TP11: Arbres binaires de recherche

MP2I Lycée Pierre de Fermat

Ce TP est à rendre sur Cahier de Prépa pour mardi 22h00. N'oubliez pas de commenter vos fonctions afin de les documenter, et de fournir des tests permettant de vérifier vos fonctions.

Type option et gestion d'erreurs

Type option

OCaml est muni par défaut d'un type appelé le **type option**. Il sert à rajouter un élément additionnel à n'importe quel type, cet élément signifiant "pas de valeur". On peut voir cet élément comme le None de python.

Le type est défini comme suit:

```
type 'a option =
  | Some 'a
  | None
;;
```

On utilise par exemple ce type pour simplifier les fonctions de recherche qui peuvent ne pas aboutir, comme une alternative aux failwith. Par exemple, la fonction suivante sert à chercher dans une liste un élément satisfaisant une fonction booléenne donnée en argument:

```
(* Renvoie le premier élément de l satisfaisant f *)
let rec satisfait_opt: (f: 'a -> bool) (l: 'a list) : ('a option) =
  match l with
  | [] -> None
  | x::q -> if f x then Some x else satisfait_opt f q
;;

satisfait (fun x -> x mod 5 = 0) [13;14;15;16;17] ;; (* vaut Some 15 *)

satisfait (fun x -> x mod 5 = 0) [11;12;13;14] ;; (* vaut None *)
```

Si l'on veut ensuite obtenir une fonction renvoyant un élément et pas une option, on peut rajouter une couche en utilisant `failwith`:

```
(* Renvoie le premier élément de l satisfaisant l. Lève une
   exception si l ne contient aucun tel élément. *)
let satisfait f l =
  match satisfait_opt f l with
  | None -> failwith "Aucun élément ne correspond"
  | Some x -> x
;;
```

Question 1. Écrire une fonction `division_opt: int -> int -> int option` effectuant une division, et renvoyant `None` en cas de division par zéro.

Question 2. Écrire une fonction `max_opt: 'a list -> 'a option` renvoyant le maximum d'une liste, et renvoyant `None` si la liste est vide.

Question 3. Écrire une fonction `option_map: ('a -> 'b)-> 'a option -> 'b option` permettant d'appliquer une fonction à une option.

Exceptions

Les **exceptions** sont des objets particuliers en OCaml qui servent à indiquer qu'un évènement imprévu a eu lieu. Par exemple, les exceptions apparaissent lorsqu'on essaie de diviser par zéro, lorsque l'on `failwith`, ou lorsqu'une assertion n'est pas vérifiée:

```
let x = 1/0 ;; (* Exception: Division_by_zero *)
let y = failwith "erreur" ;; (* Exception: Failure "erreur" *)
let z = assert (1+1 = 3) ;; (* Exception: Assert_failure ("//toplevel//", 1, 8). *)
```

Les exceptions ne sont pas des erreurs ! En effet, OCaml propose un mécanisme permettant de **rattraper une exception**. Pour cela, on utilise les mots clés **try** et **with**, comme suit:

```
let division_safe (x:int) (y:int) : int option =
  try Some (x/y) with
    Division_by_zero -> None
;;
```

Les exceptions font toutes parties du même type, `exn`, qui est donc défini comme suit:

```
type exn =
  | Division_by_zero
  | Assert_ailure of string * int * int (* nom du fichier, numéro de ligne, numéro de colonne *)
  | Failure of string
  ...
;;
```

De plus, c'est un type **extensible**, ce qui signifie que l'on peut y rajouter des constructeurs, autrement dit créer ses propres exception. On peut également lever des exceptions dans nos programmes avec la fonction `raise: exn -> 'a`, par exemple:

```
exception PasDeNombrePair of int ;;

let ajouter x y =
  if (x+y) mod 2 = 0 then raise (PasDeNombrePair (x+y))
  else x+y
;;

let a = try ajouter 1 3 with
  | PasDeNombrePair -> 1
;;
```

Reprenons la partie du TP précédente sur les expressions arithmétiques, en utilisant ces notions. Vous trouverez sur Cahier de Prépa un fichier "expr.ml" contenant du code de départ, que vous devrez adapter en répondant aux questions suivantes.

Question 4. Créer une exception permettant de signaler qu'une variable ne se trouve pas dans le contexte, et modifiez la fonction de recherche de variable en conséquence. Cette fonction lèvera donc l'exception créée en cas de variable absente.

Question 5. Écrire une fonction `add_opt : int option -> int option -> int option` permettant de faire la somme de deux options d'entiers. Si au moins l'une des deux est `None`, alors le résultat est `None`.

