R.A.I.R.O. Informatique/Computer Science
(vol. 11, n° 1, 1977, p. 21 à 34)

Libres opinions — S. KRAKOWIAK, *rédacteur*

INITIATION A LA PROGRAMMATION EN MILIEU INDUSTRIEL (*) (1)

par B. MEYER (2)

Résumé. — *En décrivant l'organisation du cours d'introduction à la programmation donne à la Direction des Études et Recherches (D.E.R.) d'E.D.F., on étudie comment les idées modernes sur la programmation peuvent être appliquées à un enseignement élémentaire, et quels sont les problèmes particuliers soulevés lorsque cet enseignement est dispensé dans un gros centre de calcul non universitaire et à l'intention de non-spécialistes.*

On émet également, à la lumière de l'expérience menée à la D.E.R., quelques remarques sur l'importance du langage de programmation utilisé pour un enseignement d'initiation. La discussion entamée dans cet article débouche en fait sur tout le problème de l'amélioration des méthodes de programmation en milieu industriel.

> *En quelque étude que ce puisse être, sans l'idée des choses représentées, les signes représentants ne sont rien. On borne pourtant toujours l'enfant à ces signes, sans jamais pouvoir lui faire comprendre aucune des choses qu'ils représentent.*
>
> Rousseau, *Émile*, II, 18

Nous voudrions dans cet article faire part de notre expérience d'enseignement de la programmation à la Direction des Études et Recherches (D.E.R.) d'E.D.F. et commenter en particulier le cours d'« Initiation à la programmation et au langage FORTRAN » donné régulièrement, depuis novembre 1975, au centre de calcul E.D.F. de Clamart par M. Brisson, M. Dostatni, M^me^ Lenglet et l'auteur.

La démarche générale qui a présidé à l'organisation de ce cours repose sur des principes méthodologiques voisins de ceux de l'expérience décrite dans [11] par M. Lucas, J. Mossière et P. Scholl, et qui concerne la formation en programmation à l'université de Grenoble à l'aide du langage ALGOL W. Cette similitude de vues nous dispensera de nous étendre sur un certain nombre de points fondamentaux; nous insisterons plutôt sur certaines différences d'appréciation

(*) Reçu février 1976.

(1) Cet article a fait l'objet d'une présentation à la session « Algorithmique et Programmation », au congrès de l'AFCET, le 5 novembre 1976.

(2) Électricité de France, Direction des Études et Recherches, Clamart.

avec les auteurs de [11], d'une part, et, de l'autre, sur les problèmes particuliers que soulève l'enseignement de la programmation en milieu industriel. Nous livrerons chemin faisant quelques réflexions sur le rôle des langages de programmation dans l'enseignement de l'algorithmique.

L'organisation du cours mené à E.D.F. a largement bénéficié de l'expérience du cours d'introduction à la programmation et au langage ALGOL W donné à l'Institut d'Informatique d'Entreprise (C.N.A.M.) en octobre 1975. Nous tenons à remercier les autres enseignants de ce cours, et en particulier M. Kaiser, pour de nombreux contacts fructueux.

1. GÉNÉRALITÉS

Le cours se caractérise tout d'abord par un certain nombre de contraintes sur lesquelles les organisateurs ne peuvent guère agir.

La première de ces contraintes est le niveau très variable des participants, parmi lesquels on rencontre aussi bien des agents techniques titulaires du seul B.E.P.C. que des ingénieurs fraîchement émoulus d'une grande école. Bien que sensibles à l'aspect « défi intellectuel » [6] de la programmation (nous avons fait de la difficulté de la programmation l'une des idées-leitmotiv du cours), nous avons décidé de ne pas appliquer de sélection initiale par le niveau scolaire. L'une des hypothèses de base de notre action est en effet que la méthodologie de la programmation, dont les principes ont émergé ces dernières années dans l'université, peut et doit être appliquée à la programmation quotidienne dans un centre de calcul tel que celui de Clamart; on ne peut guère espérer influer sur cette programmation si l'on restreint l'enseignement aux personnes de haut niveau universitaire.

La seconde des « variables liées » par le contexte est pour nous le langage de programmation. Le centre de calcul de la D.E.R., qui regroupe la plus grande partie du calcul scientifique à E.D.F., utilise essentiellement (sur un IBM 370/168 couplé à un 370/168, sous OS/MVT, bientôt MVS) le langage FORTRAN. Quels que soient les sentiments que l'on nourrit à l'égard de ce langage, il n'est pas question, dans un milieu industriel, et lorsqu'on s'adresse à des élèves pour qui l'informatique n'est qu'un outil, de passer pour la phase d'initiation par un autre langage mieux adapté — démarche parfaitement légitime lorsqu'on forme des informaticiens professionnels, qui auront de toute façon à connaître plusieurs langages. Nous avons donc été conduits à séparer de façon extrêmement systématique la présentation des *concepts* de celle de leur représentation pratique en FORTRAN. Nous reviendrons sur ce problème au paragraphe 4.

