ITÉRATION ET RÉCURSIVITÉ

Jacques ARSAC

« Les définitions implicites sont aux méthodes constructives ce qu'est le vol au travail honnête. »

Bertrand Russell

1. UN DÉBAT PASSIONNÉ.

Les informaticiens sont gens très savants, formés à une rationalité qu'ils ont appris à développer par l'emploi de leurs ordinateurs. Mais, dès qu'il s'agit de langage de programmation, leurs passions l'emporte sur leur raison. La discussion cesse de devenir scientifique. Chacun a ses préférences, qu'il défend de façon plus ou moins convaincante. Et c'est sans doute naturel et bon. Le langage est l'expression de la pensée : chacun prend celui avec lequel il s'exprime le plus facilement. Il y a des gens pour soutenir que la langue philosophique par excellence est l'allemand, et qu'un anglais ne peut philosopher à cause de sa langue maternelle. Quelle sottise! On aime sa langue maternelle, parce qu'elle est notre support culturel, qu'elle nous touche au plus près. Pour beaucoup de programmeurs, le premier langage de programmation qu'ils ont appris est leur langue maternelle, qui gardera leur préférence quoi qu'il arrive. Il serait vain de vouloir les en détourner, et ce serait prendre le risque d'une « révolution culturelle » inutile.

Dans ce débat, deux grandes familles de langages s'opposent : les impératifs, dont l'archétype est BASIC, et les récursifs, à la LISP. Plus que des langages, ce sont des formes de pensée : il y a ceux qui ont besoin de savoir comment on calcule un certain nombre, et ceux pour qui il suffit qu'il soit défini de manière précise. Nous voudrions montrer comment ceci est la résurgence en informatique d'un débat qui a agité la classe mathématique pendant près d'un siècle. Puis, sur l'exemple très simple de l'inversion d'une chaîne de caractères, nous montrerons des aspects souvent peu connus de l'itération et de la récursivité, rendant bien difficile de dire qui peut sortir vainqueur de la compétition.

2. L'IMPLICITE ET LE CONSTRUCTIF

Le raisonnement mathématique par récurrence, ou par induction, est assez ancien. Mais il fallut attendre le mathématicien Grassman, au milieu du XIX^{e} siècle, pour qu'il soit utilisé pour la définition d'une fonction. Cette utilisation n'alla

pas sans soulever de nombreuses protestations. Gotlob Frege reprocha à Grassman de ne pas avoir démontré que la fonction qu'il avait définie existait effectivement, et qu'elle était unique.

f(m,0) = m **f(m,successeur(n)) = successeur(f(m,n))**

Qu'est-ce qui prouve que ceci définit de façon unique l'addition m + n ? Frege reprochait en outre à une telle définition que le mot à définir figure dans la définition. En conséquence, on ne peut plus remplacer le mot à définir par sa définition, parce qu'il y figure. Or, selon « De l'esprit de géométrie ou de l'art de persuader » de Blaise Pascal, un des fondements de la science est « l'imposition de nom » : on fabrique un concept nouveau exprimé par des phrases, on prend un nom que l'on vide de sa signification antérieure, on l'attache à ce concept ; après quoi, on peut à chacune de ses occurrences le remplacer par sa définition. L'objection de Frege n'est pas périmée. Pour illustrer la récursivité à des professeurs de lycées, je définis mes ancêtres comme mon père, ma mère ou les ancêtres de mon père ou de ma mère. « Vous ne pouvez dire cela, le mot à définir figure dans la définition ; nous interdisons à nos élèves de le faire ». Etonnez-vous alors de la méfiance de tant de programmeurs face à LISP.

Prouver l'existence d'une fonction définie par récurrence, ou « implicitement », ne peut se faire par la seule arithmétique, c'est un problème de logique que résolut Dedekind à la fin du siècle dernier. Même faite, la démonstration ne mit pas fin aux controverses. Kronecker voulait qu'on ne parle d'ensemble que quand on peut **déterminer effectivement** de tout élément s'il est ou non dans l'ensemble. Par « déterminer effectivement » , il faut entendre que l'on possède une méthode de calcul permettant de répondre à la question de l'appartenance. Seuls existent les nombres pour lesquels on a une loi de construction. Dedekind refusait ces restrictions : il imagina la méthode des coupures pour définir les nombres réels. Ainsi les mathématiciens étaient divisés sur cette question.

