CONCOURS
CPGE
EPITA - IPSA - ESME
1 SEUL ET UNIQUE CONCOURS
POUR VOUS PRÉPARER
À UNE MULTITUDE
DE MÉTIERS

Samedi 8 avril 2023

OPTION : SCIENCES DU NUMÉRIQUE

MPI

Durée : 2 heures

Conditions particulières

Calculatrice interdite
Indiquez uniquement votre code candidat SCEI sur le QCM à insérer dans votre copie d'examen

Concours CPGE EPITA-IPSA-ESME 2023
Option Sciences du Numérique
Filière MPI

Consignes générales

Tout code doit être écrit dans les langages Ocaml ou C, suivant l'énoncé.

- Veuillez indenter votre code correctement.
- Tout ce dont vous avez besoin (fonctions, méthodes) est indiqué ci-dessous.
- Vous pouvez écrire vos propres fonctions, dans ce cas elles doivent être documentées (on doit savoir ce qu'elles font). Dans tous les cas, la dernière fonction écrite doit être celle qui répond à la question.

Consignes Ocaml

La partie I demande d'écrire du code en Ocaml. Toute fonction du module `List` est utilisable, si nécessaire.

Consignes C

La partie II demande d'écrire du code en langage C. On supposera la bibliothèque standard (`stdlib`) et la bibliothèque `stdbool` incluses. On rappelle que cette dernière bibliothèque définit les deux booléens `true` et `false`.

Introduction

Le sujet de l'option Sciences du numérique contient trois sections. Elles sont indépendantes. Le candidat est invité à parcourir intégralement le sujet avant de commencer la rédaction.

I. Algorithmique sur les arbres et programmation en langage Ocaml : Mots de Dyck et arbres binaires

Définition 1. *On appelle mot de Dyck un mot m sur l'alphabet $\{a, b\}$ tel que :*

- *m contient autant de a que de b ;*
- *tout préfixe m contient plus de a que de b ;*

Par exemple, $m = aababb$ est un mot de Dyck, car il possède trois a et trois b, et ses préfixes sont ε (le mot vide), a, aa, aab, $aaba$, $aabab$ et m, et aucun ne contient strictement plus de b que de a.

Question 1. Parmi les mots suivants, indiquer ceux qui sont de Dyck. On ne demande pas de preuve.

ε $\quad$ $aabbbaab$ $\quad$ $aaab$ $\quad$ $abab$ $\quad$ $aaabbb$

On représente dans la suite en Ocaml un mot sur l'alphabet $\{a, b\}$ par la liste de ses lettres. On définit les types suivants :

```
type lettre = A | B ;;
type mot = lettre list ;;
```

Le mot $m = aababb$ sera donc représenté par la liste `[A; A; B; A; B; B]`.

Question 2. Écrire une fonction `verifie_dyck` de type `mot -> bool` renvoyant un booléen indiquant si le mot passé en entrée est de Dyck.

On admet la propriété suivante : tout mot m de Dyck non vide se décompose de manière unique sous la forme $m = aubv$, où u et v sont des mots de Dyck. On pourra utiliser sans démonstration la caractérisation suivante : aub est le plus petit préfixe non vide de m contenant autant de a que de b.

Question 3. Donner sans démonstration la décomposition de chacun des mots de Dyck suivants :

ab $\quad$ $aababb$ $\quad$ $abababb$ $\quad$ $aabbab$

Question 4. Écrire une fonction `decompo_dyck` de type `mot -> mot * mot`, prenant en entrée un mot m supposé non vide et de Dyck (il est inutile de le vérifier), et renvoyant le couple de mots de Dyck (u, v) tel que $m = aubv$.

Il existe une bijection naturelle entre les arbres binaires stricts (tout nœud de l'arbre possède 0 ou 2 fils) et les mots de Dyck, basée sur la décomposition précédente :

- au mot vide est associé l'arbre réduit à une feuille ;
- à un mot de Dyck non vide $aubv$ où u et v sont de Dyck, on associe l'arbre binaire où le sous-arbre gauche est associé au mot u et le sous-arbre droit au mot v.

Question 5. Dessiner l'arbre associé au mot de Dyck $m = aababb$. Les nœuds ne portent pas d'étiquettes.

On définit le type suivant :

```
type arbre = F | N of arbre * arbre ;;
```

Question 6. Écrire une fonction `mot_a_arbre` de type `mot -> arbre` renvoyant l'arbre binaire strict associé à un mot de Dyck (on ne vérifiera pas que le mot passé en entrée est bien de Dyck).

Question 7. Écrire une fonction `arbre_a_mot` de type `arbre -> mot` faisant l'inverse.

