Sophus, un langage spécial pour les programmes de calcul

Alain Busser
129 rue Marius et Ary Leblond
97430 Le Tampon
abusser@ac-reunion.fr
IREM de La Réunion[1]

Sophus est un langage de programmation qui facilite la rédaction d'algorithmes de par les caractéristiques suivantes:

- Il est relativement proche de la langue française (« tant que » au lieu de « while ») ;
- Il sollicite peu la mémoire de travail, les opérations étant menées *in situ* ;
- Il permet de mener des opérations répétitives sans nécessiter l'utilisation d'un indice ;
- Il est muni d'un interpréteur en ligne (ici: http://irem.univ-reunion.fr/spip.php?rubrique173 par exemple) ce qui permet de programmer en Sophus sur smartphone etc,
- Son interpréteur est muni d'une fonction de traduction automatique en français qui rapproche encore le langage de programmation, d'un langage de description d'algorithmes.

La création de ce langage est le fruit de constatations faites en classe (et particulièrement en terminale STI2D) et au bac (STI2D) sur les difficultés linguistiques endurées par les élèves ayant des difficultés en algorithmique, en particulier la difficulté à imaginer à quoi ressemble une variable vue de l'intérieur (et surtout du fait qu'elle varie), la confusion entre le nom de la variable et sa valeur[2], un certain manque de projection dans l'avenir (après avoir multiplié la population par 0,95 que dois-je faire du résultat ?) et tout simplement des difficultés de vocabulaire, en particulier en anglais.

Le développement de Sophus s'est donc fait au cours d'une réflexion sur les mots implicites d'un algorithme (ou du cours de mathématiques), sur le vocabulaire le plus simple qui puisse

[1] La création de Sophus fait partie des productions de l'atelier développement de webApps avec CoffeeScript, animé par Alain Busser (Lycée Roland-Garros) et Florian Tobé (collège de Vincendo).
[2] Une variable a en général au moins trois attributs différents, qui sont son nom, son type et sa valeur. Un raccourci de langage souvent utilisé (et souvent source de confusion) est d'utiliser le nom pour désigner la valeur (penser au "x est une variable de type nombre" où x désigne le nom de la variable, alors que lorsqu'on calcule x+2 la même lettre x désigne implicitement la valeur de la variable (on n'additionne pas 2 à un nom en général). Pour éviter cette confusion, certains langages comme bash font grand usage d'une fonction qui, à un nom de variable, associe la valeur de la variable. Cette fonction est si souvent utilisée en bash (ou Perl ou php…) qu'elle est notée par un simple symbole "dollar". Alors le contenu de "la variable x" est noté $x. On voit souvent ce symbole dans la barre d'adresse du navigateur Internet lorsque celle-ci contient du code php. En Sophus par contre c'est le mot "valeur" qu'on utilise.

décrire un problème (ou l'algorithme qui le résout), mais aussi sur le besoin de précision qui permet de savoir, quand on parle d'un algorithme, de laisser le moins de doutes possibles sur ce dont on parle. Par exemple, un idiome central dans Sophus est celui consistant à écrire un verbe à l'infinitif quand on le pense à l'impératif, ce qui permet de ne pas tutoyer la machine.

Un mot sur les opérations dites "in situ": Depuis le langage de programmation Fortran (axé sur l'algèbre comme en témoigne son nom "formula translator"), il est d'usage de représenter l'affectation de variables par le signe d'égalité: En écrivant que M=(A+B)/2 pour calculer une moyenne, on demande à Fortran de calculer une demi-somme dans le processeur puis de la stocker dans la variable nommé M: Il y a alors trois opérations élémentaires: Chercher en mémoire les contenus de A et B, effectuer le calcul dans le CPU, puis stocker le résultat dans M. En 1959, il a paru plus simple à Grace Hopper, au sein du groupe Cobol, d'exprimer des instructions sous formes de transformations des variables: Ajouter B à A puis diviser (la nouvelle valeur de) A par 2. Tout se passe alors comme si les opérations étaient effectuées directement sur la variable, et c'est ce qui est résumé ci-après par l'expression "in situ"[3].

Le nom de Sophus est un hommage à Sophus Lie (1842-1899) qui, avec Felix Klein, a promu la théorie des groupes de transformations, et, en particulier, des transformations effectuées sur des nombres: La spécialité de Sophus est la transformation de variables *in situ*.

Sophus a sa page de téléchargement sur github. Le langage est décrit par un ensemble de fonctions en JavaScript[4] et est assorti d'un interpréteur qui fonctionne dans n'importe quel navigateur internet, même hors ligne. Il est donc possible d'utiliser Sophus dans tout logiciel qui gère le JavaScript comme par exemple CaRMetal, DGPad ou GeoGebra. L'interpréteur, lui aussi programmé en JavaScript, permet de simplifier le code (par exemple en ajoutant automatiquement les parenthèses, et virgules, qui obscurcissent le propos) et d'exporter une traduction encore plus proche du français "naturel". Malgré tout il reste nécessaire d'affiner la traduction avant de la copier-coller dans le Moodle du prof à qui on doit rendre le devoir d'algorithmique demain[5]...

Mais avant de parler d'algorithmes, commençons par les programmes de calcul, qui n'utilisent ni tests ni boucles, et sont donc plus faciles à appréhender.

I/ Changement de cadre

Voici un programme de calcul comme on en pratique couramment en cycle 4 :

---

[3] Les processeurs les plus connus ont d'ailleurs des opérations *in situ*, comme par exemple NEG (sur processeurs x86) qui *remplace* un registre du processeur par son opposé, ou SHL (décalage vers la gauche soit multiplication par 2) qui modifie directement un registre

[4] Par exemple, pour "expliquer" ce que signifie le verbe "tripler", est programmée une fonction qui triple une variable. On peut voir ce qu'elle fait en entrant `montrer tripler` dans la fenêtre de programmation et en lançant le script. On peut même tester `montrer montrer` !

[5] Par exemple `montrer 2+2==4` qui est traduit en *montrer 2+2 est égale à 4* où il est nécessaire d'insérer un "si" (et qui à l'exécution affiche "c'est vrai")

- choisir un nombre
- le multiplier par 3
- ajouter 1

Bien qu'il paraisse simple, ce programme pose des questions (notamment sur l'implicite) et permet d'aborder des notions mathématiques peu évidentes comme la composition non commutative des fonctions ou les fonctions affines. Voici quelques manières de le présenter :

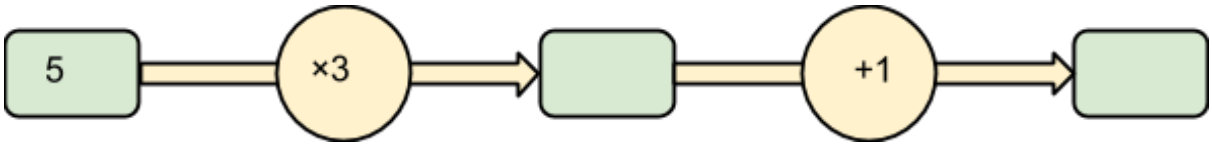
1. Tour de magie

Un magicien[6] *demande à un spectateur choisi au hasard (enfin c'est ce qu'il dit), de choisir un nombre. Puis, les yeux bandés, il lui demande successivement de multiplier ce nombre par 3, puis d'additionner 1 au résultat obtenu. Le spectateur annonce qu'il a alors trouvé 25. Le magicien va alors deviner le nombre choisi au départ. Comment ?*

Cette version montre déjà les sous-entendus de l'énoncé : Quand on dit «ajouter 1», on ne précise pas à quel nombre on est censé ajouter 1. Qui parmi les lecteurs de cet article avait imaginé que ce pût être autre chose que le triple récemment calculé ? De même une instruction implicite est à la fin, puisqu'on ne précise pas ce qu'on est censé faire du résultat du calcul, une fois celui-ci terminé. Le magicien va évidemment demander au spectateur d'annoncer publiquement le résultat. Mais ce qui est évident pour le concepteur d'un énoncé, ne l'est pas toujours pour les lecteurs de cet énoncé...

