InforMATHique

DUT INFO / 1ère année / 2014 - 2015

Licence Creative Commons

Mis à jour le 28 septembre 2014 à 16:25

Math et Info : une histoire d'amour

Guillaume CONNAN

IUT Nantes Informatique

UNIVERSITÉ DE NANTES

TABLE DES MATIÈRES

1 Infor-MATH-ique **5**
1.1 Ce module 6
1.2 La faiblesse de la force brute 6
1.3 « J'ai toujours été nul (ou bon) en maths » 7
1.4 L'art abstrait 8
1.5 Insecticide 9
1.6 Langage 10
1.7 ...les maths c'est quand même un peu dur 10
1.8 RECHERCHES 11

2 Compter sur Haskell **15**
2.1 Collège 16
2.2 Opérateurs 19
2.3 Notation polonaise 21
2.4 RECHERCHES 24

3 Let's talk about Sets **27**
3.1 C'est quoi ce Type ? 28
3.2 Notion de cardinal 37
3.3 RECHERCHES 38

4 Let's have some fun **41**
4.1 C'est quoi une fonction ? 42
4.2 Une fonction c'est more fun 43
4.3 En résumé, comment construire nos fonctions ? 47
4.4 Fonction récursives : première approche 48
4.5 Une première approche des listes sur Haskell 51
4.6 Application, composition et priorité des fonctions 55
4.7 Raisonnement par récurrence 58
4.8 Prouver que deux fonctions sont équivalentes 64
4.9 RECHERCHES 65

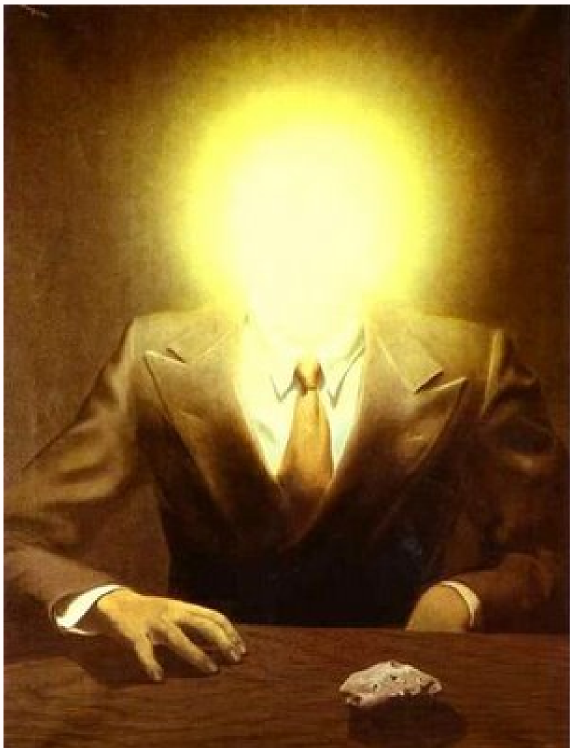
A Raccourcis emacs **77**

B Mon livre de CM1 **81**

Lectures recommandées pour aller plus loin **97**

THÈME 1

Infor-MATH-ique



Qu'allons-nous faire cette année ?

1 Ce module

Oubliez tout ce que vous avez fait jusqu'à maintenant ! Notamment dans votre manière de travailler...

Le poly que vous avez entre les mains n'est pas un résumé de cours à lire la veille d'un DS. Les exercices ne sont pas des séries de questions auxquelles on répond par oui ou non en pensant à autre chose !

AVANT LES AMPHIS, vous pourrez lire le poly pour bien suivre la séance, préparer des questions. Pendant l'amphi, d'autres notions pourront être abordées, des points pourront être développés selon vos questions : il faut être actif. Le poly est bavard, fait des digressions : c'est UN support mais pas LE support qui sera celui que vous fabriquerez vous-même.

AVANT LES TD, il faudra avoir étudié l'amphi et les TD précédents. Les exercices proposés ne sont pas fermés : ils peuvent évoluer pendant la séance, au gré de vos propositions, questions. Beaucoup de choses seront dites par vous ou moi : ces séances sont autant du cours que les amphis.

Il faudra régulièrement compléter une petite FICHE DE SYNTHÈSE dont vous pourrez disposer pendant les évaluations (ainsi que le poly).

Concernant l'ÉVALUATION : le contrôle est continu. Seront évalués :

— votre investissement en classe, vos propositions, vos questions ;

— des travaux effectués en dehors des cours : mini-projet, exercice de recherche ;

— des petits tests rapides (1/4 d'heure) en amphi sans être forcément prévenus ;

— des évaluations en TD, prévues ou non ;

— Un ou deux devoirs plus longs en amphi et prévus à l'avance.

2 La faiblesse de la force brute

Vous voulez organiser un tournoi de sumo à l'IUT pour financer votre sortie de fin d'année à Klow.

Imaginons que vous soyez 16 participants et qu'à l'issue de chaque combat un(e) étudiant(e) est qualifié(e) pour le tour suivant et l'autre éliminé(e). Tous les participant(e)s doivent combattre au moins une fois. À la fin, bien sûr, il y aura un(e) unique gagnant(e).

Combien de combats vont avoir lieu ?

Bon, 16 participants donc 8 combats. Ensuite il ne reste plus que 8 étudiant(e)s. Ça fait 4 nouveaux combats. Puis 2 combats puis un combat. Je sors ma calculatrice : $8+4+2+1=15$ combats.

Facile. En plus, on a quand même choisi 16 participant(e)s donc ça facilite les choses : on peut diviser par 2 encore et encore.

Ça peut se compliquer si on part de 7 participants par exemple. Au premier tour, un(e) étudiant(e) devra rester sur le banc puis combattre ensuite un(e) des vainqueur(e)s. On y arrive quand même : 3 combats et un(e) sur le banc. Ça fait 3 qualifié(e)s + 1 donc 2 combats puis finalement un dernier combat. Je sors encore ma calculatrice : $3+2+1=6$

Je vous laisse voir ce que ça donne pour 555 étudiants...(*pause de cinq minutes*). Ça y est ?

Oui effectivement, c'est bien ça : 554 combats.

Tiens, tiens...16 étudiant(e)s → 15 combats, 7 étudiant(e)s → 6 combats, 555 étudiant(e)s → 554 combats. Ah ah...

Ben c'est bizarre : il y a un combat de moins que de participant(e)s. Je me suis trompé dans mes calculs ? Je recommence (je fais des tests...). Toujours pareil. Je fais d'autres tests avec d'autres valeurs....pareil.

Info

Un peu de jargon informatique.

> **Brute force** */adj./*
> *Describes a primitive programming style, one in which the programmer relies on the computer's processing power instead of using his or her own intelligence to simplify the problem, often ignoring problems of scale and applying naive methods suited to small problems directly to large ones.*
> *Ken Thompson, co-inventor of Unix, is reported to have uttered the epigram "When in doubt, use brute force".*

in « The New Hacker's Dictionnary »
http://outpost9.com/reference/jargon/jargon_17.html#TAG214

Nous venons donc d'utiliser la force brute pour calculer le nombre de combats. Y aurait-il un autre moyen ?

> *Testing shows the presence, not the absence of bugs.*

Edsger W. *Dijkstra* (1969) in J.N. Buxton and B. Randell, eds, Software Engineering Techniques, April 1970, p. 16. Report on a conference sponsored by the NATO Science Committee, Rome, Italy, 27–31 October 1969.

Vous allez bientôt étudier une branche très importante de l'informatique : les tests. Cependant, souvenez-vous de l'adage de DISJKSTRA (un des plus célèbres informaticiens au monde, pris TURING 1972) : ce n'est pas parce que ça marche trois fois que ça marche toujours (cette remarque doit particulièrement attirer l'attention des utilisateurs de windows).

Vous l'avez compris, si nous pouvons trouver un moyen plus simple, rapide et surtout si nous pouvons comprendre *pourquoi* ça marche, cela nous permettra de programmer plus sûrement.

Il suffit de considérer le problème *autrement*. À chaque combat correspond un(e) étudiant(e) éliminé(e). Nous dirons bientôt que l'ensemble (fini) des combats et l'ensemble des étudiant(e)s éliminé(e)s sont en *bijection*.
Le nombre de combats est donc le même que le nombre d'étudiant(e)s éliminé(e)s. Or il n'y a qu'un(e) seul(e) étudiant(e) qui n'est pas éliminé(e) à la fin. Ceci *démontre* qu'il y aura toujours un combat de moins que le nombre d'étudiant(e)s, et ceci *sans aucun calcul*, sans dérivée, sans intégrale, sans identité remarquable...

3 « J'ai toujours été nul (ou bon) en maths »

Comment pouvez-vous le savoir ? Pensez-vous vraiment que calculer (bien ou mal) dix-mille dérivées, factoriser mille polynômes, ânonner ses identités remarquables ou le théorème de Pythagore, cela fait de vous un(e) mathématicien(ne) ?
Et si vous n'aviez encore jamais fait de mathématiques ?

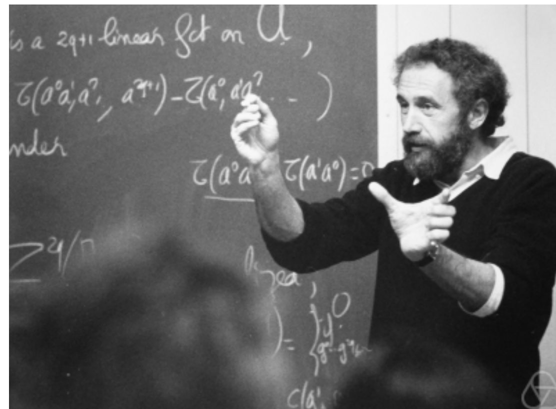
> **Knuth** */knooth'/ /n./*
> *[Donald E. Knuth's "The Art of Computer Programming"] Mythically, the reference that answers all questions about data structures or algorithms. A safe answer when you do not know : "I think you can find that in Knuth." Contrast literature, the. See also bible. There is a Donald Knuth home page at http ://www-cs-faculty.Stanford.EDU/ knuth.*

in « The New Hacker's Dictionnary »
http://outpost9.com/reference/jargon/jargon_26.html#TAG1004

La « Bible » du programmeur s'intitule « The Art of Computer Programming » [a].
La programmation tout comme les mathématiques (tout au moins quand vous programmerez dans les modules estampillés « mathématiques » à l'IUT ;-)) doivent être dans votre esprit un Art et vous devez vous considérer à la fois comme un(e) scientifique et un(e) artiste : il

a. Mais comme la Bible, quelques informaticiens ont ce livre dans leur bibliothèque mais très peu en ont lu plus que la couverture.

faut faire preuve de curiosité, d'imagination, de liberté, essayer, se tromper, recommencer, faire preuve d'originalité, trouver son style, savoir apprécier la beauté d'un programme (ou d'une démonstration, c'est la même chose). Chacun peut avoir des goûts différents, aura besoin de partager avec ses camarades, discuter de ses conjectures.
Il faudra copier quelques œuvres maîtresses, bien sûr, en allant par exemple voir le code source d'un beau langage que nous allons étudier, mais il faudra aussi créer vos propres œuvres, modestes au départ, mais c'est en créant que vous progresserez.
Un peintre est couvert de peinture, un mathématicien de craie, un informaticien de pizza et de bière mais c'est le quotidien des artistes.
En fait, jusqu'à maintenant, on a essayé en cours de faire de vous de (mauvais) ordinateurs. Il n'est certes pas négligeable de savoir calculer comme le précise Alain CONNES, membre de l'Académie des sciences, Professeur au Collège de France, à l'I.H.E.S. et à l'Université de Vanderbilt aux États-Unis qui a de plus reçu la Médaille Fields en 1982, le Prix Crafoord en 2001 et la Médaille d'or du C.N.R.S. en 2004.



> *Quand on effectue un long calcul algébrique, la durée nécessaire est souvent très propice à l'élaboration dans le cerveau de la représentation mentale des concepts utilisés. C'est pourquoi l'ordinateur, qui donne le résultat d'un tel calcul en supprimant la durée, n'est pas nécessairement un progrès. On croit gagner du temps, mais le résultat brut d'un calcul sans la représentation mentale de sa signification n'est pas un progrès.*
>
> *Alain* CONNES - *Sciences et imaginaire*

mais bon...Il est parfois bon de se mettre à la place de la machine pour mieux la faire fonctionner mais il est également utile de prendre de la distance, de voir le problème sous différents angles...comme un(e) mathématicien(ne).

4 L'art abstrait

Voici un extrait du Petit Larousse :

> **ABSTRACTION** n. f. (bas latin *abstractio*)
>
> Opération intellectuelle qui consiste à isoler par la pensée l'un des caractères de quelque chose et à le considérer indépendamment des autres caractères de l'objet.
>
> Ne pas confondre faire abstraction et abstraire.
>
> *Faire abstraction de* = ne pas faire entrer en ligne de compte dans un calcul, un raisonnement. Faire abstraction des inconvénients.
>
> *Abstraire* = isoler par la pensée pour considérer à part. Abstraire un événement de son contexte.

Alors c'est quoi être mathématicien(ne) ?! Personne ne pourra vous le dire, pas même les mathématicien(ne)s !
Disons cependant qu'un(e) mathématicien(ne) a plutôt tendance à rechercher n'importe où des structures, des motifs et à trouver des ressemblances là où on ne les voit pas du premier coup d'œil, à *abstraire* des objets pour mieux les mettre en *correspondance*. En introduction, nous

Licence Creative Commons

avons ainsi mis en correspondance les combats et les perdants : les mathématicien(ne)s ont naturellement tendance à « plonger » des objets dans d'autres environnements pour mieux les manipuler. Ces capacités ont été, et sont d'ailleurs de plus en plus développées en informatique. Nous allons très vite nous en rendre compte en programmant avec Haskell.
En mathématiques, on a également tendance a être très ouvert d'esprit et à relativiser beaucoup de choses. C'est un domaine qui a énormément évolué au cours de l'histoire, et surtout depuis un siècle : vous avez appris trois ou quatre théorèmes au lycée ? On en publie actuellement 30 000 par an ! Les théorèmes, une fois démontrés sont vrais mais souvent dans un domaine très réduit qui correspondait à l'univers de l'époque. Pour prendre un exemple simple, vous avez appris que la somme des angles d'un triangle fait 180°mais c'est faut si le triangle est tracé sur la Terre, sur un arbre...
En fait, les ronds peuvent être carrés, on peut calculer le logarithme d'un nombre négatif, une tasse de café peut être égale à un beignet, $2+2$ peut être égal à 5 : cela dépend du contexte. Il faut être très ouvert d'esprit, combattre les fausses certitudes.

> « In summary, I would like to convey a little positive message to you...I do not have any...Would two negative messages suit you ? »
>
> Woody Allen

J'aurais tendance à dire que les mathématiques sont partout : dans le CSS, dans Haskell, dans les registres de la mémoire, dans les bases de données, dans les sorties graphiques, le son, l'interface homme-machine, la spécification, les tests, la conception des langages, l'intelligence artificielle...

5 Insecticide

EN mathématiques, on prend l'habitude de relativiser mais aussi de chercher le petit grain de sable qui pourrait faire capoter notre théorie...le petit bug qui empèche notre programme de fonctionner.
Allez, je vous l'accorde, **parfois** les mathématiques traitent des calculs. Voici une petite anecdote croustillante.
La Marine US s'équipe de PC montés avec...Windows NT 4.0 et en profite pour réduire le 10% le nombre de marins à bord du USS Yorktown. Mais, comme le dira un des experts civils après la panne : « *Using Windows NT, which is known to have some failure modes, on a warship is similar to hoping that luck will be in our favor* »
Que s'est-il passé le 21 septembre 1997 ? Un opérateur a tapé par erreur un zéro sur son clavier et tout le navire a été bloqué en pleine mer :-)
« *Your $2.95 calculator, for example, gives you a zero when you try to divide a number by zero, and does not stop executing the next set of instructions. It seems that the computers on the Yorktown were not designed to tolerate such a simple failure.* »
On espère que les missiles atomiques sont mieux programmés...
Un(e) mathématicien(ne) est **parfois** cette personne maniaco-dépressive qui ne va pas vous croire sur parole mais va vérifier si votre programme n'a pas négligé un cas important qui ferait capoter tout votre travail.
C'est aussi une personne habituée à communiquer, échanger ses idées avec ses pairs via un tableau et une craie, à tenter d'être le plus clair possible (si si !). C'est pourquoi on cherchera à concevoir et même présenter son code de la manière la plus concise, limpide possible.
On ne négligera pas les exceptions et les cas pathologiques. Un drame informatique récent doit vous y faire penser.
Le 1er juin 2009, le vol Air France 447 finit tragiquement : les 228 personnes à bord meurent englouties au large des côtes du Brésil.
Beaucoup de bruit a été fait autour de cet accident, beaucoup de gens ont été blâmés, en particulier les copilotes.
Voici le problème : pour économiser du carburant, les avions volent maintenant très haut et en limite de décrochage. Il faut un guidage informatique très fin pour le permettre. Ce soir-là, l'avion est pris dans une tempête et le froid extrême obstrue les capteurs qui ne peuvent plus mesurer la vitesse du vent.

Le pilote automatique envoie alors comme message à ses collègues humains des messages trop vagues :
Les messages de panne successivement affichés sur l'ECAM n'ont pas permis à l'équipage de faire un diagnostic rapide et efficace de la situation dans laquelle l'avion se trouvait, en particulier de l'obstruction des sondes Pitot. Il n'a jamais été en mesure de faire le lien entre les messages qui sont apparus et la procédure à appliquer, alors que la lecture de l'ECAM et des messages doit faciliter l'analyse de la situation et permettre d'organiser le traitement des pannes. Plusieurs systèmes avaient pourtant identifié l'origine du problème mais n'ont généré que des messages (extrait du rapport final du Bureau d'Enquêtes et d'Analyses pour la sécurité de l'aviation civile).
Donner les conséquences et pas les causes ! Sans indications, les pilotes perdent de précieuses secondes et ne peuvent effectuer la manœuvre habituelle permettant de contrer un décrochage.

Nous allons bientôt nous initier à un langage très important qui, nous allons voir, évite un nombre important d'erreurs car il est structuré algébriquement pour son bien ou plutôt pour celui des usagers.

6 Langage

En devenant informaticien(ne), votre travail va consister à résoudre des problèmes en créant de nouveaux mondes, tout comme en mathématiques : votre client vous posera un problème qu'il voudrait que vous résolviez pour lui. Il s'exprimera le plus souvent en langage naturel, et il faudra passer à un langage beaucoup plus concret et limité, celui de la machine. Le langage mathématique se situe exactement au milieu de ces deux niveaux et vous permettra de glisser du langage naturel avec toutes ses ambiguïtés, ses sous-entendus, ses abstractions vers le langage beaucoup plus limité de votre ordinateur.

7 ...les maths c'est quand même un peu dur

Bon, il faut l'avouer, même si on va normalement voir les maths totalement différemment, ça reste une aventure. Voici le témoignage d'Andrew Wiles, le mathématicien britannique qui a enfin réussi à démontrer le grand théorème de Fermat après trois siècles d'efforts de toute la communauté mathématique. Il nous décrit par une analogie son travail en mathématique :

> *You enter the first room of the mansion and it's completely dark. You stumble around bumping into the furniture but gradually you learn where each piece of furniture is. Finally, after six months or so, you find the light switch, you turn it on, and suddenly it's all illuminated. You can see exactly where you were. Then you move into the next room and spend another six months in the dark. So each of these breakthroughs, while sometimes they're momentary, sometimes over a period of a day or two, they are the culmination of, and couldn't exist without, the many months of stumbling around in the dark that precede them.*

Et oui, ne vous inquiétez pas, vous verrez la lumière au bout du tunnel...

RECHERCHES

Recherche 1 - 1 Calcul sans calculs

Tracez un rectangle quelconque. Placez un point M n'importe où sur un côté et reliez-le aux extrémités C et D du côté opposé. Vous obtenez un magnifique triangle MCD. Calculez le rapport entre l'aire de ce triangle et l'aire du rectangle...sans aucun calcul mais en utilisant seulement votre puissant cerveau jusqu'à présent sous-exploité.

Recherche 1 - 2 Test de QI (ou JDC)

Voici une suite de nombres : 0 1 2 3 4 5.
En utilisant seulement votre puissant cerveau jusqu'à présent sous-exploité, déterminez quel est le nombre qui suit 5.

Recherche 1 - 3 Ah...les dérivées

Vous êtes d'accord avec ça :

$$\begin{aligned} 1^2 &= 1 \\ 2^2 &= 2+2 \\ 3^2 &= 3+3+3 \\ &\vdots \\ x^2 &= \underbrace{x+x+\cdots+x}_{x \text{ termes}} \end{aligned}$$

C'est vrai pour n'importe quel entier x.
Bon, je dérive la dernière égalité. J'obtiens :

$$2x = \underbrace{1+1+\cdots+1}_{x \text{ termes}} = x$$

Comme c'est vrai pour n'importe quel entier x, je peux prendre $x = 1$:

$$2 = 1$$

ou $x = 7$:

$$14 = 7$$

Recherche 1 - 4 Ah...les identités remarquables

« Reprenez donc un peu de thé » propose le Lièvre de Mars.
« Je n'ai rien pris du tout, je ne saurai donc reprendre de rien ! »
« Vous voulez dire que vous ne sauriez reprendre de quelque chose » répartit le Chapelier.
« Quand il n'y a rien, ce n'est pas facile d'en reprendre ».

— Alors comme ça, vous êtes étudiante ?

— Oui, en informatique par exemple.

Licence Creative Commons

— Alors que vaut cette fraction : un sur deux sur trois sur quatre ?
— Eh bien ...
— Elle vaut deux tiers, la devança le Loir.
— Ou trois huitièmes si vous préférez, ajouta le Lièvre de Mars.
— Ou encore un sur vingt-quatre, affirma le Chapelier.
— En fait, je crois que...
— Aucune importance ! Dîtes-nous plutôt combien vous voulez de sucre dans votre thé ?
— Deux ou trois, ça dépend de la taille de la tasse.
— Certainement pas, car de toute façon, deux ou trois c'est pareil.
— Parfaitement ! approuva le Loir en fixant Alice qui écarquillait les yeux.
— Ce n'est pourtant pas ce qu'on m'a appris, fit celle-ci.
— Pourtant, ce n'est pas compliqué à comprendre, en voici une démonstration des plus élémentaires
On sait que pour tout entier n on a successivement

$$(n+1)^2 = n^2 + 2n + 1$$

$$(n+1)^2 - 2n - 1 = n^2$$

Retranchons $n(2n+1)$ des deux côtés

$$(n+1)^2 - (n+1)(2n+1) = n^2 - n(2n+1)$$

Mézalor, en ajoutant $(2n+1)^2/4$, on obtient

$$(n+1)^2 - (n+1)(2n+1) + \frac{(2n+1)^2}{4} = n^2 - n(2n+1) + \frac{(2n+1)^2}{4}$$

Soit

$$\left((n+1) - \frac{2n+1}{2}\right)^2 = \left(n - \frac{2n+1}{2}\right)^2$$

En passant à la racine carrée, on obtient

$$(n+1) - \frac{2n+1}{2} = n - \frac{2n+1}{2}$$

d'où

$$n + 1 = n$$

Et si je prends $n = 2$, j'ai aussitôt $3 = 2$
— Alors, qu'est-ce que vous en dites ?
— Je...commença Alice.
— D'ailleurs, cela prouve que tous les entiers sont égaux, la coupa le Lièvre de Mars.
— Pas mal du tout ! Qu'en dites-vous mademoiselle l'informaticienne ?
— Je vais vous dire tout de suite ce que j'en pense.
— Ah non ! Nous préférerions de loin que vous pensiez ce que vous allez nous dire.
— C'est pareil ! grinça Alice qui commençait à en avoir assez.
— Comment ça, c'est pareil ? Dire ce que l'on pense ce serait pareil que penser ce que l'on dit ? S'étrangla le Lièvre de Mars.
— Incroyable ! Et manger ce qu'on voit ce serait pareil que voir ce qu'on mange ?
— Mais...
— Et respirer quand on dort pareil que dormir quand on respire ?
— En logique, nous vous mettons 3 sur 5.
— Autant dire moins que un.
— C'est à dire zéro, puisque si 2=3 alors 1=0.
— Parce que chez vous, 3 c'est moins que 1 ? s'indigna Alice.

Licence Creative Commons

— On se demande ce qu'on vous apprend à l'école !Tiens, et je peux même vous donner une démonstartion beaucoup plus simple.
Prenez deux nombres non nuls et égaux :
$$a = b$$
Alors
$$a^2 = ab$$
et
$$a^2 - b^2 = ab - b^2$$
Un peu d'identité remarquable :
$$(a+b)(a-b) = b(a-b)$$
Je simplifie :
$$a + b = b$$
Mais comme $a = b$:
$$2b = b$$
et je simplifie :
$$2 = 1$$

— C'est de la folie pure, pensa Alice...

Merci à Pierre Osadtchy

Recherche 1 - 5 Ah...les limites

$$\lim_{n\to+\infty} \frac{sin\ x}{n} = 0 \to \lim_{n\to+\infty} \frac{si\cancel{n}x}{\cancel{n}} = 0 \to \lim_{n\to\infty} six = 0 \to 6 = 0$$

Recherche 1 - 6 C'est logique

Dr. McCoy : *Mr. Spock, remind me to tell you that I'm sick and tired of your logic.*
Spock : *That is a most illogical attitude.*

in « Star Trek : The Galileo Seven » (1967)

Voici un petit test proposé [b] il y a quelques années à des étudiants entrant à l'Université et à des professeurs de mathématique :

Soit un entier naturel n inférieur à 20. On considère la proposition « si n est pair, alors son successeur est premier ». Quels sont les entiers qui rendent cette proposition vraie ?

Ne lisez pas ce qui suit et répondez tout de suite [c].

Ça y est ? Bon, analysons les réponses possibles. Notons $E = [\![0, 20]\!]$. Vous avez répondu...

— ... « 1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 12, 13, 15, 16, 17, 18, 19 » : vous avez triché ou bien vous avez eu un bon cours de logique au lycée et vous l'avez bien étudié. La lecture des paragraphes suivants va vous permettre d'aller plus loin ;

— ... « 2, 4, 6, 10, 12, 16, 18 » : c'est bien, vous n'avez pas triché mais malheureusement ce n'est pas la bonne réponse. La lecture des paragraphes suivants devrait vous permettre de progresser en logique pour affronter sereinement vos deux années d'IUT.

b. Cité dans la note de synthèse de l'habilitation à diriger des recherches soutenue par Viviane DURAND-GUERRIER le 16 juin 2005 http://tel.archives-ouvertes.fr/docs/00/20/16/26/PDF/hdrvdg.pdf page 47.
c. Personne ne sera au courant de votre réponse, alors n'ayez pas peur...

— ...« tous » : vous avez eu raison de ne pas sortir le jeudi soir pour travailler et vous remettre à niveau. La lecture des paragraphes suivants devrait vous permettre de découvrir la logique, l'arithmétique et la mathématique en général pour affronter sereinement vos deux années d'IUT.

Recherche 1 - 7

Les images des entiers naturels par la fonction $p : n \mapsto n^2 - n + 41$ sont des nombres premiers.

Recherche 1 - 8

Vous êtes sur la Terre que l'on va considérer comme sphérique. Vous êtes sur l'équateur. Vous avez un feutre et une équerre : vous tracez un angle droit avec l'équateur et vous allez « tout droit ». Pensez à notre histoire de somme des angles d'un triangle.

THÈME

2

Compter sur Haskell

« Ah c'qu'elles sont jolies les filles de mon pays » chantait Enrico Macias. Comme lui, vous allez être subjugué(e) par la beauté de Haskell, qui est un des langages les plus célèbres de la programmation fonctionnelle. Il ne sera pas gênant de l'explorer en parallèle de votre apprentissage du C car les concepts sont extrêmement différents et il faudra bien cloisonner ces deux langages : rien de pire que d'essayer de traduire du C en Haskell ou vice-versa. Nous allons apprendre à penser différemment en nous plongeant dans un monde pur et élégant...

> *The main idea is to regard a program as a communication to human beings rather than as a set of instructions to a computer.*
>
> D. E. KNUTH. "Literate Programming (1984)" in Literate Programming. CSLI, 1992, p. 99

Haskell CURRY

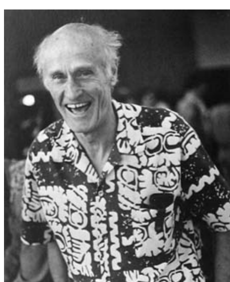
Haskell Brooks CURRY (1900-1982) est un mathématicien américain dont l'influence en informatique a été telle que trois langages de programmation portent son nom : Haskell, Brooks et Curry...
Il a fait partie de cette génération qui a lancé les fondements de la logique et de l'informatique : Alonzo CHURCH, Alan TURING, Steven KLEENE.
Ses travaux sur la logique combinatoire sont les fondements du langage de programmation fonctionnel Haskell créé dans les années 1990 et qui nous servira de support à nos travaux informatiques qui vous permettront tout au long des deux années à venir de vous familiariser avec la programmation fonctionnelle dont le rôle est de plus en plus important dans le monde informatique.

Alan TURING

Puisque nous sommes à l'Université, nous pourrions commencer ce cours en disant que les langages « à la VON NEUMANN » (comme C, Python, Java, etc.) sont basés sur les concepts de variable et d'instruction. Ces concepts sont assez intuitifs car ils correspondent à de nombreuses situations « de la vie de tous les jours » : nous agissons sur et en fonction de notre environnement immédiat. Nous portons le même nom mais nous changeons d'aspect, de taille, de poids,...Ce modèle dérive des travaux de TURING et de sa machine universelle dans les années 1930. D'autres mathématiciens ont proposés des modèles différents à la même époque : le λ-calcul d'Alonzo CHURCH, les fonctions récursives de Stephen KLEENE, la logique combinatoire de Haskell CURRY. Ces trois derniers modèles n'utilisent pas le concept de machine et de mémoire de TURING mais uniquement de valeur, de calcul d'autres valeurs à partir de fonctions qui sont elles-mêmes des valeurs.
Mmmmmm...Ça y est, tout le monde est perdu. Il va falloir changer de stratégie. Bon, revenons un peu en arrière, en cette rentrée en Sixième où vous n'étiez pas encore adolescent mais un(e) jeune enfant spontané(e) et heureux(se)...

