Table des matières

Premier devoir surveillé

Première partie

Exercices de cours et TD

1. Écrire une fonction prenant en entrée une liste triée et renvoyant cette liste privée de ses doublons.
2. Écrire une fonction pour calculer l'argmax d'une liste d'association.
3. Écrire une fonction `incr_dico` qui prend un `'a * int hashtbl` et une clé c, et qui incrémente la valeur associée à c si il y en avait déjà une dans le dictionnaire, ou l'initialise à 1 si il n'y en avait pas.
4. En déduire une fonction `compte` qui prend en entrée un tableau `t` et qui renvoie un dictionnaire contenant, pour chaque élément de `t` le nombre de fois que cet élément est présent dans `t`.
5. Écrire une fonction prenant en entrée un arbre binaire de recherche a et une valeur x et renvoyant un arbre binaire de recherche contenant les éléments de a sauf x. Si x n'est pas présent dans a, renvoyer a tel quel.
6. Écrire une fonction `est_un_tas` prenant un arbre binaire et indiquant s'il s'agit d'un tas.

Deuxième partie

Déplacement d'un cavalier aux échecs

Le but de cet exercice est d'écrire une fonction prenant en entrée deux cases d'un échiquier et renvoyant un chemin qu'un cavalier puisse emprunter pour aller de l'un à l'autre, qui soit de plus le plus court possible.

1 Notations et fonctions préliminaires

Une case sera un couple d'entiers de $[\![0, 8[\![$. Un vecteur sera représenté par un couple d'entiers relatifs.

```
type case = int*int;;
type vecteur = int*int;;
```

On fournit une fonction pour renvoyer le résultat de la translation d'une case par un vecteur :

```
let translation (x,y) (dx,dy)=
  x+dx, y+dy
;;
```

Les déplacements possibles pour un cavalier aux échecs sont les translations selon les vecteurs suivant :

```
let deplacements = [
    (2,1); (2,-1); (-2,1); (-2,-1);
    (1,2); (1,-2); (-1,2); (-1,-2)
  ]
;;
```

La fonction `translation` ainsi que la variable globale `deplacements` sont librement utilisables dans ce problème.

1. Préciser le type de la fonction `déplacement`.
2. Écrire une fonction `est_dans_le_plateau` prenant en entrée un couple de coordonnées (x, y) et indiquant si ce sont les coordonnées d'une case du plateau.
3. Écrire une fonction `cases_accessibles` prenant en entrée une case c et renvoyant la liste des cases de l'échiquier accessibles depuis c par un déplacement de cavalier.

2 L'arbre sous-jacent

Pour toute case c, on définit l'arbre des déplacements possibles depuis c par récursivité ainsi :

- Sa racine comporte une étiquette qui vaut c.
- Elle a un fils pour chaque déplacement possible d depuis la case c, et ce fils est l'arbre des déplacement possible depuis la case `translation c d`.

1. Dessiner la racine, le premier niveau, et quelques sommet du deuxième niveau pour l'arbre des déplacements possible depuis la case $(1, 1)$.
2. Quel parcours de cet arbre utiliser pour répondre au problème ?

Cet arbre est utile pour raisonner mais ne sera pas explicitement construit dans Caml.

3 La fonction principale

Pour parcourir l'arbre des déplacements possibles, nous allons utiliser une file d'attente. Cette file contiendra des couples (`iti`, `case`), où `case` est une case de l'échiquier et `iti` est l'itinéraire emprunté pour y arriver. Ce sera une liste dont le premier élément est `case` et le dernier est le point de départ du cavalier.

Autrement dit, lorsque nous visitons un nœud n de l'arbre, l'élément correspondant dans la file d'attente est (la liste des ancêtre de n, l'étiquette de n).

On rappelle les fonctions de base pour manipuler les files d'attente de Caml :

- `Queue.create : unit -> Stack.t`
- `Queue.add : 'a -> 'a Stack.t -> unit`
- `Queue.take : 'a Stack.t -> 'a`
- `Queue.is_empty : 'a Stack -> bool`

1. Écrire une procédure `enfile_nelles_cases` prenant une case c et la liste *iti* de ses ancêtre, et une file d'attente, et qui a pour effet de rajouter dans la file d'attente tous les couples `(x : : iti, x)` pour `x` une case accessible depuis c.

2. On passe alors au programme final. L'idée est de parcourir l'arbre des déplacements possibles à l'aide de la file d'attente précédemment décrite. Lorsqu'on rencontre la case d'arrivée, le résultat est l'itinéraire qui lui était associé dans la file.

