Devoir surveillé du Samedi 14 Mars

Dans l'ensemble du devoir, toutes les fonctions seront :

- *écrites en langage OCaml,*
- *précédées d'une explication des variables utilisées,*
- *précédées d'une explication de l'algorithme.*

Exercice 1

Dans cet exercice, on étudie différents langages sur l'alphabet $A = \{0, 1\}$ à deux lettres. On note A^* l'ensemble des mots construits sur l'alphabet A. Le mot vide est noté ε. En OCaml, un mot sur A est implémenté par le type :

```
type mot = bool list
```

à savoir par une liste de booléens où la lettre 0 est représentée par le booléen `false` et la lettre 1 par le bouléen `true`.

1. On note L_1 le langage des mots de A qui représentent l'écriture binaire d'un entier naturel, où le bit de poids faible se situe en fin de mot. Pour assurer l'unicité de la représentation, l'écriture binaire d'un entier ne commence jamais par 0. C'est pourquoi l'entier nul est représenté par le mot vide.
   (a) Donner l'écriture binaire de l'entier 41.
   (b) Donner l'entier représenté par le mot 10101010.
   (c) Pour un automate, que signifie la propriété d'être local standard ?
   (d) Dessiner un automate local standard $\mathcal{A}_1$ reconnaissant le langage L_1.
   (e) Écrire en OCaml une fonction `langage_1` de type `mot -> bool` qui prend en argument un mot m et qui renvoie `true` si et seulement si m appartient au langage L_1.
2. On note L_2 le langage dénoté par l'expression rationnelle $(0 + 1)^*.0$.
   (a) Dessiner un automate déterministe $\mathcal{A}_2$ reconnaissant le langage L_2.
   (b) Écrire en OCaml une fonction `langage_2` de type `mot -> bool` qui prend en argument un mot m et qui renvoie `true` si et seulement si m appartient au langage L_2.
3. On note L_3 le langage reconnu par l'automate déterministe $\mathcal{A}_3$ défini par :
   — l'ensemble d'états $Q = \{0, 1, 2\}$;
   — l'état initial $i = 0$;
   — l'ensemble d'états finals $F = \{0\}$;
   — la fonction de transition $\delta : Q \times A \to Q$ définie par :

$$\forall q \in Q,\ \forall a \in A, \quad \delta(q, a) = (2q + a) \bmod 3.$$

   On rappelle que $(n \bmod 3)$ désigne le reste de la division euclidienne de l'entier n par 3.
   (a) Dessiner l'automate $\mathcal{A}_3$.
   (b) Écrire en OCaml une fonction `langage_3` de type `mot -> bool` qui prend en argument un mot m et qui renvoie `true` si et seulement si m appartient au langage L_3.
   (c) Pour tout $n \in \mathbb{N}^*$, démontrer la propriété $\mathscr{P}(n)$ suivante :

$$\forall \omega_1, \ldots, \omega_n \in A, \quad \delta^\star(0, \omega_1 \ldots \omega_n) = \sum_{k=1}^{n} \omega_k 2^{n-k} \bmod 3$$

où la fonction $\delta^\star : Q \times A^* \to Q$ est définie par

$$\forall q \in Q,\ \forall \omega \in A^*,\ \forall a \in A, \quad \delta^\star(q, \varepsilon) = q \quad \text{et} \quad \delta^\star(q, \omega \cdot a) = \delta(\delta^\star(q, \omega), a).$$

4. On note $L_4 = L_1 \cap L_2 \cap L_3$.
   (a) Décrire simplement l'ensemble des mots du langage L_4.
   (b) Existe-t-il un automate reconnaissant le langage L_4 ?

Exercice 2

Partie 1 : Algorithmes de calcul des ensembles P, S et F.

Le but de cette partie est de déterminer les éléments caractéristiques (précisés plus loin) d'un langage dénoté par une expression rationnelle.

