Examen - Session 1

Janvier 2022

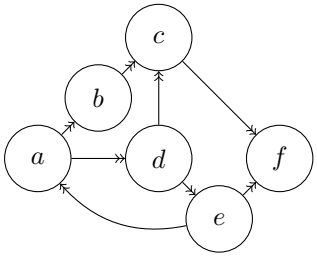
Documents autorisés: poly et notes de cours, notes de TD

L'épreuve durera 1 heure 30. Le sujet vaut 26 points.

La correction tiendra compte du respect de la syntaxe (par ex.: parenthèsage de l'application) ou de l'inutile complexité (par ex.: écrire `if e then true else false` au lieu de `e`).

Exercice I : Listes de couples

On représente un *réseau* par une liste de couples. Par exemple, le réseau de la figure ci-dessous :



est représenté par la liste de couples `ex` définie par :

```
let ex = [('a','b');('a','d');('b','c');('d','c');('d','e');('e','f');('c','f');('e','a')];;
```

Dans la suite, le mot *«liaison»* désigne un couple et le mot *«réseau»* désigne une liste de couples.

Remarque: les listes construites par les fonctions de cet exercice peuvent contenir plusieurs fois les mêmes éléments et l'ordre n'a pas d'importance.

Q1 – (2pts) Si (x, y) est une liaison du réseau r on dit que *y est accessible dans le réseau r par un chemin de longueur 1 à partir du point x.*

Définir la fonction de signature

`access1 (x:'a) (r:('a*'a) list) : 'a list`

qui construit la liste des points accessibles dans le réseau `r` par un chemin de longueur 1 à partir du point `x`.

Exemples:

```
# (access1 'f' ex);;
- : char list = []
# (access1 'b' ex);;
- : char list = ['c']
# (access1 'a' ex);;
- : char list = ['b'; 'd']
```

Q2 – (2pts) Définir la fonction de signature

`list_access1 (xs:'a list) (r:('a*'a) list) : 'a list`

qui construit la liste des points accessibles dans le réseau `r` par un chemin de longueur 1 à partir des points présents dans la liste `xs`.

Exemple:

```
# (list_access1 ['a';'b'] ex);;
- : char list = ['b'; 'd'; 'c']
```

Remarque: `['b'; 'd'; 'c']` = `['b';'d']@['c']`.

Q3 – (1pt) Les deux couples (x, y) et (y, z) forment un *chemin de longueur 2*. On dit que z *est accessible par un chemin de longueur 2 à partir de* x.

En combinant les fonctions ci-dessus, définir la fonction de signature

`access2 (x:'a) (r:('a*'a) list) : 'a list`

qui construit la liste des points accessibles dans le réseau `r` par un chemin de longueur 2 à partir du point `x`.

Exemples:

```
# (access2 'b' ex);;
- : char list = ['f']
# (access2 'a' ex);;
- : char list = ['c'; 'c'; 'e']
```

Q4 – (3pts) Un point z est accessible par un chemin de longueur n à partir d'un point x si z est accessible par un chemin de longueur 1 à partir d'un point y présent dans la liste des points accessibles par un chemin de longueur $n-1$ à partir de x.

Définir la fonction de signature

`accessn (x:'a) (n:int) (r:('a*'a) list) : 'a list`

qui construit la liste des points accessibles dans le réseau `r` par un chemin de longueur `n` à partir du point `x`. Par convention, `(list_accessn x n r)` est la liste vide si `n` < 0 et `(list_accessn x 0 r)` est égale à `[x]`.

```
# (accessn 'a' 0 ex);;
- : char list = ['a']
# (accessn 'a' 1 ex);;
- : char list = ['b'; 'd']
# (accessn 'a' 2 ex);;
- : char list = ['c'; 'c'; 'e']
# (accessn 'a' 3 ex);;
- : char list = ['f'; 'f'; 'f'; 'a']
# (accessn 'a' 4 ex);;
- : char list = ['b'; 'd']
```

Q5 – (3pts) Définir une fonction de signature

`access_infn (x:'a) (n:int) (r:('a*'a) list) : 'a list`

qui construit la liste des points accessibles dans le réseau `r` par un chemin de longueur strictement positive et inférieure ou égale à `n` à partir du point `x`. La fonction déclenche l'exception `Invalid_argument "list_access_infn"` si `n` est négatif ou nul.

```
# (access_infn 'a' 2 ex);;
- : char list = ['c'; 'c'; 'e'; 'b'; 'd']
# (access_infn 'a' 3 ex);;
- : char list = ['f'; 'f'; 'f'; 'a';'c'; 'c'; 'e'; 'b'; 'd']
# (access_infn 'd' 2 ex);;
- : char list = ['f'; 'f'; 'a'; 'c'; 'e']
```

Q6 – (2pts) Si x est accessible dans le réseau r par un chemin de longueur strictement positive à partir de x (lui-même) on dit qu'il existe un *cycle dans le réseau* r *à partir du point* x. C'est-à-dire que x figure dans la liste des points accessibles à partir de x lui-même.