1 Collège

Pour votre goûter d'anniversaire vous confectionnez 3 sacs de 9 fraises Tagada et de 5 nounours : combien y a-t-il de bonbons en tout ?
Vous traduisez le problème en calcul :



Stephen KLEENE

$3 \times (9+5)$
$\Rightarrow 3 \times 14$
$\Rightarrow 42$
mais vous auriez pu compter d'abord le nombre total de fraises et celui de nounours :
$3 \times (9+5)$
$\Rightarrow 3 \times 9 + 3 \times 5$
$\Rightarrow 27 + 3 \times 5$
$\Rightarrow 27 + 15$
$\Rightarrow 42$
Ce qu'il faut bien remarquer, c'est qu'on obtient le même résultat dans chaque cas. Pour une même entrée $3 \times (9+5)$, on a la même résultat 42. C'est ce qu'on attend d'une **fonction** et c'est un des aspects de la *pureté* de Haskell.
Bon, et maintenant, que se passe-t-il si vous avez deux invités de plus à votre anniversaire ? Ou si vous avez plus de nounours que prévus ?
Vous pouvez recommencer le calcul : $5 \times (9+5)$ ou bien vous pouvez *généraliser* le problème. Vous pouvez créer une fonction qui calcul le nombre de bonbons quelque soit le nombre d'invités, de fraises Tagada et de nounours.

Aparté technique

En C vous feriez :

C
```
#include <stdio.h>

int bonbons(int,int,int);

void main(void)
{ int x,y,z,b;

  printf("Entrez le nombre d'invités: ");
  scanf("%d" , &x);
  printf("Entrez le nombre de fraises Tagada par paquets: ");
  scanf("%d" , &y);
  printf("Entrez le nombre de nounours par paquets: ");
  scanf("%d" , &z);
  b = bonbons(x,y,z);
  printf("Le nombre total de bonbons est %d \n", b);
}

int bonbons(int x, int y, int z){
    return x*(y + z);}
```

Puis on compile et on exécute :

Shell
```
$ gcc -o bonbons bonbons.c
$ ./bonbons
Entrez le nombre d'invités: 3
Entrez le nombre de fraises Tagada par paquets: 9
Entrez le nombre de nounours par paquets: 5
Le nombre total de bonbons est 42
```

En Haskell, on va plutôt écrire dans GHCi :

Haskell
```
λ> let f x y z = x*(y + z)
λ> f 3 9 5
42
```

ou, pour être plus explicite :

Haskell
```
λ> let bonbons invites fraises nounours = invites*(fraises + nounours)
λ> bonbons 3 9 5
42
```

ou dans un fichier bonbons.hs

Haskell
```
{- fonction calculant le nombre de bonbons selon
   le nombre d'invités, de fraises Tagada et de nonours -}
bonbons invites fraises nounours =
    invites*(fraises + nounours)

main = print  (bonbons 3 9 5)
```

qu'on peut compiler :

Shell
```
$ ghc bonbons.hs -o bonbons
[1 of 1] Compiling Main             ( bonbons.hs, bonsons.o )
Linking bonbons ...
$ ./bonbonsHs
42
```

mais dans un premier temps nous nous contenterons de travailler via GHCi donc nous aurons une fenêtre avec à gauche notre fichier hs et à droite GHCi. Dans Emacs, il suffit de taper C-c C-l ou de charger le fichier avec la commande `:l` (load) ou `:r` (reload) dans GHCi

Haskell
```
λ> :l ./bonbons.hs
Ok, modules loaded: Main.
λ> bonbons 3 9 5
42
λ> bonbons 5 2 3
25
```

Bon, on s'éloigne du sujet. Ce ne sont là que des détails d'utilisation.

Aparté

Avoir pour seul but l'apprentissage d'un langage de programmation n'a que peu d'intérêt. Apprendre un langage pour résoudre des problèmes va notablement accroître vos capacités intellectuelles.
Programmer, ce n'est pas connaître la syntaxe tout comme peindre n'est pas savoir bouger un pinceau, connaître l'alphabet n'est pas être écrivain....et faire des mathématiques ne consiste pas seulement à calculer des dérivées...

Retour à l'essentiel

Que se passe-t-il (à peu près...) quand nous tapons bonbons `3 9 5` ?
D'abord il faut comprendre que dans bonbons `x y z` et bonbons `3 9 5`, les rôles joués par `x, y, z` d'une part et `3, 9, 5` de l'autre sont différents.
Il y a d'abord une phase d'*expansion* (ou *unfolding : dépliage*) de la définition de la fonction. On substitue aux *paramètres* formels `x, y, z` les *arguments* (ou paramètres effectifs) `3, 9, 5`.
Ensuite 9+5 est *réduit* en 14.
Plus précisément l'*expression* 9+5 est *évaluée* en 14.

À retenir

On appelle en Haskell les « choses » que l'on calcule des *expressions* et les « choses » qui résultent du calcul des *valeurs* : on ne peut plus les réduire.
Par exemple, $9 + 5$ est une expression et 14 une valeur. Mais 14 est aussi une expression... Toute valeur est une expression (car tout est expression...) mais toute expression n'est pas une valeur.
En fait, une expression, c'est ce qu'on rentre devant le *prompt* $\lambda >$ et une valeur, c'est ce que renvoie GHCi :

Haskell
```
λ> 9 + 5
14
λ> 14
14
λ> bonbons 3 9 5
42
```

Il y a parfois des expressions qui entraîne une infinité de calculs comme $x = x + 1$:
x
$\Rightarrow x + 1$
$\Rightarrow (x + 1) + 1$
$\Rightarrow ((x + 1) + 1) + 1$
...
Cette expression *diverge*. Comme toute expression a une valeur, on lui donne quand même un nom : $\perp$ qui se lit « antitruc ».

Preuve des propriétés d'un programme

La somme de deux entiers a une propriété bien connue : quelque soit les nombres a et b, alors $a + b = b + a$. On dit alors que l'addition des entiers est *commutative*.

Est-ce que par hasard, la fonction bonbons aurait une propriété similaire ?

Il est assez simple de démontrer que `bonbons x y z` = `bonbons x z y`.

bonbons $x\ y\ z$
$\Rightarrow x * (y + z)\ \{expansion\}$
$\Rightarrow x * (z + y)\ \{commutativit\acute{e}\}$
$\Rightarrow$ bonbons $x\ z\ y\ \ \{r\acute{e}duction\}$ La dernière étape est rarement accomplie par un langage informatique mais cela n'empêche pas un(e) informaticen(ne) de l'accomplir.

2 Opérateurs

On reste au collège. On s'intéresse à l'évaluation des opérations arithmétiques sur machine (voir Recherche 2 - 2 page 24).

Le rôle des parenthèses est le plus souvent primordial sur machine :

Haskell
```
λ> 5 + 4*3
17
```

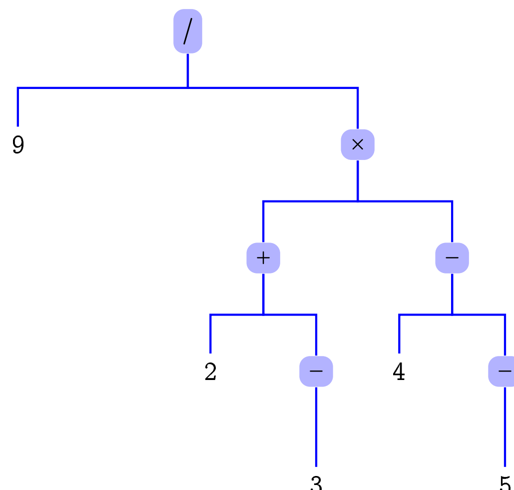
et ce quelque soit le langage car des règles de priorité ont été établies.

Sinon, des parenthèses auraient dû être mises :

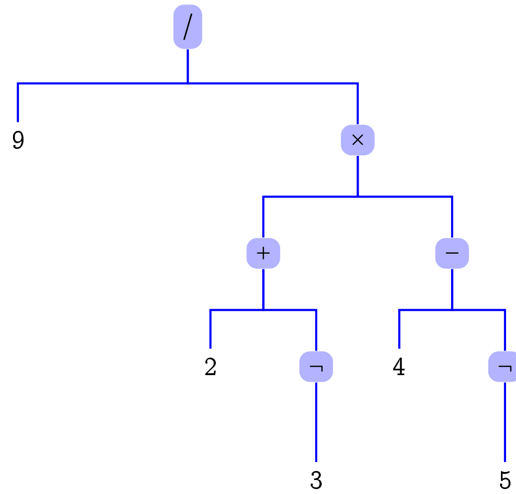
Haskell
```
λ> 9 / 2 + (-3)*4 - (-5)
-2.5
λ> 9 / ((2 + (-3)) * (4 - (-5)))
-1.0
```

Or dans un calcul plus compliqué, ne pas se soucier des parenthèses peut entraîner de mauvaises surprises.

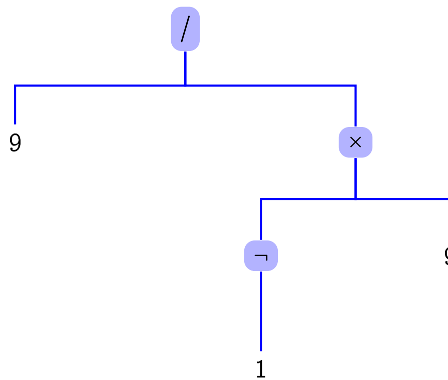
Il est sûrement plus clair de visualiser le calcul à l'aide d'un arbre



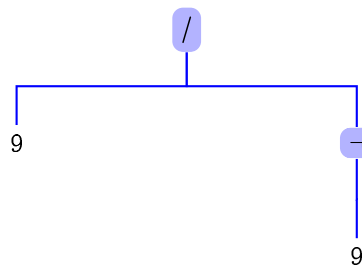
L'usage du − est ici ambigu : cet opérateur est parfois binaire, parfois unaire. Pour éviter ce problème, on peut par exemple utiliser des symboles différents :



On réduit alors l'arbre en remontant des feuilles vers la racine :



puis



et finalement $\neg 1$.
À partir de l'arbre, on obtient donc une *séquence* de lecture *infixée* en *profondeur* :

$$9/((2+\neg 3)\times(4-\neg 5))$$

L'emploi des parenthèses agit comme une sorte de *retardateur* de l'évaluation.
Comment une machine évalue une telle expression ?
Il s'agit d'abord d'étudier la Recherche 2 - 4 page 25.

Définition 2 - 1

Associativité
Soit $\star$ un opérateur quelconque défini sur un ensemble quelconque E.
On dit que l'opérateur $\star$ est **associatif** sur E si, et seulement si, pour tout triplet (x, y, z) d'élements de E on a :

$$(x \star y) \star z = x \star (y \star z)$$

Et voici une définition proche qui concerne plutôt les langages de programmation :

Définition 2 - 2

Associativité à droite (à gauche)
Avec les mêmes notations que la définition précédente, un opérateur est **associatif à droite** si, et seulement si :

$$x \star y \star z = x \star (y \star z)$$

et **associatif à gauche** si, et seulement si :

$$x \star y \star z = (x \star y) \star z$$

3 Notation polonaise

Jan Łukasiewicz

En 1920, le mathématicien-logicien-philosophe polonais Jan Łukasiewicz (1878-1956) invente la notation préfixée alors qu'il est ministre de l'éducation (on a le droit de rêver...).
35 ans plus tard, le philosophe et informaticien (!) australien (!!!) Charles Hamblin (1922 - 1985) s'en inspire.
Il a en effet en sa possession l'un des deux ordinateurs présents à l'époque en Australie. Il se rend compte que la saisie de calculs avec les opérateurs en notation infixée induit de nombreuses erreurs de saisie par oubli de parenthèses. Il pense aussi à la (très petite) mémoire des ordinateurs de l'époque.
Il trouve alors son inspiration dans le travail de Łukasiewicz pour éviter les parenthèses et il pense à mettre les opérateurs en position préfixée : ainsi il introduit la notion de *pile* (ou *stack* ou Lifo) qui économise le nombre d'adresses mémoire nécessaires.
C'est la naissance de la Notation Polonaise Inversée (NPI ou RPN). Elle économise également la taille des composants électroniques des portes logiques.
Son seul inconvénient : les mauvaises habitudes prises d'utiliser des notations infixées...
Un autre grand avantage : il faut comprendre le calcul avant de l'exécuter :-)

Pile : découverte

Une pile (en anglais *stack*) est une structure de données permettant l'insertion d'un élément et la suppression du dernier élément inséré.
L'acronyme anglais associé est LIFO (*Last In First Out*).
Alan Turing proposa dès 1945 les termes *bury* et *unbury* pour désigner les deux méthodes basiques de traitement des piles.
On utilise en français *empiler* et *dépiler* alors que les termes anglais actuellement utilisés sont *push* et *pop*.
Imaginez une partie de cartes qui nécessite une pioche. La seule carte qui nous intéresse est celle du dessus. Piocher une carte consiste à retirer la carte du dessus et à la lire (dépiler) ou rajouter une carte *sur* le tas (empiler). Toute information supplémentaire est superflue.
De nombreuses situations sont similaires en informatique et....*dans la vraie vie.*
D.E. Knuth propose cette illustration (page 240 de Knuth [1997])

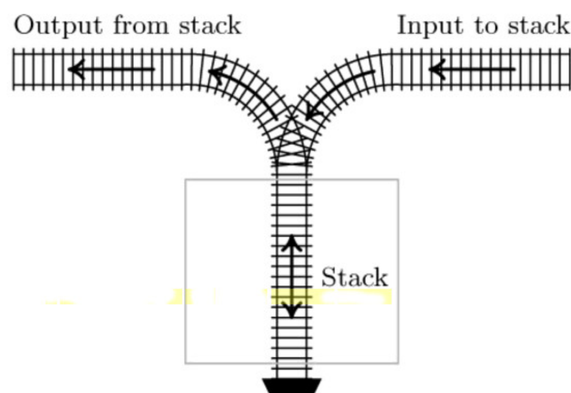
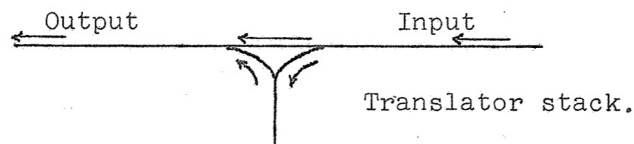


Fig. 1. A stack represented as a railway switching network.

qui reprend l'article fondateur de E.W. *Dijkstra* (Dijkstra [1961]) dont nous reparlerons bientôt :

The translation process shows much resemblance to shunting at a three way railroad junction of the following form



Mais bon, on utilise le terme *pile* en référence à un pile d'assiette : la première que l'on saisira sera la dernière à avoir été empilée.

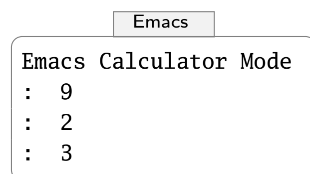
Pile et NPI

Reprenons notre expression $9/((2+\neg 3)\times(4-\neg 5))$.
La plupart des compilateurs vont plutôt la lire ainsi : `9 2 3 NEG + 4 5 NEG - * /` en utilisant une pile.
Il faut juste connaître l'*arité* de chaque opérateur, i.e. leur nombre d'opérandes, qui doit être fixe (c'est pourquoi on doit absolument distinguer **-** de **NEG** et éviter l'habituelle *surcharge* de l'opérateur **-**).

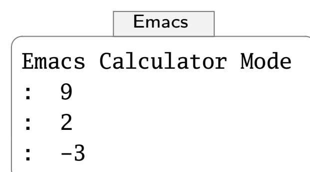
1. On lit l'expression de gauche à droite ;
2. On empile les opérandes ;
3. Dès qu'on lit un opérateur, on l'applique aux opérandes présents sur la pile selon son arité ;
4. On s'arrête quand on n'a plus rien à lire ;
5. on renvoie le dernier élément présent dans la pile.

Voici par exemple les différentes étapes du calcul effectué à l'aide du *calculator* d'Emacs :

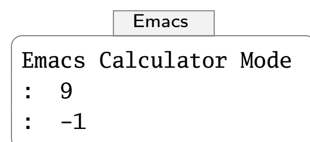
1. On entre **9** , **2** puis **3** :



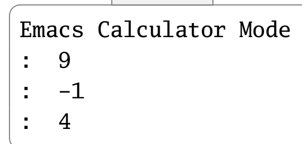
2. **NEG**



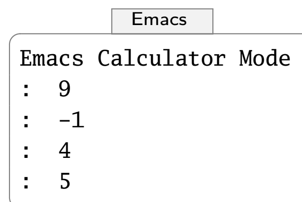
3. **+**



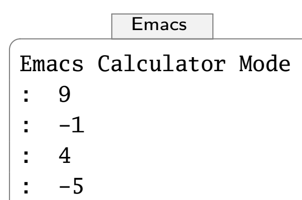
4. **4** puis Entrée



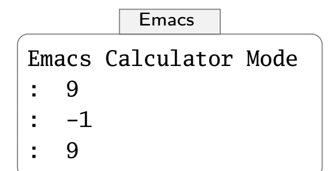
5. **5** puis Entrée



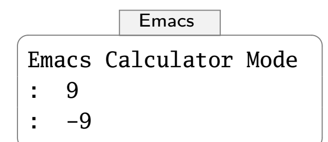
6. **NEG**



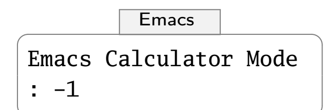
7. **-**



8. *****



9. **/**



Plusieurs avantages :

— pas besoin de parenthèses, de règles d'associativité, de priorité ;
— on voit au sommet de la pile les résultats intermédiaires ;
— le calcul est directement implémentable sur machine.

Au collège, $9/((2+\neg 3)\times(4-\neg 5))$ serait lu :

« le *rapport* entre 9 et le *produit* de la *somme* de 2 et l'*opposé* de 3 et la *différence* de 4 et de l'*opposé* de 5 »

ce qui peut se coder en :

```
/ 9 * + 2 NEG 3 - 4 NEG 5
```

C'est la méthode introduite en 1929 par Jan ŁUKASIEWICZ. On a dénommé *notation polonaise* cette méthode d'évaluation d'une expression en son honneur. On utilise également la dénomination *notation pré-fixée*. Cela explique aussi l'appellation *notation polonaise inversée* de la notation post-fixée.
Ici aussi, les opérateurs doivent être d'arité fixe et l'évaluation se fait à l'aide d'une pile.

RECHERCHES

Recherche 2 - 1 Encore des bonbons

1. Détaillez toutes les étapes de l'évaluation de l'expression bonbons (bonbons 2 3 4)5 6
2. Démontrez comme fait dans le cours que bonbons $(a - b)$ a b est évaluée par $a^2 - b^2$.

Recherche 2 - 2 Opérateur ?

Derrière le symbole + se cache en fait une fonction. Elle a la caractéristique de se placer entre ses paramètres (on dit qu'elle est en position *infixée*) et d'être symbolisée par un caractère non alphanumérique. On a l'habitude d'appeler une telle fonction un *opérateur*. Par exemple, on aurait pu définir la fonction suivante : `plus a b = a + b`.
En fait, ce n'est pas la peine sur Haskell car on peut placer l'opérateur en position *préfixée* en entourant l'opérateur de parenthèses

Haskell
```
λ> (+) 3 4
7
λ> (+) ((-) 3 4) ((*) 2 7)
13
```

En fait, la position préfixée correspond à la lecture de l'expression arithmétique :
« La somme de 3 et de 4 »
« La somme de la différence de 3 et de 4 et du produit de 2 par 7 ».
Écrivez l'expression suivante en notation infixe :

$$(5 + 4 + (2 - (3 - (6 + 4 * 5)))) * (3 * (6 - 2) * (2 - 7)) \qquad (E)$$

Comment l'évalueriez-vous sous cette forme si vous étiez une machine ?
En fait, on peut créer ses propres opérateurs du moment que l'on n'utilise pas de symbole alphanumérique ni de caractères d'échappement ni l'apostrophe qui est considérée comme alphanumérique.

Haskell
```
λ> let (<+>) x y = x + 2*y
λ> 3 <+> 5
13
```

Remplacez tous les + par des <+> dans l'expression (E) et évaluez-la.

Recherche 2 - 3 Benchmark des évaluations

Le chemin menant une expression à sa valeur n'est pas toujours le même selon les langages.
Nous allons en distinguer trois : l'évaluation par valeur et l'évaluation par nom et l'évaluation par besoin.
Dans le cas de l'évaluation par valeur, on évalue d'abord les arguments puis on « déplie » la fonction : *on va de l'intérieur vers l'extérieur.*
Dans l'évaluation par nom, on déplie d'abord la fonction puis on évalue ses arguments : *on va de l'extérieur vers l'intérieur.*
Prenons un exemple. Soit les fonctions : $\texttt{carre}\ x = x * x$ et $\texttt{somC}(x, y) = \texttt{carre}\ x + \texttt{carre}\ y$
Que se passe-t-il lors de l'appel de `somC (3,2+2)` ?

évaluation par valeur on réduit d'abord les arguments :

$\texttt{somC}\ (3, 2 + 2)$
$\texttt{somC}\ (3, 4)$
$\texttt{carre}\ 3 + \texttt{carre}\ 4$
$3 * 3 + \texttt{carre}\ 4$
$9 + \texttt{carre}\ 4$
$9 + 4 * 4$

$9+16$
25

évaluation par nom on déplie d'abord la fonction :

$\texttt{somC}\ (3,2+2)$
$\texttt{carre}\ 3+\texttt{carre}\ (2+2)$
$3*3+\texttt{carre}\ (2+2)$
$9+\texttt{carre}\ (2+2)$
$9+(2+2)*(2+2)$
$9+4*(2+2)$
$9+4*4$
$9+16$
25

évaluation par besoin comme l'évaluation par nom ici.

Il est temps de passer à un petit test comparatif. Mettez une croix dans la colonne qui correspond à l'évaluation la plus rapide pour évaluer les expressions proposées en considérant les fonctions :

$$\texttt{schmox}\ (a,b) = a*a, \qquad \texttt{bolos} = \texttt{bolos}$$

expression	APV	APN	idem
schmox (7, 8)			
schmox (3+4, 8)			
schmox (7, 2*4)			
schmox (3+4, 2*4)			
schmox (7, bolos)			
schmox (bolos, 8)			

Proposez un théorème du style : « si l'évaluation... termine alors l'évaluation.... aussi ». Que se passe-t-il pour la réciproque ?

Haskell
```
λ> let bolos = bolos
λ> let schmox (a,b) = a*a
λ> schmox (3, bolos)
9
```

Recherche 2 - 4

Bon, d'abord une question : pourquoi Haskell ne se trompe pas en calculant `5+4*3` ? Et même d'ailleurs avec `5+4+3`, il manque des parenthèses...
Et que pensez-vous de ceci :

Haskell
```
λ> let double x = x + x
λ> double 2 * double 5
40
λ> 2 + 2 * 5 + 5
17
```

et de ça :

Haskell
```
λ> (3 - 2) - 5
-4
λ> 3 - (2 - 5)
6
λ> (3 / 2) / 5
0.3
λ> 3 / (2 / 5)
7.5
```

et de cela :

Haskell
```
λ> 2 ** (3 ** 4)
2.4178516392292583e24
λ> (2 ** 3) ** 4
4096.0
```

sachant que `**` est l'opérateur des puissances de flottants.

Recherche 2 - 5 récréation

Proposez des expressions infixes, postfixes, prefixes, faites-les traduire par votre voisin(e), faites des remarques, proposez des méthodes de traduction et gardez vos notes : vous vous en servirez pour programmer tout ça....

Recherche 2 - 6

Décomposez le calcul **`/ 9 * + 2 NEG 3 - 4 NEG 5`**. Comment l'organiser sur machine ? Comparez avec la NPI. Expliquez alors le choix fait par Charles HAMBLIN d'utiliser plutôt la NPI sur machine.

Recherche 2 - 7 snake operator

Python, langage très sexy et à la mode, dispose d'une bibliothèque `operator` comprenant quelques fonctions usuelles.

Python
```
In [1]: from operator import add,mul

In [2]: mul(add(2, mul(4, 6)), add(3, 5))
Out[2]: 208
```

C'est infixe ? Postfixe ? Autre ?
Comment la machine peut évaluer cette expression ?
Il faut savoir comment Python procède.

Python
```
In [9]: def fst(c):
   ...: return c[0]
   ...:

In [10]: fst((1,2))
Out[10]: 1

In [11]: fst((1,1/0))
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-11-707229f26363> in <module>()
---→ 1 fst((1,1/0))

ZeroDivisionError: division by zero
```

Tout le monde ne fait pas pareil, vous êtes déjà au courant :

Haskell
```
λ> let fst (a,b) = a
λ> fst (1,2)
1
λ> fst (1,1/0)
1
```

Dresser un arbre d'expression qui traduit l'évalutation de `mul(add(2, mul(4, 6)), add(3, 5))` en Python et en Haskell.

THÈME

3

Let's talk about Sets

Notre exploration de la programmation fonctionnelle continue avec l'exploration des **type** mais cette notion est tellement riche et importante qu'elle mérite un chapitre à elle seule qui ne sera d'ailleurs qu'une introduction. Nous allons par la même occasion découvrir son lien avec la notion fondamentale d'**ensemble** pour mieux la comprendre.

1 C'est quoi ce Type ?

Georg Cantor

Vous avez (ou allez) étudier les types du langage C : en gros, ils indiquent comment vos objets sont représentés sur machine.
Les types en Haskell ont un rôle bien plus riche. Notre exploration des types se fera tout au long de l'année. Nous allons nous contenter d'une première approche.
Chaque expression (et donc chaque valeur) est associée à un type, ou plutôt *appartient* à un type.
Pensez à un type comme à un *ensemble* d'expressions (ou de valeurs) qui ont des choses en commun. C'est d'ailleurs cette idée qui a été proposée par Georg Cantor (1845 - 1918) à la fin du XIXᵉ siècle :

Définition 3 - 1

Principe de compréhension
Un ensemble est une collection d'objets tous distincts qu'on appelle ses éléments et qui partagent une « propriété ».

Par exemple « être un garçon » est une propriété vérifiable si l'on considère l'amas d'étudiant(e)s d'INFO 1 : on peut répondre par oui ou non à la question « truc est un garçon ».
On peut alors former l'ensemble G des garçons d'INFO 1 : ses *éléments* sont ceux qui vérifient la propriété. On peut noter alors :

$$G = \{ \text{étudiant} \mid (\text{étudiant est en INFO1}) \text{ ET } (\text{étudiant est un garçon}) \}$$

On notera indistinctement « Albert appartient à G », « Albert est un élément de G », « G contient Albert », « Albert $\in G$ », « $G \ni$ Albert ».

Notations

La notation $\in$ est due au mathématicien anglais Bertrand Russel en 1903 qui reprenait l'epsilon ε de l'italien Giuseppe Peano en 1889 mais en « l'arrondissant » compte-tenu des contraintes typographiques anglaises sous la forme ϵ qui a donné $\in$.

Un ensemble qui ne contient rien s'appelle l'*ensemble vide* et se note $\varnothing$ ou $\{\}$.

À retenir

En Haskell TOUT A UN TYPE AVANT LA COMPILATION. On parle de type *statique*. C'est très important et très utile :

— cela impose de bien clarifier sa pensée, de bien penser à la structure du programme. Avant de programmer, il faut *écrire* le type des expressions que nous allons utiliser. Ce n'est pas toujours simple mais cela a un rôle important pour...

— ... bien documenter le code : en regardant les types, on comprend mieux ce que feront vos fonctions.

— ...permettre de découvrir de nombreux bugs avant l'exécution (*run-time errors*) car elles deviennent des erreurs de compilation. Ainsi, il vaut mieux découvrir une erreur quand on tente de compiler un programme que quand on lance son exécution (par exemple un logiciel de pilotage automatique d'avion :-)). On ne soigne pas tout à la compilation mais le typage statique permet de découvrir énormément d'erreurs.

Définition 3 - 2

Axiome d'extensionalité
Deux ensembles A et B sont égaux si, et seulement si, ils contiennent les mêmes éléments. On écrit alors $A = B$.

Ça paraît idiot mais cela permet de bien cerner le problème : c'est la notion d'appartenance d'un élément à un ensemble qui est primordiale.

Il existe des types primitifs (déjà définis dans l'environnement de base) et si vous chargez une nouvelle bibliothèque, de nouveaux types seront définis. Nous apprendrons également à définir nos propres types.

Par exemple, `Integer` est l'ensemble de tous les entiers, `Char` est l'ensemble de tous les caractères, `Bool` est l'ensemble des booléens,...
GHCi propose la touche magique `:t` qui donne le type d'une expression. Et oui, Haskell devine le type de vos expressions même si vous ne l'avez pas précisé : il fait de l'*inférence de type*.

Haskell
```
λ> :t 'a'
'a' :: Char
λ> :t 2 + 2 == 4
2 + 2 == 4 :: Bool
```

À retenir

`::` se lit « ...a pour type... »

OK, pas de problème. Bon, je suppose que 3 est un entier :

Haskell
```
λ> :t 3
3 :: Num a⇒a
```

Ouh la...

Haskell
```
λ> :t 3.5
3.5 :: Fractional a⇒a
```

Ouh la la la la....

Ensembles de nombres

Vous avez appris au collège qu'il y avait des entiers positifs (naturels) qu'on regroupe dans un ensemble appelé $\mathbb{N}$ et un plus grand ensemble, $\mathbb{Z}$ qui regroupe tous les zentiers, positifs comme négatifs.
Tous les éléments de $\mathbb{N}$ sont des éléments de $\mathbb{Z}$: on dit que $\mathbb{N}$ *est inclus* dans $\mathbb{Z}$ ou que $\mathbb{Z}$ *contient* $\mathbb{N}$.

Définition 3 - 3

inclusion
L'ensemble X est inclus dans l'ensemble Y si, et seulement si, tous les éléments de X sont des éléments de Y. Cela se note

$$(X \subseteq Y) \leftrightarrow ((z \in X) \to (z \in Y))$$

On dira indifféremment « *X est contenu dans Y* », « *Y contient X* », « *X est un sous-ensemble de Y* », « *X est une partie de Y* ».

Danger

Notations
Il faudra bien distinguer $\subset$, $\subseteq$ et $\not\subseteq$. $B \subsetneq A$ signifie $(B \subseteq A$ et $B \neq A)$.
Il ne faut pas confondre avec $B \not\subseteq A$ qui exprime que B n'est pas inclus dans A. On dit que B est une partie propre de A si, et seulement si, $B \subseteq A$ avec $B \neq A$ et $B \neq \varnothing$ (il est nécessaire que A ne soit pas vide et ne soit pas un singleton).

