AFFECTER OU NE PAS AFFECTER ?

Guillaume CONNAN
Irem de Nantes

Résumé : L'algorithmique fait une entrée remarquée dans le nouveau programme de Seconde. Il y est fait clairement mention de l'apprentissage de l'affectation, des entrées-sorties et des boucles comme allant de soi, ce qui sous-entend que seule existe la programmation impérative. Cependant la programmation fonctionnelle est une alternative très proche des mathématiques et de leur enseignement dans le secondaire. Nous verrons dans quelle mesure l'enseignement de l'algorithmique via les langages fonctionnels peut être enrichissant et complémentaire d'une approche purement impérative.

Merci à Viviane Durand-Guerrier pour sa relecture qui a permis d'enrichir et recadrer cet article.

1. Conventions

L'objet de cet article n'est pas d'envisager ces deux modes de programmation avec le point de vue de l'informaticien cherchant la méthode la plus rapide mais avec celui d'un professeur *de mathématiques* s'adressant à des élèves *débutants* dans ce domaine.

Ce court article ne fournit pas d'activités clé-en-main mais donne des éléments pour alimenter la réflexion du professeur, en particulier en montrant que la programmation fonctionnelle, souvent méconnue (même si elle est implicite dans un tableur ou peut être utilisée en primaire avec la tortue Logo), est très proche des mathématiques.

Nous ne parlerons pas d'un logiciel en particulier. Cependant, afin d'illustrer de manière pratique notre propos, nous utiliserons XCAS, logiciel multifonctions spécialement conçu pour être utilisé en classe de mathématiques dont l'usage aujourd'hui est assez largement répandu et CAML, logiciel utilisant la programmation fonctionnelle (même s'il permet aussi d'avoir recours à un style impératif).

2. Programmation impérative et affectations

2.1 *Aspect technique*

La programmation impérative est la plus répandue (Fortran, Pascal, C,...) et la plus proche de la machine réelle. Son principe est de modifier à chaque instant l'état de la mémoire via les affectations. Historiquement, son principe a été initié par John VON NEUMANN et Alan TURING.

Par exemple, lorsqu'on entre avec XCAS :

```
a:=2
```

cela signifie que la case mémoire portant le petit « fanion » a contient à cet instant la valeur 2. On peut changer le contenu de cette case mémoire un peu plus tard :

```
a:=4
```

et cette fois la case a contient la valeur 4.

Le caractère chronologique de ce type de programmation est donc primordial. Regardons la suite d'instructions suivante :

```
a:=1;
b:=2;
a:=a+b; // maintenant a contient 3
b:=a-b; // maintenant b contient 3-2=1
a:=a-b;  // maintenant a contient 3-1=2
```

Cela permet d'échanger les contenus des cases mémoire a et b.

Si l'on change l'ordre d'une quelconque des trois dernières lignes, les valeurs présentes dans les cases mémoires nommées a et b vont être différentes. Si l'on regarde les deux dernières lignes, elles affectent apparemment la même valeur (a – b) à a et à b et pourtant c'est faux car les variables a et b ne contiennent pas les mêmes valeurs selon le moment où elles sont appelées.

Il faut donc être extrêmement attentif à la chronologie des instructions données. C'est un exercice intéressant en soi mais pas sans danger (c'est d'ailleurs la source de la plupart des « bugs » en informatique).

La motivation première de ce type de fonctionnement a été pratique : les ordinateurs étant peu puissants, le souci principal du concepteur de programme a longtemps été d'économiser au maximum la mémoire. Au lieu d'accumuler des valeurs dans différents endroits de la mémoire (répartition spatiale), on les stocke au même endroit mais à différents moments (répartition temporelle). C'est le modèle des machines à états (machine de TURING et architecture de VON NEUMANN) qui impose au programmeur de connaître à tout instant l'état de la mémoire centrale.

2.2 *Exploitation mathématique possible*

En classe de mathématique, cet aspect technique nous importe peu. Il nous permet cependant d'illustrer dynamiquement la notion de fonction :

```
y:=x^2+1;
x:=1;  y;  // ici y=2
x:=-5;  y;  // ici y=26
x:=1/2; y;  // ici y=5/4
```

La fonction $f : x \mapsto x^2 + 1$ dépend de la variable x. On affecte une certaine valeur à la variable ce qui détermine la valeur de son image. Il est toutefois à noter qu'en fait, on a travaillé ici sur des expressions numériques et non pas des fonctions.

On peut enchaîner des affectations :

```
y:=x^2+1;
z:=y+3;
x:=1;  y; z;  //  ici y=2 et z=5
x:=-5  y; z;  // ici y=26  et z=29
x:=1/2; y; z;  // ici y=5/4 et z=17/4
```

et illustrer ainsi la composition de fonctions.

Avec un peu de mauvaise foi (car le cas est simple et artificiel) mais pas tant que ça

puisque de tels problèmes sont source de milliers de bugs sur des milliers de programmes plus compliqués, nous allons mettre en évidence un problème majeur de ce genre de programmation quand on n'est pas assez attentif.

On crée à un certain moment une variable a à laquelle on affecte une valeur :

```
a:=2;
```

On introduit plus tard une procédure (une sorte de fonction informatique des langages impératifs) qu'on nomme f et qui effectue certains calculs en utilisant a :

```
f(x):={
  a:=a+1;
  retourne(x+a)
}
```

Alors $f(5)$ renvoie 8 mais si on redemande $f(5)$ on obtient à présent 9 et si on recommence on obtient 10, etc. car à chaque appel de la procédure f, la variable a est modifiée : ici $2 \times f(5)$ apparaît aux yeux des élèves comme étant différent de $f(5) + f(5)$!

On peut donc construire un objet qui *ressemble* à une fonction d'une variable (mais qui n'en est évidemment pas) et qui à une même variable renvoie plusieurs valeurs.

En effet, une procédure ne dépend pas uniquement des arguments entrés et donc peut changer de comportement au cours du programme : c'est ce qu'on appelle en informatique des *effets de bord*.

La *procédure f* qui a l'*apparence* d'une fonction dépendant de la seule variable x (*f(x)*:=...) est en fait une fonction de deux variables au sens mathématique du terme :

$$g : (x,a) \to x+a$$

Le premier appel de $f(5)$ est en fait $g(3,5)$ et le deuxième $g(4,5)$, etc. mais on a pu le faire en utilisant une forme qui *laisse à penser* que f est une fonction qui renvoie une infinité d'images différentes d'un même nombre. Informatiquement, a et x jouent des rôles différents puisque x est un argument de la procédure f et a est une variable globale donc « extérieure » à f (un *paramètre*). D'ailleurs, XCAS envoie un avertissement pour nous le rappeler :

```
// Warning: a, declared as global
variable(s) compiling f
```

2.3 *Programmation fonctionnelle*

Les langages fonctionnels bannissent totalement les affectations. Ils utilisent des fonctions *au sens mathématique du terme*, c'est-à-dire qu'une fonction associe une unique valeur de sortie à une valeur donnée en entrée. Les fonctions peuvent être testées une par une car *elles ne changent pas selon le moment où elles sont appelées dans le programme.* On parle de *transparence référentielle*. Le programmeur n'a donc pas besoin de savoir dans quel état se trouve la mémoire. C'est le logiciel qui s'occupe de la gestion de la mémoire.