Une autre caractéristique du cours sur laquelle nous n'avons pas une emprise complète est sa durée. Le stage dure trois semaines; bien qu'il ne soit pas possible de le faire suivre, comme dans une université, de séances hebdomadaires pendant un trimestre ou un semestre, il est cependant complété par une semaine complémentaire au bout de cinq mois.

L'avantage d'un stage en continu est que, pendant trois semaines, nous gardons un contrôle entier sur nos élèves et pouvons ainsi les préserver des « mauvaises influences »; son inconvénient, bien entendu, est que les concepts nouveaux n'ont guère le temps de mûrir entre deux séances successives.

Trois semaines sont un délai bien court pour former des programmeurs. D'une certaine manière, cependant, cette brièveté a un effet bénéfique, en ce sens qu'elle nous a amenés à effectuer un fructueux travail de réflexion sur ce qu'il est véritablement fondamental d'enseigner à des débutants.

Il est intéressant de noter que cette durée de trois semaines, qui est pour nous un seuil en dessous duquel aucun travail sérieux n'est possible, paraît souvent trop longue à des personnes extérieures à notre équipe — ce qui témoigne d'une opinion, encore largement répandue, selon laquelle la programmation est une chose facile qui s'apprend en quelques jours. Nous avons été réconfortés sur ce point par l'opinion des élèves qui, à la fin du cours, trouvent à 80 % sa durée « adéquate » ou « trop courte ».

2. ORGANISATION DU COURS

L'un des principes fondamentaux du cours est l'importance des travaux dirigés. Nous différons, semble-t-il, des auteurs de [11] en ce que nous faisons traiter des exercices, avec passage sur machine, dès le second jour de stage. L'informatique est *aussi* une science expérimentale; quiconque a enseigné la programmation (ou simplement programmé lui-même) sait que les plus beaux discours sur la conception descendante, sur les structures de contrôle, voire à un niveau plus élémentaire sur la distinction entre les différentes phases du traitement d'un programme (compilation, chargement, exécution) restent vides de sens tant que l'auditeur n'a pas écrit, perforé et soumis un programme, étudié une liste, et tant qu'il ne s'est pas heurté à l'un des problèmes fondamentaux de la programmation — celui de l'erreur.

Ceci ne signifie pas, bien entendu, que l'élève doit avoir un accès incontrôlé à la machine. En fait, il n'y a pas accès du tout : il perfore ses cartes, et les remet à l'un des enseignants, qui ne passe le programme qu'après en avoir contrôlé la validité.

Outre qu'elle décharge les élèves de tout JCL (le langage de contrôle n'est présenté, très rapidement, que l'avant-dernier jour du stage), cette méthode nous aide à insister fortement sur le principe de *correction initiale* des programmes, plutôt que sur le « debugging » (souvent appelé, par pudeur, « maintenance »). De fait, trop de manuels et de cours d'initiation continuent à propager le mythe selon lequel n'importe quel programme, même bâclé, finira par converger vers un « bon » programme après un nombre suffisant de passages sur machine (de « tests »). Le cours d'initiation à la programmation est la première et la dernière occasion d'empêcher la cristallisation atavique de ce mythe dans l'esprit des nouveaux programmeurs.

La répartition horaire est de 2 heures de cours par jour, de 10 h à 12 h, la séance de 9 h à 10 h et celle de l'après-midi étant consacrées aux travaux dirigés. Pour les TD, les élèves sont répartis en groupes de deux ou trois; chaque groupe doit mener à terme, au rythme qui lui convient, la solution de tous les problèmes posés (une douzaine pendant le stage), avec passage sur machine. Il y a normalement deux passages généraux sur machine par jour, un le matin, et un le soir. La séance de l'après-midi consiste en des travaux dirigés au sens propre, c'est-à-dire que les différents groupes cherchent à résoudre les problèmes demandés, les enseignants étant normalement là pour les mettre sur la bonne voie et, de temps en temps, faire une synthèse. La séance de 9 h à 10 h, par contre, est essentiellement consacrée à l'analyse des programmes passés la veille, et, éventuellement, à la correction d'erreurs mineures (syntaxiques par exemple).

