Éléments de théorie des langages et de théorie des ensembles pour informaticien·es

Quelles sont les « *données* » que l'on peut représenter en informatique ?
Quelles sont les « *bonnes* » représentations de ces données ?

Feuille « 02 » (du 1er septembre 2026)

NB seules les sections 1 et 2 sont à maîtriser dans le cadre du cours « Automates finis ». Les sections suivantes fournissent juste un complément utile pour les plus motivé·es d'entre vous par l'informatique théorique (voir l'explication en entête de la section 3 page 8).

Ce document [1] mélange notions de cours avec exercices à réaliser sur papier et/ou machine (en PYTHON). En particulier, les paragraphes intitulés « **RENDU** *n* » sont des exercices de programmation à rendre sur `https://teide.ensimag.fr/` (connection via un PC de l'école ou via le VPN Ensimag - voir `https://intranet.ensimag.grenoble-inp.fr/fr/informatique/vpn-ensimag`). Le format et la date limite du rendu sont indiqués sur Teide.

Les autres exercices sont corrigés, dans un document séparé, qu'il ne faut consulter qu'après avoir vraiment trouvé au moins un début de solution (même imparfait ou en partie faux). NE PAS CRAINDRE DE FAIRE DES ERREURS. Au contraire, rencontrer nos erreurs est nécessaire pour corriger nos images mentales floues ou fausses.

> *C'est quand nous craignons de nous tromper que l'erreur qui est en nous se fait immuable comme un roc.*
>
> Alexandre Grothendieck - `https://fr.wikipedia.org/wiki/Alexandre_Grothendieck`

1 Introduction (assez informelle) à la théorie des langages

On peut vouloir représenter les entiers naturels (de l'ensemble noté $\mathbb{N}$) par des séquences finies non-vides d'éléments de $\{0, 1\}$, comme « `1101` » qui représente l'entier écrit « `13` » en notation décimale. La notation décimale représente elle-même les entiers naturels comme des séquences finies non-vides d'éléments de $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Ceci conduit à la définition suivante, à la base de la **théorie des langages**.

Définition d'un mot w sur un vocabulaire V Soit V un ensemble fini de *symboles* (e.g. lettres, chiffres, caractères, idéogrammes, etc) appelé *vocabulaire*. Un mot est une séquence finie éventuellement vide d'éléments de V séparés par un *méta-symbole* dédié, généralement noté « . » (ou parfois avec juste un espace), qui ne doit pas figurer dans V. Le mot vide (i.e. la séquence vide) est noté par un autre méta-symbole, généralement « ε », qui ne doit pas non plus être dans V.

Exemple 1. Quatres mots sur $V = \{\texttt{A}, \texttt{T}, \texttt{G}, \texttt{C}\}$ (le vocabulaire des bases nucléiques de l'ADN [2]) :

ε `T` `A.C` `C.T.T.A.T.G`

Ce dernier mot pourrait peut-être coder un gène...

Exemple 2. On peut utiliser un vocabulaire V lui-même constitué de mots (V doit juste être un ensemble fini). Les mots sur V représentent alors des phrases. Typiquement, l'ensemble des mots de la langue française étant en nombre fini, on pourrait modéliser les phrases du français comme des mots (au sens de la théorie des langages) sur le vocabulaire défini par le dictionnaire, modulo conjugaison et passage au pluriel, étendu éventuellement avec les caractères de ponctuation.

1. N'hésitez pas à signaler les erreurs ou les points obscurs à `sylvain.boulme@imag.fr`, merci d'avance !
2. `https://fr.wikipedia.org/wiki/Base_azot%C3%A9e`

Notation sans séparateur Quand ce n'est pas ambigu, on peut omettre les « . ». Les 4 mots de l'exemple 1 peuvent aussi s'écrire :

ε `T` `AC` `CTTATG`

Exercice 1. Sur le vocabulaire $\{\texttt{C}, \texttt{I}, \texttt{CI}\}$ alors l'écriture sans séparateur est ambiguë. Par exemple, « `ICI` » peut correspondre à plusieurs mots. Lesquels ?

Exemple 3. On peut considérer les chaînes de caractères Python comme des mots sur le vocabulaire des caractères UTF-8 [3]. Comme « . » et « ε » appartiennent à ce vocabulaire, cela nécessite d'adapter les notations mathématiques usuelles. Le plus simple est d'utiliser celles de Python en utilisant une notation sans séparateur et entre apostrophes (*quote* en anglais). Ainsi le mot vide sera simplement écrit $''$. Dans cette notation, $'\texttt{ab}'$ représente un mot de deux symboles. [4]

Concaténation de deux mots Soient w_1 et w_2 deux mots non vides sur le même vocabulaire V. On note $w_1.w_2$ leur concanétation (formant un seul mot sur V obtenu en ajoutant le « . » entre les deux mots). Cette opération est étendue pour accepter des mots vides avec les égalités $\varepsilon.w = w$ et $w.\varepsilon = w$. Lorsque ce n'est pas ambigu, on s'autorise à laisser le symbole « . » implicite et à écrire $w_1 w_2$ à la place.

Exemple 4. Sur le vocabulaire $V = \{\texttt{a}, \texttt{b}\}$, on a ainsi $\texttt{bab}.\texttt{ba} = \texttt{babba}$. En Python, l'opérateur de concaténation sur les chaînes s'écrit `+`. [5] Donc `'bab'+'ba'` retourne `'babba'`.

Itéré d'un mot Soit w un mot sur un vocabulaire V. Soit n un entier naturel. On définit le mot w^n par récurrence sur n avec $w^0 \stackrel{def}{=} \varepsilon$ et $w^{n+1} \stackrel{def}{=} w.w^n$.

Exercice 2. Sur le vocabulaire $V = \{\texttt{a}, \texttt{b}\}$, que vaut $(\texttt{abb})^2.(\texttt{ba})^3$? En Python, l'opérateur d'itération est noté multiplicativement (ce qui est cohérent avec sa notation additive de la concaténation). Vérifiez en calculant `'abb'*2+'ba'*3`.

Longueur des mots On note $|w|$, la longueur du mot w, c'est-à-dire son nombre de symboles dans V (en comptant les occurrences multiples). Ainsi $|w_1.w_2| = |w_1|+|w_2|$. En Python, l'opérateur `len(w)` calcule la longueur de la chaîne `w`.
De façon similaire si $x \in V$, alors on note $|w|_x$ le nombre d'occurrences de x dans w. Et on a aussi $|w_1.w_2|_x = |w_1|_x + |w_2|_x$.

Exercice 3. Donner $|w|$ et $|w|_{\texttt{T}}$ de chacun des mots w de l'exemple 1. Puis, calculer $|w^n|$ en fonction de $|w|$ et n.

Exercice 4. On appelle B_0 l'ensemble des mots non vides sur le vocabulaire $\{\texttt{0}, \texttt{1}\}$. Chaque mot $w \in B_0$ représente un entier naturel noté $[\![w]\!]$, appelé **sémantique** de w, et défini par récurrence sur la longueur des mots :

$$\begin{array}{rcll} [\![a]\!] & \stackrel{def}{=} & a & \text{pour } a \in \{\texttt{0}, \texttt{1}\} \\ [\![w.a]\!] & \stackrel{def}{=} & (2 \times [\![w]\!]) + a & \text{pour } w \in B_0 \text{ et } a \in \{\texttt{0}, \texttt{1}\} \end{array}$$

On propose d'encoder un mot B_0 en Python comme un uplet non vide formé de 0 et de 1.