2. Graphique

Cette version est souvent spontanément utilisée par les élèves[7], en particulier parce qu'elle utilise peu de mots (voire pas du tout) et parce qu'elle permet d'un seul regard d'embrasser tout le programme de calcul[8]:



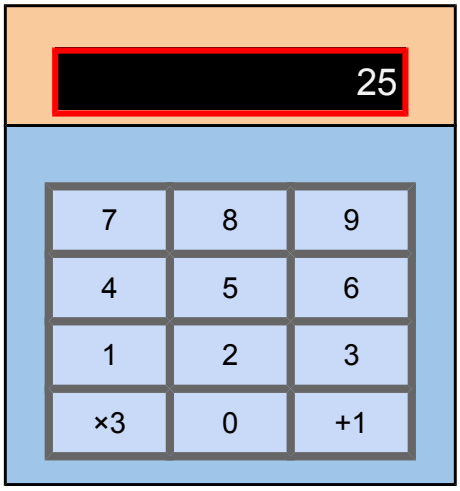
[6] Comme on va le voir, il est capable d'effectuer un calcul mental, c'est pour ça que c'est un magicien, car pour être capable d'effectuer une division par 3 sans calculatrice, il faut être magicien !
[7] On en voit aussi au brevet des collèges, les deux programmes de calcul à comparer dans le sujet "Centres étrangers (Maroc)" de 2015 étant présentés sous cette forme. On admirera au passage l'exploit consistant à représenter graphiquement l'opération "enlever le nombre de départ" qui devrait normalement comporter deux flèches en entrée...
[8] "Un dessin vaut mieux qu'un long discours": Le graphique permet d'économiser de la mémoire de travail, comme on le verra plus bas.

Ici on a mis 5 dans la première case et le but du jeu est de remplir les autres cases et surtout la dernière.

Cette forme permet de donner un caractère ludique (au moins au début) à ce genre d'activité, et de montrer (par le sens des flèches), que l'inverse d'une composée est la composée des inverses prises dans l'ordre inverse (phrase complexe pour dire que $(g \circ f)^{-1} = f^{-1} \circ g^{-1}$; cette notion liée à la non commutativité est loin d'être évidente et peut être considérée comme une sorte d'équivalent algébrique du raisonnement hypothético-déductif, puisque dans les deux cas on «remonte le temps»).

3. Calculatrice cassée

Le XXIe siècle étant celui de la technologie, on va pour être dans l'esprit du temps, imaginer des machines, *a priori* moins dangereuses que des magiciens, pour faire ces opérations successives. L'un de ces outils à la mode en ce moment est un peu post-apocalyptique puisque la calculatrice est cassée : il lui manque des touches. En plus elle est mutante puisqu'à la place des quatre opérations, elle a des opérations plutôt inhabituelles pour une calculatrice[9]:



L'exercice peut s'énoncer ainsi:

La calculatrice de Farid[10] *est un peu cassée. Elle n'a plus de touches d'opérations ni de virgule, et deux touches en bas ont un comportement inhabituel. L'une augmente d'une unité le nombre entré, l'autre le multiplie par 3. Après avoir entré un nombre, Farid a appuyé sur la touche de gauche puis celle de droite. Il a alors obtenu 25.* ***Quel était le nombre de départ ?***

---

[9] D'autres exemples, provenant essentiellement des jeux mathématiques du Monde, se trouvent ici en ligne.

[10] Farid habite dans une banlieue, comme tous les enfants qui sont cités dans les énoncés de ce genre. L'assistante sociale n'étant pas venue depuis des mois, sa calculatrice ne saurait donc qu'être cassée, le fonds social n'ayant pu être mis en œuvre pour la remplacer ou la réparer. Fort heureusement, la voisine du dessous Nirina est malgache et les malgaches sont réputés pour leur talent en réparation d'objets divers. Dans l'espoir de resserrer les liens avec elle, Farid lui a demandé de réparer sa calculatrice et comme on peut le constater, Nirina est moins douée pour réparer les calculatrices que pour faire du charme à son voisin du dessus...

On voit que ce modèle est plus large que celui du départ puisqu'on peut en théorie appuyer plusieurs fois sur les touches des différentes opérations, et ainsi rapidement découvrir la non commutativité et explorer l'espace des nombres qu'on peut obtenir de cette manière.

4. Tableur

Des questions sur les « formules » des tableurs deviennent systématiques au brevet des collèges. Or le tableur, parce qu'il est fait pour créer des tableaux de nombres, permet lui aussi (comme le graphique ci-dessus) d'embrasser tout le problème d'un seul regard. En choisissant de placer les nombres qui peuvent être choisis au départ dans la première colonne puis les étapes successives dans les colonnes suivantes, on obtient quelque chose comme ceci :

1	3	4
2	6	7
3	9	10
4	12	13
5	15	16

Une formule à entrer dans B1 pour avoir par copie vers le bas, la colonne des triples, est

=3*A1

Quant à C1, on peut y entrer

=B1+1

Le tableur est un outil très approprié parce qu'il permet de travailler sur des colonnes de nombres et non sur des nombres, et que comme pour le graphique, il facilite la recherche d'antécédents par la fonction programmée, et en particulier l'importance de remonter les calculs en partant de la fin.

Ceci dit, on peut calculer l'image de tout un tableau de nombres par une fonction numérique, dans la plupart des langages de programmation. Pour cela on dispose en général d'une fonction « map » qui permet de « mapper » la fonction sur un ensemble de nombres, comme dans une mappemonde, où on « mappe »[11] une carte sur une sphère. Dans Sophus, on préfère constituer un tableau *colonne2* à partir des 3x pour x dans *colonne1* :

```
colonne1 = [1..10]
colonne2 = (3*x pour x dans colonne1)
```

[11] On devrait peut-être plutôt dire « napper »...

```
colonne3 = (x+1 pour x dans colonne2)
montrer colonne3
```

On constate au passage la concision avec laquelle on construit des tableaux avec Sophus, la notation [1..10] étant une abréviation de [1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10] soit la liste des nombres entiers entre 1 et 10.

5. Expression algébrique

Le but de ce genre d'exercice est de donner une forme vivante aux expressions algébriques :

- soit x le nombre choisi;
- la première étape fournit 3x (image de x par une fonction linéaire);
- la seconde étape fournit 3x+1 qui est l'image de x (ou plutôt de 3x) par une fonction affine.

Le programme de calcul est donc la version programmée de cette fonction affine, ou si on préfère, la description d'un algorithme permettant de calculer cette fonction. Et deviner le nombre qui a pu donner 25, c'est résoudre l'équation 3x+1=25. Les activités proposées ci-dessous expliquent alors pourquoi on doit *d'abord* faire passer le 1 dans l'autre membre.

Le logiciel de calcul formel Xcas permet d'automatiser le changement de cadre entre l'algèbre (qui est sa spécialité puisque c'est un logiciel de calcul formel) et la programmation. En cliquant sur « prg » puis « nouveau programme » on laisse x comme nom de la variable entrée, et on choisit y comme nom de l'unique variable locale, ensuite on remplit le code du programme ainsi :

```
f(x):={
  local y;
  y := x;
  y := 3*y;
  y := y+1;
  retourne y;
}:;
```

Un clic sur « OK » a pour effet de « compiler » le programme, et ensuite pour savoir ce qu'on obtient lorsqu'on a choisi 3 comme nombre de départ, il suffit d'entrer f(3) dans la console. Mais comme on est toujours dans le calcul formel, on peut aussi faire des choses comme `simplifier(f(x))` ou `resoudre(f(x)=25)` qui concrétisent ce changement de cadre algèbre ↔ programme.

6. Algorithmes

Comme le montre la version Xcas ci-dessus, on peut programmer[12] effectivement une machine pour qu'elle réalise le calcul en question. Or pour *programmer*, il faut un *langage de programmation*, et se pose la question du choix de ce langage : parmi les centaines de langages créés depuis une soixantaine d'années, lequel ou lesquels sont les plus adaptés à l'enseignement du codage ? Y en a-t-il seulement un qui soit « idéal » pour ce genre d'apprentissage ? La question a-t-elle de l'importance ? Quoiqu'il en soit, voici diverses manières de programmer le programme de calcul en Sophus :

Version classique[13]:

```
nombre = nouvelle Variable 3
mettre dans nombre, nombre.valeur*3
mettre dans nombre, nombre.valeur+1
montrer nombre
```

Pas si classique que ça en fait puisque des spécificités sophusiennes apparaissent déjà :

- Une variable est un objet spécifique à ce langage de programmation[14], et doit être créée avant de pouvoir la faire varier.
- On ne doit pas confondre le nom de la variable (ici, « nombre ») avec son contenu (ici, un entier, qui justement varie).
- L'affectation se décrit par l'action de mettre un nombre dans la variable ce qui permet notamment de rappeler qu'une affectation n'est jamais qu'une instruction (c'est-à-dire qu'on intime à la machine l'ordre d'effectuer une action), ce que ne permet pas l'usage du symbole d'égalité[15].
- Les instructions sont rédigées à l'infinitif, fidèlement à un code implicitement pratiqué dans les sujets d'examen où l'infinitif est un impératif déguisé.

D'ailleurs si on soumet ce sophuscript à l'export (« copier-coller »), on obtient la traduction suivante qu'il est assez facile de corriger :

```
nombre = nouvelle variable initialisée à  3
```

[12] On dit aussi « coder » voire « code » puisque cela se fait dans une « coding week » d'ampleur européenne

[13] On peut aussi écrire plus brièvement quelque chose comme ceci :
nombre := 3
nombre devient 3*nombre
nombre devient nombre+1
montrer nombre

[14] En exécutant `montrer Variable` on a la description en JavaScript de cet objet : en gros, de la même manière qu'un point est un objet possédant une abscisse et une ordonnée, une variable de Sophus est un objet possédant une valeur. Tout le reste n'est que broderie autour de cette idée simple mais fondamentale...