Richard crut faire pencher la balance du côté des constructivistes par son « antinomie », montrant que le point de vue de Dedekind aboutissait à une contradiction. « On considère l'ensemble de tous les nombres que l'on peut définir en moins de 10 lignes. On les ordonne en classant leurs définitions en ordre lexicographique. Puis on fabrique un nombre dont le chiffre de rang i est le chiffre de rang i du nombre de rang i augmenté de 1 s'il est inférieur à 9, 0 sinon ». Ce nouveau nombre est défini en moins de 10 lignes, et pourtant il n'est pas dans l'ensemble. Les plus grands mathématiciens du début de ce siècle ont essayé de lever la contradiction. Pour Borel, le mal était dans la définition des nombres par des mots ordinaires, le langage naturel étant insuffisamment précis.

Ces recherches firent progresser profondément les mathématiques et la logique. Il fallait donner un sens précis aux différentes notions qui avaient été agitées dans la controverse entre les tenants de l'implicite et ceux de l'effectif. Dans les années 20, Skolem chercha à construire toute l'arithmétique par les seules voies implicites en proposant la notion de fonctions récursives primitives. Pour en montrer les limites, Ackermann imagina sa fameuse fonction comme exemple

d'une définition implicite qui n'était pas récursive primitive. La mathématicienne hongroise Rosza Péter développa l'étude des fonctions récursives. En introduisant l'opérateur de « minimisation » (qui définit le plus petit nombre ayant une certaine propriété), on obtint les fonctions récursives générales qui sont toutes les fonctions calculables. Ainsi, le calculable s'identifiait au récursif, assurant la totale puissance expressive de celui-ci : tout ce qui peut être calculé peut être dit récursivement.

La notion de fonction avait été clairement définie en théorie des ensembles par la notion d'application. Parce qu'il fallut recourir à la logique pour démontrer la validité des définitions implicites, il fallut faire pénétrer la notion de fonction en logique. Curry créa la logique combinatoire, s'éloignant assez des méthodes usuelles de définition de fonctions. Church chercha à rester au plus près du langage habituel des mathématiques, mais se montrait peu satisfait par le laxisme verbal. La formule

$$(a+b)*(a-b) = a^2 - b^2$$

doit s'accompagner de commentaires tels que : « quels que soient les nombres a et b ». On ne peut parler facilement de la fonction qui à x associe $x^2 - 3x + 1$.

$$x^2 -3x +1 \geq 1 \quad \text{et} \quad x^2 -3x + 1 \geq -x$$

n'ont pas le même sens : l'une définit un intervalle de valeurs de x sur lequel l'inégalité est satisfaite, l'autre peut être comprise comme une relation entre deux fonctions. Church imagine donc une notation, le « lambda calcul » , qui lui permet de tenir un discours précis et cohérent sur les fonctions. Son but était de reconstruire la logique mathématique, présentée depuis Frege et Russell comme fondement des mathématiques. Il imagina donc une représentation des entiers au moyen du lambda-calcul. A partir de là, il put définir les calculs au moyen de ses notations. Il émit la thèse que « l'effectivement calculable » s'identifie au « lamda - définissable ».

Dans le même temps, Turing cherchait à donner un sens à la notion de « calcul effectif ». S'appuyant sur une analyse simpliste de la psychologie d'une personne en train de faire un calcul, il inventa sa fameuse machine, oeuvre de génie telle qu'aucune variante imaginée plus tard n'a pu être montrée plus puissante. Elle est un moyen de faire des calculs, s'appuyant sur un mécanisme de commande qui lui donne des instructions en fonction de l'état dans lequel elle se trouve et de la donnée qu'elle lit. Elle répète une boucle de programme jusqu'à atteindre l'état final.

D'une certaine façon, l'informatique est issue de ces spéculations. Von Neumann joua un rôle important dans ces controverses, il fut l'élève de Turing. Les ordinateurs, comme les machines de Turing, obéissent à des instructions, et la première programmation, qui va du langage de la machine à l'assembleur puis à Fortran, est dans la tradition des constructivistes : pour calculer, je décris le processus calculatoire comme suite d'opérations.

