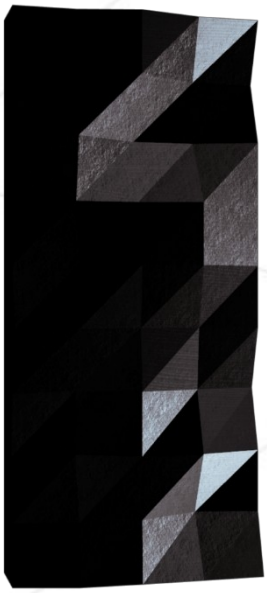


A vos marques, prêts, Boole !

Introduction à l'algèbre de Boole

Tristan Duquesne, Luc Lenôtre

Résumé : Découvrir le fonctionnement mathématique des

ordinateurs Version : 2

Contenu

Chapitre I Avant-propos

Même si nous savons aujourd'hui que de nombreux animaux ont une compréhension innée de la géométrie et des nombres, et que presque toutes les civilisations ont laissé des traces de comptage ou de géométrie, il ne s'agit pas vraiment de "mathématiques" au sens où nous l'entendons aujourd'hui. L'histoire des mathématiques, et donc de la science, commence pour ainsi dire dans le Croissant fertile. Toutefois, ce sont les travaux de plusieurs penseurs grecs qui ont réorienté les mathématiques de ces "outils de calcul primitifs" vers les puissants systèmes formels qui ont donné naissance à la science moderne.

Le début de cette transition s'est fait à travers deux penseurs en p a r t i c u l i e r , Euclid**e**t Aristote.

Euclide ne supportait pas de défendre une idée sans pouvoir la prouver. En fin de compte, le mieux qu'Euclide ait pu faire a été de limiter les présupposés (aussi appelés "postulats" ou "axiomes", des choses qui sont supposées vraies) de son monument théorique à seulement 5 idées : il a quand même réussi à construire tout le reste de sa géométrie en se basant **uniquement** sur ces 5 idées. L'œuvre de sa vie, **les Éléments,** est le deuxième livre le plus publié et le plus traduit de l'histoire de l'humanité (le premier étant la Bible).

Aristote, à la fois philosophe et scientifique (bien que cette distinction n'existait pas à son époque) a eu une vie intellectuelle extrêmement prolifique, allant bien au-delà des mathématiques "pures". Dans son ouvrage intitulé l'**Organon**, Aristote définit notamment une méthode pour calculer correctement la "vérité" : l'**Organon** est le premier traité de logique au sens où nous l'entendons aujourd'hui. Il y considère, entre autres, qu'un système logique ne peut comporter de contradiction, sinon toute idée du système devient à la fois vraie et fausse (le "principe d'explosion").

Les échos de ces travaux ont traversé les âges ; Al-Khwarizmi, Saint Thomas d'Aquin... Les apports d'autres civilisations, notamment de l'Inde, et les travaux d'un nombre difficilement dénombrable de mathématiciens, ont contribué à renforcer le langage de ces systèmes formels (algèbre) pour en faire le fondement de toutes les sciences modernes. Cependant, les idées les plus fondamentales ont été développées très tôt. La combinaison de ces deux doctrines (la rigueur d'Euclide et la logique d'Aristote) a semé les graines qui ont permis aux mathématiques de devenir l'outil fondamental de la connaissance humaine.

A la fin du 19ème siècle, avec la 2ème révolution industrielle, un grand sentiment de

sécurité s'est installé.

Un sentiment de confiance s'est emparé des scientifiques occidentaux, qui étaient alors à la pointe de la recherche scientifique mondiale. Le pouvoir de la science et de la technologie s'affirme et les scientifiques commencent à croire que tout, absolument tout, sera démontrable par la science : cette idéologie est appelée positivisme. Riemann, Lobachevsky, etc., ont réussi à résoudre la principale préoccupation d'Euclide, à savoir montrer que son 5e axiome avait des alternatives. Et en suivant les voies tracées par George Boole, qui a trouvé comment algébriser la logique d'Aristote, et Georg Cantor, qui avec sa théorie des ensembles a donné au monde un langage commun à toutes les mathématiques, les facultés de philosophie et de mathématiques de l'Occident se sont alliées pour donner naissance au domaine moderne de la logique (computationnelle).

En 1900, lors du Congrès international des mathématiciens à la Sorbonne, David Hilbert (l'un des mathématiciens les plus prolifiques et visionnaires de l'histoire, la superstar de l'époque) a donné une conférence qui est devenue légendaire. Il y a énoncé 23 problèmes mathématiques qui, selon lui, allaient définir le XXe siècle (et il avait largement raison), affirmant que l'inconnu céderait la place à la science mathématique. Certains d'entre eux ne sont toujours pas résolus aujourd'hui, et sont déjà légendaires (certains ont même un pot d'or au bout de l'arc-en-ciel...). Mais l'un des plus importants, le 2e problème, concernait les fondements des mathématiques elles-mêmes. Il s'agissait de montrer que les axiomes des mathématiques modernes (théorie des ensembles, ainsi que les axiomes de Zermelo et Fraenkel), qui permettaient de construire l'arithmétique à partir de la théorie des ensembles, conduisaient à u n e théorie bien fondée et cohérente : sans aucune contradiction.

En 1936, un Autrichien du nom de Kurt Gödel, étoile montante de cette communauté de logiciens, répond par la négative au 2ème problème de Hilbert : on ne peut pas démontrer la cohérence de l'arithmétique. Il part du minimum axiomatique fondamental : en gros, "qu'il y ait une théorie T exprimée dans un système formel (langage logique) L, etc." ; et montre que si cette théorie est assez riche pour exprimer l'arithmétique, alors il est impossible de démontrer la cohérence de la théorie avec les seuls outils de la théorie elle-même.

Sa preuve, que l'on appelle aujourd'hui les "théorèmes d'incomplétude", a une formulation extrêmement technique, mais nous allons essayer de vous en donner une idée . Pour c e l a , nous avons besoin d'un peu de vocabulaire.

Une "proposition" est une phrase qui peut être vraie ou fausse : "le temps est agréable" est une proposition ; "qui suis-je ?" n'en est pas une. Une proposition est dite "prouvable" ou "démontrable" s'il existe une preuve formelle (un calcul logique) que cette proposition est vraie. Une proposition est dite "réfutable" s'il existe une preuve formelle que cette proposition est fausse. Une proposition qui est soit démontrable, soit réfutable, est dite "décidable". Une proposition qui n'est ni démontrable ni réfutable est dite "indécidable".

Théorème 1 : "tout système (L,T) suffisamment riche pour formuler et effectuer une certaine arithmétique est incomplet". En d'autres termes, il existe, dans tout langage L avec une théorie T, définie selon certains axiomes pour pouvoir exprimer "1+1=2", au moins une proposition indécidable. Pour cela, Gödel utilise la proposition G : "le théorème G n'est pas prouvable dans la théorie T", inspirée du paradoxe du menteur ("cette phrase est fausse" : si elle est vraie, alors elle est fausse ; si elle est fausse, elle est vraie : contradiction). Si G est prouvable, alors G est vrai, donc G n'est pas prouvable : contradiction. Si G est réfutable, alors G est faux, donc il est "non non prouvable", donc

il est prouvable : contradiction. Conclusion, G n'est ni prouvable ni réfutable : G est indécidable.

