Sélection de sujets posés lors de la session 2023

Exercices de type A

Exercice 1 *Grammaires algébriques (type A)*

On considère la grammaire algébrique G sur l'alphabet $\Sigma = \{a, b\}$ et d'axiome S dont les règles sont :

$$S \to SaS \,|\, b$$

1. Cette grammaire est-elle ambiguë ? Justifier.
2. Déterminer (sans preuve pour cette question) le langage L engendré par G. Quelle est la plus petite classe de langages à laquelle L appartient ?
3. Prouver que $L = L(G)$.
4. Décrire une grammaire qui engendre L de manière non ambiguë en justifiant de cette non ambiguité.
5. Montrer que tout langage dans la même classe de langages que L peut être engendré par une grammaire algébrique non ambiguë.

Proposition de corrigé

1. Cette grammaire est ambiguë car le mot $babab \in L(G)$ admet les deux arbres syntaxiques suivants :

   Or ces derniers sont différents.

2. On constate que $L(G)$ est rationnel car dénoté par l'expression rationnelle $(ba)^\star b$.

3. Montrons par récurrence forte sur $n \in \mathbb{N}^\star$ la propriété $H(n)$ suivante : si $u \in L$ est un mot de taille n alors u est engendré par la grammaire G. C'est bien sûr le cas pour $n = 1$ puisque le seul mot de L de cette taille est b, engendré par la deuxième règle de G.

   Soit donc $n \in \mathbb{N}^\star$ et u un mot de L de taille $n+1$. Comme $|u| \geq 2$, ba est nécessairement préfixe de u et il existe donc $v \in (ba)^\star b = L$ tel que $u = bav$. Par hypothèse, ce mot v est engendré par G : il existe une dérivation telle que $S \Rightarrow^\star v$. On en déduit que

   $$S \Rightarrow SaS \Rightarrow baS \Rightarrow^\star bav = u$$

   est une dérivation licite et donc que $u \in L(G)$. Cette récurrence montre que $L \subset L(G)$.

   Montrons réciproquement que $L(G) \subset L$ en montrons par récurrence forte sur $n \in \mathbb{N}^\star$ la propriété $H(n)$ suivante : si $u \in \Sigma^\star$ se dérive de S en n dérivations alors $u \in L$. C'est acquis pour $n = 1$: le seul mot de $\Sigma^\star$ qu'on peut obtenir en une dérivation est $b \in L$.

Soit donc $n \in \mathbb{N}^\star$ et u un mot dans $L(G)$ tel que $S \Rightarrow^{n+1} u$. Comme ce mot est obtenu en au moins 2 dérivations, les règles de G nous informent que la première est nécessairement $S \to SaS$ (sans quoi ce serait $S \to b$ et dans ce cas u serait obtenu en une seule dérivation). Donc la dérivation permettant d'obtenir u se décompose en :

$$S \Rightarrow SaS \Rightarrow^n u$$

On en déduit qu'il existe $v, w \in \Sigma^\star$ et $k_1, k_2 \in [\![1, n]\!]$ tels que $u = vaw$, $S \Rightarrow^{k_1} v$, $S \Rightarrow^{k_2} w$ et $k1 + k2 = n$. L'hypothèse de récurrence (forte) s'applique à v et w et on en déduit que ces deux mots appartiennent au langage dénoté par $(ba)^\star b$ donc qu'il existe $r_1, r_2 \in \mathbb{N}$ tels que $v = (ba)^{r_1} b$ et $w = (ba)^{r_2} b$. Par conséquent, $u = (ba)^{r_1} ba (ba)^{r_2} b = (ba)^{r_1 + r_2 + 1} b \in L$.

4. On sait à présent que $L(G) = (ba)^\star b$; il s'agit donc de trouver une grammaire non ambiguë engendrant ce langage. On peut proposer par exemple la grammaire dont les règles sont :

$$S \to Tb \qquad T \to baT \,|\, \varepsilon$$

les règles sur T permettant de générer le facteur dans $(ba)^\star$ et la première de rajouter le b final.

Cette grammaire G' est non ambiguë car pour tout mot dans $L(G')$, il existe une unique dérivation permettant de le construire (donc évidemment un seul arbre syntaxique) ; cette unicité découlant du fait que dans cette grammaire un non terminal se dérive toujours en un mot qui contient au plus un seul non terminal.

5. La question demande de montrer que tout langage rationnel peut être engendré par une grammaire non ambiguë. Soit donc L un langage rationnel. Par le théorème de Kleene, il existe un automate fini $A = (\Sigma, Q, \{q_0\}, F, \delta)$ qui reconnaît L qu'on peut loisiblement supposer déterministe.

Considérons alors la grammaire dont les non terminaux sont $\{V_q \,|\, q \in Q\}$, l'axiome est V_{q_0}, les terminaux sont les lettres de Σ et dont les règles sont données par :

- Pour toute transition $q \xrightarrow{a} q'$ dans A, on ajoute la règle $V_q \to aN_{q'}$.
- Pour tout $q \in F$, on ajoute la règle $V_q \to \varepsilon$.

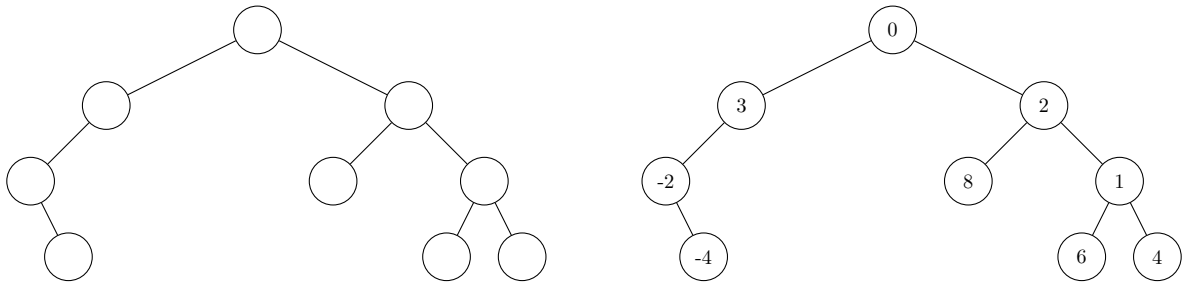
Cette grammaire engendre L de manière non ambiguë grâce au déterminisme de A.

Commentaires du jury

1. Le jury attend du candidat qu'il exhibe un mot montrant l'ambiguïté, en justifiant cette dernière via deux arbres syntaxiques différents ou bien deux dérivations gauches (ou droites) différentes.
2. Il est attendu que le candidat exhibe une expression régulière pour ce langage (qui est donc régulier). Plusieurs sont possibles et toutes sont acceptées. Aucune justification n'est nécessaire à ce stade, sauf si l'expression proposée est démesurément complexe, auquel cas le jury invite le candidat à lui expliquer. Certains candidats n'ont pas compris ce qui était attendu par « la plus petite classe de langages », le jury a alors précisé ses attentes sans pénaliser pour autant ces candidats.
3. On attend une preuve précise et rigoureuse, par exemple par double inclusion. Les explications vagues et les arguments d'évidence ne satisfont pas le jury sur cette question.
4. Comme pour la deuxième question, plusieurs grammaires sont ici possibles. On n'attend pas une preuve rigoureuse de non ambiguïté.
5. Proposer une construction correcte d'une grammaire non ambiguë pour un langage régulier (a priori à partir de l'automate associé) suffit à obtenir tous les points. Le jury n'hésitait pas à aiguiller le candidat si nécessaire pour cette question.