1. Définitions.
   Soit Σ un alphabet, ε désigne le mot vide, L un langage sur Σ.
   (a) Donner les définitions de $P(L)$, $S(L)$, $F(L)$ et $N(L)$.
   (b) Comparer au sens de l'inclusion $L\backslash\{\varepsilon\}$ et $\Big(P(L)\Sigma^* \cap \Sigma^* S(L)\Big)\backslash\Big(\Sigma^* N(L)\Sigma^*\Big)$, une preuve est attendue.
   (c) Dans le cas où : $L\backslash\{\varepsilon\} = \Big(P(L)\Sigma^* \cap \Sigma^* S(L)\Big)\backslash\Big(\Sigma^* N(L)\Sigma^*\Big)$, quel est le qualificatif donné au langage L ?
2. Propriétés.
   Soit Σ un alphabet, ε désigne le mot vide.
   (a) Donner sans aucune justification, $P(\{\varepsilon\})$, $S(\{\varepsilon\})$ et $F(\{\varepsilon\})$.
   (b) Pour tout lettre a appartenant à Σ, donner sans aucune justification, $P(\{a\})$, $S(\{a\})$ et $F(\{a\})$.
   (c) Soit L_1 et L_2 deux langages sur Σ.
   Exprimer sans aucune justification, $P(L_1 \cup L_2)$, $S(L_1 \cup L_2)$ et $F(L_1 \cup L_2)$ en fonction de $P(L_1)$, $P(L_2)$, $S(L_1)$, $S(L_2)$, $F(L_1)$ et de $F(L_2)$.
   (d) Soit L_1 et L_2 deux langages sur Σ.
   Exprimer sans aucune justification $P(L_1.L_2)$, $S(L_1.L_2)$ et $F(L_1.L_2)$ en fonction de $P(L_1)$, $P(L_2)$, $S(L_1)$, $S(L_2)$, $F(L_1)$ et de $F(L_2)$.
   (e) Soit L un langage sur Σ.
   Exprimer sans aucune justification $P(L^*)$, $S(L^*)$ et $F(L^*)$ en fonction de $P(L)$, $S(L)$ et $F(L)$.
3. Implémentation en OCaml.
   Une lettre de l'alphabet Σ est représentée en OCaml par le type `string`,
   un mot sur Σ est représenté en OCaml par le type `string`,
   un langage L sur Σ est représenté en OCaml par le type :
   ```
   type langage = string list
   ```
   les doublons sont interdits.

Nous utiliserons le type OCaml suivant pour représenter une expression rationnelle :

```
type exprat =
  | Epsilon
  | Const of string
  | Sum of exprat * exprat
  | Concat of exprat * exprat
  | Kleene of exprat
;;
```

Comment est représentée l'expression rationnelle $(a + b)^*.a$ en OCaml ?

4. Quelques fonctions utiles pour la suite.
   (a) Écrire une fonction en OCmal qui prend en argument une expression rationnelle `e` et renvoie `true` dans le cas où le mot vide appartient au langage dénoté par `e`, `false` sinon :
   `mot_vide : exprat -> bool`
   (b) Écrire une fonction en OCmal qui prend en argument, deux langages `L1` et `L2` et renvoie la réunion des deux langages `L1` et `L2` en évitant les doublons :
   `reunion : langage -> langage -> langage`
   (c) Écrire une fonction en OCaml qui prend en argument, deux langages `L1` et `L2` ne contenant, ni l'un ni l'autre, le mot vide, et renvoie le langage `L1 . L2` en évitant les doublons.
   `produit : langage -> langage -> langage`
5. Algorithmes de calcul des ensembles P, S et F.
   (a) Écrire une fonction en OCaml ayant pour argument une expression rationnelle `e` et renvoie le langage $P(L)$ où L désigne le langage dénoté par `e` :
   `calculP : exprat -> langage`
   (b) Écrire une fonction en OCaml ayant pour argument une expression rationnelle `e` et renvoie le langage $S(L)$ où L désigne le langage dénoté par `e` :
   `calculS : exprat -> langage`
   (c) Écrire une fonction en OCaml ayant pour argument une expression rationnelle `e` et renvoie le langage $F(L)$ où L désigne le langage dénoté par `e`
   `calculF : exprat -> langage`

Partie 2 : Recherche d'un mot dans un autre.

Soit q un entier supérieur ou égal à 2, Σ un alphabet constitué de q lettres.

Nous considérons deux mots sur l'alphabet Σ : le premier sera appelé *texte* et noté $t = t_0\, t_1 \ldots t_{n-1}$, n appartenant à $\mathbb{N}$ (le cas $n = 0$ correspondant au mot vide), le second mot sera appelé *motif* et noté $x = x_0\, x_1 \ldots x_{m-1}$, m appartenant à $[\![0, n]\!]$ (le cas $m = 0$ correspondant au mot vide).

L'objectif est de trouver des algorithmes permettant de savoir si le motif x est présent dans le texte t c'est-à-dire s'il existe un entier naturel k tel que :

$$t_k \ldots t_{k+m-1} = x_0\, x_1 \ldots x_{m-1}\,.$$

Dans toute la suite de cet énoncé, nous conservons ces notations sans les redéfinir systématiquement.

