Récursion et itération, exceptions, objets modifiables

Récursion et itération sur les listes

Exercice 1. Rappelons que

```
List.map f [b1;b2;...;bn] = [ (f b1); (f b2); .. .; (f bn) ]
List.fold_left f a [b1;b2;...;bn] = (f ...  (f (f a b1) b2) ... bn)
List.fold_right f [a1;...;an] b = (f a1 (f a2 ( ... (f an b))))
```

- Donner votre définitions de ces fonctions, et calculer leur types.
- Quelle entre `fold_left` et `fold_right` est récursive terminale ?
- Définir la fonction `fold_right` à partir de la fonction `fold_left` et de la fonction `List.rev`.
  Rappel : `List.rev [a1 ;a2 ;... ;an] = [an ;... ;a2 ;a1]`
- Démontrer, par induction sur la longueur des listes, l'équivalence suivante :
  `List.fold_left g a (List.map f l) =`
  `List.fold_left (fun x y -> g x (f y)) a l`

Exercice 2. Voici le fichier `length.ml` contenant deux possibles définitions de la fonction `length` :

```
let rec length1 l = match l with
    [] -> 0
  | t::q -> 1 + length1 q ;;
let length2 l =
  let rec length_acc ll acc  =
    match ll with
        [] -> acc
      | t::q -> length_acc q (acc +1)
  in
    length_acc l 0 ;;
```

1. Démontrer, par induction sur la longueur des listes, que `length1 l = length_acc l 0`.
2. A l'aide la commande `ocamlopt -S length.ml` produire un fichier assembleur `length.s`. Comparer les fonctions `camlLength__length_acc` et `camlLength__length1`.

Exercice 3. Définir un nouveau type `'a liste` ayant la forme d'un record contenant un liste conventionnelle, un fonction pour comparer les élément de la liste, un booléen pour dire si cette liste est trié.
Écrire une fonction `insert : 'a -> 'a liste -> 'a liste` qui ajoute un élément de la liste.
Écrire une fonction `sort : 'a liste -> 'a liste` qui fait le tri par insertion d'une liste donnée.

Exceptions

Exercice 4. Une possible méthode pour définir et gérer des fonctions partielles est d'utiliser le type `'a option`. Une autre possibilité est d'utiliser les mécanisme des exceptions. Réécrire le code suivant, en utilisant le mécanisme des exceptions à la place du type `'a option`.

```
let pred n = if n > 0 then Some (n-1) else None ;;
let est_str_prositif n = match pred n with
    None -> false
  | Some(_) -> true;;
```

Exercice 5. On veut écrire une fonction `essayer_les_trois` qui essaie d'évaluer trois expressions de même type `expr1`, `expr2` et `expr3` dans l'ordre, de cette façon : si une expression lève un exception, on essayera la prochaine ; si toutes les expression lèvent une exception, la fonction lèvera elle même une exception.
On propose le code suivant :

```
let essaier_les_trois = fun x y z ->
  try x with _ ->
    try y with _ ->
      try z with _ -> failwith "essaier_les_trois"
;;
```

Quel problème pose ce code ? Par exemple, quel résultat donnera l'évaluation de l'expression suivante ?

```
# essaier_les_trois (List.hd []) (List.tl []) [1];;
```

Exercice 6. Est ce que le code (incomplet) suivant vous rappelle quelque-chose ?

```
0  type variable = Var of string;;
1  type literal = Positive of variable | Negative of variable;;
2  type clause = literal list;;
3  type ens_clause = clause list;;

5  let rec algox (ens_cl : ens_clause) =
6    if est_vide ens_cl then Some(empty)
7    else
8      if contient_faux ens_cl then None
9      else
10       let
11         appliquer1 regle_unaire ens si_echec_quoi_faire =
12                ...
13       and
14         appliquer2 regle_binaire ens si_echec_quoifaire =
15                ...
16       in
17         appliquer1 regle1 ens_cl
18           fun () -> (
19             appliquer1 regle2 ens_cl
20              fun () -> (
21                appliquer1 regle3 ens_cl
22                 fun () -> (
23                   appliquer1 regle4 ens_cl
24                    fun () -> (
25                      appliquer2 regle5 ens_cl None
26                    )
27                 )
28              )
29           )
30 ;;
```

1. On se sert du mécanisme des exceptions si une règle `reglei` n'est pas applicable à un ensemble `ens` : elle lèvera l'exception `Failure "regle"`. Écrire, comme premier exemple, le code de la `regle2`.
2. Compléter la définition (récursive) de `algox` avec les définitions de `applquer1` et `appliquer2` en utilisant `try ... with`.
3. Écrire la liste des symboles non défini dans ce code, et en donner les types.
4. ... compléter le code.

Objets modifiables

Exercice 7. Implémenter le module `Pile`, dont l'interface est définie dans le fichier `pile.mli` :

```
type 'a t
exception Pile_vide
exception Pile_pleine
(** Créer une pile vide *)
val create : unit -> 'a t
val push : 'a -> 'a t -> unit
val pop : 'a t -> 'a
(** Retourne le sommet de la pile sans l'enlever de la pile *)
val peek : 'a t -> 'a
```