LA LETTRE DE L'ADELI

ET

L'ADELIEN

N°:10

Sommaire:

-Editorial:la concentration de la presse,H.Habrias
-"Je ne sais (toujours pas) enseigner la programmation" C.Pair
-"Introduction à la programmation fonctionnelle" H.Habrias
-"Manifeste contre la logique informatique (Pour l'illogisme pneumatique)" P.Fischof
-"Spécial 1993. Avis aux entreprises françaises. Quelques recettes infaillibles pour gagner" l'Adéli
-"Les composantes intellectuelles de l'informatique illustrées par la cuisine chinoise" ,Davis et Hersh, extrait de "L'Empire mathématique" (Gauthier Villars, 1988), avec autorisation

Rédaction: H.Habrias, IUT,3 rue Ml Joffre 44041 Nantes Cedex 01
tel:40 30 60 90, télécopie:40 30 60 01
Imprimé par l'IUT de Nantes

ADELI : ASSOCIATION POUR LE DÉVELOPPEMENT DE LA LOGIQUE INFORMATIQUE
ASSOCIATION A BUT NON LUCRATIF (LOI DE 1901) 67, RUE BOBILLOT, 75013 PARIS - TÉL : (1) 45 65 51 66 - C.C.P. PARIS 4720 77 H
SECRÉTARIAT PERMANENT : (1) 69 03 20 57

EDITORIAL

Concentration dans la Presse, disparition de titres.....
L'Adéli n'y échappe pas ! Le Bureau de l'Adéli vient de nous demander de faire passer les papiers de l'Adélien dans "La lettre de l'Adéli". Nous attendons les papiers de l'Adélien.... Pour commencer nous avons changé la couverture...et avons fait une concaténation de titres.

Tout çà nous a rajeuni("Changer de cravate, une nouvelle cravate vous changera"). Etudiant, nous avions créé un PPPPP (Parti Pour la Promotion de la Pensée Paillarde) et notre voisin en Cité U. avait créé l'AC (Action Canularesque), organismes qui envoyaient régulièrement des communiqués de presse au journal régional. Puis les communiqués eurent du mal à passer. Alors on créa le PPPPP-AC. C'était un événement. On eu droit à notre communiqué.

PPPPP et AC étaient totalement creux. Ils avaient d'autant plus de résonnement . Ils pratiquaient la langue de bois des partis politiques (leurs communiqués auraient pu être fabriqués par ordinateur grâce à un programme générateur de discours "arnacologique"). Résonnement n'est pas raisonnement !

Comme avec "La lettre de l'Adéli" nous recevons très peu d'écho, c'est peut-être que le raisonnement y est plus important que le résonnement. Espérons !

Henri HABRIAS

Commandeur de l'Ordre du Bitard (L.S.T.), Université de Poitiers
Membre de l'ARPAGATELE (Association du Refus du Prélèvement Automatique sur compte pour le Gaz, le TElephone et l'ElEctricité)
Membre fondateur de l'ADAR (Association de Défense des Accents Régionaux)

Claude Pair nous a autorisé à reproduire pour la "Lettre de l'Adéli" cet article paru dans une revue du Ministère de l'Education Nationale consacrée à l'informatique dans l'enseignement: "Informatiques, 2 (1988), pages 5-14

INFORMATIQUE

Claude Pair, Professeur à Nancy est actuellement Chargé de Mission auprès du Secrétariat d'Etat chargé de l'Enseignement technique.

DIDACTIQUE

JE NE SAIS (TOUJOURS) PAS ENSEIGNER LA PROGRAMMATION

Claude PAIR

1. INTRODUCTION

La récente brochure "Pratiques et savoir-faire des élèves de l'option informatique" est stimulante pour la réflexion. Jacques ARSAC distingue plusieurs pédagogies pour l'apprentissage de la programmation: apprendre un langage, regarder l'homme résolvant un problème, réaliser des projets, fournir des schémas, proposer des méthodes... et il choisit clairement la dernière au nom d'objectifs généraux assignés à cet apprentissage dans le cadre du lycée. Janine ROGALSKI replace l'élève au coeur du processus en attirant l'attention sur les conditions d'acquisition des savoirs et savoir-faire. Et d'autres articles ou comptes rendus insistent sur les objectifs, montrent les difficultés rencontrées, discutent des outils et langages utilisables, ou présentent des projets et des activités.

Cependant on entend ici ou là des échos de déception venant des lycéens de l'option informatique. Elle deviendrait trop scolaire, voire scolastique, proposant aux élèves des exercices simplistes pour s'attacher à la description formelle de leurs solutions plus qu'à la recherche et à la créativité. Excès ou trop grande précocité d'un enseignement obnubilé par le développement des méthodes, tuant la créativité qu'elles sont censées développer ? J. ROGALSKI indique d'ailleurs le danger de reproduire certaines erreurs commises dans l'enseignement des mathématiques "modernes". Déjà, il y a quelques années, un collègue mathématicien me parlait d'une dérive "bourbakiste" de l'enseignement de l'informatique.

Moi-même, j'ai beaucoup prêché pour les méthodes, définies à partir des difficultés de la programmation et des erreurs commises par les programmeurs. L'effort de réflexion des informaticiens sur la notion de programme a renouvelé les conceptions et devrait permettre d'éviter les errements passés. Mais cet effort n'a pas pris en compte les conditions psychologiques de l'apprentissage [2]. Et la "dérive bourbakiste" n'est-elle pas justement de croire qu'il suffit de se rapprocher de la vérité d'une discipline pour parvenir à un enseignement efficace ?

La période extra-informatique que j'ai eu la chance de vivre pendant quelques années m'a amené à prendre conscience que l'aspect exclusivement centré sur les disciplines de l'enseignement français, et d'abord de la formation des professeurs de lycée, conduisent à ce type de dérive, qui finalement fait passer l'élève au second plan.

Le programme de seconde de l'option (2) en donne une bonne illustration : *"Le professeur développe des exemples typiques, en insistant sur les aspects méthodologiques. En travaux pratiques, les élèves pourront reprendre ces exemples et traiter, de façon analogue, d'autres problèmes de difficulté comparable".*
N'oublie-t-on pas ici les conditions de l'apprentissage : se fonder sur une expérience de l'apprenant et ne rien présenter de l'extérieur qui ne réponde à un problème rencontré au cours de cette expérience?

Une telle réaction incite à une pédagogie où l'élève est placé dans des situations au cours desquelles il acquiert une expérience de la machine et de la programmation. Les projets se prêtent bien à une telle pédagogie. Malheureusement on aboutit ainsi à un grand nombre d'erreurs, dont on se dit qu'elles auraient pu être évitées par des conseils très simples et à beaucoup de temps apparemment perdu pour parvenir à un résultat acceptable, c'est-à-dire à un programme correct. Si on y parvient jamais. Et au sentiment chez l'enseignant qu'il ne fait pas son devoir en laissant l'élève prendre des mauvaises habitudes et en lui cachant certains acquis bien établis de la science informatique.

Cette expérience, chez l'enseignant cette fois, celle redécouverte du problème éternel de l'enseignement n'est pas inutile... et elle serait utile aussi ailleurs qu'en informatique. Mais pour qu'elle atteigne toute son efficacité et qu'elle ne conduise pas à un balottement perpétuel entre la non-directivité et l'hyper-directivité, il faut avoir une claire vision à la fois des acquis de l'informatique et des "cadres préexistants" chez l'élève, des "précurseurs" dont parle J. ROGALSKI.

Cet article voudrait être une contribution en ce sens.

2. L'EVOLUTION DE LA PROGRAMMATION

2.1 Programmer c'est écrire un texte

Pendant longtemps, ce sont les aspects syntaxiques qui ont prédominé. On était en présence de nouvelles langues, et on se préoccupait donc de leur grammaire. Comme le dit J. ARSAC, il y avait là des connaissances bien délimitées à transmettre, ce qu'on sait traditionnellement le mieux faire. Il faut ajouter qu'à cette époque les enseignants de programmation, qui étaient en même temps des chercheurs en informatique, se préoccupaient beaucoup de la traduction des langages et que là aussi l'aspect syntaxique était ce qu'il y avait de plus [illegible] bien constitué de l'informatique a été la théorie des langages sur laquelle le linguiste CHOMSKY a exercé une importante influence, et les compilateurs sont sortis de l'artisanat en étant dirigés par la syntaxe. En 1968 encore ce qui est d'abord apparu dans la définition d'Algol 68 [18], c'était une forme sophistiquée de la définition de la syntaxe, et cela a d'ailleurs beaucoup nui à ce langage.

2.2 Programmer c'est décrire des calculs

On s'est cependant vite rendu compte que le difficile n'était pas d'écrire des programmes syntaxiquement corrects. Et la fin des années soixante a connu une crise du logiciel, de la programmation, des langages. C'est qu'il ne sert à rien de savoir une langue si on connaît mal ce qu'elle exprime. Or la signification d'un programme est quelque chose de plus nouveau qu'on ne l'avait cru au premier abord.

De quoi s'agit-il ? De faire fonctionner un outil qui calcule, à condition de prendre ce terme au sens large : imprimer, tracer des graphiques, acquérir des données, commander un robot, consulter un fichier, c'est encore calculer. Un calcul est une suite de changements d'état de la machine. Mais en fait, un programme ne décrit pas un calcul, car le calcul exécuté dépend de données extérieures. Il exprime plutôt tout *un ensemble de* calculs, en généralisant.

Le problème est donc de disposer d'outils adéquats pour *décrire* un ensemble de calculs : c'est le rôle des langages de programmation - mais aussi pour le *concevoir* et le préciser progressivement. Dès l'époque du langage-machine on s'était rendu compte que le caractère [illegible] d'un langage n'est bien adapté à la conception d'un unique calcul ne l'est plus pour un ensemble de calculs, qui demande de penser simultanément plusieurs possibilités : si le saut conditionnel répondait au besoin de description, il fallait donc un support pour aider à la conception : ce fut d'abord l'organigramme.

Mais ce couple goto - organigramme conduit à une programmation "au ras du sol", où tout se trouve au même niveau, et où un calcul est conçu en une seule étape, en le balayant du début vers la fin. Cela va relativement bien pour de petits programmes ; mais lorsqu'ils ont commencé à grossir est apparue la crise dont je parlais plus haut. Elle a conduit à mettre l'accent sur d'autres outils de description (conditionnelle à deux branches, boucle "tant que") qui existaient déjà depuis plusieurs années, et à faire émerger une "méthode" : la programmation descendante structurée [6, 7, 20].

S'agit-il d'une méthode de conception ? Sans doute pas, car elle ne donne aucun guide pour la décomposition d'un problème. C'est plutôt une méthode pour préciser progressivement la description d'un ensemble de calculs dont le programmeur possède déjà une certaine représentation mentale, et en même temps pour affiner cette représentation. Encore n'aide-t-elle pas vraiment à éviter les difficultés sur l'ordre des instructions, qui naissent de la notion de variable. Et prouver qu'un programme est correct (c'est-à-dire répond à sa spécification) demande dans ce cadre des techniques relativement complexes.

2.3 Programmer c'est définir des fonctions

Si maintenant on ne possède au départ aucune représentation mentale des calculs sur quoi peut-on donc s'appuyer ? On ne dispose plus que du but des calculs à effectuer, autrement dit de l'énoncé du problème que l'on doivent résoudre, de sa *spécification*. D'ailleurs, en fait dans la majorité des cas ce n'est pas le calcul qui est important : c'est son résultat ; peu importe par où on est passé, comment on a procédé pour l'obtenir.

On parvient ainsi à un second modèle de la signification d'un programme : non plus un ensemble de calculs, mais une *fonction*, au sens mathématique du terme, qui fait passer des données aux résultats. Et programmer ce n'est plus alors décrire des calculs (point de vue *impératif, procédural, algorithmique*), mais définir des fonctions (point de vue *déclaratif, définitionnel, fonctionnel*).

