Institut National Universitaire Champollion

Types abstraits de données

Programmation Fonctionnelle en Caml
L3 INFO Semestre 5

Le but de ce TP est de revoir quelques types abstraits de données et d'en proposer des implémentations fonctionnelles. Ce sera également l'occasion de s'essayer à l'écriture de fonctionnelles récursives non élémentaires et de veiller à la gestion des exceptions.

1. La pile (à l'aide d'un type somme récursif)

On rappelle que le TAD pile est principalement caractérisé par le fait de pouvoir empiler ou dépiler l'élément au sommet de la pile en $\theta(1)$.
On peut implémenter une pile d'entiers en caml à l'aide du type somme récursif :

```
type pile = PileVide | Empile of int * pile ; ;
```

Remarque : On peut définir les piles générales par le type polymorphe :

```
type 'a pile = PileVide | Empile of 'a * 'a pile ; ;
```

Mais nous nous restreindrons ici au seul cas des piles d'entiers.
On définit également l'exception :

```
exception  pilevide ; ;
```

Écrire alors les fonctions suivantes, mettant en oeuvre les opérateurs principaux du TAD pile :

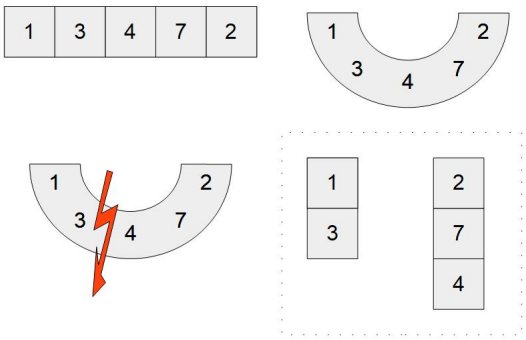
1. `estVide : pile -> bool` qui détermine si une pile est vide.
2. `sommet : pile -> int` qui retourne le sommet de la pile ou lève l'exception pilevide.
3. `empile : int -> pile -> pile` qui empile un élément au sommet de la pile.
4. `dépile : pile -> pile` qui élimine le sommet de la pile, ou lève l'exception pilevide.

2. La file (à l'aide d'un type produit)

Le TAD file est caractérisé par le fait que les éléments sont enfilés d'un côté de la file et défilés de l'autre, les deux opérations devant être de complexité constante.
Comme on ne dispose en Caml que de listes ne permettant en $\theta(1)$ que l'accès à la tête, nous allons utiliser deux listes, correspondant au début de la file (à l'endroit) et à la fin de la file (à l'envers), comme si on avait tordu la file en deux et qu'elle s'était coupée à un endroit quelconque, en deux listes.
On peut implémenter une file d'entiers en caml à l'aide du type produit :

```
type  file={Debut  : int list  ; Fin  : int list} ; ;
```

Selon ce principe, la file [1;3;4;7;2] pourra être représentée par



`{Debut =[1] ; Fin : [2 ;7 ;4 ;3]}` ou `{Debut =[1 ;3 ;4] ; Fin : [2 ;7]}`
ou `{Debut =[1 ;3] ; Fin : [2 ;7 ;4]}` ou encore `{Debut =[] ; Fin : [2 ;7 ;4 ;3 ;1]}`

qui sont toutes égales. On définit également l'exception :

```
exception filevide ; ;
```

Écrire alors les fonctions suivantes, mettant en oeuvre les opérateurs principaux du TAD file :

1. `estVide : file -> bool` qui détermine si une file est vide.
2. `transvase : file -> file` qui, alors que *Debut* est vide, transvase le contenu de la liste *Fin* dans la liste *Debut*.
3. `premier : file -> int` qui retourne le premier élément de la file ou lève l'exception filevide.
4. `enfiler : int -> file -> file` qui enfile un élément en fin de file.
5. `queue : file -> file` qui élimine le premier élément de la file, ou lève l'exception filevide.
6. `egale : file -> file -> bool` qui teste l'égalité entre deux files.

