L'APPRENTISSAGE DE LA PROGRAMMATION

Claude PAIR

Centre de Recherche en Informatique de Nancy
Chargé de Mission auprès du Secrétaire d'État
à l'Enseignement Technique

L'APPRENTISSAGE DE LA PROGRAMMATION

Claude PAIR

Centre de Recherche en Informatique de Nancy
Chargé de Mission auprès du Secrétaire d'État
à l'Enseignement Technique

1. INTRODUCTION

Dans un récent article [8], j'ai soutenu qu'il valait mieux ne pas chercher à enseigner la programmation, mais plutôt aider les élèves à construire les cadres mentaux nécessaires à l'exercice de cette activité.

La discipline à enseigner n'est pas alors le seul point de départ ; le professeur doit aussi être conscient du cadre préexistant chez l'élève. L'apprentissage a pour but de faire évoluer ce cadre pour le rapprocher plus ou moins de celui de l'expert, à travers des stades intermédiaires : la représentation que l'élève a de l'activité qu'il exerce va être confrontée à des expériences qui pourront la confirmer, l'enrichir, la remettre en question : il ne s'agit que du processus d'assimilation-accommodation de Piaget [11] !

Pour l'enseignant, construire un apprentissage demande de faire des hypothèses sur les stades intermédiaires et sur leur ordre, ce qui permet de proposer des activités pour favoriser le passage de chaque stade au suivant.

On ne peut pour cela partir uniquement de l'état actuel de la discipline : son histoire peut aider. Il faudrait aussi avoir quelque notion de psychologie de la programmation [4] ; ce champ, qui présente à la fois un caractère modélisateur et un caractère expérimental, s'est sans doute constitué trop indépendamment de la science de la programmation [1] : il est aujourd'hui nécessaire de combler le fossé des deux côtés.

Cet article veut être un essai en ce sens. Rédigé à l'occasion de l'exposé que j'ai été invité à faire au récent colloque sur la didactique de l'informatique (septembre 1988, Paris), il vise à préciser mon précédent article [8].

2. ACQUERIR UN SAVOIR-FAIRE

La programmation est un savoir-faire. Or, l'acquisition des savoir-faire est un aspect un peu méprisé dans l'enseignement, ce qui explique d'ailleurs que l'enseignement technique soit trop souvent considéré comme une formation de seconde zone.

Pourtant, on développe beaucoup de savoir-faire dans l'enseignement ou en dehors : la lecture, le calcul algébrique, la démonstration, la conduite des automobiles... A ces savoir-faire sont associés des savoirs : la prononciation de "ou" ; $(a + b)^2 = a^2 + 2\ ab + b^2$; la somme des angles d'un triangle est égale à 180° , voire "de p et p −−> q on peut déduire q" ; le code de la route... Il est évidemment très insuffisant de posséder ces savoirs pour exercer les savoir-faire correspondants ; pourtant, le plus souvent, on ne va guère plus loin que d'enseigner les savoirs et de faire faire des exercices ou de montrer à l'élève le résultat de l'activité du professeur ; et, au mieux, on lui donne de vagues conseils comme de faire un plan pour une dissertation. C'est qu'en réalité on cherche à faire acquérir moins des méthodes que des réflexes.

Pour la programmation, nous sommes plus exigeants : nous voulons que les élèves apprennent comment procéder. Et, comme cette visée est nouvelle pour eux, les élèves résistent. Mais là réside sans doute l'originalité de l'enseignement de la programmation et donc ce qu'elle peut apporter à la formation générale.

Tous les informaticiens ne sont cependant pas persuadés qu'il faut faire acquérir une méthode de programmation et, chez eux aussi, cet enseignement a parfois mauvaise presse. Certains formateurs croient encore qu'il suffit d'enseigner la syntaxe d'un langage avec quelques commentaires sur chaque instruction. D'autres, plus savants, pensent que tout est dit lorsque l'étudiant connaît les automates finis ou la sémantique dénotationnelle.

