Devoir Surveillé 3

18 novembre 2023

INFORMATIQUE MPI*

DURÉE DE L'ÉPREUVE : **4 heures**

L'usage de la calculatrice et de tout dispositif électronique est interdit.

Ce sujet comporte huit pages numérotées de 1/12 à 12/12

Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il le signale sur sa copie et poursuit sa composition en expliquant les raisons des initiatives qu'il est amené à prendre.

Tournez la page S.V.P.

Vue d'ensemble du sujet

Ce sujet est composé de 4 parties indépendantes, utilisant les langages de programmation C et OCaml.

Les différentes parties sont indépendantes et peuvent être traités dans n'importe quel ordre. Il n'est pas nécessaire d'avoir répondu aux questions d'une partie pour répondre aux questions d'une autre partie.

— La partie I s'intéresse à la non-calculabilité du problème du castor affairé, et à une preuve alternative de la non-décidabilité du problème de l'arrêt.
— La partie II s'intéresse à un jeu à deux joueurs sur les langages dont la preuve de terminaison est donnée par le lemme de Higman.
— La partie III s'intéresse aux tries comme structure de données pour représenter un ensemble de mots, à un jeu à deux joueurs sur les tries, et à une adaptation des tries pour représenter des ensembles de mots infinis.
— La partie IV s'intéresse à la définition par induction des langages réguliers en OCaml, son utilisation pour produire une expression régulière désignant ce langage, puis s'intéresse aux résiduels d'un langage régulier, ce qui donne une méthode alternative pour décider si un mot est dans un langage régulier ou non.

Pour répondre à une question, il est permis de réutiliser le résultat d'une question antérieure, même sans avoir réussi à établir ce résultat. En langage C, il est inutile de rappeler que les entêtes `<assert.h>`, `<stdbool.h>`, etc. doivent être inclus.

Quand l'énoncé demande de coder une fonction, sauf indication explicite de l'énoncé, il n'est pas nécessaire de justifier que celle-ci est correcte ou de tester que des préconditions sont satisfaites.

Le barème tient compte de la clarté des programmes : nous recommandons de choisir des noms de variables intelligibles ou encore de structurer de longs codes par des blocs ou par des fonctions auxiliaires dont on décrit le rôle. Lorsqu'une réponse en pseudo-code est permise, seule la logique de programmation est évaluée, même dans le cas où un code en C a été fourni en guise de réponse.

Partie I. Castors affairés – 20 pts

Dans cette partie, on suppose que l'exécution d'un programme OCaml se fait sur un ordinateur à mémoire infinie. En particulier, on suppose que la représentation des entiers est non bornée.
On dit qu'une fonction $f : \mathbb{N} \to \mathbb{N}$ est *calculable* s'il existe une fonction OCaml `f : int -> int`, dont l'exécution termine toujours, qui calcule les images par f.
Un castor affairé (*busy beaver* en anglais) est un programme dont l'exécution termine toujours et qui calcule une valeur la plus grande possible parmi tous les programmes de même taille. Formellement, on s'intéresse au problème d'optimisation suivant :

Castor affairé :

Instance : un entier naturel n.

Solution : la valeur k renvoyé par un programme OCaml contenant n caractères (parmi les 128 caractères possibles de l'encodage ASCII) dont l'exécution termine toujours et renvoie un entier.

Optimisation : maximiser k.

Par exemple, le programme suivant :

```
let k=99 in k*k*k
```

est un programme de taille 17, qui termine toujours et renvoie 970299. Ce n'est pas un castor affairé, en effet le programme suivant (qui n'est toujours pas un castor affairé) renvoie une valeur plus grande :

```
99999999999999999
```

Pour un entier n donné, on notera $C(n)$ la valeur maximale renvoyée par un programme OCaml de taille n qui termine et renvoie un entier. On pose $C(0) = 0$ par convention.
On cherche à montrer que le problème du castor affairé n'est pas calculable, c'est-à-dire que la fonction C n'est pas calculable.

☐ 1 – Combien existe-t-il de programmes OCaml à n caractères, en supposant qu'on dispose de 128 caractères différents possibles ? Expliquer pourquoi le fait qu'il y en ait un nombre fini ne permet pas de conclure que le problème est calculable.

Réponse 1. Il y a 128^n programmes OCaml à n caractères. On pourrait penser que cela permet de calculer le problème en testant tous ces programmes, mais certains bouclent et on ne peut pas savoir au bout d'un certain nombres de pas si le programme va s'arrêter ou pas. On ne peut donc pas juste faire tourner tous les programmes, même en parallèle.

☐ 2 – Montrer que la fonction C est correctement définie.

