Devoir surveillé n° 2 — 3h

L'utilisation d'Internet, le partage d'informations avec autrui et la consultation de tout document sont interdits pendant toute la durée de l'épreuve.

Mise en place

- Se connecter avec le nom d'utilisateur `ds2-<login>` où `<login>` est votre nom d'utilisateur du laboratoire d'informatique (tout en minuscules avec les tirets) et le mot de passe par défaut.
- Ouvrir un terminal et entrer la commande `ls`. Vous devez voir les trois fichiers suivants :
  - `ds2-<login>.ml`
  - `expect_ds2.exp`
  - `eval_ds2.sh`

  Vous écrirez vos programmes CAML exclusivement dans le fichier de nom `ds2-<login>.ml` qui se trouve à la racine de votre répertoire, sans en changer le nom ni l'emplacement.
- Ouvrir EMACS à l'aide de la commande :

```
emacs ds2-<login>.ml &
```

- Revenir dans le terminal et entrer la commande suivante :

```
bash eval_ds2.sh ds2-<login>.ml
```

  Vous devez alors voir l'évaluation de votre code : aucune fonction n'est encore définie. Au fur et à mesure de l'épreuve, vous pouvez vérifier que vos fonctions sont bien prises en compte à l'aide de cette commande (n'oubliez pas que l'on peut remonter dans l'historique des commandes avec les flèches du clavier pour éviter de les retaper à chaque fois).

Consignes

Votre programme devra respecter scrupuleusement les noms et les types indiqués (ou un type plus général ou équivalent bien entendu) car il sera corrigé automatiquement. Le code fourni devra compiler sans erreurs ni aucun message d'avertissement. Vous n'utiliserez que des caractères ASCII pour les noms de variables. Il est impératif de veiller au strict respect de ces consignes.

Lorsque l'on pose des hypothèses sur les arguments (par exemple qu'un argument est un entier naturel), il n'est pas nécessaire dans vos fonctions de le vérifier ni de prévoir le comportement si l'hypothèse est violée (par exemple si l'argument est négatif). On pourra cependant lever des exceptions (`failwith "Argument non valable"` par exemple) si on le souhaite.

Le script d'évaluation affiche `vrai` si votre fonction est bien définie et possède, a priori, le bon type. Cela ne garantit pas du tout que votre programme est correct ! On prendra donc, comme d'habitude, bien soin de tester toutes les fonctions sur des cas triviaux, puis un peu plus complexes.

Rappel EMACS

Sauvegarder	CTRL-X, CTRL-S
Interpréter la phrase courante	CTRL-X, CTRL-E
Tout interpréter	CTRL-C, CTRL-B
Interruption	CTRL-C, CTRL-K
Quitter	CTRL-X, CTRL-C

N'oubliez pas de sauvegarder ! Conseil : toujours utiliser la combinaison de commandes CTRL-X puis CTRL-S (sauvegarder) *puis* CTRL-X puis CTRL-E (interpréter).

I Écriture en base b

Proposition I.1

Soit $n \in \mathbb{N}$ et $b \in \mathbb{N}^* \setminus \{1\}$. Il existe un unique polynôme $P = \sum_{k=0}^{d} a_k X^k$ à coefficients dans $[\![0; b-1]\!]$, avec $a_d \neq 0$, tel que $n = P(b)$. La suite des coefficients $(a_k)_{k \in [\![0;d]\!]}$ est appelée *décomposition* de n dans la base b.

Dans cet exercice, on prend la liste des coefficients dans l'ordre *décroissant des indices*, ce que l'on appelle *représentation* d'un entier en base b. Par exemple, la représentation de 2017 en base 10 est $[2; 0; 1; 7]$.

Question 1

Écrire une fonction `ecriture_base` qui calcule la représentation d'un entier naturel n en base b. La complexité devra être linéaire en la taille de la représentation. On suppose que $b \geq 2$, il est inutile de le vérifier. Par exemple :

OCAML

```
ecriture_base 87 2;;
- : int list = [1; 0; 1; 0; 1; 1; 1]
```

On pourra au choix utiliser la fonction `List.rev` ou chercher une fonction auxiliaire récursive terminale.