[15] Hérité de Fortran où cela avait un sens, comme dans `moyenne = (a+b)/2` qui se compile en une suite d'instructions visant à ce que la variable `moyenne` contienne effectivement la moyenne entre a et b à la fin de l'exécution du programme Fortran. Las! Cela ne s'applique déjà plus à une incrémentation comme on en utilise dans les boucles, puisqu'il y a alors l'écriture d'une égalité i=i+1, qui a laissé perplexes plusieurs générations de mathématiciens. Aussi dès la fin des années 1950, le groupe Algol a préconisé la notation ":=" pour l'affectation, le x:=3 se notant parfois x←3 (ou, avec le logiciel statistique R, x<-3) qui suggère aussi l'idée d'action, voire de mouvement, sous-jacente à une affectation.

```
mettre dans nombre, la valeur de nombre*3
mettre dans nombre, la valeur de nombre+1
montrer nombre
```

En faisant varier des variables :

```
nombre = nouvelle Variable 3
multiplier nombre par 3
augmenter nombre de 1
montrer nombre
```

On voit la traduction presque littérale du graphique avec les flèches : on fait subir à la variable une série de modifications[16], sans avoir à se poser de questions sur la distinction entre la variable et son contenu, et sans avoir à se demander systématiquement la valeur actuelle de la variable. De plus, imaginer ce que fait la seconde ligne ne requiert que deux éléments de mémoire de travail (quelle opération effectuer, et par combien), alors que la version avec affectation en requiert deux de plus (multiplier quelle variable par 3, et que faire du résultat). Or la théorie de la charge cognitive estime à environ cinq le nombre d'éléments de mémoire de travail disponibles à un moment donné chez un élève. Lorsque quatre d'entre eux sont monopolisés par une seule ligne de programme, il ne faut pas s'étonner de ce que des élèves de terminale trouvent plus facile de programmer en Sophus qu'avec d'autres outils « témoin ». *A fortiori* on peut s'attendre à une aide similaire en collège[17].

La version courte

On peut décrire de manière encore plus simple le programme de calcul en question avec un vocabulaire suffisamment riche pour décrire chacune des instructions par un seul verbe :

```
nombre = nouvelle Variable 3
tripler nombre
incrémenter nombre
montrer nombre
```

[16] La théorie de Sophus Lie est basée sur l'étude des structures qui émergent des transformations successives: On obtient alors un groupe en général non commutatif. Ces travaux, qui sont indirectement à l'origine du programme d'Erlangen, viennent de l'étude des équations différentielles avec la notion de groupe à un paramètre : on associe à un nombre a, l'unique solution de l'équation différentielle qui prend une valeur donnée en a, et à un nombre b, la valeur de la solution en question en a+b : celle-ci dépend de a et b, et c'est une opération de groupe. Dans les programmes de calcul, le paramètre est temporel et désigne essentiellement le numéro de la ligne de programme exécutée.

[17] La saturation de la mémoire de travail étant aggravée par le manque de sommeil, l'apprentissage avec lecteur mp3 ou devant un écran, il ne faut pas s'étonner de ce que même Sophus soit insuffisant pour décrire efficacement un algorithme, quand on voit l'hygiène de vie de certains lycéens...

On peut maintenant voir assez facilement le caractère non commutatif de la composition des fonctions, en alternant d'un simple copier-coller les fonctions linéaire et affine : on voit simplement en les testant que les deux programmes de calcul ci-dessous n'ont pas le même effet[18]:

nombre = nouvelle Variable 3 **tripler nombre** *incrémenter nombre* montrer nombre	nombre = nouvelle Variable 3 *incrémenter nombre* **tripler nombre** montrer nombre

II/ Notion de variable informatique

Au programme de codage, la première notion abordée est celle de « variable informatique ». Notion fondamentale puisque pour boucler on utilise une variable appelée en général « compteur » ou « indice » et qui va justement varier au cours du bouclage. Mais notion difficile[19]... Il y a deux façons d'éviter ce genre de difficulté :

- Au lieu d'envisager une variable qui sera successivement égale à 1, puis à 2, puis à 3 etc, on manipule directement la liste des entiers entre 1 et 10. C'est un des modes de représentation des fonctions[20], et c'est un objet statique. Mais on arrive là au sujet des « compréhensions » chères aux pythoniens[21] et syntaxiquement complexes comme on le verra plus bas.
- On manipule des objets spéciaux (appelés « variables » dans Sophus), qui ne sont ni des nombres, ni des chaînes de caractères, ni des tableaux, qui possèdent un attribut appelé « valeur » (et qui lui peut être numérique) et diverses méthodes permettant de faire varier cette valeur.

Il est intéressant de constater que ces deux idées sont apparues en même temps, en 1959, la première chez John McCarthy avec Lisp, la seconde chez Grace Hopper avec Cobol. Il est intéressant de comparer les intentions de ces deux projets : Le premier destiné à des universitaires (intelligence artificielle) et le second, destiné à permettre à des non informaticiens de programmer avec des mécanismes proches du traitement de la langue naturelle. On l'aura compris au vu de ce qui précède, Sophus est le fruit de la seconde idée,

---

[18] Avec un exercice intéressant sur les équations : existe-t-il des nombres de départ pour lesquels les deux programmes de calcul donnent quand même le même résultat ? Autrement dit, résoudre l'équation 3x+1=3(x+1).

[19] Définition d'un algorithme « simple » donnée par des élèves de terminale : un algorithme simple est un algorithme dans lequel on ne modifie pas la valeur d'une variable. Soit, un algorithme sans boucle et où on multiplie à l'envi les variables, chacune ayant un rôle bien précis et dépendant de certaines autres. Par exemple moyenne=(a+b)/2. Autrement dit, des constantes liées entre elles plutôt que des variables...

[20] Le tableau de valeurs, qui peut servir à émettre des conjectures, mais est moins exhaustif que les tableaux de signe et de variation. Ces tableaux considèrent la fonction comme objet mathématique et plus tellement comme « une quantité dépendant d'une autre ».

[21] Les pythoniens sont une sympathique race d'extraterrestres qui savent (et du coup, aiment) programmer en Python. Comme il en est quelques-uns qui enseignent au collège, nul doute que leur contribution à l'enseignement du codage sera intéressante à observer. Cette invasion extraterrestre sera-t-elle endiguée par les milices scratchiennes ?

qui, aux noms (comme « nombre », « liste »...) préfère les verbes[22] (comme « augmenter », « multiplier »). Seulement les verbes ont besoin de compléments et ceux-ci, pour éviter certains implicites, devront être munis d'adverbes. La richesse[23] de Sophus, c'est donc les adverbes.

1. Opérations unaires

Certains verbes existent pour décrire l'effet d'une application linéaire, comme par exemple « tripler ». Combinés à une opération *in situ* (remplacer un nombre par son triple) ils deviennent très faciles à concevoir et peuvent même être l'occasion d'enrichir le vocabulaire (que signifie « sextupler »[24] par exemple ?). Alors, pour peu que v soit le nom d'une variable Sophus, « tripler v » est nettement plus facile à comprendre que « v prend la valeur 3×v »[25] parce que la mémoire de travail est nettement moins sollicitée. Au bac, des élèves ne savent pas, une fois qu'on a effectué la multiplication par 3, quoi faire du résultat :

- Pour que v prenne 3 fois sa valeur, on doit mémoriser qu'il faut effectuer une multiplication (un élément de mémoire de travail), par combien on doit multiplier (un deuxième élément de mémoire), que faire du produit (une affectation) et quelle variable affecter, soit 4 éléments de la mémoire de travail (5 si on compte la multiplication elle-même).
- Pour tripler une variable, on n'a besoin que de deux éléments de mémoire: L'opération de triplement, et la variable à tripler…

Sophus possède les verbes pour les multiplications par la plupart des entiers allant de 2 à 10 ainsi que par 100. Il y a aussi les verbes `incrémenter` (qui est la transformation de base dans Scratch) et `décrémenter`, lesquels représentent évidemment des additions. Pour savoir comment sont programmées ces fonctions, on peut simplement exécuter des scripts comme « montrer incrémenter » qui ont pour effet d'afficher le code JavaScript correspondant à la fonction, ou un message d'erreur si celle-ci n'existe pas.

2. Opérations sans adverbe

---

[22] Dans « Mais pourquoi ont-ils inventé les fractions ? », Nicolas Rouche représente les fractions de plusieurs manières, notamment comme nombres (¾ , c'est le quotient de 3 par 4, c'est donc juste une écriture compliquée pour 0,75) mais aussi comme opérateurs (prendre les ¾ d'une quantité, c'est modifier cette quantité, en la multipliant par 0,75).