A la fin des années 50, John MacCarthy, mais aussi le groupe dirigé par Van Winjgarden qui produisit le langage ALGOL, imaginent que les définitions

implicites ou récursives peuvent être utilisées pour commander un calcul par ordinateur, en proposant un algorithme pour calculer effectivement la valeur d'une fonction ainsi définie. En Algol 60, la récursivité est un additif aux langages impératifs. MacCarthy au contraire reprend l'héritage de Church et fonde LISP sur le lambda-calcul et la récursivité comme seule moyen d'expression.

Les praticiens, ignorant le long débat mathématique qui aboutit à ces deux courants de pensée, réagissent suivant leurs goûts ou leurs blocages psychologiques. Beaucoup, habitués dès leur plus jeune âge à refuser les définitions circulaires ont du mal à penser récursivement et ne sont à l'aise que dans la programmation impérative. Il est de fait que peu d'étudiants scientifiques manient avec aisance le raisonnement par récurrence. C'est une difficulté qui a ses racines dans notre enseignement primaire. La pratique de LOGO, ce cousin de LISP, peut-elle y changer quelque chose ? D'autres, dont je suis, gardent le sentiment de Bertrand Russell : « les définitions implicites sont aux méthodes constructives ce qu'est le vol au travail honnête ». Pour eux, restés attachés à la tradition constructiviste, les définitions implicites sont d'une grande élégance, mais ne disent pas comment se fait effectivement le calcul.

L'opposition paraît ainsi entre la définition implicite d'une fonction, généralement simple et rapide à construire pour qui est habitué à cette façon de penser, et sa définition explicite par un algorithme de calcul, plus complexe à élaborer, mais répondant à la question du « comment se fait le calcul », ce que ne fait pas la définition implicite. Par la théorie, les deux voies ont la même puissance d'expression, elles calculent les mêmes objets. Mais c'est la théorie. Suivant les cas, la simplicité change de camp, à moins que ce ne soit l'efficacité. Nous allons illustrer ceci sur un exemple.

3. L'INVERSION DE CHAINE DE CARACTÈRES

3.1. Le problème

Soit une chaîne de caractères c, de longueur n quelconque (nombre de caractères non fixé à l'avance). Il s'agit d'en former l'inverse, ou image miroir : la chaîne résultat est écrite avec les mêmes caractères que c, mais dans l'ordre inverse : l'inverse de 'NOEL' est 'LEON'. Si la chaîne de caractères est représentée dans le langage de programmation comme un tableau de caractères, le problème a une solution fort simple qui consiste a échanger entre eux les caractères de rangs i et n+1-i, pour i allant de 1 à n/2. Mais les langages qui traitent ainsi les chaînes présentent par ailleurs de nombreux inconvénients, notamment l'absence de primitives sur ces chaînes facilitant leur manipulation comme éléments d'un langage. Une chaîne doit être de longueur non fixée d'avance (quelle est la longueur d'une phrase, d'un paragraphe ?), on doit pouvoir y chercher facilement l'occurrence d'un caractère ou d'un mot. Dans les langages orientés vers ces structures de données (APL, SNOBOL, LISP, LSE) la chaîne est un objet simple

dont on ne peut modifier un caractère sans modifier toute la chaîne. Nous nous placerons dans ce cas là.

3.2. Première solution itérative

Les élèves de lycée formés à l'itération n'auront pas grand mal à imaginer une solution itérative : il faut réécrire les caractères de la chaîne dans l'ordre inverse. Si je parcours la chaîne directe de la gauche vers la droite, je construirai la chaîne inverse de la droite vers la gauche, en écrivant d'abord son dernier caractère, puis en mettant devant l'avant-dernier, puis devant l'avant - avant - dernier, etc. Au début, il n'y a rien dans la chaîne inverse v ; elle est donc initialisée par la chaîne vide, notée « '' ». « lgr (c) » désigne la longueur de c (nombre de caractères), « sch (c,i,1) » est la sous-chaîne extraite de c à partir de la position i et de longueur 1, c'est donc son caractère de rang i. Ceci est typique du fait que le langage ne connaît pas le type de donnée « **caractère** » , mais seulement la chaîne. Enfin « ! » dénote la concaténation (mise bout-à-bout) de deux chaînes.

```
v← ''
POUR i ←1 JUSQUA lgr (c) FAIRE
        v ← sch (c, i, 1)!v
BOUCLER
    *   inverse (c) = v
```

