Université Grenoble Alpes **Licence L1**
UE INF201 **Année 2017-18**

Algorithmique et Programmation Fonctionnelle
Projet facultatif : Rummikub

objectifs

Le but de ce projet est de programmer le *jeu du rummikub*, une variante du jeu de rami qui utilise des chiffres. Ce projet vous permettra dans un premier temps de vous exercer à manipuler des listes, une structure de donnée récursive importante en programmation fonctionnelle.

1 Le jeu de *rummikub*

Le jeu se joue avec des tuiles comportant une valeur et une couleur, et une tuile spéciale : un joker. Les valeurs possibles des tuiles sont des entiers de 1 à 13 associés à une couleur (bleu, rouge, jaune ou noir). Le jeu se joue avec 2 séries de telles tuiles.

Ainsi, dans le jeu, chaque *tuile* avec une valeur apparaît deux fois pour chaque couleur et il y a deux jokers. Cela fait donc 106 tuiles.

Les règles

Les règles du jeu de *rummikub* sont les suivantes [1] :

1. Chaque joueur dispose d'un certain nombre de tuiles devant lui qu'il est le seul à voir, et qui forment sa *main*. Au centre de la *table* sont disposées les tuiles déjà en jeu par paquets qui forment nécessairement des combinaisons valides. Les tuiles du paquet qui ne sont ni en jeu dans des combinaisons, ni dans les mains des joueurs, constituent la *pioche*.
2. Au début du jeu, il n'y a pas de tuile en jeu. La distribution initiale, 14 tuiles par joueur est composées de tuiles tirées au hasard parmi les tuiles de la pioche. Le score initial de chaque joueur est 0.
3. A son tour, un joueur peut piocher (tirer au hasard une tuile de la pioche et la mettre dans sa main), et terminer son tour ; ou placer un nombre non nul des tuiles de sa main dans le jeu et réorganiser toutes les tuiles en jeu en combinaisons valides. Il est donc interdit de réorganiser les tuiles en jeu si on n'en pose pas. Il n'y a pas de restriction dans la réorganisation des tuiles en jeu à condition que celles-ci soient regroupées en combinaisons valides (cf. exemples paragraphes 6 et 8.4).
4. Les combinaisons valides sont :
   — Les *suites* monochromes d'au moins 3 tuiles consécutives, par exemple, les tuiles de valeur 2, 3 et 4, de couleur rouge.
   — Les *groupes* de trois ou quatre tuiles de couleurs distinctes et comportant le même entier, par exemple trois tuiles de valeur 12 de couleur rouge, jaune et bleu.
5. Lors de sa première pose, un joueur doit poser une ou plusieurs combinaisons valides dont la valeur totale (somme des entiers inscrits sur les tuiles) est au moins 30. Le joker compte pour l'entier qu'il remplace. Il est interdit de modifier de quelque façon les autres combinaisons déjà en place lors de cette première pose.

1. dans la suite nous considèrerons le cas où il y a uniquement deux joueurs

6. Le jeu s'arrête dès qu'un joueur a vidé sa main, ou si aucun des deux joueurs ne peut plus jouer : ils ne peuvent plus piocher car la pioche est vide et ils ne peuvent plus poser de tuiles.
7. On ne marque pas de point au cours de la partie ; mais le score d'un joueur est égal au total des valeurs des tuiles présentes dans sa main ou moment où le jeu se termine. Pour ce calcul, un joker encore présent dans une main compte 30 points. L'objectif est évidemment d'avoir le score le plus faible, soit sur une partie (et donc être le joueur qui termine), soit sur le score cumulé de plusieurs parties successives.

2 Ensemble avec répétitions

Nous définissons des multi-ensembles qui sont des ensembles avec répétitions. par exemple, `me = {3,4,6,3,6,6}` est un multi-ensemble d'entiers contenant deux exemplaires de `3`, un exemplaire de `4` et trois exemplaires de `6`.

