# TD calculabilité : éléments de correction

## Exercice 1

L1 : On efface (remplace par $B$) le symbole de la première partie, et on «efface» (remplace par $X$) le symbole correspondant de la seconde partie. Il faut des états différents selon qu'on a rencontré un $a$ ou un $b$ en 1e partie. Quand on a épuisé la première partie, on est sur le $c$, on vérifie qu'il n'y a que des $X$ derrière.
Dans ce qui suit, on a : $\alpha \in \{a, b\}$ et $\beta \in \{a, b, c, X\}$.

$$\delta(q_0, c) = (q_f, c, D)$$
$$\delta(q_f, X) = (q_f, X, D)$$
$$\delta(q_f, B) = (f, B, S)$$
$$\delta(q_0, a) = (q_a, B, D) \qquad \delta(q_0, b) = (q_b, B, D)$$
$$\delta(q_a, \alpha) = (q_a, \alpha, D) \qquad \delta(q_b, \alpha) = (q_b, \alpha, D)$$
$$\delta(q_a, c) = (q_{ac}, c, D) \qquad \delta(q_b, c) = (q_{bc}, c, D)$$
$$\delta(q_{ac}, X) = (q_{ac}, X, D) \qquad \delta(q_{bc}, X) = (q_{bc}, X, D)$$
$$\delta(q_{ac}, a) = (q_2, X, G) \qquad \delta(q_{bc}, b) = (q_2, X, G)$$
$$\delta(q_2, \beta) = (q_2, \beta, G)$$
$$\delta(q_2, B) = (q_0, B, D)$$

L2 : Il faut commencer par arriver à trouver le milieu du mot (donc à le couper en deux).
*Approche 1* (intéressante algorithmiquement) :
1. on compte les symboles ($n$, codé en unaire plus loin sur le ruban), en remplaçant $a$ par $X$, $b$ par $Y$.

$$\delta(q_0, B) = (f, B, S)$$
$$\delta(q_0, a) = (q_1, X, D) \qquad \delta(q_0, b) = (q_1, Y, D)$$
$$\delta(q_1, a) = (q_1, a, D) \qquad \delta(q_1, b) = (q_1, b, D)$$
$$\delta(q_1, 1) = (q_1, 1, D) \qquad \delta(q_1, B) = (q_2, 1, G)$$
$$\delta(q_2, 1) = (q_2, 1, G) \qquad \delta(q_2, a) = (q_2, a, G) \qquad \delta(q_2, b) = (q_2, b, G)$$
$$\delta(q_2, X) = (q_3, X, D) \qquad \delta(q_2, Y) = (q_3, Y, D)$$
$$\delta(q_3, a) = (q_1, X, D) \qquad \delta(q_3, b) = (q_1, Y, D)$$

Ici on a fini la première étape, on est passé (e.g.) de «$q_0abbabb$» à «$XYYXYYq_3111111$»
2. on divise $n$ par 2 (sous-algo intéressant, qui ici vérifie en plus que $n$ est pair)

$$\delta(q_3, 1) = (q_4, X, D)$$
$$\delta(q_4, 1) = (q_4, 1, D) \qquad \delta(q_4, B) = (q_5, B, G)$$
$$\delta(q_5, 1) = (q_6, B, G)$$
$$\delta(q_6, 1) = (q_6, 1, G) \qquad \delta(q_6, X) = (q_3, 1, D)$$

Ici on a fini la 2e étape, on est arrivé (e.g) à «$XYYXYY111q_3$»
3. en se servant des 1 qui restent ($\frac{n}{2}$) on va passer à «$XYYabb$»

$$\delta(q_3, B) = (q_7, B, G)$$
$$\delta(q_7, 1) = (q_8, B, G)$$
$$\delta(q_8, 1) = (q_8, 1, G) \qquad \delta(q_8, a) = (q_8, a, G) \qquad \delta(q_8, b) = (q_8, b, G)$$
$$\delta(q_8, X) = (q_9, a, D) \qquad \delta(q_8, Y) = (q_9, b, D)$$
$$\delta(q_9, a) = (q_9, a, D) \qquad \delta(q_9, b) = (q_9, b, D) \qquad \delta(q_9, 1) = (q_9, 1, D)$$
$$\delta(q_9, B) = (q_7, B, G)$$