Théorème 2 : Si le système (L, T) contient de l'arithmétique et est cohérent (ne contient pas de contradiction ; Gödel "suppose que c'est possible", c'est-à-dire que s'il existe une preuve de cohérence, il s'agit d'une démonstration et non d'une réfutation), alors il est impossible d'obtenir une preuve de la cohérence de (L, T) en utilisant uniquement (L, T). Pourquoi ? Parce que si une preuve de la cohérence de (L,T) existait, elle permettrait, pour toute proposition de (L,T), de distinguer si cette proposition est vraie, ou si elle est fausse. En d'autres termes, si une telle preuve existait, on pourrait savoir, en passant par cette preuve, si G est vrai ou faux. Mais G est indécidable ; une telle preuve n'existe donc pas. Il n'existe donc pas de preuve de la cohérence de (L,T) dans (L,T). En fait, si une telle preuve existe, il y a contradiction, et le principe d'explosion s'applique.

Les théorèmes d'incomplétude ont été un choc extrême pour la communauté mathématique, et pour la science en général. Ils ont tué le positivisme scientifique en son cœur : même en mathématiques, il n'est pas possible de tout savoir. C'est plus de 2500 ans de questionnements scientifiques, épistémologiques et religieux qui trouvent leur réponse à cet instant : il faut faire un choix d'axiomes, de se baser sur une "croyance", même en maths...

Mais certains, loin de s'avouer vaincus, ont reformulé la question. "On ne peut pas tout savoir", certes. Et puis, grâce à la logique, on s'est aperçu que "prouver" et "calculer" étaient en fait les deux faces d'une même pièce, deux façons de voir un même phénomène universel. Du coup, la question "que puis-je savoir ?" s'est transformée en "que puis-je calculer ? C'est ainsi que, sur les cendres du programme de Hilbert et du positivisme, reprenant les travaux de logiciens et d'ingénieurs polonais, français, allemands, anglais et américains, Alan Turing et John von Neumann ont défini la notion de calcul et donné naissance à l'informatique moderne, clôturant une ère de la connaissance humaine et en donnant naissance à une nouvelle.

Ressources complémentaires :

- **LogiComix** (Apostolos Doxiadis et alii) : une bande dessinée phénoménale qui traite précisément de la période allant de l'émergence de la logique académique à la naissance de l'informatique.
- **Gödel, Escher, Bach, une tresse d'or éternelle** (Douglas Hofstadter) : un monument de la littérature, qui utilise les théorèmes d'incomplétude comme moteur d'analyse des systèmes autoréférentiels, en particulier de la cognition humaine.
- **A History of Non-Euclidean Geometry** (Extra Credits) : une série de 5 vidéos, d'une durée totale de moins d'une heure, qui retrace l'histoire des mathématiques en se basant sur les prolongements historiques du travail d'Euclide.
- **Math has a fatal flaw** (Veritasium) : une excellente vidéo qui vous aidera à mieux comprendre les théorèmes d'incomplétude de Gödel et la manière dont Gödel utilise les liens entre l'arithmétique et la logique pour prouver ses arguments.
- **Comment compter au-delà de l'infini** (VSauce) : une vidéo qui explique comment, grâce à la théorie des ensembles, les mathématiciens ont réalisé qu'il existait différents types d'infini.

Une anecdote amusante : vers la fin de sa vie, Einstein disait souvent que s'il résidait à l'IAS à Princeton, c'était uniquement pour pouvoir accompagner Gödel dans ses

promenades et parler de mathématiques avec lui.

I.1 Comment, par la logique, nous avons trompé les rochers en leur faisant croire

Message de l'utilisateur **gnhicbfjnjjbb** sur Reddit :

Pensez à un simple interrupteur. Il allume ou éteint une ampoule. Maintenant, au lieu de brancher cet interrupteur sur une ampoule, il faut le brancher sur un autre interrupteur.

Supposons, par exemple, qu'en allumant l'interrupteur de la lumière, l'autre interrupteur envoie toujours un signal "on", même s'il est éteint. Cela ressemble un peu à ce que nous appelons une "porte ou", car seul le premier interrupteur OU le second interrupteur doit être allumé pour que le second interrupteur envoie un signal "on".

Nous pouvons également avoir le concept de "et", si nous imaginons que le fait d'éteindre le premier interrupteur coupe complètement le circuit du deuxième interrupteur, de sorte que même si le deuxième interrupteur est allumé, il n' allume pas la lumière.

Une fois que nous avons "et" et "ou" (enfin, aussi "pas", mais "pas" n'est qu'un interrupteur à l'envers qui allume les choses s'il est éteint, et les éteint s'il est allumé), nous pouvons calculer tout ce que nous voulons. Par exemple, voici comment nous ferions de l'arithmétique simple :

(cela va devenir un peu dense, mais restez avec moi, car il est vraiment important que les ordinateurs soient capables de faire cela)

Tout d'abord, il faut convertir le nombre en une "représentation binaire". Il s' agit d'une façon élégante de dire "donnez à chaque nombre une étiquette qui est un modèle de ' on' et ' off' ". Par exemple, nous pouvons représenter chaque nombre de 0 à 3 par 00, 01, 10, 11. Dans notre
Dans le monde, on écrit 1 2 3 4 5 6 7 8 9 10, mais en binaire, on fait comme si les chiffres du milieu n' existaient pas, et au lieu d'écrire 2 comme "2", on l'écrit comme " 10". Il s'agit toujours de 2, mais il est désormais plus facile de le représenter avec des "on" et des "off".

Deuxièmement, nous voulons faire des additions comme des additions normales. Prenons le chiffre le plus à droite - il peut être 0 ou 1, et nous essayons de l'ajouter à un autre chiffre qui est soit 0, soit 1. Au début, nous pourrions essayer quelque chose comme une "porte ou". Alors 0+0 est 0, 0+1 est 1, et 1+0 est 1, ce qui semble bien pour l'instant. Sauf que 1 OU 1 nous donnera... aussi 1, ce que nous ne voulons pas. Nous voulons obtenir 0 et reporter un 1 à gauche (rappelez-vous, nous ne pouvons pas créer le chiffre 2, nous devons représenter 2 comme "10"). Ce que nous voulons en fait, c'est quelque chose que l'on appelle un "xor", c'est-à-dire "ou" et non pas "et". Nous prenons le résultat d'une porte "ou" et nous le "et" avec le résultat inversé d'une porte "et". Nous aurons donc 0+0 = 0, 0+1 = 1, 1+0 = 1, et 1+1 = 0. Pour nous assurer que nous additionnons réellement 1 et 1, et que nous ne l'effaçons pas simplement à 0, nous devons également enregistrer un chiffre de retenue, mais il s'agit simplement d'une porte "et". Si le premier ET le deuxième chiffre sont tous deux des 1, nous reportons un 1, sinon nous reportons un 0.

Troisièmement, nous faisons la même chose un pas plus à gauche, mais nous incluons également l e chiffre de report si nous en avons un. Nous "xons" les

chiffres pour voir si nous devons enregistrer

un 1 ou un 0 dans cette position, et si nous avons 2 ou plus de 1 (dans les portes que nous connaissons, une façon d'écrire cela est "' a et b' ou ' b et c' ou ' a et c'"), nous portons un
1 à la position suivante.

Nous pouvons donc faire des additions. En répétant l'addition, nous pouvons multiplier. Nous pouvons également effectuer une soustraction par un processus similaire. Avec la soustraction répétée, nous pouvons faire
la division longue. En gros, nous pouvons résoudre tous les problèmes mathématiques que nous voulons.