3. Voir https://fr.wikipedia.org/wiki/UTF-8.
4. Comme l'apostrophe « $'$ » fait elle-même partie du vocabulaire, pour représenter ce symbole à l'intérieur d'un mot, il faudra utiliser un mécanisme d'échappement de Python.
5. Les concepteurs de Python ont donc choisi de noter l'opération de concaténation de façon « additive », c'est-à-dire avec un `+`. Les mathématicien·nes préfèrent réserver les notations additives à des opérations *commutatives*, c'est-à-dire vérifiant « $x + y = y + x$ ». Ce n'est pas le cas de la concaténation. Pour les opérations non commutatives (comme la multiplication de matrices ou la concaténation de mots), les mathématicien·nes préfèrent une notation multiplicative (avec « $\times$ » ou « . »). Voir https://fr.wikipedia.org/wiki/Notation_multiplicative. Plus généralement, la notation additive de Python n'est pas très adaptée quand on passe de la concaténation des mots à la concaténation des langages (comme on va le voir bientôt).

Exemple : « `(1,1,0,1)` » encode le mot « `1.1.0.1` ». Définir une fonction PYTHON « `semT(w)` » qui calcule l'entier $[\![w]\!]$. Par exemple, elle vérifie :

```
assert (semT((1,1,0,1))==13)
```

Exercice 5. Au lieu de représenter les mots de B_0 par des tuples, on veut les représenter directement dans des chaînes de caractères PYTHON dans la notation sans « . ». Ainsi la chaîne PYTHON `'1101'` encode le mot « `1.1.0.1` ». Adapter « `semT` » en « `semS` » de sorte que :

```
assert (semS('1101')==13)
```

On pourra utiliser la fonction prédéfinie **int** pour convertir les chiffres `'0'` et `'1'` en nombre.

RENDU 1. Généraliser `semS` en une fonction `semF` qui retourne un flottant PYTHON sur les nombres binaires à virgule (avec un nombre fini de chiffres) – écrits sur le vocabulaire V à trois éléments : « 0 » et « 1 » et « , ». Testez que votre fonction produit des résultats corrects avec :

```
assert (semF('1101')==13)
assert (semF('1101,')==13)
assert (semF('1101,000')==13)
assert (semF('11,101')==3.625)    # voir le cours 1
assert (semF(',' + '1'*53)==0.9999999999999999)
assert (semF(',' + '1'*54)==1)
assert (semF(',' + '01'*27)==1/3)
```

Remarquons que le codage des flottants PYTHON sur 64 bits (voir https://en.wikipedia.org/wiki/Double-precision_floating-point_format) induit des résultats mathématiquement approximatifs.

Langage sur un vocabulaire V L'ensemble des mots sur V est noté V^*. Un langage L sur V est par définition un sous-ensemble de V^*.

Précision importante cette définition mathématique assimile un langage à une *syntaxe* (c'est-à-dire une notation, comme la notation décimale ou la notation binaire). Alors qu'au niveau informel (par exemple, quand on parle d'un « langage de programmation » donné), un langage correspond souvent à la combinaison d'une syntaxe avec une sémantique. Dans la suite du cours d'automates finis, on va se focaliser dans un premier temps sur les syntaxes. La prise en compte des sémantiques sera abordée en période 2.

Exercice 6. Expliciter le lien entre $\{0,1\}^*$ et B_0 de l'exo 4.

Exemple 5. Soit L un langage sur V. Étant donné un entier naturel n, on note $\lfloor L \rfloor_n$ l'ensemble des mots de L qui ont une longueur égale à n. Ainsi, $\lfloor L \rfloor_n$ est aussi un langage (fini) sur V. Avec les notations de la section 2, on définira simplement $\lfloor L \rfloor_n \stackrel{def}{=} \{w \in L \mid |w| = n\}$.

Exercice 7. Écrire en PYTHON une fonction `filtreB0(n)` qui affiche l'ensemble des mots de $\lfloor B_0 \rfloor_n$ (un par ligne). Par exemple, `filtreB0(3)` produit l'affichage ci-contre.

```
000
001
010
011
100
101
110
111
```

Indication : on écrira en fait une fonction `filtreB(n, pref)` qui, lorsque `pref` vaut un mot u, affiche l'ensemble des mots de la forme $u.w$ où w est un mot de $\lfloor \{0,1\}^* \rfloor_n$. Autrement dit `filtreB(n, u)` permet d'afficher *récursivement* l'ensemble des mots de $\lfloor \{0,1\}^* \rfloor_n$ en insérant un préfixe `u` devant chaque mot affiché. Avec les notations de la section 2, l'ensemble affiché est simplement $\{u\}.\lfloor \{0,1\}^* \rfloor_n$.

Dans la représentation B_0 de l'exo 4, tout mot w correspond au même entier que le mot $0.w$. En particulier, chaque entier peut donc avoir une représentation arbitrairement grande. On peut donc préférer la représentation sans zéro superflu, autrement dit définir l'ensemble B_1 des mots de B_0 tel que tout mot de longueur supérieur ou égale à 2 ne commence pas par `0`. Comme B_1 est un sous-ensemble de B_0, cela ne nécessite même pas d'adapter la sémantique.

100
101
110
111

RENDU 2. En utilisant directement la fonction `filtreB(n, pref)` introduite précédemment, définir une fonction PYTHON `filtreB1(n)` qui affiche $\lfloor B_1 \rfloor_n$. On vérifiera que `filtreB1(3)` produit l'affichage ci-contre.

Comme les langages sont des ensembles, la théorie des langages a besoin de s'appuyer sur une théorie des ensembles. Avant de continuer à formaliser la théorie des langages, il faut davantage formaliser la théorie des ensembles.

2 Théorie des ensembles

On définit ci-dessous les concepts mathématiques de base de la théorie des ensembles [6]. Pour faire le lien avec l'informatique, on programme aussi certains d'entre eux sur les ensembles finis de PYTHON3, et plus précisément les « *frozenset* » [7]. On observe que PYTHON3 offre des constructions syntaxiques directement inspirées de la théorie des ensembles, notamment les définitions par compréhension. [8] Ce type de constructions existent en fait de nombreux langages de programmation modernes comme HASKELL ou RUST. [9]

Une des difficultés dans la formalisation de la notion intuitive d'ensemble est d'éviter les ensembles *paradoxaux* qui mènent à une contradiction logique. Notamment le célèbre ensemble paradoxal de Russel [10] définit comme "*l'ensemble des ensembles qui n'appartiennent pas à eux-mêmes*" [11] qu'on pourrait noter informellement « $\{x \mid x \notin x\}$ ». Par exemple, un tel ensemble pourrait être construit par « $\{x \in \mathbb{E} \mid x \notin x\}$ » dans la théorie ci-dessous, en supposant l'existence d'un ensemble $\mathbb{E}$ de tous les ensembles. Ce qu'on évite donc de faire ! Ainsi, ce paradoxe s'avère très utile pour démontrer que certains ensembles, comme $\mathbb{E}$, n'existent pas (voir aussi à l'exo 25).

Pour éviter un tel risque de contradiction logique, on se limite à supposer l'existence de quelques constructions « *intuitives* » dont on se convainc à l'usage qu'elles ne sont pas contradictoires. On construit les autres ensembles à partir de ces constructions de base. C'est la méthode axiomatique. [12]

Égalité sur les ensembles (extensionnalité) On **suppose** que deux ensembles E et F sont indistinguables dans nos énoncés sur les ensembles si et seulement si $\forall\, x, x \in E \Leftrightarrow x \in F$. Dans ce cas, on note $E = F$ (l'égalité mathématique usuelle).

En PYTHON, lorsque `E` et `F` sont deux « *frozenset* », alors on peut tester leur égalité au sens mathématique avec l'expression « `E == F` ». Attention, dans d'autres cas, l'opérateur `==` de PYTHON ne correspond pas toujours à l'égalité mathématique (ce sera précisé à l'exemple 9).