La notion d'inclusion va nous permettre de réécrire l'axiome d'extensionalité :

Théorème 3 - 1

Axiome d'extensionalité version II

$$(\forall X)(\forall Y)(((X \subseteq Y) \wedge (Y \subseteq X)) \to (X = Y))$$

Ouh la, c'est quoi ces signes cabalistiques ? ! !

Kit de survie en logique des prédicats : les quantificateurs

Par exemple « *est un entier pair* » est une propriété définie sur l'ensemble $\mathbb{N}$ des entiers naturels que nous noterons P. Alors 2 vérifie P mais 3 non.
Ces notions s'introduiront naturellement sur notre ami Haskell :

Haskell
```
naturels         = [0..]
pairs            = [n | n←naturels, mod n 2 == 0]
petits_pairs     = [n | n←[0..15], mod n 2 == 0]
petits_pairs_bis = [2*k | k←[0..7]]
```

Alors on obtient par exemple :

Haskell
```
λ> petits_pairs
[0,2,4,6,8,10,12,14]
```

On notera aussi $2 \in$ `pairs` ou encore $2 \in \{x|(x \in \mathbb{N}) \wedge P(x)\}$ ou encore $2 \in \{t|(t \in \mathbb{N}) \wedge P(t)\}$ ou encore $P(2)$ ou encore $2 \in \{x|x \text{ est un entier naturel et } x \text{ est pair}\}$.
Ainsi $P(x)$ signifie que l'objet x vérifie la propriété P. Pour signifier que x ne vérifie pas P, on notera $\neg P(x)$ (« non P de x »).
On notera donc aussi $3 \notin$ `pairs` ou encore $3 \in \{x|(x \in \mathbb{N}) \wedge \neg P(x)\}$ ou encore $3 \notin \{t|(t \in \mathbb{N}) \wedge P(t)\}$ (est-ce équivalent :-) ?) ou encore $\neg P(3)$.

Considérons une autre propriété R : un nombre x vérifie R si, et seulement si, son carré vaut 2. Quelque soit l'entier x, on a $x^2 \neq 2$. On notera cette propriété par

$$\forall x(x \in \mathbb{N} \wedge \neg R(x))$$

$\forall$ est un *quantificateur universel* (c'est un A à l'envers, car en allemand, ce symbole se lit *für Alle* et a été introduit en 1934 par Gerhard GENTZEL...).

Considérons une autre propriété S : un nombre x vérifie S si, et seulement si, son carré vaut 4. Il existe au moins un entier qui vérifie S : il s'agit de 2. On note alors

$$\exists x(x \in \mathbb{N} \wedge S(x))$$

$\exists$ est l'autre quantificateur universel introduit par Giuseppe PEANO en 1897 (un E à l'envers comme dans *Esiste almeno uno*).
On peut même préciser que cet entier est *unique* en faisant suivre le quantificateur d'un point d'exclamation : $\exists! x(x \in \mathbb{N} \wedge S(x))$
Avec Haskell :

Haskell
```
s = λx→ x^2 == 4
```

donne

Haskell
```
λ> s 2
True
λ> s 2.0
True
λ> s 3
False
λ> any s [0..]
True
λ> find s [0..]
Just 2
λ> filter s [0..100]
[2]
```

Mmmmmm... On a encore beaucoup de choses à découvrir.

Types numériques : découverte

Comprendre comment fonctionnent les types numériques sur Haskell va nous occuper pas mal de temps. Nous allons juste ici en dresser un rapide petit catalogue.

Integer

`Integer` désigne un nombre *entier* en anglais de spécialité.
Donc `Integer` correspond à $\mathbb{Z}$.

Haskell

```
λ> 2^1000
107150860718626732094842504906000181056140481170553360744375038837035105112493612249319
837881569585812759467291755314682518714528569231404359845775746985748039345677748242309
854210746050623711418779541821530464749835819412673987675591655439460770629145711964776
86542167660429831652624386837205668069376
```

Int

`Int` est une partie d'`Integer`.
Et bien sur Haskell c'est pareil : `Int` regroupe une partie des entiers, à savoir ceux qui sont compris entre -2^{63} et $2^{63} - 1$.
Par exemple, prenons ce fameux grand entier :

Haskell

```
λ> let n = 2^63 - 1 :: Int
λ> n
9223372036854775807
```

Mais là où ça devient bizarre :

Haskell

```
λ> n * 2
-2
```

euh... Y a quelque chose qui cloche là-dedans...On va étudier ça en détail un peu plus tard (cf Recherche 3 - 9 page 39)
Si vous faites votre stage dans une banque et que vous codez vos entiers avec `Int`, ça peut poser des problèmes pour les gens très très riches.
En fait, l'`Int` maximum n'est pas toujours 9 223 372 036 854 775 807 : ça dépend des processeurs, des OS.
Pour le savoir, il existe des expressions `maxBound` et `minBound` qui renvoient les bornes inférieures et supérieures d'un type...borné (nous en reparlerons plus tard), ce qui est le cas de `Int` :

Haskell

```
λ> maxBound :: Int
9223372036854775807
λ> minBound :: Int
-9223372036854775808
```

On peut quand même travailler sur 8, 16 ou 32 bits si on le désire :

Haskell

```
λ> :m Data.Int
λ> minBound :: Int8
-128
λ> minBound :: Int16
-32768
λ> minBound :: Int32
-2147483648
```

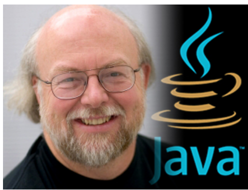
On obtient très facilement la liste de tous les éléments de `Int8` par exemple :

Licence Creative Commons

Haskell
```
λ> [minBound :: Int8 ..]
[-128,-127,-126,-125,-124,-123,-122,-121,-120,-119,-118,-117,-116,-115,-114,-113,-112,
-111,-110,-109,-108,-107,-106,-105,-104,-103,-102,-101,-100,-99,-98,-97,-96,-95,-94,
-93,-92,-91,-90,-89,-88,-87,-86,-85,-84,-83,-82,-81,-80,-79,-78,-77,-76,-75,-74,-73,
-72,-71,-70,-69,-68,-67,-66,-65,-64,-63,-62,-61,-60,-59,-58,-57,-56,-55,-54,-53,-52,
-51,-50,-49,-48,-47,-46,-45,-44,-43,-42,-41,-40,-39,-38,-37,-36,-35,-34,-33,-32,-31,
-30,-29,-28,-27,-26,-25,-24,-23,-22,-21,-20,-19,-18,-17,-16,-15,-14,-13,-12,-11,-10,
-9,-8,-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,
23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,
52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,
81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100,101,102,103,104,105,106,
107,108,109,110,111,112,113,114,115,116,117,118,119,120,121,122,123,124,125,126,127]
```

Float et Double

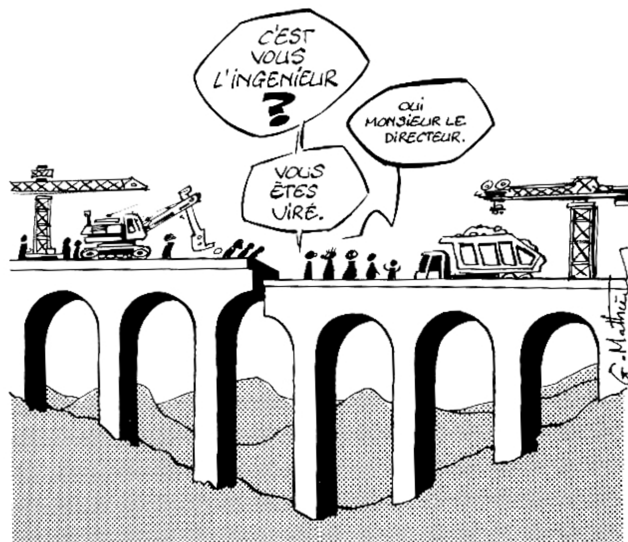
Alors là, ça devient très compliqué, tellement compliqué que nous consacrerons deux mois à l'étude des nombres à virgule flottante au printemps...



James GOSLING

> *95 % of the folks out there are completely clueless about floating-point*
>
> James GOSLING (M. Java) - 28 février 1998

On retiendra juste qu'il s'agit de l'ensemble des nombres « à virgule ». Tant qu'ils ne sont ni trop gros, ni trop petits, ni trop proches quand on les soustrait, tout va bien, ils se comportent comme s'ils étaient des décimaux. Mais quand on les utilise près des zones à risque....



Les autres types de base

Les booléens

L'ensemble des booléens est assez limité puisqu'il n'a que 2 élements. Ils appartiennent au type `Bool`.

Haskell
```
λ> maxBound :: Bool
True
λ> minBound :: Bool
False
λ> [False ..]
[False,True]
```

Ce qui est intéressant, c'est les opérations dans `Bool` mais ne vous inquiétez pas, nous allons en reparler bientôt quand nous aurons un peu plus de maturité.
Bon, donnons quand même quelques opérations de base :

Haskell

```
λ> True && False
False
λ> False && False
False
λ> False || True
True
λ> not False
True
λ> 2 + 2 == 4
True
λ> 2 + 2 > 5
False
λ> 2 + 2 /= 5
True
λ> "Iut" == "Enfer"
False
λ> "Haskell" > "C"
True
```

Les caractères

Ils appartiennent au type Char.
Ils sont en nombre fini bien qu'assez nombreux sur Haskell :

Haskell

```
λ> minBound :: Char
'\NUL'
λ> maxBound :: Char
'\1114111'
λ> take 130 [ minBound :: Char .. ]
"\NUL\SOH\STX\ETX\EOT\ENQ\ACK\a\b\t\n\v\f\r\SO\SI\DLE\DC1\DC2\DC3\DC4\NAK\SYN\ETB\CAN
\EM\SUB\ESC\FS\GS\RS\US !\"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\
\]^_`abcdefghijklmnopqrstuvwxyz{|}~\DEL\128\129"
```

Une liste de caractère est une chaîne de caractères et une liste c'est... disons un truc entre des crochets dont les éléments sont ordonnées, peuvent être répétés, peuvent s'imbriquer mais sont tous de même type. Mais c'est beaucoup plus que ça et nous leur consacrerons un chapitre.

Les « Tuples »

Bon, supposons que nous voulions faire un joli dessin en plaçant des lettres de l'alphabet sur une feuille. À chaque lettre on associe un couple de coordonnées entières (on est à l'école primaire). Par exemple, on peut vouloir informer les lecteurs de notre jeu que le point étiqueté par A se trouve au point de coordonnées (1, 2). On peut décider de résumer ça en écrivant :

Haskell

```
λ> let a = ('A',(1 ,2 ))
λ> a
('A',(1,2))
```

Quel est le type de a inféré par Haskell ?

Haskell

```
λ> :t a
a :: (Char, (Integer, Integer))
```

On a parlé tout à l'heure de liste : pourquoi ne met-on pas de crochets alors ?

Haskell

```
λ> let a = ['A',[1 ,2 ]]

<interactive>:117:14-20:
    Couldn't match expected type `Char' with actual type `[t0]'
    In the expression: [1, 2]
```

```
In the expression: ['A', [1, 2]]
In an equation for 'a': a = ['A', [1, 2]]
```

Regardons la définition de match :
match n.

1. i. *One that is exactly like another ; a counterpart.*
   ii. *One that is like another in one or more specified qualities : He is John's match for bravery.*
2. *One that is able to compete equally with another : The boxer had met his match.*
3. i. *One that closely resembles or harmonizes with another : The napkins were a nice match for the tablecloth.*
   ii. *A pair, each one of which resembles or harmonizes with the other : The colors were a close match.*
4. *Sports*
   i. *A game or contest in which two or more persons, animals, or teams oppose and compete with each other : a soccer match.*
   ii. *A tennis contest won by the player or side that wins a specified number of sets, usually two out of three or three out of five.*
5. *A marriage or an arrangement of marriage : a royal match.*
6. *A person viewed as a prospective marriage partner.*

Bref, on ne peut pas marier dans une même liste un caractère avec une liste d'entiers.
Le type communément appelé *tuple* en anglais de spécialité s'appelle un n-uplet en français. Oui, mais qu'est-ce que c'est ?
Prenons un nouvel exemple...

À table

Vous êtes embauché(e) dans un restaurant. Vous disposez de trois ensembles :

— l'ensemble des entrées :

$$E = \left\{\text{Cuisses de sauterelles panées, œuf mou, huîtres de l'Erdre}\right\}$$

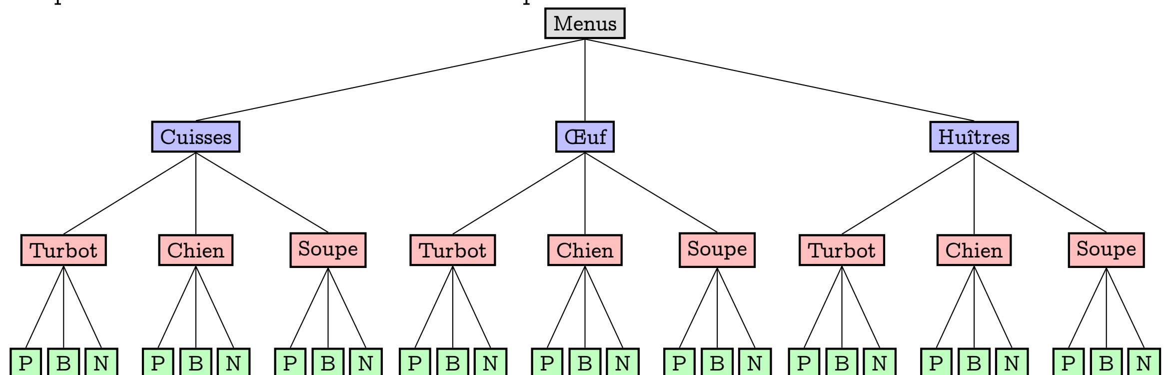
— l'ensemble des plats de résistance :

$$P = \left\{\text{Turbot à l'huile de ricin, Chien à l'andalouse, Soupe d'orties}\right\}$$

— l'ensemble des desserts :

$$D = \left\{\text{Pomme}, \text{Banane}, \text{Noix}\right\}$$

Vous avez envie de créer un nouvel ensemble, celui des menus possibles. Une première idée consiste à regrouper tout le monde en prenant $E \cup P \cup D$.
Je sais bien que la gastronomie n'est pas la principale préoccupation des jeunes au palet perverti par les tristes fastefoudes mais en général on commande UNE entrée SUIVIE D'UN plat de résistance SUIVI D'UN dessert (et pas une pizza et une bière) or la simple union que vous avez proposée peut vous faire choisir un menu totalement différent : cinq desserts et vingt entrées, dans n'importe quel ordre, par exemple.
Nous avons besoin de créer un « objet » ordonné de trois composantes, chacune étant choisie respectivement dans E, P et D.
Par exemple, (œuf mou, chien à l'andalouse, pomme) est un menu. C'est un triplet, à ne pas confondre avec l'ensemble $\left\{\text{œuf mou, chien à l'andalouse, pomme}\right\}$ qui n'est pas ordonné.
On peut avoir une vision de tous les menus possibles avec un arbre :



Pour vous faire plaisir nous allons dresser la liste des menus avec Haskell en introduisant une nouvelle arme : la création de nouveaux types...

Haskell

```haskell
data Entree  = Cuisses | Oeufs | Huitres
               deriving (Enum, Show)
data Plat    = Turbot | Chien | Soupe
               deriving (Enum, Show)
data Dessert = Pomme | Banane | Noix
               deriving (Enum, Show)

lesEntrees  = [Cuisses .. ]
lesPlats    = [Turbot .. ]
lesDesserts = [Pomme .. ]

lesMenus = [(e,p,d) | e←lesEntrees, p←lesPlats, d←lesDesserts]
```

(*Ici, des commentaires éclairés de votre professeur adoré.*)
Et dressons la liste des menus :

Haskell

```haskell
λ> lesMenus
[(Cuisses,Turbot,Pomme),(Cuisses,Turbot,Banane),(Cuisses,Turbot,Noix),
 (Cuisses,Chien,Pomme),(Cuisses,Chien,Banane),(Cuisses,Chien,Noix),
 (Cuisses,Soupe,Pomme),(Cuisses,Soupe,Banane),(Cuisses,Soupe,Noix),
 (Oeufs,Turbot,Pomme),(Oeufs,Turbot,Banane),(Oeufs,Turbot,Noix),
 (Oeufs,Chien,Pomme),(Oeufs,Chien,Banane),(Oeufs,Chien,Noix),
 (Oeufs,Soupe,Pomme),(Oeufs,Soupe,Banane),(Oeufs,Soupe,Noix),
 (Huitres,Turbot,Pomme),(Huitres,Turbot,Banane),(Huitres,Turbot,Noix),
 (Huitres,Chien,Pomme),(Huitres,Chien,Banane),(Huitres,Chien,Noix),
 (Huitres,Soupe,Pomme),(Huitres,Soupe,Banane),(Huitres,Soupe,Noix) ]
λ> length lesMenus
27
λ> :t lesMenus
lesMenus :: [(Entree, Plat, Dessert)]
```

(*Ici, des commentaires éclairés de votre professeur adoré.*)

Paire ordonnée - Produit de deux ensembles

Comment créer de l'ordre à partir du désordre...

Définition 3 - 4

Paire ordonnée
On note $\langle a, b\rangle = \{\{a\}, \{a, b\}\}$

On remarque tout de suite que les rôles de a et b ne sont pas symétriques.
L'égalité des ensembles va nous permettre de définir une égalité des paires : ce n'est pas la même ! Un(e) informathicien(ne) doit bien s'en rendre compte...
Démontrez tout d'abord que :

Théorème 3 - 2

$$(\{a, b\} =_{\text{ens}} \{x, y\}) \equiv ((a = x \wedge b = y) \vee (a = y \wedge b = x))$$

puis que

Théorème 3 - 3

$$(\langle a, b\rangle =_{\text{paire}} \langle x, y\rangle) \equiv (a = x \wedge b = y)$$

On peut alors définir le produit (cartésien) de deux ensembles comme étant l'ensemble des paires formées d'éléments de ces ensembles...

Définition 3 - 5

$$A \otimes B = \{(a, b) \mid (a \in A) \wedge (b \in B)\}$$

Pour se représenter le produit cartésien $E \otimes F$ avec $E = \{a, b, c\}$ et $F = \{1, 2, 3, 4, 5\}$, on peut évidemment l'écrire en extension : il possède $3 \times 5 = 15$ éléments (voir à ce sujet le paragraphe suivant).

$$E \otimes F = \{(a,1), (a,2), \cdots, (c,5)\}$$

mais il est souvent préférable de faire appel à un tableau du type suivant :

$\otimes$	1	2	3	4	5
a	$(a,1)$	$(a,2)$	$(a,3)$	$(a,4)$	$(a,5)$
b	$(b,2)$	$(b,2)$	$(b,3)$	$(b,4)$	$(b,5)$
c	$(c,1)$	$(c,2)$	$(c,3)$	$(c,4)$	$(c,5)$

Avec Haskell :

Haskell

```
λ> [(x,y) | x←['a','b','c'], y←[1,2,3,4,5]]
[('a',1),('a',2),('a',3),('a',4),('a',5),
 ('b',1),('b',2),('b',3),('b',4),('b',5),
 ('c',1),('c',2),('c',3),('c',4),('c',5)]

λ> [(x,y) | x←['a','b','c'], y←[1,2,3,4,5], mod y 2 == 0]
[('a',2),('a',4),('b',2),('b',4),('c',2),('c',4)]
```

Mais comment faire avec notre menu qui a trois plats ordonnés ?

n-uplets - Produit d'un nombre quelconque d'ensembles

On peut dire qu'un menu est un couple formé d'une entrée et d'un couple plat de résistance-dessert...
Définissons temporairement un triplet ordonné $\langle a, b, c\rangle$ par : $\langle a, b, c\rangle = \langle a, \langle b, c\rangle\rangle$.

Recherche

Est-ce que c'est licite ? C'est-à-dire est-ce que c'est une définition...qui définit vraiment un triplet ordonné ? Qu'est-ce que ça veut dire en fait « définit vraiment un triplet ordonné » ?
Comment va-t-on faire alors pour un quadruplet ? Un n-uplet ?
Qu'est-ce que c'est que ça : $E_1 \otimes E_2 \otimes \cdots \otimes E_n = \prod_{i=1}^{n} E_i$?

Deux fonctions utiles :

Définition 3 - 6

Projections
On note π_1 (π_2) la fonction définie sur un produit $E \otimes F$ qui a un couple associe sa première (deuxième) composante

Sur Haskell, c'est **`fst`** et **`snd`** :

Haskell

```
λ> fst ('a','b')
'a'

λ> snd ('a','b')
'b'
```

Aparté

Contrairement à votre chambre, sur machine, il est beaucoup plus compliqué de créer du désordre que de l'ordre.
Nous suivrons donc le cheminement inverse de ce qui vient d'être fait : nous partirons de n-uplets ordonnés et nous créerons des ensembles non ordonnés...

2 Notion de cardinal

Le cardinal d'un ensemble, c'est en gros le nombre de ses éléments. La notion de cardinal est intimement liée à la notion de fonction et à l'ensemble $\mathbb{N}$ des entiers naturels que nous détaillerons plus tard. Nous nous contenterons dans un premier temps d'une approche intuitive. Le cardinal de E est indifféremment noté :Card(E) ou $|E|$ ou $\#E$

Par exemple $|\varnothing| = 0$, cette notation ne devra bien sûr pas être confondue avec la valeur absolue d'un réel ou bien le module d'un nombre complexe. Quoi qu'il en soit, le contexte permettra généralement de lever l'ambiguïté.

Nous admettons les résulats suivants qui ne concernent que des ensembles **finis** : il faut en effet savoir que les ensembles infinis ont aussi un cardinal, ce qu'a introduit CANTOR à la fin du XIXe siècle...Cela l'a d'ailleurs rendu fou....

Propriétés 3 - 1

— Si $A \subseteq B$ alors $|A| \leqslant |B|$. Une conséquence intéressante est celle ci : si $A \subseteq B$ avec $|A| = |B|$ alors $A = B$.

— $|A - B| \leqslant |A|$.

— $|A \cup B| = |A| + |B| - |A \cap B|$

— **Si on a** $E_i \cap E_j = \varnothing$ **lorsque** $i \neq j$, alors $\left|\bigcup_{i=1}^{n} E_i\right| = \sum_{i=1}^{n} |E_i|$.

— $|E_1 \otimes E_2| = |E_1|.|E_2|$ et

$$\begin{aligned} |E_1 \otimes E_2 \otimes \cdots \otimes E_n| &= |E_1|.|E_2|.\cdots.|E_n| \\ |E^n| &= \left(|E|\right)^n \end{aligned}$$

Nous la démontrerons en exercice.

RECHERCHES

Recherche 3 - 1

Écrivez en compréhension l'ensemble $\mathbb{D}$ des décimaux, l'ensemble $\mathbb{Q}$ des rationnels, l'ensemble $\mathbb{C}$ des complexes. L'ensemble des réels $\mathbb{R}$ est un peu plus délicat à définir malgré son nom...

Recherche 3 - 2

L'ensemble universel est $S = \{1,2,3,4,5\}$. Pour chacune des propositions suivantes, donnez-en la négation puis la valeur de vérité.

1. $(\exists x)(x+3=10)$
2. $(\forall x)(x+3<10)$
3. $(\exists x)(x+3<5)$
4. $(\forall x)(x+3\leqslant 7)$

Recherche 3 - 3

Parmi les ensembles suivants, quels sont ceux qui sont égaux ?

1. $A = \{x \mid x \in \mathbb{R} \text{ et } x^2-4x+3=0\}$
2. $B = \{x \mid x \in \mathbb{R} \text{ et } x^2-3x+2=0\}$
3. $C = \{x \mid x \in \mathbb{N} \text{ et } x<3\}$
4. $D = \{x \mid x \in \mathbb{N} \text{ et } x<5 \text{ et } x \text{ est impair }\}$
5. $E = \{1,2\}$
6. $F = \{1,2,1\}$
7. $G = \{3,1\}$
8. $H = \{1,1,3\}$

Recherche 3 - 4

$E = \{0,1,2,3,4,5,6\}$. Définir en extension les ensembles suivants :

1. $A_1 = \{x \in \mathbb{N} \mid x^2 \in E\}$
2. $A_2 = \{x \in \mathbb{R} \mid x^2 \in E\}$
3. $A_3 = \{x \in E \mid x^2 \in E\}$
4. $A_4 = \{x \in E \mid \sqrt{x} \in E\}$
5. $A_5 = \{x \in E \mid 2x \in E\}$
6. $A_6 = \{x \in E \mid \frac{x}{2} \in E\}$

Recherche 3 - 5

$E = \{0,1,2,3,4\}$. Compléter, lorsque c'est possible, par un des symboles (il peut y avoir plusieurs solutions, mais on s'obligera à choisir celle qui donne le plus « de renseignements ») :

$$\in, \ni, \subseteq, \supseteq, =, \neq, \subsetneq, \notin, \cdots$$

1. $2 \cdots E$,
2. $\{2,3\} \cdots E$,
3. $\{2\} \cdots E$,
4. $\{2,3,4\} \cdots \{4,3,2\}$,
5. $\{2,3,4\} \cdots \{4,3,0\}$,
6. $\{\} \cdots E$,
7. $E \cdots E$,
8. $E \cdots \{E\}$,
9. $\{\} \cdots \{E\}$,
10. $E \cdots \{0,1,2,3,\ldots,10\}$,
11. $\{0,1,2,3,4,5\} \cdots E$

Recherche 3 - 6

Soit $A = \{1,2,\ldots,8,9\}$, $B = \{2,4,6,8\}$, $C = \{1,3,5,7,9\}$, $D = \{3,4,5\}$, $E = \{3,5\}$ et $\Omega = \{A,B,C,D,E\}$.
Écrivez les ensembles suivants par compréhension puis déterminez les éléments de Ω qui vérifient les conditions suivantes :

1. $X \cup B = \varnothing$;
2. $X \subseteq D \wedge X \nsubseteq B$;
3. $X \subseteq A \wedge X \nsubseteq C$;
4. $X \subseteq C \wedge X \nsubseteq A$.

Recherche 3 - 7

Trouvez des analogies et des différences entre $\subseteq$ définie sur les ensembles et $\leqslant$ définie sur les réels.

Recherche 3 - 8

Essayez d'expliquer ce qui se passe :

Haskell

```haskell
λ> elem 'Z' "Zorglub"
True
λ> 'Z' `elem` "Zorglub"
True
λ> elem 'Z' "Bulgroz"
False
λ> elem "Zor" "Zorglub"

<interactive>:46:12-20:
    Couldn't match type 'Char' with '[Char]'
    Expected type: [[Char]]
```

```
      Actual type: [Char]
    In the second argument of ‘elem’, namely ‘"Zorglub"’
    In the expression: elem "Zor" "Zorglub"
    In an equation for ‘it’: it = elem "Zor" "Zorglub"
λ> all (‘elem‘ "Zorglub") "Zorba"
False
λ> any (‘elem‘ "Zorglub") "Zorba"
True
```

Recherche 3 - 9 Complément à 2 puissance n

Le monde se sépare en 10 catégories : ceux qui comprennent cette phrase et les autres.

Zorglub in « Les aventures de Zorglub et Martine au Far-West »

Nous allons, pour favoriser la compréhension et simplifier les écritures, travailler sur 8 bits. Cela signifie que nos nombres entiers peuvent être codés avec 8 chiffres égaux à 0 ou 1 en base 2.

1. Écrivez les nombres de 0 à 17 en base 2. Quel est le plus grand nombre que l'on peut écrire avec 8 bits ?
2. En fait, on aimerait avoir des entiers signés, c'est-à-dire pouvoir représenter des entiers tant positifs que négatifs avec cette contrainte.

   On va donc réserver un bit (le plus à gauche) pour indiquer le signe du nombre : on mettra un 0 s'il est positif et un 1 s'il est négatif.

   Écrivez en compréhension l'ensemble des nombres que l'on peut ainsi représenter et qui se note sur Haskell `Int8`.
3. Bon, pas de problème...enfin si quand même un peu. Que pensez-vous de ces deux nombres :

   ```
   00000000 10000000
   ```
4. On voudrait additionner facilement ces nombres. Une idée serait d'utiliser l'addition posée comme à l'école primaire. Par exemple pour calculer $5+9$

   ```
        00000101
   +    00001001
   ------------
        00001110
   ```

   On trouve bien 14.

   Et maintenant si on veut calculer $5+(-9)$, qu'est-ce que cela donne avec ce système ?
5. Considérez maintenant le nombre `m = 11111111` et ajoutez-lui 1 (vous n'avez que 8 bits à disposition). Que vaut $m+1$? Que cela vous donne-t-il comme idée ?

   Considérez n=11111110 et calculez $n+2$.

   Considérez p=11111100 et calculez $n+4$.

   Considérez p=11111000 et calculez $n+8$.

   Considérez p=11111010 et calculez $n+6$.

   Des idées ?
6. Si l'on considère que 2^8 vaut 0 en `Int8`, ce qui est d'ailleurs le cas :

   Haskell
   ```
   λ> let n = 256 :: Int8
   λ> n
   0
   ```

   Que pensez-vous alors de x et de $2^8-|x|$ si x est négatif ?

   Comment s'écrit $2^8-|x|$ sur 8 bits ?

   Bon, vous commencez à voir ce qui se passe ?

   Alors expliquez ce qui se passe ici :

   Haskell
   ```
   λ> let n = 13 :: Int8
   λ> n * 2
   26
   ```

```
λ> n * 4
52
λ> n * 8
104
λ> n * 16
-48
λ> n * 32
-96
λ> n * 64
64
```

Généralisez, énoncez votre norme de représentation des entiers sur 8 bits, énoncez un « truc » pour trouver facilement de tête le complément à 2^8.

Recherche 3 - 10

1. Soit $A = \{0, 1\}$. Déterminer $\left(A^2 \setminus \{(0,0)\}\right) \otimes A$.
2. Soit $A = \{3, 5, 7\}$ et $B = \{a, b\}$. Déterminer A^2, B^2, $A \otimes B$, $B \otimes A$, $B^2 \otimes A$.