Ici on est en «$XYYabq_7b$». Le plus dur est fait, le reste est une adaptation de la MT pour L1.
*Approche 2* : on compte les paires de symboles, et on a directement $\frac{n}{2}$ :

$$\begin{array}{c}
\delta(q_0, B) = (f, B, S) \\
\delta(q_0, a) = (q_1, X, D) \qquad \delta(q_0, b) = (q_1, Y, D) \\
\delta(q_1, a) = (q_2, X, D) \qquad \delta(q_1, b) = (q_2, Y, D) \\
\delta(q_2, a) = (q_2, a, D) \qquad \delta(q_2, b) = (q_2, b, D) \\
\delta(q_2, 1) = (q_2, 1, D) \qquad \delta(q_2, B) = (q_3, 1, G) \\
\delta(q_3, 1) = (q_3, 1, G) \qquad \delta(q_3, a) = (q_3, a, G) \qquad \delta(q_3, b) = (q_3, b, G) \\
\delta(q_3, X) = (q_4, X, D) \qquad \delta(q_3, Y) = (q_4, Y, D) \\
\delta(q_4, a) = (q_1, X, D) \qquad \delta(q_4, b) = (q_1, Y, D)
\end{array}$$

Ici on est en «$XYYXYYq_4111$» donc similaire à la fin de la 2e étape de l'approche 1...
*Approche 3* : pas besoin de compter, on fait évoluer le mot comme suit :

$$abbabb \to XbbabT \to XYbaTT \to XYYZTT$$

Ensuite les $X$ et les $Z$ «seront» des $a$, les $Y$ et les $T$ «seront» des $b$ :

$$\begin{array}{c}
\delta(q_0, B) = (f, B, S) \\
\delta(q_0, a) = (q_1, X, D) \qquad \delta(q_0, b) = (q_1, Y, D) \\
\delta(q_1, a) = (q_1, a, D) \qquad \delta(q_1, b) = (q_1, b, D) \\
\delta(q_1, Z) = (q_2, Z, G) \qquad \delta(q_1, T) = (q_2, T, G) \qquad \delta(q_1, B) = (q_2, B, G) \\
\delta(q_2, a) = (q_3, Z, G) \qquad \delta(q_2, b) = (q_3, T, G) \\
\delta(q_3, a) = (q_3, a, G) \qquad \delta(q_3, b) = (q_3, b, G) \\
\delta(q_3, X) = (q_0, X, D) \qquad \delta(q_3, Y) = (q_0, Y, D)
\end{array}$$

Ici on est en «$XYYq_0ZTT$» et le tour est joué...
Note : on pourrait «retransformer» les $Z$ et les $T$ «au retour», en écrivant :

$$\delta(q_1, Z) = (q_2, a, G) \qquad \delta(q_1, T) = (q_2, b, G)$$

pour finir en «$XYYq_0abb$»

## Exercice 2

Pour un AFD $A = (Q, V, q_0, F, \delta)$ on a $m = (Q', \Gamma, V, B, q_0, F', \delta')$ avec :
$Q' = Q \cup \{f\}$ (nouvel état)
$\Gamma = V \cup \{B\}$ (nouveau symbole, le blanc)
$F' = \{f\}$
Si $\delta(q, X) = p$ alors $\delta'(q, X) = (p, X, D)$
Si $q \in F$ alors $\delta(q, B) = (f, B, S)$
Remarque (dite en cours) : on passe d'une reconnaissance où on décide *sur* le dernier symbole (éventuel) du mot à une reconnaissance où on décide *au-delà* du dernier symbole (éventuel), ici quand on arrive sur $B$... Dans la terminologie d'Algo1, ça s'appelle passer d'un modèle 2 à un modèle 1...

## Exercice 3

**Fonction** $M$ : l'entrée est codée sous la forme $\tilde{x}\#\tilde{y}$, avec $\tilde{0} = \varepsilon$.
Note : on ne vérifie pas que l'entrée est effectivement dan le bon format, i.e. que la config. initiale est de la forme «$q_0 1^* \# 1^*$».
On peut définir $M$ par induction :

$$\begin{gathered} 0My = 0 \\ (x+1)M0 = (x+1) \\ (x+1)M(y+1) = xMy \end{gathered}$$

la définition donne l'algo, donc la MT. *Note :* avec 2 rubans (1 pour $x$ et pour le résultat, l'autre pour $y$) c'est moins «drôle» car trop facile...
$\delta(q_0, \#) = (q_e, B, D) \to x = 0$ ... effacer le #...
$\delta(q_e, 1) = (q_e, B, D) \to$ ... puis effacer $y$
$\delta(q_e, B) = (f, B, S) \to$ ... fini (s'il faut un état final)
$\delta(q_0, 1) = (q_1, 1, D) \to$ on effacera un 1 de $x$ au retour si besoin
$\delta(q_1, 1) = (q_1, 1, D)$
$\delta(q_1, \#) = (q_2, \#, D) \to$ séparateur
$\delta(q_2, 1) = (q_3, 1, D) \to y \neq 0$, sinon c'est fini...
$\delta(q_2, B) = (q_f, B, G) \to$ ... et il faut peut-être effacer...
$\delta(q_f, \#) = (f, B, S) \to$ ... le # (dépend du modèle de MT)
$\delta(q_3, 1) = (q_3, 1, D)$
$\delta(q_3, B) = (q_4, B, G)$
$\delta(q_4, 1) = (q_5, B, G) \to y := y - 1$
$\delta(q_5, 1) = (q_5, 1, G) \to$ reculer en passant tous les 1...
$\delta(q_5, \#) = (q_5, \#, G) \to$ ... ainsi que le séparateur
$\delta(q_5, B) = (q_6, B, D)$
$\delta(q_6, 1) = (q_0, B, D) \to x := x - 1$ on a effacé le 1 de $x$...

