Des univers et des outils variés pour commencer à programmer

Ghislaine Dufourd, Jean-François Dufourd

DES UNIVERS ET DES OUTILS VARIÉS POUR COMMENCER À PROGRAMMER

Ghislaine DUFOURD et Jean-François DUFOURD

Centre Informatique et Enseignement
UER Mathématique-Informatique, Université Louis Pasteur
7, Rue René Descartes, F-67084 STRASBOURG Cédex
Tél : 88 41 63 21

RESUME

On dit souvent qu'un bon moyen pour rendre attrayante une initiation à la programmation est de faire résoudre par les élèves des problèmes ludiques ou réels, à l'aide d'environnements matériels et logiciels plus stimulants qu'un langage de programmation classique.

Dans cet article, nous examinons les apports et les exigences de l'utilisation d'outils de programmation de différentes classes. Nous montrons que, malgré le foisonnement des environnements possibles, il existe un cadre conceptuel unique, que l'on peut appréhender, par exemple, avec des univers de types hiérarchisés et la méthode déductive Médée.

Trois exemples concrets d'analyses de problèmes sont proposés dans des domaines différents. Ils sont ensuite programmés en Pascal, Logo et Dbase3.

MOTS-CLES

Didactique de la programmation - Univers - Types hiérarchisés - Méthode déductive - Langages et outils de programmation.

DES UNIVERS ET DES OUTILS VARIÉS POUR COMMENCER À PROGRAMMER

Ghislaine DUFOURD et Jean-François DUFOURD

Centre Informatique et Enseignement
UER Mathématique-Informatique, Université Louis Pasteur
7, Rue René Descartes, F-67084 STRASBOURG Cédex
Tél : 88 41 63 21

1. INTRODUCTION

On prétend souvent que l'initiation à la programmation est plus facile avec certains langages ou environnements de programmation *stimulants* pour les élèves. On le dit de Logo [Papert 81], pour l'école élémentaire, le collège et même après [Shaffer 86], car sa géométrie-tortue permet de construire rapidement des dessins attrayants sur un écran. On le dit aussi des systèmes de gestion intégrés (SGBD, tableurs, grapheurs), notamment pour le lycée [Jamm 85, Bailey 87, Favre 87, Vasseur 87], car ils correspondent bien à des préoccupations issues de la vie courante ou d'autres disciplines.

De tels systèmes permettent en effet de poser et résoudre des problèmes sortant un peu des sentiers battus. Mais encore faut-il savoir ce que cette pratique apporte réellement aux élèves, et ce qu'elle leur demande en contrepartie. Or, il règne souvent une certaine confusion entre des éléments d'appréciation de natures différentes :

- les uns de nature *conceptuelle* : l'univers d'objets et d'opérations dans lequel on évolue, et la résolution de problèmes dans cet univers ;
- les autres de nature *opérationnelle* : les langages et outils supportant cet univers et permettant de traduire les solutions des problèmes.

Dans cet article, nous tentons d'éclairer un peu ce phénomène et de dégager quelques critères d'évaluation. Nous nous plaçons dans la première période d'*initiation* à la programmation, car c'est une phase essentielle, où la vision que l'on donne de l'informatique marque les apprenants pour très longtemps. Nous nous limitons à des *univers prédéfinis* et aux *constructions algorithmiques de base* (séquentielles, conditionnelles, itératives), en vue d'un mode de programmation procédural, fonctionnel ou par objets. C'est dire que nous n'excluons par la récursivité, même si nous n'en parlons

pas ici. En revanche, nous laissons hors de notre champ d'étude la programmation logique, qui semble relever de considérations spécifiques [Arsac 87].

Pour une présentation rigoureuse des notions intervenant dans ce papier, nous nous appuyons sur deux techniques de programmation éprouvées, dont nous utiliserons quelques éléments :

- les *constructions hiérarchiques de types* [Cardelli 85, Goguen 86, Meyer 86] ;
- la méthode de *programmation déductive Médée* [Pair 79, Ducrin 84].

S'il est souhaitable que tout formateur en informatique connaisse les constructions hiérarchiques de types, il est bien sûr hors de question que ces notions soient évoquées avec les débutants. En revanche, la méthode déductive bien maîtrisée, et éventuellement simplifiée, est un excellent candidat comme cadre conceptuel à l'apprentissage [Bana 81, De Bary 87, CIE 87].

Dans la section 2, nous rappelons les principales notions utilisées par la suite sur les constructions de types et sur Médée. Puis, pour mieux cerner les points communs et les particularités des différentes approches, nous examinons successivement trois univers. Dans la section 3, nous évoluons dans les booléens, les nombres et les caractères. Dans la section 4, l'univers est celui de la géométrie-tortue. Dans la section 5, nous nous plaçons dans le cadre des bases de données relationnelles. Dans chaque univers, nous évoquons la résolution des problèmes et la programmation dans un environnement représentatif, comme Pascal, Logo ou Dbase 3. Dans la section 6, nous tentons de dégager les éléments essentiels dans une initiation à la programmation, et discutons les avantages et inconvénients des différentes approches compte tenu de la richesse de l'univers, de la sophistication des outils, du niveau et de la disponibilité des apprenants. La section 7 conclut sur cette étude et ouvre des perspectives de poursuite. Les exemples proposés peuvent être situés vers la fin de la première période d'apprentissage considérée : pour fixer les idées, disons fin de seconde ou début de première, dans l'option informatique des lycées.