Recherche 3 - 11 liste en compréhension

Lire avec attention http://lyah.haskell.fr/demarrons#je-suis-une-liste-en-comprehension
puis répondez aux petites questions posées là : http://informathix.tuxfamily.org/?q=node/89

THÈME 4

Let's have some fun

1 C'est quoi une fonction ?

Langage fonctionnel, fonction ? Est-ce que vous savez vraiment ce qu'est une fonction ?

LISP • APL • FP • Scheme • KRC • Hope
Miranda™• Erlang • Curry • Gofer • Mercury
Charity • Cayenne • Mondrian • Epigram
SML • Clean • Caml • Haskell

Everything else is just
dysfunctional
programming!

Pour définir une fonction, il faut un ensemble de départ (un type quoi) et un ensemble d'arrivée (le même type ou un autre). Une fonction associe à chaque élément du type de départ AU PLUS UN ÉLÉMENT du type d'arrivée.
En fait, une fonction c'est une *flèche* entre deux types

Exemple

Un exemple de fonction

Prenons un autre exemple : la fonction `not` qui a un booléen associe l'autre.
Comme toute chose en Haskell, `not` a un type :

Haskell

```
λ> :t not
not :: Bool→ Bool
```

`not` a pour type `Bool` flèche `Bool` : c'est une fonction qui a un booléen associe un autre booléen.
Pour les fonctions, on dit plutôt que `Bool→ Bool` est la *signature* de type de la fonction `not`.
On ne sait pas encore comment elle calcule les images mais ça ne doit pas être compliqué :

Haskell

```
λ> not False
True
λ> not True
False
```

Alors on va essayer de le faire nous-mêmes : pourquoi toujours parler anglais !

Haskell
```haskell
data Booleen = Faux | Vrai

non :: Booleen → Booleen
non    Vrai     =  Faux
non    Faux     =  Vrai
```

On donne l'image de chaque élément du type de départ. Il n'y a qu'une seule image appartenant au type d'arrivée.
Bon, c'est un peu limité comme fonction mais nous l'avons fabriqué entièrement : son ensemble de départ, son ensemble d'arrivée et son *graphe*.

Définition 4 - 1

Graphe d'une fonction
Le graphe d'une fonction est l'ensemble de tous les couples :
⟨ élément du départ, son image dans l'arrivée ⟩
ou si vous préférez :

$$\mathcal{G}(f) = \{\langle x, f(x)\rangle \mid x \in \text{départ}\}$$

et voilà notre fonction bien définie. En maths on écrirait :

$$f = \langle Booleen, Booleen, \{\langle Vrai, Faux\rangle, \langle Faux, Vrai\rangle\}\rangle$$

À retenir

Une fonction est donc un triplet ⟨ Départ, Arrivée, Graphe⟩

Danger

Ne pas oublier !
Une fonction est donc un TRIPLET ⟨ Départ, Arrivée, Graphe⟩

Danger

Ne pas oublier !
Le graphe ne suffit pas à définir une fonction : il faut préciser son type de départ et son type d'arrivée.

2 Une fonction c'est more fun

Bon, OK, maintenant, que se passe-t-il si j'ai plusieurs arguments ? Par exemple, le « et » logique qu'on appelle plutôt *conjonction*.

On réfléchit d'abord sans clavier. Le premier booléen et le deuxième booléen sont vrais si, et seulement si, ils sont vrais tous les deux, ce qui signifie que si l'on appelle « et » notre fonction, alors

> et x y vaut :
> si x = Vrai et y = Vrai alors Vrai
> dans tous les autres cas Faux

ce qui s'écrit plus joliment en Haskell :

Haskell
```haskell
et :: Booleen→ Booleen→ Booleen
et    Vrai       Vrai     =  Vrai
et    _          _        =  Faux
```

Le _ signifiant : « n'importe quelle valeur autre que celle(s) déjà étudiée(s) ».
Vraiment n'importe laquelle ? Non, comme on a signé la fonction, on doit rester dans les types annoncés.
On dit qu'on a défini la fonction par *filtrage par motif* (*pattern matching in anglais of spécialité.*)
On teste ça dans GHCi :

Haskell
```
λ> :l ./introHs
[1 of 1] Compiling Main             ( introHs.hs, interpreted )
Ok, modules loaded: Main.
λ> et Vrai Faux

<interactive>:16:1-12:
    No instance for (Show Booleen) arising from a use of ‘print’
    Possible fix: add an instance declaration for (Show Booleen)
    In a stmt of an interactive GHCi command: print it
```

Hein ? Bon, on a pas encore étudié les classes de types mais on comprend à peu près que Haskell ne sait pas comment afficher nos Booleen. On va juste lui dire de les afficher tels quels à l'aide de la fonction deriving que nous étudierons plus en détail plus tard.
On ajoute juste une ligne à notre fichier hs :

Haskell
```haskell
data Booleen = Faux | Vrai
             deriving Show
```

on sauve et on recharge dans GHCi :

Haskell
```
λ> :r
[1 of 1] Compiling Main             ( introHs.hs, interpreted )
Ok, modules loaded: Main.
λ> et Vrai Faux
Faux
```

On aurait pu remarquer une autre chose : si le premier argument est Vrai alors et Vrai x = x et si le premier argument est Faux, alors et Faux x = Faux.
On peut donc définir également notre fonction ainsi :

Haskell
```haskell
et' :: Booleen→ Booleen→ Booleen
et'    Vrai       x        =  x
et'    Faux       x        =  Faux
```

Ces fonctions sont-elles bien égales ? Mais ça veut dire quoi que deux fonctions sont égales ? (*répondez vite ou bien j'appelle Conan*).
Il faut regarder leurs types de départ et d'arrivée...mais ! ? ? Il y a trois types et deux flèches ! ! !

Curryfication

Calmons-nous. N'oubliez pas : la difficulté en informatique vient souvent de la surabondance de solutions à nos problèmes. Il y a le plus souvent beaucoup de manières de modéliser une situation. Il faut faire un (parfois le) bon choix.

Un habitué de C, Java ou Python pourrait définir la fonction « et » à l'aide de ce code spaghetti :

```
Fonction et(x,y:booléens):booléen
Début
    Si x == Vrai Alors
        Retourner y
    Sinon
        Retourner Faux
    FinSi
Fin
```

On a donc une fonction de deux variables. Un mathématicien obtus ou un adepte des bases de données pourrait alors définir sa fonction de la même manière (car il se souvient de la notion de *produit cartésien* défini au chapitre précédent) :

$$\text{et}: \begin{array}{rcl} \texttt{Booleen} \times \texttt{Booleen} & \to & \texttt{Booleen} \\ (x,y) & \mapsto & \begin{cases} \text{si } x == \text{Vrai alors } y \\ \text{sinon Faux} \end{cases} \end{array}$$

Bref, la fonction « et » serait une fonction de 2 variables. Mais il y a un problème en Haskell ...

À retenir Sur Haskell , toutes les fonctions sont d'UNE SEULE VARIABLE !

Ouvrez bien les yeux ! La fonction définie juste avant a UNE seule flèche : c'est donc bien UNE fonction.
Les fonctions définies sur Haskell comportaient DEUX flèches dans leur signature donc derrière ces fonctions se cachent en fait DEUX fonctions car une flèche = une fonction.
C'est qu'en fait, la signature de notre fonction « et2 » peut aussi s'écrire

Haskell
```haskell
et2 :: Booleen-> (Booleen-> Booleen)
et2    Vrai      x      =  x
et2    Faux      x      =  Faux
```

Ainsi « et2 » est en fait UNE fonction qui à un `Booleen` associe UNE AUTRE fonction. GHCi nous le confirme :

Haskell
```haskell
λ> :t et2 Vrai
et2 Vrai :: Booleen-> Booleen
```

Licence Creative Commons

« `et2 Vrai` » est une fonction qui à un `Booleen` associe un autre `Booleen`.
On omet le plus souvent les parenthèses de la signature car par défaut, la → de la signature est *associative à droite* sur Haskell .
On pourrait donc créer deux fonction intermédiaires et une fonction principale :

Haskell
```
idem :: Booleen→ Booleen
idem    x       =  x

tjsFaux :: Booleen→ Booleen
tjsFaux    x         = Faux

et4 :: Booleen→ Booleen→ Booleen
et4    Vrai     =  idem
et4    Faux     =  tjsFaux
```

Cela nous permet d'utiliser `idem` ou `tjsFaux` dans d'autres contextes.

Toujours plus fort : les lambda-expressions

Nous allons enfin découvrir pourquoi le symbole de Haskell contient un λ.....

Une fonction est une flèche. Elle n'a pas forcément besoin de porter un nom. On peut définir une fonction sans la nommer à l'aide de \ (qui ressemble à un λ...)

Haskell
```
λ> ((λx→ 2*x) :: (Int→ Int)) 5
10
```

ou plus simplement :

Haskell
```
λ> (λx→ 2*x) 5
10
```

Aparté

Petite histoire du λ
En 1936, Alan Turing et Alonzo Church écrivent les fondements de ce qui deviendra l'informatique. Church introduit une notation pour définir des fonctions qui était au départ une sorte d'accent circonflexe qui, après le passage de plusieurs typographes s'est transformé en Λ (lambda majuscule) puis en λ.

Cette possibilité est reprise par exemple en python :

Python
```
In [1]: (lambda x: 2*x)(5)
Out[1]: 10
```

Pour notre problème initial, cela nous permet d'écrire aussi :

Haskell
```
et4 :: Booleen→ Booleen→ Booleen
et4    Vrai     =  λx→ x
et4    Faux     =  λx→ Faux
```

Gardes ou expressions conditionnelles ?

OK, on a plein de possibilités, mais est-ce qu'on peut quand même implémenter tel quel l'algo spaghetti à la mode C/Java/Python & Co ?

Haskell
```
et5 :: Booleen→ Booleen→ Booleen
et5 y x = if y == Vrai
          then x
          else Faux
```

Une petite remarque sur le `if ... then ... else` : comme toute chose en programmation fonctionnelle, il s'agit d'une expression qui doit donc retourner une valeur.
Il ne faut donc surtout pas oublier le `else` comme on peut le faire dans les langages impératifs. C'est une sécurité supplémentaire...
« Si j'ai un parachute alors je peux sauter de l'avion » : c'est raisonnable, mais que se passe-t-il si je n'ai pas de parachute : je saute quand même ? Je tue le(a) pilote ? J'appelle Superman ? Je séduis l'hôtesse de l'air (ou le steward) ?
On évitera cependant au maximum le recours aux expressions conditionnelles car elles peuvent ralentir l'évaluation répétée de ceratines fonctions mais nous en reparlerons plus tard.
Il existe une syntaxe alternative plus fonctionnelle : l'emploi de *gardes* qui permet de calculer une image selon qu'une ou plusieurs valeurs vérifie une certaine propriété. Cela ressemble à une expression conditionnelle mais permet d'enchaîner autant de conditions que désirées.
Ça s'appelle un garde car ça garde l'évaluation : il faut montrer patte blanche.
Par exemple :

Haskell
```
placeDeCinema :: Int→ String
placeDeCinema age
      | age < 0    = "Euh, t'as bu quoi au petit déjeuner?"
      | age < 12   = "Pour toi c'est 2 euros et viens prendre un bonbon"
      | age < 18   = "Pour toi c'est 8 euros"
      | otherwise  = "T'es pas un peu vieux pour aller voir Petit Poney?"
```

Le compilateur visite les gardes de haut en bas.
En ce qui concerne notre problème, c'est assez simple :

Haskell
```
et6 :: Booleen→ Booleen→ Booleen
et6 x y
     |x == Vrai = x
     |otherwise = Faux
```

En fait, `otherwise` est un synonyme de `True` mais facilite la lecture. C'est un garde qui laisse passer tout le monde c'est pourquoi on le laisse pour la fin...

3 En résumé, comment construire nos fonctions ?

Peu importe la syntaxe, voici les étapes importantes à suivre :

SPÉCIFIER la fonction : c'est-à-dire parfois (on peut effectivement s'en passer au besoin) la nommer, donner son *type*, i.e. son domaine et son codomaine : on parle de *signature de type* et on explique ce qu'elle fait. Par exemple pour le double d'un entier sur Haskell :

Haskell
```
double :: Int→ Int
-- calcule le double d'un entier : le résultat est un entier
```

Sur Haskell , donner la signature est optionnel car le compilateur l'*infère* mais c'est une bonne pratique de l'annoncer à l'avance pour plus de sécurité. Si le type inféré par le

compilateur est différent de celui spécifié, c'est qu'il faut revoir sa copie...Cela rend de plus le code plus lisible.
On peut obtenir le type d'une expression avec la commande **:t** :

Haskell
```
λ> :t double
double :: Int→ Int
```

RÉALISER la fonction c'est associer à la fonction spécifiée une expression. Dans notre exemple, il s'agit de multiplier l'argument de la fonction par 2. Pour cela, on utilise des *paramètres formels* qui font *abstraction* de la valeur particulière du nombre dont on veut connaître le double. Ici :

Haskell
```
double n = 2 * n
```

UTILISER la fonction dans une expression en lui donnant des *paramètres effectifs* (arguments), c'est-à-dire liés à l'application particulière de la fonction :

Haskell
```
λ> double 3 + double 7
20
```

Il faut bien noter que les étapes de réalisation et d'utilisation sont en quelque sorte indépendantes, contrairement au paradigme impératif.
La réalisation *fait abstraction* de l'utilisation : l'expression fournie réalisera la spécification quelque soit le contexte.
L'utilisation *fait abstraction* de la réalisation : si on suit la spécification, on aura ce qu'on veut dans tous les cas.

4 Fonction récursives : première approche

Les mains dans le cambouis

Nous allons étudier le plus classique des problèmes : il s'agit de calculer la somme des entiers de 1 à un entier naturel n non nul donné.
La *spécification* est immédiate : on crée une fonction ayant pour argument l'entier naturet n et renvoyant l'entier naturel correspondant à la somme des entiers de 1 à n.
Il reste à s'occuper de la *réalisation*.
Appelons $S(n)$ la somme des entiers de 1 à n avec $n \in \mathbb{N} \setminus \{0\}$.
On a $S(1) = 1$ et $S(n) = n + S(n-1)$.

Haskell
```
s1 :: Int→ Int
s1 n
     | n == 1    = 1
     | otherwise = n + s1 (n -1)
```

Et si n est négatif ? Le type de départ ne l'empêche pas. Nous devons donc inclure une précaution supplémentaire :

Haskell
```
s1 :: Int→ Int
s1 n
     | n ≤ 0     = error "l'argument doit être un entier naturel non nul"
     | n == 1    = 1
     | otherwise = n + s1 (n -1)
```

Un dernier détail : à chaque étape, on testera si n est négatif ce qui est inutile car n ne peut être négatif que la première fois.

Haskell

```haskell
s1 :: Integer→ Integer
s1 n | n ≤ 0     = error "l'argument doit être un entier naturel non nul"
     |otherwise  = aux n
    where aux 1 = 1
          aux n = n + aux (n - 1)
```

Que se passe-t-il quand on calcule `s1(5)` ? On utilise le modèle de *substitution* :

```
s1(5)
5 + s1(4)
5 + (4 + s1(3))
5 + (4 + (3 + s1(2)))
5 + (4 + (3 + (2 + s1(1))))
5 + (4 + (3 + (2 + 1)))
5 + (4 + (3 + 3))
5 + (4 + 6)
5 + 10
15
```

On commence par quatre expansions suivies de cinq réductions.
La *procédure* est récursive car syntaxiquement `s1` apparaît dans sa propre définition (`n + s1 (n - 1)`).
Le *processus* est également récursif : les évaluations sont retardées. Intuitivement, cela est le cas car on progresse du cas compliqué vers le cas simple : on « descend » ici de n vers 1. Le compilateur doit *empiler* les résultats des expansions tant qu'il ne peut pas les réduire. La taille de la pile nécessaire étant proportionnelle à l'argument n, on dit qu'il s'agit d'un processus linéairement récursif.
Essayons de calculer la somme des entiers de 1 à 1 000 000 :

Haskell

```haskell
main = print (s1 1000000)
```

Shell

```
$ ghc ./introHs.hs -o som1

$ time ./som1
Stack space overflow: current size 8388608 bytes.
Use '+RTS -Ksize -RTS' to increase it.

real    0m0.067s
user    0m0.063s
sys     0m0.004s
```

Le problème, c'est que l'opérateur est strict sur ses deux arguments : cela veut dire que les deux arguments doivent être évalués avant que la somme puisse renvoyer une valeur.
L'idée est d'obtenir :
`(((1 + 2) + 3) + 4) + 5`
qui peut être évalué immédiatement au lieu de :
`5 + (4 + (3 + (2 + 1)))`
qui en fait empile 5, 4, 3 et 2 avant de commencé à réduire.
Il s'agit de partir du cas simple et de « monter » vers le cas compliqué (on parle dans ce cas habituellement d'*induction*).
Ici, la somme vaut 0 puis on ajoute 1, puis on ajoute au résultat 2, puis on ajoute au résultat 3, etc jusqu'à atteindre n.
Techniquement, on entretient un *accumulateur* de termes qui évolue et un compteur qui va évoluer de 1 jusque n.
L'accumulateur et le compteur évoluent selon la règle :

```
accumulateur ← accumulateur + compteur
compteur ← compteur + 1
```

la somme cherchée étant la valeur de l'accumulateur lorsque le compteur a dépassé n.
On essaie ça sur Haskell :

Haskell
```haskell
s2 :: Integer→ Integer
s2 n =
    let sIter acc compt
            | compt ≤ 0 = acc
            | otherwise  = sIter (acc + compt) (compt -1)
    in sIter 0 n
```

Ça va marcher :

Shell
```
$ time ./som2
Stack space overflow: current size 8388608 bytes.
Use '+RTS -Ksize -RTS' to increase it.

real    0m0.177s
user    0m0.149s
sys     0m0.028s
```

Arghh !
Cela aurait dû marcher avec la plupart des langages fonctionnels ou du moins optimisant ce qu'on appelle la *récursion terminale* (tail recursion) C'est d'ailleurs ce que fait OCaml :

OCaml
```ocaml
let rec suma = function
      |1 → 1
      |n → n + (suma (n - 1));;

let sumb n =
  let rec siter acc compt =
    if compt  ≤  0 then acc
    else siter (acc + compt) (compt - 1)
  in siter 0 n;;
```

qui donne :

OCaml
```
utop[1]> suma 1_000_000;;
Stack overflow during evaluation (looping recursion?).
utop[2]> sumb 100_000_000;;
- : int = 5000000050000000
```

Sur Ocaml par exemple, la procédure `siter` est syntaxiquement récursive car elle est définie à partir d'elle-même mais le processus n'est plus récursif ! les évaluations ne sont plus retardées :

```
sumb 5
siter 0 1
siter 1 2
siter 3 3
siter 6 4
siter 10 5
siter 15 6
15
```

Cette fois-ci, il n'y a plus de phase d'expansion suivie d'une phase de réduction. À chaque étape, il suffit de garder une trace des états de `acc` et `compt`. On parle alors de processus *itératif*.

Mais Haskell a une autre particularité : il *évalue paresseusement*. C'est une notion un peu compliquée. Disons que le compilateur n'évalue une expression que quand il est « forcé » de le faire. Nous en reparlerons plus tard éventuellement.

Il existe un moyen de forcer le compilateur à travailler : c'est la fonction `seq`. Mais elle est dangereuse en Haskell pour des raisons assez complexes... En gros, `seq x f(x)` va forcer l'évaluation de `x` avant de s'occuper de f(x).
Ici :

Haskell
```haskell
s3 n =
    let sIter acc 0      = acc
        sIter acc compt =
            let newAcc = compt + acc
            in seq newAcc (sIter newAcc (compt - 1))
    in sIter 0 n
```

Alors :

Shell
```
$ time ./som3
500000500000

real    0m0.078s
user    0m0.075s
sys     0m0.004s
```

Restons propres

Un peu technique hein ? Bon, nous allons dans un premier temps ne pas tenir compte de toutes ces subtilités.
On se contentera de créer des fonctions de ce type :

Haskell
```haskell
s1 :: Int→ Int
s1 n
    | n ≤ 0     = error "l'argument doit être un entier naturel non nul"
    | n == 1    = 1
    | otherwise = n + s1 (n -1)
```

ou mieux :

Haskell
```haskell
s4 n = sum [1 .. n]
```

ou à la rigueur :

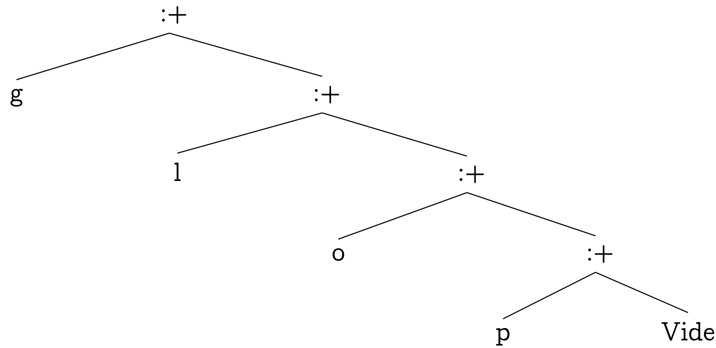
Haskell
```haskell
s5 n = foldl (+) 0 [1 .. n]
```

mais cette fonction `foldl` est encore mystérieuse et sera étudiée en temps voulu.
Mais vous aurez noté l'emploi de listes pour régler ce problème de somme d'entiers. C'est que la structure de base de la programmation fonctionnelle est justement la liste.

5 Une première approche des listes sur Haskell

Définition récursive d'un type

On ne dispose souvent que de quelques types de base : entiers, flottants, caractères, booléens.
On a pourtant besoin de types plus sophistiqués.
Par exemple, qu'est-ce qu'un mot ? Disons que c'est soit le vide, soit un caractère suivi d'un mot. Cela nous donne une règle de *construction* de proche en proche : on ajoute une à une dans l'ordre les lettres du mot pour le construire.
La structure attachée est celle d'un arbre. Voici celui correspondant au mot « glop ».



On choisit un *opérateur infixe* **:+** pour noter le *constructeur* par ajout d'un caractère à gauche. Comme il faut bien partir de quelque chose, on crée aussi le constructeur **Vide** qui construit un mot vide.
Sur Haskell , ce genre de type se crée simplement avec **data** après avoir créé un opérateur infixe avec la fonction **infixr** :

Haskell
```
infixr 1 :+

data Mot =
  Vide
  | Char :+ Mot
```

Le « r » après **infix** indique que l'opérateur est associatif à droite :
'a' :+ ('b' :+ ('c' :+ Vide)) == 'a' :+ 'b' :+ 'c' :+ Vide
Le **1** indique le niveau de priorité sur les autres opérateurs. Par exemple, le niveau de ***** est 7 alors que celui de **+** est 6.
Le **|** indique l'alternative.
On va rajouter un petit sucre dont on reparlera plus tard : la dérivation. Pour pouvoir afficher nos objets dans l'interpréteur, nous allons faire en sorte que notre type **Mot** fasse partie de la classe **Show** :

Haskell
```
data Mot =
  Vide
  | Char :+ Mot
  deriving (Show)
```

On peut maintenant créer un mot :

Haskell
```
λ> let m = 'a' :+ 'b' :+ Vide
λ> m
'a' :+ ('b' :+ Vide)
λ> :t m
m :: Mot
```

Pour pouvoir travailler sur ces « Mots », en plus des constructeurs, on a besoin de *sélecteurs* qui vont permettre de désigner les éléments qui ont permis de construire le mot. Les sélecteurs permettent en fait de *découper* les objets construits.
Ici, on construit des arbres par la gauche : nous avons besoin de distinguer la *tête* du mot qui est son premier caractère, de sa *queue* qui est le mot privé de sa tête. Attention ! Un mot vide n'a ni queue ni tête...
Par exemple, pour la tête :

Haskell
```
tete :: Mot→ Char
-- renvoie le premier caractère d'un mot avec un filtrage par motif
tete Vide = error "Mot vide !"
-- un mot vide n'a pas de tête
tete (t :+ q) = t
```

Recherche

Déterminez une fonction **queue :: Mot -> Mot** qui renvoie la queue d'un mot.

Par exemple :

Haskell
```
λ> tete m
'a'
```

On pourrait avoir besoin de *testeurs* qui nous permet de savoir comment a été construit l'objet. Ici, ce serait un test **estVide** mais on peut s'en passer grâce au *filtrage par motif*.
En ce qui concerne Haskell , on peut donc avoir à comparer un mot au mot **Vide** : il faut donc dériver notre type de la classe **Eq** :

Haskell
```
data Mot =
  Vide
  | Char :+ Mot
  deriving (Show,Eq)

estVide :: Mot→ Bool
estVide m = m == Vide

λ> estVide Vide
True

λ> estVide m
False
```

Sélecteurs et testeurs sont alors liés par la relation :

$$\neg(\text{estVide } m) \leftrightarrow \big(m = (\text{tete } m) :+ (\text{queue } m)\big)$$

Cherchons maintenant à définir une fonction sur ce type **Mot**. Par exemple, on voudrait compter le nombre de « a » dans un mot.
Commençons par la signature :

Haskell
```
nba :: Mot→ Int
-- calcule le nombre de 'a' dans un mot
```

Nous allons étudier ensuite tous les cas de construction d'un mot. Il y en a deux.

— Si le mot est vide, alors son nombre de 'a' est 0.

— Sinon, le mot est construit par **t :+ q**.
  Si t = 'a', alors nba mot = 1 + nba q, sinon, nba mot = nba q.

Il n'y a plus qu'à transcrire cela en Haskell :

Haskell
```
nba Vide     = 0
nba (t :+ q) = (nba q) + (if t == 'a' then 1 else 0)
```

Autre syntaxe possible en filtrant sur le motif de **t** : :

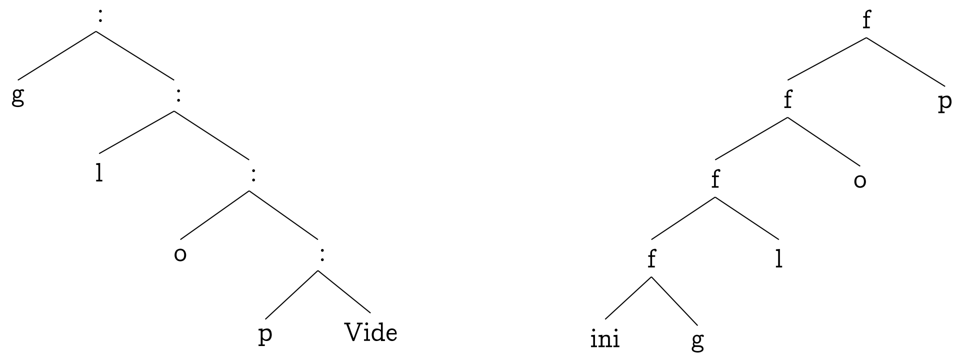
Haskell
```
nba2 Vide = 0
nba2 (t :+ q)
  | t == 'a' = (nba2 q) + 1
  |otherwise =  nba2 q
```

Une manière plus fonctionnelle de procéder est d'utiliser un pliage mais la fonction **foldl** est de signature **foldl :: (a -> b -> a) -> a -> [b] -> a** donc ne peut plier que des listes.

Il faudrait soit créer une fonction de pliage de mots.
Comme je suis très gentil, je vous propose une solution pour le pliage (pour vous aider d'ailleurs à mieux comprendre le pliage gauche...) :

Haskell
```haskell
pliage :: (a→ Char→ a)→ a→ Mot→ a
-- adapte foldl aux mots
pliage fonc ini Vide     = ini
pliage fonc ini (t :+ q) = pliage fonc (fonc ini t) q


nba4 :: Mot→ Int
-- calcule le nombre de 'a' dans un mot par pliage
nba4 mot =
  pliage (\accu t→ accu + (if t == 'a' then 1 else 0)) 0 mot
```

En fait, il faut penser à un immeuble qui s'écroule, une lunette télescopique qu'on...replie, etc.



avec ici `f = λacc lettre→ acc + (if lettre == 'a' then 1 else 0)` et `ini=0`

Zoom sur les listes

Vous étudierez en algo2 les listes d'un point de vue « bas niveau », c'est-à-dire en mettant les mains dans le cambouis de la machine.
En mathématique, nous restons propres et proches de la nature. Les listes ne seront pas des parties d'un moteur polluant mais de beaux arbres bons pour la planète. Malgré tout, ce seront des arbres un peu spéciaux car ils seront plats...
Ce type pré-existe bien sûr en Haskell . Dans un but pédagogique, nous allons cependant le reconstruire à la main.

Le constructeur liste

Bon, nous avons vu les mots dans la section précédente donc vous devriez pouvoir créer sans souci un type **`Liste`**...il faudrait juste préciser ce que nous entendons par liste !
C'est un ensemble d'éléments de *même type* placés dans un *ordre donné*.
Conventionnellement, on note les listes en plaçant leurs élément séparés par des virgules entre crochets. Par exemple, voici une liste d'entiers : $[1,6,45,-7,0]$.
Nos mots étaient en fait des listes de caractères, mais on veut aussi pouvoir créer des listes d'entiers, de booléens, de schmurx, etc.
Nous allons donc créer un type *polymorphe*...

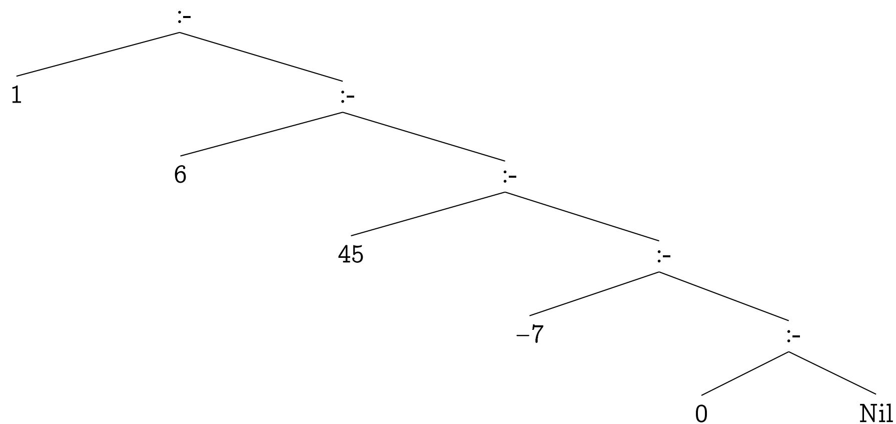
Licence Creative Commons

Haskell
```
infixr 1 :-

data Liste a =
  Nil
  | a :- (Liste a)
  deriving (Show,Eq,Ord)
```

Nous choisissons cependant de choisir nos éléments dans un ensemble ordonné pour rendre nos listes plus riches (...et surtout « triables »).
Par exemple :

Haskell
```
λ> let l1 = 1 :- 6 :- 45 :- (-7) :- 0 :- Nil
λ> l1
1 :- (6 :- (45 :- (-7 :- (0 :- Nil))))
λ> :t l1
l1 :: Liste Integer
```

La machine (et vous...) voit en fait ça :



On peut créer des listes de booléens si ça nous chante :

Haskell
```
λ> let l2 = True :- False :- True :- True :- Nil
λ> :t l2
l2 :: Liste Bool
```

et même des listes de listes !