Mais comment un être humain normal peut-il faire ce calcul si tous les chiffres sont des séquences bizarres de "on" et "off" ? Eh bien, nous pouvons relier quelques interrupteurs à des ampoules, mais des ampoules minuscules de différentes couleurs. Ce sont les pixels de l'écran. Si les interrupteurs veulent afficher le nombre binaire "11", ils peuvent allumer un motif sur l'écran qui ressemble à "3", de sorte que l'homme puisse le comprendre. Comment l'ordinateur sait-il à quoi ressemble un "3" ? Eh bien, les motifs d'allumage et d'extinction qui ressemblent à un "3" sont représentés sous la forme d'une grande formule mathématique, et notre ordinateur peut faire tous les calculs qu'il veut, de sorte qu'il peut calculer les ampoules (pixels) qu'il doit éventuellement allumer et éteindre.

Sous le capot, chaque donnée - chaque image, chaque mot - est représentée par une sorte d'étiquette numérique et passe par une looooooongue chaîne d'interrupteurs pour la transformer en un motif intelligible de pixels sur l'écran.

Il s'agit en grande partie d'un ensemble d'opérations arithmétiques très rapides. Par exemple, si vous pouvez calculer les trajectoires des rayons lumineux, vous pouvez dessiner une image en 3D sur l' écran. Il faut faire tout un tas d'équations physiques sur la façon dont la lumière rebondirait sur l'objet et atteindrait les yeux des gens, s'ils regardaient un véritable objet en 3D.
Mais nous savons comment notre ordinateur fait des maths - il s'agit d' un ensemble d'on
-Les interrupteurs d'arrêt sont reliés entre eux.

Tous ces interrupteurs sont des bouts de fil sur un morceau de silicium, et c'est ainsi que nous avons trompé les roches.

Chapitre II

Introduction

Dans ce module, nous parlerons de l'algèbre de Boole et de la théorie des ensembles, deux théories très importantes en mathématiques et en informatique.

Vous verrez également que vous avez probablement déjà une bonne maîtrise de l'algèbre de Boole sans même le savoir ! Ce sera l'occasion de pousser votre compréhension au niveau suivant.

II.1 Règles générales

- Pour ce module, les prototypes de fonctions sont décrits en Rust, mais vous pouvez utiliser le langage de votre choix. Il y a cependant quelques contraintes pour le langage de votre choix :
    - Il doit prendre en charge les types génériques
    - Il doit prendre en charge les fonctions en tant que citoyens de première classe (exemple : il prend en charge les expressions lambda).
    - Il doit implémenter nativement les opérations de type bitwise sur les types entiers, ou au moins les opérations de type bitwise sur un type bitmap/boolmap, bien que le premier soit fortement recommandé.
    - En option : prise en charge de la surcharge de l'opérateur
- Nous vous recommandons d'utiliser du papier si vous en avez besoin. Dessiner des modèles visuels, prendre des notes sur le vocabulaire technique... De cette façon, il vous sera plus facile d'assimiler les nouveaux concepts que vous découvrirez, et de les utiliser pour construire votre compréhension de concepts de plus en plus liés.
  De plus, faire des calculs de tête est vraiment difficile et sujet à l'erreur, et le fait d'avoir toutes les étapes écrites vous aidera à comprendre où vous avez fait une erreur quand vous l'avez faite.
- Ne restez pas bloqué parce que vous ne savez pas comment résoudre un exercice : utilisez l'apprentissage par les pairs ! Vous pouvez demander de l'aide sur Slack (42 ou 42-AI) ou Discord (42-AI) par exemple.

- Vous n'êtes pas autorisé à utiliser une bibliothèque mathématique, même celle incluse dans la bibliothèque standard de votre langage, sauf indication contraire explicite.
- Pour chaque exercice, il se peut que vous deviez respecter une complexité de temps et/ou d'espace donnée. Ces points seront vérifiés lors de l'évaluation par les pairs.
    - La complexité temporelle est calculée par rapport au nombre d'instructions exécutées
    - La complexité de l'espace est calculée par rapport à la quantité maximale de mémoire allouée simultanément
    - Tous deux doivent être calculés en fonction de la taille de l'entrée de la fonction (un nombre, la longueur d'une chaîne de caractères, etc...).

Chapitre III

Exercice 00 - Adder

III.0.1 Objectif

Vous devez écrire une fonction qui prend comme paramètres deux nombres naturels a et b et renvoie un nombre naturel égal à a + b . Cependant, les seules opérations que vous êtes autorisé à utiliser sont les suivantes :

- & (ET binaire)
- | (OU binaire)
- ^ (XOR par bit)
- << (décalage vers la gauche)
- >> (décalage vers la droite)

- = (affectation)

- ==, !=, <, >, <=, >= (opérateurs de comparaison)

L'opérateur d'incrémentation (++ ou += 1) **n'**est autorisé **que** pour incrémenter l'index d'une boucle et ne doit pas être utilisé pour calculer le résultat lui-même.

Vous devez également présenter une fonction principale afin de tester votre fonction, prête à être compilée (si nécessaire) et exécutée.

III.0.2 Instructions

Le prototype de la fonction à écrire est le suivant :

```
fn adder(a : u32, b : u32) -> u32 ;
```

Chapitre IV

Exercice 01 - Multiplicateur

IV.0.1 Objectif

L'objectif est le même que pour l'exercice précédent, sauf que le nombre naturel retourné est égal à
a * b. Les seules opérations que vous êtes autorisé à utiliser sont :

- & (ET binaire)
- | (OU binaire)
- ^ (XOR par bit)
- << (décalage vers la gauche)
- >> (décalage vers la droite)

- = (affectation)

- ==, !=, <, >, <=, >= (opérateurs de comparaison)

L'opérateur d'incrémentation (++ ou += 1) **n'**est autorisé **que** pour incrémenter l'index d'une boucle et ne doit pas être utilisé pour calculer le résultat lui-même.

Vous devez également présenter une fonction principale afin de tester votre fonction, prête à être compilée (si nécessaire) et exécutée.

IV.0.2 Instructions

Le prototype de la fonction à écrire est le suivant :

```
fn multiplier(a : u32, b : u32) -> u32 ;
```

Chapitre V

Exercice 02 - Code gris

V.0.1 Objectif

Vous devez écrire une fonction qui prend un entier n et renvoie son équivalent en code Gray.

Vous devez également présenter une fonction principale afin de tester votre fonction, prête à être compilée (si nécessaire) et exécutée.

V.0.2 Instructions

Le prototype de la fonction à écrire est le suivant :

```
fn gray_code(n : u32) -> u32 ;
```

Exemples :

```
println !("{}", gray_code(0)) ;
// 0
println !("{}", gray_code(1)) ;
// 1
println !("{}", gray_code(2)) ;
// 3
println !("{}", gray_code(3)) ;
// 2
println !("{}", gray_code(4)) ;
// 6
println !("{}", gray_code(5)) ;
// 7
println !("{}", gray_code(6)) ;
// 5
println !("{}", gray_code(7)) ;
```

```
// 4
println !("{}", gray_code(8)) ;
// 12
```

Chapitre VI

Exercice 03 - Évaluation booléenne

VI.0.1 Objectif

Vous devez écrire une fonction qui prend en entrée une chaîne de caractères contenant une formule propositionnelle en notation polonaise inversée, qui évalue cette formule et qui renvoie le résultat.

Vous devez également présenter une fonction principale afin de tester votre fonction, prête à être compilée (si nécessaire) et exécutée.