Ensemble vide On **suppose l'existence** de l'ensemble vide, noté $\varnothing$ tel que $\forall\, x, x \notin \varnothing$.
En PYTHON, c'est « `frozenset()` ». Parfois, on notera aussi $\{\}$ au lieu de $\varnothing$.

6. Voir aussi https://fr.wikipedia.org/wiki/Th%C3%A9orie_des_ensembles.
7. Ce sont des ensembles finis **immutables**, comme ceux des ensembles mathématiques.
Voir https://docs.python.org/3/library/stdtypes.html#frozenset.
8. Voir le tutoriel https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions.
9. PYTHON est aussi bien adapté pour programmer en théorie des ensembles, car il n'a pas de typage statique, comme la théorie des ensembles. Et il offre des fonctions de réflexion, exploitées en section 6.
10. Voir https://fr.wikipedia.org/wiki/Paradoxe_de_Russell
11. Le paradoxe est le suivant : cet ensemble appartient à lui-même si et seulement si il n'appartient pas à lui-même !
12. Voir https://fr.wikipedia.org/wiki/Syst%C3%A8me_axiomatique#M%C3%A9thode_axiomatique.
Historiquement, il y a eu une tentative de dériver toutes les constructions mathématiques à partir de cette théorie des ensembles. Mais la théorie des ensembles présentant certaines limites (notamment lorsqu'on cherche à considérer les « *structures algébriques* » comme objets mathématiques), d'autres théories alternatives ont été ensuite proposées. Une des dernières en date, apparue dans les années 2010, est l'univalence ; trouver une « bonne » théorie sur-laquelle fonder formellement les mathématiques est donc toujours un sujet de recherche.
Voir https://en.wikipedia.org/wiki/Univalent_foundations.

Ensemble des parties Pour tout ensemble E, on **suppose l'existence** de l'ensemble, noté $\mathcal{P}(E)$, des parties de E tel que $\forall\, A,\ A \in \mathcal{P}(E) \Leftrightarrow (\forall\, x, x \in A \Rightarrow x \in E)$.
Usuellement, on note aussi $A \subseteq E$ la proposition logique $A \in \mathcal{P}(E)$. En PYTHON, on écrit « `A <= E` ».
(On dit aussi que A est un sous-ensemble de E ou que A est une partie de E).

Exemples 6. $\mathcal{P}(\varnothing) = \{\varnothing\}$; $\mathcal{P}(\{1\}) = \{\varnothing, \{1\}\}$; $\mathcal{P}(\{1,2\}) = \{\varnothing, \{1\}, \{2\}, \{1,2\}\}$
et $\mathcal{P}(\{1,2,3\}) = \{\varnothing, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$

Exercice 8. Calculer $\mathcal{P}(\{0,1,2,3\})$.

Ensemble par compréhension Pour tout ensemble E et toute proposition logique « P_x » portant sur une variable « x », on **suppose l'existence** de l'ensemble « $\{x \in E \mid P_x\}$ » qui est la partie $A \in \mathcal{P}(E)$ telle que $(\forall\, x, x \in A \Leftrightarrow P_x)$.

Exemple 7. Soit $E \subseteq \mathbb{N}$.
L'ensemble des entiers impairs de E est défini par $\{x \in E \mid x \bmod 2 = 1\}$.
Si E vaut $\{x \in \mathbb{N} \mid x < 10\}$, on peut calculer cet ensemble en PYTHON par :

```
E=frozenset(range(10))
frozenset(x for x in E if x % 2 == 1)
```

On peut aussi écrire plus simplement `frozenset(x for x in range(10) if x % 2 == 1)`.

Exercice 9. Étant donné n et m dans $\mathbb{Z}$, on note $\{n, \ldots, m\} \stackrel{def}{=} \{x \in \mathbb{Z} \mid n \leq x \leq m\}$.
(1) Donner une définition plus simple de « $\{n, \ldots, m\}$ » quand $n > m$.
(2) Donner une représentation en PYTHON de $\{n, \ldots, m\}$ (en tant que *frozenset*).

Union de deux ensembles Pour tous ensembles E et F, on **suppose l'existence** de l'ensemble, noté $E \cup F$, tel que $\forall\, x, x \in E \cup F \Leftrightarrow (x \in E \vee x \in F)$.
En PYTHON, l'union de « *frozenset* » est calculée par l'opérateur « `|` ». On peut vérifier qu'elle correspond à l'ensemble attendu avec :

```
assert (E | F == frozenset(x for b in range(2) for x in (E if b==0 else F)))
```

Intersection et différence Soient E et F deux ensembles. On définit :

intersection $E \cap F \stackrel{def}{=} \{x \in E \mid x \in F\}$.
En PYTHON, l'intersection de « *frozenset* » est calculée par l'opérateur « `&` ».

différence $E \backslash F \stackrel{def}{=} \{x \in E \mid x \notin F\}$.
En PYTHON, la différence de « *frozenset* » est calculée par l'opérateur « `-` ».

Exercice 10. Sur le modèle de l'union ci-dessus, donner deux assertions PYTHON testant une égalité d'ensembles qui vérifient que « `&` » et « `-` » sont corrects vis-à-vis de la définition mathématique.

Exercice 11. Soient E un ensemble avec x et y tels que $x \in E$ et $y \in E$. Définir :
(1) par compréhension l'ensemble singleton $\{x\}$, qui contient comme x unique élément.
(2) $\{x, y\}$, la plus petite partie de E contenant x et y.

Produit cartésien Pour tous ensembles E et F, on **suppose l'existence** de l'ensemble, noté $E \times F$, des paires d'éléments de E et F, tel que $\forall\, p,\ p \in E \times F \Leftrightarrow \exists\, x \in E, \exists\, y \in F, p = (x, y)$.

Exercice 12. Définir une fonction PYTHON « `prod(E,F)` » qui calcule le produit (en tant que « *frozenset* ») des ensembles finis `E` et `F`. Elle vérifie par exemple :

```
assert(prod(range(3),{'a','b'}) == frozenset({(0,'a'),(0,'b'),(1,'a'),(1,'b'),(2,'a'),(2,'b')}))
```

Compréhension dérivée de l'image d'une expression Soient E_1, …, E_n et F des ensembles. Soit $e_{x_1,\ldots,x_n}$ une expression qui dépend (éventuellement) de variables $x_1 \ldots x_n$, telle que $\forall\, x_1 \in E_1, \ldots \forall\, x_n \in E_n, e_{x_1,\ldots,x_n} \in F$.

On définit $\{e_{x_1,\ldots,x_n} \mid x_1 \in E_1 \wedge \ldots \wedge x_n \in E_n\} \stackrel{def}{=} \{y \in F \mid \exists\, x_1 \in E_1, \ldots, \exists\, x_n \in E_n, y = e_{x_1,\ldots,x_n}\}$ (où y est une variable qui n'apparaît pas dans le contexte).

Exercice 13. Soit $n \in \mathbb{N}$. On pose $P_n \stackrel{def}{=} \{\ (i,j) \mid i \in \{0,\ldots,n\} \wedge j \in \{0,\ldots,n-i\}\ \}$.

1. Écrire une fonction Python "**def** `P(n)`" qui retourne la liste des éléments de P_n (classée pour l'ordre croissant sur les couples de Python). Vérifier que `P(3)` donne le résultat attendu.
2. Calculer le nombre d'éléments de P_n (en fonction de n).
3. Montrer que $P_n = \{\ p \in \mathbb{N} \times \mathbb{N} \mid \exists\, i \in \mathbb{N}, \exists\, j \in \mathbb{N}, p = (i,j) \wedge i + j \leq n\ \}$.
4. Soit $P \stackrel{def}{=} \{\ p \in \mathbb{N} \times \mathbb{N} \mid \exists\, n \in \mathbb{N}, p \in P_n\ \}$ qu'on notera aussi $P = \bigcup_{n\in\mathbb{N}} P_n$. Montrer que $P = \mathbb{N} \times \mathbb{N}$.