Haskell
```
λ> let a1 = 1 :- 2 :- Nil
λ> let a2 = 3 :- 4 :- Nil
λ> let a3 = 5 :- 6 :- Nil
λ> let bigA = a1 :- a2 :- a3 :- Nil
λ> :t bigA
bigA :: Liste (Liste Integer)
```

6 Application, composition et priorité des fonctions

La fonction map

La fonction `map` est extrêmement utile en programmation fonctionnelle.
Son type est le suivant : `map :: (a → b)→ [a] → [b]`. Elle consiste à *appliquer* (map en anglais of spécialité) une fonction à tous les éléments d'une liste.
Par exemple, pour ajouter 3 à tous les éléments d'une liste d'entiers :

Haskell
```
λ> map (λx→ x + 3) [1..10]
[4,5,6,7,8,9,10,11,12,13]
```

ou plus simplement :

Haskell
```
λ> map (+ 3) [1..10]
[4,5,6,7,8,9,10,11,12,13]
```

Priorité et opérateur $

Nous avons également vu des règles de priorité. Par exemple l'application de fonction est prioritaire sur les opérateurs :

Haskell
```
λ> abs 3 - abs 5
-2
λ> (abs 3) - (abs 5)
-2
λ> abs (3 - abs 5)
2
```

Nous avons également vu que l'application des fonctions était associative à gauche, ce qui signifie que `f x y z` est équivalent à `((f x)y)z`.
Cela oblige parfois à avoir recours aux parenthèse. Par exemple, si nous voulons calculer la racine carrée de la somme des carrés des entiers de 1 à 10 :

Haskell
```
λ> sqrt (sum (map (^2) [1..10]))
19.621416870348583
```

Si on ne met pas de parenthèses :

Haskell
```
λ> sqrt sum map (^2) [1..10]

<interactive>:26:10-12:
    Couldn't match expected type '[(a1→ a1)→ [t1]→ t0]'
                with actual type '(a0→ b0)→ [a0]→ [b0]'
    In the second argument of 'sqrt', namely 'map'
    In the expression: sqrt sum map (^ 2) [1 .. 10]
    In an equation for 'it': it = sqrt sum map (^ 2) [1 .. 10]
```

Mais toutes ces parenthèses, c'est un peu pénible. Pour aérer un peu le code, il existe un opérateur un peu spécial, `$` qui est juste là pour appliquer l'opérande de gauche à l'opérande de droite :

Haskell
```
infixr 0 $
($) :: (a→ b)→ a→ b
f $ x = f x
```

ouais bof...Mais attention, le niveau de priorité de `$`est le plus bas qui soit. Il agit donc comme un retardateur d'application ou plus techniquement, il transforme l'associativité à gauche de l'application en associativité à droite : `f $ g $ h x= f (g (h x))`.
Pour en revenir à notre racine carrée de somme de carrés, on aurait pu écrire :

Haskell
```
λ> sqrt $ sum $ map (^2) [1..10]
19.621416870348583
```

Concrètement, le `$` revient à ouvrir une parenthèse et à la fermer à la fin de l'expression.

Recherche

Quelle est la valeur de :

Haskell
```
map ($ 4) [(5 -), (^ 2), (div 17)]
```

En résumé :

À retenir

$ appelle son argument de gauche (qui est une fonction) sur son argument de droite (qui est une valeur)

Composition de fonctions

Vous avez vu au lycée l'opérateur de composition des fonctions :
Avec $g\colon A \to B$ et $f\colon B \to C$.

$$f \circ g : x \mapsto f\big(g(x)\big)$$

Sur Haskell, c'est pareil :

Haskell
```
infixr 9 .
(.) :: (b→ c)→ (a→ b)→ a→ c
f . g = \x→ f (g x)
```

Danger

Il faut faire très attention à la signature de type des fonctions à composer !

Par exemple, la fonction `words` appliquée sur une chaîne contenant des espaces renvoie la liste des chaînes découpées selon les espaces :

Haskell
```
λ> words "vos beaux yeux d'amour belle mourir me font Marquise"
["vos","beaux","yeux","d'amour","belle","mourir","me","font","Marquise"]
```

On voudrait connaître la longueur du mot le plus long de cette liste.
On dispose des fonctions `length` et `maximum` qui calculent la longueur et l'élément maximum d'une liste d'éléments « ordonables ».
On vérifie bien les types :

Haskell
```
λ> :t words
words :: String→ [String]
λ> :t length
length :: [a]→ Int
λ> :t maximum
maximum :: Ord a⇒[a]→ a
```

Ce qui donne point de vue type :

$$\texttt{String} \xrightarrow{\texttt{words}} \texttt{[String]} \xrightarrow{\texttt{map length}} \texttt{[Int]} \xrightarrow{\texttt{maximum}} \texttt{Int}$$

Point de vue expressions :

$$\texttt{s} \xrightarrow{\texttt{words}} \texttt{words s} \xrightarrow{\texttt{map length}} \texttt{map length (words s)} \xrightarrow{\texttt{maximum}} \texttt{maximum (map length (words s))}$$

En compressant :

$$\texttt{String} \xrightarrow{\texttt{words . (map length . maximum)}} \texttt{Int}$$

ou plus simplement, sachant que la compression est associative à droite :

$$\texttt{String} \xrightarrow{\texttt{words . map length . maximum}} \texttt{Int}$$

Haskell
```
λ> let longMotPlusLong = maximum . map length . words
λ> :t longMotPlusLong
longMotPlusLong :: String→ Int
λ> longMotPlusLong "Vos beaux yeux d'amour mourir me font"
7
```

Les trois expressions suivantes désignent la même fonction :

```
maximum . map length . words
\s→ maximum (map length (words s))
\s→ maximum $ map length $ words s
```

7 Raisonnement par récurrence

Souvenez-vous du test vu dans une section précédente : « si n est pair alors son successeur est premier ».
Pourquoi a-t-on rajouté les nombres impairs à l'ensemble des entiers rendant cette proposition vraie ?
Parce que $A \to B$ est toujours vraie sauf dans le cas A est vraie et B est fausse. En particulier $A \to B$ reste vraie quand A est fausse.
Ceci est primordial dans la démonstration par récurrence, en particulier dans le preuve de l'hérédité.
Soit P_n une proposition dépendant d'un entier n.
On démontre l'hérédité $P_n \to P_{n+1}$ en utilisant le théorème de la déduction mais le fait que $P_n \to P_{n+1}$ soit vraie n'assure pas que P_n le soit.
Il faut de plus s'assurer que la propriété est vraie au moins une fois.
Ce type de raisonnement n'est donc pas une escroquerie comme certains élèves le pensent trop souvent : « ah oui, on suppose que c'est vrai au rang n et on en déduit que c'est vrai au rang n... ».

Récursion, récurrence, induction

...des mots qui tournent autour du même thème mais qu'il faut savoir distinguer.
L'*induction* consiste à conclure à partir d'observations préalables : cela correspond à la démarche expérimentale classique. Sans précautions, cela peut conduire à certaines contre-vérités. Par exemple 3, 5 et 7 sont premiers donc on pourrait en déduire, par induction, que tous les nombres impairs à partir de 3 sont premiers et pourtant...
L'*induction mathématique* ou *raisonnement par récurrence* corrige ce défaut. On part toujours d'un résultat observé mais on *démontre* qu'il est vrai pour tous les éléments d'un ensemble donné, éventuellement infini. C'est l'induction expérimentale corrigée par la rigueur du raisonnement mathématique.
En informatique, une *fonction récursive* (ou un *type récursif*) est une fonction (ou un type) qui fait référence à elle(lui)-même dans sa définition. Ce mécanisme est très puissant et permet de condenser l'écriture d'un programme.

C'est l'induction mathématique qui nous intéresse ici. Rappelons quelques principes.
Pour prouver qu'une propriété $\mathcal{P}_n$ dépendant uniquement d'un paramètre n est vraie pour tout $n \geqslant n_0$, il faut vérifier que :

— $\mathcal{P}_{n_0}$ est vraie (on parle parfois d'initialisation) ;

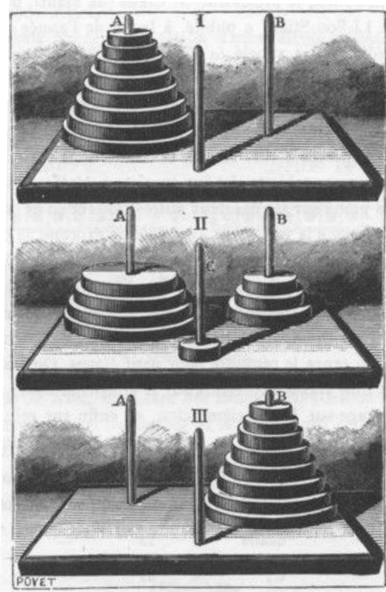
— pour tout $n \geqslant n_0$, $\mathcal{P}_n \to \mathcal{P}_{n+1}$ (on parle parfois d'hérédité).

Quand on a besoin de supposer que non seulement la propriété est vraie pour un certain n mais aussi pour tous les entiers qui lui sont inférieurs (pas seulement le père mais aussi tous les ascendants) on parle alors de *récurrence forte*.

Un jeu

Mathémator : L'été dernier, lors de la visite d'un musée, j'ai, pour une fois, impressionné mes enfants en résolvant devant eux très rapidement le problème suivant :

Licence Creative Commons



Ce casse-tête a été posé par le mathématicien français Édouard LUCAS en 1883.
Le jeu consiste en une plaquette de bois où sont plantés trois piquets. Au début du jeu, 4 disques de diamètres croissant de bas en haut sont placés sur le piquet de gauche. Le but du jeu est de mettre ces disques dans le même ordre sur le piquet de droite en respectant les règles suivantes :

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

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

Je vous propose donc de jouer vous aussi avec moi...

Hihutix (à part): *pauvres enfants, passer des vacances avec un prof de maths, vois le cauchemar* **(tout haut)** Oh, comme je suis content !

Mathémator : Essayez d'abord avec 2 puis 3 disques.

Hihutix: Bon, avec deux, je mets le disque là puis l'autre là-bas et ensuite celui-ci ici. Ça marche. Avec trois, je bouge celui-là, et puis l'autre, celui-ci ici, l'autre là, puis celui-là là et l'autre là puis le dernier ici. Ah, c'est pas le bon piquet mais ça marche à peu près. Facile votre jeu.

Mathémator : Alors essayez avec quatre, cinq, autant de disques que vous voulez.

Quelques minutes plus tard

Hihutix: Pénible votre jeu !

Mathémator : Les choses se compliquent. Il est temps de parler de la légende rapportée par LUCAS dans ses « *récréations mathématiques* » :

> *N. Claus de Siam a vu, dans ses voyages pour la publication des écrits de l'illustre Fer-Fer-Tam-Tam, dans le grand temple de Bénarès, au-dessous du dôme qui marque le centre du monde, trois aiguilles de diamant, plantées dans une dalle d'airain, hautes d'une coudée et grosses comme le corps d'une abeille. Sur une de ces aiguilles, Dieu enfila au commencement des siècles, 64 disques d'or pur, le plus large reposant sur l'airain, et les autres, de plus en plus étroits, superposés jusqu'au sommet. C'est la tour sacrée du Brahmâ. Nuit et jour, les prêtres se succèdent sur les marches de l'autel, occupés à transporter la tour de la première aiguille sur la troisième, sans s'écarter des règles fixes que nous venons d'indiquer, et qui ont été imposées par Brahma. Quand tout sera fini, la tour et les brahmes tomberont, et ce sera la fin des mondes !*

C'est un problème posé par un mathématicien mais nous allons l'étudier en informathicien. Nous voudrions répondre à plusieurs questions :

— peut-on résoudre le problème des 64 disques ?

— si oui, en combien de mouvements ?

— peut-on trouver une tactique la plus efficace possible ?

— est-ce que la fin du monde est pour bientôt ?

Au lieu d'étudier des cas séparés, nous allons généraliser un peu et considérer une tour avec n disques, n étant un entier naturel non nul.

Hihutix: Cela va compliquer les choses. Pourquoi ne pas s'en tenir à un cas précis ? C'est une perversion de mathématicien.

Mathémator : Détrompez-vous ! Au lieu d'étudier un à un des cas isolés, il est bon pour un informaticien aussi de voir qu'un programme plus général pourra traiter tous les cas. Cependant, il sera utile d'étudier des cas simples d'abord pour se donner une idée.
Après une rapide exploration avec papier et crayon, il est temps pour un informathicien d'introduire des notations adéquates.
Notons M_n le nombre minimum de mouvements pour transférer n disques. Pouvez-vous donner les premières valeurs de M_n ?

Hihutix: Bon, s'il n'y a qu'un disque, on ne pourra pas faire mieux qu'un déplacement : $M_1 = 1$. Avec deux disques, hop, hop, hop : trois mouvements ; $M_2 = 3$. Pour trois, j'avais trouvé sept déplacements mais je ne suis pas sûr que ça soit la meilleure solution.

Mathémator : C'est ce que nous allons prouver mais tout d'abord, en bon mathématicien, rajoutons un cas extrême : s'il y a zéro disque, il faudra zéro mouvement.

Hihutix: C'est un peu tiré par les cheveux.

Mathémator : Mais considérer les cas les plus triviaux permet souvent de simplifier le cas général. Il est temps d'ailleurs de voir grand mais il nous faudrait une idée. Essayons de trouver des points communs aux déplacements de deux et trois disques respectivement.

Hihutix: Ben, je mets le petit sur le piquet du milieu, le grand sur celui d'arrivée puis le petit sur ce même piquet d'arrivée.

Mathémator : Pour trois c'est pareil : je mets les deux petits sur celui du milieu, le grand sur celui d'arrivée puis les deux petits par dessus le grand.

Hihutix: Vous trichez ! Vous avez bougé deux disques en même temps.

Mathémator : En fait, j'ai vu que je savais bouger deux disques d'un piquet vers un autre. Cela me demande trois étapes qui se cachent derrière le « *je mets les deux petits sur celui du milieu* ». Cela nous donne en fait la solution : avec $n+1$ disques, je bouge les n plus petits sur le piquet du milieu (M_n mouvements), je bouge le plus grand sur le piquet d'arrivée (1 mouvement) puis je redéplace les n petits sur le grand (M_n mouvements). Ainsi :

$$M_{n+1} \leqslant 2M_n + 1$$

Hihutix: Pourquoi avoir utilisé $\leqslant$ et non pas = ? Et puis, on ne connaît pas M_n.

Mathémator : Vous touchez du doigt deux problèmes essentiels qui vont guider bon nombre de nos raisonnements.
Pour le $\leqslant$, il est essentiel de comprendre que nous avons prouvé que $2M_n + 1$ mouvements étaient **suffisants** mais nous ne savons toujours pas si ces $2M_n + 1$ mouvements sont **nécessaires**. C'est très important de comprendre la différence.
Par exemple, pour acheter un chocolat chaud à la cafétéria et l'offrir à son professeur adoré, il est **suffisant** d'avoir 10 euros dans sa poche mais ce n'est pas nécessaire. Inversement, il est **nécessaire** d'avoir au moins 20 centimes mais ce n'est malheureusement pas suffisant.
Nous avons étudié cela en détail dans un précédent chapitre sur la logique.

À retenir

On considère P et Q deux propositions (des énoncés, des faits, des formules).

- Q est une **condition nécessaire** pour avoir P si, dès que P est vraie, alors nécessairement, forcément, obligatoirement, Q est vraie. On note souvent $P \to Q$.
- Q est une **condition suffisante** pour avoir P s'il suffit que Q soit vraie pour que P soit vraie. On note souvent $P \to Q$.
- Lorsque P est à la fois condition nécessaire et condition suffisante de Q, on dit que P est une **condition nécessaire et suffisante** de Q ou encore que P est vraie **si, et seulement si**, Q est vraie. On note souvent $P \leftrightarrow Q$.

Lorsqu'une « affirmation » du type $P \leftrightarrow Q$ ou $P \leftrightarrow Q$ est vraie, on dit que c'est un **théorème**.

Nous avons développé des méthodes de preuve dans un chapitre précédent.
Revenons tout d'abord à nos disques. Peut-on faire mieux que la solution proposée ?
À un moment donné, il *faut* bien bouger le plus grand disque puisqu'il est sur le mauvais piquet. Il *faut* donc qu'il soit tout seul et il *faut* donc avoir bougé auparavant les n autres disques plus petits. Cela *nécessite au minimum* M_n mouvements. Après cela, on peut bouger le grand disque autant de fois qu'on le désire mais à un moment donné, il *faudra* le placer sur le piquet d'arrivée et déplacer les n autres disques sur le piquet d'arrivée ce qui *nécessite* à nouveau M_n mouvements.
Ainsi, il *faudra au minimum* effectuer $2M_n + 1$ mouvements.
On a donc prouvé que :

$$M_{n+1} \geqslant 2M_n + 1$$

Il est donc **nécessaire** d'effectuer $2M_n + 1$ mouvements pour déplacer les $n + 1$ disques. Or nous avons montré auparavant que ces $2M_n + 1$ mouvements étaient **suffisants**. On en déduit donc que :

$$\begin{cases} M_0 = 0 \\ M_n = 2M_{n-1} + 1 \text{ pour tout } n > 0 \end{cases}$$

Est-ce que cette formule marche bien avec nos premiers exemples ?

Hihutix: $M_1 = 2 \cdot 0 + 1 = 1$: OK. $M_2 = 2 \cdot 1 + 1 = 3$: OK. $M_3 = 2 \cdot 3 + 1 = 7$: OK.

Mathémator : Ça ne prouve pas grand chose mais ça nous rassure. De toute façon, nous avons prouvé que cette formule est bonne. Que vaut M_{64} ?

Hihutix: Ben $M_{64} = 2M_{63} + 1$. Le problème, c'est qu'on ne connaît pas M_{63}.

Mathémator : Et oui : on définit la *suite* **M** à partir d'elle-même, ce qui n'est pas très pratique. On dit qu'on a définit la suite par **récurrence**. Vous verrez cette année en informatique que ce mode de calcul n'est pas réservé aux suites que vous avez peut-être étudiées au lycée. Par exemple, en langage formel, on travaille avec les lettres d'un alphabet. On peut alors définir un mot comme étant soit le vide, soit une lettre suivi d'un mot.

Hihutix: C'est un peu idiot : en fait, vous me dites que pour savoir si un truc est un mot, faut déjà savoir si c'est un mot.

Mathémator : Vous n'êtes pas assez précis. Prenons notre alphabet et la chaîne de caractères « math ». C'est le vide suivi de la chaîne « math » qui est la lettre « m » suivie de la chaîne « ath » qui est la lettre « a » suivie de la chaîne « th » qui est la lettre « t » suivie de la lettre « h » qui est un mot puisque c'est une lettre. En remontant, on obtient que « th » est un mot puis que « ath » aussi et enfin « math » est donc un mot.

Hihutix (à part): *Ben on le savait avant. Si c'est ça c'qu'on apprend à l'IUT d'info, il est peut-être encore temps de m'inscrire en GEA.* **(tout haut)** Fabuleux ! Math est un mot ! J'aurais pas cru, comme ça, a priori.

Mathémator : Ah, comme quoi ce que nous faisons est utile.

Hihutix (à part): *Complètement gâteux le gars* **(tout haut)** Je n'en doute pas.

Mathémator : Et la fin du monde dans tout ça ? Pour connaître M_{64}, il semblerait qu'il faille connaître tous les M_k précédents.
Voyons voir, calculez jusqu'à M_6.

Hihutix: Si vous voulez :

— $M_3 = 2 \cdot 3 + 1 = 7$;

— $M_4 = 2 \cdot 7 + 1 = 15$;

— $M_5 = 2 \cdot 15 + 1 = 31$;

— $M_6 = 2 \cdot 31 + 1 = 63$;

Ouais, bof. Je ne vois pas trop où ça nous mène.

Mathémator : « *Ils ont des yeux et ils ne voient pas* ». Et pourtant, quand j'étais jeune, un enfant de six ans aurait reconnu la suite :

$$2^3 - 1,\ 2^4 - 1,\ 2^5 - 1,\ 2^6 - 1, \ldots$$

Hihutix (à part): *Ben tiens ! Les puissances à 6 ans et pourquoi pas les logarithmes à 7.*

Mathémator : Vous dites ?

Hihutix: Je réfléchissais.

Mathémator : Ah. Alors, vous avez sûrement envie de dire que tout prête à croire que :

$$M_n = 2^n - 1, \text{ pour tout entier strictement positif } n$$

Hihutix: Ben oui, c'est vrai au début donc y a pas de raison pour que ça s'arrête.

Mathémator : Encore faudrait-il le prouver. Par exemple, est-ce que la proposition suivante est un théorème :

Les entiers impairs supérieurs à 3 sont tous des nombres premiers.

Hihutix: 3 est premier, 5 est premier, 7 est premier...

Mathémator : ...donc c'est vrai !

Hihutix: Ben non. 9 est impair mais n'est pas premier...Ouais, OK, j'ai compris.

Mathémator : « Vrai pour les premiers, vrai pour tous » n'est pas un théorème ! Vous retiendrez également que...

À retenir

> Pour prouver qu'une proposition n'est pas un théorème, il suffit d'exhiber un **contre-exemple**.

Nous en avons déjà parlé. Revenons à notre récurrence ou induction.

À retenir

> Pour prouver qu'une propriété $\mathcal{P}_n$ dépendant uniquement d'un paramètre n est vraie pour tout $n \geqslant n_0$, il faut vérifier que :
>
> — $\mathcal{P}_{n_0}$ est vraie (on parle parfois d'initialisation) ;
>
> — pour tout $n \geqslant n_0$, $\mathcal{P}_n \to \mathcal{P}_{n+1}$ (on parle parfois d'hérédité).

Cela peut par exemple se prouver en *raisonnant par l'absurde* comme nous l'avons déjà vu.
Ici, pour tout entier naturel non nul n, notons $\mathcal{P}_n$ la proposition : $M_n = 2^n - 1$.
Cette proposition est vraie pour $n = 1$ puisque $M_1 = 1 = 2^1 - 1$.
Il existe donc au moins un entier naturel non nul k tel que $\mathcal{P}_k$ soit vraie.
Alors $M_{k+1} = 2M_k + 1 = 2(2^k - 1) + 1 = 2 \cdot 2^k - 2 + 1 = 2^{k+1} - 1$.
Ainsi $\mathcal{P}_k \to \mathcal{P}_{k+1}$ pour tout entier naturel non nul k.
Le tour est joué. On peut donc affirmer que $M_n = 2^n - 1$ pour tout entier naturel non nul n.

Hihutix: Donc $M_{64} = 2^{64} - 1$. Je prends ma calculatrice...$M_{64} = 18\,446\,744\,073\,709\,551\,615$. Ça ne m'arrange pas tellement pour connaître la date de la fin du monde.

Mathémator : C'est pourtant simple. Disons que les moines sont très bien entraînés et se relaient efficacement. Ils déplacent alors un disque par seconde. Cela nous donne une durée de jeu de...

Hihutix: Ah, ça je sais : $18\,446\,744\,073\,709\,551\,615/3600/24/365,25/100$ ce qui fait en gros 5.8 milliards de siècles...

Mathémator : Et comme l'Univers a grosso modo 140 millions de siècles d'existence, cela nous laisse de la marge.

Hihutix: Ouf ! Mais cela ne nous indique pas comment déplacer les huit disques du jeu initial. On sait juste qu'en étant aussi efficaces que les moines, cela nous prendra $M_8 = 2^8 - 1 = 255$ secondes.

Mathémator : Je vous laisse donc cinq minutes pour le faire.

Cinq minutes plus tard

Hihutix: Il faut se rendre à l'évidence : je ne suis pas fait pour être moine. Faut le faire pour chaque n jusqu'à 8 en fait et noter comment on a fait. Pfff... C'est pénible.

Mathémator : Et quand un informathicien a bien réfléchi sur le papier, il peut laisser faire le sale boulot à la machine. On a en effet une méthode qu'on sait être correcte. Mais il est très difficile de l'appliquer pour un grand n donné. On sait que ça marche mais on ne sait pas vraiment comment ça marche. Et c'est là qu'une programmation bien pensée devient merveilleusement

efficace. Si on a à notre disposition un langage sachant traiter la récursion, il va suffire de lui donner un minimum d'instruction.
Rappelons ici notre méthode : avec $n+1$ disques, je bouge les n plus petits sur le piquet du milieu (M_n mouvements), je bouge le plus grand sur le piquet d'arrivée (1 mouvement) puis je redéplace les n petits sur le grand (M_n mouvements).
Illustrons notre propos avec le langage Haskell qui aime les récursions.
L'opérateur **++** concatène des listes donc aussi des chaînes qui sont des listes de caractères :

Haskell
```
*Main> :t (++)
(++) :: [a]→ [a]→ [a]
*Main> [1,2,3] ++ [4]
[1,2,3,4]
*Main> "Raahhh" ++ " lovely"
"Raahhh lovely"
```

On traite les deux cas, un ou n disques :

Haskell
```
hanoi ::  String→ String→ String→ Int→ String
-- On nomme les trois piquets sous forme de chaînes.
-- On obtient la liste des mouvements sous forme d'une chaîne.
-- On déplace n disques de dep vers but en se servant de piv comme pivot.
hanoi dep but piv 1 =
  -- 1 disque : renvoie la chaîne décrivant le mvt de dep vers but
  "De " ++ dep ++ " vers " ++ but ++ "\n"
hanoi dep but piv n =
  -- n disques :
  (hanoi dep piv but (n - 1)) ++ --  on bouge n-1 disques de dep vers piv
  (hanoi dep but piv 1) ++ -- on bouge le gros disque de dep vers but
  (hanoi piv but dep (n - 1)) -- on bouge le n-1 disques du piv vers but
```

On affiche joliment le résultat :

Haskell
```
-- on affiche le résultat
tours_hanoi :: Int→ IO ()
tours_hanoi n = putStrLn ( hanoi "Tige 1" "Tige 2" "Tige 3" n)
```

Ce qui donne pour trois disques :

Haskell
```
λ> tours_hanoi 3
De Tige 1 vers Tige 2
De Tige 1 vers Tige 3
De Tige 2 vers Tige 3
De Tige 1 vers Tige 2
De Tige 3 vers Tige 1
De Tige 3 vers Tige 2
De Tige 1 vers Tige 2
```

et si l'on veut obtenir le nombre de mouvements pour déplacer de 1 à 20 disques, on change un peu le code :

Haskell
```
nb_hanoi :: Int→ Int
nb_hanoi 1 = 1
nb_hanoi n = 2 * (nb_hanoi (n-1)) + 1

λ> [nb_hanoi k | k←[1..20]]
[1,3,7,15,31,63,127,255,511,1023,2047,4095,8191,16383,32767,65535,131071,262143,
 524287,1048575]
```

Hihutix: Waouh ! Ça je préfère.