VI.0.2 Instructions

Chaque caractère représente un symbole. L'entrée ne contient que les caractères suivants :

Symbole	Mathématiques équivalent	Description
0	$\bot$	faux
1	$\top$	vrai
!	$\neg$	Négation
&	$\wedge$	Conjonction
|	$\vee$	Disjonction
^	$\oplus$	Disjonction exclusive
>	$\Rightarrow$	État des matériaux
=	$\Leftrightarrow$	Équivalence logique

Voici quelques exemples de formules propositionnelles :

```
10&
```

Équivalent :

$$\top \wedge \bot$$

```
10|
```

Équivalent : $\top \vee \bot$

```
10|1&
```

Équivalent : $(\top \vee \bot) \wedge \top$

```
101|&
```

Équivalent : $\top \wedge (\bot \vee \top)$

Si la formule n'est pas valide, le comportement de la valeur de retour est indéfini (mais nous vous encourageons à imprimer un message d'erreur).

Le prototype de la fonction à écrire est le suivant :

```
fn eval_formula(formula : &str) -> bool ;
```

Exemples :

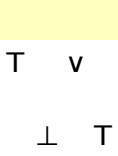
```
println !("{}", eval_formula("10&")) ;
// faux
println !("{}", eval_formula("10|")) ;
// vrai
println !("{}", eval_formula("11>")) ;
// vrai
println !("{}", eval_formula("10=")) ;
// faux
println !("{}", eval_formula("1011||=")) ;
// vrai
```

La notation polonaise inversée, bien que facile à analyser et à évaluer pour un ordinateur, est très laide et difficile à lire pour les humains.

Comment

Pourquoi ne pas utiliser une structure d'arbre syntaxique abstrait (binaire) pour analyser et modéliser votre formule ? Vous pouvez probablement ajouter des utilitaires intéressants, comme la conversion de cet arbre binaire en un arbre normal, ou des moyens de le visualiser. Vous pouvez également utiliser l'une des bibliothèques d'arbres existantes de votre langage. Faire cela maintenant pourrait probablement s'avérer utile à votre compréhension pour les exercices suivants !

Par exemple, la formule $\top \wedge (\bot \vee \top)$ peut être représentée par l'arbre suivant :



Chapitre VII

Exercice 04 - Table de vérité

VII.0.1 Objectif

Vous devez écrire une fonction qui prend en entrée une chaîne de caractères contenant une formule propositionnelle en notation polonaise inversée, et qui écrit sa table de vérité sur la sortie standard.

Vous devez également présenter une fonction principale afin de tester votre fonction, prête à être compilée (si nécessaire) et exécutée.

VII.0.2 Instructions

Chaque caractère représente un symbole. L'entrée ne contient que les caractères suivants :

Symbole	Mathématiques équivalent	Description
A...Z	$A..Z$	Variables distinctes avec des valeurs inconnues
!	$\neg$	Négation
&	$\wedge$	Conjonction
|	$\vee$	Disjonction
^	$\oplus$	Disjonction exclusive
>	$\Rightarrow$	État des matériaux
=	$\Leftrightarrow$	Équivalence logique

Une formule peut comporter jusqu'à 26 variables distinctes, une par lettre. Chaque variable peut être utilisée plusieurs fois.

A titre d'exemple, pour la formule suivante :

```
AB&C|
```

ce qui équivaut à :

$$(A \wedge B) \vee C$$

la fonction doit écrire ce qui suit sur la sortie standard :

```
$ ./ex04 | cat -e
| A | B | C | = |$
|---|---|---|---|$
| 0 | 0 | 0 | 0 |$
| 0 | 0 | 1 | 1 |$
| 0 | 1 | 0 | 0 |$
| 0 | 1 | 1 | 1 |$
| 1 | 0 | 0 | 0 |$
| 1 | 0 | 1 | 1 |$
| 1 | 1 | 0 | 1 |$
| 1 | 1 | 1 | 1 |$
```

Si la formule n'est pas valide, le comportement est indéfini (mais nous vous encourageons à imprimer un message d'erreur).

Le prototype de la fonction à écrire est le suivant :

```
fn print_truth_table(formula : &str) ;
```

Chapitre VIII

Interlude 00 - Règles de réécriture

Les règles de réécriture permettent de transformer une expression logique en une expression équivalente. Tous les systèmes logiques ont leur propre ensemble de règles qui sont autorisées et que vous êtes censé suivre (l'algèbre booléenne n'en est qu'une parmi d'autres !). En algèbre booléenne, les règles suivantes existent (liste non exhaustive) :

VIII.1 Élimination de la double négation

$$(\neg\neg A) \Leftrightarrow A$$

VIII.2 Conditions matérielles

$$(A \Rightarrow B) \Leftrightarrow (\neg A \lor B)$$

VIII.3 Équivalence

$$(A \Leftrightarrow B) \Leftrightarrow ((A \Rightarrow B) \land (B \Rightarrow A))$$

VIII.4 Les lois de Morgan

$$\neg(A \lor B) \Leftrightarrow (\neg A \land \neg B)$$
$$\neg(A \land B) \Leftrightarrow (\neg A \lor \neg B)$$

VIII.5 Distributivité

$$(A \wedge (B \vee C)) \Leftrightarrow ((A \wedge B) \vee (A \wedge C))$$
$$(A \vee (B \wedge C)) \Leftrightarrow ((A \vee B) \wedge (A \vee C))$$

Attention, contrairement à la distributivité à sens unique de × sur +, celle-ci va dans les deux sens !

Chapitre IX

Exercice 05 - Forme normale de la négation

IX.0.1 Objectif

Vous devez écrire une fonction qui prend en entrée une chaîne de caractères contenant une formule propositionnelle en notation polonaise inversée, et qui renvoie une formule équivalente en **forme normale de négation** (NNF), ce qui signifie que tous les opérateurs de négation doivent être situés juste après une variable.

Vous devez également présenter une fonction principale afin de tester votre fonction, prête à être compilée (si nécessaire) et exécutée.

IX.0.2 Instructions

Le format des formules propositionnelles est le même que celui de l'exercice précédent.

Par exemple, si la fonction prend AB& ! (équivalent : $\neg(A \wedge B)$) en entrée, elle doit retourner A!B!| (équivalent : $\neg A \vee \neg B$) en sortie.

Le résultat ne doit contenir que des variables et les symboles suivants !, & et | (même si l'entrée contient d'autres opérations).

Il peut y avoir plusieurs résultats valables, un seul est requis.

Si la formule n'est pas valide, le comportement est indéfini (mais nous vous encourageons à imprimer un message d'erreur).

Le prototype de la fonction à écrire est le suivant :

```
fn negation_normal_form(formula : &str) -> String ;
```

Exemples :

```
println !("{}", negation_normal_form("AB& !")) ;
// A!B!|
println !("{}", negation_normal_form("AB| !")) ;
// A!B!&
println !("{}", negation_normal_form("AB>")) ;
// A!B|
println !("{}", negation_normal_form("AB=")) ;
// AB&A!B!&|
println !("{}", negation_normal_form("AB|C& !")) ;
A!B!&C!| // A!B!&C!|
```

Chapitre X

Exercice 06 - Forme normale conjonctive

X.0.1 Objectif

Vous devez écrire une fonction qui prend en entrée une chaîne de caractères contenant une formule propositionnelle en notation polonaise inversée et qui renvoie une formule équivalente en **forme normale conjonctive** (CNF). Cela signifie que dans la sortie, chaque négation doit se trouver juste après une variable et chaque conjonction doit se trouver à la fin de la formule.

