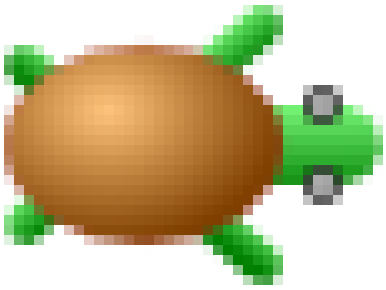
http://irem.univ-reunion.fr/spip.php?article894

La multiplication dite Â« éthiopienne Â»

- Algorithmique et programmation
- CoffeeScript
- Sofus, ou la programmation visuelle par blocs en collège (et au lycée)
-



Date de mise en ligne : mardi 19 juillet 2016

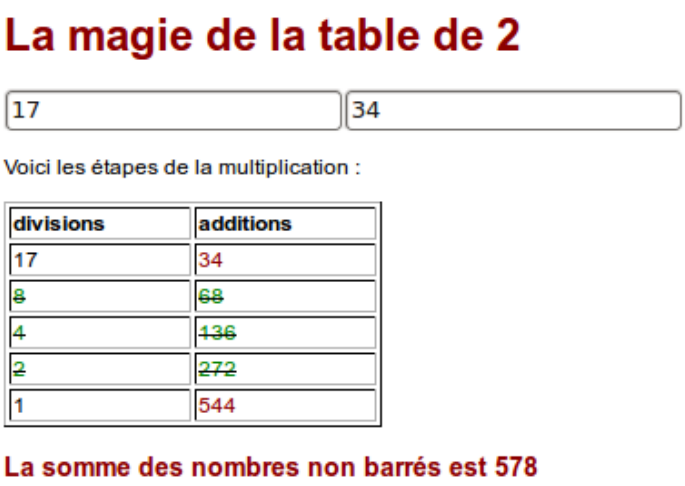
Copyright © IREM de la Réunion - Tous droits réservés

Voici un algorithme qui aura probablement du succès en cycles 3 et 4, avec l'introduction de la programmation : La multiplication des entiers telle que l'effectuaient les égyptiens de l'antiquité, et attribuée par la BBC aux éthiopiens :

Égypte ou Éthiopie ?

Cet algorithme de multiplication est attribué, dans cet article, aux égyptiens (papyrus Rhind) alors que la BBC parle de l'Éthiopie. Il n'y a pas là de contradiction puisque, selon la BBC, cette méthode est toujours utilisée en Éthiopie de nos jours alors que le papyrus Rhind faisait l'état des lieux des mathématiques de l'époque.

Voir aussi cet article qui comprend une animation interactive en html5 (annexe B) : En entrant des nombres on voit le tableau grandir ou rapetisser et certaines lignes barrées, d'autres non :



Le problème est posé dans cet article d'un site multi-langages (de programmation). Voici la description qui en est donnée :

Ethiopian multiplication is a method of multiplying integers using only addition, doubling, and halving.

Le problème qui sera traité ici est celui de la sémantique : Que signifient exactement les verbes *to double* (doubler) et *to halve* (diviser par 2) ? Aussi surprenant que cela puisse paraître, ces verbes ont des significations différentes pour les uns et les autres, comme on le voit sur les extraits de programmes ci-dessous.

Le cas du tri

On voit dans cette introduction à Python que ce langage possède deux méthodes de tri : Le tri Â« en place Â» et la fonction de tri. Le premier modifie la liste à trier :

Tri en place (remettre les nombres dans l'ordre croissant) :

L = range(8,0,-1)
L.sort()
print L <a href='http://irem.univ-reunion.fr/local/cache-code/d5ce983d474f1198ff763dcfffcfbad4.txt' style='font-family: verdana, arial, sans; font-weight: bold; font-style: normal;'>Télécharger

La liste L a alors été modifiée : Elle est rangée. Alors que la fonction crée une nouvelle liste, triée, sans modifier la liste d'origine :

L = range(8,0,-1)
print sorted(L)
print L <a href='http://irem.univ-reunion.fr/local/cache-code/9c602bda2405bd77ac9dd72f10fb96b6.txt' style='font-family: verdana, arial, sans; font-weight: bold; font-style: normal;'>Télécharger

Là la liste est restée telle quelle. Une nouvelle liste ayant été créée, cette méthode de tri utilise plus de mémoire machine. Elle est aussi plus lente, comme on peut le vérifier avec le code ci-dessous :

python -m timeit 'L=range(1000,0,-1); L.sort()'
python -m timeit 'L=range(1000,0,-1); L=sorted(L)' <a href='http://irem.univ-reunion.fr/local/cache-code/b5f96a90384883f52b9c73609fb4dd31.txt' style='font-family: verdana, arial, sans; font-weight: bold; font-style: normal;'>Télécharger

qui annonce que la première méthode met, en moyenne, 46,3 µs alors que la seconde est légèrement plus lente avec un temps moyen de 53,8 µs.