**Fonction** $P$ : ne pas traiter en entier ; mentionner que :

- 3 rubans plus simple (1 pour $x$, 1 pour $y$, 1 pour résultat), sinon des allers-retours à programmer...
- procéder de poids faible vers poids fort (comme à la main) ; comme l'énoncé ne le précise pas, on peut même supposer que les nombres sont écrits avec les poids faibles en tête, ça simplifie un peu ; sinon il faut commencer par aller au bout de $x$ et de $y$ (en en profitant pour éliminer les 0 excédentaires en tête), puis reculer...
- on aura des états différents selon la retenue à propager...
- il faut bien réfléchir aux cas d'arrêt (e.g. ne pas oublier qu'on peut «épuiser» un nombre avant l'autre...)

## Exercice 4

Noter qu'implicitement $f$ est totale (et $g$ aussi car $f$ bijective).
Si on a une fonction Python

```
def realise_f(x): ... ## Nat -> Nat
```

qui réalise $f$, alors on peut écrire

```
def realise_g(y): ## Nat -> Nat
    x = 0
    while realise_f(x)!= y:
        x += 1
    return x
```

D'un point de vue MT, pour avoir $m_g$ qui réalise $g$ : on énumère les entiers sur un ruban, et on simule $m_f$ (qui réalise $f$) sur ces entiers, jusqu'à trouver $y$ en sortie.
Si $f$ est surjective mais pas injective, alors $g(y) = \mathsf{Min}\{x : f(x) = y\}$
Si $f$ n'est pas surjective, alors $\exists y : \forall x : f(x) \neq y$ et `Fg` ne termine pas sur $y$. $g$ reste partielle calculable, avec $\mathsf{Dom}(g) = \mathsf{Im}(f)$.

## Exercice 5

1. Note : $\mathsf{R}$ fermé par complémentaire : vu en cours...
$\mathsf{R}$ fermé par intersection : $A \in \mathsf{R} \wedge B \in \mathsf{R} \Rightarrow A \cap B \in \mathsf{R}$. Soit
`def decide_A(w): ... ## Sigma* -> Bool` qui décide $A$ et
`def decide_B(w): ... ## Sigma* -> Bool` qui décide $B$. Alors

```
def decide_A_inter_B(w): ## Sigma* -> Bool
    return decide_A(x)and decide_B(w)
```

décide $A \cap B$.
En termes de MT : on exécute $m_A$, puis $m_B$ seulement si $m_A$ accepte. En pratique il faut 2 rubans avec $w$ sur chacun ; on exécute $m_A$ sur le premier, puis, si elle accepte on exécute $m_B$ sur le 2e (car $m_A$ a sans doute modifié son ruban...)
$\mathsf{R}$ fermé par union : $A \in \mathsf{R} \wedge B \in \mathsf{R} \Rightarrow A \cup B \in \mathsf{R}$.
C'est aussi facile :

```
def decide_A_union_B(w): ## Sigma* -> Bool
    return decide_A(x)or decide_B(w)
```

décide $A \cup B$.
En termes de MT : on exécute $m_A$, puis $m_B$ seulement si $m_A$ refuse. Comme ci-dessus en termes de rubans...
3. $E \in \mathsf{RE} \wedge \bar{E} \in \mathsf{RE} \Rightarrow E \in \mathsf{R}$ vu en cours...
2.
$\mathsf{RE}$ fermé par intersection : $A \in \mathsf{RE} \wedge B \in \mathsf{RE} \Rightarrow A \cap B \in \mathsf{RE}$. Soit
`def accepte_A(w): ... ## Sigma* -> Bool` qui accepte $A$ mais peut ne pas terminer si $w \notin A$, et
`def accepte_B(w): ... ## Sigma* -> Bool` qui accepte $B$ mais peut ne pas terminer si $w \notin B$. Alors

```
def accepte_A_inter_B(w): ## Sigma* -> Bool
    return accepte_A(x)and accepte_B(w)
```