Les 2 heures quotidiennes de cours magistral nous ont paru suffisantes, beaucoup de notions demandant de toute façon à être réexpliquées en TD. Notons que certains des cours sont remplacés, vers la fin du stage, par des conférences d'approfondissement données par des personnes extérieures à notre équipe : un responsable de l'exploitation (sur l'organisation du centre de calcul et du réseau de la D.E.R.), un ingénieur du système (sur la notion de système d'exploitation et sur les principes de OS/MVT ou MVS), et un responsable d'un projet de programmation extérieur au service « Informatique et Mathématiques Appliquées » (sur l'expérience d'un « gros utilisateur » de l'informatique). Ces conférences ont pour but de compenser quelque peu l'orientation générale relativement abstraite de notre cours.

Nous nous sommes limités pour l'essentiel, quant au FORTRAN enseigné, à la norme ANSI, en excluant les « améliorations » du constructeur. Quelques exceptions à cette règle concernent des caractéristiques sans lesquelles il est vraiment trop difficile de travailler, et qui seront sans doute acceptées par la prochaine norme [1], comme les constantes littérales entre apostrophes.

Afin de rendre possible l'utilisation d'instructions d'entrées-sorties sans format dès le début du cours, nous avons écrit et mis en bibliothèque un certain nombre de primitives faciles à utiliser, et dont le texte est plus tard fourni aux élèves.

L'organisation pédagogique du cours, qui sera décrite avec plus de détails au paragraphe suivant, a été influencée par un certain nombre de problèmes particuliers. Dans une grande entreprise comme E.D.F., où le traitement automatique de l'information est employé sur une large échelle, beaucoup d'agents sont de toute façon confrontés plus ou moins directement, dans leur travail quotidien, à l'informatique et aux ordinateurs. Parmi nos élèves, si 30 % déclarent ne posséder aucune notion préalable, les autres ont déjà, dans des conditions extrêmement diverses, une certaine expérience de l'informatique. L'enseignant doit, dans ces conditions, adopter un esprit quelque peu défensif, et extirper de « mauvaises » habitudes pour pouvoir inculquer de « bonnes » habitudes. Il ne suffit pas, ainsi, d'introduire le GOTO uniquement comme

une méthode de réalisation de structures de contrôle évoluées (boucles, choix), mais il faut également insister sur les dangers que présente son utilisation autonome; il ne suffit pas d'adopter un pseudo-langage de haut niveau pour la conception et la présentation des algorithmes, mais il faut également le justifier par rapport aux méthodes utilisant des organigrammes, aux idées reçues sur la différence entre « analyse » et « programmation », etc.

Lucas, Mossière et Scholl mentionnent qu'il est difficile de persuader les élèves de recourir à la conception descendante autrement, semble-t-il, que par la contrainte. Nous nous sommes heurtés au même problème, rendu plus sensible encore par l'existence de certaines habitudes acquises. Le meilleur moyen que nous avons trouvé pour le résoudre consiste à abandonner les élèves à leurs propres moyens pour un certain nombre d'exercices non triviaux; ceux qui se lancent immédiatement dans l'écriture de code FORTRAN, ou d'organigrammes, n'arrivent en général à aucune solution viable. L'enseignant peut alors montrer comment un raisonnement systématique conduisant à une décomposition logique et à des raffinements successifs, à l'aide d'un pseudo-langage de haut niveau et d'assertions intermédiaires, permet d'aboutir à une solution garantie correcte.

3. ORGANISATION PÉDAGOGIQUE

De même que les auteurs de [11], nous avons constamment cherché à introduire les concepts avant leur représentation. Pour ne pas être gênés par les particularités de FORTRAN, nous avons dès le début utilisé un « pseudo-langage » de haut niveau, de syntaxe floue et d'allure algolique. Les notions fondamentales, comme celle de variable, de structures de contrôle, de structures de données, etc., sont d'abord exposées à l'aide de ce formalisme, puis est abordée leur représentation en FORTRAN. C'est ainsi, par exemple, que l'instruction GOTO n'est jamais présentée comme une structure de contrôle *per se*, mais comme un moyen de réaliser le si ... alors ... sinon ou le tant que ... répéter.

Le tableau de la figure 1 donne l'organisation des séances de cours magistral et de travaux dirigés (pour ceux-ci, on indique le jour où chaque problème est soumis pour la première fois aux élèves). La progression suivie s'inspire de celle suggérée par R. W. Floyd [8]; elle est différente de celle de [4] et appelle quelques remarques.

