Aspects syntaxiques, sémantiques et épistémologiques de l'enseignement de l'algèbre en vue de l'utilisation des logiciels de calcul formel

Atelier animé par :

Michel JANVIER

IREM de Montpellier

Les logiciels de calcul formel, tels que Maple ou Mathematica, deviendront vite indispensables aux mathématiciens, physiciens et ingénieurs, par les possibilités de calcul qu'ils offrent, par la qualité des représentations graphiques qu'ils permettent d'obtenir, par la souplesse de leur programmation, surtout quand leurs langages sont de type fonctionnel, c'est-à-dire très proche du langage naturel en mathématiques. Ils fournissent des milliers de fonctions préprogrammées, qui permettent aux professionnels de travailler dans les domaines qui les intéressent sans avoir le souci de réinventer la roue. De vastes bibliothèques de documents électroniques sont accessibles sur la toile mondiale (World Wide Web). Ils s'imposeront comme outils de travail, de recherche et d'expérimentation pour tous les professionnels des mathématiques.

Il est donc nécessaire d'apprendre à utiliser ces instruments. Une initiation à ces logiciels a été introduite en classes préparatoires aux grandes écoles, et les écoles d'ingénieurs les ont inscrits à leurs programmes. Ils sont apparus également au programme de l'agrégation de mathématiques. Leur apprentissage intervient donc progressivement dans l'enseignement supérieur. Il semble prématuré d'introduire l'usage de ces logiciels par les élèves dans l'enseignement secondaire, tant que les notions fondamentales concernant notamment les fonctions et les relations n'ont pas été solidement implantées, et tant que la pratique du raisonnement logique, aussi bien en logique des propositions qu'en logique des prédicats, n'est pas devenue familière aux élèves. Néanmoins, il convient de préparer le terrain pour que l'apprentissage des logiciels de calcul formel puisse s'accomplir ensuite sans avoir à surmonter des obstacles infranchissables. Il s'agit moins d'introduire de nouvelles notions que d'insister par exemple en algèbre sur certains aspects syntaxiques, sur certaines démarches de pensées, d'approfondir la signification du symbolisme utilisé, pour que le passage à la programmation fonctionnelle se fasse ensuite naturellement.

La pratique de la programmation en informatique montre toute l'importance qu'il faut attacher à la syntaxe. Les vérificateurs syntaxiques rejettent impitoyablement toute expression mal formée dans un programme, et tout praticien de l'informatique sait que l'oubli d'un point-virgule ou une parenthèse que l'on a oubliée de fermer empêche le programme de fonctionner. L'informatique impose donc une grande rigueur syntaxique. Ne faudrait-il pas imposer cette même rigueur aux élèves dans l'apprentissage de l'algèbre ? L'utilisation des tableurs peut à cet égard être bénéfique, car une formule ne donnera le résultat attendu que pour autant qu'elle aura été introduite en respectant toutes les contraintes syntaxiques et toutes les priorités entre les opérations reconnues par le logiciel. Faut-il habituer les élèves à l'analyse syntaxique des expressions qu'ils emploient ?

La signification du symbolisme en mathématiques est liée au contexte où il est utilisé. Le signe égal n'a pas le même sens quand on écrit $5 = 2 + 3$, $2 + x = 0$ ou $5 = 8 \bmod(3)$. Dans le premier cas, il s'agit d'indiquer que 5 est la valeur que l'on obtient lorsque l'on fait la somme des deux entiers 2 et 3. Dans le deuxième cas, on a à faire à une équation, c'est à dire à un prédicat de poids un, la variable x prenant ses valeurs dans un ensemble de nombres qui doit être précisé.

On transforme ce prédicat en une proposition vraie ou fausse en substituant à x une valeur prise dans le domaine de définition de la variable, ou en quantifiant le prédicat. Par exemple

$$(\exists\, x)\ (x \in \mathbf{N})\ 2 + x = 0$$

est une proposition fausse. Enfin dans 5 = 8 mod(3), le signe = signifie que le couple (5,8) vérifie la propriété que 5 – 8 est divisible par 3. Le signe égal marque ici la présence d'une relation d'équivalence.

Si l'expert sait reconnaître la signification qu'il accorde aux symboles, en est-il de même des élèves ? Des études assez anciennes montrent que ce n'est pas le cas pour les débutants et que ceux-ci attribuent aux symboles un sens qui n'est pas toujours celui que leur donnent les experts.

L'introduction de l'informatique ne facilite pas la tâche des enseignants et des élèves dans ce domaine, puisque certains symboles ont des significations différentes en informatique et en mathématique. En informatique le signe égal peut être le symbole d'une assignation, définissant la valeur d'une variable, il peut provoquer l'évaluation immédiate de l'expression inscrite à sa droite et son stockage dans une case mémoire dont le nom est celui du membre de gauche de l'égalité. L'informatique conduit à distinguer par exemple des signes comme =, :=, et ==, avec des différences de sens qui sont nécessaires pour la programmation. Ces différences de sens apparaissent aussi en mathématiques, mais elles n'y sont que rarement explicitées. Comment gérer cette approche différente entre les deux disciplines ?

