Récursion[1]

La récursion[2] est un concept fondamental en mathématiques et en informatique. La définition la plus simple que l'on puisse en donner consiste à dire qu'un programme **récursif** est un programme qui s'appelle lui-même, ou qu'une fonction est **récursive** si elle est définie en référence à elle-même. Mais cela ne suffit pas. Il faut en outre qu'un tel programme (ou une telle fonction) cesse de s'appeler lui-même si l'on ne veut pas qu'il boucle. Ce qui signifie que la suite des appels générés par cette récursion doit obligatoirement aboutir à un appel final. Une fonction ou un programme récursif doit donc contenir au moins une *condition de terminaison* permettant un retour sans nouvel appel.

Nous verrons d'abord quelques exemples où la récursion *n'est pas* du tout efficace, mais nous en profiterons pour faire ressortir le lien entre des relations de récurrence mathématique simples et des programmes récursifs simples. Nous étudierons ensuite un modèle élémentaire de programme récursif de type "diviser pour résoudre" *(divide-and-conquer)* et nous étudierons en détail comment supprimer la récursivité dans un programme pour obtenir un algorithme ***itératif*** simple (donc non récursif) à base de pile.

Comme nous le verrons, nombre d'algorithmes intéressants peuvent être exprimés simplement sous forme de programmes récursifs et l'on préfère souvent exprimer une méthode en termes récursifs. Mais il est aussi fréquent de trouver un algorithme tout autant intéressant caché dans une implantation (forcément) itérative. Dans ce chapitre, nous nous intéresserons aux techniques permettant de trouver de tels algorithmes.

RELATIONS DE RECURRENCE

Les définitions récursives de fonctions sont fréquentes en mathématiques ; le type le plus simple, portant sur des arguments entiers, est la *relation de récurrence.* La fonction la plus familière de ce type est sans doute la fonction ***factorielle,*** définie par :

$$0! = 1$$

et

$$N! = N.(N-1)! \quad \text{pour tout } N \geq 1$$

[1] texte établi d'après le livre suivant :
Robert Sedgewick - *Algorithmes en langage C* - InterÉditions 1991
et dans une moindre mesure le livre suivant :
Ellis Horowitz et Sartaj Sahni - *Data Structures in Pascal* - Computer Science Press 1987

[2] On emploie normalement le terme "récursion" pour désigner le concept et le terme "récursivité" pour décrire la propriété.

A cette définition correspond directement le programme récursif simple :

Factorielle

```
int Factorielle(int N)
{
if (N==0) return 1;
else return N*Factorielle (N-1);
}
```

Si ce programme illustre les caractéristiques élémentaires de tout programme récursif — il s'appelle lui-même (avec une valeur inférieure de l'argument) et il contient une condition de terminaison dans laquelle il calcule directement le résultat — on ne peut pas cacher qu'il n'est rien d'autre qu'une boucle "pour" enjolivée et ne démontre pas vraiment la puissance d'une récursion. Il est aussi important de se rappeler qu'il s'agit d'un *programme* et non d'une équation : par exemple, ni la relation de récurrence précédente ni le programme qui en découle ne "marchent" pour une valeur négative de *N,* mais les conséquences néfastes d'un oubli de cette contrainte sont plus visibles avec le programme qu'avec la relation. L'appel **Factorielle(-1)** se traduit par une boucle infinie : il s'agit, en fait, d'une erreur de programmation très fréquente que l'on retrouve, sous forme plus subtile, dans des programmes récursifs plus complexes.

Une deuxième relation de récurrence bien connue est celle qui définit la ***suite de Fibonacci*** :

$$F_N = F_{N-1} + F_{N-2} \text{ pour tout } N>2 \qquad \text{et} \qquad F_0 = F_1 = 1.$$

Ici encore, il existe un programme récursif simple associé à cette relation :

Fibonacci récursif

```
int Fibonacci(int N)
{
if (N<=1) return 1;
else return Fibonacci(N-1)+Fibonacci(N-2);
}
```