On peut dire aussi que le tri en place est un tri vraiment effectué alors que la fonction de tri est une simulation de tri. Or lorsqu'on demande à un enfant de ranger sa chambre, ce n'est pas pour obtenir une simulation de rangement !

sémantique des verbes

Lorsqu'on va voir son patron pour demander une augmentation, on n'attend pas seulement de lui le calcul du nouveau salaire, mais bel et bien une modification du salaire Â« en place Â». De la même façon, selon wikipedia, « l'incrémentation est l'opération qui consiste à ajouter 1 (et par exension une valeur entière fixée) à un compteur ». Ce qui ne signifie pas Â« effectuer une addition Â» mais Â« modifier le compteur Â». D'ailleurs, la fonction qui, à un entier n, associe n+1, ne porte pas le même nom puisqu'elle s'appelle Â« successeur Â» depuis les axiomes de Peano. Et si on utilise deux verbes différents, c'est *a priori* pour décrire deux concepts différents...

Pour la multiplication éthiopienne, on a besoin de doublement. Or Â« doubler Â» signifie « Rendre double ; mettre le double, augmenter d'une fois autant, multiplier par deux ». Là encore, si je dis que la bière a doublé mon tour de

taille, les centimètres de graisse ne sont pas virtuels : Mon tour de taille a vraiment été modifié !

Pour la division par 2, la situation linguistique est plus compliquée : En français, il ne semble pas exister de verbe décrivant cette action comme l'anglais Â« to halve Â» ; mais suite à l'influence anglaise, la division par deux s'appelle parfois décimation par extension de la définition initiale du verbe *décimer* : « Supprimer un dixième de quelque chose » : Diviser par 2 c'est supprimer la moitié (donc plus que le dixième), ou réduire de 50%. Là encore, lorsque les romains décimaient leurs déserteurs, c'était par exécution de 10% d'entre eux et l'effectif diminuait vraiment, pas seulement par un calcul d'estimation de sa nouvelle valeur...

Double(r)

Sur le site rosettacode, la plupart des solutions utilisent une fonction qui calcule le double d'un entier, au lieu d'une instruction qui double un entier. Pourtant, dans le cas de la division par 2, le verbe *to halve* n'est pas confondable avec la fonction *half*. En français, il s'agirait de confondre Â« doubler Â» avec Â« le double de Â» et Â« diviser par 2 Â» avec le quotient Â« la moitié de Â». Par exemple la première solution donnée en Python

```
def halve(x):
 return x // 2

def double(x):
 return x * 2<div class='code_download'   style='text-align: right;'> <a
href='http://irem.univ-reunion.fr/local/cache-code/c263884e84f1b6b378a7a4fd6b9baa4d.txt'   style='font-family:
verdana, arial, sans; font-weight: bold; font-style: normal;'>Télécharger
```

qui oblige à remplacer le multiplieur et le multiplicande par leur moitié (respectivement double) :

```
 multiplier   = halve(multiplier)
 multiplicand = double(multiplicand)<div class='code_download'   style='text-align: right;'> <a
href='http://irem.univ-reunion.fr/local/cache-code/a0ae75f1b06cc6ad2f7984b37f3623de.txt'   style='font-family:
verdana, arial, sans; font-weight: bold; font-style: normal;'>Télécharger
```

Python

En fait, en Python, l'instruction Â« doubler x Â» ne peut pas être définie comme une fonction, parce que lorsqu'on fait appel à une fonction f(x), la variable x est locale, c'est-à-dire qu'elle n'existe et ne peut être modifiée que dans le corps de la fonction. Par exemple avec ce script :

```
def doubler(x):
 x += x
 return x
a = 3
doubler(a)
```

Copyright © IREM de la Réunion

print(a) <a href='http://irem.univ-reunion.fr/local/cache-code/a082603552580f8967740e4740abfb23.txt' style='font-family: verdana, arial, sans; font-weight: bold; font-style: normal;'>Télécharger

Les trois lignes de programme sont en bas : La première a pour effet que a soit égal à 3, et la dernière affiche 3 puisqu'elle affiche a et que a=3. Donc la seconde ligne n'a pas eu d'effet. Pourquoi ? Parce que doubler(a) a eu pour effet d'appeler la fonction f avec 3 dans x. Successivement, il s'est passé ceci :