accepte $A \cap B$. En termes de MT, c'est comme pour `decide_A_inter_B`, sauf qu'ici on ne terminera pas sur $w$ si `accepte_A` ne termine pas sur $w$ ou si `accepte_A` accepte $w$ et puis `accepte_B` ne termine pas sur $w$.
$\mathsf{RE}$ fermé par union : $A \in \mathsf{RE} \wedge B \in \mathsf{RE} \Rightarrow A \cup B \in \mathsf{RE}$.
Ici problème : on ne peut pas commencer par tester $w \in A$ (par `accepte_A` ou $m_A$), car on peut ne pas s'arrêter et donc ne jamais pouvoir tester $w \in B$.
Il faut faire comme en cours : faire tourner les programmes/MT en parallèle et attendre que l'un réponde OK... En termes de MT : état = couple d'états, 2 rubans...
Note : on aurait aussi pu faire tourner les programmes en parallèle pour les problèmes précédents...
Note : on peut évidemment écrire aussi tout ça en Python, C... avec de la programmation parallèle en laissant faire une partie du boulot à l'OS qui fractionne les temps de calcul... Par

contre, obtenir un programme parallèle vraiment synchrone (au sens où on peut construire $m_{A\cup B}$ à partir de $m_A$ et $m_B$ avec un pas de l'une et un pas de l'autre à chaque fois) est impossible à obtenir à partir de deux fonctions Python/C... mais ici ce n'est pas important, l'essentiel étant que chacune ait un peu de temps, de temps en temps, pour avancer un peu...
On peut aussi utiliser `termine_borné` (ou sa variante `accepte_borné`)...

## Exercice 6

Si $L$ est fini, comme il existe une infinité de fonctions totales calculables (e.g. les fonctions constantes) le résultat est immédiat.
Si $L$ est infini, par définition c'est un ensemble de mots/phrases/programmes sur un vocabulaire fini, donc $L$ est infini dénombrable, donc tout mot/phrase/programme de $L$ est repérable par un unique entier naturel $n$ ; le programme correspondant est appelé $p_n$ ; il calcule une fonction $f_n$ (les $f_n$ ne sont pas forcément distinctes — plusieurs programmes calculent la même fonction — mais c'est sans importance). L'idée est de trouver une fonction $g$, totale calculable, et distincte de chaque $f_n$. Pour cela il suffit qu'elle diffère de $f_n$ en au moins un point... D'où diagonalisation... au point $n$... Soit donc

$$g \;:\; \begin{array}{rcl} \mathbb{N} & \rightarrow & \mathbb{N} \\ n & \mapsto & f_n(n)+1 \end{array}$$

$g$ est trivialement totale calculable — quoique, à partir de $n$, il faut pouvoir trouver $p_n$ (qui calcule $f_n$) donc pouvoir énumérer, dans l'ordre, les $p_i$, ce qui implique que $L$ soit décidable (parmi les mots/phrases sur son vocabulaire on peut décider lesquel(les) sont des programmes...) C'est le boulot d'un analyseur syntaxique...
Par ailleurs $g$ diffère de chaque $f_n$ en au moins un point... d'où la réponse.
Noter qu'il est essentiel que les $f_n$ soient totales, car sinon, pour celles où $f_n(n)$ est indéfini, on a $g(n)$ indéfini, et donc $g$ ne diffère pas forcément de $f_n$...
*Question subsidiaire* : trouver l'erreur dans le raisonnement suivant en supposant que $L$ permet aussi de programmer des fonctions partielles calculables (i.e. les $f_n$ peuvent être indéfinies en certains points).
Soit $g \;:\; \begin{array}{rcl} \mathbb{N} & \rightarrow & \mathbb{N} \\ n & \mapsto & \begin{cases} f_n(n)+1 & \text{si } f_n(n) \text{ est définie} \\ 0 & \text{sinon} \end{cases} \end{array}$
$g$ est totale calculable (on peut encore ici trouver $p_n$ a partir de $n$.)
$g$ est différente de chaque $f_n$ : si $f_n$ est définie au point $n$ alors $g(n) \neq f_n(n)$ ; si $f_n$ est indéfinie au point $n$, $g$ est définie au point $n$ (et vaut 0, mais c'est sans importance)... D'où la réponse !

## Exercice 7

$g$ est une fonction car $\mathsf{Max} : \mathcal{P}(\mathbb{N}) \rightarrow \mathbb{N}$ en est une... Plus précisément : pour $X \in \mathcal{P}(\mathbb{N})$, si $X$ est borné non vide ($\Leftrightarrow$ fini non vide) $\mathsf{Max}(X)$ existe et est unique, sinon ($X$ vide ou infini) $\mathsf{Max}(X)$ est indéfini...
Toute MT avec $\delta = \emptyset$ (et $q_o \in F$ si on utilise la notion d'état final) calcule l'identité, donc $E_n$ est non vide — noter le $n+1$, sans lequel $E_0$ serait vide (il n'existe pas de MT à 0 état) — et $\mathsf{Max}(E_n)$ vaut au moins $n$ (mais c'est sans intérêt ici).
Au renommage des états près (0 .. $n$), avec 0 état initial et si on n'utilise pas la notion d'état final, vu que pour tous $q$ ($n+1$ valeurs possibles) et $X$ (3 valeurs possibles) on a