[23] Cobol a été créé et est encore utilisé (plus de 50 ans après) par des employés d'administration, de banque ou d'assurances, ce qui le prédestine à des mathématiques financières. C'est donc un langage de programmation idéal pour les maths de STMG. *A priori*, ce devrait être le cas pour Sophus aussi, comme le montrent certains des exemples ci-dessous. Ce n'était pas le but initial de la création de ce langage, et il faudrait se garder d'en induire un quelconque intérêt (!) de ses auteurs pour la finance...

[24] N'ayant pas trouvé de verbe pour le remplacement par le septuple ou le nonuple, ces verbes n'ont pas été implémentés dans Sophus. De même, il n'y a pas de verbes pour les divisions par des constantes, seul le verbe « décimer » qui signifie « diviser par 10 » ayant été trouvé. Sa connotation macabre et son isolement l'ont fait éviter.

[25] Outre le fait qu'il y a confusion entre le nom de la variable (« v ») et sa valeur (on ne multiplie pas un nom par 3), il y a également confusion entre l'ancienne valeur de v (celle qui va être multipliée par 3) et la nouvelle (ce que v est devenu). En plus, on constate qu'une instruction est ici rédigée à l'indicatif, ce qui lui enlève son statut d'instruction, celles-ci étant habituellement rédigées à l'impératif (ou, par convention héritée des sujets de bac, à l'infinitif, choix effectué pour Sophus ; on constate qu'en anglais, l'impératif et l'infinitif s'écrivent de la même manière)

Si on veut ajouter non pas 1 mais 2 à une variable, ou bien on répète deux fois l'incrémentation de la variable[26], ou bien on utilise une opération générale mais avec un autre complément d'objet: La valeur de l'incrément. En Sophus on écrira alors « augmenter v de 2 ». Oui mais de 2 quoi ? Ah ces implicites ! On conviendra que ce sont des unités lorsqu'il s'agit d'augmentation ou de diminution, de nombres sans unité lorsqu'il s'agit de multiplier ou diviser. Ces unités peuvent d'ailleurs être précisées du moment qu'on les met après le symbole « dièse » qui en fait des « commentaires » (du texte ne faisant pas partie du langage de programmation donc non exécuté). Par exemple si la tirelire de Tim contient 13,65 € et que Tim y ajoute 2,5 €, on peut calculer le contenu de la tirelire avec ce script :

```
tirelire = nouvelle Variable 13.65 # €
augmenter tirelire de 2.5 # €
montrer tirelire
```

On peut aussi augmenter un tableau de nombres d'un autre tableau de nombres (pour calculer l'image d'un point par une translation par exemple), voire augmenter une chaîne de caractères d'une autre chaîne de caractères, comme le montre cet exemple (difficile mais cité en exemple dans le programme du cycle 4) de génération d'une table de conjugaison :

```
pronom = ["je", "tu", "il", "elle", "nous", "vous", "ils", "elles"]
terminaison = ["e", "es", "e", "e", "ons", "ez", "ent", "ent"]
verbe = nouvelle Variable " "
alert "Entrer un verbe du premier groupe"
entrer verbe
racine =  verbe.valeur[..-3]
conjugaison = [ ]
pour i dans [0..7]
        verbe.valeur = racine
        augmenter verbe de terminaison[i]
        conjugaison.empiler "#{pronom[i]}  #{verbe.valeur}"
montrer conjugaison
```

La racine verbale est obtenue à partir de l'infinitif, en gardant toutes ses lettres sauf les deux dernières[27]; par exemple si le verbe est « sextupler » la racine est « sextupl ». Ensuite, on boucle sur l'indice i allant de 0 (pour le pronom « je ») à 7 (pour le pronom « elles »); dans cette boucle on remplace le verbe déjà conjugué par sa racine, on augmente le résultat de la terminaison verbale correspondant au pronom (par exemple « ons » pour le pronom « nous » ce qui donne alors « sextuplons »), et on ajoute le pronom devant, ce qui donne quelque chose comme « nous sextuplons ». Enfin la dernière action de la boucle est d'empiler les diverses phrases obtenues dans la table de conjugaison pour n'afficher qu'une seule fois celle-ci, à la fin.

Pour convertir un angle de degrés en radians, on le multiplie par π/180, ou ce qui revient au même, on multiplie par π radians et on divise par 180° : Une conversion, c'est une règle de trois[28] :

---

[26] On peut, voir la partie sur les boucles
[27] -3 et non -2 parce qu'en informatique on compte à partir de 0, pas de 1. Ah ces informaticiens !
[28] Cet exemple vient du logiciel GeoGebra et semble remonter à son créateur Markus Hohenwarter

```
angle = nouvelle Variable 60 # °
multiplier angle par π
diviser angle par 180
montrer angle # en radians maintenant
```

Donc multiplier par un nombre c'est selon le contexte, multiplier par un nombre sans unité (cas du triplement qui est une double addition x+x+x), ou multiplier par un coefficient de conversion qui a pour unité le quotient de deux unités (vitesse en km/h, masse spécifique en kg/m^3, conversions monétaires en $/€ voire cm pour « convertir » une distance en aire). Cependant, il y a des unités liées aux nombres abstraits que sont les fractions, et dans ce cas Sophus utilise le dénominateur comme adverbe, destiné à modifier le sens du complément.

3. Le dénominateur comme adverbe du numérateur

Une nouvelle ambiguïté apparaît ici, par exemple si on veut diminuer un nombre d'un dixième (ça se fait dans le commerce quand il y a des promotions) :

```
prix = nouvelle Variable 80 # €
diminuer prix de 1/10 # d'€
montrer prix
```

Le résultat affiché montre que diminuer le prix de 1/10 revient à lui enlever *un dixième d'euro*, ce qui en général pas l'intention du commerçant[29] souhaitant solder un article. En fait c'est d'*un dixième du prix* que le commerçant veut réduire celui-ci. Le taux d'échec en section technologique sur les exercices portant sur les suites arithmétiques (placements à intérêts simples) et géométriques (placements à intérêts composés) est un excellent témoin de cette difficulté. La solution de l'exercice précédent est :

```
prix = nouvelle Variable 80 # €
diminuer prix de 1 dixième
montrer prix
```

Ainsi, « dixième » est un modificateur de « 1 » (ou plutôt de son sens), autrement dit un *adverbe* qui précise la manière dont on doit considérer ce nombre 1. Des exercices de révision sur les fractions sont désormais rapides à créer avec Sophus. Du moins avec les dénominateurs les plus communs (les entiers de 2 à 10, et le cas particulier de 100 qui ne s'écrit pas « centièmes » mais « pourcents »; voir ci-après). Avec le risque que les élèves oublient comment on fait pour diminuer d'un dixième[30].

Parfois, même si c'est plutôt rare, on a besoin de connaître le montant de la diminution avant d'effectuer la soustraction[31]. C'est par une règle de trois qu'on trouve cela. Or Sophus

[29] Du moins pas officiellement...

[30] Bien que la notion de coefficient multiplicateur figure au programme de STMG, il est bon de rappeler qu'on ne peut guère oublier ce qu'on ignore...

[31] Par exemple parce qu'on veut détailler la suite des calculs, par écrit pour décharger la mémoire de travail et pour s'habituer à expliciter la démarche (autrement dit, à raisonner algorithmiquement). La mode des exercices à prise d'initiative insufflée par les mauvais résultats de PISA laisse présager

n'a pas de verbe « prendre » qui permettrait par exemple de dire que le dixième de 80 € c'est 8 €: Que ferait-on en effet de ce qu'on a pris ? La solution est donnée par les calculatrices « 4 opérations » des commerçants.Au lieu de *prendre*, on *multiplie* :

```
prix = nouvelle Variable 80 # €
multiplier prix par 1 dixième
montrer prix
```

Il y a là encore un implicite : « prendre le dixième » d'une quantité, c'est multiplier cette quantité par $\frac{1}{10}$. Autrement dit, quand on parle de fractions, « de » se traduit par « multiplier ». C'était autrefois enseigné en collège, mais bien peu de lycéens semblent le savoir… Pour *prendre la moitié des deux tiers*, on multiplie donc par le produit des deux fractions $\frac{1}{2}$ et $\frac{2}{3}$ qui est $\frac{1}{3}$. Sophus ne permet pas de le vérifier d'un coup mais si on assimile les fractions à des opérateurs de règle de trois, on applique deux opérateurs l'un derrière l'autre :

```
prix = nouvelle Variable 120 # €
multiplier prix par 2 tiers
multiplier prix par 1 demi
montrer prix
```

Enfin la division par une fraction est assez facile à explorer avec Sophus, ce qui peut aboutir à un rappel, rarement superflu de la règle « diviser par une fraction c'est multiplier par son inverse » qui a l'avantage d'éviter de se tromper d'inverse (on inverse le dividende au lieu du diviseur, voire les deux, typique chez des élèves -- nombreux -- pour qui les fractions sont vides de sens) et d'aisément se généraliser à des irrationnels...