Cependant, le résultat de l'activité de programmation n'est pas seulement un programme ; mais, pour rendre celui-ci communicable, c'est aussi un dossier qui explique plus ou moins comment le programme a été obtenu. La situation est très différente quand on veut seulement développer des réflexes, comme pour la lecture, la conduite auto, ou même la démonstration mathématique où très souvent l'exposition est très différente de la création.

Comment donc acquérir un savoir-faire ? Qu'est-ce dans ce cas qu'un cadre mental caractérisant un des stades intermédiaires par lequel va passer l'apprenant (à rapprocher de ce que J. M. Hoc nomme "un système de représentation et de traitement" [4]) ?

La première composante paraît bien être une représentation, un modèle, un concept, du type de production à obtenir : un texte Basic, appréhendé comme une suite d'instructions, ou un arbre programmatique, ou dans d'autres domaines le plan d'une maison avec ses pièces, ses murs,... pour un architecte, ou une serrure pour un serrurier,...

Il faut aussi disposer d'outils (intellectuels ou matériels) pour engendrer ce résultat par transformations successives, en spécialisant progressivement le concept pour parvenir à une réalisation qui correspond à un cahier des charges. Par exemple, pour engendrer un programme Basic, un tel outil peut être d'ajouter une instruction à la suite du texte déjà construit.

Ce couple concept du résultat-outils de génération peut s'appeler une compétence, en un sens voisin de celui que donnent à ce mot les linguistes dans le savoir-faire qui permet de produire les phrases d'une langue [2]. Il peut aussi être rapproché des types abstraits de l'informatique [3], avec un concept de l'ensemble des valeurs et leurs générateurs.

Comme en linguistique, la compétence présente en général deux aspects : l'un est matériel ou syntaxique (un programme Basic conçu comme une suite d'instructions et construit en ajoutant successivement les instructions) ; l'autre est sémantique, relatif à la fonction à remplir, à la signification du résultat à construire : c'est le second aspect qui va justifier les transformations successives aboutissant au résultat.

Encore faut-il, s'il ne s'agit pas d'une activité réflexe, savoir quel outil appliquer à chaque étape. D'où une troisième composante du cadre mental, que l'on peut appeler une méthode.

En résumé, ces cadres mentaux qui vont constituer les stades successifs d'un apprentissage peuvent être vus comme composés :

- d'une compétence, elle-même formée d'un concept des productions possibles et d'outils de génération, avec leurs aspects syntaxique et sémantique.
- d'une méthode pour choisir les outils.

3. UN APPRENTISSAGE DE LA PROGRAMMATION

En nous appuyant sur la définition qui précède, nous allons dans cette partie proposer une hypothèse sur les stades par lesquels peut passer un élève ou un étudiant ; cette hypothèse est fondée sur l'observation, mais elle devrait être confirmée expérimentalement ; elle est aussi fondée sur l'évolution historique de la programmation [8]. Nous indiquons en outre des motivations et des activités susceptibles de faire évoluer l'apprenant d'un cadre mental au suivant.

3.1. Faire avec une machine

L'expérience préalable d'un débutant, qui peut servir de précurseur à l'apprentissage [12], est celle de l'emploi d'une machine classique, ou encore d'un ordinateur utilisé en commande directe, par exemple pour faire du traitement de texte ou du dessin,... Il s'agit

alors d'obtenir un résultat, d'un type précis, par exemple une figure géométrique. La compétence est orientée vers ce type de résultat : notion de figure, outil comme ajouter un trait ou déplacer le crayon ; quant à la méthode, elle consiste à transférer ce qu'on faisait à la main dès l'instant qu'il y a une correspondance suffisamment forte entre les outils utilisés à la main et les ordres donnés à la machine.

Le renforcement de ce stade pré-programmatoire ne doit pas être négligé, surtout avec de jeunes enfants. Le langage Logo s'y prête bien [10]

3.2. Faire faire

La motivation de préparer le travail lorsqu'on ne dispose pas de machine, et de conserver les ordres à donner de manière à obtenir une exécution automatique, conduit au stade suivant.

Ici, on ne produit plus directement un résultat, mais un texte de programme, dont la sémantique est un calcul. L'outil de génération consiste à ajouter successivement les instructions. Et la méthode induite est toujours de se mettre à la place de l'ordinateur.

