TP1 - Révisions

Exercice 1 (Bases sur les listes).

1. Ecrire une fonction `minimum` prenant en entrée une liste d'entiers et renvoyant son minimum.
2. Ecrire une fonction `maximum` prenant en entrée une liste d'entiers et renvoyant son maximum.
3. Ecrire une fonction `map` prenant en entrée une fonction `f` et une liste `l` et renvoyant la liste obtenue en appliquant la fonction `f` à chaque élément de la liste `l`.

Exercice 2 (Calcul matriciel).

Ecrire une fonction `pdmatriciel` prenant en arguments deux matrices et renvoyant la matrice produit. On suppose que les matrices d'entrées sont telles que le produit existe.
On pourra utiliser la fonction `make_matrix` du module `Array` de type `int->int->'a->'a array array`.

Exercice 3 (Factorielle).

1. Ecrire un algorithme itératif `factoriellei` prenant en entrée un entier `n` et calculant sa factorielle.
2. Ecrire un algorithme récursif `factorieller` prenant en entrée un entier `n` et calculant sa factorielle.

Exercice 4 (Binôme).

Ecrire une fonction `binome` prenant comme arguments deux entiers naturels `n` et `p` et renvoyant le coefficient binomial `p` parmi `n`.

On rappelle que pour tout $0 \leq p \leq n$, $\binom{n}{p} = \dfrac{n!}{(n-p)!p!}$ et $\binom{n}{p} = \binom{n-1}{p} + \binom{n-1}{p-1}$.

Exercice 5 (Date valide).

Ecrire la fonction `dateValide` qui prend en entrée les entiers `j`, `m` et `a`, et renvoie le booléen `true` si la date représentée par le triplet `(j,m,a)` (jour, mois, année) est une date valide, et `false` sinon.

Règles de validité d'une date :

- janvier, mars, mai, juillet, août, octobre, décembre ont 31 jours ;
- avril, juin, septembre, novembre ont 30 jours ;
- février a 29 jours si l'année est bissextile, 28 sinon ;
- une année n est bissextile si n est divisible par 4, sauf si n est divisible par 100 et pas par 400 ;
- on se limitera aux dates postérieures au 15 octobre 1582, date de mise en application en France de ces règles (calendrier Grégorien).

Exercice 6 (Décomposition en base 2).

1. Soit n un entier et $(d_i)_{i\geq 0}$ sa décomposition en base 2, c'est-à-dire $n = \sum_{i\geq 0} d_i 2^i$.
   (a) Donner la relation liant d_0 et la parité de n.
   (b) Que vaut $\sum_{i\geq 0} d_{i+1} 2^i$ en fonction de n ?
   (c) En déduire une manière de déterminer d_1.
2. Ecrire une fonction `decomp2` prenant en argument un entier `n` et renvoyant une liste contenant sa décomposition en base 2.

Exercice 7 (Sous-suites de trois nombres de somme maximale).

Ecrire la fonction *indiceTripletMaximal* qui pour une liste d'entier l, supposée de cardinal au moins égal à trois, retourne l'indice du premier élément du triplet de somme maximale.

Par exemple, pour la liste `l=[1;5;7;8;6;2;7;9;4]`, La fonction retournerait 2 (si les indices commencent à 0) indice du premier élément du triplet $(7, 8, 6)$ de somme maximale égale à 21.

Exercice 8 (Implémentation des ensembles).

On utilise des listes pour représenter des ensembles.

1. Ecrire une fonction `appartenance` prenant en argument un ensemble et un élément et renvoyant un booléen indiquant si l'élément appartient à l'ensemble.
2. Ecrire une fonction `suppression` qui supprime les redondances d'une liste.

On suppose que les ensembles sont définis sans redondance.

1. Ecrire une fonction `inter` prenant en argument eux ensembles et renvoyant leur intersection.
2. Ecrire une fonction `union` prenant en argument eux ensembles et renvoyant leur union.
3. Ecrire une fonction `difference` prenant en argument eux ensembles et renvoyant leur différence.

4. Ecrire une fonction `differencesym` prenant en argument eux ensembles et renvoyant leur différence symétrique.
   On rappelle que la différence symétrique de deux ensembles A et B se définit par
   $$A\Delta B = (A \cup B) \setminus (A \cap B)\,.$$

Exercice 9 (Suites oscillantes).
Soit f une fonction entière à valeur entière définie de $[0; 10\ 000]$ dans $]0; 10\ 000]$ et u la suite définie par :

$$u_0 = a \quad \text{avec } 0 \leqslant a \leqslant 10\ 000$$
$$u_{n+1} = f(u_n)$$

1. Montrer (mathématiquement) que la suite $(u_n)_{n\in\mathbb{N}}$ est périodique à partir d'un certain rang.
2. Ecrire une fonction `appartient` prenant en arguments une liste `l` de couples et un élément `e`, de même type que les premiers éléments des couples de `l`, renvoyant un couple composé d'un booléen et d'un entier. Le booléen vaut `true` si la liste `l` contient un couple commençant par `e` et `false` sinon. L'entier désigne l'indice de la première apparition de `e` dans la liste.
3. Ecrire la fonction `periode` qui étant donnée une fonction `f` et un entier `a` retourne `p` la valeur de la période de la suite `u` et `e` un élément de la partie périodique de la suite.

