Lycée Chateaubriand
Informatique MP 2002/03
Mathieu Giraud

TP Caml n° 4
Représentations par pointeurs

Nous étudions aujourd'hui des structures définies à partir de pointeurs. Récupérez le fichier `TP4.ml` qui contient les définitions de type ainsi que quelques exemples.

1 Arbres

Les arbres binaires peuvent aussi se définir au moyen de pointeurs :

```
type 'n arbre =
  | Vide
  | Noeud of 'n noeud_pointeur
and 'n noeud_pointeur =
  {  mutable val : 'n ;
     mutable fils_g : 'n arbre ;
     mutable fils_d : 'n arbre
  } ;;
```

a. Écrire au choix deux fonctions parmi `somme` (somme des valeurs d'un arbre d'entiers), `miroir` (miroir d'un arbre) et `insere_ABR` (insertion d'une valeur dans un arbre binaire de recherche). Pour les fonctions qui renvoient un arbre, on essaiera de modifier l'arbre donné en entrée plutôt que de recréer un autre arbre. Dans ce cas, il est possible de renvoyer un tpye `unit` .

2 Files d'attente

Lorsque vous allez au guichet de la Poste, c'est généralement le premier arrivé qui est le premier servi. Cette logique FIFO *(first in, first out)* est celle de la file d'attente. Nous la programmons ici par une structure de données `'a file` qui doit permettre les opérations suivantes :

- `est_vide : 'a file -> bool` qui teste si la file est vide
- `insere : 'a file -> 'a -> 'a file` qui insère un élément dans la file
- `supprime : 'a file -> 'a * 'a file` qui renvoie l'élément le plus ancien de la file ainsi que la nouvelle file privée de cet élément
- `listage : 'a file -> 'a list` qui renvoie la liste des éléments dans la file

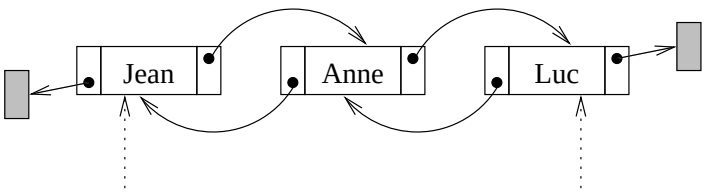
On définit aussi l'exception `exception File_vide` qui sera renvoyée lors de l'appel de `supprime` sur une file vide.

b. Pour commencer, on pose simplement `type 'a file == 'a list`. Écrire les quatres fonctions et donner leur complexité.

Pour obtenir des fonctions plus efficaces, on se propose maintenant de représenter les files d'attente au moyen de listes doublement chaînées : chaque cellule de la liste contient une valeur du type `'a` et deux liens, un vers la cellule suivante, un vers la précédente.

```
type 'a liste_double = Nil | Cellule of 'a cell
and 'a cell =
  {
    mutable lien_ar : 'a liste_double ;
    mutable valeur  : 'a ;
    mutable lien_av : 'a liste_double
  } ;;

let rec c1 = Cellule { lien_ar = Nil ; valeur = "Jean" ; lien_av =  c2 }
    and c2 = Cellule { lien_ar =  c1 ; valeur = "Anne" ; lien_av =  c3 }
    and c3 = Cellule { lien_ar =  c2 ; valeur = "Luc"  ; lien_av = Nil } ;;
```



La file d'attente est alors représentée par une structure avec des liens vers le début et la fin de la liste (les flèches en pointillés sur le dessin). Ces deux liens pointent vers la même `liste_double`, mais pas au même endroit.

```
type 'a file = { mutable tete  : 'a liste_double ;
                 mutable queue : 'a liste_double } ;;

let file1 = { tete = c1 ; queue = c3 } ;;
```

Enfin, la fonction `let nouvelle_file () = { tete = Nil; queue = Nil }` peut être utilisée pour construire une nouvelle file.

c. Écrire les quatres fonctions souhaitées et donner leur complexité.

3 Parcours colorés de graphes

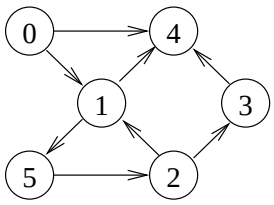
On définit ici des graphes où les sommets sont étiquetés par des chaînes de caractères et où les arcs sont orientés. Chaque sommet peut avoir une couleur.

```
type couleur = Noir | Rouge | Vert | Jaune ;;

type sommet = {
    mutable valeur : string ;
    mutable arcs : sommet list ;
    mutable coul : couleur
              } ;;

type graphe = sommet ;;
```

Un graphe `g` sera ici représenté par l'un d'un de ses sommets s_0 qu'on appellera "racine" : le graphe sera donc limité à l'ensemble des sommets atteignables à partir de s_0.




```
let rec s0 = { valeur = "s0"; arcs = [s1; s4] ; coul = Noir }
    and s1 = { valeur = "s1"; arcs = [s4; s5] ; coul = Noir }
    and s2 = { valeur = "s2"; arcs = [s1; s3] ; coul = Noir }
    and s3 = { valeur = "s3"; arcs = [s4]     ; coul = Noir }
    and s4 = { valeur = "s4"; arcs = []       ; coul = Noir }
    and s5 = { valeur = "s5"; arcs = [s2]     ; coul = Noir } ;;

let g = s0 ;;
```

d. On supppose que tous les sommets du graphe ont pour couleur `Noir`. Écrire une fonction récursive `cycle : graphe -> bool` qui teste si le graphe contient un cycle. On pourra utiliser la couleur `Rouge` pour indiquer qu'on a déjà visité un sommet.

e. On supppose toujours que tous les sommets du graphe ont pour couleur `Noir`. Écrire une fonction `enumere : graphe -> sommet list` qui renvoie la liste de tous les sommets du graphe.

f.* Essayer d'écrire une fonction `initialise : graphe -> unit` qui colore à `Noir` tous les sommets. Quel est le problème ?

g.* Voyez-vous d'autres représentations des graphes qui pourraient mener à des parcours plus simples ?

Sources

- Files d'attente : un de mes anciens TPs
- Arbres : un TP de l'année dernière