Voici en fait un exemple encore moins convaincant de la "puissance" du processus récursif ; on peut même dire qu'il *s'agit* d'un exemple convaincant de l'utilisation irréfléchie de la récursion et de l'inefficacité flagrante qui en découle. Le problème dans le cas présent est que les appels récursifs imposent des évaluations indépendantes de F_{N-1} et F_{N-2}, alors qu'il paraît naturel, en fait, d'utiliser F_{N-2} (et F_{N-3}) pour calculer F_{N-1}. Il est facile de dénombrer précisément les appels de la fonction Fibonacci précédente rencontrés dans l'évaluation de F_N : le nombre d'appels nécessaires au calcul de F_N est égal au nombre d'appels nécessaires au calcul de F_{N-1} plus celui relatif au calcul de F_{N-2}, en ignorant les cas **N** = 0 ou **N** = 1, où seul un appel est nécessaire. Cette description correspond exactement à la relation de récurrence définissant la suite de Fibonacci : le nombre d'appels de la fonction **Fibonacci** pour calculer F_N est exactement F_N. Il est bien connu que F_N est de l'ordre de φ^N, où φ est le "nombre d'or" :

$$\varphi = \frac{1+\sqrt{5}}{2} = 1,61803...$$

La cruelle vérité est que le programme précédent est un algorithme de *complexité exponentielle* (en temps) pour évaluer les éléments de la suite de Fibonacci!

En revanche, il est très facile de calculer $\mathbf{F_N}$ en temps linéaire, par le biais de l'algorithme très simple suivant :

Fibonacci itératif

```
int Fibonacci (int N)
{
int i,F1,F2,F;
if (N<2) return 1;
F1=F2=1;
for(i=1;i≤N;i++)
    {F=F1+F2;F2=F1;F1=F;}
return F;
}
```

En modifiant légérement ce programme, on peut en fait lui faire calculer (et conserver), dans pratiquement le même temps, les **N** premiers nombres de Fibonacci à condition de déclarer (ou d'allouer de la mémoire pour) un tableau de taille **N**. Remarquons cependant que, si le calcul peut se faire en temps linéaire, la suite elle-même a une croissance exponentielle. La valeur de **N** doit donc rester <u>petite</u>.

DIVISER-POUR-RESOUDRE

Beaucoup de programmes récursifs effectuent deux appels récursifs, chacun portant sur la moitié des données environ. Il s'agit du fameux modèle algorithmique "diviser-pour-résoudre" de construction algorithmique, que l'on utilise souvent pour réaliser d'importantes optimisations. Les programmes de ce type ne se ramènent en général pas à des boucles triviales, comme le programme Factorielle précédent, à cause justement du fait qu'ils contiennent deux appels récursifs. Ils n'entraînent pas non plus les répétitions de calcul du programme Fibonacci donné plus haut, car les données sont divisées sans recouvrement.

Prenons comme exemple la graduation par des marques régulières d'une règle de couturier : on désire trouver, pour chaque section, une marque à la moitié de section, une marque plus petite aux quarts de section, une encore plus petite aux huitièmes de chaque section, et ainsi de suite, comme le montre, d'une manière grossie, la figure *1*. Nous verrons qu'il existe plusieurs techniques pour accomplir ce travail ; cet exercice constitue un prototype de calculs simples de type diviser-pour-résoudre.

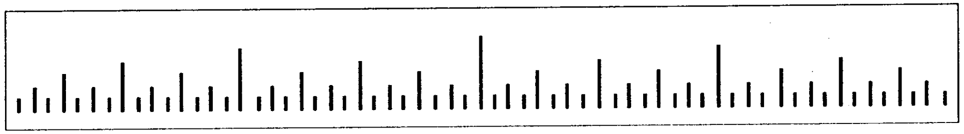

Figure 1. Divisions sur une règle

Pour obtenir une résolution de **1/2"**, on effectue un changement d'échelle et l'on transforme le problème en "placer une marque à chaque point situé entre 0 et 2", extrémités non comprises". On suppose l'existence d'une procédure Marquer(x,h) permettant de poser une marque de hauteur h à la position x. La marque médiane doit mesurer *n* unités, celles à la moitié de la partie gauche et de la partie droite, n-1 unités, etc. Le programme "diviser-pour-résoudre" suivant constitue un moyen direct d'atteindre ce but :

Graduation d'une règle

```
Règle(int g, int d, int h)
{
int m=(g+d)/2;
if (h>0)
    {
    Marquer(m,h);
    Règle(g,m,h-1);
    Règle(m,d,h-1);
    }
}
```

Ainsi, l'appel Règle(0,64,6) donne le résultat de la figure 1, après mise à échelle. La méthode sous-jacente est la suivante : pour inscrire les marques **d'un intervalle,** commencer par la plus longue du milieu. Ceci divise l'intervalle en deux parties **égales.** Placer les marques (plus petites) dans chaque moitié à l'aide de la même **procédure.**

On place une marque médiane et l'on appelle Règle pour la section gauche, on répète les mêmes actions pour la section gauche et ainsi de suite jusqu'à ce que la hauteur des marques soit nulle. Finalement, on revient des appels de Règle et l'on place les marques de la section droite d'une manière analogue.