Concaténation de langages Soient L_1 et L_2 deux langages sur un vocabulaire V. On pose $L_1.L_2 \stackrel{def}{=} \{w_1.w_2 \mid w_1 \in L_1 \wedge w_2 \in L_2\}$. On appelle $L_1.L_2$ la concaténation des langages L_1 et L_2 (il s'obtient en concaténant chacun des mots de L_1 avec chacun des mots de L_2).

Exercice 14. Donner la définition par compréhension primitive de $L_1.L_2$ (il s'agit essentiellement de trouver le bon F). Montrer ensuite comment simplifier $\{\varepsilon\}.L$ et $L.\{\varepsilon\}$.

Exercice 15. Montrer que $\lfloor V^* \rfloor_1 = V$ (cf. la notation de l'exemple 5). Remarquons que V est ici à la fois le vocabulaire et un langage sur V.

Itération de langages Soit n un entier naturel et L un langage (sur un vocabulaire V). On définit le langage (sur V) L^n par récurrence sur n, avec $L^0 \stackrel{def}{=} \{\varepsilon\}$ et $L^{n+1} \stackrel{def}{=} L.L^n$.

Itération de Kleene Soit L un langage (sur un vocabulaire V).

On définit le langage (sur V) L^* par $L^* \stackrel{def}{=} \{w \in V^* \mid \exists\, n \in \mathbb{N}, w \in L^n\}$ (qu'on aurait aussi pu noter $L^* = \bigcup_{n\in\mathbb{N}} L^n$).

On pose aussi $L^+ \stackrel{def}{=} L.L^*$.

Remarquons que cette définition « surcharge » la définition de V^* (V étant à la fois le vocabulaire et un langage sur V). Mais ce n'est pas un souci puisque cette deuxième (basée sur l'itération de Kleene) est cohérente avec la première (l'ensemble des mots sur V). [13]

Exercice 16. On considère le vocabulaire $V = \{\texttt{0}, \texttt{1}\}$. En utilisant uniquement des sous-ensembles finis de V et les opérateurs de concaténation et d'itération sur les langages, définir l'ensemble L des mots de V dont chaque symbole `0` est immédiatement suivi d'un symbole `1` comme dans `1101011101` ou `01111011`.

Exercice 17. Est-ce que les égalités suivantes sont valides (quelque soit L) ? Sinon, quelles sont les inclusions qui sont valides ? Justifier (avec des contre-exemples le cas échéant).

1. $L^* = \{w^n \mid w \in L \wedge n \in \mathbb{N}\}$?
2. $L^* = L^+ \cup \{\varepsilon\}$?
3. $L^+ = L^* \backslash \{\varepsilon\}$?

Exercice 18. En considérant le B_0 de l'exo 4, montrer que $B_0 = \{0,1\}^+$.

Exercice 19. En considérant que le B_1 du rendu 2 est formellement défini par $B_1 \stackrel{def}{=} B_0 \backslash (\{0\}.B_0)$, montrer que

$$B_1 = \{0\} \cup \{1\}.\{0,1\}^*$$

13. Autrement dit, on aurait pu appeler $\alpha(V)$ l'ensemble des mots sur V. Puis ayant défini l'itération de Kleene L^* d'un langage L en fonction de $\alpha(V)$, on aurait pu montrer $V^* = \alpha(V)$; et finalement oublier $\alpha(V)$ pour n'utiliser que V^*.

ce qui se lit par convention (le $\cup$ étant vu comme un opérateur additif moins prioritaire que les symboles multiplicatif) $B_1 = \{0\} \cup (\{1\}.(\{0,1\}^*))$.
En utilisant les égalités $(L_1 \cup L_2).L_3 = (L_1.L_3) \cup (L_2.L_3)$ et $L_3.(L_1 \cup L_2) = (L_3.L_1) \cup (L_3.L_2)$, on appliquera un raisonnement algébrique (c'est-à-dire exploitant des propriétés d'égalités) dans les langages (et les ensembles).

Exercice 20. On considère le vocabulaire $V = \{\texttt{0}, \texttt{1}, \texttt{'}, \texttt{'}\}$ (à trois symboles, le dernier représentant une virgule qu'on note entre apostrophes pour éviter la confusion avec le méta-symbole). En utilisant uniquement des sous-ensembles finis de $V \cup \{\varepsilon\}$ et les opérateurs d'union, de concaténation et d'itération sur les langages, définir un langage B_v des nombres à virgules, écrits en binaires avec un nombre fini de chiffres (dont la sémantique est donnée au rendu 1). On impose que tout mot de B_v contienne *au moins* un chiffre binaire (avant ou après la virgule) et *au plus* une virgule. Donc les 4 mots suivants sont dans B_v : « `00,00` », « `1` », « `10,` » et « `,010` ». Mais les deux mots suivants n'y sont pas : « `0,,` » et « `,` ».

Compréhension généralisée La forme précédente est généralisable avec une proposition $P_{x_1,\ldots,x_n}$ qui dépend de variables $x_1 \ldots x_n$ telle que $\forall\, x_1 \in E_1, \ldots \forall\, x_n \in E_n, P_{x_1,\ldots,x_n} \Rightarrow e_{x_1,\ldots,x_n} \in F$,
avec $\{e_{x_1,\ldots,x_n} \mid x_1 \in E_1 \wedge \ldots \wedge x_n \in E_n \wedge P_{x_1,\ldots,x_n}\}$
$\stackrel{def}{=} \{y \in F \mid \exists\, x_1 \in E_1 \ldots \exists\, x_n \in E_n, y = e_{x_1,\ldots,x_n} \wedge P_{x_1,\ldots,x_n}\}$
(où y est une variable qui n'apparaît pas dans le contexte).

Par exemple, l'item 3 de l'exo 13 aurait pu s'écrire :
« Montrer que $P_n = \{(i,j) \mid i \in \mathbb{N} \wedge j \in \mathbb{N} \wedge i + j \leq n\}$ ».

Exercice 21. Soit $V = \{\texttt{a}, \texttt{b}, \texttt{c}\}$. On pose $A = \{0,1,2,3\}$ et $B = \{\texttt{b}, \texttt{c}\}$. On définit le langage L sur V par

$$L \stackrel{def}{=} \{\texttt{a}^n.c.\texttt{b}^p \mid n \in A \wedge p \in \mathbb{N} \wedge c \in B \wedge n \leq p \leq 2n\}$$

Donner la définition par compréhension primitive de L et calculer son nombre d'éléments.

RENDU 3. Définir L de l'exo 21 à l'aide d'un schéma de compréhension Python en faisant références aux définitions de `A` et `B` ci-dessous :

```
A=range(4)
B=frozenset(('b','c'))
```

Pour faciliter le débogage, on pourra représenter L par un **tuple** (appelé `L`) plutôt qu'un **frozenset**. Vérifiez que le nombre d'éléments est bien celui attendu (via le **len** de Python).

Les deux variantes de définitions par compréhension introduites ci-dessus sont présentées d'une façon légèrement différente à la section suivante, dans le paragraphe "Image d'une application implicite".

3 Théorie des ensembles d'applications

La suite de ce document présente une théorie mathématique, initialement inventée par Cantor [14] à la fin du 19° siècle qui permet d'apporter des réponses parfois subtiles à certaines questions délicates sur la représentation d'*ensembles* de données.

