Contrôle Terminal PFITA
Durée : 1h30
Documents Non Autorisés

La qualité du code compte autant que sa correction.

Si cela vous facilite l'implantation des fonctions demandées, vous pouvez utiliser des fonctions auxiliaires, à condition d'expliquer leur rôle et de donner leur code OCaml.

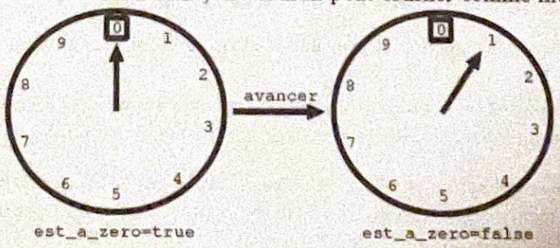
Les fonctions de la bibliothèque standard OCaml sont interdites, sauf la fonction de concaténation (`@`) que vous avez le droit d'utiliser.
Le barème est donné à titre indicatif.

Exercice 1 : Rotation de listes et TAD `CADRAN` (environ 9 points)

1. Définir la fonction `rot_left` prenant en argument une liste non vide et renvoyant la liste « décalée vers la gauche » où son premier élément est passé en queue. Par exemple `rot_left [1;2;3;4] = [2;3;4;1]`. Si la liste est vide, on lèvera une exception avec `failwith`.
2. Définir la fonction `extract` prenant en argument une liste et renvoyant le couple formé du dernier élément de la liste, et de la liste privée de cet élément. Si la liste est vide, on lèvera une exception avec `failwith`.
3. Définir la fonction `rot_right` prenant en argument une liste non vide et renvoyant la liste « décalée vers la droite » avec son dernier élément passé en tête. Par exemple `rot_right [1;2;3;4] = [4;1;2;3]`. Si la liste est vide, une exception sera levée.
4. Définir la fonction `iterer` prenant en argument un entier n, une fonction f et un argument `a`, et renvoyant la composition de n applications de f à `a`, c'est-à-dire, `iterer` n f `a` = $\underbrace{f(f(...(f}_{n\text{ fois}}(\texttt{a}))...))$. Si `n` $\leqslant 0$, la fonction renvoie directement l'élément `a`. Exemples : `iterer 3 succ 10 = 13`, `iterer 0 succ 10 = 10` (en définissant `let succ = fun n -> n + 1`).
5. Définir la fonction `rot` prenant en argument un entier n et une liste ℓ, et renvoyant la liste obtenue après `(abs` n`)` rotations vers la gauche si $n < 0$, resp. n rotations vers la droite si $n \geqslant 0$.
6. Soit la signature `tCADRAN` ci-dessous :

```
module type tCADRAN = sig
  type 'a t
  val init : 'a list  -> 'a t (* précondition: liste non vide *)
  val courant : 'a t -> 'a
  val avancer : 'a t -> 'a t
  val est_a_zero : 'a t -> bool
end
```

Un cadran (de type `'a t`) contient une liste d'éléments de type `'a`, et un élément privilégié parmi ceux-ci, qui correspond à « l'état zéro » du cadran. L'élément de tête de la liste correspond à l'« état courant » du cadran (indiqué par une flèche dans la figure ci-dessous). Lorsque ces deux éléments sont égaux, le cadran est considéré « à zéro » (ce qui est donc le cas initialement). Le cadran peut tourner comme illustré par la figure ci-dessous :



Définir un module `CADRAN` conforme à cette spécification où le type `'a t` sera donc représenté par un couple `(liste, élément zéro)`, et où la fonction `init` prendra en argument une liste non vide d'éléments supposés tous différents (on le supposera sans le vérifier), pour construire un cadran constitué de cette même liste, et de son élément de tête correspondant à l'« élément zéro ». Faire avancer le cadran revient à appliquer à la liste une rotation vers la gauche : `[0;1;2;3;4;5;6;7;8;9]` $\rightsquigarrow$ `[1;2;3;4;5;6;7;8;9;0]`.