Historiquement, ce style de programmation est né du lambda-calcul développé par Alonzo CHURCH juste avant la publication des travaux d'Alan TURING. Ce principe d'utilisation de fonctions au sens mathématique est perturbant pour des programmeurs formés aux langages impératifs, mais ne devraient pas troubler des mathématiciens...

La *récursivité* est le mode habituel de programmation avec de tels langages : il s'agit

de fonctions qui s'appellent elles-mêmes. Pendant longtemps, la qualité des ordinateurs et des langages fonctionnels n'était pas suffisante et limitaient les domaines d'application de ces langages.

On utilise également en programmation fonctionnelle des *fonctions d'ordre supérieur*, c'est-à-dire des fonctions ayant comme arguments d'autres fonctions : cela permet par exemple de créer des fonctions qui à une fonction dérivable associent sa fonction dérivée.

2.3.1 *Variable, affectation, déclaration, valeur.*

Nous allons préciser maintenant comment est traitée une variable dans les deux modes de programmation étudiés.

En programmation fonctionnelle, une variable est un nom pour une *valeur* alors qu'en programmation impérative une variable est un nom pour une *case mémoire*. En programmation fonctionnelle, on ne peut pas changer la valeur d'une variable. Nous parlerons dans ce cas de *déclaration de variable* pour la distinguer de l'*affectation de variable* des langages impératifs.

Une *variable* en programmation fonctionnelle correspond à une *constante* en programmation impérative. Par exemple, si nous écrivons dans CAML :

```
# let x=3;;
```

puis

```
# let x=4;;
```

on ne modifie pas la variable x mais on en crée une nouvelle qui porte le même nom. L'ancienne variable x est maintenant masquée et inaccessible. « Différence subtile pour informaticien coupant les cheveux en quatre » me direz-vous... eh bien non car cela peut créer des surprises si l'on n'en est pas conscient.

Considérons les déclarations suivantes sur CAML :

```
# let x=3;; (* le # est créé automatiquement par CAML pour signifier qu'il attend nos instructions. Les ;; indiquent que nous avons fini de lui « parler » *)
# let f = function
      y -> y+x;;
```

Si l'on veut la valeur de l'image de 2 par la *fonction* f :

```
# f(2);;  (* ce que nous entrons *)
- : int = 5  (* la réponse de CAML *)
```

en effet, 2 + 3 = 5 . Observez maintenant ce qui se passe ensuite :

```
# let x=0;;
x : int = 0

# f(2) ;;
- : int = 5
```

On a modifié la valeur de x et pourtant f(2) ne change pas. En effet, on a défini f en utilisant le nom x qui vaut 3 lors de la définition de f. Plus tard, on redéclare la variable x avec la valeur 0. On n'a pas modifié la valeur ! de l'ancien identificateur x et donc f(2) vaut toujours 5. Ainsi x est un nom pour la valeur 3 et f un nom pour l'objet «function y -> y + x » c'est-à-dire «function y -> y + 3 ».

Observons maintenant ce que cela donne en programmation impérative avec XCAS.

```
x:=3;
```

```
f:=y->y+x;
```

Les objets sont affectés. XCAS nous renvoie un message précisant qu'il comprend que x est une variable globale dans la définition de la *procédure* f.

```
// Warning: x, declared as global variable(s)
f(2);
                    5
```

Maintenant, ré-affectons x :

```
x:=0;
f(2);
                    2
```

Cette fois on a *ré-affecté* la case mémoire x qui contient maintenant 0 : son ancien contenu est écrasé et c'est comme s'il n'avait jamais existé. La procédure f utilise donc ce qu'elle trouve dans la case mémoire x au moment où on l'appelle et f(! 2) peut donc prendre des valeurs différentes au cours du temps.

Dans toute la suite de cet article, on distinguera donc l'*affectation de variable* (en tant que modification du contenu d'une case mémoire étiquetée) de la programmation impérative et la *déclaration de variable* de la programmation fonctionnelle qui donne un nom à une valeur pour le confort de lecture de l'utilisateur mais c'est la valeur qui est importante.

2.3.2 *Un premier exemple d'algorithme récursif*

Si nous reprenons notre exemple précédant, nous pouvons faire exprimer aux élèves que l'entier suivant n est égal à 1+ l'entier suivant $n - 1$: on ne sort pas des limites du cours de mathématiques et on n'a pas besoin d'introduire des notions d'informatique en travaillant sur cette notion.

On peut écrire un algorithme ainsi :

si $n = 0$ **alors**
1
sinon
1+ successeur de $n - 1$

La récursivité n'est pas l'apanage des langages fonctionnels. Par exemple XCAS permet d'écrire des programmes récursifs.

En français :

```
successeur(k):={
 si k==0 alors 1
 sinon 1+successeur(k-1)
 fsi // indique la fin du test si...alors...sinon
}:;
```

En anglais :

```
 successeur(k):={
 if(k==0)then{1}
 else{1+successeur(k-1)}
}:;
```

Que se passe-t-il dans le cœur de l'ordinateur lorsqu'on entre « successeur(3) » ?

On entre comme argument 3 ;

Comme 3 n'est pas égal à 0, alors successeur(3) est stocké en mémoire et vaut 1+successeur(2) ;

Comme 2 n'est pas égal à 0, alors successeur(2) est stocké en mémoire et vaut 1+successeur(1) ;

Comme 1 n'est pas égal à 0, alors successeur(1) est stocké en mémoire et vaut 1+successeur(0) ;

Cette fois, successeur(0) est connu et vaut 1 ;

successeur(1) est maintenant connu et vaut 1+successeur(0) c'est-à-dire 2 ;

successeur(2) est maintenant connu et vaut 1+successeur(1) c'est-à-dire 3 ;

successeur(3) est maintenant connu et vaut 1+successeur(2) c'est-à-dire 4 : c'est fini !

C'est assez long comme cheminement mais ce n'est pas grave car *c'est l'ordinateur qui effectue le « sale boulot »* ! Il stocke les résultats intermédiaires dans une *pile* et n'affiche finalement que le résultat. Comme il calcule vite, ce n'est pas grave. L'élève ou le professeur ne s'est occupé que de la définition récursive (mathématique !) du problème.

XCAS calcule facilement « **successeur(700)** » mais le calcul de « **successeur(7000)** » dépasse les possibilités de calcul du logiciel.

