Algorithmique
Contrôle n° 1

INFO-SUP S1#
EPITA

3 mai 2021 - 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).
  - — Répondez dans les espaces prévus, **les réponses en dehors ne seront pas corrigées** : utilisez des brouillons !
  - — Ne séparez pas les feuilles à moins de pouvoir les ré-agrafer pour les rendre.
  - — Aucune réponse au crayon de papier ne sera corrigée.
- ☐ 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 du résultat 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 (Une pincée de listes... – *5 points*)

Nous avons vu deux définitions de type abstrait pour les listes : le type "liste récursive" et le type "liste itérative". Ce qui est important, c'est de comprendre qu'en fait nous décrivons la même donnée, seule la manière de s'en servir diffère. Une façon simple de le montrer est d'écrire les opérations d'un type en terme de celles de l'autre et inversement.

1. Ecrire les opérations *tête*, *premier*, *fin* et *cons* en terme des opérations *accès*, *ième*, *supprimer* et *insérer*
2. Ecrire l'opération *insérer* en terme des opérations *listevide*, *cons*, *premier* et *fin*

Rappels

Type abstrait algébrique *Liste récursive*

TYPES
liste, place

UTILISE
élément

OPÉRATIONS

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

Type abstrait algébrique *Liste itérative*

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

Exercice 2 (Vérifier la présence – *4 points*)

Écrire la fonction `check_occ` x n *list* qui vérifie si l'élément x est présent au moins n fois dans la liste *list*.

```
# check_occ 1 1 [1; 4; 3; -10];;
- : bool = true
# check_occ 1 2 [1; 4; 3; -10];;
- : bool = false
# check_occ 1 2 [4; 1; 1; -10];;
- : bool = true
# check_occ "td" 1 ["caml"];;
- : bool = false
# check_occ "td" 3 ["td"; "caml"; "td"; "2021"; "td"; "Python"; "td"];;
- : bool = true
```

Exercice 3 (Avant dernier – *4 points*)

Écrire la fonction `insert_penultimate` x *list* qui insère l'élément x en avant-dernière position de la liste *list*. La fonction déclenche une exception `Invalid_argument` si la liste donnée en paramètre est vide.

```
# insert_penultimate (-1) [1;8;30;4];;
- : int list = [1; 8; 30; -1; 4]
# insert_penultimate "td" ["caml"];;
- : string list = ["td"; "caml"]
# insert_penultimate (42) [];;
Exception: Invalid_argument "insert_penultimate: empty list".
```

Exercice 4 (find2 – *5 points*)

1. Écrire la fonction CAML `find2 p l1 l2` dont les spécifications sont les suivantes :
   — Elle prend en paramètres une fonction booléenne à deux paramètres : `p` ainsi que deux listes : $[a_1; a_2; \cdots; a_n]$ et $[b_1; b_2; \cdots; b_n]$.
   — Elle retourne le premier couple d'éléments qui vérifie le prédicat p : la première paire d'éléments (a_i, b_i) tel que $p\ a_i\ b_i$ est vrai.
   — Elle déclenche une exception `Failure` si aucun couple (a_i, b_i) tel que $p\ a_i\ b_i$ est vrai n'a été trouvé ou si les deux listes sont de longueurs différentes.

2. Utiliser la fonction `find2` pour définir une fonction `first_shared l1 l2` qui retourne le premier élément présent dans les deux listes à la même position.

Exercice 5 (Mystery – 3 points)

Soit la fonction `mystery` définie ci-dessous

```
# let rec mystery a b c = match a with
    []   -> b
  | d::e -> let r = (mystery e b c) in c d r
;;
```

1. Donner le résultat de l'évaluation de la fonction mystery.

2. Donner les résultats des évaluations successives des phrases suivantes.

   (a) `# mystery [] 12 (let c x y = x+y in c) ;;`

   (b) `# mystery [1; 2; 3; 4; 5] 0 (function x -> function y -> x+y) ;;`