Les logiciels de calcul formel modernes emploient de moins en moins la programmation impérative, avec ses structures de contrôle comme les boucles For, While, Do, ... Ces structures étaient utiles pour les calculs pas à pas, mais la programmation fonctionnelle s'adapte fort bien aux relations définies par récurrence et à la récursivité. Si une suite récurrente $(u_n)_n$ est définie par une relation du type $u_{n+1} = f(u_n)$, le calcul du n-ème terme u_n s'obtient en appliquant à u_0 la fonction f^{n} obtenue en composant n fois la fonction f, et si la fonction f satisfait à certaines conditions la limite de la suite $(u_n)_n$ est un point fixe de f. Les logiciels de calcul formel, comme Mathematica disposent de moyens pour trouver les points fixes de fonctions. Faut-il habituer nos élèves à s'exercer à la composition des fonctions ? les initier à la notion de point fixe, qui joue de toute manière un si grand rôle en analyse (Théorème de Picard) ?

Un autre aspect de ces logiciels est la programmation par règles : règles de substitution qui remplacent certaines variables par d'autres expressions, règles de transformations des expressions dans le but d'obtenir un certain résultat. Ce mode de programmation oblige à une analyse fine des procédures de résolution des problèmes que l'on se pose. Faut-il mettre l'accent en algèbre sur les procédures de résolutions, par exemple des équations ? L'explicitation des différents procédés de résolution des problèmes posés, une méthodologie précise et clairement affichée pour la prise en compte de certaines situations, habitueraient les élèves à la décomposition des tâches nécessaire à une programmation efficace. Ce n'est là que l'application de l'un des principes du discours de la Méthode de Descartes : décomposer les problèmes en autant de sous-problèmes qu'il sera nécessaire. Faut-il faire entraîner les élèves à cette forme de pensée analytique ?

Les logiciels de calcul formel fonctionnent souvent comme des boîtes noires. Ils résolvent certaines équations algébriques à coefficients entiers de degré élevé, supérieur à 4. Ils savent décomposer en produit de facteurs premiers des nombres entiers assez grands. Ils résolvent des équations différentielles complexes. Mais l'utilisateur ignore en général comment le logiciel s'y prend pour obtenir ces résultats. Par exemple, rien dans l'enseignement de l'algèbre ne prépare, même en licence, aux bases de Groebner, nécessaires à la résolution des systèmes d'équations polynomiales de plusieurs variables à coefficients entiers. Pourtant les bases de Groebner ont été implantées dans les logiciels de calcul formel et expliquent leurs performances. Faut-il penser à

revoir les programmes d'enseignement pour que les utilisateurs de ces logiciels comprennent comment ils fonctionnent et ne soient pas obligés de se contenter de faire confiance à la machine, comme ces élèves qui ne savent plus faire une division à la main et qui doivent se contenter du résultat affiché par leur calculettes, sans contrôle possible ?

1. Aspects syntaxiques

Le langage mathématique est complexe car il utilise deux codes imbriqués, la langue naturelle et la langue symbolique. Il est évidemment illusoire de vouloir tout écrire en langue symbolique, le résultat serait totalement incompréhensible : les abus de langage sont donc inévitables. Tout texte mathématique est un compromis entre des explications en langue naturelle et des relations écrites dans un symbolisme, dont les règles sont souvent implicites et apprises par l'usage. En ce qui concerne l'enseignement français, il n'y a pas d'apprentissage spécifique du code symbolique.

En informatique les langages sont exclusivement des langages symboliques, les seuls compréhensibles par les machines. Même si les programmeurs ne sont pas astreints à écrire en langage machine, les langages de haut niveau obéissent à des syntaxes précises, ne serait-ce que pour que les compilateurs puissent les traduire en langage machine, le seul compréhensible par les processeurs des ordinateurs.

1.1. Les langages formels

Les langages formels que l'on rencontre en programmation, ou en logique sont formés d'expression dont la construction obéit à des règles précises. On part en général d'expressions atomiques données à priori. Les expressions bien formées du langage sont obtenues en combinant les expressions atomiques, des opérateurs, des parenthèses selon des règles, qui sont données souvent sous une forme inductive. On rencontre aussi de tels systèmes formels en logique.

En logique des propositions, les signes primitifs sont

- Les lettres « p, q, r,... » dites propositions élémentaires
- Les signes $\vee$ et $\neg$ (qui correspondent aux connecteurs ou et non)
- Les parenthèses (et).

Les propositions ou expressions bien formées, e.b.f., sont définies ainsi :

- Les propositions élémentaires sont des propositions
- Si P est une proposition $\neg$P aussi.
- Si P et Q sont des propositions, P $\vee$ Q aussi.

1.2. Les définitions et démonstrations inductives

On peut remarquer la forme inductive de cette définition des propositions en logique, qui indique comment obtenir de nouvelles propositions à partir de propositions élémentaires. De tels schémas d'axiomes se rencontrent souvent dans les langages informatiques. Ne doit-on pas habituer les élèves, dès le secondaire à ce mode de construction, que l'on rencontre aussi en mathématiques, à partir d'une base et de propriétés d'hérédité. On se contente au mieux de

quelques définitions par récurrence, dans l'enseignement secondaire. Par exemple, l'introduction des entiers naturels par l'axiomatique de Peano, pourrait être envisagée, sinon développée entièrement, le but étant de montrer comment tout un corpus théorique peut être développé à partir d'un petit nombre d'axiomes, et comment on peut engendrer une infinité d'éléments à partir de quelques objets de base.