Exercice 2 : Un mini-langage d'expressions régulières (environ 11 points)

1. Fonctions auxiliaires sur les listes

1.a) Définir la fonction `length` prenant en argument une liste ℓ, et renvoyant le nombre d'éléments de cette liste ℓ. Par exemple :

```
length [false; false; true] = 3
```

1.b) Définir la fonction `map` prenant en argument une fonction `f` et une liste ℓ, et renvoyant la liste des résultats de l'application de `f` à chaque élément de ℓ. Par exemple :

```
map (fun x -> x+1) [1;2;3] = [2;3;4]
```

1.c) Définir la fonction `flat_map` prenant en argument une fonction `f` et une liste ℓ, et renvoyant la concaténation des résultats de l'application de `f` à chaque élément de ℓ. Par exemple :

```
flat_map (fun x -> if x < 0 then [] else [x;x]) [-1;2;3] = [2;2;3;3]
```

1.d) Définir la fonction `exists` prenant en argument un prédicat `p` et une liste ℓ et renvoyant `true` si et seulement si ℓ contient au moins un élément vérifiant `p`. Par exemple :

```
exists (fun x -> x>0) [-1;2;-3] = true
exists (fun x -> x>0) [-1;-2;-3] = false
```

1.e) Définir la fonction `split` prenant en argument une liste ℓ de n éléments et renvoyant la liste de taille $n+1$ contenant les couples (ℓ_1, ℓ_2) tels que ℓ = `l1 @ l2`. Par exemple :

```
split [1;2;3] = [ ([],[1;2;3]); ([1],[2;3]); ([1;2],[3]); ([1;2;3],[]) ]
```

2. Expressions régulières *(rappel : vous pouvez réutiliser les fonctions des questions précédentes)*

On définit une expression régulière par le type `'a re` ci-dessous, paramétré par le type des symboles.

```
type 'a re =
  | Symbole of 'a
  | Union of 'a re * 'a re
  | Produit of 'a re * 'a re
  | Repetition of 'a re
```

Une expression régulière reconnaît un ensemble de mots défini comme suit :

- `Symbole s` : reconnaît un seul mot réduit au symbole `s`.
- `Union(r1,r2)` : l'union des ensembles de mots reconnus par `r1` ou par `r2`
- `Produit(r1,r2)` : l'ensemble des concaténations `m1.m2` des mots `m1` reconnus par `r1` et `m2` reconnus par `r2`.
- `Repetition r` : l'ensemble des concaténations $m_1.\ \ldots\ .m_k$ de mots m_i reconnus par `r`.

On notera qu'un mot reconnu par une `'a re` n'est jamais vide.

2.a) Définir la fonction `accept` prenant en argument une liste de symboles (appelée mot) et une expression régulière `r` et renvoyant `true` si et seulement si le mot est reconnu par `r`. Par exemple :

```
accept ['a';'b';'b'] (Produit(Symbole 'a', Repetition (Symbole 'b'))) = true
```

On utilisera le fait que les mots reconnus par `Repetition r` sont exactement ceux reconnus par `Union(r, Produit(r, Repetition r))`.

2.b) Définir la fonction `mots` prenant en argument un entier `n` et une expression régulière `r`, et renvoyant la liste des mots de longueur $\leqslant$ `n` reconnus par `r`. La liste pourra contenir des mots dupliqués. Exemple :

```
mots 3 (Repetition (Union (Symbole 'a', Produit (Symbole 'a', Symbole 'b')))) =
[['a']; ['a'; 'b']; ['a'; 'a']; ['a'; 'a'; 'b']; ['a'; 'a'; 'a']; ['a'; 'b'; 'a']]
```

Remarques

- `Symbole s` ne reconnaît qu'un seul mot et il est de longueur 1
- `Union(r1,r2)` reconnaît les mots reconnus par `r1` ou par `r2`
- Pour chaque mot `m1` de longueur au plus `n` reconnu par `r1`, `Produit(r1,r2)` reconnaît `m1.m2` de longueur au plus `n` si `m2` est un mot de longueur au plus `n-longueur(m1)` reconnu par `r2`.
- Comme précédemment, on utilisera le fait que les mots reconnus par `Repetition r` sont exactement ceux reconnus par `Union(r, Produit(r, Repetition r))`.