- la variable x a été créée ;
- Le contenu de a (c'est-à-dire 3) a été copié dans x (à ce stade, x=3) ;
- Le contenu de x (pour l'instant, 3) a été rajouté (+=) à x (Â« += Â» voulant dire 'augmenter de'', x vaut alors 3+3=6) ;
- Le nombre obtenu (6) est retourné ;
- La variable x ne servant plus à rien est jetée dans la poubelle (Â« ramasse-miettes Â»)

Alors a est toujours égal à 3 et x n'existe plus. Bref, raté ! En fait, le problème est que Python est un langage trop évolué et ne sait pas (du moins, simplement) modifier les variables directement dans la mémoire de la machine [1]. Il faut donc, ou bien des langages possédant déjà des instructions du genre Â« augmenter de Â» ou Â« doubler Â» (exemples typiques : Cobol et Perl 6), ou bien des langages proches de la machine (exemples : C ou l'assembleur). Voici des exemples de ces langages, décortiqués (onglets suivants) :

Cobol

```
 PROGRAM-ID. halve.
 DIVIDE n BY 2 GIVING m END-DIVIDE
 PROGRAM-ID. twice.
 MULTIPLY n by 2 GIVING m END-MULTIPLY
...
 IF odd
 ADD r TO product GIVING product END-ADD
 END-IF
 CALL "halve" USING
 BY CONTENT l,
 BY REFERENCE l
 END-CALL
 CALL "twice" USING
 BY CONTENT r,
 BY REFERENCE r
 END-CALL<div class='code_download'  style='text-align: right;'> <a href='http://irem.univ-reunion.fr/local/cache-code/392660f6c368945e5d34bc63a2ccdbeb.txt'  style='font-family: verdana, arial, sans; font-weight: bold; font-style: normal;'>Télécharger
```

Verbeux mais intéressant (on est assez proche de l'anglais naturel, et Cobol date de l'époque où on imaginait qu'un jour on programmerait des ordinateurs à la voix, permettant des choses comme ça).

En bref, Cobol permettant de multiplier ou diviser par 2, on définit des mots Â« halve Â» et Â« twice Â» à partir de ces instructions. Cobol tombant quelque peu en désuétude, il ne semble pas y avoir eu d'équivalent dans les

langages plus modernes, à part les exemples des onglets suivants (et Sofus, pour la version française).

Perl 6

En plus on a la concision puisque voici l'intégralité du code :

```
sub halve (Int $n is rw) { $n div= 2 }
sub double (Int $n is rw) { $n *= 2 }
sub even (Int $n --> Bool) { $n %% 2 }

sub ethiopic-mult (Int $a is copy, Int $b is copy --> Int) {
 my Int $r = 0;
 while $a {
 even $a or $r += $b;
 halve $a;
 double $b;
 }
 return $r;
}<div class='code_download'   style='text-align: right;'> <a
href='http://irem.univ-reunion.fr/local/cache-code/0ad236305f643fe75e149ffcab2ec203.txt'   style='font-family:
verdana, arial, sans; font-weight: bold; font-style: normal;'>Télécharger
```

Le symbole Â« dollar Â» désigne Â« la valeur de la variable Â», ce qui fait que *$n* est le contenu de la variable n ; alors *$n div= 2* veut dire que le contenu de n est divisé par 2. Noter que cette abréviation, qui vient de bash, se retrouve aussi dans le très populaire (du moins chez les élèves) php.

Le mot Â« my Â» remplace le Â« local Â» des autres langages : La variable r initialisée à 0 est intitulée Â« my r Â» parce que c'est Â« mon r à moi tout seul Â» : C'est une bonne façon de traduire la notion de variable locale. On remarque que l'idée a été reprise dans Scratch, chaque lutin pouvant avoir ses propres variables.

On remarque que le *si..alors* est développé comme la forme disjonctive d'une implication : Ou bien le contenu de a est pair, ou alors on ajoute le contenu de b à celui de r.

Forth

Forth fonctionne avec une pile, du coup la syntaxe du doublement est très concise : *2/*, qui a pour effet de placer 2 sur la pile puis de le remplacer par son produit avec le précédent sur la pile. L'algorithme est donc extrèmement court (tout l'algorithme est présent ici, il n'en manque pas un octet !), comme c'est souvent le cas avec Forth :

```
: even? ( n -- ? ) 1 and 0= ;
```

Copyright © IREM de la Réunion

```
: e* ( x y -- x*y )
 dup 0= if nip exit then
 over 2* over 2/ recurse
 swap even? if nip else + then ;<div class='code_download'  style='text-align: right;'> <a
href='http://irem.univ-reunion.fr/local/cache-code/f537ba9d09cfe18aba9ad5158c4553cd.txt'  style='font-family:
verdana, arial, sans; font-weight: bold; font-style: normal;'>Télécharger
```

Détaillons la première ligne : On place n sur la pile, puis 1, et on effectue un Â« and Â» (qui va remplacer les deux derniers nombres de la pile par leur conjonction bit par bit). Le haut de la pile contient alors n modulo 2 (parce que, en binaire, un Â« and Â» avec 1 donne exactement n modulo 2). Ensuite on effectue un test d'égalité avec 0 et on met le résultat en haut de la pile (c'est comme ça que Forth récupère les données, par lecture du haut de la pile).