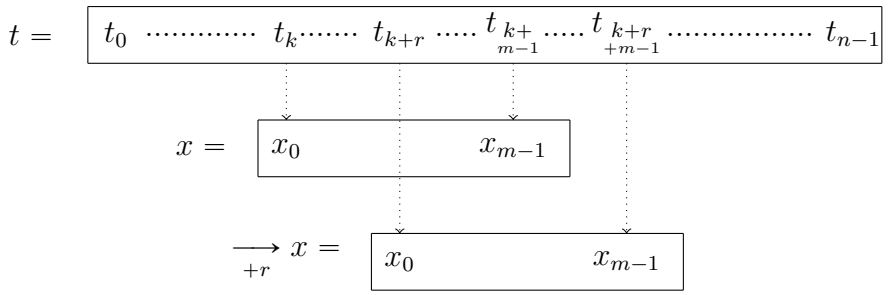
Pour la programmation en OCaml, nous supposerons qu'une lettre ou un mot est représenté par le type `string`.

Pour les calculs de complexité, l'opération élémentaire choisie est la comparaison entre les lettres.

Les algorithmes seront du type suivant : nous comparons x avec un facteur $t_k \dots t_{k+m-1}$ de t, et en cas d'échec, nous décalons x vers la droite de t d'un certain nombre de lettres et nous comparons de nouveau x avec un autre facteur de t de la forme $t_{k+r} \dots t_{k+r+m-1}$ avec r un entier supérieur ou égal à 1.

Le problème essentiel est le choix du décalage, afin de minimiser le nombre de comparaisons de lettres.

Nous pouvons représenter schématiquement ces comparaisons de la manière suivante :



6. Méthode naïve en itératif.

   La méthode la plus immédiate consiste à simplement décaler x d'une lettre, c'est-à-dire à toujours prendre : $r = 1$.

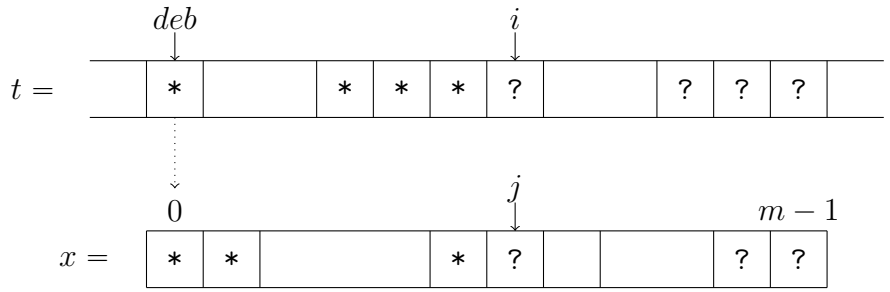
   (a) Écrire en OCaml, une fonction `recherche_iterative` de signature :

   ```
   recherche_iterative : string -> string -> bool
   ```

   qui détermine si x est ou non une sous-chaine du mot t.

   Nous respecterons l'invariant de boucle suivant :

   deb désigne l'indice dans t du début de la recherche courante de x dans t, i désigne l'indice courant dans t, j désigne l'indice courant dans x,



   où : $\forall\, k \in [\![0, j-1]\!]$, $t[deb + k] = x[k]$, et nous testons alors l'égalité : $t[i] = x[j]$.

   (b) Le but de cette question est de prouver la terminaison de l'algorithme utilisé par la fonction `recherche_iterative`.

   i. Rappeler vos connaissances sur l'ordre lexicographique sur $\mathbb{N}^2$.

   ii. En considérant les couples $(n - deb, m - j)$, démontrer que l'algorithme `recherche_ierative` se termine.

   (c) Quelle est, en fonction de n et m, la complexité au pire de cet algorithme de recherche ?

7. Méthode naïve en récursif.

   Dans cette question 7 et uniquement dans cette question 7, les mots sont représentés par une structure de type `char list`.

   Les deux raisonnements suivants vont permettre de construire une méthode récursive :
   - Le motif x est un préfixe du texte t si et seulement si x est le mot vide ou satisfait les deux conditions suivantes :
     - le premier caractère de x est identique au premier caractère de t,
     - x privé de son premier caractère est un préfixe de t privé de son premier caractère.
   - Le motif x est un sous-mot de t si et seulement si x est un préfixe de t ou x est un sous-mot de t privé de son premier caractère.

   (a) Nous allons dans un premier temps nous intéresser au problème suivant : "le motif x est-il un préfixe du mot t ?".

   Écrire en OCaml, une fonction `est_prefixe_recursif` de signature :

   ```
   est_prefixe_recursif : 'a list -> 'a list -> bool
   ```

   qui détermine si le motif x est ou non un préfixe du texte t en utilisant l'idée ci-dessus.

   (b) Nous allons maintenant nous intéresser au problème suivant : "le motif x est-il un sous-mot du texte t ?".

   Écrire en OCaml, une fonction `recherche_recursive` de signature :

   ```
   recherche_recursive : 'a list -> 'a list -> bool
   ```

   qui détermine si le motif x est ou non un sous-mot du texte t en utilisant l'idée ci-dessus.

   (c) Nous allons nous intéresser à la terminaison des deux algorithmes récursifs ci-dessus.

   i. Justifier la terminaison de l'algorithme `est_prefixe_recursif`.

   ii. Justifier la terminaison de l'algorithme `recherche_recursive`.

