Devoir surveillé n°1

Durée : 2 heures, calculatrices et documents interdits

Dans tout le devoir, vous pourrez à tout moment utiliser une fonction d'une question précédente, même si vous ne l'avez pas écrite.

1 Exercice 1

Définir une fonction de type `bool -> bool -> bool` représentant l'opérateur logique $\Longrightarrow$.

2 Exercice 2

Dans cet exercice, on construit en OCaml un type nombre représentant un entier ou un flottant :

[1]:
```
type nombre =
    |Entier of int
    |Reel of float;;
```

[1]:
```
type nombre = Entier of int | Reel of float
```

1) Écrire une fonction `addition : nombre -> nombre -> nombre` qui à deux objets de type `nombre` renverra leur somme, également de type `nombre`. Ainsi, `addition (Entier 7) (Reel 9.1)` renverra `Reel 16.1`, et `addition (Entier 7) (Entier 9)` renverra `Entier 16`.
2) Écrire une fonction `multiplication : nombre -> nombre -> nombre` qui à deux objets de type `nombre` renverra leur produit, également de type `nombre`.
3) Écrire `est_nul : nombre -> bool` qui retourne le booléen `true` si et seulement l'entrée est un nombre représentant le flottant `0.` ou l'entier `0`. Par exemple, `est_nul (Entier 17)` doit retourner `false`.
4) En déduire une fonction `division : nombre -> nombre -> nombre` qui réalise la division de deux nombres (le 1er argument divisé par le second). Cette fonction doit renvoyer un message d'erreur en cas de division par 0.

3 Exercice 3 : `takewhile` et `dropwhile` (d'après B. Petazzoni)

On appelle *préfixe* d'une liste l toute liste l_1 telle qu'il existe une liste l_2 vérifiant $l_1@l_2 = l$, où @ est l'opérateur de concaténation. Ainsi, si $l = [1;2;3;4;5]$, $[1;2;3]$ est un préfixe de l, mais pas $[3;4;5]$, ni $[1;2;4]$.

De même on apellera *suffixe* d'une liste l toute liste l_1 telle qu'il existe une liste l_2 vérifiant $l_2@l_1 = l$. Ainsi, si $l = [1;2;3;4;5]$, $[3;4;5]$ est un suffixe de l, mais pas $[3;4]$, ni $[2;4;5]$.

On considérera dans cet exercice des *prédicats* p sur les entiers, c'est-à-dire des fonctions OCaml de type `int -> bool`. Par exemple la fonction `positif` suivante est un prédicat sur les entiers :