L'algorithme (fonction Â« e Â») utilise aussi la pile, avec

- dup qui recopie le nombre tout en haut de la pile, au-dessus de lui -même (ce qui constitue une duplication) ;
- swap qui échange le dernier nombre avec l'avant-dernier dans la pile ;
- over qui place une copie de l'avant-dernier nombre de la pile par dessus le dernier (il fait donc comme dup mais recopie l'avant-dernier nombre au lieu du dernier)
- et nip, qui supprime l'avant-dernier nombre de la pile.

Qui dit Â« pile Â», dit notation polonaise inversée, donc a priori, utiliser Forth (langage) (qui a été créé pour de l'électronique embarquée, en l'occurence du pilotage de téléscope)) au collège, n'est peut-être pas une bonne idée (euphémisme)...

Fortran

C'est probablement à cause de son âge que Fortran n'est pas muni des barrières qui empêchent de toucher à ses variables :

```
 subroutine halve(v)
 integer, intent(inout) :: v
 v = int(v / 2)
 end subroutine halve

 subroutine doublit(v)
 integer, intent(inout) :: v
 v = v * 2
 end subroutine doublit
...
 call halve(plier)
 call doublit(plicand)<div class='code_download'  style='text-align: right;'> <a
href='http://irem.univ-reunion.fr/local/cache-code/ad0b8df8037fdf76a3775e0e06446a6d.txt'  style='font-family:
verdana, arial, sans; font-weight: bold; font-style: normal;'>Télécharger
```

Par contre le doublement est obtenu avec *v = v * 2* qui a laissé perplexe pas mal de monde depuis la création du langage vers le milieu des années 1950 (c'est un peu en réaction par rapport à cette notation qu'ont été créés, à la

fin des années 1950, Algol et Cobol)...

C

pour modifier une variable *in situ*, le langage C utilise un pointeur (programmation), représenté par une étoile après le nom de la variable. Pour multiplier un nombre par 10, en décimal, il suffit de le décaler d'un chiffre vers la gauche et insérer un 0 à droite. En binaire, cette opération a pour effet de doubler le nombre. Le décalage est noté *>>1* si on décale d'un seul chiffre binaire, et le décalage en place (le doublement, quoi) se note *>>=1* :

```
void halve(int *x) { *x >>= 1; }
void doublit(int *x) { *x <<= 1; }
...
 halve(&plier); doublit(&plicand);<div class='code_download'   style='text-align: right;'> <a
href='http://irem.univ-reunion.fr/local/cache-code/f61d24828b30a6f299ee140b9eb1a4af.txt'   style='font-family:
verdana, arial, sans; font-weight: bold; font-style: normal;'>Télécharger
```

Par contre C n'est ni concis ni naturel, et même si on trouve régulièrement des programmeurs en C qui arrivent en Seconde avec une certaine maîtrise du langage (et qui donc l'ont (auto-)appris au collège), ils ne sont pas majoritaires en nombre...

bash

Le langage bash est, lui aussi, proche de la machine, ce qui là aussi permet de définir des instructions (ou Â« commandes Â») de doublement etc ; là aussi le code est concis, et reprend certaines des idées précédentes :

```
halve() {
 (( $1 >>= 1 ))
}

double() {
 (( $1 <<= 1 ))
}

is_even() {
 (( ($1 & 1) == 0 ))
}

multiply() {
 local plier=$1
 local plicand=$2
 local result=0
```

```
 while (( plier > 0 ))
 do
 is_even plier || (( result += plicand ))
 halve plier
 double plicand
 done
 echo $result
}<div class='code_download'   style='text-align: right;'> <a
href='http://irem.univ-reunion.fr/local/cache-code/128788174523c25d2b02a55db75697a1.txt'   style='font-family:
verdana, arial, sans; font-weight: bold; font-style: normal;'>Télécharger
```

En bash, le symbole *$1* représente le premier argument (ou le seul s'il n'y en a qu'un). Donc doubler *$1* c'est le décaler d'un chiffre binaire vers la gauche (en ajoutant automatiquement un 0 comme nouveau dernier chiffre). Le diviser (Â« halve Â») par deux c'est le décaler dans l'autre sens (vers la droite, en perdant l'ancien dernier chiffre).