A dire vrai, ce point de vue remonte loin dans l'histoire de l'informatique, à Mc Carthy avant 1960, et il s'est incarné dans le langage LISP [14, 15] Mais ce langage, aujourd'hui très utilisé chez les informaticiens, a mis longtemps à percer réellement Peut-être à cause d'une syntaxe sans concession, à cette époque où on s'intéressait d'abord à la syntaxe Sans doute parce que le point de vue fonctionnel rebute beaucoup de gens, parce qu'il nécessite une abstraction à partir de la notion de calcul on sait combien de siècles il a fallu pour que se dégage la notion mathématique de fonction ! Certainement à cause du manque d'applications rendant véritablement utile un tel langage, orienté vers les applications non numériques Mais aussi à cause du caractère prophétique du "système" LISP la notion de programme s'évanouit celle de système d'exploitation disparaît pour faire place à un dialogue où l'utilisateur entre une expression et reçoit son résultat tout en pouvant à volonté enrichir son univers avec de nouvelles fonctions Ce n'était pas une conséquence nécessaire du point de vue fonctionnel, mais c'était révolutionnaire à l'époque du "traitement par lots", où un programme était soumis à un système d'exploitation pour être compilé en plusieurs passes, puis exécuté, et fournir des résultats à son propriétaire le lendemain

Le style fonctionnel de programmation peut cependant être dissocié du langage LISP, qui n'en est d'ailleurs qu'une incarnation imparfaite et qui peut fort bien servir (être perverti ?) avec un style impératif Il s'agit plutôt d'une méthode de conception : considérer un programme comme une fonction, définir une fonction par une expression de son résultat à partir de ses données, cette définition faisant apparaître d'autres fonctions à définir de la même façon On a là encore une programmation descendante, mais elle n'est plus liée à une représentation mentale des calculs Une tentative d'emploi de cette méthode, orientée vers la description dans un langage algorithmique, est la programmation "déductive" [8]

2.4 Programmer c'est définir des objets

Reste à savoir comment parvenir à trouver une définition du résultat à partir des données Le seul guide dont on peut disposer ici est la nature ou type des objets que sont données et résultats de sorte qu'il existe deux stratégies de définition se laisser guider par le type du résultat ou par celui d'une donnée Mais il ne s'agit pas a priori d'objets que traite directement le langage de programmation - et LISP moins que tout autre tant les objets qu'il manipule (les arbres) sont élémentaires Il faut donc définir le résultat en utilisant des fonctions qui engendrent les objets de son type, et utiliser les données à travers des fonctions qui portent sur elles Ce sont ces fonctions qui guident la définition Ce sont elles aussi qui apparaissent dans l'analyse descendante ; leur expression dépend de la représentation des types considérés, et elle aide donc à choisir cette représentation, qui doit viser à rendre ces fonctions aussi simples et efficaces que possible [16]

Finalement, de ce point de vue, programmer ce n'est plus décrire des calculs , c'est définir des fonctions, et pour cela des objets (1), qui constituent donc le guide de la conception On peut parler de *programmation orientée par les objets*, et il existe des langages pour la favoriser, comme par exemple SMALLTALK [11] On peut d'ailleurs penser que ces langages se rapprochent par divers aspects d'une logique "courante", et peuvent donc tempérer pour beaucoup d'esprits la dureté d'une programmation fonctionnelle trop teintée d'un style mathématique Mais nous ne disposons pas encore du recul suffisant pour en juger

2.5 Pour conclure sur cet historique

Je n'ai pas parlé d'autres styles de programmation qui peuvent être rattachés à la programmation fonctionnelle La programmation logique où un programme est conçu comme une relation entre données et résultats décrite dans le langage de la logique mathématique son représentant le plus connu est PROLOG [17, 5] La programmation fonctionnelle à la BACKUS [3], où une fonction est définie non pas par une expression de son résultat mais par une composition de fonctions élémentaires ou déjà définies : elle n'a donné lieu jusqu'ici qu'à des réalisations expérimentales

Du trop rapide historique qui précède, je voudrais retenir quelques points :

- il ne suffit pas de savoir exprimer un programme ; il faut apprendre aussi à le concevoir, et posséder pour cela des méthodes ;

(1) D'ailleurs, on sait depuis longtemps que programmer ce n'est pas seulement décrire des calculs, mais aussi des "structures" de données" [21]

- il y a progrès d'une programmation "au ras du sol" à une programmation structurée et descendante, puis à la programmation fonctionnelle et à la programmation orientée par les objets ;

- ce progrès porte à la fois sur la sécurité de l'expression, mais aussi sur le guidage de la conception dans le cas où on ne dispose pas *à priori* d'une représentation mentale des calculs à décrire.

Mais cela ne signifie pas que, dans l'apprentissage, on puisse faire complètement l'économie de la démarche historique et passer trop vite à une méthode, encore moins à ce que l'on considère aujourd'hui comme une "bonne" méthode.

3. PARTIR DU CADRE CONCEPTUEL PREEXISTANT CHEZ L'ELEVE

Bien entendu, ce cadre dépend du niveau d'enseignement considéré. Dans cet article, il s'agit d'abord des élèves de l'option informatique des lycées, mais beaucoup de points restent valables à un niveau supérieur, que j'ai pu davantage observer directement.

3.1 Programmer c'est utiliser un outil

Quoiqu'on ait pu dire, notamment au début de l'introduction de l'informatique au lycée, l'informatique est liée à l'ordinateur. Et c'est ce partenaire qui domine dans l'esprit des élèves : un des intérêts de l'informatique, c'est que l'élève ne se trouve pas dans une situation scolaire classique où il s'agit d'abord de satisfaire le professeur.

L'outil n'est cependant pas limité à l'ordinateur : il comprend aussi le langage qui permet de le commander. Au départ, ce ne peut donc être que ce langage qui motive l'intérêt de l'élève, ce dernier n'est pas *a priori* disponible pour un autre langage encore qui permettrait d'appliquer une méthode de conception dont il n'a aucune idée de l'existence et de la nécessité. Apprendre le langage de programmation qui va être son outil est donc un point de passage obligé.

L'élève a déjà appris à utiliser des outils, sinon dans sa vie scolaire au moins dans la vie courante. Au collège il a appris une ou deux langues étrangères : il a donc une expérience dans ces domaines : c'est que l'on apprend un outil en l'utilisant (1), pour une langue on l'apprend aussi en étant confronté avec des textes, et ensuite en acquérant des règles de grammaire.

Il semble donc que la meilleure façon de commencer soit non pas d'écrire des programmes devant les élèves et surtout pas "en insistant sur les *aspects méthodiques*", mais d'abord d'en faire utiliser, puis d'en faire lire et comprendre.

Ainsi on s'appuie bien sur les *précurseurs* existant dans l'esprit des élèves. Mais en même temps on peut préparer l'avenir. Car la compréhension d'un programme doit laisser une trace, pour être communicable. Les élèves ont l'habitude d'une telle exigence, et le professeur peut donc être plus directif sur cette trace et préparer ainsi la forme des analyses ultérieures.

On peut faire remarquer que dans un langage algorithmique (c'est l'hypothèse retenue ici, nous allons en discuter), il y a trois types d'instructions "efficaces" :

- deux donnent des valeurs à une variable (lecture et affectation)

- la troisième communique un résultat à l'utilisateur par exemple la valeur d'une variable.

Les autres instructions ne sont là que pour décrire comment sont composées ces trois instructions de base. La clé de la compréhension est la signification des variables : d'où l'idée d'en faire un lexique comme le recommande le programme de seconde, et de lier lexique et instructions donnant les valeurs aux variables.

3.2 Un outil qui agit

Aujourd'hui, les élèves qui commencent à programmer ont le plus souvent déjà utilisé un ordinateur. Et ils en ont donc une représentation qui est celle d'une machine qui agit, qui effectue une suite d'actions, et même qui interagit avec son utilisateur.

Parmi les modèles de la notion de programme évoqués au paragraphe 2, le plus adapté est donc celui qui voit un programme comme décrivant des calculs (au sens large indiqué en 2.2). Vouloir se passer de cette notion de calcul, comme le fait la programmation fonctionnelle, s'oppose aux concepts présents chez l'élève.

(1) Remarquons d'ailleurs que c'est en particulier, de plus en plus, l'expérience d'apprentissage des outils informatiques : on ne les apprend pas en lisant d'abord un manuel ou une notice. Macintosh, notamment, est allé très loin en ce sens.

Il est plutôt utile de préciser la représentation de l'outil. L'emploi d'un tableur est une excellente initiation à un certain nombre de notions présentes en programmation. Pour localiser davantage les concepts préexistants sur le langage qui sera utilisé, on peut aussi faire d'abord exécuter des instructions en mode direct, c'est-à-dire en utilisant l'ordinateur comme une machine classique.

On voit donc la supériorité pour cet aspect des langages interprétés et surtout de langages comme LISP et LOGO où il s'agit en quelque sorte du fonctionnement de base, la programmation proprement dite n'intervenant ensuite que pour enrichir l'univers de fonctions ou d'actions. Mais LISP n'est pas d'abord fait pour décrire des calculs et il suppose plutôt que la notion mathématique de fonction qui n'est pas un concept bien établi chez la plupart des élèves, en seconde certainement et même à des niveaux ultérieurs. LOGO est bien un langage impératif mais dès qu'on s'éloigne des applications les plus simples sa proximité de LISP en fait un outil un peu bâtard.

3.3 Programmer c'est faire faire un calcul

Faire exécuter une suite d'actions n'est pas nouveau pour l'élève ; pour cela, il donne ses instructions dans l'ordre où elles doivent être exécutées. L'exemple bien connu de la recette de cuisine répond à ce schéma de pensée. La "méthode" qui lui correspond consiste à imaginer un humain résolvant le problème et à transcrire sa démarche dans le langage utilisé.

Cette méthode est loin d'être inutile et on peut la prendre comme base dans beaucoup d'applications, en sachant cependant qu'elle est insuffisante pour une raison très simple : c'est qu'un programme ne décrit pas un calcul, mais un ensemble de calculs, et même le plus souvent un ensemble infini.

Or pour décrire un ensemble de calculs, on ne peut se contenter d'une vue séquentielle. On a besoin de décrire plusieurs voies. La proposition conditionnelle est un moyen linguistique pour cela, mais il oblige à rompre la pensée séquentielle. Aussi la forme la plus spontanément utilisée est la conditionnelle à une seule voie

si ... alors

qui rompt la séquence aussi peu que possible : elle est la plus spontanément utilisée dans le langage courant, mais aussi chez les programmateurs débutants.

Si on veut aller plus loin et utiliser des conditionnelles à deux voies (*si ... alors ... sinon ...*) ou davantage (cas), la difficulté de pensée est plus grande : il faut laisser une branche en suspens pendant que l'on décrit l'autre, et savoir la reprendre ensuite. Ce n'est facile ni dans la pensée, ni dans le langage. Les langages de programmation les plus rudimentaires s'en tirent par le saut, mais ce n'est pas un outil de pensée, et il faut donc distinguer outil de conception et langage de programmation. Le premier outil de conception permettant suspension et reprise est l'organigramme, qui est très éloigné de la pensée séquentielle. On se rapproche davantage d'une expression séquentielle en *nommant* les deux branches pour pouvoir les reprendre ensuite. Cette idée conduit à une expression par niveau, et elle répond à un concept présent dans l'esprit des élèves : c'est celui de *plan*, utile aussi d'ailleurs pour concevoir un texte séquentiel long, comme celui d'une composition française. Ici chaque partie du plan correspond à des actions donner une valeur à des variables, écrire des résultats, il est bon que ces actions, et notamment les variables affectées, apparaissent clairement dans sa désignation. Cela permet de les détailler ensuite, cela permet aussi une vérification : qu'aucune variable ne soit utilisée avant d'avoir reçu sa valeur.