La figure 2a (ci-dessous à gauche) montre le processus en détail et donne la liste des appels provoqués par l'appel initial Règle(0,8,3).

Remarquez la condition de terminaison `(h>0)` dans le programme. Comme h décroît strictement, cette condition provoque effectivement l'arrêt de la suite d'appels récursifs dès que la hauteur h des marques à faire est nulle.

Pour ce problème, l'ordre dans lequel les marques sont inscrites n'est pas particulièrement signifiant. On pourrait aussi bien inscrire les marques ***entre*** les deux appels récursifs : les

Règle(0,8,3)
    Marquer(4.3)
    Règle(0,4,2)
        Marquer(2,2)
        Règle(0,2,1)
            Marquer (1,1)
            Règle(0,1,0)
            Règle (1,2,0)
        Règle(2,4,1)
            Marquer(3,1)
            Règle(2,3,0)
            Règle(3,4,0)
    Règle(4,8,2)
        Marquer(6 .2)
        Règle(4,6,1)
            Marquer (5,1)
            Règle(4,5,0)
            Règle(5,6,0)
        Règle(6,8,1)
            Marquer(7,1)
            Règle(6,7,0)
	Règle(0,8,3)
	    Règle(0,4,2)
	        Règle(0,2,1)
	            Règle(0,1,0)
	            Marquer (1,1)
	            Règle (1,2,0)
	        Marquer(2,2)
	        Règle(2,4,1)
	            Règle(2,3,0)
	            Marquer(3,1)
	            Règle(3,4,0)
	    Marquer(4.3)
	    Règle(4,8,2)
	        Règle(4,6,1)
	            Règle(4,5,0)
	            Marquer (5,1)
	            Règle(5,6,0)
	        Marquer(6 .2)
	        Règle(6,8,1)
	            Règle(6,7,0)
	            Marquer(7,1)

Figure 2. Marquage des divisions sur la règle : (a) ordre préfixé (b) ordre infixé

marques de notre exemple seraient inscrites dans un ordre gauche-droite différent (figure 2b).

En fait ces deux façons de faire correspondent respectivement aux parcours **préfixé** et **infixé** de l'arbre de la figure 3 dans lequel chaque nœud correspond à une graduation de la règle.

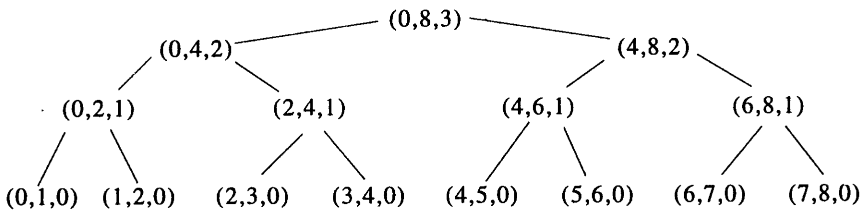

Figure 3. Arbre de **récursion** dans le marquage d'une **règle.**

Il est aussi simple d'écrire un algorithme itératif pour la même tâche. La méthode la plus directe consiste à inscrire les marques dans l'ordre en utilisant la boucle :

```
for (i - 1 ; i < N ; i++) Marquer (i, hauteur(i));
```

dans laquelle la fonction hauteur(i) retournerait le nombre de zéros consécutifs (augmenté de 1) en fin de représentation binaire de **i.** L'écriture de cette fonction en C est laissée en exercice.

Un autre algorithme itératif, ne correspondant à aucune implantation récursive, consiste à inscrire en premier les marques les plus courtes, puis les plus courtes parmi les marques non tracées et ainsi de suite. C'est cette méthode qu'adopte le très court programme suivant :

Graduation **itérative**

```
void Regle(int g, int d, int h)
{
int i,t,j,k;
for (i=h,j=d-g,k=j/2 ; i && j ; i--,j=k,k/=2)
 for (t=k+g;t<d;t+=j)
Marquer(t,i);
}
```

Il s'agit en fait du parcours par niveau des nœuds de l'arbre de la figure 3 (depuis la racine vers les feuilles). Le lecteur est invité à ecrire cette fonction, ainsi que sa version récursive, et à comparer les temps d'exécution pour des valeurs de h relativement faible (10, 20...).