Question 6. Écrire une fonction `binop_opt: ('a -> 'a -> 'a)-> 'a opt -> 'a opt -> 'a opt` permettant d'appliquer une fonction à deux arguments sur des options. Par exemple, `binop_opt (+)` sera équivalente à la fonction `add_opt` précédente.

Question 7. Modifier les fonctions d'évaluation pour qu'elles renvoient des options plutôt que des valeurs, de telle sorte qu'elles ne puissent pas lever d'exceptions et renvoient `None` en cas de variable manquante.

Arbres binaires de recherche

On implémente dans cette partie des ABR dont les informations sont stockées sur les noeuds. Plutôt que d'implémenter la structure d'ensemble, on s'intéresse à la structure de dictionnaire. On stockera donc dans les ABR des couples clé-valeur. On veut donc implémenter l'interface suivante:

```
1  (* 'k est le type des clés, 'v celui des valeurs *)
2  type ('k, 'v) abr =
3    | V
4    | N of 'k * 'v * ('k, 'v) abr * ('k, 'v) abr
5  ;;
6
7  (* valeur associée à key dans a *)
8  let get (key: 'k) (a: ('k, 'v) abr) : 'v option = ... ;;
9
10 (* met à jour a avec le couple (key, value) *)
11 let set (key: 'k) (value 'v) (a: ('k, 'v) abr) : ('k, 'v) abr = ... ;;
12
13 (* supprime la clé key de a. Si key n'est pas dans a, ne fait rien mais affiche
14    un avertissement *)
15 let del (key: 'k) (a: ('k, 'v) abr) : ('k, 'v) abr = ... ;;
```

Question 8. Implémenter les fonctions `get` et `set`.

Question 9. Écrire une fonction `extract_min: ('k, 'v) abr -> ('k * 'v) * ('k, 'v) abr` qui étant donné un arbre de recherche, renvoie le couple dont la clé est minimale, ainsi que l'arbre privé de cette clé.

Question 10. En utilisant cette fonction, implémenter la fonction `del`. N'oubliez pas l'avertissement en cas de clé manquante.

Question 11. On définit par récurrence A_n pour $n \in \mathbb{N}$:

- $A_0 = V$
- A_{n+1} est l'arbre A_n auquel on a ajouté le couple $(n+1, ())$

A_n contient donc les clés 1 à n. Écrire une fonction permettant de calculer les A_n.

Question 12. Renseignez-vous sur la fonction `Sys.time`. En l'utilisant, écrivez une fonction `time_abr: int -> float` qui calcule le temps nécessaire pour créer l'arbre $A(n)$. Tracez la courbe obtenue et expliquez pourquoi on observe cette loi.

Arbres rouge-noir

On considère maintenant des arbres dont les informations sont stockés aux feuilles. Le but de cette partie est d'implémenter les arbres rouge-noir vus en cours. On utilisera les types suivants:

```
1  type couleur = Rouge | Noir ;;
2
3  type 'a noeud_arn =
4    | Feuille of 'a
5    | Noeud of couleur * 'a * 'a noeud_arn * 'a noeud_arn
6  ;;
7
8  type 'a arn = 'a noeud_arn option ;; (* None représente l'arbre vide *)
```

Lorsqu'on implémentera une fonction sur les ARN, on implémentera une fonction sur le type `'a noeud_arn`, qui sera récursive, puis la fonction finale, sur le type `'a arn`, rajoutera une surcouche permettant de gérer le cas où l'arbre est vide. Par exemple, pour calculer le nombre d'éléments contenus dans un arn (i.e. le nombre de feuilles):

```
1  let rec feuilles_noeud (a: 'a noeud_arn) : int =
2    match a with
3    | F x -> 1
4    | N(c, x, g, d) -> feuilles_noeud g + feuilles_noeud d
5  ;;
6
7  let feuilles (a: 'a arn) : int = match a with
8    | None -> 0
9    | Some r -> feuilles_noeud r
10 ;;
```

Question 13. Écrire une fonction qui calcule la couleur de la racine d'un `'a noeud_arn`.

Question 14. Écrire une fonction qui détermine si un `'a noeud_arn` contient deux noeuds rouges successifs.

Question 15. Écrire une fonction calculant la hauteur noire d'un `'a noeud_arn`. Cette fonction devra en même temps vérifier que tous les chemins de la racine aux feuilles ont le même nombre de noeuds noirs. Si ce n'est pas le cas, elle lèvera une exception `ErreurHauteurNoire` que vous aurez défini au préalable.