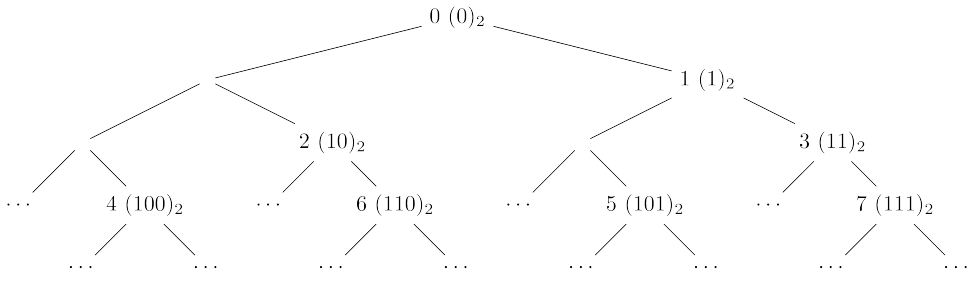
[2]:
```
let positif x = x>=0 ;;
```

[2]:
```
val positif : int -> bool = <fun>
```

1) Écrire en OCaml une fonction `negation : (int -> bool) -> (int -> bool)` qui prend en argument un prédicat p et renvoie le prédicat "non p".
2) Écrire en OCaml une fonction récursive `takewhile : (int -> bool) -> int list -> int list` prenant en argument un prédicat p sur les entiers et une liste l d'entiers, et renvoyant le plus long préfixe de l formé d'éléments vérifiant le prédicat p. Par exemple `takewhile positif [5;2;3;-1;-2;8]`renverra `[5;2;3]` alors que `takewhile positif [-5;2;3;-1;-2;8]`renverra `[]`.
3) Écrire en OCaml une fonction récursive `dropwhile : (int -> bool) -> int list -> int list` prenant en argument un prédicat p sur les entiers et une liste l d'entiers, et renvoyant la liste l privée du plus long préfixe de l formé d'éléments vérifiant le prédicat p. Par exemple `takewhile positif [5;2;3;-1;-2;8]`renverra `[-1;-2;8]` alors que `takewhile positif [-5;2;3;-1;-2;8]`renverra `[-5;2;3;-1;-2;8]`.
4) On suppose disposer d'une fonction `miroir : int list -> int list` qui prend en argument une liste d'entiers et renvoie sa liste miroir. Ainsi `miroir [1;2;3]` renvoie `[3;2;1]`. En utilisant la fonction `miroir`, écrire en OCaml une fonction `suffixe1 : (int -> bool) -> int list -> int list` prenant en argument un prédicat p sur les entiers et une liste l d'entiers, et renvoyant le plus long suffixe de l formé d'éléments vérifiant le prédicat p. Par exemple `takewhile positif [5;2;3;-1;-2;8]`renverra `[8]` alors que `takewhile positif [5;2;3;-1;-2-8]`renverra `[]`.
5) Écrire en OCaml une fonction récursive `suffixe2 : (int -> bool) -> int list -> int list` effectuant la même chose que `suffixe1`, mais qui utilisera uniquement les fonctions `negation`, `takewhile` et `dropwhile`, et pas la fonction `miroir`.

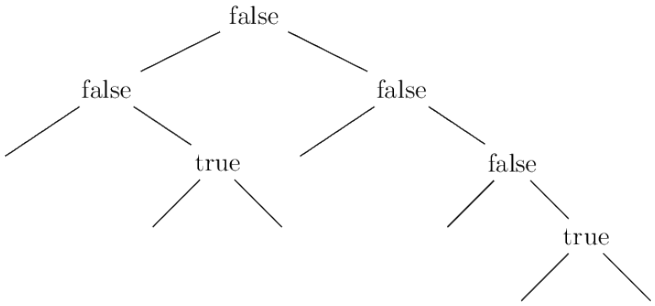
4 Exercice 4 : arbres radix (d'après J.-P. Becirspahic)

On se propose de décrire une structure de données pour représenter les ensembles finis d'entiers positifs ou nuls appelée arbres radix (radix tree en anglais). L'idée consiste en une structure arborescente basée sur l'écriture binaire de l'entier : si l'entier vaut 0 il est stocké à la racine ; sinon on le stocke dans le sous-arbre gauche si son dernier chiffre binaire est 0 (donc s'il est pair), et dans le sous-arbre droit si son dernier chiffre binaire est 1 (s'il est impair), et l'on poursuit avec les chiffres suivants. À titre indicatif, voici où se retrouvent placés les huit premiers entiers (leur écriture binaire est indiquée entre parenthèses) :



Notez que les chiffres de l'écriture binaire sont utilisés de droite à gauche (du moins significatif au plus significatif). On choisit de coder les arbres radix en Caml par des arbres avec des feuilles vides et des noeuds contenant un booléen indiquant la présence ou l'absence de l'entier correspondant dans l'ensemble représenté par l'arbre. On maintiendra en outre l'invariant suivant : l'arbre ne contient aucun noeud étiqueté par false dont les deux sous-arbres sont vides, c'est à dire de la forme :

L'ensemble $\{2; 7\}$ sera donc représenté par l'arbre suivant :