On peut juger cette introduction prématurée pour les lycéens, mais comment parviendra-t-on en logique des propositions à la décomposition canonique disjonctive ou conjonctive des propositions composées, sinon par un raisonnement de type inductif ? Or l'analyse des propositions composées, et l'étude de leurs simplifications, est une nécessité pratique pour les programmeurs. Elle est enseignée en I.U.T. et en S.T.S. L'impréparation à ce mode de raisonnement peut conduire à des blocages.

1.3. Les langages

Les langages en informatique sont considérés comme des parties d'un monoïde libre sur un alphabet fini. On part d'un nombre fini de lettres, constituant un alphabet A et tout langage est formé de chaînes de caractères construites en utilisant les lettres de l'alphabet. L'ensemble des chaînes que l'on pourrait construire à partir de l'alphabet A, y compris la chaîne vide, muni de l'opération de concaténation constitue un monoïde, noté A^* et appelé le monoïde libre construit sur l'alphabet A. Les langages sur l'alphabet A ne sont que des parties de A^*. Il faut donc préciser les règles de productions des chaînes appartenant au langage.

Ainsi dans des langages comme Fortran, Pascal, C ou même Java le type entier peut être défini de la manière suivante. On part de l'alphabet A = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -} et des expressions composées N = {<chiffre>, <entier>, <entier avec signe>, <entier sans signe>} et des règles de production suivantes :

<chiffre> → 1|2|3|4|5|6|7|8|9|0 (la barre | signifie ou)

<entier> → <entier avec signe>

<entier> → <entier sans signe>

<entier avec signe> → + <entier sans signe>

<entier avec signe> → - <entier sans signe>

<entier sans signe> → <chiffre>

<entier sans signe> → <chiffre> <entier sans signe>

Le symbole initial étant <entier>.

Ainsi –363 s'obtient à partir du symbole initial <entier> par

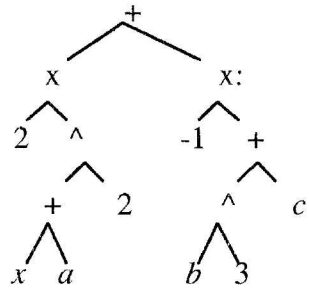
<entier> → <entier avec signe> → - <entier sans signe> → - <chiffre> <entier sans signe> →- <chiffre> <chiffre> <entier sans signe> →- <chiffre> <chiffre> <chiffre> → - 3 <chiffre< <chiffre> → -36 <chiffre> → -369.

L'ensemble constitué des signes primitifs - l'alphabet - des signes terminaux, du symbole initial et des règles de production constitue ce que l'on appelle une grammaire G. Les expressions engendrées par les règles de production formeront un langage $L(G)$. Pour modéliser les ordinateurs on utilise en informatique des modèles abstraits, comme les automates, ou les machines de Turing. Les automates reçoivent des chaînes de caractères, en acceptent certaines et en refusent d'autres. Les chaînes qui sont acceptées constituent le langage de l'automate. Kleene a pu démontrer que les langages acceptés par les automates sont les mêmes que ceux qui sont

engendrés par certaines grammaires, dites régulières. Il n'est pas question d'aborder la théorie des langages, les grammaires génératives et les automates avant l'enseignement supérieur destiné aux futurs informaticiens. Mais les expressions algébriques et les relations algébriques constituent un langage et il serait sans doute bon d'insister sur les règles de production de ces expressions et relations, et montrer qu'une expression mal formée ou ambiguë n'est pas acceptable. L'usage des tableurs, des calculatrices, qu'elles soient ou non muni d'un moteur de calcul formel, devrait permettre de sensibiliser les élèves à ces questions, puisque ces dispositifs n'accepteront que des expressions bien formées.

1.4. La structure en arbre des expressions algébriques

Sans aborder la théorie des langages dans l'enseignement secondaire, il serait utile de procéder à l'analyse de la structure en arbre des expressions algébriques. On sensibiliserait ainsi les élèves aux règles de priorité des opérateurs, ce qui les préparerait à l'indispensable analyse de l'évaluation des expressions dans un langage informatique. Ainsi l'expression $2(x + a)^2 - (b^3 + c)$ admet la structure en arbre ci-dessous



Pour évaluer cette expression on part des feuilles 2, x, a, -1, b, 3, c, puis on remonte progressivement vers la racine, en évaluant par exemple $x + a$, puis $(x + a)^2$ puis $2 \times (x + a)^2$. Par ailleurs on évalue b^3, puis $b^3 + c$ et $(-1) \times (b^3 + c)$. On évalue enfin la somme des deux sous-expressions $(x + a)^2$ et $(-1) \times (b^3 + c)$. Chacune des sous-expressions évaluées a elle-même une structure d'arbre. C'est bien ainsi qu'on calculerait la valeur numérique de cette expression si l'on attribuait à x, a, b et c des valeurs numériques. Cet apprentissage progressif du mécanisme d'évaluation d'une expression faciliterait la compréhension ultérieure du fonctionnement des langages informatiques et éviterait bien des surprises quand un ordinateur envoie un résultat insolite.