Essayons d'évaluer le temps pris par ce programme. Il y a un parcours de la chaîne directe, qui prend un temps proportionnel à sa longueur n, disons βn. Généralement (en tout cas, dans les langages énumérés ci-dessus), la concaténation se fait en recopiant bout-à-bout les chaînes à concaténer, caractère par caractère. Le temps de la concaténation de a et b est donc proportionnel à la somme des longueurs de a et de b. A l'étape i, il y a i-1 caractères dans v, et on y ajoute un caractère. Le temps de concaténation est donc $\alpha((i-1)+1) = \alpha i$. Le temps total de concaténation est la somme de ces temps pour i allant de 1 à n, soit $\alpha n\,(n+1)/2$. Le temps d'exécution du programme est donc $\alpha n\,(n+1)/2 + \beta n$.

Pour n suffisamment grand, c'est le terme quadratique qui prédominera : le temps d'inversion varie comme le carré de n. Si on double la longueur de la chaîne, on quadruple le temps d'inversion. Des essais faits en LSE89 sur un IBM PS2 pour des chaînes dont la longueur est multiple de 800 caractères ont donné les résultats suivants :

p=n/800	temps mesuré	p (p+1)/2	6*p (p+1)/2
1	6	1	6
2	18	3	18
4	65	10	60
8	242	36	216

Les temps sont mesurés en secondes, de sorte que les petites valeurs sont relativement peu précises. Il y a un assez bon accord entre la théorie et l'expérience.

3.3. Première solution récursive

Nous voulons former LEON à partir de NOEL. Nous pouvons remarquer que le premier caractère L de la donnée est le dernier du résultat, et que le reste de la donnée EON se retrouve inversé en tête du résultat NOE. Qui sait voir cela a immédiatement une définition récursive de la fonction « inverse ». Nous appellerons « tête(c) » le premier caractère de c, « reste(c) » la chaîne c privée de sa tête.

inv(c) = inv(reste(c)) ! tête(c)

Il faut un cas particulier arrêtant le calcul. Nous remarquons que l'appel récursif porte sur reste(c) qui est plus court que c. L'arrêt est à chercher du côté des chaînes courtes. La chaîne vide est son propre inverse. D'où la définition récursive

inv1 (c) = SI c = '' ALORS ''
SINON inv1 (reste(c)) ! tête(c)
IS

Essayons d'évaluer le temps nécessaire au calcul de l'inverse par cette définition. Il ne dépend pas des caractères qui forment c, mais seulement de leur nombre n. Soit t1 (n) le temps nécessaire pour calculer inv1 (c). Il faut d'abord calculer inv1 (reste(c)). Or reste(c) est c privé de son premier caractère. Sa longueur est n-1. Le temps pour l'inverser est t1 (n-1). Après quoi, il faut concaténer ce résultat avec tête(n). La longueur de inv(reste(c)) est n-1, celle de tête(c) est 1, le temps pour concaténer inv(reste(c)) et tête(c) est $\alpha((n-1)+1) = \alpha n$. Nous avons une définition implicite de t1 (n)

t1 (n) = SI n = 0 ALORS 0 SINON t1 (n-1) + αn IS

Il en résulte que t1 (n) est un polynôme du second degré en n. Il est nul pour n = 0. On trouve facilement

t1 (n) = αn (n+1) / 2

qui est la même valeur que pour la solution itérative. Malheureusement, l'expérience ne peut confirmer ce résultat. Il se produit en effet que pour n = 230 caractères, le calcul ne peut se faire, la mémoire disponible étant saturée. Pour les valeurs inférieures, le temps de calcul est trop faible pour être mesurable. Il faut rendre compte de cette saturation de la mémoire. La place nécessaire au calcul de inv(c) peut être évaluée de façon semblable. Il faut d'abord la place de c, soit n caractères, puis celle pour inverser reste(c), et enfin celle de la concaténation de n caractères, donc n caractères. Au total,