Question 2

Écrire la réciproque `lecture_base`. La complexité devra encore être linéaire. On supposera que la liste en premier argument est bien une représentation valable en base b, il n'est donc pas utile de le vérifier. Par exemple :

OCAML

```
lecture_base [1; 2; 0; 1] 3;;
- : int = 46
```

II Révisions : tri fusion et tri rapide

Question 3

Implémenter une fonction `division : 'a list -> 'a list * 'a list` qui divise une liste passée en argument en deux sous-listes de même taille, à un près. La première sous-liste doit contenir les éléments d'indices pairs et la deuxième ceux d'indices impairs. Par exemple :

OCaml
```
division [1; 4; 2; 6; 7; 1; 2; 4; 2]
- : int list * int list = ([1; 2; 7; 2; 2], [4; 6; 1; 4])
```

Question 4

Implémenter une fonction `fusion : 'a list -> 'a list -> 'a list` qui prend en argument deux listes, supposées triées, et qui renvoie la liste triée formée des éléments des deux listes (avec doublons). Par exemple :

OCaml
```
fusion [1; 2; 2; 2; 7] [1; 4; 4; 6]
- : int list = [1; 1; 2; 2; 2; 4; 4; 6; 7]
```

Question 5

Implémenter une fonction `tri_fusion : 'a list -> 'a list` qui trie la liste reçue en argument. Par exemple :

OCaml
```
tri_fusion [1; 4; 2; 6; 7; 1; 2; 4; 2]
- : int list = [1; 1; 2; 2; 2; 4; 4; 6; 7]
```

Question 6

Quelle est sa complexité dans le pire cas ? Si n désigne la longueur de la liste, définir la constante de type `int`, `complexite_tri_fusion`, dont la valeur est le numéro de la réponse correcte.

1 : $\Theta(1)$
2 : $\Theta(\log n)$
3 : $\Theta(n)$
4 : $\Theta(n \log n)$
5 : $\Theta(n^2)$
6 : $\Theta(n^3)$
7 : $\Theta(2^n)$
8 : Aucune des réponses ci-dessus.

Question 7

Écrire une fonction `partition : 'a list -> 'a -> 'a list * 'a list` tel que l'appel `partition liste pivot` partitionne la liste `liste` en deux sous-listes `(petits, grands)`. Les éléments plus petits ou égaux à `pivot` sont renvoyés dans la liste `petits` ; et ceux strictement plus grands sont renvoyés dans la liste `grands`. L'ordre relatif des élé-

ments dans les sous-listes doit être conservé. Par exemple :

OCAML
```
partition [2; 3; 1; 4; 6; 3; 1; 5; 3] 4;;
- : int list * int list = ([2; 3; 1; 4; 3; 1; 3], [6; 5])
```

Question 8

Écrire une fonction `tri_rapide : 'a list -> 'a list`, qui trie une liste par ordre croissant. On choisira toujours le premier élément comme pivot. On rappelle que le principe du tri rapide est de partitionner la liste (sans le pivot), en triant récursivement les deux sous-listes et en concaténant ces deux sous-listes triées. Comme les deux sous-listes sont triées, que les éléments de `tri petits` sont inférieurs à `pivot` et que tous ceux-ci sont inférieurs à `tri grands`, la concaténation renvoyée est bien triée. Par exemple :

OCAML
```
tri rapide [4; 2; 3; 1; 4; 6; 3; 1; 5; 3];;
- : int list = [1; 1; 2; 3; 3; 3; 4; 4; 5; 6]
```

Question 9

Quelle est sa complexité dans le pire cas ? Si n désigne la longueur de la liste, définir la constante de type `int`, `complexite_pire_tri_rapide`, dont la valeur est le numéro de la réponse correcte.

1 : $\Theta(1)$
2 : $\Theta(\log n)$
3 : $\Theta(n)$
4 : $\Theta(n \log n)$
5 : $\Theta(n^2)$
6 : $\Theta(n^3)$
7 : $\Theta(2^n)$
8 : Aucune des réponses ci-dessus.