La longueur maximale d'un chemin du réseau ne passant pas deux fois par le même point est toujours inférieure ou égale

au nombre de liaisons du réseau. Donc, *il n'y a pas* de cycle dans le réseau à partir de x si x *ne figure pas* dans la liste des points accessibles par un chemin de longueur inférieure ou égale au nombre de liaisons à partir de x.

En déduire une définition de la fonction de signature

`cycle_from (x:'a) (r:('a*'a) list) : bool`

qui détermine s'il existe un cycle dans le réseau `r` à partir du point `x`.

```
# (cycle_from 'a' ex);;
- : bool = true
# (cycle_from 'b' ex);;
- : bool = false
```

On obtient facilement une définition de `cycle_from` avec les fonctions `List.mem` et `List.length` de la bibliothèque standard.

Solution Exercice I

```
(* -- Q1 *)
let rec access1 (s:'a) (l:('a*'a) list) : 'a list =
match l with
   [] -> []
 | (x,y)::t -> if s=x then y::(access1 s t)
                      else (access1 s t);;

(* -- Q2 *)
let rec list_access1 (ls:'a list) (l:('a*'a) list) : 'a list =
match ls with
   [] -> []
 | h::t -> (access1 h l) @ (list_access1 t l);;

(* -- Q3 *)
let acess2 (s:'a) (l:('a*'a) list) : 'a list =
 (list_access1 (access1 s l) l);;

(* -- Q4 *)
let rec accessn  (s:'a) (n:int) (l:('a*'a) list) : 'a list =
  if (n<0) then []
  else if (n=0) then [s]
  else if (n=1) then (access1 s l)
  else (list_access1 (accessn s (n-1) l) l);;

(* version recursive terminale *)
let accessn (s:'a) (n:int) (l:('a*'a) list) : 'a list =
  let rec loop n r =
    if (n=0) then r
    else loop (n-1) (list_access1 r l)
  in
    if (n < 0) then []
    else (loop n [s])

(* -- Q5 *)
let rec access_infn (s:'a) (n:int) (l:('a*'a) list) : 'a list =
if n=0 then raise (Invalid_argument "bad access_infn")
else if n=1
     then (access1 s l)
     else (accessn s n l) @ (access_infn s (n-1) l);;

(access_infn 'a' 2 ex);;
(access_infn 'a' 3 ex);;
(access_infn 'd' 2 ex);;
```

```
(* -- Q6 *)
let cycle_from (s:'a) ((l:('a*'a) list)) : bool =
(List.mem s (access_infn s (List.length l) l));;
```

Exercice II : Arbres binaires et mots binaires

On utilise les structures d'arbres binaires pour représenter des *ensembles de mots binaires*. La représentation repose sur un principe analogue à celui utilisé pour les codes de Huffman.

On définit les deux symboles binaires avec le type `type bin = I | O`. Un *mot binaire* est une liste de type `(bin list)`. Le mot binaire 1001 est représenté par la liste `[I; O; O; I]`.

Dans un arbre binaire, on n'utilise pas les symboles `I` et `O` mais on fait correspondre un branchement à gauche avec le symbole binaire `I` et un branchement à droite avec le symbole `O`.

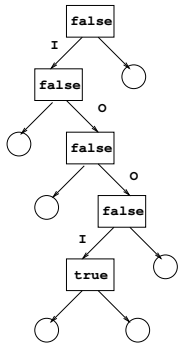
Le type des arbres binaires utilisé est celui vu en cours:

```
type 'a btree =
    Empty
  | Node of 'a * 'a btree * 'a btree
```

Les étiquettes seront des booléens qui servent à marquer si un mot est fini ou non. Un ensemble de mots binaires est donc représenté par un arbre de type `(bool btree)`.

Mot et arbre Commençons par repésenter un unique mot par un arbre.

Par exemple, le mot binaire `[I; O; O; I]` est représenté par l'arbre ci-contre. Le mot est lu de gauche à droite.
Les rectangles sont les nœuds (constructeur `Node`) et les ronds, l'arbre vide (constructeur `Empty`).
Les symboles binaires sont indiqués sur la figure pour information.
Valeur OCAML (type `(bool btree)`):



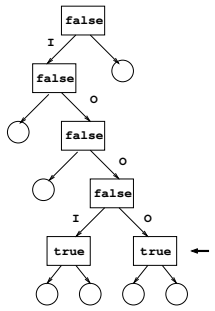

```
Node(false,
     Node (false, Empty,
                  Node (false, Empty,
                               Node (false, Node (true, Empty, Empty),
                                            Empty))),
     Empty)
```

Q1 – (2pts) Définir la fonction `btree_of_bin` qui prend en argument un mot binaire (liste de valeurs de `O` et de `I`) et qui construit l'arbre correspondant.