Exercice 2 *Minima locaux dans des arbres (type A)*

Dans cet exercice, on considère des arbres binaires étiquetés par des entiers relatifs deux à deux distincts. Un nœud est un minimum local d'un arbre si son étiquette est plus petite que celle de son éventuel père et celles de ses éventuels fils. Considérons par exemple l'étiquetage (b) de l'arbre binaire non étiqueté (a) :



(a) (b)

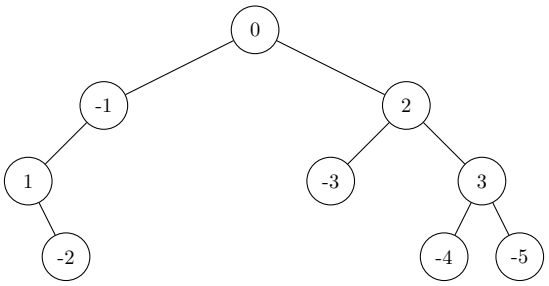
1. Déterminer le ou les minima locaux de l'arbre (b).
2. Donner une définition inductive permettant de définir les arbres binaires ainsi que la définition de la hauteur d'un arbre. Quelle est la hauteur de l'arbre (b) ?
3. Montrer que tout arbre non vide possède un minimum local.
4. Proposer un algorithme permettant de trouver un minimum local d'un arbre non vide et déterminer sa complexité.

On considère un arbre binaire non étiqueté que l'on souhaite étiqueter par des entiers relatifs distincts deux à deux de manière à maximiser le nombre de minima locaux de cet arbre.

5. Proposer sans justifier un étiquetage de l'arbre (a) qui maximise le nombre de minima locaux.
6. Proposer un algorithme qui, étant donné un arbre binaire non étiqueté en entrée, permet de calculer le nombre maximal de minima locaux qu'il est possible d'obtenir pour cet arbre. Déterminer la complexité de votre algorithme.
7. Montrer que, pour un arbre de taille $n \in \mathbb{N}$, le nombre maximal de minima locaux est majoré par $\left\lfloor \dfrac{2n+1}{3} \right\rfloor$. *On pourra remarquer que les nœuds non minimaux couvrent l'ensemble des arêtes de l'arbre.*

Proposition de corrigé

1. Les nœuds d'étiquettes -4, 0 et 1 sont les trois minima locaux.
2. Un arbre est soit un arbre vide soit un nœud formé d'une étiquette et de deux sous-arbres. La hauteur est la profondeur maximale d'une feuille, c'est-à-dire la longueur maximale d'un chemin de la racine à une feuille. La hauteur de l'arbre (b) est 3.
3. L'arbre possède un nombre fini non vide d'étiquettes et donc une étiquette de valeur minimale, qui est un minimum global et donc local.
4. Si la racine de l'arbre est un minimum local on a trouvé notre minimum local. Sinon, un des deux fils est non vide, avec une étiquette à sa racine plus petite que celle de la racine de l'arbre. Un appel récursif permet d'obtenir un minimum local de ce sous-arbre, qui est également un minimum local de l'arbre (que ce soit la racine du sous-arbre ou un descendant strict). La complexité est linéaire en la hauteur de l'arbre.
5. On propose l'étiquetage à la figure (c) dans lequels les 5 minima locaux sont étiquetés par des entiers strictement négatifs.



(c)

6. On propose une approche récursive qui pour un arbre a en entrée calcule $m(a)$ le nombre maximal de nœuds qui peuvent être des minima locaux dans un étiquetage de a, ainsi que, en même temps, la quantité $m_-(a)$ correspondant à cette même quantité mais en supposant de plus que la racine n'est pas un minimum local. Pour un arbre vide, ces deux valeurs valent 0. Pour un arbre a de fils gauche f_g et de fils droit f_d, on peut obtenir par appels récursifs les quantités $m(f_g)$, $m_-(f_g)$, $m(f_d)$ et $m_-(f_d)$. On a alors $m_-(a) = m(f_g) + m(f_d)$ et $m(a) = \max\{m_-(a), 1 + m_-(f_g) + m_-(f_d)\}$. La complexité est linéaire en la taille de l'arbre, chaque nœud est visité exactement une fois avec un nombre d'opérations constant.

7. Le résultat est vrai pour $n = 0$ et on peut donc supposer que $n \geq 1$. Considérons un étiquetage et notons X l'ensemble des nœuds qui sont des minima locaux et Y ceux qui ne le sont pas. On remarque que deux nœuds adjacents ne peuvent pas être tous les deux des minima locaux, puisque toutes les étiquettes sont deux à deux distinctes. Ainsi toute arête de l'arbre est extrémité d'au moins un nœud de Y et l'ensemble Y couvre donc toutes les arêtes. Comme chaque nœud de Y est incident à au plus 3 arêtes et qu'il y a exactement $n-1$ arêtes dans l'arbre, il faut au moins $\frac{n-1}{3}$ nœuds pour couvrir toutes les arêtes, c'est-à-dire $|Y| \geq \frac{n-1}{3}$. On a donc $|X| = n - |Y| \leq n - \frac{n-1}{3} = \frac{2n+1}{3}$ et donc $|X| \leq \left\lfloor \frac{2n+1}{3} \right\rfloor$.

Commentaires du jury

1. On attend simplement du candidat qu'il propose les minima trouvés sans justification. Une réponse orale suffit. En cas d'erreur, le candidat est invité à expliquer son raisonnement.
2. Dans le cas où le candidat propose une définition inductive des arbres avec une feuille pour cas de base, il est guidé vers une définition dont le cas de base est l'arbre vide de sorte à rendre compte du fait que les arbres considérés ne sont pas binaires stricts.
3. Plusieurs stratégies sont ici acceptées (preuve par induction ou existence d'un minimum global qui est local par exemple).
4. Si le candidat propose une solution linéaire en la taille de l'arbre, il est guidé vers une solution linéaire en la hauteur.
5. Proposer un étiquetage correct suffit à obtenir tous les points.
6. Le jury attend une complexité linéaire. Des indications peuvent être apportées pour aiguiller le candidat vers une telle solution.
7. Très peu de candidats ont pu aborder cette question.

Exercice 3 *Formules propositionnelles croissantes (type A)*

On fixe un entier $n \geq 1$ et $E = \{x_1, \cdots, x_n\}$ un ensemble de variables propositionnelles. Étant données deux applications $a : E \to \{V, F\}$ et $b : E \to \{V, F\}$ on dit que a est plus petite que b (que l'on note $a \leq b$) si :

$$\forall x \in E, a(x) = V \implies b(x) = V.$$

Dans un but de simplification des calculs, on pourra faire les abus de notation suivants : assimiler V à 1 et F à 0 et vice versa. Avec cet abus, la propriété $a \leq b$ se traduit par :

$$\forall x \in E, a(x) \leq b(x).$$

1. Étant donnée une valuation sur E, rappeler comment on l'étend naturellement en une valuation sur les formules propositionnelles.