Eh oui, un langage impératif ne traite pas efficacement le problème de pile de la récursion. C'est pourquoi, pendant longtemps, la récursion (donc également les langages fonctionnels dont c'est le mode de calcul prédominant) a été taxée d'inefficacité en informatique. Les étudiants des années 70, 80, 90 ont pratiqué l'informatique avec cet adage en tête et l'ont transmis s'ils sont devenus enseignants eux-mêmes. Nous allons voir que ce manque d'efficacité a aujourd'hui disparu.

Remarque : *par défaut, XCAS s'arrête au bout de 50 niveaux de récursion. Pour aller plus loin, il faut cliquer sur* config *à côté de «* save *». Régler alors la fenêtre «* recurs *» à 1000 par exemple*

Utilisons à présent le langage fonctionnel OCAML, développé par l'INRIA. Même s'il intègre des aspects impératifs pour faciliter l'écriture de certains algorithmes, il est en premier lieu un langage fonctionnel qui en particulier gère très efficacement la mémoire de l'ordinateur pour éviter sa saturation lors d'appels récursifs.

Le programme s'écrit comme en mathématique (mais en anglais...) :

```
# let rec successeur(k)=
   if k=0 then 1
  else 1+successeur(k-1);;
```

Alors par exemple :

```
# successeur(36000);;
- : int = 36001
```

Mais

```
# successeur(3600000);;
Stack overflow during evaluation
(looping recursion?).
```

La *pile* où sont stockés les résultats intermédiaires créés par la récursion est en effet saturée.

Aujourd'hui, les langages fonctionnels sont très efficaces et gèrent de manière intelligente la *pile*. Certains langages le font automatiquement. D'autres ne seront « efficaces » que si la récursion est *terminale*, c'est-à-dire si l'appel récursif n'est pas *enrobé* dans une autre fonction. Dans l'exemple précédent, ce n'est pas le cas car l'appel récursif « successeur(k-1)» est *enrobé* dans la fonction $X \to 1 + X$. On peut y remédier en introduisant une fonction intermédiaire qui sera récursive terminale :

```
# let rec successeur_temp(k,resultat)=
```

```
if k=0 then resultat
else successeur_temp(k-1,resultat+1);;
```

Ici l'appel récursif à successeur_temp est direct. On appelle ensuite cette fonction en prenant comme résultat de départ 1 :

```
# let successeur_bis(k)=
  successeur_temp(k,1);;
```

Alors :

```
# successeur_bis(360000000);;
- : int = 360000001
```

Il n'y a donc aucun problème pour traiter 360 millions d'appels récursifs !

Dans un premier temps, on peut laisser de côté ce problème de récursion terminale au lycée et se contenter de travailler sur de petits entiers. Ainsi, on peut choisir de travailler avec CAML ou XCAS par exemple, même si ce dernier n'est pas un langage fonctionnel.

2.4 *Un premier exemple de fonction d'ordre supérieur*

Définissons une fonction qui à deux fonctions f et g associe sa composée $f \circ g$:

```
# let compose=function
  (f,g) -> function x->f(g(x));;
```

CAML devine le type des objets introduits en répondant :

```
val compose :
('a -> 'b) * ('c -> 'a) -> 'c -> 'b = <fun>
```

Ceci signifie que compose est une fonction (le <fun> final) qui prend comme arguments deux fonctions (la première qui transforme une variable de type a en une variable de type b et la deuxième qui transforme une variable de type c en une variable de type a) et qui renvoie une fonction qui transforme une variable de type c en une variable de type b. En résumé, on peut schématiser ce qu'a compris CAML :

$g : C \longrightarrow A$ et $f : A \longrightarrow B$ donc $f \circ g : C \longrightarrow B$

Créons deux fonctions numériques :

```
# let carre = function
  x -> x*x;;
# let double = function
  x -> 2*x;;
```

Composons-les :

```
# compose(double,carre)(2);;
- : int = 8
# compose(carre,double)(2);;
- : int = 16
```

En effet, le double du carré de 2 est 8 alors que le carré du double de 2 vaut 16...

On peut cependant composer des fonctions non numériques.

Soit longueur fonction qui renvoie le nombre d'éléments d'une liste (cf section 4) :

```
# let rec longueur = function
  | [] -> 0
  | tete::corps -> 1 + longueur(corps);;
```

Soit rebours la fonction qui crée la liste décroissante des entiers de n à 1 :

```
# let rec rebours(n) =
  if n=0 then []
  else n::rebours(n-1);;
```

Il s'agit de fonctions qui à une liste associent un entier ou l'inverse. On peut malgré tout les composer avec la même fonction *compose*. On entre une liste de chaînes de caractères :

```
# compose(rebours,longueur)
  ([«a»;»b»;»c»;»d»;»e»;»f»;»g»;»h»]);;
```

et on obtient une liste d'entiers :

```
- : int list = [8; 7; 6; 5; 4; 3; 2; 1]
```

On peut faire la même chose avec XCAS :

```
compose(f,g):={
  x->f(g(x))
}
```

Puis par exemple :

```
compose(x->x^2,x->2*x)(2)
```

Un exemple encore plus puissant sera vu à la section 4.1.

3. Deux manières d'aborder un même problème

3.1 *Les tours de Hanoï*

3.1.1 *Le principe*

Ce casse-tête a été posé par le mathématicien français Édouard LUCAS en 1883. Le jeu consiste en une plaquette de bois où sont plantés trois piquets. Au début du jeu, n disques de diamètres croissant de bas en haut sont placés sur le piquet de gauche. Le but du jeu est de mettre ces disques dans le même ordre sur le piquet de droite en respectant les règles suivantes :

• on ne déplace qu'un disque à la fois ;

• on ne peut poser un disque que sur un disque de diamètre supérieur.

Essayez avec 2 puis 3 disques... Nous n'osons pas vous demander de résoudre le problème avec 4 disques !

3.1.2 *Non pas comment mais pourquoi : version récursive*

En réfléchissant récursivement, c'est enfantin !... Pour définir un algorithme récursif, il nous faut régler un cas simple et pouvoir simplifier un cas compliqué.

• *cas simple* : s'il y a zéro disque, il n'y a rien à faire !

• *simplification d'un cas compliqué* : nous avons n disques au départ sur le premier disque et nous savons résoudre le problème pour n-1 disques. Appelons A, B et C les piquets de gauche à droite. Il suffit de déplacer les n-1 disques supérieurs de A vers B (hypothèse de récurrence) puis on déplace le plus gros disque resté sur A en C. Il ne reste plus qu'à déplacer vers C les n - 1 disques qui étaient en B. (hypothèse de récurrence).

On voit bien la récursion : le problème à n disques est résolu si on sait résoudre le cas à $n - 1$ disques et on sait quoi faire quand il n'y a plus de disques.