Les structures conditionnelles sont des outils pour décrire un ensemble de calculs, mais seulement un ensemble fini. Pour passer à l'infini on a besoin d'une autre structure et l'expérience courante fournit la répétition. Il semble bien que la structure répétitive en elle-même présente moins de difficulté que la conditionnelle, car elle ne rompt que peu la pensée séquentielle : sous sa forme la plus spontanée elle s'exprime par

si recommencer (une opération) (et cela qui suivent)

ce qui se précise bien par un *tant que* conditionnel. Mais on sait - et nous verrons plus loin pourquoi - qu'il vaut mieux délimiter plus clairement les itérations. Les exemples proposés en début d'apprentissage permettent d'introduire les formes souhaitées, qui sont facilement acceptées.

- *répéter ... jusqu'à* qui est très proche du *si recommencer...* (mais *tant que... faire...* est beaucoup moins naturel : cette forme, qui ne se justifie que par des considérations d'économie de langage pour une répétition qui pourrait ne pas avoir lieu, devrait être laissée pour plus tard)
- *répéter 36 fois... fin*
- *pour i de p à n faire... fin* qui a

le mérite d'introduire un élément variable dans l'itération.

3.4 Les variables informatiques, un concept sans véritable précurseur

Il n'est en effet pas intéressant d'effectuer une répétition à l'identique : un changement se produit donc d'une étape à l'autre et ce changement porte en général sur la valeur de variables (il y a d'autres possibilités en LOGO grâce à la tortue). On rencontre là un concept : celui d'un nom dont la valeur varie au cours du temps, qui n'a pas de précurseur favorable dans l'expérience de l'élève : on ne connaît pas d'objet qui ne disparaît pas quand on en crée un autre chez lui. La variable mathématique qui ne désigne pas à proprement parler une valeur, mais qui est mise pour n'importe quelle valeur d'un ensemble ou plus précisément à laquelle on peut substituer n'importe quelle valeur et même n'importe quelle expression. Ces précurseurs induisent donc en erreur : ils sont "réducteurs" et non pas "producteurs", selon l'expression employée par J. ROGALSKI.

La notion extérieure à l'informatique qui se rapproche le plus de celle de variable est sans doute la notion mathématique de suite (en fait, la suite des valeurs de la variable). Mais ce n'est pas un précurseur parfait, car dans une suite un élément u_n ne disparaît pas lorsque l'on envisage le suivant u_{n+1}.

Surtout, elle ne semble pas suffisamment connue des élèves de seconde, voire de première, pour servir de base, à ce niveau, c'est peut-être plutôt la variable qui peut servir de précurseur à la notion de suite (et notamment de suite définie par récurrence).

Tout cela explique pourquoi cette notion de variable est celle qui fait faire le plus d'erreurs en programmation. Et c'est le principal avantage de la programmation fonctionnelle que de s'en débarrasser pour se limiter à la variable mathématique, en particulier comme argument "formel" pour décrire une fonction. Mais, comme nous l'avons déjà dit, la programmation fonctionnelle heurte le cadre conceptuel de la plupart des débutants.

Il faut donc plutôt identifier les difficultés que crée la notion de variable informatique et les traiter systématiquement. La variable *i* de boucle de *pour i de 1 à n* n'en soulève pas et c'est pourquoi cette forme est intéressante pour commencer. Dans le cas général il s'en présente deux : la coexistence de deux valeurs de la variable et l'initialisation.

La coexistence de deux valeurs de la variable se rencontre normalement à l'intérieur d'une itération : elle pose un problème sur l'ordre des instructions. Ce problème est difficile parce qu'il n'apparaît pas dans la représentation spontanée du calcul, puisque la notion de variable informatique n'y admet pas de véritable contre-partie. Dans la représentation spontanée, on envisage en effet deux valeurs simultanément, sans que la seconde empêche l'accès à la première ; et comme de plus on leur donne le même nom, on a tendance à les confondre. L'expérience suivante le montre : dans un fichier sont rangées les populations des communes de France, département après département, sous la forme d'un article (département, commune, population) ; on veut trouver la population des 95 départements. Devant un tel problème, les élèves écrivent en majorité quelque chose comme

pour i de 1 à 95 faire
s := 0
répéter
lire a
s := s + population de a
jusqu'à département de a ≠ i ou fin de fichier
...

où *a* désigne en fait deux valeurs successives qui se confondent dans l'esprit du programmeur. [19] propose à une méthode liée à ce genre de difficulté.

Je vois à ce problème — qui ne devrait pas se présenter trop tôt dans l'apprentissage — deux solutions. La première consiste à présenter des schémas de programme [12] qu'on peut évoquer au niveau de l'analyse du style : on sait les coder systématiquement. Dans l'exemple précédent :

s := somme des populations de a
pour a dans fichier jusqu'à (département de a ≠ i ou fin de fichier) exclu

L'emploi de tels schémas a d'autre part l'avantage de permettre de retrouver une pensée séquentielle lors de l'analyse. Mais si un petit nombre de schémas permet de couvrir un grand nombre d'exemples, il en échappera toujours — et il ne reste donc qu'à espérer que les schémas présentés auront induit chez l'élève une capacité suffisante de généralisation.

La seconde solution consiste à attirer l'attention des élèves sur la difficulté et, pour cela, à noter de façon distincte dans l'analyse les deux valeurs successives d'une variable *a*, par exemple *a* et *a'* : la valeur *a'* est transmise par l'étape précédente, mais *a* doit recevoir une valeur dans l'itération, et *a*

devient alors inaccessible. On doit donc se poser la question, avant d'écrire une variable, de savoir de laquelle des deux valeurs il s'agit. Dans l'exemple, cela conduit après quelques tâtonnements, à

répéter
s := « population de a »
Lire a
Jusqu'à département de a ≠ l ou fin de fichier.

Cette solution exige que, lorsque l'on est en train d'écrire une itération, on en soit bien conscient : c'est donc une justification de la délimitation des itérations par un début et une fin (ce que n'assure pas l'emploi d'un saut conditionnel).

L'initialisation crée aussi beaucoup d'erreurs car elle est en général sous-entendue dans la représentation spontanée du calcul. Ce problème est également traité dans chacune des solutions précédentes : les schémas contiennent l'initialisation. La seconde proposition implique que soit développé le réflexe d'initialisation dès que l'on utilise une valeur "ancienne" notée *a*.

3.5 Programmer, c'est concevoir un algorithme

Nous en sommes jusqu'ici restés au point de vue exprimé au début du 3.3 : imaginer un homme résolvant le problème posé, et décrire sa démarche. Nous avons seulement attiré l'attention sur les difficultés de cette description, et envisagé des moyens pour les résoudre : faire un plan pour ne pas entrer trop vite dans les détails et pouvoir ainsi conserver, à chaque niveau de l'analyse, une démarche quasi-séquentielle : ce n'est pas autre chose que la programmation descendante structurée ; enrichir cette méthode en prenant garde à l'ordre des affectations des variables, notamment pour celles qui varient dans les itérations.

Il me semble que la méthode ainsi obtenue, qui ne sort pas de la programmation impérative, est suffisante pour la plus grande partie de la programmation non professionnelle. Et donc qu'on pourrait s'arrêter là au lycée, sauf à apporter une ouverture pour montrer qu'il y a des cas où il ne suffit pas d'exprimer à partir d'une représentation mentale, mais qu'il faut vraiment concevoir, c'est-à-dire découvrir un algorithme. La rencontre d'un tel cas, où on ne sait pas vraiment démarrer, provoque un choc chez les élèves, car elle remet en question l'idée qu'ils se faisaient jusque là de la programmation. Elle souligne d'ailleurs une différence de taille entre apprentissage d'une langue étrangère et celui de la programmation : lorsque l'on apprend une langue on ne se pose pas en général simultanément le problème de concevoir ce que l'on a à dire.

Cependant on peut encore distinguer entre les problèmes qui tombent dans cette catégorie. Les plus simples sont ceux où la difficulté porte sur les objets en jeu, parce qu'ils sont éloignés d'un langage de programmation : par exemple, on veut faire du calcul algébrique formel, et les objets sont des formules algébriques. Ici, le blocage vient du fait que l'on n'ose pas exprimer l'algorithme sur de tels objets, craignant de ne pouvoir ensuite arriver à un programme. Alors, on ne parvient pas à démarrer, ou bien on choisit *a priori* une représentation des objets qui ne permet pas ensuite de trouver un algorithme (par exemple on représente les formules par des chaînes de caractères). Pour lever ce blocage, il faut, dans un premier temps, exprimer l'algorithme sur les objets du problème (curieusement, on parle alors de types "abstraits" [13, 9] et de programmation "abstraite") ; cela introduit des fonctions et des procédures, et on choisit alors une représentation des objets qui favorise l'expression de ces fonctions et procédures. Pour une formule par exemple, on s'apercevra que l'on a besoin de connaître son opérateur et ses deux opérandes, ce qui conduira à la représenter par ces trois éléments (et comme les opérandes sont en général à leur tour des formules, on obtiendra finalement une représentation en arbre) et très naturellement une définition *récursive* des fonctions.

Mais encore faut-il au départ pouvoir trouver l'algorithme sur les objets qui figurent dans l'énoncé du problème. Le cas le plus difficile est celui où même sur ces objets "non informatiques" on ne possède aucune représentation mentale des calculs à effectuer. C'est dans ce cas que la programmation déclarative devient essentielle. Si on ne sait pas comment le *calculer*, on va *définir* le résultat attendu : cela introduira des intermédiaires et notamment des fonctions, qui seront définies à leur tour, jusqu'à parvenir à des fonctions déjà connues ou en cours de définition (*récursivité*). Et pour guider la définition d'une fonction, on dispose du type des données et du type du résultat, et notamment des opérations qui les engendrent [16].

Prenons par exemple le problème bien connu des Tours de Hanoï. C'est un problème où, sauf à le connaître déjà, on ne possède pas d'idée sur l'algorithme. La donnée est un entier *n*, le résultat une suite de déplacements d'une pièce.

Partons du type de la donnée l'entier *n* est soit *0* (et aucun déplacement n'est à effectuer), soit le successeur de l'entier *n-1* ; une idée simple consiste donc à ramener (récursivement) le problème pour *n* pièces au même problème pour *n-1* pièces. Avec cette idée, il est facile de trouver une solution, même si on doit encore remarquer qu'en réalité on ne retrouve pas exactement le même problème, puisque les piquets de départ et d'arrivée ont changé : il faut donc généraliser le problème de départ en introduisant ces piquets dans les données.

4. CONCLUSION

Comme **Jacques ARSAC**, je pense qu'on doit viser à ce que la programmation développe les aptitudes de rigueur, de clarté d'expression, de précision, d'organisation, de créativité aussi, mais je ne suis pas certain qu'une véritable invention d'algorithmes soit fréquente ni surtout réalisable : je serai donc ici plus circonspect. On pourrait sans doute ajouter des acquisitions plus liées au développement logique, comme les structures d'ordre et de composition d'actions, la désignation, certains raisonnements.

Pour atteindre ces buts, fixons-nous comme objectifs de rendre l'élève

- capable d'écrire un programme lorsqu'il dispose au départ d'une représentation mentale de l'algorithme correspondant (qu'il a obtenue soit en imaginant une réalisation humaine, soit parce qu'on la lui a communiquée dans une autre discipline) ;
- capable de communiquer ce programme à un être humain, en particulier à l'aide d'un dossier explicatif ;
- conscient qu'il existe des cas où l'algorithme doit être complètement découvert, et capable d'expliquer sur quelques exemples les principes sous-jacents.

Voilà, avec beaucoup de prudence sur le dernier, ce que pourraient être à mon sens les objectifs de la programmation dans l'option des lycées. Les considérations développées dans cet article permettent alors de tracer une démarche. Je la décomposerai en trois parties, sans que cela signifie nécessairement qu'elles correspondent respectivement aux trois classes du lycée, mais il faudrait y réfléchir. La première partie est axée sur la connaissance du langage ; la seconde fait davantage place aux méthodes d'expression, de description d'ensembles de calculs ; la dernière ouvre à la conception d'algorithmes.