Question 10

Quelle est sa complexité dans le cas moyen ? Si n désigne la longueur de la liste, définir la constante de type `int`, `complexite_moyenne_tri_rapide`, dont la valeur est le numéro de la réponse correcte.

1 : $\Theta(1)$
2 : $\Theta(\log n)$
3 : $\Theta(n)$
4 : $\Theta(n \log n)$
5 : $\Theta(n^2)$
6 : $\Theta(n^3)$
7 : $\Theta(2^n)$
8 : Aucune des réponses ci-dessus.

III Nombre d'inversions dans une liste

Dans cette partie, on pourra utiliser librement les fonctions `List.length` et `List.rev` (mais sans oublier leur contribution éventuelle à la complexité).

On appelle *inversion* d'une suite finie d'éléments distincts $(x_1, \ldots, x_n)$ tout couple (i, j) tel que $i < j$ mais $x_i > x_j$. Par exemple, les inversions de $(2, 3, 1, 5, 4)$ sont (1, 3), (2, 3) et (4, 5).

Question 11

Écrire en CAML une fonction `nb_inversions : 'a list -> int` permettant de calculer le nombre d'inversions d'une liste d'éléments distincts, en testant un à un tous les couples (i, j) avec $i < j$. Par exemple :

OCAML
```
nb_inversions [2; 3; 1; 5; 4];;
- : int 3
```

Question 12

Quelle est sa complexité ? Si n désigne la longueur de la liste, définir la constante de type `int`, `complexite_nb_inversions`, dont la valeur est le numéro de la réponse correcte.

1 : $\Theta(1)$
2 : $\Theta(\log n)$
3 : $\Theta(n)$
4 : $\Theta(n \log n)$
5 : $\Theta(n^2)$
6 : $\Theta(n^3)$
7 : $\Theta(2^n)$
8 : Aucune des réponses ci-dessus.

On se propose d'adopter une stratégie *diviser pour régner* pour résoudre ce problème de manière plus efficace. Dans la stratégie diviser pour régner on commence par diviser la liste en deux.

Question 13

Écrire une fonction `decoupe : 'a list -> 'a list * 'a list` qui divise une liste en deux listes de même taille à une unité près, en conservant l'ordre. Si le résultat de `decoupe liste` est un couple `(liste1, liste2)` alors `liste1 @ liste2 = liste` et `0 <= |liste2| - |liste1| <= 1`. On garantira une complexité linéaire en $O(n)$ où n désigne la longueur de la liste. Attention : contrairement au tri fusion, il faut ici conserver l'ordre des éléments et c'est la deuxième liste qui peut éventuellement contenir un élément de plus. Par exemple :

OCAML
```
decoupe [2; 3; 1; 5; 4];;
- : int list * int list = ([2; 3], [1; 5; 4])
```

Les inversions sont alors :

- les inversions de chacune des deux sous-listes ;
- plus les inversions qui sont à cheval entre les deux.

On obtient bien sûr les inversions de chacune des deux sous-listes de manière récursive. Si les deux sous-listes sont triées, alors on peut calculer les inversions à cheval de manière efficace.

Question 14

Écrire une fonction `inversions_a_cheval : 'a list -> 'a list -> int` qui calcule le nombre d'inversions à cheval entre deux sous-listes supposées triées. On garantira une complexité en $O(n)$ où n est la somme des longueurs des deux listes. Par exemple :

OCaml

```
inversions_a_cheval [2; 3] [1; 5; 4];;
- : int = 2
```

Indication : on pourra introduire une fonction auxiliaire qui utilise une variable auxiliaire contenant la taille de la première liste.

Pour trier les deux sous-listes, on pourrait, à chaque appel récursif, utiliser une fonction de tri en $O(n \log n)$. Mais alors le coût de division et de recombinaison serait dominé par cette complexité et ne serait plus en $O(n)$. En fait, en regardant de près l'algorithme que l'on est en train de mettre en place on s'aperçoit que sa structure est très proche de celle du tri fusion. Une idée est donc de calculer le nombre d'inversions *en même temps* que l'on applique le tri fusion. Ceci permet, à chaque étape, par un même appel récursif, d'obtenir directement à la fois le nombre d'inversions des deux sous-listes *et* les deux sous-listes triées, ce qui permet alors de calculer également le nombre d'inversions à cheval de manière efficace.