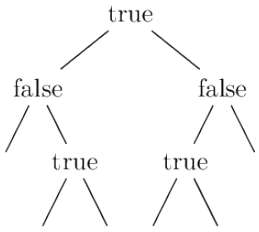
Par la suite, on conviendra d'identifier un ensemble d'entiers avec l'arbre radix qui le représente. On définit enfin le type :

[3]:
```
type arbre =
    | Nil
    | Noeud of (bool * arbre * arbre)
;;
```

[3]:
```
type arbre = Nil | Noeud of (bool * arbre * arbre)
```

Les questions sont un peu plus difficiles à partir de la question 8 ; la question 11 est de loin la plus compliquée ; pour chaque question, vous pourrez utiliser les fonctions des questions précédentes, même si vous ne les avez pas écrites.

1) Dessiner l'arbre radix représentant l'ensemble $\{2; 3; 5; 7; 11\}$.
2) L'arbre suivant est-il un arbre radix ? Énoncer (en français) un critère simple pour un élément de type arbre soit un arbre radix. On supposera désormais cette condition toujours vérifiée.



3) Soit n un entier. Si sa décomposition en base 2 est $n = \sum_{k=0}^{p} b_k 2^k$, alors la liste $[b_p; b_{p-1}; \cdots; b_1; b_0]$ représente la décomposition en base 2 de n. Que vaut alors le reste de la division euclidienne de n par 2 ? Et quelle liste représente la décomposition en base 2 du quotient de cette même division euclidienne ?
4) On souhaire écrire en OCaml une fonction `cherche : int -> arbre -> bool` qui prend en arguments un entier n et un ensemble E (représenté par un arbre radix) et qui renvoie un booléen déterminant l'appartenance de n à E. Compléter la fonction suivante à cet effet (on rappelle que si n est de type `int`, alors `n / 2` et `n mod 2` calculent respectivement le quotient et le reste de la division euclidienne de n par 2) :

[ ]:
```
let rec cherche n arbre =
    match arbre with
        | Nil  (* cas de l'ensemble vide *) ->
        | Noeud (b, fg, fd) when n = 0 ->
        | Noeud (b, fg, fd) when n mod 2 = 0 ->
        | Noeud (b, fg, fd) ->
;;
```

5) Écrire en OCaml une fonction `ajoute : int -> arbre -> arbre` qui prend en arguments un entier n et un ensemble E et qui retourne l'ensemble $E \cup \{x\}$.
6) En utilisant la fonction obtenue à la question précédente, écrire en OCaml une fonction `construit : int list -> arbre` construisant un ensemble à partir de la liste des entiers qu'il contient.
7) Écrire en OCaml une fonction `union : arbre -> arbre -> arbre` réalisant l'union de deux ensembles.
8) Écrire en OCaml une fonction `elague : arbre -> arbre` qui supprime dans un arbre radix les branches mortes, c'est-à-dire les branches de la forme

9) Écrire en OCaml une fonction `intersection : arbre -> arbre -> arbre` réalisant l'intersection de deux ensembles.
10) Écrire en OCaml une fonction `supprime : int -> arbre -> arbre` prenant en arguments un entier n et un ensemble E et retournant l'ensemble $E \backslash \{n\}$ (si n n'appartient pas à E, cette fonction ne doit pas échouer mais retourner le même ensemble E). On prendra bien soin de retourner un arbre sans branche morte.
11) Rédiger enfin en OCaml une fonction `elements : arbre -> int list` retournant la liste de tous les éléments d'un ensemble (l'ordre des éléments importe peu, mais chaque élément doit apparaître une fois et une seule dans la liste). On pourra utiliser l'opérateur `@`, permettant de concaténer deux listes.
12) Que pouvez-vous dire de deux arbres radix contenant exactement les mêmes éléments ? En déduire une fonction OCaml `egale : list -> list -> bool` prenant en arguments deux listes l_1 et l_2 et renvoyant `true` si les ensembles $\{x\ ,\ x \in l_1\}$ et $\{x\ ,\ x \in l_2\}$ sont égaux, `false` sinon.