   Dans le cas où aucun itinéraire ne permet d'arrivée à la case d'arrivée, renvoyer la liste vide.

   Compléter le squelette de code suivant :

```
let trajet depart arrivee =
  (* Renvoie un itinéraire de longueur minimale pour déplacer le cavalier de la case
     ↪ départ vers la case arrivée.*)

  let aEssayer = Queue.create () (* File de couples (itinéraire, case) *)
  and fini = ref false
  and res = ref [] (* Contiendra le résultat en sortie de boucle *)
  Queue.add ([depart], depart) aEssayer;

  while not !fini && not (Queue.is_empty aEssayer) do
    let itineraire, case = Queue.take aEssayer in
    ...
  done;
  ...
;;
```

3. *Variante :* Il peut être plus agréable de remplacer la boucle while par une fonction récursive. Cela permet notamment de se passer des variables `fini` et `res`. Compléter le code suivant :

```
let trajet2 depart arrivee =
  (* Renvoie un itinéraire de longueur minimale pour déplacer le cavalier de la case
     ↪ départ vers la case arrivée.*)

  let aEssayer = Queue.create () in
  Queue.add ([depart], depart) aEssayer;

  let rec boucle () =
    if Queue.is_empty aEssayer then []
    else
      ...
  in
  boucle ()
;;
```

4 Version complètement récursive

Pour cette dernière variante, nous utilisons des files d'attente persistantes. On suppose disposer d'un type `'a file` muni des primitives suivantes :

- `fileVide : 'a file`;
- `enfiled : 'a -> 'a file -> 'a file`;
- `defiled : 'a file -> ('a * 'a file)`.

1. Écrire une fonction `nelles_cases_enfiled` de type `case -> case list -> (case list * case)file -> (case list * case)file` qui prend en entrée une case c, l'itinéraire y menant, une file d'attente f et qui renvoie la file d'attente obtenue en rajoutant dans f toutes les cases accessibles depuis c, chacune avec l'itinéraire correspondant.
2. Compléter alors le code suivant :

```
let trajet3 depart arrivee =


  let rec boucle aEssayer =
     ...
  in

```

```
  boucle (enfiled ([depart], depart) fileVide)
;;
```

5 Optimisation

Un défaut de l'algorithme précédent est que certaines cases peuvent être visitées plusieurs fois, ce qui est une perte de temps. Pour corriger ceci on propose de se souvenir au fur et à mesure des cases déjà visitées.
Remarque : Cette technique sera utilisée en permanence dans le chapitre sur les graphes.

5.1 Version mutable

Utilisons ici une table de hachage pour enregistrer les cases visitées. Cette table sera appelée `dejaVues`, ses clefs seront les cases visitées, et ses valeurs ne seront pas utilisées, mettons des `true` (pour utiliser un minimum de mémoire).

On rappelle les noms et types des fonctions élémentaires sur les tables de hachage :

- `Hashtbl.create : int -> ('a, 'b)Hashtbl.t`
- `Hashtbl.add : ('a, 'b)Hashtbl.t -> 'a -> 'b -> unit`
- `Hashtbl.mem : ('a, 'b)Hashtbl.t -> 'a -> bool`
- `Hashtbl.find : ('a, 'b)Hashtbl.t -> 'a -> 'b`

Lorsqu'on demande de modifier une fonction on pourra soit écrire entièrement la nouvelle fonction, soit donner précisément les modifications à apporter à l'ancienne.

1. Modifier la fonction `enfile_nelles_cases`. Elle prendra désormais un argument de plus : la table de hachage `dejaVues`. Seules les cases ne figurant pas dans celle-ci pourront être enfilées, et à chaque fois qu'une case sera enfilée, elle sera également insérée dans `dejaVues`.
2. Modifier alors la fonction `trajet1` ou `trajet2` au choix pour faire en sorte qu'une même case ne soit jamais traitées deux fois.

5.2 Version persistante

1. Quelle structure de donnée persistante vous paraît adaptée pour garder en mémoire les cases déjà vues ? On supposera disposer de ce type. Indiquer les noms et les types des fonctions élémentaires correspondantes. On ne demande pas de les programmer, mais vous pourrez les utiliser librement dans la suite.
2. Modifier la fonction `nelles_cases_enfiled`.
3. Modifier la fonction `trajet3`.