La progression par rapport au stade précédent porte essentiellement sur le type de la production à obtenir : un texte (un programme) décrivant un calcul qui conduira (ultérieurement) au résultat. C'est un saut considérable par rapport au stade précédent, et on prendra progressivement conscience qu'un tel programme ou un tel calcul peut conduire à des résultats de nature extrêmement variée.

Mais là aussi existe un précurseur, un transfert à partir d'un cadre mental préexistant : celui qui consiste à décrire une action pour fixer la conduite d'une autre personne. C'est à ce stade que se placent un certain nombre de manuels élémentaires de programmation lorsque, dans leur premier chapitre, ils font appel aux recettes de cuisine pour introduire la notion de programme.

3.3. Décrire un ensemble de calculs

Au stade précédent, un programme décrit un unique calcul. On va maintenant passer au cas où le programme décrit plusieurs calculs , voire une infinité, en fonction des données. Pour y introduire, on peut faire lire aux élèves des programmes déjà écrits qui décrivent tout un ensemble de calculs en fonction de données, à l'aide d'instructions conditionnelles et itératives. On peut aussi, si l'apprenant possède une expérience mathématique, introduire des paramètres dans les programmes (procédures) conservés.

Ici encore, c'est surtout le type de production qui change, moins dans sa syntaxe que dans sa sémantique. L'outil qui consiste à ajouter une interaction est seulement étendu parce qu'on enrichit le répertoire des interactions avec la conditionnelle et l'itération.

Dans un premier temps, on conserve le même genre de méthode qui consiste à penser l'ensemble de calculs séquentiellement en transférant ce que l'on ferait à la main. Ce n'est possible que si on se place dans des cas simples :

- pour les conditionnelles, une seule instruction dans chacun des cas, ou un petit nombre d'instructions de manière à pouvoir éviter de sortir de la pensée séquentielle ;

- pour les itérations, une forme du genre

  si ... alors recommencer telle partie

  qui n'est pas très éloigné de

  répéter... jusqu'à...;

  on évite ici les itérations imbriquées.

3.4. Décrire aussi le programme

Ici, le précurseur peut être scolaire : on ne rend pas un devoir qui ne comporte pas d'explications permettant de se faire comprendre, au moins par le maître. De même, le texte d'un programme n'est pas un résultat suffisant.

Cela modifie la composante syntaxique de la compétence, et on peut ici proposer une forme de dossier de programmation comportant :

- une spécification : quelles sont les données, quels sont les résultats, comment les résultats sont définis à partir des données ; remarquons que l'introduction de la spécification fait aussi évoluer l'aspect sémantique de la notion de programme, pour la rapprocher de la notion de fonction.
- un lexique des identificateurs parallèle à la suite des instructions:

lire nom	nom (chaîne) : nom de l'ouvrier
lire nh	nh (entier):nombre d'heures de travail
lire sh	sh (réel) : salaire horaire
sal := nh*sh	sal (réel) : salaire
écrire sal	

Un tel tableau renforce l'aspect sémantique de description de programme comme une fonction, et on peut faire remarquer qu'il n'existe que trois types d'instruction "efficaces" (en dehors des compositions comme conditionnelle et itération) : donner une valeur à une variable par lecture ou par affectation, écrire un résultat. La notion de variable peut, selon l'âge des élèves, admettre un précurseur en mathématiques, ou au contraire servir de précurseur aux variables mathématiques. Enfin, ce tableau peut aider à une vérification de correction : aucune variable ne peut être utilisée, à droite d'une

affectation ou dans une écriture, avant d'avoir reçu une valeur et donc sans se trouver déjà dans le lexique.

3.5. Concevoir et décrire de manière structurée

Les restrictions faites au 3.3 avaient pour but de limiter la complexité pour permettre de se contenter de l'outil de génération qui consiste à ajouter les instructions l'une après l'autre, et donc de garder une conception séquentielle.