Réponse 2. pour $n = 0$, C est défini. Pour $n \geq 1$, C est le maximum d'un ensemble fini non vide car le nombre de programmes de taille n qui terminent et renvoie un entier est fini (cf. question 1), et le programme constitué de n fois le caractère 1 renvoie un entier.

☐ 3 – Montrer que C est une fonction croissante, puis que pour $n \in \mathbb{N}, C(n+2) > C(n)$.

Réponse 3. Soit p le programme de n caractères qui renvoie $C(n)$. Alors p auquel on a rajouté un espace renvoie $C(n)$ et est un programme de $n+1$ caractères, donc $C(n+1) \geq C(n)$. De plus, le programme p auquel on ajoute les caractères `+1` renvoie $C(n)+1$ et contient $n+2$ caractères donc $C(n)+1 \leq C(n+2)$. Soit $C(n+2) > C(n)$.

☐ 4 – Que vaut $C(1)$? Le fait qu'on puisse déterminer cette valeur contredit-il la non-calculabilité du problème ?

Réponse 4. Les seuls programmes de 1 caractère qui renvoie un entier sont ceux contenant un chiffre, le plus grand est 9. Donc $C(1) = 9$. Calculer la valeur de la fonction sur une seule entrée ne suffit pas pour justifier sa calculabilité, il faut exhiber un algorithme qui peut calculer $C(n)$ pour tout n.

On considère $f : \mathbb{N} \to \mathbb{N}$ une fonction calculable.

☐ 5 – Montrer qu'il existe un entier k tel que pour tout $n \in \mathbb{N}, f(n) \leq C(\lceil \log_{10} n \rceil + k)$.

Réponse 5. f est calculable, donc il existe une fonction OCaml `f` qui calcule ses images. On considère le programme suivant :

```
let f = (* code de f *) in
f (* écriture de n *) + 1
```

Si on note k' le nombre de caractères du code de `f`, comme l'écriture de n comporte $\lceil \log_{10}(n) \rceil$ chiffres, le nombre de caractères de ce programme est $k' + \lceil \log_{10}(n) \rceil + 18$, et il renvoie un nombre plus grand que $f(n)$ strictement. Donc $f(n) < C(\lceil \log_{10} n \rceil + k)$, avec $k = k' + 18$.

☐ 6 – Montrer qu'il existe un entier n_0 tel que $f(n_0) < C(n_0)$.

Réponse 6. $n \mapsto n - (\lceil \log_{10} n \rceil + k + 2)$ a pour limite $+\infty$ lorsque n tend vers $+\infty$, donc il existe n_0 tel que $n_0 > \lceil \log_{10} n_0 \rceil + k + 2$, et $f(n_0) \leq C(\lceil \log_{10} n_0 \rceil + k) < C(\lceil \log_{10} n_0 \rceil + k + 2) \geq C(n_0)$.

☐ 7 – Conclure.

Réponse 7. Supposons que C est calculable, alors il existe n_0 tel que $C(n_0) < C(n_0)$. C'est impossible.

On considère le problème suivant :

Arrêt :

Instance : le code source d'un programme OCaml.
Solution : est-ce que l'exécution de ce programme termine ?

☐ 8 – Montrer, en utilisant la non-calculabilité du problème du castor affairé, que le problème **Arrêt** est indécidable.

Réponse 8. Supposons que le problème de l'arrêt est décidable, et soit `halt` une fonction OCaml qui le décide. Alors, la fonction suivante calcule les images par C. Or on sait qu'une telle fonction n'existe pas, donc `halt` ne peut pas exister. (`universal` est une fonction qui permet d'évaluer un programme dont le code est donné avec le type `string`)

```
let rec prod a b = match a with
  | [] -> []
  | t::q -> prod_elem t b @ prod q b
and prod_elem x b = match b with
  | [] -> []
  | t::q -> (x^t) :: prod_elem x q

let programs_1 =
  let res = ref [] in
  for i = 0 to 127 do
    let c = char_of_int i in
    let s = String.make 1 c in
    res := s :: !res
  done; !res

let rec all_programs n =
  if n = 0 then [""]
  else
    prod (all_programs (n-1)) programs_1

let beaver n =
  List.fold_left
    (fun acc x -> if halt x then max (universal x) acc else acc)
    0 (all_programs n)
```

On a effectué une réduction (au sens de Turing) du problème du castor affairé au problème de l'arrêt.