Troisième partie
Quart de singe

1 Introduction

Le quart de singe est un jeu de lettres nécessitant au moins deux joueurs. Dans la suite, notons n le nombre de joueurs. Chacun à son tour donne une lettre. À chaque instant, la suite des lettres données depuis le début de la partie doit être le début d'un mot français. Le premier joueur qui ne peut plus ajouter de lettre sans contrevenir à cette règle perd la partie.

Exemple d'une partie à deux joueurs :

- M. X dit « z »
- M. Y répond « e »
- M. X : « u »
- M. Y : « g »
- M. X : « m »

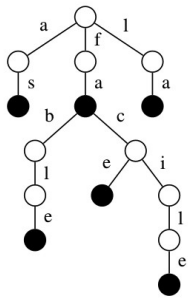
- M. Y : « e »

À ce moment, le mot formé est « zeugme »[1]. M. X peut encore le mettre au pluriel en ajoutant « s ». M. Y ne peut plus compléter le mot, il perd donc la manche.

Dans ce problème, nous utiliserons les arbres lexicographiques que nous avons découverts dans un devoir précédent.

```
#directory "/home/moi/enseignement/Informatique/bibs/";; (* Changer 'ladresse ici bien sûr *)
#use "arbreLex.ml";;
```

Rappelons-en succinctement le fonctionnement.



Chaque arête est étiquetée par une lettre. Chaque nœud est muni d'un booléen. Lorsque ce booléen est `true`, on dit que le nœud est « terminal ». Et dans ce cas, la suite des lettres étiquetant le chemin depuis la racine vers ce nœud est un des mots contenus par l'arbre. Dans l'arbre de l'exemple ci-contre, les nœuds terminaux sont noirs. Il contient les mots «as», «la», «fa», «fable», «face», et «facile».

Le type utilisé est celui-ci :

```
type arbreLex = Noeud of bool * fils
and fils = (char*arbreLex) list;;
```

Un mot sera ici une liste de caractères :

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

Une constante `feuille` est définie en cas de besoin :

```
let feuille= Noeud(true,[]);;
```

L'exemple ci-dessous :

```
let exemple=Noeud(false,[
                  'f',Noeud(false,[
                            'a', Noeud(true,[
                                      ('c', feuille)
                                    ]
                                   )
                          ;
                            ('i', feuille)
                        ])
              ]);;
```

est un arbre lexicographique contenant les mots « fa », « fac » et « fi ».

On donne à titre d'exemple une fonction permettant de tester si un mot appartient à un arbre :

```
let rec appartient (m:mot) a =
    (* Indique si le mot m est contenu dans l'arbre a. *)
    match m, a with
    |[], Noeud(term,_) -> term (* Le mot vide est reconnu ssi la racine est terminale *)
    |t::q, Noeud(_,fils)-> appartient_foret t q fils

and appartient_foret lettre mot f =
   match f with
   |[] -> false
   |(x,a)::q when x=lettre -> appartient mot a
```

1. Figure de style. Exemple de zeugme : « Elle entra chez les Carmélites, par dépit et la porte de devant. » in « Cécile », Lutin bleu

```
    |(x,_)::q when x<lettre -> appartient_foret lettre mot q
    |_ -> false
;;
```

Enfin, on suppose que tous les mots du français ont été chargés dans un arbre lié à l'identifiant global `dico_francais`.

2 Sous-arbres

1. Écrire une fonction `bon_fils` prenant une lettre x et un arbre lexicographique a et renvoyant le fils de la racine de a atteint en empruntant l'arête étiquetée par x. Si aucune arête n'est étiquetée par x, lever une erreur.
2. Écrire une fonction `sous_arbre` prenant un mot m et un arbre lexicographique a et renvoyant le sous-arbre de a atteint en suivant depuis la racine le chemin indiqué par les lettres de m.
   Ainsi `sous_arbre m dico_francais` et l'arbre contenant toutes les chaînes de caractères qui, rajoutées derrière m, forment un mot français. Autrement dit toutes les chaînes de caractères pouvant encore être jouées si le mot formé par les premières lettres lors d'une partie de quart de singe est m.
   Remarque : Cette dernière fonction ne sera pas utilisée dans la suite.