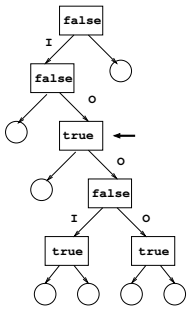
Attention: `(btree_of_bin [])` est égal à `Node(true, Empty, Empty)`.

Ensemble de mots et arbre La représentation d'un seul mot binaire en arbre n'est pas très économique, mais cela change quand on a des ensembles de mots. En effet, la représentation en arbre permet de partager les préfixes communs des mots.

Par exemple, si l'on veut construire l'ensemble contenant les mots `[I;O;O;I]`, `[I;O;O;O]` il n'y a qu'un seul nœud à ajouter à l'arbre qui représente `[I;O;O;I]`



Mieux, si on veut ajouter le mot `[I;O]` à cet ensemble, il n'y a pas de nœud à ajouter; il suffit de modifier une étiquette pour marquer la fin de `[I;O]`.



On peut «suivre» les branches de l'arbre en fonction d'un mot (`bin list`): «descendre» à gauche si le mot commence par `I`; «descendre» à droite si le mot commence par `O` et poursuivre avec le reste du mot. Lorsque le mot est épuisé, si l'étiquette du sous-arbre est `true`, le mot appartient à l'ensemble représenté par l'arbre; sinon, non.

Q2 – (3pts) Définir la fonction de signature

```
appartient (m:bin list) (bt:bool btree) : bool
```

qui renvoie `true` si `m` appartient à l'ensemble de mots binaires représenté par l'arbre `bt`.

Q3 – (4pts) Définir la fonction de signature

```
ajout (m:bin list) (bt:bool btree) : bool btree
```

qui renvoie l'ensemble obtenu par l'ajout de `m` à `bt`.

Attention: si `bt` est `Empty` alors le résulat est l'arbre qui représente le mot `m` (voir fonction `btree_of_bin`).

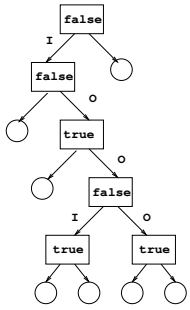
Q4 – (4pts) Définir la fonction de signature

```
retrait (m:bin list) (bt:bool btree) : (bool btree)
```

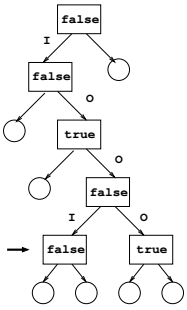
qui renvoie l'ensemble de mots binaires (sous forme d'arbre) en supprimant `m` de `bt`.

Exemples:

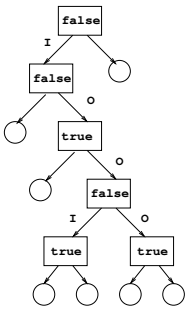
`retrait` appliquée au mot `[I;O;O;I]` et à l'arbre



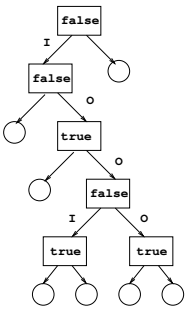
construit l'arbre



`retrait` appliquée au mot `[I;O;O]` et à l'arbre



construit le même arbre



Solution Exercice II

```
(* -- Q1 *)
let rec btree_of_bin (m:bin list) : bool btree =
  match m with
      [] -> Node(true, Empty, Empty)
    | I::m -> Node(false, btree_of_bin m, Empty)
    | O::m -> Node(false, Empty, btree_of_bin m)

(* -- Q2 *)
let rec appartient (m:bin list) (bt:bool btree) : bool =
  match m, bt with
      _, Empty -> false
    | [], Node(x,_,_) -> x
    | I::m, Node(_,bt1,_) -> appartient m bt1
    | O::m, Node(_,_,bt2) -> appartient m bt2

(* -- Q3 *)
let rec ajout (m:bin list) (bt:bool btree) : bool btree =
  match bt with
      Empty -> btree_of_bin m
    | Node(x,bt1,bt2) -> (
match m with
    [] -> Node(true, bt1, bt2)
  | I::m -> Node(x, ajout m bt1, bt2)
  | O::m -> Node(x, bt1, ajout m bt2)
      )

(* -- Q4 *)
let rec retrait (m:bin list) (bt:bool btree) : (bool btree) =
  match bt with
      Empty -> Empty
    | Node(x,bt1,bt2) -> (
match m with
    [] -> Node(false,bt1,bt2)
  | I::m -> Node(x, retrait m bt1, bt2)
  | O::m -> Node(x, bt1, retrait m bt2)
      )
```