Partie II. Lemme de Higman – 15 pts

On considère le jeu à deux joueurs suivant :

Jeu des sous-mots :

Initialisation : un ensemble de mots $L = \emptyset$;

Coups possibles : chacun leur tour, les joueurs choisissent un mot u dont aucun sous-mot n'est dans L. On met à jour $L \leftarrow L \cup \{u\}$;

Fin de partie : le joueur qui choisit le mot vide ε perds la partie. L'autre joueur gagne.

Par exemple, jouer le mot *truc* empêche de jouer les mots *trucage*, *trouc*, *autruche*, *structure*, *tirebouchon*, *introductif* ou encore *cturtutrcu*.
Voici un exemple de partie sur l'alphabet $\Sigma = \{a, b\}$ gagnée par le joueur 1 :

$$ab - aa - ba - b - a - \varepsilon$$

☐ 9 – Montrer que si l'alphabet ne contient qu'une lettre $\Sigma = \{a\}$, le joueur 1 possède une stratégie gagnante.

Réponse 9. Si le joueur 1 joue a, le seul mot restant qui peut être joué est ε.

On cherche maintenant à montrer par l'absurde que toute partie termine.
Soit u_0 un mot de taille minimal parmi les mots qui commencent une partie infinie.
Pour chaque entier i, on définit u_i comme étant un mot de taille minimale tel que $u_0 - u_1 - \ldots - u_i$ commence une partie infinie. Notons qu'en particulier, $u_0 - u_1 - \ldots$ est une partie infinie.

☐ 10 – Justifier l'existence d'un tel u_0.

Réponse 10. L'ensemble des tailles des mots qui commencent une partie infinie est une partie de $\mathbb{N}$ qui est un ensemble bien fondé, donc admet un minimum. Donc il existe bien un mot u_0 de taille minimal parmi ceux-ci.

☐ 11 – Démontrer qu'il existe une lettre x qui commence une infinité de u_i.

Réponse 11. Supposons que pour toute lettre x, il n'existe qu'un nombre fini $n(x)$ de mots u_i qui commencent par x. Alors la somme des $n(x)$ est finie car l'alphabet est fini, et le nombre de mots u_i est fini, ce qui est en contradiction avec leur définition.

Soit k le plus petit indice tel que u_k commence par x. On considère la suite de mots v_i définit comme suit :

$$v_i = \begin{cases} u_i & \text{si } i < k \\ \text{le } (i-k)\text{-ième mot de la suite } u \text{ commençant par } x\text{, privé de sa première lettre} & \text{sinon} \end{cases}$$

☐ 12 – Montrer une contradiction.

Réponse 12. $(v_i)_{i\in\mathbb{N}}$ est une partie infinie. En effet, s'il existe $i < j$ tel que v_i est sous-mot de v_j, alors $u_{\varphi(i)}$ est sous-mot de $u_{\varphi(j)}$ où $\varphi : x \mapsto x$ si $x < i, i - k$ sinon. Elle contredit la minimalité du choix de u_k.

☐ 13 – Montrer que sur un alphabet à 2 lettres $\Sigma = \{a, b\}$, le joueur 2 a une stratégie gagnante.

Réponse 13. Pour tout mot $u = u_1 \ldots u_n$, on note $\bar{u} = \bar{u}_1 \ldots \bar{u}_n$, avec $\bar{a} = b$ et $\bar{b} = a$. Pour chaque mot u joué par J1, J2 peut jouer $\bar{u}$. En effet, si il existe un $v \in L$ sous-mot de $\bar{u}$, alors $\bar{v}$ est sous-mot de u et a été joué juste avant ou juste après v. Comme la partie se termine et que J2 peut toujours jouer un mot autre que ε, c'est nécessairement J1 qui joue ε et perd.

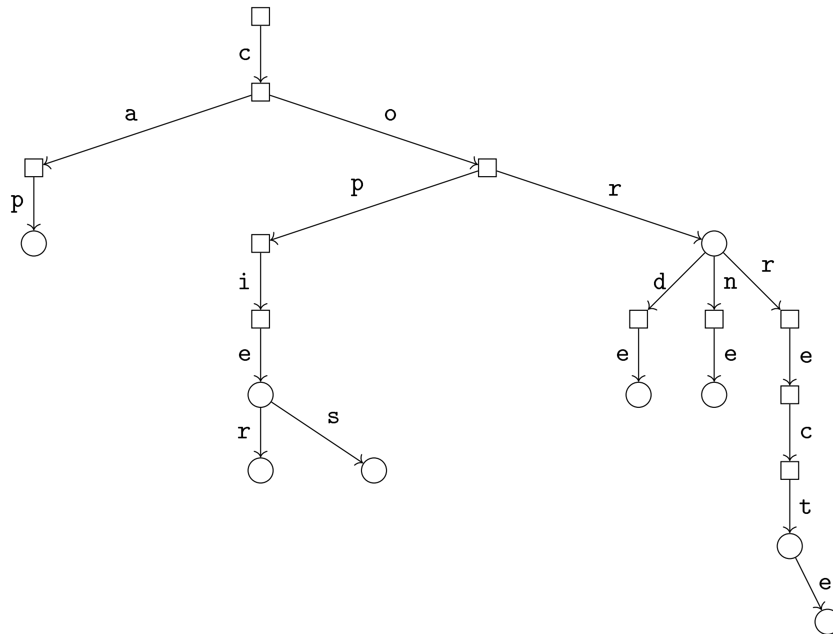
☐ 14 – Généraliser à tout alphabet.