pl (n) = n + pl (n-1) + n = pl (n-1) + 2n

La place nécessaire est elle aussi quadratique. Dans la solution itérative, à l'étape i, on avait une chaîne z de longueur i-1, puis on formait

v ← sch (c, i, 1) ! v

demandant une place de i caractères pour la nouvelle valeur de z. Mais celle-ci rendait caduque l'ancienne valeur, libérant i-1 caractères. A chaque pas de la boucle, la place utilisée augmentait de 1 caractère, donnant un encombrement linéaire et non plus quadratique. Il fallait intermédiairement un espace de travail de i caractères, inférieur à n. L'encombrement était bien linéaire, d'où la possibilité de monter jusqu'à 8 800 caractères.

Au total, la solution récursive ne prend pas plus de temps que la solution itérative (contrairement à ce qui est si souvent enseigné), mais beaucoup plus de place. Avantage à la solution itérative.

3.4. Solution itérative associée

Utilisons inv1 pour calculer l'inverse de la chaîne 'abcdef'.

inv1 ('abcdef') = inv1 ('bcdef') ! 'a'
inv1 ('bcdef') = inv1 ('cdef') ! 'b'
donc
inv1 ('abcdef') = inv1 ('cdef') ! 'b'!'a' = inv1 ('cdef') ! 'ba'
On pourrait utiliser encore une fois la définition de inv1, donnant
inv1 ('abcdef') = inv1 ('def') ! 'cba'

On peut généraliser les diverses formes obtenues en disant que dans tous les cas, l'inverse cherché est l'inverse de quelque chose suivi de quelque chose

inv1 (c) = inv1 (u) ! v

Le calcul s'arrête si on connaît l'inverse de u. Or le seul cas connu, d'après la définition, est celui où u = '' et alors inv1 (u) = '' et inv1 (u)!v = ''!v = v Si u est non vide, toujours par la définition

inv1 (u) = inv1 (reste(u)) ! tête(u)
inv1 (c) = (inv1 (reste(u)) ! tête(u)) ! v
= inv1 (reste(u)) ! (tête(u) ! v)

On retrouve la forme ci-dessus si on substitue reste(u) à u et tête(u) ! v à v. Au départ, on peut mettre inv1 (c) sous la forme inv1 (u) ! v en prenant u = c et v = ''. D'où le programme cherché

u ← c ; v ← ''
FAIRE
* inv1 (c) = inv1 (u) ! v
SI u = '' ALORS FINI IS
v ← tête(u) ! v ; u ← reste(u)
BOUCLER
* inv1 (c) = v

Interprétons ce programme. A chaque pas de boucle, la tête de u est concaténée en tête de v, et retirée de u. Pour inverser une chaîne, on enlève la tête de la chaîne directe, et on la met en tête de la chaîne inverse. C'est ici qu'on touche du doigt la différence entre l'implicite et le constructif : la définition récursive nous dit

qu'on peut inverser une chaîne de longueur n si on peut inverser une chaîne de longueur n-1. Or, on peut inverser une chaîne de longueur vide. La définition constructive nous dit comment faire : on enlève la tête de la chaîne directe, on la met en tête de la chaîne inverse. Il est vrai que c'est un plus par rapport à la définition récursive.

Ce programme itératif ressemble à celui que nous avons d'abord construit en ce que les caractères de l'inverse sont bien concaténés en tête de z, mais ils sont retirés au fur et à mesure de la chaîne directe. Dans la première forme, ils y étaient simplement lus. On peut dériver le premier programme de celui-ci en remarquant qu'en effet les têtes successives de u sont les caractères successifs de c. On économise ainsi les recopies de morceaux de plus en plus courts de c. Il est évident a priori que cette deuxième forme itérative est plus lente que la première, et qu'elle prend plus de place. On constate expérimentalement qu'on a perdu un facteur deux, à peu de choses près :

p=n/800	temps mesuré	temps première forme
1	10	6
2	35	18
4	126	65
8	479	242

3.5. Deuxième forme récursive

Nous sommes partis d'un découpage de la chaîne directe en deux parties, faisant apparaître la tête et le reste