On choisit de représenter un multi-ensemble d'éléments (de type quelconque) comme une liste **non ordonnée et de taille minimale** de couples `(x, n)`, où `x` est un élément du multi-ensemble et `n` son nombre **total** d'occurrence dans ce multi-ensemble.

En OCaml, on utilisera le type suivant pour implémenter les couple `(x, n)` :

```
type 'e melt = 'e * int
```

Un multi-ensemble est donc une liste de *multi-éléments* `'e melt`, que l'on pourra implémenter :

— soit en utilisant le type récursif suivant : `type 'e mset = Nil | Cons of 'e melt * 'e mset`.

— soit en utilisant le type prédéfini `list` : `type 'e mset = 'e melt list`.

Voici deux des `3!` implémentations possibles du multi-ensemble `me`, obtenues en utilisant la première solution :

— `let me = Cons((3,2), Cons((4,1), Cons((6,3), Nil)))`,

— `let me = Cons((6,3), Cons((3,2), Cons((4,1), Nil)))`.

En utilisant la seconde solution ces définitions s'écrivent respectivement :

— `let me = [(3,2) ; (4,1) ; (6,3)]`,

— `let me = [(6,3) ; (3,2) ; (4,1)]`.

Notons que la liste `[(6,1) ; (3, 2) ; (6,2) ; (4,1)]` n'est pas une implémentation valide : cette liste n'est pas minimal car l'élément `6` y apparait deux fois.

Définir les fonctions suivantes :

1. `isempty : 'e mset -> bool`, où `isempty s` est vrai si `s` est un muti-ensemble vide.
2. `cardinal : 'e mset -> int`, donne le cardinal d'un multi-ensemble c'est-à-dire le nombre total d'occurences d'éléments. par exemple : `cardinal me = 6`.
3. `nb_occurrences : 'e -> 'e mset -> int`, donne le nombre d'occurences d'un élément dans un muti-ensemble.
4. `member : 'e -> 'e mset -> bool`, où `(member e s)` est vrai si `e` $\in$ `s`.
5. `subset : 'e mset -> 'e mset -> bool`, où `(subset s1 s2)` est vrai si et seulement si `s1` $\subseteq$ `s2`.
6. `add : 'e melt -> 'e mset -> 'e mset`, ajoute un certain nombre d'occurences d'un élément dans le muti-ensemble.
7. `remove : 'e melt -> 'e mset -> 'e mset`, où `(remove (e, n) s)` supprime `n` occurrences de `e` dans `s`. si `n` vaut 0 alors la fonction est sans effet et si `n` est strictement supérieur au nombre d'occurences de `e` dans `s` alors toutes les occurences de `e` sont supprimées.
8. `equal : 'e mset -> 'e mset -> bool`, où `equal s1 s2 = vrai` si et seulement si `s1` et `s2` contiennent les mêmes éléments et en même nombre.

9. `sum : 'e mset -> 'e mset -> 'e mset`, où `sum s1 s2 = s1` $\uplus$ `s2` contient chaque élément en autant d'exemplaires qu'il y en a au total dans `s1` et dans `s2`.
10. `intersection : 'e mset -> 'e mset -> 'e mset`, où `intersection s1 s2 = s1` $\cap$ `s2`.
11. `difference : 'e mset -> 'e mset -> 'e mset`, où `difference s1 s2 = s1` $\setminus$ `s2`.
12. `getrandom : 'e mset -> 'e`, qui rend un élément pris au hasard dans le multi-ensemble (en tenant compte des répétitions : un élément présent 3 fois aura une probabilité d'être choisi 3 fois supérieure à celle d'un élément présent 1 seule fois). On pourra utiliser la fonction `Random.int k` qui donne un entier compris entre `0` et `k-1`.

    On pourra définir une fonction `get : int -> 'e mset -> 'e`, où `(get n s)` donne le n^{eme} élément de `s`. La fonction `get` a pour précondition que le multi-ensemble n'a pas plus de n éléments, mais ne fait aucune hypothèse sur quelque ordre que ce soit du multi-ensemble, il s'agit juste du parcours d'une représentation possible du multi-ensemble.

