Louis Gacôgne

Second colloque francophone sur la didactique de l'informatique (Namur 1990)

Spécificité de l'initiation à la programmation fonctionnelle dans la formation scientifique.

Résumé

Cette intervention a pour but de rassembler quelques réflexions issues d'un enseignement au cours de la deuxième année suivant le baccalauréat, au sein d'une école d'ingénieurs et d'en tirer le bilan. Ce cours de programmation fonctionnelle, au moyen du langage Lisp qui en est le meilleur représentant, fait suite à un enseignement de programmation classique en Pascal, donné en première année.

L'intérêt de ce cours est multiple :

_ apporter un autre point de vue sur la programmation de problèmes non numériques se posant aux informaticiens,

_ introduire un outil qui sera utilisé en dessin assisté par ordinateur,

_ mais surtout, montrer une façon cohérente et extrêmement générale de poser les problèmes au moyen du concept de récurrence, et amener, grâce à un investissement en connaissances brutes très réduit, à un nouveau savoir-faire et à une réflexion de nature épistémologique.

Deux questions préalables se posent à propos d'une didactique de l'informatique.

Finalité de l'enseignement de la programmation

Ce thème de la finalité et des contenus a souvent été débattu. On peut y répondre brièvement en écartant d'emblée les questions de l'informatique générale (notions sur le matériel, sur l'architecture des systèmes et sur les applications professionnelles), car cette partie de l'enseignement doit rester limité par exemple à quelques conférences sauf pour les étudiants se destinant précisément à l'informatique. Le débat porte généralement sur l'apprentissage de la programmation, en quoi est-il nécessaire dans la mesure où cette activité ne sera exercée que par une minorité d'individus même au sein de la corporation des informaticiens ?

A cette question, on doit retourner celle - identique - de la finalité de l'enseignement des mathématiques dont on sait bien qu'elles ne serviront en tant que telles qu'à une minorité même parmi les étudiants scientifiques. Il est tout à fait possible de répondre aux deux questions en insistant sur l'aspect formateur, pour l'esprit scientifique, de ces deux disciplines de concert avec les autres sciences.

On a souvent remarqué qu'en mathématiques, peu de connaissances étaient exigées, alors que leur but étaient plus une connaissance à un autre niveau : un savoir-faire et des méthodes. Ceci est encore bien plus vrai de la programmation, en effet, dans un langage classique comme Basic, Fortran ou Pascal il est possible de démarrer de nombreuses applications avec très peu de connaissances brutes (les mots signifiants pour demander un affichage, une entrée de données, une affectation, une itération, un test ou un débranchement), le reste est affaire de méthode, c'est-à-dire met à contribution l'esprit analytique en formation, avant même qu'il ne soit question d'examiner des algorithmes classiques.

Mais là, un enseignant serait de mauvaise foi s'il prétendait donner des méthodes à ses étudiants pour venir à bout de façon systématique de l'analyse et programmation de n'importe quel problème. Pas plus qu'on ne peut "apprendre" à dessiner, ou "apprendre" à résoudre un problème de mathématiques, on ne peut "apprendre", ni à rédiger correctement l'analyse d'un problème, ni à construire un programme bien structuré du premier coup. Il s'agit d'un exercice intellectuel (artistique dans le premier cas), qui ne peut en aucun cas se ramener à l'utilisation de recettes, et ne

peut en revanche être mené à bien que par une lente et laborieuse formation de l'esprit d'où l'intuition et le tâtonnement ne sont pas exclus.

Après avoir longuement travaillé aux transformations de programmes, à simplifier au maximum le processus de création, je me trouve confronté à un des plus vieux problèmes de l'humanité: comment invente-t-on? Il se trouve que la pratique de l'informatique m'a permis de trouver du neuf dans des sentiers infiniement battus. Y aurait-il donc dans l'approche informatique des problèmes une puissance créatrice nouvelle? Je suis tenté de recommander des heuristiques en matière de création de programmes, mais en même temps je crains toujours que l'habitude d'une certaine façon d'opérer ne bloque la possibilité de faire autrement dans des cas qui paraissent le nécessiter. Comme pour l'apprentissage du bridge, il y a des "règles", et le débutant fait bien de s'y tenir, même si il peut éventuellement en pâtir. Le bon joueur est cependant, celui qui n'est pas esclave des règles. (Arsac)

Quel rendement pour cet enseignement?

