Feuille d'exercices n°3 bis

Exercice I : Extraction de sous-liste

Q1 – Définir la fonction de signature

```
drop (n:int) (xs:'a list) : 'a list
```

qui donne la liste obtenue en privant `xs` de ses `n` premiers éléments. Si `n` ≤ 0, le résultat est `xs`; si `n` est plus grand que la taille de `xs`, le résultat est la liste vide.

Q2 – Définir la fonction de signature

```
take (n:int) (xs:'a list) : 'a list
```

donne la liste des `n` premiers éléments de `xs`. Si `n` < 0, on obtient la liste vide. Si `n` est plus grand que la taille de `xs`, on obtient `xs` toute entière.

Q3 – Déduire de ce qui précède la définition de la fonction

```
sub (i:int) (len:int) (xs:'a list) : 'a list
```

qui extrait la sous liste de `xs` de longueur `len` qui commence à la position `i`.

Exemples:

Soit `cs=('E'::'x'::'t'::'r'::'a'::'c'::'t'::'i'::'o'::'n'::[])`

`(sub (-1) 3 cs)` = `[]`

`(sub 1 (-3) cs)` = `[]`

`(sub 2 0 cs)` = `[]`

`(sub 10 3 cs)` = `[]`

`(sub 2 5 cs)` = `'t'::'r'::'a'::'c'::'t'::[]`

`(sub 2 8 cs)` = `'t'::'r'::'a'::'c'::'t'::'i'::'o'::'n'::[]`

`(sub 2 42 cs)` = `'t'::'r'::'a'::'c'::'t'::'i'::'o'::'n'::[]`

`(sub 0 10 cs)` = `cs`

Exercice II : Sous-listes lacunaires, listes sous-jacentes

On appelera *sous-listes pleines* les sous listes calculées à l'exercice précédent. On s'intéresse à d'autres formes de sous listes dans cet exercice.

Q1 – On dit qu'une liste `xs` est *sous-liste lacunaire* d'une liste `ys` lorsque tous les éléments de `xs` sont présents dans `ys` et qu'ils ont le même ordre dans les deux listes. Par exemple `3::13::23::[]` est sous-liste lacunaire de `2::3::5::7::11::13::17::19::23::27::[]`. On peux avoir de répétitions, par exemple `1::1::[]` est sous-liste lacunaire de `1::0::0::1::0::[]`. La liste vide est sous-liste lacunaire de toute liste. La liste vide n'admet que la liste vide comme sous liste lacunaire.

Définir la fonction de signature

```
sublac (xs:'a list) (ys:'a list) : bool
```

qui donne `true` si et seulement si `xs` est une sous-liste lacunaire de `ys`.

Q2 – On considère des listes d'entiers, pour simplifier.

On dit qu'une liste `xs` est *liste sous-jacente* de `ys` si on peut retrouver `ys` à partir de `xs` en remplaçant dans `xs` les 0 par des éléments de `ys`.

Exemple: `0::3::0::0::0::13::0::0::23::0::[]` est une liste sous-jacente de `2::3::5::7::11::13::17::19::23::27::[]`.

Contre-exemple: `3::0::0::13::0::23::0::[]` n'est pas une liste sous-jacente de `2::3::5::7::11::13::17::19::23::27::[]`: il manque un `0` au début et un autre entre `3` et `13`.

Notez qu'une liste et sa liste-sous-jacente ont la même taille.

Définir la fonction de signature

```
sublying (xs:int list) (ys:int lit) : bool
```

qui donne `true` si et seulement si `xs` est liste sous-jacente de `ys`.

Q3 – Définir la fonction de signature

```
stretch (xs:'a list) (ys:'a list) : 'a list
```

qui donne la liste sous-jacente de `ys` construite à partir de `xs`. Si cela n'est pas possible, c'est-à-dire, si `xs` n'est pas une sous-liste lacunaire de `ys`, la fonction `stretch` déclenche l'exception `Invalid_argument "stretch"`.

Exemples:
`(stretch (3::5::7::[]) (1::2::3::4::5::6::7::8::9::[]))` donne la liste `0::0::3::0::5::0::7::0::0::[]`
`(stretch (3::42::7::[]) (1::2::3::4::5::6::7::8::9::[]))` déclenche `Invalid_argument "stretch"`.

Il est inutile d'utiliser `sublac` pour définir `stretch`.

Exercice III : Listes d'associations ou dictionnaire

Les listes d'associations sont des listes de couples. On peut les utiliser pour simuler les structures de *dictionnaire* de Python. On reprend ici un exercice du recueil d'exercices de l'UE LU1IN001: «Recettes de cuisine».