Lorsque l'on passe à des exemples plus complexes, on ne peut conserver la linéarité de la conception qu'en évitant d'entrer d'emblée dans les détails, et donc en nommant certaines parties qui seront détaillées ultérieurement. Conserver ainsi une pensée séquentielle apparaît plus efficace que le passage à deux dimensions matérialisé par un organigramme.

On modifie donc le concept de la description du programme à produire : elle devient arborescente plutôt que linéaire. Pour le programme, c'est l'occasion d'introduire la notion de procédure. On ajoute surtout un outil complètement nouveau : décomposer une partie d'abord envisagée (et spécifiée) globalement.

Il existe ici aussi un précurseur scolaire : le plan, recommandé pour écrire une rédaction par exemple ; malheureusement, beaucoup d'élèves ne sont pas persuadés que ce soit nécessaire, ni même utile !

3.6. Introduire des variables informatiques

Dès qu'une itération contient une affectation, on sort de la relation univoque entre nom et valeur, qui était celle du cadre mental créé en 3.4. On rencontre là une vraie difficulté car cette notion de variable informatique n'a pas de véritable précurseur dans l'expérience de l'élève [8].

On peut distinguer trois étapes de complexité croissante dans la mise en place de cette notion :

a) il n'y a pas coexistence de deux valeurs de la variable dans une exécution de l'itération : c'est par exemple le cas si on calcule successivement le salaire des ouvriers d'une entreprise : pour chaque ouvrier, on lit son nom, son nombre d'heures nh, son salaire horaire sh et on calcule le salaire sal, sans référence à l'ouvrier précédent ; dans une description structurée, on programme séparément l'obtention du salaire d'un ouvrier et nom, nh, sh, sal y sont des variables mathématiques.

b) on emploie directement l'ancienne valeur pour trouver la nouvelle ; ainsi dans l'exemple précédent si on veut trouver la somme des salaires :

s := s + sal ;

dans l'esprit du programmeur, il n'y a que coexistence fugitive de l'ancienne et de la nouvelle valeur, mais on rencontre la nécessité d'initialiser la variable.

c) à l'intérieur de l'itération, on utilise dans des instructions différentes l'ancienne et la nouvelle valeur ; ici, on a toute chance de se tromper si on ne s'oblige pas à distinguer entre l'une et l'autre en les notant différemment dans le tableau des instructions, par exemple x- pour l'ancienne valeur et x pour la nouvelle ; on devra alors vérifier que x- n'est plus utilisée après qu'on a donné sa valeur à x (par lecture ou affectation) : de plus, l'emploi de x- déclenche la mise en place d'une initialisation.

On peut reprendre ici l'exemple de [8] : 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. Si on ne prend pas soin de distinguer entre les deux valeurs d'une commune qui coexistent dans l'itération, on a toute chance de parvenir au programme inexact :

```
pour i de 1 à 95 faire
s := 0
répéter          lire c
                 s := s + population de c
jusqu'à département de c = i  ou  fin de fichier
écrire i, s
```

Il fallait au contraire distinguer deux valeurs de c : le test de fin doit porter sur la commune qui suit celle dont on ajoute la population :

s := s + population de c⁻

doit précéder lire c

qui permet le test d'arrêt de l'itération interne :

département de c = i.

Il faudrait donc initialiser c, mais seulement pour le premier département, donc avant la boucle externe.

Dans le cadre mental, cela conduit à la modification du concept de variable, qui s'accompagne d'un changement dans la méthode : le concept de variable informatique n'ayant pas de précurseur, il ne suffit plus de "se mettre à la place de l'ordinateur" ; la proposition de notation qui précède permet cependant de faire plutôt porter la modification

sur la description du programme, pour ne pas trop s'écarter de la méthode suivie jusque là, en se contentant de prendre quelques précautions de précision et de vérification.

3.7. Calculer sur des objets "abstraits"

Jusque là, la méthode est donc toujours la même : se mettre à la place de l'ordinateur ; et les replâtrages qu'elle a subis pour surmonter les difficultés rencontrées (3.3., 3.5., 3.6.) la renforcent plutôt ; c'est surtout la "compétence" qui varie. Aussi la rencontre avec des problèmes où cette méthode n'est d'aucun secours pour démarrer constitue-t-elle un choc.