4. Cas particulier des pourcentages

Le symbole « % » est obtenu par déformation de la division « /100 » (on a enlevé le 1 et déplacé un 0). Cela rappelle qu'un pourcentage n'est jamais qu'une fraction de dénominateur 100. Alors tout ce qui a été dit ci-dessus peut être réinvesti avec le langage particulier des pourcentages : en particulier, diminuer un prix de 10 pourcents, ce n'est pas la même chose que le diminuer de 10 euros. Le savoir-faire de ce genre d'exercice, qui est inexistant chez beaucoup d'élèves de sections technologiques, ayant été transféré[32] à Sophus, celui-ci permet donc d'aborder automatiquement des exercices comme

- Est-ce que deux augmentations successives de 20 % équivalent à une augmentation de 40 % ?

---

toutefois que ce genre de question disparaisse des sujets d'examen.

[32] Dans certaines calculatrices 4 opérations, la touche « % » joue le même rôle d'adverbe : prendre 20 % d'un nombre se fait avec la séquence `×20%` alors qu'augmenter le nombre de 20 % se fait avec la séquence `+20%`

```
prix = nouvelle Variable 100
2 fois faire augmenter prix de 20 pourcents
montrer prix
mettreDans prix, 100
augmenter prix de 40 pourcents
montrer prix
```

- Une augmentation de 20 pourcents est-elle totalement neutralisée par une diminution de 20 pourcents ?

```
prix = nouvelle Variable 100
augmenter prix de 20 pourcents
diminuer prix de 20 pourcents
montrer prix
```

- Le pouvoir d'achat d'un enseignant diminue de 2 % par an. Descendra-t-il à la moitié de sa valeur initiale avant la fin de la carrière de l'enseignant ?

```
pouvoir = nouvelle Variable 100
année = nouvelle Variable 0
jusqu'à ce que pouvoir.valeur <= 50
        diminuer pouvoir de 2 pourcents
        incrémenter année
montrer année
```

III/ Les tests et instructions conditionnelles

Comme disait George Bernard Shaw, « celui qui sait comment il faut faire, il fait; celui qui ne sait pas comment il faut faire, il explique comment il faut faire » : Jusqu'à Brouwer et Gödel, les mathématiciens étaient trop occupés à faire des maths pour réfléchir à leur manière de fonctionner. Il ne faut donc pas s'étonner que le premier à analyser le raisonnement hypothético-déductif ait été un non mathématicien : Aristote. Or même Aristote, avec la notion de syllogisme pratique, donnait à l'expression « si … alors » plusieurs sens, qu'on retrouve aujourd'hui dans ces propositions :

Si ce triangle est rectangle alors ses angles aigus sont complémentaires	La véracité de la prémisse entraîne celle de la conclusion : lien entre propositions
Seulement si ce triangle est rectangle, ses angles aigus sont complémentaires	Réciproque de celle d'au-dessus ; elle est fausse en géométrie hyperbolique
S'il fait beau alors tu me trouveras à la plage	La réciproque « s'il pleut alors je ne serai pas à la plage » est sous-entendue
Si tu as soif alors sers-toi un verre d'eau	Ce qui est après « alors » n'est pas une

(syllogisme « pratique » d'Aristote)	proposition (elle n'est ni vraie ni fausse) mais un ordre

Donc, à moins de considérer les instructions d'un programme comme des propositions de logique déontique[33], le sens du mot « si » en programmation impérative est différent des différents sens de ce mot en logique. C'est une difficulté qui se greffe sur d'autres lorsqu'on veut rédiger des algorithmes: Il ne s'agit pas ici de relation de cause à effet, mais d'instruction conditionnelle ; et que doit-on faire si on n'a pas soif[34]? Dans ce cas, en programmation impérative, on ne fait rien. On choisit d'être économe comme le recommandait Guillaume d'Occam.

1. Les booléens en Sophus

Le terme λόγοσ (« logos ») désigne, en grec ancien, à la fois la raison et le langage. Sans aller jusqu'à dire que le raisonnement est un langage, on peut considérer que le langage est un support pour le raisonnement: «Ce qui se conçoit bien s'énonce clairement, et les mots pour le dire viennent aisément[35]», disait Boileau. La contraposée de la première partie de cette citation pourrait, du point de vue d'un correcteur de copies désespéré[36], se réécrire «si les mots ont du mal à arriver sur le papier, c'est que le concept n'est peut-être pas clair dans la tête de leur auteur». Lorsqu'on veut décrire un algorithme, ce besoin de précision dans le langage est encore accru par l'objectif ultime de traduire le résultat dans un langage de programmation destiné à une machine qui, contrairement au correcteur de copies, ignore la notion de bénéfice du doute[37].

Pour être cohérent avec une telle exigence, il convient donc qu'un langage de programmation rédige les propositions (ou « booléens ») d'une manière à la fois rigoureuse et compréhensible. Voici la proposition de Sophus sur les propositions:

```
proposition = (2+2 est différent de 4)
montrer proposition
```

L'exécution de ce script provoque l'affichage de « c'est faux » qui a le mérite de rappeler qu'il ne suffit pas d'affirmer quelque chose pour que ce soit vrai. En fait on peut avoir l'affichage « c'est vrai », mais avec cette variante :

[33] En notant ○p la proposition déontique « il est obligatoire que p », l'instruction « si x est positif alors mettre 2 dans y » se traduit en logique déontique par x>0⇒○(y=2). On remarque que l'axiome de Kripke ○(p⇒q)⇒[○p⇒○q] permet de démontrer ce genre de proposition s'il est obligatoire que x soit positif lors de l'exécution du programme. Mais est-ce obligatoire? Par ailleurs, comment faire pour que y soit égal à 2 ? Enfin, il est bon à ce stade de rappeler que la logique déontique n'est pas au programme...

[34] Le magicien du début, on l'a vu, est doué en calcul mental. Mais comme il est aussi doué en calcul rénal, son néphrologue lui a intimé l'ordre de boire de toute façon un verre d'eau, même s'il n'a pas soif. Ce qui n'invalide pas la proposition « si tu as soif alors sers-toi un verre d'eau » puisque celle-ci ne dit pas ce qu'il faut faire en cas de satiété, et donc n'interdit pas de se servir un verre d'eau...

[35] À condition d'avoir suffisamment de vocabulaire pour que les mots existent. Ce qui n'existe pas ne peut pas venir aisément...

[36] Ah bon vous aussi ça vous est arrivé ?

[37] Certains élèves de terminale prétendent que le prof de maths lui aussi ignore la notion de bénéfice du doute lors de la correction de leur copie...

```
proposition = (2+2 est différent de 4)
montrer le contraire de proposition
```

Cela permet d'explorer les tableaux de valeurs de la double négation, la conjonction et la disjonction par l'informatique, en étendant cet exemple avec des boucles :

```
a = (1<3<5)
b = 2>=3
montrer (a et b)
montrer (a ou b)
```

Les réponses « c'est faux » et « c'est vrai » sont surprenantes en phase d'acquisition de compétences, comme l'est l'affirmation par le prof que la proposition « ce triangle est rectangle ou isocèle » pour un triangle 3-4-5, parce que comme il est rectangle la proposition ne devrait pas être entièrement fausse. On peut alors se demander l'intérêt d'une exploration algorithmique par rapport à l'exploration géométrique; la réponse est hélas simple : contrairement à la géométrie, l'algorithmique est au programme de maths…

Il y a d'autres manières d'obtenir des booléens dans Sophus sans avoir à écrire True ou False ; par exemple montrer (13 dans [1..5]) affiche « c'est faux » ainsi que montrer t.estPair() si la variable t est initialisée à 13 et pas modifiée depuis.

2. Instructions conditionnelles

Aujourd'hui Farid a enfin réussi à inviter sa voisine du dessous. Pour faire bien les choses, il va à la boutique kasher du quartier, qui fait justement des promotions. L'affiche en grand dit «réduction de 5% sur toutes les factures dépassant 10 €». Pour augmenter la probabilité de ~~pécho~~ séduire la voisine, il va donc acheter toutes sortes de boissons. Voici un extrait de son ticket de caisse :

un pack de 6 bouteilles de ***Youps*** (yaourt à boire)	5,99 €
une boîte de 33cl de ***Raides Bulles*** (soda à la taurine)	1,99 €
un sachet de ***Pies Nettes*** (fruits secs)	3,79 €

Le script Sophus permettant de savoir le montant de la facture doit donc effectuer un test sur la valeur du prix total afin de savoir s'il convient, ou non, d'appliquer la réduction :

```
prix = nouvelle Variable 0
```

```
augmenter prix de 5.99
augmenter prix de 1.99
augmenter prix de 3.79
Si prix.valeur > 10
        diminuer prix de 5 pourcents
montrer prix
```

Il y a là encore un sous-entendu : si Farid, constatant qu'il n'a que 10 € sur lui, demande à ne plus acheter le sachet de fruits secs[38], il est tacitement convenu que la réduction ne s'applique pas.
On constate que les instructions qui ne doivent s'effectuer que si le prix dépasse 10 € (ici il n'y en a qu'une) sont indentées (reculées par rapport au début de la ligne) par rajout de caractères d'espace devant les instructions. Une convention issue de la programmation en Python est de choisir 4 espaces par niveau de bloc d'instruction (on peut en effet avoir besoin d'effectuer un nouveau test dans un bloc d'instructions conditionnelles, et surtout on verra plus bas que les boucles aussi font appel à des blocs indentés).