Mathémator : Je n'en doute pas, mais pour y arriver, il faudra passer par le stade « réflexion mathématiquement assistée avec papier-crayon »

8 Prouver que deux fonctions sont équivalentes

Deux fonctions sont équivalentes si elles ont le même type de départ, le même type d'arrivée et le même graphe, i.e. si, appliquées au même argument, elles donnent la même valeur.
Considérons deux fonctions simples :

Haskell
```haskell
f1 :: [Int]→ [Int]
f1    []     = []
f1    (k:ks) = abs k : f1 ks

f2 :: [Int]→ [Int]
f2 = map abs
```

Sont-ce les mêmes fonctions ?
Regardons la définition de `map` dans la bibliothèque `Data.List` :

Extrait de Data.List
```haskell
-- map f xs is the list obtained by applying f to each element of xs, i.e.,
-- map f [x1, x2, ..., xn] == [f x1, f x2, ..., f xn]
-- map f [x1, x2, ...] == [f x1, f x2, ...]
map :: (a→ b)→ [a]→ [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

Nous allons démontrer ce résultat par récurrence (ou induction : du simple vers le compliqué).
On commence par traiter le cas simple : l'argument est la liste vide.
`f1 []`
`=> [] définition de f1`
`=> map abs [] définition de map`
`=> f2 [] défintion de f2`
Nous pouvons donc supposer que `f1 ks = f2 ks` pour une certaine liste `ks` et nous allons essayer de démontrer que `f1 (k:ks)= f2 (k:ks)` pour n'importe quel entier `k`.
f1 (k :ks)
`f1 (k:ks)`
`=> abs k : f1 ks définition de f1`
`=> abs k : f2 ks hypothèse de récurrence`
`=> abs k : map abs ks définition de f2`
`=> map abs (k:ks) définition de map`
`=> f2 (k:ks) définition de f2`
Et voilà...

RECHERCHES

Recherche 4 - 1 Ensembles à partir de fonctions

On définit des ensembles par compréhension :

Haskell

```
module EnsembleCompréhension where

    type Element   = Int
    type Predicat = Element→ Bool
    data Ensemble = Ens (Predicat)
```

On utilisera un petit *truc* pour afficher joliment les listes et faire un teste de vacuité :

Haskell

```
 mini,maxi  :: Element
(mini,maxi) = (-128,127)

listeUnivers :: [Element]
listeUnivers =  [mini .. maxi]

listeVersEns :: Ensemble→ [Element]
listeVersEns    (Ens f)  =  [x | x←listeUnivers, f x]

instance  Show Ensemble where
   show = show . listeVersEns
```

Pour le reste, c'est à vous de jouer :

Haskell

```
contient :: Ensemble→ Element→ Bool

univers :: Ensemble

insere :: Element→ Ensemble→ Ensemble

union ::  Ensemble→ Ensemble→ Ensemble

inter ::  Ensemble→ Ensemble→ Ensemble

filtre :: Predicat→ Ensemble→ Ensemble

complementaire :: Ensemble→ Ensemble

difference :: Ensemble→ Ensemble→ Ensemble

estInclus :: Ensemble→ Ensemble→ Bool

estEgal :: Ensemble→ Ensemble→ Bool

pourTous :: Predicat→ Ensemble→ Bool

ilExiste :: Predicat→ Ensemble→ Bool
```

Recherche 4 - 2 Puissance

Écrivez la fonction carré d'au moins quatre façons différentes.

Recherche 4 - 3 Double

Définissez une fonction qui prend une fonction d'un argument en argument et retourne une focntion qui applique l'argument deux fois. (N'oubliez pas les signatures !)
Par exemple, si `suiv` est la fonction qui a un entier associe son successeur, alors `double suiv` sera la fonction qui à un entier x associe $x+2$.
Quelle est la valeur retournée par `(double (double double))suiv 5` ?

Recherche 4 - 4 Composition itérée

Déterminez une fonction qui prend une fonction f et un naturel n en argument et retourne composée n fois de f. o
Vous obtiendrez par exemple :

Haskell
```
λ> (compRep suiv 0) 5
5
λ> (compRep (^2) 2) 5
625
```

Recherche 4 - 5 Comparatif

Écrivez une fonction qui, étant donnée une liste d'entiers, calcule la somme des nombres pairs de cette liste, puis une autre calculant la somme des carrés pairs. (Une demi-ligne devrait suffire).
En C on pourrait faire ça :

C
```
int evenSum(int *list) {
    return accumSum(0,list);
}

int accumSum(int n, int *list) {
    int x;
    int *xs;l
    if (*list == 0) {
        return n;
    } else {
        x = list[0];
        xs = list + 1;
        if ( 0 == (x%2) ) {
            return accumSum(n+x, xs);
        } else {
            return accumSum(n, xs);
        }
    }
}
```

Recherche 4 - 6 Opérateurs logiques

Nous allons reprendre notre travail sur les opérateurs logiques (qui sont la base de l'architecture de la machine). Nous allons cette fois utiliser le type `Bool` primitif afin de pouvoir profiter des diverses constructions de fonctions vues en cours.

1. Commencer par construire des fonctions « et », « ou » et « non ».
   Comme nous allons créer pas mal d'opérateurs binaires booléens, on peut commencer par créer un synonyme pour clarifier le code :

   Haskell
   ```
   module Logique where

       -- type : crée un type synonyme pour rendre le code plus clair à lire
       type OpBinaire = (Bool→ Bool→ Bool)
   ```

Enseuite, par exemple, pour reprendre ce qui a été fait en cours :

Haskell
```
-- ET logique
et :: OpBinaire
et    True = λx  → x
et    _    = λ_  → False
```

2. Créez de même le ou exclusif ouex, l'implication ==> et l'équivalence <==>.

3. Qu'est-ce que c'est ?

Haskell
```
lesBool :: [Bool]
lesBool = [False, True]

tabVer :: OpBinaire→ [Bool]
tabVer op = [ [ op x y | x←lesBool] |  y←lesBool ]
```

Comment définiriez-vous l'égalité de deux opérateurs logiques ?

4. Vérifiez à l'aide de ces fonctions, que pour tout couple (x, y) de booléens :

i. `non (x ‘et‘ y)= (non x)‘ou‘ (non y)`

ii. `non (x ‘ou‘ y)= (non x)‘et‘ (non y)`

On appelle ces résultats les *lois de* De Morgan.

5. Exprimez tous les opérateurs précédents (ou , ouex, ==>, <==>) uniquement à l'aide de et et non.
On note habituellement « nand » l'opérateur vérifiant :

Haskell
```
λ> tabVer nand
[[True,True],[True,False]]
```

Pourquoi pourriez-vous construire l'unité arithmétique d'un ordinateur avec un lot de circuits « nand » ?

6. Est-ce que l'implication est associative ?

Recherche 4 - 7 Moyenne olympique

1. Décrire une fonction qui étant donnés quatre flottants leur associe la moyenne des deux nombres parmi les quatre qui ne sont ni le plus grand ni le plus petit. Par exemple, la moyenne olympique de 10, 8, 12 et 14 est 11 et celle de 12, 12, 12 et 12 est 12.

2. Décrire une fonction qui étant donnée une liste d'au moins quatre nombres leur associe la moyenne des nombres parmi ceux de la liste qui ne sont ni le plus grand ni le plus petit. Par exemple, la moyenne olympique de 15, 7, 10, 8, 12 et 14 est 11.

Recherche 4 - 8 Machine polonaise

Étudiez avec la plus grande attention ce tutoriel :
http://lyah.haskell.fr/resoudre-des-problemes-fonctionnellement#calculatrice-de-notation-polonaise-inverse
Puis créez une fonction qui lit une expression infixe et la transforme en expression NPI et inversement.

Recherche 4 - 9 dérivateur formel

Vous avez passé des mois à dériver des fonctions au lycée. Nous allons essayer de créer un dérivateur formel (suffisant du point de vue calul mais basique point de vue affichage).
On part de ça :

Haskell
```
module Derivation where

type Nombre   = Float
```

```
data Fonction   =
    Const Nombre
        | Id        String
        | Add       Fonction Fonction
        | Mul       Fonction Fonction
        | Inv       Fonction
        | Pow Float Fonction
        | Racine    Fonction
        | Ln        Fonction
        | Exp       Fonction
        | Cos       Fonction
        | Sin       Fonction
    deriving (Eq)

montre :: Fonction-> String
montre (Const x)     = show x
montre (Id x)        = x
montre (Add f1 f2)   = "( " ++ (montre f1) ++ " + " ++ (montre f2) ++ " )"
montre (Mul f1 f2)   = "( " ++ (montre f1) ++ " * " ++ (montre f2) ++ " )"
montre (Inv f)       = "( 1 / " ++ (montre f) ++ " )"
montre (Pow k f)     = "( " ++ (montre f) ++ " )^" ++ (show k)
montre (Racine f)    = "Rac( " ++ (montre f) ++ " )"
montre (Ln f)        = "Ln( " ++ (montre f) ++ " )"
montre (Exp f)       = "Exp( " ++ (montre f) ++ " )"
montre (Cos f)       = "Cos( " ++ (montre f) ++ " )"
montre (Sin f)       = "Sin( " ++ (montre f) ++ " )"

instance Show Fonction where
              show = montre

eval :: Fonction   -> Nombre-> Nombre
eval    (Const x)     _      =  x
eval    (Id _)        n      =  n
eval    (Add f1 f2)   n      = (eval f1 n) + (eval f2 n)
eval    (Mul f1 f2)   n      = (eval f1 n) * (eval f2 n)
eval    (Inv f)       n      = 1 / (eval f n)
eval    (Pow k f)     n      = (eval f n)**k
eval    (Racine f)    n      = sqrt (eval f n)
eval    (Ln f)        n      = log (eval f n)
eval    (Exp f)       n      = exp (eval f n)
eval    (Cos f)       n      = cos (eval f n)
eval    (Sin f)       n      = sin (eval f n)
```

Complétez alors les lignes suivantes :

Haskell

```
derive :: Fonction-> Fonction-> Fonction
derive    (Const _)   _        = ?
derive    (Id x)      v        = ?
derive    (Add f1 f2) v        = ?
derive    (Mul f1 f2) v        = ?
derive    (Inv f)     v        = ?
derive    (Pow k f)   v        = ?
derive    (Racine f)  v        = ?
derive    (Ln f)      v        = ?
derive    (Exp f)     v        = ?
derive    (Cos f)     v        = ?
derive    (Sin f)     v        = ?
```

Alors par exemple :

Haskell

```
λ> derive (Pow 2 x) x
( 1.0 * ( 2.0 * ( x )^1.0 ) )
λ> derive (Pow 3 x) x
( 1.0 * ( 3.0 * ( x )^2.0 ) )
```

```
λ> derive (Pow 2 (Racine x)) x
( ( 1.0 * ( 1 / ( 2.0 * Rac( x ) ) ) ) * ( 2.0 * ( Rac( x ) )^1.0 ) )
λ> derive (Ln (Ln (Ln x))) x
( ( ( 1.0 * ( 1 / x ) ) * ( 1 / Ln( x ) ) ) * ( 1 / Ln( Ln( x ) ) ) )
λ> derive (Ln (Cos x)) x
( ( -1.0 * ( 1.0 * Sin( x ) ) ) * ( 1 / Cos( x ) ) )
```

Recherche 4 - 10 Jeu de Mots

1. Déterminez une fonction `longueur :: Mot -> Int` qui renvoie le nombre de caractères constituant un mot.
2. Déterminez une fonction `renverse :: Mot -> Mot` qui inverse l'ordre d'un mot.
3. Déterminez une fonction `colle :: Mot -> Mot -> Mot` qui concatène deux mots.
4. Déterminez une fonction `liste_of_Mot :: Mot -> [Char]` qui à un mot associe la liste de ses caractères.

Recherche 4 - 11 Liste de fonctions sur les listes

Voici les outils minimums tels qu'ils sont décrits dans la documentation du module **Data.List** de Haskell . Adaptez-les à notre type **Liste** et donnez-leur un nom français pour éviter les conflits avec les fonctions de Haskell .

Extrait de la documentation de Data.List

```
 (++) :: [a]→ [a]→ [a]
Append two lists, i.e.,
 [x1, ..., xm] ++ [y1, ..., yn] == [x1, ..., xm, y1, ..., yn]
 [x1, ..., xm] ++ [y1, ...] == [x1, ..., xm, y1, ...]
If the first list is not finite, the result is the first list.

head :: [a]→ a
Extract the first element of a list, which must be non-empty.

last :: [a]→ a
Extract the last element of a list, which must be finite and non-empty.

tail :: [a]→ [a]
Extract the elements after the head of a list, which must be non-empty.

init :: [a]→ [a]
Return all the elements of a list except the last one. The list must be non-empty.

null :: [a]→ Bool
Test whether a list is empty.

length :: [a]→ Int
length returns the length of a finite list as an Int. It is an instance of the more general
    Data.List.genericLength, the result type of which may be any kind of number.

map :: (a→ b)→ [a]→ [b]
map f xs is the list obtained by applying f to each element of xs, i.e.,
 map f [x1, x2, ..., xn] == [f x1, f x2, ..., f xn]
 map f [x1, x2, ...] == [f x1, f x2, ...]

reverse :: [a]→ [a]
reverse xs returns the elements of xs in reverse order. xs must be finite.

foldl :: (a→ b→ a)→ a→ [b]→ a
foldl, applied to a binary operator, a starting value (typically the left-identity of the operator), and
    a list, reduces the list using the binary operator, from left to right:
 foldl f z [x1, x2, ..., xn] == (...((z ‘f‘ x1) ‘f‘ x2) ‘f‘...) ‘f‘ xn
The list must be finite.

any :: (a→ Bool)→ [a]→ Bool
Applied to a predicate and a list, any determines if any element of the list satisfies the predicate.

all :: (a→ Bool)→ [a]→ Bool
```

```
Applied to a predicate and a list, all determines if all elements of the list satisfy the predicate.

sum :: Num a⇒[a]→ a
The sum function computes the sum of a finite list of numbers.

product :: Num a⇒[a]→ a
The product function computes the product of a finite list of numbers.

maximum :: Ord a⇒[a]→ a
maximum returns the maximum value from a list, which must be non-empty, finite, and of an ordered type.
    It is a special case of maximumBy, which allows the programmer to supply their own comparison
    function.

minimum :: Ord a⇒[a]→ a
minimum returns the minimum value from a list, which must be non-empty, finite, and of an ordered type.
    It is a special case of minimumBy, which allows the programmer to supply their own comparison
    function.

takeWhile :: (a→ Bool)→ [a]→ [a]
takeWhile, applied to a predicate p and a list xs, returns the longest prefix (possibly empty) of xs of
    elements that satisfy p:
 takeWhile (< 3) [1,2,3,4,1,2,3,4] == [1,2]
 takeWhile (< 9) [1,2,3] == [1,2,3]
 takeWhile (< 0) [1,2,3] == []

elem :: Eq a⇒a→ [a]→ Bool
elem is the list membership predicate, usually written in infix form, e.g., x ‘elem‘ xs.

find :: (a→ Bool)→ [a]→ Maybe a
The find function takes a predicate and a list and returns the first element in the list matching the
    predicate, or Nothing if there is no such element.

filter :: (a→ Bool)→ [a]→ [a]
filter, applied to a predicate and a list, returns the list of those elements that satisfy the
    predicate; i.e.,
 filter p xs = [ x | x←xs, p x]

partition :: (a→ Bool)→ [a]→ ([a], [a])
The partition function takes a predicate a list and returns the pair of lists of elements which do and
    do not satisfy the predicate, respectively; i.e.,
 partition p xs == (filter p xs, filter (not . p) xs)

zip :: [a]→ [b]→ [(a, b)]
zip takes two lists and returns a list of corresponding pairs. If one input list is short, excess
    elements of the longer list are discarded.
```

Recherche 4 - 12 Ensembles à partir de listes

Créez un type Ens qui implémente les ensembles à partir des listes.
Il faudra prévoir les principales fonctionnalités comme un test d'appartenance, d'inclusion, des opération d'union, d'intersection, de différence.
Vous partirez de ça :

Haskell

```haskell
module EnsembleListe where

type Element  = Int
type Predicat = Element→ Bool
data Ensemble = Ens ([Element])

-- Partie qui utilise les listes

instance Show Ensemble where
    show (Ens liste) = show liste
```

```
estVide :: Ensemble   → Bool
estVide    (Ens liste) =  liste == []

contient :: Ensemble→ Element→ Bool
contient    (Ens liste) e       =  e ‘elem‘ liste

insere ::  Element→ Ensemble      → Ensemble
insere     e          (Ens liste)
           | (Ens liste) ‘contient‘ e = Ens liste
           | otherwise                = Ens (e:liste)

-- Partie indépendante des listes

union ::  Ensemble  → Ensemble → Ensemble

inter :: Ensemble→ Ensemble→ Ensemble

pourTous :: Predicat→ Ensemble    → Bool

ilExiste :: Predicat→ Ensemble    → Bool

estInclus :: Ensemble→ Ensemble→ Bool

estEgal :: Ensemble→ Ensemble→ Bool

filtre :: Predicat→ Ensemble→ Ensemble

cardinal :: Ensemble→ Int
```

Recherche 4 - 13

On se propose de démontrer que tous les étudiants d'IUT ont le même âge et, pour cela, on note $P(n)$ l'affirmation « *si on choisit n étudiants ($n \in \mathbb{N}^*$), il est sûr qu'ils ont tous le même âge* »
Il est clair que $P(1)$ est vraie.
Démontrons que $P(n) \to P(n+1)$. Pour cela nous supposons que $P(n)$ est vraie (c'est l'hypothèse de récurrence) et nous choisissons un groupe quelconque de $n+1$ étudiants que nous ordonnons par ordre alphabétique (pourquoi pas). D'après l'hypothèse de récurrence, les n premiers de l'ordre alphabétique ont tous le même âge ainsi que les n derniers. Comme ces deux groupes de n étudiants ont au moins un étudiant en commun, on en déduit qu'ils ont tous le même âge.
Nous venons de démontrer que $P(n) \to P(n+1)$ pour tout $n \geq 1$ et, comme $P(1)$ est vrai, $P(n)$ est toujours vrai pour tout $n \geqslant 1$. **Y a-t-il une erreur dans le raisonnement ?**

Recherche 4 - 14 Javanais for dummies

On va simplifier le javanais habituel sans tenir compte de la découpe selon les syllabes. On introduit les lettres « av » devant la première voyelle rencontrée dans le mot en partant de la gauche. Créez une foncton `javanais(mot)` :

Haskell

```
λ> javanais "chrome"
"chravome"

λ> javanais "air"
"avair"
```

Recherche 4 - 15 Preuve

Prouvez par induction les propriétés suivantes :

Haskell

```
map (f . g)  = map f . map g
map f . tail = tail . map f
map f . concat = concat . concat (map f)
```

```
f . head = head . map f
take m . take n = take (min m n)
drop m . drop n = drop (m + n)
last (reverse xs) = head xs
```

Recherche 4 - 16 Subtilité : fonction stricte

Considérons la fonction `bolos = bolos` : c'est une expression qui a pour valeur $\perp$.
Une fonction est dite stricte si, et seulement si, appliquée à `bolos`, elles renvoie $\perp$. Est-ce que toutes les fonctions sont strictes sur Haskell ?

Recherche 4 - 17 Exponentiation

1. Voici une définition très classique de la puissance entière d'un entier : $x^0 = 1$ et $x^n = x \times x^{n-1}$
   Cela donne :

   Haskell
   ```
   (°) :: Integer→ Integer→ Integer
   (°)    x         0       =  1
   (°)    x         n       =  x*x°(n-1)
   ```

   Démontrez alors que pour tous entiers naturels x, n et m, `x°(n + m)= x°n *x°m`
   Attention, il y a un piège...
2. Démontrez ensuite, toujours par induction, que `x°n *x°n = (x*x)°n`.
3. Mais il y a mieux à faire en remarquant que, quand n est pair, $x^n = (x \times x)^{n/2}$: quel est l'intérêt, algorithmiquement parlant ? Comment en tirer parti ?
   Vous pourrez avoir besoin de la fonction `div` qui retourne le quotient de la division euclidienne de deux entiers.
   Vous noterez par exemple `^!` ce nouvel opérateur.
   Souvenez-vous de vos cours de 6eme. Effectuer la division euclidienne de deux entiers a et b, b étant non nul, c'est déterminer les entiers q et r tels que :

   $$a = b \times q + r \qquad 0 \leqslant r < |b|$$

   Comparez le nombre d'étapes nécessaires pour calculer 3^{25} avec l'une et l'autre méthode. Nous essaierons de généraliser tout à l'heure mais revenons d'abord à nos moutons.
4. Il s'agit maintenant de démontrer que les fonctions `(°)` et `^!` sont équivalentes.
   Le cas n impair est assez direct. Pour le cas pair, il y a plus de travail...
5. Moralité ? Parmi ces deux codes, un est simple à lire et peu efficace, l'autre est compliqué à lire et efficace : faut-il malgré tout jeter le premier à la poubelle ?
6. Peut-on avoir un ordre de grandeur du nombre d'étapes nécessaires au calcul de x^n ? Cela peut avoir une importance capitale, notamment en cryptographie où on calcule régulièrement des entiers à la puissance des nombres d'une centaine de chiffres.
   Faisons pour cela un petit rappel sur une fonction primordiale en informatique. Nous voudrions savoir combien de bits sont nécessaires à l'écriture d'un entier...Menez l'enquête. Peut-être aurez-vous besoin d'introduire le logarithme de base 2 :

   $$\log_2(x) = \frac{\ln x}{\ln 2}$$

Recherche 4 - 18 Factorielle

La factorielle d'un entier positif, vous connaissez :

$$0! = 1 \qquad n! = n \times (n-1)!$$

Proposez une définition récursive directe et une version avec accumulateur puis prouvez par induction qu'elles sont équivalentes.

Recherche 4 - 19 CCP MP 2015

Non, non, ce n'est pas un extrait d'un livre de seconde, c'est un sujet 0 proposé pour l'épreuve de Math du concours CCP...

1. Écrire une fonction factorielle qui prend en argument un entier naturel n et renvoie $n!$ (on n'accepetra pas bien sûr de réponse utilisant la propre fonction factorielle du module math de Python ou Scilab).
2. Écrire une fonction `seuil` qui prend en argument un entier M et renvoie le plus petit entier naturel n tel que $n! > M$.
3. Écrire une fonction booléenne nommée `est_divisible`, qui pend en argument un entier naturel n et renvoie `True` si $n!$ est divisible par $n+1$ et `False` sinon.
4. On considère la fonction suivante nommée `mystere` :

Python
```python
def mystere(n):
    s = 0
    for k in range(1,n+1):
        s = s + factorielle(k)
    return s
```

   i. Quelle valeur renvoie `mystere(4)` ?
   ii. Déterminer le nombre de multiplications qu'effectue `mystere(n)`.
   iii. Proposer une amélioration du script de la fonction `mystere` afin d'obtenir une complexité linéaire.

Recherche 4 - 20 Nombres premiers

Définition

Quoi de plus simple qu'un nombre premier :

Définition 4 - 2 Un entier naturel est dit premier s'il est supérieur (i.e. supérieur ou égal) à 2 et n'est divisible que par 1 et lui-même.

et pourtant, ils renferment tant de mystères que les plus grands esprits depuis des siècles n'ont toujours pas réussi à en percer tous les secrets et ce malgré les énormes progrès technologiques et les investissements colossaux consentis par les pouvoirs tant civils que militaires pour assurer ou percer la confidentialité des transmissions de toutes natures qui continuent de dépendre d'une meilleure connaissance des nombres premiers, ce que nous verrons un peu plus loin. Qui sont-ils ? Combien sont-ils ? Où sont-ils ? À quoi servent-ils ? Nous essaierons de donner quelques éléments de réponses à ces questions.
Mais tout d'abord, pourquoi jouent-ils un rôle si important ? Fondamentalement, il existe deux manières d'engendrer $\mathbb{N}$:

— si on veut engendrer $\mathbb{N}$ en utilisant l'addition, on s'aperçoit que le nombre 1 nous suffit : on « fabrique » 2 en additionnant 1 avec lui-même ; 3 en additionnant 1 avec 2, etc.

— si on veut engendrer $\mathbb{N}$ en utilisant la multiplication, là, les choses se compliquent. Pour « fabriquer » 2, il faut le créer ; même problème pour 3. On fabrique 4 en multipliant 2 avec lui-même, mais il faut créer 5. On fabrique 6 en multipliant 3 avec 2. On crée 7. On fabrique 8 à partir de 2. On fabrique 9 à partir de 3. On fabrique 10 à partir de 2 et 5, etc.

  Les nombres que l'on est obligés de créer sont les briques nécessaires à fabriquer tous les autres. C'est bien plus compliqué que l'addition me direz-vous, mais la multiplication est plus « puissante » et nous permet d'aller bien plus vite et plus loin.

Les nombres premiers sont donc ces éléments qui nous permettent de fabriquer tous les autres. Un des premiers problèmes étudiés à été de savoir s'ils peuvent tenir dans une boîte. Euclide à répondu à cette question il y a vingt-trois siècles et la réponse est non.
Pour le prouver, nous aurons besoin d'un résultat intermédiaire :

Théorème 4 - 1 Tout entier naturel admet au moins un diviseur premier.

Si ce nombre, appelons-le n, est premier, tout va bien.

Sinon, l'ensemble de ses diviseurs étant non vide (n est dedans) et borné (par 1 et lui-même), il admet un plus petit élément $p \neq 1$. Ce nombre n'est pas divisible par un autre, sinon ce nombre plus petit que p divisant p diviserait n et alors p ne serait plus le plus petit diviseur de n. Le nombre p est donc premier et voilà.

Théorème 4 - 2 Il y a une infinité de nombres premiers.

Raisonnons par l'absurde et supposons qu'il existe exactement n nombres premiers qu'on nommera p_1, $p_2, \cdots$, p_n et appelons N le nombre

$$N = p_1 p_2 \cdots p_n + 1$$

Il est plus grand que tous les p_i, donc il n'est pas premier d'après notre hypothèse, donc il admet un diviseur premier p qui est donc un des p_i, puisqu'il n'y a qu'eux. Soit i_0 tel que $p = p_{i_0}$. Alors p divise $p_1 p_2 \cdots p_n$. Or il divise N, donc il divise leur différence $N - p_1 p_2 \cdots p_n$, c'est à dire 1, donc $p = 1$ ce qui est absurde puisqu'il est premier. Ainsi, il n'existe pas de plus grand nombre premier.

Il y en a donc une infinité et l'aventure ne fait que commencer.

Comment vérifier qu'un nombre est premier ?

Force brute

Créez une fonction `divise k n` qui teste si l'entier k divise n.

Haskell

```
λ> divise 7 91
True
λ> 7 ‘divise‘91
True
```

Créez une fonction `lesDiviseurs :: Int→ [Int]` qui renvoie la liste des diviseurs de n :

Haskell

```
λ> lesDiviseurs 100
[100,1,2,4,5,10,20,25,50]
```

Déduisez-en un test `estPremier' :: Int→ Bool`

Haskell

```
λ> estPremier (2^10 - 1)
False
```

On peut alors tirer profit de la fonction `Data.List.find` :

Doc Haskell

```
find :: (a→ Bool)→ [a]→ Maybe a
The find function takes a predicate and a list and returns the first element in the list matching the
    predicate, or Nothing if there is no such element.
```

pour déterminer une fonction `prochainPremier` qui calcule le nombre premier qui suit l'argument :

Haskell

```
λ> prochainPremier' (2^10 - 1)
Just 1031
```

Cependant, l'efficacité de cette fonction laisse à désirer...Comment peut-on améliorer `lesDiviseurs` ?

Mais si on ne cherche que les nombres premiers, a-t-on besoin de tous les diviseurs ?

Étudions le problème pour être plus efficaces

Observons les diviseurs de 1321 par exemple : si aucune des 1319 divisions ne « tombe juste », alors on pourra dire que 1321 premier. Et puis d'abord, il est impair, il ne semble pas être divisible par 3, alors pourquoi pas...

diviseur	2	3	4	5	6	7	8	9	10	11	12	13
quotient	660,5	440,3	330,3	264,2	220,2	188,7	165,1	146,8	132,1	120,1	110,1	101,6
diviseur	14	15	16	17	18	19	20	21	22	23	24	25
quotient	94,36	88,07	82,56	77,71	73,39	69,53	66,05	62,90	60,05	57,43	55,04	52,84
diviseur	26	27	28	29	30	31	32	33	34	35	36	37
quotient	50,81	48,93	47,18	45,55	44,03	42,61	41,28	40,03	38,85	37,74	36,69	35,70

Le tableau est incomplet : il reste encore à essayer les quotients de 38 à 1320, mais cela aurait été mauvais pour la planète.
Tout d'abord, nous n'avons pas trouvé de quotient entier en ce début d'enquête. Nous observons de plus que si la suite des diviseurs est croissante, celle des quotients est décroissante. Vous avez sûrement remarqué qu'à partir de 37, les quotients sont inférieurs aux diviseurs donc nous pouvons nous arrêter là : si pour un diviseur supérieur à 37, on trouvait un quotient entier q, alors en divisant 1321 par q, qui est inférieur à 37, on devrait se trouver au début de notre liste de diviseurs et ce diviseur serait entier. Le problème, c'est qu'aucune division par un entier inférieur à 37 n'a donné de quotient entier donc on peut s'épargner les 1283 divisions restantes.

Mine de rien, nous venons de franchir un grand pas dans la théorie des nombres : pour tester si l'entier 1321 est premier, il nous a suffit d'effectuer 36 divisions [a]. Deux problèmes se posent maintenant : pourquoi 36 et pouvons-nous généraliser ce résultat aux autres entiers ?
Un petit Joker : $\sqrt{1321} \approx 36,3...$
Reprenons la propriété 4 - 1 page 73. Tout naturel n admet un plus petit diviseur p qui est premier. Écartons le cas où n est premier et donnons un nom aux nombres qui restent :

Définition 4 - 3 Un entier naturel autre que 1 qui n'est pas premier est dit **composé**.

L'entier n étant composé, il s'écrit donc $n = pq$, avec q un entier supérieur à p (car p est le plus petit diviseur). Ainsi

$$p \leqslant q \quad \text{et donc} \quad p^2 \leqslant pq = n \quad \text{c'est à dire} \quad p \leqslant \sqrt{n}$$

Nous pouvons donc généraliser l'observation précédente

Théorème 4 - 3 Si un entier est composé, alors il admet un diviseur premier inférieur à sa racine carrée.

Inspirez-vous de ce résultat pour encore améliorer les fonctions précédentes.
Vous pourrez utiliser par exemple les fonctions `takeWhile` et `filter`.
Par exemple, il faut 3 secondes pour déterminer le nombre premier qui suit 2^{50} :

Haskell
```haskell
main = print (prochainPremier (2^50))
```

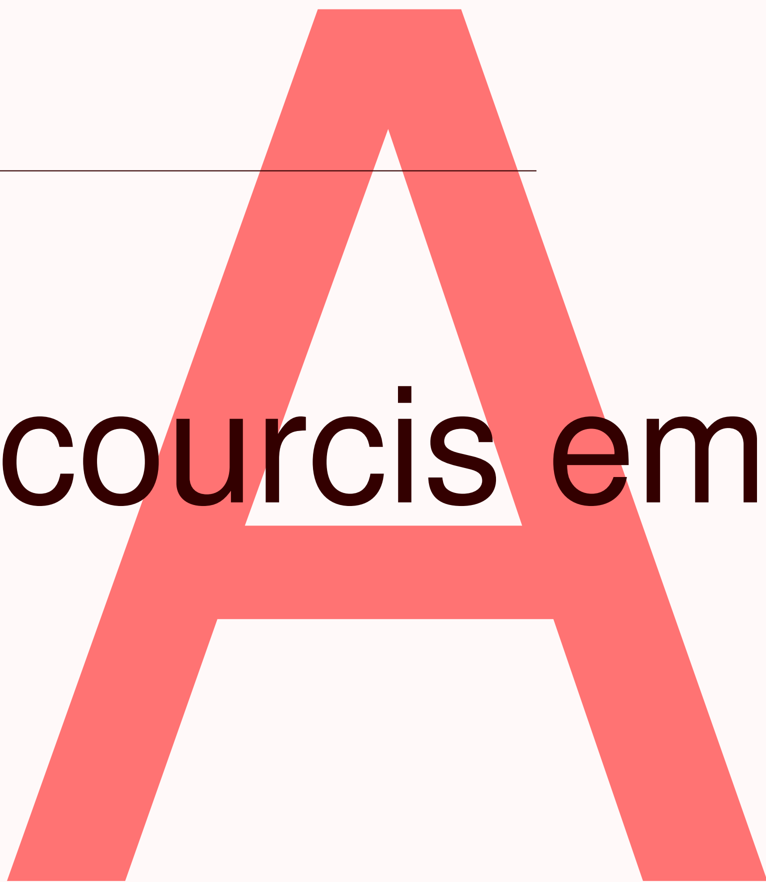
Shell
```
$ ghc premHs.hs -o premHs
$ time ./premHs
Just 1125899906842679