3. Les vecteurs (à l'aide d'une fonction)

Le vecteur impose un accès direct en temps constant à n'importe quel élément du vecteur, indépendamment de son indice.
La liste ne permet par de respecter cette contrainte, c'est pourquoi le type vecteur est bien défini en Caml. Par exemple :

```
[|1 ;2|] ; ;
[| 1.3 ;2.6|] ; ;
[| true ;false|] ; ;
[| [1 ;2 ;3] ;[2 ;3 ;5] |] ; ;
```

sont des vecteurs Caml.
Vous pouvez tester les fonctions suivantes, prédéfinies sur les vecteurs :

`vect_length, list_of_vect, vect_of_list, map_vect.`

Néanmoins il existe une implémentation très fonctionnelle des vecteurs, à l'aide... des fonctions entières.
On peut ainsi définir le vecteur [|7;4;3;2|] par la fonction :

```
let v= fun
0->7
|1->4
|2->3
|3->2
|_-> raise outOfRange ; ;
```

L'indexation se résume alors à un simple appel de fonction :$v(i)$.
Écrire la fonction `affecte : (int -> 'a) -> int * 'b -> int -> 'b` à un vecteur v et à un couple (i, j) associe le vecteur v' de même valeur que v mais tel que $v'(i) = j$.

4. Les arbres binaires

Nous avons déjà largement implémenté les arbres binaires dans les TP précédents.
On peut implémenter un type arbre binaire polymorphe de la manière suivante :

```
type ('f,'n) arbre =
Feuille of 'f
| Noeud of 'n * ('f,'n) arbre * ('f,'n) arbre ; ;
```

Écrire alors les fonctions suivantes, mettant en oeuvre les opérateurs principaux du TAD arbre binaire :

1. `feuille : ('a, 'b) arbre -> bool` qui détermine si un arbre est une feuille.
2. `droit : ('a, 'b) arbre -> ('a, 'b) arbre` et `gauche : ('a, 'b) arbre -> ('a, 'b) arbre` qui à un arbre donné associe son sous-arbre droit et son sous-arbre gauche.
3. `parcours : ('a, 'b) arbre -> 'a list` qui renvoie la liste des feuilles visités dans l'ordre du parcours en profondeur de l'arbre binaire.

5. Dictionnaires

Les fichiers dico.txt et dico.ml, fournis avec ce TP, contiennent respectivement un dictionnaire au format texte et des fonctions permettant de lire ce fichier et de créer la liste $listDico$ des mots de ce dictionnaire. On utilise également les fonctions classiques de la bibliothèque chaine.ml
Le but de cet exercice est de comparer l'efficacité de la recherche d'un mot dans ce dictionnaire en utilisant respectivement une simple liste, une table de hachage ou un arbre binaire de recherche.

1. Écrire une fonction `cherche1 : 'a -> 'a list -> bool` qui détermine si un mot est dans le dictionnaire en parcourant la liste des mots du dictionnaire.

   ```
   cherche1 "zebre" listDico ; ;
   ```

   On va maintenant représenter le dictionnaire par une table de hachage de clé (initiale,longueur). Le mot "zebre" sera alors dans la liste des mots associés à la clé (5,'z'). La table est alors de type

```
type table == ((char * int) * string list) list ; ;
Le type table est défini.
```

2. Écrire une fonction `insere : string * table -> table` qui insère une chaîne de caractères dans une telle table de hachage.
3. Écrire une fonction `list2tab : string list -> table` qui convertit une liste de mot en table de hachage.
4. Écrire une fonction `chercheTab : string -> table -> bool` qui détermine si un mot est présent dans la table de hachage.
5. Si vous avez trouvé tout cela trop facile, vous pouvez maintenant répondre aux mêmes questions en définissant votre dictionnaire à l'aide d'un arbre binaire de recherche...

6. Les graphes

Pour finir, nous ne résistons pas à l'envie d'implémenter les graphes à l'aide des listes d'adjacences, donc... pour nous d'une fonction donnant les listes voisinage. Nous nous restreindrons ici aux graphes orientés non valués. Par exemple :

```
let g1 = fun
1->[5]
|2->[1 ;4]
|3->[2]
|4->[3]
|5->[2 ;4]
| _-> raise outOfRange ; ;
g1  : int -> int list = <fun>
```

Écrire alors les fonctions suivantes :

1. `initGraphe : int -> int -> int list` qui crée le graphe à n sommets et sans arc.
2. `arcToG : int -> (int * int) list -> int -> int list` qui à un nombre de sommets n et à une liste d'arcs (i, j) associe le graphe correspondant.
3. `profond : int -> (int -> int list) -> int list` qui effectue le parcours en profondeur du graphe G à partir du sommet i et renvoie la liste des sommets parcourus dans l'ordre de première visite.