3 Utilisation de l'ordre supérieur

— Spécifier et réaliser une fonction d'ordre supérieur `fold_mset` qui applique récursivement une fonction `f` à deux arguments sur un élément d'initialisation `a` et un multi-ensemble `s`, en parcourant les multi-éléments de `s` du premier au dernier. Autrement dit, `fold_mset f a s` retourne `f` ... (`f` (`f a` s_0) s_1) ... s_{n-1}, où s_0, s_1, ... désignent les différents multi-éléments de `s`.
Remarque : si vous avez choisi de représenter les multi-ensembles en utilisant le type `list` la fonction `fold_mset` pourra être définie à l'aide de la fonction `List.fold_left`.

— En utilisant la fonction d'ordre supérieur `fold_mset`, réécrivez les fonctions suivantes de la partie précédente :

1. `cardinal` ;
2. `subset` ;
3. `sum` ;
4. `intersection`.

4 Représentation des données

Les tuiles sont représentées par le type produit `tuile`, une tuile étant une carte caractérisée par un entier et une couleur ou un joker :

```
type couleur = Bleu | Rouge | Jaune | Noir ;;
type tuile = T of (int * couleur) | Joker ;;
```

Une `combinaison` est une liste de tuiles. L'ordre dans la liste est significatif dans le cas d'une suite.

```
type combinaison = tuile list ;;
```

Les combinaisons déjà posées constituent la `table`. L'ordre dans la liste n'est pas significatif. une `pose`, qui sera utilisée pour le premier coup des joueurs, est également une liste de combinaisons.

```
type table = combinaison list ;;
type pose = combinaison list ;;
```

Le type d'une `main`, c'est-à-dire l'ensemble des tuiles que possède un joueur, et de la `pioche`, sont un muti-ensemble de tuiles.

```
type main = tuile mset ;;
type pioche = tuile mset ;;
```

Les 106 tuiles du jeu sont représentées par un multi-ensemble nommé `paquet_initial`, défini ci-dessous (en supposant que le type `'e mset` est représenté par un type récursif avec constructeurs `Cons` et `Nil`) :

```
let (paquet_initial : pioche) =
Cons (T(1,Rouge), 2) (Cons (T(2,Rouge), 2) (Cons (T(3,Rouge), 2) (Cons (T(4,Rouge), 2)
(Cons (T(5,Rouge), 2) (Cons (T(6,Rouge), 2) (Cons (T(7,Rouge), 2) (Cons (T(8,Rouge), 2)
(Cons (T(9,Rouge), 2) (Cons (T(10,Rouge), 2) (Cons (T(11,Rouge), 2) (Cons (T(12,Rouge), 2)
(Cons (T(13,Rouge), 2)
(Cons (T(1,Bleu), 2) (Cons (T(2,Bleu), 2) (Cons (T(3,Bleu), 2) (Cons (T(4,Bleu), 2)
(Cons (T(5,Bleu), 2) (Cons (T(6,Bleu), 2) (Cons (T(7,Bleu), 2) (Cons (T(8,Bleu), 2)
(Cons (T(9,Bleu), 2) (Cons (T(10,Bleu), 2) (Cons (T(11,Bleu), 2) (Cons (T(12,Bleu), 2)
(Cons (T(13,Bleu), 2)
(Cons (T(1,Jaune), 2) (Cons (T(2,Jaune), 2) (Cons (T(3,Jaune), 2) (Cons (T(4,Jaune), 2)
(Cons (T(5,Jaune), 2) (Cons (T(6,Jaune), 2) (Cons (T(7,Jaune), 2) (Cons (T(8,Jaune), 2)
(Cons (T(9,Jaune), 2) (Cons (T(10,Jaune), 2) (Cons (T(11,Jaune), 2) (Cons (T(12,Jaune), 2)
(Cons (T(13,Jaune), 2)
(Cons (T(1,Noir), 2) (Cons (T(2,Noir), 2) (Cons (T(3,Noir), 2) (Cons (T(4,Noir), 2)
(Cons (T(5,Noir), 2) (Cons (T(6,Noir), 2) (Cons (T(7,Noir), 2) (Cons (T(8,Noir), 2)
(Cons (T(9,Noir), 2) (Cons (T(10,Noir), 2) (Cons (T(11,Noir), 2) (Cons (T(12,Noir), 2)
(Cons (T(13,Noir), 2) (Cons (Joker, 2) Nil)
)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))) ;;
```