Vous devez également présenter une fonction principale afin de tester votre fonction, prête à être compilée (si nécessaire) et exécutée.

Vous pouvez vous aider de la fonction que vous avez écrite lors de l'exercice précédent.

X.0.2 Instructions

Le format des formules propositionnelles est le même que celui de l'exercice précédent.

Par exemple, si la fonction prend ABCD&|& (équivalent : $A \wedge (B \vee (C \wedge D))$ en entrée, elle doit retourner ABC|BD|&& (équivalent : $A \wedge (B \vee C) \wedge (B \vee D)$) en sortie.

Le résultat ne doit contenir que des variables et les symboles suivants !, & et | (même si l'entrée contient d'autres opérations).

Il peut y avoir plusieurs résultats valables, un seul est requis.

La taille de la formule de sortie peut croître de manière exponentielle avec la taille de l'entrée.
Il existe un algorithme pour éviter cela, mais il n'est pas obligatoire de l'utiliser.

Si la formule n'est pas valide, le comportement est indéfini (mais nous vous encourageons à imprimer un message d'erreur).

Le prototype de la fonction à écrire est le suivant :

```
fn conjunctive_normal_form(formula : &str) -> String ;
```

Exemples :

```
println !("{}", conjunctive_normal_form("AB& !")) ;
// A!B!|
println !("{}", conjunctive_normal_form("AB| !")) ;
// A!B!&
println !("{}", conjunctive_normal_form("AB|C&")) ;
// AB|C&
println !("{}", conjunctive_normal_form("AB|C|D|")) ;
// ABCD|||
println !("{}",  conjunctive_normal_form("AB&C&D&")) ;
// ABCD&&&&
println !("{}", conjunctive_normal_form("AB&!C!|")) ;
// A!B!C!||
println !("{}", conjunctive_normal_form("AB|!C!&")) ;
// A!B!C!&&
```

Toutes les expressions CNF valides pour une formule donnée ne sont pas identiques. Il existe un moyen de simplifier une CNF en utilisant ce que l'on appelle une carte de Karnaugh. Si vous le souhaitez, vous pouvez implémenter votre fonction de manière à ce qu'elle renvoie une CNF simplifiée. (Ceci est intéressant pour ceux qui s'intéressent à l a conception de circuits électroniques logiques).

Chapitre XI

Exercice 07 - SAT

XI.0.1 Objectif

Vous devez écrire une fonction qui prend en entrée une chaîne de caractères contenant une formule propositionnelle en notation polonaise inversée et qui indique si elle est satisfaisable.

Vous devez également présenter une fonction principale afin de tester votre fonction, prête à être compilée (si nécessaire) et exécutée.

Vous pouvez vous aider des fonctions que vous avez écrites dans les exercices précédents.

XI.0.2 Instructions

Le format des formules propositionnelles est le même que celui de l'exercice précédent.

La fonction doit déterminer s'il existe au moins une combinaison de valeurs pour chaque variable de la formule donnée qui fait que le résultat est ⊤. Si une telle combinaison existe, la fonction renvoie vrai, sinon elle renvoie faux.

Si la formule n'est pas valide, le comportement est indéfini (mais nous vous encourageons à imprimer un message d'erreur).

Le prototype de la fonction à écrire est le suivant :

```
fn sat(formula : &str) -> bool ;
```

Exemples :

```
println !("{}", sat("AB|")) ;
// vrai
println !("{}", sat("AB&")) ;
// vrai
println !("{}", sat("AA!&")) ;
// faux
println !("{}", sat("AA^")) ;
// faux
```

Chapitre XII

Interlude 01 - Règles d'inférence

Une règle d'inférence est une notation qui prend des prémisses (formules propositionnelles) et en déduit des conclusions.

Voici un exemple de règle de déduction.

$$\frac{A \Rightarrow B,\ A}{\therefore B}$$

L'exemple ci-dessus s'appelle **Modus Ponens** et peut être lu :

- $A \Rightarrow B$: "A implique B" ; "Si A, alors B".
- A : "A (est vrai)"
- $\therefore B$: "Donc B (est vrai)" Le

Modus Tollens est également

valable :

$$\frac{A \Rightarrow B,\ \neg B}{\therefore \neg A}$$

Le Modus Ponens et le Modus Tollens sont des règles fondamentales du raisonnement mathématique et sont très utiles en théorie des types pour l'inférence des types (ils sont notamment utilisés dans les compilateurs pour vérifier le typage d'un programme).

XII.1 Réduction à l'absurde

La réduction à l'absurde (du latin : "réduction à l'absurde") est une façon de prouver un résultat en essayant de prouver son contraire.

Si essayer de prouver que "*B n'*est pas vrai" aboutit à une contradiction, alors *B* doit être vrai.

Officiellement :

$$\frac{(A \cup \neg B) \Rightarrow \bot}{\therefore A \Rightarrow B}$$

où *A* est l'ensemble des affirmations que l'on sait déjà être vraies et *B* est la proposition à prouver.

Cette règle est basée sur un principe connu sous le nom de **loi du milieu exclu**, qui stipule que pour toute proposition *P* , si *P* est fausse, alors ¬*P* doit être vraie.

Chapitre XIII

Interlude 02 - Théorie des ensembles

XIII.1 Théorie naïve des ensembles

La théorie des ensembles est l'étude d'objets mathématiques appelés **ensembles**. Un ensemble est une collection d'objets qui peuvent être de toutes sortes (nombres, fonctions, autres ensembles, pommes de terre...), mais un ensemble ne peut pas contenir deux fois le même objet, ni se contenir lui-même (pour une raison que nous verrons plus loin). Il existe cependant des ensembles d'ensembles. On peut voir un ensemble comme un tableau qui peut être fini ou infini, et qui ne peut pas être indexé (l'ordre des objets n'a pas d'importance, seul compte le fait qu'un élément soit dans l'ensemble ou non).

Cette théorie est apparue vers les années 1870. Elle est issue des travaux de Richard Dedekind et de Georg Cantor et sert de fondement à l'ensemble des mathématiques.

XIII.2 Le paradoxe de Russell

En 1901, Bertrand Russell a découvert ce que l'on appelle aujourd'hui le **paradoxe de Russell**, qui montre que la théorie "naïve" des ensembles est erronée.

Imaginez un ensemble *A* qui contient tous les ensembles qui ne se contiennent pas eux-mêmes. Si *A ne* se contient pas lui-même, alors il doit se contenir lui-même. Mais s'il se contient, il ne doit pas se contenir...

Officiellement :

$$A = \{x \mid x \notin x\} \Rightarrow (A \in A \Longleftrightarrow A \notin A)$$

Ce paradoxe a conduit à la création d'autres théories plus avancées telles que la **théorie des ensembles de Zermelo- Fraenkel (ZF)**, la **théorie des catégories** et la **théorie des types** (cette dernière est très utile en informatique).

Pour en revenir à la théorie naïve des ensembles (qui est plus une collection de théories qu'une théorie unique), même si elle présente certains paradoxes, elle reste utile dans de nombreux cas (en particulier dans les cas suivants

lors de la comparaison de différentes versions de la théorie des ensembles).