$$\delta(q,X) = (p,Y,M) \qquad \text{ou indéfini}$$

avec $n+1$ valeurs possibles pour $p$ et 3 valeurs pour $Y$ et $M$ respectivement, il existe donc

$$((n+1)\times 3\times 3+1)^{(n+1)\times 3} = (9n+10)^{3(n+1)}$$

MT à $n+1$ états d'alphabet $\{0,1,B\}$ (ce nombre est à multiplier par $2^{n+1}$ si on utilise la notion d'état final). Donc, le renommage des états ne changeant pas la fonction calculée, $E_n$ est fini.
Donc $g$ est une fonction totale.
$g$ n'est pas calculable, en considérant $f(n) = g(n)+1$ qui le serait aussi... et dans ce cas $f$ serait réalisée par une certaine MT $m$ (qu'on peut supposer d'alphabet $\{0,1,B\}$), à $k+1$ états... alors $f(k) = g(k)+1$, donc $g(k) < f(k)$, ce qui contredit la définition de $g(k)$ qui impose $g(k) \geq f(k)$...

## Exercice 8

Note : ici on identifie les MT $m$ avec leur code et avec l'entier représenté...

Soit $f(n) =$ si $(h(n) = 0)$ alors 1 sinon indéfini.

Si $h$ est (totale) calculable, alors $f$ est (partielle) calculable. Donc il existe une MT $m$ qui réalise $f$, et qui ne s'arrête (avec résultat forcément $= 1$) que sur son domaine.
Donc $f(m) = 1 \Leftrightarrow m$ s'arrête sur l'entrée $m$.
Or $f(m) = 1 \Leftrightarrow h(m) = 0 \Leftrightarrow \mathsf{MTU}(m,m)$ ne s'arrête pas $\Leftrightarrow m$ ne s'arrête pas sur l'entrée $m$.
D'où une contradiction (noter les $\Leftrightarrow$ qui dispensent de traiter le cas où $f(m)$ est indéfini...)

On pourrait aussi dire :
Soit $f(n) = 1 - h(n)$. Avec l'hypothèse d'une $\mathsf{MTU}$ qui s'arrête ssi elle accepte, $f$ serait la fonction caractéristique de $L_d$, qui n'est pas récursif (même pas r.e.)
Encore plus simple : $h$ est la fonction caractéristique de $\overline{L_d}$ qui n'est pas récursif (sinon $L_d$ le serait...)

## Exercice 9

Traité en cours, d'une autre façon que celle suggérée par l'indication.

Ici on a $H = \{n : h(n) = 1\} = \{n : \mathsf{MTU}(n,n) \text{ s'arrête}\}$.
$H$ est r.e. (appeler $\mathsf{MTU}$ sur $(n,n)$, accepter quand arrêt). Si $H$ était récursif, sa fonction caractéristique $h$ serait calculable... Donc $\overline{H}$ n'est pas r.e. (rappel : sinon $H \in \mathsf{RE} \wedge \overline{H} \in \mathsf{RE} \Rightarrow H \in \mathsf{R}$).

On peut aussi remarquer que $\overline{H} = L_d$...
Tout ça pour illustrer qu'on peut prendre les problèmes de diverses façons.

## Exercice 10

On appelle NEIP («Non-Empty Intersection Problem») le problème (sur les GHC LL(1)) «$L(G_1) \cap L(G_2) \neq \emptyset$».
Les instances sont les couples de GHC LL(1) $(G_1, G_2)$[1]
Les instances *positives* sont celles pour lesquelles $L(G_1) \cap L(G_2) \neq \emptyset$.
On opère donc par réduction de PCP à NEIP. En profiter pour rappeler le sens (et le «sens») d'une réduction...
Soit $IP = (V; A = w_1, ...w_k; B = x_1, ...x_k)$ une instance de PCP. L'objectif est de *construire* $IN = (G_A, G_B)$ instance de NEIP telle que

[1] vu que, d'une part on peut définir l'ensemble des GHC comme un sous-ensemble décidable d'un certain $\Sigma^*$ et que, d'autre part on peut *décider* si une GHC est LL(1), on a bien affaire à un problème de décision algoritmique bien posé.

$IP$ a une solution ($IP$ instance positive de PCP) $\Leftrightarrow$ $IN$ a une solution ($IN$ instance positive de NEIP).