On dit qu'une formule propositionnelle P est *croissante* si pour tout a, b des valuations vérifiant $a \leq b$, on a :

$$a(P) = V \implies b(P) = V.$$

2. Montrer que si P, Q sont des formules croissantes, alors $P \wedge Q$ et $P \vee Q$ sont des formules croissantes.
3. Soit C une clause conjonctive satisfiable contenant au moins un littéral. Montrer qu'elle est croissante si et seulement si elle ne contient aucun littéral de la forme $\neg x$ avec $x \in E$.
4. On considère une formule propositionnelle P qui n'est ni une tautologie, ni une antilogie.
   (a) Montrer que si P est logiquement équivalente à une disjonction de clauses conjonctives dont aucune ne contient un littéral de la forme $\neg x$ avec $x \in E$, alors P est une formule propositionnelle croissante.
   (b) Réciproquement, montrer que si P est une formule propositionnelle croissante, alors elle est logiquement équivalente à une disjonction de clauses conjonctives dont aucune ne contient un littéral de la forme $\neg x$ avec $x \in E$.

Proposition de corrigé

1. On fixe $a : E \to \{V, F\}$ une valuation. On étend a par induction comme suit

$$\text{Si } P \text{ et } Q \text{ sont des formules propositionnelles, alors } \begin{array}{l} a(\neg P) = \text{ Négation de } a(P). \\ a(P \wedge Q) = a(P) \text{ et } a(Q), \\ a(P \vee Q) = a(P) \text{ ou } a(Q). \end{array}$$

2. Soient P, Q deux formules croissantes. Montrons que $P \wedge Q$ et $P \vee Q$ sont croissantes.

   Soient a, b deux valuations vérifiant $a \leq b$. En utilisant l'abus de notation, on a les égalités suivantes :

$$\begin{array}{ll} a(P \wedge Q) = \min(a(P), a(Q)), & a(P \vee Q) = \max(a(P), a(Q)), \\ b(P \wedge Q) = \min(b(P), b(Q)), & b(P \vee Q) = \max(b(P), b(Q)). \end{array}.$$

   Sachant que $a \leq b$, on a donc $a(P) \leq b(P)$ et $a(Q) \leq b(Q)$. On en déduit :

$$\min(a(P), a(Q)) \leq \min(b(P), b(Q)) \text{ et } \max(a(P), a(Q)) \leq \max(b(P), b(Q))$$

   Autrement dit $a(P \wedge Q) \leq b(P \wedge Q)$ et $a(P \vee Q) \leq b(P \vee Q)$.

   $P \wedge Q$ et $P \vee Q$ sont bien croissantes.

3. On montre par double implication.

- (Sens réciproque). Soit C une clause conjonctive satisfiable contenant au moins un littéral. Un clause contenant un unique littéral positif est clairement croissant. Par récurrence et par stabilité des formules croissantes par l'opération $\wedge$, on en déduit que tout clause conjonctive contenant uniquement des littéraux positifs est croissante.
- (Sens direct, par la contraposée). Considérons une clause conjonctive C satisfiable contenant au moins un littéral. Montrons que si celle-ci est croissante, alors tout littéral apparaissant dans C est positif.

  Supposons que C contient au moins un littéral négatif et quitte à réarranger les termes, on considère que $C = \neg x \wedge C'$ avec C' une clause conjonctive ne contenant pas x (possible car C est satisfiable).

  Considérons une valuation a vérifiant $a(C) = V$. On a alors $a(\neg x) = V$ et $a(C') = V$. Donc $a(x) = F$.

  On choisit b qui coïncide avec a sur toutes les variables propositionnelles exceptée sur x où on a $b(x) = V$. Par construction, $a \leq b$. Mais $b(C) = F$. Donc C n'est pas une formule croissante.

On a bien l'équivalence demandée.

4. (a) Si P est logiquement équivalente à une disjonction de clauses conjonctives sans littéral négatif, par stabilité de la croissance par disjonction et les clauses conjonctives sans littéral négatif étant croissante, on en déduit que P est croissante.

   (b) Récriproquement, soit P une formule croissante. On pose :

$$P' = \bigvee_{C \vDash P} C$$

   où pour deux formules P, Q, on a $P \vDash Q$ si pour toute valuation a , $a(P) = V \implies a(Q) = V$.

   Dans notre cas, C désigne une clause conjonctive croissante inférieure à P (Autrement dit, pour toute valuation a : $a(C) \leq a(P)$).

   Vérifions que P' est logiquement équivalente à P.

   - soit a une valuation vérifiant $a(P') = V$.Il existe $C \leq P$ une clause telle que $a(C) = V$. On a bien $a(P) = V$.
   - Réciproquement, si a est une valuation vérifiant $a(P) = V$, on pose :

$$C = \bigwedge_{x \text{ positif } a(x)=V} x$$

     . Remarquons que si C était indexé par l'ensemble vide, a enverrait tous les littéraux sur F et dans ce cas par croissance, P serait une tautologie ce qui contredit l'hypothèse de l'énoncé. Les propriétés suivantes sont alors vérifiées :

     – C n'est pas indexé par le vide (sinon, a enverrait tous les littéraux positifs sur F et par croissance, P serait une tautologie)

     – C contient uniquement des littéraux positifs,

     – on a $C \vDash P$ par croissance de P.

     Donc C apparaît dans P'. D'où $a(P') = V$.

   Ainsi, P et P' sont bien logiquement équivalentes.

Commentaires du jury

1. Le jury profite de cette question pour vérifier que le candidat sait faire la différence entre syntaxe et sémantique.
2. Le candidat est autorisé à utiliser l'abus introduit par l'énoncé (assimiler V à 1 et F à 0) mais le jury se réserve le droit de vérifier que le candidat a bien compris en quoi c'est un abus. Un candidat qui prouve rigoureusement la propriété souhaitée pour un des connecteurs et indique ce qu'il suffirait d'y changer pour l'autre obtient tous les points.
3. Il ne faut pas oublier de traiter le caractère nécessaire et suffisant.
4. Le jury peut donner une indication pour aider le candidat à trouver une formule permettant de répondre à cette question.

Exercice 4 *Activation de processus (type A)*

Soit un système temps réel à n processus asynchrones $i \in [\![1, n]\!]$ et m ressources r_j. Quand un processus i est actif, il bloque un certain nombre de ressources listées dans un ensemble P_i et une ressource ne peut être utilisée que par un seul processus. On cherche à activer simultanément le plus de processus possible.

Le problème de décision `ACTIVATION` correspondant ajoute un entier k aux entrées et cherche à répondre à la question : "Est-il possible d'activer au moins k processus en même temps ?"

1. Soit $n = 4$ et $m = 5$. On suppose que $P_1 = \{r_1, r_2\}$, $P_2 = \{r_1, r_3\}$, $P_3 = \{r_2, r_4, r_5\}$ et $P_4 = \{r_1, r_2, r_4\}$. Est-il possible d'activer 2 processus en même temps ? Même question avec 3 processus.
2. Dans le cas où chaque processus n'utilise qu'une seule ressource, proposer un algorithme résolvant le problème `ACTIVATION`. Évaluer la complexité de votre algorithme.