Si, comme ici, il n'y a qu'une instruction conditionnelle, on peut la mettre sur la même ligne que le test, mais dans ce cas il faut utiliser le mot « alors » :

```
prix = nouvelle Variable 0
augmenter prix de 5.99
augmenter prix de 1.99
augmenter prix de 3.79
Si prix.valeur > 10 alors diminuer prix de 5 pourcents
montrer prix
```

Une manière de se passer du mot « alors » est d'inverser le test et l'instruction :

```
prix = 5.99+1.99+3.79
prix *= 0.95 Si prix > 10
montrer prix
```

Mais cette version tend à saturer la mémoire de travail et plusieurs élèves ont tendance à effectuer la remise dans tous les cas, sans doute par manque de perception de la fin de la phrase. On reviendra sur ce problème dans les exemples de « compréhensions ».

3. Alternatives

Un programme de calcul est un moyen de représenter l'usage successif des 4 opérations avec éventuellement la mise en mémoire d'un résultat partiel (boutons M, M+,...). Dans la plupart des langages de programmation, la mise en mémoire se fait par l'affectation de variables qui servent à cela : mémoriser la valeur actuelle d'un nombre dont on a besoin au cours du calcul. Maintenant, les programmes de calcul du brevet des collèges ne font appel

[38] Pas besoin d'enlever toute la ligne 4, il suffit de la faire précéder du caractère « dièse » (également connu sous le nom de « hashtag »). De toute manière le raccourci « Control-Z » qui permet d'annuler les modifications du script, fonctionne bien sur les navigateurs Internet Chrome et Firefox.

qu'à 3 des 4 opérations[39]: addition, soustraction, multiplication. En effet, utiliser un programme de calcul avec division, c'est courir le risque qu'il y ait des « valeurs interdites ». Voici un exemple issu des « jeux mathématiques du Monde[40] », datant du 5 décembre 2000 et intitulé « le moulin à nombres[41] » :

Prenez un nombre et entrez-le dans le **moulin à nombres**.
Cet appareil va "mouliner" votre nombre de la façon suivante :

- il le multipliera par 2 ;
- retranchera 7 ;
- divisera le résultat par le nombre initial augmenté de 1 ;
- recrachera le quotient obtenu.

Si vous n'avez pas coupé l'alimentation, le moulin recommencera la même suite d'opérations à partir du nombre précédemment restitué.

Vous entrez le nombre 2001 et moulinez 2001 fois. Quel résultat s'affichera ?

Et en partant d'un autre nombre ?

Voici une traduction en Sophus, qui permet de voir qu'il est nécessaire d'avoir deux variables initialisées à la valeur de départ, parce que la première sera transformée indépendamment de la seconde :

```
nombre = nouvelle Variable 2001
diviseur = nouvelle Variable nombre.valeur
doubler nombre
diminuer nombre de 7
incrémenter diviseur
diviser nombre par diviseur
montrer nombre
```

Oui mais qu'est-ce qui se passe si, au lieu de 2001, on commençait par -1 ? La modification est simple à faire dans le script ci-dessus, il suffit de remplacer 2001 par -1, mais là, Sophus affiche: -∞. Cet affichage est à interpréter d'une manière différente selon les connaissances que l'on a en analyse :

- En cycle terminal, il signifie que la limite en -1 de la fonction homographique $\frac{2x-7}{x+1}$ est -∞ (ce qui est vrai mais en supposant qu'on a approché -1 par la droite[42]);
- Avant cela, le résultat affiché n'étant visiblement pas numérique (« non ce n'est pas un 8 couché, regardez bien l'angle que font les deux branches en le centre de

[39] L'élévation à une puissance est en effet un cas particulier de multiplication: Élever un nombre au carré, c'est le multiplier par lui-même.
[40] Dont « l'intégrale » a été publiée par les éditions Pôle avec l'ISBN 978 284 884 0741
[41] Le but ici n'est pas d'appliquer le programme de calcul mais d'itérer son application, il s'agit en effet d'un système dynamique (ou si on préfère de l'étude des puissances d'une matrice)
[42] Cette convention fait partie de la norme IEEE 754

symétrie »), est à interpréter comme un blocage avec échec du moulin à calcul: « On ne peut pas diviser par 0[43] ».

Le moulin n'est donc pas censé fonctionner avec tout nombre puisque -1 ne convient pas. C'est une bonne raison pour effectuer un test : faire quelque chose de différent selon que la valeur entrée est -1 ou autre que -1 :

```
nombre = nouvelle Variable 2001
Si nombre.valeur == -1
        alert "Le moulin est bloqué: Valeur interdite"
Sinon
        diviseur = nouvelle Variable nombre.valeur
        doubler nombre
        diminuer nombre de 7
        incrémenter diviseur
        diviser nombre par diviseur
        montrer nombre
```

Pour faire mouliner le moulin, on gagne à créer une fonction qui va accomplir ces opérations, et qu'on va baptiser unTour(). Après cela il suffira de l'appeler pour effectuer un tour :

```
unTour =  ->
        diviseur = nouvelle Variable nombre.valeur
        doubler nombre
        diminuer nombre de 7
        incrémenter diviseur
        diviser nombre par diviseur

nombre = nouvelle Variable 2001
unTour()
montrer nombre
```

L'effet obtenu est le même qu'auparavant mais cette fois-ci il est plus facile de répéter 2001 fois l'opération, il suffit de compléter l'avant-dernière ligne à cet effet :

```
unTour =  ->
        diviseur = nouvelle Variable nombre.valeur
        doubler nombre
        diminuer nombre de 7
        incrémenter diviseur
        diviser nombre par diviseur
```

[43] Sauf Chuck Norris qui peut tout faire, mais on dit que c'est Euler qui le lui a appris...

```
nombre = nouvelle Variable 2001
2001 fois faire unTour()
montrer nombre
```

On pourra, à juste titre, objecter qu'une fonction est un objet bien plus abstrait que les transformations de variables, d'autant que cette fonction n'a pas d'argument (ou antécédent)[44]. À cette objection, on peut objecter :

- que des fonctions seront probablement utilisées rapidement en Logo (initialiser, cacher la tortue, lever le crayon etc) ;
- Que l'affichage exporté par Sophus est plus parlant que le script :

```
unTour =   faire
       diviseur = nouvelle variable initialisée à la valeur de nombre
       doubler nombre
       diminuer nombre de 7
       incrémenter diviseur
       diviser nombre par diviseur

nombre = nouvelle variable initialisée à  2001
2001 fois faire unTour()
montrer nombre
```

- Qu'on peut aussi faire sans fonction, mais avec des indentations :

```
nombre = nouvelle Variable 2001
2001 fois faire  ->
        diviseur = nouvelle Variable nombre.valeur
        doubler nombre
        diminuer nombre de 7
        incrémenter diviseur
        diviser nombre par diviseur
montrer nombre
```

4. Comprendre les compréhensions

Si on veut la liste des carrés inférieurs ou égaux à 100, on peut la construire par un algorithme :

- On commence par créer une liste vide;
- On boucle sur les nombres allant de 1 à 10; pour chacun d'entre eux, on ajoute son carré à la liste.

