Algorithmique
Contrôle n° 1

INFO-SUP S1
EPITA

6 Nov. 2017 - 08 :30

Remarques (à lire !) :

- ☐ Vous devez répondre sur **les feuilles de réponses prévues à cet effet.** Aucune autre feuille ne sera ramassée (gardez vos brouillons pour vous).
  Ne séparez pas les feuilles à moins de pouvoir les ré-agrafer pour les rendre.
- ☐ La présentation est notée en moins, c'est à dire que vous êtes noté sur 20 et que les points de présentation (2 au maximum) peuvent être retirés de cette note.
- ☐ Tout code CAML non indenté ne sera pas corrigé.
- ☐ Tout code CAML doit être suivi de son évaluation : la réponse de CAML .
- ☐ En l'absence d'indication dans l'énoncé, les seules fonctions que vous pouvez utiliser sont `failwith` et `invalid_arg` (aucune autre fonction prédéfinie de CAML).
- ☐ Aucune réponse au crayon de papier ne sera corrigée.
- ☐ Durée : 2h00 (May the force...)

Exercice 1 (Types Abstraits : liste itérative (modifier) – *2 points*)

Supposons le type abstrait algébrique *liste itérative* vu en cours et dont la signature est rappelée ci-dessous.

TYPES
liste, place

UTILISE
entier, élément

OPÉRATIONS

liste-vide	:	→ liste
accès	:	liste × entier → place
contenu	:	place → élément
ième	:	liste × entier → élément
longueur	:	liste → entier
insérer	:	liste × entier × élément → liste
supprimer	:	liste × entier → liste
succ	:	place → place

On se propose de faire une extension à ce type en définissant une nouvelle opération : *modifier*. Celle-ci nous permettra de changer la valeur d'un élément existant. Son profil est le suivant :

OPÉRATIONS

modifier	:	liste × entier × élément → liste

ou *modifier(l,i,e)* modifie la valeur du $i^{ème}$ élément de la liste l en le remplaçant par l'élément e.

1. Précisez l'éventuel domaine de définition de cette opération.
2. Donner les axiomes définissant de manière suffisamment complète cette opération.

Exercice 2 (Types Abstraits : liste récursive arrière – *4 points*)

Supposons le type abstrait algébrique *liste récursive* vu en cours et dont la signature est rappelée ci-dessous.

TYPES
liste, place

UTILISE
élément

OPÉRATIONS

liste-vide	:	→ liste
tête	:	liste → place
fin	:	liste → liste
cons	:	élément × liste → liste
premier	:	liste → élément
contenu	:	place → élément
succ	:	place → place

On se propose de définir le type *liste récursive arrière* en remplaçant les opérations *tête*, *fin*, *premier* et *succ* par les opérations *queue*, *début*, *dernier* et *pred* telles que :

— L'opération *queue* donne, pour une liste non vide, la dernière place de la liste,
— L'opération *début* donne, pour une liste non vide, la liste privée de son dernier élément,
— L'opération *dernier* donne, pour une liste non vide, le dernier élément de la liste,
— L'opération *pred* donne la place précédente d'une place dans la liste.

1. Précisez les éventuels domaines de définitions de ces opérations.
2. Donner les axiomes définissant de manière suffisamment complète le type *liste récursive arrière*.

Exercice 3 (Association – *4 points*)

Écrire la fonction `assoc k list` où `list` est une liste de couples (*key*, *value*) triée par clés (key) croissantes. Les clés sont des entiers naturels non nuls. Elle retourne la valeur (value) associée à la clé (key) `k`. Si `k` n'est pas valide ou si aucun couple n'a pour clé `k`, elle déclenche une exception.

Exemples d'utilisation :

```
# assoc 5 [(1, "one"); (2, "two"); (3, "three"); (5, "five"); (8, "eight")];;
 - : string = "five"

# assoc 4 [(1, "one"); (2, "two"); (3, "three"); (5, "five"); (8, "eight")];;
 Exception: Failure "not found".

# assoc (-1) [(1, "one"); (2, "two"); (3, "three"); (5, "five"); (8, "eight")];;
 Exception: Invalid_argument "k not a natural".
```

Exercice 4 (for_all2 – *5 points*)

1. Écrire la fonction CAML `for_all2` dont les spécifications sont les suivantes :
   - Elle prend en paramètre une fonction de test (un prédicat) à deux paramètres : p ainsi que deux listes : $[a_1; a_2; \cdots; a_n]$ et $[b_1; b_2; \cdots; b_n]$.
   - Elle vérifie si toutes les paires a_i b_i vérifient le prédicat p.
   - Si elle trouve une paire telle que p a_i b_i est faux, elle retourne faux. Sinon, elle déclenche une exception `Invalid_argument` si les deux listes sont de longueurs différentes.
2. Utiliser la fonction `for_all2` pour définir une fonction qui vérifie si deux listes sont "quasi-identiques" : les valeurs aux même places ne peuvent différer de 1 au plus.

Exemples d'utilisation :

```
# almost [1; 2; 3; 4; 5] [1; 3; 2; 4; 6];;
 - : bool = true

# almost [1; 2; 3; 4; 5] [1; 4; 3; 4; 5];;
 - : bool = false
```

Exercice 5 (Combine it – *5 points*)

Écrire la fonction `combfilter` dont les spécifications sont les suivantes :
- Elle prend en paramètre une fonction à deux paramètres : f
  ainsi que deux listes : $[a_1; a_2; \cdots; a_n]$ et $[b_1; b_2; \cdots; b_n]$.
- Elle retourne la liste de toutes les paires d'éléments (a_i, b_i) tels que f a_i b_i est vrai. L'ordre des éléments des listes initiales est préservé.
- Elle déclenche une exception si les deux listes sont de longueurs différentes.

Exemple d'utilisation :

```
# let is_string_of_int e1 e2 = string_of_int e1 = e2 ;;
 val is_string_of_int : int -> string -> bool = <fun>

# combfilter is_string_of_int [1; 2; 3; 4; 5] ["1"; "0"; "0"; "4"; "5"] ;;
 - : (int * string) list = [(1, "1"); (4, "4"); (5, "5")]
```