n'entraîne aucune erreur mais n'est pas sans conséquence; en effet les valeurs initiales de x,y sont remplacées par les valeurs calculées. Enfin, les variables s et p (paramètres de sortie) reliées aux mémoires n°i et n°j, prennent, compte tenu du mécanisme, les valeurs des paramètres effectifs leur correspondant. Ainsi, un paramètre résultat peut être simultanément utilisé comme paramètre d'entrée, principe étudié ci-dessous.

C. Les paramètres d'entrée-sortie.

1. La notion de paramètre entrée sortie.

Dans les procédures ainsi définies, le mécanisme utilisé initialise les paramètres de sortie dès l'appel, car toute utilisation d'un paramètre formel de sortie est redirigée vers une mémoire correspondant à un paramètre effectif de l'appel. Ainsi, les paramètres de sortie peuvent être, algorithmiquement, considérés comme des paramètres d'entrée et sortie; c'est-à-dire fournir une donnée nécessaire à l'algorithme (comme un paramètre d'entrée) et retourner une valeur, au programme appelant, qui est stockée à la place de la donnée fournie. L'exemple typique en est l'échange du contenu de deux variables informatiques.

2. Un exemple : l'échange.

```
procédure échange(a,b)
déclaration des paramètres
        a,b: entrée sortie réel
déclaration des variables
locales
        t:réel
début
        t:=a
        a:=b
        b:=t
fin procédure
```

```
programme essai3
déclaration des variables
        x,y :réel

début
        lire(x)
        lire(y)  (1)
        échange(x,y)
        écrire(x)        (2)
        écrire(y)
fin programme
```

Examinons rapidement le fonctionnement en observant la RAM. Supposons que l'utilisateur tape les valeurs 2 et 5. Juste avant l'appel de la procédure, la mémoire vive est dans l'état suivant :

espace réservé aux mémoires	adresse des mémoires	
y 5	n°2	variables globales
x 2	n° 1	du programme

A l'appel de échange (x,y), elle devient:

espace réservé aux mémoires	adresse des mémoires	
t	n°6	variables locales
b pour toute utilisation de b prendre la mémoire ayant pour adresse le n° **2** ===>	n° 5	**paramètres de sortie** de la procédure
a pour toute utilisation de a prendre la mémoire ayant pour adresse le n° **1** ===>	n° 4	**passage par adresse ou variable**
adresse de retour (2)	n°3	
y 5	n°2	**variables globales**
x 2	n° 1	du programme

L'exécution de échange conduit à affecter à t la valeur de a c'est à dire la valeur de la mémoire n°1 soit 2, puis à a, c'est-à-dire dans la mémoire n°1, la valeur 5 de b (celle de la mémoire n°2) et mettre 2, valeur de t, dans b, c'est-à-dire dans la mémoire n°2.
Au retour de la procédure on exécute les instructions écrire(x) et écrire(y) avec la configuration suivante:

espace réservé aux mémoires	adresse des mémoires	
y 2	n°2	variables globales
x 5	n° 1	du programme

D. Principe de base des procédures.

1. Les paramètres.

Comme pour les fonctions, les paramètres intervenant dans l'algorithme de la procédure, sont dits **paramètres formels**, et les variables (du programme) intervenant dans l'appel de la procédure, **les paramètres effectifs.** Algorithmiquement, il est indispensable de définir la nature de chaque paramètre : **paramètre d'entrée** (passage par valeur), **paramètres de sortie** (passage par variable ou adresse) ou **paramètres**

d'entrée et sortie (passage aussi par variable mais avec utilisation de la valeur initiale). Notons qu'**un paramètre d'entrée ne peut pas modifier des variables du programme appelant,** ils n'induisent aucun effet de bord.

Les notions d'entrée, de sortie, ou entrée sortie, ne concernent que les variables et absolument pas l'ensemble de l'environnement (écran, clavier). Ainsi écrire(x) est l'appel d'une procédure prédéfinie qui écrit la valeur de x, x est donc un paramètre d'entrée (effectif): L'écriture à l'écran ne doit pas modifier la valeur de la mémoire x. De même, lire(x) est une procédure prédéfinie dont x est un paramètre de sortie (effectif), et non d'entrée. En effet lire(x) n'utilise pas la valeur précédente contenue dans x, mais modifie la valeur de la mémoire x du programme, par celle que l'utilisateur lui a fournie.