Après une séance d'introduction aux concepts d'information, d'informatique, d'ordinateur et de programmation (et une visite du centre de calcul), on expose :

— les objets de base manipulables (essentiellement les entiers et les chaînes de caractères, avec une allusion aux nombres flottants);

— deux structures de contrôle : l'enchaînement (succession des instructions dans le temps) et la répétition avec compteur (boucle « pour ... variant de ... à ... répéter ... »);

— quelques éléments de FORTRAN : la présentation physique d'un pro-

Figure 1.
Organisation du cours et des TD.

JOUR	COURS	TRAVAUX DIRIGES
1	Introduction aux notions d'information, d'informatique, d'ordinateur et de programmation.	Visite du centre de Calcul Utilisation d'une machine perforatrice (tambour...)
2	Structures de contrôle : a) enchaînement séquentiel b) répétition avec compteur. Objets : entiers ; constantes littérales. FORTRAN : présentation physique d'un programme ; commentaires ; appel de sous-programmes ; expressions arithmétiques simples.	Boucles simples P_1 : dessiner un échiquier (sous-programmes fournis aux élèves).
3	Variables ; types ; déclarations. Affectations. Tableaux.	P_2 : calcul de $n!$ P_3 : calcul de e. Problème de l'erreur.
4	Conditions. Alternative (si...alors...sinon.. Objets : logiques. FORTRAN : constantes, variables et expressions de type logique. Expression d'une alternative à l'aide d'un *IF* "logique" et d'un *GOTO*.	P_4 (cf. P_1) : imprimer un échiquier avec un roi, etc., en y signalant les pièces en prise. P_5 : calcul de termes de la suite de Fibonacci.
5	Ecriture d'un sous-programme. Notions sur le mode de passage des paramètres. FORTRAN : sous-programmes de types *FUNCTION* et *SUBROUTINE*.	P_6 : récrire P_2, P_3, P_4, P_5 comme des sous-programmes. P_7 : imprimer des rectangles enchevêtrés :
6	Boucles indéfinies (tant que) ; problème de la finitude d'une boucle. Assertions dans un programme. Notion de démonstration informelle.	P_8 : calculer e avec une précision donnée. Poursuite de la construction d'un "meccano" de figures géométriques simples.

JOUR	COURS	TRAVAUX DIRIGES
7	Révision. Lectures en format libre.	Poursuite et révision des exercices précédents.
8	Entrées et Sorties ; notion de fichier	Entrées et Sorties (suite du cours et applications)
9	Conférence : notion de système d'exploitation, multiprogrammation.	P_9 : Tracer un histogramme.
10	Représentation et manipulation des nombres sur machine (1)	P_{10} : Imprimer des dessins (représentant des lettres). P_{11} : Calcul du plus grand diviseur d'un entier. Déterminer si un entier donné est premier.
11	Représentation et manipulation des nombres sur machine (2)	P_{12} : le crible d'Eratosthène.
12	Conférence : le centre de calcul et le réseau de la Direction des Etudes et Recherches	P_{13} : jeu de bataille.
13	a) Compléments sur les modes de communication de données entre unités de programme FORTRAN : *COMMON*. b) Choix multiples ; FORTRAN : branchements indexés. c) Les instructions-fossiles de FORTRAN : *IF* "arithmétique" ; *EQUIVALENCE* ; *ASSIGN*.	P_{14} : sous-programmes de gestion d'un arbre binaire de recherche.
14	Conférence : l'experience et les méthodes du responsable d'une équipe de programmation.	Suite des exercices précédents. Introduction au JCL.
	Synthèse : la Méthodologie de la Programmation	P_{15} : Transformation d'expression infixe en forme polonaise postfixée.

gramme, l'appel d'un sous-programme, les constan es et expressions arithmétiques simples, et les primitives d'écriture mentionnées au paragraphe précédent.

On peut donc dès la première séance de TD aboutir à des programmes du type

```
C     IMPRESSION DES CARRÉS DES NOMBRES DE 1000 A 1100
      CALL IMPR (' ★ CARRÉS DES NOMBRES DE 1000 A 1100 ★')
      DO 10    I = 1000, 1100
         CALL ECR (I★I)
10       CONTINUE
      STOP
      END
```

On peut également faire découvrir aux élèves que des boucles peuvent être imbriquées. L'utilisation d'une présentation statique faisant ressortir cette imbrication (décalages) est obligatoire tout au long du stage.