2. UN CADRE CONCEPTUEL

Nous rappelons ici succintement les quelques éléments sur les univers et la méthode déductive nécessaires aux développements qui vont suivre.

2.1. Univers d'objets et d'opérations

Un *univers* est un ensemble de types et d'opérations qui rendent compte des objets du domaine dans lequel on travaille.

Un *type* de données est un ensemble d'objets auquel on donne un nom. Par exemple, nous désignons ici par *entier*, *réel* et *chaîne* les ensembles des entiers relatifs,

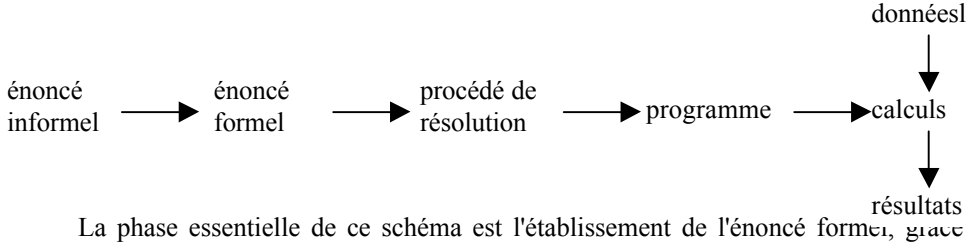
des réels et des chaînes de caractères non bornées. Ce sont des types *de base*. D'autres types sont *construits* à partir d'eux, ou d'autres déjà construits, grâce à des types *génériques* (ou constructeurs de types) paramétrés. Par exemple, nous utiliserons en section 5 le type générique *ensemble de* T, qui désigne l'ensemble des parties du type T. T est ici un paramètre formel qui peut être remplacé par n'importe quel type connu, lors d'une *instantiation*. Une autre manière de construire des types à partir d'autres est l'*héritage*, mais nous ne l'utiliserons pas ici. Ces deux mécanismes permettent de construire des *hiérarchies* de types, traduisant leurs dépendances mutuelles, dans un *réseau sémantique* [Barr 83].

Des *opérations* peuvent être définies entre les types d'un univers. Certaines d'entre elles sont dites *génériques* car elles s'appliquent à des types génériques. Ainsi, l'opération union ensembliste **U** s'applique à deux ensembles de la même instance du type générique *ensemble de* T. D'autres opérations peuvent être *héritées* par le mécanisme d'héritage de types.

La notion d'univers est en fait celle d'*algèbre hétérogène* des mathématiciens. Elle est largement utilisée aujourd'hui en informatique, notamment par le biais de langages à types génériques, comme Ada, ou de langages orientés objets, comme Smalltalk. Le langage Eiffel présenté dans [Meyer 86] est un bon compromis entre ces deux tendances.

2.2. La méthode déductive Médée

Cette méthode s'inscrit dans le processus de résolution de problèmes suivant [Pair 79] :



La phase essentielle de ce schéma est l'établissement de l'énoncé formel, grâce à l'écriture de *définitions algorithmiques* de trois catégories, pour lesquelles nous utiliserons ici les notations suivantes :

-*définition simple* de x :

x = < expression simple >

-*définition conditionnelle* de x :

x = *si* < condition > *alors* <expression 1 > *sinon* < expression 2 >

-*définition itérative* d'une suite (x_i) :

x_f = *tant que* < condition > *répéter* x_i = < expression >

Les deux premières formes ont une signification évidente. La troisième est la définition d'une suite (x_i) de terme final x_f, depuis un premier terme x_0, jusqu'à ce que `< condition >` soit fausse. Quand `< expression >` contient x_{i-1}, la suite est *récurrente*, et x_0 doit être défini à part (initialisation). Pour alléger, x_i et x_{i-1} seront notés respectivement x et @x.

Le schéma itératif

x_f = n *fois répéter* x_i = `< expression >`

est aussi utilisé quand on connaît d'avance le nombre des x_i. Quand *résultat* remplace x ou xf, il s'agit d'une *écriture* simple, conditionnelle ou itérative, sur un organe approprié, ici l'écran. Quand une `< expression >` est remplacée par *donnée* [`< chaîne >`], il s'agit d'une *lecture* par un organe approprié, ici le clavier, après envoi sur l'écran d'un message `< chaîne >` à l'utilisateur.

De manière pratique, l'énoncé formel est présenté comme un ensemble de *modules* (dont l'un est un *programme*), avec un en-tête et deux parties : à gauche un *lexique* et à droite les définitions algorithmiques des éléments du lexique, écrites de manière désordonnée. Une colonne centrale permet ensuite d'ordonner les définitions pour obtenir un *algorithme* (cf. figure 3.1.).

Quand une définition nécessite des définitions auxiliaires, celles-ci sont regroupées dans un module qui remplace une des expressions dans l'un des schémas ci-dessus (cf. figure 3.1.). Plusieurs définitions peuvent être fusionnées : on le verra notamment pour les itérations.

Ce mode d'expression est adapté à une analyse structurée, modulaire, descendante ou ascendante, et déclarative : on cherche à exprimer des *définitions* d'objets plutôt que des *calculs* [Pair 79]. On donne un nom différent à deux valeurs différentes selon le principe d'*affectation unique*, ce qui conduit à utiliser des indices pour les suites, ou les notations simplifiées présentées précédemment [Arsac 77, Ashcroft 77]. Enfin, le lexique est un moyen de documentation très important. On trouve tous les détails sur cette méthode dans [Ducrin 84], ainsi qu'une synthèse des travaux de recherche en cours à son sujet dans [Greco 86].