Un premier cas est celui où la difficulté consiste en le traitement d'objets différents de ceux que manipule le langage de programmation utilisé, et en général plus complexes. Cela conduit à enrichir à la fois le résultat à produire, les outils de production et la méthode. Pour le résultat, il s'agit dans la description du programme d'accepter des formes intermédiaires qui portent sur des objets "abstraits" [6] au sens où le langage ne les connaît pas ; sur ces objets portent des fonctions qui seront définies ultérieurement. Parmi les outils s'introduit la représentation des objets. Dans la méthode se pose alors la question : quand et comment représenter ; et la réponse consiste à représenter le plus tard possible, lorsqu'aucune décomposition n'est possible sinon, et de se laisser guider par les fonctions à réaliser sur les objets.

Ici, ce n'est donc plus exactement à la place de l'ordinateur que doit se placer le programmeur, mais à la place d'une machine abstraite qu'il convient d'inventer et qui sera ensuite réalisée en représentant ses objets et ses opérations.

3.8. Découvrir un algorithme

Mais il existe des cas où, même éventuellement en considérant une machine abstraite traitant les objets qui figurent dans l'énoncé, on ne sait pas comment démarrer la décomposition du problème. Il faut alors remettre la méthode complètement en question : on ne peut se fonder sur ce que l'on ferait à la place de l'ordinateur ; le guide sera ce que l'on connaît, la spécification, et on sera guidé par le type d'une donnée ou d'un résultat [9]. Un exemple célèbre est celui des tours de Hanoï où c'est la donnée du nombre de disques, un entier n, qui amène à la récurrence ramenant n à n-1.

Cette méthode conduit à la programmation fonctionnelle.

4. CONCLUSION

Cet article veut donner un outil de réflexion - le cadre mental d'un savoir-faire - pour analyser un mode d'apprentissage de la programmation.

La décomposition de l'apprentissage que nous avons indiquée devrait être validée expérimentalement. Elle n'est sans doute pas la seule possible, et un autre choix d'activités pourrait peut-être conduire à économiser des étapes. Par exemple, peut-on, en passant plus rapidement à la programmation fonctionnelle, éviter les difficultés liées aux variables informatiques ? Je l'ai cru longtemps ou, influencé par ce que je savais des modes de programmation les plus efficaces, j'ai voulu le croire. Mais, quel que soit l'âge des apprenants, cela ne s'est guère vérifié, en tout cas sur les problèmes complexes, c'est-à-dire pour lesquels justement une méthode affirmée est utile. Peut-être la programmation par objets [7] pourra-t-elle renouveler la problématique ; cela reste à étudier.

REFERENCES

[1]- J. ARSAC, *La Science Informatique* - Dunod, Paris (1970)

[2]- N. CHOMSKY, *Syntactic structures* - Mouton, La Haye-Paris (1957)

[3]- J.V. GUTTAG, J.J. HORNING, *The algebraic specification of abstract data types* - Acta Informatica 10 (1978), p 27-52

[4]- J.M. HOC, *Psychologie cognitive de la planification* - Presses Universitaires, Grenoble (1987)

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

[6]- B.H. LISKOV, S.N. ZILLES, *Programming with abstract data types* - SIGPLAN Notices 9 (1974), p 50-59

[7]- B. MEYER, *Object-oriented software construction*, Prentice Hall, New-York (1988)

[8]- C. PAIR, *Je ne sais (toujours) pas enseigner la programmation* - Informatiques 2 (1988), p 5-14

[9]- C. PAIR, R. MOHR, R. SCHOTT, *Construire les algorithmes*, Dunod, Paris (1988)

[10]- S. PAPERT, Mindstorms : *Children, Computers and powerful ideas*, Basic Books, New-York (1980)

[11]- J. PIAGET, *L'équilibration des structures cognitives*, P.U.F., Paris (1975)

[12]- N. CHOMSKY, *Syntactic structures* - MOUTON, La Haye-Paris (1957)