real    0m3.053s
user    0m3.048s
sys     0m0.008s
```

Vous pouvez ensuite vous lancer dans l'écriture d'une fonction qui renvoie la décomposition d'un entier en produit de facteurs premiers sachant que nous admettrons le théorème suivant (mais rien ne vous empêche de vous lancer dans une démonstration...) :

Théorème 4 - 4 Tout entier n supérieur à 2 admet une et une seule (à l'ordre près des termes) décomposition en produit fini de nombres premiers

a. Et encore, nous aurions pu faire mieux comme nous allons le voir tout de suite après.

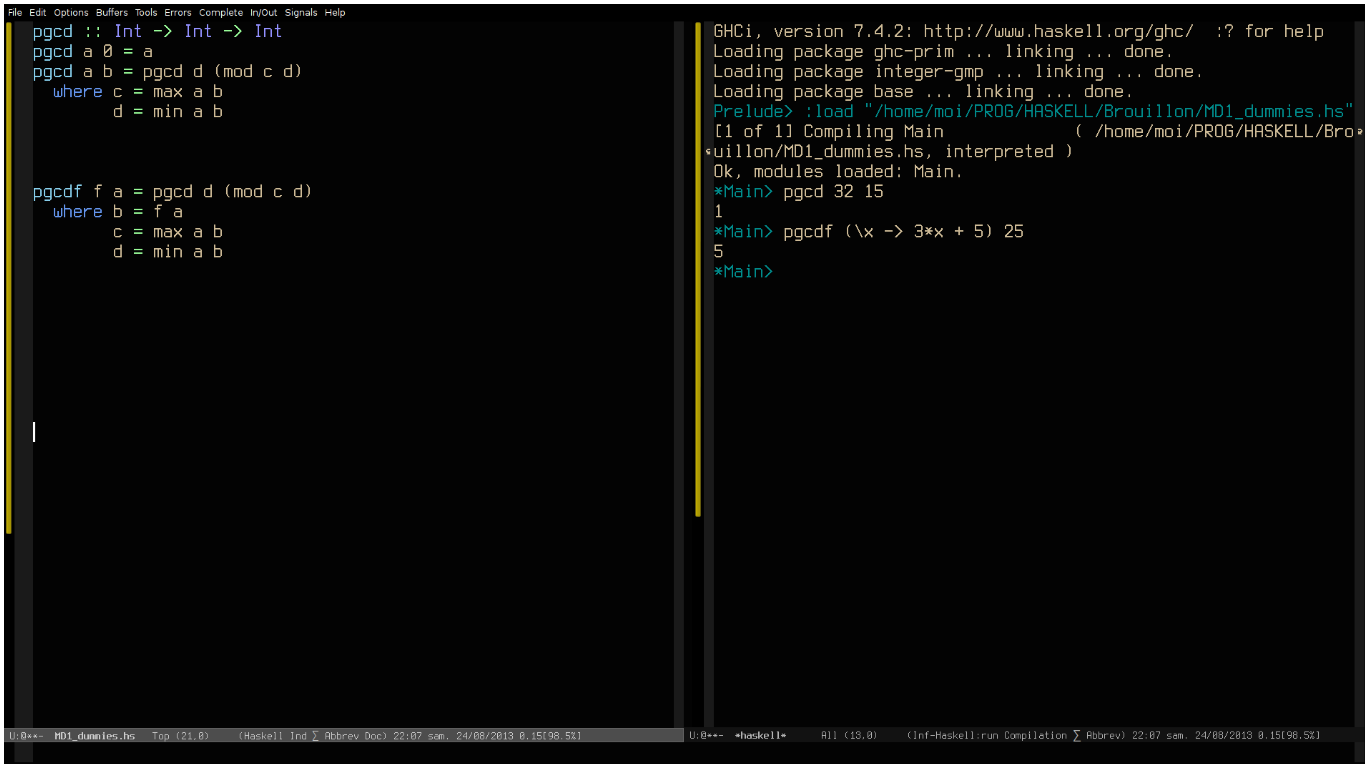
ANNEXE



Raccourcis emacs

Le secret du succès enfin dévoilé

Nous travaillerons avec emacs en mode haskell et nous partagerons notre écran en deux tampons, l'un contenant le code et l'autre GHCi :

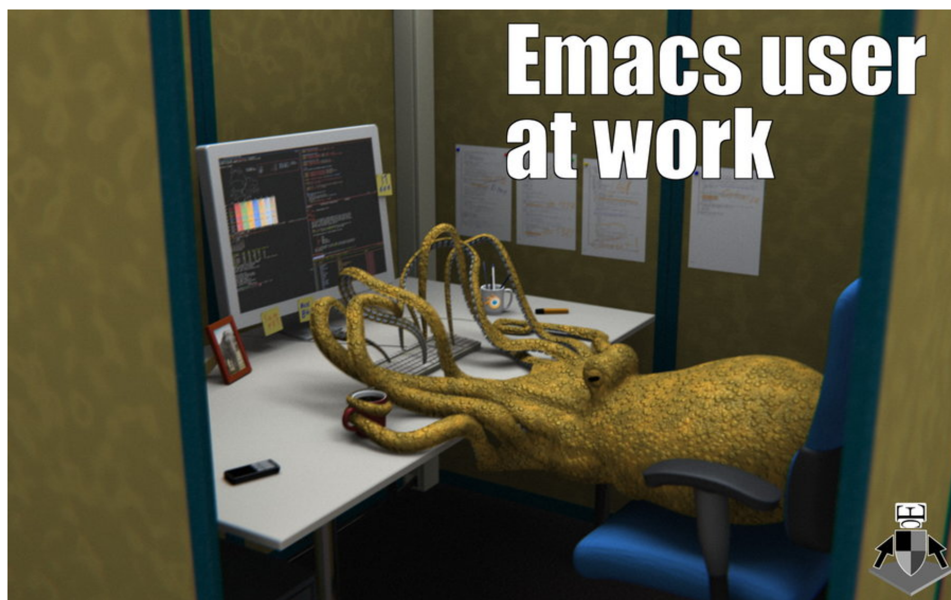


Pour cela, il faut ouvrir un petit shell depuis emacs en tapant **`M-x shell`**, créer un répertoire

Shell