5 Mise en oeuvre des règles

Les parties 5.1 et 5.2 peuvent être traitées dans n'importe quel ordre.

5.1 Validité de combinaisons

1. Le prédicat `suite_valide : combinaison -> bool`, détermine si la combinaison qui lui est passée en paramètre est une suite c'est-à-dire une succession d'au moins trois tuiles de même couleur et de valeurs consécutives.
2. Le prédicat `groupe_valide : combinaison -> bool`, détermine si la combinaison qui lui est passée en paramètre est un groupe c'est-à-dire une succession d'au moins trois tuiles de même valeur et de couleurs différentes.
3. Le prédicat `combinaison_valide : combinaison -> bool`, détermine si une combinaison est valide.
4. Le prédicat `proposition_valide : table -> bool`, détermine si la liste de combinaisons est constituée de combinaisons valides. Pour ce dernier vous pourrez utiliser la fonction d'ordre supérieur `fold`.

5.2 Calcul de points

1. La fonction `points_suite : combinaison -> int`, prend en paramètre une suite valide (précondition) et retourne le nombre de points de celle-ci.
2. La fonction `points_groupe : combinaison -> int`, prend en paramètre un groupe valide (précondition) et retourne le nombre de points de celui-ci.
3. La fonction `lespoints : pose -> int`, calcule le nombre de points associés à une pose, c'est-à-dire une liste de combinaisons.

6 Etat d'une partie

L'état d'une partie est constitué des informations suivantes :
— Les combinaisons en jeu,
— La pioche restante,
— Quels sont les joueurs qui ont déjà posé des tuiles
— A quel joueur est-ce le tour de piocher ou poser
Les joueurs sont identifiés par une constante. Nous considérons dans la suite **uniquement 2 joueurs**.

```
type joueur = j1 | j2 ;;
```

Le statut d'un joueur est constitué de son nom, d'un booléen qui est vrai si le joueur a déjà posé des tuiles sur la table, autrement dit si ce n'est pas son premier coup, et de sa main. Le statut des deux joueurs est défini par un couple dont le premier élément est le statut de `j1` et le deuxième le statut de `j2`.

```
type statutjoueur = (joueur * bool * main) ;;
type les_statuts = statutjoueur * statutjoueur ;;
```

L'état d'une partie est alors décrit par le statut des deux joueurs, l'état de la table courante, la pioche courante et le nom du joueur dont c'est le tour de jouer.

```
type etat = les_statuts * table * pioche * joueur ;;
```

Par exemple, lors du premier tour, `j1` a posé les deux combinaisons :
— (7, rouge), (7,Jaune), (7,jaune),
— (1, noir), (2, noir), (3, noir), (4, noir),
et `j2` a posé la combinaison :
— (9, rouge), (10, rouge), (11, rouge).
L'état du jeu est alors :
— joueurs :
  — j1, true, les tuiles restantes dans la main de j1
  — j2, true, les tuiles restantes dans la main de j2
— table :
  — (7, rouge), (7,Jaune), (7,jaune)
  — (1, noir), (2, noir), (3, noir), (4, noir)
  — (9, rouge), (10, rouge), (11, rouge)
— pioche : les tuiles de la pioche
— à qui le tour : j1