La répétition contrôlée par compteur est clairement moins fondamentale que la répétition de type tant que. La raison pour laquelle nous l'introduisons d'abord est que la répétition tant que est un concept difficile, et qui fait intervenir des notions comme celle de variable, de condition, etc., qui sont loin d'être triviales. Pour traiter la boucle pour ou son équivalent FORTRAN, la notion de variable n'est pas nécessaire; il suffit de la notion plus simple d'indice. L'instruction

pour i variant de m à n répéter
action (i)

est facilement compréhensible [« action (i) étant une action dépendant éventuellement d'un paramètre, on spécifie que action (m), action ($m+1$), ... action (n), doivent être exécutées successivement dans cet ordre »]. A ce niveau de présentation, on ne mentionne pas l'implantation physique d'une boucle (test et branchement); la directive CONTINUE (qui est requise des élèves pour terminer une boucle DO) est présentée comme un délimiteur syntaxique, non comme une instruction.

Notons également que, dès cette première séance, est introduit l'appel de sous-programme. Tout au long du cours, certains exercices sont étudiés avant que les élèves possèdent les outils nécessaires à leur résolution complète. Lorsqu'au cours de l'analyse descendante on bute sur une telle absence, le sous-programme nécessaire (écrit par les enseignants et mis en bibliothèque) peut être invoqué par les programmes des élèves. De ce point de vue, on pourrait dire que si la programmation est l'art de repousser les décisions [5], l'enseignement idéal de la programmation est celui qui sait retarder à bon escient l'acquisition de nouvelles connaissances; dans les deux cas, l'utilisation de sous-programmes est un moyen d'atteindre le but recherché.

Ainsi, après avoir étudié P_1 (impression d'un échiquier), on donne quelques notions sur P_4 [impression d'un échiquier, avec un R en (I_0, J_0) correspondant à un roi, et un astérisque signalant les pièces en prise].

Le programme s'écrit :

```
pour i variant de 1 à 8 répéter
|  { imprimer une ligne de l'échiquier }
|  pour j variant de 1 à 8 répéter
|  |  écrire-caractère (carac-roi (i, j, I0, J0));
|  revenir à la ligne
```

Au quatrième jour, après l'étude des alternatives, les élèves compléteront P_4 en écrivant eux-mêmes carac-roi [qui donne comme résultat le caractère à imprimer en (i,j) sachant qu'il y a un roi en (I_0, J_0)].

La notion de sous-programme joue un rôle fondamental dans le cours. L'appel de sous-programme, on l'a vu, est introduit à la première séance sur la programmation (jour 2); la technique d'écriture d'un sous-programme est exposée à la quatrième séance (jour 5) (c'est-à-dire avant la notion de boucle indéfinie). L'accent est alors mis sur la manière dont s'effectue la correspondance entre arguments réels et arguments formels; cette notion nous a paru l'une des plus difficiles à expliquer.

En présentant les boucles indéfinies de type tant que et répéter ... jusqu'à (jour 6), nous insistons sur le problème de *finitude* d'une boucle et présentons quelques éléments de *démonstration informelle* d'un programme. L'une des idées directrices du cours est que la notion de démonstration informelle peut et doit faire partie d'une initiation à la programmation dès le niveau le plus élémentaire.

Plus précisément, la méthode descendante, telle que nous la comprenons, consiste en particulier à écrire des assertions informelles avant de commencer à écrire le programme. On part de deux assertions, l'assertion initiale (hypothèse) et l'assertion finale (conclusion); la recherche de cette dernière prend souvent un temps non négligeable, qui, croyons-nous, n'est pas perdu. On remplit ensuite l'espace entre les assertions déjà écrites en insérant des assertions intermédiaires dont on espère qu'elles sont plus faciles à réaliser. Le programme se glisse alors dans les interstices entre assertions : l'élément de programme i est un programme dont l'hypothèse est l'assertion $i-1$ et qui permet d'assurer une conclusion identique à (ou impliquant) l'assertion i. Cet élément de programme est, soit directement utilisable, soit destiné à être raffiné à l'étape suivante.

En écrivant une boucle tant que, par exemple, on ne partira jamais de l'« action » à itérer mais toujours de la condition à assurer. Plus précisément, la boucle tant que est présentée comme un moyen d'assurer la validité d'une certaine assertion i : de

```
{ assertion i−1 }
− − − ? − − −
{ assertion i }
```

on passe à :

```
{ assertion i−1 }
tant que ~(assertion i) répéter
|  « une certaine action qui nous rapproche de la validité de assertion i »
{ assertion i }
```

Le niveau suivant de raffinement consiste à trouver l'« action » demandée. La notion d'invariant de boucle s'introduit naturellement.