Réponse 14. Si le nombre de lettres de Σ est pair, on peut étendre la définition de $\bar{u}$ en regroupant les lettres par paires et en échangeant ces lettres. Ainsi, le joueur 2 a une stratégie gagnante. Si le nombre de lettres de Σ est impair, le joueur 1 peut se ramener au cas précédent en inversant les rôles en jouant un mot d'une lettre. Donc il a une stratégie gagnante.

Partie III. Trie (C) – 40 pts

Définition : Un *trie* est un type particulier d'arbre $|\Sigma|$-aire représentant un ensemble fini de mots sur l'alphabet Σ. Chaque arête est étiquetée par un symbole de Σ, et chaque nœud est étiqueté par un booléen. Un mot $u = u_1 \dots u_n$ appartient à un trie t, si il existe un chemin de n arêtes issu de la racine dont les arêtes successives ont pour étiquettes respectives les lettres $u_1, u_2, \dots, u_n$ et qui termine à un nœud dont l'étiquette vaut V. Par exemple, un trie contient le mot vide ε si sa racine est étiquetée V.

FIGURE 1 – Exemple de trie contenant 9 mots. Les nœuds étiquetés par V sont représentés par des cercles, ceux étiquetés par F par des rectangles.



☐ 15 – Donner l'ensemble des mots du trie représenté à la figure 1.

Réponse 15. cap, copie, copier, copies, cor, corde, corne, correct, correcte.

Dans le cas général, pour représenter l'ensemble des arêtes partant d'un nœud, la structure de donnée abstraite la plus adaptée est un tableau associatif (ou dictionnaire).

☐ 16 – Nommer deux structures de données concrètes, qui réalisent le type abstrait tableau associatif. Pour chacune d'entre elles, rappeler sans justifier la complexité en temps de l'opération *insertion*.

Réponse 16.

- Table de hachage : complexité en temps de l'insertion $O(n)$ dans le pire des cas, où n est le nombre d'associations, et $O(1)$ en moyenne ;
- Arbres binaires de recherche : complexité en temps de l'insertion $O(d)$ où d est la profondeur de l'arbre ;
- Arbres rouges-noir : complexité en temps de l'insertion $O(\log n)$ où n est le nombre d'associations ;
- Listes d'associations : complexité en temps de l'insertion $O(1)$ si implémentée par des listes chaînées.

On suppose que l'on travaille avec les caractères de la table ASCII, correspondant aux entiers de type `char`, entre 0 et 127. On peut donc utiliser un simple tableau de taille 128 pour représenter les successeurs d'un nœud. On implémente en C les tries par la structure de données suivante :

```
struct node {
  bool lbl;
  struct node *succ[128];
};
typedef struct node* trie;
```

☐ 17 – Donner la valeur du type `trie` qui représente le trie vide, et écrire une fonction `trie eps()` qui renvoie un nouveau trie ne contenant que le mot vide ε, ainsi qu'une fonction `trie singleton(char c)` qui renvoie un nouveau trie ne contenant que le mot d'une lettre c.

Réponse 17. `NULL` représente le trie vide. (le type `trie` est un type pointeur)

```c
trie eps(){
  trie t = malloc(sizeof(struct node));
  t->lbl = true;
  for (int i = 0; i < 128; i = i + 1)
    { t->succ[i] = NULL; }
  return t;
}

trie singleton(char c){
  trie t = malloc(sizeof(struct node));
  t->lbl = false;
  for (int i = 0; i < 128; i = i + 1)
    { t->succ[i] = NULL; }
  t->succ[c] = eps();
  return t;
}
```

☐ 18 – On donne la fonction `eps` suivante qui permet de créer et de renvoyer un nouveau trie ne contenant que le mot vide ε.

```
trie eps(){
  trie t = malloc(sizeof(struct node));
  t->lbl = true;
  for (int i = 0; i < 128; i = i + 1)
    { t->succ[i] = NULL; }
  return t;
}
```

Donner la valeur du type `trie` qui représente le trie vide, et écrire une fonction `trie singleton(char c)` qui renvoie un nouveau trie ne contenant que le mot d'une lettre c.