6.1 Fonctions d'accès

Il peut être utile de définir des fonctions d'accès aux différents composants de l'état, comme par exemple les fonctions définies ci-dessous :
— `let joueur_courant : etat -> joueur`
— `let la_table : etat -> table`
— `let la_pioche : etat -> pioche`
— `let joueur_suivant : etat -> joueur`
— `let le_statut : joueur -> etat -> statutjoueur`
— `let la_main : joueur -> etat -> main`

6.2 Affichage de la table

Pour l'affichage de la table et des mains des joueurs, il peut être utile d'en avoir une représentation ordonnée. La fonction `en_ordre : tuile mset -> tuile mset` donne la représentation ordonnée du multi-ensemble de tuiles donné en paramètre. Pour trier ces tuiles, on choisit l'**ordre lexicographique** obtenu en considérant l'ordre `bleu < rouge < jaune < noir` sur les couleurs, puis, dans chaque couleur, l'ordre croissant de valeurs. Les jokers seront placés en fin de liste.

7 Initialisation

Au début de la partie chaque joueur reçoit 14 tuiles tirées au hasard dans le paquet initial.

1. La fonction `extraire : int -> pioche -> main * pioche`, est définie de la façon suivante : l'appel `(extraire n p)` rend un couple `(m, newp)` dans lequel `m` est une main de `n` tuiles tirées au hasard dans `p`, et `newp` est la pioche `p` privée de ces `n` tuiles. On pourra utiliser la fonction `getrandom` de la section 2 pour implémenter cette fonction.
2. La fonction `distribuer : pioche -> main * main * pioche`, prend en paramètre la pioche initiale et rend les mains de chacun des deux joueurs et la pioche restante.
3. La fonction `initialiser : pioche -> etat`, produit à partir du paquet initial l'état initial de la partie après distribution d'une main à chaque joueur.

8 Et maintenant jouons

Il peut être utile de disposer d'une fonction de conversion d'une table en multi-ensemble de tuiles, pour pouvoir utiliser les fonctions définies dans la partie 2 : `setof_tuiles : table -> tuile mset`. Ici encore, pensez à l'ordre supérieur.

8.1 Proposition de jeu pour un tour

1. Le prédicat `premier_coup_valide : main -> pose -> main -> bool`, détermine si un coup est valide pour un premier tour. `premier_coup_valide m0 p0 m1` rend vrai si et seulement si la main du joueur étant `m0`, la proposition de combinaisons à poser `p0` et la nouvelle main `m1` constituent un coup valide pour un premier tour.
2. Le prédicat `coup_valide : table -> main -> table -> main -> bool`, détermine si un coup est valide. `coup_valide t0 m0 t1 m1` est vrai si et seulement si avec la table `t0` et la main `m0`, il est possible de produire la table `t1` et la main `m1`.

8.2 Ajouter une tuile sur la table

La fonction `ajouter_tuile : table -> tuile -> table`, produit la table obtenue en ajoutant la tuile donnée à l'une des combinaisons de la table courante si cela est possible, et la liste vide sinon.

La table donnée est correcte c'est-à-dire formée de combinaisons valides.

8.3 Trouver des suites et groupes dans une main

1. La fonction `proposer_suite : main -> combinaison`, extrait une suite d'une main.
2. La fonction `proposer_groupe : main -> combinaison`, extrait un groupe d'une main

Indications : Une solution consiste à extraire au hasard un certain nombre de tuiles de la main et de regarder si elles forment une suite (respectivement un groupe). Si ce n'est pas le cas essayer à nouveau. le nombre d'essais reste à fixer.

Une autre solution consiste à regarder toutes les combinaisons possibles que l'on peut former avec les cartes de la main donnée. Peut-on évaluer ce nombre de combinaisons ?

8.4 Passer d'un état au suivant

La fonction `piocher : etat -> etat` produit le nouvel état après que le joueur courant ait pioché une carte. Le tour passe au joueur suivant. si la pioche est vide la fonction rend l'état passé en paramètre.

La fonction `jouer_un_coup : etat -> table -> etat` produit un nouvel état à partir d'un état donné et d'une proposition de table. On exclut le premier coup : si c'est le premier coup du joueur courant l'état rendu est l'état donné en paramètre.