1.5. La structure fonctionnelle des expressions

En repartant de la racine - ne pas oublier qu'en informatique les arbres ont la tête en bas, les racines en haut et les feuilles en bas - on peut découvrir la structure fonctionnelle des expressions algébriques. En notant Plus, l'addition, Times la multiplication et Power l'exponentiation, l'expression précédente se réécrit

Plus(Times(2, Power(Plus(x, a), 2), Times(-1, Plus(Power(b, 3), c)))

On met ainsi en lumière les lois de composition binaires qui interviennent dans l'architecture de l'expression et la manière dont on compose ces lois. Toute expression algébrique s'obtient comme produit de composition de lois de composition, ce qui montre bien la structure fonctionnelle du symbolisme mathématique. Mais même les relations ou les équations s'adaptent à cette structure. En effet une équation est un prédicat de poids n si elle comporte n inconnues, c'est à dire finalement une fonction booléenne qui prend la valeur « vraie », ou 1, si on remplace les inconnues par des solutions de l'équation et la valeur « faux », ou 0, si on

remplace au moins l'une des inconnues par une valeur qui ne figure pas parmi les solutions de l'équation. Toute assertion mathématique peut s'exprimer sous la forme d'un prédicat de poids fini, donc comme une sorte de fonction booléenne à n variables, c'est ce qui explique pourquoi les langages fonctionnels sont les mieux adaptés aux mathématiques.

Ils se sont développés à partir du λ-calcul. Ils ont également bénéficié de l'expérience de langage tels que LISP qui a permis de manipuler les listes et les enchevêtrements de parenthèses. Le langage de calcul formel le plus adapté à la programmation fonctionnelle, mais qui autorise aussi la programmation par règles, est Mathematica. Malheureusement, par suite d'une politique commerciale désastreuse il a été supplanté dans les classes préparatoires par Mapple dont les revendeurs ont eu l'intelligence d'organiser des stages de formation à l'intention des professeurs. De plus Mapple comporte encore une bonne partie de langage impératif, et il s'apparente ainsi encore au Pascal, qu'utilisaient auparavant les professeurs des classes préparatoires, il s'est imposé car il dépaysait moins que Mathematica. Pourtant Mathematica est beaucoup plus cohérent, sa programmation fonctionnelle est bien adaptée aux mathématiques, et en tant que traitement de texte, il offre la possibilité de présenter les calculs de manière fort élégante.

Pour bien illustrer la structure fonctionnelle de Mathematica, voyons comment sont définies les expressions dans ce langage. Il nous faut d'abord définir les atomes.

- Les atomes sont soit des symboles, soit certains nombres, soit des chaînes de caractères.

Un symbole est toute suite de lettres, y compris le signe $, majuscules ou minuscules, et d'entiers qui ne commencent pas par un entier (Ainsi h5Ert56 est un symbole, 5kLO n'en est pas un).

Les nombres qui sont des atomes sont soit des entiers soit des nombres décimaux. (2365 et −56.25 sont des atomes. Le rationnel _ et l'entier de Gauss 3 −4 i ne sont pas des atomes)

Les chaînes de caractères sont formées de toute suite de caractères ASCII placés entre guillemets (« Vive la République ! » est un atome).

- Les expressions sont alors définies de la manière suivante :

(base) Tout atome est une expression

(hérédité) Toute expression est de la forme

$$f[\exp_1, \exp_2, \ldots, \exp_n]$$

où n est un entier naturel, et où f, $\exp_1$, $\exp_2$, ..., $\exp_n$ sont des expressions. f s'appelle l'en-tête de l'expression.

On constate ainsi que toute expression prend bien une forme fonctionnelle, et comme l'en-tête d'une expression est elle-même une expression, on voit que l'on pourra aisément définir des fonctions qui dépendent de paramètres. De plus, les arguments des fonctions peuvent également être des fonctions, des fonctions pourront retourner comme valeur des fonctions. Dans ce langage tout est paramétrisable, c'est qui en fait sa force, et c'est ce qui l'apparente le plus au mode de raisonnement mathématique. Le Pascal servait à construire d'imposantes pyramides, des structures statiques construites par des myriades d'ouvriers mettant en place d'énormes blocs de pierre. Lisp permettait de construire des organismes, des structures dynamiques formées par des brigades ajustant des myriades d'organismes plus simples. Dans Mathematica on pourra élaborer des programmes qui s'écriront sur une seule ligne, en combinant plusieurs fonctions.

Par exemple

RepresentationRacines[poly_, z_] := ListPlot[{Re[z],Im[z]}/.Solve[N[poly==0],z], Prolog -> PointSize[0.04]]

fournit le représentation graphique dans le plan des racines d'un polynôme à coefficients complexes.

racineNewton[f_,x0_] := FixedPoint[(#-f[#]/f'[#])&,x0]

donne la valeur approchée de la racine de l'équation $f(x) = 0$ par la méthode de Newton, en partant de l'estimation x_0 de cette racine.