3. UN UNIVERS TRADITIONNEL : BOOLEENS, NOMBRES ET CHAINES

3.1. Objets et opérations

Nous nous plaçons ici dans un univers, traditionnel pour la programmation, qui nous servira à illustrer la section 2. Les types sont *booléen*, *entier*, *caractère* et *chaîne*, désignant respectivement l'ensemble des booléens, des entiers relatifs, des caractères et des chaînes de caractères non bornées. Nous les considérons ici comme des types de base, avec des opérations prédéfinies.

Dans les booléens, les opérations sont les opérations logiques habituelles, et, dans les entiers, les opérations arithmétiques entières. Dans les caractères, il n'y a que les constantes-caractères (opérations à 0 variable) et, dans les chaînes, seulement la chaîne vide et la concaténation. D'autres opérations sont mixtes : ainsi car(s, i) donne le ie caractère de la chaîne s, et x = y, où x et y sont d'un même type quelconque, est un *booléen.*

3.2. Résolution de problèmes

L'exemple suivant illustre la section 2.2. :

- *Enoncé informel :* vérifier qu'une expression arithmétique commençant au début d'une chaîne non vide, et terminée par un " ; ", est bien parenthésée : une ") " correspond toujours à une " (" qui la précède.

- *Enoncé formel et algorithme :* ils sont donnés à la figure 3.1.

```
                                  programme PARENTHESAGE
d : entier : différence entre              6 résultat = si d_f = 0
  le nombre  de " ( " et                         alors écrire('BIEN PARENTHESEE')
  le nombre de " ) ",pour la                     sinon écrire('MAL PARENTHESEE')
  portion d'expression examinée ;
                                           5 d_f, i_f, c_f = tant que c # ' ; 'et d = 0
c : caractère : ie caractère de s ;                répéter UNCAR
i : entier : place de c dans s ;           2 d_0 = 0
UNCAR : module : définit d, i              4 c_0 = car (s, i0)
  et c courants ;                          3 i_0 = 1
s : chaîne : à examiner ;                  1 s = donnée['ENTRER UNE CHAINE :']
                                        module UNCAR
                                           1 d = si @c = ' ( ' alors @d+1
                                                 sinon si @c = ' ) ' alors @d-1
                                                   sinon @d
                                           2 i = @i+1
                                           3 c = car (s, i)
```

Figure 3.1. Algorithme du problème de parenthésage

A la figure 3.1., les suites (d), (i) et (c) sont définies conjointement dans une *fusion* d'itérations. On voit par ailleurs apparaître le mécanisme de "lecture à l'avance" du caractère c dans s, une difficulté clairement identifiée en algorithmique.

3.3. Outils de programmation

Ici, de nombreux langages de programmation peuvent être utilisés de manière plus ou moins commode : LSE, Basic, Pascal, Logo, etc. A titre d'illustration, nous présentons le programme en T-Pascal [Borland 84] à la figure 3.2. On y supprime bien sûr tous les

indices et " @ ", UNCAR devient un bloc, et les instructions sont réordonnées. La fonction copy remplace car.

```
program PARENTHESAGE ;
var     s : string [255] ;
        d, i : integer ;
        c : char ;
begin
        write ('ENTRER UNE CHAINE : ') ; readln (s) ;
        d : = 0 ;
        i : = 1 ;
        c : = copy (s, i, 1) ;
        while (c < > ' ; ' ) and (d > = 0) do
        begin {UNCAR}
                if c = ' ( ' then d : = d+1
                else if c = ' ) ' then d : = d-1 ;
                i : = i+1 ;
                c : = copy (s, i, 1)
        end ; {UNCAR}
        if d = 0 then writeln ('BIEN PARENTHESEE')
        else writeln ('MAL PARENTHESEE')
end.
```

Figure 3.2. Programme Pascal du parenthésage

4. UN UNIVERS GRAPHIQUE : LA GEOMETRIE-TORTUE

4.1. Objets et opérations

Fondamentalement, l'*univers géométrie-tortue* [Abelson 84] est une *extension* de l'univers de la section 3 à un nouveau type d'objets : écran, muni d'une tortue. Ainsi, les caractéristiques d'une tortue sur un écran (figure 4.1.) sont : sa position (x, y) dans un plan euclidien (abscisse, ordonnée), son cap a (angle en degrés par rapport au nord), son apparition ou non sur l'écran, sa couleur, son crayon baissé ou non, etc. Ces éléments s'ajoutent aux spécifications générales d'un écran : ligne de partage texte-graphique, couleur du fond, etc. Toutes ces caractéristiques sont retrouvées grâce à des opérations agissant sur un écran. D'autres opérations engendrent des écrans, éventuellement à partir d'autres écrans, à l'aide de certains paramètres, notamment pour faire évoluer la tortue.