Pour le test de parité, un Â« et Â» est effectué, chiffre par chiffre, entre le nombre à tester et le nombre 1 : Tous les chiifres binaires de 1, sauf le dernier, sont nuls, et 0 est élément absorbant pour la conjonction. Donc *$1&1* ne laisse que le dernier chiffre de *$1*, soit 0 ou 1 selon que *$1* est pair ou impair.

On remarque que l'évaluation paresseuse des disjonctions est utilisée ici : si le multiplieur (Â« plier Â») est pair alors la disjonction est vraie et il n'est plus nécessaire d'évaluer son second terme. Mais si le multiplieur est impair alors la disjonction a la même valeur de vérité que son second terme qui est alors calculé : Bref, on augmente Â« result Â» de Â« plicand Â» seulement lorsque Â« plier Â» est impair. Voir à l'onglet Â« perl 6 Â» pour une façon plus naturelle d'exprimer ce principe.

assembleur

Même principe : Pour doubler, on décale (Â« shift to the left Â» abrégé en *shl*) d'un bit vers la gauche, et pour diviser par 2, on décale vers la droite (Â« shift to the right Â» abrégé en *shr*). On a choisi de faire ces opérations sur le registre ebx :

```
halve
 shr ebx, 1
 ret

double
 shl ebx, 1
 ret<div class='code_download'   style='text-align: right;'> <a
href='http://irem.univ-reunion.fr/local/cache-code/72ac9997bce235bbbc2fb01d5cef7c6f.txt'   style='font-family:
verdana, arial, sans; font-weight: bold; font-style: normal;'>Télécharger
```

Le nom de Â« EBX Â» désigne la version étendue (Â« E Â» ; parce que 32 bits c'est plus étendu que 16 bits) du registre Â« BX Â» du 8086 ; Â« BX Â» parce que c'est le registre de base (Â« B Â»). Les registres principaux du processeur 80386 sont les suivants :

- E**A**X est l'**a**ccumulateur qui est fait pour des choses comme Â« augmenter de Â» : Il écrase son ancien contenu

par la somme (on s'en sert surtout pour additionner beaucoup de nombres, ce qui fait qu'il accumule les nombres en sommes partielles, d'où son nom) ;
- E**B**X est donc le registre de **b**ase, celui dont on se sert quand on n'en utilise pas d'autres ;
- E**C**X est le registre **c**ompteur, il a un fonctionnement similaire à celui de l'accumulateur mais spécialisé dans les incrémentations constantes (de 1 si on utilise les octets, d'une puissance de 2 dans le cas général) ;
- E**D**X est le registre de **d**onnées qui est utilisé pour les échanges avec la mémoire de l'ordinateur.

On a vu plus haut que Cobol permet de doubler (avec Â« multiply n by 2 Â») et de diviser par 2 (avec Â« divide n by 2 Â»). Sofus fait en quelque sort mieux, puisque le verbe Â« doubler Â» fait partie du vocabulaire de base de Sofus. Voici donc la solution du problème de rosettacode avec Sofus :

[http://irem.univ-reunion.fr/local/cache-vignettes/L400xH389/ethiop1-eccdc.png]

C'est si beau qu'on se prend à rêver que le très obligatoire Scratch (langage) comprenne aussi des instructions comme Â« doubler Â» ou Â« diviser par 2 Â». Cela semble tout-à-fait possible, puisque Scratch contient déjà un héritage de Cobol : L'instruction Â« incrémenter de Â», dont la fantastique (mais vraie) histoire est narrée [2] ci-dessous :

Scratch 1.4

Au commencement était Mitch Resnick, qui, au milieu d'un chaos cobolien, conçut l'idée de programmer par blocs. Il programma Scratch pendant des mois, avec son équipe du MIT, puis, fatigué par les efforts de programmation (en Smalltalk tout de même), par distraction, il commit l'impensable :

Il ajouta un bloc à la Cobol dans Scratch

Si si, ce bloc, le voici dans toute la splendeur de Scratch 1.4 en anglais (Â« incrémenter Â» ayant été traduit par Â« change Â» soit Â« modifier Â») :

[http://irem.univ-reunion.fr/local/cache-vignettes/L124xH32/incenglish-c3ae2.png]

La traduction littérale étant Â« modifier n de 1 Â», on a bien là une instruction de modification sofusienne *in situ*...