Par exemple, dans la théorie de Cantor, $\mathbb{N}$ et B_0 de l'exo 4 et B_1 du rendu 2 sont définis comme des *ensembles* avec $B_1 \subseteq B_0$. Et $[\![.]\!]$ correspond à une *application* de $B_0 \to \mathbb{N}$. La théorie permet alors de comprendre pourquoi l'ensemble $\mathbb{R}$ des nombres réels **n'est pas représentable** en informatique. En particulier, il existe des nombres réels de $[0,1]$ pour lesquels il ne peut pas exister de programme permettant de calculer n'importe quel bit du développement binaire (cf. l'exo 40). Les nouvelles intuitions apportées par cette théorie ont été des étapes cruciales dans l'invention de l'informatique.

Le document continue à proposer des exercices pour vous aider à assimiler les notions présentées. Si vous êtes pressé·es, ne bloquez pas sur les exercices (lisez rapidement le corrigé). L'objectif principal est d'atteindre assez rapidement les exos 25, 34, 37, 39 et 40. Les paragraphes et exercices marqués comme « Optionnel » peuvent être vraiment ignorés en première lecture (ils sont proposés dans le cadre d'une relecture du document, pour comprendre encore un peu mieux...). De façon générale, la suite de ce document n'est pas le cœur du cours, c'est plutôt un complément utile : il fait juste partie de la culture de tout·e informaticien·ne qui se respecte (et peut aussi aider à mieux comprendre les autres cours de l'Ensimag).

Ensemble d'applications et applications comme ensembles de couples Soient E et F deux ensembles. On note

$$E \to F \stackrel{def}{=} \{f \in \mathcal{P}(E \times F) \mid \forall\, x \in E, \exists\, y \in F, f \cap (\{x\} \times F) = \{(x,y)\}\}$$

Autrement dit, pour $f \in E \to F$ et $x \in E$, il existe un unique $y \in F$, noté $f(x)$, tel que $(x,y) \in f$. On dit que f est une *application* de $E \to F$. [15] Une application f est donc elle-même un ensemble de couples. Pour définir plus facilement des applications, on introduit la notation « $x \in E \mapsto e_x$ » où « e_x » est un calcul qui dépend de la variable « x » et retournant un résultat dans F avec la signification suivante (où la variable r n'apparaît pas dans le contexte) :

$$(x \in E \mapsto e_x) \stackrel{def}{=} \{r \in E \times F \mid \exists\, x \in E, r = (x, e_x)\}$$

Soit $f \stackrel{def}{=} (x \in E \mapsto e_x)$. On a bien $\forall\, x \in E, f \cap (\{x\} \times F) = \{(x, e_x)\}$. Donc $f \in E \to F$.

Exemples 8. Par exemple, soit o_1 l'application $n \in \{2,4,5\} \mapsto 2n+1$.
On a $o_1 = \{(2,5),(4,9),(5,11)\}$ avec $o_1 \in \{2,4,5\} \to \{5,9,11\}$ et $\forall\, n \in \{2,4,5\}, o(n) = 2n+1$.
Remarquons aussi que $o_1 \in \{2,4,5\} \to \mathbb{N}$ mais que $o_1 \notin \{2,4,5\} \to \{1,\ldots,9\}$ (car $o_1(5) = 11$).
On peut aussi « prolonger » o_1 en une application $o \stackrel{def}{=} n \in \mathbb{N} \mapsto 2n+1$.
On a alors $o \in \mathbb{N} \to \mathbb{N}$, mais $o \notin \{2,4,5\} \to \mathbb{N}$ (car $o \not\subseteq \{2,4,5\} \times \mathbb{N}$).
Et, $o \cap (\{2,4,5\} \times \mathbb{N}) = o_1$.

Exercice 22. On admet que le calcul par récurrence de $[\![.]\!]$ donné en introduction définit bien un nombre de $[\![w]\!] \in \mathbb{N}$ pour tout $w \in B_0$.
(1) Comment définir $f_0 \in B_0 \to \mathbb{N}$ telle que $\forall\, w \in B_0, f_0(w) = [\![w]\!]$?

14. Voir https://fr.wikipedia.org/wiki/Georg_Cantor.

15. En anglais, « *application* » se traduit par « *map* » ou « *mapping* ». En informatique, on parle aussi de « *fonction totale* » au lieu de « *application* ». On peut définir l'ensemble des fonctions (partielles) de E dans F, noté ici $E \nrightarrow F$, par

$$E \nrightarrow F \stackrel{def}{=} \{f \in \mathcal{P}(E \times F) \mid F = \varnothing \vee \forall\, x \in E, \exists\, y \in F, f \cap (\{x\} \times F) \subseteq \{(x,y)\}\}$$

À la suite de l'exo 37, on peut alors démontrer $E \nrightarrow F \simeq E \to (\{1\} \uplus F)$.
Rem : ces notations sont celles standardisées par *Z*. Voir https://en.wikipedia.org/wiki/Z_notation.
Mais tout le monde n'utilise pas les mêmes. Notre « $E \to F$ » est souvent noté « F^E » (cf. propriétés en exo 37[8-10]).
Notre « $E \nrightarrow F$ » est parfois noté « $E \rightharpoonup F$ » voire « $E \to F$ » (attention aux confusions).
Voir https://fr.wikipedia.org/wiki/Application_(math%C3%A9matiques).

(2) Comment définir $f_1 \in B_1 \to \mathbb{N}$ telle que $\forall\, w \in B_1, f_1(w) = [\![w]\!]$?
(3) Comment définir $f_2 \in \{0,1\}^* \to \mathbb{N}$ telle que $\forall\, w \in B_0, f_2(w) = [\![w]\!]$?

Application représentable On dit qu'une *application* $f \in E \to F$ est **représentable par une fonction** PYTHON `f` si et seulement si pour tout `x` de E, `f(x)` *termine normalement* et retourne une valeur qui représente f(`x`).
Attention, l'opérateur PYTHON `==` sur les fonctions ne correspond pas à l'opérateur mathématique $=$ sur les applications. Étant donné deux applications f et g de $E \to F$, on a

$$f = g \Leftrightarrow \forall\, x \in E, f(x) = f(y)$$

Au contraire, le test « `f == g` » où `f` et `g` sont deux fonctions est équivalent à « `f is g` », c'est-à-dire au fait que `f` et `g` correspondent au même objet physique dans la mémoire de l'ordinateur. Voir les exemples ci-dessous. En fait, en conséquence du problème mentionné à l'exo 40, il ne peut pas exister un algorithme général effectuant le test d'égalité mathématique sur les applications.

Exemple 9. L'application o des exemples 8 est représentable par la fonction PYTHON (`lambda n: 2*n+1`) ou, de manière équivalente, par la fonction :

```
def o(n):
    return 2*n+1
```

Ici, le test « `(lambda n: 2*n+1) == (lambda n: 2*n+1)` » retourne `False` (chaque lambda-expression créé un nouvel objet physique en mémoire). Par contre, le test « `o == o` » retourne `True` (la variable `o` est ici associé à un unique objet physique en mémoire). Dans la suite, on préfère donc utiliser directement `is` que `==` pour comparer les fonctions PYTHON. Pour finir, remarquons que « `(lambda x:x)(o) is o` » retourne `True` : le résultat du calcul de gauche est bien l'objet nommé `o`.

Image d'une application Soient E et F deux ensembles et $f \in E \to F$. Soit $A \subseteq E$.
On note $f(A) \stackrel{def}{=} \{y \in F \mid \exists\, x \in A, y = f(x)\}$.
On dit que f est une **surjection sur** F si et seulement si $f(E) = F$. [16]

Exemple 10. Pour l'application o_1 des exemples 8, $o_1(\{2,4\}) = \{5,9\}$. De plus, o_1 est une surjection sur $\{5,9,11\}$, car $o(\{2,4,5\}) = \{5,9,11\}$. Donc, ce n'est pas une surjection sur $\mathbb{N}$.