Question 15

Écrire une fonction `tri_fusion_et_inversions : 'a list -> ('a list * int)` qui renvoie le couple formé de la liste triée et du nombre d'inversions de la liste initiale. Par exemple :

OCaml

```
tri_fusion_et_inversions [2; 3; 1; 5; 4];;
- : int list * int = ([1; 2; 3; 4; 5], 3)
```

On pourra réutiliser la fonction `fusion` écrite dans la question 4 du sujet.

Question 16

En déduire une fonction `nb_inversions_diviser_pour_regner : 'a list -> int` qui calcule le nombre d'inversions d'une liste en utilisant une stratégie *diviser pour régner*.

Question 17

Quelle est sa complexité ? Si n désigne la longueur de la liste, définir la constante de type `int`, `complexite_nb_inversions_diviser_pour_regner`, dont la valeur est le numéro de la réponse correcte.

1 : $\Theta(1)$
2 : $\Theta(\log n)$
3 : $\Theta(n)$
4 : $\Theta(n \log n)$
5 : $\Theta(n^2)$
6 : $\Theta(n^3)$
7 : $O(2^n)$
8 : Aucune des réponses ci-dessus.

IV Arbres de PATRICIA

L'objectif de ce problème est d'étudier l'utilisation des arbres PATRICIA[1] pour représenter des ensembles de mots de manière compacte tout en permettant un test d'appartenance rapide. Une représentation efficace d'ensembles de mots est importante en traitement automatique des langues naturelles par exemple.

IV.1 Mots et séquences de mots

Définition IV.1

Un mot m est une séquence de taille $n \geq 0$ de caractères c_i $1 \leq i \leq n$. Il est noté $c_1 c_2 \ldots c_n$. Sa taille n est notée $|m|$. Le mot correspondant à la séquence de taille 0 est appelé *mot vide* et noté ϵ.

Un mot est représenté en CAML par le type :

OCAML
```
type mot = char list;;
```

Exemple 1

La liste `['f'; 'a'; 'c'; 'e']` *représente le mot « face ».*

Définition IV.2

Une séquence s de taille $n \geq 0$ de mots m_i avec $1 \leq i \leq n$ est notée $(m_1, \ldots, m_n)$. Sa taille n est notée $|s|$.

Une séquence de mots est représentée en CAML par le type :

OCAML
```
type mots = mot list;;
```

En CAML les caractères de type `char` sont munis d'un ordre strict total $<$. Par exemple :

OCAML
```
'a' < 'b';;
- : bool = true
```

On définit un ordre strict partiel sur les mots, noté $\prec$, de la manière suivante. Pour deux mots m_1 et m_2, $m_1 \prec m_2$ si et seulement si m_1 et m_2 ne sont pas vides et si le premier caractère de m_1 est strictement inférieur au premier caractère de m_2. On dit que m_1 précède m_2.

Question 18

Écrire une fonction `precede : mot -> mot -> bool` qui implémente cette relation de précédence stricte.

1. *Practical Algorithm To Retrieve Information Coded In Alphanumeric* ce qui se traduit par « algorithme pratique pour retrouver des informations codées par des caractères alphanumériques »

Question 19

Écrire une fonction `prefixer : mot -> mots -> mots` telle que l'appel `prefixer m s` sur un mot `m` et une séquence de mots `s` renvoie une séquence de mots contenant les mêmes mots que `s` préfixés par le mot `m`. Par exemple, si la séquence `s` représente les mots (*faire*, *o*, *brouille*), et si `m` représente le mot *dé*, alors la séquence renvoyée représente la séquence (*défaire*, *déo*, *débrouille*).

Question 20

Quelle est la complexité de cette fonction ? Définir la constante `complexite_prefixer`, de type `int`, dont la valeur est le numéro de la réponse correcte.