On sait pourquoi cet algorithme va réussir mais on ne sait pas comment s'y prendre étape par étape. On va donc appeler à l'aide l'ordinateur en lui demandant d'écrire les mouvements effectués à chaque étape. On commence par créer une fonction qui affichera le mouvement effectué :

```
let mvt depart arrivee=
print_string
```

```
(«Deplace un disque de la tige «^depart^»
vers la tige «^arrivee);
print_newline();;
```

Le programme proprement dit :

```
let rec hanoi a b c= function
| 0 -> ()  (*0 disque : on ne fait rien*)
| n -> hanoi a c b (n-1);
    (*n – 1 disques sont placés dans l'ordre
    de a vers b*)
  mvt a c;    (*on déplace le disque
              restant en a vers c*)
hanoi b a c (n-1) (*n-1 disques sont
    placés dans l'ordre de b vers c*);;
```

Les phrases entre (* et *) sont des commentaires. Par exemple, dans le cas de 3 disques :

```
# hanoi «A» «B» «C» 3;;
Deplace un disque de la tige A vers la tige C
Deplace un disque de la tige A vers la tige B
Deplace un disque de la tige C vers la tige B
Deplace un disque de la tige A vers la tige C
Deplace un disque de la tige B vers la tige A
Deplace un disque de la tige B vers la tige C
Deplace un disque de la tige A vers la tige C
- : unit = ()
```

3.1.3 *Non pas pourquoi mais comment : version impérative*

Prenez un papier et un crayon. Dessinez trois piquets A, C et B et les trois disques dans leur position initiale sur le plot A. Déplacez alors les disques selon la méthode suivante :

- déplacez le plus petit disque vers le plot suivant selon une permutation circulaire A–C–B–A ;
- un seul des deux autres disques est déplaçable vers un seul piquet possible.

Réitérez (en informatique à l'aide de boucles...) ce mécanisme jusqu'à ce que tous les disques soient sur C dans la bonne position. On voit donc bien ici comment ça marche ; il est plus délicat de savoir pourquoi on arrivera au résultat. Le programme est lui-même assez compliqué à écrire : vous trouverez une version en C (235 lignes...) à cette adresse :

http://files.codes-sources.com/fichier.aspx ?id=38936&f=HANOI%5cHANOI.C

3.2 *Suites définies par une relation* $u_{n+1} = f(u_n)$

3.2.1 *Sans affectations*

La relation $u_{n+1} = 3u_n + 2$ valable pour tout entier naturel n et la donnée de $u_0 = 5$ constituent un algorithme de calcul récursif de tout terme de la suite (u_n).

si $n = 0$ **alors**
uo
sinon
3×u(n-1,uo)+2 **Fonction** u(*n,uo*)

Sa traduction dans des langages sachant plus ou moins traiter de tels algorithmes est directe. Par exemple, en XCAS, cela devient en français :

```
u(n,uo):={
  si n==0 alors uo
  sinon 3*u(n-1,uo)+2
}
```

et en anglais :

```
u(n,uo):={
if(n==0){return(uo)}
else{3*u(n-1,uo)+2}
}
```

En CAML :

```
# let rec u(n,uo)=
  if n=0 then uo
  else 3*u(n-1,uo)+2;;
```

Ici encore, l'accent est mis sur le « pourquoi » et non pas le « comment » même s'il est beaucoup plus simple de le savoir que dans le cas des tours de Hanoï. On illustre le cours de 1ère sur les suites.

3.2.2 *Avec affectations*

Au lieu de stocker de nombreuses valeurs dans une pile, on affecte une seule case mémoire aux valeurs de la suite qui sera *ré-affectée*, au fur et à mesure du déroulement du programme.

Entrées :
n,uo
Initialisation :
temp ← uo
début
pour *k de 1 jusque n* **faire**
temp ← $3 \times$ temp+2
fi**n**

Algorithme 2 : Suite récurrente : version impérative

Cela donne en français avec XCAS :

```
u_imp(n,uo):={
local k,temp;
temp:=uo;
 pour k de 1 jusque n faire
  temp:=3*temp+2;
 fpour; // marque la fin de la boucle
retourne(temp);
}
```

En anglais :

```
u_imp(n,uo):={
local k,temp;
temp:=uo;
for(k:=1;k<=n;k:=k+1){
temp:=3*temp+2;
}
return(temp);
}
```

Dans ce cas, on voit comment faire pour calculer u_n à partir de u_0. La méthode peut paraître moins « élégante » mais elle est peut-être plus « concrètement » assimilable.

3.3 *Calculs de sommes*

On veut calculer la somme des entiers de 1 à n.

3.3.1 *Sans affectation*

Il suffit de remarquer que cela revient à ajouter n à la somme des entiers de 1 à $n-1$ sachant que la somme des entiers de 1 à 1 vaut... 1. L'écriture est donc simple et mathématique :

si $n = 1$ **alors**
1
sinon
1+som_ent(n-1)

En français avec XCAS :

```
som_ent(n):={
  si n==1 alors 1
  sinon n+som_ent(n-1)
  fsi
}:;
```

En anglais :

```
som_ent(n):={
if(n==1){1}
```

```
else{n+som_ent(n-1)}
};;
```

En CAML :

```
# let rec som_ent(n)=
    if n=1 then 1
    else n+som_ent(n-1);;
```

Par exemple :

```
# som_ent(100000);;
- : int = 705082704
```

On peut même généraliser cette procédure à n'importe quelle somme du type $\sum_{k=k_0}^{k=n} f(k)$

```
# let rec som_rec(f,ko,n)=
    if n=ko then f(ko)
    else f(n)+som_rec(f,ko,n-1);;
```

3.3.2 *Avec affectation*

Entrées :
n(entier naturel)
Initialisation :
S ← 0
début
pour *k de 1 à n* **faire**
S ← S+k
retourner *S*
fin

Traduction XCAS en français :

```
som(n):={
local S;
S:=0;
pour k de 1 jusque n faire
  S:=S+k;
fpour;
retourne(S)
}
```

En anglais :

```
 som(n):={
local S;
S:=0;
for(k:=1;k<=n;k++){S:=S+k};
return(S)
}
```

On a une approche physique avec l'algorithme impératif : S vaut 0 au départ, et on demande S en sortie ; S ne vaut donc plus 0 en général... Cela peut être troublant pour des élèves qui ont du mal à résoudre des équations simples et oublient ce qui se cache derrière le symbole « = ». Il y aurait une égalité en mathématiques et une autre en informatique !

Cependant, cette manière de voir peut être plus intuitive : si on considère S comme le solde de son compte en banque, on surveille son compte régulièrement en ajoutant k à chaque visite. Le compte porte le même nom, S, mais son contenu change régulièrement.

La version impérative peut de plus être directement liée à l'écriture :

$$\sum_{k=k_0}^{k=n} k$$

qu'on lit « somme des k pour k variant de 1 à n » où la boucle « pour » apparaît de manière naturelle.

3.4 *Partie entière*

On définit la partie entière d'un réel x comme étant le plus grand entier inférieur à x.