On souhaite montrer que `ACTIVATION` est NP-complet.

3. Donner un certificat pour ce problème.
4. Écrire en pseudo code un algorithme de vérification polynomial. On supposera disposer de trois primitives, toutes trois de complexité polynomiale :
   (a) `appartient(c,i)` qui renvoie `Vrai` si le processus `i` est dans l'ensemble d'entiers `c`.
   (b) `intersecte(`$\mathtt{P}_i$`,R)` qui renvoie `Vrai` si le processus `i` utilise une ressource incluse dans un ensemble de ressources `R`.
   (c) `ajoute(`$\mathtt{P}_i$`,R)` qui ajoute les ressources $\mathtt{P}_i$ dans l'ensemble `R` et renvoie ce nouvel ensemble.

En théorie des graphes, le problème `STABLE` se pose la question de l'existence dans un graphe non orienté $G = (S, A)$ d'un ensemble d'au moins k sommets ne contenant aucune paire de sommets voisins. En d'autres termes, existe-t-il $S' \subset S$, $|S'| \geq k$ tel que $s, p \in S' \Rightarrow (s, p) \notin A$?

5. En utilisant le fait que `STABLE` est NP-complet, montrer par réduction que le problème `ACTIVATION` est également NP-complet.

Commentaires du jury

1. On attend du candidat une réponse en oui / non avec une précision de quels processus activer dans le cas où la réponse est oui et une très rapide justification dans le cas où la réponse est non. Une réponse orale suffit.
2. De multiples réponses sont possibles sur cette question et sont acceptées du moment qu'elles sont clairement décrites et correctement analysées. On n'attend pas une complexité optimale.
3. Une courte phrase décrivant la nature d'un tel certificat et indiquant sa polynomialité en la taille de l'entrée suffit à obtenir tous les points.
4. On attend un algorithme utilisant à bon escient les primitives proposées par le sujet.
5. Le jury est attentif au fait que tous les arguments soient bien présents : description de la réduction, preuve que c'en est une et justification de son caractère polynomial pour le caractère NP-difficile ; caractère NP pour pouvoir conclure quant à la NP-complétude.

Exercices de type B

Exercice 5 *Langages locaux (type B)*

Consignes : Cet énoncé est accompagné d'un code compagnon en OCaml localite.ml *fournissant le type décrit par l'énoncé et quelques fonctions auxiliaires : il est à compléter en y implémentant les fonctions demandées. On privilégiera un style de programmation fonctionnel.*

On considère un alphabet Σ. Si L est un langage sur Σ, on note :

- $P(L) = \{a \in \Sigma \mid a\Sigma^\star \cap L \neq \emptyset\}$ l'ensemble des premières lettres des mots de L.
- $D(L) = \{a \in \Sigma \mid \Sigma^\star a \cap L \neq \emptyset\}$ l'ensemble des dernières lettres des mots de L.
- $F(L) = \{m \in \Sigma^2 \mid \Sigma^\star m \Sigma^\star \cap L \neq \emptyset\}$ l'ensemble des facteurs de longueur 2 des mots de L.
- $N(L) = \Sigma^2 \setminus F(L)$ l'ensemble des mots de taille 2 qui ne sont pas facteurs de mots de L.

On rappelle qu'un langage L est dit *local* si et seulement si l'égalité de langages suivantes est vérifiée :

$$L \setminus \{\varepsilon\} = (P(L)\Sigma^\star \cap \Sigma^\star D(L)) \setminus (\Sigma^\star N(L) \Sigma^\star)$$

1. Calculer les ensembles $P(L), D(L), F(L)$ et $N(L)$ dans le cas où L est le langage dénoté par l'expression régulière $a^\star(ab)^\star + aa$ sur l'alphabet $\{a, b\}$. Ce langage est-il local ? On vérifiera la cohérence entre les réponses à cette question et celles obtenues via les fonctions demandées dans la suite de l'énoncé.

On cherche dans la suite de l'exercice à concevoir un algorithme répondant à la spécification suivante :

Entrée : Une expression régulière e sur un alphabet Σ ne faisant pas intervenir le symbole $\emptyset$.
Sortie : Vrai si le langage dénoté par e est local, faux sinon.

Par défaut, dans la suite de l'énoncé, "expression régulière" signifie "expression régulière ne faisant pas intervenir le symbole $\emptyset$". Les expressions régulières seront manipulées en OCaml via le type somme suivant :

```
type regexp =
  | Epsilon
  | Letter of string (*La chaîne en question sera toujours de longueur 1*)
  | Union of regexp * regexp
  | Concat of regexp * regexp
  | Star of regexp
```

On propose tout d'abord de calculer les ensembles $P(L)$, $D(L)$ et $F(L)$ à partir d'une expression régulière dénotant L. Ces ensembles seront représentés en OCaml par des listes de chaînes de caractères qui vérifieront les propriétés suivantes :

- Elles sont triées dans l'ordre croissant selon l'ordre lexicographique.
- Elles sont sans doublons.

L'énoncé fournit une fonction `union` permettant de calculer l'union sans doublons de deux listes triées. La définition inductive d'une expression régulière invite à calculer inductivement les ensembles $P(L)$, $D(L)$ et $F(L)$. C'est ce que propose la fonction `compute_P` fournie par l'énoncé.

2. En supposant que la fonction `contains_epsilon : regexp -> bool` renvoie `true` si et seulement si le langage dénoté par l'expression régulière en entrée contient le mot ε, justifier brièvement la correction de `compute_P`.
3. Implémenter la fonction `contains_epsilon`.

4. Sur le modèle de `compute_P`, implémenter une fonction `compute_D : regexp -> string list` permettant le calcul de l'ensemble $D(L)$ étant donnée une expression régulière dénotant le langage L.
5. Expliquer en langage naturel comment calculer récursivement l'ensemble $F(L)$ étant donnée une expression régulière e dénotant le langage L. Si $e = e_1e_2$ on pourra exprimer $F(L)$ en fonction notamment de $P(L_2)$ et $D(L_1)$ où L_1 (resp. L_2) est le langage dénoté par e_1 (resp. e_2).
6. Écrire une fonction `prod : string list -> string list -> string list` calculant le produit cartésien des deux listes en entrée, qu'on pourra supposer triées dans l'ordre lexicographique croissant et sans doublons, puis qui pour chaque couple de chaînes dans la liste obtenue les concatène. Par exemple :

```
prod ["a";"c";"e"] ["b";"c"] = ["ab";"ac";"cb";"cc";"eb";"ec"]
```

7. En déduire une fonction `compute_F : regexp -> string list` déterminant l'ensemble $F(L)$ étant donnée une expression régulière dénontant le langage L.

Dans les questions qui suivent, on ne demande PAS d'implémenter les algorithmes décrits.