On se rend compte sur ces exemples combien la programmation est ramassée. Imaginez un programme en Pascal avec les mêmes fonctionnalités ! On voit que la fonction racineNewton prend pour argument aussi bien la fonction f que la valeur numérique x_0. La fonction FixedPoint recherche le point fixe de la fonction $x \rightarrow x - f(x)/f'(x)$, en partant du point x_0.

Pour pouvoir tirer partie de ces langages il faudra donc dominer touts les concepts liés à la notion de fonction. Comme les mathématiques font grand usage de cette notion il serait donc utile d'habituer les élèves à effectuer les opérations les plus fréquentes que l'on rencontre en manipulant des fonctions, comme la composition, la recherche de points fixes, la currification (du nom du mathématicien Curry) c'est à dire l'opération qui consiste à fixer une ou plusieurs variables dans une fonction de plusieurs variables.

2. Aspects sémantiques

Prenons l'exemple du signe égal : «= ». La relation d'identité $a = a$ n'a évidemment aucun intérêt pratique. Savoir que l'objet a est identique à l'objet a ne fait pas avancer la réflexion. En fait en mathématiques le signe égal signale toujours la présence d'une relation d'équivalence, la relation

$$(a+b)^2 = a^2 + 2ab + b^2$$

écrite dans un anneau commutatif signifie que les deux expressions $(a+b)^2$ et $a^2 + 2ab + b^2$ sont équivalentes, c'est à dire qu'en partant des deux éléments a et b on obtient le même élément en calculant d'abord la somme de a et b, puis en élevant le résultat au carré ou en calculant successivement le carré de a, le double produit $2ab$, le carré de b et en faisant enfin la somme des résultats obtenus. On pourrait fournir des exemples analogues à l'infini.

2.1. Les conceptions des jeunes élèves sur le signe égal

Les jeunes enfants savent comparer la cardinalité de deux ensembles, en dénombrant séparément les éléments de chacun d'eux. Vergnaud [1982] a démontré que dès le début de l'école primaire ils possédaient le «théorème en acte » :

$$Card(A \cup B) = Card(A) + Card(B)$$

si A et B sont des ensembles disjoints. Dans ce cas, il suffit en effet de rassembler les objets formant la collection A et les objets formant la collection B pour obtenir la collection, on commence par dénombrer les éléments de A et on poursuit le dénombrement quand on a rassemblé les deux collections pour obtenir le nombre d'éléments de. Mais alors la relation

$$3 + 5 = 8$$

est comprise, à l'école primaire comme «3 et 5 font 8 », En conséquence la relation

$$8 = 3 + 5$$

n'a aucun sens. A ce stade l'égalité n'est pas symétrique. Le signe «= » étant compris comme l'indication d'une opération à effectuer une opération, une relation comme 3 = 3 n'a aucun sens. A ce niveau l'égalité n'est pas réflexive. Comment dans ces conditions parvenir à la compréhension de relations du type

$$3 + 2 = 4 + 1 \ ?$$

Denmark [1976] a montré qu'elle pouvait s'acquérir dès l'âge de 6 ans, à partir d'activités sur les balances. Mais le signe égal n'est pas pour autant considéré à cet âge comme un signe relationnel. Avant l'âge de 10 ans deux expressions reliées par un symbole relationnel tel que le signe «= » doivent d'abord être évaluées et remplacées par le résultat du calcul avant d'être comparées. Ainsi on effectuera les calculs 3 + 2 =5 et 4 + 1 = 5.

2.2. Installation du signe égal comme symbole d'une relation

La conception du signe «= » comme l'indication d'un calcul à effectuer subsiste longtemps. Vergnaud [1979] pose à des élèves ayant treize ans le problème suivant : « Dans une forêt existante 217 arbres ont été plantés. Quelques années plus tard, 425 arbres, parmi les plus vieux, ont été abattus. La forêt compte actuellement 1063 arbres. Combien en avait-elle au départ ? » Il obtient la production suivante :

$$1063 + 217 = 1280 - 425 = 755$$

Le résultat final, 755, est juste mais la première égalité écrite est fausse. 1280 est le résultat de l'addition de 1063 et 217, l'élève s'autorise à enchaîner sur le calcul de la soustraction de 425 à 1280, le deuxième signe égal signale le résultat de ce calcul.

Kieran [1980] obtient avec des sujets ayant entre 12 et 14 ans l'installation du signe égal comme symbole d'une relation et comme incitation à effectuer une opération en faisant manipuler des identités arithmétiques

- avec une opération : 2 x 6 = 4 x 3
- avec deux opérations : 2 x 6 = 10 + 2
- avec plusieurs opérations : 7 x 2 + 3 – 2 = 5 x 2 – 1 + 6