3.4.1 *Sans affectation*

On part du fait que la partie entière d'un nombre appartenant à [0;1[est nulle. Ensui-

te, on « descend » de x vers 0 par pas de 1 si le nombre est positif en montrant que :

$$[x] = 1 + [x - 1]$$

Si le nombre est négatif, on « monte » vers 0 en montrant que :

$$[x] = -1 + [x + 1]$$

L'algorithme peut alors s'écrire :

si *x est positif et x < 1* **alors**
0
sinon
si *x est positif* **alors**
1+partie_entiere(x-1)
sinon
-1+partie_entiere(x+1)

En français avec XCAS :

```
partie_entiere(x):={
 si ((x>=0) et (x<1)) alors 0
 sinon si (x>0)
      alors 1+partie_entiere(x-1)
      sinon -1+partie_entiere(x+1)
      fsi
 fsi
}
```

En anglais :

```
partie_entière(x):={
 if((x>=0) and (x<1)){0}
 else{ if(x>0){1+partie_entiere(x-1)}
      else{-1+partie_entiere(x+1)}
   }
};;
```

En CAML :

```
# let rec partie_entiere (x)=
      if x >= 0. && x < 1.
      then 0.
      else if x > 0.
         then   1. +. partie_entiere (x -. 1.)
      else -.1. +. partie_entiere (x +. 1.);;
```

Remarque : la double esperluette correspond à l'opérateur logique « ET ». CAML travaille avec deux types de nombres : les entiers et les flottants. Les opérations avec les entiers suivent les notations habituelles. Lorsqu'on travaille avec des flottants (c'est le cas ici car x est un réel quelconque) le symbole habituel de l'opération est suivi d'un point.

On notera que l'algorithme récursif s'applique ici à des *réels* et non plus à des *entiers* : on dépasse ainsi le cadre des suites. Pour que l'algorithme fonctionne, on raisonne sur des intervalles et non plus sur une valeur initiale.

Ici, on réfléchit à la notion *mathématique* de partie entière. On met en place des résultats *mathématiques* intéressants.

3.4.2 *Avec affectation*

Pour les nombres positifs, on part de 0. Tant qu'on n'a pas dépassé x, on avance d'un pas. La boucle s'arrête dès que k est strictement supérieur à x. La partie entière vaut donc $k - 1$. Dans le cas d'un réel négatif, il faut cette fois reculer d'un pas à chaque tour de boucle et la partie entière est la première valeur de k à être inférieure à x :

Entrées :
x(réel)
Initialisation :
k ← 0
début
si *x>=0* **alors**

tant que *k*<=*x* **faire**
k ← k+1
retourner *k-1*
sinon
tant que *k*>*x* **faire**
k ← k-1
retourner *k*
fin

Avec XCAS en Français :

```
pe(x):={
local k;
k:=0;
si x>=0 alors
  tant que k<=x faire
    k:=k+1;
  ftantque; // marque la fin de la boucle
  return(k-1);
sinon
  tant que k>x faire
    k:=k-1;
  ftantque;
  retourne(k);
fsi;
}:;
```

Avec XCAS en anglais :

```
pe(x):={
local k;
k:=0;
if(x>=0){
  while(k<=x){k:=k+1};
  return(k-1);
     }
else{
  while(k>x){k:=k-1};
  return(k);
   }
}:;
```

On utilise ici une ré-affectation (k devient $k+1$ ou $k-1$) et une « boucle while » avec un test et une action d'affectation tant que le test est vrai. Il s'agit d'un traitement physique de la mémoire. On perd un peu de vue la notion mathématique, mais on donne une vision dynamique de la recherche de la partie entière : on peut imaginer un petit « personnage » se promenant sur la droite des réels en faisant des pas de longueur 1 et qui continue à avancer *tant qu'* il n'a pas dépassé le réel.

3.5 *Développement en fractions continues*

3.5.1 *Pratique du développement*

Vous connaissez l'algorithme suivant :

$$\begin{aligned} 172 &= 3\times 51 + 19 \\ 51 &= 2\times 19 + 13 \\ 19 &= 1\times 13 + 6 \\ 13 &= 2\times 6 + 1 \\ 6 &= 6\times 1 + 0 \end{aligned}$$

• On peut donc facilement compléter la suite d'égalité suivante :

$$172/51 = 3 + 19/51 = 3 + 1/(51/19) = \\ = 3 + 1/(2+13/19)$$

• Quand tous les numérateurs sont égaux à 1, on dit qu'on a développé 172/51 en fraction continue et pour simplifier l'écriture on note :

$$172/51 = [3 ; 2 ; ...]$$

• Par exemple, on peut développer 453/54 en fraction continue. Dans l'autre sens, on peut écrire [2 ; 5 ; 4] sous la forme d'une fraction irréductible.

Nous allons construire les algorithmes correspondants.

3.5.2 *Sans affectation*

On suppose connus les algorithmes donnant le reste et le quotient d'une division euclidienne.

fc(a,b)

si *b=0* **alors**
retourner *liste vide*
sinon
retourner *[quotient(a,b),fc(b,reste(a,b))]*

En français en XCAS :

```
fc(a,b):={
  si b==0 alors []
  sinon concat(iquo(a,b),fc(b,irem(a,b)))
  fsi
};;
```

En anglais :

```
fc(a,b):={
  if(b==0)then{[]}
  else{concat(iquo(a,b),fc(b,irem(a,b)))}
};;
```

La commande concat(élément,liste) ajoute élément au début de liste. Alors fc(172,51) renvoie [3,2,1,2,6]

En CAML

```
# let rec fc (a,b) =
      if b = 0
       then []
       else  a/b :: fc(b,a-a/b*b);;
```

Pour ajouter un élément à une liste sur OCAML, la syntaxe est élément::liste. L'algorithme est très simple à écrire et à comprendre.

3.5.3 *Avec affectation*

Entrées : 2 entiers a et b
Initialisation :
num ← a
den ← b
res ← reste(num,den)
Liste ← vide
début
tant que *res≠0* **faire**
Liste ← Liste,quotient(num,den)
num ← den
den ← res
res ← reste(num,den)
fin
retourner *[Liste,quotient(num,den)]*

En français avec XCAS :

```
frac_cont(a,b):={
local num,den,res,Liste;
num:=a;
den:=b;
res:=irem(num,den);
Liste:=NULL;
 tantque res>0 faire
   Liste:=Liste,iquo(num,den);
   num:=den;
   den:=res;
   res:=irem(num,den);
 ftantque
retourne([Liste,iquo(num,den)]);
};;
```

En anglais :

```
frac_cont(a,b):={
local num,den,res,Liste;
num:=a;
den:=b;
```

```
res:=irem(num,den);
Liste:=NULL;
while(res>0){
   Liste:=Liste,iquo(num,den);
   num:=den;
   den:=res;
   res:=irem(num,den);
}
return([Liste,iquo(num,den)]);
}:;
```

Ici, la manipulation de plusieurs niveaux d'affectation et d'une boucle while est assez délicate. La chronologie des affectations est primordiale : on ne peut proposer cette version à des débutants...