LEON ⟶ L EON

Nous aurions pu séparer le dernier caractère

LEON ⟶ LEO N

Ceci ne change rien aux performances de la procédure récursive. La forme itérative associée opère en enlevant le dernier caractère de la chaîne directe et en le concaténant au bout de la chaîne inverse. Rien de nouveau. Mais nous aurions pu aussi découper en morceaux ayant chacun plusieurs caractères. Reprenant l'idée d'un partage en parties égales, on peut découper ainsi

LEON ⟶ LE ON

donnant pour l'inverse

NO EL ⟶ NOEL

L'inverse d'une chaîne est l'inverse de sa seconde moitié suivi de l'inverse de sa première moitié. Là encore, la longueur des morceaux à inverser diminue, mais beaucoup plus vite parce qu'on la divise par 2 à chaque fois. On arrivera ainsi à la longueur 1, chaîne d'un seul caractère qui est son propre inverse.

```
inv2 (c) = SI lgr (c)=1 ALORS c
           SINON inv2 (sch (c, p+1,' ') ! inv2 (sch (c, 1, p))
           IS
           AVEC p = ent (lgr (c) / 2)
```

Cette définition paraît beaucoup plus mauvaise que la première : il y a **deux appels récursifs** et non plus un. Nous allons voir qu'il n'en est rien. Evaluons d'abord le temps d'exécution t2 (n). En gros, c'est le temps d'inversion de la deuxième moitié, plus celui de l'inversion de la première moitié, plus celui de la concaténation des deux. De façon plus précise, nous allons considérer les deux cas où n est pair, n = 2p puis n impair n = 2p+1, à cause de la définition de p

$$t2(2p) = 2\ t2(p) + \alpha\ 2p$$
$$t2(2p+1) = t2(p) + t2(p+1) + \alpha(2p+1)$$

Nous ne développons pas ici les calculs nécessaires à la résolution de cette définition implicite. On trouve que $t2(n) \approx \alpha\ n \log2(n)$ (c'est très exactement une ligne brisée dont les sommets sont les points de cette courbe pour les n qui sont des puissances de 2). L'important est que le temps de calcul n'est plus quadratique, comme dans toutes les formes étudiées jusqu'ici. **Il est beaucoup plus court** Voici les temps mesurés, comparés à la meilleure version itérative :

n/800	temps mesuré	temps première forme
1	4	6
2	10	18
4	19	65
8	39	242

Nous n'avons pas gagné une fraction du temps de la solution itérative, nous avons changé d'ordre de grandeur. La solution récursive est beaucoup plus efficace que la solution itérative, parce qu'elle est fondée sur un partage dichotomique.

Nous pourrions étudier la place nécessaire comme nous l'avons déjà fait plus haut, mais il est inutile de multiplier les calculs. Il suffit de noter que pour inverser la chaîne de longueur n, on inverse successivement ses deux moitiés, en sorte que les places nécessaires ne s'additionnent pas : il suffit de la place nécessaire pour inverser une moitié, plus celle de la concaténation. La place nécessaire est linéaire. De fait, la saturation se produit pour 8 800 caractères, comme pour la première solution itérative.

Il n'y a pas de façon simple de construire une forme itérative équivalente, à cause des deux appels récursifs. On peut tenter de réaliser itérativement la stratégie récursive. On inverse d'abord la seconde moitié, ce qui amène à inverser d'abord sa seconde moitié, et, de proche en proche, les deux derniers caractères. Pour inverser 'abcd', on forme d'abord 'dc', puis 'ba' puis on les concatène en 'dcba'. Pour la chaîne 'abcdefgh' on aura ainsi successivement en mémoire

chaînes stockées	temps de formation	total
'hg'	2α	2α
'hg' 'fe'	2α	4α
'hgfe'	4α	8α
'hgfe' 'dc'	2α	10α
'hgfe' 'dc' 'ba'	2α	12α
'hgfe' 'dcba'	4α	16α
'hgfedcba'	8α	24α

Notons que la première forme itérative aurait demandé un temps de concaténation de 36α. La difficulté d'une solution itérative opérant par dichotomie est claire sur cet exemple : il faut gérer des chaînes intermédiaires (3 pour une chaîne de 8 caractères, log2 (n) dans le cas général), et cela rend la forme itérative extrêmement complexe. Nous avons essayé de l'écrire, sans succès raisonnable.