8. Décrire en pseudo-code ou en langage naturel un algorithme permettant de calculer un automate reconnaissant $(P(L)\Sigma^\star \cap \Sigma^\star D(L)) \setminus (\Sigma^\star N(L)\Sigma^\star)$ étant donnée une expression régulière dénotant L.
9. Décrire un algorithme permettant de détecter si le langage dénoté par une expression régulière est local ou non.
10. Pourquoi est-il légitime de ne considérer que les expressions régulières ne faisant pas intervenir $\emptyset$? Comment modifier l'algorithme obtenu dans le cas où cette contrainte n'est plus vérifiée ?

Proposition de corrigé

1. On obtient $P(L) = \{a\}$, $D(L) = \{a, b\}$, $F(L) = \{aa, ab, ba\}$ et donc $N(L) = \{bb\}$. Le langage L n'est pas local : s'il l'était, il devrait contenir le mot aba ce qui n'est pas.
2. La seule difficulté est de justifier la disjonction de cas pour les concaténations. Si $e = e_1e_2$ et $\varepsilon \notin L(e_1)$, une première lettre d'un mot de e est nécessairement une première lettre d'un mot de e_1 (la réciproque étant évidente). Sinon, un mot de $L(e)$ peut aussi commencer de la même façon qu'un mot de $L(e_2)$.
3. On filtre sans grande difficulté.

```
let rec contains_epsilon (e:regexp) :bool = match e with
  |Epsilon -> true
  |Letter _ -> false
  |Union (e1, e2) -> (contains_epsilon e1) || (contains_epsilon e2)
  |Concat (e1, e2) -> (contains_epsilon e1) && (contains_epsilon e2)
  |Star _ -> true
```

4. Comme pour `compute_P`, la seule difficulté est le cas d'une concaténation.

```
let rec compute_D (e:regexp) :string list  = match e with
  |Epsilon -> []
  |Letter a -> [a]
  |Union (e1, e2) -> union (compute_D e1) (compute_D e2)
  |Concat (e1, e2) when contains_epsilon e2 ->
     union (compute_D e1) (compute_D e2)
  |Star e1 |Concat (_, e1) -> (compute_D e1)
```

5. Soit e une expression régulière et L le langage qu'elle dénote. On calcule $F(L)$ inductivement :
   - Si $e = \varepsilon$ ou $e = a$ avec $a \in \Sigma$, $F(L) = \emptyset$.
   - Si $e = e_1 + e_2$, $F(L) = F(L_1) \cup F(L_2)$ où L_i est le langage dénoté par e_i.
   - Si $e = e_1 e_2$, un facteur de taille 2 d'un mot de L est : soit un facteur de taille 2 de L_1, soit un facteur de taille 2 de L_2, soit un facteur "à cheval entre L_1 et L_2" c'est-à-dire dont la première lettre est la fin d'un mot de L_1 et la deuxième est le début d'un mot de L_2. Autrement dit :
   $$F(L) = F(L_1) \cup F(L_2) \cup D(L_1)P(L_2)$$
   - Si $e = e_1^\star$, on obtient similairement au cas précédent $F(L) = F(L_1) \cup D(L_1)P(L_1)$.

6. On peut remplacer le `List.map` par une petite fonction auxiliaire au besoin.

```
let rec prod (l1:string list) (l2:string list) = match l1, l2 with
  |[], _ |_, [] -> []
  |t::q, l -> union (List.map (fun x -> t^x) l) (prod q l)
```

7. C'est une traduction de la question 5.

```
let rec compute_F (e:regexp) :string list = match e with
  |Epsilon |Letter _ -> []
  |Union (e1, e2) -> union (compute_F e1) (compute_F e2)
  |Concat (e1, e2) ->
     let d_e1 = compute_D e1 and p_e2 = compute_P e2 in
     union (union (compute_F e1) (compute_F e2)) (prod d_e1 p_e2)
  |Star e1 ->
     let d_e1 = compute_D e1 and p_e1 = compute_P e1 in
     union (compute_F e1) (prod d_e1 p_e1)
```

8. On remarque qu'une fois calculé $F(L)$ on en déduit immédiatement $N(L)$ en calculant tous les facteurs de taille 2 de Σ et en éliminant ceux qui sont dans $F(L)$.

   Il est ensuite très facile de concevoir un automate pour chacun des langages $P(L)\Sigma^\star$, $\Sigma^\star D(L)$ et $\Sigma^\star N(L)\Sigma^\star$; les ensembles $P(L)$, $D(L)$ et $N(L)$ étant finis. De plus, on sait algorithmiquement construire l'intersection (automate produit) et le complémentaire (déterminisation puis échange des états finaux et non finaux) d'automates donc on peut construire un automate reconnaissant
   $$P(L)\Sigma^\star \cap \Sigma^\star D(L) \cap (\Sigma^\star N(L)\Sigma^\star)^c = (P(L)\Sigma^\star \cap \Sigma^\star D(L)) \setminus (\Sigma^\star N(L)\Sigma^\star)$$

9. On propose la méthode suivante :
   - Calculer un automate A reconnaissant $(P(L)\Sigma^\star \cap \Sigma^\star D(L)) \setminus (\Sigma^\star N(L)\Sigma^\star)$ via la question 8.
   - Calculer un automate B reconnaissant $L \setminus \{\varepsilon\}$ à partir de l'expression régulière dénotant L, par exemple à l'aide de l'algorithme de Berry-Sethi.
   - Calculer l'automate $A\Delta B$ reconnaissant la différence symétrique de ces deux langages (la différence symétrique se construisant à partir d'unions, intersections et complémentaires, opérations qu'on sait faire sur des automates finis).
   - A l'aide d'un parcours, déterminer si un état final de $A\Delta B$ est accessible depuis un de ses états initiaux. Si non, le langage reconnu par $A\Delta B$ est vide donc les langages reconnus par A et B sont égaux donc L est local. Si oui, le langage reconnu par $A\Delta B$ n'est pas vide donc les langages reconnus par A et B son différents et L n'est pas local.

10. Si e est un expression régulière telle que $L(e) \neq \emptyset$, il existe une expression régulière e' ne contenant pas le symbole $\emptyset$ telle que $L(e') = L(e)$. Cela se montre aisément par induction sur e.

    Si on autorise les expressions régulières contenant $\emptyset$, et qu'on souhaite à partir de e déterminer si $L(e)$ est local, on pourrait donc :

    - Déterminer si $L(e)$ est vide à l'aide d'un automate reconnaissant ce langage.
    - S'il est vide, il est local. Sinon, on peut inductivement transformer e en e' telle que $L(e) = L(e')$ et e' ne contient pas le symbole $\emptyset$ et on applique l'algorithme de la question 9 à e'.

    Un code source localite_corrige.ml est aussi disponible.