Par exemple, si l'état courant `e_cour` contient les éléments suivants :

- joueurs :
  - j1, true, { ((7, noir), 2), ((8, rouge), 1), ((3, noir), 1), ((5, noir), 2) }
  - j2, true, les tuiles restantes dans la main de j2
- jeu :
  - (7, rouge), (7,bleu), (7,jaune)
  - (1, noir), (2, noir), (3, noir), (4, noir)
  - (9, rouge), (10, rouge), (11, rouge)
- pioche : les tuiles de la pioche
- à qui le tour : j1

Si on considère la table `t_proposee` suivante :

- (7, rouge), (7,bleu), (7,jaune) (7, noir)
- (1, noir), (2, noir), (3, noir)
- (3, noir), (4, noir), (5, noir)
- (8, rouge), (9, rouge), (10, rouge), (11, rouge)

L'appel `jouer_un_coup e_cour t_proposee` est correct et produit l'état suivant :

- joueurs :
  - j1, true, { ((7, noir), 1), ((5, noir), 1) }
  - j2, true, les tuiles restantes dans la main de j2
- jeu :
  - (7, rouge), (7,bleu), (7,jaune) (7, noir)
  - (1, noir), (2, noir), (3, noir)
  - (3, noir), (4, noir), (5, noir)
  - (8, rouge), (9, rouge), (10, rouge), (11, rouge)
- pioche : les tuiles de la pioche
- à qui le tour : j2

La fonction `jouer_premier_coup : etat -> pose -> etat` produit un nouvel état à partir d'un état donné et d'une proposition pour le premier coup du joueur courant. Si ce n'est pas le premier coup du joueur courant la fonction rend l'état donné en paramètre ; de même si la proposition n'est pas correcte (par exemple, si la somme des points est inférieure ou égale à 30).

Vous pouvez maintenant jouer des coups successifs en utilisant les fonctions précédemment définies.

Par exemple, un scénario peut être :

— début de partie : `let etatinit = initialiser paquet_initial ;;`
— j1 joue son premier coup
  — `let prop = [[T(2,Jaune); T(3,Jaune); T(4,Jaune)];`
    `[T(7,Jaune); T(8,Jaune); Joker]];;`
  — `let e1 = jouer_premier_coup etatinit prop;;`
— j2 pioche `let e2 = piocher e1;;`
— j1 pioche `let e3 = piocher e2;;`
— j2 pioche `let e4 = piocher e3;;`
— j1 pioche `let e5 = piocher e4;;`
— j2 pioche `let e6 = piocher e5;;`
— j1 ajoute des cartes a la table
  — `let nt = [[T(2,Jaune); T(3,Jaune); T(4,Jaune)];`
    `[T(7,Jaune); T(8,Jaune); Joker; T(10,Jaune)]];;`
  — `let e7 = jouer_un_coup e6 nt;;`
— j2 joue son premier coup
  — `let prop = [[Joker; T (11, Rouge); T (11, Bleu)]];;`
  — `let e8 = jouer_premier_coup e7 prop;;`
— j1 pioche `let e9 = piocher e8;;`
— j2 ajoute des cartes a la table
  — `let nt = [[T (2, Jaune); T (3, Jaune); T (4, Jaune); T (5, Jaune)];`
    `[T (7, Jaune); T (8, Jaune); Joker; T (10, Jaune)];`
    `[Joker; T (11, Rouge); T (11, Bleu)]];;`
  — `let e10 = jouer_un_coup e9 nt;;`
— etc...

A vous d'exhiber un scénario en utilisant vos propres fonctions.

9 Pour aller plus loin

Un plus est bien sûr de mettre en oeuvre le moteur qui enchaine les coups jusqu'à la fin de la partie. on peut envisager un moteur qui permet de faire joueur successivement un joueur puis l'autre en rentrant les tables successives au clavier. On peut aussi envisager de faire jouer un joueur contre l'ordinateur.

Jouez bien