Ce parcours correspond à une méthode générique de résolution algorithmique dans laquelle on résout un problème en traitant d'abord des sous-problèmes triviaux (faire les marques de hauteur h, puis celles de hauteur h-1,et ainsi de suite) et en combinant ces solutions pour résoudre des sous-problèmes un peu plus grands et ainsi de suite, jusqu'à résolution du problème entier. Cette approche pourrait porter le nom de "combiner-pour-résoudre". Bien que tout algorithme récursif admette une implantation itérative, il n'est pas toujours possible d'organiser les calculs de la façon précédente ; beaucoup de programmes récursifs dépendent de l'ordre spécifique dans lequel les sous-problèmes sont résolus.

PARCOURS RECURSIFS D'ARBRES

La façon la plus simple de parcourir un arbre est sans doute celle offerte par une implantation récursive. Par exemple, le programme suivant visite les nœuds d'un arbre binaire dans l'ordre infixé :

Parcours **infixé** récursif

```
ParcoursInfixé(struct noeud *t)
{
if (t != NULL)
    {
    ParcoursInfixé(t->g);
    Visiter(t);
    ParcoursInfixé(t->d);
    }
}
```

L'implantation reflète exactement la définition de l'ordre infixé :

> *"Si l'arbre n'est pas vide, parcourir le sous-arbre gauche, visiter la racine puis parcourir le sous-arbre droit".*

De manière évidente, on obtient le parcours en ordre préfixé en plaçant l'appel de *Visiter* avant les deux appels récursifs et l'ordre de parcours postfixé en le plaçant après.

Il est possible de demander au programme récursif précédent de mettre en évidence différentes propriétés des arbres en le modifiant légèrement et en changeant l'implantation de la procédure Visiter. Par exemple, le programme suivant calcule pour chaque nœud de l'arbre qu'il parcourt, les coordonnées **x** et **y** où doit être dessiné le nœud de l'arbre dans une représentation de cet arbre. Ce qui suppose que la structure relative à ces nœuds comprenne deux champs entiers supplémentaires pour **x** et **y**.

Dessin d'arbre

```
Visiter (struct noeud *t)
{
t -> x = x += dx;
t -> y = y;
}

ParcoursInfixé(struct noeud *t)
{
y += dy;
if (t != NULL)
    {
    ParcoursInfixé(t->g);
    Visiter(t);
    ParcoursInfixé(t->d);
    }
y -= dy;
}
```

Le programme utilise deux variables globales, **x** et **y**, toutes deux initialisées à zéro, et deux autres variables globales **dx** et **dy** qui sont respectivement les pas d'incréments en x et en y. La variable **x** donnera la position horizontale du nœud visité ; il lui est ajouté **dx** à chaque nœud visité dans l'ordre infixé. La variable **y** garde trace du niveau du nœud dans l'arbre. Chaque fois que **ParcoursInfixé** descend dans l'arbre, cette variable est incrémentée de **dy** ; chaque fois que la procédure remonte dans l'arbre, elle est décrémentée de cette même quantité **dy**.

D'une manière analogue, on pourrait écrire des programmes récursifs pour calculer la longueur de chemin d'un arbre, trouver d'autres méthodes pour le dessiner ou pour évaluer une expression qu'il représente, etc.

Quel est le rapport entre l'implantation récursive d'un parcours d'arbres et celle itérative du même parcours ? Sans aucun doute, ces deux programmes sont très voisins puisqu'ils engendrent, pour tout arbre choisi, la même série d'appels de Visiter. Nous allons maintenant étudier une méthode de suppression de la récursion, c'est à dire permettant d'obtenir une implantation itérative à partir d'un algorithme récursif.

Nous appliquerons ensuite cette méthode aux programmes de parcours d'arbre.