Dans la vision habituelle de la géométrie-tortue, il n'y a bien sûr qu'un seul écran-et une seule tortue-considéré comme *global*, et de ce fait, n'apparaissant pas comme opérande ou résultat des opérations écran ou tortue. Celles-ci sont alors des *procédures* qui *modifient* l'écran-tortue par effet de bord. Il en est ainsi des *primitives* utilisées par la

suite : VE [resp. VT] vide l'écran graphique [resp. texte] ; LC [resp. BC] lève [resp. baisse] le crayon ; FPOS fixe la position de la tortue selon les coordonnées en argument ; AV [resp. TD] la fait avancer [resp. tourner à droite] d'un segment [resp. d'un angle] de mesure donnée en paramètre ; POS-OPT est une fonction renvoyant le couple des coordonnées du point désigné par le crayon optique, ORIGINE remet la tortue en position initiale.

Pour une question de commodité, les noms de ces primitives sont ceux de Logo. Mais nous utilisons ces opérations dans le paragraphe suivant comme des (fonctions-) procédures habituelles, avec les notations usuelles en mathématiques, notamment avec les paramètres entre parenthèses.

4.2. Résolution de problèmes

A titre d'exemple, nous présentons l'analyse d'un problème simple de dessin interactif.

-*Enoncé informel* : écrire un programme pour tracer plusieurs triangles isométriques ABC, rectangles en A, ce point étant situé au-dessus de l'hypoténuse BC, elle-même parallèle à l'axe des x. On donne le nombre de triangles, la mesure du côté AB et celle de l'angle en B. Le point B de chaque triangle est fixé par pointage au crayon optique (figure 4.1.).

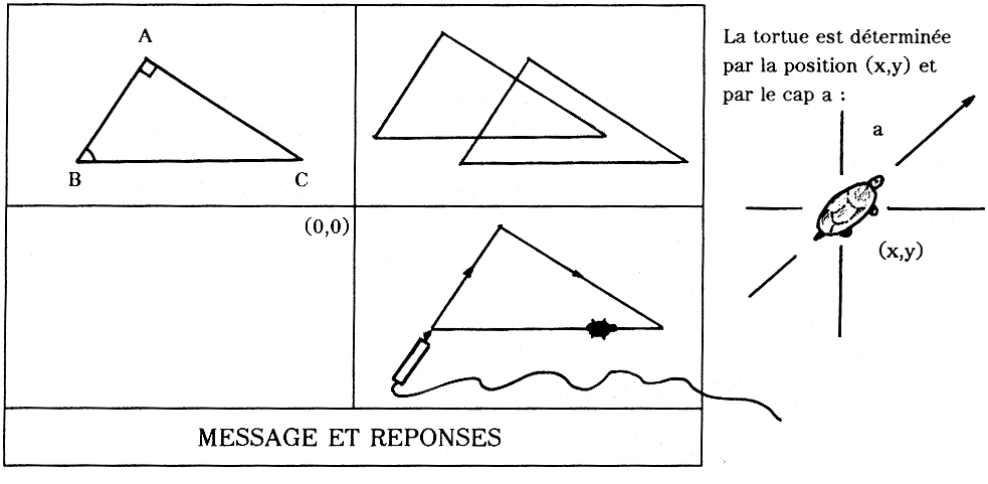


Figure 4.1. Tortue et dessin de triangles rectangles isométriques

- *Enoncé formel et algorithme* : on les trouve à la figure 4.2.

```
                    programme TRIANGLES : affiche un nombre fixé de triangles
n : entier : nombre de triangles ;          1 résultat = VE, VT
UNTRIANGLE : module : dessin                6 résultat = n fois répéter UNTRIANGLE
  d'un triangle ;                           5 n = donnée ['NOMBRE DE TRIANGLES :']
B : entier : mesure de l'angle B ;          2 B = donnée ['MESURE DE L'ANGLE B :']
AB : entier : mesure du côté AB ;           3 AB = donnée ['MESURE DU COTE AB :']
AC : réel : mesure du côté AC ;             4 AC = AB x tg (B)
                                  module UNTRIANGLE
PB : couple : position de B                 2 résultat = LC, FPOS (PB), BC,
                                              TD (90-B), AV (AB), TD (90),
                                              AV (AC), FPOS (PB), LC, ORIGINE
                                            1 PB =POS-OPT['POINTER LA PLACE DE B ']
```

Figure 4.2. Algorithme pour dessiner des triangles

4.3. Outils de programmation

La notion de géométrie-tortue est associée étroitement au langage Logo [Papert 81] : la figure 4.3. est la traduction de l'algorithme précédent en Logo-Plus [ACT 85], sous la forme de deux procédures. Mais d'autres langages permettent aujourd'hui de travailler en géométrie-tortue, T-Pascal par exemple.

```
POUR TRIANGLES
VE  VT
ECRIS [MESURE DE L'ANGLE B :]  DONNE "B LM
ECRIS [MESURE DU COTE AB :]  DONNE "AB LM
DONNE "AC PROD  :AB DIV SIN  :B COS  :B
ECRIS [NOMBRE DE TRIANGLES :]  DONNE "N LM
REPETE  :N [UNTRIANGLE]
FIN

POUR UNTRIANGLE
ECRIS [POINTER LA PLACE DE B] DONNE "PB  POS-OPT
LC  FPOS :PB   BC  TD  90  -  :B  AV :AB  TD  90
AV  :AC  FPOS  :PB  LC  ORIGINE
FIN
```

Figure 4.3. Programme Logo du dessin de triangles

Aussi convient-t-il ici d'affirmer nettement la distinction entre l'univers conceptuel géométrie-tortue et les outils permettant de le mettre en oeuvre, dont Logo n'est désormais que l'un des représentants, même s'il est le plus illustre.