Le second sujet préliminaire à une réflexion pédagogique, est celui très rarement évoqué, de la notion de rendement dans l'enseignement. Il serait en effet curieux d'essayer d'évaluer les quantités de connaissances reçues, la quantité réellement acquise par l'étudiant, et ceci rapporté au temps utilisé pour une formation donnée. Dans l'enseignement supérieur, les temps sont généralement comptés en heures de cours ou de travaux dirigés, dans l'enseignement secondaire il sont comptés en heures hebdomadaires, or l'enseignement secondaire délivre par exemple pour l'apprentissage du calcul algébrique élémentaire, un total d'environ 500 heures réparties sur cinq ans, puis, dans le cas des sections scientifiques encore environ 500 heures réparties sur les deux dernières années de lycée pour une introduction véritable de mathématiques avec notamment le début de l'analyse, et il n'est plus que de 350 heures pour le premier cycle dans le cas d'études courtes comme celles de techniciens supérieurs.

Quant à l'informatique, la situation peut être fort variable. Mon expérience personnelle est d'enseigner dans une école d'ingénieurs où il n'est prévu que 80 heures pour cette discipline dans les deux premières années, dont d'ailleurs les trois quarts se situent en première année. Que faire dans ces conditions, surtout si l'on veut ne pas trop s'écarter de ce qui se fait d'une manière à présent généralisée en France, à savoir un enseignement centré sur quelques problèmes algorithmiques, essentiellement en liaison avec les programmes de mathématiques, et mis en œuvre au moyen du langage Pascal? Que faire pour donner un aperçu des problèmes de nature plus symbolique (l'intelligence artificielle restant un bien grand mot), et que faire pour libérer l'étudiant de l'aspect fort contraignant du Pascal?

Le rôle que peut jouer la programmation fonctionnelle

A toutes ces questions, la programmation fonctionnelle permet d'apporter des réponses neuves et somme toute satisfaisantes. Comme on le détaillera un peu plus loin, l'apprentissage de la programmation fonctionnelle par le langage Lisp qui la représente le mieux, permet par la simplicité de ses principes de base, par la très grande cohérence de ses notations, (contairement au Logo) et par sa capacité à traduire des problèmes complexes, de fortement contribuer à la formation scientifique pour un investissement en connaissances pures, et en temps de travail réduits. Il permet surtout, en très peu de temps, d'accéder à des problèmes qu'il serait pratiquement impossible d'aborder dans un style de programmation impératif et non récursif.

Après plus de vingt-cinq ans, Lisp continue à susciter l'entousiasme de ses utilisateurs. Cet "optimum local dans l'espace des langages de programmation"(Mac Carthy) n'a pas encore de successeur offrant un "rapport qualité/prix" comparable. On connaît ses insuffisances, ses défauts mais il reste le moyen le pluscommode pour

dicter sa volonté à un ordinateur, grand ou petit. De plus sa structure de base est simple mais profondément intéressante : apprendre Lisp, c'est apprendre un des plus beaux chapitres de l'informatique. Enfin, et c'est sans doute le plus important, Lisp "laisse dans une certaine mesure l'imagination intacte" comme l'écrivait P. Greussay il y a dix ans : rare vertu pour un langage de programmation. Il est difficile d'expliquer le charme de Lisp à ceux qui ne l'ont jamais pratiqué, ou qui en sont restés aux premiers balbutiements. C'est un mélange de rigueur, d'élégance mathématique, et de précision dans la conduite de la machine. Le même mot revient chaque fois qu'on veut en parler : la beauté. J'y ajouterai celui de la culture : en Lisp se sont exprimés depuis vingt-cinq ans certains des meilleurs programmeurs du monde. Ils ont découvert et résolu quantité de problèmes, et laissé un héritage incroyablement riche, auquel seule la connaissance du langage donne accès. En matière de programmation, Lisp c'est la liberté. (J.F.Perrot)

A l'heure actuelle, une telle initiation au langage Lisp est indispensable dans tout enseignement d'informatique, il peut enrichir l'étudiant en lui ouvrant d'autre horizons que ceux, maintenant classiques, d'une programmation impérative.