On obtient alors quelque chose comme ceci:

[44] Ceci dit les fonctions sans argument sont connues de tous ceux qui simulent des lancers de dés avec un tableur : Alea est une fonction, et se note donc Alea() pour éviter de faire faire n'importe quoi au tableur.

```
carrés = [ ]
pour x dans [1..10]
        carrés.empiler x*x
montrer carrés
```

Mais il y a plus simple : on peut directement construire la liste comme celle des x² pour x entre 1 et 10 :

```
carrés = (x*x pour x dans [1..10])
montrer carrés
```

On appelle cela la description de la liste *par compréhension*. Ce terme, issu du monde de Python et Ruby, est un peu malheureux parce que beaucoup de lycéens bloquent sur ce genre de description statique : on voit là encore en action la fameuse saturation de la mémoire de travail. Le phénomène s'aggrave encore si en plus on veut mettre un test dans la description, comme par exemple si on veut les carrés pairs :

```
carrésPairs = [ ]
pour x dans [1..10]
        Si x%2 == 0
                carrésPairs.empiler x*x
montrer carrésPairs
```

Cet algorithme est complexe parce qu'il y a un test dans une boucle (on n'empile que les carrés des nombres pairs). Mais la version courte est moins bien comprise[45] à cause de l'inversion entre le test et ce qui en résulte :

```
carrésPairs = (x*x pour x dans [1..10] quand x%2==0)
montrer carrésPairs
```

C'est dommage, parce que le crible d'Eratosthène est assez facile à programmer avec ces « compréhensions » (ici, le calcul des nombres premiers inférieurs à 200) :

```
crible = [2..200]
pour d dans [2..200]
        crible devient (x pour x dans crible quand x<=d ou x%d != 0)
montrer crible
```

IV/ Répétitions

1. Les répétitions de Logo

[45] Les non germanophones qui du mal à comprendre une phrase qui a été rédigée en allemand, parce que les verbes tous reportés à la fin, cela, ont, ont été, savent bien. Sauf Maître Yoda (mais lui, il est allemand).

La plupart des langages de programmation permettent de boucler à l'aide d'un indice que l'on affecte par incrémentation au cours du bouclage. Pourtant les enfants qui programment en Logo apprennent très vite comment dessiner un carré : en répétant 4 fois quelque chose. Cette façon de faire, hélas absente de la plupart des langages de programmation, a donc été adoptée dans Sophus. Par exemple, on place 250 € à 2 pourcents par an pendant 10 ans. Ce sont des intérêts composés, c'est-à-dire que la banque va à la fin de chaque année, calculer le montant du capital (c'est-à-dire augmenter celui-ci de 2 pourcents) et l'arrondir au centime le plus proche, puis replacer le nouveau capital. On voudrait savoir s'il ne serait pas plus avantageux de ne faire le calcul qu'une fois au bout des 10 ans, car les erreurs d'arrondi risquent de s'accumuler pendant les 10 ans. Voici l'algorithme proposé :

- On crée deux variables *capital* et *ct* (comme « capital théorique »), initialisées à 250.
- On répète 10 fois les deux opérations consistant à élever *capital* de 2 % puis arrondir au centime près.
- On fait pareil pour *ct* mais sans l'arrondi.
- On soustrait *ct* à *capital* pour savoir de combien on s'est fait arnaquer.
- On affiche le résultat.

Le script :

```
placement = 250
capital = nouvelle Variable placement
10 fois faire
        augmenter capital de 2 pourcents
        arrondir capital à 2 décimales
ct = nouvelle Variable placement
10 fois faire augmenter ct de 2 pourcents
diminuer capital de ct
montrer capital
```

L'affichage 0,001 apprend que dans ce cas, on a gagné un dixième de centime avec cette méthode. Mais on a tout intérêt (!) à essayer d'autres valeurs du placement et du taux d'intérêt pour voir ce que ça change. Par exemple si on avait placé 200 € on aurait perdu presque un centime...

2. Les répétitions avec indice

Pour additionner les nombres impairs, la méthode précédente fonctionne mais elle est peu pratique (il faut créer et gérer l'indice)[46] :

```
somme = nouvelle Variable 0
indice = nouvelle Variable 1
liste = [ ]
```

[46] Par contre c'est pratique pour calculer des intégrales par la méthode des rectangles: On augmente x de dx et on augmente l'intégrale de f(x)dx. On doit juste préalablement calculer dx à partir du nombre de répétitions et de la longueur de l'intervalle.

```
10 fois faire
        augmenter somme de indice
        augmenter indice de 2
        liste.empiler somme.valeur
montrer liste
```

Là encore, l'export est plus lisible que le script :

```
somme = nouvelle variable initialisée à  0
indice = nouvelle variable initialisée à  1
liste = [ ]
10 fois faire
      augmenter somme de indice
      augmenter indice de 2
      empiler la valeur de somme dans liste
montrer liste
```

Mais, pour le confort de programmation, on peut aussi faire varier l'indice dans une boucle à la syntaxe plus classique :

```
somme = 0
liste = [ ]
pour indice dans [1..20] par pas de 2
        somme devient somme+indice
        liste.empiler somme
montrer liste
```

Sans parler des incompréhensibles compréhensions :

```
somme = 0
somme += indice pour indice dans [1..20] par pas de 2
montrer somme
```

3. Condition de sortie

Il arrive comme avec l'algorithme d'Euclide ou la suite de Collatz, qu'on ne sache pas d'avance quand arrêter de boucler. On a alors besoin d'une condition de sortie, qui est un booléen (une proposition logique). La plupart des langages de programmation utilisent le mot-clé « while » pour matérialiser de telles boucles. Mais la négation ayant elle aussi tendance à saturer la mémoire de travail[47], « tant qu'on n'a pas fini » est jugé plus compliqué, conceptuellement, que « jusqu'à ce qu'on ait fini ». Le langage Pascal possédait un « until » qui a malheureusement disparu des langages de programmation ultérieurs, à l'exception de CoffeeScript qui a servi de support à Sophus et permet donc de programmer selon les deux paradigmes.

[47] Le « vous n'êtes pas sans ignorer que… » est rarement perçu comme une erreur parce que, donnant une triple négation, il est perçu comme double négation par toute mémoire de travail n'ayant plus, pour une raison ou une autre, que deux emplacements disponibles. C'est la raison pour laquelle on déconseille en général l'usage de doubles négations.

Par exemple, la suite de Collatz version double négation :

```
u = nouvelle Variable 13
liste = [u.valeur]
Tant que u.valeur > 1
    Si u.estPair()
        diviser u par 2
    Sinon
        tripler u
        incrémenter u
    liste.empiler u.valeur
montrer liste
```

Et la version naturelle :

```
u = nouvelle Variable 13
liste = [u.valeur]
Jusqu'à ce que u.valeur == 1
    Si u.estPair()
        diviser u par 2
    Sinon
        tripler u
        incrémenter u
    liste.empiler u.valeur
montrer traduction
```

Difficile de résister à l'envie de programmer d'autres algorithmes classiques de cette manière :

- L'algorithme d'Euclide :

```
pgcd = (a,b) ->
        Jusqu'à ce que b == 0
                [a,b] devient [b,a%b]
        a

montrer pgcd 32, 24
```

- l'algorithme babylonien (ou de Heron d'Alexandrie) pour calculer une racine carrée à 3 décimales près[48] :

```
rac = (x) ->
    Si x>=0
       [a,b] = [1,x]
       Jusqu'à ce que -0.0001<b-a<0.00001
          a = (a+b)/2
          b = x/a
```

[48] On remarque que la condition de sortie peut s'écrire sans utiliser de valeur absolue, grâce à un encadrement

```
        a
    Sinon
        "imaginaire"

montrer rac 16
```

(la fonction renvoie soit la valeur de a, soit le mot « imaginaire » selon le signe de x).

- Résolution de l'équation $x^3=25$ par dichotomie :

```
borneInf = 2
borneSup = 3
Jusqu'à ce que borneSup-borneInf<0.0001
    moyenne = (borneInf+borneSup)/2
    Si cube(moyenne)<25
        borneInf devient moyenne
    Sinon
        borneSup devient moyenne
montrer moyenne
```

- Calcul d'un logarithme népérien par l'algorithme de Briggs :

L'algorithme consiste à construire parallèlement une suite de nombres et leurs logarithmes. On extrait itérativement la racine de x jusqu'à ce le résultat soit proche de 1; dans ce cas une bonne approximation de son logarithme est 1 de moins que lui ; ensuite on double ce nombre autant de fois qu'on avait extrait la racine carrée.

```
ln = (x) ->
    Si x >= 0
        compteur = nouvelle Variable 0
        y = nouvelle Variable x
        Jusqu'à ce que 0.999<y.valeur<1.001
            extraireLaRacineDe y
            incrémenter compteur
        diminuer y de 1
        compteur.valeur fois faire
            doubler y
    y.valeur

montrer ln 2
```

- Calcul d'une intégrale par la méthode des rectangles

On souhaite calculer une valeur approchée de $\int_1^3 \frac{1}{1+x^2}dx$ avec la méthode des rectangles à gauche: On divise l'intervalle [1;3] en 1000 subdivisions de longueur dx=0,002 (un millième de la longueur de l'intervalle) et on crée une variable I à laquelle on va additionner des expressions de la forme f(x)dx (ce qui justifie la notation intégrale de Leibniz pour des élèves de terminale pour qui cette notation est souvent vide de sens). Une amélioration de l'algorithme consiste à mettre dx en facteur, c'est-à-dire additionner des f(x)

et seulement quand on a fini, multiplier « l'intégrale » par dx. Une seule difficulté conceptuelle subsiste : on doit s'arranger pour que la fonction f porte non sur un nombre, mais sur une variable.

```
f = (x) ->          # f est fonction de x
        y = nouvelle Variable x.valeur
        élever y au carré
        augmenter y de 1
        inverser y
        y

[a,b] = [1,3]     # intervalle sur lequel on va intégrer

dx = nouvelle Variable b
diminuer dx de a
diviser dx par 1000

I = nouvelle Variable 0
x = nouvelle Variable a
jusqu'à ce que x.valeur >= b
        augmenter I de f(x)
        augmenter x de dx

multiplier I par dx
montrer I
```

L'exécution du script indique que l'intégrale vaut environ 0,464 et il est facile d'obtenir plus précis en remplaçant 1000 par 1000000 (mais c'est plus long, et il faut remplacer « montrer » par « alert ») ou de modifier la fonction ou l'intervalle. En plus, il est facile de modifier ce script pour avoir, au lieu de la méthode des rectangles à gauche, celle des rectangles à droite.