3.6 *Dichotomie*

Pour résoudre une équation du type $f(x) = 0$, on recherche graphiquement un intervalle $[a,b]$ où la fonction semble changer de signe. On note ensuite m le milieu du segment $[a,b]$. On évalue le signe de $f(m)$. Si c'est le même que celui de $f(a)$ on remplace a par m et on recommence. Sinon, c'est b qu'on remplace et on recommence jusqu'à obtenir la précision voulue.

3.6.1 *Sans affectation*

On appellera *eps* la précision voulue.

si *b-a est plus petit que la précision eps*
alors
(b+a)/2
sinon
si *f(a) et f((b+a)/2) sont de même signe*
alors
dico_rec(f,(b+a)/2,b,eps)
sinon
dicho_rec(f,a,(b+a)/2,eps)

On remarquera ici que la récursion se fait sans faire intervenir d'entier : le test d'arrêt est effectué sur la précision du calcul. On va rajouter justement un compteur pour savoir combien de calculs ont été effectués.

En français avec XCAS :

```
dicho_rec(f,a,b,eps,compteur):={
 si evalf(b-a) sinon si f(a)*f(0.5*(b+a))>0
     alors dicho(f,0.5*(b+a),b,eps,compteur+1)
     sinon dicho(f,a,0.5*(b+a),eps,compteur+1)
      fsi
 fsi
}:;
```

En anglais

```
dicho_rec(f,a,b,eps,compteur):={
if(evalf(b-a)if(f(a)*f(0.5*(b+a))>0)
{dicho(f,0.5*(b+a),b,eps,compteur+1)}
 else{dicho(f,a,0.5*(b+a),eps,comteur+1)}
}:;
```

En CAML :

```
# let rec dicho_rec(f,a,b,eps,compteur)=
   if abs_float(b-.a)
   else if f(a)*.f(0.5*.(b+.a))>0.
then  dicho_rec(f,0.5*.(b+.a),b,eps,comp-
teur+1)
 else  dicho_rec(f,a,0.5*.(b+.a),eps,comp-
teur+1);;
```

ce qui donne :

```
# dicho_rec( (fun x->(x*.x-.2.)),1.,2.,0.00001,0);;
- : float * int = (1.41421127319335938,18)
```

3.6.2 *Avec affectation*

Entrées : une fonction f, les bornes a et b, une précision p
Initialisation : aa ← a, bb ← b
début

tant que *bb-aa>p* **faire**
si *signe de f((aa+bb)/2) est le même que celui de f(bb)* **alors**
bb ← (aa+bb)/2
sinon aa ← (aa+bb)/2
retourner *(aa+bb)/2* fi**n**

Remarque : *on ne peut pas modifier les arguments d'entrée a et b c'est pourquoi on introduit des variables locales aa et bb qui seront ré-affectées au cours de la procédure.*

Avec XCAS, il faut juste ne pas oublier de régler quelques problèmes de précision. On rajoute aussi pour le plaisir un compteur qui nous dira combien de « tours de boucles » le programme aura effectué.

```
dicho(F,p,a,b):={
local aa,bb,k,f;
aa:=a;
bb:=b;
epsilon:=1e-100;
f:=unapply(F,x);
compteur:=0;
 while(evalf(bb-aa,p)>10^(-p)){
   if( f(0.5*(bb+aa))*f(bb)>0 )
     then{bb:=evalf((aa+bb)/2,p+1)}
     else{aa:=evalf((aa+bb)/2,p+1)}
     k:=k+1;
   }
return evalf((bb+aa)/2,p+1)+ «est la  solution trouvée après» +(k+1)+ «itérations»;
}:;
```

En reprenant le même exemple :

```
dicho(x^2-2,5,1,2)
```

On obtient la même réponse :

```
1.414211273 est la solution trouvée
après 18 itérations.
```

Les deux versions sont ici assez similaires avec toujours peut-être une plus grande simplicité d'écriture dans le cas récursif.

4. Algorithme récursif ne signifie pas nécessairement $u_{n+1} = f(u_n)$

Si les suites définies par une relation $u_{n+1} = f(u_n)$ sont facilement décrites par un algorithme récursif, ce n'est pas le seul champ d'action de la récursion.

Dès qu'on connaît une situation simple et qu'on est capable de simplifier une situation compliquée, on peut utiliser la récursion. Par exemple, reprenons l'exemple de la section 2.4 qui calcule la longueur d'une liste :

• la liste vide est de longueur nulle (cas simple) ; la longueur d'une liste est égale à 1+ la longueur de la liste sans son premier élément.

Cela donne en traduction directe :

```
# let rec longueur = function
        [] -> 0
  | tete::corps -> 1+longueur(corps);;
```

La barre verticale permet de définir une fonction par morceaux en distinguant les cas. L'écriture x::liste signifie qu'on rajoute *x* en tête de la liste liste.

CAML trouvant lui-même le type des variables utilisées sait qu'on va lui entrer une liste de n'importe quoi (puisqu'on peut l'écrire sous la forme tete rajoutée à corps) et que longueur sera un entier (puisqu'on lui ajoute 1).

4.1 *Ordre supérieur et récursion : la dérivation formelle en 2 temps 3 mouvements...*

Il est facile de calculer à partir de n'importe quel langage de programmation une approximation du nombre dérivé d'une certaine fonction en un certain réel. Par exemple avec CAML :

```
# let deriv_num(f,x,dx) = (f(x+.dx) -. f(x)) /. dx;;
```

Pour obtenir une approximation du nombre dérivé en 0 de la fonction sinus :

```
# deriv_num(sin,0.,0.00000001);;
- : float = 1.
```

Cependant, il ne faut pas oublier qu'un ordinateur ne travaille pas sur des réels mais des flottants en nombre fini ce qui conduit à des petites erreurs d'approximation qui peuvent s'accumuler... jusqu'à l'absurde.

Définissons en effet une fonction qui renvoie la fonction dérivée « approchée » :

```
# let fonc_deriv_num(f) = function
      x -> (f(x +. 0.0000000001) -
                . f(x)) /. 0.0000000001;;
```

Demandons d'évaluer ensuite la dérivée cinquième de sinus en 0 :

```
# fonc_deriv_num(fonc_deriv_num(fonc_deriv_num(fonc_deriv_num(fonc_deriv_num(sin)))))(0.);;

- : float = -1.3322676295501873e+25
```

C'est-à-dire que $\sin^{(5)}(0) < -10^{25}$ ce qui peut laisser perplexe !