L'un des buts recherchés par l'emploi de ce genre d'approche est de faire une séparation aussi nette que possible entre ce qui reste encore, en programmation quotidienne, du domaine de l'intuition et de l'expérience — par exemple la classe d'« actions » vers laquelle, on va spontanément orienter sa recherche —, et ce qui est par contre déductible de façon quasi systématique, comme la séquence des assertions.

Deux séances (jours 10 et 11) sont consacrées à la représentation et à la manipulation des nombres sur ordinateur, avec application à l'IBM 370; on présente quelques-unes des difficultés que soulèvent les calculs sur machine. Ces séances ne sont pas une introduction à l'analyse numérique; elles visent simplement à attirer l'attention des élèves sur ce qu'il advient des « nombres » lorsque l'on s'avise de les faire manipuler par un ordinateur. Il s'agit de leur faire prendre conscience, selon l'expression de [9], du fait qu'« un livre de maths ne suffit pas ».

Au cours de la séance du treizième jour, est exposée la notion de commun, suivie d'une discussion des différents modes de transmission d'information entre unités de programme en FORTRAN. L'attention des élèves est attirée sur les dangers de ce que nous appelons le « commun poubelle » (un COMMON fourre-tout présent dans toutes les unités d'un programme); il s'agit là d'une pratique extrêmement fréquente chez les programmeurs FORTRAN, destinée en particulier à simuler l'allocation dynamique, et provoquant d'interminables séries d'erreurs qui font la joie des spécialistes de l'« assistance aux utilisateurs ».

Au cours de la même séance, sont présentées, dans l'esprit « défensif » que nous avons décrit plus haut, les « instructions fossiles » de FORTRAN : IF arithmétique, GOTO « assigné » et directive EQUIVALENCE (cette dernière pouvant cependant être utilisée pour des définitions de macros très restreintes).

La dernière séance, enfin, est un cours de synthèse sur les problèmes de la programmation. On y résume un certain nombre de concepts comme la conception descendante, on donne quelques aperçus sur des méthodes de travail telles que celles de Mills et Baker (« Chief Programmer Teams », *voir* [2]); mention y est faite brièvement (trop brièvement) de problèmes fondamentaux comme celui de la complexité; enfin, on donne aux élèves un certain nombre de règles du genre « n'écrivez pas d'unités de programme de plus d'une ou deux pages » — « pensez à la correction avant de pensez à l'efficacité », etc. règles qui ne constituent évidemment pas en elles-mêmes une méthodologie de la programmation, ni même de véritables « éléments de style » [10], mais simplement une sorte de pense-bête du programmeur, qui permet d'éviter de nombreux écueils.

Notons qu'à aucun moment nous n'avons cherché à présenter notre cours comme un cours de « programmation structurée », bien qu'il ait parfois été

compris comme tel de l'extérieur ([1]). Nous pensons en effet que l'expression a été tellement galvaudée qu'elle en a perdu toute signification. En outre, quelle que soit l'influence que les travaux des théoriciens de la programmation structurée ont pu avoir sur l'élaboration de notre enseignement, il est clair que celui-ci se situe au niveau beaucoup plus bas d'une initiation à la programmation tout court; ce niveau est le seuil de connaissances au-dessous duquel il nous paraîtrait peu sage d'ouvrir aux programmeurs l'accès à la machine.

4. LANGAGES ET ENSEIGNEMENT

Nous voudrions dans ce paragraphe énoncer un certain nombre de remarques sur l'importance du langage utilisé dans un enseignement de la programmation.

On entend assez couramment affirmer que le langage choisi pour une initiation à la programmation importe peu, l'essentiel étant que les concepts importants soient clairement exposés, dégagés des idiosyncrasies du langage. On rencontre même l'opinion suivant laquelle un langage de relativement bas niveau comme FORTRAN, ou un langage manifestement trop complexe comme PL/1, sont préférables à des langages plus « structurés » parce qu'ils permettent précisément à l'enseignant de mieux faire la distinction entre les concepts et leur représentation.

