Listes, définition de types, évaluation

Listes

Exercice 1. Considérer la fonction

```
let rec longeur = function
     [] -> 0
   | t::q -> 1 + longeur q ;;
```

– Donner les étapes de l'évaluation par valeur de l'expression `longeur [1;2;3]`.
– Quel rapproche peut-on faire à la (définition de la) fonction `longeur` ? (Suggestion : examiner la complexité en temps et espace de cette fonction.)
– Proposer une amélioration de cette fonction.

Exercice 2.

algox.ml

```
 1 : let rec length = function
 2 :     [] -> 0
 3 :   | x::xs -> 1 + length(xs);;
 4 :
 5 : let rec split liste n =
 6 :   if n <= 0 then ([],liste )
 7 :   else
 8 :     match liste with
 9 :         [] -> ([],[])
10 :       | x::xs ->
11 :           let (l1,l2) = split xs (n-1) in
12 :             (x::l1,l2) ;;
13 :
14 : let rec merge liste1 liste2 lesseq =
15 :   match  (liste1,liste2) with
16 :       ([],_) -> liste2
17 :     | (_,[]) -> liste1
18 :     | (x::xs,y::ys) ->
19 :         if lesseq x y then
20 :           x::(merge xs liste2 lesseq)
21 :         else
22 :           y::(merge liste1 ys lesseq) ;;
23 :
24 : let rec algox lesseq = function
25 :     [] -> []
26 :   | [x] -> [x]
27 :   | liste  ->
28 :       let
29 :         moitie = length liste / 2
30 :       in
31 :       let
32 :         (l1,l2) = split liste moitie
33 :       in
34 :         merge ( algox lesseq l1) ( algox lesseq l2) lesseq ;;
```

Expliquer ce que fait le programme `algox.ml`

Exercice 3. Écrire une fonction

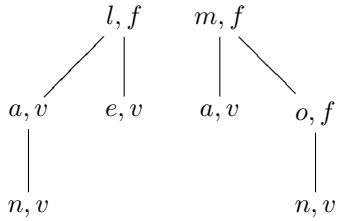
```
qsort : 'a list -> ('a -> 'a -> bool) -> 'a list
```

qui prend en paramètre une liste, une fonction pour comparer les éléments de cette liste, et retourne la liste trié. L'algorithme « trie vite » (quicksort en anglais) sera utilisé pour trier les listes.

Définitions de types

Arbres lexicaux

Un arbre lexicale a la forme suivante :



On se sert des arbres lexicaux pour représenter les dictionnaires, i.e. collections de mots. Par exemple l'arbre ci-dessus représente le dictionnaire $\{la, lan, le, ma, mon\}$. On peut définir le type des arbres en Ocaml comme il suit :

```
# type lex_noeud = Letter of char * bool * lex_arbre
  and
  lex_arbre = lex_noeud list;
# type mot = char list;
```

Exercice 4.
- Écrire une fonction `exists : mot -> lex_arbre -> bool` pour tester si un mot appartient au dictionnaire.
- Écrire une fonction `insert : mot -> lex_arbre ->  lex_arbre` pour ajouter un mot dans un dictionnaire.
- Écrire une fonction `construire : mot list -> lex_arbre` qui construit le dictionnaire à partir d'une liste de mots.

Ensembles et graphes

Exercice 5. Proposer trois définitions différentes d'un type `ens` pour représenter les ensembles. Écrire une liste d'opérations sur les ensembles qu'on souhaiterai implémenter, ainsi que le « prototype »de chaque fonction.

Exercice 6. Proposer une définition de type pour les graphes. Écrire une liste d'opérations sur les graphes qu'on souhaiterai implémenter. Écrire le « prototype »de chaque fonction.

Évaluation par valeur (et par nom)

Exercice 7. Expliquer ce qu'il se passe pendant la session suivante :

```
# let x = 3 + 5;;
# let rec fonction = function
   [] -> x
 | (tete::queue) -> x + tete + (fonction queue);;
# let x = 2;;
# fonction [x;3;x+x;5];;
```

Exercice 8. Considérer la fonction `sialorssinon`, définie comme il suit :

```
let sialorssinon x y z = if x then y else z ;;
```

- Comparer les évaluations par valeur des expressions
  - `sialorssinon true (3 + 4) (5 + 6)`,
  - `if true then (3 + 4) else (5 + 6)`,
- Proposer trois expressions `x`, `y`, et `z` telles que les évaluations par valeur de `sialorssinon x y z` et , `if x then y else z` ne donnent pas le même résultat.
- Que se passe-t'il si dans un langage fonctionnel dont la stratégie d'évaluation est par valeur, on ne possède ni des constructeurs comme `if ... then ... else ...` ?