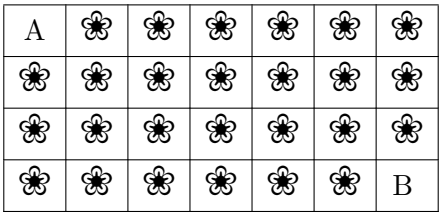
Commentaires du jury

1. On attend du candidat qu'il donne les ensembles demandés sans justification, possiblement à l'oral uniquement puis qu'il exhibe un mot montrant la non localité avec une brève justification.
2. Le jury attend principalement deux arguments : le fait que l'on raisonne par induction et la justification de la disjonction de cas dans le cas d'une concaténation.
3.
4.
5. Le jury attend un argument inductif et que le candidat précise les cas délicats (étoile et concaténation). Cette question peut être abordée en même temps que la question 7.
6. Le jury souhaite une fonction récursive et la demande si cette approche n'est pas proposée. Le candidat est libre de proposer une solution faisant intervenir les fonctions du module `List` ou de s'aider d'une fonction auxiliaire.
7.
8. Une description haut niveau de la construction de cet automate (exploitant par exemple les stabilités des langages rationnels) suffit.
9. Le jury s'attend à ce que l'égalité de ces deux langages soit vérifiée par le biais d'automates les reconnaissant, à nouveau par une description de haut niveau.
10. Là encore une justification haut niveau du fait qu'on peut se ramener à une expression régulière ne contenant pas le symbole vide suffit (on ne demande par exemple pas une preuve par induction de ce fait, simplement le fait qu'on pourrait le faire).

Exercice 6 *Récolte dynamique de fleurs (type B)*

Consignes : Cet énoncé est accompagné d'un code compagnon en C bouquet_enonce.c *fournissant certaines des fonctions mentionnées dans l'énoncé : il est à compléter en y implémentant les fonctions demandées. La ligne de compilation* gcc -o main.exe *.c -lm *vous permet de créer un exécutable* main.exe *à partir du ou des fichiers C fournis. Vous pouvez également utiliser l'utilitaire* make. *En ligne de commande, il suffit de taper* make. *Dans les deux cas, si la compilation réussit, le programme peut être exécuté avec la commande* ./main.exe. *Si vous désirez forcer la compilation de tous les fichiers, vous pouvez au préalable nettoyer le répertoire en faisant* make clean *et relancer* make.

Une petite fille se trouve en haut à gauche (case A) d'un champ modélisé par un tableau rectangulaire de taille $m \times n$ et doit se rendre dans la case B en bas à droite du champ où réside sa grand-mère (figure ci-dessous).



Chaque case du tableau, *y compris les cases A et B*, contient un certain nombre de fleurs. La petite fille, qui connaît depuis sa position initiale le nombre de fleurs de chaque case, doit se déplacer vers B de case en case, les seuls mouvements autorisés étant vers le bas ou vers la droite. À chaque déplacement, elle récolte les fleurs de la case atteinte. L'objectif pour elle est alors de faire le bouquet avec le plus de fleurs possible lors de son déplacement pour l'offrir à sa grand-mère.

1. On considère le champ suivant :

0 (A)	1	2	3
1	2	3	4
2	3	4	0
3	4	0	1 (B)

   Donner le nombre maximal de fleurs cueillies par la petite fille.

2. On note $n(i,j)$ le nombre maximum de fleurs que la petite fille peut récolter en se déplaçant de A à la case (i,j). Exprimer $n(i,j)$ en fonction de $n(i-1,j)$ et $n(i,j-1)$. En déduire une fonction récursive de prototype `int recolte(int champ[m][n], int i, int j)` qui, étant données les coordonnées `i`, `j` d'une case, calcule le nombre maximum de fleurs cueillies par la petite fille de A à la case (`i`, `j`).

3. On suppose $m = n = 4$ et on effectue donc un appel à `recolte(champ,3,3)` pour résoudre le problème posé. Donner le nombre de fois où votre fonction calcule le nombre de fleurs maximum cueillies dans la case $(1,1)$ (deuxième case de la diagonale).

D'une manière générale, le nombre d'appels à la fonction récursive est important. On a donc intérêt à transformer l'algorithme récursif en algorithme dynamique. On propose de déclarer dans le programme principal un tableau `fleurs` dont la case (i,j) est destinée à contenir la récolte maximale que la petite fille peut obtenir en cheminant de A vers la case (i,j).

4. Dans quel ordre remplir le tableau `fleurs` de sorte à éviter de recalculer une valeur ?

5. Écrire une fonction de prototype `int recolte_iterative(int champ[m][n], int i, int j, int fleurs[m][n])` qui calcule, stocke dans `fleurs[i][j]` et retourne la cueillette maximale obtenue en parcourant le champ de A à la case (`i`, `j`).

La fonction `recolte_iterative` permet de déterminer la cueillette maximale en (`i`, `j`) mais ne précise pas le chemin parcouru pour l'obtenir.

6. Écrire la fonction de prototype `void deplacements(int fleurs[m][n], int i, int j)` qui affiche la suite des déplacements effectués par la petite fille sur un chemin permettant de récolter le nombre maximum de fleurs entre (0,0) et (`i`, `j`).
7. Insérer un appel de `deplacements` dans la fonction `recolte_iterative` pour afficher le chemin parcouru.

Proposition de corrigé

1. On trouve 11 fleurs.
2. La récolte en une case dépend récursivement de celle de la case du haut et de celle de la case de gauche. La formule générique est donc à adapter lorsqu'on se trouve sur le bord gauche ou le bord haut du champ.

```
int recolte(int champ[m][n], int i, int j){
  /* Cas de base */
  if ( (i == 0) && (j == 0) )
   return champ[0][0];
  if (i == 0)
    return champ[0][j] + recolte(champ,0, j - 1);
  if (j == 0)
    return champ[i][0] + recolte(champ,i - 1, 0
  /* Cas général */
  return champ[i][j] + max(recolte(champ,i - 1, j),recolte(champ,i, j - 1));
}
```

3. On trouve 6 appels à `recolte`.
4. On calcule d'abord les valeurs des cases sur les bords haut et gauche puis on propage soit en remplissant les lignes de gauche à droite, soit en remplissant les colonnes de haut en bas.
5. Il s'agit d'une traduction des questions 2 et 4. Le code ci-dessous corrige aussi la question 7.

```
int recolte_iterative(int champ[m][n], int i, int j,int fleurs[m][n]){
  int x, y;
  fleurs[0][0] = champ[0][0];
  /* Bord haut */
  for (x = 1; x <= i; x++) {
    fleurs[x][0] = champ[x][0] + fleurs[x - 1][0];
  }
  /* Bord gauche */
  for (y = 1; y <= j; y++) {
    fleurs[0][y] = champ[0][y] + fleurs[0][y - 1];
  }
  /* Autres cases */
  for (y = 1; y <= j; y++) {
    for (x = 1; x <= i; x++) {
      fleurs[x][y] = champ[x][y] + max(fleurs[x - 1][y], fleurs[x][y - 1]);
    }
  }

  deplacements(fleurs,i, j);
  return fleurs[i][j];
}
```

6. On commence par distinguer les cas limites. Puis on appelle récursivement `deplacements` en observant quelle est la case voisine qui avait permi d'obtenir le plus de fleurs.

```
void deplacements(int fleurs[m][n], int i, int j){
  if (i == 0 && j == 0) {
       printf("Case A, ");
       return;
  }
  if (i == 0) {
       deplacements(fleurs,0, j - 1);
       printf("Aller à droite, ");
       return;
  }
  if (j == 0) {
       deplacements(fleurs,i - 1, 0);
       printf("Descendre, ");
       return;
  }
  if (fleurs[i - 1][j] > fleurs[i][j - 1]) {
     deplacements(fleurs,i - 1, j);
     printf("Descendre, ");
  }
  else {
     deplacements(fleurs,i, j - 1);
     printf("Aller à droite, ");
  }
}
```

7. Voir le code proposé à la question 5.

Un code source bouquet_corrige.c est aussi disponible.