Exercice 10 (Tri d'une liste).

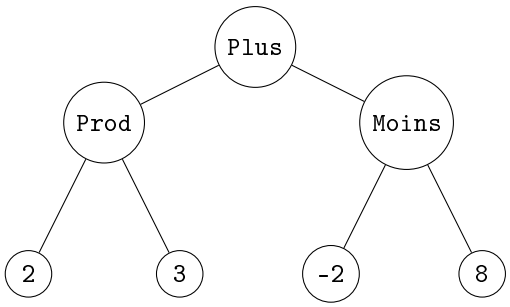
1. Ecrire une fonction `tri` qui pour une liste d'entiers `l` renvoie la même liste triée, par ordre croissant ou décroissant.
2. Prouver la terminaison de votre algorithme `tri`.
3. Prouver que votre algorithme tri bien la liste `l`.
4. Donner et prouver la complexité de votre algorithme `tri`.

Exercice 11 (Tri d'une liste (bis)).
Ecrire une fonction `vrai-tri` qui trie une liste d'entiers en une complexité au pire $\mathcal{O}(nln(n))$ (avec n la taille de la liste).

Exercice 12 (Arbres et expressions arithmétiques).
On définit le type `exparith` (pour expression arithmétique) par :

```
type exparith= E of int | Plus of exparith*exparith
                        | Moins of exparith*exparith
                        | Prod of exparith*exparith ;;
```

Ainsi `Plus(Prod(E(2),E(3)),Moins(E(-2),E(8)))` est de type `exparith`.
On peut la représenter à l'aide d'un arbre binaire où les feuilles sont des entiers et les noeuds des opérateurs :



1. On définit la hauteur d'une expression arithmétique comme la hauteur de l'arbre binaire associé. Ecrire une fonction `hauteur` de signature `exparith -> int` calculant la hauteur d'une expression arithmétique.
2. Ecrire une fonction `evalue : exparith -> int` évaluant une expression arithmétique.
3. Ecrire une fonction `pair : exparith -> bool` renvoyant `true` si l'expression arithmétique est paire et `false` sinon.
   (a) En utilisant la fonction `evalue`.
   (b) Sans utiliser la fonction `evalue`.

Exercice 13 (Pour les 5/2 qui devraient normalement avoir fait tout le reste en moins de temps qu'il n'en faut pour le dire).

On définit le type `log` (pour expression logique) comme suit :

```
type log= Top | Bottom
              | V of string
              | Et of log * log
              | Ou of log * log
              | Non of log
              | Xor of log * log
              | Implique of log * log
              | Equivalent of log * log ;;
```

1. On définit la hauteur d'une expression logique comme la hauteur de l'arbre binaire associé. Ecrire une fonction `hauteur` de signature `log -> int` calculant la hauteur d'une expression logique.
2. La longueur d'une expression logique et le nombre de nœuds de l'arbre associé. Ecrire une fonction `longueur` de signature `log -> int` calculant la hauteur de `e`.
3. Ecrire une fonction `variables` de signature `log -> string list` retournant la liste des variables apparaissant dans une expression logique. Attention : chaque variable apparaissant dans `e` doit apparaître une et une seule fois dans la liste résultat.
4. Ecrire une fonction `association` de type `'a -> ('a * 'b) list -> 'b` prenant en entrée un élément `x` de type `'a` et une liste `l` contenant des couples de la forme `'a * 'b` et retournant `y` tel que le couple `(x; y)` est le premier couple apparaissant dans la liste avec comme première composante `x`. (Remarque : `List.assoc` fait déja cela, mais il faut savoir la réEcrire !)
5. Ecrire une fonction `evalue : log -> (string * bool) list -> bool` évaluant une expression logique avec une distribution de vérité.
6. Ecrire une fonction `distributions` de type `string list -> (string * bool) list list` engendrant toutes les distributions de vérités sur un ensemble de variables `v`.
7. En déduire une fonction `est_tautologie` de signature `log -> bool`, retournant un booléen suivant si l'expression logique est une tautologie ou non.