```
$ mkdir ~/Maths/
$ mkdir ~/Maths/INFO1/
$ mkdir ~/Maths/INFO1/MD/
```

pour y placer votre Fichier d'extension **`hs`** : **`C-x C-f ~/Maths/INFO1/MD/dummy.hs`** et lancer GHCI avec **`C-c C-b`**.



DÉPLACEMENTS	
C-v	Pagedown
M-v	Pageup
C-p	Ligne du dessus (previous line)
C-n	Ligne du dessous (next line)
C-b	Caractère précédent (back)
C-f	Caractère suivant (next)
M-b	Mot précédent
M-f	Mot suivant
C-a	Début de ligne
C-e	Fin de ligne
M-a	Début de phrase
M-e	Fin de phrase
M-<	Fin de fichier
M->	Début de fichier
C-l	Curseur en milieu de page
C-u C-v	Curseur en haut de page

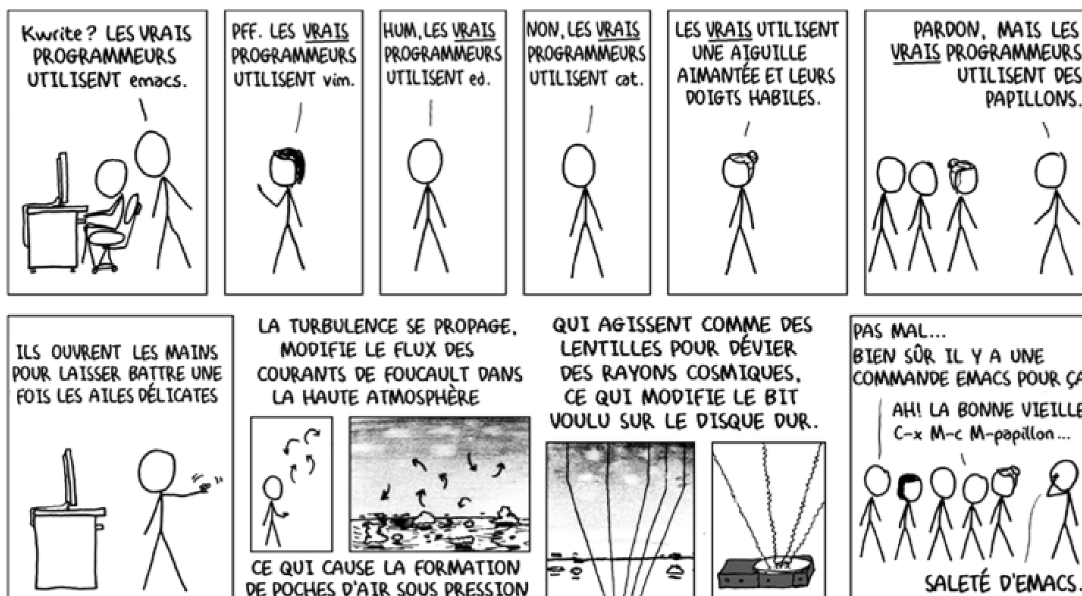
COMMANDES PRINCIPALES (MENU FICHIER)	
C-x C-f	Ouvrir fichier (tab de complétion)
C-x C-s	Sauvegarder fichier
C-x C-c	Quitter Emacs
C-z	Sortie provisoire (récupération possible par %emacs)
C-g	Arrêt d'une commande (ou sortie d'un sous-menu de commande)
ESC ESC ESC	Idem (mais sortie d'un recursive editing level)

ÉDITION	
C-x u	Annulation (ou **C-_**)
C-x z	Répéter
Suppr	Suppression du caractère avant
C-d	Suppression du caractère après
M-suppr	Suppression du mot avant
M-d	Suppression du mot après
C-k	Couper (kill) jusqu'en fin de ligne
M-k	Couper (kill) jusqu'en fin de phrase
C-x k	Couper (kill) le buffer
C-espace	Marquer (début d'un copier/couper)
C-w	Fin d'un couper (kill)
M-w	Fin d'un coller
M-h	Marque le paragraphe
C-x h	Marque la totalité du buffer
C-x C-espace	Coller (pop) global mark
C-y	Coller (yank du killed)
M-y	Passage entre les différents kills (anciens copier, après un **C-y**)
C-s	Recherche après (suivi de **C-w** recherche le mot sous le curseur)
C-r	Recherche avant (suivi de **C-w** recherche le mot sous le curseur)

M-Shift ù	Remplacer
Espace	Pour remplacer l'occurence suivante
Suppr	Pour passer l'occurence sans la remplacer
!	Pour remplacer toutes les occurences
M-u	Mot en majuscule
M-l	Mot en minuscule
M-c	Mot en capitales

TAMPONS (BUFFERS)	
C-x C-b	Liste des buffers (fichiers ouverts)
C-x s	Sauvegarde les buffers (pose la question)
C-x 0	Supprime cette fenêtre
C-x 1	Supprime les autres fenêtre
C-x 2	Divise la fenêtre en 2 verticalement
C-x 3	Divise la fenêtre en 2 horizontalement
C-x o	Passage d'un écran à l'autre (other)
C-x ^	Aggrandir la fenêtre
C-v	Pagedown dans l'autre fenêtre (ESC **C-v** en cas de non méta)
C-x 4 C-f	nomFichier Ouverture de nomFichier dans une fenêtre en bas
C-x 4 b	Fermeture fenêtre

DIVERS	
C-u chiffre	Itération d'une action (ex : **C-u 5 C-n** Descend de 5 lignes)
C-x	Commande suivi d'un seul caractère
M-x	Commande à partir d'une commande texte (tab de complétion)
C-h ?	Aide générale
C-h c nomCommande	Description de la commande
C-h k nomCommande	Aide sur la commande
C-h f nomFonction	Description d'une fonction
C-h a nom	(Apropos) : liste les commandes contenant le nom
C-h i	Lire les infos (Manuels en-ligne)
ESC !	Commande shell



xkcd.com, traduction P. Gambette

ANNEXE B

Mon livre de CM1

Voici quelques pages d'un livre de CM1 des années 1970...Cela constitue une bonne base d'exercices ;-)

Exercices et problèmes

1. On a l'ensemble A = { 1 ; 3 ; 5 ; 7 ; 9 }. Comment peut-on appeler cet ensemble ?
2. Ecris l'ensemble B des voyelles de notre alphabet.
3. A-t-on le droit de dire que les noms brochet, carpe, grenouille sont des éléments d'un ensemble de noms de poissons ?
4. Forme un ensemble C de mots qui commencent par s, un ensemble D de mots qui se terminent par un u.
5. Ecris l'ensemble E des rois qui ont régné en France entre les années 1600 et 1780.
6. Regarde sur ton dictionnaire ce qu'on appelait une « merveille du monde » Quels étaient les éléments de cet ensemble ?
7. Regarde sur un atlas quels sont les éléments de l'ensemble des Etats qui ont une frontière commune avec la France.
8. Quand on cite les éléments d'un ensemble **on n'a pas le droit de citer deux fois le même élément.** Jean a écrit F = { Louis XIV ; Louis XV ; le roi Soleil } Quelle faute a-t-il commise ?

9. Les ensembles A = { m ; p ; r ; l ; k } et B = { p ; l ; k ; r ; m } sont-ils égaux ?
10. Le ensembles C = { 1 ; 7 ; 3 ; 6 } et D = { 6 ; 7 ; 3 ; 1 ; 5 } sont-ils égaux ?
11. On sait que E = { 13 ; 20 ; 12 ; 16 } que F = { 20 ; 16 ; x ; 12 } et que E = F .Trouve x.

12. On donne A = { 25 ; 36 ; 59 ; 41 ; 17 ; 10 } .Quel est l'ensemble B des nombres de l'ensemble A dans lesquels on trouve le chiffre 1 ? Quel est l'ensemble C des nombres de l'ensemble A dans lesquels on trouve le chiffre 8 ?

13. Les éléments d'un ensemble B sont choisis parmi ceux d'un univers A :

éléments de A	x	y	z	t	u	v
pour B, numéros des éléments de A	1	0	0	1	1	1

Quels sont les éléments de B ?
Ecris sur une feuille, comme sur le livre :
Mets en évidence les ensembles A et B.

+ x + y
+ t + u + z + w
+ v

14. Les éléments d'un ensemble D sont choisis parmi ceux d'un univers C on sait que C = { ◇ ; △ ; ▭ ; ⊠ } et que D = { ⊠ ; △ }
complète le tableau :

éléments de C.	
pour D numéros des éléments de C	

Sur un même dessin, obtiens un schéma représentant les ensembles C et D.

Exercices et problèmes

1. Voici les ensembles A = { 2 ; 9 ; 6 ; 4 ; 7 } et B = { 2 ; 4 ; 6 ; 8 ; 10 }
Quelle est leur intersection C ? Fais un diagramme.

2. Tu récites la suite des nombres entiers naturels : 0 ; 1 ; 2... A est l'ensemble des nombres que tu dis après 197, mais avant 205. B est l'ensemble des nombres que tu dis après 200, mais avant 210. Ecris les éléments de A, de B, de leur intersection C. Fais un diagramme.

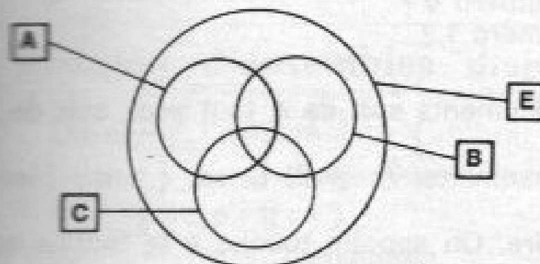
3. On donne l'univers des noms d'animaux E = { chat ; brochet ; tigre ; serpent ; poule ; vache ; mouton ; éléphant ; aigle }

éléments de E	c	b	t	s	p	v	m	e	a
pour A, N^{os} des éléments de E	1	0	1	0	0	1	1	1	0
pour B, N^{os} des éléments de E	1	0	0	0	1	1	1	0	0

Ecris les éléments de A. Qualifie les par un caractère commun (pense aux pattes).
Ecris les éléments de B et qualifie les (as-tu déjà visité une ferme ?)
Quelle est l'intersection C des ensembles A et B ? Qualifie les éléments de C.
Fais un diagramme pour les ensembles E, A, B, C

4. On te donne un univers d'enfants **E** = { Bernard ; François ; Christophe ; Didier ; Alain ; Régis ; Michel ; Pierre ; Georges ; Xavier }. Ceux de ces enfants qui lisent « Tintin » forment un ensemble **A.** Ceux de ces enfants qui lisent « Pilote » forment un ensemble **B.** Ceux de ces enfants qui lisent « Spirou » forment un ensemble **C.**

éléments de E	B	F	C	D	A	R	M	P	G	X
pour **A,** N^{os} des éléments de E	1	1	1	0	0	1	1	1	0	0
pour **B,** N^{os} des éléments de E	0	1	1	1	1	1	1	0	0	0
pour **C,** N^{os} des éléments de E	0	1	1	1	0	1	0	1	1	0



Ecris les éléments de : **A,B,C.**
Ecris les éléments de :
H, intersection de **A** et **B.**
I, intersection de **A** et **C**
J, intersection de **B** et **C.**
Y a-t-il des éléments communs à **A, B, C** ?
Leur ensemble est K.
Place les prénoms sur le diagramme ci-contre.

5. On te donne l'univers E = { Canada ; Etats-Unis ; Mexique ; Colombie ; Vénézuela ; Brésil ; Pérou ; Bolivie } . Consulte une carte de l'Amérique.
Quels numéros faut-il associer aux éléments de E pour obtenir l'ensemble A des Etats baignés par l'Océan Pacifique, pour obtenir l'ensemble B des Etats baignés par l'Océan Atlantique ?
Quelle est l'intersection C de A et B ? Fais un diagramme pour E, A, B, C.

6. On te donne l'univers de lettres E = { a ; b ; c ; d ; e ; f ; g ; h ; i }
Essaie de trouver les éléments d'un ensemble A, d'un ensemble B pour que le cardinal de A soit 3, le cardinal de B, 5, le cardinal de l'intersection C de A et B, 1. Toujours avec cardinal A = 3, cardinal B = 5, est-il possible d'avoir cardinal C = 2 ou bien cardinal C = 3, ou bien cardinal C = 4 ? Fais un diagramme dans chaque cas possible.

7. Tu as un jeu de 32 cartes. Qualifie l'intersection de l'ensemble des cartes rouges et de l'ensemble des cartes noires.

8. Prends une boîte de blocs logiques. Regarde les blocs un à un et mets dans un sac les grands triangles bleus au fur et à mesure que tu les trouves. Reprends tous les blocs, entoure d'une ficelle l'ensemble A des grands triangles et l'ensemble B des triangles bleus. Vois-tu l'ensemble des triangles bleus et grands ?

Exercices et problèmes

I

1. On donne l'ensemble E des chiffres $E = \{0 ; 1 ; 2 ; 3 ; 4 ; 5 ; 6 ; 7 ; 8 ; 9\}$
Ecris le sous-ensemble A des chiffres impairs. Forme le sous-ensemble B complémentaire de A par rapport à E. Comment appeler les éléments de B ?

2. On donne l'ensemble E des lettres de l'alphabet. Ecris les éléments du sous-ensemble A des consonnes. Forme le sous-ensemble B complémentaire de A par rapport à E. Comment appeler les éléments de B ?

3. On donne $E = \{1 ; 2 ; 3 ; 4 ; 5 ; 6 ; 7 ; 8 ; 9\}$, $A = \{1 ; 3 ; 7 ; 9\}$, $B = \{2 ; 4 ; 6 ; 8\}$. $C = \{1 ; 3 ; 5 ; 7 ; 9\}$, $D = \{2 ; 4 ; 5 ; 6 ; 8\}$. Dis si les sous-ensembles suivants sont complémentaires par rapport à E

A et B ; C et D ; A et D ; B et C

II

4. Tu as l'ensemble des blocs logiques d'une boîte et trois sacs. Dans le premier sac tu mets les blocs triangulaires ; dans le deuxième sac tu mets les blocs non triangulaires. Dans le troisième sac, y a-t-il des blocs à mettre ?

5. Tu as l'ensemble des blocs logiques d'une boîte. Montre un bloc carré et **non** bleu. Prends les blocs un à un et mets tous les blocs carrés et non bleus dans un sac.
Reprends tous les blocs ; forme sur ta table l'ensemble des blocs carrés, l'ensemble des blocs non-bleus ; entoure-les chacun d'une ficelle. Où sont placés les blocs carrés et non-bleus ?

6. Recommence comme au numéro 5 avec les blocs **non**-rouges **et non**-petits.

7. Recommence comme au numéro 5 avec les blocs carrés **ou non**-bleus.

8. Recommence comme au numéro 5 avec les blocs **non**-rouges **ou non**-petits.

9. Recommence comme au numéro 5 avec les blocs non-jaunes et non-ronds, puis avec les blocs non-jaunes ou non-ronds.

III

10. Réponds par oui ou bien par non aux questions suivantes. Peux-tu calculer (9-6), (6-3), (3-6), (6-6), (8-1), (2-7), (4-4), (123 465-0), (100 001-99 999), (4350-4035) ?

11. Calcule les nombres représentés par des lettres :

$2 + x = 9$ $\quad 15 = a + 4$ $\quad 8 + 8 = m$
$y = (7 - 3)$ $\quad (17 - q) = 9$ $\quad (18 - 9) = n$
$1 = (6 - z)$ $\quad 0 = (c - 5)$ $\quad p + 5 = 10$
$(t - 8) = 3$ $\quad d + d = 8$ $\quad 14 - q = 7$

12. A un couple de nombres, on fait correspondre un nombre par une addition ou bien une soustraction. Mets dans les cases les signes convenables :
$(8 ; 6) \to \square \to 14$ $\quad (13 ; 10) \to \square \to 3$ $\quad (15 ; 7) \to \square \to 8$ $\quad (3 ; 3) \to \square \to 6$ $\quad 17 ; 9 \to \square \to 8$
Si on te propose $(5 ; 0) \to \square \to 5$, quels signes peux-tu mettre ?
Si on te propose $(0 ; 5) \to \square \to 5$, as-tu le choix ?

13. Mets dans chacun des cas suivants V (vrai) ou bien F (faux) :
$(12 - 4) = 8$; $(6 - 5) < 9$; $4 > (12 - 6)$; $(10 - 9) < 2$; $(15 - 8) > 5$
$(14 - 6) > 10$; $9 = (12 - 5)$; $0 < (11 - 3)$; $5 > (13 - 9)$; $9 < (16 - 8)$

14. Quelqu'un dit : « (8 — 6) est plus grand que 3 » ; a-t-il raison ?
Quelqu'un dit : « Il est faux d'affirmer que (8 — 6) est plus grand que 3 » ; a-t-il raison ?

ercices et problèmes

1. Voici un ensemble de fleuves F = { Seine ; Loire ; Rhône ; Garonne } et un ensemble de cours d'eau : { Marne ; Allier ; Saône ; Oise ; Somme }. Représente ces ensembles et trace les traits fléchés signifiant : «....a pour affluent...» Trace le tableau représentant tous les couples (fleuve ; cours d'eau). Indique comme au paragraphe I ceux de ces couples qui associent un fleuve et un de ses affluents.

2. On a représenté un tableau indiquant les années de naissance des enfants Lesage. Construis un schéma qui te donnera les mêmes renseignements que ce tableau. Complète :

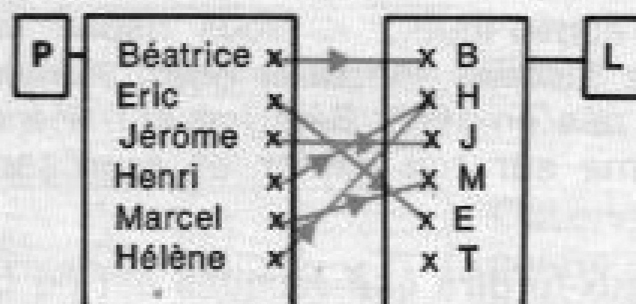
...... est né en 1960 ;

...... est né en 1958 ;

Olivier est né en .....

	Gérard	Henri	Nathalie	Olivier	Nicole
1958	1	0	0	0	0
1960	0	1	0	1	0
1963	0	0	0	0	1
1968	0	0	1	0	0

3. On a représenté un schéma, essaie de l'interpréter. Fais un tableau donnant les mêmes renseignements :



4. Représente le schéma de l'ensemble des villes : V = { Nice ; Paris ; Lille ; Lyon ; Toulouse } puis le schéma de l'ensemble des départements D = { Seine ; Rhône ; Nord ; Alpes Maritimes ; Haute-Garonne }. Trace les traits fléchés que tu veux de chaque ville à un département convenable. Peux-tu donner une interprétation aux traits que tu as tracés ?

6	11
7	12

26. COMPOSITION DES RELATIONS. RELATIONS RECIPROQUES

I Composition des relations

1. Relations non numériques

Pour les vacances de Noël, un groupe d'amis parisiens se sépare. Jean va à Lille, Pierre à Lyon, Henri à Rennes, Alain à Bordeaux et Marc à Toulouse. Jean s'est renseigné au bureau de la gare sur les horaires de départ des trains.

Il a représenté un schéma indiquant l'heure de départ de chacun.

Que signifient les traits fléchés noirs ? les traits fléchés bleus ? les traits fléchés gris ?

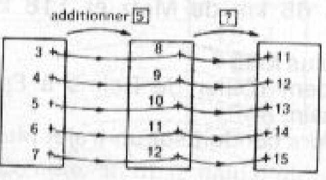
Ecris les couples (garçon ; ville) dont les termes se correspondent.

Ecris les couples (ville ; heure) dont les termes se correspondent.

Ecris les couples (garçon ; heure) dont les termes se correspondent. Que remarques-tu ?

Sur le dessin, comment Jean est-il relié à « 9 ; 05 ». (Il y a deux possibilités, soit un trait fléché bleu, soit un trait fléché noir puis fléché gris.)

2. Relations numériques

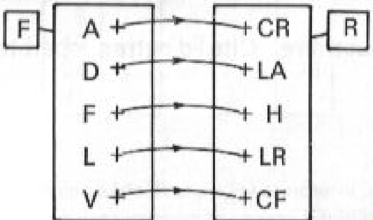


Olivier a dessiné avec des ensembles de nombres un schéma dont les traits fléchés sont semblables à ceux que nous avons tracés dans le n° 1.

Interprète les traits fléchés noirs, les traits fléchés bleus, les traits fléchés gris.

II Relations réciproques

1) Relations non numériques



Chacune des filles : Anne (A), Denise (D), Françoise (F), Lydie (L), Véronique (V) a récité une fable de La Fontaine.

L'ensemble des filles est représenté par un schéma (chaque fille est désignée par l'initiale de son prénom)

Chaque fable est désignée par des initiales dans le schéma de l'ensemble R.

Il y a : le Corbeau et le Renard (CR), le Loup et l'Agneau (LA), le Héron (H), le Lion et le Rat (LR), et la Cigale et la Fourmi (CF).

Les traits fléchés noirs signifient... a récité...

Ecris les couples dont les termes sont reliés par les traits fléchés noirs.

Qui a récité « le Héron » ? Qui a récité « la Cigale et la Fourmi » ?

Tu peux répondre : « Françoise a récité « Le Héron », à la première question.

Mais tu peux donner le même renseignement par la phrase :

« Le Héron » a été récité par Françoise.

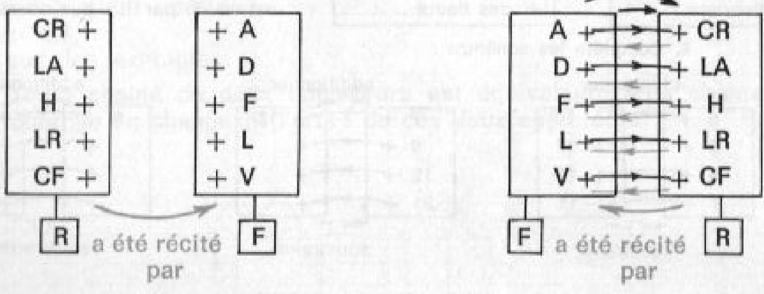
Réponds à la seconde question de deux manières.

Donne d'autres exemples.

Trace sur le tableau de gauche les traits fléchés signifiant :... a été récité par...

Tu aurais pu donner les mêmes renseignements sur le premier tableau comme cela a été fait sur le schéma de droite.

Ecris les couples : (récitation-fille) dont les termes sont reliés par les traits fléchés bleus.



2. Ecris en base trois, en base quatre, en base cinq, en base six :

a) le nombre de doigts de tes mains ;
b) le nombre de crayons de ta boîte de crayons de couleurs ;
c) le nombre de tubes de ta boîte de peinture.

3. Les nombres suivants sont-ils écrits ?

a) 2 1 3 en base trois ? pourquoi ?
b) 2 4 2 en base trois ? en base quatre ? pourquoi ?
c) 3 3 5 en base trois ? en base quatre ? en base cinq ? pourquoi ?

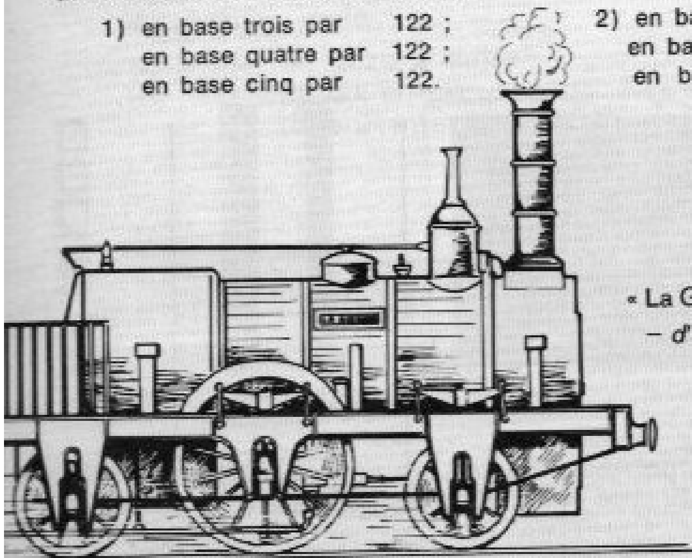
4. J'indique le nombre qui précède et celui qui suit :

	nombre qui précède	nombre donné	nombre qui suit
1) base trois		1 1 1	
		1 2 1	
		2 0 1	
		2 1 1	
		2 2 1	
2) base quatre		1 2 3	
		1 3 2	
		2 0 1	
		2 1 0	
		2 1 3	
		2 2 2	
		2 3 1	
		3 0 0	
3) base cinq		2 3 4	
		2 4 2	
		3 0 0	
		3 0 3	
		3 1 1	
		4 4 3	

5. Je dessine les billes dénombrées :

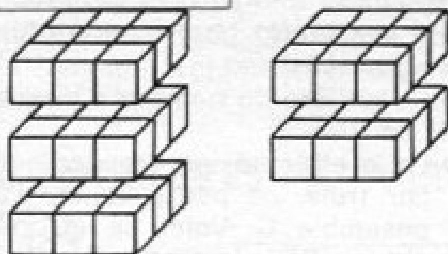
1) en base trois par 122 ;
en base quatre par 122 ;
en base cinq par 122.

2) en base trois par 200 ;
en base quatre par 102 ;
en base cinq par 33.



« La Gironde » - Creusot - 1839
– *d'après cl. « La vie du rail »*

III Sous-unités



Voici un ensemble A de petites plaques.
En utilisant le tableau, écris le cardinal de A en prenant la petite barre comme unité-repère.

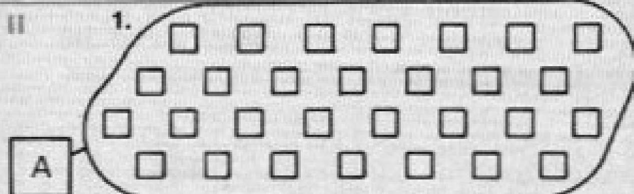
cubes	petites plaques	petites barres	petits cubes
1	2	0	

↑ position de la virgule

Pour indiquer l'absence de petite barre, on met un zéro dans la colonne des petites barres.
On écrit : en petites barres, le cardinal de l'ensemble de petites plaques est exprimé par l'écriture 120.
Ecris le cardinal de A en prenant le petit cube comme unité-repère.

Exercices et problèmes

II

1. 

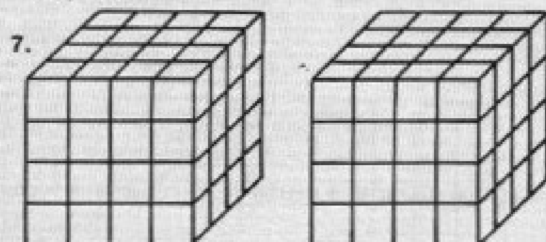
On a représenté un ensemble A de petits cubes. Exprime en base quatre le cardinal de A en prenant comme unité : la petite barre, la petite plaque, le grand cube.

2. Exprime en base cinq le cardinal de l'ensemble A (exercice n° 1) en prenant comme unité : la petite barre, la petite plaque, le grand cube.
3. Exprime en base six le cardinal de l'ensemble (exercice n° 1) en prenant comme unité : la petite barre, la petite plaque, le grand cube.
4. A, B, C, sont des ensembles de petits cubes. On a effectué les groupements en base quatre, les résultats obtenus ont été portés sur le tableau. Exprime le cardinal de chacun de ces ensembles en prenant comme unité repère le petit cube, la petite barre, la petite plaque, le grand cube.

ensembles	cubes	petites plaques	petites barres	petits cubes
A	2	1	0	0
B		2	0	3
C			1	3

5. Dessine un ensemble de croix dont le cardinal est l'unité repère étant le groupement de cinq croix, exprimé par l'écriture 12.
6. A est un ensemble de croix. Un groupement remplace cinq croix, un grand groupement remplace cinq groupements.
Exprime le cardinal de A en base cinq en prenant pour unité-repère le grand groupement, si le cardinal de A (l'unité-repère étant la croix) est 10 ; 213 ; 2 ; 100.

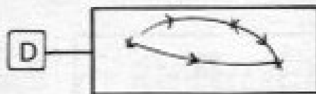
III

7. 

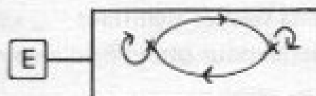
Le matériel utilisé est le matériel base quatre.
Voici un ensemble de grands cubes. Exprime en base quatre le cardinal de cet ensemble en prenant pour unité le grand cube, la petite plaque, la petite barre, le petit cube.

III Propriété de la relation : «... est diviseur de... » dans l'ensemble des nombres entiers

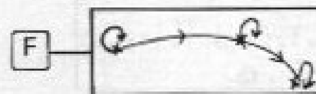
- Choisis deux nombres. Représente l'ensemble de ces deux nombres. Peux-tu relier les points qui les représentent par un trait signifiant "...est diviseur de ..."? (Cela dépend des nombres choisis).



On a représenté un ensemble de trois nombres et des traits fléchés signifiant "... est diviseur de ..." . Ce schéma est-il complet ?
Trouve un ensemble de nombres qui conviennent.



On a représenté un ensemble de deux nombres et des traits fléchés. Les traits ne peuvent pas avoir pour signification "... est diviseur de ...". Pourquoi ?
Supprime un trait pour que le schéma puisse avoir cette signification. Trouve des couples convenables de nombres.



On a représenté un ensemble de nombres et des traits fléchés signifiant : "... est diviseur de ..." Ce schéma n'est pas complet. Pourquoi ? Complète-le. Trouve un ensemble convenable de nombres.

IV Recherche des diviseurs d'un nombre

Recherchons tous les diviseurs du nombre 12. Nous savons que ce sont des nombres inférieurs ou égaux à 12. Tu dois donc essayer tous les nombres.
L'ensemble $\{1, 2, 3, 4, 6, 12\}$ est appelé l'ensemble des diviseurs du nombre 12.

Tableau

12												
11												
10												
9												
8												
7												
6												
5												
4	1	1	0									
3	1	0	1									
2	1	1	0									
1	1	0	0									
↖	1	2	3	4	5	6	7	8	9	10	11	12

est diviseur de

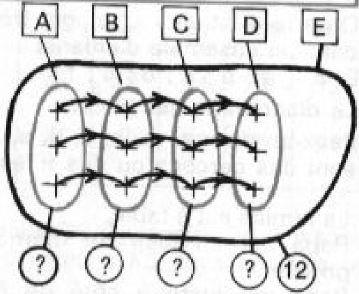
Complète le tableau en écrivant des "1" dans les cases correspondant aux couples dont le premier terme est un diviseur du second, 0 dans les autres cases.
Ce tableau permet de lire directement les diviseurs d'un nombre. Par exemple, en suivant la bande de 12 on voit que 1,2,3,4,6,12 sont des diviseurs de 12. Quels sont les diviseurs de 8, 9, 10,11 ?
Certains nombres n'ont pour diviseur que 1 et eux-mêmes. Fais-en la liste.

cahiers ?

46. DEFINITION DE LA DIVISION DES NOMBRES NATURELS

I Partition d'un ensemble en sous-ensembles disjoints de même cardinal

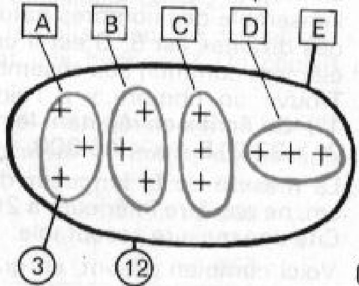
- On veut partager 12 bonbons entre quatre enfants.
  Chacun doit avoir le même nombre de bonbons que les autres.
  Cela revient à partager un ensemble E de cardinal 12 en quatre sous-ensembles de même cardinal.
  On attribue successivement un élément à chaque sous-ensemble comme l'indiquent les traits fléchés noirs.



(1)

Les bonbons distribués sont successivement $(4 \times 1) = 4$, $(4 \times 2) = 8$, $(4 \times 3) = 12$. 4, 8, 12 sont des **multiples de 4.**
Le partage est terminé.
On convient d'écrire $(12 : 4) = 3$.
Au couple (12 ; 4) correspond le nombre 3 par une opération qui s'appelle **division.**
Le premier terme du couple (12 ; 4) est le dividende, le second terme est le diviseur. 3 est le quotient exact de 12 par 4.

- On a une pile de 12 cahiers.
  Il faut distribuer 3 cahiers par élève. Combien d'élèves peut-on servir ?
  Tout à l'heure, on connaissait le nombre de sous-ensembles. On cherchait le cardinal de chacun.
  Maintenant, on connaît le cardinal de chaque sous-ensemble. On cherche le nombre de sous-ensembles. On trace le schéma.



(2)

- Les cahiers distribués sont successivement
  $(3 \times 1) = 3$; $(3 \times 2) = 6$; $(3 \times 3) = 9$; $(3 \times 4) = 12$
  La distribution est terminée.
  3, 6, 9, 12 sont des **multiples de 3**
  On écrit $(12 : 3) = 4$.
- Regarde le premier schéma de la leçon : « Définition de la multiplication des nombres naturels ». Ce schéma ressemble au schéma (2) que nous avons dessiné aujourd'hui.

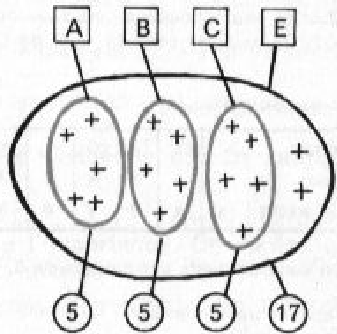
La division exacte $(12 : 3) = 4$ est liée à la multiplication : $12 = (3 \times 4)$.

- A quelles multiplications sont liées les divisions :
  (12 : 4) ; (15 : 3) ; (18 : 2) ?

II Division avec reste

On veut distribuer 17 cerises entre 3 enfants. Les règles du jeu sont les suivantes :

- Chacun doit avoir le même nombre de cerises que les autres.
- Chacun doit avoir le plus possible de cerises.



Peut-on trouver x tel que $17 = (3 \times x)$?
Non, car 17 n'est pas multiple de 3.
L'écriture (17 : 3) ne représente pas un nombre naturel.
Complète le tableau :

nombre de cerises reçues par chaque enfant	1	2		......
nombre de cerises distribuées	3	6	.	......

Que constates-tu ?
$3 \times 5 < 17$; tu peux donner 5 cerises par enfant.
$3 \times 6 > 17$; tu ne peux pas donner 6 cerises par enfant.
$3 \times 5 < 17 < 3 \times 6$.
Tu peux aussi écrire : $17 = (3 \times 5) + 2$ et $2 < 3$.
5 est le quotient entier de 17 par 3. On note $(17 \div 3) = 5$.
Tu donnes 5 cerises à chacun et il te reste 2 cerises.
Au couple (17 ; 3), tu as fait correspondre le couple (5 ; 2).
dividende diviseur quotient reste
Quand tu écris $17 = (3 \times 5) + 2$, écris bien le diviseur **avant** le quotient pour pouvoir le reconnaître et n'oublie pas l'inégalité $2 < 3$ car le reste doit être inférieur au diviseur.

un séjour de 11 jours.
La longueur d'un rouleau de fil de fer mesure 25 m. Quelle est la mesure de la longueur de 9 rouleaux ?
Quel est le prix de 9 disques à 21 F l'un ?

48. DIVISION DES NOMBRES NATURELS : LE DIVISEUR A UN CHIFFRE. LE QUOTIENT EST QUELCONQUE

I Division et addition

Cas de deux quotients exacts.

Complète le tableau.

x	y	z	(x : z)	(y : z)	(x : z) + (y : z)	x+y	(x+y) : z
12	16	4					
28	14	7					
45	36	9					

Que remarques-tu ?

Cas d'un seul quotient exact.

Complète le tableau.

x	y	z	(x ÷ z)	(y : z)	(x ÷ z) + (y : z)	x+y	(x+y) ÷ z
35	40	8					
29	18	6					
31	10	5					

Que remarques-tu ?

Cas où il n'y a pas de quotient exact.

As-tu le droit d'écrire (40 ÷ 6) + (9 ÷ 6) = (49 ÷ 6) ?
non 6 + 1 ≠ 8

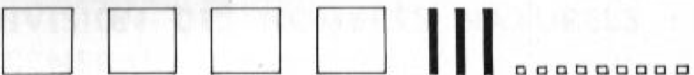
Applications

86 : 2 = (80 : 2) + (6 : 2) = 40 + 3 = 43.
57 ÷ 5 = (50 : 5) + (7 ÷ 5) = 11.
124 : 4 = (120 : 4) + (4 : 4) = 31.
42 ÷ 3. As-tu le droit d'écrire (40 ÷ 3) + (2 ÷ 3) ? Non.
Trouve une autre idée.
96 : 2. On a le droit d'écrire (90 : 2) + (6 : 2). Trouve une autre idée, pour faire une opération plus simple que 90 : 2 ; pense à 80.

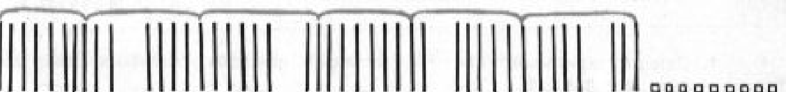
Manipulation

On cherche le quotient entier de 439 par 6.

Représente 439 avec du matériel : petites plaques, petites barres, petits cubes (on pourrait choisir aussi des billets, des pièces).



1. Peux-tu faire 6 sous-ensembles de plaques de même cardinal ?
non.
Remplace les plaques par des barres.



2. Peux-tu faire 6 sous-ensembles de barres de même cardinal ?
oui.
Chaque sous-ensemble sera de 7 barres.
Il reste une barre qu'on ne peut partager. Remplace-la par des cubes.

3. Peux-tu faire 6 sous-ensembles de cubes de même cardinal ?
Oui, chaque sous-ensemble est de 3 cubes. Il reste 1 cube.
Complète : 439 = (6 × ☐) + ☐, ☐ < ☐

Calcul :

Tu cherches le quotient entier de 439 par 6. Quel est le nombre de chiffres de ce quotient ?

6 × 1 = 6 6 × 10 = 60 6 × 100 = 600
6 × 10 < 439 < 6 × 100 Le quotient a deux chiffres.

Tu cherches d'abord le chiffre des dizaines, ensuite le chiffre des unités.

Le nombre des dizaines de 439 est 43. Tu trouves le chiffre des dizaines du quotient en cherchant le quotient entier de 43 par 6.

Ce quotient est 7.

Il reste une dizaine qu'on remplace par 10 unités.

Le nombre des unités à partager est 19.
19 ÷ 6 = 3. Il reste une unité.

disposition pratique détaillée

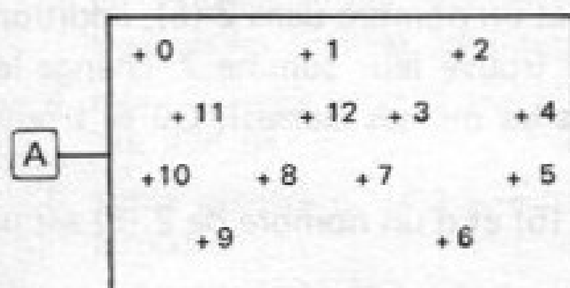
439	6
42	73
19	
18	
1	

d'ordinaire on n'écrit pas 42 et 18

87 CLASSEMENT PAR 5

I Restes dans la division par 5

1. Effectue les divisions de 21 par 5 et 46 par 5.
Tu trouves le même reste : 1
Nous dirons que 46 donne dans la division par 5, le même reste que 21. A est l'ensemble des nombres de zéro à 12. Trace les traits fléchés signifiant "... donne le même reste, dans la division par 5 que ..." Recommence le schéma de façon que les flèches ne se coupent pas.



Est-ce possible ? Quel classement peux-tu effectuer dans l'ensemble A ?

2. Classons les nombres entiers.
Comme nous venons de classer les nombres entiers de zéro à 12, nous allons essayer de classer tous les nombres entiers.
Quels sont les restes possibles dans la division d'un nombre par 5 ?
Traçons 5 colonnes. Nous mettrons dans la première les nombres qui donnent pour reste 0, dans le second ceux qui donnent pour reste 1 et ainsi de suite ...

restes	0	1	2	3	4
	0	1	2	3	4
nombres	5	6	7	8	9
	10	11	12	13	14
	...	...	...	...	...

Continue à placer les nombres suivants de la même manière (par exemple jusqu'à 30).
Quels sont les nombres de la 1 ère colonne ? de la 2ème ? la 3 ème ?

Fais la différence entre deux nombres quelconques de la même colonne. Qu'observes-tu ?

3. Notation et vocabulaire.

André, Jean et Pierre font partie de la même équipe. Pour désigner cette équipe on peut aussi bien dire : équipe d'André, équipe de Jean ou bien équipe de Pierre.

L'ensemble des nombres de la 1ère colonne sera appelé la classe de zéro par 5 et noté $\dot{0}$ (5), ou bien classe de 5 par 5 et noté $\dot{5}$(5), ou bien classe de tout autre nombre de la colonne $\dot{10}$ (5), $\dot{15}$(5) ...) ;
Egalité : $\dot{0}$ (5), $\dot{5}$ (5) désignent le même ensemble, tu peux écrire : $\dot{0}$ (5) $= \dot{5}$ (5) ou bien, plus simplement $\dot{0} = \dot{5}$ (5).
De la même manière les nombres de la deuxième colonne constituent : $\dot{1}$ (5) ou bien $\dot{6}$ (5), ... et $\dot{1} = \dot{6} = \dot{11}$ (5).

4. Comment trouver la classe d'un nombre sans effectuer la division ?
Observe le chiffre des unités des nombres du tableau pour découvrir une règle.
Complète le tableau avec les notations les plus simples ($\dot{0}, \dot{1}, \dot{2}, \dot{3}, \dot{4}$).

Chiffres des unités	0	1	2	3	4	5	6	7	8	9
Classe par 5			$\dot{2}$				$\dot{1}$			

II Addition des classes

Prends un nombre dans $\dot{1}$ (5) et un nombre dans $\dot{2}$ (5), additionne ces deux nombres. Où se trouve leur somme ? change les nombres (en les prenant dans les mêmes classes). Où se trouve la nouvelle somme ?
La somme d'un nombre de $\dot{1}$ (5) et d'un nombre de $\dot{2}$ (5) est un nombre de $\dot{3}$ (5)
Nous écrivons ceci $\dot{1} \dot{+} \dot{2} = \dot{3}$ (5) (nous mettons un point sur +, car ce n'est pas l'addition ordinaire et nous lisons un pointé plus deux pointé égale trois pointé).
Complète les égalités : $\dot{0} \dot{+} \dot{2} = \square$; $\dot{1} \dot{+} \dot{3} = \square$; $\dot{3} \dot{+} \dot{2} = \square$

$\dot{0}$	$\dot{1}$	$\dot{2}$	$\dot{3}$	$\dot{4}$
0	1	2	3	4
5	6	7	8	9
.	.	.	.	.
.	.	.	.	.

$\dot{3} \dot{+} \dot{2} = \dot{0}$

Pour donner tous les résultats on dresse une table d'addition.
Complète la avec les notations les plus simples.
Remarque. Effectue 4+3; peux-tu écrire $\dot{4} \dot{+} \dot{3} = \dot{7}$?
$\dot{4} \dot{+} \dot{4}$; peux-tu écrire $4+4 = 3$?
Il ne faut pas confondre l'addition des classes avec celle des entiers.

$\dot{+}$	$\dot{0}$	$\dot{1}$	$\dot{2}$	$\dot{3}$	$\dot{4}$
$\dot{0}$			$\dot{2}$		
$\dot{1}$			$\dot{3}$	$\dot{4}$	
$\dot{2}$	$\dot{2}$	$\dot{3}$		$\dot{0}$	
$\dot{3}$		$\dot{4}$	$\dot{0}$		
$\dot{4}$					

Multiplication des classes : ce que tu viens de faire pour l'addition, tu peux le faire de la même manière pour la multiplication et dresser une table de multiplication.

III Divisibilité par 5

Quels sont dans le tableau du I les nombres qui sont divisibles par 5 ? (ceux de la 1 ère colonne)
Peux-tu trouver un moyen de savoir quand un nombre est divisible par 5 sans effectuer la division ?

- **Les nombres divisibles par 5 sont ceux qui ont pour chiffre des unités zéro ou bien cinq.**

Exercices

1.

11 · 21 · 9 · 7
· 53 · 19 · 50 · 22 · 45
14 · 28 · 1 · 55

Trace les traits fléchés signifiant : "... donne le même reste dans la division par 5, que.." Classe les nombres de l'ensemble A, d'après les traits fléchés (tu auras soin de refaire un schéma plus propre).

2. Ecris chacun des nombres suivants sous la forme; $(5 \times a) + b$ et $b < a$ puis classe les nombres suivant leur reste dans la division par 5 :
17, 32, 26, 30, 5, 9, 22, 38.

3. En adoptant les notations les plus simples, indique dans quelle classe se trouvent les nombres suivants : 6, 3, 7, 11, 17, 38, 45, 52, 73, 94, 203.

4. Même exercice qu'en 3 avec les nombres : 62, 107, 109, 112, 120, 204, 216, 428, 542.

5. Même exercice qu'en 3 avec les nombres : 6 738, 4 724, 21 894, 26 215, 30 000.

6. Note chacune des classes suivantes sous la forme la plus simple (exemple : $\dot{7} = \dot{2}\,(5)$), $\dot{8}\,(5)$, $\dot{9}\,(5)$, $\dot{13}\,(5)$, $\dot{25}\,(5)$, $\dot{32}\,(5)$, $\dot{47}\,(5)$, $\dot{54}\,(5)$.

7. Complète le tableau suivant en utilisant la table.

a	$\dot{2}$	$\dot{1}$	$\dot{3}$	$\dot{4}$	$\dot{2}$	$\dot{0}$	$\dot{1}$	$\dot{4}$	$\dot{0}$
b	$\dot{0}$	$\dot{4}$	$\dot{3}$	$\dot{2}$	$\dot{3}$	$\dot{2}$	$\dot{4}$	$\dot{2}$	$\dot{2}$
c	$\dot{2}$	$\dot{2}$	$\dot{4}$	$\dot{3}$	$\dot{3}$	$\dot{4}$	$\dot{3}$	$\dot{0}$	$\dot{0}$
$a \dotplus b$									
$(a \dotplus b) \dotplus c$									
$b \dotplus c$									
$a \dotplus (b \dotplus c)$									

Compare les bandes encadrées en bleu.
Trouve une conclusion.
As-tu déjà écrit des égalités du même genre ?

8. Effectue les opérations suivantes : $\dot{2} \dotplus \dot{4} \dotplus \dot{1}\,(5)$; $\dot{3} \dotplus \dot{0} \dotplus \dot{2} \dotplus \dot{3} \dotplus \dot{1}\,(5)$;
$\dot{1} \dotplus \dot{2} \dotplus \dot{3} \dotplus \dot{2} \dotplus \dot{3} \dotplus \dot{4}\,(5)$; $\dot{1} \dotplus \dot{4} \dotplus \dot{4} \dotplus \dot{4} \dotplus \dot{4} \dotplus \dot{2}\,(5)$; $\dot{0} \dotplus \dot{3} \dotplus \dot{2} \dotplus \dot{2} \dotplus \dot{3} \dotplus \dot{4} \dotplus \dot{2} \dotplus \dot{1} \dotplus \dot{3}\,(5)$

9. Complète le tableau suivant en écrivant 1 si le nombre est divisible par 5, 0 sinon.

nombres	27	45	59	76	97	200	121	228	553	610	1225	3004
divisibilité par 5												

10. On remplace autant qu'on peut 48 pièces de 1 centime par des pièces de 5 centimes. Combien reste-t-il de pièces de 1 centime à la fin de l'opération?
Même question quand on remplace 70 pièces de 1 franc par des pièces de 5 francs.
Même question quand on remplace 40 billets de 10 francs par des billets de 50 francs.

Nombres	39	78	40	125	900	1652	9874	10001	25373
Divisibilité par 2									

2.

Divisibles par 2	Non divisib. par 2

Place dans le tableau les nombres :
325, 612, 318, 213, 17, 300, 2 652, 3 839, 4 615, 2, 9, 306, 205, 4 912, 17 822.

3. Dans chacun des schémas suivants, entoure l'ensemble des nombres divisibles par 2.

+2 +7 +191 +13
+6 +8 +80
+174
+619

+29 +713 +2 913
+202 +116 +3 011
+94 250
+2 046 +801 +1 018 +322

4. Effectue les additions suivantes de classes par 2, notées $\dot{0}$ ou bien $\dot{1}$.

$\dot{1}+\dot{0}+\dot{1}=....(2)$; $\dot{1}+\dot{1}+\dot{1}+\dot{0}+\dot{1}=....(2)$; $\dot{1}+\dot{0}+\dot{1}+\dot{0}+\dot{1}=....(2)$

$\dot{1}+\dot{1}+\dot{0}+\dot{0}+\dot{1}=....(2)$; $\dot{1}+\dot{1}+\dot{1}+\dot{1}+\dot{1}+\dot{1}=....(2)$.

II

5. Dans cet exercice nous allons chercher un moyen rapide de trouver la classe par 4 d'un nombre.
Rappel : Dans le tableau de classement par 4, prends un nombre x. Additionne 4 à ce nombre; dans quelle colonne se trouve x+4? Dans quelle colonne se trouve x+(2X4)? x+(3X4)? x+(17X4)?
Prends un nombre; par exemple 1 531, il peut s'écrire 1 531 = 1 500 + 31.
Fais le même raisonnement avec 617, 828, 9 602, 18 800.
Conclus. Tout nombre (plus grand que 100) est dans la même classe par 4 que le nombre formé par ses deux derniers chiffres.
Enonce une règle de divisibilité par 4.

6. Tu as trouvé dans le 6 la règle de divisibilité par 4 : un nombre est divisible par 4 si le nombre formé par ses deux derniers chiffres est divisible par 4.
On te rappelle qu'en général une année est bissextile si le nombre qui la désigne est divisible par 4. En utilisant la règle de divisibilité par 4 dis si les années suivantes étaient bissextiles? 1907, 1916, 1784, 1839, 1922 1940. 1984 sera-t-elle bissextile ?

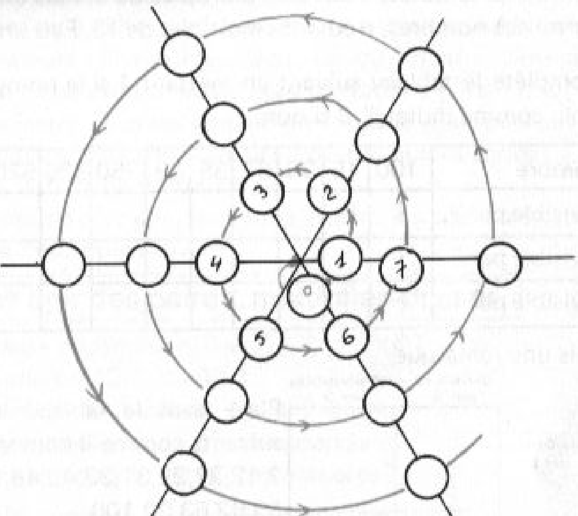
7. Le règlement interdit le stationnement devant chez moi du 1er au 15, l'autorisant les autres jours.
Combien de jours pourrai-je stationner devant chez moi en 1975? 1976?

(9 x 9) + 8 ; (2 x 9) + 1 ; (3 x 8) + 2 ; (5 x 5) + 4 ; (7 x 8) + 4 ; (9 x 7) + 8.

90 PROBLEMES SUR CLASSEMENTS ET DIVISIBILITE

I Classement

1.



Ecris la suite des nombres en suivant le sens des flèches.
Place ensuite les nombres de chaque demi-ligne droite sur ce tableau. Les nombres naturels ne sont-ils pas ainsi classés ?

0	1	2	3	4	5
6	7				

2. Les nombres : 13, 21, 32, 36, 40, 45, 77 sont écrits en base dix. Si tu voulais les écrire en base trois, quels sont ceux qui auraient pour chiffre des unités 0, 1, 2 ? Quels sont ceux qui auraient pour chiffre des unités 0, 1, 2, 3, 4 si tu voulais les écrire en base cinq ?

3. Les nombres 48, 74, 108, 220, 236, 1805, sont écrits en base dix. Trouve pour chacun d'eux le chiffre des unités, si tu voulais les écrire en base trois, en base cinq, en base neuf.

4. On a effectué les additions de classes suivantes; indique pour chacune d'elles à quel classement se rapportent ces classes en complétant les parenthèses.

$\dot{1}+\dot{1}=\dot{0}$ (□) ; $\dot{3}+\dot{2}=\dot{1}$ (□) ; $\dot{4}+\dot{3}=\dot{2}$ (□);
$\dot{1}+\dot{2}=\dot{0}$ (□) ; $\dot{8}+\dot{6}=\dot{5}$ (□) ; $\dot{5}+\dot{3}=\dot{2}$ (□)

LECTURES RECOMMANDÉES POUR ALLER PLUS LOIN

H. Abelson, G. Sussman et J. Sussman,
Structure and interpretation of computer programs :,
vol. MIT electrical engineering and computer science series, MIT Press, ISBN 0-262-51087-1, 1996.

J. Backus,
« Can programming be liberated from the von neumann style ? a functionnal style and its algebra of programs »,
Dans Communications of the ACM, vol. 21, p. 613–640, ACM, août 1978.

N. Bourbaki,
Éléments de mathématique - Théorie des ensembles,
Hermann, 1960.

A. Brygoo, T. Durand, M. Pelletier, C. Queinnec et M. Soria,
Programmation récursive (en Scheme),
Dunod, 2004.

E. W. Dijkstra,
« Algol 60 translation : An algol 60 translator for the x1 and making a translator for algol 60 »,
cwireport MR 34/61, Stichting Mathematisch Centrum, adresse : http://oai.cwi.nl/oai/asset/9251/9251A.pdf, 1961,
(ALGOL Bulletin, 1).

K. Doets et J. van Eijck,
The haskell road to logic, math and programming, 2004.

Y. Esposito,
« Ressources haskell, notamment en prog web », adresse : http://yannesposito.com/, 2014.

D. E. Knuth,
The Art of Computer Programming, Volume 1 (3rd Ed.) : Fundamental Algorithms,
Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, ISBN 0-201-89683-4, 1997.

J. Kun,
« Math inter programming », adresse : http://jeremykun.com/, 2014.

M. Lipovača et V. Robert,
« Apprendre haskell vous fera le plus grand bien ! », adresse : http://lyah.haskell.fr/, 2013.

S. Lipschutz et M. Lipson,
Schaum's outlines : Discrete mathematics,
Mac Graw & Hill, 2007.

J. Munro,
Discrete mathematics for computing,
Chapman & Hall, 1992.

P.-C. Scholl, M.-C. Fauvet, F.Lagnier et F.Maraninchi,
Cours d'informatique : langages et programmation,
Masson, 1993.

D. Spivak,
Category theory for scientists, 2013.

THIRIOUX, GASPARI, MIREBEAU ET LEYRAT,
Mathématique contemporaine - CM1,
Magnard, 1970.

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

E. VIOLARD,
Mini-manuel de Programmation fonctionnelle,
vol. Informatique, Dunod, ISBN 9782100707010, 2014.