Exercice 23. Soit g l'application définie par $g \stackrel{def}{=} (n \in \mathbb{N} \mapsto 7 + |n - 10|)$ (où $|x|$ est la valeur absolue de x qui vaut x si $x \geq 0$ et $-x$ sinon).
Existe-t-il un sous-ensemble de $\mathbb{N}$ sur-lequel g est une surjection ?

Exercice 24. On considère que E est une **représentation complète** de F s'il existe une application de $E \to F$ qui est une surjection (sur F). Est que B_0 et B_1 sont des représentations complète de $\mathbb{N}$? Par exemple, est-ce que f_0 et f_1 définies à l'exercice 22 sont des surjections (sur $\mathbb{N}$) ?

Théorème de Cantor Soit E un ensemble et $f \in E \to \mathcal{P}(E)$. Alors, f n'est pas une surjection sur $\mathcal{P}(E)$.

16. On dit aussi que f est « surjective sur F » (ou « dans F »). Usuellement, les mathématiciens utilisent une variante des définitions données ici dans laquelle toute application f est définie comme un triplet (E, F, g) avec E et F ensembles et $g \in E \to F$, ce dernier étant appelé le *graphe* de l'application f. Quand on dit alors que « f est surjective » (sans préciser l'ensemble d'arrivée), cela signifie simplement que g est une surjection sur F. La définition choisie dans ce document, qui assimile l'application et son graphe, est juste un peu plus simple (mais n'est donc pas complètement standard). Voir https://fr.wikipedia.org/wiki/Application_(math%C3%A9matiques)#D%C3%A9finition.

On déduit de ce théorème un premier résultat d'impossibilité de représentation informatique : si on admet que toutes les données informatiques sont représentables sur $\{0,1\}^*$, alors l'ensemble $\mathcal{P}(\{0,1\}^*)$ n'a pas lui-même de représentation complète. On va affiner ce résultat avec l'aide de la théorie de l'équipotence donnée en section 5.

Exercice 25. Démontrer le théorème de Cantor énoncé ci-dessus, en reformulant le paradoxe de Russel (mentionné en introduction) avec :

$$A \stackrel{def}{=} \{x \in E \mid x \notin f(x)\}$$

Image d'une application implicite On peut noter aussi $f(E)$ sous la forme « $\{f(x) \mid x \in E\}$ » en laissant f implicite, comme dans $\{2n+1 \mid n \in \mathbb{N}\}$ qui définit l'ensemble des entiers naturels impairs.
Lorsque $E = \{x \in U \mid P(x)\}$, on peut aussi éviter d'avoir à expliciter E en notant $f(E)$ par « $\{f(x) \mid x \in U \wedge P(x)\}$ ».

Exemple 11. L'ensemble des entiers impairs inférieurs à 9 est donné par $\{2n+1 \mid n \in \mathbb{N} \wedge n < 5\}$. En PYTHON, ce dernier ensemble est calculé par :

```
frozenset(2*n+1 for n in range(5))
```

Autrement dit, l'ensemble PYTHON « `frozenset(`$f(x)$`for`x`in`E`if`$p(x)$`)` » où p est à valeur dans les booléens, est noté mathématiquement « $\{f(x) \mid x \in E \wedge p(x) = \texttt{True}\}$ ».

Exercice 26. Soient $E_1 \stackrel{def}{=} \{2n \mid n \in \mathbb{N} \wedge n < 3\}$ et $E_2 \stackrel{def}{=} \{3m \mid m \in \mathbb{N} \wedge m < 3\}$, calculer $F \stackrel{def}{=} \{x \times y \mid x \in E_1 \wedge y \in E_2\}$.

Exercice 27 [Optionnel]. Soit E un ensemble et $x \in E$. Soient $X \stackrel{def}{=} \{x\}$ et $Y \stackrel{def}{=} \mathcal{P}(E \backslash X)$. Démontrer

$$\mathcal{P}(E) = Y \cup \{A \cup X \mid A \in Y\}$$

En déduire une fonction récursive PYTHON « `power(E)` » calculant $\mathcal{P}(\texttt{E})$ lorsque `E` est un « *frozenset* ».

Exercice 28 [Optionnel]. Soit $f \in E_1 \to (E_2 \to \mathbb{N})$ l'application « $x \in E_1 \mapsto (y \in E_2 \mapsto x \times y)$ », c-à-d. qui étant un $x \in E_1$ retourne une application qui dépend de x.
En PYTHON, on peut par exemple la représenter par `f = (lambda x: (lambda y: x*y))`.
Montrer comment construire le F de l'exo 26 en utilisant uniquement une définition par compréhension de la forme « $\{z \in \mathbb{N} \mid P(z)\}$ » où P utilise f.

4 Théorie des injections

Injection Soient E et F deux ensembles avec $f \in E \to F$.
On dit que f est une **injection** si et seulement si $\forall\, x \in E, \forall\, y \in E, f(x) = f(y) \Rightarrow x = y$
(ou, de manière équivalente, ssi $\forall\, x \in E, \forall\, y \in E, x \neq y \Rightarrow f(x) \neq f(y)$).
On pose $f^{-1} \stackrel{def}{=} \{(f(x), x) \mid x \in E\}$. Montrer que si f est une injection, alors $f^{-1} \in f(E) \to E$.

Exercice 29. Soit g de l'exo 23. Montrer que g n'est pas une injection.

Exercice 30. Soit $f \in E \to F$. Montrer que :
(1) Si f est une injection alors $\forall\, x \in E, f^{-1}(f(x)) = x$ et $\forall\, y \in f(E), f(f^{-1}(y)) = y$.
(2) Si f est une injection alors f^{-1} est une surjection sur E i.e. $f^{-1}(f(E)) = E$.
(3) S'il existe $g \in f(E) \to E$ tel que $\forall\, x \in E, g(f(x)) = x$ alors f est une injection et $g = f^{-1}$.
(4) Si f est une injection alors f^{-1} est aussi une injection et $f^{-1^{-1}} = f$.

Bijection Soient E et F deux ensembles avec $f \in E \to F$.
On dit que f est une **bijection sur** F si et seulement si f est une surjection sur F qui est aussi une injection.
Rem 1 : Toute injection $f \in E \to F$ est aussi une bijection sur $f(E)$.
Rem 2 : La composée de deux injections (resp. surjections) est une injection (resp. surjection) ; donc la composée de deux bijections est une bijection.

Exercice 31. Lorsque $f \in E \to F$ est bijective (sur F), on dit qu'elle constitue une **représentation canonique** de F par E : tout élément de F a une représentation unique sur E.
Est-ce que f_0 et f_1 définies à l'exercice 22 sont des bijections (sur $\mathbb{N}$) ?

Exercice 32. Soient $f \in E \to F$ et $g \in F \to E$ telles que
$$\forall\, x \in E, g(f(x)) = x \text{ et } \forall\, y \in F, f(g(y)) = y.$$
Montrer que f (resp. g) est bijective sur F (resp. sur E) et que $f^{-1} = g$ et $g^{-1} = f$.

5 Théorie de l'équipotence

Équipotence/Théorème de Cantor-Bernstein
Soient E_1 et E_2, $i_1 \in E_1 \to E_2$ et $i_2 \in E_2 \to E_1$ avec i_1 et i_2 injections.
Alors, il existe une bijection dans $E_1 \to E_2$.
Dans ce cas, on dit que les ensembles E_1 et E_2 sont **équipotents** et on note $E_1 \simeq E_2$.
Voir la preuve sur `https://fr.wikipedia.org/wiki/Th%C3%A9or%C3%A8me_de_Cantor-Bernstein`.