L'initiation au Lisp

Rappelons tout d'abord qu'en Lisp toutes les expressions s'écrivent en notations préfixées et parenthésées. Pédagogiquement ces notations surprennent l'étudiant qui est habitué depuis des années aux notations mathématiques traditionnelles, où se mêlent des notations préfixe telles que sin(x), infixe comme x + y, suffixe (x!, x^2), ou encore des deux côtés comme la valeur absolue, est troublé par une cette écriture. C'est là un des inconvénients pédagogiques du Lisp, cependant l'expérience montre qu'on peut s'y habituer assez facilement, et cette facilité d'apprentissage doit être mise sur le compte de la cohérence que n'ont point les notations mathématiques héritées de l'histoire.

Le second point, qui, lui, est fondamental, est l'aspect toujours fonctionnel qui déroute complètement le programmeur habitué au Basic ou au Pascal. On ne fait jamais autre chose en Lisp, que de demander la valeur d'une expression, à savoir, généralement, la valeur d'une fonction f pour des valeurs données à ses arguments x, y, z, ... Là encore un problème de notation vient troubler l'étudiant puisque la valeur f(x,y,z,...) désirée, est demandée à la machine sous la forme (f x y z ...).

Enfin, puisqu'il n'y a que des fonctions il faut donner les fonctions de bases, essentiellement les fonctions de traitement de listes, soit une dizaine de fonctions (quote, eval, car, cdr, cons, list, eq, null, atom, cond, defun)

Les grands principes d'une bonne programmation en Lisp, pour reprendre une expression de J. Arsac citée plus haut, me paraissent être les suivants :

_ Ne pas se préocupper des problèmes d'entrées-sorties, ceci répugne parfois aux étudiants qui sont pressés de soigner la présentation au détriment de l'essentiel, mais permet une épargne de temps importante.

_ Se dispenser d'itérations, le Lisp étant parfaitement adapté à la récursivité, c'est ce grand principe qui doit être exploité : dans presque tous les cas, une fonction sera définie au moyen d'une succession de conditions, qui, pour simplifier, sont les cas (généraux) où il suffit de reporter le même problème à des arguments "plus petits", et aussi les cas particuliers correspondants en fait aux "tests d'arrêts" des enchaînement récursifs.

L'étudiant est obligé de faire - provisoirement - un certain acte de foi : accepter ce qu'il ne découvrira que petit-à-petit, la facilité d'expression récursive de quantité de problèmes, et la puissance de la fonction "cond" qui en permet une présentation claire. Il y a là un barrage psychologique tout à fait analogue à celui rencontré lors de l'enseignement du calcul algébrique élémentaire dont nous parlions précédemment, mais à ceci près, que la durée de cette attente peut être réduite à quelques heures.

_ Eviter les affectations, et donc les transformations physiques, et d'une manière générale tous les effets de bord.

L'aspect passage des paramètres, uniquement par valeurs, est en principe connu de celui qui a déjà programmé en Pascal. La situation - toujours dans l'esprit d'une initiation - est en fait beaucoup plus simple que celle qui est rencontrée dans l'apprentissage du Pascal, où de véritables fonctions au sens mathématique, ne peuvent se traduire lorsqu'elles sont de type structurées, que par des procédures ayant des paramètres passés par valeur (les données) et des paramètres passés par adresse (les résultats). Cette façon de rassembler les données des résultats n'est pas naturelle à l'esprit de celui qui l'aborde, et nous gagnons donc ici encore en cohérence et en simplicité.

_ En élargissant, on peut dire qu'aucune utilisation de variables même locales n'est nécessaire. On pourrait d'ailleurs émettre un principe général qui consiste à construire des fonctions possédant beaucoup de paramètres. Signalons d'ailleurs que l'étudiant débutant commet souvent de grandes confusions entre variables globales, locales et paramètres, mais qu'il est par ailleurs perplexe devant ce manque total de variables, lorsqu'il connaît déjà le Pascal. Ceci pourrait constituer un argument en faveur d'une initiation à la programmation directement par le Lisp.