5. UN UNIVERS DE RELATIONS : BASES DE DONNEES

5.1. Objets et opérations

Ici, encore, l'*univers relationnel* apparaît fondamentalement comme une *extension* de celui de la section 3 à de nouveaux types de n-uplets (ou structures) et relations [Dufourd 85]. Ainsi

$$type\ U = <u_1 : U_1 ; ... ; u_n : U_n>$$

est la définition d'un nouveau type U comme produit cartésien (à l'ordre près) des types U_1, ... , U_n, avec les sélecteurs de champs u_1, ... , u_n. Les éléments de U sont des *n-uplets*, ou *structures* (au sens des "records" de Pascal). Alors,

$$type\ rU = ensemble\ de\ U$$

est la définition d'un nouveau type rU comme ensemble des parties de l'ensemble U. Les éléments de rU sont des *relations* ou (sous-) *ensembles* (au sens de Pascal).

Dans cet univers, les *opérations* sur les n-uplets sont : construction d'un n-uplet, projection sur une (ou plusieurs) composantes. Celles sur les relations sont les opérations ensemblistes habituelles : union, intersection, complémentaire, cardinal, etc., et les opérations de l'algèbre relationnelle : sélection, projection, composition, division, etc., [Date 81]. Ainsi, à la section 5.2., la sélection dans une relation r des n-uplets vérifiant un prédicat $p(u_1, ... , u_n)$ est notée $r \mid p(u_1, ... , u_n)$, la différence ensembliste est notée - , {x} est la relation réduite au n-uplet x et card(r) est le cardinal de la relation r. Il est aussi possible de parcourir les éléments d'une relation dans un certain ordre, de définir des index, d'accéder à un n-uplet d'index donné, etc., comme on le fait avec les fichiers indexés. Ainsi, fdr(r) est *vrai* ssi le parcours de r est achevé ; prem(r) [resp. suc(r)] renvoie le premier n-uplet [resp. le n-uplet suivant] de la relation r, dans l'ordre des numéros internes croissants. Ces opérations donnent à l'univers relationnel une grande puissance, mais une complexité certaine.

5.2. Résolution de problèmes

Nous nous plaçons dans le cadre d'une application simple de gestion de bibliothèque, avec des relations sur les livres, les élèves, les emprunts, etc. [Jamm 85]. Nous ne travaillons ici que sur les livres, avec deux transactions élémentaires.

- *Enoncé informel :* créer une transaction pour y compter les ouvrages de Victor Hugo publiés chez Gallimard, et une autre pour y effacer les n-uplets correspondant aux éditions Maspéro (sans aucun contrôle).

- *Enoncé formel et algorithme :* le type des livres et celui des relations sur les livres peuvent être décrits par :

```
type livre = <    n : entier ; {n° de livre}
                  a : chaîne ; {auteur}
                  t : chaîne ; {titre}
                  e : chaîne ; {éditeur}
                  ... > ;
type rlivre = ensemble de livre ;
```

Nous donnons aux figures 5.1. et 5.2. deux versions de chacune des transactions : COMPTAGE et EFFACEMENT.

programme COMPTAGE : version 1

```
rl : rlivre : ensemble des livres          2 résultat =
   de la bibliothèque ;                        écrire (card(rl|a = 'HUGO'
                                                       et e =' GALLIMARD'))
                                           1 rl = donnée
```

programme COMPTAGE : version 2

```
t : entier : nombre des ouvrages           5 résultat = écrire (tf)
   pertinents déjà trouvés ;               4 tf , xf = tant que non fdr(rl) répéter
rl : rlivre : ensemble des livres                t = si a(@x) = 'HUGO'
   de la bibliothèque ;                               et e(@x) = 'GALLIMARD'
                                                    alors @t+1 sinon @t
x : livre : n-uplet courant ;                    x = suc(rl)
                                           3 t0 = 0
                                           1 rl = donnée
                                           2 x0 = prem(rl)
```

Figure 5.1. Deux versions de COMPTAGE

programme EFFACEMENT : version 1

```
rl : rlivre : état courant de              2 rl1 = rl0 - rl0 | e = 'MASPERO'
   l'ensemble des livres ;                 1 rl0 = donnée
```

```
                    programme EFFACEMENT : version 2
rl : rlivre : état courant de                3 rl_f , x_f = tant que non fdr(rl) répéter
   l'ensemble des livres ;                         rl = si e(@x) = 'MASPERO'
x : livre : n-uplet courant ;                         alors @rl - {@x} sinon @rl
                                                   x = suc(@rl)
                                             2 x_0 = prem(rl_0)
                                             1 rl_0 = donnée
```

Figure 5.2. Deux versions d'EFFACEMENT

Les deux premières versions utilisent toute la puissance des primitives du modèle relationnel, alors que les secondes se rapprochent plus d'une gestion de fichier traditionnelle, où l'on observera à nouveau la technique de lecture à l'avance.