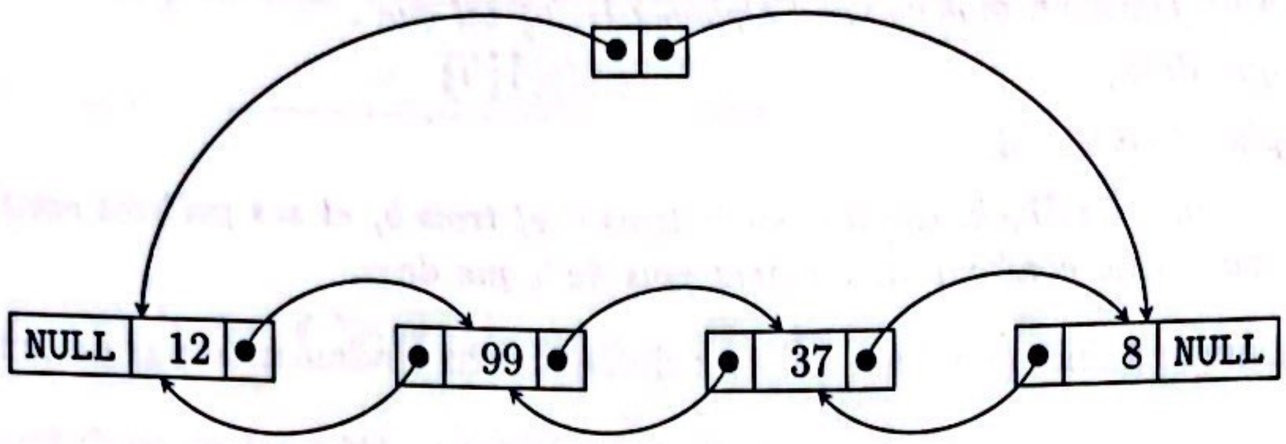
II. Algorithmique et programmation en langage C : structure de liste doublement chaînée et applications

II.1. Structure de liste doublement chaînée

On souhaite réaliser une structure de liste doublement chaînée d'entiers, permettant le retrait et l'ajout des deux côtés. Pour réaliser une telle structure, on utilise, comme sur la figure ci-dessous :

- un type `cellule` à trois champs : une valeur (entière), et deux pointeurs vers les cellules situées à sa gauche et à sa droite si elles existent, `NULL` sinon ;

— un type `dbliste` à deux champs, contenant deux pointeurs vers les cellules situées tout à gauche et tout à droite.



On définit donc les structures suivantes :

```
typedef struct cellule cellule ;
struct cellule
{
  int valeur;
  cellule* suiv;
  cellule* prec;
} ;

typedef struct dbliste dbliste ;
struct dbliste
{
  cellule* gauche;
  cellule* droite;
} ;
```

On prendra garde au fait que :

— une liste doublement chaînée ne contenant aucun élément sera représentée par un élément de type `dbliste` pour lequel les deux pointeurs `gauche` et `droite` sont `NULL` ;

— une liste doublement chaînée contenant un seul élément sera représentée par un élément de type `dbliste` pour lequel les deux pointeurs `gauche` et `droite` pointent vers la même cellule.

On définit également la fonction d'initialisation suivante :

```
void init(dbliste* q)
{
  q->gauche = NULL ;
  q->droite = NULL ;
}
```

Question 8. Écrire la fonction d'en tête `int elem_g(dbliste* q)` permettant de renvoyer l'entier situé à gauche de la liste doublement chaînée pointée par `q`, en supposant celle-ci non vide.

Question 9. Expliquer, à l'aide de schémas, le principe de l'ajout à gauche d'un entier dans la liste doublement chaînée. On fera attention à distinguer le cas d'une liste initialement vide ou non.

Question 10. Écrire la fonction d'en-tête `void push_g(dbliste* q, int v)` permettant l'ajout de l'entier `v` à gauche de la liste doublement chaînée pointée par `q`. On rappelle que `malloc(sizeof(cellule))` permet de réserver un espace mémoire de la taille d'une cellule.

On suppose dans la suite avoir écrit les fonctions d'en-tête suivantes :

— `bool vide(dbliste* q)` : renvoyant un booléen indiquant si `q` pointe sur une liste doublement chaînée vide ou non ;

— `void push_g (dbliste* q, int v)` (resp. `void push_d (dbliste* q, int v)`) permettant l'ajout de l'entier `v` à gauche (resp. à droite) de la liste doublement chaînée pointée par `q` ;

— `int elem_g(dbliste* q)` (resp. `int elem_d(dbliste* q)`) renvoyant l'entier situé à gauche (resp. à droite) de la liste doublement chaînée pointée par `q`, en supposant celle-ci non vide ;

— `int take_g(dbliste* q)` (resp. `int take_d(dbliste* q)`) supprimant et renvoyant l'entier situé à gauche (resp. à droite) de la liste doublement chaînée pointée par `q`, en supposant celle-ci non vide. La mémoire rendue disponible est libérée.