Commentaires du jury

1. On attend une réponse orale ; si elle est correcte, le jury ne demande même pas de justification.
2. Il est attendu des candidats qu'ils soient attentifs aux cas de base. Ils peuvent factoriser la présentation de leur code et l'établissement de la relation de récurrence.
3. Plusieurs méthodes sont acceptées pour répondre à cette question ; le candidat peut par exemple compter le nombre d'appels en modifiant la fonction précédente ou à la main en déployant l'arbre d'appels.
4. Une réponse orale claire peut suffire. Le candidat peut aussi s'aider d'un schéma indiquant les dépendances entre les différents termes à calculer.
5.
6. On attend bien l'affichage des déplacements, et pas uniquement leur calcul. La mise en forme de l'affichage (avec sauts de lignes, tabulations, ...) est en revanche libre.
7. Le jury est attentif à l'endroit où l'appel à cette fonction est effectué.

Exercice 7 *Chemins simples sans issue (type B)*

Consignes : Cet exercice est à traiter en OCaml. Le fichier chemins_simples.ml *est fourni avec ce sujet. Il est à compléter en y implémentant les fonctions demandées.*

L'objectif de cet exercice est de programmer une fonction générant la liste des chemins simples sans issue d'un graphe. On rappelle les définitions d'un graphe, d'un chemin, et on donne leur représentation en OCaml.

Un *graphe orienté* est un couple (V, E) où V est un ensemble fini (ensemble des sommets), E est un sous-ensemble de $V \times V$ où tout élément $(v_1, v_2) \in E$ vérifie $v_1 \neq v_2$ (ensemble des arcs).

Étant donné un graphe $G = (V, E)$ un *chemin non vide* de G est une suite finie $s_0, \cdots, s_n$ de sommets de V avec $n \geq 0$ et vérifiant $\forall i \in \{0, \cdots, n-1\}, (s_i, s_{i+1}) \in E$. On dit que ce chemin est *simple* si $s_0, \cdots, s_n$ sont distincts deux à deux. On dit qu'il est *sans issue* si pour tout s_{n+1} sommet tel que $(s_n, s_{n+1}) \in E$, s_{n+1} appartient à $\{s_0, \cdots, s_n\}$.

Dans la suite, les graphes considérés sont définis sur un ensemble de sommets de la forme $\{0, 1, \cdots, n-1\}$. Pour représenter un graphe en OCaml, on utilise le type suivant :

```
type graphe = int list array
```

qui correspond à un encodage par un tableau de listes d'adjacence. Par exemple, le graphe

$$G_1 = (\{0,1,2,3\}, \{(0,1),(0,3),(2,0),(2,1),(2,3),(3,1)\})$$

est représenté par le tableau `[|[1;3];[];[0;1;3];[1]|]`. L'ordre dans lequel sont écrits les éléments dans les listes importe peu. Par contre, l'emplacement des listes dans le tableau est important. Par exemple, `[|[];[0];[0;3;1];[1]|]` représente le graphe

$$G_2 = (\{0,1,2,3\}, \{(1,0),(2,0),(2,1),(2,3),(3,1)\}\})$$

On rappelle différentes fonctions pouvant être utiles :

- `List.filter : ('a -> bool) -> 'a list -> 'a list` où l'expression `List.filter f l` est la liste obtenue en gardant uniquement les éléments `x` de `l` vérifiant `f`.
- `List.iter :('a -> unit) -> 'a list -> unit` où `List.iter f l` correspond à `(f a0);(f a1);...;(f an)` dans le cas où on a `l = a0::a1::...::an::[]`.
- `List.rev : 'a list -> 'a list` est une fonction qui renvoie le retourné d'une liste. Par exemple, `List.rev [3;1;2;2;4]` est égal à `[4;2;2;1;3]`.
- `Array.length : 'a array -> int` est une fonction qui renvoie la longueur d'un tableau.

Les questions de programmation sont à traiter dans le fichier chemins_simples.ml. L'utilisation d'autres fonctions de la bibliothèque que celles mentionnées sont à reprogrammer.

1. Écrire une fonction `est_sommet : graphe -> int -> bool` où `est_sommet g a` est égal à `true` si `a` est un sommet du graphe `g` et `false` sinon.
2. Écrire une fonction `appartient : 'a list -> 'a -> bool` où `appartient l x` est égal à `true` si `x` est un élément de `l` et `false` sinon.
3. Écrire une fonction `est_chemin : graphe -> int list -> bool`, où `est_chemin g l` est égal à `true` si `l` est un chemin de `g` et `false` sinon. On suppose que la liste vide représente le chemin vide, qui est bien un chemin et que les éléments de `l` sont bien des sommets du graphe `g`.
4. Compléter la fonction `est_chemin_simple_sans_issue : graphe -> int list -> bool`, où `est_chemin g l` est égal à `true` si `l` est un chemin simple sans issue de `g` et `false` sinon. On supposera que les éléments de `l` sont des sommets du graphe `g` et que le chemin vide n'est pas simple sans issue.

5. On cherche à écrire une fonction qui construit la liste des chemins simples sans issue d'un graphe. Pour cela, on procède à l'aide de parcours en profondeur et d'un algorithme de retour sur trace. Compléter le code de la fonction `genere_chemins_simples_sans_issue` présent dans le fichier chemins_simples.ml et qui permet de générer la liste des chemins simples sans issue d'un graphe.
6. Écrire des expressions donnant les listes des chemins simples pour les deux graphes G_1 et G_2.
7. Expliciter la complexité des fonctions `appartient` et `est_chemin_simple_sans_issue`.

Proposition de corrigé

1. Voici le code demandé :
```
let est_sommet g a =  (0 <=a) && (a < Array.length g)
```
2. Une proposition de code :
```
let rec appartient liste a = match liste with
    | [] -> false
    | b::suite -> (b = a) || (appartient suite a)
```
3. Une proposition de code :
```
let rec est_chemin g liste = match liste with
   | [] -> true
   | [a] -> est_sommet g a
   | a::b::suite -> (appartient (g.(a)) b) && (est_chemin g (b::suite))
```
4. Voici le code demandé :
```
let est_chemin_simple_sans_issue g liste =
   let n = Array.length g in
   let visites = Array.make n false in
   let rec test_aux liste = match liste with
       | [] -> false
       | [a] -> [] = (List.filter (fun x -> not visites.(x)) (g.(a)))
       | a::b::suite ->
           begin
               visites.(a) <- true ;
               (appartient g.(a) b) && (not visites.(b))
               && (test_aux (b::suite))
           end
       in
       test_aux liste
```
5. Voici un exemple de solution :
```
let genere_chemins_simples_sans_issue (g:graphe) =
  let taille = Array.length g in
  (*garde en mémoire les chemins déjà trouvés*)
  let liste_chemins = ref [] in
  (*garde en mémoire les sommets en cours de visite *)
  let visites = Array.make taille false in
  (*garde en mémoire le début d'un chemin*)
  let chemin_courant_envers = ref [] in
```

```
(*trouve tous les chemins simples sans issue commençant par s*)
let rec profondeur s =
  if not visites.(s) then begin
    visites.(s) <- true ;
    chemin_courant_envers := s::(!chemin_courant_envers) ;
    let voisins_libres =
      List.filter (fun x -> not visites.(x)) g.(s)
    in
    if voisins_libres = [] then begin
      liste_chemins := (List.rev !chemin_courant_envers)::(!liste_chemins)
    end else begin
      List.iter profondeur voisins_libres
    end;
    (*pour revenir en arrière *)
    visites.(s) <- false ;
    chemin_courant_envers := List.tl !chemin_courant_envers ;
  end
in
for i = 0 to (taille-1) do
  profondeur i
done ;
!liste_chemins
```