5.3. Outils de programmation

Pour un tel univers, il est clair que le meilleur outil d'implantation est un SGBD relationnel. A titre d'illustration, nous donnons à la figure 5.3. la version 1 de COMPTAGE et la version 2 de EFFACEMENT dans le langage de manipulation d'un système très répandu : Dbase 3 [Ashton 85]. Bien sûr n'importe quel SGBD relationnel convient, certains "collant" d'ailleurs mieux à la vue conceptuelle précédente. En effet, dans le langage de Dbase 3, les écritures implicites sont nombreuses, ce qui allège le texte, mais est source de mauvaise interprétation ou d'erreurs de programmation.

```
TEXT COMPTAGE : version 1
ENDTEXT
USE RL
COUNT FOR (A = 'HUGO') .AND. (E = 'GALLIMARD')
RETURN
TEXT EFFACEMENT : version 2
ENDTEXT
USE RL
DO WHILE .NOT. EOF ( )
        IF (E = 'MASPERO')
        DELETE
        ENDIF
        SKIP
ENDDO
RETURN
```

Figure 5.3. Deux transactions en Dbase 3

6. UNE DISCUSSION DES DIFFERENTES APPROCHES

- Des problèmes "ludiques" ou "réels" plus stimulants,

De toute évidence, plus l'univers dans lequel on évolue est composé d'objets complexes et d'opérations puissantes, plus il est facile de donner des problèmes "ludiques" ou "réels" stimulants pour des élèves.

Cependant, dans cette recherche de problèmes intéressants, le risque de *débordement* est grand. Ainsi par exemple, on a rapidement besoin, dans un univers graphique-tortue, du théorème de Pythagore ou des fonctions trigonométriques, comme dans l'exercice de la section 4, où il est même question de "couple". Chacun a en mémoire les abus auxquels a conduit dans l'école élémentaire la tentative d'inculquer des notions hors de portée et "hors programme" à de jeunes enfants.

Il faut tempérer aussi l' idée de *problème réel*, car un "vrai" problème réel est souvent fort complexe. Ainsi, mettre à jour des volumes de données importants, comme dans l'exercice de la section 5, ne suffit pas à faire vraiment de la gestion. Les choses sont en fait plus compliquées : une base de données réelle est souvent une collection de relations, et pas une relation isolée. Elle doit en outre satisfaire en permanence des *contraintes d'intégrité*. Il ne faut pas, par exemple, supprimer des ouvrages qui ont été empruntés par des élèves sans effacer aussi ces emprunts, après éventuellement avoir fait revenir les livres. Dans le cas contraire, on risque la pire chose pour une base de données : *l'inconsistance*. Ainsi, le vrai problème prend tout de suite d'autres proportions.

- une stimulation qui se paie par un apprentissage et des manipulations plus longs,

Plus l'univers est complexe, plus les *prérequis* des élèves sont importants. Nous le disions précédemment pour le graphique-tortue, et il en est de même pour le relationnel, si l'on veut résoudre des problèmes intéressants.

Le plus souvent, l'immersion dans un univers nouveau doit s'accompagner d'une période d'*apprentissage* des objets et opérations, par nature assez proche de celui des mathématiques. Alors, on utilise l'informatique comme un auxiliaire pour cet apprentissage, en manipulant, par exemple en mode de bureau, les outils qui seront utilisés par la suite pour l'implantation.

Ainsi, on a beaucoup dit et écrit sur l'intérêt et les difficultés de l'initiation des jeunes enfants au graphique-tortue pour se répérer et se mouvoir dans le plan. Aujourd'hui, la *tortue-plancher* est devenue un outil pédagogique banal de l'école élémentaire, et l'exécution directe de procédures Logo en mode bureau permet d'appréhender cet univers de manière systématique.

La bonne compréhension des primitives relationnelles passe aussi par des manipulations directes des commandes d'un SGBD. C'est d'ailleurs nécessaire pour construire rapidement des relations-jeux d'essai afin de tester des transactions comme celles de la section 5. Cela signifie de nombreuses heures consacrées à cet apprentissage du système, qui doit être pensé par l'enseignant de manière méthodique et progressive.

En revanche, on conviendra que l'initiation est particulièrement rapide pour un univers numérique, un peu plus longue pour les chaînes, même si, là encore, les manipulations en mode bureau sont indispensables. Enfin, les deux autres univers étant des extensions de celui-ci, il faut, de toute manière, passer par lui pour comprendre la suite...

- d'où un risque d'oublier l'essentiel...

Nous entrons ici dans le débat classique sur l'informatique, voire sur l'enseignement en général : quelles sont les places respectives de la *théorie* et de la *pratique* ? Mettre l'accent sur les concepts (univers, méthode de résolution de problèmes) risque de faire oublier la mise en oeuvre réelle (machine, langage, système). Inversement, trop manipuler peut contrarier la bonne structuration de la pensée [Henderson 87] et les nécessaires évolutions ultérieures.

Il faut être clair : une initiation à l'algorithmique et à la programmation demande, pour être efficace, de s'intéresser à *tout le processus* conduisant de l'énoncé d'un problème à sa résolution concrète sur ordinateur. Et si l'élève doit acquérir à la fois des concepts et des outils, l'enseignant doit répondre à la difficile question du partage d'un temps, nécessairement limité, entre théorie et pratique. On voit donc que le choix de tel ou tel univers et environnement de programmation doit être pensé aussi dans cette perspective.