Exercice 33 [Optionnel]. Soit $n \in \mathbb{N}$.
(1) Montrer que pour tout $m \in \mathbb{N}$ tel que $n < m$,
il n'existe pas d'injection dans $\{1, \ldots, m\} \to \{1, \ldots, n\}$.
(2) En déduire que pour tout $m \in \mathbb{N}$, $\{1, \ldots, n\} \simeq \{1, \ldots, m\} \Leftrightarrow n = m$.

Exercice 34. Supposons deux équipotences $E_1 \simeq F_1$ et $E_2 \simeq F_2$. Montrer que :
(1) $E_1 \times E_2 \simeq F_1 \times F_2$.
(2) $E_1 \to E_2 \simeq F_1 \to F_2$.
(3) On n'a pas forcément $E_1 \cup E_2 \simeq F_1 \cup F_2$.

Union disjointe Soient E_1 et E_2 deux ensembles. On note $E_1 \uplus E_2 \stackrel{def}{=} (\{1\} \times E_1) \cup (\{2\} \times E_2)$.

Exercice 35 [Optionnel]. Définir une fonction Python « `disjoint_union(E1,E2)` » qui en fait le calcul sur les ensembles finis, avec un calcul par compréhension plus efficace que « `prod(1,E1) | prod(2,E2)` » n'appelant ni « `prod` » ni « | ».

Exercice 36. Montrer que
(1) si $E_1 \simeq F_1$ et $E_2 \simeq F_2$ alors $E_1 \uplus E_2 \simeq F_1 \uplus F_2$.
(2) $E_1 \uplus E_2 \simeq (E_1 \cup E_2) \uplus (E_1 \cap E_2)$.

Exercice 37. Soient E, E_1, E_2, E_3 des ensembles, $x \in E$ et $n, m \in \mathbb{N}$ quelconques. Montrer les équipotences suivantes (éventuellement, on peut s'amuser à programmer les bijections sous-jacentes en Python) :
(1) $\{x\} \simeq \{1\}$.
(2) $E_1 \times E_2 \simeq E_2 \times E_1$.
(3) $E_1 \uplus E_2 \simeq E_2 \uplus E_1$.
(4) $(E_1 \times E_2) \times E_3 \simeq E_1 \times (E_2 \times E_3)$.
(5) $(E_1 \uplus E_2) \uplus E_3 \simeq E_1 \uplus (E_2 \uplus E_3)$.
(6) $E_1 \times (E_2 \uplus E_3) \simeq (E_1 \times E_2) \uplus (E_1 \times E_3)$.
(7) $\varnothing \times E = \varnothing$ et $\{1\} \times E \simeq E$ et $\varnothing \uplus E \simeq E$.

(8) $\varnothing \to E \simeq \{1\} \simeq E \to \{1\}$ et $\{1\} \to E \simeq E$.
(9) $E_1 \to (E_2 \to E_3) \simeq (E_1 \times E_2) \to E_3$.
(10) $(E_1 \to E_3) \times (E_2 \to E_3) \simeq (E_1 \uplus E_2) \to E_3$.
(11) $\mathcal{P}(E) \simeq E \to \{0,1\}$.
(12) $\{1,\ldots,n\} \times \{1,\ldots,m\} \simeq \{1,\ldots,n \times m\}$.
(13) $\{1,\ldots,n\} \uplus \{1,\ldots,m\} \simeq \{1,\ldots,n+m\}$.
(14) $\{1,\ldots,n\} \to \{1,\ldots,m\} \simeq \{1,\ldots,m^n\}$.

Cardinal fini d'un ensemble Soit E ensemble. On dit que E est de **cardinal fini** si et seulement si il existe $n \in \mathbb{N}$ tel que $E \simeq \{1,\ldots,n\}$. Dans ce cas, d'après l'exo 33(2), il existe un unique n vérifiant cette équipotence et on note $\mathrm{card}(E) = n$.

Exercice 38. Soient E, E_1 et E_2 de cardinaux finis. Montrer les égalités suivantes à partir des équipotences de l'exo 37(11-14), puis de l'exo 36(2).
(1) $\mathrm{card}(E_1 \times E_2) = \mathrm{card}(E_1) \times \mathrm{card}(E_2)$.
(2) $\mathrm{card}(E_1 \uplus E_2) = \mathrm{card}(E_1) + \mathrm{card}(E_2)$.
(3) $\mathrm{card}(E_1 \to E_2) = \mathrm{card}(E_2)^{\mathrm{card}(E_1)}$.
(4) $\mathrm{card}(\mathcal{P}(E)) = 2^{\mathrm{card}(E)}$.
(5) $\mathrm{card}(E_1 \cup E_2) = \mathrm{card}(E_1) + \mathrm{card}(E_2) - \mathrm{card}(E_1 \cap E_2)$.

Equipotences classiques entre ensembles infinis On pose $[0,1] \stackrel{def}{=} \{x \in \mathbb{R} \mid 0 \leq x \leq 1\}$ et $2\mathbb{N} \stackrel{def}{=} \{2n \mid n \in \mathbb{N}\}$ et $2\mathbb{N}_{+1} \stackrel{def}{=} \{2n+1 \mid n \in \mathbb{N}\}$. On peut montrer les équipotences suivantes :
(1) $B_1 \simeq \mathbb{N}$
(2) $\mathbb{N} \simeq \mathbb{N}\backslash\{0\} \simeq \{0,1\}^*$
(3) $\mathbb{N} \simeq 2\mathbb{N}$ et $\mathbb{N} \simeq 2\mathbb{N}_{+1}$
(4) $\mathbb{N} \simeq \mathbb{N} \times \mathbb{N}$
(5) $\mathbb{N} \simeq \mathbb{Z}$
(6) $\mathbb{N} \simeq \mathbb{Q}$
(7) $[0,1] \simeq \mathbb{N} \to \{0,1\} \simeq \mathcal{P}(\mathbb{N})$
Rem : on déduit de (7) que $\mathbb{R}$ qui contient $[0,1]$ n'a pas de représentation complète sur $\mathbb{N}$.

Exercice 39. Démontrer les équipotences ci-dessus. Rem : sur les équipotences entre ensembles infinis, il est généralement plus facile d'exhiber une paire d'injections que de donner directement une bijection (qu'on pourra au final construire via Cantor-Bernstein).

6 Application à la (non)calculabilité en PYTHON

L'exo 39(7), combiné avec le théorème de Cantor (et sa preuve à l'exo 25), montre qu'il existe des nombres réels de $[0,1]$ non-représentables dans $\mathbb{N}$. On va maintenant adapter ces idées pour construire un nombre réel non représentable en PYTHON, c'est-à-dire un nombre réel de $[0,1]$ pour lequel il ne peut pas exister de programme calculant le i° bit du développement binaire en fonction de i. On reformule ici en PYTHON quelques idées sur les nombres réels (in)calculables, initialement proposées par Alan Turing.[17] Plus généralement, on illustre l'existence du non-calculable : le raisonnement effectué ici est adaptable à d'autres langages de programmation et à d'autres types de données non-dénombrables.