Réponse 18. `NULL` représente le trie vide. (le type `trie` est un type pointeur)

```c
trie singleton(char c){
  trie t = malloc(sizeof(struct node));
  t->lbl = false;
  for (int i = 0; i < 128; i = i + 1)
    { t->succ[i] = NULL; }
  t->succ[c] = eps();
  return t;
}
```

☐ 19 – Écrire une fonction `bool is_in(trie t, char w[])`, qui prend en entrée un trie t et un mot w et qui renvoie `true` si $w \in t$ et `false` sinon.

Réponse 19.

```c
bool is_in(trie t, char w[]){
  for (int i = 0; t != NULL && w[i] != '\0'; i = i + 1) {
    t = t->succ[w[i]];
  }
  return (t != NULL && t->lbl);
}
```

☐ 20 – Écrire une fonction `trie uni(trie t1, trie t2)`, qui prend en entrée deux tries t_1 et t_2 et qui renvoie un nouveau trie t contenant les éléments de t_1 et les éléments de t_2.

Réponse 20.

```c
trie uni(trie t1, trie t2){
  if (t1 == NULL && t2 == NULL) { return NULL; }
  if (t1 == NULL) {return uni(t2, t1);}
  trie t = malloc(sizeof(struct node));
  t->lbl = t1->lbl || (t2 != NULL && t2->lbl);
  for (int i = 0; i < 128; i = i + 1){
    if (t2 == NULL)
      { t->succ[i] = uni(t1->succ[i], NULL); }
    else
      { t->succ[i] = uni(t1->succ[i], t2->succ[i]); }
  }
  return t;
}
```

☐ 21 – Écrire une fonction `trie cat(trie t1, trie t2)`, qui prend en entrée deux tries t_1 et t_2 contenant les langages T_1 et T_2 et qui renvoie un nouveau trie t contenant exactement les mots de T_1T_2.

Réponse 21.

```c
void free_trie(trie t){
  if (t == NULL) { return; }
  for (int i = 0; i < 128; i = i + 1){
    if (t->succ[i] != NULL) { free(t->succ[i]); }
  }
  free(t);
}

trie cat(trie t1, trie t2){
  if (t1 == NULL || t2 == NULL) { return NULL; }
  trie t = malloc(sizeof(struct node));
  for (int i = 0; i < 128; i = i + 1){
    t->succ[i] = cat(t1->succ[i],t2);
  }
  if (t1->lbl) {
    trie r = uni(t,t2);
    free_trie(t);
    return r;
  }
  return t;
}
```

On peut remarquer qu'un trie dont aucun nœud n'est étiqueté V ne contient aucun mot. Tout sous-arbre ne contenant aucun nœud étiqueté V peuvent donc être remplacé par le pointeur `NULL`.

☐ 22 – Écrire une fonction `void trim(trie t)` qui élague un trie t pour que toutes les feuilles soient étiquetées V.

Réponse 22.

```c
//le type proposé ne permet pas de gérer le cas de base
//voici une proposition de correction de ce qui est attendu
trie trimmed(trie t){
  if (t == NULL) { return NULL; }
  int flag = true;
  for (int i = 0; i < 128; i = i + 1){
    t->succ[i] = trimmed(t->succ[i]);
    if (t->succ[i] != NULL) { flag = false; }
  }
  if (flag) { free_trie(t); return NULL; }
  return t;
}

//et une version respectant les types demandés
void trim(trie t){
  if (t == NULL) { return; }
  for (int i = 0; i < 128; i = i + 1){
    t->succ[i] = trimmed(t->succ[i]);
  }
}
```

Jeu sur un trie

On définit le jeu à deux joueurs suivant :

Premier au langage L :

Initialisation : un mot vide $w = \varepsilon$;
Coups possibles : chacun leur tour, les joueurs ajoutent une lettre au mot w ;
Fin de partie : le jeu s'arête dès que $w \in L$. Le joueur qui a joué le dernier coup a gagné.

Si $\varepsilon \in L$, le jeu n'est pas très intéressant et on notera par convention que le joueur 2 a gagné (puisque le joueur 1 devrait à présent jouer).

☐ 23 – Montrer que dans le cas général, on peut ne pas avoir de gagnant, mais que si le trie est élagué, par exemple avec la fonction `trim` de la question précédente, alors nécessairement un des deux joueurs gagne.