Pour remédier à ce problème, il paraît préférable de travailler avec une dérivation formelle (ou symbolique) mais cela peut sembler une tache bien délicate. La dérivation doit être en effet considérée comme une fonction d'ordre supérieur ayant pour valeur une fonction (la fonction dérivée). On ne travaille plus sur des variables numériques.

Cependant, avec un langage permettant de définir récursivement des fonctions d'ordre supérieur, cela va se faire très rapidement.

On crée d'abord un type de variable (en plus des types existant comme int, float, list, etc.) qu'on appelle *formel* par exemple et qui va faire la liste et définir les expressions formelles que l'on veut dériver :

```
# type formel =
   | Int of int
    | Var of string
    | Add of formel * formel
    | Sous of formel*formel
    | Mul of formel * formel
    | Div of formel*formel
    | Ln of formel
    | Cos of formel
    | Sin of formel
    | Puis of int*formel
    | Rac of formel
    | Exp of formel;;
```

Par exemple, Var (comme variable...) sera de type formel mais prendra comme argument une chaîne de caractère (string). On entrera donc les variables à l'aide de guillemets.

De même, Add est de type formel*formel, c'est-à-dire que c'est un objet formel qui prend comme argument un couple d'objets formels.

On définit ensuite récursivement une fonction *deriv* qui va dériver nos expressions selon des règles que nous allons établir :

```
# let rec deriv(f, x) =
  match f with
  | Var y when x=y -> Int 1
  | Int _ | Var _ -> Int 0
  | Add(f, g) -> Add(deriv(f, x), deriv(g, x))
  | Sous(f, g) -> Sous(deriv(f, x), deriv(g, x))
  | Mul(f, g) -> Add(Mul(f, deriv(g, x)),
                     Mul(g, deriv(f, x)))
  | Div(f,g) -> Div(Sous(Mul(deriv(f,x),g),
                Mul(f,deriv(g,x))),Mul(g,g))
  | Ln(f) -> Div(deriv(f,x),f)
  | Cos(f) -> Mul(deriv(f,x),Mul(Sin(f), Int(-1)))
  | Sin(f) -> Mul(deriv(f,x),Cos(f))
  | Puis(n,f) -> Mul(deriv(f,x),Mul(Int(n),Puis(n-1,f)))
  | Rac(f) -> Mul(deriv(f,x),Div(Int(1),
                     Mul(Int(2),Rac(f))))
  | Exp(f) -> Mul(deriv(f,x),Exp(f));;
```

Par exemple, quand la variable Var est en fait la *variable* au sens mathématique, sa dérivée est 1. En effet, Var regroupe à la fois la variable de la fonction mathématique mais aussi les paramètres formels éventuels.

La ligne suivante dit que toute autre variable (donc paramètre) et toute constante numérique entière (Int) a pour dérivée 0. Ensuite on explique comment dériver une somme, une différence, un cosinus, etc.

Par exemple, dérivons $x \to kx^2$.

On définit deux variables k et x :

```
# let x , k = Var «x», Var «k»;;
```

Puis on demande la dérivée :

```
# deriv(Mul(k,Puis(2,x)),»x»);;
```

On obtient :

```
- : formel =
Add (Mul (Var «k», Mul (Int 1, Mul (Int 2,
Puis (1, Var «x»)))), Mul (Puis (2, Var «x»), Int 0))
```

qu'il reste à « déchiffrer »...

$$k \times 1 \times 2 \times x^{-1} \times x^2 \times 0 = 2\,kx$$

Pour $\ln(1/x)$:

```
# deriv(Ln(Div(Int(1),x)),»x»);;
```

On obtient :

```
- : formel =
Div
    (Div (Sous (Mul (Int 0, Var «x»),
    Mul (Int 1, Int 1)), Mul (Var «x»,
    Var «x»)),Div (Int 1, Var «x»))
```

C'est-à-dire : ((0*x – 1*1)/(x*x))/(1/x)

Just for fun : dérivons $\ln(\ln(\ln(x)))$

```
# deriv(Ln(Ln(Ln(x))),»x»);;
```

On obtient :

```
- : formel = Div (Div (Div (Int 1, Var «x»),
          Ln (Var «x»)), Ln (Ln (Var «x»)))
```

C'est-à-dire : $((1/x)/\ln(x))/\ln(\ln(x))$

On devrait être extrêmement impressionné par ce qui vient d'être fait ici ! A partir de la définition d'un type de variable formel et d'une fonction deriv, on a pu calculer de manière symbolique des dérivées de toutes les fonctions étudiées au lycée ! Et on est parti de pratiquement rien sur les fonctions numériques ! On a juste précisé quelle était la règle de dérivation basique pour chacune d'elles et leur somme/produit/rapport. Ces règles basiques viennent de démonstrations mathé-

matiques à partir de la définition du nombre dérivé, de l'exponentielle, du logarithme, des formules trigonométriques. Ensuite, on a pu garder des résultats exacts et non pas des approximations numériques même après avoir fait appel à l'ordinateur.

L'informatique devient ainsi un prolongement de l'exactitude absolue des mathématiques et non plus seulement un outil d'expérimentations faites d'approximations. On peut alors penser au logiciel COQ d'assistance de preuve qui est un système de manipulation de preuves mathématiques formelles écrit en CAML.

La seule petite entorse à notre règle ascétique est la fonction Puis qui utilise la somme des entiers. On aurait pu l'écrire :

```
| Puis(n,f) -> Mul(d(f,x),
    Mul(Int(n),Puis(Sous(Int(n),Int(1)),f)))
```

ou ne pas la définir du tout et se contenter de Mul(x,x) mais bon, on a un peu gagné en lisibilité... Il reste d'ailleurs à effectuer un traitement informatique de ces deux fonctions pour qu'elles affichent leurs résultats de manière plus lisible. Cependant, cela prendrait du temps et l'exercice consistant à décrypter les réponses peut être formateur en classe.

5. Alors : avec ou sans affectation ?

Beaucoup de professeurs (et donc d'élèves...) sont habitués à traiter certains problèmes à l'aide d'un tableur... qui fonctionne récursivement !

En effet, pour étudier par exemple les termes de la suite $u_{n+1} = 3u_n + 2$ de $n = 1$ à $n = 30$ avec $u_1 = 7$, on entre = 7 dans la cellule *A*1 et = 3**A*29 + 2 dans la cellule *A*! bch30 et on « fait glisser » de la cellule *A*30 jusqu'à la cellule *A*2. On a plutôt l'habitude de faire glisser vers le bas mais pourquoi pas vers le haut ?

Passer d'un « faire glisser » à l'écriture de l'algorithme correspondant est une évolution naturelle ce que l'introduction d'un langage impératif n'est pas.

La notion de fonction est naturellement illustrée par l'utilisation d'un langage fonctionnel alors que les procédures des langages impératifs peuvent prêter à confusion.