3 Nombre de feuilles

1. Pourquoi la notion de feuille est-elle cruciale lors d'une partie de quart de singe ?
   solution : Lorsqu'on aboutit à une feuille, il n'est plus possible de rajouter une lettre, et le joueur dont c'est le tour perd la partie.
2. Écrire une fonction permettant de compter les feuilles d'un arbre lexicographique.
   Indication : `feuille` n'est pas un constructeur. En revanche, `Noeud(true, [])` est un motif valide qui reconnaît les feuilles.

   solution :

```
let rec nb_feuilles  a =
  match a with
    | Noeud(_,[])  -> 1
    | Noeud(_,fils)-> nb_feuilles_foret fils

and nb_feuilles_foret lf =
    match lf with
    |[]   -> 0
    |(_,a)::q  -> nb_feuilles a + nb_feuilles_foret q
;;
```

3. Soit a l'arbre des chaînes de caractères pouvant encore être jouées à un moment de la partie. Soit j le joueur dont c'est le tour. On rappelle que le nombre de joueurs est noté n. En quoi les feuilles de a de profondeur divisible par n sont-elles importantes pour j ?
   solution : Si on aboutit sur une de ces feuilles dans la suite de la partie est perdue pour j. En revanche, si on aboutit sur une feuille de profondeur non divisible par n, c'est un autre joueur qui perd.
4. Écrire une fonction `nb_feuilles_a_bonne_prof` prenant un arbre a et un entier n et renvoyant le nombre de feuilles de a dont la profondeur n'est pas divisible par n.
   Indication : Le plus simple est peut être d'écrire une fonction auxiliaire prenant en argument supplémentaire p la profondeur du nœud actuel modulo n.

5. Une stratégie basique est alors de choisir dans l'arbre des chaînes de caractère restantes le fils qui contient un maximum de feuilles de profondeur non divisible par n.
   Programmer une fonction `meilleur_fils` correspondante. Elle prendra en arguments un entier n et un arbre a et renverra le triplet (nf, f, x) où f est le fils de a ayant le maximum de feuilles de profondeur non divisible par n, nf est ce nombre de feuilles, et x est la lettre menant à f depuis la racine de a.

4 Simulation d'une partie

Dans la suite du problème, on suppose que $n = 2$. On appellera 1 et -1 les deux joueurs de sorte que si un joueur est j, l'autre joueur est $-j$.

Dans cette partie on propose de programmer une partie d'un joueur humain contre un ordinateur. La fonction `read_line` de type `unit -> string` interrompt l'exécution du programme pour récupérer la prochaine ligne tapée par l'utilisateur. Ainsi `let lettre = (read_line()).[0] in` permet de récupérer un caractère rentré par l'utilisateur.

Par ailleurs pour afficher une lettre, on peut utiliser `print_char`.

Programmer alors une partie entre un joueur et un ordinateur. L'ordinateur choisira la lettre jouée au moyen de `meilleur_fils`. Le premier joueur pour lequel l'arbre des chaînes restantes est une feuille perd.

5 Minimax

L'algorithme du minimax est l'algorithme classique pour jouer un jeu à deux joueurs au tour par tour.

Nous fixons un entier p, et l'ordinateur va essayer de prévoir les actions de son adversaire à p tours à l'avance. Pour ce, il va supposer que son adversaire utilise le même algorithme que lui-même. Si $p = 0$, il utilise une stratégie naïve semblable à la précédente.

Le programme repose sur une fonction `score` qui prend l'arbre a des chaînes de caractères restantes, le joueur actuel j et le paramètre p, et qui renvoie un entier qui doit mesurer les chances qu'a le joueur j de gagner. On fera en sorte que le score de joueur j sera toujours l'opposé de celui du joueur $-j$.

Ce score sera calculé ainsi :

- Si $p = 0$ on prend `nb_feuilles_a_bonne_prof a n` comme à la partie précédente.
- Si a est vide, on renvoie `min_int`, le plus entier possible puisque dans ce cas j a perdu.
- Dans le cas général, on calcule le score de $-j$ pour toutes les lettres qu'il est possible de jouer (c'est-à-dire tous les fils de a). D'une part on va jouer la lettre faisant en sorte que le score de $-j$ soit le plus petit possible, et d'autre part le score de $-j$ est l'opposé du score de j. Il faut donc renvoyer l'opposé du minimum des scores de $-j$ pour tous les fils de a.

  Dans les appels récursifs, on enverra comme paramètre de profondeur $p - 1$ au lieu de p afin d'assurer la terminaison.

1. Programmer la fonction `score`.
2. En déduire une fonction `meilleur_fils_minimax` qui prend l'arbre a des chaînes de caractères restantes, le joueur actuel j et le paramètre p et qui renvoie le couple (lettre à jouer, fils correspondant). La lettre à jouer sera celle qui mène au plus grand score pour le joueur j.