La fatigue de Mitch Resnick lui a fait oublier, soit de créer d'autres instructions similaires (Â« multiplier par Â» etc) soit d'enlever cette étrange instruction étrangère. Mais les traductions de Â« change by Â» sont un peu surprenantes (voir les autres onglets) :

En français

Dans la traduction française de Scratch 1.4, le bloc sofusien vu précédemment s'est traduit ainsi :

[http://irem.univ-reunion.fr/local/cache-vignettes/L133xH34/incfrench-262a7.png]

Il s'agit d'une traduction littérale :

- Â« change Â» est devenu Â« changer Â» (Â« modifier Â» ou Â« incrémenter Â» eût été plus judicieux)
- Â« by Â» est devenu Â« par Â» au lieu de Â« de Â»...

Mais le pire reste à venir (onglets suivants, sauf le prochain qui est une pause)...

Au Québec

C'est au Québec, probablement sous l'influence de Pierre Couillard, qu'une nouvelle traduction, prenant mieux en compte ce bloc d'incrémentation, est venue : Dans Scratch 1.4 elle apparaît dans le menu des langues, juste sous le français Â« officiel Â». Son nom étant Â« français (Canada) Â». Et là, heureuse surprise, on comprend beaucoup mieux ce que fait cette incrémentation sofusienne :

[http://irem.univ-reunion.fr/local/cache-vignettes/L114xH33/incquebec-506c3.png]

Alors là, on dirait vraiment du Cobol ! Dans un monde idéal, l'équipe de Scratch aurait tenu compte de la sagesse des québécois et aurait remplacé son bloc sofusien par Â« to n add 1 Â». Et dans ce monde scratchien idéal, l'équipe responsable de la traduction Â« français officiel Â» aurait alors suivi, voire précédé (en français ça devait être facile) la traduction québécoise. Mais le monde de Scratch est loin d'être idéal, sinon la version 2.0 n'aurait jamais été programmée en Flash [3]...

En créole

La version créole (haïtien) de Scratch est encore plus mystérieuse :

[http://irem.univ-reunion.fr/local/cache-vignettes/L145xH32/inccreole-9e380.png]

Â« mettre n en place de 1 Â»...

Certes Haïti c'est le pays du vaudou, mais là, on fait mieux que l'affectation traditionnelle qui met 1 à la place de n, on remplace carrément des constantes par des variables !

Il est difficile de deviner de quelle langue s'est inspirée la version créole mais si c'est la version québécoise il y a là un phénomène linguistique particulièrement intéressant !

Copyright © IREM de la Réunion

Scratch 2.0

De mieux en mieux, la version de Scratch qui sera utilisée dans tous les collèges de France et de Navarre dans quelques semaines, dit ceci :

[http://irem.univ-reunion.fr/local/cache-vignettes/L154xH33/incscratch-fa581.png]

Franchement, il n'est pas illusoire de penser qu'au cours de la simulation d'un programme de calcul, un collégien fasse quelque chose comme ceci :

[http://irem.univ-reunion.fr/local/cache-vignettes/L171xH88/delireig1-878ba.png]

Il n'est pas illusoire non plus de penser que ledit collégien ait du mal à comprendre que n soit égal à 81 et non à 1 (Â« mais puisque j'ai remplacé n par 1, n devrait être devenu 1 et non 81, n'est-pas M'sieur ? Â»). Il n'est pas illusoire de penser que le prof de maths qui ne s'attendait pas à ce coup-là soit lui aussi surpris au point qu'il ait du mal à expliquer le comportement illogique du logiciel.

Comment après ces approximations syntaxiques, peut-on exiger de la rigueur d'adolescents qui n'en voient même pas dans le logiciel qu'on leur a imposé (ainsi qu'à leur prof) ?

Snap !

Snap ! est plus cohérent dans sa version française :

[http://irem.univ-reunion.fr/local/cache-vignettes/L128xH33/incsnap-4a930.png]

Sauf que des mots l'ordre pas très logique n'est [4] : Â« Ajouter 1 à n Â» est plus facile à comprendre que Â« Ajouter à n 1 Â», surtout s'il n'y a pas de virgule entre Â« n Â» et Â« 1 Â».

Il est à remarquer que bien qu'il soit possible d'effectuer une opération *in situ* dans Snap ! comme le montre ce bloc, là non plus aucun effort n'a été fait jusque là pour généraliser ce genre de bloc, et Snap ! ne possède pas plus de Â« doubler Â» que son ancêtre Scratch [5].