II.2. Élement majoritaire dans une liste

On s'intéresse dans cette sous-partie au problème suivant : savoir si un tableau contient un élément majoritaire. Si un tableau a une taille $n > 0$, un élément de ce tableau est-dit majoritaire s'il apparaît strictement plus que $\lfloor n/2 \rfloor$ fois (on rappelle que $\lfloor . \rfloor$ dénote la partie entière). Par exemple, un tableau de taille 5 contenant comme éléments 2, 7, 2, 2, 9 possède 2 comme élément majoritaire, alors qu'un tableau contenant 2, 7, 2, 2, 9, 8 n'a pas d'élément majoritaire.

On commence par une résolution naïve du problème.

Question 11. Écrire une fonction d'en tête `int occurences(int t[], int taille, int x)` prenant en entrée un tableau de taille `taille`, un élément `x`, et renvoyant le nombre de fois que `x` apparaît dans le tableau.

Question 12. En déduire une fonction d'en tête `bool majoritaire(int t[], int taille)` prenant en entrée un tableau de taille `taille` et renvoyant un booléen indiquant s'il possède un élément majoritaire.

Question 13. Donner la complexité de votre fonction en fonction de la taille notée n du tableau.

On propose une résolution plus efficace dans la suite. Cette approche consiste à calculer d'abord un unique candidat majoritaire, et vérifier si celui-ci l'est effectivement dans un second temps. La première partie de l'approche est détaillée dans l'algorithme ci-dessous. Pour que celui-ci soit plus visuel, on propose l'analogie suivante : le tableau d'entiers est un seau de balles qui peuvent avoir une certaine couleur (deux éléments identiques du tableau correspondent à deux balles de même couleur). On dispose d'un tube se comportant comme une pile, et d'une corbeille.

Algorithme 1 :

Entrée : Le seau de balles
Sortie : une couleur de balle
Créer un tube vide, une corbeille vide;
Placer une balle du seau dans le tube ;
tant que *il reste des balles dans le seau* **faire**
  prendre une balle dans le seau;
  si *la balle est de la même couleur que celle au sommet du tube* **alors**
    mettre la balle à la corbeille
  sinon
    empiler la balle dans le tube;
    si *la corbeille est non vide* **alors**
      prendre une balle de la corbeille et la mettre dans le tube
retourner *la couleur de la balle au sommet du tube*

Le but des questions qui suivent est de montrer que si une couleur était majoritaire dans le seau, c'est celle renvoyée par l'algorithme.

Question 14. Montrer que « toutes les balles éventuellement présentes dans la corbeille ont même couleur que celle au sommet du tube » est un invariant de la boucle.

Question 15. Montrer que « deux balles situées côte à côte (ou l'une en dessous de l'autre, plutôt) dans le tube ont des couleurs différentes » est un invariant de la boucle.

Question 16. Considérons à la fin de l'algorithme une couleur qui n'est pas celle de la balle au sommet du tube. Montrer à l'aide de ces invariants qu'elle apparaissait moins de $\frac{n}{2}$ fois dans le seau, avec n le nombre initial de balles.

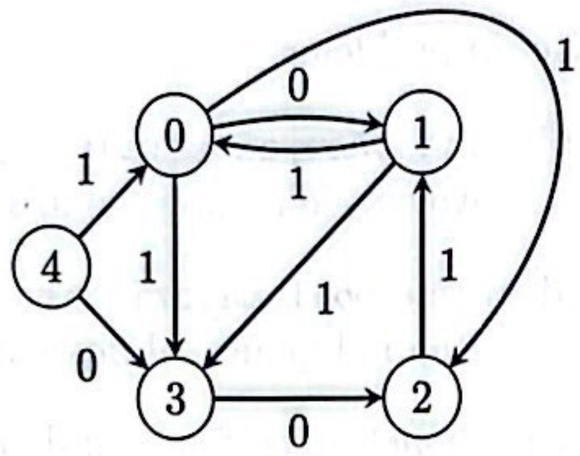
On en déduit donc que pour voir si un tableau admet un élément majoritaire, il suffit d'appliquer l'algorithme précédent, et de tester si l'élément renvoyé est majoritaire. On passe à l'implémentation.