Nous voudrions à la lumière d'une double expérience (utilisation de FORTRAN et utilisant d'ALGOL W) nous porter vivement en faux contre cette affirmation. L'enseignant utilisant FORTRAN est obligé de consacrer une part considérable de son temps et de ses efforts à mettre en garde les élèves contre des constructions mal adaptées, ou contre des opérations dont la sémantique ne correspond pas au nom; cela va du sempiternel et inévitable problème du signe « = », qui malgré toutes les explications et tous les cavéats met toujours un certain temps à se faire reconnaître par les élèves comme un opérateur d'affectation, à l'effet de la boucle DO dans le cas où sa première borne est supérieure à la seconde (effet non précisé par la norme du langage, mais que les compilateurs traduisent par une exécution du corps de boucle, alors qu'on attendrait une instruction nulle), en passant par les contorsions nécessaires à la manipulation de caractères, etc. L'apprentissage de la programmation et la compréhension de la notion d'algorithme demandent, quel que soit le langage utilisé, un effort conceptuel considérable et l'adoption d'une démarche nouvelle; cette tâche est rendue plus lourde encore s'il faut de surcroît assimiler les caractéristiques baroques d'un langage inadapté. Il est clair en outre que le langage, en programmation comme dans les autres domaines de l'activité humaine, est difficilement dissociable de la pensée; il suffit d'avoir étudié la manière dont les habitués de FORTRAN (ou de tout autre langage) conçoivent

([1]) Au point que tel stagiaire a été envoyé suivre au préalable, chez un constructeur, un cours « classique » d'initiation à FORTRAN, ce qui lui a surtout causé des difficultés de compréhension supplémentaires.

et écrivent leurs programmes, pour voir à quel point un langage de programmation peut influencer, ou handicaper, le raisonnement (1).

Nous énonçons ci-dessous un certain nombre de caractéristiques *minimales* que doit posséder, selon nous, un système d'écriture de programmes pour être propre à l'enseignement. Par « système d'écriture de programmes », nous entendons à la fois le langage, le compilateur et son environnement, les trois nous paraissant indissociables. Bien qu'aucun système de qualité « industrielle » (c'est-à-dire permettant à la fois l'enseignement et l'écriture de programmes « réels ») ne réponde actuellement, à notre connaissance, à toutes ces conditions, la réalisation d'un tel système ne nous paraît pas, en 1976, une utopie. Par contre, nous ne pensons pas que la pédagogie de la programmation puisse beaucoup progresser avant qu'un tel système soit disponible *à la fois pour l'enseignement et pour la production.*

a) *Caractéristiques du langage*

- simplicité et petite « taille » (de l'ordre de celle de FORTRAN ou de PASCAL);
- structures de contrôle adéquates (semblables à celles de PASCAL ou d'ALGOL W). La présence ou l'absence du GOTO est un problème secondaire; le GOTO ne doit pas intervenir pour l'enseignement d'initiation;
- structures de données adéquates : tableaux, enregistrements, structures récursives;
- deux caractéristiques à la « simulation » desquelles les programmeurs FORTRAN consacrent, nous l'avons vu, des efforts considérables et toujours recommencés :
  — la récursivité,
  — les tableaux à bornes dynamiques (au niveau de chaque sous-programme);
- structure de blocs :
  — instruction composée,
  — nomenclature locale, qui peut être limitée, comme en FORTRAN ou en PASCAL, au niveau du sous-programme;
- définition précise des modes de transmission de données entre unités de programmes;
- des instructions d'entrées-sorties simples, avec mise en pages automatique;
- obligation de déclarer tous les symboles;
- possibilité de demander au compilateur un certain nombre de contrôles sémantiques (protection de certaines variables en écriture par exemple).

b) *Caractéristiques du système de compilation*

- compilateur diagnostiquant un maximum d'erreurs, sans « effet d'avalanche », mais ne les corrigeant pas;

(1) Rousseau : *Donnez aux enfants tant de synonymes qu'il vous plaira : vous changerez les mots, non la langue : ils n'en sauront jamais qu'une.*

- tables de références croisées, tables de symboles, etc.;
- diagnostics aussi *peu équivoques* que possible. Le compilateur que nous avons à notre disposition produit des diagnostics du genre : « vous avez fait l'erreur... ou l'erreur... ou l'erreur... (...) », avec souvent sept ou huit possibilités, ce qui sape à la base tout effort d'enseignement progressif des différents éléments d'un langage;
- diagnostics en français. Ce point pourra faire sourire; il nous paraît impératif dès lors qu'on s'adresse à des élèves qui sont hors de l'université;
- contrôle optionnel de la validité des noms à l'exécution (indices de tableaux, etc.);
- contrôle optionnel des erreurs à l'exécution, avec diagnostics formulés, chaque fois que c'est possible, en termes du programme-source, liste des valeurs des variables en clair, trace du programme, etc. Les possibilités offertes par le compilateur ALGOL W (voir [12]) sont à cet égard remarquables;
- possibilité d'utiliser des bibliothèques standard ou non, et des programmes écrits dans d'autres langages.