Blockly

Blockly lui aussi s'inspire fortement de Scratch et lui aussi est développé en JavaScript. Voici comment sa tortue traduit le fameux bloc sofusien d'incrémentation *en place* :

Copyright © IREM de la Réunion

[http://irem.univ-reunion.fr/local/cache-vignettes/L280xH39/incblockly-720b9.png]

Pas mal, à part le côté pléonasme (le plus souvent quand on incrémente c'est de 1). Il est dommage que le bloc Â« décrémenter de 1 Â» n'ait pas (encore ?) été ajouté à Blockly. Mais les techniques exposées ici permettent de fabriquer ses propres blocs et c'est la raison pour laquelle Sofus a été programmé dans Blockly et non dans Scratch : Dans le premier cas c'est possible, dans l'autre non...

Comme souvent en algorithmique, l'usage du tableur contribue à repérer les valeurs successives des variables, tout simplement en les écrivant l'une en-dessous de l'autre dans une colonne (une colonne par variable) du tableur. Voici le rendu en 4 colonnes obtenu avec le tableur gnumeric :

[http://irem.univ-reunion.fr/local/cache-vignettes/L325xH284/etableur1-5911b.png]

Ce fichier a été obtenu de la manière suivante :

- Les nombres 17 et 34 à multiplier ont été placés en A1 et en C1 respectivement.
- Le reste de la division de 17 par 2 a été placé en B1, avec la formule *=mod(A1 ;2)*
- Le produit de C1 et B1 a été placé en D1 avec la formule *=B1*C1* (comme ça si le nombre en A est pair, la multiplication donne 0 et on n'a pas besoin, à la fin, de test sur la parité ; c'est de l'algèbre de Boole) ;
- Dans la cellule A2 a été entrée la formule *=(A1-B1)/2* qui effectue la division par 2 (après avoir soustrait le nombre 1 si A1 est impair, ce qui garantit que le résultat de la division est entier) ;
- Dans la cellule C2 a été entré le doublement avec *=C1*2*
- les contenus des cellules B1 et D1 ont été copiés vers le bas en B2 et D2 ;
- Puis toute la ligne 2 a été recopiée vers le bas, 10 fois.
- Enfin, dans la cellule D15 (choisie au hasard) a été entrée la formule *=sum(D1:D12)* qui donne le produit des nombres entrés en A1 et C1.

Voici le fichier obtenu (exporté en odt pour ne pas nécessiter d'installer un tableur libre, léger, gratuit et très adapté à la statistique inférentielle, merci qui ?), dans lequel il suffit de modifier les contenus de A1 et C1 pour calculer d'autres produits :
<a href="http://irem.univ-reunion.fr/IMG/ods/ethiopian.ods" title='OpenDocument Spreadsheet - 5.9 ko'
type="application/vnd.oasis.opendocument.spreadsheet">[OpenDocument Spreadsheet - 5.9 ko]

Outre le fait qu'avec ce fichier tableur, on a Â« privilégié le changement de cadre Â», la version tableur permet de poser une question qui n'avait pas de sens avec la version Â« algo Â» :

En allant jusqu'à la ligne 12 comme ci-dessus, quel est le plus grand nombre qu'on peut mettre dans A1 pour que le produit dans D15 soit correct ?

On remarque en passant que la question ne se pose pas pour le second facteur, et même, que cet algorithme ne montre pas que la multiplication est commutative.

Avec Mathem@algo

Le site mathem@algo est basé sur le changement de registre, on peut donc s'en servir pour aller

- de l'algorithme au tableur ;
- du tableur au calcul formel

ceci, de manière presque automatique, et avec à la clé la possibilité de démontrer que cet algorithme calcule bien un produit (en fait, avec les tests successifs sur la parité du premier facteur, on peut juste prouver que, ce facteur étant fixé, la fonction qui, au second facteur, associe le produit, est linéaire ; c'est déjà ça).

Une fois rendu sur le site mathem@algo, on clique hardiment sur l'onglet Â« Blockly/Xcas Â» situé tout à gauche, et là, on choisit l'extension tableur Xcas (ou on clique sur le lien précédent) puis on programme l'algorithme donné ci-dessus (version Sofus) en renommant les variables *_A*, *_B* et *_C* (la raison de ces noms bizarres est donnée ci-dessous). Si on manque de hardiesse, on peut aussi charger directement le programme dans mathem@algo : <a href="http://irem.univ-reunion.fr/IMG/zip/ethableur.bly.zip" title='Zip - 758 octets' type="application/zip">[Zip - 758 octets]

Le programme ressemble à ceci :

[http://irem.univ-reunion.fr/local/cache-vignettes/L328xH329/raffinat1-8f73a.png]

En le lançant (par clic sur le bouton), on crée un affichage donnant le produit, mais également, le remplissage du tableur se fait :

[http://irem.univ-reunion.fr/local/cache-vignettes/L366xH222/raffinat2-90114.png]

C'est pour ça que les variables devaient s'appeler *_A* etc : Les noms des colonnes du tableur Xcas sont A, B, C etc, et la variable *_B* s'affiche étape par étape dans la colonne B. Et ce sont bien des formules qui sont entrées dans le tableur !