XIII.3 Symboles

La théorie des ensembles introduit de nouveaux symboles, parmi lesquels (liste non exhaustive) :

- $\varnothing$: L'ensemble vide, l'ensemble sans éléments.
- $a \in B$: "a dans B" (il est vrai que a est un élément de l'ensemble B)
- $a \notin B$: "a pas dans B"
- $\forall a \in B$: "Pour tout a dans B"" (la proposition est vraie pour toute valeur a dans B)
- $\exists a \in B$: "Il existe a dans B" (la proposition est vraie pour au moins une valeur a dans B)
- $A \subseteq B$: A est un sous-ensemble de B (tous les éléments de A sont dans B, et A peut être égal à B elle-même)
- $A \not\subseteq B$: A n'est pas un sous-ensemble de B (certains éléments de A ne sont pas dans B)
- $A \supseteq B$: A est un sur-ensemble de B (tous les éléments de B sont dans A, et A peut être égal à B lui-même)
- $A \not\supseteq B$: A n'est pas un sur-ensemble de B (certains éléments de B ne sont pas dans A)

Le nombre d'éléments d'un ensemble A est appelé son **cardinal**. Il est souvent noté $card(A)$ ou $|A|$. Si un ensemble est infini, son cardinal est un nombre infini. Notez qu'il existe plusieurs cardinaux infinis distincts.

XIII.4 Lien entre la logique et la théorie des ensembles

En fait, la théorie des ensembles est basée sur la logique elle-même ; ou peut-être que c ' est l' inverse, et que l'on peut définir la logique à partir de la théorie des ensembles ! Quoi qu'il en soit, de nombreux concepts entre les deux sont équivalents/synonymes. (Plutôt cool, non ?). Les opérations logiques suivantes ont un équivalent dans la théorie des ensembles (ce ne sont pas les seules opérations qui existent) :

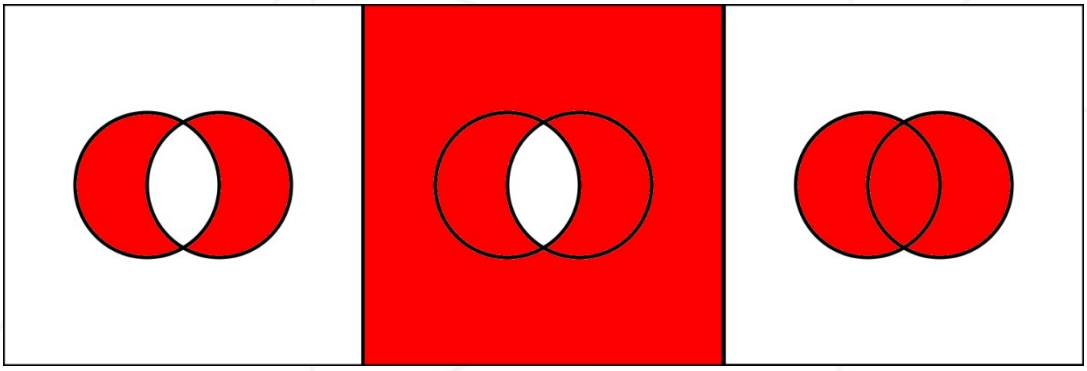
Algèbre booléenne	Booléen Nom de l'algèbre	Théorie des ensembles	Nom de la théorie des ensembles
$\neg A$	Négation	A^C ou $\overline{A}$	Complément
$A \vee B$	Disjonction	$A \cup B$	Union
$A \wedge B$	Conjonction	$A \cap B$	Intersection

La plupart des autres opérations peuvent être construites à partir du complément et de l'union (oui, même l'intersection le peut). Les règles de réécriture que nous avons

mentionnées précédemment (double négation, lois de De Morgan, ...) fonctionnent également sur les ensembles.

XIII.5 Diagrammes de Venn

Vous avez probablement déjà vu un diagramme de Venn. Ils sont utilisés pour exprimer les opérations logiques et les opérations sur les ensembles.



Ces trois diagrammes représentent des opérations sur deux ensembles, chaque ensemble étant représenté par un cercle (dans un ensemble global qui est un rectangle). Le résultat de l'opération est la partie colorée. **Pouvez-vous deviner de quelles opérations il s'agit ?**

Chapitre XIV

Exercice 08 - Powerset

XIV.0.1 Objectif

Vous devez écrire une fonction qui prend en entrée un ensemble d'entiers et renvoie son ensemble de puissances.

Vous devez également présenter une fonction principale afin de tester votre fonction, prête à être compilée (si nécessaire) et exécutée.

XIV.0.2 Instructions

Soit *A* un ensemble. Soit $B = \mathsf{P}(A)$ l'ensemble des puissances de *A*.

La fonction doit prendre l'ensemble *A* en entrée et renvoyer l'ensemble *B*.

Il est supposé que l'ensemble passé en paramètre est valide (pas d'éléments en double). Le prototype de la fonction à écrire est le suivant :

```
fn powerset(set : Vec<i32>) -> Vec<Vec<i32>> ;
```

Chapitre XV

Exercice 09 - Évaluation des ensembles

XV.0.1 Objectif

Vous devez écrire une fonction qui prend en entrée une chaîne de caractères contenant une formule propositionnelle en notation polonaise inversée et une liste d'ensembles (chacun contenant des nombres), puis qui évalue cette liste et renvoie l'ensemble résultant.

Vous devez également présenter une fonction principale afin de tester votre fonction, prête à être compilée (si nécessaire) et exécutée.

XV.0.2 Instructions

Chaque caractère représente un symbole. L'entrée ne contient que les caractères suivants :

Symbole	Mathématiques équivalent	Description
A...Z	*A*..*Z*	Ensembles distincts
!	¬	Négation
&	∧	Conjonction
|	∨	Disjonction
^	⊕	Disjonction exclusive
>	⇒	État des matériaux
=	⇔	Équivalence logique

Chaque lettre représente un ensemble qui est transmis à la fonction. L'ensemble *A* est le premier ensemble, l'ensemble *B* est le deuxième ensemble, etc...

L'ensemble global est considéré comme l'union de tous les ensembles donnés comme paramètres.

Si la formule n'est pas valide ou si le nombre d'ensembles fournis dans la liste n'est pas égal au nombre de variables dans la formule, le comportement n'est pas défini (nous vous encourageons toutefois à imprimer un message d'erreur approprié).

Le prototype de la fonction à écrire est le suivant :

```
fn eval_set(formula : &str, sets : Vec<Vec<i32>>) -> Vec<i32> ;
```

Exemples :

```
let sets = {
    {0, 1, 2},
    {0, 3, 4},
} ;
let result = eval_set("AB&", &sets) ;
// [0]

let sets = {
    {0, 1, 2},
    {3, 4, 5},
} ;
let result = eval_set("AB|", &sets) ;
// [0, 1, 2, 3, 4, 5]

let sets = {
    {0, 1, 2},
} ;
let result = eval_set("A !", &sets) ;
// []
```

Rappelez-vous : L'ordre des éléments dans l'ensemble résultant n'a pas d'importance.

Chapitre XVI

Interlude 03 - Des courbes qui remplissent l'espace

XVI.1 Ensembles communs

Nous avons parlé précédemment de la théorie naïve des ensembles. Mais nous n'avons pas parlé des ensembles couramment utilisés en algèbre, en arithmétique et en analyse (liste non exhaustive) :

Symbole	Nom	Type le plus proche en le langage C	Notes
N	Nombres naturels	unsigned int	N comprend zéro, alors que N* ne
Z	Entiers	int	
Q	Nombres rationnels	flotteur	Représente l'ensemble des nombres qui peut être exprimée comme la fraction de deux nombres entiers : $Q = \{\frac{a}{b} \mid a \in Z, b \in (Z \setminus \{0\})\}$
R	Nombres réels	flotteur	
C	Nombres complexes	complexe	A nombre complexe peut être représenté par $a + bi$ où $i^2 = -1$

Il existe quelques différences entre les types mathématiques et les types C, par exemple (liste non exhaustive) :

- int peut déborder, mais pas N ni Z. Techniquement, intX, où *X* est le nombre de bits, correspond à l'espace $Z/(^{2X})Z$ ("anneau des entiers modulo ^{2X}"), qui agit en quelque sorte comme une horloge où il y a ^{2X} heures et où l'on ne peut atterrir que sur les heures.
- float a une précision finie (c'est un ensemble de valeurs rationnelles binaires spécifiques, utilisées pour approximer d'autres valeurs) alors que Q et R ont une précision infinie.