1) On ne peut construire que sur une connaissance de l'outil, c'est-à-dire essentiellement du langage réellement utilisé (et non pas d'un langage idéal, mythique ou œcuménique). Il faut donc commencer par là. L'apprentissage du langage devrait partir d'exemples de programmes couvrant un sous-ensemble qui permette des applications variées (par exemple lecture et écriture simples, affectation, conditionnelle, une forme d'itération, procédure sans paramètre, peut-être tableaux) ; il se poursuivrait par la construction par les élèves de programmes utilisant les mêmes notions, puis par une première synthèse sur les constructions rencontrées, aboutissant à un manuel d'utilisation. On pourrait ensuite étendre le sous-ensemble, pour aboutir à un projet, c'est-à-dire à un travail où l'élève a la totale maîtrise de la réalisation : pour que ce soit possible à ce niveau, il faut donc que le sujet soit suffisamment modeste. Le projet répond à la nécessité d'agir de manière autonome et de commettre des erreurs pour apprendre.

Cette première partie ne fait qu'une part réduite à la méthodologie. La méthode consiste seulement à partir de ce que ferait un être humain. Mais la compréhension des exemples de départ laisse une trace écrite : lexique, analyse, spécification de ce que réalise le programme (données et résultats), la forme recommandée dans les commentaires pour la classe de seconde me paraît ici bien adaptée, car susceptible d'évoluer au fur et à mesure des besoins. Puis, lorsque l'élève écrit un programme il fournit avec lui un document semblable, mais comme il s'agit maintenant d'un "thème" et non plus d'une "version", il procède en sens inverse : spécification, puis l'analyse et le lexique, enfin programme et jeu d'essais. Pour maîtriser le projet, il est conduit à faire un plan, qui peut se programmer à l'aide de procédures, et on s'avance ainsi vers la programmation structurée. De plus, un projet révèle l'ampleur des tâches de programmation, fait constater les erreurs et peut donc motiver les élèves pour aller plus loin.

2) La seconde partie doit confirmer que les choses sont moins simples qu'on aurait pu le croire. De nouvelles applications, portant notamment sur les itérations, feront comprendre qu'on ne peut se fier complètement à une transposition de ce que ferait un homme. Il faut donc affiner ses méthodes : programmation structurée, enrichie, comme nous l'avons sug-

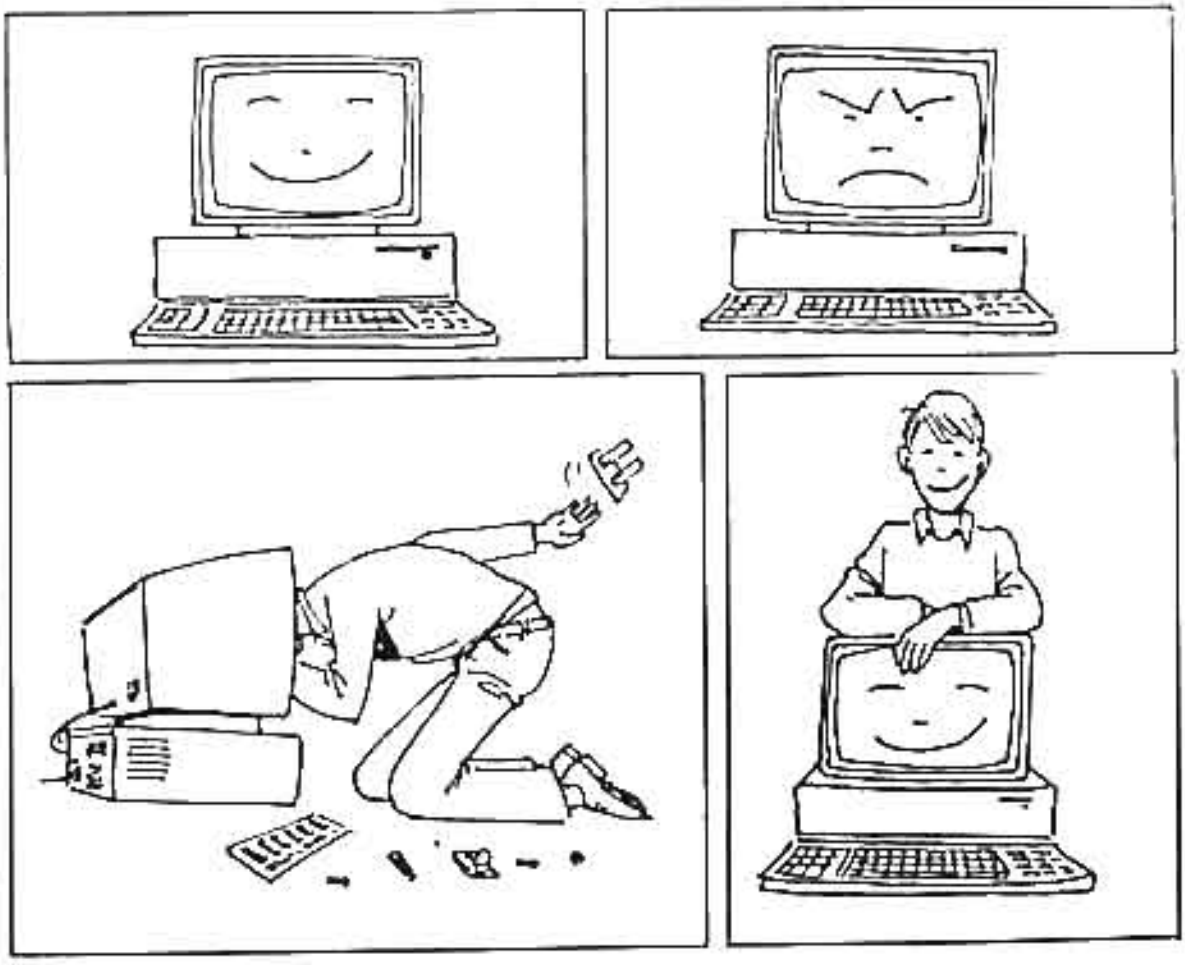
géré en 3.4, par des schémas usuels et une technique d'analyse des itérations où des variables évoluent ; choix entre variable simple et tableau pour traiter des valeurs successives.

3) L'ouverture sur les méthodes de conception traite essentiellement des exemples pour lesquels il faut vraiment découvrir l'algorithme. Sur eux, elle montrera quelques techniques comme

- donner des définitions à partir du résultat (ce qui dans les documents d'analyse utilisés, veut dire que les diverses lignes sont d'abord écrites dans l'ordre où se présentent les définitions, et plus tard ordonnées ;
- définir récursivement des fonctions et des procédures ;
- utiliser une programmation "abstraite", c'est-à-dire programmer avec les objets du problème avant de choisir leur représentation dans le langage.

Finalement, dans cette démarche, toutes les formes de pédagogie dégagées par J. ARSAC ont leur place, mais pas au même moment. En particulier, il ne faut pas se précipiter trop vite sur les méthodes, surtout si l'on entend par là les méthodes de découverte d'algorithmes. Une méthode c'est d'ailleurs, en tout cas pour les non professionnels, quelque chose que chacun doit se construire par lui même. Alors, il faut s'inspirer de celles qui existent, en retenir ce qui paraît bon à chaque stade, plutôt que de les suivre aveuglément... n'en déplaise à mon ami Amédée DUCRIN à qui j'ai enseigné de 1973 à 1981 : ce n'est qu'ensuite que j'ai compris que je ne savais (toujours) pas enseigner la programmation. Parfois, je me dis cependant que je pourrais (enfin) aider les élèves et les étudiants à l'apprendre.

Claude PAIR
Centre de recherche en informatique de Nancy
B.P. 239
54506 Vandoeuvre Cedex



REFERENCES

1 - **Pratique et savoir-faire des élèves de l'option informatique** (1987). Ministère de l'Education Nationale, Direction des Lycées et Collèges

ARSAC, J. - Des pédagogies pour l'option informatique des lycées. 1-13

ROGALSKI, J - Acquisition de savoirs et savoir-faire. 14-22

Voir aussi les articles de G. AMAND (objectifs et évaluation), M. GAUDRAT (Pascal, une économie informatique), N. WAITER et M. SOUFFRON (classe de seconde), R. FAVRE-NICOLIN et A. HIRLIMANN ainsi que A. AUTHIER, F. GAYDIER, M. HELALI (projets), R. BENETOLLO (Logo, Prolog, Lisp), R. FAVRE-NICOLIN et J.B. MARANINCHI (tableurs)

2 - **ANTONMATTEI, P.** - Introduction d'un enseignement optionnel complémentaire d'informatique en classes de seconde, première, terminale. *B.O.E.N.* n° 25, 20 juin 1985

3 - **BACKUS, J.** (1978) - Can Programming be liberated from the Von Neumann Style ? A Functional Style and its Algebra of Programs. *Comm. ACM* 21, 613-641

4 - **CHOMSKY, N.** (1957) - *Syntactic Structures*. Mouton, La Haye - Paris

5 - **COLMERAUER, A., KANOUI, H., VAN CANEGHEM, M.** (1983) - Prolog, bases théoriques et développements actuels. *Technique et Science Informatiques*, 2, 271-311

6 - **DAHL, O.J., DIJSKTRA, E.W., HOARE, C.A.R.** (1972) - *Structured Programming*. Academic Press, Londres, New York

7 - **DIJKSTRA, E.W.** (1976) - *A Discipline of Programming*. Prentice Hall, Englewood Cliffs

8 - **DUCRIN, A.** 1984 - *Hyperprogrammation*. Dunod, Paris

9 - **GUTTAG, J.V., HORNING, J.J.** (1978) - The Algebraic Specification of Abstract Data Types. *Acta Informatica* 10, 27-52

10 - **HOC, J.M.** (1982) - L'étude psychologique de l'activité de programmation : une revue de la question. *Technique et Science Informatiques*, 1, 383-392

11 - **KAY, A., GOLDBERG, A.** (1976) - *Smalltalk 72, Instruction Manual*. Xerox Research Center, Palo Alto

12 - **LUCAS, M., PEYRIN, J.P., SCHOLL, P.C.** (1983) - *Algorithmique et représentation des données*. Masson, Paris

13 - **LISKOV, B.H., ZILLES, S.N.** (1974) - Programming with Abstract Data Types. *SIGPLAN Notices*, 9, 50-59

14 - **Mc CARTHY, J.** (1978) - History of Lisp. *SIGPLAN Notices*, 13, 8, 217-223

15 - **Mc CARTHY, J., ABRAHAMS, P.W., EDWARDS, D.J., HART, T.P., LEVIN, M.I.** (1962) - *Lisp 1.5 Programmer's Manual*. MIT Press, Cambridge Mass.

16 - **PAIR, C., MOHR, R., SCHOTT, R.** (à paraître) - *Construire des algorithmes*. Dunod, Paris

17 - **ROUSSEL, P.** (1975) - *Prolog, manuel de référence et d'utilisation*. Université d'Aix-Marseille, Groupe d'intelligence Artificielle

18 - **A. VAN WIJNGAARDEN** et coll. (1969) - *Report on the algorithmic language Algol 68*. Springer, Berlin. Traduction française par le groupe Algol de l'AFCET, Hermann, Paris (1972)

19 - **J.D. WARNIER, FLANAGAN, B.M.** (1972) - *Entraînement à la programmation*. Editions d'Organisation, Paris

20 - **WIRTH, N.** (1973) - *Systematic Programming : an Introduction*, Prentice Hall, Englewood Cliffs. Traduction française, Masson, Paris (1977)

21 - **WIRTH, N.** (1976) - *Algorithms + Data Structures = Programs*, Prentice Hall, Englewood Cliffs.