MÉTHODE DE SUPPRESSION DE LA RÉCURSION

Supprimer la récursion est une opération à laquelle se trouve confronté tout compilateur lorsqu'il doit traduire un programme récursif en langage machine. La méthode que nous allons présenter est à peu près celle qu'utilise en général un compilateur : Il remplace *tout* appel de procédure ou fonction par une série générique d'instructions dont l'effet est de :

> "sauvegarder les valeurs des variables locales et l'adresse de la prochaine instruction sur la pile, définir les valeurs des paramètres de la procédure et aller au début de celle-ci".

et il remplace de même *tout* retour de procédure ou de fonction par des instructions dont l'action est de :

> "dépiler l'adresse de retour et les valeurs des variables locales, mettre à jour les variables et aller à la bonne adresse de retour".

Nous allons voir ceci en détail ci-après sur un exemple.

Remarquons que la suppression de récursion peut parfois commencer par une étape préliminaire très efficace qui est la **suppression de récursion finale**. Nous la présenterons dans l'exemple (parcours d'arbre) du paragraphe suivant, car il n'est pas possible de l'utiliser dans l'exemple choisi ici (copie d'une liste).

Considérons une liste pour laquelle la structure de chaque nœud est la suivante :

tag (booléen)	**data** ou **lien** (suivant la valeur de tag)	**suivant** (pointeur vers nœud suivant)

Ce que l'on peut définir en C par une union imbriquée dans une structure :

```
struct noeudlist {
                    int tag;
                    union    {
                             char data[SZDATA];
                             struct noeudlist * lien;
                             }dl;
                    struct noeudlist * suivant;
                    } ;
typedef struct noeudlist * listpointer;
```

On supposera en outre que si tag est vrai, le champ dl doit être utilisé comme lien, et sinon comme donnée.

Considérons alors la fonction suivante permettant récursivement la copie d'une telle liste vers une nouvelle liste. Cette fonction a donc un argument de type listpointer et retourne un pointeur de même type.

fonction récursive de copie de listes

```
listpointer Copy(listpointer p)
{
listpointer q;
if (p==NULL) return p;
else
    {
    q=(listpointer)malloc(sizeof(struct noeudlist));
    if (q==NULL) {puts("Manque de mémoire"); exit(); }
    q->tag = p->tag;
    if (! p->tag)
        strcpy(q->dl.data,p->dl.data);
    else
        q->dl.lien=Copy(p->dl.lien);
    q->suivant=Copy(p->suivant);
    }
return q;
}
```

Nous allons maintenant reprendre cette fonction en précisant les remplacements à faire pour supprimer la récursion.

Fonction non récursive de copie de listes

```
listpointer Copy(listpointer p)
{
listpointer q;
debut:
if (p==NULL)      {
            vret=p;
            goto fin;
            }
else
    {
    q=(listpointer)malloc(sizeof(struct noeudlist));
    if (q==NULL) {puts("Manque de mémoire"); exit(); }
    q->tag = p>tag;
    if (! p->tag)
        strcpy(q->dl.data,p->dl.data);
    else
            {
            Empile(1);
            Empile(p);
            Empile(q);
            p=p->dl.lien;
            goto debut;
            1:
            q->dl.lien=vret;
            }
        Empile(2);
        Empile(p);
        Empile(q);
        p=p->suivant;
        goto debut;
        2:
        q->suivant=vret;
    }
vret=q;
fin:
if (PileVide() return vret;
else
    {q=Depile(); p=Depile();}
switch(Depile())
    {
    case 1: goto 1;
    case 2: goto 2;
    ...
    }
}
```

Tout 'return' intermédiaire est remplacé par :
- une affectation éventuelle de la valeur de retour
- un branchement vers la fin de la fonction

Chaque appel de la fonction est remplacé par :
- L'empilement d'une étiquette ou adresse de retour
- L'empilement des variables attribuées aux arguments et des variables locales
- L'affectation des arguments pour simuler le passage par valeur
- Un branchement vers le début de la fonction
- La définition d'une étiquette de retour spécifique à chaque remplacement
- L'utilisation éventuelle (dans une expression) de la valeur de retour

On remplace le 'return' final par :
- L'affectation éventuelle de la valeur de retour
- La création éventuelle d'une étiquette repérant la fin de la fonction.
- Si la pile est vide : un vrai retour
- Sinon : dépilement des variables locales et des variables associées aux arguments (en ordre inverse)
- Puis un branchement vers l'étiquette de retour, laquelle est obtenue par dépilement

Il faudra ensuite supprimer tous les 'goto' en les remplaçant par des 'while' ou des 'do-while'. Et éventuellement essayer de simplifier, par exemple en supprimant les empilements inutiles.