Ainsi, nous avons une forme récursive, significativement plus rapide que toute forme itérative, et pas plus encombrante. La récursivité est ici beaucoup plus avantageuse que l'itération. Ceci invalide tous les discours qui présentent la récursivité comme intellectuellement avantageuse, mais moins efficace. Elle peut être aussi beaucoup plus efficace.

4. UNE VARIANTE : LE MIROIR D'UN NOMBRE

Nous avons étudié l'inverse d'une chaîne dans un contexte où elle n'est pas un tableau de caractères. Un nombre, dans son écriture en base quelconque (10 par exemple, celle dont nous avons l'habitude), est aussi une chaîne de caractères ayant cette propriété. On peut en former l'inverse, qui est aussi la représentation d'un nombre dans la même base : nous l'appelons le « miroir » du premier. 743 a pour miroir 347. Essayons de le définir récursivement, comme ci-dessus. Il est difficile d'isoler le chiffre de gauche (il faudrait connaître l'ordre de grandeur du nombre), mais il est facile d'isoler le chiffre de droite : c'est le reste de la division du nombre par 10. Le reste du nombre est son quotient entier par 10.

mod (743, 10) = 3 entier (743 / 10) = 74

Le résultat est obtenu en écrivant bout-à-bout 3 et le miroir de 74. Il faut donc multiplier 3 par la bonne puissance de 10, et ajouter le miroir de 74

miroir (347) = 3 * 100 + miroir (74)

Dans le cas général

miroir (n) = h * mod (n, 10) + miroir (ent (n / 10))

où h est la bonne puissance de 10. Reste à la définir. Notons d'abord que ent (n / 10) est plus petit que n (sauf si n = 0), et donc l'arrêt de la récursivité est à chercher du côté des petites valeurs de n. Si $n < 10$, il est son propre miroir. Si $10 \leq n < 100$, h vaut 10. Si $100 \leq n < 1000$, h vaut 100. Toutes les fois que n est dans une tranche

10 fois plus grande, h est multiplié par 10. Donc, h(n) = 10 * h(ent(n / 10)). Avec les valeurs que nous venons de trouver, n < 10 donne h = 1 D'où le résultat cherché

```
miroir(n) =     SI n < 10 ALORS n
        SINON h(n) * mod(n, 10) + miroir(ent(n / 10))
        IS
h(n) = si n < 10 ALORS 1 SINON 10 * h(ent(n / 10)) IS
```

Il faut une paire de fonctions récursives pour arriver à définir la fonction miroir. On peut essayer d'autres découpages : quelle que soit la façon de faire, il faut une fonction d'ordre de grandeur telle que h.

Nous avons donné une stratégie itérative : on enlève la tête de la chaîne directe, et on la concatène en tête de la chaîne inverse. Mais il est difficile de prendre la tête d'un nombre. Nous avons vu qu'on pouvait aussi enlever le dernier caractère de la chaîne directe et le mettre en queue de la chaîne inverse. Le dernier chiffre de n est mod(n, 10). Pour l'écrire à droite de r, on multiplie r par 10 et on l'ajoute au résultat. D'où un programme itératif immédiat

```
lire n ; r ← 0
TQ n ≠ 0 FAIRE
        r ← 10 * r + mod(n, 10) ; n ← ent(n / 10)
BOUCLER
afficher r
```

Pour n = 1000, il calcule comme miroir 0001, et l'écrit 1. Ainsi, nous avons un programme itératif très simple, formé d'une seule boucle, sans équivalent récursif simple : il faut une paire de fonctions récursives pour dire la même chose. Bien sûr, connaissant le résultat, on peut arriver à fabriquer une procédure récursive raisonnablement simple. Mais c'est après un détour par l'itératif qui est ici premier. Encore qu'il n'a été possible que par la deuxième forme itérative donnée plus haut, laquelle n'est pas venue directement, mais a été déduite d'une forme récursive. Tout ceci est inextricablement embrouillé.