Réponse 23. Erreur d'énoncé ici car il est possible qu'il n'y ait pas de gagnant, même si le trie est élagué. Dès qu'on joue un caractère qui emmène vers le pointeur `NULL`, plus personne ne peut gagner.
Suggestion de correction :
les coups ne sont autorisés que si w est préfixe d'un mot du langage L.
Mais alors ce jeu n'est plus défini par rapport à un trie. Si on veut garder l'esprit de la question, il faut définir le jeu comment autorisant tous les déplacements dans le trie qui ne mènent pas à un pointeur `NULL`. Ainsi si le trie n'est pas élagué, on peut se retrouver dans une branche qui ne contient aucun état final, mais si il est élagué, alors les mots autorisés sont uniquement les préfixes des mots de L. Comme L est fini, la partie termine nécessairement au bout de au plus n étapes où n est la taille du plus grand mot de L. Le mot w est alors un mot de L (car w ne peut pas être préfixe strict d'un mot de L), et le dernier joueur à avoir joué gagne.

Pour ce jeu, on peut encore simplifier le trie en supprimant tous les successeurs d'un nœud étiqueté par V. En effet, si on atteint l'un de ses nœuds, la partie s'arrête.

☐ 24 – Écrire une fonction `void simplify(trie t)` qui prend en entrée un trie élagué et qui supprime tous les successeurs des nœuds étiquetés par V. À la fin, on a donc un trie dont toutes les feuilles et uniquement les feuilles sont étiquetés par V.

Réponse 24.

C
```c
void simplify(trie t){
  if (t == NULL) { return; }
  if (t->lbl) {
    for (int i = 0; i < 128; i = i + 1)
      { t->succ[i] = NULL; }
  }
  else for (int i = 0; i < 128; i = i + 1)
         { simplify(t->succ[i]); }
}
```

Nous allons utiliser le théorème de Sprague-Grundy pour calculer une stratégie gagnante. On définit le *nombre de Grundy* d'une position par récurrence comme étant le plus petit entier n'étant le nombre de Grundy d'aucun de ses successeurs.
Choisir à chaque étape comme successeur un état dont le nombre de Grundy est 0 est une stratégie gagnante.

☐ 25 – Justifier que le nombre de Grundy existe bien pour chaque position bien que sa définition ne comporte pas de cas de base.

Réponse 25. Les nombres de Grundy des successeurs d'une position forment une partie finie de $\mathbb{N}$. Son complémentaire dans $\mathbb{N}$ est donc infini donc non-vide. Comme $\mathbb{N}$ est bien fondé, cet ensemble admet un élément minimal.

☐ 26 – Étant donné un ensemble X d'entiers. Comparer la valeur du plus petit entier naturel n'appartenant pas à X avec le cardinal de X, $|X|$.

Réponse 26. Dans le pire des cas, $X = \{0; 1; \ldots; |X| - 1\}$, et la plus petite valeur n'appartenant pas à X est $|X|$, sinon, une des valeurs entre 0 et $|X| - 1$ est absente de X. Dans tous les cas, le plus petit entier naturel n'appartenant pas à X est inférieur ou égal à $|X|$.

☐ 27 – Écrire une fonction `int mex(int n, int *tab)` qui prend en entrée un tableau de taille n et qui renvoie le plus petit entier naturel qui n'est pas présent dans le tableau. La complexité de cette fonction doit être $O(n)$.

Réponse 27.

```c
int mex(int n, int *tab){
  bool *present = malloc((n+1) * sizeof(bool));
  for (int i = 0; i <= n; i = i + 1){
    present[i] = false;
  }
  for (int i = 0; i < n; i = i + 1){
    if (tab[i] >= 0 && tab[i] <= n){
      present[tab[i]] = true;
    }
  }
  int res = 0;
  while (present[res]) {
    res = res + 1;
  }
  return res;
}
```

☐ 28 – Écrire une fonction `int grundy(trie t)` qui prend en entrée un trie et renvoie le nombre de grundy de sa racine.

Réponse 28.

```c
int grundy(trie t){
  if (t == NULL) { return 0; }
  bool present[129];
  for (int i = 0; i < 128; i = i + 1){
    if (t->succ[i] != NULL){
      present[grundy(t->succ[i])] = true;
    }
  }
  int res = 0;
  while (present[res]) {
    res = res + 1;
  }
  return res;
}
```

Adapter les tries pour représenter un langage infini

On définit une valeur `l` de type `trie` avec les instructions suivantes :

```c
trie l = eps();
l->sons['a'] = l;
```

Notons que l n'est pas un arbre mais un graphe orienté dont les arêtes et les sommets sont étiquetés, car il possède un cycle.

☐ 29 – Décrire le langage des mots w tels que `is_in(l, w)` renvoie `true`.