La règle suivante s'applique :

$$\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{R} \subseteq \mathbb{C}$$

XVI.2 Une structure mathématique : Les groupes

Un groupe est une structure (que vous pouvez considérer comme l'"environnement" dans lequel vous travaillez) qui dispose d'un ensemble et d'un fonctionnement binaire interne.

Une **opération binaire interne** (également appelée "opération binaire stable" ou "opérateur binaire fermé") est une opération qui prend deux éléments de l'ensemble et renvoie un élément qui fait toujours partie du même ensemble. Alors qu'une **opération binaire externe** est une opération qui prend deux éléments de l'ensemble et renvoie un élément dans **un autre** ensemble.

Des exemples d'opérations binaires sont + (addition) ou × (multiplication). Mais il faut faire attention à l'ensemble pour lequel elles sont définies ! Par exemple, - (soustraction) n'est pas fermé sur N, mais il est fermé sur Z.

Un groupe est déclaré comme suit :

$$(G, -)$$

où G est un ensemble et - une opération binaire.

Un groupe doit respecter les propriétés suivantes :

- **Opération binaire interne** : $\forall a, b \in G, (a - b) \in G$ (pour tout a et b dans G, le résultat de $a - b$ est toujours dans G)
- **Associativité** : $\forall (a, b, c) \in G^3, a - (b - c) = (a - b) - c$ (l'ordre des parenthèses n'a pas d'importance)
- **Elément identité** : $\exists e \in G, \forall a \in G, e - a = a - e = a$ (il existe un élément e, appelé identité (ou unité, ou plus rarement, neutre), qui n'a aucun effet sur les autres, pour l'opération donnée).
- **Symétrie** : $\forall a \in G, \exists b \in G, a - b = b - a = e$, où e est l'élément identité (pour tous les éléments de G, il existe un élément inverse).

Si l'une des conditions ci-dessus n'est pas respectée, la structure

n'est pas un groupe. Par exemple, les structures suivantes sont des

groupes :

- (Z, +) : Le groupe additif des entiers

- (R \{0}, ×) : Le groupe multiplicatif des nombres réels (à l'exclusion de 0)

Pouvez-vous dire quel est leur élément d'identité respectif ?

Et les éléments suivants ne sont pas des groupes :

- (Z, ×)
- (R, ×)

Pouvez-vous comprendre pourquoi ?

Un groupe abélien (ou groupe commutatif) est un groupe qui remplit une condition supplémentaire :

- **Commutativité** : $\forall (a, b) \in G^2$, $a * b = b * a$ (l'ordre des opérandes n'a pas d'importance)

Ainsi, les groupes ci-dessus (Z, +) et (R \{0}, ×) sont également tous deux abéliens.

L'**ordre d'**un groupe est le **cardinal** de son ensemble

Pour plus d'informations sur les groupes, regardez :

- Formule d'Euler avec introduction à la théorie des groupes
- Théorie des groupes, abstraction et monstre à 196 883 dimensions

Pheww ! Je sais, cela peut sembler un peu confus si vous débutez, mais vous devez vous rappeler qu'il s'agit d'un sujet très abstrait. Les exemples et contre-exemples sont très utiles, et ils abondent pour des termes tels que "groupe", c'est donc un vocabulaire très important à long terme, puisqu'il rend les choses beaucoup plus faciles. Savoir que quelque chose est un "groupe" est un moyen rapide et facile de communiquer ce que vous êtes autorisé à faire avec un certain ensemble + combinaison d'opérations.

Si cela vous semble trop difficile pour l'instant, vous pouvez ignorer les mots "structure" et "groupe" dans la section suivante et y revenir plus tard.

XVI.3 Morphismes

Un morphisme est l'abstraction d'une fonction. Fondamentalement, un morphisme est un mappage de valeurs d'une structure vers une structure du même type.

On dit d'un morphisme qu'il préserve la structure car si la structure source est un groupe, l'ensemble cible d'un morphisme de groupe sera également un groupe (mais pas nécessairement le même).

Un morphisme f qui fait passer des valeurs de l'ensemble A à l'ensemble B est déclaré comme suit :

$$f : A \to B$$

Si le morphisme prend deux arguments (ou un vecteur à deux dimensions), il est déclaré comme suit :

$$f : A^2 \to B$$

L'**opérateur de composition** peut être appliqué à deux morphismes. Il est défini comme suit :

$$(f \circ g)(x) = f\,(g(x))$$

où f et g sont deux morphismes.

Si nous considérons la "collection de tous les ensembles" (ce qu'on appelle une catégorie dans la théorie des catégories, vous pouvez considérer les catégories comme des types de programmation), vous verrez le plus souvent la catégorie Set, pour laquelle les morphismes sont des fonctions. Je dis "le plus souvent", car il est possible de définir un autre choix de morphismes pour transformer la "collection de tous les ensembles" en une catégorie.

Ces "fonctions ensemblistes" ont des propriétés importantes qu'elles peuvent vérifier ou non. Nous disons qu'une fonction définie est :

- **injectif** : si chaque élément de l'ensemble source (domaine) est mis en correspondance avec **exactement un** élément de l'ensemble cible (codomaine)
- **surjectif** : si chaque élément de l'ensemble cible (codomaine) est mappé à partir d'**au moins un** élément de l'ensemble source (domaine)
- **bijective** : si elle est à la fois injective et surjective, ce qui signifie que chaque élément de la source et de la cible est mis en correspondance exactement 1 à 1

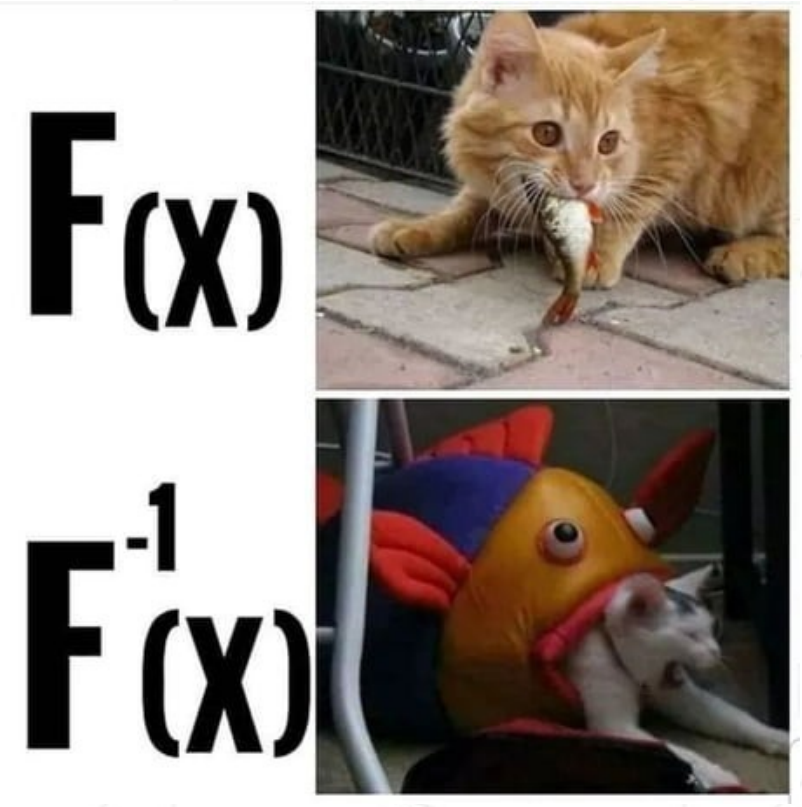
En théorie générale des catégories, ces notions, très importantes pour la théorie des ensembles, se généralisent respectivement à celles de "monomorphisme", "épimorphisme" et "isomorphisme".