MANIFESTE
CONTRE LA LOGIQUE INFORMATIQUE
(POUR L'ILLOGISME PNEUMATIQUE)

– 1 –

L'important n'est pas le besoin de l'homme mais le bien-être de la machine (et son fonctionnement optimal).

– 2 –

Un bon système d'information doit refléter avant toute chose la technologie de l'ordinateur employé ; il sera démantelé et reconstitué fondamentalement à chaque nouvelle version matérielle et logicielle optée par le constructeur ou le prestataire.

– 3 –

Il faut d'abord réaliser et faire. Il faut réfléchir et modéliser plus tard. On perd trop de temps à discuter et à consulter, à conceptualiser, à établir les objectifs et la pertinence.

– 4 –

Une démarche hiérarchique d'analyse méthodique serait à la rigueur admissible, si l'on partait du bas pour aller vers le haut, si on se préoccupait de l'infini détail avant de se préoccuper de l'objectif global.
Référentiel et finalités doivent conserver secret et mystère qui en font la saveur et l'authenticité.

– 5 –

Quoi de plus absurde que de rechercher les résultats que l'on veut obtenir, et les données que l'on veut stocker pour construire un système d'information, voire les décisions à prendre.
On doit tout d'abord prévoir les "*WRITE*", les "*SEND*", les "*READ*" et les "*RECEIVE*" qui nous garantiront un bon fonctionnement, en fonction de l'architecture des processeurs de la machine utilisée.

– 6 –

On demande trop souvent aux utilisateurs et aux gestionnaires leur avis.
Le rôle de ceux-ci est d'abord (pourtant !) de s'adapter aux produits décidés et offerts par les techniciens, technocrates, commerciaux et financiers pour leur plus grand bonheur obligatoire (et ne pas se mêler de ce qui ne les regarde pas).

– 7 –

L'informatique est nécessaire partout et toujours sur le globe, pour toutes les utilisations, pourvu que cela permette d'épuiser les trop-pleins de stocks des sociétés nationales et internationales.
Les pays du tiers-monde et les foyers doivent se convaincre de leurs nouveaux besoins dont ils ne sont pas conscients. Il est urgent que le marché éponge l'offre, comme de naturel.

– 8 –

C'est la machine électronique qui doit prendre les décisions, l'homme doit les exécuter.
Cela afin d'être compétitifs et optimums (face aux extra-terrestres).

– 9 –

Plus un système d'information et de décision est compliqué (et incompréhensible), moins il est contestable et donc contesté !

– 10 –

La complexité non maîtrisable est l'empreinte du vrai talent, voire du génie. Peu la méritent vraiment.

– 11 –

A chacun son système d'information. A chacun sa méthode informatique. Interfaces, passerelles et traducteurs feront le reste. Et dieu reconnaîtra les siens ...

– 12 –

La logique, point trop n'en faut. Rien ne vaut l'inspiration du moment et l'humeur météorologique.
Le résultat doit toujours être un sujet permanent d'étonnement, de mauvaise humeur créatrice ou de de joie naïve et surprise.

Alphonse LEUNULTON
(alias Pierre FISCHOF)

ADELI

Spécial 1993

Avis aux entreprises françaises

Quelques recettes infaillibles pour gagner

Il ne se passe pas une conférence, un article, sans que l'on agite devnnt les entrepreneurs frnnçais médusés, l'épouvantail de In transition 1992 - 1993 (le marché unique européen).

Pour exorciser cette frayeur, les gourous brandissent les amulettes du tiercé magique :

Productivité Qualité Compétitivité.

L'ADELI va plus loin et dévoile, à votre profit, les recettes éternelles du succès quotidien.

Résistez impérativement à toute tentation de répondre à un courrier.

Vos partennires vous tendent des pièges sournois, en sollicitant, à tout propos, un écrit sous votre signature.
Prenez garde, ceci pourrait être considéré comme la trace indélébite d'une décision, dont vous supporteriez seul les conséquences.

Attention à cette provocation.

Tout courrier reçu n'engage que celui qui vous l'a adressé.

Tout courrier négligé finira bien par s'enliser dans un désordre, soigneusement entretenu. Votre bureau surchargé impressionnera vos visiteurs en soulignant l'image flatteuse d'un cadre très occupé.

Face à votre inertie, votre interlocuteur sera amené à réitérer sa demande écrite, avec une animosité croissante que vous pourrez toujours lui reprocher ultérieurement.

Privilégiez la convivialité de l'expression orale par rapport à la rigidité de l'écrit.

Sachez exploiter votre téléphone

N'hésitez pas à harceler l'assistante de votre correspondant par l'intermédiaire de votre assistante.

Si par miracle vous réussissez à joindre votre correspondant, au moment le moins opportun pour lui, noyez l'objet de votre appel en évoquant pêle-mêle :

- vos difficultés à le joindre,
- vos griefs envers des tiers étrangers à l'affaire,
- vos appréciations du comportement de vos collègues,
- vos commentaires sur la décadence de l'entreprise.

Toute conversation inférieure à 20 minutes n'est pas digne de votre appel et libérerait deux temps (le vôtre et le sien) dont vous ne sauriez que faire.

Si vous ne parvenez pas à joindre votre correspondant, dictez à des intermédiaires de circonstance, des messages impératifs et mystérieux, qu'ils déformeront impunément.

Sachez aménager l'espace de convivialité des réunions

Horaires

Une réunion commence quand vous arrivez et se termine quand vous partez. Elle s'interrompt pendant que vous passez vos coups de fil.

La réunion est destinée à recueillir votre expression individuelle

Evitez de prêter la moindre attention aux propos de vos interlocuteurs; ceci vous empêcherait de préparer les belles phrases que vous devez prononcer,celles qui ouvrent des perspectives dans des domaines étrangers à l'ordre du jour étriqué que certains avaient peut-être frileusement délimité.

Ne rédigez jamais de comptes rendus

Il serait stupide de réduire la richesse des échanges verbaux en quelques phrases froides, qui, de plus, laisseraient supposer le moindre accord entre les différents participants.

Parlez la langue de bois

Pratiquez cet art subtil que maîtrise la hiérarchie qui grimpe; sachez l'imiter pour suivre la même trajectoire de carrière. L'ADELIEN vous proposera, dans ses prochains numéros, quelques règles élémentaires.

Obligez vos Fournisseurs à connaître vos circuits administratifs.

Il appartient au Fournisseur qui a pris la responsabilité de travailler avec vous, et à lui seul, à ses propres frais, d'explorer les méandres de vos circuits administratifs.

Sanctionnez impitoyablement toute erreur de vos fournisseurs

- Si la facture ne comporte pas le n° de commande que vous avez attribué a posteriori,

- si la facture est adressée au bénéficiaire du service, seul connu du Fournisseur, alors que vous disposez d'un organisme payeur, dont il ignore jusqu'à l'existence.

bloquez et attendez.

N'avertissez jamais votre Fournisseur de ses maladresses, il serait capable d'y remédier rapidement ce qui enrayerait tout votre système répressif.

Eprouvez la souplesse de votre Fournisseur

N'hésitez pas à lui demander de reporter la facturation des prestations de l'année n à l'année n + 1, ou vice-versa, d'anticiper la facturation des prestations de l'année n à l'année n - 1.

N'hésitez pas à lui demander de masquer la facturation de certaines prestations sous des libellés fantaisistes, conformes aux exigences formelles de votre Contrôle Budgétaire.

Ne formulez jamais vos caprices a priori, laissez votre Fournisseur établir et envoyez sa pièce, attendez quelques mois avant de lui faire part de vos suggestions (par téléphone bien sûr).

Conduisez votre Fournisseur, de concession en concession, au delà des limites de la légalité. N'ayez aucune inquiétude c'est lui et lui seul qui s'expliquera avec le fisc !

ADELI

Faites de la trésorerie avec vos petites dettes.

La bonne exécution d'une prestation n'est pas une condition suffisante pour en effectuer le règlement.

En retardant le règlement des petites sommes vous ferez fructifier votre trésorerie (taux de base Banque de France + n %).

Ces intérêts perçus compenseront bien la moitié des dépenses (recherches, communications téléphoniques, demandes de justificatifs) qu'entraînera chez vous cette inertie.

Consolez-vous en pensant que les pertes du Fournisseur seront encore plus fortes que les vôtres.

En Conclusion

Vous n'aurez aucun mal à implanter dans vos entreprises ces quelques règles simples qui correspondent à une culture naturelle, qu'il serait criminel de vouloir contrarier.

Vous gagnerez en efficacité en favorisant une information qui se diffusera par ses canaux naturels dans l'entreprise : les couloirs, les toilettes.

Vous offrirez à chaque individu la fierté de jouer un rôle de vecteur actif d'information, en lui accordant la liberté d'ajouter tout commentaire de son cru à l'information qu'il colporte.

Vos Fournisseurs seront d'autant plus enclins à vous fournir un service de qualité qu'ils seront englués dans les tâches administratives que vous leur imposez.

N'écoutez pas le chant mélodieux des sirènes qui vous susurrent :

- de communiquer par écrit, en vous obligeant à vous engager en exprimant vos convictions, (comme si vous aviez des convictions !)
- de régler personnellement les petites dépenses et de vous faire rembourser par note de frais au mépris des règles formelles de la comptabilité analytique,
- de respecter les autres.

Suivez nos conseils, vous nous en donnerez des nouvelles en 1993 et au-delà.

Remerciements

L'ADELI tient à remercier, sincèrement et particulièrement, les entreprises suivantes :

A.F.P.A.	Pessac
Automobiles Citroën	Neuilly sur Seine
Banque WORMS	Paris La Défense
C.E.S.I. Ile de France	Gentilly
Eaton	Cluses
Etat Major Armée de Terre	Paris
Electricité de France	G.R.P.T./ Région Parisienne
France Télécom	Paris
H.E.C./I.S.A.	Jouy en Josas
I.B.S.I.	Francheville
La Poste	Paris
Orsys	Paris
Sofresid	Montreuil

qui lui ont fourni, grâcieusement, les éléments informatifs de cet article que nous aurions été incapables d'imaginer, seuls.

Alain COULON
juin 1989

Les composantes intellectuelles de l'informatique

Extrait de " l'Empire Mathématique" de Davis et Hersh (Gauthier Villars 1988). Un excellent livre d'un excellent rapport qualité/prix.
Nous remercions les Editions Bordas pour l'autorisation de reproduction pour "La lettre de l'Adélé".

DRESSER DES LISTES, autres que des listes d'articles alimentaires ou de nombres premiers, peut devenir une occupation dangereuse, tournant à la manie. Finalement, une liste n'est jamais qu'un résumé, un condensé en quelques lignes d'une situation qui peut être excessivement complexe. Mais quand il s'agit de listes qui visent à expliquer un type d'activités donné en fonction d'un autre type d'activités, le terrain devient glissant... Certains spécialistes mettent en garde contre une telle démarche, considérant qu'elle consiste à poser une question dépourvue de signification pour essayer ensuite de la résoudre. Peut-on en quelques lignes résumer des pensées et des activités qui couvrent des centaines d'années et concernent des millions de gens ? Cela n'empêche que nous aimons à établir et lire de telles listes.

Citons par exemple celle dressée par le professeur Wong Chu-Ming, de l'université de Rennes.

La Renaissance, l'essor de la science et de la technologie, la Révolution industrielle qui en fut la conséquence, sont des phénomènes apparus en Occident et non en Orient. C'est là un fait historique. L'Orient n'ignorait ni la technologie ni les mathématiques ; à maints égards, il était même en avance sur l'Occident. Mais cette explosion se produisit bel et bien en Europe. Pourquoi ?

4. La pensée rationnelle ;
5. La considération de la nature comme objective ;
6. Une vision mécaniste de la nature ;
7. Des recherches expérimentales ;
8. La création de modèles mathématiques :
   a) exactitude qualitative,
   b) écriture fonctionnelle,
   c) variables abstraites.