Son objectif était d'introduire la notion d'inconnue en remplaçant certains nombres dans ces identités arithmétiques :

$$7 \leftrightarrow y + 3 - 2 = 5 \leftrightarrow y - 1 + 6$$

Mais la résolution des équations algébriques suppose non seulement de concevoir le signe égal comme la marque d'une relation d'équivalence, mais suppose également la maîtrise de la notion d'équations équivalentes, c'est à dire d'équations ayant les mêmes solutions. Ce concept supplémentaire se met en place lentement puisque l'on constate encore au lycée des traces des conceptions antérieures. Par exemple Byers et Herscovics trouvent pour la résolution de l'équation $x + 3 = 7$, la production suivante

$$\begin{aligned} x + 3 &= 7 \\ &= 7 - 3 \\ &= 4 \end{aligned}$$

ce qui montre que le sujet n'utilise pas encore de manière cohérente le signe égal comme le symbole d'une relation d'équivalence.
Clément signale le calcul suivant de la dérivée de la fonction f par un étudiant en premier cycle d'université :

$$\begin{aligned} f(x) &= \sqrt{x^2+1} \\ &= (x^2+1)^{1/2} \\ &= \frac{1}{2}(x^2+1)^{-1/2} D_x(x^2+1) \\ &= \frac{1}{2}(x^2+1)^{-1/2}(2x) \\ &= x(x^2+1)^{-1/2} \\ &= \frac{x}{\sqrt{x^2+1}} \end{aligned}$$

où le signe égal est encore conçu comme une incitation à transformer l'expression déjà obtenue, et non comme le signe que la nouvelle expression est équivalente à la précédente. C'est encore un signe d'action et non le signe d'une relation.

2.3. Le signe égal comme signe d'affectation

En informatique, dans les langages impératifs, comme le Pascal, ou même dans des langages orientés objets comme Java, le signe «= » ou le signe « := » est utilisé comme symbole d'affectation.

$$a := 5$$

attribue à la variable a la valeur 5. Cette utilisation du symbole vient de l'idée qu'il faut attribuer une case mémoire à la variable a et que dans cette case on va déposer la valeur 5. Avec cette manière de procéder le signe «= » ou le signe «:= » ne peut pas être symétrique puisque les deux expressions situées de par et d'autre du symbole n'ont pas le même statut. On rencontrerait ici un obstacle si on voulait introduire prématurément la programmation de type impératif avant que le signe égal ait bien été installé en mathématique comme le symbole d'une relation d'équivalence.

2.4. Le signe égal comme indication d'une règle de substitution

Dans un langage de type fonctionnel, comme Mathematica, l'expression $a = b$ est une abréviation de la fonction Set[a, b], et correspond à la règle de substitution de a par b. Là encore cette relation est asymétrique, et on pourrait utiliser une flèche au lieu du signe «= » :

$$a \blacklozenge b$$

On verra plus loin que l'utilisation de la programmation par règle permet de simuler les procédures de résolutions des problèmes. Elle est donc fort utile et très adaptée aux mathématiques. Mais, ici encore, le signe égal dans ce langage n'a pas le même sens qu'en mathématiques. Le logiciel utilisera aussi le signe «:= » , comme celui d'une règle retardée, qui sera appliquée ultérieurement. Si on écrit

$$y := x^2 + 1$$

cela signifie que dans les futures occurrences de la lettre y il faudra la remplacer par. Ce procédé pourra servir à définir des fonctions. Dans une équation comme

$$2x = 5$$

l'égalité n'est vérifiée que pour certaines valeurs de x. Cette équation est en réalité un prédicat de poids un, qui peut être interprété comme une fonction booléenne prenant les valeurs «vrai » ou «faux ». En Mathematica elle s'écrira

$$2\,x == 3$$

qui est une abréviation de EqualQ[2 x, 3]. Dans les trois exemples précédents Mathematica emploi des symboles différents, alors qu'en mathématiques on aurait écrit simplement

$$a = b\ ,$$

on aurait parler de la fonction

$y = x^2 + 1$

et de l'équation

$2x = 3$.

La précision est indispensable en informatique pour que les machines puissent produire les résultats que l'on attend d'elles. Les mathématiques sont écrites et lues par des experts qui sont en mesure de décrypter la signification des symboles utilisés, même si leur interprétation n'est pas toujours univoque.

2.5. La signification des variables en informatique et en algèbre

Il y a un autre domaine où l'informatique, du moins dans les langages impératifs, donne à la notion de variable, un sens différent de celui qu'il prend en mathématiques. En mathématiques la variable représente un élément inconnu ou non spécifié dans un ensemble. Dans un langage impératif, une variable peut être soit une donnée explicite du problème, soit un compteur, qui dans une structure de contrôle évite les boucles infinies, soit une variable d'accumulation, qui est actualisée en fonction de l'avancement des calculs, soit une variable de transfert, c'est à dire une variable c qui permet de mémoriser le contenu d'une variable au moment de l'échange des valeurs prises par deux variables a et b.

Renan Samurçay [1985] a étudié les difficultés cognitives rencontrées par des élèves de lycée, débutants dans l'apprentissage d'un langage impératif. Les variables d'accumulation constituent des obstacles notamment en ce qui concernent leur initialisation. Les élèves semblent faire un meilleur usage des compteurs car ces variables sont institutionnalisées dans l'enseignement lors de l'apprentissage de la structure de boucle, mais ici encore elles sont systématiquement initialisée à 0 par les débutants, et elle s'incrémentent automatiquement de 1 : des difficultés apparaissent dès que l'on s'éloigne des modèles canoniques.