Idée de base : produire par $G_A$ (resp. $G_B$) les séquences non vides de $w_i$ (resp. $x_i$) (pour l'instant on ne se préoccupe pas trop du caractère LL(1)) :

$$\begin{array}{lcl} G_A & : & S_A \rightarrow w_1S_A \mid ... \mid w_kS_A \mid w_1 \mid ... \mid w_k \\ G_B & : & S_B \rightarrow x_1S_B \mid ... \mid x_kS_B \mid x_1 \mid ... \mid x_k \end{array}$$

Problème : les longueurs des dérivations peuvent différer. On peut faire remarquer que les grammaires étant linéaires à droite, les langages sont réguliers, les grammaires donnent les automates, et la non-vacuité de l'intersection des langages réguliers est décidable...
Ex : $w_1 = aa$, $x_1 = a$ : $S_A \Rightarrow aa$ et $S_B \Rightarrow aS_B \Rightarrow aa$

Idée suivante : ajouter un «compteur» de réécritures :

$$\begin{array}{lcl} G_A & : & S_A \rightarrow w_1S_A\# \mid ... \mid w_kS_A\# \mid w_1\# \mid ... \mid w_k\# \\ G_B & : & S_B \rightarrow x_1S_B\# \mid ... \mid x_kS_B\# \mid x_1\# \mid ... \mid x_k\# \end{array}$$

Attention : tous les # doivent être regroupés après tous les $w_i/x_i$.
Avec l'exemple précédent ok :
$S_A \Rightarrow aa\#$ mais $S_B \Rightarrow aS_B\# \Rightarrow aa\#\#$
Problème : l'ordre des indices n'a pas à être identique.
Ex : $w_1 = a, w_2 = b$, $x_1 = b, x_2 = a$ : $S_A \Rightarrow aS_A\# \Rightarrow ab\#\#$ et $S_B \Rightarrow aS_B\# \Rightarrow ab\#\#$

Conclusion : il faut en plus «coder les indices» dans les dérivations.
Noter que $k$ est fixé. On prend donc $k$ nouveaux terminaux $\#_1, ...\#_k$ :

$$\begin{array}{lcl} G_A & : & S_A \rightarrow w_1S_A\#_1 \mid ... \mid w_kS_A\#_k \mid w_1\#_1 \mid ... \mid w_k\#_k \\ G_B & : & S_B \rightarrow x_1S_B\#_1 \mid ... \mid x_kS_B\#_k \mid x_1\#_1 \mid ... \mid x_k\#_k \end{array}$$

Sur un exemple simple ($w_1 = a, w_2 = ba, x_1 = ab, x_2 = a$) on voit :
$S_A \Rightarrow aS_A\#_1 \Rightarrow aba\#_2\#_1$ et $S_B \Rightarrow abS_A\#_1 \Rightarrow aba\#_2\#_1$
De façon générale :
$L(G_A) = \{w_{i_1}...w_{i_n}\#_{i_n}...\#_{i_1} \quad \forall n > 0, \forall i_1...i_n \in [1, k]^n\}$
$L(G_B) = \{x_{i_1}...x_{i_n}\#_{i_n}...\#_{i_1} \quad \forall n > 0, \forall i_1...i_n \in [1, k]^n\}$
et donc $\exists n > 0, \exists i_1...i_n \in [1, k]^n : w_{i_1}...w_{i_n} = x_{i_1}...x_{i_n} \quad \Leftrightarrow \quad L(G_A) \cap L(G_B) \neq \emptyset$
Du coup, ici on a prouvé que la variante de NEIP *pour les GHC quelconques* est indécidable. Mais ça ne nous dit pas automatiquement que NEIP l'est si on se *restreint* aux GHC LL(1)[2]. Ça serait le cas si $G_A$ et $G_B$ construites ci-dessus étaient LL(1), mais ça ne l'est pas. On pourrait par contre factoriser *systématiquement* les règles dans $G_A$ et $G_B$ et ça suffirait[3], mais ce n'est pas simple ($w_i$ et $w_j$ peuvent avoir un préfixe commun, et $w_i$ et $w_k$ un autre, on peut même avoir $w_i = w_l$...) Le plus simple est de partir de la solution qu'on a obtenue, et de «faire passer» les $\#_i$ devant...[4] Ça donne :

[2]dans le même ordre d'idée, voir la remarque ci-dessus sur les GHC linéaires à droite
[3]il n'y a pas de règles récursives à gauche
[4]on aurait aussi pu tout de suite fournir cette solution...

$$\begin{array}{llllllllllll} G_A & : & S_A \to \#_1 S_A w_1 & \mid & ... & \mid & \#_k S_A w_k & \mid & \#_1 w_1 & \mid & ... & \mid & \#_k w_k \\ G_B & : & S_B \to \#_1 S_B x_1 & \mid & ... & \mid & \#_k S_B x_k & \mid & \#_1 x_1 & \mid & ... & \mid & \#_k x_k \end{array}$$