- et de sombrer dans la technique,

Trop se focaliser sur un outil particulier, se laisser entraîner à en explorer les moindres recoins peut cacher ce qui est fondamental au profit de détails anecdotiques et passagers. Le cas des SGBD est typique : en moins de cinq ans, on en a vu défiler dans l'enseignement au moins une dizaine de bonne qualité. Ils reposent tous sur les mêmes concepts, mais diffèrent en surface, un peu sur le plan des fonctionnalités, surtout sur celui des commandes ou des interfaces (clavier, souris, fenêtrage, graphisme, etc.). Et le mouvement s'accélère avec la diffusion des systèmes intégrés, dont on annonce presque chaque semaine un nouveau représentant. Bien connaître la théorie aide à comparer et à changer rapidement de produit.

En étant trop près des outils, on finit par se persuader que ce que l'on fait avec est vraiment différent, et relève de concepts et de méthodes d'analyse particuliers. Ainsi, que de dithyrambe à propos de la "démarche Logo", et quelle exagération à propos des "langages de 4e génération" des SGBD !

- alors, qu'au fond, tout repose sur la notion d'univers...

Les notions d'*univers*, de *types* d'objets et d'*opérations* offrent un cadre conceptuel unifié [Cardelli 85, Goguen 86], qui permet de comprendre, classer, hiérarchiser les éléments intervenant en analyse-programmation.

Pourtant, la notion de type a été très discutée en informatique, notamment par rapport aux langages applicatifs non typés, comme Lisp. Or, le programmeur Lisp n'applique pas ses fonctions au hasard sur n'importe quels objets [Cardelli 85]. Et le plus souvent à présent, avant de se lancer dans la programmation, il commence par se définir une "couche orientée-objets", lui permettant de décrire son univers de travail sous la forme d'un réseau sémantique avec des types (ou classes) hiérarchisés.

Ainsi, tout environnement utilisé pour l'initiation à la programmation repose sur un univers dont la description relève toujours des *mêmes notions fondamentales*. Cet univers peut ensuite être rendu opérationnel par l'intermédiaire de différents outils ou langages de programmation. Il importe donc que l'enseignant ait toujours à l'esprit cette dichotomie entre univers et environnement concret, et qu'il communique cette idée à ses élèves, en évitant bien sûr de "théoriser" à ce sujet.

- et un même schéma de résolution de problèmes.

En effet, quels que soient l'univers dans lequel on évolue et l'environnement de programmation utilisé, la résolution de problèmes repose sur les mêmes considérations. Nous l'avons montré dans trois univers distincts, mais nous pourrions le faire de manière plus générale. Il est possible de suivre les mêmes étapes du *même grand schéma de résolution*, différentes voies étant bien sûr possibles à un niveau plus fin [Gram 86].

Une des phases essentielles est celle de l'écriture d'un *énoncé formel*. Nous l'avons présenté ici sur des problèmes simples, à l'aide de la méthode déductive, ce qui est raisonnable au niveau d'une initiation, à condition de ne pas exiger "trop de formalisme" [Ferchichi 87]. On constate alors qu'il s'agit toujours de donner des *définitions algorithmiques* de trois grandes classes : définitions simples, conditionnelles (y compris récursives) et itératives. Cette méthode nous semble bien adaptée à l'initiation, mais nous n'en faisons pas un dogme : ce que nous disons ici de l'algorithmique peut aussi être illustré dans d'autres cadres évoqués dans [Arsac 87], peut-être de manière moins précise cependant.

Il importe enfin de distinguer nettement cette phase de celle d'*implantation* dans un environnement particulier, avec un langage de programmation où entrent en jeu des considérations techniques qui n'ont pas grand'chose à voir avec la résolution de problèmes proprement dite.

7. CONCLUSION

Nous avons voulu lutter ici contre la tendance toujours tenace à vouloir "apprendre à programmer en ..." Basic, Logo, Pascal, Lisp, Ada, APL, Dbase, etc., plutôt que de chercher à "apprendre à programmer" tout court. En effet, s'il est important de connaître des outils de programmation, il ne l'est pas moins de savoir poser, formuler et résoudre des problèmes dans un univers clairement identifié. Et l'on peut trouver une manière unique de formuler précisément les énoncés algorithmiques dans tout univers.

Cette aptitude correspond à un premier niveau d'analyse-programmation, qu'il sera ensuite possible d'enrichir en restant toujours dans le même cadre conceptuel [Dufourd 88]. Les niveaux suivants consistent alors à apprendre à *compléter* des univers par de nouvelles opérations (fonctions et procédures), puis à les *concevoir* et à les *construire* soi-même (construction de types par généricité et héritage).

Nous avons examiné ici trois univers, mais ce que nous avons dit s'applique sans trop de difficultés à des univers sous-jacents à d'autres types d'applications et outils de programmation, comme les fichiers, les tableurs, voire certains outils de géométrie assistée par ordinateur [Allard 86], ou même de C.A.O. [Autodesk 85]. Des expériences sont actuellement tentées dans l'académie de Strasbourg avec toute une palette d'outils, et Médée comme moyen d'expression "pivot" pour définir des objets. Seule reste encore à l'écart la programmation logique, qui, malgré des tentatives encourageantes, ne paraît pas s'accompagner encore d'une méthode de programmation satisfaisante.

De telles expériences doivent se multiplier, d'une part pour mieux asseoir les concepts, les méthodes et les techniques de la programmation, et d'autre part pour les enseigner à des publics d'élèves variés, plus faciles à stimuler dans certains univers et avec certaines classes d'outils. Mais il faut toujours garder en tête la profonde unité des concepts de l'analyse-programmation !