Question 17. Comment obtenir une structure de pile à l'aide de la structure de liste doublement chaînée ? On ne demande pas de coder.

Question 18. Écrire une fonction d'en tête `bool majoritaire_eff(int t[], int taille)` renvoyant la même chose que la fonction de la question 12, mais en suivant l'approche développée dans cette partie. Vous représenterez le tube comme une pile, et la corbeille par un simple entier (indiquer ce qu'il représente). Quelle est la complexité de l'approche ?

II.3. Parcours en largeur dans un graphe 0/1

On considère dans cette sous-section des graphes orientés pondérés, où le poids d'un arc ne peut-être que 0 ou 1. Voici un tel exemple de graphe :



Dans un tel graphe, la distance entre deux sommets s et t est le plus petit poids d'un chemin (s'il en existe au moins un) reliant s à t. Par exemple, la distance entre les sommets 0 et 2 est 1. Plusieurs chemins de poids 1 relient 0 à 2, comme $0 \to 2$ ou $0 \to 1 \to 3 \to 2$.

Question 19. On a déclaré une constante `TAILLE` qui est le nombre de sommets du graphe (5 dans l'exemple ci-dessus). Écrire une fonction d'en tête `void parcours(int graph[TAILLE][TAILLE], int larg[TAILLE], int s)` telle que :

- `graph` est la matrice d'adjacence du graphe : pour $0 \leq i, j \leq$ `TAILLE` $- 1$, `graph[i][j]` contient le poids de l'arc (i, j) si celui-ci existe (0 ou 1), -1 sinon. On suppose que le graphe est sans boucle donc la diagonale ne contient que des -1 ;
- `larg` est un tableau de taille `TAILLE`, à remplir ;
- `s` est un sommet du graphe, donc un entier entre 0 et `TAILLE` $- 1$;

cette fonction modifie le tableau `larg` de sorte qu'à la fin de la fonction `larg[i]` contient la distance entre s et i pour tout sommet i entre 0 et `TAILLE` $- 1$. Si le sommet i n'est pas accessible depuis s, on convient de poser `larg[i]=-1`. Par exemple, avec le graphe représenté ci-dessus, et $s = 0$, voici l'état du tableau `larg` à la fin de l'algorithme :

i	0	1	2	3	4
`larg[i]`	0	0	1	1	-1

Indication : utiliser une liste doublement chaînée pour gérer les sommets à traiter. Si l'arc $i \to j$ est examiné et que j est découvert pour la première fois, insérer j à gauche de la liste doublement chaînée si `graph[i][j]=0`, à droite si `graph[i][j]=1`.

III. Lemme d'Arden et langage reconnu par un automate

Définitions/Notations.

- L'ensemble des langages rationnels sur un alphabet Σ fixé et la plus petite partie de $\mathcal{P}(\Sigma^*)$ contenant $\emptyset$, $\{\varepsilon\}$ et les langages $\{a\}$ pour $a \in \Sigma$, et stable par union, concaténation et étoile de Kleene. On rappelle que pour L un langage, $L^* = \cup_{n \geq 0} L^n$, avec L^n l'ensemble des mots s'écrivant comme concaténation de n mots de L.
- Un automate fini déterministe et un quintuplet $(\Sigma, Q, q_0, F, \delta)$ où :
  - Σ est l'alphabet ;
  - Q est un emsemble fini : l'ensemble des états de l'automate ;
  - $q_0 \in Q$ est l'état initial ;
  - $F \subset Q$ est l'ensemble des états terminaux ;
  - δ est la fonction de transition : c'est une application partielle de $Q \times \Sigma$ vers Q. Un couple $(q, a) \in Q \times \Sigma$ pour lequel $\delta(q, a)$ n'est pas défini est appelé un blocage de l'automate.
- pour $\mathcal{A}$ un tel automate, il est commode d'introduire la fonction de transition étendue δ^*, qui est une application partielle de $Q \times \Sigma^*$ vers Q, définie par

$$\forall q \in Q, \delta(q, \varepsilon) = q \quad \text{et} \quad \forall (q, a, m) \in Q \times \Sigma \times \Sigma^*, \delta^*(q, am) = \begin{cases} \delta^*(\delta(q, a), m) & \text{si ces transitions sont définies} \\ \text{blocage} & \text{sinon.} \end{cases}$$

III.1. Lemme d'Arden

Dans cette sous-partie, on s'intéresse à l'équation $L = (A \cdot L) \cup B$, où l'inconnue est le langage L, A et B étant deux langages fixés.

Question 20. Montrer que $L = A^* \cdot B$ est solution de l'équation.