8. Algorithme KMP.

   Dans cette section, nous étudions un algorithme qui fut inventé indépendamment par Knuth et Pratt et par Morris ; leurs travaux furent publiés conjointement en 1977. Cet algorithme a un temps d'exécution en $O(n+m)$ dans le pire des cas.

   Illustrons l'idée de cet algorithme par un exemple. Nous recherchons le motif *maman* dans le texte *monpapimamamiemonpapamamaman*,

   et nous allons nous retrouver sur la comparaison :

$$\begin{array}{l|l|l} monpapimamamiemonpapa & ma\underline{ma}m & an \\ & ma\underline{ma}n & \end{array}$$

   et le dernier caractère du motif n ne correspond pas au caractère m du texte.

   L'idée de l'algorithme KMP est d'utiliser le fait que nous savons déjà que les deux caractères soulignés qui forment un préfixe du motif sont identiques aux caractères du texte et nous allons alors poursuivre la recherche en décalant le motif de deux caractères (au lieu d'un par la méthode naïve) et de commencer la comparaison directement au troisième caractère du motif (au lieu de recommencer au début du motif par la méthode naïve) :

$$\begin{array}{l|l|l} monpapimamamiemonpapama & \underline{ma} & man \\ & \underline{ma} & man \end{array}$$

   Il n'est pas évident de comprendre que nous n'allons pas, avec ces décalages de plusieurs caractères d'un coup, rater l'apparition du motif.

(a) Écrire une fonction `compare_sub_strings` de signature :

```
compare_sub_strings : string -> int -> string -> int -> int -> int
```

prenant pour argument un mot v, un entier k, un mot w, un entier l et un entier s et retournant le plus grand entier naturel j inférieur ou égal à s tel que : $v_k...v_{k+j-1} = w_l...w_{l+j-1}$,
et précisons que le cas : $v_k \neq w_l$ se traduit par : $j = 0$.

(b) En déduire une fonction `test_egalite_sub_strings` de signature :

```
test_egalite_sub_strings : string -> int -> string -> int -> int -> bool
```

prenant pour argument un mot v, un entier k, un mot w, un entier l et un entier s et renvoyant un booléen caractérisant le fait : $v_k...v_{k+s-1} = w_l...w_{l+s-1}$.

(c) Bords d'un mot.
Soit s appartenant à $\mathbb{N}^*$, $w = w_0 \dots w_{s-1}$ un mot non vide.
Nous appelons *bord de* w le plus long préfixe strict de w (c'est-à-dire différent de w) qui soit également un suffixe de w, précisons que le bord de w pouvant éventuellement être le mot vide de longueur 0, il a donc une longueur j comprise entre 0 et $s-1$.
Par exemple, le bord de *macadama* est *ma*, le bord de *mama* est *ma*, le bord de *maman* est le mot vide.
En testant les préfixes stricts du mot w un par un en partant du plus grand, écrire une fonction `border_length` de signature :

```
border_length : string → int
```

qui calcule la longueur du bord d'un mot w.

(d) Pour implémenter l'algorithme de Knuth-Morris-Pratt, nous aurons besoin de connaitre la longueur du bord de chaque préfixe de x, c'est-à-dire de tous les mots de la forme $x_0x_1 \dots x_{j-1}$, j appartenant à $[\![1, m]\!]$.
A l'aide de la fonction `border_length`, écrire en OCaml, une fonction `borders` de signature :

```
borders : string -> int array
```

prenant pour argument le mot x et renvoyant un tableau β tel que pour tout j appartenant à $[\![1, m]\!]$, $\beta(j)$ contient la longueur du bord du préfixe $x_0x_1...x_{j-1}$, et par convention, nous posons : $\beta(0) = -1$.

(e) Le principe de l'algorithme de Knuth, Morris et Pratt est d'utiliser la comparaison de x avec $t_kt_{k+1}...t_{k+m-1}$ pour déterminer le décalage à réaliser avant d'effectuer la comparaison suivante :
— dans le cas où : $t_k \neq x_0$, nous décalons simplement x d'une lettre vers la droite,
— sinon, nous considérons j maximal tel que : $t_k \dots t_{k+j-1} = x_0 \dots x_{j-1}$ puis nous décalons alors x de $j - \beta(j)$ lettres vers la droite et nous commençons la comparaison de x avec ce nouveau facteur de t à la lettre $x_{\beta(j)}$.
Montrer qu'en faisant ce décalage, nous n'avons pas "oublié" une apparition de x dans t.

(f) En utilisant la fonction `borders`, définir une fonction `kmp` de signature :

```
kmp: string → string → bool
```

qui implémente l'algorithme de Knuth, Morris et Pratt.