Il faut être conscient que cette transformation est en fait en général appliquée par tout compilateur qui se respecte, chaque appel de fonction étant effectivement remplacé par une suite d'empilements, et chaque retour par une suite de dépilements. La où l'on peut améliorer les choses, c'est ensuite, lorsqu'ayant supprimé tous les 'goto' en les remplaçant par des 'while' ou des 'do-while', on fera des simplifications en supprimant, par exemple, les empilements inutiles.

SUPPRESSION DE LA RECURSION : Un exemple de PARCOURS D'ARBRE

Nous partons d'une implantation récursive du parcours préfixé d'un arbre :

Parcours **préfixé** récursif

```
Parcours(struct noeud *t)
{
if (t != NULL)
    {
    Visiter(t);
    Parcours(t->g);
    Parcours(t->d);
    }
}
```

Suppression de la récursion finale

Avant d'appliquer la méthode telle qu'elle a été vue au paragraphe précèdent, nous allons supprimer le dernier appel récursif (le deuxième ici) et le remplacer par un simple 'goto' (précédé d'une affectation). Cette technique très connue, qui porte le nom de *suppression de récursion finale,* est implantée sur de nombreux compilateurs[3].

Elle ne peut être utilisée que si le dernier appel précède immédiatement le retour de la fonction. Cela signifie qu'il n'y a aucune instruction (même pas d'affectation[4]) entre ce dernier appel et le retour. C'est bien ce qu'on a dans le cas présent, puisque le deuxième appel de parcours n'est suivi d'aucune instruction. Lorsque le deuxième appel doit être effectué, c'est Parcours qui est invoqué (avec l'argument t -> d) ; à la fin de cet appel, l'appel *courant* de Parcours se termine aussi[5]. :

Suppression du **deuxième** appel récursif

```
Parcours(struct noeud *t)
{
test:     if (t == NULL) goto x ;
    Visiter(t);
    Parcours(t->g);
    t=t->d;
    goto test;
x:  return;
}
```

[3] En fait, elle correspond (en assembleur) au remplacement des deux instructions :
(**CALL** *adresse* / **RET**) par un simple **JMP** *adresse*.

[4] sauf dans le cas où cette affectation serait depuis la valeur retournée par cet appel vers la variable qui contient la valeur à retourner. Exemple :

```
q=Fonction(...);
return q;
```

[5] Un autre 'goto' est utilisé dans le test initial pour éviter une sortie de bloc, et pour simplifier les transformations suivantes.

Les programmes récursifs seraient bien moins viables et efficaces sur des systèmes qui ne disposeraient pas de la *suppression de récursion finale*, car c'est elle qui permet d'éviter les aberrations et les complications comme celles qui ont été signalées pour les fonctions *Factorielle* et *Fibonacci*.

Suppression du premier appel récursif

Pour supprimer le premier appel récursif nous allons maintenant appliquer la méthode présentée dans le paragraphe précédent :

Il n'y a qu'une seule variable locale ***t***, que l'on empile et l'on effectue un 'goto' au début de la procédure (étiquette ***test***). Puisqu'il n'y a qu'un appel récursif à supprimer, il n'y a qu'une seule adresse de retour, ***dte***, qui est fixée et il est donc inutile de l'empiler. A la fin de la procédure, on met à jour ***t***, à partir de la valeur du sommet de pile et on retourne à l'adresse ***dte***. Lorsque la pile est vide, on revient du premier appel de Parcours.

Suppression du **premier** appel récursif

```
Parcours(struct noeud *t)
{
test:     if (t == NULL) goto p ;
    Visiter(t);
    Empiler(t);
    t=t->g;
    goto test;
dte:      t=t->d;
    goto test;
p:  if (PileVide()) goto x
    t=Dépiler();
    goto dte;
x:  return;
}
```

Simplification du programme résultant (et suppression des 'goto')

La récursion a été supprimée, mais le programme obtenu est assez indigeste car il regorge de **'goto'**. Il est possible de supprimer ces derniers[6], d'une manière "mécanique", pour obtenir une série d'instructions plus structurée. En premier lieu, la suite d'instructions entre l'étiquette ***dte*** et le deuxième **'goto *test*'**, étant entourée de **'goto'**, peut être déplacée, ce qui élimine l'étiquette ***dte*** et son 'goto' associé. De plus, on remarque que l'on assigne la valeur ***t->d*** à ***t*** après dépilement : il serait plus judicieux d'empiler cette valeur. Enfin, les instructions entre l'étiquette ***test*** et le premier **'goto *test*'** ne représentent qu'une simple boucle '**tant que**', ce qui donne :