L'expérience a montré que ces généralités sur les notations, la programmation fonctionnelle, un aperçu sur la représentation interne des listes, et l'énoncé d'une dizaine de fonctions de bases prend un peu plus de deux heures. En alternant les cours avec les séances de travaux pratiques, pendant au moins six heures, l'étudiant doit donc avoir la patience de se plier à de petits exercices abstraits tels que :

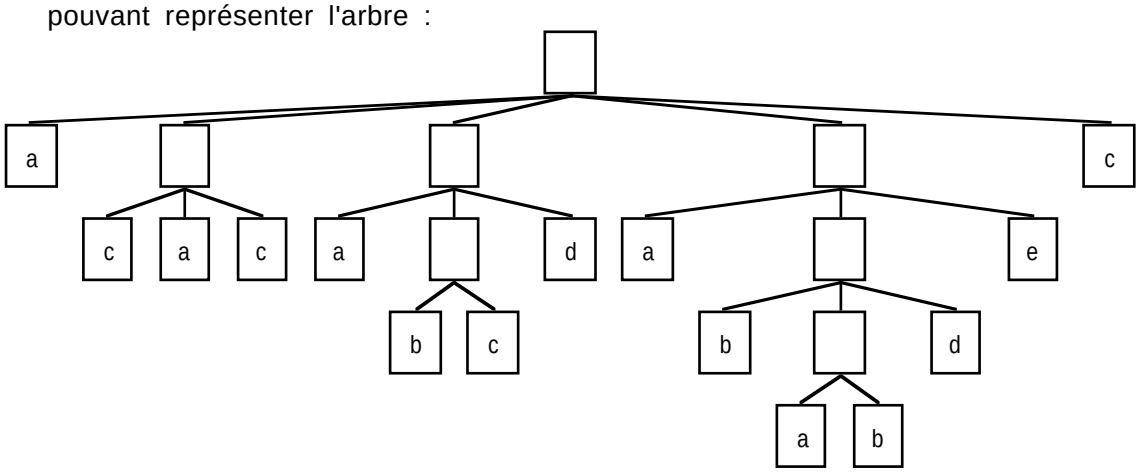
_ retirer un élément d'une liste,
_ concaténer deux listes,
_ exprimer la tête ou la queue d'une liste,
_ écriture des prédicats d'appartenance et d'inclusion au sens ensembliste etc...

Mais c'est là qu'il comprend bien, par l'exemple, l'idée de récurrence. C'est long lorsque l'on souffre de l'abstraction, mais c'est en fait très bref comparé aux heures passées pour l'apprentissage de n'importe quelle notion nouvelle de mathématiques.

Un exemple montrera l'élégance d'une programmation récursive, il s'agit de construire une fonction calculant le nombre de chemins dans une arborescence, ou si l'on préfère de compter le total des éléments (pas seulement les 5 éléments au premier niveau, mais les "éléments" à toutes les profondeurs, il y en a 15) figurant dans une liste telle que :

(a (c a c) (a (b c) d) (a (b (a b) d) e) c)

pouvant représenter l'arbre :



La programmation de cette fonction que l'on appelera f se fait par une condition en trois clauses :

```
(de      f  (X)  (cond  ((null  X)     0 )
                        ((atom  X)     1 )
                        ( t      ( +     (f  (car  X))    (f  (cdr  X))   )   )   ))
```

En clair, nous avons le cas de l'objet vide qui compte pour 0, celui d'un atome (une chaîne alphanumérique) qui compte pour 1, et le cas général qui se contente de dire qu'il faut additionner ce que l'on trouve dans le premier objet de la liste X, avec tout ce qui va suivre. Si on y prend garde le paramètre X peut être n'importe quelle expression, il est alors certain que c'est une liste dans le troisième cas.

Cet exemple est assez spectaculaire, car pour produire le même résultat, dans un langage tel que Pascal, il faudrait un apprentissage régulier d'au moins un an sur les types de données, les pointeurs etc...

On peut encore remarquer à propos de cette fonction, comme de toutes les autres, que lors de leur recherche, l'élève arrive assez bien par un processus d'assimilation à exprimer en français, puis en Lisp, le cas général, mais qu'il lui faut plus de peine pour trouver les cas particuliers qui sont toujours à placer en premier (les tests d'arrêt). C'est pourquoi il faut montrer par l'exemple ces fonctions en les recherchant au tableau, plutôt que de les donner toutes faites.

Il faut donc compter au moins six heures avant d'aborder quelques sujets concrets tels que :

_ les tris par insertion, et par fusion, et l'évaluation de leurs complexités,
_ la transcription des chiffres romains,
_ l'équivalence de deux listes à un symbole X près,
_ l'algorithme de Hörner etc ...