Dans cet exercice on utilise la notation des listes «entre crochets»: on écrit `[x1; x2; ..; xn]` pour `x1::x2::  ..::xn::[]`

On modélise le dictionnaire des ingrédients d'un ensemble de recettes par une liste de type `(string * (string list)) list`. On pose (pour alléger l'écriture):

```
type dico = (string * (string list)) list
```

On a, par exemple

```
let dessert = [
    ("gateau chocolat" , ["chocolat"; "oeuf"; "farine"; "sucre"; "beurre"]);
    ("gateau yaourt" , ["yaourt"; "oeuf"; "farine"; "sucre"]);
    ("crepes" , ["oeuf"; "farine"; "lait"]);
    ("quatre-quarts" , ["oeuf"; "farine"; "beurre"; "sucre"]);
    ("kouign amann" , ["farine"; "beurre"; "sucre"])
 ]
```

qui est de type `dico`.

La fonction de la bibliothèque standard `List.assoc` permet d'obtenir la liste des ingrédients d'une recette. Par exemple: `(List.assoc "crepes" dessert)` vaut `["oeuf"; "farine"; "lait"]`

Q1 – Définir la fonction

`nb_ingredients (r:string) (rdic:dico) : int`

qui donne le nombre d'ingrédients de la recette `r` dans le dictionnaire `rdic`. Si la recette n'existe pas dans le dictionnaire, on obtiendra l'exception `Not_found`.

Par exemple: `(nb_ingredients "crepe" dessert)` vaut 3.

On utilise la fonction `List.length` de la bibliothèque standard.

Q2 – Définir la fonction

`recette_avec (i:string) (rdic:dico) : string list`

qui donne la liste des recettes du dictionnaire `rdic` qui utilisent l'ingrédient `i`.

On utilise la fonction `List.mem` de la bibliothèque standard.

Q3 – Définir la fonction

`recette_sans (rdic:dico) (i:string) : string list`

où `rdic` est un dictionnaire de recettes et qui donne la liste des recettes de `rdic` qui n'utilisent pas l'ingredient `i`.

On peut utiliser `List.mem`

Q4 – Dans la suite, on construira des listes dans laquelle on ne veut pas de répétition par union de deux listes. On définit pour cela la fonction

```
union (xs:'a list) (ys:'a list) : 'a list
```

donne la liste qui ajoute, sans répétion tous les éléments de `xs` à `ys`.

L'ordre des éléments dans la liste n'importe pas. On peut utiliser la fonction `List.mem`.

Par exemple:
`(union [] [4;2])` donne `[4;2]`
`(union [1;2;1] [])` donne `[1;2]` (ou `[2;1]`)
`(union [1;2;1] [4;2])` donne `[1;4;2]` (ou `[4;1;2]`, etc.)

Remarque: le résulat ne contient pas de répétition si `ys` n'en contient pas. Par exemple `(union [1] [2;2])` peut donner `[1;2;2]`.

Q5 – Définir la fonction

```
tous_ingredients (rdic:dico) : string list
```

qui donne la liste de tous les ingrédients mentionnés dans le dictionnaire de recettes `rdic`.

Q6 – On veut maintenant construire un dictionnaire d'ingrédients à partir d'un dictionnaire de recettes. C'est-à-dire, un dictionnaire qui donne les recettes qui utilisent un ingrédient. Par exemple, le dictionnaire des ingrédients de `dessert` est la liste:

```
[ ("lait",     ["crepes"]);
  ("beurre",   ["gateau chocolat"; "quatre-quarts"; "kouign amann"]);
  ("oeuf",     ["gateau chocolat"; "quatre-quarts"; "crepes"; "gateau yaourt"]);
  ("yaourt",   ["gateau yaourt"]);
  ("sucre",    ["kouign amann"; "gateau chocolat";"quatre-quarts"; "gateau yaourt"]);
  ("farine",   ["kouign amann"; "gateau chocolat";
                "quatre-quarts"; "crepes"; "gateau yaourt"]);
  ("chocolat", ["gateau chocolat"])]
```

C'est une liste de type `dico`.

Définir la fonction

```
dico_ingredients (rdic:dico) : dico
```

qui donne le dictionnaire des ingrédients du dictionnaire de recette `rdic`.

Utilisez les fonctions précédente ainsi qu'une fonction récursive locale.

Q7 – Définir la fonction

```
ingredient_principal (idic:dico) : string
```

où `idic` est un dictionnaire d'ingrédients et qui donne le nom de l'ingrédient utilisé par le plus grand nombre de recettes dans `idic`.

Pour notre exemple, c'est `"farine"`.

La fonction déclenche l'exception `Invalid_argument "idic"` si `idic` est vide.

Utilisez la fonction `List.length`.

C'est une fonction de recherche d'un max dans une liste (voir cours).