Université Pierre et Marie Curie	option Informatique 2002−2003 feuille de TP numéro 5	**Langage *Objective Caml***

Exercice 1 Signaler une erreur

Question 1.1 Exceptions prédéfinies

Essayer d'évaluer les déclarations suivantes :

```
let rec f = function 0 -> 1 | n -> n * f (n-1) ;;
let g x = 1 / x ;;
let _ =  f (-1) ;;
let _ =  g 0 ;;
```

Ecrire une fonction `g'` qui appelle la fonction `g` et qui ne lève que l'exception `Invalid_argument` avec un message d'explication. Ecrire de même une fonction `f'` qui appelle `f` et qui ne lève que `Failure` avec un message d'explication de l'erreur.

La liste des exceptions prédéfinies est dans : *Userman, section "Chapter 18: The core library", "Exceptions"*

Question 1.2 Déclaration d'exceptions

Ecrire les fonctions `f''` et `g''` qui appellent respectivement `f` et `g`, et qui lèvent une exception `MauvaisArgument` permettant de connaître l'entier ayant provoqué l'erreur.

Question 1.3 Liste d'association

On veut implémenter la structure de données des listes d'associations. On les modélise par des valeurs de type `('a * 'b) list`, `'a` étant le type des clefs, `'b` étant le type des éléments associés.

Exemple en utilisant les fonctions de la librairie standard, module `List`.

```
let l = [1, 'a' ; 2, 'b' ; 3, 'c'] ;;
List.assoc 2 l ;;
List.assoc 5 l ;;
List.mem_assoc 0 l ;;
let l' = List.remove_assoc 2 in List.assoc 2 l' ;;
```

Ecrire votre propre version des fonctions `assoc`, `mem_assoc` et `remove_assoc`. Vérifier qu'elles ont les mêmes types que celles du module `List`.

On veut écrire une fonction `assoc'` qui ne lève plus l'exception `Not_found`, mais une exception `ClefPasDansLaListe` permettant de connaître la valeur de la clef en question. Vérifier le type de `assoc'`, remarque(s) ?

Exercice 2 Les structures de contrôle impératives

Le langage *Objective Caml* dispose d'une construction *while ... do ... done*, mais ne dispose pas des primitives break et continue, qui permettent respectivement de sortir du corps de la boucle et de passer à l'itération suivante. On veut émuler ces primitives au moyen d'exceptions.

Question 2.1 Ecrire les trois fonctions suivantes

```
val continue : unit -> 'a
val break : unit -> 'a
val while_do : (unit -> bool) -> (unit -> 'a) -> unit
```

Question 2.2 Tester ces fonctions avec le code suivant :

```
let c = ref 0 ;;
let test () = !c < 100 ;;

let corps () =
  incr c;
  if !c = 5 then continue();
  if !c > 20 then continue();
  if !c = 10 then break();
  print_int !c; print_newline ()
  ;;

let run_loop = while_do test corps; while_do test corps;;
```

Qu'affiche cette dernière phrase (`run_loop`) ?

Exercice 3 Optimisation

Question 3.1 Optimisation par sortie de boucle/récursion
Ecrire une fonction `mult_list`, telle que `mult_list l` retourne le produit de tous les entiers de la liste `l`. Optimiser la fonction `mult_list` en utilisant une exception pour abréger le calcul lorsque cela est possible.

Question 3.2 Usage de la construction `try ... with`
Ecrire une fonction `make_list`, telle que `make_list n` retourne la liste des `n` premiers entiers.
Ecrire une fonction `timer`, telle que `timer f` exécute `f` cent fois et retourne le temps processeur nécessaire à ce calcul.*N.B. Voir la fonction time du module Sys dans la librairie standard.*
Ecrire les fonctions `mult_list1` et `mult_list2` en mettant la construction `try ... with` respectivement à l'intérieur et à l'extérieur de la récursion. Comparer leurs temps d'exécution sur une liste de 10 000 entiers. Conclusion...

Exercice 4 Et si l'on n'avait pas d'exception...

On suppose dorénavant que l'on ne dispose plus des exceptions dans le langage. On veut donc définir un type `'a result`, permettant de modéliser une valeur de type `'a` ou une erreur dans un calcul d'une valeur de type `'a`. Sous ces hypothèses, écrire des versions modifiées des fonctions `assoc`, `mem_assoc` et `remove_assoc`.