Comme de nombreux collègues, j'assure des « colles MAPLE » en classes préparatoires scientifiques depuis plusieurs années. Les élèves de ces classes ont habituellement un niveau en mathématique supérieur au niveau moyen rencontré en Seconde et pourtant la plupart des étudiants éprouvent de réelles difficultés à maîtriser un langage de programmation impératif comme MAPLE et certains n'y arrivent pas après une année d'initiation !

Evidemment, nous n'aborderons pas les mêmes problèmes en Seconde mais pouvoir se concentrer sur les mathématiques et simplement traduire nos idées grâce au clavier de manière assez directe nous permettra de préserver nos élèves de nombreux pièges informatiques sans parler des notions trompeuses que l'affectation peut introduire dans leurs esprits.

Cependant, utiliser des algorithmes récursifs nécessite de maîtriser un tant soit peu... la notion de récurrence. Or cette étude a été réservée jusqu'à maintenant aux

seuls élèves de Terminale S. L'étude des suites débute depuis de nombreuses années en Première et chaque professeur a fait l'expérience des difficultés des élèves à maîtriser cette notion.

Ainsi, même si (ou plutôt parce que) les algorithmes récursifs peuvent apparaître proches des notions mathématiques qu'ils permettent d'étudier, ils demandent de comprendre une notion mathématique délicate, la récurrence. L'avenir au lycée de l'étude des phénomènes évolutifs modélisés par une relation $u_{n+1} = f(u_n)$ demeure cependant incertain avec une classe de Première S passant à quatre heures de mathématiques par semaine. Il faut surtout noter que le champ d'action des algorithmes récursifs dépasse largement la notion de suite numérique. On peut raisonner récursivement dès qu'on connaît un état simple et qu'on peut simplifier un état compliqué.

D'un autre côté, les algorithmes utilisant les affectations contournent la difficulté de la récurrence, facilitent l'étude de certains mécanismes comme le calcul matriciel, permettent souvent une illustration « physique » d'un phénomène mathématique et sont en tous cas complémentaires des versions sans affectation.

Ils doivent de plus être étudiés car ils sont largement répandus dans le monde informatique.

Il ne faut enfin pas oublier les difficultés que pourraient rencontrer des professeurs de mathématiques n'ayant pas réfléchi à ces problèmes d'algorithmique et qui ne pourront pas être formé(e)s avant de devoir enseigner une notion méconnue, sans manuels et sans le recul nécessaire.

Mais il ne faudrait pas pour autant abandonner l'idée d'initier les élèves à une « algorithmique raisonnée ». La notion de test par exemple (SI...ALORS...SINON) devrait être abordée par le plus grand nombre car un effort est demandé sur les notions basiques de logique dans le nouveau programme de Seconde. Un travail spécifique sur ces tests en algorithmique peut ainsi enrichir l'enseignement des mathématiques (et dans tout style de programmation). La notion de fonction peut être étudiée sans danger avec un langage fonctionnel et avec prudence avec un langage impératif. D'un point de vue plus informatique, faire découvrir aux élèves que le « dialogue » avec la machine ne se réduit pas à des « clics » mais que chacun peut « parler » presque directement à la machine en lui tapant des instructions est également un apprentissage enrichissant.

Quant au match avec ou sans affectation, il ne peut se terminer par l'élimination d'un des participants : il faut savoir que l'un ou l'autre sera plus adapté à la résolution d'un problème et l'emploi *combiné* des deux types de programmation abordés dans cet article peut surtout permettre d'illustrer sous des angles différents une même notion mathématique et donc illustrer de manière enrichissante nos cours, ce qui induit la nécessité de connaître différentes manières de procéder pour choisir la plus efficace. Cela accroît d'autant plus la difficulté d'une introduction improvisée de l'algorithmique au lycée...

Mais les difficultés rencontrées par les étudiants débutants de l'enseignement supérieur doivent nous inciter à la prudence et à nous demander si un enseignement informatique optionnel en seconde et plus poussé dans les classes scientifiques supérieures ne serait pas plus souhaitable. On pourra lire à

ce sujet l'opinion de Bernard PARISSE, développeur de XCAS.

6. Documentation et prolongements

De nombreux algorithmes pour le lycée sont catalogués à l'adresse suivante :

http://download.tuxfamily.org/tehessinmath/les%20pdf/PafAlgo.pdf

Liens pour CAML :

Une passionnante introduction à OCAML par deux de ses papas...

http://caml.inria.fr/pub/distrib/books/llc.pdf

Le manuel de référence de CAML par Xavier LEROY et Pierre WEIS :

http://caml.inria.fr/pub/distrib/books/manuel-cl.pdf

Une introduction à CAML :

http://fr.wikibooks.org/wiki/Objective_Caml

Ressources OCAML :

http://www.pps.jussieu.fr/Livres/ora/DA-OCAML/

Des installations clé-en-main de CAML pour tous les systèmes d'exploitation :

http://pagesperso-orange.fr/jean.mouric/

Liens pour XCAS :

— la maison mère :

http://www-fourier.ujf-grenoble.fr/%7Eparisse/giac_fr.html

— de nombreux exemples d'utilisation au lycée :

http://tehessin.tuxfamily.org

BIBLIOGRAPHIE :

BARTHE, Daniel : La mesure du cercle. Tangente, avril 2009, Nr. 36 HS, pp 15–16

CONNAN, Guillaume et GROGNET, Stéphane : Guide du calcul avec des logiciels libres. Dunod, 2008

COUSINEAU, Guy et MAUNY, Michel : Approche fonctionnelle de la programmation. Ediscience, 1995

COUTURIER, Alain et JEAN-BAPTISTE, Gérald : Programmation fonctionnelle - Spécifications & applications. Cépaduès-Éditions, 2003

DARTE, Alain et VAUDENAY, Serge : Algorithmique et optimisation. Dunod, 2001

DONZEAU-GOUGE, Véronique et *al.* : Informatique - Programmation - Tomes 1, 2 et 3. Masson, 1986

ENGEL, Arthur et REISZ, Daniel : Mathématique élémentaire d'un point de vue algorithmique. CEDIC,1979

GACÔGNE, Louis : Programmation par l'exemple en Caml. Ellipses, 2004

HABRIAS, Henri : Spécification avec B. Cours IUT de Nantes, département informatique, octobre 2006

LEROY, Xavier et WEIS, Pierre : Manuel de référence du langage Caml. Inter-Editions, 1993

LEROY, Xavier et WEIS, Pierre : Le langage Caml. Dunod, 1999

MONASSE, Denis : Option informatique : cours complet pour la sup MPSI. Vuibert, 1996

MONASSE, Denis : Option informatique : cours complet pour la spé MP et MP*. Vuibert, 1997

QUERCIA, Michel : Algorithmique. Vuibert, 2002

SAUX PICART, Philippe : Cours de calcul formel - Algorithmes fondamentaux. Ellipses, 1999

VEIGNEAU, Sébastien : Approches impérative et fonctionnelle de l'algorithmique. Springer, 1999