Y'a plus qu'à factoriser :

$$\begin{array}{rll} G_A & : & S_A \to \#_1 Y_1 \quad \mid \quad ... \quad \mid \quad \#_k Y_k \\ & (\forall i) & Y_i \to S_A w_i \quad \mid \quad w_i \\ G_B & : & S_B \to \#_1 Z_1 \quad \mid \quad ... \quad \mid \quad \#_k Z_k \\ & (\forall i) & Z_i \to S_B x_i \quad \mid \quad x_i \end{array}$$

Vu que $\mathit{Prem}(S_A) = \mathit{Prem}(S_B) = \{\#_i\}$ et que $\{\#_i\} \cap V = \emptyset$ on vérifie aisément que $G_A$ et $G_B$ sont LL(1).
On a donc maintenant :
$L(G_A) = \{\#_{i_n}...\#_{i_1} w_{i_1}...w_{i_n} \quad \forall n > 0, \forall i_1...i_n \in [1,k]^n\}$
$L(G_B) = \{\#_{i_n}...\#_{i_1} x_{i_1}...x_{i_n} \quad \forall n > 0, \forall i_1...i_n \in [1,k]^n\}$
D'où la conclusion (NEIP déc. $\Rightarrow$ PCP déc., donc PCP indéc. $\Rightarrow$ NEIP indéc.)
Ensuite, il s'agit de montrer l'indécidabilité du problème de l'ambiguïté des GHC. Pour cela il « suffit » de réduire NEIP à ce problème-ci (appelons-le Amb).
La réduction consiste donc, à partir de 2 GHC $G_1$ et $G_2$, LL(1) mais *quelconques*, à *construire* une GHC $G$ telle que
$(G_1, G_2) \in \text{NEIP} \Leftrightarrow \ G \in \text{Amb}$
ou, autrement dit, en supposant $V_{T1} = V_{T2}$ (ce qui est toujours possible), telle que
$(\exists w : S_1 \Rightarrow^* w \ \wedge \ S_2 \Rightarrow^* w) \Leftrightarrow (\exists u \in V_T^*$ avec au moins 2 arbres de dérivations dans $G)$
Cette fois, la solution la plus intuitive est la bonne. Il suffit en effet de prendre $V_T = V_{T1} = V_{T2}$, $V_N = V_{N1} \cup V_{N2} \cup \{S\}$, en supposant $V_{N1} \cap V_{N2} = \emptyset$ (quitte à renommer des non-terminaux) et $S$ un nouveau non-terminal qui sera l'axiome, et $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2 \cup \{S \to S_1, S \to S_2\}$.
$G_1$ et $G_2$ étant LL(1), elles sont non ambiguës, et donc la seule possibilité d'ambiguïté dans $G$ est pour un mot $w$ qu'on peut dériver en commençant par appliquer *l'une ou bien l'autre* des deux règles ajoutées ci-dessus, qui sont les seules qu'on peut trouver à la racine de l'arbre...
Donc $G$ est ambiguë ssi $\exists w : S \Rightarrow S_1 \Rightarrow^* w$ et $S \Rightarrow S_2 \Rightarrow^* w$, CQFD.

## Exercice 11

Vu en cours, mais on peut en rediscuter (e.g. évoquer les différences de notation entre le cours et l'énoncé de l'exo...) et aussi bien faire noter le «*Rappel*». Aussi faire chercher des propriétés triviales... autres que «$L$ est r.e.» (toujours vraie) ou «$L$ n'est pas r.e.» (toujours fausse)...

## Exercice 12

Identifier $m$ et $<m>$
1. Appliquer Rice : exhiber un langage r.e. $L$ (donc une MT $m$) qui a la propriété, et un autre $L'$ (donc une MT $m'$) qui ne l'a pas... Exemples :
1.1 : $L = \{w\}, L' = \emptyset$
1.2 : $L = \{\varepsilon\}, L' = \emptyset$
1.3 : $L = \emptyset, L' = \{\varepsilon\}$
1.4 : $L = \{a\}^*, L' = \emptyset$
1.5 : $L = \emptyset, L' = \{a\}^*$
1.6 : $L = \emptyset, L' = \{a^n b^n\}$

1.7 : $L = \emptyset, L' = \{a^n b^n c^n\}$
1.8 : $L = \emptyset, L' = L_u$
1.9 : $L = L_u, L' = \emptyset$

En fait, il y a un petit piège pour 1.4 : la propriété est triviale pour $k = 0$... Donc $L_0$ est récursif... car on sait que la propriété est vraie pour toutes les MT... Question à méditer : existe-t-il une propriété sémantique des MT, triviale, mais pour laquelle on ne sait pas dire si elle est toujours vraie, ou toujours fausse (auquel cas elle serait indécidable) ?