5. LE POINT DE LA SITUATION

Il est difficile de tirer des conclusions d'un exemple. Nous pouvons tout au plus forger des hypothèses, mais Pascal nous a instruit sur leur nature dans sa « réponse au Père Noël » :

> « une hypothèse peut être de trois sortes. Car quelquefois on conclut un absurde manifeste de sa négation, et alors l'hypothèse est véritable et constante ; ou bien on conclut un absurde manifeste de son affirmation, et alors l'hypothèse est tenue pour fausse ; et lorsqu'on n'a pu encore tirer d'absurde ni de sa négation, ni de son affirmation, l'hypothèse demeure douteuse ; de sorte que, pour faire qu'une hypothèse soit évidente, il ne suffit pas que tous les phénomènes s'en ensuivent, au lieu que, s'il s'ensuit quelque chose de contraire à un seul des phénomènes, cela suffit pour assurer sa fausseté. »

Le Père Noël en question était le recteur des Jésuites à Paris, qui s'en était pris à Pascal au sujet de son interprétation des expériences de Toricelli. On a affirmé que la récursivité était plus facile à construire que l'itération, mais qu'elle était moins efficace. Un seul contre-exemple suffit à invalider ces affirmations : nous avons exhibé une fonction qui est plus facile à construire itérativement que récursivement, et une autre qui est beaucoup plus efficace récursivement qu'itérativement. Mais il y a de nombreux exemples en sens inverse, ne fût-ce que celui des nombres de Fibonacci.

Dès lors, il faut renoncer à des conclusions définitives en ce domaine. Il y a deux façons de définir un calcul, l'une implicite, par la récursivité, l'autre constructive, par l'itération. Leurs puissances d'expression sont les mêmes. Leurs avantages changent suivant le problème considéré. Restent donc uniquement les goûts et formes de pensée des programmeurs. Il y en a qui ne sont pas très à l'aise avec la récurrence, et encore moins avec la récursivité. Qu'ils utilisent donc l'itération! Il y en a pour qui la récurrence est une forme usuelle de raisonnement. Ils sont parfaitement à l'aise avec les définitions implicites. Tant mieux pour eux, ils iront le plus souvent plus vite au résultat, mais souvent avec une moindre efficacité. Quelques fois, ils obtiendront les meilleurs résultats. Parfois, ils auront du mal à obtenir leurs définitions implicites. L'expérience montre qu'ils se recrutent plutôt chez les forts en mathématiques.

Heureux ceux qui peuvent jouer sur les deux tableaux. Ils tireront de chaque forme de programmation ce qu'elle peut offrir de meilleur. Le problème de l'enseignant est plus compliqué. Que choisir pour l'initiation des élèves à la programmation ? La première réponse est qu'on enseigne le mieux ce que l'on maîtrise le mieux. Si le professeur doit se forcer pour enseigner un de ces modes de programmation, les élèves s'en apercevront, les résultats ne seront pas extraordinaires. Qu'il utilise donc celui qui lui va le mieux. S'il maîtrise également les deux modes, il y a les conditionnements psychologiques et les fonds culturels des élèves. Peut-on les connaître avant de choisir la programmation à enseigner ? Il paraît vain d'espérer gagner sur les deux tableaux : il est déjà difficile d'obtenir de bons résultats avec un des modes, il faudrait beaucoup de temps et d'efforts pour que les élèves soient également performants dans la récursivité et dans l'itération. En ce qui me concerne, me fiant à la remarque que la récurrence n'est pas une forme de raisonnement généralement maîtrisée par les élèves, et que les matheux y sont les mieux préparés, je privilégie la forme itérative. Sa pratique servira à développer l'usage de la récurrence, et préparera à la récursivité. Ce n'est qu'une conclusion personnelle, les précautions prises ici doivent assez le montrer. Il n'y a pas de vérité révélée, ni prouvée par une théorie, dans ce chapitre de la pédagogie.

Jacques ARSAC

Professeur émérite à l'Université P. et M. Curie
Correspondant de l'Académie des Sciences