On note $\mathbb{S}$ l'ensemble des chaînes de caractères PYTHON. On fournit ci-dessous une fonction PYTHON appelée `str2int` qui représente une injection de $\mathbb{S} \to \mathbb{N}$ (non surjective), ayant pour inverse `int2str` de `str2int`$(\mathbb{S}) \to \mathbb{S}$. Comme la fonction PYTHON **str** peut représenter une injection triviale de $\mathbb{N} \to \mathbb{S}$ (d'inverse **int**), on a $\mathbb{S} \simeq \mathbb{N}$. Techniquement, `str2int(s)` transforme `s` en liste d'octets, via `s.encode` puis transforme cette liste d'octets en entier. La fonction `int2str(n)` procède

17. Voir https://fr.wikipedia.org/wiki/Nombre_r%C3%A9el_calculable.

dans l'ordre inverse. Remarquons que la valeur de l'entier retourné par `str2int` peut dépendre de variable d'environnement (e.g. l'encodage des caractères est ou non en UTF-8) mais `int2str` est bien censé faire la conversion dans l'autre sens. Par exemple, sur mon PC, `str2int('coucou')` produit l'entier `129121270460259`. Appliquer `int2str` sur cet entier redonne bien `'coucou'`.

```
def str2int(s):
    return int.from_bytes(s.encode(),sys.byteorder)

def int2str(n):
    if n>0:
        return n.to_bytes(1+(int.bit_length(n)-1)//8,sys.byteorder).decode()
    elif n==0:
        return ""
    raise ValueError
```

Par ailleurs, la fonction **eval** exécute une chaîne de caractères représentant une expression Python. Par exemple, **eval**`('7*6')` calcule `42`. Attention, une chaîne (délimitée par une paire de `"` ou `'`) est elle-même un calcul « trivial ». Ainsi **eval**`("'7*6'")` calcule `'7*6'`. Donc **eval**(**eval**`("'7*6'")`) retourne `42`. Grâce à `str2int`, on peut donc aussi encoder une expression Python dans un entier de $\mathbb{N}$. Par exemple, le calcul Python `7*6` peut être encapsulé dans la chaîne `'7*6'` puis encodé dans l'entier `str2int('7*6')` qui vaut `3549751` sur mon PC. L'évaluation de cet entier comme expression est effectué par la fonction `eval_code` qui ci-dessous : sur mon PC, `eval_code(3549751)` calcule `42`.

```
eval_code = lambda n:eval(int2str(n))
```

Remarquons qu'une telle lambda-expression est elle-même une expression Python qui calcule une fonction. Ainsi sur mon PC, `str2int('lambda n:eval(int2str(n))')` retourne `258377039265558152228767679365992533372818606600125068173676`. Le calcul Python ci-dessous retourne donc aussi `42`.

```
eval_code(258377039265558152228767679365992533372818606600125068173676)(3549751)
```

Et le code ci-dessous réussit sans erreur :

```
cec = str2int("eval_code")
assert (eval_code(cec)(cec) is eval_code)
```

Remarquons par contre que « `eval_code(0)(0)` » provoque une erreur à l'exécution, exactement comme « **eval**`("")(0)` ». Dans le cas général, l'exécution de `eval_code` pourrait aussi ne pas terminer (boucle infinie).

Exercice 40. Soit

$$\begin{aligned} f \stackrel{def}{=} & \quad \{(n,0) \mid n \in \mathbb{N} \wedge \textit{l'expression «}\ \texttt{eval\_code}(n)(n)\ \textit{» termine sans erreur et ne retourne pas } 0\} \\ \cup & \quad \{(n,1) \mid n \in \mathbb{N} \wedge \textit{«}\ \texttt{eval\_code}(n)(n)\ \textit{» ne termine pas, ou avec erreur, ou en retournant } 0\} \end{aligned}$$

On vérifie que $f \in \mathbb{N} \to \{0,1\}$. Et on définit $r \in [0,1]$ avec

$$r \stackrel{def}{=} \sum_{n\in\mathbb{N}} \frac{f(n)}{2^{n+1}}$$

(1) Montrer que $\frac{1}{2} \leq r < 1$.
(2) Montrer que f n'est pas représentable en Python. Autrement dit, il ne peut pas exister de programme qui, étant donné un entier $n \in \mathbb{N}$, calcule $f(n)$, c'est-à-dire le $(n+1)^\circ$ bit du développement en binaire de r. On considère donc que le nombre réel r n'est pas calculable.

Programmes auto-reproducteurs *[Optionnel]* Dans ce qui précède, on a utilisé des techniques d'auto-référence (i.e. faire référence à soi-même), pour construire des paradoxes absurdes prouvant des théorèmes d'inexistence. Il y a bien d'autres formes d'auto-références utilisées en programmation. Par exemple, on peut écrire des programmes qui s'auto-reproduisent, c'est-à-dire capable de générer leur propre code dans une de leurs structures de données, pendant leur exécution. Certains virus informatiques sont basés sur cette technique. En gros, l'idée est de produire un texte qui exprime quelque chose d'analogue à :

> *Recopie deux fois la ligne suivante sans l'interpréter.*
> *Recopie deux fois la ligne suivante sans l'interpréter.*

Il faut essentiellement traduire de façon spécifique les concepts « Recopie », « ligne », « suivante », etc, en incluant le retour à la ligne, la ponctuation et les espaces. C'est ce qu'on appelle un *quine*, du nom de l'inventeur du concept. [18]

Voilà un premier exemple Python qu'on « modifiera » pour en fabriquer d'autres. Ci-dessous, la chaîne assignée à `quine1` encode une expression Python qui s'auto-reproduit lorsqu'on l'évalue avec **eval**. Autrement dit, **eval**`(quine1)` retourne la chaîne `quine1` : celle-ci est un point-fixe de **eval**. Ci-dessous, l'opérateur Python `%` applique un format à une valeur et produit une chaîne. On exploite le format `%r` qui sera substitué à l'intérieur de la chaîne englobante par la représentation de la valeur donnée en argument. [19] On utilise aussi la possibilité de faire des affectations de variables avec l'opérateur « `=:` » dans des expressions de uplet. [20]

```
quine1 = "(a:='(a:=%r,a%%a)[-1]',a%a)[-1]"
assert (quine1==eval(quine1))
```

Voilà une variante avec une instruction qui s'affiche elle-même (ici, on construit en quelque sorte un point-fixe de **print**).

```
print((a:='print((a:=%r,a%%a)[-1])',a%a)[-1])
```

Et voilà une autre instruction qui s'affiche elle-même...

```
(a:='(a:=%r,print(a%%a))[-1]',print(a%a))[-1]
```

Exercice 41 [Optionnel]. En suivant les modèles ci-dessus, construire les quines suivants :
(1) un entier `quine2` point-fixe de `eval_code` ; autrement dit, il vérifie `eval_code(quine2)==quine2`.
(2) un entier `is_not_me` qui représente une application sachant distinguer son code entier des autres entiers ; autrement dit, pour tout `n`, l'exécution de `eval_code(is_not_me)(n)` termine sans erreur et calcule `(0 if n==is_not_me else 1)`.
Attention, on pourrait penser à des solutions très simples, utilisant directement une forme de *récursivité*, comme :

```
quine2 = str2int("quine2")
```

Sauf que cette mauvaise solution « *triche* », car elle donne une définition du nombre `quine2` qui n'est pas indépendante du nom qui lui est donné. La récursivité est bien une forme d'auto-référence, mais ce n'est pas ce qui demandé ici. On attend ici en fait que le code suivant fonctionne aussi sans erreur (et même si on remplace le « `aux` » ci-dessous par autre chose) :

```
aux = quine2
quine2 = 0
assert (aux == eval_code(aux))
```

Or, ce code provoque une erreur avec la mauvaise solution ci-dessus.

18. Voir https://fr.wikipedia.org/wiki/Quine_(informatique).
19. Voir https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting.
20. Une expression de uplet Python comme « `(e1,e2,e3,e4)[-1]` » retourne la valeur de la composante de droite du uplet, c'est-à-dire ici le résultat du calcul de « `e4` », après avoir évaluer les autres expressions de gauche à droite.