Réponse 29. a^*

☐ 30 – Écrire une fonction `trie star(trie t)` qui prend en entrée une valeur t de type `trie` et qui renvoie une nouvelle valeur de type trie telle que si T est le langage des mots w tels que `is_in(t, w)` renvoie `true`, alors `is_in(star(t),w)` renvoie `true` si et seulement si $w \in T^*$.

Réponse 30. Question difficile, séparée ici en 3 fonctions (non testées, à utiliser à vos risques et périls). La complexité est difficile à analyser.

```c
//ajoute les mots de l à t
void add(trie t, trie l){
  for (int i = 0; i < 128; i = i + 1){
    if (t->succ[i] == NULL)
      { t->succ[i] = l->succ[i]; }
    else
      { add(t->succ[i],l->succ[i]); }
  }
}

//ajoute les mots de l à partir de chaque noeud d'étiquette V
void parcours(trie t, trie l){
  if (t == NULL) { return; }
  if (t->lbl)
    { add(t, l); }
  for (int i=0; i < 128; i = i + 1){
    parcours(t->succ[i], l);
  }
}

trie star(trie t){
  trie copie = uni(t, NULL);
  parcours(copie, t);
}
```

On s'intéresse à présent au jeu d'accessibilité à deux joueurs suivant :

Jeu d'accessibilité au langage L :

Initialisation : un mot vide $w = \varepsilon$;

Coups possibles : chacun leur tour, les joueurs ajoutent une lettre au mot w ;

Fin de partie : le jeu s'arête, et le joueur 1 gagne lorsque $w \in L$. Le joueur 2 gagne si la partie est infinie, ou si w n'est préfixe d'aucun mot de L.

Quel que soit le langage L, un des deux joueurs possède une stratégie gagnante.

☐ 31 – Donner deux langages pour lesquels le joueur 1 possède une stratégie gagnante et deux langages pour lesquels le joueur 2 possède une stratégie gagnante.

Réponse 31. Si L contient un mot de 1 lettre, le joueur 1 possède une stratégie gagnante (jouer ce mot).
Si $L = \Sigma^3$, le joueur 1 possède une stratégie gagnante (il gagne nécessairement après avoir jouer pour la deuxième fois).
Si $L = \emptyset$, le joueur 1 ne peut pas gagner et le joueur 2 possède une stratégie gagnante.
Si $L = aaa^*$ et $\Sigma = \{a, b\}$, alors le joueur 2 possède une stratégie gagnante en jouant la lettre b.
Si $L = \{ab; ba\}$, le joueur 2 possède une stratégie gagnante qui consiste à jouer la même lettre que le joueur 1.

☐ 32 – Proposer un algorithme qui à partir d'un langage L permet de déterminer quel joueur possède une stratégie gagnante dans le jeu d'accessibilité à L.

Réponse 32. Étant donné un trie modifié (automate déterministe), élagué et simplifié (cf. questions précédentes) correspondant au langage L, si un joueur peut jouer une lettre qui l'emmène dans un état dont le nombre de Grundy est 0, alors il possède une stratégie gagnante en jouant cette lettre.

Partie IV. Résiduels (OCaml) – 25 pts

On défini en OCaml les langages régulier par induction par le type récursif suivant :

```
type 'a langreg =
  | Empty (* ensemble vide *)
  | Eps (* singleton contenant le mot vide *)
  | Char of 'a (* singleton contenant une lettre *)
  | U of 'a langreg * 'a langreg (* union/disjonction *)
  | Cat of 'a langreg * 'a langreg (* concatenation *)
  | Star of 'a langreg (* etoile de Kleene *)
```

On rappelle que $\emptyset^* = \{\varepsilon\}$.

☐ 33 – Écrire une fonction `has_eps : 'a langreg -> bool` qui prend en entrée un langage régulier `l` et renvoie `true` si `l` contient le mot vide ε, et `false` sinon.

Réponse 33.

```
let rec has_eps = function
  | Empty | Char _ -> false
  | Eps | Star _ -> true
  | U (a,b) -> has_eps a || has_eps b
  | Cat (a,b) -> has_eps a && has_eps b
```

☐ 34 – Simplifier les expressions régulières suivantes :
- Pour la concaténation :
  - εu
  - $u\varepsilon$
  - $\emptyset u$
  - $u\emptyset$
- Pour l'union :
  - $\emptyset|u$
  - $u|\emptyset$
- Pour l'étoile :
  - $(u^*)^*$

Réponse 34.
- Pour la concaténation :
  - $\varepsilon u = u$
  - $u\varepsilon = u$
  - $\emptyset u = \emptyset$
  - $u\emptyset = \emptyset$
- Pour l'union :
  - $\emptyset|u = u$
  - $u|\emptyset = u$
- Pour l'étoile :
  - $(u^*)^* = u^*$

☐ 35 – Écrire une fonction `simplify : 'a langreg -> 'a langreg` qui simplifie au maximum la description d'un langage régulier en utilisant les règles données par la question précédente, ainsi que $\emptyset^* = \{\varepsilon\}$.