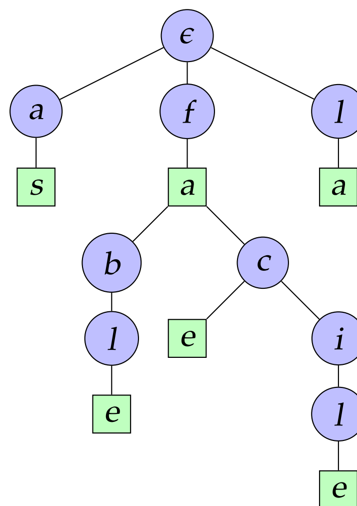
1 : $\Theta(1)$
2 : $\Theta(|m|)$
3 : $\Theta(|s|)$
4 : $\Theta(|s|\log(|m|))$
5 : $\Theta(|m|\log(|s|))$
6 : $\Theta(|m||s|)$
7 : $\Theta(|m|^{|s|})$
8 : Aucune des réponses ci-dessus.

IV.2 Arbres lexicographiques et arbres PATRICIA

Les arbres lexicographiques sont des arbres n-aires utilisés pour représenter des ensembles de mots. Les nœuds sont étiquetés par un caractère (sauf la racine qui est étiquetée par le mot vide ϵ). La séquence des caractères qui étiquettent les nœuds le long d'un chemin de la racine (exclue) de l'arbre jusqu'à un nœud (inclus) forme donc un mot. Un nœud peut être terminal si le mot qui conduit de la racine à ce nœud appartient à l'ensemble, ou non-terminal si ce mot n'est qu'un préfixe d'un mot de l'ensemble.

Exemple 2

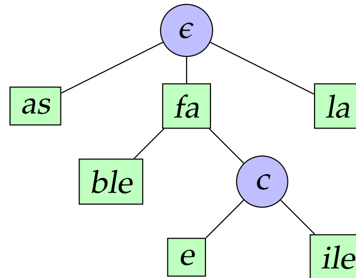
Voici un exemple d'arbre lexicographique représentant l'ensemble composé des mots : « as », « fa », « fable », « face », « facile » et « la ». Les nœuds terminaux sont représentés par un carré vert alors que les nœuds non-terminaux sont représentés par un cercle bleu.



Dans les arbres PATRICIA, les nœuds sont étiquetés par des mots au lieu de caractères. Les chemins ne contenant aucun branchement sont inutiles et sont donc remplacés par un unique nœud étiqueté par le mot composé des caractères qui étiqueteraient ce chemin.

Exemple 3

Voici un exemple d'arbre PATRICIA représentant le même ensemble que l'exemple précédent, composé des mots : « as », « fa », « fable », « face », « facile » et « la ». Les nœuds terminaux sont représentés par un carré vert alors que les nœuds non-terminaux sont représentés par un cercle bleu.



Définition IV.3

Un arbre PATRICIA a est composé de nœuds qui peuvent être terminaux ou non-terminaux. L'ensemble des nœuds de a est noté $\mathcal{N}(a)$, l'ensemble des nœuds terminaux est noté $\mathcal{T}(a)$. La racine de a est un nœud. Chaque nœud $n \in \mathcal{N}(a)$ est composé d'une séquence éventuellement vide de p fils $f_i \in \mathcal{N}(a)$ notée $\mathcal{F}(n) = (f_1, \ldots, f_n)$ avec $p \in \mathbb{N}$ et d'une séquence éventuellement vide de p étiquettes associées m_i notée $\mathcal{E}(n) = (m_1, \ldots, m_n)$ qui sont des mots non vides. Un arbre PATRICIA est valide si et seulement si :

- un nœud n qui ne possède pas de fils est terminal ;
- les étiquettes de chaque nœud doivent être distinctes et ordonnées de manière croissante selon la relation $\prec$ définie dans la question précédemment.

☞ **Remarque 1**

La racine de l'arbre est un nœud terminal si et seulement si le mot vide est dans l'ensemble de mots représentés.