[6] remarquons cependant que l'utilisation de 'goto' dans les tests et les boucles est obligée au niveau du langage machine. Le travail d'un compilateur est donc pratiquement terminé à ce point de la suppression de récursion. Un bon compilateur peut malgré tout optimiser dans une certaine mesure. En ce qui nous concerne, nous allons supprimer ces 'goto' afin de rendre le programme plus lisible car mieux structuré et nous allons aussi simplifier.

Suppression des 3 **premiers** 'goto'

```
Parcours (struct noeud 't>
{
test: while(t!=NULL)
    {
    Visiter (t)
    Empiler(t->d);
    t-t->g;
    }
if ( PileVide () ) goto x;

t = Dépiler();
goto test;
x : return;
}
```

La seconde boucle d'instructions peut aussi être traduite par une boucle "tant que" moyennant un empilement supplémentaire (de l'argument initial **t** à l'entrée de Parcours), ce qui donne un programme sans 'goto' :

Suppression des derniers 'goto'

```
Parcours (struct noeud 't)
{
    Empiler(t);
    while (! PileVide ())
    {
    t = Dépiler();
    while(t!=NULL)
           {
           Visiter(t);
           Empiler(t->d);
            t=t->g;
           }
    }
}
```

Ceci est la méthode de parcours préfixé itérative "standard". Il est instructif d'oublier quelques instants comment on l'a trouvée et de tenter de se convaincre directement que ce programme effectue réellement un parcours préfixé.

A vrai **dire,** la structure de boucles imbriquées de ce programme peut être simplifiée[7], moyennant l'ajout de quelques empilements supplémentaires :

- Suppression de l'imbrication -

```
Parcours (struct noeud 't)
{
    Empiler(t);
    while (! PileVide ())
    {
    t = Dépiler();
           if(t!=NULL)
                  {
                  Visiter(t);
                  Empiler(t->d);
                  Empiler(t->g);
                  }
           }
    }
```

Ce programme commence à ressembler de manière étonnante à l'algorithme récursif de parcours préfixé original, mais les deux sont pourtant *très* différents. Une différence de taille est que ce dernier programme peut être exécuté dans pratiquement n'importe quel environnement de programmation, alors que l'implantation récursive ne peut l'être que dans un environnement permettant la récursion. Même dans un tel environnement, la version itérative a toutes chances d'être bien plus efficace.

Notons enfin que ce programme empile des sous-arbres vides, en conséquence de la décision prise dans l'implantation récursive originale de vérifier que le sous-arbre n'est pas vide avant toute autre action. On pourrait ne lancer l'appel récursif que sur les sous-arbres non vides en testant **t->g** et **t->d** Cette remarque permet d'obtenir, à partir de la version précédente, l'algorithme à base de pile pour parcours préfixé déjà vu :

Parcours préfixé : implantation itérative avec pile

```
ParcoursPréfixé(struct noeud *t)
{
 Empiler(t);
 while ( !PileVide())
   {
    t= Dépiler();Visiter (t);
    if(t->d!=NULL) Empiler(t->d);
    if(t->g!=NULL) Empiler(t->g);
   }
}
```

Le cas des parcours infixé et postfixé

La même méthode peut être appliquée sans trop de problèmes au programme de parcours **infixé** d'un arbre. En ce qui concerne le parcours **postfixé**, la suppression de récursion finale ne peut pas s'appliquer. Il y a donc deux appels à supprimer en utilisant la méthode du § précédent, et la simplification du programme obtenu s'avère plus délicate.

Ces deux suppressions de récursion sont laissées en exercices au lecteur.

CONCLUSION

Tout algorithme récursif peut être ainsi "dérécursifié" . En fait, c'est une des tâches essentielles d'un compilateur. Bien que la technique de suppression de la récursion que nous venons de présenter soit assez complexe, elle aboutit souvent à une implantation itérative efficace et conduit à une meilleure compréhension de la nature des opérations effectuées.

[7] Cette partie est sans doute ce qui est le plus difficilement automatisable.