Pour rendre justice à cet auteur (aussi bien qu'à Needham et à Wong), je dois reconnaître que j'ai repris ces points dans leur sécheresse, sans rapporter tous les commentaires et réflexions qui les accompagnent.

Face à une liste de ce type, j'aime à tâcher de me représenter ce qu'aurait produit l'absence de tel ou tel de ces éléments. Imaginons par exemple ce que seraient les Etats-Unis si la plupart de nos concitoyens n'avaient pas le goût du travail, si notre management était entièrement paralysé, si nous pensions de manière entièrement superstitieuse ou mystique, si nous nous fiions aux autorités établies au lieu de chercher par nous-mêmes et d'expérimenter (tout cela ne tend-il pas, d'ailleurs, à devenir réalité ?). A quoi ressemblerait notre vie quotidienne si nous concevions l'univers comme un oeuf holistique [2], si nous rattachions tous les phénomènes les uns aux autres selon un certain ordre hiérarchique, si notre conception des processus physiques renvoyait à un Jupiter lançant ses foudres sur des satyres malfaisants ?

La liste de Rapp m'a également amené à me poser une question analogue quant aux mathématiques : quelles sont les composantes intellectuelles sur lesquelles elles reposent ?

Il m'a alors paru que l'esprit mathématique ne cesse de ruminer et de réexaminer ses matériaux de départ, selon certains modes opératoires de prédilection. Il est ami de la quantité, de l'ordre, de la symétrie. Il a le goût de l'abstraction. Il recherche des énoncés bien définis, exprimés dans des langages précis, élémentaires, analytiques, utilisant des symboles admis de tous et des abréviations concises. Il aime rencontrer des énigmes et en trouver la solution. La pensée déductive est son élément naturel. Il cherche des transformations, des recettes standardisées, des liaisons logiques ou structurelles. Il pense en termes de oui ou non, de vrai ou faux. Et même, si l'on en croit Leibniz, il estime effectivement possible de fonder tout l'univers intellectuel sur des 0 et des 1. Il estime avec Parménide que ce qui est logique doit être vrai.

Selon le professeur Wong, voici les raisons du retard pris par la science chinoise par rapport à la science occidentale :

1. La rigidité de la structure sociale, où la classe des fonctionnaires lettrés était la classe dominante [...]
2. Le fait que le langage écrit, de nature essentiellement littéraire et métaphorique, restait très éloigné du parler courant et du langage technique, de sorte que les inventeurs avaient beaucoup de peine à faire connaître leurs découvertes.
3. Le mépris pour le travail manuel, symbolisé par le fait que les lettrés se laissaient pousser les ongles ; et le mépris pour les activités mercantiles, qui réduisait les commerçants à former une des classes sociales les plus basses.
4. Une logique liée à un langage essentiellement concret et non analytique, qui ne permettait pas l'élaboration des raisonnements axiomatiques et déductifs nécessaires au développement scientifique. Les applications pratiques d'un problème apparaissaient plus importantes que son exposé théorique.
5. L'immensité de l'Empire, les guerres et les révolutions dont il souffrit, qui expliquent la difficulté des relations scientifiques et pourquoi certaines avancées importantes furent perdues ou mal transmises [...]
6. Le fait qu'en Occident, l'homme est la mesure de toutes choses, tandis qu'en Chine chaque problème était considéré sous un jour particulier [...]

Wong Chu-Ming, in *Scientific Change*, ouvrage collectif dirigé par A.C. Crombie.

Cette liste se présente comme un résumé d'un long article du professeur Joseph Needham intitulé *«Misères et succès de la tradition scientifique chinoise»* [1]. Needham, qui fait autorité en la matière, considère lui-même que la liste de Wong n'est pas inutile. Cependant, en lisant le commentaire de Needham, on s'aperçoit que si tous les éléments énumérés par Wong y figurent en effet, ils sont agrémentés de tant de "si", de "mais", de "peut-être", de "par ailleurs", que l'on se prend à douter qu'une telle liste ait véritablement un sens. Il n'empêche qu'elle existe et qu'on peut certainement en tirer certains enseignements. Elle ne soutient pas, par exemple, que la révolution technologique se serait produite en Europe parce que les Anglais et les Espagnols avaient ramené d'Amérique la pomme de terre au milieu du XVI' siècle...

Considérons une seconde liste qui n'est pas sans rapport avec la première. Dans un ouvrage très stimulant, F. Rapp essaie d'identifier les exigences et perspectives intellectuelles essentielles à une civilisation technologique. Il en dénombre huit, que voici :

1. La valeur accordée au travail ;
2. Un management efficace :
3. Le penchant à la créativité technologique ;

L'esprit mathématique, confronté à un univers flou et indéterminé, essaie de trouver des énoncés précis concernant des phénomènes chaotiques ou aléatoires. Il s'entend à considérer les choses sur un mode probabiliste ou statistique.

C'est à partir de ces considérations que j'ai dressé ma liste.

Les composantes intellectuelles de la mathématisation.

1. La capacité de symboliser, d'abstraire et de généraliser les expériences immédiates de nature numérique ou spatiale. Un sens aigu de la quantité, de l'espace et du temps ;
2. La capacité d'établir des dichotomies tranchées ; oui ou non, vrai ou faux, 1 ou 0 ;
3. La capacité d'établir des chaînes causales essentielles ; si A, alors B. La capacité de relier entre elles de telles chaînes et de raisonner ;
4. La capacité et la volonté d'extraire du réel un équivalent abstrait et de considérer la manipulation formelle de cet équivalent abstrait comme une représentation adéquate de la réalité des phénomènes ;
5. La capacité et le goût de jouer avec des symboles même s'ils ne représentent rien de concret, créant ainsi un monde proprement imaginaire.

Une telle liste, en réalité, met certes en relief certains traits importants d'un mode de pensée, mais les isole du contexte social et matériel où il est compris. Elle ne tient pas compte de notre intérêt pour les rythmes propres à la nature tant physique que biologique, et pour les phénomènes souvent étranges que l'on y rencontre. Elle néglige le rôle qu'ont joué la navigation et les voyages, la construction et la destruction, le machinisme et la communication. Elle oublie l'importance, à l'origine des mathématiques, de tout ce qui concerne le commerce, la propriété, les rapports juridiques et psychologiques entre l'abstraction monétaire et les biens concrets. Elle ne tient aucun compte des représentations métaphysiques et religieuses. Malgré ces évidentes limitations, elle

mérite d'être examinée et discutée.

Qu'on me permette de pousser plus loin. Est-il encore trop tôt pour discuter des ingrédients intellectuels de l'informatique ? Peut-être ; mais je vais m'y efforcer.

L'informatique, fille de la technologie mais surtout des mathématiques, reste proche de celles-ci : un informaticien, de manière générale, fonctionne à maints égards comme un mathématicien. Le cordon ombilical a cependant été coupé et il est clair qu'une discipline vraiment nouvelle et un nouveau type de spécialistes sont en train d'apparaître.

Les vrais amoureux de l'informatique préfèrent les machines aux êtres humains. Ils se blotissent amoureusement auprès de leurs terminaux. Ils préfèrent l'action à la pensée et disent plus souvent : "Viens, je vais te montrer un truc génial", que : "Je vais t'expliquer une idée géniale". Il leur arrive néanmoins d'éprouver un état d'illumination mentale où l'action concrète passe au second plan : un informaticien qui a trouvé une idée de programme peut avoir tendance à imaginer celui-ci comme déjà existant, et déjà débarrassé de toute erreur d'écriture.

Les informaticiens sont d'une nature pour le moins impatiente et promettent toujours monts et merveilles. Tout problème leur apparaît comme un noeud gordien qu'ils sauront trancher, et le monde entier comme réductible à des programmes. Ils font souvent preuve d'une nette irritation face à des domaines comme la politique ou la littérature, qui résistent à l'informatisation. Néanmoins, même sur ces questions qu'ils cherchent plutôt à éviter, ils ont tendance à se dire : que tout cela est stupide, mal conçu, primitif, alors que quelques algorithmes bien choisis permettraient de mettre là-dedans un peu d'ordre ! Les informaticiens n'ont pas non plus un sens très développé de l'histoire ; ils considèrent bien davantage le présent et l'avenir que le passé.

Ils sont portés à faire ingurgiter à leurs clients ce qu'eux-mêmes sont en mesure de faire, plutôt que d'étudier sérieusement les besoins de ces clients.

Que peut-on dire, dans ces conditions, des composantes intellectuelles de l'informatique ? Toutes celles que nous avons proposées concernant les mathématiques restent ici valables, mais il convient de les considérer sous un jour un peu différent. Voici la liste que je propose :

1. Pensée algorithmique : programmes, recettes, standardisation de toute activité professionnelle par réduction à une séquence de tâches élémentaires. Volonté frénétique de tout ramener à des décisions rudimentaires du type oui/non ;
2. Pensée modulaire : mise en place d'algorithmes standard utilisables comme autant de "boîtes noires" dans les contextes les plus variés.

Ces deux aspects peuvent sembler s'opposer mais sont d'une certaine façon complémentaires : d'une part, analyse et décomposition des

Calcul informatique et révolution culturelle

LES COMPOSANTES INTELLECTUELLES

DE L'INFORMATIQUE

ILLUSTREES

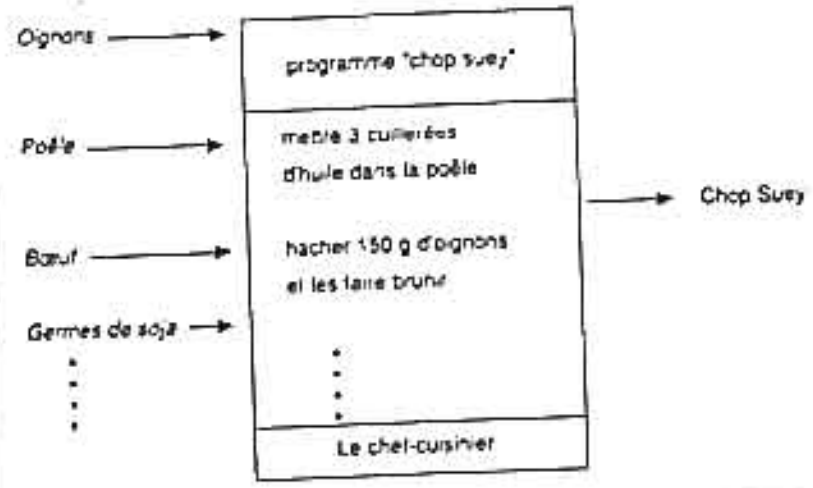
PAR

LA CUISINE CHINOISE

(Les détails du schéma ci-dessous n'ont aucune prétention à l'exactitude culinaire).