Question 16. En utilisant les fonctions précédentes, écrire une fonction permettant de vérifier qu'un arbre de type `'a arn` est bien un ARN. Testez la sur de nombreux exemples pour vérifier que vous couvrez bien toutes les possibilités.

On implémente maintenant l'insertion. On rappelle que pour faire une insertion dans un ARN, on effectue une insertion classique d'ABR, en colorant le nouveau noeud en rouge. Puis, on corrige les éventuelles anomalies. Pour cela, on utilise une fonction auxiliaire qui permet de

transformer un arbre tel que l'un des enfants ainsi qu'un de ses enfants sont rouges en un ARN valide mais dont la racine peut être rouge. Référez vous au cours pour les quatre cas possibles.

Le schéma d'insertion sera donc:

- Pour insérer dans une feuille, on crée un nouveau noeud permettant de séparer les deux feuilles du nouvel arbre
- Sinon, on insère récursivement dans le bon sous-arbre, puis on corrige l'arbre obtenu.

Question 17. Implémenter une fonction `correctionARN: 'a noeud_arn -> 'a noeud_arn` qui corrige l'anomalie d'un ARN et le transforme en ARN relaxé. Cete fonction ne sera pas récursive.

Question 18. Implémenter une fonction `insertARNrelax: 'a -> 'a noeud_arn -> 'a noeud_arn` qui insère dans un ARN, et renvoie un arbre dont la racine est éventuellement rouge.

Question 19. En déduire une fonction `insertionARN` permettant d'insérer un élément dans un arbre rouge-noir. L'arbre résultant devra être un ARN, vérifiez le dans vos tests en utilisant la fonction de vérification que vous avez codé.

On définit la suite d'arbres $(B_n)_{n\in\mathbb{N}}$ comme suit:

- $B_0 = F(0)$
- $B_{n+1} = \textbf{insertionARN}(n+1, B_n)$ pour $n \in \mathbb{N}$

Question 20. Dessinez B_{14} (Lorsque vous affichez le résultat avec OCaml, l'indentation est faite de manière adaptée aux arbres). Vérifiez à la main que c'est un ARN. Quelle est sa hauteur noire ?

Question 21. Écrivez une fonction permettant de calculer la suite $(B_n)_{n\in\mathbb{N}}$. Vérifiez sur quelques exemples que ce sont bien des ARN, et que la hauteur de B_n est inférieure à $2\log_2(2n+1)$.

Question 22. Comparez les temps de création des B_n à celui des A_n, la suite d'arbres binaires de recherche introduite à la question précédente. Quelle est la complexité temporelle de la création de B_n ?

On s'intéresse maintenant à l'utilisation des ARN pour implémenter une fonction de tri. Pour trier une liste L, l'idée est d'insérer chaque élément de L dans un ARN, puis de calculer la liste des éléments de l'ARN dans l'ordre croissant.

Question 23. Renseignez vous sur la fonction `List.fold_right` et utilisez la pour écrire une fonction `insertion_liste: 'a list -> 'a arn -> 'a arn` permettant d'insérer une liste d'éléments dans un ARN.

Question 24. Écrire une fonction `listerARN` permettant de lister dans l'ordre croissant tous les éléments d'un ARN. N'oubliez pas que seules les feuilles encodent l'information !

Question 25. (**Bonus**) Faire la fonction précédente en $\mathcal{O}(n)$, donc sans utiliser la concaténation.

Question 26. En déduire une fonction `tri_arn: 'a list -> 'a list` permettant de trier une liste en utilisant un ARN. Quelle est sa complexité ?

Cette application des ARN motive leur utilisation pour implémenter des ensembles ou des dictionnaires. Nous avons vu plus tôt dans l'année que les tables de hachage permettent des complexité en $\mathcal{O}(1)$ pour les opérations d'insertion, de suppression et de recherche. Cependant, l'information dans une table de hachage ne suit aucune structure par rapport aux clés. Dans un ABR, l'information est stockée en suivant l'ordre des clés. Ceci permet de faire certaines requêtes sur les ABR de manière très efficace par rapport aux tables de hachage.

Question 27. Écrire une fonction permettant d'obtenir l'étiquette minimale d'un ARN.

Question 28. Écrire une fonction `query_range: 'a -> 'a -> 'a arn -> 'a list` telle que `query_range a b t` renvoie la liste des éléments de t compris entre a et b.

Question 29. (**Bonus**) Implémenter la suppression dans les ARN.