Un arbre PATRICIA est représenté par les types mutuellement récursifs `patricia` et `fils`. Le type `fils` représente la séquence des fils et étiquettes associée à un nœud, c'est-à-dire une liste de paires contenant un élément de type mot et un élément de type `patricia`. On utilise donc le type CAML suivant :

OCAML
```
type patricia = Noeud of bool * fils
and fils = (mot * patricia) list;;
```

Dans l'appel `Noeud (term, fils)`, les paramètres `term` et `fils` sont respectivement un drapeau[2] booléen qui indique si le nœud est terminal ou non terminal et la liste des fils et étiquettes associées à ce nœud de l'arbre. Chaque élément de cette liste est une paire contenant d'une part le mot qui étiquette le lien entre le nœud et son fils, et d'autre part le fils.

Exemple 4

Vous trouverez dans votre programme l'expression qui définit l'arbre PATRICIA `ex_4 : patricia` *représenté graphiquement plus haut. Il ne faut pas pas modifier ni supprimer cet exemple dans votre code. N'hésitez pas à tester vos fonctions sur cet exemple.*

2. En informatique, un drapeau ou fanion (*flag* en Anglais) est une valeur binaire (de type `bool`) indiquant le résultat d'une opération ou le statut d'un objet.

Question 21

Écrire une fonction `creer : mot -> patricia` telle que l'appel `creer m` sur un mot `m` renvoie un arbre PATRICIA valide contenant uniquement le mot `m`.

Question 22

Écrire une fonction `compter : patricia -> int` telle que l'appel `compter a` sur un arbre PATRICIA valide `a` renvoie le nombre de mots contenus dans l'arbre `a`. L'algorithme utilisé ne devra parcourir qu'une seule fois l'arbre `a`.

Question 23

Écrire une fonction `extraire : patricia -> mots` telle que l'appel `extraire a` sur un arbre PATRICIA valide `a` renvoie une séquence de mots, croissante pour $\prec$, contenant les mêmes mots que l'arbre `a`. L'algorithme utilisé ne devra parcourir qu'une seule fois l'arbre `a`.

Question 24

Écrire une fonction `valider : patricia -> bool` telle que l'appel `valider a` sur un arbre PATRICIA `a` renvoie la valeur `true` si et seulement si l'arbre PATRCIA `a` est valide. L'algorithme utilisé ne devra parcourir qu'une seule fois l'arbre `a`.

Question 25

Écrire une fonction `accepter : mot -> patricia -> bool` telle que l'appel `accepter m a` sur un mot `m` et un arbre PATRICIA valide `a` renvoie la valeur `true` si l'arbre `a` contient le mot `m` et la valeur `false` sinon. L'algorithme utilisé ne devra parcourir que la branche de l'arbre `a` qui permet de représenter le mot (hors de question par exemple d'extraire tous les mots puis de faire une recherche dans cette liste).

Question 26

On ajoute les mots « facteur », « lampe » et « lame » dans l'ensemble de mots de l'exemple précédent. Définir une constante `taille_ex_4_avec_ajout` de type `int * int` qui renvoie le couple formé du nombre de nœuds terminaux et non-terminaux de l'arbre de PATRICIA qui représente cet ensemble.

Question 27

Écrire une fonction `ajouter : mot -> patricia -> patricia` telle que `ajouter m a` sur un mot `m` et un arbre PATRICIA valide `a` renvoie un arbre PATRICIA valide contenant le mot `m` et les mêmes mots que l'arbre `a`. L'algorithme utilisé ne devra parcourir qu'une seule fois l'arbre `a`. On pourra vérifiera cette fonction en ajoutant les mots « facteur », « lampe » et « lame » dans l'arbre PATRICIA `ex_4` par exemple.

Question 28

Écrire une fonction `fusionner : patricia -> patricia -> patricia` telle que l'appel `fusionner a1 a2` sur deux arbres PATRICIA valides `a1` et `a2` renvoie un arbre PATRICIA valide contenant exactement les mots contenus dans les arbres `a1` et `a2`. L'algorithme utilisé ne devra parcourir qu'une seule fois les arbres `a1` et `a2`.