Le perfectionnement

Par la suite il est possible au bout donc d'une dizaine d'heures d'aborder des problèmes nécessitant un dédoublement en plusieurs fonctions éventuellement mutuellement récursives. Mais donnons un exemple, qui à première vue contredit ce qui a été dit plus haut, il s'agit du calcul de la moyenne d'une liste de valeurs numériques. C'est un des premiers sujets que l'on peut donner en programmation impérative, à cause de sa simplicité, alors qu'ici la récursivité semble moins bien adaptée, et pourtant c'est un excellent exemple méthodologiquement parlant, car il faut se mettre à la place de l'éxecutant au cours de son travail.

Si nous prenons l'image du vendeur désirant établir la moyenne de ses ventes au cours de la journée, c'est quelqu'un qui n'a besoin à tout moment que d'avoir noté le nombre N de ventes, et le total S de ces ventes, il aura pour le reste de la journée une liste L de montants de ventes.

_ Chaque étape consiste à avancer d'un cran dans la lecture de la liste L, en modifiant les paramètres N et S.

_ La fin du problème a lieu lorsque la liste L est vide.

D'où la fonction :

```
(de moyenne_bis (N S L)
    (cond  ((null  L)    (/  S  N))
           ( t     (moyenne_bis  (+  N  1)  (+  S  (car  L))  (cdr  L)))   )))
```

_ Quant au démarrage du problème, on peut le réaliser grâce à une autre fonction:

```
(de moyenne (L)  (moyenne_bis  0  0  L))
```

Ce partage d'un problème en plusieurs fonctions se passant autant de paramètres qu'il faut, est extrêmement général et se retrouve dans beaucoup de thèmes qui peuvent être immédiatement abordés après cette introduction, par exemple :

_ l'addition et la multiplication des grands nombres représentés par la liste de leurs chiffres,
_ la longueur du plus grand plateau d'éléments consécutifs identiques au sein d'une liste,
_ un petit système expert sans variables,
_ le problème de la dérivation formelle des fonctions d'une variable,
_ la mise en œuvre d'un "backtrack" pour l'exploration d'une arborescence.

En conclusion, on peut dire que le point de vue fonctionnel (encore appelé applicatif ou transformationnel) apporte un bon moyen d'enchaîner les différentes parties d'un problème, et ceci tant en analyse ascendante que descendante. Son aspect déroutant doit apporter un profit à l'étudiant qui s'enrichit d'une nouvelle utilisation de notions déjà acquises plus ou moins profondément. Problème déjà souligné par C. Pair et aussi par J. Rogalski : *"Dans la construction de nouveaux savoirs, des notions ou des représentations appartenant à un domaine de connaissances préexistant peuvent jouer le rôle de précurseurs."* mais ajoute-t-elle, ceux-ci peuvent avoir un rôle producteur ou réducteur. Ainsi la notion de fonction étudiée en mathématiques est un "précurseur" utile, c'est d'ailleurs le prérequis indispensable, cependant tout ce qui concerne la programmation impérative antérieurement étudiée peut constituer un obstacle. Pour nous, il apparaît que seul l'assimilation patiente par l'exemple permet d'acquérir des méthodes telles que la description récursive des algorithmes, ou l'aspect "récursivités mutuelles" réalisé par des fonctions qui se "passent la main" en "passant" des valeurs.

Notre expérience montre que cet enseignement porte d'autant mieux ses fruits que les étudiants ont déjà une certaine maturité en programmation classique, et qu'alors ces méthodes de travail sont d'autant plus riches qu'elles ouvrent la voie d'une programmation approfondie, en se plaçant au dessus des connaissances brutes et des algorithmes.

Références

J. Arsac La récurrence en programmation Revue de mathématiques spéciales Décembre 81

J.Arsac Quelques sujets de réflexion à propos de la création de programmes in Séminaire d'informatique théorique 82-83 LITP (Universités Paris VI et Paris VII)

J.Arsac La conception des programmes. revue "Pour la Science" Novembre 84

L.Gacôgne Programmation, du procédural au fonctionnel, du fonctionnel au déclaratif. (éd. Eyrolles 88)

J.M. Hoc L'étude psychologique de l'activité de programmation : une revue de la question. Techniques et sciences informatiques 1982 vol.1 n°5

C.Pair L'apprentissage de la programmation. Colloque francophone sur la didactique de l'informatique Université R.Descartes Paris 88

J.F.Perrot Préface du livre de J.P. Roy et G. Kiremitdjian "Lire Lisp" (éd. Cedic Nathan)

J. Rogalski Acquisitions de savoirs et de savoir faire en informatique. Cahier de didactique n°43 IREM-Paris VII