Réponse 35.

```
let rec simplify = function
  | Empty -> Empty
  | Eps -> Eps
  | Char c -> Char c
  | U (a,b) -> begin match simplify a, simplify b with
                 | Empty, l | l, Empty -> l
                 | a, b -> U (a,b)
                end
  | Cat (a,b) -> begin match simplify a, simplify b with
                   | Eps, l | l, Eps -> l
                   | Empty, _ | _, Empty -> Empty
                   | a, b -> Cat (a,b)
                  end
  | Star a -> begin match simplify a with
                | Star l -> Star l
                | l -> Star l
               end
```

☐ 36 – Écrire une fonction `string_of_char : char -> string` qui prend en entrée un caractère et renvoie une chaîne de caractère contenant son caractère, éventuellement précédé du caractère \ s'il s'agit d'un des caractères suivants : `(|)*\`.
On pourra utiliser `String.make 1 c` pour créer une chaîne de caractère contenant une fois le caractère `c`.

Réponse 36.

```
let string_of_char c =
  let s = String.make 1 c in
  if c = '(' || c = '|' || c = ')' || c = '*' || c = '\\' then
    "\\"^s
  else s
```

☐ 37 – Écrire une fonction `regex : char langreg -> string` qui prend en entrée un langage régulier et renvoie une expression régulière qui le désigne.

Réponse 37.

```
let rec regex = function
  | Empty -> raise (Invalid_argument "Empty language")
  | Eps -> ""
  | Char c -> string_of_char c
  | Cat (a,b) -> regex a ^ regex b
  | U (a,b) -> "("^regex a^"|"^regex b^")"
  | Star l -> "("^regex l^")*"
```

<u>Définition :</u> Étant donné deux langages X et Y, on appelle résiduel à gauche de Y par rapport à X l'ensemble suivant :

$$X^{-1}Y = \{u \mid \exists x \in X, y \in Y, y = xu\}$$

Pour un mot u, on note $u^{-1}Y = \{u\}^{-1}Y$.

☐ 38 – Expliciter $a^{-1}(ab)^*$.

Réponse 38. $b(ab)^*$

☐ 39 – Expliciter $a^{-1}((\varepsilon|a)(a|b))$.

Réponse 39. $\{\varepsilon; a; b\}$

☐ 40 – Écrire une fonction `residual : 'a -> 'a langreg -> 'a langreg` qui prend en entrée une lettre a et un langage régulier L et renvoie le résiduel à gauche de L par $\{a\}$, $a^{-1}L$.

Réponse 40.

```
let rec residual c = function
  | Char x when x = c -> Eps
  | Empty | Eps | Char _ -> Empty
  | Cat (a,b) -> if has_eps a then
                   U (Cat (residual c a, b), residual c b)
                 else
                   Cat (residual c a, b)
  | U (a,b) -> U(residual c a, residual c b)
  | Star l -> Cat(residual c l, Star l)
```

☐ 41 – En déduire une fonction `residual_word : 'a list -> 'a langreg -> 'a langreg`, qui prend en entrée un mot u sous forme d'une liste de lettres, et un langage régulier L et renvoie le résiduel à gauche de L par $\{u\}$, $u^{-1}L$.

Réponse 41.

```
let rec residual_word w l = match w with
  | [] -> l
  | t::q -> residual_word q (residual t l)
```

☐ 42 – En déduire une fonction `is_in : 'a list -> 'a langreg -> bool`, qui prend en entrée un mot u sous forme d'une liste de lettres, et un langage régulier L et renvoie `true` si $u \in L$ et `false` sinon.

Réponse 42.

```
let is_in w l = has_eps (residual_word w l)
```

☐ 43 – Montrer que pour tout langage régulier L sur un alphabet Σ, $\left\{u^{-1}L \mid u \in \Sigma^*\right\}$ est fini.

Réponse 43. Plus facile en raisonnant sur les automates déterministes (ou trie modifiés en suivant la section précédente). Chaque état (nœud) correspond à un résiduel de L, et tout résiduel par rapport à un mot u correspond à l'état obtenu en suivant le mot u depuis l'état initial (racine).

Fin de l'épreuve