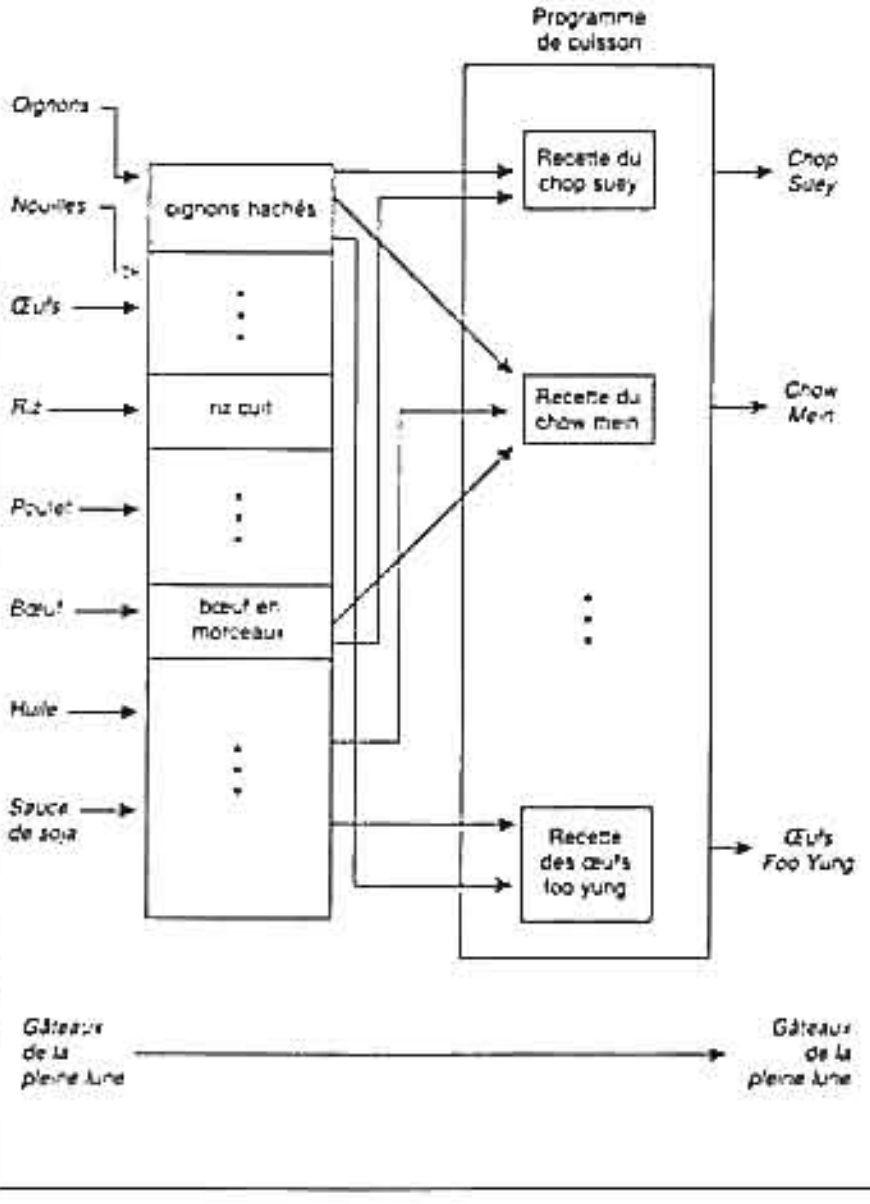
La pensée algorithmique

La recette du chop suey réduite à un algorithme qui sera ensuite mis en œuvre par le chef-cuisinier.



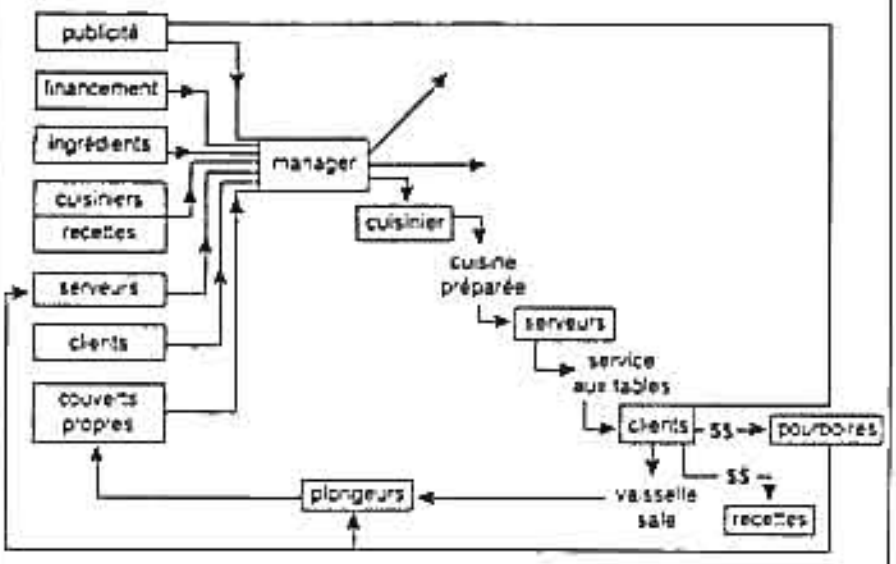
Programmation modulaire

Plusieurs plats ont été décrits sous forme algorithmique et mis en rapport quant aux ingrédients et aux modes de cuisson communs.



Organigramme

L'organisation d'un restaurant conçu comme un système ; détermination du menu et de l'organisation.



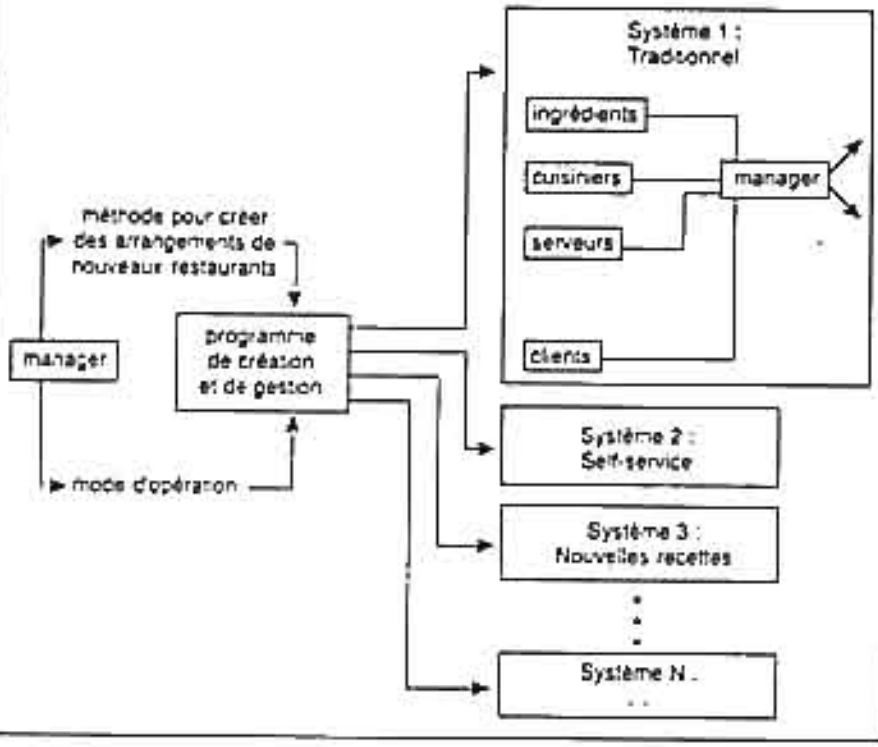
tâches, de l'autre synthèse et recomposition. Décisions microscopiques et résultats de grande ampleur vont ainsi finalement de pair.

3. Pensée systémique. Il faut entendre par là une amplification de ce que nous venons d'appeler pensée modulaire : l'assemblage massif d'éléments standardisés peu nombreux permet la création d'une structure de grande ampleur. La manière d'organiser entre eux les éléments composants s'avère souvent ici plus importante que la nature même de ceux-ci ;
4. Pensée en termes d'état ou de mode dans un système donné ; ainsi distinguera-t-on, en traitement de texte, les rubriques "création de document" et "modification de document" ;
5. Métapensée [3] ou pensée "au second niveau", c'est-à-dire fabrication de programmes dont le point de départ et le résultat sont également des programmes.

Il est clair que ces façons de voir se heurtent nécessairement à un certain nombre d'habitudes psychologiques, sociologiques, technolo-

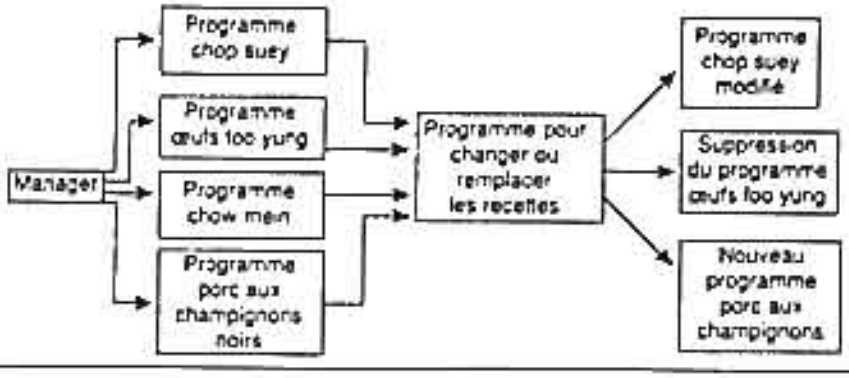
Réarrangement

Un manager peut sans peine réarranger des sous-systèmes pour créer des restaurants différents.



Métapensée : programme de variation

Le programme de variation part de certaines recettes pour les modifier, les supprimer ou en ajouter de nouvelles.



giques. Elles supposent en effet une volonté d'intervenir dans certains secteurs jusque-là réservés à l'activité d'hommes concrets et de remplacer celle-ci par des "équivalents" informatiques. Il faut également disposer d'un matériel (ou hardware) capable de mettre en oeuvre nos programmes abstraits, et accepter qu'inversement ce matériel exige de nous des développements mathématiques inédits. Jusqu'ici, le développement de l'informatique tant au niveau théorique que purement technique a dépassé les espoirs les plus fous. Mais il n'est pas sans intérêt d'imaginer ce qui pourrait se produire au cas où l'évolution du hardware s'arrêterait soudain à un certain niveau, comme ce fut par exemple le cas pour la machine à vapeur.

Si nous apprécions tant l'automobile, au point qu'elle en soit venue à jouer un rôle central dans notre existence, cela tient évidemment au fait que tout un chacun a sous les pieds, grâce à cet engin, une puissance et une rapidité étonnantes ; l'engouement pour les ordinateurs — que ce soit chez les programmeurs, les amateurs ou les simples tâcherons — doit certainement être rattaché, de la même manière, au sentiment que l'on dispose, au bout des doigts, d'une fabuleuse intelligence.

Il peut être utile de rappeler ici quelle représentation l'on se fait communément d'un génie mathématique ou logique : celle d'un individu au front large, mais par ailleurs myope et souffreteux, qui reste en permanence plongé dans des pensées tout à fait étrangères au monde réel. En combat singulier, le premier barbare venu le ferait sans peine rouler dans la poussière. Seule, par conséquent, la puissance que lui procure l'entendement lui permettrait de rester en vie...

Notes

1. Cet article forme un chapitre de l'ouvrage paru en français sous le titre *la Tradition scientifique chinoise*, collection "Savoir", Hermann, 1974. (N. de l'E.)
2. Holistique, holisme : terme créé vers 1925 par un général boer (J.C. Smuts) pour indiquer la tendance de notre univers à construire des unités de complication toujours croissante. (N. de l'E.)
3. Métapensée, «Metathinking» : programme destiné à intervenir dans les programmes préexistants. L'auteur accorde beaucoup d'importance à cette apparente capacité de la machine à «prendre des initiatives» ou plutôt à intervenir dans ses programmes. (N. de l'E.)

LES FONDEMENTS:

Après l'article de M.Gauthier sur l'Affectation, voici

INTRODUCTION A LA PROGRAMMATION FONCTIONNELLE:
Les fonctions

D'après l'excellent livre de A.J.Field et P.G.Harrison: "Functional programming"(Addison-Wesley),1988, 600 pages

1-Pures fonctions:

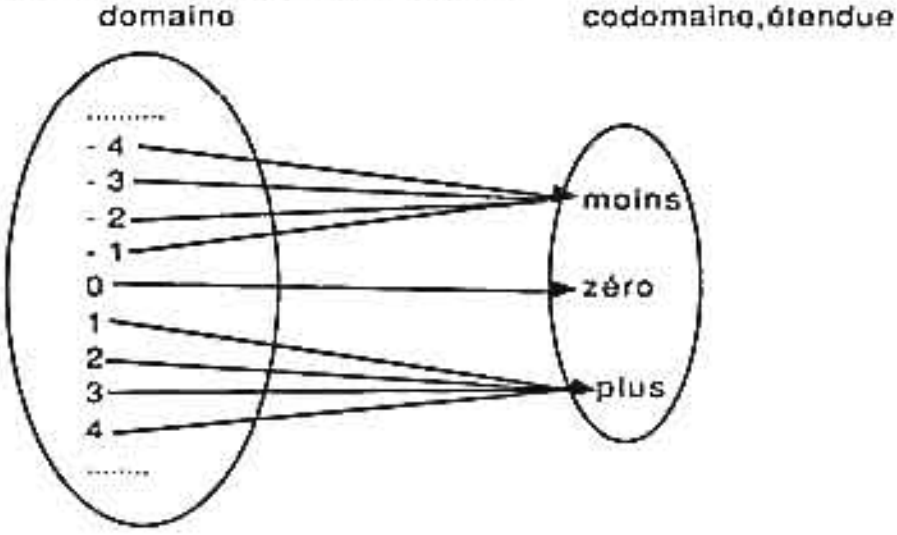
Il s'agit de la notion mathématique de fonction.