2. Appel d'une procédure.

Une procédure est utilisée comme une instruction, avec la liste de ses paramètres effectifs. Pour les paramètres d'entrée on peut, comme pour les fonctions, utiliser des valeurs ou des identificateurs; Par contre pour les paramètres de sortie, ou d'entrée sortie, seuls des identificateurs de variables sont acceptés. Comme le montre l'exemple sompro(y,x,x,y) étudié ci-dessus, un même identificateur peut apparaître plusieurs fois; mais ce procédé est dangereux, et conduit à des résultats faux et imprévisibles si la règle suivante n'est pas respectée :

Un même identificateur ne doit pas être utilisée pour plusieurs paramètres effectifs de sortie.

Par contre un même identificateur peut être utilisé comme un paramètre (effectif) d'entrée et un paramètre (effectif) de sortie. Le paramètre de sortie peut encore être un paramètre d'entrée-sortie; Dans ce cas la valeur de l'identificateur avant l'appel est perdu au retour.

3. L'indépendance.

Les identificateurs des paramètres formels et des variables locales de la procédure sont parfaitement indépendants des identificateurs des variables et paramètres effectifs du programme appelant. En particulier, l'utilisation des adresses pour les paramètres de sorties évite tout conflit et ambiguïté par rapport aux identificateurs des mémoires du programme.

L'utilisation d'une variable globale dans une procédure est admise, comme pour les fonctions, avec les mêmes conséquences (perte de l'indépendance). Le principe est le même que pour les fonctions (voir chapitre fonction).

4. Généralisation de l'appel.

Le mécanisme d'appel, par un programme, d'une procédure, fonctionne lorsque la procédure est appelée dans une fonction ou une autre procédure. Les paramètres effectifs de l'appel sont alors les paramètres formels ou les variables locales du sous programme appelant. Ainsi, un paramètre de sortie de la dernière procédure renvoie à

l'adresse d'une mémoire de la première procédure; qui renverra, si elle est encore un paramètre de sortie, à l'adresse d'une mémoire du programme.
Le lecteur peut donc remplacer dans toutes les explications "programme appelant" par "bloc appelant" où bloc désigne soit le programme, soit une fonction, soit une procédure. La seule limitation aux appels successifs est la taille de l'espace réservé aux mémoires.

5. Fonction ou procédure en algorithmique?

Comme toute fonction peut être transformée en procédure il n'existe pas de règle absolue, mais une coutume bien établie. La fonction est utilisée s'il y a au moins un paramètre d'entrée et une et une seule sortie. En particulier on choisit une procédure s'il n'y a pas d'entrée, pas de sortie ou deux sorties et plus.

VI. Les fonctions procédurales et fonctions mixtes

Dans de nombreux langages, il n'existe pas nécessairement d'une part les procédures et d'autre part les fonctions. Un langage pédagogique comme le Pascal, utilise ces deux notions, mais les langages professionnels reposent essentiellement sur la notion de fonction. Ces langages sont dits fonctionnels.

A. Les fonctions procédurales

En principe les fonctions acceptent dans tous les langages que certains de ses paramètres soient passés par adresse, notamment pour des paramètres ayant une structure complexe comme les listes. Principalement cette possibilité évite un passage par valeur qui peut être long. Mais on peut, avec cette possibilité (passage par adresse des paramètres), forcer une fonction à retourner plusieurs valeurs. Si de plus elle ne retourne pas de valeur alors elle fonctionne comme une procédure. On dit que c'est une fonction procédurale. Certains logiciels (Maple, Foxpro) permettent même de les appeler comme des procédures. Ceci justifie l'utilisation de mot réservé " proc " dans la définition d'une fonction en Maple.

B. Les fonctions mixtes

On dit qu'une fonction est mixte quand elle retourne non seulement une valeur, mais d'autres valeurs par l'intermédiaire de ses paramètres passés par adresse. On pourrait employer le terme de procédure fonctionnelle. En Maple les fonctions sont des fonctions mixtes, qui sont procédurales si la valeur affectée à la fonction est nulle.