2. Pour chaque $w$, réduire $L_w$ à $L_u$ ($m \in L_w \Leftrightarrow (w, m) \in L_u$). En Python :

```
def accepte_Lw (m):
  accepte_Lu (m, w)
```

3. Constuire une MT qui énumère les $w_i$ et simule («appelle») $\mathsf{MTU}(m, w)$ sur chacun, et qui accepte dès que $\mathsf{MTU}$ accepte... $\rightarrow$ ne marche pas (imaginer que $m$ ne s'arrête pas sur $\varepsilon$...)) L'idée est bonne, mais il faut aller au-delà... Imaginer une MT qui, pour une $m$ donnée en entrée, énumère les $w_i$, qui simule *un pas* de $m$ sur $w_0$, puis un pas de $m$ sur $w_0$ et un pas de $m$ sur $w_1$, puis un pas de $m$ sur $w_0$, un pas de $m$ sur $w_1$ et un pas de $m$ sur $w_2$, etc... et qui s'arrête en acceptant dès qu'un des $w_i$ est accepté... Exo subsidiaire : écrire une telle MT !!! Note : on peut étendre les MT pour qu'elles aient un nombre variable (non borné mais toujours fini) de rubans et de têtes de lecture...

4. $L_e = \overline{L_{ne}}$, or $L_{ne}$ non récursif, donc $L_e$ non r.e.

5. L'idée est la même que pour $L_{ne}$, sauf qu'il faut en plus un compteur de mots acceptés, et la MT s'arrête (en acceptant) dès que le compteur vaut $k$...

6. On opère une réduction de $\overline{L_u}$ vers $L_r$ (attention, là on passe au niveau méta du méta...) : on transforme *par un algorithme* un couple $(m, w)$ en une MT $m'$ telle que $(m, w) \in \overline{L_u} \Leftrightarrow m' \in L_r$. $\overline{L_u}$ n'étant pas r.e. (sinon $L_u \in \mathsf{R}$), on en déduira que $L_r$ n'est pas r.e.
$m'$ (dont l'entrée $x$ est un couple $[p, z]$) commence par simuler $m$ sur $w$ (fait comme $\mathsf{MTU}(m, w)$) puis :

1. si $w$ n'est pas accepté, n'accepte pas $x$ (évident si $m$ ne s'arrête pas sur $w$)
2. si $w$ est accepté, simule $p$ sur $z$ (fait comme $\mathsf{MTU}(p, z)$).

On en déduit :

- si $(m, w) \in \overline{L_u}$ (cas 1.) alors $L(m') = \emptyset \in \mathsf{R}$ donc $m' \in L_r$.
- si $(m, w) \notin \overline{L_u}$ (cas 2.) alors $L(m') = L(\mathsf{MTU}) \notin \mathsf{R}$ donc $m' \notin L_r$.

D'où la conclusion...
En pseudo-code Python, ça donne :

```
def accepte_comp_Lu(m, w):
  def accepte_X(p, z):
    accepte_Lu(m, w)
    accepte_Lu(p, z)
  return accepte_Lr(accepte_X)
```

7. On opère à nouveau une réduction de $\overline{L_u}$ vers $L_{nr}$ (avec les mêmes notations)... en «inversant», mais c'est plus compliqué :
$m'$ est une MT (d'entrée $x = [p, z]$) qui simule, «en parallèle», $m$ sur $w$ et $p$ sur $z$, et accepte ($x$) dès qu'une des deux accepte.
Alors :

- si $(m, w) \in \overline{L_u}$ ($m$ n'accepte pas $w$) alors $L(m') = L(\mathsf{MTU}) = L_u \notin \mathsf{R}$ donc $m' \in L_{nr}$
- si $(m, w) \notin \overline{L_u}$ alors on n'a pas «$m$ n'accepte pas $w$», et donc $m$ accepte $w$ ; par conséquent $m'$ accepte $x$ ($\forall x$), donc $L(m') = \Sigma^* \in \mathsf{R}$ donc $m' \notin L_{nr}$... ouf !!!

  En pseudo-code Python, ça donne :

```
def accepte_comp_Lu(m, w):
   def accepte_X(p, z):
      for k in Nat():
         if accepte_borné(m, w, k)or accepte_borné(p, z, k):
            return
   return accepte_Lnr(accepte_X)
```

## Exercice 13

Ici on a juste un petit problème, car le «problème» porte sur des couples...
1. Réduction triviale : si $L = L'$ était décidable, alors $L = \emptyset$ le serait...
2. si $L \subseteq L'$ était décidable, alors $L = L'$ le serait...
Autre façon de voir :
Pour 1. par exemple : on peut voir le problème $P$ comme un $P_{L'}$ pour $L'$ fixé...

## Exercice 14

Cf. poly