Dans les langages fonctionnels, comme Mathematica, la notion de variable se rapproche davantage de la notion de variable en mathématique. En fait ce langage utilise la notion de modèle. Ainsi $x\_$ représente n'importe quel objet, x_Integer représente une variable prenant ses valeurs dans l'ensemble des entiers naturels, x_Rational représente une variable prenant ses valeurs dans l'ensemble des nombres rationnels. On peut construire autant de modèle que l'on désire permettant ainsi de travailler dans des ensembles non triviaux, comme des espaces fonctionnels.

2.6. La programmation par règles et les procédures de résolution des problèmes en algèbre

L'avantage de la programmation par règles est qu'elle imite le raisonnement mathématique, en particulier en algèbre, lorsqu'il utilise une succession de procédures pour résoudre un problème comme la résolution d'une équation. On trouvera ci-dessous la reproduction d'un cahier électronique construit en Mathematica qui définit des règles, qui reproduisent les procédures de résolution d'une équation du premier degré : passage du membre de droite dans le membre de gauche avec changement de signe, écriture du premier membre comme un polynôme du premier degré en l'inconnue x et résolution de l'équation lorsqu'elle prend l'une des formes $x + b = 0$, $a\,x = b$, $b + a\,x = 0$.

Résolution de l'équation du premier degré

Considérons une équation du premier degré d'inconnue x

In[1]:= eq =(x-1)/4 -((x-5)/4 - (12 - 2x)/5)/8 == (x-9)/ 2 - 7/ 8

$$\text{Out[1]}= \frac{\frac{12\ -2\,x}{5}+\frac{5-x}{4}}{8}+\frac{-1+x}{4}== -\left(\frac{7}{8}\right)+\frac{-9+x}{2}$$

Introduisons les règles de résolution des équations du premier degré

Règle 1 : passage du membre de droite dans le membre de gauche avec changement de signe

In[2]:= règle1 = mbg_ = mbd_ -> mbg - mbd ;

Règle 2 : Ecriture du premier membre comme un polynôme du premier degré en x

In[3]:= règle2 = mb_ = 0 :-> Collect[Expand[mb],x]== 0 ;

Règle 3 : Résolution de l'équation quand elle prend une des formes $x+ b=0$, $a\,x = 0$ (il n'est pas nécessaire de préciser que a est non nul), $b + a\,x = 0$.

In[4]:= règle3 = {x + b_ == 0 -> x ==-b,

a_ x == 0 -> x == 0, b + a x == 0 -> x == -b/a}

Appliquons ces règles à notre équation.

In[5]:= eq/.règle1/.règle2/.règle3

$$\text{Out[5]}= \text{x} == \frac{893}{53}$$

Vérifions que l'on peut aussi résoudre l'équation $0\,x = 0$.

In[6]:= eq = 0 x == 0

Eq /. règle1 /. règle2 /. règle3

Out[6]= True

Out[7]= True

Vérifions que l'on peut aussi résoudre l'équation 7- 0 x = 0.

In[7]:= eq = 7 - 0 x == 0

Eq /. règle1 /. règle2 /. règle3

Out[8]= False

Out[9]= False

Créons une fonction qui résoudra les équations du premier degré quel que soit le nom de l'inconnue.

```
In[10] :=
 résoudre[eq_,x_] := eq /. mbg_ = = mbd_ -> mbg - mbd = = 0 /.
       mb_ = = 0 :-> Collect[Expand[mb] , x] = = 0 /.
{x + b_ = = 0 -> x = = -b, a_ x = = 0 -> x = = 0, b_ + a_ x = = 0 -> x = = - b/a }
```

Appliquons notre fonction à une équation d'inconnue *t*.

```
In[11] :=
 Eq = (2 t + a) / b - (t - b) / a = = 3 a t + (a - b)^2 / a / b ;
In[12] := résoudre[eq,t]
```

$$\text{Out[12]= } t == \frac{-2}{-\left(\frac{1}{a}\right) - 3a + \frac{2}{b}}$$

Le programmeur a ainsi appris au logiciel à résoudre les équations du premier degré, et vérifié que le logiciel avait bien compris sa leçon, et qu'il savait même répondre correctement dans le cas où l'équation n'admet pas de solutions et dans le cas où elle en admet une infinité ; dans ces deux derniers cas la réponse est «false » ou «true », ce qui signifie que le prédicat, que constitue l'équation est toujours faux ou toujours vrai. Il est bien clair que les règles utilisées correspondent à des procédures de résolution assez grossières et que l'on pourrait les affiner, Ce sont des règles que pourrait utiliser un élève expérimenté, pour un débutant il faudrait sans doute davantage détailler les opérations.

Cette possibilité de Mathematica d'imiter les procédures de résolution des problèmes d'algèbre peut être employée pour construire des didacticiels d'entraînement à la mise en œuvre des procédures de résolution. A l'aide de ces programmes les élèves pourraient s'exercer à la résolution par exemple des équations, ou en analyse à la dérivation ou l'intégration des fonctions, par expérimentations successives des différentes procédures qui leur ont été enseignées, et apprendre ainsi à les utiliser de manière pertinente et dans l'ordre adéquat. On trouve déjà sur le site de Wolfram Research des didacticiels de ce type.