Question 21. Montrer que c'est la solution minimale de l'équation pour l'inclusion, c'est-à-dire que si L est solution, $A^* \cdot B \subset L$.

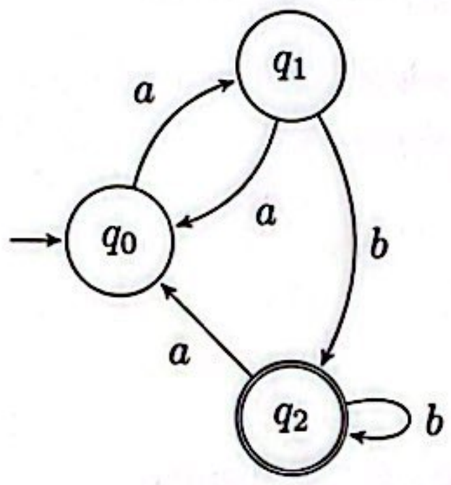
Question 22. Montrer que si $\varepsilon \notin A$, $L = A^* \cdot B$ est l'unique solution de l'équation. On pourra supposer L solution de l'équation et montrer par récurrence forte la propriété $P(n)$: « Si m est un mot de L de longueur n, alors $m \in A^* \cdot B$ ».

On a donc montré le lemme suivant :

Lemme 2 (d'Arden). *L'équation $L = (A \cdot L) \cup B$ possède $A^* \cdot B$ comme solution minimale pour l'inclusion. Si de plus $\varepsilon \notin A$, c'est la seule solution.*

III.2 Un exemple d'application

On considère l'automate suivant, où q_0 est l'état initial, et $F = \{q_2\}$.



Pour $i \in \{0, 1, 2\}$, on note $L_i = \{m \in \Sigma^*, \delta^*(q_i, m) \in F\}$.

Question 23. Montrer que $L_2 = \{\varepsilon\} \cup bL_2 \cup aL_0$. *Indication :* pour m un mot non vide de L_2, justifier que s'il commence par la lettre a, il est dans aL_0, et s'il commence par la lettre b, il est dans bL_2.

Question 24. À l'aide du lemme d'Arden, exprimer L_2 en fonction de L_0.

Question 25. Donner des équations similaires pour L_0 et L_1.

Question 26. À l'aide du lemme d'Arden, déterminer une expression rationnelle dénotant L_0 (le langage reconnu par l'automate).

III.3 Cas général

À l'aide du lemme d'Arden, on souhaite redémontrer le résultat de cours suivant :

Théorème 3. *Le langage reconnu par un automate fini déterministe est un langage rationnel.*

On va en donner une preuve constructive, c'est-à-dire une méthode permettant de calculer une expression rationnelle reconnaissant le langage, à l'aide du lemme d'Arden. Dans la suite, on se donne $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$ un automate fini déterministe quelconque. On suppose $Q = \{q_0, \dots, q_{n-1}\}$, et on note $L_i = \{m \in \Sigma^*, \delta^*(q_i, m) \in F\}$, comme dans la section précédente.

Question 27. Justifier que pour tout $i \in [\![0, n-1]\!]$,

$$L_i = \left(\bigcup_{j=0}^{n-1} A_{i,j} L_j \right) \bigcup B_i$$

où $A_{i,j} \subset \Sigma$ et $B_i \in \{\emptyset, \{\varepsilon\}\}$. Décrire $A_{i,j}$ et B_i en fonction de δ.

On a donc obtenu un système d'équations en les $(L_i)_{0 \leq i \leq n-1}$. Voyons comment éliminer une inconnue.

Question 28. 1. Expliquer comment exprimer L_{n-1} en fonction de $L_0, \dots, L_{n-2}$, à l'aide du lemme d'Arden.

2. Montrer qu'il existe des langages $(A'_{i,j})_{0\leq i,j\leq n-2}$ rationnels, ne contenant pas ε, et des langages $(B'_i)_{0\leq i\leq n-2}$ rationnels, tels que

$$\forall i \in [\![0, n-2]\!] \quad L_i = \left(\bigcup_{j=0}^{n-2} A'_{i,j} L_j \right) \bigcup B'_i$$

Exprimer précisément les $(A'_{i,j})_{0\leq i,j\leq n-2}$ et $(B'_i)_{0\leq i\leq n-2}$ en fonction des $(A_{i,j})_{0\leq i,j\leq n-1}$ et $(B_i)_{0\leq i\leq n-1}$.

Question 29. Montrer que L_0 est rationnel. On pourra procéder par récurrence, pour cela énoncer clairement la propriété à démontrer.