Feuille d'exercices n° 7

Exercice 1. Écrire une fonction qui prend deux arguments entiers `x` et `y`, et qui retourne le résultat de la division entière de `x` par `y`. Si la valeur de `y` est nulle, la fonction capture l'exception prédéfinie levée par le langage et retourne 0.

Exercice 2. Écrire une fonction qui, pour un réel x donné en paramètre, retourne la valeur $\log(\sin(x))$ lorsque $\sin(x) > 0$, et affiche le message `"argument invalide"` sinon.

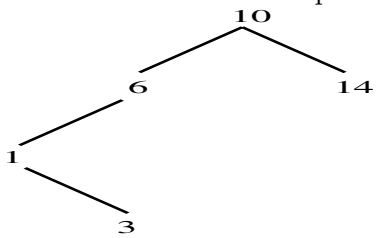
Exercice 3.

1. Définir une fonction `trouve` qui prend en arguments un prédicat `p` et une liste `l`, et qui renvoie le premier élément de `l` qui vérifie la propriété `p` ; si aucun élément ne convient, la fonction lève une exception que vous aurez définie préalablement.
2. Utiliser la fonction `trouve` pour définir une fonction `trouver_bis` qui renvoie une liste contenant uniquement le premier élément de `l` qui vérifie la propriété `p` ou une liste vide si aucun élément ne convient.

Exercice 4. Écrire une fonction `zip` qui reçoit deux listes `[a1 ; ... ; an]` et `[b1 ; ... ; bn]` en arguments et qui renvoie la liste des couples `[(a1, b1) ; ... ; (an; bn)]`. En quoi cette fonction fait-elle usage des exceptions ?

Exercice 5. Un arbre binaire est dit équilibré si, pour chaque nœud, les fils gauche et droit sont des arbres de même hauteur.

1. Définir le type inductif `'a arbre`.
2. Définir la variable `ab` correspondant à l'arbre suivant :



3. Version naïve
   a) Écrire une fonction `hauteur : arbre -> int` qui renvoie la hauteur d'un arbre binaire.
   b) Écrire une fonction `equilibre : arbre -> bool` qui vérifie si un arbre binaire est équilibré à l'aide de `hauteur`.
   c) Pourquoi la fonction `equilibre` n'est-elle pas efficace ?
4. Version améliorée
   d) On définit une exception `Desequilibre`, que l'on utilise pour programmer une nouvelle fonction `equi_hauteur : arbre -> int`. Cette fonction renvoie la hauteur d'un arbre équilibré, et lève l'exception `Desequilibre` si elle détecte un déséquilibre dans l'arbre. Cette exception doit être levée dès que possible.
   e) Définir enfin une fonction `equilibre_rapide : arbre -> bool` qui utilise la précédente et renvoie un booléen indiquant si l'arbre reçu en paramètre est équilibré ou non.

Exercice 6. Le but de cet exercice est de faire deviner un nombre (tiré au hasard) à l'utilisateur en `i` coups au maximum. Le nombre à deviner doit être compris entre deux bornes qui seront données en paramètres. Si l'utilisateur trouve le nombre en moins de `i` coups, alors la fonction écrit `Bravo` sinon elle écrit `Echec`.

1. Écrire une fonction `lire_entier` qui prend en arguments deux entiers qui sont les bornes entre lesquelles le nombre à deviner se trouve, qui les affiche à l'utilisateur, puis qui lit l'entier tapé par l'utilisateur. Si ce nombre est bien compris entre les bornes, la fonction renvoie l'entier, sinon elle lève deux exceptions `Erreur_bornes` et `Erreur_lire`.
2. Ajouter à `lire_entier` la fonction `main` de telle façon que si le fichier lireEntier.ml qui les contient est compilé puis exécuté, il donne le comportement suivant :

```
./lireEntier.out
Début du jeu : borne inf : 1
borne sup : 50
Entrer un entier entre ces 2 bornes : 45
45
./lireEntier.out
Début du jeu : borne inf : 1
borne sup : 50
Entrer un entier entre ces 2 bornes : 98
Erreur bornes !
./lireEntier.out
Début du jeu : borne inf : 1
borne sup : 50
Entrer un entier entre ces 2 bornes : a
Erreur lecture !
```

3. Écrire une seconde version de la fonction `lire_entier` qui cette fois, repose la question jusqu'à ce que l'utilisateur entre un nombre compris entre les bornes.
4. Écrire la fonction `deviner` qui prend en arguments le nombre de tentatives autorisées pour deviner le nombre ainsi que les bornes. Pour générer de manière aléatoire le nob-mbre à deviner, vous pouvez utiliser la fonction `Random.int` spécifiée comme suit :

```
Random.int bound returns a random integer between 0 (inclusive) and bound (exclusive).
   bound must be greater than 0 and less than 2**30.
```

   Lors de l'exécution du code, on donnera les 3 arguments en ligne de commande.
5. Écrire une nouvelle version de la fonction `deviner` qui dit à chaque fois si le nombre à deviner est plus grand ou plus petit que le nombre entré.