3. Les connaissances mathématiques nécessaires à une bonne utilisation du calcul formel

Nous avons vu combien les logiciels de calcul formel exigeaient une application rigoureuse de la syntaxe et émis l'hypothèse qu'un apprentissage de l'analyse syntaxique en algèbre pourrait contribuer à faciliter la prise en main du calcul symbolique sur ordinateur. Nous avons souligné l'importance qu'il y avait à donner du sens au symbolisme mathématique, qui est malheureusement souvent polysémique, et nous avons vu, par ailleurs, que cette polysémie ne peut pas être acceptée en programmation. Nous pensons donc que l'enseignement de l'algèbre devrait mettre aussi l'accent sur l'analyse sémantique du symbolisme. Ces deux remarques montrent tout l'intérêt de l'enseignement de l'algèbre dans l'enseignement secondaire, en n'en faisant pas une simple application mécanique de règles, mais en insistant au contraire sur l'analyse syntaxique et l'analyse sémantique.

La dernière question qui se pose est celle des connaissances mathématiques nécessaires à une bonne utilisation des logiciels de calcul formel. En effet ces logiciels sont dotés de possibilités étonnantes comme la résolution de systèmes d'équations polynomiales à coefficients entiers de degré supérieur à 5. Nous savons depuis Galois qu'il n'y a pas de formules utilisant des radicaux pour parvenir à ce résultat, En fait, dans ce cas les logiciels de calcul formel utilisent les bases de Gröbner, qui sont des systèmes de générateurs d'idéaux dans des anneaux de polynômes à plusieurs indéterminées sur l'anneau **Z** des entiers relatifs. Pour éviter cet effet de boite noire, il faudrait peut être introduire dans l'enseignement de l'algèbre, en licence ou en maîtrise, quelques notions d'algèbre utilisées par ces logiciels. On connaît bien entendu l'inertie des programmes, et leur très lente évolution. Elle ne devrait pas être de mise dans l'enseignement supérieur. L'absence d'enseignement interdisciplinaire explique aussi que souvent l'enseignement des mathématiques reste très théorique et coupé des applications que l'on fait de cette discipline dans d'autres domaines comme l'informatique, la physique ou même l'économie. Ainsi l'enseignement de l'algèbre linéaire ignore superbement l'application du calcul matriciel en électronique, ou les matrices de Leontieff en économie, l'interprétation des valeurs propres dans les systèmes dynamiques, etc. Pourtant la connaissance de ces utilisations des théories abstraites pourrait soutenir la motivation des étudiants. Galois serait sans doute surpris d'apprendre que les corps finis et la théorie des espaces vectoriels sur les corps finis est à la base des codes détecteurs et correcteurs d'erreurs, indispensables au bon fonctionnement de nos modernes modems, sans lesquels Internet ne fonctionnerait pas. Sans cette théorie aurions-nous des images satellitaires fiables et à quoi aurait servi le très coûteux lancement du télescope Hubble ?

Les algébristes ne savent sans doute pas mettre en évidence tous les apports de leur discipline, qui a joué un rôle fondamental dans le développement de l'informatique théorique, notamment en théorie des langages, et a aidé à la mise au point des logiciels de calcul symbolique.

4. Conclusion

Il semble donc qu'une adaptation de l'enseignement de l'algèbre aux nouvelles technologies et aux nouveaux moyens de calcul serait souhaitable.

Par ailleurs, une bonne compréhension du fonctionnement et de la programmation de ces logiciels permet de mieux analyser les concepts fondamentaux de l'algèbre, et invite à mettre l'accent dès le début de l'apprentissage de cette discipline sur l'analyse syntaxique et sémantique.

Les nouveaux moyens de calcul, loin de nous rendre incapables d'effectuer les opérations que l'on faisait à la main, comme certains le craignent, obligent au contraire à l 'analyse des algorithmes de calcul et nous invitent à acquérir de nouvelles connaissances.

Bibliographie

BOSCHET F., 1987, "*Fonctions du code symbolique dans le discours mathematique*", Educational Studies in Mathematics 18 [1987] 19-34.

BYERS V., HERSCOVICS N., 1977, "*Understanding school mathematics*", Mathematics Teaching., 81, 24-27.

CLEMENT J., 1980, *"Algebra word problem solutions ; Analysis of a common misconception"*, commincation au colloque annuel de l'American Educational Research Association, Boston.

DENMARK T., BARCO E., VORAN J., 1976, "Final report : *"A teaching experiment on equality"*, PMDC Technical Report No 6, Florida State University (ERIC Document Reproduction Service N0 ED144805).

KIERAN C., 1981, "*Concepts associated with the equality symbol"*, Educational Studies in Mathematics, 12 [1981] 317-326.

SAMURÇAY RENAN, 1985, *"Signification et fonctionnement du concept de variable informatique chez des élèves débutants"*, Educational Studies in Mathematics, 16 [1985], 143-161,

VERGNAUD G., 1982, *"A classification of cognitive tasks and operations of thought involved in addition and substraction problems"*, in T.P. Carpenter, J. M. Moser and T.A. Romberg (eds), Addition and Substraction : a Cognitive Perspective, Hillsdale, Lawrence Erlbaum.