6. Voici un exemple de solution :

```
let liste1 = genere_chemins_simples_sans_issue g1
let liste2 = genere_chemins_simples_sans_issue g2
```

7. La complexité de `appartient` est en $O(|l|)$: en effet, on effectue un parcours de liste.

   On note A l'ensemble des arêtes du graphe en argument. La complexité de `est_chemin_sans_issue` est en $O(|l| + |A|)$. En effet, pour chaque sommet s de la liste l, on effectue en terme de calcul de l'ordre de $1 + |g.(s)|$. En sommant sur tous les éléments de l (dans le cas où il serait tous distincts), on trouve une complexité en $O(|l|+|A|)$. Si un même sommet apparaît deux fois, le calcul est interrompu lors de la visite d'une première répétition et on retrouve la même complexité.

Un code source chemins_simples_corrige.ml est également disponible.

Commentaires du jury

1. Le candidat ne doit pas oublier de tester la positivité de l'entier en entrée.
2. Le jury s'attend à une solution récursive. Dans cet exercice une approche impérative n'est néanmoins pas pénalisée.
3. Même remarque que pour la question précédente.
4. Plusieurs variantes de réponses à cette question sont possibles.
5. Le jury s'attend à ce que les ajouts dans une liste soient faits en tête, quitte à renverser la liste à la fin.
6. Le jury souhaite voir le résultat des tests demandés dans cette question : les candidats qui souhaitent compiler leur code doivent donc coder des fonctions d'affichage pertinentes pour obtenir tous les points ici.
7. Le jury attend une borne précise sur ces complexités. Si le candidat propose une borne en $\mathcal{O}(|l||A|)$ pour `est_chemin_simple_sans_issue`, il est invité à l'affiner.

Exercice 8 *Calculs avec les flottants (type B)*

Consignes : Cet énoncé est accompagné d'un code compagnon en C, flottants.c *fournissant les structures de données et certaines fonctions mentionnées dans l'énoncé. Il est à compléter en y implémentant les fonctions demandées. La ligne de compilation* gcc -o main.exe *.c -lm *vous permet de créer un exécutable* main.exe*. Vous pouvez également utiliser l'utilitaire* make*. En ligne de commande, il suffit de taper* make*. Dans les deux cas, si la compilation réussit, le programme peut être exécuté avec la commande* ./main.exe*. Si vous désirez forcer la compilation de tous les fichiers, vous pouvez au préalable nettoyer le répertoire en faisant* make clean *et relancer* make*.*

Un nombre réel x est représenté en machine en base 2 par un flottant qui a un signe s, une mantisse m et un exposant e tel que $x = s \times m \times 2^e$. Dans la norme IEEE 754, en convention normalisé la partie entière de la mantisse est 1 qui est un bit caché. En simple précision, le signe est codé sur 1 bit, la partie décimale de la mantisse sur 23 bits et l'exposant sur 8 bits. En double précision, le signe est codé sur 1 bit, la partie décimale de la mantisse sur 52 bits et l'exposant sur 11 bits.

Dans cet exercice, on observe le résultat de calculs obtenus par un programme. On pourra utiliser la fonction de signature : `double pow(double v, double p)` qui calcule v^p.

1. Dans la fonction principale `main`, on a défini 3 variables a, b, c de type `double`. Compléter le code pour calculer et afficher le résultat des opérations $(a+b)+c$ et $a+(b+c)$. Que constatez-vous ?

2. Compte tenu des approximations faites lors du codage, on peut trouver plusieurs nombres x tels que $1+x=1$ après un calcul fait par la machine. Le plus petit nombre représentable exactement en machine et supérieur à 1 s'écrit $1+\epsilon$, avec ϵ un réel appelé ϵ machine. On admet que ϵ s'écrit sous la forme 2^{-n} avec n un entier naturel strictement positif. Écrire une fonction de signature `double epsilon()` qui renvoie la valeur de n. Justifier cette valeur.

3. On considère une suite $(u_n)_{n\in\mathbb{N}}$ définie par

$$\begin{cases} u_0 &= 2 \\ u_1 &= -4 \\ u_n &= 111 - \frac{1130}{u_{n-1}} + \frac{3000}{u_{n-1}\times u_{n-2}} \text{ si } n \geq 2 \end{cases}$$

   Écrire une fonction de signature `double u(int n)` qui renvoie la valeur du terme u_n.

4. La limite théorique de la suite $(u_n)_{n\in\mathbb{N}}$ est 6. Compléter la fonction `main` afin d'afficher les 22 premiers termes de la suite. Vers quelle valeur semble tendre la suite ?

5. On définit une liste chaînée de nombres à l'aide d'une structure `nb` comportant un double et un pointeur vers une structure `nb` définie ci-dessous

```
struct nb {double x; struct nb* suivant;};
```

   Écrire une fonction de signature `double somme(struct nb* tab)` qui calcule la somme des éléments de la liste `tab`.

6. L'algorithme suivant permet d'augmenter la précision du calcul lors du calcul d'une somme.

> **Entrée :** Une liste l de réels triée dans l'ordre croissant de taille au moins 2.
> **Sortie :** La somme des réels contenus dans la liste l.
> **tant que** la liste l contient strictement plus d'un élément
> | Calculer la somme $s = x + y$ des deux premiers éléments x et y de l
> | Supprimer x et y de l
> | Insérer s dans l de sorte à ce que l reste triée
> **renvoyer** l'unique élément de l

   - Compléter la fonction `somme2` qui implémente cet algorithme.
   - La fonction proposée ne prend pas en compte un cas d'insertion. Illustrer ce propos.

Commentaires du jury

1. Le jury attend une brève explication du comportement constaté.
2. Le jury attend une explication faisant le lien entre la valeur trouvée et le format de représentation des flottants rappelé dans l'énoncé.
3. Une fonction récursive naïve suffit à obtenir tous les points à cette question mais le candidat est bien entendu libre de stocker les valeurs de la suite demandée pour éviter d'avoir à les recalculer.
4. La question ne demande pas de justifier la limite théorique. Le jury doit voir apparaître les premiers termes correctement et clairement affichés.
5.
6. À cette question, un candidat qui aurait fait le choix d'implémenter `somme2` directment et sans utiliser le code déjà fourni et en expliquant sa démarche aurait eu tous les points.