V/ Jeu de nim

Voici une ébauche de jeu de nim programmé en Sophus. Il s'agit ici du jeu dit de la soustraction, où on enlève 1, 2 ou 3 objets d'un tas (de haricots par exemple). Le gagnant est celui qui a enlevé le dernier objet du tas. Le script a l'air long parce qu'on cherche à contrôler la qualité des choix du joueur.

```
auHasard = (n) -> Math.ceil Math.random()*n
tas = nouvelle Variable 21
jusqu'à ce que tas.valeur <=0
    jeu = auHasard 3
    alert "J'ai enlevé #{jeu} objets du tas qui en contenait #{tas.valeur}"
    diminuer tas de jeu
    Si tas.valeur <= 0
        alert "j'ai gagné"
    Sinon
```

```
    alert "Il reste donc #{tas.valeur} objets; combien veux-tu en enlever: 1, 2 ou 3?"
    réponse = nouvelle Variable 0
    jusqu'à ce que 1 <=  réponse.valeur <= 3
        entrer réponse
        arrondir réponse à 1 # unité près
    diminuer tas de réponse
    Si tas.valeur <= 0
        alert "Tu as gagné"
```

Pour améliorer ce jeu on peut

- choisir une stratégie meilleure que la réponse au hasard (ligne 4)
- faire un jeu de nim à plusieurs tas (le joueur entrerait le numéro du tas et le nombre de graines à enlever de ce tas)
- passer à la version “qui perd gagne”
- passer à “soustraire un carré” où le nombre de graines à enlever est n’importe quel carré parfait
- tenter un affichage dessinant les graines plutôt que les nombres de graines :

```
auHasard = (n) -> Math.ceil Math.random()*n
dessin = (n) -> ("o" pour i dans [0...n]).join("")
tas = nouvelle Variable 21
jusqu'à ce que tas.valeur <=0
    jeu = auHasard 3
    alert "J'ai enlevé #{dessin jeu} du #{dessin tas.valeur}"
    diminuer tas de jeu
    Si tas.valeur <= 0
        alert "j'ai gagné"
    Sinon
        alert "Il reste donc #{dessin tas.valeur}; combien veux-tu enlever: 1, 2 ou 3?"
        réponse = nouvelle Variable 0
        jusqu'à ce que 1 <=  réponse.valeur <= 3
            entrer réponse
            arrondir réponse à 1 près
        diminuer tas de réponse
        Si tas.valeur <= 0
            alert "Tu as gagné"
```

Le déroulé d’une partie ressemble alors à ceci :

```
J'ai enlevé o du oooooooooooooooooooo
Il reste donc ooooooooooooooooooo; combien veux-tu enlever: 1, 2 ou 3?
3
J'ai enlevé o du ooooooooooooooooo
Il reste donc oooooooooooooooo; combien veux-tu enlever: 1, 2 ou 3?
3
J'ai enlevé o du ooooooooooooo
Il reste donc oooooooooooo; combien veux-tu enlever: 1, 2 ou 3?
```

```
3
J'ai enlevé oo du ooooooooo
Il reste donc ooooooo; combien veux-tu enlever: 1, 2 ou 3?
3
J'ai enlevé ooo du oooo
Il reste donc o; combien veux-tu enlever: 1, 2 ou 3?
1
Tu as gagné
```

Conclusion provisoire

Les exemples donnés ici montrent, on l'espère, l'omniprésence d'implicites dans les mathématiques[49] comme dans toutes les activités humaines. Cela n'est pas surprenant lorsqu'on pense à la classification des langages par Noam Chomsky : si un langage de programmation, quel qu'il soit, est infiniment plus complexe qu'un langage régulier, il est probable que les langues naturelles sont infiniment plus complexes que les langages de programmation, n'en déplaise à Leibniz et son *calculemus*. Créer un langage de programmation suffisamment proche de la langue naturelle[50] pour rendre naturelle la description d'un algorithme est donc impossible. Ce n'est pas une raison pour ne pas essayer, d'autres l'ont déjà fait, avec plus ou moins de succès. Mais il faut pour cela prendre conscience du fait qu'un programme, qu'il soit de calcul comme ici, ou de construction, évolue dans un univers restreint et que le vocabulaire utilisé peut être restreint à cet univers restreint. Un exemple qui n'a rien perdu de son actualité est Shrdlu, qui, lorsqu'on lui a demandé «Trouve un bloc plus grand que celui que tu tiens et mets-le dans la boîte», a répondu "Par « le », je suppose que vous voulez dire « le bloc plus grand que celui que je tiens »". Ce genre de question, un élève se les pose souvent mais sans nécessairement oser les prononcer à voix haute, ce qui peut expliquer bien des erreurs.

Sophus est libre, comme un logiciel libre. On a donc le droit de le mettre dans une page internet[51], de signaler les bugs[52], de l'utiliser et le faire utiliser sans restriction, de l'améliorer (de préférence en laissant tout le monde profiter de ces améliorations) … et de ne pas noter trop sévèrement les élèves qui s'en seraient inspirés le jour de l'examen. Des améliorations sont en cours, comme par exemple celle permettant de donner une valeur vectorielle à une variable[53], ou l'intégration à des logiciels comme Blockly, CaRMetal ou ekoarun. Et ses auteurs espèrent à ses utilisateurs autant de joie pour l'utiliser, qu'ils en ont eu pour le créer…

---

[49] Même dans les démonstrations, où on voit souvent des choses comme "cette étape est triviale" ou "il est évident que" ou "ce résultat est bien connu". Une définition de "bien connu" a d'ailleurs été proposée avec humour: On dit qu'un fait est bien connu lorsque d'une part l'auditoire ne le connaît pas, d'autre part on ne sait pas le démontrer...

[50] Par exemple, LSE, Linotte, Execalgo, AlgoBox, Scratch en français et depuis peu, Blockly...

[51] Cette page contient 4 copies de l'interpréteur Sophus; cette autre en contient 7...

[52] L'endroit prévu pour cela est celui-ci.

[53] Si on code l'équation 2x-5y=8 par le tableau [2,-5,8] et l'équation 3x+4y=7 par le tableau [3,4,7], on peut résoudre le système formé par ces deux équations en doublant la seconde, triplant la première puis lui soustrayant la seconde.

Bibliographie et webographie:

- Le code source de Sophus: https://github.com/AlainBusser/Sophus
- La rubrique consacrée à Sophus sur le site de l'IREM de la Réunion: http://irem.univ-reunion.fr/spip.php?rubrique173
- Un article sur la construction du nombre avec Sophus: http://revue.sesamath.net/IMG/pdf/boites_sophus.pdf
- Celui qui a donné son nom au langage: https://fr.wikipedia.org/wiki/Sophus_Lie
- Celle qui a inventé l'idée d'un "langage de programmation pour les nuls": https://fr.wikipedia.org/wiki/Grace_Hopper
- Nicolas Rouche: "Pourquoi ont-ils inventé les fractions ?": Présentation de nombres comme opérateurs.
- Stella Baruk: "L'âge du capitaine" et "Si 7=0", parle beaucoup des mots implicites dans les énoncés de mathématiques
- Stephen Kleene: "Logique mathématique": Le chapitre sur les prédicats évoque la distinction entre une variable et son nom
- Yves Martin: Manipulation directe sur tableur (concrétisation des variables numériques par des curseurs, en 2006 déjà!)
- Jean Dieudonné: article "groupes de Lie" dans *Encyclopedia Universalis*, un formidable exemple de "vulgarisation de haut niveau"
- Un outil pour analyser des phrases de langages naturels, très pratique pour *deviner* ce qui peut se passer dans la tête d'un lecteur en *regardant* ce qui se passe dans celle d'un ordinateur: http://www.nltk.org/
- Frédéric Nef: "Logique et langage, essai de sémantique intensionnelle" sur le délicat sujet de la sémantique
- Elisabeth Busser & Gilles Cohen: "L'intégrale des jeux mathématiques du Monde", une mine d'exercices ludiques