Exemple: fonction "signe"

1)On peut utiliser des équations:

....
signe (-3)= moins
signe(-2)=moins
signe(0)=zero
signe(1)=plus
signe(2)=plus

2)On peut aussi faire une représentation sagittale:



"La lettre de l'Adeli n°10"

3)Enfin on peut spécifier une règle:

	moins	si x<0
Signe(x) =	zero	si x=0
	plus	si x>0

x est appelé le paramètre formel de la fonction signe et représente tout élément du domaine de la fonction.

Comme la fonction s'applique à tous les éléments possibles du domaine, elle est dite totale.

Voici un exemple de fonction partielle:

	moins	si x <0
Signe2(x)=		
	plus	si x >0

Il n'y a pas de règle pour couvrir le cas où x = 0. On dit que signe2 est indéfinie quand x = 0.

Une fonction peut être vue comme une boîte noire.

Exemple:



6 est appelé ici le paramètre réel de la fonction i.e. la valeur qui est réellement fournie à la fonction. Fournir un paramètre réel à une fonction est appelé l'application d'une fonction et l'on dit que Signe est appliquée à 6, ce qui signifie que la règle pour Signe est invoquée en utilisant 6

comme paramètre réel. Paramètres formels et réels d'une fonction seront appelés ses arguments.

On peut utiliser une notation mathématique:
signe(6)

signe(6)--->plus

On peut lire le signe ---> comme "égal".

On peut joindre des boîtes noires. C'est ce qu'on appelle de la composition de fonctions.

ex:

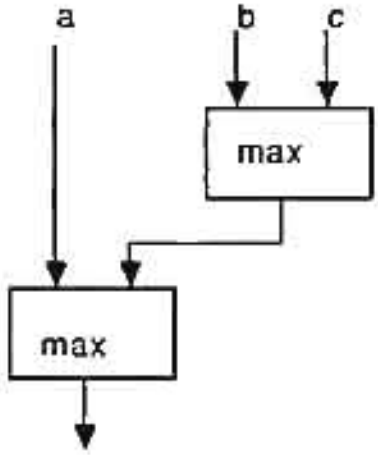
max (m,n) = m si m > n
n sinon

max(1,7) --->7.

On peut aussi utiliser max pour construire une fonction plus complexe. Par exemple, nombre maximum parmi trois nombres:

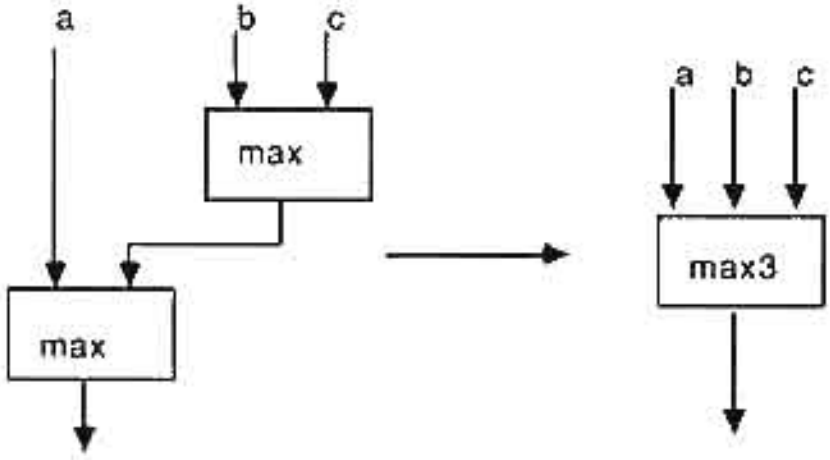
max3(a,b,c) =	a	si a>=b et a>c ou a<=c et a>b
	b	si b>=a et b>c ou b>=c et b>a
	c	si c >= a et c>b ou c>=b et c>a
	a	sinon

(Le cas "sinon" est nécessaire si a=b=c). Ceci n'est pas une définition très élégante. En voici une autre:



max3(a,b,c) = max(a, max(b,c))

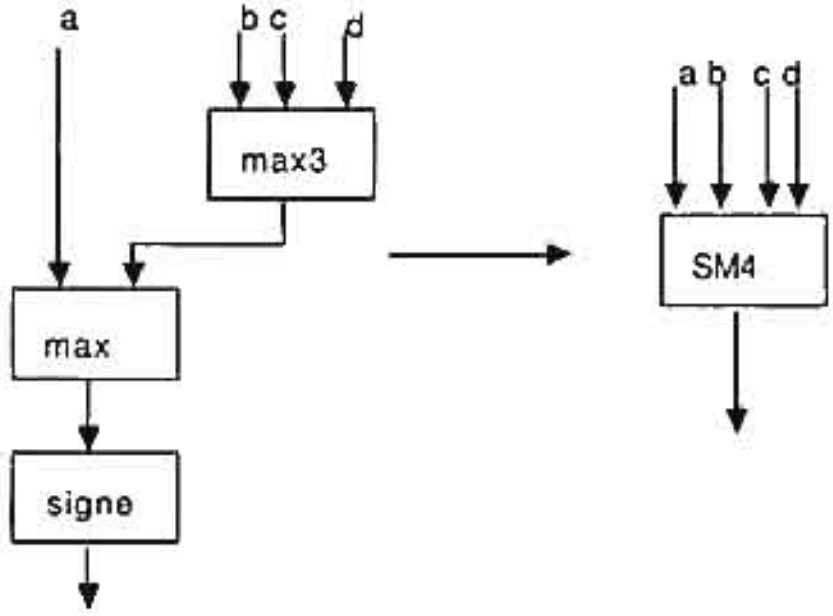
Comme max3 fournit une transformation déterministe de triplets de nombres en nombres, nous pouvons la traiter elle-même comme une boîte noire.



On peut maintenant oublier le fonctionnement interne de cette nouvelle boîte et l'utiliser comme unité ou bloc de construction pour d'autres fonctions, comme la suivante qui donne le signe du plus grand de quatre nombres a,b,c et d en utilisant la fonction Signe et la fonction Max, déjà définies:

SM4 (a,b,c,d) = Signe (max(a,max3(b,c,d)))

ce qui peut être représenté comme une boîte noire:



2-La transparence référentielle:

La propriété fondamentale des fonctions mathématiques qui nous autorise à "brancher" ainsi ensemble des boites noires est appelée la propriété de transparence référentielle. Cela signifie que chaque expression dénote une valeur unique qui ne peut être modifiée par l'évaluation de l'expression ou du fait du partage de cet expression par différentes parties d'un programmme. L'évaluation d'une expression ne fait que changer la forme d'une expression mais en aucun cas sa valeur. Toutes les références à la valeur sont donc équivalentes à la valeur elle-même et le fait que l'expression puisse être utilisée par différentes parties d'un programme ne change rien à cette valeur.

La transparence référentielle est donc ce qui distingue les fonctions mathématiques des fonctions que l'on peut écrire dans les langages de programmation impératifs comme Pascal. Dans ces derniers langages, on autorise les fonctions à se référer à des données globales et on autorise **l'affectation destructrice** , ce qui peut changer la valeur d'une donnée d'une invocation à l'autre de la même fonction . On appelle souvent de tels changements dynamiques de la valeur d'une donnée des effets de bord. Du fait de tels effets, la valeur produite par une fonction peut changer même si ses arguments sont les mêmes à chaque appel.
Ceci rend la fonction très difficile à utiliser parce que pour déterminer quelle valeur la fonction va générer, il faut considérer la valeur courante

de la donnée globale. Ceci nécessiste de considérer l'historique du calcul puisque c'est lui qui détermine la valeur de la donnée globale à chaque instant. Aussi dit-on que les langages impératifs sont référentiellement opaques.

Pour illustrer l'opacité référentielle des langages impératifs, regardons un exemple de programme écrit en syntaxe Pascal:

```
program exemple (output);
var drapeau:boolean;

function f (n:integer):integer;
begin
 if drapeau then f:=n
            else f:= 2*n;
 drapeau:= not drapeau
end;

begin
 drapeau:= true;
 writeln (f(1) + f(2));
 writeln( f(2) + f(1))
end.
```

Si on exécute ce programme, on obtient deux nombres inscrits sur le terminal: 5 et 4.

Ceci est plutôt curieux car les mathématiques nous apprennent que l'addition est commutative ce qui nous autorise à substituer x + y à y + x pour tout x et tout y alors que dans l'expression Pascal f(1) + f(2) fournit une réponse qui est différente de celle de l'expression f(2) + f(1) !

Le problème, bien sûr, vient de ce que la fonction f définie ci-dessus est très différente des fonctions mathématiques que nous avons vu jusqu'ici.

Notre vue d'une fonction mathématique est celle d'une boîte noire qui calcule des valeurs de sortie seulement en termes de valeurs d'entrée. La fonction définie dans le programme ci-dessus est un exemple classique d'une fonction qui est autant dépendante de donnée globale que de ses propres paramètres. Le problème ici est que la valeur de la donnée globale

(i.e.drapeau) dans le programme Pascal peut être modifiée et que c'est cela qui détruit la transparence référentielle du langage. La valeur de la fonction primitive + ne change pas, c'est-à-dire qu'elle dénote toujours la fonction qui additionne des objets. Le coeur du problème dans le programme Pascal est donc l'affectation destructrice:

drapeau:= **not** drapeau

qui change la valeur de drapeau. Si drapeau est vrai avant l'exécution de l'énoncé alors il deviendra faux après et vice versa. Un tel énoncé n'est pas autorisé en mathématiques; le raisonnement mathématique est basé sur l'idée d'égalité et de remplacement d'une expression par une autre qui signifie la même chose, i.e. qui dénote la même valeur. Par exemple, nous pouvons remplacer l'expression

4 + 8

par l'expression

12

parce que les deux expressions sont des dénotations de la même valeur, i.e. le nombre 12.

C'est une caractéristique des programmes fonctionnels que de ne pas avoir d'affectations destructrices. Au lieu de voir une variable comme une place pour une valeur qui peut être périodiquement mise à jour en affectant différentes valeurs à cette variable, les variables dans un programme fonctionnel sont comme des variables mathématiques: si elles existent alors elles ont une valeur et la valeur ne peut changer. Alors qu'un programme est une séquence d'énoncés impératifs qui décrivent comment l'ordinateur doit résoudre un problème en termes de changement d'états (mise à jour de variables affectables), un programme fonctionnel décrit ce qui est à calculer, c'est-à-dire que le programme est seulement une expression, définie en termes de fonctions prédéfinies et de fonctions définies par l'utilisateur, valeur de l'expression qui constitue le résultat du programme. Il n'y a pas de notion d'état de programme et il n'y a pas de notion d'histoire de programme.

RESUME:

-Les pures fonctions (mathématiques) peuvent être utilisées pour construire des programmes

-De nouvelles fonctions peuvent être construites par composition de vieilles fonctions

-Les langages basés sur la programmation par fonctions possèdent la transparence référentielle.

Notes de H.Habrias