Si un morphisme f est bijectif, il existe un morphisme inverse f^{-1}tel que :

$$(f^{-1} \circ f)(x) = (f \circ f^{-1})(x) = id_X\,(x) = x$$

Où $x \in X$ et id_X est le morphisme d'identité sur l'ensemble X. Remarquez que id_X est l'élément d'identité dans les groupesde fonctions d'ensembles, et ce que vous

voyez ci-dessus est la propriété de symétrie pour un groupe de fonctions avec l'opération $\circ$.



Un petit mème pour garder votre attention jusqu'à présent :D

Il existe beaucoup de vocabulaire supplémentaire autour des morphismes tels que les **automorphismes**, les **endomorphismes**, les **isomorphismes**, etc...

Pour plus d'informations, vous pouvez lire : https://en.wikipedia.org/wiki/Morphism.

XVI.4 Courbes de remplissage de l'espace

Une courbe de remplissage est une courbe continue qui associe un intervalle fermé $[0 ; 1] \in R$ à un ensemble de valeurs dans une ou plusieurs dimensions, de manière à couvrir l'ensemble de l'espace.

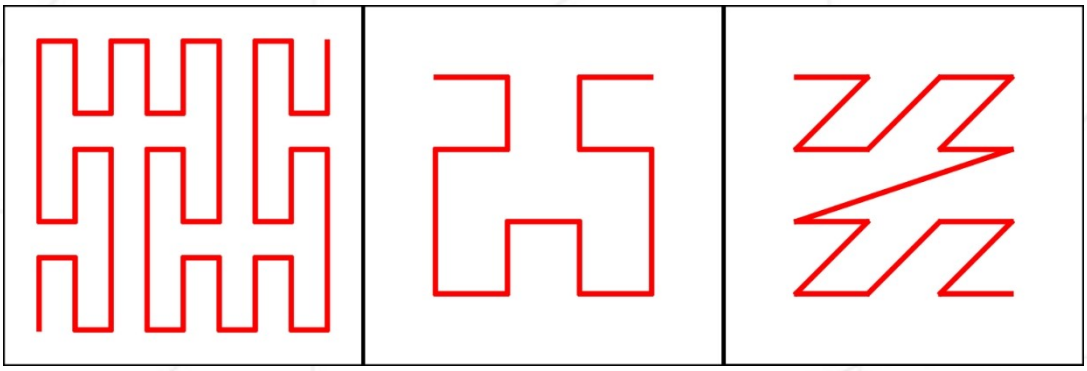
Une courbe de remplissage de l'espace est définie comme suit :

$$f : [0 ; 1] \in R \rightarrow M^n$$

Où [0, 1] est l'ensemble source (il contient des valeurs de 0 à 1, telles que 0,543543 et 0,3333...), et $n \in N$ est le nombre de dimensions d'un manifold M (une forme géométrique ou un espace) que nous souhaitons couvrir.

Il est important de noter que f **doit** être bijective.

Les courbes de remplissage sont définies par un certain nombre d'itérations. Plus le nombre d'itérations est élevé, plus la courbe est précise (plus elle couvre d'espace).



Deuxième itération de 3 courbes différentes de remplissage de l'espace, de gauche à droite : la **courbe de Peano**, la **courbe de Hilbert** et la **courbe de Lebesgue** (également appelée **courbe d'ordre Z)**.

La courbe d'ordre Z en particulier est très importante car elle est très utilisée en informatique. En gros, en prenant la fonction inverse d'une courbe de remplissage d'espace, on peut "stocker un espace entier à l'intérieur d'une ligne". Par exemple, certains GPU utilisent cette courbe pour stocker des textures dans la mémoire et réduire les échecs de cache (mémoire).

Recherchez ce que signifie la continuité dans le contexte d'une courbe de remplissage de l'espace.

Chapitre XVII

Exercice 10 -

Courbe

XVII.0.1 Objectif

Vous devez écrire une fonction (l'inverse d'une courbe de remplissage, utilisée pour encoder des données spatiales dans une ligne) qui prend une paire de coordonnées en deux dimensions et lui assigne une valeur unique dans l'intervalle fermé $[0 ; 1] \in \mathrm{R}$.

Vous devez également présenter une fonction principale afin de tester votre fonction, prête à être compilée (si nécessaire) et exécutée.

XVII.0.2 Instructions

Soit f une fonction et soit A un ensemble tel que :

$$f : (x, y) \in [[0 ; {}^{216}- 1]]^2 \subset \mathrm{N}^2 \rightarrow A$$

$$A \subset [0 ; 1] \subset R$$

Où ?

$$card(A) = card([[0 ; {}^{216}- 1]]^2)$$

La fonction f ci-dessus est bijective et représente la fonction à mettre en œuvre. Vous êtes libre d'utiliser la méthode que vous voulez tant qu'elle reste bijective.

Si l'entrée est en dehors de la plage, le comportement est indéfini (mais nous vous encourageons à imprimer un message d'erreur).

Le prototype de la fonction est le suivant :

```
fn map(x : u16, y : u16) -> f64 ;
```

Cet exercice n'est pas obligatoire.

Chapitre XVIII

Exercice 11 - Fonction inverse

XVIII.0.1 Objectif

Vous devez écrire la fonction inverse f^{-1} de la fonction f de l'exercice précédent (il s'agit donc cette fois d'une courbe de remplissage d'espace, utilisée pour décoder les données d'une ligne dans un espace).

Vous devez également présenter une fonction principale afin de tester votre fonction, prête à être compilée (si nécessaire) et exécutée.

XVIII.0.2 Instructions

Soit f et A la fonction et l'ensemble de l'exercice précédent, alors :

$$f^{-1} : A \rightarrow [[0 ; {}^{216}- 1]]^2 \subset \mathrm{N}^2$$

La fonction ci-dessus doit être implémentée de telle sorte que les expressions suivantes soient vraies pour les valeurs comprises dans l'intervalle :

$$(f^{-1} \circ f)(x, y) = (x, y)$$
$$(f \circ f^{-1})(x) = x$$

Si l'entrée est en dehors de la plage, le comportement est indéfini (mais nous vous encourageons à imprimer un message d'erreur).

Le prototype de la fonction est le suivant :

```
fn reverse_map(n : f64) -> (u16, u16) ;
```

Cet exercice n'est pas obligatoire.

Chapitre XIX

Soumission et évaluation par les pairs

Rendez votre travail dans votre dépôt `Git`comme d'habitude. Seul le travail contenu dans votre dépôt sera évalué lors de la soutenance. N'hésitez pas à vérifier les noms de vos fichiers pour vous assurer qu'ils sont corrects.