5. CONCLUSION

Nous pensons avoir atteint, en réalisant ce cours, l'un des buts initiaux, qui était de montrer que les études modernes sur la programmation ont atteint un niveau suffisant pour être applicables à un enseignement élémentaire, et que les compromis nécessaires dans l'environnement d'un centre de calcul non universitaire (langages, horaires, niveau des élèves) laissent cependant la possibilité pour un enseignement de bon niveau.

Ceci dit, il est un certain nombre de problèmes que nous n'avons pas su résoudre de manière satisfaisante. Nous n'avons pu, ainsi, éviter une certaine dispersion (il est difficile de trouver une unité sous-jacente à des sujets tels que la conception descendante, les FORMATS du fortran (huitième jour), et les rudiments du JCL d'IBM (quatorzième jour)). Un certain nombre de points importants n'ont pas été étudiés assez sérieusement, soit par manque de temps (nous aurions ainsi aimé développer la notion de structure de données de façon systématique, proposer un exercice de traitement de fichiers, etc.), soit parce que leur représentation en fortran demande un outillage disponible seulement en fin de stage (c'est le cas de la récursivité). Nous n'avons pas, faute de temps, couvert le langage FORTRAN dans sa totalité, et avons traité rapidement des points importants, comme par exemple, au cours de la première session, les lectures avec format (les écritures ont cependant été étudiées en détail). Il se pose ici des problèmes de choix : ceux qui auront à utiliser les services de « nos » programmeurs pourront nous reprocher d'avoir peu insisté sur ce dernier point, alors que nous avons traité des exercices éloignés des problèmes de la programmation quotidienne, comme P_{14} (transformation d'expression infixe en expression polonaise post-fixée). Nous avons dans un tel cas préféré une

introduction à des concepts importants et relativement difficiles (manipulation d'une pile, manipulation de quantités symboliques, notions élémentaires sur la manière dont fonctionne un compilateur) à l'étude de notions que le programmeur peut retrouver dans un manuel de référence.

Nous ne nous faisons guère d'illusions sur la portée réelle d'un enseignement tel que celui que nous dispensons à des « promotions » d'une trentaine de nouveaux programmeurs dans un centre de calcul qui en compte déjà plusieurs centaines. A cet égard, et quelle que soit la satisfaction des élèves à l'issue du cours, le succès ou l'échec de notre expérience ne sera apparent qu'au vu de leur comportement ultérieur dans la jungle de la programmation. Il est clair que l'on ne pourra transformer sérieusement les habitudes de programmation que par une action de formation difficile et de longue haleine.

BIBLIOGRAPHIE

1. ANSI : *FORTREV*, projet de norme FORTRAN, document X3J3/69 75-07-24, 1975.
2. F. T. BAKER, *Chief Programmer Team Management of Production Programming*, IBM Systems Journal, vol. 11, n° 1, 1972.
3. R. CONWAY et D. GRIES, *An introduction to Programming, A Structured Approach Using PL/1 and PL/C*, Winthrop, 1973.
4. E. DIJKSTRA, *A Short Introduction to the Art of Programming*, Rapport EWD 316, The Eindhoven, 1971.
5. E. W. DIJKSTRA, *Notes on Structured Programming*, in *Structured Programming* (DAHL, DIJKSTRA, HOARE), Academic Press, 1973.
6. E. W. DIJKSTRA, *The Humble Programmer*, vol. CACM 15, n° 10, 1972.
7. E. W. DIJKSTRA, *On the Teaching of Programming, i. e. on the Teaching of Thinking*, Rapport EWD 473, 1975.
8. R. W. FLOYD, *Introduction to Programming and the ALGOL W Language*, Stanford University, Computer Science Department, 1970.
9. G. E. FORSYTHE, M. A. MALCOLM et C. B. MOLER, *Computer Methods for Mathematical Computations*, Stanford University, Computer Science Department, 1972.
10. B. W. KERNIGHAN et P. J. PLAUGER, *The Elements of Programming Style*, McGraw-Hill, 1974.
11. M. LUCAS, J. MOSSIÈRE et P. C. SCHOLL, *Initiation à la programmation. Réflexions et propositions*, R.A.I.R.O., vol. B-3, mars 1975, p. 5-25.
12. E. SATTERTHWAITE, *Source Language Debugging Tools*, Stanford University, Computer Science Department, 1975.
13. N. WIRTH, *Systematic Programming: An Introduction*, Prentice-Hall, 1973.