Mais une toute petite modification ouvre une toute autre perspective : Remlacer l'entrée numérique de *_B* par un nom de variable, par exemple Â« x Â» :

[http://irem.univ-reunion.fr/local/cache-vignettes/L330xH325/raffinat3-b2317.png]

En fait on a aussi remplacé l'entrée de la première ligne par 100, parce que 17 ne donnait que deux lignes à additionner. Et en exécutant ce nouveau programme, on voit que, quelle que soit l'entrée choisie pour B (et pas seulement un nombre entier), on a bien son centuple à la fin, comme le montre la boîte de dialogue modal obtenue :

[http://irem.univ-reunion.fr/local/cache-vignettes/L400xH167/raffinat4-3037e.png]

Il en dit des choses intéressantes ce *raffinat.perso* ! Merci raffinat.perso ! L'expression est même simplifiée par Xcas. Cela prouve algébriquement que, si le premier facteur est 100, l'algorithme calcule le centuple du second facteur. On peut répéter à l'envi ce genre de preuve, par exemple si le premier facteur était 17, on aurait Â« 17*x Â» à la sortie [<a href='#nb6' class='spip_note' rel='footnote' title='Ceci suggère une preuve du programme : Montrer par récurrence que si le (...)' id='nh6'>6].

Le tableur, quant à lui, donne des éléments de preuve qui simplifie sa rédaction dans ce cas précis (en additionnant les expressions de la deuxième colonne correspondant à des nombres impairs de la première colonne et en mettant Â« x Â» en facteur dans le résultat ; les étapes intermédiaires du calcul de la somme se trouvant dans la troisième

Copyright © IREM de la Réunion

colonne, le calcul formel à effectuer est donc 4x+32x+64x=100x) :

[http://irem.univ-reunion.fr/local/cache-vignettes/L371xH283/raffinat5-3cb8f.png]

Pour finir, le logiciel why3 sert à prouver qu'un programme fait bien ce qu'il est censé faire. Voici la preuve par ce logiciel que l'algorithme Â« éthiopien Â» effectue bien une multiplication : http://toccata.lri.fr/gallery/binar...

---

[1] Il s'agit là du paradigme de la programmation fonctionnelle, qui « considère le calcul en tant qu'évaluation de fonctions mathématiques et n'admet ni changement d'état ni modification des données » selon Wikipedia. La programmation fonctionnelle fait grand usage de la notion d'objet de première classe et un tel objet doit « pouvoir être passé comme paramètre à une procédure ou une fonction ». Or il est dit, toujours sur wikipedia, d'une fonction informatique, que lorsqu'elle « possède des paramètres d'entrée, elle en prend dans les implémentations actuelles une copie, au lieu de travailler sur les véritables variables ». Or le doublement est une procédure (informatique), c'est-à-dire une fonction ne renvoyant pas de valeur pertinente. On n'en utilise que son effet de bord (informatique) et du coup, le doublement est incompatible avec la notion de fonction pure qui prévaut en programmation fonctionnelle. Il est intéressant à ce propos de constater que Lisp qui fut le premier langage de programmation fonctionnelle, est né en même temps que Cobol.

[2] Suis-je plutôt d'humeur marrante, narrante ou navrante ? Aux lecteurs d'en juger...

[3] Rappelons que lorsque Jens Mönig a proposé à l'équipe du MIT sa version de BYOB en JavaScript, il s'est trouvé face à un mur...

[4] Par là passer Yoda a dû

[5] La discussion est lisible ici. On y voit que la création de blocs avec effets de bords est possible mais pas facile ni souhaitée : L'expérimentation de Snap en écoles primaires semble être au contraire axée sur la programmation purement fonctionnelle.

[6] Ceci suggère une preuve du programme :

- Montrer par récurrence que si le premier facteur est une puissance de 2, alors le programme calcule son produit par le second facteur ;
- Généraliser en utilisant la distributivité de la multiplication par rapport à l'addition.

La seconde étape peut être abordée en collège, la première non, mais on peut essayer avec plusieurs puissances de 2 avec mathem@algo et faire, osons le dire, une induction (logique). Ahem...

Copyright © IREM de la Réunion