REFERENCES.-

[Abelson 84] H. ABELSON & A.A. DI SESSA, *Turtle Geometry*, MIT Press, 6th ed., 1984.

[ACT 85] ACT INFORMATIQUE, *Logo Plus™ - Manuel de Références*, Hatier, 1985.

[Allard 86] J.-C. ALLARD & AL., *Géométriciel : Un Outil pour la Géométrie au Lycée et au Collège*, IREM, Grenoble, 1986.

[Arsac 77] J. ARSAC, *La Construction de Programmes Structurés*, Dunod, 1977.

[Arsac 87] J. ARSAC, *Des Pédagogies pour l'Option Informatique des Lycées*, *in* Pratiques et Savoir-faire des Elèves de l'Option Informatique, MEN-DLC, CRDP, Poitiers, 1987, p. 1-13.

[Ashcroft 77] E.A. ASHCROFT & W.W. WADGE, *LUCID , a Nonprocedural Language with Iteration*, CACM, 20 (71), 1977, p. 519-526.

[Ashton 85] ASHON-TATE, *Dbase III* : *Plus™- Manuel de Références*, 1985.

[Autodesk 85] A.G. AUTODESK, *Autocad™, v. 2.1., User Ref. Pub.* 106-006F, 1985.

[Bailey 87] G. BAILEY, *Spreadsheets and Databases - Alternatives to Programming for Non Computer Science Majors*, 18th ACM Tech. Symp. on Comp. Sc. Educ., St Louis, 1987, p. 499-503.

[Bana 81] C. BANA, G. DUFOURD, B. JARAY & J.-P. FINANCE, *Using Computer Science in Order to Teach Mathematics*, IFIP WCCE 81, Lausanne, 1981, p. 171-176.

[Barr 83] A. BARR & E.A. FEIGENBAUM, *The Handbook of Artificial Intelligence*, vol. 1, Pitman, 1983.

[Borland 84] BORLAND-FRACIEL, *Turbo-Pascal™- Manuel de Références*, 1984.

[Cardelli 85] L. CARDELLI & P. WEGNER, *On Understanding Types, Data Abstraction, and Polymorphism*, ACM-Comp. Surveys, 17 (4), 1985, p. 471-522.

[CIE 87] CENTRE INFORMATIQUE ET ENSEIGNEMENT, *Algorithmique et Programmation*, vol. 1 et 2, supports de cours, ULP Strasbourg, 1987.

[Date 81] C.-J. DATE, *An Introduction to Database Systems*, 3rd ed., Addison-Wesley, 1981.

[De Bary 87] C. DE BARY, *Un système Interactif de Spécification destiné à l'Enseignement*, Journées AFCET-GROPLAN 1987, BIGRE-GLOBULE, 55, 1987, p. 47-58.

[Ducrin 84] A. DUCRIN, *Programmation*, vol. 1 et 2, Dunod, 1984.

[Dufourd 85] J.-F. DUFOURD, *Types Abstraits et Bases de Données Relationnelles*, TSI, 4 (4), 1985, p. 339-349.

[Dufourd 88] J.-F. DUFOURD,*Vers un Cadre Unique pour Spécifier et Construire des Programmes*, TSI, 7 (3), 1988, pp. 281-293.

[Favre 87] R. FAVRE-NICOLIN & J.-B. MARANINCHI, *Tableur et Récursivité en classe de Seconde*, *in* Pratique et Savoir-faire des Elèves de l'Option Informatique, MEN-DLC, CRDP, Poitiers, 1987, p. 82-88.

[Ferchichi 87] A. FERCHICHI & A. JOUA, *Teaching First Year Programming : a Proposal*, ACM-SIGCSE, 19 (3), p. 48-52.

[Goguen 86] J.-A. GOGUEN, *Rewriting and Interconnecting Software Components*, IEEE-Computer, 18 (2), 1986, p. 16-28.

[Gram 86] A. GRAM, *Raisonner pour Programmer*, Dunod, 1986.

[Greco 86] GRECO Programmation du C.N.R.S., *Bilans et Perspectives*, Talence, 1986, p. 45-47.

[Henderson 87] P. B. HENDERSON, *Modern Introductory Computer Science*, 18th ACM Tech. Symp. on Comp. Sc. Educ., St Louis, 1987, p. 183-189.

[Jamm 85] F. JAMM, *Démarche Pédagogique pour une Initiation à l'Algorithmique à travers l'Utilisation d'un SGBD*, projet péd. CIE, ULP Strasbourg, 1985.

[Meyer 86] B. MEYER, *Genericity versus Inheritance*, 1st ACM Symp. on Obj. Orient. Syst., Lang. and Appl., Portland, 1986, p. 391-405.

[Pair 79] C. PAIR, *La Construction des Programmes*, RAIRO Informatique, 13 (2), 1979, p. 113-137.

[Papert 81] S. PAPERT, *Jaillissement de l'Esprit*, trad. franç., Flammarion, 1981.

[Shaffer 86] D. SHAFFER, *The Use of Logo in an Introductory Computer Science Course*, ACM-SIGCSE, 18 (4), 1986, p. 28-31.

[Vasseur 87] P. VASSEUR, *Les Progiciels dans l'Enseignement de l'Informatique*, Note tech., CFIAP, Poitiers, 1987.